《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 软件工程一（2）班
学 生 姓 名 ：	陈亚楠
学　　号 ：	16340041
时　　间 ：	2018　年　6　月　22　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<－rs + rt

（2）sub rd, rs, rt

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

完成功能：rd<－rs － rt

（3）addi rt, rs, **immediate**

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

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

==>逻辑运算指令

（4）or rd, rs, rt

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

功能：rd<－rs | rt

（5）and rd, rs, rt

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

功能：rd<－rs & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

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

==>移位指令

（7）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

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

==>比较指令

（8） slt rd, rs, rt 带符号数

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

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

（9） sltiu rt, rs,immediate 不带符号

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

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

==>存储器读写指令

（10）sw rt, **immediate(**rs)

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

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

（11）lw rt, **immediate**(rs)

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

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

==>分支指令

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

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

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

(13) bltz rs,**immediate**

110110	rs(5 位)	00000	**immediate**

功能：if(rs<0) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4

==>跳转指令

（14）j addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

（15）jr rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（16）jal addr

111010	addr[27:2]

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],2'b00}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（17）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变pc的值，pc保持不变。

三. 实验原理

多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段：

⑴ 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。

⑵ 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。

⑶ 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。

⑷ 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。

⑸ 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。

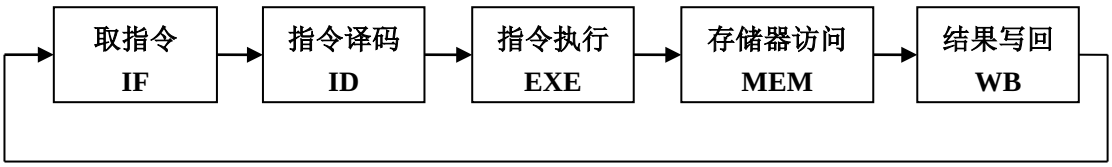
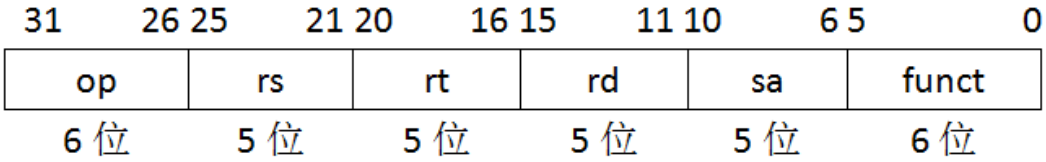


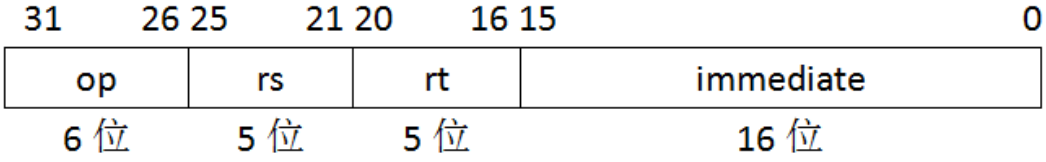
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式：

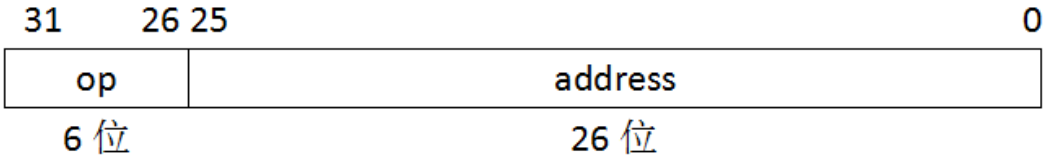
R 类型：



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。

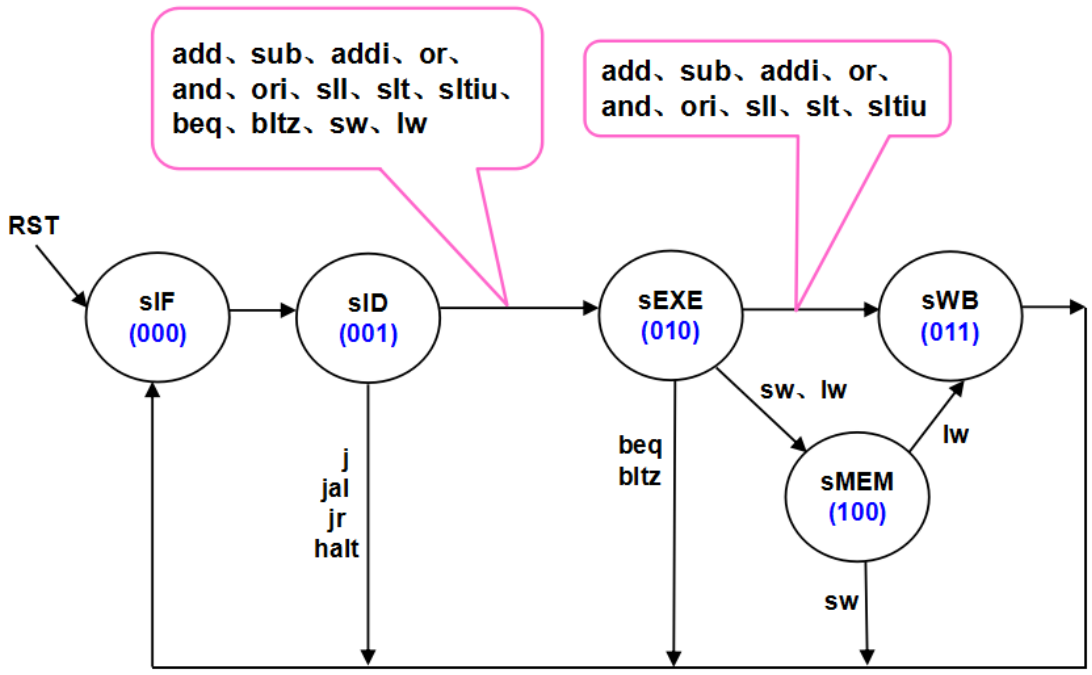


图 2 多周期 CPU 状态转移图

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。

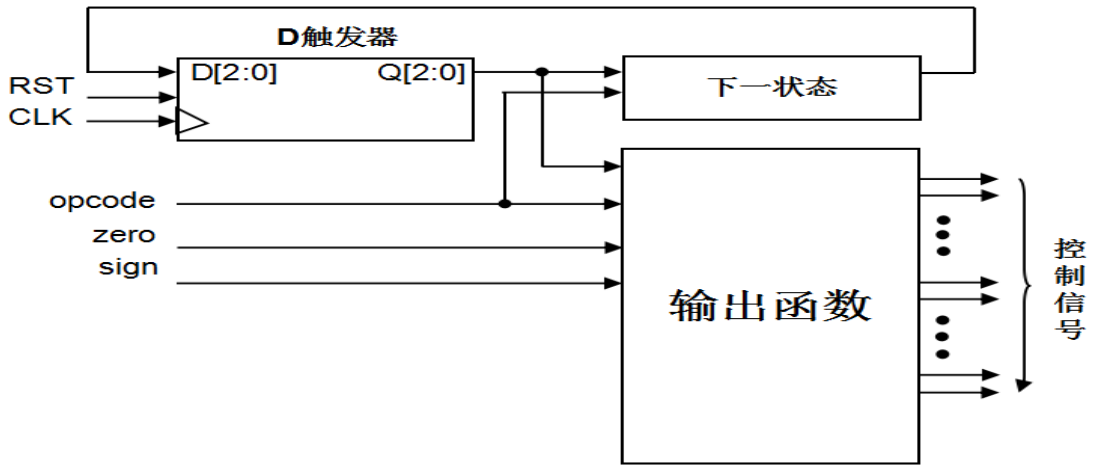


图 3 多周期 CPU 控制部件的原理结构图

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。

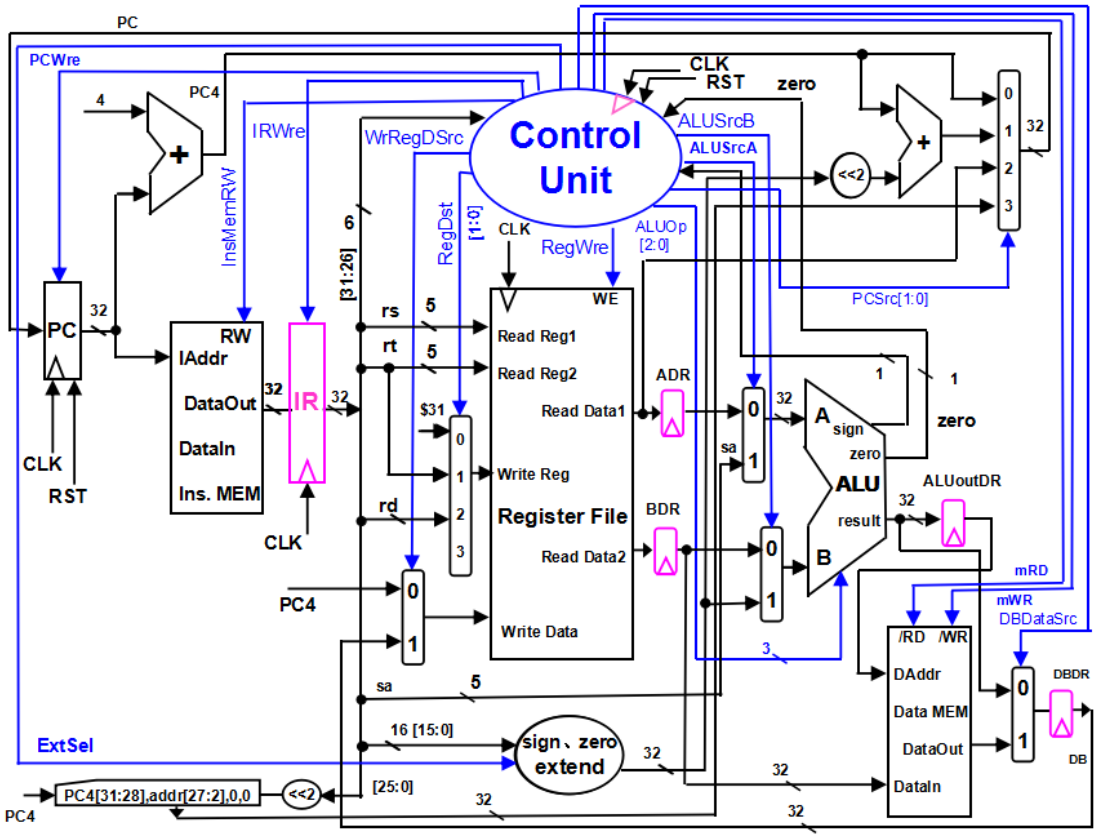


图 4　多周期 CPU 数据通路和控制线路图

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、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<<A	B 左移 A 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
111	Y = A ⊕ B	异或

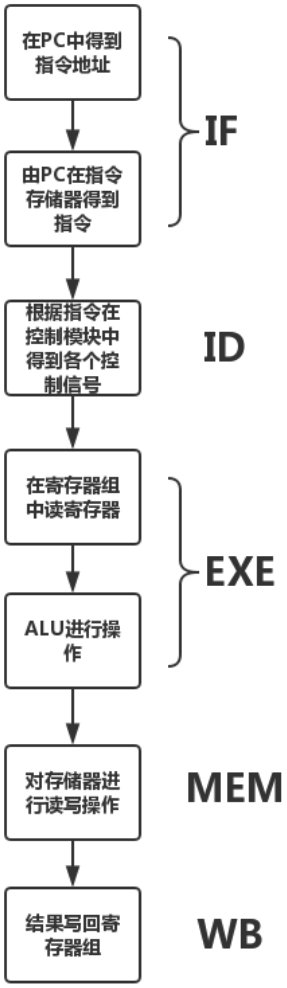
四. 实验器材

电脑一台，Xilinx Vivado 2018.1开发软件一套，Basys3实验板一块。

五. 实验过程与结果

1.CPU设计的思想、方法：

（1）CPU的工作流程图：



（2）建立控制信号真值表：

控制信号表给出了本次实验中的每个控制信号的作用，我们可以由该表看出各个控制信号与指令之间的关系，并根据这种关系建立控制信号与指令之间的真值表：

表3

Instruction	Op	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WrRegDSrc
add	000000	1	0	0	0	1	1
sub	000001	1	0	0	0	1	1
addi	000010	1	0	1	0	1	1
or	010000	1	0	0	0	1	1
and	010001	1	0	0	0	1	1
ori	010010	1	0	1	0	1	1
sll	011000	1	1	0	0	1	1

slt	100110	1	0	0	0	1	1
sltiu	100111	1	0	1	0	1	1
sw	110000	1	0	1		0	
lw	110001	1	0	1	1	1	1
beq(zero=0)	110100	1	0	0		0	
beq(zero=1)	110100	1	0	0		0	
bltz(zero=0)	110110	1	0	0		0	
bltz(zero=1)	110110	1	0	0		0	
j	111000	1				0	
jr	111001	1				0	
jal	111010	1				1	0
halt	111111	0				0	

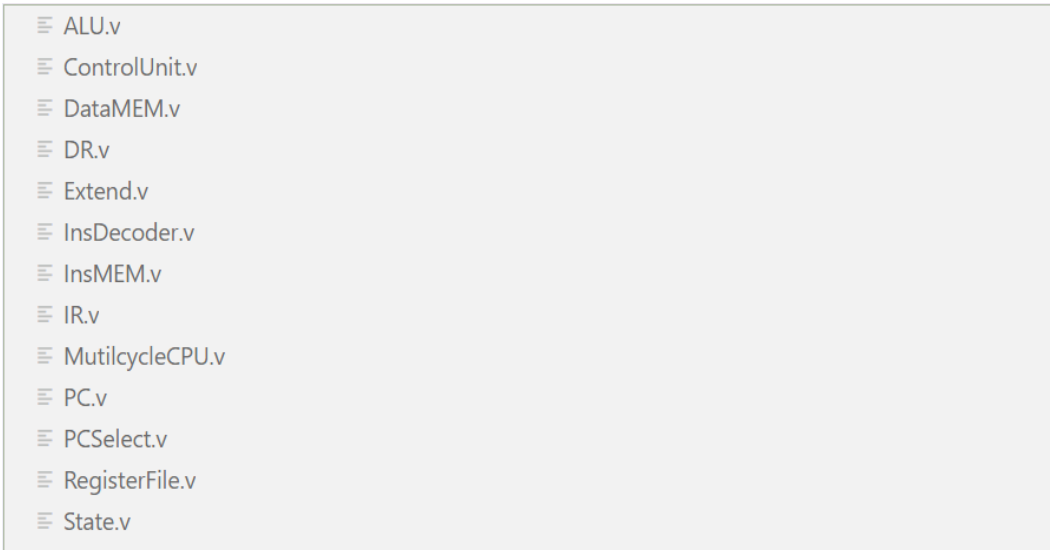
表3（续表）

Instruction	Op	mRD	mWR	IRWre	ExtSel	PCSrc	RegDst
add	000000					00	10
sub	000001					00	10
addi	000010				1	00	01
or	010000					00	10
and	010001					00	10
ori	010010				0	00	01
sll	011000					00	10
slt	100110					00	10
sltiu	100111				0	00	01
sw	110000		1		1	00	
lw	110001	1			1	00	01
beq(zero=0)	110100				1	00	
beq(zero=1)	110100				1	01	
bltz(zero=0 或 sign=1)	110110				1	01	
bltz(zero=1 或 sign=0)	110110				1	00	
j	111000					11	
jr	111001					10	
jal	111010					11	00

halt	111111						

需要注意的是，控制信号PCWre、mRD、mWR、IRWre还与状态State有关，这一部分的判断在ControlUnit模块实现。

（3）在本次实验中，根据数据通路图将CPU分为了总计13个模块，分别为ALU（算术逻辑运算单元）、ControlUnit（控制单元）、DataMEM（数据存储器）、DR（切分数据通路）、Extend（立即数扩展）、InsDecoder（指令译码器）、InsMEM（指令存储器）、IR（指令寄存器，使指令代码保持稳定）、PC（程序计数器）、PCSelect（计算下一个PC）、RegisterFile（寄存器组）、State（判断CPU执行阶段），以及顶层模块MutilcycleCPU。



下面就各个模块的代码实现加以阐述：

① ALU（算术逻辑运算单元）：

ALU总计八种操作，详见表2ALU运算功能表。操作数为RegA和RegB，Sign为结果的符号位，当ALUResult为0时Zero为1。

```
always @(ALUOpcode or RegA or RegB) begin
    case(ALUOpcode)
        3'b000 : Result = RegA + RegB;
        3'b001 : Result = RegA - RegB;
        3'b010 : Result = (RegA < RegB) ? 1 : 0;
        3'b011 : Result = (((RegA < RegB) && (RegA[31] == RegB[31]))
                           || ((RegA[31] == 1 && RegB[31] == 0))) ? 1 :
0;
        3'b100 : Result = Regb << RegA;
        3'b101 : Result = RegA | RegB;
        3'b110 : Result = RegA & RegB;
```

```
        3'b111 : Result = RegA ^ RegB;
        default : Result = 32'h00000000;
    endcase
end
```

② ControlUnit（控制单元）：

该模块的主要功能是根据操作码得到相应的各个控制信号的值，详情见表3控制信号真值表，需要注意的是在多周期CPU中，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。

```
assign PCWre = (Op == 6'b111111 || State != 3'b000) ? 0 : 1;
assign ALUSrcA = (Op == 6'b011000) ? 1 : 0;
assign ALUSrcB = (Op == 6'b000010 || Op == 6'b010010
               || Op == 6'b100111 || Op == 6'b110000
               || Op == 6'b110001) ? 1 : 0;
assign DBDataSrc =  Op == 6'b110001 ? 1 : 0;
assign RegWre = (Op == 6'b111111 || Op == 6'b110000
               || Op[5:1] == 5'b11100 || Op[5:2] == 4'b1101
               || (State != 3'b011 && Op != 6'b111010)) ? 0 : 1;
assign WrRegDSrc = Op == 6'b111010 ? 0 : 1;
// 仅当一条指令执行完毕后
assign InsMemRw = State == 3'b000 ? 1 : 0;
// 与状态有关
assign mRD = (Op == 6'b110001 && State == 3'b100) ? 1 : 0;
assign mWR = (Op == 6'b110000 && State == 3'b100) ? 1 : 0;
assign IRWre = State == 3'b000 ? 1 : 0;
assign ExtSel = (Op == 6'b010010 || Op == 6'b100111) ? 0 : 1;
```

下面代码仅展示一条指令作为说明（使用case语句方法逐个产生部分指令控制信号）：

```
always @(Op or Zero or Sign) begin
    case(Op)
     6'b000000: begin // add
        PCSrc = 2'b00;
        RegDst = 2'b10;
        ALUOp = 3'b000;
```

③ DataMEM（数据存储器）：

该模块用于sw和lw这两个指令。定义ram用于储存数据，一个单位有8个寄存器。输入address为在ram进行操作的地址，nRD==1时为读，将ram中的数据写入Dataout中。nWR==1时为写，将writeData中的数据写入ram，时钟下降沿触发。

```
assign DataOut[7:0] = (mRD == 1) ? ram[Address + 3] : 8'bz;
assign DataOut[15:8] = (mRD == 1) ? ram[Address + 2] : 8'bz;
```

```
assign DataOut[23:16] = (mRD == 1) ? ram[Address + 1] : 8'bz;
assign DataOut[31:24] = (mRD == 1) ? ram[Address ] : 8'bz;

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

④ DR（切分数据通路）：

DR的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。其中，ADR、BDR、ALUOutDR、DBDR 四个寄存器的具体功能在顶层模块中实现，这四个寄存器不需要写使能信号。

```
module DR(
    input CLK,
    input [31:0] DRIn,
    output reg [31:0] DROut
    );
    always @(negedge CLK) begin
        DROut <= DRIn;
    end
endmodule
```

ADR、BDR、ALUOutDR、DBDR 四个寄存器的具体实现（以ADR为例）：

```
DR ADR(
    .CLK(CLK),
    .DRIn(ReadData1),
    .DROut(ADROut)
);
```

⑤ Extend（立即数扩展）

该模块为符号扩展或零扩展，根据输入的控制信号ExtSel判断要进行扩展类型；

```
module Extend(
    input ExtSel,
    input [15:0] ExtendIn,
    output reg [31:0] ExtendOut
    );
    always @(ExtSel or ExtendIn) begin
    ExtendOut[15:0] = ExtendIn;
        if (ExtSel == 0 || ExtendIn[15] == 0)
            ExtendOut[31:16] = 16'h0000;
```

```
        else if (ExtSel == 1 && ExtendIn[15] != 0)
            ExtendOut[31:16] = 16'hffff;
        else
            ExtendOut = 32'hzzzz;
    end
endmodule
```

⑥ InsDecoder（指令译码器）：

在这一模块中，我们对指令进行译码，将指令根据其组成部分进行划分：

```
assign op = Instruction[31:26];
assign rs = Instruction[25:21];
assign rt = Instruction[20:16];
assign rd = Instruction[15:11];
assign sa = Instruction[10:6];
assign immediate = Instruction[15:0];
assign addr = Instruction[25:0];
```

⑦ InsMEM（指令存储器）：

这一模块将保存在.txt文件中的指令数据读入存储器中，当读使能信号有效时，则将存储器中的指令数据输出：

```
module InsMEM(
    input InsMemRW,
    //Iaddr，指令地址输入端口
    input [31:0] IAddr,
    output reg [31:0] DataOut
    );
    //存储器定义，存储单元长度为8，总计100个存储单元
    reg [7:0] InsMem [99:0];

    initial begin
        $readmemb("C:/Users/cheny/Desktop/ECOP/testData.txt", InsMem);
    end

    always @(InsMemRW or IAddr) begin
        if (InsMemRW == 1) begin
        // ？IAddr理解
            DataOut[31:24] = InsMem[IAddr];
            DataOut[23:16] = InsMem[IAddr+1];
            DataOut[15:8] = InsMem[IAddr+2];
            DataOut[7:0] = InsMem[IAddr+3];
        end
    end
endmodule
```

⑧ IR（指令寄存器，使指令代码保持稳定）：

我们增加模块IR，目的是使指令代码保持稳定：

```
module IR(
    input CLK,
    input IRWre,
    input [31:0] IRDataIn,
    output reg [31:0] IRDataOut
    );
    always @(posedge CLK) begin
        if (IRWre)
            IRDataOut <= IRDataIn;
    end
endmodule
```

⑨ PC（程序计数器）：

PC模块的输入有CLK（时钟）、Reset（初始化）、PCWre（PC变化的使能）、PCIn（计算得到的下一时刻的PC），该模块为下降沿触发，当Reset==0时初始化当前PC为0，否则PC=PCIn。

```
always @(negedge CLK or negedge RST) begin
     if (RST == 0) begin
         pc = 32'h00000000;
         resetPC = 1;
     end
     else if (PCWre) begin
         if (resetPC == 1) begin
             pc = 32'h00000000;
             resetPC = 0;
         end else begin
             pc = PCIn;
         end
     end
end
```

⑩ PCSelect（计算下一个PC）：

该模块根据输入的控制信号 PCSrc 的值计算下一条指令的地址：

一般情况下，大多数指令没有跳转，直接执行下一条指令，由于每一条指令占用4个指令寄存器，下一条指令的地址就等于当前指令的地址加4；当指令为beq、bltz时，要跳转到的指令地址是4的倍数，最低两位是“00”，因此在将扩展的立即数放进指令码中的时候，要右移2位；当指令为j时，除了j指令的6位操作码外，addr就只有26位，因为PC一定能被4整除，后两位就默认为0，然后前四位则为下一指令的前四位，这样就导致了指令的跳转有一定的范围。

```
always @(PCSrc or PC4 or ExtendImme or Addr or RegData) begin
    case(PCSrc)
```

```
        2'b00 : PCOut = PC4;
        2'b01 : PCOut = PC4 + (ExtendImme << 2);
        2'b10 : PCHandled = RegData;
        2'b11 : begin
            PCOut[31:28] = PC4[31:28];
            PCOut[27:2] = Addr[25:0];
            PCOut[1:0] = 2'b00;
        end
    endcase
end
```

⑪ RegisterFile（寄存器组）：

在这一模块中，两个读数的寄存器ReadReg1和ReadReg2在时钟周期到来时将rs和rt寄存器中的数读出；WriteReg在时钟周期经过ALU等一系列操作后将得到的数据写回寄存器。

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

⑫ State（判断CPU执行阶段）：

在多周期CPU中，CPU的下个状态取决于指令操作码和当前状态，我们根据多周期状态转移图，对CPU的状态进行设定：

```
always @(posedge CLK) begin
    if (RST == 0)
        State = 3'b0;
    else begin
        case(State)
            3'b000 : State = 3'b001;
            3'b001 : begin
                if (Op[5:3] == 3'b111)
                    State = 3'b000;
                else
                    State = 3'b010;
            end
            3'b010 : begin
                if (Op[5:2] == 4'b1101)
                    State = 3'b000;
```

```
                else if (Op[5:2] == 4'b1100)
                    State = 3'b100;
                else
                    State = 3'b011;
            end
            3'b011 : State = 3'b000;
            3'b100 : begin
                if (Op[2:0] == 3'b000)
                    State = 3'b000;
                else
                    State = 3'b011;
            end
        endcase
    end
end
```

⑬ MulticycleCPU（顶层模块）：

顶层模块起到了一个连接的功能，对于顶层模块，各里面的内容被封装起来，我们所需要做的就是将相应的接口（及输入和输出）对应起来。在本次实验中，我将数据通路图中选择器的部分在顶层模块中实现。需要注意的是在各个模块之间的输入输出的变量需要为wire，不能为reg。

选择器功能实现：

```
// ALU
assign RegA = ALUSrcA == 0 ? ADROut : sa;
assign RegB = ALUSrcB == 0 ? BDROut : ExtendOut;
// RegisterFile
assign WriteReg = (RegDst[1] == 0) ? ((RegDst[0] == 0) ? 5'b11111 : rt) :
((RegDst[0] == 0) ? rd : 0);
assign WriteData = WrRegDSrc == 0 ? PC4 : DBDROut;
// RAM
assign DBDRIn = DBDataSrc == 0 ? ALUResult : DataMEMOut;
```

2.验证CPU的正确性：

（1）测试程序段：

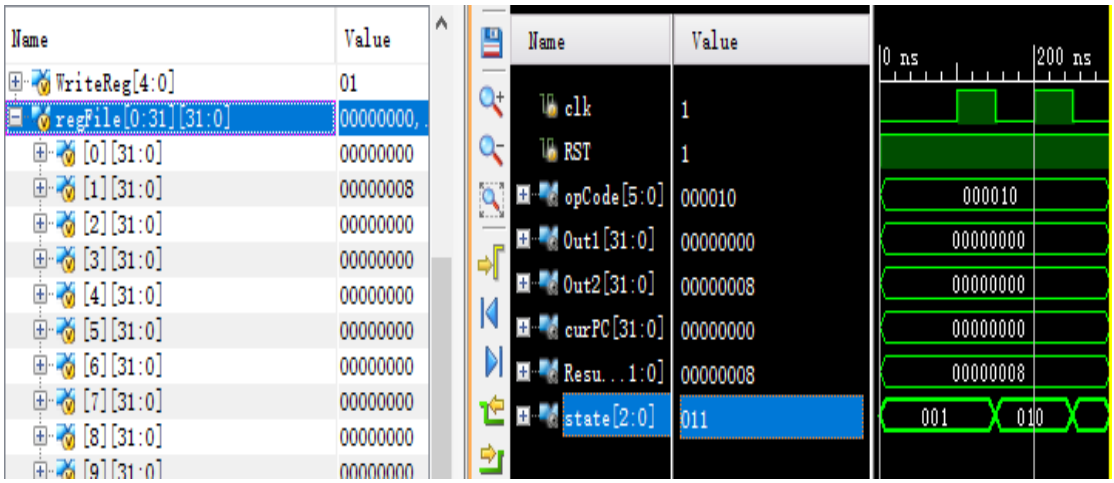
地址	汇编程序	指令代码					
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addi $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	or $3,$2,$1	010000	00010	00001	0001 1000 0000 0000	=	40411800
0x0000000C	sub $4,$3,$1	000001	00011	00001	0010 0000 0000 0000	=	04612000
0x00000010	and $5,$4,$2	010001	00100	00010	0010 1000 0000 0000	=	44822800

0x00000014	sll $5, $5, 2	011000	00000	00101	0010 1000 1000 0000	=	60A50000
0x00000018	beq $5, $1, -2(=, 转14)	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000	=	E8000010
0x00000020	slt $8, $12, $1	100110	01100	00001	0100 0000 0000 0000	=	99814000
0x00000024	addi $13, $0, -2	000010	00000	01101	1111 1111 1111 1110	=	080DFFFE
0x00000028	slt $9, $8, $13	100110	01000	01101	0100 1000 0000 0000	=	990D4800
0x0000002C	sltiu $10, $9, 2	100111	01001	01010	0000 0000 0000 0010	=	9D2A0002
0x00000030	sltiu $11, $10, 0	100111	01010	01011	0000 0000 0000 0000	=	9D4B0000
0x00000034	addi $13, $13, 1	000010	01101	01101	0000 0000 0000 0001	=	09AD0001
0x00000038	bltz $13, -2 (<0, 转34)	110110	01101	00000	1111 1111 1111 1110	=	D9A0FFFE
0x0000003C	j 0x000004C	111000	00000	00000	0000 0000 0001 0011	=	E0000013
0x00000040	sw $2, 4($1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004
0x00000044	lw $12, 4($1)	110001	00001	01100	0000 0000 0000 0100	=	C42C0004
0x00000048	jr $31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x0000004C	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

（2）指令检查：

①在CPU工作之前，初始化PC和State均为0，执行第一条指令：

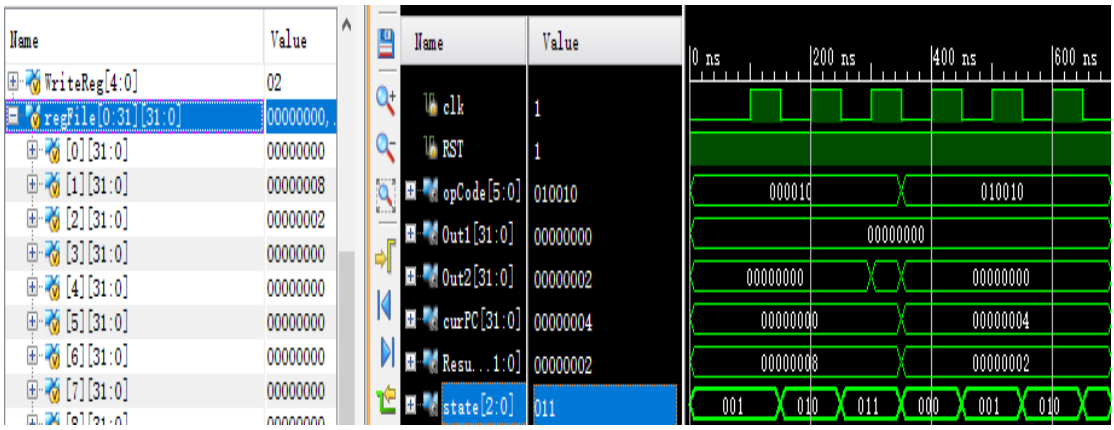
0x00000000	addi $1, $0, 8



此时，写入寄存器$1的值为00000008H，指令执行正确；

②执行第二条指令：

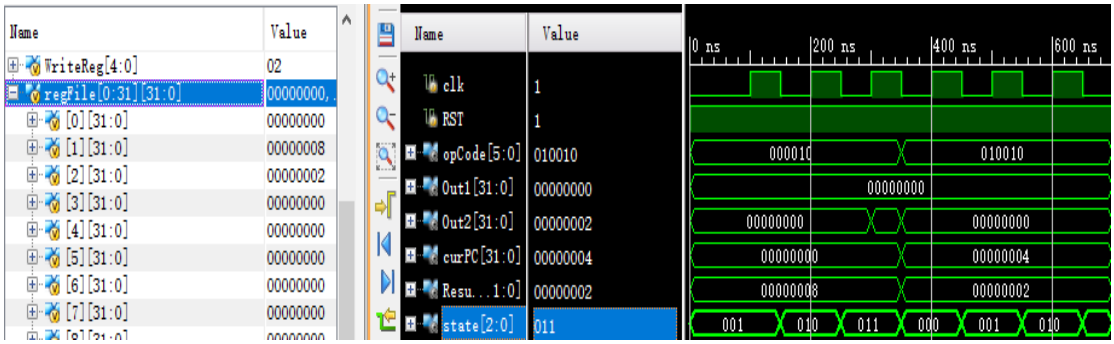
0x00000004	ori $2, $0, 2



此时，$0 = 00000000H，与扩展后的立即数00000002H进行或运算，结果00000002H存储在$2中，指令执行正确；

③执行第三条指令：

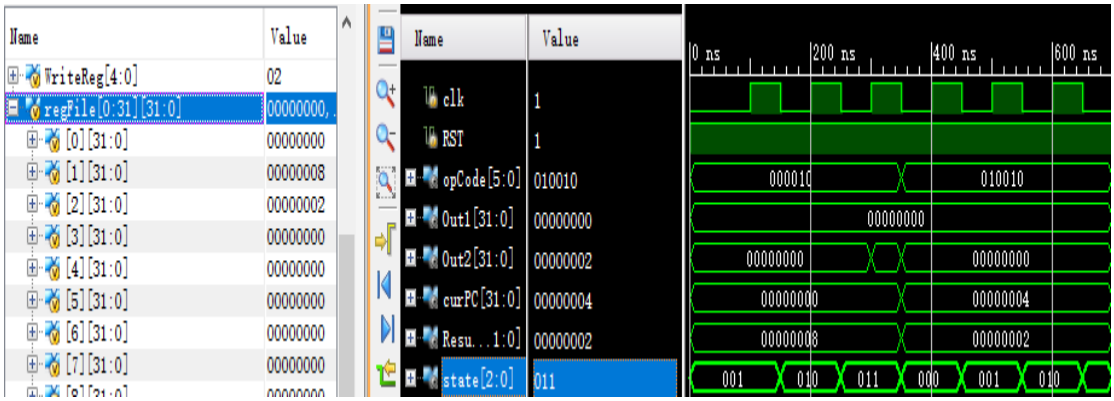
0x00000008	or $3, $2, $1



此时，$2 = 00000002H，$1 = 00000008H，两个寄存器中的数值进行或运算后结果为0000000AH存储在$3中，指令执行正确。

④执行第四条指令：

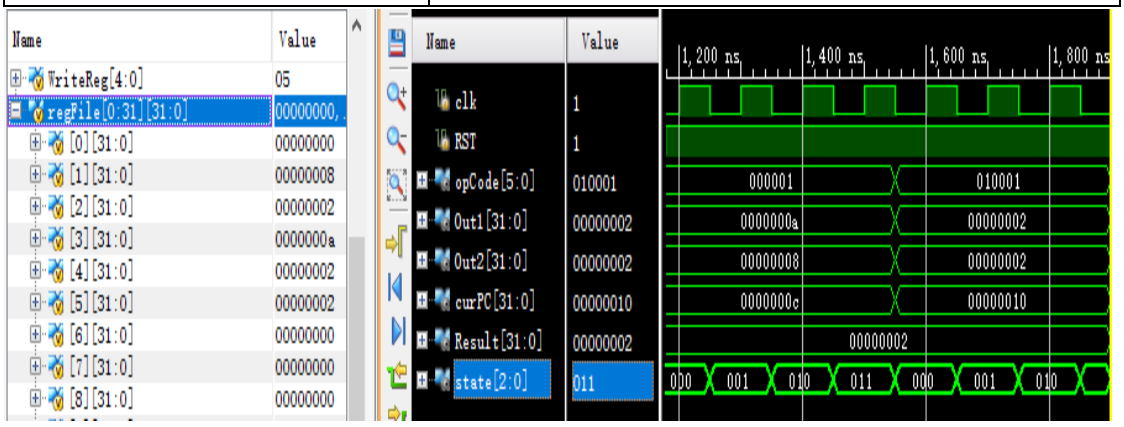
0x0000000C	sub $4, $3, $1



此时，$3 = 0000000AH，$1 = 00000008H，两个寄存器中的数值进行相减运算后结果为00000002H存储在$4中，指令执行正确。

⑤执行第五条指令：

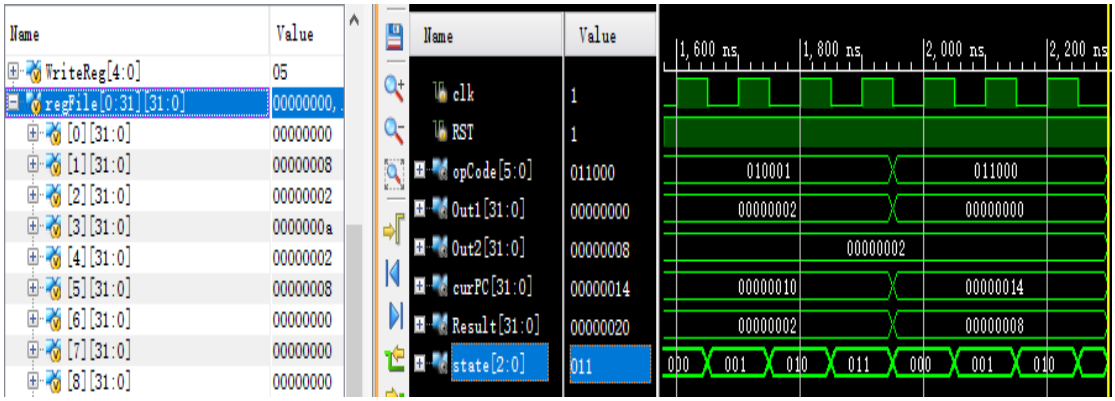
0x00000010	and $5, $4, $2



此时，$4 = 00000002H，$2 = 00000002H，两个寄存器中的数值进行与运算后结果为00000002H存储在$5中，指令执行正确。

⑥执行第六条指令：

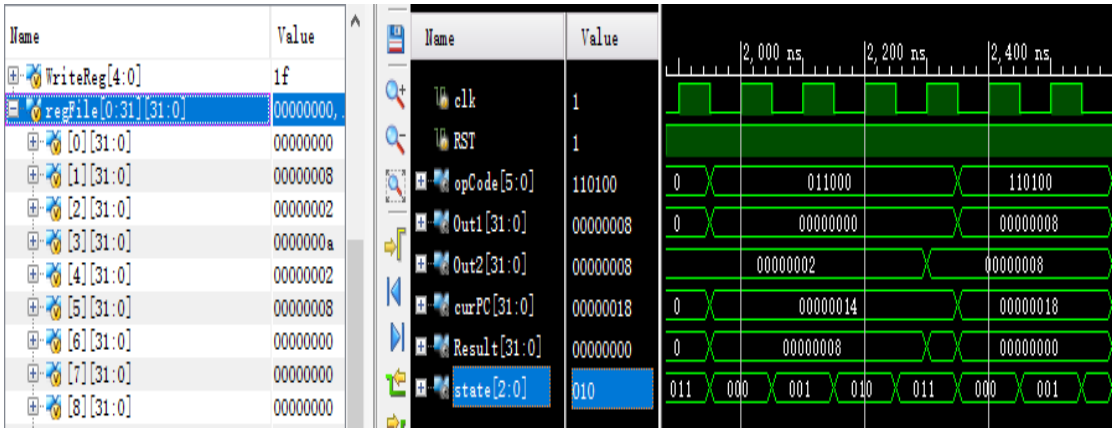
0x00000014	sll $5, $5, 2



此时，$5 = 00000002H，左移2位后结果为00000008H存储在$5中，指令执行正确。

⑦执行第七条指令：

0x00000018	beq $5, $1, -2(=, 转 14)

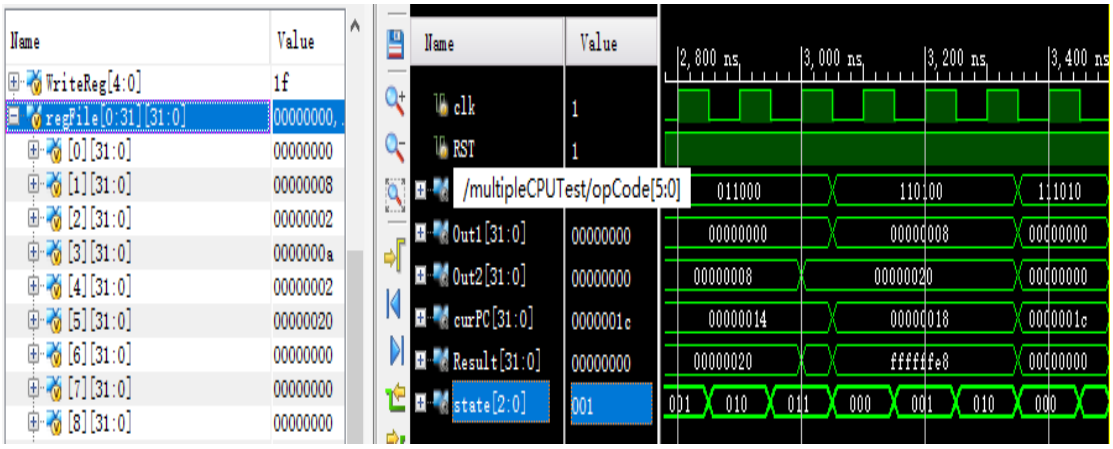


此时，$1 = 00000008H，$5 = 00000008H，两个寄存器中的数值进行比较后结果

相等，指令跳转，指令执行正确。

⑧执行第八条指令：

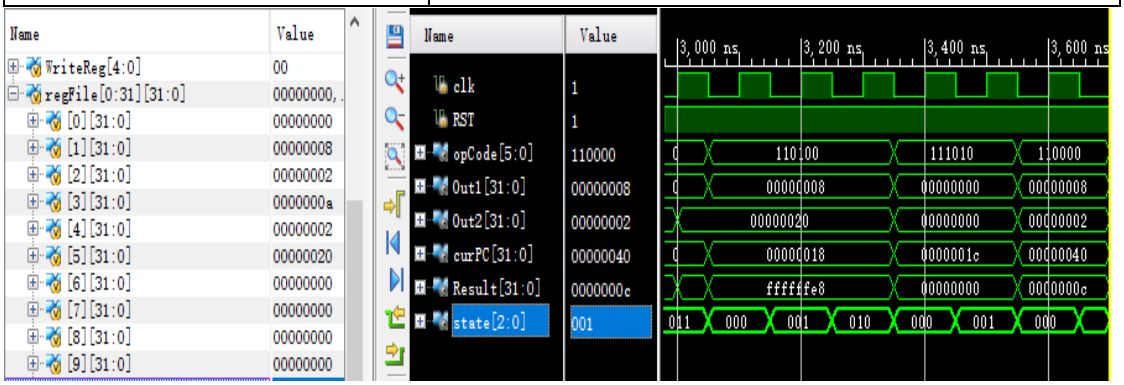
0x0000001C	jal 0x0000040



此时，指令会跳转到地址为0x0000040的指令，此时$31 = 00000020H。

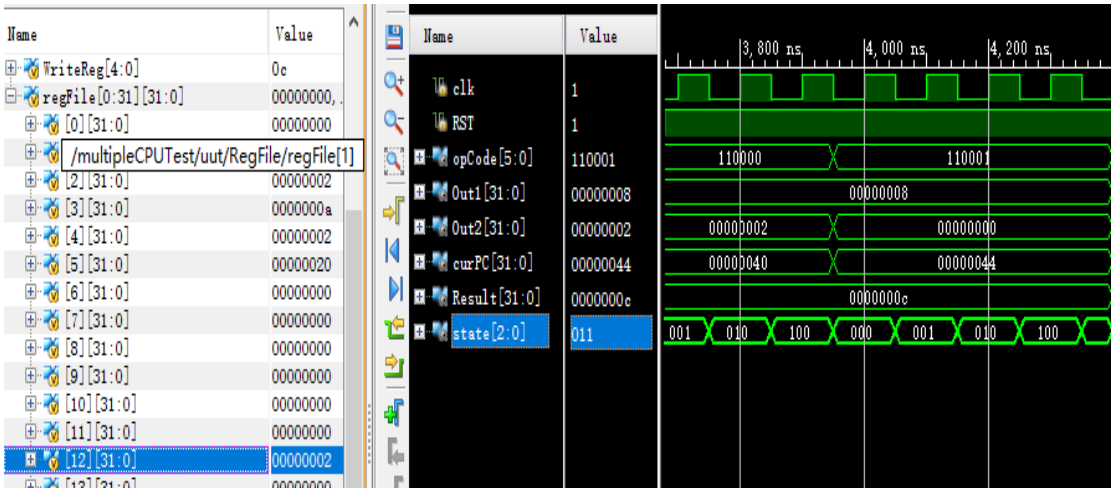
⑨

0x00000040	sw $2, 4($1)



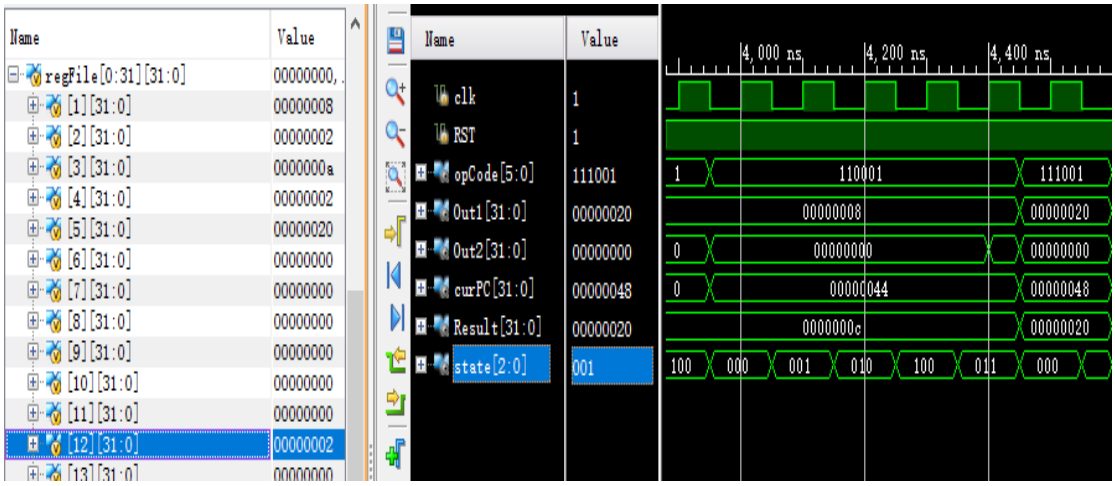
⑩

0x00000044	lw $12, 4($1)



⑪

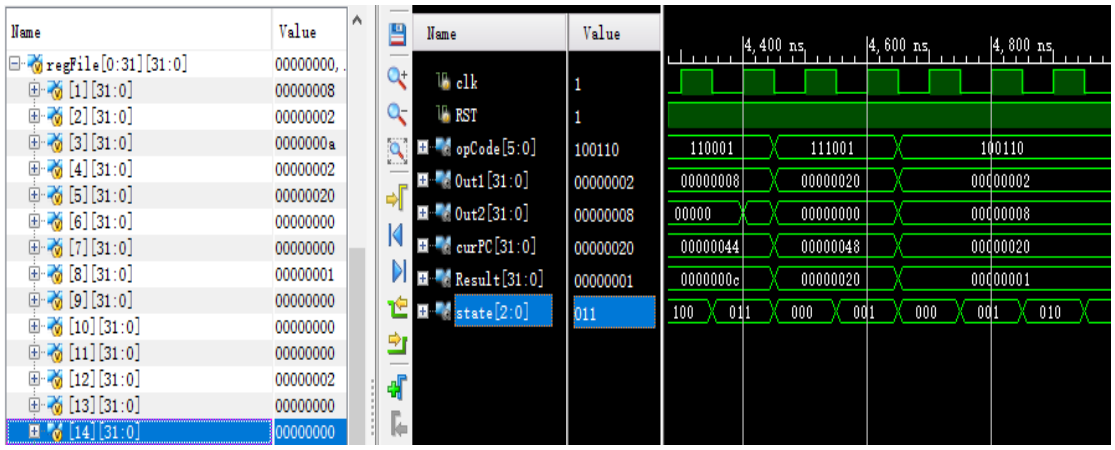
0x00000048	jr $31



此时，指令会跳转到$31所存地址。

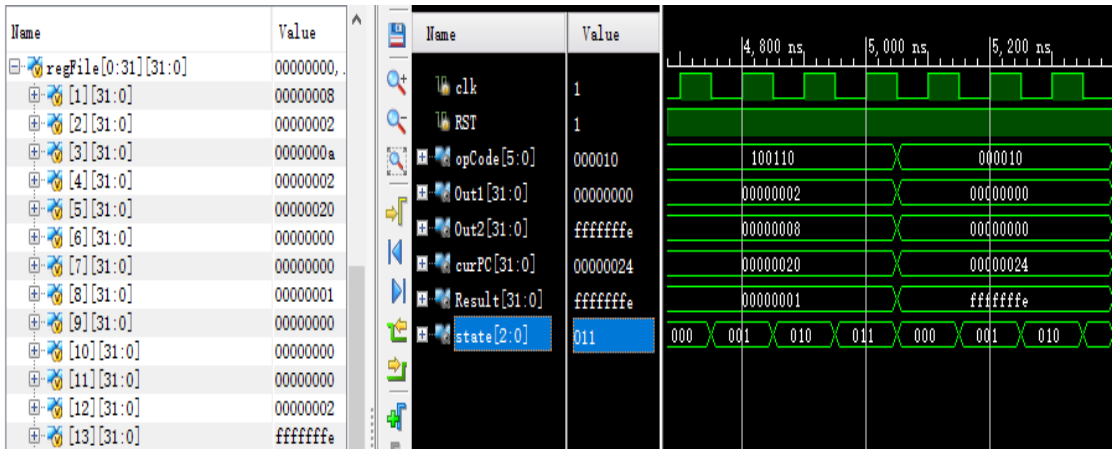
⑫

0x00000028	slt $9, $8, $13



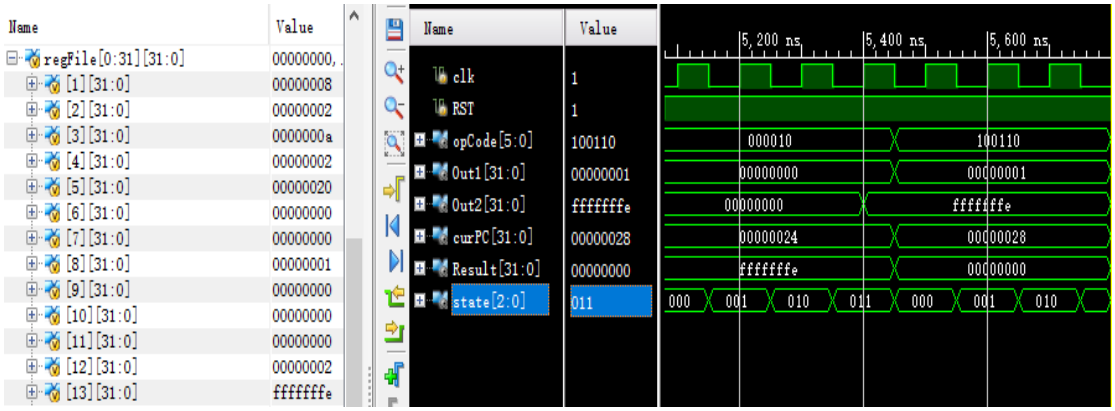
⑬

0x00000024	addi $13, $0, -2



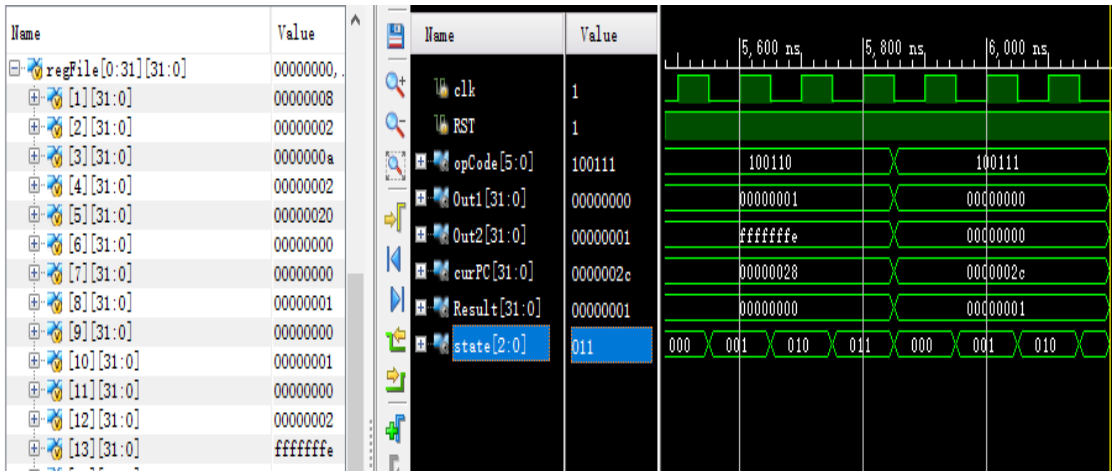
⑭

0x00000028	slt $9, $8, $13



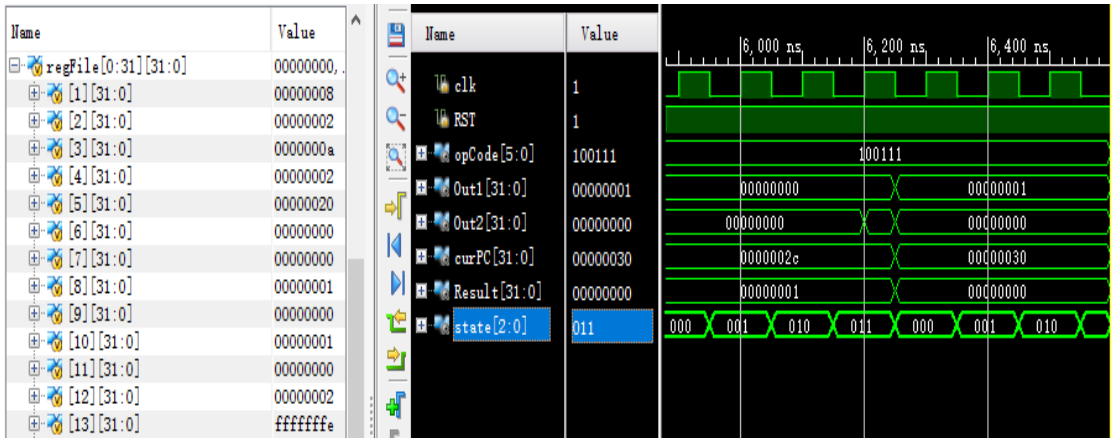
⑮

0x0000002C	sltiu $10, $9, 2



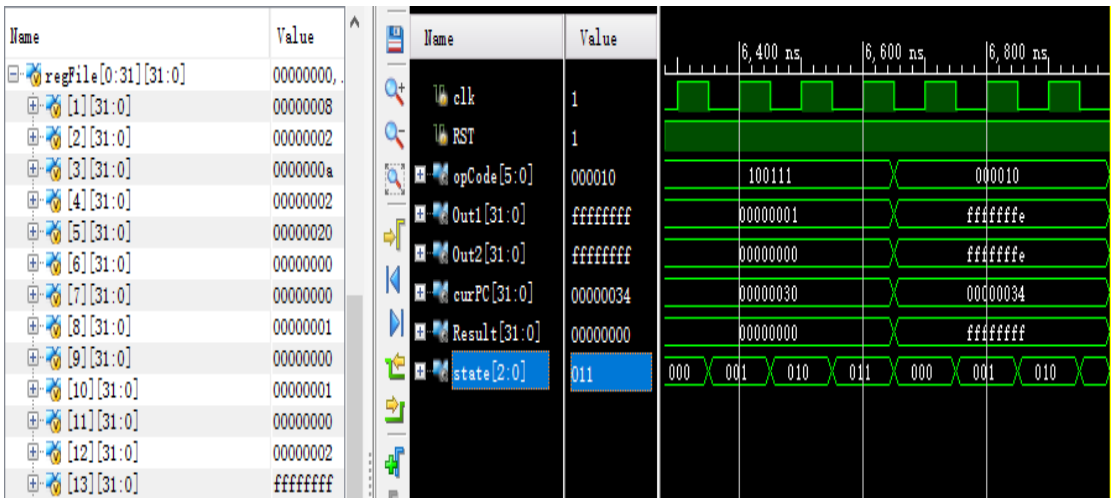
⑯

0x00000030	sltiu $11, $10, 0



⑰

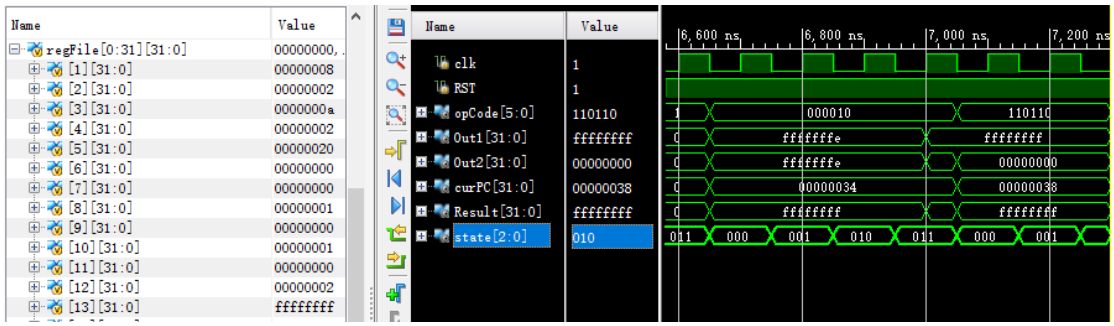
0x00000034	addi $13, $13, 1



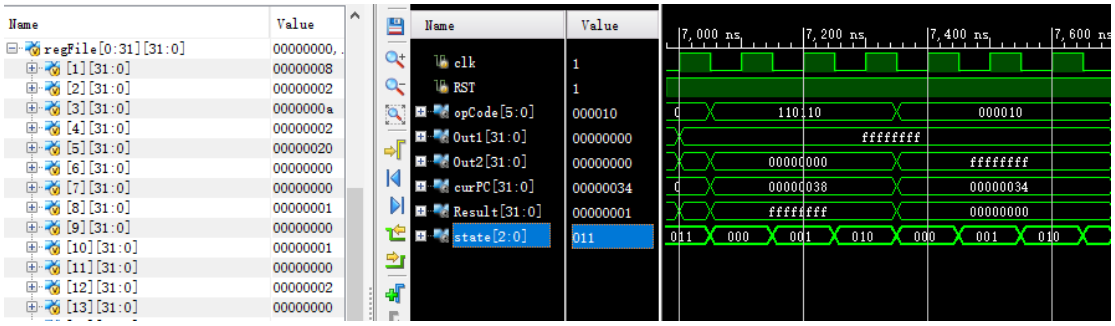
⑱

0x00000038	bltz $13,-2 (<0, 转 34)

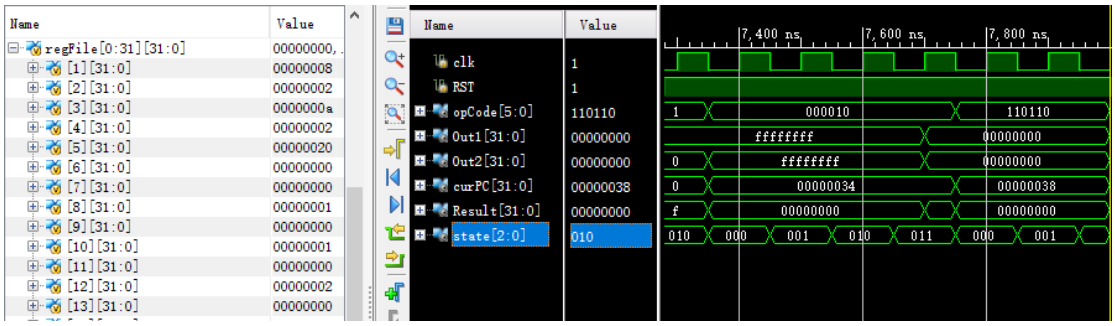
$13中内容小于0，所以跳转到地址**0x00000034** ：



跳转后执行addi指令，将$13中内容加1：

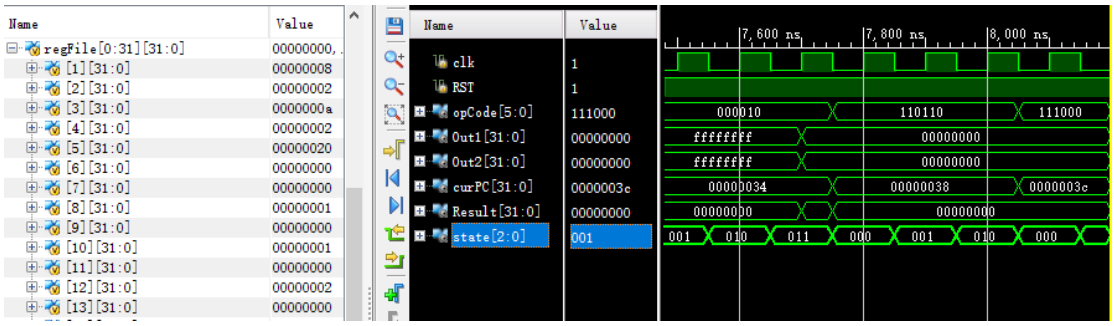


执行bltz，$13中内容为0，正常跳转：



⑲

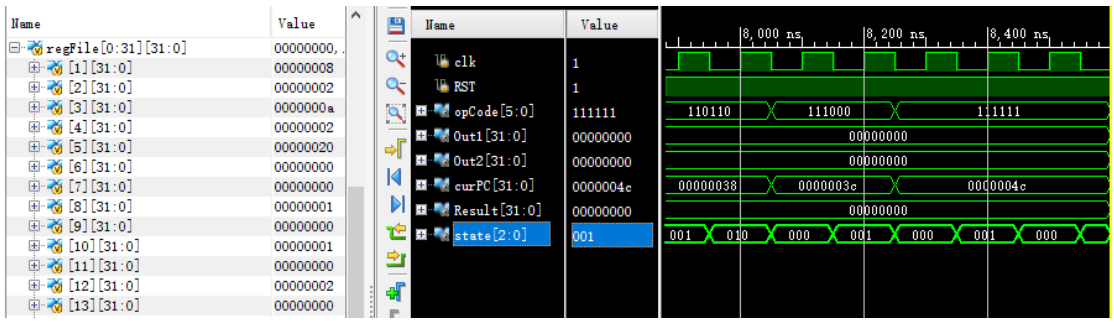
0x0000003C	j 0x000004C



跳转到地址**0x000004C**，执行停机指令。

⑳

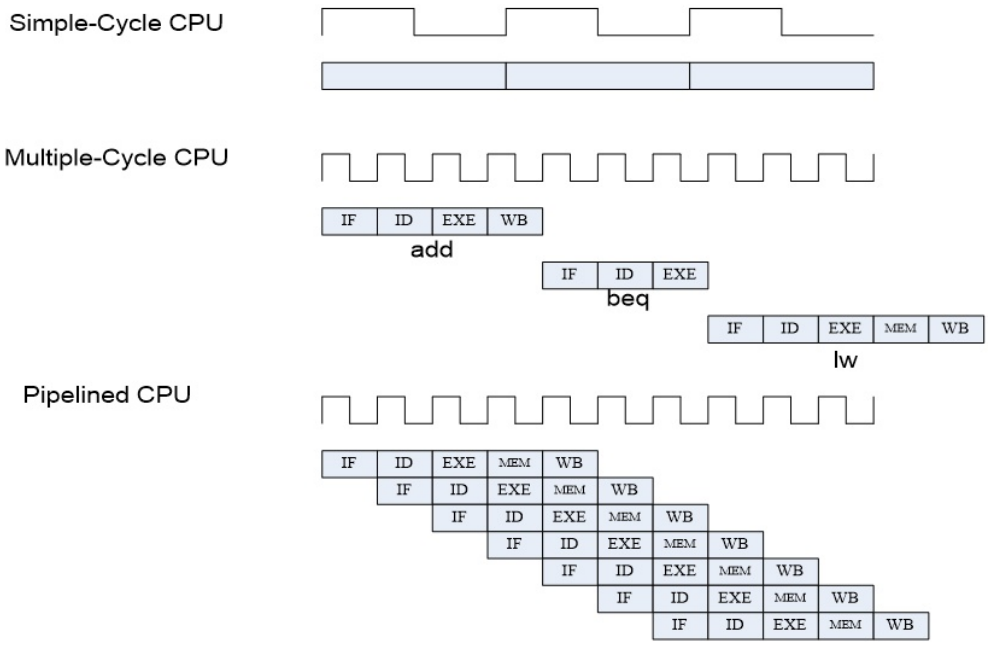
0x0000004C	halt



此时，PC不改变，不会再跳转到其他指令。

六. 实验心得

1、弄清楚了单周期CPU，多周期CPU，流水线CPU的辨析：多周期CPU与单周期CPU的区别在于，多周期CPU在一个时间周期内只执行一条指令中的一个阶段，而单周期CPU是在一个时间周期内执行一条完整的指令。由于并不是所有的指令都需要执行最多的5个步骤，所以多周期CPU的执行时间会比单周期CPU的略快，思考的重点就是如何实现数据内容的缓存，如何在时钟周期更替的时候使得控制信号量和相关寄存器里面的值符合预期的要求；多周期CPU与流水线CPU的区别在于，多周期CPU指令的执行也还是顺序执行的，而流水线CPU的执行时属于并行执行的，明白了这一点后，就可以明白在考虑多周期CPU运行过程的时候，集中点都是在一条特定的指令上，不用考虑冒险。下面这个图很好的解释三者的区别：



2、总体了解了多周期CPU的实现流程：设计实验时，多周期CPU 需要将CPU 划分成IF、ID、EXE、MEM、WB 这五个部分，每个部分由一个周期完成，每个部分都有来自于control unit 的控制信号控制。IF 通过PCWre控制，只有当PCWre==1 时，才会跳转到下一个PC，然后从指令存储器中取出新的指令；ID通过IRWre 控制，只有当IRWre==1 时，IR 寄存器中才会存入新的指令，然后从寄存器组中读入相应的数据；EXE 没有相应的控制数据，每一个时钟都会进行计算，但是也通过ADR、BDR、ALUoutDR 保持了ALU 输入和输出的相对稳定；MEM 通过RD 和WR 控制，当RD==1 时为读内存，WR==1 时为写内存，若二者都为0 则MEM 不执行；WB 通过RegWre 控制，只有RegWre==1时才会有寄存器的写回，即WB。

3、在触发方面，我一开始都采用了时钟上升沿触发，但在仿真之后发现在ControlUnit模块State变化之后的一个时钟其他部分才会变化，产生了延迟，就导致数据紊乱，所以在ControlUnit部分中的一些各个阶段的控制信号我采用了时钟下降沿触发，其他模块我采用了时钟上升沿触发或者电平触发，但是写回我还是使用了时钟下降沿触发，解决了这个问题。

4、课程小结：

在本学期的实验课上，我学会了很多，第一件事就是要实践出真知。我们在理论课所学的很多知识仅仅只是浮于表面，不能算是真正的掌握，但是在实验课上，自己亲手完成了单周期和多周期CPU之后，感觉自己对于CPU的内部构造以及如何工作就明晰了很多，PC、InsMEM等也不仅仅只是一个名词，在做完实验之后可以很清楚的了解其功能、构造、如何工作。实验课让我对于知识的掌握更加牢固。正如在做单周期CPU时，即使在理论课上

学习过了，但是依然觉得困难重重，但是在做完单周期CPU 之后再做多周期CPU，就可以很容易找到单周期和多周期的区别和共同点来构造多周期CPU。

其次就是需要细心，尤其是在做实验的时候，在实验过程中，我经常容易将变量弄混，或者将测试代码写错，在仿真时导致结果不正确，需要很长时间来debug。

细心之外，耐心也很重要，要相信bug总会一个一个地修改完，在debug时，思路要清晰，在仿真图中查找寄存器组中的寄存器数值是否正确，若不正确查找从哪一步开始发生错误，从源头找到错误，一条条指令分析下去。

在本学期计算机组成原理实验中，感觉自己获益匪浅。虽然花费了很多时间完成实验，但是每一次实验之后感觉对于知识的理解又有一种质的突破，在设计实验时我们不仅仅是被动地学习，更是对于各个器件、状态主动地思考，在思考中所获得的知识是很难在课本上得到的，更不会有思考和debug 这些过程的体会。