《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 3 班
学 生 姓 名 ：	郑康泽
学 号 ：	17341213
时 间 ：	2018 年 11 月 20 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

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt。

（3）addi rt , rs ,**immediate**

000010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 做符号扩展再参加“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 & (sign-extend)**immediate；immediate** 做符号扩展再参加“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 **|** (sign-extend)**immediate；immediate** 做符号扩展再参加“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。

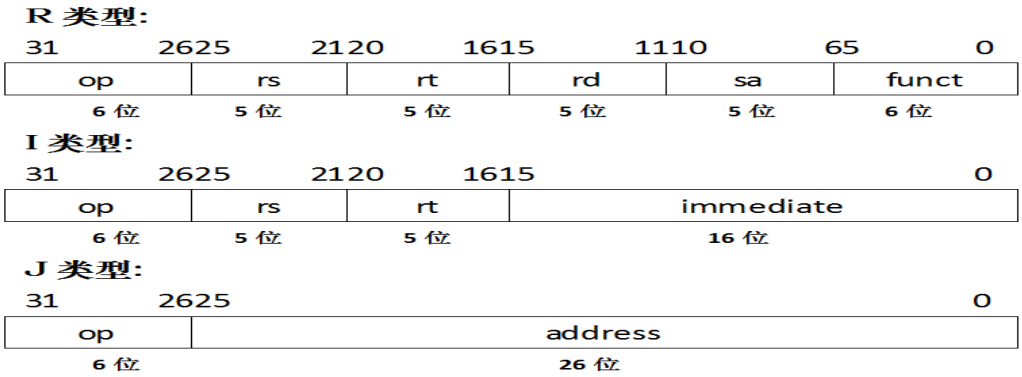
(4) 存储器访问(**MEM**)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(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。

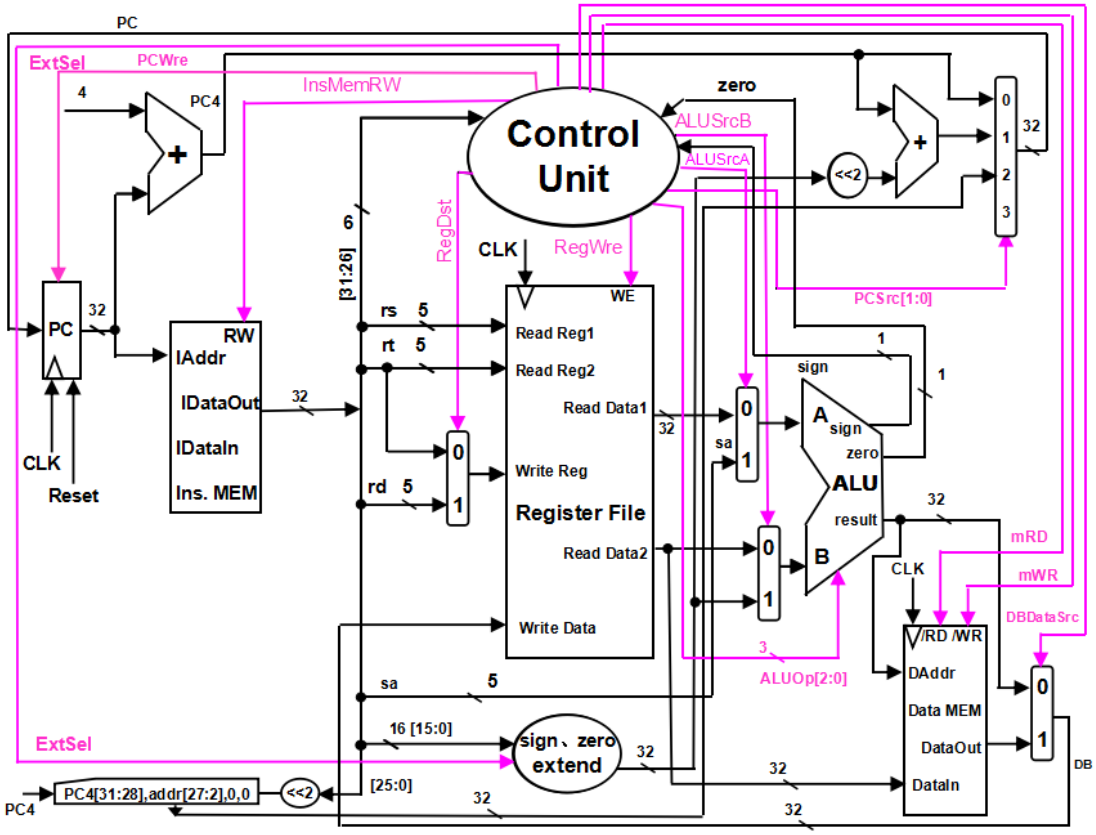


图 2　单周期 CPU 数据通路和控制线路图

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、addi、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、addi、sub、ori、or、and、andi、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addi、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：addi、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：sll	(sign-extend)**immediate（符号扩展）**，相关指令：slti、sw、lw、beq、bne、bltz 、addi、andi、ori
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0); 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1); 10：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，相关指令：j; 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

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

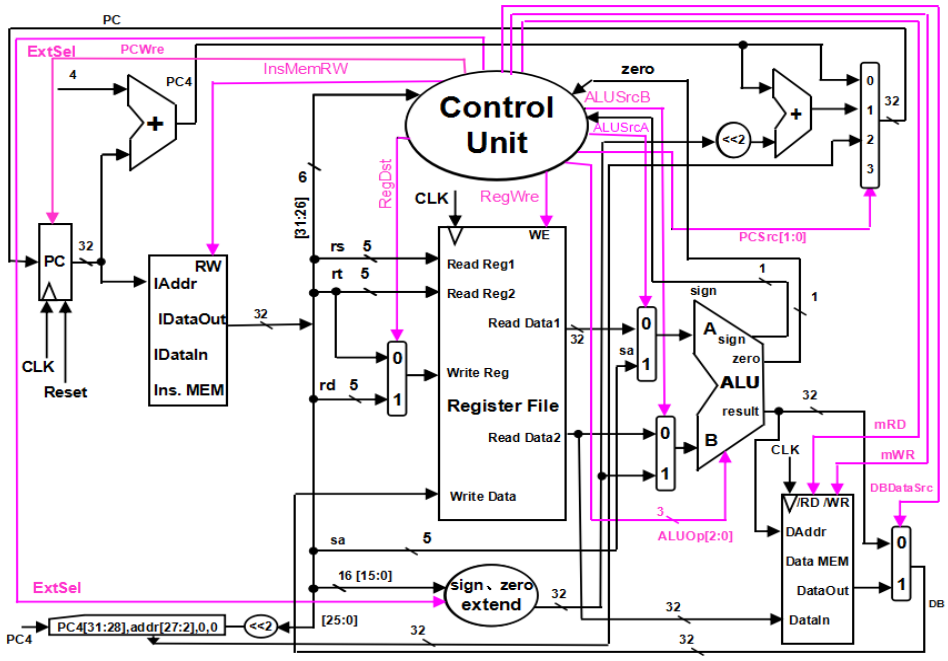
ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A – B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y=（A<B）?1: 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

四、实验设备

PC 机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

五、实验过程与结果

1、



（1）CPU设计的思想与方法在这张图表现得很清楚，就是各个模块各自发挥作用，完成自己的工作，基本流程是：PC在时钟的上升沿到时，将将执行的指令的地址发送给指令存储器，指令存储器将指令取出后，将指令的各个部分发送给各个模块，例如将前6位发送给控制单元，然后控制单元发出相关的信号，指导各个模块执行相应操作；又将21~25位、16~20位、11~15位分别发送寄存器组、寄存器组和写寄存器复选器、写寄存器复选器；将0~15位发送给（无）符号位扩展器等等，通过寄存器读出的数据或者（无）符号位扩展器输出的数据送给ALU，计算出结果，并且写进寄存器的数据是来自ALU计算的结果还是从存储器读出的数据，这也是控制单元的控制信号决定，还有下一条指令的地址也是控制信号决定的。这大概是CPU的工作流程。

（2）控制信号表的作用就是指导各个模块应如何工作，应该应用到各个模块的输入去。

	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	mRD	mWR	RegDst	ExtSel	PCSrc	ALUOp
add	1	0	0	0	1	1	0	0	1	X	00	000
sub	1	0	0	0	1	1	0	0	1	X	00	001
addi	1	0	1	0	1	1	0	0	0	1	00	000
andi	1	0	1	0	1	1	0	0	0	1	00	100
and	1	0	0	0	1	1	0	0	1	X	00	100
ori	1	0	1	0	1	1	0	0	0	1	00	011
or	1	0	1	0	1	1	0	0	1	X	00	011
sll	1	1	0	0	1	1	0	0	1	0	00	010
slti	1	0	1	0	1	1	0	0	0	1	00	110
sw	1	0	1	X	0	1	0	0	X	1	00	000
lw	1	0	1	1	1	1	1	1	0	1	00	000
beq	1	0	0	X	0	1	0	0	X	1	01	000
bne	1	0	0	X	0	1	0	0	X	1	01	000
bltz	1	0	X	X	0	1	0	0	X	1	01	000
j	1	X	X	X	0	1	0	0	X	1	10	XXX
halt	0	X	X	X	0	0	0	0	X	X	11	XXX

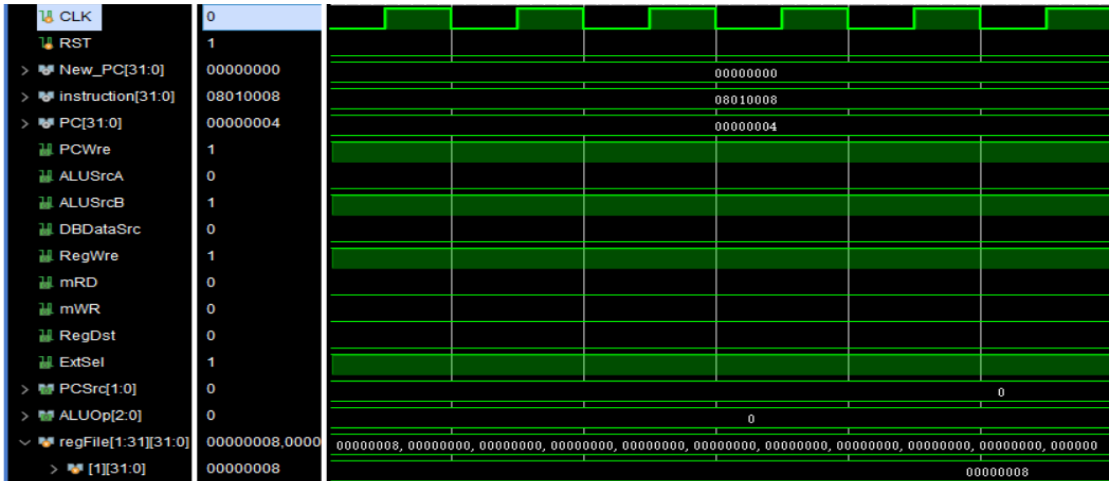
（3）相关模块及代码：

```
module PC1(
    input PCWre,                // PC输出可改控制信号
    input CLK,                  // 时钟信号
    input RST,                  // 重置信号
    input [31:0] Ins_in,        // 下一条指令的地址
    output reg[31:0] Ins_out    // 当前指令的地址
    );
module Mul_PCSrc(
    input [1:0] PCSrc,          // 获得下一条指令地址的方式
    input [31:0] PC,            // 当前指令的地址
    input [31:0] Input1,        //   当前指令的第0~第15位扩展成32位
```

```
    input [25:0] Input2,                //   当前指令的第0~第25位
    output reg [31:0] NewPC             //   下一条指令的地址
    );
module Extend(
    input ExtSel,                   // 扩展方式
    input [15:0] data_in,               // 当前指令的第0~第15位
    output reg [31:0] data_out  //   扩展成32位的结果
    );
module Ins_Memory(
    input [31:0] Ins_addr,          // 指令的地址
    output reg [31:0] Ins_data_out   // 指令
    );
module Control_Unit(
    input [5:0] opcode,                 // 当前指令的前6位
    input zero,                         // ALU计算结果是否为0
    input sign,                         // ALU计算结果为正为负
    output reg PCWre,                   // PC可改信号
    output reg ALUSrcA,             // ALU第一个操作数选择信号
    output reg ALUSrcB,             // ALU第二个操作数选择信号
    output reg DBDataSrc,           // 写寄入存器的来源选择信号
    output reg RegWre,                  // 寄存器可写信号
    output reg mRD,                 // 存储器可读信号
    output reg mWR,                 // 存储器可写信号
    output reg RegDst,                  // 目的寄存器的选择信号
    output reg ExtSel,              // 扩展方式选择信号
    output reg [1:0] PCSrc,         // PC改变方式选择信号
    output reg [2:0] ALUOp              // ALU计算方式选择信号
    );
module RegFile(
    input CLK,                          // 时钟信号
```

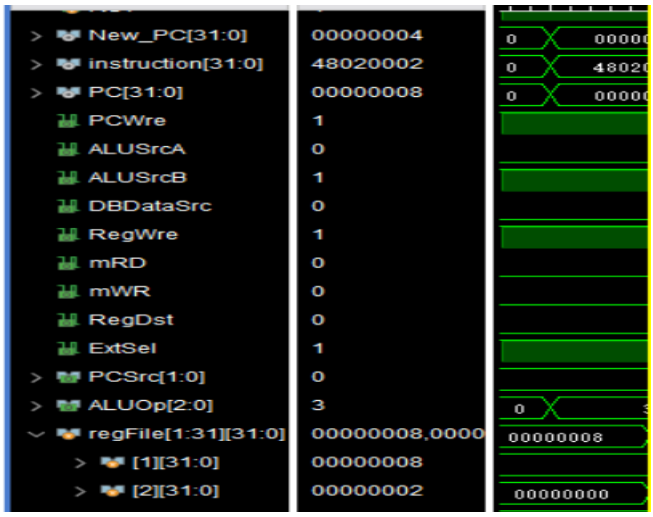
```
    input RegWre,                   // 寄存器写信号
    input [4:0] ReadReg1,       // rs寄存器
    input [4:0] ReadReg2,       // rd寄存器
    input [4:0] WriteReg,       // 目的寄存器
    input [31:0] WriteData,     // 写寄存器的数据
    output [31:0] ReadData1,    // rs寄存器数据
    output [31:0] ReadData2     // rd寄存器数据
    );
module ALU32(
    input [2:0] ALUopcode,      // ALU计算方式选择
    input [31:0] rega,          // ALU第一个操作数
    input [31:0] regb,          // ALU第二个操作数
    output reg[31:0] result,    // ALU计算结果
    output zero,                    // 结果是否为0
    output sign                     // 结果为正为负
    );
    module Data_Memory(
        input CLK,                  // 时钟
        input [31:0] address,   // 访问地址
        input [31:0] writeData, // 写入存储器数据
        input mRD,                  // 可读信号
        input mWR,                  // 可写信号
        output [31:0] Dataout   // 读存储器的数据
        );
```

2、 第一条指令：addi　$1,$0,8



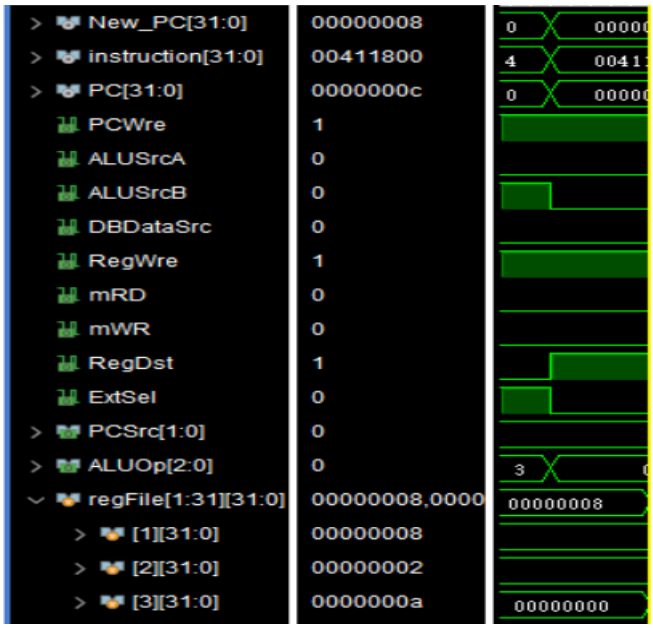
从上到下依次是时钟、reset信号、当前PC、当前指令的机器代码、下一条PC、各个信号、以及目的寄存器的值（0+8=8）

第二条指令：ori　$2,$0,2



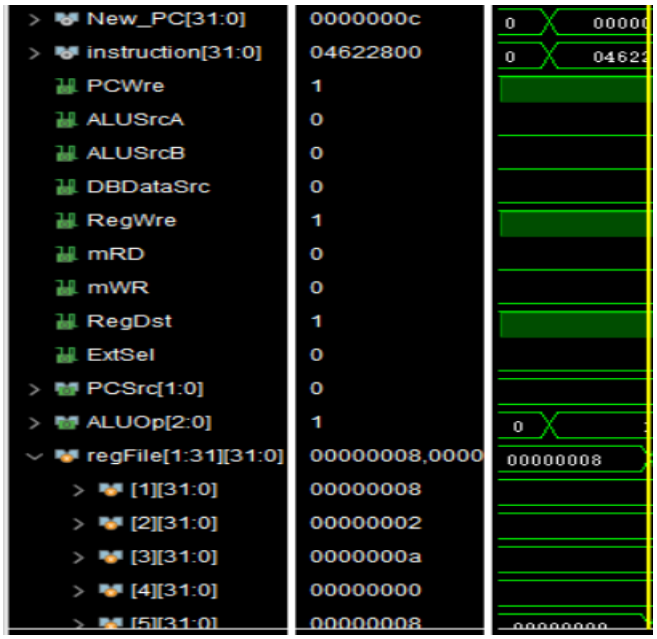
因为2|0 = 2，所以2号寄存器值为2

第三条指令：add　$3,$2,$1



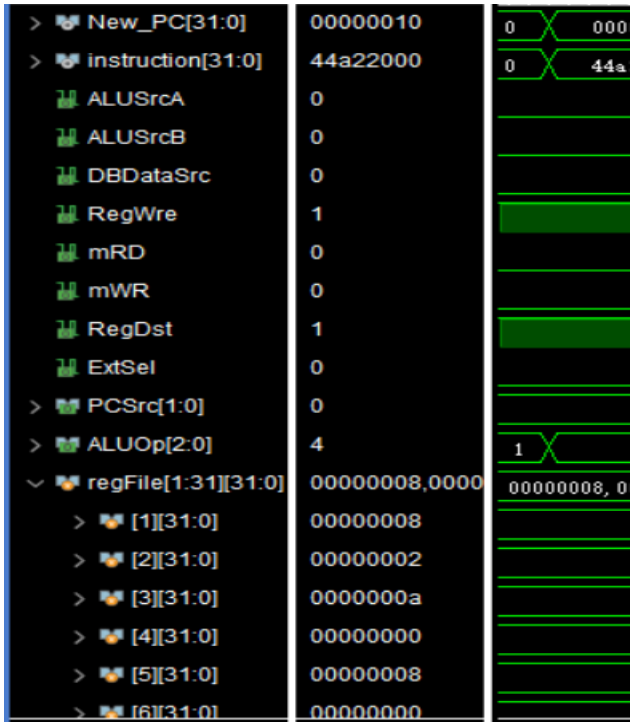
1号寄存器的值+2号寄存器的值=3号寄存器的值，所以3号寄存器的值为0AH

第四条指令：sub　$5,$3,$2



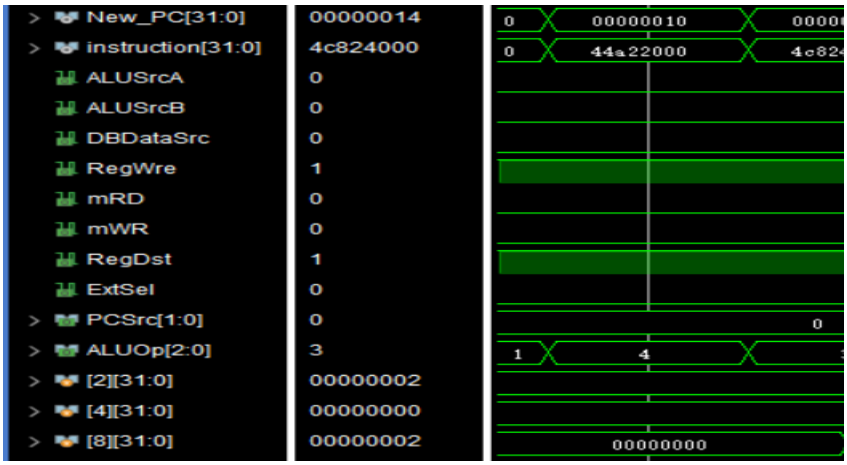
10-2 = 8，所以5号寄存器值为8

第五条指令：and　$4,$5,$2



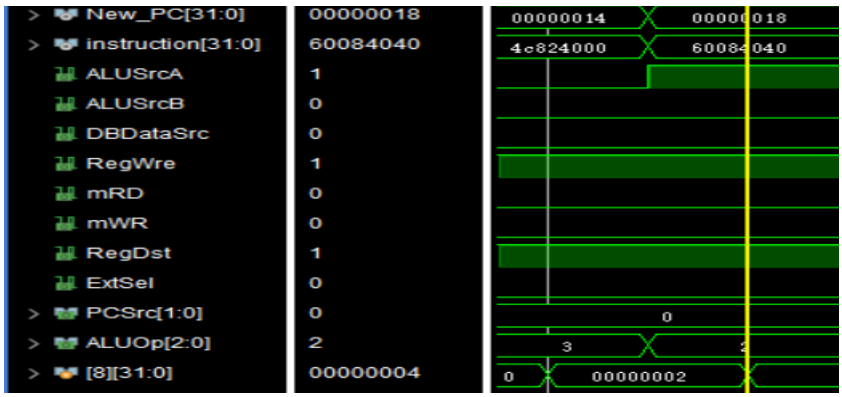
8&2 = 0，所以6号寄存器值为0

第六条指令：or　$8,$4,$2



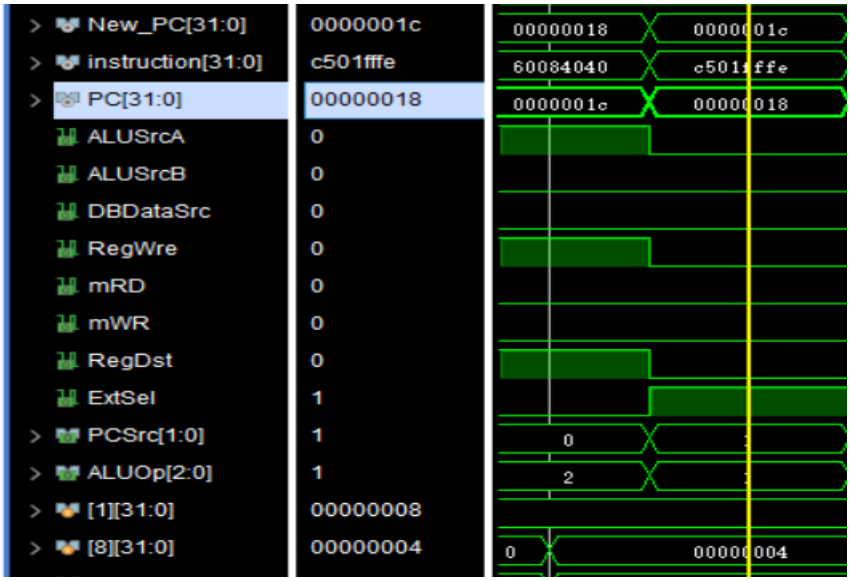
0|2 = 2，所以8号寄存器值为2

第七条指令：sll　$8,$8,1



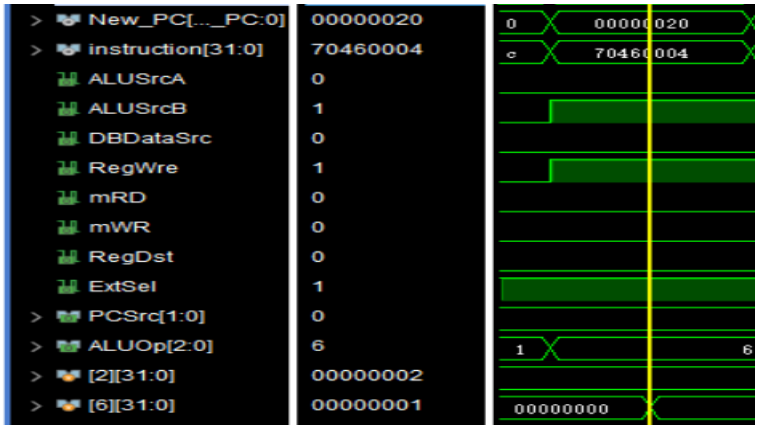
2<<1 = 4，所以8号寄存器值为4

第八条指令： **bne $8,$1,-2**



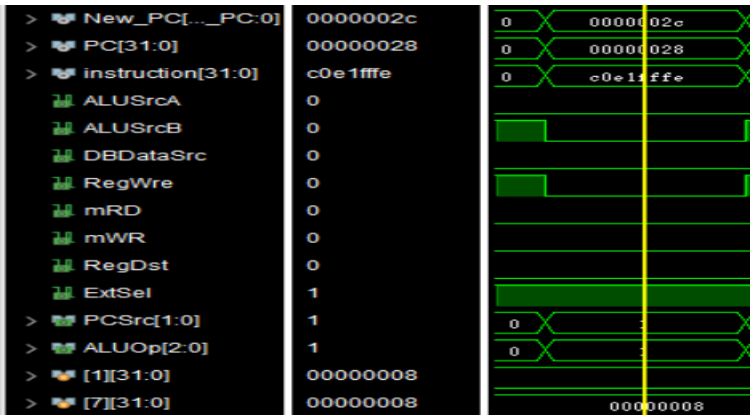
因为八号寄存器值不等于一号寄存器值，所以下一条PC为18H

第九条指令：slti　$6,$2,4



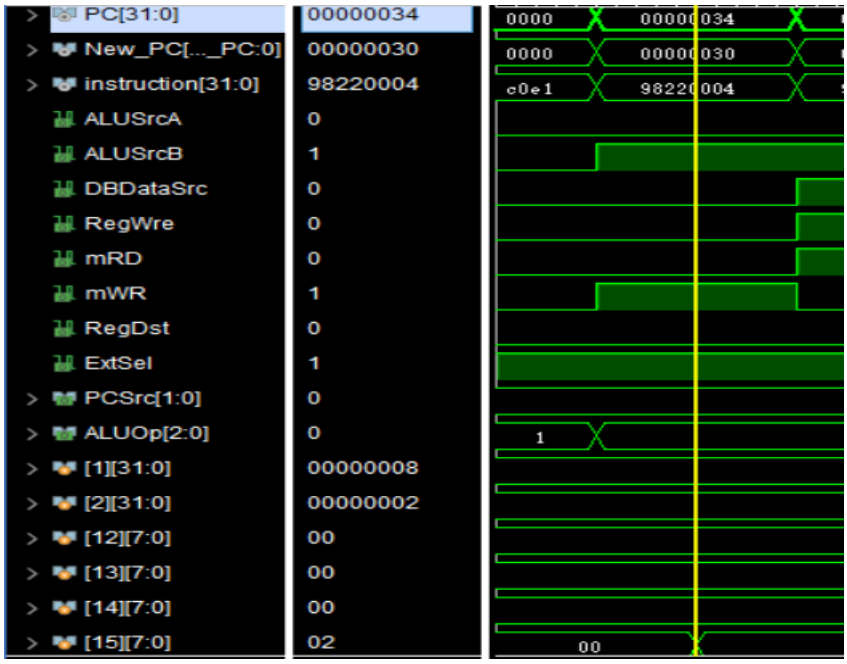
2号寄存器值为2，小于4，所以6号寄存器值为1

第十条指令：**beq $7,$1,-2**



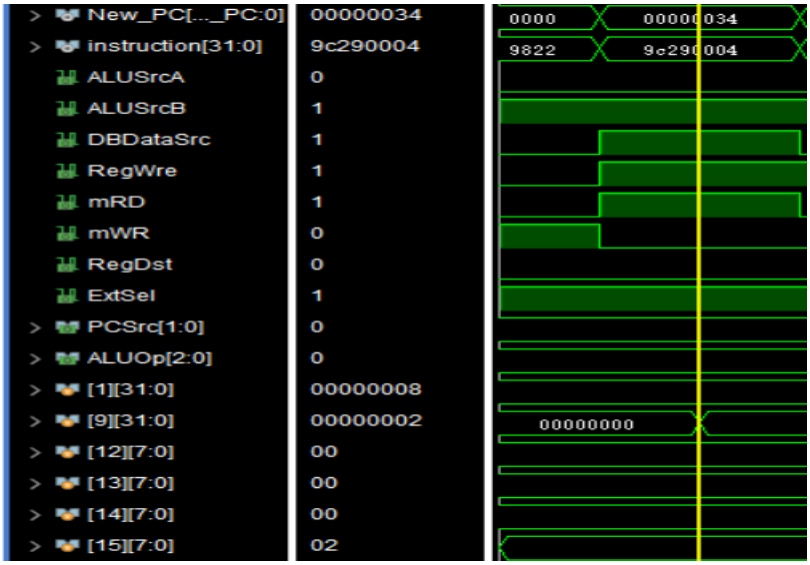
因为1号寄存器值等于7号寄存器值，所以下一条PC为28H

第十一条指令：sw　$2,4($1)



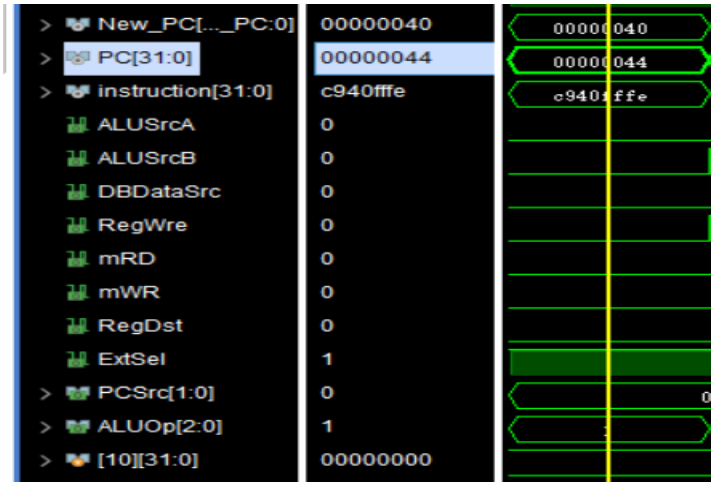
将2号寄存器值2放入存储器，开始地址为1号寄存器值+4即8+4=12，因为设计的CPU为大端存储，所以低位在高地址，所以地址为15的位置的值为2

第十二条指令：lw　$9,4($1)



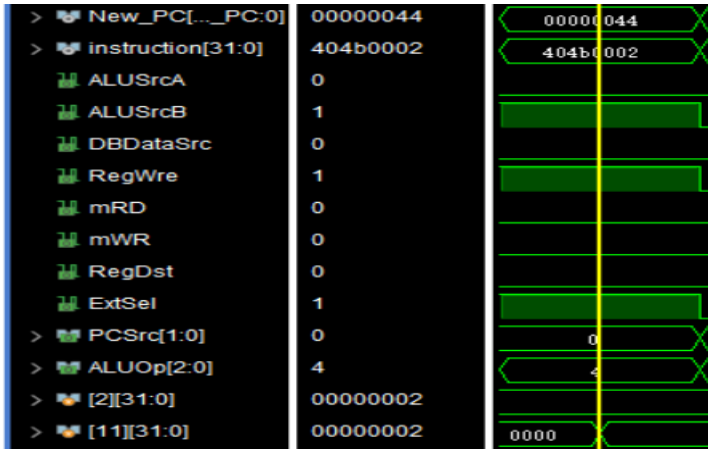
9号寄存器load从12H开始的一个字的数据，所以9号寄存器的值为2

第十三条指令：**bltz $10,-2**



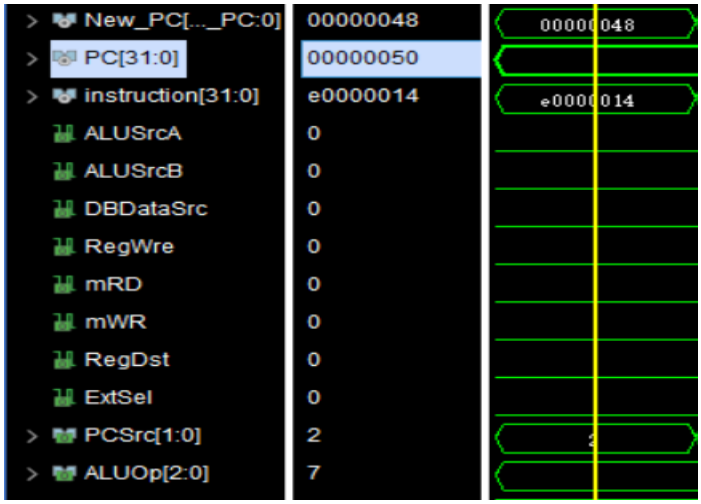
因为10号寄存器值为0，不小于0，所以下一条PC直接加4，也就是44H

第十四条指令：**andi　$11,$2,2**



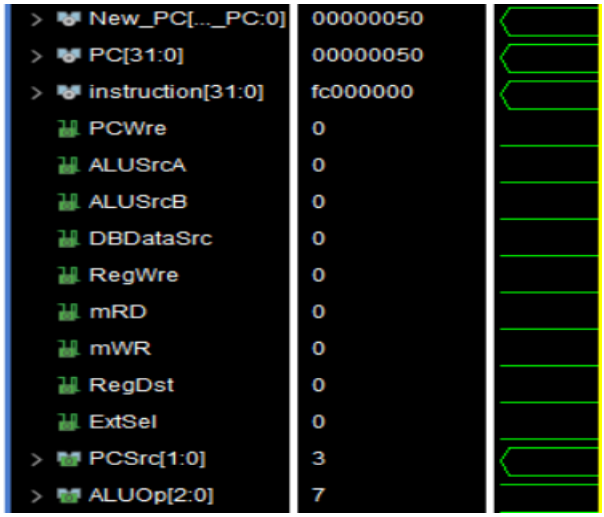
2&2=2，所以11号寄存器值为2

第十五条指令：**j　0x00000050**



J指令，无条件跳转，目标地址是50H,所以下一条PC为50H

第十六条指令：halt



停机指令，所以PCWre为0，不能修改PC，并且寄存器不能写，存储器也不能写。

3、（1）首先在之前设计的CPU的顶层模块上，将要显示的数据的前面改成output；之后要设计一个按键消抖处理，如果不处理，就会产生类似毛刺的上下抖动，这样按一次键就不知道执行了多少条指令了，所以要消抖，至于怎么消抖，我查了资料，大概就是不断取样（取样间隔非常短），如果一定次数的取样中都是高电平，就输出高电平，反之亦然；还有在7段数码管显示的模块，就很简单了，只需要将要显示的数据送进这个显示模块，并且因为一次只能显示一个数字，所以要快频率地改变显示的某一位，产生余晖效应，长生4个数字一直亮着的错觉。

展示消抖模块:

```
module ButtonDeounce(
    input clk,
    input but_in,
    output    but_out
    );
        reg [21:0] count_low;
        reg [21:0] count_high;
        reg key_out_reg;
        always @(posedge clk)
        if(but_in ==1'b0)
            count_low <= count_low + 1;
```

```
        else
            count_low <= 0;
        always @(posedge clk)
            if(but_in ==1'b1)
                count_high <= count_high + 1;
            else
                count_high <= 0;
        always @(posedge clk)
            if(count_high == 5000)
                key_out_reg <= 1;
            else if(count_low == 5000)
                key_out_reg <= 0;
        assign but_out = !key_out_reg;
endmodule
```

（2）连续五条指令的运行结果

addi　$1,$0,8

当前PC：下一条PC

rs寄存器地址：rs寄存器数据

rd寄存器地址：rd寄存器值

ALU结果输出：DB总线数据

ori　$2,$0,2

当前PC：下一条PC

rs寄存器地址：rs寄存器数据

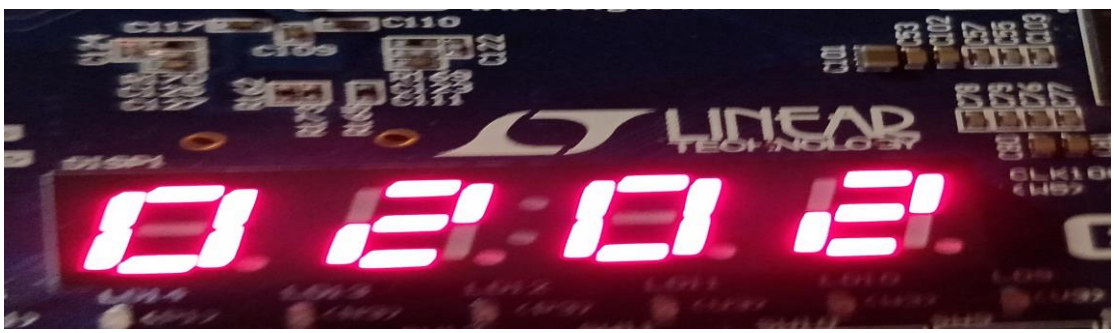
rd寄存器地址：rd寄存器值

ALU结果输出：DB总线数据

add　$3,$2,$1

当前PC：下一条PC

rs寄存器地址：rs寄存器数据



rd寄存器地址：rd寄存器数据

ALU结果输出：DB总线数据

sub　$5,$3,$2

当前PC：下一条PC

rs寄存器地址：rs寄存器数据

rd寄存器地址：rd寄存器数据

ALU结果输出：DB总线数据

and　$4,$5,$2

当前PC：下一条PC

rs寄存器地址：rs寄存器数据

rd寄存器地址：rd寄存器数据

ALU输出结果：DB总线数据

六、实验心得

（1）整个CPU做下来，有种上学期做数字电路实验的感觉，只不过上学期提供了需要的74IP原件，而这学期各个模块是需要自己写，然后写顶层模块就像是在给每个模块连线。

（2）写CPU各个模块还是相对简单的，只要看着数据通路的通路，每个模块有什么输入，输出，怎么工作，但是必须注意，PC的改变是在时钟上升沿，而写数据是在时钟下降沿。还有初始化非常重要，比如PC，如果不初始化，就一直会是XXXX。当然，我在写CPU的过程也不是一帆风顺的，因为模块写多了，在顶层模块里面，实例化的时候，容易弄混一些输入输出，只能在仿真的时候检查错误，相当麻烦，所以后来想了想，其实一些复选器其实可以不写成一个模块，可以在顶层模块里进行数据的选择，这样可以减少至少4个模块。

（3）一开始看烧板要求的时候，不懂什么是消抖，为什么要进行消抖处理，问了同学才知道，我们按了按键后，并不能产生一个方形的波形，而是会像毛刺一样，上下抖动，一会高电平，一会又掉到低电平，这样按一次按键，不知道会执行多少条指令了，所以要进行消抖处理。我百度了消抖处理，查看了相关资料，明白了具体消抖的处理，就是在极短的时间内，不断重复取样，只有取到一定次数，并且在这些取样中，都是高电平，就认为是高电平，反之亦然。

（4）有一个问题就是，我用了同学的消抖处理和网上的消抖处理，感觉两个的核心思想都差不多，但同学的消抖处理使得第一条指令没有写回寄存器，这是我们非常疑惑的。

（5）建议：这次实验报告真的很麻烦，要拍很多照，截很多图，希望下次流水线CPU的要求可以低点。