《计算机组成原理与接口技术实验》

实验报告

（实验三）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 软件工程四（7）班
学 生 姓 名 ：	徐伟元
学　　号 ：	16340261
时　　间 ：	2018　年　6　月　21　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:

==>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。

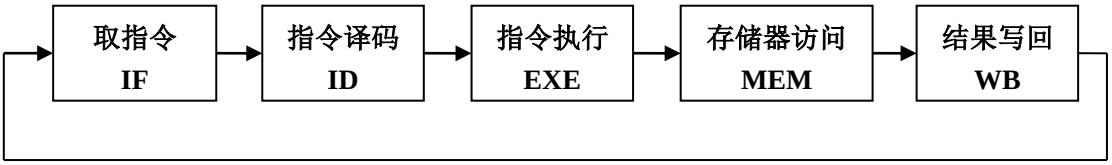
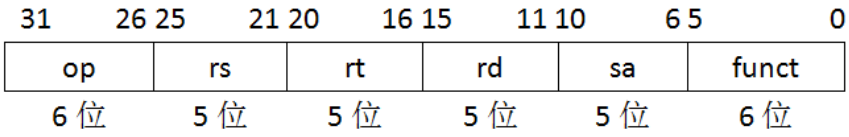


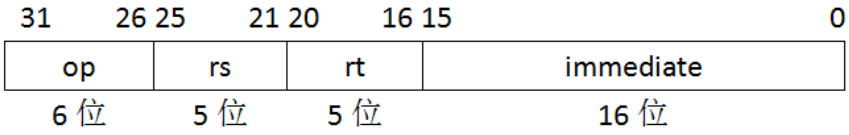
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式：

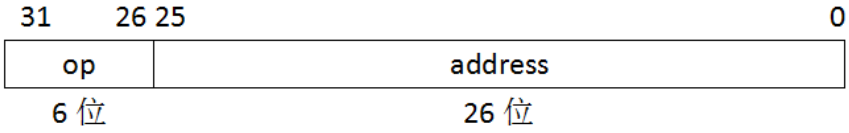
R 类型：



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。

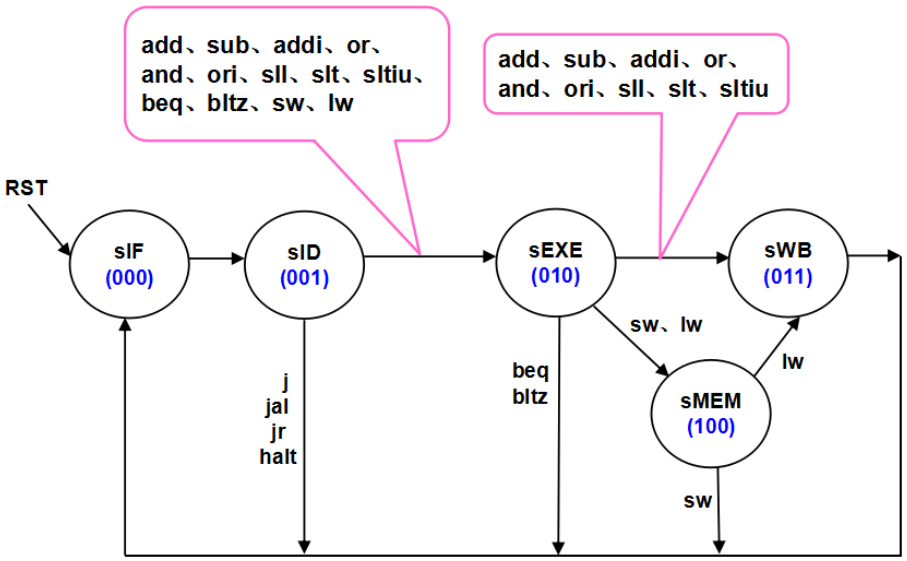


图 2 多周期 CPU 状态转移图

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。

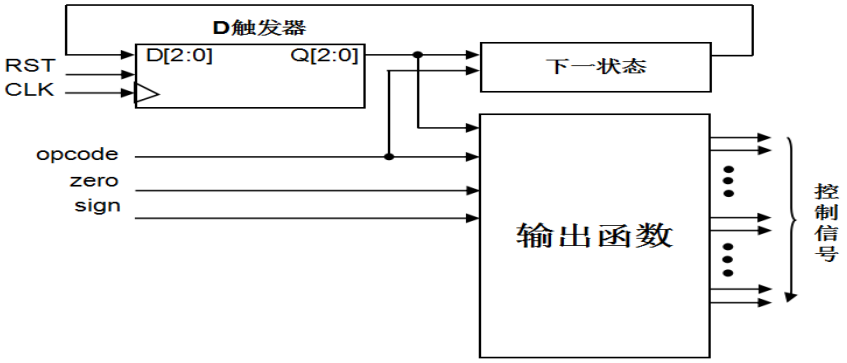


图 3 多周期 CPU 控制部件的原理结构图

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。

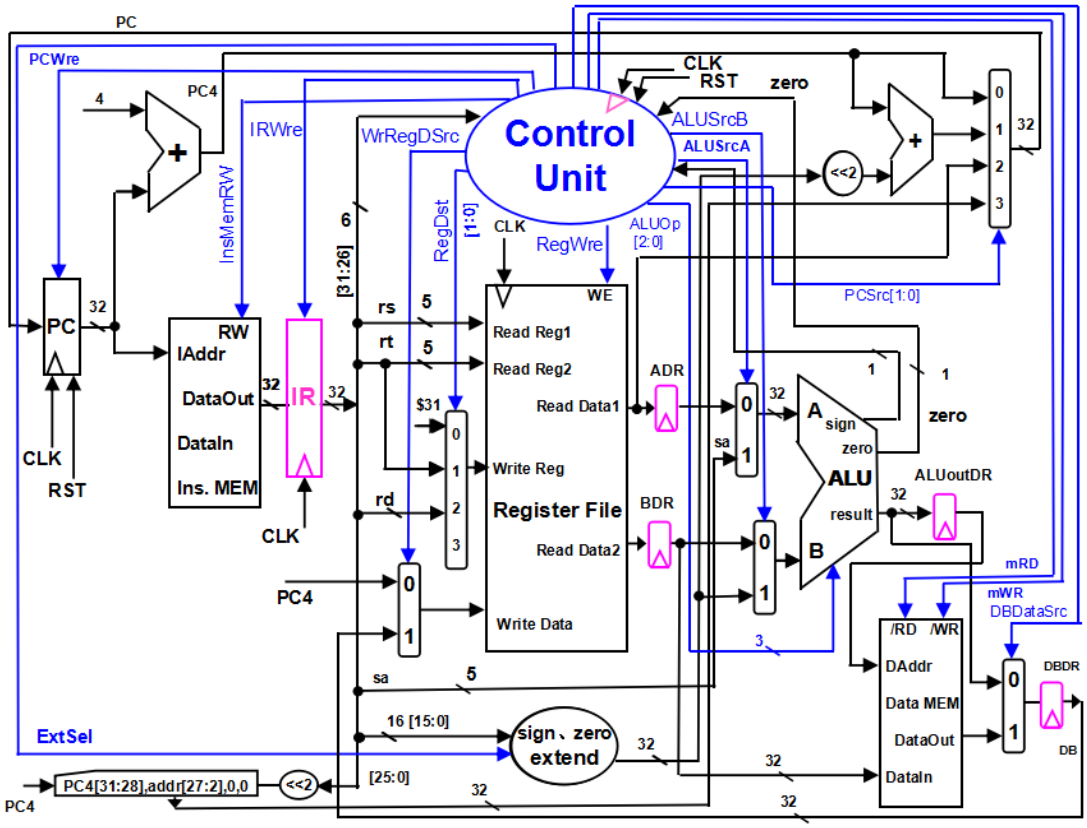


图 4 多周期 CPU 数据通路和控制线路图

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	$Y = A \oplus B$	**异或**

四、实验设备

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

五、实验过程与结果

1. 模块设计思路与代码

对于与单周期逻辑和代码相同的模块，不多做赘述。代码展示关键部分。

1) PC

除 NextPC 增加了在寄存器组输出的跳转地址外，与单周期一致，所以，增加 NextPC 的 case 判断即可。

```
case (PCSrc)
  2'b00: NextPC = PC + 4;
  2'b01: NextPC = PC + 4 + (Immediate << 2);
  2'b10: NextPC = ReadData1;
  2'b11: NextPC = JPC;
  default: NextPC = PC + 4;
endcase
```

2) InstructionMemory

与单周期无区别，更改指令集文件路径即可。

3) IR

在时钟下降沿写入指令二进制码，读取则无需时钟信号。

```
module IR(clk, IRWre, dataIn, dataOut);
  input clk;
  input IRWre;
  input [31:0] dataIn;
  output reg [31:0] dataOut;

  always @(negedge clk) begin
    if (IRWre) begin
      dataOut <= dataIn;
    end
  end
endmodule
```

4) RegisterFile

与单周期无区别，注意写寄存器地址的三选一选择器和写数据的二选一选择器的加入。使用 case 完成选择器模块，代码简单，略去。

5) SignZeroExtend

与单周期无区别。

6) ADR & BDR & ALUOutDR & DBDR

切割数据通路，注意时钟下降沿写即可。四者逻辑相同，命名不同。

```
module DR(dataIn, clk, dataOut);
  input clk;
  input [31:0] dataIn;
  output reg [31:0] dataOut;

  always @(negedge clk) begin
    dataOut <= dataIn;
  end
endmodule
```

7) ALU

注意结合实验原理中 ALU 真值表，利用 case 语句完成功能对应即可。

```
always @(ALUop or A or B) begin
    case (ALUop)
        3'b000 : Y = (A + B);
        3'b001 : Y = (A - B);
        3'b010 : Y = (A < B) ? 1 : 0;
        3'b011 : Y = (((A < B) && (A[31] == B[31])) ||
                      ((A[31] && !B[31]))) ? 1 : 0;
        3'b100 : Y = (B << A);
        3'b101 : Y = (A | B);
        3'b110 : Y = (A & B);
        3'b111 : Y = (A ^ B);
        default : Y = 0;
    endcase
end

assign Zero = (Y == 0) ? 1 : 0;
assign Sign = Y[31];
```

8) DataMemory

仍旧采用 8 位一字节的大端存储模式，与单周期保持一致。

9) ControlUnit

我们关注结合控制信号作用表与状态转移图，实现控制信号的赋值。按照控制部件的结构原理图，我们可以得出，控制单元首先根据当前状态和指令，进行状态转移。

```
always @(posedge clk or negedge rst) begin
  if (!rst) begin
        State <= 3'b000;
```

```
  end
  else begin
    case (State)
      `sIF: State <= `sID;
      `sID: begin
          if (opCode == `opJ || opCode == `opJal ||
              opCode == `opJr || opCode == `opHalt) State = `sIF;
          else  State <= `sEXE;
      end

      `sEXE: begin
        if (opCode == `opBeq || opCode == `opBltz) State <= `sIF;
        else if (opCode == `opSw || opCode == `opLw) State <=
`sMEM;
        else  State <= `sWB;
      end

      `sMEM: begin
          if (opCode == `opSw)  State <= `sIF;
          else if (opCode == `opLw)  State <= `sWB;
      end

      `sWB:   State <= `sIF;

      default: State <= `sIF;
    endcase
  end
end
```

控制信号的输出，取决于当前状态和指令。根据控制信号功能表与状态，列出状态，信号，指令真值表(电子文档目录内)。由于部分信号在不同指令下赋值在不同状态，下面根据真值表并逐一分析各控制信号的赋值：

a) **ALUSrcA**

只在执行阶段且 sll 指令时为 1。

```
ALUSrcA = (State == `sEXE && opCode == `opSll) ? 1 : 0;
```

b) **ALUSrcB**

只在执行阶段且 addi，opri，sltiu，sw，lw 指令时为 1。

```
ALUSrcB = (State == `sEXE && (opCode == `opAddi ||
           opCode == `opOri || opCode == `opSltiu ||
           opCode == `opSw || opCode == `opLw)) ? 1 : 0;
```

c) **DBDataSrc**

lw 指令时值为 1。

```
DBDataSrc = (opCode == `opLw) ? 1 : 0;
```

d) **mRD**

访存阶段且 lw 指令时值为 1。

```
mRD = (State == `sMEM && opCode == `opLw) ? 1 : 0;
```

e) **mWR**

访存阶段且 sw 指令时值为 1。

```
mWR = (State == `sMEM && opCode == `opSw) ? 1 : 0;
```

f) **ExtSel**

执行阶段且指令为 ori，sltiu 时，值为 0。

```
ExtSel = (State == `sEXE && (opCode == `opOri ||
          opCode == `opSltiu)) ? 0 : 1;
```

g) **RegDst**

真值表为：

状态	指令	值
sID	jal	00
sWB	addi、ori、sltiu、lw	01
其他	任意指令	10

```
if (State == `sID && opCode == `opJal)  RegDst = 2'b00;
else if (State == `sWB && (opCode == `opAddi ||
          opCode == `opOri || opCode == `opSltiu ||
          opCode == `opLw)) RegDst = 2'b01;
else  RegDst = 2'b10;
```

h) **WrRegDSrc**

只在译指阶段且指令为 jal 时值为 0。

```
WrRegDSrc = (State == `sID && opCode == `opJal) ? 0 : 1;
```

i) **RegWre**

在译指阶段且指令为 jal(写 31 号寄存器) 或写回阶段(当前状态所有指令)时，值为 1。

```
RegWre = ((State == `sID && opCode == `opJal) || State == `sWB) ?
1 : 0;
```

j) **IRWre**

IR 寄存器在取指阶段可写，值为 1。

```
IRWre = (State == `sIF) ? 1 : 0;
```

k) **PCSrc**

真值表为：

状态	指令	值
sIF,sID	j，jal	11
	jr	10
sEXE	beq(zero = 1)	01
	bltz(sign = 1 且 zero = 0)	
其他	任意	00

```
if ((State == `sIF || State == `sID) && (opCode == `opJr))
    PCSrc = 2'b10;
```

```
else if ((State == `sIF || State == `sID) &&
        (opCode == `opJ || opCode == `opJal)) PCSrc = 2'b11;
else if (State == `sEXE && ((opCode == `opBeq && zero) ||
        (opCode == `opBltz && sign && !zero)))  PCSrc = 2'b01;
else  PCSrc = 2'b00;
```

l) **ALUOp**

对照 ALU 功能表与指令功能，可以得出指令对应 ALU 功能关系。

```
case(opCode)
    `opSub, `opBeq, `opBltz:
        ALUOp = 3'b001;
    `opOr, `opOri:
        ALUOp = 3'b101;
    `opAnd:
        ALUOp = 3'b110;
    `opSll:
        ALUOp = 3'b100;
    `opSlt:
        ALUOp = 3'b011;
    `opSltiu:
        ALUOp = 3'b010;
    default:
        ALUOp = 3'b000;
endcase
```

m) **PCWre**

为了保证在每一条指令最开始（状态 000）的时钟上升沿 PC 发生改变，需要**在上一条指令的最后一个下降沿，将 PCWre 置为 1，这样才能保证 PC 在每一条指令的最开始的时钟上升沿发生改变。对应状态转移图，可以完成 PCWre 的赋值。**

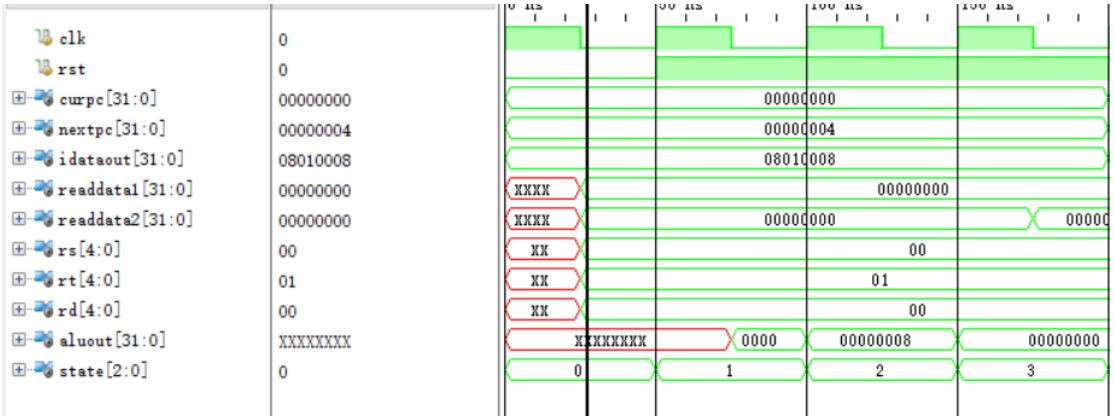
```
always @(negedge clk) begin
  case (State)
    `sID: begin
      if (opCode == `opJ || opCode == `opJal ||
          opCode == `opJr)  PCWre <= 1;
      end
    `sEXE: begin
      if (opCode == `opBeq || opCode == `opBltz) PCWre <= 1;
        end
    `sMEM: begin
      if (opCode == `opSw)  PCWre <= 1;
    end
    `sWB: PCWre <= 1;
    default:  PCWre <= 0;
  endcase
```

end

2. 仿真结果与分析

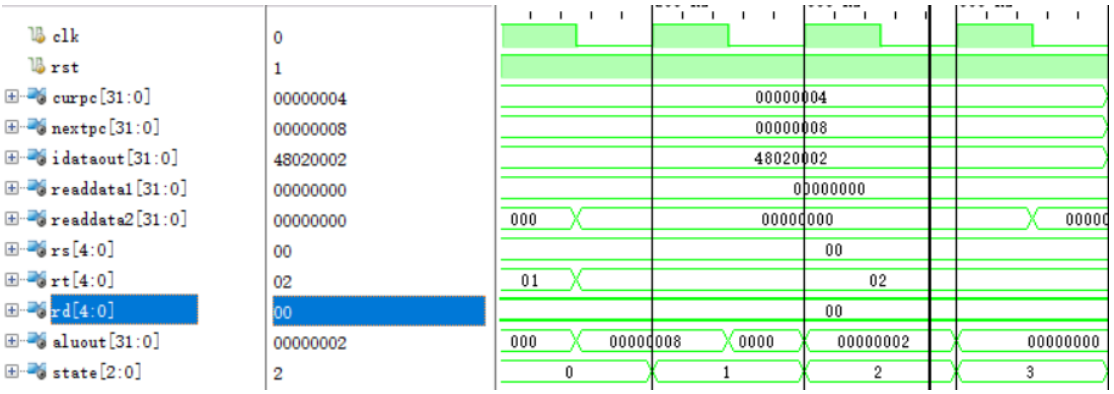
a) addi $1, $0, 8

rs 为 0，rt 为 1， aluout 在 sEXE(010) 阶段输出为 8 ($1 <= $0 + 8 = 0 + 8)，执行正确。



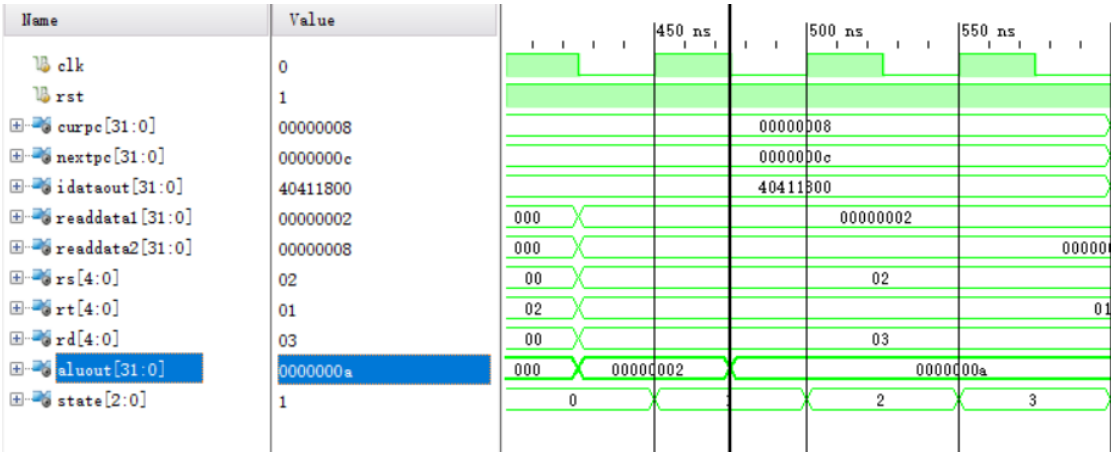
b) ori $2, $0, 2

rs 为 0， rt 为 2，aluout 在 sEXE(010) 阶段输出为 2 ($2 <= $0 | 2 = 0 | 2)，执行正确。



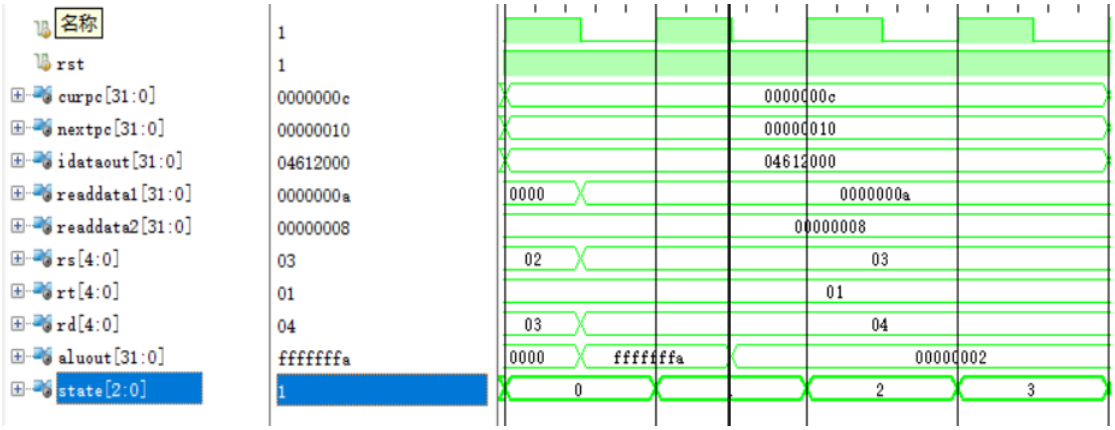
c) or $3, $2, $1

rs 为 2，rt 为 1，rd 为 3，aluout 为 10 ($3 <= $2 | $1 = 2 | 8 = 10)，执行正确。



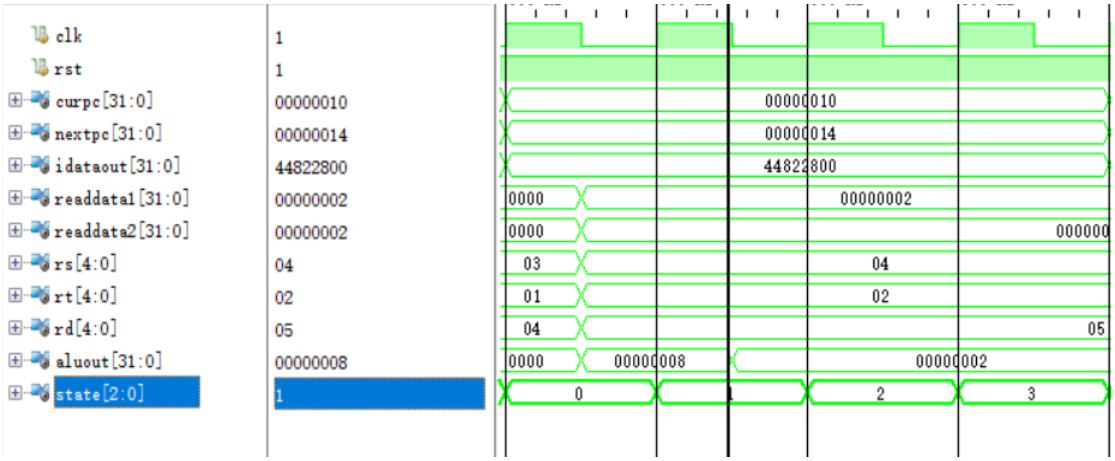
d) sub $4, $3, $1

rs 为 3，rt 为 1，rd 为 4，aluout 为 2 ($4 <= $3 - $1 = 10 – 8 = 2)，执行正确。



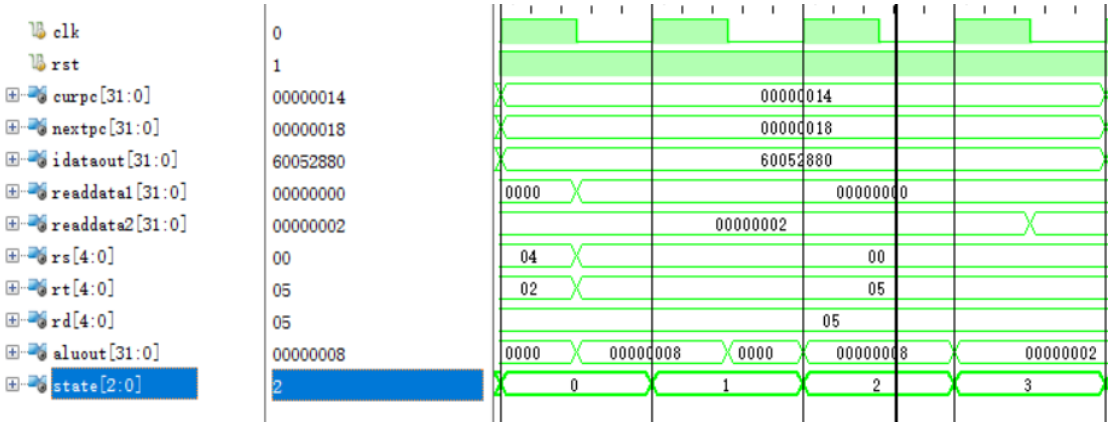
e) and $5, $4, $2

rs 为 4，rt 为 2，rd 为 5，aluout 为 2 ($5 <= $4 & $2 = 2 & 2 = 2)，执行正确。



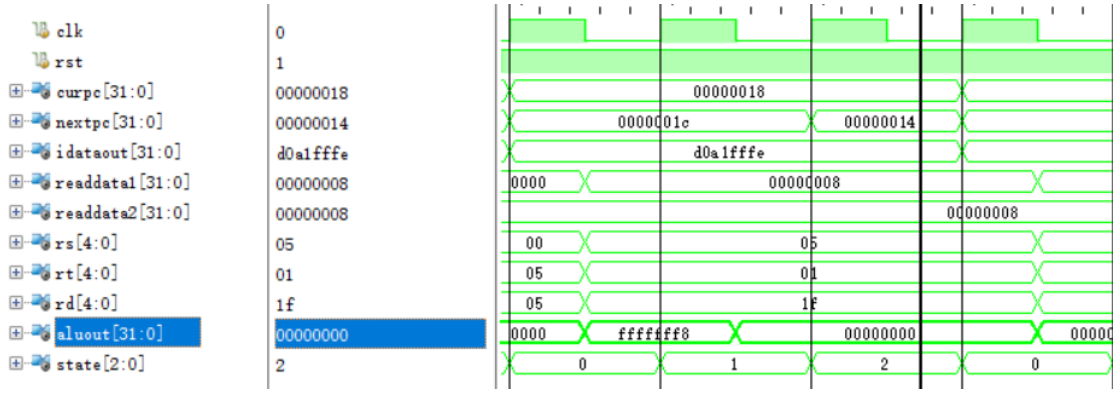
f) sll $5, $5, 2

rt 为 5，rd 为 5，aluout 为 8 ($5 <= $5 << 2 = 2 << 2 = 8)，执行正确。



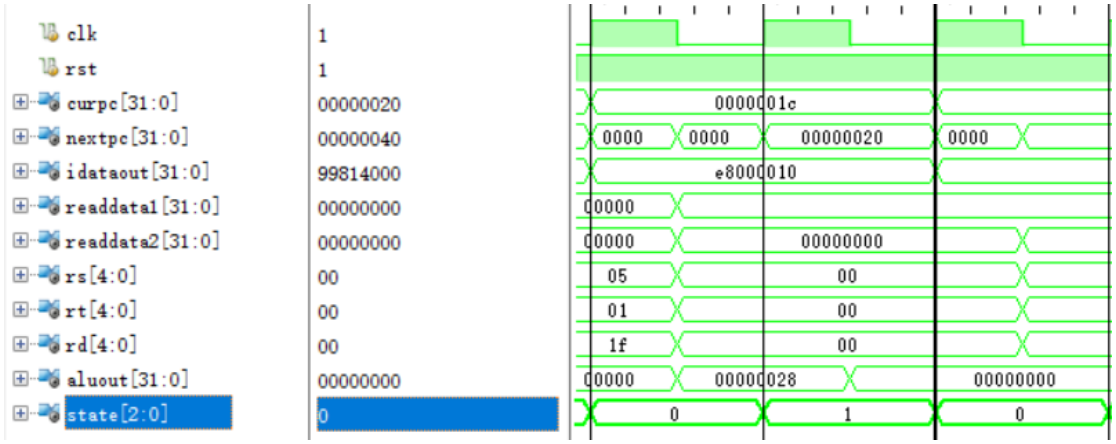
g) beq $5, $1, -2

在执行阶段(sEXE)，运算后得到 nextPC 为 14，且不执行后续阶段，跳转新 PC，执行正确。



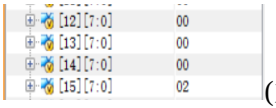
h) jal 0x00000040

译指阶段(sID，001)后调用子程序，无后续阶段，NextPC 为 40，执行正确。

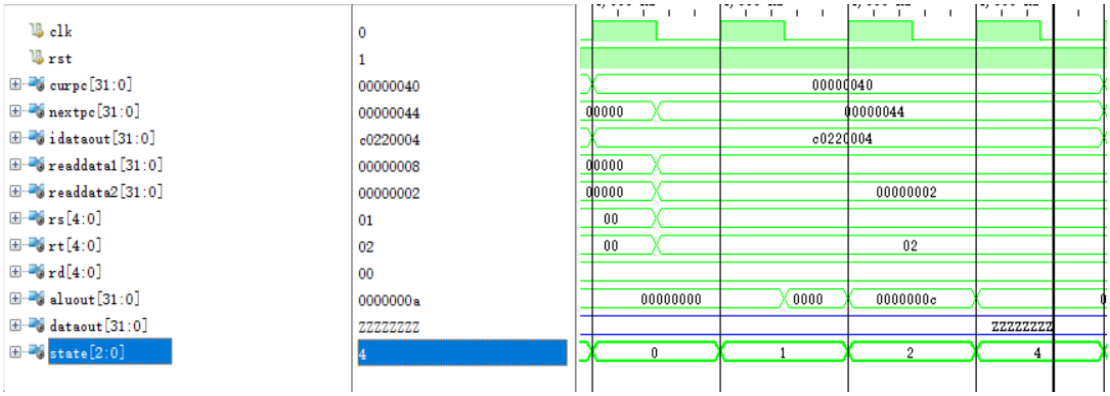


i) sw $2, 4($1)

rs 为 1，rt 为 2。执行阶段(sEXE)，aluout 结果为 12(12 = 4 + $1 = 4 + 8)，

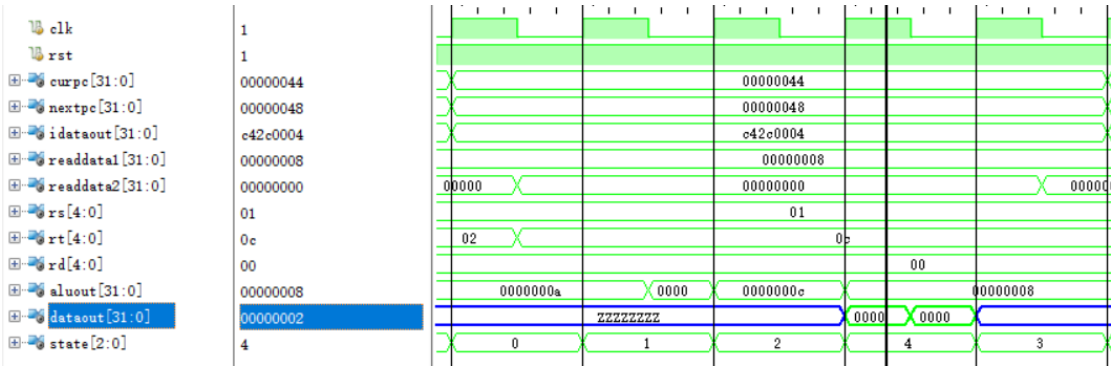
写入数据存储器，memory[12] = $2 = 2。 

(大端存储)



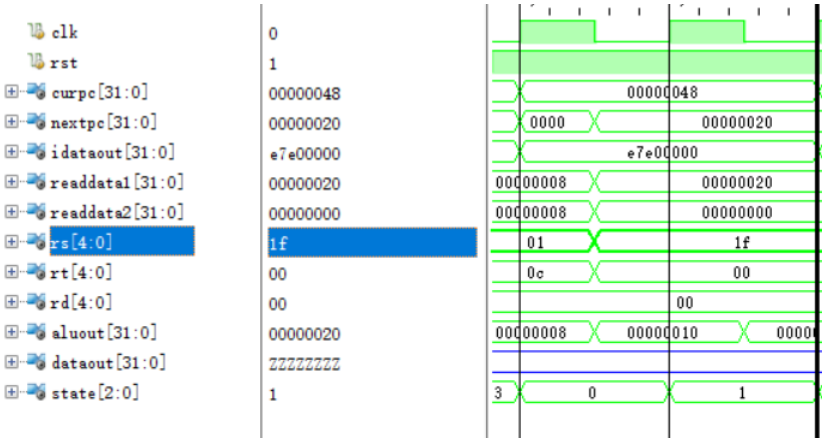
j) lw $12, 4($1)

经历 sIF(000)->sID(001)->sEXE(010)->sMEM(100)->sWB(011)阶段。在 sEXE(010)阶段，aluout 为 12(12 = 4 + $1 = 4 + 8)，读取 memory[12]；在 sMEM(100)阶段，取出数据，dataout 为 2；最后写回，执行正确。



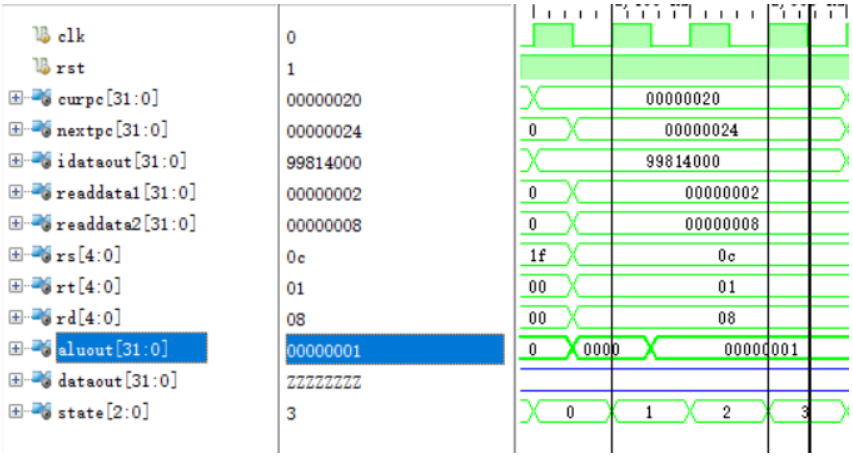
k) jr $31

子程序调用完毕，跳转回原地址，从$31 寄存器读取原地址。readdata1 值为 20(子程序调用时写入$31)。NextPC 为 20，执行正确。



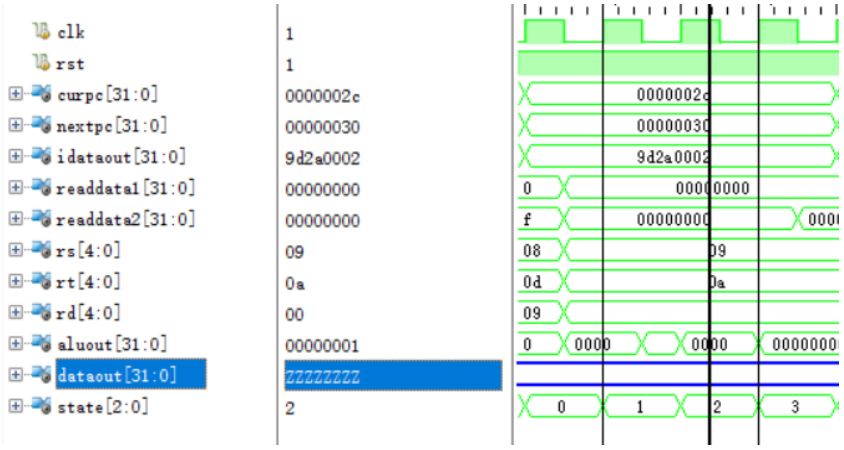
l) slt $8, $12, $1

rs 为 12，rt 为 1，rd 为 8。sEXE(010)阶段，aluout 为 1($12 = 2 < 8 = $1)，执行正确。



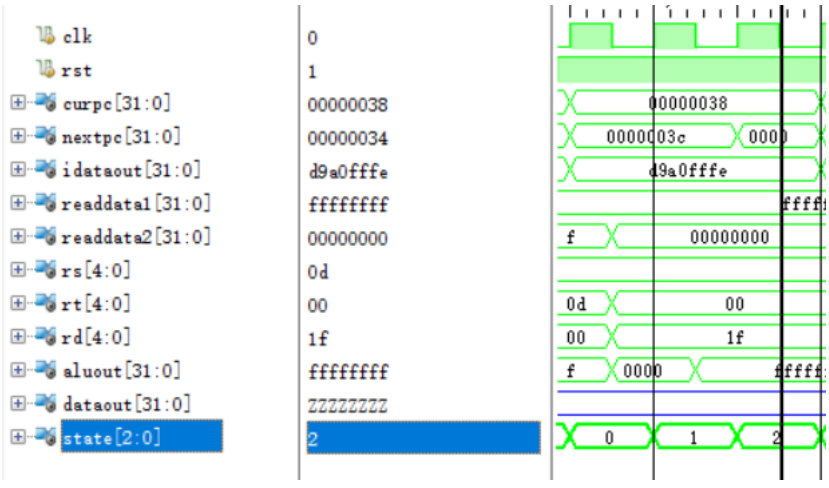
m) sltiu $10, $9, 2

rs 为 9，rt 为 10，aluout 为 1($10 <= $9 = 0 < 2)，执行正确。



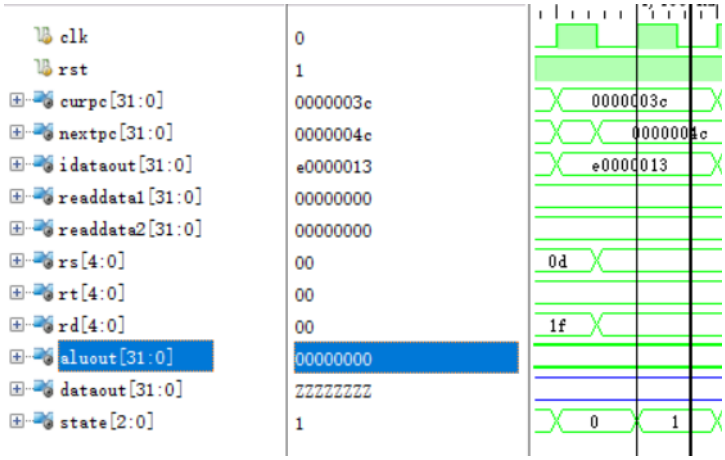
n) bltz $13, -2

rs 为 13，$13 = -1 < 0，所以 NextPC 为 34，执行正确。



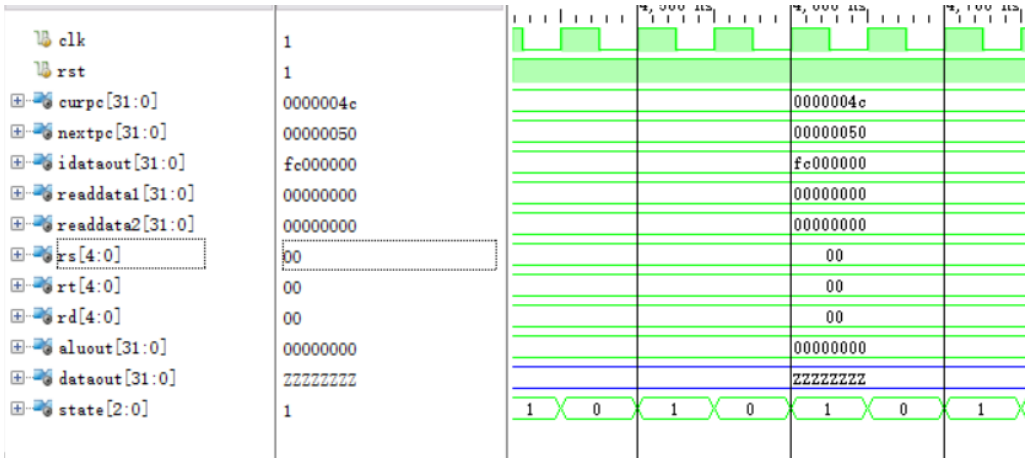
o) j 0x0000004C

NextPC 为 4C，只有 sIF(000)和 sID(001)阶段，执行正确。



p) halt

停机指令，PC 不改变，执行正确。



3. 烧板结果

单周期与多周期的区别主要在于多周期的状态划分;仿真与烧板的区别仅在于顶层模块的不同。我们在显示数据的选择上，增加状态的显示，以表达当前执行状态。将选择信号变成 3 位选择信号，增加 100 状态为状态显示。

```
3'b100: begin
    out[15:3] = 0;
    out[2:0] = State;
end
```

通过部分指令进行烧板的情况说明:

i. **addi $1, $0, 8**

sEXE(010)执行阶段，ALU 输出 8，1 号寄存器值仍为 0；sWB(011)阶段，计算结果写入寄存器，1 号寄存器值为 8。

CurPC：NextPC

State | Rs : ReadData1

State | ALUOut : DataOut

State | Rs : ReadData1

State | ALUOut : DataOut

ii. **jal 0x00000040**

取指阶段(sIF)，NextPC 为 20，译指阶段(sID)，NextPC 为 40，执行正确。

State | CurPC：NextPC

State | CurPC：NextPC

iii. **sw $2, 4($1)**

Rs 为 1，数据为 8；Rt 为 2，数据为 2。在写回阶段，DataOut 值为 2，执行正确。

CurPC：NextPC

Rs：ReadData1 | Rt：ReadData2

State | ALUOut：DataOut

iv. **j 0x0000004C**

取指阶段(sIF)，NextPC 为 40，译指阶段(sID)，NextPC 为 4C，执行正确。

State | CurPC：NextPC

State | CurPC：NextPC

v. **halt**

PC 不再变化，执行正确。

CurPC：NextPC

六. 实验心得

有了设计单周期 CPU 的经验，这次设计多周期 CPU 的工作流程就明确了很多：根据数据通路图，将 CPU 的各个模块分离设计，只关注模块本身的输入输出与逻辑功能。设计完成所有模块后，就是设计顶层文件，将模块整合在一起，并加入仿真文件进行测试。

许多模块与单周期有着共同之处，但 control unit 就大不相同了。多周期采取分状态执行指令的方法。我们分为了取指，译指，执行，访存，写回五个状态。根据状态转移图，构造出状态的转移代码，然后根据不同状态，给控制信号赋值。对于控制信号与状态的挂钩，一定要仔细考虑。一开始将跳转指令的 PCWre 放在取值阶段，而译指阶段没有，导致 PC 显示正确，但是并未发生跳转，就是没有及时写入 PC。

在寄存器组中，针对 0 号寄存器，进行了保护：写入数据时，会判断是否为 0 号寄存器，若是，则禁止写入。

第一次仿真的时候，前两条指令执行正确，但是后面的就错了，不过指令地址与指令二进制码都与测试文件中相同。模块与单周期基本相同，基本模块是不会出错的，所以重新检查了 control unit 的代码。发现问题如下：开始时，将控制信号按照状态赋值，采用 case 语句，但状态改变后，前面工作的过的模块的控制信号依旧是正常工作的信号，所以变成了类似流水线的执行方式，所有模块不停工作，但测试指令基本都是数据相关或名相关的指令，所以后面的结果就全乱了。于是更改为 if-else 语句进行赋值，在不同条件下值不同。成功运行。

对于仿真输出确定之后，进行每条指令的 debug。在跳转指令处出现了问题，NextPC 显示正确，但是并未跳转，依旧按照 PC + 4 执行。发现控制信号 PCWre 只在 sIF 阶段值为1。取指阶段为 1，而译指阶段为0，跳转指令只执行这两个阶段，如果不维持信号，

则无法将跳转地址送入 InstructionMemory。

烧板子的时候，需要注意每次按键是一个时钟周期，也就是一次状态转换。我们观察指令执行结果，需要对照当前状态，如在 sEXE(010) 执行阶段查看 ALU 的输出。