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）：	17 计教学 3 班
学 生 姓 名 ：	郑康泽
学 号 ：	17341213
时 间 ：	2018 年 12 月 17 日

成 绩 ：

实验三：多周期 CPU 设计与实现

（完成时间：十五、十六、十七周）

一、 实验目的

(1) 认识和掌握多周期数据通路图的构成、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：

==>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）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（5）andi rt , rs ,**immediate**

010001	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

（6）ori rt , rs ,**immediate**

010010	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。

（7）xori rt , rs , **immediate**

010011	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

==>移位指令

（8）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

（9）slti　rt, rs,immediate　　带符号

100110	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。

（10）slt　rd, rs, rt　　带符号

100111	rs(5 位)	rt(5 位)	rd(5 位)	reserved

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

==>存储器读写指令

（11）sw　rt, **immediate(**rs)

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。

（12）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

(13）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。

(14）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。

（15）bltz　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

（16）j　addr

111000	addr[27:2]

功能：pc < －{(pc+4)[31:28],addr[27:2],2'b00}，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。

（17）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc < －　rs，跳转。

==>调用子程序指令

（18）jal　addr

111010	addr[27:2]

功能：调用子程序，pc < －　{(pc+4)[31:28],addr[27:2],2'b00}；$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

（19）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

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。

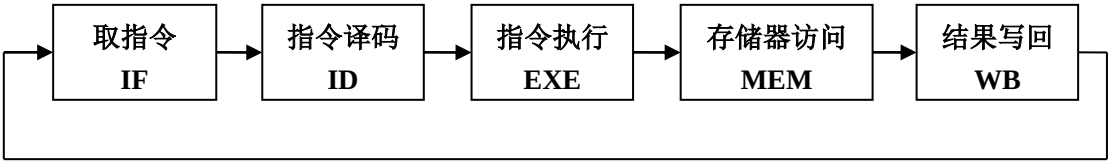


图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

31 26	25 21	20 16	15 11	10 6	5 0
op	rs	rt	rd	sa	funct
6 位	5 位	5 位	5 位	5 位	6 位

I 类型：

31 26	25 21	20 16	15 0
op	rs	rt	immediate
6 位	5 位	5 位	16 位

J 类型：

31 26	25 0
op	address
6 位	26 位

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。

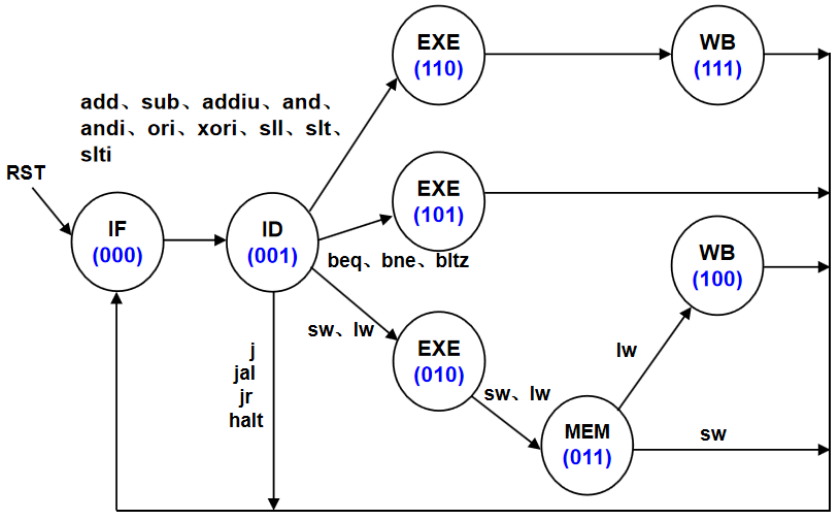


图 2 多周期 CPU 状态转移图

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。

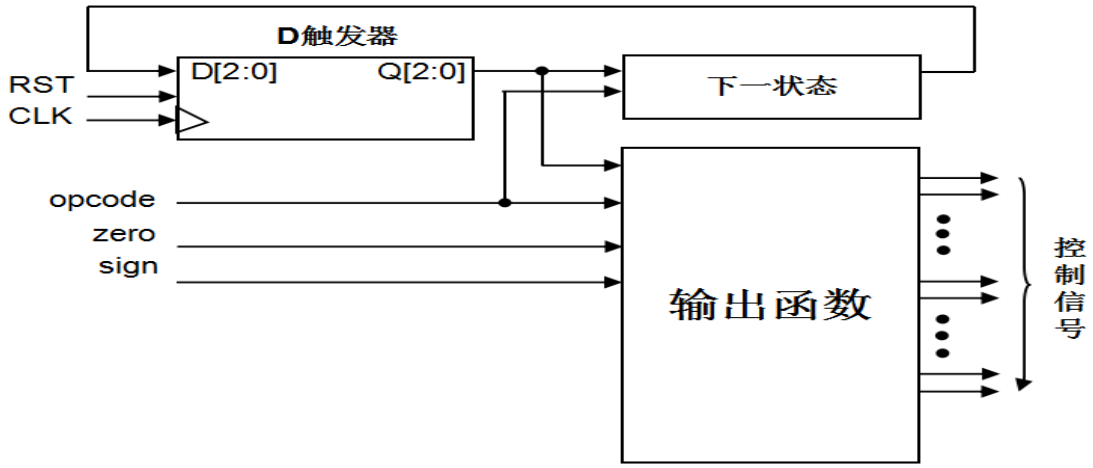


图 3 多周期 CPU 控制部件的原理结构图

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。

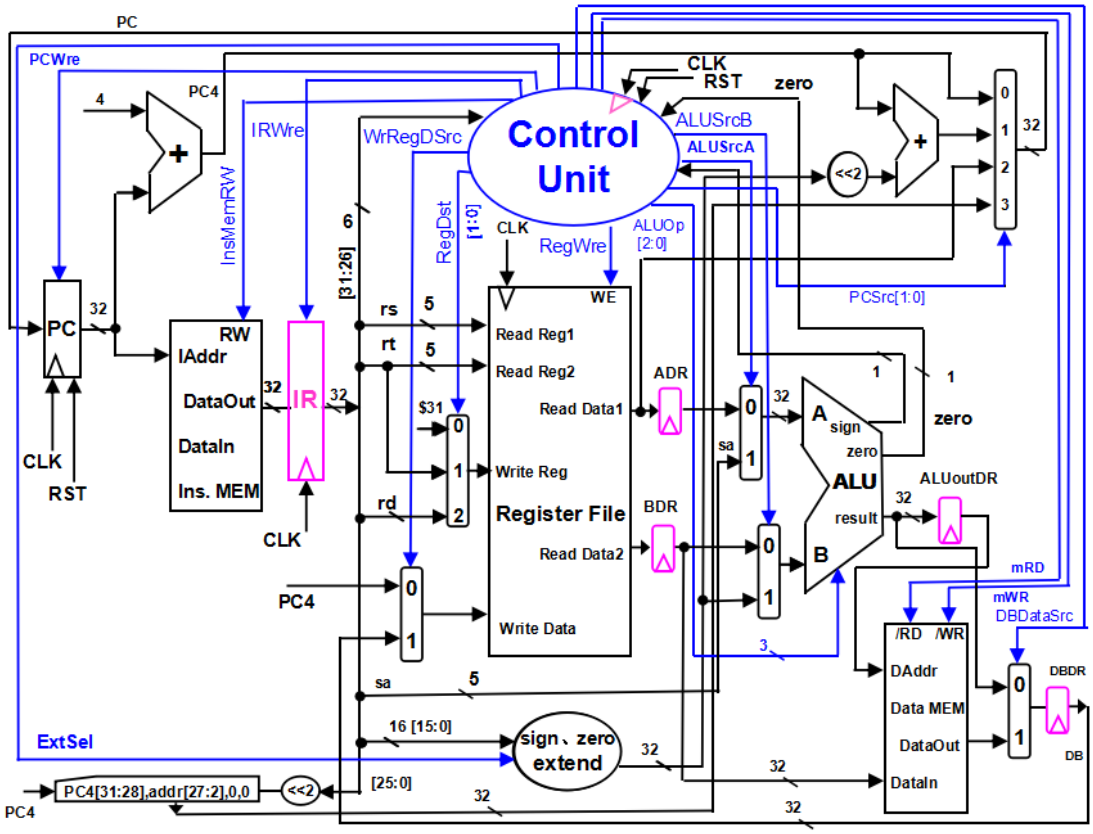


图 4 多周期 CPU 数据通路和控制线路图

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、addiu、and、andi、ori、xori、slt、slti、sw、lw、beq、bne、bltz	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、and、slt、sll、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addiu、andi、ori、xori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti	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、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti、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、addiu、sub、and、andi、ori、xori、sll、slt、slti、lw
mRD	存储器输出高阻态	**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**
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：andi、xori、ori;	(sign-extend)**immediate**，相关指令：addiu、slti、lw、sw、beq、bne、bltz;
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、and、andi、ori、xori、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0); 01：pc<－pc+4+(sign-extend)**immediate ×4**，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1); 10：pc<－rs，相关指令：jr; 11：pc<－{pc[31:28],addr[27:2],2'b00}，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：addiu、andi、ori、xori、slti、lw； 10：rd 字段，相关指令：add、sub、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**

laddr，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 = 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<B 不带符号
110	Y=(((A<B)&&(A[31] == B[31])) ||((A[31]==1&& B[31]	比较 A<B 带符号

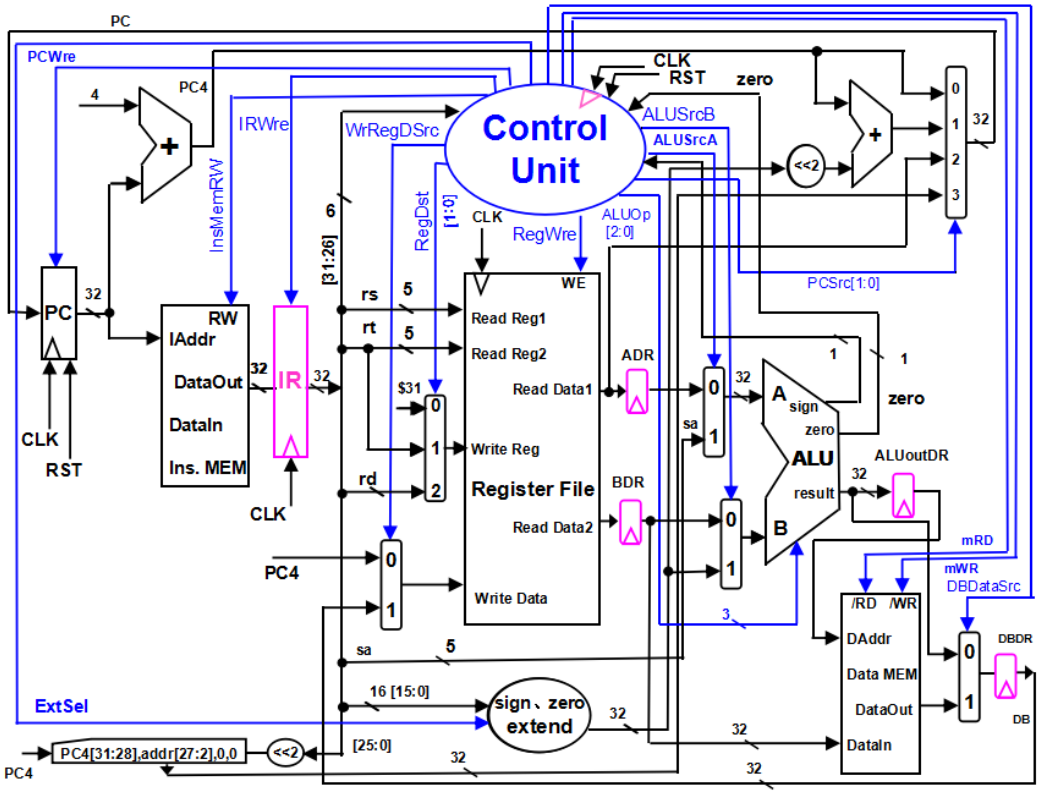
	== 0))) ? 1:0	
111	Y = A ⊕ B	异或

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.实验思想：



（1）核心思想：

多周期CPU设计核心思想在于一条指令需要多个时钟周期来实现，指令的不同也会使得需要时钟周期的个数不同。一个时钟周期对应相应的阶段，基本上有五个阶段：IF、ID、EXE、MEM、WB，相应的阶段对应相应的动作。不同指令需要不同阶段来实现。比如addi、add、subi、sub、sll、xori、andi、and、or、ori、slt、slti等算数类型的指令，就需要IF、ID、EXE、WB阶段；bne、beq、bltz等有条件跳转指令，就需要IF、ID、EXE阶段；sw指令需要IF、ID、EXE、MEM阶段；lw指令需要IF、ID、EXE、MEM、WB阶段，j、jr、jal等无条件跳转指令还有halt指令，就只需要IF、ID阶段。

（2）关于各个模块的上升沿下降沿问题：

一开始啥都不懂的情况下，我就选择了状态变换的模块以下降沿触发，即状态的变换是在每个时钟的下降沿改变的。接下来，我决定了PC的改变是在上升沿触发，然后我觉得IR寄存器应该是在第二个阶段才能有输出，所以我也把它设置为上升沿触发，同理，ADR、BDR、ALUoutDR也是上升沿触发，但DBDR不能用上升沿触发，因为在下降沿到来时，CPU已经进入WB阶段，mRD无效了，所以在上升沿的时候，已经不能读储存器，所以我应该在进入WB前先把数据读出来，所以我设置DBDR下降沿触发；而寄存器写依旧是下降沿写，没问题，但是存储器不能不用时钟控制，因为要写的地址在上升沿时才到，而mWR有效在要写的地址出来之前，所以就可能乱修改了存储器的值，所以我用下降沿控制储存器写。这些上升沿下降沿都是Debug后决定的。

（3）不同阶段使不同信号有效：

比如在IF阶段，应使得PCWre有效，当然halt指令除外；在ID阶段，应使得IRWre有效，这样才能使得取出指令能出来，还有要置好ALUSrcA、ALUSrcB、DBDataSrc、WrRegDSrc等信号，当然遇到jal指令，应该使RegWre有效，让下一条指令的地址写进$31寄存器，并且RegDst=0，写进$31寄存器，这个过程相当于译码的过程；在EXE阶段，根据获得的ALUOp，进行相应的操作；在MEM阶段（如果有的话），根据mWR、mRD信号，进行相应操作；在WB阶段，根据要写的目标寄存器以及要写的数据，进行写操作。

（4）相关模块及代码：

```
module PC(
    input CLK,                      // 时钟
    input RST,                      // 重置信号
    input PCWre,                    // PC能改信号
    input [31:0] PC_In,             // 下一条PC
    output reg [31:0] PC_Out        // 当前PC
    );

module Ins_Mem(
    input [31:0] PC,                    // 当前指令地址
    output [31:0] Instruction           // 当前指令
    );
```

```
module IR_Register(
    input CLK,                  // 时钟
    input [31:0] Ins_in,        // 取出的指令
    input IRWre,                // IR能输出信号
    output reg [31:0] Ins_out   // 输出当前指令
    );

module Register_File(
    input CLK,                  // 时钟
    input RegWre,               // 寄存器能写信号
    input [4:0] Read_Reg1,      // 要读寄存器的地址
    input [4:0] Read_Reg2,      // 要读寄存器的地址
    input [4:0] Write_Reg,      // 要写寄存器的地址
    input [31:0] Write_Data,    // 要写的数据
    output [31:0] Read_Data1,   // 读寄存器的数据
    output [31:0] Read_Data2    // 读寄存器的数据
    );

module State_Transform(
    input CLK,                  // 时钟
    input RST,                  // 重置信号
    input [5:0] opcode,         // 指令前6位
    input [2:0] state_in,       // 上一个状态
    output reg [2:0] state_out  // 当前状态
    );

module Control_Unit(
    input zero,                 // ALU结果是否为0
    input sign,                 // ALU结果符号位
    input [2:0] state,          // 当前状态
```

```
    input [5:0]opcode,          // 指令前6位
    output reg PCWre,           // PC能写信号
    output reg IRWre,           // IR能写信号
    output reg ALUSrcA,         // ALU第一个操作数选择信号
    output reg ALUSrcB,         // ALU第二个操作数信号
    output reg DBDataSrc,       // 写回寄存器数据选择信号
    output reg RegWre,          // 寄存器能写信号
    output reg WrRegDSrc,       // 写寄存器选择信号
    output reg mRD,             // 存储器能读信号
    output reg mWR,             // 存储器能写信号
    output reg ExtSel,          // 有无符号扩展信号
    output reg [1:0] PCSrc,     // 下一条PC选择信号
    output reg [1:0] RegDst,    // 写寄存器选择信号
    output reg [2:0] ALUOp      // ALU进行什么操作信号
    );

module ALU(
    input [2:0] ALUOp,          // ALU进行什么操作信号
    input [31:0] Data1,         // 第一个操作数
    input [31:0] Data2,         // 第二个操作数
    output reg [31:0] Result,   // 计算后的结果
    output sign,                // 计算结果符号
    output zero                 // 计算结果是否为零
    );

module Data_Mem(
    input CLK,                  // 时钟
    input mRD,                  // 存储器能写信号
    input mWR,                  // 存储器能读信号
    input [31:0] Data_In,       // 要写的数据
```

```
    input    [31:0] Data_Addr,     // 要写的地址
    output    [31:0] Data_Out     // 读出的数据
    );

module DBDR(
    input CLK,                    // 时钟
    input DBDataSrc,              // 写回寄存器数据选择信号
    input [31:0] Data_Out,        // 从存储器读出的信号
    input [31:0] Result,          // ALU计算结果
    output reg [31:0] DBDR_out// 写回寄存器的数据
    )

module ADR(
    input CLK,                    // 时钟
    input [31:0] Read_Data1,      // 进来的数据
    output reg [31:0] ADR_out   // 上升沿到来时输出数据
    );

module BDR(
    input CLK,                    // 时钟
    input [31:0] Read_Data2,      // 进来的数据
    output reg [31:0] BDR_out   // 上升沿到来时输出数据
    );
```

2.过程展示

PC_Out：当前指令地址

PC_In：下一条指令地址

Ins_Out:：当前指令

state_out：当前状态

ADR_out：ADR的输出

BDR_out：BDR的输出

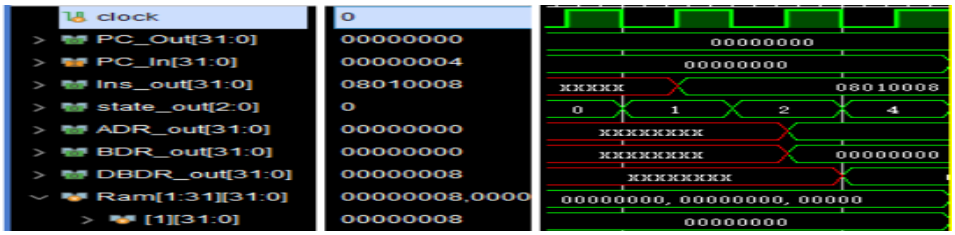
DBDR_out：DBDR的输出

ALUoutDR_out：ALUoutDR的输出

MEM：存储器单元

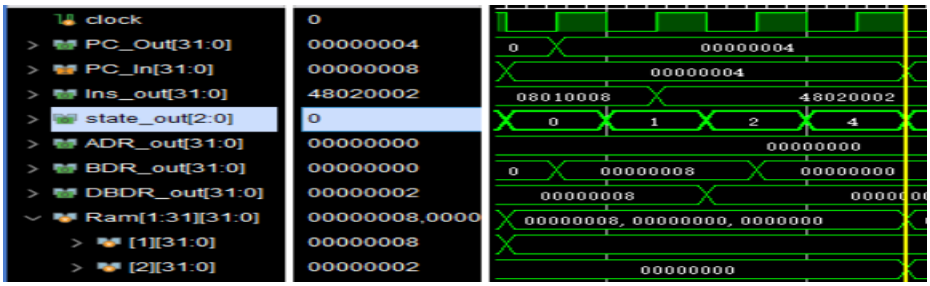
Ram：寄存器组

（1）addi　$1,$0,8



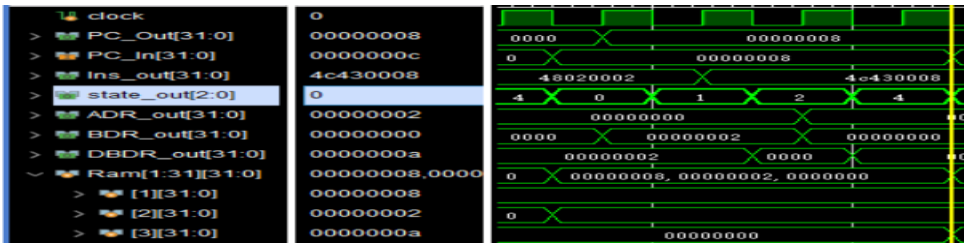
可以看出，状态是在每个下降沿改的，指令是在ID阶段的上升沿输出的，ADR、BDR是在EXE阶段的上升沿有输出的，0+8=8的结果是在WB阶段写进去的。

（2）ori　$2,$0,2



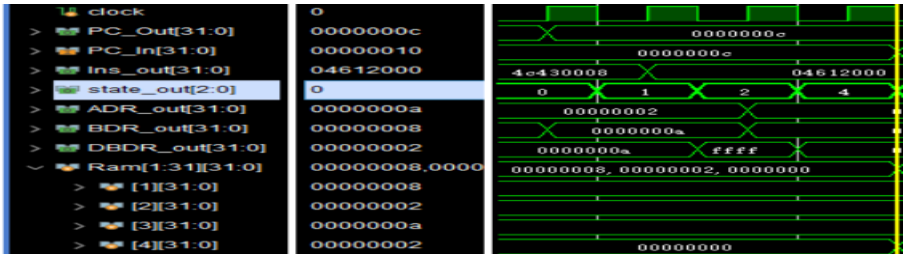
0|2=2，所以在WB阶段2号寄存器值成2。

（3）xori　$3,**$2**,8



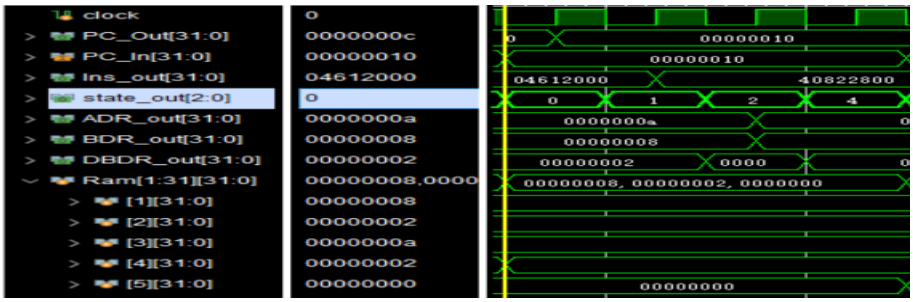
2+8=10,所以3号寄存器值变成10。

（4）sub　$4,**$3**,$1



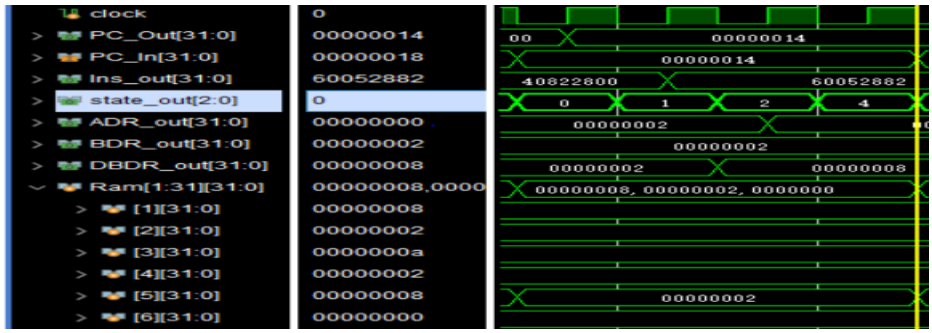
10-8=2，所以4号寄存器值变为2

（5）and　$5,$4,$2



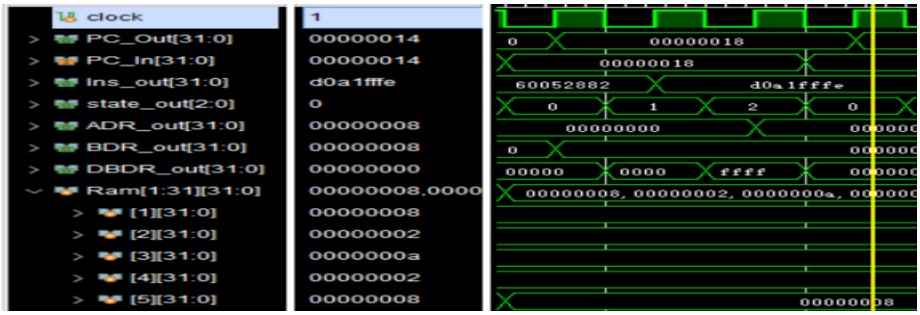
2&2=2，5号寄存器值变为2

（6）**sll　$5,$5,2**



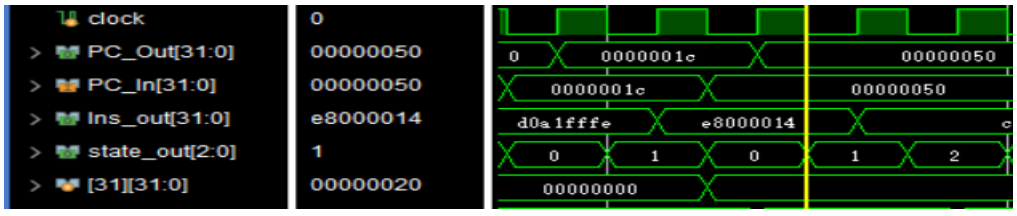
2<<2=8，所以5号寄存器值变为8。

（7）**beq　$5,$1,-2(=,转 14)**



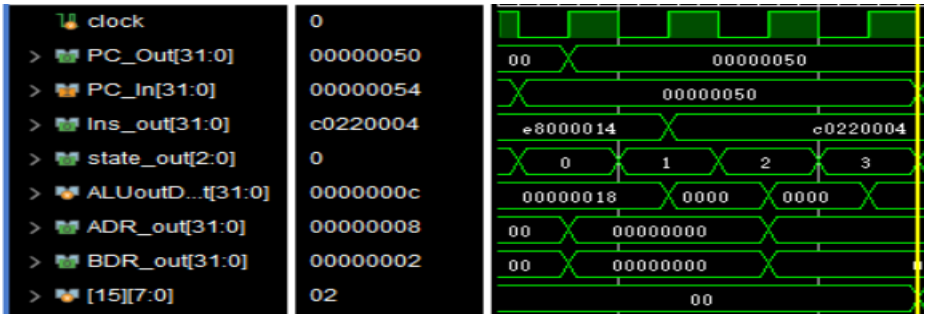
由于5号寄存器的值与1号寄存器的值相同，所以下一条指令的地址应该为14H，可以看出在这条指令执行完后，下一条指令地址变成14H。

（8）**jal　0x0000050**



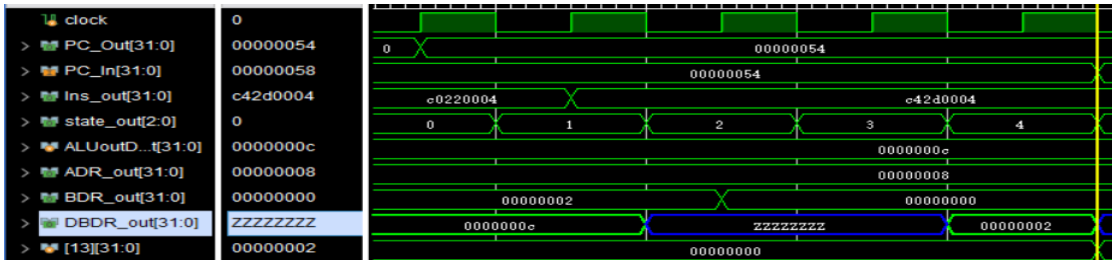
该指令执行完后，下一条指令地址PC_In变成50H，并且在下降沿31号寄存器的值变成了20H，也就是18H下一条指令的地址。

（9）sw　$2,4($1)



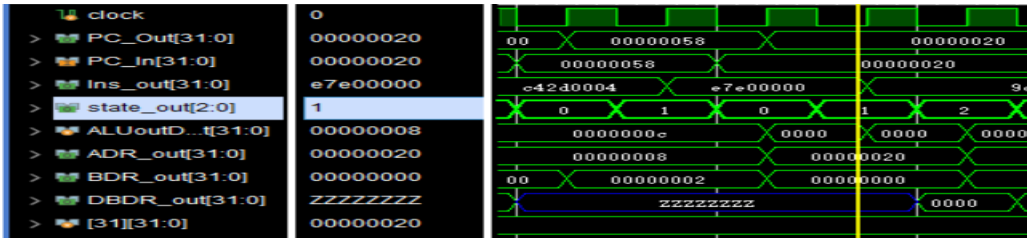
1号寄存器值为8，所以2号寄存器的值应写进地址从12到15存储器单元，可以看见ALUoutDR在Mem阶段的上升沿才有输出，在下降沿写进存储器。由于大端储存，所以15号单元的值是2.

（10）lw　$13,4($1)



lw指令是唯一一条需要所有阶段的指令，DBDR_out出现高阻态是因为进入ID状态后，DBDataSrc已被设置为选择存储器值作为写回值，但因为还未到Mem阶段，mRD还未有效，读出的值一直为高阻态。可以看到13号寄存器值改成2.

（11）**jr　$31**

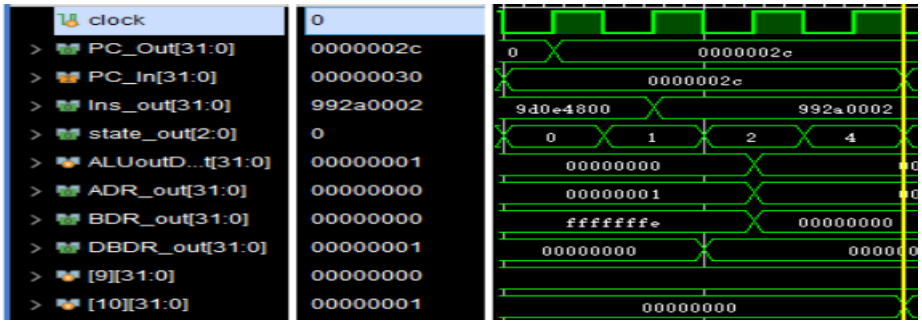


跳回地址为31号寄存器值，所以执行完这条指令，下一条PC的值为20H，因为31号寄存器的值为20H。

（12）**slt　$8,$13,$1**

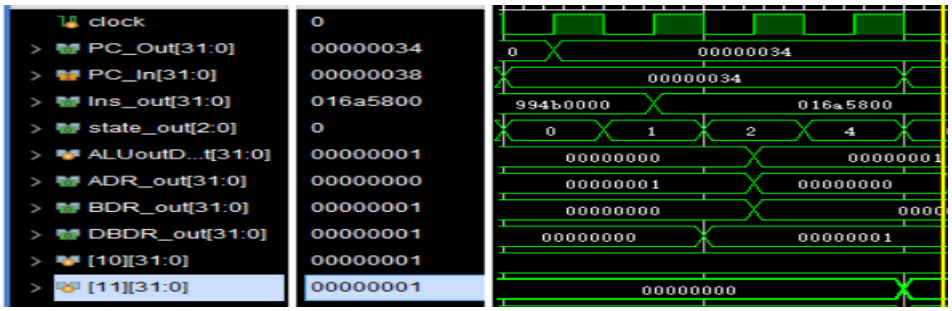
因为13号寄存器值为2，1号寄存器值为8，所以13号寄存器值小于1号寄存器值，所以8号寄存器值要置一。

（13）**slti　$10,$9,2**



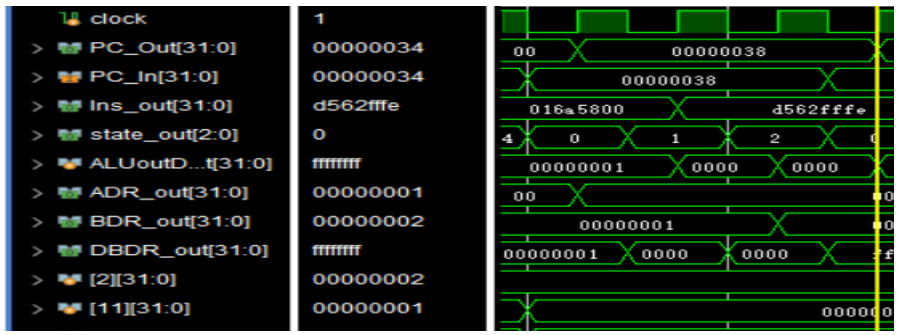
9号寄存器值为0，小于2，所以10号寄存器值要置一。

（14）add　$11,$11,$10



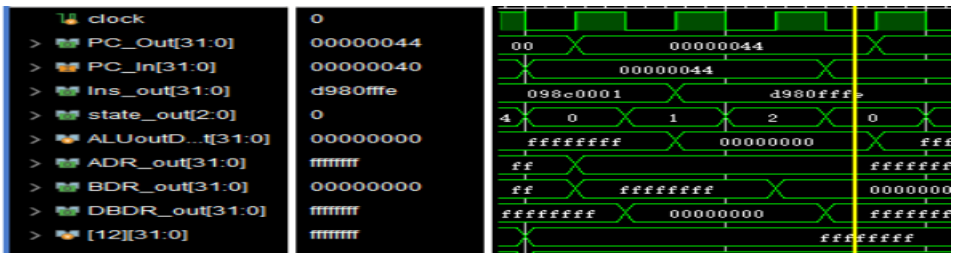
1+0=1，所以11号寄存器值变为1。

（15）**bne　$11,$2,-2 (≠,转34)**



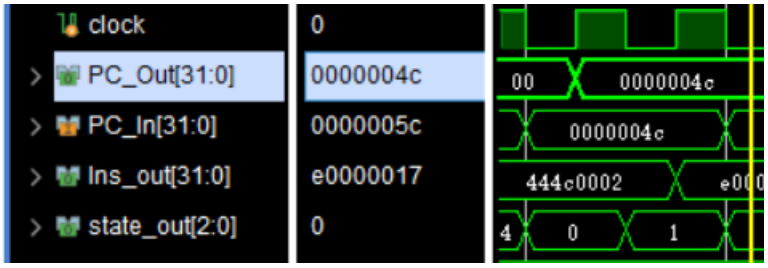
11号寄存器值和2号寄存器值不等，所以下一条指令的地址变为34H。

（16）**bltz　$12,-2 (<0,转 40)**



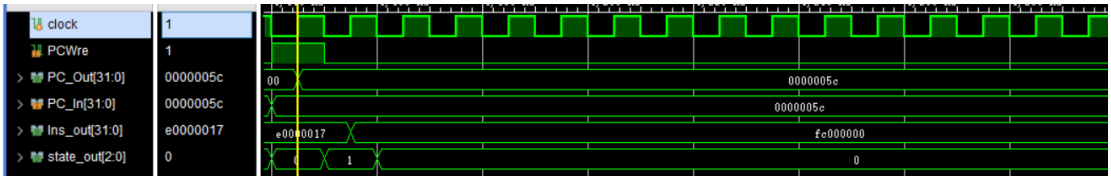
因为12号寄存器值为-1，小于0，所以下一条指令地址为40H。

（17）j　0x000005C



无条件跳转，所以下一条指令地址为5CH。

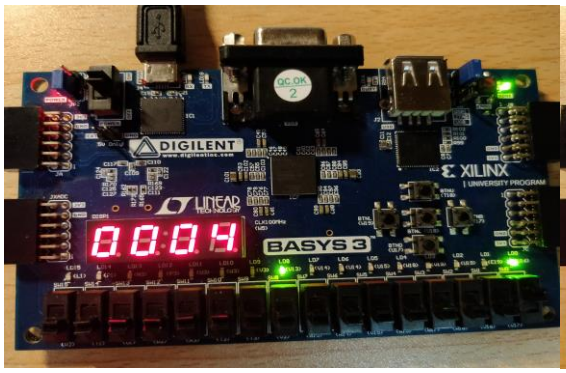
（18）halt



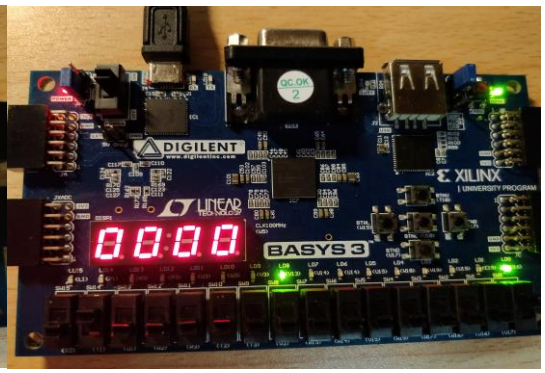
停机指令，PCWre无效，所以指令地址永远不会变，定格在5CH。

3.烧板展示(只展示前5条指令的WB阶段的相关信息)

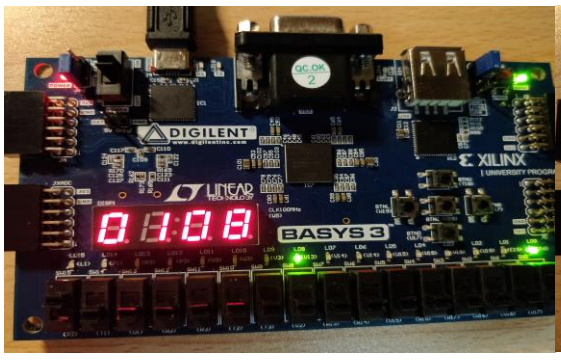
（1）addi　$1,$0,8



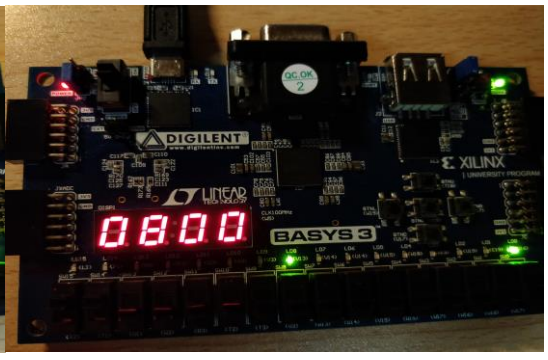
当前PC：下一条PC



RS地址：RS数据

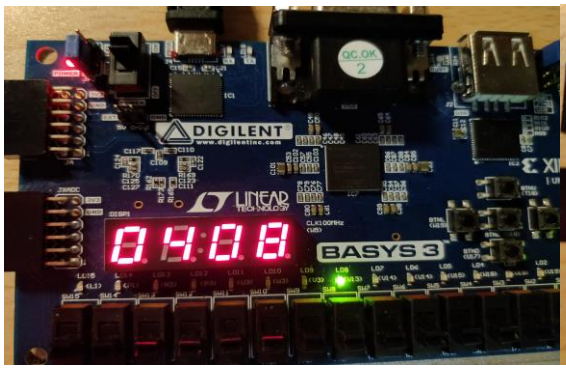


RT地址：RT数据



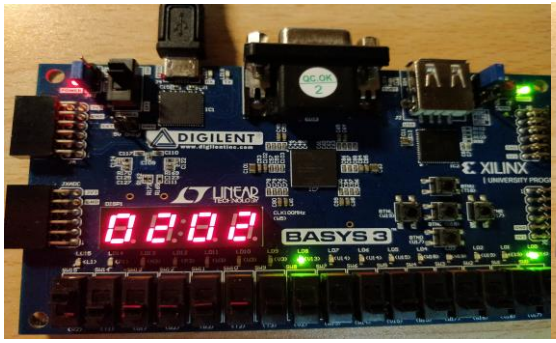
ALU计算结果：存储器数据

（2）ori　$2,$0,2

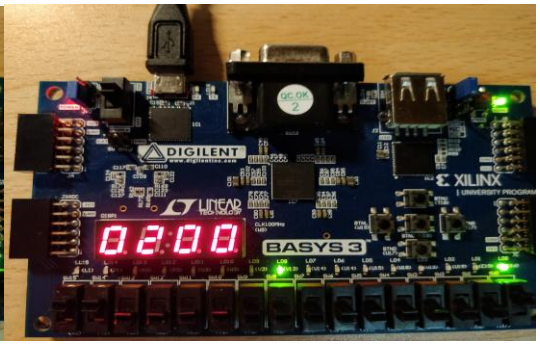


当前PC：下一条PC

RT地址：RT数据

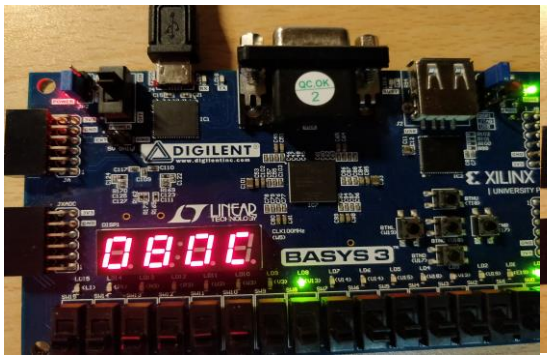


RS地址：RT数据

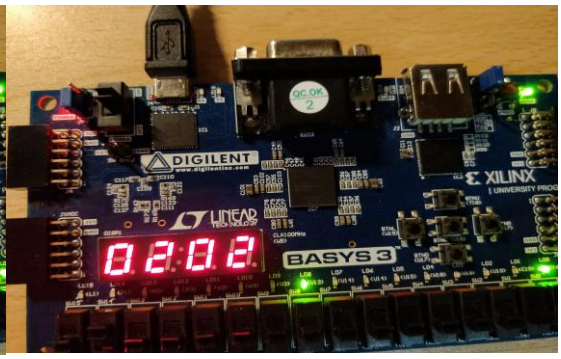


ALU计算结果：存储器数据

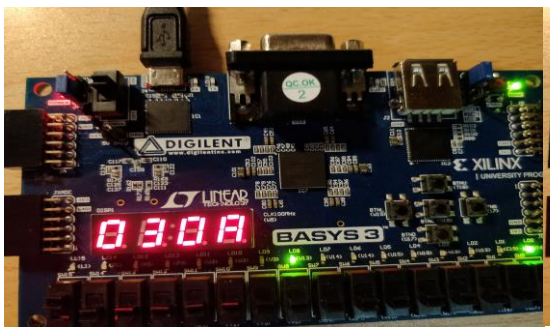
（3）xori　$3,**$2**,8



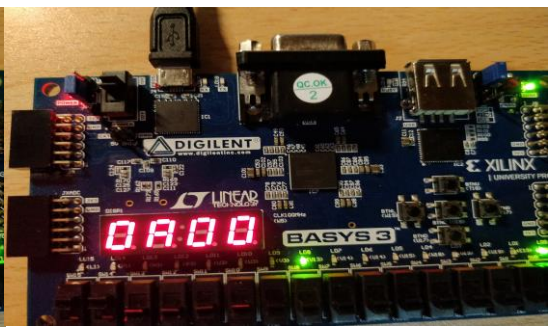
当前PC：下一条PC



RS地址：RS数据

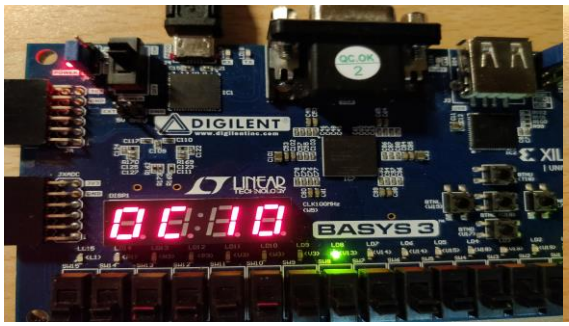


RT地址：RT数据

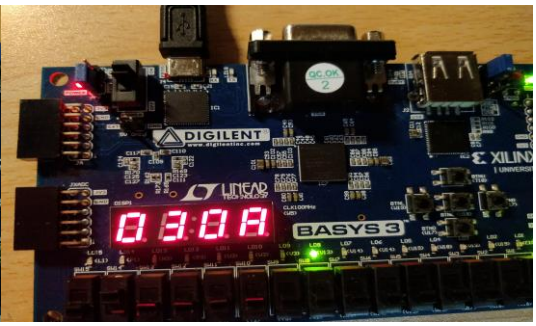


ALU计算结果：存储器数据

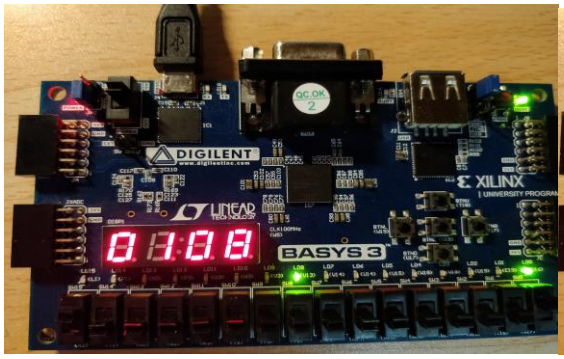
（4）sub　$4,**$3**,$1



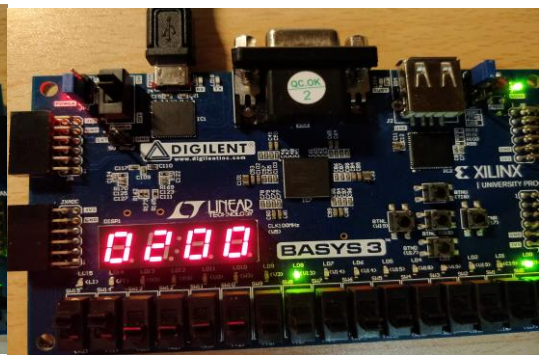
当前PC：下一条PC



RS地址：RS数据

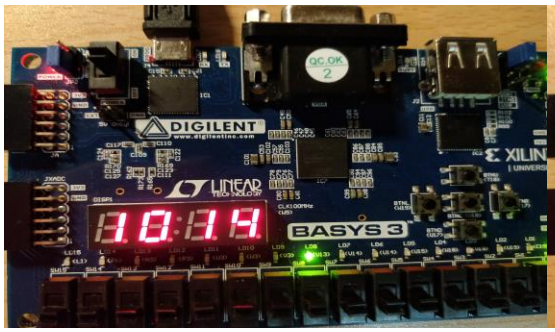


RT地址：RT数据

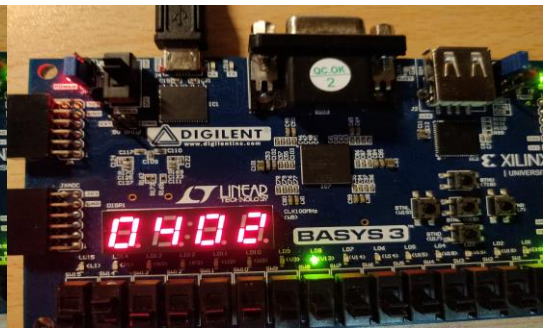


ALU计算结果：存储器数据

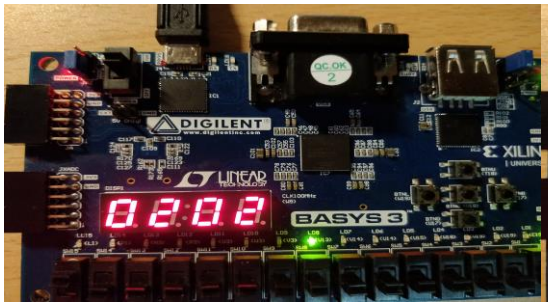
（5）and　$5,**$4**,$2



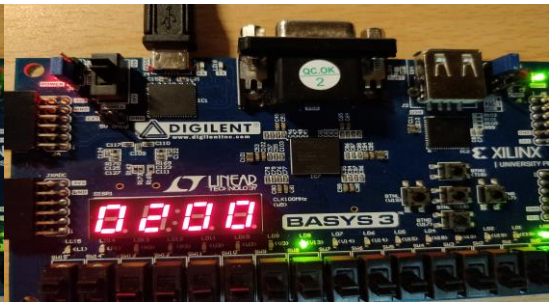
当前PC：下一条PC



RS地址：RS数据



RT地址：RT数据



ALU计算结果：存储器结果

PS：由于烧板要求更高更复杂，所以烧板的CPU程序与仿真的CPU程序有所不同，所以烧板的结果与仿真有点不同。

4.关于编译器：

只需把需要翻译的指令写进一个名为“instruction.txt”的文本里，当然路径要一致，即可获得指令的二进制代码，代码在“out.txt”里。实现这个编译器非常简单，只需要将相同的指令归类，并执行同样的读写操作即可。

六、实验心得

（1）多周期难处就在于得思考各个模块是该上升沿触发还是下降沿触发，并且每个人想法都不同，所以没有什么可以借鉴的。一开始，我也是随便选择上升沿下降沿，最后在波形中看到各种错误，去修改。我在实现 CPU 遇到了两个上升沿下降沿的问题：DBDR 不能用上升沿触发，因为在下降沿到来时，CPU 已经进入 WB 阶段，mRD 无效了，所以在上升沿的时候，已经不能读储存器，所以我应该在进入 WB 前先把数据读出来，所以应设置 DBDR 下降沿触发；而存储器不能不用时钟控制，因为要写的地址在上升沿时才到，而 mWR 有效在要写的地址出来之前，所以就可能乱修改了存储器的值，所以我用下降沿控制储存器写这样上升沿还是下降沿就确定了。写完 CPU，对时序电路的理解更加深刻了。总体上感觉有了单周期的铺垫，还是比较容易去实现多周期的，不难，但是还是要时间去磨出来。

（2）关于烧板，花了比实现 CPU 还久了一倍的时间，到现在还是不明白为什么？一开始，我照搬了上次烧板的显示模块和消抖模块，然后就发现板子按起来就像是单周期的样子，然后检查了两天，愣是没有找到错误。最后把状态显示成 LED 的样子，发现按起来不是单周期的样子了，但 PC 永远都是 0，这简直就把我搞心态崩了。后来，我又把 PCWre 用 LED 显示出来，发现 PCWre 本该只在 IF 阶段才有效的，变成 EXE、MEM、WB 阶段也有效了（我是在 ID 阶段置零的，后面的阶段没有赋值，照理来说，应该都会是 0），所以我又在 EXE、MEM、WB 阶段加上了赋值语句，最后终于成功了。总的来说，烧板比实现 CPU 难多了，我的板子要求太高了。

（3）经过了多周期的设计，了解了其与单周期的不同，多周期CPU在教材上是没有讲到的，所以实验课也是学习到新知识的，不全部是要将学到的知识实现出来，不是学不到新知识的，哪里都可以有新知识学的。