《计算机组成原理实验》

实验报告

单周期 CPU

（实验二）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 计算机类 3 班
学 生 姓 名 ：	王永锋
学　　号 ：	16337237
时　　间 ：	2017　年　11　月　20　日

目录

成 绩 ：

实 验 二 ： 单周期CPU设计与实现

一、 实验目的

(1) 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；
(2) 掌握单周期 CPU 的实现方法，代码实现方法；
(3) 认识和掌握指令与 CPU 的关系；
(4) 掌握测试单周期 CPU 的方法；
(5) 掌握单周期 CPU 的实现方法。

二、 实验内容

设计一个单周期 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** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

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

功能：rd←rs + rt。**reserved 为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate；immediate** 做“0”扩展再参加“或”运算。

（5）and rd , rs , rt

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

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（7）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

==>比较指令

（8） slt rd, rs, rt 带符号数

011100	rs(5 位)	rt(5 位)	rd(5 位)	reserved

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

==> 存储器读/写指令

（9）sw rt ,**immediate**(rs) 写存储器

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

功能：memory[rs+ (sign-extend)**immediate**]←rt；**immediate** 符号扩展再相加。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。

（10) lw rt , **immediate**(rs) 读存储器

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

功能：rt ← memory[rs + (sign-extend)**immediate**]；**immediate** 符号扩展再相加。

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

（11）beq　rs,rt,**immediate**

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

功能：if(rs=rt) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。**immediate 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 immediate 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”。

（12）bne　rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2　else pc ←pc + 4

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

（13）bgtz　rs,**immediate**

110010	rs(5 位)	00000	**immediate**

功能：if(rs>0) pc←pc + 4 + (sign-extend)**immediate** <<2　else pc ←pc + 4

==>跳转指令

（14）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],0,0}，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 位地址了，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（15）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三、 实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（**如果晶振的输**

出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。)

CPU 在处理指令时，一般需要经过以下几个步骤：

(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。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

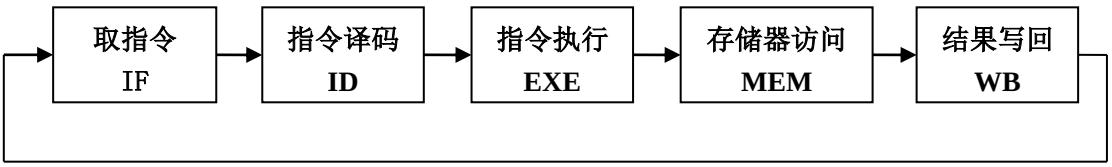
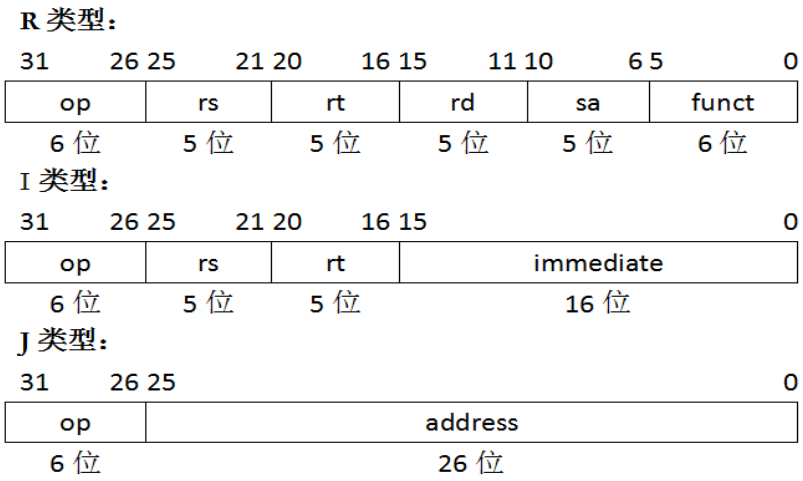


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：



其中，

op：为操作码；

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

rt：可读可写。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：只写。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与操作码配合使用；

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。

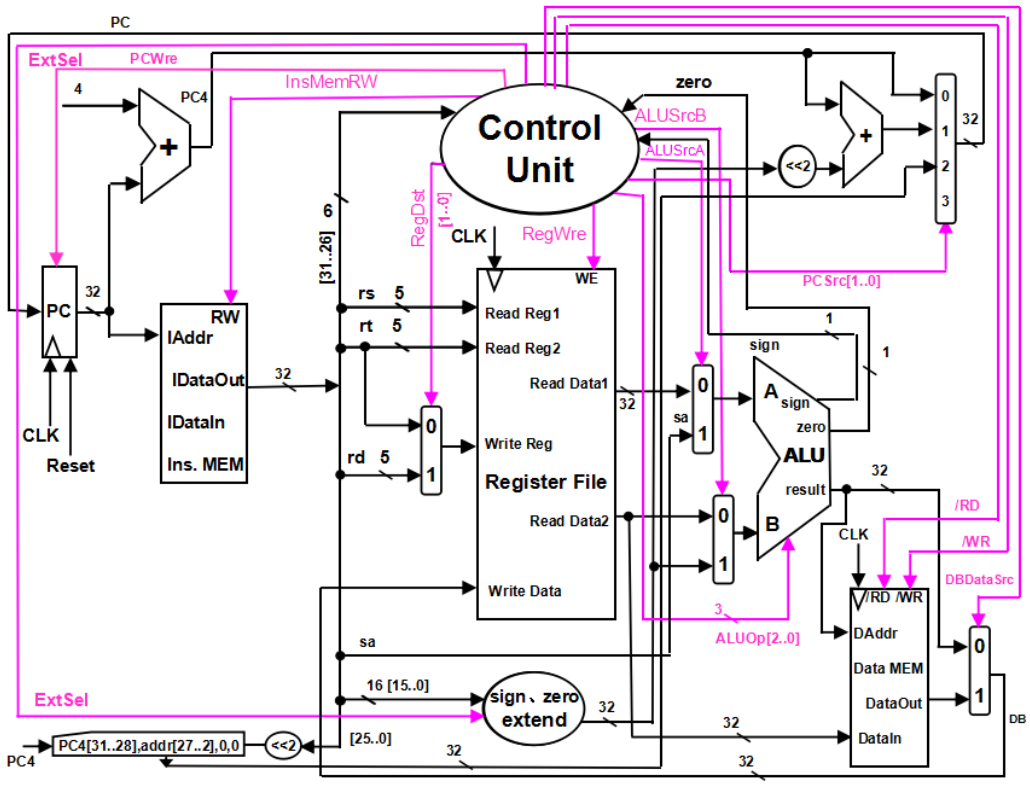


图 2　单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 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，先给出寄存器地址，读操作时，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。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1　控制信号的作用

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、bgtz、slt、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、slt、beq、bne、bgtz	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slt、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、bne、bgtz、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slt、sll、

		lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
nRD	读数据存储器，相关指令：lw	输出高阻态
nWR	写数据存储器，相关指令：sw	无操作
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、slt、sll
ExtSel	(zero-extend)**immediate(0 扩展)，**相关指令：ori	(sign-extend)**immediate (符号扩展)**，相关指令：addi、sw、lw、bne、bne、bgtz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slt、sll、sw、lw、beq(zero=0)、bne(zero=1)、bgtz(sign=1，或 zero=1); 01：pc<－pc+4+(sign-extend)**immediate**，相关指令：beq(zero=1)、bne(zero=0)、bgtz(sign=0，zero=0); 10：pc<－{(pc+4)[31..28],addr[27..2],0,0}，相关指令：j; 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，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

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，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据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端口，其地址来源 rt 或 rd 字段

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

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<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号

110	if (A<B &&(A[31] == B[31]))
	Y = 1;
	else if (A[31] && !B[31] Y = 1;
	else Y = 0;
		比较 A 与 B
		带符号

111	Y = A ⊕ B	异或

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表，再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。另外，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、 硬件设计实验过程

1. 设计数据通路图，确定所需实现子模块

单周期 CPU 的运行一般分为 5 个步骤：取指，译码，执行，访存，写回，更新 PC，这 5 个步骤在图 1-1 中用绿色的虚线分了出来。

图中，红色的为控制信号，用于控制部件的工作状况

黄色的为控制中心的控制信号，用以向控制单元输送当前工作信号。

黑色的为数据/地址信号，且线条粗的为 32 位数据。

从图中可知，我们在实现 CPU 的时候，主要需要实现的模块有六个：

各模块的功能如下表 1-1 所示

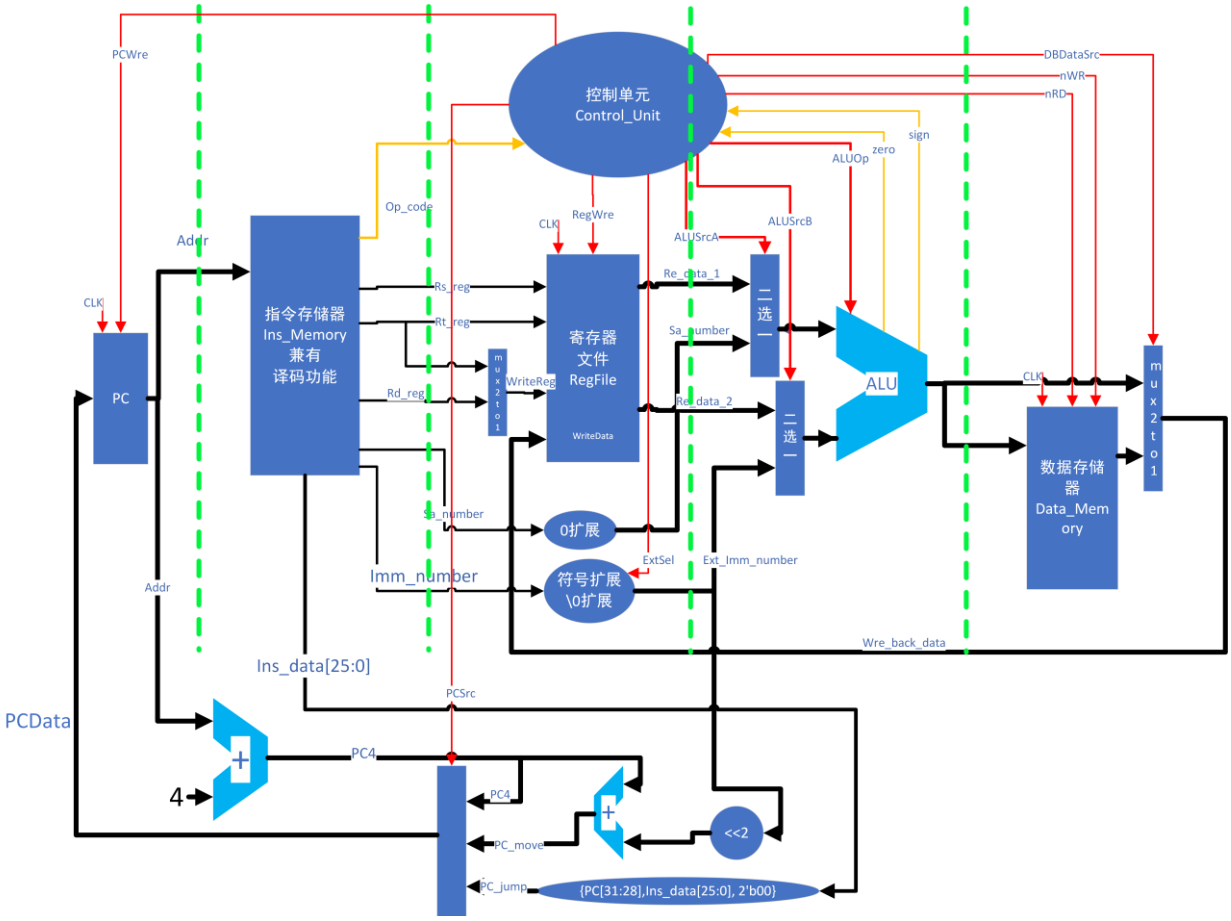


图 五-1 单周期 CPU 数据通路图

模块名	功能
PC 模块	PCWre = 0 时 不更改地址 【同步更改地址】 **时钟上升沿**且 PCWre = 1 时读取下一条地址
INS_MEMORY 模块	预先读取好指令 根据输入的 Addr，输出相应位置的指令 并将指令进行译码
REGFILE 模块	【异步读寄存器】 根据输入的寄存器号 ReadReg1，ReadReg2， 直接输出对应的数据

	【同步写寄存器】 在时钟下降沿且 RegWre = 1 时 将 Re_back_data 写回 WriteReg 对应的寄存器
ALU 模块	根据 ALUOp[2:0]执行对应的功能 具体的功能表见表 1-3
DATA_MEMORY 模块	【同步读存储器】 根据输入的地址，当 nRD = 0 时 直接输出对应的数据 【异步写存储器】 在时钟下降沿且 nWR = 0 时 向指定地址写入数据
CONTROL_UNIT 模块	根据输入的 6 位的 Op_code 以及 zero，sign 信号 输出对应的控制信号，具体控制信号见表 1-1

表 五-1 各模块功能详情

2. 确定控制单元信号表，编写控制单元模块

指令	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	nRD	nWR	ExtSel	PCSrc	RegDst	ALUOp
add	1	0	0	0	1	1	1	X	00	1	000
addi	1	0	1	0	1	1	1	1	00	0	000
sub	1	0	0	0	1	1	1	X	00	1	001
ori	1	0	1	0	1	1	1	0	00	0	011
and	1	1	0	0	1	1	1	X	00	1	100
or	1	0	0	0	1	1	1	X	00	1	011
sll	1	1	0	0	1	1	1	X	00	1	010
slt	1	0	0	0	1	1	1	X	00	1	110
sw	1	0	1	X	0	1	0	X	00	0	000
lw	1	0	1	1	1	0	1	X	00	0	000
beq	1	0	0	1	0	1	1	1	zero	0	111
bne	1	0	0	1	0	1	1	1	~zero	0	111
bgtz	1	0	0	1	0	1	1	1	~(sign|zero)	0	001
j	1	X	X	X	X	X	X	X	10	X	XXX
halt	0	X	X	X	X	X	X	X	XX	X	XXX

表 五-2 控制信号与指令的关系表

3. 设计 ALU 模块并确定功能

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号

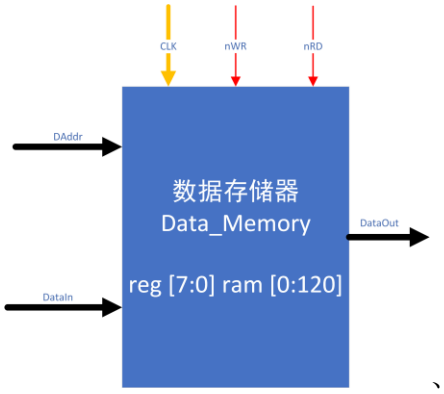
110	if (A<B &&(A[31] == B[31]))
	Y = 1;
	else if (A[31] && !B[31] Y = 1;
	else Y = 0;
		比较 A 与 B
		带符号

111	Y = A ⊕ B	异或

表 五-3 ALU 功能表

4. 设计 Data_Memory 模块

模块端口设计：



端口名称	相关说明
CLK	时钟输入 在 nWR 为 0 的前提下 当**时钟下降沿**到达时触发写入
NWR	写控制端口 低电平有效
NRD	读控制端口 低电平有效
DADDR[31:0]	需要读或写的内存的地址

DATAIN[31:0]	需要写入内存的数据
DATAOUT	从内存读取的数据

该模块的代码：

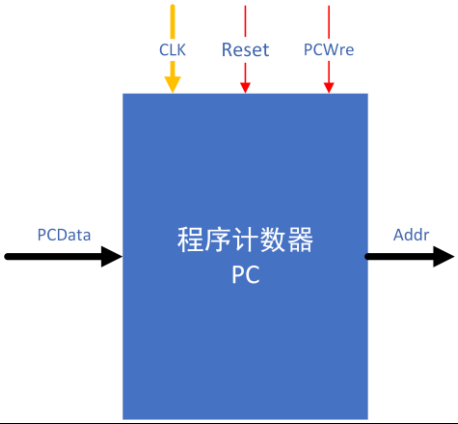
```
`timescale 1ns / 1ps
module Data_Memory(
    input CLK,
    input [31:0] DAddr,
    input [31:0] DataIn,// [31:24], [23:16], [15:8], [7:0]
    output [31:0] Dataout,
    input nRD, // 低电平有效，读控制信号
    input nWR // 西电平有效，写控制信号
);
    reg [7:0] ram [0:120];

    assign Dataout[7:0] = (nRD==0)?ram[DAddr + 3]:8'bz;
    assign Dataout[15:8] = (nRD==0)?ram[DAddr + 2]:8'bz;
    assign Dataout[23:16] = (nRD==0)?ram[DAddr + 1]:8'bz;
    assign Dataout[31:24] = (nRD==0)?ram[DAddr]:8'bz;

    always@( negedge CLK ) begin
        if( nWR==0 ) 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
endmodule
```

需要注意的几个地方：

1. 该 CPU 读写内存的方式为大端法。大端法，即低位数据放在高地址处，会影响数据存储器将 32 位的数据存储到以字节为单位的内存中，也同样会影响从内存中读取数据到寄存器。
2. 数据存储器为时钟下降沿时进行写入，这与 PC 模块在时钟上升沿时进行地址的转移是相互搭配的。在时钟上升沿的时候进行组合电路的运行，下降沿的时候进行数据的存储，这样能够确保数据读写与指令执行之间不会发生冲突。

5. PC 的设计

端口设计如图所示




```
`timescale 1ns / 1ps
// 注意时钟上升沿还是下降沿触发
module PC(
    input CLK,
    input Reset,// 0:初始化 PC 为 0  1:接受新地址
    input PCWre,// 0 不更改  1 更改
    input [31:0] PCData,
    output reg [31:0] Addr
);
    always@(posedge CLK or negedge Reset) begin
        if(Reset == 0)
            Addr = 0;
        else begin
            if (PCWre == 0)
                Addr = Addr;
            else
                Addr = PCData;
        end
    end
endmodule
```

需要说明的问题：

1. 在敏感表中，加入了关于 Reset 的检测，是出于这样的考虑：
   如果在敏感表中没有关于 Reset 下降沿的检测，那么在进行单步调试的时候，只要不按按钮，不论怎么调 Reset 都没有，因为无法触发语句执行。为了方便重置程序计数器 PC，就采用了检测 Reset 下降沿的做法，这样子，就可以在不按按钮的前提下，直接发送 Reset 信号得到重置。
2. PC 模块采用了时钟上升沿来进行 PC 的更新。
3. 当 PCWre = 0 时，出现了类似锁存器的硬件，这是因为需要配合 halt 指令。一旦运行 halt 指令，程序的地址就会一直维持在当前的地址，不进行改变。
4.

   其他的模块就不继续赘述，源代码可见 github 仓库

   https://github.com/WalkerYF/CPU_single_cycle

6. 编写 CPU_top 模块并使用代码测试

按照数据通路连接好 top 模块，并使用了在文件附件中《关于测试单周期 CPU 的简单方法》中给出的代码进行测试。该代码能够涵盖该 CPU 设计文档中所实现的所有指令。

关于详细的测试过程，请看第六节：仿真波形图解读

地址	*汇编程序*	*指令代码*				
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000000	addi $1,$0,8	000001	00000	00001	0000 0000 0000 1000	= 04010008
0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010	= 40020002
0x00000008	add $3,$2,$1	000000	00010	00001	00011 000 0000 0000	= 00411800
0x0000000C	sub $5,$3,$2	000010	00011	00010	00101 000 0000 0000	= 08622800
0x00000010	and $4,$5,$2	010001	00101	00010	00100 000 0000 0000	= 44A22000
0x00000014	or $8,$4,$2	010010	00100	00010	01000 000 0000 0000	= 48824000
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 00 0000	= 60084040
0x0000001C	**bne $8,$1,-2 (≠,转 18)**	110001	01000	00001	1111 1111 1111 1110	= C501FFFE
0x00000020	slt $6,$2,$1	011100	00010	00001	00110 000 0000 0000	= 70413000
0x00000024	slt $7,$6,$0	011100	00110	00000	00111 000 0000 0000	= 70C03800
0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000	= 04E70008
0x0000002C	**beq $7,$1,-2 (≠,转 28)**	110000	00111	00001	1111 1111 1111 1110	= C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100	= 98220004
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	= 9C290004
0x00000038	**bgtz $9,1 (>0,转 40)**	110010	01001	00000	0000 0000 0000 0001	= C9200001
0x0000003C	**halt**	111111	00000	00000	0000 0000 0000 0000	= FC000000
						=
0x00000040	**addi $9,$0,-1**	000001	00000	01001	1111 1111 1111 1111	= 0409FFFF
0x00000044	**j 0x00000038**	111000	00000	00000	0000 0000 0011 1000	= E000000E
0x00000048						=

表格 五-1 测试代码样例

六、 实验结果：仿真波形图解

注意：以下代码的运行是具有时间上的连贯性的，要注意寄存器的值

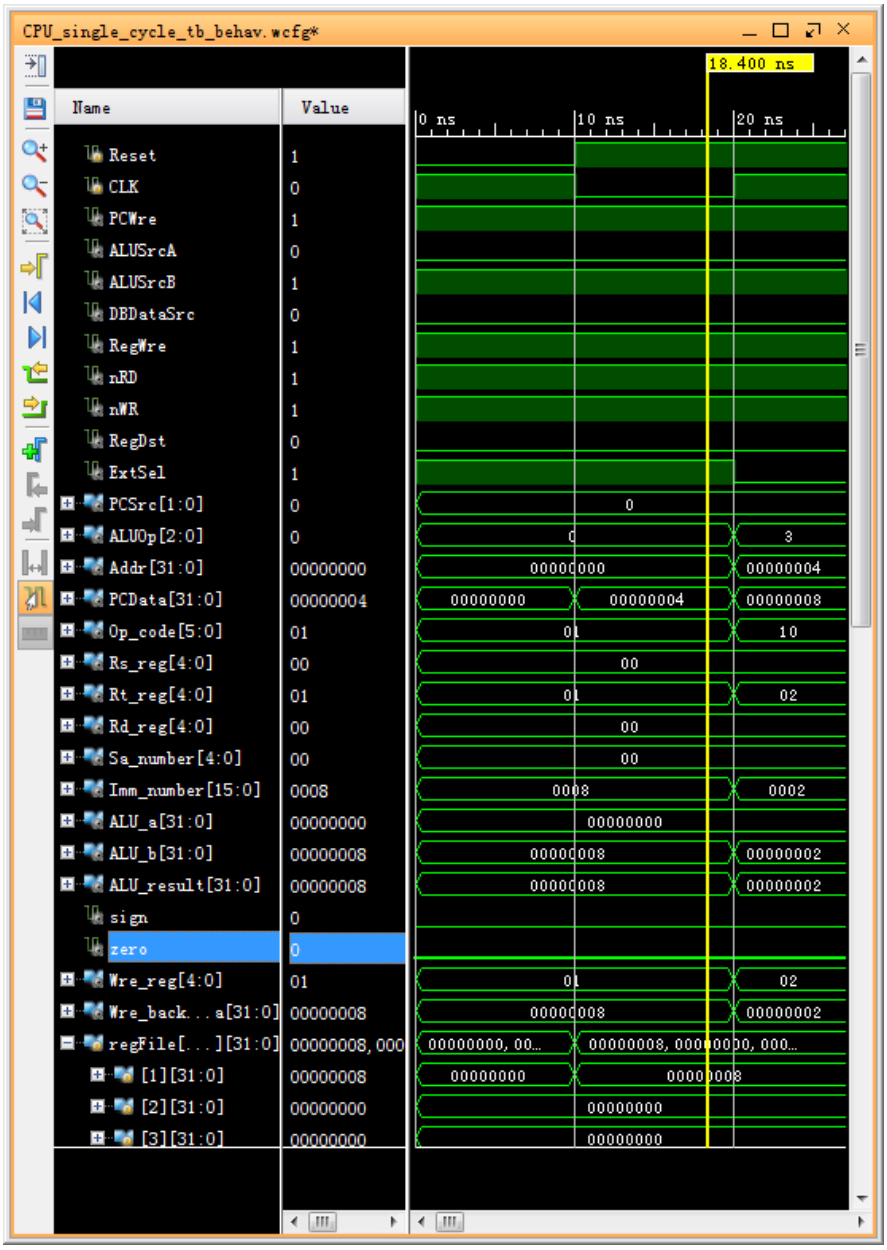
1. addi $1, $0, 8

部分重要信号：

当前步骤	**addi $1,$0,8**
取指	Addr = 0x00000000
译码	Addi $1, $0, 8 换成机器码为 *op（6）* *rs(5)* *rt(5)* *rd(5)/immediate (16)* *000001* 00000 00001 0000 0000 0000 1000 因此译码得： Op_code = 000001 Rs_reg = 0 Rt_reg = 1 Rd_reg = 0 Sa_number = 0 Imm_number = 0000 0000 0000 1000
执行	检查 ALU 输入： ALUOp = 0 （加运算） ALU_a = 0 ALU_b = 8 (二选一数据选择器选择了立即数) ALU_result = 8
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 1 Wre_back_data = 8 在时钟下降沿，数字 8 写入 1 号寄存器
更新 PC	下一条指令 PCData = 0x00000004

当前寄存器状态

$1	**8**
其余	0



相关控制信号如图所示，与前面表5-2一致

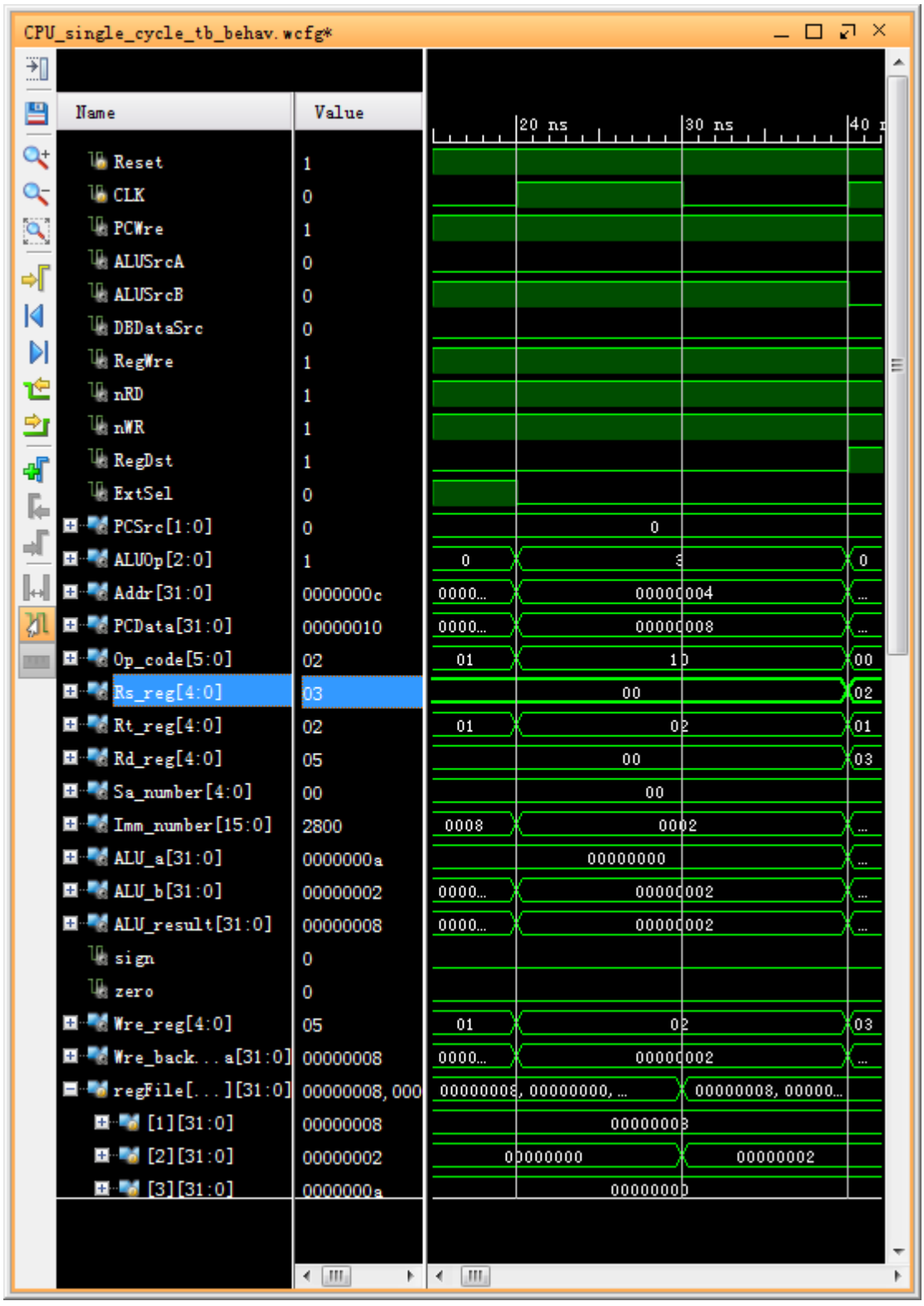
2. ori $2,$0,2

部分重要信号：

当前步骤	**ori $2,$0,2**
取指	Addr = 0x00000004
译码	**ori $2,$0,2** 换成机器码为 *op（6）* *rs(5)* *rt(5)* *rd(5)/immediate (16)* *010000* 00000 00010 0000 0000 0000 0010 因此译码得： Op_code = 000001 Rs_reg = 0 Rt_reg = 2 Rd_reg = 0 Sa_number = 0 Imm_number = 0000 0000 0000 0010
执行	检查 ALU 输入： ALUOp = 3，或运算 ALU_a = 0 ALU_b = 2 (二选一数据选择器选择了立即数) ALU_result = 2
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 2 Wre_back_data = 2 在时钟下降沿，数字 2 写入 2 号寄存器
更新 PC	下一条指令 PCData = 0x00000008

当前寄存器状态

$1	**8**
$2	2
其余	0


CPU_single_cycle_tb_behav.wcfg*
Name
Value
20 ns
30 ns
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 0
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 1
ExtSel 0
PCSrc[1:0] 0
ALUOp[2:0] 1
Addr[31:0] 0000000c
PCData[31:0] 00000010
Op_code[5:0] 02
Rs_reg[4:0] 03
Rt_reg[4:0] 02
Rd_reg[4:0] 05
Sa_number[4:0] 00
Imm_number[15:0] 2800
ALU_a[31:0] 0000000a
ALU_b[31:0] 00000002
ALU_result[31:0] 00000008
sign 0
zero 0
Wre_reg[4:0] 05
Wre_back...a[31:0] 00000008
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a

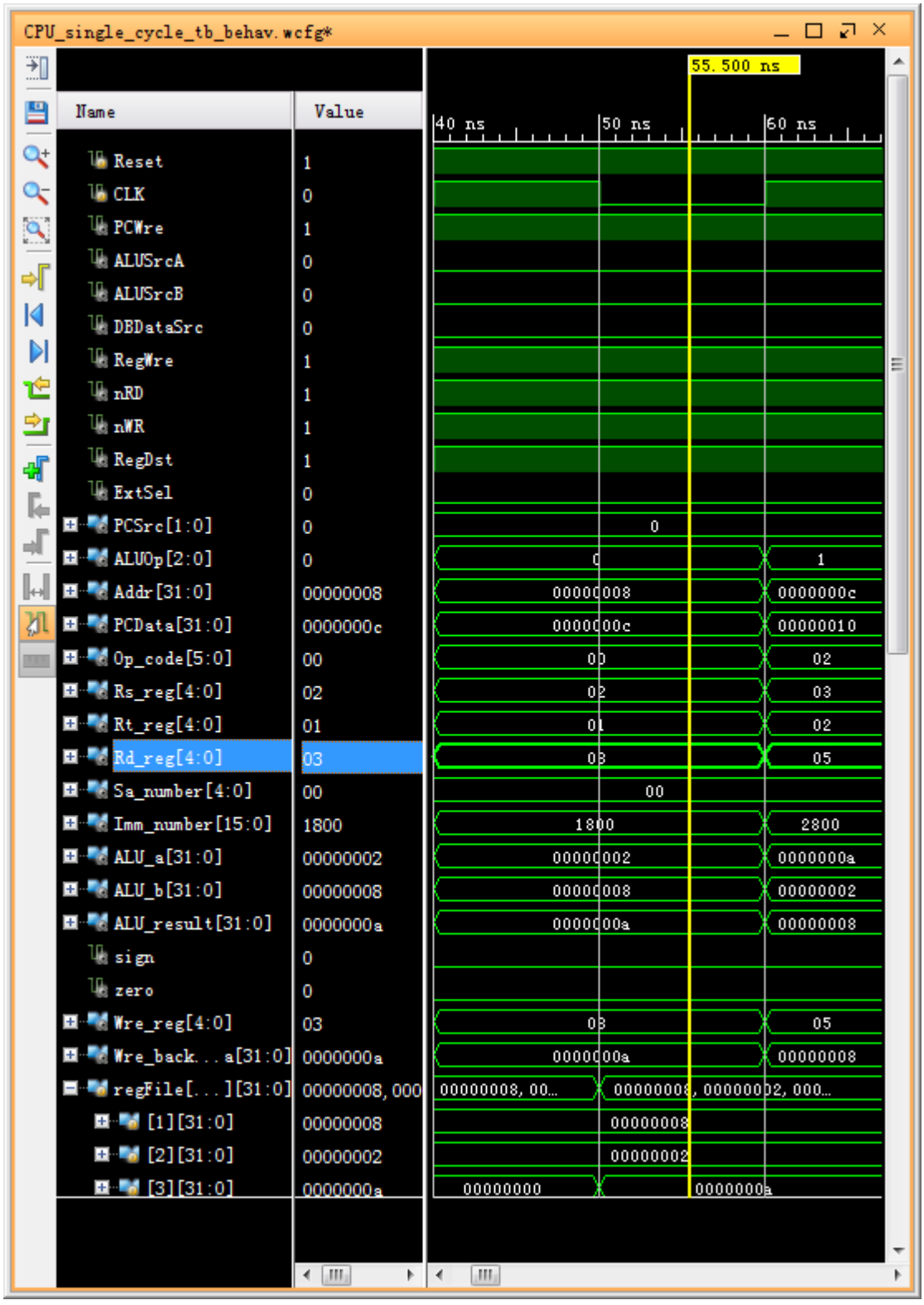
3. add $3,$2,$1

部分重要信号：

当前步骤	add $3,$2,$1
取指	Addr = 0x00000008
译码	add $3,$2,$1 换成机器码为 *op（6）* *rs(5)* *rt(5)* *rd(5)/immediate (16)* *000000* 00010 00001 00011 000 0000 0000 因此译码得： Op_code = 000000 Rs_reg = 2 Rt_reg = 1 Rd_reg = 3 Sa_number = 0 Imm_number = 0001 1000 0000 0010
执行	检查 ALU 输入： ALUOp = 0，加运算 ALU_a = 2 ALU_b = 8 ALU_result = 10 (0xA)
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 3 Wre_back_data = 0xA 在时钟下降沿，数字 0xA 写入 3 号寄存器
更新 PC	下一条指令 PCData = 0x0000000c

当前寄存器状态

$1	**8**
$2	2
$3	a
其余	0

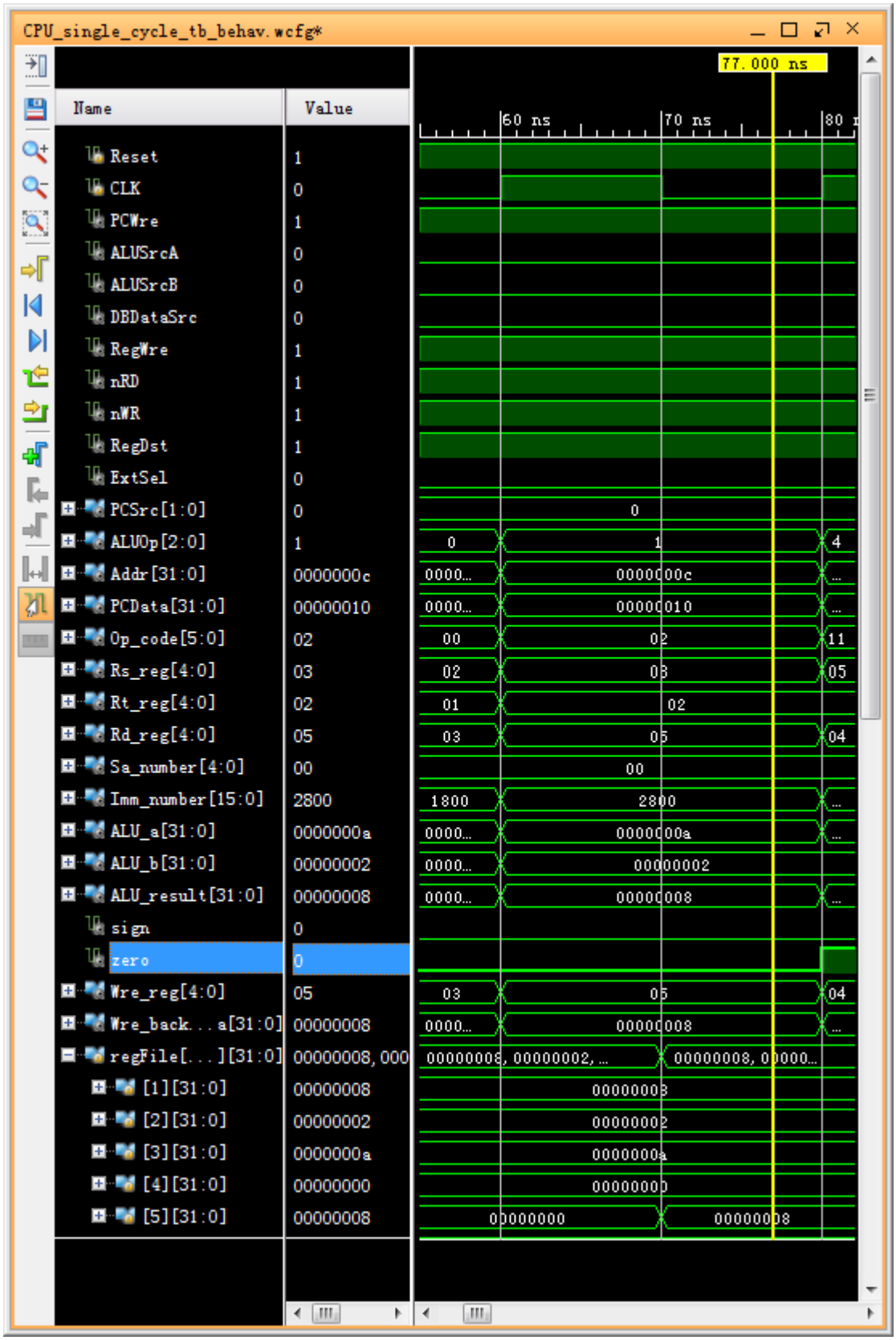

CPU_single_cycle_tb_behav.wcfg*
55.500 ns
40 ns
50 ns
60 ns
Name
Value
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 0
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 1
ExtSel 0
PCSrc[1:0] 0
ALUOp[2:0] 0
Addr[31:0] 00000008
PCData[31:0] 0000000c
Op_code[5:0] 00
Rs_reg[4:0] 02
Rt_reg[4:0] 01
Rd_reg[4:0] 03
Sa_number[4:0] 00
Imm_number[15:0] 1800
ALU_a[31:0] 00000002
ALU_b[31:0] 00000008
ALU_result[31:0] 0000000a
sign 0
zero 0
Wre_reg[4:0] 03
Wre_back...a[31:0] 0000000a
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a

4. sub $5,$3,$2

当前步骤	sub $5,$3,$2
取指	Addr = 0x0000000c
译码	sub $5,$3,$2 换成机器码为 *op（6）* *rs(5)* *rt(5)* *rd(5)/immediate (16)* *000010* 00011 00010 00101 000 0000 0000 因此译码得： Op_code = 000010 Rs_reg = 3 Rt_reg = 2 Rd_reg = 5 Sa_number = 0 Imm_number = 0010 1000 0000 0000
执行	检查 ALU 输入： ALUOp = 1，减法运算 ALU_a = a ALU_b = 2 ALU_result = 8
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 5 Wre_back_data = 8 在时钟下降沿，数字 8 写入 5 号寄存器
更新 PC	下一条指令 PCData = 0x000000010

当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
其余	0

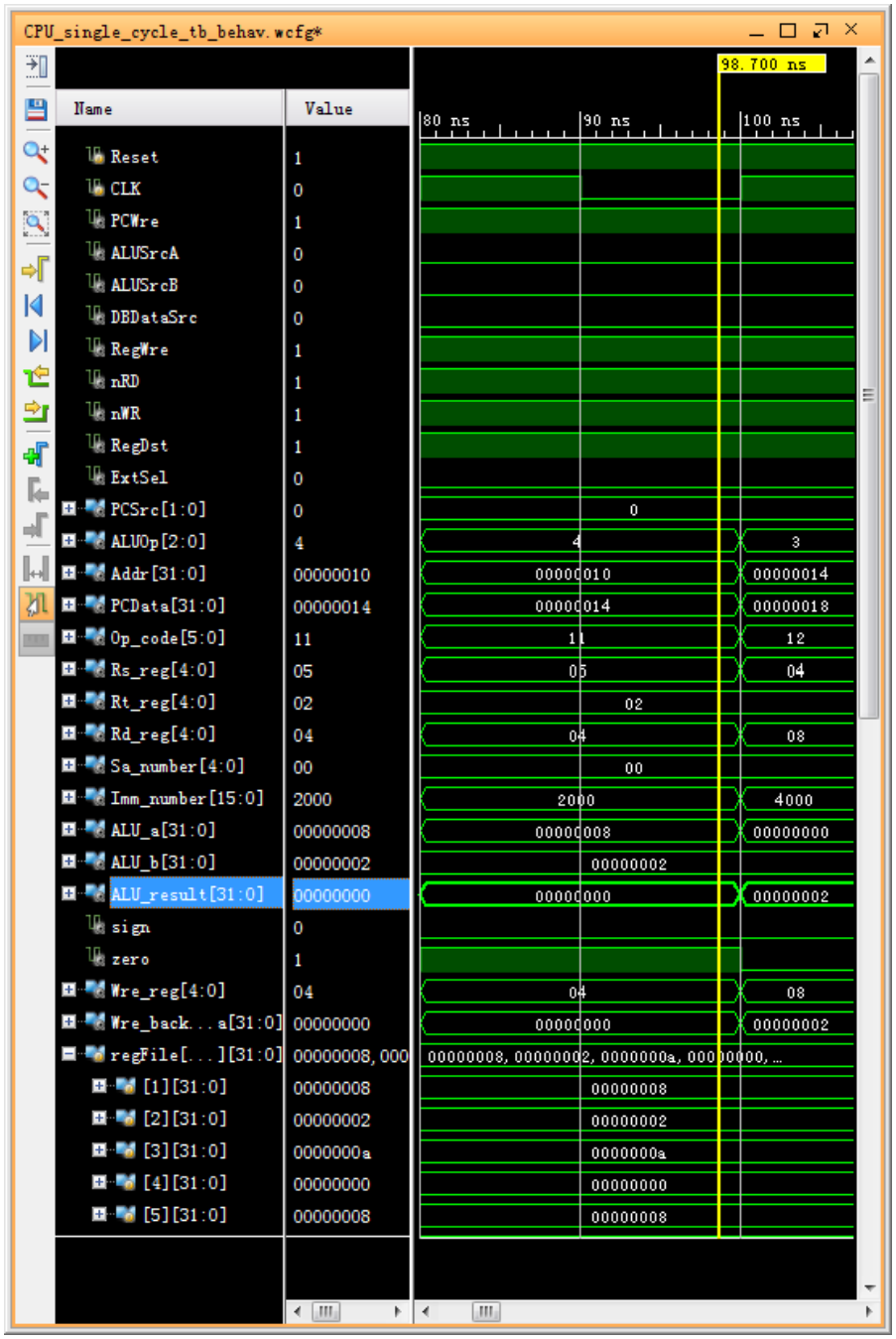

CPU_single_cycle_tb_behav.wcfg*
77.000 ns
60 ns
70 ns
Name
Value
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 0
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 1
ExtSel 0
PCSrc[1:0] 0
ALUOp[2:0] 1
Addr[31:0] 0000000c
PCData[31:0] 00000010
Op_code[5:0] 02
Rs_reg[4:0] 03
Rt_reg[4:0] 02
Rd_reg[4:0] 05
Sa_number[4:0] 00
Imm_number[15:0] 2800
ALU_a[31:0] 0000000a
ALU_b[31:0] 00000002
ALU_result[31:0] 00000008
sign 0
zero 0
Wre_reg[4:0] 05
Wre_back...a[31:0] 00000008
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
[4][31:0] 00000000
[5][31:0] 00000008

5. and $4,$5,$2

当前步骤	and $4,$5,$2
取指	Addr = 0x00000010
译码	and $4,$5,$2 换成机器码为 op（6） rs(5) rt(5) rd(5)/immediate (16) 010001 00101 00010 00100 000 0000 0000 因此译码得： Op_code = 010001 Rs_reg = 5 Rt_reg = 2 Rd_reg = 4 Sa_number = 0 Imm_number = 0010 0000 0000 0000
执行	检查 ALU 输入： ALUOp = 4，与运算 ALU_a = 8 ALU_b = 2 ALU_result = 0
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 4 Wre_back_data = 0 在时钟下降沿，数字 0 写入 4 号寄存器
更新 PC	下一条指令 PCData = 0x000000014

当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
其余	0

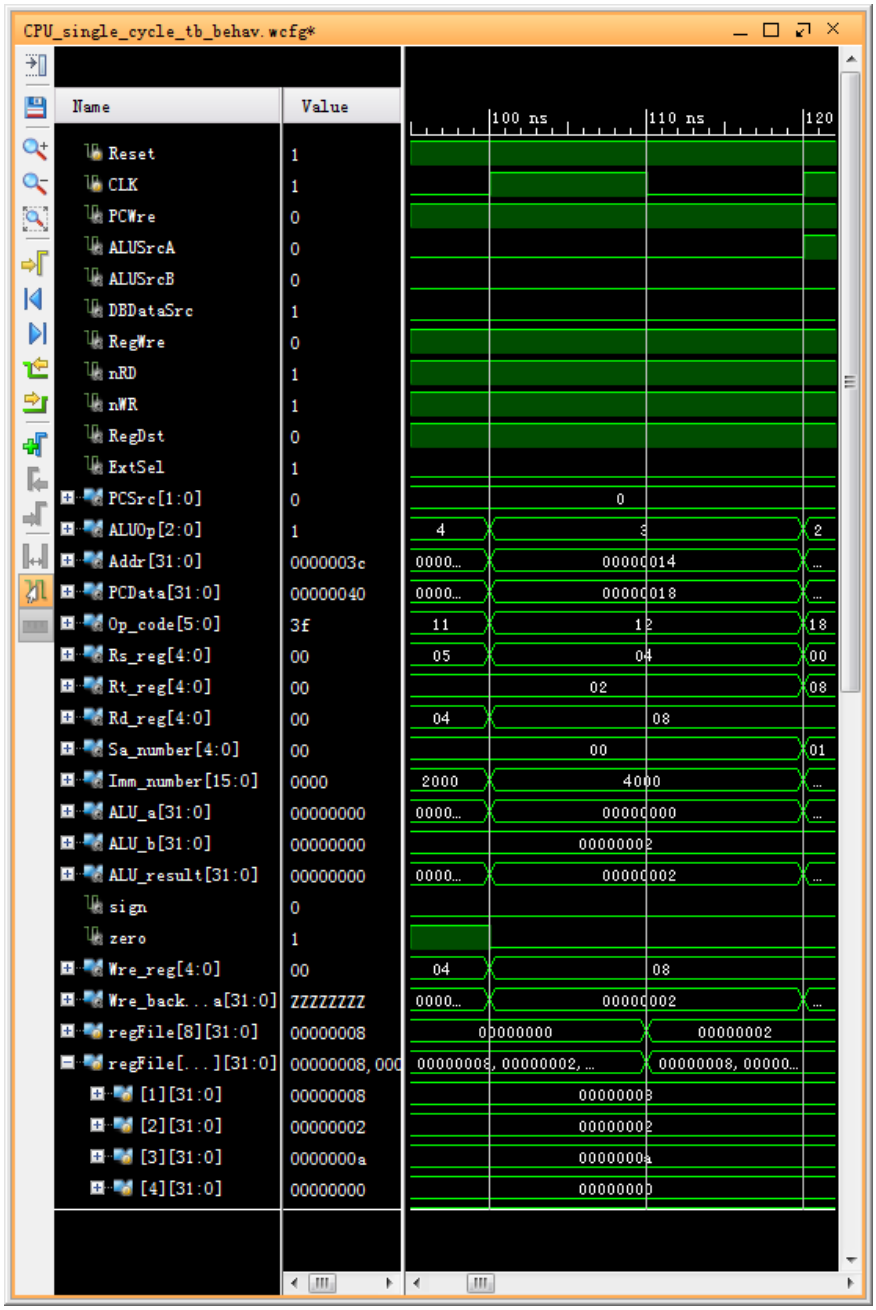

CPU_single_cycle_tb_behav.wcfg*
98.700 ns
80 ns
90 ns
100 ns
Name
Value
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 0
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 1
ExtSel 0
PCSrc[1:0] 0
ALUOp[2:0] 4
Addr[31:0] 00000010
PCData[31:0] 00000014
Op_code[5:0] 11
Rs_reg[4:0] 05
Rt_reg[4:0] 02
Rd_reg[4:0] 04
Sa_number[4:0] 00
Imm_number[15:0] 2000
ALU_a[31:0] 00000008
ALU_b[31:0] 00000002
ALU_result[31:0] 00000000
sign 0
zero 1
Wre_reg[4:0] 04
Wre_back...a[31:0] 00000000
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
[4][31:0] 00000000
[5][31:0] 00000008
00000008, 00000002, 0000000a, 00000000, ...

6. or $8,$4,$2

当前步骤	or $8,$4,$2
取指	Addr = 0x00000014
译码	or $8,$4,$2 换成机器码为 op（6） rs(5) rt(5) rd(5)/immediate (16) 010010 00100 00010 01000 000 0000 0000 因此译码得： Op_code = 010010 Rs_reg = 3 Rt_reg = 2 Rd_reg = 5 Sa_number = 0 Imm_number = 0010 1000 0000 0000
执行	检查 ALU 输入： ALUOp = 011，或运算 ALU_a = 0 ALU_b = 2 ALU_result = 2
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 8 Wre_back_data = 2 在时钟下降沿，数字 2 写入 8 号寄存器
更新 PC	下一条指令 PCData = 0x000000018

当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
$8	2
其余	0

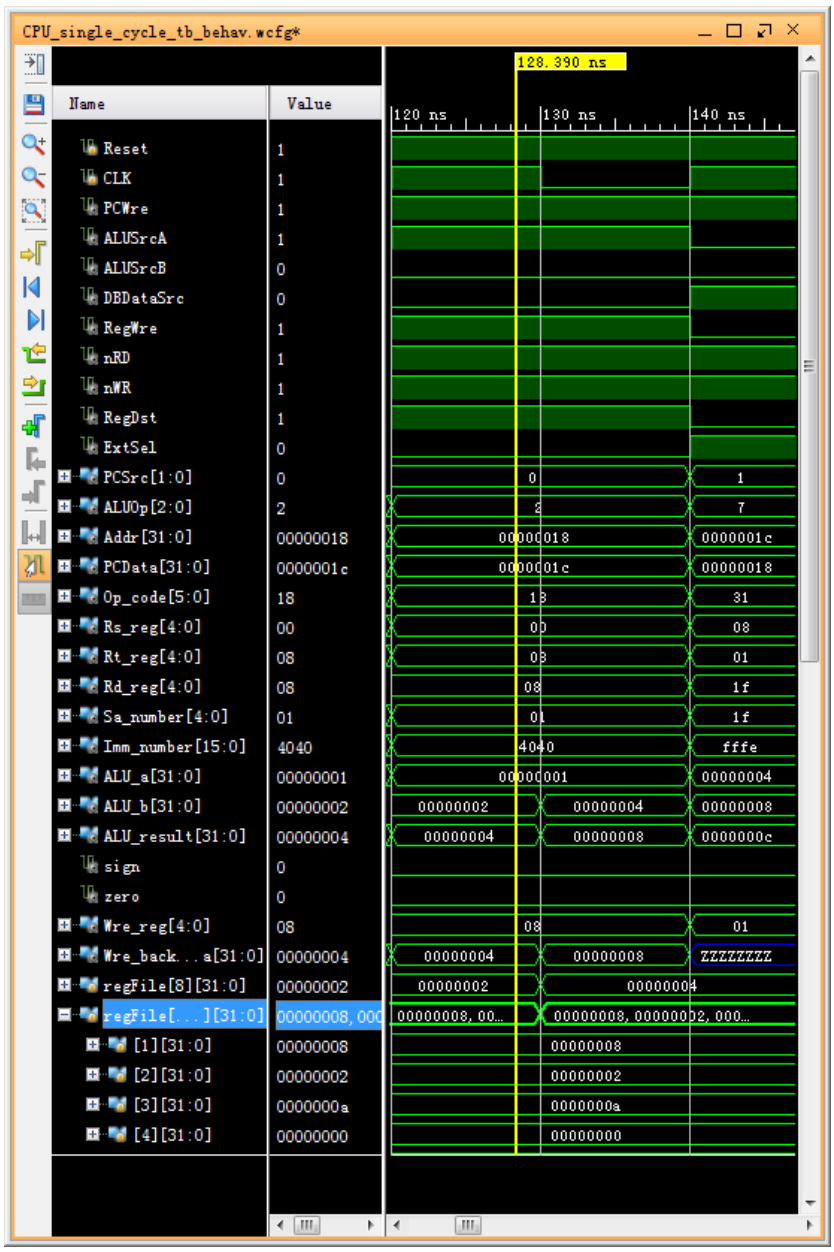

CPU_single_cycle_tb_behav.wcfg*
Name
Value
100 ns
110 ns
120
Reset 1
CLK 1
PCWre 0
ALUSrcA 0
ALUSrcB 0
DBDataSrc 1
RegWre 0
nRD 1
nWR 1
RegDst 0
ExtSel 1
PCSrc[1:0] 0
ALUOp[2:0] 1
Addr[31:0] 0000003c
PCData[31:0] 00000040
Op_code[5:0] 3f
Rs_reg[4:0] 00
Rt_reg[4:0] 00
Rd_reg[4:0] 00
Sa_number[4:0] 00
Imm_number[15:0] 0000
ALU_a[31:0] 00000000
ALU_b[31:0] 00000000
ALU_result[31:0] 00000000
sign 0
zero 1
Wre_reg[4:0] 00
Wre_back...a[31:0] zzzzzzzz
regFile[8][31:0] 00000008
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
[4][31:0] 00000000

7. sll $8,$8,1

当前步骤	sll $8,$8,1
取指	Addr = 0x00000018
译码	or $8,$4,$2 换成机器码为 op（6） rs(5) rt(5) rd(5)/immediate (16) 011000 00000 01000 01000 00001 00 0000 因此译码得： Op_code = 011000 Rs_reg = 0 Rt_reg = 3 Rd_reg = 8 Sa_number = 1 Imm_number = 01000 00001 00 0000
执行	检查 ALU 输入： ALUOp = 010，向左移位运算 ALU_a = 1 ALU_b = 2 ALU_result = 4
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 8 Wre_back_data = 4 在时钟下降沿，数字 4 写入 8 号寄存器
更新 PC	下一条指令 PCData = 0x00000001c

当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
$8	4
其余	0

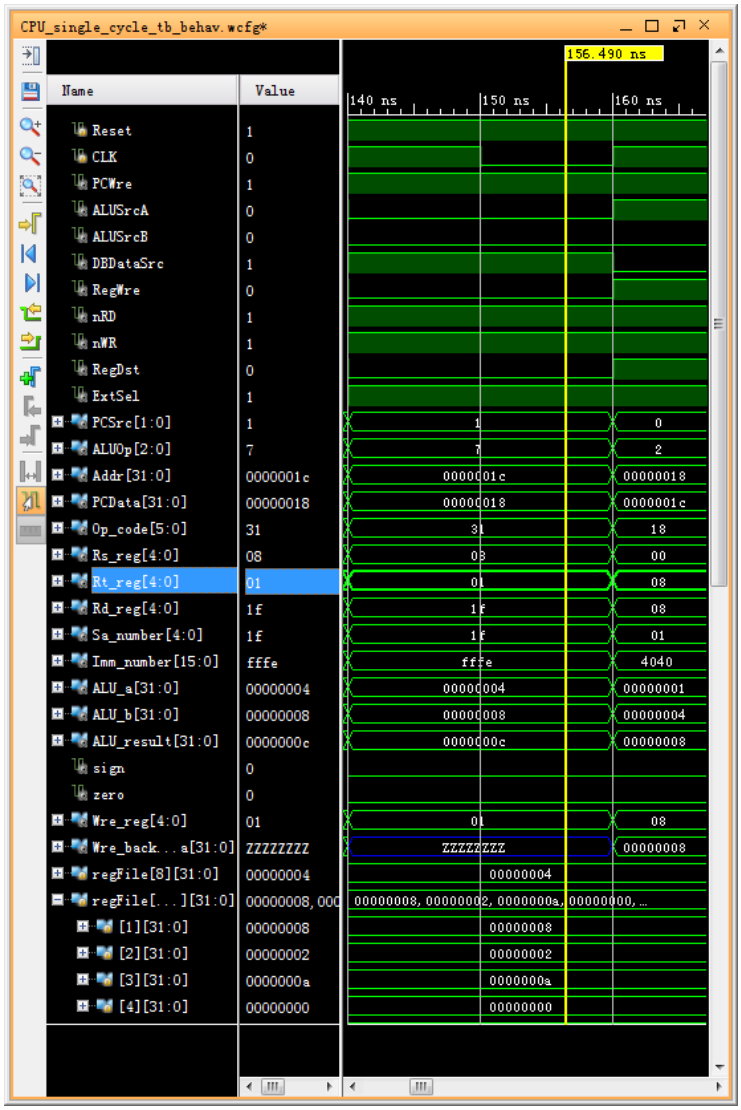

CPU_single_cycle_tb_behav.wcfg*
128.390 ns
120 ns
130 ns
140 ns
Name
Value
Reset 1
CLK 1
PCWre 1
ALUSrcA 1
ALUSrcB 0
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 1
ExtSel 0
PCSrc[1:0] 0
ALUOp[2:0] 2
Addr[31:0] 00000018
PCData[31:0] 0000001c
Op_code[5:0] 18
Rs_reg[4:0] 00
Rt_reg[4:0] 08
Rd_reg[4:0] 08
Sa_number[4:0] 01
Imm_number[15:0] 4040
ALU_a[31:0] 00000001
ALU_b[31:0] 00000002
ALU_result[31:0] 00000004
sign 0
zero 0
Wre_reg[4:0] 08
Wre_back...a[31:0] 00000004
regFile[8][31:0] 00000002
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
[4][31:0] 00000000

8. bne $8,$1,-2 (≠,转 18)

当前步骤	bne $8,$1,-2
取指	Addr = 0x0000001c
译码	bne $8,$1,-2 换成机器码为 op（6） rs(5) rt(5) rd(5)/immediate (16) 110001 01000 00001 1111 1111 1111 1110 因此译码得： Op_code = 110001 Rs_reg = 8 Rt_reg = 1 Rd_reg = 1f Sa_number = 1f Imm_number = 1111 1111 1111 1110
执行	检查 ALU 输入： ALUOp = 111，异或运算 ALU_a = 4 ALU_b = 8 ALU_result = c
访存	并不需要访问存储器，也不需要写存储器
写回	不写寄存器
更新 PC	下一条指令 PCData = 0x000000018 由于 zero 信号为 0，说明两个数不相等 执行跳转

当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
$8	4
其余	0



一些说明：

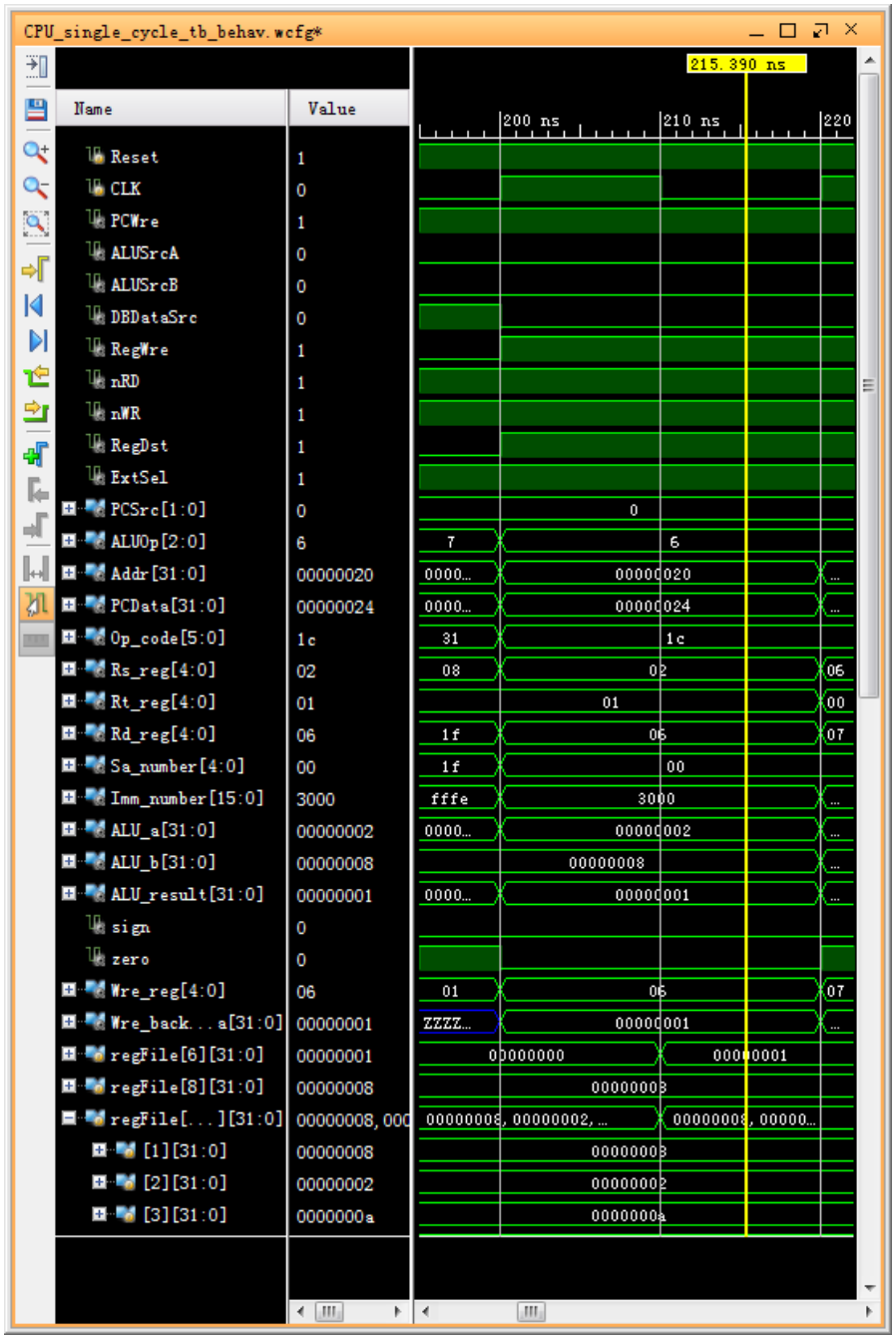
跳转回去后，再次执行移位指令，$8 寄存器中的值从 4 变为 8，与$1 相等，不执行跳转，遇到进行下一条语句。

9. slt $6,$2,$1

当前步骤	slt $6,$2,$1
取指	Addr = 0x00000020
译码	slt $6,$2,$1 换成机器码为 op（6） rs(5) rt(5) rd(5)/immediate (16) 011100 00010 00001 00110 000 0000 0000 因此译码得： Op_code = 011100 Rs_reg = 2 Rt_reg = 1 Rd_reg = 6 Sa_number = 0 Imm_number = 00110 000 0000 0000
执行	检查 ALU 输入： ALUOp = 010，比较大小运算就，若 a<b 则返回 1 ALU_a = 2 ALU_b = 8 ALU_result = 1
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 6 Wre_back_data = 1 在时钟下降沿，数字 6 写入 1 号寄存器
更新 PC	下一条指令 PCData = 0x000000024

当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
$6	1
$8	8
其余	0

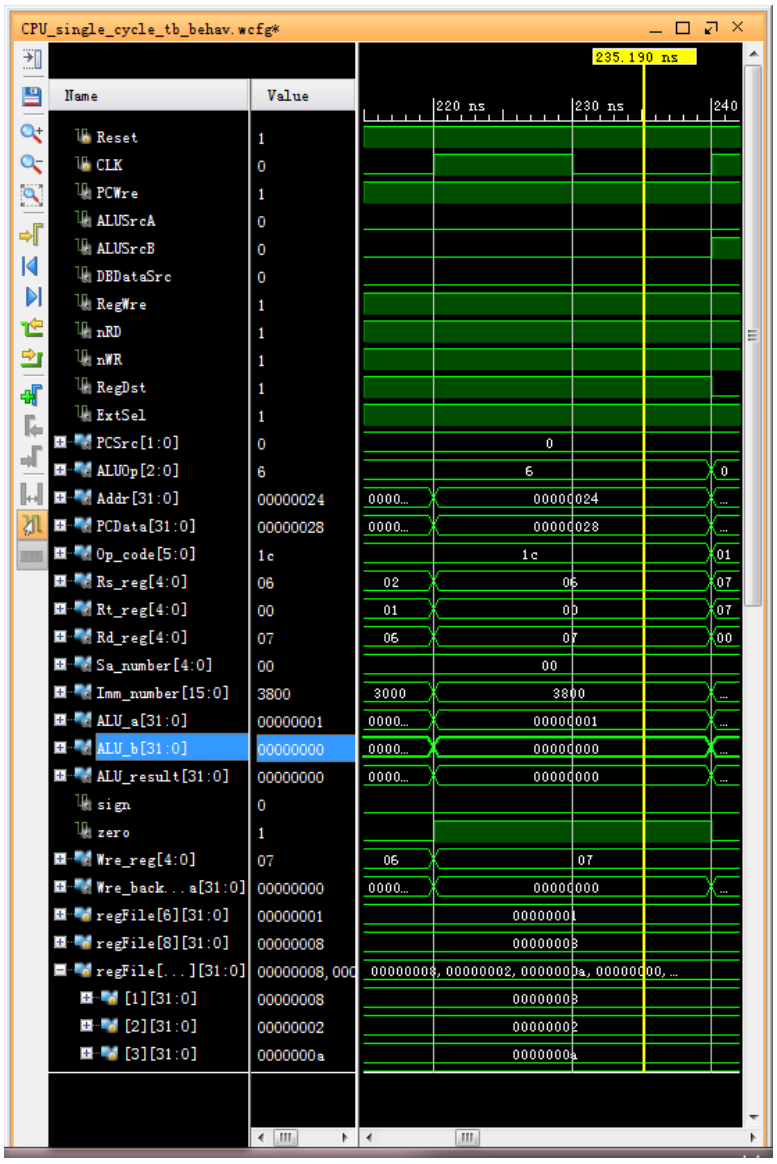

CPU_single_cycle_tb_behav.wcfg*
215.390 ns
200 ns
210 ns
220
Name
Value
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 0
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 1
ExtSel 1
PCSrc[1:0] 0
ALUOp[2:0] 6
Addr[31:0] 00000020
PCData[31:0] 00000024
Op_code[5:0] 1c
Rs_reg[4:0] 02
Rt_reg[4:0] 01
Rd_reg[4:0] 06
Sa_number[4:0] 00
Imm_number[15:0] 3000
ALU_a[31:0] 00000002
ALU_b[31:0] 00000008
ALU_result[31:0] 00000001
sign 0
zero 0
Wre_reg[4:0] 06
Wre_back...a[31:0] 00000001
regFile[6][31:0] 00000001
regFile[8][31:0] 00000008
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a

10.slt　$7,$6,$0

当前步骤	slt　$7,$6,$0
取指	Addr = 0x00000024
译码	slt　$7,$6,$0 换成机器码为 op　rs(5)　rt(5)　rd(5)/immediate (16) 011100　00110　00000　00111 000 0000 0000 因此译码得： Op_code = 011100 Rs_reg = 6 Rt_reg = 0 Rd_reg = 7 Sa_number = 0 Imm_number = 00111 000 0000 0000
执行	检查 ALU 输入 ALUOp = 110，带符号比较运算 ALU_a = 1 ALU_b = 0 ALU_result = 0
访存	并不需要访问存储器，也不需要写存储器
写回	Wre_Reg = 7 Wre_back_data = 0 在时钟下降沿，数字 0 写入 7 号寄存器
更新 PC	下一条指令 PCData = 0x000000028 由于 zero 信号=1，说明两个数不相等

当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
$6	1
$8	8
其余	0


CPU_single_cycle_tb_behav.wcfg*
235.190 ns
220 ns
230 ns
240
Name
Value
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 0
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 1
ExtSel 1
PCSrc[1:0] 0
ALUOp[2:0] 6
Addr[31:0] 00000024
PCData[31:0] 00000028
Op_code[5:0] 1c
Rs_reg[4:0] 06
Rt_reg[4:0] 00
Rd_reg[4:0] 07
Sa_number[4:0] 00
Imm_number[15:0] 3800
ALU_a[31:0] 00000001
ALU_b[31:0] 00000000
ALU_result[31:0] 00000000
sign 0
zero 1
Wre_reg[4:0] 07
Wre_back...a[31:0] 00000000
regFile[6][31:0] 00000001
regFile[8][31:0] 00000008
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a

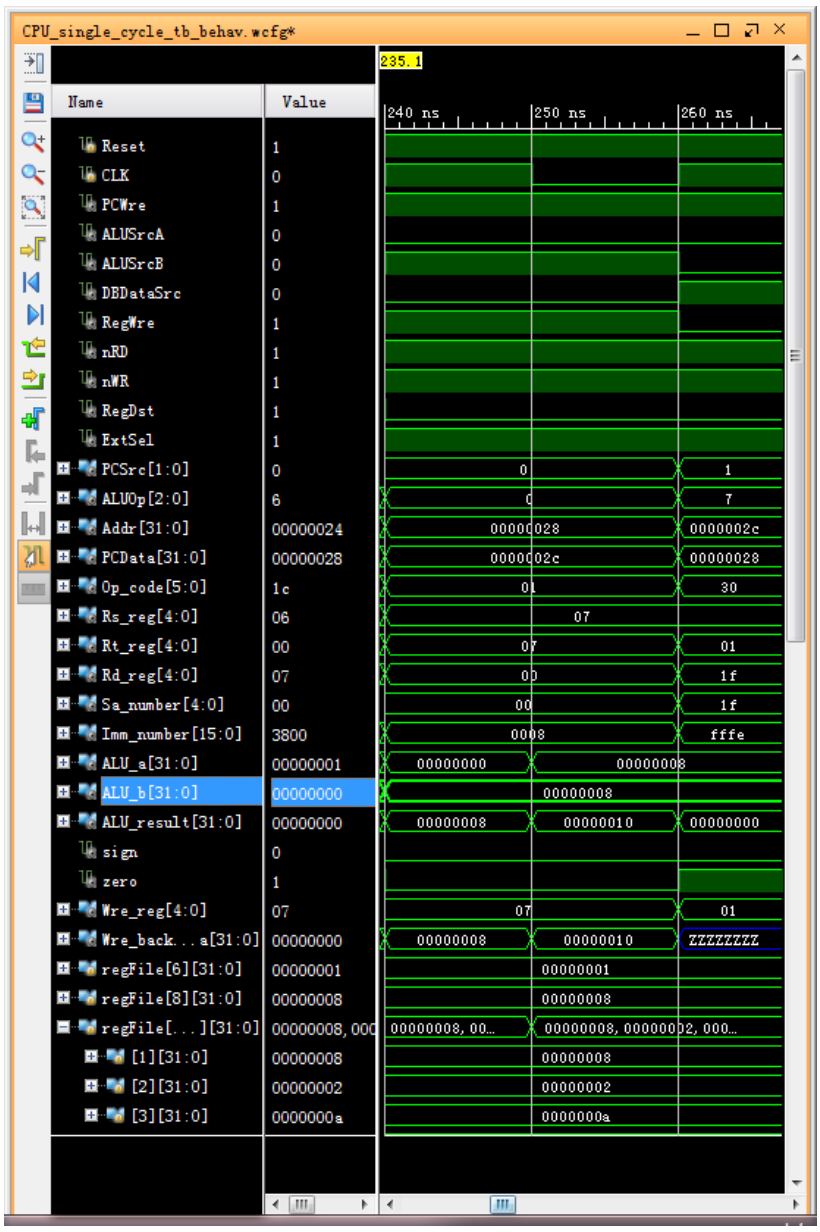
11.addi $7,$7,8

该指令已经描述过，此处不再赘述，仅给出当前状态。

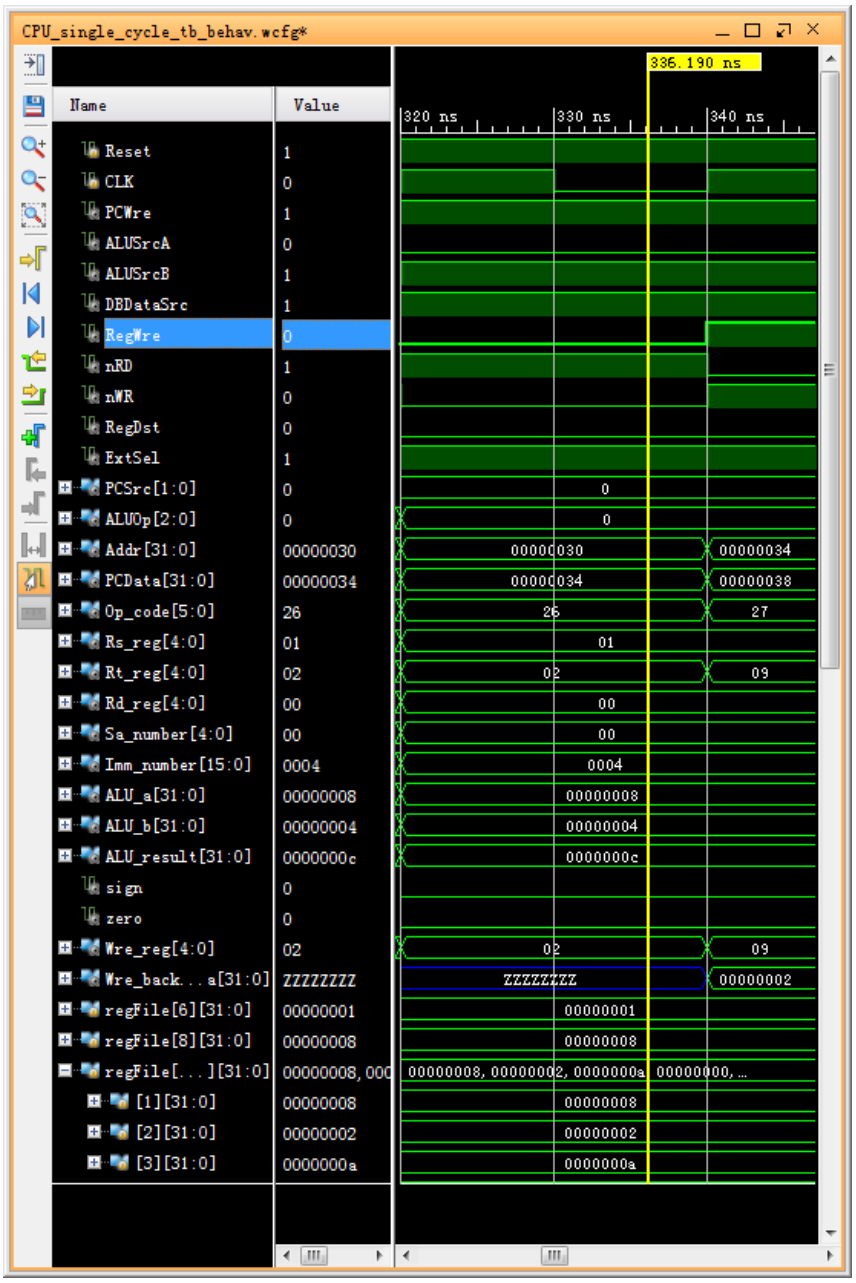
当前寄存器状态

$1	**8**
$2	2
$3	a
$5	8
$6	1
$7	8
$8	4
其余	0

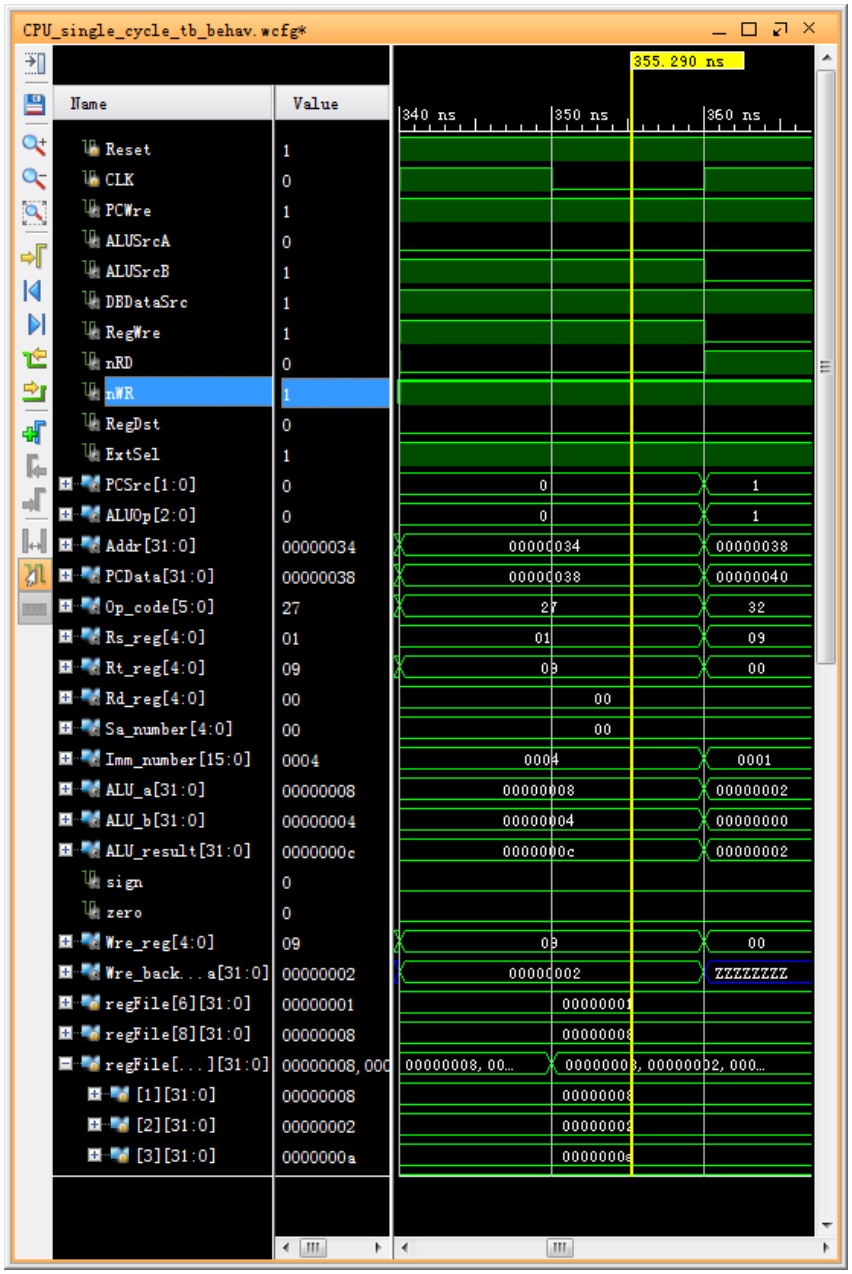
12.beq $7,$1,-2 (≠,转 28)



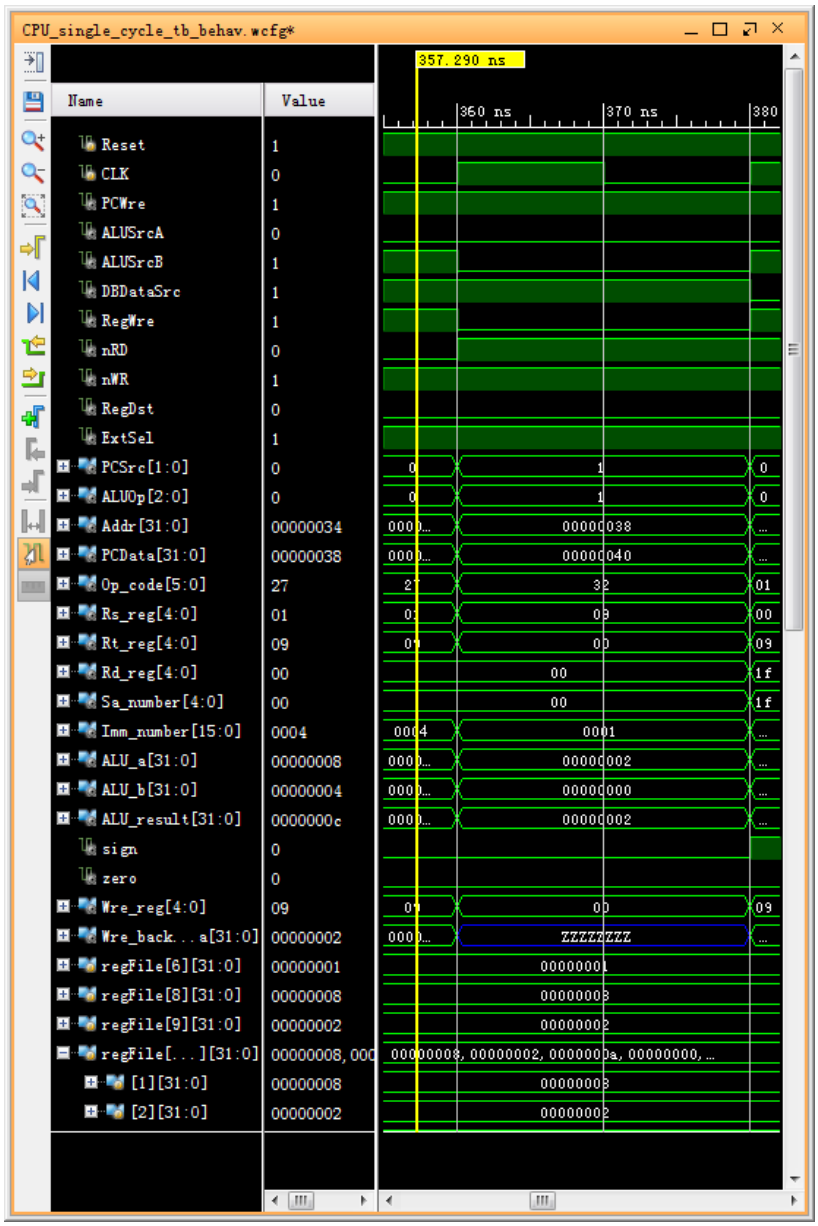
13.sw　$2,4($1)


CPU_single_cycle_tb_behav.wcfg*
Name
Value
336.190 ns
320 ns
330 ns
340 ns
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 1
DBDataSrc 1
RegWre 0
nRD 1
nWR 0
RegDst 0
ExtSel 1
PCSrc[1:0] 0
ALUOp[2:0] 0
Addr[31:0] 00000030
PCData[31:0] 00000034
Op_code[5:0] 26
Rs_reg[4:0] 01
Rt_reg[4:0] 02
Rd_reg[4:0] 00
Sa_number[4:0] 00
Imm_number[15:0] 0004
ALU_a[31:0] 00000008
ALU_b[31:0] 00000004
ALU_result[31:0] 0000000c
sign 0
zero 0
Wre_reg[4:0] 02
Wre_back...a[31:0] ZZZZZZZZ
regFile[6][31:0] 00000001
regFile[8][31:0] 00000008
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
00000030
00000034
26
27
01
02
09
00
0004
00000008
00000004
0000000c
ZZZZZZZZ
00000002
00000001
00000008, 00000002, 0000000a
00000000, ...

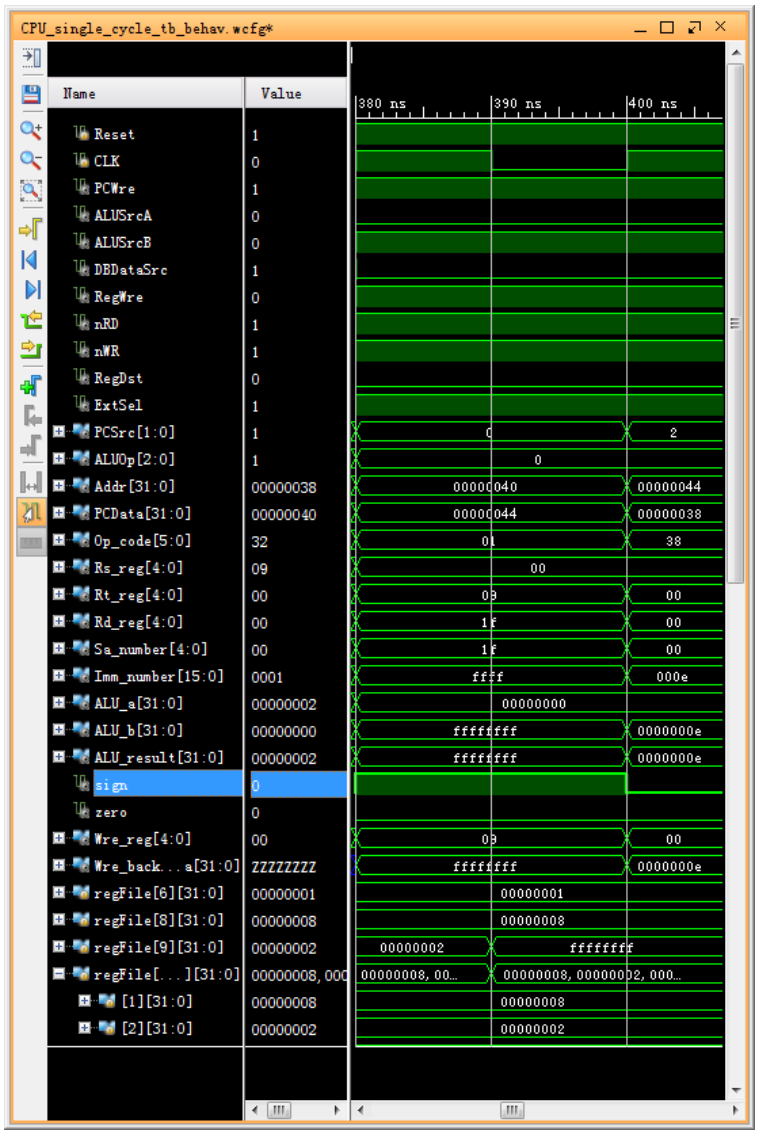
14.lw $9,4($1)


CPU_single_cycle_tb_behav.wcfg*
355.290 ns
340 ns
350 ns
360 ns
Name
Value
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 1
DBDataSrc 1
RegWre 1
nRD 0
nWR 1
RegDst 0
ExtSel 1
PCSrc[1:0] 0
ALUOp[2:0] 0
Addr[31:0] 00000034
PCData[31:0] 00000038
Op_code[5:0] 27
Rs_reg[4:0] 01
Rt_reg[4:0] 09
Rd_reg[4:0] 00
Sa_number[4:0] 00
Imm_number[15:0] 0004
ALU_a[31:0] 00000008
ALU_b[31:0] 00000004
ALU_result[31:0] 0000000c
sign 0
zero 0
Wre_reg[4:0] 09
Wre_back...a[31:0] 00000002
regFile[6][31:0] 00000001
regFile[8][31:0] 00000008
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a

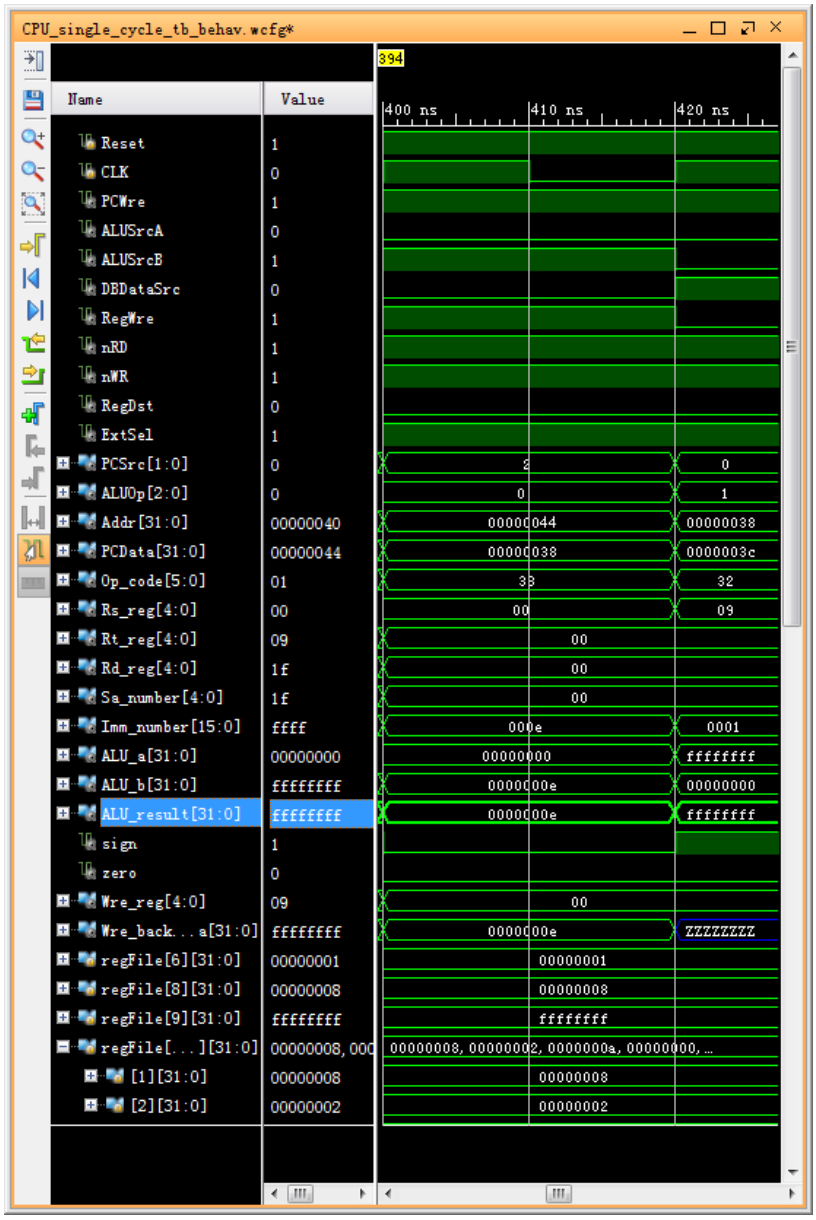
15.bgtz　$9,1 (>0,转 40)



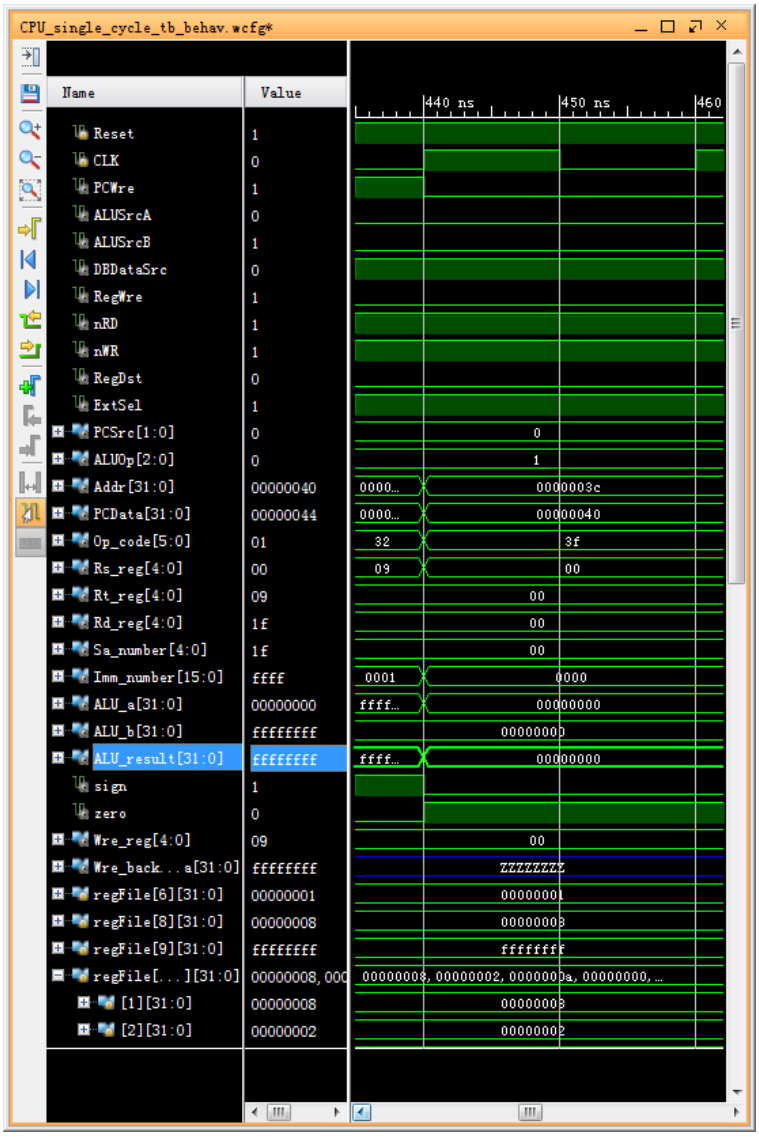
16.addi　$9,$0,-1


CPU_single_cycle_tb_behav.wcfg*
Name
Value
380 ns
390 ns
400 ns
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 0
DBDataSrc 1
RegWre 0
nRD 1
nWR 1
RegDst 0
ExtSel 1
PCSrc[1:0] 1
ALUOp[2:0] 1
Addr[31:0] 00000038
PCData[31:0] 00000040
Op_code[5:0] 32
Rs_reg[4:0] 09
Rt_reg[4:0] 00
Rd_reg[4:0] 00
Sa_number[4:0] 00
Imm_number[15:0] 0001
ALU_a[31:0] 00000002
ALU_b[31:0] 00000000
ALU_result[31:0] 00000002
sign 0
zero 0
Wre_reg[4:0] 00
Wre_back...a[31:0] ZZZZZZZZ
regFile[6][31:0] 00000001
regFile[8][31:0] 00000008
regFile[9][31:0] 00000002
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002

17.j　0x00000038



18.halt


CPU_single_cycle_tb_behav.wcfg*
Name
Value
440 ns
450 ns
460
Reset 1
CLK 0
PCWre 1
ALUSrcA 0
ALUSrcB 1
DBDataSrc 0
RegWre 1
nRD 1
nWR 1
RegDst 0
ExtSel 1
PCSrc[1:0] 0
ALUOp[2:0] 0
Addr[31:0] 00000040
PCData[31:0] 00000044
Op_code[5:0] 01
Rs_reg[4:0] 00
Rt_reg[4:0] 09
Rd_reg[4:0] 1f
Sa_number[4:0] 1f
Imm_number[15:0] ffff
ALU_a[31:0] 00000000
ALU_b[31:0] ffffffff
ALU_result[31:0] ffffffff
sign 1
zero 0
Wre_reg[4:0] 09
Wre_back...a[31:0] ffffffff
regFile[6][31:0] 00000001
regFile[8][31:0] 00000008
regFile[9][31:0] ffffffff
regFile[...][31:0] 00000008, 000
[1][31:0] 00000008
[2][31:0] 00000002
00000008, 00000002, 0000000a, 00000000, ...

七、 在 Basy3 实验板上显示

1. 按键消抖模块

按键消抖这里采用了延时的操作
通过延时 20ms，得到稳定的按键的状态。
按键消抖模块的代码如下：

```
module debouncing (
    clk,
    rst,// 低电平有效
    key_n,
    key_pulse
);
// 只会检测下降沿，然后下降沿一到达就释放一个时钟周期的高电平

    parameter       N  =  1;                    //要消除的按键的数

    input            clk;
    input            rst;                      //低电平有效
    input   [N-1:0]   key_n;                     //输入的按键
    output  [N-1:0]   key_pulse;                 //按键动作产生的脉冲
    wire key;
    wire key_pulse_n;
    assign key = ~key_n;
        reg     [N-1:0]   key_rst_pre;
//定义一个寄存器型变量存储上一个触发时的按键值
        reg     [N-1:0]   key_rst;
//定义一个寄存器变量储存储当前时刻触发的按键值

        wire    [N-1:0]   key_edge;
//检测到按键由高到低变化是产生一个高脉冲

        //利用非阻塞赋值特点，将两个时钟触发时按键状态存储在两个寄存器变量中
        always @(posedge clk  or  negedge rst)
          begin
             if (!rst) begin
                 key_rst <= {N{1'b1}};
//初始化时给 key_rst 赋值全为 1，{}中表示 N 个 1
                 key_rst_pre <= {N{1'b1}};
             end
```

```
            else begin
                key_rst <= key;
//第一个时钟上升沿触发之后 key 的值赋给 key_rst,同时 key_rst 的值赋给
key_rst_pre
                key_rst_pre <= key_rst;
//非阻塞赋值。相当于经过两个时钟触发，key_rst 存储的是当前时刻 key 的值，
key_rst_pre 存储的是前一个时钟的 key 的值
            end
          end

       assign  key_edge = key_rst_pre & (~key_rst);
//脉冲边沿检测。当 key 检测到下降沿时，key_edge 产生一个时钟周期的高电平

       reg [21:0]    cnt;
//产生延时所用的计数器，系统时钟 100MHz，要延时 20ms 左右时间，至少需要 21 位计数
器

       //产生 20ms 延时，当检测到 key_edge 有效是计数器清零开始计数
       always @(posedge clk or negedge rst)
          begin
            if(!rst)
               cnt <= 21'h0;
            else if(key_edge)
               cnt <= 21'h0;
            else
               cnt <= cnt + 1'h1;
            end

       reg     [N-1:0]   key_sec_pre;
//延时后检测电平寄存器变量
       reg     [N-1:0]   key_sec;


       //延时后检测 key，如果按键状态变低产生一个时钟的高脉冲。如果按键状态是高
的话说明按键无效
       always @(posedge clk  or  negedge rst)
         begin
            if (!rst)
                key_sec <= {N{1'b1}};
            else if (cnt==18'h3ffff)
                key_sec <= key;
         end
      always @(posedge clk  or  negedge rst)
         begin
```

```
            if (!rst)
                key_sec_pre <= {N{1'b1}};
            else
                key_sec_pre <= key_sec;
        end
    assign  key_pulse_n = key_sec_pre & (~key_sec);
    assign  key_pulse = ~key_pulse_n;

endmodule
```

2. 编写 top 模块，实例化 CPU 并显示信号

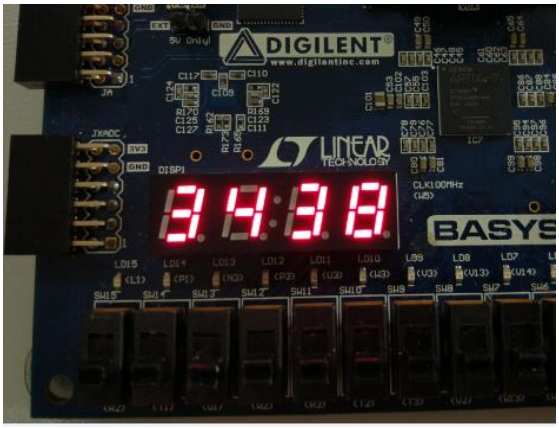
八、 实验结果：basys3 板上运行 CPU

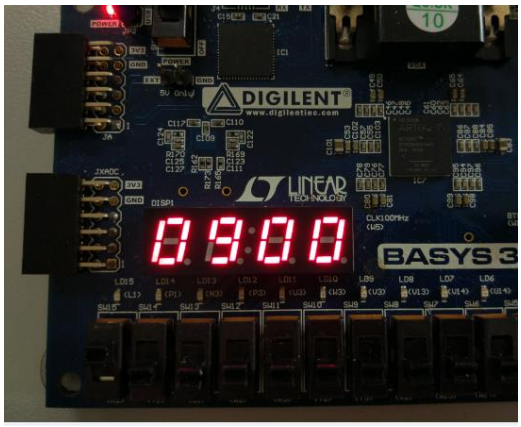
以这三条指令为例

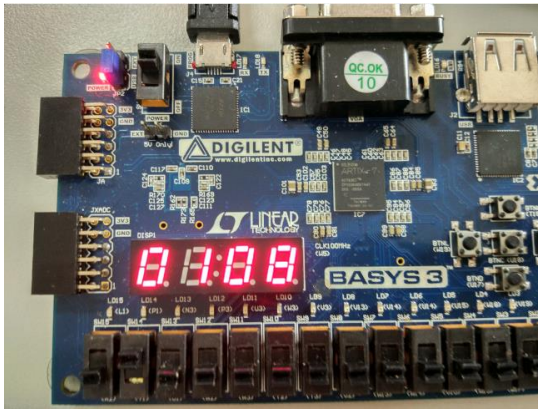
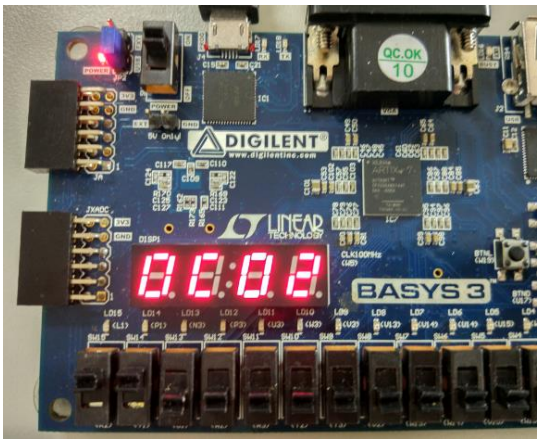
地址 汇编程序 指令代码

		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)		16 进制数代码
0x00000034	`lw $9,4($1)`	100111	00001	01001	0000 0000 0000 0100	=	9C290004
0x00000038	**`bgtz $9,1 (>0,转 40)`**	110010	01001	00000	0000 0000 0000 0001	=	C9200001
0x00000040	**`addi $9,$0,-1`**	000001	00000	01001	1111 1111 1111 1111	=	0409FFFF
0x00000044	**`j 0x00000038`**	111000	00000	00000	0000 0000 0011 1000	=	E000000E

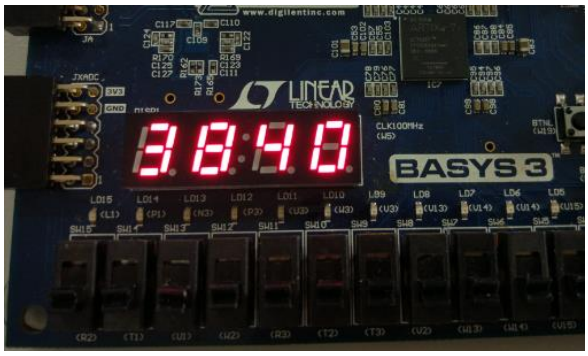
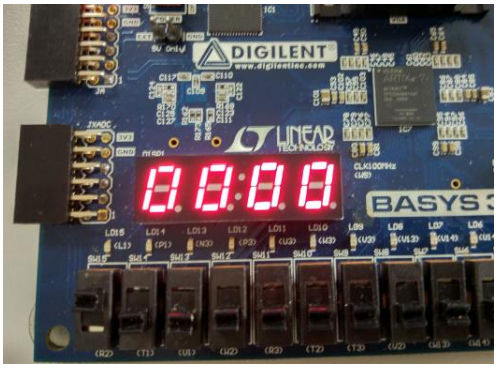
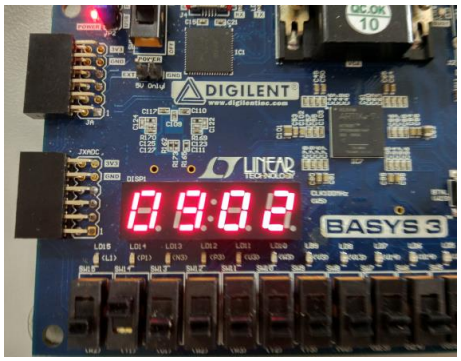
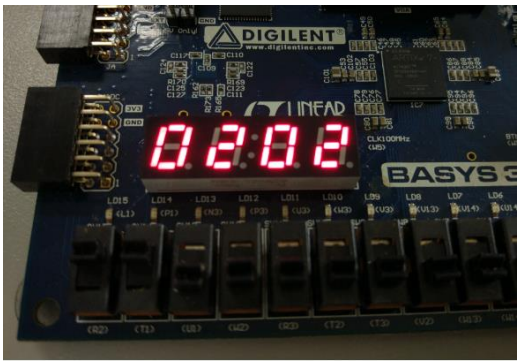
1. lw $9,4($1)



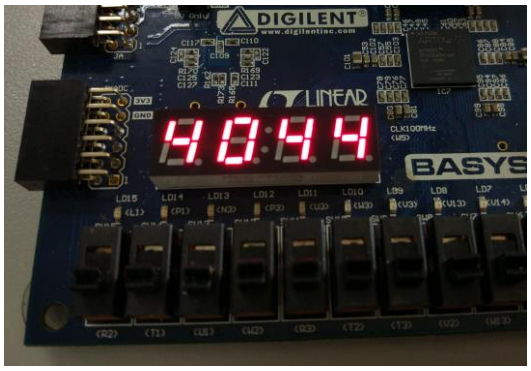


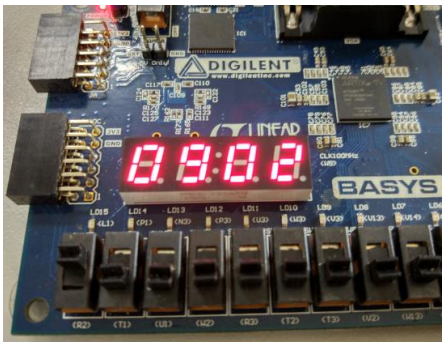



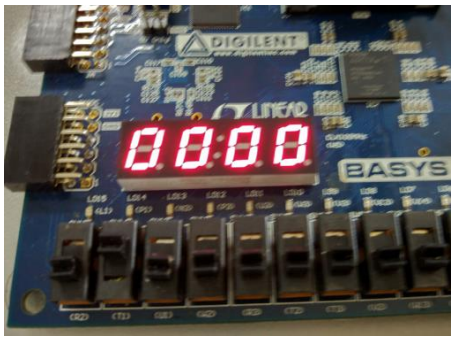
2. bgtz　$9,1 (>0,转 40)

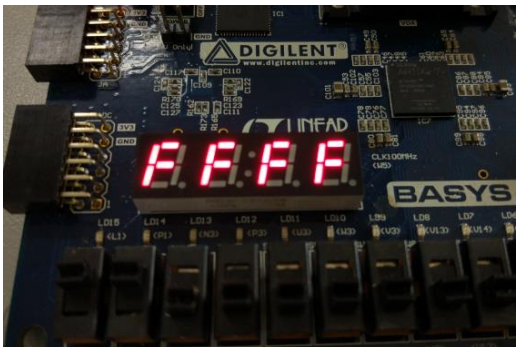





3. addi $9,$0,-1

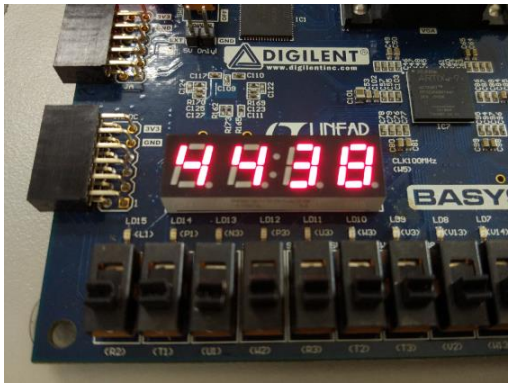


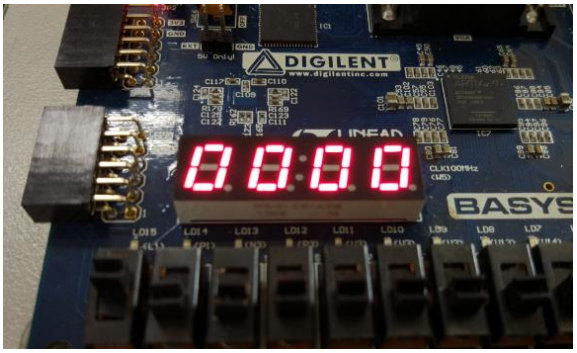


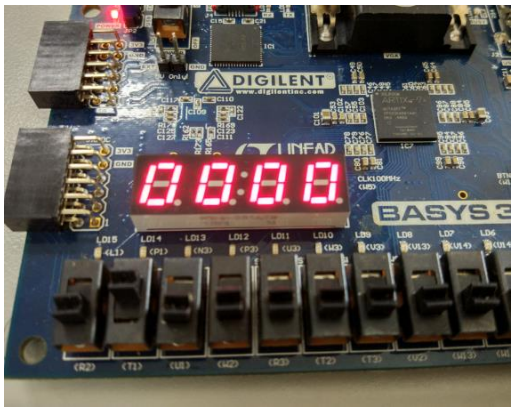


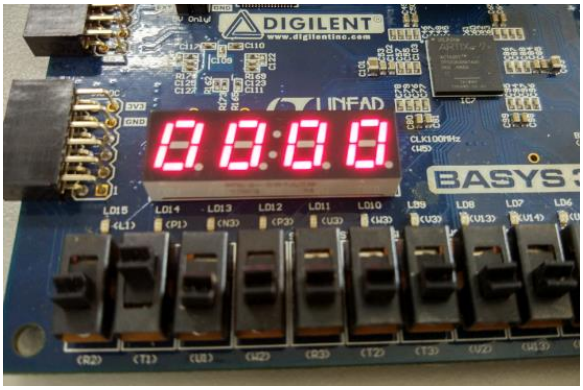


4. j 0x00000038









九、 实验心得

1. 项目编写心得

通过编写一个略微大型的硬件模块，大概知道了在设计硬件中我们所需要遵循的几个原则。

首先是模块化，层次化的重要性。

设计一个大型的硬件，代码中所涉及的线，寄存器，数量一定不小。若是不采用模块化，层次化的设计去帮助硬件简化设计，第一，编写硬件的人，很难在保证如此多线、寄存器的情况下依然能够编写的代码保证硬件的正常运行。第二，过于复杂的模块，会给 debug 带来很大的困难。在我设计 CPU 的时候，先使用数据通路图，梳理了 top 模块应有的连线，然后根据数据通路图，确定各子模块的功能，并编写测试。确保各子模块功能正确且完整后，才去编写 top 模块并测试。在这个过程中，虽然 CPU 是一个略微复杂的设计，但是每一个子模块的设计，其实难度并不高，这也是采用了模块化，层次化设计的优点所在。

其次，在每一个子模块写完之后，必须使用测试模块再进行测试。

之所以要对模块进行测试，是因为自己写的模块很多时候光看代码看不出错误。Verilog 语言中，有很多隐藏的坑点，如我们可以隐式实例化线，这样子我们线的名字就算是写错了也不会报错。而这就会给你带来一个高阻态的 bug，怎么会有一个端口连接不上呢？类似这样的 bug 还有很多，而这样的 bug，很多时候可以通过子模块的单元测试来排除缩小范围。

Debug 技能，也在这一次设计中得到了不少的提升。

硬件设计中，仿真波形图出现与自己预想不一致的情况其实是一件很正常的事情。考验我们的，是如何缩小问题范围，如何定位问题所在。在各种 bug 中，我摸索出了一套较为可行的 debug 方法。在仿真波形图中，我们从出现问题的信号入手，找到信号是从哪个模块传出来的，然后深入模块内部，检查预期信号到底是否成功生成。若成功生成，考虑模块与外部连线出了问题，如果连模块内部的信号都有相同的问题，那就从模块内部继续开始找。

2. 曾经遇到过的问题

2.1. 已解决：寄存器的写入：在下降沿写入还是上升沿写入？

其实老师给的代码中，寄存器是在时钟上升沿写入的。本来也没想多改，但是在仿真中发现，如果寄存器在时钟上升沿才写入，这个时候下一条指令已经拿上上一条指令的结果在算了，根本没有用到刚写入的值。因此为了解决这个问题，把寄存器的写入改为时钟下降沿触发。其实，时钟上升沿开始执行，时钟下降沿将结果写入，这样的安排更加适合让 CPU 有条不紊地运行。

2.2. 问题：多重时钟信号驱动时序模块

在编写代码的时候，曾出现过这样一个问题。一开始我的按键消抖模块没有时钟输入，只有按键输入，然后代码里写在按键输入到达上升沿的时候输出一个正脉冲。但是这样的代码无论如何都不能实现成功，老是提示错误。后来在按键消抖模块加入了时钟并用时钟上升沿驱动而不是按键信号上升沿之后就解决了问题。我想，这应该是 vivado 内部不允许使用多重时钟的问题。按我原来的写法，其实我的按键输入也相当于是一个时钟输入，但是这样的话，我的模块相当于有了两个相互独立的时钟，在网上查了下，说这样可能会出问题，vivado 里的每一个时序模块的时钟都需要基于一个系统时钟来驱动，不允许多重时钟，否则难以同步。但其实问题具体是如何的我也有点不清楚。

2.3. BUG：Reset 重置信号与第一个时钟周期同时产生

有这样的问题，当时钟上升沿和重置信号回到高电平同时发生，或者重置信号回到高电平先于时钟上升沿发生的时候，原本 PCData 其实已经是 PC+4 了，上升沿一到达，PC 变为 PC+4，这虽然看起来没有什么影响，但是其实隐含着这样的危险：没有运行第一条指令，直接从第二条指令开始运行，这样就导致了程序的异常。后来想清楚了。重置信号回到高电平的时候，电路中的组合电路部分（如 ALU）已经完成了运行，作为一个完整的时钟周期，应该是给一个时钟下降沿来使计算的结果写进寄存器和存储器。因

此原本每次按键是触发一个正脉冲，在 debug 后变为每次按键触发一个负脉冲。这样子，每次按键，先在时钟下降沿中，将运算结果写入寄存器和存储器以备下一条指令运行，然后在时钟上升沿中切换 PC 地址进行运算。通过这样的方式，解决了这样的问题。