《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（7）班
学 生 姓 名 ：	张吉祺
学　　号 ：	16340286
时　　间 ：	2018　年　5　月　15　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

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

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。

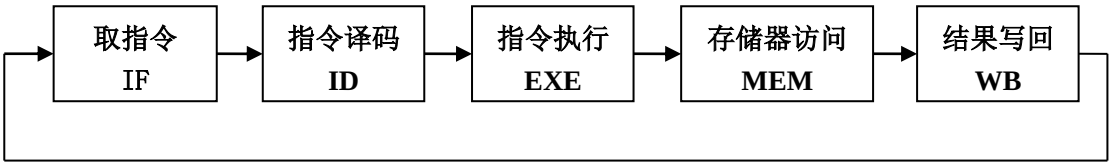
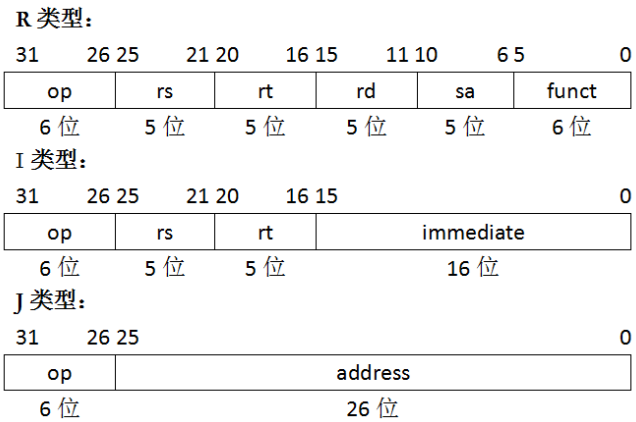


图 1　单周期 CPU 指令处理过程

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（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。

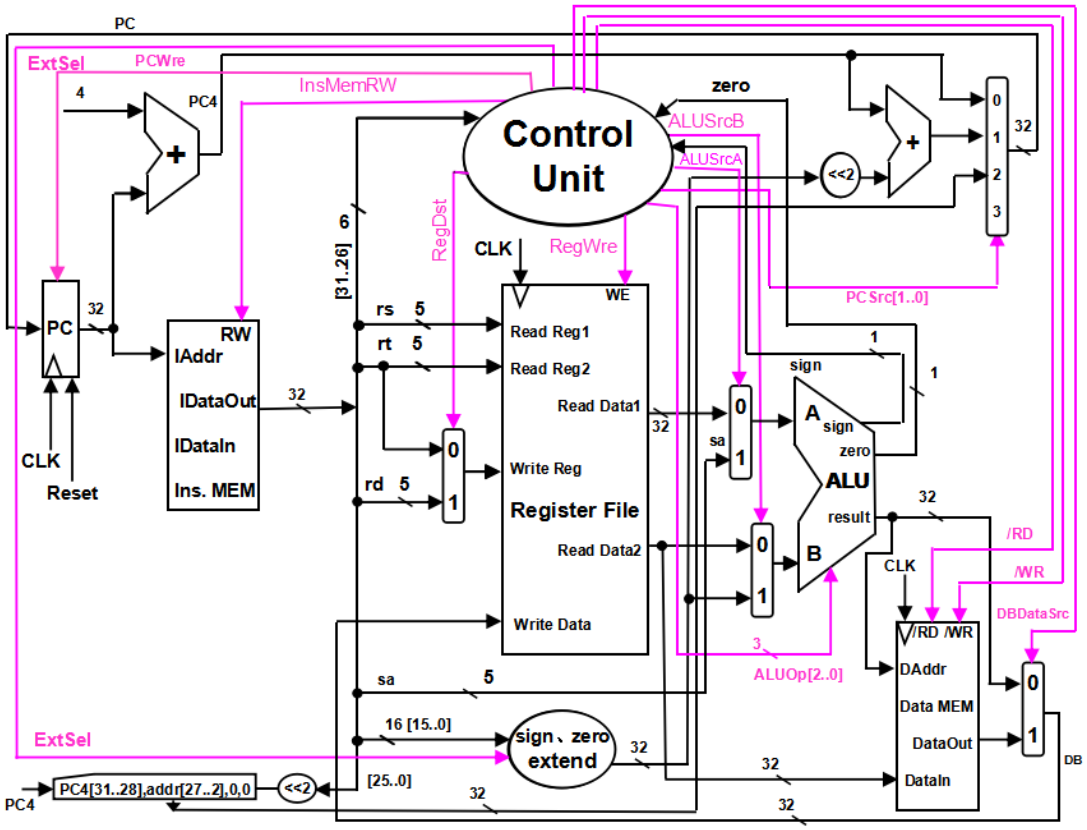


图 2　单周期 CPU 数据通路和控制线路图

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。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，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。

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

围绕“**图2 单周期CPU数据通路和控制线路图**”将CPU划分为PC、ALU、RegisterFile、SignZeroExtend、InstructionMemory、DataMemory、ControlUnit几个独立的模块，通过“**表3 控制信号与指令之间的关系表**”将各信号之间的逻辑关系对应到各模块，并实现各模块负责的功能，最后由顶层模块综合所有模块组成单周期CPU。

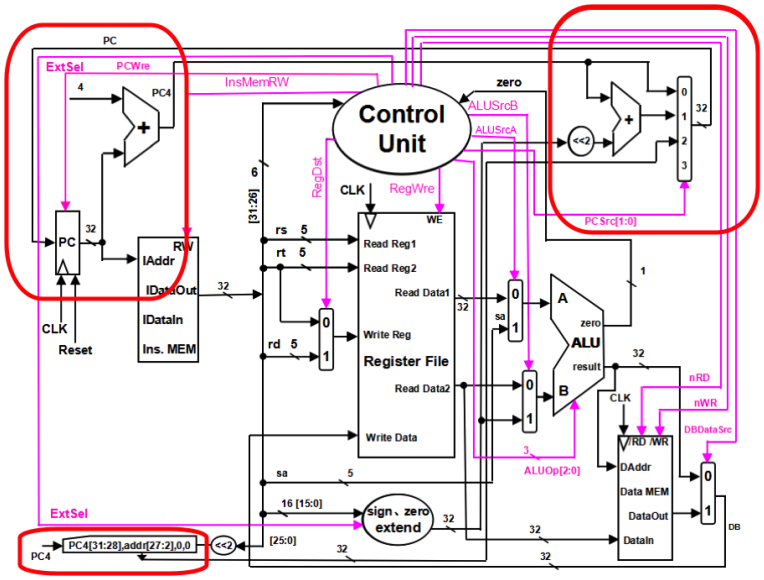
表3 控制信号与指令之间的关系表

指令 \ 控制信号	PCWre	ExtSel	DBDataSrc	ALUSrcB	ALUSrcA	PCSrc	ALUOp	RegWre	RegDst
addi	1	1	0	1	0	00	000	1	0
ori	1	0	0	1	0	00	011	1	0
add	1	1	0	0	0	00	000	1	1
sub	1	1	0	1	0	00	001	1	1
and	1	1	0	0	0	00	100	1	1
or	1	1	0	0	0	00	011	1	1
sll	1	1	0	0	1	00	010	1	1
bne	1	1	0	0	0	01	001	0	1
slti	1	1	0	1	0	00	110	1	0
beq	1	1	0	0	0	01	001	0	1
sw	1	1	0	1	0	00	000	0	1
lw	1	1	1	1	0	00	000	1	0
j	1	1	0	0	0	10	000	0	1
half	0	1	0	0	0	00	000	0	1

1、各模块的设计：

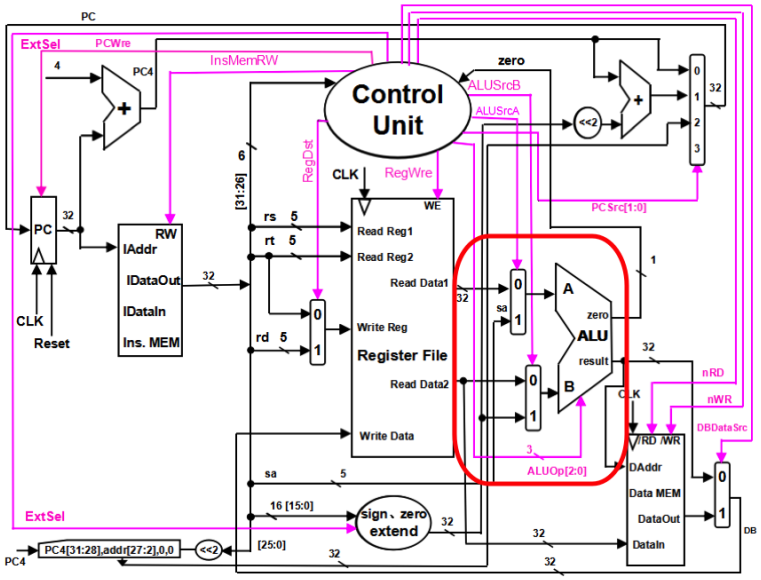
PC模块：主要功能是输出指令，输入接口为PCWre、PCSrc、Reset、PCAddress、PC、

时钟信号。PC模块在时钟信号上升沿时，按PCWre、PCSrc信号和相应规则修改PC值，将PC值输出到IAddr。




```
case(PCSrc)
    2'b00:Address<=Address+4;// pc<—pc+4
    2'b01:Address<=Address+4+(immediate*4);// pc<—pc+4+immediate*4
    2'b10:begin //pc<—{(pc+4)[31:28],addr[27:2],2{0}}
        Address<=Address+4;
        Address<={Address[31:28],address,2'b00};
    end
    default:Address<=0;
endcase
```

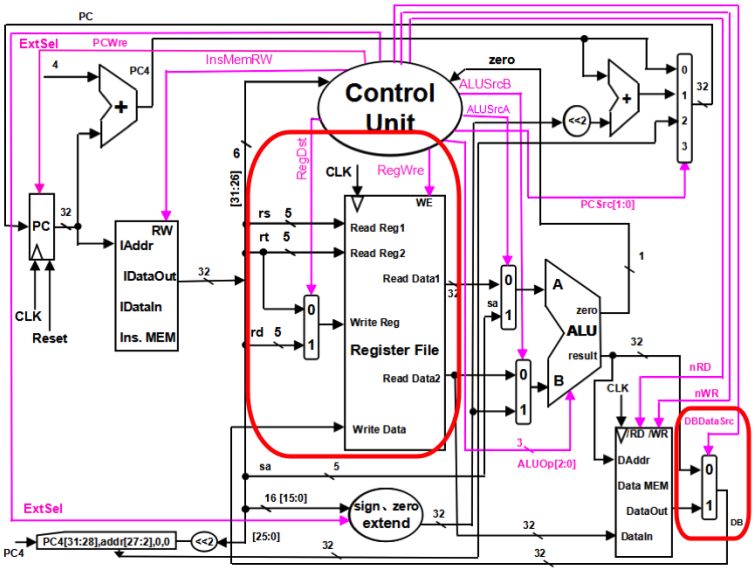
ALU模块：主要功能是进行逻辑运算。ALUOp信号控制算术逻辑运算功能，ALUSrcA、ALUSrcB信号控制对应的数据输入，运算结果输出至result，zero输出运算结果是否为0。



根据 表2 ALU运算功能表 明确对应关系。

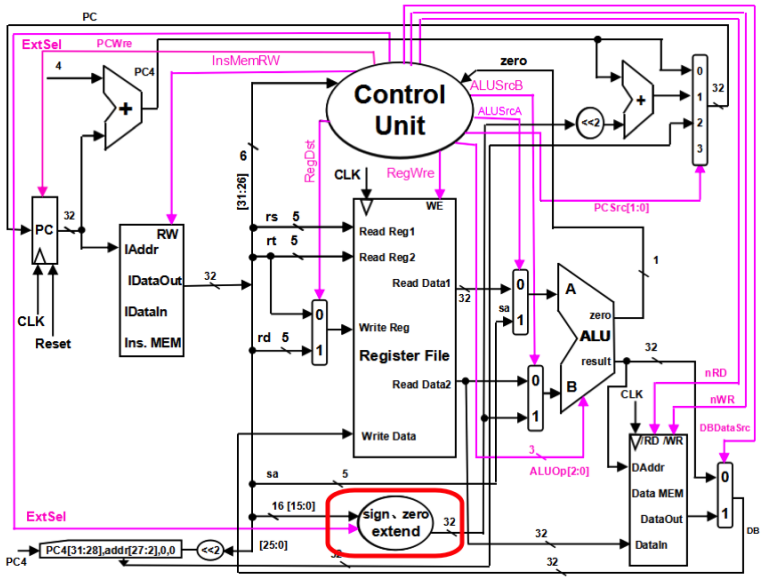
```
    case(ALUOp)
        3'b000:result=A+B;
        3'b001:result=A-B;
        3'b010:result=B<<A;
        3'b011:result=A|B;
        3'b100:result=A&B;
        3'b101:result=(A<B)?1:0;
        3'b110:result=((A<B&&A[31]==B[31])||(A[31]&&!B[31]))?1:0;
        3'b111:result=A^B;
        default:result=0;
    endcase
    zero=(result==0)?1:0;
```

RegisterFile模块：寄存器堆模块。RegWre控制存取功能，考虑竞争与冒险，在时钟的下降沿并RegWre信号同时为1时进行写入操作。



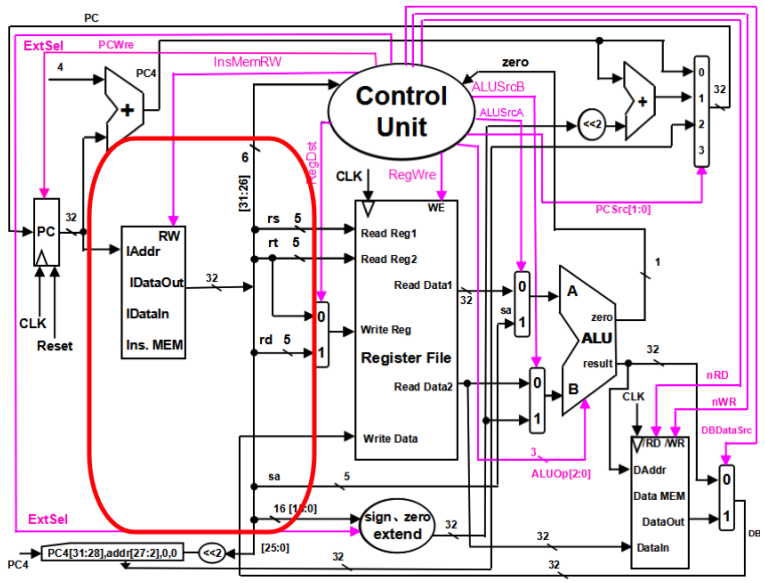

```
    //读取
    assign ReadData1=Register[rs];
    assign ReadData2=Register[rt];
    //写入
    always@(negedge CLK)begin
        if(RegWre&&WriteReg)
            Register[WriteReg]<=WriteData;
    End
```

SignZeroExtend模块：符号和零扩展模块。根据扩展规则补全0或1，扩展成32位数据输出。



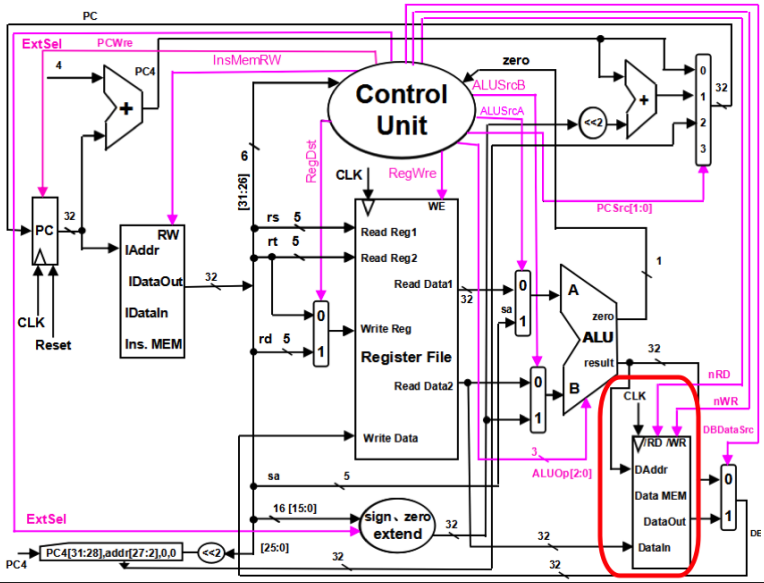

```
    assign ext[15:0]=immediate;
    assign ext[31:16]=ExtSel?(immediate[15]?16'hffff:0):0;
```

InstructionMemory模块：指令存储器模块。将PC模块的IAdrr信号即指令地址转化为对应的指令信号，输出到op、rs、rt、rd、immediate。InsMemRW控制读或写。



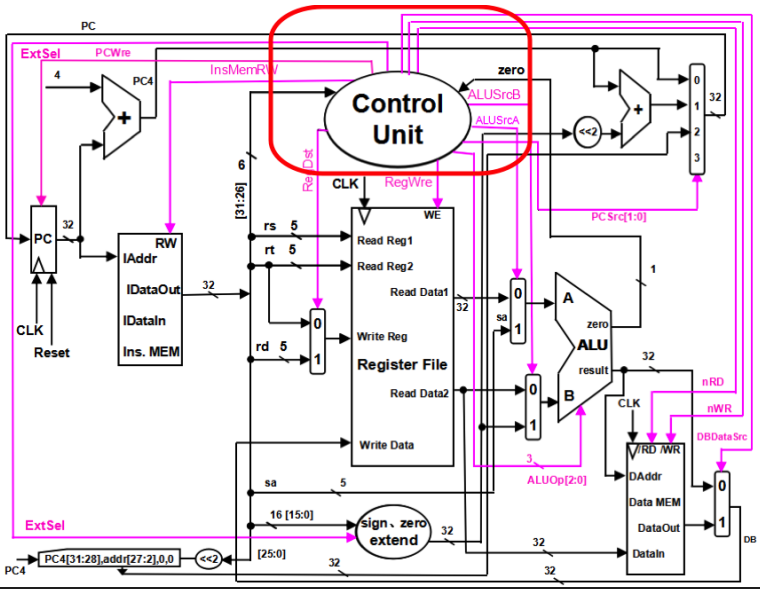

```
    assign op=Instruction[IAddr+3][7:2];
    assign rs={Instruction[IAddr+3][1:0],Instruction[IAddr+2][7:5]};
    assign rt=Instruction[IAddr+2][4:0];
    assign rd=Instruction[IAddr+1][7:3];
    assign sa={Instruction[IAddr+1][2:0],Instruction[IAddr][7:6]};
    assign immediate={Instruction[IAddr+1][7:0],Instruction[IAddr][7:0]};
    assign
address={Instruction[IAddr+3][1:0],Instruction[IAddr+2][7:0],Instruction[IAd
dr+1][7:0],Instruction[IAddr][7:0]};
```

DataMemory模块：数据存储器模块。数据存储器存储单元宽度使用8位，设置存储器数量为64个。DataMemRW控制读写操作。为避免竞争与冒险，设定在DataMemRW为1且时钟信号为下降沿时执行写入。




```
    reg [7:0] Data [0:63];

    assign
DataOut=(!nRD)?{Data[DAddr*4+3],Data[DAddr*4+2],Data[DAddr*4+1],Data[DAddr*4
]}:32'bz; // z 为高阻态

    always@(negedge CLK)begin
        if(!nWR)begin
            Data[DAddr*4+3]<=DataIn[31:24];
            Data[DAddr*4+2]<=DataIn[23:16];
            Data[DAddr*4+1]<=DataIn[15:8];
            Data[DAddr*4]<=DataIn[7:0];
        end
    end
```

ControlUnit模块：控制单元模块。根据指令输出各模块的控制信号控制其他模块执行完成完整的功能。通过case语句，按照“**表3 控制信号与指令之间的关系表**”，将op信号对应到各模块的控制信号。



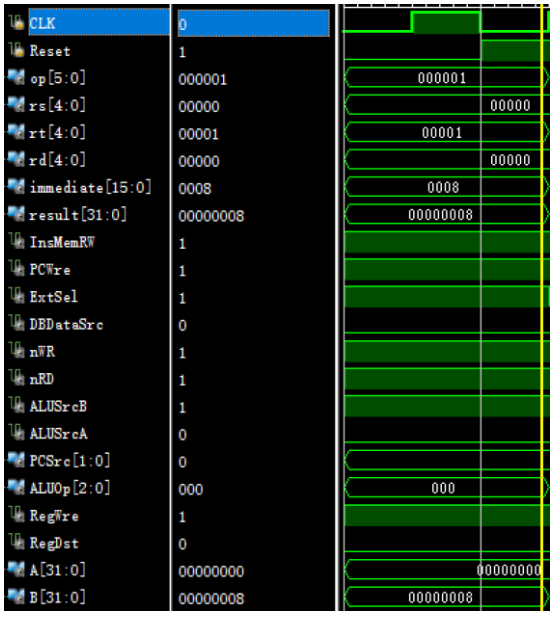

```
    //写指令存储器
    assign InsMemRW=1;
    //half PC 不更改
    assign PCWre=(op==6'b111111)?0:1;
    //ori 0 扩展
    assign ExtSel=(op==6'b010000)?0:1;
    //lw 来自数据存储器（Data MEM）的输出
    assign DBDataSrc=(op==6'b100111)?1:0;
    //sw 写数据存储器
    assign nWR=(op==6'b100110)?0:1;
    //lw 读数据存储器
    assign nRD=(op==6'b100111)?0:1;
    //addi、ori、slti、sw、lw 来自 sign 或 zero 扩展的立即数
    assign
ALUSrcB=(op==6'b000001||op==6'b010000||op==6'b011011||op==6'b100110||op==6'b
100111)?1:0;
    //sll 来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}
    assign ALUSrcA=(op==6'b011000)?1:0;
    //j 10：pc<—{(pc+4)[31:28],addr[27:2],2{0}}
    assign PCSrc[1]=(op==6'b111000)?1:0;
    //beq(zero=1)、bne(zero=0) 01：pc<—pc+4+(sign-extend)immediate
    assign
PCSrc[0]=((op==6'b110000&&zero==1)||((op==6'b110001)&&zero==0))?1:0;
    //and、slti 100：与 110：比较 A 与 B 带符号
    assign ALUOp[2]=(op==6'b010001||op==6'b011011)?1:0;
    //sll、slti、ori、or 010：B 左移 A 位 011：或 110：比较 A 与 B 带符号
    assign
ALUOp[1]=(op==6'b011000||op==6'b011011||op==6'b010000||op==6'b010010)?1:0;
    //sub、ori、or、beq、bne 001：减 011：或 111：异或
```

```
    assign
ALUOp[0]=(op==6'b000010||op==6'b010000||op==6'b010010||op==6'b110000||op==6'
b110001)?1:0;
    //beq、bne、sw、halt、j 无写寄存器组寄存器
    assign
RegWre=(op==6'b110000||op==6'b110001||op==6'b100110||op==6'b111111||op==6'b1
11000)?0:1;
    //addi、ori、lw、slti 写寄存器组寄存器的地址，来自 rt 字段
    assign
RegDst=(op==6'b000001||op==6'b010000||op==6'b100111||op==6'b011011)?0:1;
```

2、验证CPU正确性：

开始时Reset信号有一个上升沿，开始进入代码段。接着在第一个时钟上升沿信号到来后，CPU开始按顺序执行各条指令。

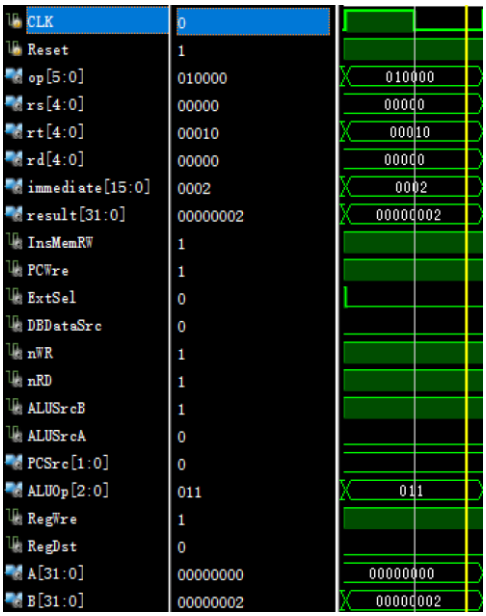
地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addi $1,$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008



将$0（值为0）与立即数8相加，ALU的ALUOp为000，执行加，A为0，B为8，结果result值为8，存入寄存器$1。

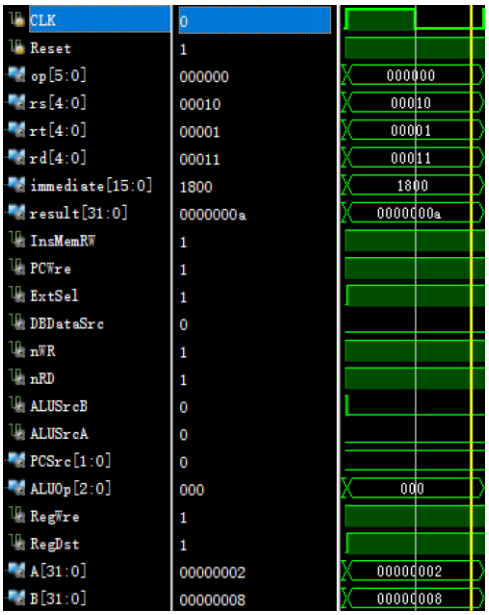
地址	汇编程序	指令代码				
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码

0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010	=	40020002



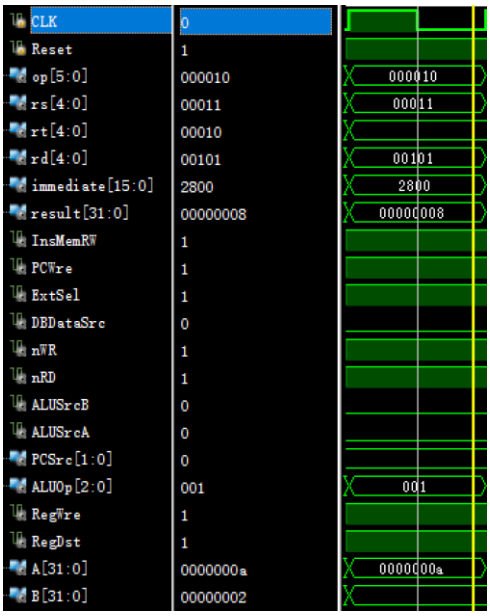
将$0（值为0）与立即数2相或，ALU的ALUOp为011，执行或，A为0，B为2，结果result为2，存入寄存器$2。

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**
0x00000008	add $3,$2,$1	000000	00010	00001	00011 00000000000	=	00411800



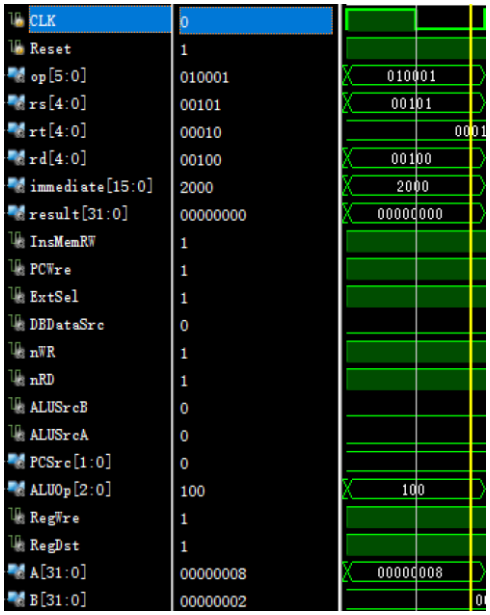
将$2（值为2）与$1（值为8）相加，ALU的ALUOp为000，执行加，ALU输入均为寄存器值，ALUSrcA和ALUSrcB均为0，A为2，B为8，结果result为a，存入寄存器$3。

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	
0x0000000C	sub $5,$3,$2	000010	00011	00010	00101 00000000000	=	08622800



将$3（值为a）与$2（值为2）相减，ALU的ALUOp为001，执行减，ALU输入均为寄存器值，ALUSrcA和ALUSrcB均为0，A为a，B为2，结果result为8，存入寄存器$5。

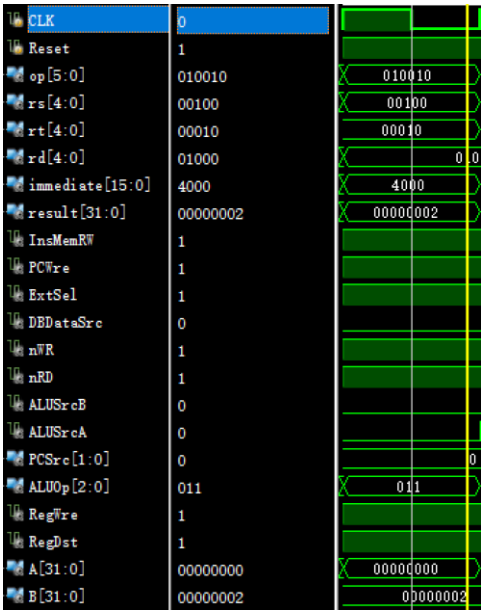
地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000010	and $4,$5,$2	010001	00101	00010	00100 00000000000	=	44A22000



将$5（值为8）与$2（值为2）相与，ALU的ALUOp为100，执行与，ALU输入均为寄存器值，ALUSrcA

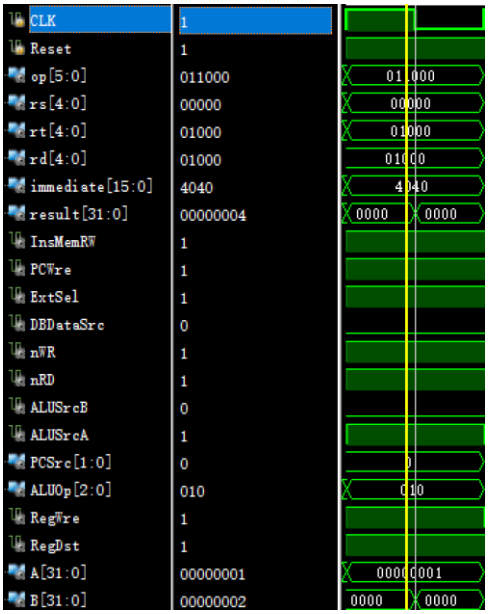
和ALUSrcB均为0，A为8，B为2，结果result为0，存入寄存器$4。

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	
0x00000014	or $8,$4,$2	010010	00100	00010	01000 00000000000	=	48822000



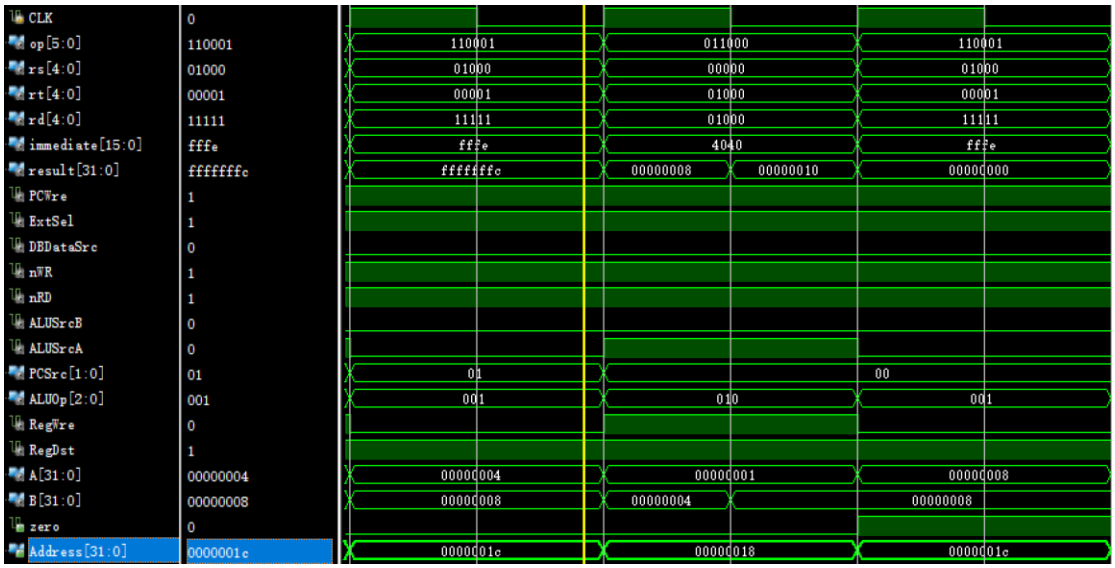
将$4（值为0）与$2（值为2）相或，ALU的ALUOp为011，执行或，ALU输入均为寄存器值，ALUSrcA和ALUSrcB均为0，A为0，B为2，结果result为2，存入寄存器$8。

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	
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 000000	=	60084040



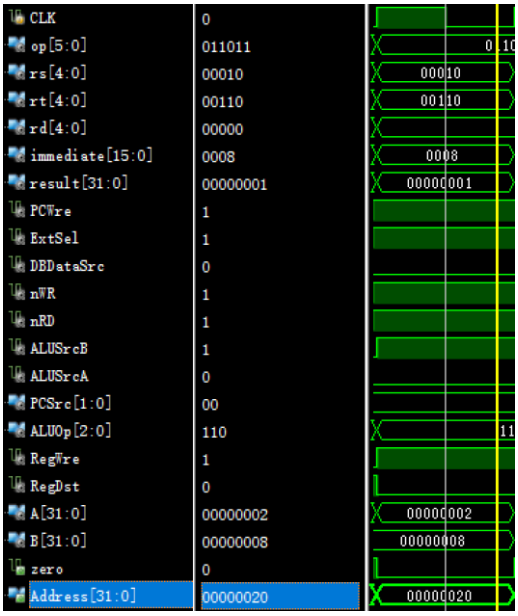
将$8（值为2）和立即数1传入ALU，ALU的ALUOp为010，执行左移，A输入为立即数，ALUSrcA为1，B输入为寄存器值，ALUSrcB为0，A为1，B为2，结果result为4，存入寄存器$8。

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	
0x0000001C	bne $8,$1,-2 (≠,转18)	110001	01000	00001	1111 1111 1111 1110	=	C501FFFE



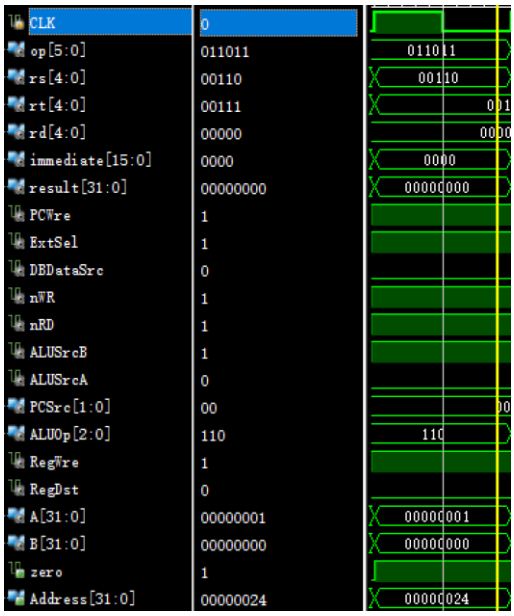
将寄存器$8（值为4）和$1(值为8)传入ALU，ALU的ALUOp为001,执行相减，ALU输入均为寄存器值,ALUSrcA和ALUSrcB均为0,zero为0,PCSrc为01,对PC执行为PC+4+(sign-extend)(-2)*4（即0x00000018),在CLK的下个上升沿到来时Address变为00000018,回到上一步再次执行寄存器$8值左移1位，$8值变为8。再执行此指令，ALU的zero信号为1，执行下一步。

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	
0x00000020	slti $6,$2,8	011011	00010	00110	0000 0000 0000 1000	=	6C460008



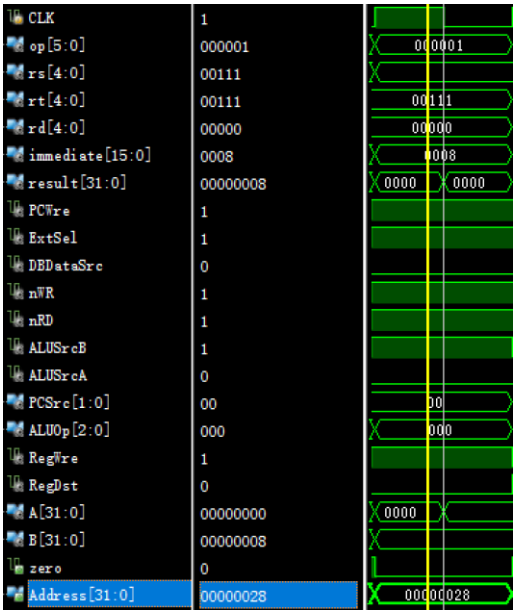
将$8（值为2）和立即数1传入ALU，ALU的ALUOp为110，执行带符号比较，A输入为寄存器值，ALUSrcA为0，B输入为立即数，ALUSrcB为1，A为2，B为8，结果result为1，存入寄存器$6。

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	
0x00000024	slti $7,$6,0	011011	00110	00111	0000 0000 0000 0000	=	6CC70000



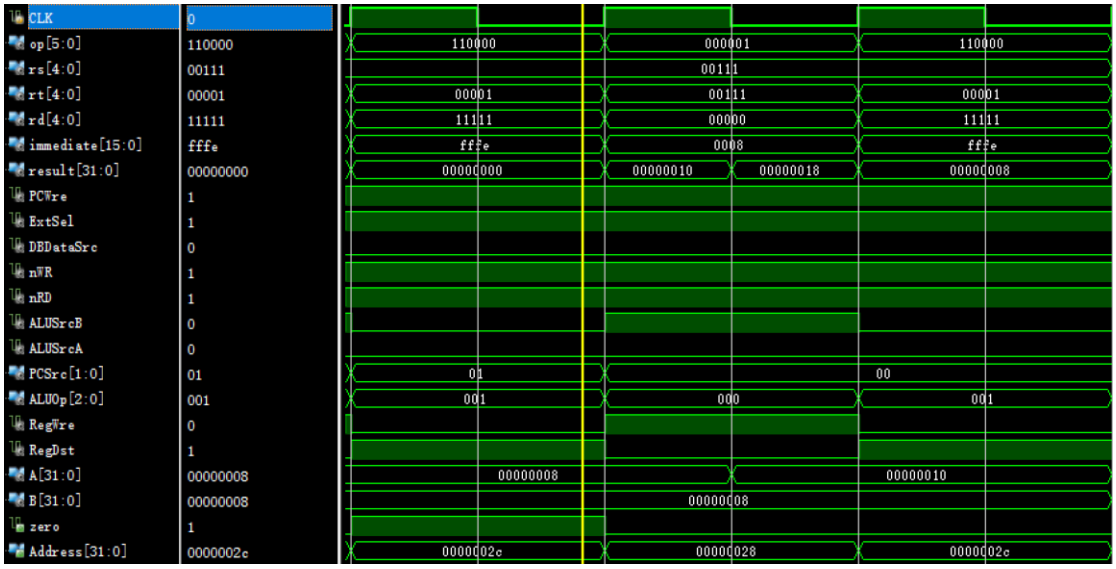
将$6（值为1）和立即数0传入ALU，ALU的ALUOp为110，执行带符号比较，A输入为寄存器值，ALUSrcA为0，B输入为立即数，ALUSrcB为1，A为1，B为0，结果result为0，存入寄存器$7。

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	
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000	=	04E70008



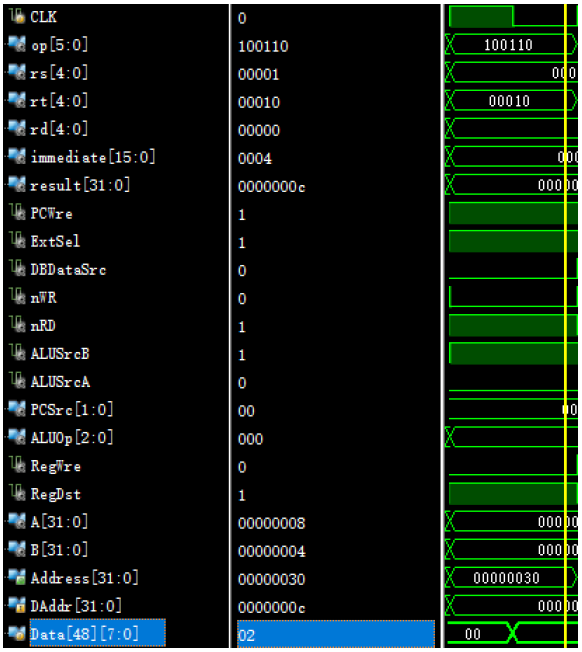
将$7（值为0）与立即数8相加，ALU的ALUOp为000，执行加，A为0，B为8，结果result值为8，存入寄存器$7。

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	
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	=	C0E1FFFE



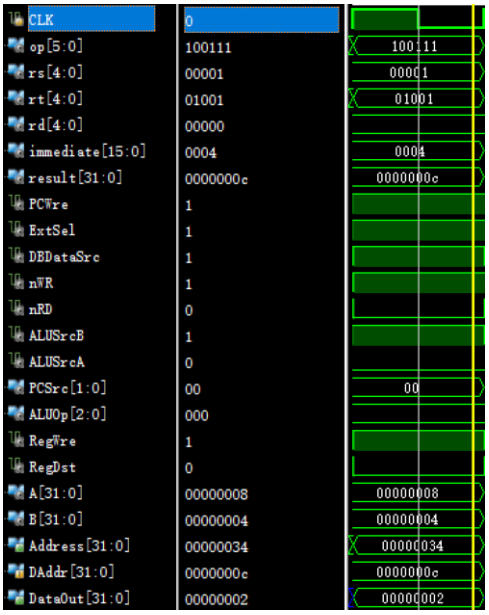
将寄存器$7（值为8）和$1(值为8)传入ALU，ALU的ALUOp为001,执行相减，ALU输入均为寄存器值,ALUSrcA和ALUSrcB均为0,zero为1,PCSrc为01,对PC执行为PC+4+(sign-extend)(-2)*4（即0x00000028),在CLK的下个上升沿到来时Address变为00000028,回到上一步再次执行寄存器$7值加上立即数8，$7值变为16。再执行此指令，ALU的zero信号为0，执行下一步。

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	
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100	=	98220004



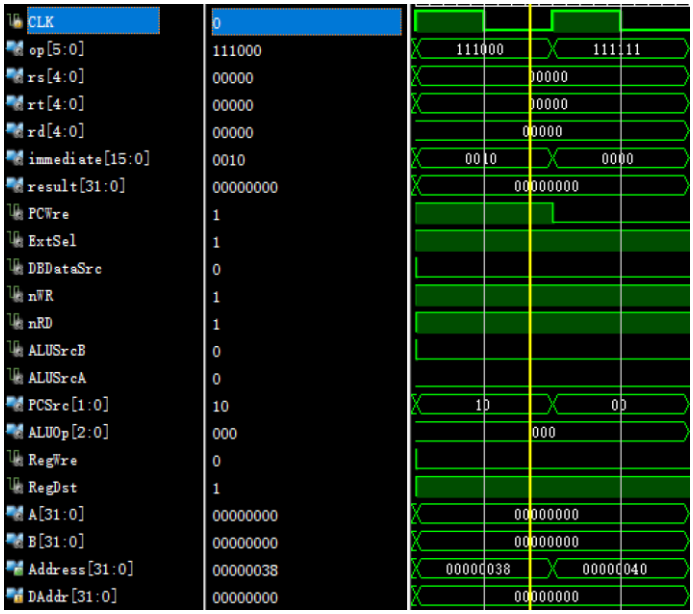
此时nWR变为0，写数据存储器，将$2（值为2）的值即DataIn存入DataMemory中,DAddr为$1（值为8）加上偏移量4即c,在CLK为下降沿时写入，Data[48][7:0]变为02。

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	
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	=	9C290004



此时nRD变为0，读数据存储器，将DAddr为$1（值为8）加上偏移量4即c位置即Data[48]的值2取出来，并存入寄存器$9中。

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	
0x00000038	**j 0x00000040**	111000	0000 0000 0000 0000 0000 0100 00			=	E0000010
0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010	=	040A000A
0x00000040	**halt**	111111	00000	00000	0000000000000000	=	FC000000



PCSrc为10，对PC执行为{(pc+4)[31:28],addr[27:2],2{0}}（即0x00000040），在CLK的下个上升沿到来时Address变为00000040，跳过了0000003C，执行half指令，PCwre变为0，停机。

3、实现：

数码管显示格式：左边两位数码管 BB：右边两位数码管 BB。

SW_in = 00:显示 当前 PC 值:下条指令 PC 值

SW_in = 01:显示 RS 寄存器地址:RS 寄存器数据

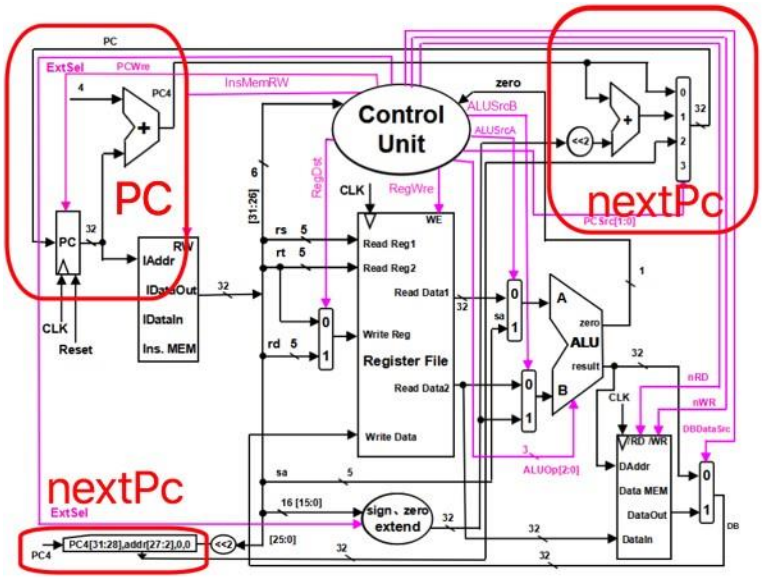
SW_in = 10:显示 RT 寄存器地址:RT 寄存器数据

SW_in = 11:显示 ALU 结果输出 :DB 总线数据

由于前面设计的 CPU 顶层模块 I/O ports 较多，需要根据实验板上的接口再设计一个更顶层的模块。且为了输出上述内容，需要将其添加到原来的顶层模块输出接口。

为了输出 nextPC，需要对前面设计的 PC 模块进行拆分，拆分为 PC 模块和 nextPc 模块。

```
module SingleCycleCPU(
      input CLK,
      input Reset,
      output [31:0] PCAddress,
      output [31:0] nextPC,
      output [4:0] rs,
      output [4:0] rt,
      output [31:0] RegisterFileReadData1,
      output [31:0] RegisterFileReadData2,
      output [31:0] ALUResult,
      output [31:0] DataMemoryDataOut
   );
```



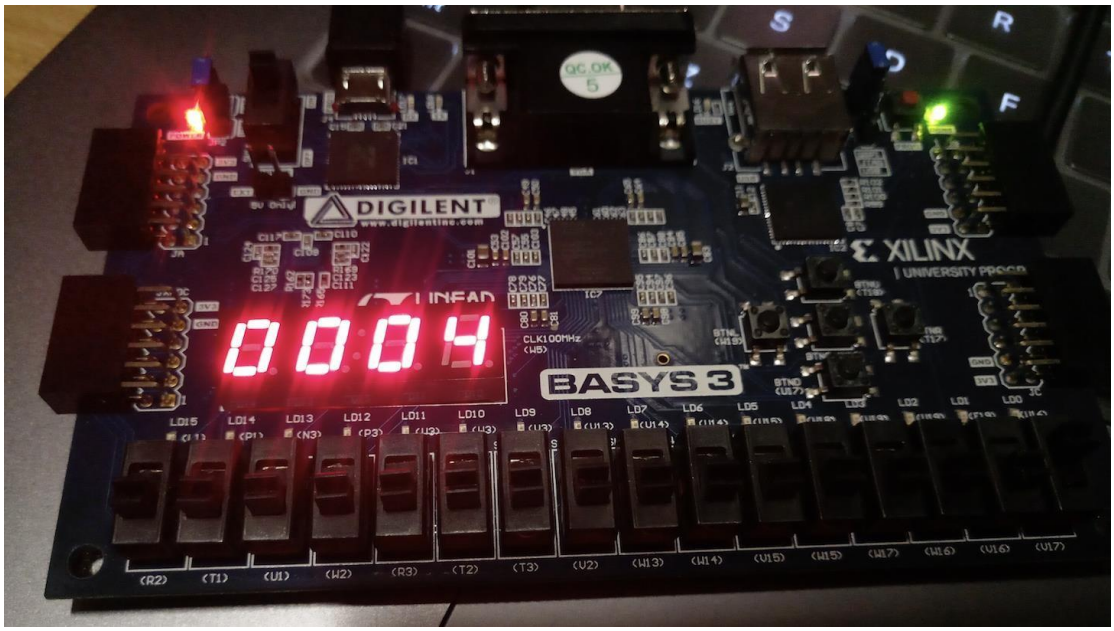

```
module PC(
       input CLK,
       input Reset,
       input [31:0] nextPC,
       input PCWre,
       output reg [31:0] Address
    );
module nextPc(
       input [1:0] PCSrc,
       input [31:0] Address,
       input [31:0] immediate,
       input [25:0] address,
       output reg [31:0] nextPC
    );
```

此外，还需要添加时钟分频、按键触发、开关控制、数码管显示模块，最终最顶层的模块:

```
Clock_div clock_div(W5,CLK);

    Button button(CLK,button,trigger);

SingleCycleCPU

singleCycleCPU(trigger,Reset,PCAddress,nextPC,rs,rt,RegisterFileReadData1,Re

gisterFileReadData2,ALUResult,DataMemoryDataOut);     Src

src(src,PCAddress[7:0],nextPC[7:0],rs,rt,RegisterFileReadData1[7:0],Register

FileReadData2[7:0],ALUResult[7:0],DataMemoryDataOut[7:0],data);

    Display display(CLK,Reset,data,seg,an,dp);
```

再根据接口修改一下仿真代码，观察波形确定无误后，进行综合、实现和烧写。

板上 Reset(右侧 V17)置 1，SW_in(左侧 R2,T1)开始测试。



列举其中几条指令：

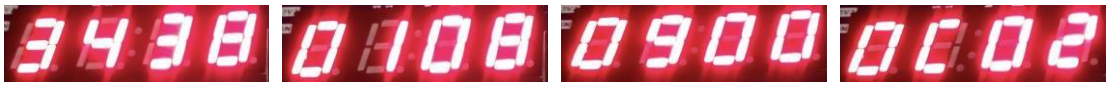
地址	汇编程序	指令代码					
		op（6）	**rs(5)**	**rt(5)**	**rd(5)/immediate (16)**	**16** 进制数代码	
0x00000000	addi $1,$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008

当前 PC:下条指令 PC　RS 地址:RS 寄存器数据　RT 地址:RT 寄存器数据　ALU 结果:DB 数据

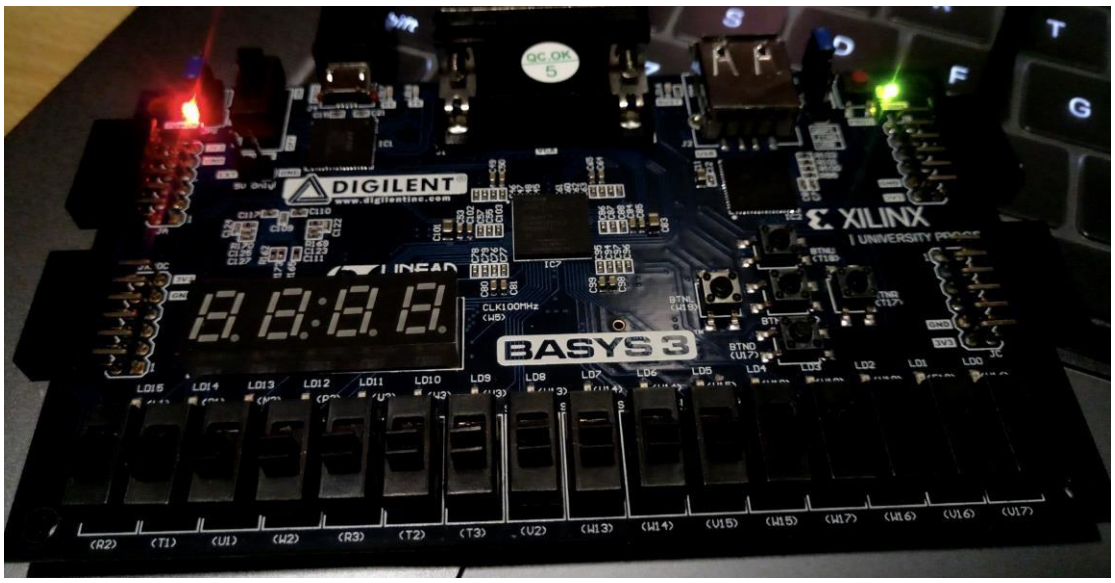
0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010	=	40020002

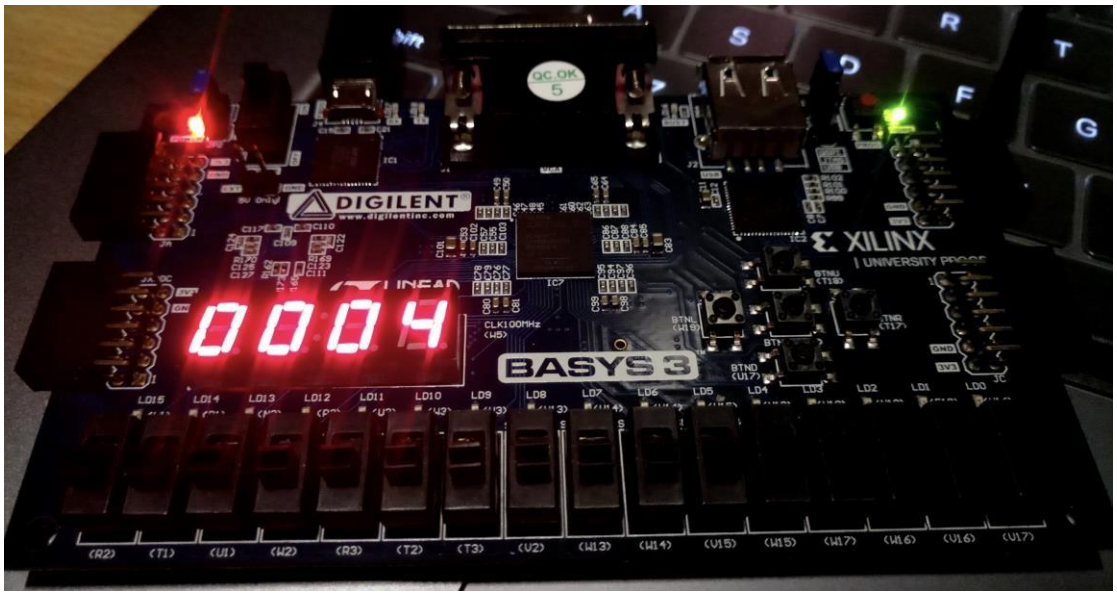
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	=	C0E1FFFE

0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	=	9C290004



执行到 half 指令后，要重新从头执行，可以 Reset 置 0 后再置 1。





再次回到开始的状态。

六. 实验心得

通过对单周期CPU的设计，不仅加深了对CPU结构、数据通路的学习和理解，也初步学习了使用Verilog HDL语言进行编程设计，也熟悉了Vivado的使用。

1、wire和reg的区别：

i. 仿真时使用软件思路，Verilog HDL语言面对的是编译器，wire对应于连续赋值，如assign。reg对应于过程赋值，如always，initial。

ii. 综合时使用电路思路，Verilog HDL语言面对的是综合器，wire综合出来一般情况下是一根导线。Reg综合出来不一定是register，在always中以 always @ (a or b or c) 形式的，即不带时钟边沿的，综合出来是组合逻辑，只是net；以 always @ (posedge clk)

形式的，即带有边沿的，综合出来一般是时序逻辑，才是以Flip-Flop形式表示的register触发器。

iii．设计中，输入信号一般来说不能判断出上一级是寄存器输出还是组合逻辑输出，对于本级来说，就当成一根导线，即wire型。而输出信号是reg还是组合逻辑输出即wire和reg型都可以。但一般整个设计（即顶层模块）的外部输出是reg输出，这比较稳定、扇出能力好。

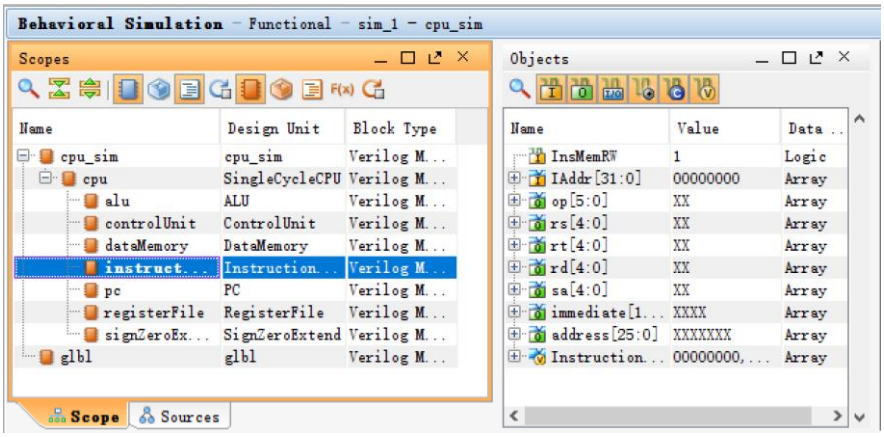
iv．assign语句应用wire，否则编译器报错；元件实例化时必须用wire型，因为wire为无逻辑连线，输入什么就的输出什么，如 assign　　c = a & b; 综合时c是连接到a & b综合成 a、b与门的输出线，综合出来的是与门而不是c，因此always语句对wire变量赋值编译器会报错。

v．reg型数据保持最后一次的赋值，而wire型数据需要持续的驱动。

vi．Verilog HDL语法规定调用子模块时，输出端口只能用wire类型变量进行映射。

2、Vivado的使用：

i．仿真时可通过Scope窗口观察各模块的变量的值。



ii．Messages窗口中可排查错误，有时应与Tcl Console相结合使用。