



《计算机组成原理实验》

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 计算机类 4 班

学生姓名 : 杨志成

学号 : 16337281

时间 : 2017 年 12 月 12 日

成 绩 :

实验二：单周期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: $rd \leftarrow rs | rt$

- (5) and rd, rs, rt

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

功能: $rd \leftarrow rs \& rt$

- (6) ori rt, rs, immediate

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

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

==>移位指令

- (7) sll rd, rt, sa

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

功能: $rd \leftarrow rt \ll (zero-extend)sa$, 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) slti rt, rs, immediate 带符号

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

功能: if ($rs < (sign-extend)immediate$) $rt = 1$ else $rt = 0$, 具体请看表 2 ALU 运算功能表, 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: $memory[rs + (sign-extend)immediate] \leftarrow rt$ 。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

(11) lw rt, immediate(rs)

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

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

==>分支指令

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

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

功能: if($rs = rt$) $pc \leftarrow pc + 4 + (sign-extend)immediate \ll 2$ else $pc \leftarrow pc + 4$

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

110101	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: if($rs \neq rt$) $pc \leftarrow pc + 4 + (sign-extend)immediate \ll 2$ else $pc \leftarrow pc + 4$

(14) bgtz rs,immediate

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

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

==>跳转指令

(15) j addr

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

功能: $pc \leftarrow \{pc+4\}[31:28],addr[27:2],0,0\}$, 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。

(16) 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**

(17) jal addr

111010	addr[27..2]
--------	-------------

功能: 调用子程序, pc $\leftarrow \{(pc+4)[31:28],addr[27:2],0,0\}$; \$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**

(18) halt (停机指令)

111111	00000000000000000000000000000000(26位)
--------	---------------------------------------

不改变 pc 的值, pc 保持不变。

三. 实验原理

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

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

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

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

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

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

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



图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式:

R 类型:

31	26 25	21 20	16 15	11 10	6 5	0
op	rs	rt	rd	sa	funct	

6 位 5 位 5 位 5 位 5 位 6 位

I 类型:

31	26 25	21 20	16 15	0
op	rs	rt	immediate	

6 位 5 位 5 位 16 位

J 类型:

31	26 25	0
op	address	

6 位 26 位

其中，

op: 为操作码；

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

rt: 为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd: 为目的操作数寄存器，寄存器地址（同上）；

sa: 为位移量（shift amt），移位指令用于指定移多少位；

funct: 为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

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

address: 为地址。

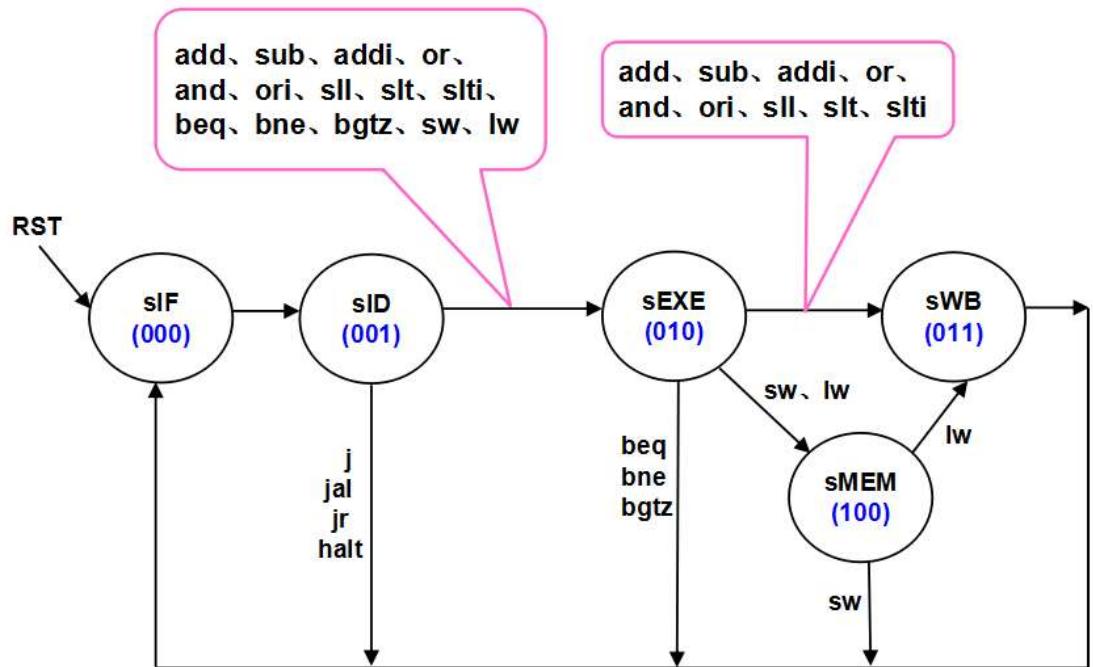


图 2 多周期 CPU 状态转移图

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

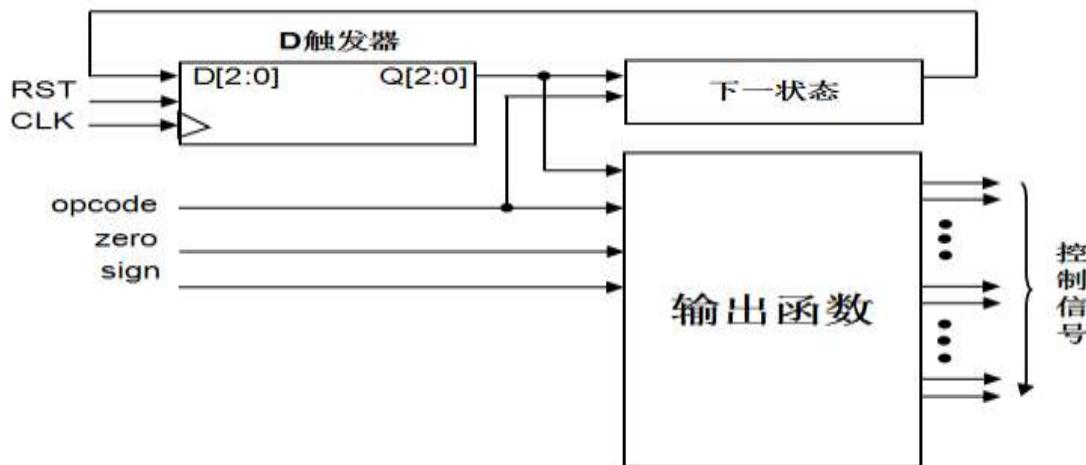


图 3 多周期 CPU 控制部件的原理结构图

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

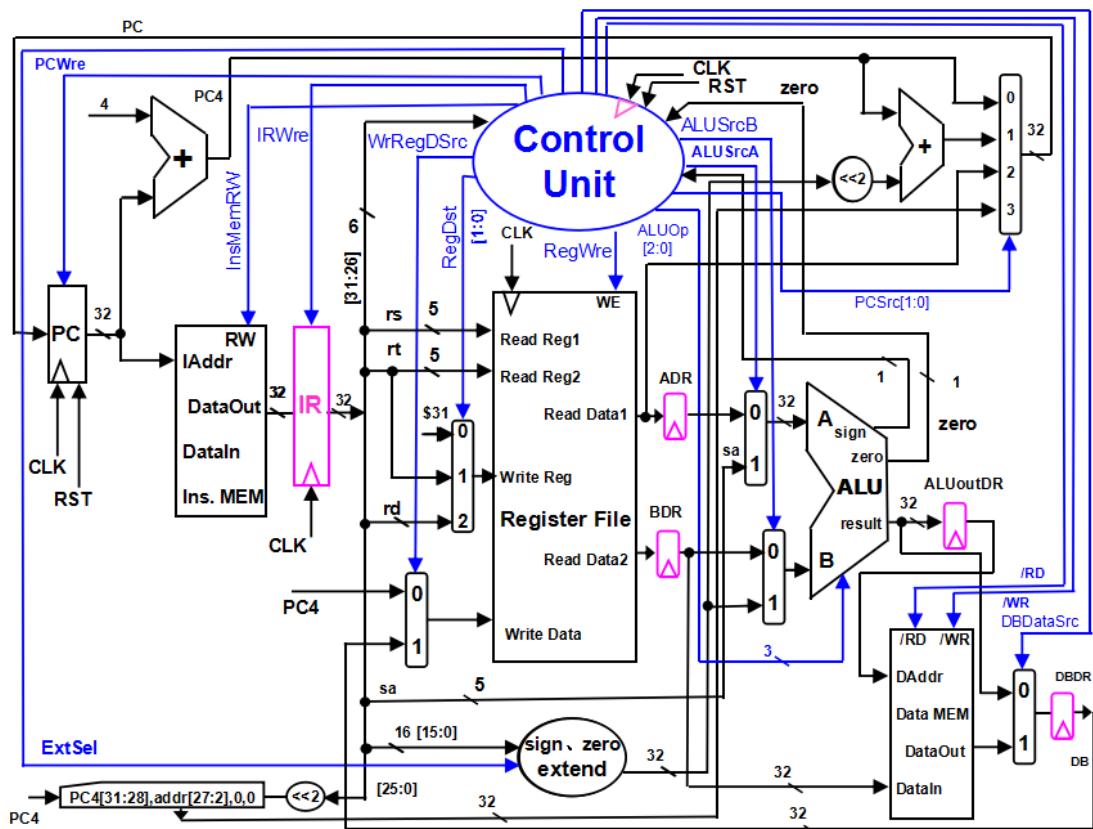


图 4 多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时，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、bne、bgtz、slt、slti、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 {27{0}},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、bne、bgtz、slt、sll	来自 sign 或 zero 扩展的立即数, 相关指令: addi、ori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出, 相关指令: add、sub、addi、or、and、ori、slt、slti、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、bne、bgtz、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、slti、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、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
/RD	读数据存储器, 相关指令: lw	存储器输出高阻态
/WR	写数据存储器, 相关指令: sw	无操作
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;	(sign-extend) immediate , 相关指令: addi、lw、sw、beq、bne、bgtz;
PCSrc[1..0]	00: pc<-pc+4, 相关指令: add、addi、sub、or、ori、and、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1, 或 zero=1); 01: pc<-pc+4+(sign-extend) immediate , 相关指令: beq(zero=1)、bne(zero=0)、bgtz(sign=0, 且 zero=0); 10: pc<-rs, 相关指令: jr; 11: pc<-[pc[31:28],addr[27:2],0,0}, 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、slti、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	<pre>if (A < B && (A[31] == B[31])) Y = 1; else if (A[31] == 1 && B[31] == 0) Y = 1; else Y = 0;</pre>	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

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) 实验设计

通过单周期 CPU 的实验，这一次多周期 CPU 设计就更为轻松一些，大部分模块可以沿用上一次实验的成果。

根据数据通路图，构建底层模块：PC 模块，指令存储器，数据存储器，寄存器组，ALU（算术逻辑单元），控制单元。

并还需要几个辅助操作部件：加法器，多选器，符号扩展单元，左移单元，以及数据延迟单元。

最后，构建顶层模块将底层模块串联起来，并在通过测试后，使得 CPU 能正常工作。

(2) 实验步骤

【1】控制单元设置：

根据各控制信号与相应指令之间的相互关系，就可以得出控制信号与指令之间的关系表，再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

Stage	Ins	Sign	Zero	PCWre	ALUSrc A	ALUSrc B	ExtSel[1..0]	DBData Src	RegWr e	WrReg Data	IRWre	PCSrc [1..0]	DataMemRW	RegDst [1..0]	InsMem RW	ALUOp [2..0]	
if 0	x	x	x	1	x	x	xx	x	0	x	1	xx	0	xx	1	xxx	
id 001	j	x	x	0	x	x	xx	x	1	x	0	11	0	xx	x	xxx	
	jal	x	x	0	x	x	xx	x	0	x	0	11	0	xx	x	xxx	
	jr	x	x	0	x	x	xx	x	0	x	0	10	0	xx	x	xxx	
	hlt	x	x	0	x	x	xx	x	0	x	0	xx	0	xx	x	xxx	
exe1 110	add	x	x	0	0	0	xx	x	0	x	0	xx	0	xx	x	000	
	sub	x	x	0	0	0	xx	x	0	x	0	xx	0	xx	x	001	
	addi	x	x	0	0	1	10	x	0	x	0	xx	0	xx	x	000	
	or	x	x	0	0	0	xx	x	0	x	0	xx	0	xx	x	011	
	and	x	x	0	0	0	xx	x	0	x	0	xx	0	xx	x	100	
	ori	x	x	0	0	1	01	x	0	x	0	xx	0	xx	x	011	
	sllt	x	x	0	0	0	xx	x	0	x	0	xx	0	xx	x	110	
	slli	x	x	0	0	1	10	x	0	x	0	xx	0	xx	x	110	
exe2 101	sll	x	x	0	1	1	00	x	0	x	0	xx	0	xx	x	010	
	beq	x	0	0	0	0	10	x	0	x	0	00	0	xx	x	001	
	beq	x	1	0	0	0	10	x	0	x	0	01	0	xx	x	001	
	bne	x	0	0	0	0	10	x	0	x	0	01	0	xx	x	001	
	bne	x	1	0	0	0	10	x	0	x	0	00	0	xx	x	001	
	bgtz	0	x	0	0	0	10	x	0	x	0	01	0	xx	x	110	
	bgtz	1	x	0	0	0	10	x	0	x	0	00	0	xx	x	110	
	sw	x	0	0	1	10	x	0	x	0	xx	0	xx	x	000		
exe3 010	lw	x	x	0	0	1	10	x	0	x	0	xx	0	xx	x	000	
	smem	x	x	0	x	x	10	x	0	x	0	0	1	xx	x	xxx	
	011	lw	x	0	x	x	10	x	0	x	0	xx	0	xx	x	xxx	
	wb1	add	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx
	sub	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx	
	addi	x	x	0	x	x	xx	0	1	1	0	0	0	01	x	xxx	
	or	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx	
	and	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx	
wb2 100	ori	x	x	0	x	x	xx	0	1	1	0	0	0	01	x	xxx	
	move	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx	
	sllt	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx	
	slli	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx	
	sll	x	x	0	x	x	xx	0	1	1	0	0	0	10	x	xxx	
	lw	x	x	0	x	x	xx	1	1	1	0	0	0	0	x		

【2】模块基本架构:

```

② Mul_Cycle_CPU (Mul_Cycle_CPU.v) (22)
  ③ _PC - PC (PC.v)
  ③ _Add_4 - Add (ADD.v)
  ③ _InsMem - InsMem (InsMem.v)
  ③ _CtrlUnit - CtrlUnit (CtrlUnit.v)
  ③ _IR - IR (IR.v)
  ③ _Mux1 - Mux_2_1 (Mux_2_1.v)
  ③ _Mux2 - Mux_3_1 (Mux_3_1.v)
  ③ _RegFile - RegFile (RegFile.v)
  ③ ADR - DataDelay (DataDelay.v)
  ③ EDR - DataDelay (DataDelay.v)
  ③ _SignalExtend - SignExtend (SignExtend.v)
  ③ _Mux3 - Mux_2_1 (Mux_2_1.v)
  ③ _Mux4 - Mux_2_1 (Mux_2_1.v)
  ③ _ALU - ALU (ALU.v)
  ③ ALUout - DataDelay (DataDelay.v)
  ③ _DataMem - DataMem (DataMem.v)
  ③ _Mux5 - Mux_2_1 (Mux_2_1.v)
  ③ ALUIMDRE - DataDelay (DataDelay.v)
  ③ _MoveL - MoveL (MoveL.v)
  ③ _Add_AB - Add (ADD.v)
  ③ _PCJump - PCJump (PCJump.v)
  ③ _Mux6 - Mux_4_1 (Mux_4_1.v)

```

【3】本次实验简单编译器构造

首先介绍正则表达式的概念：

正则表达式是对字符串（包括普通字符（例如，a 到 z 之间的字母）和特殊字符（称为“元字符”））操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。正则表达式是一种文本模式，模式描述在搜索文本时要匹配的一个或多个字符串。

首先将 MIPS 汇编指令变成统一的格式：

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
汇编指令			\$	数字	,	\$	数字	,	(\$)	数字						

这个样子就便于处理相应的字符串

按此格式化的 MIPS 指令为：

```

addi $1 ,$0 , 8
ori $2 ,$0 , 2
or $3 ,$2 ,$ 1
sub $4 ,$3 ,$ 1
and $5 ,$4 ,$ 2
sll $5 ,$5 , 2
beq $5 ,$1 , -2
jal OX00000048
slt $8 ,$12,$ 1
addi $14,$0 , -2
slt $9 ,$8 ,$14
slti $10,$9 , 2
slti $11,$10, 0
add $11,$11,$ 8
bne $11,$2 , -2
addi $2 ,$2 , -1
bgtz $2 ,-2
j OX00000054
sw $2 ,4($1)
lw $12.4($1)
jr $31
halt

```

接下来我们可以用 case 语句或者 if else 来实现正则表达式的功能：

```

if(opcode == "add ")
{
else if(opcode == "sub ")
{
else if(opcode == "addi")
{
else if(opcode == "or ")
{
else if(opcode == "and ")
{
else if(opcode == "ori ")
}

```

【4】多周期指令二进制代码

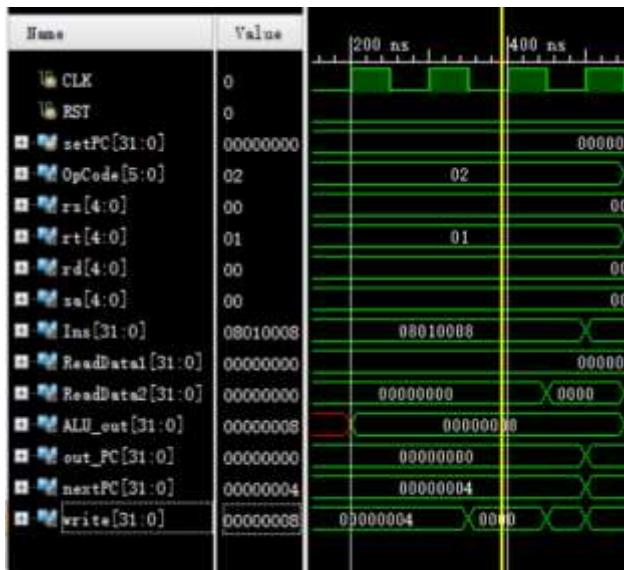
地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x000000000	addi \$1,\$0,8	000010	00000	00001	0000 0000 0000 1000	= 08010008	
0x000000004	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010		
0x000000008	or \$3,\$2,\$1	010000	00010	00001	0001 1000 0000 0000		
0x00000000C	sub \$4,\$3,\$1	000001	00011	00001	0010 0000 0000 0000		
0x000000010	and \$5,\$4,\$2	010001	00100	00010	0010 1000 0000 0000		
0x000000014	sll \$5,\$5,2	011000	00000	00101	0010 1000 1000 0000		
0x000000018	beq \$5,\$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110		
0x00000001C	jal 0x0000048	111010	00000	00000	0000 0000 0001 0010		
0x000000020	slt \$8,\$12,\$1	100110	01100	00001	0100 0000 0000 0000		
0x000000024	addi \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110		
0x000000028	slt \$9,\$8,\$14	100110	01000	01110	0100 1000 0000 0000		
0x00000002C	slti \$10,\$9,2	100111	01001	01010	0000 0000 0000 0010		
0x000000030	slti \$11,\$10,0	100111	01010	01011	0000 0000 0000 0000		
0x000000034	add \$11,\$11,\$8	000000	01011	01000	0101 1000 0000 0000		
0x000000038	bne \$11,\$2,-2(≠,转 34)	110101	01011	00010	1111 1111 1111 1110		
0x00000003C	addi \$2,\$2,-1	000010	00010	00010	1111 1111 1111 1111		
0x000000040	bgtz \$2,-2 (>0,转 3C)	110110	00010	00000	1111 1111 1111 1110		
0x000000044	j 0x0000054	111000	00000	00000	0000 0000 0001 0101		
0x000000048	sw \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100		
0x00000004C	lw \$12,4(\$1)	110001	00001	01100	0000 0000 0000 0100		
0x000000050	jr \$31	111001	11111	00000	0000 0000 0000 0000		
0x000000054	halt	111111	00000	00000	0000 0000 0000 0000	= FC000000	

(3) 实验结果

【1】 addi \$1,\$0,8

在 1 号寄存器内写入 8

波形图:



PC 指向 0x00000000

指令代码 0x08010008

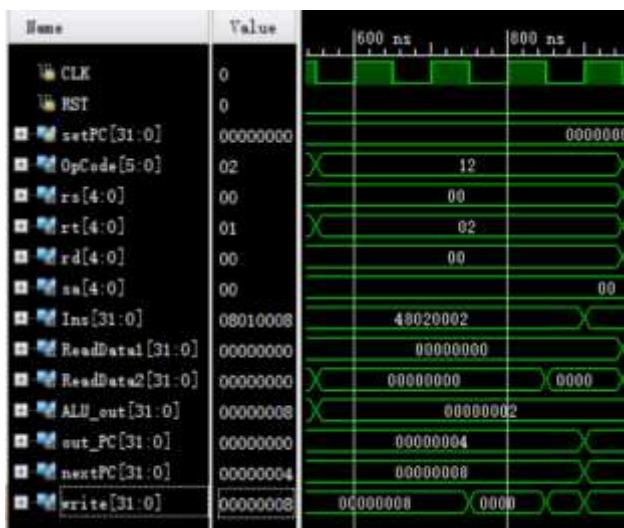
下一条指令地 0x00000004

State 运行一个周期, ALU 输出 8, 存入 1 号寄存器

【2】 ori \$2,\$0,2

在 2 号寄存器内写入 2

波形图:



PC 指向 0x00000004

指令代码 0x08010008

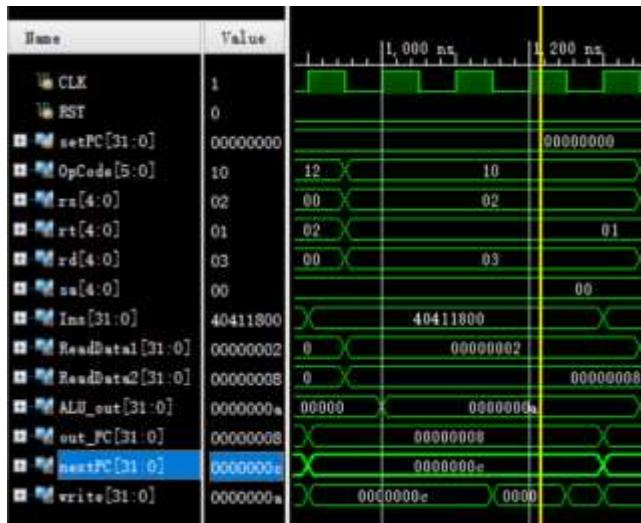
下一条指令地址: 0x00000008

State 运行一个周期, ALU 输出 2, 存入 2 号寄存器

【3】 or \$3,\$2,\$1

将寄存器 2 和寄存器 1 中的内容做或运算，写入寄存器 3

波形图：



PC 指向 0x00000008

指令代码 Ox 40411800

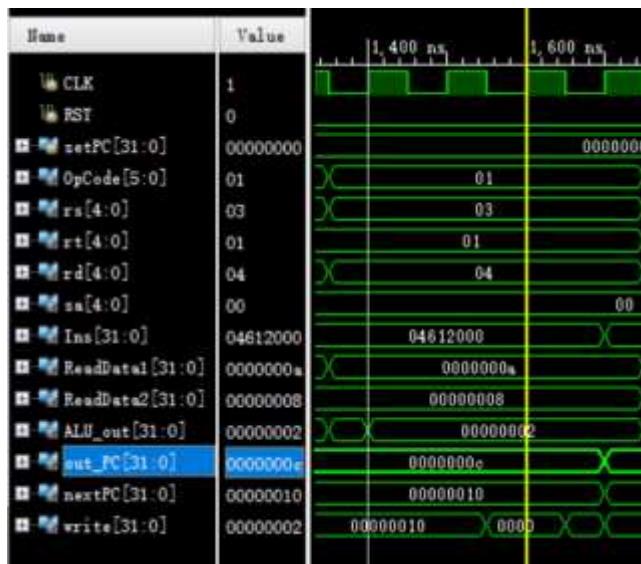
下一条指令地址：Ox0000000c

ALU 输出 0000000a，并将结果写入寄存器 3

【4】 sub \$4,\$3,\$1

寄存器 3 减寄存器 1，并将结果写入寄存器 4： $10 - 8 = 2$

波形图：



PC 指向 0x0000000c

指令代码：Ox04612000

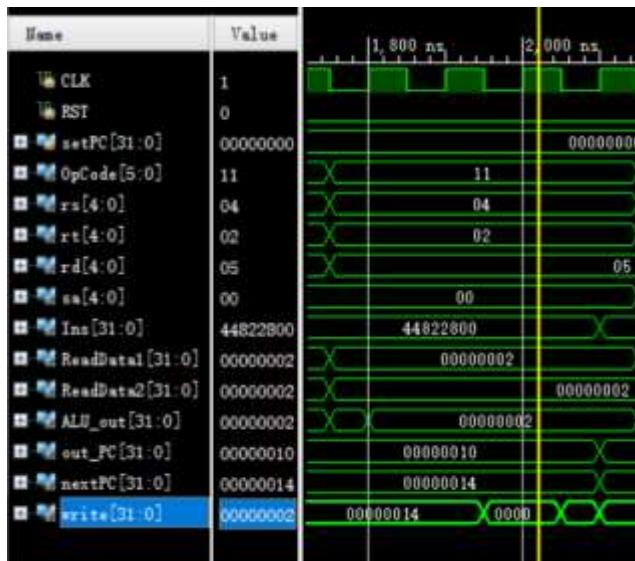
下一个指令地址：Ox00000010

State 运行一个周期，ALU 输出 00000002，并写入寄存器 4

【5】 and \$5,\$4,\$2

将寄存器 4 和寄存器 2 做与运算，并将结果存入寄存器 5，
 $2 \text{ and } 2 = 2$

波形图：



PC 指令 $0x00000010$

下一条指令地址： $0x00000014$

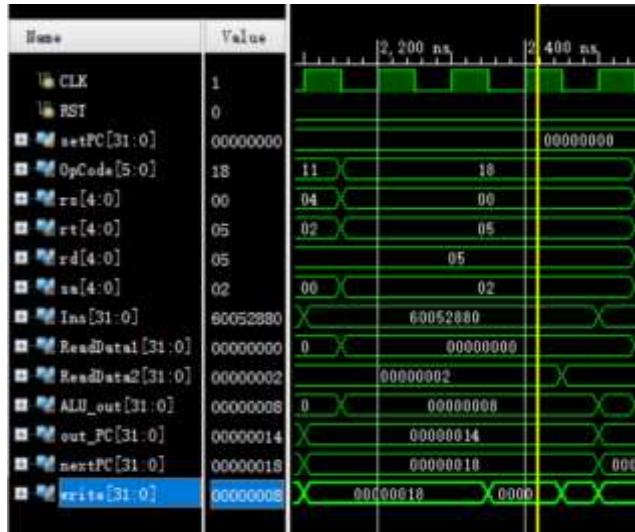
指令代码： $0x44822800$

State 运行一个周期，ALU 输出 00000002 ，并写入寄存器 5

【6】 sll \$5,\$5,2

将寄存器 5 的内容左移两位

波形图：



PC 指向 $0x00000014$

下一条指令地址： $0x00000018$

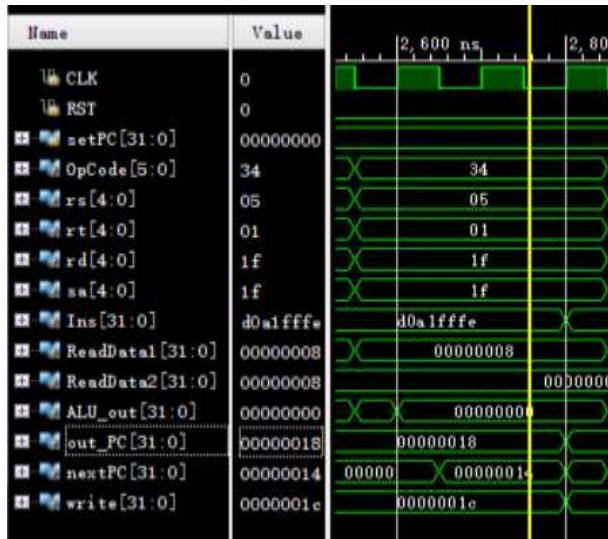
指令代码： $0x60052880$

State 运行一个周期，ALU 输出 00000008 ，并写入寄存器 5

【7】 beq \$5,\$1,-2

若寄存器 1 与寄存器 5 内容相等，则指令转向 PC-4

波形图：



PC 指向：Ox00000018

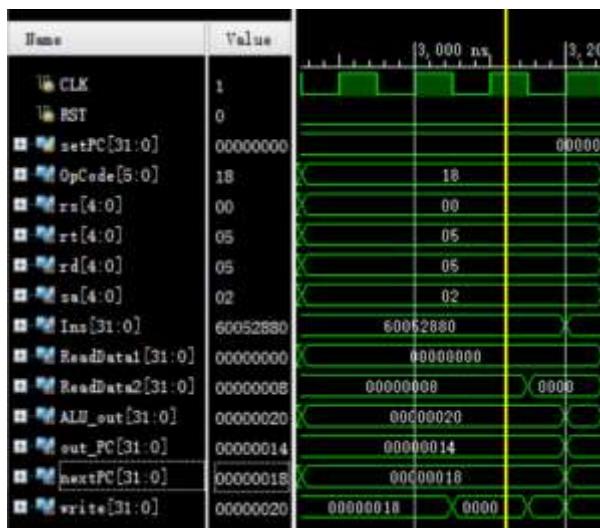
下一条指令地址：Ox00000014

State 运行一个周期，ALU 判断出指令需要跳转到 Ox00000014

【8】 sll \$5,\$5,2

将寄存器 5 的内容左移两位

波形图：



PC 指向：Ox00000014

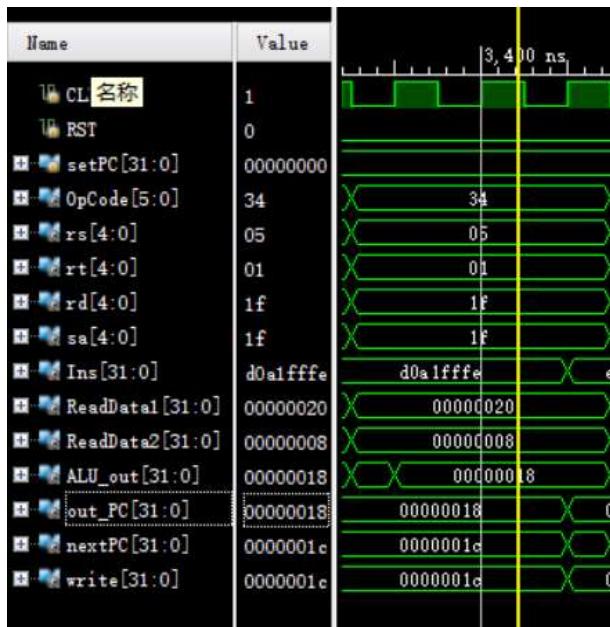
下一条指令地址：Ox00000018

State 运行一个周期，ALU 输出 32，并写入寄存器 5

【9】 beq \$5,\$1,-2

若寄存器 1 与寄存器 5 内容相等，则指令转向 PC-4

波形图：



PC 指向：0x00000018

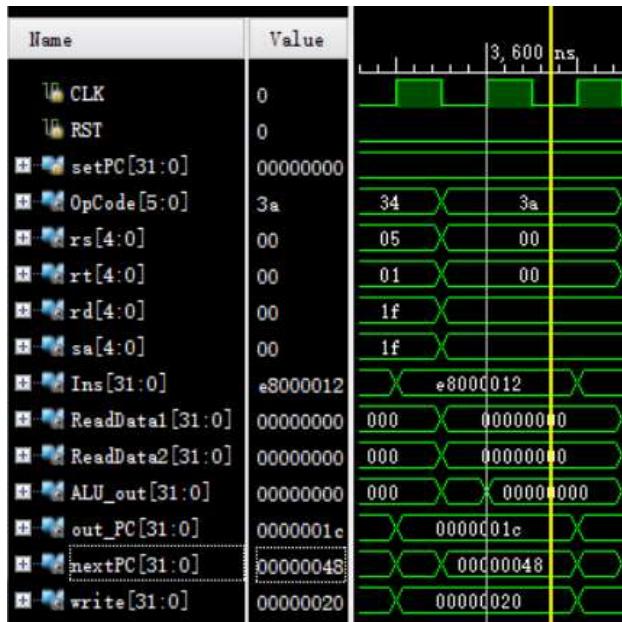
下一条指令地址：0x0000001c

State 运行一个周期，ALU 判断出指令不需要跳转

【10】 jal 0x00000048

指令跳转到 0x00000048，并将下一条指令地址 PC+4 写入寄存器 31

波形图：



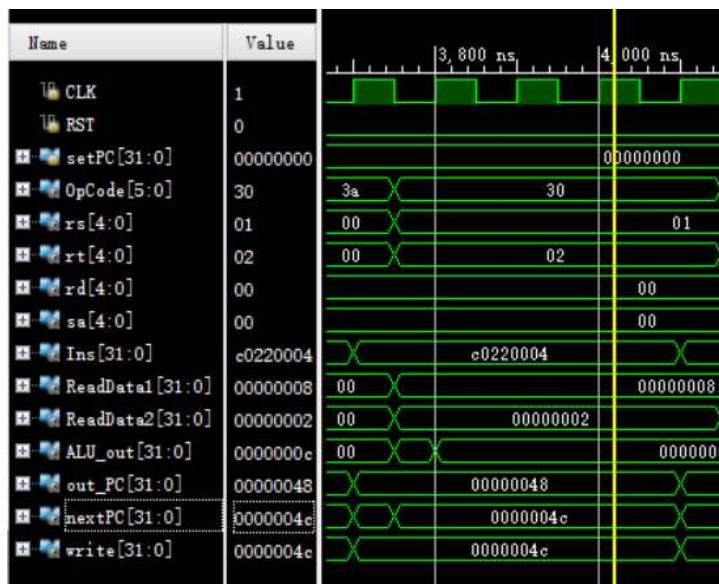
PC 指向 0x0000001c

下一条指令地址：0x00000048

State 与运行一个周期，指令跳转到 0x00000048，并将 0X00000020 写入寄存器 31

【11】 sw \$2,4(\$1)

将寄存器 2 的内容写入，[\$1 + 4]的内存单元中
波形图：



PC 指向 Ox00000048

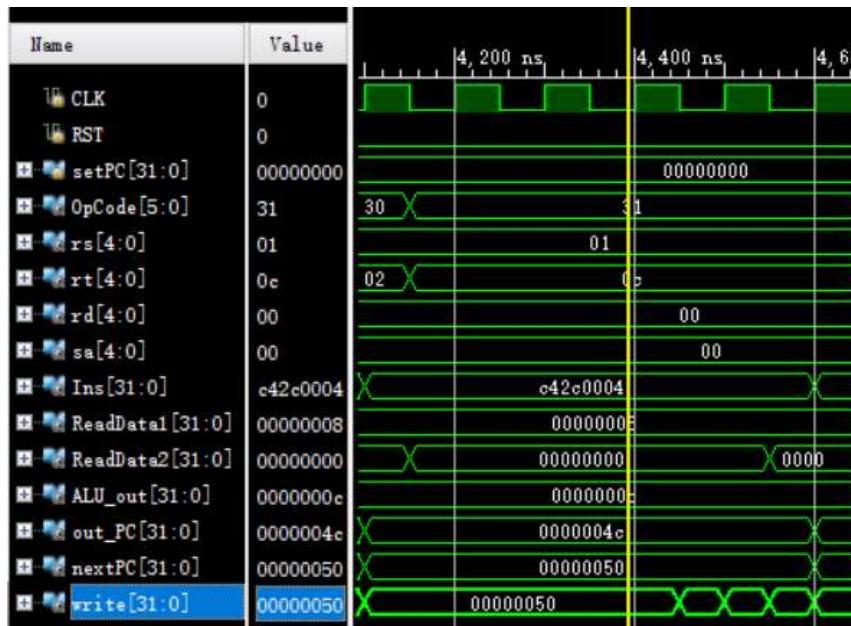
下一条指令地址：Ox0000004c

State 运行一个周期，12 号储存器的值为 2

【12】 lw \$12,4(\$1)

将内存单元[\$1 + 4]的内容写入寄存器 12

波形图：



PC指向Ox0000004c

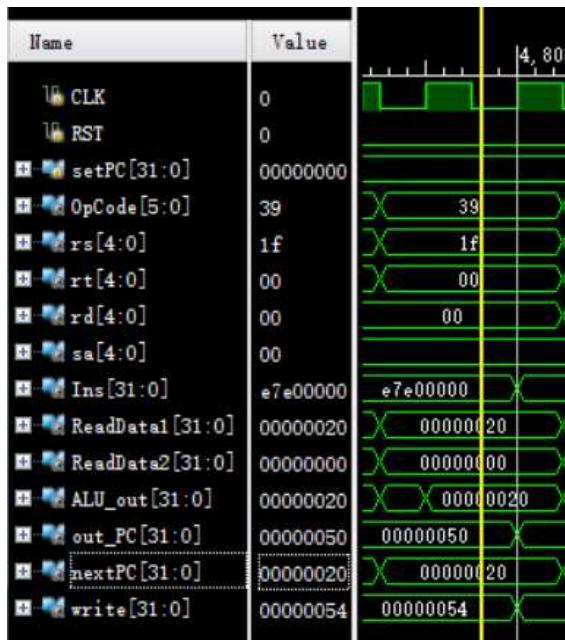
下一条指令地址：Ox00000050

State运行一个周期，将内存单元的内容2,写入寄存器12

【13】jr \$31

将指令跳转到寄存器31所指向的内存单元中

波形图：



PC指向Ox00000050

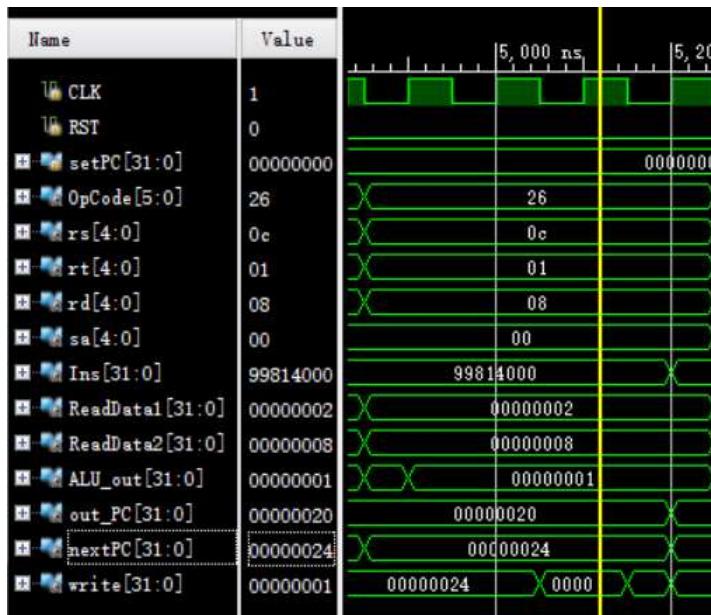
下一条指令地址：Ox000000020

State运行一个周期，指令跳转到Ox000000020

【14】slt \$8,\$12,\$1

比较寄存器12和寄存器1，若小于则将寄存器8置为1，否则为0

波形图：



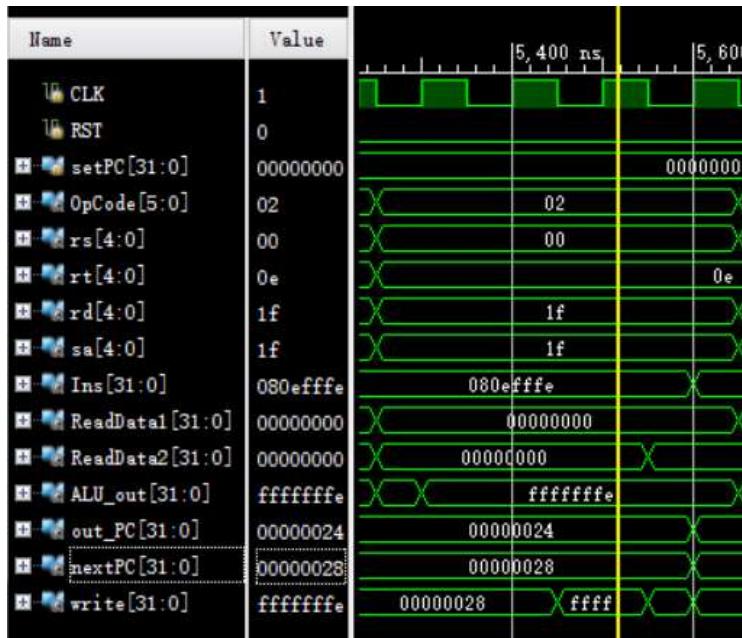
PC指向Ox000000020

下一条指令地址：Ox000000024

State运行一个周期，将ALU运行结果1写入寄存器8

【15】addi \$14,\$0,-2计算 $0 + (-2)$ 的值，并写入寄存器14

波形图：



PC指向0x00000024

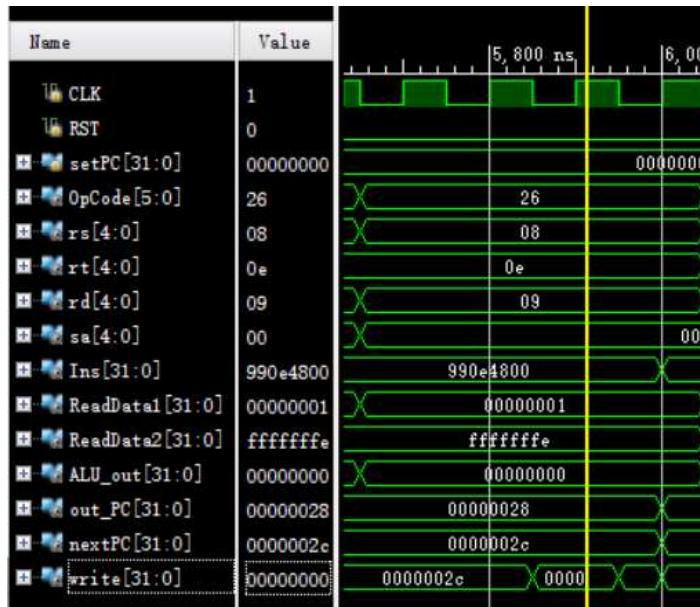
下一条指令地址：0x00000028

State运行一个周期，将ALU运行结果-2，写入寄存器14

【16】slt \$9,\$8,\$14

比较寄存器8和寄存器14的值，若小于则将寄存器9置为1，否则置为0

波形图：



PC指向0x00000028

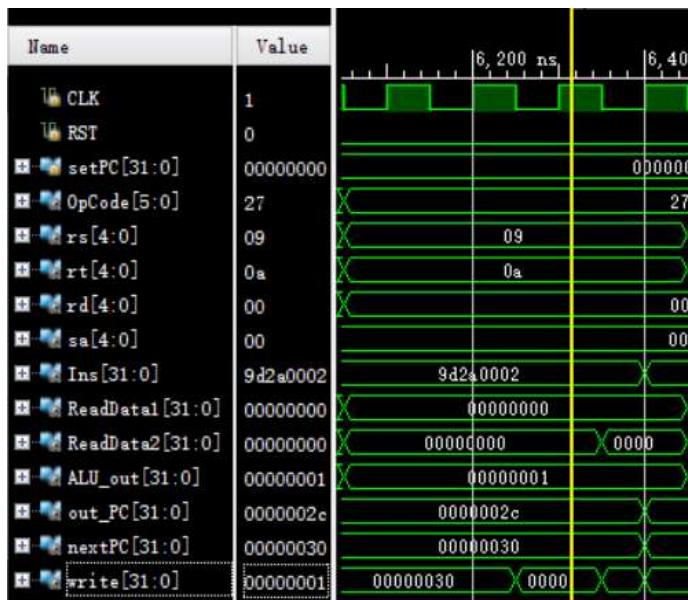
下一条指令地址：0x0000002c

State运行一个周期，将ALU运行结果0，写入寄存器9

【17】 slti \$10,\$9,2

比较寄存器 9 和 2 的值，若小于则将寄存器 9 置为 1，否则置为 0

波形图：



PC 指向 0x0000002c

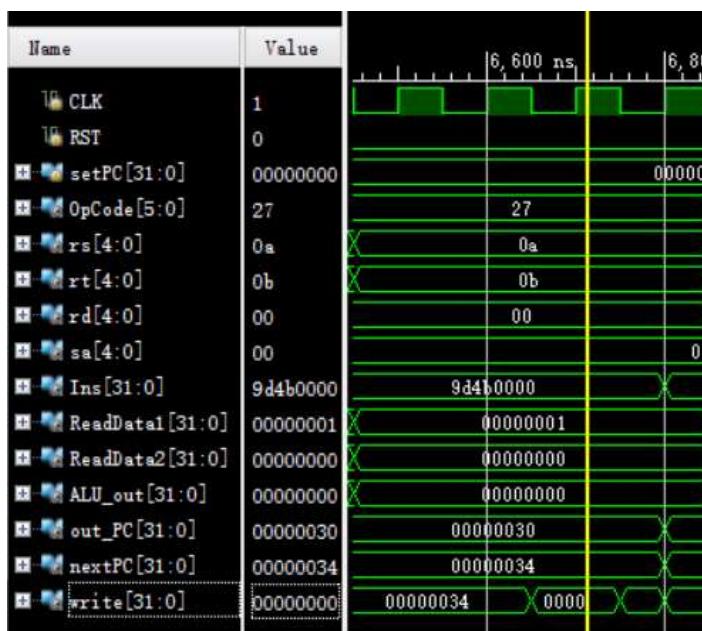
下一条指令地址 0x00000030

State 运行一个周期，将 ALU 运行结果 1，写入寄存器 10

【18】 slti \$11,\$10,0

比较寄存器 10 和 0 的值，若小于则将寄存器 9 置为 1，否则置为 0

波形图：



PC 指向 0x00000030

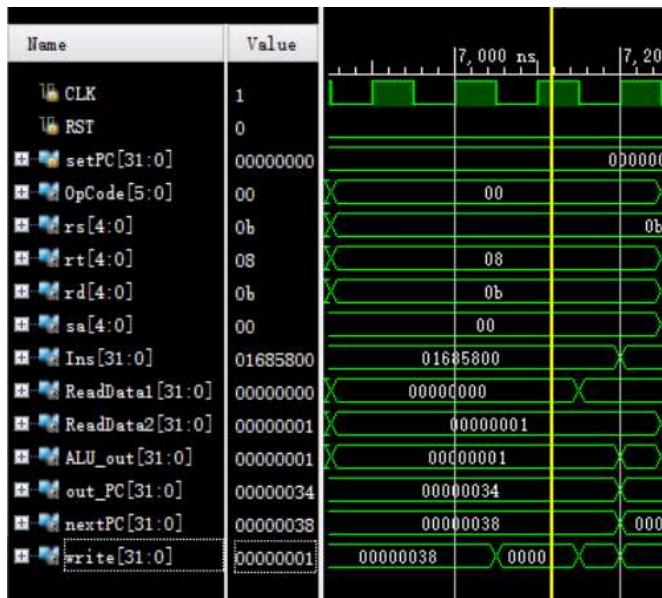
下一条指令地址：0x00000034

State 运行一个周期，将 ALU 运行结果 0，写入寄存器 11

【19】 add \$11,\$11,\$8

将寄存器 11 和寄存器 8 相加，并将结果写入寄存器 11

波形图：



PC 指向 Ox00000034

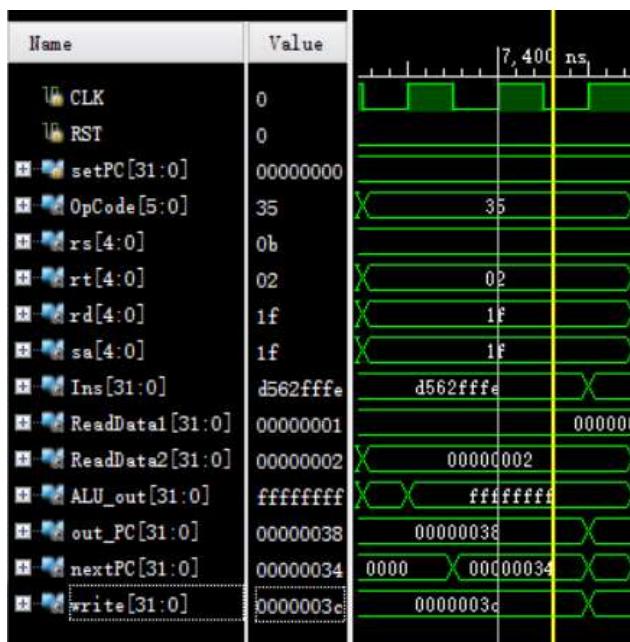
下一条指令地址：Ox00000038

State 运行一个周期，将 ALU 结果（ $0 + 1 = 1$ ），写入寄存器 11

【20】 bne \$11,\$2,-2

比较寄存器 11 和寄存器 2 的内容，若不相等则跳转至 PC-4，否则不跳转。

波形图：



PC 指向 Ox00000038

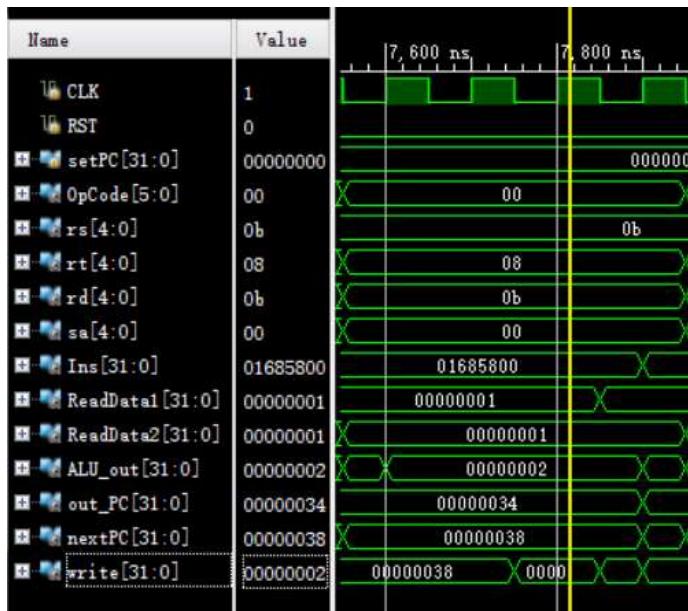
下一条指令地址：Ox00000034

State 运行一个周期，指令跳转至 Ox00000034

【21】 add \$11,\$11,\$8

将寄存器 11 和寄存器 8 相加，并将结果写入寄存器 11

波形图：



PC指向0x00000034

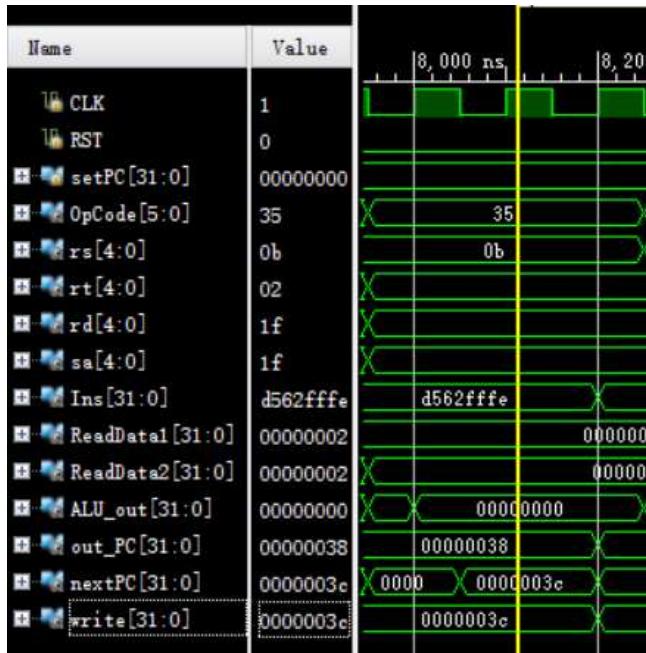
下一条指令地址：0x00000038

State运行一个周期，将ALU运行结果（1+1=）2,写入寄存器11

【22】 bne \$11,\$2,-2

比较寄存器 11 和寄存器 2 的内容，若不相等则跳转至 PC-4，否则不跳转。

波形图：



PC 指向 0x00000038

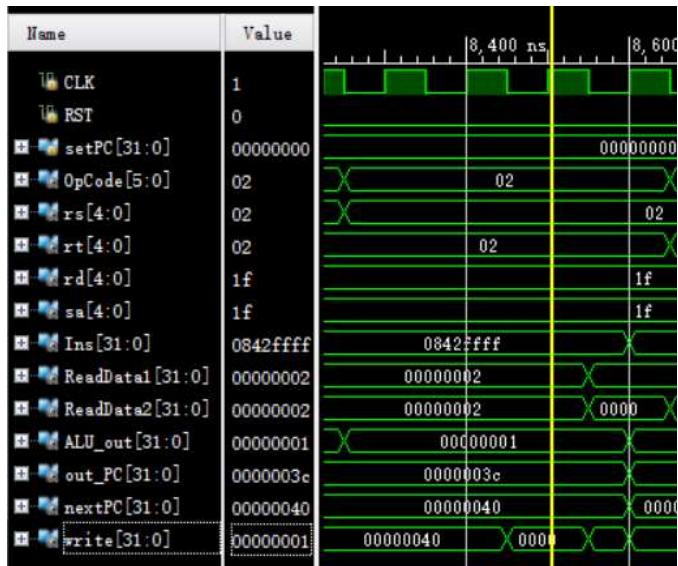
下一条指令地址：0x0000003c

State 运行一个周期，判断出指令不需要跳转

【23】 addi \$2,\$2,-1

将寄存器 2 的值-1

波形图:



PC 指向 0x00000003c

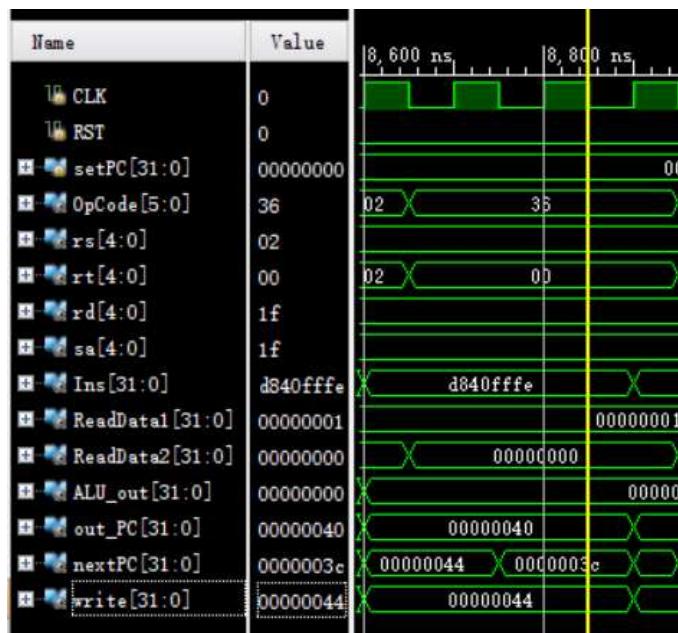
下一条指令地址 0x000000040

State 运行一个周期，将 ALU 运行结果 (2-1=) 1 写入寄存器 2

【24】 bgtz \$2,-2

若寄存器 2 的值大于 0，则指令跳转到 PC-4，否则，不跳转

波形图:



PC 指向 0x000000040

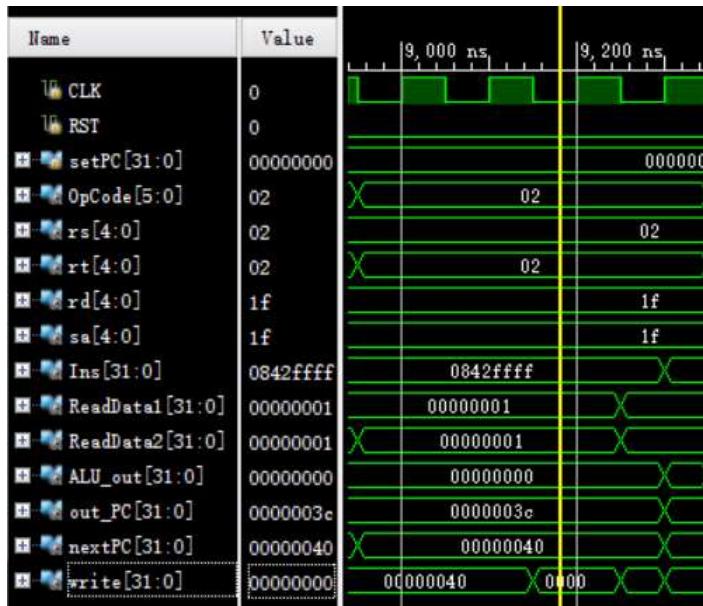
下一条指令地址: 0x00000003c

State 运行一个周期，指令跳转至 0x00000003c

【25】 addi \$2,\$2,-1

将寄存器 2 的值-1

波形图:



PC 指向 0x0000003c

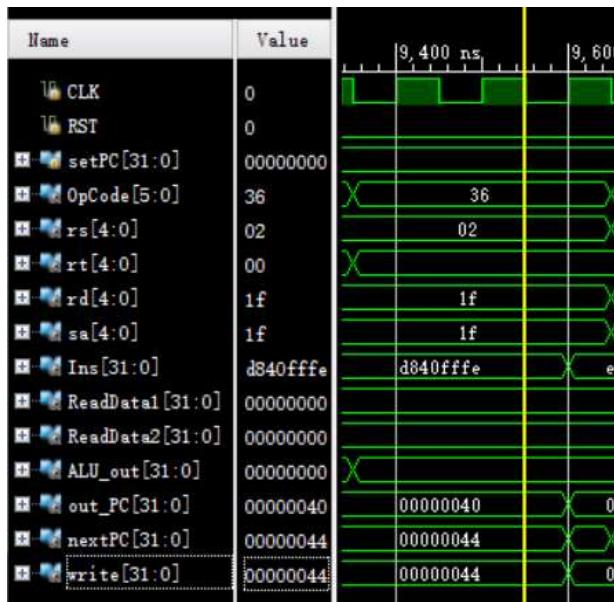
下一条指令地址: 0x00000040

State 运行一个周期, 将 ALU 运行结果 (1-1=) 0, 写入寄存器 2

【26】 bgtz \$2,-2

若寄存器 2 的值大于 0, 则指令跳转到 PC-4, 否则, 不跳转

波形图:



PC 指向 0x00000040

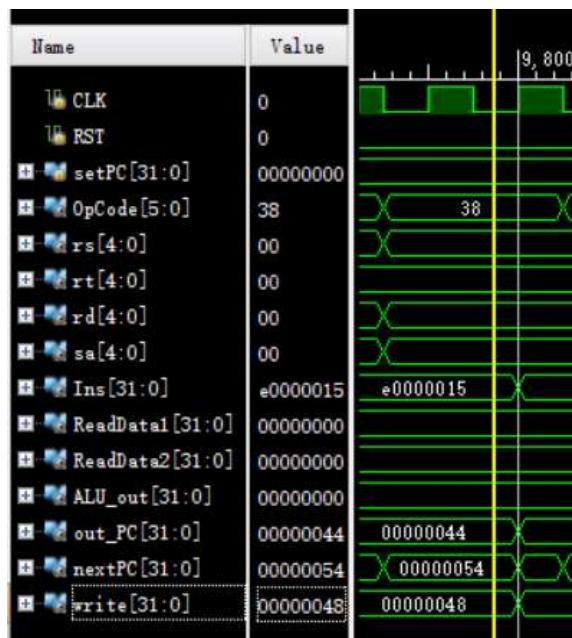
下一条指令地址: 0x00000044

State 运行一个周期, 判断出指令不需要跳转

【27】 j 0x00000054

指令跳转至 0x00000054

波形图:



PC 指向 0x00000044

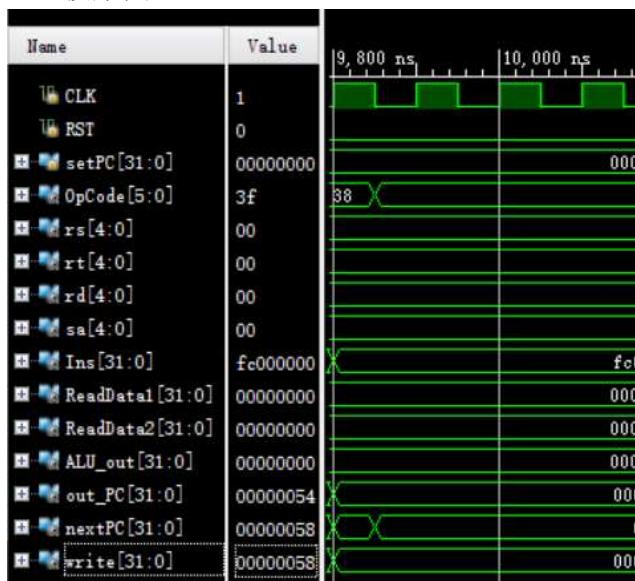
下一条指令地址 0x00000054

State 运行一个周期，指令跳转至 0x00000054

【28】 halt

停机指令

波形图:



State 运行一个周期后，停机

(4) verilog 代码分析:

【1】 controlunit:

```

85      always @(State or in_op ) begin
86          case(State)
87              IF:
88                  Next = ID;
89              ID: begin
90                  case (in_op[5:3])
91                      3'b111:           // j, jal, jr, halt
92                          Next = IF;
93                      3'b110: begin
94                          if (in_op == Beq || in_op == Bne || in_op == Bgtz)
95                              Next = EXE2;    // beq
96                          else
97                              Next = EXE3;    // sw, lw
98                      end
99                      default:         // add, sub, addi, or, and, ori, sll, slt, slti
100                         Next = EXE1;
101                     endcase
102                 end
103             EXE1:
104                 Next = WB1;
105             EXE2:
106                 Next = IF;
107             EXE3:
108                 Next = MEM;
109             MEM: begin
110                 if (in_op == Lw)
111                     Next = WB2;

```

此为状态转换部分，根据指令的 op 确定下一步的状态

注意 always 里触发条件为 State or in_op

```

122      always @(State or in_op) begin
123
124          if (State == IF && in_op != Halt)
125              PCWr = 1;
126          else
127              PCWr = 0;
128
129          if (in_op == Sll)
130              ALUSrcA = 1;
131          else
132              ALUSrcA = 0;
133
134          if (in_op == Addi || in_op == Slti || in_op == Ori || in_op == Lw || in_op == Ss)
135              ALUSrcB = 1;
136          else
137              ALUSrcB = 0;
138
139          if (State == EXE2)
140              ALUINCReg = 1;
141          else
142              ALUINCReg = 0;
143
144          if (State == WB1 || State == WB2 || in_op == jal)
145              RegWr = 1;
146          else
147              RegWr = 0;

```

此处根据 State 和 指令 op 确定信号

注意always里触发条件为 State or in_op

【2】delay模块

```

21 module DataDelay(CLK, in, out);
22     input wire CLK;
23     input wire [31:0] in;
24     output reg [31:0] out;
25
26     initial begin
27         out = 0;
28     end
29
30     always @(posedge CLK) begin
31         out <= in;
32     end
33
34 endmodule
35

```

根据时钟下降沿触发，将out值赋值为in

用该模块可组成ADR, BDR, ALUout模块

【3】IR模块

```

21 module IR(CLK, IRWr, in_IR, out_IR);
22     input wire CLK, IRWr;
23     input wire [31:0] in_IR;
24     output reg [31:0] out_IR;
25
26     always @ (negedge CLK) begin
27         if (IRWr)
28             out_IR = in_IR;
29     end
30
31 endmodule

```

注意：时钟下降沿触发

【4】四选一模块

```

21 module Mux_4_1(Ctrl, in_0, in_1, in_2, in_3, out);
22     input wire [1:0] Ctrl;
23     input wire [31:0] in_0, in_1, in_2, in_3;
24     output reg [31:0] out;
25
26     always @ (Ctrl or in_0 or in_1 or in_2 or in_3) begin
27         if (Ctrl == 2'b00)
28             out = in_0;
29         else if (Ctrl == 2'b01)
30             out = in_1;
31         else if (Ctrl == 2'b10)
32             out = in_2;
33         else if (Ctrl == 2'b11)
34             out = in_3;
35     end

```

根据ctrl的信号选择输出数据

【5】部分顶层模块

```

39      always @(*) begin
40          OpCode = DIIns[31:26];
41          rs = DIIns[25:21];
42          rt = DIIns[20:16];
43          rd = DIIns[15:11];
44          sa = DIIns[10:6];
45          ExtSa = { {27{1'b0}}, DIIns[10:6] };
46          nextPC = in_PC;
47          write = TheWriteData;
48      end

```

注意此处的always触发条件应该为*
不然容易发生延迟问题，这是一个比较重要的点

【6】InsMem

```

21 module InsMem(InsMemRW, in_IM, out_IM);
22     input wire InsMemRW;
23     input wire [31:0] in_IM;
24     output reg [31:0] out_IM;
25     reg [7:0] InsData [0:127];
26
27 initial begin
28     out_IM = 0;
29     $readmemb("C:/Users/DELL/Desktop/ECOP-16337281-03/binarycode.txt", InsData);
30 end
31
32 always @ (InsMemRW or in_IM) begin
33     out_IM[31:24] = InsData[in_IM + 0 - 32'h00000000];
34     out_IM[23:16] = InsData[in_IM + 1 - 32'h00000000];
35     out_IM[15:8] = InsData[in_IM + 2 - 32'h00000000];
36     out_IM[7:0] = InsData[in_IM + 3 - 32'h00000000];
37 end

```

第29行readmemb函数读取二进制代码
由control_unit单元的控制信号InsMemRW决定是读还是写

【7】ALU

```

1 module ALU(A, B, ALUOp, zero, sign, result);
2   input [31:0] A, B;
3   input [2:0] ALUOp;
4   output zero;
5   output sign;
6   output reg [31:0] result;
7   initial begin
8     result = 0;
9   end
10  assign zero = (result==0);
11  assign sign = (result>0)?1:0;
12  always @(A or B or ALUOp) begin
13    case(ALUOp)
14      3'b000: result <= A + B;
15      3'b001: result <= A - B;
16      3'b010: result <= B << A;
17      3'b011: result <= A | B;
18      3'b100: result <= A & B;
19      3'b101: result <= (A<B)?1:0;
20      3'b110: begin // 带符号比较
21        if (A<B && (A[31] == B[31]))
22          result <= 1;
23        else if (A[31] == 1 && B[31]==0)
24          result <= 1;
25        else result <= 0;
26      end
27      3'b111: result <= (A & ~B) | (~A & B);
28      default: result <= 0;
29    endcase
30  end
31 endmodule

```

ALU的功能由ALUOp决定，该代码主要由case语句实现

ALU功能表在先前已经给出，ALUOp由控制单元生成

(5) 在basys3板上烧制本次多周期CPU

我给上述代码添加了一些新的模块，如按键消抖，数码管显示（具体代码请见附录），其中用按键代替了时钟信号。具体实现情况如下：

开关说明：（以下数据都来自CPU）（SW15、SW14、SW0为Basys3板上开关名；BTNR为按键名）

开关SW_in (SW15、SW14)状态情况如下。显示格式：左边两位数码管BB : 右边两位数码管BB。以下是数码管的显示内容。

SW_in = 00: 显示当前 PC值:下条指令PC值

SW_in = 01: 显示 RS寄存器地址:RS寄存器数据

SW_in = 10: 显示 RT寄存器地址:RT寄存器数据

SW_in = 11: 显示 ALU结果输出 :DB总线数据。

复位信号 (reset) 接开关SW0，按键 (单脉冲) 接按键BTNR。

第一条指令：

addi \$1,\$0,8

在 1 号寄存器内写入 8

在第一条指令运行结束后，跳转到第二条指令时

Sw_in=00:

Sw_in=01



Sw_in=10:

Sw_in=11:

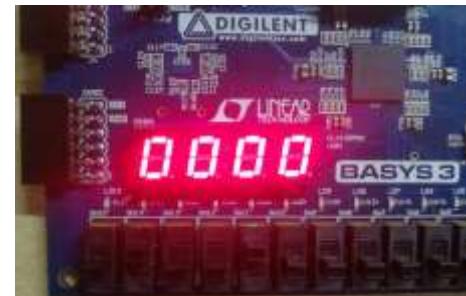
**第二条指令：**

ori \$2,\$0,2

在 2 号寄存器内写入 2

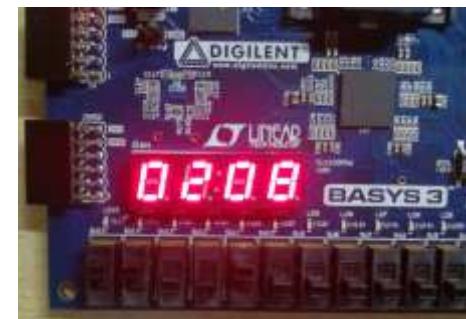
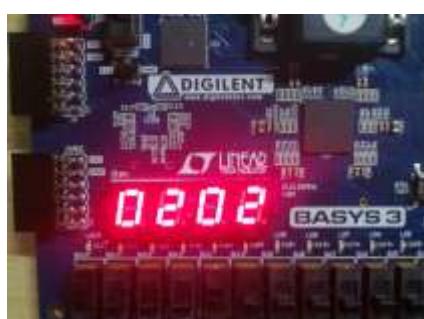
Sw_in=00:

Sw_in=01:



Sw_in=10:

Sw_in=11:



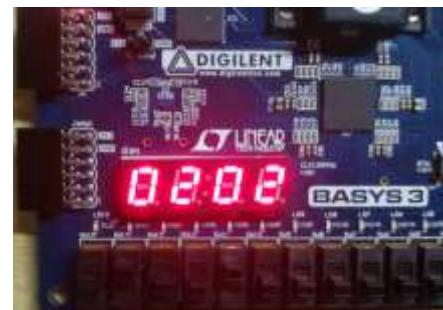
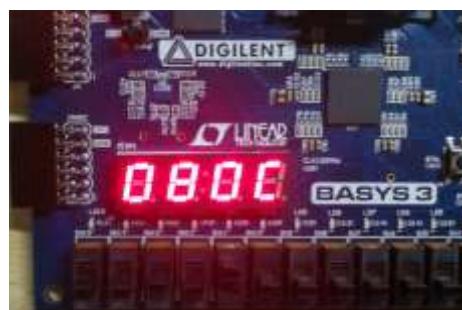
第三条指令:

or \$3,\$2,\$1

将寄存器 2 和寄存器 1 中的内容做或运算，写入寄存器 3

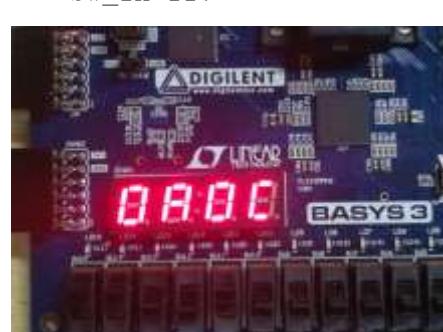
Sw_in=00:

Sw_in==01:



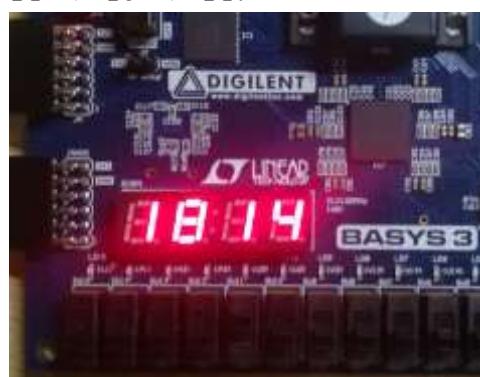
Sw_in=10:

Sw_in=11:

**指令跳转过程:**

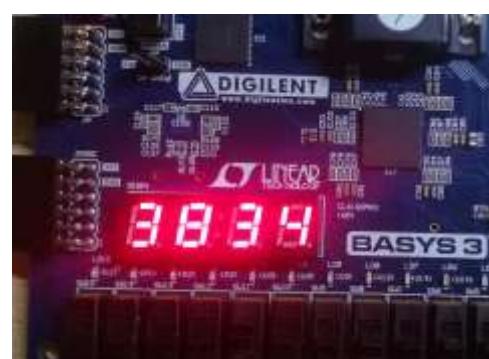
14 -> 18 -> 14:

1C->48

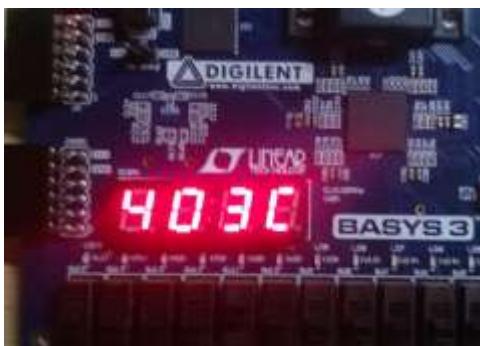


50->10:

38->34:



40->3C:



44->54



停机:



六. 实验心得

多周期 CPU 设计的一些心得

有了单周期 CPU 的铺垫，多周期 CPU 做起来比之前有底气了许多。

本次多周期实验，与上几周的单周期实验有很多共同的地方，比如寄存器组，以及基本数据通路图大致相同。而主要的不同在于，多周期的控制单元写法与单周期的写法大不相同，当时单周期我采用的是 case 把每种情况都列出来，而这一次多了很多的情况，因此尝试了使用逻辑表达式，效果也是极好的，可读性也更高。而各种状态码决定不同的操作，这一点相信在流水线设计中也会有更多的涉及。

虽然在 BUG 调试方面仍然花了很长的时间，但是对于一些常见的 BUG 能做出较快的反应，而不像之前一样一步一步慢慢调试，

以下是我遇到的一部分问题：

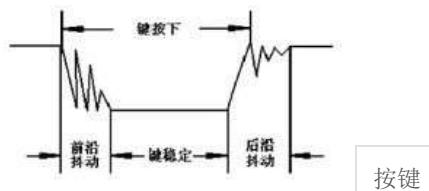
1. 比如出现高阻态，一般就是顶层模块连接出了问题，不是没有声明变量，就是名称打错了。

2. 多周期 CPU 的难点在于控制单元的设计，因为分成了许多阶段，真值表规模变大，代码长度也随之变大，很容易出现打错，打漏的情况，这也导致我在调试的时候花费了很多时间。

3. 另外一点就是时钟触发的问题，我的代码一开始是按要求，除了 PC 为上升沿触发，其余均为下降沿触发，但是在执行 lw 指令的时候，出现冲突。将 ADR、BDR 和 ALUM2DR 改为上升沿触发就解决了问题，对于这一类问题，主要还是要把各种指令在多周期 CPU 中运行的情况了然于胸，才能有底气的去 debug，不然只能是茫然地盯着代码不知所措。

4. 烧板时我发现当按下一次按钮，结果时钟跳转了多个周期，这时我便意识到，我的按键防抖模块没有写好，我通过百度查询到：

抖动时间的长短由按键的机械特性决定，一般为 5ms~10ms。这是一个很重要的时间参数，在很多场合都要用到。



按键稳定闭合时间的长短则是由操作人员的按键动作决定的，一般为零点几秒至数秒。键抖动会引起一次按键被误读多次。为确保 CPU 对键的一次闭合仅作一次处理，必须去除键抖动。

按键防抖代码：

```

1  `timescale 1ns / 1ps
2  module Light( in_key, out_key, clk);
3
4    input in_key, clk;
5    output out_key;
6    reg delay1, delay2, delay3;
7    always@( posedge clk )//CLR_SDM
8      begin
9        delay1 <= in_key;
10       delay2 <= delay1;
11       delay3 <= delay2;
12     end
13     assign out_key = delay1&delay2&delay3;
14   endmodule

```

因为只有当连续三个时钟周期按键电平都未改变的情况下，才可以认为此时按键稳定，这一个思想比较巧妙。

5. 一开始在 `controlUnit` 模块没有使用状态转移的转换代码，因此 `controlUnit` 写的非常紊乱，后来重写 `controlUnit` 后，用 `000,001…` 来充当一个指令周期的各个阶段，使得整个 `controlUnit` 模块看起来比以前更加有条理，`debug` 时也更好看出问题所在。可见代码的整洁对一个项目而言有多大的意义。

这学期计算机组成原理实验课的体会

这学期的三次实验都非常有意义，在理论课上，我们或许能够了解到汇编语言，了解到 CPU 的操作步骤，但是只有真正开始实践了你才会知道你自己掌握知识的程度。无论是 MIPS 汇编语言程序设计还是 CPU 的设计，我们都得从零开始学习很多新的知识，而这些知识大多是在课堂之外学到了，所以这门实验课大大地提高了我们的自学能力。

MIPS 汇编语言没有接触过，因此就会产生一种无从下手的感觉，也让我知道，知识不能仅仅局限于课本，这样才能学到真正有用的知识。CPU 的设计也是如此，因为 Verilog 语言以前基本上没有接触，常常就会有一种无所适从的感觉，尤其是在单周期 CPU 设计的时候。所以实验课还要大胆尝试，不会写代码的时候也要硬着头皮把自己的想法写出来，错了还可以再改，而不应该原地踏步不付诸实践，让时间白白流逝，自己也没学到真正的东西。到了多周期的时候，情况就好转很多，虽然仍然有许多地方会迷茫。我想这与代码的熟练度有关系，就像我们刚学 C++ 的时候，从不敢打代码，到老出莫名其妙的 BUG 一样，只要练习量代码量上去了，一些问题自然也就迎刃而解，不攻自破。

此外在本学期试验中，我也学会了遇到问题要善于与同学进行交流，对于同一个问题，每个人的见解和思路是不一样的，在日常的学习中，交流是必不可少的。与同学交流，可以不断提高我们的对知识掌握的水平。这学期的机组实验课让我认识到了这点。

CPU 十分精巧复杂，任何一个环节出错，带来的结果是灾难性的，这次试验当中我遇到了不少的问题，都是一些细微的错误，比如在顶层模块中一不小心把线连错，等等。错误看似不小，但是对结果所造成的影响，确很大。

CPU 的设计是需要很细心的，因为很小的一点错误就得花上大量的时间来查错修改，所以这对我们的细心程度也是一种考验。也正因为有很多 BUG 要调，同时也考验了我们的耐心。总的来说，这次的实验课的收获很大，首先巩固了我们在理论课上所学习的知识，对 CPU 各种指令的执行过程都有了进一步的理解，看着自己设计出来的 CPU，虽然过程坎坷，但是却很有成就感。

没有接触过计算机或者对计算机不是特别了解的人可能觉得计算机特别神秘而且不知道为什么它可以实现那么复杂的功能，而就我而言越是深入学习越是渴望了解其工作原理。自己以后也会继续这个样子保持对新的事物的好奇，保持对计算机知识的渴望。

路漫漫其修远兮，吾将上下而求索。