《计算机组成原理实验》

实验报告

（实验三）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	17 计教学 2 班
学 生 姓 名 ：	刘斯宇
学　　号 ：	17341110
时　　间 ：	2018　年　12　月　17　日

成 绩 ：

实验三 ： 多周期CPU设计与实现

一、 实验目的

(1) 认识和掌握多周期数据通路图的构成、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。

在本文档中，提供的相关内容对于设计可能不足或甚至有错误，希望同学们在设计过程中如发现有问题，请你们自行改正，进一步补充、完善。谢谢！

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。

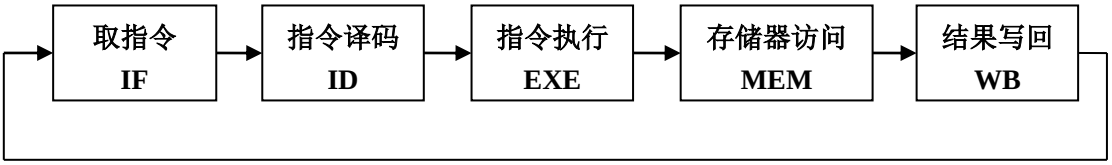
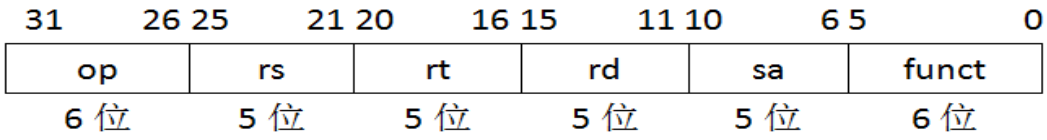


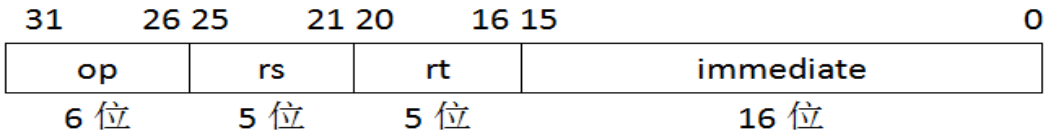
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式:

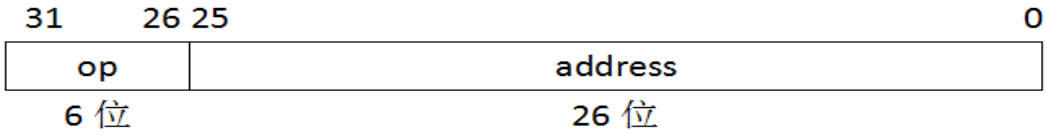
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Load）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；

address：为地址。

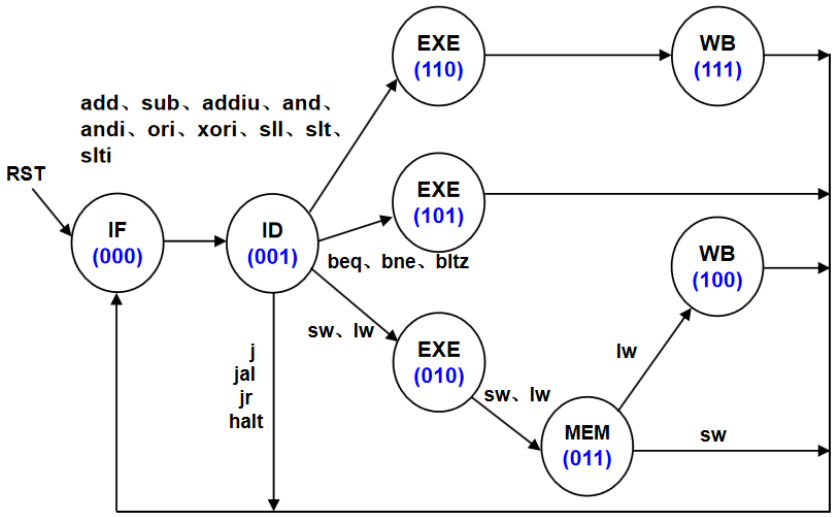


图 2 多周期 CPU 状态转移图

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

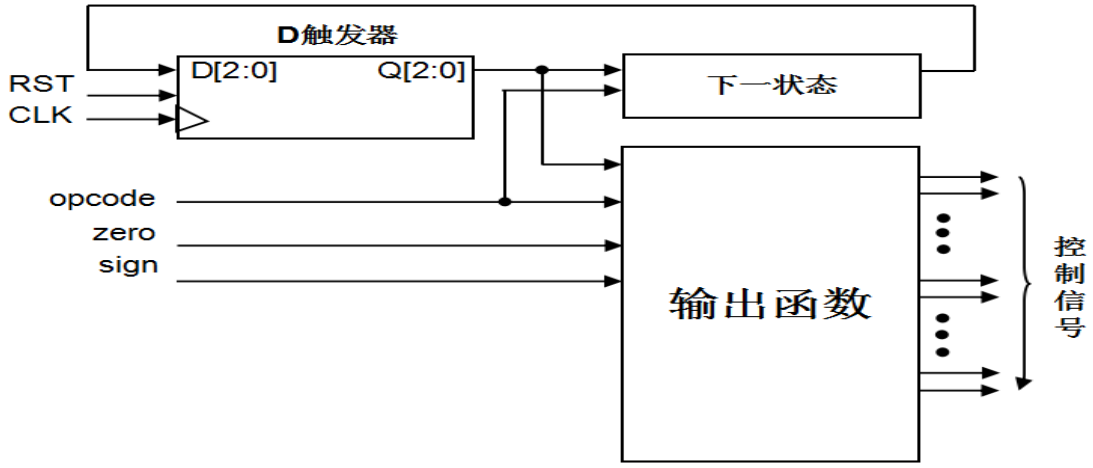


图 3 多周期 CPU 控制部件的原理结构图

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

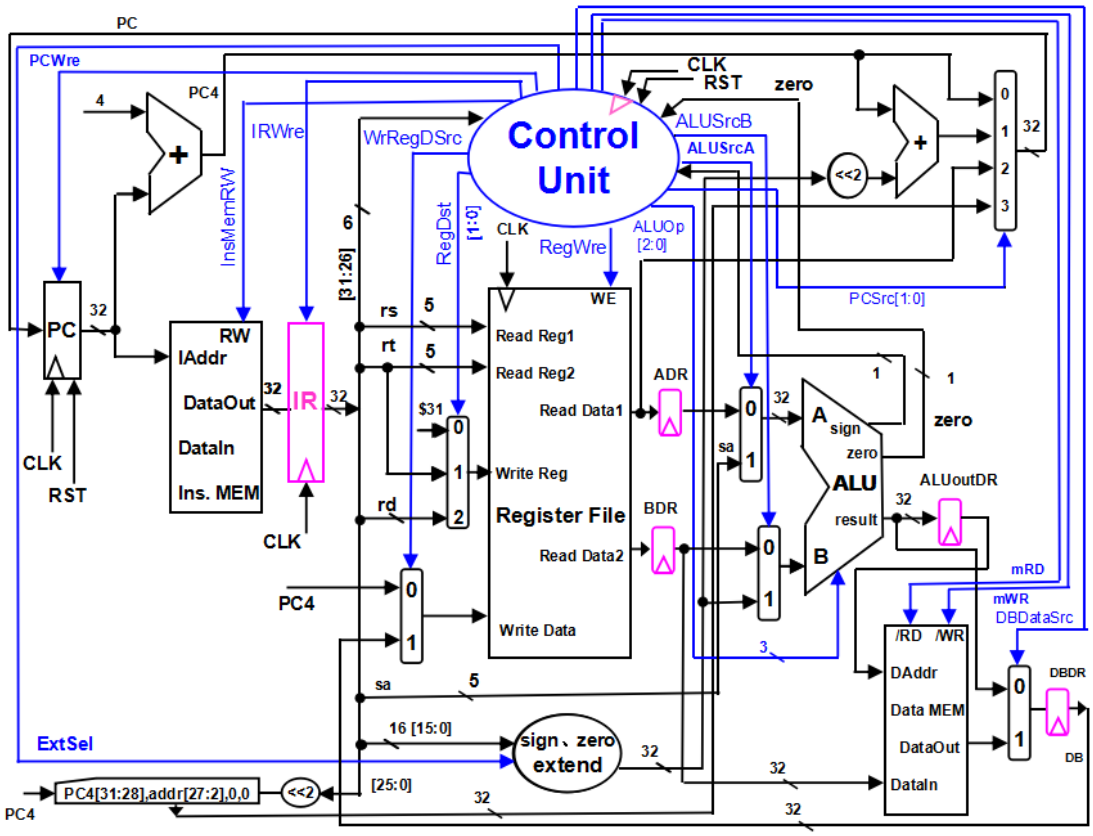


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时不需要时钟信号，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、	来自 sign 或 zero 扩展的立即数，相关指令：addiu、andi、ori、xori、slti、

	beq、bne、bltz	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

五. 实验过程与结果

（1）设计了 PC，寄存器，寄存器堆，ALU，控制端元，RAM，ROM，扩展等模块

（2）最后，构建顶层模块将底层模块串联起来，并在通过测试后，使得 CPU 能正常工 作。

这次多周期，我主要设计如下的模块：

分解指令单元

```
module DivINS(
        input [31:0] instruction,
        output reg[5:0] op,
        output reg[4:0] rs,
        output reg[4:0] rt,
        output reg[4:0] rd,
        output reg[4:0] sa,
        output reg[15:0] immediate,
        output reg[25:0] addr
    );

    initial begin
        op = 5'b00000;
        rs = 5'b00000;
        rt = 5'b00000;
        rd = 5'b00000;
    end

    always@(instruction)
    begin
        op = instruction[31:26];
        rs = instruction[25:21];
        rt = instruction[20:16];
        rd = instruction[15:11];
        sa = instruction[10:6];
        immediate = instruction[15:0];
        addr = instruction[25:0];
    end
endmodule
```

这个模块我主要是为了分解指令，因为当时做单周期的时候，我没有这个模块，由于我的粗心，导致很多本来可以减少的 bug，所以这次设计多周期，我一上来就设计了这个模块。

IR 模块//上升沿触发

```
module IR(
        input [31:0] instruction,
        input CLK,
        input IRWre,
        output reg[31:0] IRInstruction
    );
    initial begin
        IRInstruction = 0;
```

```
    end
    always@(posedge CLK)
    begin
        if(IRWre) begin
            IRInstruction <= instruction;
        end
    end
endmodule
```

IR 指令寄存器，目的是使指令代码保持稳定，我们采用的是上升沿触发，主要是为了区别 PC 写的下降沿。

PCchange 单元

```
module PCchange(
        input Reset,
        input [1:0] PCSrc,
        input [31:0] IMM,
        input [25:0] addr,
        input [31:0] curPC,
        input [31:0] rs,
        output reg[31:0] nextPC
    );

    initial begin
        nextPC = 0;
    end

    reg [31:0] pc;

    always@(*)
    begin
        if(!Reset) begin
            nextPC = 0;
        end
        else begin
            pc = curPC + 4;
            case(PCSrc)
                2'b00: nextPC = curPC + 4;//add 4
                2'b01: nextPC = curPC + 4 + IMM * 4;//beq,bne
                2'b10: nextPC = rs;//jalr
                2'b11: nextPC = {pc[31:28],addr,2'b00};//j
            endcase
        end
    end
endmodule
```

这个单元主要是为了解决 PC 变化的各种情况，我将四种情况都写到了一个模块中，主要是为了方便简化模块。

PC 模块//上升沿触发

```
module PC(
       input CLK,
       input Reset,
       input PCWre,
       input [31:0] nextPC,
       output reg[31:0] curPC
    );
    initial begin
        curPC = 32'b0;
    end
    always@(posedge CLK or negedge Reset)
    begin
        if(!Reset)
            begin
                curPC <= 0;
            end
        else
            begin
                if(PCWre)
                    begin
                        curPC = nextPC;
                    end
                else
                    begin
                        curPC = curPC;//停机
                    end
            end
    end
endmodule
```

ADR，BDR，ALUoutDR，DBDR 寄存器模块：上升沿触发

```
module DataReg(
        input CLK,
        input [31:0] In,
        output reg[31:0] Out
    );

    initial begin
        Out = 0;
    end
```

```
    always@(posedge CLK) begin
        Out <= In;
    end
```

控制端元

分成不同状态，分别给出了不同的控制信号。

```
module Control(
        input CLK,
        input Reset,
        input zero,
        input [5:0] op,
        output reg IRWre,
        output reg PCWre,
        output reg ExtSel,
        output reg InsMemRW,
        output reg WrRegDSrc,
        output reg [1:0] RegDst,
        output reg RegWre,
        output reg ALUSrcA,
        output reg ALUSrcB,
        output reg [1:0]PCSrc,
        output reg [2:0]ALUOp,
        output reg mRD,
        output reg mWR,
        output reg DBDataSrc
    );

    reg [2:0] state, nextState;
    parameter [2:0] iniState = 3'b111,
                    sIF = 3'b000,
                    sID = 3'b001,
                    sEXE = 3'b010,
                    sMEM = 3'b100,
                    sWB = 3'b011;
    initial begin
        state = iniState;
        PCWre = 0;
        InsMemRW = 0;
        IRWre = 0;
        RegWre = 0;  ;
        ExtSel = 0;
        PCSrc = 2'b00;
        RegDst = 2'b11;
        ALUOp = 0;
```

```
        ExtSel = 0;
        WrRegDSrc = 0;
        ALUSrcA = 0;
        ALUSrcB = 0;
        DBDataSrc = 0;
        mRD = 0;
        mWR = 0;
    end

    always@(posedge CLK) begin
        if(!Reset) begin
            state <= sIF;
        end else begin
            state <= nextState;
        end
    end

    always@(state or op or zero) begin
        case(state)
            iniState : nextState = sIF;
            sIF: nextState = sID;
            sID: begin
                case(op[5:3])
                    3'b111: nextState = sIF;
                    default: nextState = sEXE;
                endcase
            end
            sEXE: begin
                if((op == 6'b110100) ||(op == 6'b110101)|| (op == 6'b110110))
begin
                    nextState = sIF;
                end else if(op == 6'b110000 || op == 6'b110001) begin
                    //sw,lw
                    nextState = sMEM;
                end else begin
                    nextState = sWB;
                end
            end
            sMEM: begin
                if(op == 6'b110000) begin
                    //sw
                    nextState = sIF;
                end else begin
                    //lw
```

```
                    nextState = sWB;
                end
            end
            sWB: nextState = sIF;
        endcase

        if(nextState == sIF && op != 6'b111111 && state != iniState)
         begin
            PCWre = 1;
            InsMemRW = 1;
        end else begin
            PCWre = 0;
            InsMemRW = 0;
        end

        if(state == sIF || nextState == sID) begin
            IRWre = 1;
        end else begin
            IRWre = 0;
        end


        if(op == 6'b011000)
         begin
            ALUSrcA = 1;
        end
        else begin
            ALUSrcA = 0;
        end

        if(op == 6'b000010 ||op==6'b010001|| op == 6'b010010||op == 6'b010011
|| op == 6'b110000 || op == 6'b110001 || op == 6'b100110)
        begin
           ALUSrcB = 1;
        end else
        begin
           ALUSrcB = 0;
        end

        if(op == 6'b110001)
         begin
            DBDataSrc = 1;
        end
        else begin
```

```
            DBDataSrc = 0;
        end

        if((state == sWB && op != 6'b110100&&op != 6'b110101 && op != 6'b110000
&& op != 6'b110110) || (op == 6'b111010 && state == sID))
        begin
            RegWre = 1;
            if(op == 6'b111010)
            begin
                WrRegDSrc = 0;
                RegDst = 2'b00;
            end
            else begin
                WrRegDSrc = 1;
                if(op == 6'b000010||op==6'b010001 || op == 6'b010010 ||op ==
6'b010011|| op == 6'b100110 || op == 6'b110001)
                begin
                    RegDst = 2'b01;
                end
                else begin
                    RegDst = 2'b10;
                end
            end
        end else begin
            RegWre = 0;
        end

        if(op != 6'b111111)
            InsMemRW = 1;
        mRD = (op == 6'b110001) ? 1 : 0;

        mWR = (state == sMEM && op == 6'b110000) ? 1 : 0;


        ExtSel = (op == 6'b000010 || op == 6'b110001 ||op == 6'b110101|| op
== 6'b110000 || op == 6'b110100 || op == 6'b110110) ? 1 : 0;

        if(op == 6'b111001)
         begin
            PCSrc = 2'b10;
        end
        else if((op == 6'b110100 && zero==0)||(op == 6'b110101 && zero==1)
|| (op == 6'b110110 && !zero))
        begin
```

```
            PCSrc = 2'b01;
        end
        else if(op == 6'b111010 || op == 6'b111000)
        begin
            PCSrc = 2'b11;
        end
        else begin
            PCSrc = 2'b00;
        end
        case(op)
            6'b000000: ALUOp = 3'b000; //add
            6'b000010: ALUOp = 3'b000;  // addi
            6'b010011: ALUOp = 3'b111;  // xori
            6'b010010: ALUOp = 3'b101;  // ori
            6'b010000: ALUOp = 3'b110;  // and
            6'b000001: ALUOp = 3'b001;  // sub
            6'b010001: ALUOp = 3'b110;  // andi
            6'b011000: ALUOp = 3'b100;  // sll
            6'b110100: ALUOp = 3'b001;  // beq
            6'b110101: ALUOp = 3'b001;  // bne
            6'b100111: ALUOp = 3'b010;  // slt
            6'b100110: ALUOp = 3'b010;  // slti
            6'b110110: ALUOp = 3'b001;  // bltz
            6'b110000: ALUOp = 3'b000;  //sw
            6'b110001: ALUOp = 3'b000;  //lw

        endcase
    end
endmodule
```

ALU 模块

```
module ALU(
        input ALUSrcA,
        input ALUSrcB,
        input [31:0] ReadData1,
        input [31:0] ReadData2,
        input [4:0] sa,
        input [31:0] extend,
        input [2:0] ALUOp,
        output reg zero,
        output reg[31:0] result
    );
    reg [31:0] A;
    reg [31:0] B;
```

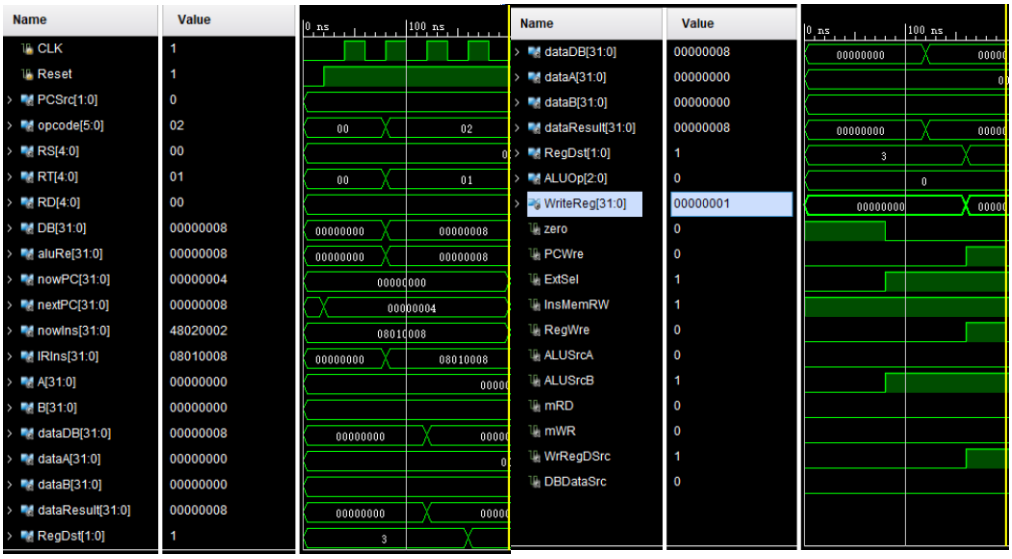
```
    initial begin
        result = 0;
        zero = 0;
    end
    always@(ReadData1 or ReadData2 or ALUSrcA or ALUSrcB or ALUOp)
    begin
        A = (ALUSrcA == 0) ? ReadData1 : sa;
        B = (ALUSrcB == 0) ? ReadData2 : extend;
        case(ALUOp)
            3'b000: result = A + B;
            3'b001: result = A - B;
            3'b010: result = B << A;
            3'b011: result = A | B;
            3'b100: result = A & B;
            3'b101: result = (A < B) ? 1 : 0;
            3'b110: result = (((A < B) && (A[31] == B[31] )) ||( ( A[31] ==1
&& B[31] == 0))) ? 1:0;
            3'b111: result = A ^ B;
        endcase
        zero = (result == 0) ? 1 : 0;
    end
endmodule
```

测试代码：

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	
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	00100 000 0000 0000	=	04612000
0x00000010	and $5,$4,$2	010000	00100	00010	00101 000 0000 0000	=	40822800
0x00000014	sll $5,$5,2	011000	00000	00101	00101 00010 00 0000	=	60052880
0x00000018	beq $5,$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	jal 0x0000050	111010	00000	00000	0000 0000 0101 0000	=	E8000050
0x00000020	slt $8,$13,$1	100111	01101	00001	01000 000 0000 0000	=	9DA14000
0x00000024	addiu $14,$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFE
0x00000028	slt $9,$8,$14	100111	01000	01110	01001 000 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	01011 000 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 0101 1100	=	E000005C
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	0000 0000 0000 0000	=	FC000000

下面就具体的波形来介绍每一条指令的执行情况：

0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008

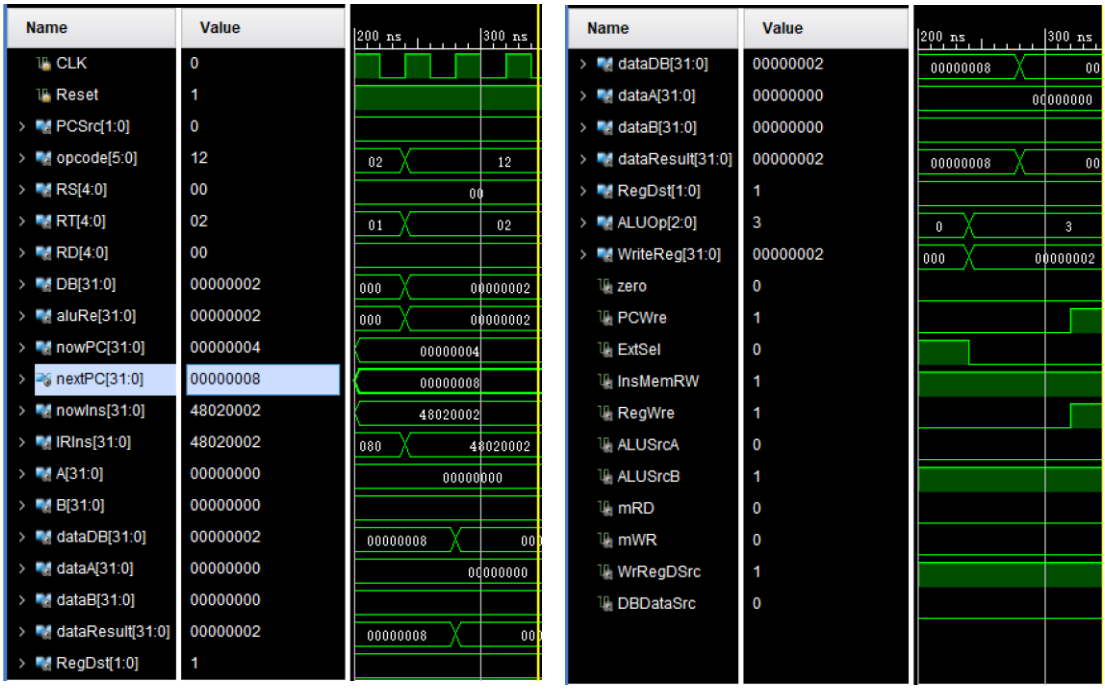


这条指令的作用是$1=$0+8，我们可以看到 aluRe=8 正确.

截至现阶段为止，各寄存器的值如下：

$1	8	$2		$3		$4		$5	
$6		$7		$8		$9		$10	
$11		$12		$13		$14		$15	

0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002

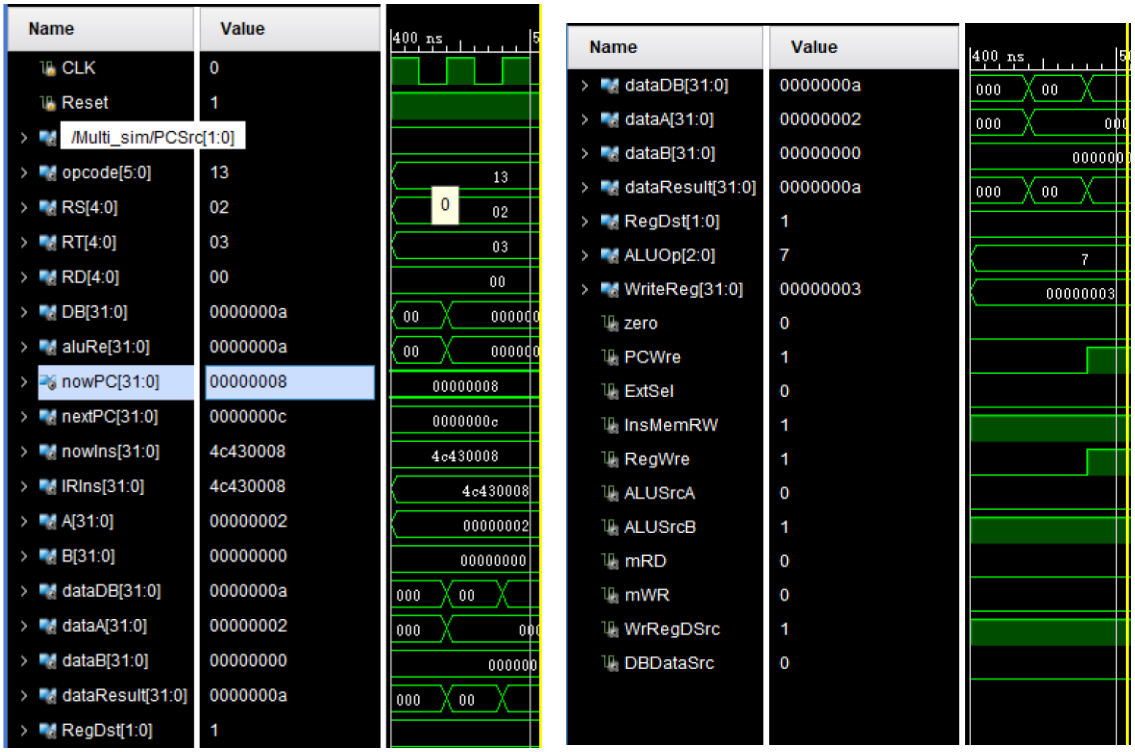


这条指令的作用是$2=$0|2，我们可以看到 aluRe=2 正确.

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3		$4		$5	
$6		$7		$8		$9		$10	
$11		$12		$13		$14		$15	

0x00000008	xori $3,$2,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008

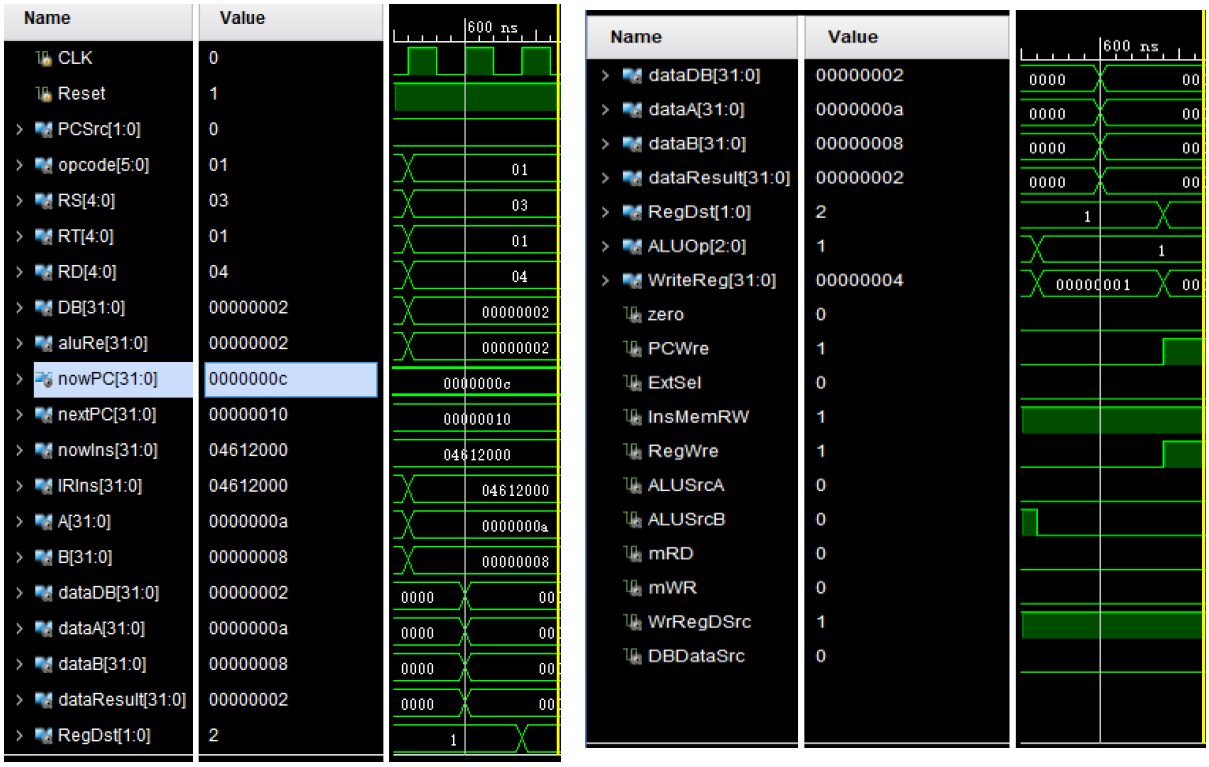


这条指令的作用是$3=$2^8，我们可以看到 aluRe=10 正确.

截至现阶段为止，各寄存器的值如下:

$1	8	$2	2	$3	10	$4		$5	
$6		$7		$8		$9		$10	
$11		$12		$13		$14		$15	

0x0000000C	sub $4,$3,$1	000001	00011	00001	00100 000 0000 0000	=	04612000

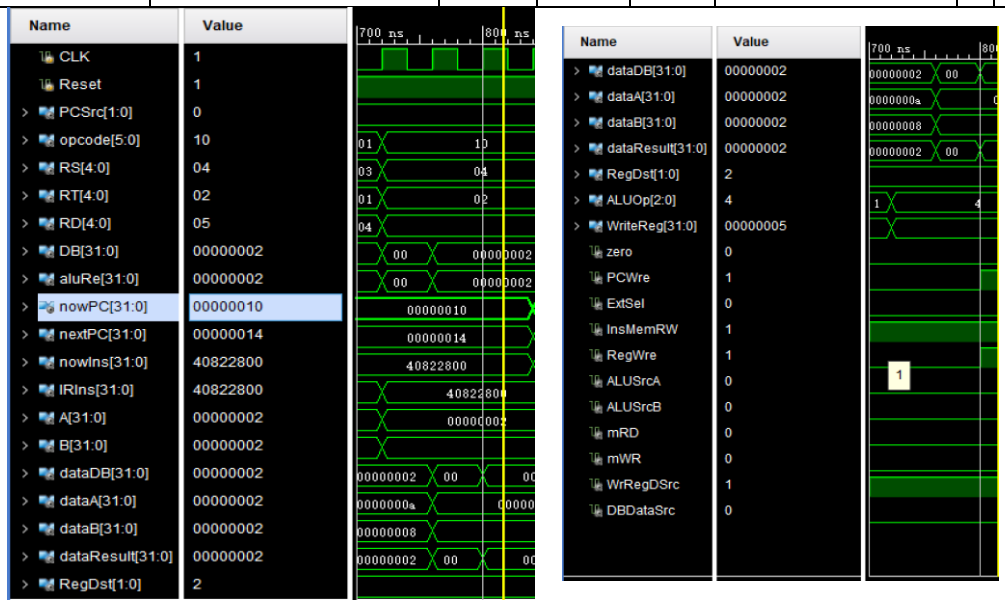


这条指令的作用是**$4=$3-$1，我们可以看到 aluRe=2 正确.**

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	
$6		$7		$8		$9		$10	
$11		$12		$13		$14		$15	

0x00000010	and $5,$4,$2	010000	00100	00010	00101 000 0000 0000	=	40822800

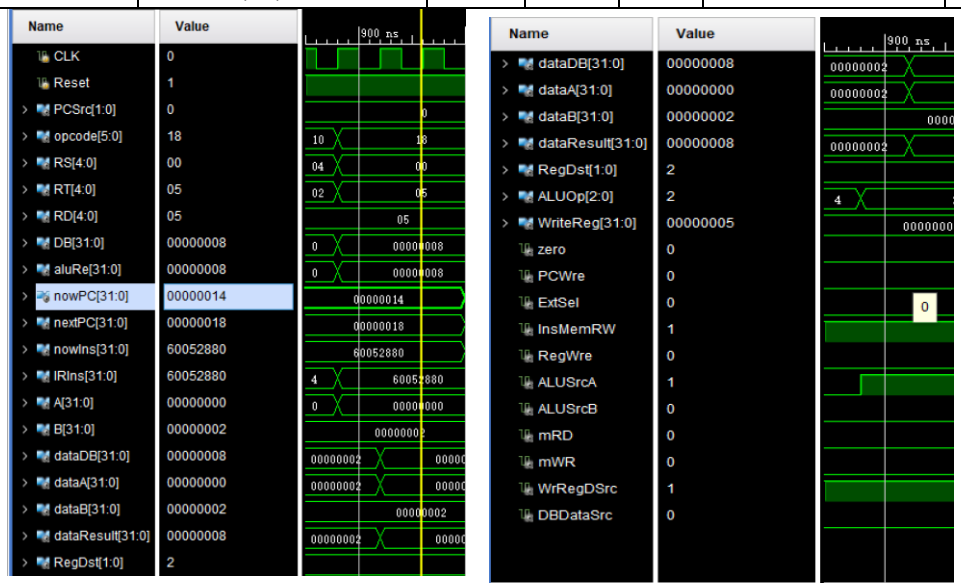


这条指令的作用是$5=$4&$2，我们可以看到 aluRe=2 正确.

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	4
$6		$7		$8		$9		$10	
$11		$12		$13		$14		$15	

0x00000014	sll $5,$5,2	011000	00000	00101	00101 00010 00 0000	=	60052880

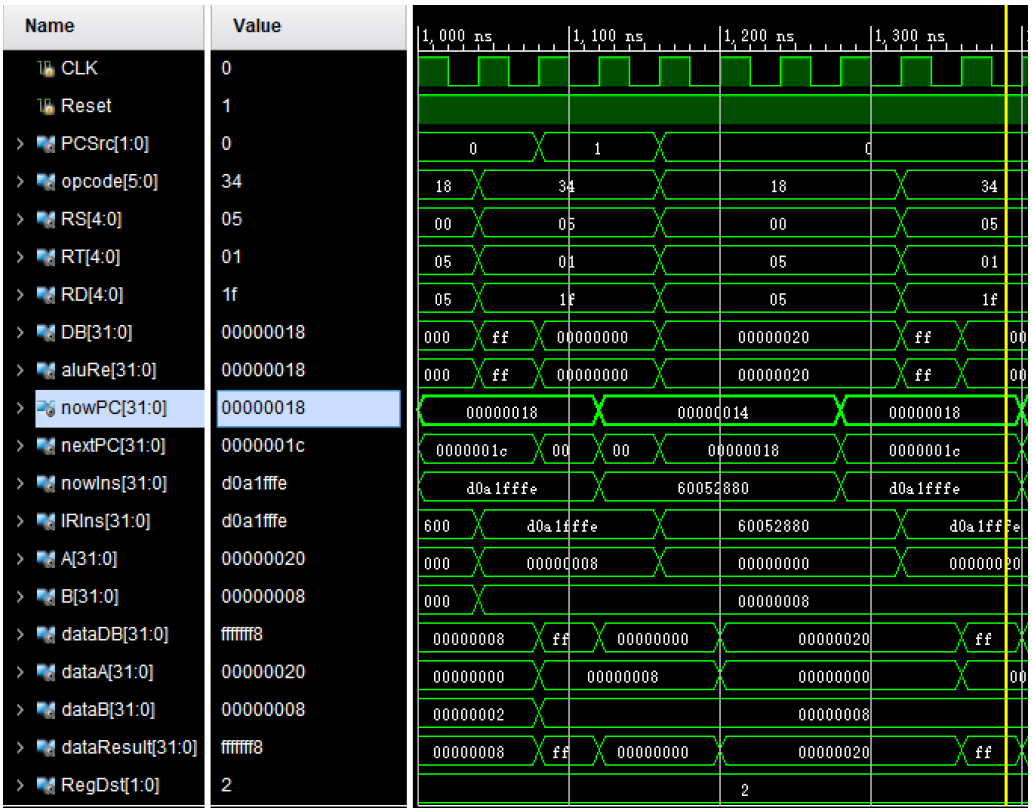


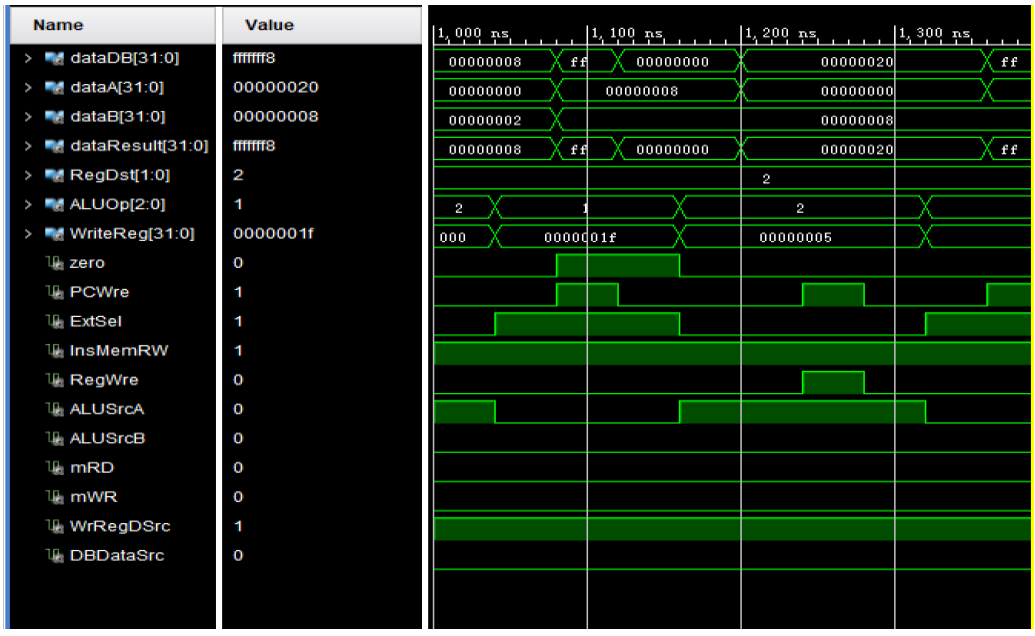
这条指令的作用是$5=$5<<2，我们可以看到 aluRe=8 正确.

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	8
$6		$7		$8		$9		$10	
$11		$12		$13		$14		$15	

0x00000018	beq $5,$1,-2(=,转 14)	110100	00001	00101	1111 1111 1111 1110	=	D025FFFE
0x00000014	sll $5,$5,2	011000	00000	00101	00101 00010 00 0000	=	60052880
0x00000018	beq $5,$1,-2	110100	00001	00101	1111 1111 1111 1110	=	D025FFFE



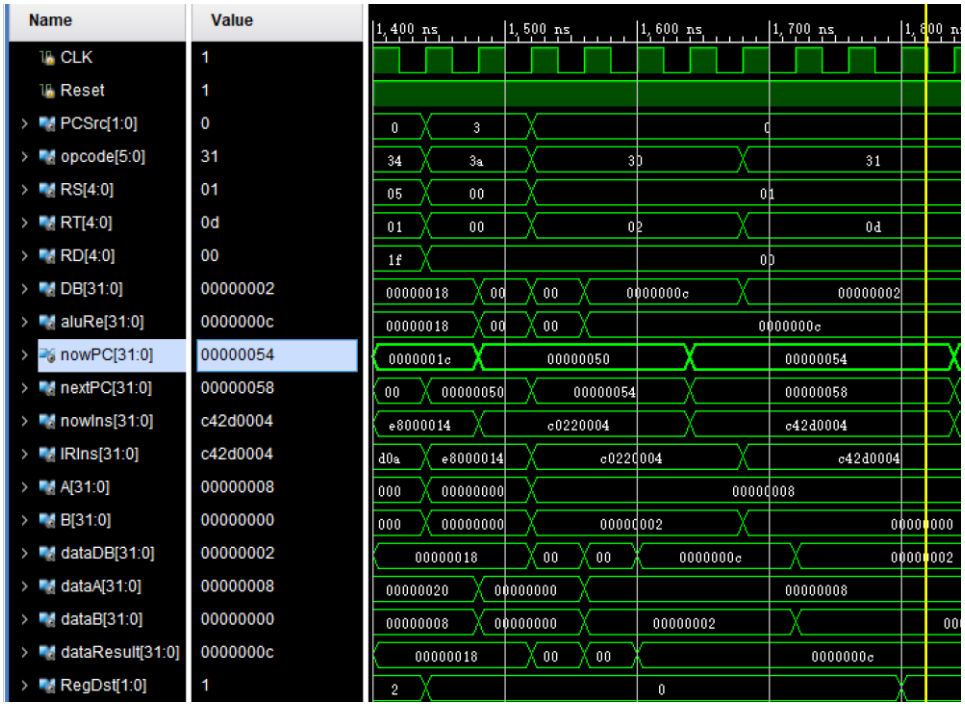


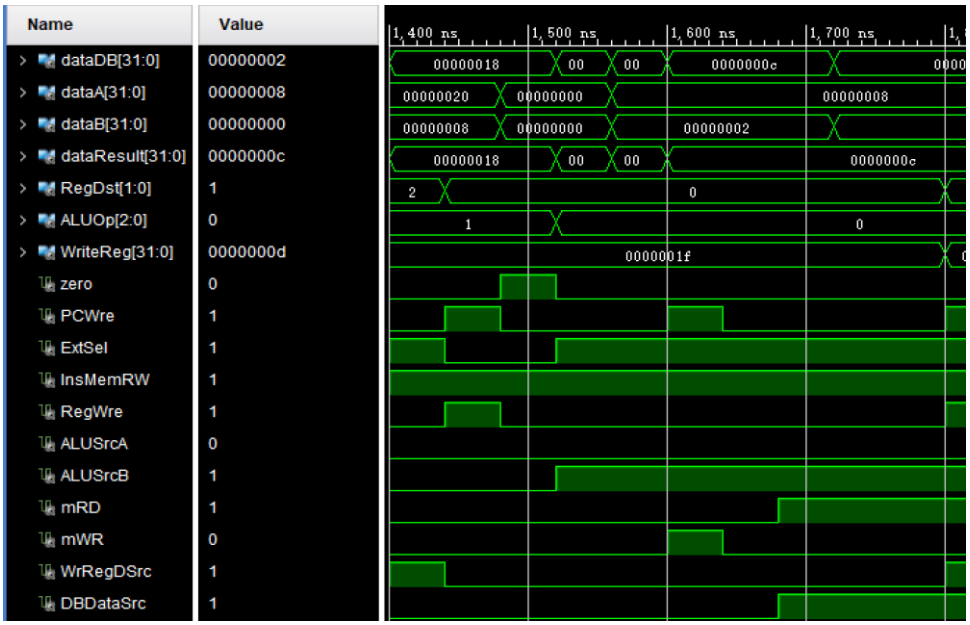
第一次 beq 的时候，由于$5=$1=8,所以跳到 0x00000014，之后$5=32，

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8		$9		$10	
$11		$12		$13		$14		$15	

0x0000001C	jal 0x0000050	111010	00000	00000	0000 0000 0101 0000	=	E8000050
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



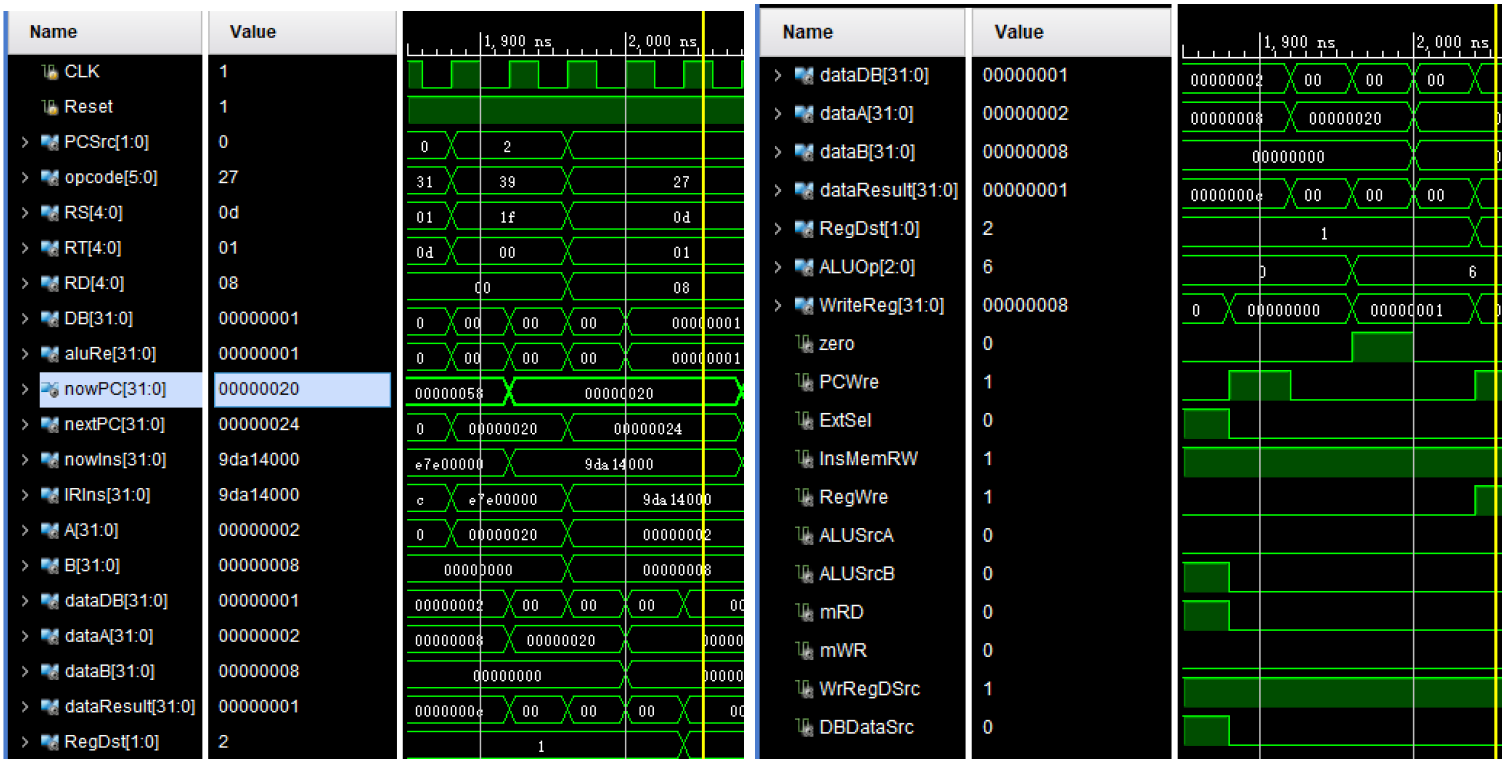


这三条指令的目的是先跳到，0x00000050，根据波形我们发现是正确的，然后把$2 的值存到地址为 12 的内存中，再将内存 12 的值取出来，我们发现 dataDB=2 正确，WriteReg=13，所以这三条指令的执行是正确的。

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8		$9		$10	
$11		$12		$13	2	$14		$15	

0x00000058	jr $31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x00000020	slt $8,$13,$1	100111	01101	00001	01000 000 0000 0000	=	9DA14000

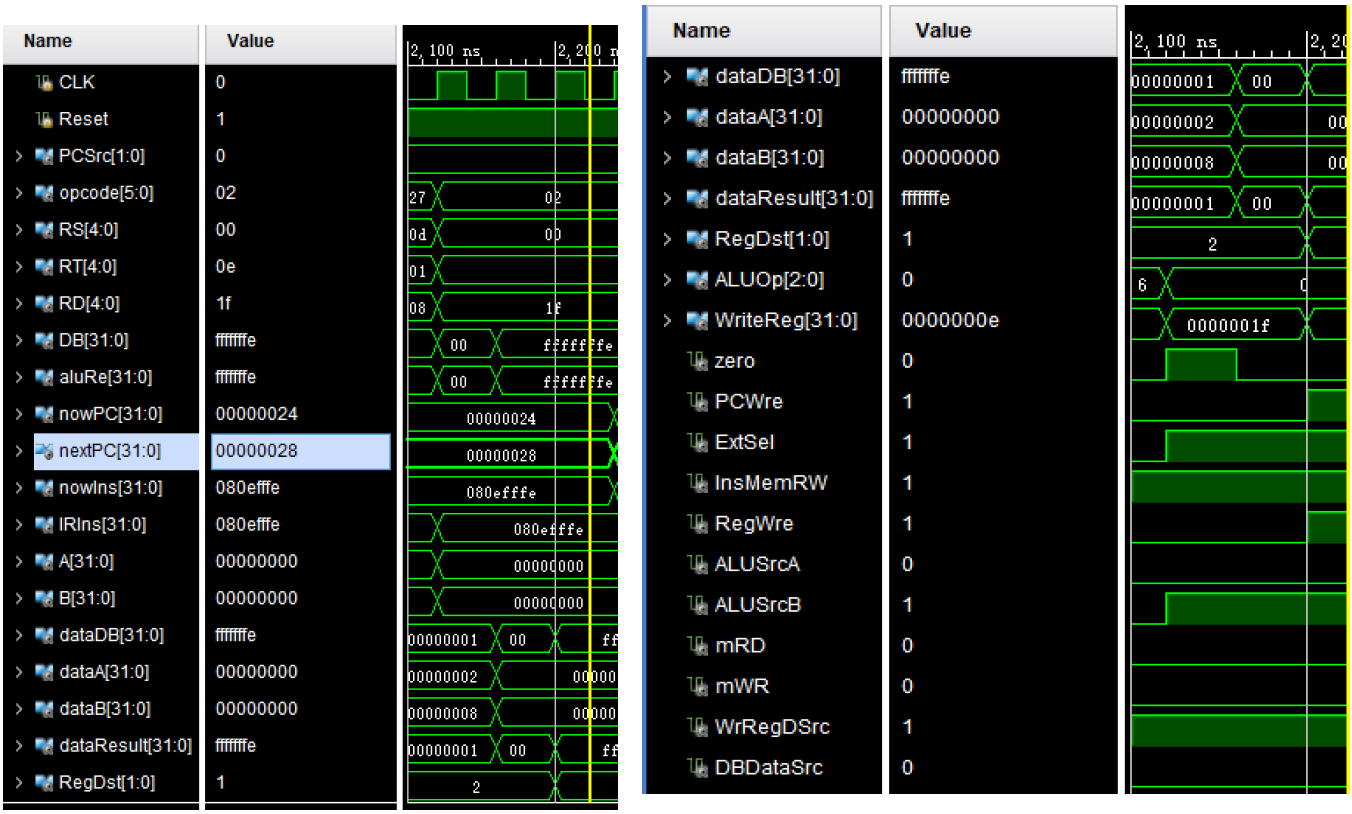


这两条指令的目的是先跳到（$31=0x00000020），根据波形我们发现是正确的，然后由于$13=2<$1=8,所以$8=1;

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9		$10	
$11		$12		$13	2	$14		$15	

0x00000024	addiu $14,$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFE

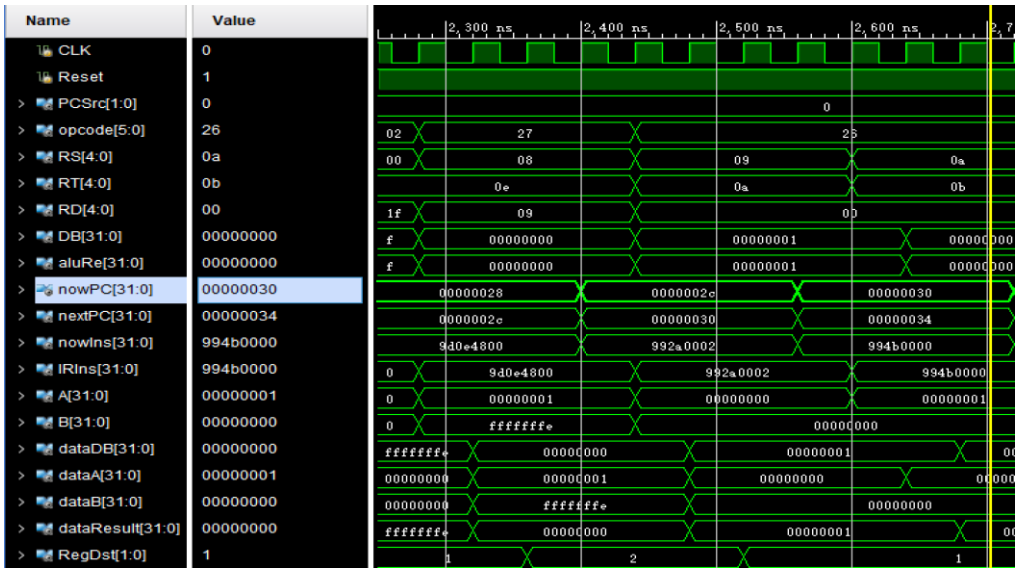


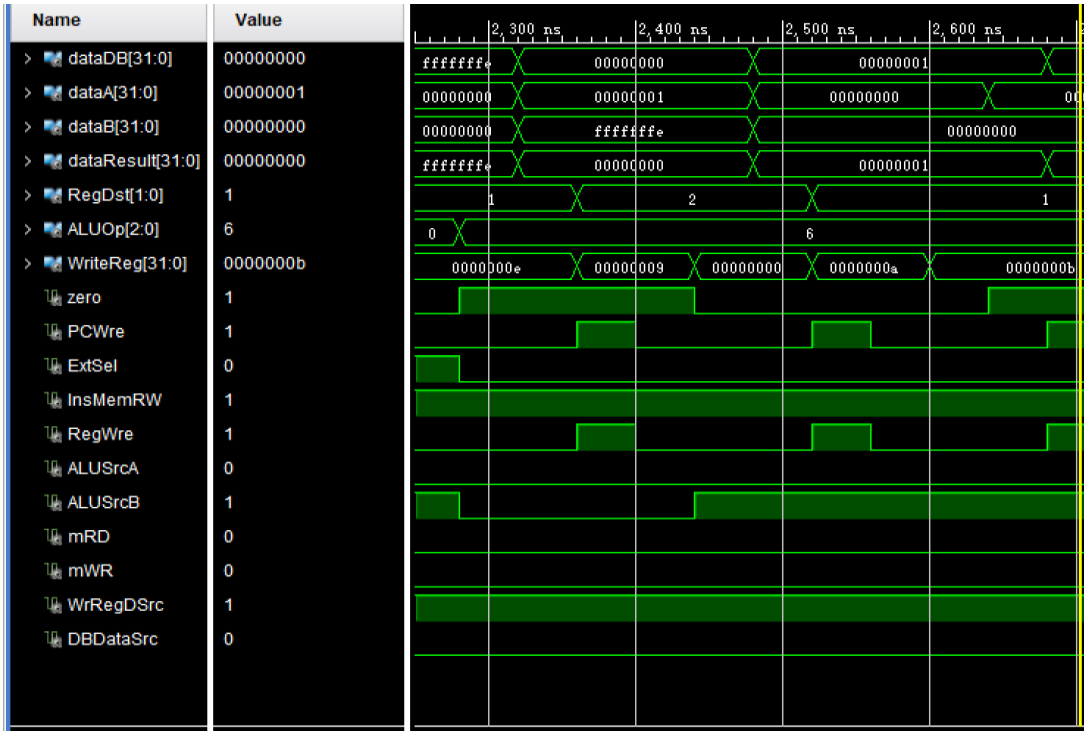
这条指令的作用是$14=$0-2,观察波形发现 aluRe=-2，正确，

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9		$10	
$11		$12		$13	2	$14	-2	$15	

0x00000028	slt $9,$8,$14	100111	01000	01110	01001 000 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





由于$8=1>$14=-2,所以$9=0;

由于$9=0<2,所以$10=1;

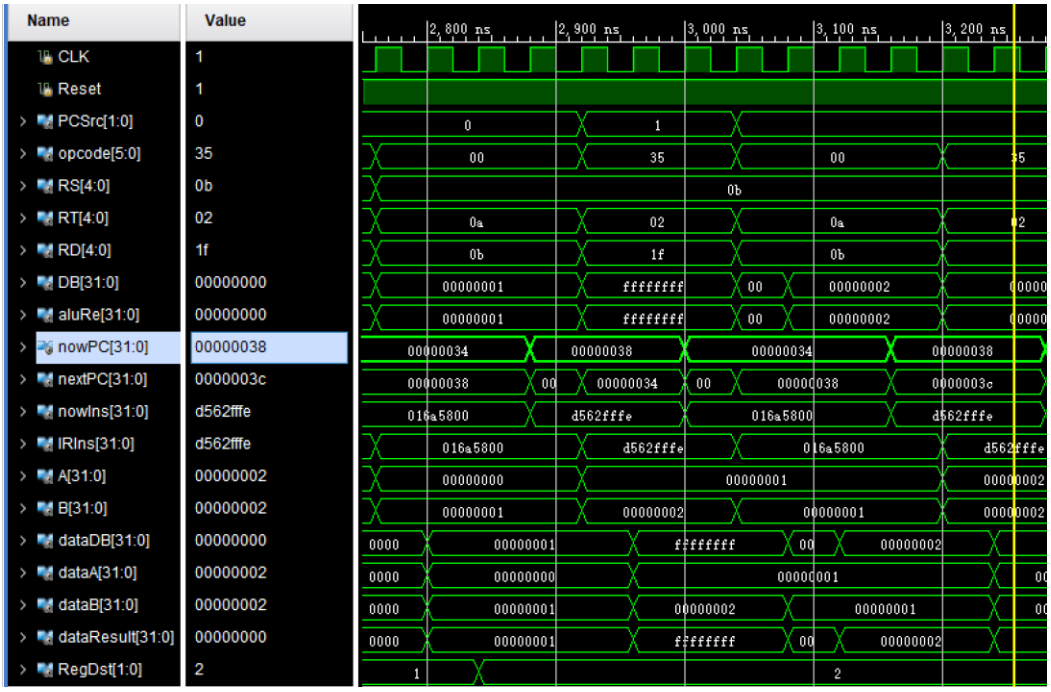
由于$10=1>0,所以$11=0;

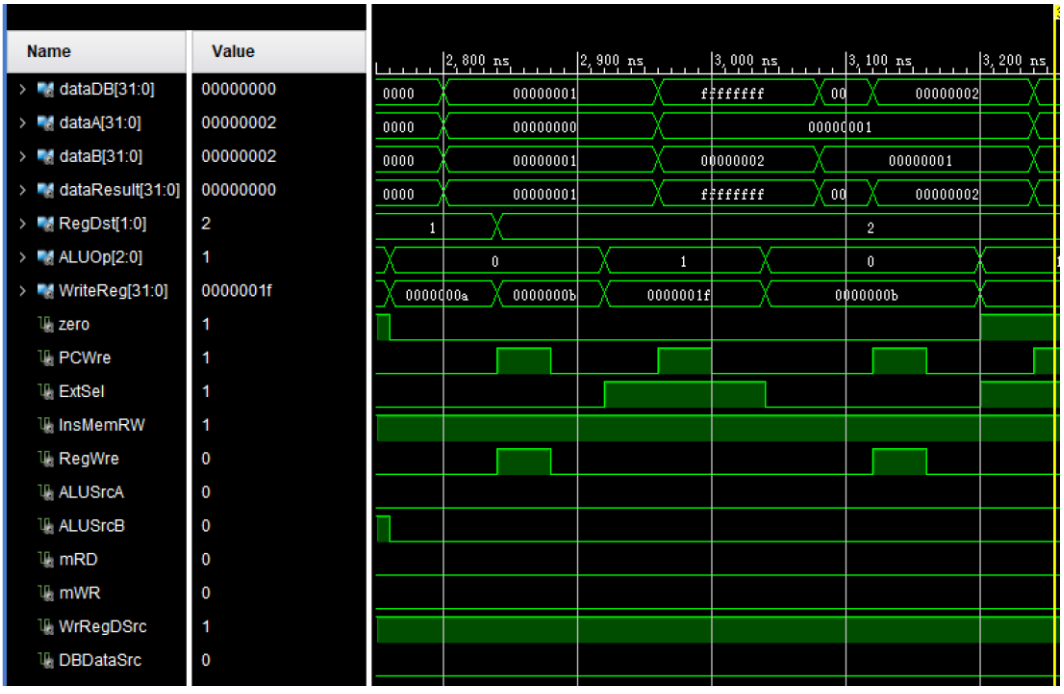
观察波形发现正确，

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9	0	$10	1
$11	0	$12		$13	2	$14	-2	$15	

0x00000034	add $11,$11,$10	000000	01011	01010	01011 000 0000 0000	=	016A5800
0x00000038	bne $11,$2,-2 (≠,转 34)	110101	00010	01011	1111 1111 1111 1110	=	D44BFFFE
0x00000034	add $11,$11,$10	000000	01011	01010	01011 000 0000 0000	=	016A5800
0x00000038	bne $11,$2,-2	110101	00010	01011	1111 1111 1111 1110	=	D44BFFFE





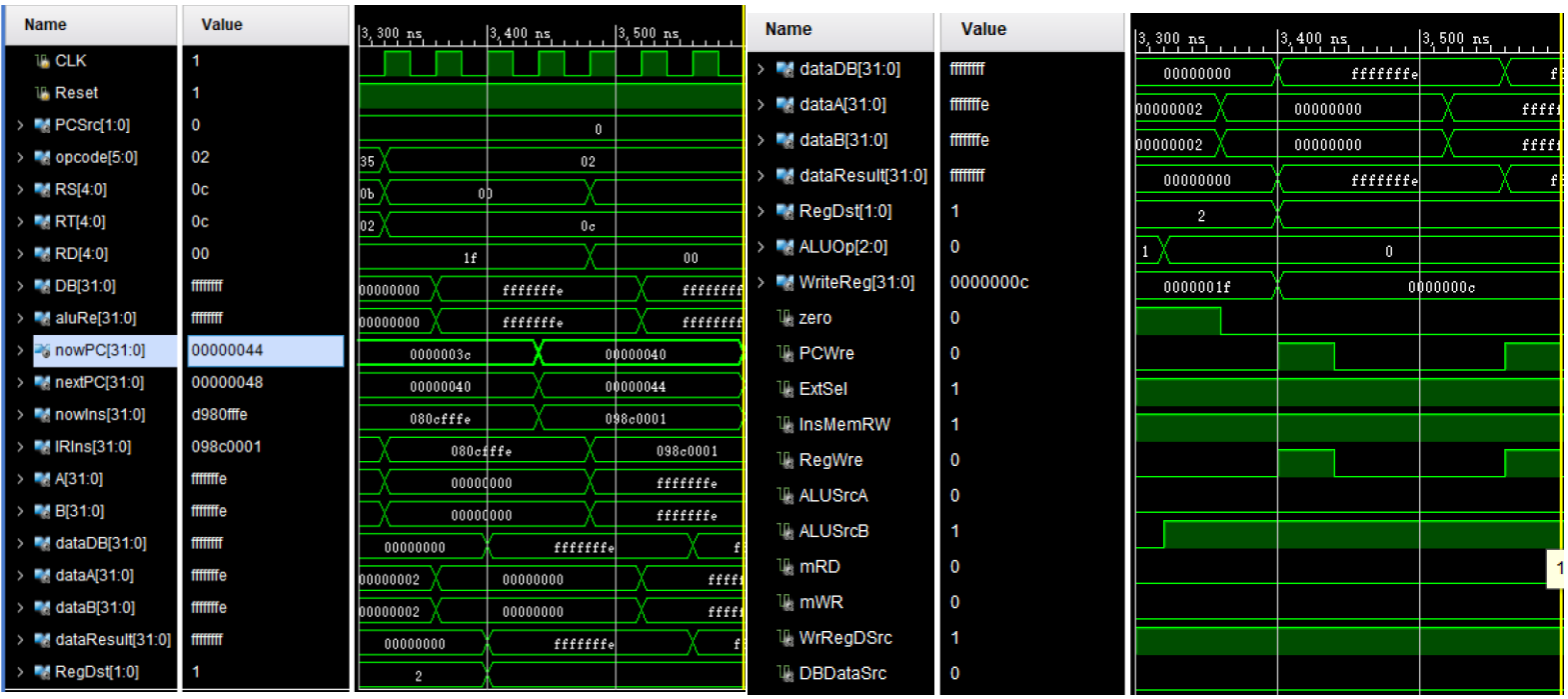
这四条指令的效果是先将$11 加上$10，此时$11=1，由于$11=1≠$2=2，所以跳到 0x00000034，将$11 加上$10，此时$11=2，由于$11=2=$2=2，此时便不再进行跳转了。

观察波形发现正确，

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9	0	$10	1
$11	2	$12		$13	2	$14	-2	$15	

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



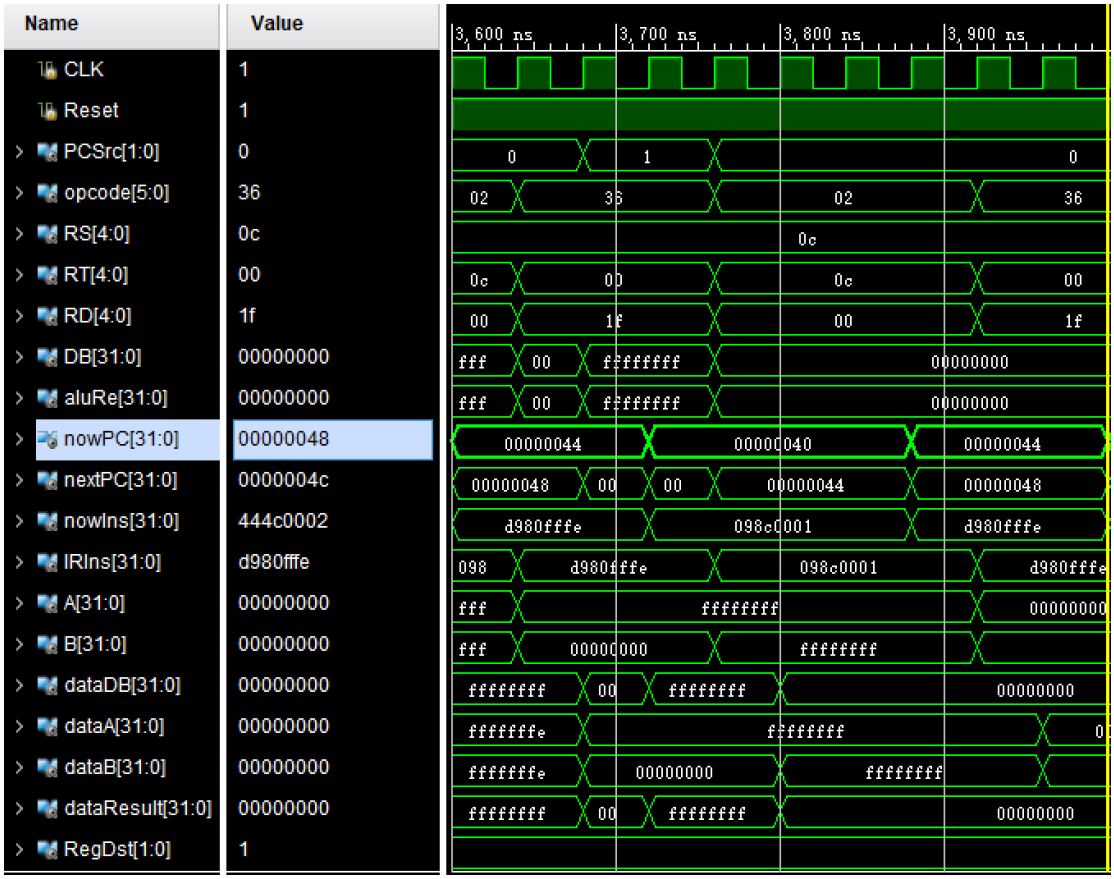
这两条指令的效果是$12=$0+（-2）=-2;

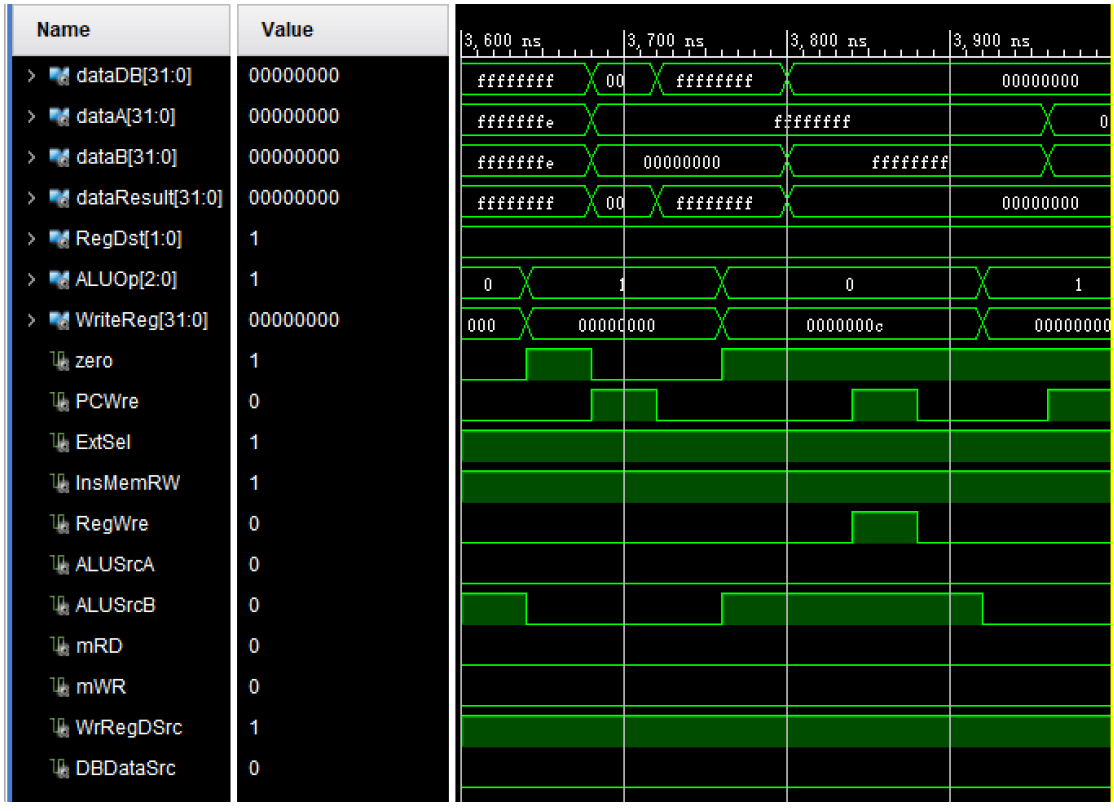
$12=$12+1=-1

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9	0	$10	1
$11	2	$12	-1	$13	2	$14	-2	$15	

0x00000044	bltz $12,-2 (<0,转 40)	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE
0x00000040	addiu $12,$12,1	000010	01100	01100	0000 0000 0000 0001	=	098C0001
0x00000044	bltz $12,-2	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE



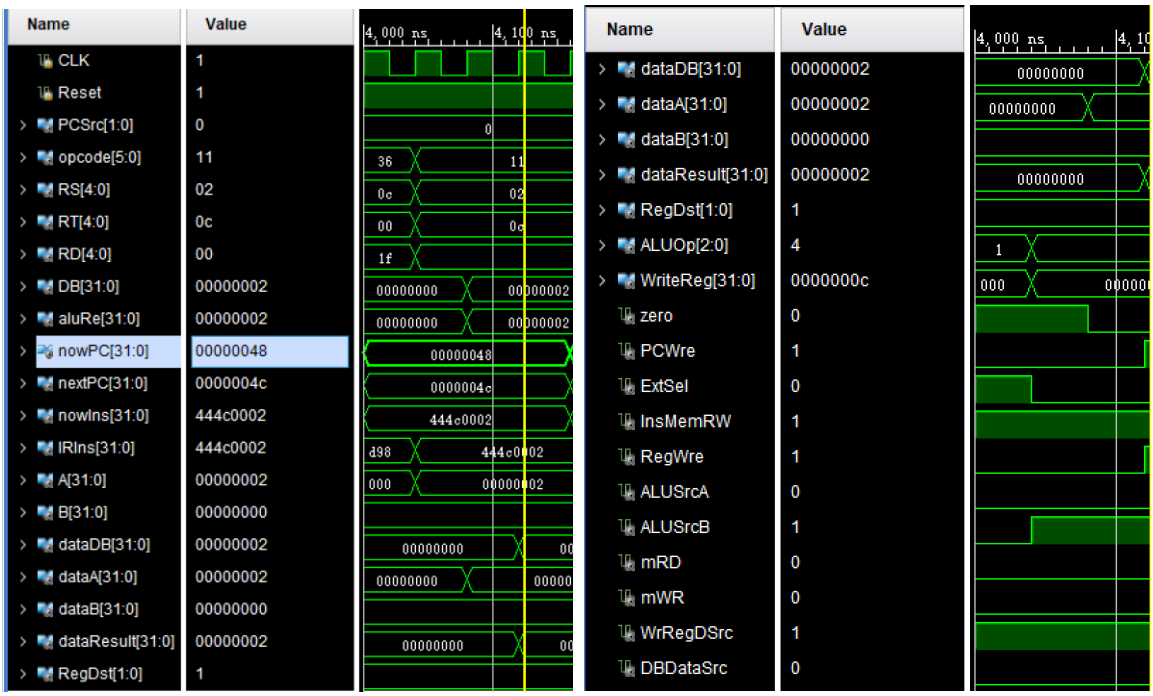


由于$12=-1<0，所以跳转到 0x00000040，$12=$12+1=0，此时不再进行跳转了，观察波形发现正确。

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9	0	$10	1
$11	2	$12	0	$13	2	$14	-2	$15	

0x00000048	andi $12,$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002

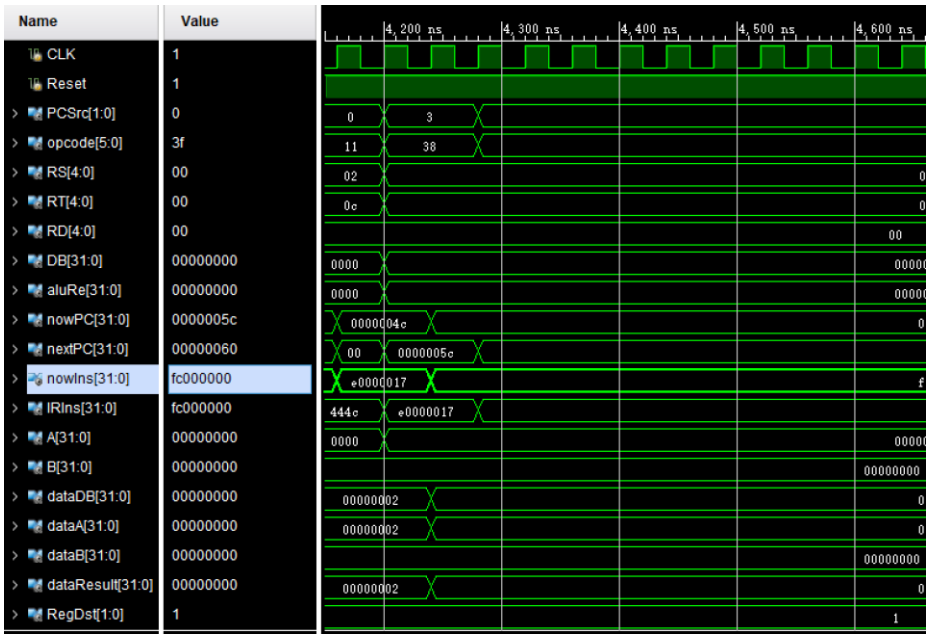


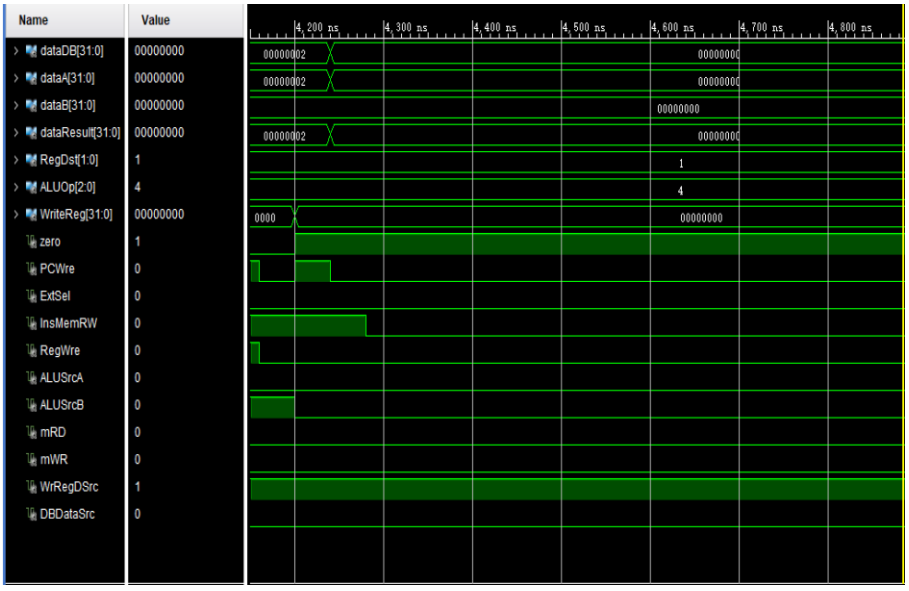
这条指令的目的是$12=$2$2=2;

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9	0	$10	1
$11	2	$12	2	$13	2	$14	-2	$15	

0x0000004C	j 0x000005C	111000	00000	00000	0000 0000 0101 1100	=	E000005C
0x0000005C	Halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000





这两条指令的作用是跳到 0x000005C，然后停机。

观察波形发现正确。

截至现阶段为止，各寄存器的值如下：

$1	8	$2	2	$3	10	$4	2	$5	32
$6		$7		$8	1	$9	0	$10	1
$11	2	$12	2	$13	2	$14	-2	$15	

Basys 板：

0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008

如图所示，当前指令为 00，下一条指令为 04，RS 为 00，RT 为 01，aluRE 为 08，writedata 为 08。

0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002

如图所示，当前指令为 04，下一条指令为 08，RS 为 00，RT 为 02，aluRE 为 02，writedata 为 02。

0x00000008	xori $3,$2,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008

如图所示，当前指令为 08，下一条指令为 0C，RS 为 02，RT 为 03，aluRE 为 0a，writedata

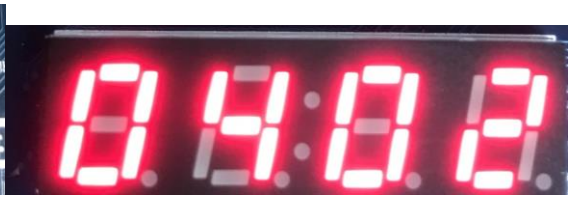
为 0a。

0x0000000C	sub $4,$3,$1	000001	00011	00001	00100 000 0000 0000	=	04612000




如图所示，当前指令为 0C，下一条指令为 10，RS 为 03，RT 为 01，aluRE 为 02，writedata 为 02。

0x00000010	and $5,$4,$2	010000	00100	00010	00101 000 0000 0000	=	40822800

如图所示，当前指令为 10，下一条指令为 14，RS 为 04，RT 为 02，aluRE 为 02，writedata 为 02。

0x00000014	sll $5,$5,2	011000	00000	00101	00101 00010 00 0000	=	60052880


如图所示，当前指令为 14，下一条指令为 18，RS 为 00，RT 为 05，aluRE 为 08，writedata 为 08。

实验心得

多周期实验终于在两周之内完成了，其实还是挺开心的。这次多周期的实验过程比单周期的顺利一些，做的时间也少了一些，但是里面遇到的困难还是很多的，其实我第一周就能够稍微将波形调出来了，但是老师说一定要有指令寄存器以及 ADR,BDR,ALUoutDR,DBDR 这些寄存器，于是乎，我就没能检查成，所以我干脆重新写了一遍，上网查资料，发现网上的好多错误，也不知道为什么他们当时能模拟出来，于是只能硬着脑皮自己写，发现很多模块跟单周期几乎一样，于是便直接套用了，还有就是我单周期的时候没有将指令划分，导致我各种各样的的 bug 漫天飞，于是这个多周期我决定加上了 INSDIV 模块，还有一个就是我舍弃了 MUX 模块，我直接在需要用要多选器的模块中选择，减少了模块的数量，同时也减少了 bug 发生的概率。

首先我遇到的致命的问题就是 PC 没写入的问题，跟很多人一样，就是因为寄存器的上升沿和下降沿的问题，这个我试了很多方案，找了很多的资料，发现寄存器上升沿和下降沿不能够随便的安排，需要十分小心的安排，最后我寄存器使用了上升沿之后波形顺利的出来了。

这次我遇到的第二个大 bug 是 beq 不进行跳转，原来是 ZERO 和 RESET 的问题，唉，我真的是十分的不认真，我总会在这些细节上面翻跟头，我一会按照自己的来，一会又按照课件上面来，所以导致我的信号是混乱的，所以出现了这样的 BUG。

接下来我又遇到了跟单周期一样的 bug，在 bne 的时候，跳转出现了死循环，不知道为什么死循环这样的错误总是缠着我，我一直以为是我的指令弄错了，最后我才知道原来是 ALU 结果不对，归根到底就是控制单元的问题，于是我解决了这个 bug，这让我意识到很多时候你觉得不会发生的问题往往是问题的源头，

于是我告诫自己不要急躁，一点点的对，幸运的是波形能够出来了，这就让我舒了一大口气。

最后在烧板子的时候，因为我忘了顶层文件，直接用的波形的文件，导致了很多的 bug，于是导致了第一次烧板子失败，第二次失败是申请的内存太大，不过最后也都解决了，所以也是很开心的。

还有一些其他的小错误：因为多周期 CPU 的难点在于控制单元的设计，分成了许多阶段，真值表规模变大，代码长度也随之变大，很容易出现打错，打漏的情况，这也导致我在调试的时候花费了很多时间。

大二计算机组成原理实验课课程总结：

大半个学期的计算机组成原理实验课终于来到了尾声，从刚开始的一无所知，但现在也还是了解了一些 Verilog 语言的基本语法，从刚开始的害怕到现在的也能勇敢的面对基础的 CPU 设计，这期间真的经历了很多的困难。不知道多少次因为找不到 BUG 而心累到不想做了，可是之后还是接着做起来，自己确实成熟了很多，我真的有好几次都处在了崩溃的边缘，所幸有一群帮我的同学和老师们帮我找错误，才让我顺利的完成了这门课程的学习。

计算机的内部构造真的是一件神奇的事情，同时也是十分精细的设计，一点都不能出错，出一点点错误都会导致整个的崩溃。其实我刚开始做 CPU 的时候内心是害怕的，因为内容实在是太庞杂了，需要记住的东西很多，我又不是一个细心的人，这导致我经常失败，使我的自信心有些受损，然后我发现其实细心也不是一件那么难的事情，我只需要拿一张纸将关键的信号，关键的步骤记录下来就可以了，于是乎，我慢慢的静了下来，我突然觉得能在椅子上坐一下午的人居然也能是我。

同时，在学习的过程中，我发现并没有什么教科书式的答案，很多方法都能够解决问题，只要思维对了，想法正确，就可以勇敢的去尝试，其结果不会让我失望的，每当看到自己的想法实现的时候，真的其实内心也不是高兴，只是觉得这是一种理所当然，因为这是合理的推理。

这学期真的是炼狱般的生活，每天都在忙碌着，忙碌着，有急躁也有想放弃，但是我突然想起来以前在书上看到的一句话“这个世界上只有两种人，一种懂二进制的人，一种不懂二进制的人。”觉得说的太好了，这简单的二进制码蕴含着多少智慧啊!这是多少人智慧的结晶！

课程结束了，我觉得在这门课上所学会的知识远远不是设计 CPU 这么简单，它让我知道了原来耐心，信心对研究是多么重要的一件事情，在以后的学习以及生活中，我都会记起这门课带给我的内心的平静，我们永远年轻，永远热泪盈眶，我们在路上！