《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）	：	18 软件工程 2 班
时　　间	：	2019 年 12 月 15 日
组　　员	：	鲁睿 18342067 罗炜乐 18342069

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）addiu 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。

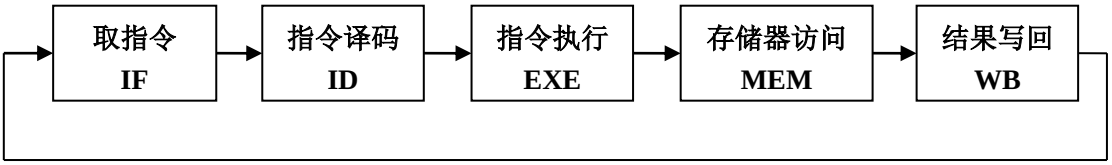


图 1　多周期 CPU 指令处理过程

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。

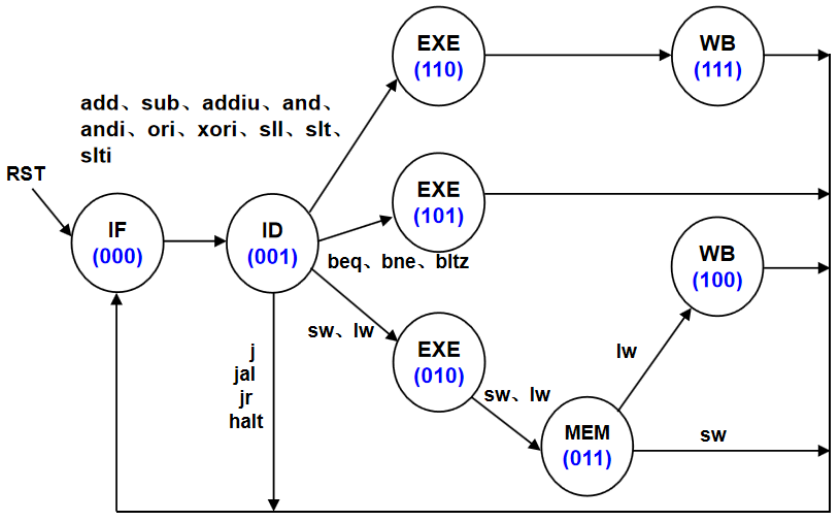


图 2 多周期 CPU 状态转移图

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。

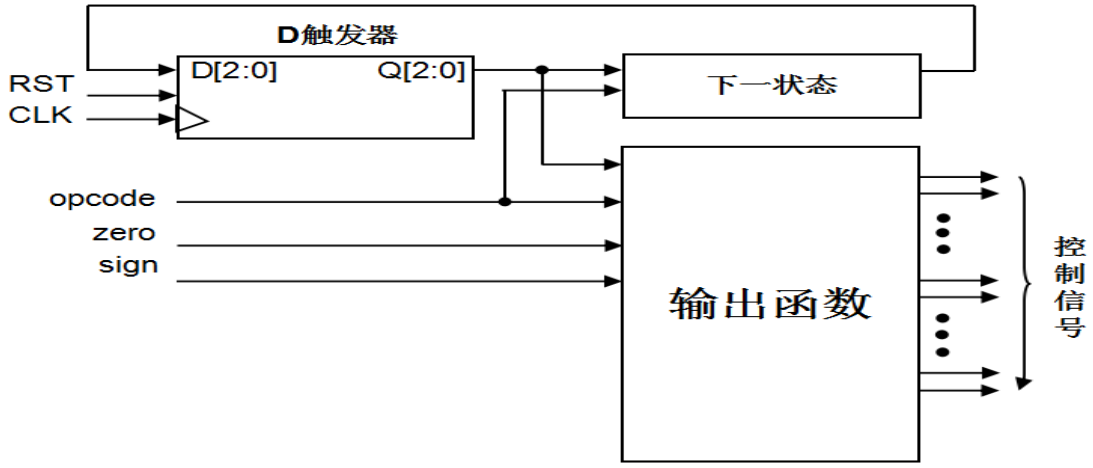


图 3 多周期 CPU 控制部件的原理结构图

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。

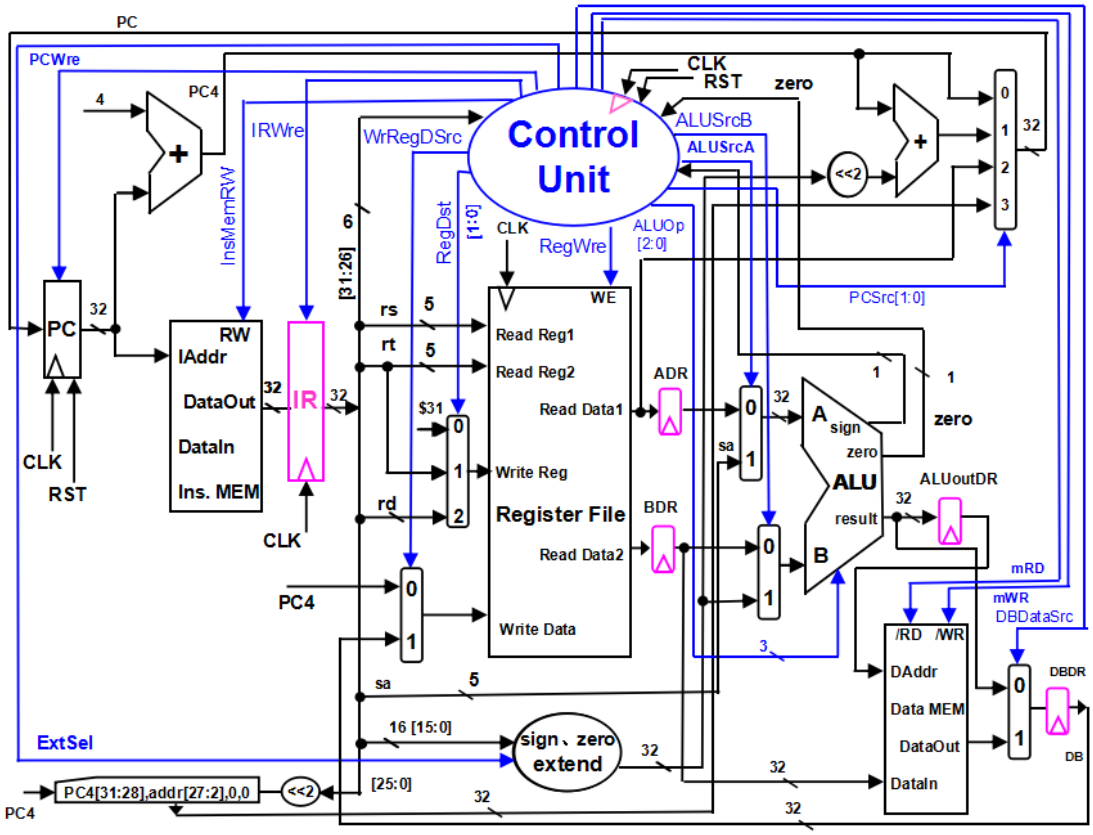


图 4　多周期 CPU 数据通路和控制线路图

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
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
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

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 = 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] == 0))) ? 1:0	比较 A<B 带符号
111	Y = A ⊕ B	异或

另外，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(表 3)，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。

表3 控制信号、执行状态转换表

状态	指令	控制信号														
		PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WrRegDSrc	InsMemRW	mRD	mWR	IRWre	ExtSel	PCSrc[1:0] (zero:0/1)	RegDst[1:0]	ALUOp[2:0]	
IF(000)	other_ins	1	x	x	x	0	x	1	x	x	1	x	xx		xxx	
	halt	0	x	x	x	0	x	0	x	x	0	x	xx	xx	xxx	
ID(001)	other_ins	0	x	x	x	0	x	0	x	x	0	x	xx	xx	xxx	
	j	0	x	x	x	0	x	0	x	x	0	x	11	xx	xxx	
	jal	0	x	x	x	1	0	0	x	x	0	x	11	00	xxx	
	jr	0	x	x	x	0	x	0	x	x	0	x	10	xx	xxx	
	halt	0	x	x	x	0	x	0	x	x	0	x	xx	xx	xxx	
EXE(110)	add	0	0	0	x	0	x	0	x	x	0	x	00	xx	000	
	sub	0	0	0	x	0	x	0	x	x	0	x	00	xx	001	
	addiu	0	0	1	x	0	x	0	x	x	0	1	00	xx	000	
	and	0	0	0	x	0	x	0	x	x	0	x	00	xx	100	
	andi	0	0	1	x	0	x	0	x	x	0	0	00	xx	100	
	ori	0	0	1	x	0	x	0	x	x	0	0	00	xx	011	
	xori	0	0	1	x	0	x	0	x	x	0	0	00	xx	111	
	sll	0	1	0	x	0	x	0	x	x	0	x	00	xx	010	
	slt	0	0	0	x	0	x	0	x	x	0	x	00	xx	110	
	slti	0	0	1	x	0	x	0	x	x	0	1	00	xx	101	
EXE(101)	beq	0	0	0	x	0	x	0	x	x	0	1	00/01	xx	001	
	bne	0	0	0	x	0	x	0	x	x	0	1	01/00	xx	001	
	bltz	0	0	0	x	0	x	0	x	x	0	1	00/01	xx	001	
EXE(010)	sw	0	0	1	x	0	x	0	x	x	0	1	00	xx	xxx	
	lw	0	0	1	x	0	x	0	x	x	0	1	00	xx	xxx	
MEM(011)	sw	1	x	x	0	0	x	0	0	1	0	x	xx	xx	xxx	
	lw	0	x	x	1	1	x	0	1	0	0	x	xx	xx	xxx	
WB(100)	lw	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx	
WB(111)	add	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx	
	sub	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx	
	addiu	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx	
	and	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx	
	andi	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx	
	ori	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx	
	xori	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx	
	sll	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx	
	slt	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx	
	slti	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx	

显而易见的是，在得到上表之后，我们就可以轻易地得到相关控制信号的逻辑表达式，这样就可以很方便地实现ControlUnit功能模块了。

一. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

二. 实验过程与结果

1. 多周期CPU与单周期CPU对比

考虑到已经实现了单周期CPU,在做多周期CPU时，可以借鉴单周期CPU的实现方法，下面对单周期CPU和多周期CPU做一个比较：

相同：

ALU模块相同，大部分寄存器组模块相同

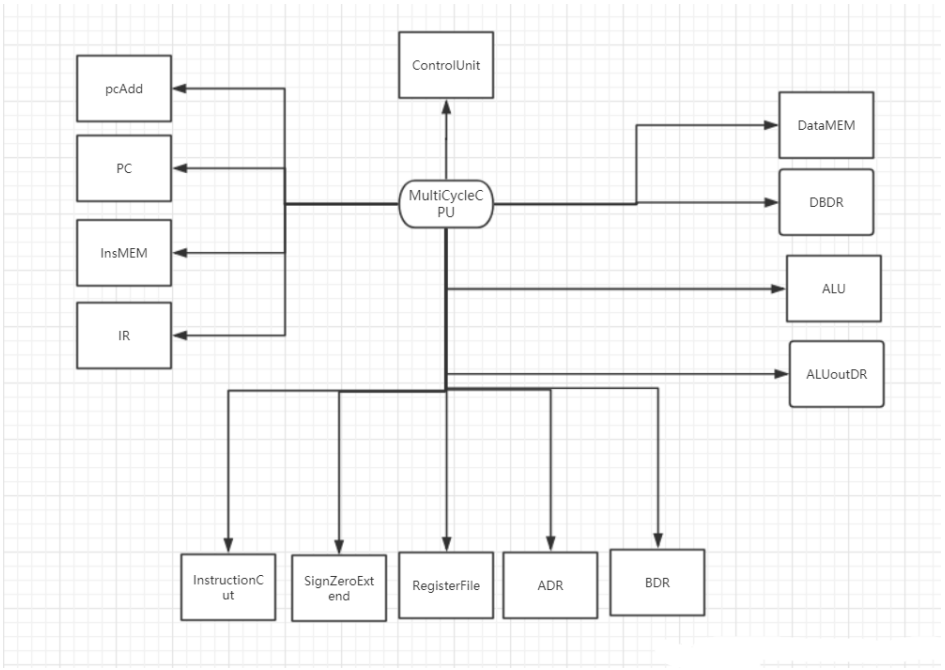
不同：

（1） 多周期CPU新增了指令寄存器IR，以及ADR、BDR、ALUoutDR、DBDR四个寄存器；

（2） ControlUnit.v模块有较大变化，主要是在其中加入了一个内置的状态寄存器，用于各个状态之间的跳转，并且各个控制信号也有一定调整；

（3） 新增了xori,jal,jr三条指令。

2. 多周期CPU整体模块概览



在多周期CPU的设计过程中，我们同样采用模块化设计的方法，，其中ControlUnit是核心控制模块，它兼具各个控制信号的切换以及状态切换的功能。

3. 各个模块实现

Hard_CPU.v

功能：作为顶层模块，用于连接mCPU、display、remove_shake、clock_slow几个模块，并且指定不同按键模式下数码管所显示的内容。

```
1.  `timescale 1ns / 1ps
2.
3.  module Hard_CPU(
4.      input[2:0] display_mode,
5.      input CLK, Reset, Button,
6.      output[3:0] AN, //数码管位选择信号
```

```
    output[7:0] Out
);

    wire[31:0] ALU_Out, CurPC, WriteData, Reg1Out, Reg2Out, instcode, IRInstruction, ADROut, BDROut, DBDROut, ALUoutDROut, Ext_Imm, ALU_Input_A, ALU_Input_B, newAddress, WriteRegAddr;
    wire myCLK, myReset;
    reg[3:0] store; //记录当前要显示位的值
    wire[2:0] currentState;

    mCPU multipleCPU(myCLK, myReset, CurPC, newAddress, instcode, IRInstruction, ADROut, BDROut, DBDROut, ALUoutDROut, Ext_Imm, Reg1Out, Reg2Out, ALU_Input_A, ALU_Input_B, WriteData, WriteRegAddr, currentState);
    remove_shake remove_shake_clk(CLK, !Button, myCLK);
    remove_shake remove_shake_reset(CLK, Reset, myReset);
    clk_slow slowclk(CLK, myReset, AN);
    display show_in_7segment(store, myReset, Out);

    always@(myCLK)begin
       case(AN)
            4'b1110:begin
                case(display_mode)
                    3'b000: store <= newAddress[3:0];
                    3'b010: store <= ADROut[3:0];
                    3'b100: store <= BDROut[3:0];
                    3'b110: store <= WriteData[3:0];
                    default: store <= {3'b000,currentState[0]};
                endcase
            end
            4'b1101:begin
                case(display_mode)
                    3'b000: store <= newAddress[7:4];
                    3'b010: store <= ADROut[7:4];
                    3'b100: store <= BDROut[7:4];
                    3'b110: store <= WriteData[7:4];
                    default: store <= {3'b000,currentState[1]};
                endcase
            end
            4'b1011:begin
                case(display_mode)
                    3'b000: store <= CurPC[3:0];
                    3'b010: store <= IRInstruction[24:21];
                    3'b100: store <= IRInstruction[19:16];
                    3'b110: store <= ALUoutDROut[3:0];
```

```
                    default: store <= {3'b000,currentState[2]};
                endcase
            end
            4'b0111:begin
                case(display_mode)
                    3'b000: store<= CurPC[7:4];
                    3'b010: store <= {3'b000,IRInstruction[25]};
                    3'b100: store <= {3'b000,IRInstruction[20]};
                    3'b110: store <= ALUoutDROut[7:4];
                    default: store <= 4'b0000;
                endcase
            end
        endcase
    end
endmodule
```

mCPU.v

功能：mCPU功能模块的顶层文件，连接各个功能模块

```
`timescale 1ns / 1ps

module mCPU(
    input CLK,
    input Reset,
    output[31:0] CurPC, nextAddr, instcode, IRInstruction, ADROut, BDROut, DBDROut, ALUoutDROut, Ext_Imm, Reg1Out, Reg2Out, ALU_Input_A, ALU_Input_B, WriteData,
    output[4:0] WriteRegAddr,
    output[2:0] currentState
);
    wire ExtSel, PCWre, InsMemRW, ALUSrcA, ALUSrcB, RD, WR, DBDataSrc, zero, sign, IRWre, WrRegDSrc, RegWre;
    wire[2:0] ALUOp;
    wire[1:0] PCSrc, RegDst;
    wire[31:0] MemOut, PC4, InsSrcImm, InsSrc1, InsSrc3, ALU_Out, WBData;

    PC pc(CLK, Reset, PCWre, nextAddr, CurPC);
    InsMem insmem(CurPC, InsMemRW, instcode);//not
    tempReg IR(CLK, instcode, IRWre, IRInstruction);//hkhk
    Adder pc4_adder(CurPC, 32'b00000000000000000000000000000100, PC4);
    Mux_3 reg_w_choose(RegDst, 5'b11111, IRInstruction[20:16],IRInstruction[15:11], WriteRegAddr);
```

```
    Mux_2 writedata_choose(WrRegDSrc, PC4, DBDROut, WriteData);
    RegisterFile regfile(RegWre, CLK, IRInstruction[25:21], IRInstruction[20:16], WriteRegAddr, WriteData, Reg1Out, Reg2Out);
    tempReg ADR(CLK, Reg1Out, 1'b1, ADROut);
    tempReg BDR(CLK, Reg2Out, 1'b1, BDROut);
    Extend extend(ExtSel, IRInstruction[15:0], Ext_Imm);
    Mux_2 alua_choose(ALUSrcA, ADROut, {27'b000000000000000000000000000,IRInstruction[10:6]}, ALU_Input_A);
    Mux_2 alub_choose(ALUSrcB, BDROut, Ext_Imm, ALU_Input_B);
    ALU alu(ALUOp, ALU_Input_A, ALU_Input_B, ALU_Out, zero, sign);
    tempReg ALUoutDR(CLK, ALU_Out, 1'b1, ALUoutDROut);
    tempReg DBDR(CLK, WBData, 1'b1, DBDROut);
    DataMem datamem(CLK, RD, WR, ALUoutDROut, BDROut, MemOut);
    Mux_2 db_choose(DBDataSrc, ALU_Out, MemOut, WBData);
    LeftShift_2 after_imm_extend(Ext_Imm, InsSrcImm);
    LeftShift_2 addr_shift({2'b00, PC4[31:28], IRInstruction[25:0]}, InsSrc3);
    Adder next_pc1(InsSrcImm, PC4, InsSrc1);
    Mux_4 nextpc_choose(PCSrc, PC4, InsSrc1, Reg1Out,InsSrc3, nextAddr);
    ControlUnit control_unit(CLK, IRInstruction[31:26], zero, sign, Reset, ExtSel, PCWre, RegWre, InsMemRW, RegDst, ALUSrcA, ALUSrcB, PCSrc, ALUOp, RD, WR, IRWre, WrRegDSrc, DBDataSrc, currentState);
endmodule

```

PC.v（该文件与单周期CPU的文件一样）

功能：根据控制信号PCWre，判断pc是否改变以及根据Reset信号判断是否重置。

实现思路：将时钟信号的上升沿和控制信号Reset作为敏感变量，使得pc在上升沿的时候发生改变或被重置。

```
`timescale 1ns / 1ps

module PC(
    input CLK, Reset, PCWre,
    input [31:0] newAddr,
    output reg [31:0] PCAddr
    );
    initial begin
        PCAddr = 0;
    end
    always@(posedge CLK or negedge Reset) begin
```

```
        if(Reset==0) PCAddr = 0;
        else if(PCWre) PCAddr = newAddr;
    end
endmodule
```

InsMem.v（该文件与单周期CPU的文件一样）

功能：依据当前pc和信号量InsMemRW，读取指令寄存器中，相对应地址的指令。

实现思路：将pc的输入作为敏感变量，当pc发生改变的时候，则进行指令的读取，根据相关的地址，输出指令寄存器中相对应的指令。

```
`timescale 1ns / 1ps

module InsMem(
    input [31:0] IAddr,
    input RW,
    output reg [31:0] ins
    );
    reg[7:0] mem[255:0];
    initial begin
        $readmemb("C:/Users/1/Desktop/mCPU/input.txt", mem);
    end
    always@(IAddr or RW) begin
        if (RW) ins = {mem[IAddr], mem[IAddr + 1], mem[IAddr + 2], mem[IAddr + 3]};
    end
endmodule
```

寄存器 ADR、BDR、ALUoutDR、DBDR

功能：下一周期上升沿到达时保存上一周期的状态，从而达到不同周期数据的稳定。

实现思路：在时钟上升沿将相应的数据写入寄存器中。四个寄存器功能相同，可以使用用一个模块实例得到。

```
`timescale 1ns / 1ps

module tempReg(
        input CLK,
        input [31:0] IData,
        input write,
        output reg[31:0] OData
    );
```

```
9.
10.     initial begin
11.         OData = 0;
12.     end
13.
14.     always@(posedge CLK) begin
15.         if (write == 1) OData <= IData;
16.     end
17. endmodule
```

IR（与 ADR 等寄存器共用同一个文件）
功能：为了使指令代码保持稳定;
实现思路：将时钟上升沿作为敏感信号，同时依据信号量 IRWre，对 IR 进行写入。

```
1.  `timescale 1ns / 1ps
2.
3.  module tempReg(
4.          input CLK,
5.          input [31:0] IData,
6.          input write,
7.          output reg[31:0] OData
8.      );
9.
10.     initial begin
11.         OData = 0;
12.     end
13.
14.     always@(posedge CLK) begin
15.         if (write == 1) OData <= IData;
16.     end
17. endmodule
```

Adder.v （该文件与单周期 CPU 的文件一样）
功能：该模块用于计算下一个 PC 值

```
1.  `timescale 1ns / 1ps
2.
3.  module Adder(
4.      input[31:0] old1,
5.      input[31:0] old2,
6.      output[31:0] res//reg?address?
7.      );
8.      assign res = old1 + old2;
9.  endmodule
```

Mux_2.v

该模块共有 4 个位置被使用到，分别是用于 Write reg 信号选择、aluA 数据选择、aluB 数据选择、以及 DB 数据选择

```
1.  `timescale 1ns / 1ps
2.
3.  module Mux_2(
4.      input Select,
5.      input[31:0] in1,
6.      input[31:0] in2,
7.      output[31:0] out
8.      );
9.      assign out = Select ? in2 : in1;
10. endmodule
```

RegisterFile.v（该文件与单周期 CPU 的文件一样）

功能：寄存器组，通过控制单元输出的控制信号，进行相对应的读或写操作。

实现思路：当 ReadReg1 或者 ReadReg2 发生改变的时候，即对寄存器组进行数据读取。至于数据写入，则选择在时钟的下降沿时候进行操作，同时写入信号（RegWre）必须为 1 且写入寄存器的地址不能为$0。至于数据的写入地址则在任意信号发生改变则依据写入地址的控制信号（RegDst）进行修改，使得其在数据写回阶段确保为正确的写入地址。

```
1.  `timescale 1ns / 1ps
2.  module RegisterFile(
3.      input WE,
4.      input CLK,
5.      input[4:0] ReadReg1,
6.      input[4:0] ReadReg2,
7.      input[4:0] WriteReg,
8.      input[31:0] WriteData,
9.      output[31:0] ReadData1,
10.     output[31:0] ReadData2
11.     );
12.     reg[31:0] registers[0:31];
13.     integer i;
14.     initial begin
15.         for (i = 0; i < 32; i = i + 1) registers[i] <= 0;
16.     end
17.     assign ReadData1 = ReadReg1 ? registers[ReadReg1] : 0;
18.     assign ReadData2 = ReadReg2 ? registers[ReadReg2] : 0;
19.     assign Reg31 = registers[31];
20.     always@(posedge CLK) begin
21.         if (WE && WriteReg) registers[WriteReg] <= WriteData;
22.     end
23. endmodule
```

Extend.v （该文件与单周期 CPU 的文件一样）

功能：用于立即数的符号位扩展，当 ExtSel 信号为 0 时做无符号扩展，为 1 时做有符号扩展

```
1.  `timescale 1ns / 1ps
2.
3.  module Extend(
4.      input ExtSel,
5.      input [15:0] immed,
6.      output [31:0] extendImmed
7.      );
8.      assign extendImmed = {ExtSel && immed[15] ? 16'hffff : 16'h0000, immed};

9.  endmodule
```

ALU.v （该文件与单周期 CPU 的文件一样）

该部分为算术逻辑单元，用于逻辑指令计算和跳转指令比较。ALUOp 用于控制算数的类型，A、B 为输入数，result 为运算结果，zero、sign 主要用于 beq、bne、bltz 等指令 的判断。

ALU 算术逻辑单元的功能是根据控制信号从输入的数据中选取对应的操作数，根据操作码进行运算并输出结果与零标志位。具体的操作逻辑见 表 2

```
1.  `timescale 1ns / 1ps
2.
3.  module ALU(
4.      input [2:0] ALUopcode,
5.      input [31:0] rega,
6.      input [31:0] regb,
7.      output reg [31:0] result,
8.      output zero,
9.      output sign
10.     );
11.     //    parameter ALU_ADD = 3'b000;
12.     //    parameter ALU_SUB = 3'b001;
13.     //    parameter ALU_SLL = 3'b010;
14.     //    parameter ALU_OR = 3'b011;
15.     //    parameter ALU_AND = 3'b100;
16.     //    parameter ALU_SLTU = 3'b101;
17.     //    parameter ALU_SLT = 3'b110;
18.     //    parameter ALU_XOR = 3'b111;
19.     assign zero = (result == 0) ? 1 : 0;
20.     assign sign = result[31];
21.     always @( ALUopcode or rega or regb ) begin
22.         case (ALUopcode)
```

```
            3'b000 : result = rega + regb;
            3'b001 : result = rega - regb;
            3'b010 : result = regb << rega;
            3'b011 : result = rega | regb;
            3'b100 : 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 : result = rega ^ regb;    //异或
            default : result = 8'h00000000;
        endcase
    end
endmodule
```

DataMem.v （此文件相比单周期 CPU 更改了大小端，单周期 CPU 大小端设置反了。）

该部分控制内存存储，用于内存存储、读写。用 256 大小的 8 位寄存器数组模拟内存， 采用小端模式。DataMenRW 控制内存读写。由于指令为真实地址，所以不需要左移 2 位.

```
`timescale 1ns / 1ps

module DataMem(
    input CLK,
    input mRD,
    input mWR,
    input[31:0] DAddr,
    input[31:0] DataIn,
    output reg[31:0] DataOut
    );

    reg[7:0] dataMemory[255:0];
    integer i;
    initial begin
        for (i = 0; i < 256; i = i + 1) dataMemory[i] <= 0;//没有自增运算符
    end

    always@(mRD or DAddr) begin
        if (mRD) begin
```

```
            DataOut[31:24] <= dataMemory[DAddr+3];
            DataOut[23:16] <= dataMemory[DAddr+2];
            DataOut[15:8] <= dataMemory[DAddr+1];
            DataOut[7:0] <= dataMemory[DAddr];
        end
    end

    always@(negedge CLK) begin
        if (mWR) begin
            dataMemory[DAddr+3] <= DataIn[31:24];
            dataMemory[DAddr+2] <= DataIn[23:16];
            dataMemory[DAddr+1] <= DataIn[15:8];
            dataMemory[DAddr] <= DataIn[7:0];
        end
    end
endmodule
```

LeftShift.v （该文件与单周期 CPU 的文件一样）

该模块分别用于 j 类指令的立即数左移运算、以及 I 类指令的立即数左移运算。

```
`timescale 1ns / 1ps

module LeftShift_2(
    input[31:0] in,
    output[31:0] out
    );
    assign out = in << 2;
endmodule
```

ControlUnit.v

功能：控制单元，依据指令的操作码（op)、标记符（ZERO）以及当前 CPU 状态，依据表三 控制信号、指令以及执行状态之间的相互关系，输出相匹配控制信号量。

实现思路：设计一个 CPU 当前状态的有限状态机，依据指令的操作码（op)、当前状态以及重置信号（RST)，其状态在下降沿(为了避免竞争冒险)的时候发生，同时依据当前的状态、操作码（op）以及标记符（ZERO）修改并且输出控制信号。

以下为当前状态的有限状态机的代码。

以指令 addiu　$1,$0,8 为例，在一开始执行时，状态初始化为 IF；

然后匹配到 IF 状态，下一状态置为 ID；

下一个周期匹配到 ID，由于 addiu 不属于 beq,bne,bltz,sw,lw,j,jal,jr,halt,置为 EXEAL

下一个周期匹配到 EXEAL，下一状态置为 WBAL

```
initial begin
    currentState = IF;
end

//PCWre, RegWre, mWR 是写信号，一个阶段的周期不能写其他阶段的值，除了在最后可以写 PC 外
always@(negedge CLK or negedge reset) begin
    if(reset == 0) begin  //初始化什么也不能写
        currentState <= IF;
        PCWre = 0;
        mWR = 0;
        IRWre = 0;
    end
    else begin
        case (currentState)
            IF: begin
                currentState <= ID;
                PCWre = 0;
                mWR = 0;
                IRWre = 1;
            end
            ID: begin
                case (op)
                    INS_BEQ, INS_BNE, INS_BLTZ: currentState <= EXEBR;
                    INS_SW, INS_LW: currentState <= EXELS;
                    INS_J, INS_JAL, INS_JR, INS_HALT: begin
                        if (op == INS_HALT) PCWre = 0;
                        else PCWre = 1;//跳转指令结束，写 pc
                        currentState <= IF;
                    end
                    default: currentState <= EXEAL;
                endcase
                IRWre = 0;
            end
            EXEAL:begin
                currentState <= WBAL;
            end
            EXELS:begin
                currentState <= MEM;
                //如果指令为 SW，mem 阶段允许写内存
                if (op == INS_SW)  mWR = 1;
            end
            MEM:begin
                if (op == INS_SW) begin
```

```
                    currentState = IF;
                    //指令 sw 结束，写 PC
                    PCWre=1;
                end
                else begin
                    currentState = WBL;
                    //指令 lw 的 wb 阶段写寄存器
                end
                mWR = 0;
            end
            default: begin
                currentState <= IF;
                PCWre = 1;
            end
        endcase
    end
end
```

本模块为多周期 CPU 中最重要的模块，需要注意部分信号在特定的 CPU 状态才能输出相应的使能，否则将出现错误。

```
`timescale 1ns / 1ps

module ControlUnit(
    input CLK,
    input[5:0] op,
    input zero,
    input sign,
    input reset,
    output ExtSel,
    output reg PCWre,
    output RegWre,
    output InsMemRW,
    output[1:0] RegDst,
    output ALUSrcA,
    output ALUSrcB,
    output[1:0] PCSrc,
    output[2:0] ALUOp,
    output mRD,
    output reg mWR,
    output reg IRWre,
    output WrRegDSrc,
    output DBDataSrc,
    output reg[2:0] currentState//try
```

```
24.     );
25.     //对指令和 ALU 的操作码定义常量
26.     parameter INS_ADD = 6'b000000;
27.     parameter INS_SUB = 6'b000001;
28.     parameter INS_ADDIU = 6'b000010;
29.     parameter INS_AND = 6'b010000;
30.     parameter INS_ANDI = 6'b010001;
31.     parameter INS_ORI = 6'b010010;
32.     parameter INS_XORI = 6'b010011;
33.     parameter INS_SLL = 6'b011000;
34.     parameter INS_SLTI = 6'b100110;
35.     parameter INS_SLT = 6'b100111;
36.     parameter INS_SW = 6'b110000;
37.     parameter INS_LW = 6'b110001;
38.     parameter INS_BEQ = 6'b110100;
39.     parameter INS_BNE = 6'b110101;
40.     parameter INS_BLTZ = 6'b110110;
41.     parameter INS_J = 6'b111000;
42.     parameter INS_JR = 6'b111001;
43.     parameter INS_JAL = 6'b111010;
44.     parameter INS_HALT = 6'b111111;
45.
46.     parameter ALU_ADD = 3'b000;
47.     parameter ALU_SUB = 3'b001;
48.     parameter ALU_SLL = 3'b010;
49.     parameter ALU_OR = 3'b011;
50.     parameter ALU_AND = 3'b100;
51.     parameter ALU_SLTU = 3'b101;
52.     parameter ALU_SLT = 3'b110;
53.     parameter ALU_XOR = 3'b111;
54.
55.     parameter[2:0] IF = 3'b000;
56.     parameter[2:0] ID = 3'b001;
57.     parameter[2:0] EXELS = 3'b010;//lw sw
58.     parameter[2:0] MEM = 3'b011;
59.     parameter[2:0] WBL= 3'b100;//lw
60.     parameter[2:0] EXEBR = 3'b101;//beq bne bltz
61.     parameter[2:0] EXEAL = 3'b110;//Arithmetic and Logic
62.     parameter[2:0] WBAL = 3'b111;
63.
64.
65.
66.     initial begin
67.         currentState = IF;
```

```
68.     end
69.
70.     //PCWre, mWR 是写信号，一个阶段的周期不能写其他阶段的值，除了在最后可以写 PC 外
71.     always@(negedge CLK or negedge reset) begin
72.         if(reset == 0) begin  //初始化什么也不能写
73.             currentState <= IF;
74.             PCWre = 0;
75.             mWR = 0;
76.             IRWre = 0;
77.         end
78.         else begin
79.             case (currentState)
80.                 IF: begin
81.                     currentState <= ID;
82.                     PCWre = 0;
83.                     mWR = 0;
84.                     IRWre = 1;
85.                 end
86.                 ID: begin
87.                     case (op)
88.                         INS_BEQ, INS_BNE, INS_BLTZ: currentState <= EXEBR;
89.                         INS_SW, INS_LW: currentState <= EXELS;
90.                         INS_J, INS_JAL, INS_JR, INS_HALT: begin
91.                             if (op == INS_HALT) PCWre = 0;
92.                             else PCWre = 1;//跳转指令结束，写 pc
93.                             currentState <= IF;
94.                         end
95.                         default: currentState <= EXEAL;
96.                     endcase
97.                     IRWre = 0;
98.                 end
99.                 EXEAL:begin
100.                     currentState <= WBAL;
101.                 end
102.                 EXELS:begin
103.                     currentState <= MEM;
104.                     //如果指令为 SW，mem 阶段允许写内存
105.                     if (op == INS_SW)  mWR = 1;
106.                 end
107.                 MEM:begin
108.                     if (op == INS_SW) begin
109.                         currentState = IF;
110.                         //指令 sw 结束，写 PC
111.                         PCWre=1;
```

```
112.                        end
113.                        else begin
114.                            currentState = WBL;
115.                            //指令 lw 的 wb 阶段写寄存器
116.                        end
117.                        mWR = 0;
118.                    end
119.                    default: begin
120.                        currentState <= IF;
121.                        PCWre = 1;
122.                    end
123.                endcase
124.            end
125.        end
126.        assign RegWre = currentState == WBAL || currentState == WBL || currentS
    tate == ID && op == INS_JAL;
127.        assign ALUSrcA = op == INS_SLL;
128.        assign ALUSrcB = op == INS_ADDIU || op == INS_ANDI || op == INS_ORI ||
    op == INS_XORI || op == INS_SLTI || op == INS_SW || op == INS_LW;
129.        assign DBDataSrc = op == INS_LW;
130.        assign WrRegDSrc = op != INS_JAL;
131.        assign InsMemRW = 1;
132.        assign mRD = op == INS_LW;
133.        assign RegDst = op == INS_JAL ? 2'b00: op == INS_ADD || op == INS_SUB |
    | op == INS_AND || op == INS_SLT || op == INS_SLL ? 2'b10 : 2'b01;
134.        assign ExtSel = op != INS_ANDI && op != INS_ORI && op != INS_XORI;
135.        assign PCSrc = op == INS_J || op == INS_JAL ? 2'b11 : op == INS_JR ? 2'
    b10 : (op == INS_BEQ && zero == 1) || (op == INS_BNE && zero == 0) || (op ==
     INS_BLTZ && sign == 1) ? 2'b01 : 2'b00;
136.        assign ALUOp = op == INS_SUB || op == INS_BEQ || op == INS_BNE || op ==
     INS_BLTZ ? ALU_SUB:
137.                        op == INS_SLL ? ALU_SLL:
138.                        op == INS_ORI ? ALU_OR:
139.                        op == INS_ANDI || op == INS_AND ? ALU_AND:
140.                        op == INS_SLT || op == INS_SLTI ? ALU_SLT:
141.                        op == INS_XORI ? ALU_XOR : ALU_ADD;
142. endmodule
```

remove_shake.v（该文件与单周期 CPU 的文件一样，但忘记在单周期的实验报告放出来）

功能：按键消抖

实现思路：按键抖动的接触时间是十分短，由于 basys3 开发板上本来就有时钟，可以使用开发板上的时钟在按键发生时计时，达到一定时长才认定为是按键。

```
1.  `timescale 1ns / 1ps
2.  module remove_shake(
3.      input clk, key_in,
4.      output key_out);
5.      parameter SAMPLE_TIME = 20000;
6.      reg[21:0] count_low;
7.      reg[21:0] count_high;
8.      reg key_out_reg;
9.
10.     always@(posedge clk) begin
11.         count_low <= !key_in ? 0 : count_low + 1;
12.         count_high <= key_in ? 0 : count_high + 1;
13.         if(count_high == SAMPLE_TIME)
14.             key_out_reg <= 1;
15.         else if(count_low == SAMPLE_TIME)
16.             key_out_reg <= 0;
17.     end
18.     assign key_out = !key_out_reg;
19. endmodule
```

clk_slow.v（该文件与单周期 CPU 的文件一样，但忘记在单周期的实验报告放出来）

功能：时钟分频模块，使得数码管有合适的刷新速度。

实现思路：与消抖思路类似，运用开发板上自带的时钟进行记录，每达到 100000 次就改变一次显示数码管。

```
1.  `timescale 1ns / 1ps
2.
3.  module clk_slow(input CLK, input Reset, output reg[3:0] AN);
4.      reg[16:0] hide;
5.      parameter DIVISION = 100000;
6.      initial begin
7.          hide <= 0;
8.          AN <= 4'b0111;
9.      end
10.     //数码管开始刷新
11.     always@(posedge CLK) begin
12.         if(Reset == 0)begin
13.             hide <= 0;
```

```
            AN <= 4'b0000;

        end
        else begin
            hide <= hide + 1;//分频功能
            if(hide == DIVISION) begin
                hide <= 0;
                case(AN)
                    4'b1110:begin
                        AN <= 4'b1101;
                    end
                    4'b0000:begin
                        AN <= 4'b1101;
                    end
                    4'b1101:begin
                        AN <= 4'b1011;
                    end
                    4'b1011: begin
                        AN <= 4'b0111;
                    end
                    4'b0111: begin
                        AN <= 4'b1110;
                    end
                endcase
            end
        end
    end
endmodule
```

display.v（该文件与单周期 CPU 的文件一样，但忘记在单周期的实验报告放出来）
功能：7 段数码管显示功能模块

```
`timescale 1ns / 1ps

module display(
    input[3:0] Store,
    input Reset,
    output reg[7:0] Out
    );
    always@(Store or Reset)begin
        if(Reset == 0) begin
            Out = 8'b01111111;
        end
```

```
        else begin
            case(Store)
                4'b0000 : Out = 8'b1100_0000; //0;  '0'-亮灯,  '1'-熄灯
                4'b0001 : Out = 8'b1111_1001; //1
                4'b0010 : Out = 8'b1010_0100; //2
                4'b0011 : Out = 8'b1011_0000; //3
                4'b0100 : Out = 8'b1001_1001; //4
                4'b0101 : Out = 8'b1001_0010; //5
                4'b0110 : Out = 8'b1000_0010; //6
                4'b0111 : Out = 8'b1101_1000; //7
                4'b1000 : Out = 8'b1000_0000; //8
                4'b1001 : Out = 8'b1001_0000; //9
                4'b1010 : Out = 8'b1000_1000; //A
                4'b1011 : Out = 8'b1000_0011; //b
                4'b1100 : Out = 8'b1100_0110; //C
                4'b1101 : Out = 8'b1010_0001; //d
                4'b1110 : Out = 8'b1000_0110; //E
                4'b1111 : Out = 8'b1000_1110; //F
                default : Out = 8'b0000_0000; //不亮
            endcase
        end
    end
endmodule
```

4. 编写编译器，将 MIPS 汇编程序编译为二进制机器码
   功能：此步骤用于我们将要进行的测试步骤的前置功能，即产生一个测试用的二进制代码文件，然后我们就可以用它来进行测试。
   实现思路：先使用哈希表将指令名字和相应二进制码对应，再遍历将各种符号’$’ ’,’ ’(’ ’)’删去，从而使用 sstream 处理每条指令，将其放到 vector 容器里。根据每条汇编指令(已在 vector 中)的各部分的长度(例如：**jal 0x0000050 我看作长度为 2，slt $8,$13,$1 我看作长度为 4**）进行 case 的分类，再通过条件判断语句确认是哪条指令，再根据该指令把已经 push 到 vector 容器的数字变成相应的 rs、rt、rd、immediate 等。

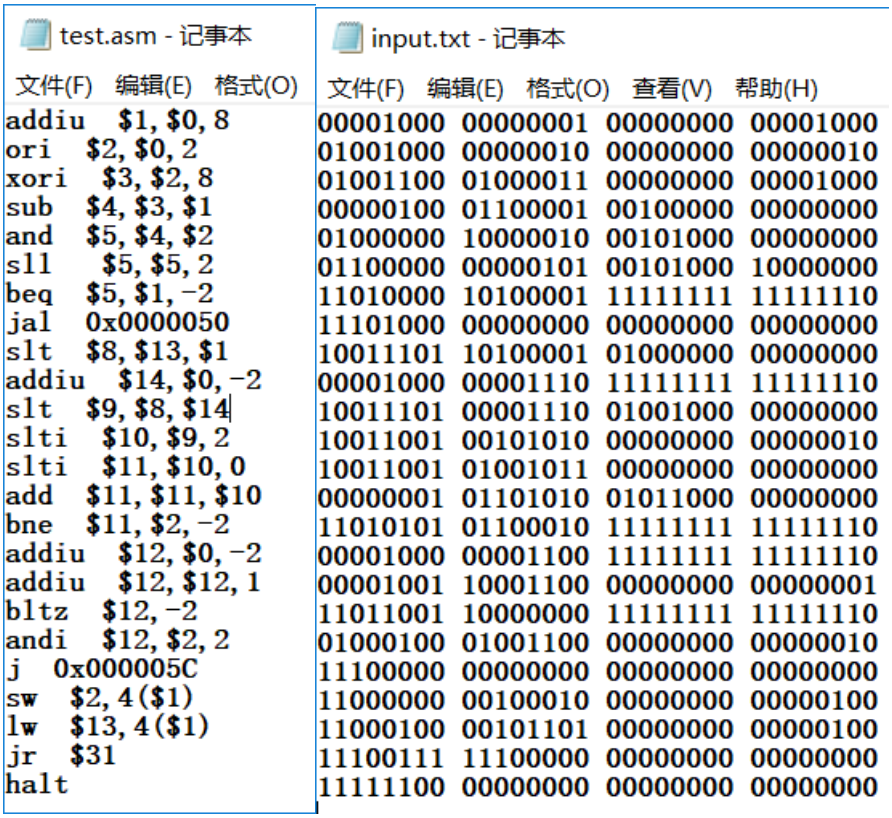
```
#include <fstream>
#include <sstream>
#include <unordered_map>
#include <string>
#include <vector>
#include <bitset>
using namespace std;
#define BINSTR(str, len) (bitset<len>(stoi(str, nullptr, 0)).to_string())

```

```
string complier(string str);

int main() {
    ifstream fin("test.asm");
    ofstream fout("input.txt");
    string str;
    while (getline(fin, str)) {
        str = complier(str);
        for (int i = 0; i < str.size(); ++i) {
            fout << str[i];
            if (i % 8 == 7) {
                fout << ' ';
            }
        }
        fout << '\n';
    }
    return 0;
}

string complier(string str) {
    static unordered_map<string, string> mp{
        {"add", "000000"},
        {"sub", "000001"},
        {"addiu", "000010"},
        {"and", "010000"},
        {"andi", "010001"},
        {"ori", "010010"},
        {"xori", "010011"},
        {"sll", "011000"},
        {"slti", "100110"},
        {"slt", "100111"},
        {"sw", "110000"},
        {"lw", "110001"},
        {"beq", "110100"},
        {"bne", "110101"},
        {"bltz", "110110"},
        {"j", "111000"},
        {"jr", "111001"},
        {"jal", "111010"},
        {"halt", "111111"}
    };
    for (auto & c : str) {
        if (c == '$' || c == '(' || c == ')' || c == ',') {
            c = ' ';
```

```
54.         }
55.     }
56.     vector<string> vec_str;
57.     istringstream iss(str);
58.     while (iss >> str) {
59.         vec_str.push_back(str);
60.     }
61.     switch (vec_str.size())
62.     {
63.     case 0:
64.         return "";
65.         break;
66.     case 1:
67.         str = mp[vec_str[0]] + string(26, '0');
68.         break;
69.     case 2:
70.         str = mp[vec_str[0]];
71.         if (vec_str[0] == "jr") {
72.             str += BINSTR(vec_str[1], 5) + string(21, '0');
73.         }
74.         else {
75.             str += bitset<26>(stoi(vec_str[1], nullptr, 0) >> 2).to_string()
    ;
76.         }
77.         break;
78.     case 3:
79.         str = mp[vec_str[0]] + BINSTR(vec_str[1], 5) + string(5, '0') + BINS
    TR(vec_str[2], 16);
80.         break;
81.     case 4:
82.         str = mp[vec_str[0]];
83.         if (vec_str[0] == "sw" || vec_str[0] == "lw") {
84.             str += BINSTR(vec_str[3], 5) + BINSTR(vec_str[1], 5) + BINSTR(ve
    c_str[2], 16);
85.         }
86.         else if (vec_str[0] == "sll") {
87.             str += string(5, '0') + BINSTR(vec_str[2], 5) + BINSTR(vec_str[1
    ], 5) + BINSTR(vec_str[3], 5) + string(6, '0');
88.         }
89.         else if (vec_str[0] == "beq" || vec_str[0] == "bne") {
90.             str += BINSTR(vec_str[1], 5) + BINSTR(vec_str[2], 5) + BINSTR(ve
    c_str[3], 16);
91.         }
92.         else if (vec_str[0].find('i') != vec_str[0].npos) {
```

```
93.             str += BINSTR(vec_str[2], 5) + BINSTR(vec_str[1], 5) + BINSTR(ve
    c_str[3], 16);
94.         }
95.         else {
96.             str += BINSTR(vec_str[2], 5) + BINSTR(vec_str[3], 5) + BINSTR(ve
    c_str[1], 5) + string(11, '0');
97.         }
98.     default:
99.         break;
100.     }
101.     return str;
102. }
```

运行该编译器的结果展示如下：



5. 多周期 CPU 功能测试

首先，将测试用的指令读入，然后进行仿真检验。

测试指令转换如下表：

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	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	xori $3,$2,8	010011	00010	00011	0000 0000 0000 1000	=	4c430008
0x0000000C	sub $4,$3,$1	000001	00011	00001	0010 0000 0000 0000	=	04612000

0x00000010	and $5,**$4**,$2	010000	00100	00010	0010 1000 0000 0000	=	40822800
0x00000014	**sll $5,$5,2**	011000	00000	00101	0010 1000 0000 0000	=	60052800
0x00000018	**beq $5,$1,-2(=,转 14)**	110100	00101	00001	1111 1111 1111 1110	=	d052fffe
0x0000001C	**jal 0x0000050**	111010	00000	00000	0000 0000 0001 0100	=	e8000014
0x00000020	**slt $8,$13,$1**	100111	01101	00001	0100 0000 0000 0000	=	9da14000
0x00000024	**addiu $14,$0,-2**	000010	00000	01110	1111 1111 1111 1110	=	080efffe
0x00000028	**slt $9,$8,$14**	100111	01000	01110	0100 1000 0000 0000	=	9d0e4800
0x0000002C	**slti $10,$9,2**	100110	01001	01010	0000 0000 0000 0010	=	992a0002
0x00000030	**slti $11,$10,0**	100110	01010	01011	**0000 0000 0000 0000**	=	994b0000
0x00000034	add $11,**$11**,$10	000000	01011	01010	**0101 1000 0000 0000**	=	016a5800
0x00000038	**bne $11,$2,-2 (≠, 转 34)**	110101	01011	00010	**1111 1111 1111 1110**	=	d562fffe
0x0000003C	addiu $12,$0,-2	000010	00000	01100	**1111 1111 1111 1110**	=	080cfffe
0x00000040	addiu $12,$12,1	000010	01100	01100	**0000 0000 0000 0001**	=	098c0001
0x00000044	**bltz $12,-2 (<0, 转 40)**	110110	01100	00000	**1111 1111 1111 1110**	=	d980fffe
0x00000048	andi $12,$2,2	010001	00010	01100	**0000 0000 0000 0010**	=	444c0002
0x0000004C	**j 0x000005C**	111000	00000	00000	**0000 0000 0000 0000**	=	e0000000
0x00000050	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	=	c0220004
0x00000054	lw $13,4($1)	110001	00001	01101	0000 0000 0000 0100	=	c42d0004
0x00000058	**jr $31**	111001	11111	00000	0000 0000 0000 0000	=	e7e00000
0x0000005C	**halt**	111111	00000	00000	0000000000000000	=	FC000000

表 4 测试指令转换表

注：上表 4 中的指令转化除了用于检验多周期 CPU 的功能是否正常外，还用来检验前面实现的编译器功能是否正常。

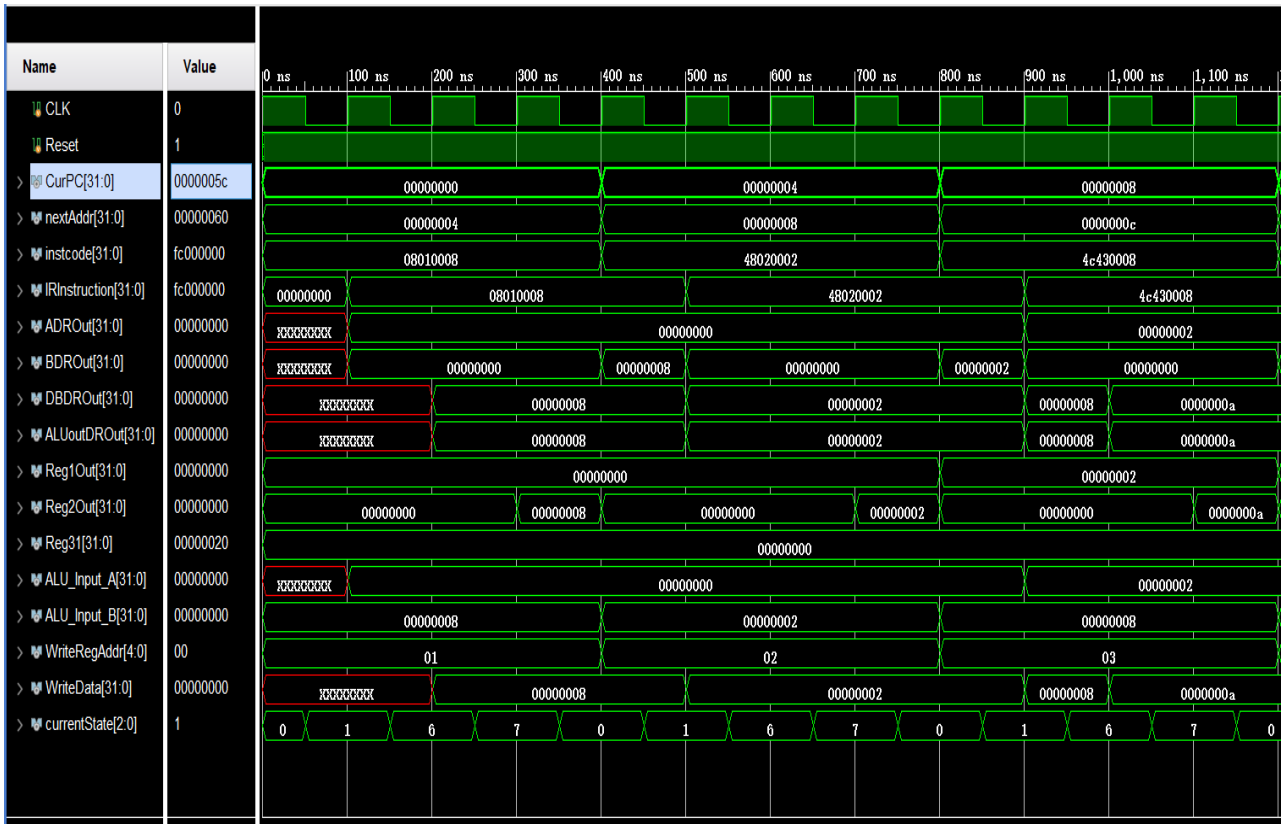
sim.v

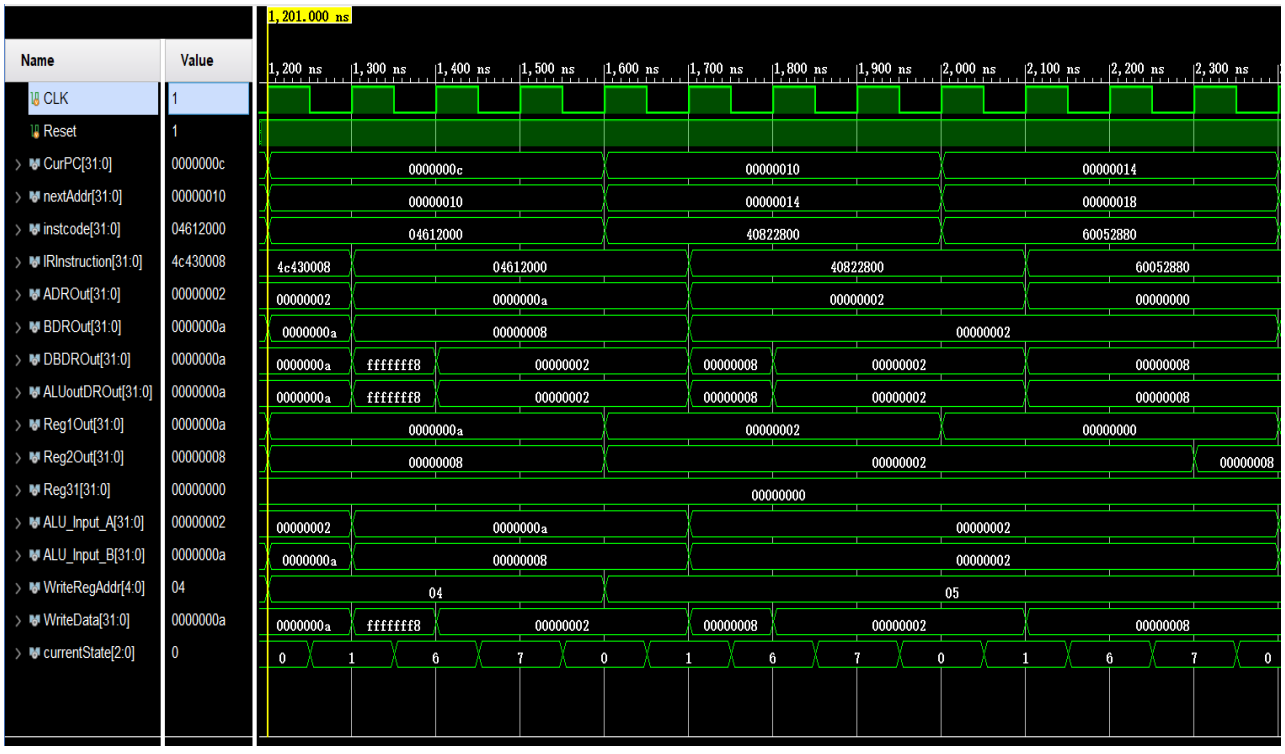
仿真模块，给一个时钟信号给 mCPU

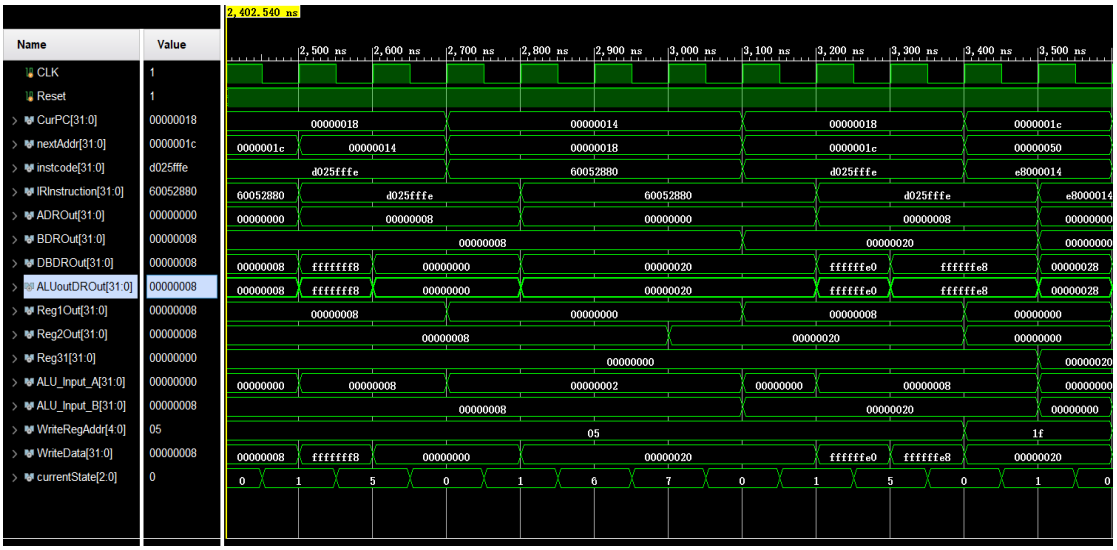
```
1.  `timescale 1ns / 1ps
2.
3.  module sim;
4.      reg CLK;
5.      reg Reset;
6.      wire[31:0] CurPC, nextAddr, instcode, IRInstruction, ADROut, BDROut, DBD
    ROut, ALUoutDROut, WriteData, Reg1Out, Reg2Out, Reg31, ALU_Input_A, ALU_Inpu
    t_B;
7.      wire[4:0] WriteRegAddr;
8.      wire[2:0] currentState;
9.      mCPU multipleCPU(CLK, Reset, CurPC, nextAddr, instcode, IRInstruction, A
    DROut, BDROut, DBDROut, ALUoutDROut, Reg1Out, Reg2Out, Reg31, ALU_Input_A, A
    LU_Input_B, WriteData, WriteRegAddr, currentState);
```

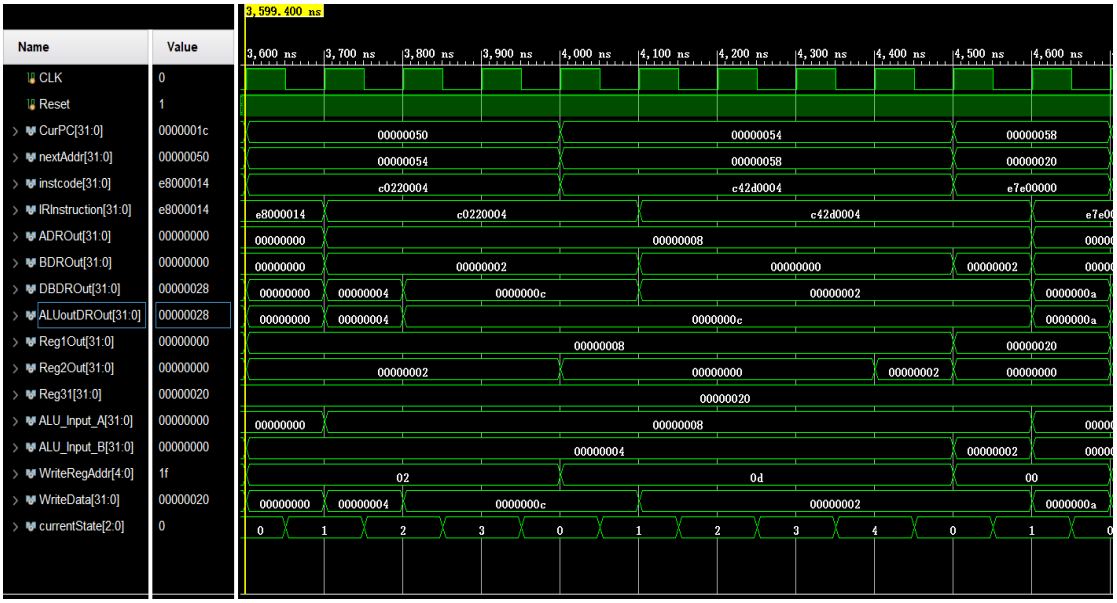
```
10.     initial begin
11.         CLK = 1;
12.         Reset = 0;
13.         #1;
14.         Reset = 1;
15.         forever #50 CLK = !CLK;
16.     end
17. endmodule
```

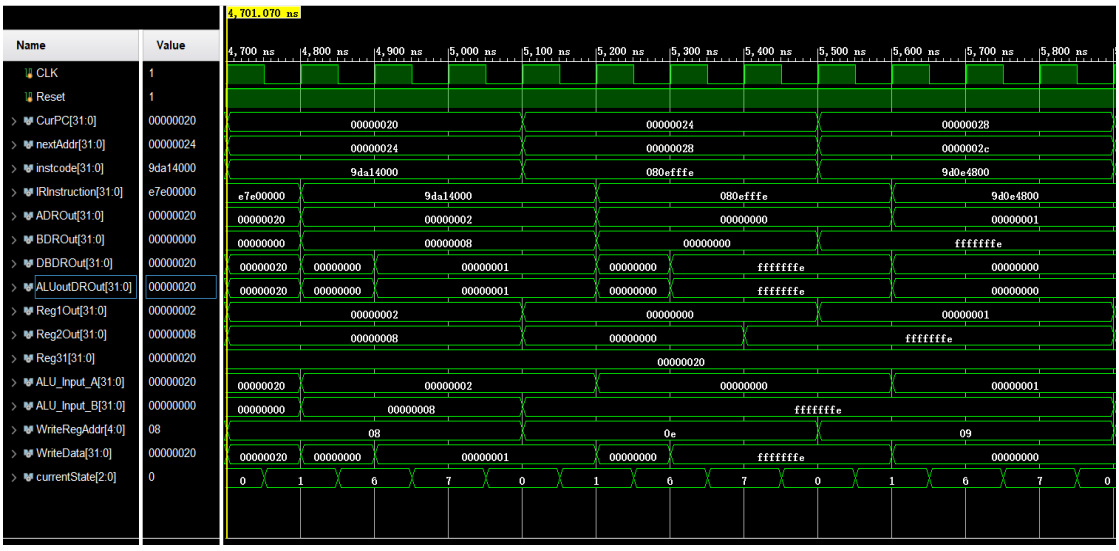
仿真波形如下图所示：

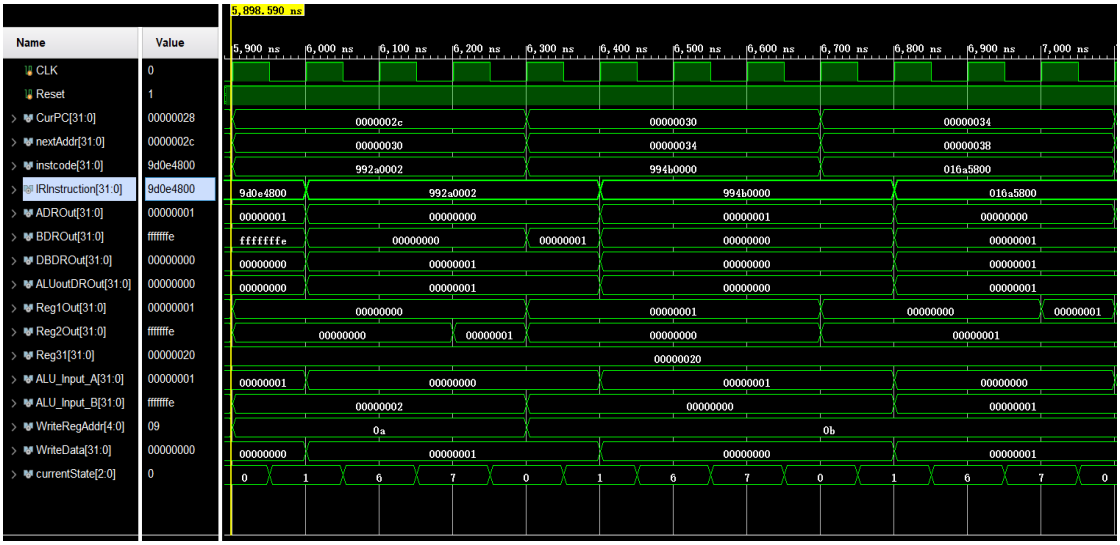


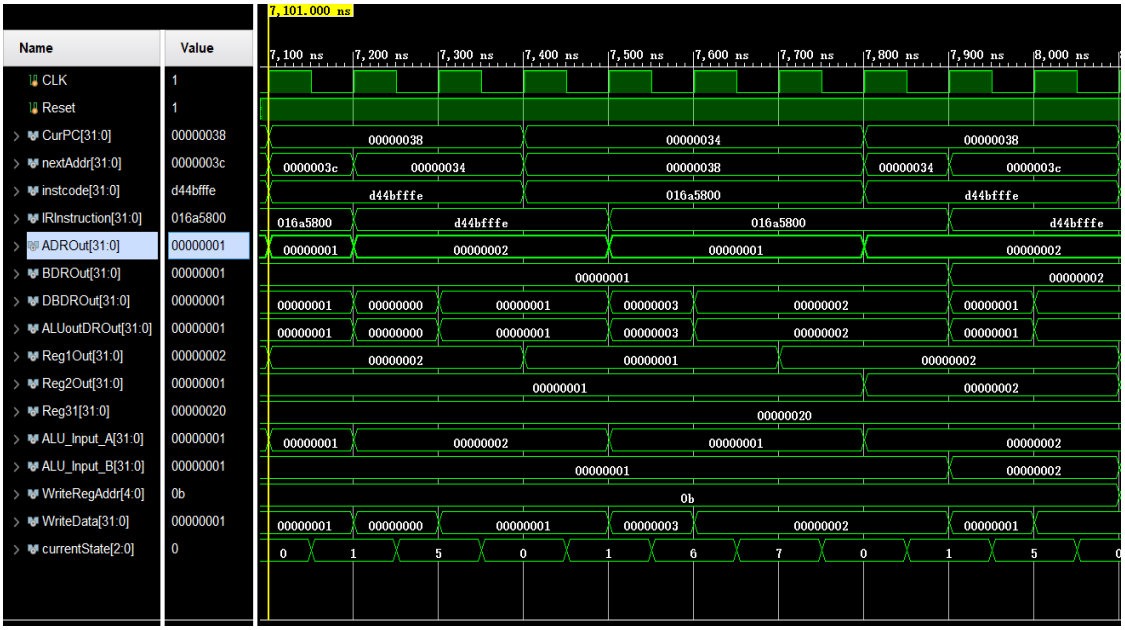


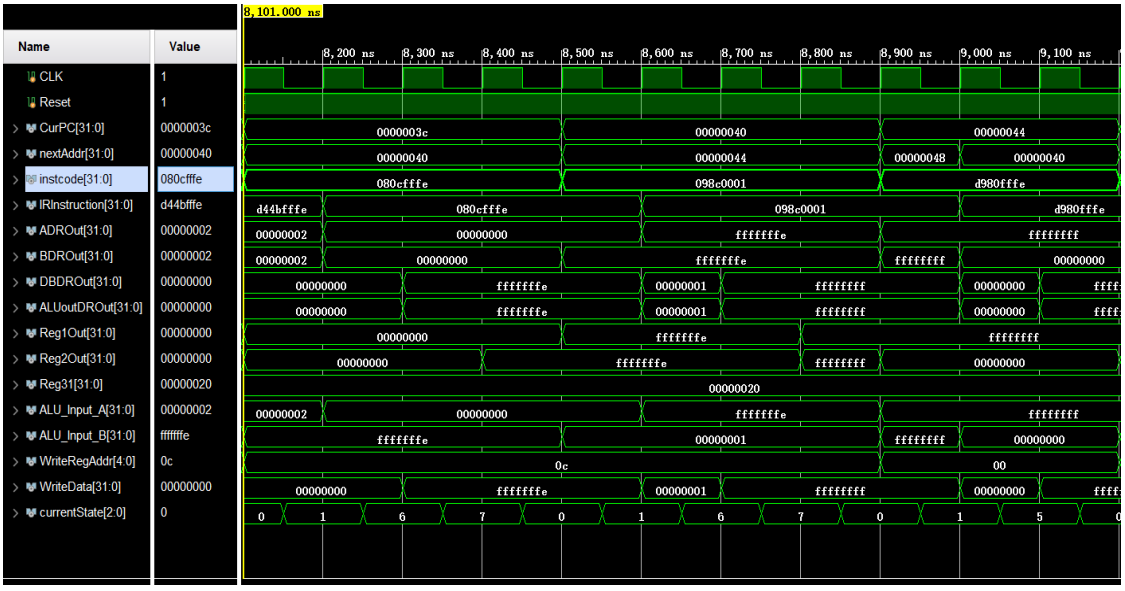

2,402.540 ns
Name
Value
CLK 1
Reset 1
CurPC[31:0] 00000018
nextAddr[31:0] 0000001c
instcode[31:0] d025fffe
IRInstruction[31:0] 60052880
ADROut[31:0] 00000000
BDROut[31:0] 00000008
DBDROut[31:0] 00000008
ALUoutDROut[31:0] 00000008
Reg1Out[31:0] 00000008
Reg2Out[31:0] 00000008
Reg31[31:0] 00000000
ALU_Input_A[31:0] 00000000
ALU_Input_B[31:0] 00000008
WriteRegAddr[4:0] 05
WriteData[31:0] 00000008
currentState[2:0] 0

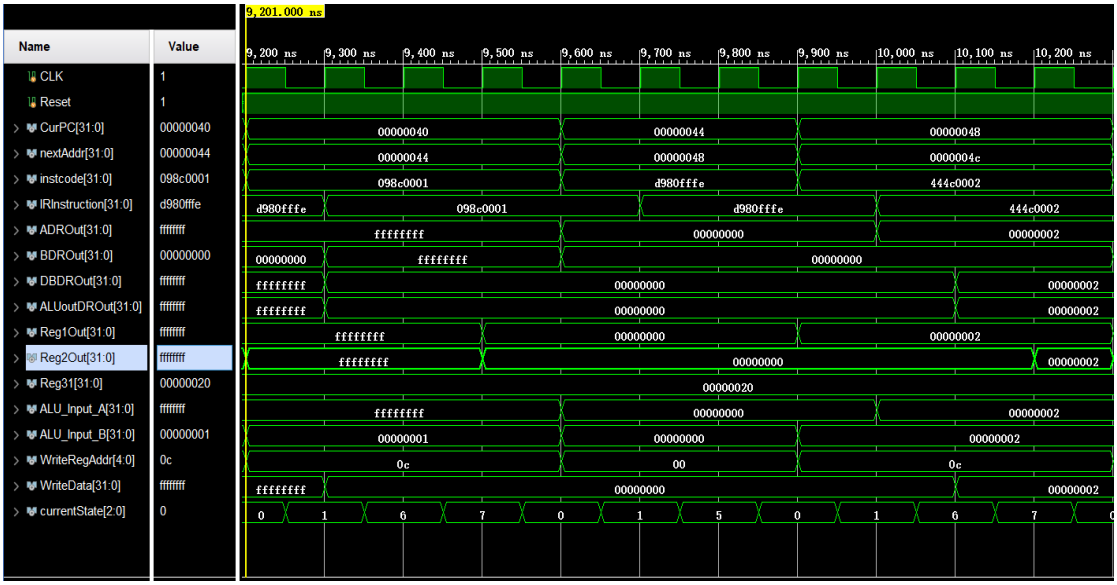

3,599.400 ns
Name
Value
CLK 0
Reset 1
CurPC[31:0] 0000001c
nextAddr[31:0] 00000050
instcode[31:0] e8000014
IRInstruction[31:0] e8000014
ADROut[31:0] 00000000
BDROut[31:0] 00000000
DBDROut[31:0] 00000028
ALUoutDROut[31:0] 00000028
Reg1Out[31:0] 00000000
Reg2Out[31:0] 00000000
Reg31[31:0] 00000020
ALU_Input_A[31:0] 00000000
ALU_Input_B[31:0] 00000000
WriteRegAddr[4:0] 1f
WriteData[31:0] 00000020
currentState[2:0] 0

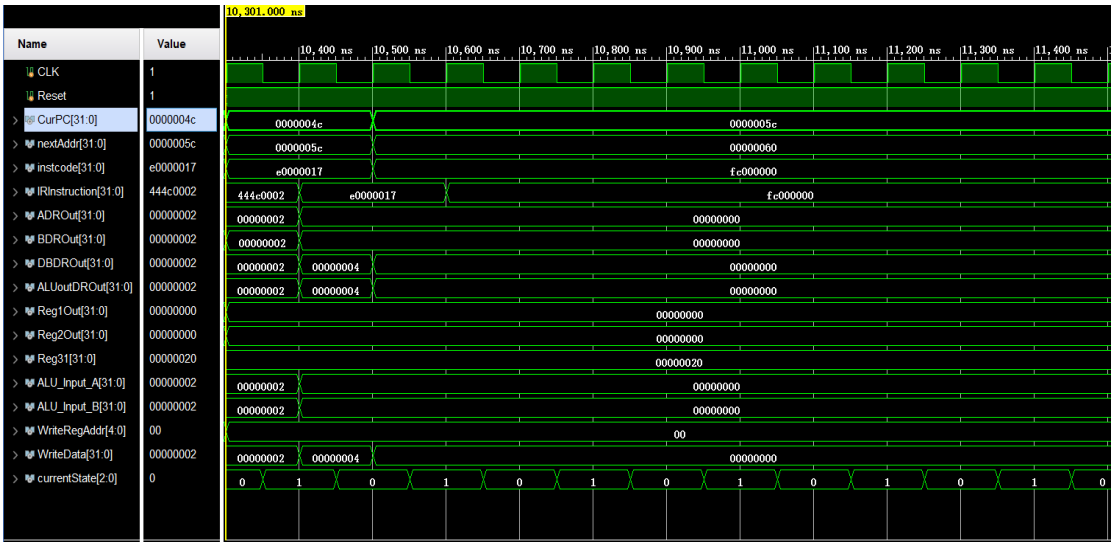

4,701.070 ns
Name
Value
CLK 1
Reset 1
CurPC[31:0] 00000020
nextAddr[31:0] 00000024
instcode[31:0] 9da14000
IRInstruction[31:0] e7e00000
ADROut[31:0] 00000020
BDROut[31:0] 00000000
DBDROut[31:0] 00000020
ALUoutDROut[31:0] 00000020
Reg1Out[31:0] 00000002
Reg2Out[31:0] 00000008
Reg31[31:0] 00000020
ALU_Input_A[31:0] 00000020
ALU_Input_B[31:0] 00000000
WriteRegAddr[4:0] 08
WriteData[31:0] 00000020
currentState[2:0] 0


5,898.590 ns
Name
Value
CLK 0
Reset 1
CurPC[31:0] 00000028
nextAddr[31:0] 0000002c
instcode[31:0] 9d0e4800
IRInstruction[31:0] 9d0e4800
ADROut[31:0] 00000001
BDROut[31:0] fffffffe
DBDROut[31:0] 00000000
ALUoutDROut[31:0] 00000000
Reg1Out[31:0] 00000001
Reg2Out[31:0] fffffffe
Reg31[31:0] 00000020
ALU_Input_A[31:0] 00000001
ALU_Input_B[31:0] fffffffe
WriteRegAddr[4:0] 09
WriteData[31:0] 00000000
currentState[2:0] 0

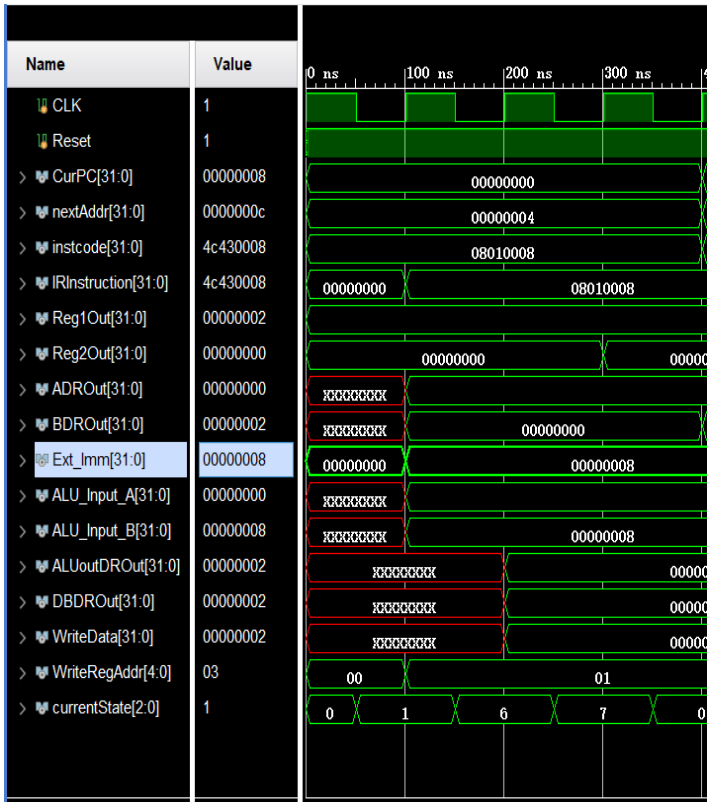

7,101.000 ns
Name
Value
CLK 1
Reset 1
CurPC[31:0] 00000038
nextAddr[31:0] 0000003c
instcode[31:0] d44bfffe
IRInstruction[31:0] 016a5800
ADROut[31:0] 00000001
BDROut[31:0] 00000001
DBDROut[31:0] 00000001
ALUoutDROut[31:0] 00000001
Reg1Out[31:0] 00000002
Reg2Out[31:0] 00000001
Reg31[31:0] 00000020
ALU_Input_A[31:0] 00000001
ALU_Input_B[31:0] 00000001
WriteRegAddr[4:0] 0b
WriteData[31:0] 00000001
currentState[2:0] 0


8,101.000 ns
Name
Value
CLK 1
Reset 1
CurPC[31:0] 0000003c
nextAddr[31:0] 00000040
instcode[31:0] 080cfffe
IRInstruction[31:0] d44bfffe
ADROut[31:0] 00000002
BDROut[31:0] 00000002
DBDROut[31:0] 00000000
ALUoutDROut[31:0] 00000000
Reg1Out[31:0] 00000000
Reg2Out[31:0] 00000000
Reg31[31:0] 00000020
ALU_Input_A[31:0] 00000002
ALU_Input_B[31:0] fffffffe
WriteRegAddr[4:0] 0c
WriteData[31:0] 00000000
currentState[2:0] 0


9,201.000 ns
Name
Value
CLK 1
Reset 1
CurPC[31:0] 00000040
nextAddr[31:0] 00000044
instcode[31:0] 098c0001
IRInstruction[31:0] d980fffe
ADROut[31:0] ffffffff
BDROut[31:0] 00000000
DBDROut[31:0] ffffffff
ALUoutDROut[31:0] ffffffff
Reg1Out[31:0] ffffffff
Reg2Out[31:0] ffffffff
Reg31[31:0] 00000020
ALU_Input_A[31:0] ffffffff
ALU_Input_B[31:0] 00000001
WriteRegAddr[4:0] 0c
WriteData[31:0] ffffffff
currentState[2:0] 0


10,301.000 ns
Name
Value
CLK 1
Reset 1
CurPC[31:0] 0000004c
nextAddr[31:0] 0000005c
instcode[31:0] e0000017
IRInstruction[31:0] 444c0002
ADROut[31:0] 00000002
BDROut[31:0] 00000002
DBDROut[31:0] 00000002
ALUoutDROut[31:0] 00000002
Reg1Out[31:0] 00000000
Reg2Out[31:0] 00000000
Reg31[31:0] 00000020
ALU_Input_A[31:0] 00000002
ALU_Input_B[31:0] 00000002
WriteRegAddr[4:0] 00
WriteData[31:0] 00000002
currentState[2:0] 0

6. 对各个指令执行过程详解



addiu $1,$0,8

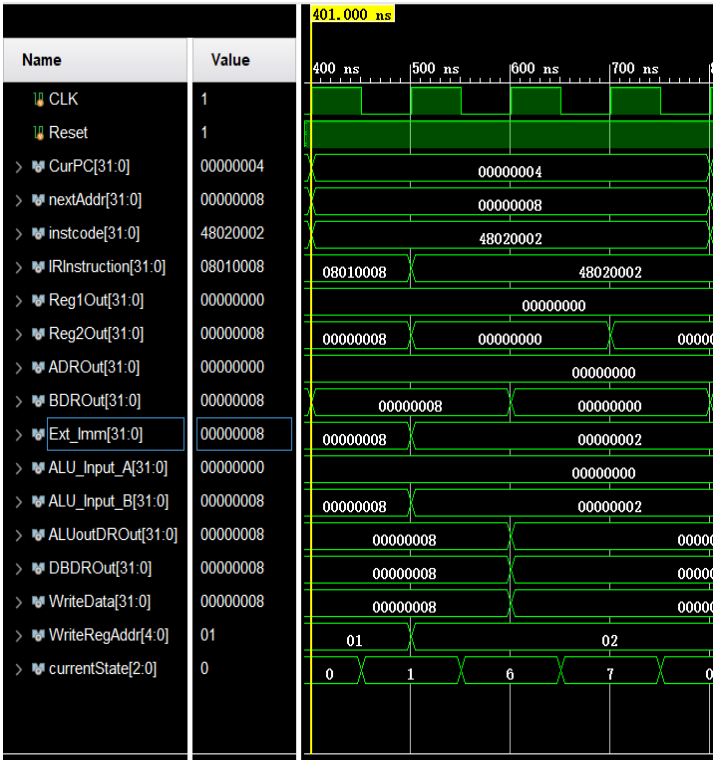
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 00；
下条指令地址为 04；

001(ID):Rs 寄存器为 00，值为 00(ADROut)；
Imm 值为 08；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 08(ALUoutDROut)

111(WB)：写回寄存器 rt 为 01，值为 08(WriteData)；



ori $2,$0,2

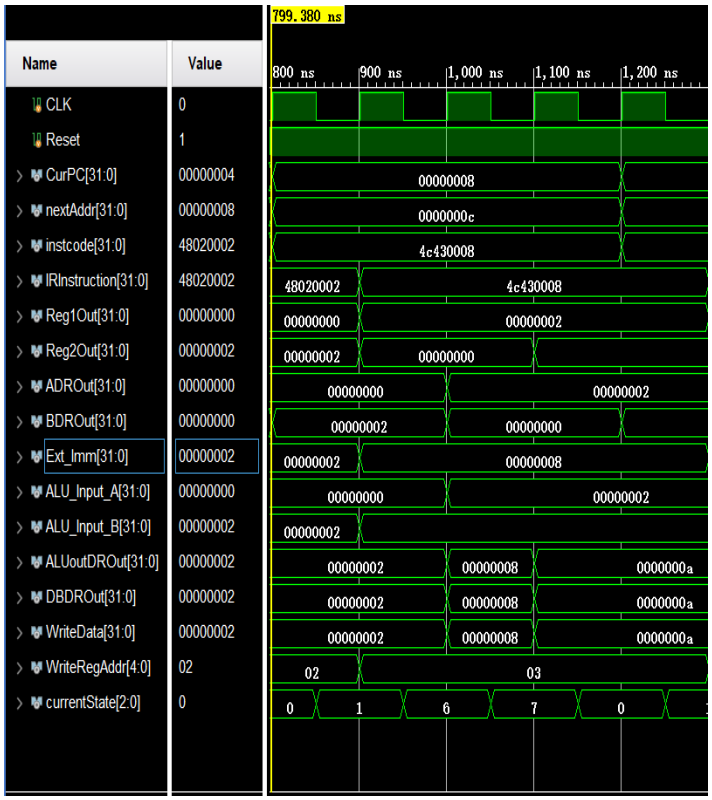
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 04；
下条指令地址为 08；

001(ID):Rs 寄存器为 00，值为 00(ADROut)；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 02(ALUoutDROut)

111(WB)：写回寄存器 rt 为 02，值为 02(WriteData)；



xori $3,$2,8

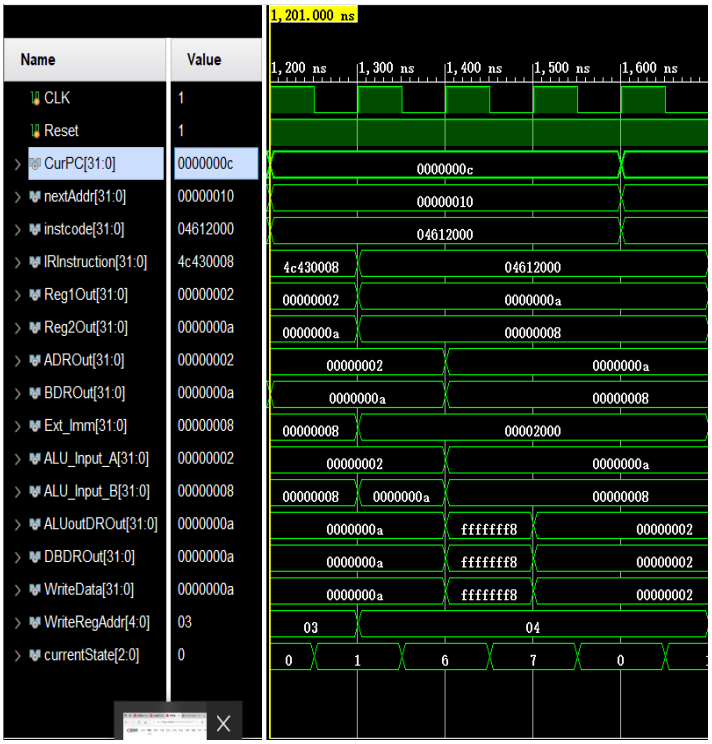
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 08；
下条指令地址为 0c；

001(ID):Rs 寄存器为 02，值为 02(ADROut)；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 0a(ALUoutDROut)

111(WB)：写回寄存器 rt 为 03，值为 0a(WriteData)；



sub $4,$3,$1

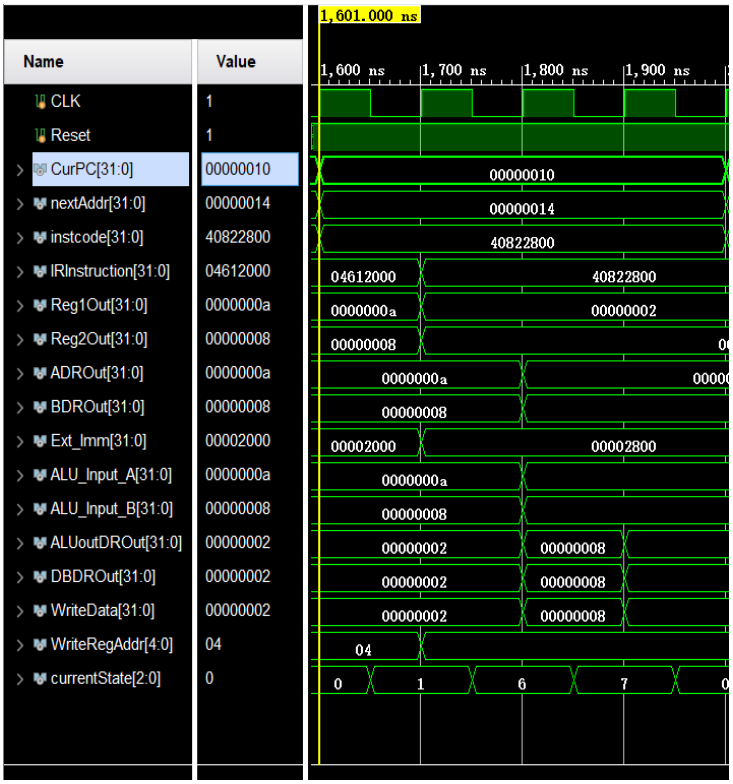
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 0c；
下条指令地址为 10；

001(ID):Rs 寄存器为 03，值为 0a(ADROut)；
Rt 寄存器为 01，值为 08(BDROut);

110(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 02(ALUoutDROut)

111(WB)：写回寄存器 rd 为 04，值为 02(WriteData)；



and $5,$4,$2

该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 10；
下条指令地址为 14；

001(ID):Rs 寄存器为 04，值为 02(ADROut)；
Rt 寄存器为 02，值为 02(BDROut);

110(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 02(ALUoutDROut)

111(WB)：写回寄存器 rd 为 05，值为 02(WriteData)；

sll $5,$5,2

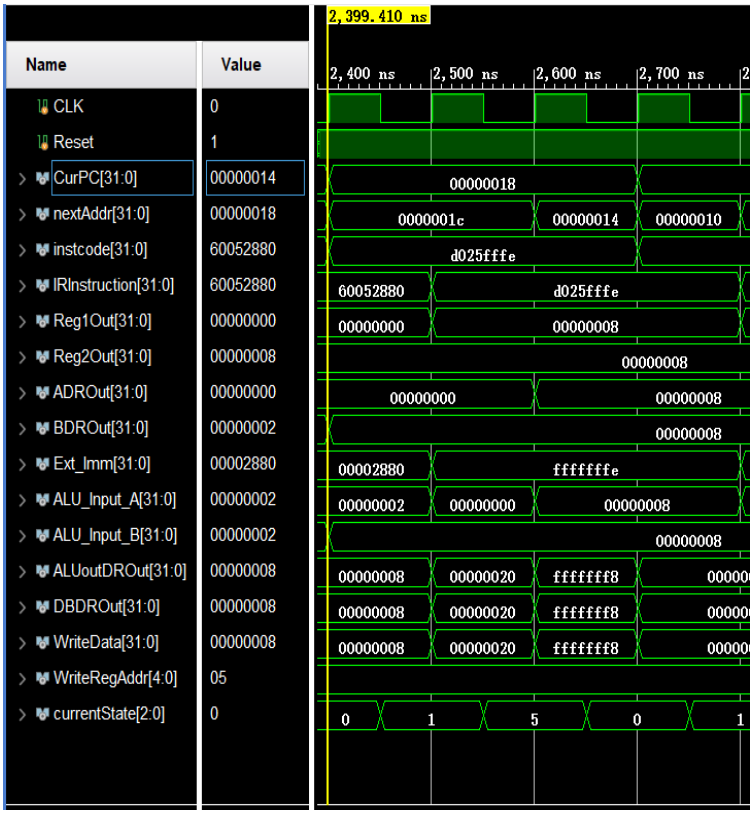
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 14；
下条指令地址为 18；

001(ID):Rt 寄存器为 05，值为 02(ADROut)；
sa 值为 02(BDROut);

110(EXE)：ALU_A、ALU_B 分别来自 rt, ra; 结果为 08(ALUoutDROut)

111(WB)：写回寄存器 rd 为 05，值为 08(WriteData)；



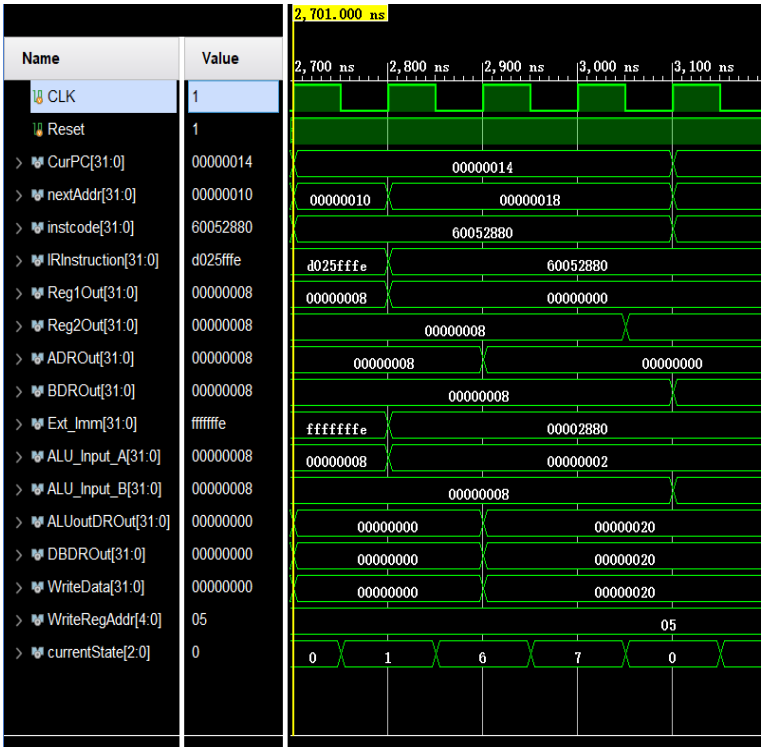
beq $5,$1,-2(=,转 14)

该指令执行 3 个时钟周期，分别对应状态转换图中的 000->001->101 三个状态；

000(IF):当前 PC 值为 18；

001(ID):Rs 寄存器为 05，值为 08(ADROut)；
Rt 寄存器为 01，值为 08(BDROut);
Imm 值为-2；

101(EXE)：ALU_A、ALU_B 分别来自 rs, rt;
结果为-08(ALUoutDROut)
所以下条指令地址为 14(1c-08=14)



sll $5,$5,2

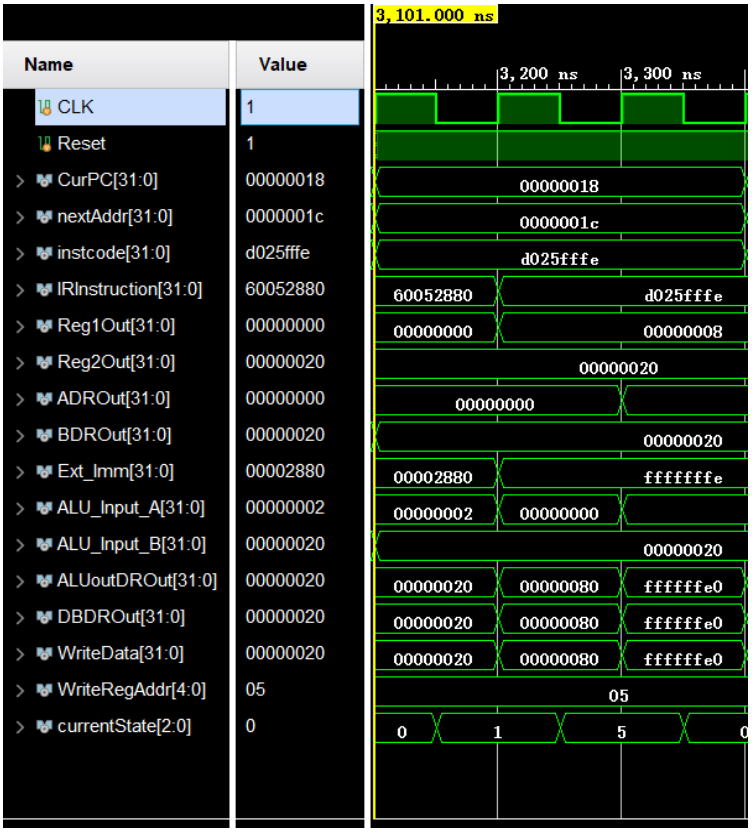
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 14；
下条指令地址为 18；

001(ID):Rt 寄存器为 05，值为 08(ADROut)；
sa 值为 02(BDROut);

110(EXE)：ALU_A、ALU_B 分别来自 rt, ra;
结果为 20(ALUoutDROut)

111(WB)：写回寄存器 rd 为 05，值为 20(WriteData)；



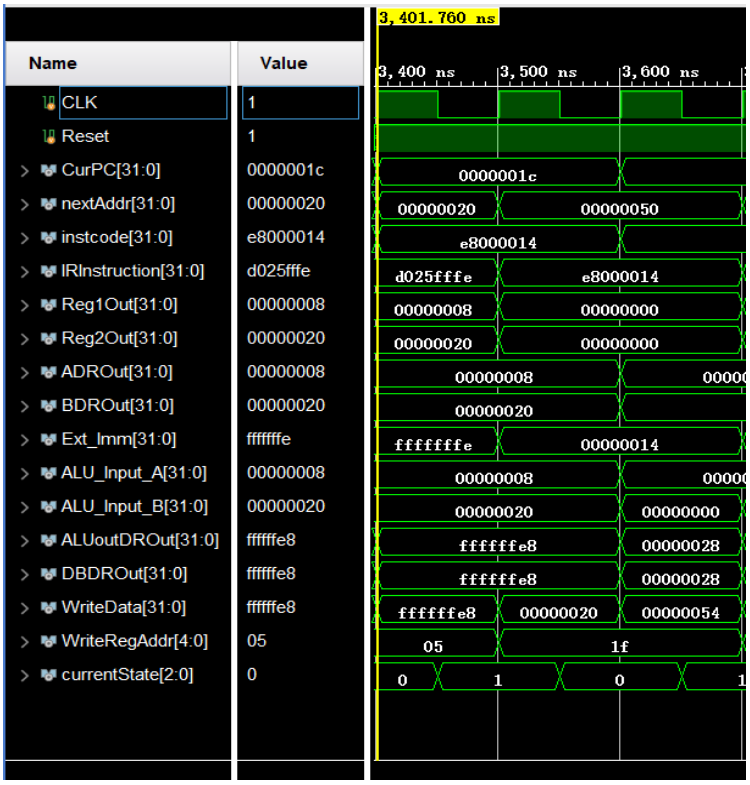
beq $5,$1,-2(=,转 14)

该指令执行 3 个时钟周期，分别对应状态转换图中的 000->001->101 三个状态；

000(IF):当前 PC 值为 18；
下条指令地址为 1c

001(ID):Rs 寄存器为 05，值为 20(ADROut)；
Rt 寄存器为 01，值为 08(BDROut);

101(EXE)：ALU_A、ALU_B 分别来自 rs, rt;

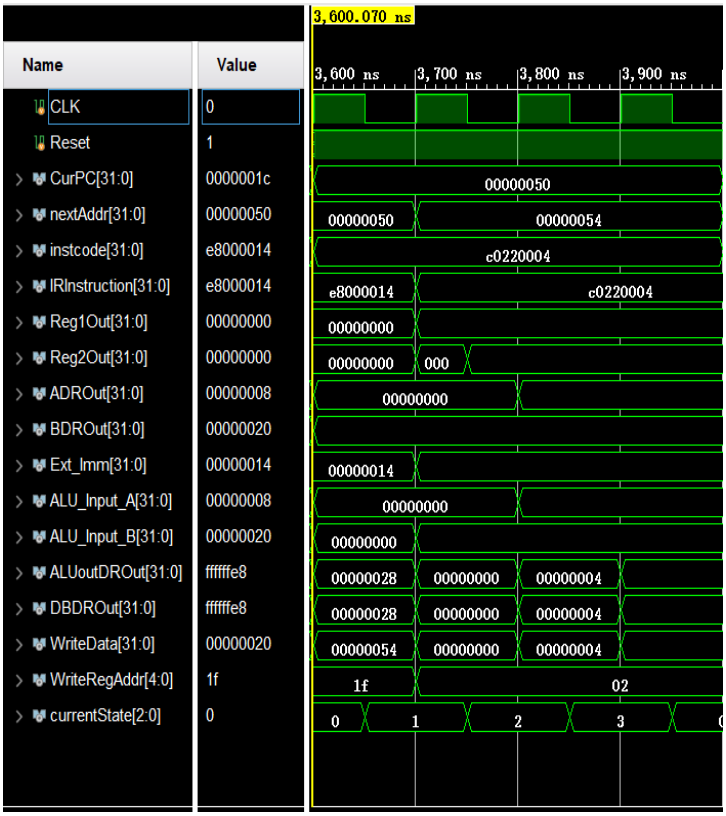


jal 0x0000050

该指令执行 2 个时钟周期，分别对应状态转换图中的 000->001 两个状态；

000(IF):当前 PC 值为 18；

001(ID):下条指令地址为 1c



sw $2,4($1)

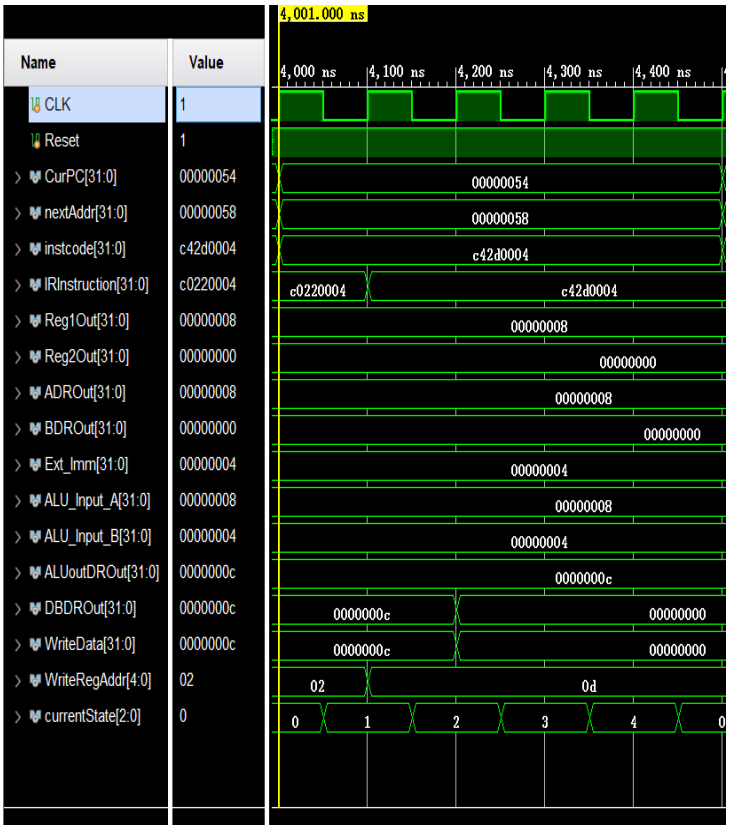
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->010->011 四个状态；

000(IF):当前 PC 值为 50；
下条指令地址为 54；

001(ID):Rs 寄存器为$1，值为 08(ADROut)；imm 值为 04；

010(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 0c(ALUoutDROut)

011(MEM)：将寄存器 rt $2 的值 2 存进 DAddr(ALUoutDROut) = 0c 里。



lw $13,4($1)

该指令执行 5 个时钟周期，分别对应状态转换图中的 000->001->010->011->100 五个状态；

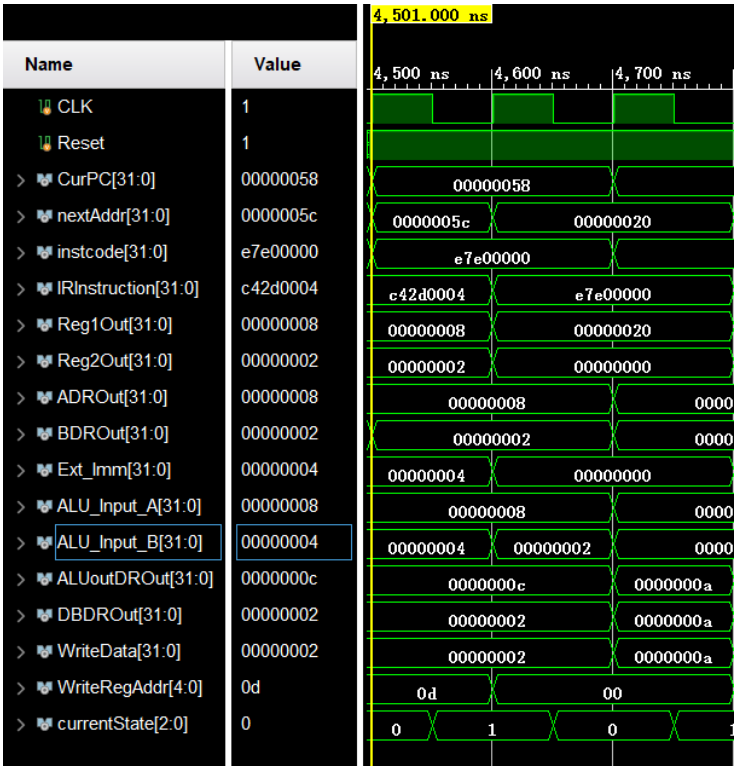
000(IF):当前 PC 值为 54；
下条指令地址为 58；

001(ID):Rs 寄存器为 01，值为 08(ADROut)；imm 值为 04；

010(EXE):ALU_A、ALU_B 分别来自 rt, imm; 结果为 0c(ALUoutDROut)

011(MEM)：读取 DataMem 值为 00(DBDROut)；

100(WB)：写回寄存器 rt 为 0d, 值为 00(WriteData);



jr $31

该指令执行 2 个时钟周期，分别对应状态转换图中的 000->001 两个状态；

000(IF):当前 PC 值为 50；

001(ID):Rs 寄存器为 $31，值为 20(ADROut)；
下条指令为 20

slt $8,$13,$1

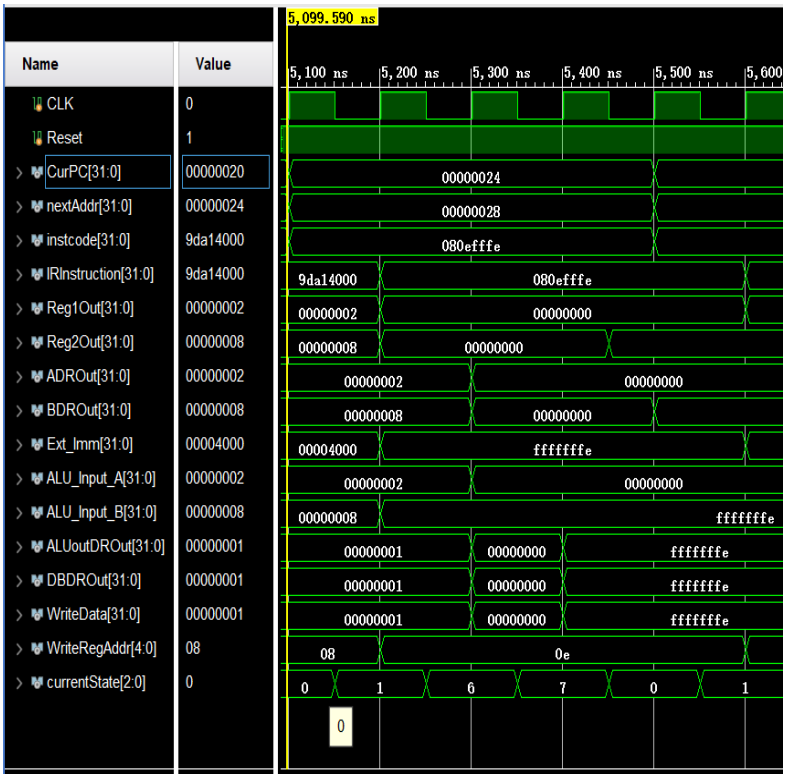
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 20；
下条指令地址为 24；

001(ID):Rs 寄存器为 13，值为 02(ADROut)；
Rt 寄存器为 01，值为 08(BDROut);

110(EXE)：ALU_A、ALU_B 分别来自 rs, rt;
结果为 01(ALUoutDROut)

111(WB)：写回寄存器 rd 为 08，值为 01(WriteData)；



addiu $14,$0,-2

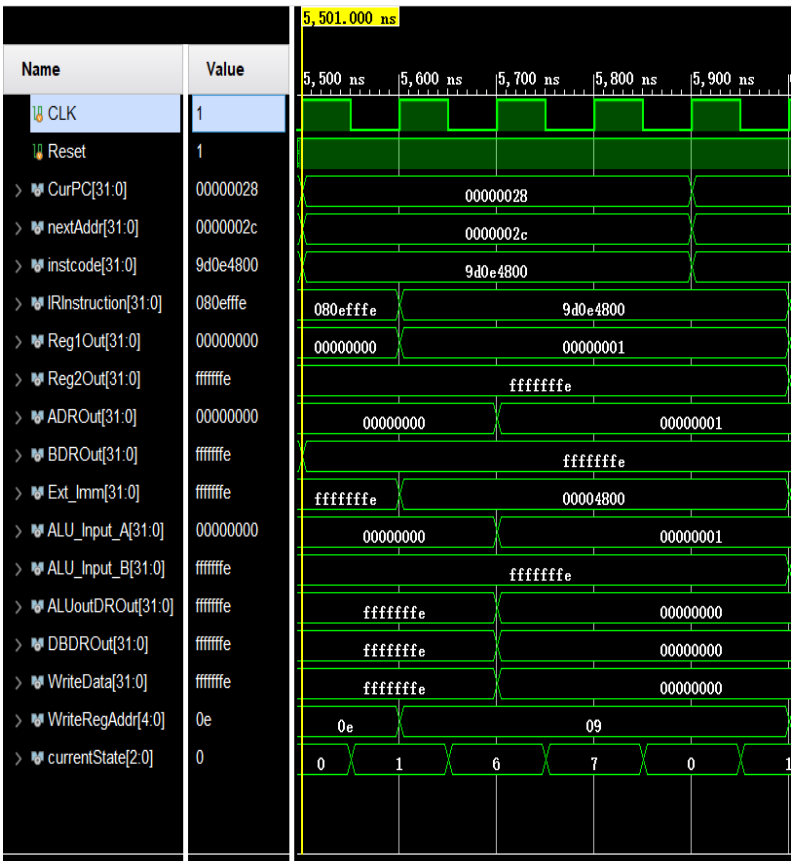
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 24；
下条指令地址为 28；

001(ID):Rs 寄存器为 00，值为 00(ADROut)；
Imm 值为-2；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为-2(ALUoutDROut)

111(WB)：写回寄存器 rt 为 0e，值为 -2(WriteData)；



slt $9,$8,$14

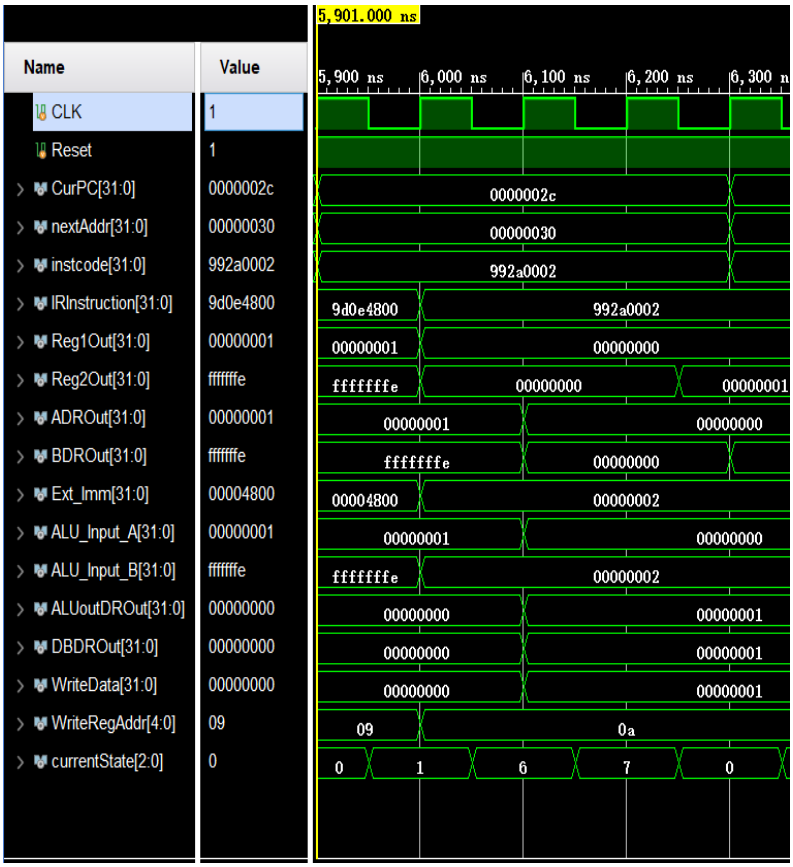
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 28；
下条指令地址为 2c；

001(ID):Rs 寄存器为 08，值为 01(ADROut)；
Rt 寄存器为 14，值为-2(BDROut);

110(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 00(ALUoutDROut)

111(WB)：写回寄存器 rd 为 09，值为 00(WriteData)；



slti $10,$9,2

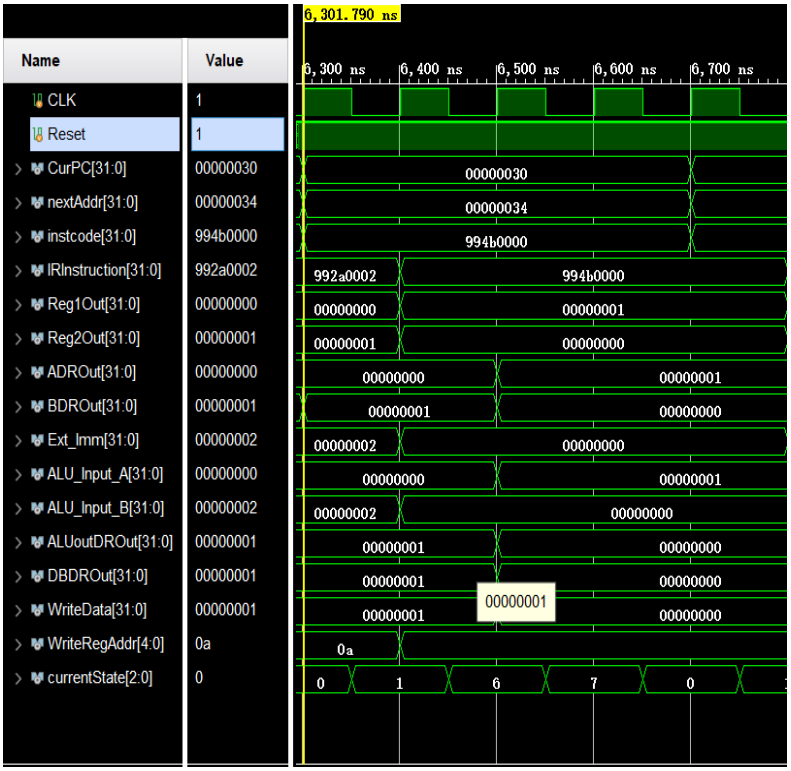
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 2c；
下条指令地址为 30；

001(ID):Rs 寄存器为 09，值为 00(ADROut)；
imm 值为 02(Ext_Imm);

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 01(ALUoutDROut)

111(WB)：写回寄存器 rt 为 10，值为 01(WriteData)；



slti $11,$10,0

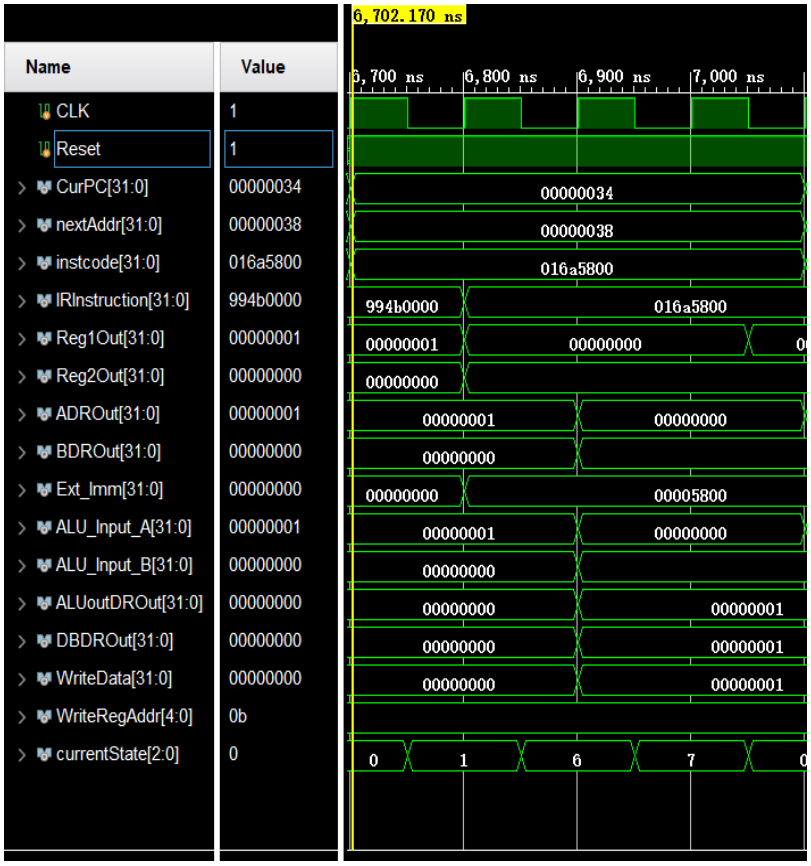
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 30；
下条指令地址为 34；

001(ID):Rs 寄存器为 10，值为 01(ADROut)；
imm 值为 00(Ext_Imm);

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 00(ALUoutDROut)

111(WB)：写回寄存器 rt 为 11，值为 00(WriteData)；



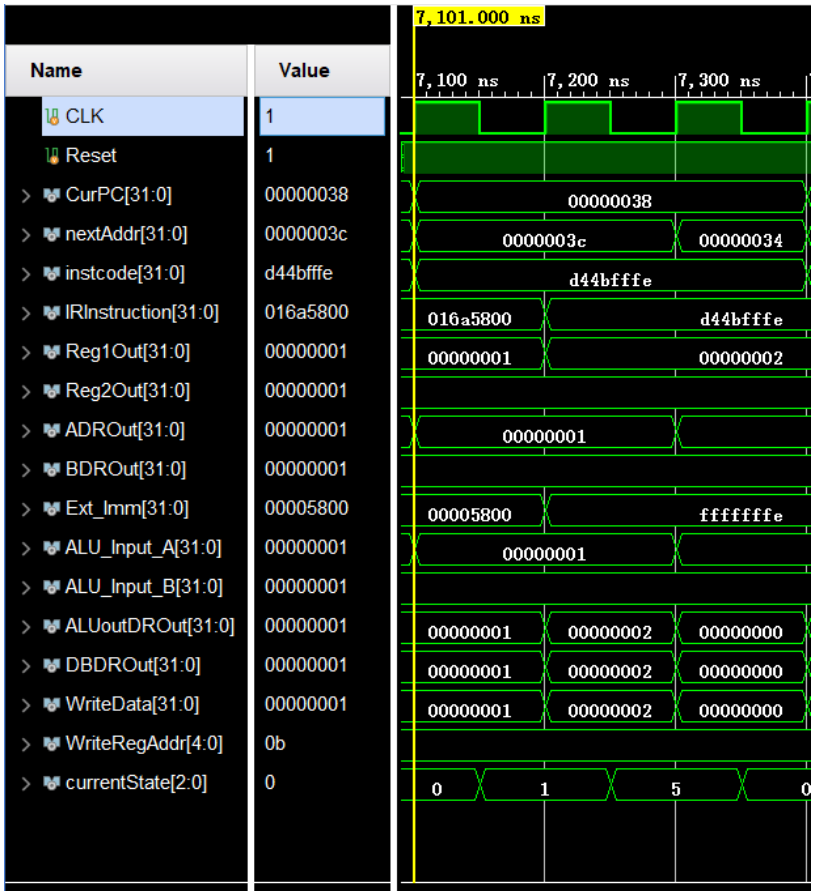
add $11,**$11**,$10
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 34；
下条指令地址为 38；

001(ID):Rs 寄存器为 11，值为 00(ADROut)；
Rt 寄存器为 10，值为 01(BDROut);

110(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 01(ALUoutDROut)

111(WB)：写回寄存器 rd 为 11，值为 01(WriteData)；



bne $11,$2,-2 (≠,转 34)
该指令执行 3 个时钟周期，分别对应状态转换图中的 000->001->101 三个状态；

000(IF):当前 PC 值为 38；

001(ID):Rs 寄存器为 11，值为 01(ADROut)；
Rt 寄存器为 02，值为 01(BDROut);
Imm 值为-2；

101(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 00(ALUoutDROut)
下条指令地址为 34

add $11,$11,$10

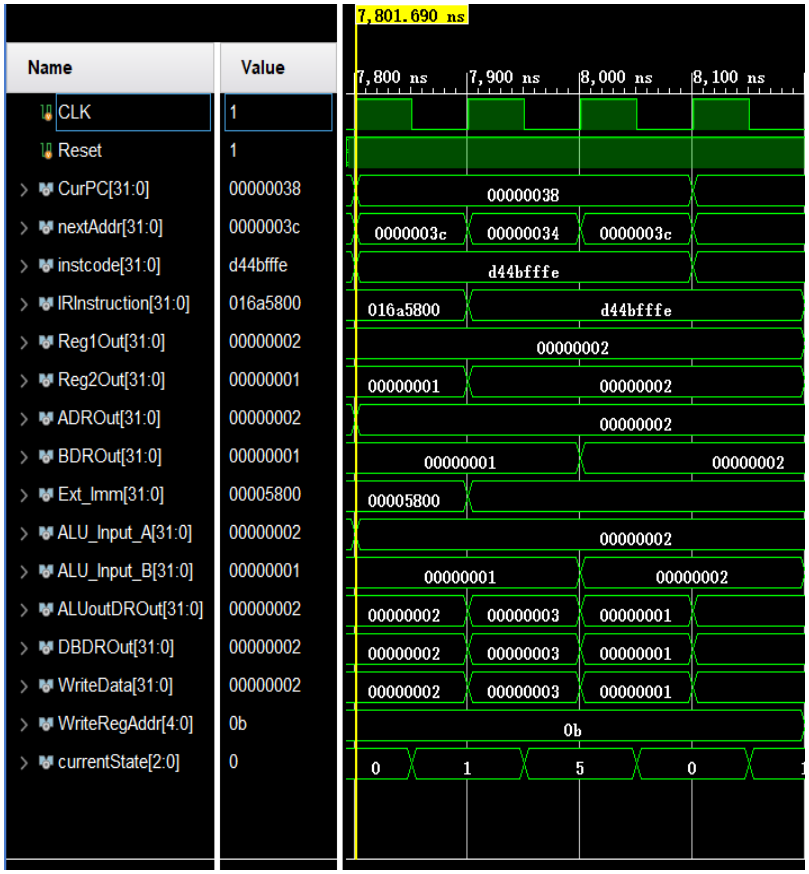
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 34；

001(ID):Rs 寄存器为 11，值为 01(ADROut)；
Rt 寄存器为 10，值为 01(BDROut);
下条指令地址为 38；

110(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 02(ALUoutDROut)

111(WB)：写回寄存器 rd 为 11，值为 02(WriteData)；



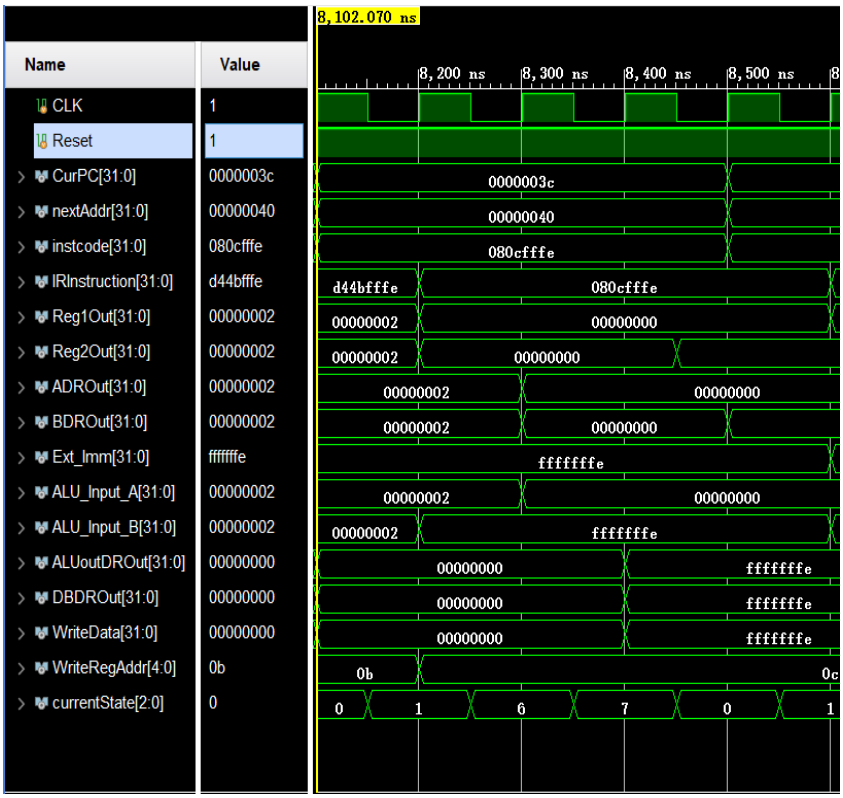
bne $11,$2,-2 (≠,转 34)

该指令执行 3 个时钟周期，分别对应状态转换图中的 000->001->101 三个状态；

000(IF):当前 PC 值为 38；

001(ID):Rs 寄存器为 11，值为 02(ADROut)；
Rt 寄存器为 02，值为 01(BDROut);
Imm 值为-2；

101(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 01(ALUoutDROut)
下条指令地址为 3c



addiu $12,$0,-2

该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 3c；
下条指令地址为 40；

001(ID):Rs 寄存器为 00，值为 00(ADROut)；
Imm 值为-2；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 -2(ALUoutDROut)

111(WB)：写回寄存器 rt 为 0c，值为 -2(WriteData)

addiu $12,$12,1

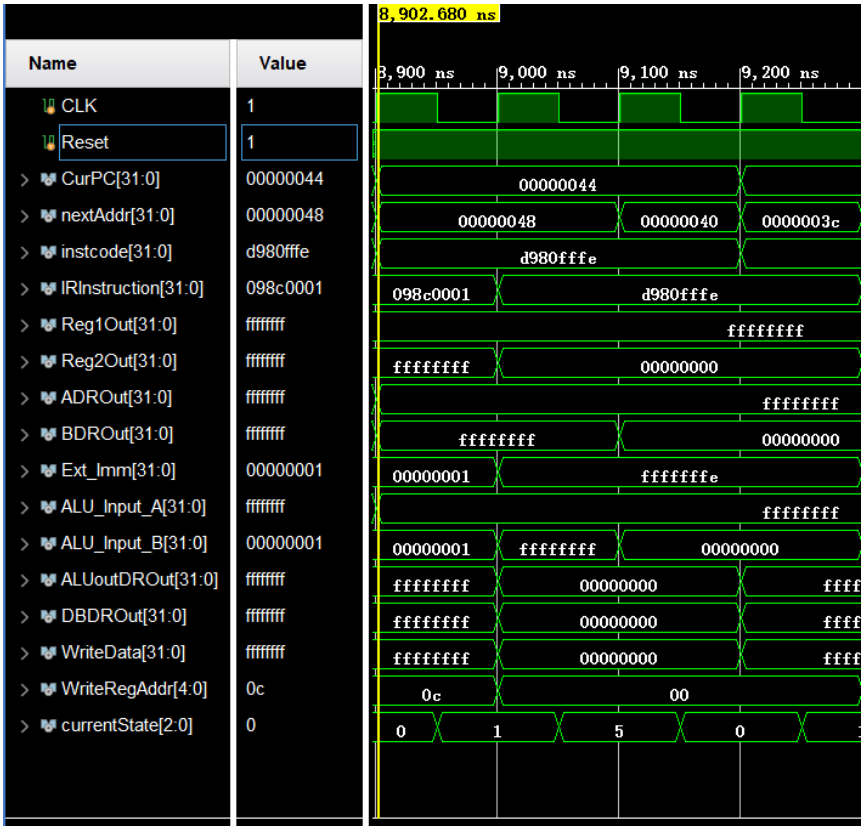
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 40；
下条指令地址为 44；

001(ID):Rs 寄存器为 12，值为 -2(ADROut)；
Imm 值为 1；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 -1(ALUoutDROut)

111(WB)：写回寄存器 rt 为 0c，值为-1(WriteData)



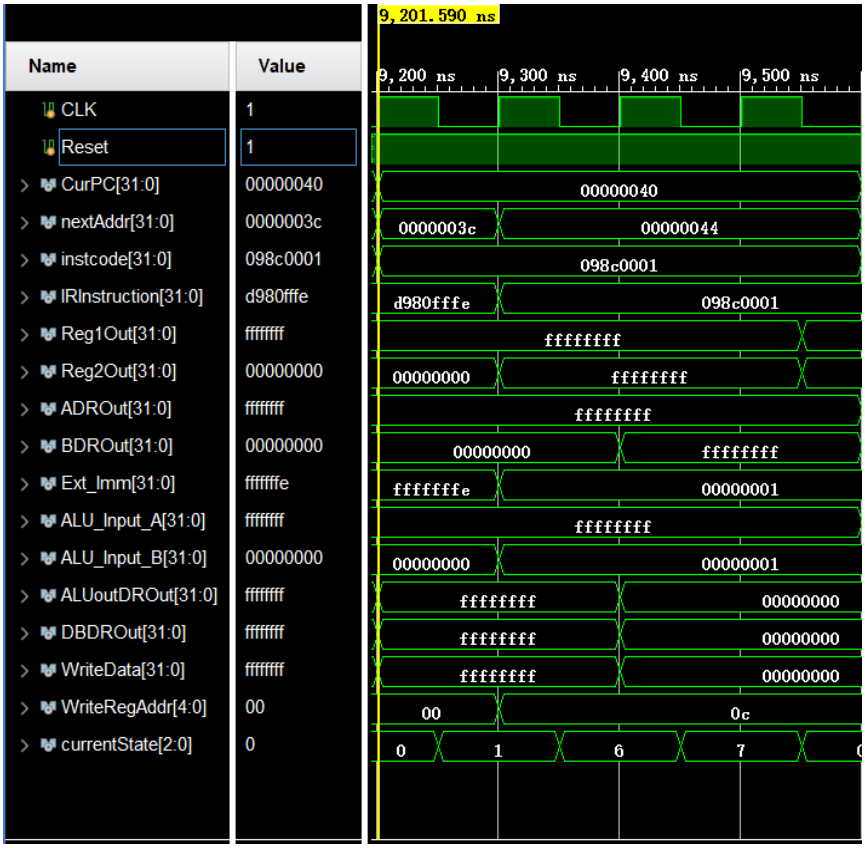
bltz $12,-2 (<0,转 40)

该指令执行 3 个时钟周期，分别对应状态转换图中的 000->001->101 三个状态；

000(IF):当前 PC 值为 44；

001(ID):Rs 寄存器为 12，值为 -1(ADROut)；
Rt 寄存器为 00，值为 00(BDROut);
Imm 值为-2；

101(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 00(ALUoutDROut)
下条指令地址为 40



addiu $12,$12,1

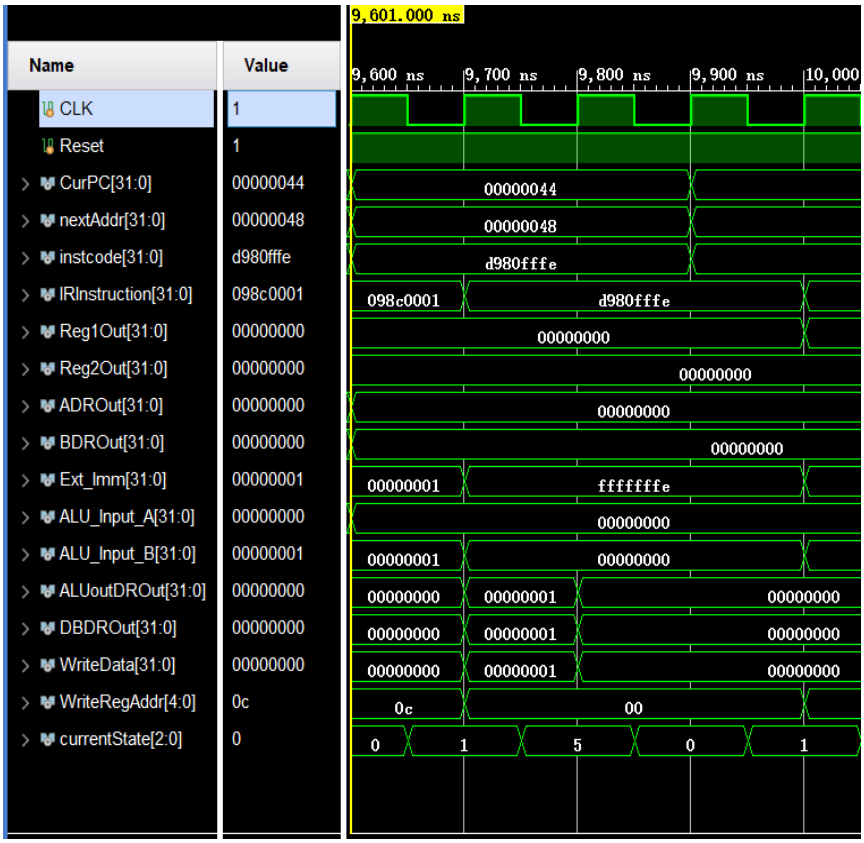
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 40；
下条指令地址为 44；

001(ID):Rs 寄存器为 12，值为 -1(ADROut)；
Imm 值为 1；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 00(ALUoutDROut)

111(WB)：写回寄存器 rt 为 0c，值为 00(WriteData)



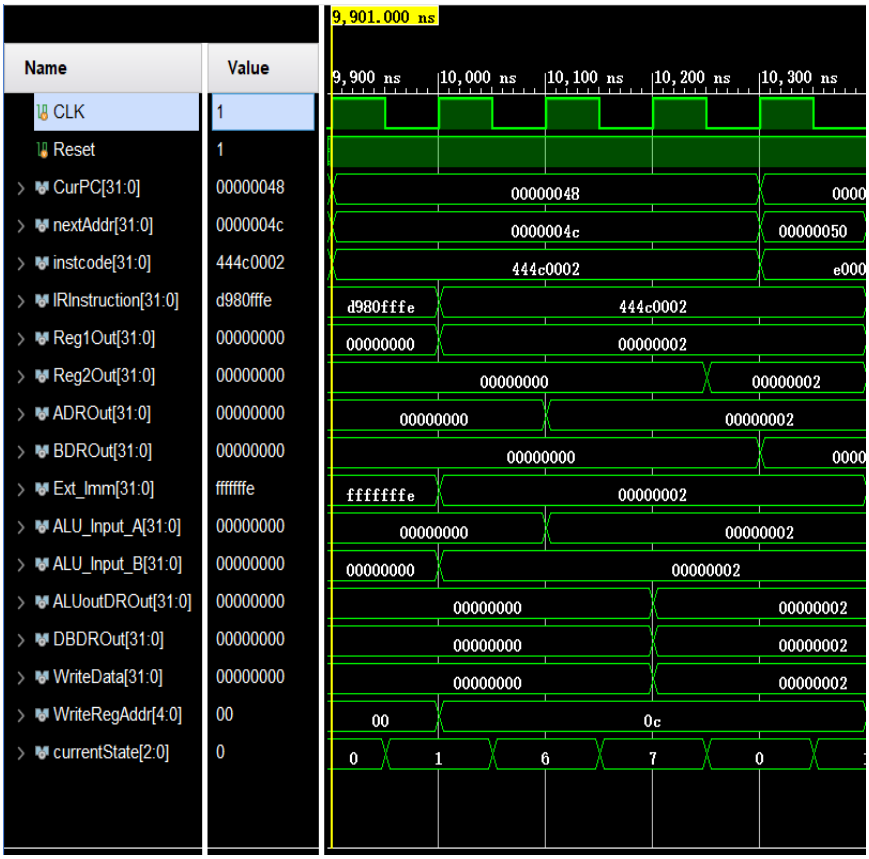
bltz $12,-2 (<0,转 40)

该指令执行 3 个时钟周期，分别对应状态转换图中的 000->001->101 三个状态；

000(IF):当前 PC 值为 44；
下条指令地址为 48

001(ID):Rs 寄存器为 12，值为 00(ADROut)；
Rt 寄存器为 00，值为 00(BDROut);
Imm 值为-2；

101(EXE)：ALU_A、ALU_B 分别来自 rs, rt; 结果为 00(ALUoutDROut)



andi $12,$2,2

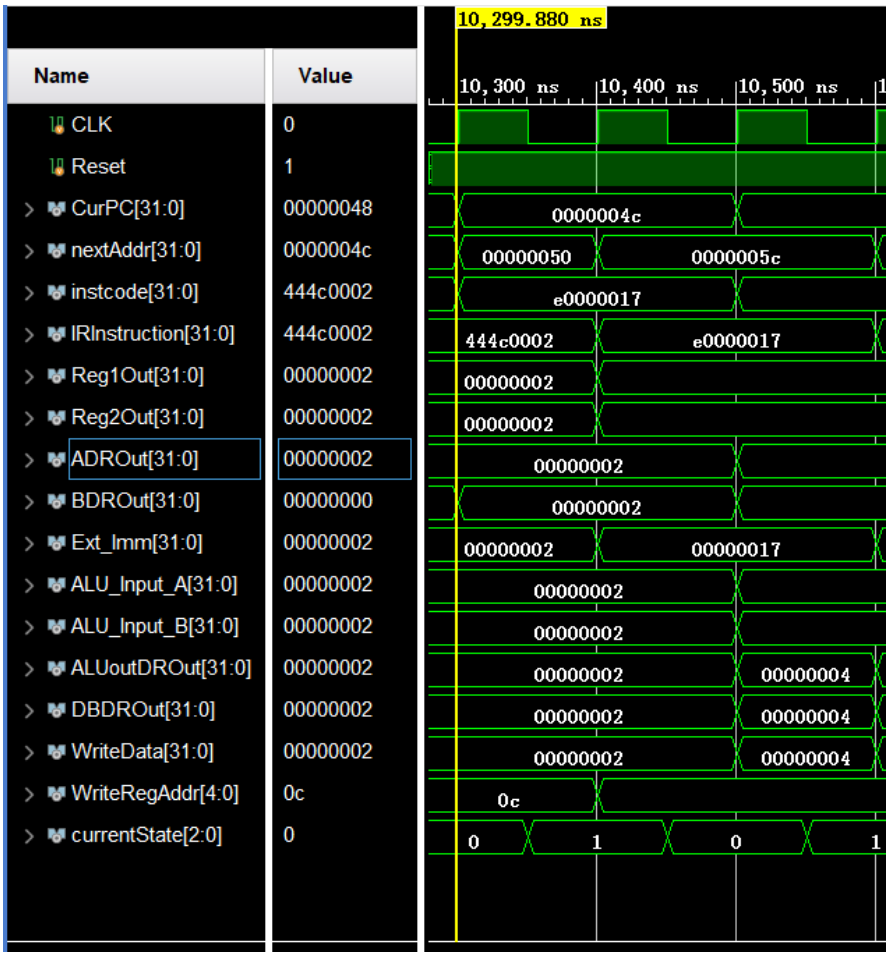
该指令执行 4 个时钟周期，分别对应状态转换图中的 000->001->110->111 四个状态；

000(IF):当前 PC 值为 48；
下条指令地址为 4c；

001(ID):Rs 寄存器为 02，值为 02(ADROut)；
Imm 值为 2；

110(EXE)：ALU_A、ALU_B 分别来自 rs, imm; 结果为 02(ALUoutDROut)

111(WB)：写回寄存器 rt 为 0c，值为 02WriteData)

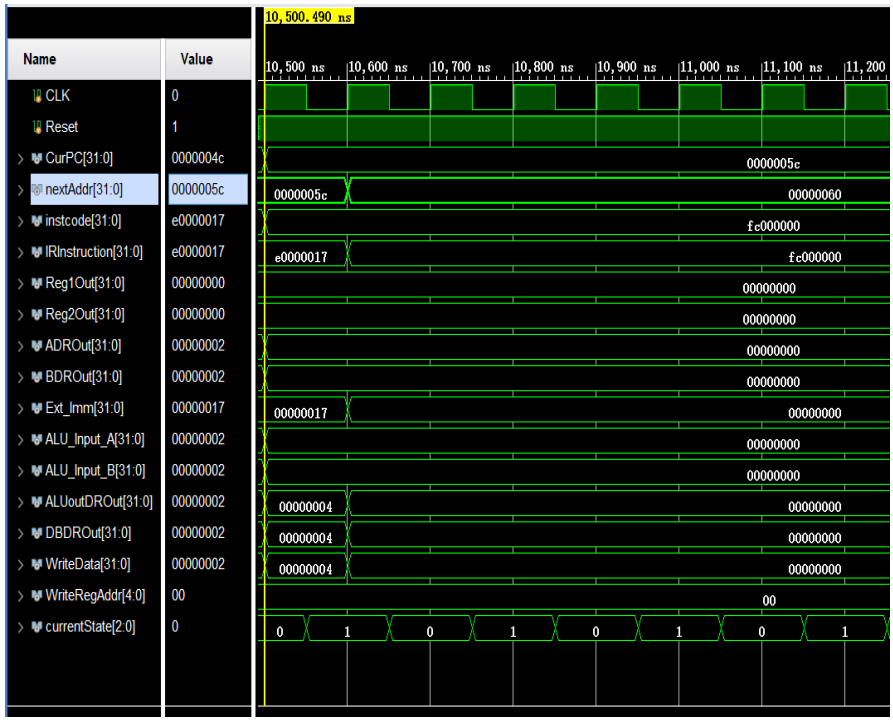


j 0x000005C

该指令执行 2 个时钟周期，分别对应状态转换图中的 000->001 两个状态；

000(IF):当前 PC 值为 4c；

001(ID):下条指令地址为 5c



halt

该指令执行 2 个时钟周期，分别对应状态转换图中的 000->001 两个状态；

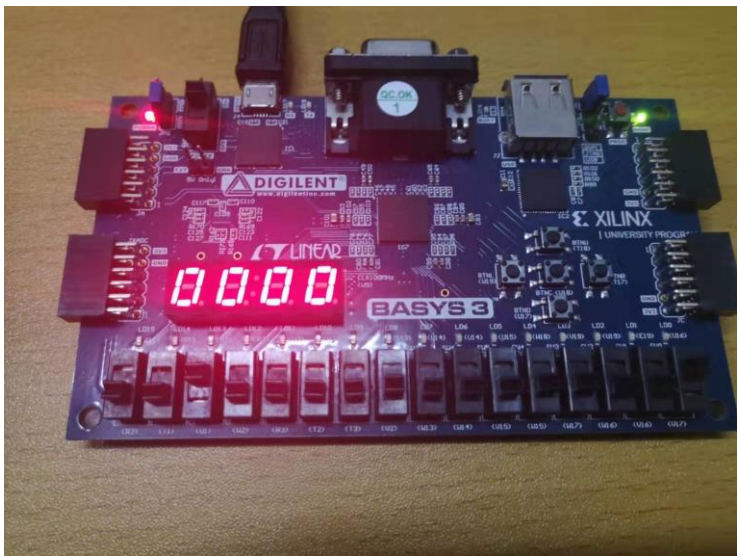
000(IF):当前 PC 值为 5c；

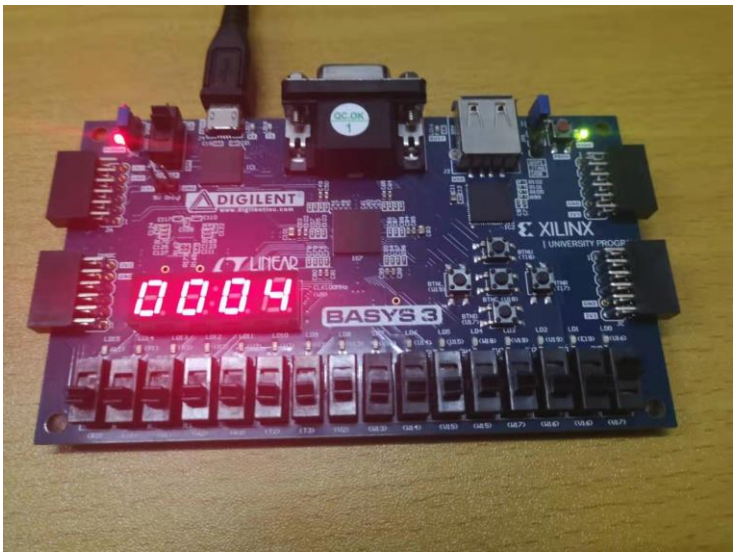
001(ID):下条指令地址为 60,但是一直读不到下条指令，所以程序暂停；

7. 烧写开发板之后功能展示

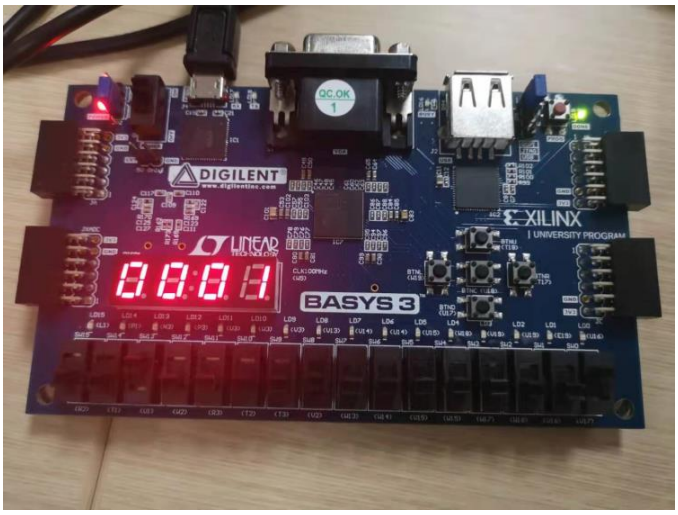
（一） addiu $1,$0,8

状态为 000(IF，下图的右三位)：当前 PC 为 00，下条指令地址为 04



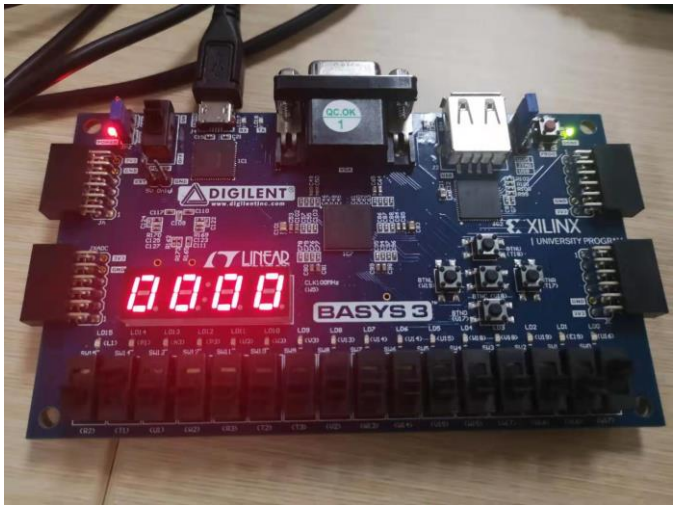


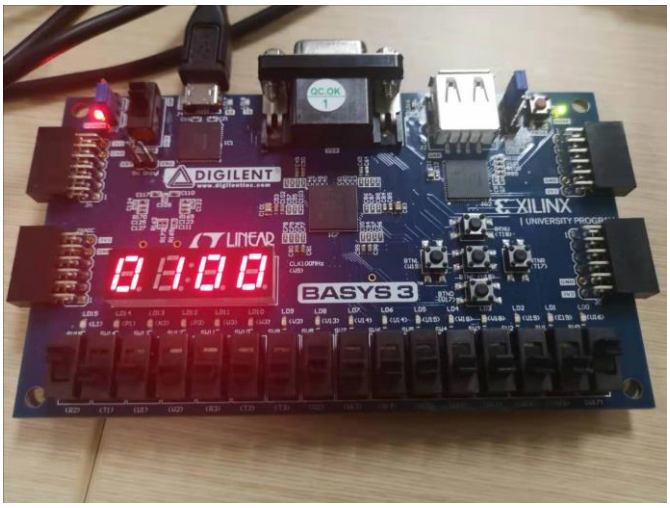
状态为 001(ID，下图的右三位)：



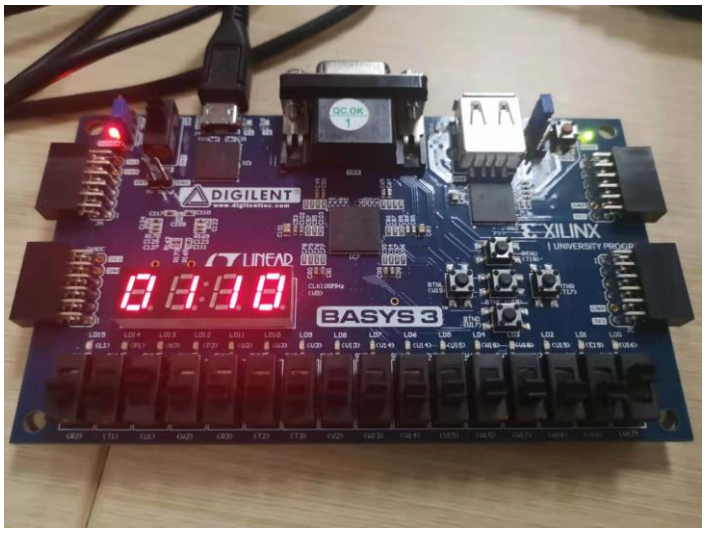
此时 rs 为 00，值为 00；

rt 为 01，值为 00(因为还没有写回值)

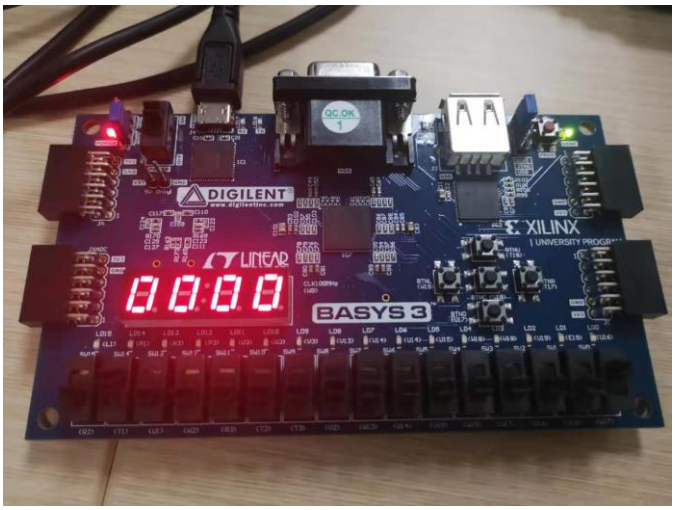


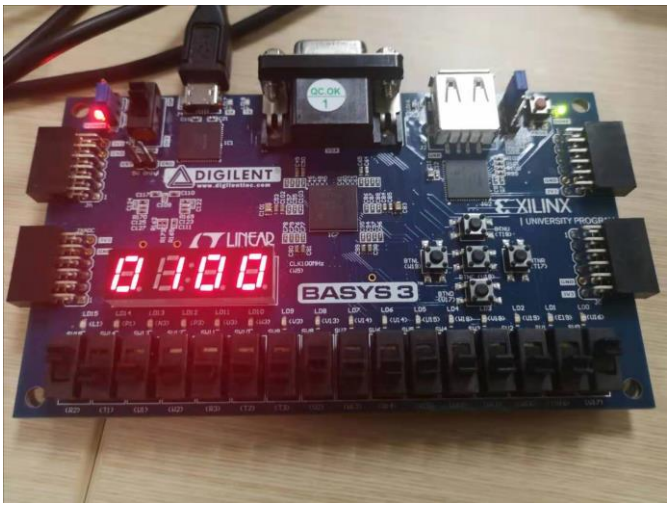


状态为 110(EXE，下图的右三位)：

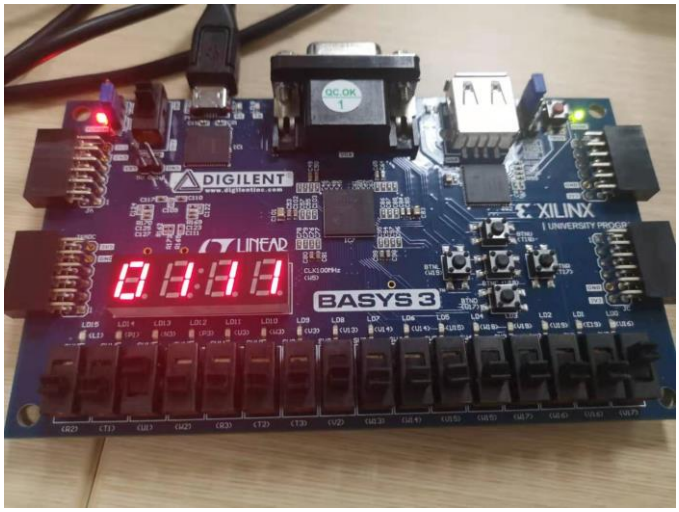


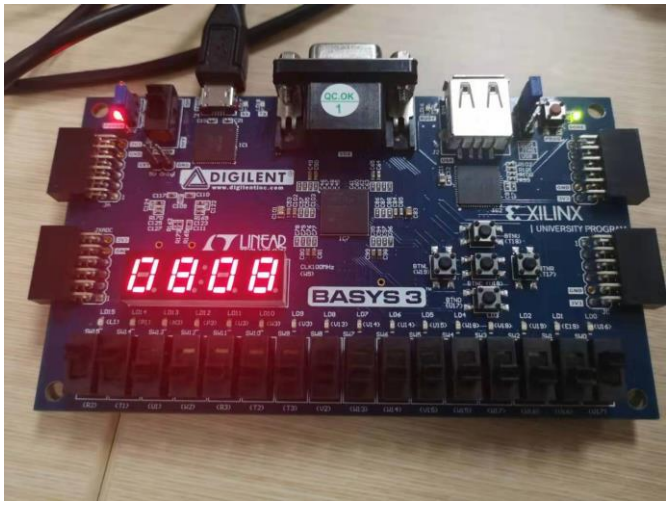
此时 rs,rt 的值都还未发生改变；





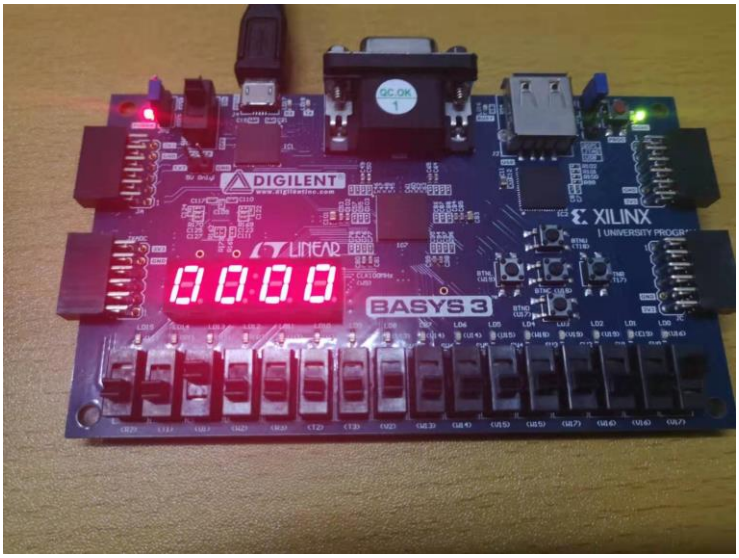
状态为 111(WB，下图的右三位)：此时写回 rt 是 01,DBDROut 与 WriteData 都为 08；

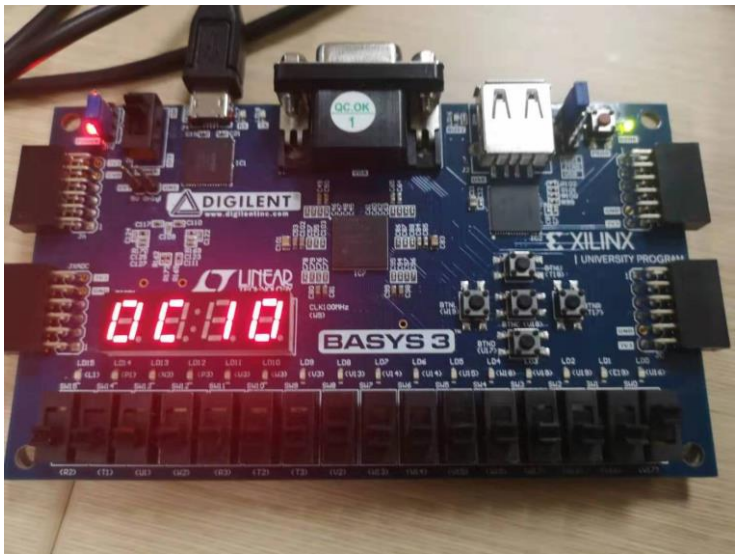




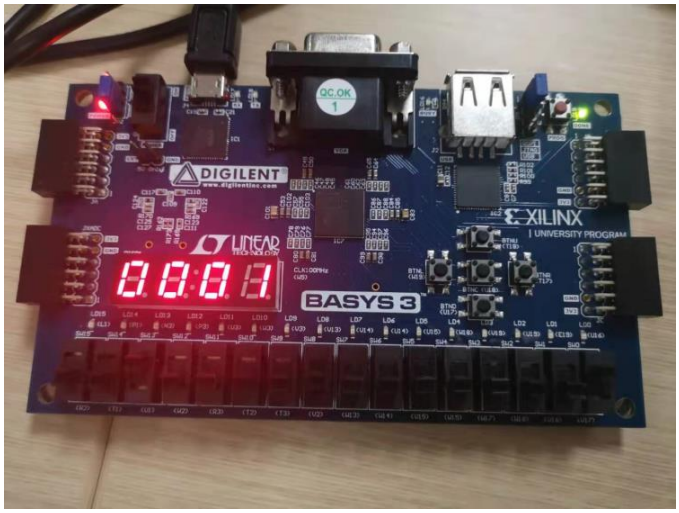
（二）sub　$4,$3,$1

状态为 000(IF，下图的右三位)：当前 PC 为 0c，下条指令地址为 10;

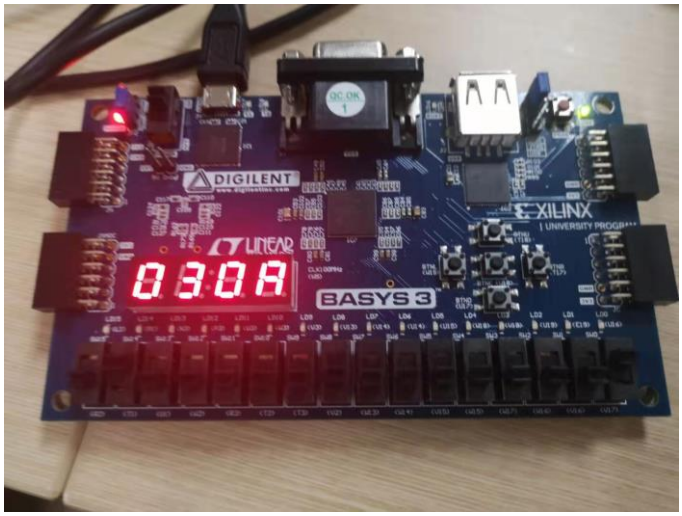


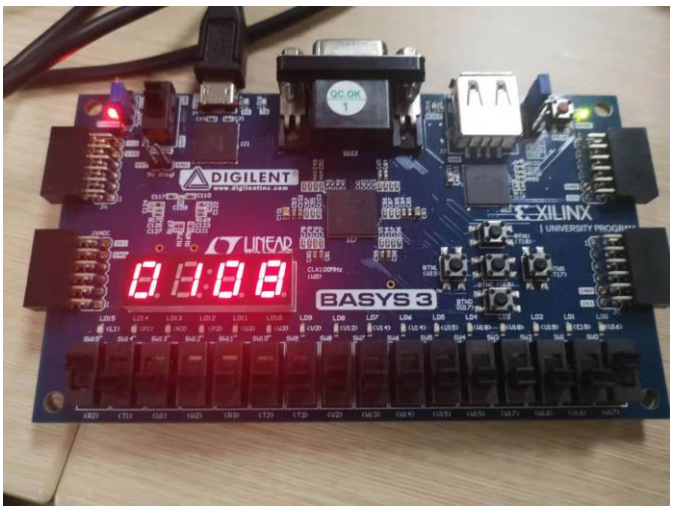


状态为 001(ID，下图的右三位):

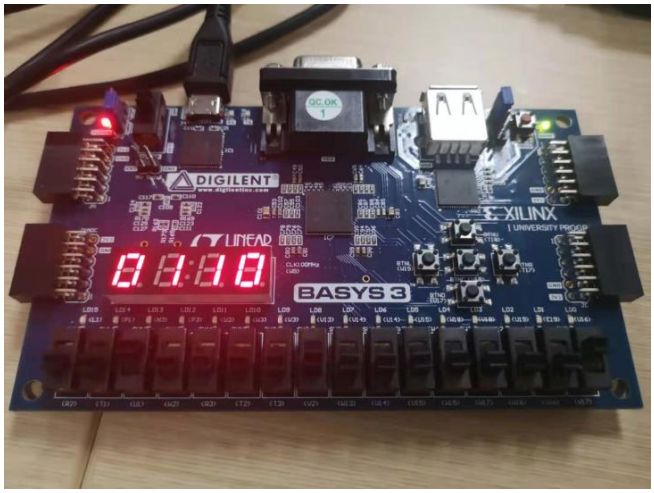


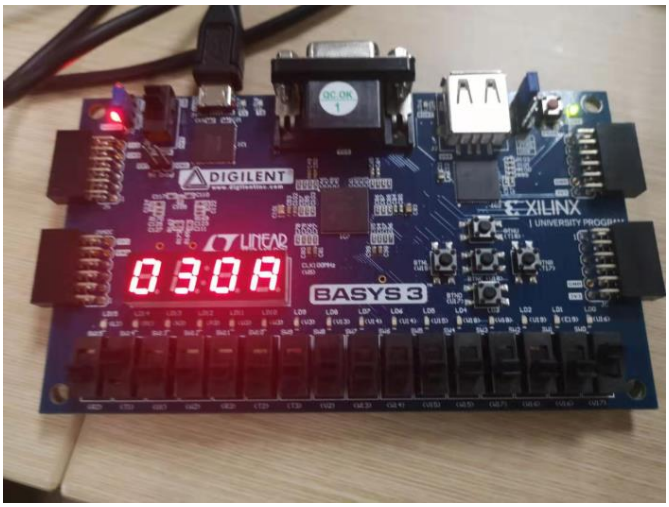
此时 rs 为 03，值为 0a(ADROut)；
rt 为 01，值为 08(BDROut)

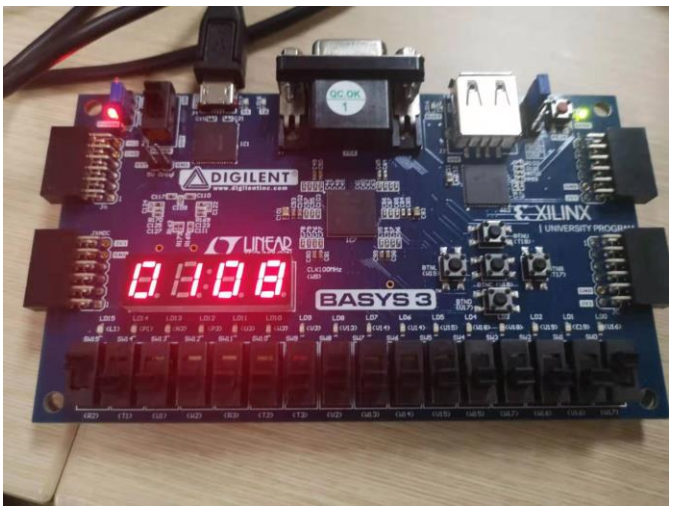




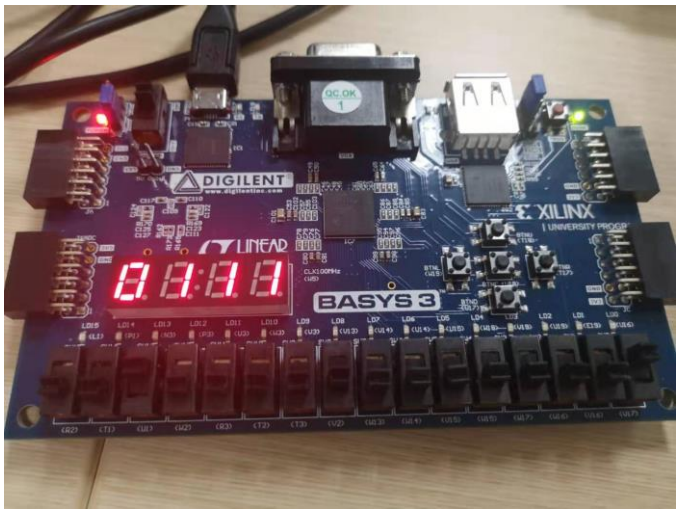
状态为 110(EXE，下图的右三位):

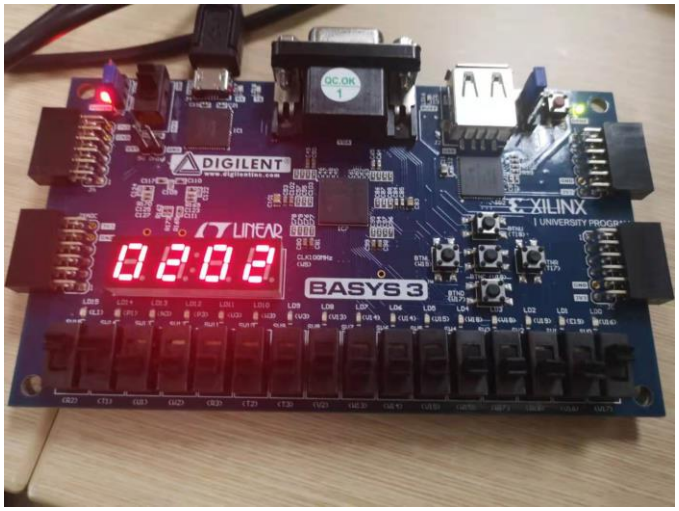






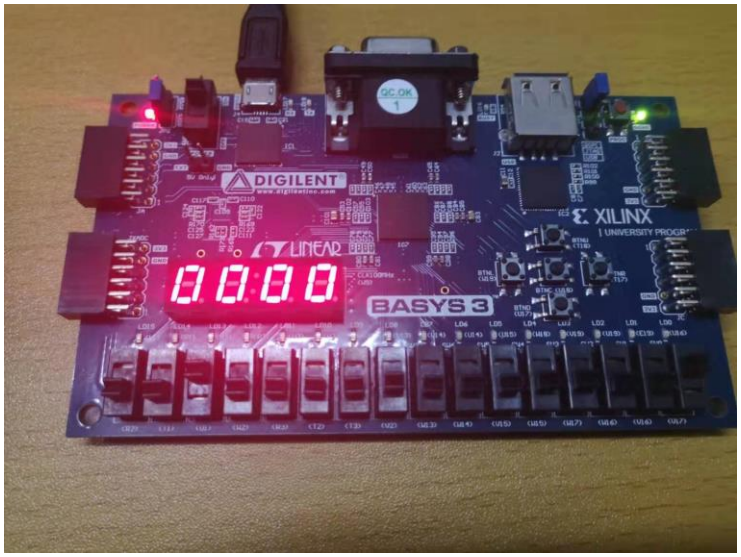
状态为 111(WB，下图的右三位)：此时写回 rd 是 04,DBDROut 与 WriteData 都为 02

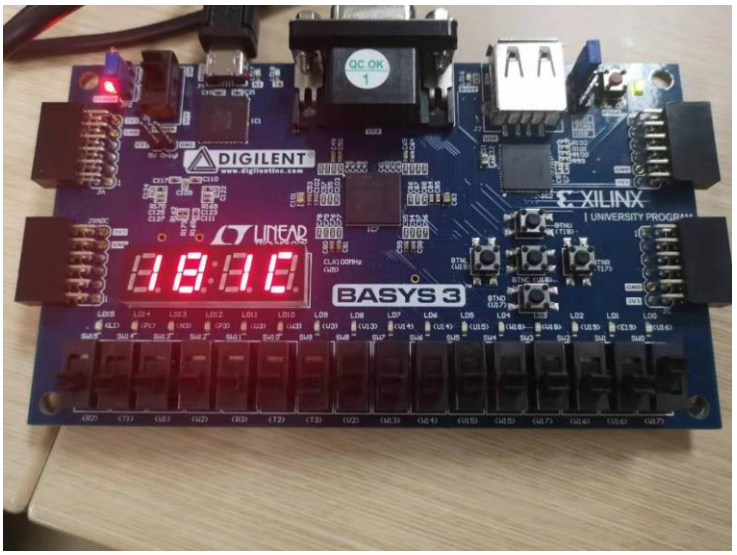




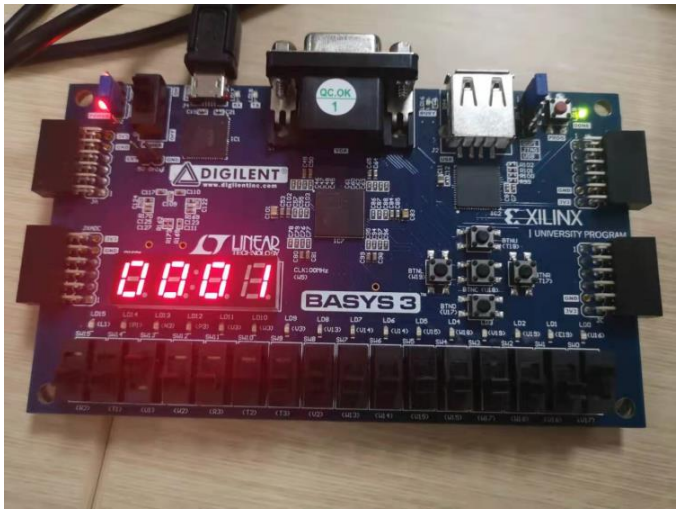
（三）beq　$5,$1,-2(=,转 14)

000(IF，下图的右三位):当前 PC 值为 18；



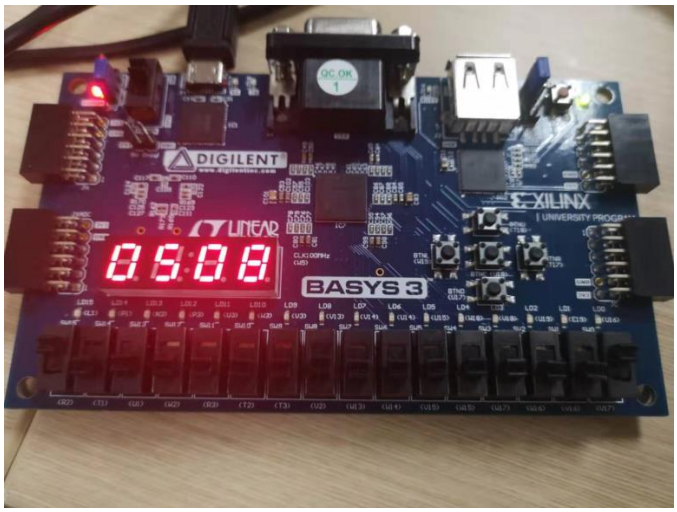


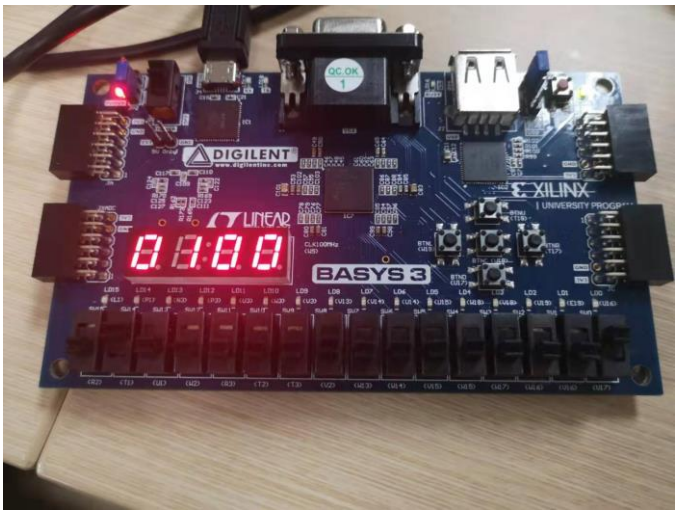
状态为 001(ID，下图的右三位):



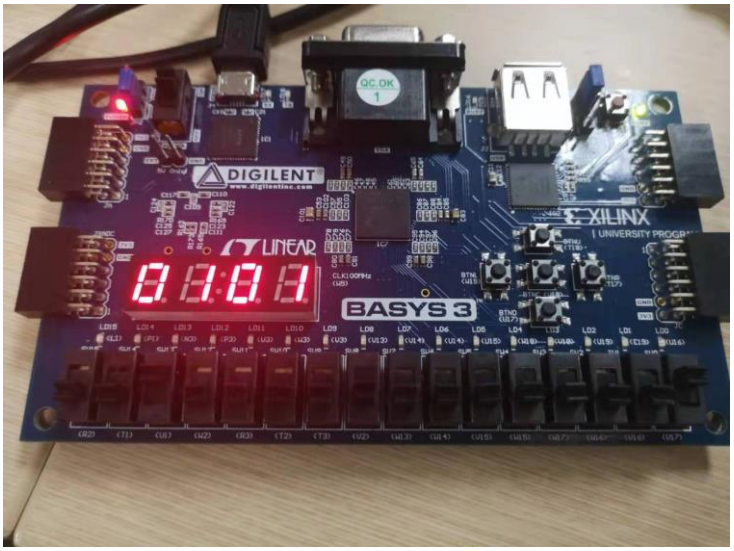
此时 rs 为 05，值为 08(ADROut)；

Rt 为 01，值为 00(实际为 08)
因为显示 rt 值用的是 IRInstruction 上升沿触发，而 IRInstruction 与 ADROut 输出不同步，所以要在下一个周期才能看到值为 08

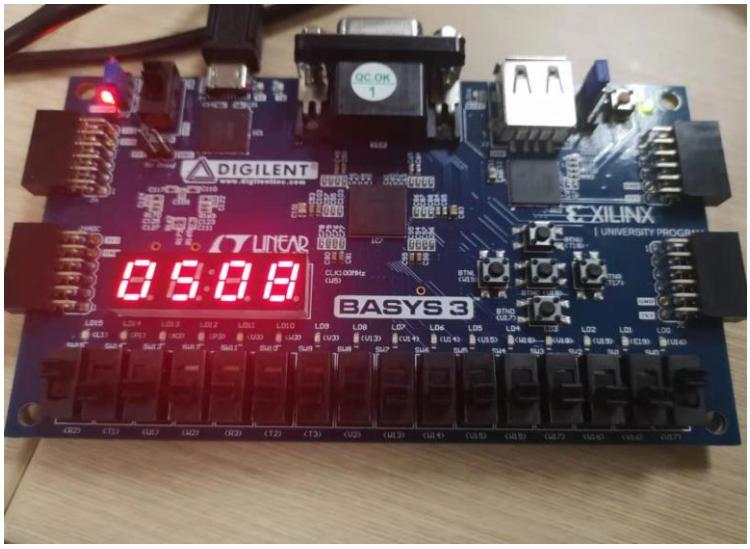


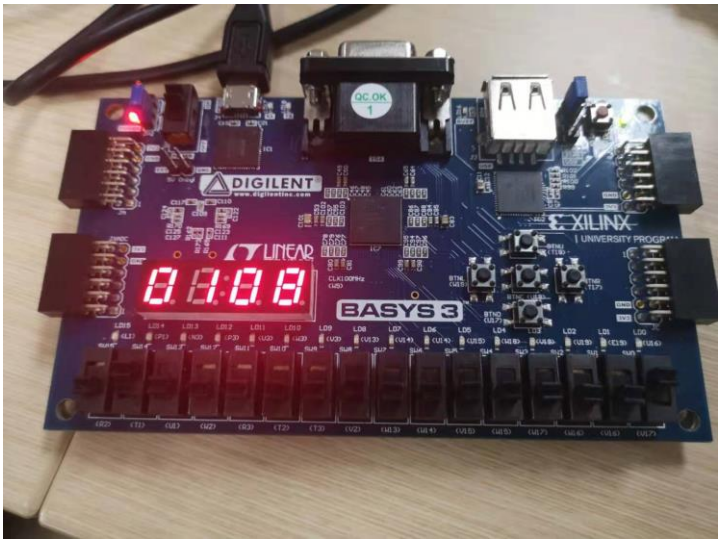


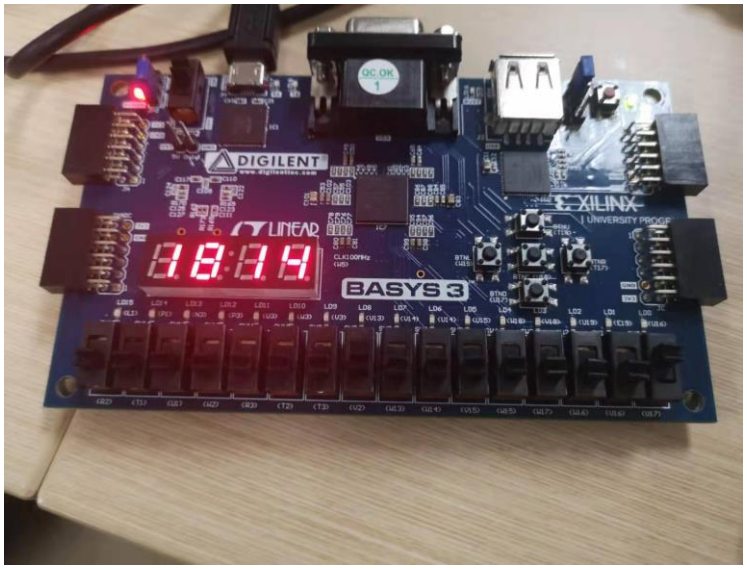
状态为 101(EXE，下图的右三位):



根据上个周期的说明，此时 rt 显示值为 08；
Rs 没有发生改变；



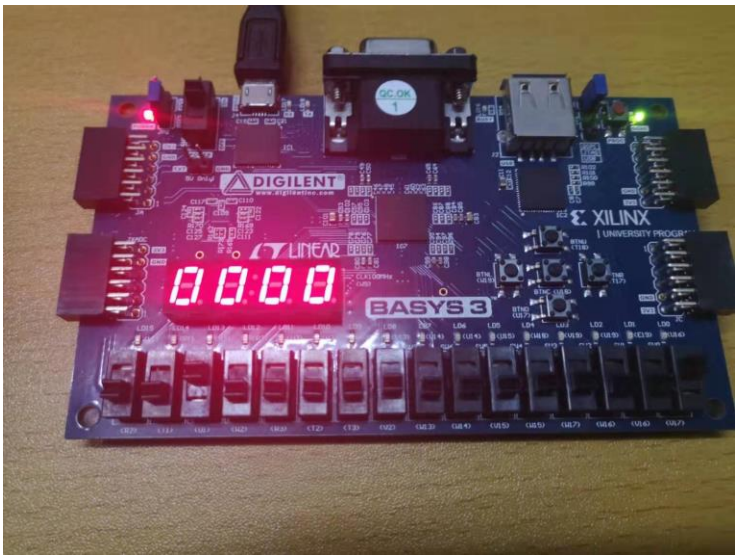


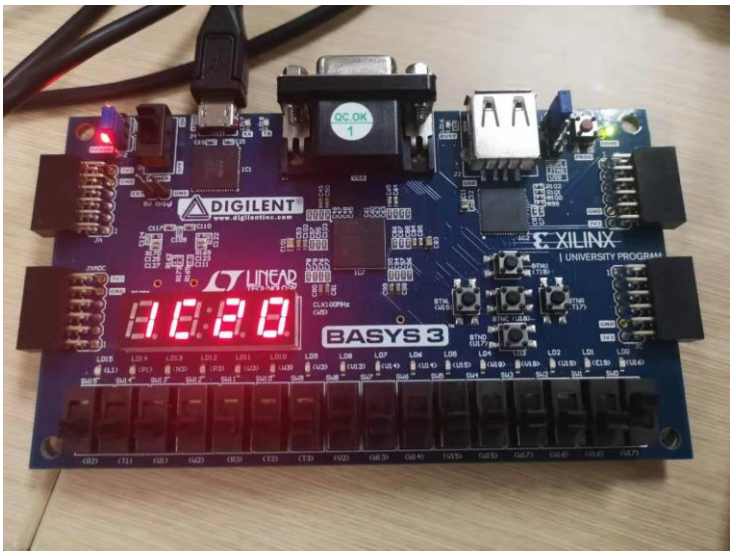


运行得到下条指令地址为 14

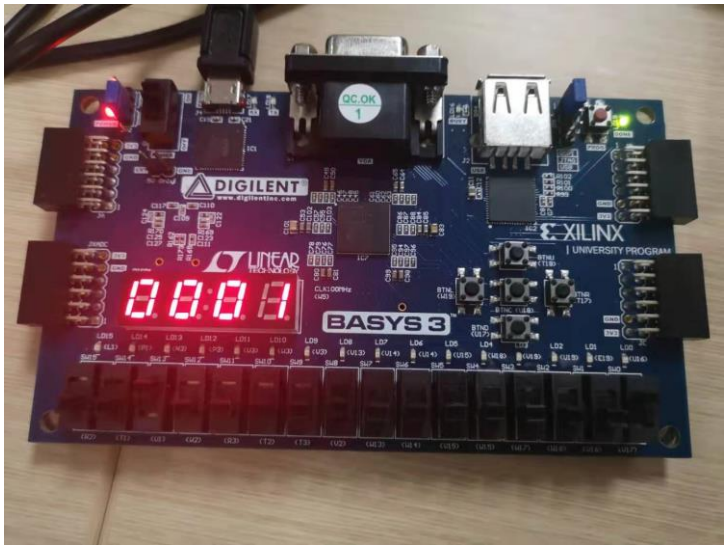
（四）jal　0x0000050

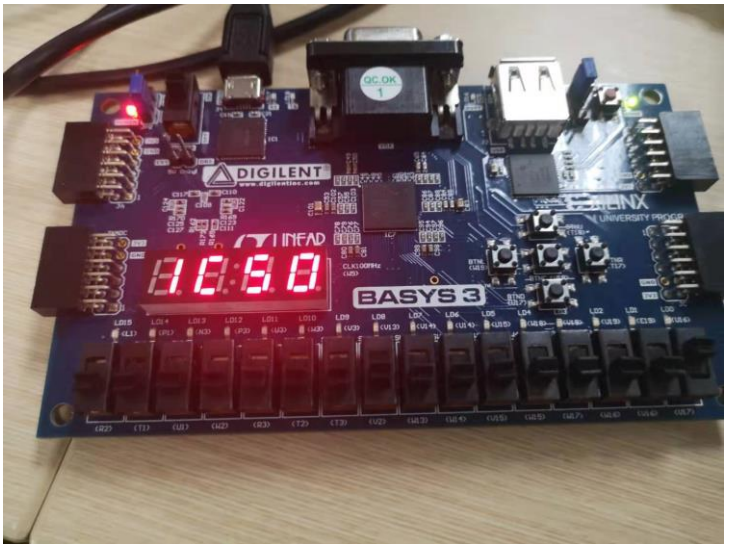
000(IF，下图的右三位):当前 PC 值为 1c





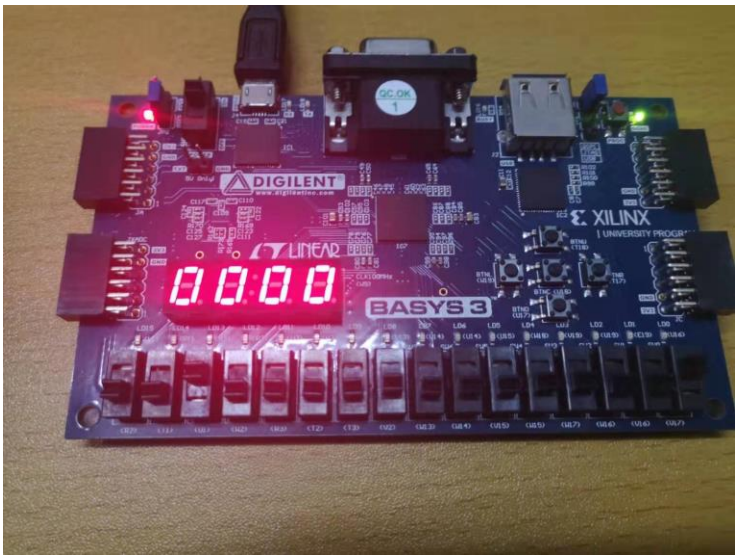
状态为 001(ID，下图的右三位):下条指令地址为 50

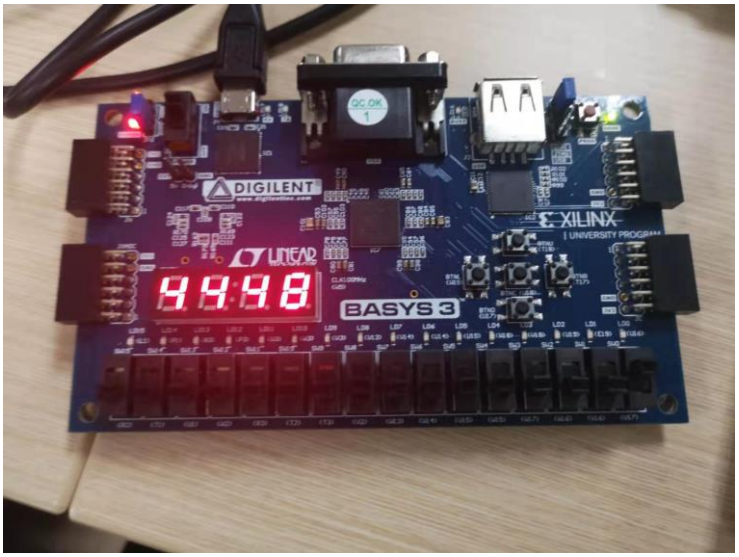




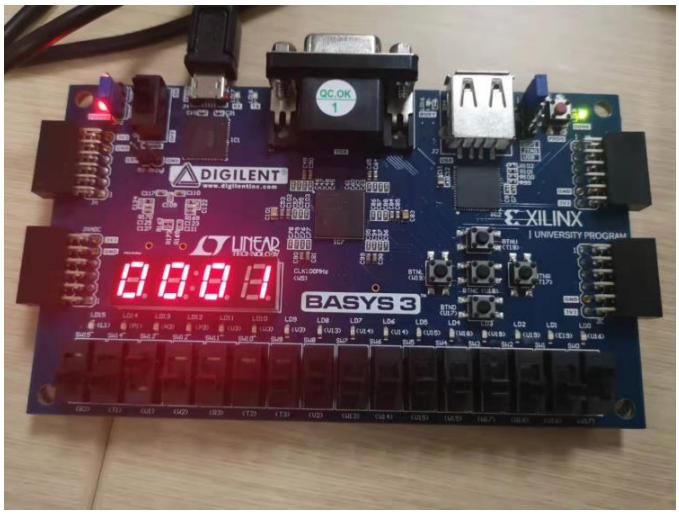
（五）**bltz　$12,-2 (<0,转 40)**

000(IF，下图的右三位):当前 PC 值为 44

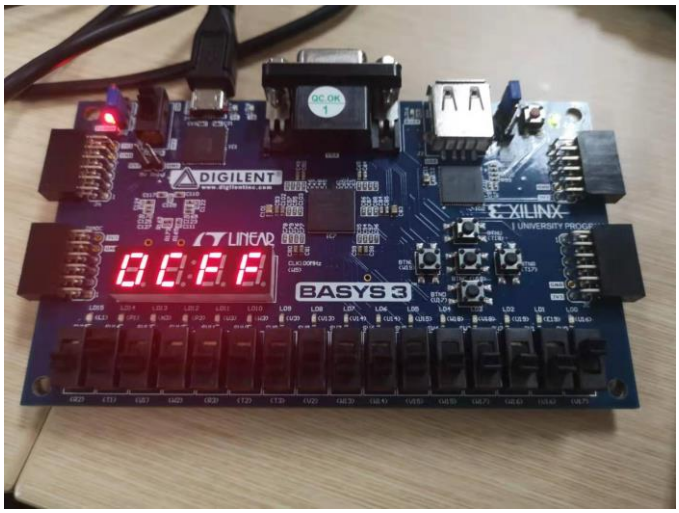


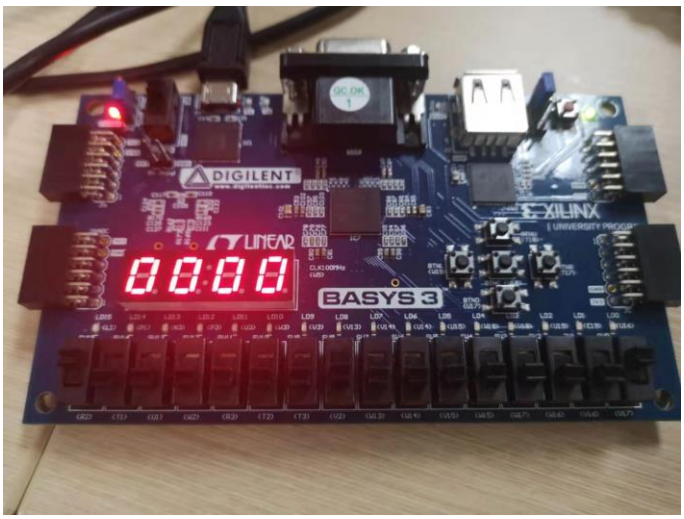


状态为 001(ID，下图的右三位):

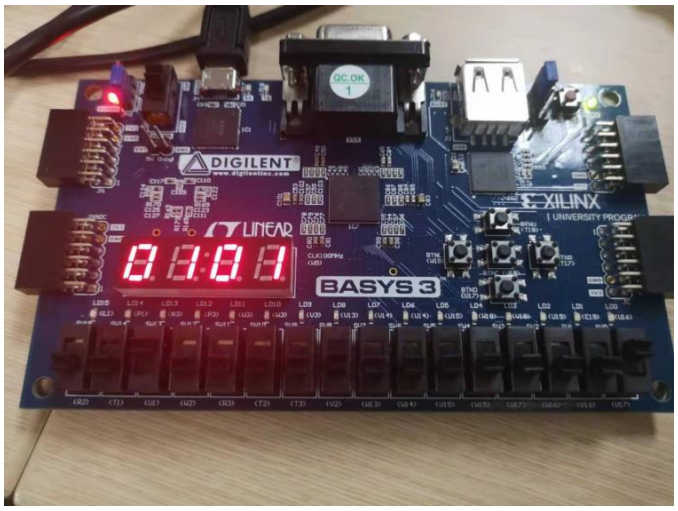


此时此时 rs 为 0c，值为 ff(ADROut)；
Rt 为 00，值为 00(BDROut)

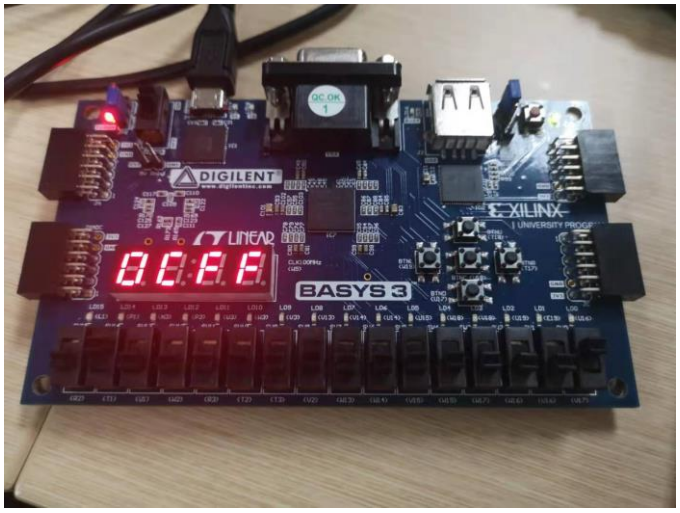


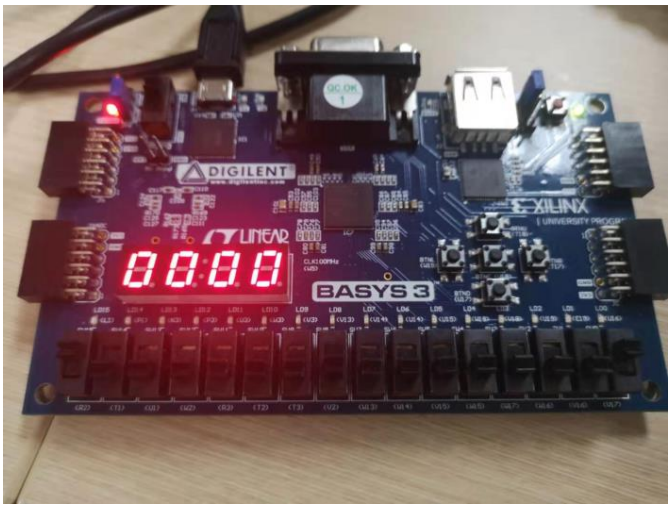


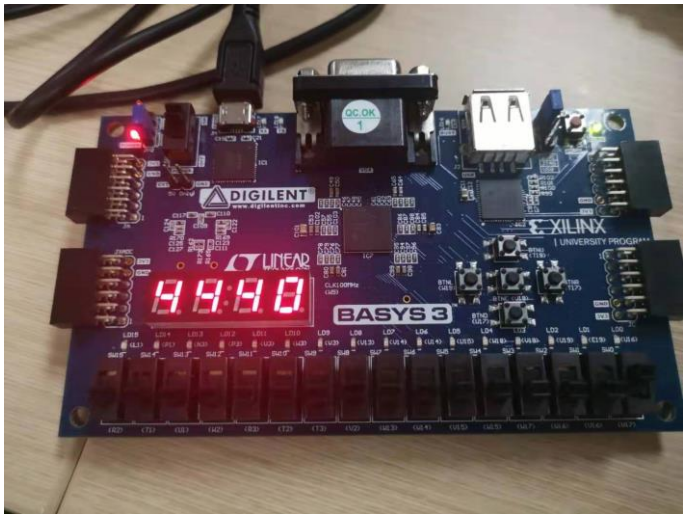
状态为 101(EXE，下图的右三位):



根据上个周期的说明，此时 rt 显示值为 08；
Rs 没有发生改变；







运行得到下条指令地址为 40

四、 实验心得

在本次实验中，因为有了单周期 CPU 的铺垫，在实现多周期 CPU 的时候的确省去了一些麻烦，很多文件都可以直接用单周期 CPU 的文件，但在实验过程中还是遇到了一些问题。这些问题有一部分是由于竞争冒险产生的，还有一些是一时的大意，打错了字符或者是信号的变化写错了…都会产生极大的麻烦。

在 Hard_CPU.v 中，由于使用了单周期 CPU 的代码，后面再完善多周期 CPU 的时候，忘记了改显示模式部分内容，导致 3'b000 等代码都是 2'b000，更难受的 VIVADO 没有报错，这就导致了一直做到烧完板子之后发现有问题，回来找了很久才发现是这里出了问题。在 mCPU.v 中，也是由于使用单周期 CPU 部分的代码，导致忘记在多周期 CPU 中使用的是 instcode 来给包括 RegisterFile.v，Extend.v 在内的多个文件模块都使用了 inscode，但实际上应该使用 IRInstruction，这导致包括 beq,bne,bltz 等指令在内的多条指令都产生了非常迷惑的行为，并且符号扩展也在 IF 阶段就已经开始了，这个 bug 也是找了很久才发现。

除了上面我们的个人问题之外，模板上有些问题也是让人哭笑不得。有数条指令在单周期的文档和多周期的文档中竟然是不一样的（例如 and 在单周期的文档是 010001，在多周期的文档是 010000），导致在修改单周期的 CU 模块变成多周期的 CU 模块时一脸懵逼。

不过在花费很多心思完成这个实验后，也明显地感觉到我们对 CPU 的了解进一步加深。虽然 dubug 的时间比写 bug 的时间长了数倍，但在 debug 的过程中不断发现自己对多周期 CPU 的认识误区，也是收获良多的。在最后验证了正确性的一刻，自己也有完成一个小工程的成就感，想必以后在学术研究课题抑或是工程得到完成也会有这样的感觉吧。

既更深入了解了CPU，又增强了自己对事物钻研的耐心，更重要的是这次实验又提高了我们对计组的兴趣!