《计算机组成原理与接口技术实验》

实验报告

（实验三）

学 院 名 称：数据科学与计算机学院

专业（班级）：16 软件工程四（7）班

学 生 姓 名：杨元昊

学 号：16340274

时 间：2018 年 6 月 11 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原理及其设计方法；

(2) 掌握多周期 CPU 的实现方法，代码实现方法；

(3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；

(4) 掌握多周期 CPU 的测试方法;

(5) 掌握多周期 CPU 的实现方法。

二、 实验内容

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

==>算术运算指令

（1）add rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd< - rs + rt

（2）sub rd, rs, rt

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

完成功能：rd< - rs - rt

（3）addi rt, rs, **immediate**

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

功能：rt< - rs + (sign-extend)**immediate**

==>逻辑运算指令

（4）or　rd, rs, rt

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

功能：rd<-rs | rt

（5）and　rd, rs, rt

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

功能：rd<-rs & rt

（6）ori　rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

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

==>移位指令

（7）sll　rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<-rt<<(zero-extend)sa，左移 sa 位　，(zero-extend)sa

==>比较指令

（8）　slt　rd, rs, rt　　带符号数

100110	rs(5 位)	rt(5 位)	rd(5 位)	reserved

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

（9）sltiu　rt, rs,immediate　　不带符号

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if (rs <(zero-extend)**immediate)**　rt =1 else　rt=0，具体请看表 2 ALU 运算功能表，不带符号

==>存储器读写指令

（10）sw　rt, **immediate(**rs)

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

功能：memory[rs+ (sign-extend)**immediate**]< - rt。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（11）lw　rt, **immediate**(rs)

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

功能：rt < -　memory[rs + (sign-extend)**immediate**]。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==>分支指令

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

110100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs=rt) pc < - pc + 4 + (sign-extend)**immediate** <<2 else pc < - pc + 4

(13）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

（14）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。

（15）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc < - rs，跳转。

==>调用子程序指令

（16）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

（17）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。

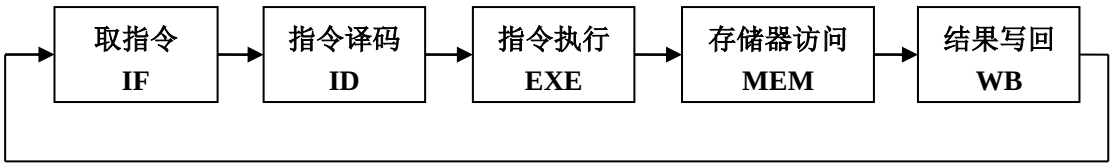
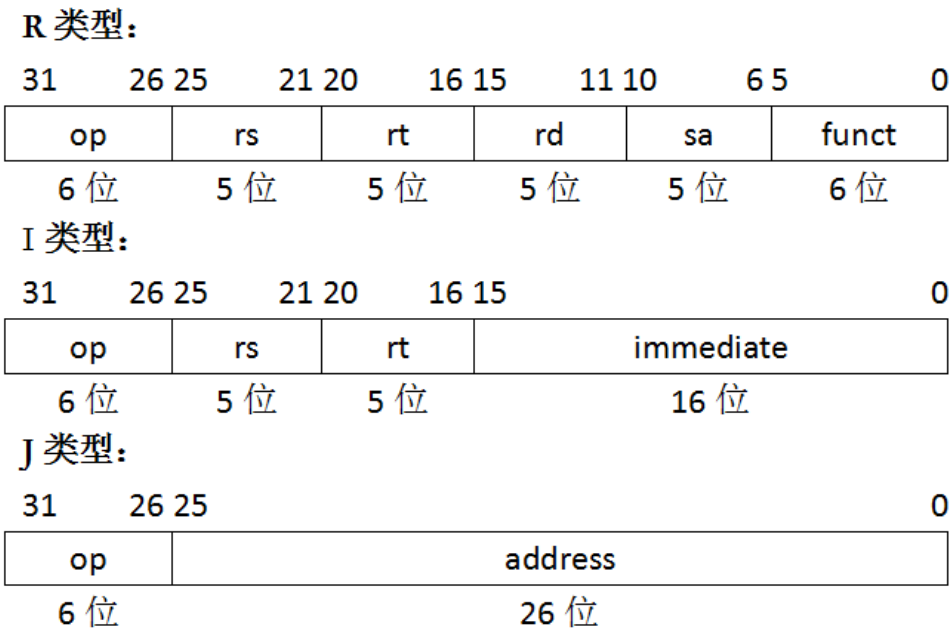


图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式：



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。

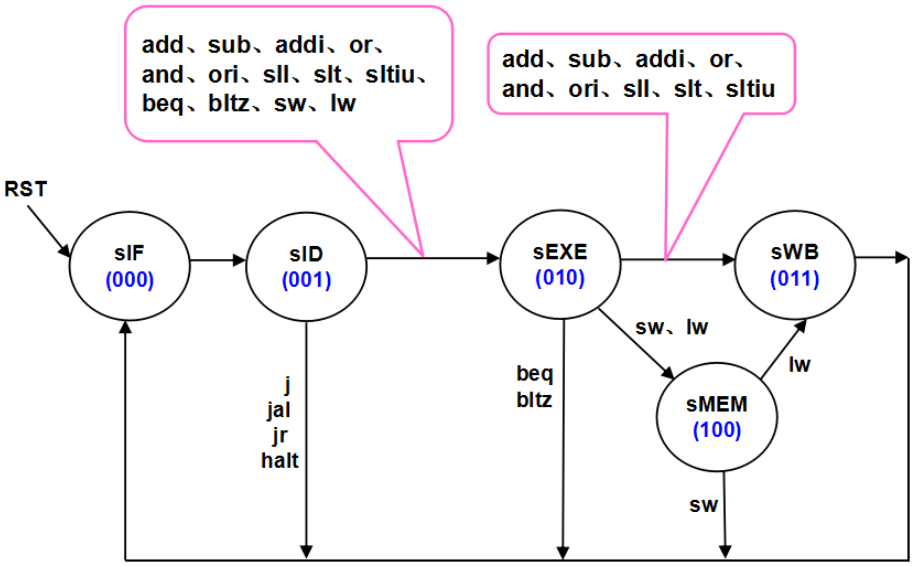


图 2 多周期 CPU 状态转移图

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。

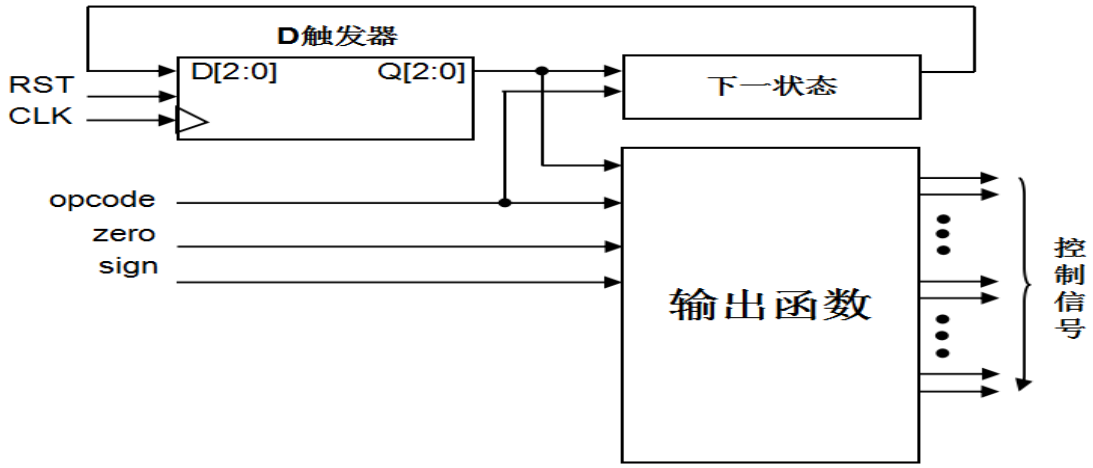


图 3 多周期 CPU 控制部件的原理结构图

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。

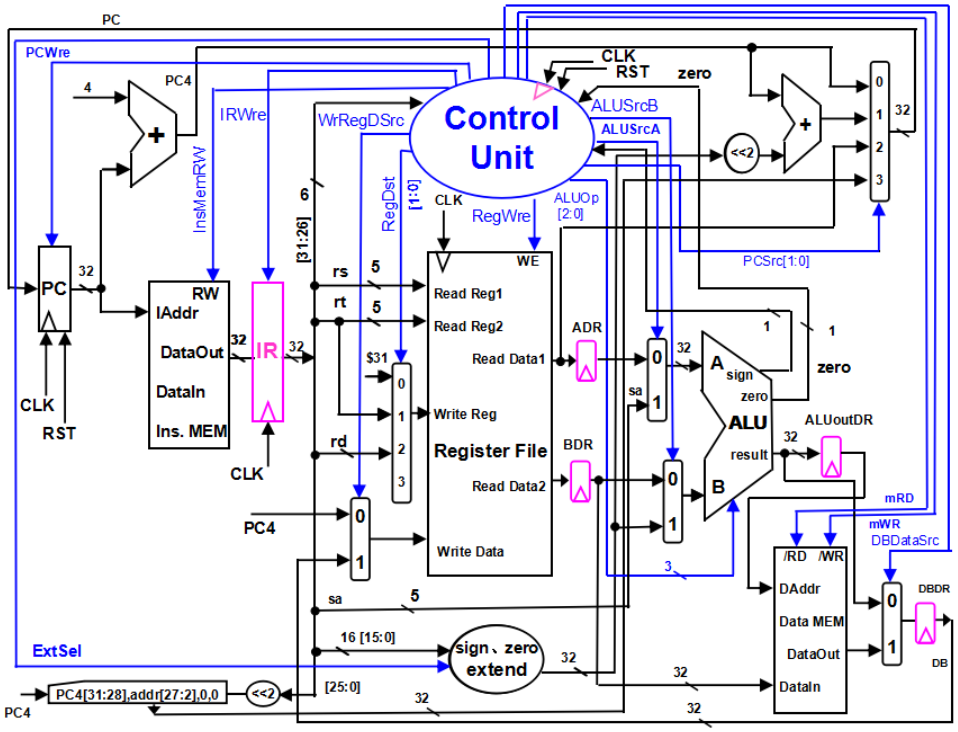


图 4 多周期 CPU 数据通路和控制线路图

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、addi、or、and、ori、beq、bltz、slt、sltiu、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、sltiu、lw、sw

	bltz、slt、sll	
DBDataSrc	来自 ALU 运算结果的输出 ,相关指令：add、sub、addi、or、and、ori、slt、sltiu、sll	来自数据存储器(Data MEM)的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bltz、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、slt、sltiu、sll、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4 ），相关指令：jal，写 $31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据 ,相关指令：add、addi、sub、or、and、ori、slt、sltiu、sll、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：ori、sltiu；	(sign-extend)**immediate**，相关指令：addi、lw、sw、beq、bltz；
PCSrc[1:0]	00：pc< - pc+4，相关指令：add、addi、sub、or、ori、and、	

	slt、sltiu、sll、sw、lw、beq(zero=0)、bltz(sign=0，或 zero=1)； 01：pc< - pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bltz(sign=1，zero=0)； 10：pc< - rs，相关指令：jr； 11：pc< - {(pc+4)[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：addi、ori、sltiu、lw； 10：rd 字段，相关指令：add、sub、or、and、slt、sll； 11：未用；
ALUOp[2:0]	ALU 8 种运算功能选择(000-111)，看功能表

相关部件及引脚说明：

Instruction Memory：**指令存储器**

Iaddr，指令地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：**数据存储器**

Daddr，数据地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

/RD，数据存储器读控制信号，为 0 读

/WR，数据存储器写控制信号，为 0 写

Register File：**寄存器组**

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器，其地址输入端口（rt、rd）

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

IR： 指令寄存器，用于存放正在执行的指令代码

ALU： 算术**逻辑单元**

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

sign，运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A – B	减
010	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
011	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 &&	比较 A 与 B 带符号

	regb[31] == 0))) ? 1:0	
100	**Y = B<<A**	**B 左移 A 位**
101	**Y = A ∨ B**	**或**
110	**Y = A ∧ B**	**与**
111	**Y = A ⊕ B**	**异或**

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、实验设计

相比于单周期 cpu，多周期 cpu 要求我们对时序控制很严格，否则总是会出现各类各样的 bug。下面我们以表格的形式总结下各个指令涉及的阶段，以及不同阶段对时钟周期的处理。

指令	IF 阶段	ID 阶段	EXE 阶段	MEM 阶段	WB 阶段
Add	√	√	√		√
Sub	√	√	√		√
Addi	√	√	√		√
Or	√	√	√		√
And	√	√	√		√
Ori	√	√	√		√
Sll	√	√	√		√
Slt	√	√	√		√
Sltiu	√	√	√		√
Sw	√	√	√	√	

Lw	√	√	√	√	√
Beq	√	√	√		
Bltz	√	√	√		
J	√	√			
Jr	√	√			
Jal	√	√			
Halt	√	√			

指令阶段/时钟周期	IF 阶段	ID 阶段	EXE 阶段	MEM 阶段	WB 阶段
上升沿	控制器模块进行运算，更新 PC 模块和 IR 模块的写信号	控制器模块进行运算,更新寄存器的写信号并根据 IF 阶段 IR 寄存器的写信号来写入新数据	控制器模块进行运算，更新 Alu 操作码	控制器模块进行运算	控制器模块进行运算,更新寄存器写信号 PCsrc
下降沿	根据上升沿的写信号，让 PC	根据本阶段上升沿的寄存器	向 ADR，BDR 模块写入值	向内存单元中写入值	向寄存器中写入值

	模块更新，写入新地址	写信号来写入数据			

根据数据通路图，我们可以设计如下的模块，与单周期 cpu 一致的将会粗略的描述或略过不讲。

PC 模块：

输入时钟信号和重置信号，PC 状态更改信号 PCWre 和下一个 PC 值，当 PCWre 为 0 时 PC 模块输出的值就是当前 PC+4 的值，否则就会做相应的变化，当重置信号为真是 PC 的值将变为 0。关键代码如下

```
    if (!Reset) begin
        NextPC = PC + 4;
    end
    else begin
        case (PCSrc)
            2'b00: NextPC = PC + 4;
            2'b01: NextPC = PC + 4 + (Immediate << 2);
            2'b10: NextPC = JPC;
            default: NextPC = PC + 4;
        endcase
    end
```

指令存储模块：

与单周期 CPU 一模一样，且只涉及读取并不复杂，故略去。

IR 寄存器模块：

在之前的 PC 模块设计中，时钟下降沿到来时 PC 模块会进行相关操作。而过了一个周期。当时钟上升沿到来时，IR 寄存器就会暂存指令存储模块中当前 PC 处的各类指令。关键代码如下

```
always@(posedge CLK) begin
    Op_code <=   Ins_Data[31:26];
    Rs_reg <= Ins_Data[25:21];
    Rt_reg <= Ins_Data[20:16];
    Rd_reg <= Ins_Data[15:11];
    Sa_number <= Ins_Data[10:6];
    Imm_number <= Ins_Data[15:0];
end
```

四选一数据选择器：

根据 control unit 输出的控制信号，和输入的 rt 寄存器地址，rd 寄存器地址，第 31 号寄存器地址，判断最终写入的寄存器的地址是多少。比如 jal 指令就会写第 31 号寄存器。除此之外也承担着 PC 值改变的选择，（四个选项：pc< - pc+4， pc< - pc+4+(sign-extend)**immediate**, pc< - rs, pc< - {(pc+4)[31:28],addr[27:2],2'b00}）关键代码如下

```
always@(*) begin
    if (Reset == 0) begin
        DataOut = 0;
    end
```

```
        else begin
            case(sel)
                2'b00: DataOut = DataIn1;
                2'b01: DataOut = DataIn2;
                2'b10: DataOut = DataIn3;
                2'b11: DataOut = DataIn4;
            endcase
        end
    end
```

寄存器模块：

当时钟下降沿来临时，就会将数据写入寄存器，当下降沿的清空信号来临时就会将所有寄存器中数据置为 0。而读取寄存器的值是同步的。关键代码如下

```
    assign    ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
    assign    ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
    always @ (negedge CLK or negedge RST) begin
        if (RST==0) begin
            for(i=1;i<32;i=i+1) regFile[i] <= 0;
        end
        else if(RegWre == 1 && WriteReg != 0)
            regFile[WriteReg] <= WriteData;
    end
```

临时存储的 ADR 与 BDR 模块：

当时钟上升沿到来时，寄存器中读出来的值会暂时放置在这两个寄存器模块中。分别放 RS 寄存器和 Rt 寄存器中的值。关键代码如下

```
reg [31:0] data;
always@(posedge CLK) begin
    data = Data_in;
end
```

数据扩展模块：

实现扩展立即数的功能，根据输入的选择位来决定是有符号位扩展还是无符号位扩展。

与单周期 cpu 一样，不再贴出代码。

二选 1 数据选择模块

共有两个这样的模块 ,其中 AluSrcA 是选择 Rs 寄存器的值还是移位指令的 sa ,AluSrcB 是选择 Rt 寄存器的值还是经过扩展的立即数。与单周期 cpu 一样，不再贴出代码。

Alu 运算模块：

与单周期 CPU 的设计一致，根据 Alu 指令码的值对输入的两个数据做不同操作，并将结果输出。与单周期 cpu 一样，不再贴出代码。

数据存储器模块：

在时钟下降沿时将数据写入内存单元，而数据的读取则是异步操作。关键代码如下

```
assign Dataout[7:0] = (nRD==0)?ram[DAddr + 3]:8'bz;
assign Dataout[15:8] = (nRD==0)?ram[DAddr + 2]:8'bz;
assign Dataout[23:16] = (nRD==0)?ram[DAddr + 1]:8'bz;
assign Dataout[31:24] = (nRD==0)?ram[DAddr]:8'bz;
always@( negedge CLK ) begin
```

```
        if( nWR==0 ) begin
            ram[DAddr] <= DataIn[31:24];
            ram[DAddr+1] <= DataIn[23:16];
            ram[DAddr+2] <= DataIn[15:8];
            ram[DAddr+3] <= DataIn[7:0];
        end
```

32 位数据的二选一模块：

选择内存单元中的值或是 Alu 的运算结果。与之前所述一致，故不贴代码了。

DBDR 模块：

暂时储存内存单元读出的值，与上文 ADR，BDR 模块一致，不再赘述。

控制器模块：

这个模块是 CPU 设计的核心，时钟上升沿触发。它又分为两个子模块。一个模块用于实现状态的转移，将当前状态转变为下一个状态。另一个模块根据转移的状态和指令中的 Opcode 来对各种信号量赋值，进而实现对其他功能模块的操作。关键代码如下

```
always@(*) begin
        case(cur_state)
            sIF: begin
                n_state = sID;
            end
            sID: begin
                if (Opcode == j|| Opcode == jr || Opcode == halt || Opcode == jal)
                    n_state = sIF;
```

```
        else
            n_state = sEXE;
    end
            sEXE: begin
        if (Opcode == beq || Opcode == bne || Opcode == bltz) begin
            n_state = sIF;
        end
        else begin
            if (Opcode == sw || Opcode == lw) begin
                n_state = sMEM;
            end
            else begin
                n_state = sWB;
            end
        end
    end
    sMEM: begin
        if (Opcode == sw)
            n_state = sIF;
        else
            n_state = sWB;
    end
```

```
            sWB: begin
                n_state = 3'b000;
            end
            default: begin
                n_state = 3'b000;
            end
```

接着我们需要编写在 Basys3 实验板上显示的代码

这部分内容与单周期 cpu 设计的基本一致，在此仅简单描述思路。

首先因为系统的时钟频率远远高于人眼的视觉频率极限 ,于是我们需要特定的时钟频率来刷新扫描数码管，所以我们需要时钟分频模块，改变时钟频率。

其次我们需要通过按键来模拟 cpu 周期，每按下一次按键，将信号值取反，做为 CPU 时钟。值得注意的是防抖处理，我们对按键正信号或负信号进行取样，如果取样周期达到了预设周期，则输出正信号或负信号。

最后我们需要实现数据选择器和 三八译码电路，根据输入选择要显示的数据并在数码管上译码显示出来。这部分的代码在走马灯实验上已经完整地给出了。

六、仿真与测试

测试指令如下所示，以 jupyter notebook 为编写工具，编写的简单编译器会自动读取指令，并输出对应的二进制串。

地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)		16 进制数代码
0x00000000	addi $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010		48020002
0x00000008	or $3,$2,$1	010000	00010	00001	0001 1000 0000 0000		40411800
0x0000000C	sub $4,$3,$1	000001	00011	00001	0010 0000 0000 0000		04612000
0x00000010	and $5,$4,$2	010001	00100	00010	0100 1000 0000 0000		44824800
0x00000014	sll $5,$5,2	011000	00000	00101	0010 1000 1000 0000		60052880
0x00000018	beq $5,$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110		D0A1FFFE
0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000		E8000010
0x00000020	slt $8,$12,$1	100110	01100	00001	0100 0000 0000 0000		99814000
0x00000024	addi $13,$0,-2	000010	00000	01101	1111 1111 1111 1110		080DFFFE
0x00000028	slt $9,$8,$13	100110	01000	01101	0100 1000 0000 0000		990D4800
0x0000002C	sltiu $10,$9,2	100111	01001	01010	0000 0000 0000 0010		9d2A0002
0x00000030	sltiu $11,$10,0	100111	01010	01011	0000 0000 0000 0000		9d4B0000
0x00000034	addi $13,$13,1	000010	01101	01101	0000 0000 0000 0001		09AD0001
0x00000038	bltz $13,-2 (<0,转 34)	110110	01101	00000	1111 1111 1111 1110		D9A0FFFE
0x0000003C	j 0x000004C	111000	00000	00000	0000 0000 0001 0011		E0000013
0x00000040	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100		C0220004
0x00000044	lw $12,4($1)	110001	00001	01100	0000 0000 0000 0100		C42C0004
0x00000048	jr $31	111001	11111	00000	0000 0000 0000 0000		E7E00000
0x0000004C	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000
0x00000050							
0x00000054							

关键代码如下，并将相关代码放在代码文件夹中。

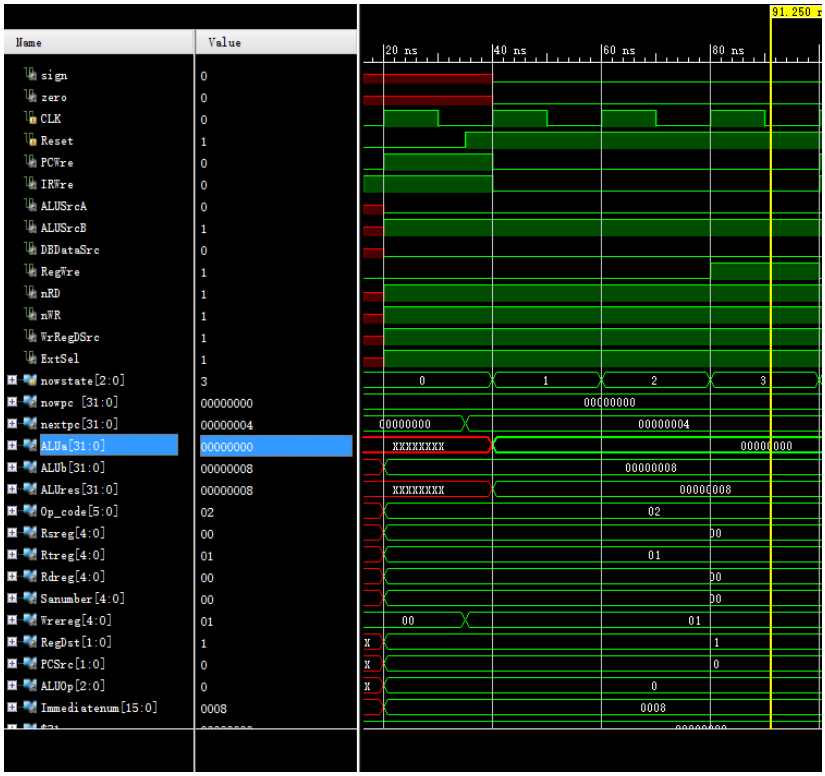
```
def jr(num):
    str = '111001' + ''.join(getbin(num[0],5) + '0' * 21)
    return litering_by_three(str)

def halt(num):
    return litering_by_three('1'* 5 + '0' * 27)
# map the inputs to the function blocks
options = {'addi': addi,
           'ori' : ori,
           'or' : OR,
           'sub' : sub,
           'and' : And,
           'sll' : sll,
           'beq' : beq,
           'jal' : jal,
           'slt':slt,
           'sltiu': sltiu,
           'bltz':bltz,
           'j':j,
           'sw':sw,
           'lw':lw,
           'jr':jr,
           'halt':halt
}
```

```
In [42]: f = open('cpu-test.txt', 'r', encoding="utf8")
         #print(sys.getdefaultencoding())
         result = f.readlines()
         sys.stdout = open("result.txt", "w")
         for i in result:
             #print(i)
             r = re.findall('-*\d+\w*',i)
             command = re.findall('([^\s]+)',i)
             #print(command)
             #print(r)
             if(command[0] != 'jal' and command[0] != 'j'):
                 num = list(map(int,r))
                 #print(type(num[1]))
                 print(options[command[0]](num))
```

以下表格中除了 operationcode 外，其他数字都是 16 进制表示，而 operationcode 用二进制标志，括号内的数字是 16 进制。（注：鉴于前七条指令中已经包含了三种指令，

因此 basys3 板上的测试情况仅以前七条指令为例）

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000000	addi $1,$0,8	000010	00000	00001	0000 0000 0000 1000	= 08010008



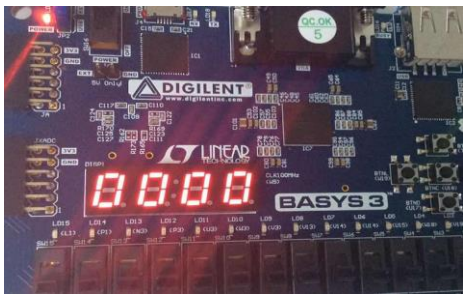
符号	数值	备注
OP_code	000010（02）	Addi，为 I 型指令
Nowpc	00000000	
Nextpc	00000004	
Alua	00000000	
Alub	00000008	
ALUres	00000008	Alu 最终运算结果是 8
Rsreg	00	
Rtreg	01	

Immediatenum	0008	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 0 和 4

取指令阶段

rs 寄存器地址和其中数据

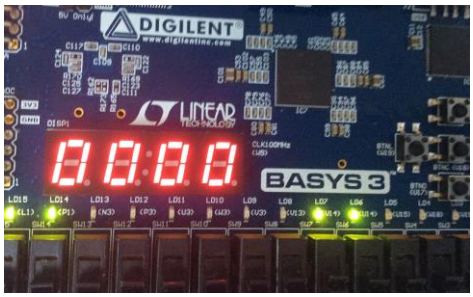


rt 寄存器地址和其中数据

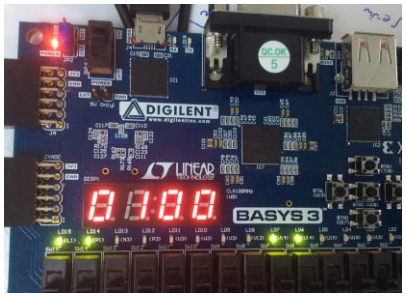
立即数和写回寄存器的值

接着在指令译码阶段

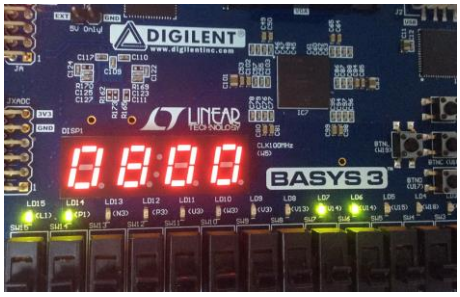
Rs 寄存器地址和数据的值是



Rt 寄存器地址和数据的值是



alu 输出结果和数据总线的值是



在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是

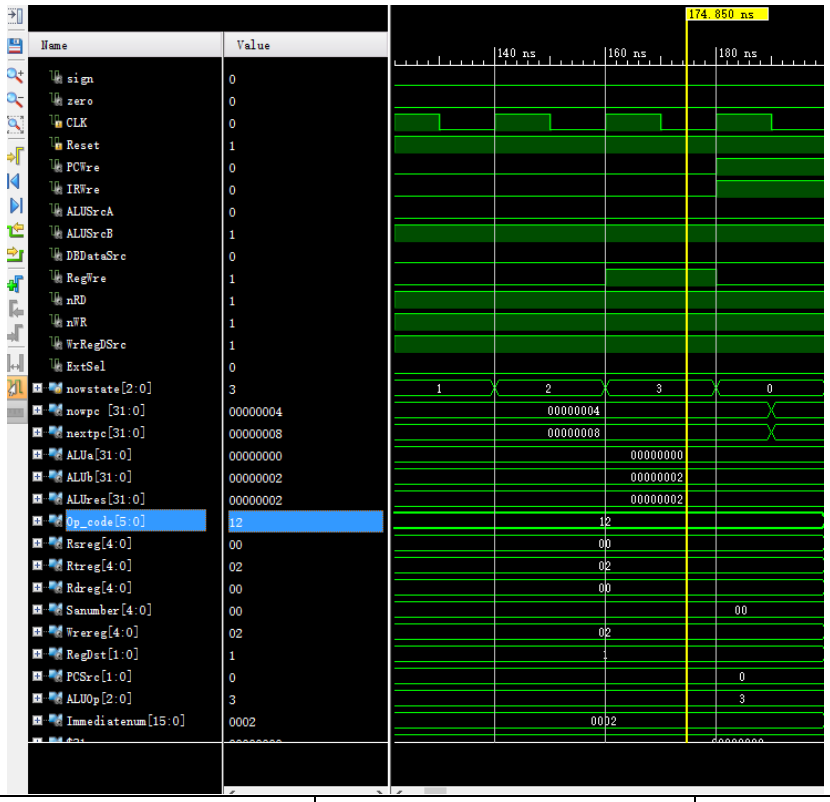
在结果写回阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值是

alu 输出结果和数据总线的值与上一阶段一样

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	48020002



符号	数值	备注
OP_code	010010（12）	ori，为 I 型指令
Nowpc	00000004	
Nextpc	00000008	
Alua	00000000	
Alub	00000002	
ALUres	00000002	Alu 最终运算结果是 2
Rsreg	00	
Rtreg	02	
Immediatenum	0002	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值

RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 4 和 8

取指令阶段

rs 寄存器地址和其中数据与上一阶段一样

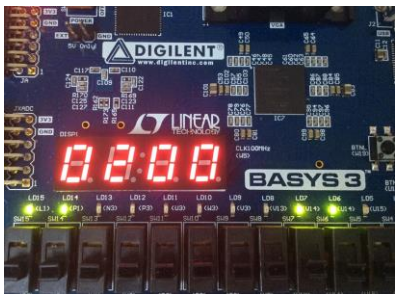
rt 寄存器地址和其中数据与上一阶段一样

立即数和写回寄存器的值与上一阶段一样

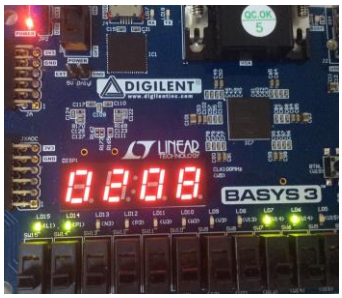
接着在指令译码阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值是



alu 输出结果和数据总线的值是



在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是

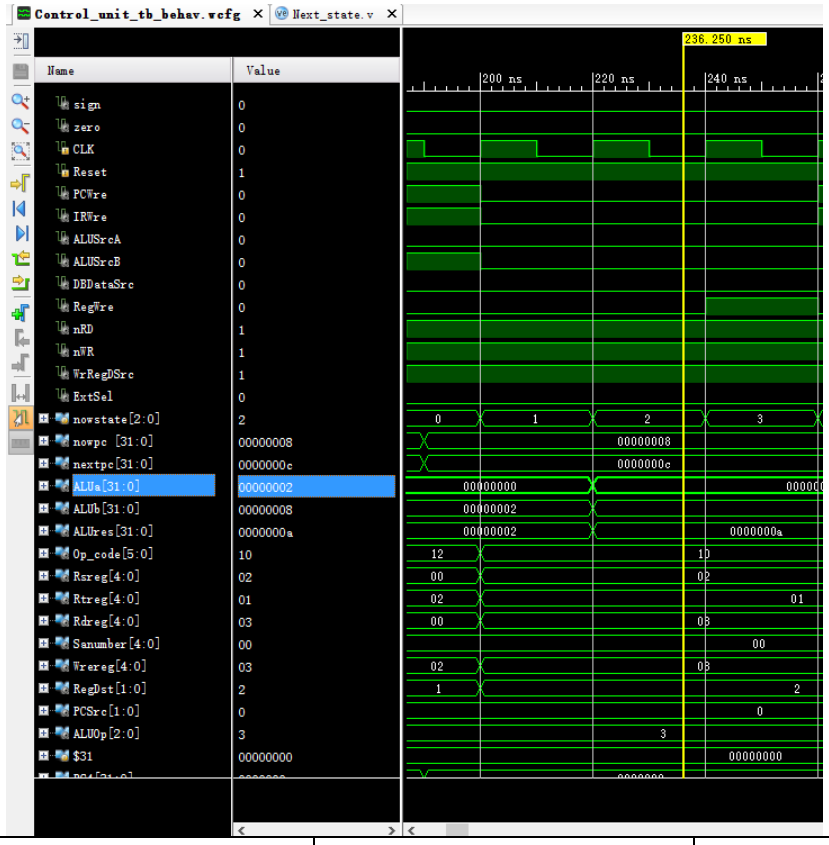
在结果写回阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值是

alu 输出结果和数据总线的值与上一阶段一样

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000008	or $3,$2,$1	010000	00010	00001	0001 1000 0000 0000	40411800



符号	数值	备注
OP_code	010000（10）	or，为 R 型指令
Nowpc	00000008	
Nextpc	0000000c	
Alua	00000002	
Alub	00000008	
ALUres	0000000a	Alu 最终运算结果是 12
Rsreg	02	
Rtreg	01	
Rdreg	03	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值

RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 8 和 12

取指令阶段

rs 寄存器地址和其中数据与上一阶段一样

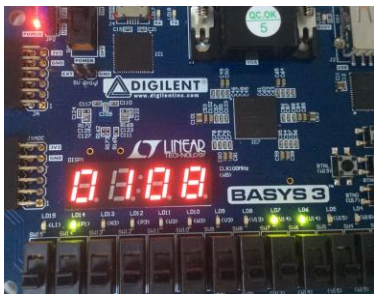
rt 寄存器地址和其中数据与上一阶段一样

立即数和写回寄存器的值与上一阶段一样

接着在指令译码阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值是



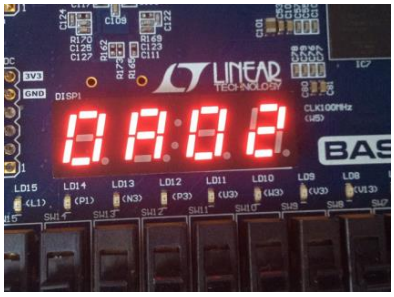
alu 输出结果和数据总线的值与上阶段一样

在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是



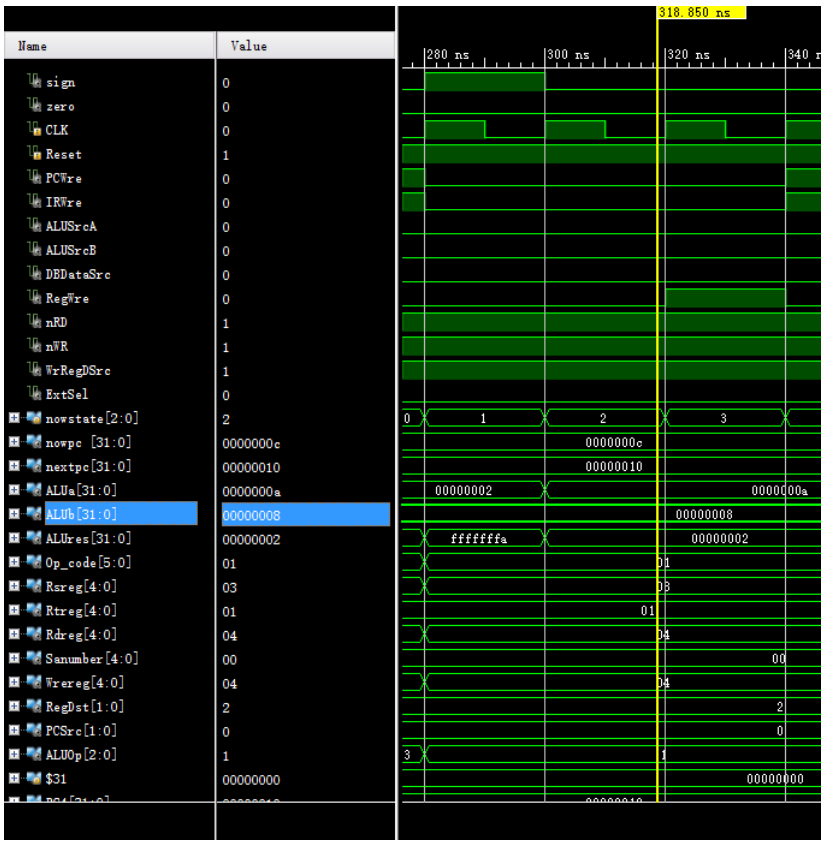
在结果写回阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是

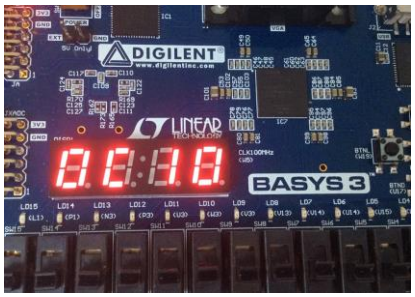
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x0000000C	sub $4,$3,$1	000001	00011	00001	0010 0000 0000 0000	04812000



符号	数值	备注
OP_code	000001（1）	sub，为 R 型指令
Nowpc	0000000c	
Nextpc	00000010	
Alua	0000000a	
Alub	00000008	
ALUres	00000002	Alu 最终运算结果是 2
Rsreg	03	
Rtreg	01	
Rdreg	04	
PCSrc	0	正常的 PC 加 4 作为下个 PC

		值
RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 12 和 16



取指令阶段

rs 寄存器地址和其中数据与上一阶段一样

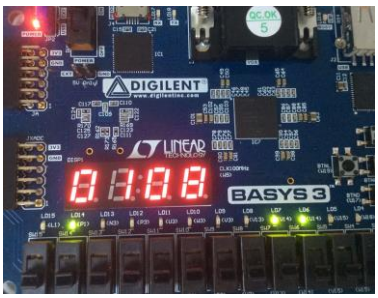
rt 寄存器地址和其中数据与上一阶段一样

立即数和写回寄存器的值与上一阶段一样

接着在指令译码阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值



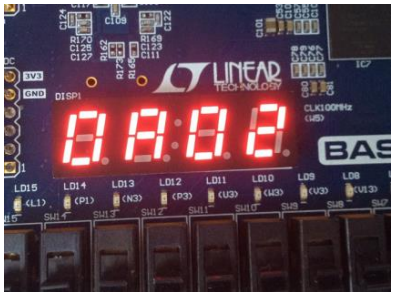
alu 输出结果和数据总线的值与上阶段一样

在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是



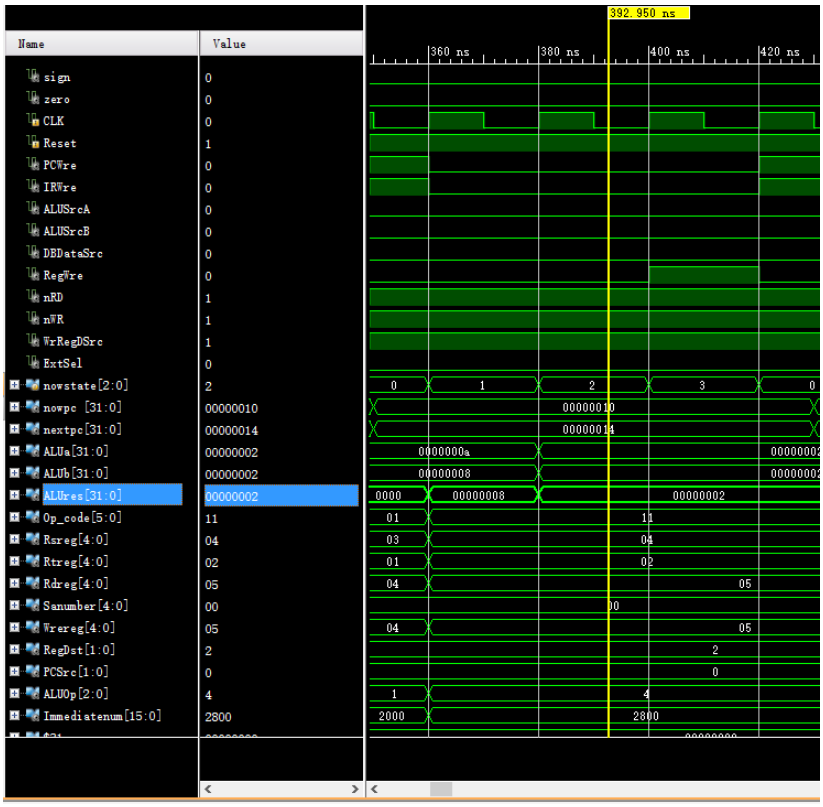
在结果写回阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值

地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)		16 进制数代码
0x00000010	and $5,$4,$2	010001	00100	00010	0100 1000 0000 0000		44824800



符号	数值	备注
OP_code	010001（11）	and，为 R 型指令
Nowpc	000000010	
Nextpc	000000014	
Alua	00000002	
Alub	00000002	
ALUres	00000002	Alu 最终运算结果是 2
Rsreg	04	
Rtreg	02	
Rdreg	05	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值

RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 16 和 20

取指令阶段

rs 寄存器地址和其中数据与上一阶段一样

rt 寄存器地址和其中数据与上一阶段一样

立即数和写回寄存器的值与上一阶段一样

接着在指令译码阶段

Rs 寄存器地址和数据的值是

Rt 寄存器地址和数据的值是

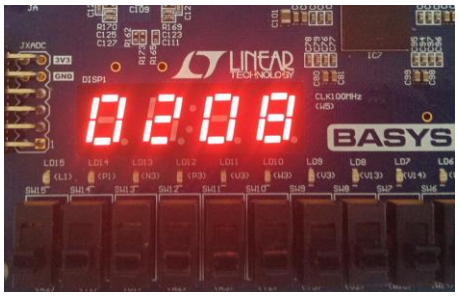
alu 输出结果和数据总线的值是

在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是

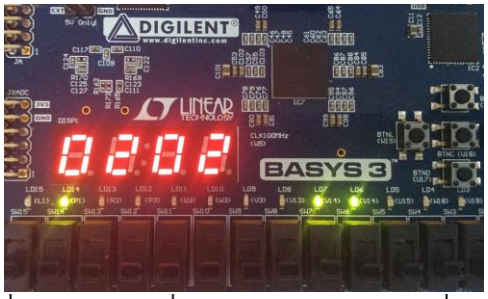


在结果写回阶段

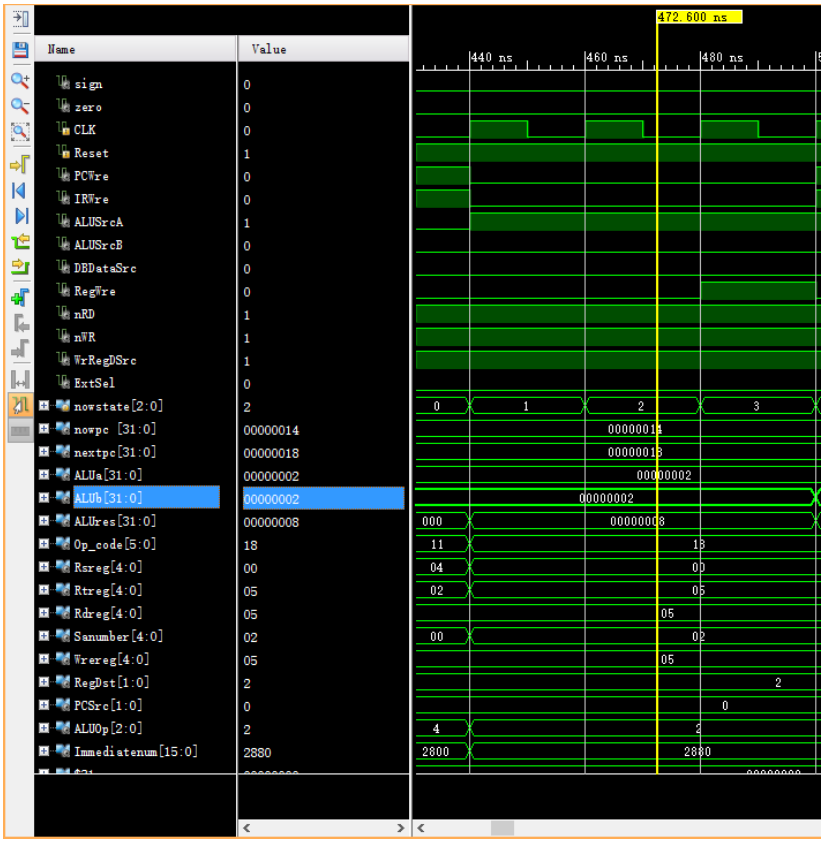
Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是



地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000014	sll $5,$5,2	011000	00000	00101	0010 1000 1000 0000	60052880



符号	数值	备注
OP_code	011000（11）	sll，为 I 型指令
Nowpc	00000014	
Nextpc	00000018	
Alua	00000002	
Alub	00000002	

ALUres	00000008	Alu 最终运算结果是 8（2 左移两位）
Rtreg	05	
Rdreg	05	
Sanumber	2	移位数是 2
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 20 和 24

取指令阶段

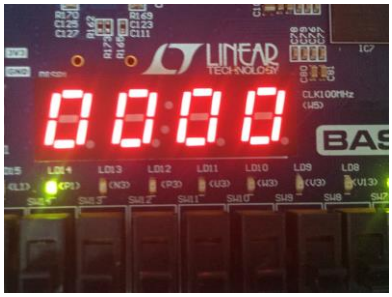
rs 寄存器地址和其中数据与上一阶段一样

rt 寄存器地址和其中数据与上一阶段一样

立即数和写回寄存器的值与上一阶段一样

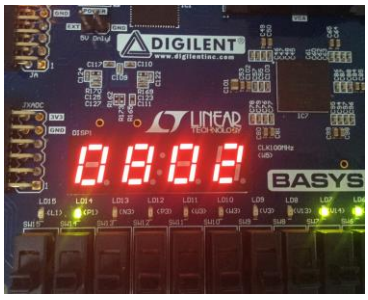
接着在指令译码阶段

Rs 寄存器地址和数据的值是



Rt 寄存器地址和数据的值是

alu 输出结果和数据总线的值是



在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

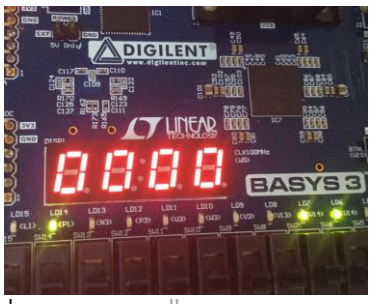
alu 输出结果和数据总线的值是

在结果写回阶段

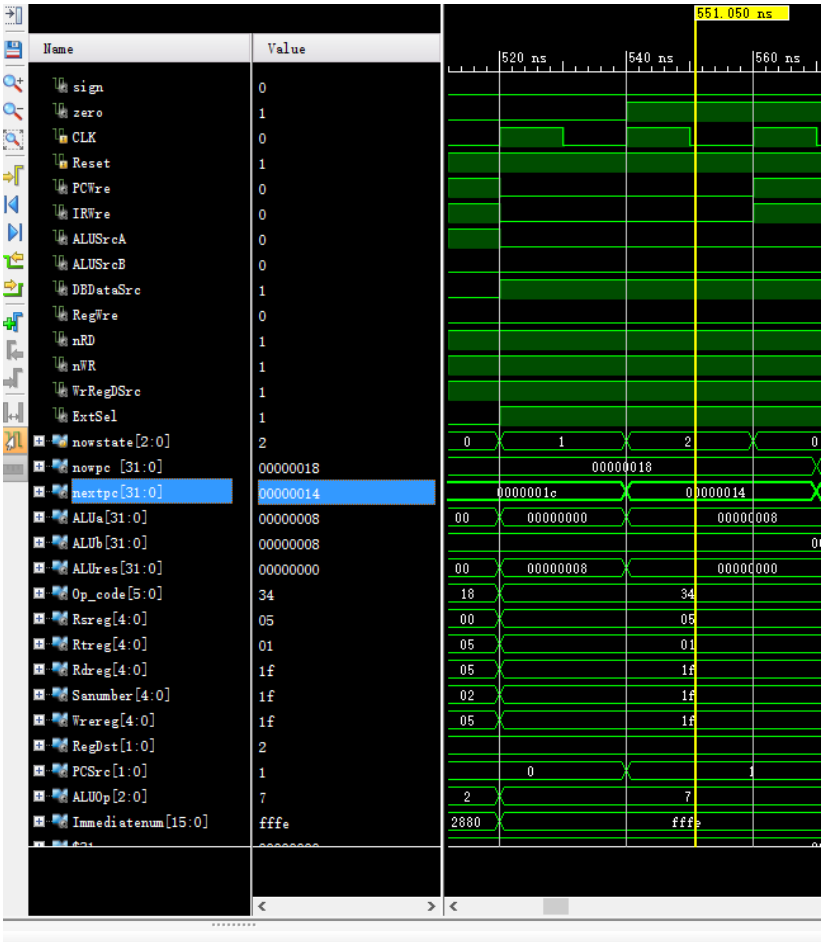
Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是



地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000018	beq $5,$1,-2(=转 14)	110100	00101	00001	1111 1111 1111 1110	D0A1FFFE



符号	数值	备注
OP_code	011000（11）	Beq

Nowpc	00000018	
Nextpc	00000014	执行跳转指令
Alua	00000008	
Alub	00000008	
ALUres	00000000	五号寄存器和 1 号寄存器的值都是 8，最终 alu 计算结果是 0
Rsreg	05	
Rtreg	01	
PCSrc	1	根据立即数确定下个 PC 值
RegWre	一直是 0	不涉及写回阶段
Nowstate	0,1,2	涉及 IF,ID,EXE 阶段，依次读取指令，指令译码，运算

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 24 和 20

取指令阶段

rs 寄存器地址和其中数据

rt 寄存器地址和其中数据与上一阶段一样

立即数和写回寄存器的值与上一阶段一样

接着在指令译码阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值是

alu 输出结果和数据总线的值与上阶段一样

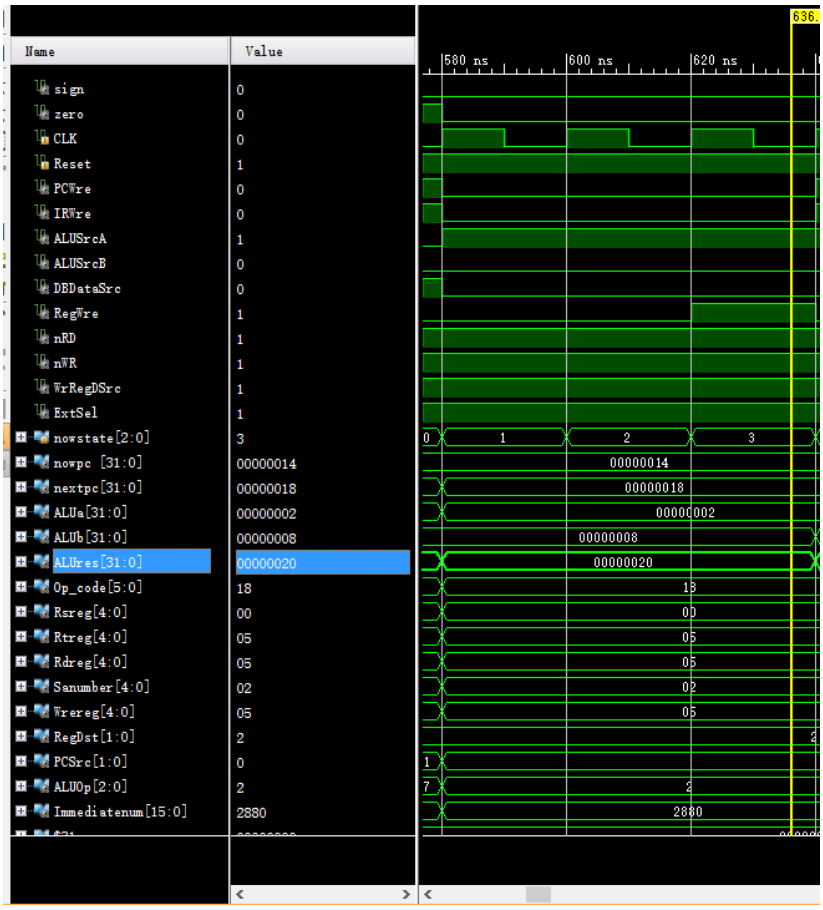
在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值与上一阶段一样

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000014	sll $5,$5,2	011000	00000	00101	0010 1000 1000 0000	60052880



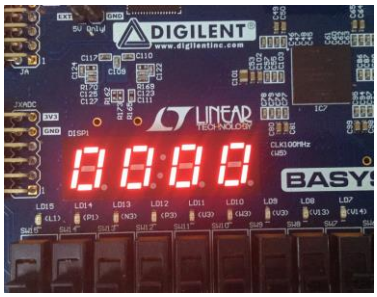
符号	数值	备注
OP_code	011000（11）	sll，为 I 型指令
Nowpc	00000014	
Nextpc	00000018	
Alua	00000002	
Alub	00000008	
ALUres	00000020	Alu 最终运算结果是 32（8 左移两位）
Rtreg	05	
Rdreg	05	
Sanumber	2	移位数是 2

PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

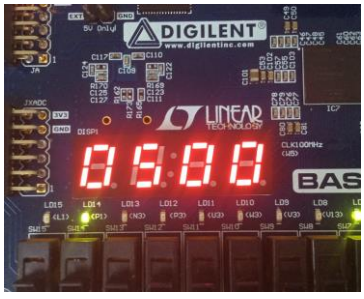
在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 20 和 24

取指令阶段

rs 寄存器地址和其中数据



rt 寄存器地址和其中数据



立即数和写回寄存器的值

接着在指令译码阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值

在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值与上一阶段一样

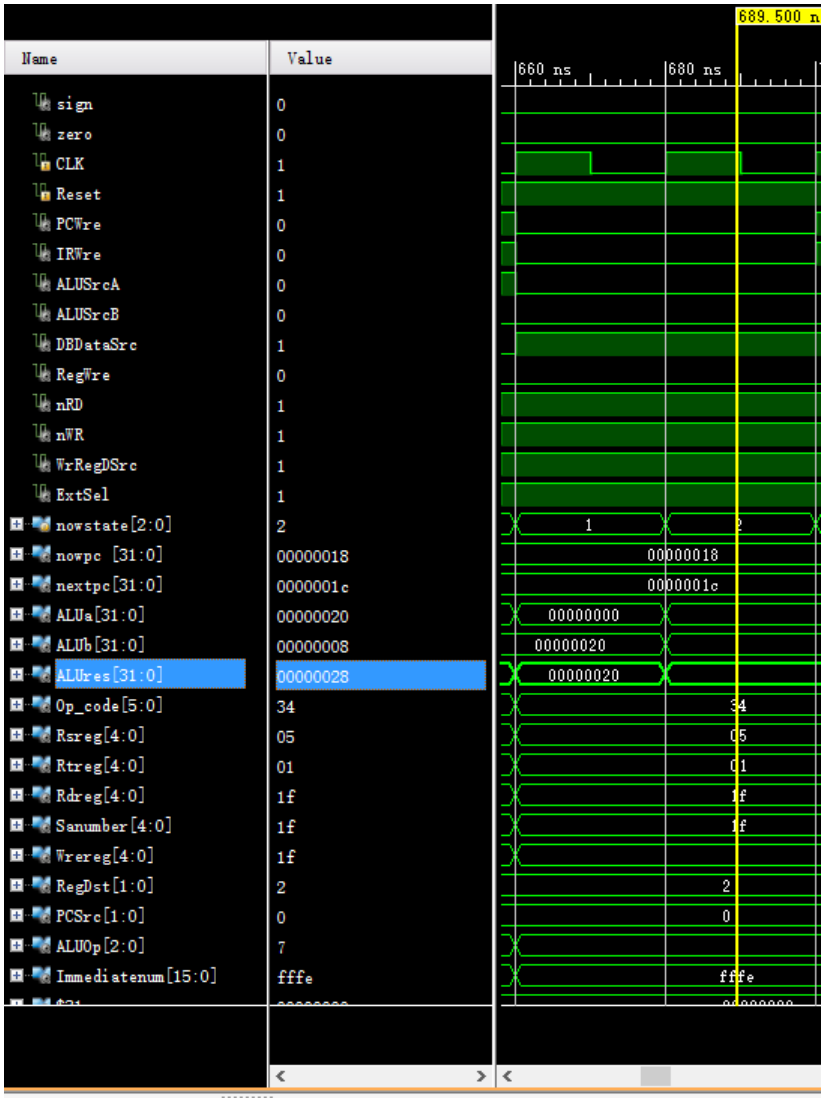
在结果写回阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值

alu 输出结果和数据总线的值是

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000018	beq $5,$1,-2(=转 14)	110100	00101	00001	1111 1111 1111 1110	D0A1FFFE



符号	数值	备注
OP_code	011000（11）	Beq，为 I 型指令
Nowpc	00000018	

Nextpc	0000001c	执行跳转指令
Alua	00000020	
Alub	00000008	
ALUres	00000028	五号寄存器和 1 号寄存器的值分别是 32 和 8，最终 alu 计算结果不为 0
Rsreg	05	
Rtreg	01	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	一直是 0	不涉及写回阶段
Nowstate	0,1,2	涉及 IF,ID,EXE 阶段，依次读取指令，指令译码，运算

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 24 和 28

取指令阶段

rs 寄存器地址和其中数据与上一阶段一样

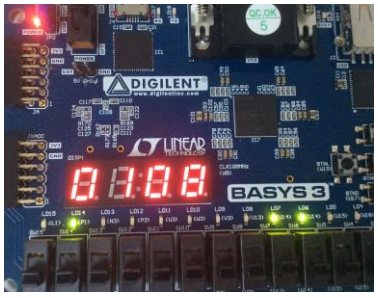
rt 寄存器地址和其中数据

立即数和写回寄存器的值：

接着在指令译码阶段

Rs 寄存器地址和数据的值

Rt 寄存器地址和数据的值是



alu 输出结果和数据总线的值

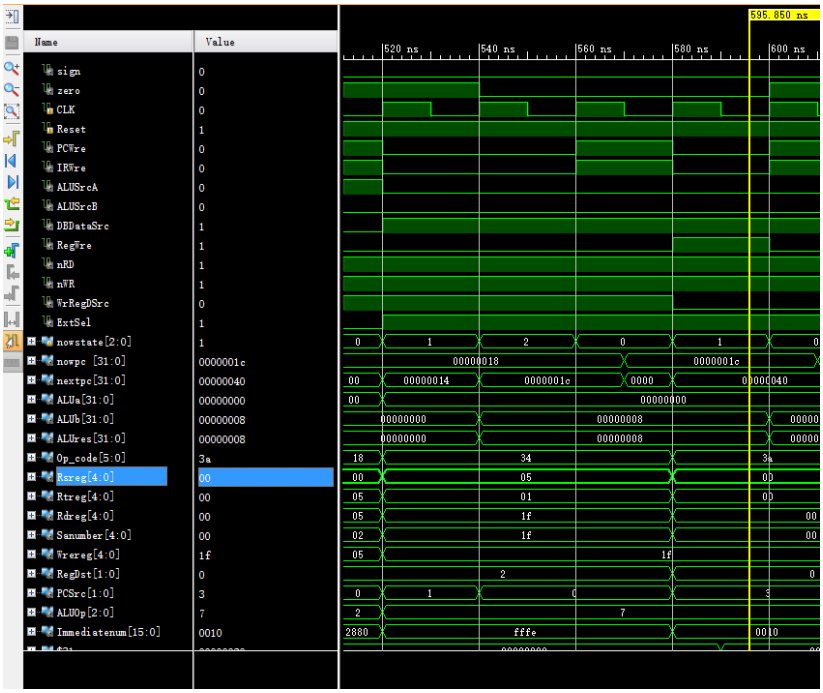
在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是与上一阶段一样

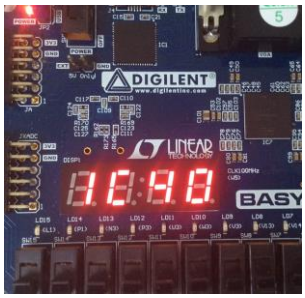
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000	E8000010



符号	数值	备注
OP_code	010000（10）	Jal，为 J 型指令
Nowpc	0000001c	
Nextpc	00000040	
Rsreg	02	
Rtreg	01	
Rdreg	03	
PCSrc	3	根据地址，做出 PC 的跳转

RegWre	0 到 1	在阶段 1 时为 1，把 PC+4 的值写入$31 中，作为保存
Nowstate	0,1,	涉及 IF,ID,依次读取指令，指令译码，

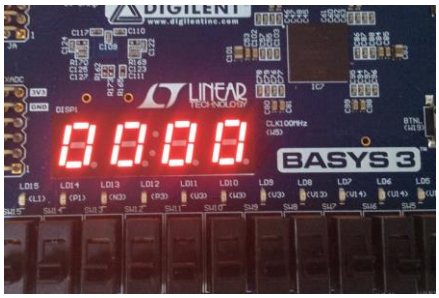
在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 12 和 16



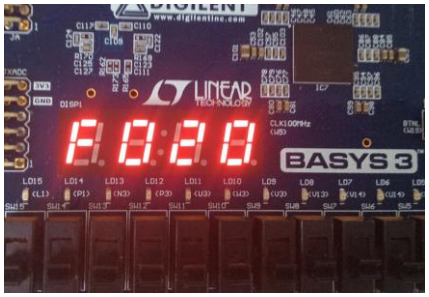
取指令阶段

rs 寄存器地址和其中数据与上阶段一样

rt 寄存器地址和其中数据



立即数和写回寄存器的值

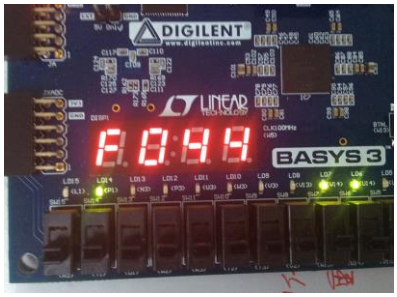


接着在指令译码阶段

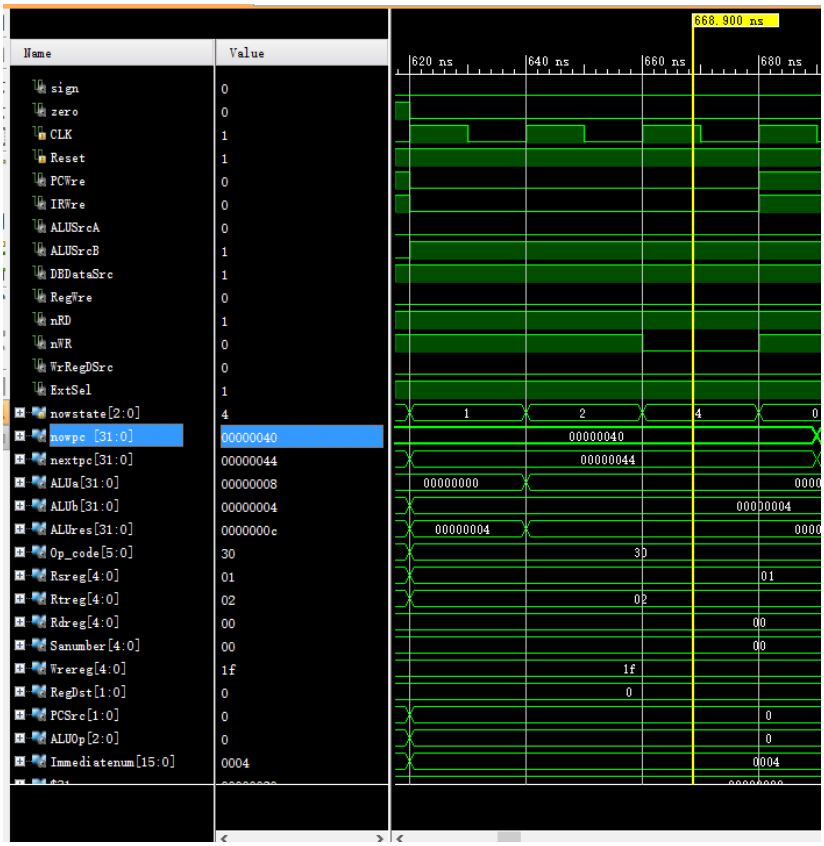
Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是



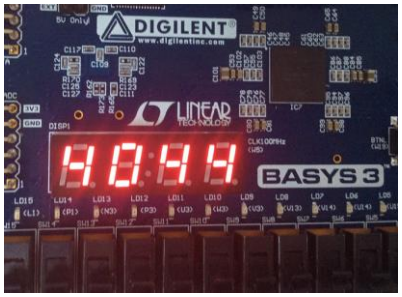
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000040	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	C0220004



符号	数值	备注
OP_code	110000（30）	sw，为 I 型指令
Nowpc	00000040	
Nextpc	00000044	
Alua	00000008	

Alub	00000004	
ALUres	0000000c	将 rt 中的值 2 写入 ram【12】及之后的字节
Rsreg	01	
Rtreg	02	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	一直是 0	不涉及写回阶段
Nowstate	0,1,2,4	涉及 IF,ID,EXE,MEM 阶段，依次读取指令，指令译码，运算，访问内存

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 64 和 68



取指令阶段

rs 寄存器地址和其中数据

rt 寄存器地址和其中数据

立即数和写回寄存器的值

接着在指令译码阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值与上一阶段一样

在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值是

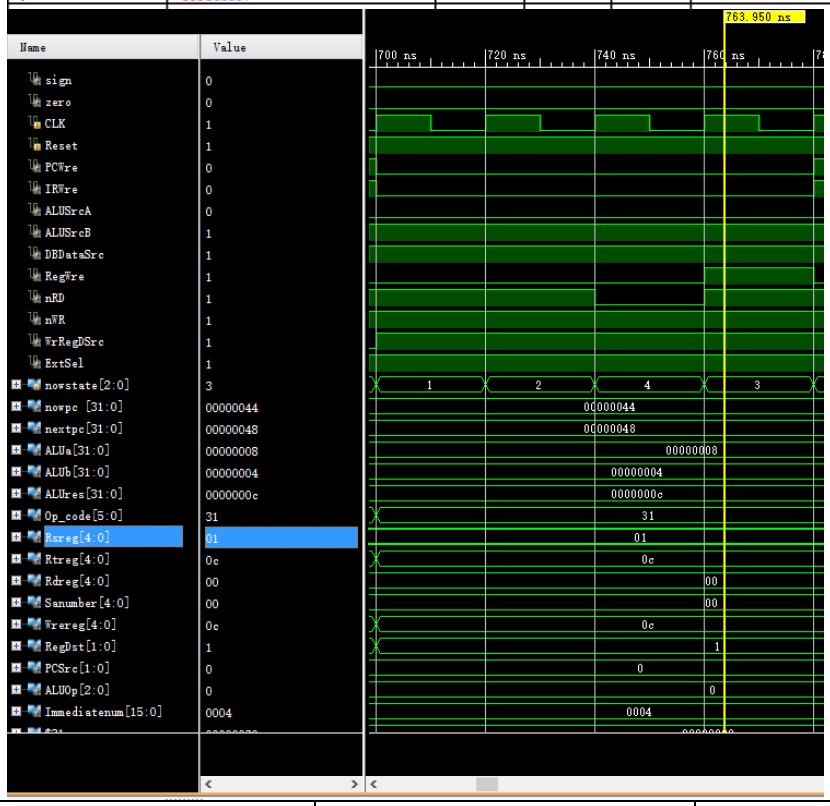
在访问内存阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值与上阶段一样

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
0x00000044	lw $12,4($1)	110001	00001	01100	0000 0000 0000 0100	C42C0004



符号	数值	备注
OP_code	110001（31）	lw，为 I 型指令
Nowpc	00000044	
Nextpc	00000048	
Alua	00000008	
Alub	00000004	
ALUres	0000000c	计算地址的值是 12
Rsreg	01	
Rtreg	0c	

PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	先 0 后 1	阶段 3 进行写回
Nowstate	0,1,2,4,3	涉及 IF,ID,EXE,MEM,WB 阶段 ,依次读取指令 ,指令译码，运算，访问内存，结果写回

在 Basys3 板上，此时 currentPC 和 nextPC 的值分别是 68 和 72

取指令阶段

rs 寄存器地址和其中数据

与上一阶段一样

rt 寄存器地址和其中数据

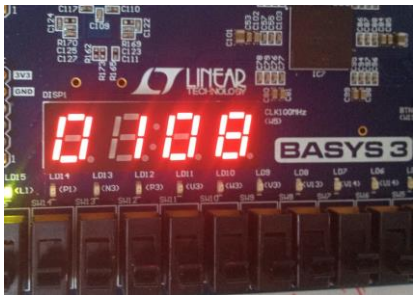
与上一阶段一样

立即数和写回寄存器的值

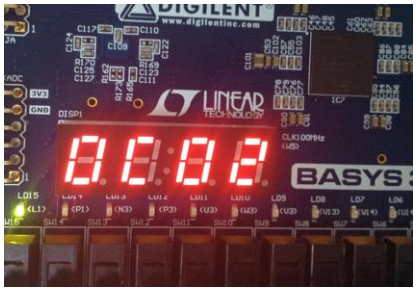
与上一阶段一样

接着在指令译码阶段

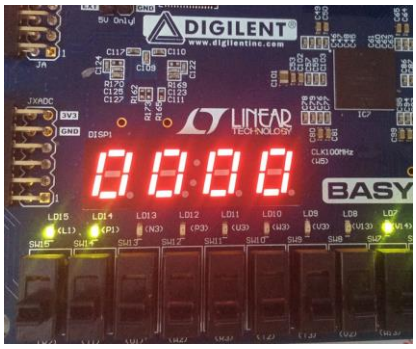
Rs 寄存器地址和数据的值是



Rt 寄存器地址和数据的值是



alu 输出结果和数据总线的值是

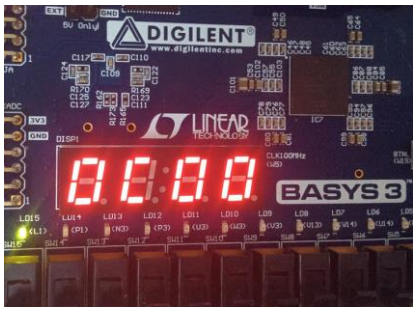


在指令执行阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值是与上一阶段一样

alu 输出结果和数据总线的值是



在内存访问阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值与上一阶段一样

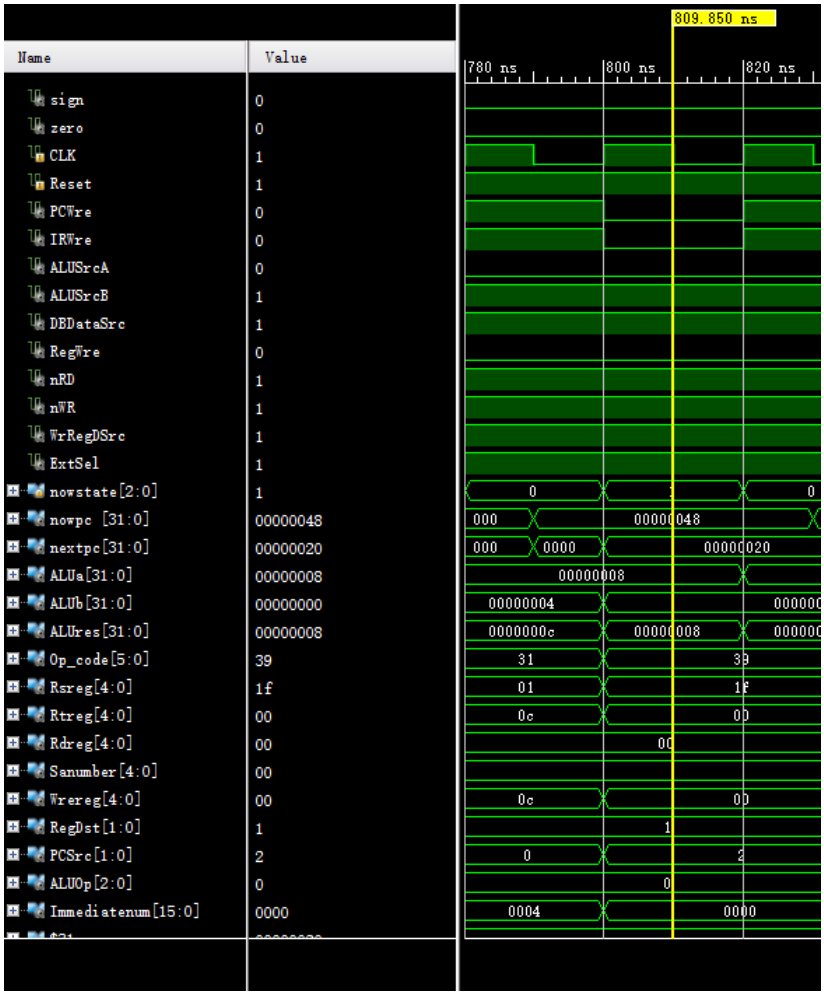
在结果写回阶段

Rs 寄存器地址和数据的值与上一阶段一样

Rt 寄存器地址和数据的值与上一阶段一样

alu 输出结果和数据总线的值与上一阶段一样

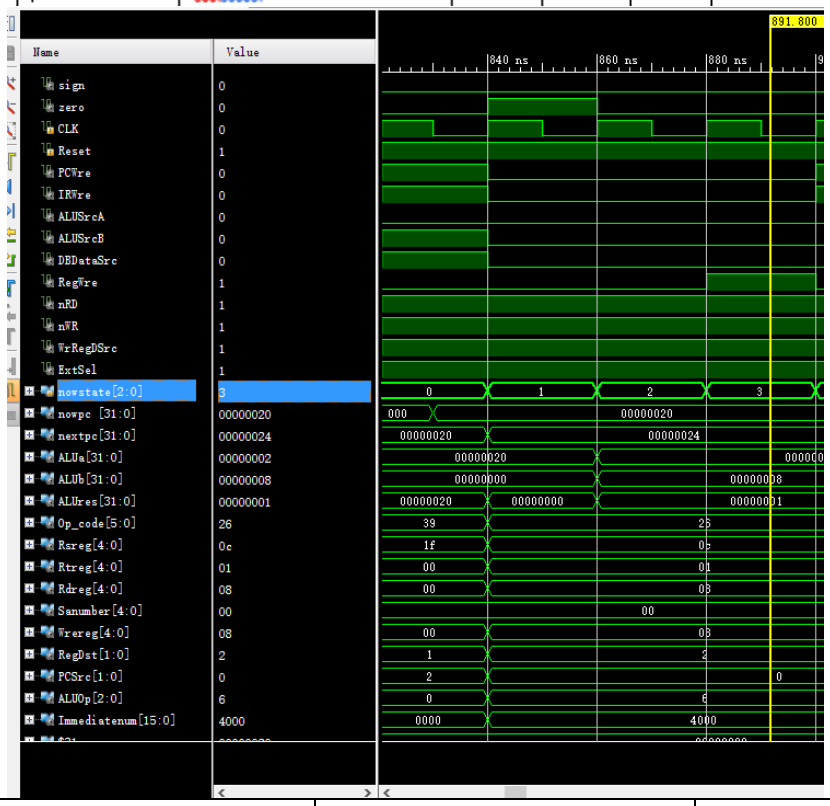
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000048	jr $31	111001	11111	00000	0000 0000 0000 0000	E7E00000



符号	数值	备注

OP_code	110001（31）	Jr 指令
Nowpc	00000048	
Nextpc	00000020	31 号寄存器中的值是 20
Rsreg	01	
Rtreg	0c	
PCSrc	2	根据寄存器的值，PC 做出跳转
RegWre	一直是 0	
Nowstate	0,1	涉及 IF,ID 阶段，依次读取指令，指令译码

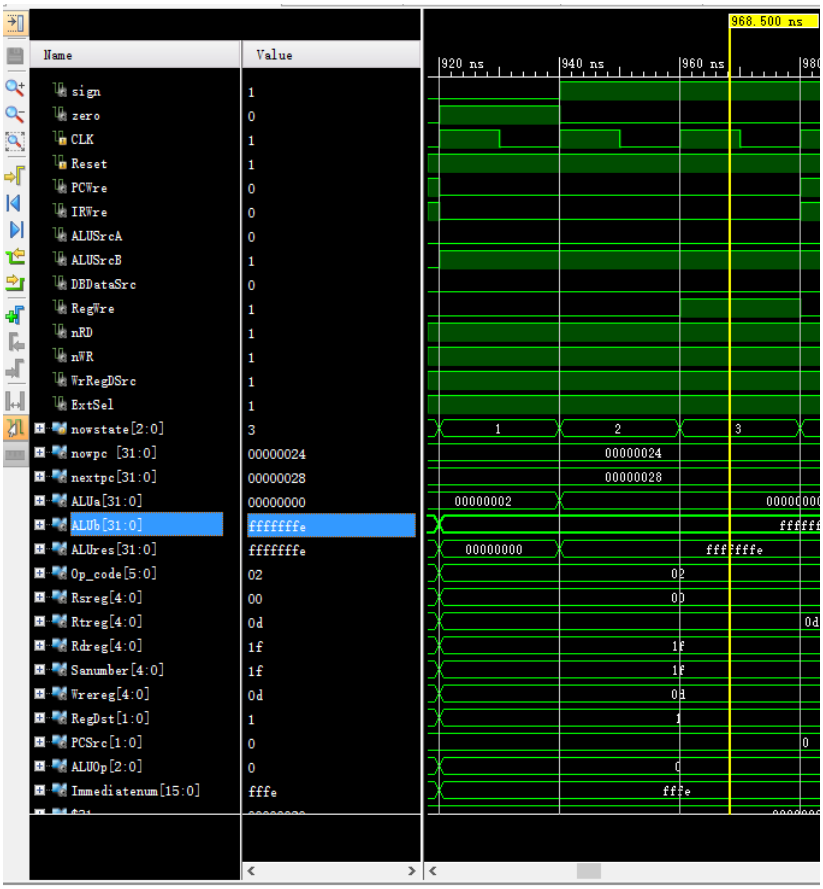
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000020	slt $8,$12,$1	100110	01100	00001	0100 0000 0000 0000	99814000



符号	数值	备注

OP_code	100110（26）	slt，为 R 型指令
Nowpc	000000020	
Nextpc	000000024	
Alua	00000002	
Alub	00000008	
ALUres	00000001	Alu 最终运算结果是 1(因为 12 号寄存器的值是 2， 1 号寄存器的值是 8， 2<8，所以结果为 1，写入 rd 寄存器
Rsreg	0c	
Rtreg	01	
Rdreg	08	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令，指令译码，运算和结果写回

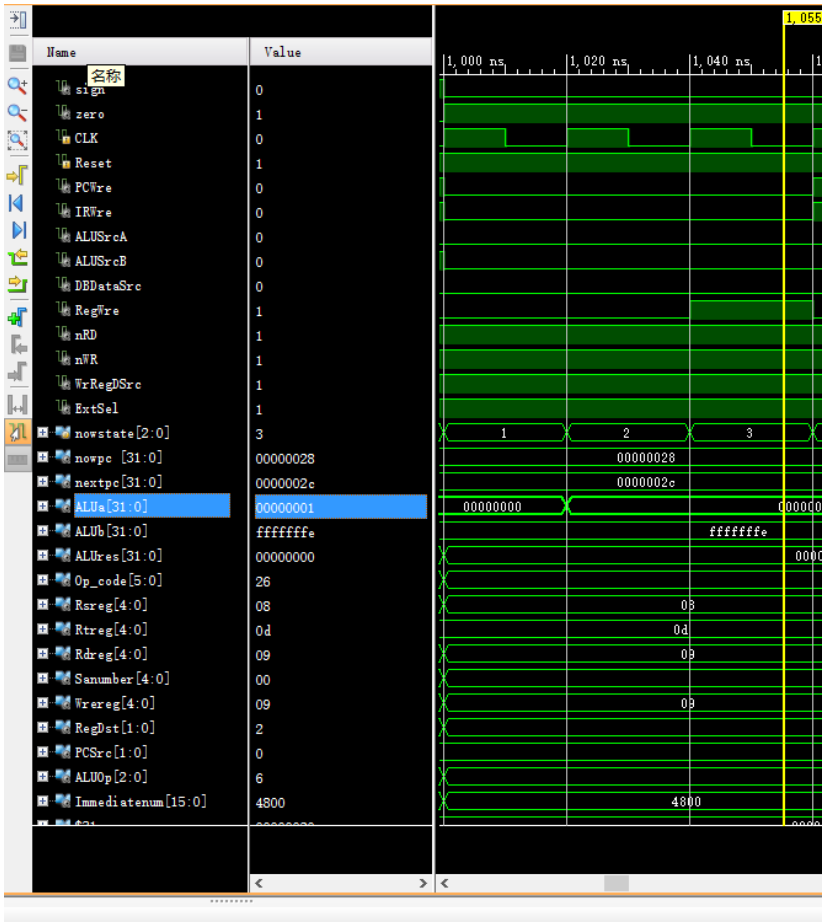
地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)		16 进制数代码
0x00000024	addi $13,$0,-2	000010	00000	01101	1111 1111 1111 1110		080DFFFE



符号	数值	备注
OP_code	000010（2）	addi 为 I 型指令
Nowpc	00000024	
Nextpc	00000028	
Alua	00000000	
Alub	ffffffe	
ALUres	ffffffe	最终值是-2
Rsreg	01	
Rtreg	0c	
Immediatenum	fffe	-2 的补码表示
PCSrc	0	正常的 PC 加 4 作为下个 PC

		值
RegWre	先 0 后 1	阶段 3 进行写回
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算，结果写回

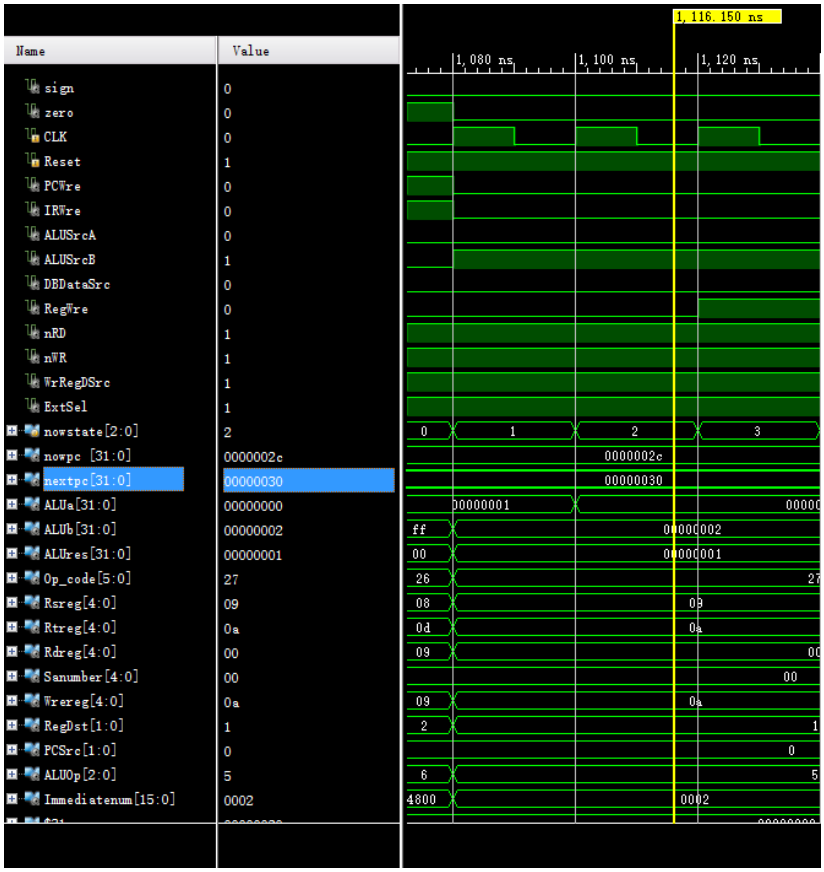
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000028	slt $9,$8,$13	100110	01000	01101	0100 1000 0000 0000	990D4800



符号	数值	备注
OP_code	100110（26）	slt，为 R 型指令
Nowpc	000000028	

Nextpc	00000002c	
Alua	00000001	
Alub	fffffffe	
ALUres	0	Alu 最终运算结果是 0,因为 rs 寄存器中的值 1 大于 rt 寄存器中的值-1
Rsreg	0,8	
Rtreg	0d	
Rdreg	09	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	0 到 1	在阶段 3 时为 1 ,控制寄存器写入
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算和结果写回

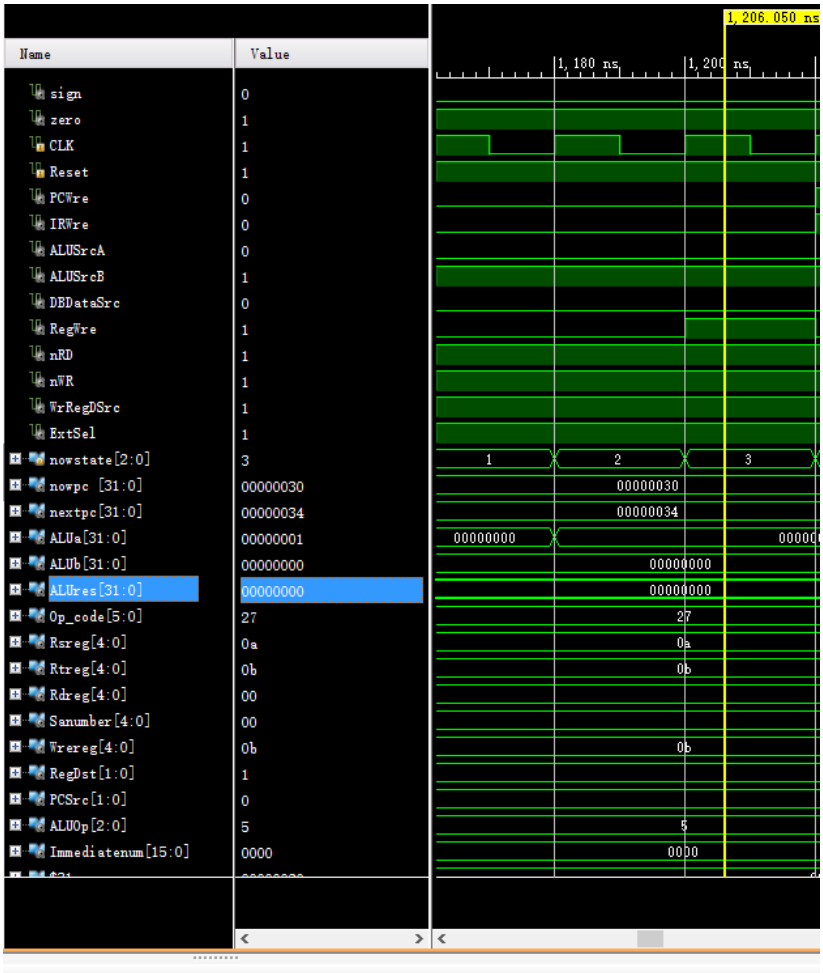
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x0000002C	sltiu $10,$9,2	100110	01001	01010	0000 0000 0000 0010	992A0002



符号	数值	备注
OP_code	100111（26）	sltiu 为 I 型指令
Nowpc	0000002c	
Nextpc	00000030	
Alua	00000000	
Alub	00000002	
ALUres	00000001	最终值是 1，因为 rs 寄存器中的值小于立即数，1 写入 rt、12 号寄存器
Rsreg	09	
Rtreg	0a	

Immediatenum	0002	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	先 0 后 1	阶段 3 进行写回
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 指令译码 运算，结果写回

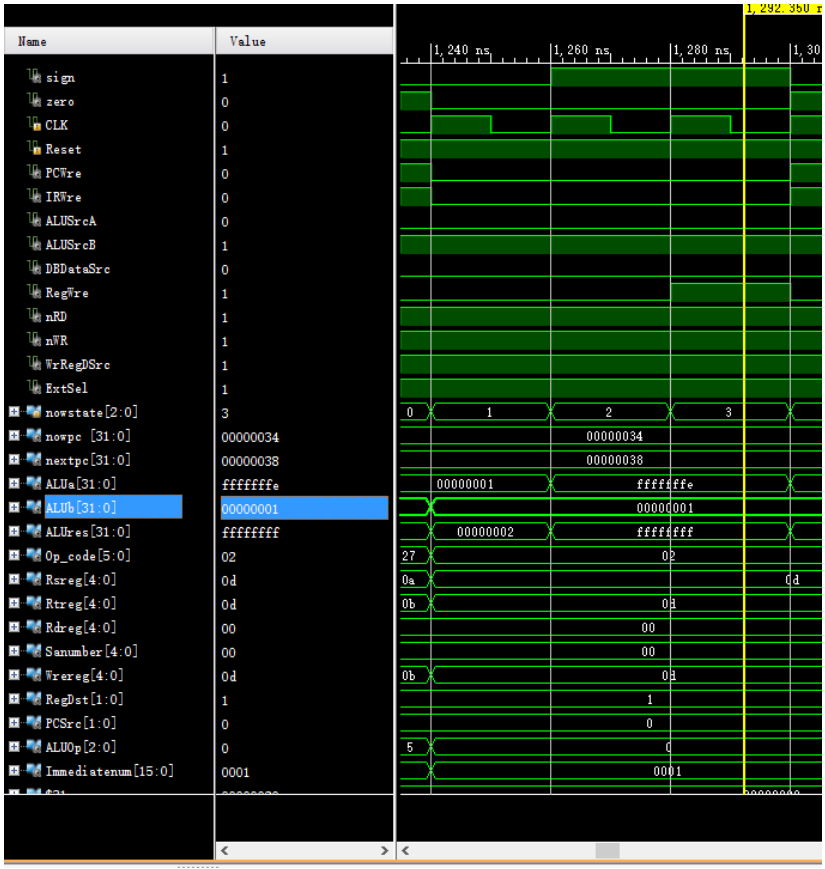
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000030	sltiu $11,$10,0	100110	01010	01011	0000 0000 0000 0000	994B0000



符号	数值	备注

OP_code	100111（26）	sltiu 为 I 型指令
Nowpc	0000002c	
Nextpc	00000030	
Alua	00000001	
Alub	00000000	
ALUres	00000000	最终值是 0，因为 rs 寄存器中的值大于于立即数，1 写入入 rt、11 号寄存器
Rsreg	0a	
Rtreg	0b	
Immediatenum	0000	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	先 0 后 1	阶段 3 进行写回
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算，结果写回

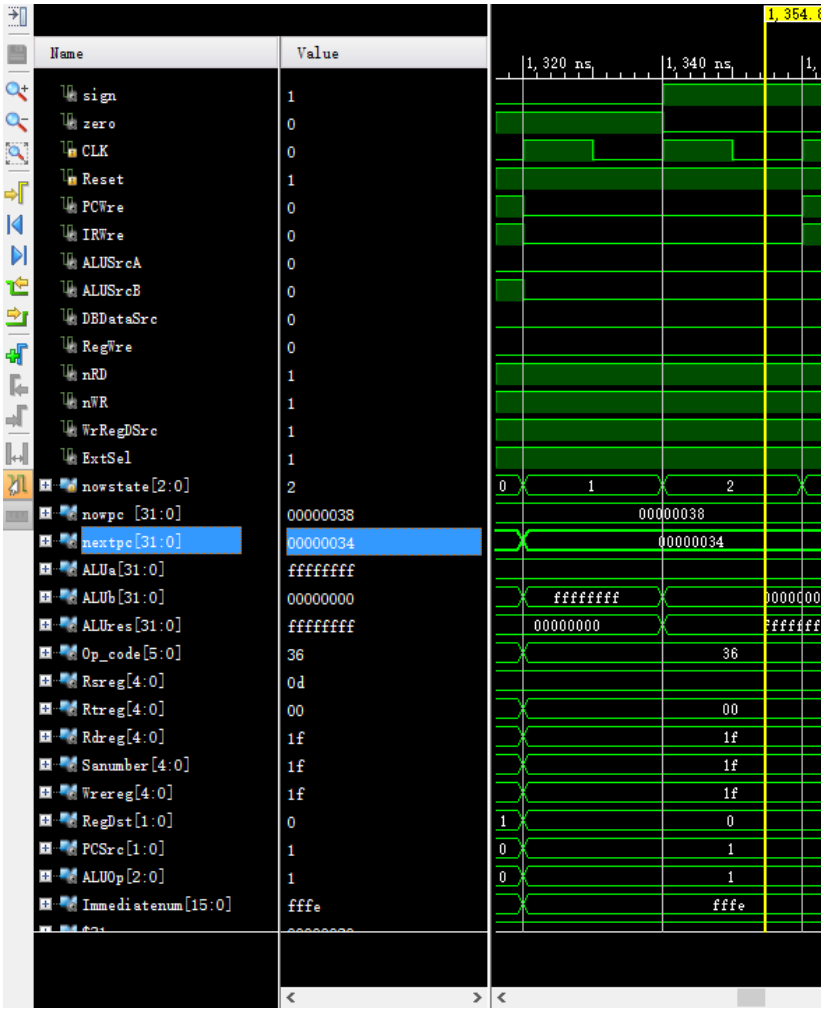
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000034	addi $13,$13,1	000010	01101	01101	0000 0000 0000 0001	09AD0001



符号	数值	备注
OP_code	000010（2）	addi 为 I 型指令
Nowpc	00000034	
Nextpc	00000038	
Alua	ffffffe	
Alub	00000001	
ALUres	fffffff	最终值是-1
Rsreg	01	
Rtreg	0c	
Immediatenum	fffe	-2 的补码表示
PCSrc	0	正常的 PC 加 4 作为下个 PC

		值
RegWre	先 0 后 1	阶段 3 进行写回
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算，结果写回

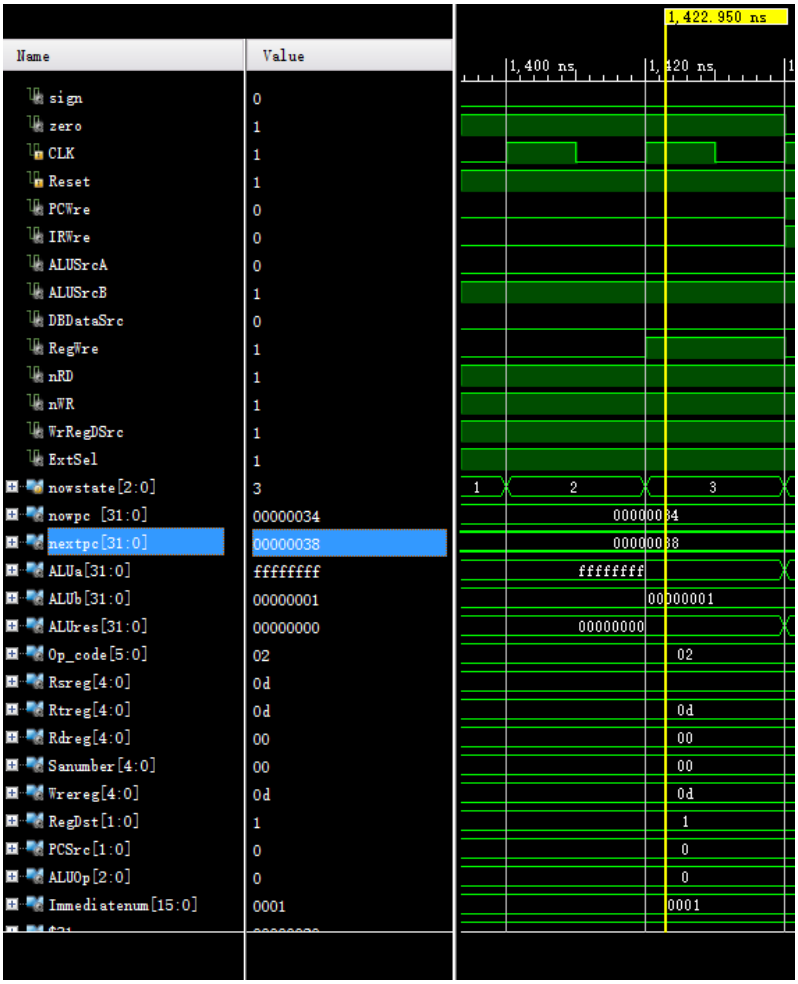
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000038	bltz $13,-2 (<0,转 34)	110110	01101	00000	1111 1111 1111 1110	D9A0FFFE



符号	数值	备注
OP_code	110110（36）	Bltz，为 I 型指令

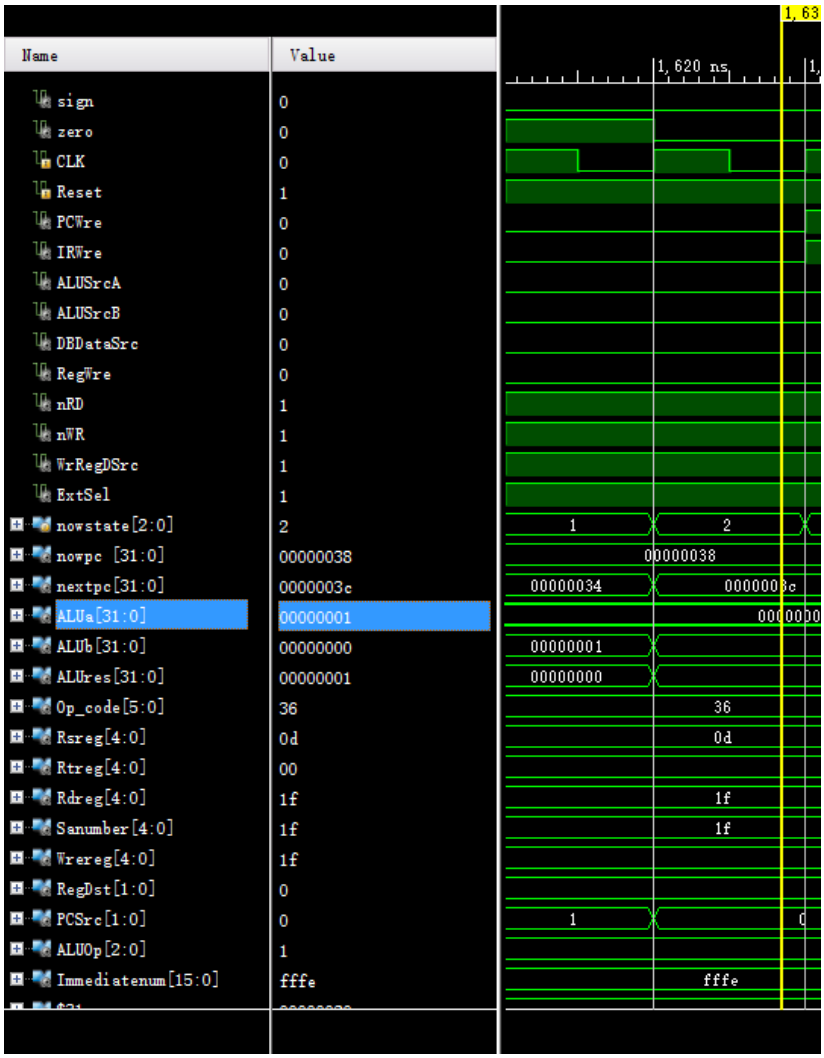
Nowpc	00000038	
Nextpc	00000034	执行跳转指令
Rsreg	0d	
Rtreg	00	
PCSrc	1	执行关于立即数的指令跳转
RegWre	一直是 0	不涉及写回阶段
Nowstate	0,1,2	涉及 IF,ID,EXE 阶段，依次读取指令，指令译码，运算

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000034	addi $13,$13,1	000010	01101	01101	0000 0000 0000 0001	09AD0001



符号	数值	备注
OP_code	000010（2）	addi 为 I 型指令
Nowpc	00000034	
Nextpc	00000038	
Alua	fffffff	
Alub	00000001	
ALUres	00000000	最终值是 0，最终写回的 RegDst 值为 0
Rsreg	0d	
Rtreg	0d	
Immediatenum	0001	
PCSrc	0	正常的 PC 加 4 作为下个 PC 值
RegWre	先 0 后 1	阶段 3 进行写回
Nowstate	0,1,2,3	涉及 IF,ID,EXE,WB 阶段，依次读取指令 ,指令译码 ,运算，结果写回

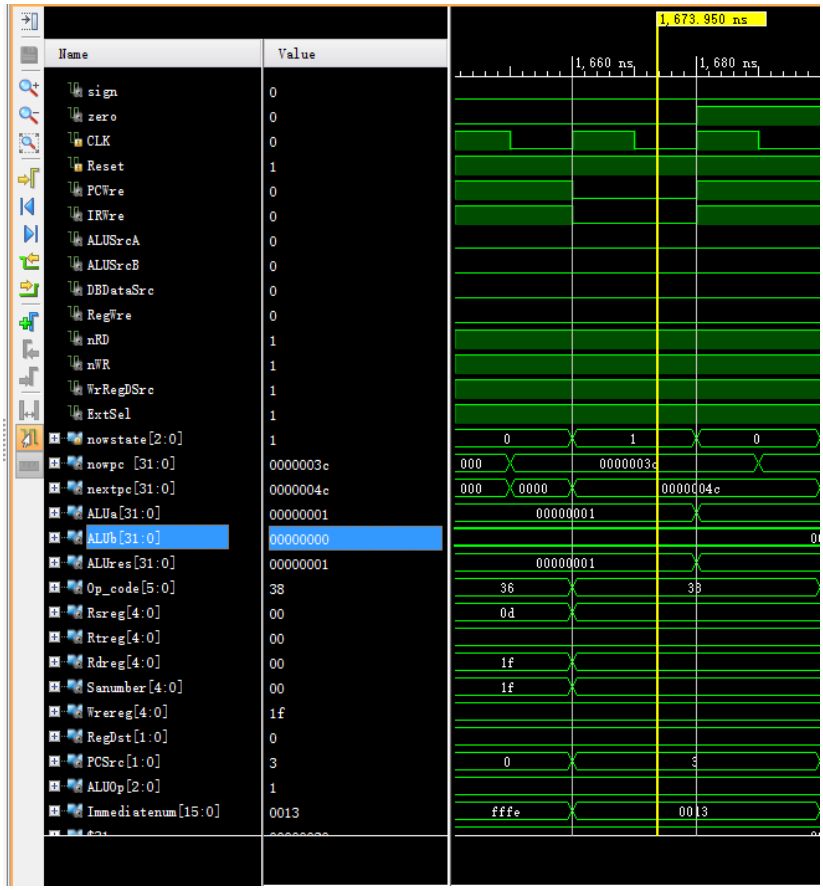
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000038	bltz $13,-2 (<0,转 34)	110110	01101	00000	1111 1111 1111 1110	D9A0FFFE



符号	数值	备注
OP_code	110110（36）	Bltz，为 I 型指令
Nowpc	00000038	
Nextpc	0000003c	
Rsreg	0d	
Rtreg	00	
PCSrc	0	正常 PC+4 作为下一个 PC 值
RegWre	一直是 0	不涉及写回阶段
Nowstate	0,1,2	涉及 IF,ID,EXE 阶段，依次读

		取指令，指令译码，运算

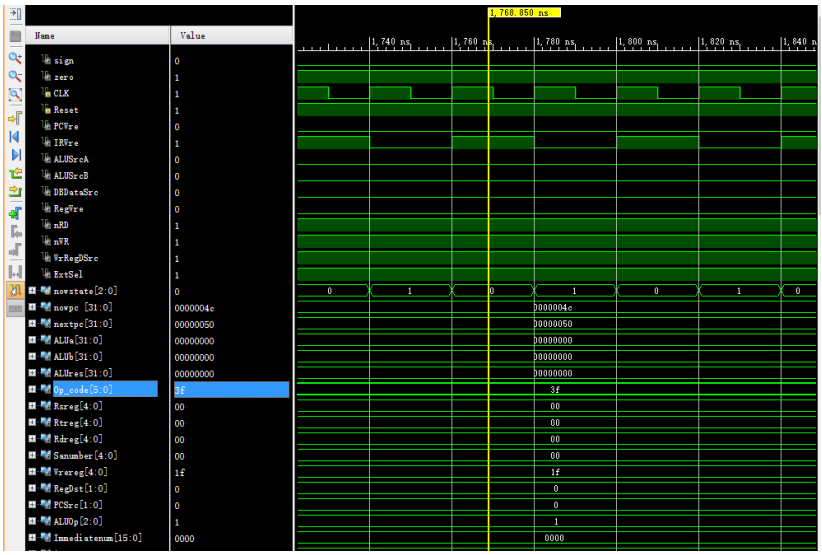
地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x0000003C	j 0x000004C	111000	00000	00000	0000 0000 0001 0011	E0000013



符号	数值	备注
OP_code	111000（38）	J 为 J 型指令
Nowpc	0000003c	
Nextpc	0000004c	做出跳转指令
PCSrc	3	根据地址和 PC 的值，做出 PC 的跳转
RegWre	一直是 0	

Nowstate	0,1,	涉及 IF,ID,依次读取指令，指令译码，

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
0x0000004C	halt	111111	00000	00000	0000 0000 0000 0000	= FC000000



符号	数值	备注
OP_code	111000（3f）	Halt 停机指令
Nowpc	0000004c	
Nextpc	00000050	做出跳转指令
PCSrc	0	此时因为是停机指令，PC 的值将永远保持不变
RegWre	一直是 0	
Nowstate	0,1,	涉及 IF,ID,依次读取指令，指令译码，

七、总结与感悟

本次设计是在单周期 CPU 的基础上完成的，有些模块没有改动或者改动很少，最重要

且最关键的部分是对各个指令阶段内容含义的理解和时序控制的掌握。

首先记录下曾经遇到的问题。

- 部分指令如 Pcsrc 等容易出现高阻态状态或不确定值

  解决方案： 在 debug 过程中发现对同一变量做了多次赋值，这样容易就会使变量的值变成不确定。于是在顶层模块和底层模块中，需要严格区分输入输出变量和局部变量，让局部变量通过 assign 语句实时赋值给输出变量而非对输出变量多次赋值。

- 对 jal 指令的执行周期不清楚

  解决方案：网上看到了相关内容，配合老师课件明确了 jal 指令在第一个时钟周期里只是更新 PC，而在第二个时钟周期指令译码阶段，jal 需要更新 IR 模块并向 31 号寄存器写入 PC+4 的值.

- 烧到 basys3 板上不能写寄存器

  解决方案 :经过反复调试与测试，修改数据选择功能，在 basys3 板上显示 PCwre，Regdst 等信号量的值，发现所有指令都没有被正确译码，然后在 verilog 的输出框发现有一个警告说指令文件没有权限读或写或执行。经过网上查找资料，将文件引入了进来，作为 memory file，警告消失，basys3 板上读写寄存器正常。不清楚这是原因导致在单周期 cpu 正常读取，但在多周期 cpu 烧板时出现的问题。

  接着描述下自己的主要感悟。

- 竞争与冒险问题

  首先要注意到的是寄存器和数据存储器的读取是异步的，不需要时钟信号。而关于写指令，在单周期 CPU 设计中我们采用了时钟下降沿触发寄存器写操作，于是相对应的为了保证 jal 指令可以正常的在时钟下降沿前得到写寄存器$31 的指令，我

们需要设计 control unit 模块是时钟上升沿触发。同样的道理，PC 模块也采用下降沿触发。另外为了保证指令执行的正确，在 IF 阶段读取指令后，为了不让 IR 模块和 PC 模块的写冲突，IR 模块采用上升沿触发。与寄存器类似的，写入内存单元也是下降沿触发。详细的表格已在之前给出。

- 模块化设计

  像多周期 cpu 相比单周期 cpu 又多了几个临时寄存器部件，这样代码中涉及的线和寄存器数量是非常多的，如果不采用模块化的思想设计，那么不止编写代码时思路会很混乱，debug 也会非常困难。自己一开始需要仔细阅读数据通路图，类比面向对象的设计思想：将 CPU 的各个模块进行分离，实现每个模块的时候，只关注这个模块本身的输入输出与逻辑功能。设计完成所有模块后，就是设计顶层文件，将模块整合在一起，完成整体 CPU 的设计。

- 寄存器组的设计

  Registerfile 模块中既涉及到组合逻辑也涉及时序逻辑，组合逻辑要求电路的输出只取决于电路的输入，而时序逻辑要求电路的输出取决于电路的输入和电路当前所储存的状态。因此寄存器模块使用这两种电路保证了寄存器模块的输出依赖于模块内保存的值，但是却不依赖于时钟信号的问题。在多周期 CPU 中，更关键的一点是我们增加了多个临时寄存器，因为一条指令要经历多个时钟周期，如果分不清楚各个临时寄存器在哪个周期输出数据很容易会造成错误。具体的关系见下表

寄存器	IR	ADR	BDR	AluoutDR	DBDR
阶段	IF	ID	ID	EXE	EXE/MEM

- 控制单元的设计

  在 CU 模块中，控制信号的变化只取决于当前状态和操作码，而每当时钟上升沿到

来时，CU 模块都会对信号进行一次更新。与单周期 CPU 不同的是，多周期 CPU 对于同一条指令，在不同的时钟周期中，所需要的控制信号也会不相同。因此，它需要一个状态转移的模块，来处理状态变化，将状态的变化与控制信号的变化分开来使代码更易写，变化更稳定。