



# 《计算机组成原理与接口技术实验》

## 实验报告

(实验三)

学院名称 : 数据科学与计算机学院

专业(班级) : 16 软件工程三 (6) 班

学生姓名 : 吴聪

学号 : 16340237

时间 : 2018 年 6 月 21 日

# 成 绩 :

---

## 实验三：多周期CPU设计与实现

---

### 一. 实验目的

- (1) 认识和掌握多周期数据通路原理及其设计方法；
- (2) 掌握多周期 CPU 的实现方法，代码实现方法；
- (3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
- (4) 掌握多周期 CPU 的测试方法；
- (5) 掌握多周期 CPU 的实现方法。

### 二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：**(说明：操作码按照以下规定使用，都给每类指令预留扩展空间，后续实验相同。)**

#### **==>算术运算指令**

- (1) add rd, rs, rt

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

功能:  $rd \leftarrow -rs + rt$

- (2) sub rd, rs, rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

完成功能:  $rd \leftarrow -rs - rt$

- (3) addi rt, rs, immediate

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

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

#### **==>逻辑运算指令**

- (4) or rd, rs, rt

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

功能:  $rd \leftarrow -rs | rt$

- (5) and rd, rs, rt

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

功能:  $rd \leftarrow -rs \& rt$

- (6) ori rt, rs, immediate

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

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

#### **==>移位指令**

- (7) sll rd, rt, sa

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

功能:  $rd \leftarrow rt \ll (zero\text{-extend})sa$ , 左移 sa 位, (zero-extend)sa

### ==>比较指令

(8) slt rd, rs, rt 带符号数

100110	rs(5位)	rt(5位)	rd(5位)	reserved
--------	--------	--------	--------	----------

功能: if ( $rs < rt$ )  $rd = 1$  else  $rd = 0$ , 具体请看表 2 ALU 运算功能表, 带符号

(9) sltiu rt, rs, immediate 不带符号

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

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

### ==>存储器读写指令

(10) sw rt, immediate(rs)

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

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

(11) lw rt, immediate(rs)

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

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

### ==>分支指令

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

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

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

(13) 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

(14) 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。

(15) jr rs

111001	rs(5位)	未用	未用	reserved
--------	--------	----	----	----------

功能:  $pc \leftarrow rs$ , 跳转。

**==>调用子程序指令**

(16) jal addr

111010	addr[27:2]
--------	------------

功能：调用子程序， $pc \leftarrow \{(pc+4)[31:28],addr[27:2],2'b00\}$ ;  $\$31 \leftarrow pc+4$ ，返回地址设置；子程序返回，需用指令  $jr \$31$ 。跳转地址的形成同  $j addr$  指令。

**==>停机指令**

(17) halt (停机指令)

111111	00000000000000000000000000(26 位)
--------	----------------------------------

不改变  $pc$  的值， $pc$  保持不变。

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

### 三. 实验原理

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

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

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

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

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

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

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

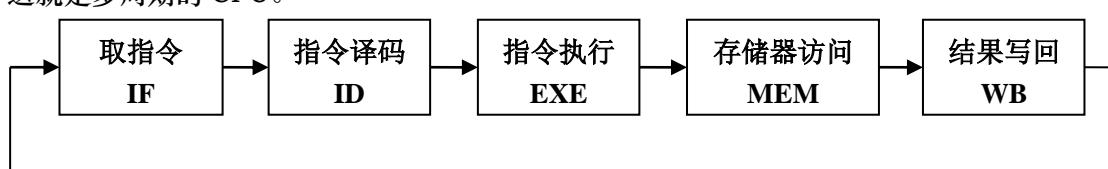


图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式:

### R 类型:

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

6 位      5 位      5 位      5 位      5 位      6 位

### I 类型:

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

6 位      5 位      5 位      16 位

### J 类型:

31	26 25	0
op	address	

6 位                          26 位

其中，

**op:** 为操作码；

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

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

**rd:** 为目的操作数寄存器，寄存器地址（同上）；

**sa:** 为位移量 (shift amt)，移位指令用于指定移多少位；

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

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

**address:** 为地址。

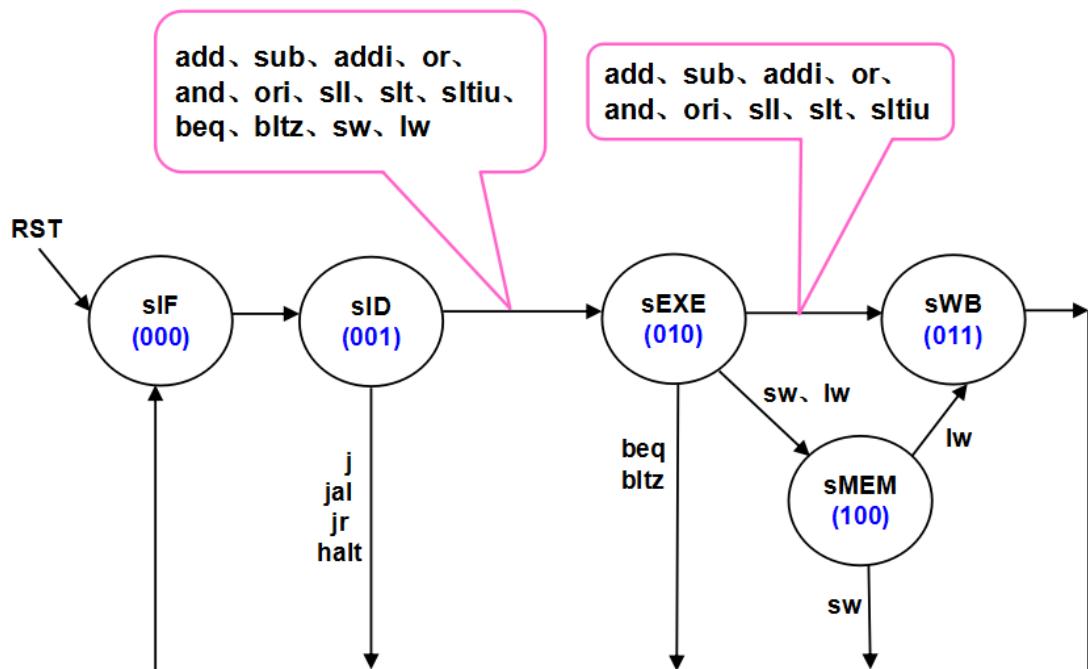


图 2 多周期 CPU 状态转移图

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

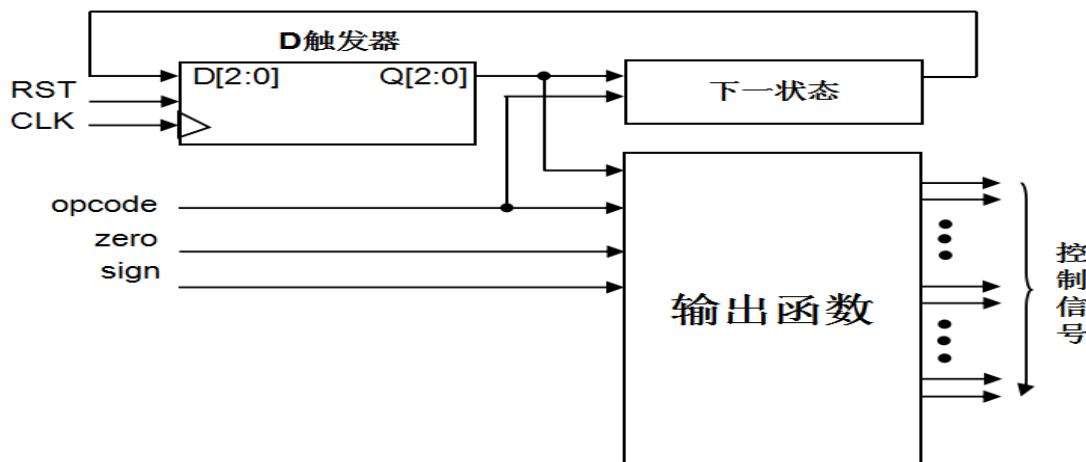


图 3 多周期 CPU 控制部件的原理结构图

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

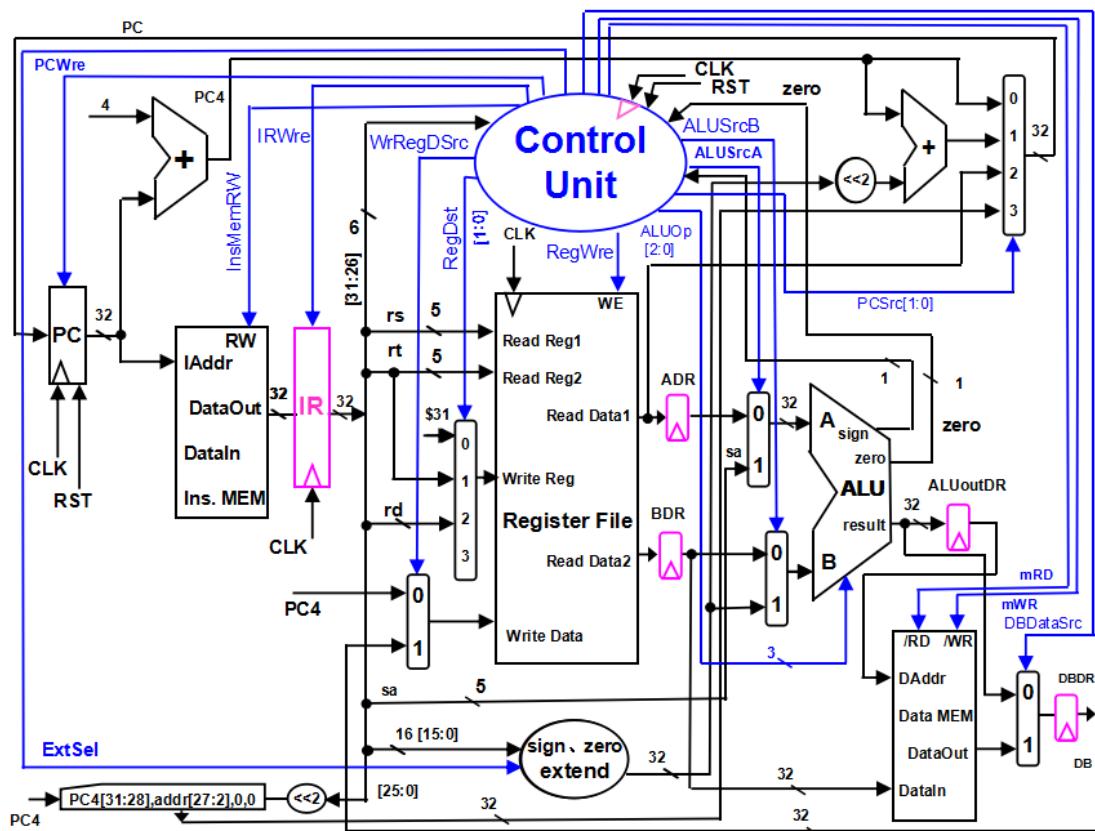


图 4 多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器不需要写使能信号，其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1 控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bltz、slt、sltiu、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 $\{27\{1'b0\}, sa\}$ ，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指	来自 sign 或 zero 扩展的立即数，相关

	令: add、sub、or、and、beq、bltz、slt、sll	指令: addi、ori、sltiu、lw、sw
DBDataSrc	来自 ALU 运算结果的输出, 相关指令: add、sub、addi、or、and、ori、slt、sltiu、sll	来自数据存储器 (Data MEM) 的输出, 相关指令: lw
RegWre	无写寄存器组寄存器, 相关指令: beq、bltz、j、sw、jr、halt	寄存器组寄存器写使能, 相关指令: add、sub、addi、or、and、ori、slt、sltiu、sll、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4) , 相关指令: jal, 写 \$31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据, 相关指令: add、addi、sub、or、and、ori、slt、sltiu、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	读数据存储器, 相关指令: lw
mWR	无操作	写数据存储器, 相关指令: sw
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后, 这个信号也接着发出, 在时钟上升沿, IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)immediate, 相关指令: ori、sltiu;	(sign-extend)immediate, 相关指令: addi、lw、sw、beq、bltz;
PCSrc[1:0]	00: pc<-pc+4, 相关指令: add、addi、sub、or、ori、and、slt、sltiu、sll、sw、lw、beq(zero=0)、bltz(sign=0, 或 zero=1); 01: pc<-pc+4+(sign-extend)immediate, 相关指令: beq(zero=1)、bltz(sign=1, zero=0); 10: pc<-rs, 相关指令: jr; 11: pc<-(pc+4)[31:28],addr[27:2],2'b00}, 相关指令: j、jal;	
RegDst[1:0]	写寄存器组寄存器的地址, 来自: 00: 0x1F(\$31), 相关指令: jal, 用于保存返回地址 (\$31<-pc+4) ; 01: rt 字段, 相关指令: addi、ori、sltiu、lw; 10: rd 字段, 相关指令: add、sub、or、and、slt、sll; 11: 未用;	
ALUOp[2:0]	ALU 8 种运算功能选择(000-111), 看功能表	

**相关部件及引脚说明:****Instruction Memory: 指令存储器**

Iaddr, 指令地址输入端口

DataIn, 存储器数据输入端口

DataOut, 存储器数据输出端口

RW, 指令存储器读写控制信号, 为 0 写, 为 1 读

**Data Memory: 数据存储器**

Daddr, 数据地址输入端口

DataIn, 存储器数据输入端口

DataOut, 存储器数据输出端口

/RD, 数据存储器读控制信号, 为 0 读

/WR, 数据存储器写控制信号, 为 0 写

### Register File: 寄存器组

Read Reg1, rs 寄存器地址输入端口

Read Reg2, rt 寄存器地址输入端口

Write Reg, 将数据写入的寄存器, 其地址输入端口 (rt、rd)

Write Data, 写入寄存器的数据输入端口

Read Data1, rs 寄存器数据输出端口

Read Data2, rt 寄存器数据输出端口

WE, 写使能信号, 为 1 时, 在时钟边沿触发写入

### IR: 指令寄存器, 用于存放正在执行的指令代码

### ALU: 算术逻辑单元

result, ALU 运算结果

zero, 运算结果标志, 结果为 0, 则 zero=1; 否则 zero=0

sign, 运算结果标志, 结果最高位为 0, 则 sign=0, 正数; 否则, sign=1, 负数

表 2 ALU 运算功能表

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

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

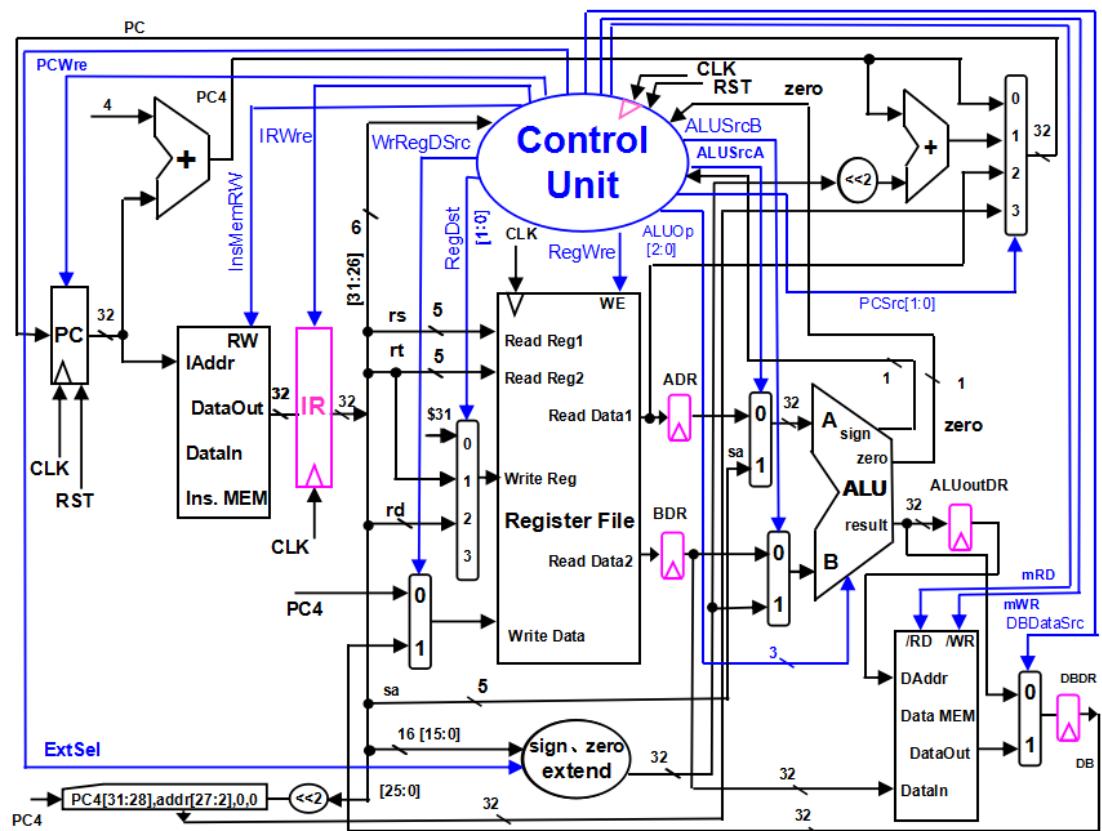
## 四. 实验器材

电脑一台, Xilinx Vivado 软件一套, Basys3板一块。

## 五. 实验过程与结果

(1) CPU设计的思想和方法:

a) 多周期CPU数据通路和控制线路图



以上即为多周期 CPU 数据通路和控制线路图，对比单周期 CPU 可以发现其实二者非常相近，多周期 CPU 在单周期 CPU 的基础上添加了一些元件。

由图可知，在多周期 CPU 中，多个部件各司其职，我们很容易想到使用模块化的思想，将上图进行分解，分解可得大纲如下：

- ✧ PC (程序计数器) ✓
- ✧ IM (指令存储器) ✓
- ✧ IR (指令寄存器) ✓
- ✧ RegisterFile (寄存器组) ✓
- ✧ Multiplexer (选择器)
  - PCMultiplexer (PC 选择器) ✓
  - WRMultiplexer (WriteReg 选择器) ✓
  - Multiplexer32 (32 位通用 2 选 1 选择器) ✓
- DB 选择器
- WriteData 选择器
- A 选择器

### ➤ B 选择器

- ✧ SignZeroExtend (立即数扩展单元) ✓
- ✧ ALU (算逻运算单元) ✓
- ✧ DM (数据存储器) ✓
- ✧ DR (数据寄存器) ✓
  - ADR
  - BDR
  - ALUoutDR
  - DBDR
- ✧ Control Unit (控制中心单元)。✓
  - DFlipFlop (D 触发器组) ✓
  - NextState (下一状态部件) ✓
  - OutputFunc (输出函数部件) ✓

我们容易将图分解为 9 类模块，对于其中的某类模块还可以选择性的细分，比如 Multiplexer (选择器) 等等。上面大纲中打✓的即为综合考虑后编写代码实现的模块。

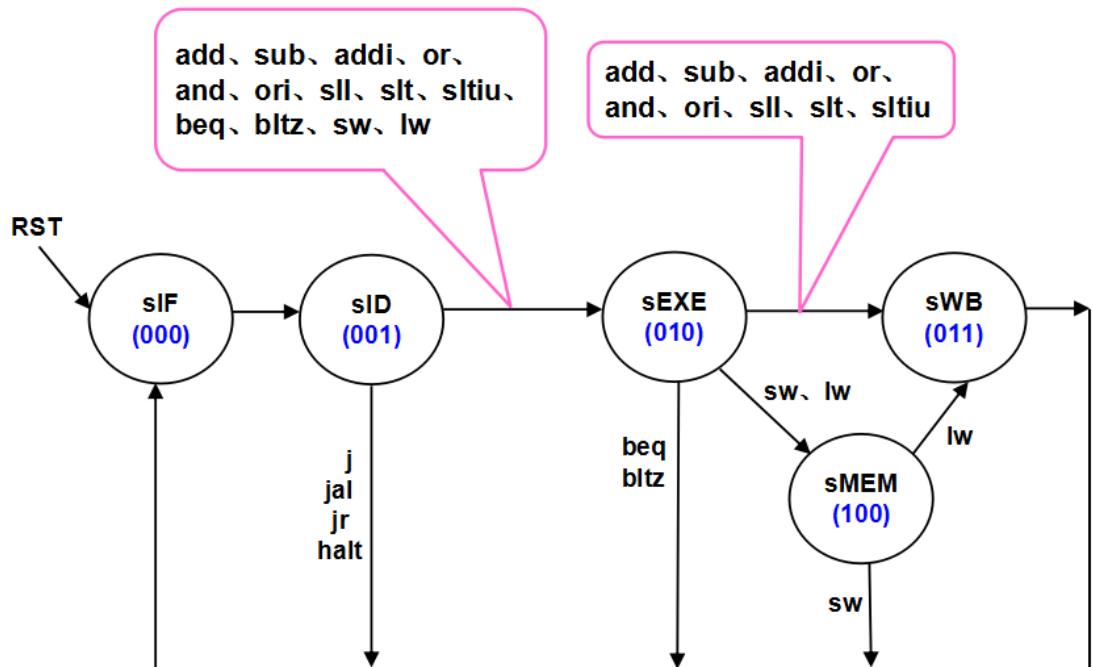
对于每一个功能模块，我们在设计时只需要关注其输入与输出，并按模块的功能和逻辑设计输入和输出之间的联系即可。

显然，只是实现大纲中打✓的模块还是不够的，因为分解所得的模块是相互独立的，此外没有其他任何模块进行统筹。我们还需要一个顶层模块，专用于驱动所实现的 CPU 中的分解的模块。在顶层模块中，我们的主要工作是生成这些部件，并对这些部件进行连接。

此外，为测试我们编写的 CPU 是否正确运行，我们还有必要设计一个测试模块，用于仿真模拟，检验每条指令是否按我们预期运行。

对于上述的数据通路和控制线路图，个人做了微小的改动：为 RegisterFile 也设置 Reset 输入，用于重置寄存器组。

### b) 多周期 CPU 状态转移图



综合多周期CPU数据通路和控制线路图和多周期CPU状态转移图，对各指令的运行情况进行分析。

- ✧ sIF：该状态下主要进行取指令操作：IM根据IAddr获取指令数据，然后在时钟的下降沿，IM的输出（也即指令数据）被载入到IR中去，这些工作完成后，其他部件便可以从IR的输出上获取到指令，这意味着取指令的完成。
- ✧ sID：该状态下主要对指令进行译码，主要体现在Control Unit根据sIF后从IR得到的op进行分析，从而进一步地产生大量的控制信号
  - 对于j、jal、jr和halt，这些控制信号显得比较有用一些：控制信号控制PC选择器的最终输出newAddress，控制必要的写操作等等
  - 对于其他指令，很多控制信号似乎暂时没有必要产生。这些指令在sID状态基本上什么都不做，我们只需要保证该状态结束前ADR和BDR已经正确的把ReadData1和ReadData2载入即可
- ✧ sEXE：该状态下主要进行ALU的算逻运算操作：ALU进行必要的运算，得到运算结果result、sign以及zero。其中result在该状态结束前需要已被载入到ALUoutDR中。
  - 对于beq和bltz，zero终于被求得，PCSrc[1:0]从而正确地指示下一个指令地址，这便于beq和bltz在下一步进行调跳转（分支跳转）

- 对于add、sub、addi、or、and、ori、sll、slt、sltiu，由于我们后续直接进行写操作，所以我们在该状态下，直接也将result载入到DBDR中去
- 对于sw和lw，后续进入sMEM状态，DBDataSrc选择的1输入还未确定，所以DBDR暂不载入数据
  - ◇ sMEM：该状态下主要进行存储器的读写。
- 对于sw，将DataIn写入指定DAddr对应的内存中的存储空间后，工作就已经完成。
- 对于lw，将指定DAddr对于的内存中的存储空间的数据获取后，从DataOut输出，然后再sMEM状态结束前，把该数据载入到DBDR中去，便于lw后续在sWB的写操作
  - ◇ sWB：该状态下主要进行寄存器的写回操作：将指定的WriteData写入到指定的WriteReg中去

c) 控制信号与指令及指令状态的关系表

state	Ins	ze r o	PC Wre	Ext Sel	Ins Mem RW	IR Wre	Wr Reg	Reg Dst D Src	Reg Wre	ALU Src A	ALU Src B	PC Src	ALU Op [2:0]	m RD	m WR	DB Data Src
sIF (000)	x (/halt)	x	1	x	1	1	x	xx	0	x	x	xx	xxx	x	x	x
	halt	x	0	x	1	1	x	xx	0	x	x	xx	xxx	x	x	x
sID (001)	j	x	0	x	0	0	x	xx	0	x	x	11	xxxx	x	x	x
	jal	x	0	x	0	0	0	00	1	x	x	11	xxx	x	x	x
	jr	x	0	x	0	0	x	xx	0	x	x	10	xxx	x	x	x
	halt	x	0	x	0	0	x	xx	0	x	x	xx	xxx	x	x	x
sEXE (010)	add	x	0	x	0	0	x	xx	0	0	0	00	000	x	x	0
	sub	x	0	x	0	0	x	xx	0	0	0	00	001	x	x	0
	addi	x	0	1	0	0	x	xx	0	0	1	00	000	x	x	0
	or	x	0	x	0	0	x	xx	0	0	0	00	101	x	x	0
	and	x	0	x	0	0	x	xx	0	0	0	00	110	x	x	0
	ori	x	0	0	0	0	x	xx	0	0	1	00	101	x	x	0
	sll	x	0	x	0	0	x	xx	0	1	0	00	100	x	x	0
	slt	x	0	x	0	0	x	xx	0	0	0	00	011	x	x	0
	sltiu	x	0	0	0	0	x	xx	0	0	1	00	010	x	x	0
	beq	0	0	1	0	0	x	xx	0	0	0	00	001	x	x	x

		1	0	1	0	0	x	xx	0	0	0	01	001	x	x	x
bltz	0	0	1	0	0	x	xx	0	0	0	01	011	x	x	x	
	1	0	1	0	0	x	xx	0	0	0	00	011	x	x	x	
sw	x	0	1	0	0	x	xx	0	0	1	00	000	x	x	x	
lw	x	0	1	0	0	x	xx	0	0	1	00	000	x	x	x	
	x	0	x	0	0	x	xx	0	x	x	00	xxx	1	0	1	
sMEM (100)	lw	x	0	x	0	0	x	xx	0	x	x	00	xxx	0	1	x
	sw	x	0	x	0	0	x	xx	0	x	x	00	xxx	0	1	x
sWB (011)	add	x	0	x	0	0	1	10	1	x	x	00	xxx	x	x	x
	sub	x	0	x	0	0	1	10	1	x	x	00	xxx	x	x	x
	addi	x	0	x	0	0	1	01	1	x	x	00	xxx	x	x	x
	or	x	0	x	0	0	1	10	1	x	x	00	xxx	x	x	x
	and	x	0	x	0	0	1	10	1	x	x	00	xxx	x	x	x
	ori	x	0	x	0	0	1	01	1	x	x	00	xxx	x	x	x
	sll	x	0	x	0	0	1	10	1	x	x	00	xxx	x	x	x
	slt	x	0	x	0	0	1	10	1	x	x	00	xxx	x	x	x
	sltiu	x	0	x	0	0	1	01	1	x	x	00	xxx	x	x	x
	lw	x	0	x	0	0	1	01	1	x	x	00	xxx	x	x	x

以上即为控制信号与指令及指令状态关系表。我们不难从多周期CPU数据通路和控制线路图中看到Control Unit的重要性。对比单周期CPU中的控制信号表，可以发现在多周期CPU中，控制信号的生成除了和指令有关（指令的op部分），还和CPU的状态有关。在一条指令的执行过程中，控制信号不再像单周期CPU那样从头到尾一成不变，而是随着CPU执行时的状态变化而变化。

由图可知，Control Unit并没有状态输入，所以状态的产生和转换是在Control Unit中完成的，也即Control Unit内置了状态机。从Control Unit的op输入可知，Control Unit的工作是分析指令的op部分（也即对指令进行译码），然后根据CPU的状态产生控制信号去控制其他的模块的工作。但注意，Control Unit和顶层模块依然有区别，它的工作需要确保指令op的输入，而这需要PC模块和INSMEM模块的一些预备工作。

#### d) 各模块说明及其实现

##### ① PC (程序计数器)

程序计数器，保存当前执行的指令的地址，在Control Unit的控制下更新指令地址

- ✧ 输入: newAddress[31:0] CLK RST PCWre
- ✧ 输出: curAddress[31:0]

```

`timescale 1ns / 1ps

module PC(
    input CLK,
    input RST,
    input PCWre,
    input [31:0] newAddress,
    output reg [31:0] curAddress

);

initial begin
    curAddress = 0;
end

always@(PCWre or RST) begin
    if(!RST) begin
        curAddress = 0;
    end
    else if(PCWre) begin
        curAddress = newAddress;
    end
end

endmodule

```

在PC的实现中，使PC的值的更新和重置都受电平敏感信号PCWre和RST触发而非时序敏感信号触发CLK和RST触发，这样有利于解决延迟问题。

## ② IM（指令存储器）

**指令存储器，根据给定的指令地址输入，对指令进行寻址并将获取到的指令输出**  
 整个模块的工作如下：通过IAddr获取指令所在的地址，我们就可以找到对应的指令，将其存储在指令寄存器内。然后在InsMemRW的控制信号下进行输出具体指令。

- ✧ 输入： IAddr[31:0] IDataIn[31:0] InsMemRW
- ✧ 输出： DataOut[31:0]

```

`timescale 1ns / 1ps

module IM(
    input InsMemRW, // 读写控制

```

```

    input [31:0] IAddr,           // 指令地址
    input [31:0] IDataIn,         // 指令代码输入
    output reg [31:0] IDataOut   // 指令代码输出
);

reg [7:0] Memory[0:127];

initial begin
$readmemb("D:/vivado/MultipleCycleCPU/data/instruction.txt",
Memory);
end

always@(IAddr or InsMemRW)
begin
if(InsMemRW) begin
    IDataOut[31:24] = Memory[IAddr];
    IDataOut[23:16] = Memory[IAddr + 1];
    IDataOut[15:8] = Memory[IAddr + 2];
    IDataOut[7:0] = Memory[IAddr + 3];
end
end

endmodule

```

### ③ IR (指令寄存器)

**指令寄存器，寄存器存储当前执行的指令**

- ✧ 输入: InsDataIn[31:0] CLK IRWre
- ✧ 输出: InsDataOut[31:0]

`timescale 1ns / 1ps

```

module IR(
    input [31:0] InsDataIn,
    input CLK,
    input IRWre,
    output reg [31:0] InsDataOut
);

always@(negedge CLK) begin
if(IRWre) begin
    InsDataOut <= InsDataIn;
end

```

```

    end
endmodule

```

该模块是在多周期CPU中新添加的模块，其作用是使指令代码保持稳定。

实验原理部分提示的是在时钟上升沿，IR接受指令数据输入，但思考后我改成了下降沿。

在时钟上升沿阶段，多周期CPU需要先让IM根据给定的IAddr获取到相应的指令数据，这个获取行为由InsMemRW来控制，而实际上InsMemRW和IRWre信号几乎是同时从Control Unit发出的。那么，可能出现以下情况，IR的InsDataIn输入还没有从IM得到（相应的指令数据），在这个上升沿选择载入指令数据到IR是有风险的。

整个sIF（取指令）阶段，我们完全可以在时钟周期的前半段让IM先获取到相应的指令数据，然后在时钟周期的后半段让IR载入指令数据。

#### ④ RegisterFile（寄存器组）

寄存器组，所有寄存器的读写在这里进行

- ✧ 输入： CLK RST RegWre rs[4:0] rt[4:0] WriteReg[4:0]  
WriteData[31:0]
- ✧ 输出： ReadData1[31:0] ReadData2[31:0]

```
`timescale 1ns / 1ps
```

```

module RegisterFile(
    input CLK,                      // 时钟
    input RegWre,                   // 写使能信号
    input RST,                      // 重置信号
    input [4:0] rs,                 // Read Reg1
    input [4:0] rt,                 // Read Reg2
    input [4:0] WriteReg,           // 将被写入数据的寄存器
    input [31:0] WriteData,          // 将被写入的数据
    output [31:0] ReadData1,         // rs 数据
    output [31:0] ReadData2          // rt 数据
);

reg [31:0] register[0:31];

integer i;
initial begin

```

```

    for(i = 0; i < 32; i = i + 1) begin
        register[i] = 0;
    end
end

assign ReadData1 = register[rs];
assign ReadData2 = register[rt];

// 在时钟下降沿将 WriteData 写入 WriteReg
always@(negedge CLK) begin
    // 在时钟下降沿检查 RST 是否为 0, 是则重置寄存器组
    if(!RST) begin
        for(i = 1; i < 32; i = i + 1) begin
            register[i] <= 0;
        end
    end
    else if(RegWre && WriteReg != 0) begin
        register[WriteReg] <= WriteData;
    end
end
end

```

**endmodule**

在本模块中，我们一共设立了32个32位的寄存器，其中0号寄存器恒为0。为什么是设立32个呢？这和指令中表示寄存器地址的字段长度有关，比如rs、rt和rd长度都为5位，也即最多只能表示 $2^5$ 个寄存器，也即32个寄存器。

该模块主要具有两个功能，其一是读取rs寄存器和rt寄存器的具体数据。读取操作不需要控制信号的控制，为连续赋值，一旦rs或者rt寄存器的数据发生变化，则输出ReadData1和ReadData2也紧接着发生变化。

其二是将需要写入的数据写到指定的寄存器中去，这一步通过RegWre来控制，在时钟的下降沿到来时，将WriteData写入到WriteReg中去。

此外，该模块还有一个同步重置的操作，若RST为0，在时钟下降沿将触发重置操作，这一步将所有寄存器的值重置为0。

## ⑤ Multiplexer (选择器) > PCMultiplexer (PC选择器)

PC选择器，用于选择PC模块的新地址

✧ 输入：CLK S0[31:0] S1[31:0] S2[31:0] S3[31:0] PCSrc[1:0]

■ PCSrc[1:0]:

- ◆ 00: pc<-pc+4
- ◆ 01: pc<-pc+4+(sign-extend)immediate
- ◆ 10: pc<-rs
- ◆ 11: pc<-(pc+4)[31:28],addr[27:2],2'b00

✧ 输出: PCOut[31:0]

```
`timescale 1ns / 1ps
```

```
module PCMultiplexer(
    input CLK,
    input [1:0] PCSrc,
    input [31:0] S0,
    input [31:0] S1,
    input [31:0] S2,
    input [31:0] S3,
    output reg [31:0] PCOut
);

    always@(posedge CLK) begin
        case(PCSsrc)
            2'b00: PCOut <= S0;
            2'b01: PCOut <= S1;
            2'b10: PCOut <= S2;
            2'b11: PCOut <= S3;
        endcase
    end

endmodule
```

该模块原本没有CLK输入，但在仿真过程中出现了问题，问题主要出现在旧指令的结束和新指令sIF的中间阶段。

首先我们知道，在多周期CPU中，控制信号的值在一条指令的执行中并非是一成不变的，其值在不同的状态中可能不同。而时钟周期的变化往往也伴随着状态的变化，状态的变化也使得控制信号发生了变化。

原本在一个时钟上升沿到来后，PC的值应该更新为期望的新Address。但在PC的值更新为期望的新Address之前（该newAddress由期望的PCSrc指定），PCSrc的值先一步因为时CLK的改变而改变了，这就使得PC被更新为错误的由PCSrc指定的新Address。恰好在旧指令的结束和新指令的sIF中，一旦提

供的newAddress在一个时钟上升沿到来后是错误的，那么该错误的newAddress就被更新为PC的值。

通过添加CLK输入，确保在时钟上升沿才出发PCOut的输出（而不是原先的连续赋值），这样就可以保证即使PCSrc的值先一步因为时钟CLK的改变而改变，PCOut也保存着我们期望的值。

## ⑥ Multiplexer (选择器) > WRMultiplexer (WriteReg选择器)

WriteReg选择器，用于选择RegisterFile模块的WriteReg

◆ 输入: S0[4:0] S1[4:0] S2[4:0] RegDst[1:0]

- RegDst[1:0]:

- ◆ 00: 0x1F(\$31)，相关指令: jal，用于保存返回地址\$31<-pc+4；
- ◆ 01: rt字段
- ◆ 10: rs字段
- ◆ 11: 未用

◆ 输出: WROut[4:0]

```
`timescale 1ns / 1ps
```

```
module WRMultiplexer(
    input [1:0] RegDst,
    input [4:0] S0,
    input [4:0] S1,
    input [4:0] S2,
    output [4:0] WROut
);

    assign WROut = RegDst[1] ? S2 : (RegDst[0] ? S1 : S0);

endmodule
```

## ⑦ Multiplexer (选择器) > Multiplexer32 (32位通用2选1选择器)

简单的2选1选择器，用于选择两个32位的输入。

选择器的输入control和输出Out在不同的地方有更加具体的意义，比如在用于DB选择的DB选择器中，control对应着DBDataSrc，Out对应着DBOut。

◆ 输入: S0[31:0] S1[31:0] control

```

✧ 输出: Out[31:0]
`timescale 1ns / 1ps

module Multiplexer32(
    input control,
    input [31:0] S0,
    input [31:0] S1,
    output [31:0] Out
);

    assign Out = control ? S1 : S0;
endmodule

```

对应多周期CPU数据通路和控制线路图，需要4个该选择器，用作：

- DB选择器，用于选择DB上的数据 (ALU模块产生的result和DM模块产生的DataOut)
- WriteData选择器，用于选择RegisterFile模块的WriteData
- A选择器，选择ALU模块的A操作数
- B选择器，选择ALU模块的B操作数

### ⑧ SignZeroExtend (立即数扩展单元)

立即数扩展单元，根据ExtSel控制信号对输入的immediate(立即数)进行带符号扩展或零扩展，由16位扩展到32位

```

✧ 输入: ExtSel Immediate[15:0]
✧ 输出: ExtendImmediate[31:0]
`timescale 1ns / 1ps

module SignZeroExtend(
    input [15:0] Immediate,
    input ExtSel, // 0 => zero-extend, 1 => signed-extend
    output [31:0] ExtendImmediate
);

    assign ExtendImmediate[15:0] = Immediate;
    assign ExtendImmediate[31:16] = ExtSel ? (Immediate[15] ?
16'hffff : 16'h0000) : 16'h0000;

endmodule

```

## ⑨ ALU (算逻运算单元)

算逻运算单元，根据Control Unit的ALUOp来对A B两个操作数进行相应的运算操作。

◇ 输入: A[31:0] B[31:0] ALUOp[2:0]

■ ALUOp[2:0]:

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

◇ 输出: result[31:0] zero sign

```
`timescale 1ns / 1ps

module ALU(
    input [2:0] ALUOp,           // ALU 操作控制
    input [31:0] A,              // A 输入
    input [31:0] B,              // B 输入
    output reg zero,             // 运算结果零标志
    output reg sign,             // 运算结果符号位
    output reg [31:0] result     // 运算结果
);

always@(ALUOp or A or B) begin
    case (ALUOp)
        3'b000 : result = (A + B);
        3'b001 : result = (A - B);
        3'b010 : result = (A < B) ? 1 : 0;
        3'b011 : result = (((A < B) \&\& (A[31] == B[31])) || (A[31]
&& !B[31])) ? 1 : 0;
        3'b100 : result = (B << A);
        3'b101 : result = (A | B);
    endcase
end
```

```

      3'b110 : result = (A & B);
      3'b111 : result = (A ^ B);
      default : result = 0;
endcase

zero = (result == 0) ? 1 : 0;
sign = result[31];
end

endmodule

```

对应最上方的数据通路图，容易发现有个地方需要修改，也即ALUSrcA所选择的S1部分。该部分对应输入指令的sa部分，但显然在数据通路图中没有明确指出sa的零扩展，所以在写代码的时候要记得对sa还需要记得额外做处理。

## ⑩ DM (数据存储器)

数据存储器，根据给定的存储器的数据地址，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，专用于处理lw和sw指令

- ✧ 输入: DAddr[31:0] DataIn[31:0] mRD mWR
- ✧ 输出: DataOut[31:0]

```

`timescale 1ns / 1ps

module DM(
    input mRD,           // 读数据使能输入
    input mWR,           // 写数据使能输入
    input [31:0] DAddr,   // 数据内存地址 (Memory)
    input [31:0] DataIn,  // 写入内存的数据, sw
    output [31:0] DataOut // 从内存读出的数据, lw
);

reg [7:0] Memory[0:128]; // 8 位长的数据存储单元

integer i;
initial begin
    for(i = 0; i < 128; i = i + 1) begin
        Memory[i] = 0;
    end
end

```

```

// 读数据, mRD == 1 => 读, mRD == 0 => 输出高阻态
// 大端模式, 所以要从低地址获取高位数据
assign DataOut[31:24] = (mRD) ? Memory[DAddr] : 8'bz;
assign DataOut[23:16] = (mRD) ? Memory[DAddr + 1] : 8'bz;
assign DataOut[15:8] = (mRD) ? Memory[DAddr + 2] : 8'bz;
assign DataOut[7:0] = (mRD) ? Memory[DAddr + 3] : 8'bz;

// 写数据, mWR == 1 => 写, mWR == 0 => 无操作
// 同样遵循大端模式, 高位数据存放到低地址
always@(mWR or DataIn) begin
    if(mWR) begin
        Memory[DAddr] = DataIn[31:24];
        Memory[DAddr + 1] = DataIn[23:16];
        Memory[DAddr + 2] = DataIn[15:8];
        Memory[DAddr + 3] = DataIn[7:0];
    end
end

endmodule

```

这个模块是整个CPU中比较难理解的一个模块。这个模块实际上是专门用来处理lw和sw指令的。

DAddr输入隐隐约约暗示了一些信息，而该输入ALU的运算结果result，这表明result可以是一个地址，而在众多指令中，把ALU结果作为地址的只有lw和rw。

mRD输入专门用于控制模块的读部分，当mRD为1时，它执行读操作，将数据存储器中的数据读到寄存器中，此时显然执行的是lw。

mWR输入专门用于控制模块的写部分，当mWR为1时，它执行写操作，将寄存器中的数据写到数据存储器（内存）中，此时输入DataIn作用就很明了了：DataIn输入来自寄存器组的ReadData2，也即指令中rt字段对应的地址的数据，该数据将在DM中被写到内存中去，此时显然执行的是sw。注意在执行sw时，DataOut是不重要的，因为sw指令的工作已经在模块内完成，无需做其他的输入。

## ⑪ DR (数据寄存器)

数据寄存器，寄存器存储CPU运行过程中的一些数据，用于切分数据通路，将大延迟变为多段小延迟。

◆ 输入： CLK DRIn[31:0]

```

✧ 输出: DROut[31:0]

`timescale 1ns / 1ps

module DR(
    input CLK,           // 时钟信号
    input [31:0] DRIn,   // 数据输入
    output reg [31:0] DROut // 数据输出
);
    always@(negedge CLK) begin
        DROut <= DRIn;
    end

endmodule

```

对应多周期CPU数据通路和控制线路图，需要4个数据寄存器，用作：

- ADR：用于存储A操作数
- BDR：用于存储B操作数
- ALUoutDR：用于存储ALU计算的结果result
- DBDR：用于存储DB

## ⑫ Control Unit (控制中心单元)

控制单元，根据指令操作码和当前状态来产生相应的控制信号，发放到各个模块。

根据前文的控制单元原理结构图，可知其能分为三个小模块：触发器模块，下一状态模块和输出函数模块。

```

✧ 输入: CLK RST op[5:0] zero sign
✧ 输出: PCWre ExtSel InsMemRW IRWre WrRegDSrc
          RegDst[1:0] RegWre ALUSrcA ALUSrcB PCSrc[1:0]
          ALUOp[2:0] mRD mWR DBDataSrc
`timescale 1ns / 1ps

```

```

module ControlUnit(
    input CLK,           // 时钟信号
    input RST,           // 重置信号
    input [5:0] op,       // 指令操作码
    input zero,          // ALU 运算结果零标志
    input sign,          // ALU 运算结果符号位
    output PCWre,         // PC 值更新控制信号
    output ExtSel,        // 立即数扩展控制信号

```

```

    output InsMemRW,           // 指令存储器写使能信号
    output IRWre,              // 指令寄存器写使能信号
    output WrRegDSrc,          // 写入寄存器数据源选择信号
    output [1:0] RegDst,        // 写入寄存器选择信号
    output RegWre,              // 寄存器组写使能信号
    output ALUSrcA,             // A 操作数数据源选择信号
    output ALUSrcB,             // B 操作数数据源选择信号
    output [1:0] PCSrc,          // 下一条指令地址的选择信号
    output [2:0] ALUOp,          // ALU 操作选择信号
    output mRD,                  // 数据存储器读使能信号
    output mWR,                  // 数据存储器写使能信号
    output DBDataSrc,            // DB 数据源选择信号
    output [2:0] StateIn,         // 状态输入信号
    output [2:0] StateOut
);

DFlipFlop dff(.StateIn(StateIn), .CLK(CLK), .RST(RST),
    .StateOut(StateOut));
NextState ns(.NowStateIn(StateOut), .op(op),
    .NextStateOut(StateIn));
OutputFunc
opf(.NowStateIn(StateOut), .op(op), .zero(zero), .sign(sign),
    .PCWre(PCWre), .ExtSel(ExtSel), .InsMemRW(InsMemRW), .IRWre(IRWre),
    .WrRegDSrc(WrRegDSrc), .RegDst(RegDst), .RegWre(RegWre),
    .ALUSrcA(ALUSrcA),
    .ALUSrcB(ALUSrcB), .PCSrc(PCSsrc), .ALUOp(ALUOp), .mRD(mRD),
    .mWR(mWR),
    .DBDataSrc(DBDataSrc)
);

endmodule

为提高代码可读性，也为更方便编写代码，创建了如下指令操作码和状态码头文件
`timescale 1ns / 1ps

// 指令操作码
`define opAdd 6'b000000
`define opSub 6'b000001
`define opAddi 6'b000010
`define opOr 6'b010000
`define opAnd 6'b010001
`define opOri 6'b010010

```

```

`define opSll 6'b011000
`define opSlt 6'b100110
`define opSltiu 6'b100111
`define opSw 6'b110000
`define opLw 6'b110001
`define opBeq 6'b110100
`define opBltz 6'b110110
`define opJ 6'b111000
`define opJr 6'b111001
`define opJal 6'b111010
`define opHalt 6'b111111

// 状态码
`define sIF 3'b000
`define sID 3'b001
`define sEXE 3'b010
`define sMEM 3'b100
`define sWB 3'b011

```

### ⑬ DFlipFlop (D触发器组) In Control Unit (控制中心单元)

D触发器，更确切的说是由3个D触发器一起构成的一个D触发器组，其用于保存当前的状态

- ✧ 输入: RST CLK StateIn[2:0]
- ✧ 输出: StateOut[2:0]

```

`timescale 1ns / 1ps

module DFlipFlop(
    input [2:0] StateIn,           // 输入状态
    input CLK,                     // 时钟信号
    input RST,                     // 重置信号
    output reg [2:0] StateOut     // 输出状态
);

    always@(posedge CLK) begin
        if(!RST) StateOut <= 3'b000;
        else StateOut <= StateIn;
    end

endmodule

```

#### ⑯ nextState (下一状态部件) In Control Unit (控制中心单元)

下一状态部件，给定当前状态输入和指令的操作码op，输出下一状态

◇ 输入: NowStateIn[2:0] op[5:0]

◇ 输出: NextStateOut[2:0]

```
`timescale 1ns / 1ps
```

```
`include "head.v"
```

```
module nextState(
    input [2:0] NowStateIn,           // 当前状态输入
    input [5:0] op,                  // 指令操作码
    output reg [2:0] NextStateOut   // 输出下一状态
);

    always@(NowStateIn or op) begin
        case (NowStateIn)
            // `sHalt: NextStateOut = `sIF;
            `sIF: NextStateOut = `sID;
            `sID: begin
                case (op)
                    `opJ, `opJal, `opJr, `opHalt: NextStateOut =
`sIF;
                    default: NextStateOut = `sEXE;
                endcase
            end
            `sEXE: begin
                case (op)
                    `opBeq, `opBltz: NextStateOut = `sIF;
                    `opSw, `opLw: NextStateOut = `sMEM;
                    default: NextStateOut = `sWB;
                endcase
            end
            `sMEM: begin
                case (op)
                    `opSw: NextStateOut = `sIF;
                    `opLw: NextStateOut = `sWB;
                endcase
            end
            `sWB: NextStateOut = `sIF;
        endcase
    end

endmodule
```

### ⑯ OutputFunc (输出函数部件) In Control Unit (控制中心单元)

输出函数部件，综合输入的状态信息、指令操作码以及ALU结果信息并进行分析，输出相应的控制信号

```

`timescale 1ns / 1ps

`include "head.v"

module OutputFunc(
    input [2:0] NowStateIn, // 当前状态输入
    input [5:0] op,         // 指令操作码
    input zero,             // ALU 运算结果零标志
    input sign,              // ALU 运算结果符号位
    output reg PCWre,       // PC 值更新控制信号
    output reg ExtSel,       // 立即数扩展控制信号
    output reg InsMemRW,     // 指令存储器写使能信号
    output reg IRWre,        // 指令寄存器写使能信号
    output reg WrRegDSrc,    // 写入寄存器数据源选择信号
    output reg [1:0] RegDst,   // 写入寄存器选择信号
    output reg RegWre,        // 寄存器组写使能信号
    output reg ALUSrcA,       // A 操作数数据源选择信号
    output reg ALUSrcB,       // B 操作数数据源选择信号
    output reg [1:0] PCSrc,      // 下一条指令地址的选择信号
    output reg [2:0] ALUOp,      // ALU 操作选择信号
    output reg mRD,            // 数据存储器读使能信号
    output reg mWR,              // 数据存储器写使能信号
    output reg DBDataSrc       // DB 数据源选择信号
);

```

```

    always@(NowStateIn) begin
        PCWre = (NowStateIn == `sIF && op != `opHalt) ? 1: 0;
        ExtSel = (NowStateIn == `sEXE) ? ((op == `opOri || op ==
`opSltiu) ? 0 : 1) : 1;
        InsMemRW = (NowStateIn == `sIF) ? 1 : 0;
        IRWre = (NowStateIn == `sIF) ? 1 : 0;

```

```

WtRegDSrc = (NowStateIn == `sID && op == `opJal) ? 0 : 1;
RegDst[1] = (NowStateIn == `swB) ? ((op == `opAddi || op ==
`opOri || op == `opSltiu || op == `opLw) ? 0 : 1) : 0;
RegDst[0] = (NowStateIn == `swB) ? ((op == `opAddi || op ==
`opOri || op == `opSltiu || op == `opLw) ? 1 : 0) : 0;
RegWre = (NowStateIn == `swB) ? 1 : ((NowStateIn == `sID &&
op == `opJal) ? 1 : 0);
ALUSrcA = (NowStateIn == `sEXE && op == `opSll) ? 1 : 0;
ALUSrcB = (NowStateIn == `sEXE) ? ((op == `opAddi || op ==
`opOri || op == `opSltiu || op == `opSw || op == `opLw) ? 1 : 0) :
0;
ALUOp[2] = (NowStateIn == `sEXE) ? ((op == `opOr || op ==
`opAnd || op == `opOri || op == `opSll) ? 1 : 0) : 0;
ALUOp[1] = (NowStateIn == `sEXE) ? ((op == `opAnd || op ==
`opSlt || op == `opSltiu || op == `opBltz)) : 0;
ALUOp[0] = (NowStateIn == `sEXE) ? ((op == `opSub || op ==
`opOr || op == `opOri || op == `opSlt || op == `opBeq || op == `opBltz) ?
1 : 0) : 0;
mRD = (NowStateIn == `sMEM && op == `opLw) ? 1 : 0;
mWR = (NowStateIn == `sMEM && op == `opSw) ? 1 : 0;
DBDataSrc = (NowStateIn == `sMEM && op == `opLw) ? 1 : 0;
end

always@(NowStateIn or zero) begin
PCSsrc[1] = (NowStateIn == `sID) ? 1 : 0;
PCSsrc[0] = (NowStateIn == `sID) ? ((op == `opJ || op == `opJal) ?
1 : 0) : ((NowStateIn == `sEXE) ? (((op == `opBeq && zero == 1) ||
(op == `opBltz && zero == 0)) ? 1 : 0) : 0;
end
endmodule

```

以上控制信号的赋值依据为前文的控制信号与指令及指令状态的关系表。

## (2) 验证CPU设计的准确性（指令测试）

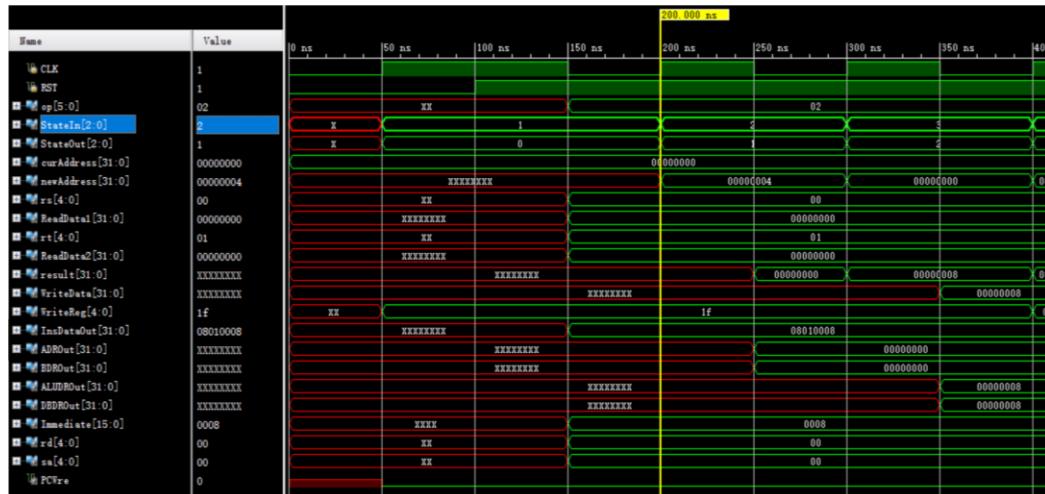
在该部分我们使用老师提供的指令进行逐条的测试。测试覆盖了所有实验要求的指令（除add）。观察指令的运行，若符合预期，则表明CPU设计正确。

测试程序段如下表：

地址	汇编程序	指令代码					
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x000000000	addi \$1,\$0,8	0000 10	00 000	0 0001	0000 0000 0000 1000	=	08010008

0x00000004	ori \$2,\$0,2	0100 10	00 000	0 0010	0000 0000 0000 0010	=	48020002
0x00000008	or \$3,\$2,\$1	0100 00	00 010	0 0001	0001 1000 0000 0000	=	40411800
0x0000000C	sub \$4,\$3,\$1	0000 01	00 011	0 0001	0010 0000 0000 0000	=	04612000
0x00000010	and \$5,\$4,\$2	0100 01	00 100	0 0010	0010 1000 0000 0000	=	44822800
0x00000014	sll \$5,\$5,2	0110 00	00 000	0 0101	0010 1000 1000 0000	=	60052880
0x00000018	beq \$5,\$1,-2(=,转 14)	1101 00	00 101	0 0001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	jal 0x00000040	1110 10	00 000	0 0000	0000 0000 0001 0000	=	E8000010
0x00000020	slt \$8,\$12,\$1	1001 10	01 100	0 0001	0100 0000 0000 0000	=	99814000
0x00000024	addi \$13,\$0,-2	0000 10	00 000	0 1101	1111 1111 1111 1110	=	080DFFFFE
0x00000028	slt \$9,\$8,\$13	1001 10	01 000	0 1101	0100 1000 0000 0000	=	990D4800
0x0000002C	sltiu \$10,\$9,2	1001 11	01 001	0 1010	0000 0000 0000 0010	=	9D2A0002
0x00000030	sltiu \$11,\$10,0	1001 11	01 010	0 1011	0000 0000 0000 0000	=	9D4B0000
0x00000034	addi \$13,\$13,1	0000 10	01 101	0 1101	0000 0000 0000 0001	=	09AD0001
0x00000038	bltz \$13,-2(<0,转 34)	1101 10	01 101	0 0000	1111 1111 1111 1110	=	D9A0FFFFE
0x0000003C	j 0x0000004C	1110 00	00 000	0 0000	0000 0000 0001 0011	=	E0000013
0x00000040	sw \$2,4(\$1)	1100 00	00 001	0 0010	0000 0000 0000 0100	=	C0220004
0x00000044	lw \$12,4(\$1)	1100 01	00 001	0 1100	0000 0000 0000 0100	=	C42C0004
0x00000048	jr \$31	1110 01	11 111	0 0000	0000 0000 0000 0000	=	E7E00000
0x0000004C	halt	1111 11	00 000	0 0000	0000 0000 0000 0000	=	FC000000
0x00000050							
0x00000054							

进行仿真模拟后，出现波形图如下：

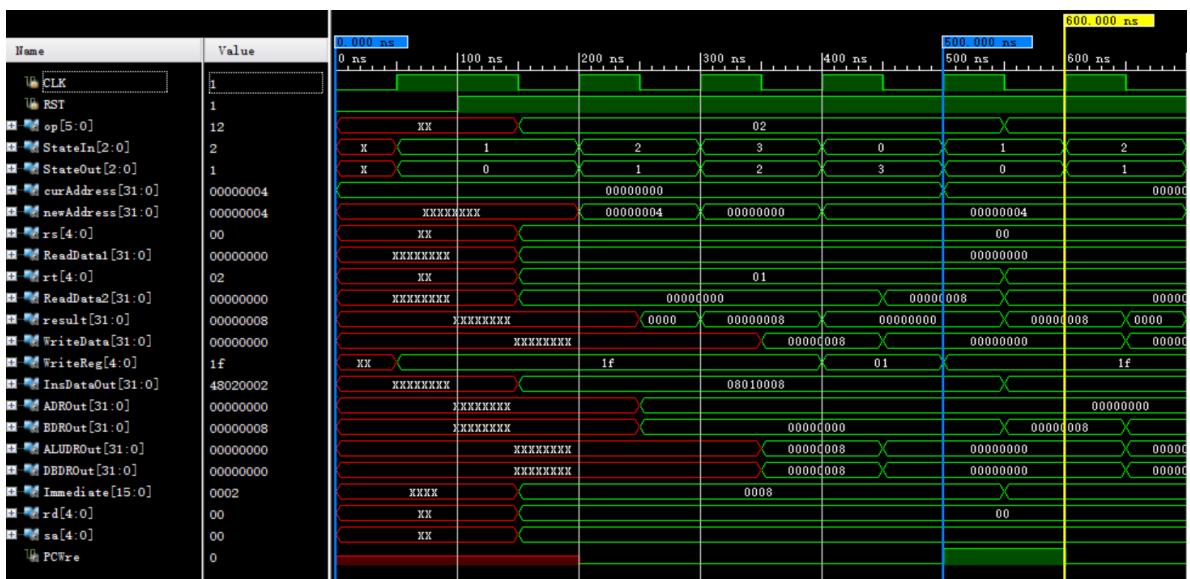


为方便起见，我们先稍微对上图进行说明

- CLK: 时钟信号
- RST: 重置信号
- op: 指令的操作码op部分
- StateIn: 表示当前CPU的下一个状态
- StateOut: 表示当前CPU的状态

- curAddress: 当前执行指令的地址
- newAddress: 下一条执行指令的地址
- rs: 指令的rs部分
- ReadDara1: rs对应的寄存器存放的数据
- rt: 指令的rt部分
- ReadData2: rt对应的寄存器存放的数据
- result: ALU运算所得结果
- WriteData: 将被写入寄存器的数据
- WriteReg: 将被写入的寄存器
- InsDataOut: 指令寄存器IR的输出
- ADROut: 数据寄存器ADR输出
- BDROut: 数据寄存器BDR输出
- ALUDROut: 数据寄存器ALUDR输出
- DBDROut: 数据寄存器DBDR输出
- Immediate: 指令的立即数部分, 也即后16位
- rd: 指令的rd部分
- sa: 指令的sa部分
- PCWre: PC值更新信号

### ① addi \$1,\$0,8 (对于第一条指令, 我会给出相对详细的解释)



图中两根蓝线所包围部分即为该条指令的仿真波形。

由图可知，该条指令执行了4个时钟周期，当前指令地址为00000000。

在第一个时钟周期时，当前CPU状态为0（StateOut值为0），为sIF状态。

在sIF状态的CLK下降沿处，InsDataOut的值变为08010008（为该条指令的16进制数代码），这表示指令已从存储单元中成功获取。指令成功获取后，指令从而也能成功地分段输出，可以看到，rs为00，对应0号寄存器，数据为0，rt为01，对应1号寄存器，数据为0，Immediate值为8。

在第二个时钟周期时，当前CPU状态为1（StateOut值为1），为sID状态。

在sID状态的CLK下降沿处，ADROut和BDROut的值变为00000000，这表示ADR和BDR已经正确地把ReadData1（rs数据）和ReadData2（rt数据）载入。

在第三个时钟周期时，当前CPU状态为2（StateOut值为2），为sEXE状态。

在sEXE状态的CLK上升沿处，result求得值为8（\$0 + 8 = 0 + 8 = 8），然后在sEXE状态的CLK下降沿处，result的值被载入到ALUDR和DBDR中，WriteData被指定为result数据8。

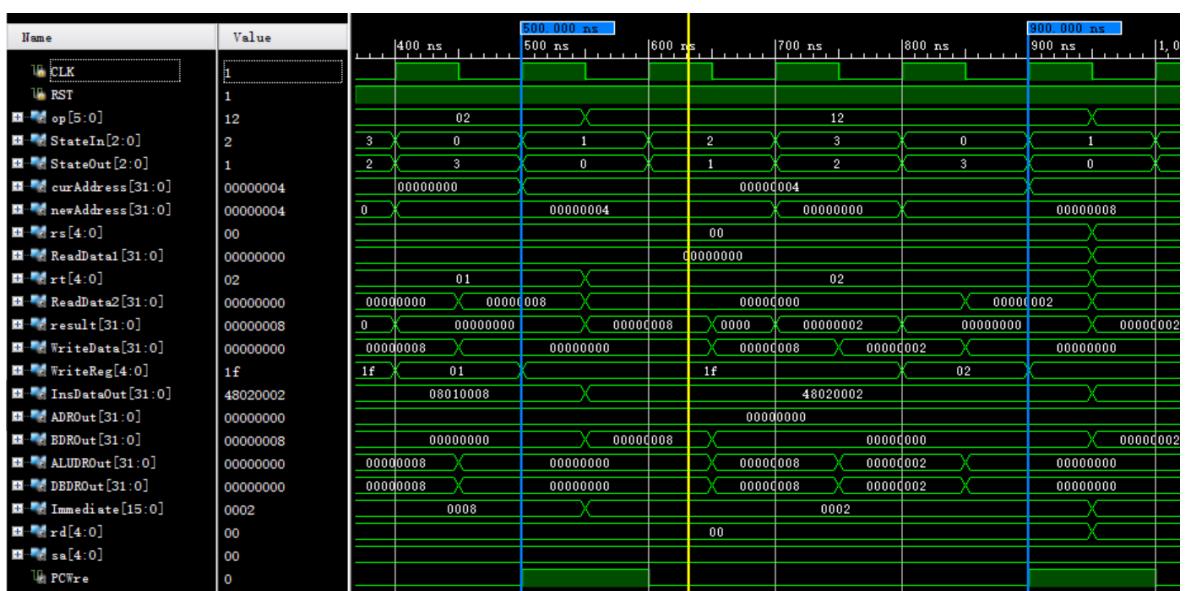
在第四个时钟周期时，当前CPU状态为3（StateOut值为3），为sWB状态。

在sWB状态的CLK上升沿处，WriteReg被指定为1号寄存器，然后在sWB状态的CLK下降沿处rt的值从0变为8，这表示运算结果8已被写入到1号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8

## ② ori \$2,\$0,2



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000004。

sIF时，InsDataOut的值变为48020002（为该条指令的16进制数代码）。

rs为00，对应0号寄存器，数据为0，rt为02，对应2号寄存器，数据为0，Immediate值为2。

sID时，ADR成功载入rs数据0，BDR成功载入rt数据0。

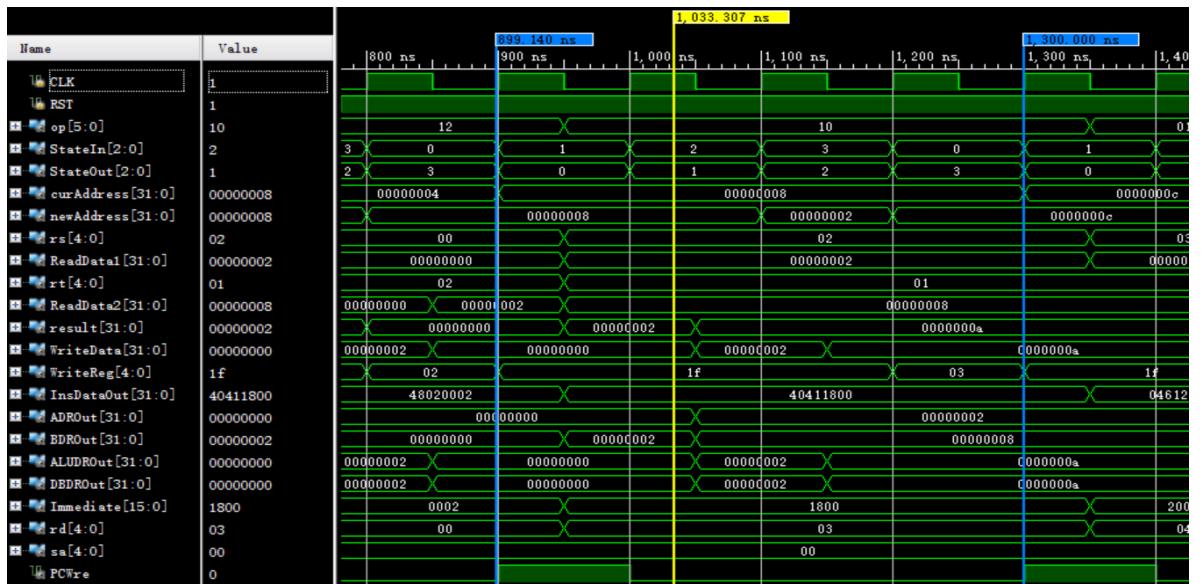
sEXE时，result被求得结果为2（\$0 + 2 = 0 + 2 = 2），ALUDR和DBDR载入result数据2。WriteData被指定为result数据2。

sWB时，WriteReg被指定为2号寄存器，WriteData数据2被写入2号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2。

③ or \$3,\$2,\$1



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000008。

sIF时，InsDataOut的值变为40411800（为该条指令的16进制数代码）。

rs为02，对应2号寄存器，数据为2，rt为01，对应1号寄存器，数据为8，rd为03，对应3号寄存器。

sID时，ADR成功载入rs数据2，BDR成功载入rt数据8。

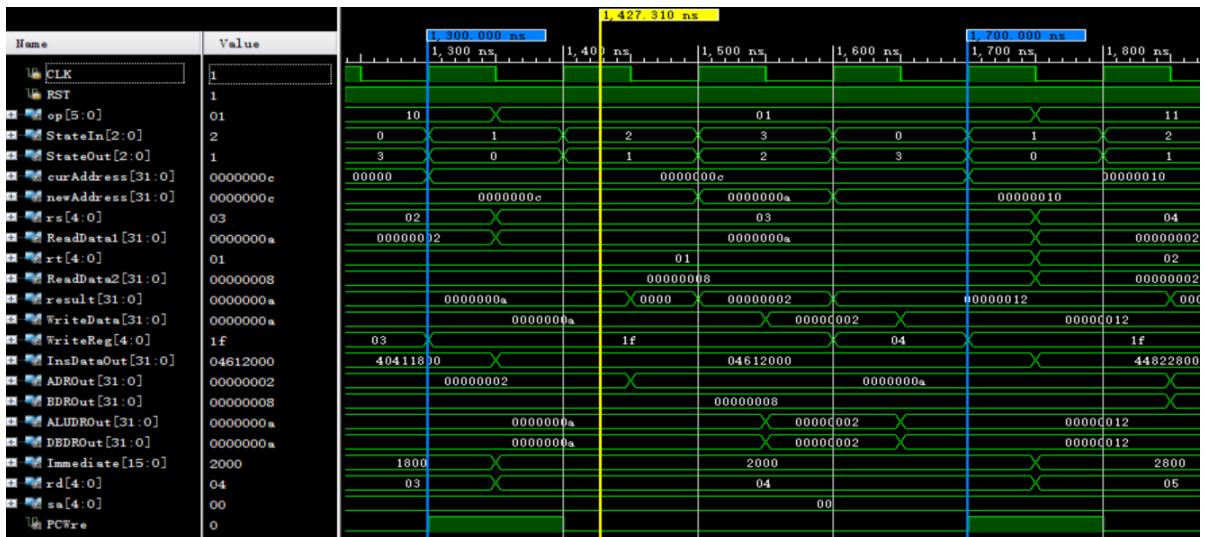
sEXE时，result被求得结果为10（\$2 + \$1 = 2 + 8 = 10），ALUDR和DBDR载入result数据10。WriteData被指定为result数据10。

sWB时，WriteReg被指定为3号寄存器，WriteData数据10被写入3号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10。

#### ④ sub \$4,\$3,\$1



由图可知，该条指令执行了4个时钟周期，当前指令地址为0000000c。

sIF时，InsDataOut的值变为04612000（为该条指令的16进制数代码）。

rs为03，对应3号寄存器，数据为10，rt为01，对应1号寄存器，数据为8，rd为04，对应4号寄存器。

sID时，ADR成功载入rs数据10，BDR成功载入rt数据8。

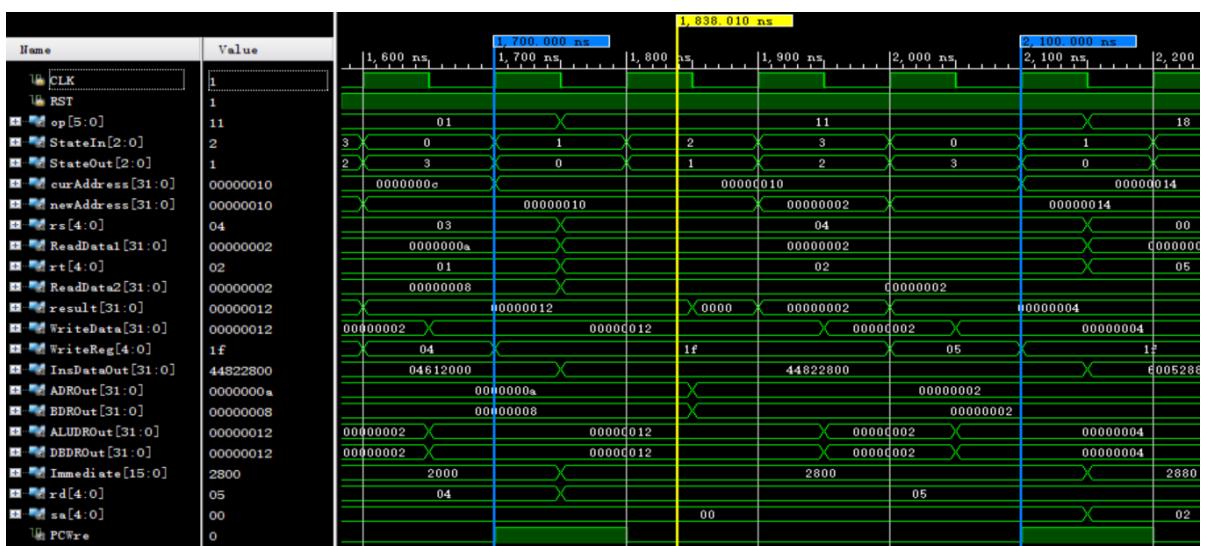
sEXE时，result被求得结果为2 (\$3 - \$1 = 10 - 8 = 2)，ALUDR和DBDR载入result数据2。WriteData被指定为result数据2。

sWB时，WriteReg被指定为4号寄存器，WriteData数据2被写入4号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2。

#### ⑤ and \$5,\$4,\$2



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000010。

sIF时，InsDataOut的值变为44822800（为该条指令的16进制数代码）。

rs为04，对应4号寄存器，数据为2，rt为02，对应2号寄存器，数据为2，rd为05，  
对应5号寄存器。

sID时，ADR成功载入rs数据2（\$4 & \$2 = 2 & 2 = 2），BDR成功载入rt  
数据2。

sEXE时，result被求得结果为2，ALUDR和DBDR载入result数据2。  
WriteData被指定为result数据2。

sWB时，WriteReg被指定为5号寄存器，WriteData数据2被写入5号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 2。

#### ⑥ sll \$5,\$5,2



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000014。

sIF时，InsDataOut的值变为60052880（为该条指令的16进制数代码）。

rs为00，对应0号寄存器，数据为0，rt为05，对应5号寄存器，数据为2，rd为05，  
对应5号寄存器，sa为02，表示左移2位。

sID时，ADR成功载入rs数据0，BDR成功载入rt数据2。

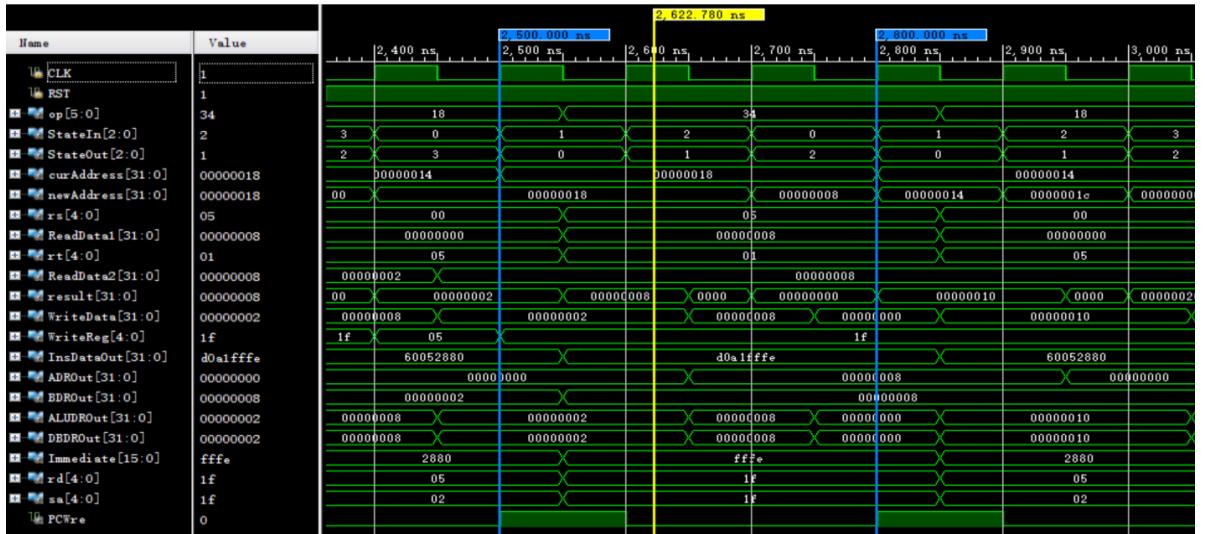
sEXE时，result被求得结果为8（\$5 << 2 = 2 << 2 = 8），ALUDR和DBDR  
载入result数据8。WriteData被指定为result数据8。

sWB时，WriteReg被指定为5号寄存器，WriteData数据8被写入5号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 8。

#### ⑦ beq \$5,\$1,-2

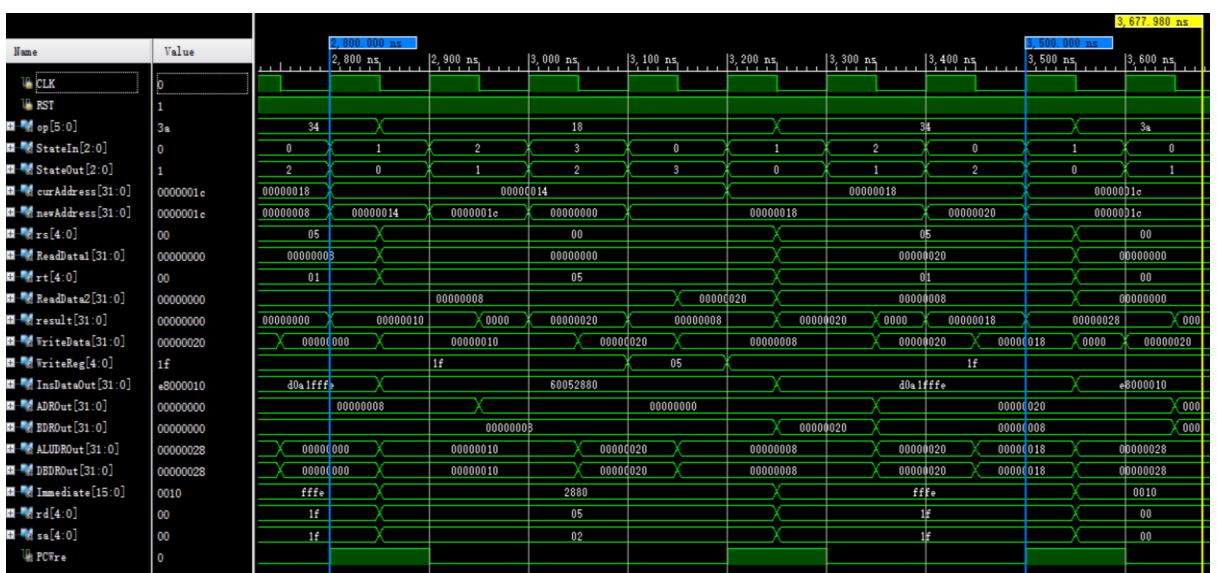


由图可知，该条指令执行了3个时钟周期，当前指令地址为00000018。

sIF时，InsDataOut的值变为d0a1ffe（为该条指令的16进制数代码）。rs为05，对应5号寄存器，数据为8，rt为01，对应1号寄存器，数据为8，Immediate值为-2（如图中fffe）。

sID时，ADR成功载入rs数据8，BDR成功载入rt数据8。

sEXE时，result被求得结果为0 ( $\$5 - \$1 = 8 - 8 = 0$ )，ALUDR和DBDR载入result数据0。WriteData被指定为result数据0。求得result结果为0可知，\$5和\$1的值相等，所以进行分支跳转，跳转到 $PC + 4 + (-2 * 4) = PC - 4$ ，也即上一条指令，由图可知，下一条指令的地址确实为00000014。



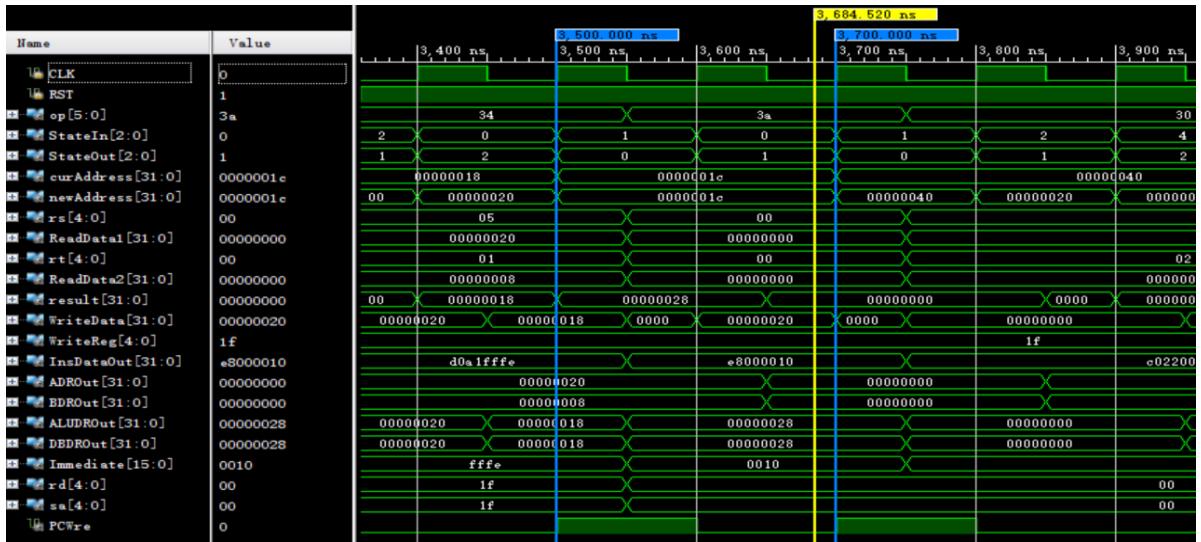
分支跳转到00000014后，\$5再次进行移位， $\$5 < 2 = 8 \ll 2 = 32$ ，再次执行移位后，\$5的值由8变为32。此时再回到00000018， $\$5 = 32 \neq \$1 = 8$ ，

跳转到PC + 4，也即0000001C。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32。

#### ⑧ jal 0x00000040



由图可知，该条指令执行了2个时钟周期，当前指令地址为0000001C。

sIF时，InsDataOut的值变为e8000010（为该条指令的16进制数代码）。

sID时，WriteData被指定为数据00000020（这也即PC + 4），WriteReg

被指定为1f，也即\$31（31号寄存器）。该状态下，PC + 4被写入到\$31中去，

同时指定CPU将要跳转到的地址0x0000040（由图可知，下一条指令地址确实为00000040）。

综上可知，指令执行完全正确。

#### ⑨ sw \$2,4(\$1)



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000040。

sIF时，InsDataOut的值变为c0220004（为该条指令的16进制数代码）。

rs为01，对应1号寄存器，数据为8，rt为02，对应2号寄存器，数据为2，Immediate值为4。

sID时，ADR成功载入rs数据8，BDR成功载入rt数据2。

sEXE时，result被求得结果为12 ( $4 + \$1 = 4 + 8 = 12$ )，该结果即为将被写入的存储器中存储单元的数据地址，我们将把\$2的数据写到12的数据地址所指定的存储单元。

sMEM时，\$2的数据写到12的数据地址所指定的存储单元。

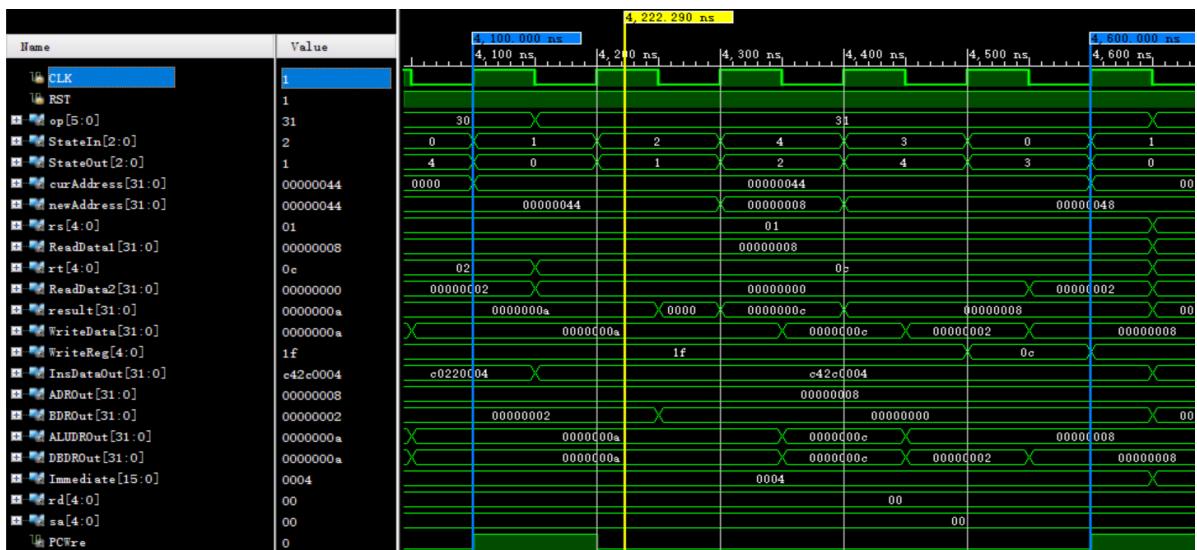
我们现在查看Memory的情况，检查\$2的数据是否成功写入：

Memory[0:12...]	00, 00, 00, 00, 00, ...	Array
[0][7:0]	00	Array
[1][7:0]	00	Array
[2][7:0]	00	Array
[3][7:0]	00	Array
[4][7:0]	00	Array
[5][7:0]	00	Array
[6][7:0]	00	Array
[7][7:0]	00	Array
[8][7:0]	00	Array
[9][7:0]	00	Array
[10][7:0]	00	Array
[11][7:0]	00	Array
[12][7:0]	00	Array
[13][7:0]	00	Array
[14][7:0]	00	Array
[15][7:0]	02	Array
[16][7:0]	00	Array
[17][7:0]	00	Array
[18][7:0]	00	Array
[19][7:0]	00	Array
[20][7:0]	00	Array
[21][7:0]	00	Array
[22][7:0]	00	Array
[23][7:0]	00	Array
[24][7:0]	00	Array

根据大端模式，低地址存放数据高位，所以要把\$2的值2 (0x00000002) 存入内存的话，那么12应该存放00，13存放00，14存放00而15存放02，对比上图发现完全一致，这表明\$2的值存入到内存相应地址的操作成功了。

综上可知，指令执行完全正确。

⑩ lw \$12,4(\$1)



由图可知，该条指令执行了5个时钟周期（也只有lw指令需要执行5个时钟周期），当前指令地址为00000044。

sIF时，InsDataOut的值变为c42c0004（为该条指令的16进制数代码）。

rs为01,对应1号寄存器,数据为8,rt为0c,对应12号寄存器,数据为0,Immediate值为4。

sID时，ADR成功载入rs数据8，BDR成功载入rt数据0。

sEXE时，result被求得结果为12 ( $4 + \$1 = 4 + 8 = 12$ )，该结果即为将被读取的存储器中存储单元的数据地址，我们将读取12的数据地址所指定的存储单元。

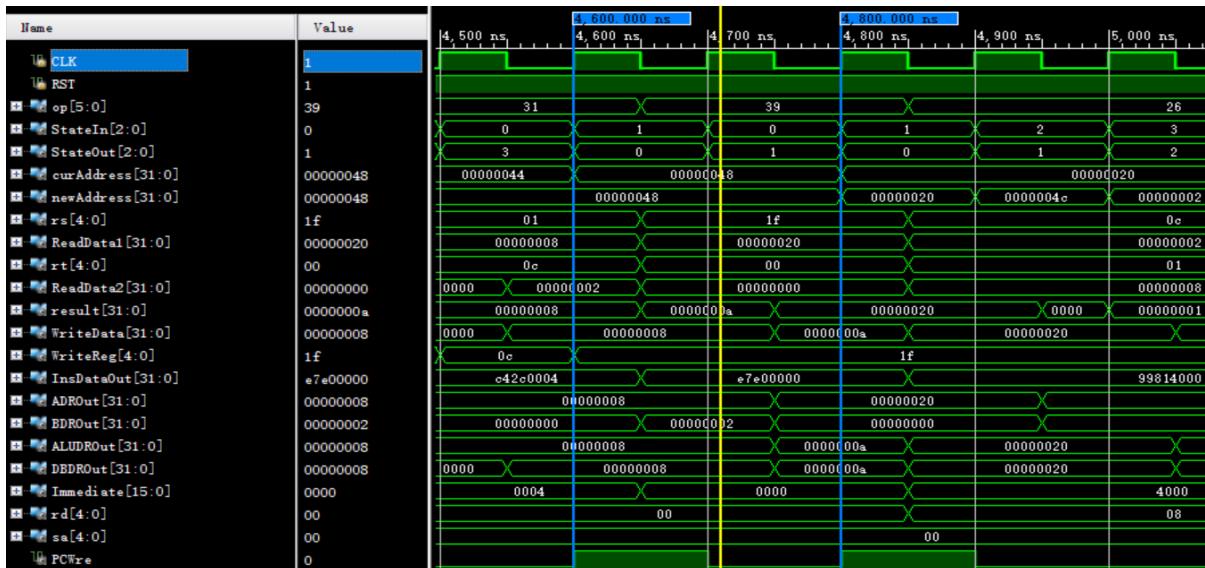
sMEM时，从12的数据地址所指定的存储单元中获取数据。在CLK下降沿，DBDROut载入数据2，WriteData被指定为数据2（2即为从Memory[12:15]获读出的数据，在上一步sw时我们将\$2的数据写入Memory[12:15]，这样证明了我们上一步的写入是正确的）

sWB时，WriteReg被指定为12号寄存器，WriteData数据2被写入12号寄存器。

综上可知，指令执行完全正确。

此时,  $\$0 = 0$ ,  $\$1 = 8$ ,  $\$2 = 2$ ,  $\$3 = 10$ ,  $\$4 = 2$ ,  $\$5 = 32$ ,  $\$12 = 2$ ;  
Memory[12:15] = 2。

⑪ jr \$31



由图可知，该条指令执行了2个时钟周期，当前指令地址为00000048。

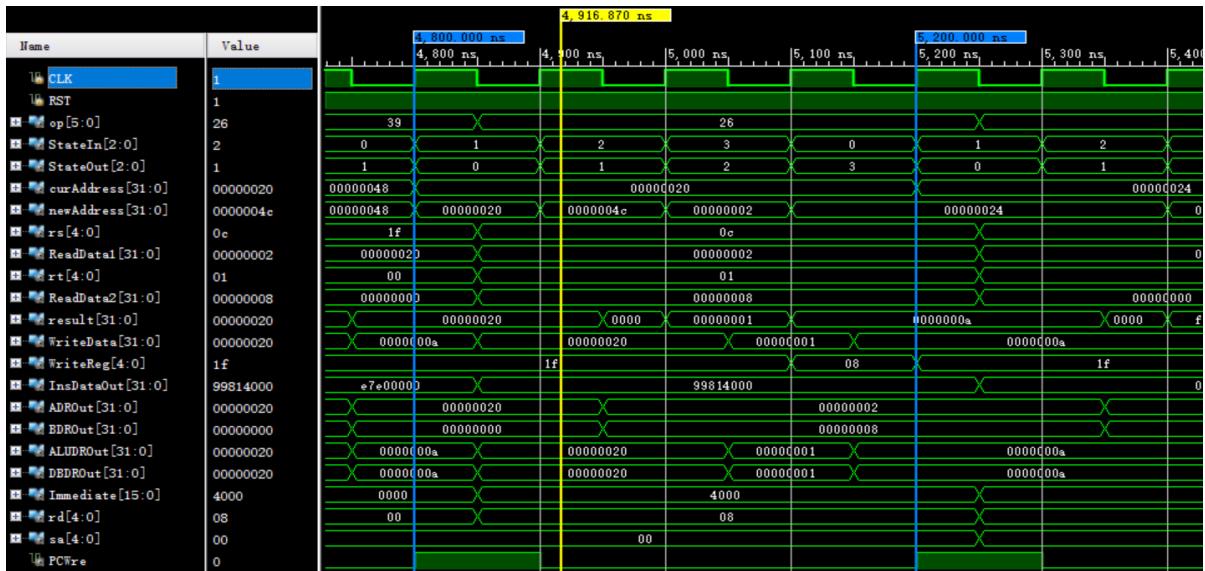
sIF时，InsDataOut的值变为e7e00000（为该条指令的16进制数代码）。

rs为1f，对应31号寄存器，数据为20。

sID时，CPU跳转到31号寄存器所指定的指令地址0x0000020（由图可知，下一条指令地址确实为00000020）。

综上可知，指令执行完全正确。

## ⑫ slt \$8,\$12,\$1



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000020。

sIF时，InsDataOut的值变为99814000（为该条指令的16进制数代码）。

rs为0c，对应12号寄存器，数据为2，rt为01，对应1号寄存器，数据为8，rd为08，对应8号寄存器。

sID时，ADR成功载入rs数据2，BDR成功载入rt数据8。

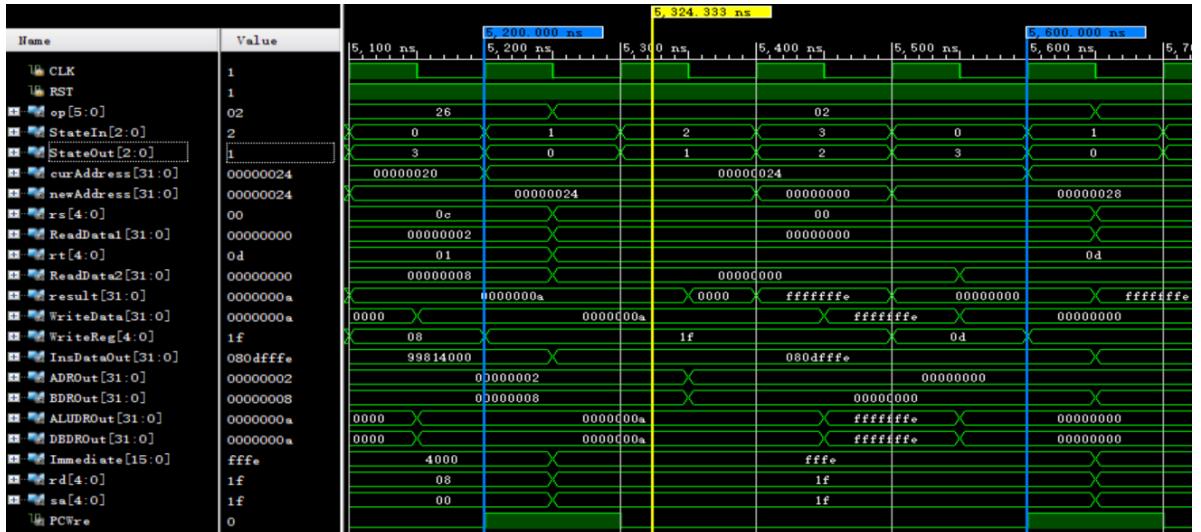
sEXE时，result被求得结果为1 ( $\because \$12 = 2 < \$1 = 8, \therefore$  需要置位)，ALUDR和DBDR载入result数据1。WriteData被指定为result数据1。

sWB时，WriteReg被指定为8号寄存器，WriteData数据1被写入8号寄存器。

综上可知，指令执行完全正确。

此时， $\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32, \$12 = 2, \$8 = 1$ ；  
Memory[12:15] = 2。

### ⑬ addi \$13,\$0,-2



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000024。

sIF时，InsDataOut的值变为080dffffe（为该条指令的16进制数代码）。rs为00，对应0号寄存器，数据为0，rt为0d，对应13号寄存器，数据为0，Immediate值为-2(如图中ffffe)。

sID时，ADR成功载入rs数据0，BDR成功载入rt数据0。

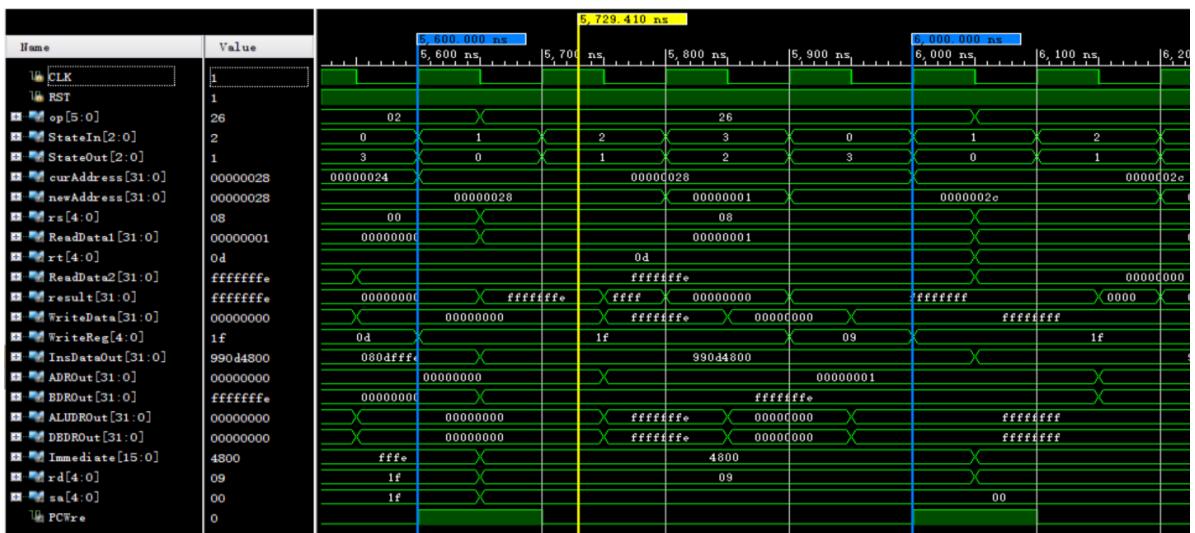
sEXE时，result被求得结果为-2 ( $\$0 + (-2) = 0 - 2 = -2$ )，ALUDR和DBDR载入result数据-2。WriteData被指定为result数据-2。

sWB时，WriteReg被指定为13号寄存器，WriteData数据-2被写入13号寄存器。

综上可知，指令执行完全正确。

此时， $\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32, \$8 = 1, \$12 = 2, \$13 = -2$ ；Memory[12:15] = 2。

### ⑭ slt \$9,\$8,\$13



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000028。

sIF时，InsDataOut的值变为990d4800（为该条指令的16进制数代码）。

rs为08，对应8号寄存器，数据为1，rt为0d，对应13号寄存器，数据为-2，rd为09，对应9号寄存器。

sID时，ADR成功载入rs数据1，BDR成功载入rt数据-2。

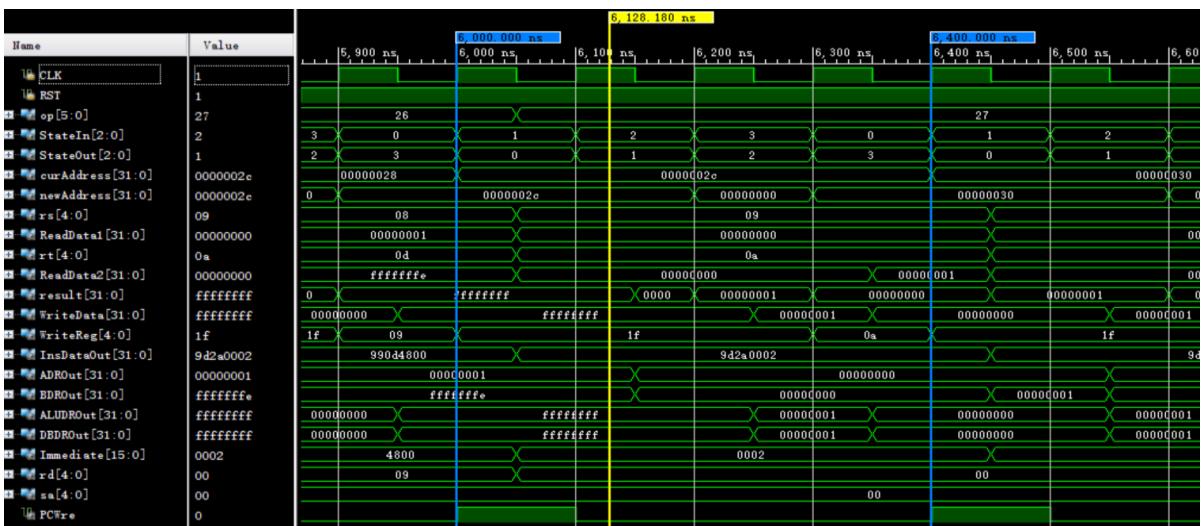
sEXE时，result被求得结果为0 ( $\because \$8 = 1 \leftarrow \$13 = -2$ ,  $\therefore$ 需要复位)，ALUDR和DBDR载入result数据0。WriteData被指定为result数据0。

sWB时，WriteReg被指定为9号寄存器，WriteData数据0被写入9号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32, \$8 = 1, \$9 = 0, \$12 = 2, \$13 = -2；Memory[12:15] = 2。

### ⑯ sltiu \$10,\$9,2



由图可知，该条指令执行了4个时钟周期，当前指令地址为0000002c。

sIF时，InsDataOut的值变为9d2a0002（为该条指令的16进制数代码）。

rs为09，对应9号寄存器，数据为0，rt为0a，对应10号寄存器，数据为0，Immediate值为2。

sID时，ADR成功载入rs数据0，BDR成功载入rt数据0。

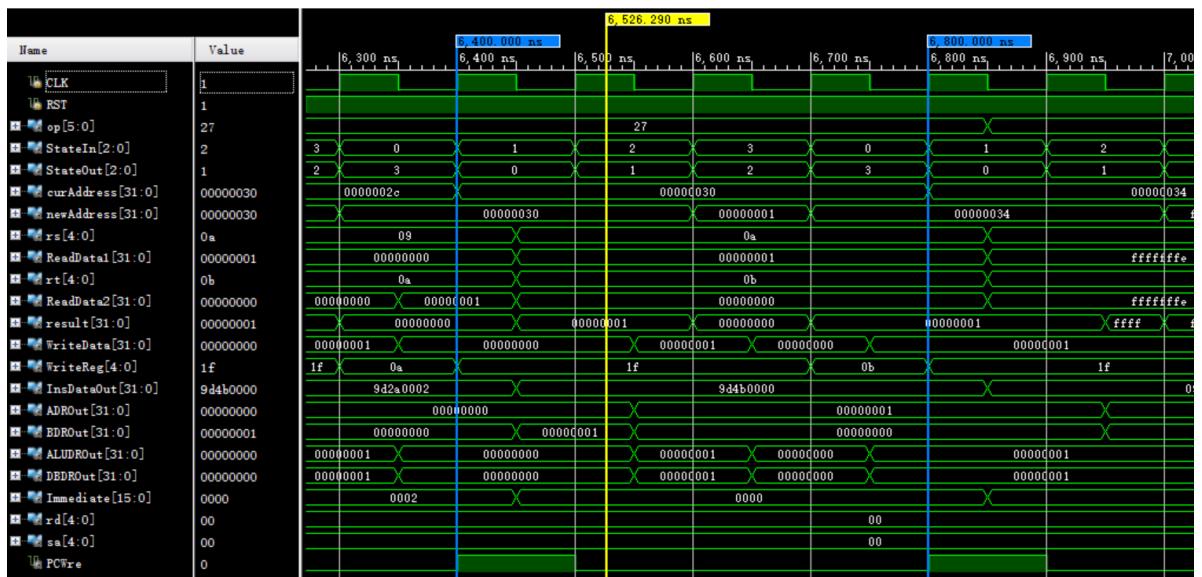
sEXE时，result被求得结果为1（ $\because \$9 = 0 < 2$ ,  $\therefore$ 需要置位），ALUDR和DBDR载入result数据1。WriteData被指定为result数据1。

sWB时，WriteReg被指定为10号寄存器，WriteData数据1被写入10号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32, \$8 = 1, \$9 = 0, \$10 = 1, \$12 = 2, \$13 = -2; Memory[12:15] = 2。

#### ⑯ sliu \$11,\$10,0



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000030。

sIF时，InsDataOut的值变为9d4b0000（为该条指令的16进制数代码）。

rs为0a，对应10号寄存器，数据为1，rt为0b，对应11号寄存器，数据为0，Immediate值为0。

sID时，ADR成功载入rs数据1，BDR成功载入rt数据0。

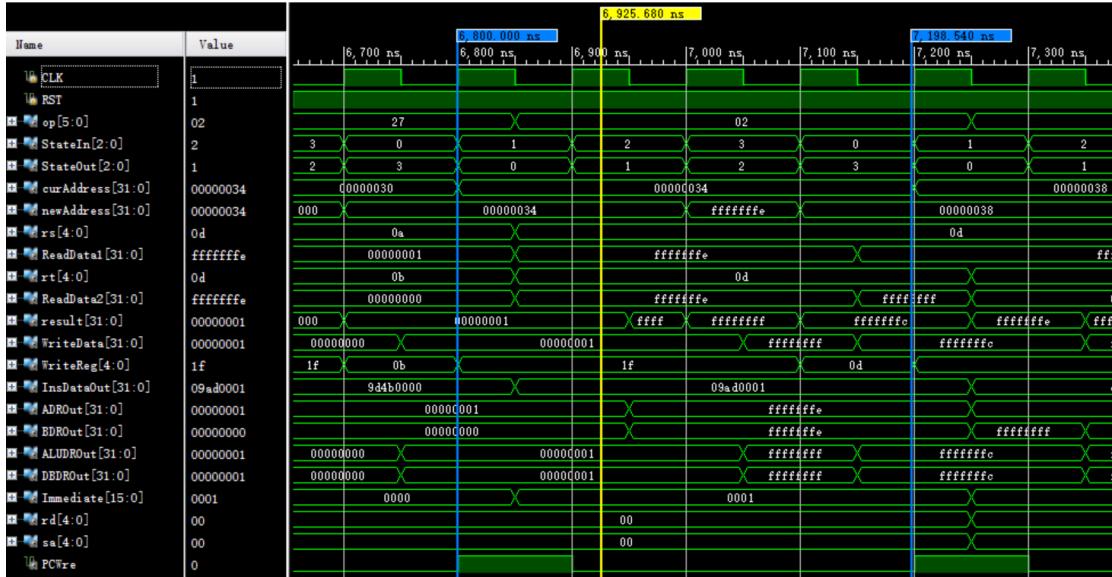
sEXE时，result被求得结果为0（ $\because \$10 = 1 \neq 0$ ,  $\therefore$ 需要复位），ALUDR和DBDR载入result数据0。WriteData被指定为result数据0。

sWB时，WriteReg被指定为11号寄存器，WriteData数据0被写入11号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32, \$8 = 1, \$9 = 0, \$10 = 1, \$11 = 0, \$12 = 2, \$13 = -2; Memory[12:15] = 2。

⑯ addi \$13,\$13,1



由图可知，该条指令执行了4个时钟周期，当前指令地址为00000034。

sIF时，InsDataOut的值变为09ad0001（为该条指令的16进制数代码）。

rs为0d，对应13号寄存器，数据为-2，rt为0d，对应13号寄存器，数据为-2，

Immediate值为1。

sID时，ADR成功载入rs数据-2，BDR成功载入rt数据-2。

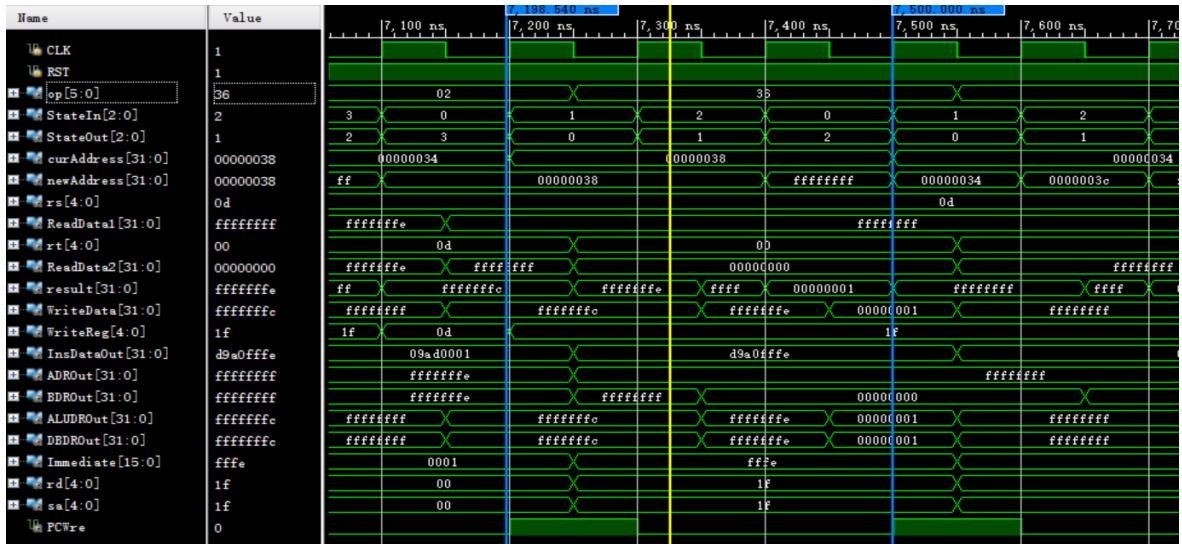
sEXE时，result被求得结果为-1 (\$13 + 1 = -2 + 1 = -1)，ALUDR和DBDR载入result数据-1。WriteData被指定为result数据-1。

sWB时，WriteReg被指定为13号寄存器，WriteData数据-1被写入13号寄存器。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32, \$8 = 1, \$9 = 0, \$10 = 1, \$11 = 0, \$12 = 2, \$13 = -1; Memory[12:15] = 2。

⑰ bltz \$13,-2 (<0,转34)

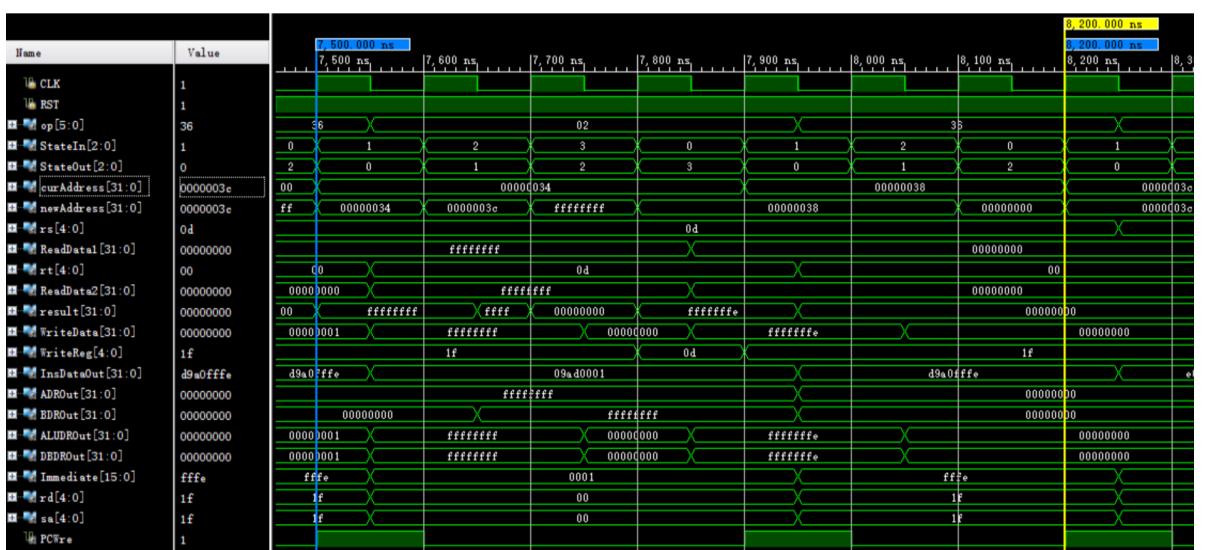


由图可知，该条指令执行了3个时钟周期，当前指令地址为00000038。

sIF时，InsDataOut的值变为d9a0ffe（为该条指令的16进制数代码）。rs为0d，对应13号寄存器，数据为-1，rt为00，对应0号寄存器，数据为0，Immediate值为-2（如图中fffe）。

sID时，ADR成功载入rs数据-1，BDR成功载入rt数据0。

sEXE时，result被求得结果为1 ( $\because \$13 = -1 < 0$ ,  $\therefore \text{ltz}$ 成立，结果为true，也即1），ALUDR和DBDR载入result数据1。WriteData被指定为result数据1。求得result结果为1可知，\$13小于0，所以进行分支跳转，跳转到 $\text{PC} + 4 + (-2 * 4)$  =  $\text{PC} - 4$ ，也即上一条指令，由图可知，下一条指令的地址确实为00000034。



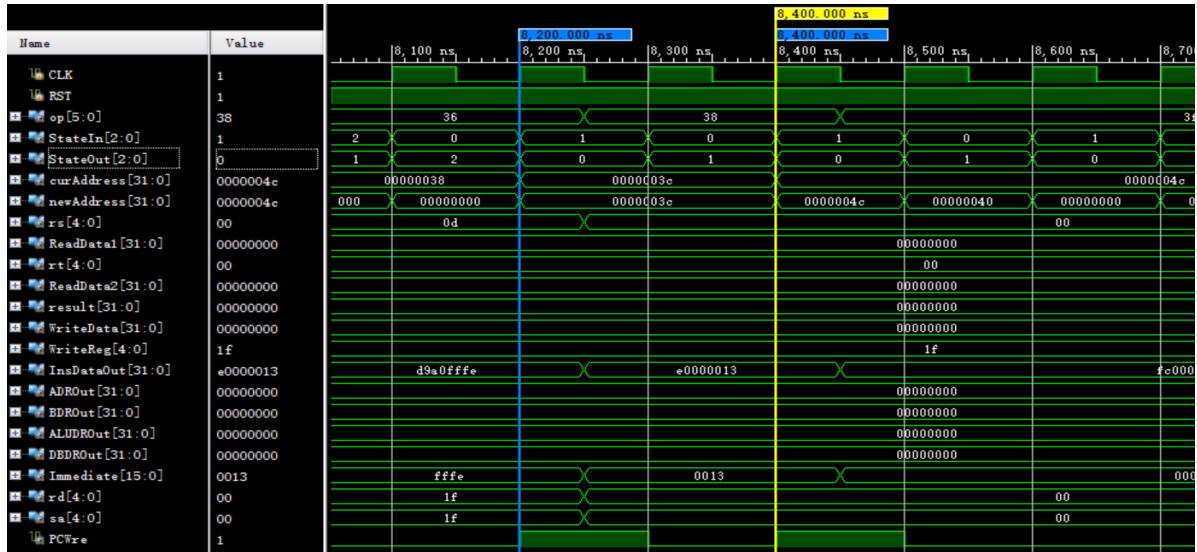
分支跳转到00000034后，\$13再次被加1， $\$13 + 1 = -1 + 1 = 0$ ，再次执行加操作后，\$13的值由-1变为0。

此时再回到00000038， $\$13 = 0 \not< 0$ ，跳转到 $\text{PC} + 4$ ，也即0000003C。

综上可知，指令执行完全正确。

此时，\$0 = 0, \$1 = 8, \$2 = 2, \$3 = 10, \$4 = 2, \$5 = 32, \$8 = 1, \$9 = 0, \$10 = 1, \$11 = 0, \$12 = 2, \$13 = 0; Memory[12:15] = 2。

### ⑯ j 0x000004C



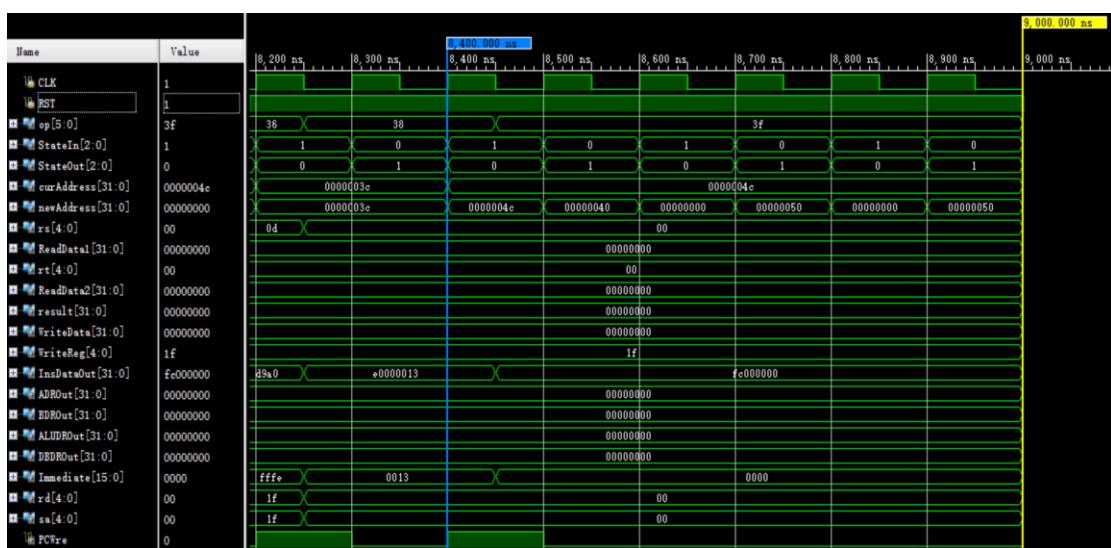
由图可知，该条指令执行了2个时钟周期，当前指令地址为0000003C。

sIF时，InsDataOut的值变为e0000013（为该条指令的16进制数代码）。

sID时，指定CPU将要跳转到的地址0x000004C（由图可知，下一条指令地址确实为00000040）。

综上可知，指令执行完全正确。

### ⑰ halt



由图可知，该条指令执行了2个时钟周期，当前指令地址为0000004C。

sIF时，InsDataOut的值变为fc000000（为该条指令的16进制数代码）。

sID时，PCWre值为0，PC值不更新，CPU将继续处于当前地址0000004C（由图可知，下一条指令地址确实保持为0000004C，且在后续的时钟周期也一直保持不变，Halt实现）

综上可知，指令执行完全正确。

### (3) CPU在Basys3板上的运行情况

#### a) 烧板设计

##### ① 时钟分频：

我们知道Basys3的W5引脚提供了时钟输入，但是该时钟输入频率很高，所以需要进行分频处理。

```
`timescale 1ns / 1ps

module ClkDiv(
    input CLK,      // 时钟信号
    input RST,      // 重置信号
    output divClk   // 分频后的时钟信号
);

reg [31:0] temp;
always@(posedge CLK or negedge RST) begin
    if(!RST) begin
        temp <= 0;
    end
    else begin
        temp <= temp + 1;
    end
end

assign divClk = temp[17];

endmodule
```

不管怎么说，我们的根本目的是降低时钟信号的频率。那么我们可以采用如上方法做到这一点：使用一个寄存器，每次原始时钟信号的上升沿对寄存器进行递增，检查寄存器的指定位，并将其作为新的时钟信号，这就已经实现了分频/降频。

##### ② 通过按键来控制CPU的运行：

显然，为了在Basys板上清楚直观并且方便地检查CPU的运行情况，我们最好通过按键来控制CPU的运行。

```
`timescale 1ns / 1ps

module PressKey(
    input divClk,    // 时钟信号
    input Key,       // 该输入直接与板上的Button 相关（定为 T17）
    output reg Led   // 输出作为一个信号，用于更新Led
);

    always@(posedge divClk) begin
        if(!Key) begin
            Led <= 0;
        end
        else begin
            Led <= 1;
        end
    end
endmodule
```

我们可以通过以上方法来实现。

因为按键时存在按键抖动的情况，所以我们不能直接把按键输入（Key）作为CPU的时钟输入。为解决按键抖动问题，我们可以是按键输入的检查受时钟时序信号的触发。

时钟虽然经过分频，但实际上时钟频率依然非常高，我们把分频后的时钟作为敏感信号依然可以保证按键输入的检查是非常频繁的。

在高频率的按键输入检查下，**如果按键未按下**，那么按键输入（Key）为0，我们输出（Led）0；**如果按键开始按下了（但还未松开）**，此时按键输入（Key）从0变为1，输出（Led）从0变为1，等到我们最后再**松开按键**，此时按键输入（Key）重新从1变为0，输出（Led）从1变为0。

所以，按下按键就对应着CPU时钟输入的上升沿，松开按键就对应着CPU时钟输入的下降沿。

### ③ 数据选择显示

为了能够较为清楚地了解CPU中指令的运行情况（像我们看前面的仿真波形一样），我们需要直观地显示一些数据，但是在Basys3板上只有一个4

位的数码管，所以，为了显示更多的数据，我们需要设计一个数据选择显示模块。只要我们设置好选择输入，就能在LED上选择到自己想看到的数据。

```

`timescale 1ns / 1ps

module DispData(
    input [2:0] Selector,
    input [7:0] curAddress,
    input [7:0] newAddress,
    input [4:0] rs,
    input [7:0] ReadData1,
    input [4:0] rt,
    input [7:0] ReadData2,
    input [7:0] result,
    input [7:0] WriteData,
    input [2:0] StateIn,
    input [2:0] StateOut,
    input PCWre,
    input [1:0] PCSrc,
    input [7:0] ADROut,
    input [7:0] BDROut,
    input [7:0] ALUDROut,
    input [7:0] DBDROut,
    output reg [15:0] out      // 选择后的输出
);

initial begin
    out <= 0;
end

always@(Selector or curAddress or newAddress or rs or
ReadData1
    or rt or ReadData2 or result or WriteData) begin

    case (Selector)
        3'b000: begin
            out[15:8] = curAddress;
            out[7:0] = newAddress;
        end
        3'b001: begin
            out[15:13] = 0;
            out[12:8] = rs;
            out[7:0] = ReadData1;
        end
    end
end

```

```

    3'b010: begin
        out[15:13] = 0;
        out[12:8] = rt;
        out[7:0] = ReadData2;
    end
    3'b011: begin
        out[15:8] = result;
        out[7:0] = WriteData;
    end
    3'b100: begin
        out[15:11] = 0;
        out[10:8] = StateIn;
        out[7:3] = 0;
        out[2:0] = StateOut;
    end
    3'b101: begin
        out[15:9] = 0;
        out[8] = PCWre;
        out[7:2] = 0;
        out[1:0] = PCSrc;
    end
    3'b110: begin
        out[15:8] = ADROut;
        out[7:0] = BDROut;
    end
    3'b111: begin
        out[15:8] = ALUDROut;
        out[7:0] = DBDROut;
    end
endcase
end

endmodule

```

选择输入：

- 000：数码管高两位显示当前地址curAddress，低两位显示下一条执行指令的地址newAddress
- 001：数码管高两位显示rs，低两位显示rs数据ReadData1
- 010：数码管高两位显示rt，低两位显示rt数据ReadData2
- 011：数码管高两位显示ALU运算结果result，低两位显示将被写入到寄存器的数据WriteData

- 100: 数码管高两位显示CPU下一状态StateIn，低两位显示CPU当前状态StateOut
- 101: 数码管高两位显示PC更新控制信号PCWre，低两位显示PC更新选择信号PCSrc
- 110: 数码管高两位显示ADR内存储的数据ADROut，低两位显示BDR内存储的数据BDROut
- 111: 数码管高两位显示ALUDR内存储的数据ALUDROut，低两位显示DBDR内存储的数据DBDROut

#### ④ 数码管扫描显示

我们无法做到在一个4位数码管上同时显示多个不同的数。要做到这一点，就需要借助扫描显示，每次只显示1个位，然后在极短的时间后又显示下一个位，利用人眼的视觉暂留效应，让人看起来好像同时显示了多位数字一样。

既然是扫描显示，那么我们的第一个问题是选位。

```
`timescale 1ns / 1ps

module Display(
    input divClk,
    input RST,
    input [15:0] Out,
    output reg [3:0] AN,
    output reg [6:0] DispCode
);

// 选择要显示的位
reg[1:0] selectPos;
always@(posedge divClk or negedge RST) begin
    if(!RST) begin
        selectPos <= 0;
    end
    else begin
        selectPos <= selectPos + 1;
    end
end
end
```

我们的LED只有4位，所以selectPos只需要2位就足够显示0、1、2和3共四个数字。divClk虽然经过分配，但频率仍然非常高，所以selectPos将会

在0~3这四个数字中迅速地扫描，而0~3数字对应着LED的4个位，这意味着我们实际上在通过高速的选位过程来进行LED的扫描。

选位完成后，我们需要得出具体需要显示的数是什么。

```
// 根据已选择的位找出要显示的数
reg [3:0] digits;
always@(*) begin
    case(selectPos)
        0: digits = Out[3:0];
        1: digits = Out[7:4];
        2: digits = Out[11:8];
        3: digits = Out[15:12];
        default: digits = Out[3:0];
    endcase
end
```

不管需不需要扫描显示，我们最基本的任务显示依然离不开译码。这里我们最终要把数据显示在七段数码管上，这需要我们对选出的数进行译码操作。

```
// 对选出的数进行7位译码
always@(*) begin
    case(digits)
        // '0' - 相应段点亮, '1' - 相应段熄灭
        4'h0: DispCode = 7'b0000001;           //0
        4'h1: DispCode = 7'b1001111;           //1
        4'h2: DispCode = 7'b0010010;           //2
        4'h3: DispCode = 7'b0000110;           //3
        4'h4: DispCode = 7'b1001100;           //4
        4'h5: DispCode = 7'b0100100;           //5
        4'h6: DispCode = 7'b0100000;           //6
        4'h7: DispCode = 7'b0001111;           //7
        4'h8: DispCode = 7'b0000000;           //8
        4'h9: DispCode = 7'b0000100;           //9
        4'hA: DispCode = 7'b0001000;           //A
        4'hB: DispCode = 7'b1100000;           //B
        4'hC: DispCode = 7'b0110001;           //C
        4'hD: DispCode = 7'b1000010;           //D
        4'hE: DispCode = 7'b0110000;           //E
        4'hF: DispCode = 7'b0111000;           //F
        default: DispCode = 7'b1111111;         //全熄灭
    endcase
end
```

最后，我们需要提供数码管输入，要做到扫描显示，对于阴极的Basys3学习板上的LED，我们提供的数码管输入应该高速进行在0111、1011、1101

和1110这四个数的切换，其中0的位置取决于我们最开始的选位。

```
// 更新数码管输入，使其只显示选中位
// '0'-相应位会显示，'1'-相应位不被显示
always@(*)begin
    AN = 4'b1111;
    AN[selectPos] = 0;
end

endmodule
```

### b) CPU在Basys3板上的运行情况

T17:按钮控制CPU运行,V17:Reset,[R2 T1 U1]选择LED数据显示(e.g.R2 = T1 = U1 = 0时，LED显示curAddress和newAddress)，R2为高位，U1为低位。

以下，对于每条指令的每个状态，给出8张图片，格式如下：

[curAddress newAddress] [rs ReadData1] [rt ReadData2] [result WriteData]  
[StateIn StateOut] [PCWre PCSrc] [ADROut BDROut] [ALUDROut DBDROut]

LED上显示的数据在仿真的波形图中也能看到，对比给定指令下这些输出的数据和仿真的数据是否相同即可。我们在报告里只显示随机在板上测试的5条指令。

② ori \$2,\$0,2

➤ sIF



The image shows two digital displays. The top display shows the address bus with the value 0404000002000000. The bottom display shows the data bus with the value 0100000000000000.

➤ sID



The image shows two digital displays. The top display shows the address bus with the value 0408000002000000. The bottom display shows the data bus with the value 0201000200000000.

➤ sEXE



The image shows a single digital display showing the address bus with the value 0400000002000202.

0302 0000 0000 0202

➤ sWB

0408 0000 0202 0000

0003 0000 0000 0000

⑥ sll \$5,\$5,2

➤ sIF

14 14 0000 0502 0204

0100 0000 0002 0404

➤ sID

14 18 0000 0502 0202

0201 0002 0002 0202

➤ sEXE

1400 0000 0502 0808

0302 0000 0002 0808

➤ sWB

14 18 0000 0508 0202

0003 0000 0002 0202

⑧ jal 0x0000040

➤ sIF

1C 1C 0000 0000 0028

0100 0000 0000 2828

➤ sID

1C 20 0000 0000 0020

0001 0003 0000 0000

⑩ lw \$12,4(\$1)

➤ sIF

4444	0108	0C00	0808
0100	0000	0800	0808

➤ sID

4448	0108	0C00	0808
0201	0002	0800	0808

➤ sEXE

4408	0108	0C00	0C0C
0402	0000	0800	0C0C

➤ sMEM

4448	0108	0C00	0800
0304	0000	0800	0800

➤ sWB

4448	0108	0C00	0808
0003	0000	0800	0808

⑯ bltz \$13,-2 (第一次执行, <0, 转34)

➤ sIF

3B38	0dFF	0000	FFFF
0100	0000	FF00	FCFC

➤ sID

3B3C	0dFF	0000	FFFF
0201	0002	FF00	FFFF

➤ sEXE

**38FF 0dFF 0000 0101**

**0002 0001 FF00 0101**

对比以上5条指令和相应的仿真波形图，全部一致。CPU在Basys3学习板上的测  
完毕。测试得出CPU设计正确。

## 六. 实验心得

本次的多周期CPU设计基本上基于先前的单周期CPU。在这次实验中，我更加深入地  
学习和使用了Verilog来进行硬件编程，同时也学习了多周期CPU的具体原理，不仅理解了  
多周期CPU和单周期CPU的区别，还通过对比两次实验加深了多周期CPU和单周期CPU的  
理解。

单周期CPU一个时钟周期执行一条指令，在设计时我不需要特别清楚一条指令执行的  
具体细节；而多周期CPU需要多个时钟周期来执行一条指令，这就使得我在设计时必须对  
指令进行分析然后将其分解为几个部分以分配给多个时钟周期。在单周期CPU设计时，我  
虽然在报告中定义了sIF、sID、sEXE、sMEM和sWB，但我在设计时实际上并没有非常  
仔细地去思考一条指令执行的具体过程；而现在在多周期CPU设计里，你必须明白一条指  
令的具体执行过程：状态从头到尾如何转换，每个状态下CPU的那些部件需要工作等等，  
这大大加深了我对单周期CPU的理解。

多周期CPU设计依然是使用模块化的思想，读懂那副多周期CPU控制线路和数据通路  
图，并将其分解为9类模块，其中部分模块完全与单周期CPU的模块相同，可以直接使用单  
周期CPU的模块。

虽然，多周期CPU设计基本上基于先前的单周期CPU，但在实验过程中还是遇到了很  
多麻烦，遇到麻烦的部分主要还是在烧板验证阶段：在仿真阶段测试程序的仿真波形完全正  
确，但是烧到板子上以后就出现种种问题了，要么是地址不正常地更新，要么是第一条指令  
没有正确地读入。想要解决这些问题，仿真波形用处很少，因为仿真波形本身是正确的，为了  
真正解决这些问题，我不得不把很多控制信号在Basys3学习板上显示出来，对这些控制  
信号进行分析来找到问题的根源。

这种仿真和烧板上结果的不对等体现了理想情况和现实情况上的差别，相比于仿真的理  
想情况，烧板验证对控制信号的准确性要求更高。比如，在仿真时，一个PCWre在CLK上

升沿的时候变为1，在这个CLK周期内，CPU的PC的值如期望那般更新，在整个时钟周期结束后，PCWre重新变为0，在下一个时钟周期，PC的值保持不变，这是理想情况；但是在烧板时，PC的值变了两次，一次是在本次CLK周期内，CPU的PC的值如期更新，第二次是在下个时钟周期到来时，PCWre的值没有及时变为0，这导致PC的值在下个时钟周期的上升沿再次更新了一次，其后PCWre的值才变为0。为解决这个问题，我通过使用中间变量和组合逻辑来保证PCWre在CLK上升沿变为1，然后只保持半个CLK周期而不是整个CLK周期。

本次实验其他的一个难点就是延迟问题了，不过这个相对来说还是比较好解决的。

在烧板上踩了无数的坑之后，我深刻意识到理想和现实的差距，也让我明白控制信号表设计的一定要非常严谨非常精确，控制信号的严谨和精确保证了CPU运行的精确，同时这种严谨和精确也会便于我们Debug，毕竟有些错误很可能不是因为一个原因造成的。