



《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 软件工程四 (8) 班

学生姓名 : 周远笛

学 号 : 16340311

时 间 : 2018 年 6 月 10 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: $rd \leftarrow -rt \ll (zero-extend)sa$, 左移 sa 位 , (zero-extend)sa**==>比较指令**

(8) slt rd, rs, rt 带符号数

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

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

(9) sltiu rt, rs,immediate 不带符号

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

功能: if ($rs < (zero-extend)immediate$) $rt = 1$ else $rt = 0$, 具体请看表 2 ALU 运算功能表, 不带符号**==>存储器读写指令**

(10) sw rt, immediate(rs)

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

功能: $memory[rs + (sign-extend)immediate] \leftarrow rt$ 。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: $rt \leftarrow memory[rs + (sign-extend)immediate]$ 。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($rs=rt$) $pc \leftarrow pc + 4 + (sign-extend)immediate \ll 2$ else $pc \leftarrow pc + 4$

(13) bltz rs,immediate

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

功能: if($rs < 0$) $pc \leftarrow pc + 4 + (sign-extend)immediate \ll 2$ 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	00000000000000000000000000000000(26位)
--------	---------------------------------------

不改变 pc 的值, pc 保持不变。

三. 实验原理

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

- (1) 取指令(IF): 根据程序计数器 pc 中的指令地址, 从存储器中取出一条指令, 同时, pc 根据指令字长度自动递增产生下一条指令所需要的指令地址, 但遇到“地址转移”指令时, 则控制器把“转移地址”送入 pc , 当然得到的“地址”需要做些变换才送入 pc 。
- (2) 指令译码(ID): 对取指令操作中得到的指令进行分析并译码, 确定这条指令需要完成的操作, 从而产生相应的操作控制信号, 用于驱动执行状态中的各种操作。
- (3) 指令执行(EXE): 根据指令译码得到的操作控制信号, 具体地执行指令动作, 然后转移到结果写回状态。

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

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

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

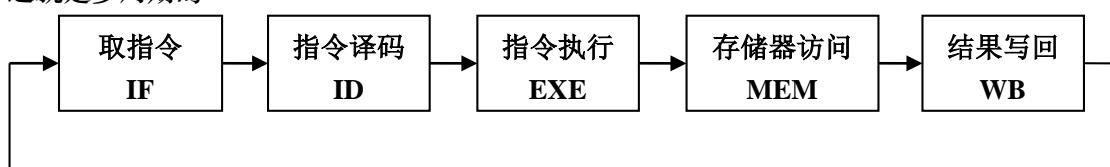


图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

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

I 类型：

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

J 类型：

31	26 25	0
op	address	
6 位	26 位	

其中，

op: 为操作码；

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

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

rd: 为目的操作数寄存器，寄存器地址（同上）；

sa: 为位移量 (shift amt)，移位指令用于指定移多少位；

funct: 为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

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

address: 为地址。

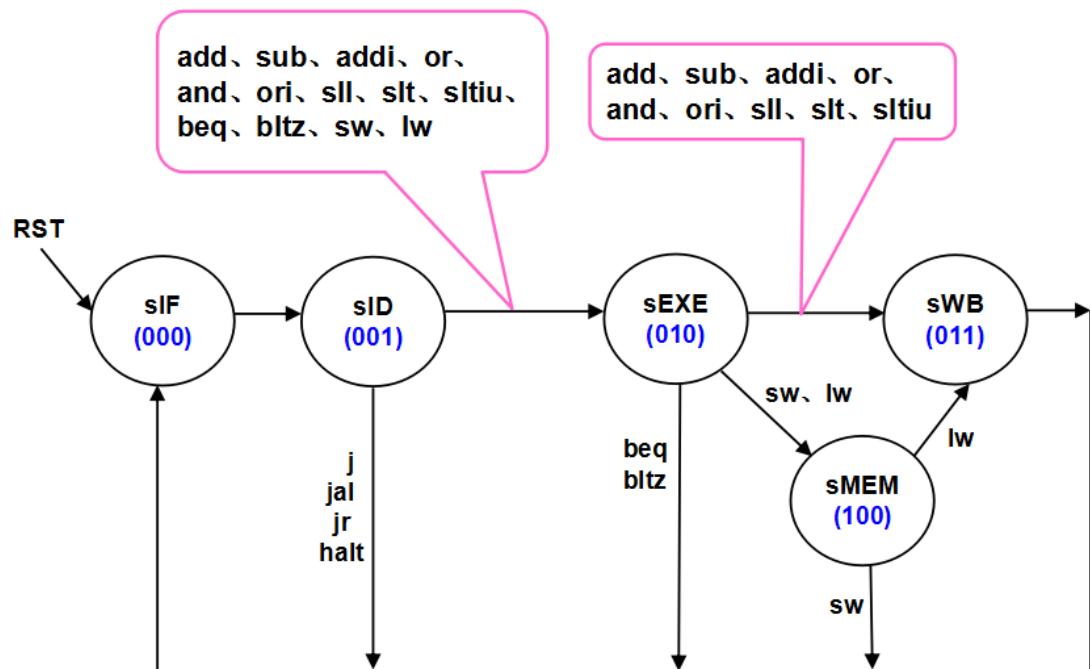


图 2 多周期 CPU 状态转移图

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

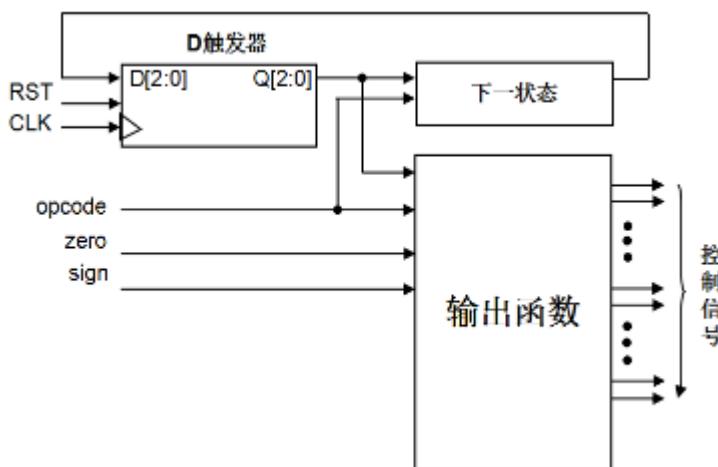


图 3 多周期 CPU 控制部件的原理结构图

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

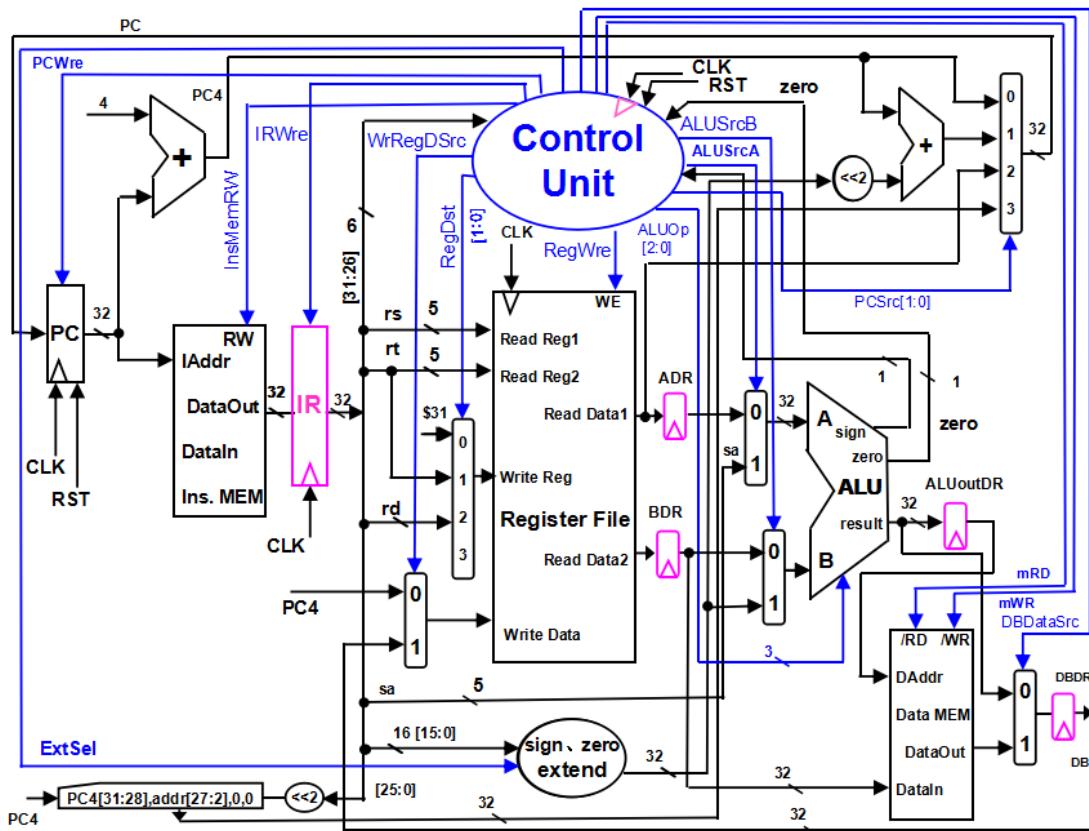


图 4 多周期 CPU 数据通路和控制线路图

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

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。

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

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**(一) CPU模块设计**

整体设计思路与单周期类似, 其中多周期 CPU 在模块方便比单周期多了一个 IR、四个 DR 以及从控制单元发出的相应信号; 此外, 由于多周期 CPU 要将一个指令周期划分为多个阶段(这篇实验中采用的是分为五个阶段, 当然每个指令并不一定需要经历所有的阶段), 由于这种复杂性, 一些数据选择器的选项需要扩充, 比如 PCSrc 和 WriteReg 的数据选择器。

除了模块的变化之外, 多周期最重要的就是确定根据当前状态如何选择时钟信号触发相应的写操作, 既不会产生先读后写的冲突问题, 也不会浪费过多时间等待。本实验采用了如下策略:

模块	上升沿	下降沿
Program Counter	输出下一条指令地址	
Instruction Register	输出寄存器中指令	接收存储 PC 输出的指令
Control Unit	状态转移	
Register File		写寄存器
ADR/BDR/ALUoutDR/DBDR	输出寄存器中的数据	接受存储数据

I. Program Counter

PC模块的设计我分为了两个部分, 一个是main PC (PC.v), 是在时钟上升沿或者重置信号下降沿时对重置信号以及CU输出的PCWre信号做出判断以及响应的。如果为清零的状态, 则PC置为0; 否则如果PCWre = 1, 能够改变PC的值时, 将nextPC赋值给PC。

部分代码如下:

```
always@(posedge clk or negedge reset)
begin
```

```

if(reset == 0)
    IAddr <= 0;
else if (PCWre == 1)
    IAddr <= nextPC;
end

```

另一个部分是nextPC，包括了对PCSrc数据选择的判断和分支语句的加上立即数操作、跳转语句的计算地址操作以及子程序调用、顺序执行的赋值计算操作。当reset或PCSrc或当前PC地址或者当前立即数的值发生变化时，随之变化。

部分代码如下：

```

case (PCSrc)
2'b00:
    IAddr = currentPC + 4;
2'b01: // pc<-pc+4+(sign-extend)immediate
    IAddr = currentPC + 4 + (immediate<<2);
2'b10:
    IAddr = readData1;
2'b11:
begin // pc<-(pc+4)[31:28],addr[27:2],2'b00
    IAddr[31:28] = currentPC[31:28];
    IAddr[27:2] = jump;
    IAddr[1:0] = 0;
end
default:
    IAddr = 8'hFFFFFF;
endcase

```

2. Instruction Memory

指令控制模块最开始要从事先写好的二进制代码读取指令，它由控制单元发出的InsMemRW或者当前PC地址变化时响应，当InsMemRW为0，将指令从IDataOut输出。和单周期的实现相似。

关键代码：

```

initial
begin
    $readmemb( "C:/Users/Sherry/Desktop/rom_data.txt" , rom);
end
always@(InsMemRW or IAddr)

begin
    if(InsMemRW == 0)
        begin
            IDatOut[31:24] = rom[IAddr];
            IDatOut[23:16] = rom[IAddr+1];
            IDatOut[15:8] = rom[IAddr+2];
            IDatOut[7:0] = rom[IAddr+3];
        end
    end

```

3. Control Unit

多周期CPU的控制单元兼顾着状态转移和发出控制信号两个角色。

(1) 状态转移

根据实验原理中图2可以得到状态转移真值表：

当前状态	指令	下一状态
------	----	------

X	RST	IF(000)
IF(000)	X	ID(001)
ID(001)	Add, sub, addi, or, and, sll, slt, sltiu, beq, bltz, sw, lw	EXE(010)
	J, jal, jr, halt	IF(000)
EXE(010)	Add, sub, addi, or, and, ori, sll, slt, sltiu	WB(011)
	Sw, lw	MEM(100)
	Beq, bltz	IF(000)
MEM(100)	Sw	IF(000)
	lw	WB(011)
WB(011)	X	IF(000)

根据状态转移真值表即可得到关键部分代码（代码中的注释由真值表易得）：

```

if(RST == 0)
begin
    state <= 3'b000; // IF
end
else
begin
    if(state == 3'b000) // in IF
        state <= 3'b001; // => ID
    else if(state == 3'b001) // in ID
        begin
            // sub, addi, or,
            // and, ori, sll,
            // slt, sltiu, beq,
            // bltz, sw, lw => EXE 010
            if(op == 6'b000001 || op == 6'b000010 || op == 6'b010000
               || op == 6'b010001 || op == 6'b010010 || op == 6'b011000
               || op == 6'b100110 || op == 6'b100111 || op == 6'b110100
               || op == 6'b110110 || op == 6'b110000 || op == 6'b110001)
                state <= 3'b010;
            // j,jal,jr,halt => IF 000
            else if (op == 6'b111000 || op == 6'b111010
                      || op == 6'b111001 || op == 6'b111111)
                state <= 3'b000;
        end
    else if(state == 3'b010) // in EXE
        begin
            // sub, addi, or,
            // and, ori, sll,
            // slt, sltiu => WB 011
            if (op == 6'b000001 || op == 6'b000010 || op == 6'b010000
               || op == 6'b010001 || op == 6'b010010 || op == 6'b011000
               || op == 6'b100110 || op == 6'b100111)
                state <= 3'b011;
            // sw, lw => MEM 100
        end
    else
        state <= 3'b000;
end

```

```

        else if (op == 6'b110000 || op == 6'b110001)
            state <= 3'b100;
        // beq, bltz => IF 000
        else if (op == 6'b110100 || op == 6'b110110)
            state <= 3'b000;
    end
    else if (state == 3'b011) // in WB => IF
        state <= 3'b000;
    else if (state == 3'b100) // in MEM => WB
    begin
        // sw => IF
        if(op == 6'b110000)
            state <= 3'b000;
        // lw => WB
        else if(op == 6'b110001)
            state <= 3'b011;
    end
end

```

(2) 发出控制信号

控制单元的14个信号的输出，要根据输入的指令和当前状态进行判断。下表写出了多周期CPU各个阶段的工作部件以及相应的控制信号。

阶段	工作部件	控制信号
IF	PC, InsMEM, IR.	IRWre
ID	Registers, WriteRegMux, (nextPC).	WrRegDSrc, PCSrc, RegDst, PCWre(跳转)
EXE	NextPC, ALU, SignZeroExtend, ALUAMux, ALUBMux.	ALUSrcA, ALUSrcB, ExtSel, PCSrc(分支), ALUOp
MEM	DataMem, DBMux.	DBDataSrc, Mrd, Mwr, PCSrc(sw), PCWre(sw)
WB	nextPC, Registers	PCSrc, RegWre.

上面那张表说明了信号变化的条件（即if语句中的内容），至于具体根据什么指令输出什么信号则要参考下表：

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, 相关指令: add、sub、or、and、beq、bltz、slt、sll	来自 sign 或 zero 扩展的立即数, 相关指令: 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	

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述	对应指令
000	$Y = A + B$	加	Addi, j, Jal, Jr,

			Sw, lw
001	$Y = A - B$	减	Sub
010	$Y = (A < B) ? 1: 0$	比较 A 与 B 不带符号	sltiu
011	$Y = (((regA < regB) \&\& (regA[31] == regB[31])) ((regA[31] == 1 \&\& regB[31] == 0))) ? 1: 0$	比较 A 与 B 带符号	slt
100	$Y = B \ll A$	B 左移 A 位	sll
101	$Y = A \vee B$	或	Or, ori
110	$Y = A \wedge B$	与	And, andi
111	$Y = A \square B$	异或	Beq bltz

结合上面两张表，可以得到 CU 输出所有信号的条件和值，具体代码如下：

```

always@(RST or zero or sign or op or state)
begin
    // EXE: ALUSrcA: sll => 1; else: add, sub, addi, or, and, ori, beq, bltz,
    slt, sltiu, sw, lw => 0;
    if(state == 3'b010 && op == 6'b011000)
        ALUSrcA = 1;
    else
        ALUSrcA = 0;
    // EXE: ALUSrcB: addi, ori, sltiu, lw, sw => 1; else : add, sub, or, and,
    beq, blitz, slt, sll => 0;
    if(state == 3'b010 && (op == 6'b000010 || op == 6'b010010 || op ==
    6'b100111 || op == 6'b110001 || op == 6'b110000))
        ALUSrcB = 1;
    else
        ALUSrcB = 0;
    // MEM: DBDataSrc:lw => 1; else: add, sub, addi, or, and, ori, slt, sltiu,
    sll => 0
    if(state == 3'b100 && op == 6'b110001)
        DBDataSrc = 1;
    else
        DBDataSrc = 0;
    // WB 阶段或 ID 阶段译码得到调用子程序指令 jal => 1
    if(state == 3'b011 || (state == 3'b001 && op == 6'b111010))
        RegWre = 1;
    else
        RegWre = 0;
    // ID: WrRegDSrc: jal => 0
    if(state == 3'b001 && op == 6'b111010)
        WrRegDSrc = 0;
    else
        WrRegDSrc = 1;
    // MEM: mRD: lw => 1
    if(state == 3'b100 && op == 6'b110001)
        mRD = 1;
    else
        mRD = 0;
    // MEM: mWR: sw => 1
    if(state == 3'b100 && op == 6'b110000)
        mWR = 1;
    else
        mWR = 0;
    // IF: IRWre=>1
    if(state == 3'b000)
        IRWre = 1;
    else
        IRWre = 0;

```

```

// EXE: ori, slitu: ExtSel => 0
if(state == 3'b010 && (op == 6'b010010 || op == 6'b100110))
    ExtSel = 0;
else
    ExtSel = 1;
// ID: j, jal => 11, jr => 10
if(state == 3'b001)
begin
    if(op == 6'b111000 || op == 6'b111010)
        PCSrc = 2'b11;
    else
        PCSrc = 2'b10;
end
// EXE: beq(zero = 1),bltz(sign = 1, zero = 0)
if(state == 3'b010)
begin
    if((op == 6'b110100 && zero == 1)
    ||(op == 6'b110110 && sign == 1 && zero == 0))
        PCSrc = 2'b01;
    else
        PCSrc = 2'b00;
end
// MEM: PCSrc:sw => 00
if(state == 3'b100 && op == 6'b110000)
    PCSrc = 2'b00;
// WB: PCSrc:
if(state == 3'b011)
    PCSrc = 2'b00;
// ID: jal => 00
if(state == 3'b001 && op == 6'b111010)
    RegDst = 2'b00;
// WB: addi, ori, sltiu, lw => 01, else => 10
if(state == 3'b011)
begin
    if(op == 6'b000010 || op == 6'b010010 || op == 6'b100111 || op ==
6'b110001)
        RegDst = 2'b01;
    else
        RegDst = 2'b10;
end
if(state == 3'b010)
begin
    if(op == 6'b000001) // sub
        ALUOp=3'b001;
    else if(op == 6'b100111)// sltiu
        ALUOp = 3'b010;
    else if(op == 6'b100110) // slt
        ALUOp = 3'b011;
    else if(op == 6'b011000) // sll
        ALUOp = 3'b100;
    else if((op == 6'b010010)|| (op == 6'b010000)) // ori, or
        ALUOp = 3'b101;
    else if(op == 6'b010001) // and
        ALUOp = 3'b110;
    else if((op == 6'b110100)|| (op == 6'b110110)) // beq, bltz
        ALUOp = 3'b111;
    else
        ALUOp = 3'b000;
end
else
    ALUOp=3'b000;
if(state == 3'b001) begin // 在ID阶段j, jal, jr都要让PCWre = 1
    if(op == 6'b111000 || op == 6'b111010 || op == 6'b111001)
        PCWre <= 1;
    else
        PCWre <= 0;
end

```

```

else if (state == 3'b010) begin // 在EXE阶段, beq,bltz 要让PCWre = 1
    if(op == 6'b110100 || op == 6'b110110)
        PCWre <= 1;
    else
        PCWre <= 0;
end
else if (state == 3'b011) // 在WB阶段, PCWre = 1
    PCWre = 1;
else if (state == 3'b100) begin // 在MEM阶段, sw 要让PCWre = 1;
    if(op == 6'b110000)
        PCWre <= 1;
    else PCWre <= 0;
end
else // halt
    PCWre <= 0;
end

```

4. Register File

寄存器组的实现与单周期CPU相似，在程序之初将32个32bit寄存器赋初值0，寄存器组在输入的寄存器号不为0号寄存器时读取对应寄存器的值，且读取寄存器的值不需要信号触发。写信号在时钟的下降沿或者reset的下降沿触发，如果重置信号有效（reset==0），则清空寄存器组，每个寄存器赋值0，否则当控制单元输出了写寄存器的信号而且输入的被写入寄存器号有效（非0）时，就将Write Data的内容写进指定WriteReg对应的寄存器。

关键代码如下：

```

assign Read_Data1 = (Read_Reg1 == 0) ? 0 : RegFile[Read_Reg1];
assign Read_Data2 = (Read_Reg2 == 0) ? 0 : RegFile[Read_Reg2];

initial begin
    for (i = 0; i < 32; i = i + 1)
        RegFile[i] = 0;
end
always @ (negedge clk or negedge reset)
begin
    if(reset == 0)
    begin
        for(i = 1; i < 32; i = i + 1)
            RegFile[Write_Reg] <= 0;
    end
    else if(RegWre == 1 && Write_Reg != 0) //写
        RegFile[Write_Reg] <= Write_Data;
end

```

5. ALU

根据ALU真值表对A, B两个输入的操作数进行加、减、或、与……的操作并从result输出操作后的结果，并判断result是否为0输出zero，将result最高作为sign信号输出。

```

case (ALUOp)
    3'b000://add
        result = InA+InB;
    3'b001://minus
        result = InA-InB;
    3'b010:
        result = (InA < InB) ? 1:0;
    3'b011:// or
begin
    if(InA < InB && (InA[31] == InB[31]))
        result = 1;
    else if (InA[31] == 1 && InB[31] == 0)

```

```

        result = 1;
    else
        result = 0;
    end
3'b100:// and
result = InB << InA;
3'b101: // < unsigned
result = InA|InB;
3'b110: // < signed
result = InA & InB;
3'b111: // xor
result = InA ^ InB;
default:
begin
    result = 32'h00000000;
    $display( "no match");
end
endcase

```

6. sign, zero extend

根据ExtSel信号判断进行位扩展还是符号扩展。

ExtSel	立即数的最高位	操作
1	0	将输出数据的16-31位补全为0
1	1	将输出数据的16-31位补全为1
0	X	将输出数据的16-31位补全为0

关键代码:

```

assign DataOut[15:0] = immediate[15:0];
assign DataOut[31:16] = (ExtSel == 1) ? ((immediate[15] == 1) ?
16'hFFFF: 16'h0000):16'h0000;

```

7. Data Memory

根据控制单元发出的mWR和mRD信号判断是否读以及是否写，其中读不需要信号触发，将数据存储单元的中DAddr号的单元的值传出给DataOut；而写信号则需要时钟下降沿触发，并要把存储器外部输入的数据DataIn写在存储器以DAddr为起始地址对应的四个存储器单元。

关键代码如下：

```

assign DataOut[7:0] = (nRD==1)?ram[DAddr + 3]:8'bz; // z 为高阻态
assign DataOut[15:8] = (nRD==1)?ram[DAddr + 2]:8'bz;
assign DataOut[23:16] = (nRD==1)?ram[DAddr + 1]:8'bz;
assign DataOut[31:24] = (nRD==1)?ram[DAddr]:8'bz;

always@( negedge clk )
begin // 用时钟下降沿触发写存储器
    if( nWR==1 )
    begin
        ram[DAddr] <= DataIn[31:24];
        ram[DAddr+1] <= DataIn[23:16];
        ram[DAddr+2] <= DataIn[15:8];
        ram[DAddr+3] <= DataIn[7:0];
    end
end

```

8. 2 to 1 MUX(32bit)

这段代码在顶层模块的实现中被复用了4次，分别为：寄存器组的writeData输入，ALUA和ALUB输入，选择DB写回数据是来自ALU的结果还是数据存储器的值。

实现如下：

```
assign DataOut = (signal == 0) ? DataIn1 : DataIn2;
```

9. 4 to 1 MUX(5bit)

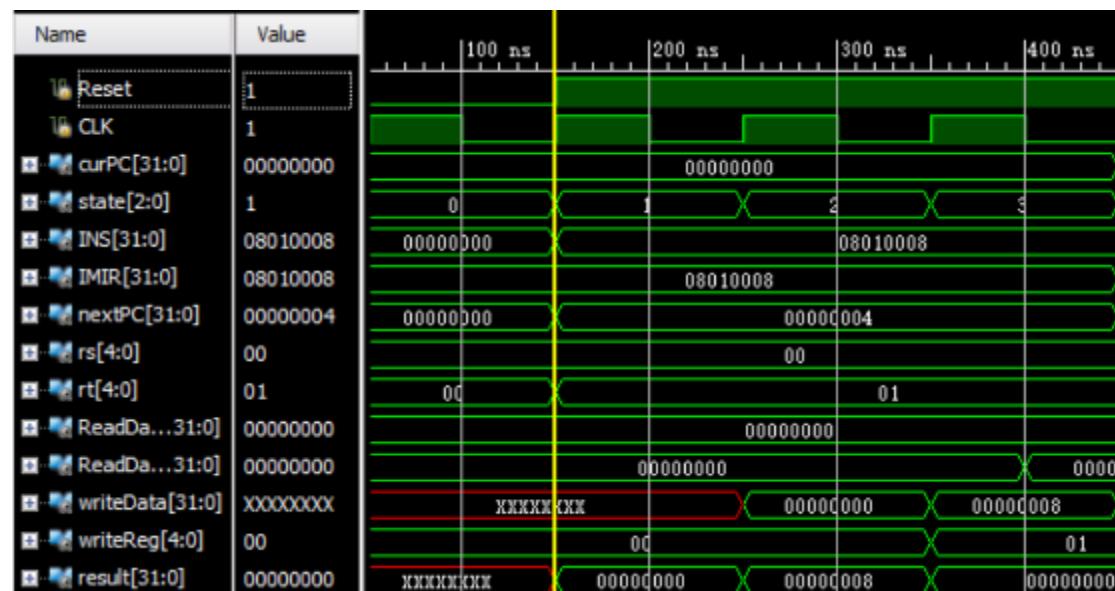
这段代码在选择写回寄存器时使用，同为数据选择器，实现与序号8大同小异。关键代码如下：

```
assign WriteReg = (RegDst == 2'b00) ? 5'b11111 : (RegDst == 2'b01) ?  
rt : rd;
```

(二) 验证CPU正确性

1. 0x00000000指令: addi \$1,\$0,8

(1) 仿真结果:



(2) 分析:

IF阶段: 这是第一条指令，当前PC值（curPC）为0. 且IMIR（即InsMem）已经输出了PC=0时的指令。

ID阶段: 当Reset信号置为1时，读到指令（INS = 08010008），rt, rs获得寄存器号\$1, \$0，同时ReadData1和ReadData2读到两个寄存器的值均为0. 同时reset的上升沿使nextPC的值也被计算出来，因为是addi指令，下一个PC的值应为PC+4 = 0000 0004。

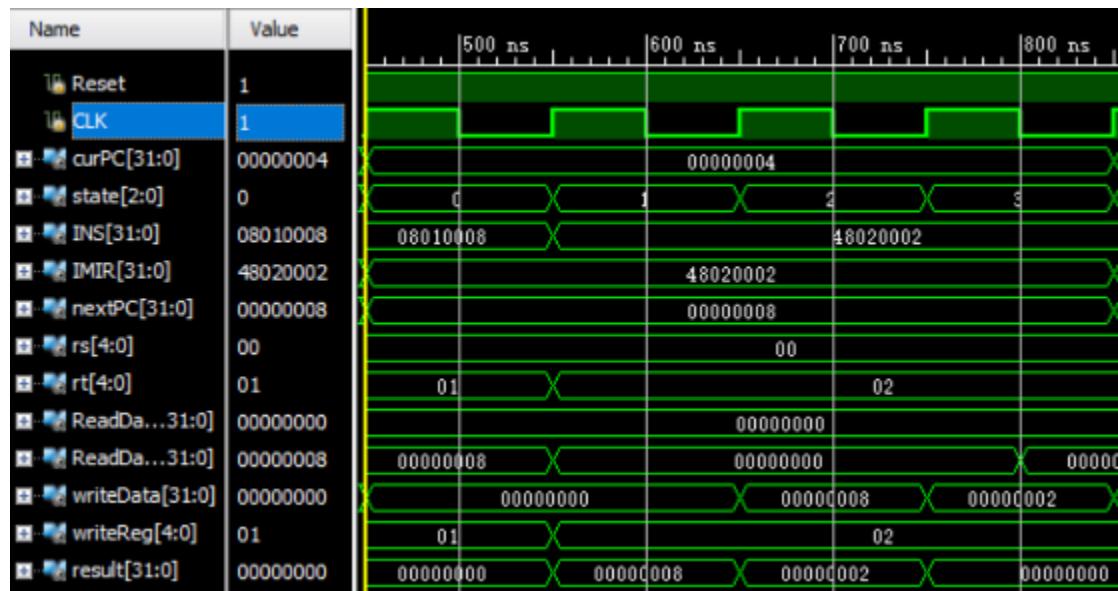
EXE阶段: 状态转移到2，result得到结果8。

WB阶段: 状态转移到3，写回写段要写的数据为EXE阶段result得到的结果8，要写回的

寄存器为 $rt = \$1$ 。

2. $0x00000004$ 指令: $ori \$2,\$0,2$

(1) 仿真结果:



(2) 分析:

IF阶段: 当前PC值 (curPC) 为4, 由于是ori运算, 所以下一个PC应当是当前PC地址+4, 为0000 0008。CurPC和nextPC经检验正确。

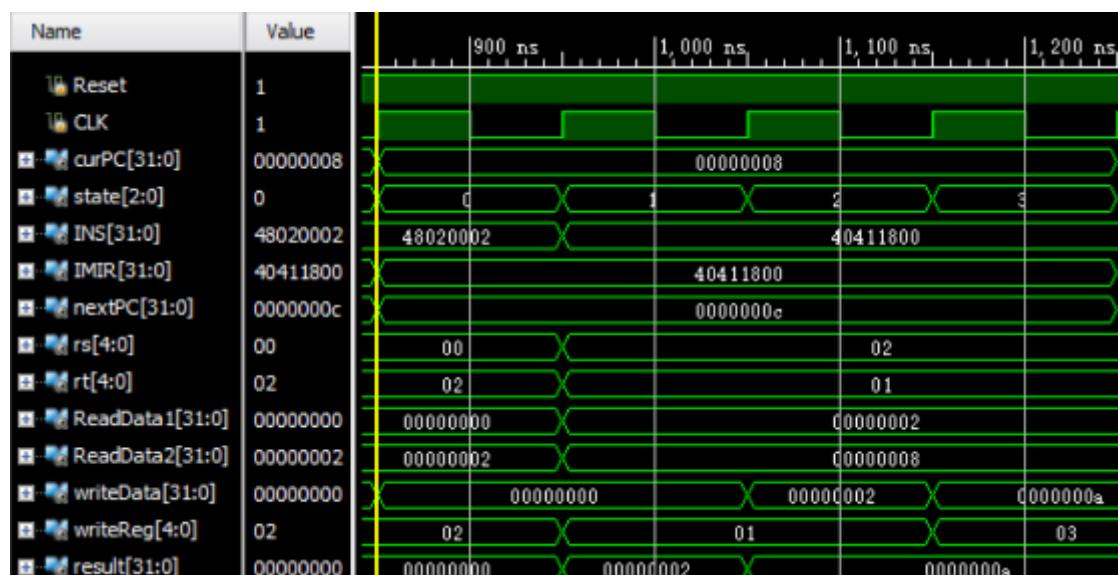
ID阶段: 将从InsMem中读到的指令从IR中输出, $rs = \$0$, $rt = \$2$, 因为这两个寄存器的值都没有被修改过, $ReadData1 = ReadData2 = 0$.

EXE阶段: 进行或运算, 得到 $result = 2$.

WB阶段: 写回步骤中writedata为EXE阶段result的结果, 为2. 写回的寄存器为 $rt = \$2$.

3. $0x00000008$ 指令: $add \$3,\$2,\$1$

(1) 仿真结果:



(2) 分析:

IF阶段: InsMem读到PC = 8的指令 (IMIR);

ID阶段: IR输出了最新指令 (INS), 译码得到rs = \$2, rt = \$1. 其中\$2的值在序号(2)中被修改为2, \$1的值在序号(1)中被修改为8.

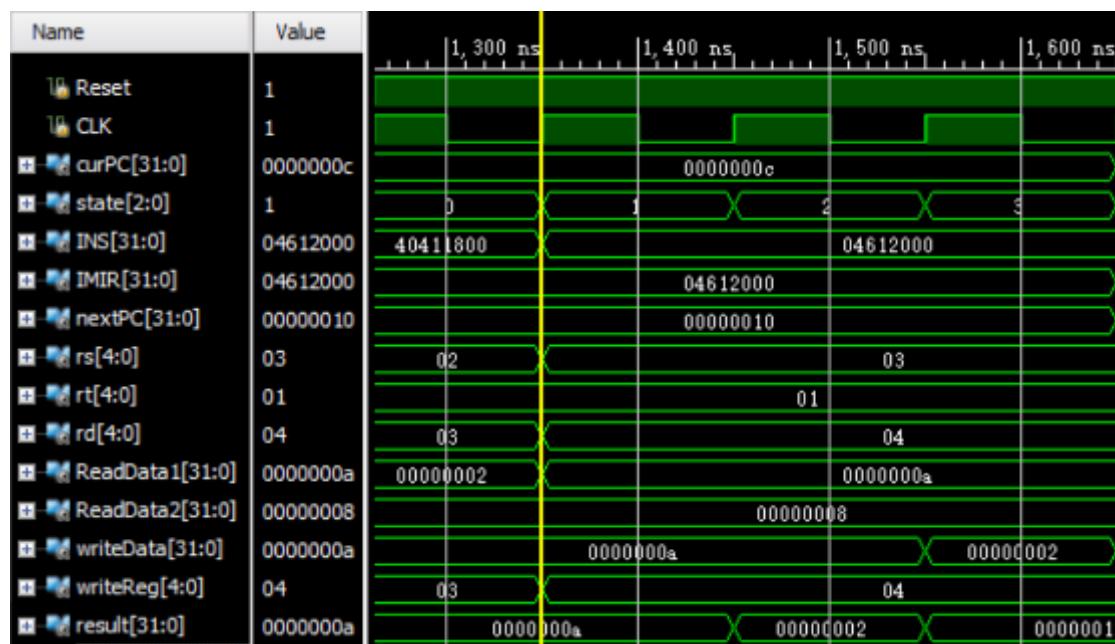
EXE阶段: result计算的得到ReadData1和ReadData2的add结果为10 (a)。

WB阶段: 将EXE阶段result的值为a写入寄存器\$3

【由于所有的**IF阶段**和**ID阶段**从IR中输出指令的步骤在每一条指令中效果一样, 下面的指令描述中略去】

4. 0x0000000C指令: sub \$4,\$3,\$1

(1) 仿真结果:



(2) 分析:

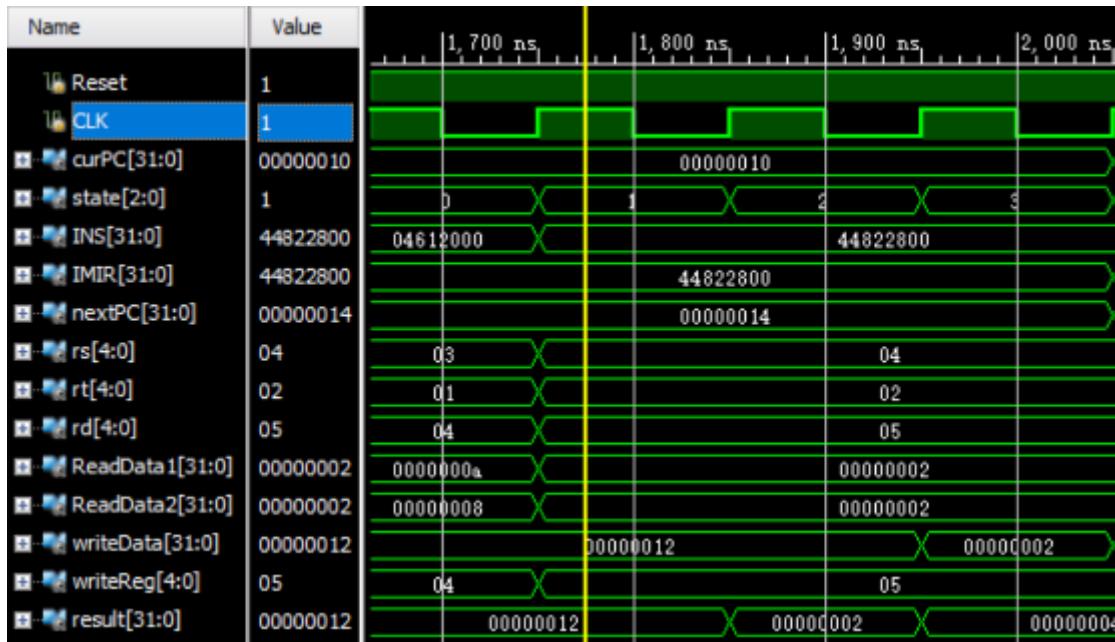
ID阶段: 译码得到rs = \$3, rt = \$1, rd = \$4. 由于\$3的值在序号3的指令中被修改为a, \$1的值在序号1的指令中被修改为8.

EXE阶段: result = \$3 - \$1 = 10 - 8 = 2.

WB阶段: writeData为EXE阶段result的值为2, 目标寄存器为rd = \$4.

5. 0x00000010指令: and \$5,\$4,\$2

(1) 仿真结果:



(2) 分析:

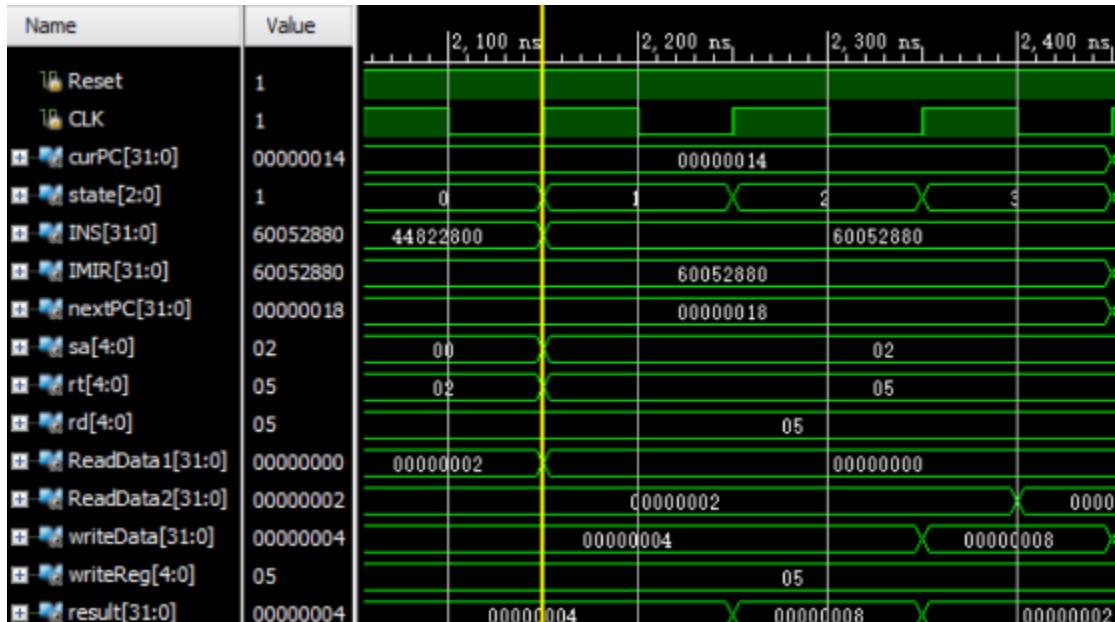
ID阶段: 译码得到 $rs = \$4$, $rt = \$2$, $rd = \$5$. 由于\$4的值在序号4的指令中被修改为2, \$2的值在序号2的指令中被修改为2, $ReadData1 = ReadData2 = 2$.

EXE阶段: $result = \$4 \& \$2 = 2 \& 2 = 2$.

WB阶段: $writeData$ 为EXE阶段 $result$ 的值为2, 目标寄存器为 $rd = \$5$.

6. 0x00000014指令: sll \$5,\$5,2

(1) 仿真结果:



(2) 分析:

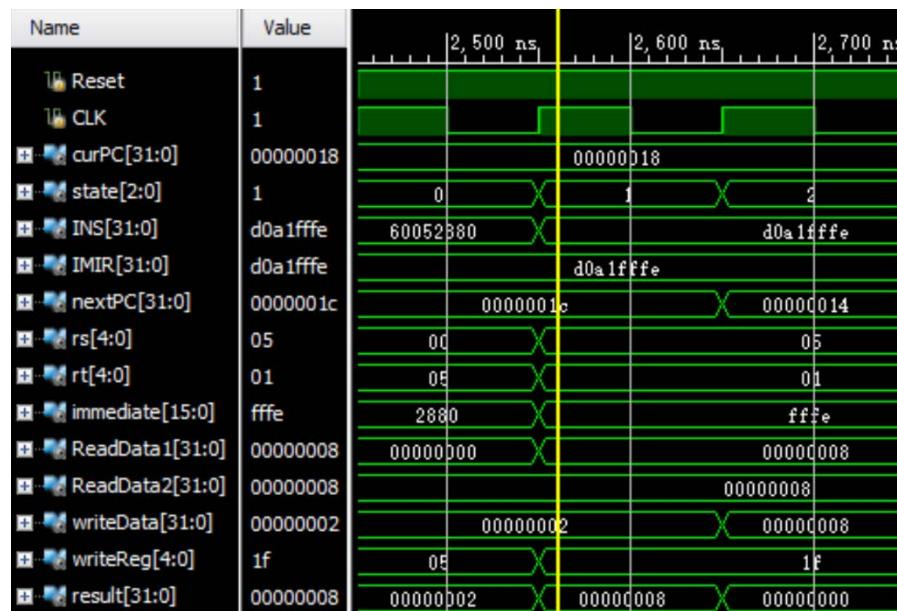
ID阶段: 译码得到 $rt = \$5$, $rd = \$5$, $sa = 2$. 由于\$5的值在序号5的指令中被修改为2, $ReadData2 = 2$. (这里rs未用为0\$0, 所以 $ReadData1$ 的结果为0)

EXE阶段: result = \$5 << 2 = 2 << 2 = 8.

WB阶段: writeData为EXE阶段result的值为8, 目标寄存器为rd = \$5.

7. *0x00000018指令: beq \$5,\$1,-2*

(1) 仿真结果:



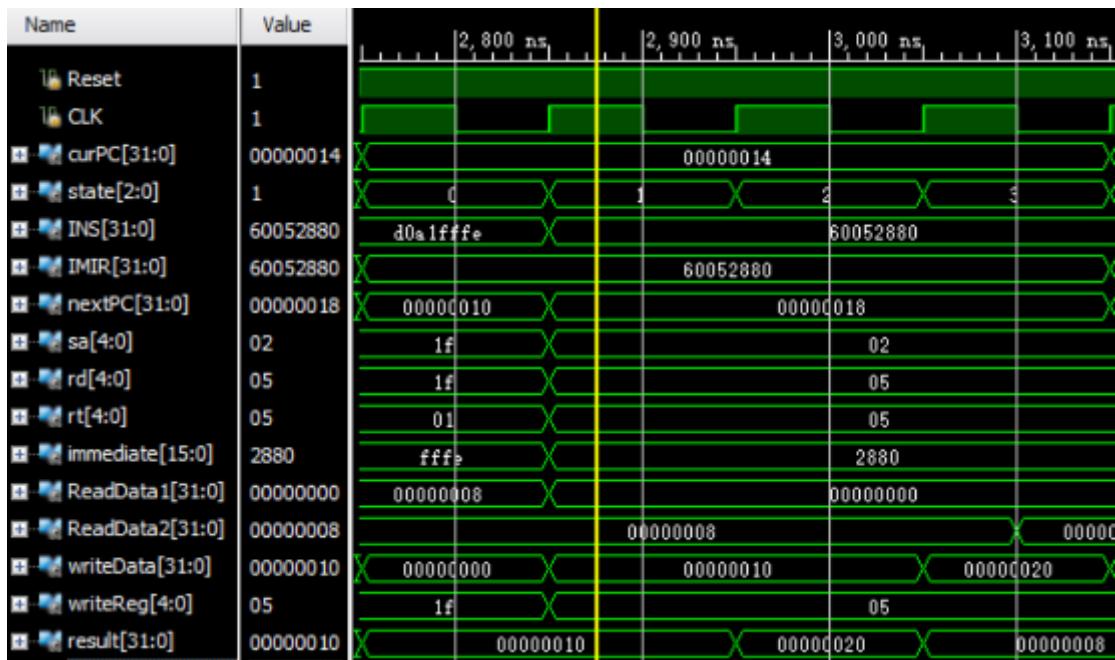
(2) 分析:

ID阶段: 译码得到rs = \$5, rt = \$1, immediate = -2. 由于\$5的值在序号6的指令中被修改为8, \$1的值在序号1的指令中被修改为8, ReadData1 = ReadData2 = 8.

EXE阶段: $\because (\$5) = (\$1), \therefore \text{result} = 0, \text{zero} = 1, \text{nextPC} = \text{pc} + 4 + (\text{sign} - \text{extend})\text{immediate} \ll 2 = (18)_{16} + 4 - 8 = (14)_{16}$

8. *0x00000014指令: sll \$5,\$5,2*

(1) 仿真结果:

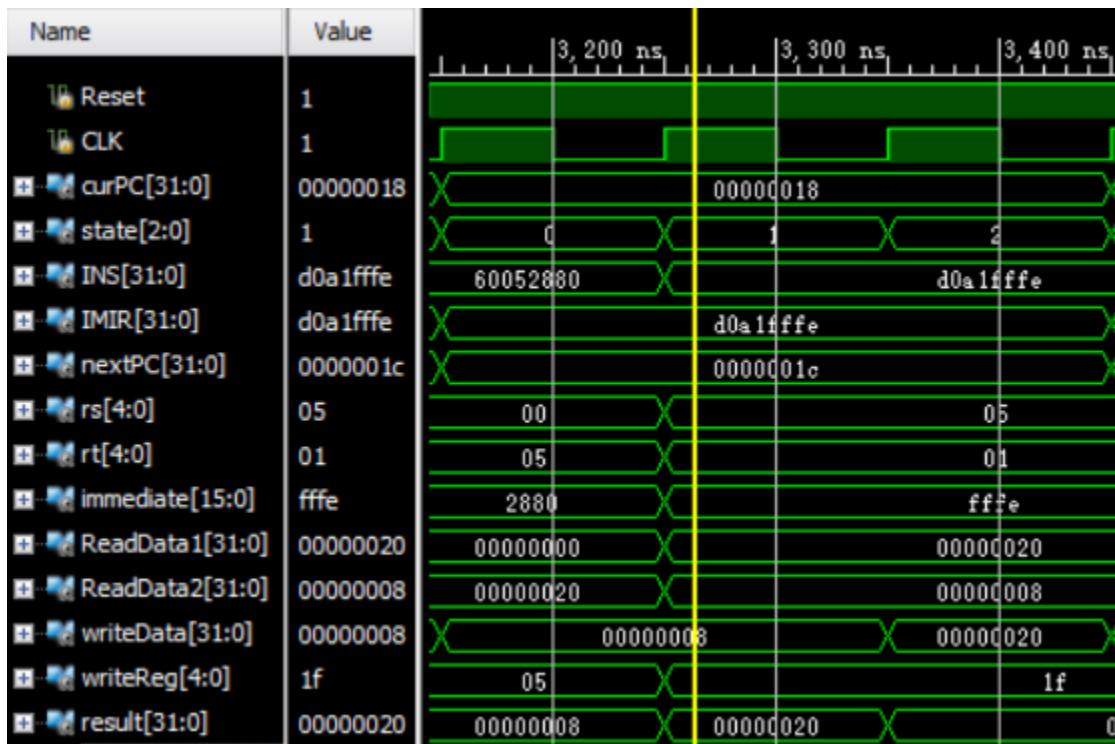


(2) 分析:

同序号6，只是最初读寄存器的值变为了8. EXE阶段：计算得到的是 $8 \ll 2 = (10\ 0000)_2$. WB阶段：将结果20写回寄存器\$5.

9. 0x00000018指令: beq \$5,\$1,-2

(1) 仿真结果:



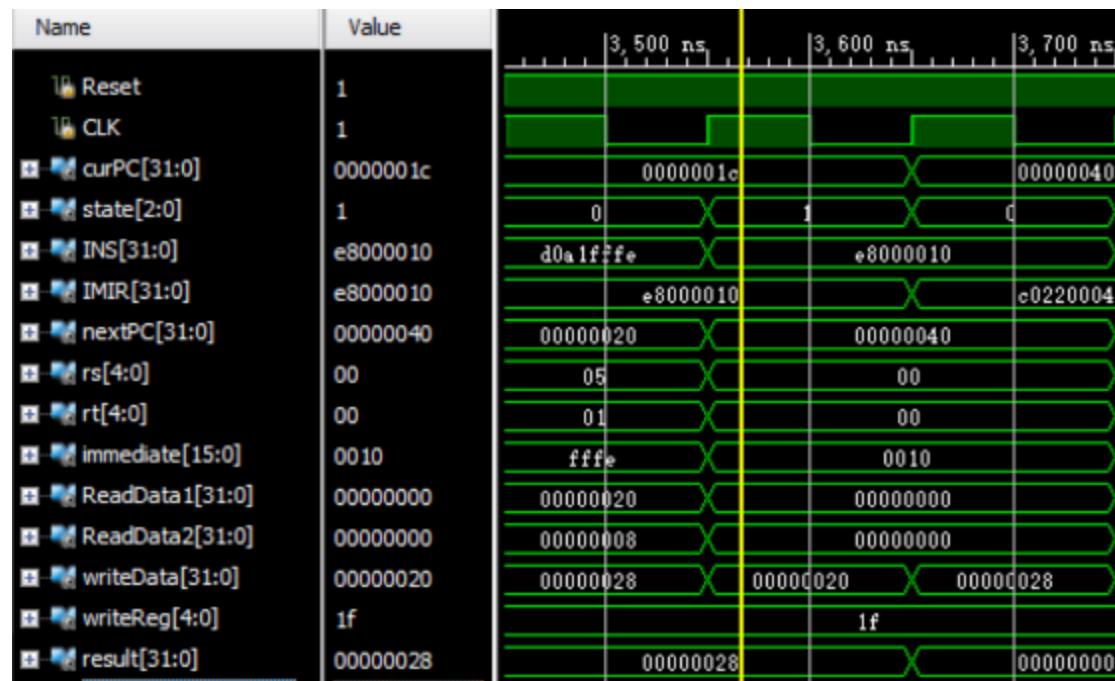
(2) 分析:

同序号7，只是在序号8中，\$5的值被修改为 $20 \neq (\$1) = 8$. EXE阶段result $\neq 0$, zero =

0. 不进行分支转移, $\text{nextPC} = (1C)_{16}$.

10. $0x00000001C$ 指令: $\text{jal } 0x00000040$

(1) 仿真结果:

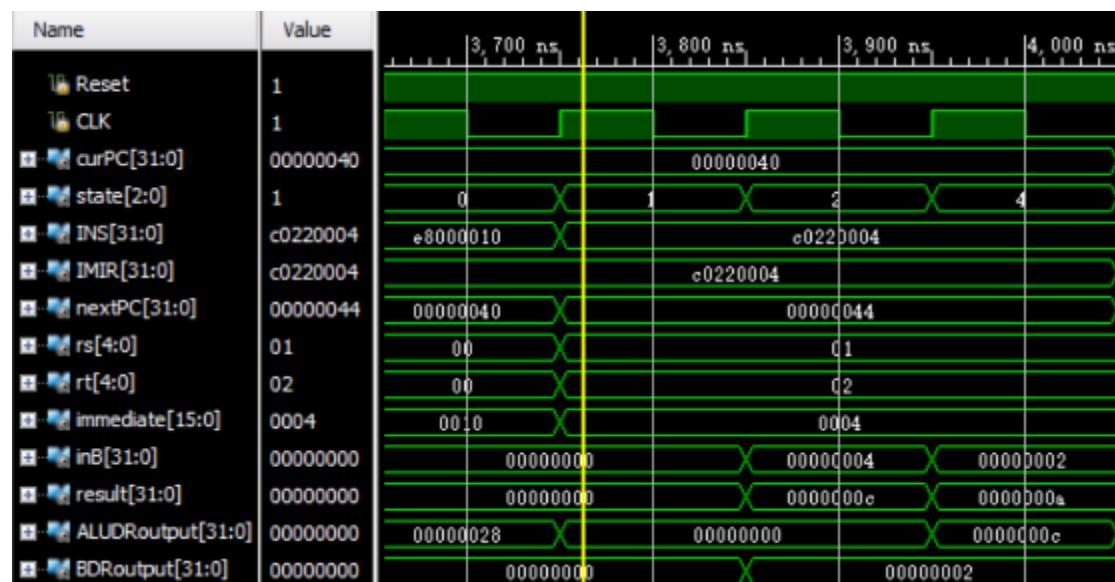


(2) 分析:

ID阶段: 计算出nextPC的值0x0000 0040. 同时将PC+4的值20写入寄存器\$31.

11. $0x000000040$ 指令: $sw \$2,4(\$1)$

(1) 仿真结果:



(2) 分析:

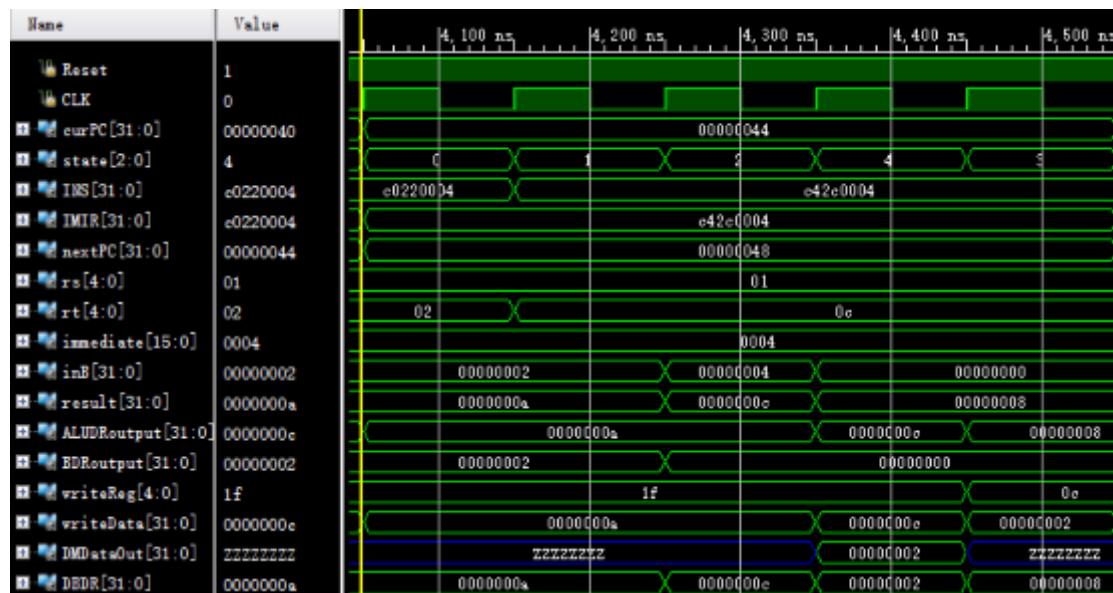
ID阶段: 译码得到rt = \$2, rd = \$1, immediate= 4.

EXE阶段: result = (\$1) + 4 = c, 计算出在数据存储器中的地址为12.

MEM阶段: BDRoutput为(\$2)=2, 作为要向数据存储器写入的数据; ALUDRoutput为EXE阶段result的结果, 为存储器的目标地址。

12. 0x00000044指令: lw \$12,4(\$1)

(1) 仿真结果:



(2) 分析:

ID阶段: 译码得到rs = \$1, rt = \$12, immediate= 4.

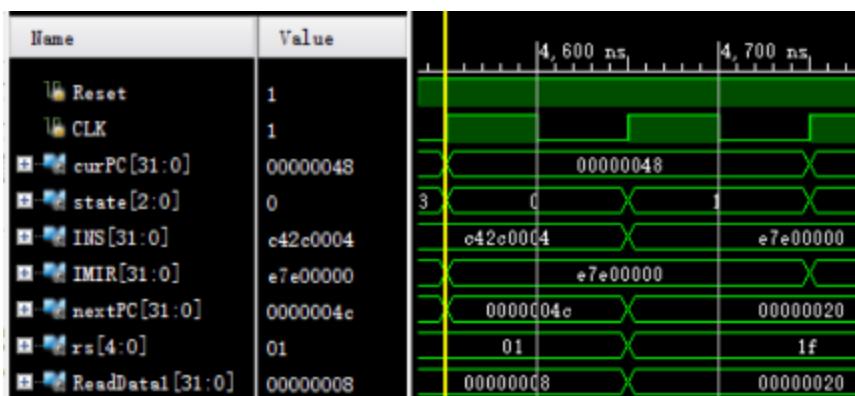
EXE阶段: result = (\$1) + 4 = c, 计算出在数据存储器中的地址为12.

MEM阶段: 数据存储器根据ALU计算的result作为地址读出相应的数据并输出到DB, 如图中DMDdataOut是数据存储器的输出数据, 为2, 与上一步(序号12)中存入数据存储器的数据一致, DBDR是数据选择器存入寄存器DBDR的数据。

WB阶段: 将数据存储器输出的数据写回到寄存器中, 指令要求写回存储器\$12, 途中WriteReg为C, writeData为2, 与指令相符。

13. 0x00000048指令: jr \$31

(1) 仿真结果:

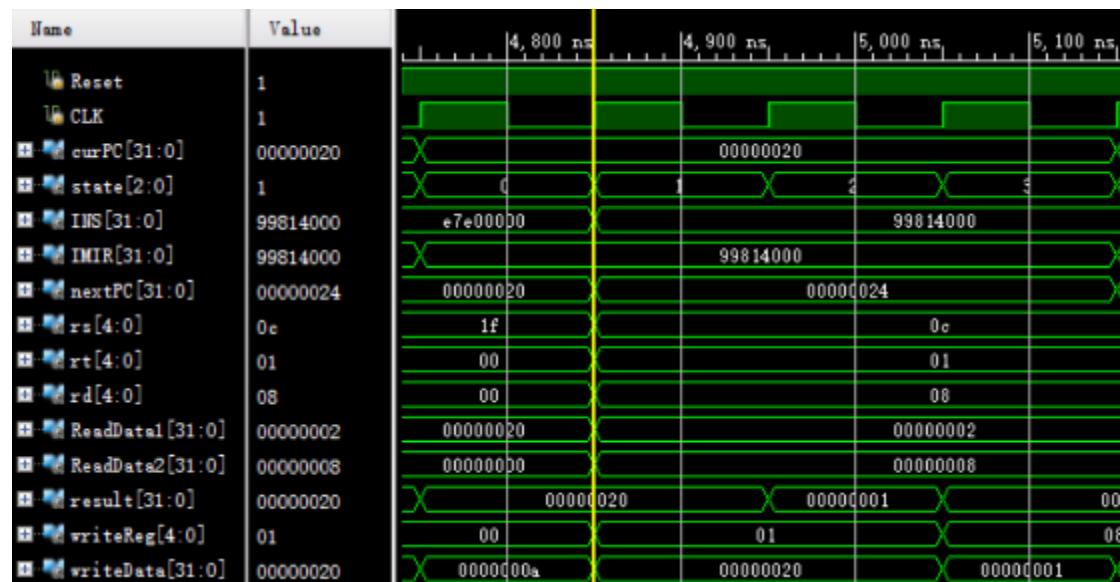


(2) 分析:

ID阶段: nextPC = \$31, 由图在ID阶段, rs变为31, readData1和nextPC都变为调用子程序时存入的下一条指令的地址0x00000020了。

14. 0x00000020指令: slt \$8,\$12,\$1

(1) 仿真结果:



(2) 分析:

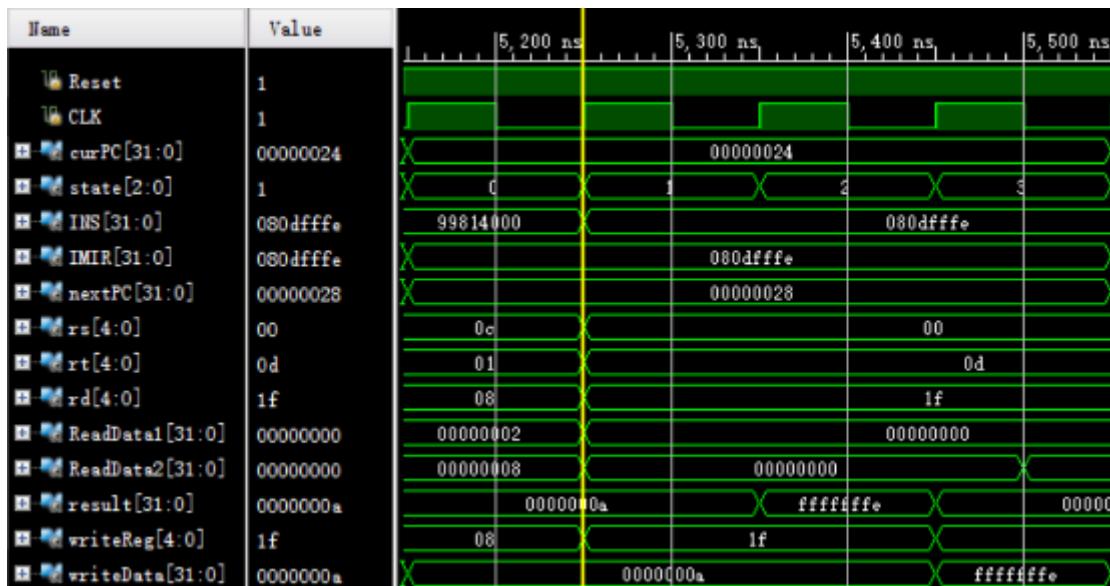
ID阶段: 译码得到rs = \$12, rt = \$1, rd = \$8. 由于在序号12的指令中将寄存器\$12的值赋为了2, 序号1的指令将寄存器\$1的值赋值为8, 因此ReadData1应为2, ReadData2应为8. 仿真结果正确。

EXE阶段: 比较rs和rt的数据, $\because 2 < 8$, result = 1.

WB阶段: writeReg = rd = 1, writeData为EXE阶段计算出的结果1, 与仿真结果相符。

15. 0x00000024指令: addi \$13,\$0,-2

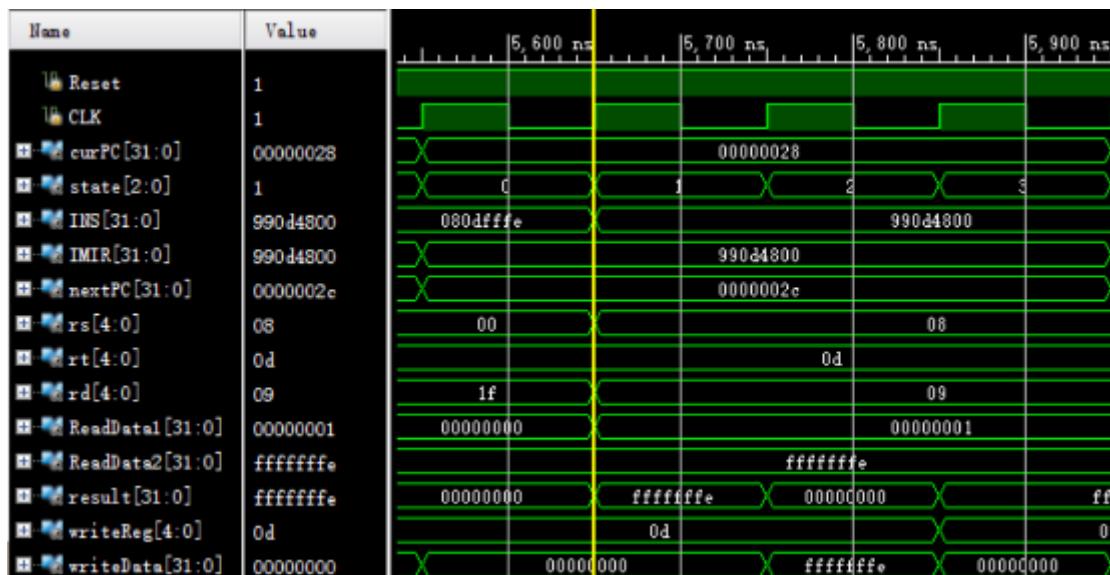
(1) 仿真结果:



(2) 分析:

同序号1，ID阶段Rs = 0, rt = 13, ReadData1 = 0, ReadData2 = 0; EXE阶段算的结果为-2; WB阶段writeReg = rd = \$13, writeData为EXE阶段的result = -2。

16. 0x00000028指令: slt \$9,\$8,\$13

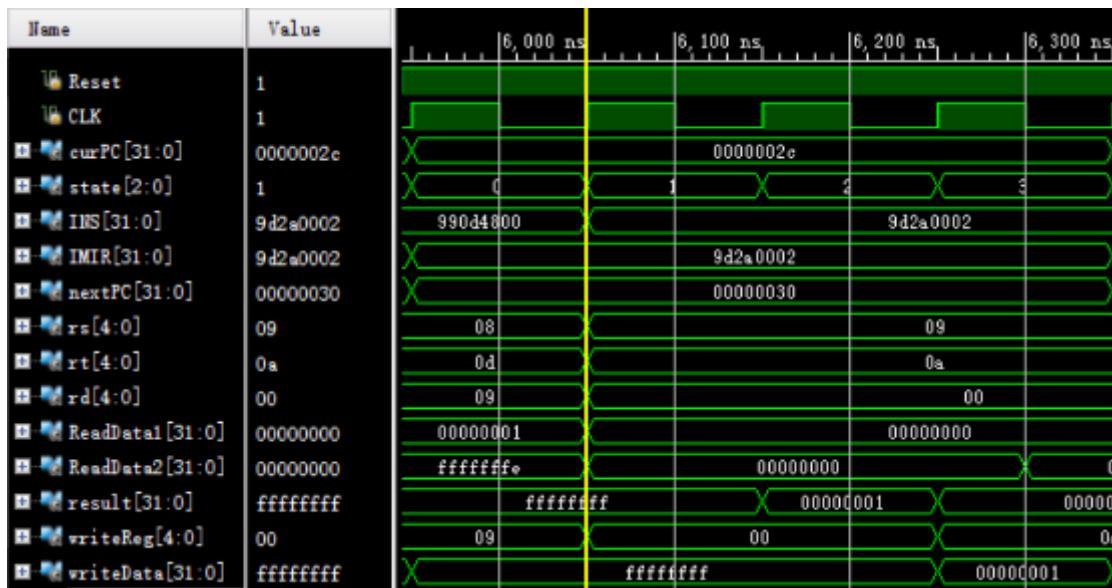


分析:

同序号14，ID阶段: Rs = 8, rt = 13, rd = 9, ReadData1 = 1, ReadData2 = -2; EXE阶段: Result = 0; WB阶段: writeData = result = 0; writeReg = \$9.

17. 0x0000002C指令: sltiu \$10,\$9,2

(1) 仿真结果:



(2) 分析:

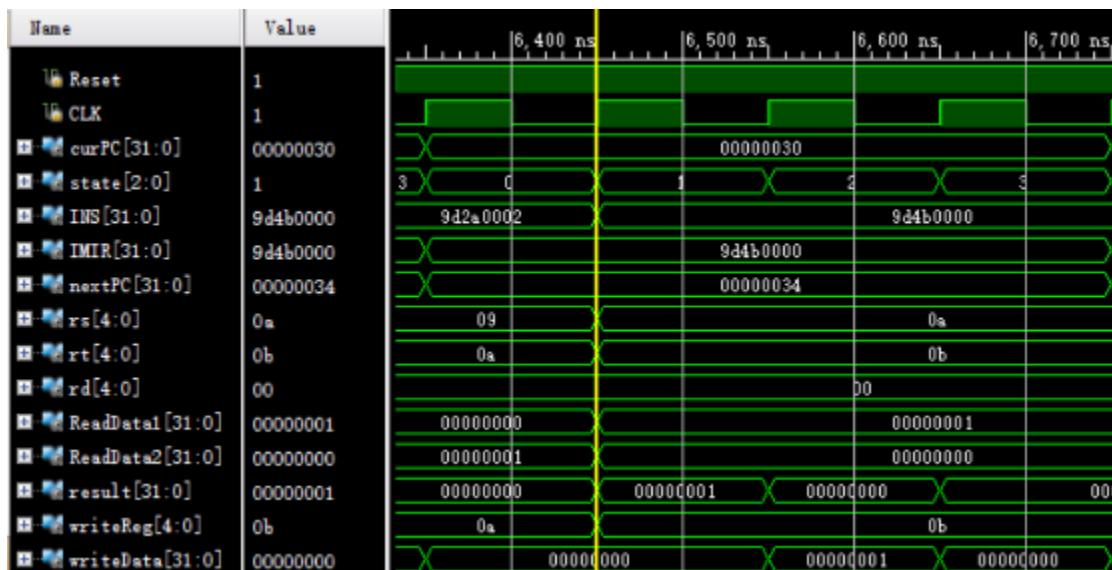
ID阶段: 译码得到 $rs = \$9$, $rt = \$10$. 由于\$9在序号16中被写为0, \$10从初始化后没有做过修改, 因此 $ReadData1 = ReadData2 = 0$.

EXE阶段: 比较rs和rt的数据, $\because 0 < 2$, $result = 1$.

WB阶段: $writeReg = rt = a$, $writeData$ 为EXE阶段计算出的结果1, 与仿真结果相符。

18. 0x00000030指令: sltiu \$11,\$10,0

(1) 仿真结果:

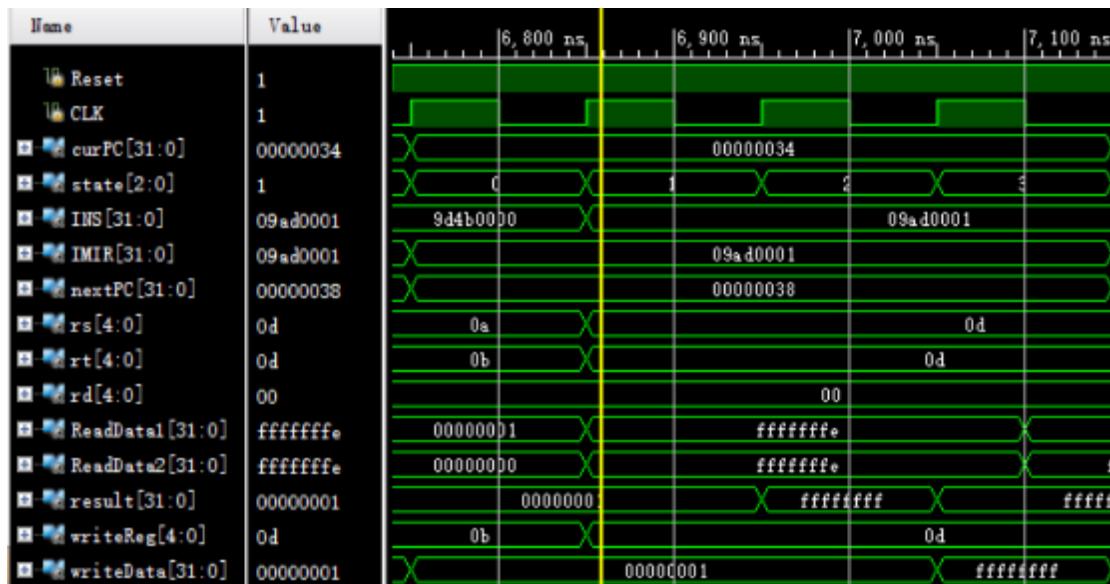


(2) 分析:

同序号17, ID阶段译码得到 $Rs = b$, $rt = a$, $ReadData1 = 0$ (从初始化到目前未作更改), $ReadData2 = 1$ (序号17中更改为1); EXE阶段: 由于 $1 > 0$, 计算得到 $Result = 0$; WB阶段 $WriteData = result = 0$, $writeReg = rs = b$; 仿真结果正确。

19. 0x00000034指令: addi \$13,\$13,1

(1) 仿真结果:

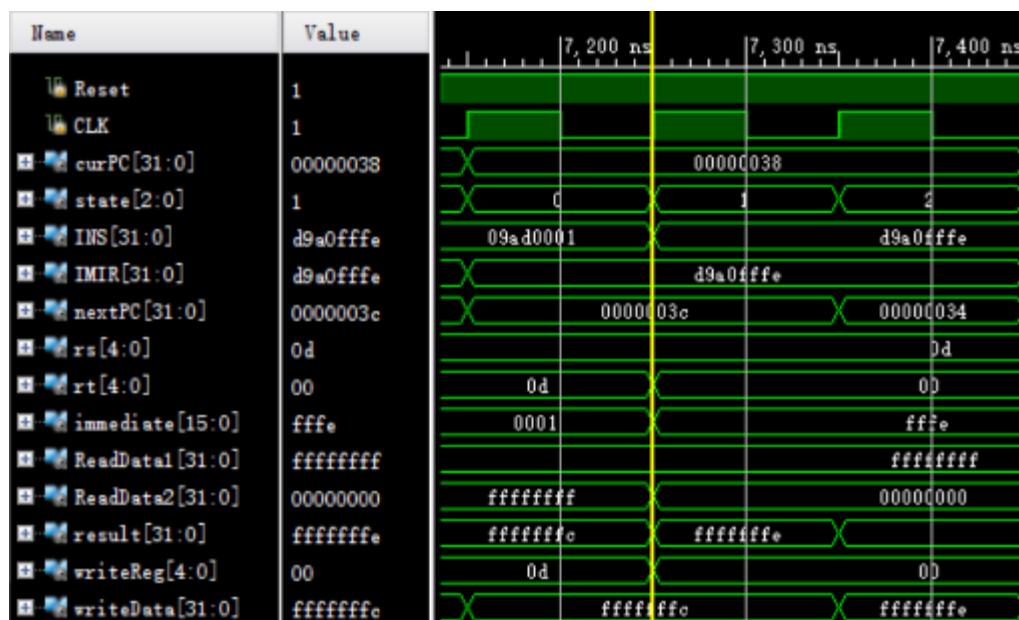


(2) 分析:

同序号1, ID阶段译码得到rs = \$d, rt = \$d, ReadData1 = -2, ReadData2 = -2; EXE阶段 result = -2+1 = -1; WB阶段writeReg = \$d, writeData = -1. 与仿真结果相符。

20. 0x00000038指令: bltz \$13,-2 (<0, 34)

(1) 仿真结果:



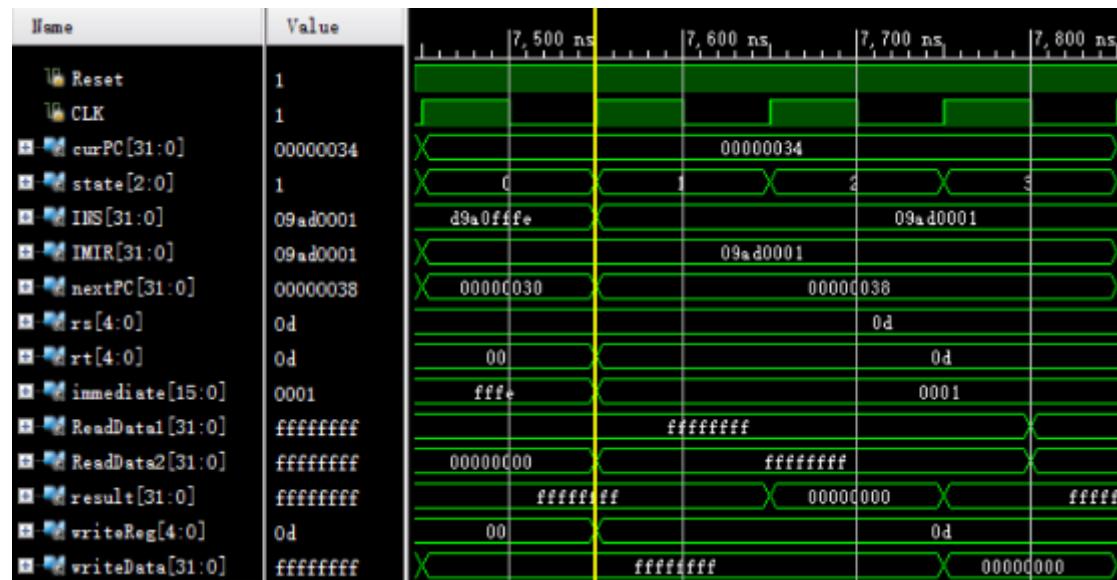
(2) 分析:

ID阶段: 译码得到Rs = 13, rt = 0, 由于在序号19中\$d被赋值为-1. ReadData1 = -1, ReadData2 = 0.

EXE阶段: 由于-1<0, PC = PC + 4 + (-2) <<2 = PC + 4 - 8 = 34.

21. 0x00000034指令: addi \$13,\$13,1

(1) 仿真结果:

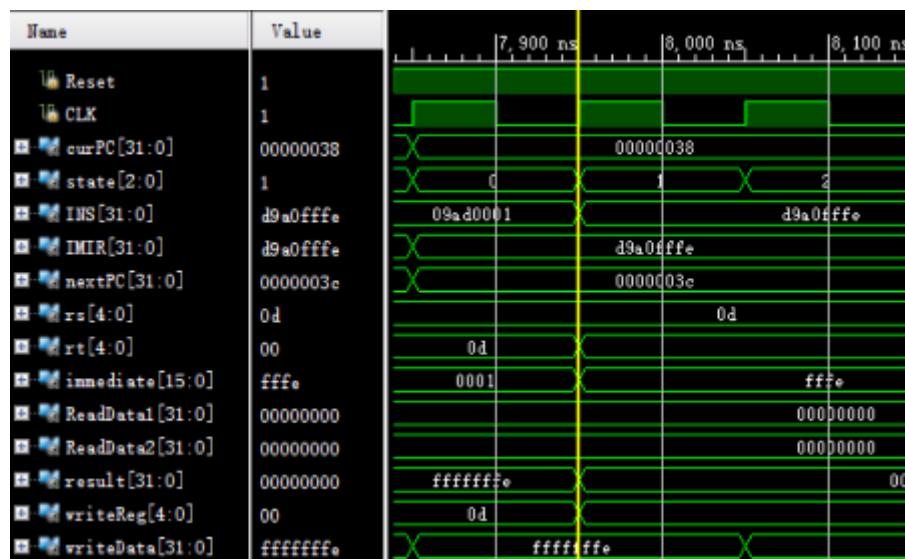


(2) 分析:

同19, 只是在序号19的指令中, \$d的值被修改为1. EXE阶段result = -1 + 1 = 0.

22. 0x00000038指令: bltz \$13,-2

(1) 仿真结果:

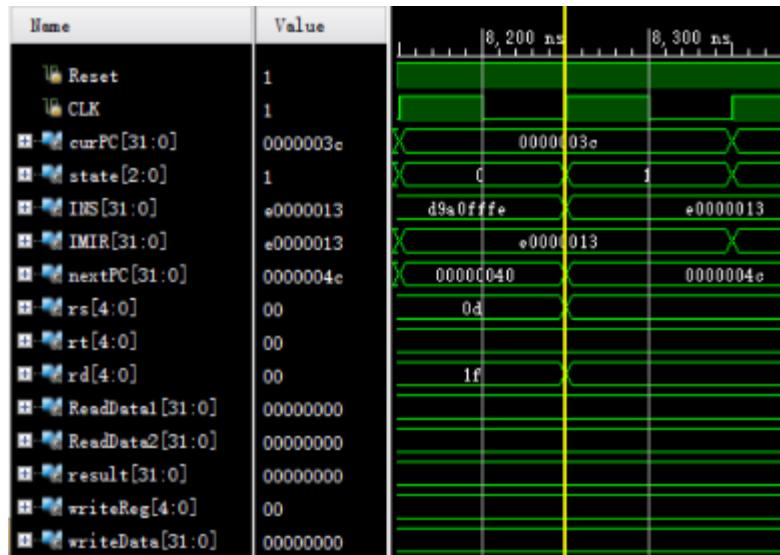


(2) 分析:

同序号20, 只不过序号21中将\$d赋值为0, EXE阶段由于 $0 = 0$, PC = PC + 4. NextPC = 3C

23. 0x0000003C指令: j 0x0000004C

(1) 仿真结果:

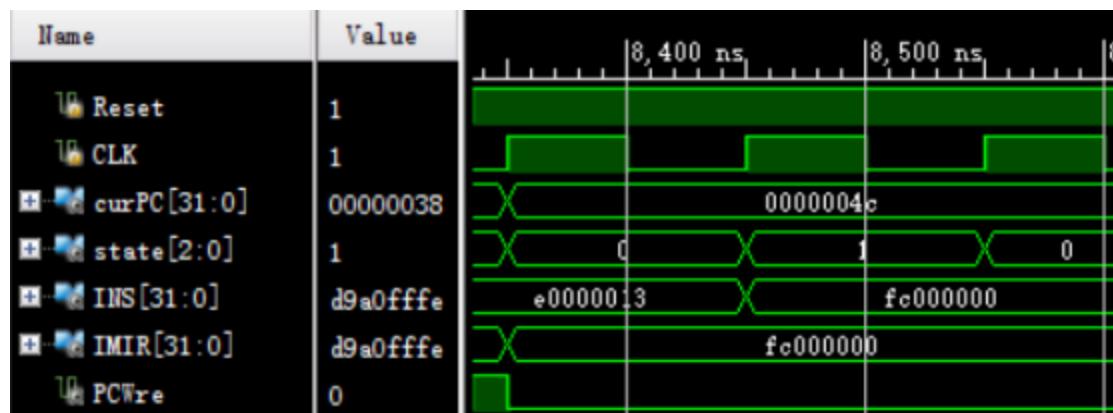


(2) 分析:

ID阶段: 译码得到nextPC = 4C.

24. 0x00000004C指令: halt

(1) 仿真结果:



(2) 分析:

PC值没有发生变化, 仿真结果正确。

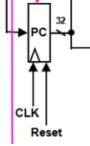
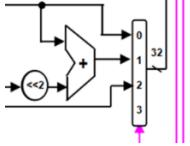
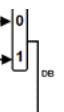
(三) 在Basys3板上运行所设计的CPU

烧板的底层模块一共有五部分。

时钟分频模块: 这个模块输入系统时钟, 根据分频输出两个不同频率的时钟信号, 分别供给按键模块的时钟频率, 同时也是数码管上数字刷新的周期, 和数码管显示的时钟频率。

按键模块: 每个按键周期检测实验板上是否有按钮输入, 如有, 则输出对应信号。这个信号会作为CPU的时钟输入(手动时钟)。

选择模块: 接收两个开关的输入, 将CPU中的八个变量传入, 根据开关选择对应的两个变量输出, 真值表如下:

Sw15	Sw14	AN3 AN2	AN1 AN0
0	0	当前 PC 值 	下条指令 PC 值 
0	1	RS 寄存器号 	RS 寄存器数据 Read Data1 [32]
1	0	RT 寄存器号 	RT 寄存器数据 Read Data2 [1]
1	1	ALU 输出结果 result 	DB 数据总线 DB 

数码管显示模块: 获取选择模块的输入, 将数字分配到 4 个 7 段位数码管上, 并根据其段位数码管显示器字形码转换。

顶层模块: 用 wire 将对应的变量相连接, 囊括五个底层模块 (CPU + 上述四个烧板过程新增的底层模块)。

实验结果:

1. 0x00000000 指令: addi \$1,\$0,8

IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0	WB AN3 AN2 AN1 AN0
Sw15	Sw14		
0	0	currentPC nextPC 	currentPC nextPC 
0	1	RS 号 RS 值 	RS 号 RS 值 
1	0	RT 号 RT 值 	RT 号 RT 值 
1	1	ALUresult 写入数据 	ALUresult 写入数据 

2. 0x00000014 指令: sll \$5,\$5,2

Sw15	Sw14	IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0	WB AN3 AN2 AN1 AN0
0	0	currentPC nextPC 0000	currentPC nextPC 0000	currentPC nextPC 0000	currentPC nextPC 0000
0	1	RS 号 RS 值 0000			
	0	RT 号 RT 值 0000	RT 号 RT 值 0000	RT 号 RT 值 0000	R 号 RT 值 0000
1	1	ALUresult 写入数据 0000	ALUresult 写入数据 0000	ALUresult 写入数据 0000	ALUresult 写入数据 0000

3. 0x00000018 指令: beq \$5,\$1,-2

Sw15	Sw14	IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0
0	0	currentPC nextPC 0000	currentPC nextPC 0000	currentPC nextPC 0000
0	1	RS 号 RS 值 0000	RS 号 RS 值 0000	RS 号 RS 值 0000
1	0	RT 号 RT 值 0000	RT 号 RT 值 0000	RT 号 RT 值 0000
1	1	ALUresult 写入数据 0000	ALUresult 写入数据 0000	ALUresult 写入数据 0000

4. 0x0000001C 指令: jal 0x00000040

IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0
Sw15	Sw14
currentPC nextPC 	currentPC nextPC
RS 号 RS 值 	RS 号 RS 值
RT 号 RT 值 	RT 号 RT 值
ALUresult 写入数据 	ALUresult 写入数据

5. **0x00000040 指令: sw \$2,4(\$1)**

IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0	WB AN3 AN2 AN1 AN0
Sw15	Sw14		
currentPC nextPC 	currentPC nextPC 	currentPC nextPC LD14 LD13 LD12 LD11 LD10	currentPC nextPC
RS 号 RS 值 	RS 号 RS 值 	RS 号 RS 值 	RS 号 RS 值
RT 号 RT 值 	RT 号 RT 值 	RT 号 RT 值 	R 号 RT 值
ALUresult 写入数据 	ALUresult 写入数据 	ALUresult 写入数据 	ALUresult 写入数据

6. **0x00000044 指令: lw \$12,4(\$1)**

IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0	MEM AN3 AN2 AN1 AN0	WB AN3 AN2 AN1 AN0
0 0 currentPC nextPC 	0 currentPC nextPC 	0 currentPC nextPC 	0 currentPC nextPC 	0 currentPC nextPC
0 1 RS 号 RS 值 	1 RS 号 RS 值 	1 RS 号 RS 值 	1 RS 号 RS 值 	1 RS 号 RS 值
1 0 RT 号 RT 值 	0 RT 号 RT 值 	0 RT 号 RT 值 	0 R 号 RT 值 	0 R 号 RT 值
1 1 ALUresult 写入数据 	0 ALUresult 写入数据 	0 ALUresult 写入数据 	0 ALUresult 写入数据 	0 ALUresult 写入数据

7. 0x00000048 指令: jr \$31

Sw15	Sw14	IF AN3 AN2 AN1 AN0	ID
0 0 currentPC nextPC 	0 currentPC nextPC 	0 currentPC nextPC 	0 currentPC nextPC
0 1 RS 号 RS 值 	1 RS 号 RS 值 	1 RS 号 RS 值 	1 RS 号 RS 值
1 0 RT 号 RT 值 	0 RT 号 RT 值 	0 RT 号 RT 值 	0 RT 号 RT 值
1 1 ALUresult 写入数据 	0 ALUresult 写入数据 	0 ALUresult 写入数据 	0 ALUresult 写入数据

8. 0x00000020 指令: slt \$8,\$12,\$1

IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0	WB AN3 AN2 AN1 AN0
Sw15	Sw14		
0	0	currentPC nextPC 	currentPC nextPC
0	1	RS 号 RS 值 	RS 号 RS 值
1	0	RT 号 RT 值 	RT 号 RT 值
1	1	ALUresult 写入数据 	ALUresult 写入数据

9. 0x00000030 指令: `sltiu $11,$10,0`

IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0	WB AN3 AN2 AN1 AN0
Sw15	Sw14		
0	0	currentPC nextPC 	currentPC nextPC
0	1	RS 号 RS 值 	RS 号 RS 值
1	0	RT 号 RT 值 	RT 号 RT 值
1	1	ALUresult 写入数据 	ALUresult 写入数据

10. 0x00000034 指令: `addi $13,$13,1`

Sw15	Sw14	IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0	WB AN3 AN2 AN1 AN0
0	0	currentPC nextPC 	currentPC nextPC 	currentPC nextPC 	currentPC nextPC
0	1	RS 号 RS 值 	RS 号 RS 值 	RS 号 RS 值 	RS 号 RS 值
1	0	RT 号 RT 值 	RT 号 RT 值 	RT 号 RT 值 	R 号 RT 值
1	1	ALUresult 写入数据 	ALUresult 写入数据 	ALUresult 写入数据 	ALUresult 写入数据

11. 0x00000038 指令: bltz \$13,-2 (<0, 34)

Sw15	Sw14	IF AN3 AN2 AN1 AN0	ID AN3 AN2 AN1 AN0	EXE AN3 AN2 AN1 AN0
0	0	currentPC nextPC 	currentPC nextPC 	currentPC nextPC
0	1	RS 号 RS 值 	RS 号 RS 值 	RS 号 RS 值
1	0	R 号 RT 值 	RT 号 RT 值 	RT 号 RT 值
1	1	ALUresult 写入数据 	ALUresult 写入数据 	ALUresult 写入数据

12. 0x0000003C 指令: j 0x0000004C

Sw15	Sw14	IF				ID			
		AN3	AN2	AN1	AN0	AN3	AN2	AN1	AN0
0	0	currentPC	nextPC			currentPC	nextPC		
0	1	RS 号	RS 值			RS 号	RS 值		
1	0	RT 号	RT 值			RT 号	RT 值		
1	1	ALUresult	写入数据			ALUresult	写入数据		

13. 0x0000004C 指令: halt

Sw15	Sw14	IF			
		AN3	AN2	AN1	AN0
0	0	currentPC	nextPC		
0	1	RS 号	RS 值		
1	0	RT 号	RT 值		
1	1	ALUresult	写入数据		

六. 实验心得

由于单周期已经做过了，所以多周期做起来相对简单很多，虽然最后的仿真结果和烧板过程和单周期大同小异，但在实现多周期的过程中最困难的是对于寄存器读写的设计，尤其是新增的四个Data Register，不过先划分为5个周期再想清楚先写后读或者先读后写的顺序，问题也就迎刃而解了。

由于理论课本没有专门介绍过多周期CPU，而是直接进入了流水CPU。虽然理论课老师讲过但依然觉得有点云里雾里，但考虑到流水CPU其实就是将多周期CPU加上了时间重叠的过程。所以每一步读什么、写什么，哪些信号序号传入寄存器，一方面这个实验老师给

出的数据通路图已经说明，另一方面参考流水CPU的五阶段四个流水寄存器都传入了什么数据，再除去流水CPU专有的会在下一周期被更新的那些（实际多周期CPU不会被覆盖），整体的数据通路基本就设计好了。

和单周期实验相同，我的实验步骤仍然是：1.代码实现各个模块；2.解决语法错误无法run起来的各种error（通常是用二分注释法定位错误的位置，因为vivado只会对比较明显的语法错误报错，有些逻辑错误不会报具体的行位置）；3.填写测试表，并将每一个阶段（周期）主要的数据值写出；4.根据仿真输出的结果与测试表对应；5.当遇到主要值输出与预期不相符，从CU开始debug，根据测试表确定每个周期CU输出各个信号的值，查找与输出结果不相符的信号，如果没有再通过错误数据的特性猜测错误位置，迅速定位，输出对应的值。

目前还有一个仍未解决的问题出现在最开始的初始化阶段，会发现reset变为高电位之后第一个状态已经变为了译码ID阶段。另外就是烧板的过程中IF阶段应该是0000，但由于烧板的过程中需要reset = 1的时候数码管才会亮，所以第一个阶段一定是0004.

总的来说，这次实验相对单周期CPU要简单一些，因为单周期的时候很多内容都没有讲过，但多周期的时候理论课的CPU设计部分已经讲完。而且做完这次实验让我对计组的了解从纸上谈兵变成了真枪实战，还是发现了一些自己之前理解错误的地方，果然应证了“实践是检验真理的唯一标准”这句话。