



《计算机组成原理实验》

实验报告

(实验三)

学院名称 : 数据科学与计算机学院

专业(班级) : 17 软件工程 2 班

学生姓名 : 张伟焜

学号 : 17343155

时间 : 2018 年 12 月 22 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) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
- (4) 掌握多周期 CPU 的测试方法；
- (5) 掌握多周期 CPU 的实现方法。

二、 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：

==>算术运算指令

- (1) add rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow -rs + rt$ 。

- (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) addiu rt, rs, immediate

000010	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

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

==>逻辑运算指令

- (4) and rd , rs , rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: $rd \leftarrow rs \& rt$; 逻辑与运算。

- (5) andi rt , rs ,immediate

010001	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs \& (\text{zero-extend})\text{immediate}$; immediate 做“0”扩展再参加“与”运算。

- (6) ori rt , rs ,immediate

010010	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs | (\text{zero-extend})\text{immediate}$; immediate 做“0”扩展再参加“或”运算。

- (7) xori rt , rs , immediate

010011	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $rt \leftarrow rs \oplus (\text{zero-extend})\text{immediate}$; immediate 做“0”扩展再参加“异或”运算。

==>移位指令

(8) sll rd, rt,sa

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

功能: $rd \leftarrow rt \ll (zero\text{-extend})sa$, 左移 sa 位, (zero-extend)sa。**==>比较指令**

(9) slti rt, rs,immediate 带符号

100110	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: if ($rs < (sign\text{-extend})immediate$) $rt = 1$ else $rt=0$, 具体请看表 2 ALU 运算功能表, 带符号。

(10) slt rd, rs, rt 带符号

100111	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。**==>存储器读写指令**

(11) sw rt, immediate(rs)

110000	rs(5 位)	rt(5 位)	immediate(16 位)	
--------	---------	---------	-----------------	--

功能: $memory[rs + (sign\text{-extend})immediate] \leftarrow rt$ 。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

(12) lw rt, immediate(rs)

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

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

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

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

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

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

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

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

(15) bltz rs,immediate

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

功能: if($rs < \$0$) $pc \leftarrow pc + 4 + ((sign\text{-extend})immediate \ll 2)$ else $pc \leftarrow pc + 4$ 。**==>跳转指令**

(16) j addr

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

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

(17) 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

(18) 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

(19) halt (停机指令)

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

不改变 pc 的值，pc 保持不变。

三、 实验原理

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

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

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

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

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

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

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

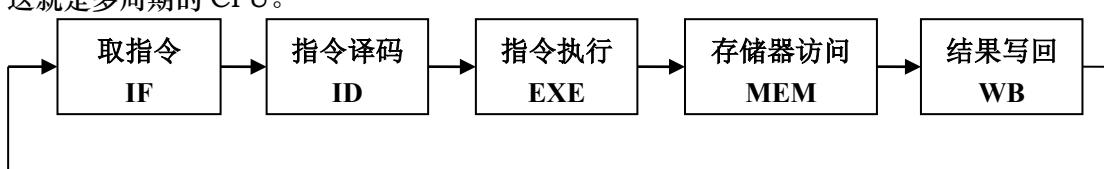


图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

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

I 类型：

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

J 类型：

31	26 25	0
op	address	
6 位	26 位	

其中，

op: 为操作码；

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

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

rd: 为目的操作数寄存器，寄存器地址（同上）；

sa: 为位移量 (shift amt)，移位指令用于指定移多少位；

funct: 为功能码，在寄存器类型指令中 (R 类型) 用来指定指令的功能；

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

address: 为地址。

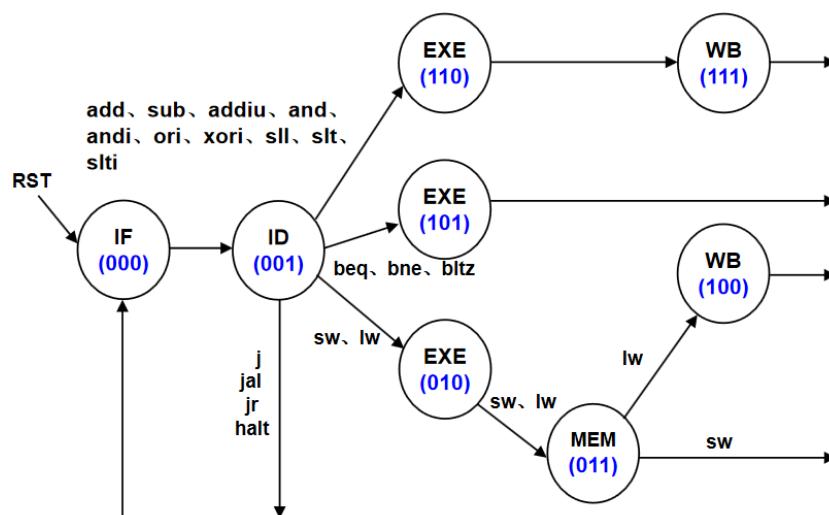


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 sIF 状态转移到 sID 就是无条件的；有些是有条件的，例如 sEXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作

码决定。每个状态代表一个时钟周期。

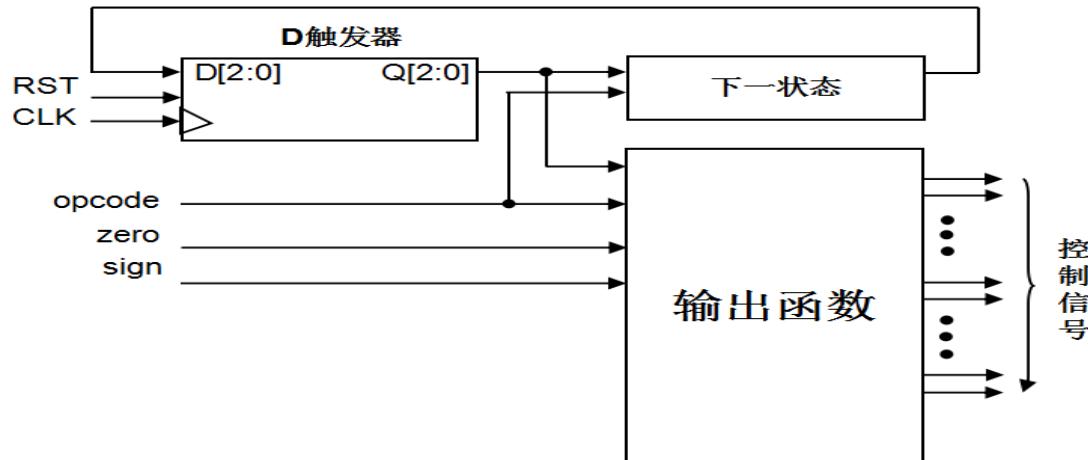


图 3 多周期 CPU 控制部件的原理结构图

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

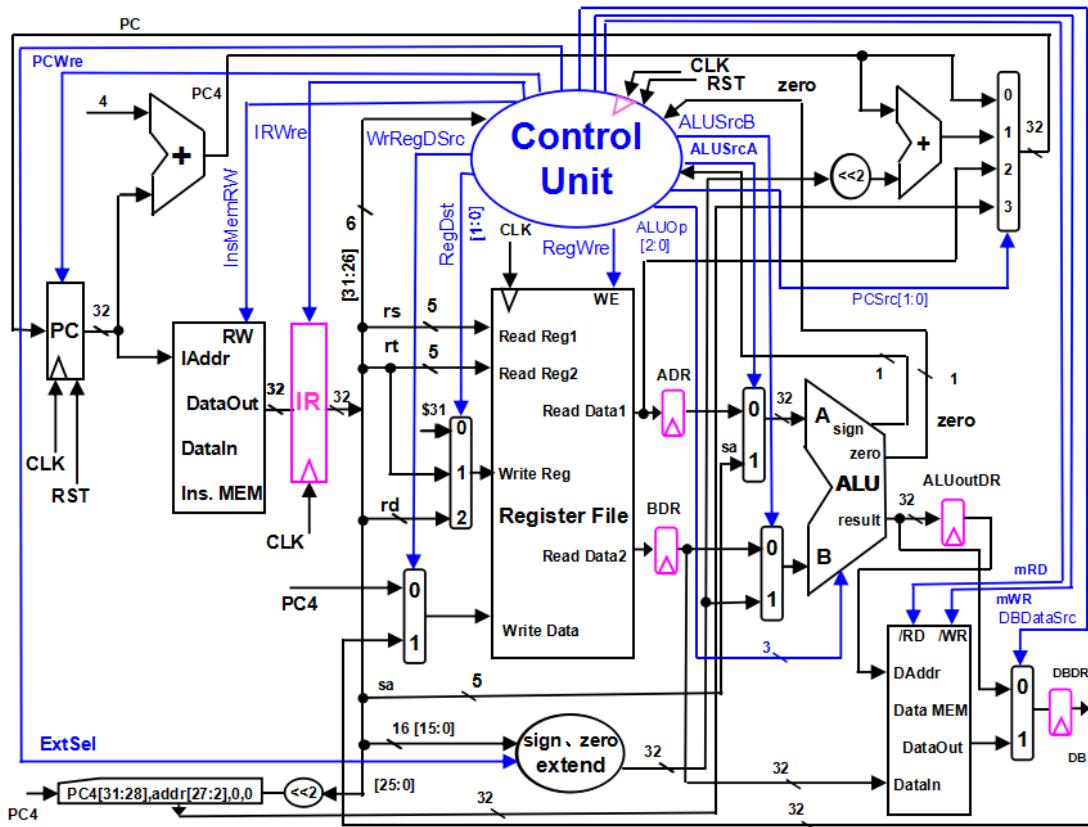


图 4 多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄

存器地址（编号），读操作时不需要时钟信号，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、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
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；
PCSsrc[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

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	$Y = A \oplus B$	异或

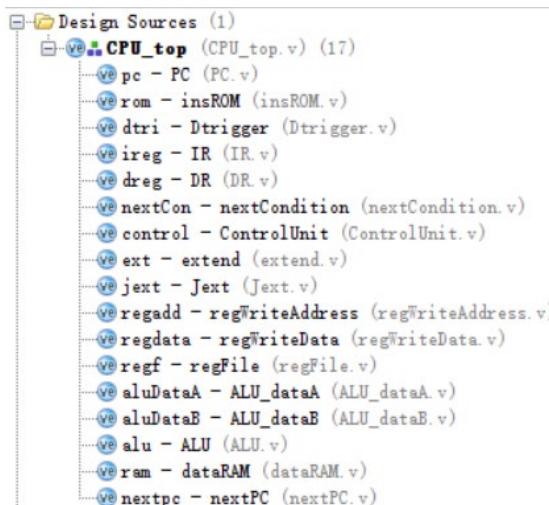
四、实验设备

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

五、实验过程与结果

1. 多周期 CPU 设计的思想、方法：

同单周期 CPU 设计类似，本次实验采用模块化的设计方法，将多周期 CPU 划分为 17 个底层模块和 1 个顶层模块。多周期 CPU 主要分为如下器件（模块），PC，指令存储器，控制单元，ALU，寄存器组，存储器，立即数扩展单元，除此之外，还有 D 触发器模块（改变状态），状态产生模块，指令寄存器模块（IR）和数据寄存器模块（DR）。 （具体模块如下）



根据数据通路图，仅有控制单元，PC，指令寄存器，数据寄存器，寄存器组需要时钟触发，而其它器件，相应的功能依靠组合逻辑电路实现需要电平触发，没有时钟输入。相比单周期CPU，多周期CPU的不同之处在于存储器改为电平触发，并且引入了指令寄存器IR和数据寄存器DR来存储指令和数据，从而使指令保持稳定并划分数据延迟。

在多周期CPU中，每条指令的执行划分为多个阶段，不同指令需要的阶段数不同，每条指令最多有五个阶段，取址，译码，执行，访存，写回五个阶段。不同阶段对应不同的状态码。

多周期CPU的核心控制模块包括三个模块：状态生成器(nextCondition)，D触发器(Dtrigger)和控制信号生成器(ControlUnit)。状态生成器（电平触发）根据操作码和当前状态得出下一状态，D触发器（时钟上升沿触发）存储并改变当前状态，控制信号生成器（电平触发）根据当前状态，操作码和标志信号确定控制信号。

PC指明了当前执行的指令地址，在时钟上升沿到来并且可写时更新指令地址。指令存储器按地址读取指令后指令存储在指令寄存器内（译码阶段），通过时钟触发（上升沿触发）

将相应的操作数送至各个模块。控制单元通过时钟触发出控制信号。其他模块接受控制信号后执行指令，将数据暂时存储在数据寄存器中，有需要的话再写入存储器或寄存器组。寄存器组写操作需要时钟下降沿触发，而存储器访问只需要电平触发。

其余模块大致同单周期CPU模块相同。(具体代码将在设计流程中展示)

在顶层模块中，将各个主要器件和数据模块组合，按数据通路图所示正确传递参数。

2.多周期CPU设计流程：

根据实验内容中的指令说明及控制信号的作用，写出指令与控制信号之间的关系。

状态	指令名称	PCWr e	ALUS rcA	ALUS rcB	DBDa taSrc	Reg Wre	WrRe gDSr c	mRD	mWR	IRWr e	ExSel	PCSr c	RegD st	ALU Op
sIF(0 00)	除 halt	1	x	x	x	x	x	x	x	1	x	xx	xx	xxx
	halt	0	x	x	x	x	x	x	x	1	x	xx	xx	xxx
sID(0 01)	j	0	x	x	x	0	x	x	x	0	x	11	xx	xxx
	jal	0	x	x	x	1	0	x	x	0	x	11	00	xxx
	jr	0	x	x	x	0	x	x	x	0	x	10	xx	xxx
	halt	0	x	x	x	0	x	x	x	0	x	xx	xx	xxx
sEXE(110)	add	0	0	0	x	0	x	x	x	0	x	00	xx	000
	sub	0	0	0	x	0	x	x	x	0	x	00	xx	001
	addiu	0	0	1	x	0	x	x	x	0	1	00	xx	000
	and	0	0	0	x	0	x	x	x	0	0	00	xx	100
	andi	0	0	1	x	0	x	x	x	0	0	00	xx	100
	ori	0	0	1	x	0	x	x	x	0	0	00	xx	011
	xori	0	0	1	x	0	x	x	x	0	0	00	xx	111
	sll	0	1	0	x	0	x	x	x	0	0	00	xx	010
	slt	0	0	0	x	0	x	x	x	0	x	00	xx	110
	slti	0	0	1	x	0	x	x	x	0	1	00	xx	110
sEXE(101)	beq(z ero=1)	0	0	0	x	0	x	x	x	0	1	01	xx	001
	bne(z ero=0)	0	0	0	x	0	x	x	x	0	1	01	xx	001
	bltz(s ign=1)	0	0	0	x	0	x	x	x	0	1	01	xx	001
sEXE(010)	sw	0	0	1	x	0	x	x	x	0	1	00	xx	000
	lw	0	0	1	x	0	x	x	x	0	1	00	xx	000
sME M(01 1)	sw	0	x	x	0	x	x	0	1	0	x	xx	xx	xxx
	lw	0	x	x	1	x	x	1	0	0	x	xx	xx	xxx
sWB(111)	add	0	x	x	0	1	1	x	x	0	x	xx	10	xxx
	sub	0	x	x	0	1	1	x	x	0	x	xx	10	xxx

	addiu	0	x	x	0	1	1	x	x	0	x	xx	01	xxx
	and	0	x	x	0	1	1	x	x	0	x	xx	10	xxx
	andi	0	x	x	0	1	1	x	x	0	x	xx	01	xxx
	ori	0	x	x	0	1	1	x	x	0	x	xx	01	xxx
	xori	0	x	x	0	1	1	x	x	0	x	xx	01	xxx
	sll	0	x	x	0	1	1	x	x	0	x	xx	10	xxx
	slt	0	x	x	0	1	1	x	x	0	x	xx	10	xxx
	slti	0	x	x	0	1	1	x	x	0	x	xx	01	xxx
sWB(100)	lw	0	x	x	1	1	1	x	x	0	x	xx	01	xxx

由上表，写出ControlUnit模块。

ControlUnit:

通过条件判断给控制信号赋值。一位的控制信号依据当前操作码和状态赋值，多位的控制信号（PCsrc,ALUop等）按位分别赋值。当敏感信号（操作码，状态码，零标志，符号标志，pc）变化时，会对控制信号重新赋值。注：PCWre仅在000状态为1时，会由于数据延迟，导致PC不能在000状态同时转换，采用的解决方式是：在000状态的前一个状态将PCWre赋值为1，使PC在下一个000状态能够及时更改。

依据上表，控制模块代码如下：

```
module ControlUnit(pc,condition, opCode, zero, sign, PCWre, IRWre, ALUSrcA, ALUSrcB,
DBDataSrc, RegWre, RD, WR, ExtSel, PCSrc, RegDst, ALUOp, WrRegDSrc);
    input [31:0] pc;
    input [5:0] opCode;
    input [2:0] condition;
    input zero, sign;
    output reg PCWre, IRWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, RD, WR, ExtSel,
WrRegDSrc;
    output reg [1:0] PCSrc, RegDst;
    output reg [2:0] ALUOp;

    always@(condition or opCode or zero or sign or pc) begin
        IRWre = (condition == 3'b000) ? 1 : 0;
        PCWre = (condition==3'b111 || condition==3'b101 || condition==3'b100
        || (condition==3'b011&&opCode==6'b110000) || (condition==3'b001&&(opCode == 6'b111000
        || opCode == 6'b111001 || opCode == 6'b111010)))?1:0;
        RD = opCode == 6'b110001 ? 1 : 0;
        WR = (condition == 3'b011) ? (opCode == 6'b110000 ? 1 : 0) : 0;
        RegWre = (condition == 3'b111 || condition == 3'b100 || (condition==001 &&
opCode == 6'b111010)) ? 1 : 0;
        ALUSrcA = (opCode == 6'b011000) ? 1 : 0;
        ALUSrcB = (opCode == 6'b000010 || opCode == 6'b010001 || opCode ==
6'b010010 || opCode == 6'b010011 || opCode == 6'b100110 || opCode == 6'b110000 ||
opCode == 6'b110001) ? 1 : 0;
        DBDataSrc = (opCode == 6'b110001) ? 1 : 0;
        WrRegDSrc = (opCode == 6'b111010 && condition==001 ) ? 0 : 1;
```

```

ExtSel = (opCode == 6'b010000 || opCode == 6'b010001 || opCode == 6'b010010
|| opCode == 6'b010011 || opCode == 6'b011000) ? 0 : 1;
    PCSrc[1] = (opCode == 6'b111000 || opCode == 6'b111001 || opCode ==
6'b111010) ? 1 : 0;
    PCSrc[0] = ((opCode == 6'b110100 && zero == 1) || (opCode == 6'b110101 && zero
== 0) || (opCode == 6'b110110 && sign==1) || opCode == 6'b111000 || opCode ==
6'b111010) ? 1 : 0;
    RegDst[1] = (opCode == 6'b000010 || opCode == 6'b010001 || opCode ==
6'b010010 || opCode == 6'b010011 || opCode == 6'b100110 || opCode == 6'b110001 ||
opCode == 6'b111010) ? 0 : 1;
    RegDst[0] = (opCode == 6'b000010 || opCode == 6'b010001 || opCode ==
6'b010010 || opCode == 6'b010011 || opCode == 6'b100110 || opCode == 6'b110001) ? 1 : 0;
    ALUOp[2] = (opCode == 6'b010000 || opCode == 6'b010001 || opCode ==
6'b010011 || opCode == 6'b100111 || opCode == 6'b100110) ? 1 : 0;
    ALUOp[1] = (opCode == 6'b010010 || opCode == 6'b010011 || opCode ==
6'b011000 || opCode == 6'b100110 || opCode == 6'b100111) ? 1 : 0;
    ALUOp[0] = (opCode == 6'b000000 || opCode == 6'b000010 || opCode ==
6'b010000 || opCode == 6'b010001 || opCode == 6'b011000 || opCode == 6'b100110 ||
opCode == 6'b110000 || opCode == 6'b110001 || opCode == 6'b100111) ? 0 : 1;
    end
endmodule

```

nextCondition:

按照状态转换表，根据当前状态确定下一状态（注意，此时状态不改变。状态的改变在D触发器模块中进行）。

```

module nextCondition(con, opcode, nextcon);
    input[2:0] con;
    input[5:0] opcode;
    output reg [2:0] nextcon;
    always@(con) begin
        case(con)
            3'b000:nextcon = con + 1;
            3'b001:if(opcode == 6'b111000 || opcode == 6'b111010 || opcode ==
6'b111001 || opcode == 6'b111111)
                nextcon = 3'b000;
            else if(opcode == 6'b110100 || opcode == 6'b110101 || opcode ==
6'b110110)
                nextcon = 3'b101;
            else if(opcode == 6'b110000 || opcode == 6'b110001)
                nextcon = 3'b010;
            else
                nextcon = 3'b110;
            3'b110:nextcon = 3'b111;
            3'b101:nextcon = 3'b000;
        endcase
    end
endmodule

```

```

3'b010:nextcon = 3'b011;
3'b011:if(opcode == 6'b110000)
    nextcon=3'b000;
else if(opcode == 6'b110001)
    nextcon=3'b100;
3'b111:nextcon = 3'b000;
3'b100:nextcon = 3'b000;
endcase
end
endmodule

```

Dtrigger:

由时钟上升沿触发，改变当前状态。Reset=0时，将状态初始化为000。

```

module Dtrigger(clk, reset, conin, conout);
    input clk,reset;
    input[2:0] conin;
    output reg [2:0] conout;
    always@(posedge clk)
        if(reset == 0)
            conout = 3'b000;
        else
            conout = conin;
endmodule

```

insROM:

insROM实现的是从编写好的指令文件Rom_data.txt文件中读取所有的指令，按照8位为1个单元的格式读取所有指令，按顺序存储到自定义的8位存储器insROM中，并在之后的测试过程中依次调用。关键代码如下：

```

module insROM ( pc, RW, ins);
    input [31:0] pc;
    input RW;//在单周期CPU中始终为0
    output reg [31:0] ins;
    reg [7:0] rom [99:0];
    initial begin //加载数据到指令存储器insROM
        $readmemb
        ("D:/Sophomore/ECOP-2018/ECOP-17343155-03/CPU_sim/rom.txt", rom);
    end
    always @( RW or pc ) begin
        if (RW==1) begin //将指令读出 大端方式
            ins[31:24] = rom[pc];
            ins[23:16] = rom[pc+1];
            ins[15:8] = rom[pc+2];
            ins[7:0] = rom[pc+3];
        end
    end
endmodule

```

```

    end
endmodule

```

IR:

指令寄存器，在时钟上升沿到来时，将从insROM传来的指令进行存储并划分出操作码，rs,rt,rd,立即数。

```

module IR(clk, ins, IRWre, op, rs, rt, rd, sa, immediate, jumpimmediate);
    input clk,IRWre;
    input[31:0] ins;
    output reg [5:0] op;
    output reg [4:0] rs, rt, rd, sa;
    output reg [15:0] immediate;
    output reg [25:0] jumpimmediate;
    always@(posedge clk) begin
        if(IRWre) begin          //IR
            op = ins [31:26];
            rs = ins [25:21];
            rt = ins [20:16];
            rd = ins [15:11];
            immediate = ins [15:0];
            sa = ins [10:6];
            jumpimmediate = ins [25:0];
        end
    end
endmodule

```

DR:

DR 包括：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。时钟上升沿触发。

```

module DR(clk, reg1, reg2, result, ramdata,aluA, aluB, ramadd, wdata,DBDataSrc);
    input clk, DBDataSrc;
    input[31:0] reg1, reg2, result, ramdata;
    output reg [31:0] aluA, aluB, ramadd, wdata;
    always@(posedge clk) begin
        aluA = reg1;
        aluB = reg2;
        ramadd = result;
        wdata = DBDataSrc ? ramdata : result;
    end
endmodule

```

ALU:

ALU模块，输入为3位功能码（结合ALU运算功能表）和两个需要运算的32位数，输出为零标志，符号标志和32位运算结果。功能码决定了进行的运算类型，运算结果为零时，

零标志为1，其余情况为0， 运算结果为正或零，符号标志为0，结果为负，符号标志为1。

```

module ALU(ALUopcode, rega, regb, zero, sign, result);
    input [2:0] ALUopcode;
    input [31:0] rega;
    input [31:0] regb;
    output reg zero;
    output reg sign;
    output reg [31:0] result;
    always @( ALUopcode or regb or rega) begin
        case (ALUopcode)
            3'b000 : result = rega + regb;
            3'b001 : result = rega - regb;
            3'b010 : result = regb << rega;
            3'b011 : result = rega | regb;
            3'b100 : result = rega & regb;
            3'b101 : result = (rega < regb)?1:0; //无符号比较
            3'b110 : if (rega<regb &&(rega[31] == regb[31] )) //有符号比较
                result = 1;
            else if ( rega[31]==1 && regb[31]==0)
                result = 1;
            else result = 0;
            3'b111 : result = rega ^ regb;
        default : begin
            result = 32'h00000000;
            $display ("ERROE");
        end
    endcase
    sign = result[31];
    zero = (result==0) ? 1:0;
end
endmodule

```

extend:

立即数扩展模块， ExSel决定扩展类型， 0为零扩展， 1为符号扩展。

```

module extend(immediate, ExtSel, out);
    input [15:0] immediate;
    input ExtSel;
    output [31:0] out;
    assign out[15:0] = immediate;
    assign out[31:16] = ExtSel? (immediate[15]? 16'hffff : 16'h0000) : 16'h0000;
    //ExSel为1， 符号位扩展， ExSel为0， 零扩展
endmodule

```

ALU_dataA:

ALU输入A选择模块，根据控制信号选择寄存器的值或sa。

```
module ALU_dataA(ALUsrcA, rega, sa, dataA);
    input ALUsrcA;
    input [31:0] rega;
    input [4:0] sa;
    output [31:0] dataA;
    assign dataA = ALUsrcA ? {{27{0}},sa} : rega;
endmodule
```

ALU_dataB:

ALU输入B选择模块，根据输入信号选择寄存器的值或立即数。

```
module ALU_dataB(ALUsrcB, regb, immediate, dataB);
    input ALUsrcB;
    input [31:0] regb;
    input [31:0] immediate;
    output wire [31:0] dataB;
    assign dataB = ALUsrcB ? immediate : regb;
endmodule
```

regFile:

寄存器组模块，依据传入地址读相应寄存器的值，当时钟下降沿到来时将数据写入对应寄存器，RegWre是写使能信号，为1时可以写寄存器，为0时只可读，同时0号寄存器的值不可修改，恒为0。注：reg0,reg1,...,reg14,reg31是为了方便在仿真结果中进行检查。

```
module regFile(CLK, RST, RegWre, ReadReg1, ReadReg2, WriteReg, WriteData, ReadData1,
ReadData2, reg0, reg1, reg2, reg3, reg4, reg5, reg8, reg9, reg10, reg11, reg12, reg13, reg14,
reg31);
    input CLK;
    input RST;
    input RegWre;
    input [4:0] ReadReg1,ReadReg2,WriteReg;
    input [31:0] WriteData;
    output [31:0] ReadData1,ReadData2,reg0, reg1, reg2, reg3, reg4, reg5, reg8, reg9,
reg10, reg11, reg12, reg13, reg14, reg31;
    reg [31:0] regFile[0:31];
    integer i;
    assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
    assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
    assign reg0 = regFile[0];
    assign reg1 = regFile[1];
    assign reg2 = regFile[2];
    assign reg3 = regFile[3];
    assign reg4 = regFile[4];
    assign reg5 = regFile[5];
```

```

assign reg8 = regFile[8];
assign reg9 = regFile[9];
assign reg10 = regFile[10];
assign reg11 = regFile[11];
assign reg12 = regFile[12];
assign reg13 = regFile[13];
assign reg14 = regFile[14];
assign reg31 = regFile[31];
always @ (negedge CLK) begin // 必须用时钟边沿触发
    if (RST==0) begin
        for(i=0;i<32;i=i+1)
            regFile[i] <= 0;
    end
    if(RegWre == 1 && WriteReg != 0) // WriteReg != 0, $0 寄存器不能修改
        regFile[WriteReg] <= WriteData; // 写寄存器
    end
endmodule

```

regWriteAddress:

寄存器写地址选择模块，选择需要写的寄存器是rd，rt，还是\$31。

```

module regWriteAddress(RegDst, rtreg, rdreg, regwrite);
    input [1:0] RegDst;
    input [4:0] rtreg;
    input [4:0] rdreg;
    output [4:0] regwrite;
    assign regwrite = (RegDst==2'b00) ? 5'b11111 : (RegDst==2'b01 ? rtreg : rdreg);
endmodule

```

regWriteData:

寄存器写数据选择模块，选择需要写进寄存器的值来自数据总线还是PC。

```

module regWriteData(WrRegDSrc, PC, DB, datawrite);
    input WrRegDSrc;
    input [31:0] PC, DB;
    output [31:0] datawrite;
    assign datawrite = WrRegDSrc ? DB : PC+4;
endmodule

```

dataRAM:

数据存储器模块，mRD信号为1时可读，mWR信号为1时可写，写时将数据写入地址值对应的存储单元，读时将地址对应的存储单元的值输出。

```

module dataRAM(address, writeData, mRD, mWR, Dataout);
    input [31:0] address;
    input [31:0] writeData;
    input mRD;

```

```

input mWR;
output [31:0] Dataout;
reg [7:0] ram [0:60];
// 读
assign Dataout[7:0] = (mRD==1)?ram[address + 3]:8'bz; // z为高阻态
assign Dataout[15:8] = (mRD==1)?ram[address + 2]:8'bz;
assign Dataout[23:16] = (mRD==1)?ram[address + 1]:8'bz;
assign Dataout[31:24] = (mRD==1)?ram[address ]:8'bz;
// 写
always@( mWR ) begin
    if( mWR==1 ) begin
        ram[address] <= writeData[31:24];
        ram[address+1] <= writeData[23:16];
        ram[address+2] <= writeData[15:8];
        ram[address+3] <= writeData[7:0];
    end
end
endmodule

```

Jext:

跳转立即数扩展，将j指令中的立即数扩展为32位地址，最后两位为0，前四位为PC+4的前四位。

```

module Jext(pc, jimme, jpc);
    input [31:0] pc;
    input [25:0] jimme;
    output [31:0] jpc;
    assign jpc[27:2] = jimme[25:0];
    assign jpc[1:0] = 2'b00;
    assign jpc[31:28] = pc[31:28];
endmodule

```

PC:

PC模块，PC的值为指令在指令存储器中的地址，使用同步清零的方法，当上升沿到来时，Reset为0则清零，PCWre为1代表可更改，为0代表不可更改，即停机状态。

```

module PC(clk, Reset, PCWre, nextAddress, Address);
    input clk, Reset, PCWre;
    input [31:0] nextAddress;
    output reg [31:0] Address;
    always @(posedge clk)begin
        if (Reset == 0)
            Address <= 0;
        else if(PCWre==1)
            Address <= nextAddress;
    end
endmodule

```

nextPC:

确定PC的下一个值，由PCSrc决定，00则为PC+4，即顺序执行；01则为偏移对应的指令条数；10则跳转至对应的地址(jr指令)；11则跳至立即数对应的地址(j, jal指令)。

```
module nextPC(pc, PCSrc, imme, rsData, jpc, nextpc);
    input [31:0] pc;
    input [1:0] PCSrc;
    input [31:0] imme;
    input [31:0] jpc;
    input [31:0] rsData;
    output reg [31:0] nextpc;
    always@(pc or PCSrc) begin
        case(PCSrc)
            2'b00:nextpc = pc + 4;
            2'b01:nextpc = pc + (imme << 2) +4;
            2'b10:nextpc = rsData;
            2'b11:nextpc = jpc;
            default:;
        endcase
    end
endmodule
```

CPU_top:

顶层模块，实例化其余底层模块，并将其组合起来。

```
module CPU_top(
    input clk, Reset,
    output zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, RD,
    WR, ExtSel, IRWre, WrRegDSrc,
    output [1:0] PCSrc, RegDst,
    output [2:0] ALUOp, con, nextcon,
    output [4:0] rs, rd, rt, sa, regwrite,
    output [5:0] Opcode,
    output [31:0] DB, Pc, nextPC, ins, Regdata1, Regdata2, WRegdata, Result, ExtOut, Jpc,
    Ramout, dataA, dataB, reg0, reg1, reg2, reg3, reg4, reg5, reg8, reg9, reg10, reg11, reg12, reg13,
    reg14, reg31
);
    wire [15:0] immediate;
    wire [25:0] jimme;
    wire [31:0] aluA, aluB, Ramadd;
    assign InsMemRW=1;
    PC pc(clk, Reset, PCWre, nextPC, Pc);
    insROM rom(Pc, InsMemRW, ins);
    Dtrigger dtri(clk, Reset, nextcon, con);
    IR ireg(clk, ins, IRWre, Opcode, rs, rt, rd, sa, immediate, jimme);
    DR dreg(clk, Regdata1, Regdata2, Result, Ramout, aluA, aluB, Ramadd, DB,
```

```

DBDataSrc);
nextCondition nextCon(con, Opcode, nextcon);
ControlUnit control(Pc,con, Opcode, zero, sign, PCWre, IRWre, ALUSrcA, ALUSrcB,
DBDataSrc, RegWre, RD, WR, ExtSel, PCSrc, RegDst, ALUOp, WrRegDSrc);
extend ext(immediate, ExtSel, ExtOut);
Jext jext(Pc, jimme, Jpc);
regWriteAddress regadd(RegDst, rt, rd, regwrite);
regWriteData regdata(WrRegDSrc, Pc, DB, WRegdata);
regFile regf(clk,Reset,RegWre, rs, rt, regwrite, WRegdata, Regdata1, Regdata2, reg0,
reg1, reg2, reg3, reg4, reg5, reg8, reg9, reg10, reg11, reg12, reg13, reg14, reg31);
ALU_dataA aluDataA(ALUSrcA, Regdata1, sa, dataA);
ALU_dataB aluDataB(ALUSrcB, Regdata2, ExtOut, dataB);
ALU alu(ALUOp, dataA, dataB, zero, sign, Result);
dataRAM ram(Ramadd, aluB, RD, WR, Ramout);
nextPC nextpc(Pc, PCSrc, ExtOut, Regdata1, Jpc, nextPC);
endmodule

```

3.多周期CPU仿真

完善测试表格，写出测试代码：

(测试程序段)

地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addiu \$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	xori \$3,\$2,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008
0x0000000C	sub \$4,\$3,\$1	000001	00011	00001	0010 0000 0000 0000	=	04612000
0x00000010	and \$5,\$4,\$2	010000	00100	00010	0010 1000 0000 0000	=	40822800
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 0x00000050	111010	00000	00000	0000 0000 0001 0100	=	E8000014
0x00000020	slt \$8,\$13,\$1	100111	01101	00001	0100 0000 0000 0000	=	9DA14000
0x00000024	addiu \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFE
0x00000028	slt \$9,\$8,\$14	100111	01000	01110	0100 1000 0000 0000	=	9D0E4800
0x0000002C	slti \$10,\$9,2	100110	01001	01010	0000 0000 0000 0010	=	992A0002

0x00000030	slti \$11,\$10,0	100110	01010	01011	0000 0000 0000 0000	=	994B0000
0x00000034	add \$11,\$11,\$10	000000	01011	01010	0101 1000 0000 0000	=	016A5800
0x00000038	bne \$11,\$2,-2 (≠, 转 34)	110101	01011	00010	1111 1111 1111 1110	=	D562FFFE
0x0000003C	addiu \$12,\$0,-2	000010	00000	01100	1111 1111 1111 1110	=	080CFFFE
0x00000040	addiu \$12,\$12,1	000010	01100	01100	0000 0000 0000 0001	=	098C0001
0x00000044	bltz \$12,-2 (<0, 转 40)	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE
0x00000048	andi \$12,\$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002
0x0000004C	j 0x0000005C	111000	00000	00000	0000 0000 0001 0111	=	E0000017
0x00000050	sw \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004
0x00000054	lw \$13,4(\$1)	110001	00001	01101	0000 0000 0000 0100	=	C42D0004
0x00000058	jr \$31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x0000005C	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

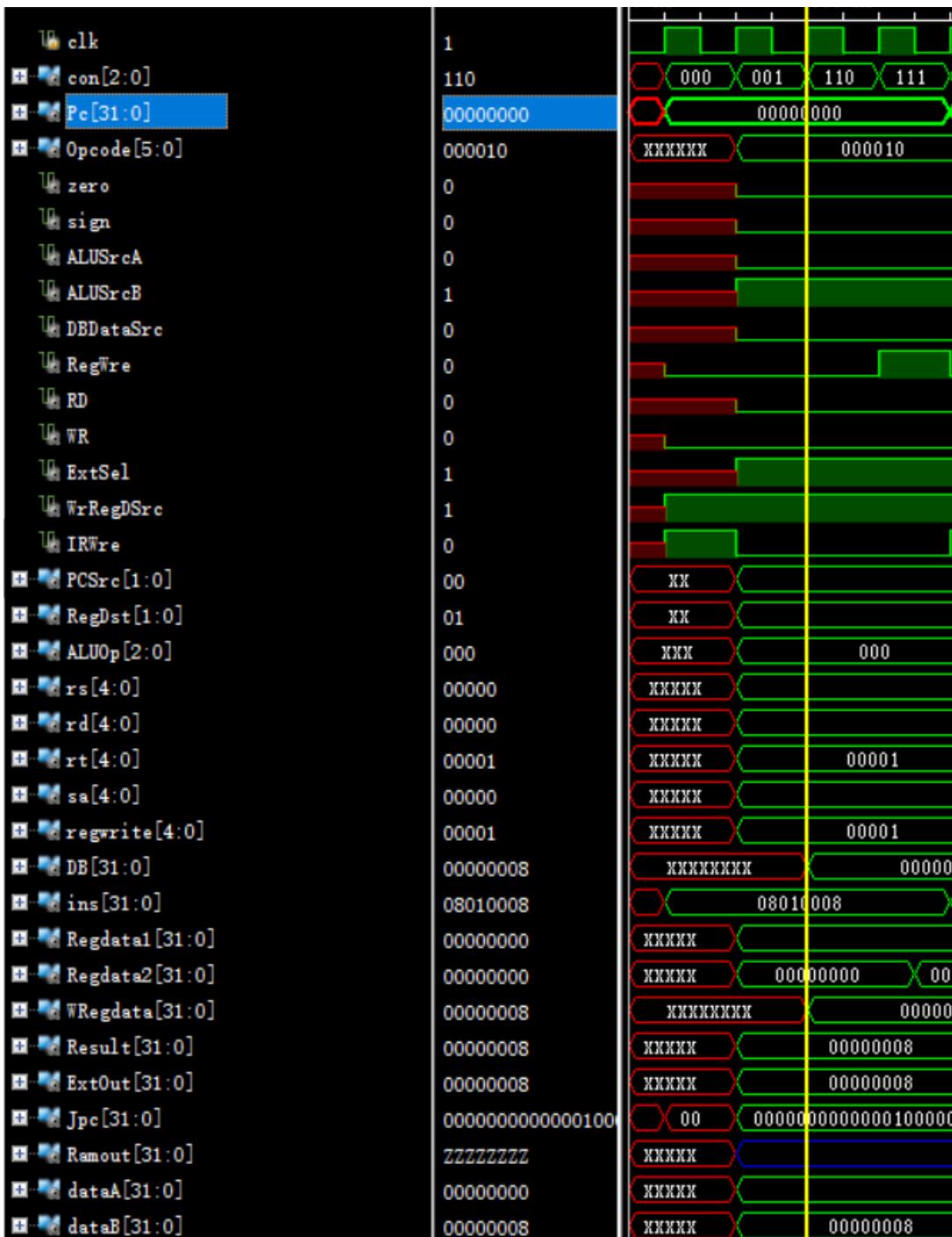
(测试代码)

```

00001000 00000001 00000000 00001000
01001000 00000010 00000000 00000010
01001100 01000011 00000000 00001000
00000100 01100001 00100000 00000000
01000000 10000010 00101000 00000000
01100000 00000101 00101000 10000000
11010000 10100001 11111111 11111110
11101000 00000000 00000000 00010100
10011101 10100001 01000000 00000000
00001000 00001110 11111111 11111110
10011101 00001110 01001000 00000000
10011001 00101010 00000000 00000010
10011001 01001011 00000000 00000000
00000001 01101010 01011000 00000000
11010101 01100010 11111111 11111110
00001000 00001100 11111111 11111110
00001001 10001100 00000000 00000001
11011001 10000000 11111111 11111110
01000100 01001100 00000000 00000010
11100000 00000000 00000000 00010111
11000000 00100010 00000000 00000100
11000100 00101101 00000000 00000100
11100111 11100000 00000000 00000000
11111100 00000000 00000000 00000000

```

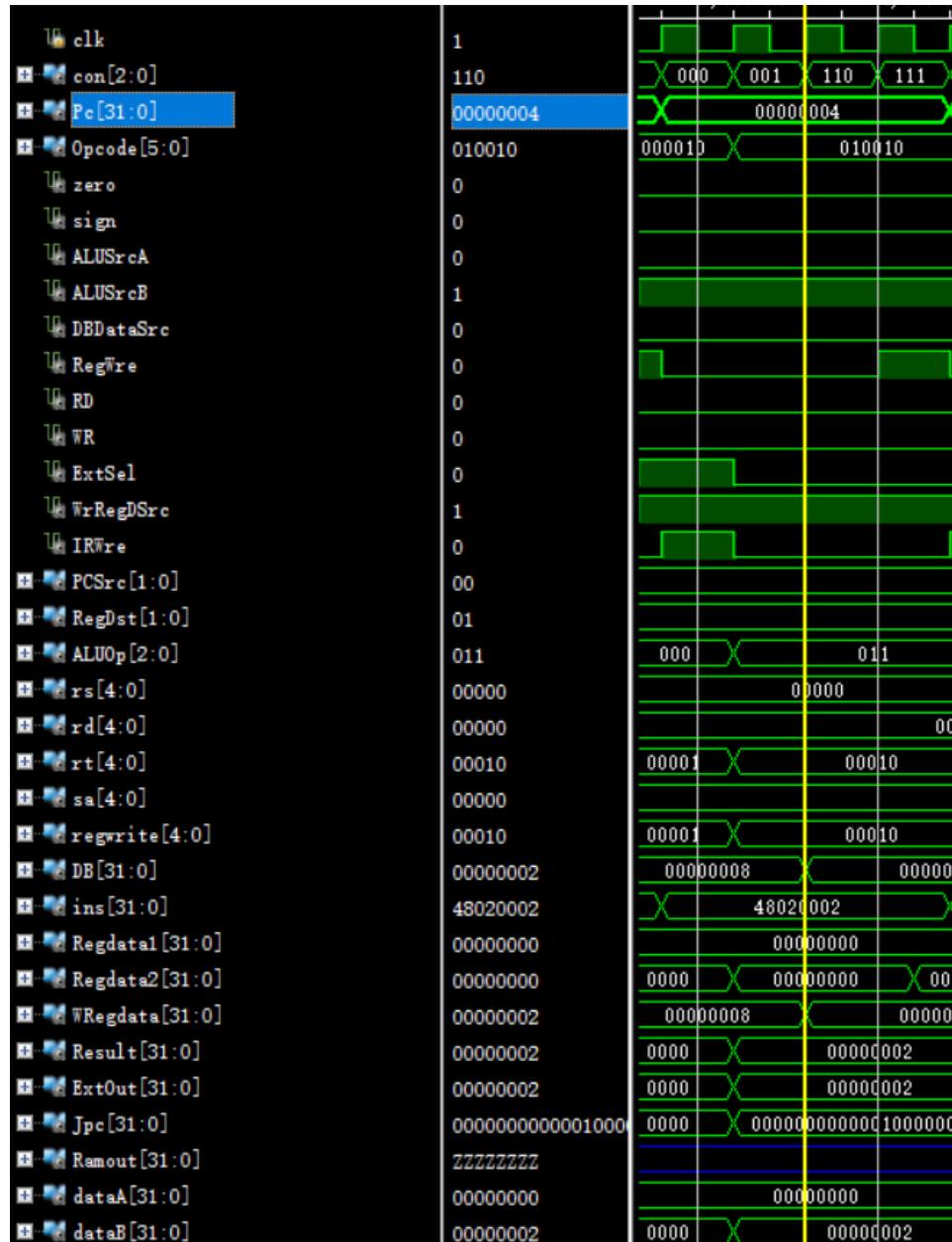
(1) addiu \$1,\$0,8



开始时，reset为0，PC=00，condion=000。

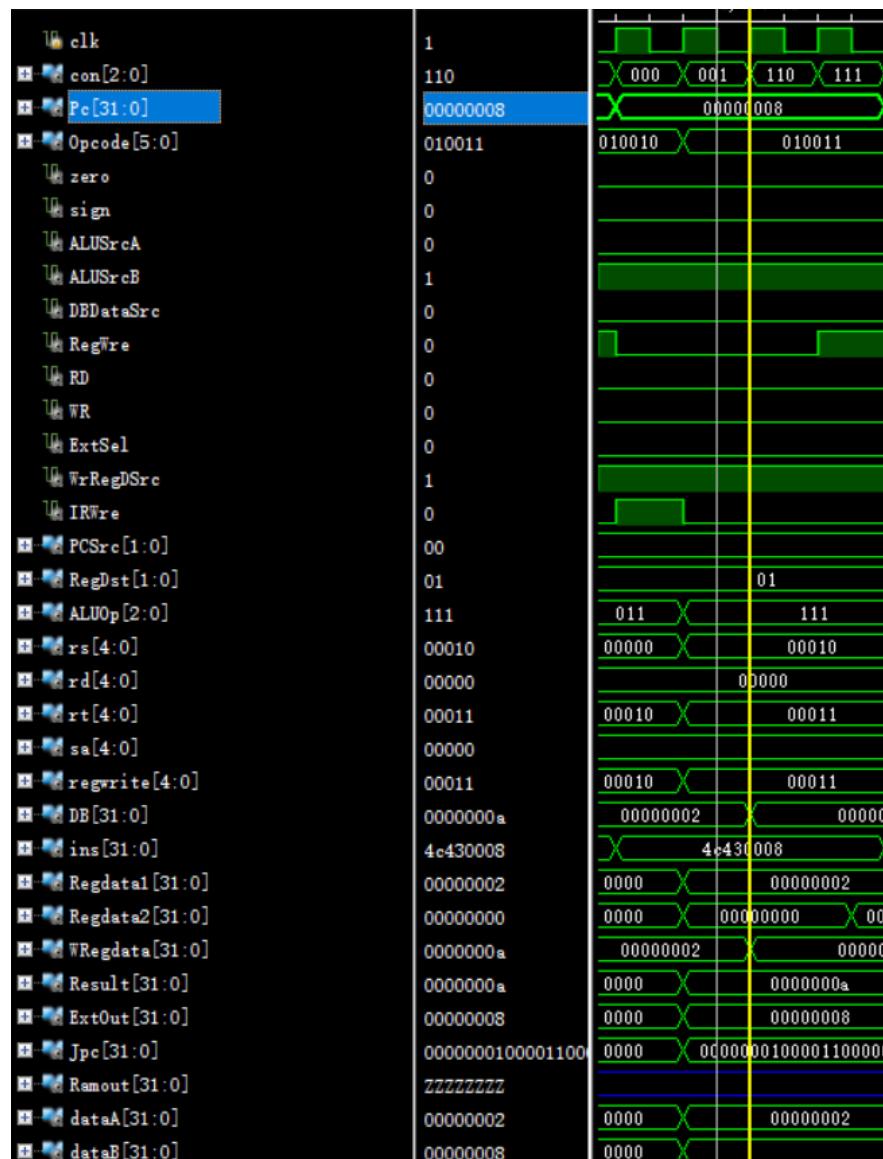
Pc=00, 执行addiu \$1,\$0,8。000状态取指, 001状态译码 (写入IR), 操作码为000010, rt=1, rs=0, ALUOp=000。dataA来自\$0,dataB来自立即数8的符号扩展, 在110状态进行加运算。结果为8, 在111写回状态写入rt (此时\$1=8)。PCsrc=00,nextPC=04。

(2) ori \$2,\$0,2



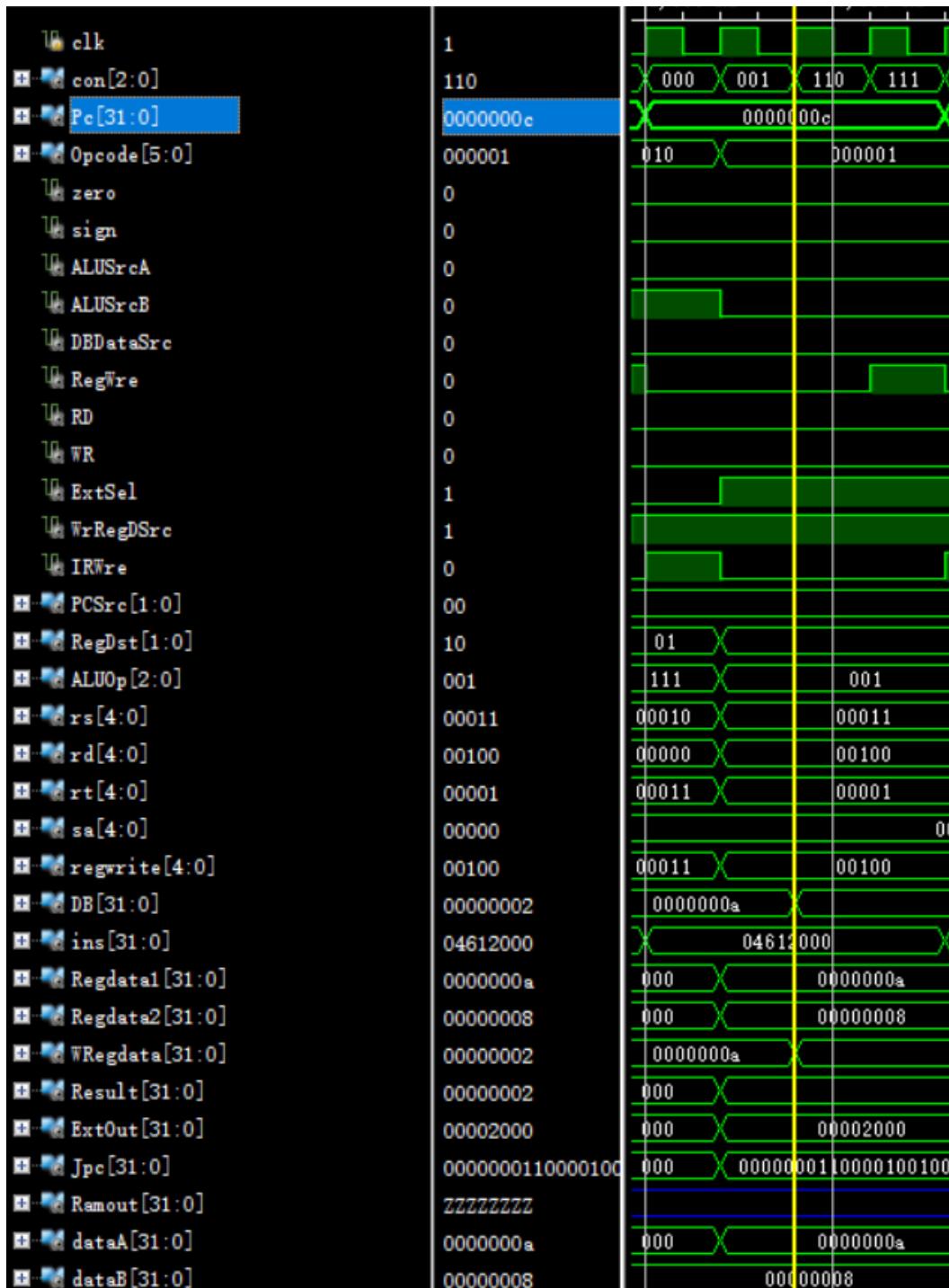
Pc=04, 执行ori \$2,\$0,2。000状态取指, 001状态译码, 操作码为010010, rt=2, rs=0, ALUOp=011。dataA来自\$0,dataB来自立即数2的零扩展, 在110状态进行或运算。结果为2, 在111写回状态写入rt (此时\$1=8 \$2=2)。PCsrc=00,nextPC=08。

(3) xori \$3,\$2,8



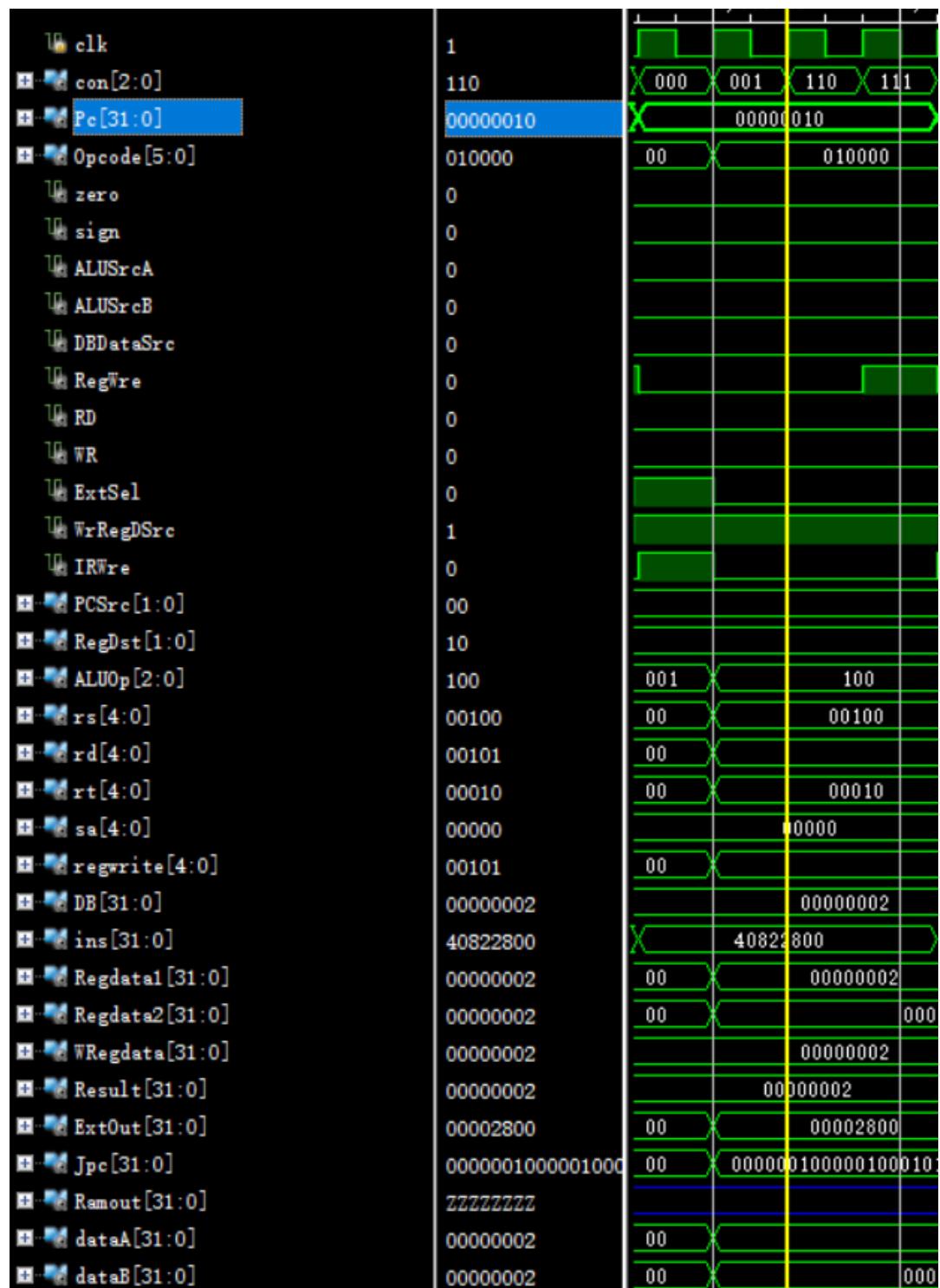
Pc=08, 执行xori \$3,\$2,8。000状态取指, 001状态译码, 操作码为010011, rt=3, rs=2, ALUOp=111。dataA来自\$2,dataB来自立即数8的零扩展, 在110状态进行异或运算。结果为10, 在111写回状态写入rt (此时\$1=8 \$2=2 \$3=10)。PCsrc=00,nextPC=0C。

(4) sub \$4,\$3,\$1



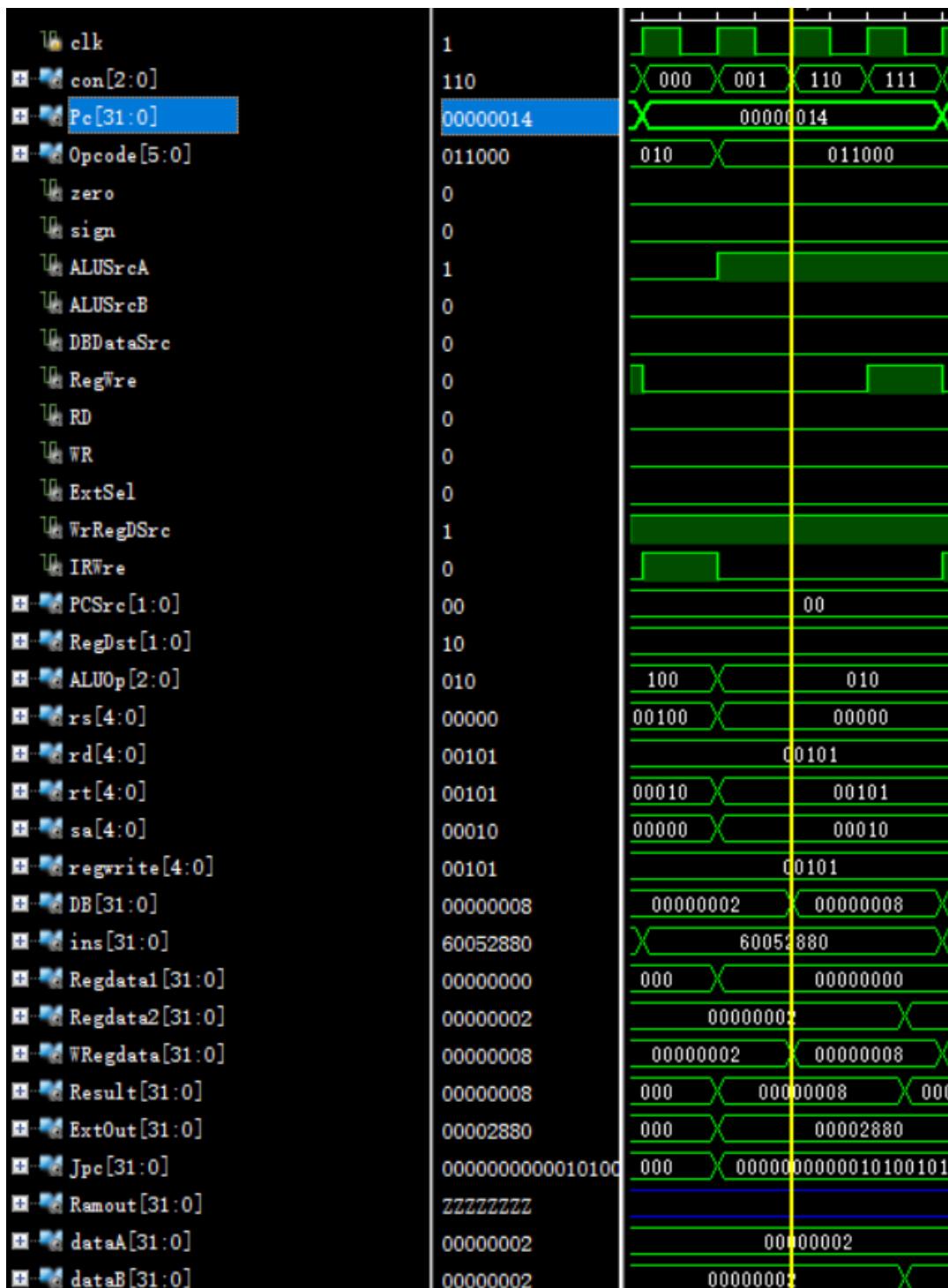
Pc=0C, 执行sub \$4,\$3,\$1。000状态取指，001状态译码，操作码为000001, rt=1, rs=3, rd=4, ALUOp=001。dataA来自\$3,dataB来自\$1, 在110状态进行减运算。结果为2，在111写回状态写入rd (此时\$1=8 \$2=2 \$3=10 \$4=2)。PCsrc=00,nextPC=10。

(5) and \$5,\$4,\$2



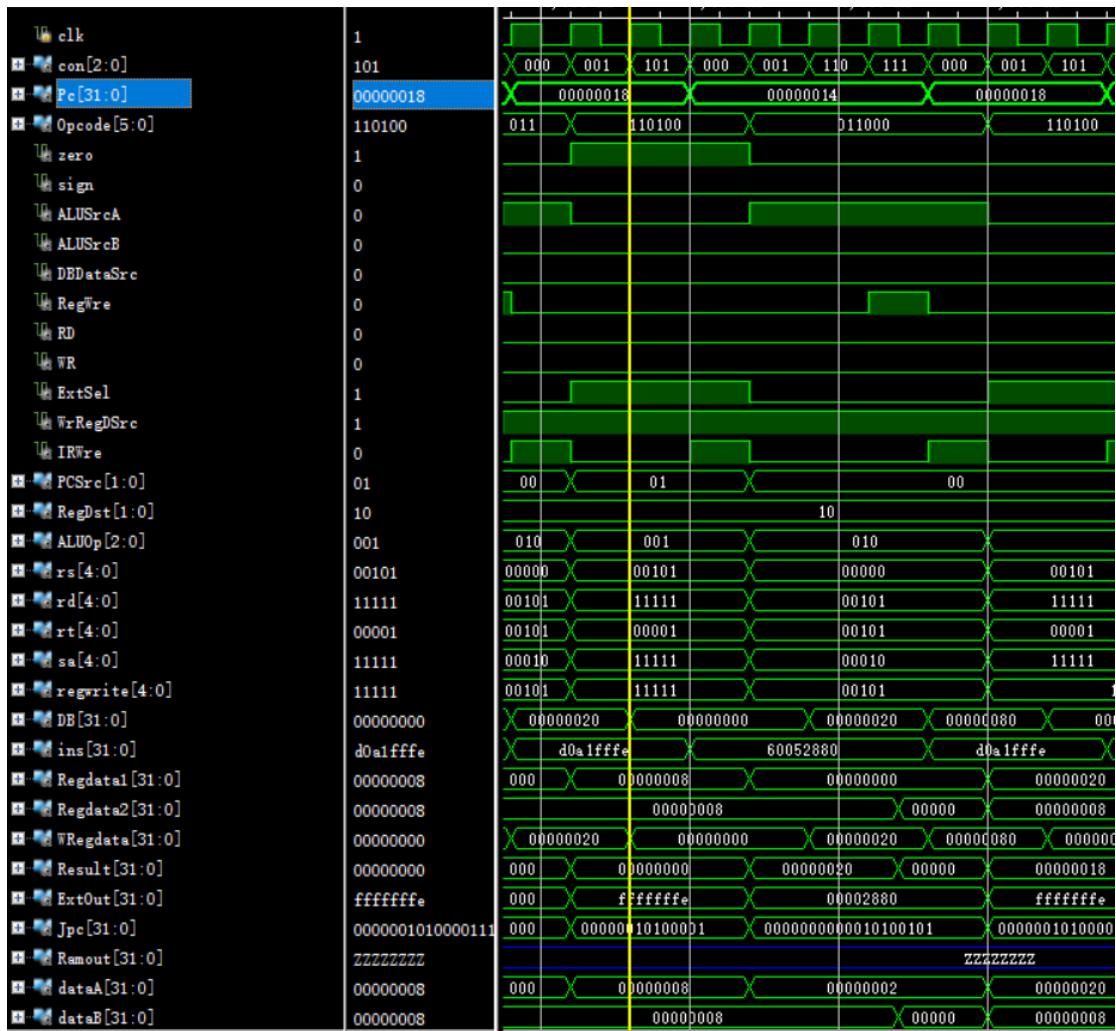
Pc=10, 执行and \$5,\$4,\$2。000状态取指，001状态译码，操作码为010000, rt=2, rs=4, rd=5, ALUOp=100。dataA来自\$4,dataB来自\$2, 在110状态进行与运算。结果为2，在111写回状态写入rd (此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=2)。PCsrc=00,nextPC=14。

(6) sll \$5,\$5,2



Pc=14, 执行sll \$5,\$5,2。000状态取指，001状态译码，操作码为011000, rt=5, rd=5, ALUOp=010。dataA来自移位数2,dataB来自\$5，在110状态进行左移运算。结果为8，在111写回状态写入rd (此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=8)。PCsrc=00,nextPC=18。

(7) beq \$5,\$1,-2(=,转14)



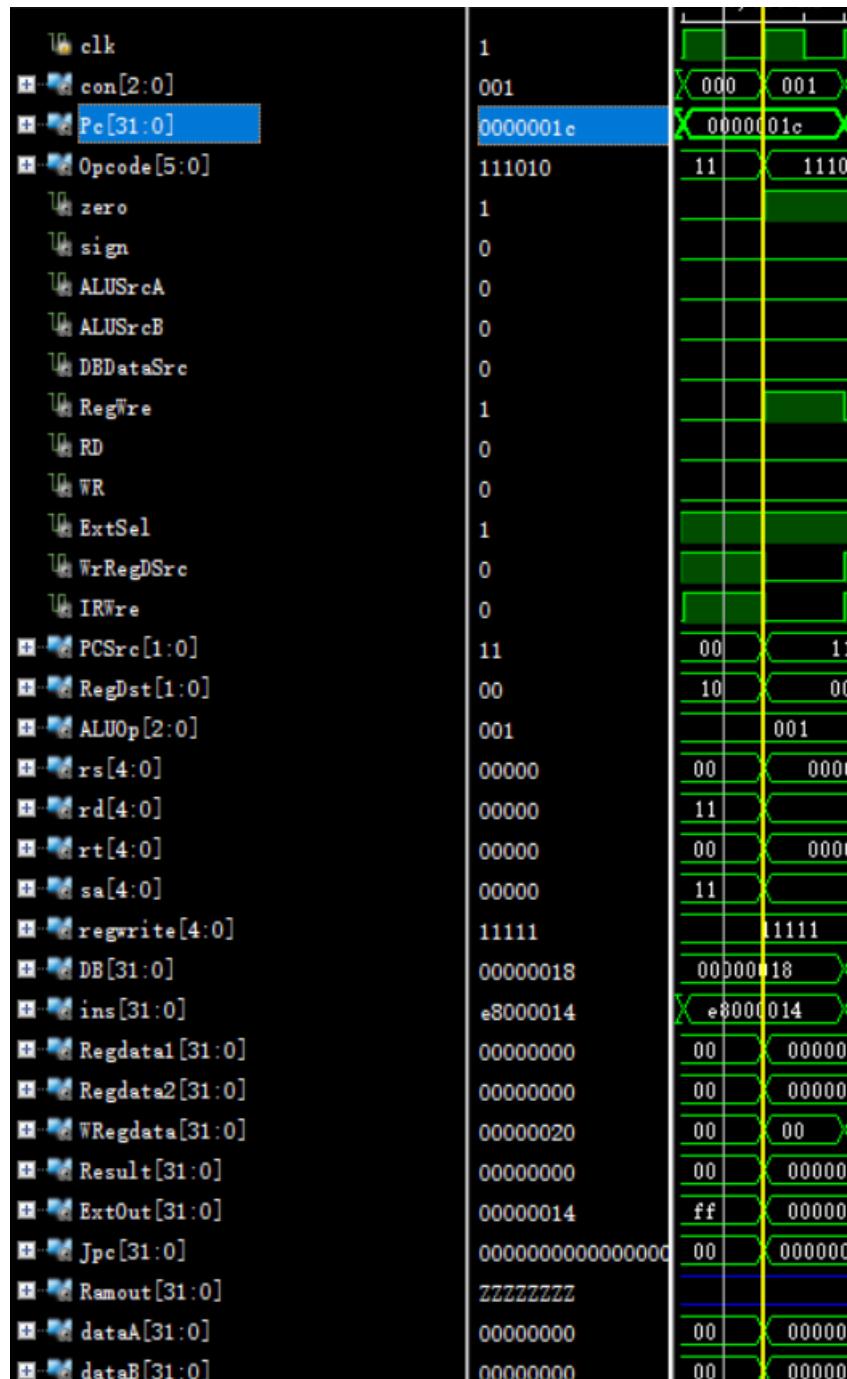
PC=18, 执行beq \$5,\$1,-2, 000状态取指, 001状态译码, 操作码为110100, rs=5, rt=1, 立即数为-2, ALUOp=001。立即数为需要跳转的指令数, 需要进行符号扩展再左移两位, dataA来自rs, dataB来自rt, 在101状态进行减法运算。结果为0。 PCSrc=01, $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$, nextPC=14.
(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=8)

再次执行sll \$5,\$5,2。结果为\$5=32。 PCSrc=00,nextPC=18.

第二次执行beq \$5,\$1,-2。 rs!=rt, PCSrc=00,nextPC=1C.

(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32)

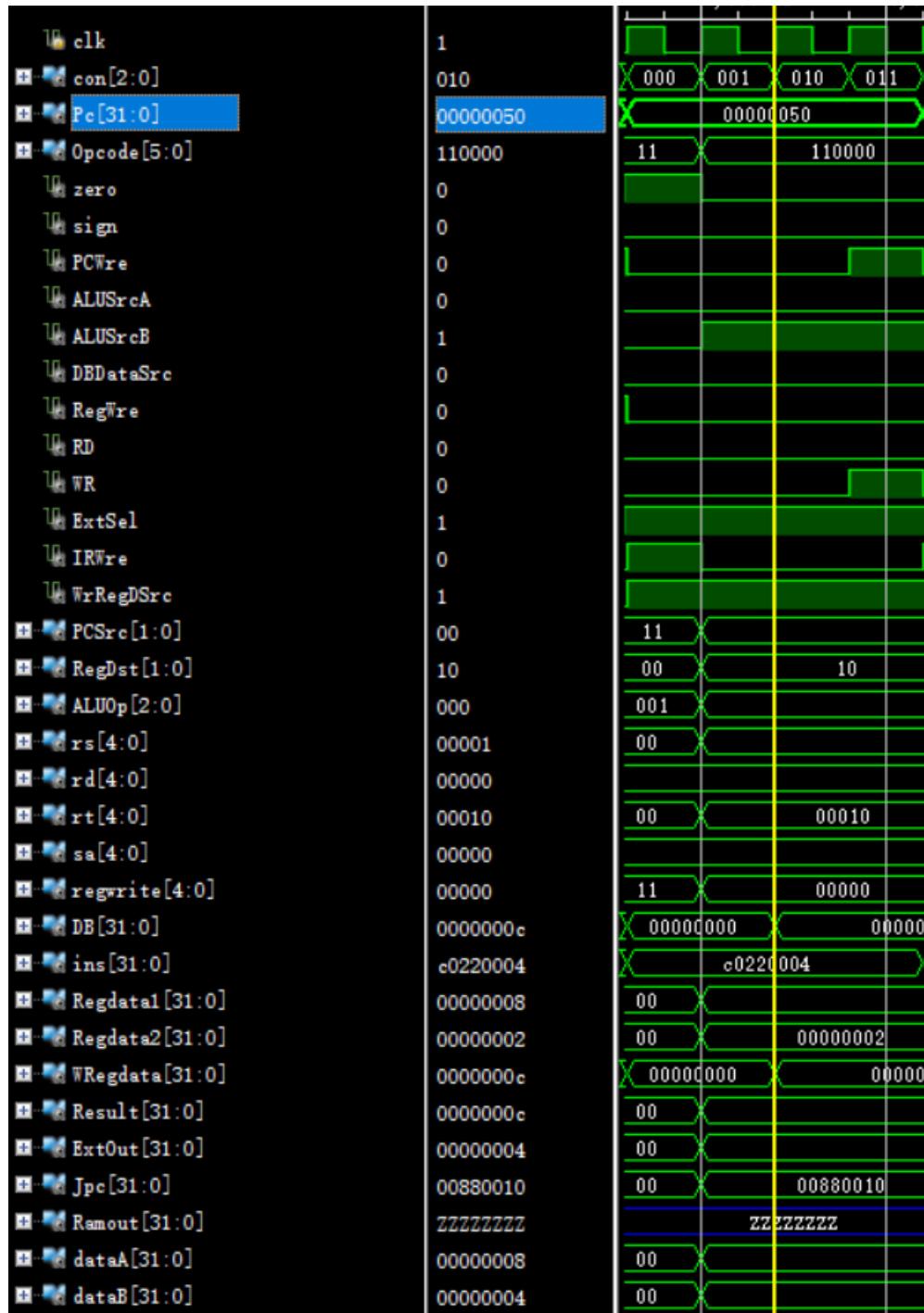
(8) jal 0x0000050



Pc=1C，执行jal 0x0000050，有两个状态，000时PC更新，001状态译码，同时指令寄存器写入指令，RegDst为00，001状态时31号寄存器写入数据，值为PC+4，立即数为20，左移两位为80，十六进制数为50，与PC的最高四位拼接得到跳转地址50，nextPC=50

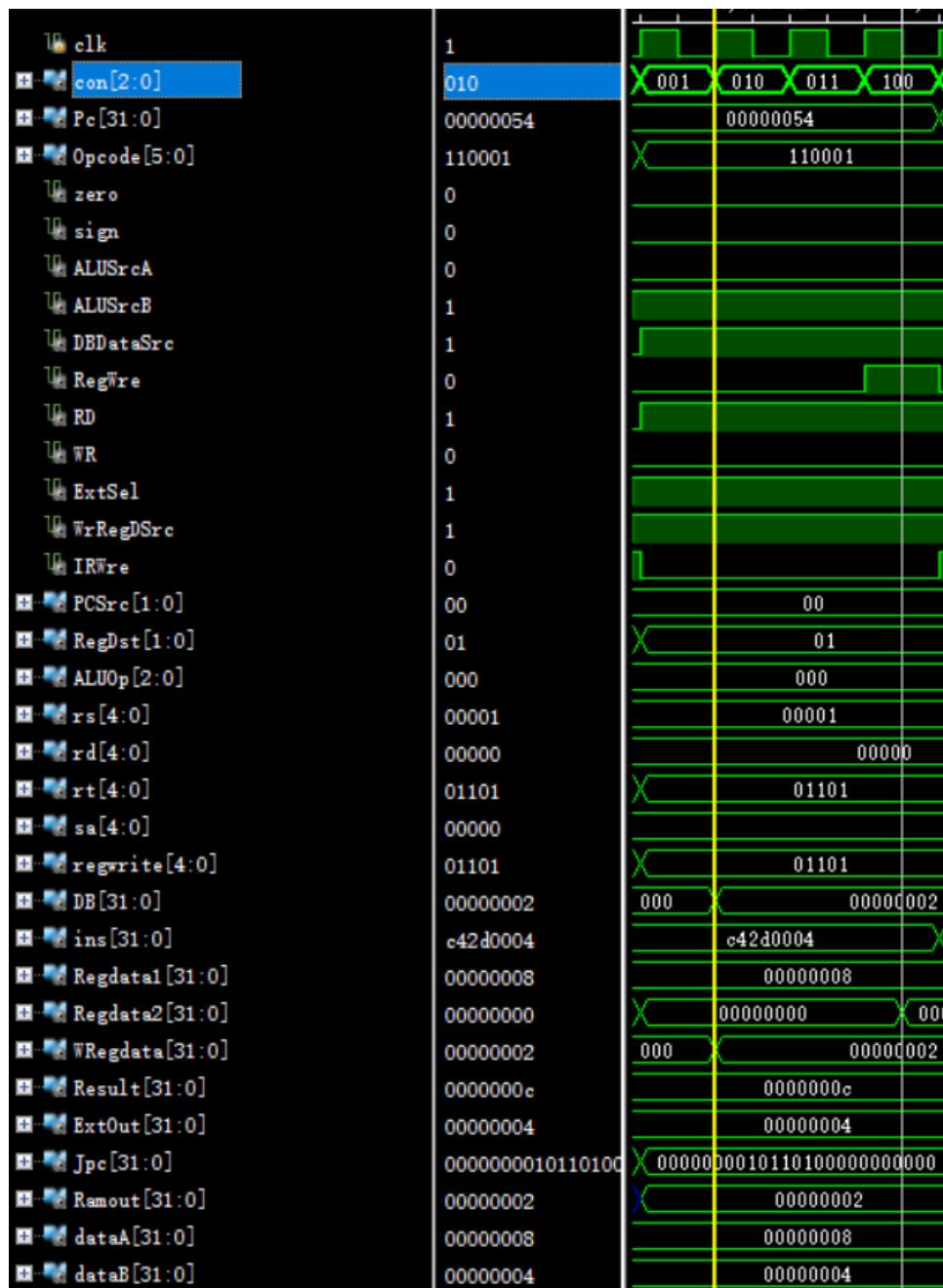
(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$31=20)

(9) sw \$2,4(\$1)



PC=50: 指令为sw \$2,4(\$1), 操作码为110000, rs=1, rt=2, 立即数为4, 存储器可写, ALU操作码为000, dataA来自\$1, DataB来自符号扩展的立即数, 执行加法操作, 结果为12, 即需要写的存储器地址为12, 值为Regdata2即\$2的值, 在011访存状态, 写入存储器中, nextPC=54。

(10) lw \$13,4(\$1)



PC=54: 指令为lw \$13,4(\$1), 操作码为110001, rs=1, rt=13, 立即数为4, 存储器可读, 寄存器可写, ALU操作码为000, DataA来自\$1, DataB来自符号扩展的立即数, 执行加法操作, 结果为12, 即需要读的存储器地址为12, 在100写回状态, 将该存储单元的值即Ramout写入\$13(regwrite)中, nextPC=58。

(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$13=2 \$31=20)

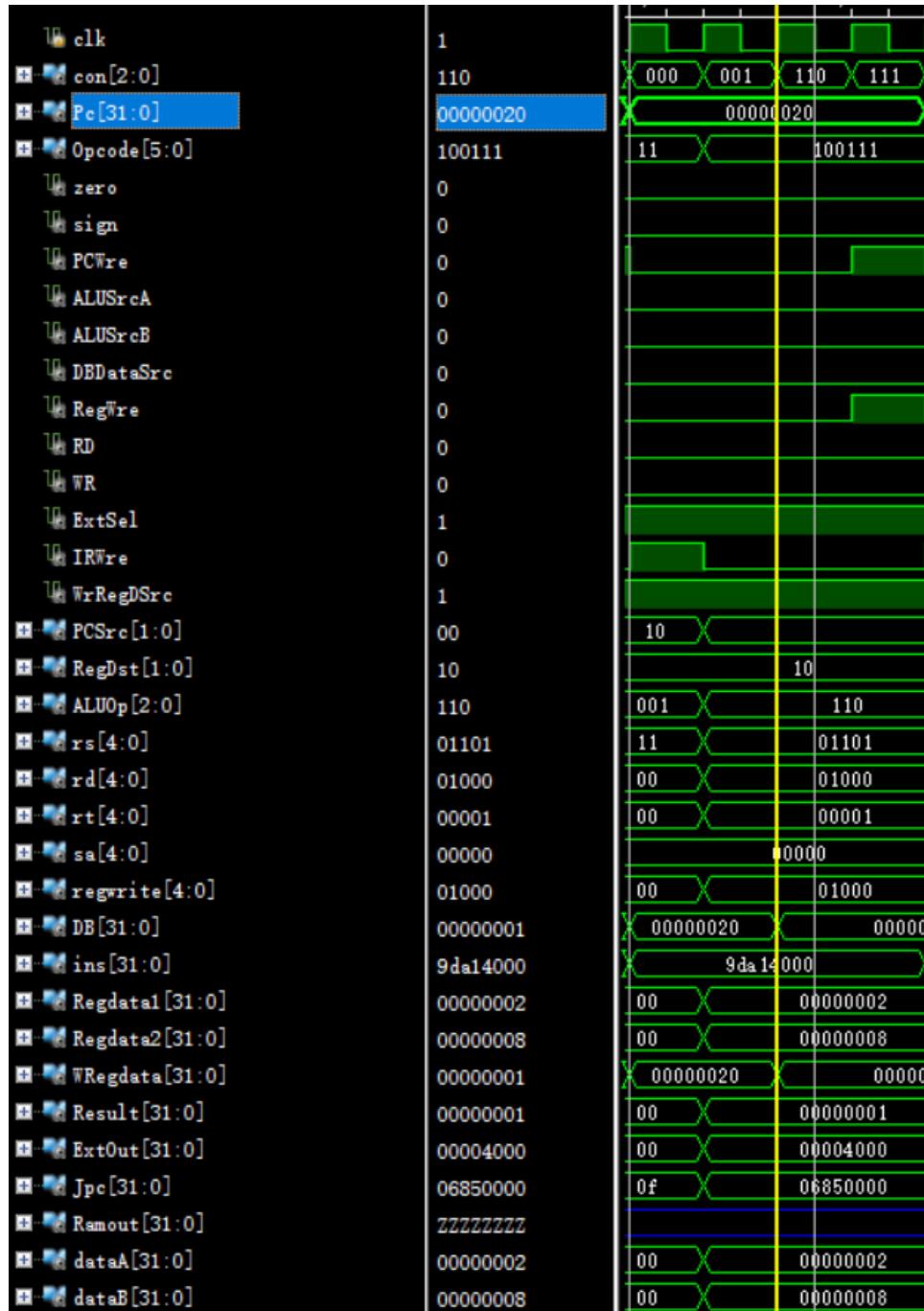
(11) jr \$31

clk	1		
con[2:0]	001	000	001
Pc[31:0]	00000058	X	00000058
Opcode[5:0]	111001	110	11100
zero	0		
sign	0		
ALUSrcA	0		
ALUSrcB	0		
DEDataSrc	0		
RegWre	0	L	
RD	0	0	
WR	0		
ExtSel	1		
WrRegDSrc	1		
IRwre	0	0	1
PCSrc[1:0]	10	00	10
RegDst[1:0]	10	01	
ALUOp[2:0]	001	000	001
rs[4:0]	11111	00001	1111
rd[4:0]	00000		00000
rt[4:0]	00000	01101	00000
sa[4:0]	00000		
regwrite[4:0]	00000	01101	00000
DB[31:0]	00000002	00000002	
ins[31:0]	e7e00000	X	e7e00000
Regdata1[31:0]	00000020	000	0000000
Regdata2[31:0]	00000000	000	0000000
WRegdata[31:0]	00000002	00000002	X
Result[31:0]	00000020	000	0000000
ExtOut[31:0]	00000000	000	0000000
Jpc[31:0]	0000111100000000	000	000011
Ramout[31:0]	ZZZZZZZ	000	
dataA[31:0]	00000020	000	0000000
dataB[31:0]	00000000	000	0000000

PC=58, 指令为jr \$31, 有两个状态, 000时PC更新, 001状态译码, PCSrc

为10, 下个地址来自rs对应的寄存器即31号寄存器, nextPC=20。

(12) slt \$8,\$13,\$1

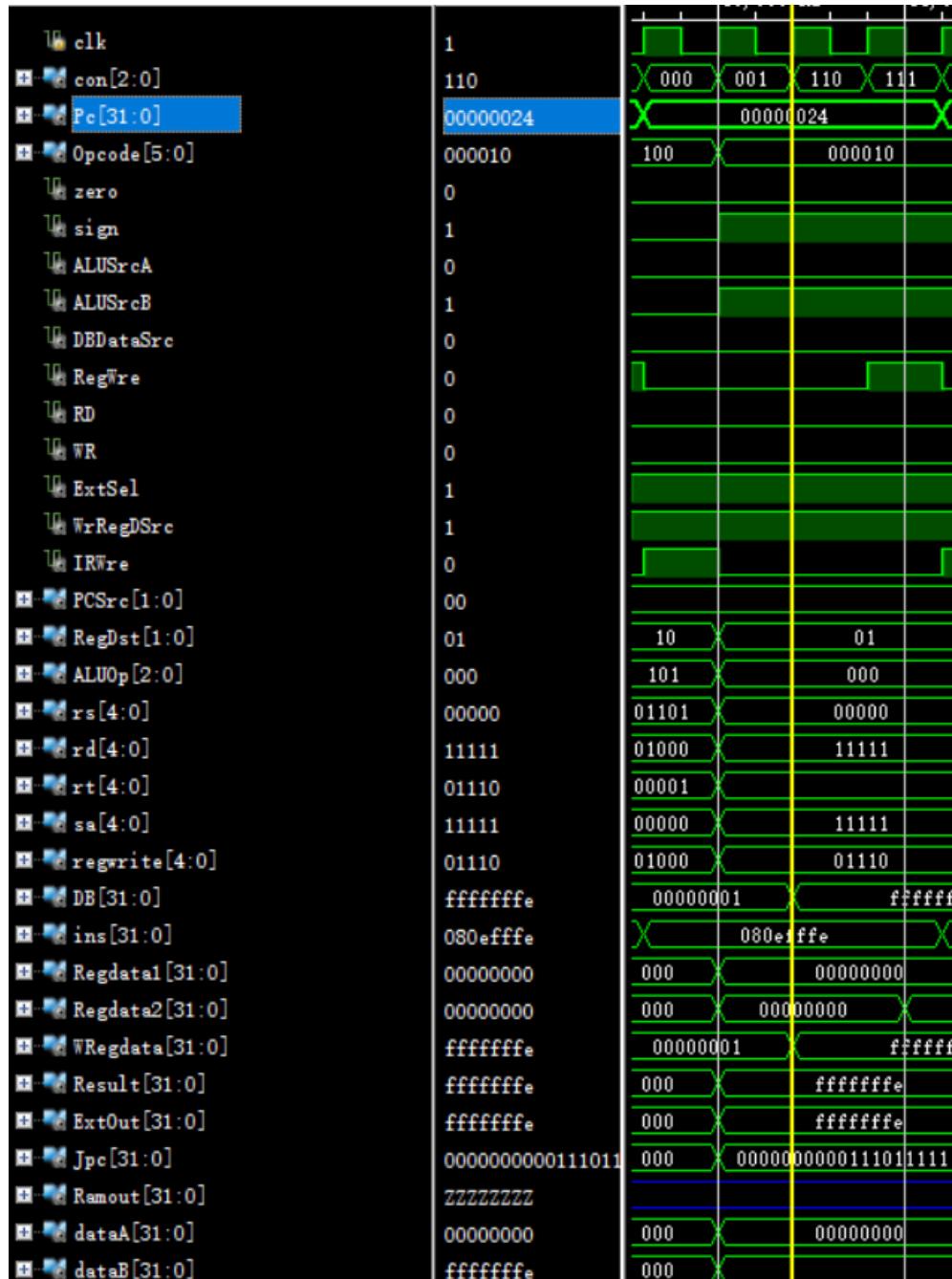


PC=20，指令为slt \$8,\$13,\$1，有四个状态，000时PC更新，111时9

号寄存器写入数据，写入寄存器的值来自ALU运算结果，ALU操作码为110，执行带符号比较，dataA来自\$13，dataB来自\$1，dataA小于dataB，结果为1，在111状态写到\$8，nextPC=24。

(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$13=2 \$31=20)

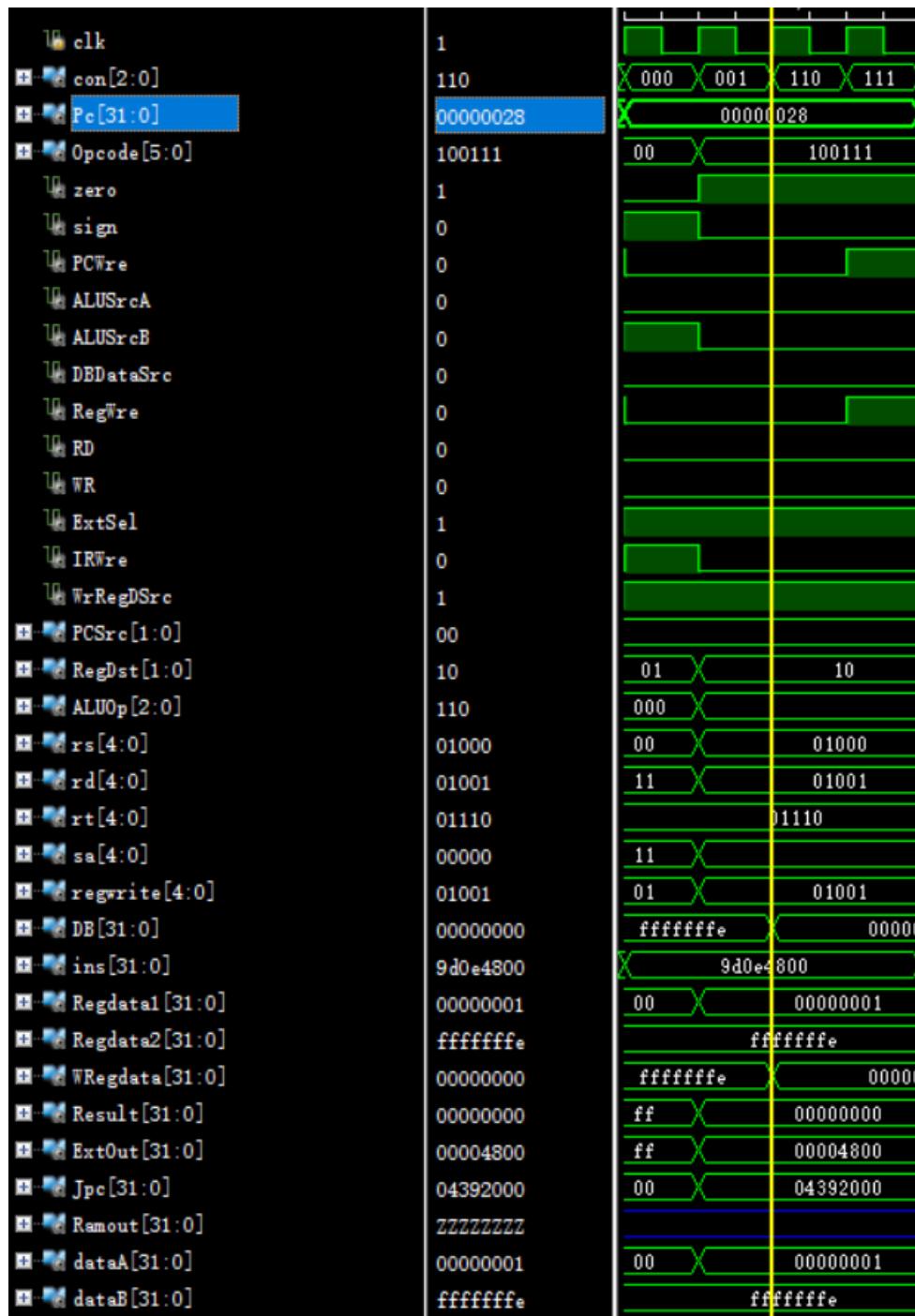
(13) addiu \$14,\$0,-2



Pc=24, 执行addiu \$14,\$0,-2。 000状态取指， 001状态译码， 操作码为000010, rt=14, rs=0, ALUOp=000。 dataA来自\$0,dataB来自立即数-2的符号扩展，在110状态进行加运算。结果为-2，在111写回状态写入rt。
PCsrc=00,nextPC=28。

(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$13=2 \$14=-2 \$31=20)

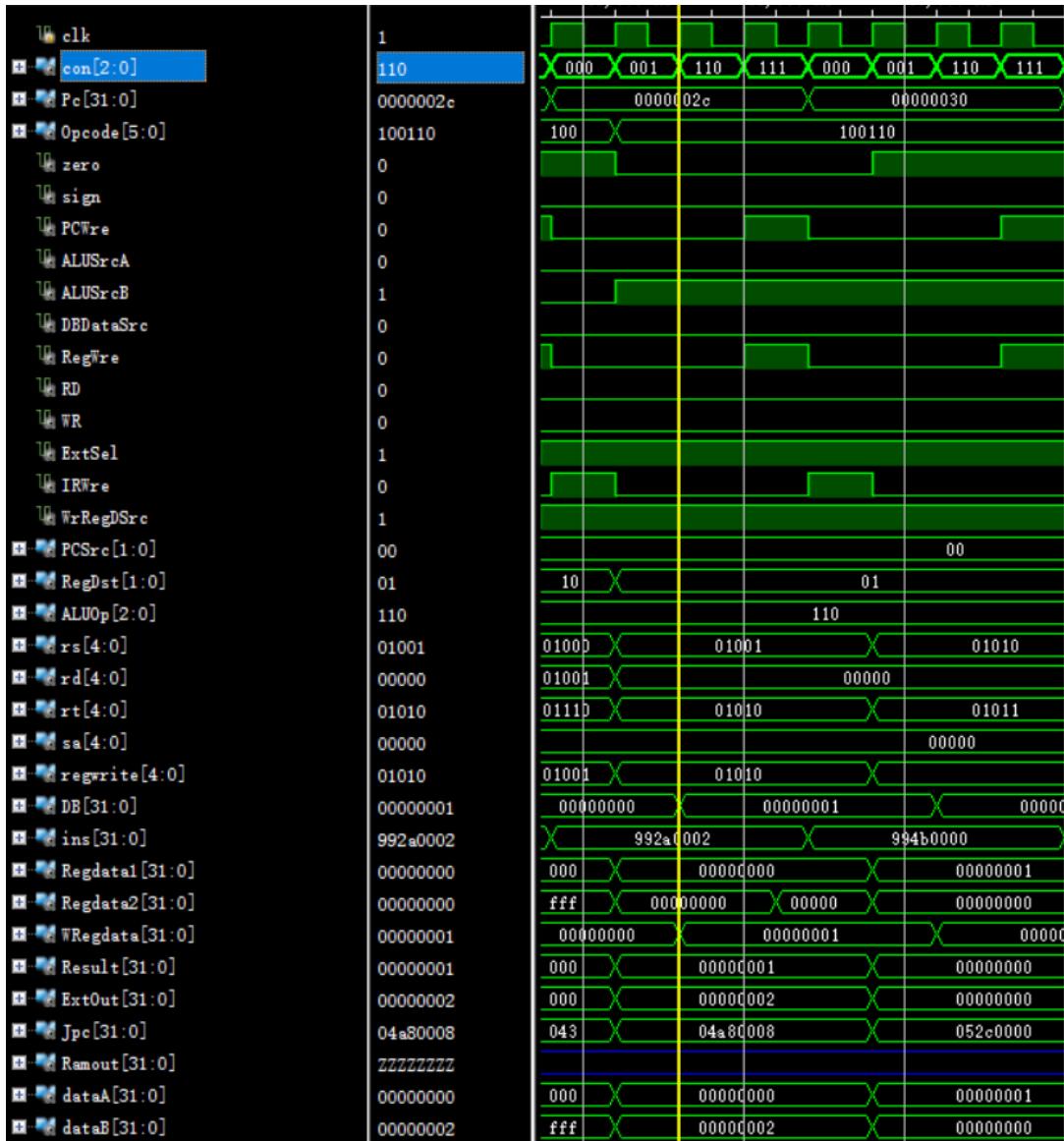
(14) slt \$9,\$8,\$14



PC=28，指令为slt \$9,\$8,\$14，有四个状态，000时PC更新，111时9号寄存器写入数据，写入寄存器的值来自ALU运算结果，ALU操作码为110，执行带符号比较，dataA来自\$8，dataB来自\$14，dataA大于dataB，结果为0，在111状态写到\$9, nextPC=2C。

(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$13=2 \$14=-2 \$31=20)

(15) slti \$10,\$9,2 slti \$11,\$10,0



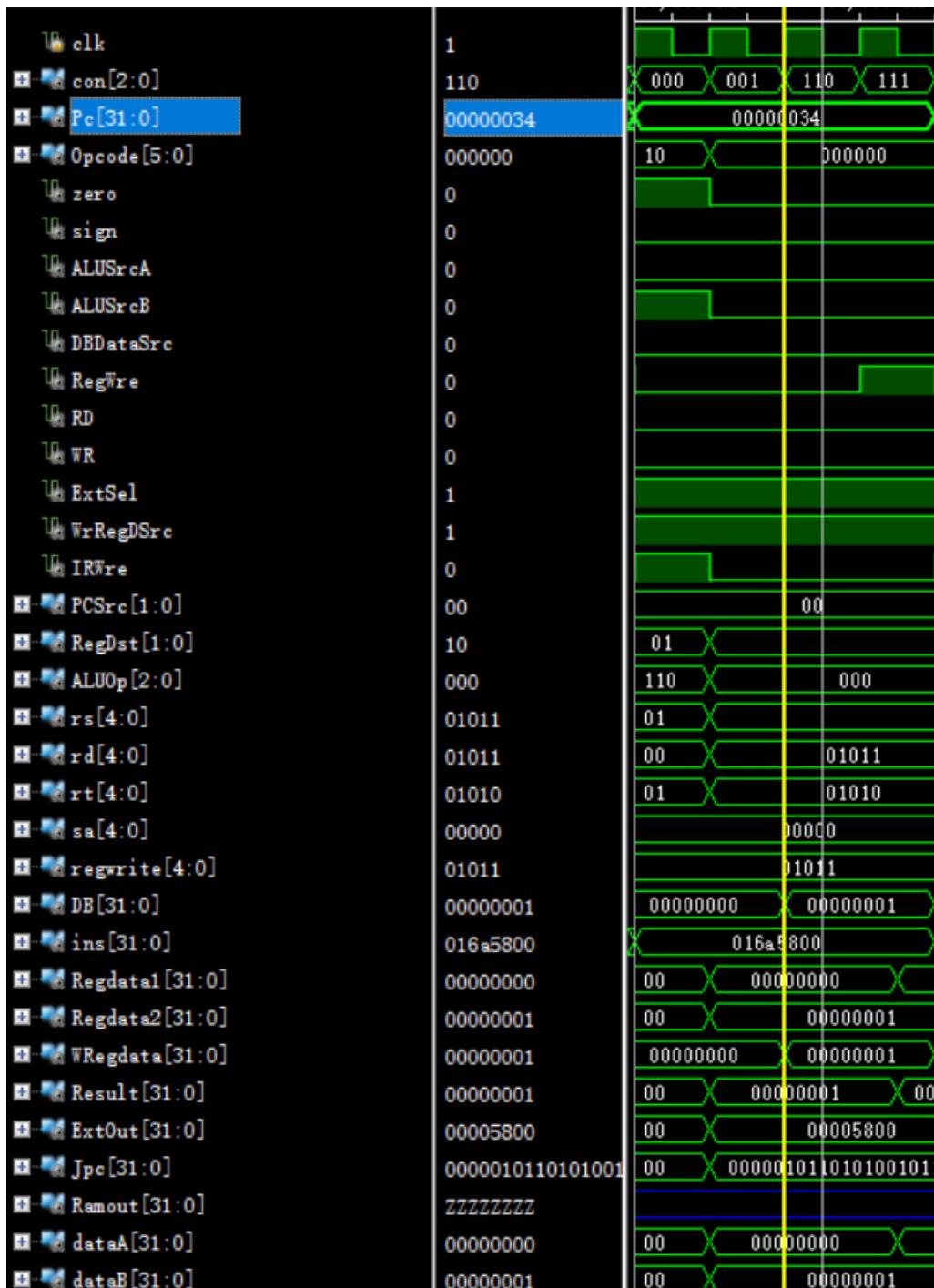
PC=2C, 指令为slti \$10,\$9,2, 有四个状态, 000时PC更新, 111时10号

寄存器写入数据, 写入寄存器的值来自ALU运算结果, ALU操作码为110, 执行带符号比较, dataA来自\$9, dataB来自立即数2的符号扩展, dataA小于dataB, 结果为1, 在111状态写到\$10, nextPC=30。

(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$13=2 \$31=20)

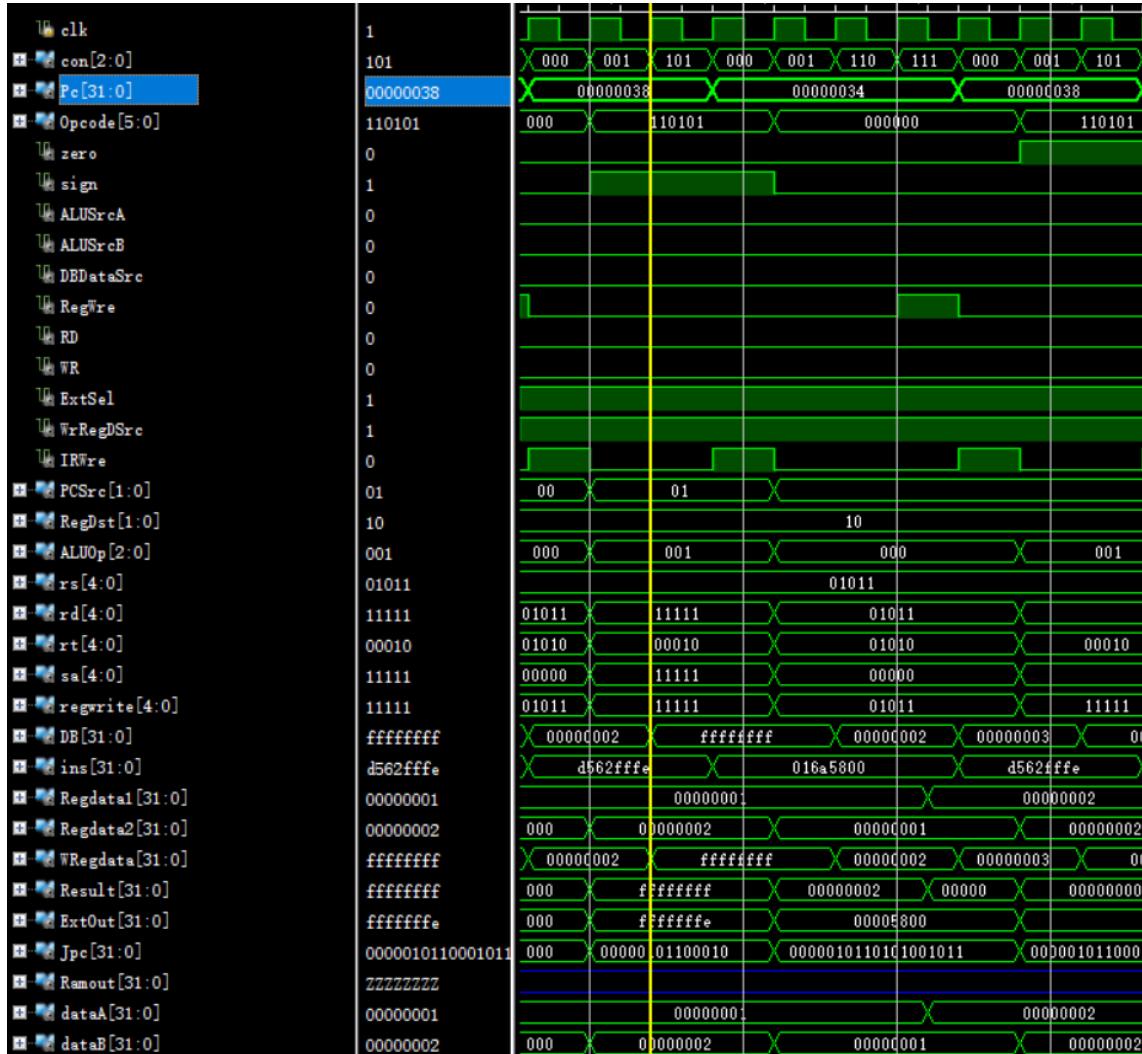
PC=30, 指令为slti \$11,\$10,0 执行过程同上, \$11=0在111状态写入, nextPC=34。 (此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=0 \$13=2 \$14=-2 \$31=20)

(16) add \$11,\$11,\$10



Pc=34, 执行add \$11,\$11,\$10。000状态取指, 001状态译码, 操作码为000000, rt=10, rs=11, rd=11, ALUOp=000。dataA来自\$11,dataB来自\$10, 在110状态进行减运算。结果为1, 在111写回状态写入rd. (此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=1 \$13=2 \$14=-2 \$31=20)。PCsrc=00,nextPC=38。

(17) bne \$11,\$2,-2 (\neq , 转34)



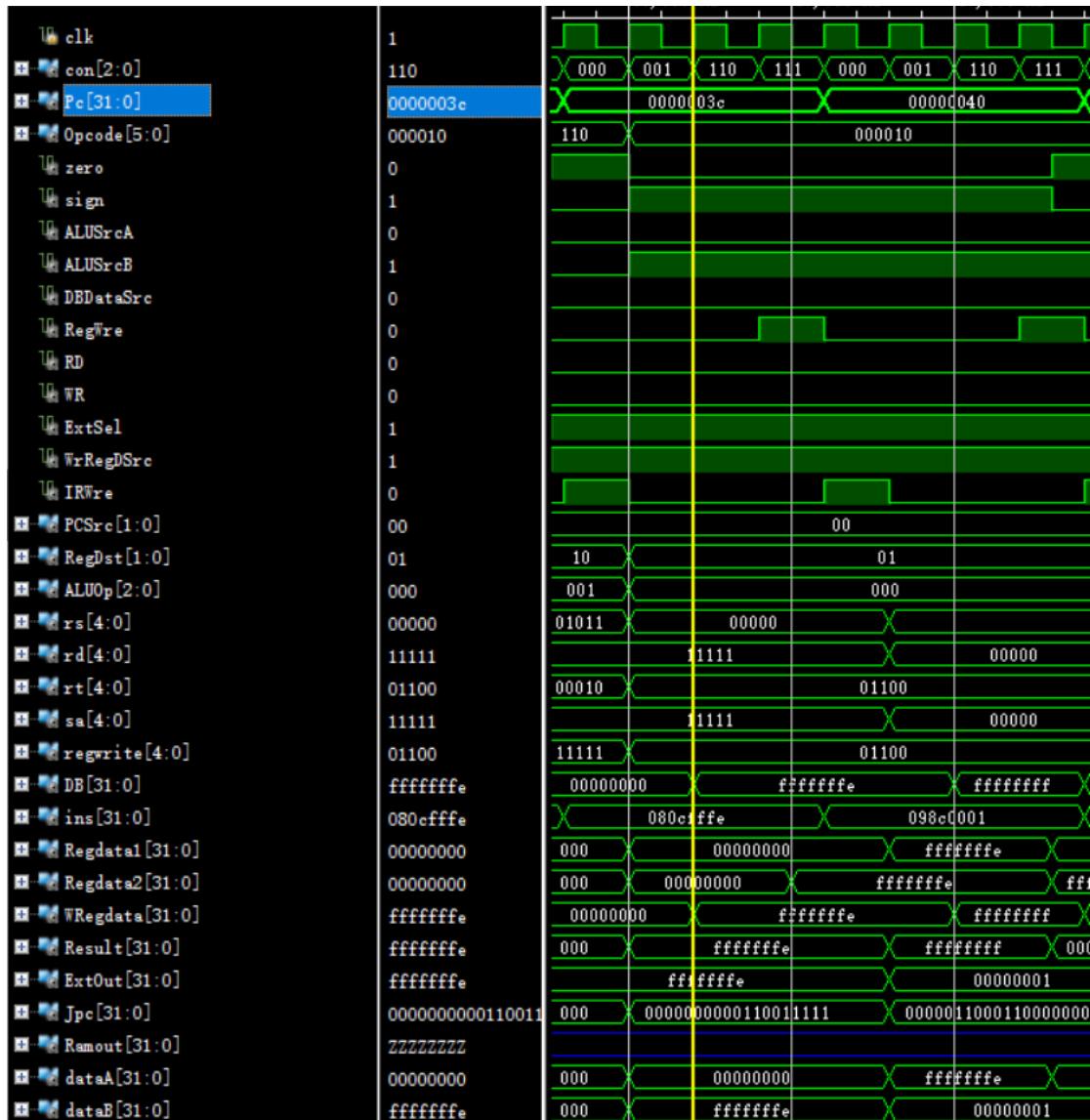
Pc=38, 执行bne \$11,\$2,-2, 000状态取指, 001状态译码, 操作码为110101, rs=11,rt=2,立即数为-2,ALUOp=001。立即数为需要跳转的指令数, 需要进行符号扩展再左移两位, dataA来自rs,dataB来自rt,在101状态进行减法运算。结果为-1。PCsrc=01, $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$, nextPC=34.

再次执行add \$11,\$11,\$10。结果为\$11=2。PCsrc=00,nextPC=38.

第二次执行bne \$11,\$2,-2。rs==rt, PCsrc=00,nextPC=3C.

(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=2 \$13=2
\$14=-2 \$31=20)

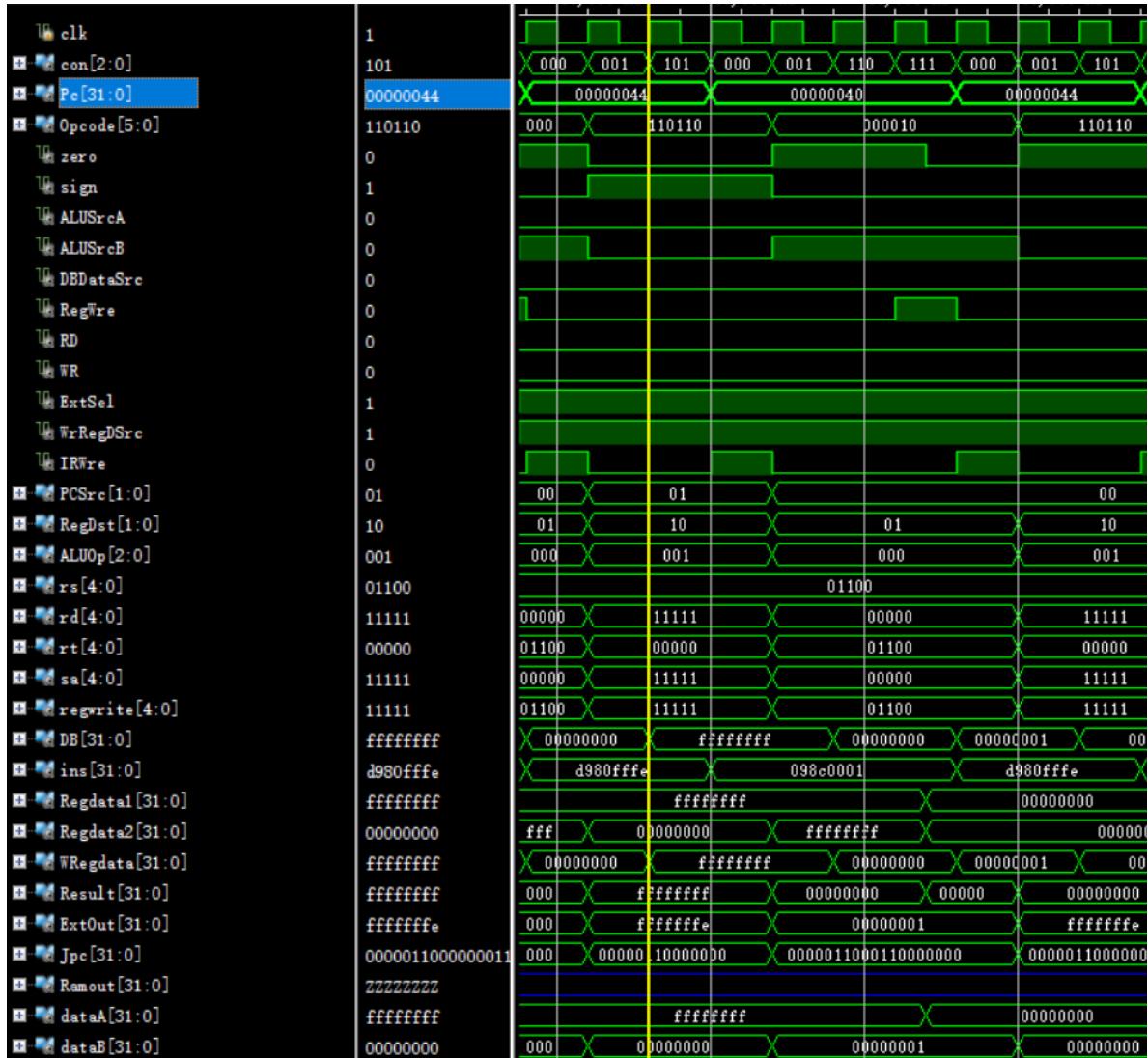
(18) addiu \$12,\$0,-2 addiu \$12,\$12,1



Pc=3C, 执行addiu \$12,\$0,-2。000状态取指, 001状态译码, 操作码为000010, rt=12, rs=0, ALUOp=000。dataA来自\$0,dataB来自立即数-2的符号扩展, 在110状态进行加运算。结果为-2, 在111写回状态写入rt。PCsrc=00,nextPC=40。(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=2 \$12=-2 \$13=2 \$14=-2 \$31=20)

Pc=40, 执行addiu \$12,\$12,1。指令执行过程同上, 结果为\$12=-1, 在111状态写入。PCsrc=00,nextPC=44。(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=2 \$12=-1 \$13=2 \$14=-2 \$31=20)

(19) bltz \$12,-2 (<0,转40)



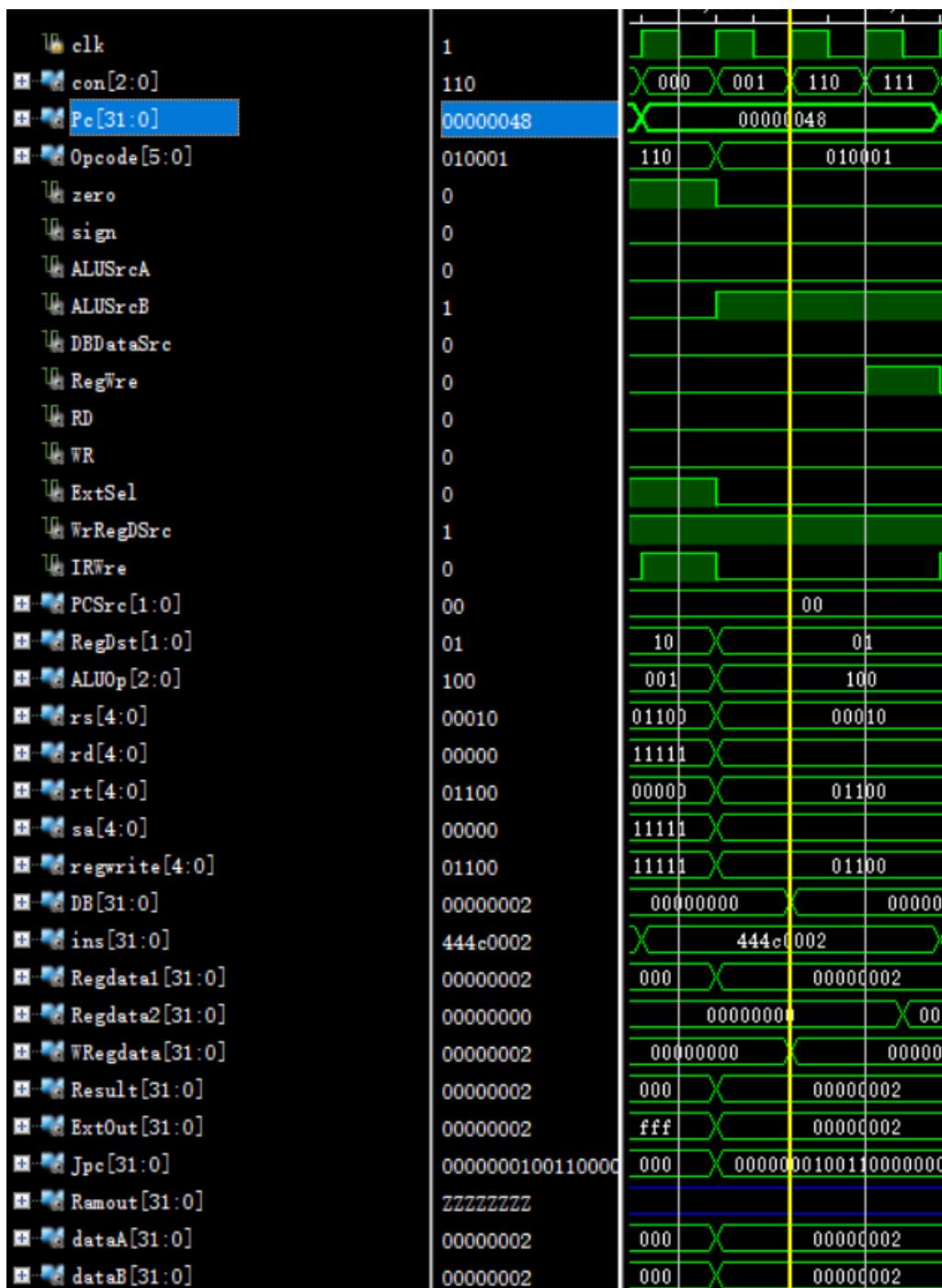
Pc=44, 执行bltz \$12,-2, 000状态取指, 001状态译码, 操作码为110110, rs=12, 立即数为-2, ALUOp=001。立即数为需要跳转的指令数, 需要进行符号扩展再左移两位, dataA来自rs, dataB来自\$0, 在101状态进行减法运算。结果为-1。PCsrc=01, $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$, nextPC=40.

再次执行addiu \$12,\$12,1。结果为\$12=0。PCsrc=00,nextPC=44.

第二次执行bltz \$12,-2。rs==0 PCsrc=00,nextPC=48.

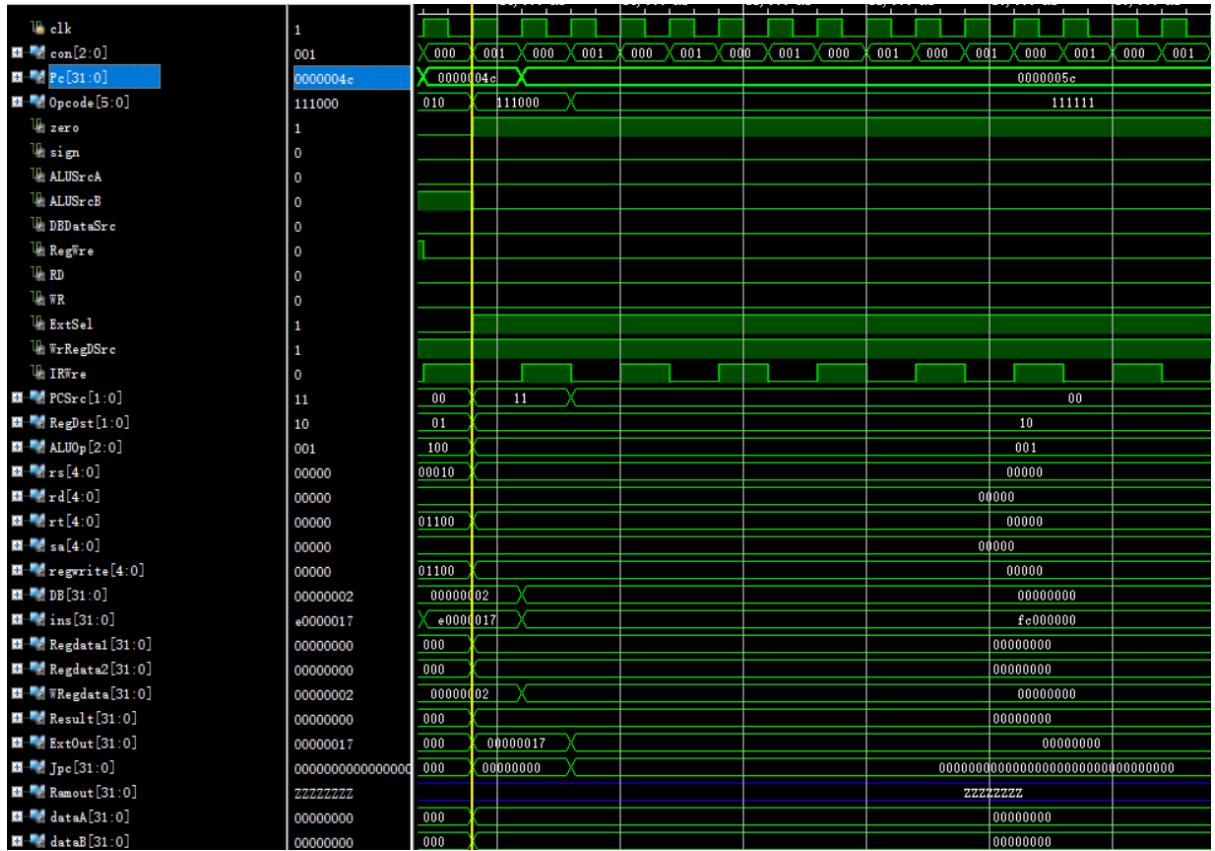
(此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=2 \$12=0
\$13=2 \$14=-2 \$31=20)

(20) andi \$12,\$2,2



Pc=48, 执行andi \$12,\$2,2。 000状态取指， 001状态译码， 操作码为010001, rt=12, rs=2, ALUOp=100。 dataA来自\$2,dataB来自立即数2的0扩展，在110状态进行与运算。结果为2，在111写回状态写入rt (此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=2 \$12=2 \$13=2 \$14=-2 \$31=20)。 PCsrc=00,nextPC=4C。

(21) j 0x000005C



功能: $pc \leftarrow \{pc+4\}[31:28], addr[27:2], 2'b00\}$, 无条件跳转。

PC=4C, 操作码为111000, 立即数为23, 将立即数左移2位, 得到92, 十六进制表示为5C, 再与PC最高四位拼接, 得到跳转地址为5C, PC的下一个值为5C。

(22) halt

PC=5C: 指令为halt, PCwre=0, PC不再变化, 停机。

此时\$1=8 \$2=2 \$3=10 \$4=2 \$5=32 \$8=1 \$9=0 \$10=1 \$11=2 \$12=2 \$13=2 \$14=-2
\$31=20 结合下图检验, 结果正确。

(结束时各个寄存器内数值如下)

reg0[31:0]	00000000
reg1[31:0]	00000008
reg2[31:0]	00000002
reg3[31:0]	0000000a
reg4[31:0]	00000002
reg5[31:0]	00000020
reg8[31:0]	00000001
reg9[31:0]	00000000
reg10[31:0]	00000001
reg11[31:0]	00000002
reg12[31:0]	00000002
reg13[31:0]	00000002
reg14[31:0]	ffffffe
reg31[31:0]	00000020

4.多周期CPU实现

(1)实现思路及代码

分频器模块，利用计数器计数，得到190Hz时钟信号用于驱动数码管和消抖采样。

```
module CLK_DIV(
    input clk,
    output clk_190hz
);
reg [17:0] q;
always @ (posedge clk)
begin
    q<=q+1;
end
assign clk_190hz = q[17];
endmodule
```

按键消抖模块，依据时钟信号采样，三个样点一致时，可视为信号稳定。

```
module debouncing(clk,key_in,key_out);
input clk;
input key_in;
output key_out;
reg dout1,dout2,dout3;
assign key_out = dout1 | dout2 | dout3;
always@(posedge clk)
begin
    dout1<=key_in;
    dout2<=dout1;
    dout3<=dout2;
end
endmodule
```

数据选择模块，选择需要显示的数据，四个数码管最多显示四个十六进制数，也就是16位，将显示数据按位赋值给data，前八位与后八位分别对应一个数据。

```
module show( Sel, PC, NextPC ,rs, ReadData1, rt, ReadData2, Result, DB, Out);
input [1:0] Sel;
input [4:0] rs,rt;
input [31:0] PC, NextPC, ReadData1, ReadData2, Result, DB;
output reg [15:0] Out;
always @ (*)begin
    case(Sel)
        0: begin
            Out[15:8] = PC[7:0];
```

```

        Out[7:0] = NextPC[7:0];
    end
    1: begin
        Out[15:8] = {3'b000,rs[4:0]};
        Out[7:0] = ReadData1[7:0];
    end
    2: begin
        Out[15:8] = {3'b000,rt[4:0]};
        Out[7:0] = ReadData2[7:0];
    end
    3:begin
        Out[15:8] = Result[7:0];
        Out[7:0] = DB[7:0];
    end
endcase
end
endmodule

```

数码管显示模块，使用滚动扫描的方法，将输入的16位信号显示出来，每个时钟周期内只有一个数码管亮，时钟周期合适的话，利用视觉残留，实现同时显示的效果，每个显示的16进制数都要译码，得到驱动每一段数码管的信号。

```

module Hex7seg(
    input wire [15:0] data,
    input clk,
    output reg [6:0] a_to_g,
    output reg [3:0] dig
);
    reg [1:0] sel;
    reg [3:0] num;
    initial
        sel = 0;
    always @ (posedge clk )
    begin
        sel = sel+1;
    end
    always @ (sel)
    begin
        case(sel)
            0: num = data[3:0];
            1: num = data[7:4];
            2: num = data[11:8];
            3: num = data[15:12];
            default: num = data[3:0];
        endcase
    end
endmodule

```

```

end
always @ (sel)
begin
case(sel)
    0: dig = 4'b0111;
    1: dig = 4'b1011;
    2: dig = 4'b1101;
    3: dig = 4'b1110;
endcase
end
always @ (num)
begin
    case(num)
        0: a_to_g = 7'b0000001;
        1: a_to_g = 7'b1001111;
        2: a_to_g = 7'b0010010;
        3: a_to_g = 7'b0000110;
        4: a_to_g = 7'b1001100;
        5: a_to_g = 7'b0100100;
        6: a_to_g = 7'b0100000;
        7: a_to_g = 7'b0001111;
        8: a_to_g = 7'b0000000;
        9: a_to_g = 7'b0000100;
        10:a_to_g = 7'b0001000;
        11:a_to_g = 7'b1100000;
        12:a_to_g = 7'b0110001;
        13:a_to_g = 7'b1000010;
        14:a_to_g = 7'b0110000;
        15:a_to_g = 7'b0111000;
    default: a_to_g = 7'b1111111;
endcase
end
endmodule

```

实现文件顶层模块，将原顶层模块和在数码管显示模块实例化并组合。

```

module imptop(
    input clk,
    input step,
    input Reset,
    input [1:0] Sel,
    output [6:0] atog,
    output [3:0] enlight
);
    wire clk190hz,key_out,zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre,

```

```

InsMemRW, RD, WR, ExtSel, IRWre, WrRegDSrc;
    wire [1:0] PCSrc, RegDst;
    wire [2:0] ALUOp, con, nextcon;
    wire [4:0] rs, rd, rt, sa, regwrite;
    wire [5:0] Opcode;
    wire [15:0] data;
    wire [31:0] DB, Pc, nextPC, ins, Regdata1, Regdata2, WRegdata, Result, ExtOut, Jpc,
Ramout, dataA, dataB, reg0, reg1, reg2, reg3, reg4, reg5, reg8, reg9, reg10, reg11, reg12, reg13,
reg14, reg31;
    CLK_DIV div(clk, clk190hz);
    debouncing key(clk190hz, step, key_out);
    show out(Sel, Pc, nextPC, rs, Regdata1, rt, Regdata2, Result, WRegdata, data);
    Hex7seg hex(data, clk190hz, atog, enlight);
    CPU_top cputop(key_out, Reset, zero, sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc,
RegWre, InsMemRW, RD, WR, ExtSel, IRWre, WrRegDSrc, PCSrc, RegDst, ALUOp, con, nextcon,
rs, rd, rt, sa, regwrite, Opcode, DB, Pc, nextPC, ins, Regdata1, Regdata2, WRegdata, Result,
ExtOut, Jpc, Ramout, dataA, dataB, reg0, reg1, reg2, reg3, reg4, reg5, reg8, reg9, reg10, reg11,
reg12, reg13, reg14, reg31);
endmodule

```

(2)结果

前六条指令结果如下：

(图片从左到右依次为：当前指令地址PC:下条指令地址PC；RS寄存器地址:RS寄存器数据；RT寄存器地址:RT寄存器数据；ALU结果输出 :DB总线数据)

(注：除特殊说明外，以下图片均为写回状态时拍摄)

(1)addiu \$1,\$0,8:



Pc=00,执行addiu \$1,\$0,8。000状态取指，001状态译码（写入IR），操作码为000010，rt=1，rs=0，ALUOp=000。dataA来自\$0,dataB来自立即数8的符号扩展，在110状态进行加运算。结果为8，在111写回状态写入rt (\$1=8)。PCsrc=00,nextPC=04。

(2)ori \$2,\$0,2:



Pc=04,执行ori \$2,\$0,2。000状态取指，001状态译码，操作码为010010，rt=2，rs=0，ALUOp=011。dataA来自\$0,dataB来自立即数2的零扩展，在110状态进行或运算。结果为2，在111写回状态写入rt (\$2=2)。PCsrc=00,nextPC=08。

(3)xori \$3,\$2,8:



Pc=08,执行xori \$3,\$2,8。000状态取指，001状态译码，操作码为010011，rt=3，

rs=2, ALUOp=111。dataA来自\$2,dataB来自立即数8的零扩展，在110状态进行异或运算。结果为10，在111写回状态写入rt (\$3=10)。PCsrc=00,nextPC=0C。

(4)sub \$4,\$3,\$1:



Pc=0C,执行sub \$4,\$3,\$1。000状态取指, 001状态译码, 操作码为000001, rt=1, rs=3, rd=4, ALUOp=001。dataA来自\$3,dataB来自\$1, 在110状态进行减运算。结果为2, 在111写回状态写入rd (\$4=2)。PCsrc=00,nextPC=10。

(5)and \$5,\$4,\$2:



Pc=10,执行and \$5,\$4,\$2。000状态取指, 001状态译码, 操作码为010000, rt=2, rs=4, rd=5, ALUOp=100。dataA来自\$4,dataB来自\$2, 在110状态进行与运算。结果为2, 在111写回状态写入rd (\$5=2)。PCsrc=00,nextPC=14。

(6)sll \$5,\$5,2:



Pc=14,执行sll \$5,\$5,2。000状态取指, 001状态译码, 操作码为011000, rt=5, rd=5, ALUOp=010。dataA来自移位数2,dataB来自\$5, 在110状态进行左移运算。结果为8, 在111写回状态写入rd (\$5=8)。PCsrc=00,nextPC=18。(最后一张图0808为执行阶段拍摄)

六、实验心得

在实验中遇到的问题及解决方法:

在实验中遇到了大大小小的许多问题，下面仅列举两个较大的问题。

第一个问题是时钟触发的协调，由于多个模块都使用时钟触发，并且有一定关联关系，所以存在竞争与冒险。起初PCWre信号在000状态时置为1，其余状态均为0。会出现PC会延迟到译码状态（001）时才更改，出现了延迟。

经分析，D触发器和PC均在时钟上升沿触发，在第一个时钟上升沿到达后，状态转变为000，PCWre=1，在遇到上升沿的同时，PC模块需判断PCWre是否为1，此时由于延迟，PCWre仍未0，所以在000状态PC未能及时转换。

解决方案1：将状态与PC变换错开。首先确定PC使用上升沿触发，而IR即指令寄存器要等PC更新后再存储指令，因此使用下降沿触发。控制单元中的D触发器改为下降沿触发，PC更新是在状态000的中间。

效果如图：



解决方案2：PCWre提前设置。这也是本实验最终采用的方案（采用本方案的原因是：状态与PC对齐更符合现实情况，而且便于烧板时观察各个状态的变化）。解决方式是在000状态的前一个状态将PCWre置为1，这样在下一个时钟上升沿到达时，即000状态，PC改变

第二个问题，sw和lw错误。lw时Ramout显示XXX。经分析，是控制单元中控制信号没能及时更改。也是因为前一个问题，导致Opcode没能即使改变，从而影响sw和lw。

解决方法：在控制单元中，将PC添加至敏感信号中，这样可以依据PC来确定mRD和mWR的值，解决了延时问题。

感悟及收获：

本次实验花费了将近三周时间，比单周期CPU设计难度更高。虽然大部分代码是从单周期CPU直接引用过来，再做改动，但是控制单元，D触发器，状态生成这些模块的难度较大。

我体会到，多周期CPU相对于单周期CPU来说，最显著的不同就是在多周期CPU中一条指令的执行不在局限于一个时钟周期，而是根据时钟周期划分为多个执行过程，并通过时钟触发来进行状态之间的转移（D触发器）。因此，设计多周期CPU时就不再是考虑一条指令在时钟周期里怎么执行，而是要考虑在第几个时钟周期（也即哪个状态）、改变什么信号、执行那个任务的问题。这也是我遇到困难最多的地方。

仿真过程中，和单周期实验时一样，波形出了较多错误。这让我进一步巩固了通过波形图debug的方法。即在波形图中找出与预期值不同的控制信号，返回顶层模块中，通过实例化语句，找出该控制信号对应的产生模块和作为参数使用的模块，这样可以较快地确定bug位置，节省时间。

vivado烧板过程也遇到了一些问题。主要问题是实验板上的调试较为麻烦，需要根据当前指令及其当前状态同波形图一一比对，需要反复按按钮进行找BUG、调试，才能发现问题出在哪里并想办法解决。

虽然实验中遇到的问题和困难很多，但是确实从中学到了不少知识。这次试验进一步加深了我对多周期CPU理论知识的掌握，又让我巩固了Verilog语言。在一点一点地debug，最终完成实验后，我收获了很大的成就感和满足感。

七、课程总结

不知不觉就到了学期末，计算机组成原理实验课也即将结束。对我来说，这学期的三次大实验一次比一次有挑战，需要花费的时间也很长。实验过程很痛苦，但我从中学到了不少知识。通过实验，我对计算机组成原理的理论知识有了更深刻的理解，也学会将理论与实际结合，能够完成简单的单周期和多周期CPU设计，同时也初步掌握了MIPS汇编语言和Verilog语言。

第一次的汇编程序设计实验让我更好地理解了MIPS这一汇编语言。结合老师给的参考资料，通过冒泡排序的汇编语言实现，我对每条指令的功能有了深入的认识。这使我对计算机组成原理理论课知识的第二章有了更清晰的理解，并帮助我掌握了汇编程序基本的设计思想，对程序的具体执行过程以及数据的存储与寻址有了深刻的理解。

单周期CPU设计实验让我初步了解了计算机硬件的运行方式，明白了各个硬件如何相互连接共同完成指令的执行。同时，我也熟悉了Verilog硬件描述语言，学会灵活运用真值表，并掌握了模块化的设计方法：自顶向下，将复杂的功能划分为简单的功能来实现，之后在顶层模块调用各个模块，构成一个整体。通过烧板，我对电路和硬件的理解也更进一步，并将上学期的数字电路实验学到的知识运用了起来。

多周期CPU设计实验让我清晰地领会到指令的分段执行过程。而且，我对CPU的理解相较于完成单周期CPU的时候又更为深入了。多周期CPU中每条指令的执行过程有着不同的阶段，而一个时钟周期对应一个阶段。多周期CPU的设计比单周期CPU设计更具挑战，例如：时钟触发可能会带来各种各样的问题。不仅如此，这次试验更让我重视了竞争与冒险，在组合逻辑电路和时序逻辑电路的设计中要格外注意。

总之，对我来说，这门实验课十分具有挑战性，让我收获了许多。在无数次的DEBUG过程中，我不仅仅学会了如何发现并解决问题，更学会了耐心、细心地去编程，逐渐拥有了强大地心理素质。更重要的是，通过实验课，我对理论课学到的知识有了更深入的体会与认识，实现了理论与实践相结合的学习方式。

最后，向老师及几位助教表示衷心的感谢，你们辛苦了。