《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 软件工程二（4）班
学 生 姓 名 ：	刘俊峰
学　　号 ：	16340150
时　　间 ：	2018　年　6　月　28　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：

(1) 取指令(IF)：根据程序计数器 pc 中的指令地址，从存储器中取出一条指令，同时，pc 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 pc，当然得到的“地址”需要做些变换才送入 pc。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

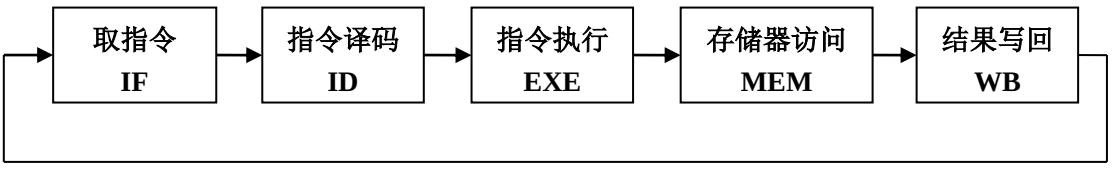
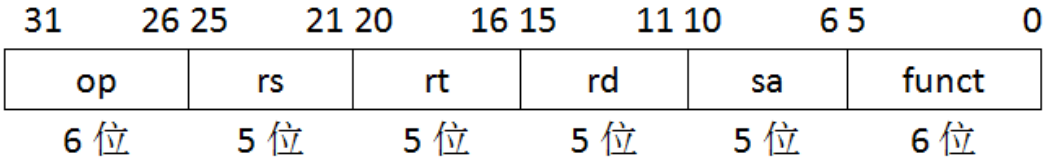


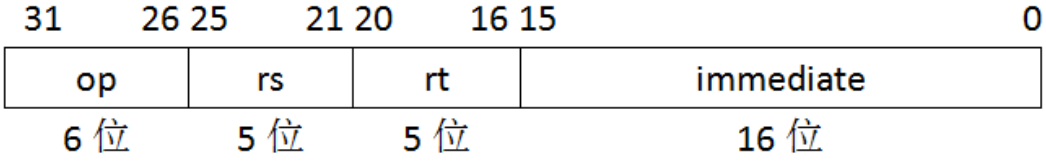
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式:

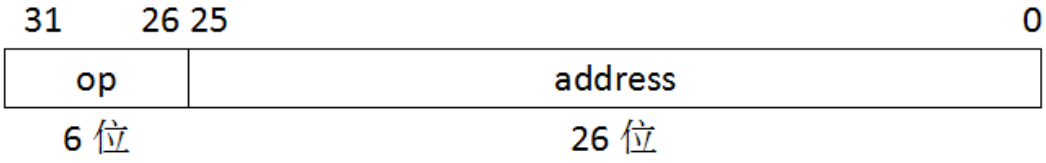
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

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

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

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

address：为地址。

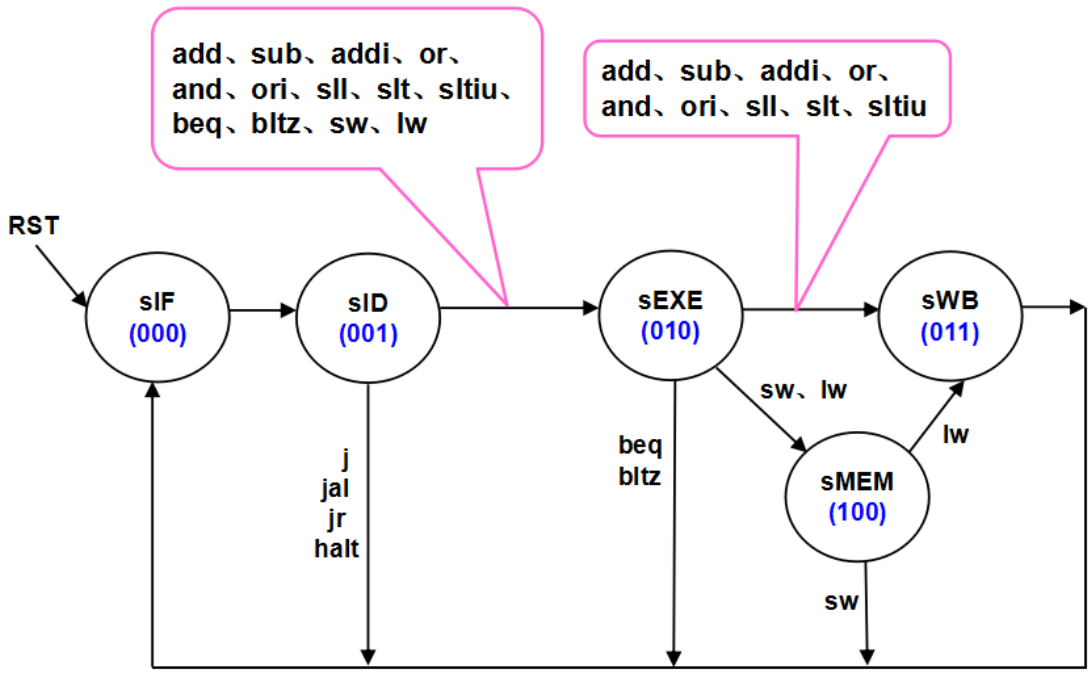


图 2 多周期 CPU 状态转移图

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

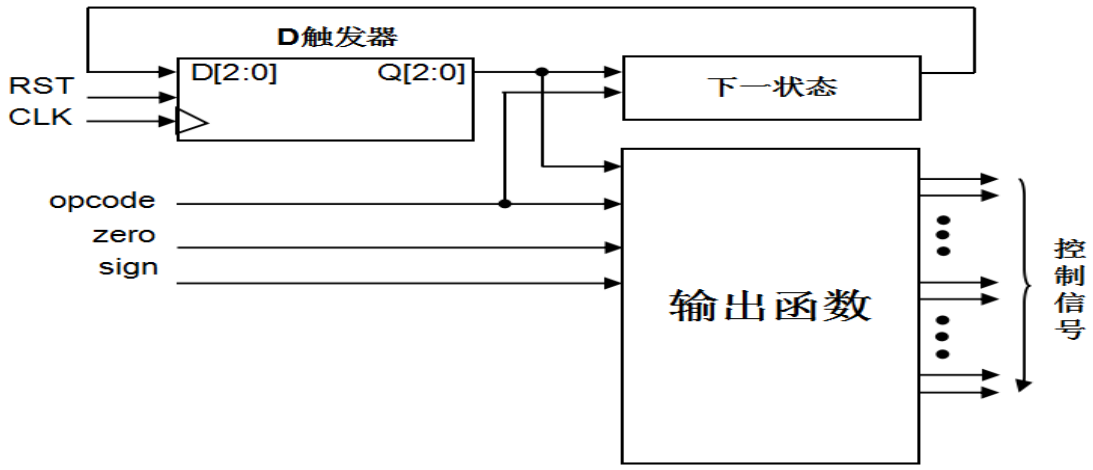


图 3 多周期 CPU 控制部件的原理结构图

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

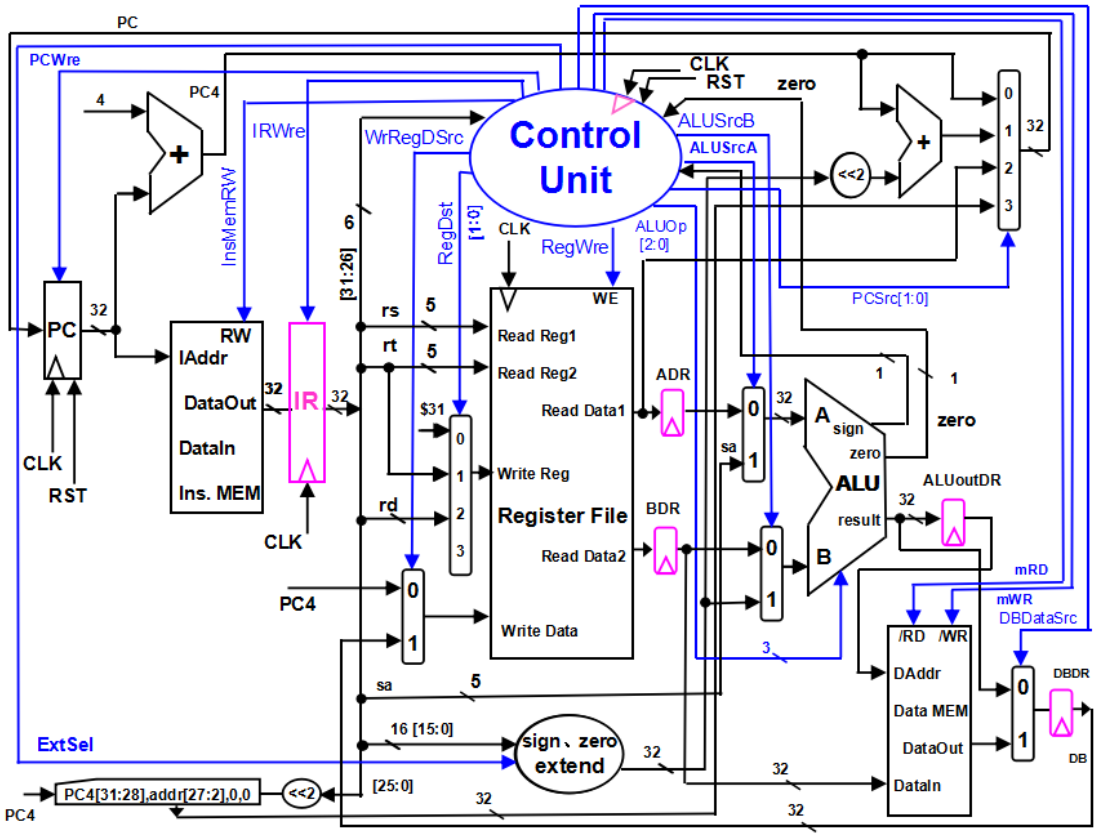


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号，**其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1　控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、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	来自 sign 或 zero 扩展的立即数，相关

	令：add、sub、or、and、beq、bltz、slt、sll	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

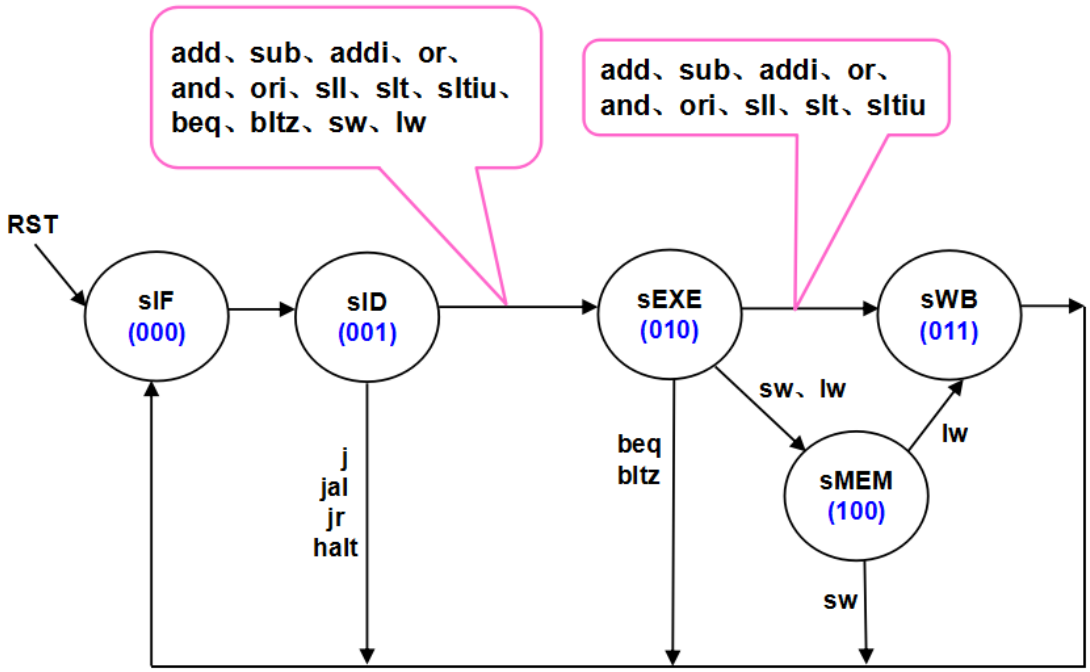
电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五．实验过程与结果

1．设计思想

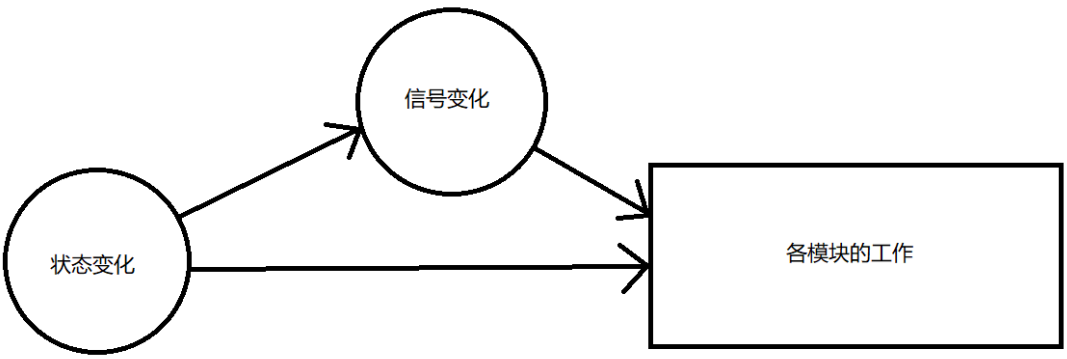
多周期的模块分布跟单周期基本相似。不同的地方在于，多周期cpu将指令的执行详细分成了IE, ID, EXE, MEM, WB五个状态，对于不同的指令会有不同的状态变化。对于简单的指令可以通过更少的状态来降低指令执行的时间；对于不需要进

行内存访问或者寄存器写入的指令，可以跳过这个步骤。不同的指令对应的状态大概是这样：



从图上可以看到，5个不同的状态，需要3位来表示。用到所有状态的指令只有lw。也就是说，其他的指令只需要四个或更少的时钟周期就可以执行完，最多也只需要五个时钟周期。

因此，多周期cpu的控制信号设计应该是面向状态的。也就是说，随着时钟周期变化的是状态，在状态发生改变的时候，控制信号才根据不同的状态产生不同的变化。而根据这些信号和状态，其他模块再判断是否应该工作，或者说执行什么样的动作。用图来表示就是：



了解了这些设计思想之后，就是开始设计的时候了。

2. cpu各模块的设计

a．控制模块的分析和设计

控制模块无论在单周期还是在多周期cpu中都是最为重要的单元。它掌控了所有的信号。根据上图叙述的，我们的控制单元应该可以分为两个部分：状态控制和信号控制。由于状态控制是可以作用于信号控制的。因此，我们应该让状态信号随着时钟信号的到来自动更新，而控制信号由状态改变而触发改变。先讲讲状态信号的改变。

```
always @(posedge clk) begin
    state <= next_state;
end
always @(state or op) begin
  case(state)
    3'b000: next_state = 3'b001;
    3'b001: begin
      if (add || sub || addi || _or || _and || _ori || sll || slt || sltiu || beq
      || bltz || sw || lw) begin
        next_state = 3'b010;
      end
      else begin
        next_state = 3'b000;
      end
    end
    3'b010: begin
      if (beq || bltz) begin
        next_state = 3'b000;
      end
      else if (sw || lw) begin
        next_state = 3'b100;
      end
      else begin
        next_state = 3'b011;
      end
    end
    3'b011: next_state = 3'b000;
    3'b100: begin
      if (lw) begin
        next_state = 3'b011;
      end
      else begin
        next_state = 3'b000;
      end
    end
  endcase
end
```

如上图所示。状态一共由两个变量管理。一个是当前状态，另外一个是下一状态。当时钟上升沿到来时，状态更新位下一状态。而下一状态的更新则是由当

前状态及指令共同决定。

接下来的是控制信号的更新。和单周期不同，这次的逻辑结构是对不同的信号单独进行讨论。（单周期是对不同的指令进行讨论）

```
always @(state) begin
  if (state == 3'b000 && !halt) begin
    PCWre = 1;
      IRWre = 1;
  end
  else
    PCWre = 0;
  InsMemRW = 1;
  if (add || sub || addi || _or || _and || _ori || beq || bltz || slt || sltiu || sw || lw)
    ALUSrcA = 0;
  else
    ALUSrcA = 1;
  if (add || sub || _or || _and || beq || bltz || slt || sll)
    ALUSrcB = 0;
  else
    ALUSrcB = 1;
  if (add || addi || sub || _or || _and || _ori || slt || sltiu || sll)
      DBDataSrc = 0;
  else
    DBDataSrc = 1;
  if (beq || bltz || j || sw || jr || halt)
    RegWre = 0;
  else
    RegWre = 1;
  if (jal)
    WrRegDSrc = 0;
  else
    WrRegDSrc = 1;
  if (lw)
    mRD = 1;
  else
    mRD = 0;
  if (sw)
    mWR = 1;
  else
    mWR = 0;
  if (_ori || sltiu)
    ExlSel = 0;
```

```
else
  mWR = 0;
if (_ori || sltiu)
  ExlSel = 0;
else
  ExlSel = 1;
if (add || addi || sub || _or || _ori || _and || slt || sltiu || sll || sw || lw)
  PCSrc = 2'b00;
else if ((beq && zero == 0) || (bltz && zero == 1) || (bltz && sign == 0))
    PCSrc = 2'b00;
else if ((beq && zero == 1) || (bltz && zero == 0) || (bltz && sign == 1))
    PCSrc = 2'b01;
else if (jr)
    PCSrc = 2'b10;
else if(j || jal)
  PCSrc = 2'b11;
if (jal)
  RegDst = 2'b00;
else if (addi || _ori || sltiu || lw)
  RegDst = 2'b01;
else if (add || sub || _or | _and || slt || sll)
  RegDst = 2'b10;
else
  RegDst = 2'b11;
if (add || addi || lw || sw)
  ALUOp = 3'b000;
else if (sub || beq || bltz)
  ALUOp = 3'b001;
else if (sltiu)
  ALUOp = 3'b010;
else if (slt)
  ALUOp = 3'b011;
else if (sll)
    ALUOp = 3'b100;
else if (_or || _ori)
  ALUOp = 3'b101;
else if (_and)
  ALUOp = 3'b110;
else
  ALUOp = 3'b111;
```

补充一点，上图中能直接用指令的名称当判断条件是因为在控制模块的开头对不同的指令二进制码进行了讨论，具体如下

```
wire add, sub, addi, _or, _and, _ori, sll, slt, sltiu, sw, lw, beq, bltz, j, jr,
jal, halt;
assign add = (op == 6'b000000) ? 1 : 0;
assign sub = (op == 6'b000001) ? 1 : 0;
assign addi = (op == 6'b000010) ? 1 : 0;
assign _or = (op == 6'b010000) ? 1 : 0;
assign _and = (op == 6'b010001) ? 1 : 0;
assign _ori = (op == 6'b010010) ? 1 : 0;
assign sll = (op == 6'b011000) ? 1 : 0;
assign slt = (op == 6'b100110) ? 1 : 0;
assign sltiu = (op == 6'b100111) ? 1 : 0;
assign sw = (op == 6'b110000) ? 1 : 0;
assign lw = (op == 6'b110001) ? 1 : 0;
assign beq = (op == 6'b110100) ? 1 : 0;
assign bltz = (op == 6'b110110) ? 1 : 0;
assign j = (op == 6'b111000) ? 1 : 0;
assign jr = (op == 6'b111001) ? 1 : 0;
assign jal = (op == 6'b111010) ? 1 : 0;
assign halt = (op == 6'b111111) ? 1 : 0;
```

这些重复多次使用的变量可以用wire将它们存起来，在接下来的讨论中就不需要对6位的指令方法进行讨论，节省了开发时间。

到这里，基本上最复杂的控制模块就完成了。将这些控制模块输出，就可以对整个cpu进行控制了。

b. ALU模块的设计

ALU模块总体变化不大，唯一有变化的是加入了sign信号以及一些ALUOp的变更（这个真的血坑，ALUOp居然跟单周期的不一样？！很多bug不能马上找出来。希望老师能统一一下两个实验的ALUOp，免得继续坑害师弟师妹们）。总体的设计沿用了单周期的设计思想。

```
assign opA = ALUSrcA == 0 ? ReadData1 : {{27{1'b0}},sa[4:0]};
assign opB = ALUSrcB == 0 ? ReadData2 : immediate;
assign zero = result == 0 ? 1 : 0;
assign sign = result > 0 ? 1 : 0;
always @(ALUOp or ReadData1 or sa or ReadData2 or immediate or ALUSrcA or ALUSrcB) begin
    case(ALUOp)
        3'b000:
            result <= opA + opB;
        3'b001:
            result <= opA - opB;
        3'b010:
            result <= opA < opB ? 1: 0;
        3'b011:
            result <= (((opA<opB) && (opA[31] == opB[31] )) ||( ( opA[31] ==1 && opB[31] == 0))) ? 1:0;
        3'b100:
            result <= opB << opA;

        3'b101:
            result <= opA | opB;
        3'b110:
            result <= opA & opB;
        3'b111:
            result <= opA ^~ opB;
    endcase
end
```

c. 指令存储模块

也跟单周期的类似。指令的输出绑定指令地址。

```
initial begin
    $readmemb ("C:/Users/75654/Desktop/cpu_multi/ins_data.txt", im);
end
assign IDataOut[7:0]= RW == 1 ? im[IAddr+3] : 8'bz;
assign IDataOut[15:8]= RW == 1 ? im[IAddr+2] : 8'bz;
assign IDataOut[23:16]= RW == 1 ? im[IAddr+1] : 8'bz;
assign IDataOut[31:24]= RW == 1 ? im[IAddr] : 8'bz;
```

要注意一下存储指令的txt的编写，格式错误也可能导致一些神奇的bug。（还没遇到）

正常的写法

```
00001000 00000001 00000000 00001000
01001000 00000010 00000000 00000010
01000000 01000001 00011000 00000000
00000100 01100001 00100000 00000000
01000100 10000010 00101000 00000000
01100000 00000101 00101000 10000000
11010000 10100001 11111111 11111110
11101000 00000000 00000000 00010000
10011001 10000001 01000000 00000000
00001000 00001101 11111111 11111110
10011001 00001101 01001000 00000000
10011101 00101010 00000000 00000010
10011101 01001011 00000000 00000000
00001001 10101101 00000000 00000001
11011001 10100000 11111111 11111110
11100000 00000000 00000000 00010011
11000000 00100010 00000000 00000100
11000100 00101100 00000000 00000100
11100111 11100000 00000000 00000000
11111100 00000000 00000000 00000000
```

d. 指令寄存器的设计

用于存储指令，加入了解码的功能。

```
assign IR = IDataOut;
assign op = IR[31:26];
assign rs = IR[25:21];
assign rt = IR[20:16];
assign rd = IR[15:11];
assign immediate16 = IR[15:0];
assign sa = IR[10:6];
```

在单周期cpu中没有这个模块，因此解码和指令的存储都是放在顶层模块中的。将这些功能抽象出来，放在一个模块里，可以让项目结构更加清晰。

e. PC模块的设计

在多周期的指令集中加入了jal，jr等跳转指令，因此PC模块的控制信号PCSrc也进行了一些相应的变化

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；

值得注意的是，jal指令在跳转之前会将pc的地址存在31号寄存器中。但是，并不是在jr指令中直接跳转回这条指令（这样会导致死循环），而是跳到31号寄存器所存的pc地址的下一个指令地址，即pc+4.

大体实现如下

```
assign addressPlu4 = address + 4;
initial begin
    address <= 0;
end
always @(negedge CLK or negedge reset) begin
    if(reset == 0)
        address <= 0;
    else if(PCWre && state == 3'b000) begin
        case(PCSrc)
            2'b00:
                address <= address + 4;
            2'b01:
                address <= address + 4 + immediate * 4;
            2'b10: begin
                address <= ReadData1;
            end
            2'b11: begin
                address[31:28]<=addressPlu4[31:28];
                address[27:2]<=IDataOut[25:0];
                address[1:0]<=2'b00;
            end
        endcase
    end
end
```

f. 内存模块的实现

内存模块负责lw，sw指令对内存的读和写。由于加入了状态机制，因此不能让模块完全由时钟周期来控制，应该加入状态信息，让其在正确的周期对内存进行写入，不然，写入的结果有可能会出错。读取模块影响的是lw指令，但是不需要特地加入状态来控制输出，只需要在寄存器的写入中加入相应的控制信号和状态，就能保证lw不会出错。

```
// 读
assign Dataout[7:0] = (RD==1)?ram[address + 3]:8'bz;
assign Dataout[15:8] = (RD==1)?ram[address + 2]:8'bz;
assign Dataout[23:16] = (RD==1)?ram[address + 1]:8'bz;
assign Dataout[31:24] = (RD==1)?ram[address ]:8'bz;
// 写
always@( negedge clk ) begin
    if( WR==1 && state == 3'b100) 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
```

g. 寄存器组模块

如上文所说，这里需要加入一个写入的状态信号。另外，这次写入信号变成了2位，有三种不同的写入信号。因此要用case判断一下。当然也可以用if-else

这里用的是if-else的写法。

```
always @ (negedge CLK or negedge RST) begin
    /*case(RegDst)
        2'b00: WriteReg = 2'h1f;
        2'b01: WriteReg = rt;
        2'b10: WriteReg = rd;
        2'b11: WriteReg = 0;
    endcase*/
    if (RegDst == 2'b00)
        WriteReg = 2'h1f;
    else if (RegDst == 2'b01)
        WriteReg = rt;
    else if (RegDst == 2'b10)
        WriteReg = rd;
    else
        WriteReg = 0;
    if (RST==0) begin
        for(i=1;i<32;i=i+1)
            regFile[i] <= 0;
    end
    else if(RegWre == 1 && WriteReg != 0 && state == 3'b011)
        regFile[WriteReg] <= WriteData;
    else if (RegWre == 1 && state == 3'b001 && RegDst == 2'b00)
        regFile[31] <= address + 4;
end
```

h. 数位拓展模块

这是最简单的模块，只要判断是符号拓展还是无符号拓展即可。

```
module signZeroExtend(ExtSel,immediate16,immediate32);
input ExtSel;
input [15:0] immediate16;
output [31:0] immediate32;
assign immediate32 = ExtSel == 0 ? {{16{1'b0}},immediate16} : {{16{immediate16[15]}},immediate16};
endmodule
```

i. 顶层模块

比上次的顶层模块稍微要简单一点。顶层模块依然是将所有的其他模块连接起来。新增了IR模块

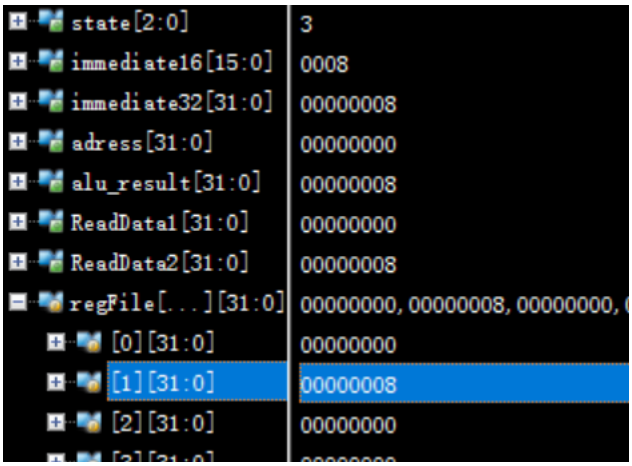
```
IR ir(CLK,state,IDataOut,op,rs,rt,rd,immediate16,sa);
PC pc(CLK,pc_reset,PCWre,state,PCSrc,immediate32,ReadData1,IDataOut,adress);
RegFile regFile(CLK,regfile_reset,RegWre,RegDst,
DBDataSrc,WrRegDSrc,rs,rt,rd,alu_result,ram_dataout,adress,
state,ReadData1,ReadData2);
RAM ram(CLK,alu_result,ReadData2,mRD,mWR,state,ram_dataout);
IM im(adress,rw,IDataOut);
ALU alu(ALUOp,ReadData1,sa,ReadData2,immediate32,ALUSrcA,ALUSrcB,alu_result,zero,sign);
ControlUnit control(CLK,op,zero,sign,PCWre,ALUSrcA,ALUSrcB,
DBDataSrc,RegWre,WrRegDSrc,InsMemRW,mRD,mWR,IRWre,RegDst,ExtSel,PCSrc,
ALUOp,state);
signZeroExtend extend(ExtSel,immediate16,immediate32);
endmodule
```

3. 测试

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	00010	00000	00000000 00000010		48020002
0x00000008	or $3,**$2**,$1	010000	00 010	00001	00011000 00000000		40411800
0x0000000C	sub $4,**$3**,$1	000001	00 011	00001	00100000 00000000		04612000
0x00000010	and $5,**$4**,$2	010001	00 100	00010	00101000 00000000		44822800
0x00000014	**sll $5,$5,2**	011000	00 000	00101	00101000 10000000		60052880
0x00000018	**beq $5,$1,-2(=,转 14)**	110100	00 101	00001	11111111 11111110		D0a1ffe
0x0000001C	**jal 0x0000040**	111010	00 000	00000	00000000 00010000		E8000010
0x00000020	**slt $8,$12,$1**	100110	01 100	00001	01000000 00000000		99814000
0x00000024	**addi $13,$0,-2**	000010	00 000	01101	11111111 11111110		080dfffe
0x00000028	**slt $9,$8,$13**	100110	01 000	01101	01001000 00000000		990d4800
0x0000002C	**sltiu $10,$9,2**	100111	01 001	01010	00000000 00000010		9d2a0002
0x00000030	**sltiu $11,$10,0**	100111	01 010	01011	**00000000 00000000**		9d4b0000
0x00000034	addi $13,$13,1	000010	01 101	01101	**00000000 00000001**		09ad0001
0x00000038	**bltz $13,-2 (<0,转 34)**	110110	01 101	00000	**11111111 11111110**		D9a0fffe
0x0000003C	**j 0x000004C**	111000	00 000	00000	**00000000 00010011**		E0000013
0x00000040	sw $2,4($1)	110000	00 001	00010	**00000000 00000100**		C0220004
0x00000044	lw $12,4($1)	110001	00 001	01100	**00000000 00000100**		C42c0004
0x00000048	**jr $31**	111001	11 111	00000	**00000000 00000000**		E7e00000
0x0000004C	**halt**	111111	00000	00000	0000000000000000	=	FC000000
0x00000050							
0x00000054							

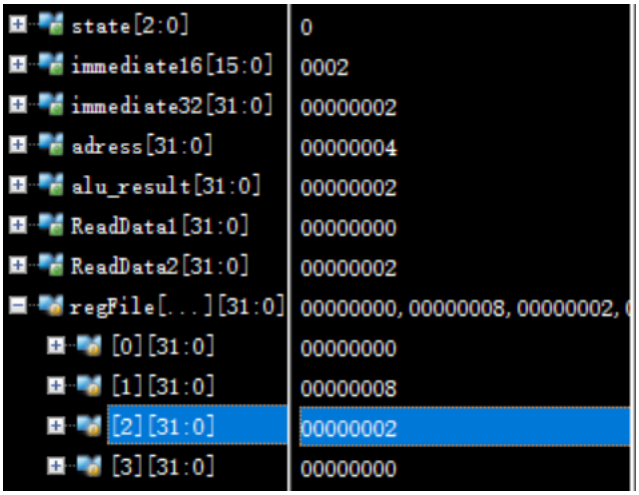
a. addi $1,$0,8

0 + 8，将结果写入到1号寄存器



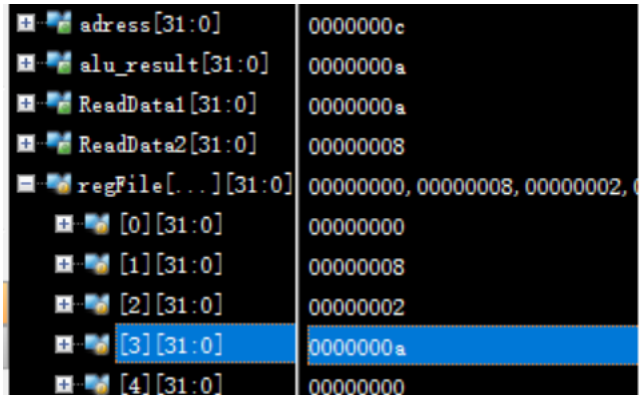
b. ori $2,$0,2

0 或2，将结果写入到2号寄存器



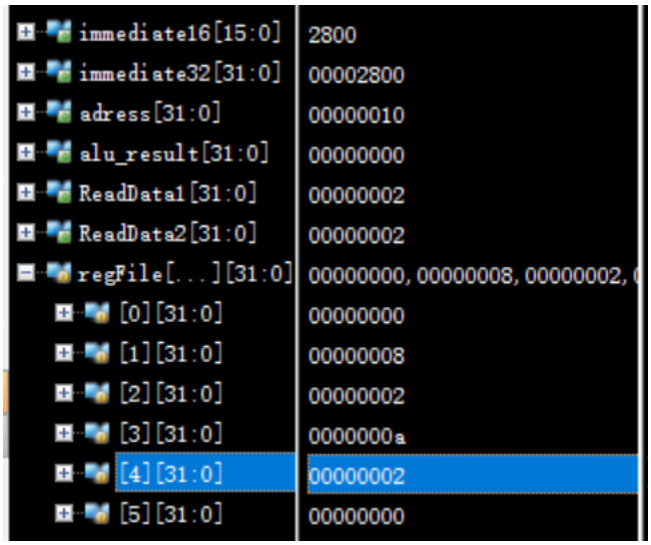
c. or $3,**$2**,$1

1号寄存器或2号寄存器写入到三号寄存器。结果预测为10



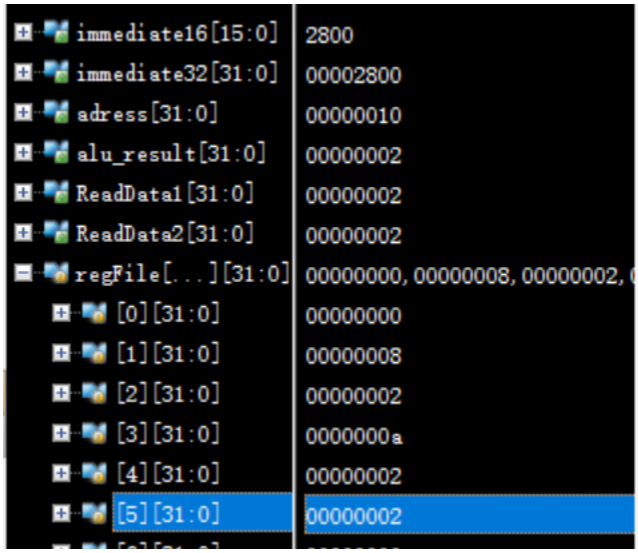
d. sub $4,**$3**,$1

3号寄存器减去1号寄存器，结果写到4号寄存器。预测为2



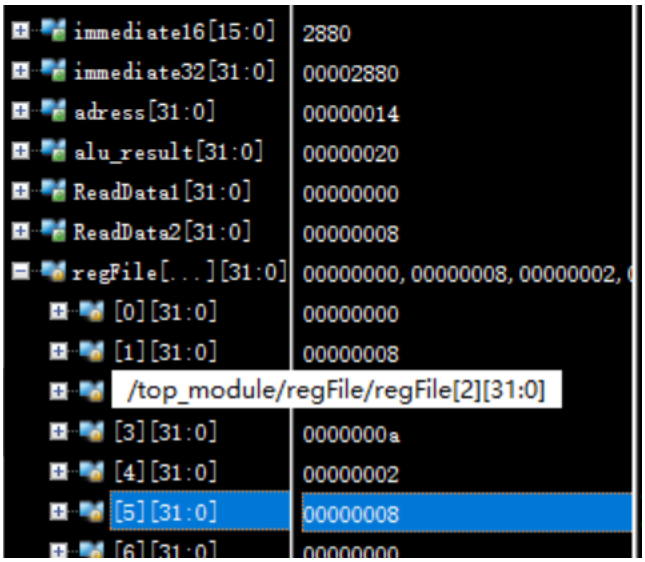
e. and $5,$4,$2

4号寄存器与2号寄存器写入到5号寄存器。预测结果为2

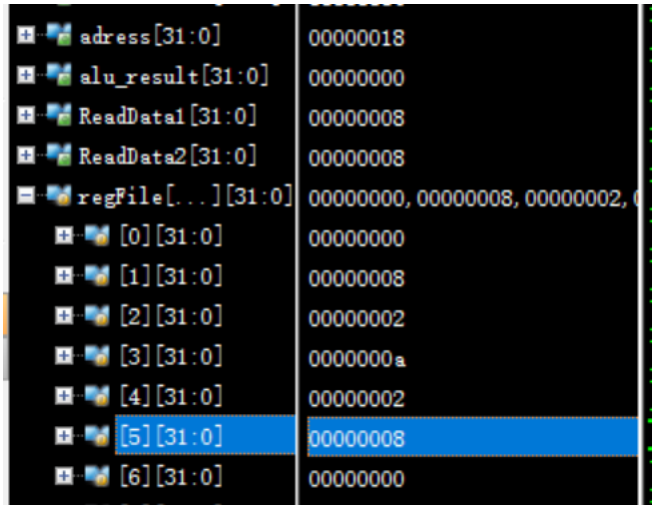


f. **sll $5,$5,2**

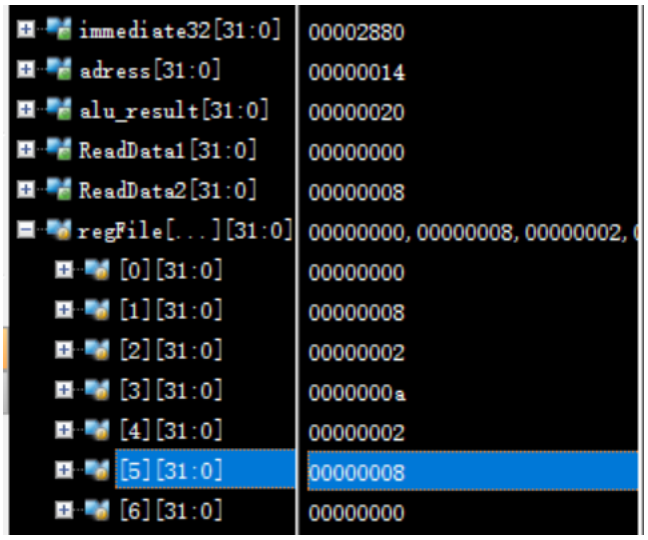
5号寄存器左移两位，写入到5号寄存器，结果预测为8.



g. **beq $5,$1,-2(=,转14)**

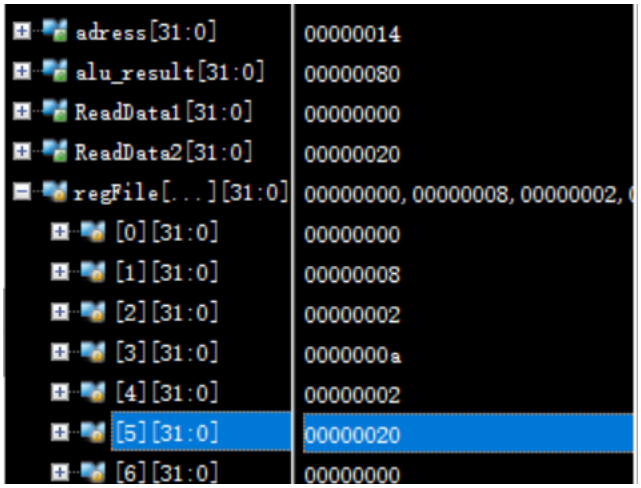


下一个地址为14



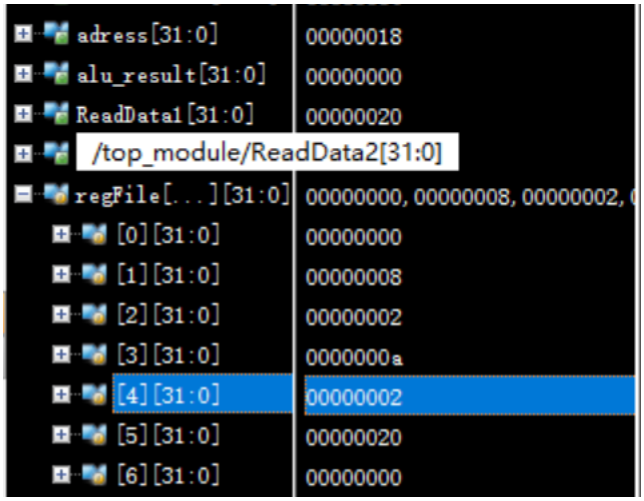
h. **sll $5,$5,2**

5号寄存器移动两位，写入到5号寄存器。预测为32

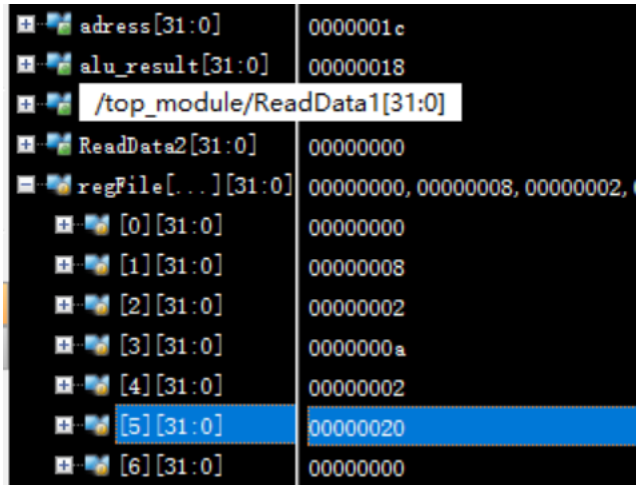


I. **beq $5,$1,-2**

不等于，跳转到下一条。

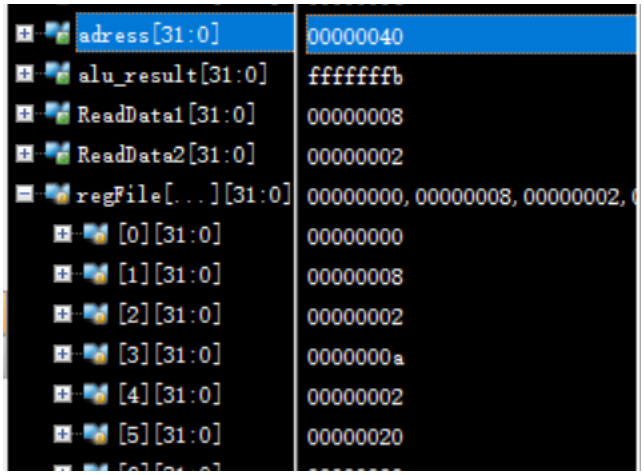


地址为1C



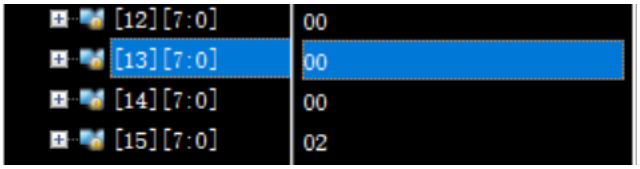
i. **jal 0x0000040**

跳到pc40，20写入到31号寄存器。



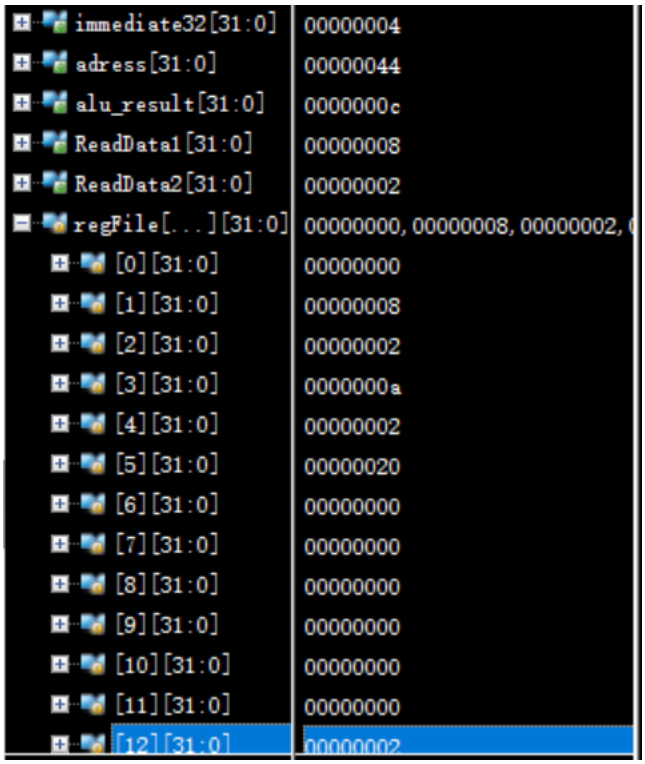
j. sw $2,4($1)

将2号寄存器写入到内存地址为12的空间中。



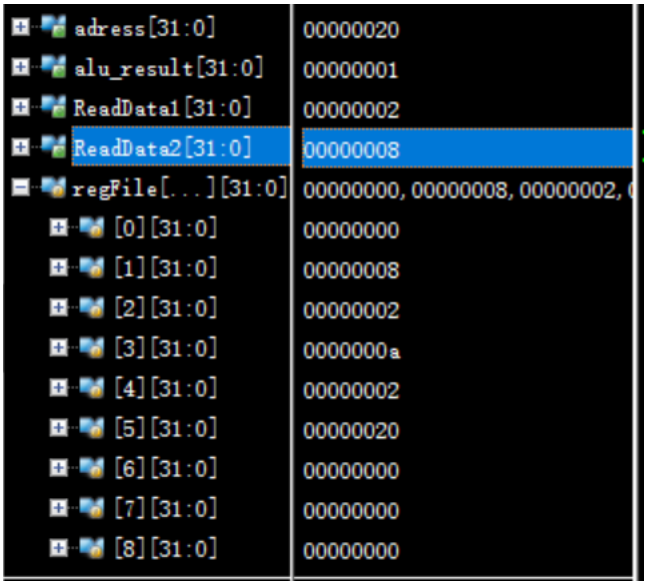
k. lw $12,4($1)

将内存空间12为起始地址的内容写入到12号寄存器。12号寄存器变为2



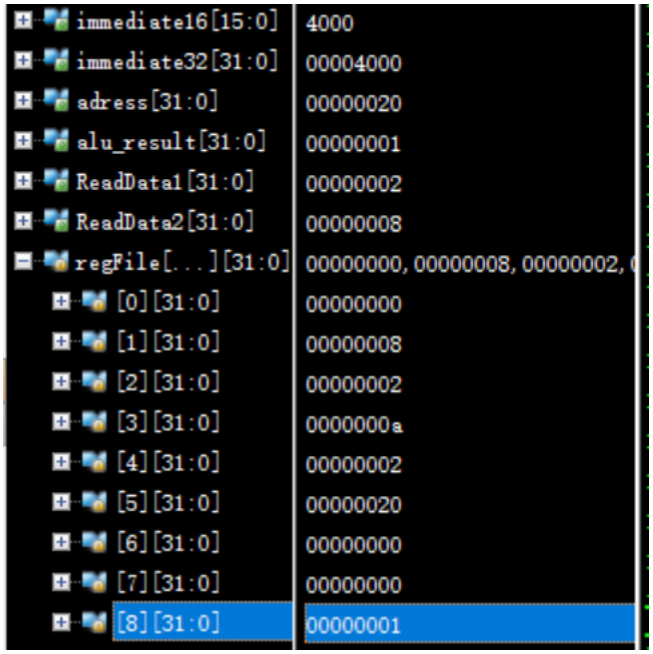
l. **jr $31**

跳回PC20



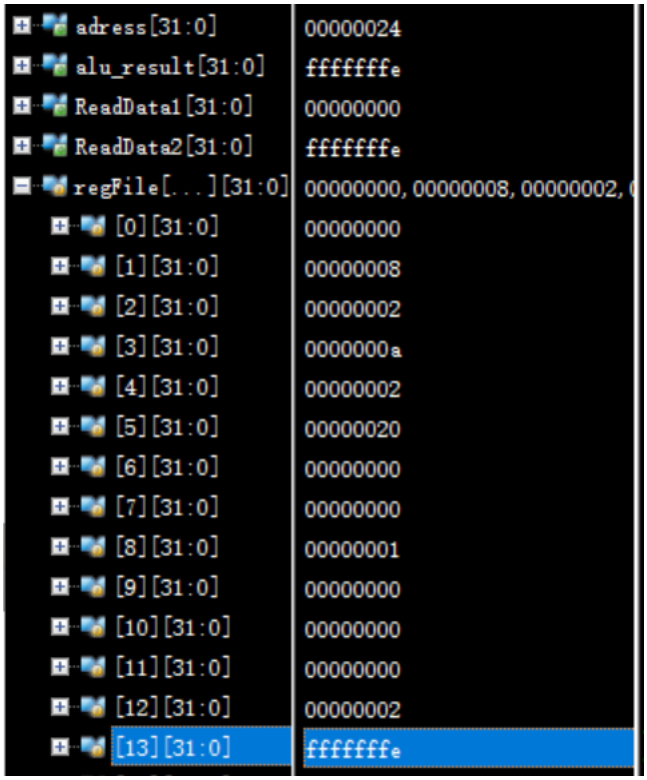
m. **slt $8,$12,$1**

如果12号寄存器中的值小于1号寄存器，8号寄存器写入1.预测8号寄存器写入1



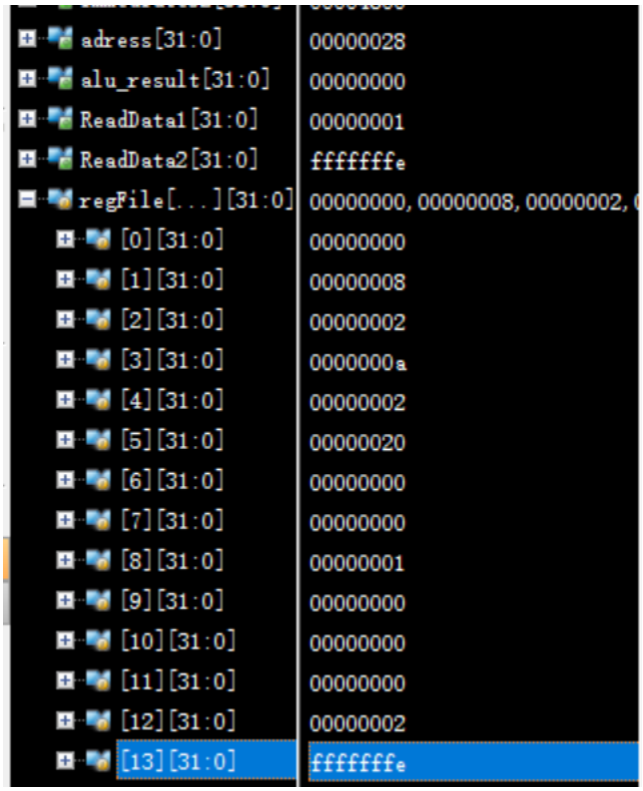
n. **addi $13,$0,-2**

0-2写入到13号寄存器。13号寄存器结果为-2



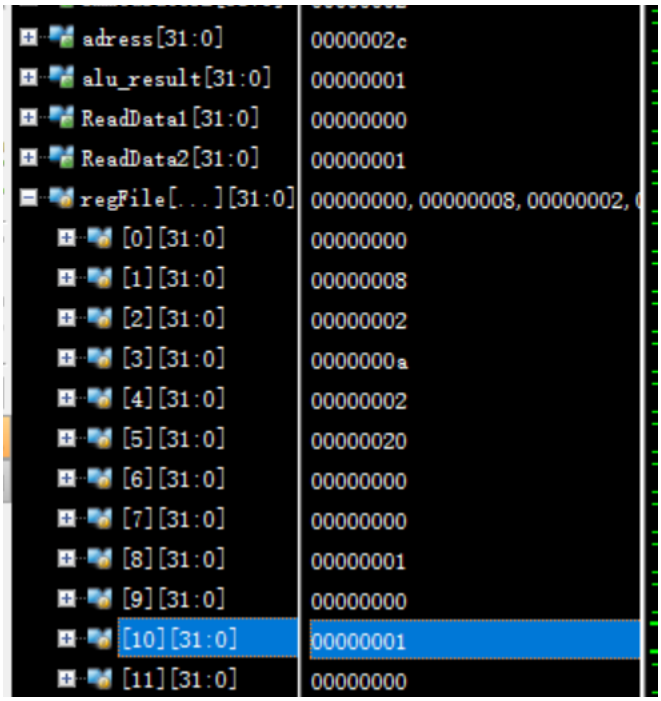
o. **slt $9,$8,$13**

8号寄存器小于13号寄存器，9号寄存器写入1.预测9号为0



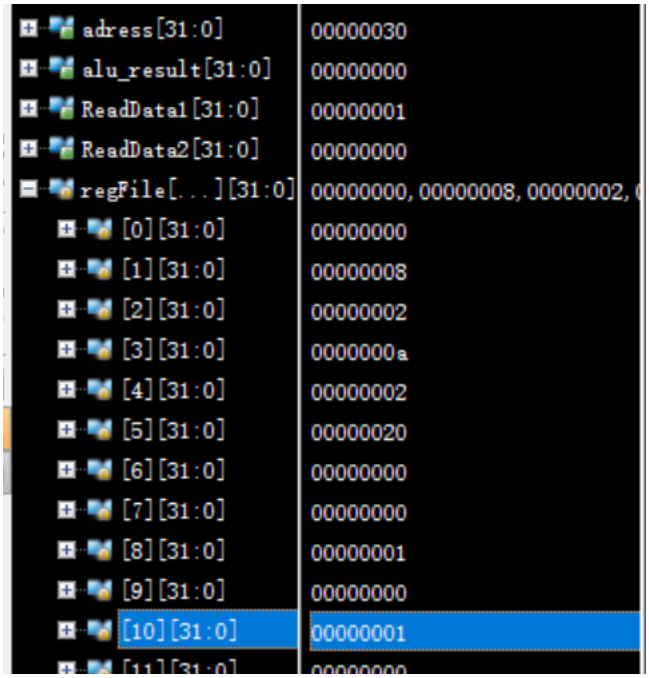
p. **sltiu $10,$9,2**

9号寄存器比2小，则10写入1.预测10写入1



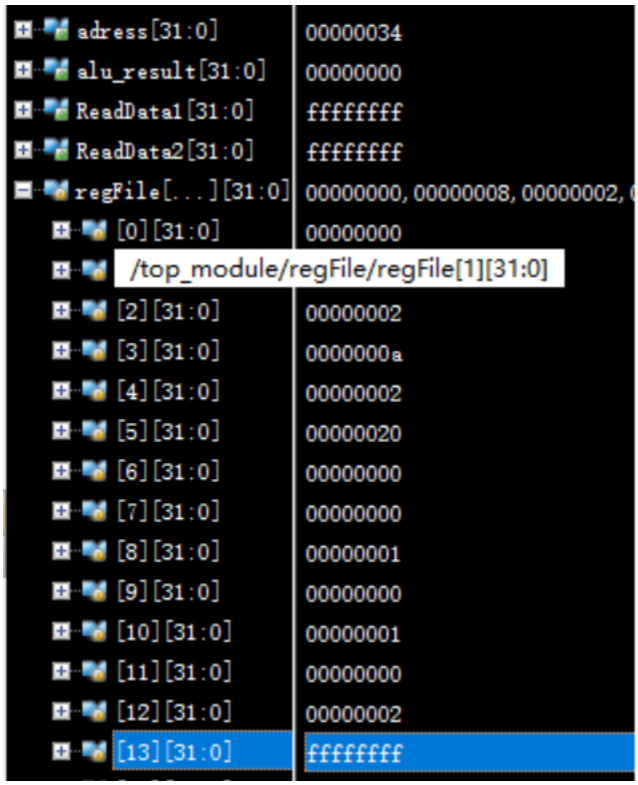
q. **sltiu $11,$10,0**

10比0小，则11写入1.预测11为0



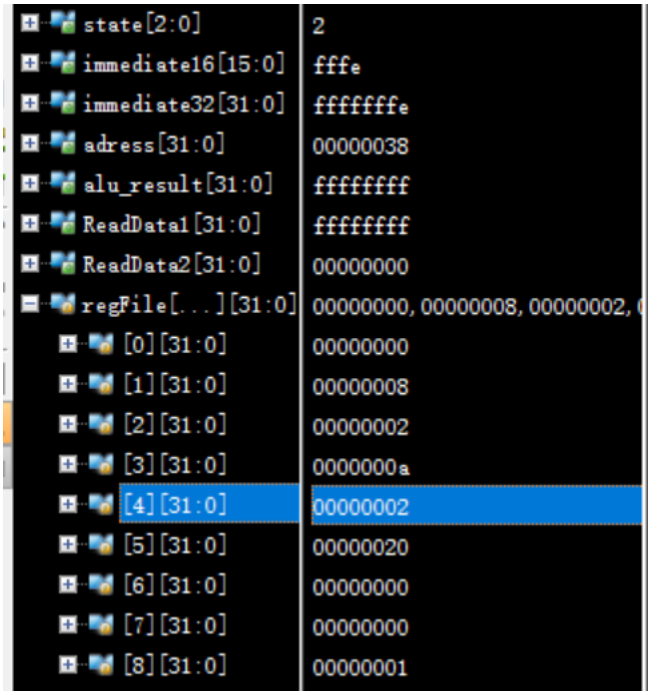
r. addi $13,$13,1

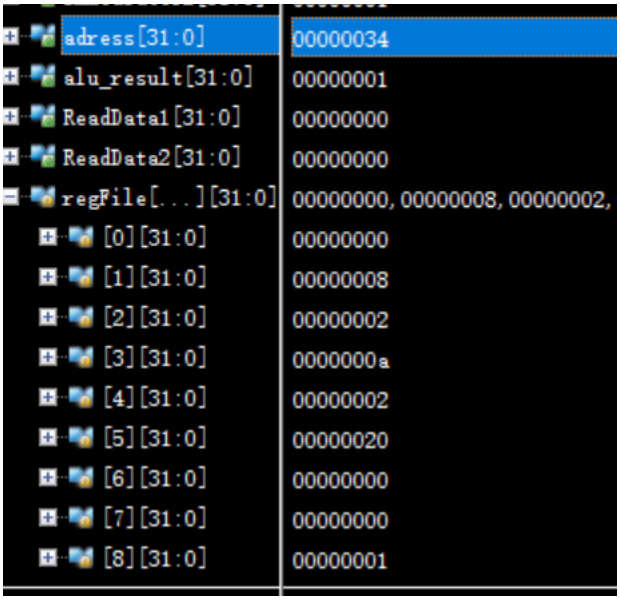
13号寄存器的值加1，写入13。预测为-1



s. **bltz $13,-2 (<0,转34)**

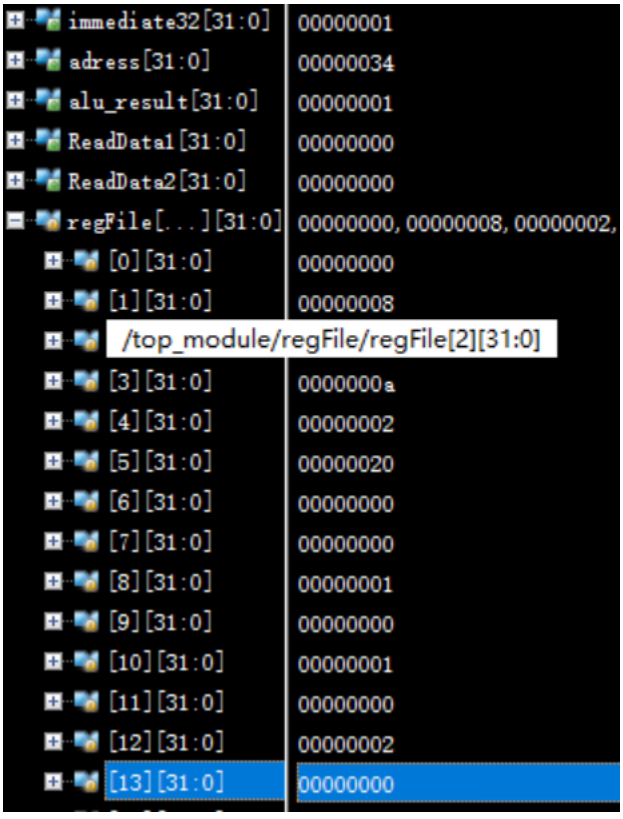
13号寄存器的值小于0，则跳回到34.预测跳回34





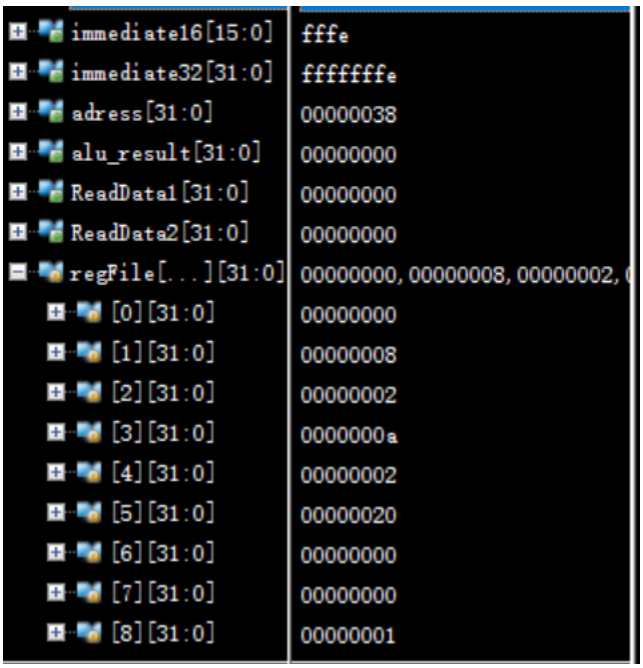
t. addi $13,$13,1

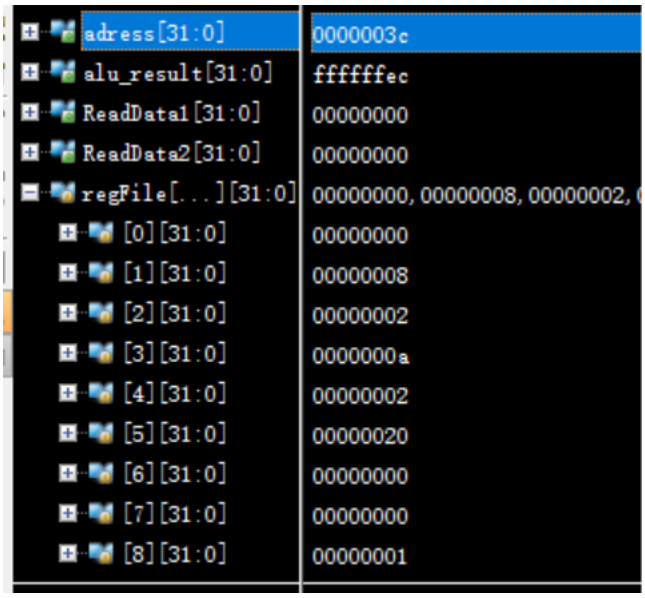
13号寄存器的值加1，写入到13号寄存器。预测13号寄存器的值为0



u. **bltz $13,-2**

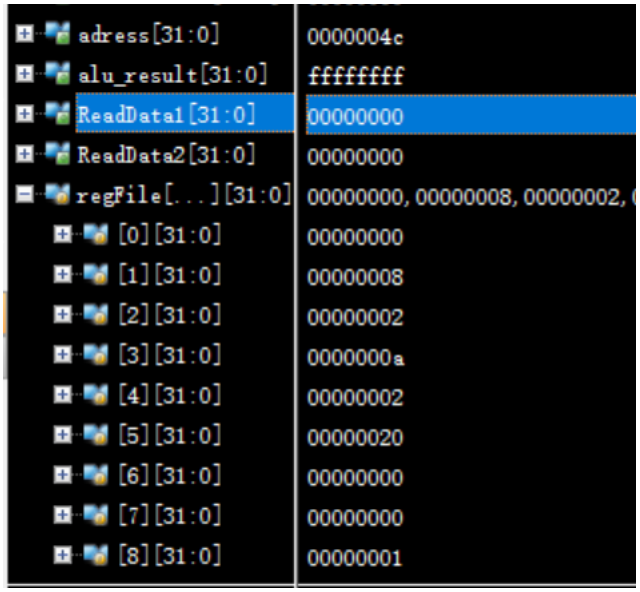
不跳转



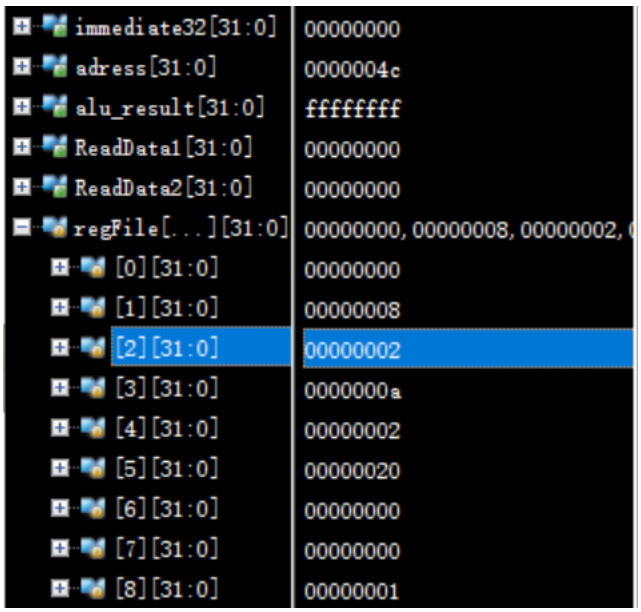


v. **j 0x000004C**

跳转到4C



w. **halt**



六. 实验心得

多周期cpu的设计跟单周期基本相似，在了解了单周期cpu的设计之后，这次设计可以在较短的时间内完成。但是和单周期一样，同样的遇到了不少问题。

首先是控制模块。这个模块是cpu设计中最重要的部分。多周期cpu的设计中，控制模块引入了状态控制机制。状态是由指令和时钟共同控制的，状态和指令控制着其他控制信号，进而控制其他各个模块的工作。在控制状态更迭的过程中，遇到的第一个问题依旧是时钟沿控制的问题。在一开始的设计中，pc和状态更迭都用了时钟的下降沿，但这样会导致地址、

指令变成高阻态（即不能读出来），进而影响状态更迭。处理方法也很简单，就是让它们由不同的时钟沿触发。另外一个问题，在更新其余控制信号的时候，由于每个指令对应的2进制码都有6位，由于位数太多，很容易造成手误进而影响控制信号。这时候就要用其他编程语言的思想，让这些指令变成有意义的符号，这样就不容易造成手误。在这里采用的方法是声明wire变量，每个wire变量对应一个指令，变量名设置为指令名。

其次是ALU模块。ALU模块是这次debug花费时间最多的模块。由于老师换掉了ALU控制信号对应的操作，沿用上次的模块就会造成错误。跟其他错误混在一起很难想得到回事ALUOp的问题。当然，解决方法也很简单。强烈建议老师修改一下这两个文档，统一一下单周期和多周期的ALUOp，以免继续坑害师弟师妹。

多周期cpu的设计是这学期最后一个实验了，总结一下计组实验课给我带来的影响吧。在这三次实验的设计中，最有意义的是让我们了解了课堂上不怎么记得的汇编指令，以及似懂非懂的cpu。在理论课上，看到分线杂乱的数据通路图，完全不知道信号怎样在cpu中传递。在设计了两次cpu之后，对cpu有了一个更加清晰的了解——我觉得这也是计组实验课最重要的意义。

课程总结

这学期一共做了三个实验：汇编语言、单周期 cpu 和多周期 cpu 的设计。在第一个实验里，我们熟悉了 c 语言到汇编语言的转化，学会了用汇编语言编写简单的程序。这对我们理解编译和链接机制有很大的帮助。第二个实验是单周期 cpu 的设计。我认为这是难度比较大的一次实验设计。尽管我们在理论课上已经学习了 cpu 的数据通路，但是，在自己设计时，还要想清楚很多细节，并且将这些模块有机地连接在一起，使他们成为一个能够协同工作的整体——我认为这是对我们很大的一个挑战。另外，在这次实验里，还需要写 basys3 板，要求我们用晶体管显示寄存器的信息，使得我们要写更多的模块。最后一个实验是多周期 cpu 的设计。由于有了单周期 cpu 的基础和现成的模块，在设计多周期 cpu 的过程中遇到的问题相对较少——只要重新设计控制模块以及对其他模块进行修改，就可以设计出一个多周期的 cpu。

我认为，计组实验是一门很重要的实验课。虽然学分占比不高，但是能让我们更加细致的了解理论课上的知识，更直观的接触 cpu 的基本工作原理，能让我们学到比理论课上更多的知识。因此，我建议计组实验课的学分可以适当的提高，让同学们能够更加重视这门实践课，这对巩固同学们的理论基础是相当有帮助的。

同时，我也建议老师能够在讲完了要讲的内容之后，能够早一点让我们下课，让我们自己安排时间。想留下来的同学可以呆在实验室，而其他同学可以去做别的事情。实验室里没有网络，对我们学习影响相当大。