



《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 : 袁之浩

学号 : 16340282

时间 : 2018 年 6 月 16 日

成 绩 :

实验二：多周期CPU设计与实现

一. 实验目的

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

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：

==>算术运算指令

- (1) add rd, rs, rt

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

功能: $rd \leftarrow rs + rt$

- (2) sub rd, rs, rt

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

完成功能: $rd \leftarrow rs - rt$

- (3) addi rt, rs, immediate

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

功能: $rt \leftarrow rs + (\text{sign-extend})\text{immediate}$

==>逻辑运算指令

- (4) or rd, rs, rt

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

功能: $rd \leftarrow rs | rt$

- (5) and rd, rs, rt

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

功能: $rd \leftarrow rs \& rt$

- (6) ori rt, rs, immediate

010010	rs(5 位)	rt(5 位)	immediate	
--------	---------	---------	-----------	--

功能: $rt \leftarrow rs | (\text{zero-extend})\text{immediate}$

==>移位指令

- (7) sll rd, rt, sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved
--------	----	---------	---------	----	----------

功能: $rd \leftarrow rt \ll (zero\text{-extend})sa$, 左移 sa 位 , $(zero\text{-extend})sa$

==>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 ($rs < (zero\text{-extend})immediate$) $rt = 1$ 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: $memory[rs + (sign\text{-extend})immediate] \leftarrow rt$ 。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: $rt \leftarrow memory[rs + (sign\text{-extend})immediate]$ 。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($rs = rt$) $pc \leftarrow pc + 4 + (sign\text{-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

(13) bltz rs, immediate

110110	rs(5 位)	00000	immediate
--------	---------	-------	-----------

功能: if($rs < 0$) $pc \leftarrow pc + 4 + (sign\text{-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

==>跳转指令

(14) j addr

111000	addr[27:2]
--------	------------

功能: $pc \leftarrow \{(pc+4)[31:28],addr[27:2],2'b00\}$, 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: $pc \leftarrow rs$, 跳转。

==>调用子程序指令

(16) jal addr

111010	addr[27:2]
--------	------------

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

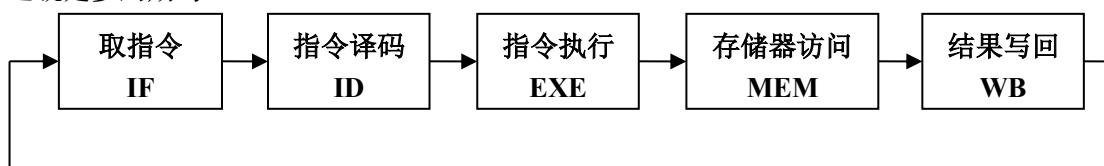


图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式：

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。

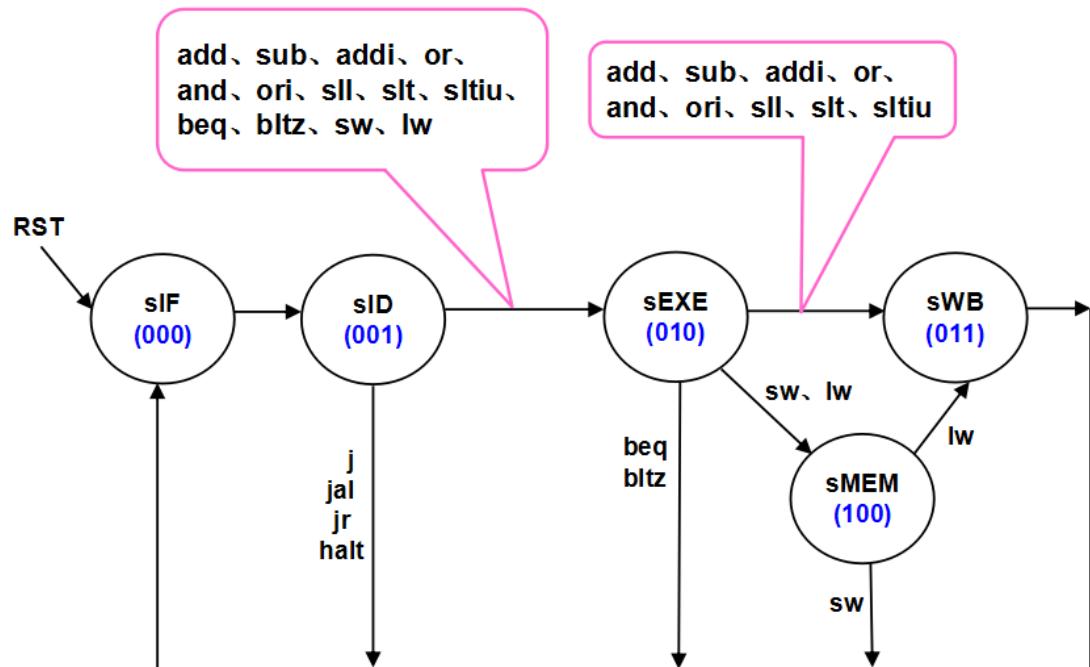


图 2 多周期 CPU 状态转移图

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。

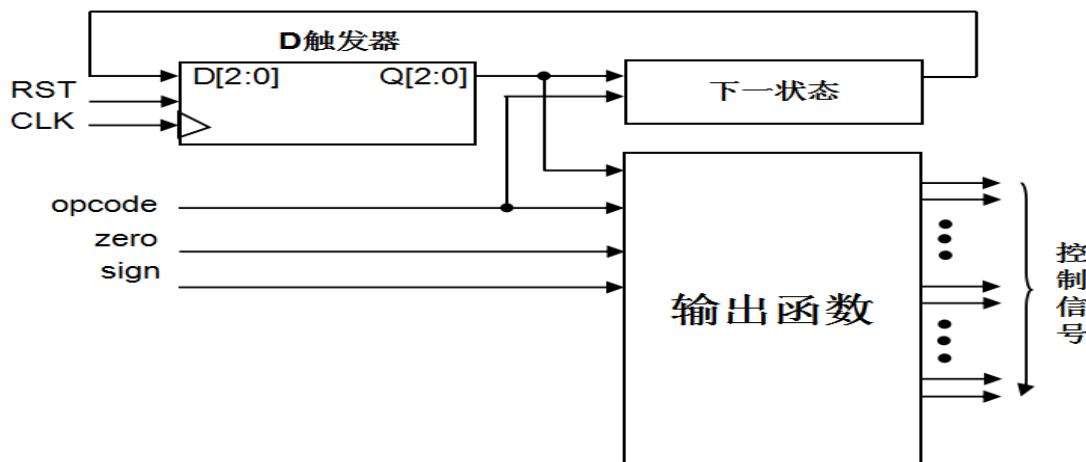


图 3 多周期 CPU 控制部件的原理结构图

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。

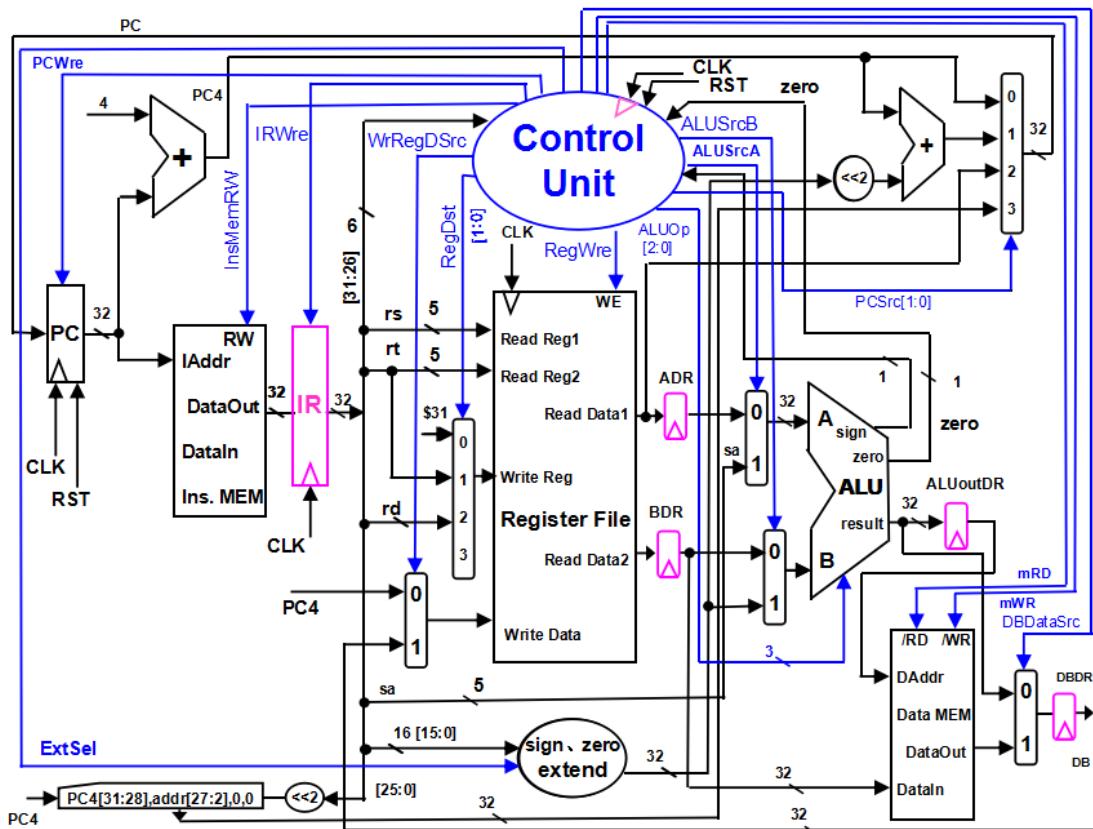


图 4 多周期 CPU 数据通路和控制线路图

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、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、bltz、slt、sll	来自 sign 或 zero 扩展的立即数, 相关指令: addi、ori、sltiu、lw、sw
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	$Y = (((regA < regB) \&\& (regA[31] == regB[31])) ((regA[31] == 1 \&\& regB[31] == 0))) ? 1 : 0$	比较 A 与 B 带符号
100	$Y = B \ll A$	B 左移 A 位
101	$Y = A \vee B$	或
110	$Y = A \wedge B$	与
111	$Y = A \Box B$	异或

四. 实验器材

电脑一台, Xilinx Vivado 软件一套, Basys3板一块。

五. 实验过程与结果

1、CPU设计

多周期的cpu设计思路与单周期相同, 依然是先设计每个模块, 再在顶层模块中合到一起。

pc模块：这次将pc和四选一合在了一起写，PCWre控制是否读取下一条指令，PCSrc为控制选位信号

```

module PC(input clk, Reset, PCWre,
           input [1:0] PCSrc,
           input wire [31:0] imm, addr, RDout1,
           output reg [31:0] Address);

    always @(posedge clk) begin // 00000000000000000000000000000000
        if (Reset == 0) begin
            Address = 0;
        end else if (PCWre) begin
            if (PCSrc == 2'b00) begin
                Address = Address+4;
            end else if (PCSrc == 2'b01) begin
                Address = imm*4+Address+4;
            end else if (PCSrc == 2'b10) begin
                Address = RDout1;
            end else if (PCSrc == 2'b11) begin
                Address = addr;
            end
        end
    end
endmodule

```

PCAAddr模块，用于立即数跳转指令

```

module PCAAddr(input [25:0] in_addr,
               input [31:0] PC0,
               output reg [31:0] addr);
    wire [27:0] mid;
    assign mid = in_addr << 2;
    always @(in_addr or PC0 or mid) begin
        addr <= {PC0[31:28], mid[27:0]};
    end
endmodule

```

InsMemory：指令存储器，需要注意的是读取时的触发信号，将指令和指令寄存器合在一起，触发沿不同

```
module InsMemory(input [31:0] addr,
                  input InsMemRW, IRWre, clk,
                  output reg [31:0] ins);

    reg [31:0] ins_out;
    reg [7:0] mem [0:127];

    initial begin
        $readmemb("D:/vivado/MultiPeriodCPU_basys/store.txt", mem);
        //ins_out = 0;
    end

    always @(negedge clk) begin
        if (InsMemRW) begin
            ins_out[31:24] = mem[addr];
            ins_out[23:16] = mem[addr+1];
            ins_out[15:8] = mem[addr+2];
            ins_out[7:0] = mem[addr+3];
        end
    end

    always @(posedge clk) begin
        if (IRWre) ins <= ins_out;
    end

endmodule
```

RegFile: 寄存器组，这里也将选位合在了一起

```

module RegFile(input [4:0] rs, rt, rd,
               input clk, RegWre, WrRegDSrc,
               input [1:0] RegDst,
               input [31:0] PC4, memData,
               output reg [31:0] data1, data2);

    reg [31:0] i_data;

    reg [4:0] temp;

    reg [31:0] register [0:31];
    integer i;
    initial begin
        for (i = 0 ; i < 32; i = i+1)
            register[i] = 0;
    end

    always @(negedge clk) begin
        case(RegDst)
            2'b00: temp = 5'b11111;
            2'b01: temp = rt;
            2'b10: temp = rd;
            default temp = 0;
        endcase

        i_data = WrRegDSrc? memData : PC4;
        data1 = register[rs];
        data2 = register[rt];

        if ((temp != 0) && (RegWre == 1)) begin //
            register[temp] <= i_data;
        end
    end

```

DataLate: 数据延迟, 这是多周期的特点, 为了复用各个单元, 加入寄存器, 存储上次计算的值用于下一个段, 这个模块可以多次使用。

```

module DataLate(input [31:0] i_data,
                input clk,
                output reg [31:0] o_data);

    always @(posedge clk) begin
        o_data = i_data;
    end

endmodule

```

Extend: 数据位拓展, 和上次类似, 根据ExtSel, 确定如何补位

```

module Extend(input [15:0] in_num,
              input [1:0] ExtSel,
              output reg [31:0] out);

    always @(in_num or ExtSel) begin
        case(ExtSel)
            2'b00: out <= {{27{0}}, in_num[10:6]]; // 00 sa
            2'b01: out <= {{16{0}}, in_num[15:0]]; // 00000000 0
            2'b10: out <= {{16{in_num[15]}}, in_num[15:0]]; // default: out <= {{16{in_num[15]}}, in_num[15:0]};
        endcase
    end

endmodule

```

ALU: 计算单元和上次一样

```

module ALU(input [31:0] ReadData1, ReadData2, inExt,
           input ALUSrcA, ALUSrcB,
           input [2:0] ALUOp,
           output wire zero,
           output wire sign,
           output reg [31:0] result);

    initial begin
        result = 0;
    end

    wire [31:0] A, B;
    assign A = ALUSrcA? inExt : ReadData1;
    assign B = ALUSrcB? inExt : ReadData2;
    assign zero = (result? 0 : 1);
    assign sign = result[31];

    always @(A or B or ALUOp) begin
        case(ALUOp)
            3'b000: result = A + B; // A + B
            3'b001: result = A - B; // A - B
            3'b010: result = (A < B ? 1 : 0); // 00AB
            3'b011: result = (((A < B) && (A[31]==B[31])) || ((A[31]==1&&B[31]==0)))?1:0;
            3'b100: result = B << A; // ABBB
            3'b101: result = A | B; // 0
            3'b110: result = A & B; // 0
            3'b111: result = (~A & B) | (A & ~B); // 00
        default: result = 0;
        endcase
    end

endmodule

```

controlUnit: 控制单元，和上次区别很大，先确定控制器有几个状态，需要存储这次的状态和下一次的状态。

```

parameter [2:0] sif = 3'b000, // IF state
              sid = 3'b001, // ID state
              exe = 3'b010, // EXE state
              smem = 3'b100, // MEM state
              swb = 3'b011; // WB state

```

在时钟下降沿的时候改变一次状态

```
always @(negedge clk) begin
    if (Reset == 0) begin
        state <= sif;
    end else begin
        state <= next_state;
    end
    state_out = state;
end
```

根据当前状态和读到的指令确定下一状态

```
always @(state or opcode) begin
    case(state)
        sif: next_state = sid;
        sid: begin
            case (opcode[5:3])
                3'b111: next_state = sif; // j, jal, jr, halt@@
                default: next_state = exe; // else
            endcase
        end
        exe: begin
            case (opcode)
                6'b110100: next_state = sif; //beq
                6'b110110: next_state = sif; //bltz
                6'b110000: next_state = smem; //sw
                6'b110001: next_state = smem; //lw
                default: next_state = swb; //else
            endcase
        end
        smem: begin
            if (opcode == 6'b110001) next_state = swb; // lw@@
            else next_state = sif; // sw@@
        end
        default: next_state = sif;
    endcase
end
```

最后根据状态，操作码，还有ALU返回的信号确定各种控制信号的值。

```
always @(state or opcode or zero or sign) begin
    // @@PCWre@@
    if (state == sif && opcode != halt) PCWre = 1;
    else PCWre = 0;

    // @@InsMemRW@@
    InsMemRW = 1;
```

2、波形图说明

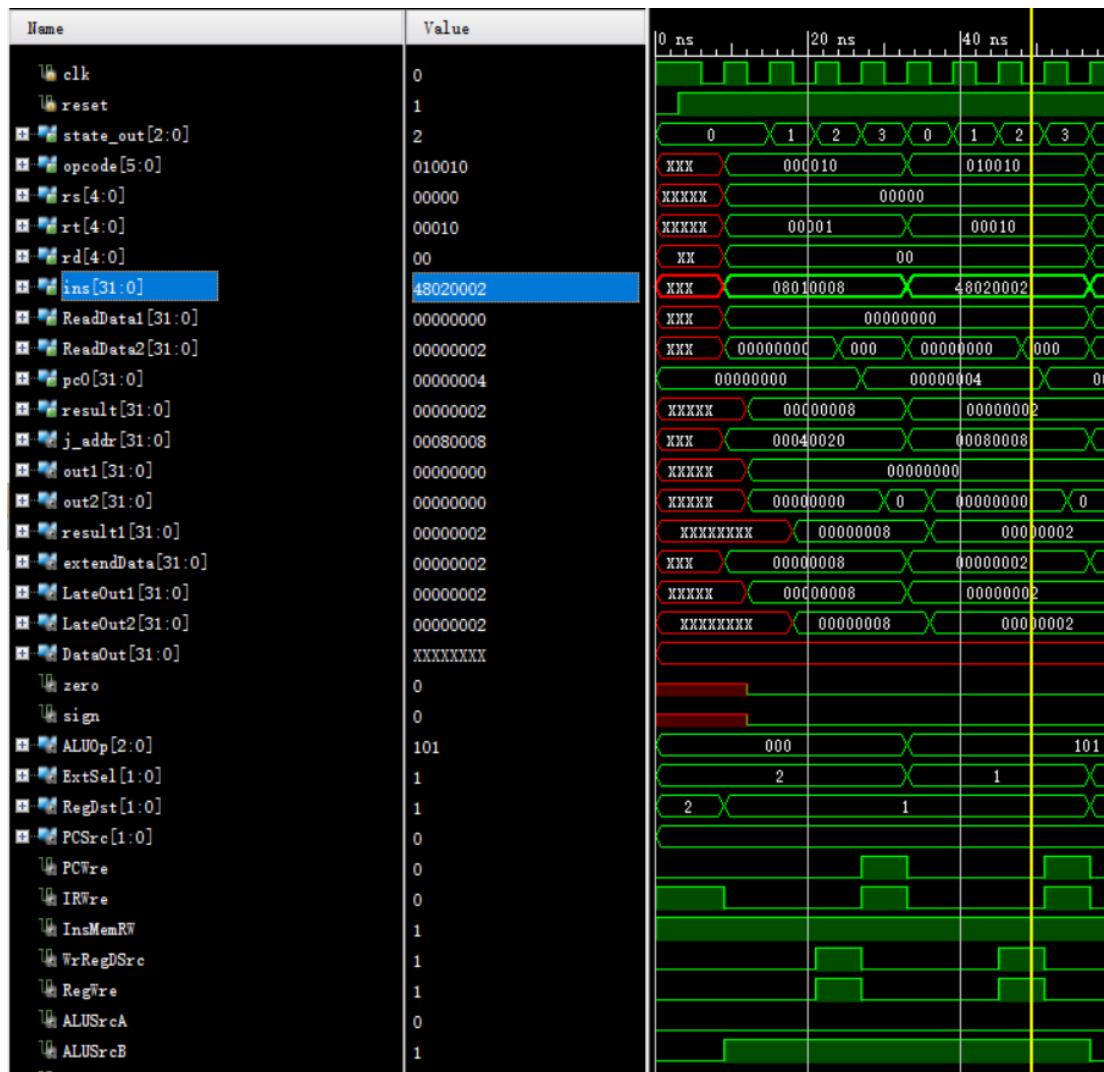
地址	汇编程序	指令代码
----	------	------

		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



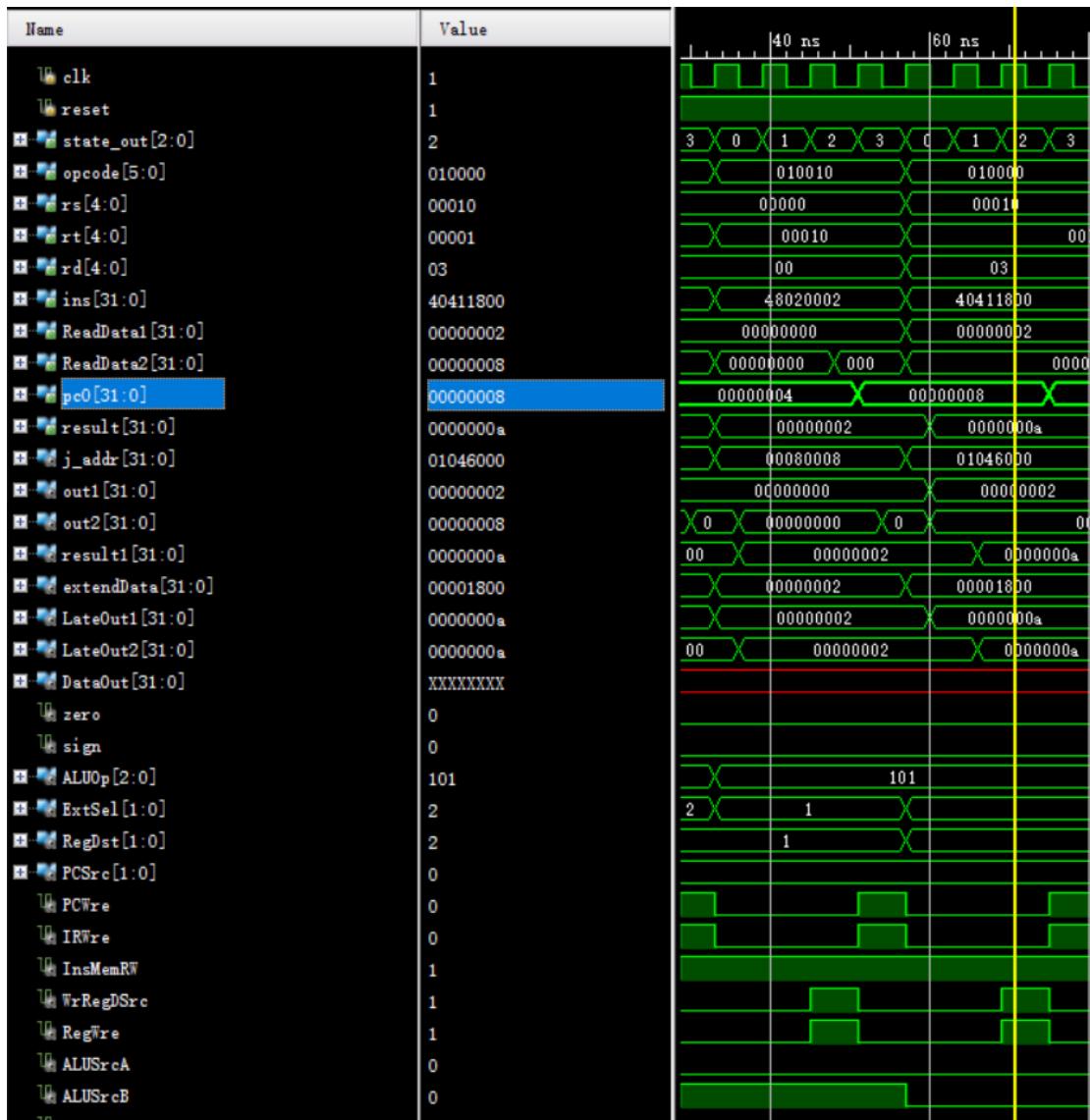
指令地址为pc0，值为00000000，InsMemory输出ins的值为08010008，rs的值为00000，寄存器\$0，rt的值为00001，寄存器\$1，立即数经过符号拓展后extendData的值为00000008，ALUSrcA为0，选择out1，ALUSrcB为1，选择extendData，操作码ALUOp为000，执行加法运算，计算结果result为00000008，经过一个二选一和ALUM2DR，到regfile的输入端lateOut2，WrRegDSrc的值为1，写入寄存器\$1.

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



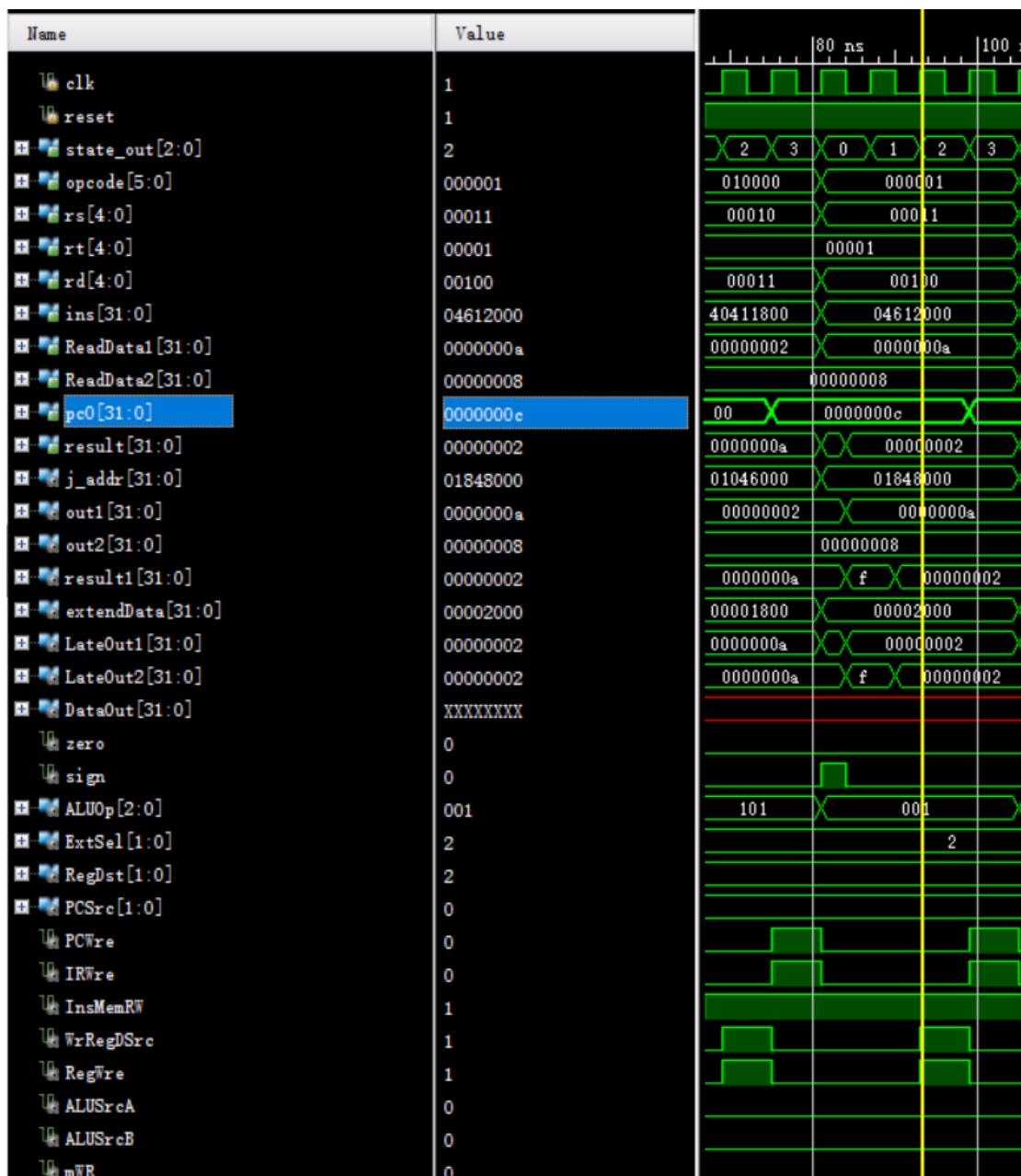
指令地址为pc0，值为00000004，InsMemory输出ins的值为48020002，rs的值为00000，寄存器\$0，rt的值为00010，寄存器\$2，立即数经过符号拓展后extendData的值为00000002，ALUSrcA为0，选择out1，ALUSrcB为1，选择extendData，操作码ALUOp为101，执行或操作，结果result为00000002，经过一个二选一和ALUM2DR，到regfile的输入端memData，WrRegDSrc的值为1，写入寄存器\$2.

0x00000008	or \$3,\$2,\$1	010000	00010	00001	00011		40411800
------------	----------------	--------	-------	-------	-------	--	----------



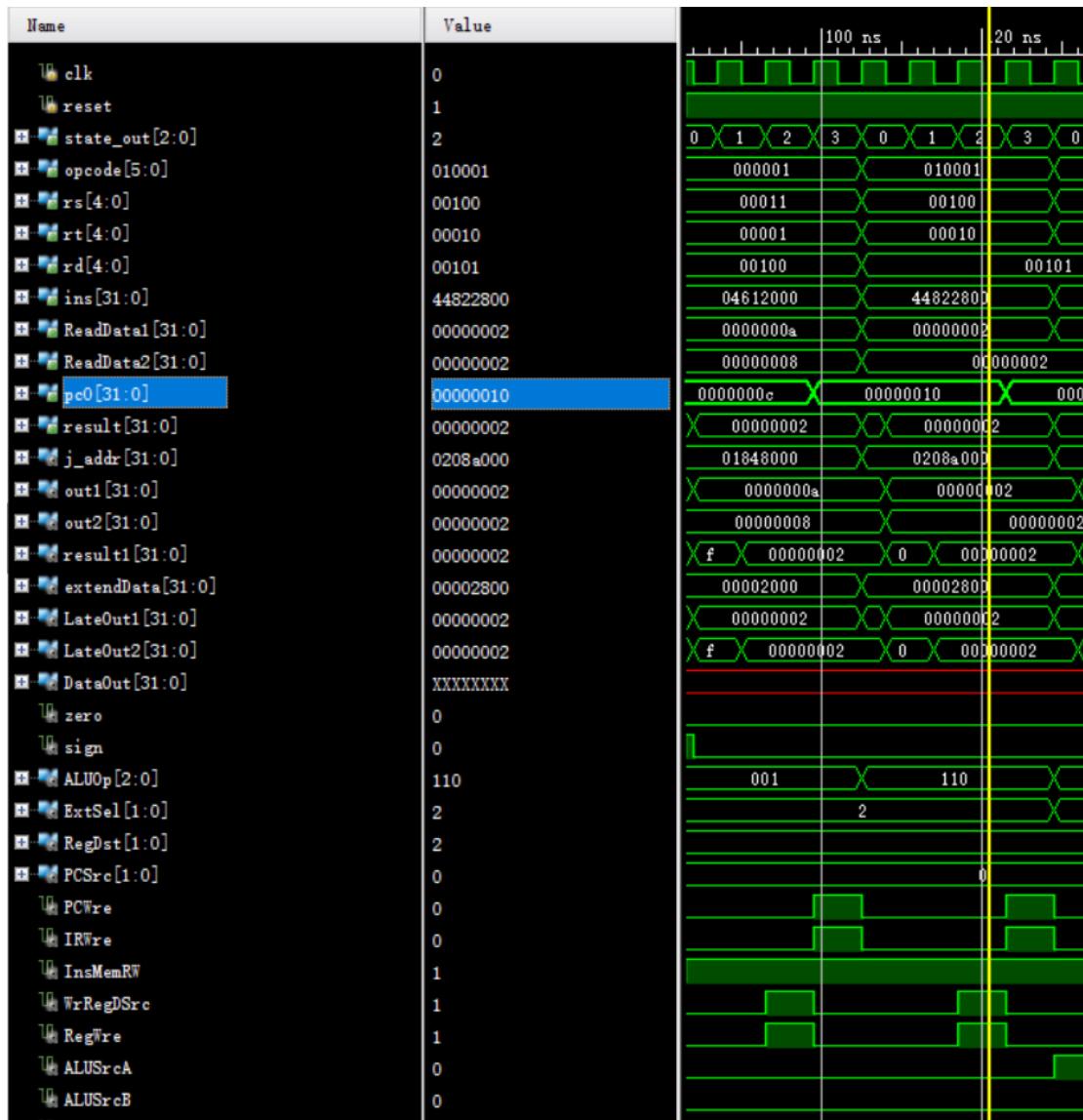
指令地址为pc0，值为00000008，InsMemory输出ins的值为40411800，rs的值为00010，寄存器\$2，rt的值为00001，寄存器\$1，rd的值为00011，寄存器\$3，\$2的值输出到out1，为00000002，\$1的值输出到out2，为00000008，ALUSrcA为0，选择out1，ALUSrcB为0，选择out2，操作码ALUOp为101，执行或操作，结果result为0000000a，经过一个二选一和ALUM2DR，到regfile的输入端memData，WrRegDSrc的值为1，写入寄存器\$3.

0x0000000C	sub \$4,\$3,\$1	000001	00011	00001	00100		04612000
------------	-----------------	--------	-------	-------	-------	--	----------



指令地址为pc0，值为0000000c，InsMemory输出ins的值为04621200，rs的值为00011，寄存器\$3，rt的值为00001，寄存器\$1，rd的值为00100，寄存器\$4，\$3的值输出到out1，为0000000a，\$1的值输出到out2，为00000008，ALUSrcA为0，选择out1，ALUSrcB为0，选择out2，操作码ALUOp为001，执行减操作，结果result为00000002，经过一个二选一和ALUM2DR，到regfile的输入端memData，WrRegDSrc的值为1，写入寄存器\$4.

0x00000010	and \$5,\$4,\$2	010001	00100	00010	00101		44822800
------------	-----------------	--------	-------	-------	-------	--	----------



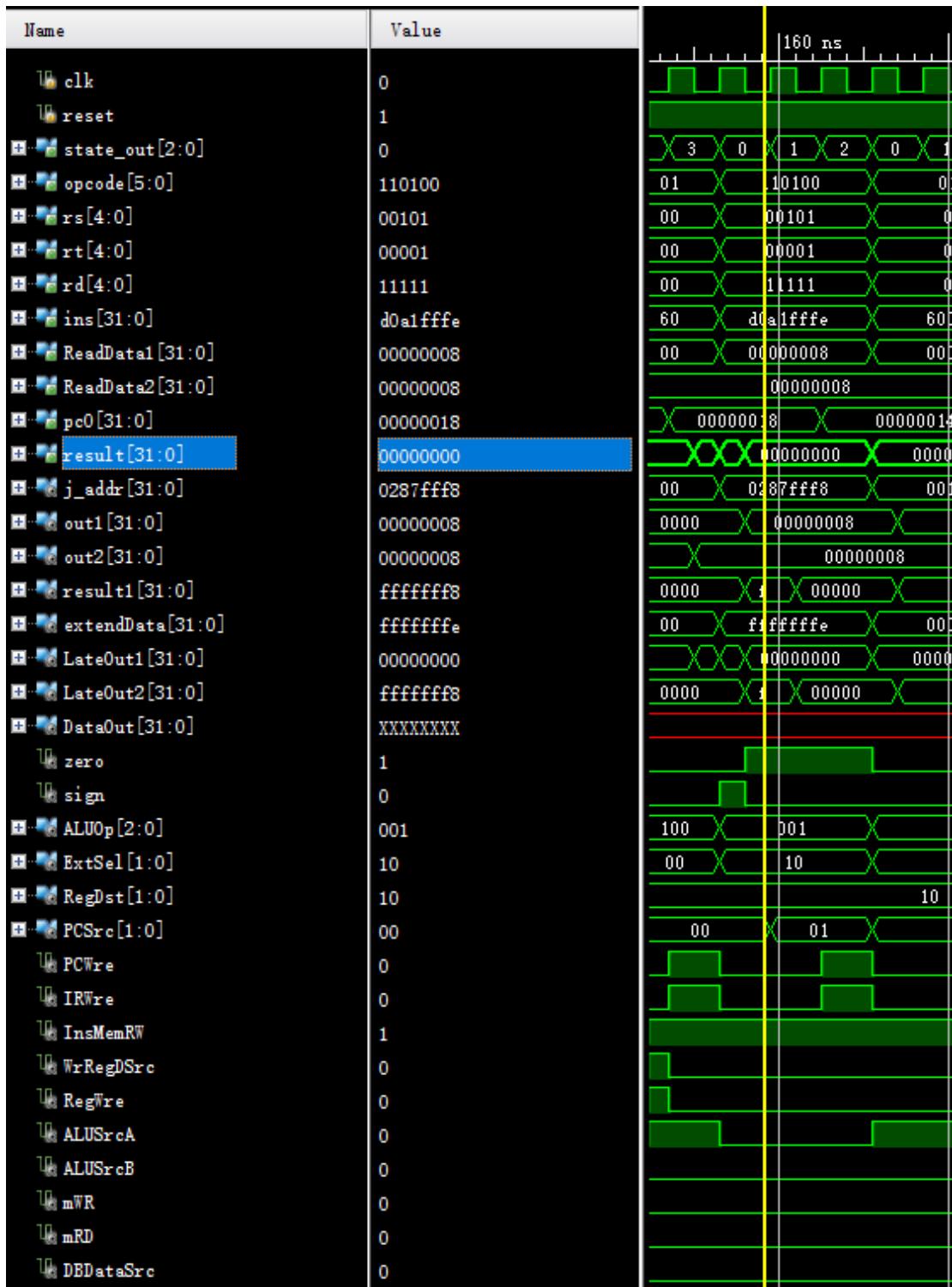
指令地址为pc0，值为00000010，InsMemory输出ins的值为04621200，rs的值为00100，寄存器\$4，rt的值为00010，寄存器\$2，rd的值为00101，寄存器\$5，\$4的值输出到out1，为00000002，\$2的值输出到out2，为00000002，ALUSrcA为0，选择out1，ALUSrcB为0，选择out2，操作码ALUOp为110，执行与操作，结果result为00000002，经过一个二选一和ALUM2DR，到regfile的输入端LateOut2，WrRegDSrc的值为1，写入寄存器\$5.

0x00000014	sll	\$5,\$5,2	011000		00101	00101 00010		60052880
------------	-----	-----------	--------	--	-------	-------------	--	----------



指令地址为pc0，值为00000014，InsMemory输出ins的值为60052880，rd的值为00101，寄存器\$5，rt的值为00101，寄存器\$5，sa经过符号拓展后extendData的值为00000002，ALUSrcA为1，选择extendData，ALUSrcB为0，选择out2，操作码ALUOp为100，执行移位操作，结果result为00000008，经过一个二选一和ALUM2DR，到regfile的输入端memData，WrRegDSrc的值为1，写入寄存器\$5.

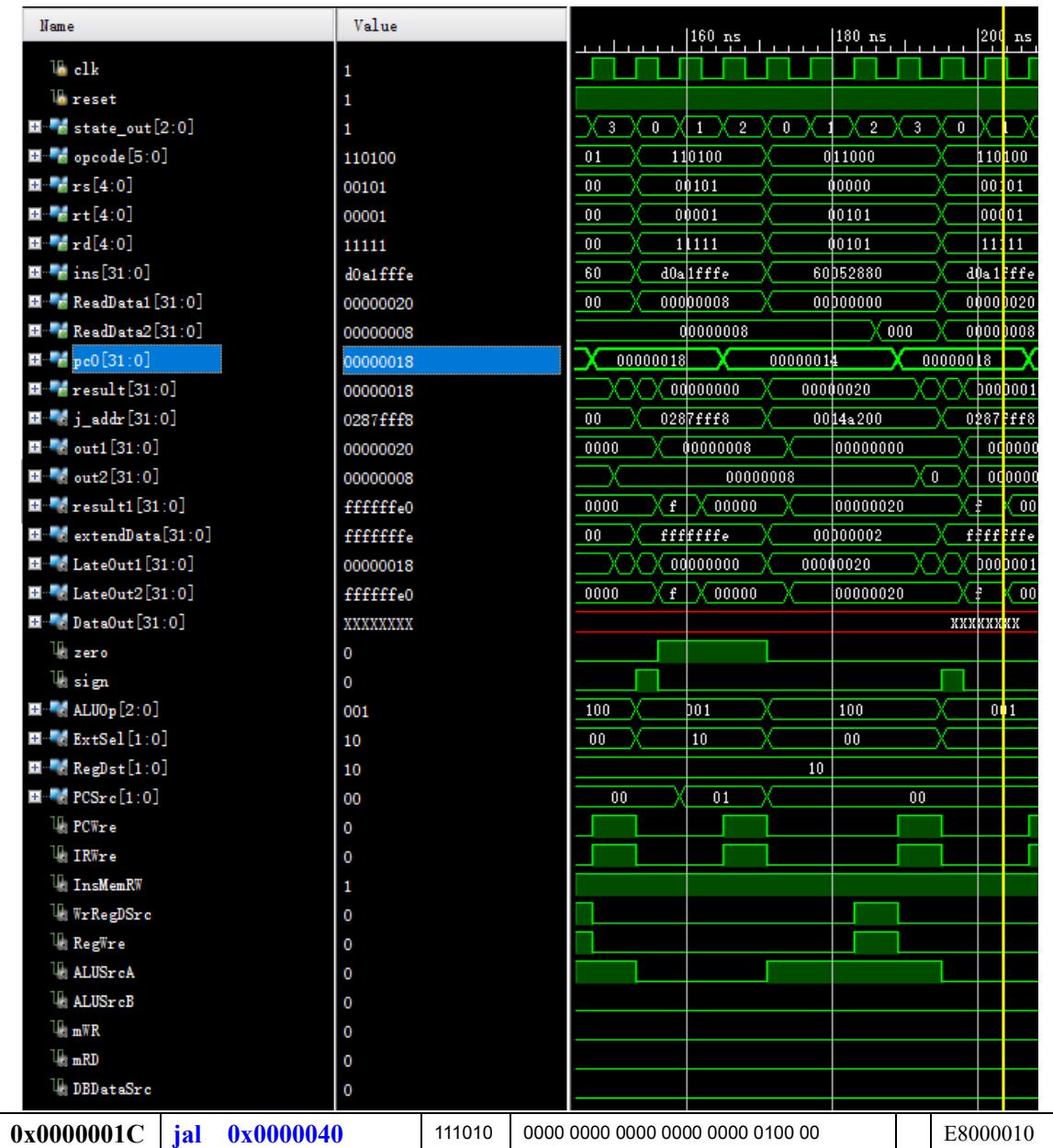
0x00000018	beq \$5,\$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110		D0A1FFFE
------------	------------------------	--------	-------	-------	---------------------	--	----------

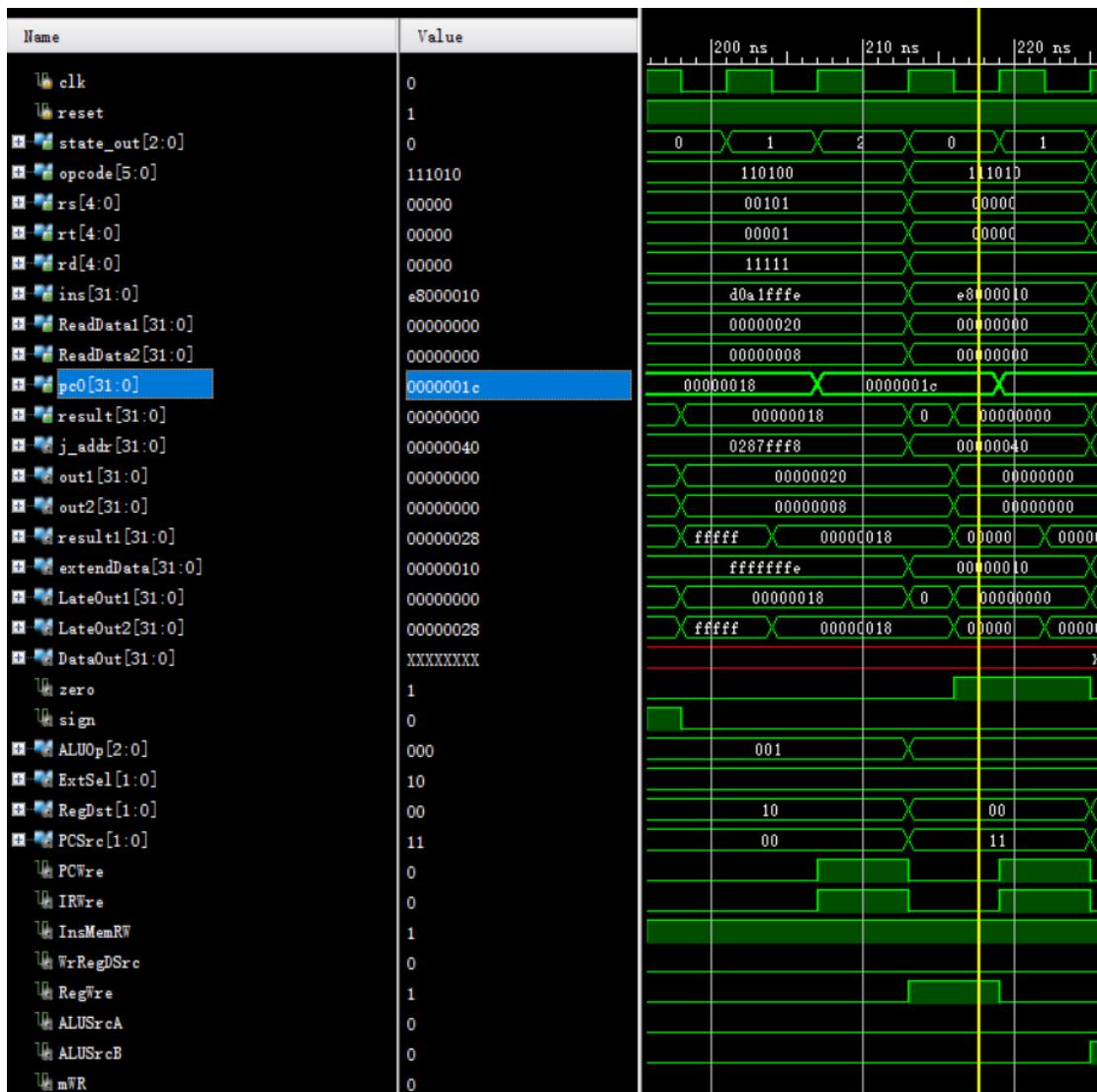


指令地址为pc0，值为00000018，InsMemory输出ins的值为d0a1ffffe，rs的值为00101，寄存器\$5，rt的值为00001，寄存器\$1，立即数经过符号拓展后extendData的值为fffffffffe，\$5的值00000008送入out1，\$1的值00000008送入out2，ALUSrcA为0，选择out1，ALUSrcB为0，选择out2，操作码ALUOp为001，执行减操作，结果result为00000000，zero标志位为1，表示相等，下一个状态PCSrc为01，pc模块选择pc+4+立即

数左移两位后的值，pc0变为00000014.进入循环。

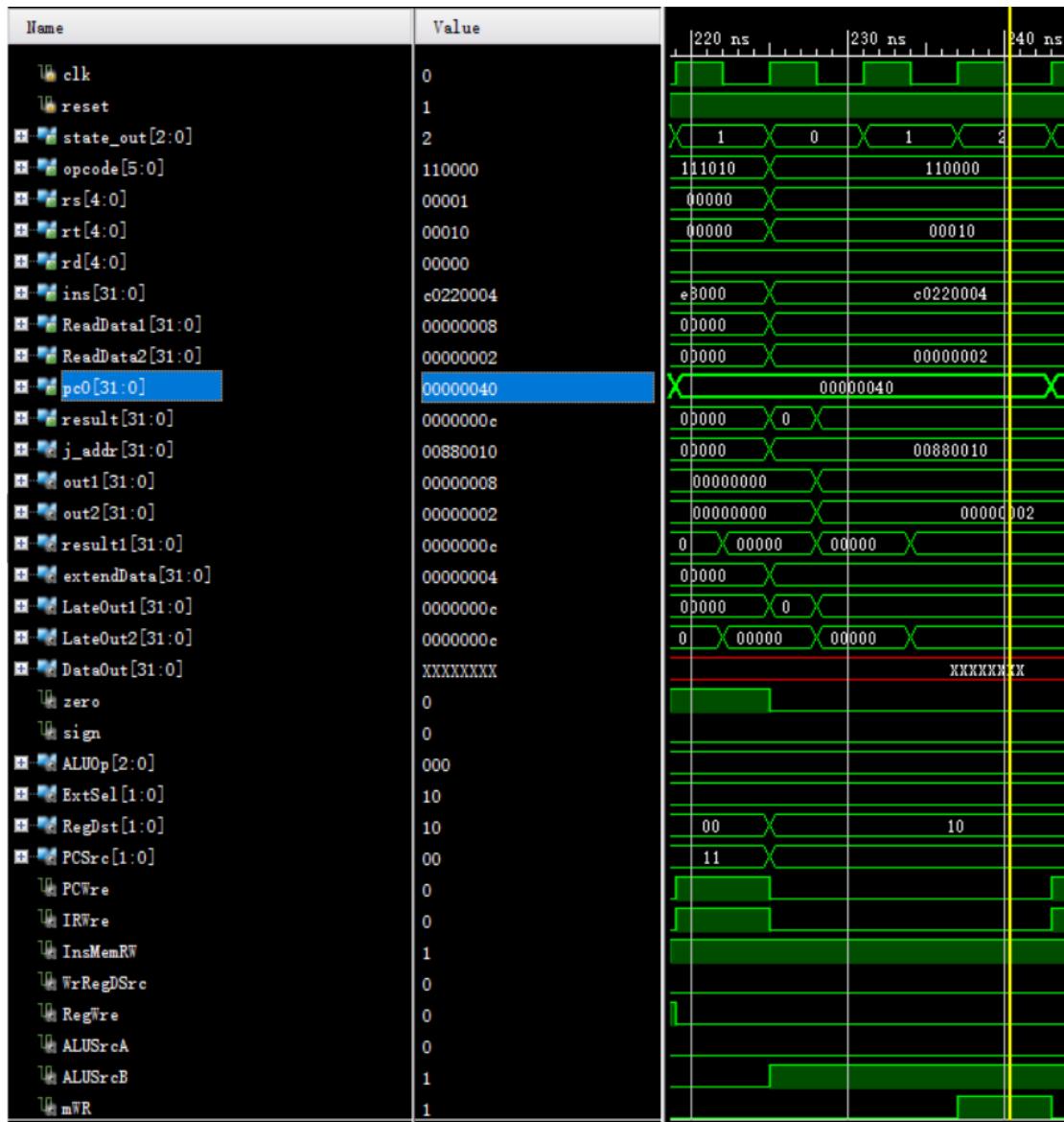
循环结果





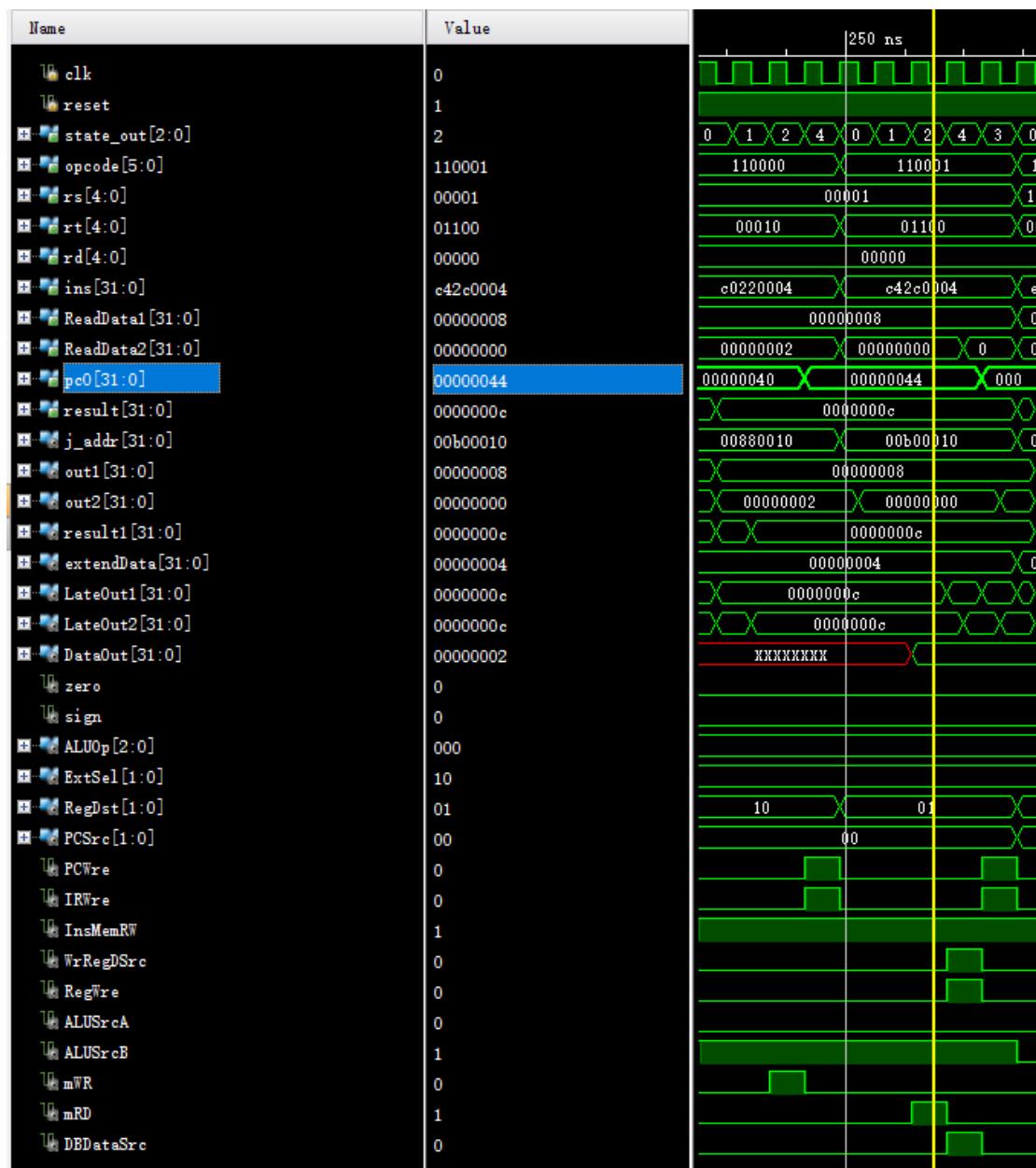
指令地址为pc0，值为0000001c，当前地址和输入地址送入PCAddr模块，经过处理后j_addr输出00000040为下一条指令的地址，再送入PC， PCSrc的值为11，选择送入的j_addr。下一条指令将跳转到00000040

0x00000040	SW \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100	C0220004
-------------------	---------------	--------	-------	-------	---------------------	----------



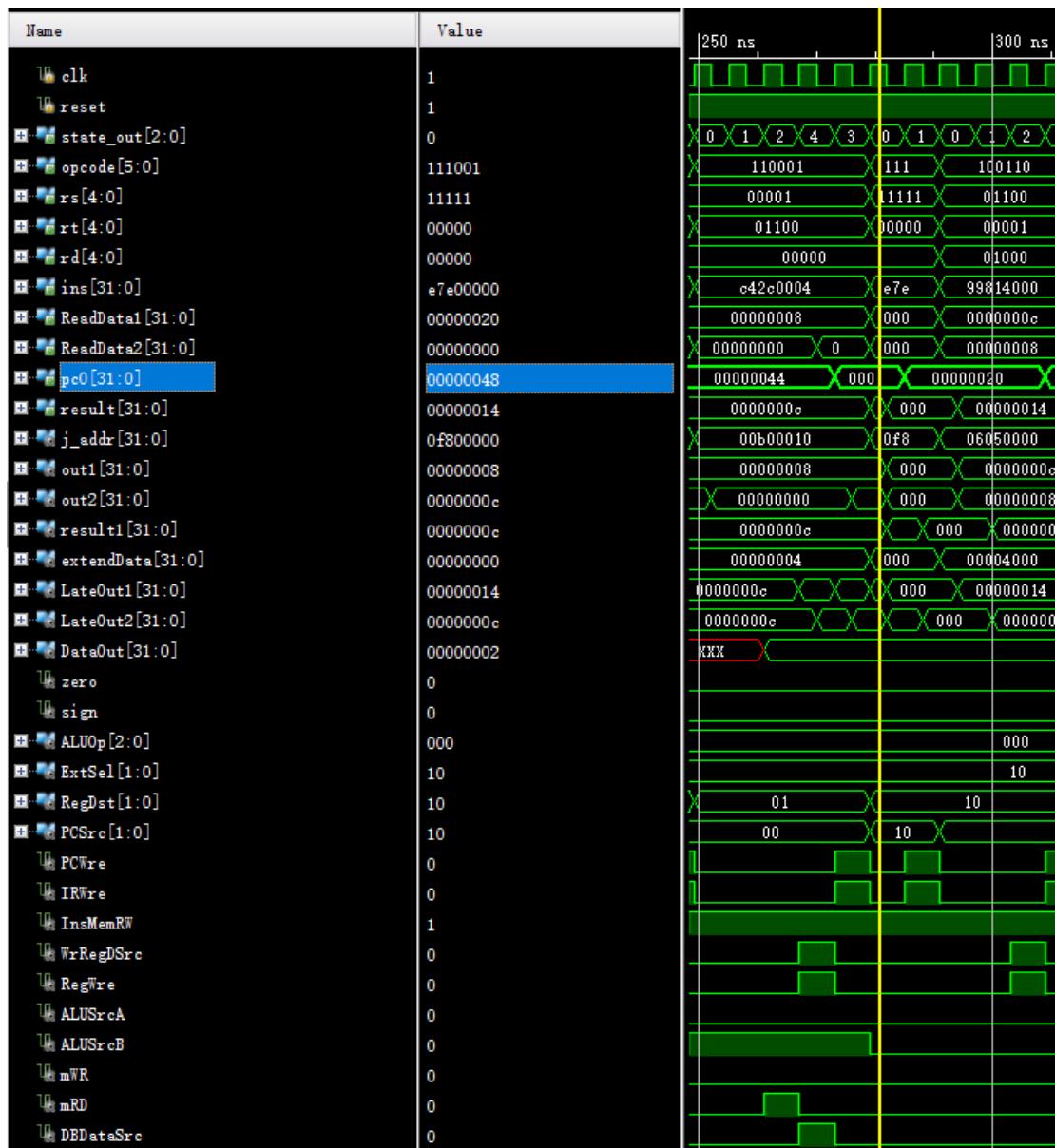
指令地址为pc0，值为00000040，ins的值为c0220004，rs的值为00001，即寄存器\$1，输出到out1，rt的值为00010，即寄存器\$2，输出到out2立即数经符号拓展后的值为00000004，ALUSrcA的值为0，选择out1，ALUSrcB的值为1，选择extendData，ALUOp的值为000，执行加操作，result为0000000c，送入datamemory模块作为地址，out2作为数据，mWR为1，写入存储器。

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



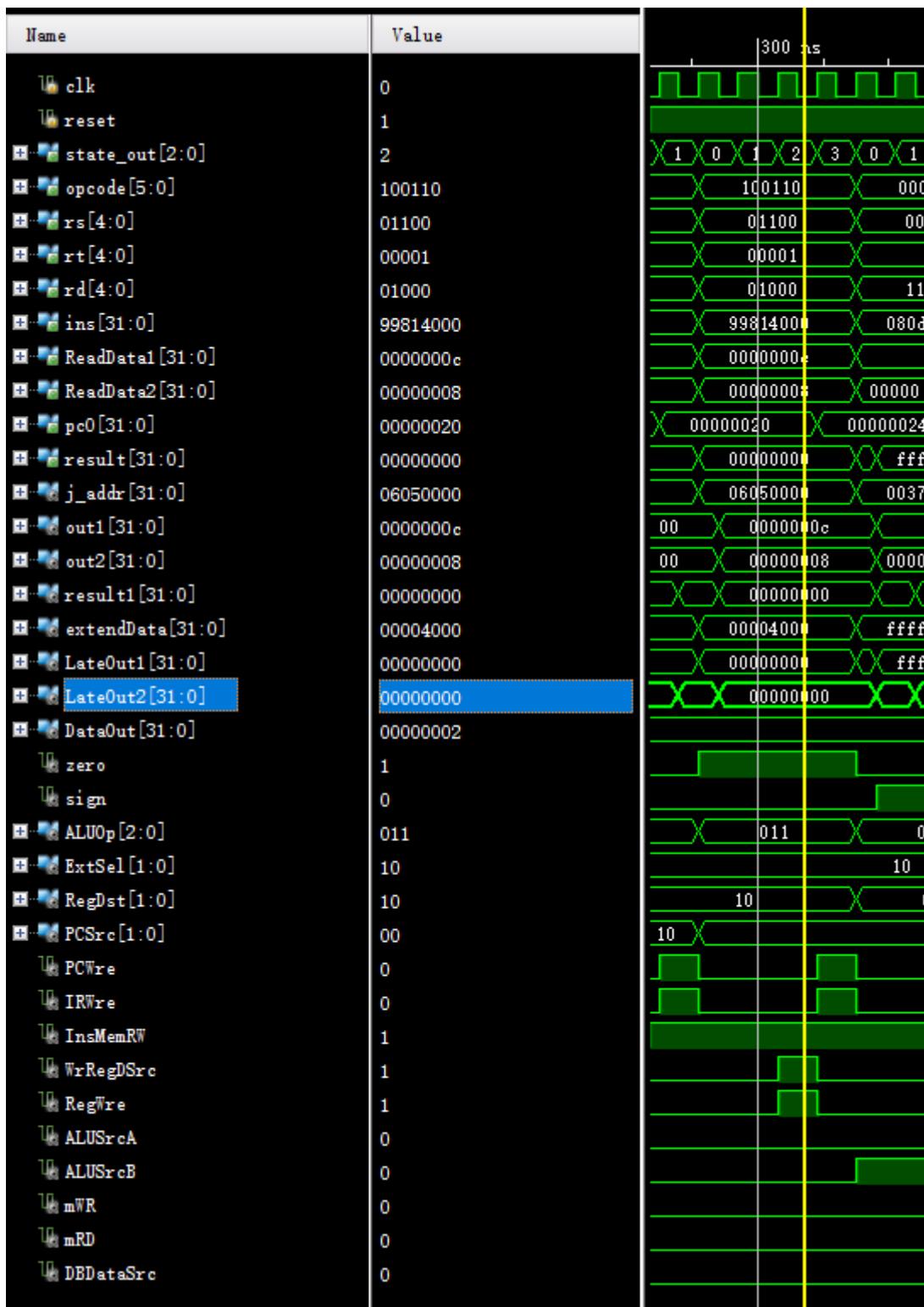
指令地址为pc0，值为00000044，ins的值为c42c0004，rs的值为00001，即寄存器\$1，输出到out1，rt的值为01100，即寄存器\$12，输出到out2，立即数经符号拓展后的值为00000004，ALUSrcA的值为0，选择out1，ALUSrcB的值为1，选择extendData，ALUOp的值为000，执行加操作，result为0000000c，送入datamemory模块作为地址，mRD为1，存储器输出DataOut，值为00000002，送入到regfile，RegDst为01，RegWre为1，WrRegDSrc为1，在下一阶段，选择rt的寄存器进行写入，写入的值为LateOut，00000002。

0x00000048	jr \$31	111001	11111			E7E00000
------------	---------	--------	-------	--	--	----------



指令地址为pc0，值为00000048，ins的值为e7e00000,rs的值为11111，即寄存器\$31，值输出到ReadData1，为00000020，输出到pc模块，PCSrc为10，选择readData1作为下一条指令pc0的值。

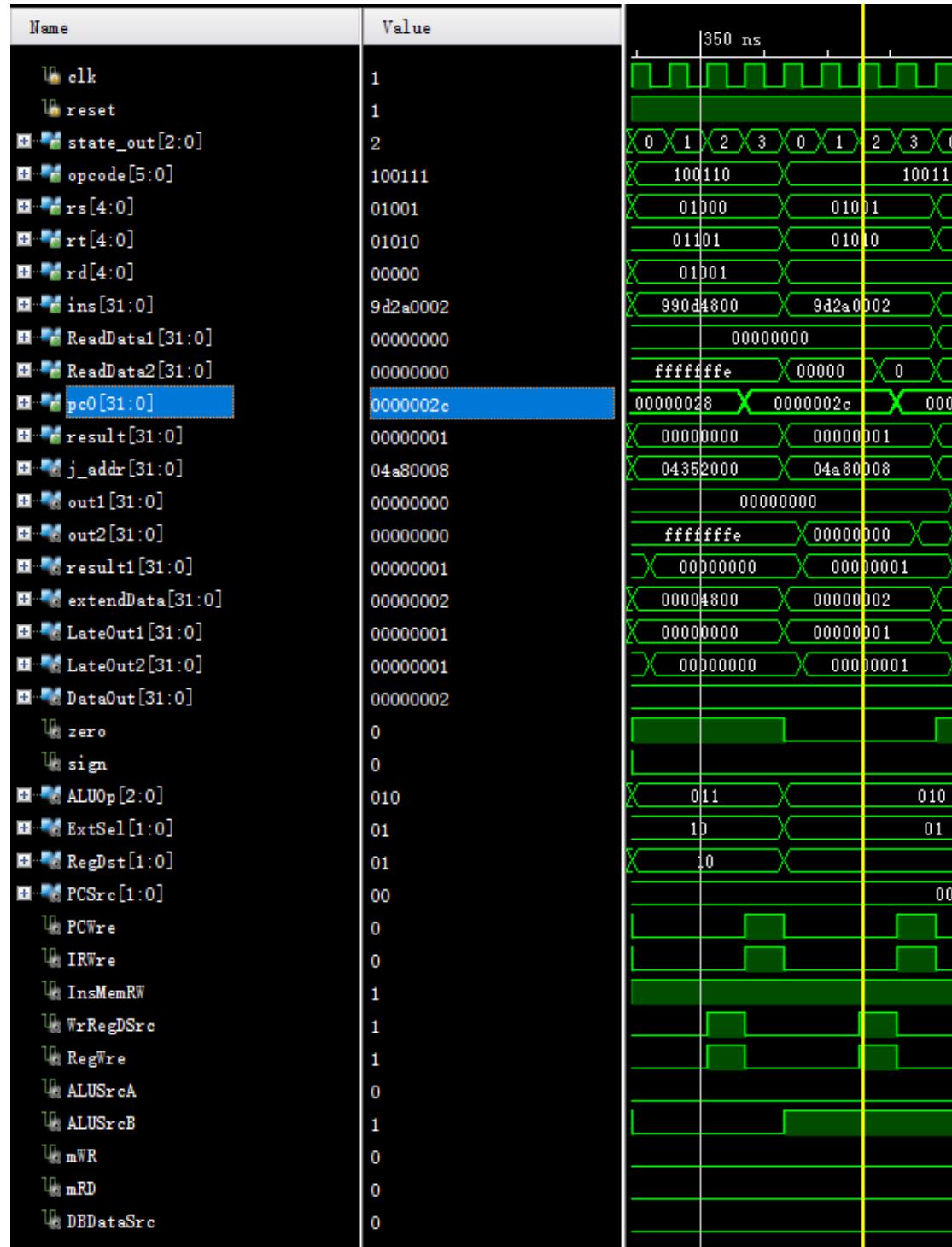
0x00000020	slt \$8,\$12,\$1	100110	01100	00001	01000		99814000
------------	------------------	--------	-------	-------	-------	--	----------



指令地址为pc0，值为00000020，InsMemory输出ins的值为99814000，rs的值为01100，寄存器\$12，rt的值为00001，寄存器\$1，rd的值为01000，寄存器\$8，\$12的值输出到out1，为0000000c，\$1的值输出到out2，为00000008，ALUSrcA为0，选择out1，ALUSrcB为0，选择out2，操作码ALUOp为110，执行带符号比较操作，结果result为00000000，经过一个二选一和ALUM2DR，到regfile的输入端LateOut2，WrRegDSrc的

值为1，写入寄存器\$8.

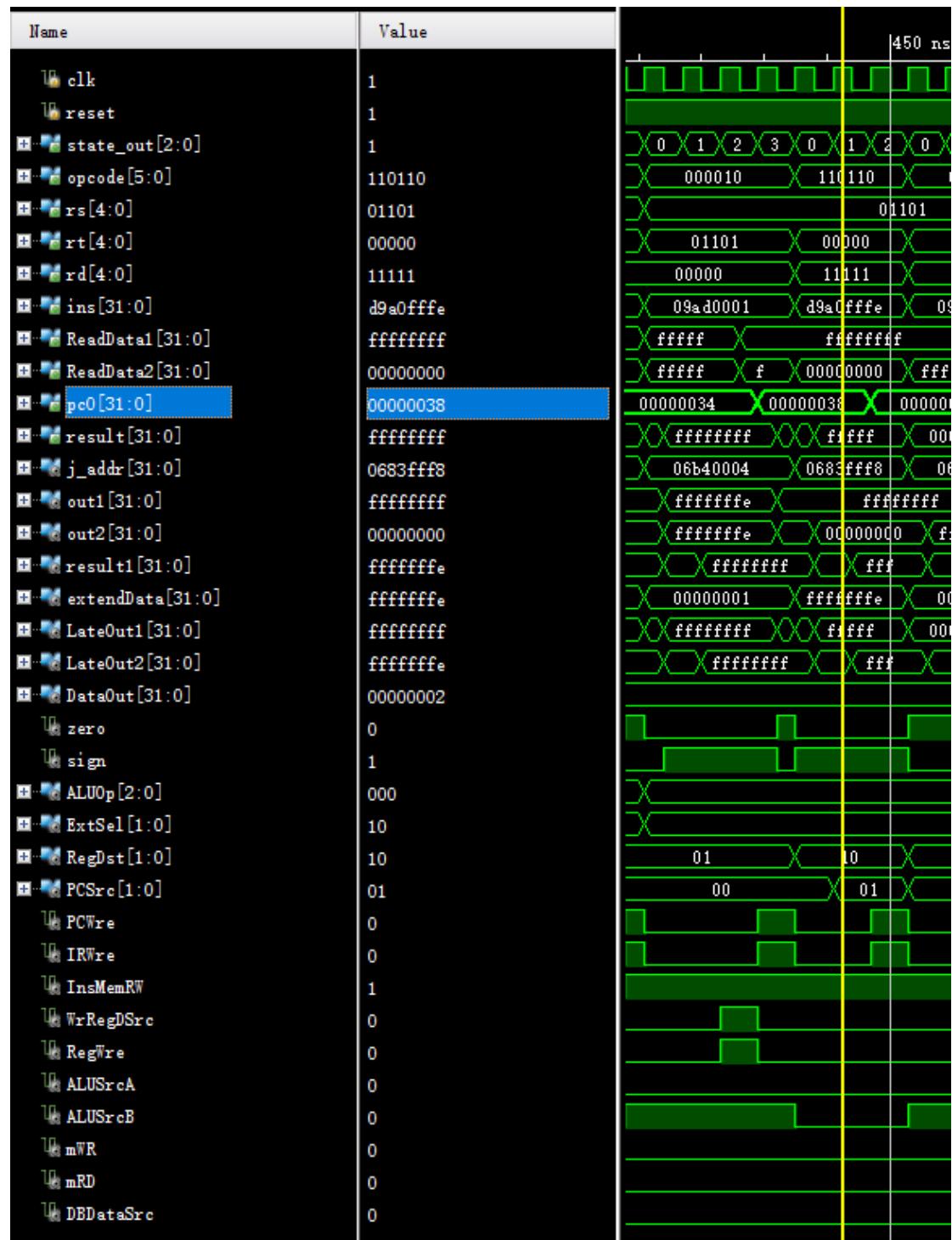
0x0000002C	sltiu \$10,\$9,2	100111	01001	01010	0000 0000 0000 0010	9D2A0002
------------	------------------	--------	-------	-------	---------------------	----------



指令地址为pc0，值为0000002c，InsMemory输出ins的值为9d2a002，rs的值为01001，寄存器\$9，值为0000000，rt的值为01010，寄存器\$10，立即数经过符号拓展后extendData的值为00000002，ALUSrcA为0，选择out1，ALUSrcB为1，选择extendData，操作码ALUOp为010，执行带符号比较运算，计算结果result为0000001，

经过一个二选一和ALUM2DR，到regfile的输入端lateOut2，WrRegDSrc的值为1，写入寄存器\$10.

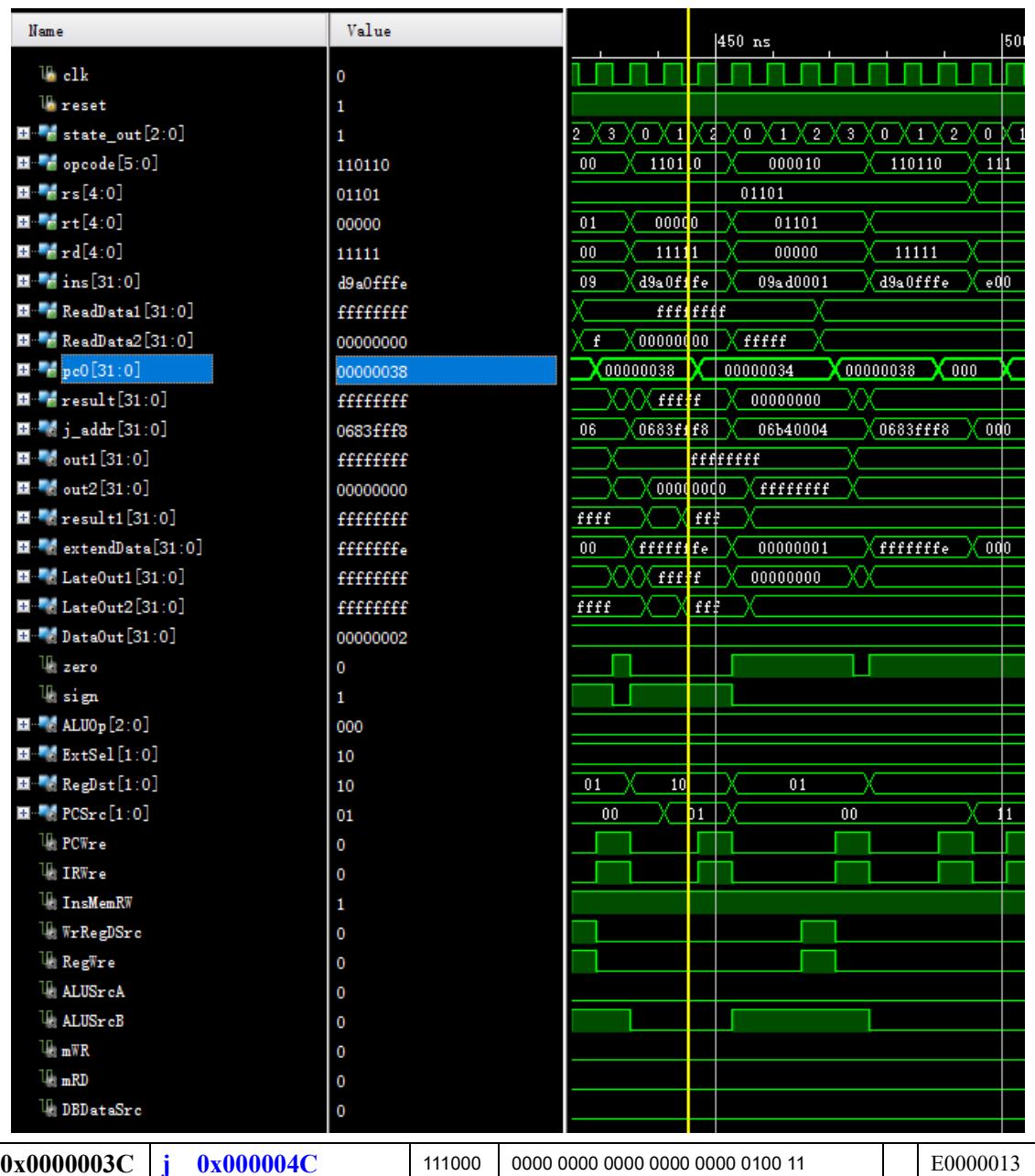
0x00000038	bltz \$13,-2 (<0,转 34)	110110	01101	00000	1111 1111 1111 1110		D9A0FFE
------------	------------------------	--------	-------	-------	---------------------	--	---------

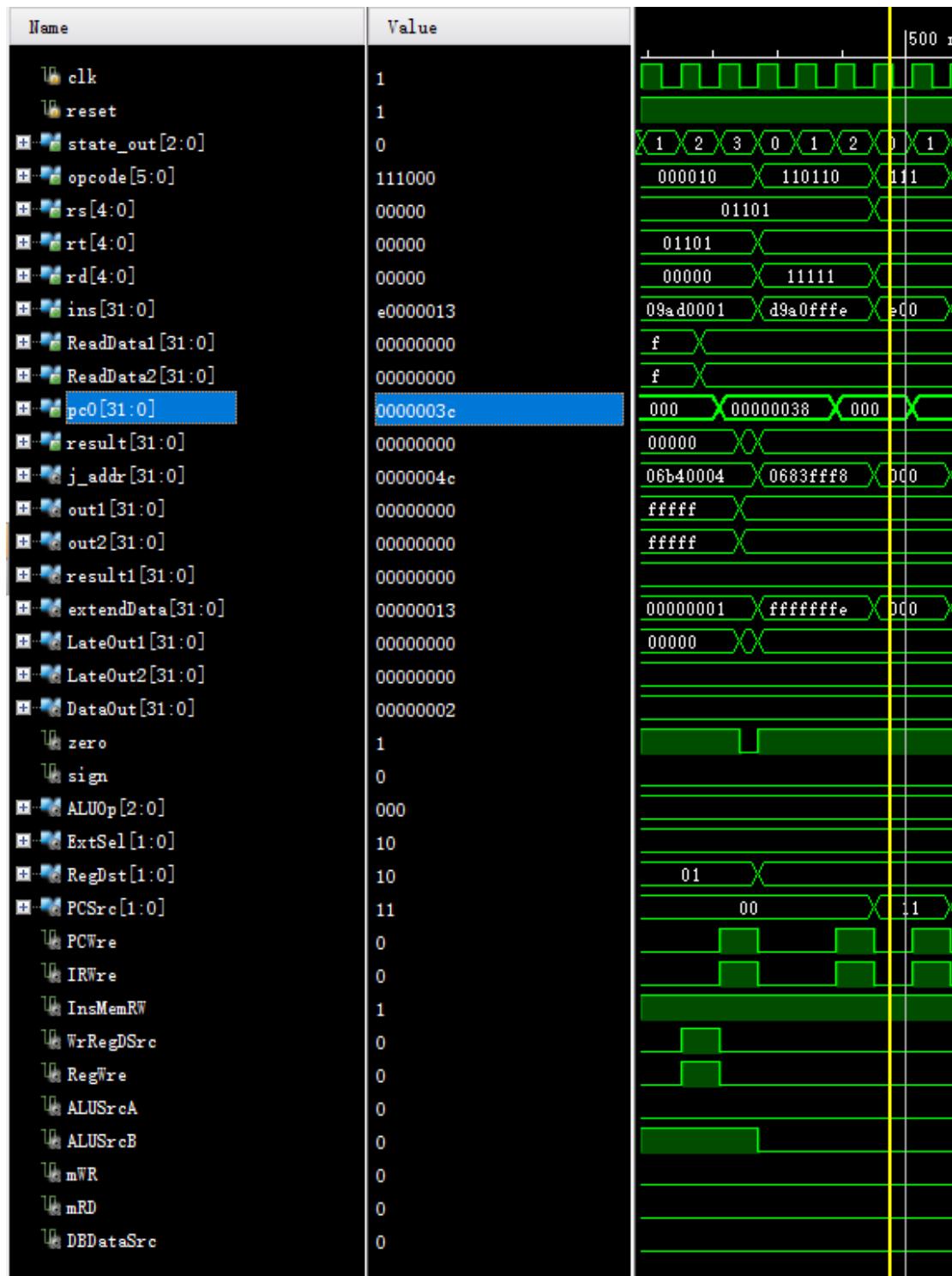


指令地址为pc0，值为00000038，InsMemory输出ins的值为9da0ffe，rs的值为01101，即寄存器\$13，值为ffffffff，输出到out1，rt的值为00000，即寄存器\$0，值为00000000，输出到out2，ALUSrcA的值为0，选择out1，ALUSrcB的值为0，选择out2，

操作码ALUOp为001，执行减操作，结果result为fffffff, zero标志位为0，表示不相等，sign为1，表示负数，说明rs<0，下一个状态PCSrc为01，pc模块选择pc+4+立即数左移两位后的值，pc0变为00000034.进入循环

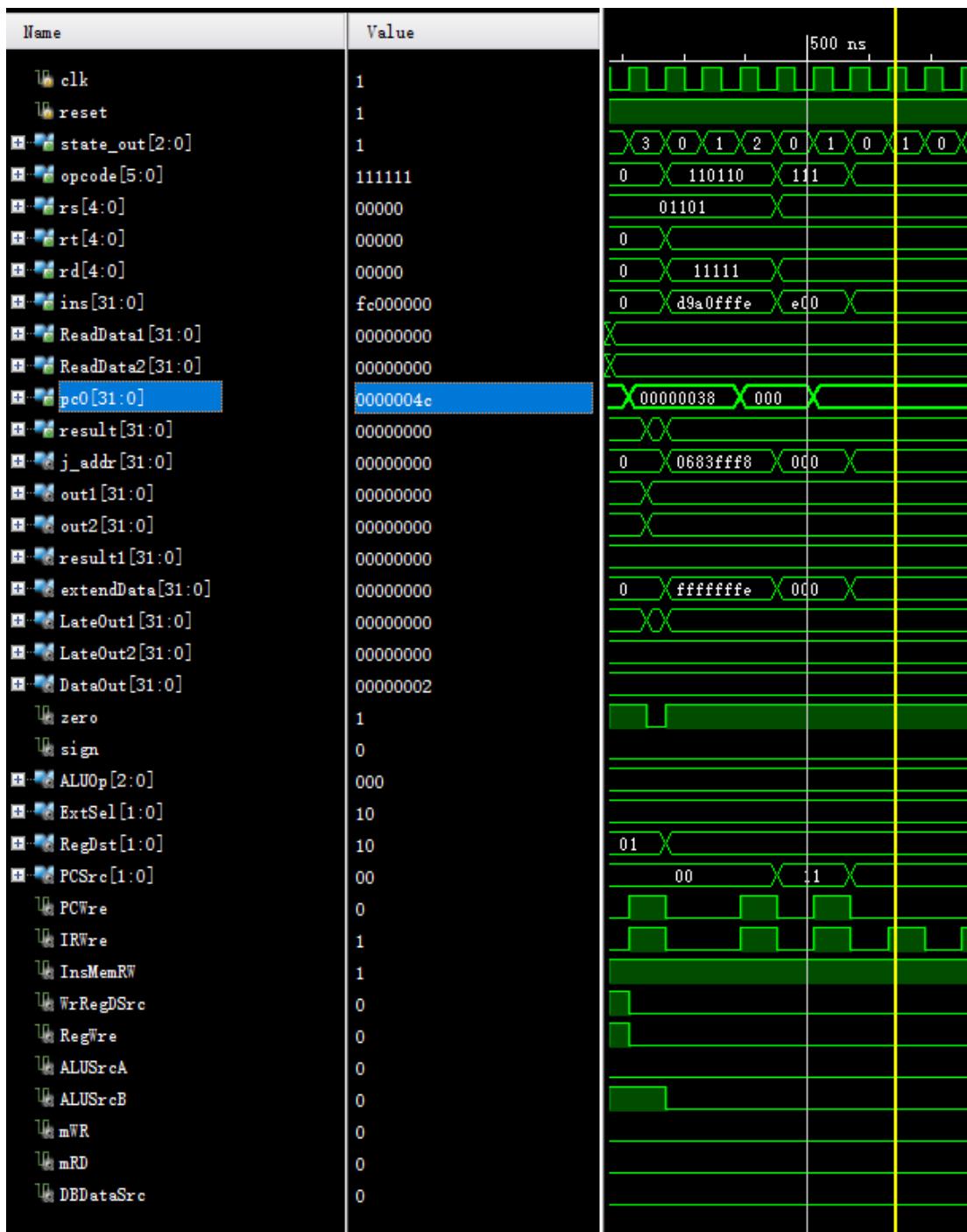
循环过程





指令地址为pc0，值为0000003c，InsMemory输出ins的值为e0000013，addr输入PCAddr模块，扩充为跳转地址j_addr，值为0000004c，PCSrc为11,选择j_addr，下一条指令地址为0000004c。

0x0000004C	halt	111111	00000	00000	00000000000000000000	=	FC000000
------------	------	--------	-------	-------	----------------------	---	----------



指令地址为pc0，值为0000004c，InsMemory输出ins的值为fc000000，PCWre为0，不继续读取指令，停机。

3. basys3板实现

在前面仿真的基础上加上时钟分屏模块和LED控制模块

clk_div: 时钟分频模块，mclk为basys板上的时钟频率，clkCPU为按键消抖，clk190为扫描的时钟信号

```

reg [26:0]q;
reg [1:0] counter;

initial begin
    q=0;
end

always@(posedge mclk)
begin
    q <= q + 1;
    |
    if(reset == 0)
        clkCPU = 1; //reset=000000000
    else
        begin
            if(button == 1)
                begin
                    clkCPU <= 1;//0000CPU000
                    counter <= 0;
                end
            else
                begin
                    counter <= counter + 1;
                    if(counter == 3)
                        clkCPU <= 0; //CPU00
                end
        end
    end
end

assign clk190 = q[10];

```

LED模块，根据分频后的时钟信号进行选位，switch是两个按键开关的值，，number是要显示的值，pos是数码管的控制信号。

```

always@(posedge clk190)
begin
    counter <= counter + 1;

//BBBBB(AN3)BB
if(counter == 0)
begin
    case (switch)
        2'b00:number <= address[7:4];
        2'b01:number = {3'b000, rs[4]};
        2'b10:number = {3'b000, rt[4]};
        2'b11:number = result[7:4];
    endcase

    pos <= 4'b0111;
end

//BBBBB(AN2)BB
else if(counter == 1)
begin
    case (switch)
        2'b00:number <= address[3:0];
        2'b01:number = rs[3:0];
        2'b10:number = rt[3:0];
        2'b11:number = result[3:0];
    endcase

    pos <= 4'b1011;
end

//BBBBB(AN1)BB
else if(counter == 2)
begin

```

main的输入输出

```

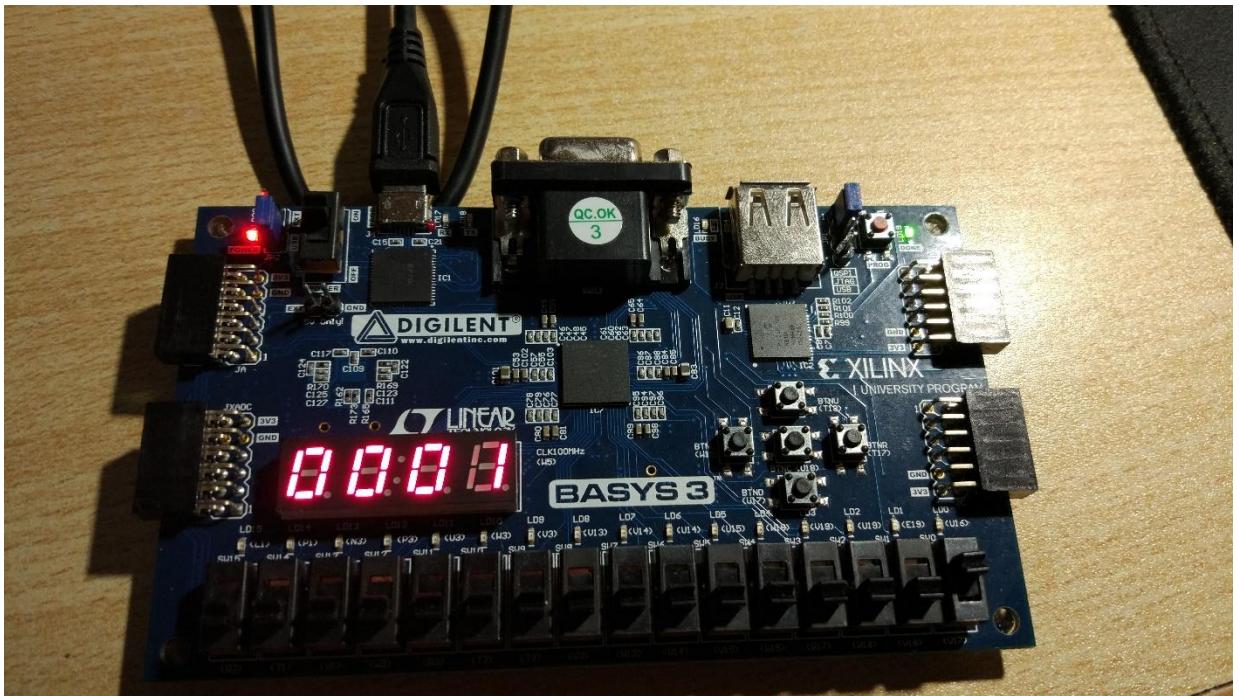
module main(input mclk, reset,button,
            input [1:0] switch,
            output [3:0] pos,
            output [7:0] a_to_g
);

```

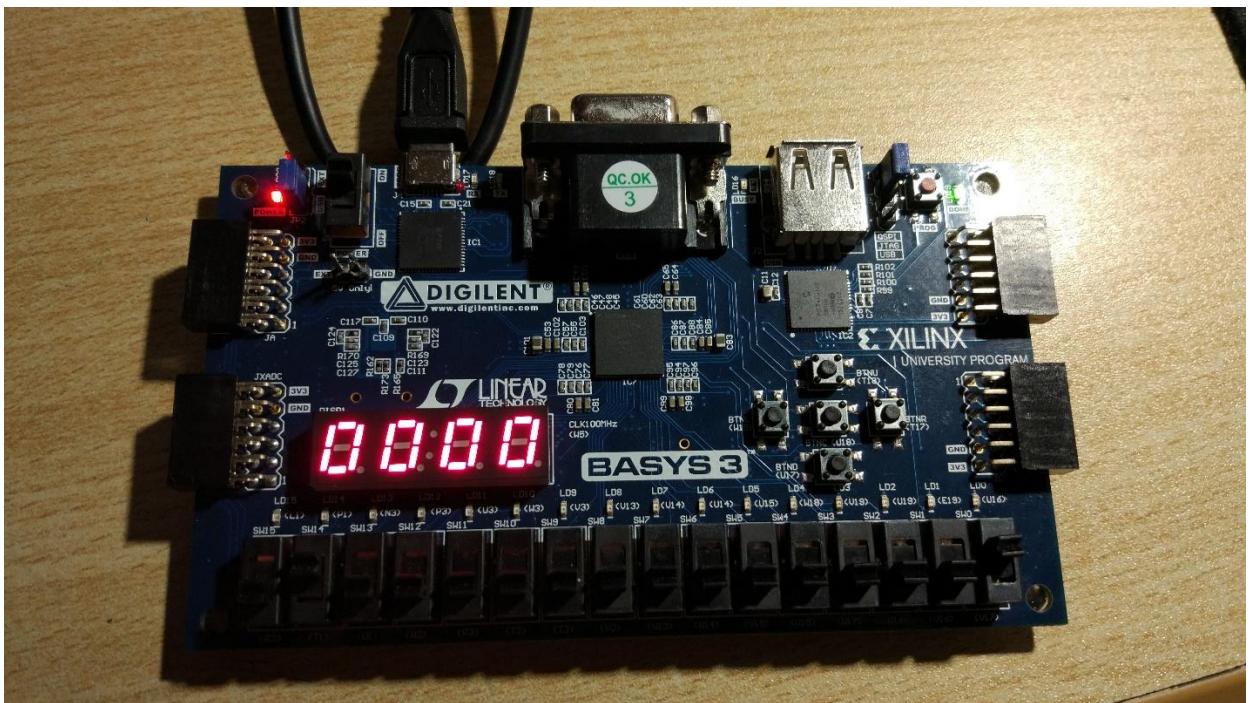
由于是多周期cpu，一条指令需要几个时钟周期，我就把之前显示下一条指令的地方换成了控制器状态

第一条指令

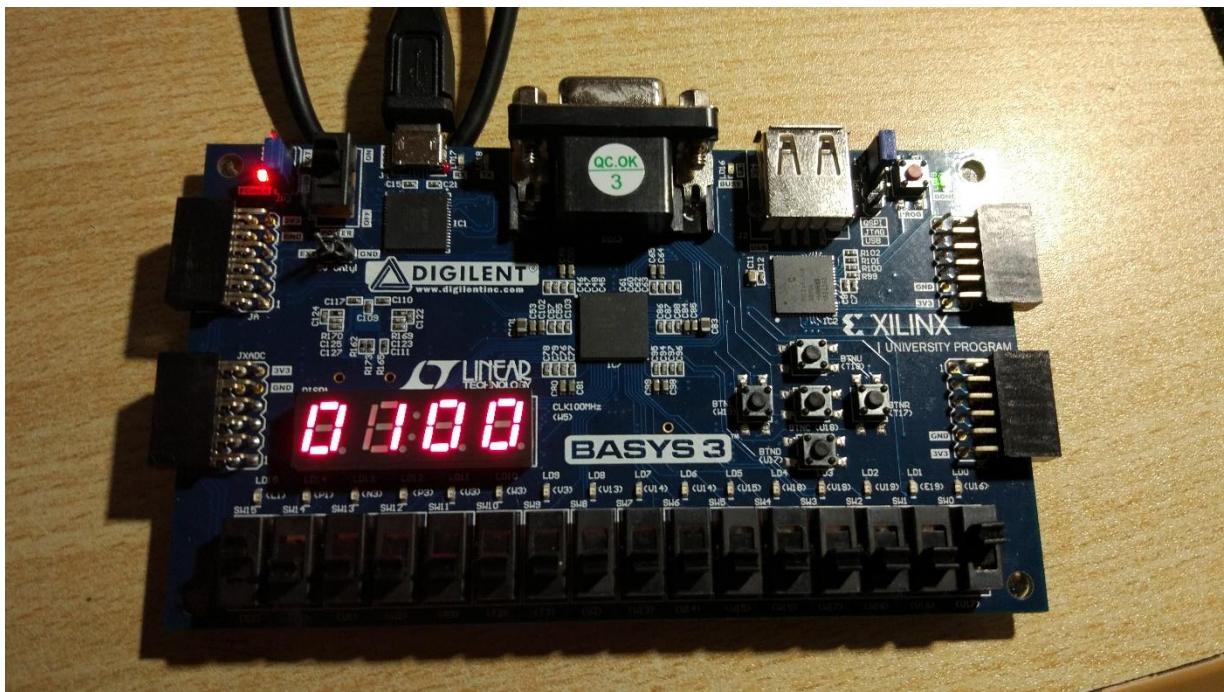
当前指令地址：控制器状态



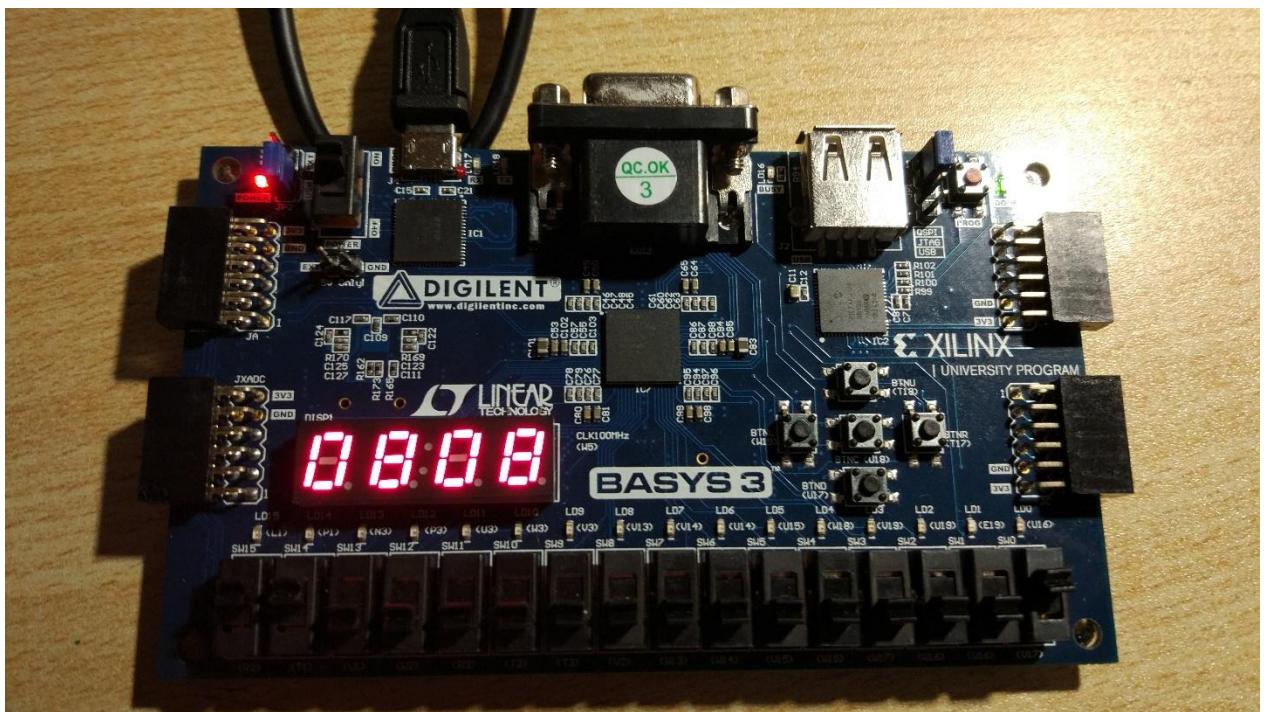
rs: rs的值



rt: rt的值

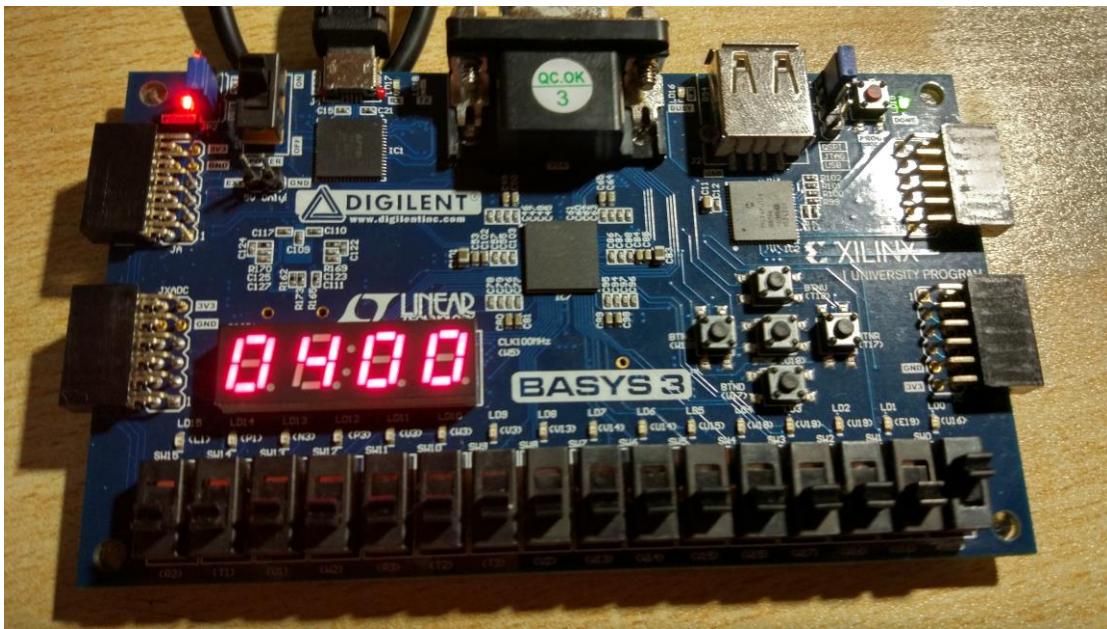


result: dataout

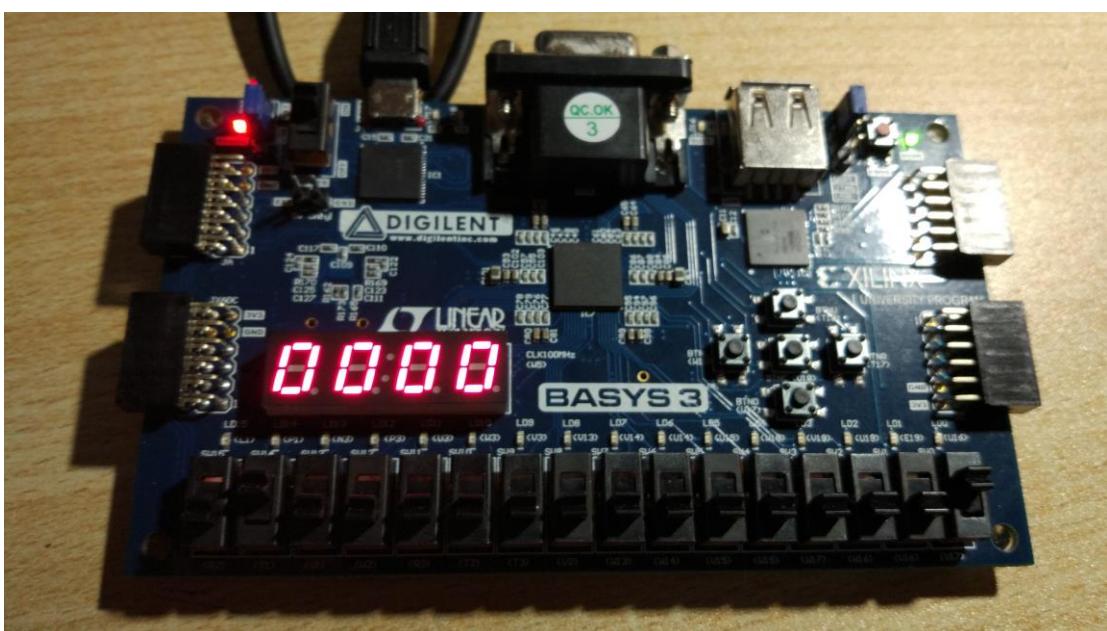


第二条指令

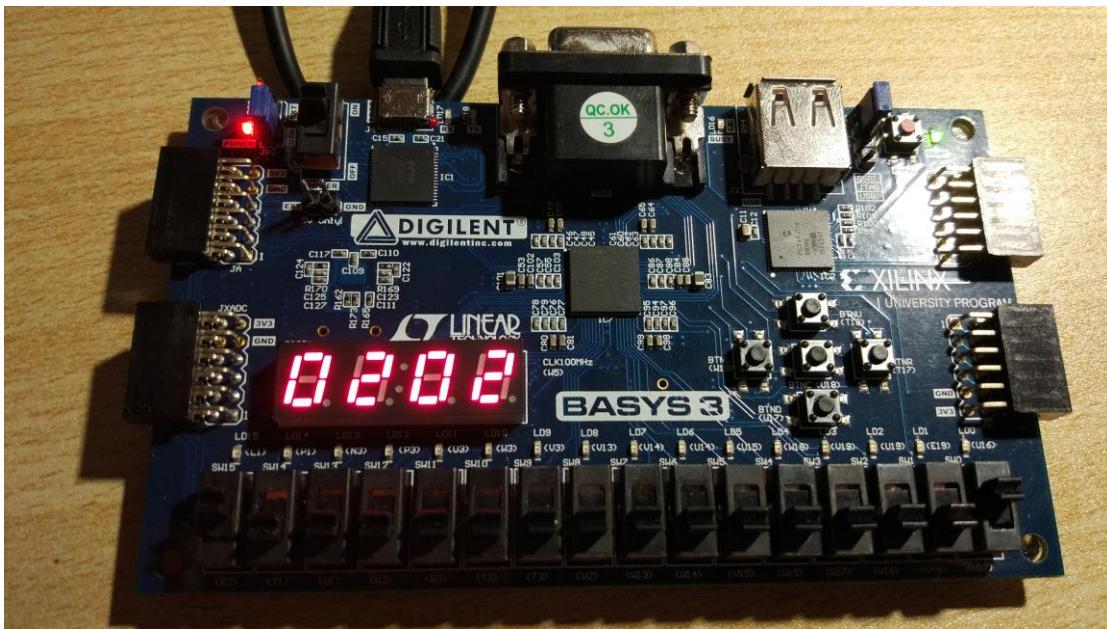
当前指令地址：控制器状态



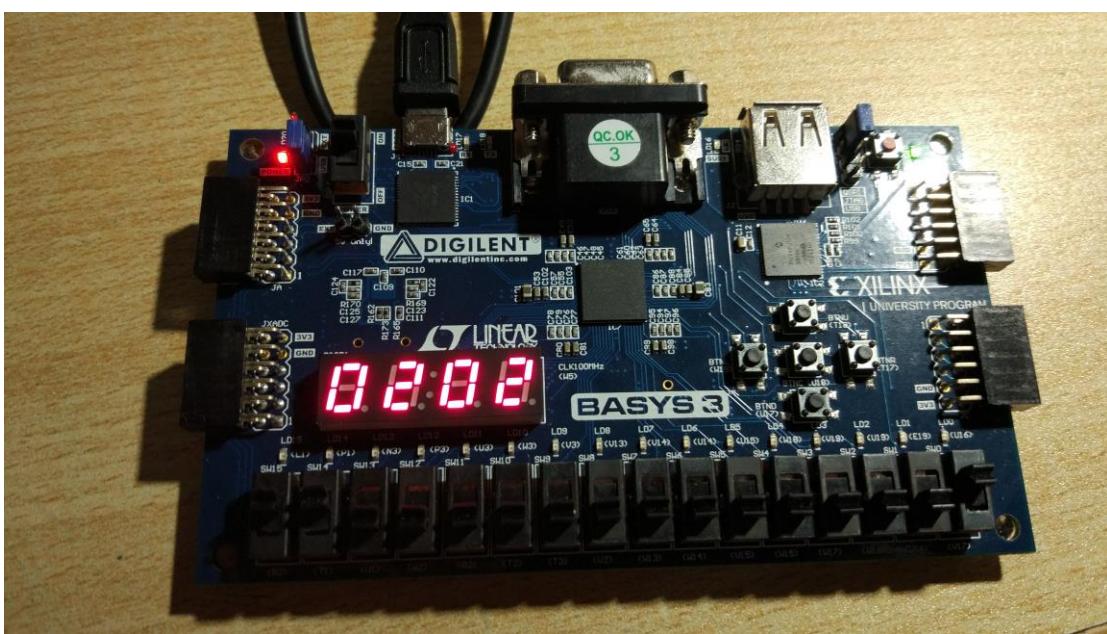
rs: rs的值



rt: rt的值

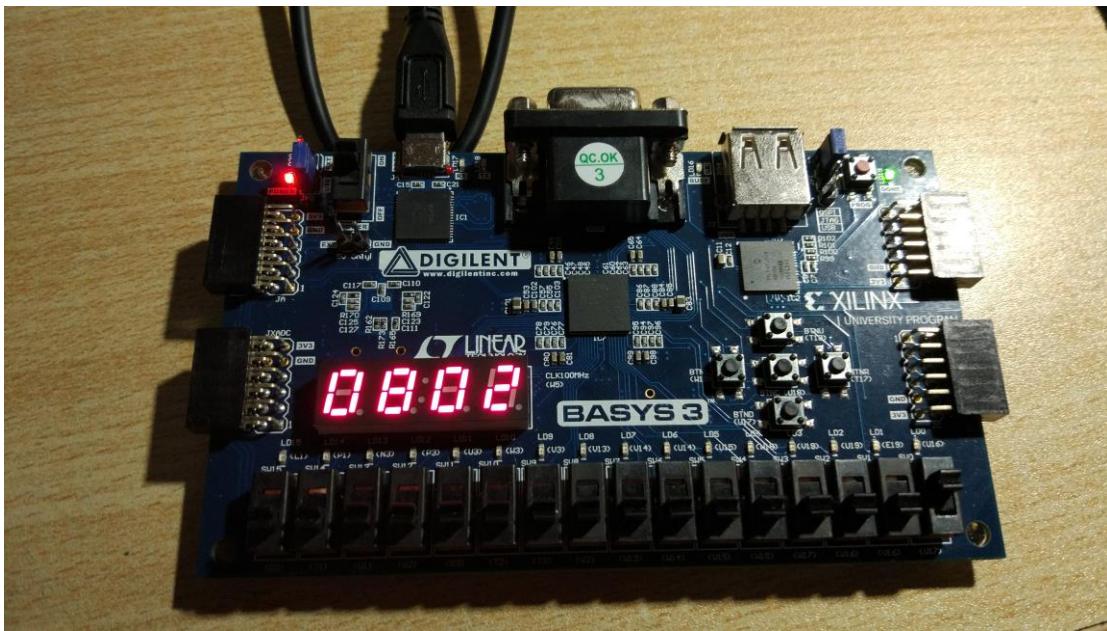


result: dataout

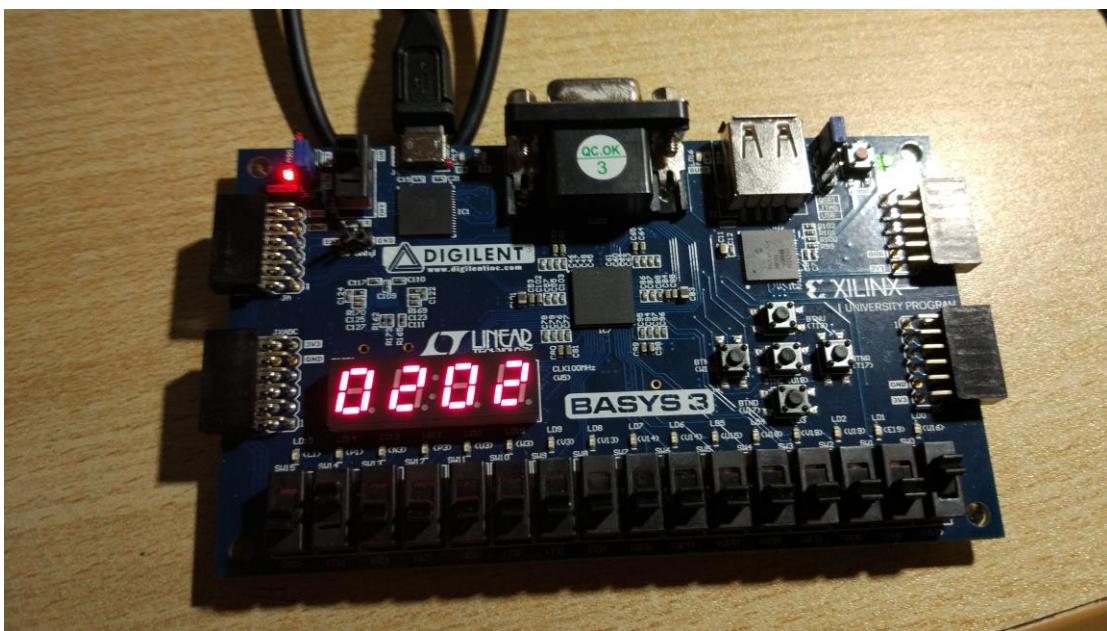


第三条指令

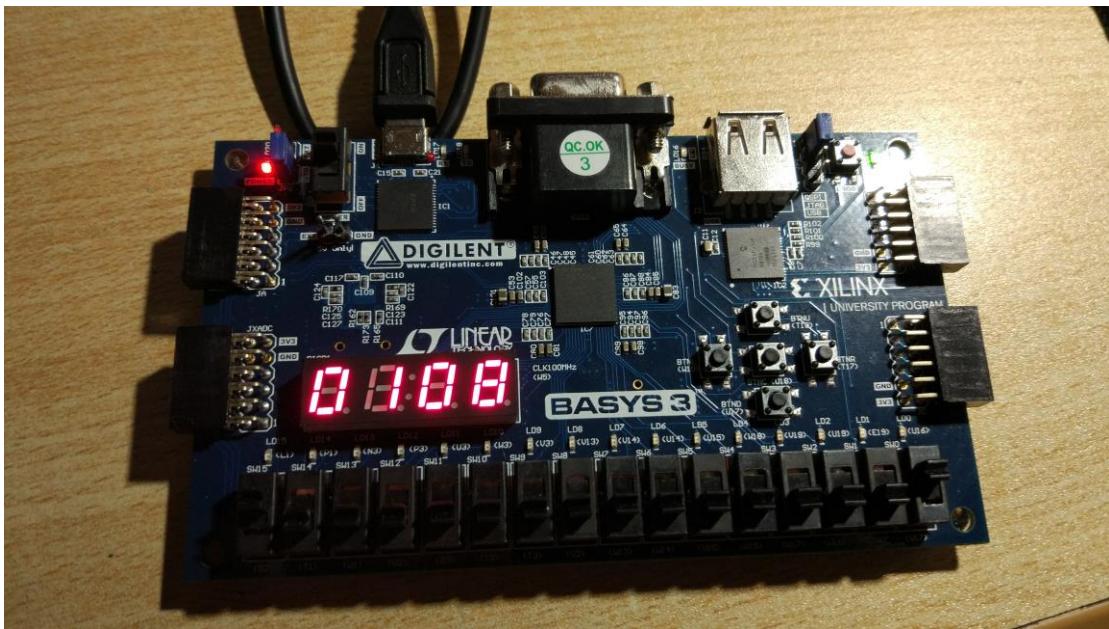
当前指令地址：控制器状态



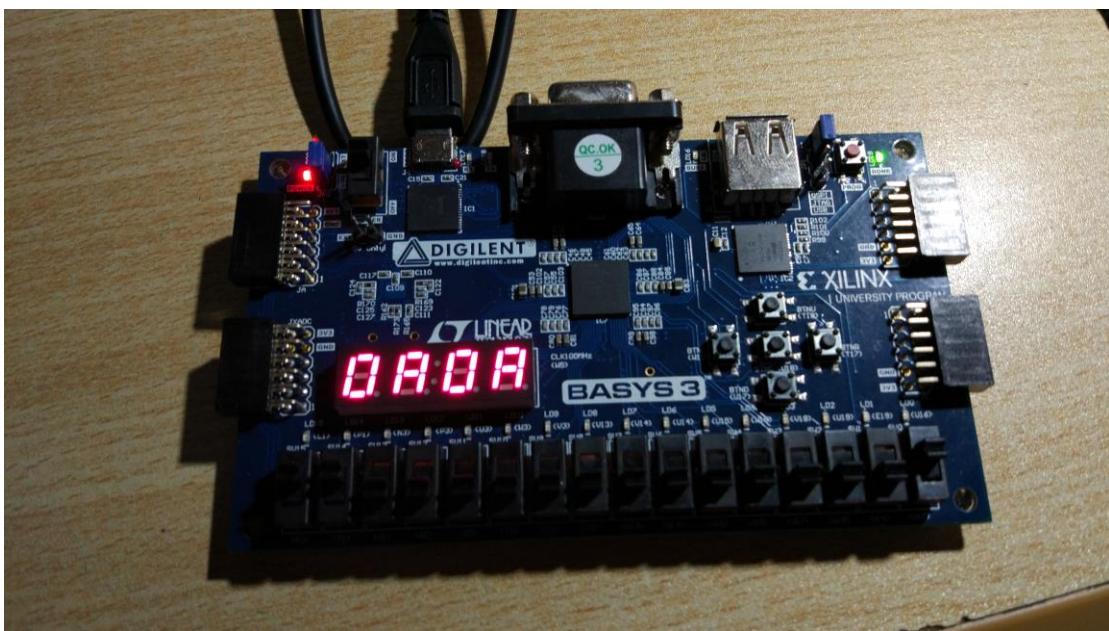
rs: rs的值



rt: rt的值



result: dataout



六. 实验心得

有了上次单周期设计的经验，这次的多周期要容易很多，一上来思路就很清晰，先分开写小模块，再连线，单周期cpu中将每一个小模块都单独写了出来，这次就把很多数据选择器和其他部件结合在了一起，代码简洁了很多，也更自然了。多周期和单周期的不同之处主要在于控制模块和新增了几个寄存器。寄存器比较简单，只是暂时存储了一下数据，写成一个模块，还可以复用。控制模块要先确定控制器的状态，不同的状态对应不同的控制信号，但是有些指令的控制信号比较特殊，所以还需要结合指令还有ALU返回的信号来写。

在写板的时候发现了仿真的代码的很多问题。因为仿真的时候一切正常，但写到板上却又不对，最后发现综合的阶段提示了很多警告，但我一开始并没有在意，导致De了很久的bug，这些提示信息非常重要，它们不会影响仿真的结果，但会影响写板，比如说在main模块里不声明下面连线用到的变量，直接连线，也可以仿真成功。我遇到的一个一个比较严重的问题就是inferring latch for variable，推断出锁存器，当if或者case语句不完整的时候，就会出现这样的问题，因为变量保存了上一次的值，在组合逻辑中插入了时序逻辑，会导致状态机出现不可控的状态。最简单的改法是将always组合逻辑换成时序逻辑，因为我们很多时候需要保存上一次的值。这里又会有一个问题，就是不同组件间的时序需要协调好，经过几次调试，我最终确定PC采用时钟上升沿，regFile采用时钟下降沿，控制单元也采用时钟下降沿。上述问题我在写板的代码里进行了修改，但没有修改仿真的代码文件，所以如果有冲突以写板的代码为准。

这次实验结束，也宣告着本学期计组实验的结束，经过这三次实验，学会了vivado和VHDL语言的使用，对计算机cpu的运行机制也有了更深刻的了解，在实验过程中也学到了很多理论没有注意到的知识，还有老师严肃认真，一丝不苟的态度也很值得我们学习，总的来说，这门课让我受益匪浅。