中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验三）

学 院 名 称	：	数据科学与计算机学院
专业（班级）	：	16 计算机类 4 班
学 生 姓 名	：	郑映雪
学　　号	：	16337327
时　　间	：	2017　年　12　月　17　日

成 绩 ：

实验三 ： 多周期CPU设计与实现

一. 实验目的

1. 认识和掌握多周期数据通路原理及其设计方法；
2. 掌握多周期CPU的实现方法，代码实现方法；
3. 编写一个编译器，将MIPS汇编程序编译为二进制机器码；
4. 掌握多周期CPU的测试方法;
5. 掌握多周期CPU的实现方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：（**说明：操作码按照以下规定使用，都给每类指令预留扩展空间，后续实验相同。**）

==>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

（2）sub rd, rs, rt

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

完成功能：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**

==>逻辑运算指令

（4）or rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd<－rs | rt

（5）and rd, rs, rt

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

功能：rd<－rs & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

功能：rt<－rs | (zero-extend)**immediate**

==>移位指令

（7）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

（8） slt　rd, rs, rt　 带符号数

100110	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

（9） slti　rt, rs,immediate　　带符号

100111	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)

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

功能：memory[rs+ (sign-extend)**immediate**]<－rt。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)

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

功能：rt <－ memory[rs + (sign-extend)**immediate**]。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** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110100	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

(13) bne　rs,rt, **immediate** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110101	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

（14）bgtz　rs,**immediate**

110110	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

（15）j　addr

111000	addr[27:2]

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

（16）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（17）jal　addr

111010	addr[27..2]

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],0,0}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（18）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变 pc 的值，pc 保持不变。

三.实验原理

1、设计原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段：

(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。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

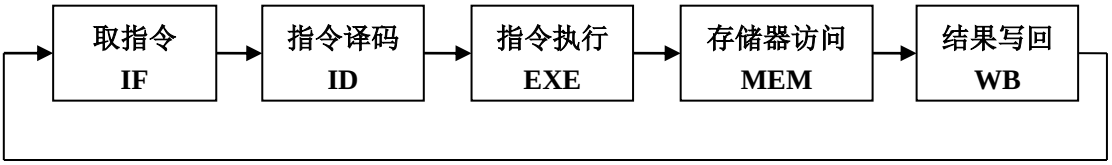
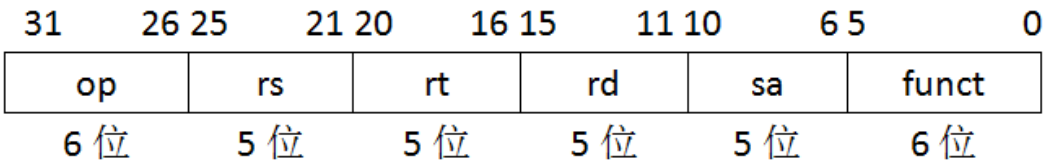


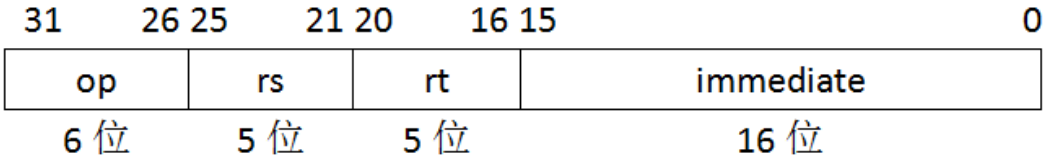
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式:

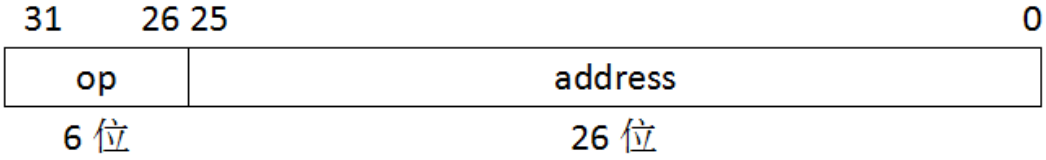
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

rt：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：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；

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。

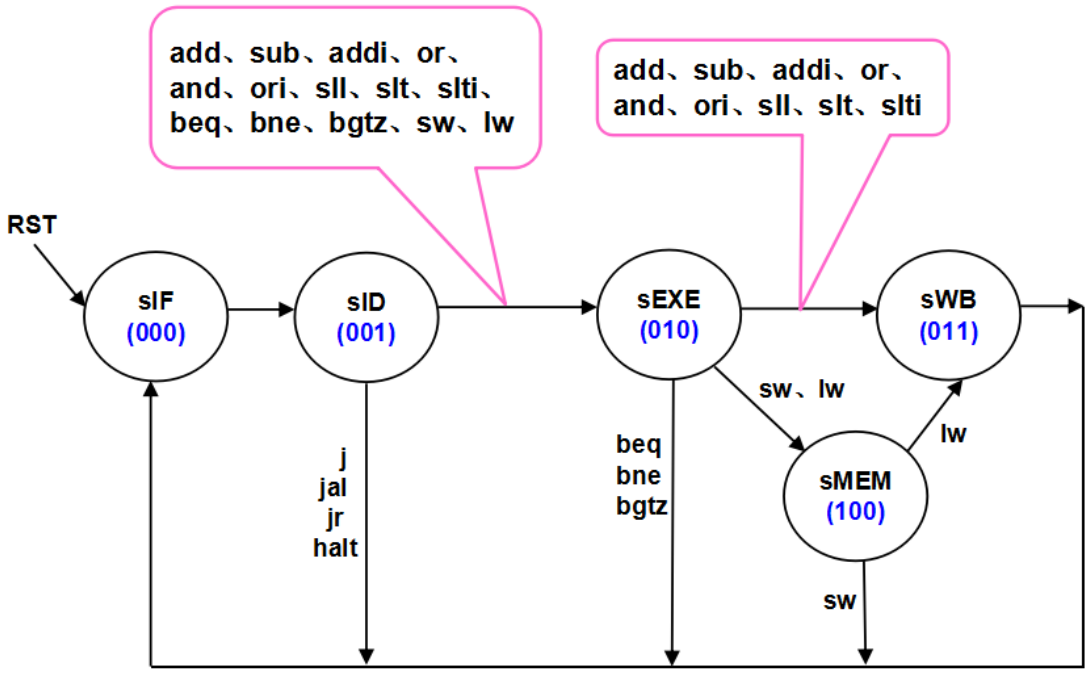


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 sIF 状态转移到 sID 就是无条件的；有些是有条件的，例如 sEXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

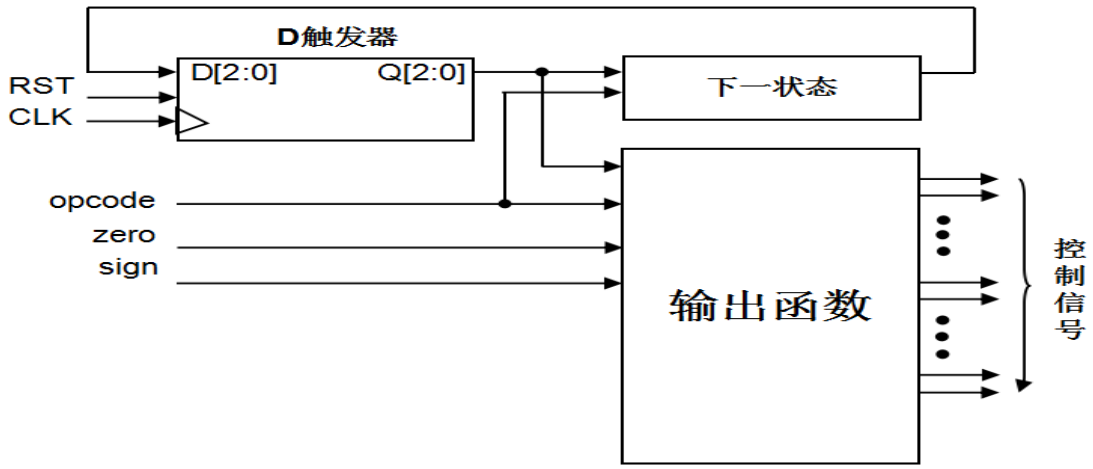


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

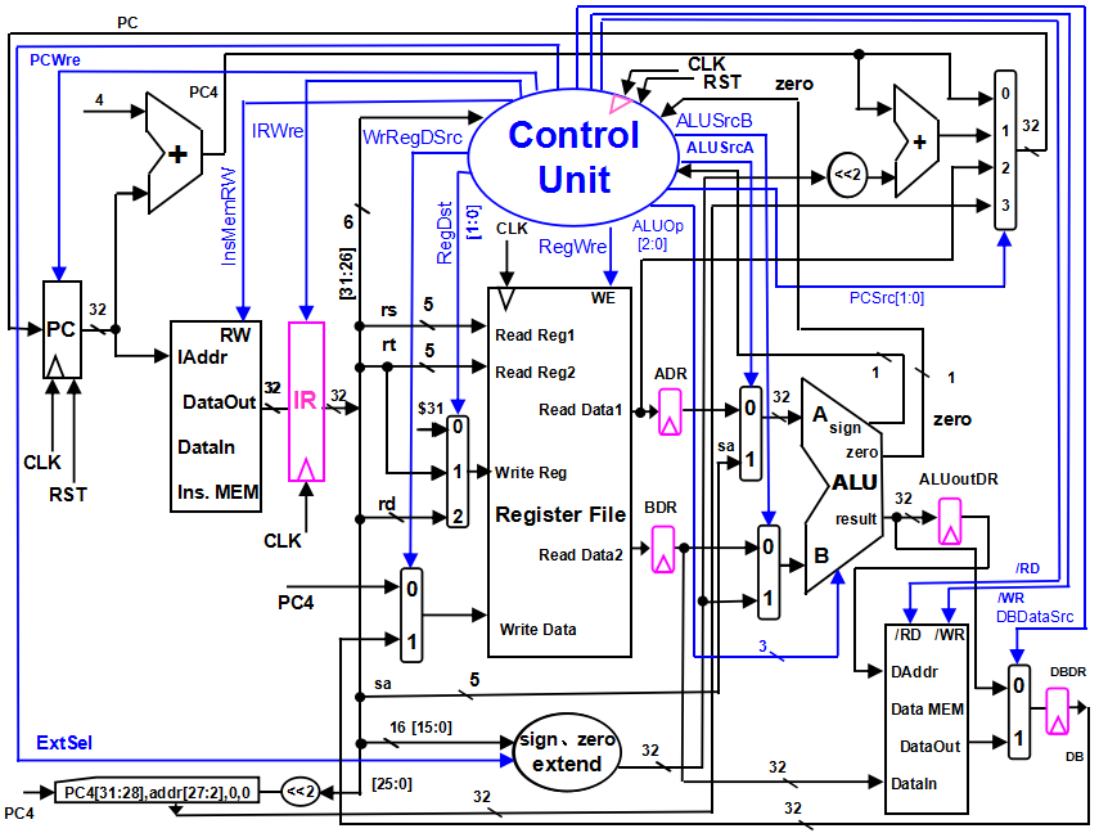


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 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，给出寄存器地址（编号），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。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号，**其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1　控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、bgtz、slt、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	来自 sign 或 zero 扩展的立即数，相关

	令：add、sub、or、and、beq、bne、bgtz、slt、sll	指令：addi、ori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出,相关指令：add、sub、addi、or、and、ori、slt、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、bgtz、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、slti、sll、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4），相关指令：jal，写 $31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据，相关指令：add、addi、sub、or、and、ori、slt、slti、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	存储器输出高阻态
/WR	写数据存储器，相关指令：sw	无操作
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后，这个信号也接着发出，在时钟上升沿，IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)**immediate**，相关指令：ori;	(sign-extend)**immediate**，相关指令：addi、lw、sw、beq、bne、bgtz;
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、slti、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，且 zero=0); 10：pc<－rs，相关指令：jr; 11：pc<－{pc[31:28],addr[27:2],0,0}，相关指令：j、jal;	
RegDst[1..0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addi、ori、slti、lw; 10：rd 字段，相关指令：add、sub、or、and、slt、sll; 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，指令地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

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，数据地址输入端口

DataIn，存储器数据输入端口
DataOut，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，其地址输入端口（rt、rd）
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

IR：　指令寄存器，用于存放正在执行的指令代码

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= (A<B) ?1: 0	比较 A 与 B 不带符号
011	if (A<B && (A[31] == B[31]))　Y = 1; else if (A[31] ==1 && B[31] == 0)　Y = 1; else　Y = 0;	比较 A 与 B 带符号
100	Y = B<<A	B 左移 A 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
111	Y = A ⊕ B	异或

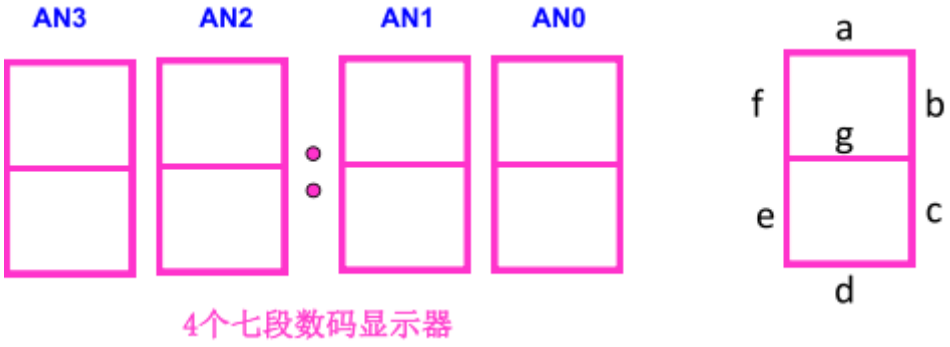
2、显示原理

分频：所谓“分频”，就是把输入信号的频率变成成倍地低于输入频率的输出信号。文献资料上所谓用计数器的方法做“分频器”的方法，只是众多方法中的一种。它的原理是：把输入的信号作为计数脉冲，由于计数器的输出端口是按一定规律输出脉冲的，所以对不同的端口输出的信号脉冲，就可以看作是对输入信号的“分频”。至于分频频率是怎样的，由选用的计数器所决定。如 果是十进制的计数器那就是十分频，如果是二进制的计数器那就

是二分频，还 有四进制、八进制、十六进制等等。以此类推。

在 Vivado中，clock 在 W5引脚的默认频率是一亿赫兹，则需进行时钟分频至190hz进行扫描显示，一次只显示一位数字，但由于190hz非常大，每次显示的时间差非常小，所以可以有四位数同时显示的效果。

七段阴极数码管中，a~g的暗亮可以组成各个数的表示（如图所示），即可通过控制每个数a~g的暗亮来将二进制数直观地表达为十六进制数。



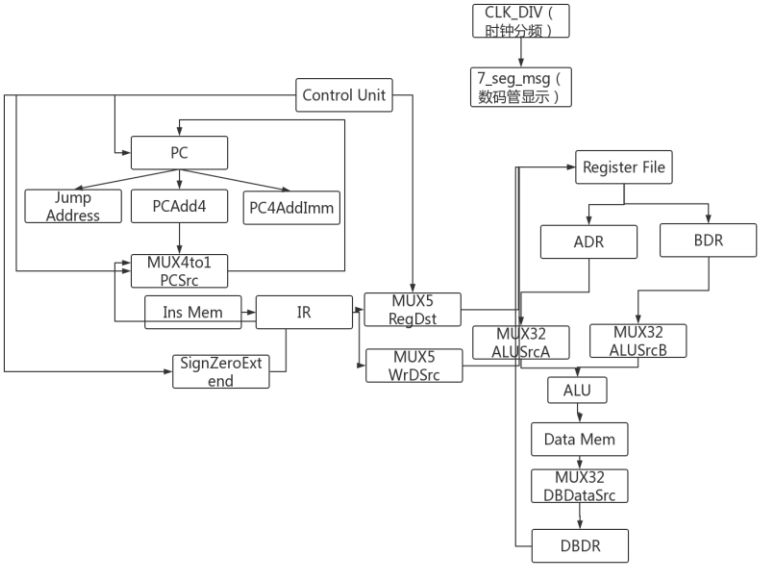
四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

1、实验过程：

①遵循单周期CPU的数据通路图，在顶层模块中分各个模块，设计图如下：

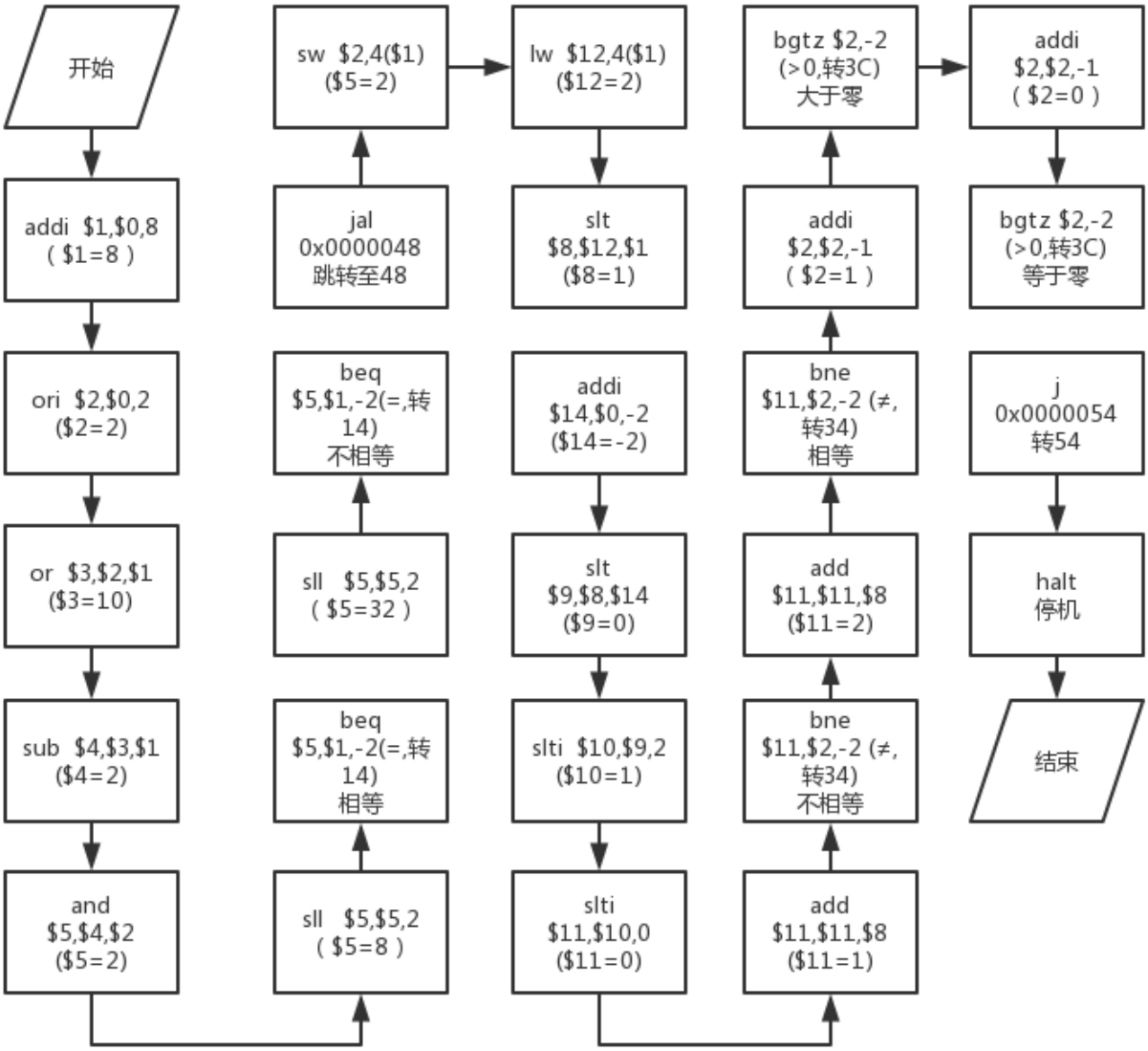


②填写控制信号与指令关系表：

		PCWre	ExtSel	InsMemRW	IRWre	WrRegDSr	RD	WR	DBDataSrc	ALUSrcA	ALUSrcB	RegWre	RegDst	PCSrc	ALUOp
IF		1	1	1	1	0	0	1	0	0	1	0	00	11	000
ID	add	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	sub	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	addi	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	or	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	and	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	ori	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	sll	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	slt	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	slti	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	beq	1	1	1	0	0	0	1	0	0	0	0	00	00	000
	bne	1	1	1	0	0	0	1	0	0	0	0	00	00	000
	bgtz	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	sw	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	lw	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	j	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	jal	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	jr	1	1	1	0	0	0	1	0	0	0	1	00	10	000
	halt	0	1	1	0	0	0	1	0	0	0	0	00	11	000
	add	1	1	1	0	0	0	1	0	0	0	0	00	11	000
	sub	1	1	1	0	0	0	1	0	0	0	0	00	11	001
	addi	1	1	1	0	0	0	1	0	0	1	0	00	11	000
	or	1	1	1	0	0	0	1	0	0	0	0	00	11	101
EXE	and	1	1	1	0	0	0	1	0	0	0	0	00	11	110
	ori	1	0	1	0	0	0	1	0	0	1	0	00	11	101
	sll	1	1	1	0	0	0	1	0	1	1	0	00	11	100
	slt	1	1	1	0	0	0	1	0	0	0	0	00	11	100
	slti	1	1	1	0	0	0	1	0	0	0	0	00	11	001
	beq	1	1	1	0	0	0	1	0	0	0	0	00	11	001
	bne	1	1	1	0	0	0	1	0	0	0	0	00	11	001
	sw	1	1	1	0	0	0	1	0	0	1	0	00	11	000
	lw	1	1	1	0	0	0	1	0	0	1	0	00	11	000
	bgtz	1	1	1	0	0	0	1	0	0	0	0	00	11	001
WB	add	1	1	1	0	1	0	1	0	0	0	1	10	00	000
	sub	1	1	1	0	1	0	1	0	0	0	1	10	00	000
	addi	1	1	1	0	1	0	1	0	0	0	1	01	00	000
	or	1	1	1	0	1	0	1	0	0	0	1	10	00	000
	and	1	1	1	0	1	0	1	0	0	0	1	10	00	000
	ori	1	1	1	0	1	0	1	0	0	0	1	01	00	000
	sll	1	1	1	0	1	0	1	0	0	0	1	10	00	000
	slt	1	1	1	0	1	0	1	0	0	0	1	10	00	000
	slti	1	1	1	0	1	0	1	0	0	0	1	10	00	000
	sw	1	1	1	0	1	1	0	0	0	0	1	01	00	000
	lw	1	1	1	0	1	0	1	0	0	0	1	01	00	000
MEM	sw	1	1	1	0	0	0	1	0	0	0	0	00	00	000
	lw	1	1	1	0	0	0	1	1	0	0	0	00	00	000

注：本表所有x的信号皆按方便之故取了固定值。

③画出测试代码的流程图：



④各模块设计：

a. controlunit：

输入信号：CLK ,时钟控制 RST，清零控制

in，即opcode

zero，零标志位，用于判断运算结果是否为0

sign，符号位，用于判断符号正负

输出信号：reg InsMemRW, PCWre, ExtSel, DBDataSrc, WR, RD, ALUSrcA, ALUSrcB, RegWre, RegDst,[2:0]PCSrc,[1:0]ALUOp。具体释义见实验原理中的信号解释表。 curstate:当前指令所在状态。

【control unit】关键代码：

controlunit分为两段。

第一段是状态机的编写，这也是与单周期CPU的controlunit最不同的地方，根据当前的状态和通过读取Opcode判断下一个状态是什么。代码如下：

```
always @(posedge CLK or negedge RST) begin
        if (RST == 0) curstate <= 3'b000;
        else if (curstate == 3'b000) curstate <= 3'b001;
        else if (curstate == 3'b001) begin
            if (opcode == 6'b111000 || opcode == 6'b111010 || opcode == 6'b111001 || opcode == 6'b111111) curstate <= 3'b000;
            else curstate <= 3'b010;
        end
        else if (curstate == 3'b010) begin
            if (opcode == 6'b110100 || opcode == 6'b110101 || opcode == 6'b110110) curstate <= 3'b000;
            else if (opcode == 6'b110000 || opcode == 6'b110001) curstate <= 3'b100;
            else curstate <= 3'b011;
        end
        else if (curstate == 3'b100) begin
            if (opcode == 6'b110000) curstate <= 3'b000;
            else curstate <= 3'b011;
        end
        else if (curstate == 3'b011) curstate <= 3'b000;
    end
```

第二段与单周期CPU相似，是根据Opcode和当前状态设定各个指令的状态。具体代码按照前面的我列的指令关系表书写，为了避免冗余，不在这里重复贴上。

b.PC相关模块，包括PC+4模块，PC+imm模块，Jump模块和选择MUX模块。

输入信号：pcin，上一条PC的值。

in（用于jump模块），指令的25~0位。

imm（用于PC+imm模块），指令的immediate部分。

RST，用于PC的清零。

PCWre,用于PC的停止。

[1:0]PCSrc，用于Pcmux的选择。

CLK，时钟下降沿触发。

输出信号:Pcout，下一条PC的值。

【PC模块】相关代码：

```
always @(posedge CLK or negedge Reset) begin     //PC 总模块
      if (Reset == 0) PCout <= 0;
      else if (PCWre) PCout <= PCin;
end
always @(PC4 or imm) begin       //PC+imm 模块：
      PC4_imm = PC4 + (imm << 2);
end
always @(PCin) begin       //PC+4 模块
     PCout = PCin + 4;
end
always @(PC4 or PC4_imm or jump or PCSrc or Reg_Data) begin
      if (PCSrc == 2'b000) PCout = PC4;
      else if (PCSrc == 2'b001) PCout = PC4_imm;
      else if (PCSrc == 2'b011) PCout = jump;
      else PCout = Reg_Data;
end
```

//注意此处与单周期 CPU 不同，它为 jal、jr 指令设置了从寄存器中读取下一条指令的代码的机制，所以选择器增加了一个选择方向。

c.InsMem模块

输入信号：[31:0] addr,32位机器码（本实验中从文件中读取）

RW，控制读写寄存器。

输出信号：[31:0]dataout，读取出的机器码输送出去。

【InsMem模块】相关代码：

```
reg [7:0] rom [99:0];
initial begin
      $readmemb ("C:/new/test.txt", rom);   //根据文件存放地址适当修改读取地址
end
always @(addr) begin   //为 0，读存储器，大端数据存储模式
            dataOut[31:24] = rom[addr];
            dataOut[23:16] = rom[addr+1];
            dataOut[15:8] = rom[addr+2];
            dataOut[7:0] = rom[addr+3];
end
```

d.RegFile及其相关选择模块

输入信号:

CLK,时钟触发。

RST,寄存器清零。

RegWre,是否写寄存器。

RegDst（用于读取选择模块），选择读取哪个寄存器。

[4:0] ReadReg1,读寄存器1地址。

[4:0] ReadReg2,读寄存器2地址。

[4:0] WriteReg,写寄存器。

[31:0] WriteData,往寄存器中写入数据。

输出信号:

[31:0] ReadData1,输出寄存器1的数据。

[31:0] ReadData2，输出寄存器2的数据。

【RegFile及其选择模块】相关代码

```
//输入寄存器类型的选择
    always @(RegDst or rt or rd) begin
        if (RegDst == 1) Write_Reg = rd;
        else Write_Reg = rt;
    end
//寄存器组
reg [31:0] regFile[1:31]; //寄存器必须用 reg 类型
    integer i;
    assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1]; //读寄存器数据
    assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
    wire clk1;
    always @ (negedge CLK or negedge RST) begin //必须用时钟边沿触发
        if (RST==0) begin
            for(i=1;i<32;i=i+1) begin
                if(i==WriteReg)    regFile[WriteReg] <= WriteData;
                else regFile[i] <= 0;
            end
        end
        else if(RegWre == 1 && WriteReg != 0) // WriteReg != 0，$0  寄存器不能修改
            regFile[WriteReg] <= WriteData; //  写寄存器
    end
```

e.ALU及其相关选择模块

输入信号：

[2:0] ALUopcode,根据aluopcode选择相应计算，具体见前表。

[31:0] rega,读入的第一个需要计算的数据。

[31:0] regb,读入的第二个需要计算的数据。

[31:0] Read_Data1,Read_Data2用于读取数据。（用于ALU选择模块）

[4:0] sa,偏移位，用于sll指令。（用于ALUA选择模块）

ALUSrcA,选择是用寄存器还是偏移量作为计算数据。（用于ALUA选择模块）

ALUSrcB,选择是用寄存器还是符号扩展为计算数据。（用于ALUB选择模块）

输出信号：

[31:0] result,计算结果。

zero,零标志位的结果。

sign，符号位的结果。

【ALU及其选择模块】相关代码

```
//第一个计算数的选择
    always @(ALUSrcA or sa or Read_Data1) begin
        if (ALUSrcA == 0) ALUA = Read_Data1;
        else ALUA = {{27{1'b0}}, sa};
    end
//第二个计算数的选择
    always@(ALUSrcB or Read_Data2 or Extend_out) begin
        if (ALUSrcB == 1) ALUB = Extend_out;
        else ALUB = Read_Data2;
    end
//ALU 部分代码
assign zero = (result==0)?1:0;
    assign sign = (result[31] == 1)?1:0;
    always @( ALUopcode or rega or regb ) begin
        case (ALUopcode)
            3'b000 : result = rega + regb;
            3'b001 : result = rega - regb;
            3'b010 : result = regb << rega;
            3'b100 : result = rega & regb;
            3'b011 : result = rega | regb;
            3'b101 : result = (rega < regb)?1:0; //  不带符号比较
            3'b110 : begin //  带符号比较
```

```
                    if (rega<regb &&(( rega[31] == 0 && regb[31]==0) || (rega[31] == 1 &&
regb[31]==1))) result = 1;
                    else if (rega[31] == 0 && regb[31]==1) result = 0;
                    else if ( rega[31] == 1 && regb[31]==0) result = 1;
                    else result = 0;
                    end
                3'b111 : begin
                    result = rega ^ regb;
                end
            endcase
        end
```

f.DataMem及其相关选择模块

输入信号:

CLK，用于时钟触发。

nRD，判断是否读取存储器。

nWR，判断是否写入存储器

[31:0] address，写入的地址。

[31:0]writeData,存储器的内存单元。

DBDataSrc（用于选择模块），选择输出到DB的数据是ALU还是内存里。

输出信号:

[31:0]dataout，读取出的数据。

[31:0]DB，输出给数据总线的数据。

【DataMem及其选择模块】相关代码

```
//DataMem 模块
    reg [7:0] ram [0:60]; //写寄存器
    assign Dataout[7:0] = (nRD==0)?ram[address + 3]:8'bz;
    assign Dataout[15:8] = (nRD==0)?ram[address + 2]:8'bz;
    assign Dataout[23:16] = (nRD==0)?ram[address + 1]:8'bz;
    assign Dataout[31:24] = (nRD==0)?ram[address ]:8'bz;
    always@(negedge CLK) begin   //读寄存器
        if( nWR==0 ) begin
            ram[address] <= writeData[31:24];
            ram[address+1] <= writeData[23:16];
```

```
                ram[address+2] <= writeData[15:8];
                ram[address+3] <= writeData[7:0];
            end
        end


//DB 选择模块
        always @(ALUresult or DataOut or DBDataSrc) begin
            if (DBDataSrc == 1) DB = DataOut;
            else DB = ALUresult;
        end
```

g.符号扩展模块

输入信号:

[15:0] in,指令低16位。

ExtSel,选择是符号扩展还是0扩展

输出信号:

[31:0]out，输出的扩展后的数。

【符号扩展】相关代码

```
        always @(in or ExtSel) begin
            if (ExtSel == 0) out =    {{16{1'b0}}, in};
            else if (in[15] == 0) out = {{16{1'b0}}, in};
            else out = {{16{1'b1}}, in};
        end
```

另附：下面几个模块是与单周期不同的地方，它添加了IR、ALUoutDR、ADR、BDR、DBDR的寄存器模块，以实现多周期CPU的不同状态时各个器件的运行稳定。代码如下:

h.IR模块

输入信号:

InsOut,读取指令后的输入

IRWre,控制是否输出

CLK，时钟周期

输出信号：

op, rs, rt, rd, sa, immediate,addressJmp，在此模块中完成对指令的译码。

【IR译码模块】相关代码

```
always @(posedge CLK) begin
    if (IRWre == 1) begin
        op[5:0] <= InsOut[31:26];
        rs[4:0] <= InsOut[25:21];
        rt[4:0] <= InsOut[20:16];
        rd[4:0] <= InsOut[15:11];
        sa[4:0] <= InsOut[10:6];
        immediate[15:0] <= InsOut[15:0];
        addressJmp[25:0] <= InsOut[25:0];
    end
end
```

i.ADR、BDR模块

输入信号：

Readdata1，Readdata2，ALU数据。

CLK，时钟周期

输出信号：

readdata1、readdata2，对ALU数据进行寄存器存储，等待时钟信号到来。

【ADR、BDR模块】相关代码

```
always @(posedge CLK) readdata1<= Read_Data1;//ADR 主体
always @(posedge CLK) readdata2 <= Read_Data2;//BDR 主体
```

j.ALUoutDR模块

输入信号：

ALUresult，ALU计算结果。

输出信号：

ALUout，对结果进行暂时存储并等待时钟信号到来。

【ALUoutDR模块】相关代码

```
always @(posedge CLK) ALUout <= ALUresult;
```

k.DBDR模块

输入信号：

DBData，数据总线输出结果。

输出信号：

DB，对结果进行暂时存储并等待时钟信号到来。

【DBDR模块】相关代码

```
always @(posedge CLK) DB <= DBData;
```

显示模块部分：

时钟分频

```
reg [40:0] q;
always @ (posedge clk) begin
    q <= q + 1;
end
assign clk190 = q[17]; // 190 Hz 用于扫描输出
```

七段码输出

```
reg [1:0] s;
reg [3:0] digit;
wire [3:0] aen;
assign aen = 4'b1111; //一共四位数码管，每一位显示什么数字
always @ (* )
    case (s)
        0: //数码管最低位
        begin
            case (change)
                2'b00:digit=pcnext2;
                2'b01:digit=rsdata2;
                2'b10:digit=rtdata2;
                2'b11:digit=dbdata2;
            endcase
        end
        1:
        begin
            case (change)
```

```
                    2'b00:digit=pcnext1;
                    2'b01:digit=rsdata1;
                    2'b10:digit=rtdata1;
                    2'b11:digit=dbdata1;
                endcase
            end
            2:
            begin
                case (change)
                    2'b00:digit=pcpre2;
                    2'b01:digit=rsaddr2;
                    2'b10:digit=rtaddr2;
                    2'b11:digit=result2;
                endcase
            end
            3:
            begin
                case (change)
                    2'b00:digit=pcpre1;
                    2'b01:digit=rsaddr1;
                    2'b10:digit=rtaddr1;
                    2'b11:digit=result1;
                endcase
            end
        endcase
    always @ ( * )
        case (digit)
            4'b0000 : dispcode = 8'b1100_0000; //0 '0'-亮灯，'1'-熄灯
            4'b0001 : dispcode = 8'b1111_1001; //1
            4'b0010 : dispcode = 8'b1010_0100; //2
            4'b0011 : dispcode = 8'b1011_0000; //3
            4'b0100 : dispcode = 8'b1001_1001; //4
            4'b0101 : dispcode = 8'b1001_0010; //5
            4'b0110 : dispcode = 8'b1000_0010; //6
            4'b0111 : dispcode = 8'b1101_1000; //7
            4'b1000 : dispcode = 8'b1000_0000; //8
            4'b1001 : dispcode = 8'b1001_0000; //9
            4'b1010 : dispcode = 8'b1000_1000; //A
            4'b1011 : dispcode = 8'b1000_0011; //b
            4'b1100 : dispcode = 8'b1100_0110; //C
            4'b1101 : dispcode = 8'b1010_0001; //d
            4'b1110 : dispcode = 8'b1000_0110; //E
            4'b1111 : dispcode = 8'b1000_1110; //F
            default : dispcode = 8'b0000_0000;
```

```
        endcase
    always @ (posedge clk190)
        begin //每次 190HZ 的频率闪过就选择亮哪一盏灯
            s<=s+1;
        end

     //确保每个数码管上都有显示数字
    always @ ( *)
    begin
        an = 4'b1111; //共阴极，全部不亮
        if (aen[s] == 1) //控制哪一位亮
            an[s] = 0;
    end
```

⑤顶层模块设计，调用各个模块代码即可。

⑥运行VIVADO，进行仿真、综合和烧录，对比输入机器码的文件观察各个数据。具体操作不再赘述。

2、实验结果

i.仿真结果

1.addi $1,$0,8

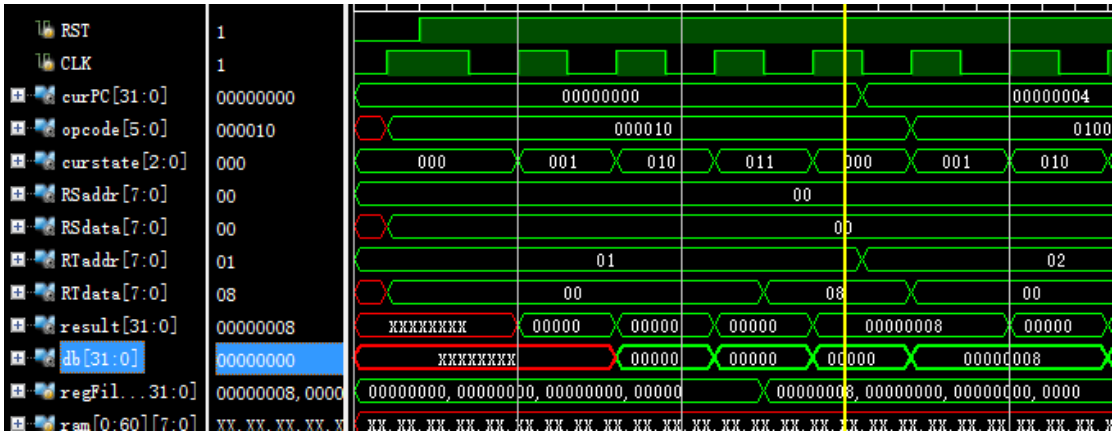
由图可知此指令要经历4个阶段。

IF：当前PC:00 下条PC:04

ID：读出opcode为000010， rs编号：0 rt编号：1，执行加法操作。

EXE：ALU执行加法操作。

WB：成功计算结果并写入1号寄存器中（$1=8），符合实验预期。



2.ori　$2,$0,2

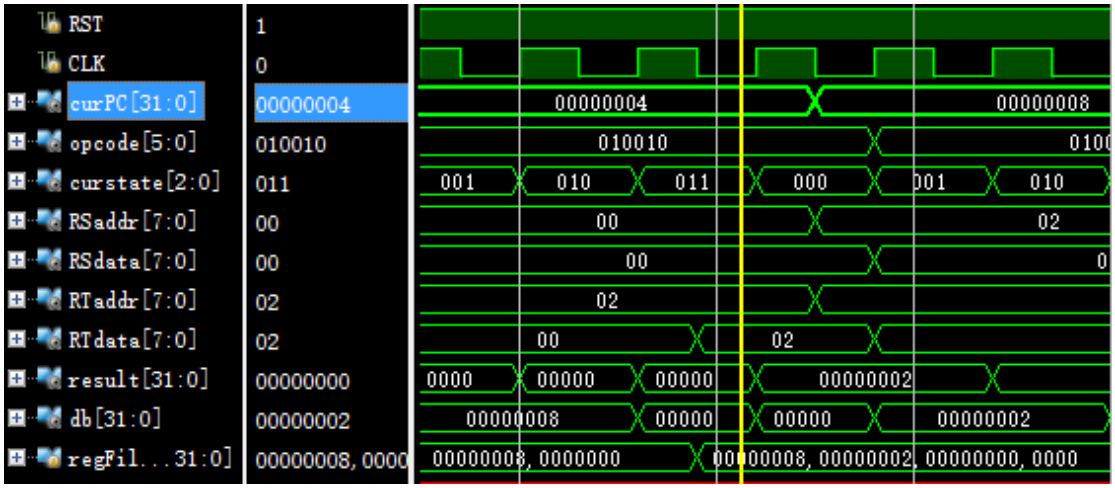
由图可知此指令要经历4个阶段。

IF：当前PC:04　下条PC:08

ID：读出opcode为010010，　rs编号：0 rt编号：2，执行或操作。

EXE：ALU执行或操作。

WB：成功计算结果并写入2号寄存器中（$2=2），符合实验预期。



3.or　$3,$2,$1

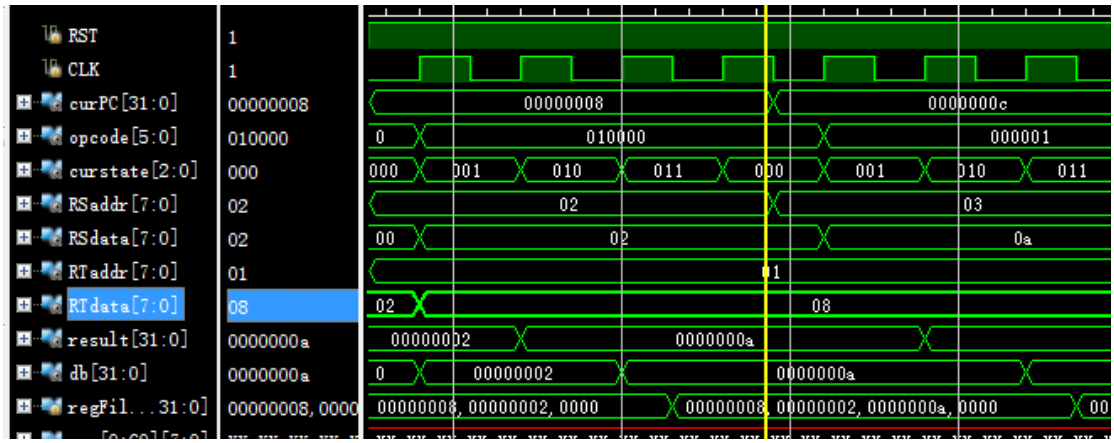
由图可知此指令要经历4个阶段。

IF：当前PC:08　下条PC:0c

ID：读出opcode为010000，　rs编号：2 rt编号：1 rd编号：3，执行或操作。

EXE：ALU执行或操作。

WB：成功计算结果并写入,3号寄存器中($3=a)，符合实验预期。



4.sub $4,$3,$1

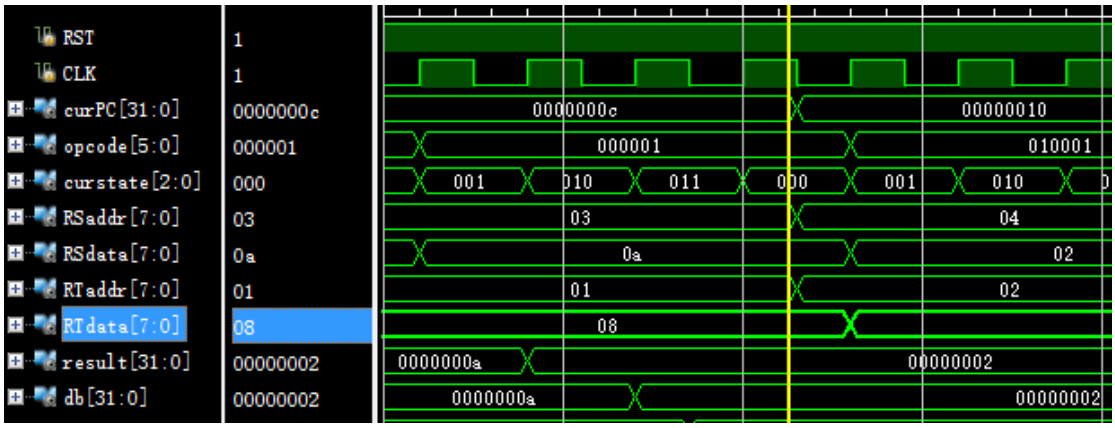
由图可知此指令要经历4个阶段。

IF：当前PC:0c 下条PC:10

ID：读出opcode为000001， rs编号：3 rt编号：1 rd编号：4，执行减操作。

EXE：ALU执行减操作。

WB：成功计算结果并写入4号寄存器中($4=2)，符合实验预期。



5.and $5,$4,$2

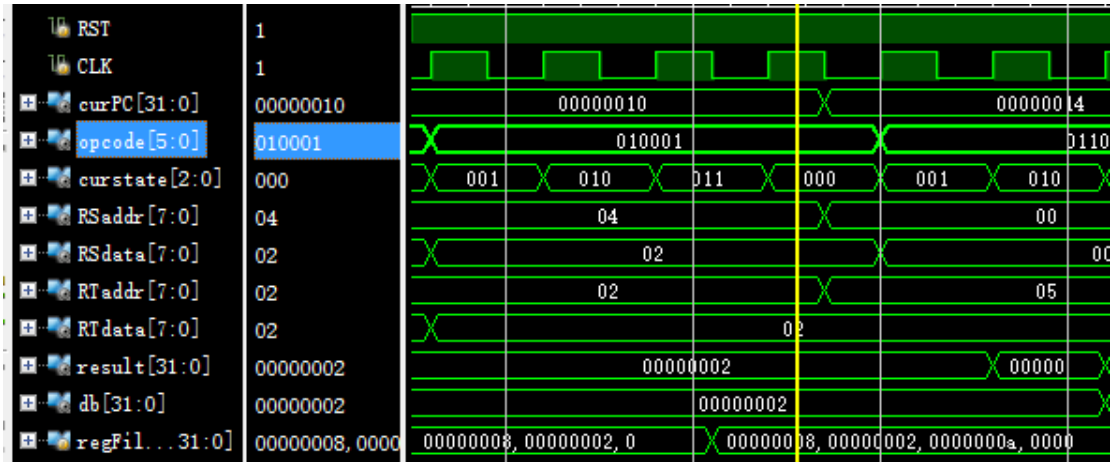
由图可知此指令要经历4个阶段。

IF：当前PC:10 下条PC:14

ID：读出opcode为010001， rs编号：4 rt编号：2 rd编号：5，执行与操作。

EXE：ALU执行与操作。

WB：成功计算结果并写入5号寄存器中（$5=2），符合实验预期。



6.sll　$5,$5,2

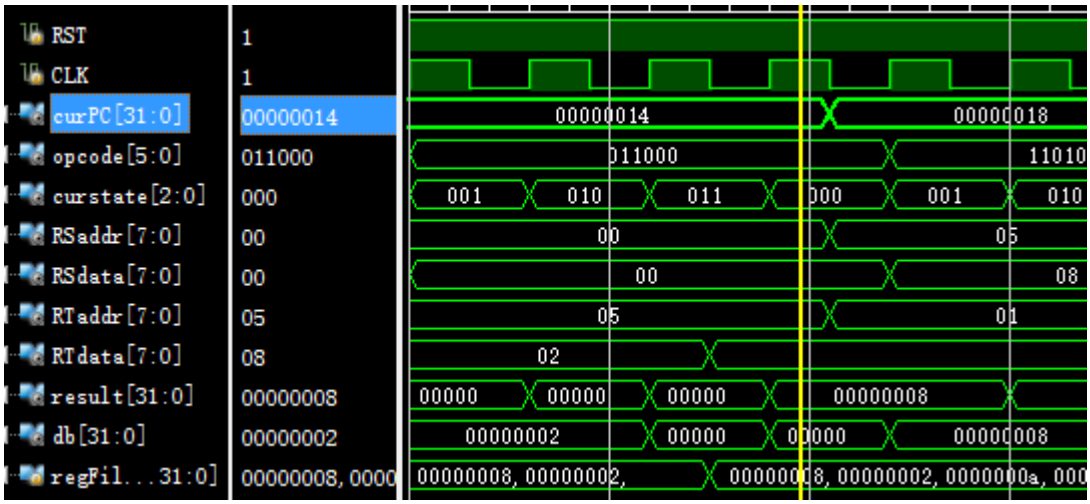
由图可知此指令要经历4个阶段。

IF：当前PC:14　下条PC:18

ID：读出opcode为011000，　rs编号：5 rt编号：5 ，执行左移操作。

EXE：ALU执行左移操作。

WB：成功计算结果并写入5号寄存器中（$5由2变为8），符合实验预期。



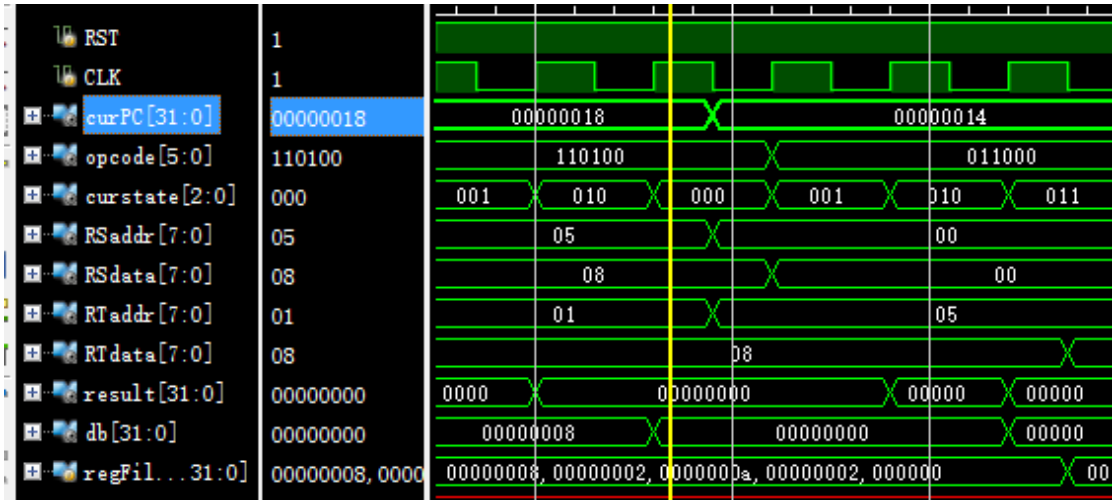
7.beq　$5,$1,-2(=,转14)

由图可知此指令要经历3个阶段。

IF：当前PC:18　下条PC:14

ID：读出opcode为110100，　rs编号：5 rt编号：1 ，执行相等则转移操作。

EXE：ALU执行减法操作，比较发现二者相等，转到pc=14处。符合实验预期



8.sll　$5,$5,2

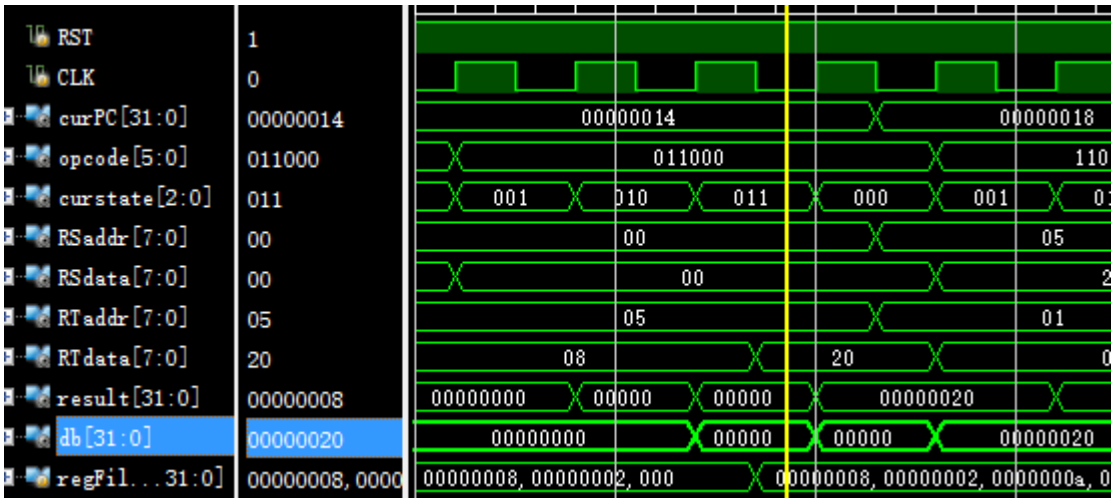
由图可知此指令要经历4个阶段。

IF：当前PC:14 下条PC:18

ID：读出opcode为011000， rs编号：5 rt编号：5 ，执行左移操作。

EXE：ALU执行左移操作。

WB：成功计算结果并写入5号寄存器中（$5由8变为20H），符合实验预期。



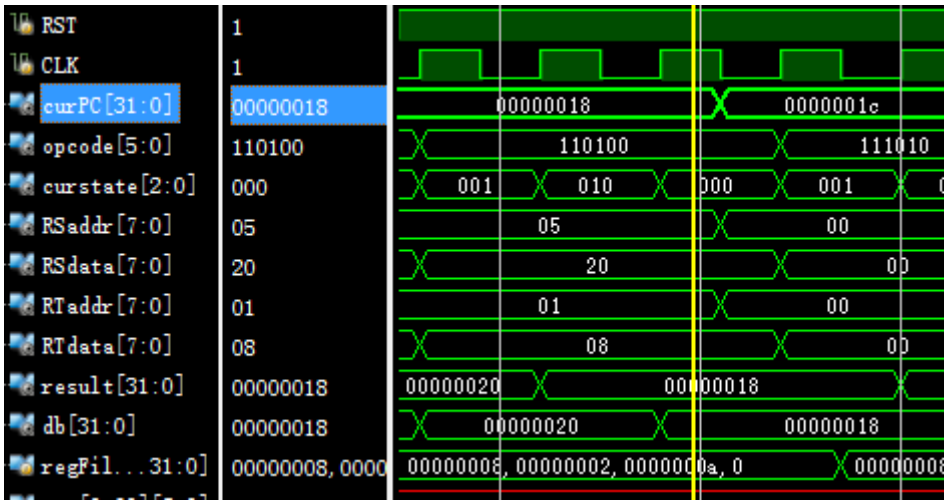
9.beq　$5,$1,-2(=,转14)

由图可知此指令要经历3个阶段。

IF：当前PC:18 下条PC:1c

ID：读出opcode为110100， rs编号：5 rt编号：1 ，执行相等则转移操作。

EXE：ALU执行减法操作，比较发现二者不相等，继续执行pc=1c处。符合实验预期。

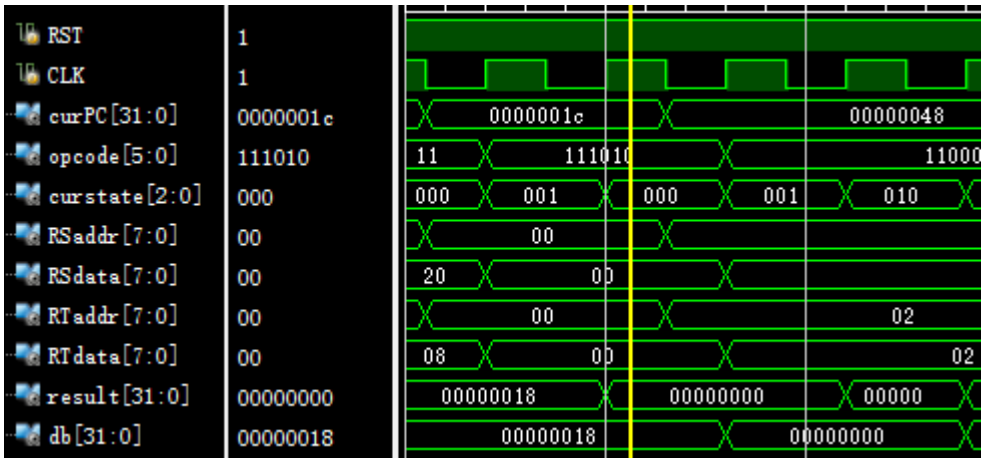


10.jal　0x0000048

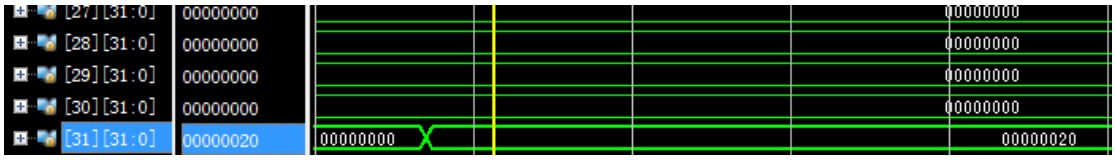
由图可知此指令要经历2个阶段。

IF：当前PC:1c 下条PC:48

ID：读出opcode为111010，将下一条PC指令存入31号寄存器，执行jal操作。



我们可以看到此时31号寄存器已经存入了1c指令的下一条指令所在的PC 20。符合实验预期。



11. sw　$2,4($1)

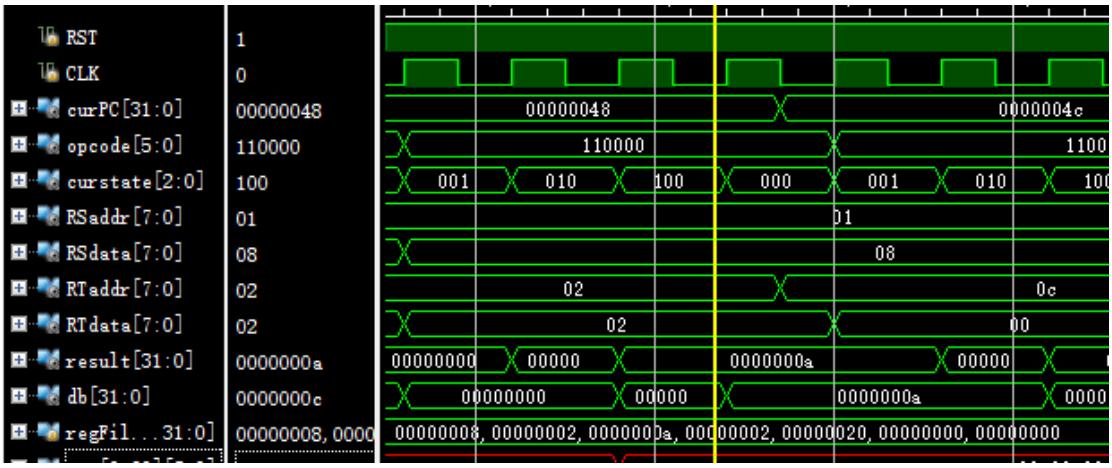
由图可知此指令要经历4个阶段。

IF：当前PC:48 下条PC:4c

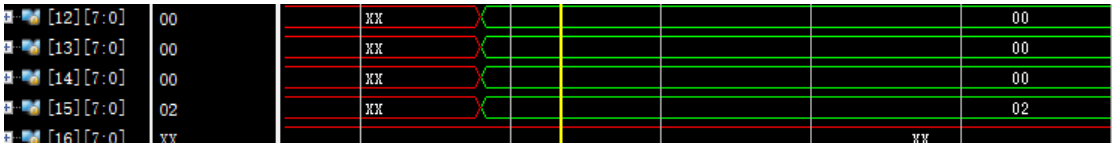
ID：读出opcode为110000， rs编号：1 rt编号：2 ，执行存数操作。

EXE：执行读寄存器操作。

MEM：执行写入内存单元操作。此时$1寄存器中显示是8号单元，加上immediate的值4即为12单元。即我们需要将2号寄存器的数写入内存单元的编号12单元。



下图我们可以看到2号寄存器的数2已经写入内存单元的编号12单元。



12. lw $12,4($1)

由图可知此指令要经历5个阶段。

IF：当前PC:4c 下条PC:50

ID：读出opcode为110001， rs编号：4 rt编号：12 ，执行取数操作。

EXE：执行读寄存器操作。

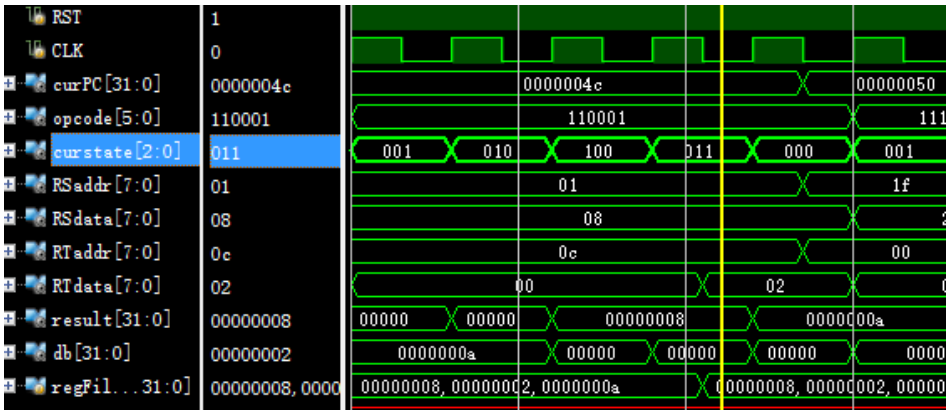
MEM：执行读取内存单元操作。

WB：执行从内存单元取数进入寄存器保存的操作。

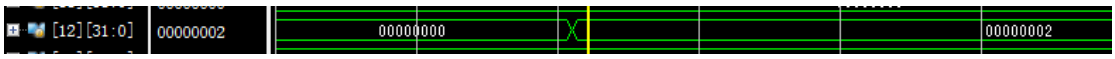
我们可以看到$1+4的内存单元就是我们刚刚操作的第12单元，第12单元已经存入了一个数2，此时将它放进12号寄存器。

同时，lw指令也是所有指令中状态最多的指令，经过了全部五个状态，同时也是利用元件最多的指令。

具体如下两张图显示：



下图我们可以看到12号寄存器已经存入了刚刚的12号内存单元存放的数2.符合实验预期。

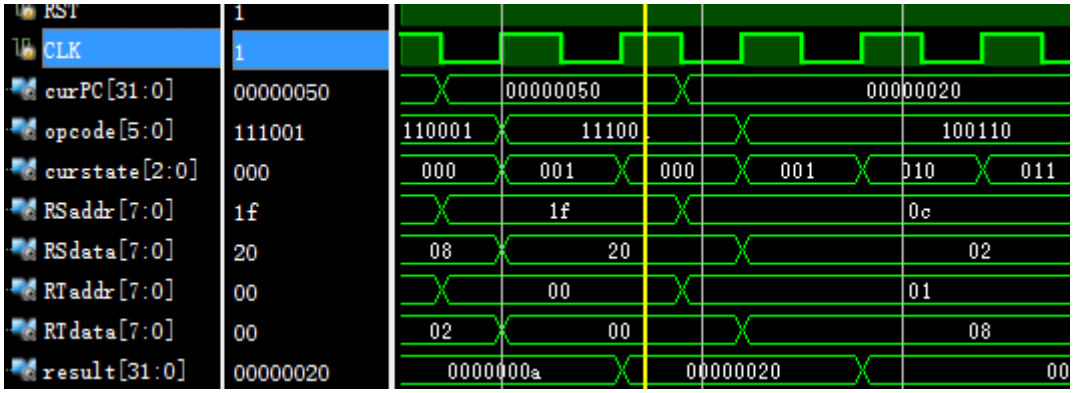


13. jr $31

由图可知此指令要经历2个阶段。

IF：当前PC:50 下条PC:20

ID:读出opcode为111001,返回jal的下一条指令,即一开始存入31号寄存器的PC=20处的指令。符合实验预期。



14. slt $8,$12,$1

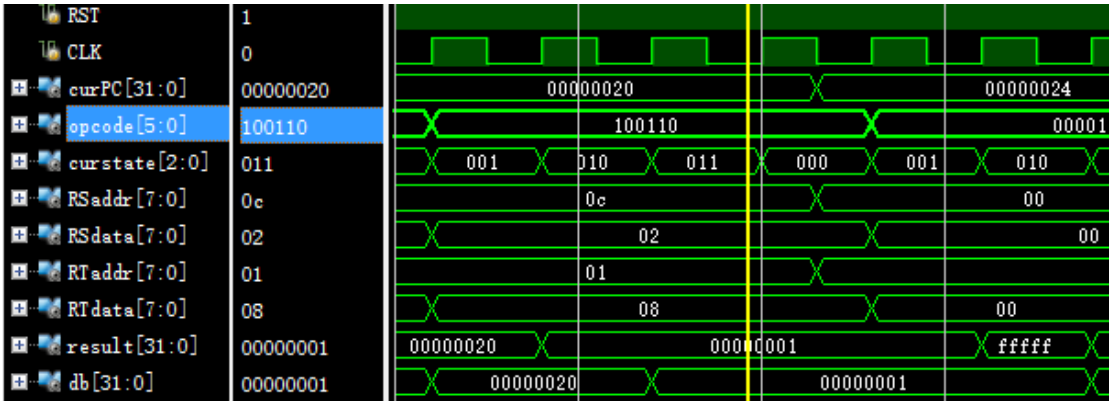
由图可知此指令要经历4个阶段。

IF：当前PC:20 下条PC:24

ID：读出opcode为100110， rs编号：12 rt编号：1 rd编号：8，执行小于则置位操作。

EXE：ALU执行减法操作进行比较。

WB：发现12号寄存器存的数小于1号寄存器存的数，8号寄存器置位，即$8=1。符合实验预期。



15. addi　$14,$0,-2

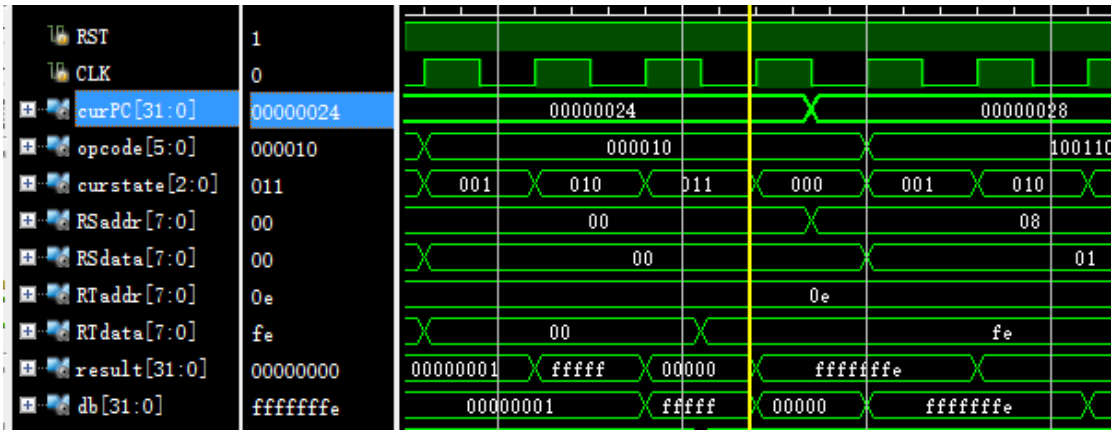
由图可知此指令要经历4个阶段。

IF：当前PC:24　下条PC:28

ID：读出opcode为000010，　rs编号：0 rt编号：14，执行加法操作。

EXE：ALU执行加法操作。

WB：将0-2=-2的值写入14号寄存器，可以发现图中显示的是-2的补码，符合实验预期。



16. slt　$9,$8,$14

由图可知此指令要经历4个阶段。

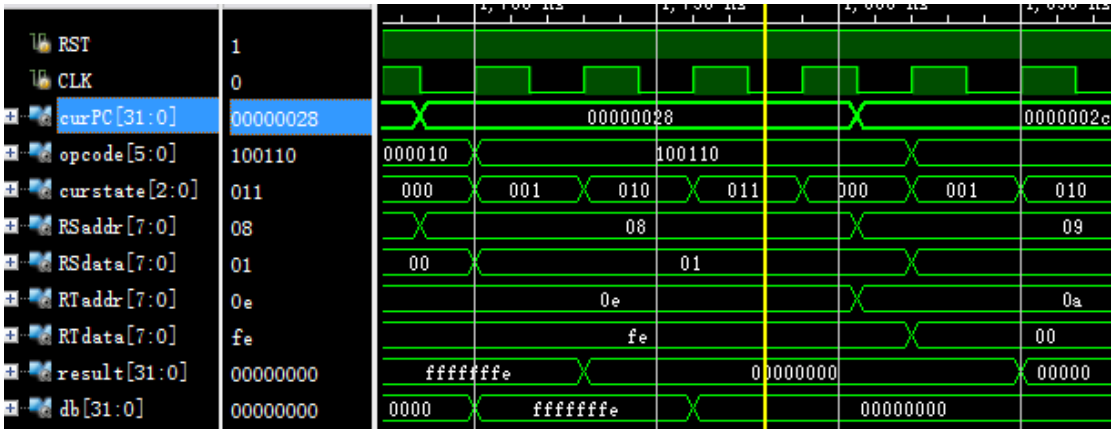
IF：当前PC:28　下条PC:2c

ID：读出opcode为100110，　rs编号：8 rt编号：14 rd编号：9，执行小于则置位操

作。

EXE：ALU执行减法操作进行比较。

WB：发现8号寄存器大于14号寄存器，$9复位，即$9=0。



17. slti　$10,$9,2

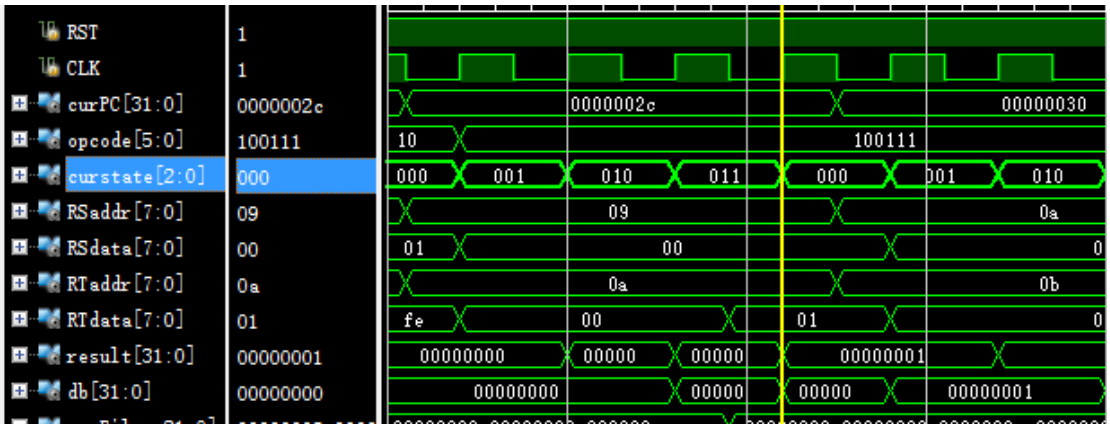
由图可知此指令要经历4个阶段。

IF：当前PC:2c　下条PC:30

ID：读出opcode为100111，　rs编号：9 rt编号：10　，执行小于则置位操作。

EXE：ALU执行减法操作进行比较。

WB：发现9号寄存器所存数小于2，则10号寄存器置位，$10=1.符合实验预期。



18. slti　$11,$10,0

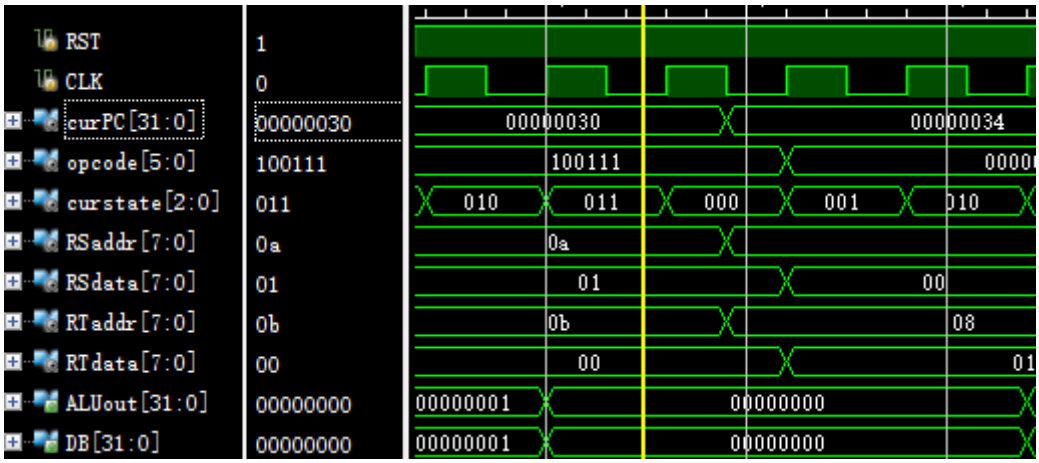
由图可知此指令要经历4个阶段。

IF：当前PC:30　下条PC:34

ID：读出opcode为100111，　rs编号：10 rt编号：11　，执行小于则置位操作。

EXE：ALU执行减法操作进行比较。

WB：发现10号寄存器所存数大于0，则11号寄存器复位，$11=0.符合实验预期。



19. add　$11,$11,$8

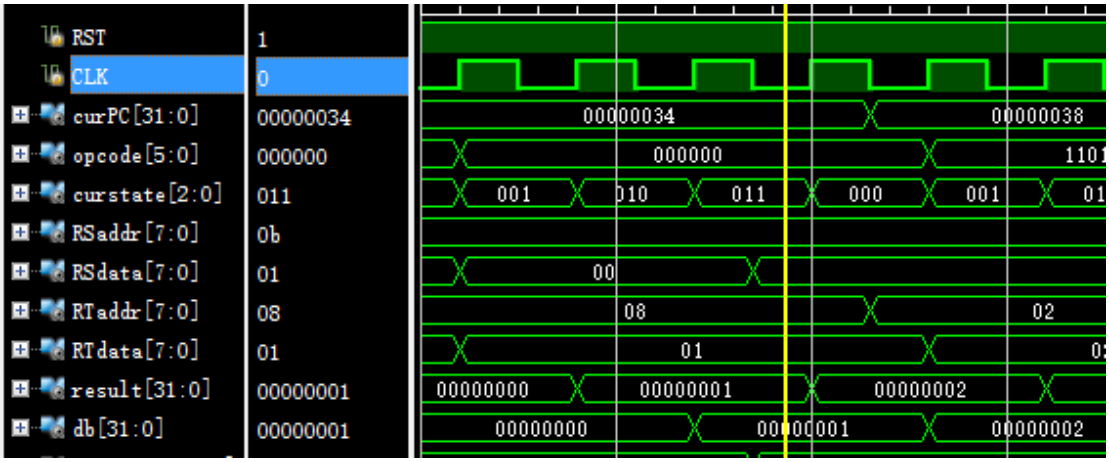
由图可知此指令要经历4个阶段。

IF：当前PC:34　下条PC:38

ID：读出opcode为000000，　rs编号：11 rt编号：8 rd编号：11，执行加法操作。

EXE：ALU执行加法操作。

WB：将计算结果存入$11（$11=0+1=1），符合实验预期。



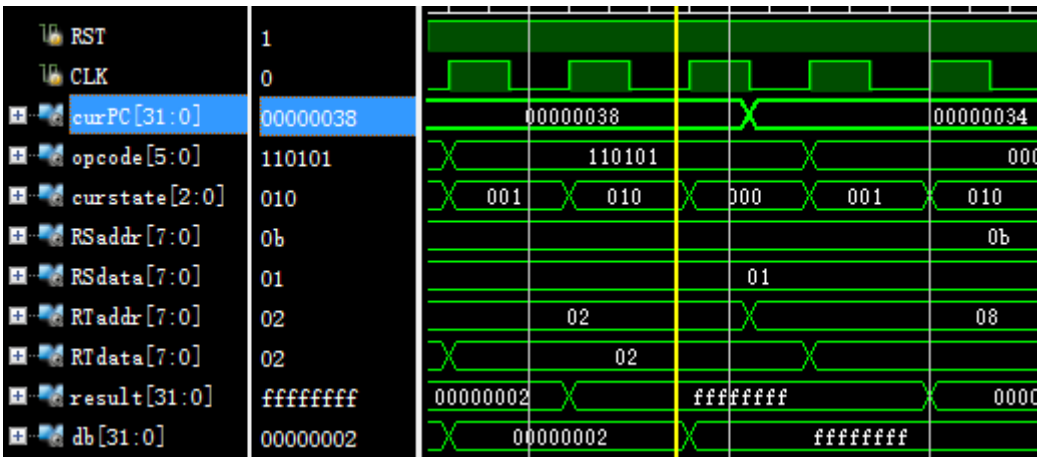
20. bne　$11,$2,-2 (≠,转34)

由图可知此指令要经历3个阶段。

IF：当前PC:38　下条PC:34

ID：读出opcode为110101，　rs编号：2 rt编号：11　，执行不等则转移操作。

EXE：ALU执行减法操作，比较发现二者不相等，转到pc=34处。符合实验预期



21. add $11,$11,$8

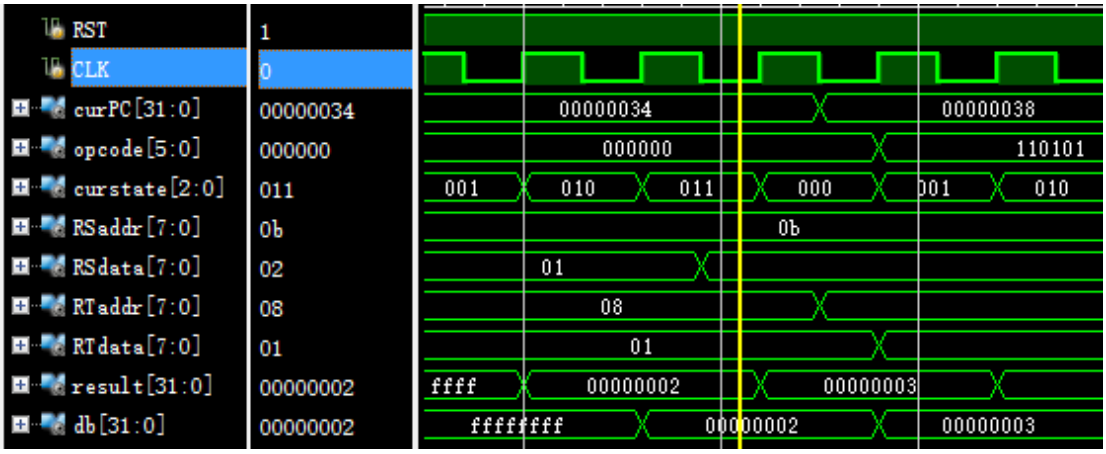
由图可知此指令要经历4个阶段。

IF：当前PC:34 下条PC:38

ID：读出opcode为000000， rs编号：11 rt编号：8 rd编号：11，执行加法操作。

EXE：ALU执行加法操作。

WB：将计算结果存入$11（$11=1+1=2），符合实验预期。



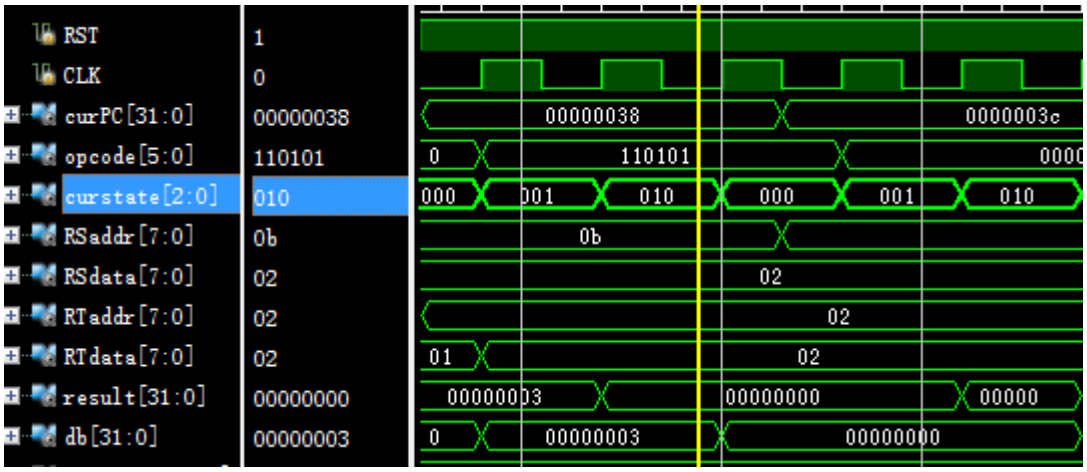
22. bne $11,$2,-2 (≠,转34)

由图可知此指令要经历3个阶段。

IF：当前PC:38 下条PC:3c

ID：读出opcode为110101， rs编号：2 rt编号：11 ，执行不等则转移操作。

EXE：ALU执行减法操作，比较发现二者相等，继续执行pc+4的指令。符合实验预期



23. addi $2,$2,-1

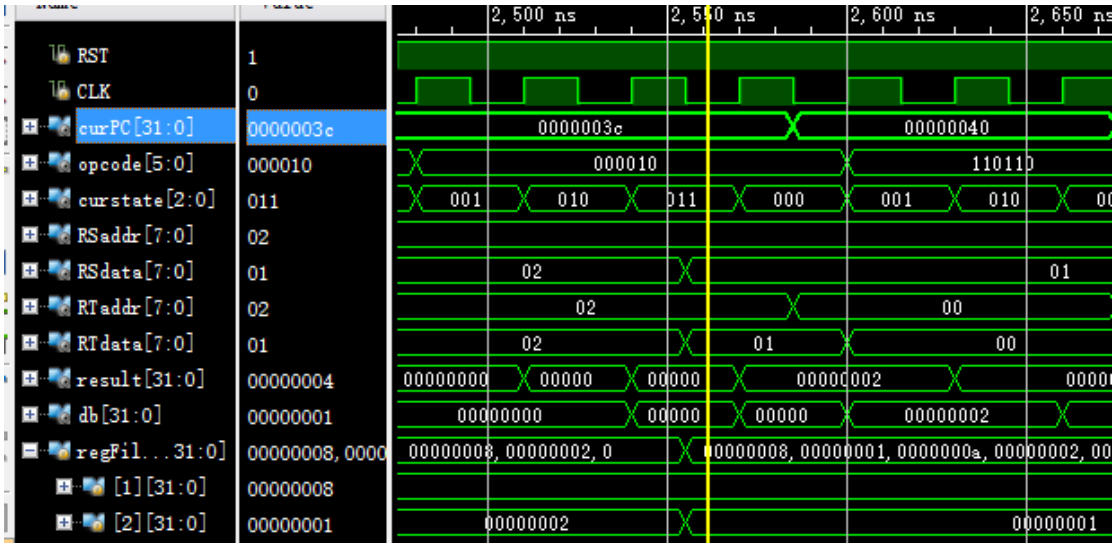
由图可知此指令要经历4个阶段。

IF：当前PC:3c 下条PC:40

ID：读出opcode为000010， rs编号：2 rt编号：2 ，执行加法操作。

EXE：ALU执行加法操作。

WB：将计算结果存入$2（可以看到db输出的数为$2=2-1=1），符合实验预期。



24. bgtz $2,-2 (>0,转3C)

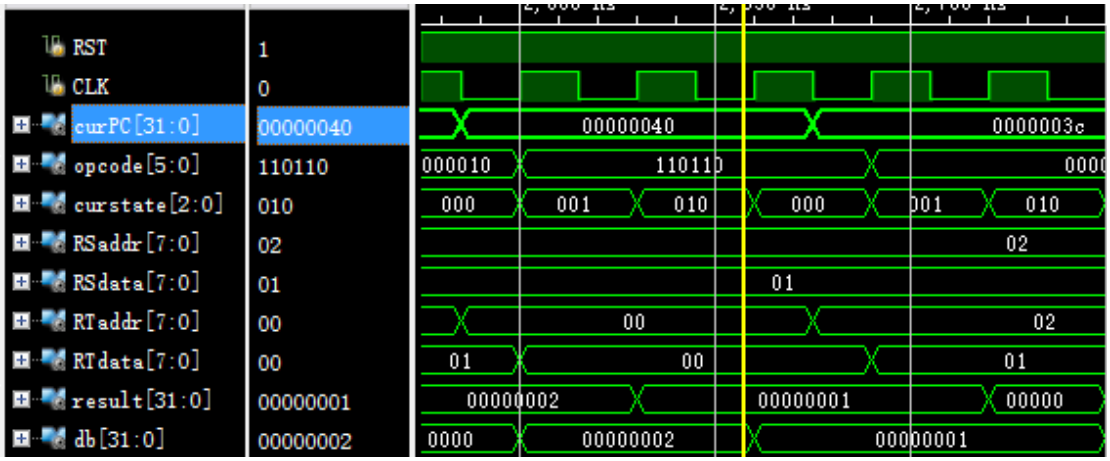
由图可知此指令要经历3个阶段。

IF：当前PC:40 下条PC:3c

ID：读出opcode为110110， 执行大于0则转移操作。

EXE：ALU执行减法操作，比较发现2号寄存器中存数为1大于0，转至3c。符合实验

预期。



25. addi　$2,$2,-1

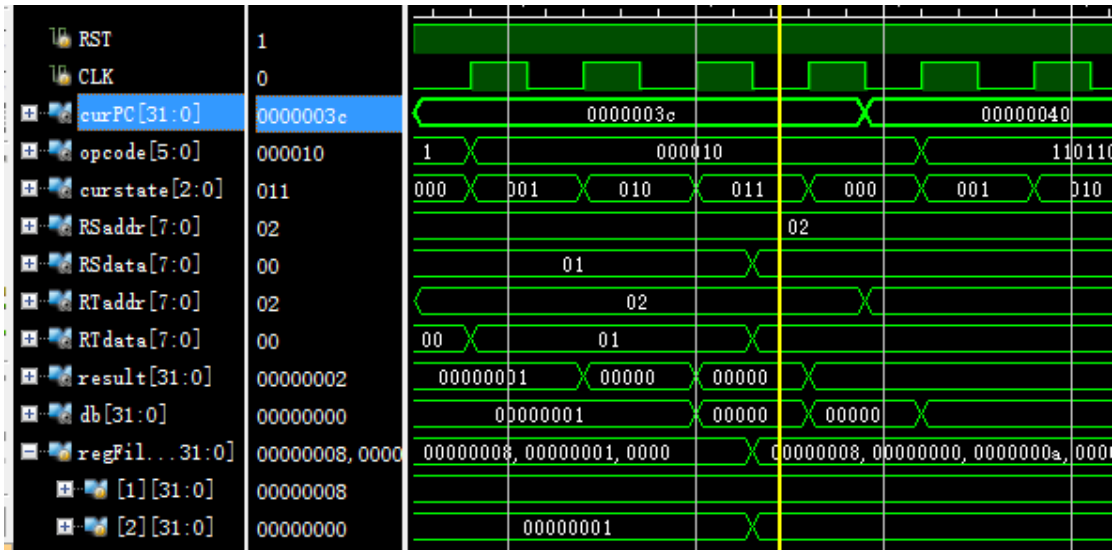
由图可知此指令要经历4个阶段。

IF：当前PC:3c 下条PC:40

ID：读出opcode为000010， rs编号：2 rt编号：2 ，执行加法操作。

EXE：ALU执行加法操作。

WB：将计算结果存入$2（可以看到db输出的数为$2=1-1=0），符合实验预期。



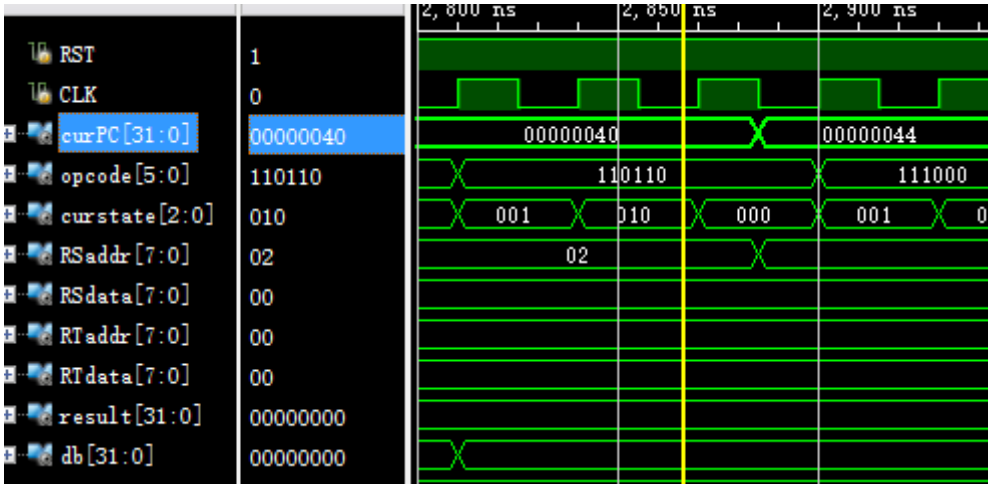
26. bgtz $2,-2 (>0,转3C)

由图可知此指令要经历3个阶段。

IF：当前PC:40 下条PC:44

ID：读出opcode为110110， 执行大于0则转移操作。

EXE：ALU执行减法操作，比较发现2号寄存器中存数为0等于0，不大于0，继续执行pc+4。符合实验预期。

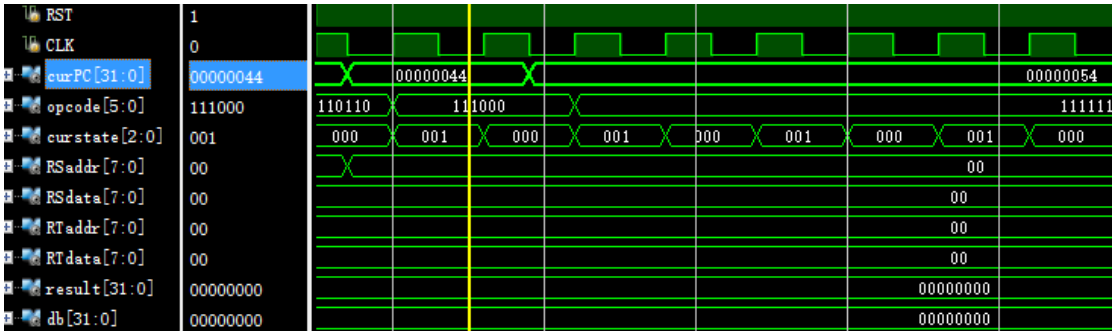


27. j　0x0000054

由图可知此指令要经历2个阶段。

IF：当前PC:44　下条PC:54

ID：读出opcode为111000，执行j操作。

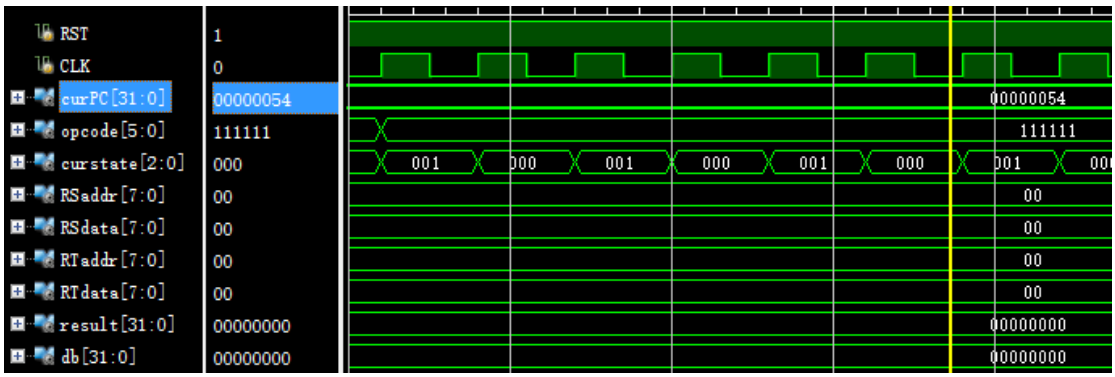


28. halt

由图可知此指令要经历2个阶段。

IF：当前PC:54，由于停机，PC停止跳转。

ID：读出opcode为111111，执行停机操作。

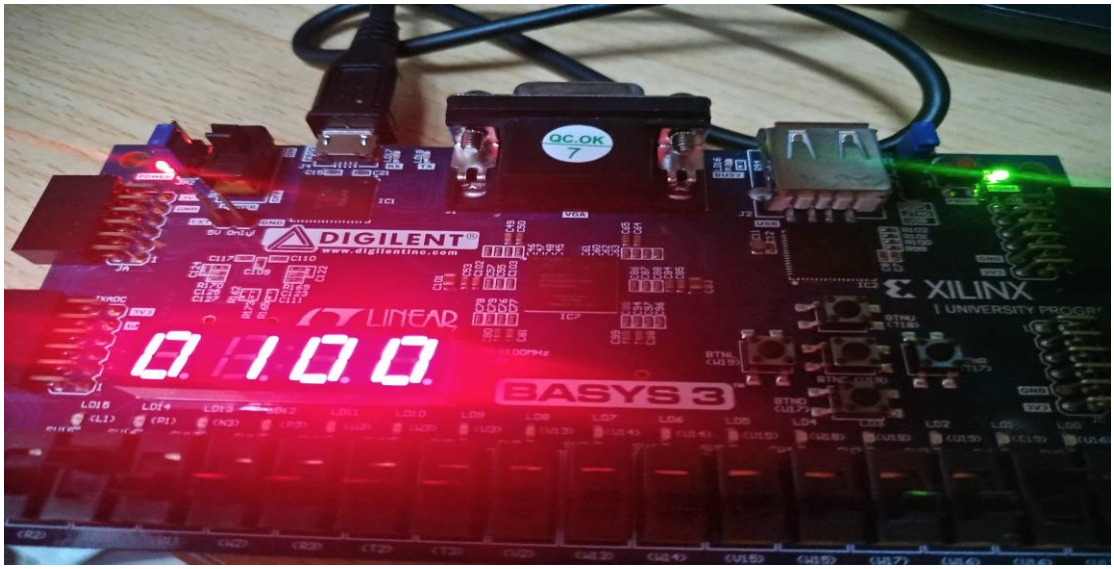


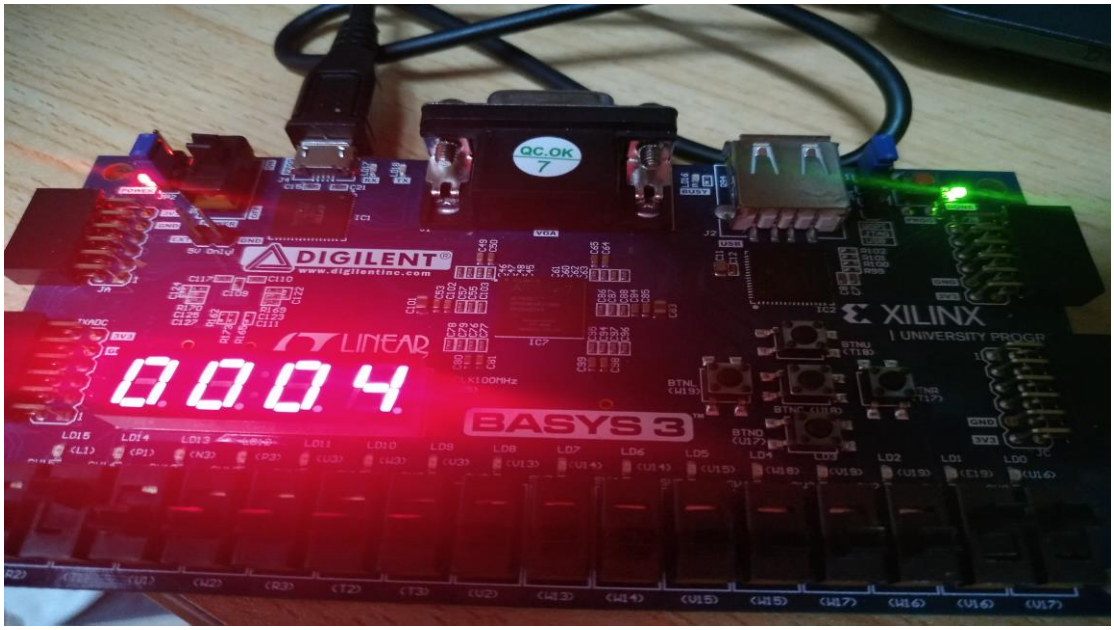
2、烧板结果

注：四张图依次为：i.当前PC 下一条PC ii.rs地址 rs数据 iii.rt地址 rt数据。iv.ALU数据，DB数据。

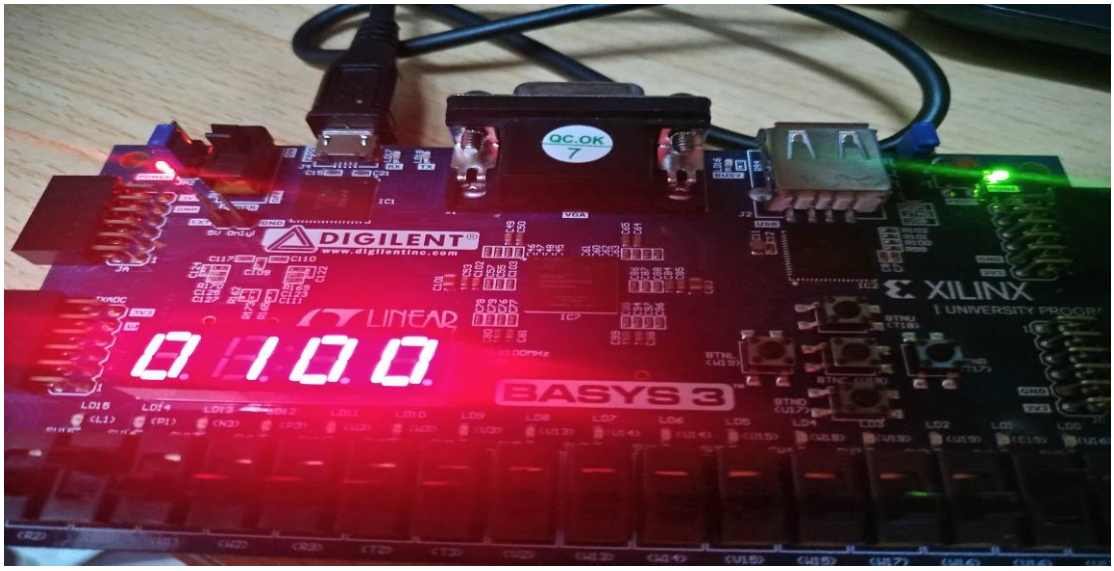
1. addi $1,$0,8

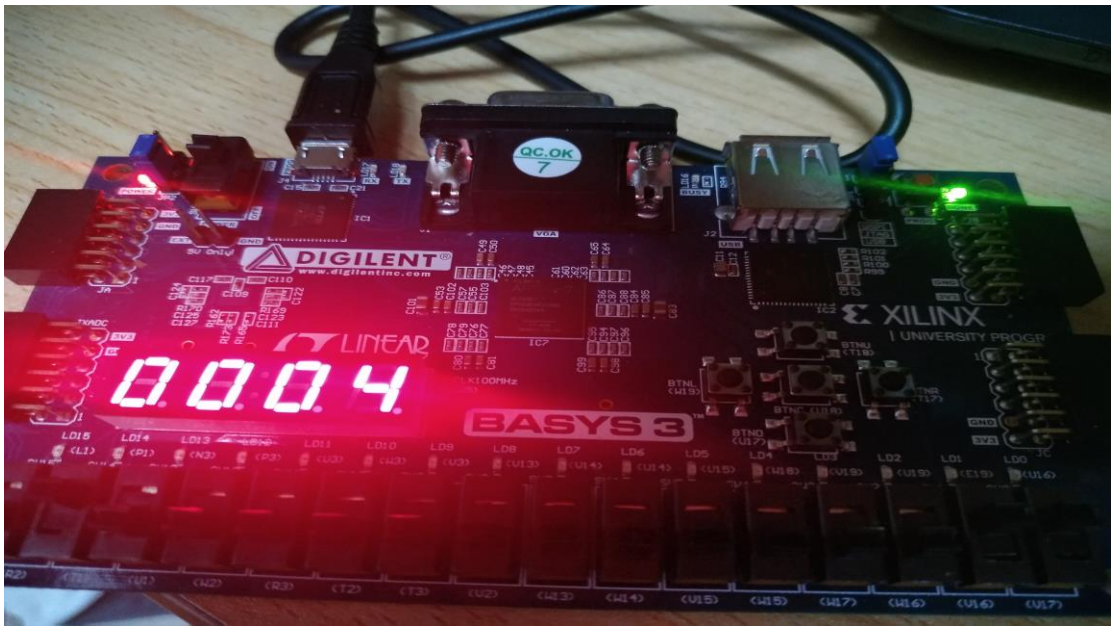
①IF周期如下面四张图片显示。此时PC显示正确，寄存器地址正确，别的部分还在未准备状态。





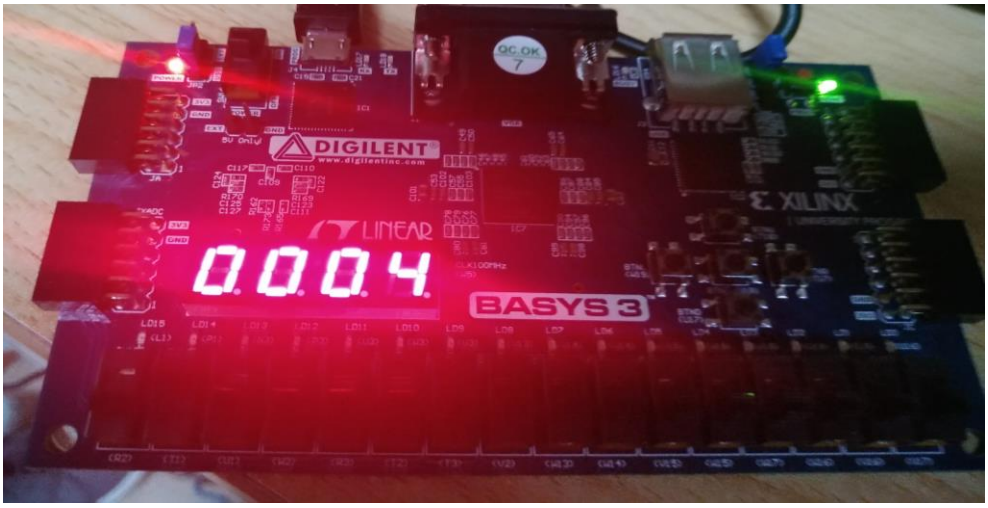
②ID周期如下面四张图片显示。此时PC显示正确，寄存器地址正确，别的部分还在未准备状态，译码部分并未能显示在板上，故与上面几张没有什么不同。

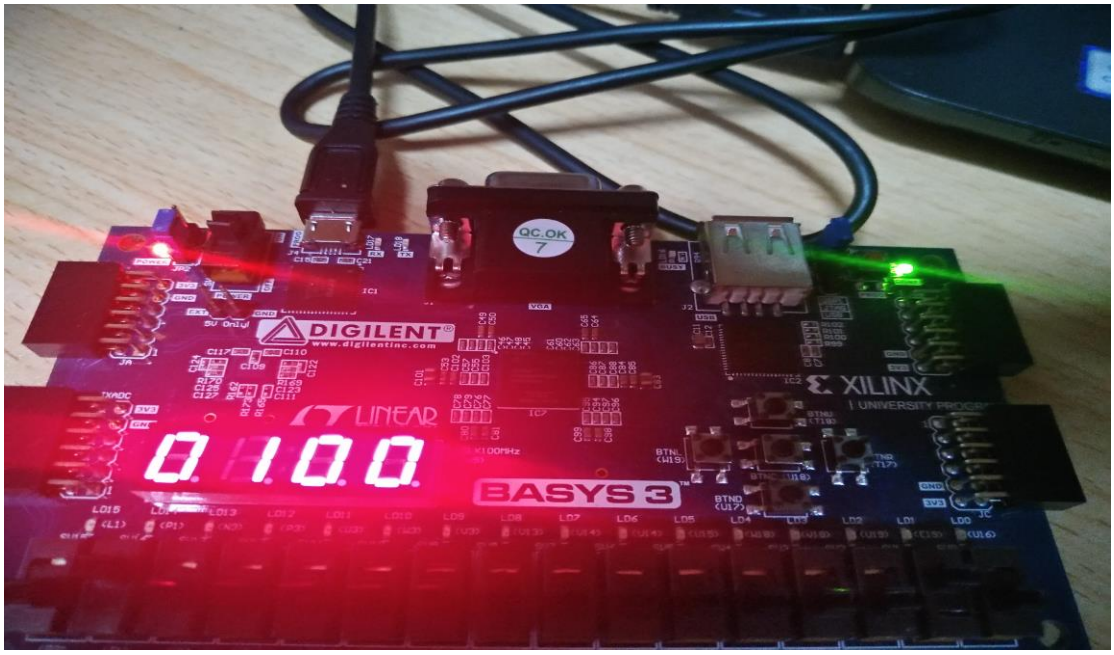


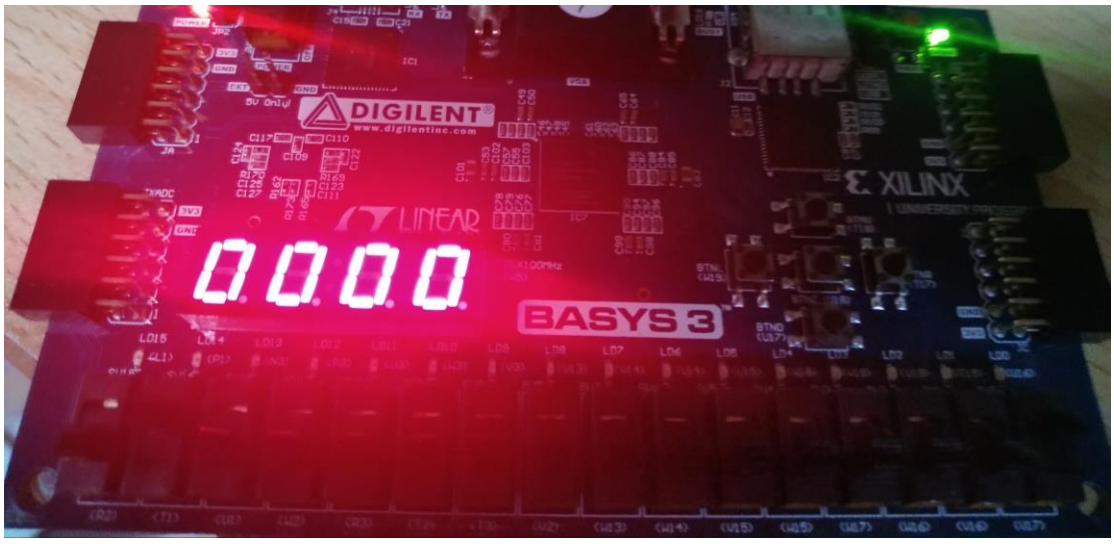


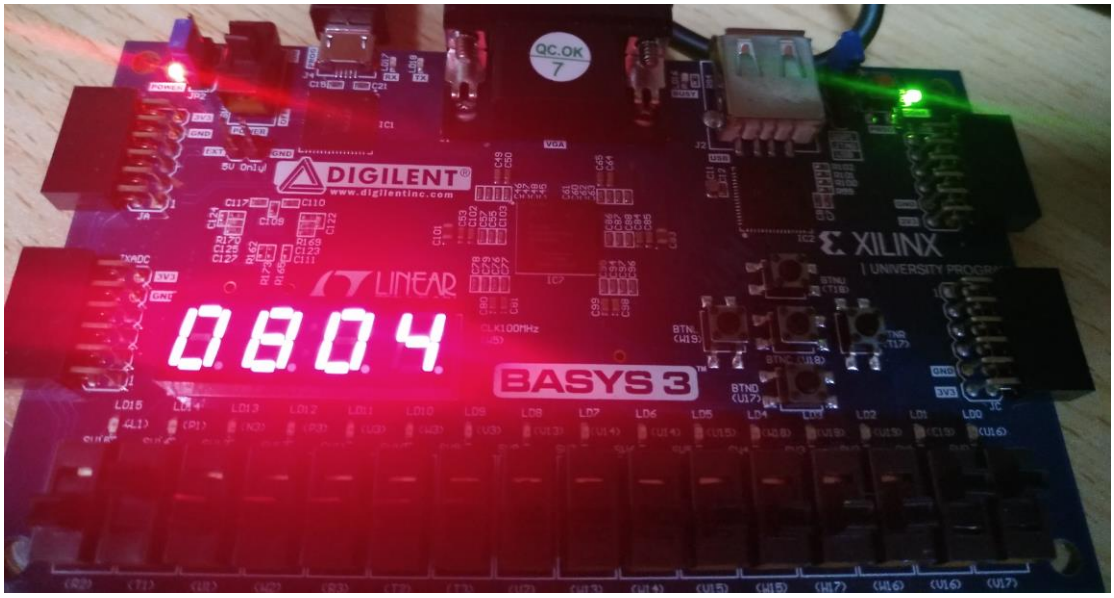
③EXE周期如下面四张图片显示。此时PC显示正确，寄存器地址正确。值得注意的是

此时的ALU已经运算完毕出结果了。

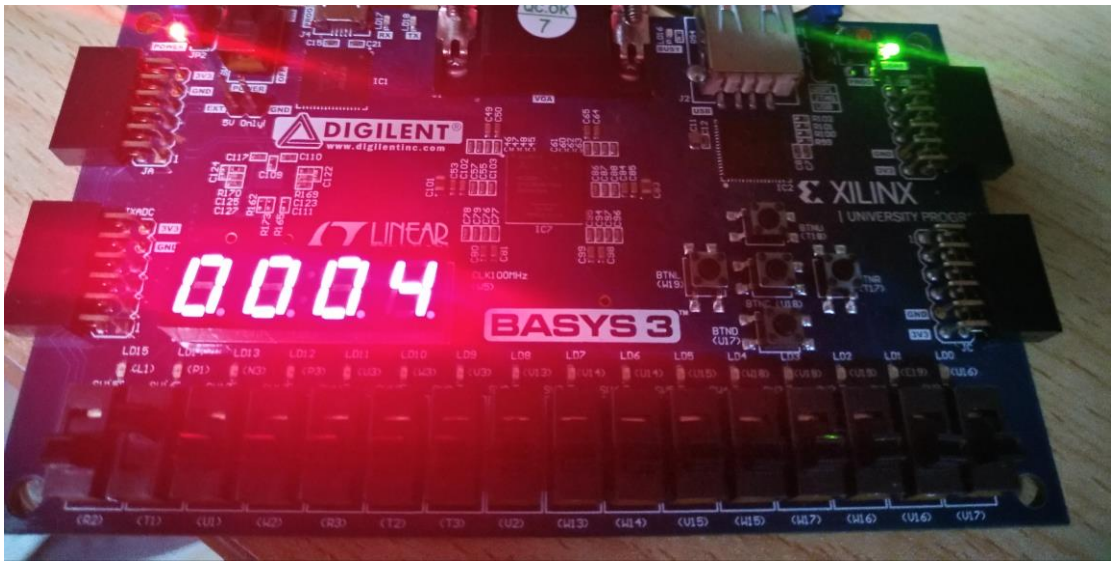


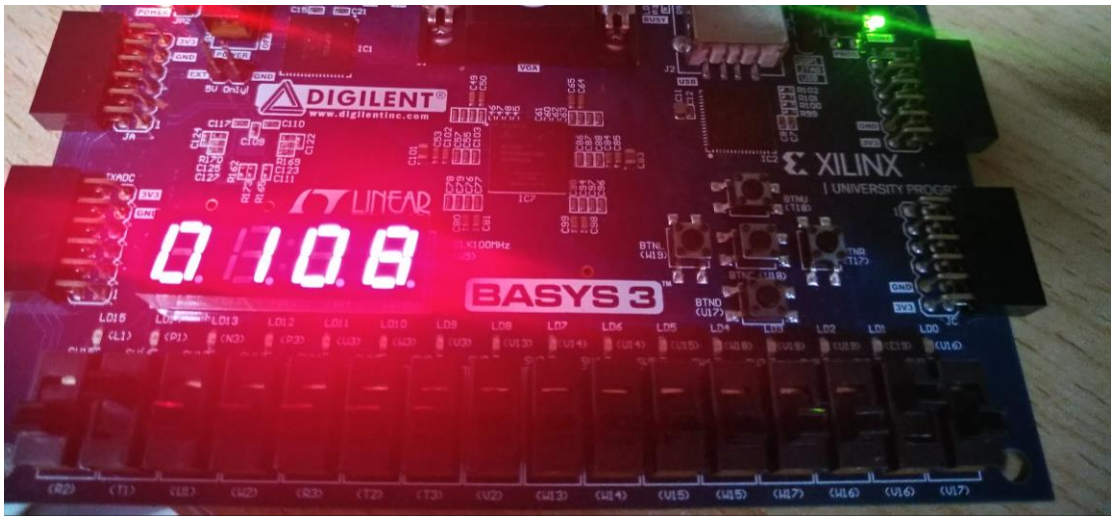


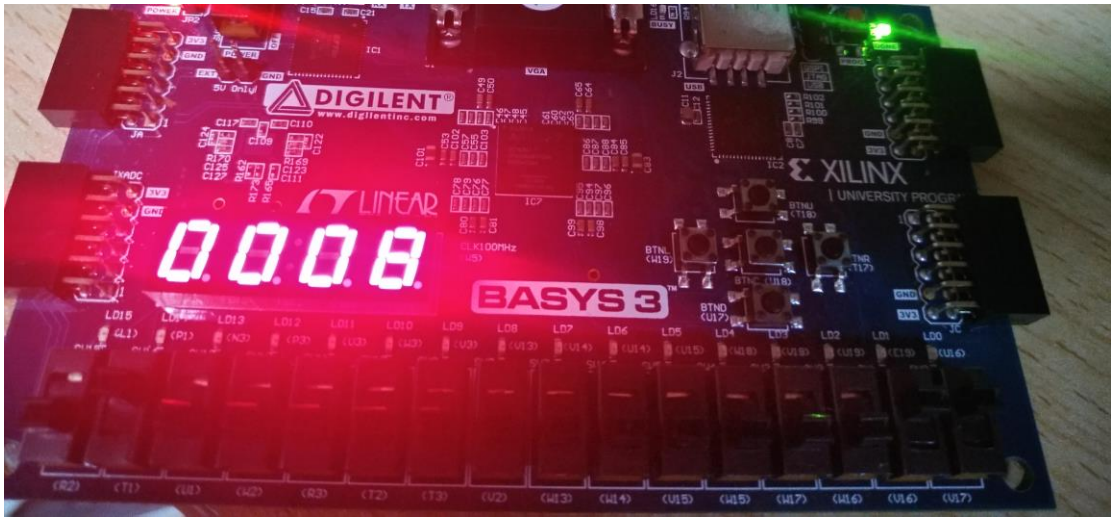




④WB阶段如下所示，可以看到此时ALU已经将数据传送给DB，DB已经输送出数据并且1号寄存器的值也已经为8，整条指令完成。

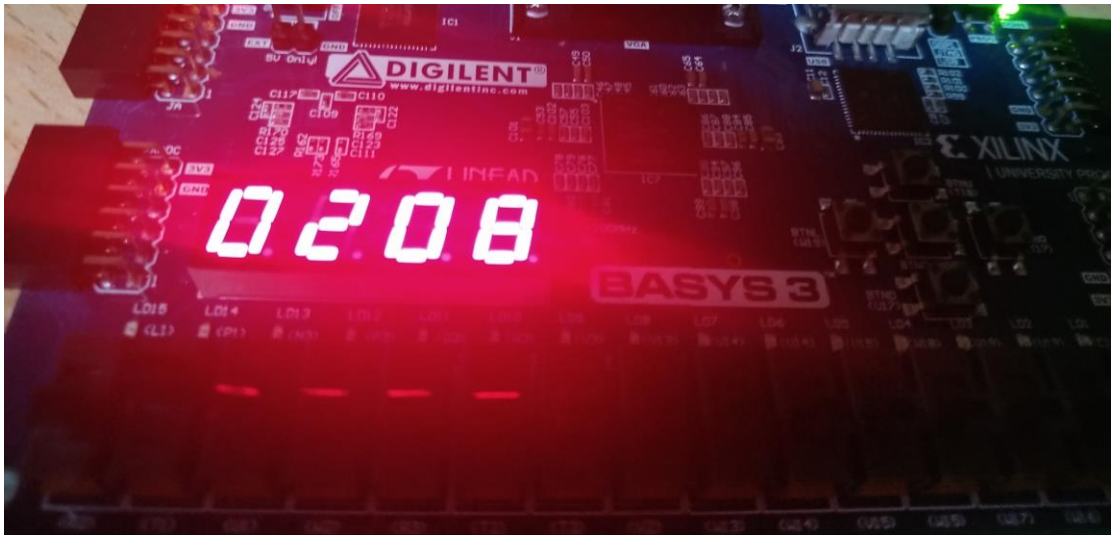






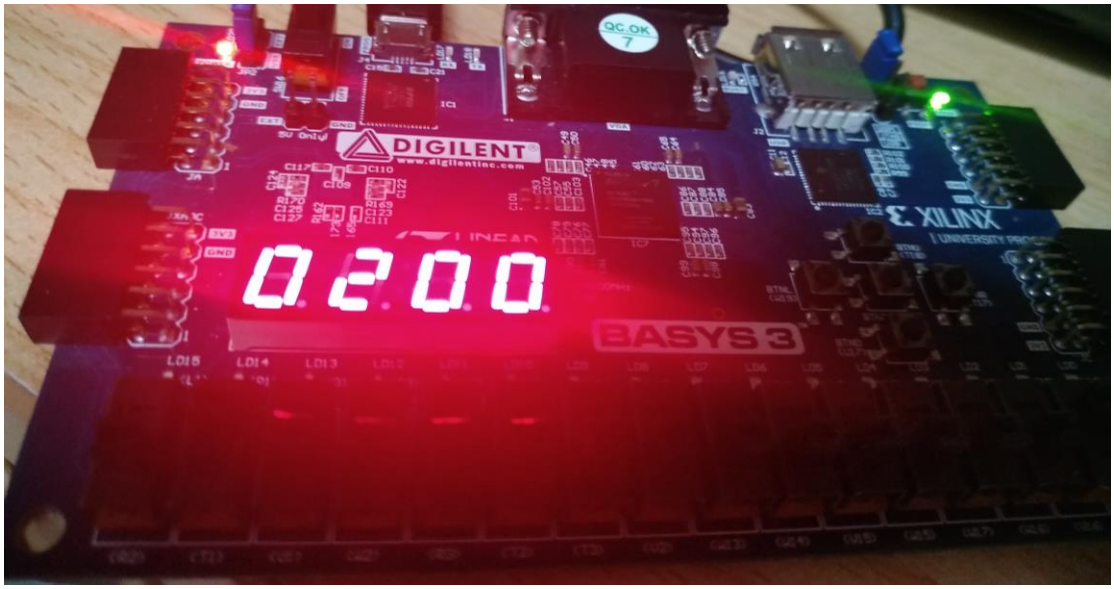
2、ori　$2,$0,2

①在IF阶段我们可以看到PC显示正确，但由于下一个阶段还没来临，寄存器和数据显示的仍为上一条的数据。

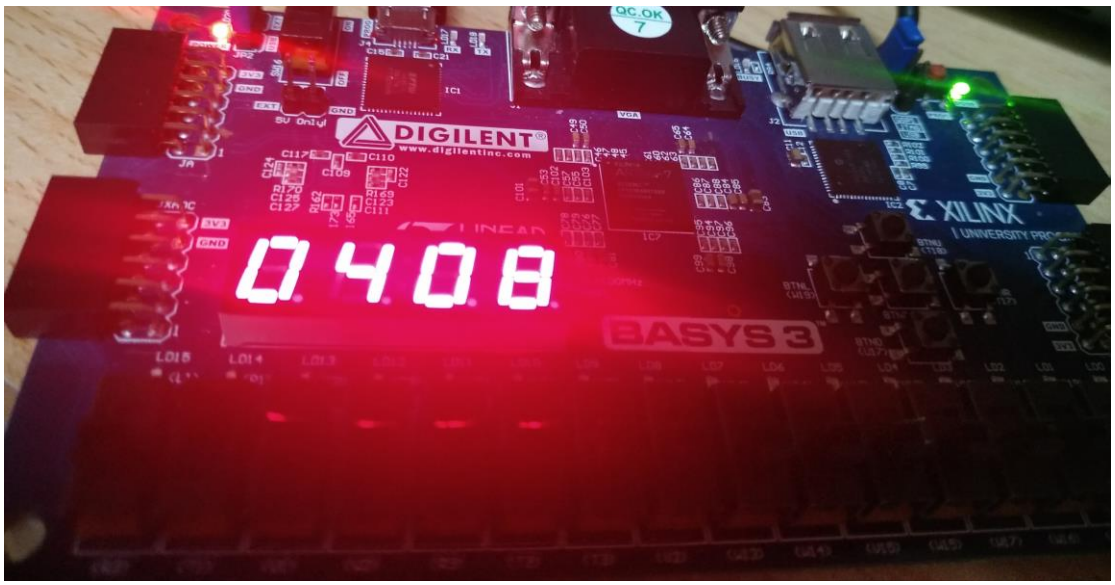


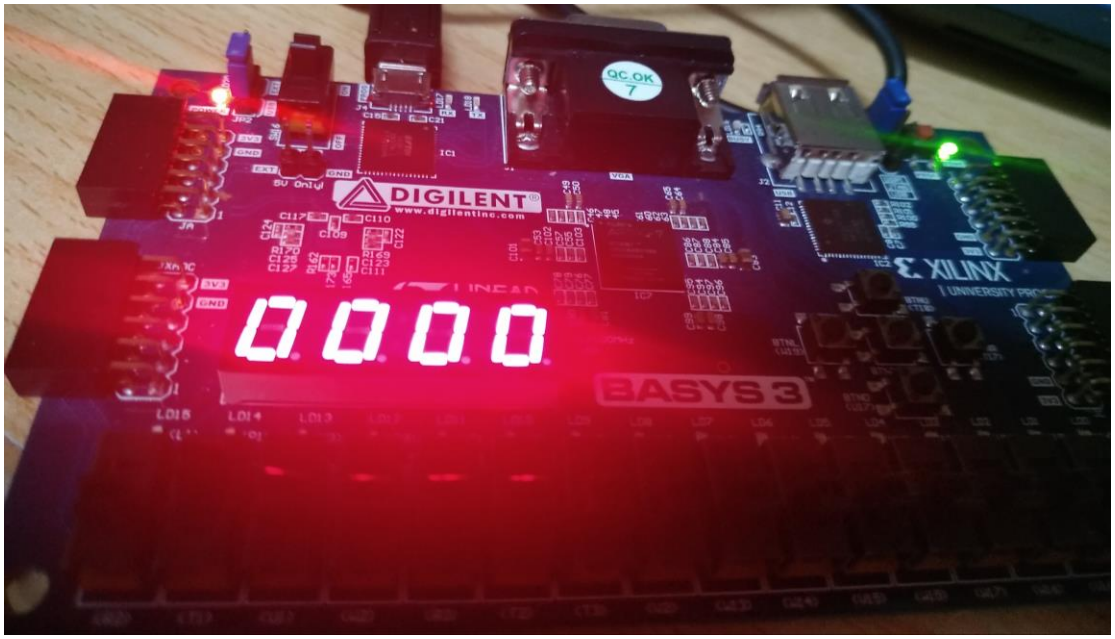
②ID阶段与上一条指令一样，在【显示在板上的方面】没有什么不同，故不截图了。

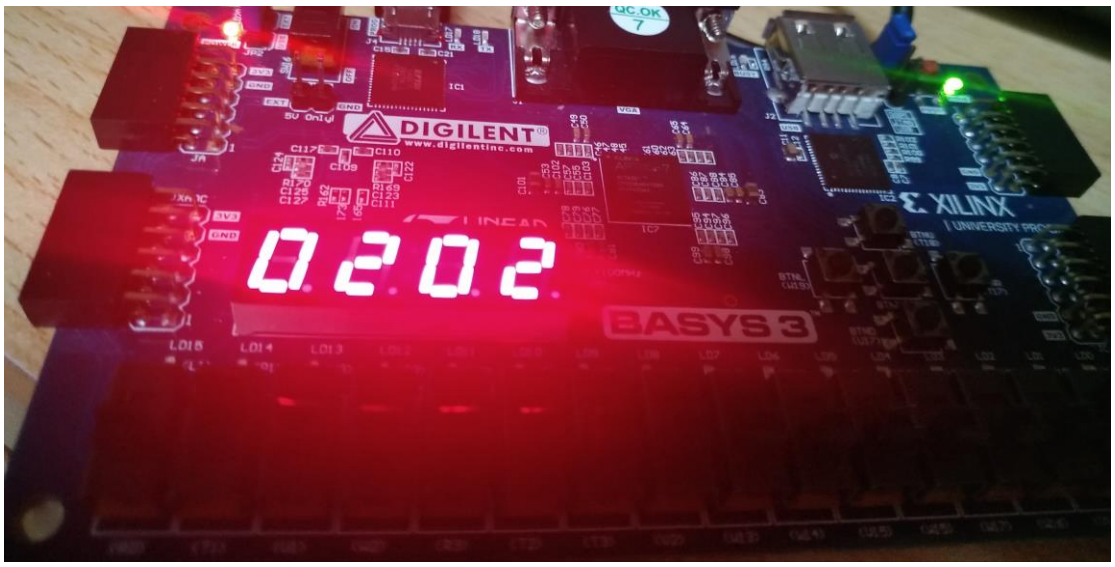
③EXE阶段，我们可以看到ALU已经计算好了结果“2”并准备传送。

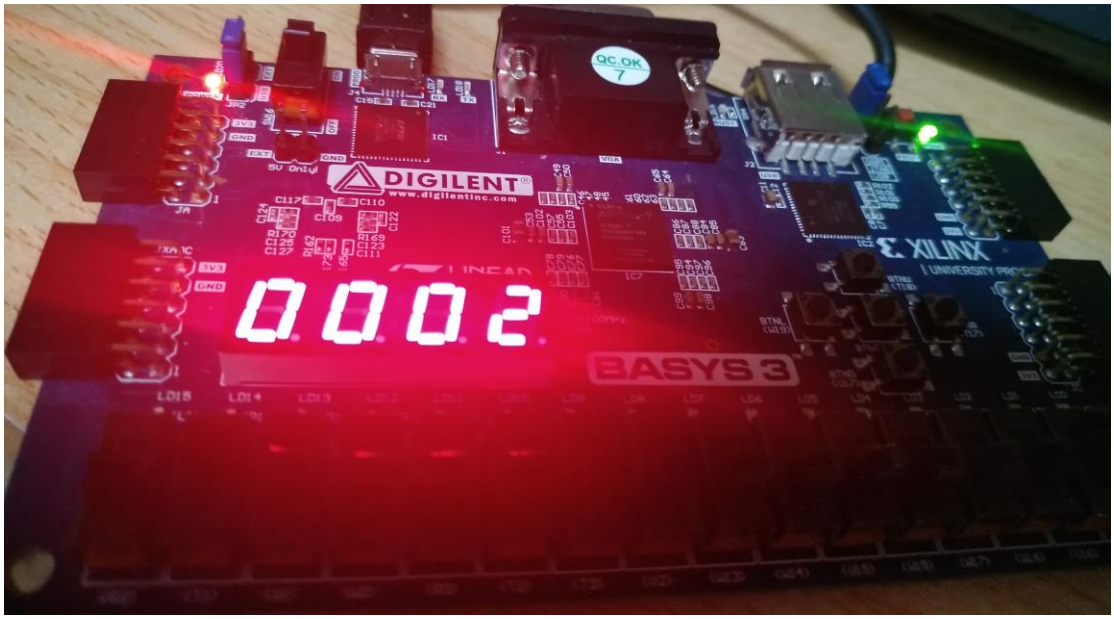


④EXE阶段，我们可以发现ALU的数据已经送到了DB输出，同时完成了写寄存器的操作，2号寄存器已经显示了数据“2”，传输成功。


0408
DIGILENT
BASYS 3
XILINX


0000
DIGILENT
BASYS 3
XILINX


0202
DIGILENT
BASYS 3
XILINX



……

因每条代码都需16张照片，为了避免实验报告冗长，我在此处只放两条指令不同状态时的图片，别的指令如果需要，我会在检查实验时展示。

六、实验心得

1、本次实验心得：

我觉得多周期实验相对于单周期实验来说更难了一些，但好在我们可以在单周期CPU的基础上修改，而不是像单周期CPU那样每一部分都需要重新写。多周期CPU主要是控制单元需要重新写，另外加上几个DR，别的方面改动不大。控制单元的编写只要理解了原理就变得简单，所以只要对着老师给的资料中的各个信号表以及那张结构图，很快就能写好控制单元。

多周期CPU的编写中，我发现与单周期CPU不同之处是，当PC是上升沿触发时，和控制单元里的许多控制信号存在竞争与冒险，在仿真时也许不会出问题，但到烧板时就受硬件影响可能出现问题。所以在多周期CPU中，我将PC设置成下降沿触发，这样避开了冒险，同时PC的上升沿和下降沿触发对烧板后时钟控制没有什么影响（毕竟时钟单步触发，也就是按一下键的短促时间而已）。

本次实验中我发现了一个与众不同的指令——jal。jal按流程来说是有一个写寄存器的

流程的，但老师给的表中jal只有IF、ID两个阶段。正当我纳闷这该怎么写的时候，我发现老师给的原理图中在RegDst的MUX选择模块中直接增加了一条$31写入的分支，这样我可以直接在控制单元里控制RegDst信号来使$31被写入数据，节省了不少麻烦。这个发现很有趣，嘻嘻。

仿真时，ALU和DB经常冒出来奇奇怪怪的数字。后来我才发现，这两个元件在自己应当运转的部分之外仍在通电，仍在工作，读出的可能是上一条指令的数据，也可能是指令改变导致ALUOP改变，ALU仍用上一条指令的数据在本条指令的其它状态下进行了新的运算操作，只不过是没有把结果写进寄存器或者内存而已。一开始我以为是我的代码写错了，后来发现在EXE状态下ALU的输出和在WB状态下DB的输出都符合实验预期，才发现并不是我写错了，是ALU和DB的正常现象。只是烧板时和单周期CPU不同。单周期CPU一个时钟信号即可看到所有的结果，但是多周期CPU烧板时，需要我按键到相应的状态才看得见结果，比不上单周期烧板看得方便。

综合时，我发现综合老是显示失败，但是我的代码根本就没有什么错误，一下子就慌了。我改了许多地方但是都显示失败。绝望之时，我试着把文件复制到C盘下，这次综合却对了。这下我发现了原因:原来我一开始把工程放在桌面时，桌面路径是来自于我的用户文件夹(中文名）的。综合失败是因为路径中含有中文啊！其实之前数电实验用protues时也不允许文件的路径里有中文名。以后做工程时我会尽量不把它们放在有中文名的路径中，以免出现不必要的错误。

烧板过程很顺利，没有出现什么bug。此时我发现单周期时发现的那个问题，即一开始没有一个下降沿使得1号寄存器写入8的bug在多周期CPU实验中就不存在了——因为一开始默认的只是第一条指令的IF阶段，到WB阶段时我早就用按键控制了，可以轻松地给它写入。原来多周期CPU本身构造就可以解决单周期CPU里出现的很多问题呀!

2、本学期课程心得

这学期的课程学习完，我最大的体会就是计算机组成原理这门课程真的不简单，无论是理论课还是实验课要学的东西太多，智商一时跟不上的我在上面花费了太多的时间。这门实验课也让我花了大量的时间在编写两个CPU上。上学期学习数电时，我看有些学霸级别的同学的大作业是单周期的CPU，就羡慕得不行，以为那是天方夜谭的东西。没想到这学期我在磕磕绊绊中也都按时做完了两个工程，心里还是有些成就感的。这个学期为了两个工程

付出了很多心血，因为一点点致命错误焦头烂额仔细查错到处询问，因为写实验报告凌晨五点才睡……但是这些时间我觉得都没有白费，它起码让我完成了我的两个大作业，我想上学期的我应该没有想到我现在也能做出来CPU了。

但是，这些成果是在老师已经给出了各种指令机器码、大部分元件的verilog代码和原理图的情况下完成的。我明白如果没有老师给的这些资料，我大概花很久很久也不能做出来，所以我明白自己的能力还有待提高。三个实验中，第一个实验我花费了两个小时，第二个实验花费了两个星期，第三个实验在第二个实验的基础上仍花费了一个星期。三个实验难度可以说是层层递进。

这一学期以来，我在这门课上花费的精力是很多的。说不累是假的，但感谢老师给了我们很多干货，让我觉得我真正学到了东西，也感谢老师给了我们做实验的资料和机会。虽然为了实验熬夜的时候也曾抱怨过疲累，但做完后我感觉老师给的机会让我实践了所学的知识的可应用性。同时我也感谢我的朋友们在我迷惑的时候解答我的问题，给予我他们能给的帮助。其实我发现大家一起讨论时可以解决许多问题，节省了许多的时间，这也说明了意见交流的重要性。总之，在老师的实验课期间，不管是课上还是课下，我觉得我收获颇丰。