



《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 (2) 班

学生姓名 : 刘宇庭

学号 : 16340158

时间 : 2018 年 6 月 24 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: $rd \leftarrow rs + rt$

(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: $rt \leftarrow rs + (\text{sign-extend})immediate$

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

==>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 \leftarrow 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 $\leftarrow \{(pc+4)[31:28],addr[27:2],2'b00\}$; \$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

(17) halt (停机指令)

111111	000000000000000000000000000000(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。

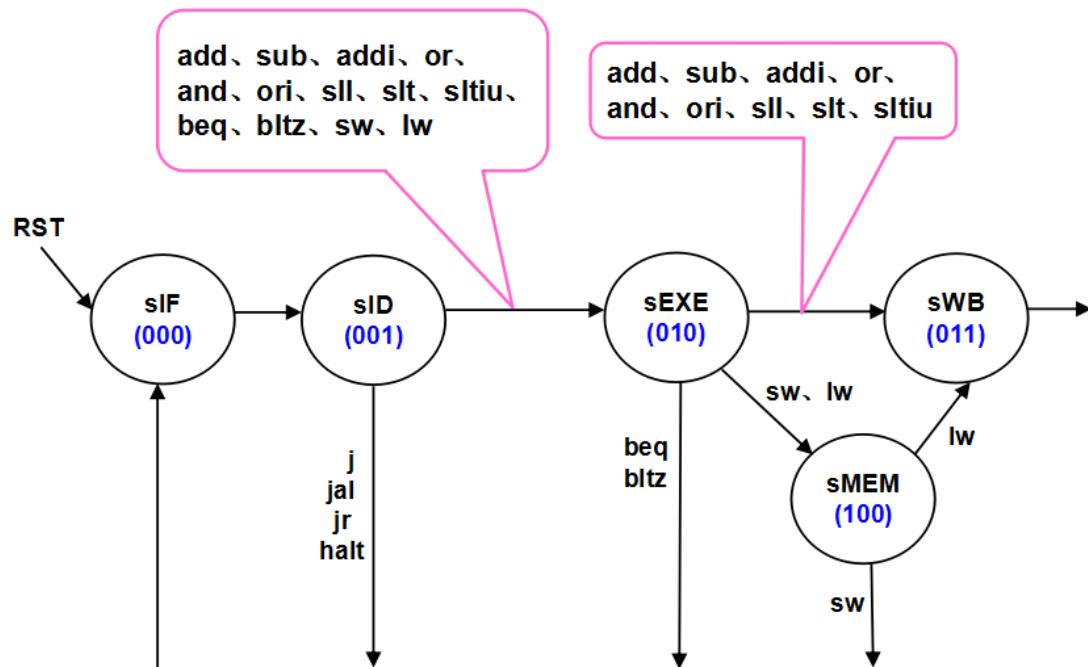


图 2 多周期 CPU 状态转移图

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。

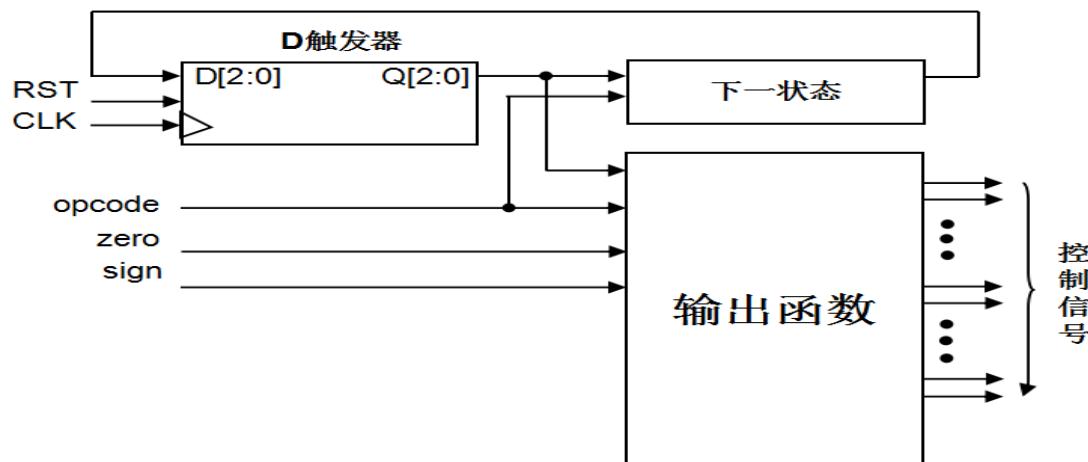


图 3 多周期 CPU 控制部件的原理结构图

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。

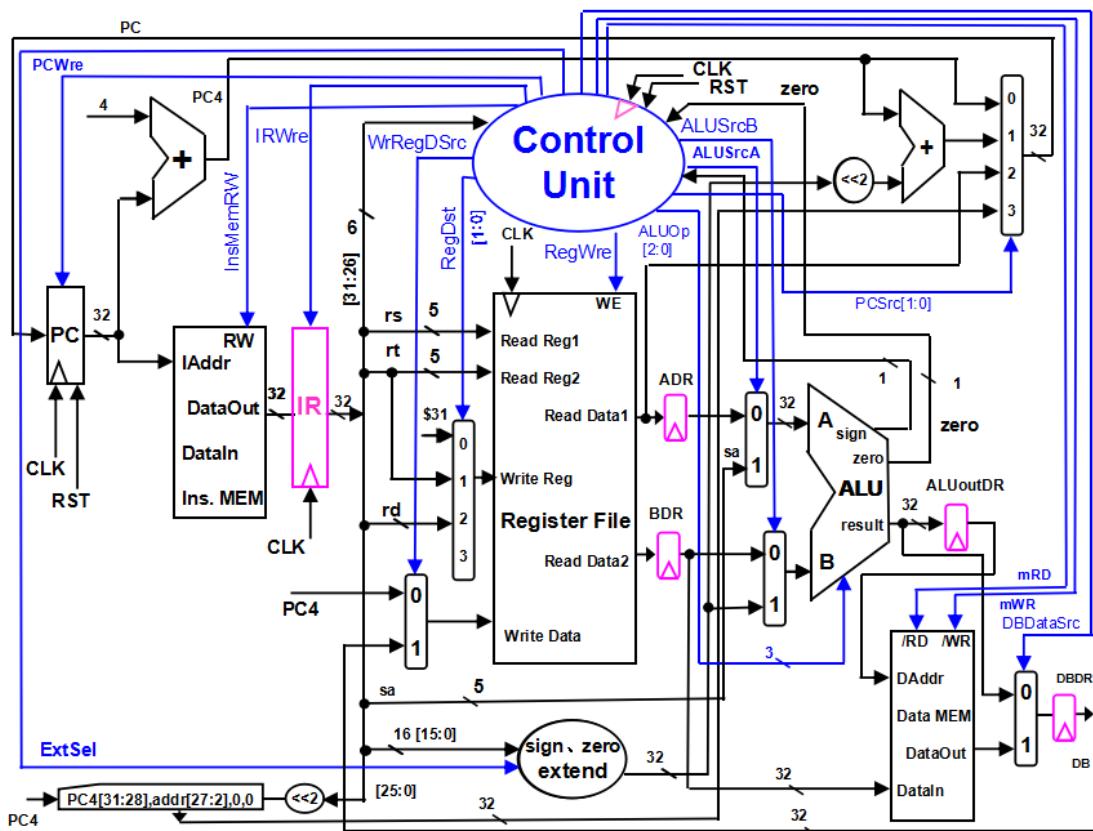


图 4 多周期 CPU 数据通路和控制线路图

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	$Y = B \ll A$	B 左移 A 位
101	$Y = A \vee B$	或
110	$Y = A \wedge B$	与
111	$Y = A \oplus B$	异或

值得注意的问题, 设计时, 用模块化、层次化的思想方法设计, 关于如何划分模块、如何整合成一个系统等等, 是必须认真考虑的问题。

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

设计思路以及设计流程:

① 控制信号、指令以及执行状态相互关系的过程

理清控制信号、指令以及执行状态三者之间的关联是设计多周期 CPU 的第一个步

骤，也是主要难点之一。在多周期 CPU 中，部分控制信号遇到在特定的状态下控制单元才对该控制信号进行相应的改变，而部分信号则与当前执行状态无关，可以在取指令阶段时就赋予需要的值。例如，对于任何一条指令，信号量 RegWre 的状态时比较需要注意的，假如再对于需要写寄存器组的指令一开始就对所有信号量赋予相应的值，那么寄存器组的写使能端信号量 RegWre 为使能状态，此时，在指令执行处理过程中，可能不止在结果写回的时候写入数据，在指其余的执行过程中都可能发生寄存器组写入数据，这时候可能导致原来寄存器组一些数据被修改替换了，再次使用的时候得到的结果与预期的就不一样了，存在比较大的危害性。同时指令的执行不一定需要所有的模块，比如跳转指令，其无需对数据寄存器进行读写操作，则数据寄存器的控制信号 mRD、mWR 都无需使用到，因此为了防止出现一些不必要的错误，统一将指令相对应的无关的使能控制信号（x）默认为低电平（0），无需 ALU 运算的（例如跳转指令）默认将其操作变为加操作（000）。

状态	指令	信号量													
		PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WrRegDSrc	InsMemRW	mRD	mWR	IRWre	ExtSel	PCSrc[1:0](zero 0/1)	RegDst[1:0]	ALUOp[3:0]
sIF(000)	ins	1	x	x	x	0	x	1	x	x	1	x	xx	xx	xxx
	halt	0	x	x	x	0	x	0	x	x	0	x	xx	xx	xxx
sID(001)	ins	0	x	x	x	0	x	0	x	x	0	x	xx	xx	xxx
	j	0	x	x	x	0	x	0	x	x	0	x	11	xx	xxx
	jal	0	x	x	x	1	0	0	x	x	0	x	11	00	xxx
	jr	0	x	x	x	0	x	0	x	x	0	x	10	xx	xxx
	halt	0	x	x	x	0	x	0	x	x	0	x	xx	xx	xxx
sEXE(010)	addi	0	0	1	x	0	x	0	x	x	0	1	00	xx	000
	ori	0	0	1	x	0	x	0	x	x	0	0	00	xx	101
	or	0	0	0	x	0	x	0	x	x	0	x	00	xx	101
	sub	0	0	0	x	0	x	0	x	x	0	x	00	xx	001
	and	0	0	0	x	0	x	0	x	x	0	x	00	xx	110
	sll	0	1	0	x	0	x	0	x	x	0	0	00	xx	100
	beq	0	0	0	x	0	x	0	x	x	0	1	00/01	xx	001
	slt	0	0	0	x	0	x	0	x	x	0	x	00	xx	011
	sltiu	0	0	1	x	0	x	0	x	x	0	0	00	xx	010
	bltz	0	0	0	x	0	x	0	x	x	0	1	01/00	xx	001
	sw	0	0	1	x	0	x	0	x	x	0	1	00	xx	000
	lw	0	0	1	x	0	x	0	x	x	0	1	00	xx	000
	sw	0	x	x	0	0	x	0	0	0	1	0	x	xx	xxx
sMEM(100)	lw	0	x	x	1	1	x	0	1	0	0	x	xx	xx	xxx
	addi	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx
	ori	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx
	or	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx
	sub	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx
	and	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx
	sll	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx
sWB(011)	beq	0	x	x	0	0	1	0	x	x	0	x	xx	11	xxx
	slt	0	x	x	0	1	1	0	x	x	0	x	xx	10	xxx
	sltiu	0	x	x	0	1	1	0	x	x	0	x	xx	01	xxx
	bltz	0	x	x	0	0	1	0	x	x	0	x	xx	11	xxx
	sw	0	x	x	0	0	1	0	x	x	0	x	xx	11	xxx

表3 控制信号、指令以及执行状态之间的相互关系

完成控制信号、指令以及执行状态的关系表以后，对于如何实现多周期 CPU 依旧很困惑，主要不清楚如何将一条分多个时钟周期执行并且能够保持正确，以及如何保证中间过程不会使用到非对应的数据或在寄存器中写入了错误数据。此时，思考实验原理中的图 2 多周期 CPU 状态转移图，总结多周期 CPU 状态情况，五个执行状态并非所有的指令都有，则我们需要创建一个状态机，具体如实验原理中的图 3 多周期 CPU 控制部件的原理结构图，依据指令的操作码以及当前的 CPU 状态，在时钟的上升沿触发得到下一个 CPU 状态，同时控制信号亦依据当前状态进行修改。例如，在指令写回寄存器组阶段--sWB 阶段的时候，控制信号 RegWre 为 1，即寄存器组可以写此时，至于其他的执行阶段，则为 0，即寄存器组使能端失效，不能写，依据当前 CPU 的状态设置控制信号可以有效的避免写入错误数据入寄存器组。结合实验原理中的图 4 多周期 CPU 数据通路和控制线路图，首先将其模块

化，分成多个模块相连起来，同时与之前设计的单周期 CPU 不同的是，需要添加一些临时数据的寄存器，其中寄存器 ADR 和 BDR 用来保留寄存器组中 rs 和 rt 中相对应的内容，寄存器 ALUoutDR 用来保留 ALU 的运算结果，寄存器 DBDR 用来保留写回数据，这些寄存器的主要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，大延迟变成多个分段小延迟。同时，为了使指令代码保持稳定，需要增加指令寄存器 IR。

② 多周期 CPU 模块划分与实现

依据图 2 多周期 CPU 数据通路和控制线路图，将 CPU 划分为 14 个模块。其中没有具体细分每个功能模块，将一些数据选择的模块并入到需要的功能模块中，没有完全依据多周期 CPU 数据通路图进行划分，否则需要过多模块，划分的太过冗余。模块划分结果如图三所示。

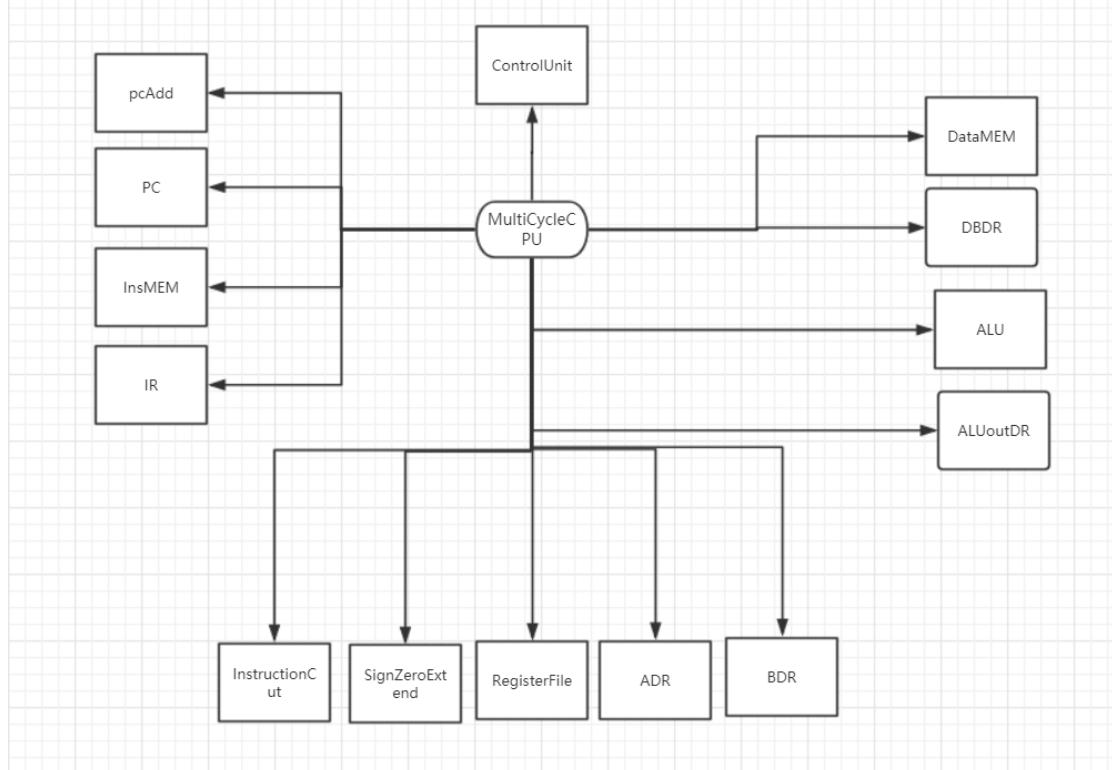


图 3 多周期 CPU 模块划分图

模块介绍：

1. pcAdd

模块功能：根据控制信号 PCSrc，计算获得下一个pc以及控制信号 Reset 重置。

实现思路：首先决定何时引起触发，将里面涉及的信号量作为敏感变量，主要是为了确保下一条pc能够正确得到。

主要实现代码：

```

always@(*)
begin
    if(!RST) begin
        nextPC = 0;
    end
    else begin
        pc = curPC + 4;
        case(PCSrc)
            2'b00: nextPC = curPC + 4;
            2'b01: nextPC = curPC + 4 + immediate * 4;
            2'b10: nextPC = rs;
            2'b11: nextPC = {pc[31:28], addr, 2'b00};
        endcase
    end
end

```

2. PC

模块功能: 根据控制信号PCWre, 判断pc是否改变以及根据Reset信号判断是否重置。

实现思路: 将时钟信号的上升沿和控制信号Reset作为敏感变量, 使得pc在上升沿的时候发生改变或被重置。

主要实现代码:

```

always@(posedge CLK or negedge RST)
begin
    if(!RST) // Reset == 0, PC = 0
        begin
            curPC <= 0;
        end
    else
        begin
            if(PCWre) // PCWre == 1
                begin
                    curPC <= nextPC;
                end
            else // PCWre == 0, halt
                begin
                    curPC <= curPC;
                end
        end
    end

```

3. InsMEM

模块功能: 依据当前pc和信号量InsMemRW, 读取指令寄存器中, 相对应地址的指令。

实现思路: 将pc的输入作为敏感变量, 当pc发生改变的时候, 则进行指令的读取, 根

据相关的地址，输出指令寄存器中相对应的指令。

主要实现代码：

```

reg [7:0] rom[128:0]; // 存储器定义必须用reg类型，存储器存储单元8位长度，共128个存储单元，可以存32条指令

// 加载数据到存储器rom。注意：必须使用绝对路径
initial
begin
    $readmemh("F:\\Vivado\\MultiCycleCPU\\romData.txt", rom);
end

//大端模式
always@(IAddr or InsMemRW)
begin
    //取指令
    if(InsMemRW)
        begin
            IDataOut[7:0] = rom[IAddr + 3];
            IDataOut[15:8] = rom[IAddr + 2];
            IDataOut[23:16] = rom[IAddr + 1];
            IDataOut[31:24] = rom[IAddr];
        end
end

```

4. IR

模块功能：为了使指令代码保持稳定。

实现思路：将时钟上升沿作为敏感信号，同时依据信号量 IRWre，对 IR 寄存器进行写入。

主要实现代码：

```

always@(posedge CLK)
begin
    if(IRWre) begin
        IRIInstruction <= instruction;
    end
end

```

5. InstructionCut

模块功能：对指令进行分割，获得相对应的指令信息。

实现思路：根据各种类型的指令结构，将指令分割，得到相对应的信息。

主要实现代码：

```

always@(instruction)
begin
    op = instruction[31:26];
    rs = instruction[25:21];
    rt = instruction[20:16];
    rd = instruction[15:11];
    sa = instruction[10:6];
    immediate = instruction[15:0];
    addr = instruction[25:0];
end

```

6. ControlUnit

模块功能：控制单元，依据指令的操作码（op）、标记符（ZERO）以及当前 CPU 状态，依据表三 控制信号、指令以及执行状态之间的相互关系，输出相匹配控制信号量。

实验思路：设计一个 CPU 的状态机，依据指令的操作码（op）、当前状态以及重置信号（RST），其状态在上升沿的时候发生，同时依据当前的状态、操作码（op）以及标记符（ZERO）修改并且输出控制信号。本模块为多周期 CPU 中最重要的模块，需要注意部分信号在特定的 CPU 状态才能输出相应的使能，否则将出现错误。

主要实现代码：

```

1. reg [2:0] state, nextState;      //记录状态
2. parameter [2:0] iniState = 3'b111,
3.                      sIF = 3'b000,
4.                      sID = 3'b001,
5.                      sEXE = 3'b010,
6.                      sMEM = 3'b100,
7.                      sWB = 3'b011;
8.
9. //状态机
10. always@(posedge CLK) begin
11.     if(!RST) begin
12.         state <= sIF;
13.     end else begin
14.         state <= nextState;
15.     end
16. end
17.
18. always@(state or op or zero) begin
19.     // 状态更新
20.     case(state)
21.         iniState : nextState = sIF;
22.         sIF: nextState = sID;
23.         sID: begin
24.             case(op[5:3])
25.                 3'b111: nextState = sIF;      //指令 j,jal,jr,halt

```

```
26.         default: nextState = sEXE;
27.     endcase
28. end
29. sEXE: begin
30.     if((op == 6'b110100) || (op == 6'b110110)) begin
31.         //beq,bltz
32.         nextState = sIF;
33.     end else if(op == 6'b110000 || op == 6'b110001) begin
34.         //sw,lw
35.         nextState = sMEM;
36.     end else begin
37.         nextState = sWB;
38.     end
39. end
40. sMEM: begin
41.     if(op == 6'b110000) begin
42.         //sw
43.         nextState = sIF;
44.     end else begin
45.         //lw
46.         nextState = sWB;
47.     end
48. end
49. sWB: nextState = sIF;
50. endcase
51.
52. // 信号量
53. // PCWre and InsMemRW
54. if(nextState == sIF && op != 6'b111111 && state != iniState) begin
55.     // halt
56.     PCWre = 1;
57.     InsMemRW = 1;
58. end else begin
59.     PCWre = 0;
60.     InsMemRW = 0;
61. end
62.
63. // IRWre
64. if(state == sIF || nextState == sID) begin
65.     IRWre = 1;
66. end else begin
67.     IRWre = 0;
68. end
69.
```

```

70.    // ALUSrcA
71.    if(op == 6'b011000) begin
72.        // sll
73.        ALUSrcA = 1;
74.    end else begin
75.        ALUSrcA = 0;
76.    end
77.
78.    // ALUSrcB
79.    if(op == 6'b000010 || op == 6'b010010 || op == 6'b110000 || op == 6'b110
001 || op == 6'b100111) begin
80.        // addi,ori,sw,lw,sltui
81.        ALUSrcB = 1;
82.    end else begin
83.        ALUSrcB = 0;
84.    end
85.
86.    // DBDataSrc
87.    if(op == 6'b110001) begin
88.        // lw
89.        DBDataSrc = 1;
90.    end else begin
91.        DBDataSrc = 0;
92.    end
93.
94.    // RegWre and WrRegDSrc and RegDst
95.    if((state == sWB && op != 6'b110100 && op != 6'b110000 && op != 6'b11011
0) || (op == 6'b111010 && state == sID)) begin
96.        // 非beq, sw, blitz
97.        RegWre = 1;
98.        if(op == 6'b111010) begin
99.            // jal
100.            WrRegDSrc = 0;
101.            RegDst = 2'b00;
102.        end else begin
103.            WrRegDSrc = 1;
104.            if(op == 6'b000010 || op == 6'b010010 || op == 6'b100111 || op
== 6'b110001) begin
105.                // addi, ori, sltui, lw
106.                RegDst = 2'b01;
107.            end else begin
108.                // add, sub, or, and, slt, sll
109.                RegDst = 2'b10;
110.            end

```

```

111.      end
112.    end else begin
113.      RegWre = 0;
114.    end
115.
116.    // InsMemRW
117.    if(op != 6'b111111)
118.      InsMemRW = 1;
119.
120.    // mRD
121.    mRD = (op == 6'b110001) ? 1 : 0; // lw
122.
123.    // mWR
124.    mWR = (state == sMEM && op == 6'b110000) ? 1 : 0; // sw
125.
126.    // ExtSel
127.    ExtSel = (op == 6'b000010 || op == 6'b110001 || op == 6'b110000 || op =
128.      = 6'b110100 || op == 6'b110110) ? 1 : 0; // addi、lw、sw、beq、bltz
129.
130.    // PCSrc
131.    if(op == 6'b111001) begin
132.      // jr
133.      PCSrc = 2'b10;
134.    end else if((op == 6'b110100 && zero) || (op == 6'b110110 && !zero)) be
135.      gin
136.      // beq 和 bltz 跳转
137.      PCSrc = 2'b01;
138.    end else if(op == 6'b111010 || op == 6'b111000) begin
139.      // j,jal
140.      PCSrc = 2'b11;
141.    end else begin
142.      PCSrc = 2'b00;
143.
144.    // ALUOp
145.    case(op)
146.      6'b000010: ALUOp = 3'b000; // addi
147.      6'b010010: ALUOp = 3'b101; // ori
148.      6'b010000: ALUOp = 3'b101; // or
149.      6'b000001: ALUOp = 3'b001; // sub
150.      6'b010001: ALUOp = 3'b110; // and
151.      6'b011000: ALUOp = 3'b100; // sll
152.      6'b110100: ALUOp = 3'b001; // beq
153.      6'b100110: ALUOp = 3'b011; // slt

```

```

153.      6'b100111: ALUOp = 3'b010; // sltiu
154.      6'b110110: ALUOp = 3'b001; // bltz
155.      6'b110001: ALUOp = 3'b000; // sw
156.      6'b110000: ALUOp = 3'b000; // lw
157.    endcase
158. end

```

7. RegisterFile

模块功能：寄存器组，通过控制单元输出的控制信号，进行相对应的读或写操作。

实现思路：当ReadReg1或者ReadReg2发生改变的时候，即对寄存器组进行数据读取。

至于数据写入，则选择在时钟的下降沿时候进行操作，同时写入信号（RegWre）必须为1且写入寄存器的地址不能为\$0。至于数据的写入地址则在任意信号发生改变则依据写入地址的控制信号（RegDst）进行修改，使得其在数据写回阶段确保为正确的写入地址。

主要实现代码：

```

reg [31:0] regFile[0:31]; // 寄存器定义必须用reg类型
integer i;
initial begin
    for (i = 0; i < 32; i = i + 1) regFile[i] <= 0;
end

always@(ReadReg1 or ReadReg2)
begin
    ReadData1 = regFile[ReadReg1];
    ReadData2 = regFile[ReadReg2];
    //$display("regfile %d %d\n", ReadReg1, ReadReg2);
end

always@(negedge CLK)
begin
    // $0恒为0，所以写入寄存器的地址不能为0
    if(RegWre && WriteReg)
        begin
            regFile[WriteReg] <= WriteData;
        end
end

always@(*)
begin
    case(RegDst)
        2'b00: WriteReg = 31;
        2'b01: WriteReg = ReadReg2;
        2'b10: WriteReg = rd;
    endcase;
end

```

8. SignZeroExtend

模块功能：根据指令相关的控制信号ExtSel，对立即数进行扩展。

实现思路：根据控制信号ExtSel判断是0扩展还是符号扩展，然后进行相对应的扩展

主要实现代码：

```
assign extendImmediate[15:0] = immediate;
assign extendImmediate[31:16] = ExtSel ? (immediate[15] ? 16'hffff : 16'h0000) : 16'h0000;
```

9. ALU

模块功能：算术逻辑单元，对两个输入依据ALUOp进行相对应的运算。

实现思路：依据实验原理中的ALU运算功能表(表2)完成操作码对应的操作，当ReadData1、ReadData2、ALUSrcA、ALUSrcB、ALUOp任意发生改变的时候，即进行运算。

主要实现代码：

```
reg [31:0] A;
reg [31:0] B;

initial begin
    result = 0;
    zero = 0;
end

always@(ReadData1 or ReadData2 or ALUSrcA or ALUSrcB or ALUOp)
begin
    //定义两个输入端口
    A = (ALUSrcA == 0) ? ReadData1 : sa;
    B = (ALUSrcB == 0) ? ReadData2 : extend;
    case(ALUOp)
        3'b000: result = A + B;
        3'b001: result = A - B;
        3'b010: result = (A < B) ? 1 : 0;
        3'b011: result = (((ReadData1 < ReadData2) && (ReadData1[31] == ReadData2[31])) || ((ReadData1[31] == 1 && ReadData2[31] == 0))) ? 1:0;
        3'b100: result = B << A;
        3'b101: result = A | B;
        3'b110: result = A & B;
        3'b111: result = A ^ B;
    endcase
    zero = (result == 0) ? 1 : 0;
end
```

10. DataMEM

模块功能：数据存储器，通过控制信号，对数据寄存器进行读或者写操作，并且此处模块额外合并了输出DB的数据选择器，此模块同时输出写回寄存器组的数据DB。

实现思路：将相关信号量作为敏感变量，假如读数据寄存器的信号量（mRD）为1的时候，则对数据寄存器进行读操作，假如写数据寄存器的信号量（mWR）为1的时候，则对数据寄存器进行写操作。

主要实现代码：

```

reg [7:0] ram [0:31];      // 存储器定义必须用reg类型

always@(mRD or DAddr or DBDataSrc)
begin
    //读
    DataOut[7:0] = mRD ? ram[DAddr + 3] : 8'bzz; // z 为高阻态
    DataOut[15:8] = mRD ? ram[DAddr + 2] : 8'bzz;
    DataOut[23:16] = mRD ? ram[DAddr + 1] : 8'bzz;
    DataOut[31:24] = mRD ? ram[DAddr] : 8'bzz;

    DB = (DBDataSrc == 0) ? DAddr : DataOut;
end

always@(mWR or DAddr)
begin
    //写
    if(mWR)
        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
    //$/display("mwr: %d $12 %d %d %d", mWR, ram[12], ram[13], ram[14], ram[15]);
end

```

11. 寄存器 ADR、BDR、ALUoutDR、DBDR

模块功能：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。

实现思路：在时钟上升沿将相应的数据写入寄存器中。四个寄存器功能相同，可以使用一个模块实例得到。

主要实现代码：

```

1. module TempReg(
2.     input CLK,
3.     input [31:0] IData,
4.     output reg[31:0] OData
5. );
6.
7.     initial begin
8.         OData = 0;
9.     end
10.
11.    always@(posedge CLK) begin
12.        OData <= IData;

```

```

13.      end
14. endmodule

```

顶层模块：MultiCycleCPU

实现思路：在顶层模块中将各个已实现的底层模块进行实例，并且用verilog语言将各个模块用线连接起来

实例模块：

```

1. ControlUnit ControlUnit(.CLK(CLK),
2.                         .RST(RST),
3.                         .zero(zero),
4.                         .op(op),
5.                         .IRWre(IRWre),
6.                         .PCWre(PCWre),
7.                         .ExtSel(ExtSel),
8.                         .InsMemRW(InsMemRW),
9.                         .WrRegDSrc(WrRegDSrc),
10.                        .RegDst(RegDst),
11.                        .RegWre(RegWre),
12.                        .ALUSrcA(ALUSrcA),
13.                        .ALUSrcB(ALUSrcB),
14.                        .PCSsrc(PCSsrc),
15.                        .ALUOp(ALUOp),
16.                        .mRD(mRD),
17.                        .mWR(mWR),
18.                        .DBDataSrc(DBDataSrc));
19.

20. pcAdd pcAdd(.RST(RST),
21.                 .PCSsrc(PCSsrc),
22.                 .immediate(extend),
23.                 .addr(addr),
24.                 .curPC(curPC),
25.                 .rs(A),
26.                 .nextPC(nextPC));
27.

28. PC PC(.CLK(CLK),
29.          .RST(RST),
30.          .PCWre(PCWre),
31.          .nextPC(nextPC),
32.          .curPC(curPC));
33.

34. InsMEM InsMEM(.IAddr(curPC),

```

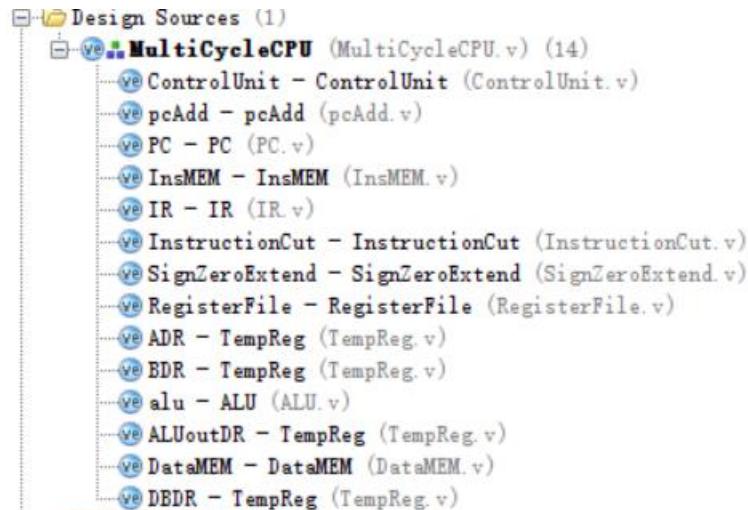
```
35.           .InsMemRW(InsMemRW),
36.           .IDataOut(instruction));
37.
38. IR IR(.instruction(instruction),
39.         .CLK(CLK),
40.         .IRWre(IRWre),
41.         .IRInstruction(IRInstruction));
42.
43. InstructionCut InstructionCut(.instruction(IRInstruction),
44.                                 .op(op),
45.                                 .rs(rs),
46.                                 .rt(rt),
47.                                 .rd(rd),
48.                                 .sa(sa),
49.                                 .immediate(immediate),
50.                                 .addr(addr));
51.
52. SignZeroExtend SignZeroExtend(.immediate(immediate),
53.                               .ExtSel(ExtSel),
54.                               .extendImmediate(extend));
55.
56. RegisterFile RegisterFile(.CLK(CLK),
57.                           .ReadReg1(rs),
58.                           .ReadReg2(rt),
59.                           .rd(rd),
60.                           .WriteData(WrRegDSrc ? dataDB : curPC + 4),
61.                           .RegDst(RegDst),
62.                           .RegWre(RegWre),
63.                           .ReadData1(A),
64.                           .ReadData2(B),
65.                           .WriteReg(WriteReg));
66.
67. TempReg ADR(.CLK(CLK),
68.               .IData(A),
69.               .OData(dataA));
70.
71. TempReg BDR(.CLK(CLK),
72.               .IData(B),
73.               .OData(dataB));
74.
75. ALU alu(.ALUSrcA(ALUSrcA),
76.           .ALUSrcB(ALUSrcB),
77.           .ReadData1(dataA),
78.           .ReadData2(dataB),
```

```

79.         .sa(sa),
80.         .extend(extend),
81.         .ALUOp(ALUOp),
82.         .zero(zero),
83.         .result(result));
84.
85. TempReg ALUoutDR(.CLK(CLK),
86.                     .IData(result),
87.                     .OData(dataResult));
88.
89. DataMEM DataMEM(.mRD(mRD),
90.                     .mWR(mWR),
91.                     .DBDataSrc(DBDataSrc),
92.                     .DAddr(result),
93.                     .DataIn(dataB),
94.                     .DataOut(DataOut),
95.                     .DB(DB));
96.
97. TempReg DBDR(.CLK(CLK),
98.                     .IData(DB),
99.                     .OData(dataDB));

```

文件结构:



③ 多周期 CPU 正确性验证

a) 仿真文件

i. 实例一个多周期 CPU

```

    MultiCycleCPU uut (.CLK(CLK),
                         .RST(RST),
                         .curPC(curPC),
                         .nextPC(nextPC),
                         .instruction(instruction),
                         .IRInstruction(IRInstruction),
                         .op(op),
                         .rs(rs),
                         .rt(rt),
                         .rd(rd),
                         .DB(DB),
                         .dataDB(dataDB),
                         .A(A),
                         .dataA(dataA),
                         .B(B),
                         .dataB(dataB),
                         .result(result),
                         .dataResult(dataResult),
                         .PCSrc(PCSsrc),
                         .zero(zero),
                         .PCWre(PCWre),
                         .ExtSel(ExtSel),
                         .InsMemRW(InsMemRW),
                         .RegDst(RegDst),
                         .RegWre(RegWre),
                         .ALUSrcA(ALUSrcA),
                         .ALUSrcB(ALUSrcB),
                         .ALUOp(ALUOp),
                         .mRD(mRD),
                         .mWR(mWR),
                         .DBDataSrc(DBDataSrc),
                         .WrRegDSrc(WrRegDSrc),
                         .WriteReg(WriteReg)
);

```

ii. 初始化并产生时钟信号

```

initial begin
    CLK = 0;
    RST = 0;
    #20 RST = 1;
    forever #20 CLK = ~CLK;
end

```

b) 测试

地址	汇编程序	指令代码					16进制数代码
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)		
0x00000000	addi \$1,\$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002

0x00000008	or \$3, \$2, \$1	010000	00010	00001	0001 1000 0000 0000	=	40411800
0x0000000C	sub \$4, \$3, \$1	000001	00011	00001	0010 0000 0000 0000	=	04612000
0x00000010	and \$5, \$4, \$2	010001	00100	00010	0010 1000 0000 0000	=	44822800
0x00000014	sll \$5, \$5, 2	011000	00000	00101	0010 1000 1000 0000		60052880
0x00000018	beq \$5, \$1, -2(=, 转 14)	110100	00101	00001	1111 1111 1111 1110		D0A1FFFE
0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000		E8000010
0x00000020	slt \$8, \$12, \$1	100110	01100	00001	0100 0000 0000 0000		99814000
0x00000024	addi \$13, \$0, -2	000010	00000	01101	1111 1111 1111 1110		080DFFFE
0x00000028	slt \$9, \$8, \$13	100110	01000	01101	0100 1000 0000 0000		990D4800
0x0000002C	sltiu \$10, \$9, 2	100111	01001	01010	0000 0000 0000 0010		9D2A0002
0x00000030	sltiu \$11, \$10, 0	100111	01010	01011	0000 0000 0000 0000		9D4B0000
0x00000034	addi \$13, \$13, 1	000010	01101	01101	0000 0000 0000 0001		09AD0001
0x00000038	bltz \$13, -2 (<0, 转 34)	110110	01101	00000	1111 1111 1111 1110		D9A0FFFE
0x0000003C	j 0x000004C	111000	00000	00000	0000 0000 0001 0011		E0000013
0x00000040	sw \$2, 4(\$1)	110000	00001	00010	0000 0000 0000 0100		C0220004
0x00000044	lw \$12, 4(\$1)	110001	00001	01100	0000 0000 0000 0100		C42C0004
0x00000048	jr \$31	111001	11111	00000	0000 0000 0000 0000		E7E00000
0x0000004C	halt	111111	00000	00000	00000000000000000000	=	FC000000
0x00000050							
0x00000054							

表 4 程序测试段

使用表 4 进行测试 CPU 正确性，将其中的指令写入一个文件 romData.txt。在模块 InsMEM 中进行读入（使用的路径为绝对路径）

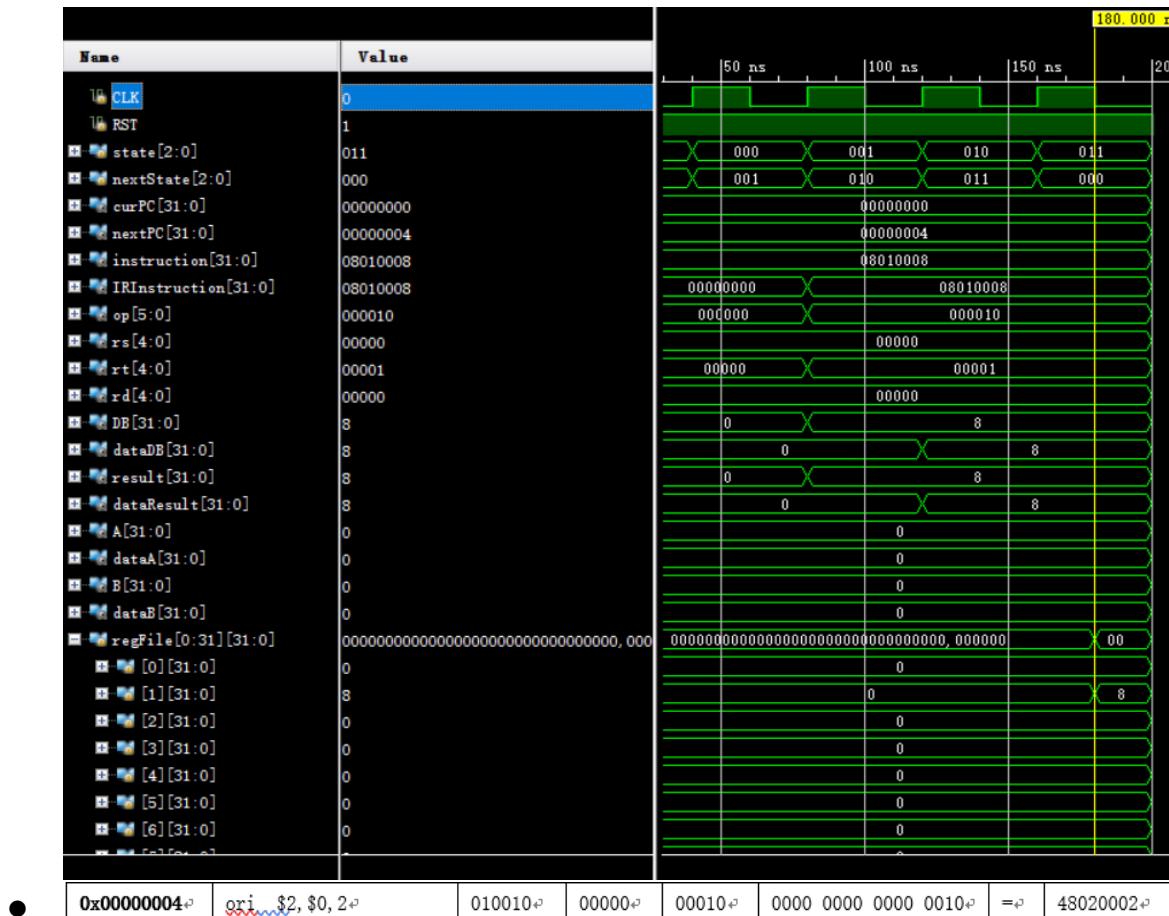
```
// 加载数据到存储器rom。注意：必须使用绝对路径
initial
begin
    $readmemh("F:\\Vivado\\MultiCycleCPU\\romData.txt", rom);
end
```

仿真测试验证

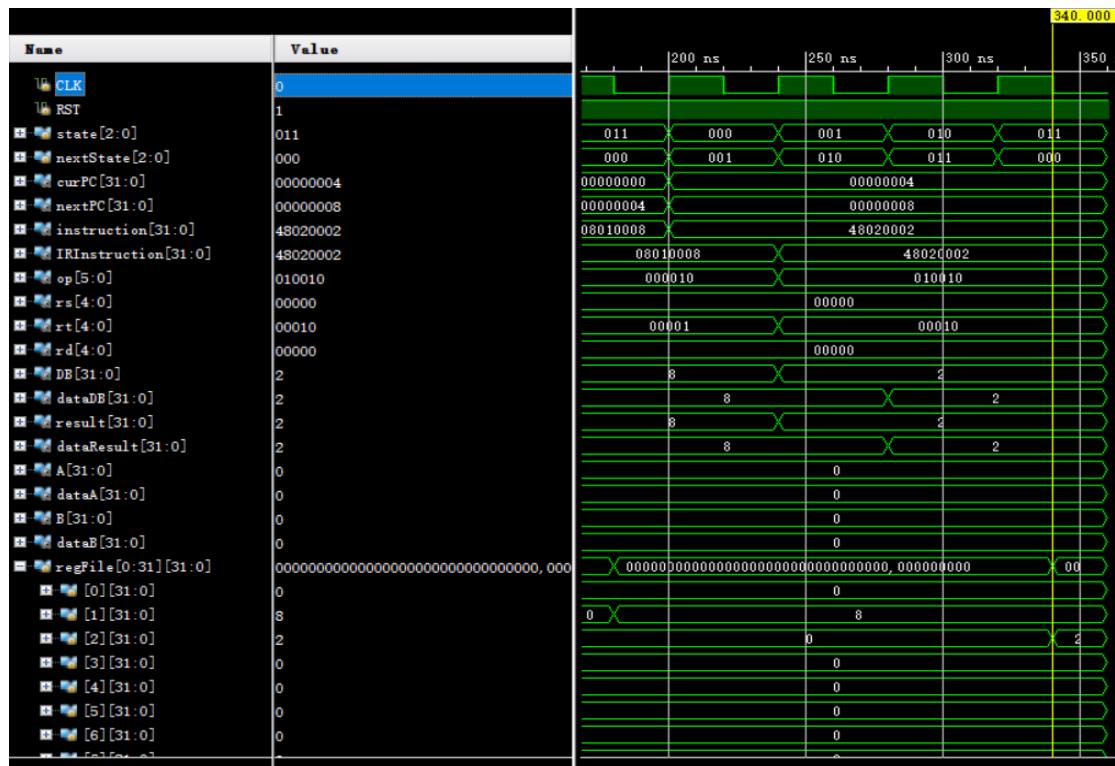
(注：仿真输出中 A、B 为寄存器组的两个输出并非 ALU 的两个输入，state 为当前 CPU 状态，nextState 为下个 CPU 状态，,curPC 为当前 PC，nextPC 为下一条 PC，其余与上文中的控制信号对应，指令 instruction、curPC、nextPC、IRInstruction 为十六进制输出，ALU 计算结果 result 和写回数据 DB 以及 dbA,dbB,dbResult,dbDB 为有符号整数输出，dbA,dbB,dbResult,dbDB 分别为 ADR、BDR、ALUopDR、DBDR 寄存器存储的数据，IRInstruction 为指令寄存器存储的数据，数据寄存器 ram 没有进行初始化，寄存器组 regFile 都初始为 0)

●	0x00000000	addi \$1, \$0, 8	000010	00000	00001	0000 0000 0000 1000	=	08010008
---	------------	------------------	--------	-------	-------	---------------------	---	----------

指令为加法操作指令，将 0 号寄存器的数据 (0) 与立即数 8 相加，并且将运算结果 (8) 写回 1 号寄存器。需要取指令，指令译码，指令执行，结果写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。

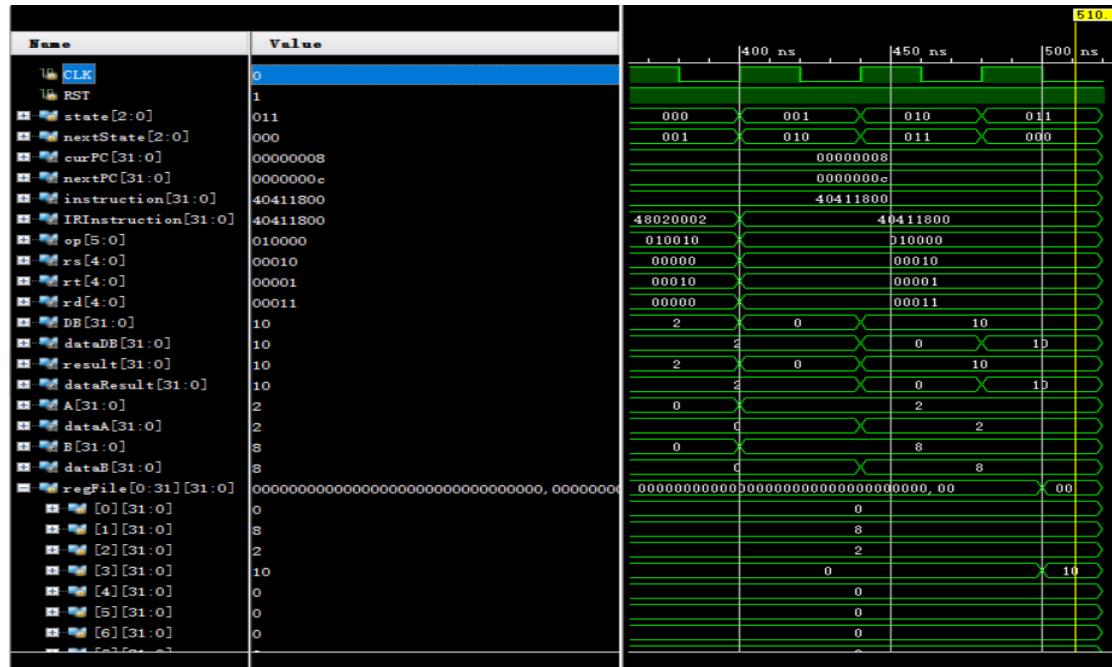


指令为或操作指令，将 0 号寄存器的数据 (0) 与立即数 2 进行或操作，并且将运算结果 (2) 写回 2 号寄存器。需要取指令，指令译码，指令执行，结果写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



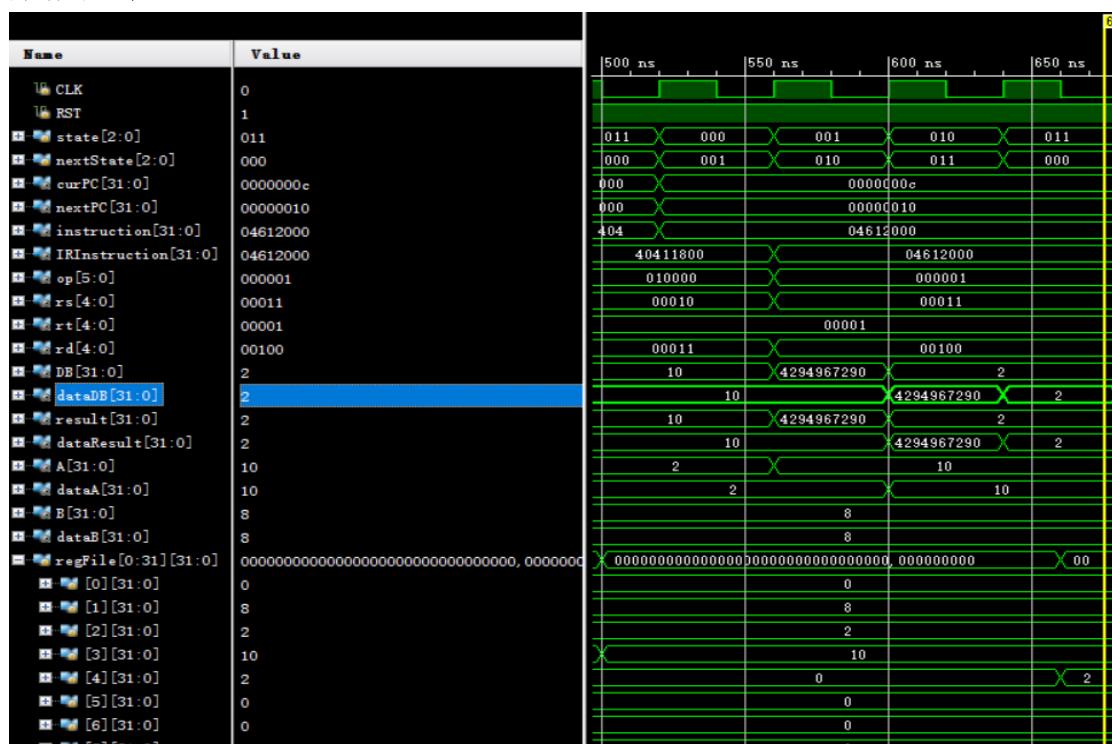
0x00000008	or \$3, \$2, \$1	010000	00010	00001	0001 1000 0000 0000	=	40411800
------------	------------------	--------	-------	-------	---------------------	---	----------

指令为或操作指令，将 2 号寄存器的数据 (2) 与 1 号寄存器的数据 (8) 进行或操作，并且将运算结果 (10) 写回 3 号寄存器。需要取指令，指令译码，指令执行，结果写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



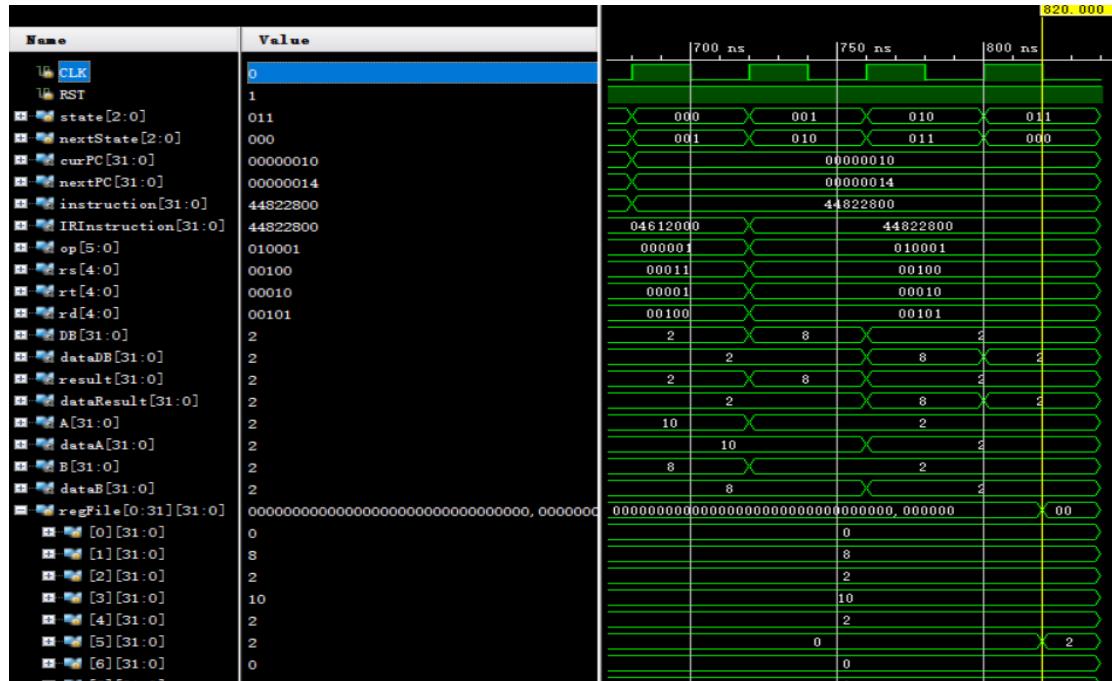
0x0000000C	sub \$4, \$3, \$1	000001	00011	00001	0010 0000 0000 0000	=	04612000
------------	-------------------	--------	-------	-------	---------------------	---	----------

指令为减操作指令，将 3 号寄存器的数据 (10) 与 1 号寄存器的数据 (8) 进行减操作，并且将运算结果 (2) 写回 4 号寄存器。需要取指令，指令译码，指令执行，结果写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



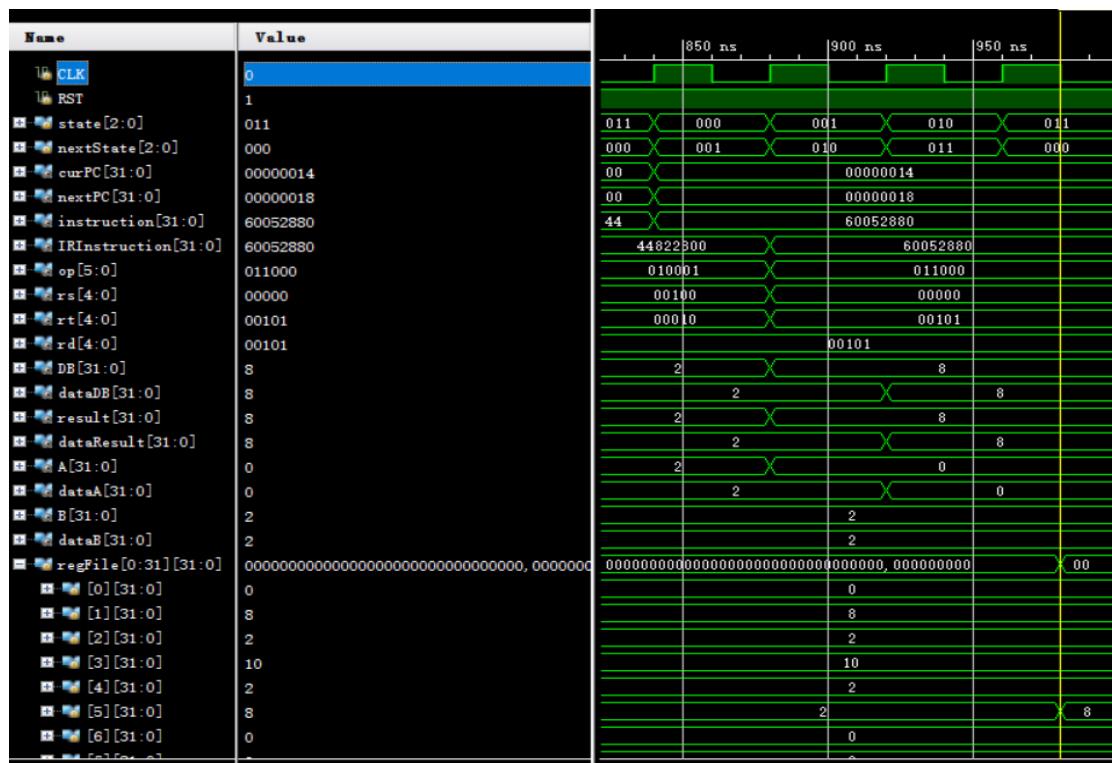
0x00000010	and \$5, \$4, \$2	010001	00100	00010	0010 1000 0000 0000	=	44822800
------------	-------------------	--------	-------	-------	---------------------	---	----------

指令为与操作指令，将 4 号寄存器的数据 (2) 与 2 号寄存器的数据 (2) 进行与操作，并且将运算结果 (2) 写回 5 号寄存器。需要取指令，指令译码，指令执行，结果写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



0x00000014	sll \$5, \$5, 2	011000	00000	00101	0010 1000 1000 0000	=	60052880
------------	-----------------	--------	-------	-------	---------------------	---	----------

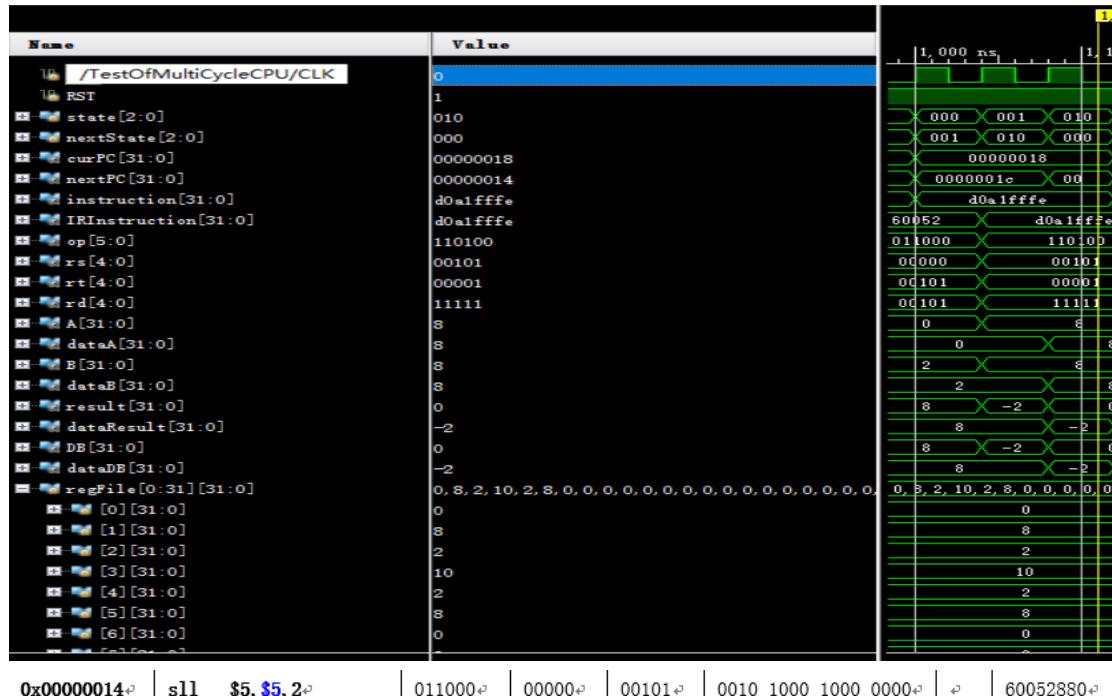
指令为移位操作指令，将 5 号寄存器的数据 (2) 左移 2 位，并且将运算结果 (8) 写回 5 号寄存器。需要取指令，指令译码，指令执行，结果写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



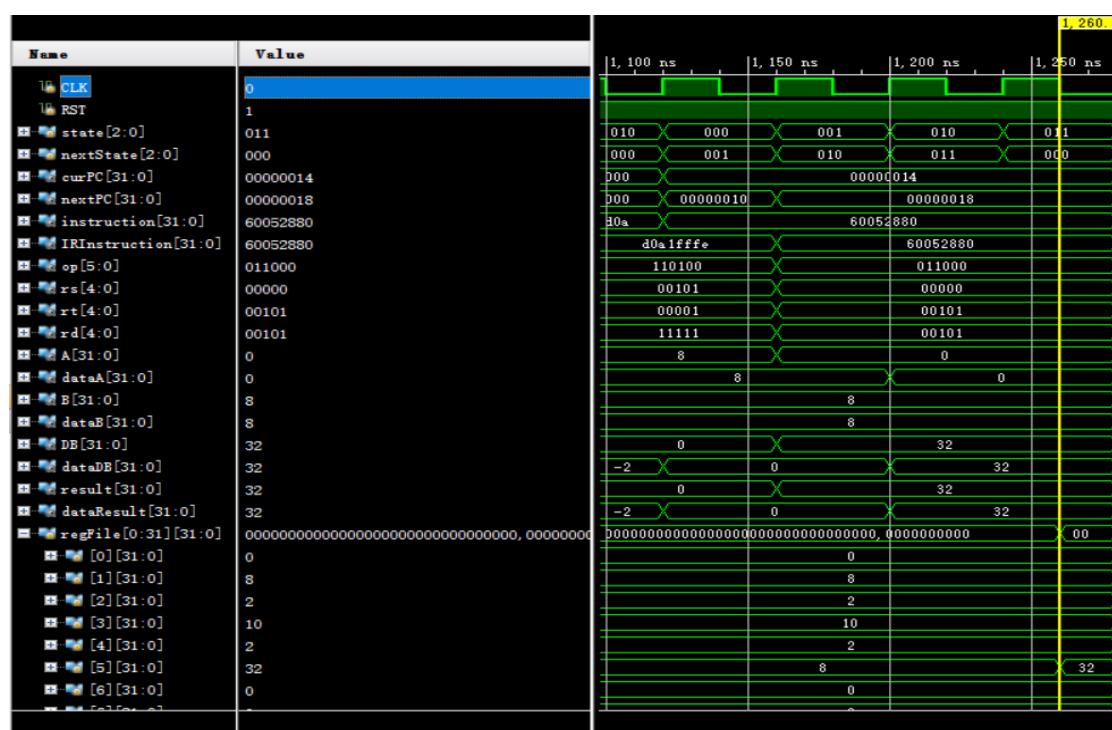
- 0x00000018 ← beq \$5, \$1, -2 (=, 转 14) ← 110100 ← 00101 ← 00001 ← 1111 1111 1111 1110 ← ← DOA1FFFF ←

指令为分支指令，将 5 号寄存器的数据 (8) 与 1 号寄存器的数据 (8) 进行比较，如果二者相等则跳转，跳转然后执行上一条指令(地址: 0x00000014)，否则继续顺序执行。本条指令将发生跳转，因为\$5 与\$1 所存储的数据相等。不需要存储器访问和数据写回，因此三个周期即可完成。

执行 beq \$5, \$1, -2:

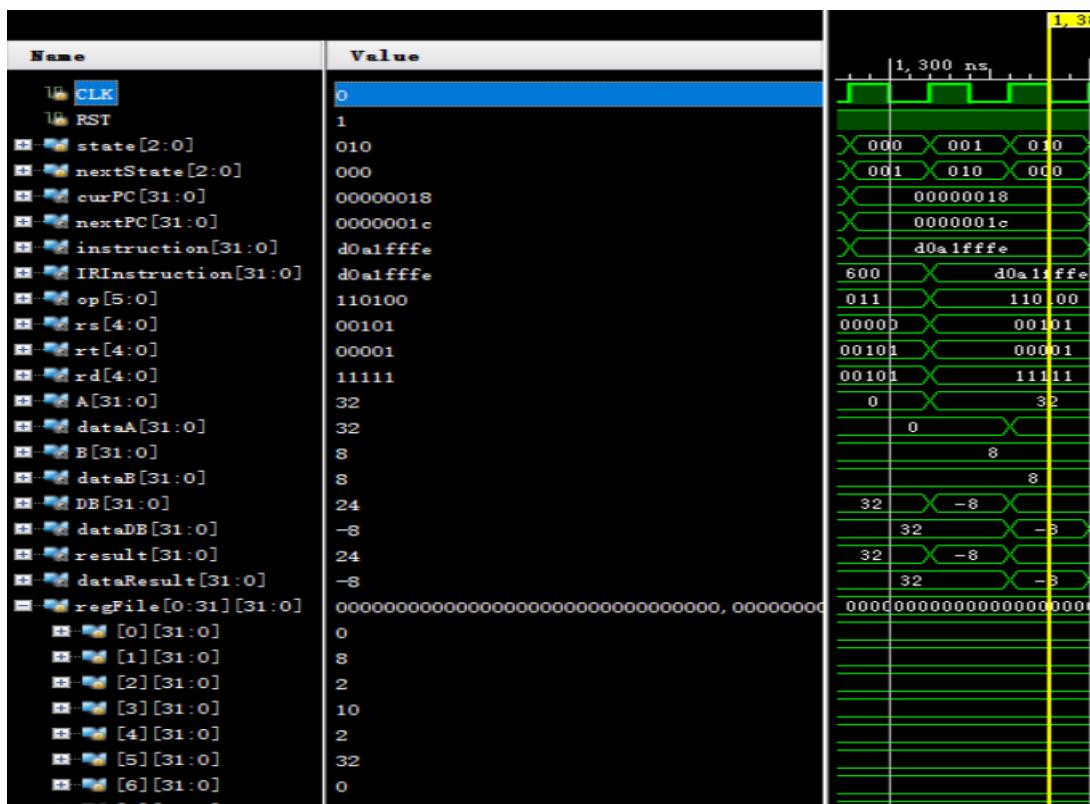


- 发生跳转，执行指令寄存器中地址为 0x00000014 所存储的指令 (sll \$5,\$5,2)，指令为移位操作指令，将 5 号寄存器的数据 (8) 左移 2 位，并且将运算结果 (32) 写回 5 号寄存器。需要取指令，指令译码，指令执行，结果写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



0x00000018 | beq \$5,\$1,-2(=, 转 14) | 110100 | 00101 | 00001 | 1111 1111 1111 1110 | | DOA1FFFF |

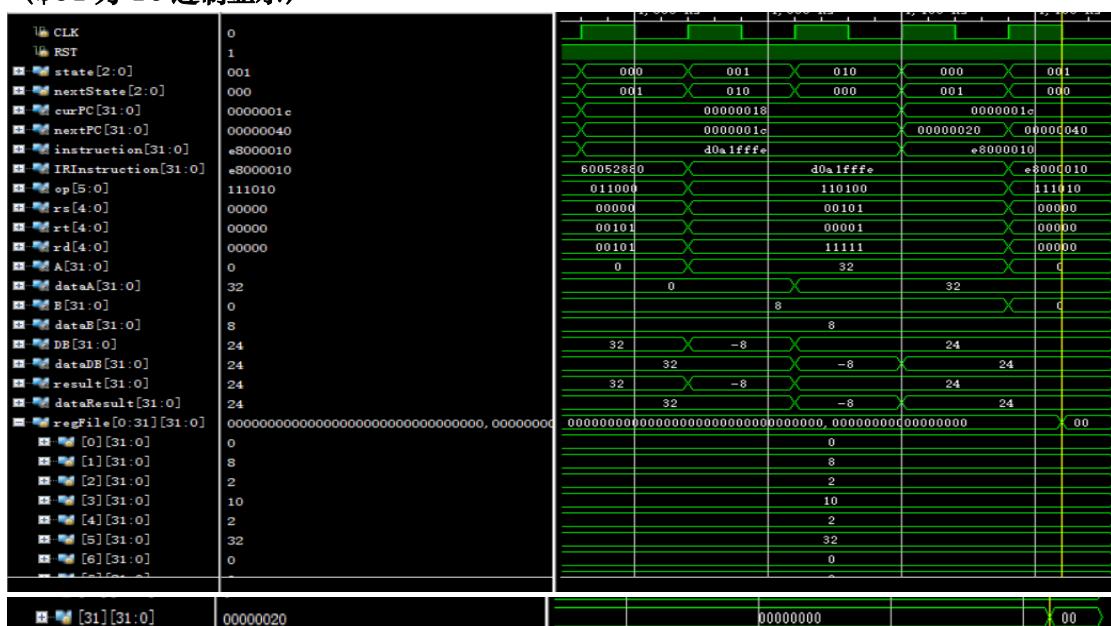
顺序执行，再次执行 beq \$5,\$1,-2，此时\$5 存储的数据为 32，\$1 存储的数据为 8，二者不相等，不发生跳转。需要取指令，指令译码，指令执行三个步骤，不需要存储器访问和数据写回，因此三个周期即可完成。



● | 0x0000001C | jal 0x00000040 | 11010 | 00000 | 00000 | 0000 0000 0001 0000 | | E8000010 |

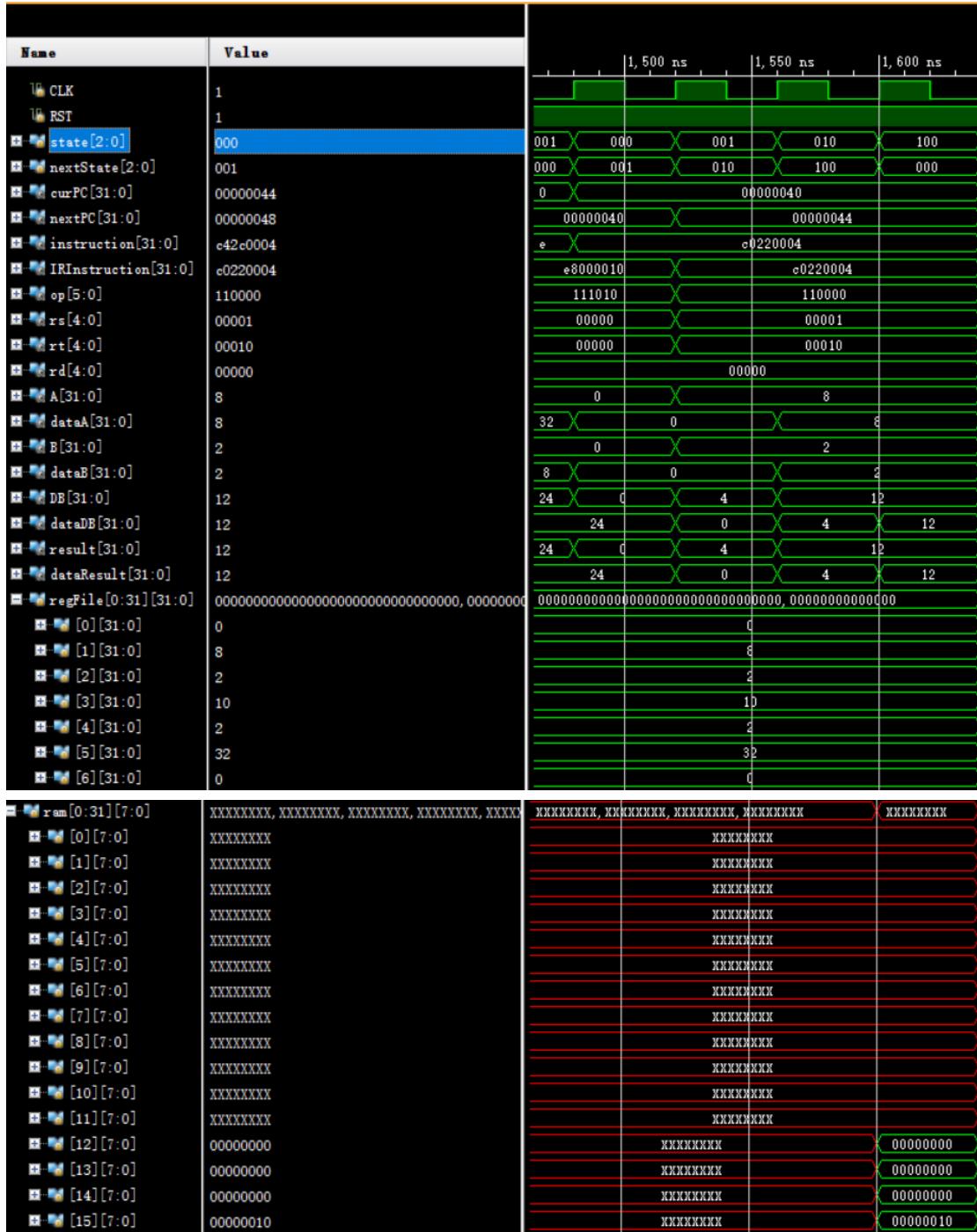
本指令为调用子程序指令，调用指令寄存器中地址 0x00000040 所存储的指令，同时将寄存器组中的 31 号寄存器存储当前 PC+4 的值 (0x00000020)。指令需要取指令、指令译码两个步骤，不需要指令执行、存储器访问和数据写回，因此两个周期即可完成。

(\$31 为 16 进制显示)



0x00000040 ↗ sw \$2,4(\$1) ↗ 110000 ↗ 00001 ↗ 00010 ↗ 0000 0000 0000 0100 ↗ ↗ C0220004 ↗

跳转后执行指令寄存器中地址为 0x00000040 所存储的指令。该指令执行将\$2 寄存器的内容 (2) 保存到\$1 寄存器内容 (8) 和立即数符号扩展后的数 (4) 相加作为地址 (12) 的内存单元中，写入模式为大端。指令需要取指令、指令译码、指令执行、存储器访问五个步骤，不需要数据写回，因此四个周期即可完成。

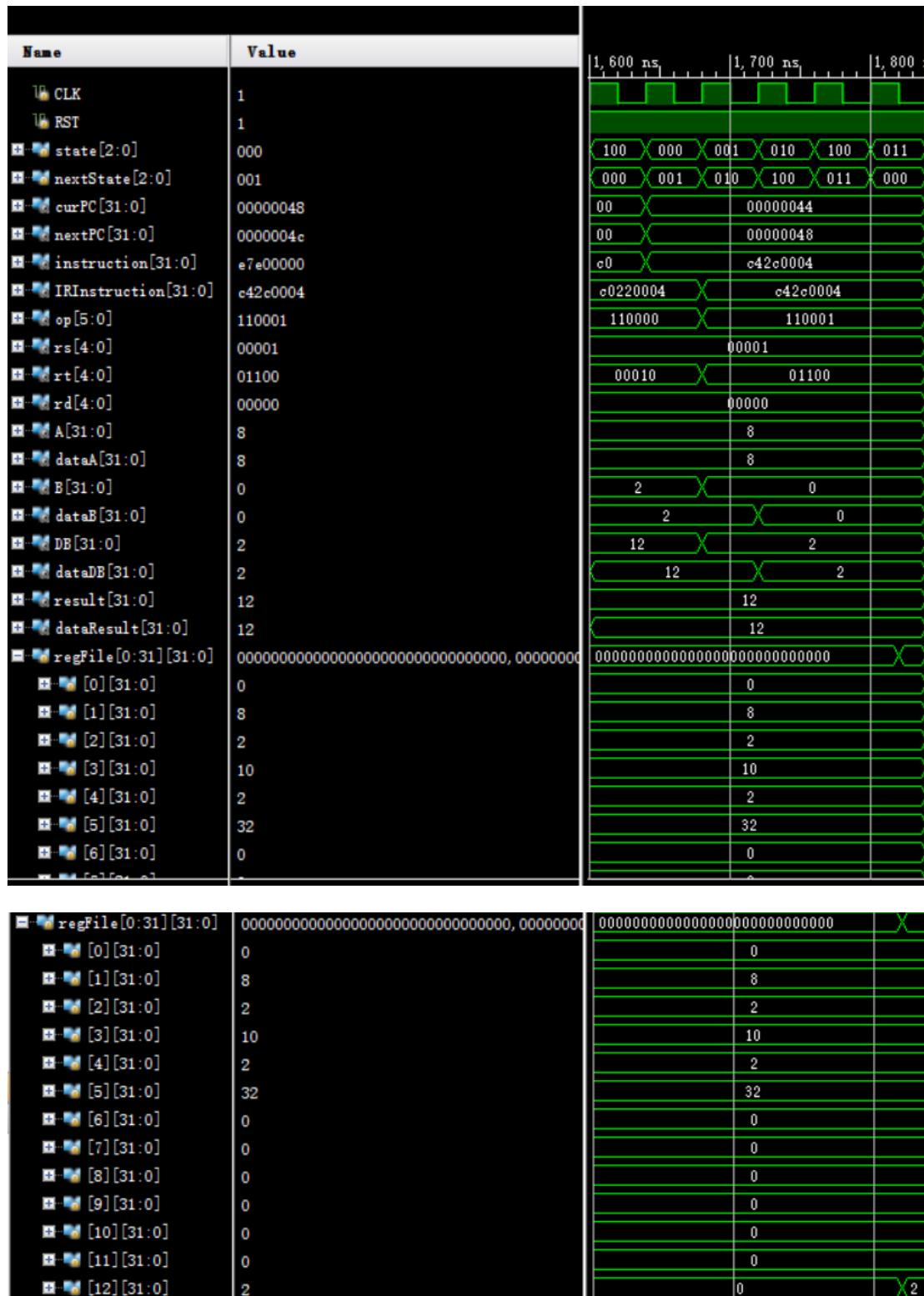


顺序执行指令:

0x000000044 ↗ lwr \$12,4(\$1) ↗ 110001 ↗ 00001 ↗ 01100 ↗ 0000 0000 0000 0100 ↗ ↗ C42C0004 ↗

该指令执行将\$1 寄存器的内容 (8) 和立即数扩展后的数 (4) 相加作为地址的内存单元中的数 (12)，然后保存到\$12 中。指令需要取指令、指令译码、指令执行、存储器

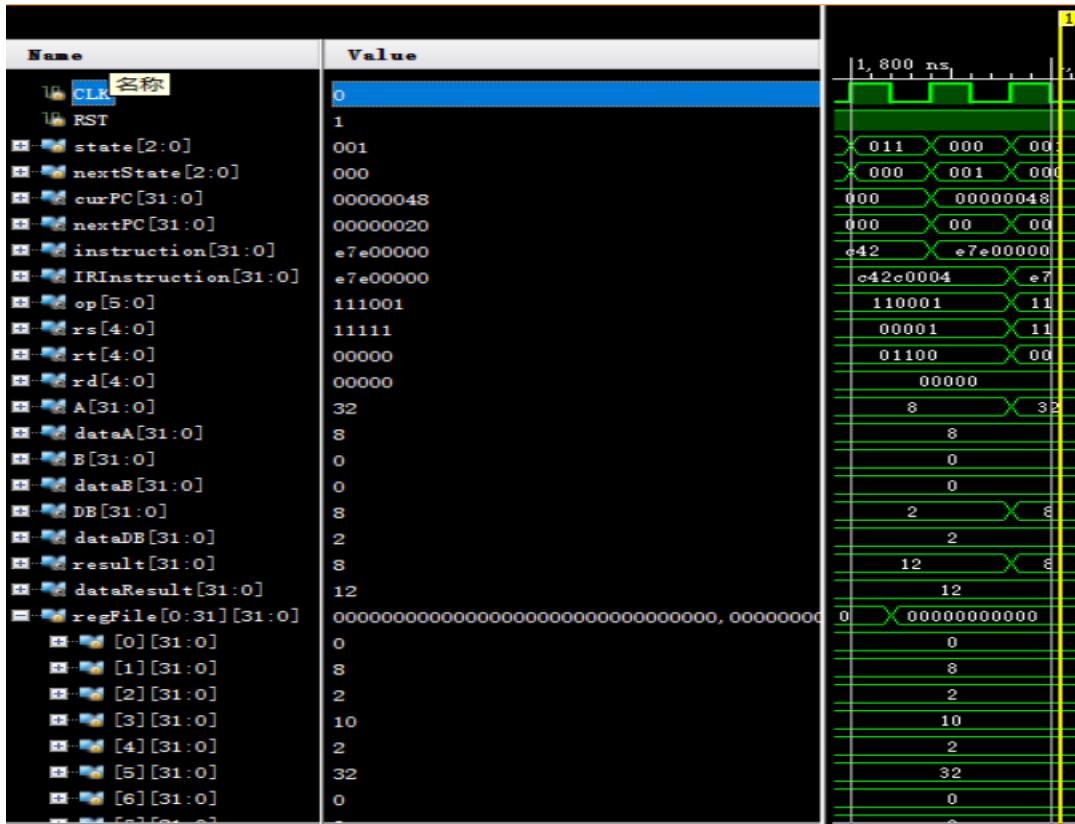
访问、结果写回五个步骤，因此五个周期即可完成。



指令顺序执行：

0x00000048^{op} | jr^{op} \$31^{op} | 111001^{op} | 11111^{op} | 00000^{op} | 0000 0000 0000 0000^{op} | op | E7E00000^{op}

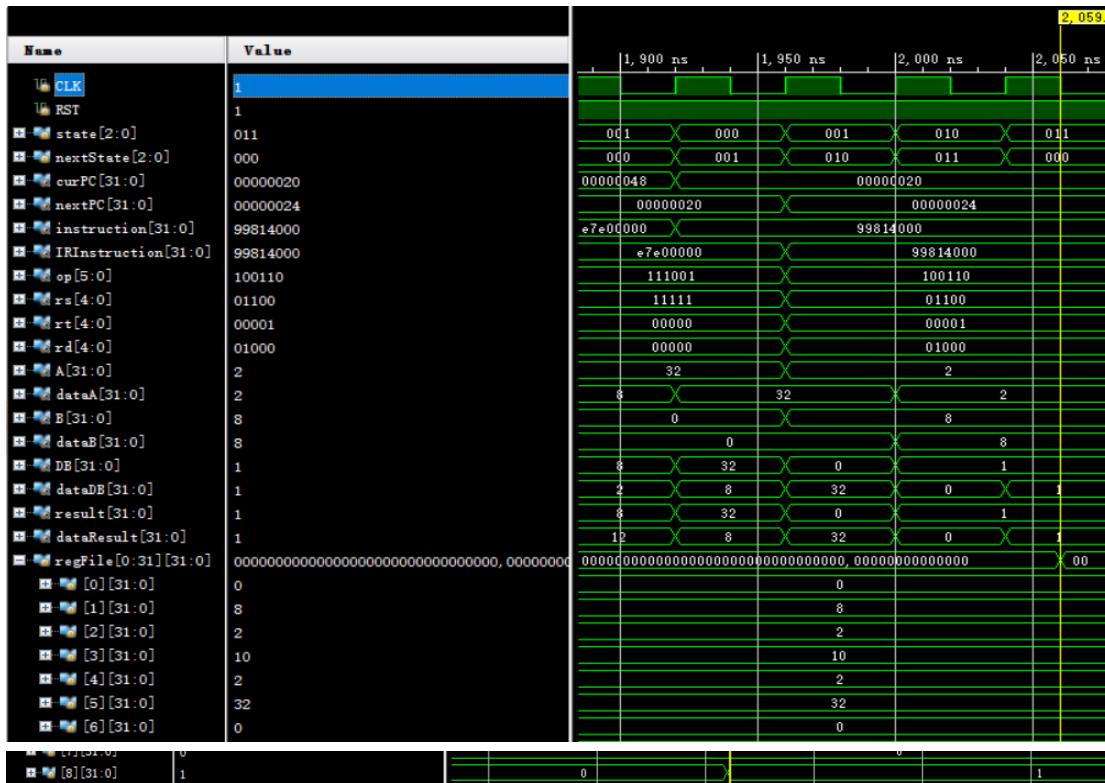
本指令为跳转指令，跳转到\$31 寄存器所存储的地址 (0x00000020)。指令需要取指令和指令译码两个步骤，不需要指令执行、存储器访问和数据写回，因此两个周期即可完成。



跳转到\$31 (0x00000020), jal 的子程序调用结束。

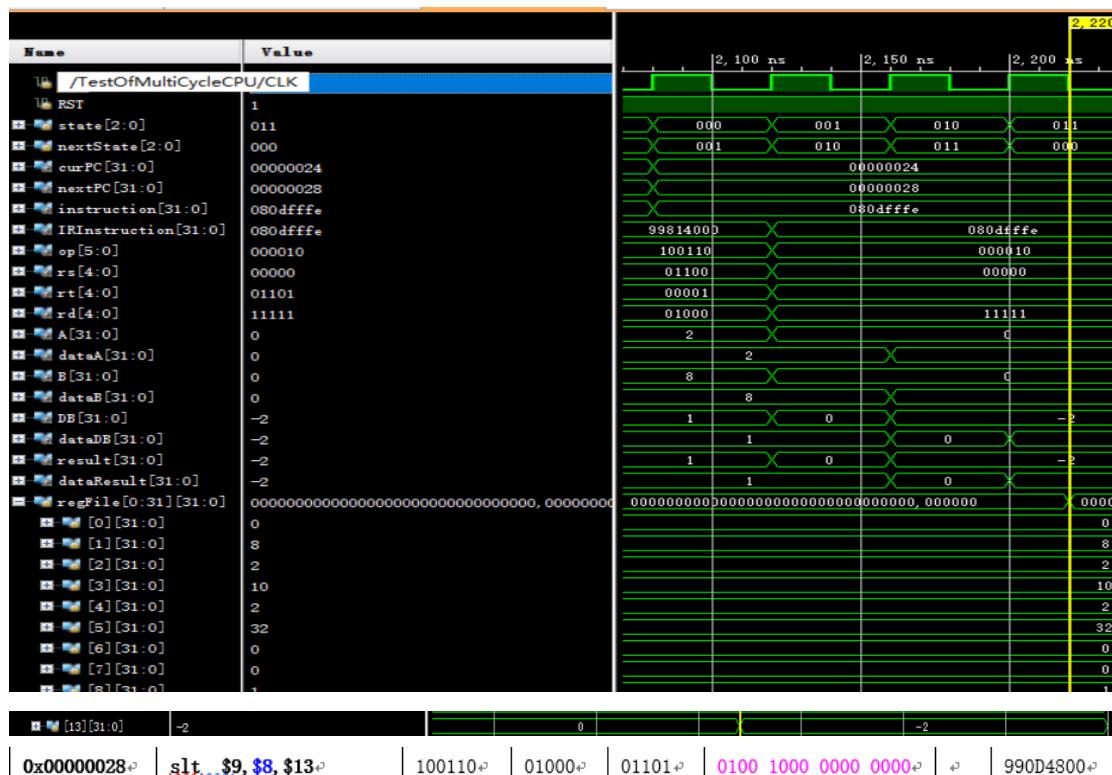
● $0x00000020 \quad | \quad \text{slt} \$8, \$12, \$1 \quad | \quad 100110 \quad | \quad 01100 \quad | \quad 0100 \ 0000 \ 0000 \ 0000 \quad | \quad 99814000$

指令为比较指令, 将\$12 寄存器的数据 (2) 与\$1 寄存器的数据 (8) 比较, 若\$12 的数据小于\$1 寄存器的数据, 则将 1 写入\$8, 否则写入 0, 本指令将 1 写入\$8。不需要存储器访问, 因此四个周期即可完成, 在最后一个时钟周期的下降沿写入数据。



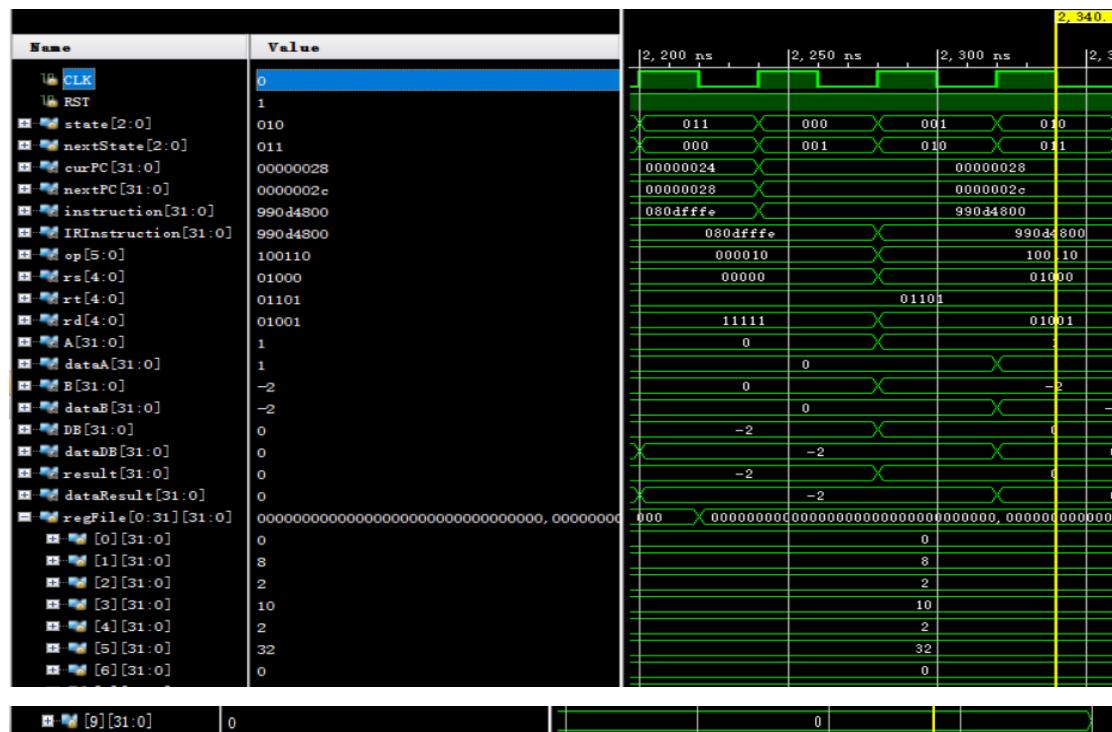
- 0x00000024** addi \$13, \$0, -2 000010 00000 01101 11111111111111110 0 080DFFFFE

指令为加操作指令，将\$0 寄存器的数据 (0) 与立即数 (-2) 相加，将结果 (-2) 写入\$13。指令需要取指令、指令译码、指令执行和数据写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



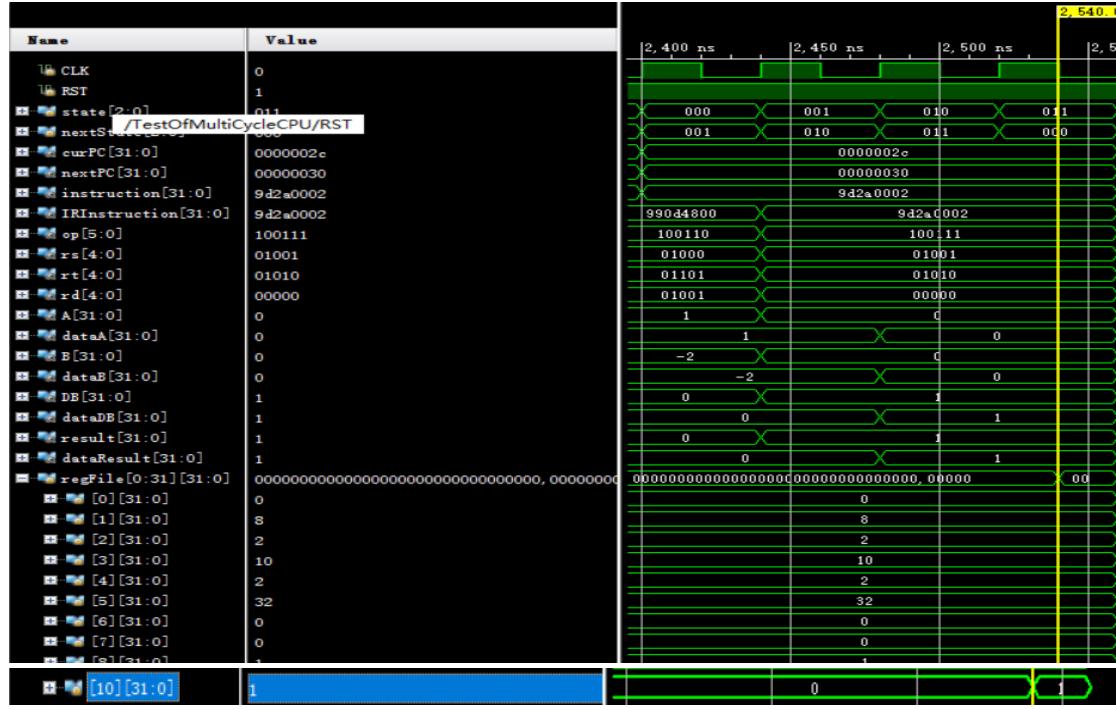
- 0x00000028 ↴ slt \$9, \$8, \$13 ↵ 100110 ↴ 01000 ↴ 01101 ↴ 0100 1000 0000 0000 ↵ ↵ 990D4800 ↵

指令为比较指令，将\$8 寄存器的数据（1）与\$13 寄存器的数据（-2）比较，若\$8 的数据小于\$13 寄存器的数据，则将 1 写入\$9，否则写入 0，本指令将 0 写入\$9。不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



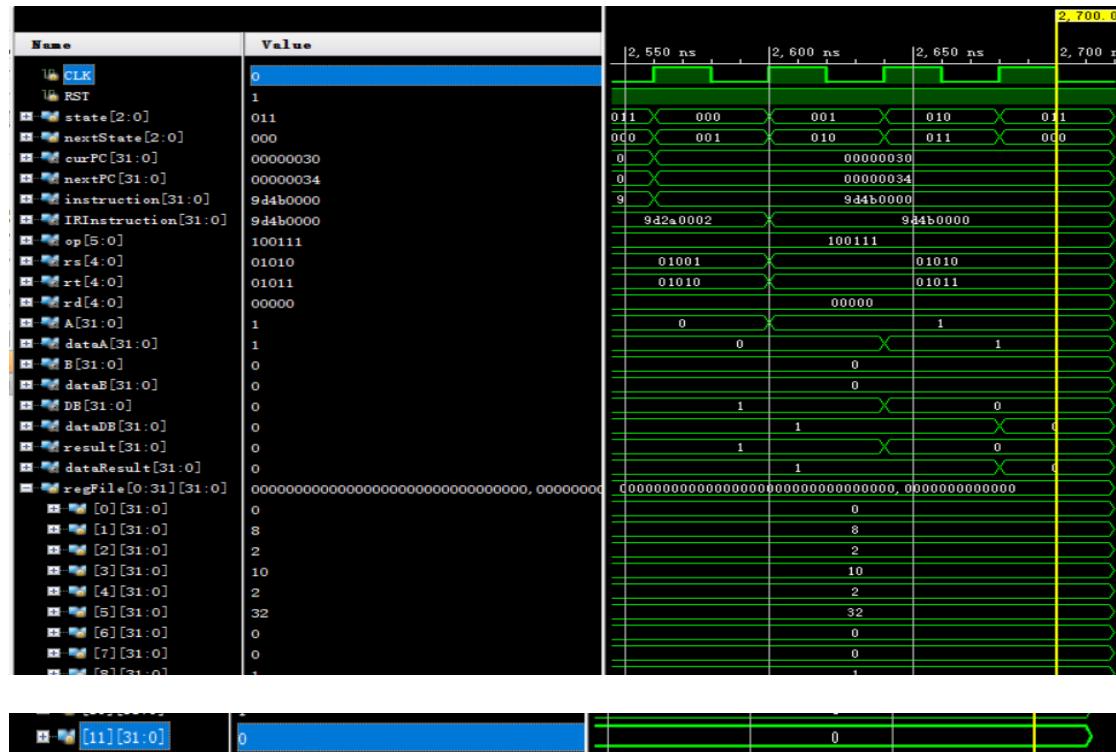
0x0000002C	sltiu \$10,\$9,2	100111	01001	01010	0000 0000 0000 0010		9D2A0002
------------	------------------	--------	-------	-------	---------------------	--	----------

指令为比较指令，将\$9 寄存器的数据（0）与立即数（2）比较，若\$9 的数据小于 2，则将 1 写入\$10，否则写入 0，本指令将 1 写入\$10。不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



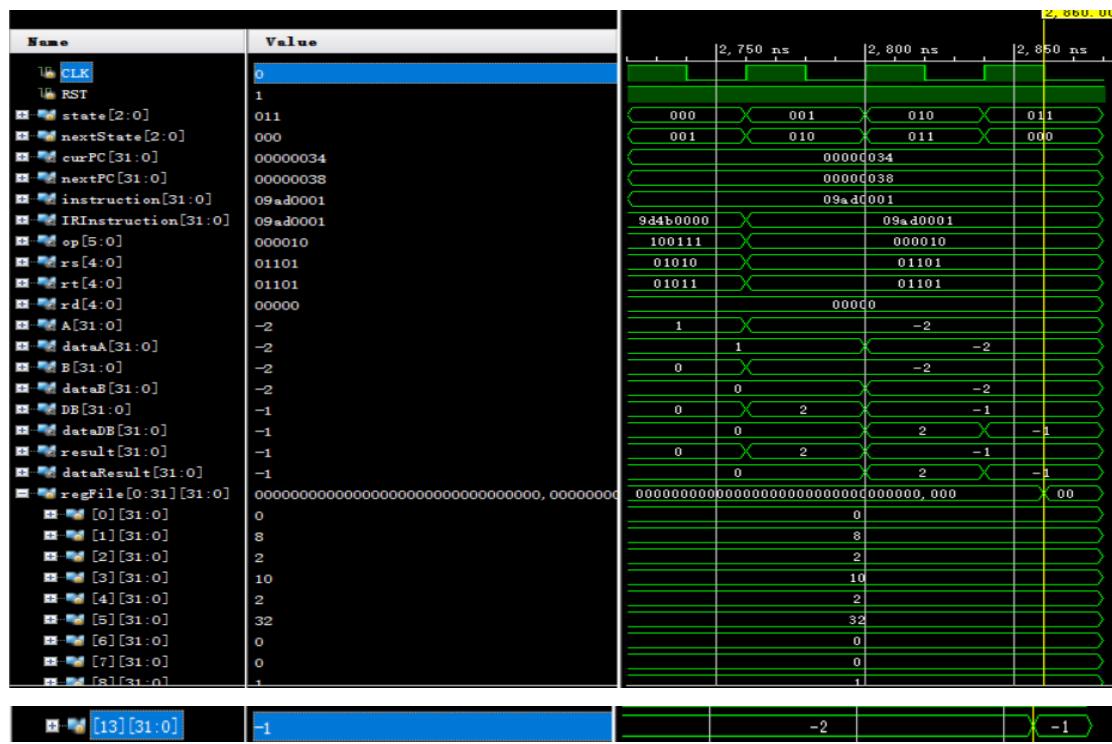
0x00000030	sltiu \$11,\$10,0	100111	01010	01011	0000 0000 0000 0000		9D4B0000
------------	-------------------	--------	-------	-------	---------------------	--	----------

指令为比较指令，将\$10 寄存器的数据（1）与立即数（0）比较，若\$10 的数据小于 0，则将 1 写入\$10，否则写入 0，本指令将 0 写入\$11。不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



0x00000034 ^v	addi \$13,\$13,1 ^v	000010 ^v	01101 ^v	01101 ^v	0000 0000 0000 0001 ^v	^v	09AD0001 ^v
-------------------------	-------------------------------	---------------------	--------------------	--------------------	----------------------------------	--------------	-----------------------

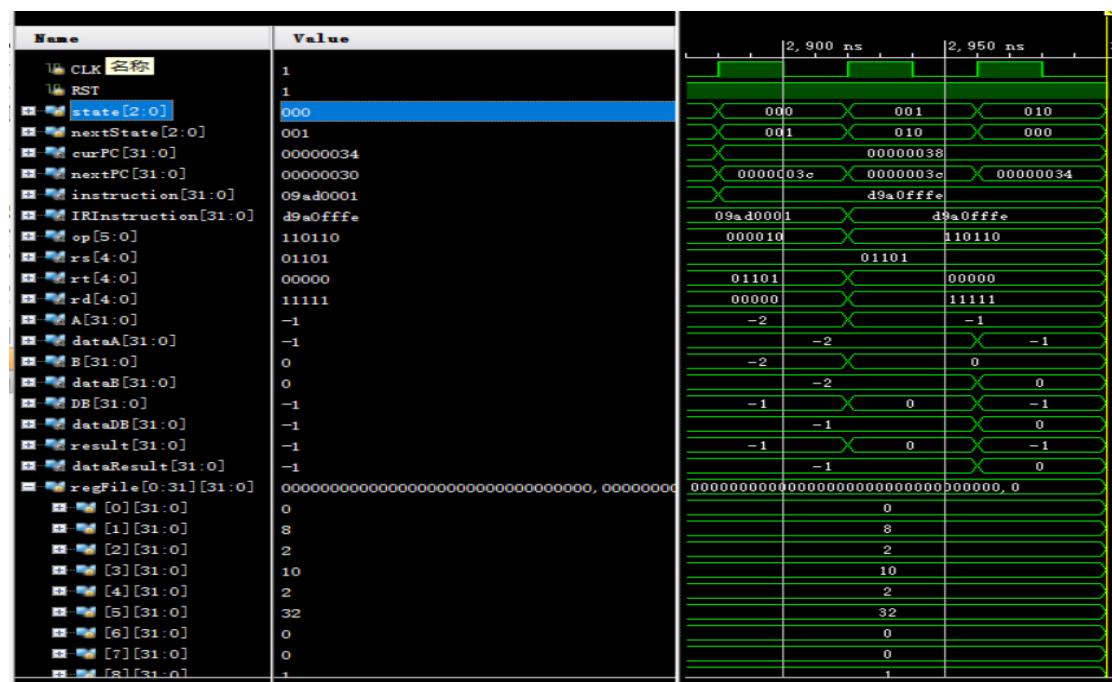
指令为加操作指令，将\$13 寄存器的数据 (-2) 与立即数 (1) 相加，将结果 (-1) 写入\$13。指令需要取指令、指令译码、指令执行和数据写回四个步骤，不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。



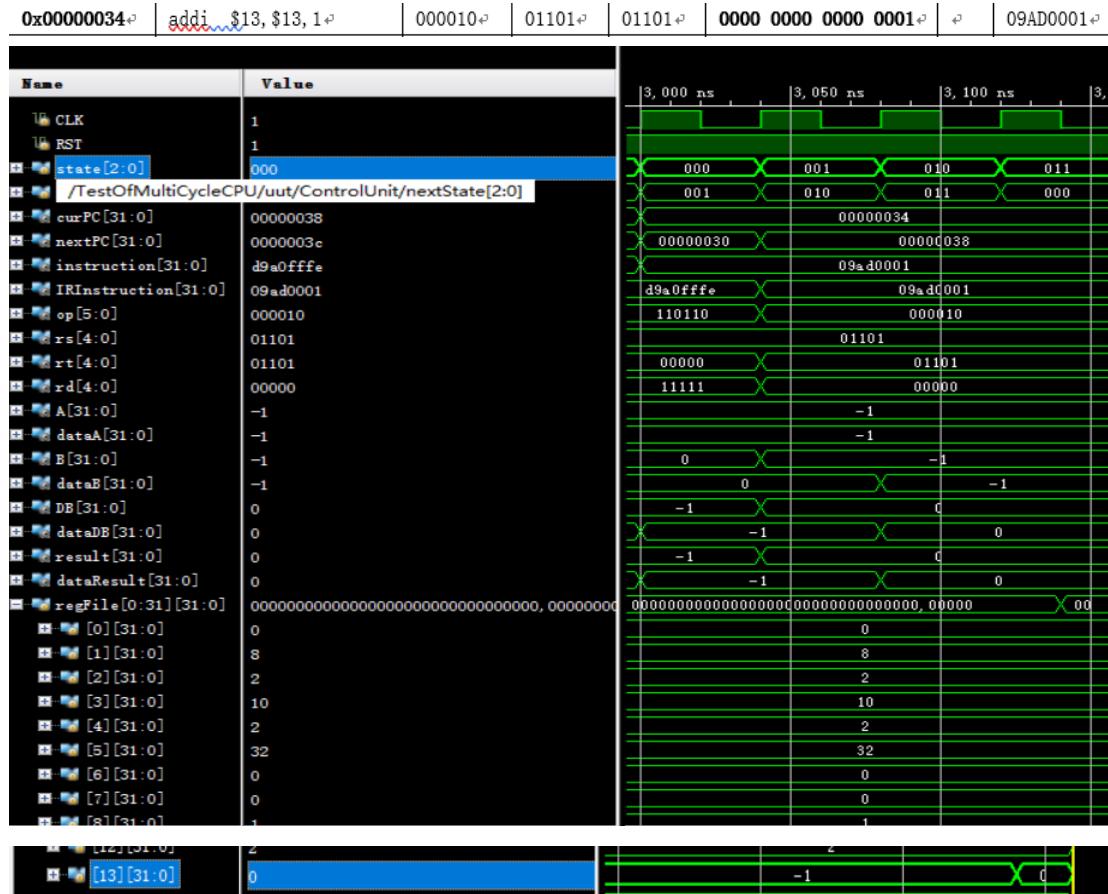
0x00000038 ^v	bltz \$13,-2 (<0, 转 34) ^v	110110 ^v	01101 ^v	00000 ^v	1111 1111 1111 1110 ^v	^v	D9A0FFFE ^v
-------------------------	--------------------------------------	---------------------	--------------------	--------------------	----------------------------------	--------------	-----------------------

指令为分支指令，将 \$13 号寄存器的数据 (-1) 与 0 进行比较，如果小于则跳转，跳转然后执行上一条指令(地址: 0x00000034)，否则继续顺序执行。本条指令将发生跳转，因为\$13 所存储的数据 (-1) 小于 0。指令需要取指令、指令译码和指令执行三个步骤，不需要存储器访问和数据写回，因此三个周期即可完成。

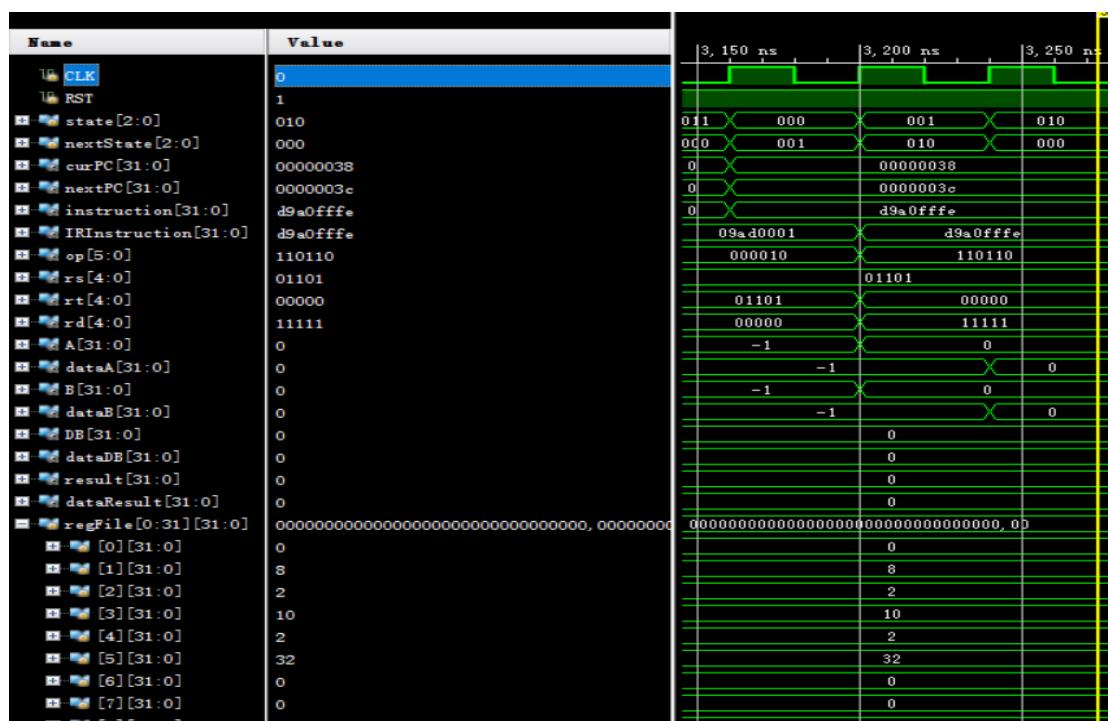
执行 bltz \$13, -2:



发生跳转，执行指令寄存器中地址为 0x00000034 所存储的指令 (addi \$13,\$13,1) , 指令为加操作指令，将\$13 的数据 (-1) 加 1，并且将运算结果 (0) 写回寄存器\$13。不需要存储器访问，因此四个周期即可完成，在最后一个时钟周期的下降沿写入数据。

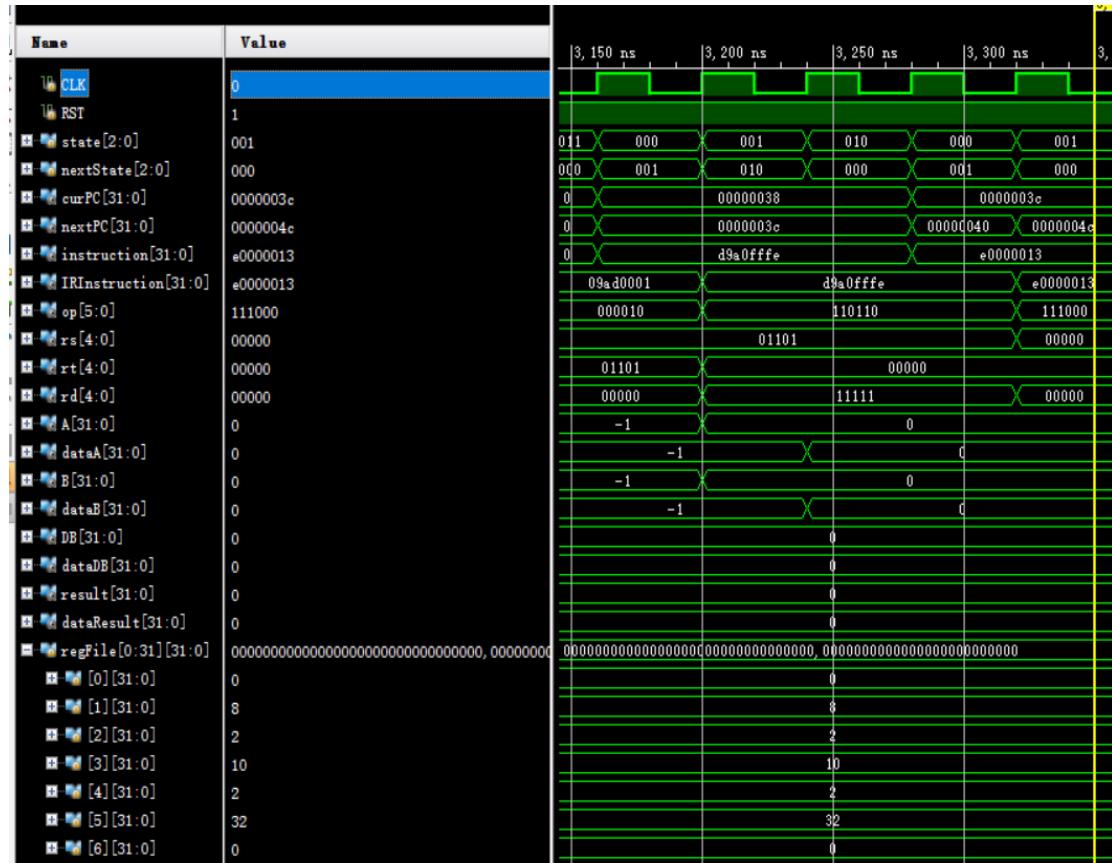


顺序执行，再次执行 bltz \$13,-2，此时\$13 存储的数据为 0，小于 0 不满足，不发生跳转。指令不需要存储器访问和数据写回，因此三个周期即可完成。



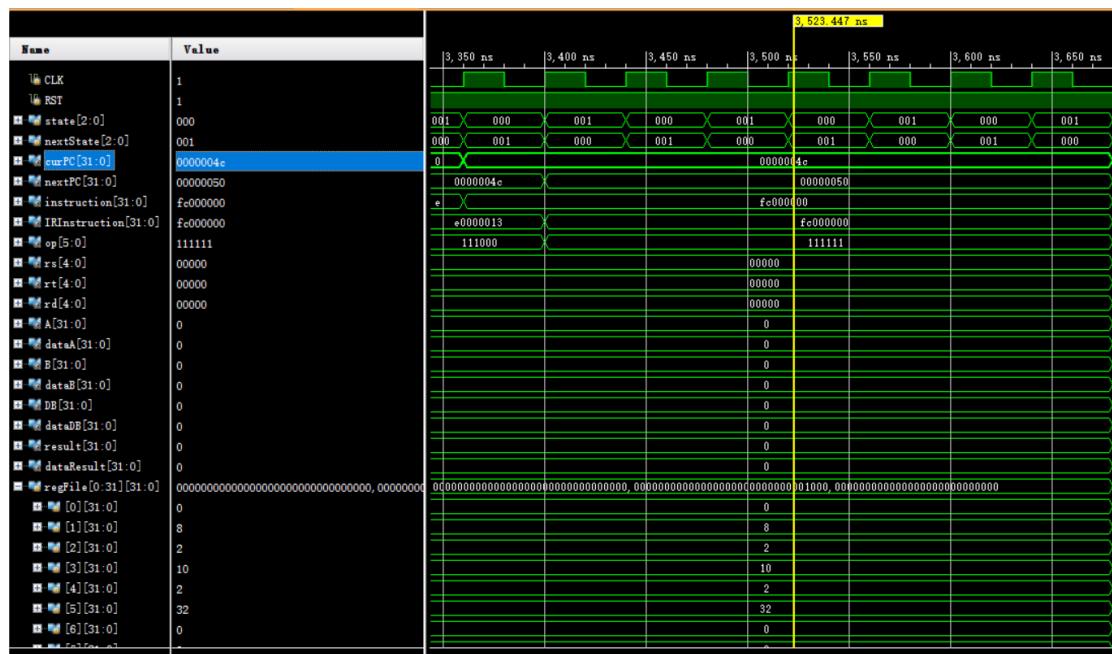
0x00000003C	j, 0x0000004C	111000	00000	00000	0000 0000 0001 0011	=	E0000013
-------------	---------------	--------	-------	-------	---------------------	---	----------

本指令为跳转指令，跳转到地址 0x0000004C。指令需要取指令和指令译码，不需要指令执行、存储器访问和数据写回，因此两个周期即可完成。



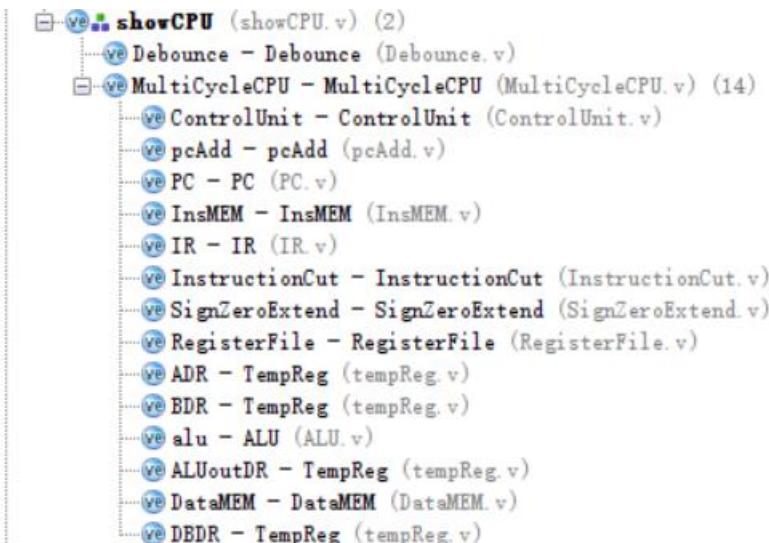
0x00000004C	halt	111111	00000	00000	00000000000000000000000000000000	=	FC000000
-------------	------	--------	-------	-------	----------------------------------	---	----------

本指令为停机指令。(PC 不改变)



④ Basys3 板上运行设计的多周期 CPU

- a) 设计：额外创建一个新的 project，主要还是将所设计的多周期 CPU 实现的代码复制进去，添加了一个按键消抖的模块，同时添加一个新的顶层模块 showCPU，将消抖模块和设计的多周期 CPU 设计作为子模块。
- b) 文件结构



c) 消抖模块

- 消抖原因和目的：脉冲按键与电平按键通常采用机械式开关结构，按键信号在开关拨片与出点接触多次弹跳以后才会稳定，在按键过程中，可能出现多个脉冲，为了实现每次按键只执行一条指令，则必须对其进行消抖，防止一次产生多个脉冲，导致执行多条指令。
- 主要代码：

```

module Debounce(
    input clk,
    input key,
    output out
);

reg delay1, delay2, delay3;
assign out = delay1&delay2&delay3;
always@(posedge clk)//CLK 100M
begin
    delay1 <= key;
    delay2 <= delay1;
    delay3 <= delay2;
end
endmodule
  
```

d) 顶层模块

- 实例化 CPU 模块和消抖模块

```

Debounce Debounce(.clk(clock_100Mhz), .key(click), .out(CpuCLK));

MultiCycleCPU MultiCycleCPU(.CLK(CpuCLK),
                           .RST(Reset),
                           .curPC(curPC),
                           .nextPC(nextPC),
                           .instruction(instruction),
                           .IRInstruction(IRInstruction),
                           .op(op),
                           .rs(rs),
                           .rt(rt),
                           .rd(rd),
                           .DB(DB),
                           .dataDB(dataDB),
                           .A(A),
                           .dataA(dataA),
                           .B(B),
                           .dataB(dataB),
                           .result(result),
                           .dataResult(dataResult),
                           .PCSrc(PCSrc));

```

ii. 七段数码管显示

```

1. //扫描频率
2. always @(posedge clock_100Mhz)
3. begin
4.     refresh_counter <= refresh_counter + 1;
5. end
6. assign LED_activating_counter = refresh_counter[20:19];
7.
8. //显示板块
9. always @(*)
10. begin
11.     case(LED_activating_counter)
12.         2'b00: begin
13.             Anode_Activate = 4'b0111;
14.             if(!S0 && !S1) begin
15.                 LED_BCD = curPC[7:4];
16.             end
17.             else if(!S0 && S1) begin
18.                 LED_BCD ={3'b000, rs[4]};
19.             end
20.             else if(S0 && S1) begin
21.                 LED_BCD ={3'b000, rt[4]};
22.             end
23.             else begin

```

```
24.                     LED_BCD = dataResult[7:4];
25.                 end
26.             end
27.         2'b01: begin
28.             Anode_Activate = 4'b1011;
29.             if(!S0 && !S1) begin
30.                 LED_BCD = curPC[3:0];
31.             end
32.             else if(!S0 && S1) begin
33.                 LED_BCD = rs[3:0];
34.             end
35.             else if(S0 && !S1) begin
36.                 LED_BCD = rt[3:0];
37.             end
38.             else begin
39.                 LED_BCD = dataResult[3:0];
40.             end
41.         end
42.         2'b10: begin
43.             Anode_Activate = 4'b1101;
44.             if(!S0 && !S1) begin
45.                 LED_BCD = nextPC[7:4];
46.             end
47.             else if(!S0 && S1) begin
48.                 LED_BCD = dataA[7:4];
49.             end
50.             else if(S0 && !S1) begin
51.                 LED_BCD = dataB[7:4];
52.             end
53.             else begin
54.                 LED_BCD = dataDB[7:4];
55.             end
56.         end
57.         2'b11: begin
58.             Anode_Activate = 4'b1110;
59.             if(!S0 && !S1) begin
60.                 LED_BCD = nextPC[3:0];
61.             end
62.             else if(!S0 && S1) begin
63.                 LED_BCD = dataA[3:0];
64.             end
65.             else if(S0 && !S1) begin
66.                 LED_BCD = dataB[3:0];
67.             end
```

```

68.           else begin
69.               LED_BCD = dataDB[3:0];
70.           end
71.       end
72.   endcase
73. end
74. // Cathode patterns of the 7-segment LED display
75. always @(*)
76. begin
77.     case(LED_BCD)
78.         4'b0000: LED_out = 7'b0000001; // "0"
79.         4'b0001: LED_out = 7'b1001111; // "1"
80.         4'b0010: LED_out = 7'b0010010; // "2"
81.         4'b0011: LED_out = 7'b0000110; // "3"
82.         4'b0100: LED_out = 7'b1001100; // "4"
83.         4'b0101: LED_out = 7'b0100100; // "5"
84.         4'b0110: LED_out = 7'b0100000; // "6"
85.         4'b0111: LED_out = 7'b0001111; // "7"
86.         4'b1000: LED_out = 7'b0000000; // "8"
87.         4'b1001: LED_out = 7'b0000100; // "9"
88.         4'b1010: LED_out = 7'b0001000; //A
89.         4'b1011: LED_out = 7'b1100000; //B
90.         4'b1100: LED_out = 7'b0110001; //C
91.         4'b1101: LED_out = 7'b1000010; //D
92.         4'b1110: LED_out = 7'b0110000; //E
93.         4'b1111: LED_out = 7'b0111000; //F
94.     default: LED_out = 7'b0000000; //不亮
95. endcase
96. end

```

e) 约束文件

```

1. # Clock signal
2. set_property PACKAGE_PIN W5 [get_ports clock_100Mhz]
3.     set_property IOSTANDARD LVCMOS33 [get_ports clock_100Mhz]
4. set_property PACKAGE_PIN R2 [get_ports S0]
5.     set_property IOSTANDARD LVCMOS33 [get_ports S0]
6. set_property PACKAGE_PIN T1 [get_ports S1]
7.     set_property IOSTANDARD LVCMOS33 [get_ports S1]
8. set_property PACKAGE_PIN V17 [get_ports Reset]
9.     set_property IOSTANDARD LVCMOS33 [get_ports Reset]
10. set_property PACKAGE_PIN U18 [get_ports click]
11.     set_property IOSTANDARD LVCMOS33 [get_ports click]
12. #seven-segment LED display
13. set_property PACKAGE_PIN W7 [get_ports {LED_out[6]}]

```

```

14.    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[6]}]
15.    set_property PACKAGE_PIN W6 [get_ports {LED_out[5]}]
16.    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[5]}]
17.    set_property PACKAGE_PIN U8 [get_ports {LED_out[4]}]
18.    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[4]}]
19.    set_property PACKAGE_PIN V8 [get_ports {LED_out[3]}]
20.    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[3]}]
21.    set_property PACKAGE_PIN U5 [get_ports {LED_out[2]}]
22.    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[2]}]
23.    set_property PACKAGE_PIN V5 [get_ports {LED_out[1]}]
24.    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[1]}]
25.    set_property PACKAGE_PIN U7 [get_ports {LED_out[0]}]
26.    set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[0]}]
27.    set_property PACKAGE_PIN U2 [get_ports {Anode_Activate[0]}]
28.    set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[0]}]
29.    set_property PACKAGE_PIN U4 [get_ports {Anode_Activate[1]}]
30.    set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[1]}]
31.    set_property PACKAGE_PIN V4 [get_ports {Anode_Activate[2]}]
32.    set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[2]}]
33.    set_property PACKAGE_PIN W4 [get_ports {Anode_Activate[3]}]
34.    set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[3]}]

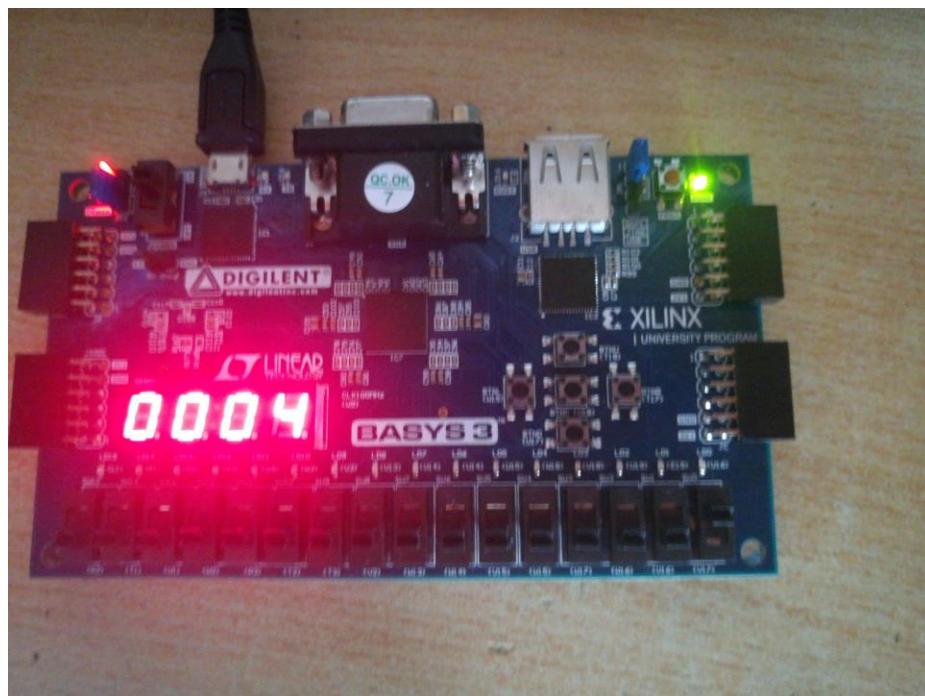
```

f) 结果显示（程序测试段与仿真测试的时候为同一段）

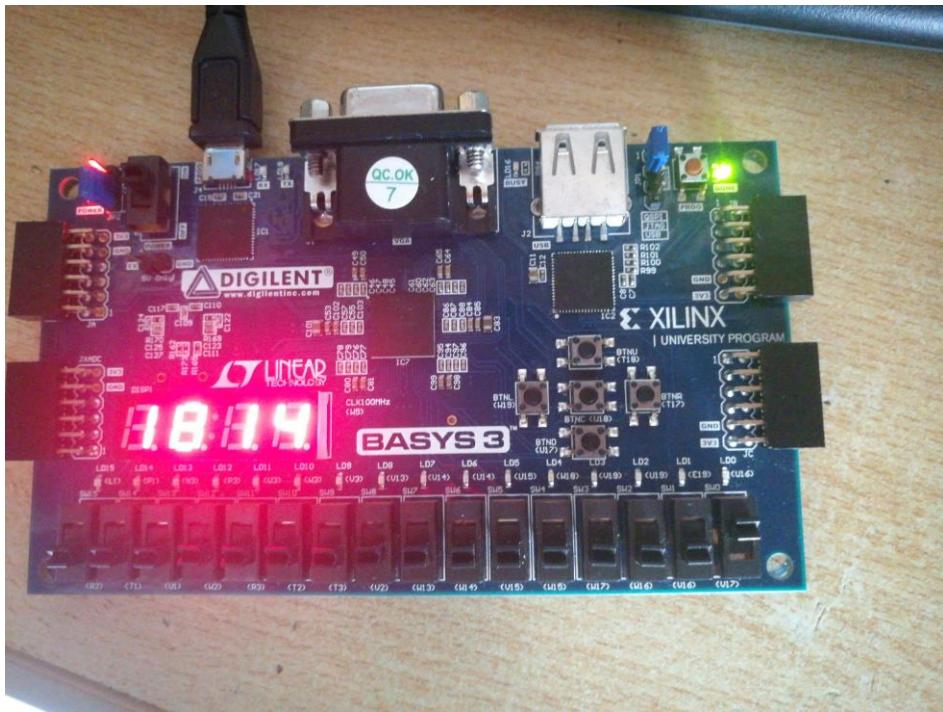
注意：每按一次 CPU 状态才切换一次，因此一条指令完成需要按与其需要的运行周期数相同的次数，只展示需要跳转部分时候当前 PC 和下一个 PC

显示：当前 PC + 下一 PC

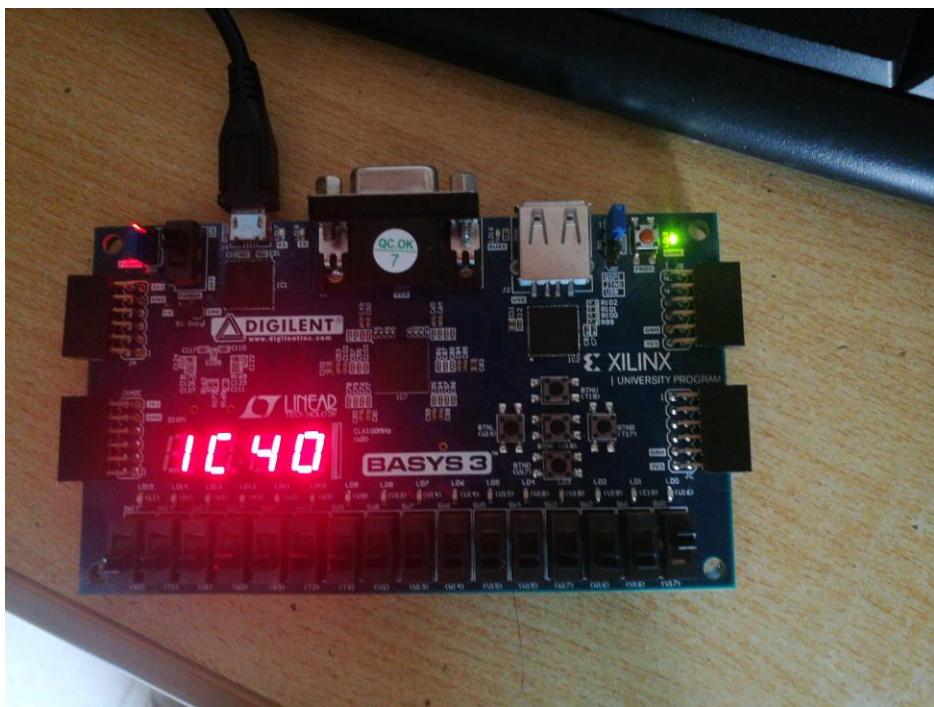
◆ 初始：



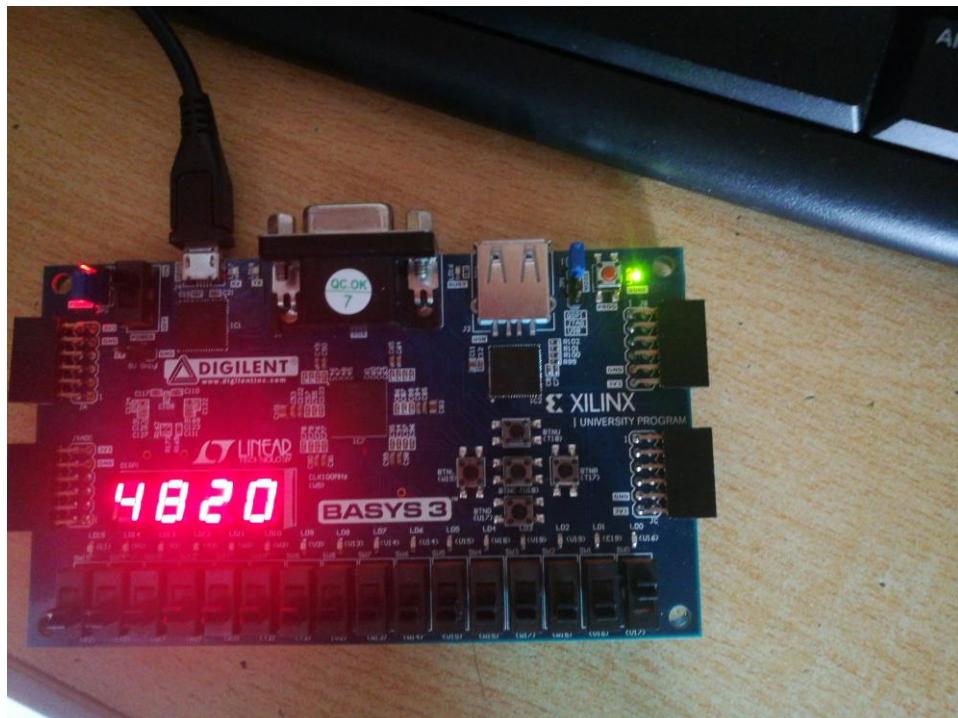
◆	0x00000018	beg \$5,\$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110	0	D0A1FFE	4
---	------------	------------------------	--------	-------	-------	---------------------	---	---------	---



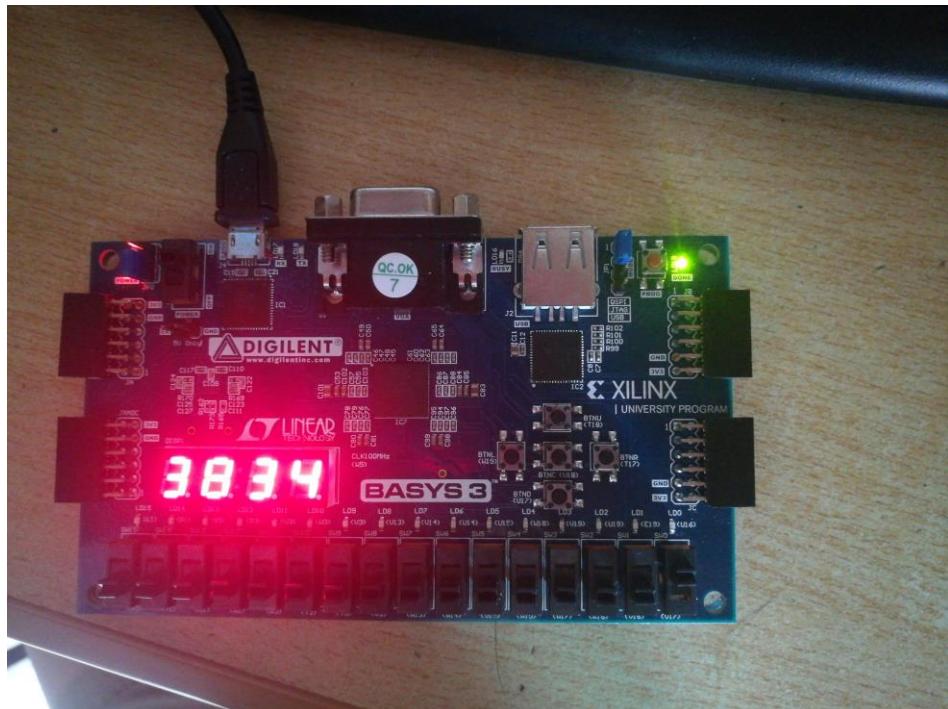
◆	0x0000001C	jal 0x0000040	111010	00000	00000	0000 0000 0001 0000	0	E8000010	4
---	------------	---------------	--------	-------	-------	---------------------	---	----------	---



◆	0x00000048	jr \$31	111001	11111	00000	0000 0000 0000 0000	·	E7E00000
(注: \$31 存的是 0x00000020)								



◆	0x00000038	bltz \$13, -2 (<0, 转 34)	110110	01101	00000	1111 1111 1111 1110	·	D9A0FFF





六. 实验心得

本次多周期CPU设计实验主要沿用之前设计的单周期CPU的设计模块，对部分模块进行了修改，以及新增一些寄存器模块。相比单周期CPU的实验，多周期CPU设计起来感觉比单周期CPU设计简单，主要是一开始设计单周期CPU一点都不懂，不知道如何开始，经过单周期CPU的设计后，对于多周期的设计思路也更加清晰，主要实现难点在于控制单元，一开始对于如何将一个指令如何分成多个周期执行，并且保证多周期执行也不会影响指令执行的正确性，这是我觉得多周期CPU设计的主要难点，主要是理清楚实验原理中执行一条时候在不同时候CPU的状态的情况，并据此建立一个CPU的状态机，通过指令的操作码、运算结果标记符（ZERO）控制信号确定当前CPU的状态，同时在时钟的上升沿对其状态进行切换，同时需要注意的便是某些控制信号需要等到特定的状态才可以变成相应的信号值，如果一开始就变成相应值的话就会出现不必要的错误，主要需要对注意对寄存器的写使能信号的控制。理清多周期CPU的实现思路以后，多周期CPU的实现就比较简单了，除了部分控制信号需要在特定CPU状态下为使能状态，还需要注意每个模块的敏感信号的设置，触发条件需要注意，其余的问题都不大。

至于Basys3板方面则按照原单周期CPU的思路实现，修改不大，依然需要注意的就是对按钮需要消抖处理。执行一条指令按按钮的次数与其需要周期数相同。烧板方面

本次问题也不大，主要坑在设计单周期CPU的时候都遇到了并且解决了，本次实验比较顺利。

本次多周期CPU设计实验，复习了下计组理论课多周期CPU的部分，真正去实现以后与之前相比还是收获很多的，加深了理解。关于机组实验这门课程，总的来说还是收获比较多的，学习了mips汇编编写、单周期CPU设计和多周期CPU设计，遇到的难题比较多，有的是verilog语言不熟原因，主要还是自己理论知识不过关，最后完成这些机组实验让我对于机组相关的理论知识理解的更加深入，更加透彻，本门课程收获很大。