《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	17 计教学 2 班
学 生 姓 名 ：	刘斯宇
学　　号 ：	17341110
时　　间 ：	2018　年　11　月　10　日

成 绩 ：

实验二 ： 单周期CPU设计与实现

一. 实验目的

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

二、 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

算术运算指令

（1）add rd , rs, rt

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

功能：rd←rs + rt。**reserved 为预留部分，即未用，一般填“0”。**

（2）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt。

（3）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate;**
immediate 符号扩展再参加“加”运算。

逻辑运算指令

（4）andi rt , rs ,**immediate**

010000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs & (zero-extend)**immediate;**
immediate 做“0”扩展再参加“与”运算。

（5）and　rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（6）ori　rt , rs ,**immediate**

010010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate**；**immediate** 做“0”扩展再参加“或”运算。

（7）or　rd , rs , rt

010011	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（8）sll　rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa(5 位)	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa。

==>比较指令

（9） slti　rt, rs,**immediate**　带符号数

011100	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

存储器读/写指令

（10）sw　rt ,**immediate**(rs) 写存储器

100110	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：memory[rs+ (sign-extend)**immediate**]←rt；immediate 符号扩展再相加。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（11) lw　rt , **immediate**(rs) 读存储器

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

功能：rt ← memory[rs + (sign-extend)**immediate**]；immediate 符号扩展再相加。

即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

分支指令

（12）beq　rs,rt,**immediate**

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

功能：if(rs=rt) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。**immediate 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 immediate 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（13）bne　rs,rt,**immediate**

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

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2　else pc ←pc + 4

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

（14）bltz　rs,**immediate**

110010	rs(5 位)	00000	**immediate**(16 位)

功能：if(rs<$zero) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4。

跳转指令

（15）j　addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，无条件跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

停机指令

（16）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三、实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（**如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。**）

CPU 在处理指令时，一般需要经过以下几个步骤：

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

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完

成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

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

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

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

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

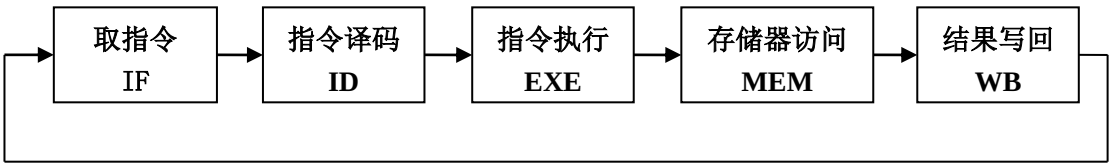
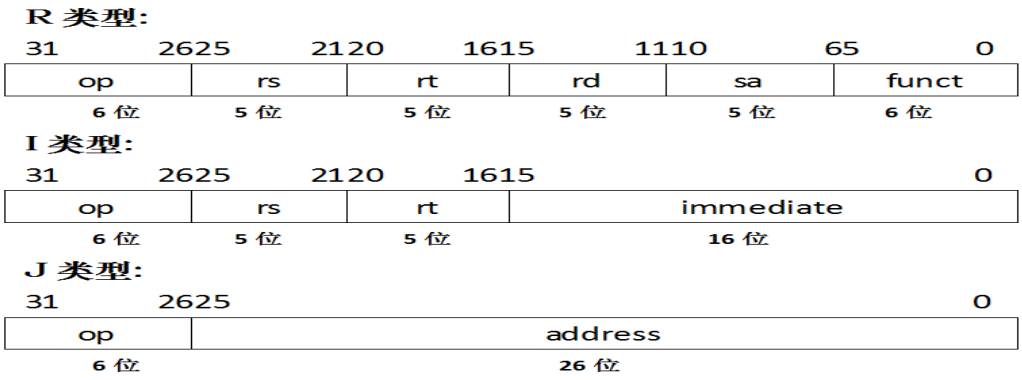


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：



其中，

op：为操作码；

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

rt：可读可写。为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：只写。为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能与操作码配合使用；

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

address：为地址。

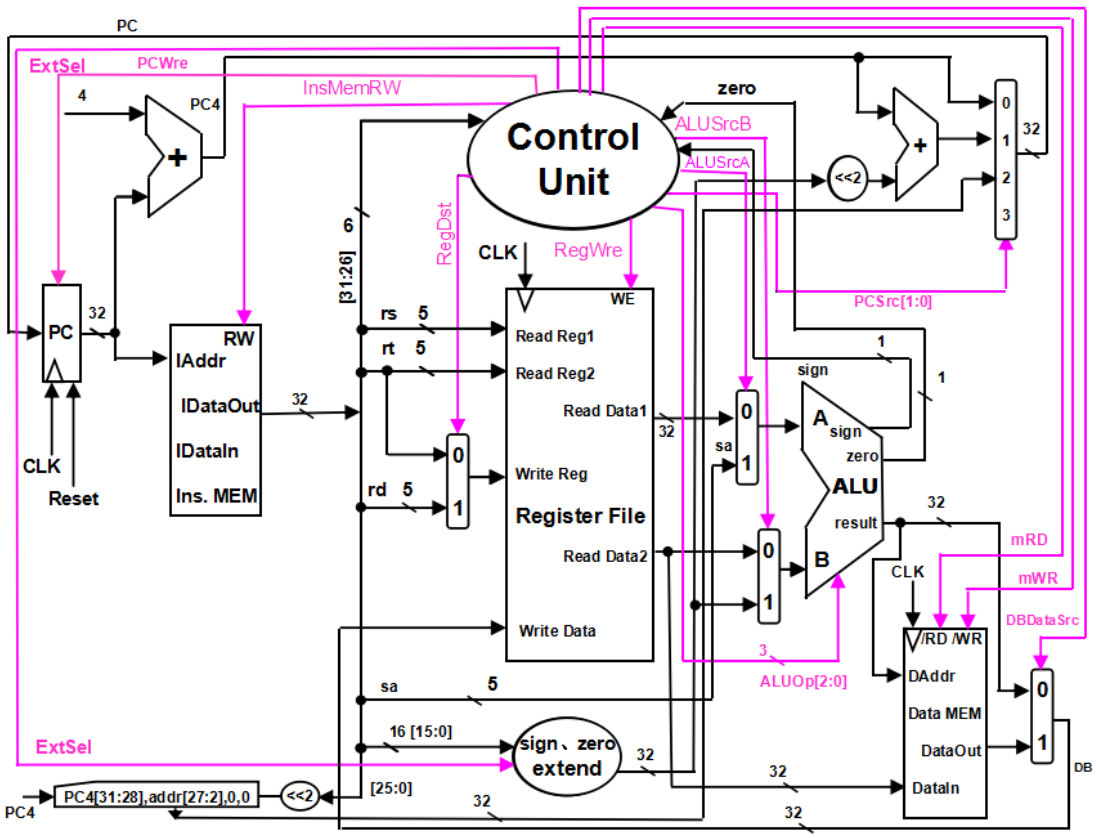


图 2　单周期 CPU 数据通路和控制线路图

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

表 1　控制信号的作用

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、	来自数据存储器（Data MEM）的输出，相关指令：lw

	and、andi、slti、sll	
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)**immediate(0 扩展)**，相关指令：andi、ori	(sign-extend)**immediate (符号扩展)**，相关指令：addiu、slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)； 01：pc<－pc+4+(sign-extend)**immediate<<2**，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1)； 10：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j； 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：指令存储器，

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器，

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

/RD，数据存储器读控制信号，为 0 读

/WR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU： 算术逻辑单元

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

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

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A<B 不带符号
110	Y=(((A<B)&&(A[31] == B[31])) ||((A[31]==1&& B[31] == 0))) ? 1:0	比较 A<B 带符号
111	Y = A ⊕ B	异或

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化的思想方法设计，关于 ALU 设计、存储器设计、寄存器组设计等等，也是必须认真考虑的问题。

四．实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五．实验过程与结果

1、 测试程序段

地址	汇编程序	指令代码					
		op（6）	**rs(5)**	**rt(5)**	**rd(5)/immediate (16)**	**16 进制数代码**	
0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010		48020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000		00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	0010 1000 0000 0000		04622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 1000 0000 0000		44a22000
0x00000014	or $8,$4,$2	010011	00100	00010	0100 0000 0000 0000		4c824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000		60084040
0x0000001C	**bne $8,$1,-2 (≠,转 18)**	110001	01000	00001	1111 1111 1111 1110		C501FFFE
0x00000020	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100		70460004
0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000		70C70000
0x00000028	addiu $7,$7,8	000010	00111	00111	0000 0000 0000 1000		08E70008
0x0000002C	**beq $7,$1,-2 (=,转 28)**	110000	00111	00001	1111 1111 1111 1110		C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100		98220004
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100		9C290004
0x00000038	addiu $10,$0,-2	000010	00000	01010	1111 1111 1111 1110		080AFFFE
0x0000003C	addiu $10,$10,1	000010	01010	01010	0000 0000 0000 0001		094A0001
0x00000040	**bltz $10,-2(<0,转 3C)**	110010	01010	00000	1111 1111 1111 1110		C940FFFE
0x00000044	**andi $11,$2,2**	000010	00010	01011	0000 0000 0000 0010		084B0002
0x00000048	**j 0x00000050**	111000	00000	00000	0000 0000 0001 0100		E0000014
0x0000004C	or $8,$4,$2	010011	00100	00010	0100 0000 0000 0000		
0x00000050	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000

2、 控制信号与指令关系

Op		PCWre	ALUSrcA	ALUSrcB	RegWre	RegDst	IsMemRW	ExtSel	ALUOp	PCSrc	/RD	/WR	DBData
000000	**add**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**000**	**00**	**1**	**1**	**0**
000010	**addiu**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**000**	**00**	**1**	**1**	**0**
000001	**sub**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**001**	**00**	**1**	**1**	**0**
010010	**ori**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**011**	**00**	**1**	**1**	**0**
010001	**and**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**100**	**00**	**1**	**1**	**0**
000010	**andi**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**100**	**00**	**1**	**1**	**0**
010011	**Or**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**011**	**00**	**1**	**1**	**0**
011000	**Sll**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**010**	**00**	**1**	**1**	**0**
011100	**Slti**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**110**	**00**	**1**	**1**	**0**
100110	**Sw**	**1**	**0**	**1**	**0**	**x**	**1**	**1**	**000**	**00**	**1**	**0**	**x**
100111	**Lw**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**000**	**00**	**0**	**1**	**1**
110000	**Beq**	**1**	**0**	**0**	**0**	**x**	**1**	**1**	**001**	**00/01**	**1**	**1**	**x**
110001	**Bne**	**1**	**0**	**0**	**0**	**x**	**1**	**1**	**001**	**00/01**	**1**	**1**	**x**
110010	**bltz**	**1**	**0**	**0**	**0**	**x**	**1**	**1**	**110**	**00/01**	**1**	**1**	**x**
111000	**J**	**1**	**x**	**x**	**0**	**x**	**1**	**x**	**x**	**10**	**1**	**1**	**x**
111111	**Halt**	**0**	**x**	**x**	**0**	**x**	**1**	**x**	**x**	**x**	**1**	**1**	**x**

几大模块

(1)ALU模板

ALU.v *

```
`timescale 1ns / 1ps
module ALU(a, b, ALUOp, zero,sign, result);
    input [31:0] a, b;
    input [2:0] ALUOp;
    output zero;
    output sign;
    output reg [31:0] result;
   initial begin
        result = 0;
   end
    assign zero=(result!=0?0:1);
    assign sign=(result>0)?1:0;
    always @(a or b or ALUOp) begin
        case(ALUOp)
          3'b000:result=a+b;                    //加法
          3'b001:result=a-b;                    //减法
          3'b010:result=a<<b;                   //移位
          3'b011:result=a|b;                    //或
          3'b100:result=a&b;                    //与
          3'b101:result=(a<b)?1:0;              //不带符号的比较
          3'b110: begin                         //带符号的比较，补码的比较
                  if(a<b && a[31]==b[31])
                      result=1;
                  else if(a[31]==1&&b[31]==0)
                       result=1;
                  else result=1;
                  end
           3'b111:result=(a^b);                 //异或
           default:result=0;
        endcase
       end
```

通过case语句不同的ALUOp来选择不同的算术操作，详细的对应关系见表2 ALU运算功能表

（2）RAM

RAM.v

C:/Users/82470/Desktop/资料/计组全部实验报告/Single_CPU/Single_CPU.srcs/sources_1/new/RAM.v

```
`timescale 1ns /1ns
    module RAM(
        input clk,
        input [31:0] address,
        input [31:0] writeData, // [31:24], [23:16], [15:8], [7:0]
        input mRD, // 为0，正常读
        input mWR, // 为0，写
        output [31:0] Dataout
        );
    reg [7:0] ram [0:60]; // 存储器定义必须用reg类型
    // 读
        assign Dataout[7:0] = (mRD==0)?ram[address + 3]:8'bz;
        assign Dataout[15:8] = (mRD==0)?ram[address + 2]:8'bz;
        assign Dataout[23:16] = (mRD==0)?ram[address + 1]:8'bz;
        assign Dataout[31:24] = (mRD==0)?ram[address ]:8'bz;
    // 写
        always@( negedge clk ) begin // 用电平信号触发写存储器
        if( mWR==0 ) begin
        ram[address] = writeData[31:24];
        ram[address+1] = writeData[23:16];
        ram[address+2] = writeData[15:8];
        ram[address+3] = writeData[7:0];
        end
        end
    endmodule
```

RAM模块如上图所示。分为读和写两种操作，mRD，mWR都是低电平有效，分别对应读和写。ram表示存储器。

（3）ROM

Untitled 1 | dataSelect_5_Bit.v | dataSelect_32_Bit.v | pc.v | ROM.v

C:/Users/82470/Desktop/资料/计组全部实验报告/Single_CPU/Single_CPU.srcs/sources_1/new/ROM.v

```
`timescale 1ns/10ps
module ROM (idataout, addr, InsMemRW);
    input InsMemRW;//使能信号
    input [31:0] addr; //2^5条指令地址 32条指令读取
    output reg [31:0] idataout;
    reg [7:0] mem [0:127]; //最大读取32条指令
    initial begin
        $readmemb("C:/Users/82470/Desktop/INS.txt", mem);
          idataout = 0;
     end
    always @( addr or InsMemRW)
        if (InsMemRW) begin
          idataout[31:24] = mem[addr];
          idataout[23:16] = mem[addr+1];
          idataout[15:8] = mem[addr+2];
          idataout[7:0] = mem[addr+3];
        end
endmodule
```

（4）PC

pc.v

C:/Users/82470/Desktop/资料/计组全部实验报告/Single_CPU/Single_CPU.srcs/sources_1/new/pc.v

```
`timescale 1ns / 1ps
module pc(clk, i_pc, pcWire, reset, o_pc);
 input wire clk, pcWire, reset;
 input wire [31:0] i_pc;
 output reg [31:0] o_pc;
 always @(posedge clk) begin
  if (reset) begin//初始
    o_pc = 0;
  end else if (pcWire) begin
    o_pc = i_pc;
  end else if (!pcWire) begin //停机
     o_pc = o_pc;
  end
 end
endmodule
```

下面是pc+4的模块

Untitled 1 | pcAddFour.v

:/Users/82470/Desktop/资料/计组全部实验报告/Single_CPU/Single_CPU.srcs/sources_1/new/pcAddFour.v

```
`timescale 1ns / 1ps
module pcAddFour(i_pc, o_pc);
  input wire [31:0] i_pc; //2^5条指令地址 32条指令读取
  output wire [31:0] o_pc;
  assign o_pc = i_pc + 4;
endmodule
```

下面是branch指令相关的pc模块

Untitled 1 | pcAddFour.v | pcAddImm.v

:/Users/82470/Desktop/资料/计组全部实验报告/Single_CPU/Single_CPU.srcs/sources_1/new/pcAddImm.v

```
`timescale 1ns / 1ps
module pcAddImm(now_pc, addNum, o_pc);
  input [31:0] now_pc, addNum;
  output [31:0] o_pc;
  assign o_pc = now_pc + (addNum * 4); //左移两位
endmodule
```

下面是更新pc的选择器，

dataSelect_321.v

C:/Users/82470/Desktop/资料/计组全部实验报告/Single_CPU/Single_CPU.srcs/sources_1/new/dataSelect_321.v

```
`timescale 1ns / 1ps
module dataSelect_321(PCSrc,A,B,C,out);
input [1:0]PCSrc;
input [31:0]A;
input [31:0]B;
input [31:0]C;
output reg [31:0]out;
always@(*) begin
      case(PCSrc)
      'b00:
      begin
         out = A;
      end
      'b01:
      begin
         out = B;
      end
      'b10:
      begin
         out = C;
      end
      endcase
   end
endmodule
```

（5）Signed-Extend

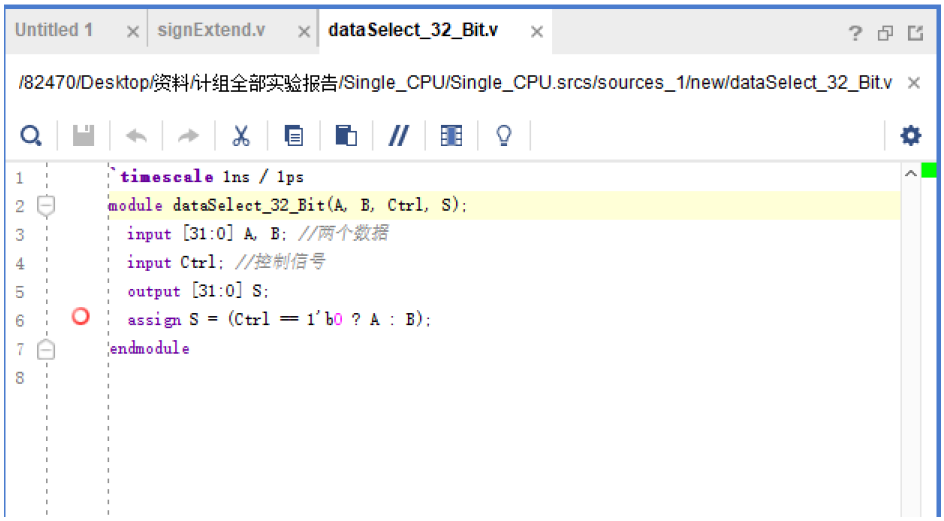
Untitled 1 | signExtend.v

:/Users/82470/Desktop/资料/计组全部实验报告/Single_CPU/Single_CPU.srcs/sources_1/new/signExtend.v

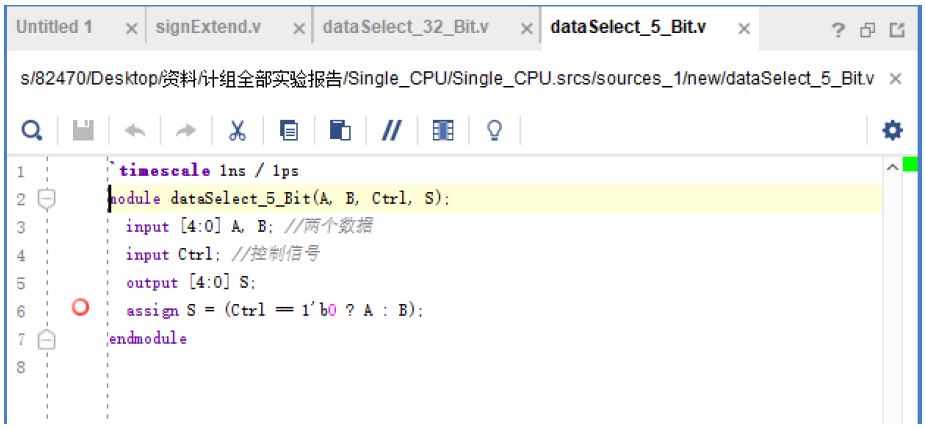
```
`timescale 1ns / 1ps
module signExtend(i_num, extSel, o_num);
  input wire[15:0] i_num;
  input wire extSel;
  output reg[31:0] o_num;
  initial begin
    o_num = 0;
  end
  always @(i_num or extSel) begin
    if (extSel) begin
      o_num <= {{16{i_num[15]}}, i_num[15:0]};
    end
    else begin
        o_num[15:0]<=i_num[15:0];
    end
  end
endmodule
```

上面是符号扩展和非富豪扩展单元。当需要符号扩展的时候，我们需要将最高位扩展，不需要符号扩展的时候，只需要把o_num的低16位赋给输出就好（因为输出初始化为0）

（6）Data_Select

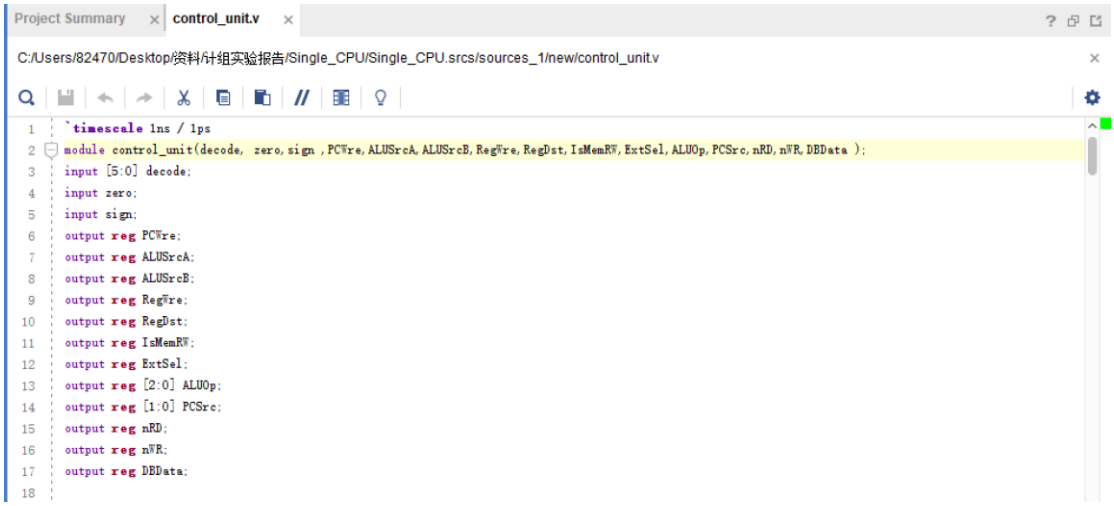


根据Ctrl的值来选择不同的值输出。



上图是为了选择sa而专门设计的五位选择器。

（7）control_unit



具体的各个信号见上面的控制信号与指令的表格。

下面来详细的说明一下波形图中的变量名称:

nowIns:当前的指令

nowpc:当前的pc

nexAddr：下一条指令的pc

aluRe：ALU运算器的运算结果

writeData：需要写进寄存器或者存储器的数值

readData1:就相当于rt

readData2：相当于rd

ExtendData：进行扩展后的数字

RAMDataout:只在lw指令中将寄存器中的值读出来

Jump_out：j跳转的指令地址

aluOp:选择进行什么运算

insToCtr：opcode

PcSrc：00 pc+4；01：Branch；10：j型跳转

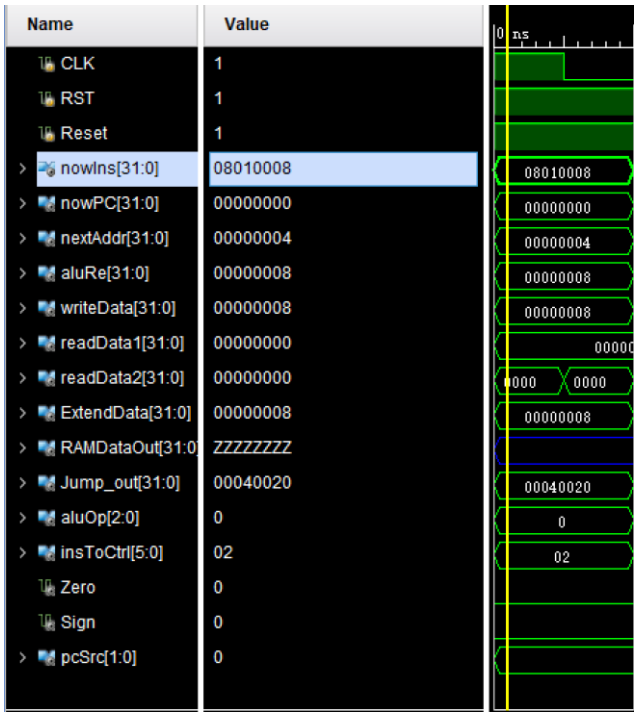
下面我们一条指令一条指令的进行验证

（1） addi $1,$0,8

这条指令的目的是（$1）=（$0）+8，我们现在来看波形：readData1=0（$0），aluRe是8（0+8=8，正确），writeData是8，因此这条指令的执行是完全正确的。

该条指令结束后，寄存器的情况如下：

$1=8
$2
$3
$4
$5
$6
$7
$8
$9
$10
$11



（2） ori $2,$0,2

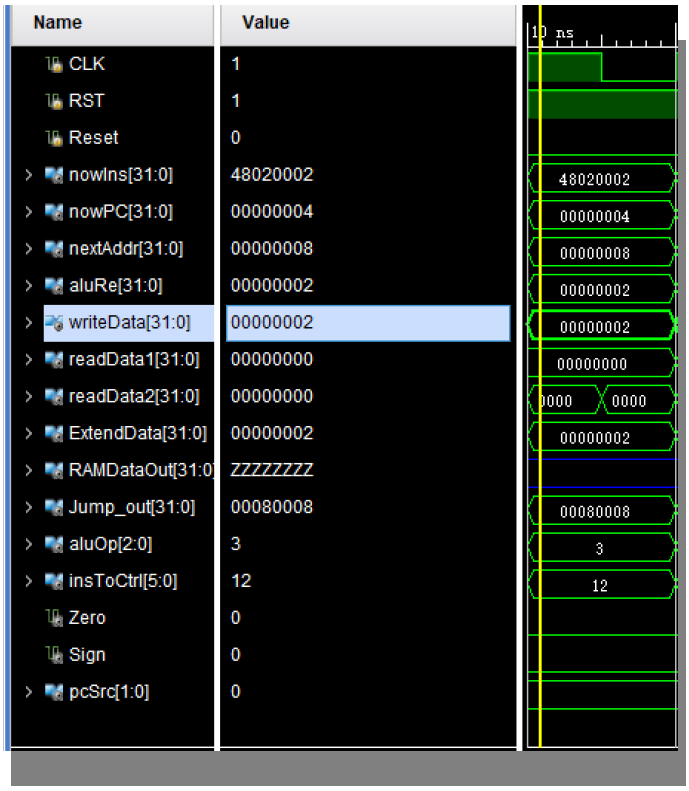
这条指令的目的是（$2）=（$0）| 2，

000 | 010 =010；

我们现在来看波形：readData1=0（$0），**aluRe**是2（正确），**writeData是2，因此这条指令的执行是完全正确的。**

该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3
$4
$5
$6
$7
$8
$9
$10
$11

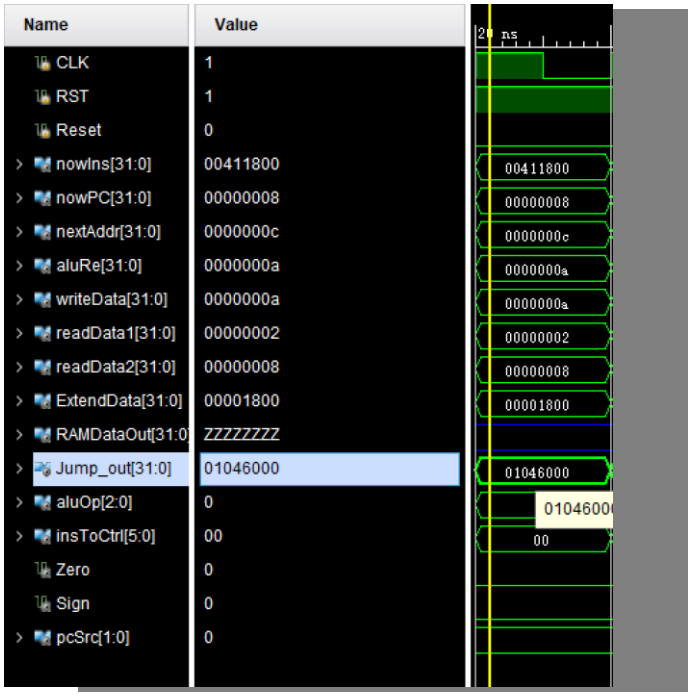


（3） **add $3,$2,$1**

这条指令的目的是（$3）=（$2）+（$1），

我们现在来看波形：readData1=2（$2），readData2=1（$1），aluRe是10（正确），writeData是10，，因此这条指令的执行是完全正确的。该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4
$5=8
$6
$7
$8
$9
$10
$11



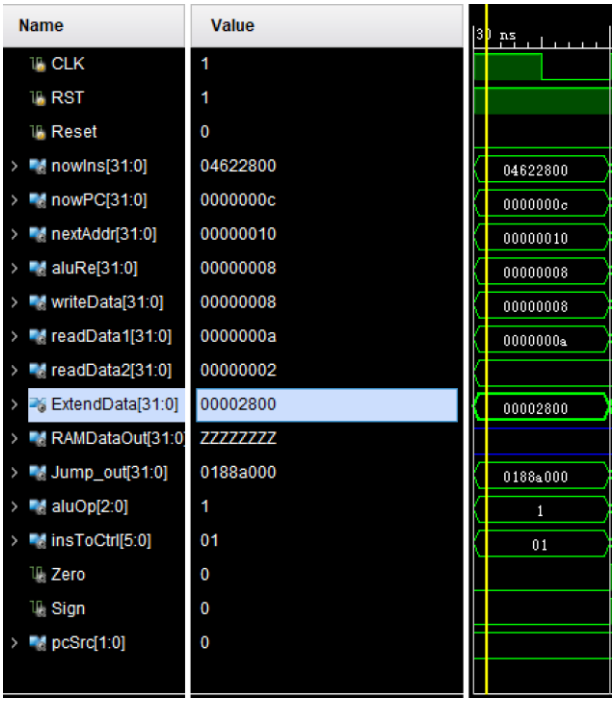
(4)sub $5,$3,$2

这条指令的目的是（$5）=（$3）-（$2），

我们现在来看波形：readData1=3（$3），readData2=2（$3），**aluRe**是8（正确），**writeData是8，因此这条指令的执行是完全正确的。**

该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4
$5=8
$6
$7
$8
$9
$10
$11



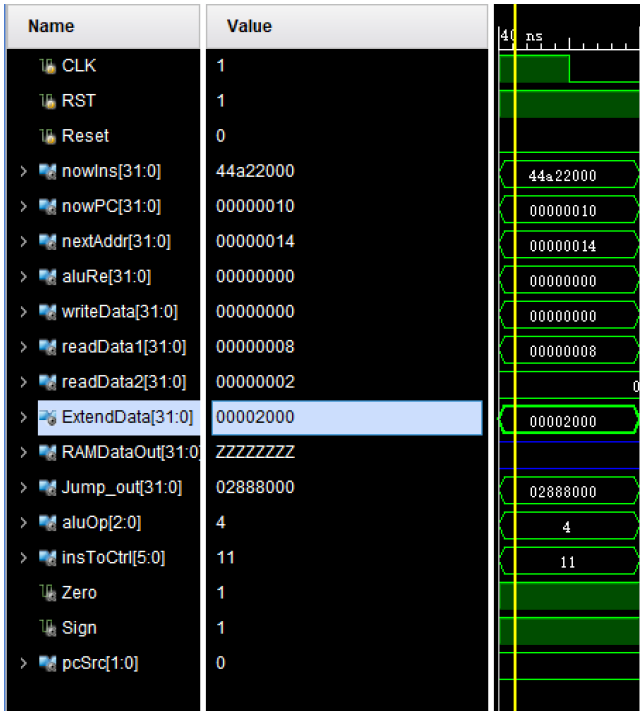
(5)and　$4,$5,$2

这条指令的目的是（$4）=（$5）&（$2），

我们现在来看波形：readData1=8（$5），readData2=2（$3），**aluRe**是0（正确），**writeData是0，因此这条指令的执行是完全正确的。**

该条指令结束后，寄存器情况如下：

$1=8
$2=2
$3=10
$4=0
$5=8
$6
$7
$8
$9
$10
$11



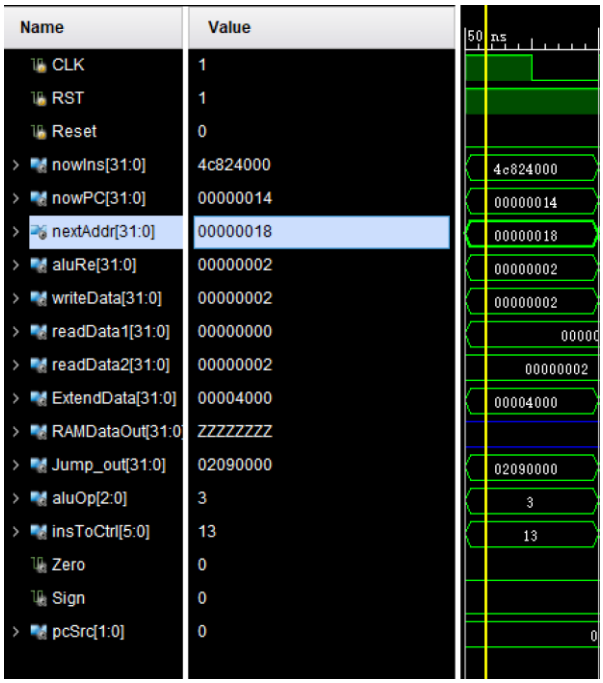
（6）or　$8,$4,$2

这条指令的目的是（$8）=（$4）|（$2），

我们现在来看波形：readData1=0（$3），readData2=2（$3），**aluRe**是2（正确），**writeData是8，因此这条指令的执行是完全正确的。**

该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4=0
$5=8
$6
$7
$8=2
$9
$10
$11



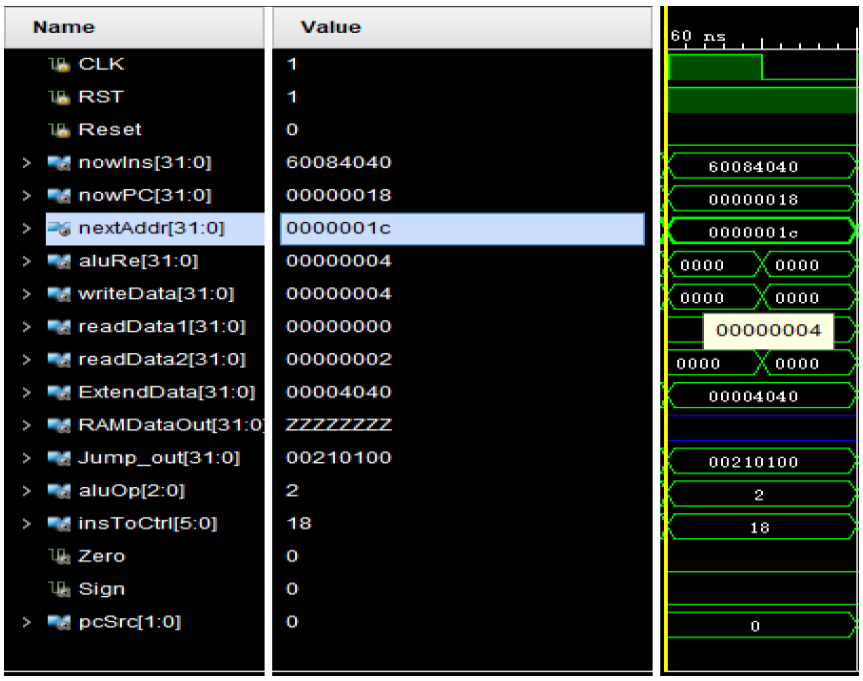
(7)sll $8,$8,1

这条指令的目的是（$8）=（$8）<<1，

我们现在来看波形：readData1=0（$3），readData2=2（$3），**aluRe**是2（正确），**writeData是8，因此这条指令的执行是完全正确的。**

该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4=0
$5=8
$6
$7
$8=4
$9
$10
$11



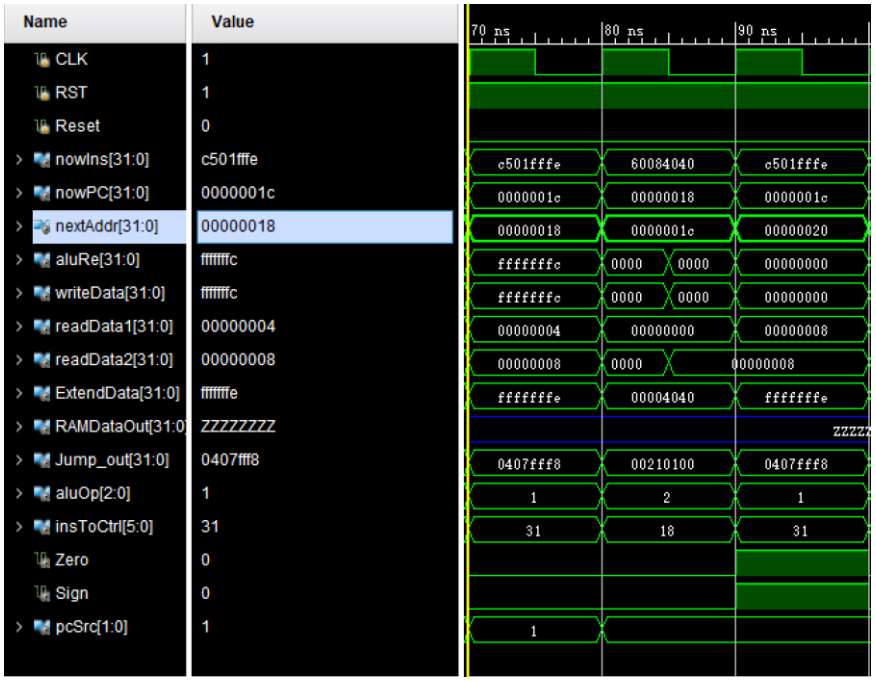
(8)bne $8,$1,-2 （≠,转18)

这条指令的目的是跳转指令，由于（$1）=8≠（$8）=4，所以指令要跳到18，然后（$8）=8，再次进入指令bne $8,$1,-2 ，但是这一次，由于（$1）=8=（$8）=8，所以不会进行跳转。

同时我们也可以观察到第一次进入bne $8,$1,-2，nexAdress=00000018，符合预期结果。因此执行都是正确的

该三条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4=0
$5=8
$6
$7
$8=8
$9
$10
$11



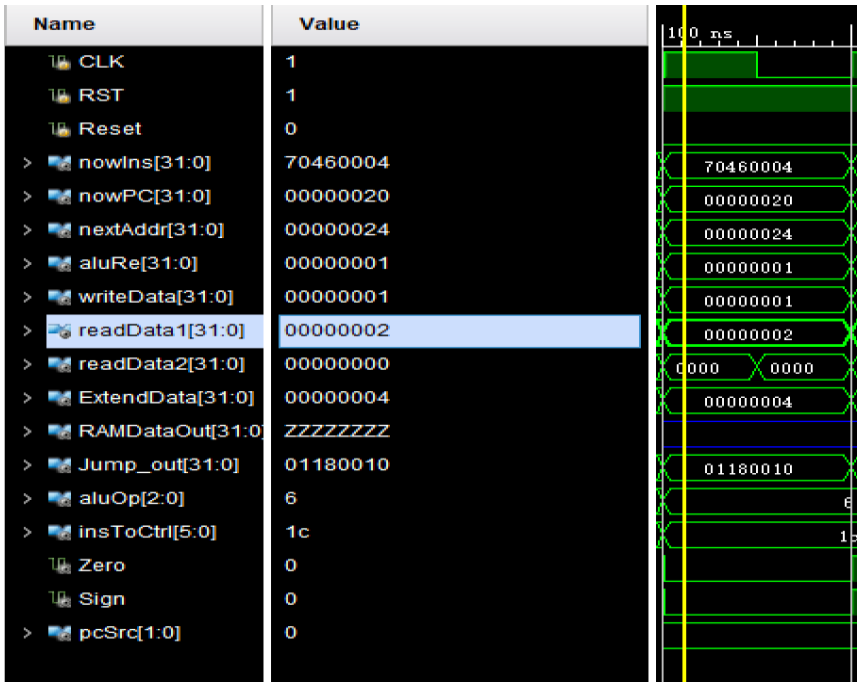
（9）slti $6,$2,4

这条指令的目的是进行比较，因为（$2）=2<4,所以$6=1；

我们现在来看波形：readData1=2（$2），aluRe是1（正确），writeData是1，因此这条指令的执行是完全正确的。

该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7
$8=8
$9
$10
$11



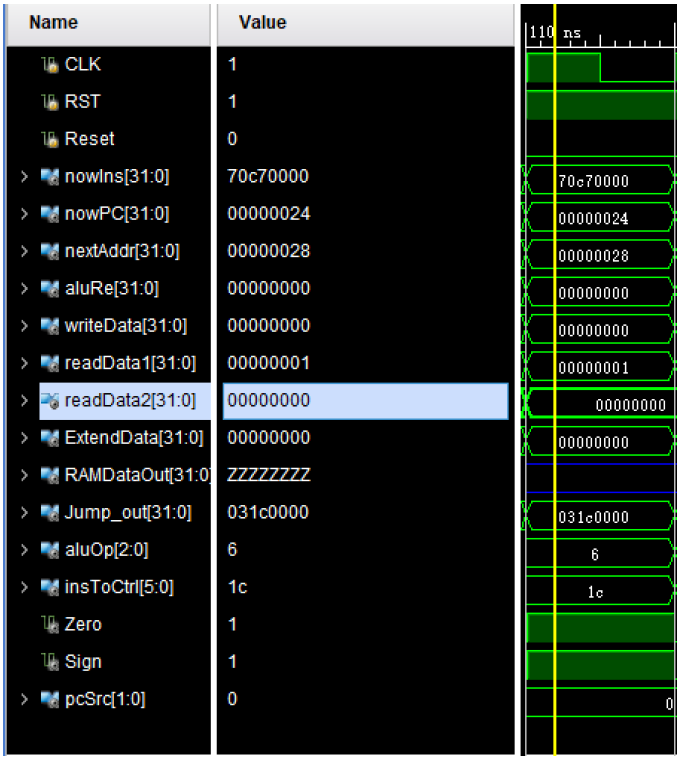
(10)slti $7,$6,0

这条指令的目的是进行比较，因为（$6）=1>0,所以**$7=0；**

我们现在来看波形：readData1=6（$2），**aluRe**是0（正确），**writeData是0，因此这条指令的执行是完全正确的。**

该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=0
$8=8
$9
$10
$11

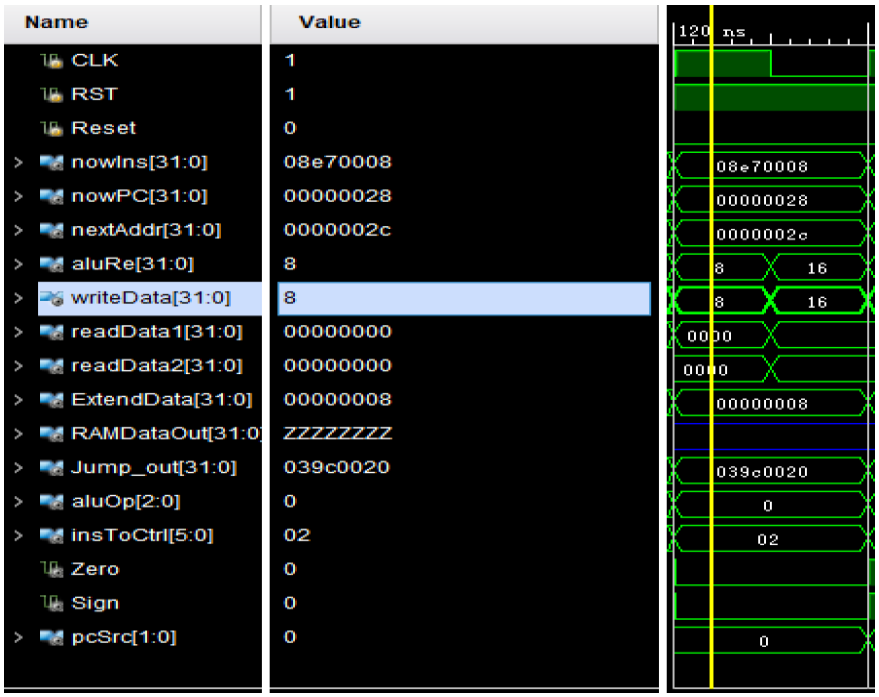


(11)addiu　$7,$7,8

这条指令的目的是（$7）=（$7）+8，我们现在来看波形：readData1=0（$0），aluRe是8（0+8=8，正确），writeData是8，因此这条指令的执行是完全正确的。

该条指令结束后，寄存器的情况如下：

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=8
$8
$9
$10
$11



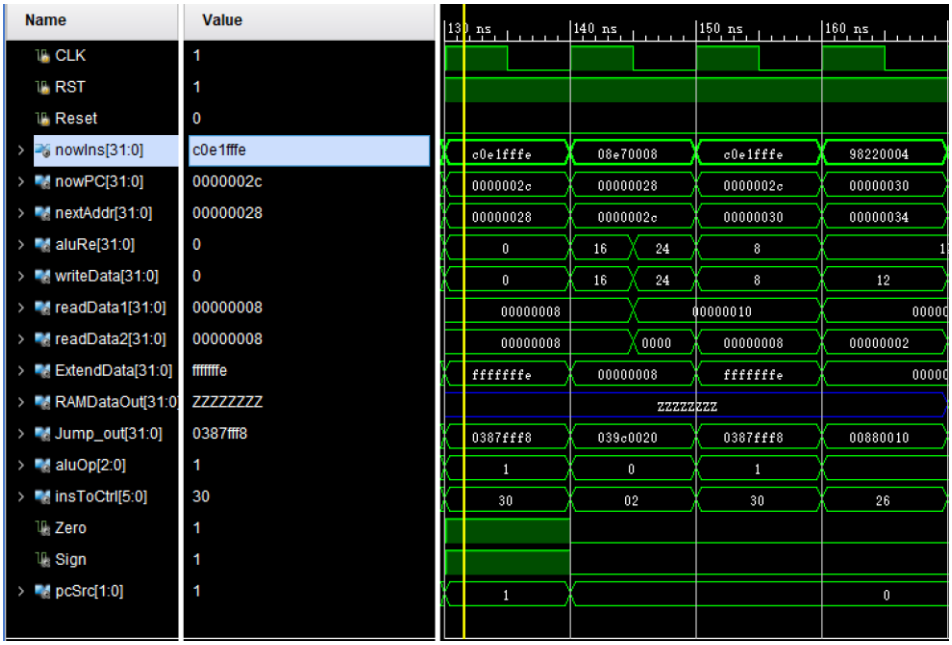
(12)beq $7,$1,-2　(=,转28)

这条指令的目的是跳转指令，由于（$1）=8==（$7），所以指令要跳到28，然后（$7）=16，再次进入指令beq　$7,$1,-2　，但是这一次，由于（$1）=8≠（$7）=16，所以不会进行跳转。

同时我们也可以观察到第一次进入beq $7,$1,-2，nexAdress=00000028，符合预期结果。因此执行都是正确的

该三条指令结束后，寄存器的情况如下：

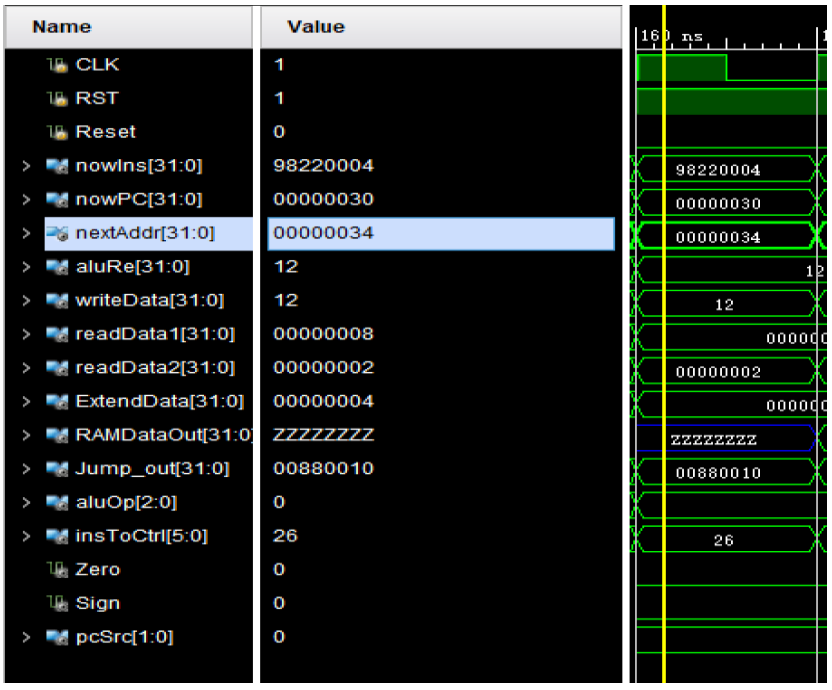
$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9
$10
$11



(13)sw　$2,4($1)

这条指令的目的是将寄存器$2的值存到mem[12]中；

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9
$10
$11

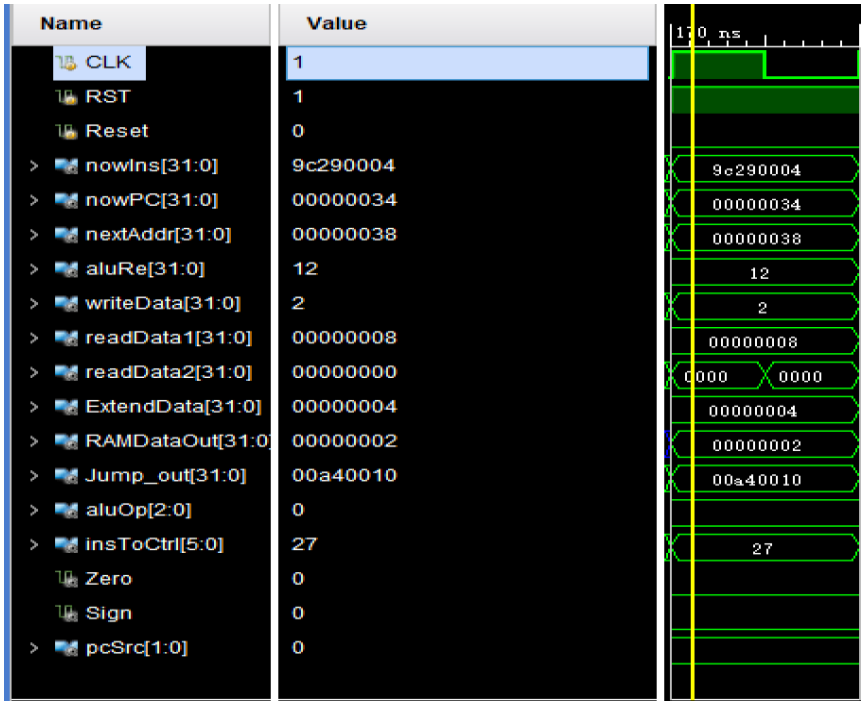


（14）lw　$9,4($1)

这条指令的目的是将mem[12]的值存到寄存器$9中，

同时我们发现波形中跟SW指令有关的RAMDataOut输出不再为高阻抗了。

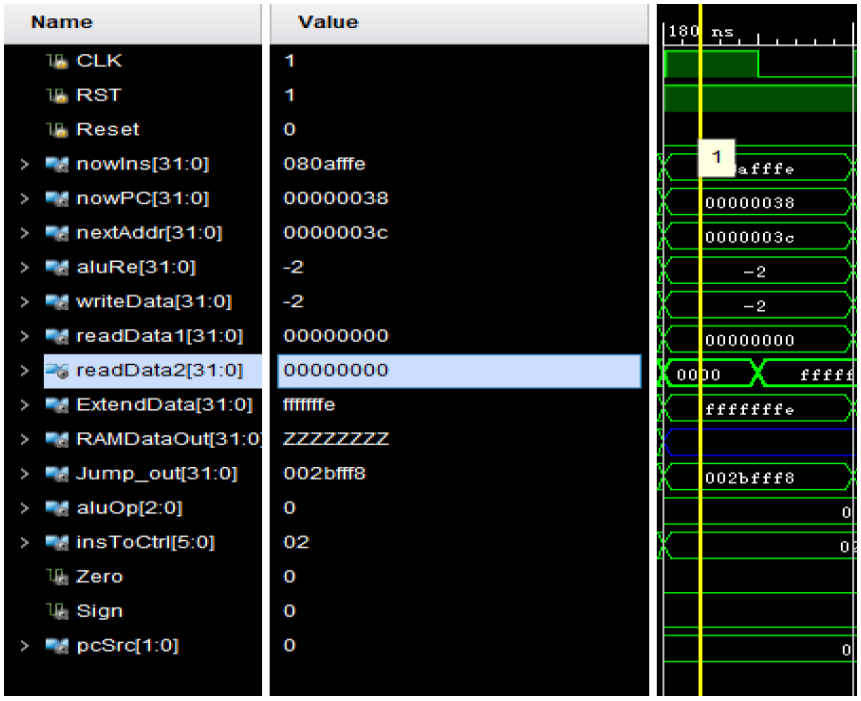
$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9=2
$10
$11



（15）addiu　$10,$0,-2

跟上面的addiu指令一样的效果，不再赘述。

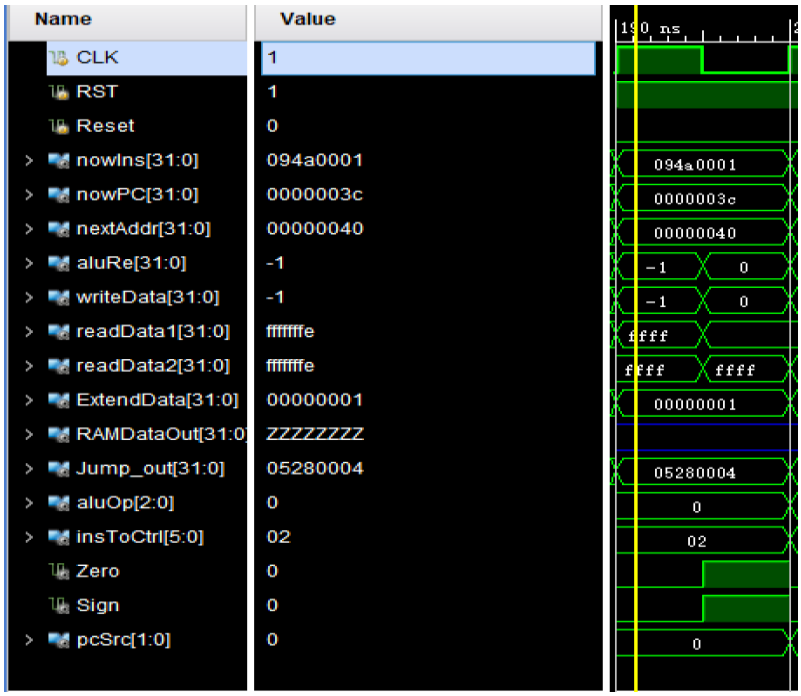
$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9=2
$10=-2
$11



（16）addiu　$10,$10,1

同上。

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9=2
$10=-1
$11

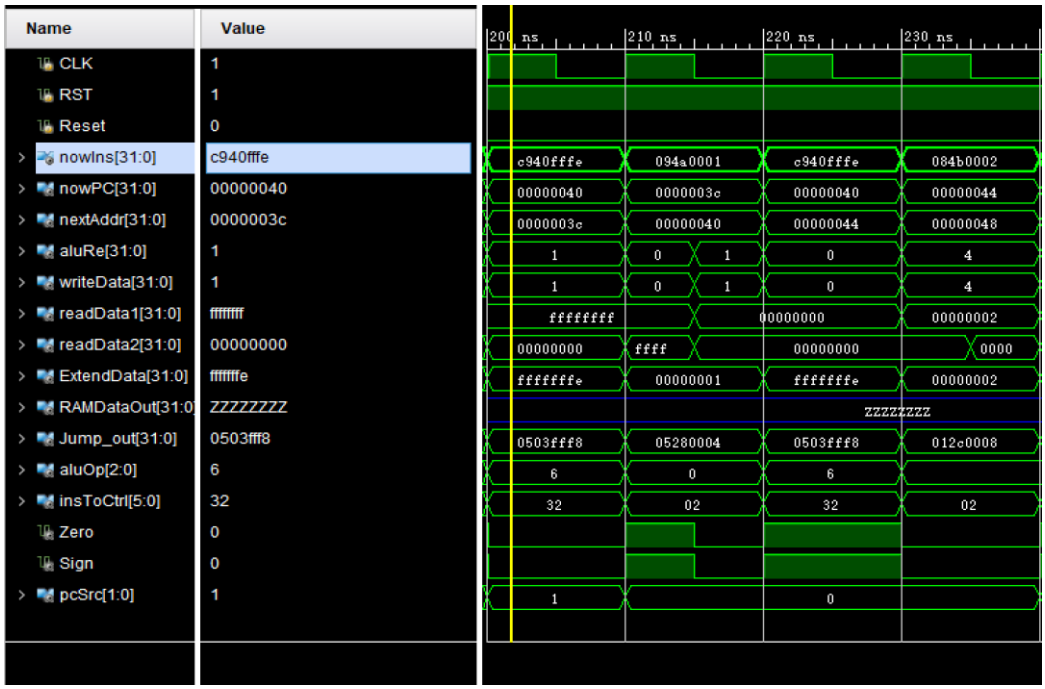


（17）bltz $10,-2(<0,转3C)

由于$10=-1<0，所以我们需要跳转到3c，此后，$10=0，第二次进入指令**bltz $10,-2，但是第二次不会进行跳转了。**

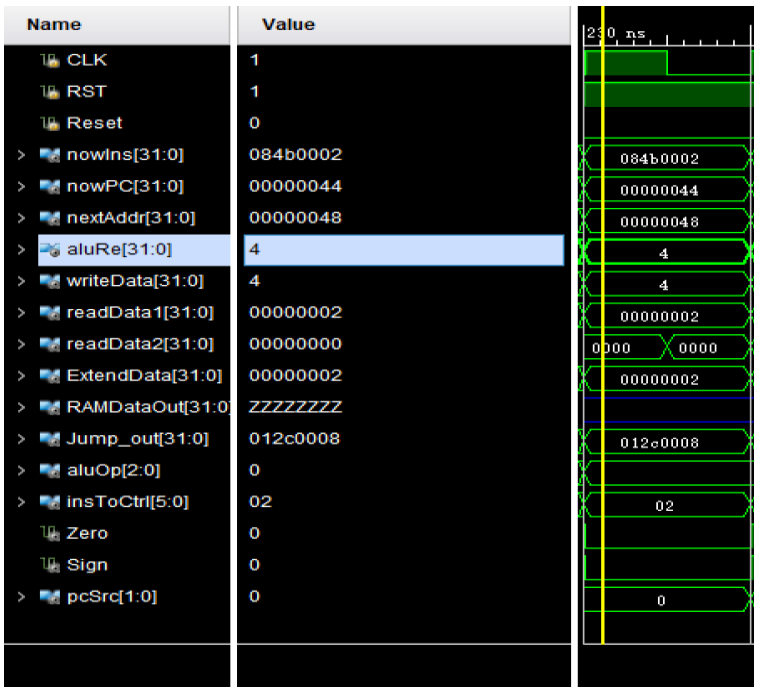
同时我们也可以观察到第一次进入**bltz $10,-2，nexAdress=0000003C，符合预期结果。因此执行都是正确的**

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9=2
$10=-1
$11



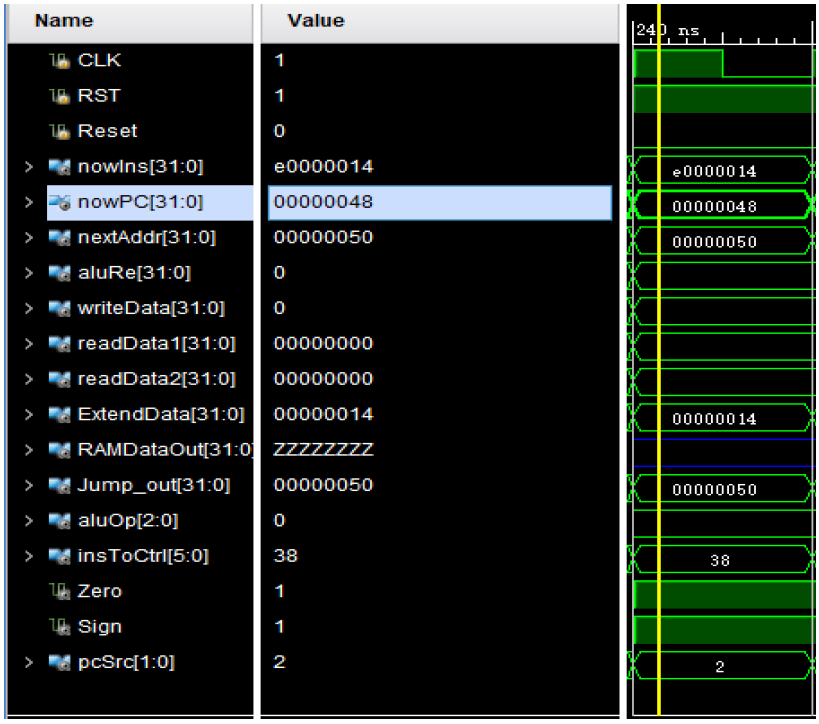
(18) andi $11,$2,2

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9=2
$10=-1
$11



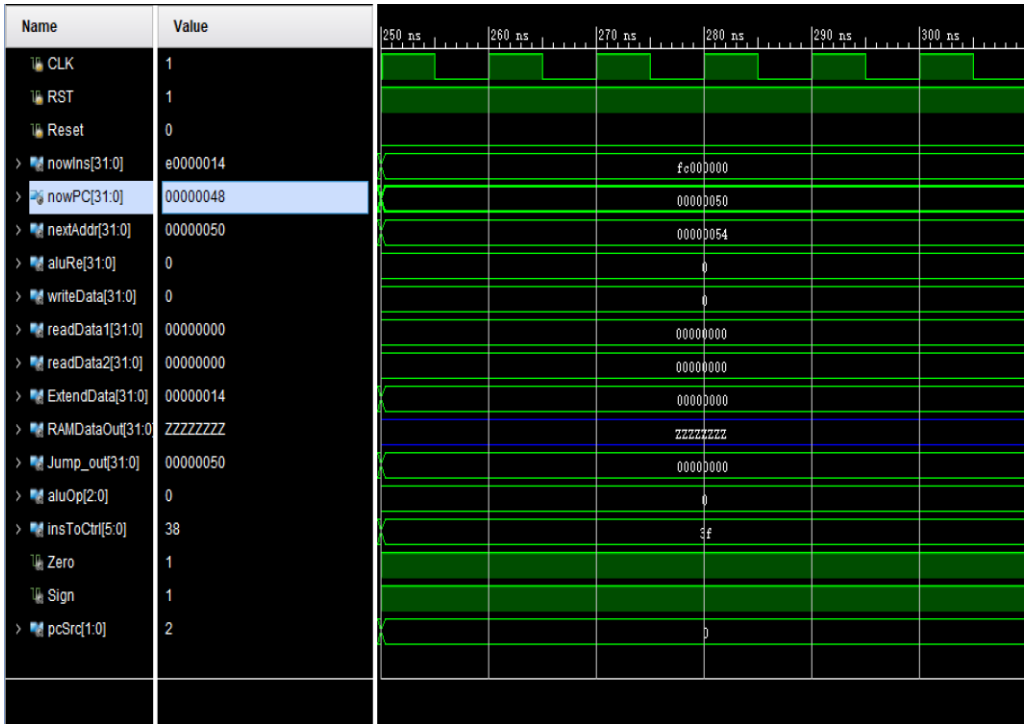
(19) j 0x00000050

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9=2
$10=-1
$11



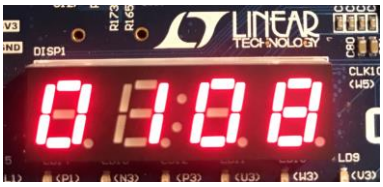
Halt

$1=8
$2=2
$3=10
$4=0
$5=8
$6=1
$7=16
$8
$9=2
$10=-1
$11



Basys板

0x00000000 addiu $1,$0,8



图一表示该条指令为00，下一条指令为04；
图二表示rt=$1=8;
图三表示rd=$0=0;
图四表示aluRe=8，writeData=8.

0x00000004 ori $2,$0,2



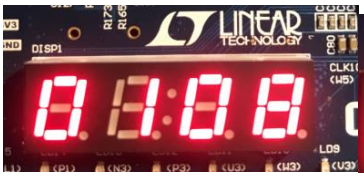

图一表示该条指令为04，下一条指令为08；图二表示rt=$2=2;

图三表示rd=$0=0;

图四表示aluRe=2，writeData=2.

0x00000008 add $3,$2,$1






图一表示该条指令为08，下一条指令为0c；

图二表示rt=$2=2;

图三表示rd=$1=8;

图四表示aluRe=a，writeData=a.

0x0000000C sub $5,$3,$2



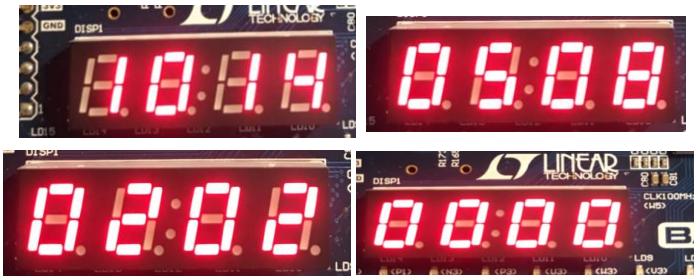

图一表示该条指令为0c，下一条指令为10；

图二表示rt=$3=a;

图三表示rd=$2=2;

图四表示aluRe=8，writeData=8.

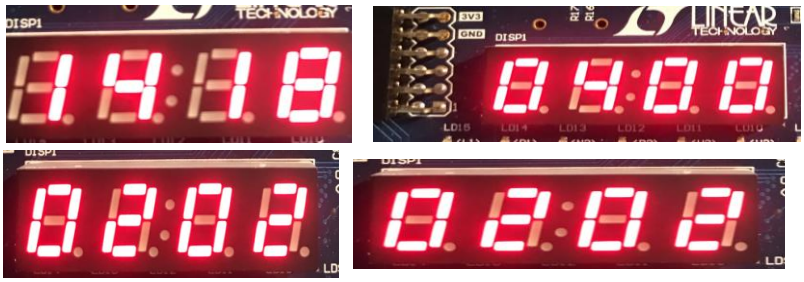
0x00000010 and $4,$5,$2



图一表示该条指令为10，下一条指令为14；

图二表示rt=$5=8;
图三表示rd=$2=2;
图四表示aluRe=0，writeData=0.

0x00000014 or $8,$4,$2



图一表示该条指令为14，下一条指令为18；

图二表示rt=$4=0;
图三表示rd=$2=2;
图四表示aluRe=2，writeData=2.

Basys板的设计思路：

1、实现CPU在板上运行需要两个时钟信号，CPU工作时钟和Basys3板系统时钟。CPU 工作时钟即为按键，是CPU正常工作时钟信号，按键必须进行消抖处理；Basys3板系 统时钟即为板提供的正常工作时钟信号，即为100MHZ。Basys3板系统时钟信号引脚 对应管脚W5。

2、每个按键周期，4个数码管都必须刷新一次。数码管位控信号 AN3-AN0是1110、1101、1011、0111，为0时点亮该数码管，当然，还应该为数码管各位“1gfedcba”引脚输出信号，最高位为“1”。比如，“当前PC值”低8位中的高4位和低4 位，必须经下页转换后送给数码管各引脚。 显示模块设计大概分为4个部分：

（1）对Basys3板系统时钟信号进行分频，分频的目的用于计数器；

（2）生成计数器，计数器用于产生4个数。这4数用于控制4个数码管；

（3）根据计数器产生的数生成数码管相应的位控信号（输出）和接收CPU来的相应数据；

（4）将从CPU 接收到的相应数据转换为数码管显示信号，再送往数码管显示（输出），即下页内 容。 还必须清楚，数码管显示的内容是受开关控制的，不同情况显示内容是不同的。

消抖处理的模块：

fangdou.v *

C:/Users/82470/Desktop/vivadofile/Single_CPU/Single_CPU.srcs/sources_1/new/fangdou.v

```
    reg key_rrr,key_rr,key_r;
    always @(posedge /*clk_100Hz*/ clk or negedge reset) begin
        if(!reset)
            begin
                key_rrr <= 1'b1;
                key_rr <= 1'b1;
                key_r <= 1'b1;
            end
        else
            begin
                key_rrr <= key_rr;
                key_rr <= key_r;
                key_r <= key_in;
            end
    end
    //----------------------------------------------------------
    assign key_out = key_rrr & key_rr & key_r;    // debkey的值为三个寄存器相与(各位都独立)

endmodule
```

因为只有当连续三个时钟周期按键电平都未改变的情况下，才可以认为此时按键稳

实验心得：

我终于在各种考试的重压下将单周期CPU设计了出来，里面的种种困难是显然的。首先一上来我其实是比较顺利的做出来了几大模块，因为老师的ppt上面都是很详细的，所以这个不太难，但是做到一半我就发现了问题，因为老师给的opcode上下不是一样的，所以我只好重新再更正老师给的ppt，于是乎，我很顺利地做出来了波形，当然在这期间我也遇到了一个问题就是debug的时候大小写不分，于是在我的sim模块出现了极其多的bug，所幸这些问题都能过通过一点一点的解决，还有就是我对模块的调用特别的生疏，现在觉得模块就相当于c++的函数，直接传参数就行了，但是我第一次做的时候还是特别的不熟练，对reg变量和wire变量的使用都是极其的模棱两可。还有就是我终于懂了原来顶层文件就是连一个电路图出来。

于是就来到了烧板的阶段，这个阶段是最痛苦的，我整整花了一个星期的时间来解决。遇到的第一个问题是我不知道为什么老师在课上面给我们讲要消除抖动，通过我在网上查找资料，我发现是这个原因：用按钮模拟时钟需要消抖处理，可以设置tmp，每隔一小段间隔t才将tmp更新为当前按钮值，那么短时间内的重复按

下也只会产生单步脉冲。其次，我之前一直很粗心大意，我根本不区分=>和=，导致一开始basys板显示乱七八糟，还有就是always不能混用边沿敏感和电平敏感。边沿敏感即指定上升沿或下降沿触发，常用于时钟信号，也可用于异步信号；电平敏感即电平变化时触发，常用于变量。最让我无语的一个bug是我的中文路径，导致我一直不能够实现综合，没有任何错误却failed，问了很多的同学都不能找出问题，于是换了一台电脑，发现是因为中文路径的问题，确实是很绝望了呢。还有一个低级的错误是我引脚的分配有问题，导致我的basys板的最后一位一直显示0，最后才发现是因为引脚分配错误。这期间还有一个错误是我的beq指令陷入了死循环，不觉明厉。于是我按照一个同学的解决方案：先按下按钮不松开，再把reset置1，再松开按钮，然后就可以正常运行了，我并不知道到底为什么这样就能够解决这个问题。

总结：

纸上得来终觉浅，绝知此事要躬行。Cpu真的是一个精细的活，任何一个环节出错了，都可能导致整个项目的崩溃。在做单周期CPU这期间，我问了好多的同学的问题，里面遇到了各种各样的奇奇怪怪的问题，真的是不知所措，尝试了很多种的方法，当每一次都宣告失败的时候，真的是很难受，不过最后终于做出来的感觉还是极其好的。记得当初上单周期cpu的理论课的时候，我只能的是很懵逼，不知道为什么要那样进行，不知道这样设计有什么好处，反正就是很被动的接受了书本上的那些结论，时间一长就什么都忘了，怪不得需要开实验课呢，感觉自己写了单周期的cpu之后，我终于懂了各种信号之间的关系，以及各种部件的联系与交互，真的是很神奇，不知道当初设计出cpu的人是谁，真的是太聪明了。做科研真的是一件需要特别仔细的事情，如果像我之前那样对概念模棱两可，那可能真的就什么都做出来了，需要无穷的耐心和热情，这可能就是我现在所缺少的把。总之，单周期cpu的设计不仅帮我巩固了自己的知识，还锻炼了我的实操能力，锻炼了我看波形的能力，又帮我静了下来。挺满足的。