



《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

学生姓名 : 朱多煜

学号 : 16340314

专业 (班级) : 16软件工程四 (8) 班

时间 : 2018 年 6 月 25 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: $rd \leftarrow rs + rt$

- (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: $rd \leftarrow rs \mid rt$

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

(13) **bltz rs,immediate**

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

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

==>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	000000000000000000000000(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：

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	

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。

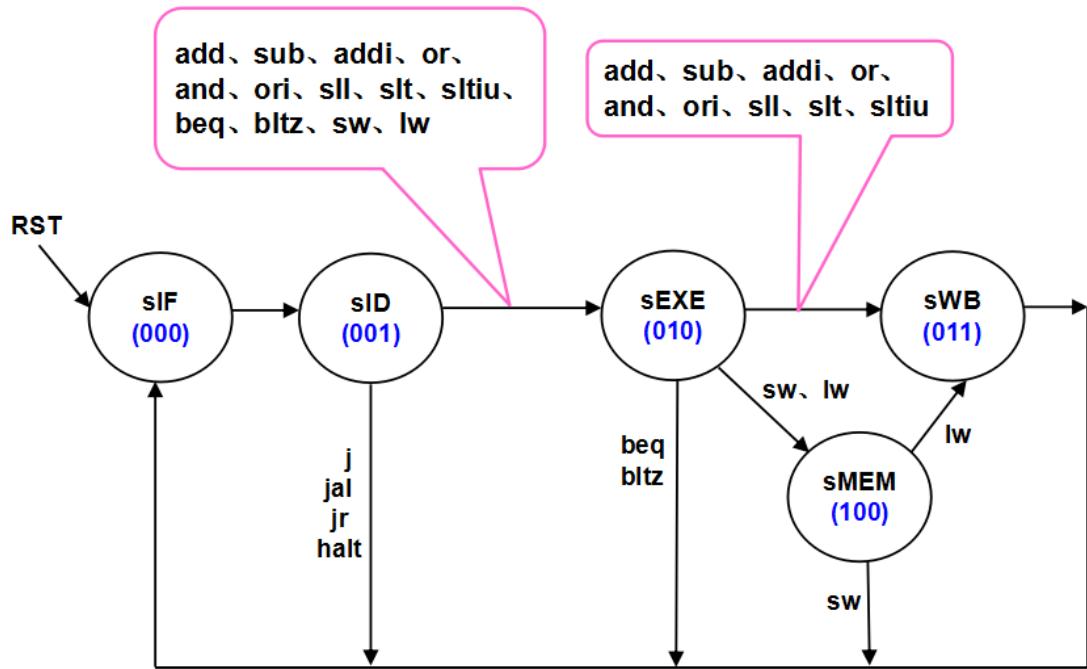


图2 多周期CPU状态转移图

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。

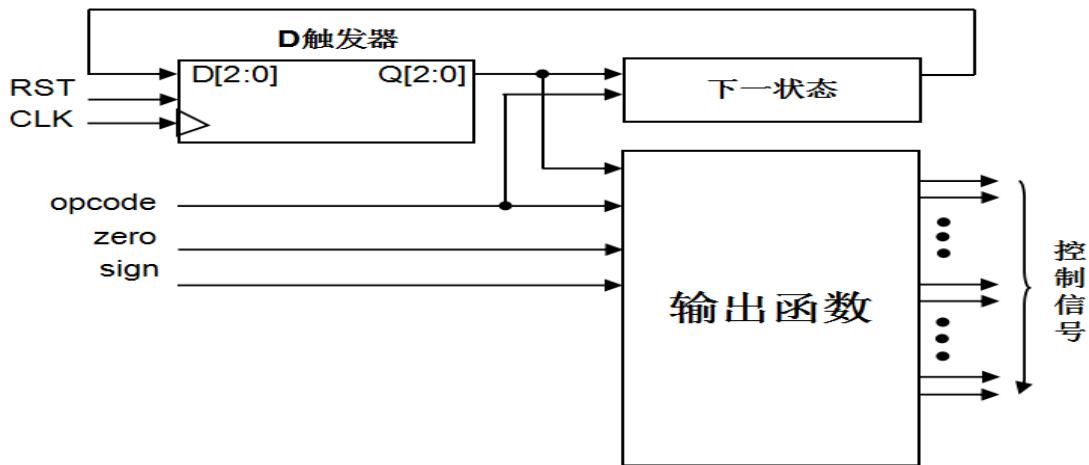


图3 多周期CPU控制部件的原理结构图

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。

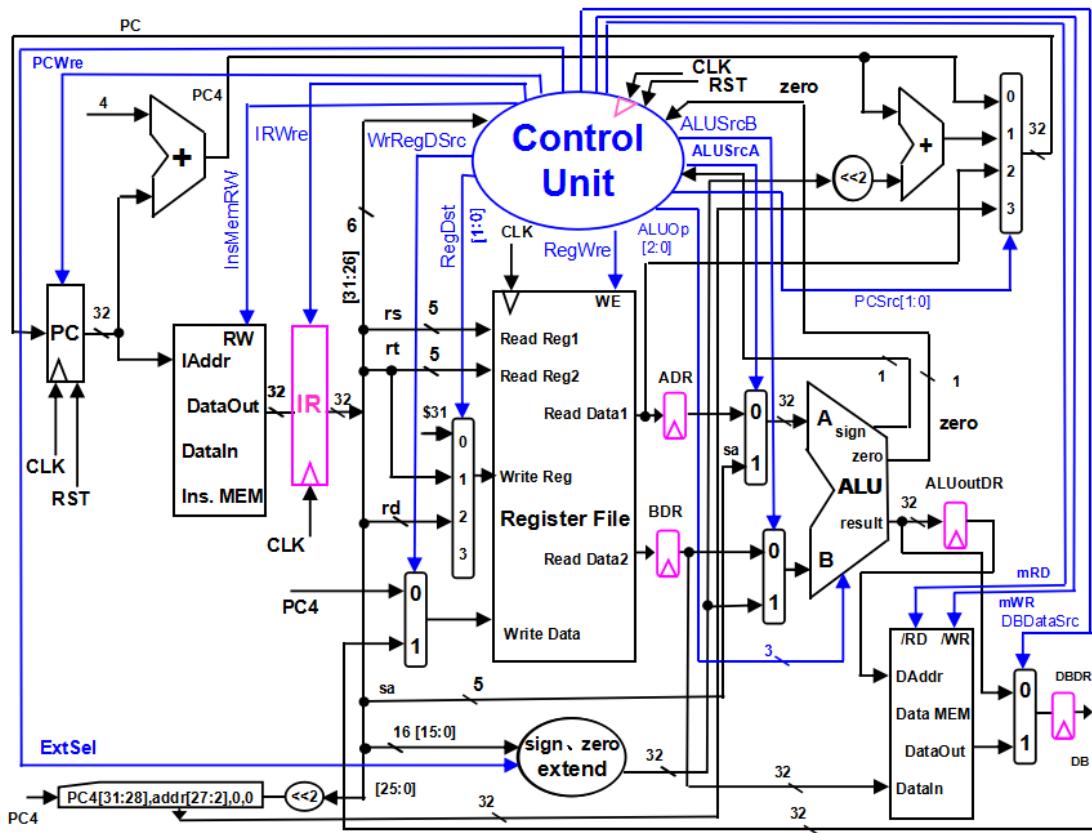


图4 多周期CPU数据通路和控制线路图

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。

四. 实验器材

PC机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

五. 实验分析与设计

1. 各模块的代码实现以及设计思路

(1) PC模块——PC模块主要功能是获取下一条要执行的指令，并输出该指令。由PCWre来控制指令改变的格式，代码如下：

```
`timescale 1ns / 1ps
module PC(
    input CLK,
    input Reset,
    input [31:0] PC,
    input PCWre,
    output reg [31:0] IAddr
);
    initial IAddr<=0;
    always @(posedge CLK or negedge Reset)begin
        if(Reset==0) IAddr<=0;
        else if(PCWre) IAddr<=PC;
        else IAddr<=IAddr;
    end
endmodule
```

(2) 指令改变模块——在数据通路图中，获取下一个指令的方式有三种，第一种方式为在原地址的基础上加4，第二种为将传入的立即数左移两位，再加上原指令加4，第三种为获取到所需要跳转的指令。各改变方式的代码如下

<1> 直接加4

```
module AddPC1(
    input [31:0] PC,
    output [31:0] PC4
);
    assign PC4=PC+4;
endmodule
```

<2> 左移立即数后加原指令，再加4

```
module AddPC2(
    input [31:0] PC4,
    input [31:0] Extend_Out,
    output [31:0] NewPC
);
    assign NewPC=PC4+(Extend_Out<<2);
endmodule
```

<3> 指令跳转

```

module jumpPC(
    input [31:0] PC4,
    input [25:0] addr,
    output reg [31:0] JumpPC
);
    initial JumpPC<=0;
    always@(addr or PC4) begin
        JumpPC[1:0]=2'b00;
        JumpPC[27:2]=addr[25:0];
        JumpPC[31:28]=PC4[31:28];
    end
endmodule

```

(3) 指令选择模块——指令选择模块通过PCSrc输入去决定，下一条指令来自于以上三条指令中的哪一个，模块代码和真值表如下

PCSrc	nextPC
00	PC+4
01	PC4+(immediate<<2)
10	ReadDate
11	JumpPC

```

module NextPC(
    input [31:0] IAddr,
    input [31:0] PC4,
    input [31:0] SkipPC,
    input [31:0] rs,
    input [31:0] JumpPC,
    input [1:0] PCSrc,
    input isHalt,
    output reg [31:0] NextPC
);
    always@(PC4 or SkipPC or JumpPC or PCSrc or isHalt)begin
        if(isHalt==1)
            NextPC<=IAddr;
        else begin
            case(PCSsrc)
                2'b00:NextPC<=PC4;
                2'b01:NextPC<=SkipPC;
                2'b10:NextPC<=rs;
                2'b11:NextPC<=JumpPC;
            endcase
        end
    end
endmodule

```

(4) 指令存取模块——将我们需要测试的指令从文件中读入，并储存。模块等我代码实现如下

```
'timescale 1ns / 1ps
module ROM(
    input InsMemRW,
    input [31:0] IAddr,
    output reg [31:0] IDataOut
);

reg [7:0] mem [0:99];

initial begin
    $readmem("E:/Xilinx/workbench/Multi_period_cpu/test.txt", mem);
    IDataOut<=0;
end
always @(InsMemRW or IAddr)
begin
    IDataOut[7:0] = mem[IAddr + 3];
    IDataOut[15:8] = mem[IAddr + 2];
    IDataOut[23:16] = mem[IAddr + 1];
    IDataOut[31:24] = mem[IAddr];
end
endmodule
```

(5) 寄存器模块——寄存器模块的实现包括三个部分，一部分是WriteReg的输入选择，另一个部分是WriteData的输入选择，还有一部分就是寄存器的主体部分。寄存器主体部分由于和单周期CPU的功能一样，所以直接沿用了上次老师所给的代码，没有做修改。输入选择部分除了之前的rd和rt外，还新增加了一个输入，命名为input31。WriteData部分也与单周期时的相同，这三部分的代码如下：

<1> 主体部分

```
'timescale 1ns / 1ps
|
module RegFile(
    input CLK,
    input RST,
    input RegWre,
    input [4:0] ReadReg1,
    input [4:0] ReadReg2,
    input [4:0] WriteReg,
    input [31:0] WriteData,
    output [31:0] ReadData1,
    output [31:0] ReadData2
);
reg [31:0] regFile[1:31]; // 32x32bit register file
integer i;

assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1]; // 0为复位值
assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];

always @ (posedge CLK or negedge RST) begin // 在时钟上升沿或复位时
    if (RST==0) begin
        for(i=1;i<32;i=i+1)
            regFile[i] <= 0;
    end
    else if(RegWre == 1 && WriteReg != 0) // WriteReg != 0表示写操作
        regFile[WriteReg] <= WriteData; // 将写入的数据写入到指定地址
end
endmodule
```

<2>WriteReg 输入选择部分

```
module ChooseWriteReg(
    input [4:0] input31,
    input [4:0] rt,
    input [4:0] rd,
    input [1:0] RegDst,
    output reg [4:0] WriteReg
);
    always @ (input31 or rt or rd or RegDst) begin
        case(RegDst)
            2'b00: WriteReg<=input31;
            2'b01: WriteReg<=rt;
            2'b10: WriteReg<=rd;
        endcase
    end
endmodule
```

<3>WriteData输入选择部分

```
module ChooseWriteData(
    input [31:0] PC4,
    input [31:0] DB,
    input WrRegDSrc,
    output [31:0] WriteData
);
    assign WriteData=WrRegDSrc?DB:PC4;
endmodule
```

(6)立即数扩展模块——立即数扩展模块实现了将16位的立即数扩展为32位，以便于其参加之后模块的运算。该模块由ExtSel输入控制是有符号扩展还是无符号扩展。真值表和代码如下：

ExSel	补全
0	进行无符号扩展
1	进行有符号扩展

```

`timescale 1ns / 1ps
module extend(
    input [15:0] datain,
    input ExtSel,
    output reg [31:0] result
);
    always@(datain or ExtSel) begin
        result[15:0]<=datain[15:0];
        if(ExtSel==1 && datain[15]==1)
            result[31:16]<=16'b1111_1111_1111_1111;
        else
            result[31:16]=0;
    end
endmodule

```

(7) ALU模块——ALU模块分为三个部分，包括A端的输入选择模块，B端的输入选择模块和ALU主体模块。两个输入模块分别由ALUSrcA，ALUSrcB控制输出信息的选择。模块A在ReadData1和sa中选择输出，模块B在ReadData2和扩展后的立即数中选择输出。ALU主体模块的真值表和各模块的代码如下。

<1> ALU主体模块真值表

表1 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	$Y = A \oplus B$	异或

<2>ALU主体模块代码——只需要在单周期的基础想，按照真值表做一些简单修改

```

`timescale 1ns / 1ps

module ALU(
    input [2:0] ALUOpcode,
    input [31:0] A ,
    input [31:0] B ,
    output sign ,
    output zero ,
    output reg [31:0] result
);

initial result=0;

always@(ALUOpcode or A or B) begin
    case(ALUOpcode)
        3'b000: result <= A + B;
        3'b001: result <= A - B;
        3'b010: result <= (A < B)?1:0;
        3'b011: result <= (((A-B) && (A[31] == B[31]))||((A[31] == 1 && B[31] == 0)))?1:0;
        3'b100: result <= B << A;
        3'b101: result <= A | B ;
        3'b110: result <= A & B ;
        3'b111: result <= A ^ B ;
    endcase
end
    assign zero = ( result == 0 ) ? 1:0;
    assign sign = result[31];
endmodule

```

<3>输入选择模块——与单周期输入选择模块不同，由于A、B两个模块的功能完全一样，所以我们只实现了一个模块，然后在顶层文件中利用不同输入，创建不同的实例。

```

module ChooseInput(
    input [31:0] Data1,
    input [31:0] Data2,
    input control,
    output [31:0] OutData
);
    assign OutData=control?Data2:Data1;
endmodule

```

(8) 数据储存和输出模块——该模块包括数据的储存模块和输出模块两个部分.其中数据储存的部分的代码也与单周期CPU的代码相同，利用老师所给的代码，代码如下：

<1> 数据储存模块

```

`timescale 1ns / 1ps
module RAM(
    input clk,
    input [31:0] address,
    input [31:0] writeData,
    input nRD,
    input nWR,
    output [31:0] Dataout
);

reg [7:0] RAM [0:31];
integer i;
initial
begin
    for(i = 0; i < 32; i = i+1) RAM[i] <= 0;
end

assign Dataout[7:0] = (nRD==0)?RAM[address + 3]:8'bz;
assign Dataout[15:8] = (nRD==0)?RAM[address + 2]:8'bz;
assign Dataout[23:16] = (nRD==0)?RAM[address + 1]:8'bz;
assign Dataout[31:24] = (nRD==0)?RAM[address ]:8'bz;

always@( negedge clk ) begin
    if( nWR==0 ) begin
        RAM[address] <= writeData[31:24];
        RAM[address+1] <= writeData[23:16];
        RAM[address+2] <= writeData[15:8];
        RAM[address+3] <= writeData[7:0];
    end
end
endmodule

```

<2>数据输出模块——数据输出模块的功能类似于之前寄存器模块的输入选择，均由一个控制信号，控制在两个输入中选择一个输出，所以也采用之前的模块，在顶层文件利用不同的输入，构建不同的实例。

(9) 指令代存模块——即数据通路图中新加的的IR模块，目的是为了实现指令执行的分段。为了保证指令代码的稳定性，添加一个使能的信号，在时钟上升沿，IR接收从指令存储器送来的指令代码

```

module IR(
    input CLK,
    input IRWre,
    input [31:0] DataOut,
    output reg [31:0] InsOut
);
    reg [31:0] Ins_Register;
    initial Ins_Register=0;//initial the ins data=0

    always @(posedge CLK) begin
        if(IRWre==1)//write
            Ins_Register<=DataOut;
        else
            Ins_Register<=Ins_Register;
    end

    always @(posedge CLK) begin//read
        InsOut<=Ins_Register;
    end
endmodule

```

(10)数据代存模块——即数据通路中新加入的ADR、BDR、ALUoutDR、DBDR部件，与指令代存模块的目的相同，为了实现一条指令执行的分段。不需要使能信号。

```

module DR(
    input CLK,
    input [31:0] Datain,
    output reg [31:0] Dataout
);
    reg [31:0] DataRegisterMem;
    initial DataRegisterMem<=0;
    //negedge save posedge out
    always@(negedge CLK) begin
        DataRegisterMem<=Datain;
    end

    always@(posedge CLK) begin
        Dataout<=DataRegisterMem;
    end

endmodule

```

(10) transTocut模块——并不是属于CPU的必要模块，是在查找相关资料时发现的一个简化输出的模块，便于我们仿真时核对指令的正确性。其主要原理就是在多位的输出中截取我们需要的那几位，具体实现的代码如下：

```

`timescale 1ns / 1ps
module transTocut(
    input [31:0] address,new_address,data1,data2,result,DB,
    input [4:0] rs,rt,
    output [7:0] addr_4,new_addr_4,rs_4,data1_4,rt_4,data2_4,result_4,
    output [7:0] DB_4
);

    assign    addr_4[7:0] = address[7:0];
    assign    new_addr_4[7:0] = new_address[7:0];
    assign    rs_4[4:0] = rs[7:0];
    assign    rs_4[7:5] = 0;
    assign    data1_4[7:0] = data1[7:0];
    assign    rt_4[4:0] = rt[7:0];
    assign    rt_4[7:5] = 0;
    assign    data2_4[7:0] = data2[7:0];
    assign    result_4[7:0] = result[7:0];
    assign    DB_4[7:0] = DB[7:0];
endmodule

```

(10) ControlUnit模块——该模块可以说是整个单周期CPU的控制中心，通过op的输入来确定指令的格式，并通过各命令的输出，去控制其他模块的运行。其中个控制信号的作用以及相关指令如下：

表2 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、sliu、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、sliu、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

相对于单周期CPU, 其控制信号的变化会更复杂, 所以我用了与上一次代码不同的实现方式, 在单周期CPU我是对于每一个指令取设置输出, 而这次我则是从每一个元件出发去赋值, 其代码如下:

```

module ControlUnit(
    input zero,
    input sign,
    input RST,
    input CLK,
    input [5:0] opcode,
    output reg [1:0] RegDst,
    output reg WrRegDSrc,
    output reg InsMemRW,
    output reg IRWre,
    output reg PCWre,
    output reg ExtSel,
    output reg DBDataSrc,
    output reg mWR,
    output reg mRD,
    output reg ALUSrcB,
    output reg ALUSrcA,
    output reg [1:0] PCSrc,
    output reg [2:0] ALUOp,
    output reg RegWre,
    output reg isHalt,
    output reg [2:0] state
);

initial begin
    state=3'b000;
    PCWre=1;
    ALUSrcA=0;
    ALUSrcB=0;
    DBDataSrc=0;
    ReqWre=1;
    InsMemRW=0;
    mRD=1;
    mWR=1;
    RegDst=2'b00;
    ALUOp=3'b000;
    IRWre=0;
    WrRegDSrc=1;
    isHalt=0;
    ExtSel=0;
    PCSrc=2'b00;
end

always@(posedge CLK or negedge RST)begin//on posedge set state ,D chufaqi
    if(RST==0)begin
        state<=3'b000;
        isHalt<=0;
    end
    else begin

        if(state==3'b000) //next state 000->001
            state=3'b001;
        else if(state==3'b001) begin//in state 001
            if((opcode==6'b111000)||| (opcode==6'b111001)||| (opcode==6'b111010)||| (opcode==6'b111111))// j jr jal halt
                state<=3'b000;
            else state=3'b010; //if not j jr jal halt then next is 010
        end
        else if(state==3'b010)begin
            if((opcode==6'b110100)||| (opcode==6'b110110))//beq bltz
                state<=3'b000;
            else if((opcode==6'b110000)||| (opcode==6'b110001))//sw lw
                state<=3'b100;
            else state<=3'b011;//next
        end
        else if(state==3'b100) begin//sMEM
            if(opcode==6'b110001) state<=3'b011;
            else state=3'b000;
        end
        else//else
            state<=3'b000;
    end
end

always@{RST or state or zero or opcode or sign}begin

    //ALUSrcA
    if((state==3'b010|||state==3'b011) && opcode==6'b011000)//var ALUSrcA,only in state sEXE
        ALUSrcA=1;
    else
        ALUSrcA=0;

    //ALUSrcB
    if(state==3'b010) begin//in sEXE
        if((opcode==6'b000010)||| (opcode==6'b010010)||| (opcode==6'b100111))//addi ori sltiu
            (opcode==6'b110000)||| (opcode==6'b110001))//sw lw
            ALUSrcB=1;
        else
            ALUSrcB=0;
    end
    else
        ALUSrcB=0;

    //DBDataSrc
    if(state==3'b100&&opcode==6'b110001)//lw sMEM
        DBDataSrc=1;
    else
        DBDataSrc=0;

    //RegWre
    if(state==3'b011) RegWre=1;//éšèl#011çš„mçñkaçtil??
    else if(state==3'b001&&opcode==6'b111010) RegWre=1; //jal
    else RegWre=0;

```

```

//WrRegDSrc
if(state==3'b001&&opcode==6'b111010)WrRegDSrc=0;
else WrRegDSrc=1;

//InsMemRW

//mRD
if(state==3'b100&&opcode==6'b110001)mRD=0;
else mRD=1;

//mWR
if(state==3'b100&&opcode==6'b110000)mWR=0;
else mWR=1;

//IRWre
if(state==3'b000) IRWre=1;//000 to load pc
else IRWre=0;

//ExtSel
if(state==3'b010)begin
    if((opcode==6'b000010)||| (opcode==6'b110000)||| (opcode==6'b110001)||| (opcode==6'b110100)||| (opcode==6'b110110))ExtSel=1;
    else ExtSel=0;
end
else ExtSel=0;

//PCSrc
if(state==3'b001)begin
    if((opcode==6'b111000)||| (opcode==6'b111010))PCSrc=2'b11;//j jal
    else if(opcode==6'b111001)PCSrc=2'b10;//jr
    else if(opcode==6'b111111)PCSrc=2'b2;
    else PCSrc=2'b00;
end
else if(state==3'b010)begin
    if((opcode==6'b110100&&zero==1)||| (opcode==6'b110110&&zero==0&&sign==1))PCSrc=2'b01;//beq bltz
    else PCSrc=2'b00;
end
else PCSrc=2'b00;

//RegDst
if(state==3'b001&&opcode==6'b111010)RegDst=2'b00;
else if(state==3'b11) begin
    if((opcode==6'b000010)||| (opcode==6'b10010)||| (opcode==6'b100111)||| (opcode==6'b110001)) RegDst=2'b01;//addi ori sltiu lw
    else RegDst=2'b10;
end
else RegDst=2'b10;

```

```

//ALUOp
if(state==3'b010)begin
    if(opcode==6'b000001)ALUOp=3'b001;//sub alu is -
    else if((opcode==6'b010000)||| (opcode==6'b010010))ALUOp=3'b101;//or ori alu is |
    else if(opcode==6'b010001)ALUOp=3'b110;//and alu is &
    else if(opcode==6'b011000)ALUOp=3'b100;//sll alu is <<
    else if(opcode==6'b100110)ALUOp=3'b011;//slt alu is compare_with_sign
    else if(opcode==6'b100111)ALUOp=3'b010;//sltiu alu is no sign compare
    else if((opcode==6'b110100)||| (opcode==6'b110110))ALUOp=3'b111;//beq bltz
    else ALUOp=3'b000;
end
else ALUOp=3'b000;

//isHalt
if(opcode==6'b111111)isHalt=1;
else isHalt=0;

end

always@(posedge CLK) begin//next state wheather pc can write
if(state==3'b001)begin
    if((opcode==6'b111000)||| (opcode==6'b111001)||| (opcode==6'b111010))PCWre<=1;//j jal jr
    else PCWre=0;
end
else if(state==3'b010)begin
    if((opcode==6'b110100)||| (opcode==6'b110110))PCWre<=1;//beq bltz
    else PCWre=0;
end
else if(state==3'b100)begin
    if(opcode==6'b110000)PCWre<=1;
    else PCWre=0;
end
else if(state==3'b011)PCWre<=1;
else PCWre=0;
end
endmodule

```

(11)顶层文件——将各模块连接起来，实现代码如下

```

module MultiCPU(
    input CLK,
    input reset,
    output [31:0] curPC,
    output [31:0] nextPC,
    output [31:0] ReadData1,
    output [31:0] ReadData2,
    output [4:0] rs,
    output [4:0] rt,
    output [31:0] result,
    output [31:0] DataOut,
    output [2:0] state
);

wire PCWre, InsMemRW, IRWre, sign, zero, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, mRD, mWR, isHalt, ExtSel, WrRegDSrc;
wire [1:0] RegDst, PCSrc;
wire [2:0] ALUOp, State;
wire [4:0] WriteReg;
wire [31:0] readData1, readData2, Result, dataOut, NextPC, IAddr, address, InsOut, WriteData, PC4, DBDataIn, DBDataOut, Data1, Data2, ExtOut, ExtendSa, InA, InB, JumpPC, SkipPC, DAddr;
wire [4:0] Reg31;

assign Reg31=5'b11111;

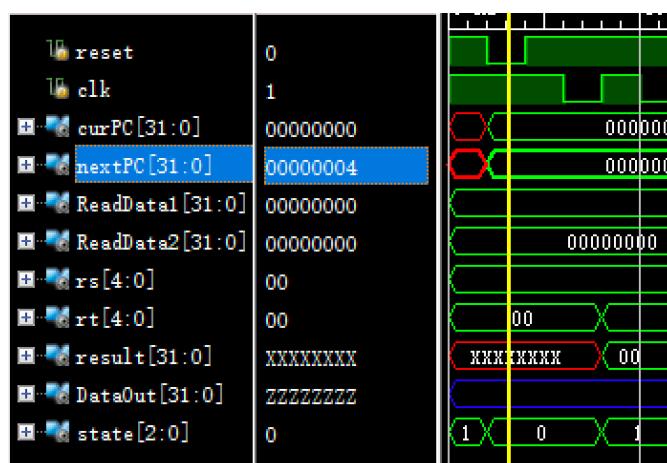
PC pc(CLK, reset, NextPC, PCWre, IAddr);
AddPC1 pc_4(IAddr, PC4);
AddPC2 sp(PC4, ExtOut, SkipPC);
jumpPC j1(PC4, InsOut[25:0], JumpPC);
NextPC selectNextPC(IAddr, PC4, SkipPC, readData1, JumpPC, PCSrc, isHalt, NextPC);
ROM rom(InsMemRW, IAddr, address);
IR InReg(CLK, IRWre, address, InsOut);
ControlUnit cu(zero, sign, reset, CLK, InsOut[31:26], RegDst, WrRegDSrc, InsMemRW, IRWre, PCWre, ExtSel, DBDataSrc, mWR, mRD, ALUSrcB, ALUSrcA, PCSrc, ALUOp, RegWre, isHalt, State);
ChooseWriteReg select_2_5(Reg31, InsOut[20:16], InsOut[15:11], RegDst, WriteReg);
ChooseWriteData selectWriteData(PC4, DBDataOut, WrRegDSrc, WriteData); // (PC4, DBDataOut, WrRegDSrc, WriteData);
RegFile regfile(CLK, reset, RegWre, InsOut[25:21], InsOut[20:16], WriteReg, WriteData, readData1, readData2);

endmodule

```

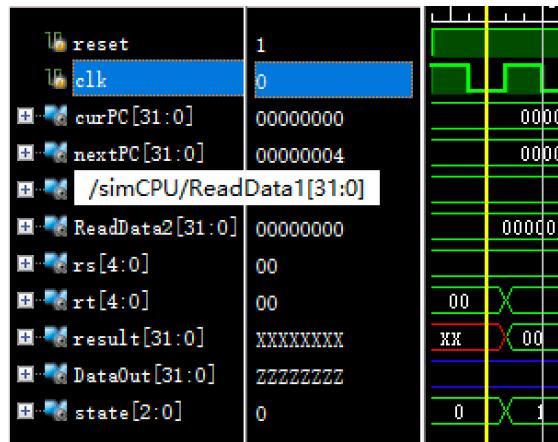
2. 指令执行情况

- (1) 初始时 —— 初始情况时reset为0，CPU未开始工作，PC的输出地址不变，故当前地址为0，下一条地址为4。

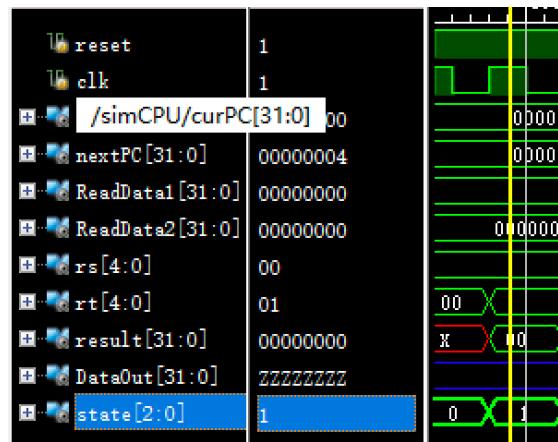


(2)立即数相加——addi \$1, \$0, 8

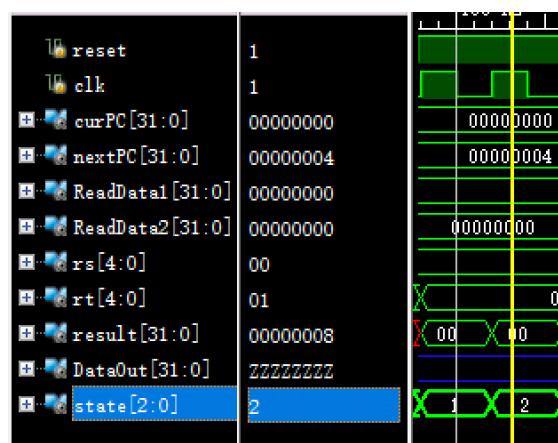
首先第一阶段取指令，取出的第一条指令地址为0x00000000， 默认递增的下一条指令为0x00000004， 其他部分不变



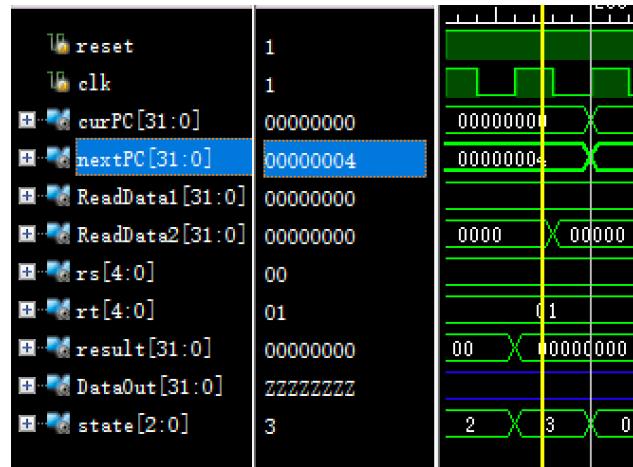
接着，下一次时钟上升沿，使CPU进入第二阶段指令译码，此时将rt赋值为1，代表1号寄存器，其他部分不变。



接着，下一次时钟上升沿时CPU进入第三阶段，指令执行，此时ALU会执行命令，计算 $0+8$ ，并将结果8输出。

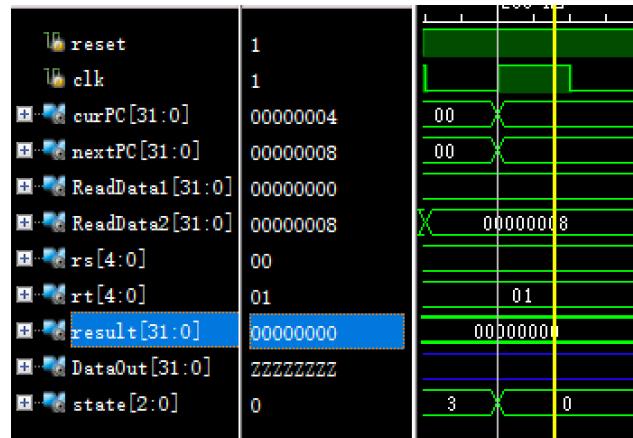


然后，由于立即数加法不经过存储器访问阶段，在下一次时钟上升沿时CPU进入第四阶段，结果写会。在时钟下降沿时寄存器1的值变为8.

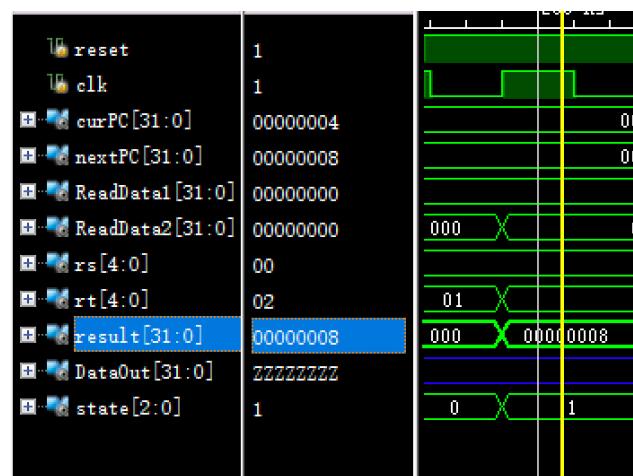


(3)立即数的或运算——ori \$2,\$0,2

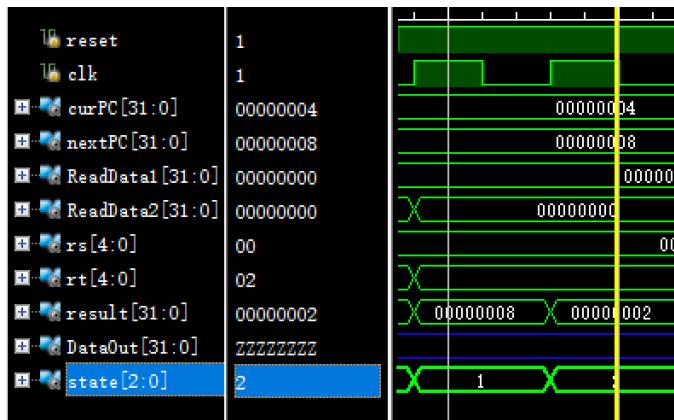
第一阶段取指令，获得的指令为0x00000004，下一条指令为0x00000008，其他部分不变。



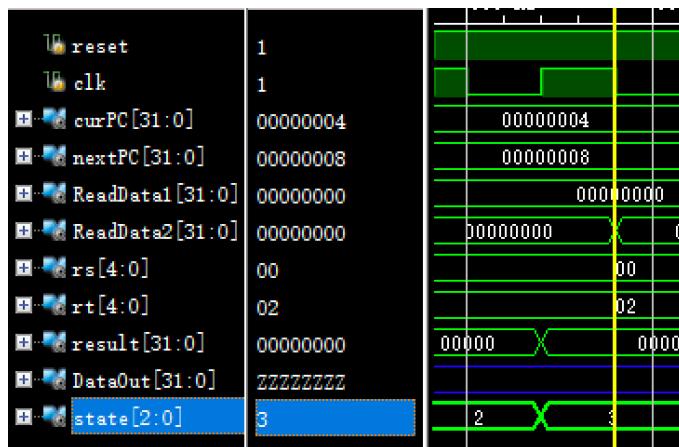
第二阶段指令译码，rs表示0号寄存器，rt表示2号寄存器，其他部分不变



第三阶段指令执行，进行ALU计算，输出结果为2，其他不变

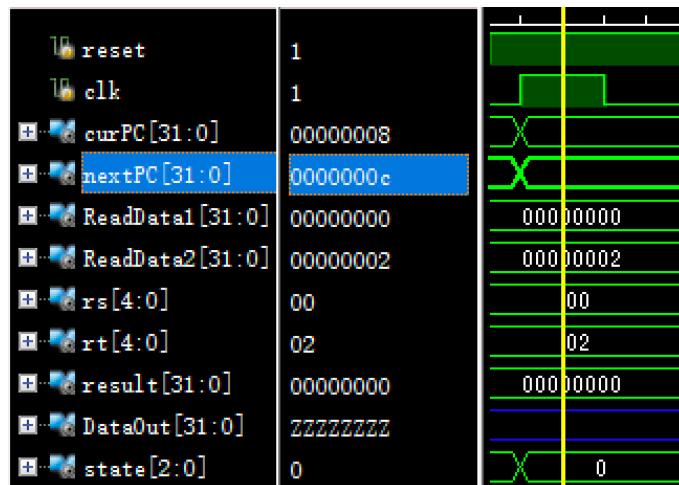


第四阶段，由于不经过储存器访问步骤，所以进行结果写回，在时钟下降沿时2号寄存器的值变为2

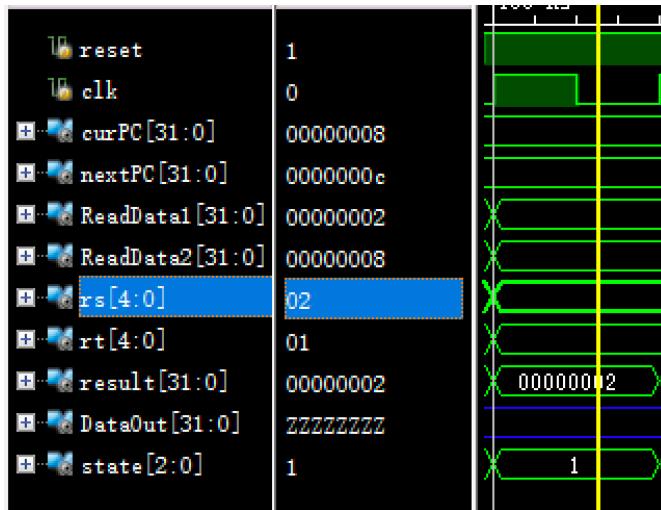


(4)寄存器的或运算—— or \$3,\$2,\$1

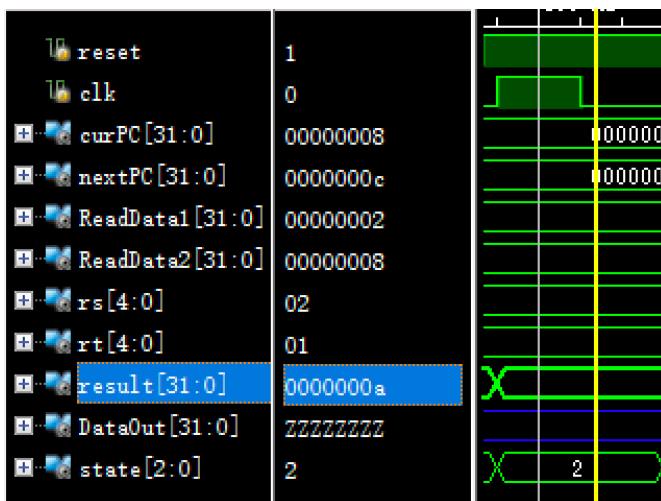
同样第一阶段取指令，进行指令变化，下一条指令递增变为0x0000000C



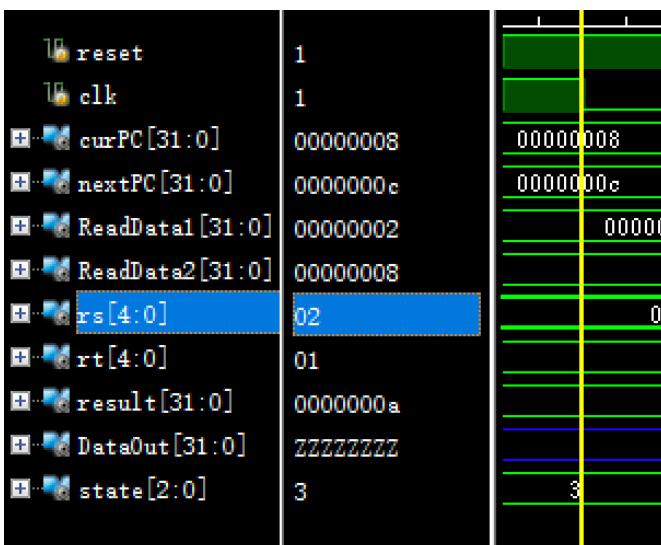
然后第二阶段，指令译码，rs表示2号寄存器，rt表示1号寄存器



接着指令执行，ALU执行或运算，将8与2取或，得到10 (a)

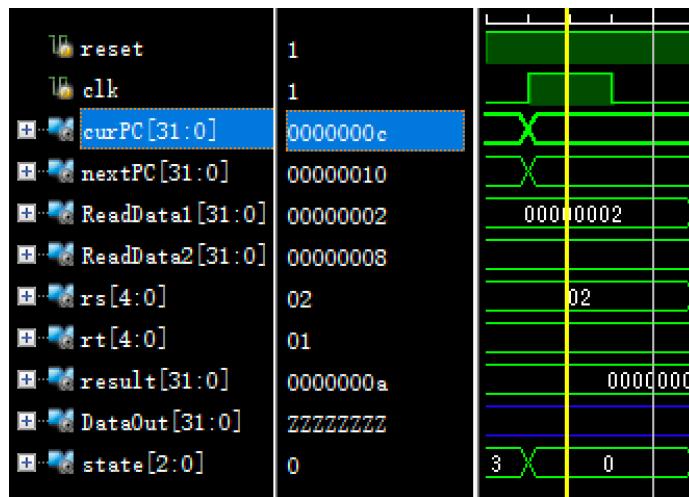


然后结果写回，在时钟下降沿时，将结果写入到3号寄存器中，由于输出并不显示3号寄存器，及其值，所以这一阶段波形图出来状态变化，其他并不变。

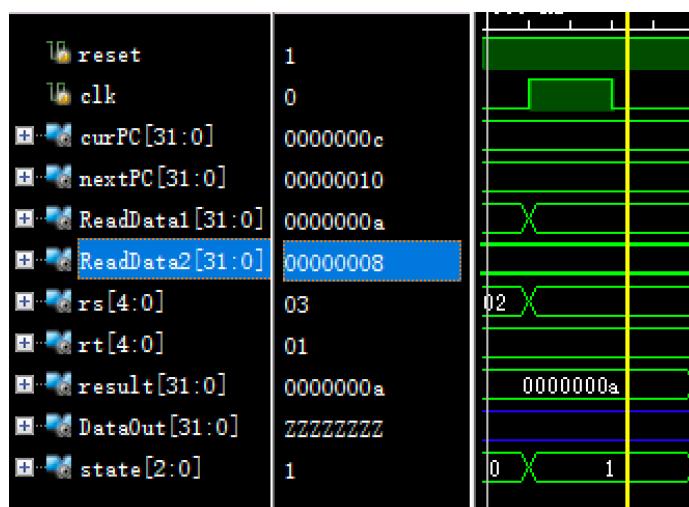


(5)寄存器的减法运算——sub \$4,\$3,\$1

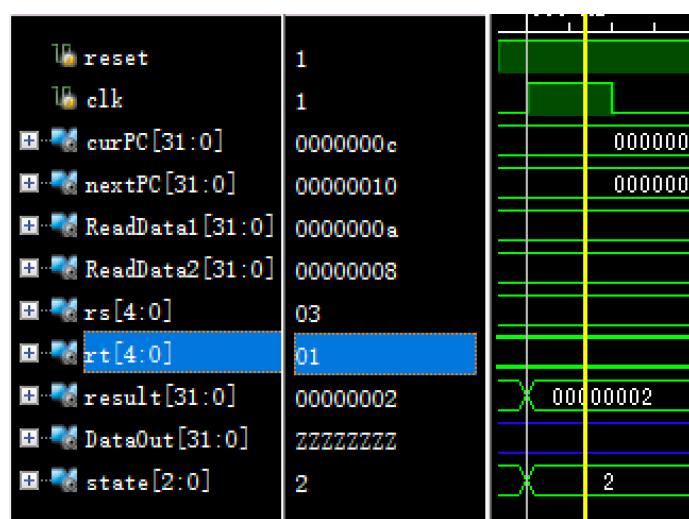
首先取指令，当前指令变为0x0000000C，下一条指令变为0x00000010



接着指令译码，rs表示3号寄存器，rt表示1号寄存器



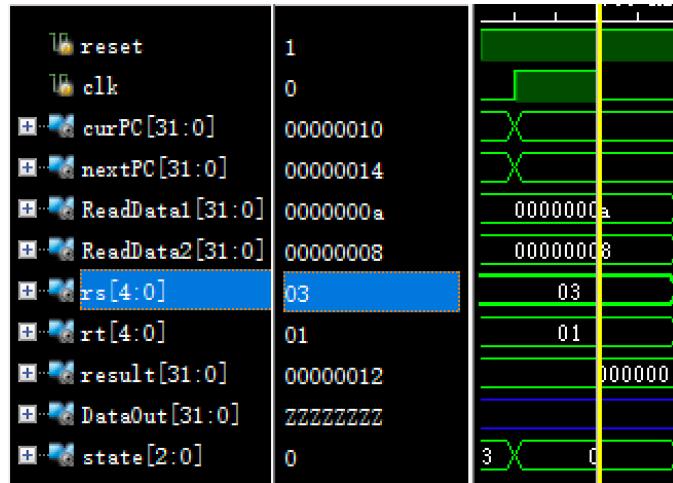
然后进行指令执行，ALU进行减法运算，将3号寄存器里的10减去1号寄存器里的8，结果得到2



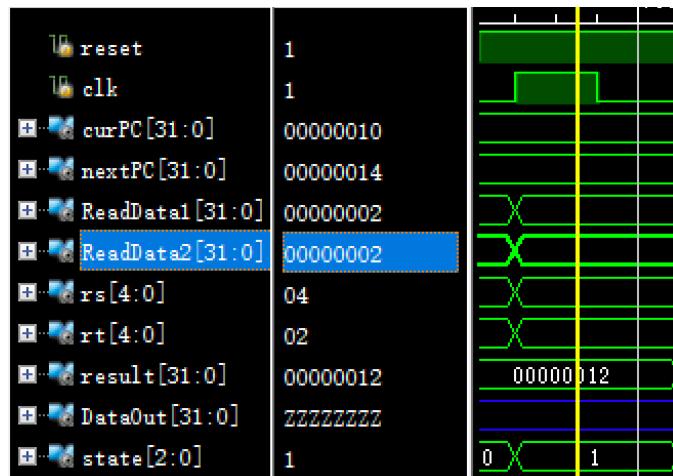
同样，第四阶段，将计算结果写入4号寄存器。

(6)与运算——and \$5,\$4,\$2

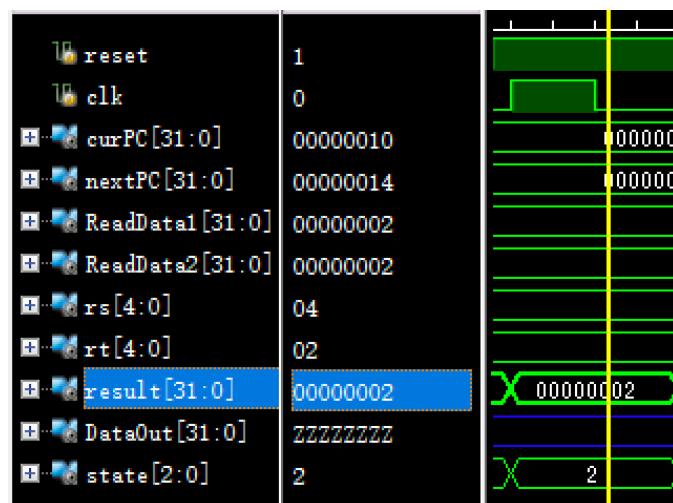
第一阶段，当前地址变为0x00000010，下一条地址顺延到0x00000014



第二阶段，rs表示4号寄存器，rt表示2号寄存器，4号寄存器中的值为2，2号寄存器中的值也为2



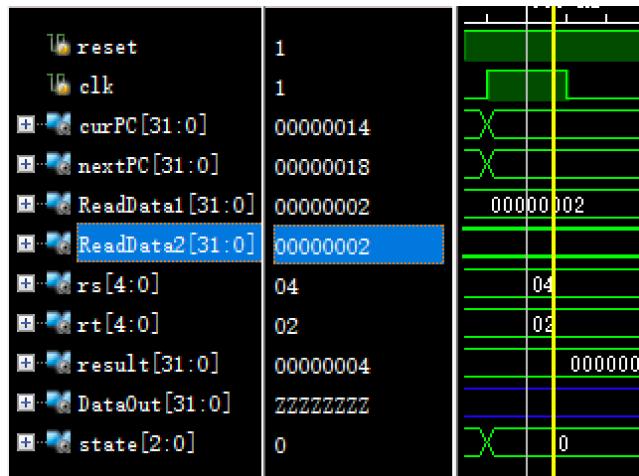
第三阶段，ALU进行与运算，对两个2取或，得到结果2



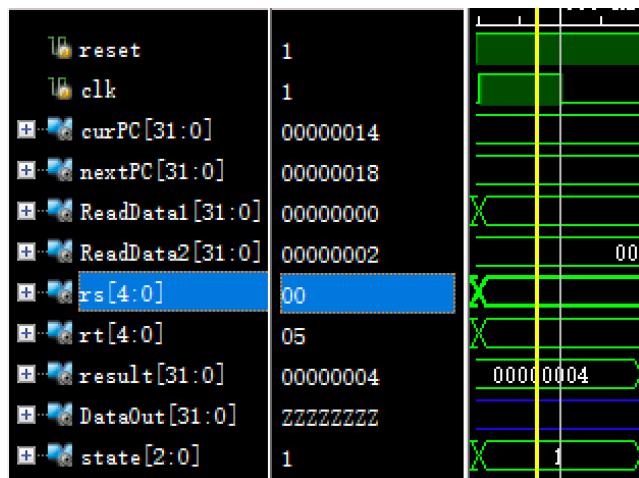
第四阶段，结果写回，将2写入5号寄存器

(7)位移运算——sll \$5,\$5,2

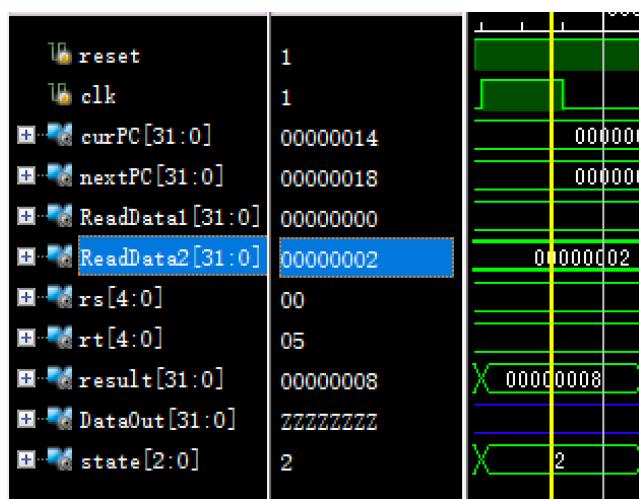
第一阶段，指令读取，当前指令变为0x00000014，下一条指令顺延至0x00000018



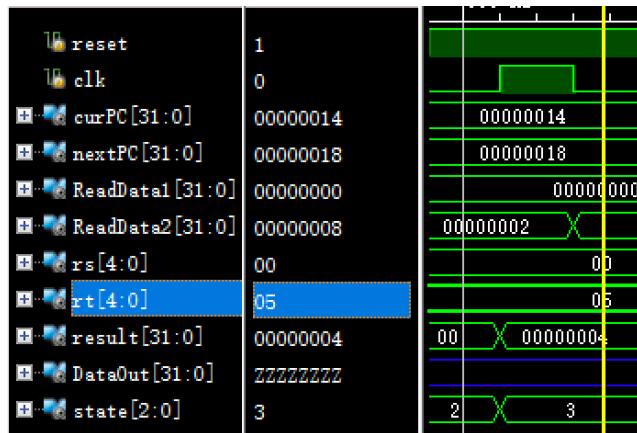
第二阶段，指令译码，由于有立即数，所以rs变为0，rt表示5号寄存器，其中的值为2



第三阶段，指令执行，ALU将2左移两位，相当于2乘4，得到结果为8

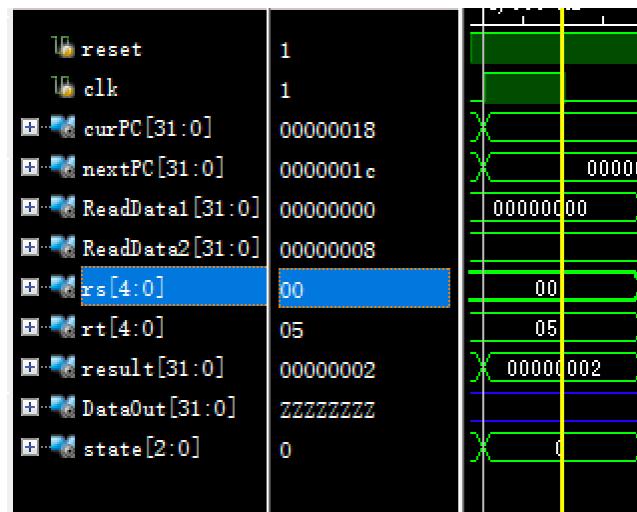


第四阶段，结果写回，在时钟下降沿时将8写入到5号寄存器中

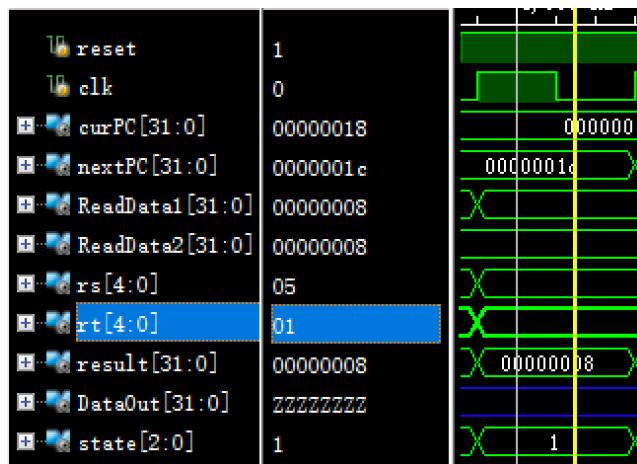


(8)相等时跳转指令——**beq \$5,\$1,-2(=,转14)**

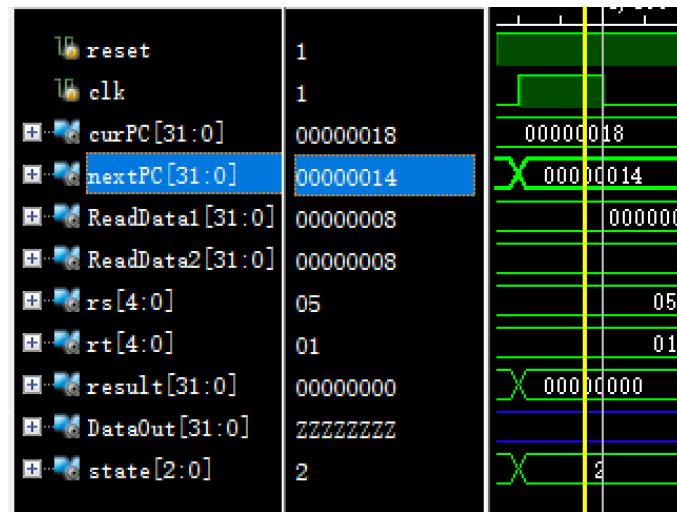
第一阶段，读取指令，当前指令变为0x00000018，下一条指令顺延为0x0000001C



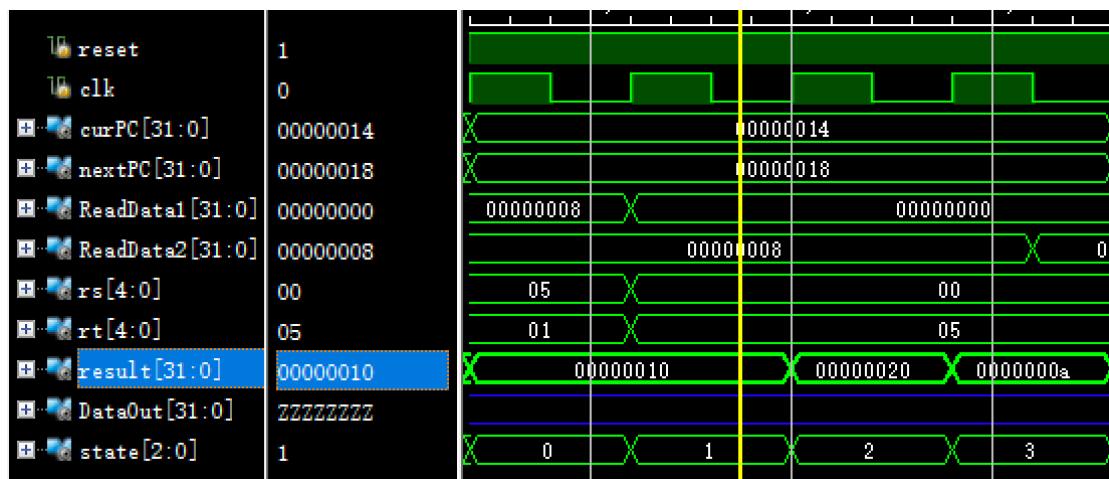
第二阶段，指令译码，rs表示5号寄存器，rt表示1号寄存器，其中的值都为8



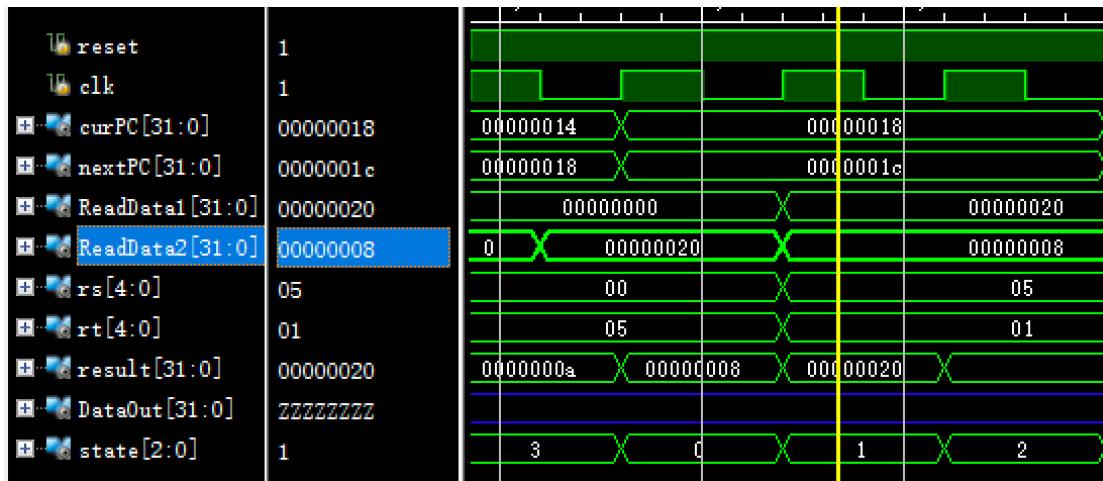
第三阶段，指令执行，判断5号寄存器和1号寄存器里的值是否相等，因为都为8，相等，所以结果返回0，并且-2计算应该跳转到0x00000014，将下一条指令地址改变为0x00000014



由于没有存储器访问操作和结果写回操作，所以直接跳转到0x00000014处执行
同样第一阶段当前指令变为0x00000014，下一条指令顺延至0x0000001，第二阶段rt表示5号寄存器，其中的值为8，rs为0，之后一个阶段进行左移运算，将8变为32（20），然后第四阶段，时钟下降沿时将32写入5号寄存器

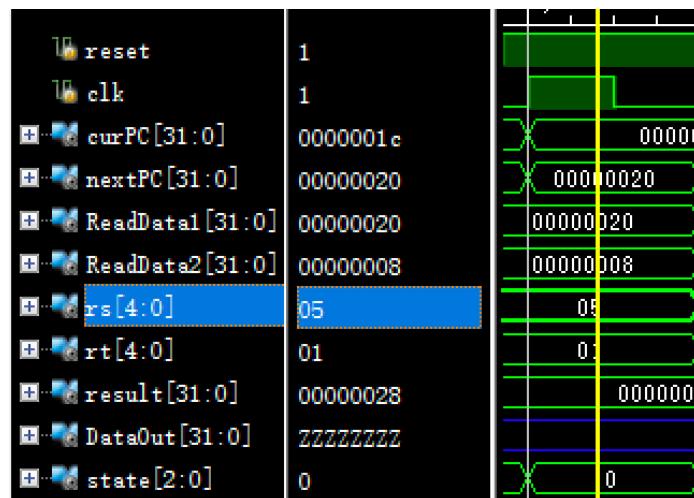


此时在执行一次相等时跳转指令，由于5号寄存器中的值已经变成了32，所以不再相等，ALU计算结果返回为1，下一条指令保持0x0000001C不变

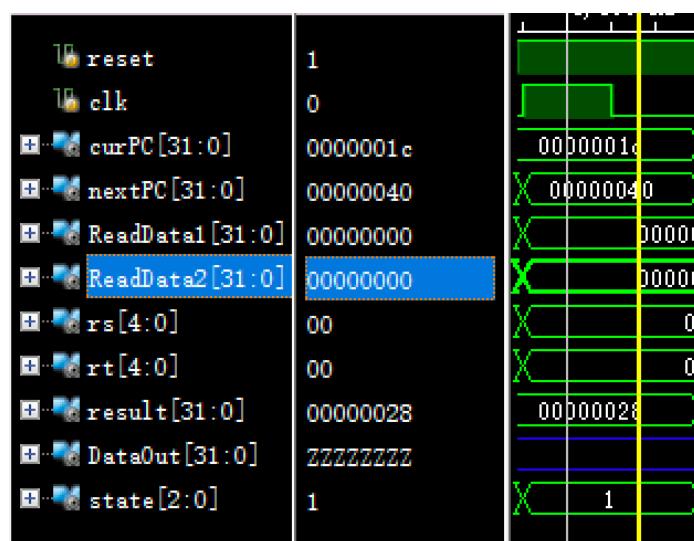


(9)调用子程序指令——jal 0x00000040

首先第一阶段取指令，当前指令变为0x0000001C，下一条指令顺延至0x00000020



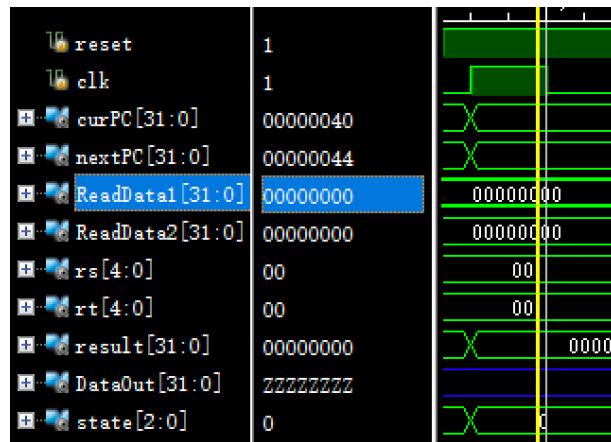
第二阶段，指令译码，发现需要跳转到的指令地址，将下一条指令的地址改变为0x00000040



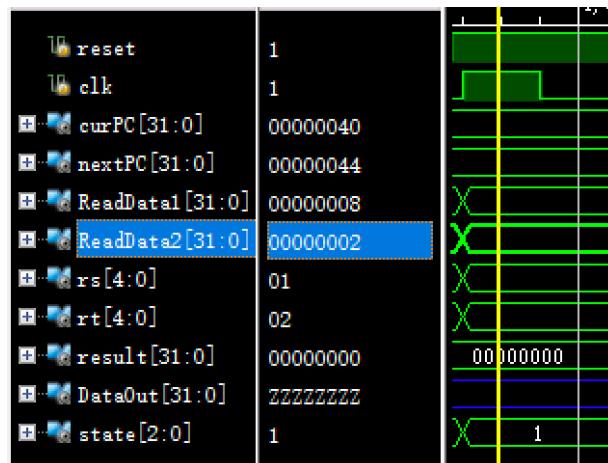
接着直接跳转，无指令执行和结果写回的步骤

(10)写储存器指令——sw \$2,4(\$1)

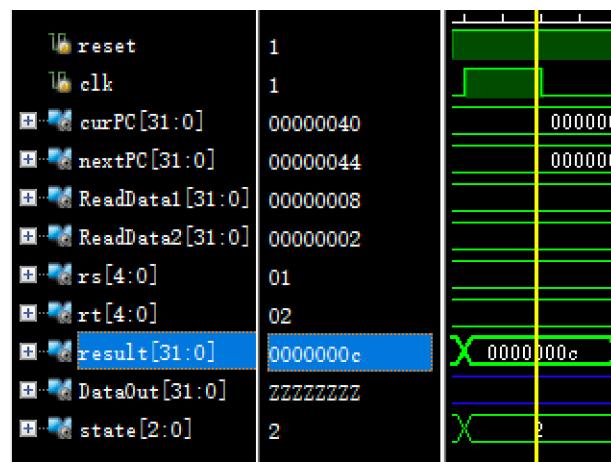
首先第一阶段，按照上一条子程序调用的指令，当前指令地址跳转到0x0000040，
下一条指令地址顺延到0x00000044



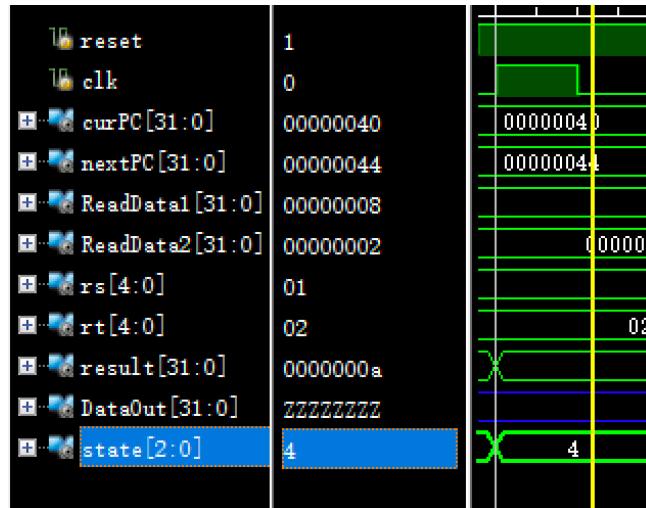
接着第二阶段，指令译码，rs表示寄存器1，rt表示寄存器2



然后第三阶段，指令执行，ALU计算1号寄存器的值加4，因为1号寄存器的值为8，
所以结果为12 (c)

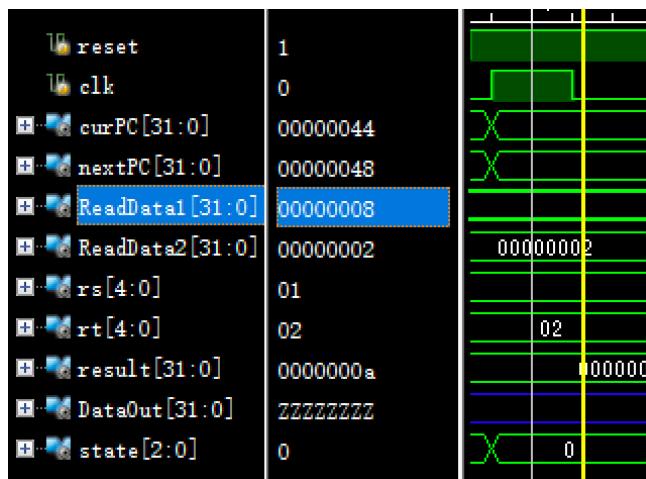


第四阶段，执行储存器访问状态，将数据写入到指定的储存单元

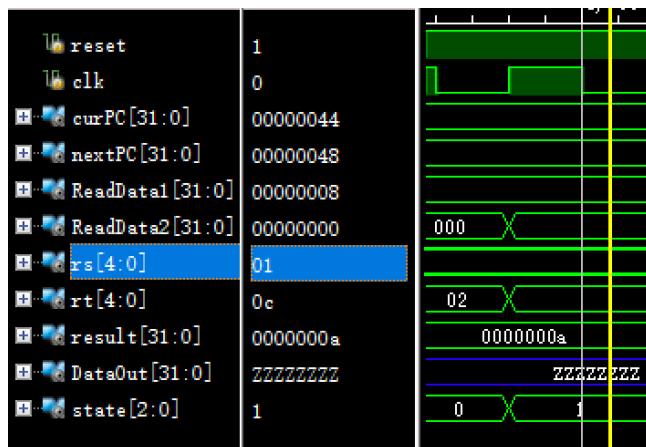


(11)读储存器指令——lw \$12,4(\$1)

第一阶段，读取指令，当前指令变为0x00000044，下一条指令顺延为0x00000048



第二阶段，指令译码，rs表示1号寄存器，rt表示12号寄存器



第三阶段，指令执行，ALU计算1号寄存器的值加4，因为1号寄存器的值为8，所以结果为12 (c)

reset	1	
clk	1	
curPC[31:0]	00000044	
nextPC[31:0]	00000048	
ReadData1[31:0]	00000008	
ReadData2[31:0]	00000000	00
rs[4:0]	01	
rt[4:0]	0c	
result[31:0]	0000000c	X 0000000c
DataOut[31:0]	zzzzzzzz	zzzzzzzz
state[2:0]	2	X 2

第四阶段，存储器访问，从memor[12]中上一条命令写入的值2，并改变输出的高阻态，输出2

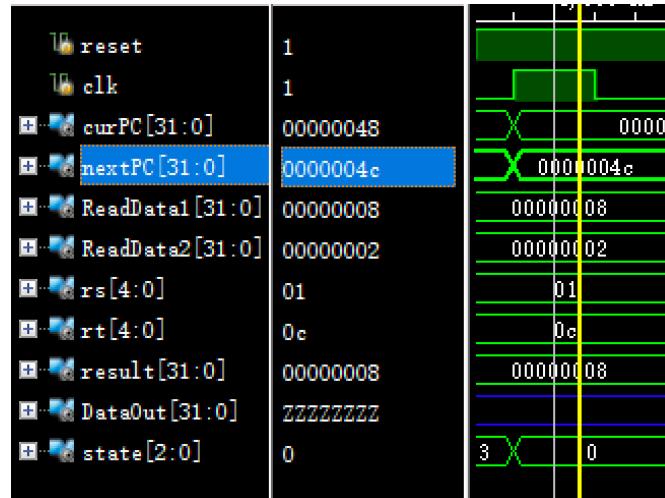
reset	1	
clk	0	
curPC[31:0]	00000044	0000004
nextPC[31:0]	00000048	0000004
ReadData1[31:0]	00000008	
ReadData2[31:0]	00000000	00000000
rs[4:0]	01	
rt[4:0]	0c	
result[31:0]	00000008	
DataOut[31:0]	00000002	00000002
state[2:0]	4	2

第五阶段，结果写回，在时钟下降沿时将输出2写入到12号寄存器中

reset	1	
clk	0	
curPC[31:0]	00000044	00000044
nextPC[31:0]	00000048	00000048
ReadData1[31:0]	00000008	00000008
ReadData2[31:0]	00000002	0000 X 0
rs[4:0]	01	01
rt[4:0]	0c	0c
result[31:0]	00000008	00000008
DataOut[31:0]	zzzzzzzz	zzzzzzzz
state[2:0]	3	X 3

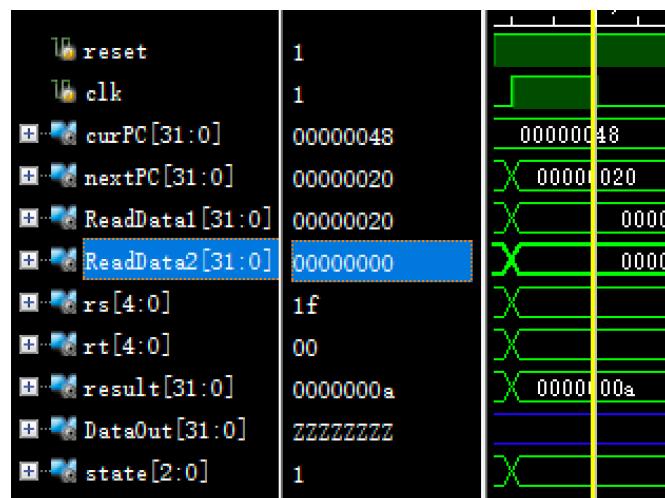
(12) 跳转指令--jr \$31

第一阶段，取指令，当前指令变为0x00000048，下一条指令顺延到0x0000004C



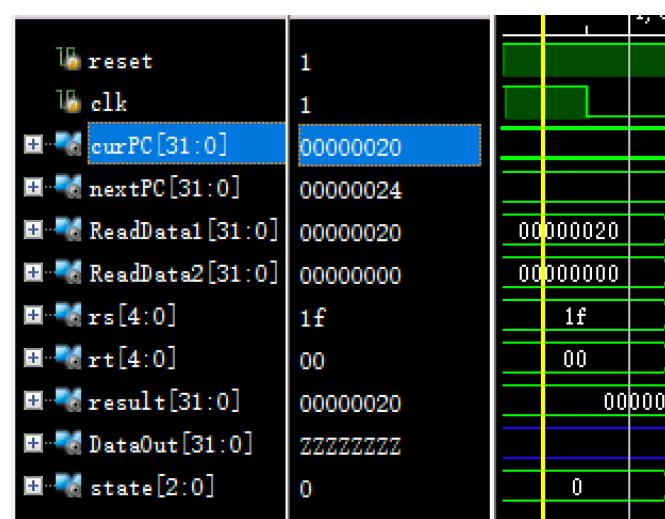
第二阶段，指令译码，得到需要跳转到的指令位置为31号寄存器中的值，其值为20

于是将下一条指令的地址改变为0x00000020

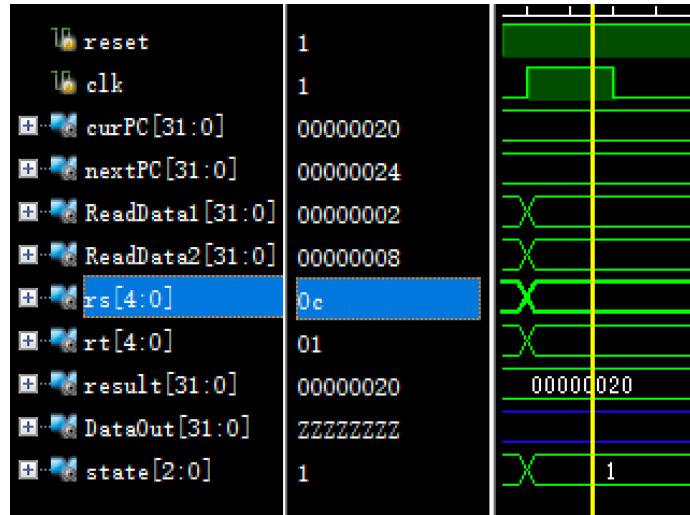


(13) 比较指令--slt \$8,\$12,\$1

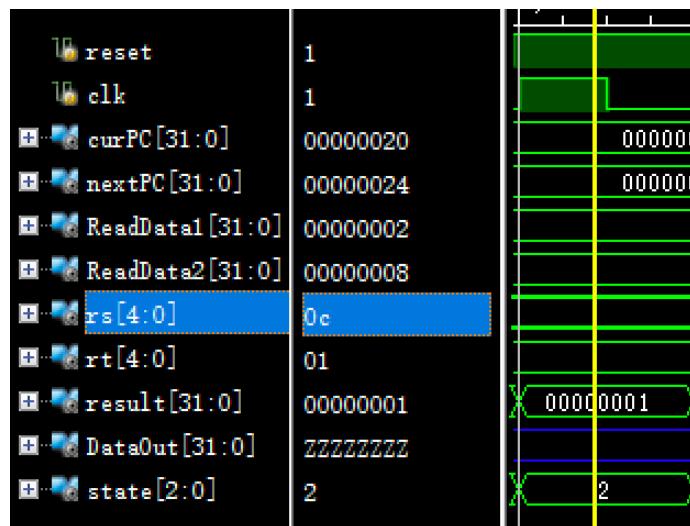
第一阶段取指令，当前指令变为0x00000020，下一条指令顺延为0x00000024



第二阶段，指令译码，rs表示12号寄存器，其值为2，rt表示1好寄存器，其值为8

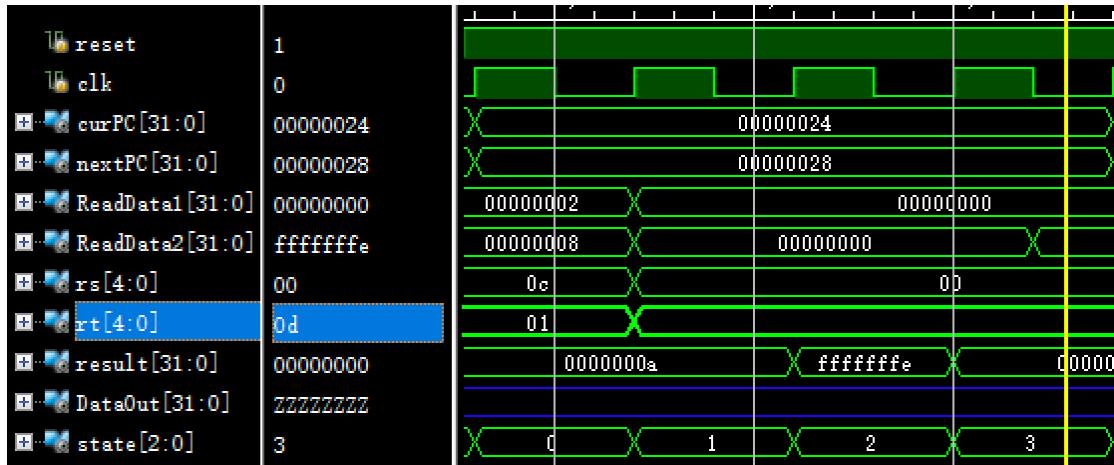


第三阶段，指令执行，ALU比较rs和rt的值，由于rs<rt所以输出结果为1

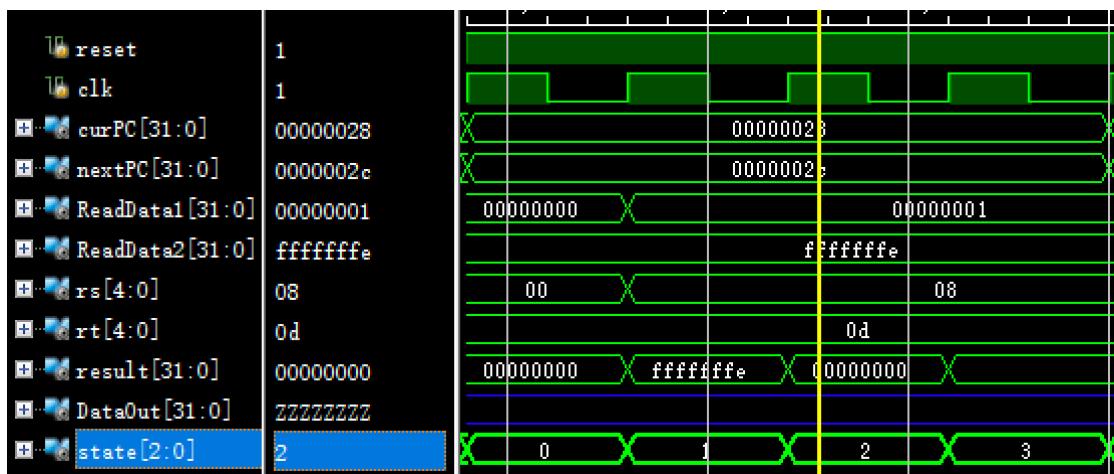


之后第四阶段，将1下时钟下降沿时写入8号寄存器中。（由于波形图基本没有变化，所以就不放截图了）

然后执行立即数加法指令 — addi \$13,\$0,-2，第一阶段先将当前指令地址变为0x00000024，下一条指令的地址执行顺延到0x00000028，接着指令译码，rt表示为13号寄存器，接着进入第三阶段，ALU计算0+(-2)的到结果-2，并在第四阶段，时钟下降沿时将其写入13号寄存器中



然后再执行比较指令 —— `slt $9,$8,$13`，第一阶段先将当前指令地址变为 `0x00000028`，下一条指令的地址执行顺延到 `0x0000002c`，接着指令译码，rs 表示 8 号寄存器，其值为 1，rt 表示为 13 号寄存器，其值为 -2，接着进入第三阶段，ALU 比较 rs 与 rt 的大小，因为 $rs > rt$ ，所以计算结果返回 0，并在第四阶段，时钟下降沿时将 0 写入 9 号寄存器中

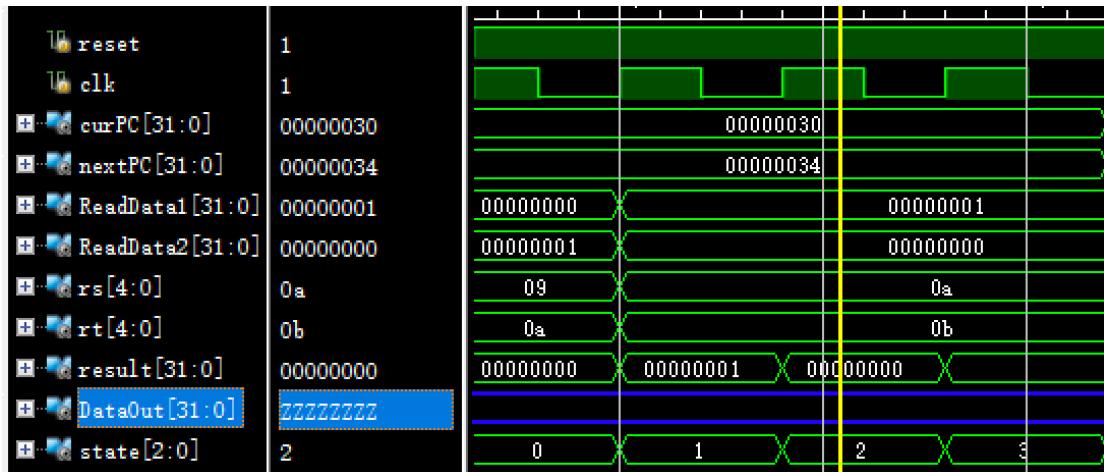


(14) 不带符号立即数比较指令 —— `sltiu $10,$9,2`

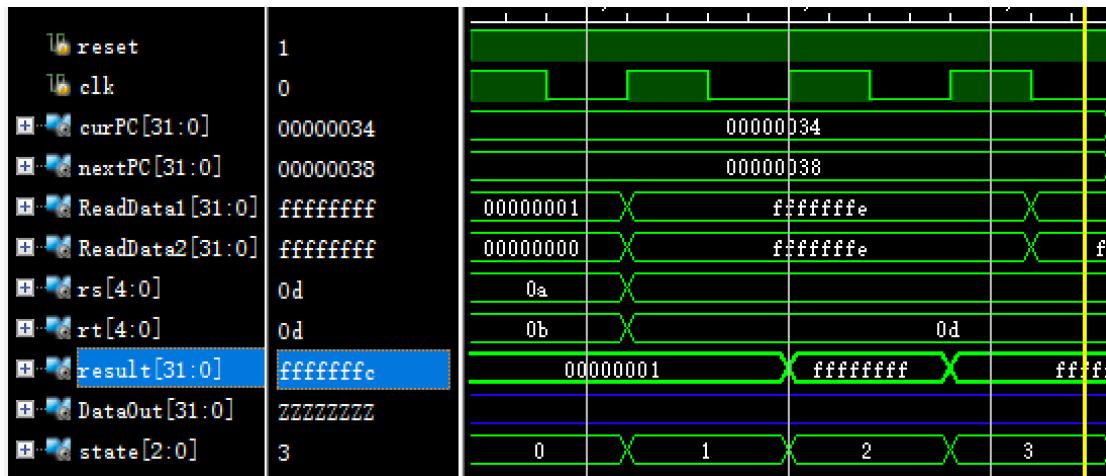
同样，首先更新指令地址，将当前指令地址变为 `0x0000002c`，下一条指令的地址执行顺延到 `0x00000030`，接着指令译码，rs 表示 9 号寄存器，其值为 0，rt 表示为 10 号寄存器，其值也为 0，接着进入第三阶段，ALU 比较 rs 与无符号扩展后的立即数的大小，因为 $rs < 2$ ，所以计算结果返回 1，并在第四阶段，时钟下降沿时将 1 写入 10 号寄存器中



紧接着又是一条同样类型的比较指令——**sltiu \$11,\$10,0**，首先更新指令地址，将当前指令地址变为0x00000030，下一条指令的地址执行顺延到0x00000034，接着指令译码，rs表示10号寄存器，其值为1，rt表示为11号寄存器，其值为0，接着进入第三阶段，ALU比较rs与无符号扩展后的立即数的大小，因为rs>0，所以计算结果返回0，并在第四阶段，时钟下降沿时j将1写入11号寄存器中

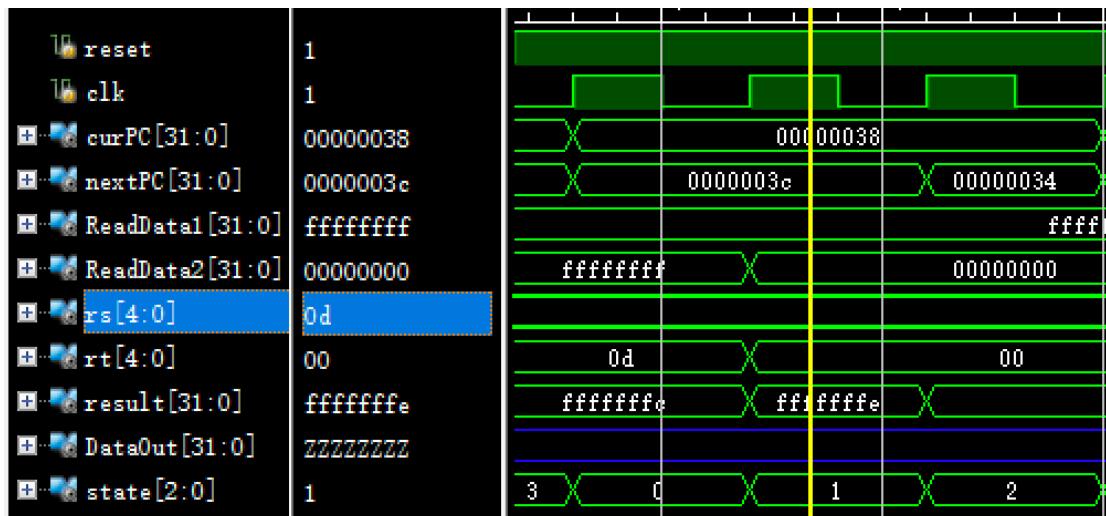


然后，我们在执行一条立即数加法的指令——**addi \$13,\$13,1**，首先更新指令地址，将当前指令地址变为0x00000034，下一条指令的地址执行顺延到0x00000038，接着指令译码，rs，rt均表示为13号寄存器，其值为-2，接着进入第三阶段，ALU进行加法计算，计算-2+1的值，并输出结果-1，接着在第四阶段，时钟下降沿时j将-1写入13号寄存器中



(15)与0比较的分支指令——bltz \$13,-2 (<0,转34)

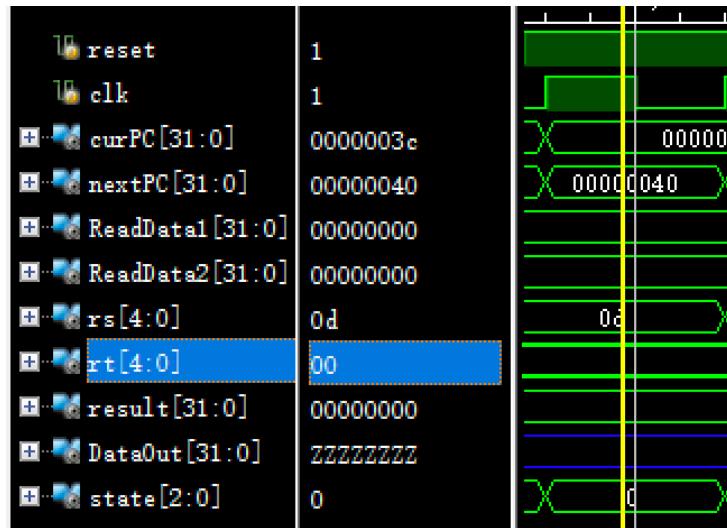
同样第一阶段，取指令，将当前指令更新为0x00000038，下一条指令顺延到0x0000003C。然后，第二阶段指令译码，rs表示13号寄存器，其值为-1，rt为0。第三阶段，ALU比较rs的值域0的大小，由于-1<0，所以根据立即数计算出要跳转到0x00000034，并将下一条指令的地址改变为0x00000034



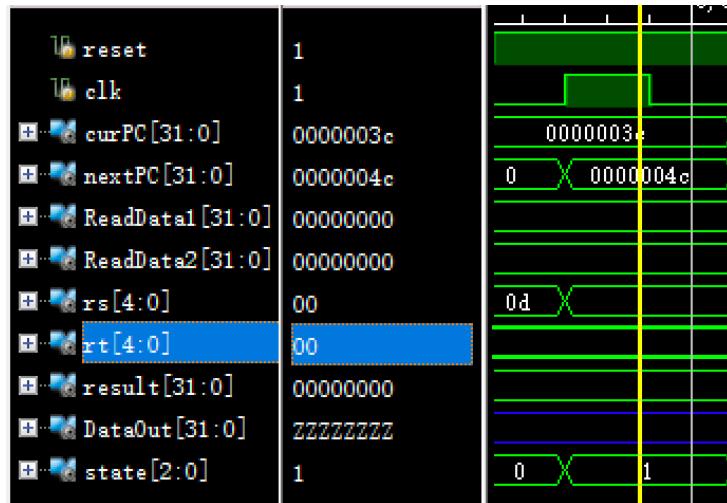
由于跳转到34，所以重新执行一次立即数加法指令，将13号寄存器中的值变为0，然后再执行比较指令时，因为0不再小于0，所以不会跳转，下一条指令的地址为0x0000003C。

(16)跳转指令——j 0x0000004C

第一阶段，取指令，将当前指令地址更新为0x0000003C，下一条指令地址顺延至0x00000040

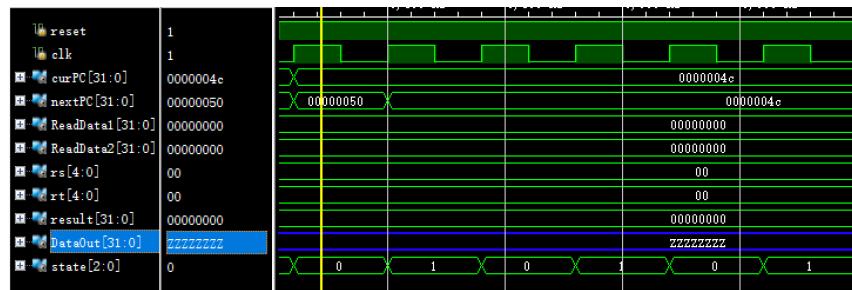


第二阶段，指令译码，得到需要跳转的指令地址，将下一条指令的地址更新为0x0000004C

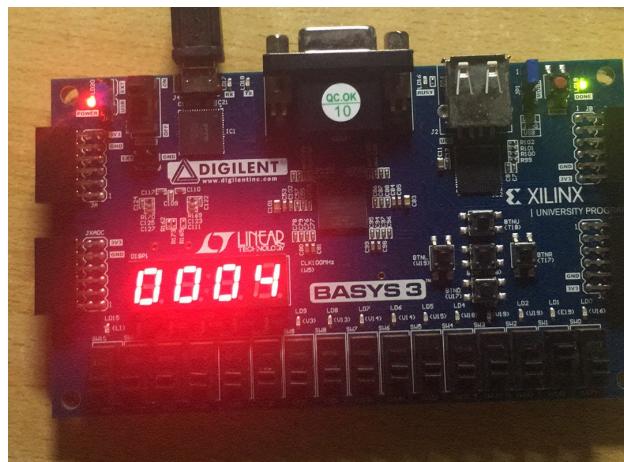


(17)停机指令——halt

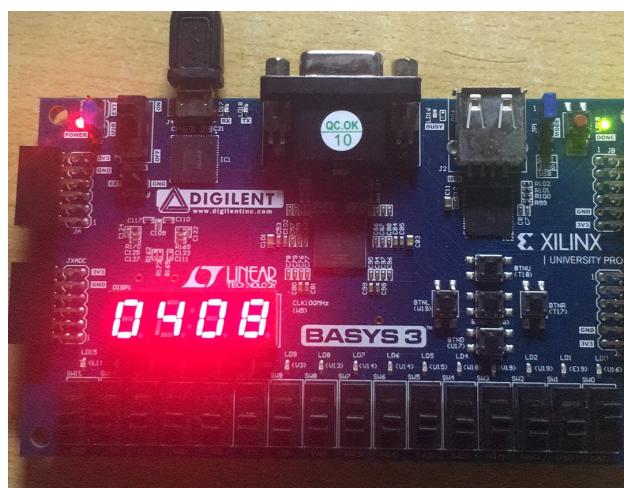
第一阶段更新当前指令地址为0x0000004C，然后顺延到下一条指令地址0x00000050，接着第二阶段，指令译码，识别出停机指令，之后状态不再改变。



3. 烧板

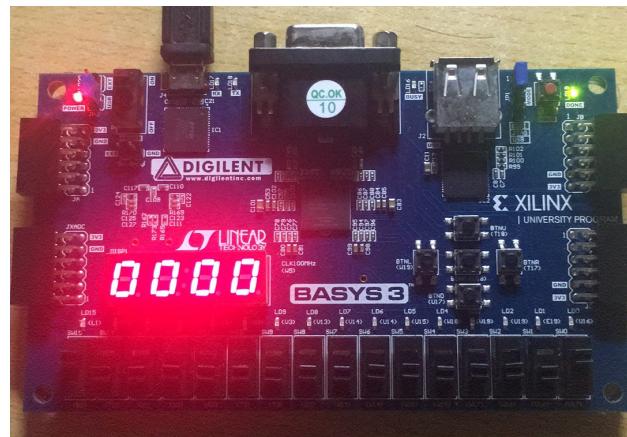


LED灯左边两位显示当前指令地址，右边两位表示下一条指令地址，最右边的开关为reset，右边倒数第二个开关模拟时钟信号。左边第一个开关可以在LED上面显示寄存器rs的值，左边第二个开关可以显示寄存器rt的值。左边第三个开关可以显示当前处于指令的那哪一个状态

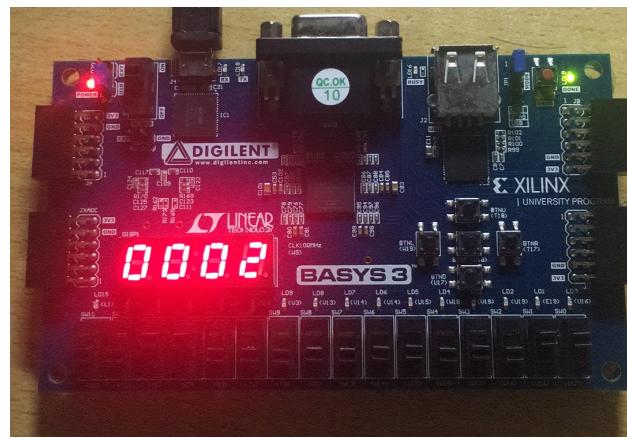




当开关模拟到上面一条指令的第二阶段时，两个寄存器中的值均为0



当前处于该指令的第二阶段



六. 实验心得

1. 与单周期CPU相比，多周期CPU的各模块设计并没有多大的区别，只是添加了一些新的功能。主要需要我们实现的是要利用触发器，达到将一条指令的运行过程分块的目的，而这也正是多周期CPU与单周期CPU最根本的区别。所以只要理解多周期CPU的原理，懂得如何利用触发器，看懂数据通路图就基本解决了我们设计时的难点，而这几个部分老师在课堂上和所给的资料中已经基本讲解清楚了，所以难度也不是特别大。

2. 在设计过程中，如果我们的设计对即时性要求比较高，或者电路比较复杂，我们就需要尽量避免利用reg的格式去储存输出，而利用assign来代替。我初始时是依旧按照之前单周期CPU的实现方式，利用reg实现的各个模块输出，但是在后来的仿真和烧板过程中，就会出现竞争与冒险的情况，这主要原因是因为在每次利用reg格式去输出的时候是会出现延迟的，而这种延迟在单周期CPU上不会有特别大的影响，但是在多周期CPU中因为各过程始终需要时钟出发，而且数据通路的复杂性更高所以就会影响到我们的数据输出。但assign是即时的输出，可以最小化数据传输时的时间延迟，也就可以避免竞争与冒险的发生。

3. 利用分层分块的思想可以将一个大的工程或者项目化整为零，简化我们实现的过程，帮我们找到突破口。但是在分模块实现时，要注意检测每个模块的正确性，有时候往往一个小模块中的小错误，就会造成我们得不到输出，或者得到的输出结果错误。

4. 如果使用文件读入的方式去得到我们编写的程序的指令时，一定要注意路径的问题。因为我是虚拟机中进行设计的，导致在使用路径时会出现一些奇怪的问题。同样是vivado的项目，但在单周期CPU中可以读取到文件的路径写法，在多周期CPU的项目中却一直读取不了，后面重新创建了一次项目文件才解决了这个问题。

5. 在这次设计中，我还改正了再单周期CPU中模块繁复的问题，实现了相同类型模块的统一构建，简化了实现过程，减少了很多的无用功，而且个模块的用途也更佳明确。