



《计算机组成原理实验》 实验报告

(实验四)

学 院 名 称 : 数据科学与计算机学院

专业 (班级) : 18 计教学 3 班

学 生 姓 名 : 谢俊杰

学 号 : 18340181

时 间 : 2019 年 12 月 9 日

成绩：

实验四：多周期CPU设计与实现

一. 实验目的

1. 认识和掌握多周期数据通路图的构成、原理及其设计方法；
2. 掌握多周期CPU的实现，代码实现方法；
3. 认识和掌握指令与CPU的关系；
4. 掌握多周期CPU的测试方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。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)	00000 100000
--------	---------	---------	---------	--------------

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

(2) sub rd, rs, rt

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

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

(3) addiu rt, rs, **immediate**

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

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

(4) addi rt, rs, **immediate**

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

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

==> 逻辑运算指令

(5) andi rt, rs, **immediate**

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

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

(6) and rd, rs, rt

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

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

(7) ori rt, rs, **immediate**

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

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

(8) or rd, rs, rt

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

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

(9) sll rd, rt, sa

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

功能: $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rt}] \ll \text{sa}$ 。**==>比较指令**(10) slti rt, rs, **immediate** 带符号数

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

功能: if $\text{GPR}[\text{rs}] < \text{sign_extend}(\text{immediate})$ $\text{GPR}[\text{rt}] = 1$ else $\text{GPR}[\text{rt}] = 0$ 。

(11) slt rd, rs, rt

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

功能: if $\text{GPR}[\text{rs}] < \text{sign_extend}(\text{immediate})$ $\text{GPR}[\text{rt}] = 1$ else $\text{GPR}[\text{rt}] = 0$ 。

(12) movn rd, rs, rt

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

功能: if $\text{GPR}[\text{rt}] \neq 0$ then $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}]$ 。**==> 存储器读/写指令**(13) sw rt, **offset** (rs) 写存储器

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

功能: $\text{memory}[\text{GPR}[\text{base}] + \text{sign_extend}(\text{offset})] \leftarrow \text{GPR}[\text{rt}]$ 。(14) lw rt, **offset** (rs) 读存储器

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

功能: $\text{GPR}[\text{rt}] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{sign_extend}(\text{offset})]$ 。(15) lhu rt, **offset** (rs) 读存储器

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

功能: $\text{GPR}[\text{rt}] \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{sign_extend}(\text{offset})]$ 。**==> 分支指令**(16) beq rs, rt, **offset**

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

功能: if $(\text{GPR}[\text{rs}] = \text{GPR}[\text{rt}])$ $\text{pc} \leftarrow \text{pc} + 4 + \text{sign_extend}(\text{offset}) \ll 2$ else $\text{pc} \leftarrow \text{pc} + 4$

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

(17) bne rs, rt, **offset**

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

功能: if(GPR[rs] != GPR[rt]) $pc \leftarrow pc + 4 + \text{sign_extend}(\text{offset}) \ll 2$

else $pc \leftarrow pc + 4$

(18) bltz rs, offset

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

功能: if(GPR[rs] < 0) $pc \leftarrow pc + 4 + \text{sign_extend}(\text{offset}) \ll 2$

else $pc \leftarrow pc + 4$ 。

==>跳转指令

(19) j addr

000010	addr(26 位)		
--------	------------	--	--

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

说明: 由于 MIPS32 的指令代码长度占 4 个字节, 所以指令地址二进制数最低 2 位均为 0, 将指令地址放进指令代码中时, 可省掉! 这样, 除了最高 6 位操作码外, 还有 26 位可用于存放地址, 事实上, 可存放 28 位地址, 剩下最高 4 位由 pc+4 最高 4 位拼接上。

(20) jr rs

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

功能: $PC \leftarrow GPR[rs]$, 跳转。

(21) jal addr

000011	addr(26 位)		
--------	------------	--	--

功能: 调用子程序, $PC \leftarrow \{PC[31:28], \text{addr}, 2' b0\}$; $GPR[\$31] \leftarrow pc+4$, 返回地址设置; 子程序返回, 需用指令 jr \$31。跳转地址的形成同 j addr 指令。

==> 停机指令

(22) halt

111111	0000000000000000000000000000(26 位)		
--------	------------------------------------	--	--

功能: 停机; 不改变 PC 的值, PC 保持不变。

三. 实验原理

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

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

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

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

(4) 存储器访问(MEM): 所有需要访问存储器的操作都将在这个步骤中执行, 该步骤给

出存储器的数据地址,把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

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

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

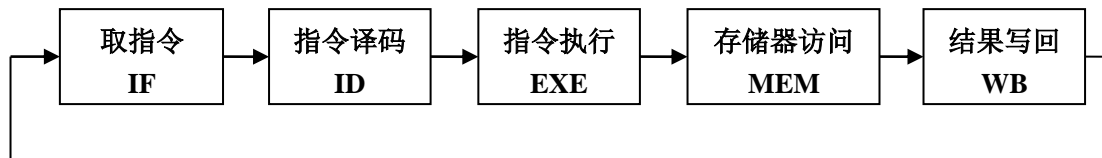


图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式:

R 类型:

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

I 类型:

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

J 类型:

31	26 25	0
op	address	
6 位	26 位	

其中,

op: 为操作码;

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

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

rd: 为目的操作数寄存器,寄存器地址(同上);

sa: 为位移量(shift amt),移位指令用于指定移多少位;

funct: 为功能码,在寄存器类型指令中(R 类型)用来指定指令的功能;

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

address: 为地址。

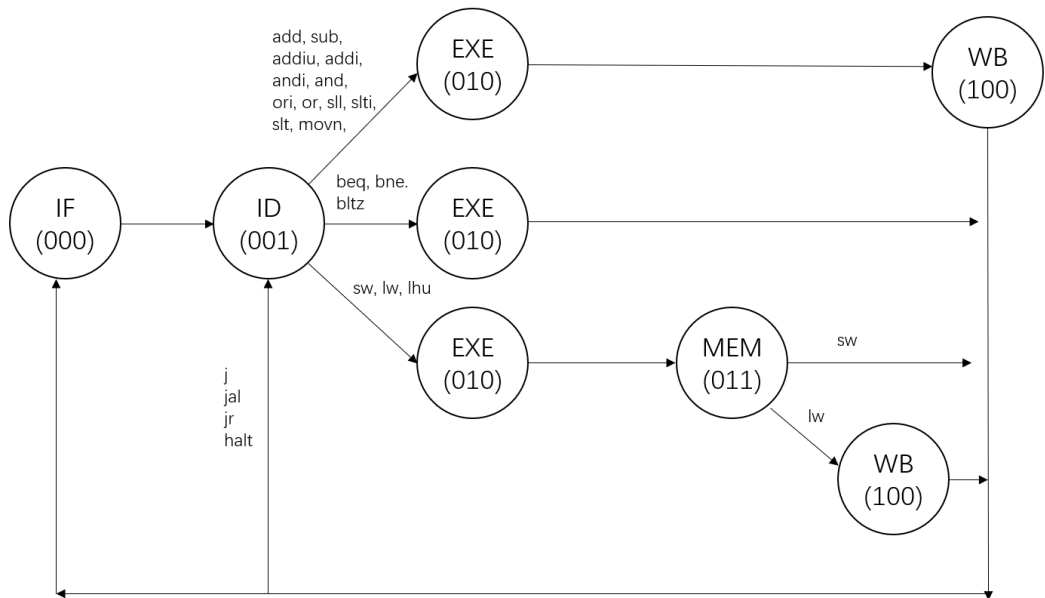


图 2 多周期 CPU 状态转移图

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

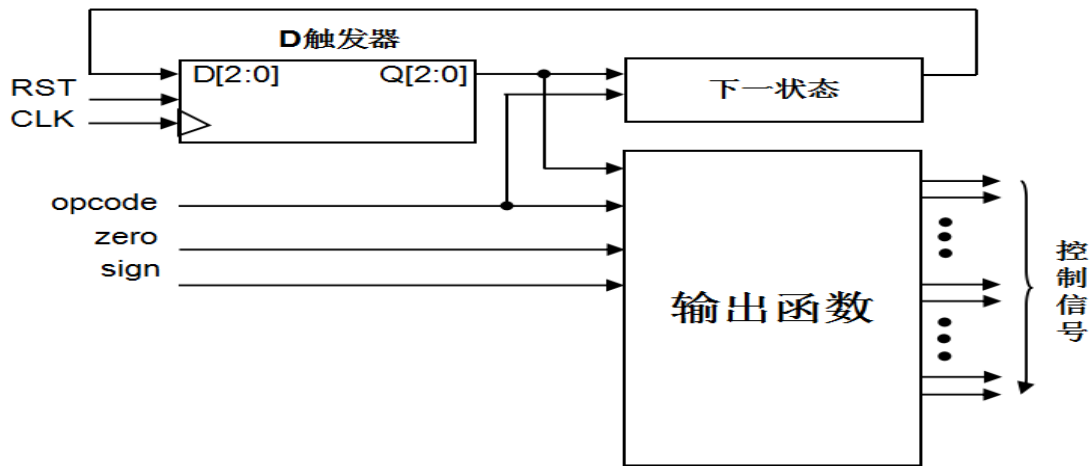


图 3 多周期 CPU 控制部件的原理结构图

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

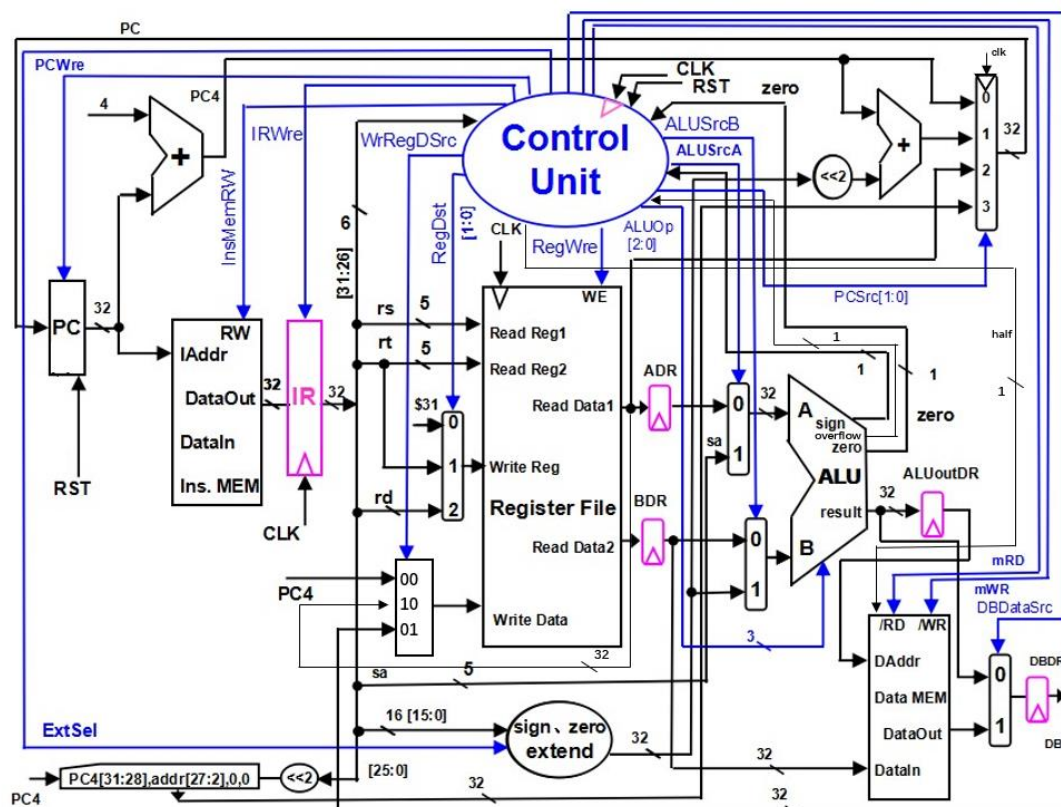


图 4 多周期 CPU 数据通路和控制线路图

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。

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;
PCSrc[1..0]	00: $pc \leftarrow pc+4$, 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: $pc \leftarrow pc+4+(sign-extend)immediate \times 4$, 相关指令: beq(zero=1)、bne(zero=0)、bltz(sign=1); 10: $pc \leftarrow rs$, 相关指令: jr; 11: $pc \leftarrow \{pc[31:28], addr[27:2], 2'b00\}$, 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 ($\$31 \leftarrow pc+4$) ; 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

mRD, 数据存储器读控制信号, 为 1 读

mWR, 数据存储器写控制信号, 为 1 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	$Y = (((A < B) \& \& (A[31] == B[31])) \vee ((A[31] == 1 \& \& B[31] == 0))) ? 1 : 0$	比较 A<B 带符号
111	$Y = B$	movn 用

四. 实验器材

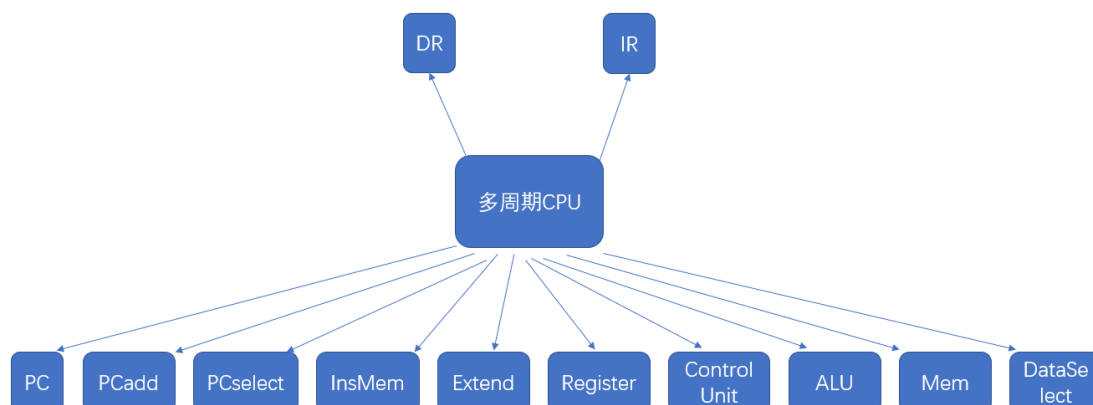
电脑一台, Xilinx Vivado 软件一套, Basys3板一块。

五. 实验过程与结果

1、CPU设计的思想、方法

(1) 在这里, 我采用的是自顶向下的设计方法, 而在有单周期实验的基础下, 我将整个多周期CPU的设计划分为12个具体的功能模块, 分别为PC、PCadd (PC+4)、PCselect (选择PC)、InsMem (指令存储器)、Extend (符号或零扩展)、Register (寄存器组)、

ControlUnit（控制单元）、ALU（运算器）、Mem（数据存储器）、DataSelect（数据选择器）、DR（切分数据通路的寄存器）、IR（指令寄存器），即如图所示：



在实现过程中，每个功能模块的实现只简单关心其输出和输入即其内部的逻辑关系（具体的参考数据通路图），而暂时忽略各模块间的联系。

然后将各功能模块的功能具体实现后封装成IP核（每个模块就是一个IP核），再根据数据通路的图，设计一个顶层文件，在该文件中调用上述IP核并将相应接口相连，但是要注意的是需要进行单元测试，如果在顶层设计的仿真测试出现问题时，会耗费大量时间去debug，所以我对每个功能模块在实现后都会进行仿真测试，以保证其独立时的正确性。

（2）其次，从数据通路的图上可以看出，各功能模块在各阶段都需要控制单元产生的不同控制信号组合来进行相应指令的相应操作，同时控制单元也会接受信号如操作码、功能码从而产生相应的控制信号。

而对于多周期不同状态间的转移，合并到控制单元模块之中，使用D触发器保存当前状态和根据现阶段的状态和控制信号去生成下一个状态，是时序逻辑电路。即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、当前状态和一些反馈标志信号。所以，这里需要将指令、操作码（、功能码）、操作、控制信号和状态对应建立一个真值表（在控制单元模块说明处示出）。

（3）各模块的具体实现和说明

1) PC模块

功能：更新当前PC值

输入：PC写使能信号PCWre、复位信号Reset、下个PC的地址nexPC

输出：当前PC的地址curPC

PC的值的改变受PCWre和Reset两者影响：当Reset为0时，当前PC值就会被初始化为0；当Reset不为0时，PCWre为1时，改变当前PC值，即curPC等于nexPC。

本来PC模块是由clk的时序敏感信号触发的，但是由于在仿真时发现clk上升沿来临时PCWre没有及时变为1，跳转指令也不能及时地在第一个取指阶段跳转到正确的指令，故将clk去掉并以PCWre上升沿和ReSet的下降沿作为电平敏感信号触发PC的改变。具体代码为：

```
initial
    curPC = 0;
always @(posedge PCWre or negedge Reset)
begin
    if (Reset == 0)
        curPC = 0;
    else
        if (PCWre == 1)
            curPC = nexPC;
```

2) PCadd模块

功能：将输入的PC值+4

输入：输入的当前PC值

输出：输出的PC+4的值

由于该模块仅是组合逻辑电路，将输出定义为wire类型，使用assign语句并使用阻塞赋值。具体代码如下：

```
assign o_PC1 = i_PC1 + 4;
```

3) PCselect模块

功能：根据对输入的数据进行运算和选择，输出下一条PC地址

输入：时钟信号clk、输入的已+4的PC、PCSrc、偏移量立即数immediate、地址address

输出：选择得到的下一条PC地址

该模块的主要逻辑实则是4选1数据选择器，根据PCSrc的值去选择下一条PC的地址，具体的选择如下表：

PCSrc[1:0]	nexPC
00	PC<-PC+4

01	$PC \leftarrow PC + 4 + \text{immediate} * 4$
10	$PC \leftarrow rs$
11	$PC \leftarrow \{(PC + 4)[31:28], \text{address}[25:0], 2'b00\}$

注意到该表中右边的 $(PC+4)$ 就是输入的PC地址。同时,输入的immediate是从 $PC+4$ 地址开始和转移到的指令之间指令条数,在计算地址值时需要左移2位即乘以4。

但是,原本该模块是没有clk输入的,但是在仿真没问题的基础上烧板子时出现了问题,主要是在指令执行到一定数量后时延积累,使得部分指令的PCSrc在PCWre改变后再改变,使得指令更新为错误的指令,所以在此我加入clk信号使得PCSrc稳定下来待确认写入后再发生改变,确保在写入时仍然为期望值。

而在这里使用时钟下降沿的原因是,在仿真和上板子实现时,使得一些跳转指令也能正常地在最后所处的状态也能在nexPC显示即将要跳转的地址,以便更好地检查正确性。

具体代码如下:

```

initial
    o_PC2 = 0;
always @(negedge clk)
begin
    case (PCSrc)
        2'b00: o_PC2 = i_PC2;
        2'b01: o_PC2 = i_PC2 + immediate * 4;
        2'b11:
            begin
                PC_high4 = i_PC2;///
                o_PC2 = {PC_high4[31:28], address[25:0], 2'b00}
            end
        default: o_PC2 = rs_data;
    endcase
end

```

4) InsMem模块

功能: 根据输入的当前PC地址,在预先存储指令的指令存储器中寻找到相应指令的二进制码并输出。

输入: 指令寄存器的读使能信号InsMemRW、当前PC地址i_addr

输出: 相应指令的二进制码o_ins。

由于需要存储的指令条数为49条,而每条指令为32位,根据要求,指令存储器存储单元宽度为8位,所以指令存储器中至少需要196个存储单元,而在此我使用了256个8位存储

单元的指令存储器memory:

```
reg [7:0]memory[0:255];
```

在模块开始运行时，从txt文件读入程序指令到指令存储器memory中，注意到MIPS的大端存储方式，指令高位放在低地址上：

```
initial
begin
    $readmemb("D:/code/freshman/digital/multiCPU/ins.txt", memory);
end
```

再根据输入的PC地址i_addr在指令存储器进行定位，当InsMemRW为1时取出当前相应的指令，由于MIPS是大端方式存储，高位存放在低地址，故低地址的数据拼接在高位：

```
always @(*)
begin
    if (InsMemRW == 1)
        o_ins = {memory[i_addr][7:0], memory[i_addr+1][7:0],
                  memory[i_addr+2][7:0], memory[i_addr+3][7:0]};
end
```

5) Extend模块

功能：根据符号/零选择信号对输入数据进行对应的符号/零扩展

输入：16位立即数i_immediate、选择信号ExtSel

输出：32位扩展后的立即数o_imme

该模块主要逻辑也是一个2选1数据选择器，当ExtSel为0时，高位补0，进行零扩展；

当ExtSel为1时，高位补符号位，进行符号扩展。

```
always @(*)
begin
    if (ExtSel == 0)
        o_imme = {16'h0000, i_immediate[15:0]};
    else
        begin
            if (i_immediate[15] == 0)
                o_imme = {16'h0000, i_immediate[15:0]};
            else
                o_imme = {16'hffff, i_immediate[15:0]};
        end
    end
end
```

6) Register模块

功能：给出两个寄存器的地址，对这两个寄存器进行读取数据的操作，同时给出要写入

寄存器的地址和要写入的数据，在写使能信号为1且在时钟下降沿处，将数据写入相应寄存器中。

输入：时钟clk、写使能信号RegWre、读取的寄存器地址ReadReg1、ReadReg2、写入的寄存器地址WriteReg、写入的数据WriteData

输出：读取到的寄存器中的数据ReadData1、ReadData2

首先，在该模块开始运行时，需要对开辟的寄存器组（32个32位存储单元）进行初始化为0：

```
reg [31:0]Reg[0:31];
integer i;

initial
begin
    for (i = 0; i < 32; i = i + 1)
        Reg[i] <= 0;
end
```

其次，由于读操作时不需要时钟信号，输出端可以直接输出相应数据；而对于写操作，由于0号寄存器不能被更改，故当写使能信号为1且要写入的寄存器地址不为0时，则可以在时钟上升沿触发写入相应数据，否则不能写入。

```
assign ReadData1 = Reg[ReadReg1];
assign ReadData2 = Reg[ReadReg2];

always @(posedge clk)
begin
    if (WriteReg != 0 && RegWre == 1)
        Reg[WriteReg] <= WriteData;
end
```

7) ControlUnit模块

功能：根据6位操作码、6位功能码、零信号、符号信号、溢出信号和当前状态去选择生成各控制信号的值，且在时钟上升沿触发根据当前所处状态和当前指令生成下一状态，使其他模块能够按照指令正常地运行。

输入：时钟clk、零信号zero、符号信号sign、溢出信号overflow、6位操作码op、6位功能码funct、复位信号ReSet

输出：真值表中除输入外的信号

状态的转移发生在在控制单元模块内，故在控制单元内部声明两个3位的变量，分别为

当前状态state和下一状态nex_state，而状态的转移需要时钟上升沿触发，且在ReSet为0时，需要将状态赋为000，即取指阶段；若ReSet为1时，则当前状态赋为下一状态。

```

reg [2:0]state;
reg [2:0]nex_state;

always@(posedge clk)
begin
    if (Reset == 0) begin
        state <= 3'b000;
        led <= 5'b00001;
    end
    else begin
        state <= nex_state;
    end
end

```

同时，下一状态取决于当前指令操作码（、功能码）和当前状态，而不同指令在不同状态的控制信号是由指令、当前状态和一些标志信号决定的。由于需要处理的信号较多，故在代码中使用case语句对各信号根据下真值表逐项进行赋值。部分代码如下：

```

3'b001: begin//sID
    case (op)
        6'b000010: begin//j
            PCWre = 0;
            RegWre = 0;
            InsMemRW = 0;
            mRD = 0;
            mWR = 0;
            IRWre = 0;
            PCSrc = 2'b11;
            nex_state = 3'b000;
        end
    end

```

真值表如下：

state	ins	op	funcnt	Reset	zero	sign	overflow	half	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	WtRegDS	InsMemRW	mRD	mWR	IRWre	ExtSel	PCSrc[1:0]	RegDst[1:0]	ALUOp[2:0]
000(IF)	halt	111111		1	x	x	x	x	0	x	x	x	0	x	1	0	0	1	x	xx	xx	xxx
	xxxx	xxxxxx		1	x	x	x	x	1	x	x	x	0	x	1	0	0	1	x	00	xx	xxx
001(ID)	j	00010		1	x	x	x	x	0	x	x	x	0	x	0	0	0	0	x	11	xx	xxx
	jal	00011		1	x	x	x	x	0	x	x	x	1	00	0	0	0	0	x	11	00	xxx
	jr	00000	001000	1	x	x	x	x	0	x	x	x	0	x	0	0	0	0	x	10	xx	xxx
	halt	111111		1	x	x	x	x	0	x	x	x	0	x	0	0	0	0	x	xx	xx	xxx
	xxxx	xxxxxx		1	x	x	x	x	0	x	x	x	0	x	0	0	0	0	x	00	xx	xxx
010(EXE)	add	00000	100000	1	x	x	x	x	0	0	0	0	0	x	0	0	0	0	x	00	xx	000
	sub	00000	100010	1	x	x	x	x	0	0	0	0	0	x	0	0	0	0	x	00	xx	001
	and	00000	100100	1	x	x	x	x	0	0	0	0	0	x	0	0	0	0	x	00	xx	100
	or	00000	100101	1	x	x	x	x	0	0	0	0	0	x	0	0	0	0	x	00	xx	011
	sll	00000	000000	1	x	x	x	x	0	1	0	0	0	x	0	0	0	0	x	00	xx	010
	slt	00000	101010	1	x	x	x	x	0	0	0	0	0	x	0	0	0	0	x	00	xx	110
	movn	00000	001011	1	x	x	x	x	0	0	0	0	0	x	0	0	0	0	x	00	xx	111
	addiu	001001		1	x	x	x	x	0	0	1	0	0	x	0	0	0	0	1	00	xx	000
	addi	001000		1	x	x	x	x	0	0	1	0	0	x	0	0	0	0	1	00	xx	000
	andi	001100		1	x	x	x	x	0	0	1	0	0	x	0	0	0	0	0	00	xx	100
	ori	001101		1	x	x	x	x	0	0	1	0	0	x	0	0	0	0	0	00	xx	011
	slti	001010		1	x	x	x	x	0	0	1	0	0	x	0	0	0	0	1	00	xx	110
	beq	000100		1	1	x	x	x	1	0	0	0	x	0	x	0	0	0	1	01	xx	001
				1	0	x	x	x	1	0	0	0	x	0	x	0	0	0	1	01	xx	001
	bne	000101		1	0	x	x	x	0	0	0	0	x	0	x	0	0	0	1	01	xx	001
				1	1	x	x	x	0	0	0	0	x	0	x	0	0	0	1	01	xx	001
	bltz	000001		1	x	1	x	x	0	0	0	0	x	0	x	0	0	0	1	01	xx	001
				1	x	0	x	x	0	0	0	0	x	0	x	0	0	0	1	00	xx	001
	sw	101011		1	x	x	x	x	0	0	1	x	0	x	0	0	0	0	1	00	xx	000
	lw	100011		1	x	x	x	x	0	0	1	x	0	x	0	0	0	0	1	00	xx	000
	lhu	100101		1	x	x	x	x	0	0	1	x	0	x	0	0	0	0	1	00	xx	000
011(MEM)	sw	101011		1	x	x	x	x	0	x	x	x	0	x	0	0	1	0	x	00	xx	xxx
	lw	100011		1	x	x	x	0	0	x	x	1	0	0	x	0	1	0	x	00	xx	xxx
	lhu	100101		1	x	x	x	1	0	0	x	x	1	0	x	0	1	0	x	00	xx	xxx
100(WB)	lw	100011		1	x	x	x	0	0	x	x	x	1	01	0	0	0	0	x	00	01	xxx
	lhu	100101		1	x	x	x	0	0	x	x	x	1	01	0	0	0	0	x	00	01	xxx
	xxxx	000000		1	x	x	0	x	0	x	x	x	1	01	0	0	0	0	x	00	10	xxx
	add	000000	100000	1	x	x	x	1	x	0	x	x	x	0	x	0	0	0	0	00	xx	xxx
	sub	000000	100010	1	x	x	x	1	x	0	x	x	x	0	x	0	0	0	0	00	xx	xxx
	addi	001000		1	0	x	x	1	x	0	x	x	x	0	x	0	0	0	0	00	xx	xxx
	movn	000000	001011	1	0	0	x	x	0	0	x	x	x	1	10	0	0	0	x	00	10	xxx
				1	1	x	x	x	0	0	x	x	x	0	xx	0	0	0	0	00	xx	xxx
	else			1	x	x	x	x	0	x	x	x	1	01	0	0	0	0	x	xx	01	xxx

8) ALU模块

功能：根据控制单元模块生成的ALUOp信号，选择相应的算术逻辑运算，同时根据运

算后得到的结果对zero、sign和overflow输出信号赋值，从而反馈给控制单元。

输入：操作数A、操作数B和算术逻辑选择信号ALUOp

输出：结果result、零信号zero、符号信号sign

算术逻辑选择信号和相应的算术逻辑是一一对应的关系，具体如下表所示（该表来源于要求文档）。

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

而对于生成的信号，当result为0时，zero信号为1，反之为0；而当result<0时，即result的最高位为1时，sign信号为1，反之为0，具体代码如下：

```

always @(*)
begin
    case(ALUOp)
        3'b000: begin
            result = input_A + input_B;
            overflow = ((input_A[31] & input_B[31] & (~result[31])) |
                        ((~input_A[31]) & (~input_B[31]) & result[31]));
        end
        3'b001: begin
            result = input_A - input_B;
            overflow = ((input_A[31] & (~input_B[31]) & (~result[31])) |
                        ((~input_A[31]) & input_B[31] & result[31]));
        end
        3'b010: result = input_B << input_A;
        3'b011: result = input_A | input_B;
        3'b100: result = input_A & input_B;
        3'b101: result = (input_A < input_B) ? 1 : 0;
        3'b110: result = (((input_A < input_B) && (input_A[31] == input_B[31])) ||
                        ((input_A[31] == 1) && (input_B[31] == 0))) ? 1 : 0;
        3'b111: result = input_B;
        default: result = 0;
    endcase
end

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

```


9) Mem模块

功能：根据相应控制信号，读取对应地址的数据或由对应地址对存储器进行写入数据操作。

输入：时钟clk、读使能信号mRD、写使能信号mWR、数据地址DataAddr、要写入的数据DataIn

输出：读取出的相应的数据DataOut

在该模块中，首先申请128个8位存储单元的空间作为存储器的空间，然后初始化为0。

```
reg [7:0]memory[0:127];
integer i;
initial
begin
    for (i = 0; i < 128; i = i + 1) //初始化
        memory[i] <= 0;
end
```

在时钟下降沿来临时且写使能信号为1时，将输入的32位数据拆分为4个8位的数据，由大端方式存储到存储器中，高位存放在低地址；在读使能信号为1时，将相应地址的数据拼接成为一个32位的数据输出。具体代码如下：

```

always @(*)
begin
    if (mWR == 1)
    begin
        memory[DataAddr] <= DataIn[31:24];
        memory[DataAddr + 1] <= DataIn[23:16];
        memory[DataAddr + 2] <= DataIn[15:8];
        memory[DataAddr + 3] <= DataIn[7:0];
    end
end

always @(*)
begin
    if (mRD == 1)
    begin
        if (half == 0)
            DataOut <= {memory[DataAddr][7:0],
                        memory[DataAddr + 1][7:0],
                        memory[DataAddr + 2][7:0],
                        memory[DataAddr + 3][7:0]};
        else
            DataOut <= {16'h0000, memory[DataAddr][7:0],
                        memory[DataAddr + 1][7:0]};
        end
    else
        DataOut <= 32'hzzzzzzzz;
    end
end

```

10) DataSelect模块

功能：数据选择器，根据信号译码选择数据

输入：信号signal、两个数据input_A、input_B

输出：选择的其中一个输入数据

该模块主要是数据选择器，将需要多个选择输入数据的模块连接起来。由于在该多周期CPU数据通路中需要用到较多数据选择器，且部分基本大同小异，故取其中一个二选一数据选择器的具体代码如下：

```

assign output_C2 = (signal2 == 0) ? input_A2 : input_B2;

```

11) DR（数据寄存器）

功能：将数据通路切分，在多周期CPU运行期间存储对应阶段结束后的一些数据项，在下一状态来临时和clk上升沿触发，可直接输入数据使用，避免大延迟的发生。

输入：时钟信号clk、输入的数据input_A

输出：输出的数据output_C

在数据通路图中，对应的DR有ADR、BDR、ALUoutDR、DBDR，分别用于译码、译码、执行、访存阶段后的数据存储。具体代码如下：

```
always @(posedge clk)
begin
    output_C <= input_A;
end
```

12) IR (指令寄存器)

功能：存储当前正在执行的指令，并根据R型、I型、J型指令完成译码功能，即将当前指令的各部分分别取出并输出。

输入：时钟信号clk、指令二进制码instruction、写使能信号IRWre

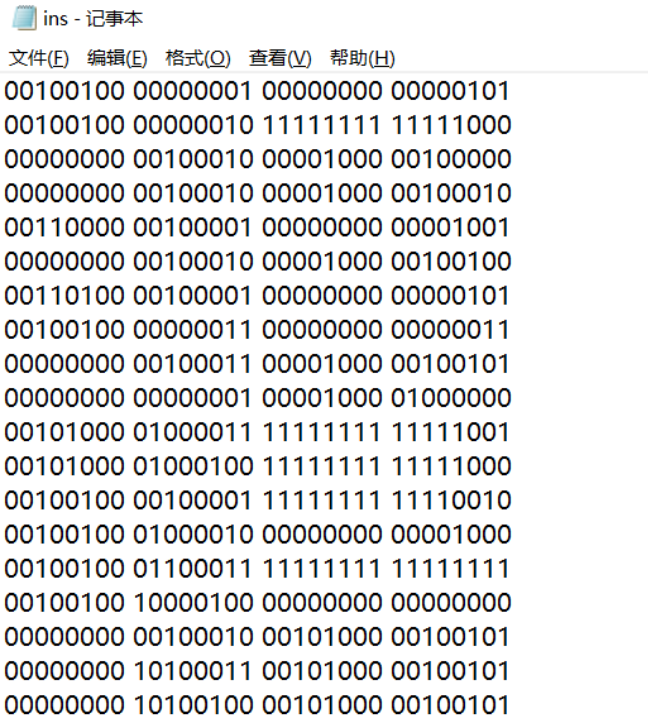
输出：指令各部分划分的数据集合（如op、rs、rt、rd等）

当IRWre为1，且clk处于上升沿时，将指令根据R、I、J型指令的不同格式，拆分各段数据输出，否则数据通路中的译码后的指令不变。具体代码为：

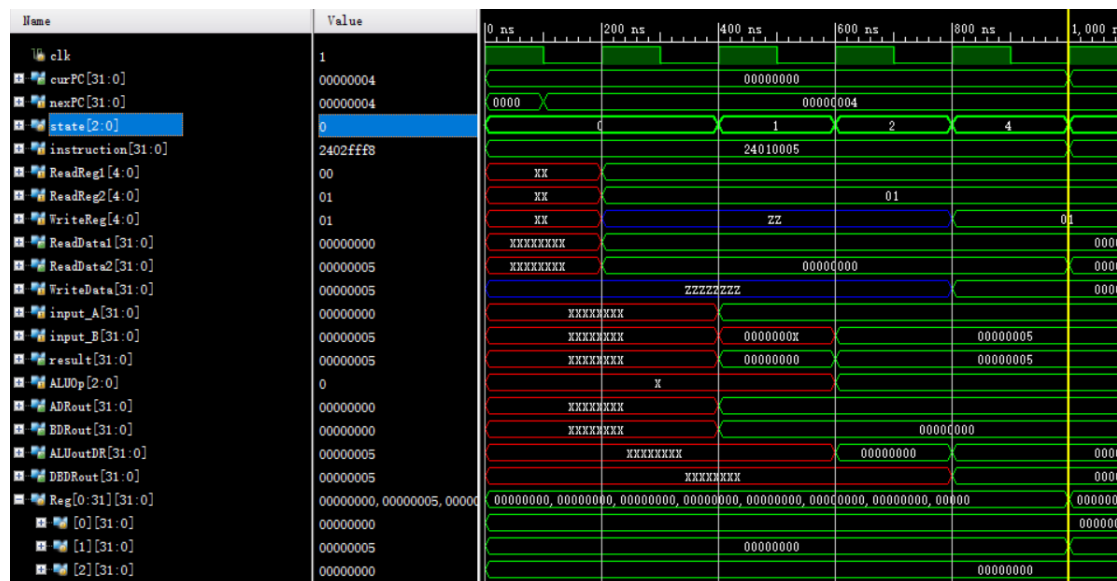
```
always @(posedge clk)
begin
    if (IRWre == 1) begin
        op <= instruction[31:26];
        rs <= instruction[25:21];
        rt <= instruction[20:16];
        rd <= instruction[15:11];
        sa <= instruction[10:6];
        funct <= instruction[5:0];
        immediate <= instruction[15:0];
        address <= instruction[25:0];
    end
end
```

2、验证CPU正确性

对CPU的正确性的检测中，使用给出的实验检查时的测试程序段进行测试，由于需要检测的是每一指令的正确性，便能得出CPU的正确性，故对于一些重复出现的同类型指令，说明部分侧重在第一次出现的地方。首先将给出的程序段转换成32位二进制指令，每8位用空格隔开，并存放指令寄存器部分要读取的文件中，部分指令如下图：



指令1: addiu \$1,\$0,5



(1) 在仿真波形图中, curPC表示当前指令的地址为0x00000000, instruction为当前指令的十六进制代码值, 经对比, 取出的指令正确。

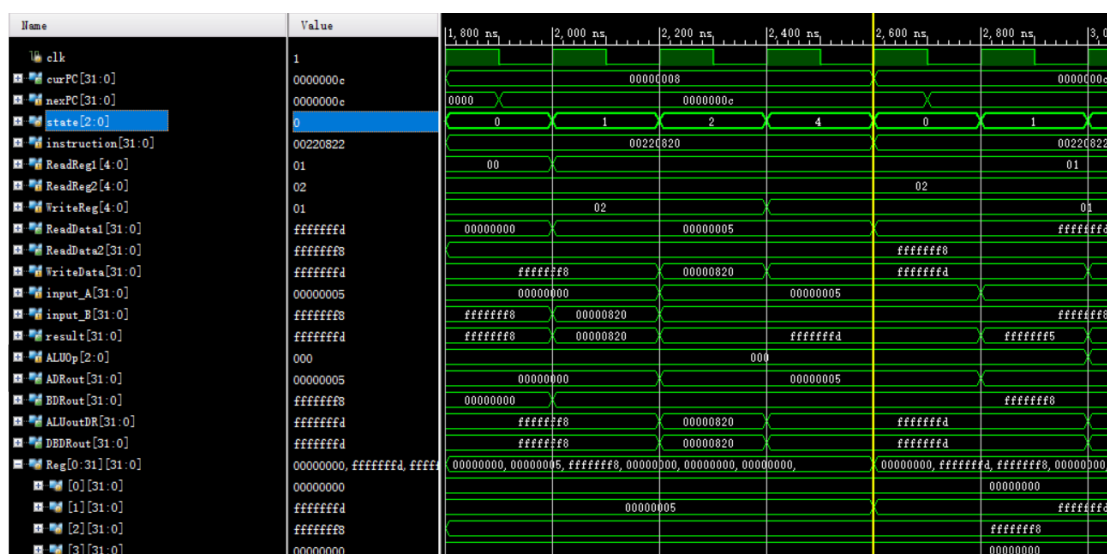
(2) 在第一个时钟周期（前面有些是ReSet时的部分），CPU的状态state为0，为sIF状态。成功取指后可以看出，rs为\$0，数据为0，rt为\$1，数据为0。

(3) 在第二个时钟周期，CPU的状态为1，为sID状态。在该状态的时钟上升沿处，ADROUT、BDROUT都成功读入为0，即rs、rt数据。

(4) 在第三个时钟周期, CPU的状态为2, 为sEXE状态。在该状态中, ALU的input_A、input_B分别为0、5, ALUop为000, 即两数相加, 运算结果为result=0+5=5, 并且该状态结束时, 下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为5。

(5) 在第四个时钟周期, CPU的状态为4, 为sWB状态。在该状态中, 要写入的寄存器WriteReg为01, 即\$1, 写入的数据WriteData为5, 在该状态结束时, 下一个状态时钟上升沿处将数据写回寄存器, 此时可看见\$1的值写为5。

指令2 add \$1, \$1, \$2



(1) 在仿真波形图中, curPC表示当前指令的地址为0x00000008, instruction为当前指令的十六进制代码值, 经对比, 取出的指令正确。

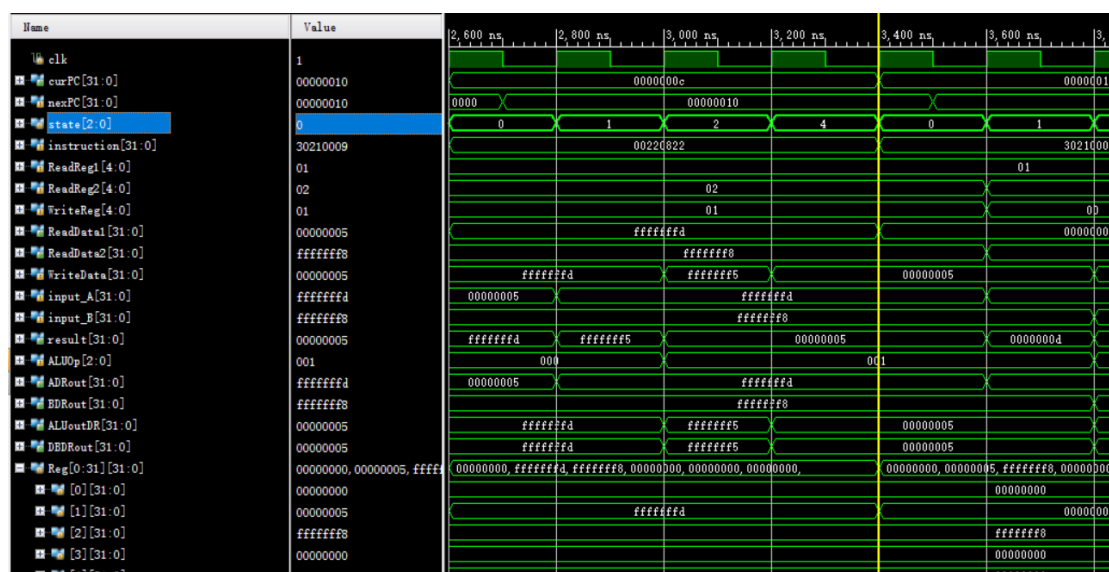
(2) sIF状态: 成功取指后可以看出, rs为\$1, 数据为5, rt为\$2, 数据为-8。

(3) sID状态: 在该状态的时钟上升沿处, ADRout、BDRout都成功读入为5、-8, 即rs、rt数据。

(4) sEXE状态: 在该状态中, ALU的input_A、input_B分别为5、-8, ALUop为000, 即两数相加, 运算结果为result=5+ (-8) =-3, 并且该状态结束时, 下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为-3。

(5) sWB状态: 在该状态中, 要写入的寄存器WriteReg为01, 即\$1, 写入的数据WriteData为-3, 在该状态结束时, 下一个状态时钟上升沿处将数据写回寄存器, 此时可看见\$1的值写为-3。

指令3 sub \$1, \$1, \$2



(1) 在仿真波形图中，curPC表示当前指令的地址为0x0000000c，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。

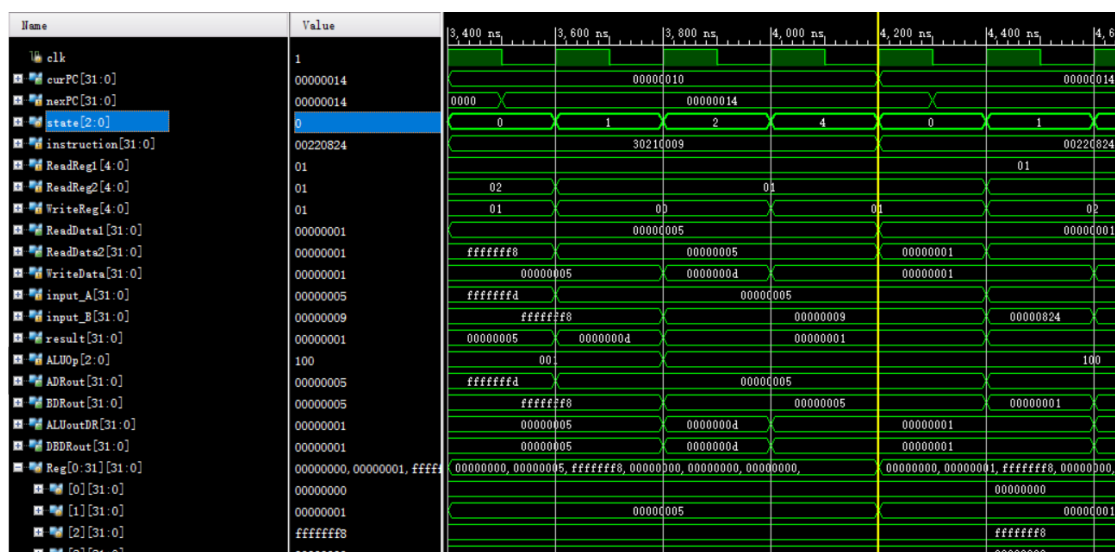
(2) sIF状态：成功取指后可以看出，rs为\$1，数据为-3，rt为\$2，数据为-8。

(3) sID状态：在该状态的时钟上升沿处，ADRout、BDRout都成功读入为-3、-8，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为-3、-8，ALUOp为001，即两数相减，运算结果为result=-3-(-8)=5，并且该状态结束时，下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为5。

(5) sWB状态：在该状态中，要写入的寄存器WriteReg为01，即\$1，写入的数据WriteData为5，在该状态结束时，下一个状态时钟上升沿处将数据写回寄存器，此时可看见\$1的值写为5。

指令4 andi \$1, \$1, 9



(1) 在仿真波形图中, curPC表示当前指令的地址为0x00000010, instruction为当前指令的十六进制代码值, 经对比, 取出的指令正确。

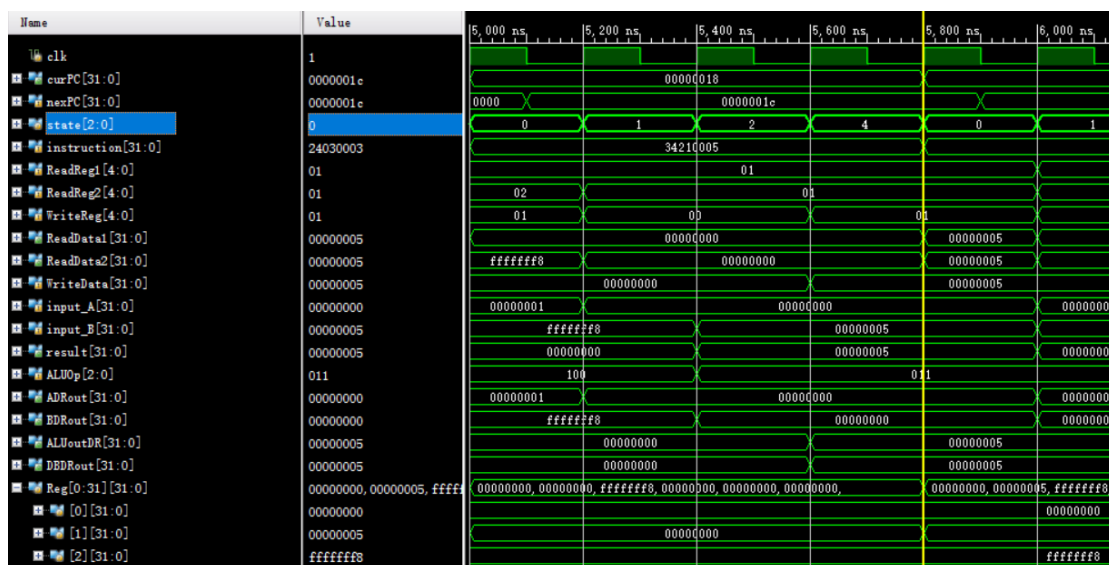
(2) sIF状态: 成功取指后可以看出, rs为\$1, 数据为5, rt为\$1, 数据为5。

(3) sID状态: 在该状态的时钟上升沿处, ADROUT、BDRROUT都成功读入为5、5, 即rs、rt数据。

(4) sEXE状态: 在该状态中, ALU的input_A、input_B分别为5、9, ALUop为100, 即两数按位相与, 运算结果为result=5&9=1, 并且该状态结束时, 下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为1。

(5) sWB状态: 在该状态中, 要写入的寄存器WriteReg为01, 即\$1, 写入的数据WriteData为1, 在该状态结束时, 下一个状态时钟上升沿处将数据写回寄存器, 此时可看见\$1的值写为1。

指令5 and \$1, \$1, \$2



(1) 在仿真波形图中，curPC表示当前指令的地址为0x00000018，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。

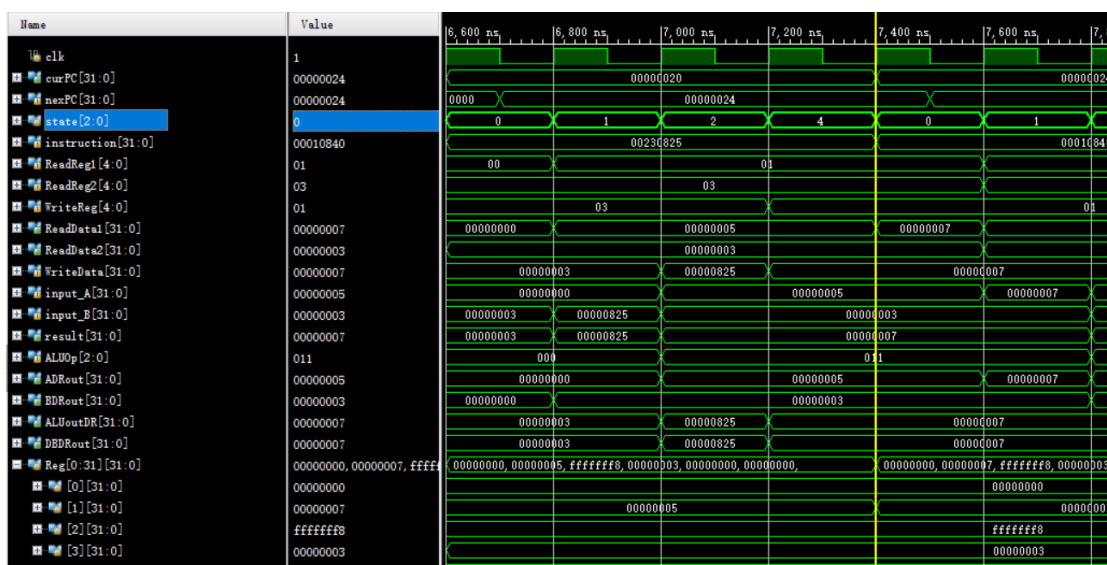
(2) sIF状态：成功取指后可以看出，rs为\$1，数据为0，rt为\$1，数据为0。

(3) sID状态：在该状态的时钟上升沿处，ADRout、BDRout都成功读入为0、0，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为0、5，ALUOp为011，即两数按位相或，运算结果为result=0|5=5，并且该状态结束时，下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为5。

(5) sWB状态：在该状态中，要写入的寄存器WriteReg为01，即\$1，写入的数据WriteData为5，在该状态结束时，下一个状态时钟上升沿处将数据写回寄存器，此时可看见\$1的值写为5。

指令7 or \$1, \$1, \$3



(1) 在仿真波形图中，curPC表示当前指令的地址为0x00000020，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。

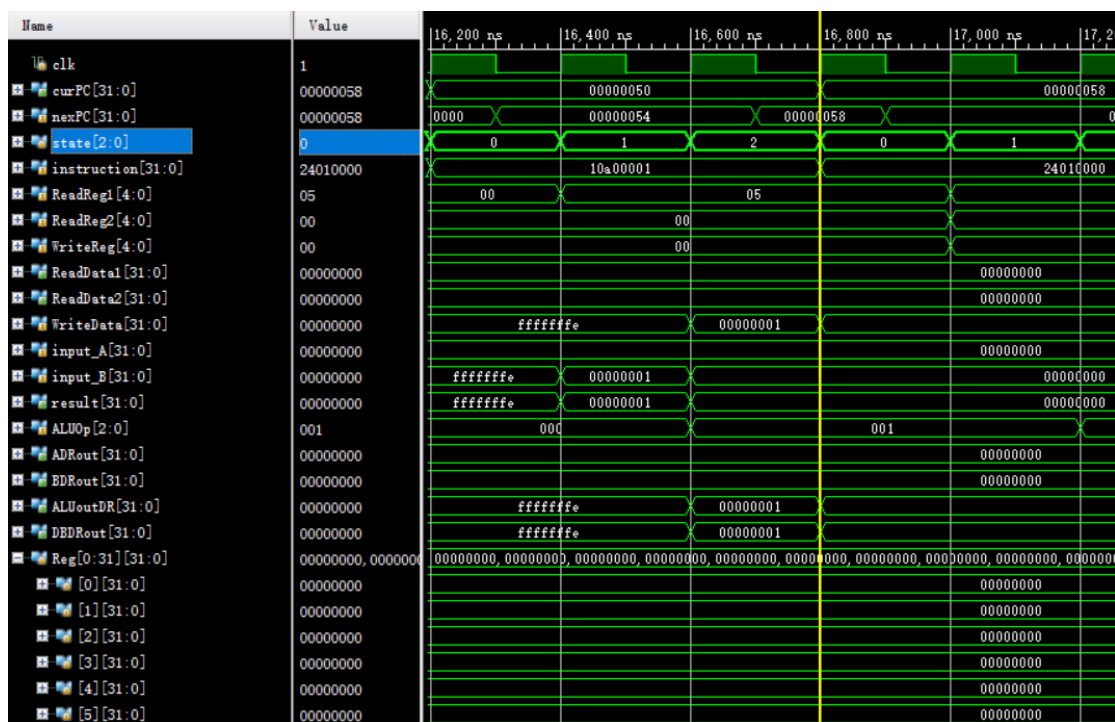
(2) sIF状态：成功取指后可以看出，rs为\$1，数据为5，rt为\$3，数据为3。

(3) sID状态：在该状态的时钟上升沿处，ADRout、BDRout都成功读入为5、3，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为5、3，ALUOp为011，即两数按位相或，运算结果为 $result = 5 | 3 = 7$ ，并且该状态结束时，下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为7。

(5) sWB状态：在该状态中，要写入的寄存器WriteReg为01，即\$1，写入的数据WriteData为7，在该状态结束时，下一个状态时钟上升沿处将数据写回寄存器，此时可看见\$1的值写为7。

指令8 sll \$1, \$1, 1



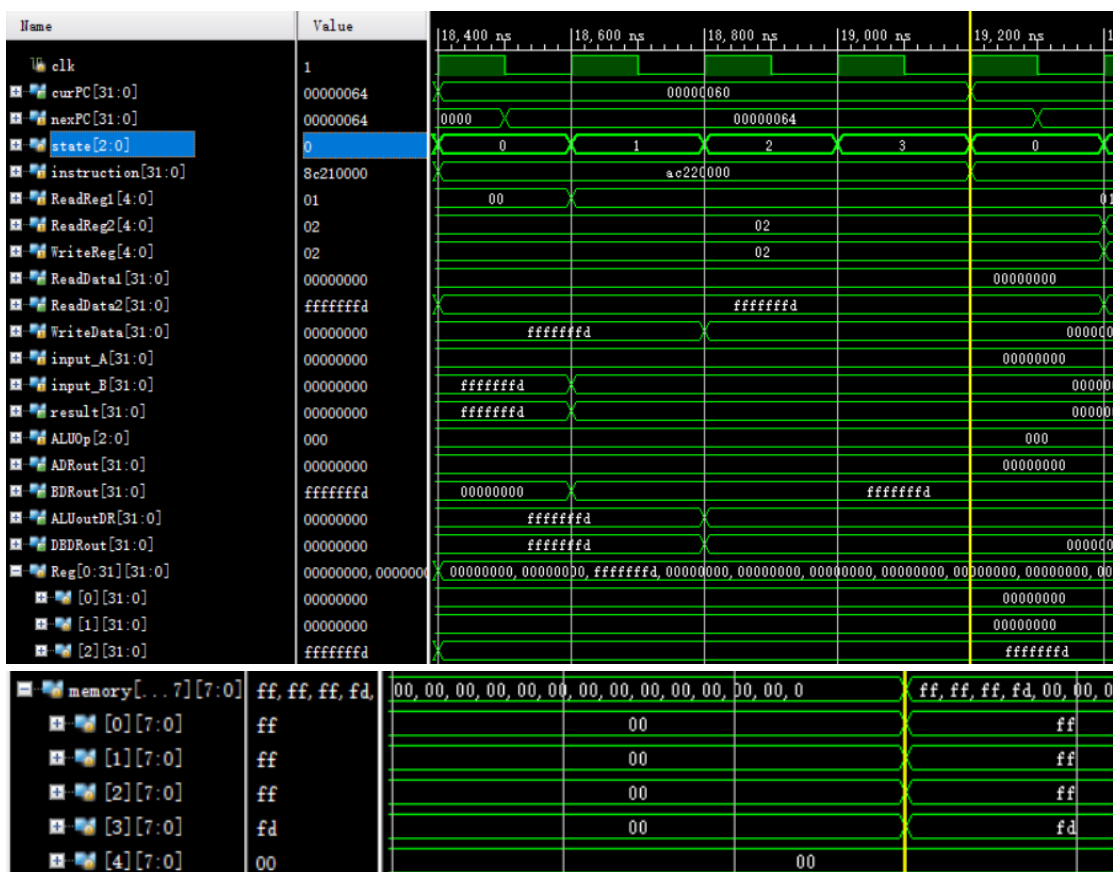
(1) 在仿真波形图中，curPC表示当前指令的地址为0x00000050，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。该指令经过以下3个状态。

(2) sIF状态：成功取指后可以看出，rs为\$5，数据为0，rt为\$0，数据为0。

(3) sID状态：在该状态的时钟上升沿处，ALURout、BDRout都成功读入为0、0，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为0、0，ALUOp为001，即两数相减，运算结果为result=0-0=0，两数相等，则zero为1即需要跳转到地址0x58，且在该状态可看见nexPC更改为0x00000058，且在下一条指令的取指状态时可以知道该指令已被正确执行。

指令11 sw \$2, 0(\$1)



(1) 在仿真波形图中, curPC表示当前指令的地址为0x00000060, instruction为当前指令的十六进制代码值, 经对比, 取出的指令正确。该指令经过以下4个状态。

(2) sIF状态: 成功取指后可以看出, rs为\$1, 数据为0, rt为\$2, 数据为-3。

(3) sID状态: 在该状态的时钟上升沿处, ADRout、BDRout都成功读入为0、-3, 即rs、rt数据。

(4) sEXE状态: 在该状态中, ALU的input_A、input_B分别为0、0, ALUop为000, 即两数相加, 运算结果为result=0+0=0, 并且该状态结束时, 下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为0。

(5) sMEM状态: 在该状态中, 需要将\$2的数据-3写入数据存储器中, 在该状态结束时, 下一个状态时钟上升沿处将数据写回寄存器, 此时可看见数据存储器的第0、1、2、3个存储单元分别被写为ff、ff、ff、fd。

指令12 lw \$1, 0(\$1)



(1) 在仿真波形图中，curPC表示当前指令的地址为0x00000064，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。该指令经过以下5个状态。

(2) sIF状态：成功取指后可以看出，rs为\$1，数据为0，rt为\$1，数据为0。

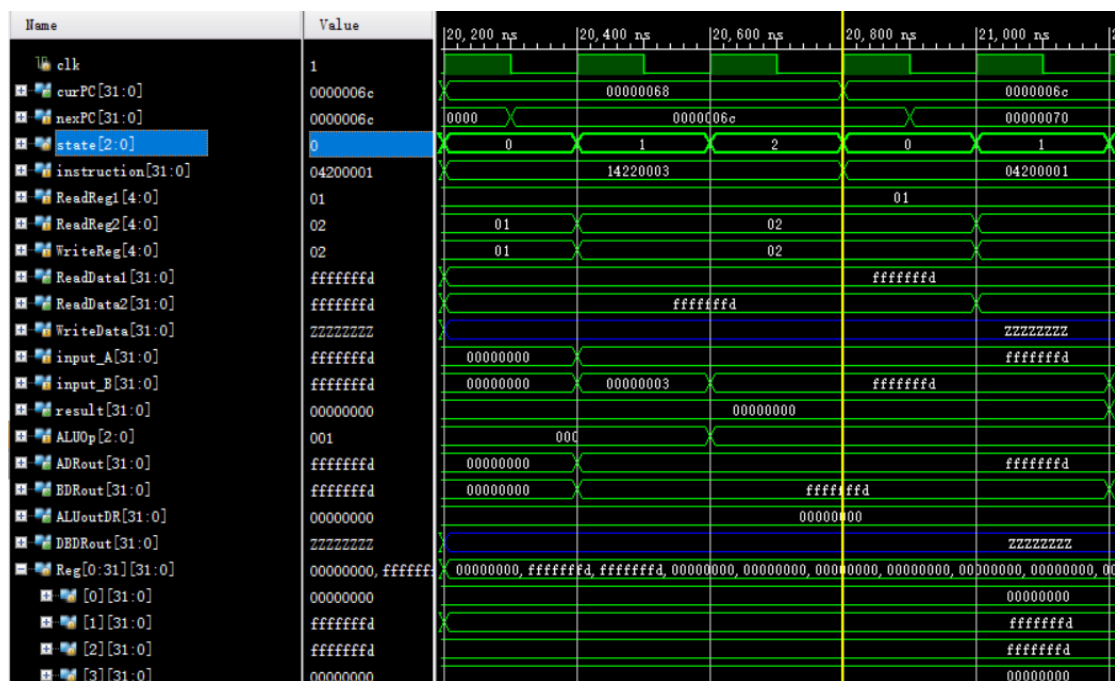
(3) sID状态：在该状态的时钟上升沿处，ADROUT、BDRout都成功读入为0、0，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为0、0，ALUOp为000，即两数相加，运算结果为result=0+0=0，并且该状态结束时，下一个状态时钟上升沿处ALUoutDR成功读入为0。

(5) sMEM状态：在该状态中，需要将数据存储器的第0、1、2、3个存储单元的数据分别取出组合，并DBDRout成功读入为-3。

(6) sWB状态：在该状态中，要写入的寄存器WriteReg为01，即\$1，写入的数据WriteData为-3，在该状态结束时，下一个状态时钟上升沿处将数据写回寄存器，此时可看见\$3的值写为-3。

指令13 bne \$1, \$2, 3



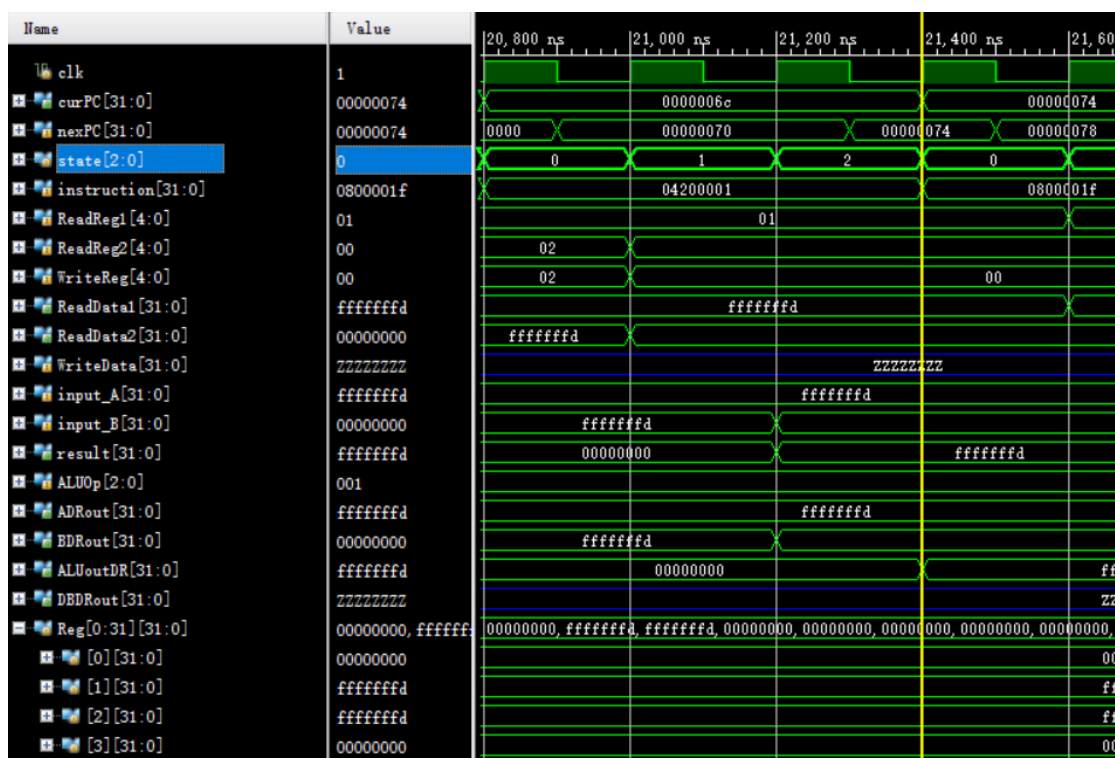
(1) 在仿真波形图中，curPC表示当前指令的地址为0x00000068，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。该指令经过以下3个状态。

(2) sIF状态：成功取指后可以看出，rs为\$1，数据为-3，rt为\$2，数据为-3。

(3) sID状态：在该状态的时钟上升沿处，ADRout、BDRout都成功读入为-3、-3，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为-3、-3，ALUOp为001，即两数相减，运算结果为 $\text{result} = -3 - (-3) = 0$ ，两数相等，则zero为1即不需要跳转到地址0x68，且在该状态可看见nexPC更改为0x0000006c，且在下一条指令的取指状态时可以知道该指令已被正确执行。

指令14 bltz \$1, 1



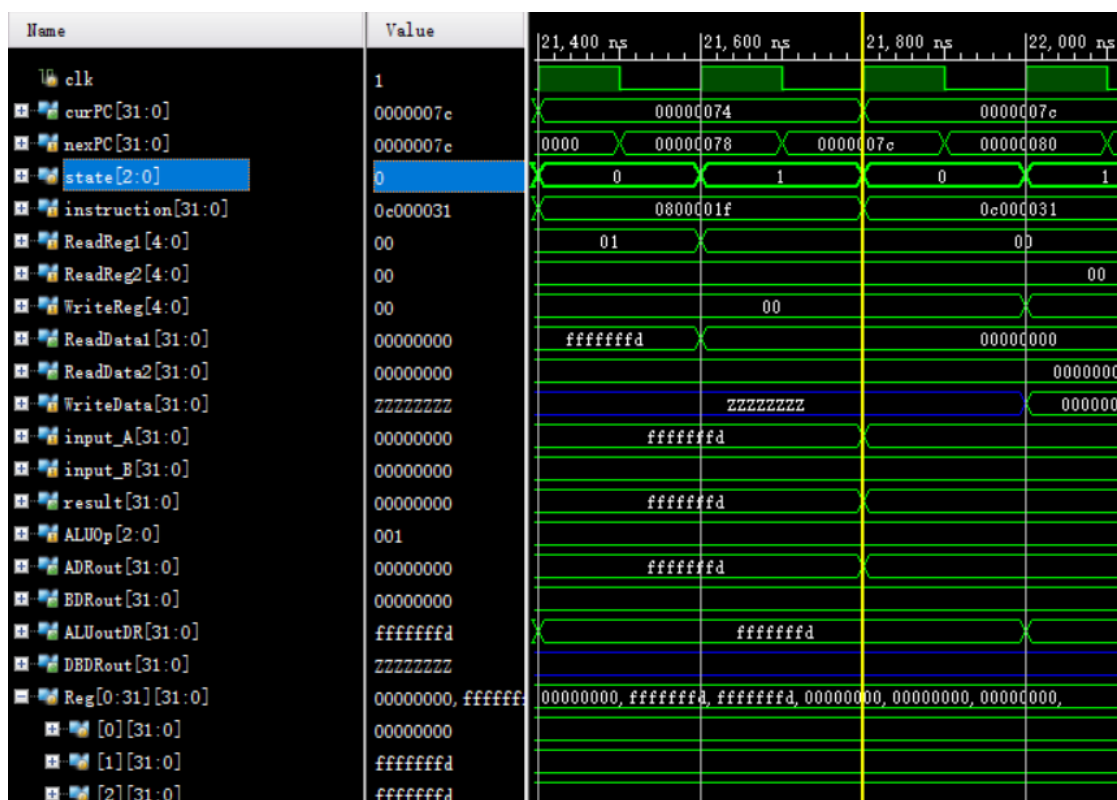
(1) 在仿真波形图中，curPC表示当前指令的地址为0x0000006c，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。该指令经过以下3个状态。

(2) sIF状态：成功取指后可以看出，rs为\$1，数据为-3，rt为\$0，数据为0。

(3) sID状态：在该状态的时钟上升沿处，ADRout、BDRout都成功读入为-3、0，即rs、rt数据。

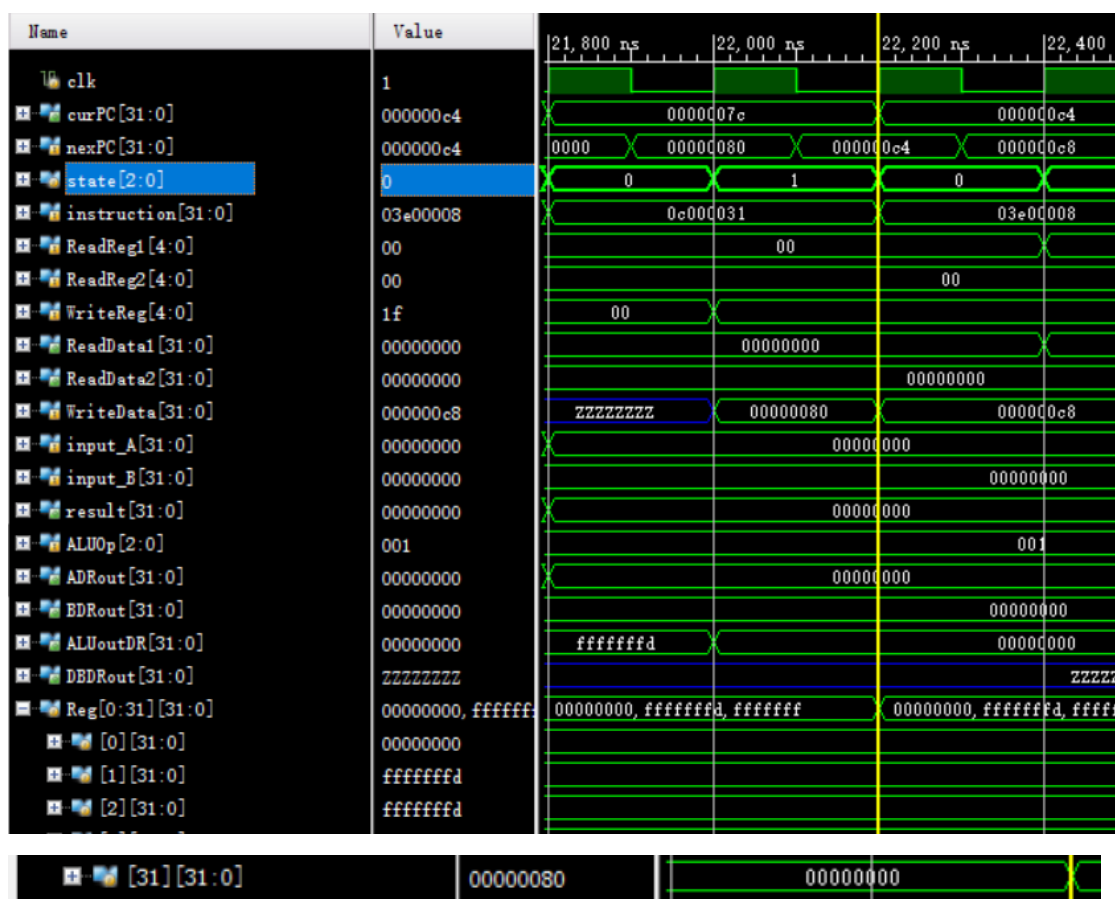
(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为-3、0，ALUop为001，即两数相减，运算结果为 $result = -3 - 0 = -3 < 0$ ，则sign为1即需要跳转到地址0x74，且在该状态可看见nexPC更改为0x00000074，且在下一条指令的取指状态时可以知道该指令已被正确执行。

指令15 j 0x1f



- (1) 在仿真波形图中，curPC表示当前指令的地址为0x00000074，。该指令经过以下2个状态。
- (2) sIF状态：成功取指后可看见instruction的值为正确指令值。
- (3) sID状态：nexPC变更为0x0000007c， 且在下一条指令的取指状态时可以知道该指令已被正确执行。

指令16 jal 0x31

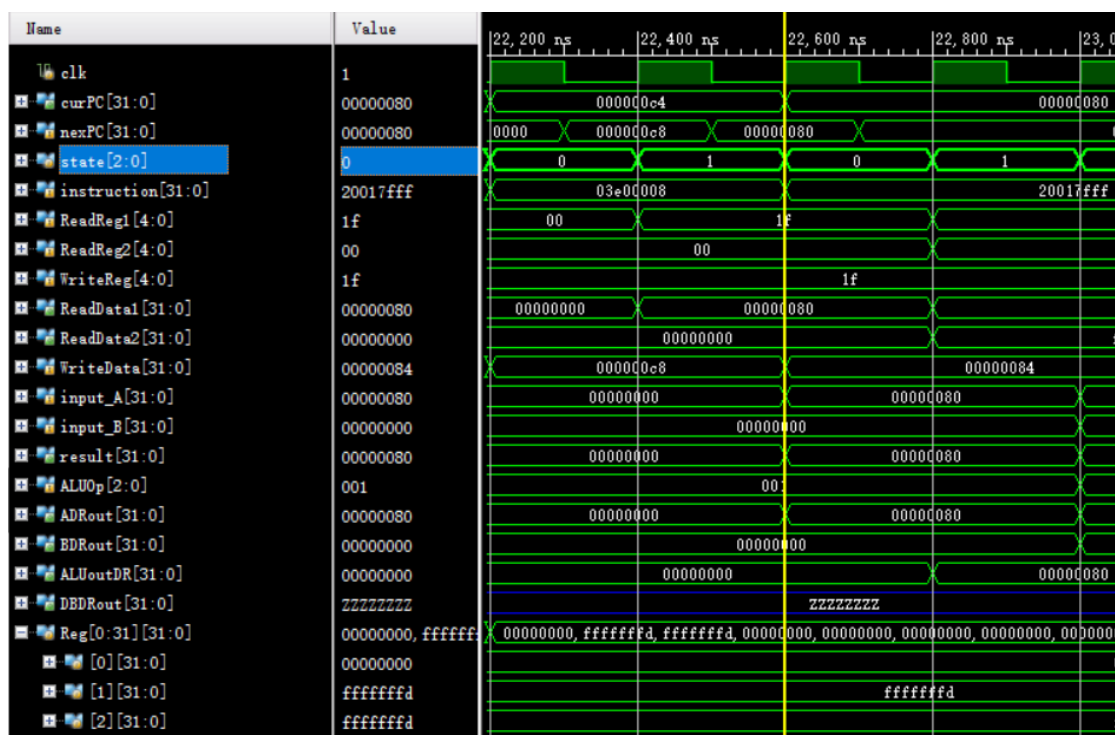


(1) 在仿真波形图中，curPC表示当前指令的地址为0x0000007c。该指令经过以下2个状态。

(2) sIF状态：成功取指后可看见instruction的值为当前指令的十六进制代码值，经对比，取出的指令正确。

(3) sID状态：在该状态中，WriteData的数据为curPC+4，WriteReg为\$31，在该状态结束时，下一个状态时钟上升沿处将数据写回寄存器，此时可看见\$31的值写为0x80。而且nexPC变更为0x000000c4，且在下一条指令的取指状态时可以知道该指令已被正确执行。

指令17 jr \$31

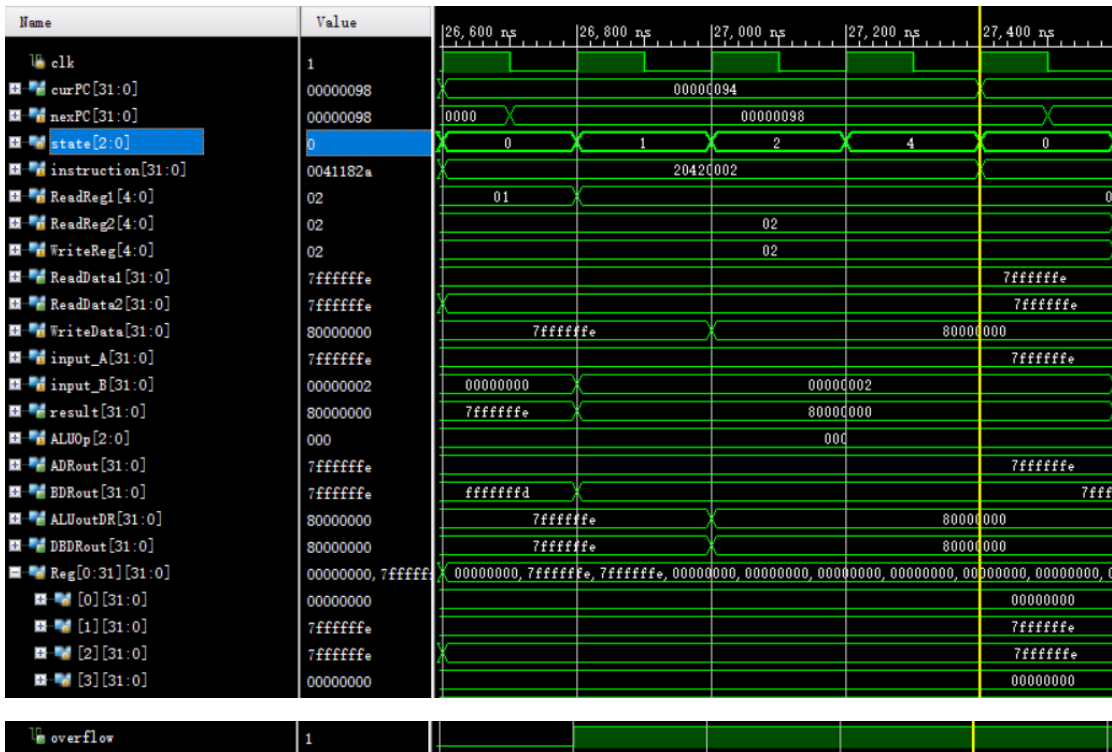


(1) 在仿真波形图中，curPC表示当前指令的地址为0x000000c4。该指令经过以下2个状态。

(2) sIF状态：成功取指后可看见instruction的值为当前指令的十六进制代码值，经对比，取出的指令正确。此时rs为\$31，数据为0x80。

(3) sID状态：在该状态中，nexPC变更为0x00000080即\$31的内容，且在下一条指令的取指状态时可以知道该指令已被正确执行。

指令18 addi \$2, \$2, 2



(1) 在仿真波形图中, curPC表示当前指令的地址为0x00000094, instruction为当前指令的十六进制代码值, 经对比, 取出的指令正确。

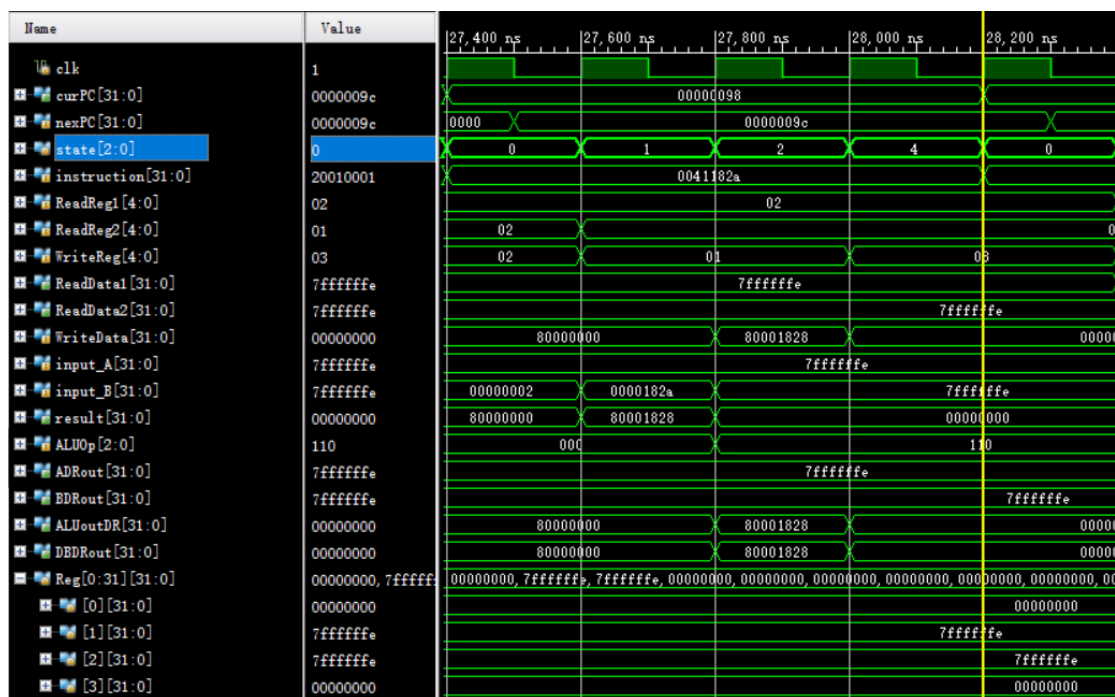
(2) sIF状态：成功取指后可以看出，rs为\$2，数据为7ffffffeH，rt为\$3，数据为7ffffffeH。

(3) sID状态: 在该状态的时钟上升沿处, ADROUT、BDROUT都成功读入为7ffffffeH、7ffffffeH, 即rs、rt数据。

(4) sEXE状态: 在该状态中, ALU的input_A、input_B分别为7ffffffeH、2, ALUop为000, 即两数相加, 运算结果为result=7ffffffeH+2=80000000H, 并且该状态结束时, 下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为80000000H。

(5) sWB状态: 在该状态中, 要写入的寄存器WriteReg为02, 即\$2, 写入的数据WriteData为80000000H, 但由于addi指令在结果溢出时不写回寄存器, 故由于overflow=1, 不写回\$2, 在该状态结束时, 下一个状态时钟上升沿处将数据写回寄存器, 此时可看见\$2的值依然为7ffffffeH。

指令19 slt \$3, \$2, \$1



(1) 在仿真波形图中，curPC表示当前指令的地址为0x00000098，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。

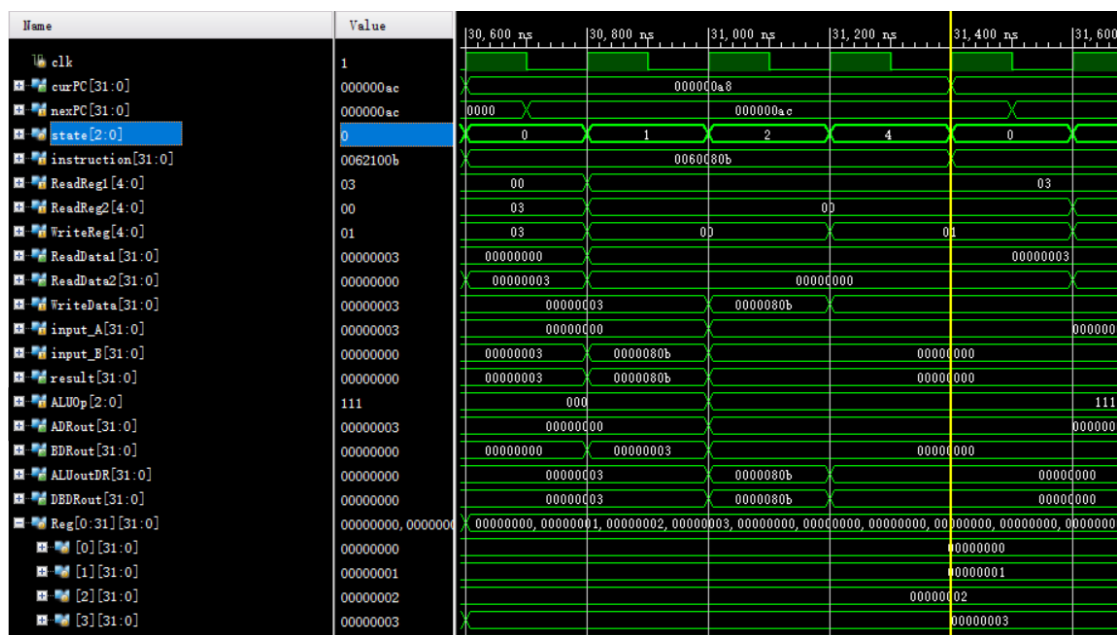
(2) sIF状态：成功取指后可以看出，rs为\$2，数据为7ffffffeH，rt为\$1，数据为7ffffffeH。

(3) sID状态：在该状态的时钟上升沿处，ADRoute、BDRout都成功读入为7ffffffeH、7ffffffeH，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为7ffffffeH、7ffffffeH，ALUOp为110，即两带符号数进行比较，运算结果为result=(7ffffffeH=7ffffffeH)=0，并且该状态结束时，下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为0。

(5) sWB状态：在该状态中，要写入的寄存器WriteReg为03，即\$3，写入的数据WriteData为0，在该状态结束时，下一个状态时钟上升沿处将数据写回寄存器，此时可看见\$3的值写为0。

指令20 movn \$1, \$3, \$0



(1) 在仿真波形图中，curPC表示当前指令的地址为0x000000a8，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。

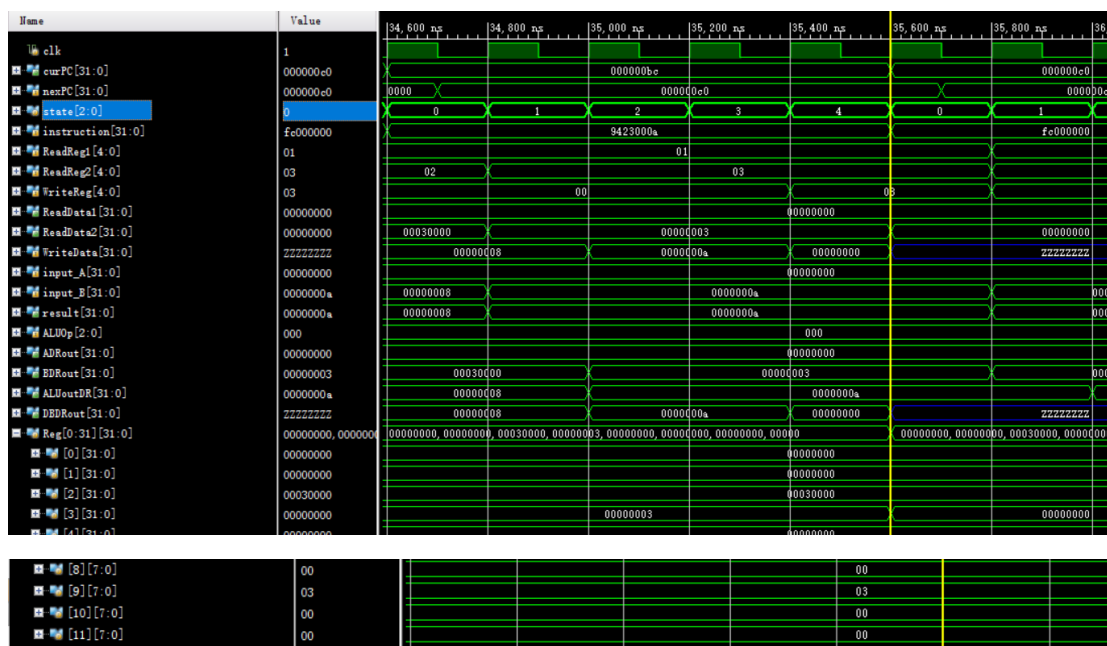
(2) sIF状态：成功取指后可以看出，rs为\$3，数据为3，rt为\$0，数据为0。

(3) sID状态：在该状态的时钟上升沿处，ADROUT、BDRout都成功读入为3、0，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为3、0，ALUOp为111，即movn用的op，运算结果为result=input_B=0，zero=0，并且该状态结束时，下一个状态时钟上升沿处ALUoutDR和DBDR都成功读入为0。

(5) sWB状态：在该状态中，WriteReg为01，即\$1，WriteData为3，而由于zero为0，即无需改写rd，故可看见\$1的值仍为1，没有被改写。

指令21 lhu \$3, 10(\$1)



(1) 在仿真波形图中，curPC表示当前指令的地址为0x000000bc，instruction为当前指令的十六进制代码值，经对比，取出的指令正确。该指令经过以下5个状态。

(2) sIF状态：成功取指后可以看出，rs为\$1，数据为0，rt为\$3，数据为3。

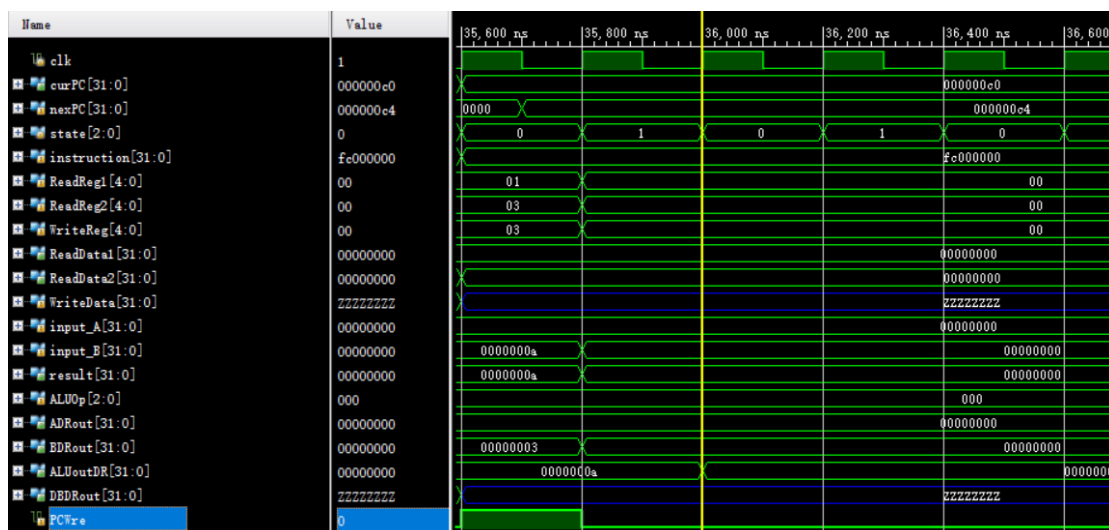
(3) sID状态：在该状态的时钟上升沿处，ADROUT、BDRout都成功读入为0、3，即rs、rt数据。

(4) sEXE状态：在该状态中，ALU的input_A、input_B分别为0、10，ALUop为000，即两数相加，运算结果为result=0+10=10，并且该状态结束时，下一个状态时钟上升沿处ALUoutDR成功读入为10。

(5) sMEM状态：在该状态中，需要将数据存储器的第10、11个存储单元的数据分别取出组合，并DBDRout成功读入为0。

(6) sWB状态：在该状态中，要写入的寄存器WriteReg为03，即\$3，写入的数据WriteData为0，在该状态结束时，下一个状态时钟上升沿处将数据写回寄存器，此时可看见\$3的值写为0。

指令22 halt



(1) 在仿真波形图中，curPC表示当前指令的地址为0x000000c0，。该指令经过以下2个状态。

(2) sIF状态：成功取指后可看见instruction的值为正确指令值。

(3) sID状态：PCWre为0， 且在后续的时钟周期内指令均为halt，即实现了停机。

3、在Basys3板上的实现

(1) 在板上运行CPU时，需要通过4个数码管显示器来观察CPU的相应指令的执行情况，由于在该CPU上指令地址、数据地址的范围都在0-255内，寄存器地址范围为0-31，也就是只使用了低8位，每2位数码管便可以显示出上述数据，根据烧板文档的说明，配合两个开关SW_in（SW15，SW14），有以下显示内容：

SW_in=00, 当前指令地址PC： 下条指令地址PC

SW_in=01, RS寄存器地址： RS寄存器数据

SW_in=10, RT寄存器地址： RT寄存器数据

SW_in=11, ALU结果输出： DB总线输出

而同时Reset信号也接在开关上（SW0），CPU的执行时钟依靠按键（单脉冲）即BTNR单步执行，以便于查看不同信息。

(2) 在板上显示数码管的内容时，需要对时钟信号进行分频，分频的目的是计数，采用扫描式显示，利用数码管的余晖效应和人眼的视觉暂留效应以达到多个数码管“同时”被点亮的效果。而结合上学期数电的知识，此处的分频具体代码如下：

```

parameter num = 49999;
integer i = 0;

always @(posedge clk)
begin
    if (i >= num)
    begin
        i <= 0;
        clk_div <= ~clk_div;
    end
    else
        i <= i + 1;
    end
end

```

(3) 由于相应数码管被点亮时, 位控信号AN0-AN3分别为0111、1011、1101、1110, 所以使用上述分频后的时钟计数4次循环以点亮数码管。具体代码如下:

```

initial
    count <= 2'b00;
always @(posedge clk)
begin
    if (count == 2'b11)
        count <= 2'b00;
    else
        count <= count + 1;
    end
end

```

(4) 由上述计数器生成的数对应生成数码管位控信号和根据SW_in的闭合情况生成CPU执行时需要显示的信息段, 然后再将信息段按照每个数码管(8位)划分进对应的数码管:

```

always @(*)
begin
    case (SW_in)
        2'b00: total_data = {curPC[7:0], nexPC[7:0]};
        2'b01: total_data = {2'b00, rs[4:0], rs_data[7:0]};
        2'b10: total_data = {2'b00, rt[4:0], rt_data[7:0]};
        2'b11: total_data = {ALU_result[7:0], DB_data[7:0]};
        default: total_data = 16'h0000;
    endcase
end

```

```

always @(*)
begin
    case (count)
        2'b00: begin
            AN_count = 4'b1110;
            display_data = total_data[3:0];
        end
        2'b01: begin
            AN_count = 4'b1101;
            display_data = total_data[7:4];
        end
        2'b10: begin
            AN_count = 4'b1011;
            display_data = total_data[11:8];
        end
        2'b11: begin
            AN_count = 4'b0111;
            display_data = total_data[15:12];
        end
        default: begin
            AN_count = 4'bxxxx;
            display_data = 4'bxxxx;
        end
    endcase
end

```

(5) 将上述生成的要显示的数据送到数码管显示信号的转换器中，即烧板文档中共阳极的代码段，其低7位即可送往数码管分别连接显示。

(6) 消抖处理

由于CPU时钟是由按键（单脉冲）产生的，而人在按下去时会产生中间的波动脉冲，使得被误认为高/低电平，而对于这个消抖处理我并不是很明白，所以我是在百度百科上找到的“在用基于Verilog语言的时序逻辑电路设计按键消抖电路时，通常认为机械抖动的最大周期是20ms，对每一个时钟脉冲信号对按键状态进行取样，以便进行按键消抖处理。在程序中设置一个计数器，来采集按键的值，若按键的值在20ms内都是低电平或者高电平，则可确定这次是人为按键。”这一段话，并结合其下代码，写下了以下消抖处理代码，而在Reset=0时将CPU时钟置为1是为了确保在第一条指令的执行中必须有时钟的下降沿，从而顺利使CPU正常执行。（注：消抖处理是在完成仿真、烧板后加上的，而加上前并没有出现问题，但为防止在某次在板子上操作时翻车而加上了，加上后也没有出现问题。然而在加

上后, 在对于仿真时钟周期的初始化设计时出现了问题, 即不知道怎样去仿真按键的CPU时间周期。所以在烧板时将CPU封装成IP核再加上了该模块, 但在仿真时则使用去除抖动处理的工程进行仿真)

```

reg check1, check2, check3;
reg clk_div = 0; //20ms时钟检测抖动

integer i = 0;
always @(posedge clk)
begin
    if (i >= 999999)
    begin
        clk_div = ~clk_div;
        i <= 0;
    end
    else
        i = i + 1;
    end
always @(posedge clk_div or negedge Reset)
begin
    if (Reset == 0)
    begin
        check1 <= 1;
        check2 <= 1;
        check3 <= 1;
    end
    else
    begin
        check1 <= i_clk;
        check2 <= check1;
        check3 <= check2;
    end
end
assign o_clk = check1 & check2 & check3;

```

(7) 操作方法: 先将Reset (SW0) 置零按下一次BTNR时钟初始化PC值为0, 再将Reset置为1后, 每按下一次BTNR, 即执行一条指令, 置SW_in为不同的值可查看不同信息。前6条指令在板上运行的贴图如下, 顺序为:

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总线输出

指令1 0x00000000 addiu \$1,\$0,5

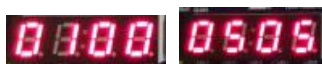
sIF:



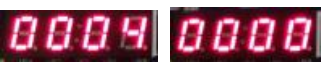
sID:



sEXE:



sWB:

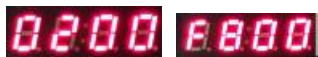
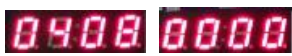


指令2 0x00000004 addiu \$2,\$0,-8

sIF:



sID:





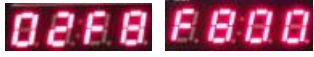


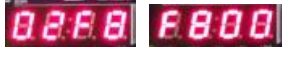


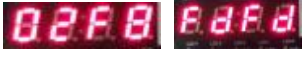



sEXE:





sWB:












指令3 0x00000008 add \$1,\$1,\$2

sIF:  

sID:  

sEXE:  

sWB:  


指令4 0x0000000C sub \$1,\$1,\$2

sIF:  

sID:  

sEXE:  

sWB:  


指令5 0x00000010 andi \$1,\$1,9

sIF:  

sID:  



六. 实验心得

首先, 在单周期CPU实验的基础上, 有了一定的经验和debug能力, 所以在多周期CPU的实验中比单周期遇到的困难更少了, 完成的时间也比单周期时的要短, 对两者之间的区别、原理也更为理解。其实多周期CPU数据通路中的大部分模块都可以用单周期CPU的或者在其基础上稍加修改而得到, 我感觉多周期CPU的实验要比单周期更能深刻理解每条指令在不同阶段所用到的部件和控制信号的变化, 控制信号多也就意味着对其的把握和模块的是否使用更为重要, 所以需要在建立控制信号的真值表时就需要格外注意控制信号和输入信号、指令、所处的状态间的关系的正确性, 一个信号的写错或者错误判断都会使得在测试代码中出现问题。所以在多周期的实验中也加深了对于指令如何运行、状态如何转换、信号如何不断的变化等的理解。

其次, 多周期CPU依然采用模块化思想, 将数据通路中的每个小部件都划分为一个个模块, 在实现各个模块的过程中不必过于注重它们之间的联系, 主要还是其内在的逻辑关系, 各模块设计后最好能进行单元测试, 像写程序一样做好测试; 然后再用一个顶层设计文件将各个模块连接起来, 而在实际实验过程中, 我是使用了block design的连接方法, 将各个模块封装为IP核再调用连接, 这样做的好处就是逻辑性较强, 在debug和检查顶层文件设计的过程也变得容易, 可读性也更强。

然后, 我在这个实验中遇到了这样的一个问题: 仿真和烧板的结果不一致, 主要表现在时延上。当指令运行到一定数目后, 跳转指令不能正常地进行如期跳转, 在多次写小段代码和测试代码的测试对比下发现, 很可能是在经过一定的时钟周期后, 时延积累使得PCWre信号和PCSrc信号的变化不一致, 导致跳转到错误指令。而我使用的解决方法是将PCSelect模块加入clk时钟信号, 将PCSrc稳定半个周期使跳转后再更改为下一值, 确保跳转到的指令为期望指令。

最后，我感受到了防抖模块的重要性。在单周期CPU实验中，可能是因为需要按动按钮的次数较少，在没有防抖模块下也没有明显的抖动现象。然而在多周期中，也可能是由于需要多次按动来检查，可以明显看到状态连续跳转，在需要明确查看某条指令的某个状态时很不方便。所以，在百度的帮助下写了防抖模块并成功防抖。

在这次实验中，其实最大的感触还是“理想很丰满，现实很骨感”，仿真成功并不代表着烧板就能成功，需要考虑的因素还有很多，比如时延、机械抖动等等。