



《计算机组成原理实验》

实验报告

(实验四)

学院名称 : 数据科学与计算机学院

专业(班级) : 18 计教学 3 班

学生姓名 : 王若琪

学号 : 18340166

时间 : 2019 年 12 月 10 日

成 绩 :

实验四：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图的构成、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) 认识和掌握指令与 CPU 的关系；
- (4) 掌握多周期 CPU 的测试方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 在单周期指令集的基础上增加实现以下指令功能操作。本次实验中需要实现运算操作的溢出判断：ALU 运算操作溢出时，ALU 需给出一位溢出信号（部分指令可能需要用到该信号。对于溢出发生时，需要能检测识别出，且不写回溢出错误结果，但不需要设计异常处理功能）。需设计的指令与格式如下，指令的具体描述和功能以 mips 官方文档为准：

==>逻辑运算指令

- (1) ADDI rt, rs, immediate

001000	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能：GPR[rt] \leftarrow GPR[rs] + sign_extend(immediate); immediate 做符号扩展再参加“加”运算。

==>比较指令

- (1) SLT rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 101010
--------	---------	---------	---------	--------------

功能：if (GPR[rs] < GPR[rt]) GPR[rd] = 1 else (GPR[rd] = 0)。

- (2) MOVN rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 001011
--------	---------	---------	---------	--------------

功能：if GPR[rt] \neq 0 then GPR[rd] \leftarrow GPR[rs]。

==>访存指令

- (1) LHU rt, offset(base)

100101	base(5 位)	rt(5 位)	offset(16 位)
--------	-----------	---------	--------------

功能：GPR[rt] \leftarrow memory[GPR[base] + offset]。

==>跳转指令

(1) JR rs

000000	rs(5位)	0000000000	未用	001000
--------	--------	------------	----	--------

功能: $PC \leftarrow GPR[rs]$, 跳转。**==>调用子程序指令**

(1) JAL addr

000011	addr[27:2]
--------	------------

功能: 调用子程序, $PC \leftarrow \{PC[31:28], addr, 2'b0\}$; $GPR[$31] \leftarrow pc+4$, 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。

以前的指令:

==> 算术运算指令

(1) add rd , rs, rt

000000	rs(5位)	rt(5位)	rd(5位)	00000 100000
--------	--------	--------	--------	--------------

功能: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$ 。

(2) sub rd , rs , rt

000000	rs(5位)	rt(5位)	rd(5位)	00000 100010
--------	--------	--------	--------	--------------

功能: $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$ 。

(3) addiu rt , rs ,immediate

001001	rs(5位)	rt(5位)	immediate(16位)
--------	--------	--------	----------------

功能: $GPR[rt] \leftarrow GPR[rs] + sign_extend(immediate)$; immediate 做符号扩展再参加“与”运算。**==> 逻辑运算指令**

(4) andi rt , rs ,immediate

001100	rs(5位)	rt(5位)	immediate(16位)
--------	--------	--------	----------------

功能: $GPR[rt] \leftarrow GPR[rs] and zero_extend(immediate)$; immediate 做 0 扩展再参加“与”运算。

(5) and rd , rs , rt

000000	rs(5位)	rt(5位)	rd(5位)	00000 100100
--------	--------	--------	--------	--------------

功能: $GPR[rd] \leftarrow GPR[rs] and GPR[rt]$ 。

(6) ori rt , rs ,immediate

001101	rs(5位)	rt(5位)	immediate(16位)
--------	--------	--------	----------------

功能: $GPR[rt] \leftarrow GPR[rs] or zero_extend(immediate)$ 。

(7) or rd , rs , rt

000000	rs(5位)	rt(5位)	rd(5位)	00000 100101
--------	--------	--------	--------	--------------

功能: $GPR[rd] \leftarrow GPR[rs] or GPR[rt]$ 。**==>移位指令**

(8) sll rd, rt,sa

000000	00000	rt(5位)	rd(5位)	sa(5位)	000000
--------	-------	--------	--------	--------	--------

功能: GPR[rd] \leftarrow GPR[rt] << sa。

==>比较指令

(9) slti rt, rs,immediate 带符号数

001010	rs(5位)	rt(5位)	immediate(16位)
--------	--------	--------	----------------

功能: if GPR[rs] < sign_extend(immediate) GPR[rt] = 1 else GPR[rt] = 0。

==> 存储器读/写指令

(10) sw rt, offset(rs) 写存储器

101011	rs(5位)	rt(5位)	offset(16位)
--------	--------	--------	-------------

功能: memory[GPR[base] + sign_extend(offset)] \leftarrow GPR[rt]。

(11) lw rt, offset(rs) 读存储器

100011	rs(5位)	rt(5位)	offset(16位)
--------	--------	--------	-------------

功能: GPR[rt] \leftarrow memory[GPR[base] + sign_extend(offset)]。

==> 分支指令

(12) beq rs,rt, offset

000100	rs(5位)	rt(5位)	offset(16位)
--------	--------	--------	-------------

功能: if(GPR[rs] = GPR[rt]) pc \leftarrow pc + 4 + sign_extend(offset) << 2
else pc \leftarrow pc + 4

特别说明: offset 是从 PC+4 地址开始和转移到的指令之间指令条数。offset 符号扩展之后左移 2 位再相加。为什么要左移 2 位? 由于跳转到的指令地址肯定是 4 的倍数 (每条指令占 4 个字节), 最低两位是“00”, 因此将 offset 放进指令码中的时候, 是右移了 2 位的, 也就是以上说的“指令之间指令条数”。

(13) bne rs,rt, offset

000101	rs(5位)	rt(5位)	offset(16位)
--------	--------	--------	-------------

功能: if(GPR[rs] != GPR[rt]) pc \leftarrow pc + 4 + sign_extend(offset) << 2
else pc \leftarrow pc + 4

(14) bltz rs, offset

000001	rs(5位)	00000	offset(16位)
--------	--------	-------	-------------

功能: if(GPR[rs] < 0) pc \leftarrow pc + 4 + sign_extend(offset) << 2
else pc \leftarrow pc + 4。

==> 跳转指令

(15) j addr

000010	addr(26位)
--------	-----------

功能: PC \leftarrow {PC[31:28], addr, 2'b0}, 无条件跳转。

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。

==> 停机指令

(16) halt

111111	000000000000000000000000(26 位)
--------	--------------------------------

功能：停机；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。

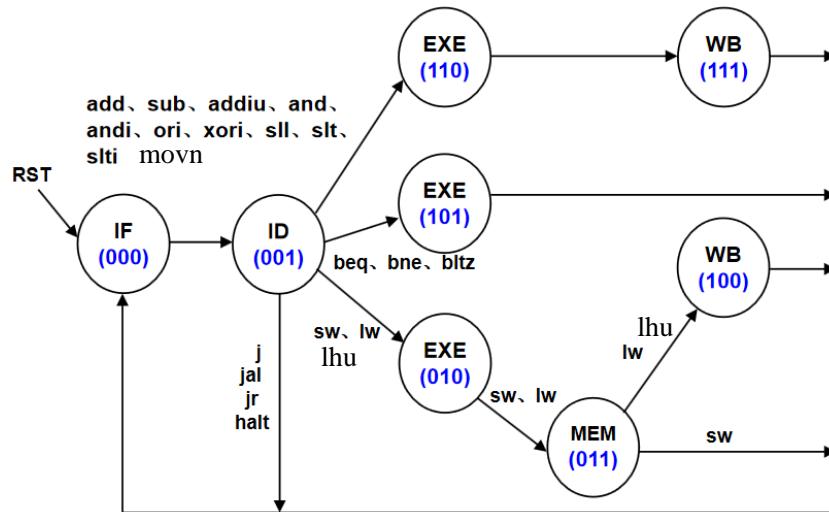


图 2 多周期 CPU 状态转移图

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。

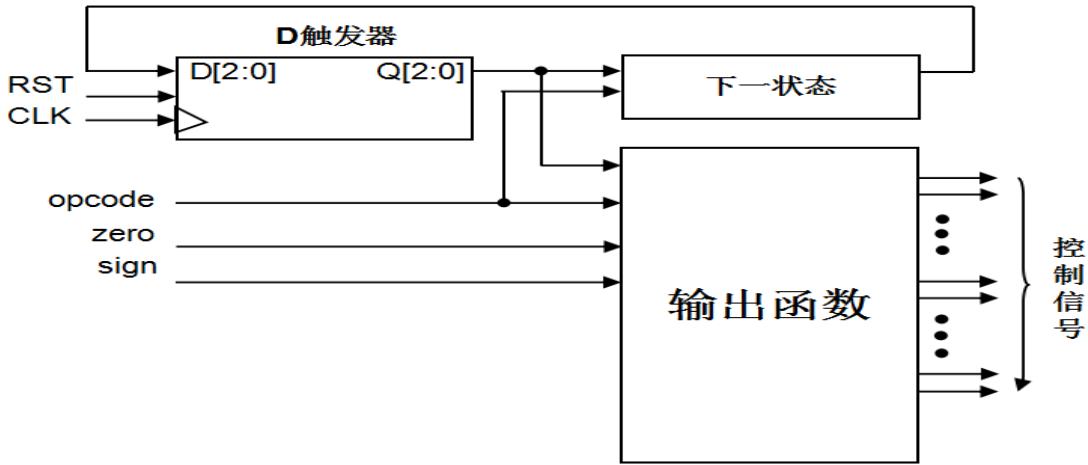


图 3 多周期 CPU 控制部件的原理结构图

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。

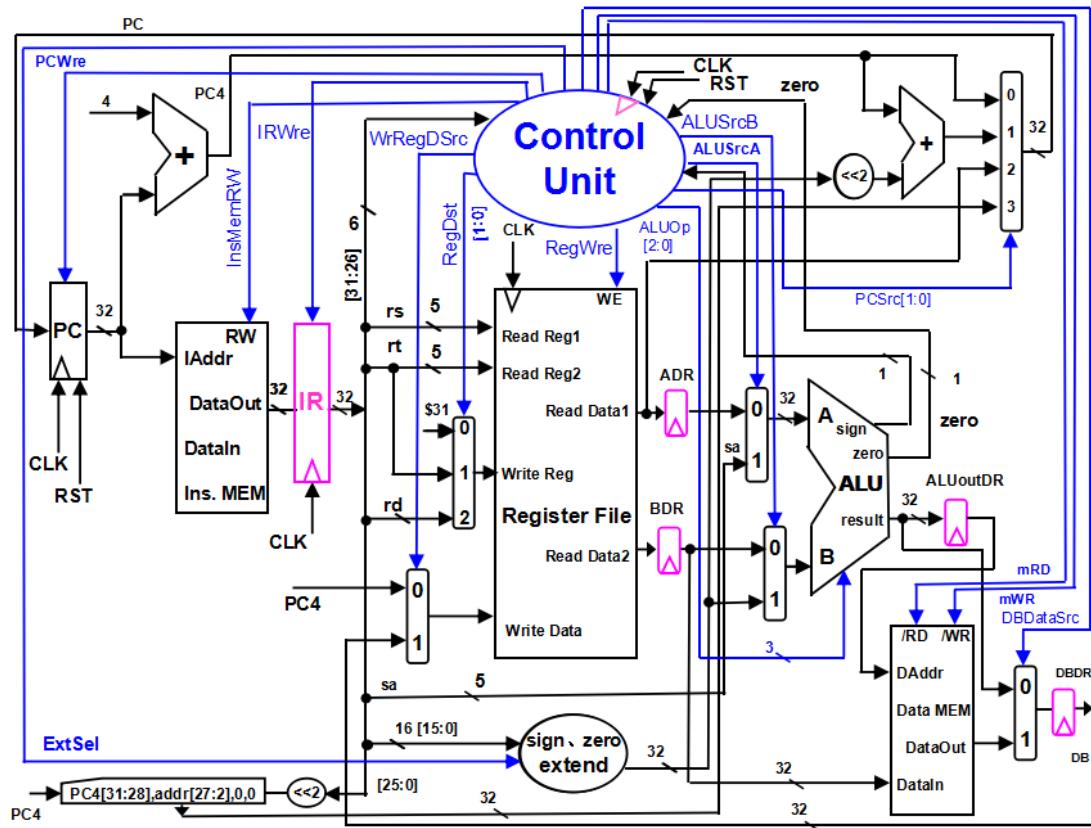


图 4 多周期 CPU 数据通路和控制线路图

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。

以下信号表仅供参考。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、addiu、and、andi、ori、xori、slt、slti、sw、lw、beq、bne、bltz	来自移位数 sa, 同时, 进行 (zero-extend)sa, 即 {27{1'b0},sa}, 相关指令: sll
ALUSrcB	来自寄存器堆 data2 输出, 相关指令: add、sub、and、slt、sll、beq、bne、bltz	来自 sign 或 zero 扩展的立即数, 相关指令: addiu、andi、ori、xori、slti、lw、sw、LHU、ADDI
DBDataSrc	来自 ALU 运算结果的输出, 相关指令: add、sub、addiu、and、andi、ori、xori、sll、slt、slti	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、bltz、j、sw、jr、halt	寄存器组寄存器写使能, 相关指令: add、sub、addiu、and、andi、ori、xori、sll、slt、slti、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、addiu、sub、and、andi、ori、xori、sll、slt、slti、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: andi、xori、ori;	(sign-extend)immediate, 相关指令: addiu、slti、lw、sw、beq、bne、bltz;
PCSrc[1..0]	00: pc<-pc+4, 相关指令: add、addiu、sub、and、andi、ori、xori、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0); 01: pc<-pc+4+(sign-extend)immediate ×4, 相关指令: beq(zero=1)、	

	bne(zero=0)、bltz(sign=1); 10: pc<-rs, 相关指令: jr; 11: pc<-[pc[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: addiu、andi、ori、xori、slti、lw; 10: rd 字段, 相关指令: add、sub、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
overflow, 判断溢出。

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 < B 不带符号
110	$Y = (((A < B) \&\& (A[31] == B[31])) ((A[31] == 1 \&\& B[31] == 0))) ? 1 : 0$	比较 A < B 带符号
111	$Y = A$ (与单周期不同)	方便实现 movn

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。

单周期实验中由于时延问题，写 PC 的时钟时上升沿，访存和访问寄存器组的时钟时下降沿。本次多周期实验，由于每个状态都有寄存器缓存值，所以所有时钟信号都在上升沿或下降沿时检测触发。

四. 实验器材

电脑一台，ModelSim软件一套，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

1. 确定主要模块：

首先先明确单周期CPU的整体设计思路，即为分模块设计，将CPU需要的各个模块分开设计，最后用一个顶层文件将它们连接起来。由于实验室提供的结构图有不完善或者过于繁琐的地方，所以我并没有完全按照这个图来设计，我将我设计的CPU分为了这些模块：

(1) Control Unit 控制单元，负责根据指令和输入的zero、sign来决定输出控制其他各个单元的控制信号。与单周期不同的是，多周期的控制单元多出来一个有限状态机的设计。

(2) ALU 算数逻辑单元，用作算数逻辑运算：根据控制信号从输入的数据中选取对应的操作数，并进行对应的运算操作并输出result、zero、sign，并且此次实验新增了一个溢出判断，即为输出overflow。

(3) PC 程序计数器，用来存放当前执行指令的地址。接收PC Choose模块传进来的下一条PC并等待输出。

(4) Instruction Memory 指令存储器，可以保存并输出指令。从文件读入所有的32位指令，并储存，根据输入的地址选择对应的指令输出。

(5) Data Memory 数据存储器，负责存储数据，在需要时输出数据。

(6) Register File 寄存器堆，储存程序所需要的临时数据。

(7) PC Choose 用来计算下一个指令的地址并传给PC。根据不同的PCSsrc信号选择不同的跳转方式，输出正确的下一条指令PC。

(8) Sign Zero Extension 位扩展，可以零扩展或符号扩展。

(9) Mux 多选器，用于从输入的多个数据中选一个输出。

(10) Top 顶层模块，用于将上述部分连接起来，构成真正的CPU。

(11) IR 指令寄存器，目的是使指令代码保持稳定。

(12) ADR、BDR、ALUoutDR、DBDR四个寄存器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。

2. 先按照指令要求，写出各阶段控制信号表：

		PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WrRegDSrc	InsMemRW	mRD	mWR	IRWre	ExtSel	PCSsrc[1:0]	RegDst[1:0]	ALUOp[2:0]
IF	x	0	X	X	X	0	X	1	0	0	1	X	X	X	X
ID	j	1	X	X	X	0	X	1	0	0	0	X	11	X	X
	jal	1	X	X	X	1	0	1	0	0	0	X	11	00	X
	jr	1	X	X	X	0	X	1	0	0	0	X	10	X	X
	halt	1	X	X	X	0	X	1	0	0	0	X	X	X	X
EXEa	add	0	0	0	X	0	X	1	0	0	0	X	X	X	000
	sub	0	0	0	X	0	X	1	0	0	0	1	X	X	001
	addiu	0	0	1	X	0	X	1	0	0	0	1	X	X	000
	addi	0	0	1	X	0	X	1	0	0	0	1	X	X	000
	andi	0	0	1	X	0	X	1	0	0	0	0	X	X	100
	and	0	0	0	X	0	X	1	0	0	0	X	X	X	100
	ori	0	0	1	X	0	X	1	0	0	0	0	X	X	011
	or	0	0	0	X	0	X	1	0	0	0	X	X	X	011
	sll	0	1	0	X	0	X	1	0	0	0	1	X	X	010
	slt	0	0	0	X	0	X	1	0	0	0	X	X	X	110
	slti	0	0	1	X	0	X	1	0	0	0	1	X	X	110
	movn	0	0	0	X	0	X	1	0	0	0	X	X	X	111
	beq	1	0	0	X	0	X	1	0	0	0	1	zero=1?01:00	X	001
	bne	1	0	0	X	0	X	1	0	0	0	1	zero=1?00:01	X	001
	bltz	1	0	0	X	0	X	1	0	0	0	1	sign=1?01:00	X	001
EXEl	sw	0	0	1	X	0	X	1	0	0	0	1	X	X	000
	lw	0	0	1	X	0	X	1	0	0	0	1	X	X	000
	lhu	0	0	1	X	0	X	1	0	0	0	0	X	X	000
MEM	sw	1	X	X	X	0	X	1	0	1	0	X	X	X	X
	lw	0	X	X	X	0	X	1	1	0	0	X	X	X	X
	lhu	0	X	X	X	0	X	1	1	0	0	X	X	X	X
WBa	立即数运算	1	X	X	0	OF=0?1:0	1	1	0	0	0	X	00	01	X
	寄存器运算	1	X	X	0	OF=0?1:0	1	1	0	0	0	X	00	10	X
WBm	lw	1	X	X	1	1	1	1	0	0	0	X	00	01	X
	lhw	1	X	X	1	1	1	1	0	0	0	X	00	01	X

3. Control Unit

模块说明：

Control Unit 控制单元，负责根据指令和输入的zero、sign来决定输出控制其他各个单元的控制信号。根据控制单元真值表列出表达式，达到信号控制的目的。

代码解释：

控制单元与单周期最大的不同就是加了有限状态机，控制state的变化，代码如下：

```

    always@(*)begin
        if(RST)begin
            case(state)
                IF:nextstate=ID;
                ID:begin
                    if(OP==beq||OP==bne||OP==bltz) nextstate<=EXEb;
                    else if(OP==sw||OP==lw||OP==lhu) nextstate<=EXEls;
                    else if(OP==j||OP==jal||(OP==Rtype&&func==jr)) nextstate<=IF;
                    else if(OP==halt);
                    else nextstate<=EXEa;
                end
                EXEa:nextstate<=WBa;
                EXEb:nextstate<=IF;
                EXEls:nextstate<=MEM;
                MEM:begin
                    if(OP==lw||OP==lhu) nextstate<=WBm;
                    else nextstate<=IF;
                end
                WBa:nextstate<=IF;
                WBm:nextstate<=IF;
                default:nextstate<=IF;
            endcase
        end
    end

```

接下来就和单周期一样，按照真值表来写出控制单元模块的代码，如下是我认为的核心部分，相当于用表达式翻译了一次真值表，这部分代码如下：

```

assign HALT=(OP==halt);
assign iflhu=(OP==lhu);
assign PCWre=(nextstate==IF && OP!=halt);
assign ALUSrcA=(OP==Rtype&&func==sll);
assign ALUSrcB=(OP==addiu||OP==andi||OP==ori||OP==slti||OP==sw||OP==lw||OP==addi||OP==lhu);
assign DBDataSrc=(OP==lw||OP==lhu);
assign RegWre=(state==ID&&OP==jal||(state==WBa&&(!overflow)&&(! (OP==Rtype&&func==movn&&B==0)))||(state==WBm));
assign WrRegDSrc=(OP!=jal);
assign InsMemRw=1;
assign mRD=(state==MEM&&(OP==lw||OP==lhu));
assign mWR=(state==MEM&&OP==sw);
assign RegDst[0]=(OP==lw||OP==lhu||OP==addiu||OP==addi||OP==andi||OP==ori||OP==slt);
assign RegDst[1]=(OP==Rtype&&OP!=jr);
assign IRWre=(state==IF);
assign ExtSel=(OP!=andi&&OP!=ori&&OP!=lhu);
assign ALUOp[0]=(OP==Rtype&&func==sub||OP==Rtype&&func==or_||OP==ori||OP==beq||OP==bne||OP==bltz||OP==Rtype&&func==movn);
assign ALUOp[1]=(OP==Rtype&&func==or_||OP==ori||OP==Rtype&&func==sll||OP==slt||OP==Rtype&&func==slt||OP==Rtype&&func==movn);
assign ALUOp[2]=(OP==andi||OP==Rtype&&func==and_||OP==slt||OP==Rtype&&func==slt||OP==Rtype&&func==movn);
assign PCSrc[0]=(OP==beq&&zero==1||OP==bne&&zero==0||OP==bltz&&sign==1||OP==j||OP==jal);
assign PCSrc[1]=(OP==j||(OP==Rtype&&func==jr)||OP==jal);
assign ifNeedOf=(OP==add||OP==sub||OP==addi);

```

4. ALU

模块说明：

ALU 算数逻辑单元，用作算数逻辑运算：根据控制信号从输入的数据中选取对应的操作数，并进行对应的运算操作并输出result、zero、sign、overflow。

代码解释：

在输入的所有值中任意一个变化时，ALU就开始根据输入的ALUOp控制信号执行不同的运算，此外我将movn也设置在了ALU模块中，在操作码为movn时，ALUOp将会变成111，result是A，用不用写会寄存器交由控制单元的判断。此外有设计了溢出判断，输出overflow信号。 ALU模块核心部分如下：

```

reg [32:0] temp;
assign sign=result[31];
assign zero=(result==0);
assign overflow=(ifNeedOf&&A[31]^B[31]^result[31]^temp[32]);
always @(*) begin
    case(ALUOp)
        3'b000:begin
            result<=A+B;
            temp<=A+B;
        end
        3'b001:begin
            result<=A-B;
            temp<=A-B;
        end
        3'b010:result<=B<<A;
        3'b011:result<=A|B;
        3'b100:result<=A&B;
        3'b101:result<=(A<B)?1:0;
        3'b110:result<=(((A<B)&&(A[31]==B[31]))||((A[31]==1&&B[31]==0)))?1:0;
        3'b111:result<=A;
        default:begin
            result=0;
        end
    endcase
end

```

5. Instruction memory

模块说明：

Instruction Memory指令存储器，可以保存并输出指令。从文件读入所有的32位指令，并储存，根据输入的地址选择对应的指令输出。输出后将指令传给其他部分进行处理。

代码解释：

此部分和单周期基本一样。指令存储器是将从文件里读入的指令，根据输入的PC的值来选择指令进行输出，因为助教说无法32位一起读入，所以只能切割成8位的，再拼接成32位的，当RW或Iaddr值改变时，并且RW控制信号为1时，读取指令。代码的核心部分如下：

```

reg[7:0] instruction[0:255];
initial begin
    $readmemh("input.txt",instruction); //vivado烧板时需要改为绝对路径
end
always@(RW or IAddr)begin
    if(RW)begin
        IDataOut[31:24]=instruction[IAddr];
        IDataOut[23:16]=instruction[IAddr+1];
        IDataOut[15:8]=instruction[IAddr+2];
        IDataOut[7:0]=instruction[IAddr+3];
    end
end

```

6. Data Memory

模块说明：

数据储存器负责在时钟上升沿时，根据控制信号将传入的数据根据传入的地址写入并保存，当读取时根据控制信号和传入的地址输出正确的数据。

代码解释：

数据存储器主要负责数据的读写与储存。在我设计的模块中，当RD和DAddr中任何一个值有改变时，并且此时RD信号为1时，就开始读取。当时钟上升沿，并且WR信号为1时，就开始写。设计核心部分如下：

```

always@(RD or DAddr)begin
    if(RD)begin
        DataOut[31:24]=data[DAddr];
        DataOut[23:16]=data[DAddr+1];
        DataOut[15:8]=data[DAddr+2];
        DataOut[7:0]=data[DAddr+3];
    end
end
always@(posedge CLK)begin
    if(WR)begin
        data[DAddr]=DataIn[31:24];
        data[DAddr+1]=DataIn[23:16];
        data[DAddr+2]=DataIn[15:8];
        data[DAddr+3]=DataIn[7:0];
    end
end

```

7. PC

模块说明：

PC 程序计数器，用来存放当前执行指令的地址。接收PC Choose模块传进来的下一条PC并等待输出。

代码解释：

PC程序计数器模块先初始化了PC值为0，然后在时钟上升沿或者是Reset信号下降沿开始，如果Reset为0，则PC清零，如果Reset不为0，则下一个输出的PC就是经过PCchoose传进来的新的PC值。PC模块核心部分如下：

```

initial begin
    nextPC<=0;
end
always@(posedge CLK or negedge Reset)begin
    if(Reset==0)begin
        nextPC=0;
    end
    if(Reset!=0&&PCWre==1)begin
        nextPC=inPC;
    end
end

```

8. PCchoose**模块说明：**

用来计算下一个指令的地址并传给PC。根据不同的PCSrc信号选择不同的跳转方式，输出正确的下一条指令PC。

代码解释：

PCchoose模块负责根据接收到的控制信号来计算下一个PC值，这一模块和图中的有较大的不同，相当于把图中右上角和左下角的几个模块合为一体，我认为这样更简洁一些。其中当控制信号PCsrc为00时，nextPC<=PC4；当控制信号PCsrc为01时，nextPC<=PC4+singedImmediate*4；当控制信号PCsrc为10时，nextPC<={PC4[31:28],addr[25:0],2'b00}。该模块设计的核心部分如下：

```

initial begin
    nextPC<=0;
end
wire[31:0] PC4=curPC+4;
always@(*)begin
    if(!RST) nextPC<=0;
    else begin
        if(PCSsrc==2'b00)begin
            nextPC<=PC4;
        end
        if(PCSsrc==2'b01)begin
            nextPC<=PC4+signedImmediate*4;
        end
        if(PCSsrc==2'b10)begin
            nextPC<=rs;
        end
        if(PCSsrc==2'b11&&!HALT)begin
            nextPC<={PC4[31:28],addr[25:0],2'b00};
        end
    end
end

```

9. Register File

模块说明:

通用寄存器组中存放数据都是32位的，有32个寄存器。负责存放程序需要用到的临时数据。

根据控制信号和输入的寄存器号进行对应的读写操作。

代码解释:

通用寄存器组主要负责寄存器的读写和对数据的存储，一共有32个，这里需要注意一点就是0号寄存器的值是不能改变的，它一直为0，所以在写代码时需要特别注意一下0号寄存器。其余就是根据传入的地址去找到相应的寄存器，然后进行读写操作。与上次实验不同的是，为了实现lhu，我直接把0扩展步骤加入到寄存器组的写操作中去。部分代码的核心部分如下：

```

always@(posedge CLK or negedge RST)begin
    if(RST==0)begin
        for(i=0;i<32;i=i+1)register[i]<=0;
    end
    if (WriteReg && WE) begin
        if(iflhu==1) register[WriteReg]<={16'b0000000000000000,WriteData[31:16]};
        else register[WriteReg]<=WriteData;
    end
end

```

10. 零扩展和符号位扩展部分非常简单，就是根据传入的控制信号，对16位的数据进行零

扩展或符号位扩展，成为32位的数据。代码核心部分如下：

```
always@(*)begin
    extended[15:0]<=immediate;
    if(ExtSel==0)begin
        extended[31:16]<=16'h0000;
    end
    else begin
        extended[31:16]<=immediate[15] ? 16'hffff : 16'h0000;
    end
end
```

11. Mux多选器我写了两种，一种是选择32位数据的，一种是选择5位数据的，思路很简单，就不赘述了，五位的代码如下，32位的同理：

```
module Mux5(
    input choice,
    input[4:0] in0,
    input[4:0] in1,
    output[4:0] out
);
    assign out=choice?in1:in0;
endmodule
```

12. IR 指令寄存器，目的是使指令代码保持稳定。在时钟上升沿输出发生变化：

```
always@(posedge CLK or negedge RST)begin
    if(RST==0) IROut<=ins_in;
    else if(IRWre) IROut<=ins_in;
end
```

13. ADR、BDR、ALUoutDR、DBDR四个寄存器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。也是在时钟上升沿变化：

```
always@(posedge CLK)begin
    out<=in;
end
```

14. 烧板部分代码说明见后文烧板部分。

下面以测试文件3中的指令为例来分析16种不同指令的波形是否正确：

程序测试段3：

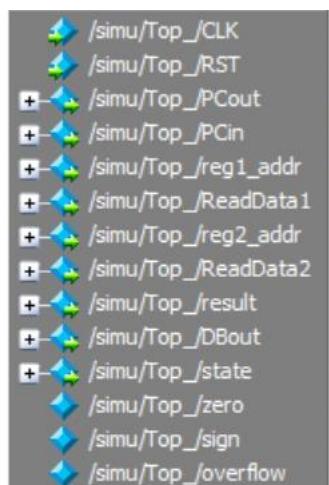
地址	汇编程序	指令代码					16 进制数代码
		op(6)	rs(5)	rtt(5)	rd(5)/immediate (16)		
0x00000000	addiu \$1,\$0,10	001001	00000	00001	0000000000001010	=	24 01 00 0a
0x00000004	addiu \$2,\$0,-3	001001	00000	00010	1111111111111101		24 02 ff fd
0x00000008	and \$1,\$1,\$2	000000	00001	00010	0000100000100100		00 22 08 24
0x0000000C	ori \$1,\$1,5	001101	00001	00001	000000000000000101		34 21 00 05
0x00000010	andi \$2,\$2,15	001100	00010	00010	0000000000000111		30 42 00 0f
0x00000014	sll \$1,\$1,1	000000	00000	00001	0000100001000000		00 01 08 40
0x00000018	or \$1,\$1,\$2	000000	00001	00010	0000100000100101		00 22 08 25
0x0000001C	slti \$3,\$1,31	001010	00001	00011	0000000000001111		28 23 00 1f
0x00000020	slti \$4,\$1,32	001010	00001	00100	000000000000100000		28 24 00 20
0x00000024	add \$1,\$2,\$4	000000	00010	00100	0000100000100000		00 44 08 20
0x00000028	sub \$1,\$2,\$1	000000	00010	00001	0000100000100010		00 41 08 22
0x0000002C	addiu \$1,\$1,1	001001	00001	00001	0000000000000001		24 21 00 01
0x00000030	addiu \$2,\$2,-13	001001	00010	00010	1111111111110011		24 42 ff f3
0x00000034	addiu \$3,\$3,-0	001001	00011	00011	0000000000000000		24 63 00 00
0x00000038	addiu \$4,\$4,-1	001001	00100	00100	1111111111111111		24 84 ff ff
0x0000003C	or \$5,\$1,\$2	000000	00001	00010	0010100000100101		00 22 28 25
0x00000040	or \$5,\$5,\$3	000000	00101	00011	0010100000100101		00 a3 28 25
0x00000044	or \$5,\$5,\$4	000000	00101	00100	0010100000100101		00 a4 28 25
0x00000048	addiu \$0,\$0,-2	001001	00000	00000	1111111111111110		24 00 ff fe
0x0000004C	beq \$5,\$0,1	000100	00101	00000	0000000000000001		10 a0 00 01
0x00000050	halt	111111	00000	00000	0000000000000000		fc 00 00 00
0x00000054	addiu \$1,\$0,0	001001	00000	00001	0000000000000000		24 01 00 00
0x00000058	addiu \$2,\$0,-1	001001	00000	00010	1111111111111111		24 02 ff ff
0x0000005C	sw \$2,0(\$1)	101011	00001	00010	0000000000000000		ac 22 00 00
0x00000060	lw \$1,0(\$1)	100011	00001	00001	0000000000000000		8c 21 00 00
0x00000064	bne \$1,\$2, 3	000101	00001	00010	000000000000000101		14 22 00 05
0x00000068	bltz \$1,1	000001	00001	00000	0000000000000001		04 20 00 01
0x0000006C	halt	111111	00000	00000	0000000000000000		fc 00 00 00
0x00000070	j 0x1e(0x78)	000010	00000	00000	0000000000001110		08 00 00 1e
0x00000074	halt	111111	00000	00000	0000000000000000		fc 00 00 00
0x00000078	jal 0x30(0xC0)	000011	00000	00000	000000000000110000		0c 00 00 30
0x0000007C	addi \$1,\$0,32767	001000	00000	00001	0111111111111111		20 01 7f ff
0x00000080	sll \$1,\$1,16	000000	00000	00001	0000100000000000		00 01 0c 00
0x00000084	addi \$1,\$1,32767	001000	00001	00001	0111111111111111		20 21 7f ff
0x00000088	addi \$1,\$1,32767	001000	00001	00001	0111111111111111		20 21 7f ff

0x0000008C	addi \$2,\$1,0	001000	00001	00010	0000000000000000		20 22 00 00
0x00000090	addi \$2,\$2,2	001000	00010	00010	0000000000000010		20 42 00 02
0x00000094	slt \$3,\$2,\$1	000000	00010	00001	0001100000101010		00 41 18 2a
0x00000098	addi \$1,\$0,1	001000	00000	00001	0000000000000001		20 01 00 01
0x0000009C	addi \$2,\$0,2	001000	00000	00010	00000000000000010		20 02 00 02
0x000000A0	addi \$3,\$0,3	001000	00000	00011	00000000000000011		20 03 00 03
0x000000A4	movn \$1,\$3,\$0	000000	00011	00000	0000100000001011		00 60 08 0b
0x000000A8	movn \$2,\$3,\$2	000000	00011	00010	0001000000001011		00 62 10 0b
0x000000AC	addiu \$1,\$0,0	001001	00000	00001	0000000000000000		24 01 00 00
0x000000B0	sll \$2,\$2,16	000000	00000	00010	0001010000000000		00 02 14 00
0x000000B4	sw \$2,8(\$1)	101011	00001	00010	0000000000001000		ac 22 00 08
0x000000B8	lhu \$3,10(\$1)	100101	00001	00011	0000000000001010		94 23 00 0a
0x000000BC	halt	111111	00000	00000	0000000000000000		fc 00 00 00
0x000000C0	jr \$31	000000	11111	00000	0000000000001000		03 e0 00 08

指令对应波形分析：

为了验证CPU设计的正确性，可以通过检查一些输入输出数据以及控制信号的波形来判断，

我这里将要分析的数据及部分信号如下：



其中PCout、PCin分别指当前指令、下一条指令。reg1_addr, readData1, reg2_addr, readData2分别对应读寄存器组的两个寄存器的编号和读取结果: readreg1, readData1, readreg2, readData2。DBout指要写入寄存器组的内容。Result指ALU运算的结果, zero、sign是ALU输出的判断是否为0和正负号的结果。Overflow值ALU运算结果是否溢出。其他是一系列控制信号, 命名与要求一致, 不做赘述。

下面开始逐条分析：



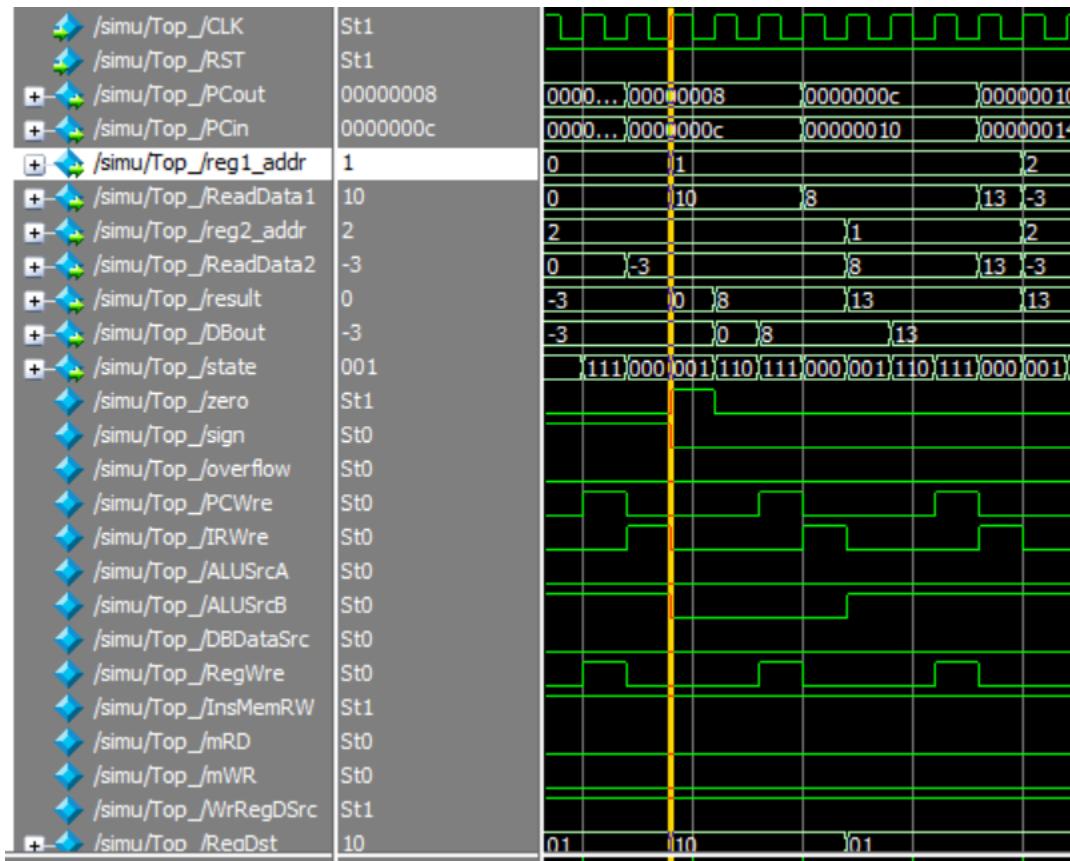
这条指令是要把\$0中的值加上立即数10的结果写到1号寄存器中，不需要判断溢出。

如上图所示，在执行第一条指令addiu时，在state为000阶段即IF阶段时，取出PC在0x00的指令，在state为001即ID阶段时，得到PCin（即为下一指令地址）为0x04，顺序执行，符合要求，并且可以看出在ID阶段时，0号寄存器和1号寄存器中的值都为0。当state为110即EXEa阶段时，ALU的结果为10，即为算出0+10的结果，正确。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为10。经检查，其他控制信号也符合预期。

0x00000004	addiu \$2,\$0,-3	001001	00000	00010	111111111111101		24 02 ff fd
------------	------------------	--------	-------	-------	-----------------	--	-------------

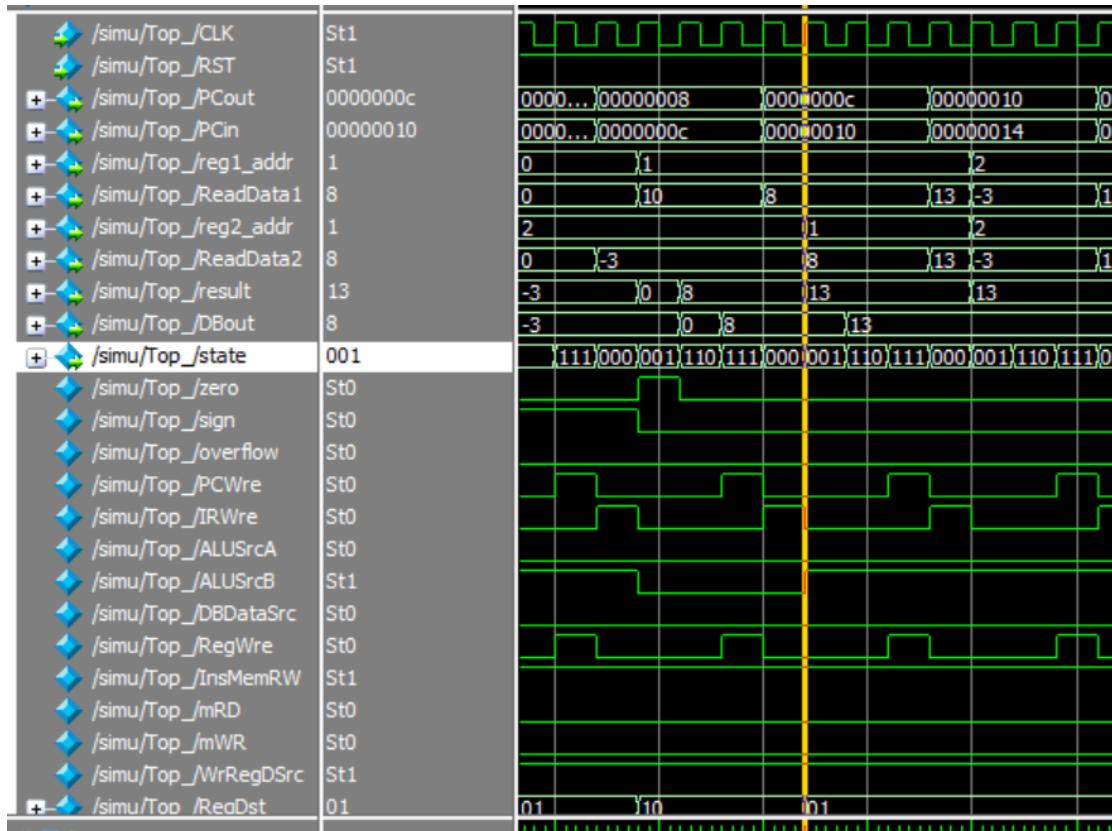
第二条指令也是addiu，与上述过程同理，不做赘述。执行完这一条后，2号寄存器中的值为-3。

0x00000008	and \$1,\$1,\$2	000000	00001	00010	0000100000100100		00 22 08 24
------------	-----------------	--------	-------	-------	------------------	--	-------------



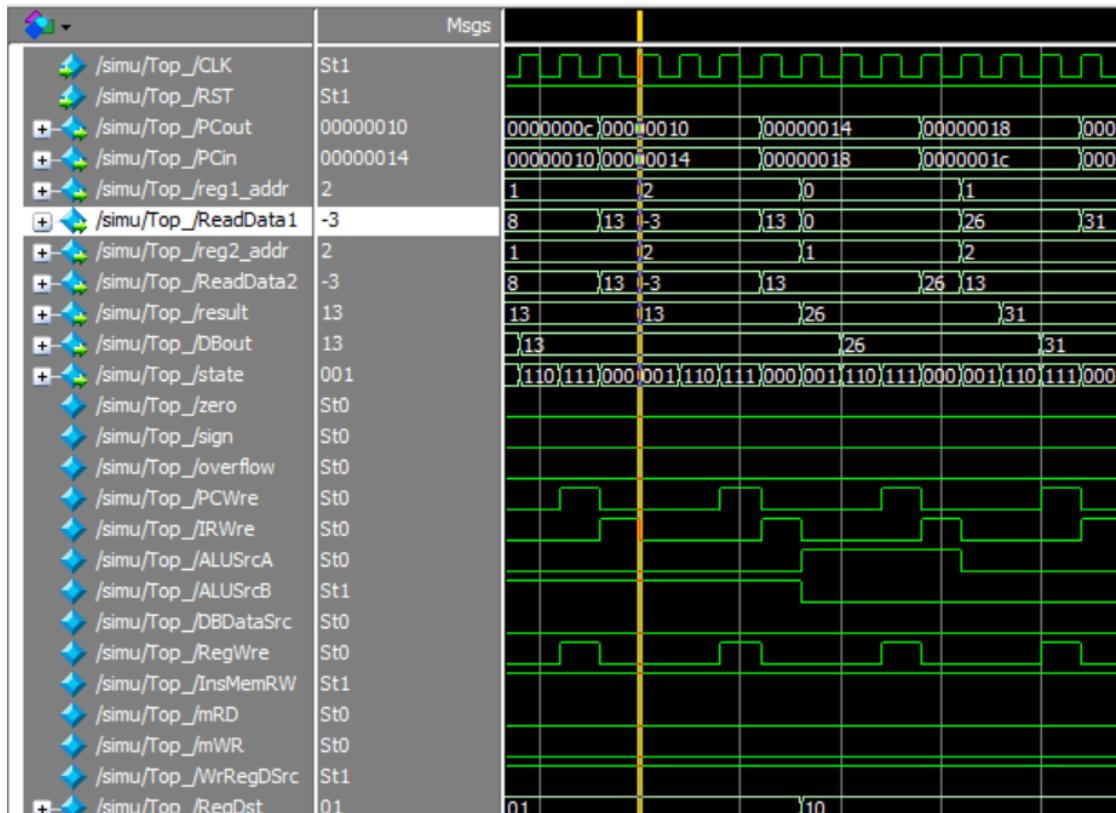
这一条and指令是要把\$1中的值与\$2中的值按位与，将结果写到1号寄存器中。如上图所示，在state为000阶段即IF阶段时，取出Pcout（当前指令地址）在0x08的指令，在state为001即ID阶段时，可以看出PCin（即为下一指令地址）为0xC，顺序执行，符合要求，并且可以看出在ID阶段时，1号寄存器的值为10，2号寄存器中的值为-3。当state为110即EXEa阶段时，ALU的结果为8，即为算出 10 与 -3 的结果，正确。当state为111(WBa) 时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为8。经检查，其他控制信号也符合预期。

0x0000000C	ori \$1,\$1,5	001101	00001	00001	0000000000000101		34 21 00 05
------------	---------------	--------	-------	-------	------------------	--	-------------



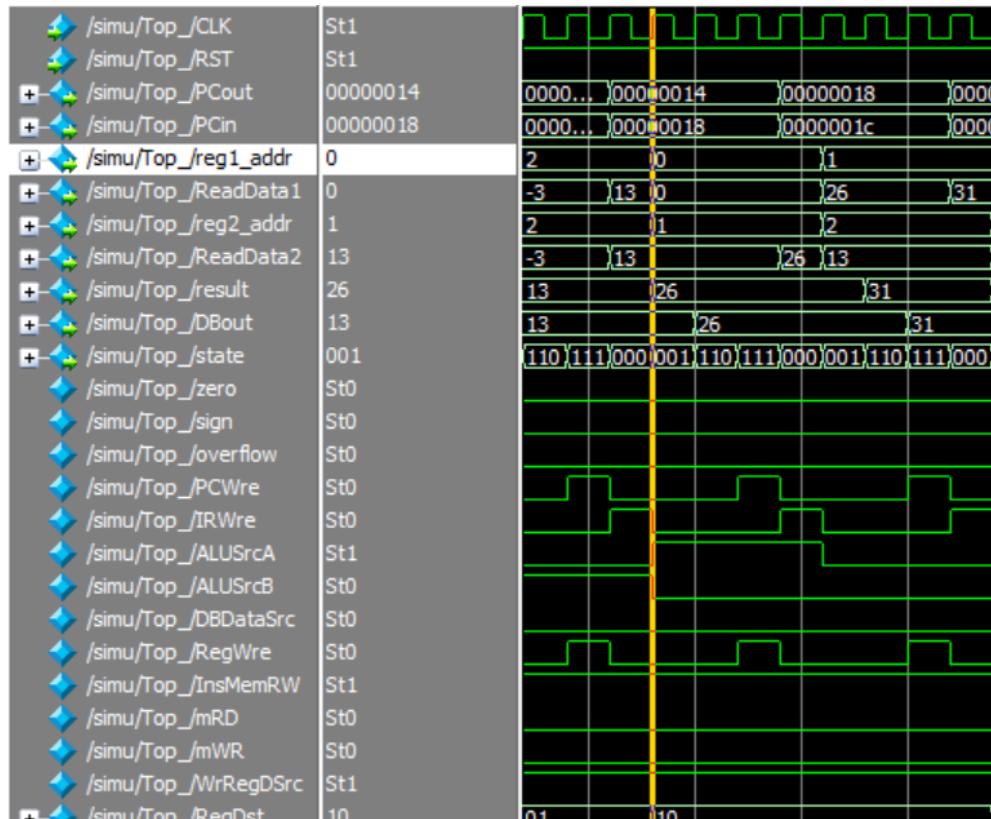
这条指令是要把\$1中的值与立即数5按位或，结果写到1号寄存器中。如上图所示，在执行这条指令的过程中，在state为000阶段即IF阶段时，取出PC在0x0c的指令，在state为001即ID阶段时，可以看出PCin（即为下一指令地址）为0x10，顺序执行，符合要求，并且可以看出在ID阶段时，1号寄存器中的值为8。当state为110即EXEa阶段时，ALU的结果为13，即为算出 8或5 的结果，正确。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为13。经检查，其他控制信号也符合预期。

0x000000010	andi \$2,\$2,15	001100	00010	00010	0000000000001111		30 42 00 0f
-------------	-----------------	--------	-------	-------	------------------	--	-------------



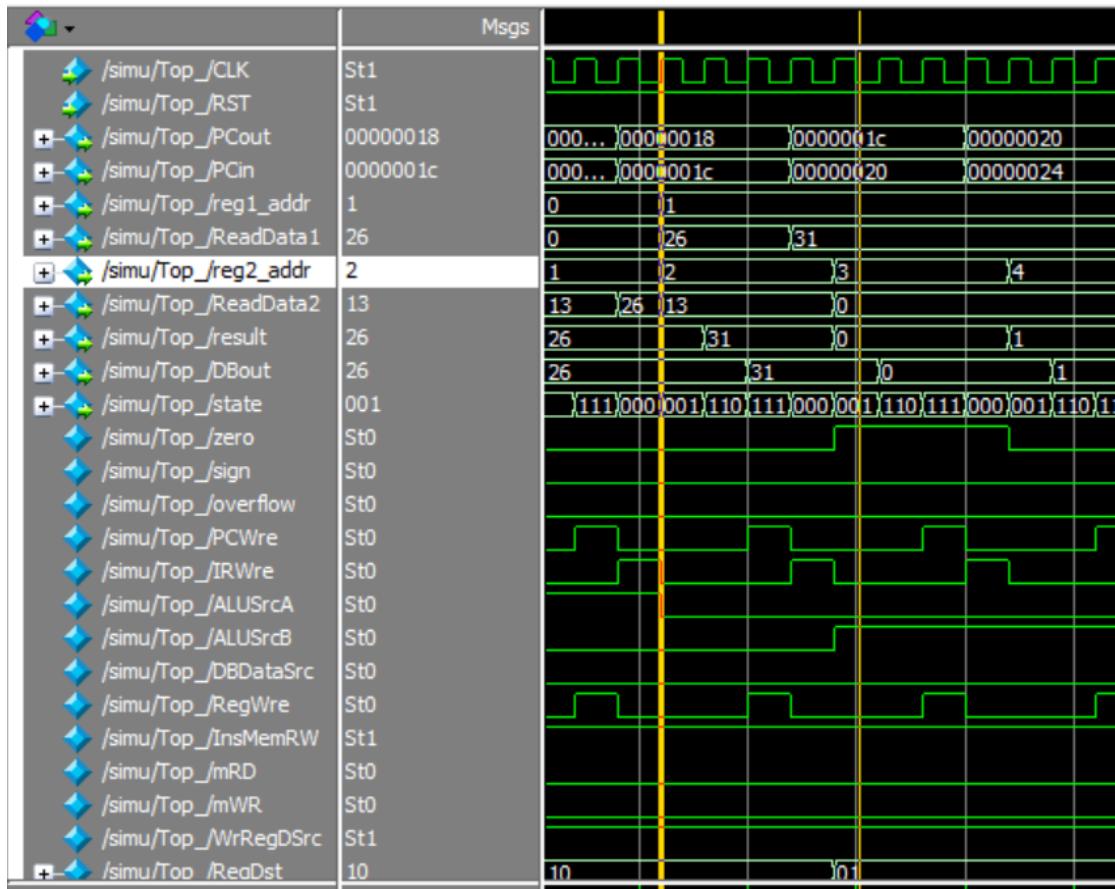
这条指令是要把\$2中的值与立即数15按位与，结果写到2号寄存器中。如上图所示，在执行这条指令的过程中，在state为000阶段即IF阶段时，取出PC在0x10的指令，在state为001即ID阶段时，可以看出PCin（即为下一指令地址）为0x14，顺序执行，符合要求，并且可以看出在ID阶段时，2号寄存器中的值为-3。当state为110即EXEa阶段时，ALU的结果为13，即为算出 -3与5 的结果，正确。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器2中的值变为13。经检查，其他控制信号也符合预期。

0x000000014	sll \$1,\$1,1	000000	00000	00001	0000100001000000		00 01 08 40
-------------	---------------	--------	-------	-------	------------------	--	-------------

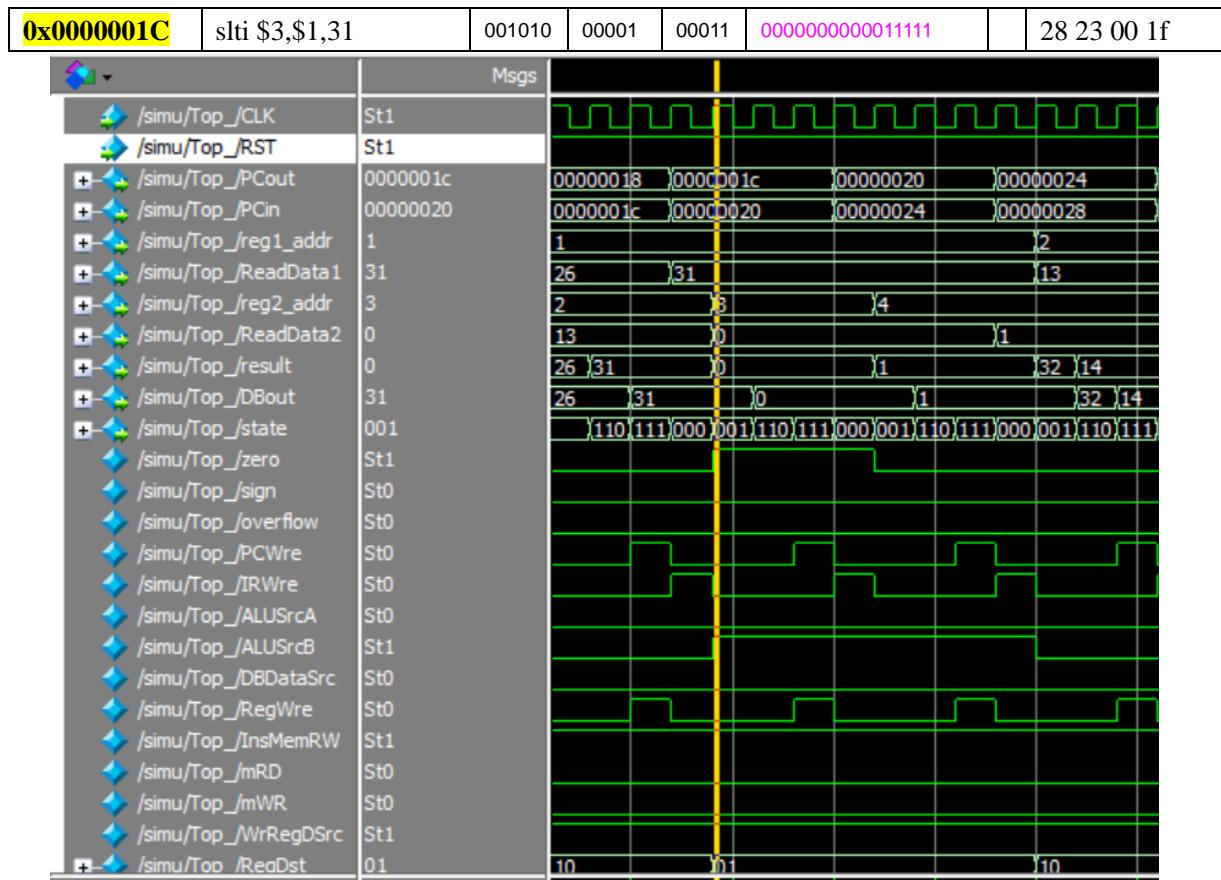


这一条sll指令是要把\$1中的值左移一位，将结果写到1号寄存器中。如上图所示，在state为000阶段即IF阶段时，取出PCout（当前指令地址）在0x14的指令，在state为001即ID阶段时，可以看出PCin（即为下一指令地址）为0x18，顺序执行，符合要求，并且可以看出在ID阶段时，1号寄存器的值为13。当state为110即EXEa阶段时，ALU的结果为26，即为 13×2 的结果，正确。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为26。经检查，其他控制信号也符合预期。

0x00000018 or \$1,\$1,\$2 000000 00001 00010 **0000100000100101** | 00 22 08 25



这一条or指令是要把\$1中的值与\$2中的值按位或，将结果写到1号寄存器中。如上图所示，在state为000阶段即IF阶段时，取出PCout（当前指令地址）在0x18的指令，在state为001即ID阶段时，可以看出PCin（即为下一指令地址）为0x1C，顺序执行，符合要求，并且可以看出在ID阶段时，1号寄存器的值为26，2号寄存器中的值为13。当state为110即EXEa阶段时，ALU的结果为31，即为算出 26 或 13 的结果，正确。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为31。经检查，其他控制信号也符合预期。

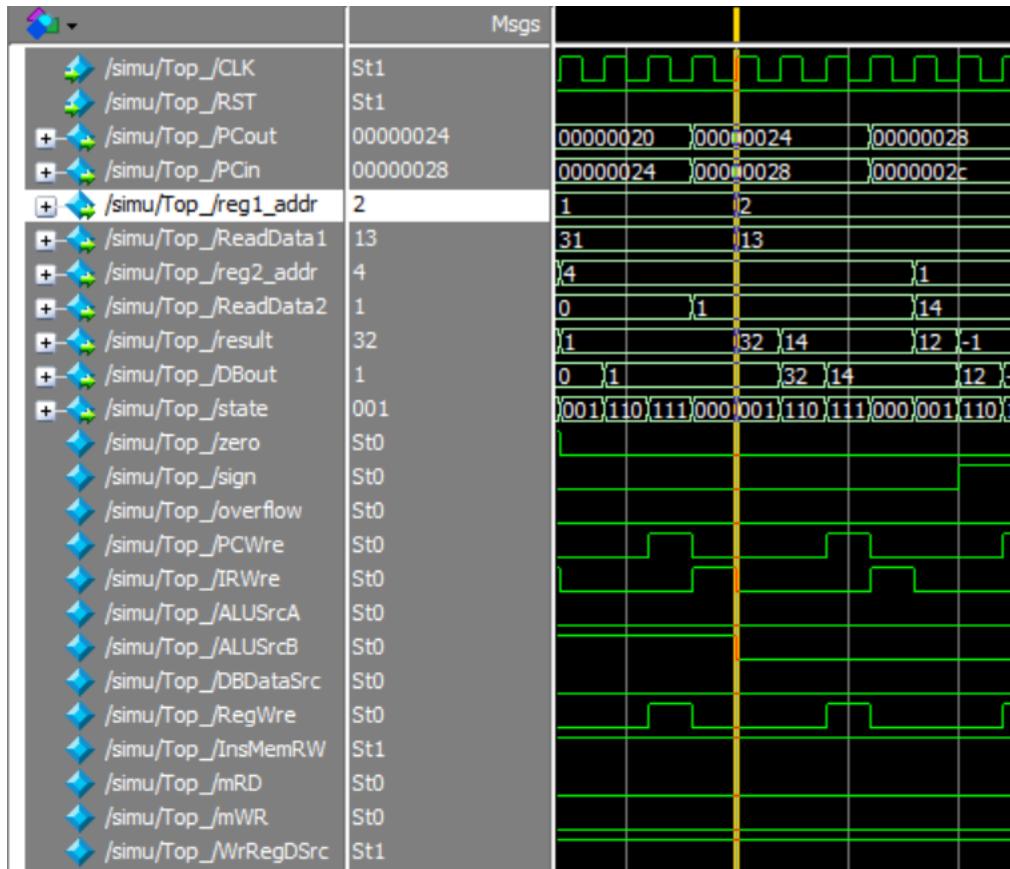


Slti, 小于则置位，本条指令目的是比较1号寄存器中的值与立即数31，如果1号寄存器中的值小于31，则将结果1写入3号寄存器，否则，将结果0写入3号寄存器。如上图所示，在state为000阶段即IF阶段时，取出PCout（当前指令地址）在0x1c的指令，在state为001即ID阶段时，可以看出PCin（即为下一指令地址）为0x20，顺序执行，符合要求，并且可以看出在ID阶段时，1号寄存器的值为31，等于立即数31，所以在EXE（110）阶段，ALU作带符号比较结果（result）为0，当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组。执行完这一条后，寄存器3中的值仍为0。经检查，其他控制信号也符合预期。

0x000000020	slti \$4,\$1,32	001010	00001	00100	000000000000100000		28 24 00 20
-------------	-----------------	--------	-------	-------	--------------------	--	-------------

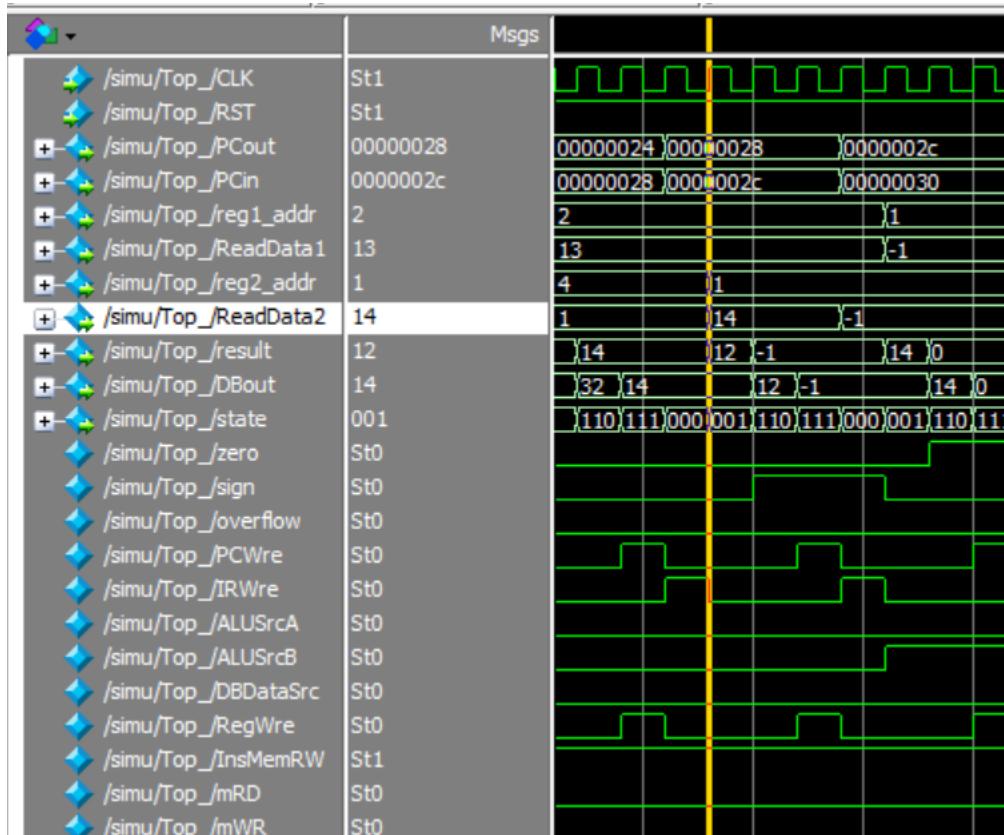
这一指令同上，不做赘述。执行之后4号寄存器值为1。

0x00000024	add \$1,\$2,\$4	000000	00010	00100	0000100000100000		00 44 08 20
------------	-----------------	--------	-------	-------	------------------	--	-------------



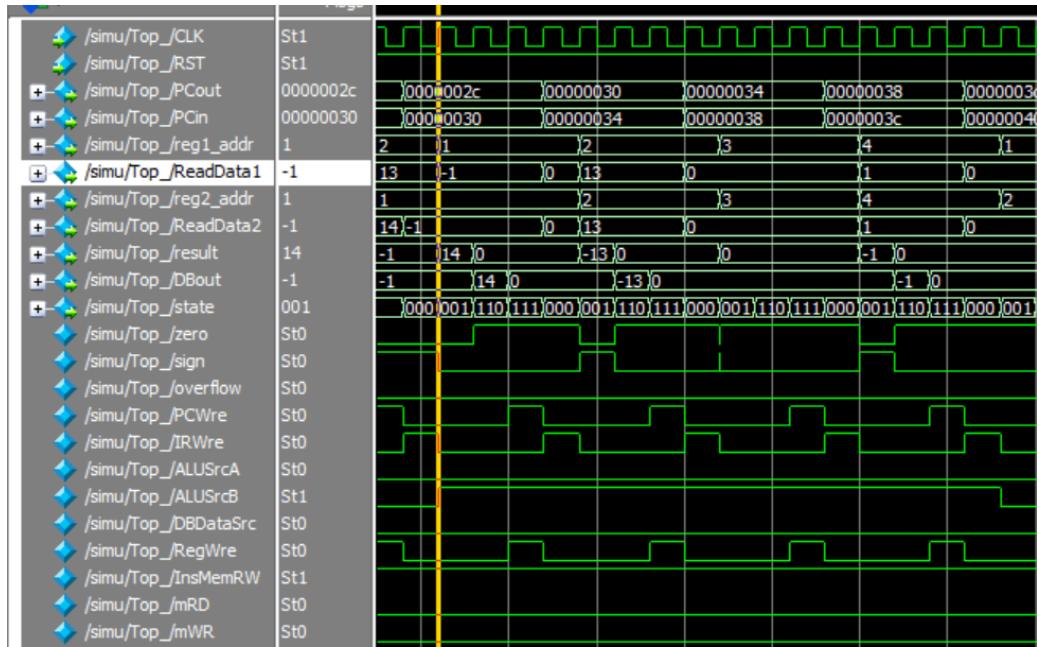
这条指令是要把2号寄存器中的值与4号寄存器中的值相加，结果写到1号寄存器中，并且需要判断溢出。如上图所示，在本条指令state为000阶段即IF阶段时，取出PC在0x24的指令，在state为001即ID阶段时，可见PCin（即为下一指令地址）为0x28，顺序执行，符合要求，并且可以看出在ID阶段时，2号寄存器值为13，4号寄存器值为1。当state为110即EXEa阶段时，ALU的结果为14，即为算13+1的结果，正确，并且溢出overflow为0，没有溢出。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为14。经检查，其他控制信号也符合预期。

0x00000028	sub \$1,\$2,\$1	000000	00010	00001	0000100000100010		00 41 08 22
------------	-----------------	--------	-------	-------	------------------	--	-------------



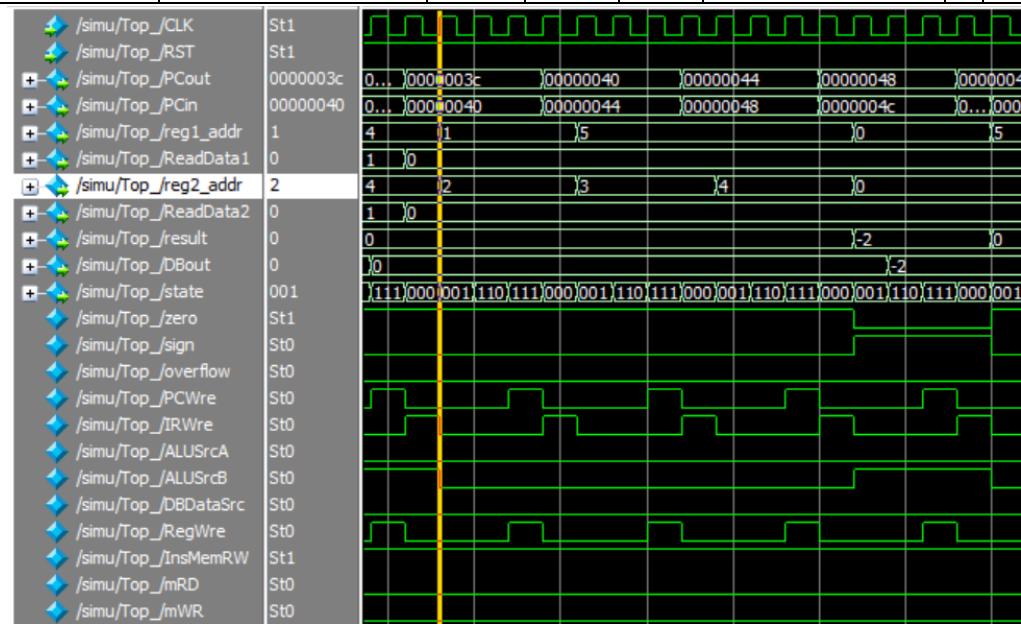
这条指令是要把2号寄存器中的值与1号寄存器中的值相减，结果写到1号寄存器中，并且需要判断溢出。如上图所示，在本条指令state为000阶段即IF阶段时，取出PC在0x28的指令，在state为001即ID阶段时，可见PCin（即为下一指令地址）为0x2c，顺序执行，符合要求，并且可以看出在ID阶段时，2号寄存器值为13，1号寄存器值为14。当state为110即EXEa阶段时，ALU的结果为-1，即为算13-14的结果，正确，并且溢出overflow为0，说明没有溢出。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为-1。经检查，其他控制信号也符合预期。

0x0000002C	addiu \$1,\$1,1	001001	00001	00001	0000000000000001		24 21 00 01
0x00000030	addiu \$2,\$2,-13	001001	00010	00010	1111111111110011		24 42 ff f3
0x00000034	addiu \$3,\$3,-0	001001	00011	00011	0000000000000000		24 63 00 00
0x00000038	addiu \$4,\$4,-1	001001	00100	00100	1111111111111111		24 84 ff ff



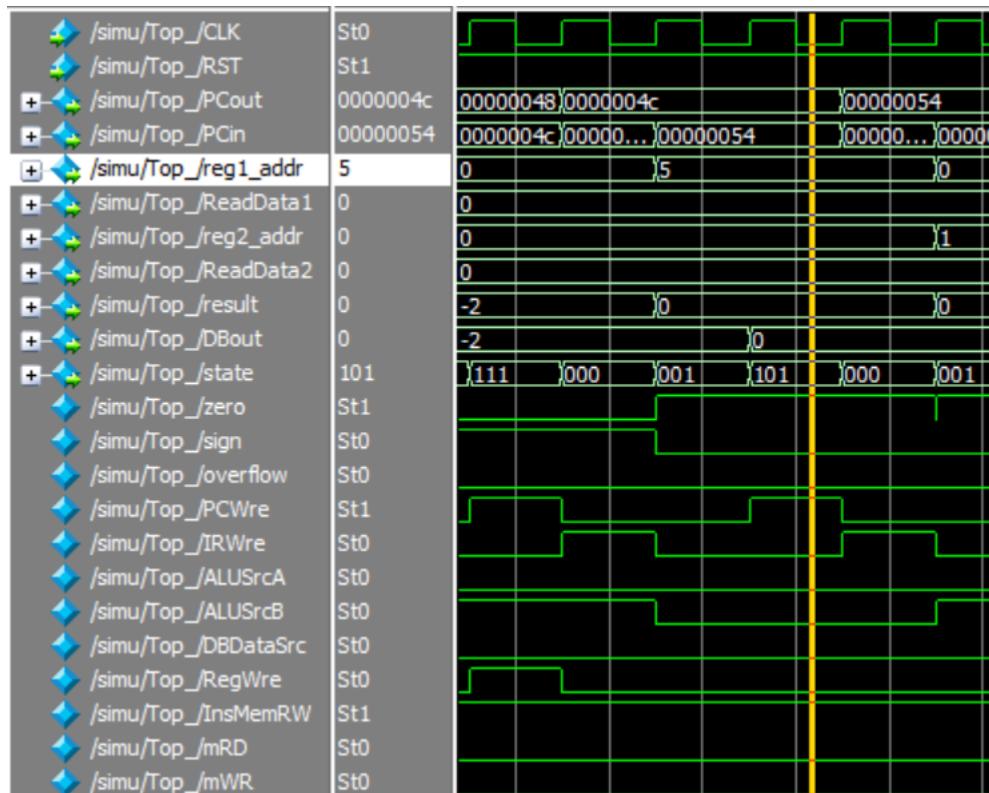
这四条addiu指令原理在之前已经分析过，这里不做赘述。执行完这四条之后，1号寄存器中的值为0，2号寄存器中的值为0，3号寄存器中的值为0，4号寄存器中的值为0。

0x0000003C	or \$5,\$1,\$2	000000	00001	00010	0010100000100101		00 22 28 25
0x00000040	or \$5,\$5,\$3	000000	00101	00011	0010100000100101		00 a3 28 25
0x00000044	or \$5,\$5,\$4	000000	00101	00100	0010100000100101		00 a4 28 25
0x00000048	addiu \$0,\$0,-2	001001	00000	00000	1111111111111110		24 00 ff fe



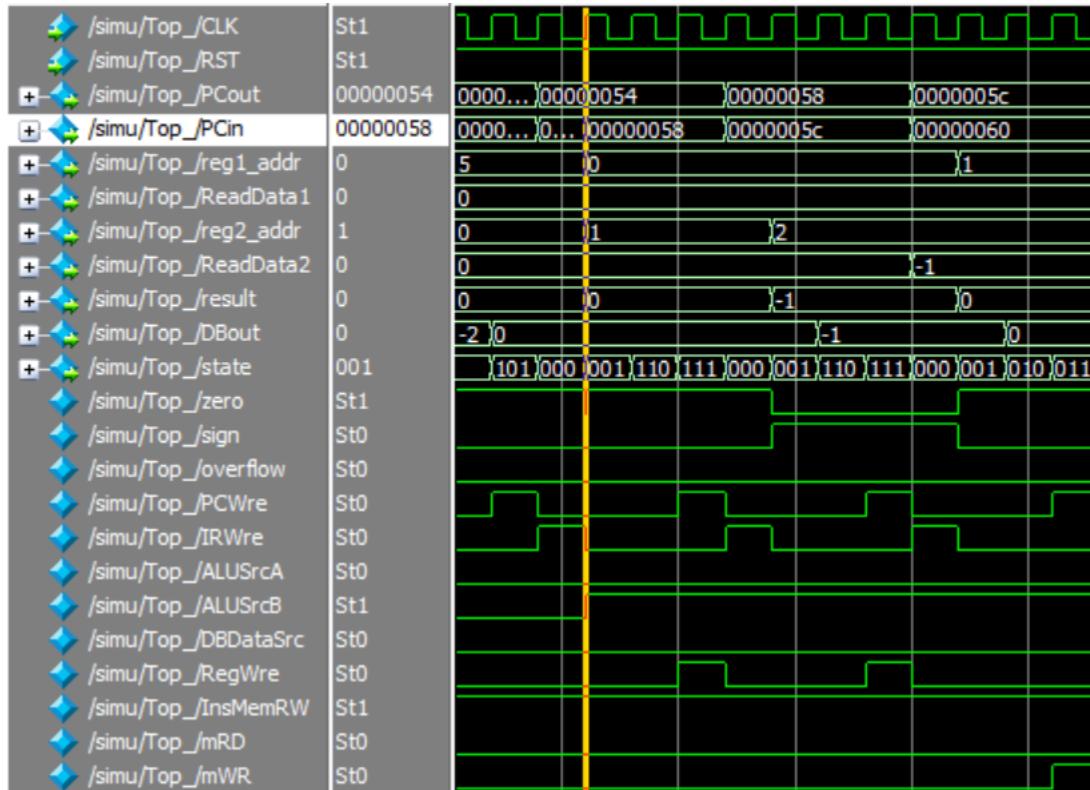
这四条指令的类型也在之前分析过，故不做赘述，当执行完毕后，所有寄存器的内容都是0。并且在 addiu \$0,\$0,-2 这条指令处，因为0号寄存器的值不能更改，所以保持0。

0x00000004C	beq \$5,\$0,1	000100	00101	00000	0000000000000001		10 a0 00 01
-------------	---------------	--------	-------	-------	------------------	--	-------------



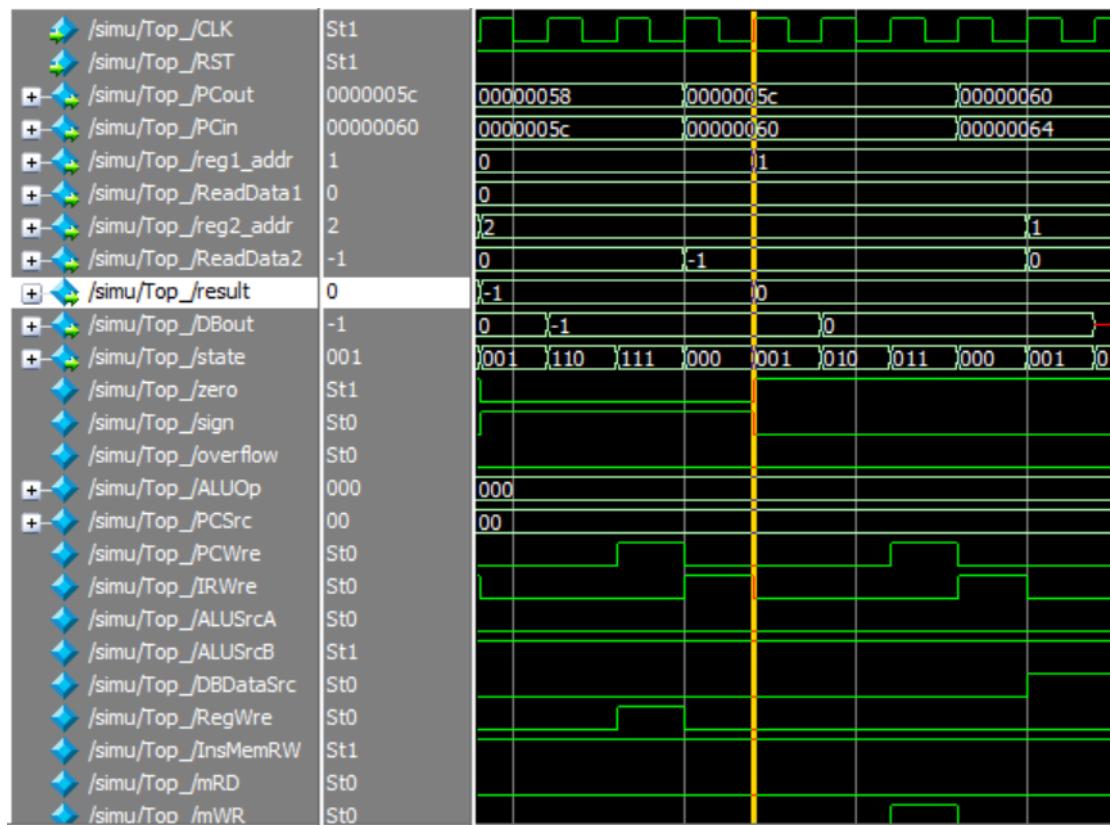
这条Beq指令目的是比较5号寄存器中的值与0号寄存器中的值，如果相等，则跳到这条指令的下一条指令的下一条，即这条指令往后的第2条指令，如果不相等则顺序执行。如图所示，可见5号寄存器中的值为0 (ReadData1)，0号寄存器中的值为0 (ReadData2)，通过ALU减法运算得到zero为1，所以PCSrc为01，所以nextPC<=PC4+signedImmediate*4，故当前PC (PCout)值为4c，下一条指令PC (PCin)为0x54，符合预期。

0x000000054	addiu \$1,\$0,0	001001	00000	00001	0000000000000000		24 01 00 00
0x000000058	addiu \$2,\$0,-1	001001	00000	00010	1111111111111111		24 02 ff ff



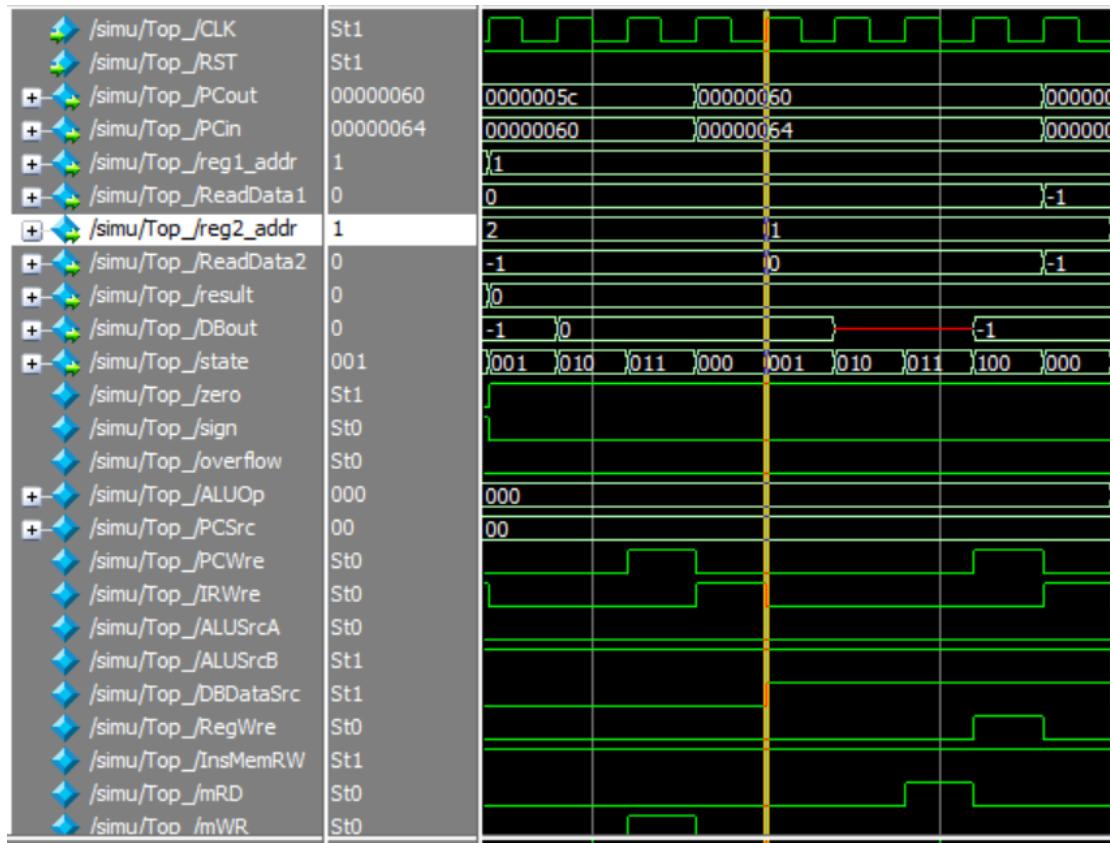
这两条指令的类型也在之前分析过，故不做赘述。当执行完毕后，2号寄存器中的值为-1，其余寄存器的内容都是0。

0x00000005C	sw \$2,0(\$1)	101011	00001	00010	0000000000000000		ac 22 00 00
--------------------	---------------	--------	-------	-------	------------------	--	-------------



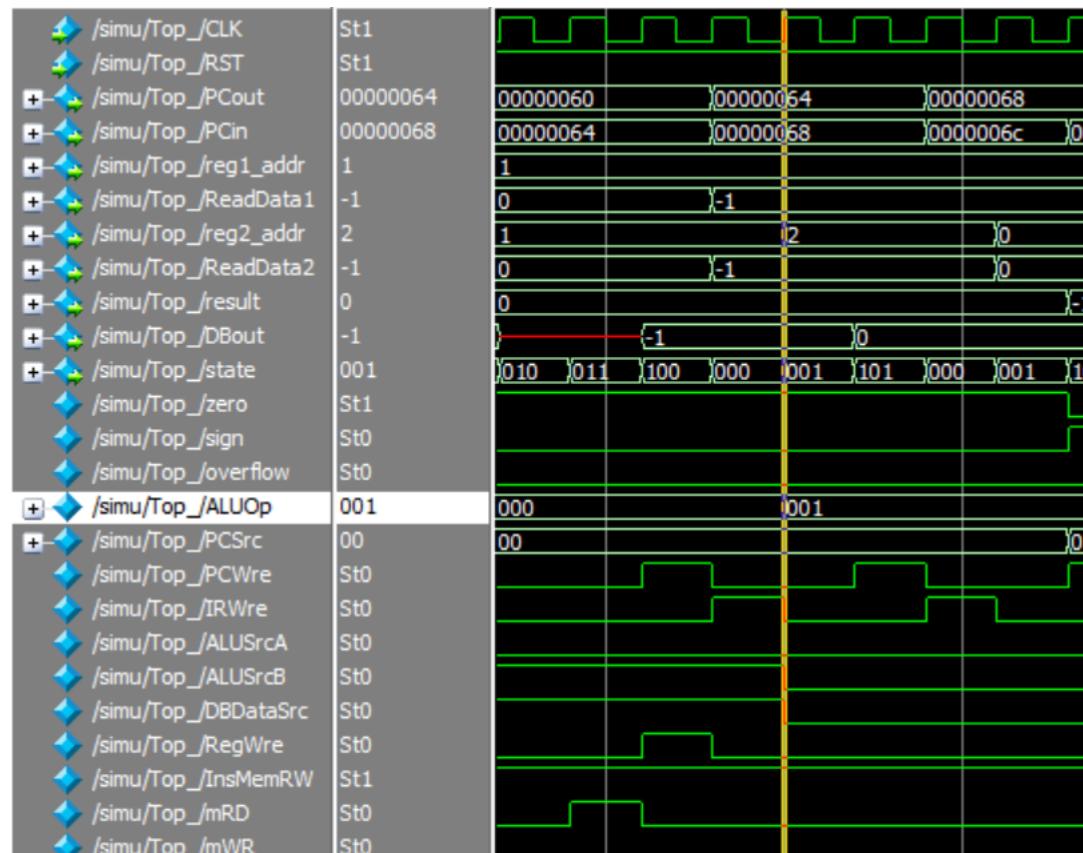
本条指令目的是将1号寄存器中的值加上偏移量0，求得一个地址，并将2号寄存器中的内容存到以这个地址开始的一个字的空间里。如上图所示，在本条指令state为000阶段即IF阶段时，取出PC在0x5c的指令，在state为001即ID阶段时，可见PCin（即为下一指令地址）为0x60，顺序执行，符合要求。可见1号寄存器中的值为0（ReadData1），2号寄存器中的值为-1（ReadData2）。立即数为0，ALU加运算（ALUOp为000）的结果（result）为0，即要将-1存到存储器中0地址的位置。在state为011（MEM）阶段时，mWR变为1，在时钟上升沿写入存储器。写入结果的正确性将在下一条指令得到验证。此外，该指令其他控制信号如图都符合要求，正确。

0x000000060	lw \$1,0(\$1)	100011	00001	00001	0000000000000000		8c 21 00 00
-------------	---------------	--------	-------	-------	------------------	--	-------------



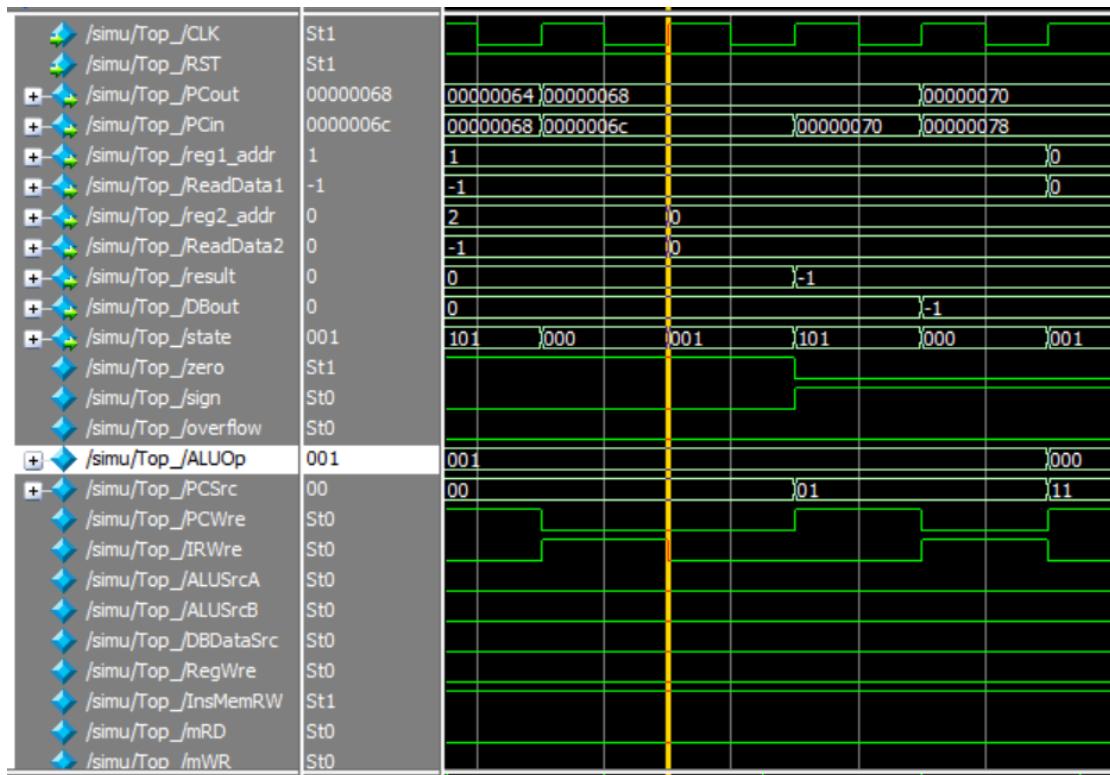
本条指令目的是将1号寄存器中的值加上偏移量0，求得一个地址，并读取此地址开始的一个字中的内容，将其存到1号寄存器中。如图所示，当前PC (PCout) 值为60，下一条指令PC (PCin) 为64，顺序执行指令，符合实验要求和预期。可见1号寄存器中的值一开始为0 (ReadData1)，ALU加运算 (ALUOp为000) 的结果 (result) 为0，即要将存储器中0地址的数取出，写入1号寄存器中，由上一条指令可知此地址中存着-1，如图所示，DBout的值在state为011 (MEM) 阶段的后期上升沿变为-1，说明取出的数正确。最后WB阶段 (state=100)，RegWre变为1，并在下一个时钟上升沿将-1写入寄存器组，这一过程可以在图中一号寄存器的值的下降沿改变看出，由0变为-1。这样同时证明了上一条指令sw的正确性。此外，该指令其他控制信号如图都符合要求，正确。这条指令执行结束后，1号寄存器值变为-1。

0x00000064	bne \$1,\$2,3	000101	00001	00010	0000000000000101		14 22 00 05
------------	---------------	--------	-------	-------	------------------	--	-------------



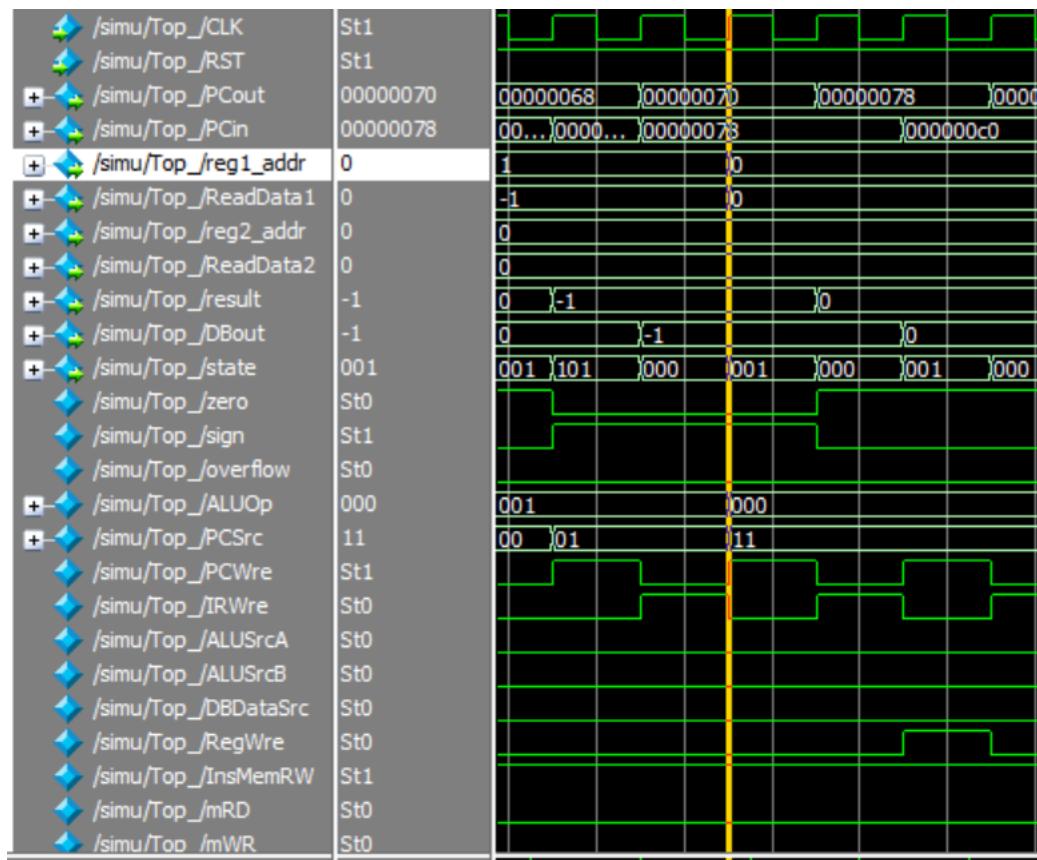
这条Bne指令目的是比较1号寄存器中的值与2号寄存器中的值，如果不相等，则跳到这条指令的下一条指令，即这条指令往后的第2条指令，如果相等则顺序执行。如图所示，在state为001即ID阶段，可见1号寄存器中的值为-1（ReadData1），2号寄存器中的值为-1（ReadData2），state为101即EXE阶段，通过ALU减法（ALUOp为001）运算得到result为0，zero为1，所以PCSrc为00，所以顺序执行下一条指令，故当前PC（PCout）值为64，下一条指令PC（PCin）为68，符合预期。其他控制信号和结果也都符合预期。可验证bne指令在本CPU下可以正确执行。由于对于PC选择的控制信号和上一条不变化，所以对PC的选择过程从波形中不能明显看出，只能看出结果正确。

0x00000068	bltz \$1,1	000001	00001	00000	0000000000000001		04 20 00 01
------------	------------	--------	-------	-------	------------------	--	-------------



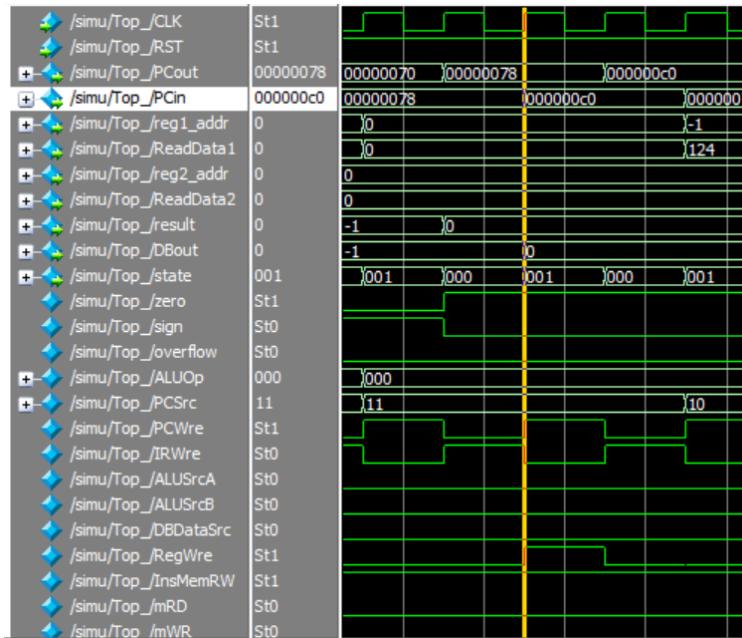
这条bltz指令的目的是，比较1号寄存器中的值与0，如果小于0，则 $pc \leftarrow pc + 4 + sign_extend(offset) \ll 2$ ，如果大于等于0，则 $pc \leftarrow pc + 4$ ，顺序执行。如图所示，当前PC (PCout) 为68。图中可见，在state为001 (ID) 阶段，1号寄存器中的值是-1。在state为101 (EXE) 阶段ALU中进行减法运算，将其与0相减，结果 (result) 为-1，sign为1，说明1号寄存器中的值小于0。此时PCSrc为01，所以下一个PC (PCin) 在sEXE阶段变为 $68+4+4*1=70$ (16进制)，符合预期。其他控制信号和结果也都符合预期。可验证bltz指令在本CPU下可以正确执行。

0x000000070	j 0x1e(0x78)	000010	00000	00000	00000000000011110		08 00 00 1e
-------------	--------------	--------	-------	-------	-------------------	--	-------------



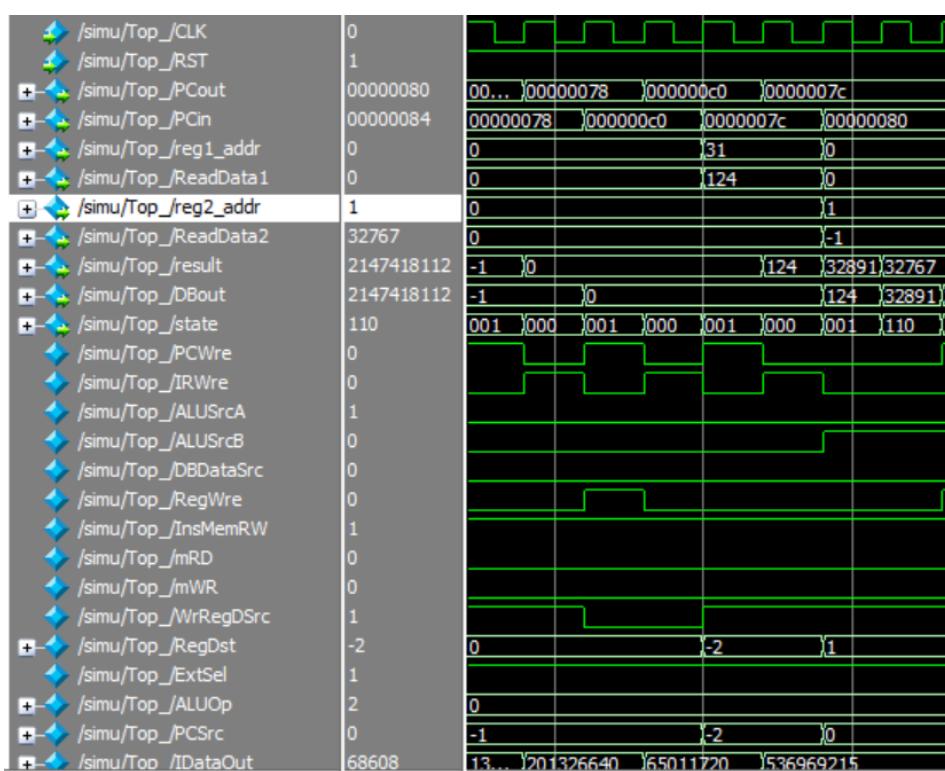
本条J指令的目的是让它跳转到0x78的位置，如图所示，本条指令执行了IF和ID两个阶段，当前PC (PCout) 值为70，下一条指令PC (PCin) 为78，PCSrc在ID阶段变为11。符合实验要求。

0x000000078	jal 0x30(0xC0)	000011	00000	00000	000000000000110000		0c 00 00 30
-------------	----------------	--------	-------	-------	--------------------	--	-------------

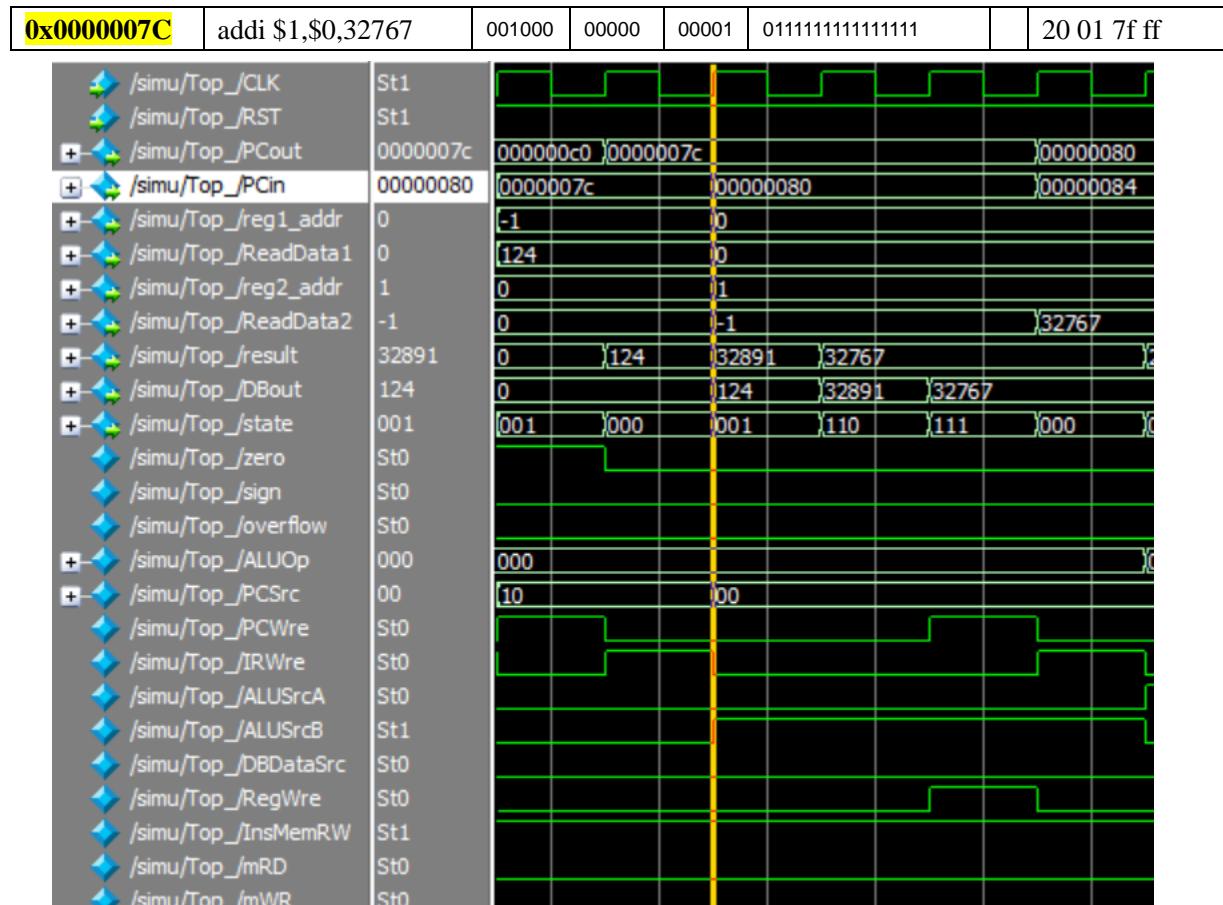


这条指令调用子程序， $PC \leftarrow \{PC[31:28], addr, 2'b0\}$; $GPR[$31] \leftarrow pc+4$ ，返回地址设置；子程序返回，需用指令 $jr \$31$ 。跳转地址的形成同 $j\ addr$ 指令。如图所示，本条指令执行了IF和ID两个阶段，当前PC (PCout) 值为78，下一条指令PC (PCin) 在ID阶段变为c0，PCSrc为11。符合实验要求。

0x000000C0	jr \$31	000000	11111	00000	0000000000001000		03 e0 00 08
------------	---------	--------	-------	-------	------------------	--	-------------

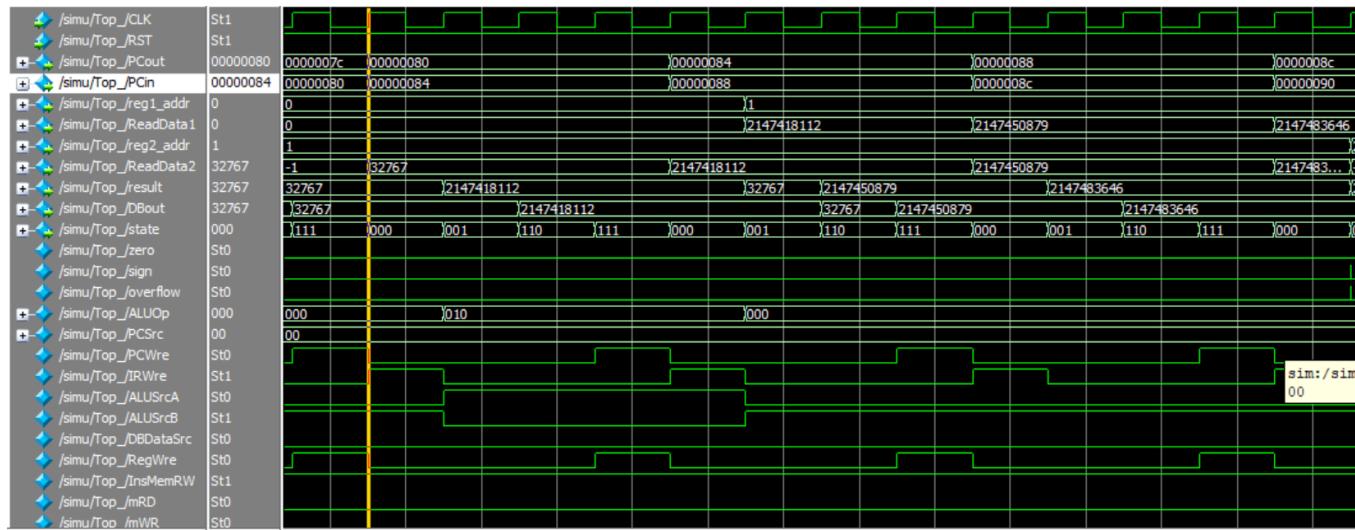


这条指令的目的是跳回31号寄存器中存的地址，如图，当前指令地址（PCout）为0xc0，在ID阶段得到31号寄存器中的值为124，转化成16进制即为0x7c，正是之前jal时存入的地址。可见该指令执行正确。



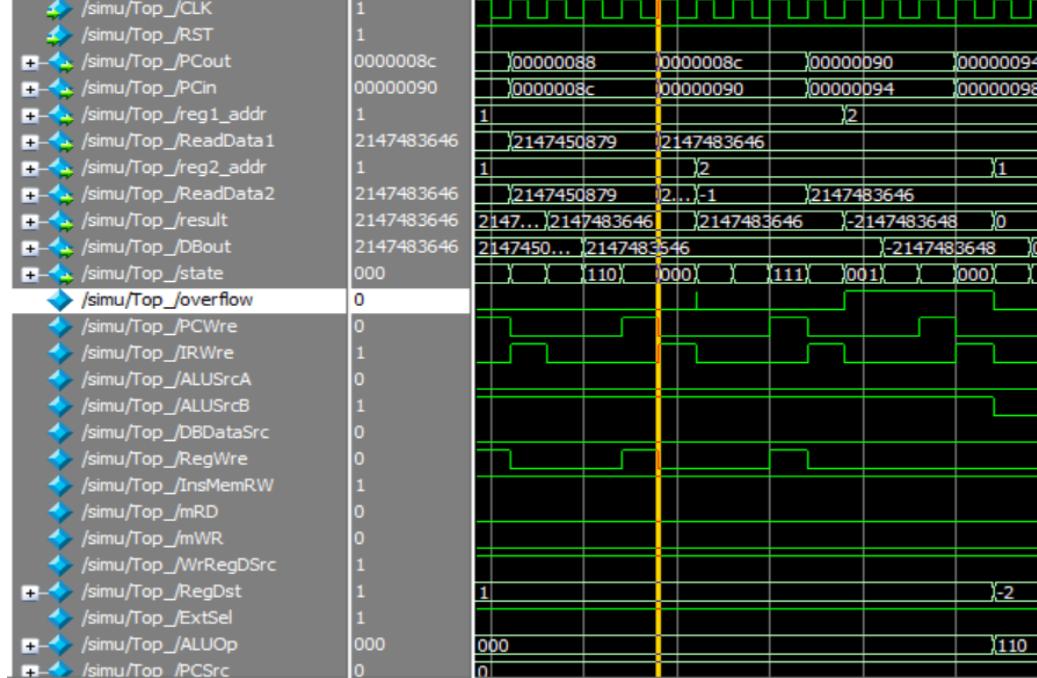
这条指令是要把\$0中的值加上立即数32767的结果写到1号寄存器中，并且需要判断溢出。如上图所示，在执行本条指令时，在state为000阶段即IF阶段时，取出PC在0x7c的指令，在state为001即ID阶段时，得到PCin（即为下一指令地址）为0x80，顺序执行，符合要求，并且可以看出在ID阶段时，0号寄存器中的值为0。当state为110即EXEa阶段时，ALU的result为，即为算出 $0+32767$ 的结果，正确，并且没有溢出，overflow信号为0。当state为111（WBa）时，RegWre变为1，并在下一个时钟上升沿将结果写入寄存器组，这一过程可以在图中观察出来。执行完这一条后，寄存器1中的值变为32767。经检查，其他控制信号也符合预期。

0x00000080	sll \$1,\$1,16	000000	00000	00001	000011000000000000		00 01 0c 00
0x00000084	addi \$1,\$1,32767	001000	00001	00001	0111111111111111		20 21 7f ff
0x00000088	addi \$1,\$1,32767	001000	00001	00001	0111111111111111		20 21 7f ff

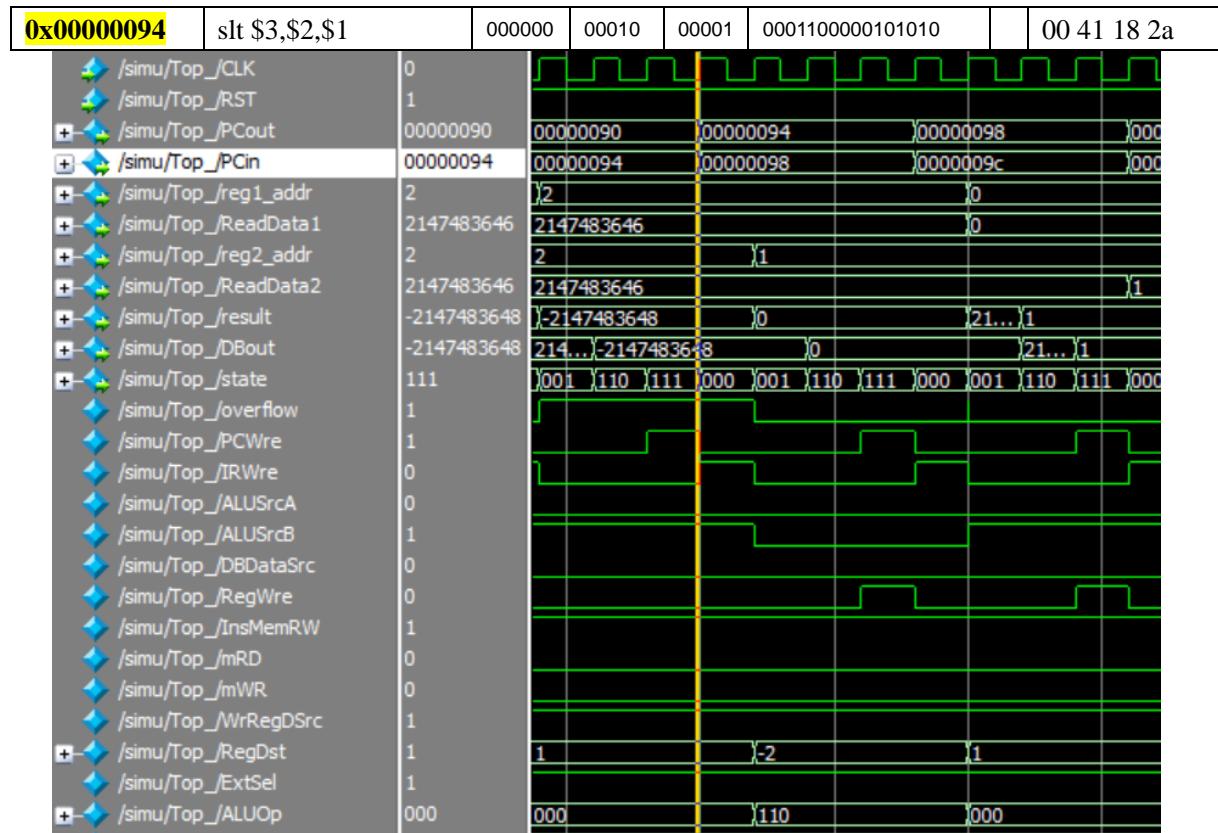


这三条指令先是将32767左移16位，变成“0111 1111 1111 1110 0000 0000 0000 0000”，再加32767，成为“0111 1111 1111 1110 1111 1111 1111 1111”，再加32767，变成“0111 1111 1111 1111 1111 1111 1111 1110”。这段过程均没有溢出，故结果写入了寄存器组。

0x0000008C	addi \$2,\$1,0	001000	00001	00010	000000000000000000		20 22 00 00
0x00000090	addi \$2,\$2,2	001000	00010	00010	000000000000000010		20 42 00 02

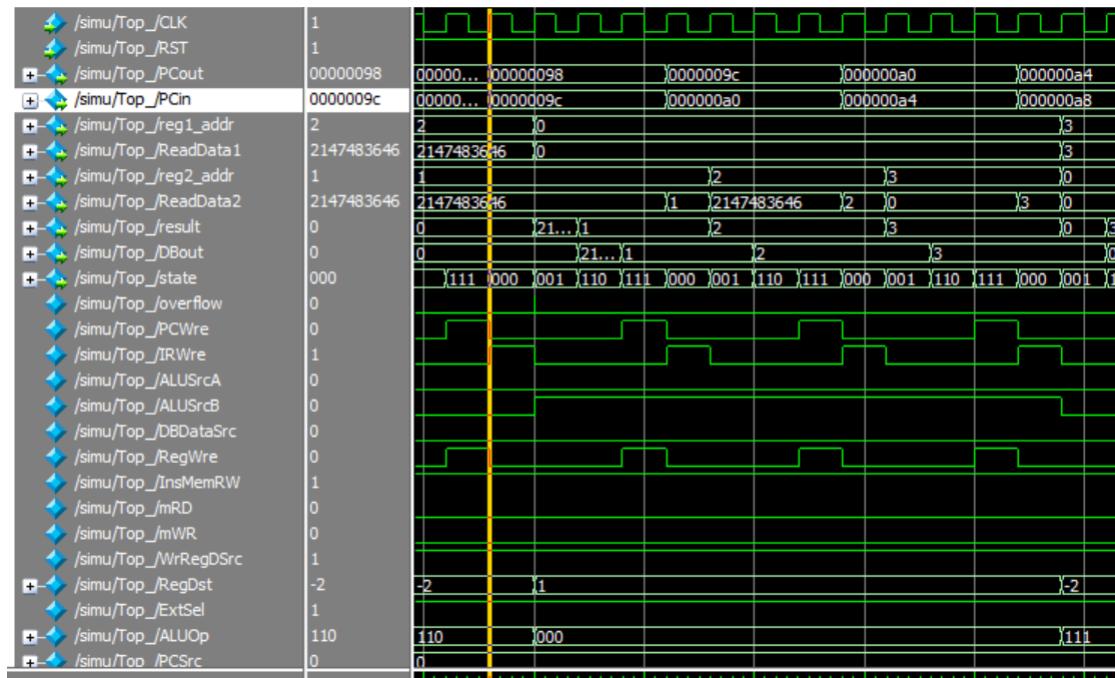


这两条指令也需要判断溢出，先是将“0111 1111 1111 1111 1111 1111 1111 1111 1110”加0，没有溢出，结果写到2号寄存器中，再是将2号寄存器中的“0111 1111 1111 1111 1111 1111 1111 1111 1110”加立即数2，结果溢出，故overflow变为1，并且RegWre为0，不会写入寄存器组，从波形可以看出，2号寄存器的值在本条指令没有变化。



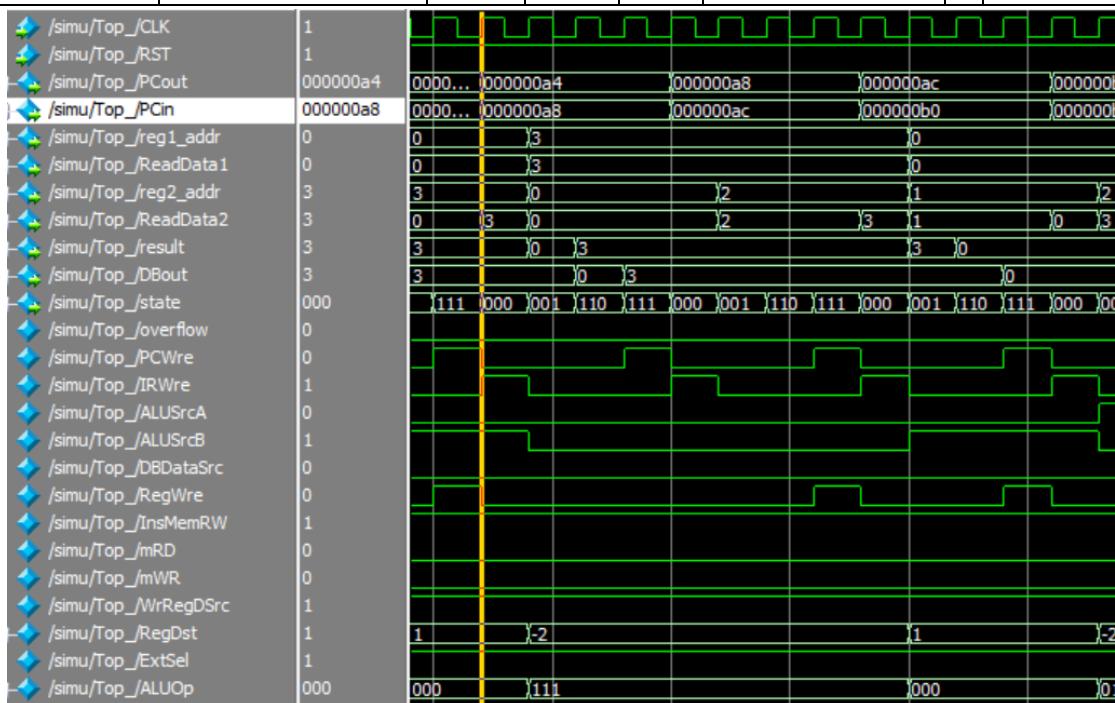
小于则置位指令，由于2号寄存器中的值与1号寄存器中相等，所以ALU的result为0，将0写入3号寄存器。

0x00000098	addi \$1,\$0,1	001000	00000	00001	0000000000000001		20 01 00 01
0x0000009C	addi \$2,\$0,2	001000	00000	00010	00000000000000010		20 02 00 02
0x000000A0	addi \$3,\$0,3	001000	00000	00011	00000000000000011		20 03 00 03



这三条指令的类型在上文已经分析过，故不再赘述。这三条指令执行后，1号、2号、3号寄存器中的值分别为1、2、3。

0x000000A4	movn \$1,\$3,\$0	000000	00011	00000	0000100000001011		00 60 08 0b
0x000000A8	movn \$2,\$3,\$2	000000	00011	00010	0001000000001011		00 62 10 0b
0x000000AC	addiu \$1,\$0,0	001001	00000	00001	0000000000000000		24 01 00 00



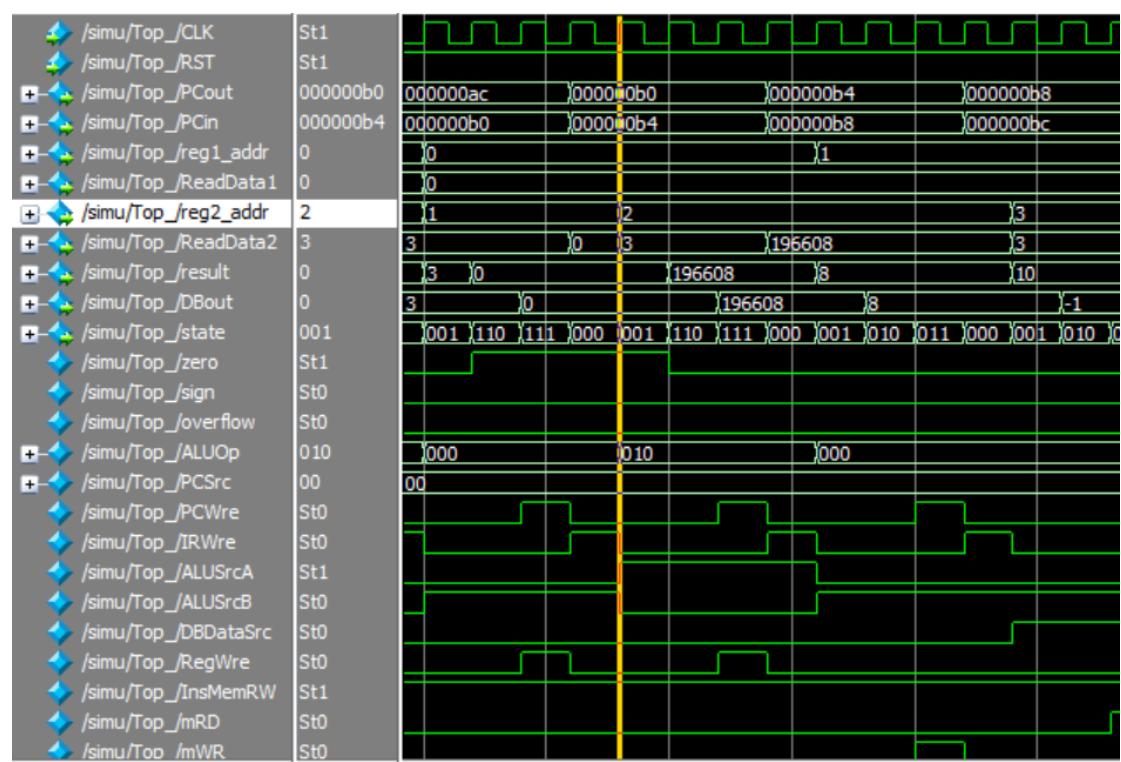
a4：因为0号寄存器中的值等于0，所以不写寄存器，在本指令的WB阶段RegWre为0，不写。

a8：因为2号寄存器中的值不为0，所以要将3号寄存器中的值写入2号寄存器，在本指令的WB阶段RegWre为1，将3写入2号寄存器。

ac：1号寄存器中的值变为0。

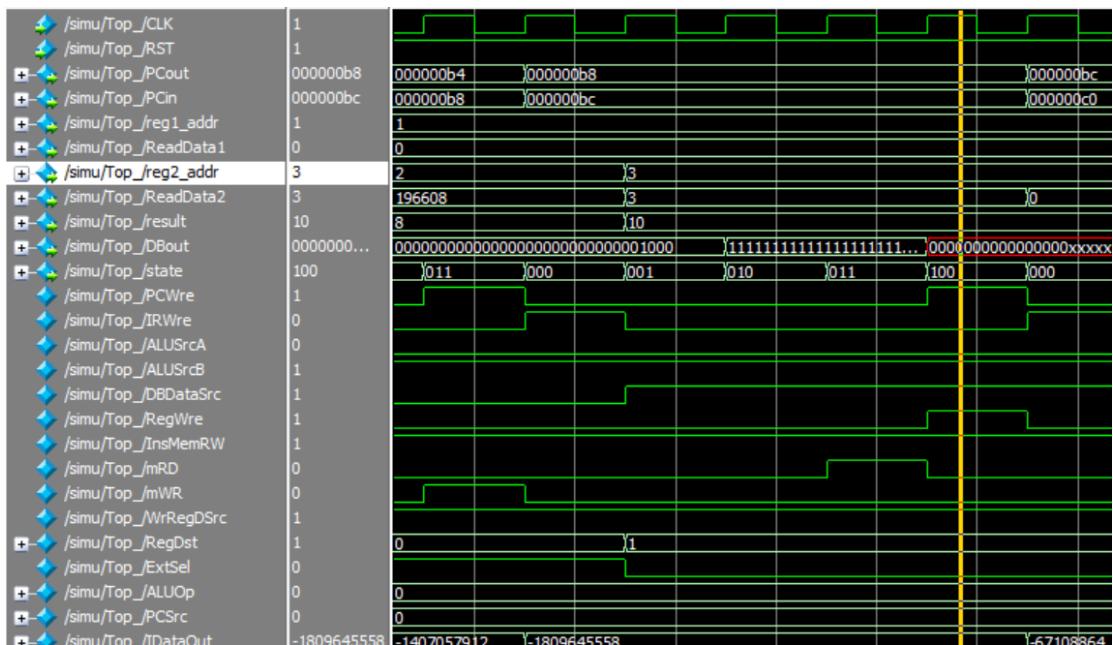
到目前为止，2号寄存器和3号寄存器值都为3，31号寄存器中存了跳转地址，其余寄存器都为0。

0x000000B0	sll \$2,\$2,16	000000	00000	00010	0001010000000000		00 02 14 00
0x000000B4	sw \$2,8(\$1)	101011	00001	00010	0000000000001000		ac 22 00 08

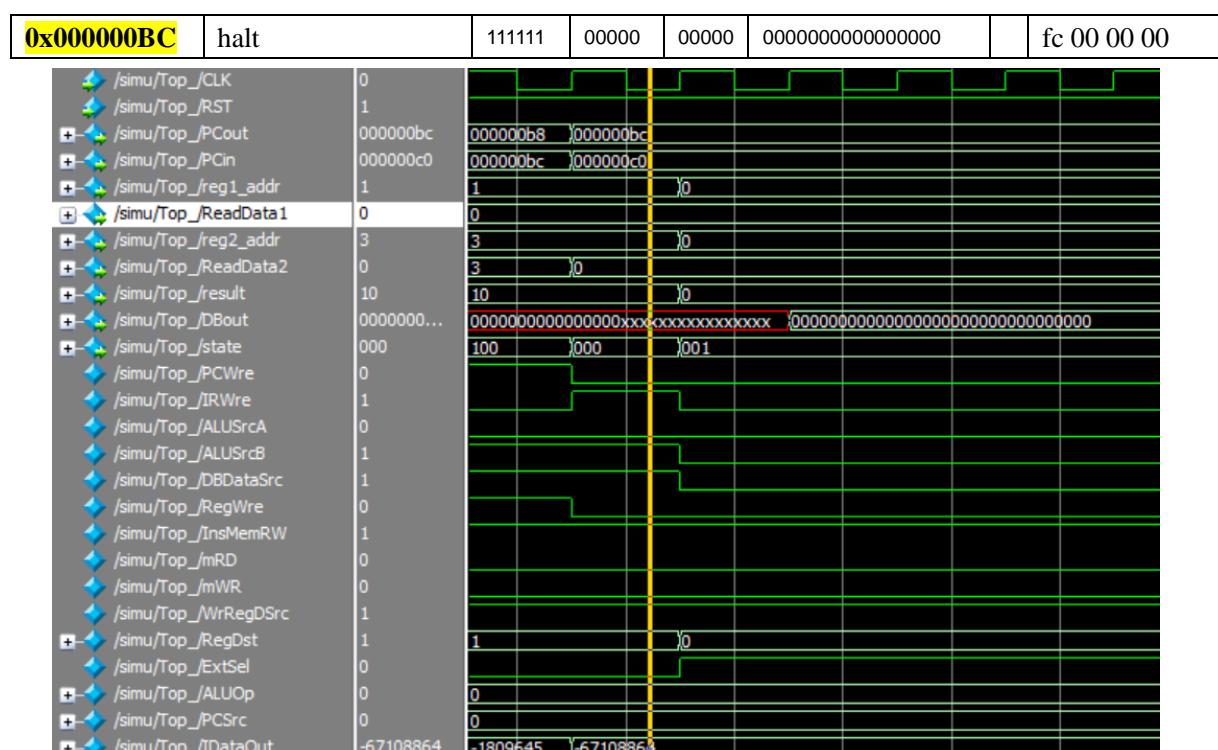


这两条指令结束之后，将196608存到1号寄存器中的值加8的地址中。因为前文有对这两种指令的详述，故分析略。

0x000000B8	lhu \$3,10(\$1)	100101	00001	00011	0000000000001010		94 23 00 0a
------------	-----------------	--------	-------	-------	------------------	--	-------------



Lhu是将1号寄存器中的值加10的地址开始的半个字取出来，经无符号扩展，存入3号寄存器中去，其他具体过程与前文lw相似，故不做赘述，最终3号寄存器中的值变为0。



停机指令使得PC不再变化，其他控制信号等也停止变化。

所有指令结束后的寄存器组:

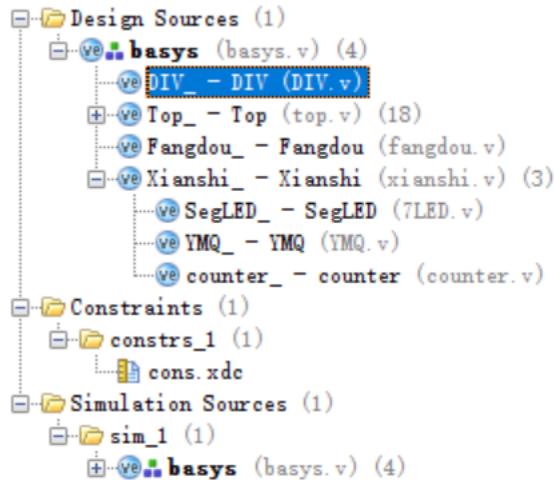
Memory Data - /simu/Top_RegisterFile_register - Default	
	Goto: []
00000000	00000000000000000000000000000000 00000000000000000000000000000000
00000002	00000000000000000000000000000000 00000000000000000000000000000000
00000004	00000000000000000000000000000000 00000000000000000000000000000000
00000006	00000000000000000000000000000000 00000000000000000000000000000000
00000008	00000000000000000000000000000000 00000000000000000000000000000000
0000000a	00000000000000000000000000000000 00000000000000000000000000000000
0000000c	00000000000000000000000000000000 00000000000000000000000000000000
0000000e	00000000000000000000000000000000 00000000000000000000000000000000
00000010	00000000000000000000000000000000 00000000000000000000000000000000
00000012	00000000000000000000000000000000 00000000000000000000000000000000
00000014	00000000000000000000000000000000 00000000000000000000000000000000
00000016	00000000000000000000000000000000 00000000000000000000000000000000
00000018	00000000000000000000000000000000 00000000000000000000000000000000
0000001a	00000000000000000000000000000000 00000000000000000000000000000000
0000001c	00000000000000000000000000000000 00000000000000000000000000000000
0000001e	00000000000000000000000000000000 000000000000000000000000000000001111100

所有指令结束后的Data Memory:

Memory Data - /simu/Top_DataMem_data - Default	
	Goto: []
00000000	11111111 11111111 11111111 11111111
00000004	xxxxxxxx xxxx xxxx xxxx xxxx
00000008	00000000 00000011 00000000 00000000
0000000c	xxxxxxxx xxxx xxxx xxxx xxxx
00000010	xxxxxxxx xxxx xxxx xxxx xxxx
00000014	xxxxxxxx xxxx xxxx xxxx xxxx
00000018	xxxxxxxx xxxx xxxx xxxx xxxx
0000001c	xxxxxxxx xxxx xxxx xxxx xxxx
00000020	xxxxxxxx xxxx xxxx xxxx xxxx
00000024	xxxxxxxx xxxx xxxx xxxx xxxx
00000028	xxxxxxxx xxxx xxxx xxxx xxxx
0000002c	xxxxxxxx xxxx xxxx xxxx xxxx
00000030	xxxxxxxx xxxx xxxx xxxx xxxx
00000034	xxxxxxxx xxxx xxxx xxxx xxxx
00000038	xxxxxxxx xxxx xxxx xxxx xxxx
0000003c	xxxxxxxx xxxx xxxx xxxx xxxx

烧板过程：

烧板的模块结构如图：



分频器，将自带的时钟频率分成适合数码管循环显示的时钟频率：

```

1 `timescale 1ns / 1ps
2 module DIV(
3     input clk,
4     output reg DIVout
5 );
6     initial begin
7         DIVout<=0;
8     end
9
10    integer i=0;
11    always @(posedge clk) begin
12        if(i>=49999) begin
13            DIVout<=~DIVout;
14            i<=0;
15        end
16        else i<=i+1;
17    end
18 endmodule

```

防抖模块，防止一次按键产生多个时钟上升沿：

```

1 `timescale 1ns / 1ps
2 module Fangdou(
3     input CLK,
4     input bottom,
5     output myCLK
6 );
7     reg temp3, temp2, temp1;
8     always @(posedge CLK) begin
9         temp1<=bottom;
10        temp2<=temp1;
11        temp3<=temp2;
12    end
13    assign myCLK=temp3 && temp2 && temp1;
14 endmodule

```

扫描输出显示模块的连接部分：

```

23     assign AN = ~ANf;
24     SegLED SegLED_(num, DispCode);
25     YMQ YMQ_(cnt[0], cnt[1], ANf);
26     counter counter_(CLK, RST, cnt);

```

数码管，主要就是将16进制数与7段数码管的显示对应起来：

```

6 always @( num ) begin
7     case (num)
8         4'b0000 : dispcode = 8'b1100_0000; //0
9         4'b0001 : dispcode = 8'b1111_1001; //1
10        4'b0010 : dispcode = 8'b1010_0100; //2
11        4'b0011 : dispcode = 8'b1011_0000; //3
12        4'b0100 : dispcode = 8'b1001_1001; //4
13        4'b0101 : dispcode = 8'b1001_0010; //5
14        4'b0110 : dispcode = 8'b1000_0010; //6
15        4'b0111 : dispcode = 8'b1101_1000; //7
16        4'b1000 : dispcode = 8'b1000_0000; //8
17        4'b1001 : dispcode = 8'b1001_0000; //9
18        4'b1010 : dispcode = 8'b1000_1000; //A
19        4'b1011 : dispcode = 8'b1000_0011; //B
20        4'b1100 : dispcode = 8'b1100_0110; //C
21        4'b1101 : dispcode = 8'b1010_0001; //D
22        4'b1110 : dispcode = 8'b1000_0110; //E
23        4'b1111 : dispcode = 8'b1000_1110; //F
24        default : dispcode = 8'b0000_0000;
25    endcase
26 end

```

译码器模块:

```

2 module YMQ(
3     input A,
4     input B,
5     output reg [3:0] Y_Out
6 );
7     always@(A or B) begin
8         case({B,A})
9             2'b000: Y_Out = 4'b0001;
10            2'b001: Y_Out = 4'b0010;
11            2'b010: Y_Out = 4'b0100;
12            2'b011: Y_Out = 4'b1000;
13            default: Y_Out = 4'b0000;
14        endcase
15    end
16 endmodule

```

计数器模块:

```

2 module counter(
3     input CLK,
4     input RST,
5     output reg[1:0] num
6 );
7     always@(posedge CLK or negedge RST) begin
8         if(num==2'b11) num<=2'b00;
9         else num<=num+2'b01;
10    end
11 endmodule

```

实验板显示:

- 1、当前指令地址PC:下条指令地址PC?
- 2、RS寄存器地址:RS寄存器数据?
- 3、RT寄存器地址:RT寄存器数据?
- 4、ALU结果输出 :DB总线 数据?

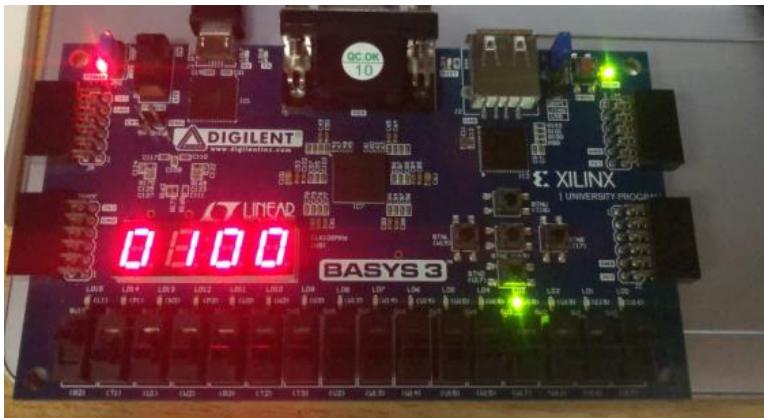
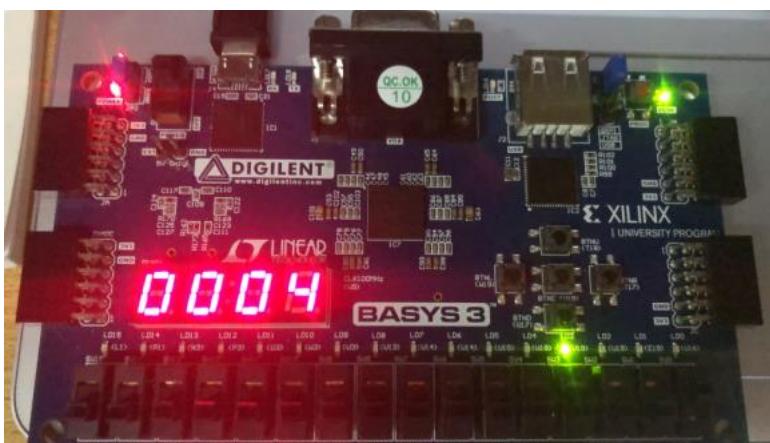
后五个流水灯从左到右分别代表五个阶段: IF, ID, EXE, MEM, WB

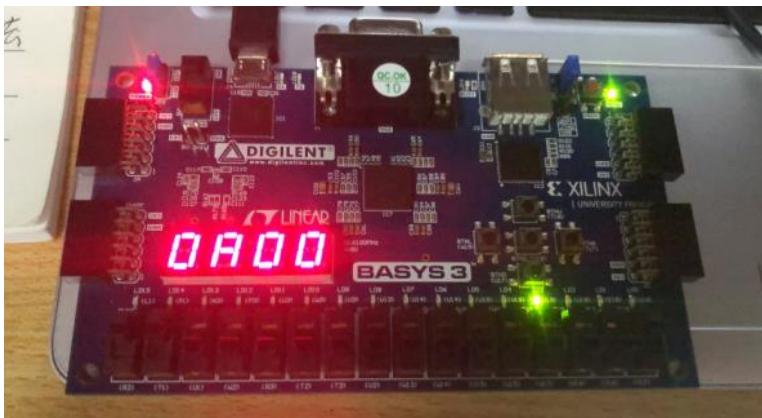
前两个拨码开关用来确定显示内容。

第一条指令IF阶段的PC显示:

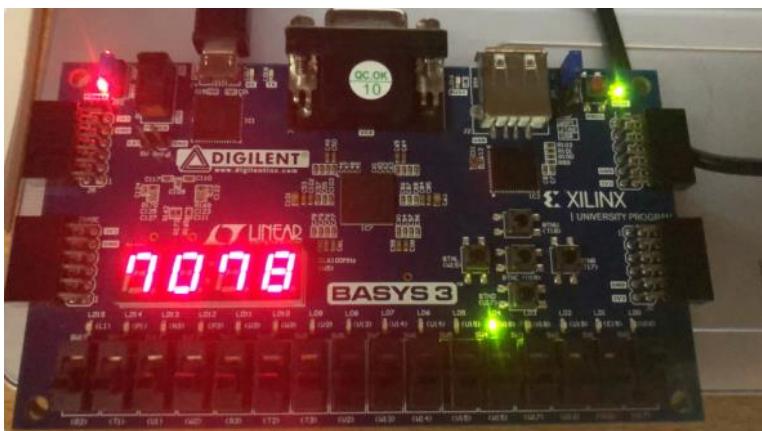


第一条指令ID阶段的四种显示：





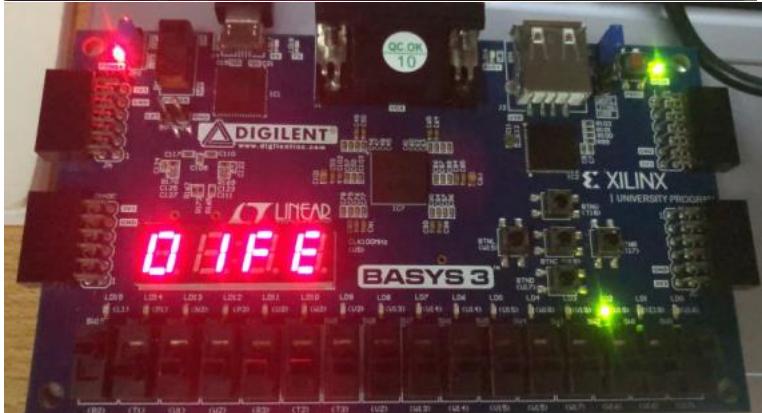
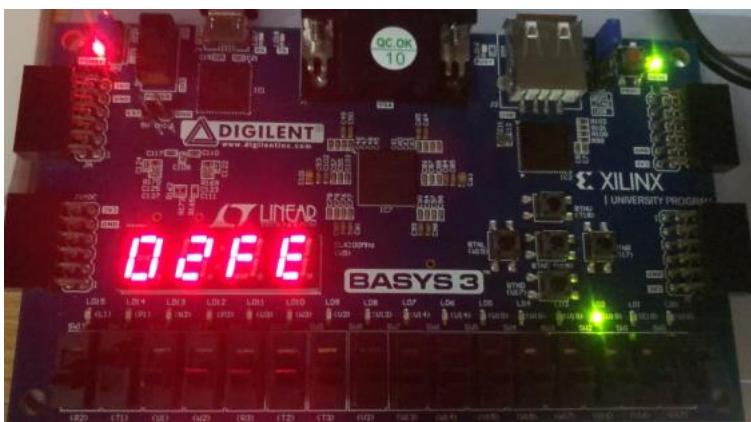
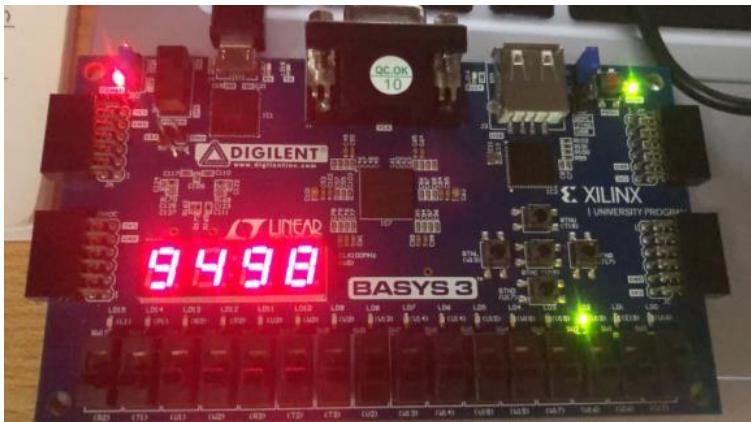
在PC为70时，当前和下一条PC的显示：



在PC为78时，当前和下一条PC的显示：



在PC为94时，EXE阶段的4种显示：



六. 实验心得

刚仿真出波形的时候，发现ALU的result一直都是0，我很奇怪。后来经过分析波形，发现ALU中的A和B总是0，没有变过。最后找了半天bug，终于发现是在顶层模块中，ALU的输入连错了。接着又发现寄存器组的数据都没有变化过，还发现WrRegDSrc一直都是高阻态，原来是写控制信号表的时候竟然忘记写这个控制信号了，而且同时发现RegWre表达式也是错的。再后来发现在lw指令之后总会提前停止，经过查看波形，发现是当state为WBm时，无法产生正确的nextState，经检查代码，发现是MEM的机器码011写成了和WBm一样的100导致case时出错，真是太粗心了。改正这个错误后，新的波形又只能停在PC为0x80的地方，这个问题我找了很久，最终才反应过来，是因为我没有给InsMEM足够的空间，导致之后的指令没有都装进来。后来又又又发现jal不能正确跳转，原因竟然又是PCchoose模块里将10写成11了。最后最令我抓狂的一个bug是我将6' b100101写成了100101，忘记加6' b了，导致lhu总执行不对，这一小小的错误让我找了2个小时。我总是在这种地方栽跟头浪费时间，可见粗心害人，以后写代码的时候应该步步回头检查，防止出现这么多bug。

最后烧板子的时候，我遇到了关于CLK的报错，到处查资料，最后才想起来助教上课讲过要在约束文件里加一句话，这才解决问题。

通过这次多周期CPU实验，我真的收获了很多。这是一个磨练人的作业，因为它要注意太多太多细节了，经常一点小错误就会使结果变得乱七八糟。可谓是牵一发而动全身。我一直都不是一个很细心的人，但我这次用了一天时间打代码，却用了整整一周时间来烧板子和debug，感觉我的心态进步了许多，不会再因为一点小问题而急躁生气。于此同时，通过做实验，自己设计多周期CPU，使我对多周期的理论知识也有了更深刻的认识。