中山大學
SUN YAT-SEN UNIVERSITY

《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）： 16 计算机类 2 班

学 生 姓 名 ： 欧阳润宇

学 号 ： 16337188

时 间 ： 2017 年 11 月 21 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；
(5) 掌握单周期 CPU 的实现方法。
(6) 在 Basys3 实验板上模拟单周期 CPU 的工作。

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** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

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

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

（2）addi rt , rs ,**immediate**

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

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

（3）sub rd , rs , rt

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

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

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

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

（5）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

（7）andi rd , rs , rt

010011	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。

（8）xori　rd , rs , rt

010101	rs(5 位)	rt(5 位)	rd(5 位)	reserved

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

==>移位指令

（9）sll　rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

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

==>比较指令

（10）　slt　rd, rs, rt　　带符号数

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

功能：if (rs<rt)　rd =1 else　rd=0，具体请看表 2 ALU 运算功能表，带符号

==> 存储器读/写指令

（11）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。

（12) 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

（13）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”。

(14) bne　rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

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

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

（15）bgtz　rs,**immediate**

110010	rs(5 位)	00000	**immediate**

功能：if(rs>0) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4

==>跳转指令

（16）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],0,0}，无条件跳转。

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

==> 停机指令

（17）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。

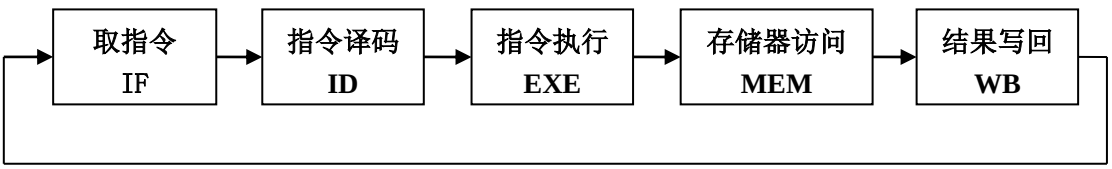
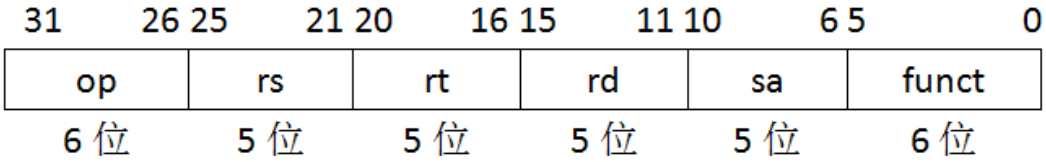


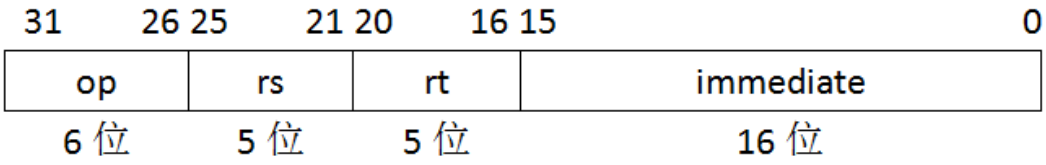
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

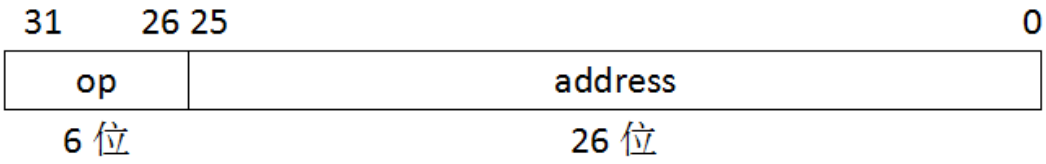
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

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

rd：**只写**。为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

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

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

address：为地址。

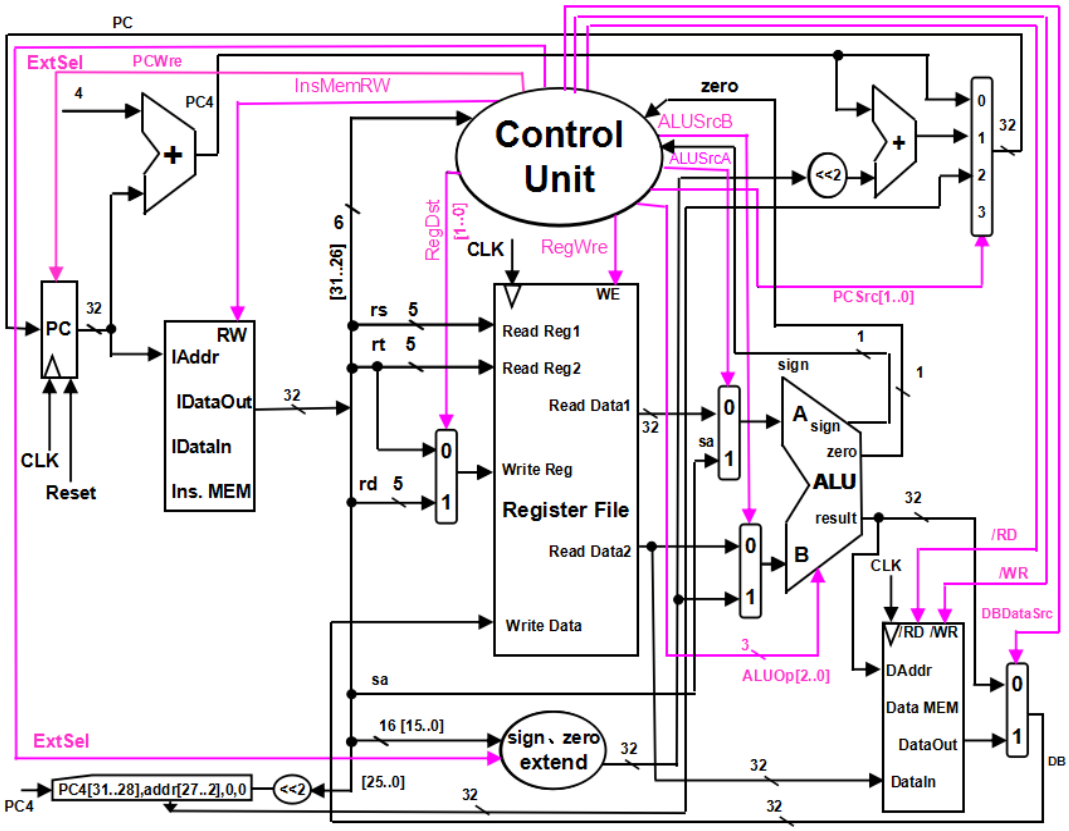


图 2　单周期 CPU 数据通路和控制线路图

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，输出端就直接输出相应数据；而在写操作时，在 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。

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、模块设计与代码：

（1） PC：

a、输入输出和真值表

Input:PrcWre(1 位，控制单元的输出)、PrcSrc(1 位，控制单元的输出)、

immediate（32 位，扩展单元的输出)、address（25 位，指令存储器的输出)、CLK（时钟)、Reset

Output:PC

PrcWre	PrcSrc	Reset	PC(次态)
x	x	0	0
0	x	1	不变
1	00	1	PC+4
1	01	1	PC+4+(sign-extend)immediate
1	10	1	{(PC+4)[31..28],address[27..2],0,0}
1	11	1	不变

b、核心代码：

```
module JumpPC(PC, InAddress, Out);
input [31:0] PC;
input [25:0] InAddress;
output reg [31:0] Out;

always @(PC or InAddress)
begin
     Out[31:28]=PC[31:28];
     Out[27:2]=(InAddress>>2);
     Out[1:0]=0;
end

endmodule
```

25位的InAddress来自指令存储器，这一模块是专门用于跳转指令的下一条PC地址

```
module NextPC(Reset, PC, Immediate, JumpPC, PCSrc, NextPC);
input Reset;
input [31:0] PC, Immediate;
input [31:0] JumpPC;
input [1:0] PCSrc;
output reg [31:0] NextPC;


always@(Reset or PCSrc or PC or Immediate or JumpPC) begin
        if(Reset==0)NextPC=PC+4;
        else begin
            case(PCSrc)
                2'b00: NextPC=PC+4;
                2'b01: NextPC=PC+4+(Immediate<<2);
                2'b10: begin
                    NextPC=JumpPC;
                end
                2'b11:NextPC=PC;
            endcase
        end
end

endmodule
```

JumpPC是JumpPC模块的输出，Immeditate是扩展单元的输出，PC是上一条指令，PCSrc是控制单元的输出。这一模块用于得到下一条PC。

```
module PC(CLK, Reset, NextPC, PCWre, IAddr);
input CLK, Reset;
input [31:0] NextPC;
input PCWre;
output reg [31:0] IAddr;

initial begin
    IAddr=0;
end

always @(posedge CLK or negedge Reset)
begin
    if(Reset==0)IAddr<=32'hFFFFFFFC;
    else if(PCWre==1||NextPC==0)IAddr<=NextPC;
    //else IAddr<=32'bz;
end
endmodule
```

这一模块用于在时钟上升沿让PC改换成下一条PC，需要注意的是PC初始化为-4，这是为了在basys3板上模拟CPU的时候能够按下按键之后CPU才开始工作，同时也方便寄存器组在时钟下降沿写入数据。

(2) InstructionMemory:

a、输入输出和真值表

Input: InsMemRW（控制单元的输出)、IDataIn(用于写指令的输入)、IAddr(PC 的输出)

Output:InsOut(op、rs、rt、rd、sa、immediate)

在指令存储器中，有一个长度为八位的储存器组（有 100 个储存器）Memory，其中每 4 个储存器储存一条指令，总共储存 20 条指令。

InsOut 则为当前 PC 所对应的指令，其中：

InsOut[31:24]=Memory[IAddr];

InsOut[23:16]=Memory[IAddr+1];

InsOut[15:8]=Memory[IAddr+2];

InsOut[7:0]=Memory[IAddr+3];

对于 InsOut 又可以将它划分为一下的几个部分：

op=Memory[i][31:26],

rs=Memory[i][25:21],

rt=Memory[i][20:16],

rd=Memory[i][15:11],

sa= Memory[i][10:6]（仅仅用于左移），
immediate=Memory[i][15:0](用于有立即数的运算),
address=Memory[i][25:0](仅用于 j、halt

b、代码

```
module InstructionMemory(InsMemRW,IDataIn,IAddr,Out);
input InsMemRW;
input [31:0] IDataIn,IAddr;
output reg [31:0] Out;

reg [7:0] Memory[0:99];
initial begin
   $readmemb("E:/Vivado/SingleStyleCPU/data/InstructionMemory.txt",Memory);
end

always @(IAddr or InsMemRW) begin
    begin
        Out[31:24]=Memory[IAddr];
        Out[23:16]=Memory[IAddr+1];
        Out[15:8]=Memory[IAddr+2];
        Out[7:0]=Memory[IAddr+3];
    end
end

endmodule
```

一开始先将txt文件中储存的指令读取到寄存器组中，注意op、rs等输出不在本模块中体现，只在顶层模块体现。

(3) ControlUnit:

a、输入输出和真值表

Input: sign、zero（ALU 的输出）、Op（来自指令储存器）
Output:PCWre、ALUSrcA、ALUSrcB、DBDataSrc、RegWre、RD、WR、RegDst、ExtSel、PCSrc（2 位）、ALUOP（3 位）

控制信号的作用

输出	状态“0”	状态“1”
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、bgtz、slt、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、slt、beq、bne、bgtz	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、sw、lw

DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slt、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bgtz、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slt、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
RD	读数据存储器，相关指令：lw	输出高阻态
WR	写数据存储器，相关指令：sw	无操作
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、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、slt、sll
ExtSel	(zero-extend)**immediate(0 扩展)，**相关指令：ori	(sign-extend)**immediate（符号扩展）**，相关指令：addi、sw、lw、bne、bne、bgtz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1，或 zero=1); 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)、bgtz(sign=0，zero=0); 10：pc<－{(pc+4)[31..28],addr[27..2],0,0}，相关指令：j; 11：pc<-pc（即 PC 不改变），相关指令：halt	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，见 ALU 功能表	

真值表如下：

Op	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	InsMemRW	RegWre
000000(add)	1	0	0	0	0	1
000001(addi)	1	0	1	0	0	1
000010(sub)	1	0	0	0	0	1
010000(ori)	1	0	1	0	0	1
010001(and)	1	0	0	0	0	1
010010(or)	1	0	0	0	0	1
010011(andi)	1	0	1	0	0	1
010100(xor)	1	0	0	0	0	1

010101(xori)	1	0	1	0	0	1
011000(sll)	1	1	0	0	0	1
011100(slt)	1	0	0	0	0	1
100110(sw)	1	0	1	0	0	0
100111(lw)	1	0	1	1	0	1
110000(beq)	1	0	0	0	0	0
110001(bne)	1	0	0	0	0	0
110010(bgtz)	1	0	0	0	0	0
111000(j)	1	0	0	0	0	0
111111(halt)	0	0	0	0	0	0

Op	RD	WR	RegDst	ExtSel	PCSrc	ALUOP
000000(add)	1	1	1	0	00	000
000001(addi)	1	1	0	1	00	000
000010(sub)	1	1	1	0	00	001
010000(ori)	1	1	0	0	00	011
010001(and)	1	1	1	0	00	100
010010(or)	1	1	1	0	00	011
010011(andi)	1	1	0	0	00	100
010100(xor)	1	1	1	0	00	111
010101(xori)	1	1	0	0	00	111
011000(sll)	1	1	1	0	00	010
011100(slt)	1	1	1	0	00	110
100110(sw)	1	0	1	1	00	000
100111(lw)	0	1	0	1	00	000
110000(beq)	1	1	1	1	00(zero=0) 01(zero=1)	111
110001(bne)	1	1	1	1	01(zero=0) 00(zero=1)	111

110010(bgtz)	1	1	1	1	00(sign=1||zero=1) 01(sign=0&&zero=0)	111
111000(j)	1	1	1	0	10	000
111111(halt)	1	1	1	0	00	000

（注：三条分支指令用异或来实现）

（在读写指令、跳转指令和停机指令时设置ALU的控制输入为000）

b、代码：

```
assign PCWre=(Op==6'b111111)?0:1;
assign ALUSrcA=(Op==6'b011000)?1:0;
assign
ALUSrcB=((Op==6'b000001)||(Op==6'b010000)||(Op==6'b010011)|
|(Op==6'b010101)||(Op==6'b100110)||(Op==6'b100111))?1:0;
assign DBDataSrc=(Op==6'b100111)?1:0;
assign
RegWre=((Op==6'b100110)||(Op==6'b110000)||(Op==6'b110001)||
(Op==6'b110010)||(Op==6'b111000)||(Op==6'b111111))?0:1;
assign InsMemRW=0;//本次实验没有写指令
assign RD=(Op==6'b100111)?0:1’bz;
assign WR=(Op==6'b100110)?0:1;
assign
RegDst=((Op==6'b000001)||(Op==6'b010000)||(Op==6'b010011)||(
Op==6'b010101)||(Op==6'b100111))?0:1;
assign
ExtSel=((Op==6'b000001)||(Op==6'b100110)||(Op==6'b100111)||(
Op==6'b110000)||(Op==6'b110001)||(Op==6'b110010))?1:0;
assign PCSrc[1]=((Op==6'b111000)||(Op==6'b111111))?1:0;
assign
PCSrc[0]=((Op==6'b110000&&Zero==1)||(Op==6'b110001&&Zero==
0)||(Op==6'b110010&&Zero==0&&Sign==0)||(Op==6'b111111))?1:
```

0;

assign

ALUOp[2]=((Op==6'b010001)||(Op==6'b010011)||(Op==6'b010100)||(Op==6'b010101)||(Op==6'b011100)||(Op==6'b110000)||(Op==6'b110001)||(Op==6'b110010))?1:0;

assign

ALUOp[1]=((Op==6'b010000)||(Op==6'b010010)||(Op==6'b010100)||(Op==6'b010101)||(Op==6'b011000)||(Op==6'b011100)||(Op==6'b110000)||(Op==6'b110001)||(Op==6'b110010))?1:0;

assign

ALUOp[0]=((Op==6'b000010)||(Op==6'b010000)||(Op==6'b010010)||(Op==6'b010100)||(Op==6'b010101)||(Op==6'b110000)||(Op==6'b110001)||(Op==6'b110010))?1:0;

（4）Select:

代码：

```
module Select5(
    input Select,
    input [4:0] DataA,DataB,
    output [4:0] Data
    );
    assign Data=Select?DataB:DataA;
endmodule
```

```
module Select32(
    input Select,
    input [31:0] DataA,DataB,
    output [31:0] Data
);

assign Data=Select?DataB:DataA;

endmodule
```

这两个模块用于对数据进行选择。Select5用于选择需要写入的寄存器的地址（用RegDst进行选择）。Select32用于选择ALU的输入（用ALUSrcA和ALUSrcB进行选择）和写入寄存器的数据（用DBDataSrc进行选择）。

（5）RegisterFile:

a、输入输出和真值表

Input:DBDataSrc(来自控制单元)、ReadReg1(来自指令储存器的rs)、ReadReg2(来自指令储存器的 rt)、WriteReg (对来自指令存储器

的 rs、rd 进行选择后的结果)、WriteData (对 ALU 的 Result 和数据储存器的 DataOut 进行选择的结果)、WE(来自控制单元的输出 RegWre)、CLK（时钟)、Reset

Output:ReadData1、ReadData2

Register 是一个长度为 32 位的寄存器组（有 32 个寄存器)，初始化为 0。输出即为寄存器组中 ReadReg1 和 ReadReg2 对应的数据，即 ReadData1=Register[ReadReg1];

ReadData2=Register[ReadReg2];

在时钟的下降沿，如果 WE 是 1，则将数据写入寄存器，即：Register[WriteReg]<=WriteData;

（注意不能对 0 号寄存器写入数据）

b、代码

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

assign ReadData1=Register[ReadReg1];
assign ReadData2=Register[ReadReg2];

always @(negedge CLK)
begin
    if(Reset==0)begin
        Register[0]<=0;
    end
    else if(WE==1&&WriteReg!=0) Register[WriteReg]<=WriteData;
end
```

(6) SignZeroExtend:

a、输入输出和真值表

input:Immediate（来自指令存储器的输出）、ExtSel（控制单元的输出）、Sign（ALU的输出）

output:ExtOut

ExtSel	Sign	扩展方式
0	x	前 16 位补 0
1	0	前 16 位补 0
1	1	前 16 位补 1

b、代码：

```
module SignZeroExtend(Immediate,ExtSel,Sign,ExtOut);
input [15:0] Immediate;
input ExtSel,Sign;
output [31:0] ExtOut;

assign ExtOut[15:0]=Immediate[15:0];
assign ExtOut[31:16]=(ExtSel==1&&(Immediate[15]==1||Sign==1))?16'hFFFF:16'h0000;

endmodule
```

(7) ALU:

a、输入输出和真值表:

input：InA(经过选择后的寄存器组的输出 ReadData1 或扩展后的 Sa),InB(经过选择后的寄存器组的输出 ReadData2 或扩展后的 Immediate)、 ALUOp

output:Result、Zero、Sign

ALU 运算功能表

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

110	if (A<B &&(A[31] == B[31]))
	Y = 1;
	else if (A[31] && !B[31] Y = 1;
	else Y = 0;
		比较 A 与 B 带符号

111	Y = A ⊕ B	异或

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

b、代码

```
always @(ALUOp or InA or InB)

begin

    case(ALUOp)
```

```
3'b000:begin
    Result=(InA+InB);//A+B
  end
3'b001:begin
    Result=(InA-InB);//A-B
  end
3'b010:begin
    Result=(InB<<InA);//B<<A
  end
3'b011:begin
    Result=(InA|InB);//A|B
  end
3'b100:begin
    Result=(InA&InB);//A&B
  end
3'b101:begin
    Result=((InA<InB)?1:0);//A<B（无符号）
  end
3'b110:begin
    if(InA<InB&&(InA[31]==InB[31]))Result=1;
    else if(InA[31]&&!InB[31])Result=1;
    else Result=0;//A<B（有符号）
  end
3'b111:begin
    Result=(InA^InB);//A^B
  end
default:begin
    Result=0;
  end
```

```
        endcase
    end
    assign Zero=(Result==0)?1:0;
    assign Sign=Result[31];
```

(8) DataMemory:

a、输入输出和真值表:

Input:DAddr (ALU 的 Result 输出)、CLK、RD (来自控制单元的输出)、WR(来自控制单元)、DataIn(来自寄存器的 ReadData2)
Output:DataOut
在模块中有一个长度为 8 位的储存器组（共有 60 个储存器）Memory，初始化为 0

RD	WR	DataOut	功能
0	1	Memory[DAddr]	读数据，DataOut=Memory[DAddr]
1	0	1	写数据，Memory[DAddr]=DataIn
1	1	1	无

b、代码

```
assign DataOut[7:0] = (RD==0)?memory[DAddr + 3]:8'bz;
assign DataOut[15:8] = (RD==0)?memory[DAddr + 2]:8'bz;
assign DataOut[23:16] = (RD==0)?memory[DAddr + 1]:8'bz;
assign DataOut[31:24] = (RD==0)?memory[DAddr]:8'bz;

always@( negedge CLK )
begin
    if( WR==0 ) begin
        memory[DAddr] <= DataIn[31:24];
        memory[DAddr+1] <= DataIn[23:16];
        memory[DAddr+2] <= DataIn[15:8];
        memory[DAddr+3] <= DataIn[7:0];
    end
end
```

在RD不为0时，数据存储器的输出为高阻抗；

数据存储器写数据是在时钟的下降沿触发

(9) 顶层模块:

Input:CLK、Reset
Output:curPC(当前的 PC 地址)、NextPC(下一条 PC 地址)、Rs、Rt、

ReadData1（寄存器组的输出 1,对应 Rs)、ReadData2(寄存器组的输出 2,对应 Rt)、Result(ALU 的结果)、DataOut（数据存储器的输出）

顺序:

PC->InstructionMemory->ControlUnit->RegisterFile->SignZeroExtend->ALU->DataMemory

(省略了其中的一些选择模块)

2、 仿真结果和分析:

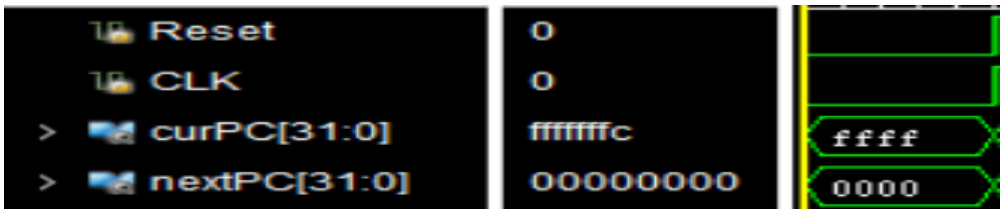
（1） 时钟代码:

```
initial begin
    CLK = 0;
    Reset = 0;
    #50;
    begin
        Reset=1;
        CLK=1;
    end
    forever #50 CLK=~CLK;
end
```

初始化时钟为 0，Reset 为 0。50ns 后启动，Reset 为 1，时钟为 1。此后 Reset 始终为 1，时钟周期为 100ns。

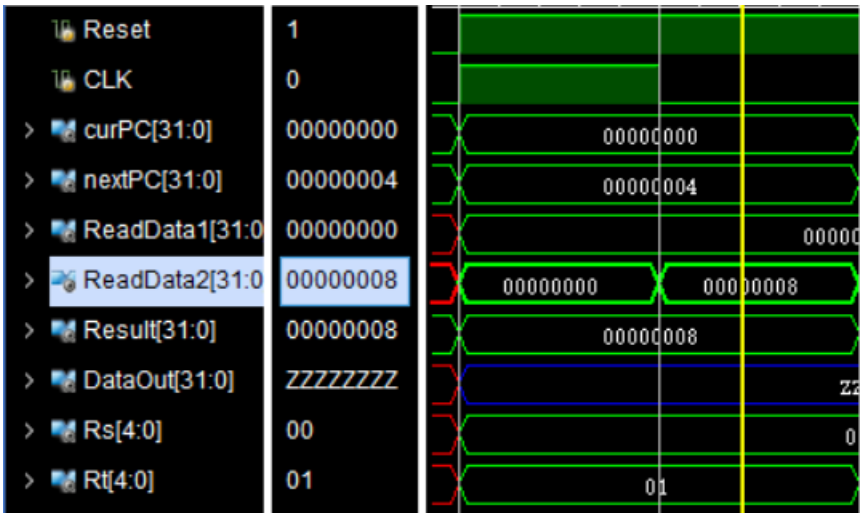
（2）仿真结果和分析:

初始情况:



CPU未开始工作时，当前地址为-4，下一条地址为0，正确。

立即数加法:



指令：addi　$1,$0,8

当前的地址为0，下一条地址为4，正确。

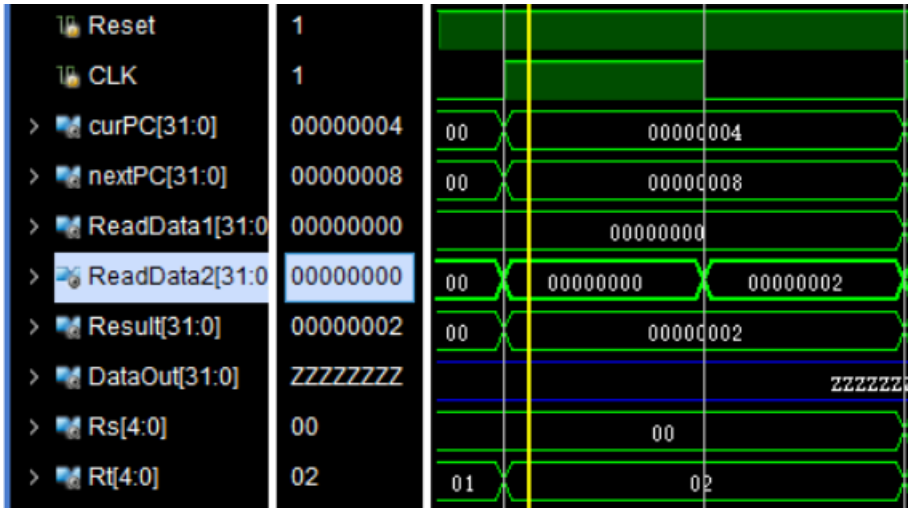
Rs为0号寄存器，对应的ReadData1的值为0，正确。

Rt为1号寄存器，对应的ReadData2的值在CLK=1时为0，在时钟的下降沿，$1=$0+8=8，ReadData2的值变为8，正确。

ALU的运算结果为0+8=8，DataOut在无需读数据时输出为高阻抗，正确。

（$1=8）

立即数逻辑或运算：



指令：ori　$2,$0,2

当前的地址为4，下一条地址为8，正确。

Rs为0号寄存器，对应的ReadData1的值为0，正确。

Rt为2号寄存器，对应的ReadData2的值在CLK=1时为0，在时钟的下降沿，$2=$0|2=2，ReadData2的值变为2，正确。

ALU的运算结果为0|2=2，DataOut在无需读数据时输出为高阻抗，正确。

($2=2)

加法运算：

add　$3,$2,$1

当前的地址为8（C），下一条地址为12（C），正确。

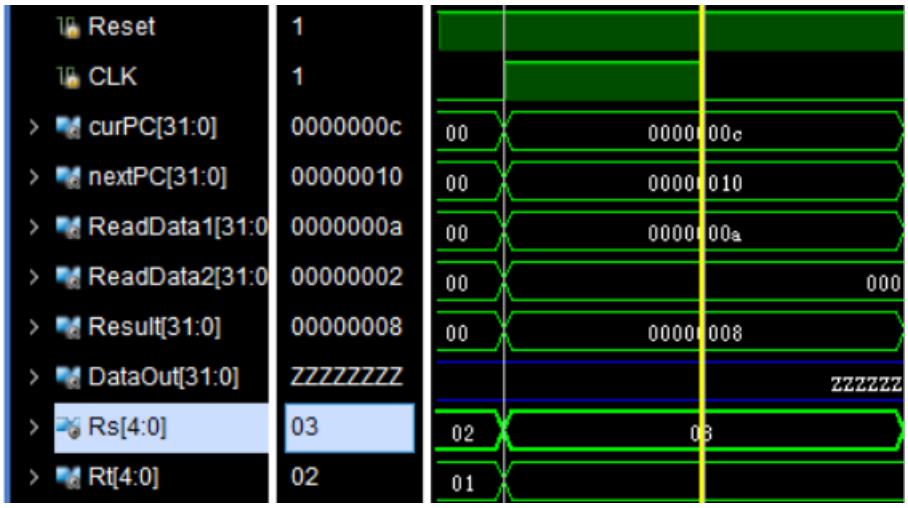
Rs为2号寄存器，对应的ReadData1的值为2，正确。

Rt为1号寄存器，对应的ReadData2的值为8，正确。（本条指令无需对寄存器Rt写入数据）

ALU的运算结果为8+2=10（a），DataOut在无需读数据时输出为高阻抗，正确。

($3=10)

减法运算：



sub $5, $3, $2

当前的地址为12（C），下一条地址为16（10），正确。

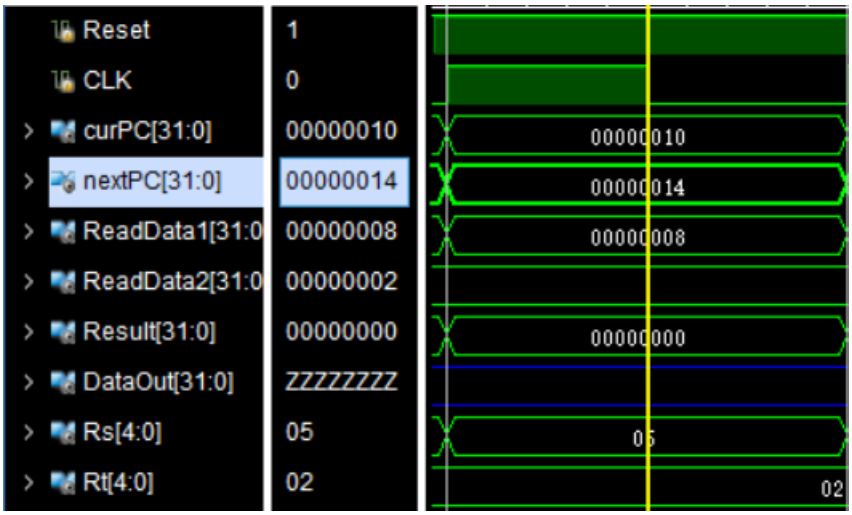
Rs为3号寄存器，对应的ReadData1的值为10，正确。

Rt为2号寄存器，对应的ReadData2的值为2，正确。（本条指令无需对寄存器Rt写入数据）

ALU的运算结果为10-2=8，DataOut在无需读数据时输出为高阻抗，正确。

（$5=8）

逻辑与运算：



and $4,$5,$2

当前的地址为16（10），下一条地址为20（14），正确。

Rs为5号寄存器，对应的ReadData1的值为8，正确。

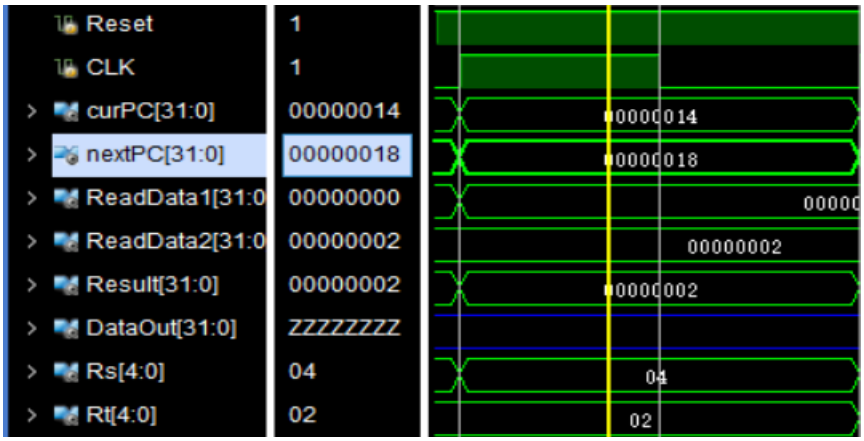
Rt为2号寄存器，对应的ReadData2的值为2，正确。（本条指令无需对寄存器Rt

写入数据）

ALU的运算结果为8&2=0，DataOut在无需读数据时输出为高阻抗，正确。

（$4=0）

逻辑或运算：



or　$8,$4,$2

当前的地址为20（14），下一条地址为24（18），正确。

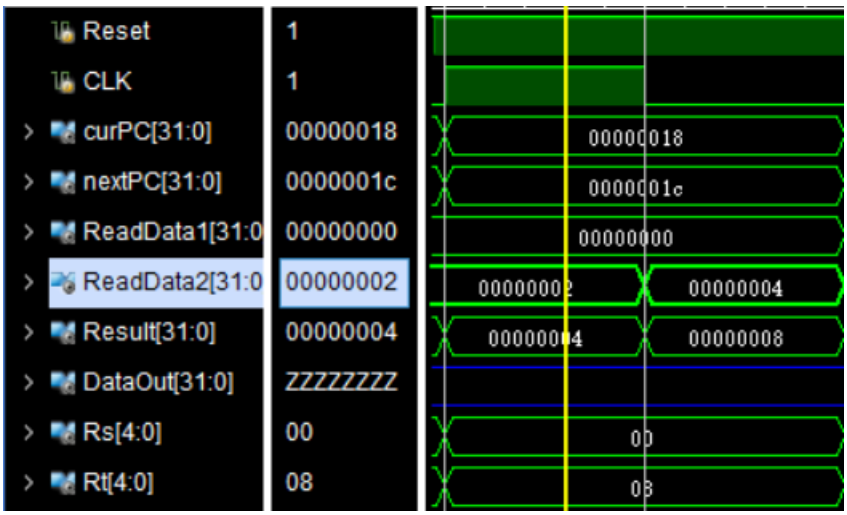
Rs为4号寄存器，对应的ReadData1的值为0，正确。

Rt为2号寄存器，对应的ReadData2的值为2，正确。（本条指令无需对寄存器Rt写入数据）

ALU的运算结果为0|2=2，DataOut在无需读数据时输出为高阻抗，正确。

（$8=2）

左移运算：



指令：sll　$8,$8,1（第二条）

当前的地址为24（18），下一条地址为28（1C），正确。

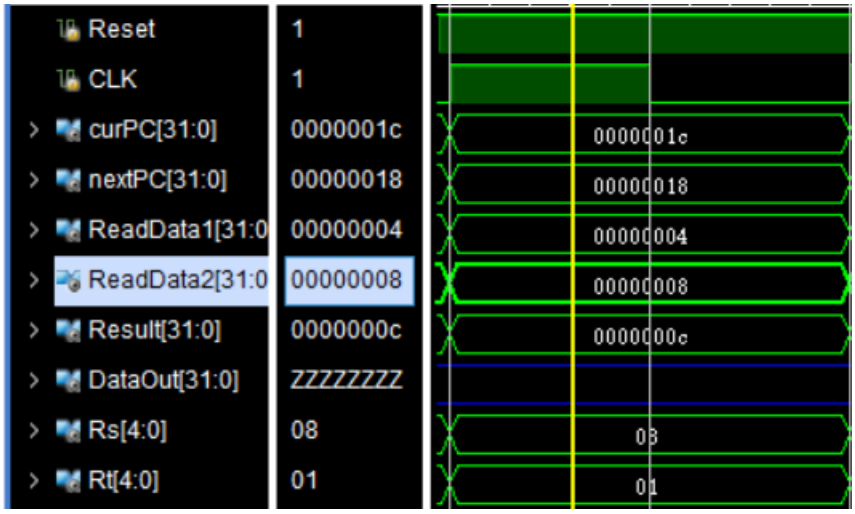
Rs为0号寄存器，对应的ReadData1的值为0，正确。

Rt为8号寄存器，对应的ReadData2的值在CLK=1时为2，在时钟的下降沿，$2=$8<<1=4，ReadData2的值变为4，正确。

ALU的运算结果为2<<1=4，DataOut在无需读数据时输出为高阻抗，正确。

($8=4)

不等时跳转：



指令：bne $8,$1,-2　(≠,转18)

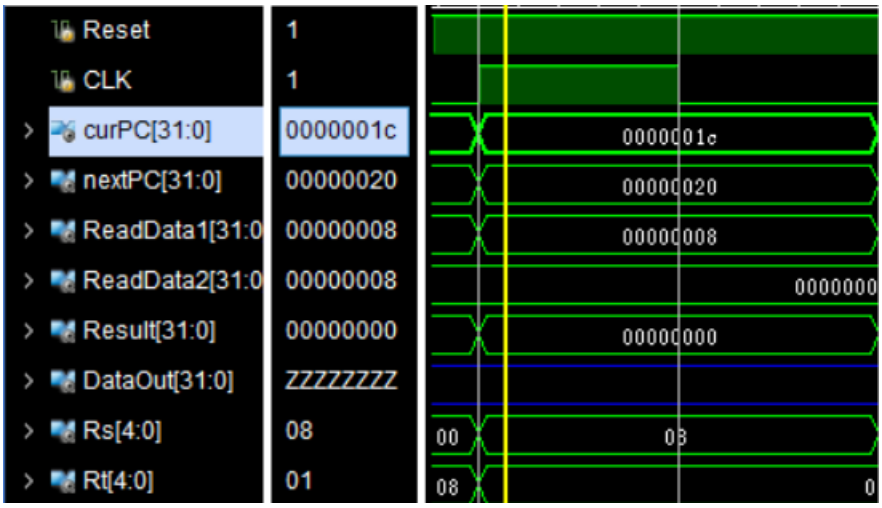
当前的地址为28（1C），由于$8=4,$1=8,两者不相等，故跳转到24（18）。下一条地址为24（18），正确。

Rs为8号寄存器，对应的ReadData1的值为4，正确。

Rt为1号寄存器，对应的ReadData2的值为8，正确

ALU的运算结果为8^4=12(C)(异或)，DataOut在无需读数据时输出为高阻抗，正确。

跳转到上一条指令后，$8=$8<<1=8,再回到这一条指令后，波形如下：



当前的地址为28（1C），由于$8=8,$1=8,两者不相等，故跳转到32（20）。下一条地址为32（20），正确。

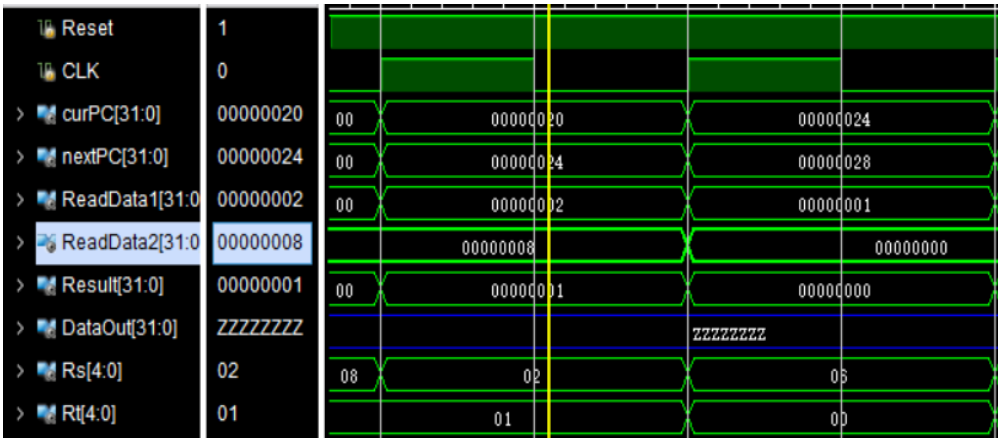
Rs为8号寄存器，对应的ReadData1的值为8，正确。

Rt为1号寄存器，对应的ReadData2的值为8，正确

ALU的运算结果为8^8=0，DataOut在无需读数据时输出为高阻抗，正确。

($8=8)

比较：



指令：slt　$6,$2,$1、slt　$7,$6,$0

当前的地址从为32（20）到36（24），下一条地址从36（24）到40（28），正确

第一条指令Rs为2号寄存器，对应的ReadData1的值为2，正确

第一条指令Rt为1号寄存器，对应的ReadData2的值为8，正确

第一条指令ALU的运算结果为1（说明2<8），DataOut在无需读数据时输出为高

阻抗，正确。此后$6=1;

第二条指令Rs为6号寄存器，对应的ReadData1的值为1，正确

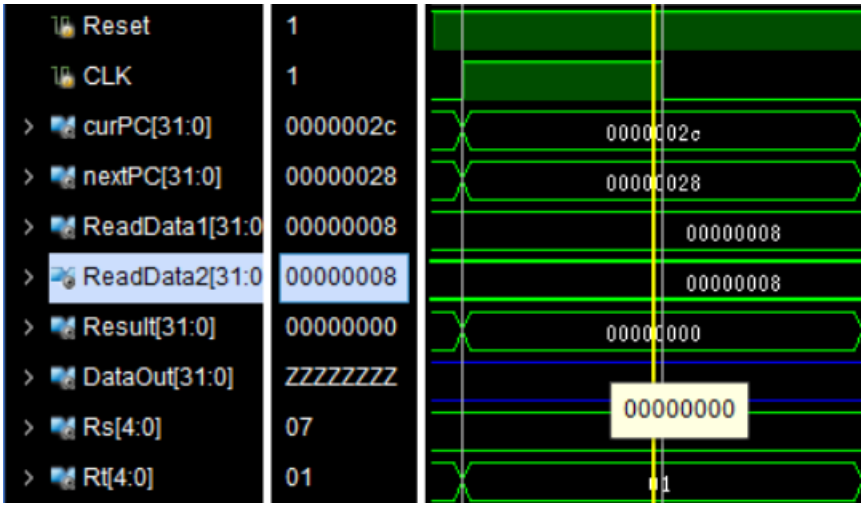
第二条指令Rt为0号寄存器，对应的ReadData2的值为0，正确

第二条指令ALU的运算结果为0（说明1>0），DataOut在无需读数据时输出为高阻抗，正确。

($6=1,$7=0)

（下一条指令为addi $7,$7,8，此后$7=8）

相等时跳转：



指令：beq $7,$1,-2 （≠,转28)

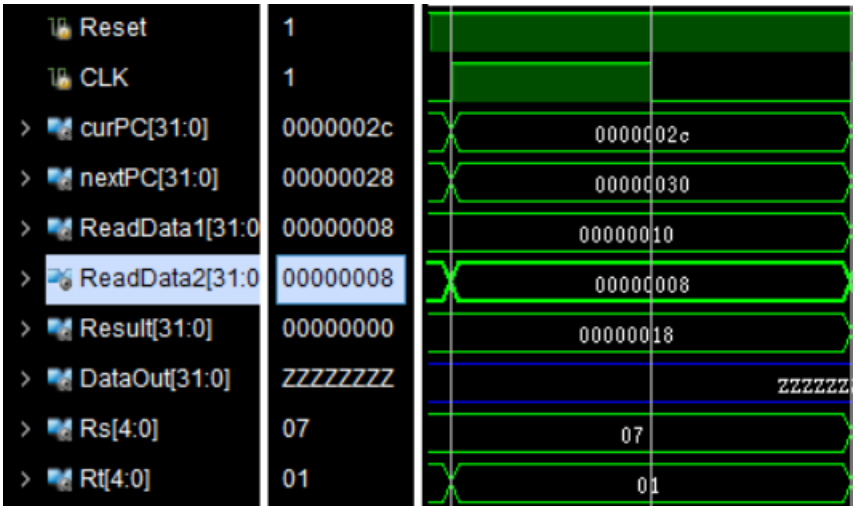
当前的地址为44（2C），由于$7=8,$1=8,两者相等，故跳转到40（28）。下一条地址为40（28），正确。

Rs为7号寄存器，对应的ReadData1的值为8，正确。

Rt为1号寄存器，对应的ReadData2的值为8，正确

ALU的运算结果为8^8=0(异或)，DataOut在无需读数据时输出为高阻抗，正确。

跳转到上一条指令（addi $7,$7,8）后，$7=$7+8=16,再回到这一条指令后，波形如下：



当前的地址为44（2C），由于$8=16（10）,$1=8,两者不相等，故跳转到48（30）。

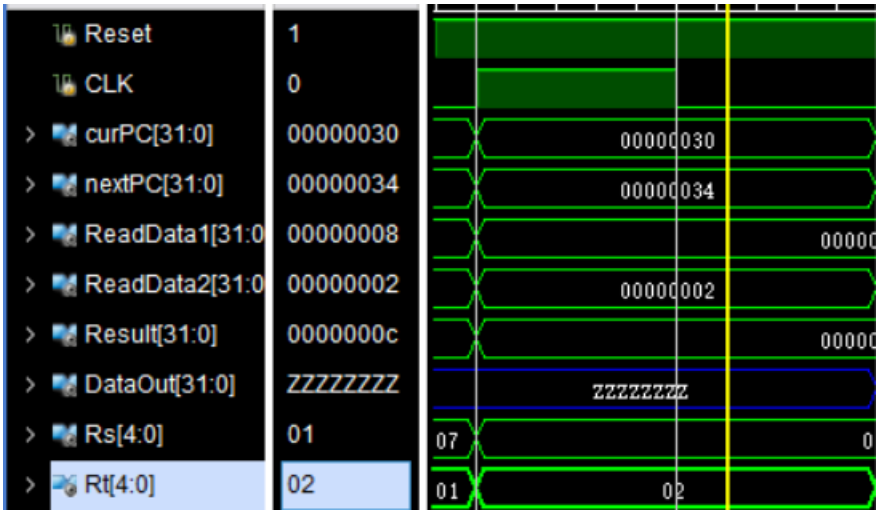
下一条地址为48（30），正确。

Rs为8号寄存器，对应的ReadData1的值为16（10），正确。

Rt为1号寄存器，对应的ReadData2的值为8，正确

ALU的运算结果为16^8=24（18），DataOut在无需读数据时输出为高阻抗，正确。

存储器写：



指令：sw　$2,4($1)

当前的地址为48（30），下一条地址为52（34），正确。

Rs为1号寄存器，对应的ReadData1的值为8，正确。

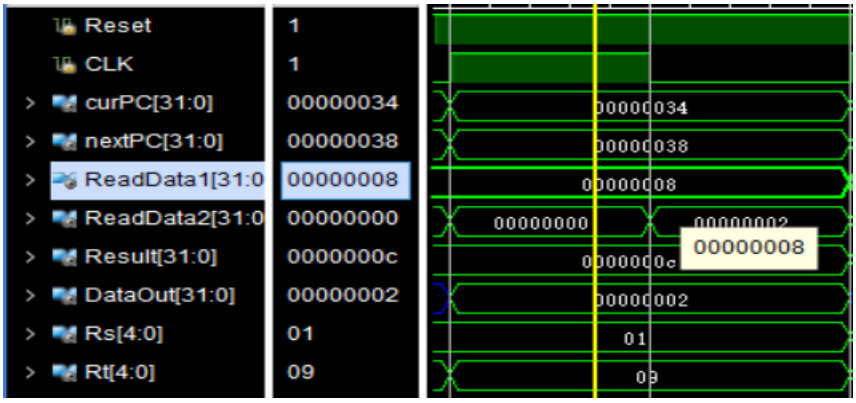
Rt为2号寄存器，对应的ReadData2的值为2，正确。

（在写指令时设置ALU的控制输入为000，此时ALU的输入A为8，输入B为立即数4）

ALU的运算结果为8+4=12(C)，DataOut在无需读数据时输出为高阻抗，正确。

（此时在数据存储器组的第12、13、14、15个存储器写入数据2）

存储器读：



指令：lw　$9,4($1)

当前的地址为52（34），下一条地址为56（38），正确。

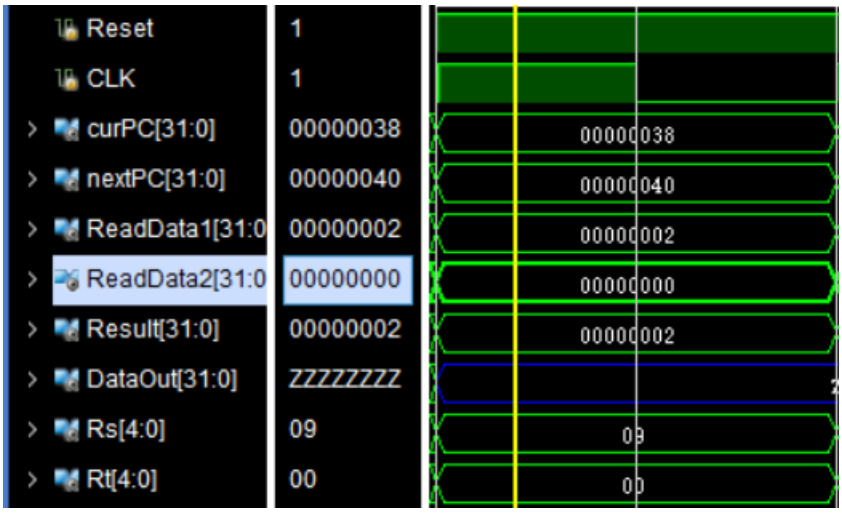
Rs为1号寄存器，对应的ReadData1的值为8，正确。

Rt为9号寄存器，对应的ReadData2的值在CLK=1时为0，在时钟的下降沿，$9=4（$1）=2，ReadData2的值变为2，正确。

（在读指令时设置ALU的控制输入为000，此时ALU的输入A为8，输入B为立即数4）

ALU的运算结果为8+4=12(C)，数据存储器此时输出4（$1）的数据2**（在上一条语句中数据存储器组的第12、13、14、15个存储器写入数据2）**,DataOut的输出为2，正确。

大于0跳转



指令：bgtz　$9,1 (>0,转 40)

当前的地址为56（38），由于$9=2>0，故跳转到64（40）。下一条地址为40（28），正确。

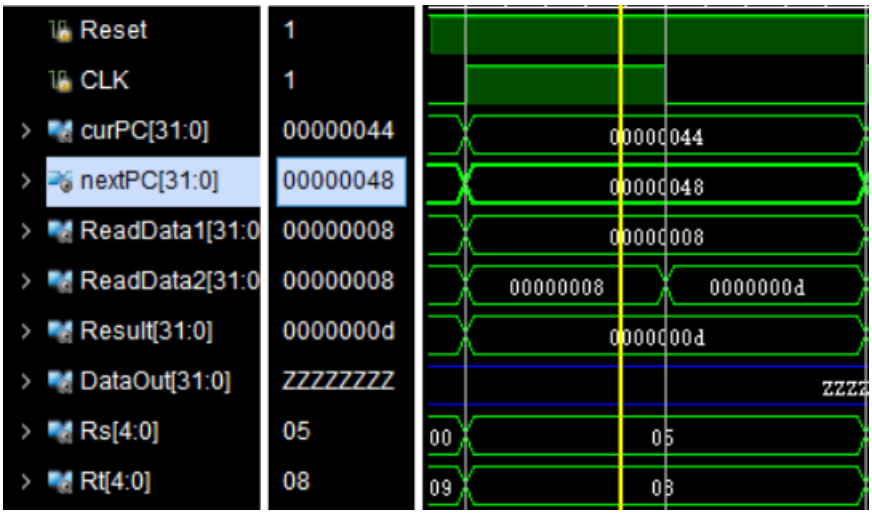
Rs为9号寄存器，对应的ReadData1的值为2，正确。

Rt为0号寄存器，对应的ReadData2的值为0，正确

ALU的运算结果为2^0=2(异或)，DataOut在无需读数据时输出为高阻抗，正确。

跳转到40对应的指令（addi　$9,$0,-1）后，$9=$0-1=-1。

立即数逻辑异或



指令：xori $8,$5,5

当前的地址为68（44），下一条地址为72（48），正确。

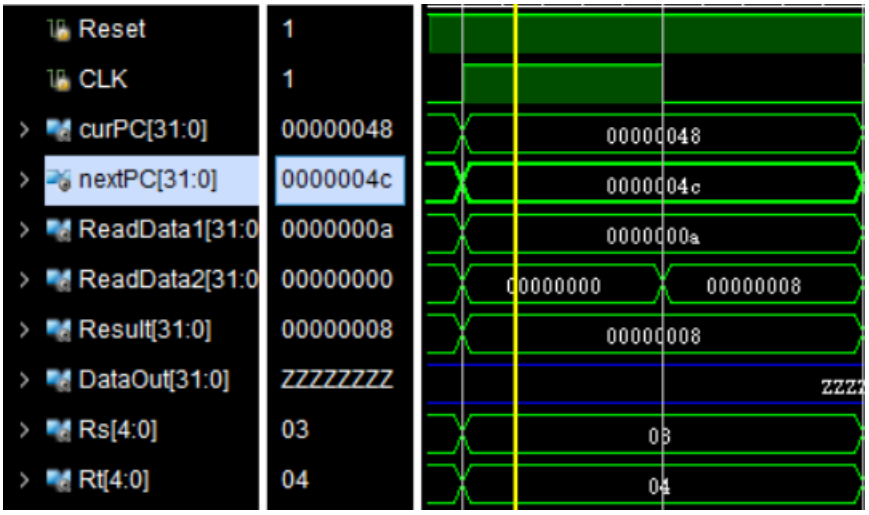
Rs为5号寄存器，对应的ReadData1的值为8，正确。

Rt为8号寄存器，对应的ReadData2的值在CLK=1时为8，在时钟的下降沿，$8=$5^5=13(d)，ReadData2的值变为13，正确。

ALU的运算结果为8^5=13，DataOut在无需读数据时输出为高阻抗，正确。

($8=13)

立即数逻辑与



指令：andi $4,$3,9

当前的地址为72（48），下一条地址为76（4c），正确。

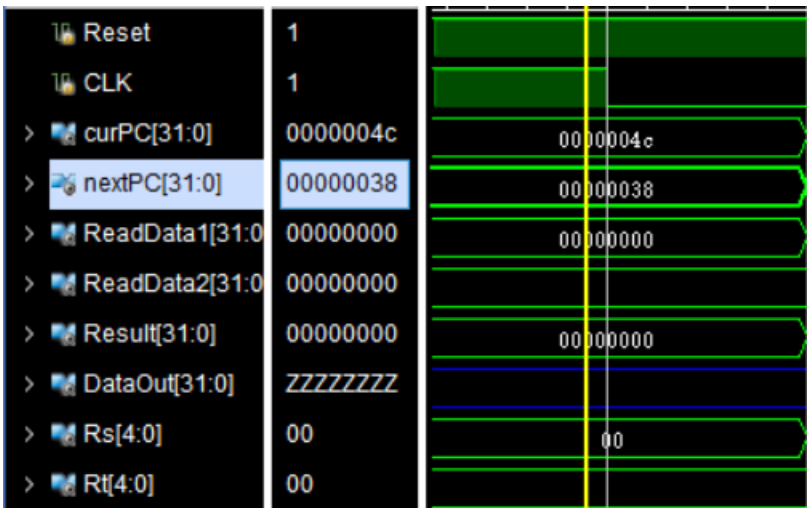
Rs为3号寄存器，对应的ReadData1的值为10(a)，正确。

Rt为4号寄存器，对应的ReadData2的值在CLK=1时为0，在时钟的下降沿，$4=$3&9=8，ReadData2的值变为8，正确。

ALU的运算结果为10&9=8，DataOut在无需读数据时输出为高阻抗，正确。

($4=8)

跳转：



指令：j　0x00000038

当前的地址为76（4c），下一条地址为56（38）（即需要跳转的地址），正确。

Rs为0号寄存器，对应的ReadData1的值为0，正确。

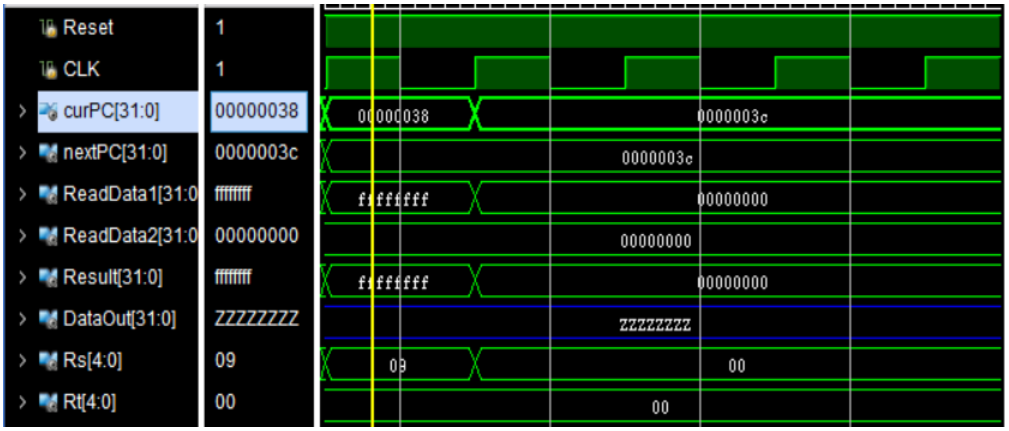
Rt为0号寄存器，对应的ReadData2的值为0，正确。

ALU的运算结果为0+0=0，DataOut在无需读数据时输出为高阻抗，正确。

($4=8)

（跳转到bgtz　$9,1即地址38对应的指令后，由于$9=-1<0，此时不会跳转到40对应的指令，而会直接到下一个地址即3c）

停机：



（可见PC从38直接到了3C）

指令：halt

当前的地址为3c（4c），此后的地址均为3c（38）（即地址不再更改），正确。

其余所有输出的值都保持为0（DataOut是高阻态）

3、 Basys3实验板上的模拟：

（1） 结果示例（每一条指令的结果图顺时针方向依次为PC:NextPC;Rs:ReadData1;Rt:ReadData2;Result:DataOut）

addi $1,$0,8

ori $2,$0,2

add $3,$2,$1

bne $8,$1,-2 (≠,转18)

sw $2,4($1)

lw　$9,4($1)

j　0x00000038

Halt

（2）　关于Basys3实验板的烧写设计：

时钟分频模块：将CLK分频成CLK190和CLK3，用CLK190作为按键模块的时钟输入，CLK3用于移位寄存器的时钟输入。

按键模块:按下按键即得到一个时钟的上升沿,将得到的时钟作为整个CPU的时钟输入。**（注意要有防抖设置，即在分配引脚处加一条语句 set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets Key_IBUF]）**

移位寄存器模块：用于处理CPU的输出，得到一个16位的输出信号。

显示模块：将移位寄存器的输出信号显示到板上

五、实验心得和体会

1、写代码时遇到过的坑：

（1）Wire和Reg不能弄混，即assign只能对Wire使用，而在initial段和always段进行赋值操作的一定是Reg;

（2）注意一个wire被assign之后，会随着其等于号右边的信号改变而改变，并不只是简单的赋值；

（3）在模块中定义reg中间变量时需要视情况对该变量进行初始化，否则在仿真中的输出会体现为XXXX;

（4）敏感信号中不能时序逻辑和组合逻辑信号混用；

（5）注意在组合电路中不要出现输出返回到输入的情况，比如a=a+4等，为了防止这种情况，需要检查代码，检查是否有if而无对应的else，有case而无defaultd等，以避免产生锁存器。

（6）在调用某个模块时，一定要注意输入和输出的顺序！一定要注意输入和输出的顺序！一定要注意输入和输出的顺序！

（7）对于posedge CLK or negedge Reset这样的敏感信号，一定要指明当Reset等于0的时候要做什么。

2、设计时遇到的坑：

（1）一开始没看要求，忘记了指令储存器和数据储存器的长度只有8位，后来才发现自己写错了。

（2）将输入输出信号的选择（比如ALU和数据存储器）杂糅进同一个模块里，造成代码重复且冗杂。

（3）一开始忽略了寄存器组和数据存储器组写指令是下降沿触发的，直接改过来之后发现，在第一条指令处寄存器组并不能直接写入数据。于是便将PC初始化为-4，于是在时钟的第一个下降沿处，寄存器组写入第一条指令的结果成功。

（4）在设计停机指令时误以为停机就是全部重置，后来才改过来。

（5）最后一个问题尚未解决：在我第一份代码处，仿真过了但一直烧不上板，连续四天的报错都是在生成比特流处，有一个关于“形成组合逻辑环”的error，检查多次代码都未能解决这个问题，最后将代码重写一次才解决了。