中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理与接口技术实验》

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 软件工程四（8）班
学 生 姓 名 ：	庄铸滔
学　　号 ：	16340320
时　　间 ：	2018　年　5　月　28　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。

二. 实验内容

实验的具体内容与要求。

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。

==>移位指令

（7）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

（8）slti　rt, rs,immediate　　带符号

011011	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

（9）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。

（10) 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

（11）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”。

（12）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。

==>跳转指令

（13）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],2{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

（14）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

简述实验原理和方法，**必须有数据通路图及相关图**。

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。

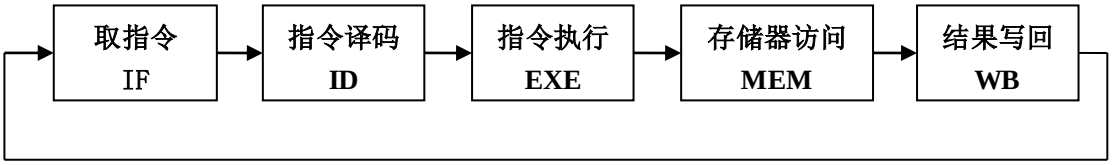
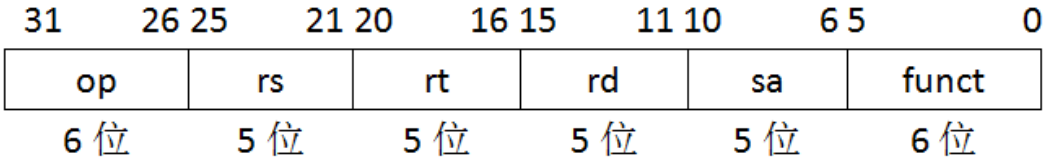


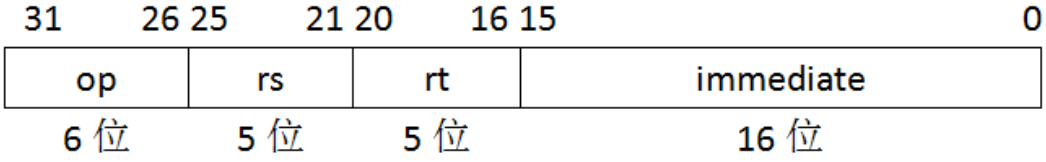
图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

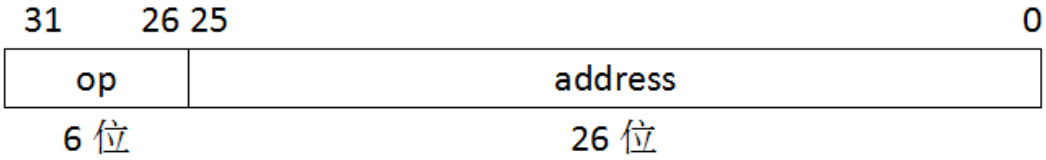
R 类型：



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。

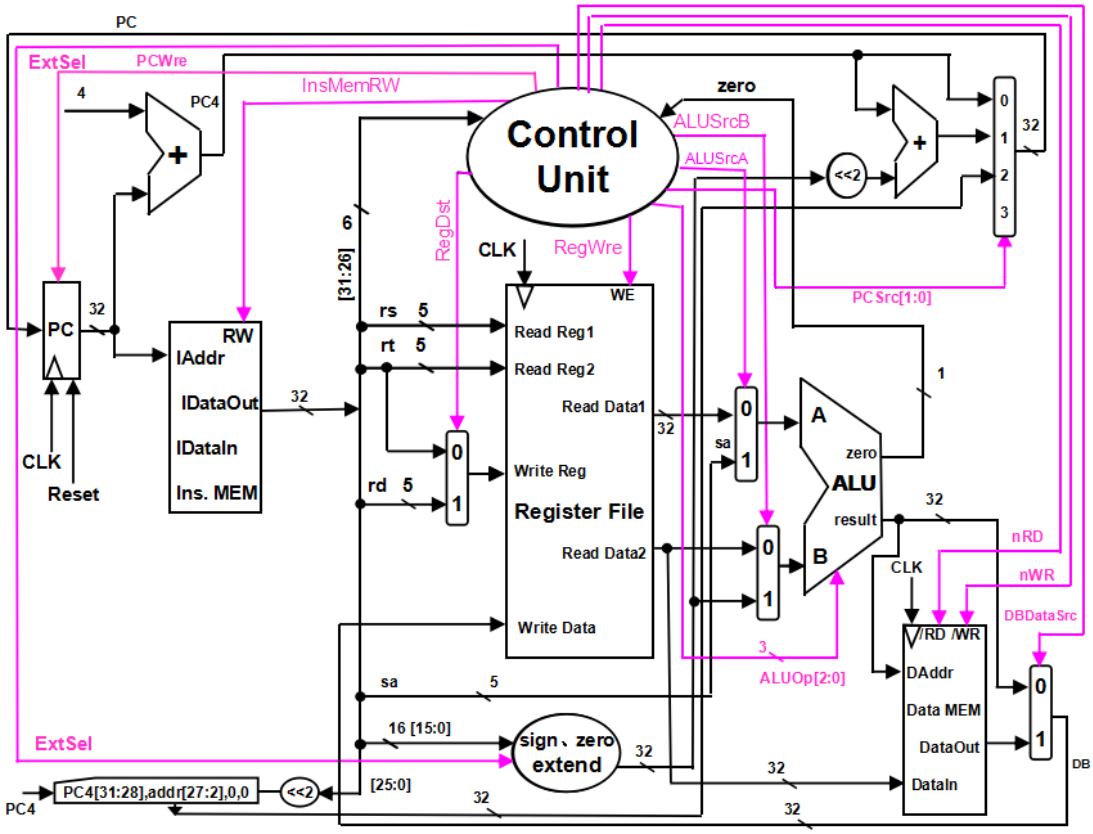


图 2　单周期 CPU 数据通路和控制线路图

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 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、addi、or、and、ori、beq、bne、slti、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、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slti、sll、lw

InsMemRW	写指令存储器	读指令存储器(Ins. Data)
nRD	输出高阻态	**读数据存储器，相关指令：lw**
nWR	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：addi、ori、lw、slti	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：ori	(sign-extend)**immediate** (符号扩展)，相关指令：addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)； 10：pc<－{(pc+4)[31:28],addr[27:2],2{0}}，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

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加

001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

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、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，是必须认真考虑的问题。

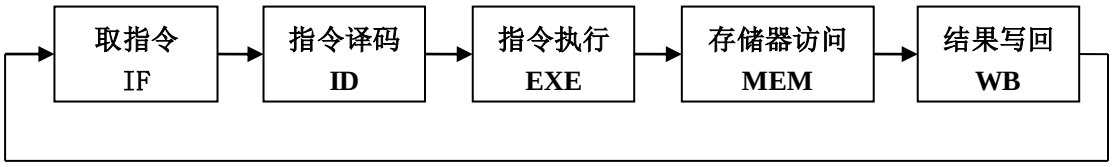
四．实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五．实验过程与结果

1． CPU设计的思想，方法

根据CPU模块化的设计方法以及数据通路图，针对CPU指令处理过程



，可以分为PC，InstructionMemory，RegisterFile，ALU，DataMemory，ControlUnit五个主要模块，下面将分别讲述五个模块的设计方法和实现原理。 在实现之前要明白的一点是：每个模块的功能都是接收输入信号，经过处理后，产生输出信号，中间的处理逻辑则需要我们根据对应的指令去设计

(1) PC:

a. 输入输出：

Input: CLK,Reset,signZeroExtend,address,PCSrc,PCWre

Output:PC

b. 模块代码及说明

PC.v

```
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;
end
```

这个小模块是在CLK为上升沿时改变PC为NextPC，输出PC值，初始化为-4，方便在始终下降沿写入数据，寄存器组

NextPc.v

```
always @( PCSrc or JumpPC or Immediate) begin
    if(Reset ==0) NextPC=PC+4;
     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
```

根据PCSrc，选择三个PC输入中的一个输出到PC中，Immediate 为立即数，PC为上一条指令地址

JumpPC.v

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

为J指令所用，25位来自指令存储器

(2) InstructionMemory

a. 输入输出

Input: InsMemRW, IDataIn, Iaddr

Output: IDataOut(表示输出PC指令地址对应的指令)

b. 模块代码及说明

InstructionMemory.v

```
initial begin
    $readmemb("C:/Users/Administrator/singleCycleCpu/CPUInsMem.txt",Memory);
    for(i=0;i<24;i=i+1)  $display("%h%h%h%h",Memory[i*4+0],Memory[i*4+1],Memory[i*4+2],Memory[i*4+3]);
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
```

接收InsMemRW表示是读指令还是写指令，在Memory中已经提前从文件中读取了要测试的若干条指令。在指令存储器中，有一个长度为8位的存储器组，其中每4个存储器存储一条指令，采用大端存储，即高位在低位，预先存储的指令如下

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	addi $1,$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008
0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010		40020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000		00411800
0x0000000C	sub $5,$3,$2	000010	00011	00010	0010 1000 0000 0000		08622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000		44A22000
0x00000014	or $8,$4,$2	010010	00100	00010	0100 0000 0000 0000		48824000
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,$2,8	011011	00010	00110	0000 0000 0000 1000		6C460008
0x00000024	slti $7,$6,0	011011	00110	00111	0000 0000 0000 0000		6CC70000
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000		04E70008

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	**j 0x00000040**	111000	00000	00000	0000 0000 0100 0000		E0000040
0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010		040A000A
0x00000040	**halt**	111111	00000	00000	0000000000000000	=	FC000000

(3) ControlUnit.v

a. 输入输出

Input: zero,Op(来自指令存储器)

Output: Reset, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, nRD, nWR, RegDst, ExtSel, PCSrc[1..0], ALUOp[2..0],

其中各个输出的值与表一控制信号对应

//输入输出真值表

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、addi、or、and、ori、beq、bne、slti、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、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
nRD	输出高阻态	**读数据存储器，相关指令：lw**
nWR	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：addi、ori、lw、slti	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：ori	(sign-extend)**immediate** (符号扩展)，相关指令：addi、slti、sw、lw、beq、

		bne
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)； 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)； 10：pc<－{(pc+4)[31:28],addr[27:2],2{0}}，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	

b. **模块代码及其说明**

```
assign PCWre = (Op == 6'b111111)?0:1;
assign ALUSrcA = (Op== 6'b011000)?1:0;
assign ALUSrcB = ((Op == 6'b000001) || (Op == 6'b010000) || (Op == 6'b011011) || (Op ==6'b100110) || (Op == 6'b100111) ) ? 1:0;
assign DBDataSrc = (Op == 6'b100111)?1:0;
assign RegWre = ((Op == 6'b110000) || (Op == 6'b110001) || (Op == 6'b100110) || (Op == 6'b111111) ||(Op == 6'b111000)) ? 0: 1;
assign InsMemRW = 0;
assign mRD = (Op == 6'b100111) ?1:0;
assign mWR  = (Op == 6'b100110) ?1:0;
assign RegDst = ((Op == 6'b000001) || (Op==6'b010000) || (Op == 6'b100111) || (Op==6'b011011))?0:1;
assign ExtSel = (Op == 6'b010000)?0:1;
assign PCSrc[0] = ((Op == 6'b110000 && zero == 1) || (Op == 6'b110001 && zero == 0) || (Op==6'b111111)) ?1:0;
assign PCSrc[1] = ((Op == 6'b111000) || (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) ||(
//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'b110000) ||(Op==6'b110001) ||(Op==6'b0110
//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'
assign ALUOp[0] =((Op == 6'b000010) || (Op == 6'b010000) ||(Op == 6'b010010) ||(Op == 6'b010100) ||(Op == 6'b010101) || (Op==6'b110000) ||(Op==6'b110001))?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;
endmodule
```

（4） RegisterFile.v

a. **输入输出**

Input:ReadReg1(rs),ReadReg2(rt),WriteReg(rt,rd)，WriteData，CLK，WE

Output: ReadData1(第一个操作数) ,ReadData2（第二个操作数）

Register 表示寄存器中的32个寄存器，每一个都是32位长，且初始化为0，输出为ReadReg1和ReadReg2对应寄存器的值

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

b. **模块代码和说明**

```
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 (WE ==1 && WriteReg!=0) Register[WriteReg]<=WriteData;
end
endmodule
```

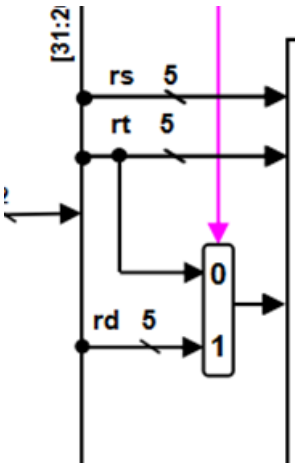
(5) Select

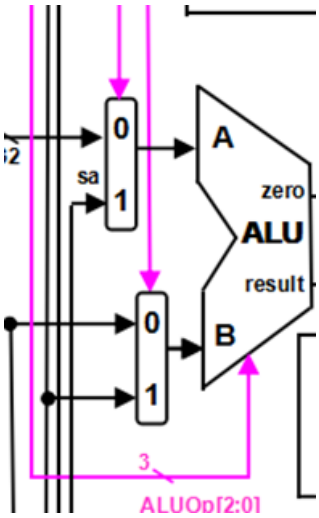
(a) 输入输出：

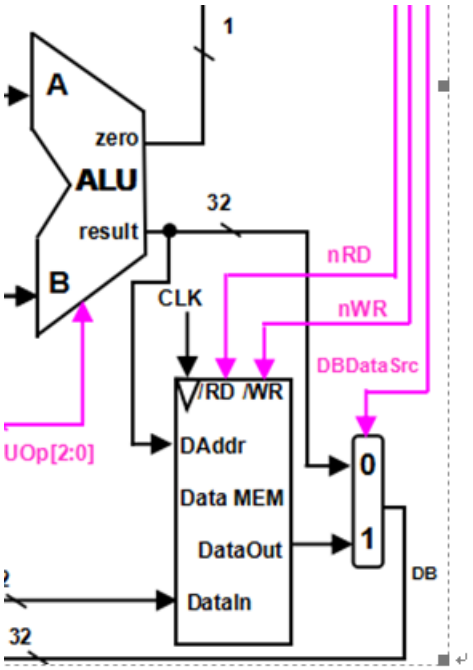
Input :DataA,DataB, select,

Output:out

选择器用于在两个数据中进行选择，5用于RegDst选择，32用于ALUSrcA,ALUSrcB, DBDataSrc选择







(b) 代码

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

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

(6) ALU.v

Op	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	InsMemRW	RegWre	/RD	/WR	RegDst	Ext Sel
000000(add)	1	0	0	0	0	1	0	0	1	0
000001(addi)	1	0	1	0	0	1	0	0	0	1

000010(sub)	1	0	0	0	0	1	0	0	1	0
010000(ori)	1	0	1	0	0	1	0	0	0	0
010001(and)	1	0	0	0	0	1	0	0	1	0
010010(or)	1	0	0	0	0	1	0	0	1	0
010011(andi)	1	0	1	0	0	1	0	0	0	0
010100(xor)	1	0	0	0	0	1	0	0	1	0
010101(xori)	1	0	1	0	0	1	0	0	0	0
011000(sll)	1	1	0	0	0	1	0	0	1	0
011011(slti)	1	0	1	0	0	1	0	0	0	1
011100(slt)	1	0	0	0	0	1	0	0	1	0
100110(sw)	1	0	1	0	0	0	0	1	1	1
100111(lw)	1	0	1	1	0	1	1	0	0	1
110000(beq)	1	0	0	0	0	0	0	0	1	1
110001(bne)	1	0	0	0	0	0	0	0	1	1
110010(bgtz)	1	0	0	0	0	0	0	0	1	1
111000(j)	1	0	0	0	0	0	0	0	1	0
111111(halt)	0	0	0	0	0	0	0	0	1	0

Op	PCSrc	ALUOP
000000(add)	00	000
000001(addi)	00	000
000010(sub)	00	001
010000(ori)	00	011
010001(and)	00	100
010010(or)	00	011
010011(andi)	00	100
010100(xor)	00	111

010101(xori)	00	111
011000(sll)	00	010
011011(slti)	00	110
011100(slt)	00	110
100110(sw)	00	000
100111(lw)	00	000
110000(beq)	00(zero=0) 01(zero=1)	111
110001(bne)	01(zero=0) 00(zero=1)	111
110010(bgtz)	00(zero=1) 01(zero=0)	111
111000(j)	10	000
111111(halt)	00	000

(a) 输入输出

Input:InA,InB,ALUOp[2:0]

Output: zero, result

(b) 代码说明

根据控制单元输出的ALUOp的值，对InA,InB做不同的运算，然后输出结果，

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

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

(7) DataMemory.v

(a) 输入输出

Input：CLK,RD,WR,DAddr,DataIn

Output：DataOut

nRD	输出高阻态	读数据存储器，相关指令：lw
nWR	无操作	写数据存储器，相关指令：sw

有一个长度为8位的存储器，用来在内存中存储数据，注意是在时钟下降沿触发的

(b) 代码说明

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

//由于MIPS存储器采用大端字节寻址，大端存储，高位放在低位
always @(negedge CLK)
begin
    if (WR==1) 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
```

(8) signzeroExtend.v

(a) 输入输出

Input:ExtSel,Immediate

Output:ExtOut

当ExtSel =1表示符号扩展时，则要考虑到立即数的最高位，从而进行符号扩展。

(b) 代码说明

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

assign ExOut[15:0] = Immediate[15:0];
assign ExOut[31:16] = (ExtSel == 1 && (Immediate[15]==1))?16'hFFFF:16'h0000;
endmodule
```

(9)singleCycleCPU 顶层模块

此模块用来将各个小模块连接起来组合成整个CPU，执行顺序如下

PC->Instruction->ControlUnit->RegisterFile->ALU->DataMemory

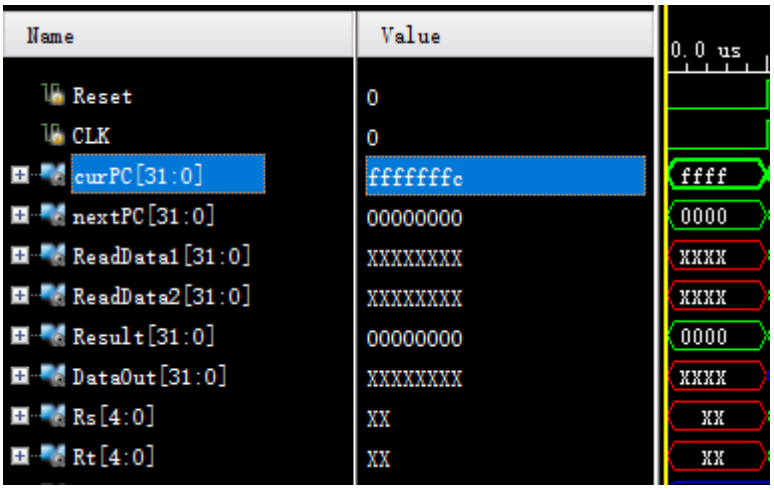
2. 仿真情况和说明

(1) 时钟代码

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

(2)

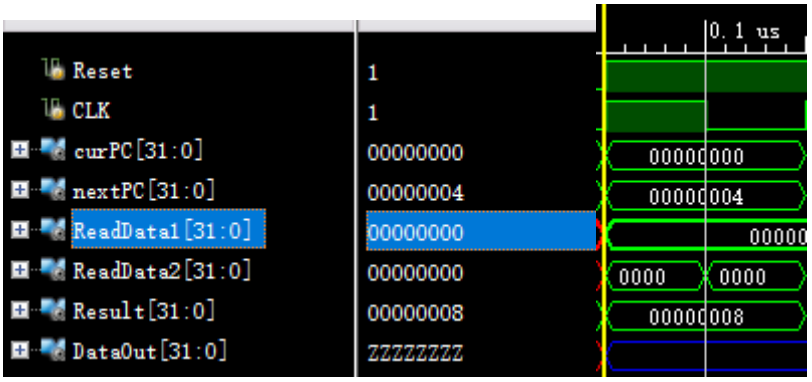
初始，当前为-4，下一条为0



addi $1 ,$0,8

当前指令为0，下一条为4，rs为0号寄存器，rt为1号寄存器

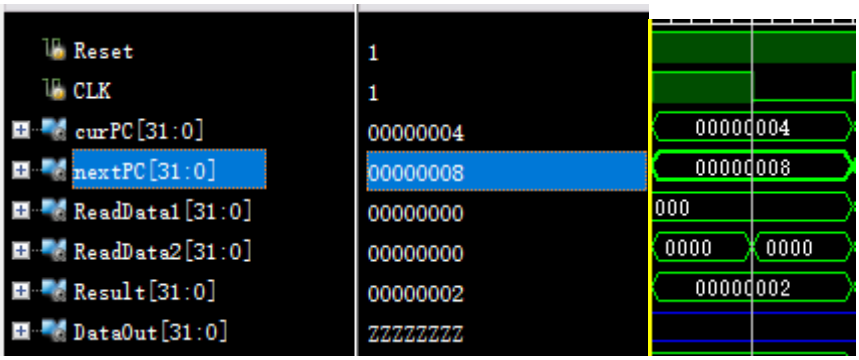
$1 = 8



ori　$2,$0,2

指令正确为4，rs为0号寄存器，rt为2号寄存器，ALU的运算结果为0|2 = 2，result=2，ReadData2 为2，表示$2的值，正确，DataOut在无需读数据时为高阻抗，正确

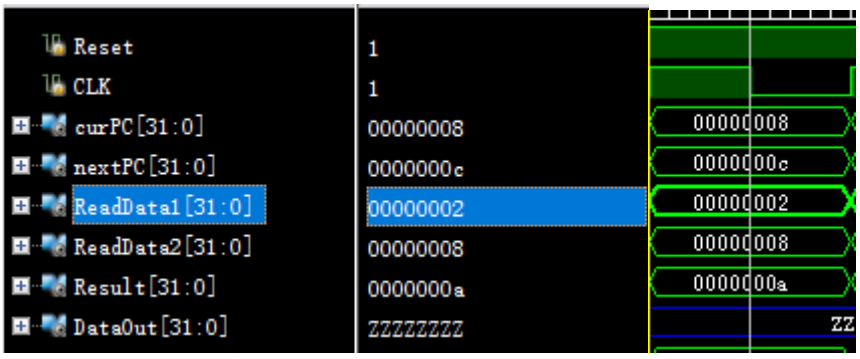
$2=2



add　$3,$2,$1

指令为8，正确，rs为二号寄存器，对应ReadData1为2，正确，rt为1号寄存器，对应ReadData2为8，正确，ALU运算结果8+2 =10，DataOut高阻抗，正确

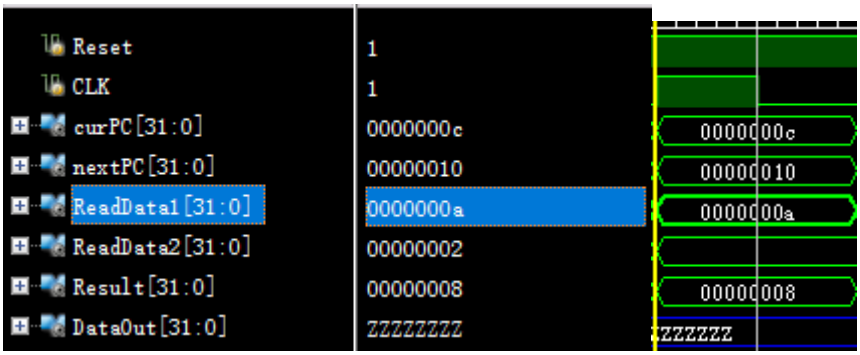
$3=10



sub　$5,$3,$2

指令为12正确，rs为3号寄存器，ReadData1值为10正确，rt为2号寄存器，ReadData2的值为2，正确，ALU 10-2=8

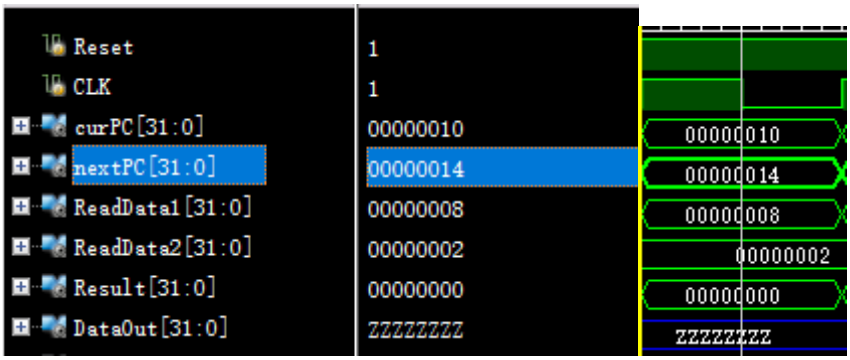
$5=8



and $4,$5,$2

指令为16，正确，rs为5号寄存器，ReadData1的值为8，正确，rt为2号寄存器,ReadData2的值为2，正确，8&2=0，result=0正确

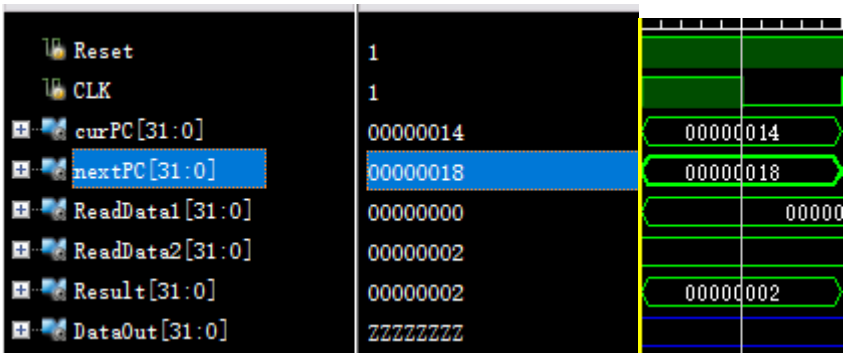
$4=0



or $8,$4,$2

指令为20（14），正确，rs为4号寄存器，ReadData1的值为0，rt为2号寄存器,ReadData2的值为2，result = 2 0|2=2，正确

$8=2



sll　$8,$8,1

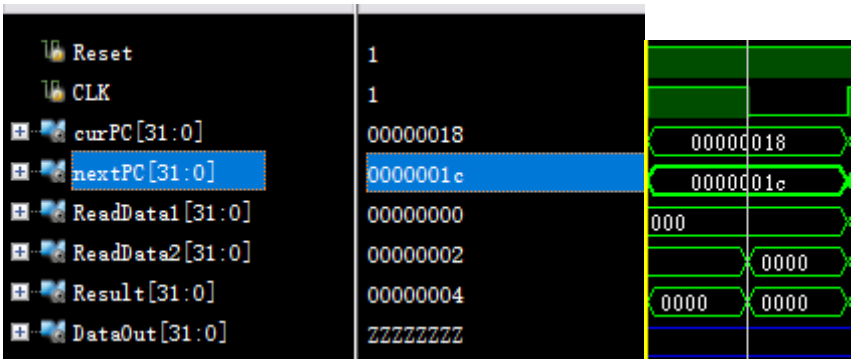
指令：sll　$8,$8,1（第二条）

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)



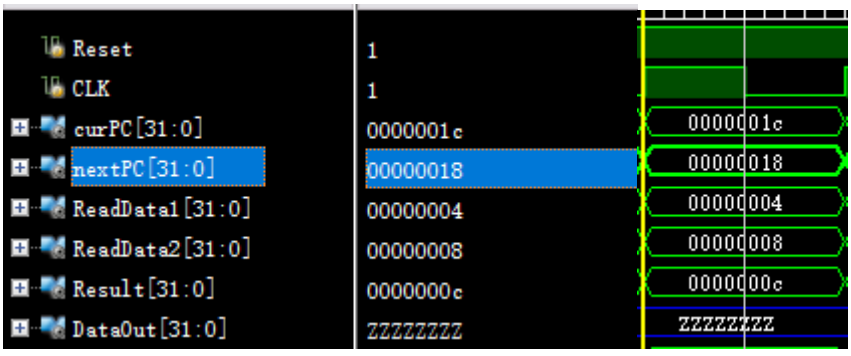
bne　$8,$1,-2 (≠,转18)

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。

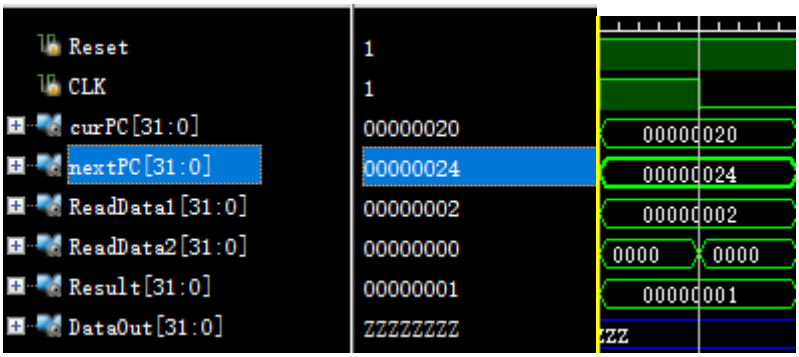
Rt为1号寄存器，对应的ReadData2的值为8，正确

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。



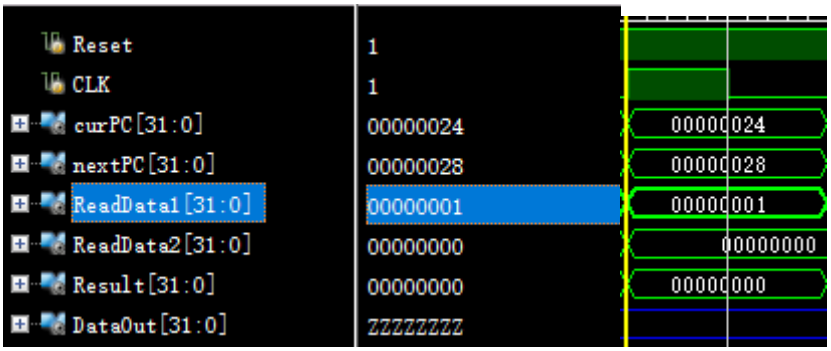
slti　$6,$2,8

指令为32 ，rt = $6, rs=$2,immediate=8 2<8,result =1正确，$6=1



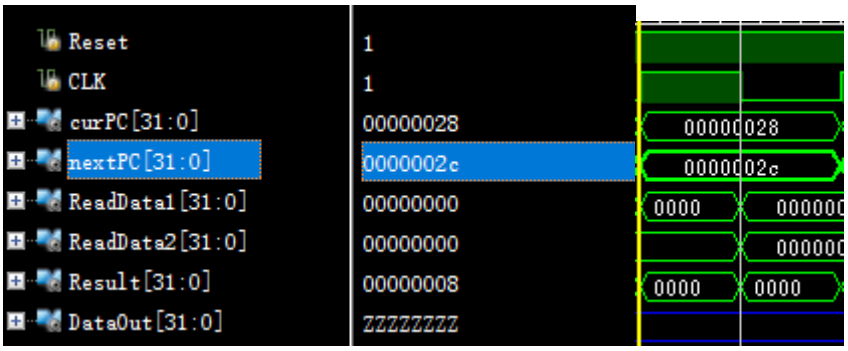
slti　$7,$6,0

指令为36，rt=$7,rs=$6,immediate=0, 1>0 ,result =0,正确，$7=0



addi $7,$7,8

指令为40，rt=$7 rs=$7,immediate =8 reuslt=8，正确 $7=8

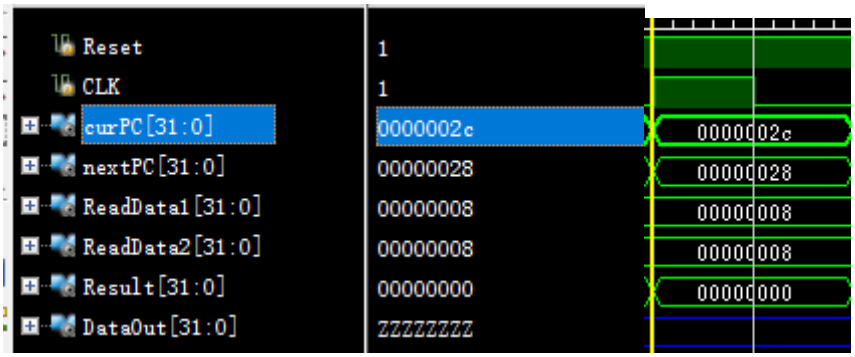


beq $7,$1,-2 (=,转28)

指令为44，rs=$7 rt=$1, immediate=-2 result=0，异或，相等，则跳转回上一条指令，

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：

两者不等，则跳转下一条指令，



sw　$2,4($1)

当前的地址为48（30），下一条地址为52（34），正确。

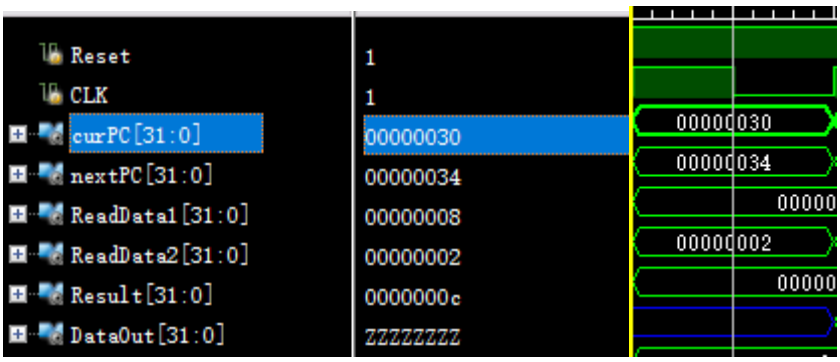
Rs为1号寄存器，对应的ReadData1的值为8，正确。

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）



lw　$9,4($1)

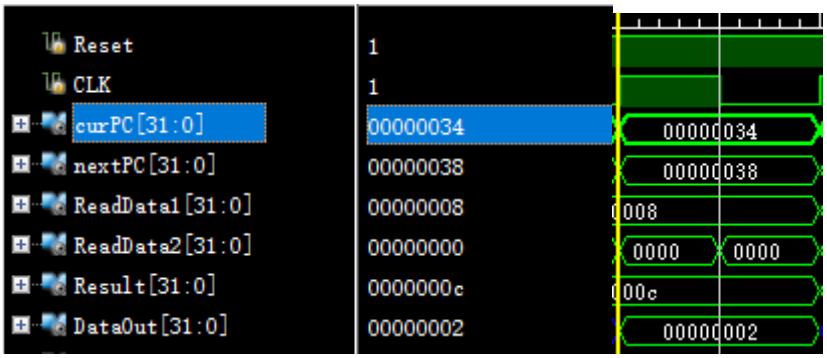
当前的地址为52（34），下一条地址为56（38），正确。

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。

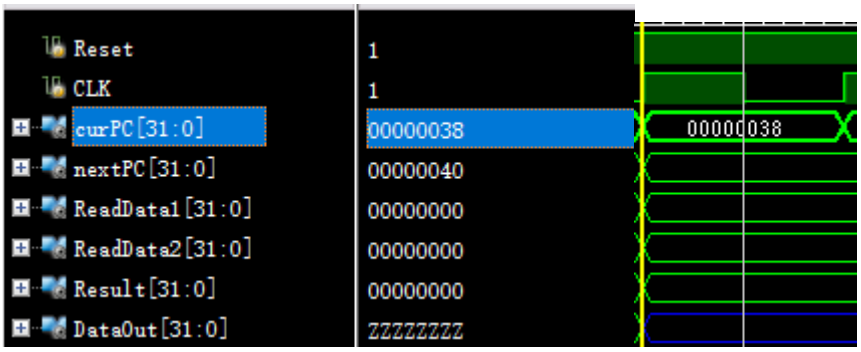
（在读指令时设置ALU的控制输入为000，此时ALU的输入A为8，输入B为立即数4）

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。



j　0x00000040

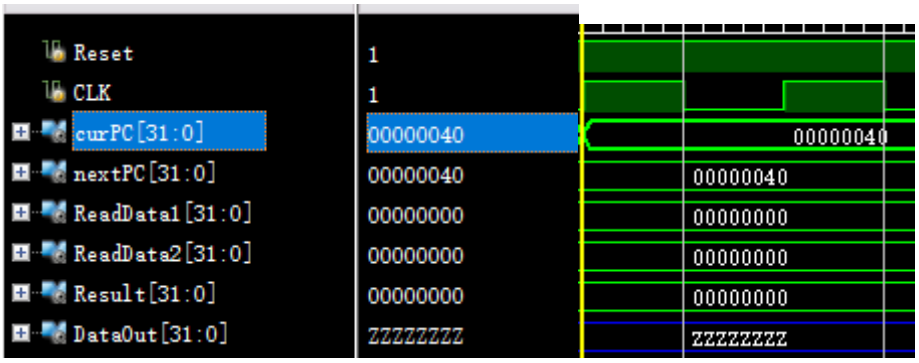
当前指令为56，下一条为64，正确，表示直接跳转halt指令



addi　$10,$0,10

halt

所有值都为0，停机指令



六. 实验心得

体会和建议。

1.模块化的设计思想很有用，可以将任务小部分的分开，在遇到错误的时候只需查找对应模块即可，帮助我们更好的debug

2. 注意wire 和reg的不同，wire的值可以紧跟输入的值改变而改变

3. reg 只能在initial中赋值初始化

4. 注意MIPS是大端存储，即高位在低位，设计的时候要考虑到这点

5. 注意指令存储器和数据存储器的长度为8位

6. 寄存器组和数据存储器都是下降沿触发的,初始化不位ffffffc

，则不能直接写入数据，在时钟第一个下降沿处，才能写入寄存器指令

7. 注意在CPUInsMem.txt中 ，每一条指令每8位要有空格，这样才能正确读入到指令存储器中

8.要根据数据通路图来连线，注意实例化各个小模块的先后顺序和线的连接顺序