



《计算机组成原理实验》

实验报告

单周期 CPU

(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 3 班

学生姓名 : 王永锋

学号 : 16337237

时间 : 2017 年 12 月 14 日

目录

目录	2
一、 实验目的	4
二、 实验内容	4
三、 实验原理	6
四、 实验器材	11
五、 硬件设计实验过程	11
1. 设计数据通路图, 确定所需实现子模块	11
2. 确定控制单元信号表, 编写控制单元模块	13
3. 设计 ALU 模块并确定功能	14
4. 设计 Data_Memory 模块	14
5. PC 的设计	15
6. 编写 CPU_top 模块并使用代码测试	17
六、 实验结果 : 仿真波形图解	18
1. addi \$1, \$0, 8	18
2. ori \$2,\$0,2	18
3. . or \$3,\$2,\$1	19
4. sub \$4,\$3,\$1	19
5. and \$5,\$4,\$2	20
6. sll \$5,\$5,2	20
7. beq \$5,\$1,-2(=, 转 14)	21
8. sll \$5,\$5,2	21
9. beq \$5,\$1,-2(!=, 不跳)	22
10. jal 0x0000048	23
11. sw \$2,4(\$1)	24
12. lw \$12,4(\$1)	24
13. jr \$31	25
14. slt \$8,\$12,\$1	26
15. addi \$14,\$0,-2	26
16. slt \$9,\$8,\$14	27
17. slti \$10,\$9,2	27
18. slti \$11,\$10,0	28
19. add \$11,\$11,\$8	28

20.	bne \$11,\$2,-2 (\neq , 转 34).....	29
21.	add \$11,\$11,\$8	29
22.	bne \$11,\$2,-2 ($=$, 不转).....	30
23.	addi \$2,\$2,-1.....	31
24.	bgtz \$2,-2 (>0 , 转 3C).....	31
25.	addi \$2,\$2,-1.....	32
26.	bgtz \$2,-2 (>0 , 转 3C).....	32
27.	j 0x0000054.....	33
28.	halt	33
七、	在 Basy3 实验板上显示.....	34
1.	按键消抖模块	34
2.	编写 top 模块, 实例化 CPU 并显示信号.....	35
八、	实验结果 : basys3 板上运行 CPU	36
1.	以下图片的说明.....	36
2.	jal 0x0000048w \$9,4(\$1).....	37
3.	sw \$2,4(\$1)	38
4.	lw \$12,4(\$1).....	39
5.	slt \$8,\$12,\$1	41
九、	实验心得	41
1.	项目编写心得	41
2.	曾经遇到过的问题	42
2.1.	已解决 : 寄存器的写入 : 在下降沿写入还是上升沿写入 ?	42
2.2.	关于多周期 CPU 的时序控制.....	42
2.3.	质疑 : 关于 jal 指令的设计	43
2.4.	Bug : 烧板后寄存器无法被写入	44
十、	期末总结	45

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: $rd \leftarrow rt \ll (zero\text{-extend})sa$, 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 ($rs < (sign\text{-extend})immediate$) $rt = 1$ 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: $memory[rs + (sign\text{-extend})immediate] \leftarrow rt$ 。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: $rt \leftarrow memory[rs + (sign\text{-extend})immediate]$ 。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($rs = rt$) $pc \leftarrow pc + 4 + (sign\text{-extend})immediate \ll 2$ 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($rs \neq rt$) $pc \leftarrow pc + 4 + (sign\text{-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

(14) bgtz rs, immediate

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

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

在本文档中, 提供的相关内容对于设计可能不足或甚至有错误, 希望同学们在设计过程中如发现有问题, 请你们自行改正, 进一步补充、完善。谢谢!

三、 实验原理

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

- (1) 取指令(IF): 根据程序计数器 pc 中的指令地址, 从存储器中取出一条指令, 同时, pc 根据指令字长度自动递增产生下一条指令所需要的指令地址, 但遇到“地址转移”指令时, 则控制器把“转移地址”送入 pc, 当然得到的“地址”需要做些变换才送入 pc。
- (2) 指令译码(ID): 对取指令操作中得到的指令进行分析并译码, 确定这条指令需要完成的操作, 从而产生相应的操作控制信号, 用于驱动执行状态中的各种操作。
- (3) 指令执行(EXE): 根据指令译码得到的操作控制信号, 具体地执行指令动作, 然后转移到结果写回状态。
- (4) 存储器访问(MEM): 所有需要访问存储器的操作都将在这个步骤中执行, 该步骤给出存储器的数据地址, 把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
- (5) 结果写回(WB): 指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

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



图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

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

I 类型：

31	26 25	21 20	16 15	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型：

31	26 25	0
op	address	
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。

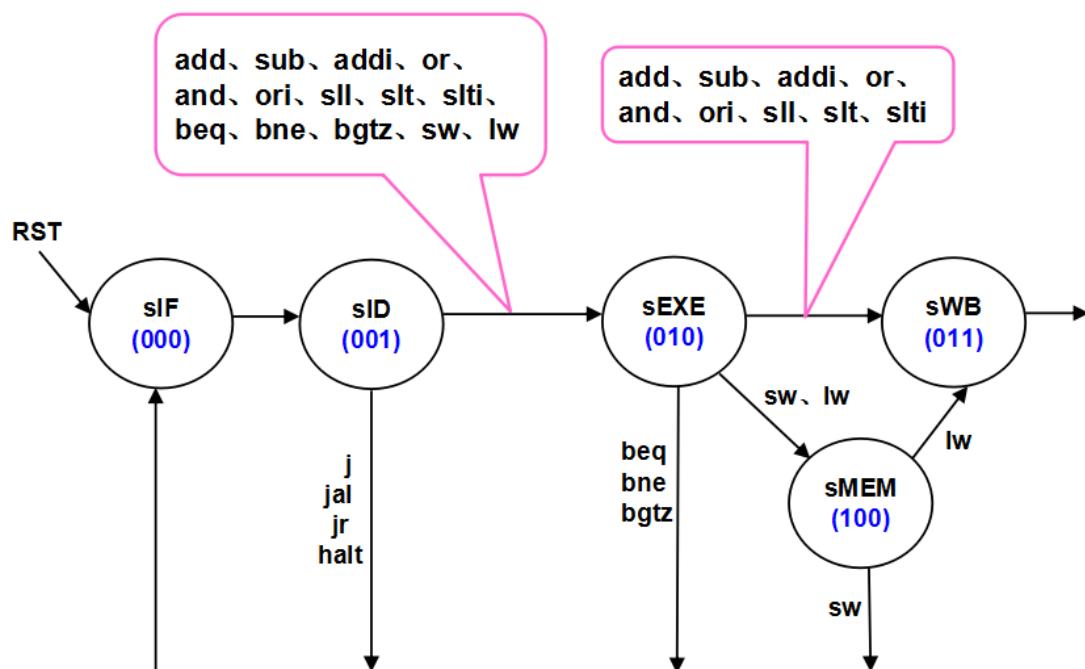


图 2 多周期 CPU 状态转移图

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。

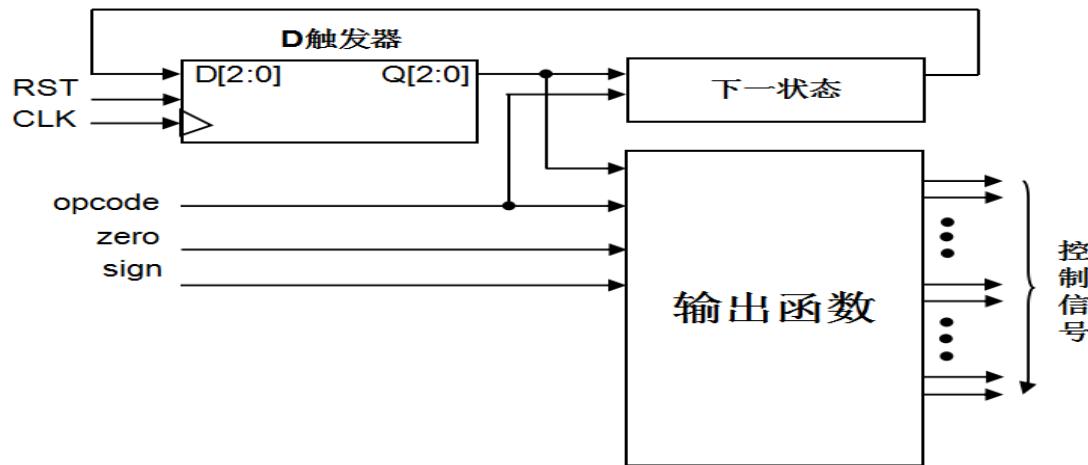


图 3 多周期 CPU 控制部件的原理结构图

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。

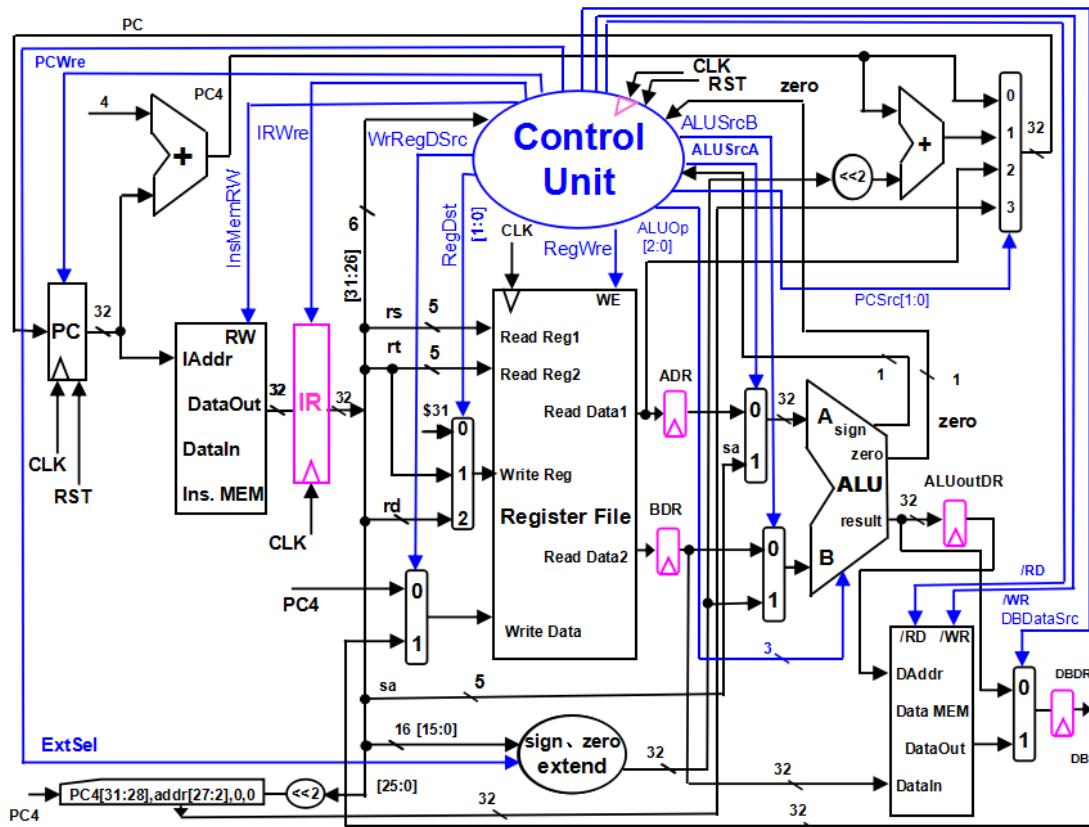


图 4 多周期 CPU 数据通路和控制线路图

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	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与
101	$Y = (A < B) ? 1 : 0$	比较 A 与 B 不带符号

110	<pre> if (A < B &&(A[31] == B[31])) Y = 1; else if (A[31] && !B[31]) Y = 1; else Y = 0; </pre>	比较 A 与 B 带符号
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、 硬件设计实验过程

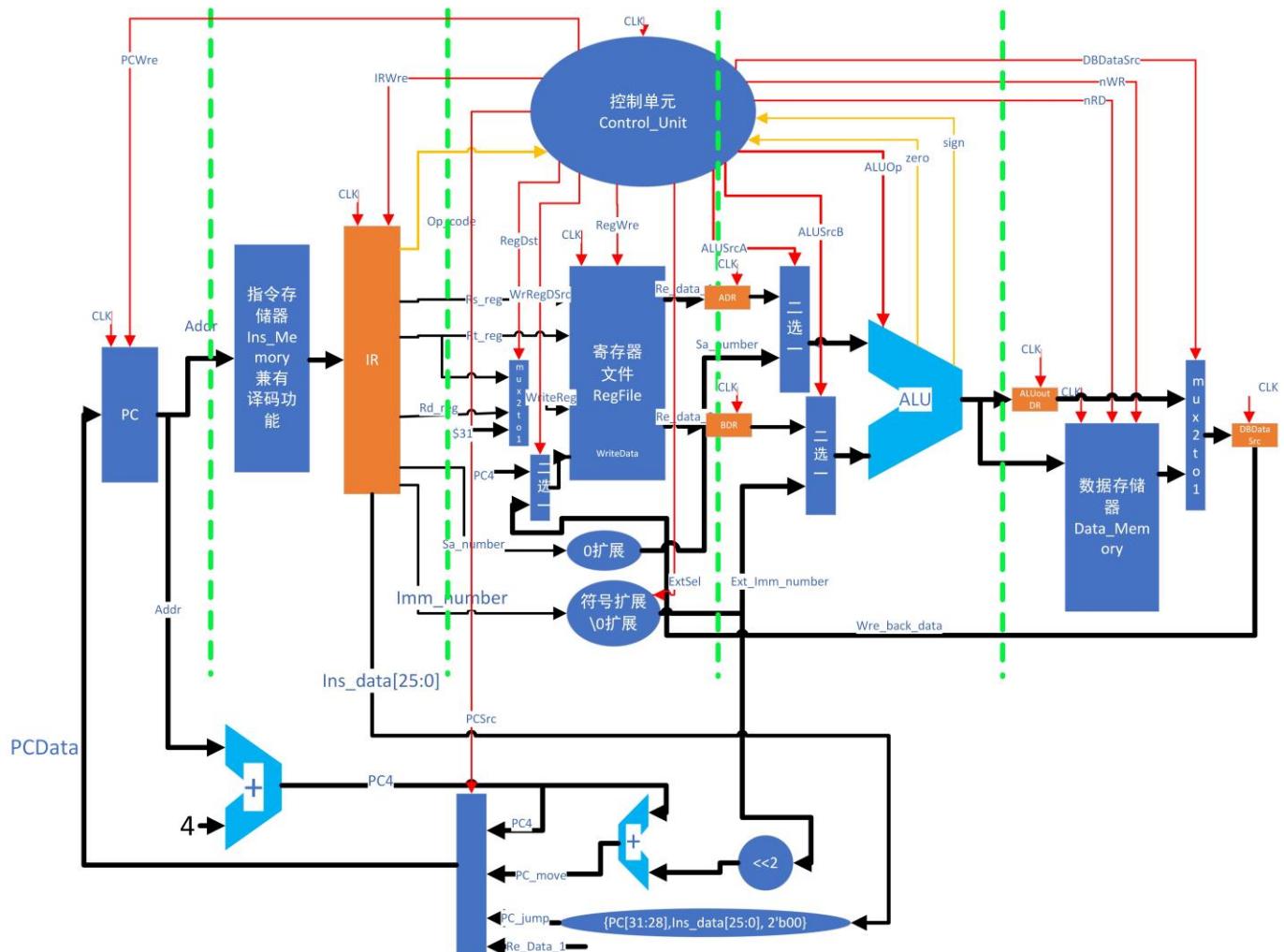
1. 设计数据通路图，确定所需实现子模块

多周期 CPU 的运行一般分为 5 个步骤：取指，译码，执行，访存，写回，更新 PC，这 5 个步骤在图 1-1 中用绿色的虚线分了出来。

图中，红色的为控制信号，用于控制部件的工作状况

黄色的为控制中心的控制信号，用以向控制单元输送当前工作信号。

黑色的为数据/地址信号，且线条粗的为 32 位数据。



从图中可知，我们在实现 CPU 的时候，主要需要实现的模块有六个：

各模块的功能如下表 1-1 所示

模块名	功能
PC 模块	<p>$\text{PCWre} = 0$ 时 不更改地址 【同步更改地址】</p> <p>时钟上升沿且 $\text{PCWre} = 1$ 时 读取下一条地址</p>
INS_MEMORY 模块	<p>预先读取好指令</p> <p>根据输入的 Addr，输出相应位置的指令</p> <p>并将指令进行译码</p>
REGFILE 模块	<p>【异步读寄存器】</p> <p>根据输入的寄存器号 ReadReg1, ReadReg2,</p> <p>直接输出对应的数据</p> <p>【同步写寄存器】</p> <p>在时钟下降沿且 RegWre = 1 时</p>

	将 Re_back_data 写回 WriteReg 对应的寄存器
ALU 模块	根据 ALUOp[2:0] 执行对应的功能 具体的功能表见表 1-3
DATA_MEMORY 模块	【同步读存储器】 根据输入的地址, 当 nRD = 0 时 直接输出对应的数据 【异步写存储器】 在时钟下降沿且 nWR = 0 时 向指定地址写入数据
CONTROL_UNIT 模块	根据输入的 6 位的 Op_code 以及 zero, sign 信号 输出对应的控制信号, 具体控制信号见表 1-1

表 五-1 各模块功能详情

2. 确定控制单元信号表, 编写控制单元模块

指令	ALUSrc A	ALUSr cB	DBDat aSrc	nRD	nWR	ExtSel	PCSrc	RegDst	WrReg DSrc	ALUOp
add	0	0	0	1	1	X	00	10	1	000
addi	0	1	0	1	1	1	00	01	1	000
sub	0	0	0	1	1	X	00	10	1	001
ori	0	1	0	1	1	0	00	01	1	011
and	1	0	0	1	1	X	00	10	1	100
or	0	0	0	1	1	X	00	10	1	011
sll	1	0	0	1	1	X	00	10	1	010
slt	0	0	0	1	1	X	00	10	1	110
slti	0	1	0	1	1	1	00	01	1	11-
sw	0	1	X	1	0	X	00	XX	1	000
lw	0	1	1	0	1	X	00	01	1	000
beq	0	0	1	1	1	1	zero	XX	1	111
bne	0	0	1	1	1	1	~zero	XX	1	111
bgtz	0	0	1	1	1	1	~(sign zero)	XX	1	001
j	X	X	X	X	X	X	11	XX	1	XXX
jr	X	X	X	X	X	X	10	XX	1	XXX
jal	X	X	X	X	X	X	11	00	0	XXXs
halt	X	X	X	X	X	X	XX	X		XXX

1. IRWre 信号

a) 在 IF 阶段为 1, 其余为 0

2. PCWre 信号

- a) 在 IF 阶段（除了 halt 指令）为 1
 b) 其余为 0，防止 PC 被修改。
3. RegWre 信号
- a) 在 WB 阶段为 1
 b) 如果在 ID 阶段且此时指令为 jal，则 RegWre 也为 1
 c) 其他情况为 0;

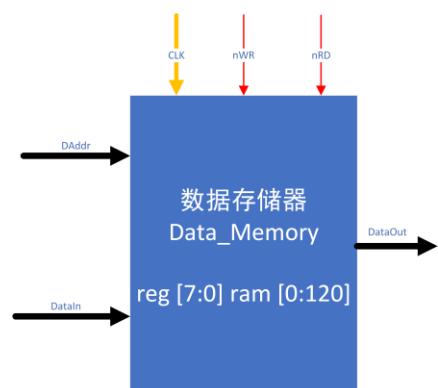
3. 设计 ALU 模块并确定功能

ALUOp[2..0]	功能	描述
000	$Y = A + B$	加
001	$Y = A - B$	减
010	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与
101	$Y = (A < B) ? 1 : 0$	比较 A 与 B 不带符号
110	<pre>if (A < B && (A[31] == B[31])) Y = 1; else if (A[31] && !B[31]) Y = 1; else Y = 0;</pre>	比较 A 与 B 带符号
111	$Y = A \oplus B$	异或

表 五-2 ALU 功能表

4. 设计 Data_Memory 模块

模块端口设计：



端口名称	相关说明
CLK	时钟输入 在 nWR 为 0 的前提下 当时钟下降沿到达时触发写入
nWR	写控制端口 低电平有效

NRD	读控制端口 低电平有效
DADDR[31:0]	需要读或写的内存的地址
DATAIN[31:0]	需要写入内存的数据
DATAOUT	从内存读取的数据

该模块的代码:

```

`timescale 1ns / 1ps
module Data_Memory(
    input CLK,
    input [31:0] DAddr,
    input [31:0] DataIn,// [31:24], [23:16], [15:8], [7:0]
    output [31:0] Dataout,
    input nRD, // 低电平有效, 读控制信号
    input nWR // 高电平有效, 写控制信号
);
    reg [7:0] ram [0:120];

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

    always@( negedge CLK ) begin
        if( nWR==0 ) begin
            ram[DAddr] <= DataIn[31:24];
            ram[DAddr+1] <= DataIn[23:16];
            ram[DAddr+2] <= DataIn[15:8];
            ram[DAddr+3] <= DataIn[7:0];
        end
    end
endmodule

```

需要注意的几个地方:

1. 该 CPU 读写内存的方式为大端法。大端法, 即低位数据放在高地址处, 会影响数据存储器将 32 位的数据存储到以字节为单位的内存中, 也同样会影响从内存中读取数据到寄存器。
2. 数据存储器为时钟下降沿时进行写入, 这与 PC 模块在时钟上升沿时进行地址的转移是相互搭配的。在时钟上升沿的时候进行组合电路的运行, 下降沿的时候进行数据的存储, 这样能够确保数据读写与指令执行之间不会发生冲突。

5. PC 的设计

端口设计如图所示



```

`timescale 1ns / 1ps
// 注意时钟上升沿还是下降沿触发
module PC(
    input CLK,
    input Reset, // 0: 初始化 PC 为 0 1: 接受新地址
    input PCWre, // 0 不更改 1 更改
    input [31:0] PCData,
    output reg [31:0] Addr
);
    always@(negedge CLK or negedge Reset) begin
        if(Reset == 0)
            Addr = 0;
        else begin
            if (PCWre == 0)
                Addr = Addr;
            else
                Addr = PCData;
        end
    end
endmodule

```

需要说明的问题：

1. 在敏感表中，加入了关于 `Reset` 的检测，是出于这样的考虑：
如果在敏感表中没有关于 `Reset` 下降沿的检测，那么在进行单步调试的时候，只要不按按钮，不论怎么调 `Reset` 都没有，因为无法触发语句执行。为了方便重置程序计数器 PC，就采用了检测 `Reset` 下降沿的做法，这样子，就可以在不按按钮的前提下，直接发送 `Reset` 信号得到重置。
2. PC 模块采用了时钟上升沿来进行 PC 的更新。
3. 当 `PCWre = 0` 时，出现了类似锁存器的硬件，这是因为需要配合 `halt` 指令。一旦运行 `halt` 指令，程序的地址就会一直维持在当前的地址，不进行改变。
- 4.

其他的模块就不继续赘述，源代码可见 `github` 仓库

https://github.com/WalkerYF/CPU_multi_cycle

6. 编写 CPU_top 模块并使用代码测试

按照数据通路连接好 top 模块，并使用了在文件附件中《关于测试多周期 CPU 的简单方法》中给出的代码进行测试。该代码能够涵盖该 CPU 设计文档中所实现的所有指令。

关于详细的测试过程，请看第六节：仿真波形图解读

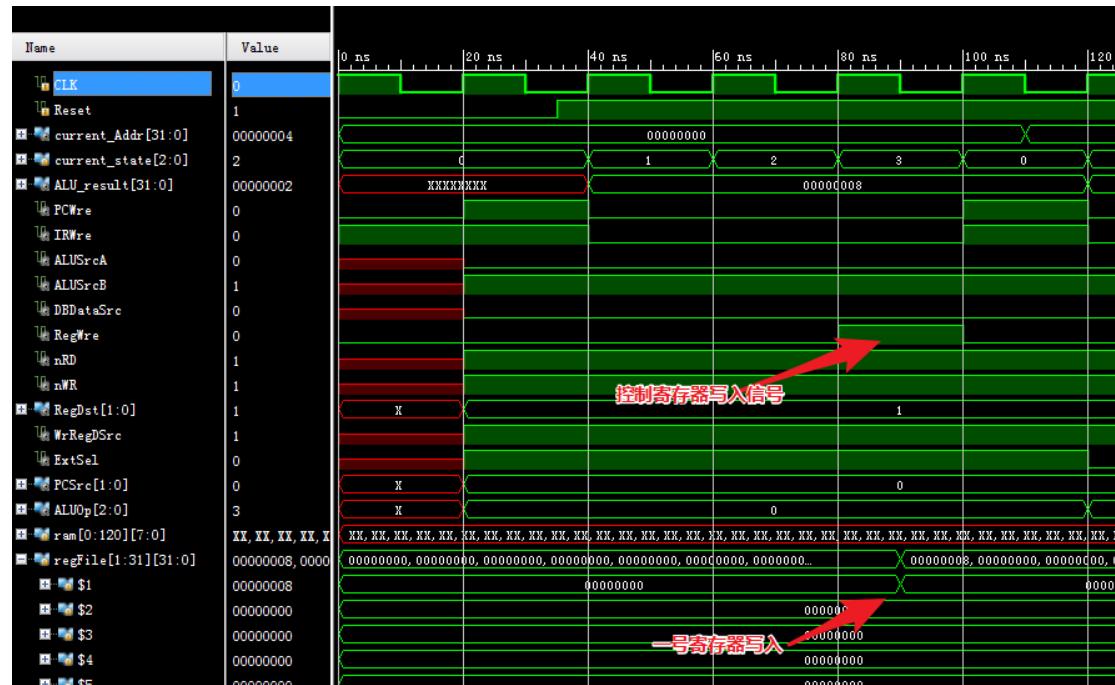
地址	汇编程序	指令代码				
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000000	addi \$1,\$0,8	000010	00000	00001	0000 0000 0000 1000	= 08010008
0x00000004	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010	= 48020002
0x00000008	or \$3,\$2,\$1	010000	00010	00001	00011 000 0000 0000	= 40411800
0x0000000C	sub \$4,\$3,\$1	000001	00011	00001	00100 000 0000 0000	= 04612000
0x00000010	and \$5,\$4,\$2	010001	00100	00010	00101 000 0000 0000	= 44822800
0x00000014	sll \$5,\$5,2	011000	00000	00101	00101 00010 00 0000	= 60052880
0x00000018	beq \$5,\$1,-2 (=,转 14)	110100	00101	00001	1111 1111 1111 1110	= D0A1FFE
0x0000001C	jal 0x0000048	111010	00000	00000	000000 0000 0100 10	= E8000012
0x00000020	slt \$8,\$12,\$1	100110	01100	00001	01000 000 0000 0000	= 99814000
0x00000024	addi \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110	= 080EFFFE
0x00000028	slt \$9,\$8,\$14	100110	01000	01110	01001 000 0000 0000	= 990E4800
0x0000002C	slti \$10,\$9,2	100111	01001	01010	0000 0000 0000 0010	= 9D2A0002
0x00000030	slti \$11,\$10,0	100111	01010	01011	0000 0000 0000 0000	= 9D4B0000
0x00000034	add \$11,\$11,\$8	000000	01011	01000	01011 000 0000 0000	= 01685800
0x00000038	bne \$11,\$2,-2 (≠,转 34)	110101	01011	00010	1111 1111 1111 1110	= D562FFFE
0x0000003C	addi \$2,\$2,-1	000010	00010	00010	1111 1111 1111 1111	= 0842FFFF
0x00000040	bgtz \$2,-2 (>0,转 3C)	110110	00010	00000	1111 1111 1111 1110	= D840FFFE
0x00000044	j 0x0000054	111000	00000	00000	000000 0000 0101 01	= E0000015
0x00000048	sw \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100	= C0220004
0x0000004C	lw \$12,4(\$1)	110001	00001	01100	0000 0000 0000 0100	= C42C0004
0x00000050	jr \$31	111001	11111	00000	0000 0000 0000 0000	= E7E00000
0x00000054	halt	111111	00000	00000	0000 0000 0000 0000	= FC000000

表格 五-1 测试代码样例

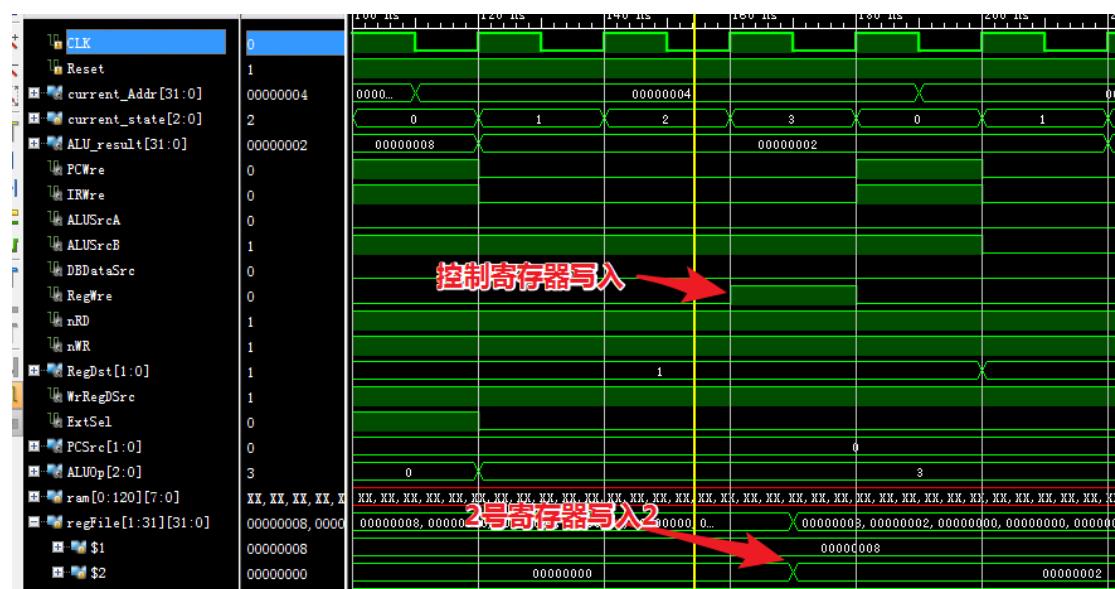
六、 实验结果：仿真波形图解

注意：以下代码的运行是具有时间上的连贯性的，要注意寄存器的值

1. addi \$1, \$0, 8



2. ori \$2,\$0,2



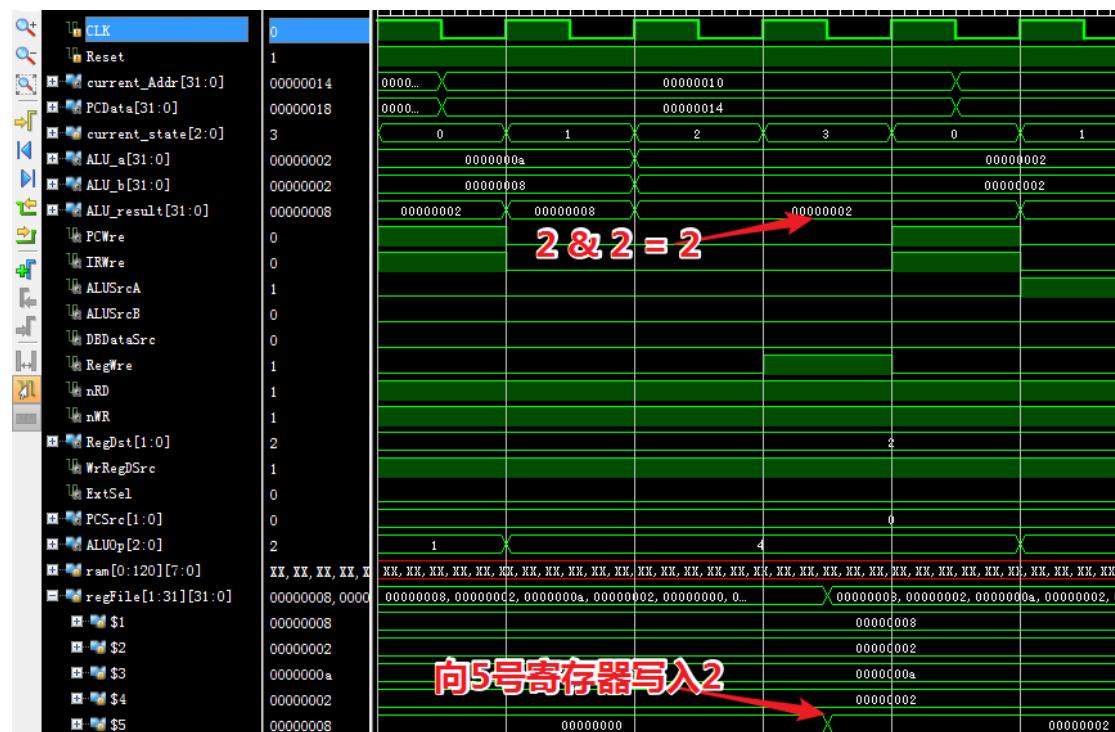
3. . or \$3,\$2,\$1



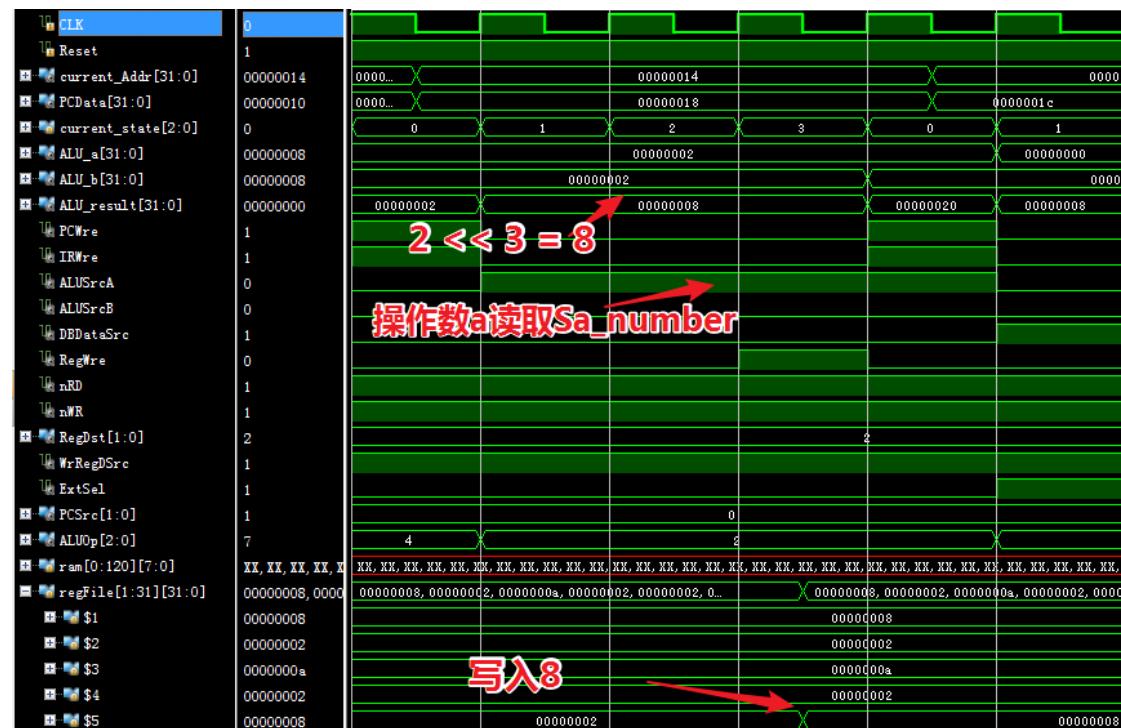
4. sub \$4,\$3,\$1.



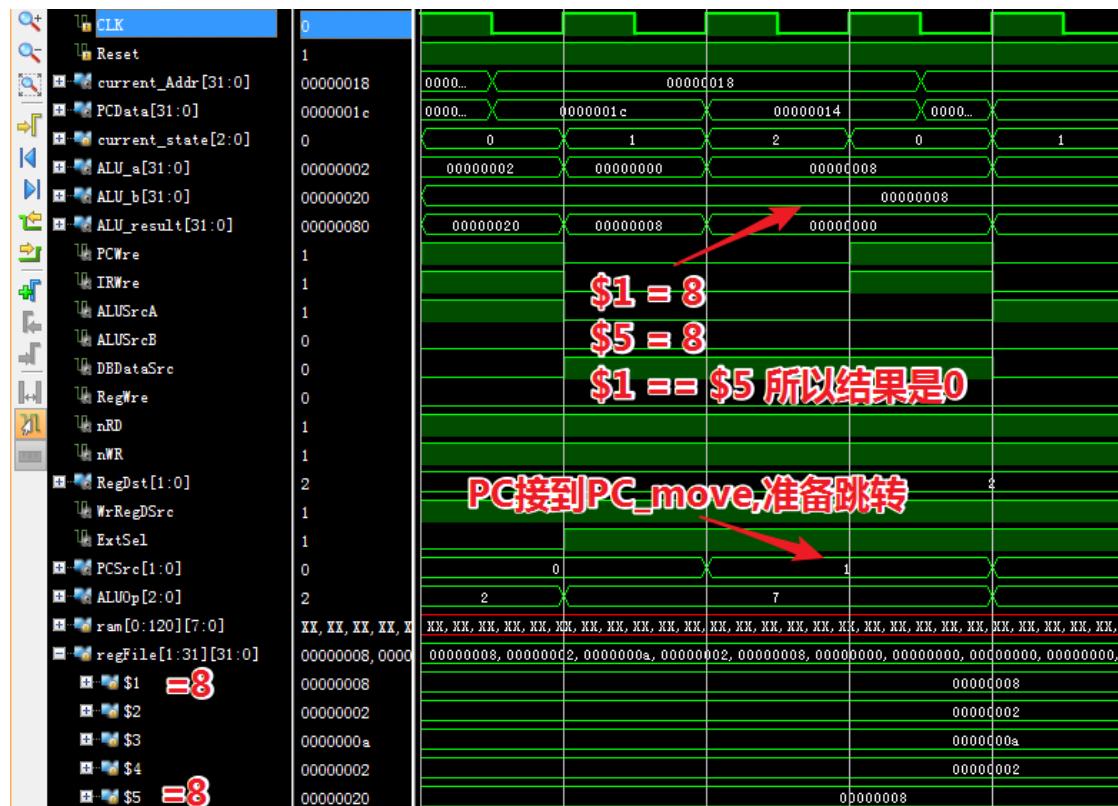
5. and \$5,\$4,\$2.



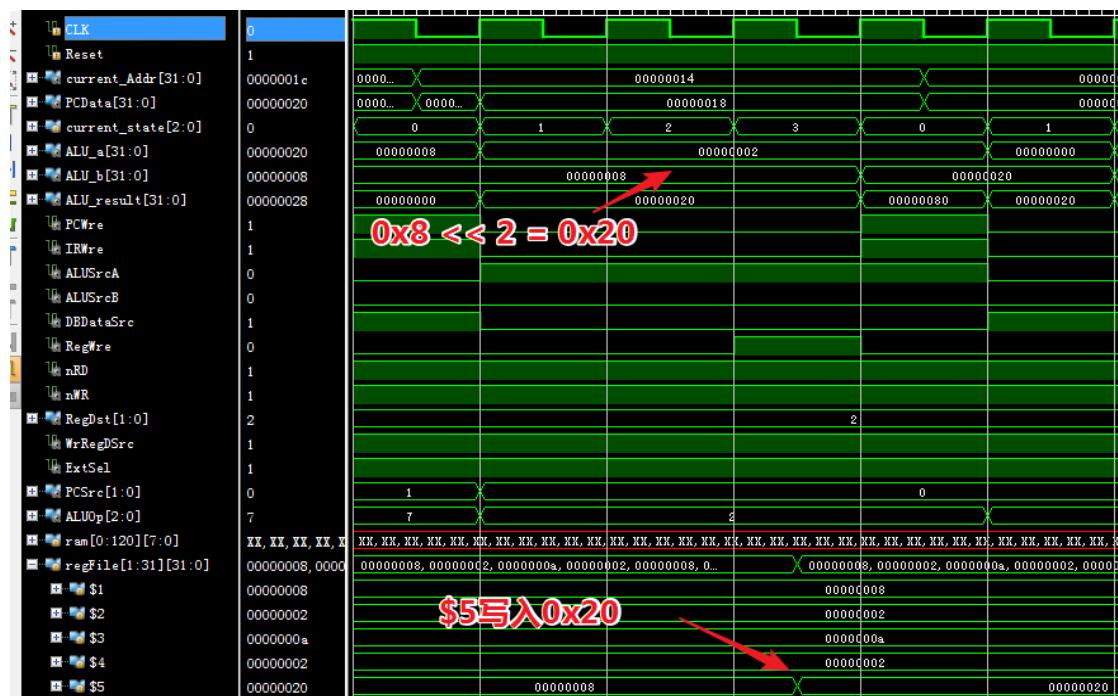
6. sll \$5,\$5,2.



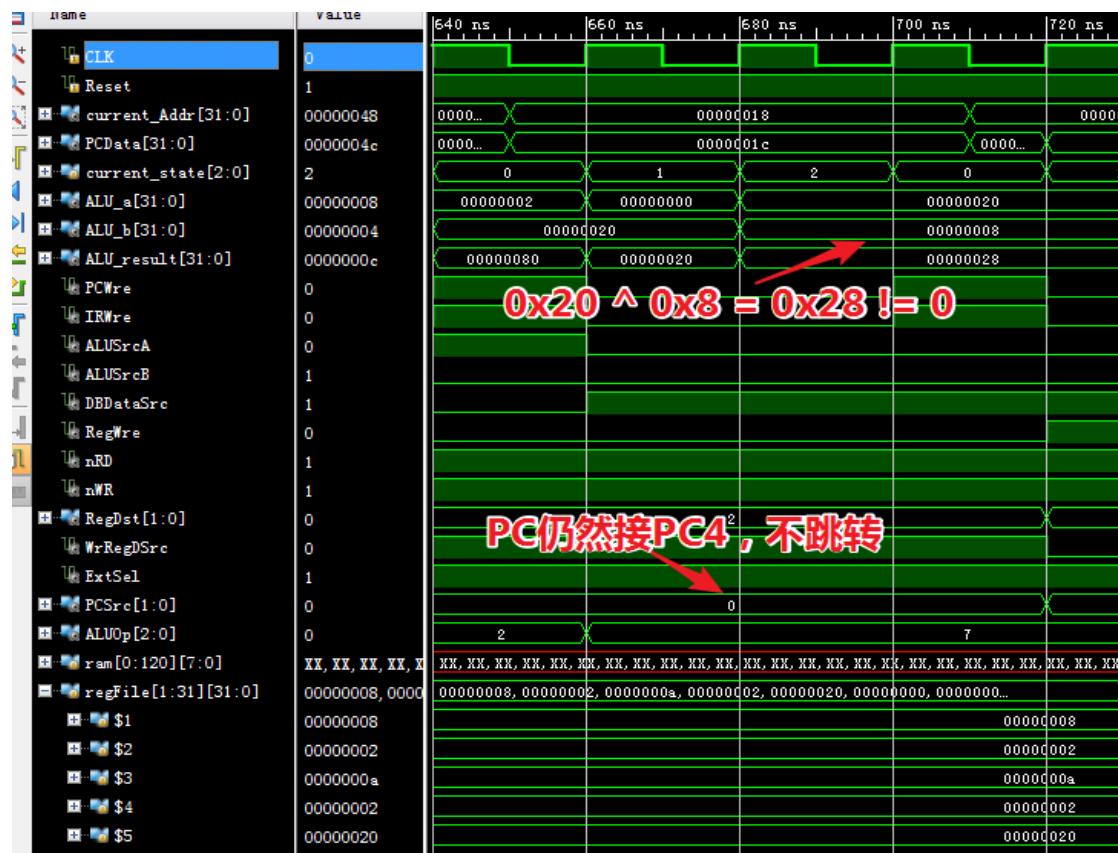
7. beq \$5,\$1,-2(=,转 14).



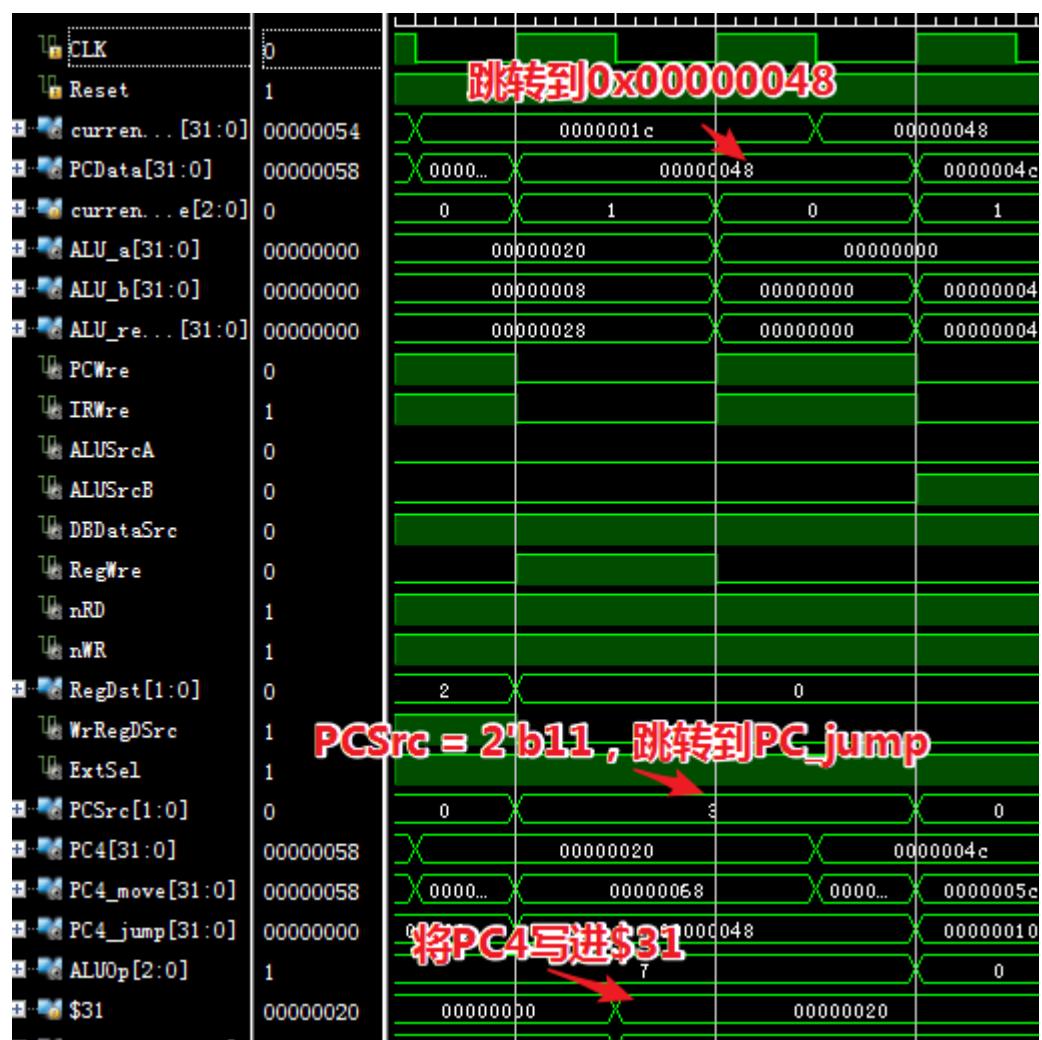
8. sll \$5,\$5,2.



9. beq \$5,\$1,-2(!=,不跳)



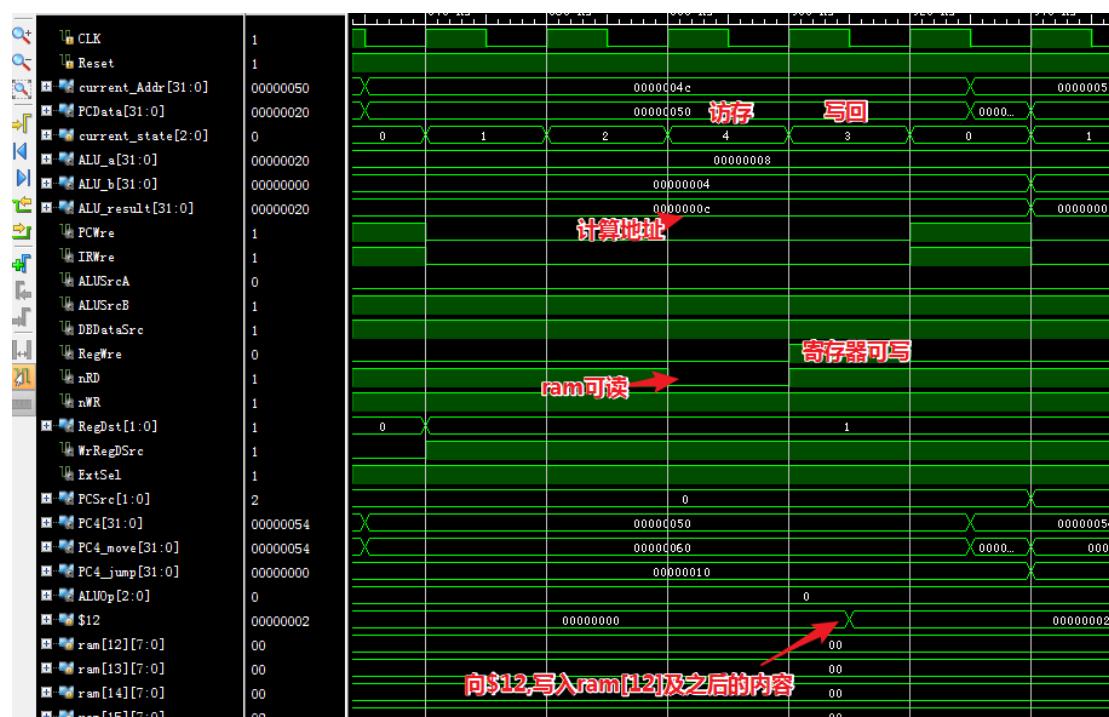
10.jal 0x0000048



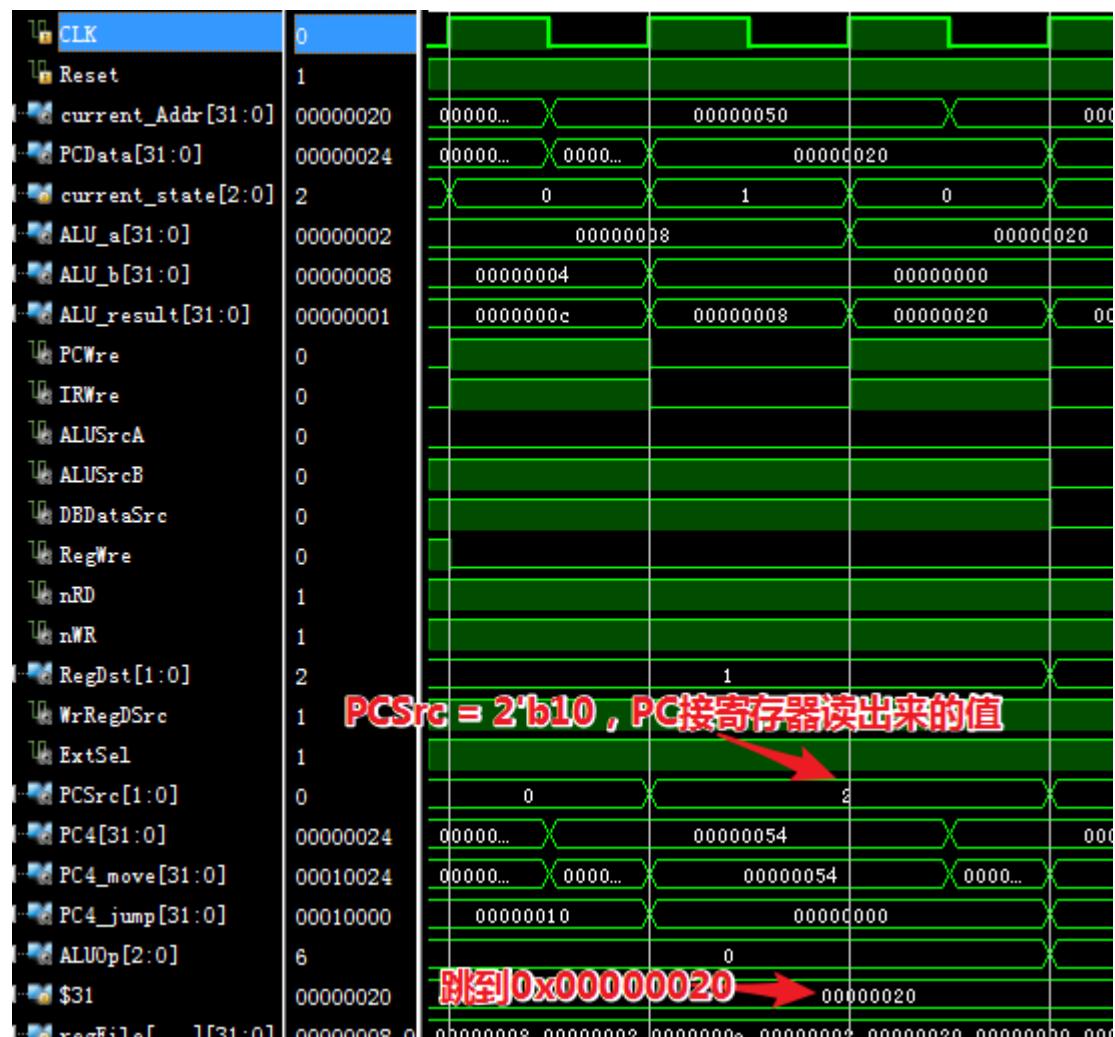
11.sw \$2,4(\$1)



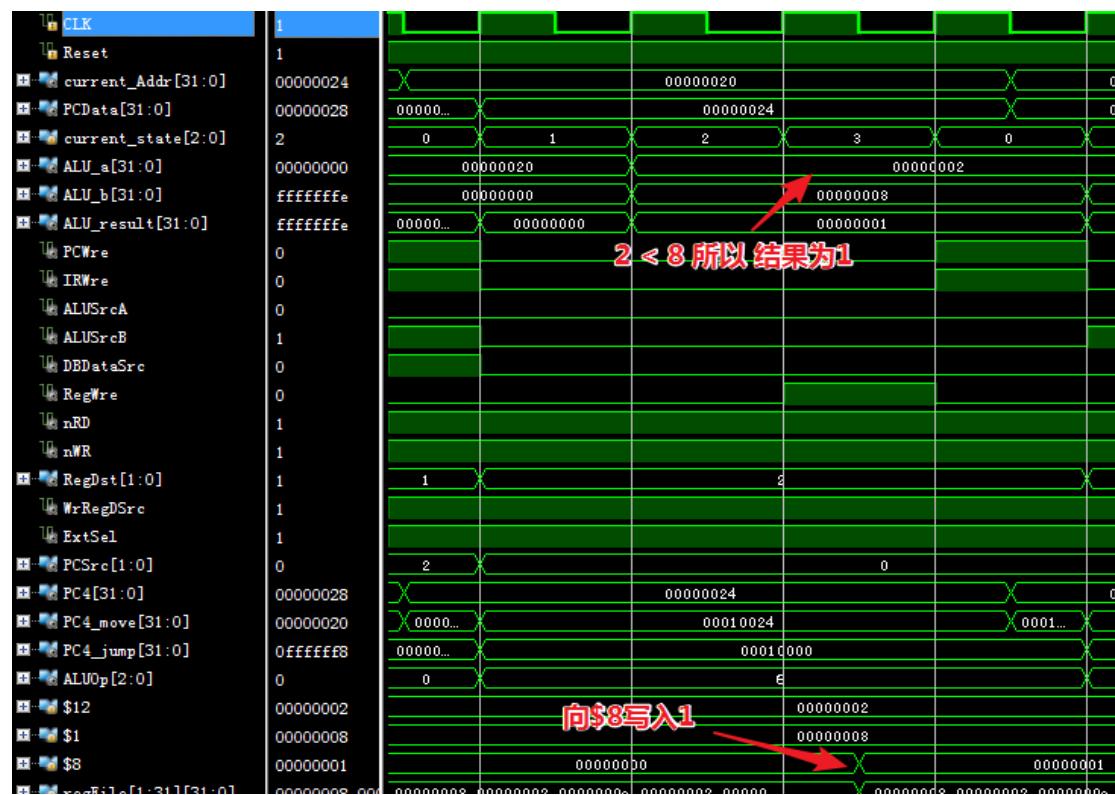
12.lw \$12,4(\$1)



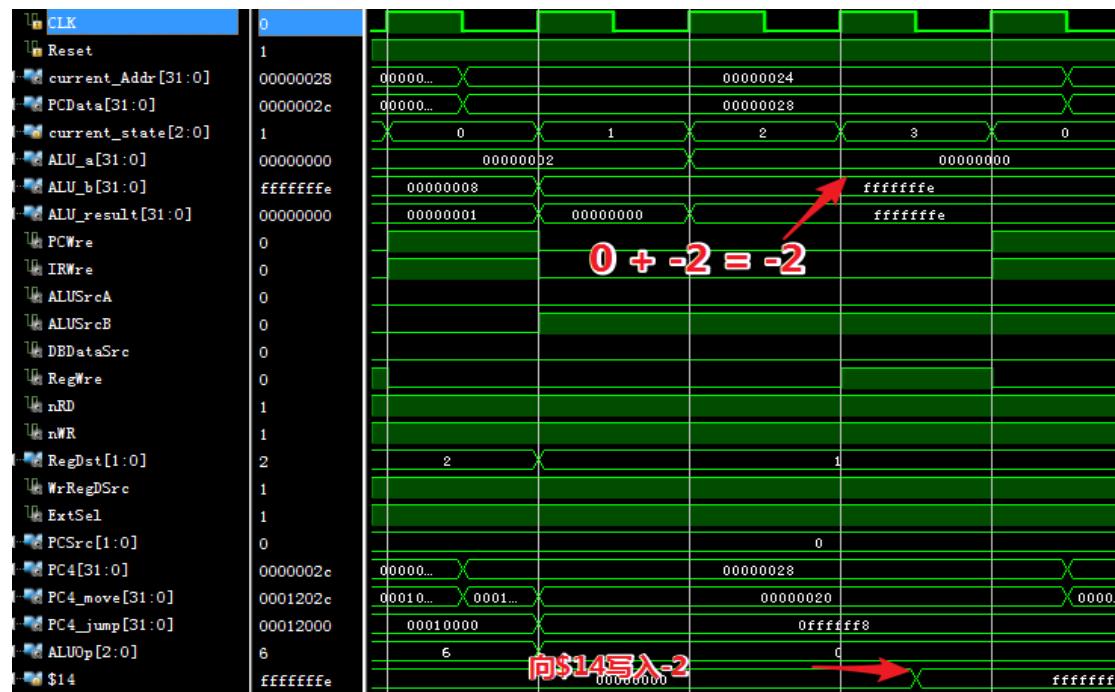
13.jr \$31



14.slt \$8,\$12,\$1



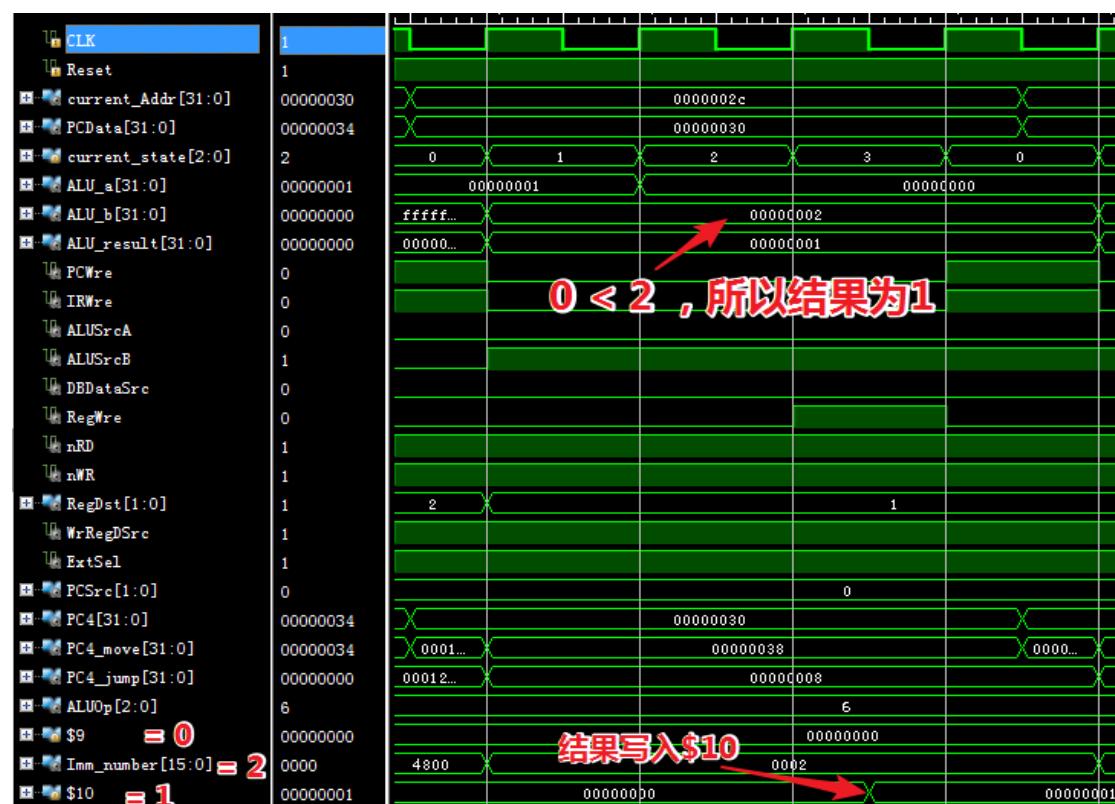
15.addi \$14,\$0,-2



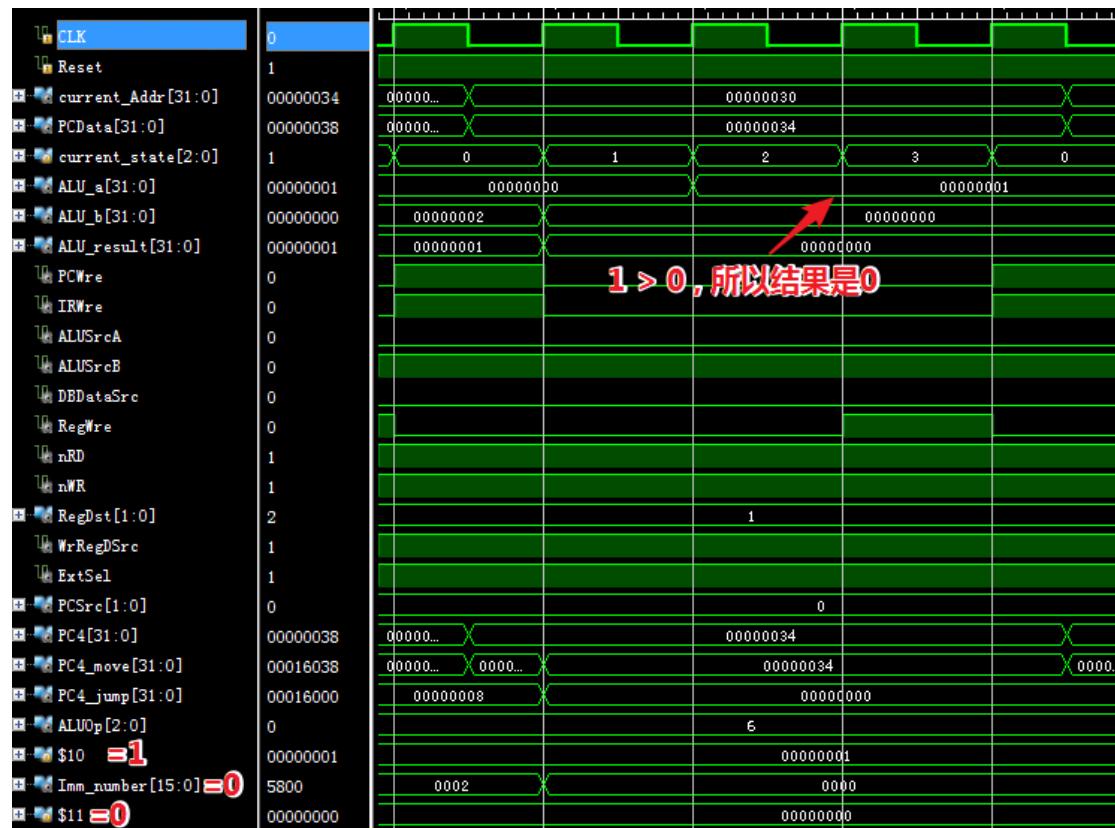
16.slt \$9,\$8,\$14



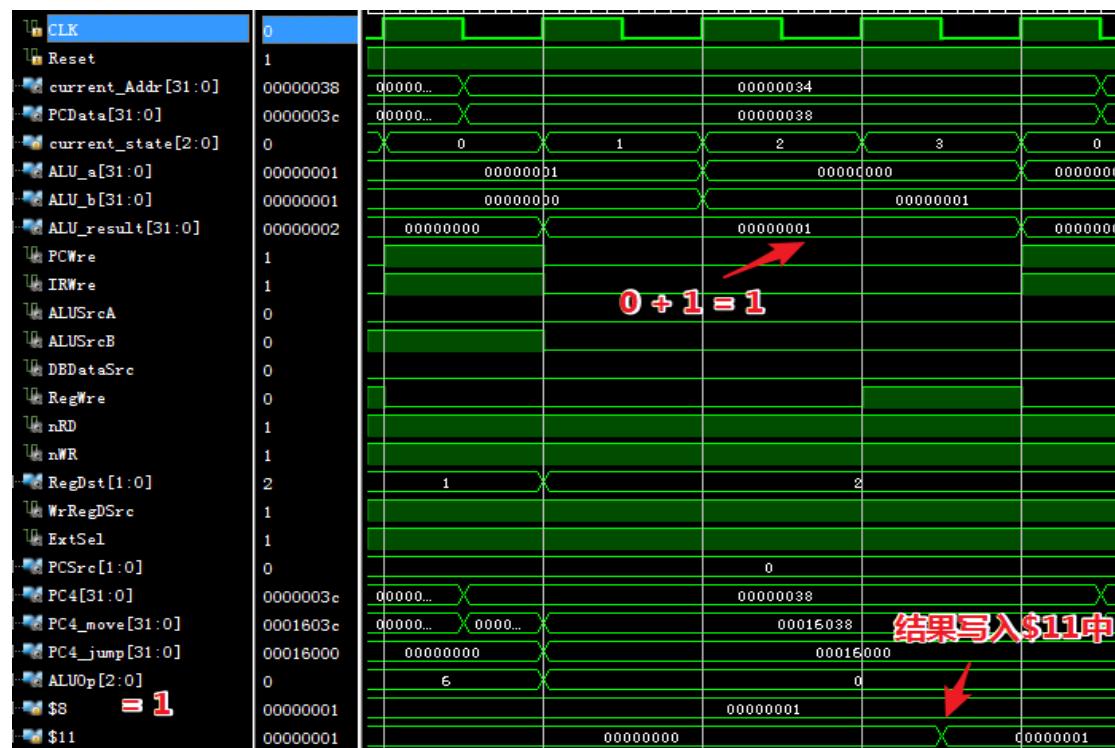
17.slti \$10,\$9,2

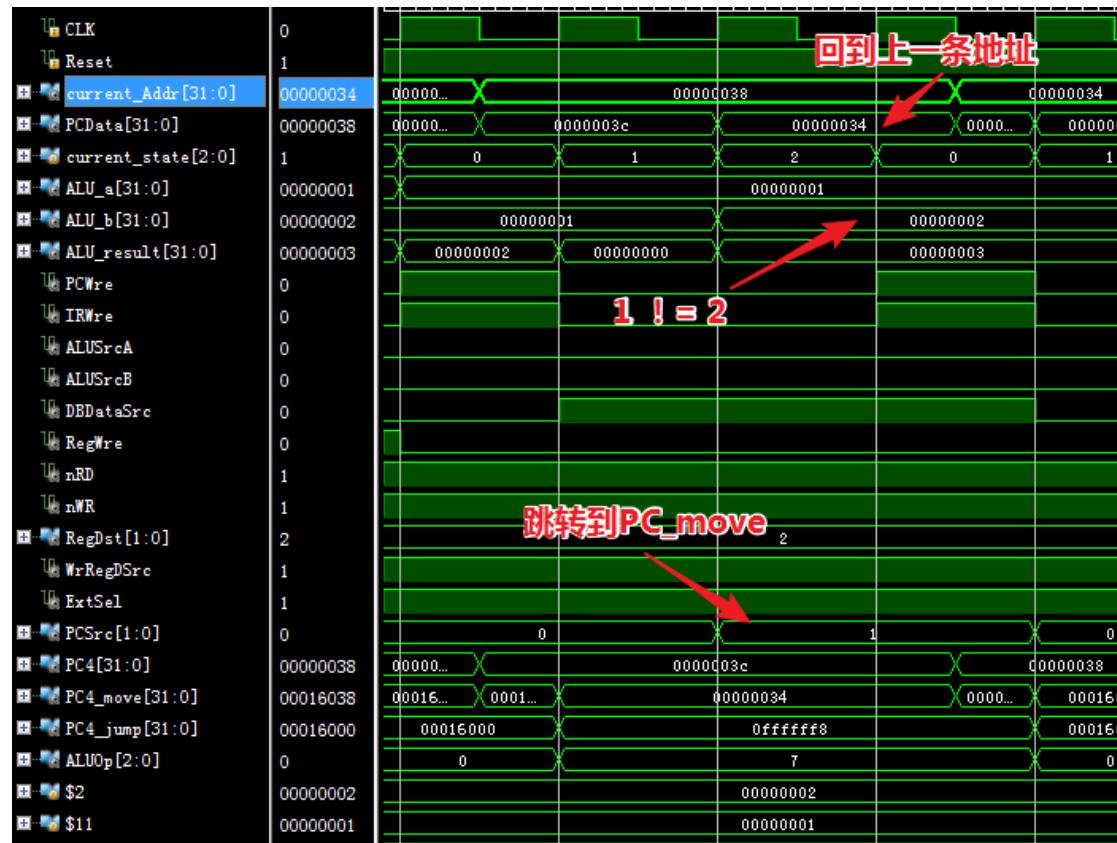


18.slti \$11,\$10,0

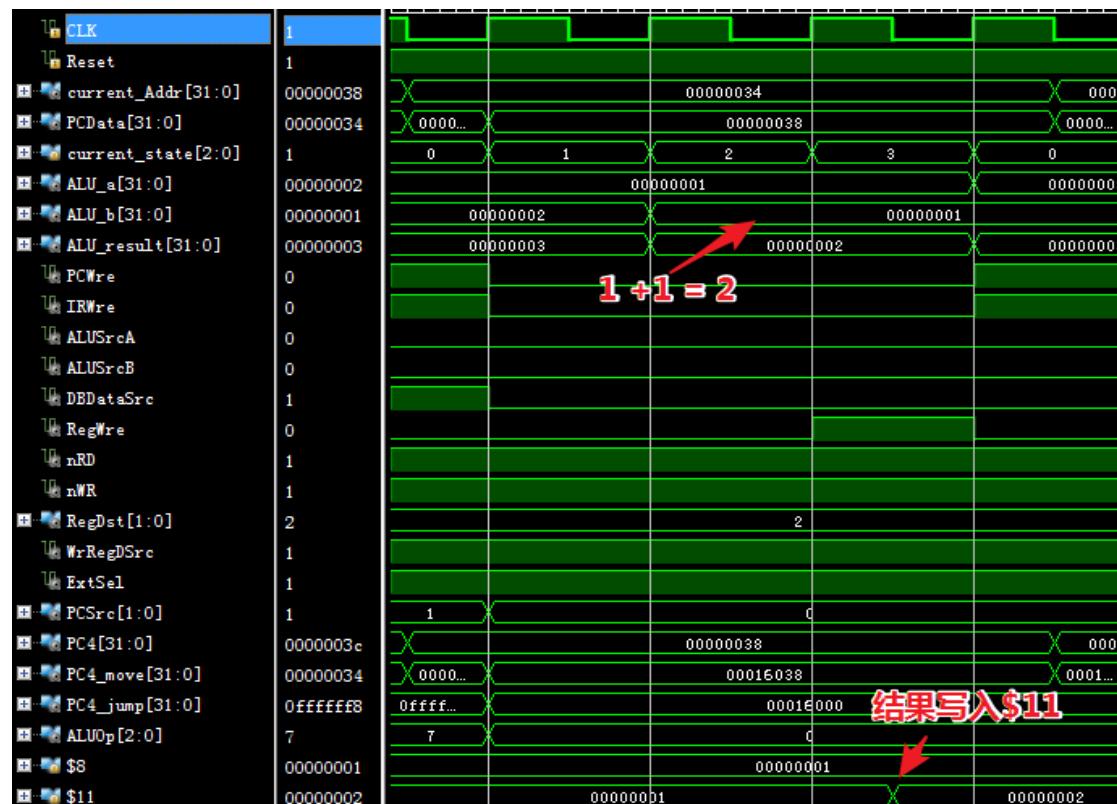


19.add \$11,\$11,\$8

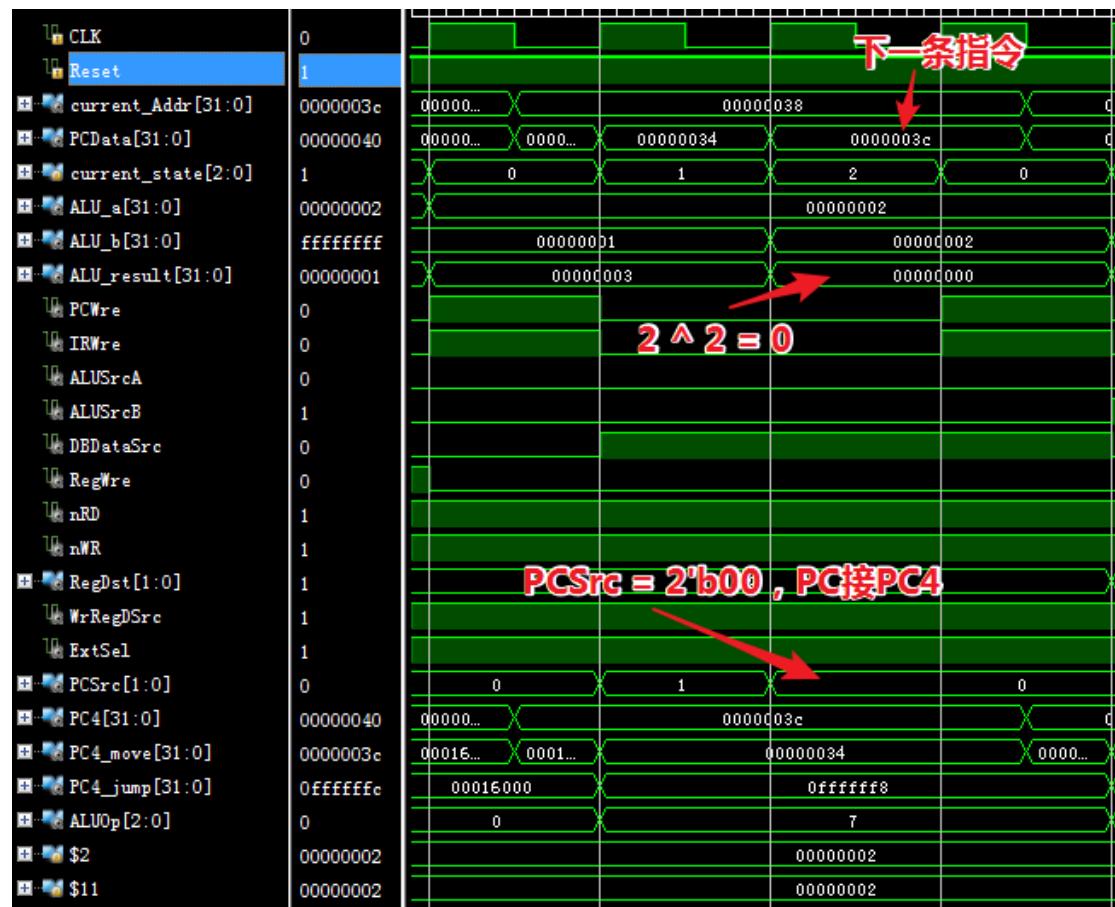


20.bne \$11,\$2,-2 (\neq , 转 34)

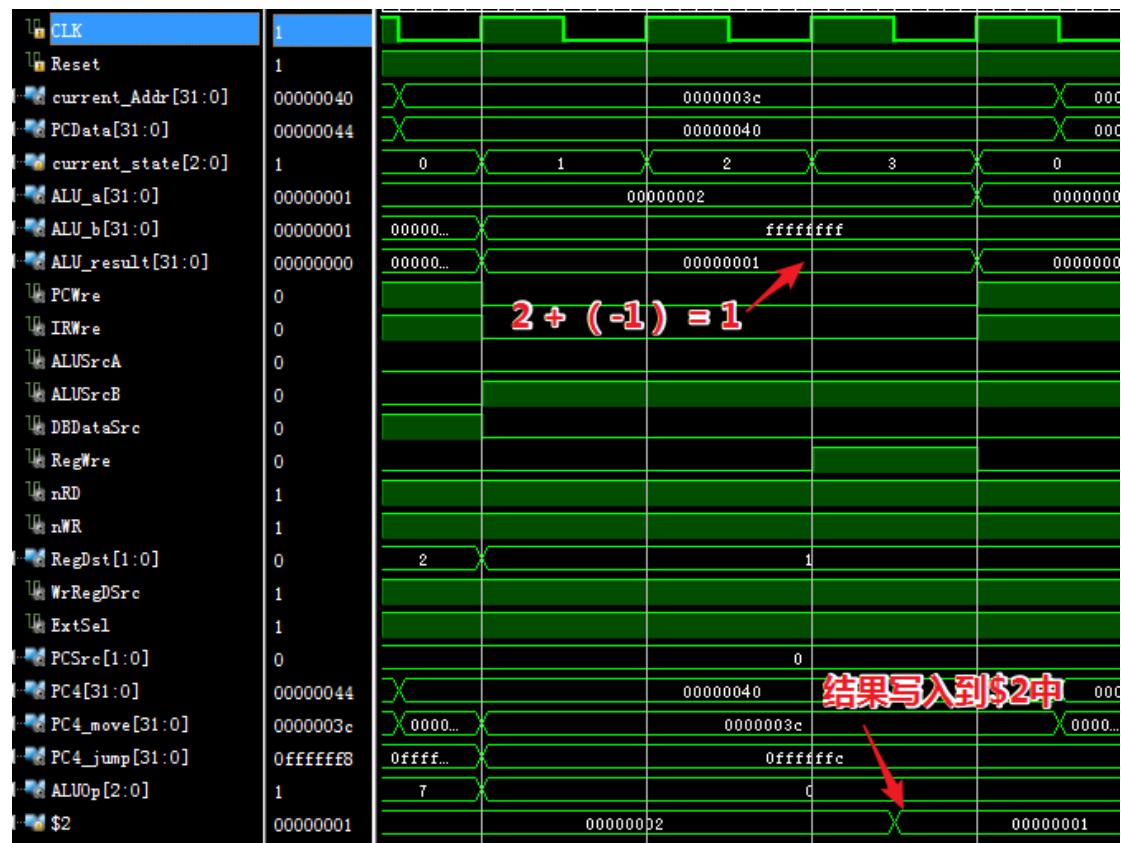
21.add \$11,\$11,\$8



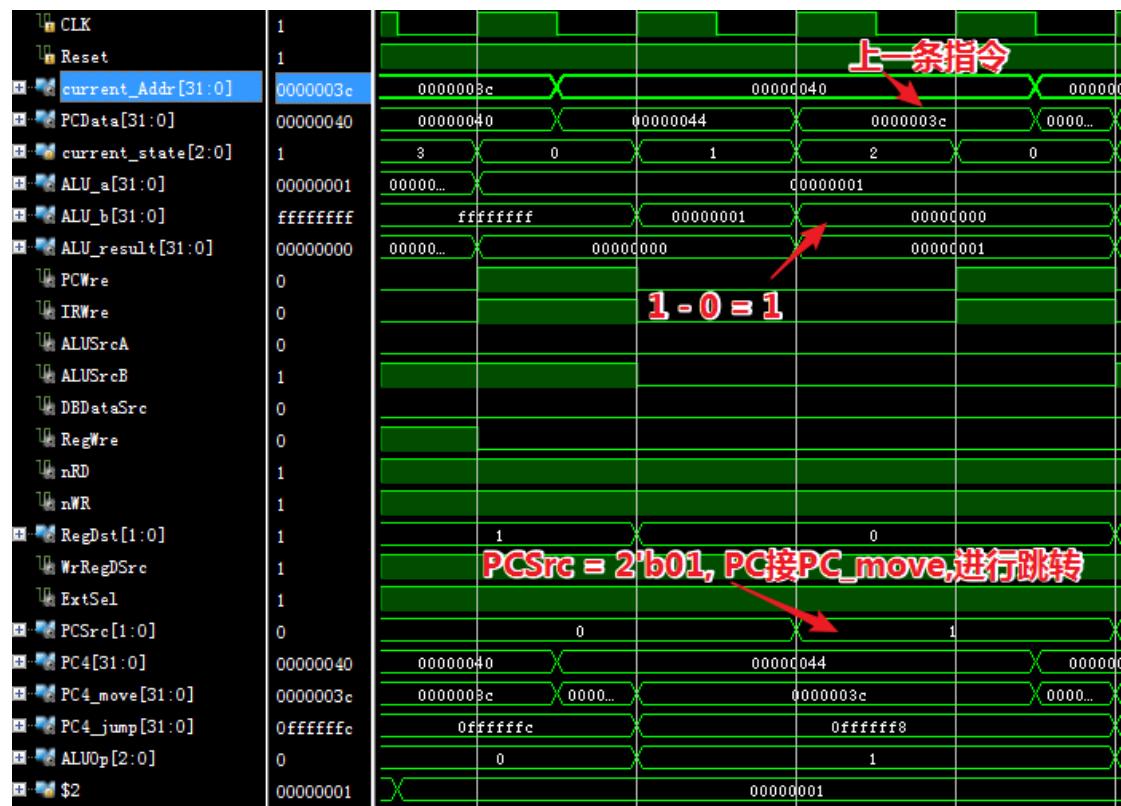
22.bne \$11,\$2,-2 (=,不转)



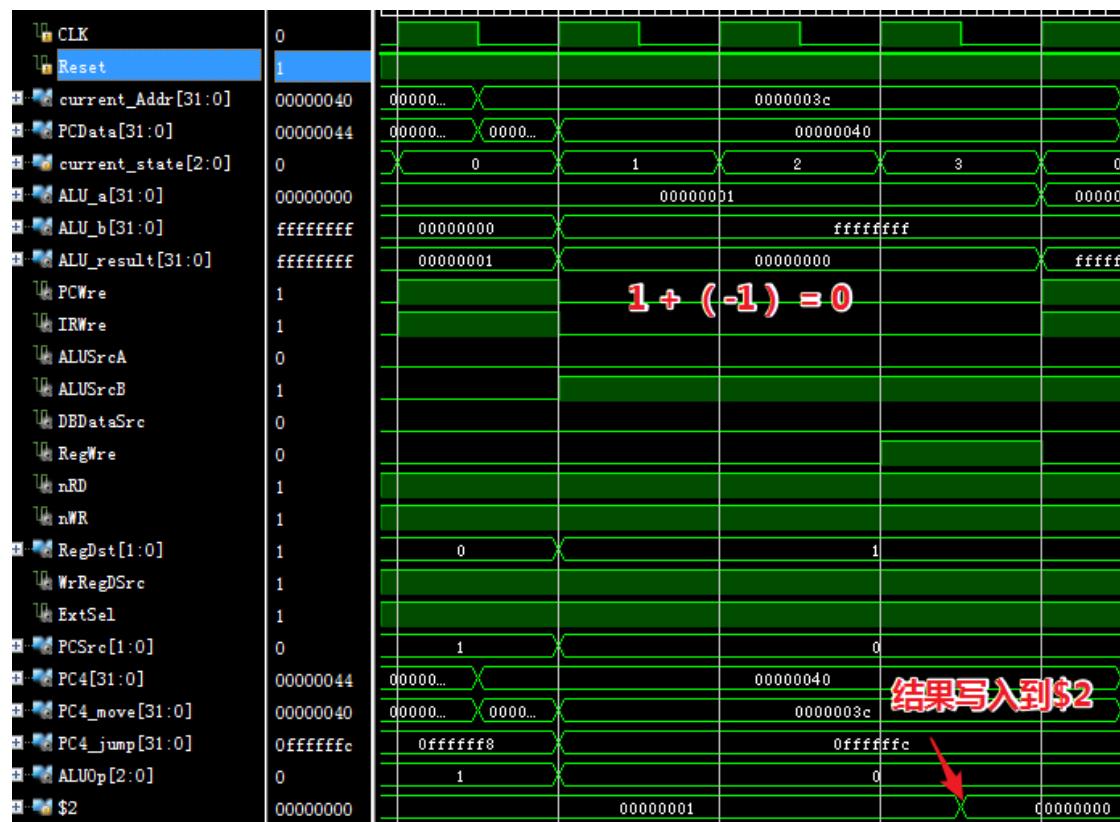
23.addi \$2,\$2,-1



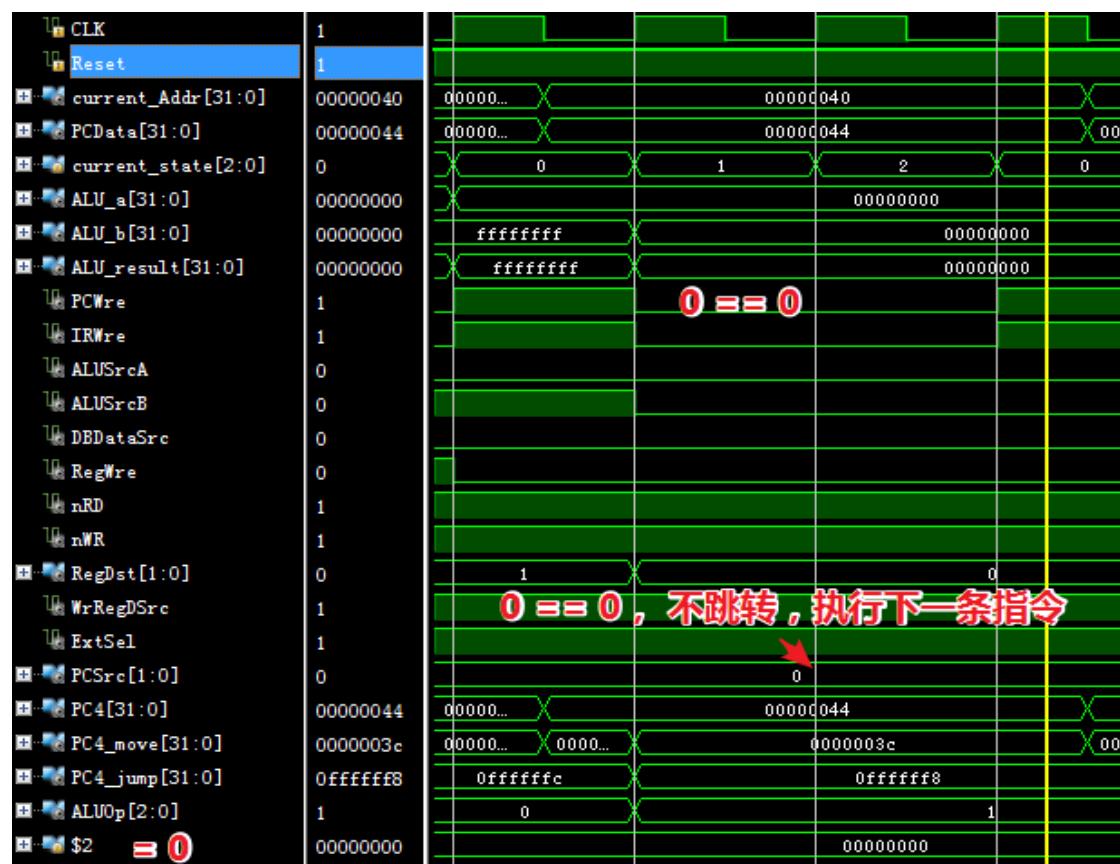
24.bgtz \$2,-2 (>0, 转 3C)



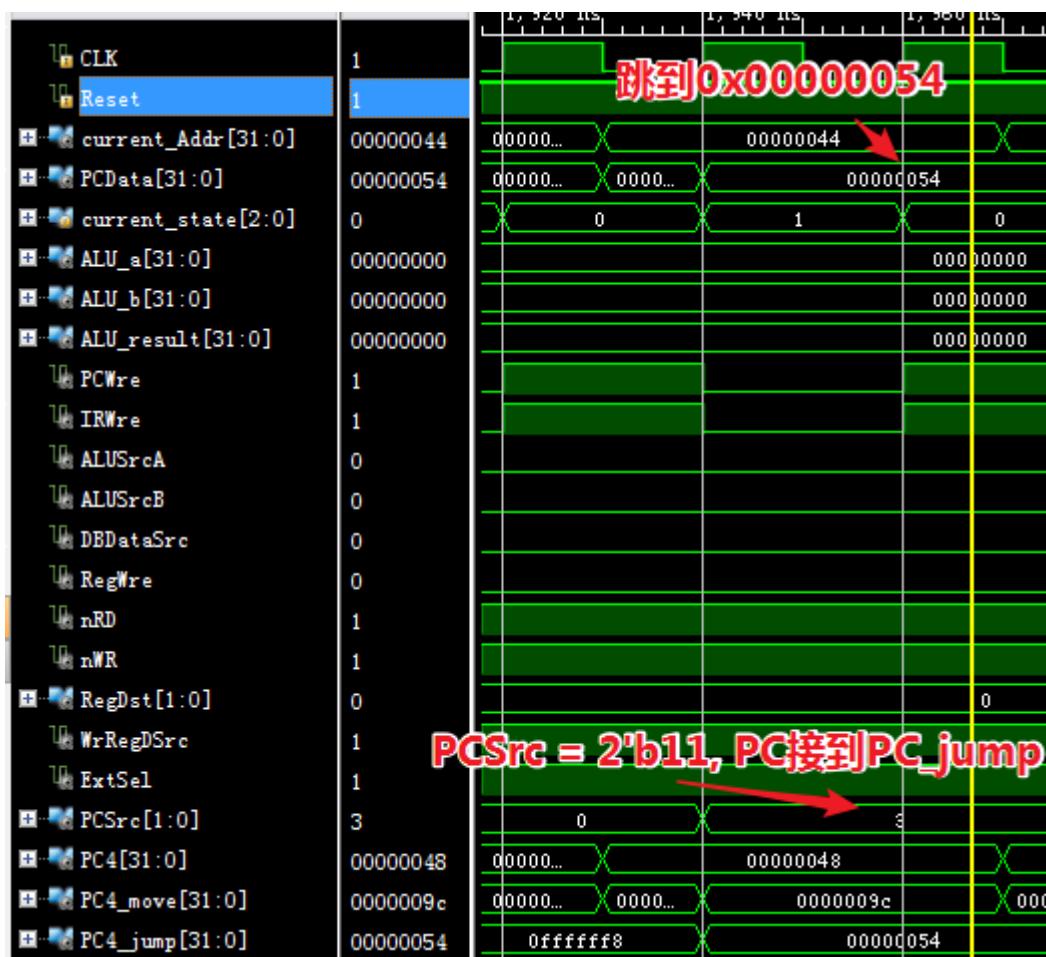
25.addi \$2,\$2,-1



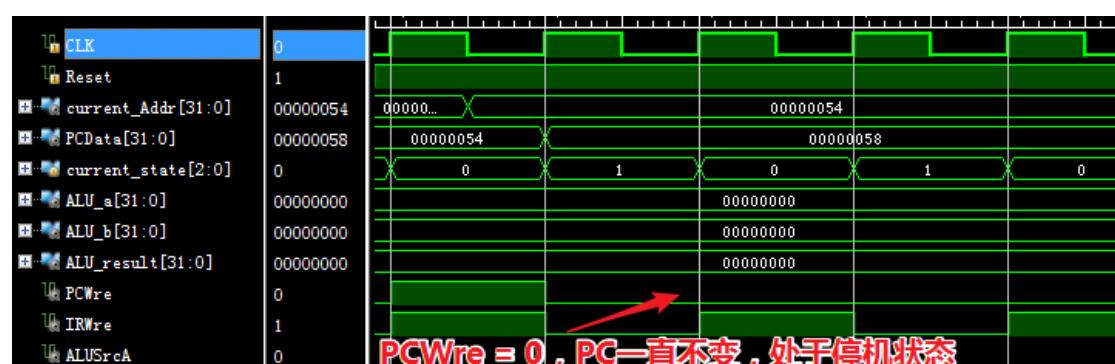
26.bgtz \$2,-2 (>0, 转 3C)



27.j 0x0000054



28.halt



七、 在 Basy3 实验板上显示

1. 按键消抖模块

按键消抖这里采用了延时的操作
通过延时 20ms，得到稳定的按键的状态。
按键消抖模块的代码如下：

```

module debouncing (
    clk,
    rst, // 低电平有效
    key_n,
    key_pulse
);
// 只会检测下降沿，然后下降沿一到达就释放一个时钟周期的高电平
parameter N = 1; //要消除的按键的数
input clk;
input rst; //低电平有效
input [N-1:0] key_n; //输入的按键
output [N-1:0] key_pulse; //按键动作产生的脉冲
wire key;
wire key_pulse_n;
assign key = ~key_n;
reg [N-1:0] key_RST_Pre;
//定义一个寄存器型变量存储上一个触发时的按键值
reg [N-1:0] key_RST;
//定义一个寄存器变量存储当前时刻触发的按键值
wire [N-1:0] key_EDGE;
//检测到按键由高到低变化是产生一个高脉冲
//利用非阻塞赋值特点，将两个时钟触发时按键状态存储在两个寄存器变量中
always @(posedge clk or negedge rst)
begin
    if (!rst) begin
        key_RST <= {N{1'b1}};
    //初始化时给 key_RST 赋值全为 1, {}中表示 N 个 1
        key_RST_Pre <= {N{1'b1}};
    end
    else begin
        key_RST <= key;
    //第一个时钟上升沿触发之后 key 的值赋给 key_RST,同时 key_RST 的值赋给
    //key_RST_Pre
        key_RST_Pre <= key_RST;
    //非阻塞赋值。相当于经过两个时钟触发，key_RST 存储的是当前时刻 key 的值,
    //key_RST_Pre 存储的是前一个时钟的 key 的值
    end
end

```

```

assign key_edge = key_rst_pre & (~key_rst);
//脉冲边沿检测。当 key 检测到下降沿时，key_edge 产生一个时钟周期的高电平

reg [21:0] cnt;
//产生延时所用的计数器，系统时钟 100MHz，要延时 20ms，至少需要 21 位计数器
//产生 20ms 延时，当检测到 key_edge 有效是计数器清零开始计数
always @(posedge clk or negedge rst)
begin
    if(!rst)
        cnt <= 21'h0;
    else if(key_edge)
        cnt <= 21'h0;
    else
        cnt <= cnt + 1'h1;
end

reg [N-1:0] key_sec_pre;
//延时后检测电平寄存器变量
reg [N-1:0] key_sec;
//延时后检测 key，如果按键状态变低产生一个时钟的高脉冲。如果按键状态是高
//的话说明按键无效
always @(posedge clk or negedge rst)
begin
    if (!rst)
        key_sec <= {N{1'b1}};
    else if (cnt==18'hffff)
        key_sec <= key;
end
always @(posedge clk or negedge rst)
begin
    if (!rst)
        key_sec_pre <= {N{1'b1}};
    else
        key_sec_pre <= key_sec;
end
assign key_pulse_n = key_sec_pre & (~key_sec);
assign key_pulse = ~key_pulse_n;

endmodule

```

2. 编写 top 模块，实例化 CPU 并显示信号

八、 实验结果：basys3 板上运行 CPU

以这五条指令为例

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
0x00000001C	jal 0x00000048		111010	00000	00000	000000 0000 0100 10	E8000012
0x000000048	sw \$2,4(\$1)		110000	00001	00010	0000 0000 0000 0100	C0220004
0x00000004C	lw \$12,4(\$1)		110001	00001	01100	0000 0000 0000 0100	C42C0004
0x000000050	jr \$31		111001	11111	00000	0000 0000 0000 0000	E7E00000
0x000000020	slt \$8,\$12,\$1		100110	01100	00001	01000 000 0000 0000	99814000

1. 以下图片的说明

图片以四宫格形式呈现每一个阶段的状态

左上角为当前地址：下一个地址

右上角为 rs 寄存器：及其内容

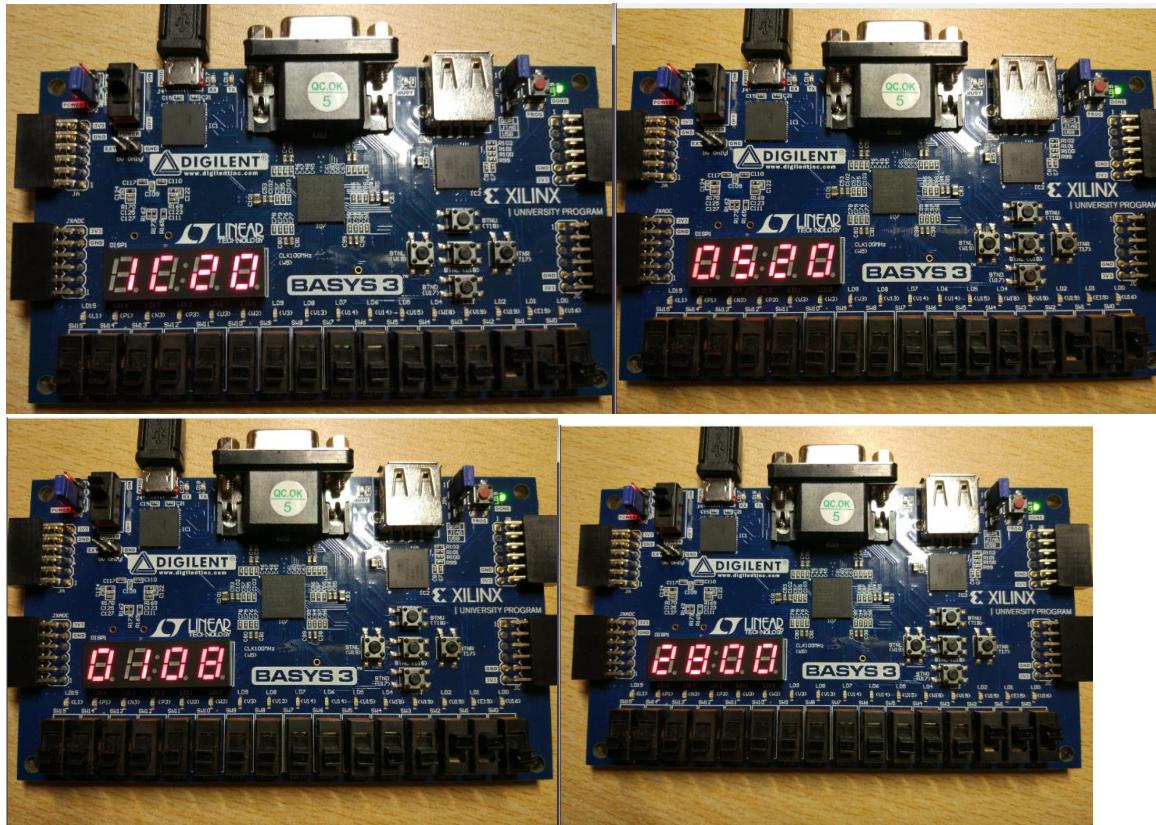
左下角为 rd 寄存器：及其内容

右下角为 ALU 结果：写回寄存器的值

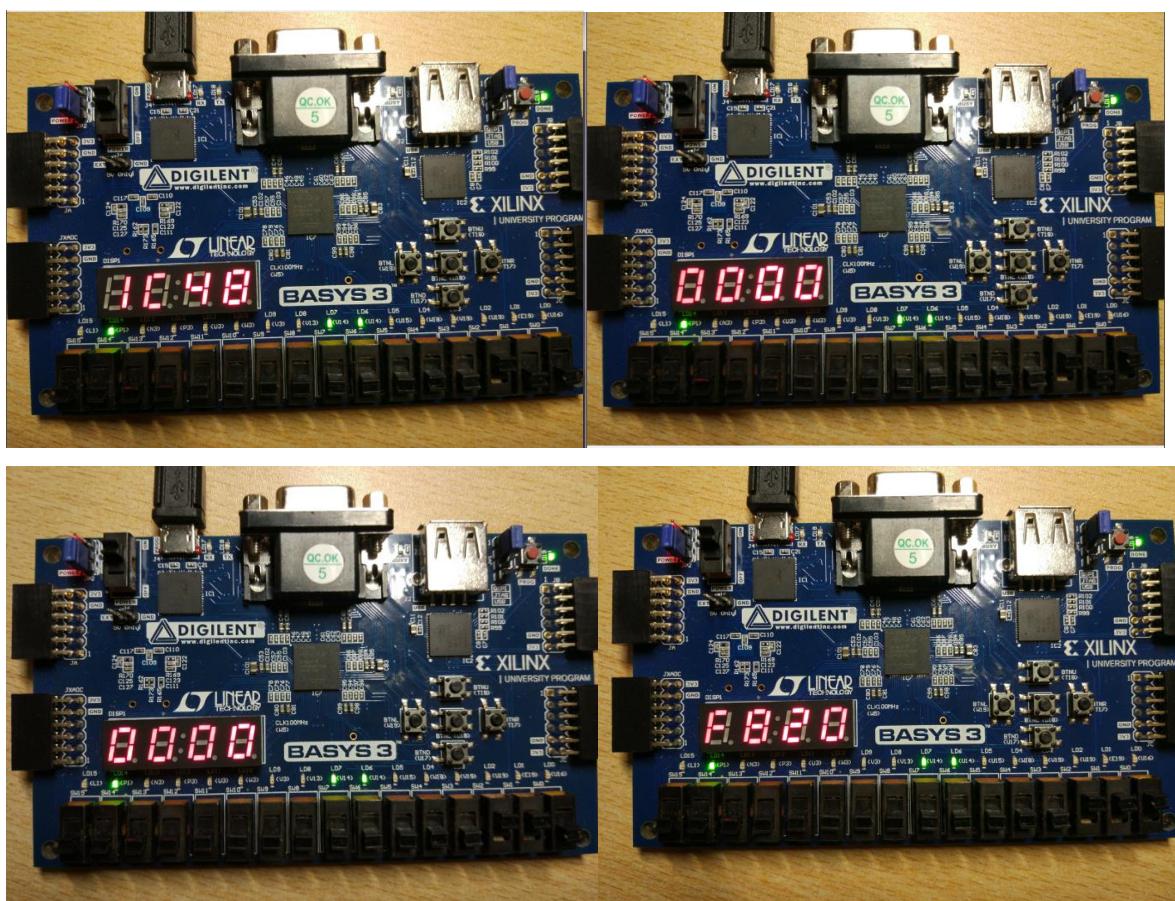
部分周期展示了两个阶段，如跳转指令：jal 以及 jr

2. jal 0x0000048w \$9,4(\$1)

以下是 IF 阶段，刚开始还没有更新控制单元，下一条地址仍然为 PC4

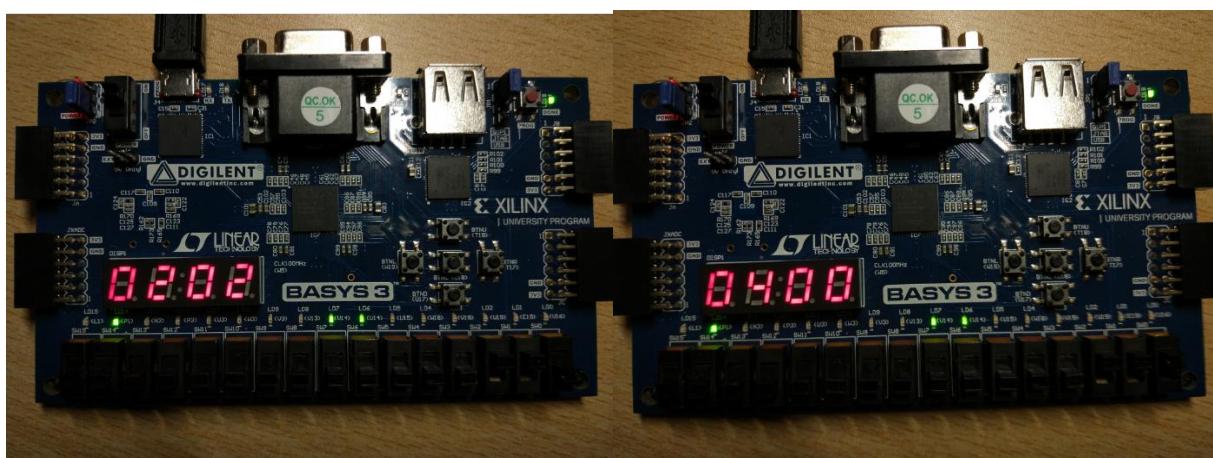
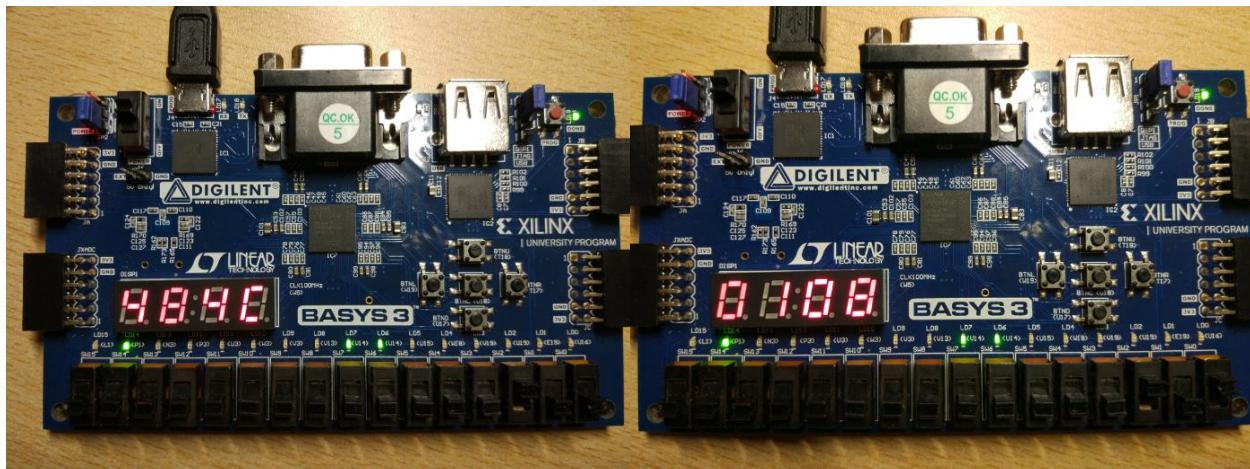


以下为 ID 阶段，可见下一地址的修改。

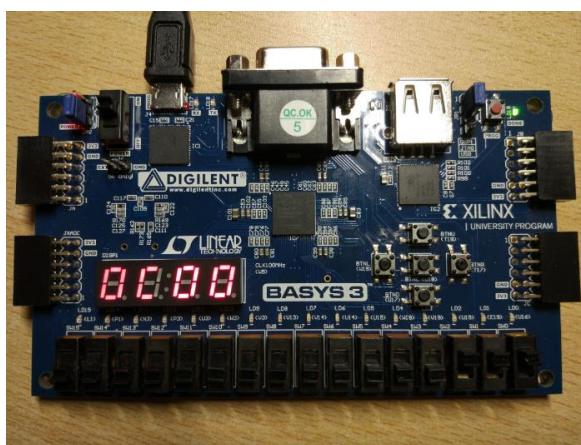


3. sw \$2,4(\$1)

这个为 ID 阶段的 sw 指令



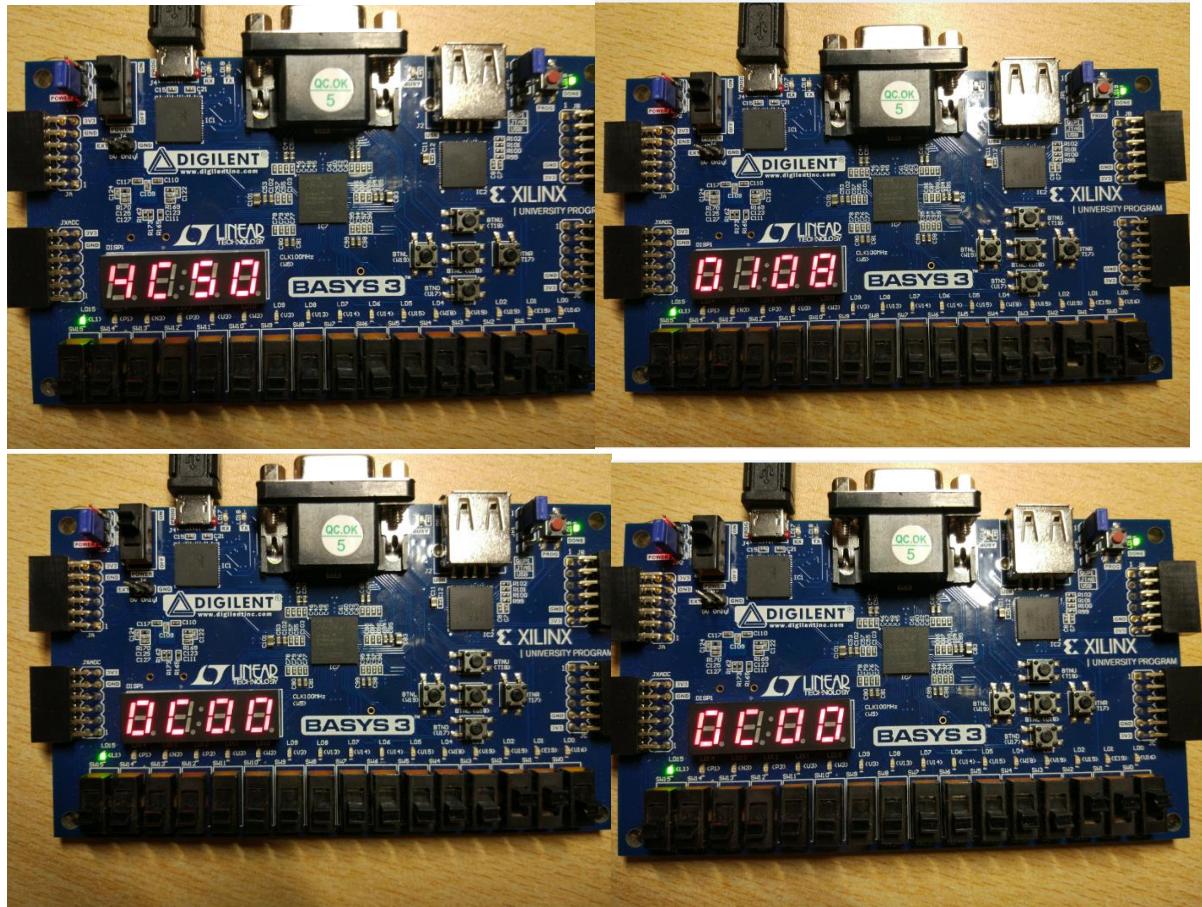
最后一个图:



这最后一个图展示了在 EXE 阶段后 ALU 结果的变化，ALU 计算出了应该存的地址。

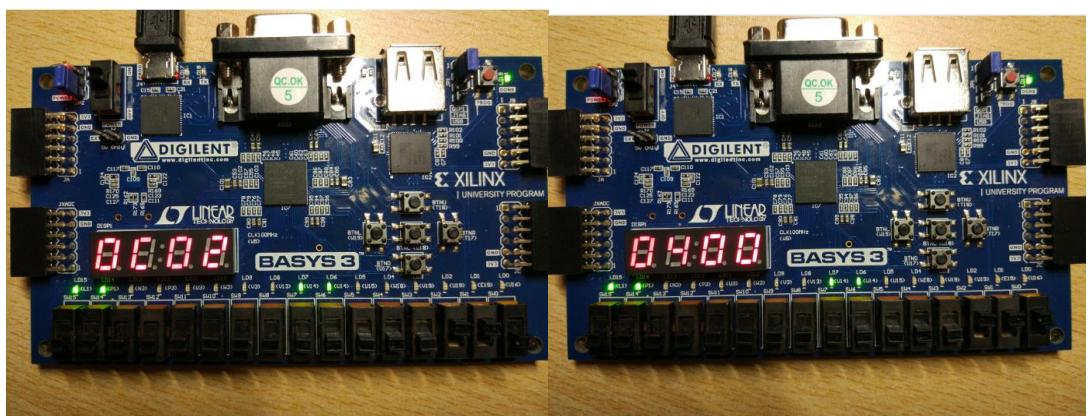
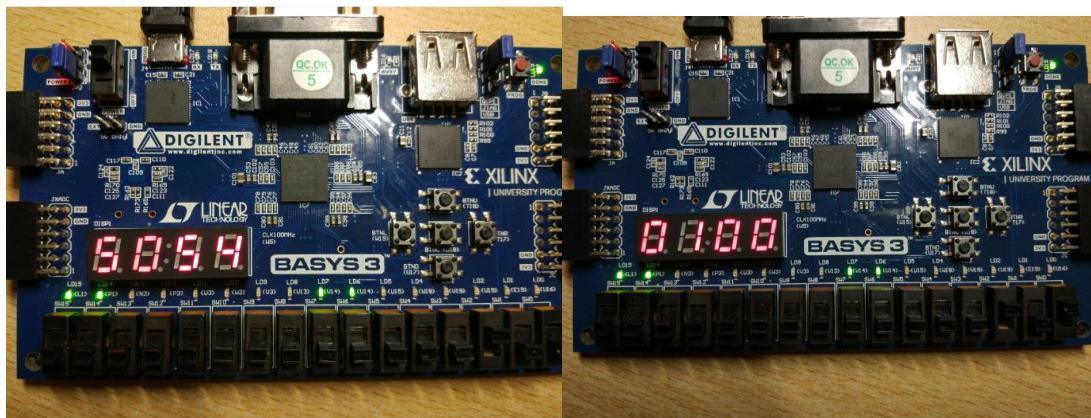
4. lw \$12,4(\$1)

这个为 WD 阶段的 lw 指令。

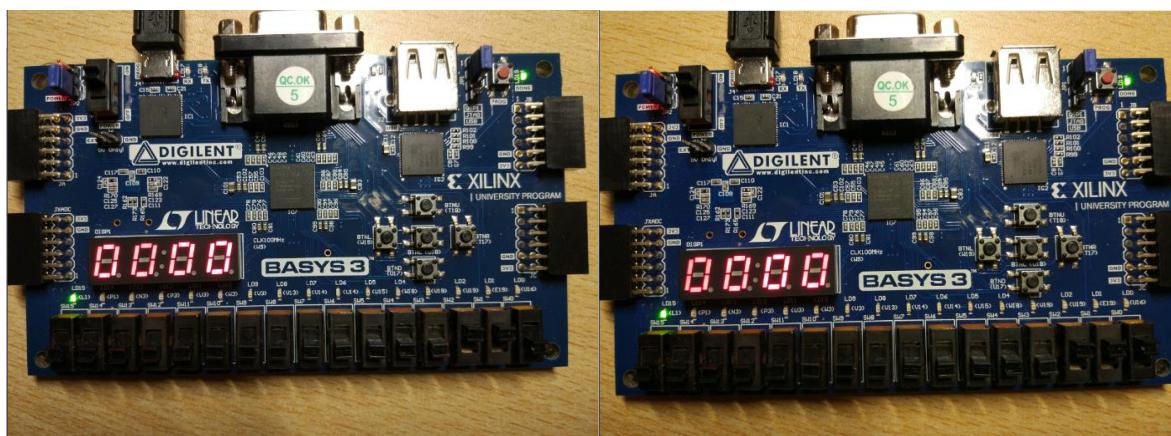
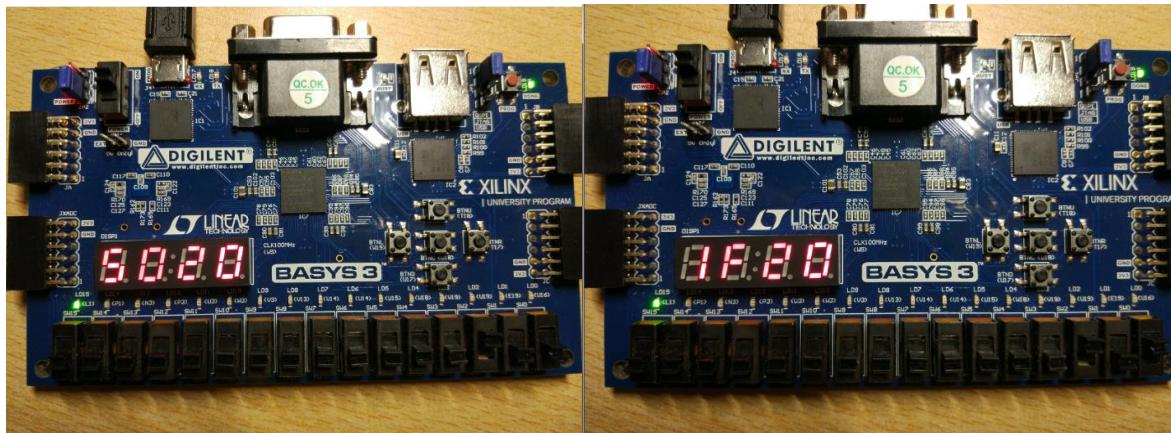


4.1.1. jr \$31

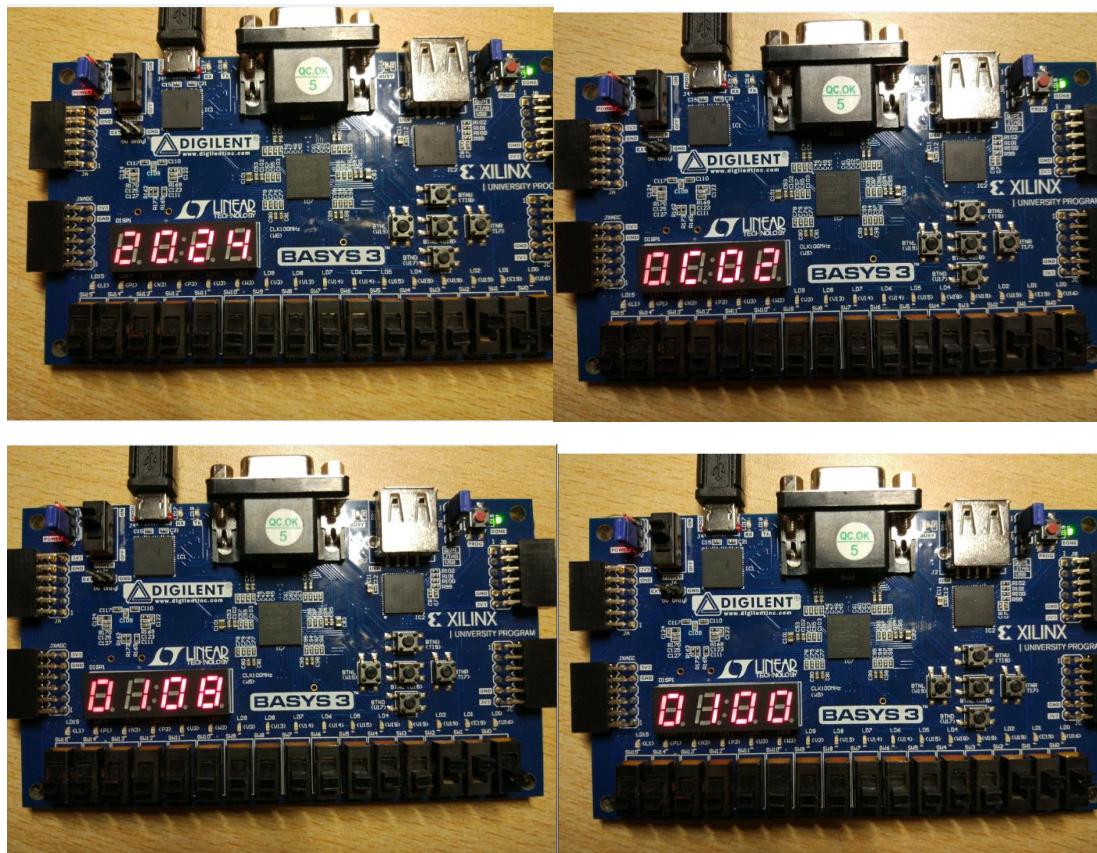
此为 IF 阶段



以下是 ID 阶段，可见下一个地址的变动（见以下九宫格左上角）



5. slt \$8,\$12,\$1



九、 实验心得

1. 项目编写心得

通过编写一个略微大型的硬件模块，大概知道了在设计硬件中我们所需要遵循的几个原则。

首先是模块化，层次化的重要性。

设计一个大型的硬件，代码中所涉及的线，寄存器，数量一定不小。若是不采用模块化，层次化的设计去帮助硬件简化设计，第一，编写硬件的人，很难在保证如此多线、寄存器的情况下依然能够编写的代码保证硬件的正常运行。第二，过于复杂的模块，会给 **debug** 带来很大的困难。在我设计 CPU 的时候，先使用数据通路图，梳理了 **top** 模块应有的连线，然后根据数据通路图，确定各子模块的功能，并编写测试。确保各子模块功能正确且完整后，才去编写 **top** 模块并测试。在这个过程中，虽然 CPU 是一个略微复杂的设计，但是每一个子模块的设计，其实难度并不高，这也是采用了模块化，层次化设计的优点所在。

其次，在每一个子模块写完之后，必须使用测试模块再进行测试。

之所以要对模块进行测试，是因为自己写的模块很多时候光看代码看不出错误。

Verilog 语言中，有很多隐藏的坑点，如我们可以隐式实例化线，这样子我们线的名字就算是写错了也不会报错。而这就会给你带来一个高阻态的 **bug**，怎么会有这样一个端口连接不上呢？类似这样的 **bug** 还有很多，而这样的 **bug**，很多时候可以通过子模块的单元测试来排除缩小范围。

Debug 技能，也在这一次设计中得到了不少的提升。

硬件设计中，仿真波形图出现与自己预想不一致的情况其实是一件很正常的事情。

考验我们的，是如何缩小问题范围，如何定位问题所在。在各种 **bug** 中，我摸索出了一套较为可行的 **debug** 方法。在仿真波形图中，我们从出现问题的信号入手，找到信号是从哪个模块传出来的，然后深入模块内部，检查预期信号到底是否成功生成。若成功生成，考虑模块与外部连线出了问题，如果连模块内部的信号都有相同的问题，那就从模块内部继续开始找。

2. 曾经遇到过的问题

2.1. 已解决：寄存器的写入：在下降沿写入还是上升沿写入？

其实老师给的代码中，寄存器是在时钟上升沿写入的。本来也没想多改，但是在仿真中发现，如果寄存器在时钟上升沿才写入，这个时候下一条指令已经拿上上一条指令的结果在算了，根本没有用到刚写入的值。因此为了解决这个问题，把寄存器的写入改为时钟下降沿触发。其实，时钟上升沿开始执行，时钟下降沿将结果写入，这样的安排更加适合让 CPU 有条不紊地运行。

2.2. 关于多周期 CPU 的时序控制

在一开始编写的时候，多周期 CPU 的时序控制是很混乱的。在一开始曾想过 **IR** 模块需要使用时钟下降沿触发。可是后来出现一个问题。在 **jal** 指令中，总是无法在 **sID** 阶段写回寄存器。之后仔细检查了一下仿真波形图才发现，如果 **IR** 模块采用时钟下降沿触发，那么 **Opcode** 的更新就必须等到时钟下降沿之后，而这就会导致本应该在下降沿读取写信号从而写寄存器\$31 的寄存器模块不能在时钟下降沿前得到写信号，从而导致 **jal** 指令无法写入寄存器。

为了解决这一个问题，我又重新思考了一下多周期 CPU 的时序控制。目前我的多

周期 CPU 时序控制如下：

sIF 阶段：

时钟上升沿到达，更新 **Control_unit**，更新连接 **PC** 模块的写信号，同时更新 **IR** 模块的写信号，为之后 **PC** 的写和 **IR** 模块的写做准备

时钟下降沿到达，**PC** 模块下降沿更新，写入新的地址。

sID 阶段：

时钟上升沿到达，更新 **Control_unit**，包括寄存器文件的写信号，同时更新 **IR** 模块。

这里 **IR** 是否能够更新，由上一个周期更新的 **IR** 模块的写信号来决定。

时钟下降沿到达，如果需要写入寄存器则在这个下降沿写入。

sEXE 阶段：

时钟上升沿到达，更新 **Control_unit**，更新 **ALU** 的操作码输入

时钟下降沿到达，更新 **ARD** 以及 **BDR**，从而更新 **ALU** 模块的操作数。

sMEM 阶段：

时钟上升沿到达：更新 **control_unit**

时钟下降沿到达：向存储器写入值

sWE 阶段：

时钟上升沿到达：更新 **Control_unit**，更新寄存器文件的写信号

时钟下降沿到达：向寄存器写入值。

2.3. 质疑：关于 **jal** 指令的设计

jal 指令的设计中，老师给出的设计方法，**jal** 指令只有两个时钟周期。在第一个时钟周期，**jal** 指令和别的指令一样，也只是更新 **pc**，但是第二个时钟周期，**jal** 指令不仅需要更新 **IR** 模块，还需要向\$31 寄存器写入 PC4 的值。当想到这里的时候我就有点疑问，因为秉承着简单的原则，一个时钟周期就应该只做一件事情，写寄存器这一件事情就应该只在 **sWE** 阶段完成呀。

想到这里，我先自己改了 **CPU** 的设计。我将 **jal** 指令的时钟周期增加到 3 个，并且给他增加了 **sWE** 阶段，这样子的实现，是没有 **bug** 的。

但是后来问老师的话，老师并不同意这一种做法，理由有两点：

第一点是，这样子增加了指令的时钟周期，会大大降低 **CPU** 的效率，这个在工业设计上是不允许的。、

第二点是，这样子的设计，已经经过了之前几届学生的验证，并没有错误，在 **mips** 的相关教材中也是这样的介绍的。

我虽然并没有完全被说服，但以增加代码复杂性作为代价换来的效率提高似乎也可以接受？后来就把设计改掉了。

2.4. Bug：烧板后寄存器无法被写入

当我完成 CPU 之后，进行烧板，可是烧板却出现了一个很严重的问题：无论我运行什么指令，寄存器总是写不进去。这个问题一度让我陷入死胡同，无法解决。

找了个时间专心 debug。我先从检查时钟触发开始，看看在时钟上升沿的时候寄存器写信号有没有及时更新？看看时钟下降沿能否被触发？我在 `top` 模块中接出了一条信号连到 `basy3` 实验板上观测寄存器写信号的更新，同时又把按键触发一次高电平改成了一次按键只改变时钟的电平。这就意味着我要按两次按键才能够完成一个完整的时钟周期。

这一次 debug 并没有找到原因。但缩小了问题的范围。在这个过程中，我发现我的时钟是没有问题的，无论是上升沿还是下降沿都能够被触发，同时寄存器的写信号也在应该的时候及时更新，但是即使是更新了写信号之后，在时钟下降沿依然无法将值写进寄存器中？

我仔细思考什么因素会影响寄存器的写入。时钟已经检查过了，写入的值也能够及时更新，控制能否写入的信号也没有问题，下面就只剩下一个了，控制写寄存器号的信号还没有检查？

把赌注押在了这一个模块，我在 CPU 输出的信号中，增加了查看写寄存器值的信号。当综合，实现，生成比特文件，烧板都完成之后，我惊讶的发现，无论是什么指令，写寄存器号一直都是 0！

应该是找到了问题所在，可是为什么一直都是 0 呢？一直没想明白。

无意间看了下 vivado 报的 warning，看到 vivado 提示我说有一个模块没有把所有端口连好。我检查了一下，有一个多选器的 `Reset` 端口没有在 `top` 模块上连接。心里一惊，万一烧板之后，这个 `Reset` 端口没有连接，就默认为 0 呢！

修改之后，果然是这一个原因。

最后，一句忠告：vivado 报的 warning 约等于 error，即使仿真过了，但是这些 warning 很容易会影响烧板后的实现的。

十、期末总结

这个学期，在“计算机组成与设计实验”这一门课程中，我完成的任务有：

1. mips 汇编程序设计
2. 单周期 CPU 设计
3. 多周期 CPU 设计

从这一门课程中，我的收获可以分为两个方面：一方面是作业的知识点，一方面是对硬件级开发的了解。

从作业的知识点来讲，在第一次作业，mips 汇编程序设计中，我写了一个比较简单的程序，用来实现冒泡排序。实现这个排序的想法来自于书本《计算机组成与设计 硬件/软件接口》中对汇编语言的介绍，在那本书中，详细的解释了在汇编程序设计中所涉及到的过程调用，堆栈使用，还有各种常见寄存器的使用。其中感悟最大的莫过于在过程调用中堆栈的使用了。在以前使用高级语言写程序的时候，我只是知道自动变量是会在栈中申请空间，会自动销毁，但是具体是如何申请空间的，内部的机制并不清楚。多亏使用过汇编语言，很多高级语言隐藏的细节，在汇编语言中写的清清楚楚。就如创建一个自动变量，就相当于先让栈指针 -8，然后再在相应的栈空间中写入相应的变量值。还有调用函数时，同样是使用栈，申请子过程的专属空间，从而子过程不会与父过程想干扰。在这样的前提下，递归地调用函数也成为了可能。

但是，汇编语言同样隐藏了很多我们平时不会仔细思考的细节。就如，汇编语言如何实现运行语句的跳转的？还有，各种指令到底是如何影响寄存器的值的？在计算机体系结构中，一层层的隔离，让我们方便的使用底层硬件给我们提供的各种接口。这样的封装方便了我们的使用，但是却让我们渐渐失去了了解底层硬件的动力。在单周期 CPU 的设计中，很多在汇编指令中隐藏的细节得以展现，我们因此知道，充当汇编语言与 CPU 之间中介的机器语言。我们因此也才知道，CPU 是如何读取机器语言，并识别指令的含义。各种在汇编语言中使用的寄存器，内存，在 CPU 的设计中，我们也才知道他们与时序的重要关系。

CPU 作为一个受着时钟控制的小芯片，我们平时 CPU 主频相信听得不少，但是我们对 CPU 中的时钟又有多少了解呢？严格来说，时钟并不在 CPU 内部，而是外来的，控制 CPU 状态的一组时序信号。但正是有了时钟信号，指令运行得以井然有序，内存读写得以摆脱竞争与冒险。当然，一切运行良好的前提，还在于 CPU 的时序设计是否优良。

CPU 的时序设计，一向是最难的地方。在单周期 CPU 中我们或许能够蒙混过关，但在老师的严格要求下，多周期 CPU 需要确保 PC，还有各存储器部件均需要时钟控制。这给我们带来了困难，但也让我们有了更加深入了解 CPU 时序设计的契机。时序设计中，最重要的事情是需要避免竞争与冒险。于是我最初的设计想法其实挺美好的：时钟上升沿更新控制信号，下降沿写各种寄存器与存储器。最初的仿真问题也还好，但是在其中的一条指令里出现了问题，jal 指令中，这样的时序设计并不完美。在前面的实验总结中，也说明了这一个问题。后来通过修改时序设计，修改了一点控制信号，问题才得以解决。

第二方面的感悟是硬件级的开发工作。仿真与烧板，这是一个在硬件级开发中常常面临的调试开发环境。我们在 basy3 实验板开发的时候，我们总是会遇这样的问题：明明仿真已经过了，但是烧到板上却不对！当遇到这类问题只会哀嚎而不去思考的话，我们就失去了一个进一步思考硬件级开发工作的机会。事实上，硬件中有着很多在仿真中不能模拟的特性，如信号的延迟（就如竞争与冒险），如部分线路的连接。我曾经在开发 CPU 的过程中，有一个模块没有连好线。这样子带来的结果就是虽然我在仿真中

得到了正确的结果，但是由于有一个端口并没有连线，状态并不确定，在烧到板上后，该端口的状态与仿真不一致便成为了使得 CPU 工作异常的重要原因。这也给了我一个启示，硬件的开发过程中，我们使用的 **verilog** 语言其实和普通的语言非常不一样。因为在这门语言的每一句，都意味着硬件的生成，而我们一旦在代码中，没有遵循规范，就很容易产生与我们预想不一样的硬件，从而产生错误。总结起来就是，必须遵循 **verilog** 的规范来实现硬件，仔细纠正正在综合，实现过程中产生的各种 **warning**。只有严格再严格，我们才能够高效的开发硬件，并且让硬件的运行处于可控状态中。