



《计算机组成原理实验》 实验报告

(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 : 王凯祺

学 号 : 16337233

时 间 : 2017 年 11 月 15 日

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

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

功能：rd←rs + rt。

(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: $rd \leftarrow rt \ll (\text{zero-extend})sa$, 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: $\text{memory}[rs + (\text{sign-extend})immediate] \leftarrow rt$; **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: $rt \leftarrow \text{memory}[rs + (\text{sign-extend})immediate]$; **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) $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

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) $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

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) $pc \leftarrow pc + 4 + (\text{sign-extend})immediate \ll 2$ else $pc \leftarrow pc + 4$

==>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	00000000000000000000000000000000(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指的是一条指令用一个时钟周期完成的CPU。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。

CPU在处理指令时，一般需要经过以下几个步骤：

- (1) 取指令：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。遇到“地址转移”指令时，控制器把“转移地址”送入PC。
- (2) 指令译码：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：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：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：将指令执行的结果或者从存储器中得到的数据写回相应的寄存器中。



图 1 单周期 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: 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: 位移量 (shift amount), 移位指令用于指定移多少位;

funct: 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: 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: 地址。

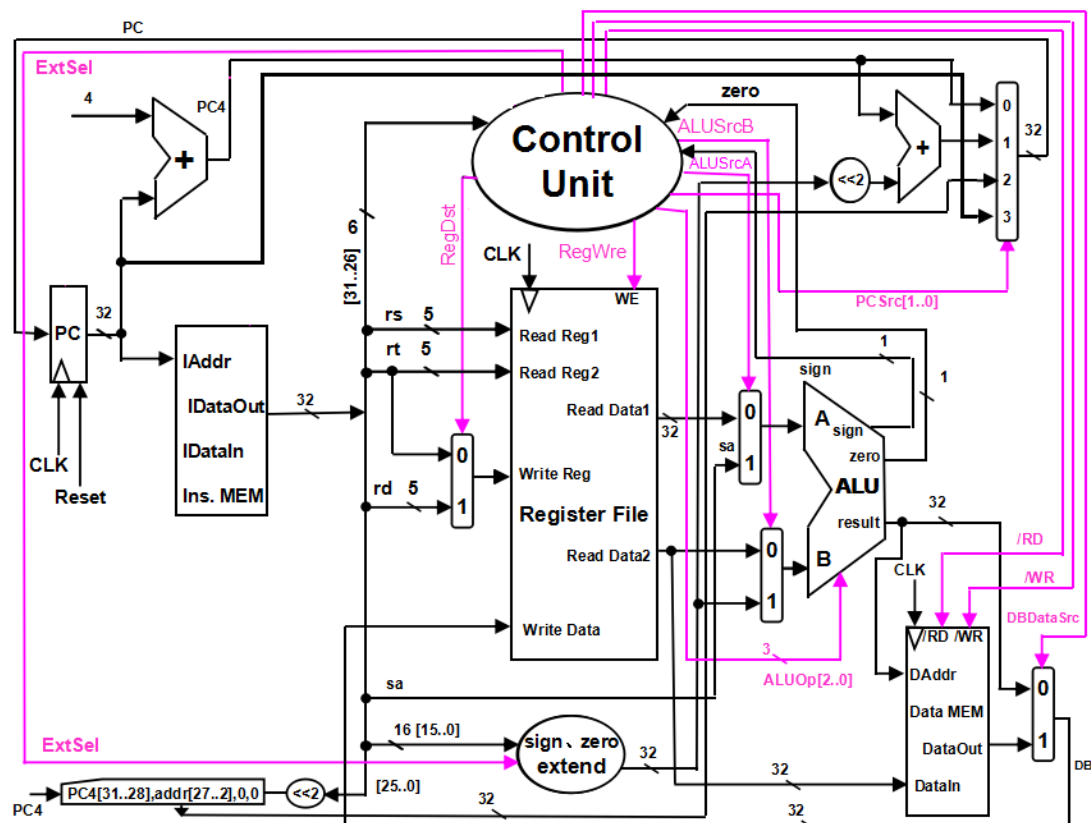


图 2 单周期 CPU 数据通路和控制线路图

图2是一个单周期CPU数据通路和控制线路图。其中,指令存储在指令存储器ROM中,数据存储在数据存储器RAM中。访问指令存储器时,须给定PC,才能读取指令;访问数据存储器时,须给定内存地址和读/写信号,才能进行读/写操作。对于寄存器组,先给出寄存器地址,读操作时,输出端直接输出相应数据;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,ALU运算功能表如表2所示。

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
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
/RD	读数据存储器, 相关指令: lw	输出高阻态

/WR	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: $pc \leftarrow pc+4$ ，相关指令：add、addi、sub、or、ori、and、slt、sll、sw、lw、beq、bne、bgtz； 01: $pc \leftarrow pc+4+(\text{sign-extend})\text{immediate}$ ，相关指令：beq、bne、bgtz； 10: $pc \leftarrow \{(pc+4)[31..28], \text{addr}[27..2], 0, 0\}$ ，相关指令：j； 11: $pc \leftarrow pc$ ，相关指令：halt	
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：

指令存储器（ROM）：

rd，指令存储器使能端输入，低电平有效

addr，指令存储器地址输入端口

dataOut，指令存储器数据输出端口

数据存储器（RAM）：

clk，数据存储器时钟信号输入，下降沿触发

address，数据存储器地址输入端口

writeData，数据存储器数据输入端口

nRD，数据存储器“读”操作使能端输入，低电平有效

nWR，数据存储器“写”操作使能端输入，低电平有效

Dataout，数据存储器数据输出端口

寄存器（RegFile）：

CLK，寄存器时钟信号输入端口，下降沿触发

RST，寄存器重置信号输入端口，低电平清除寄存器所有数据

RegWre，寄存器“写”操作使能端输入，高电平有效

ReadReg1，寄存器地址输入端口（读1）

ReadReg2，寄存器地址输入端口（读2）

WriteReg，寄存器地址输入端口（写）

WriteData，寄存器数据输入端口

ReadData1，寄存器数据输出端口1

ReadData2，寄存器数据输出端口2

PC（PC）：

clk，PC时钟信号输入，上升沿触发

reset，PC重置信号输入端口，低电平PC置零

PCIn，PC新地址输入端口

PCOut，PC当前地址输出端口

算逻运算器（ALU）：

ALUopcode，ALU操作码输入

rega，ALU操作数输入端A

regb, ALU操作数输入端B
result, ALU运算结果输出端
zero, ALU运算结果标志。结果为0, 则zero=1; 否则zero=0
sign, ALU运算结果标志。结果最高位为0, 则sign=0; 否则sign=1

表 2 ALU 运算功能表

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

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 (见表3), 再用条件分支实现, 这样控制单元部分就完成了。

表 3 控制信号与指令之间的关系表

操作	操作码 Op	ALU SrcA	ALU SrcB	PC Src	Ext Sel	Reg Wre	Reg Dst	ALU Op	/RD	/WR	DBData Src
add	000000	0	0	0	X	1	1	000	1	1	0
addi	000001	0	1	0	1	1	0	000	1	1	0
sub	000010	0	0	0	X	1	1	001	1	1	0
ori	010000	0	1	0	0	1	0	011	1	1	0
and	010001	0	0	0	X	1	1	010	1	1	0
or	010010	0	0	0	X	1	1	011	1	1	0
sll	011000	1	0	0	0	1	1	110	1	1	0
slt	011100	0	0	0	X	1	1	101	1	1	0
sw	100110	0	1	0	1	0	X	000	1	0	0
lw	100111	0	1	0	1	1	0	000	0	1	1
beq	110000	0	0	0/1	1	0	X	001	1	1	0
bne	110001	0	0	0/1	1	0	X	001	1	1	0
bgtz	110010	0	0	0/1	1	0	X	001	1	1	0

j	111000	X	X	2	X	X	X	X	X	X	X
halt	111111	X	X	3	X	X	X	X	X	X	X

四. 实验器材

电脑一台, Xilinx Vivado 软件一套, Basys 3实验板一块。

五. 实验过程与结果

1、CPU设计的思想、方法:

我实现的单周期CPU包含控制器模块、数据存储器模块、程序计数器 (PC) 块、寄存器模块和算逻运算单元 (ALU) 模块。这些模块以存储器为中心, 用控制器和ALU建立数据通路。

CPU设计流程图:



控制信号表的作用: 既能使自己理清思路, 建立起模块与模块的联系, 又能使别人明白我设计的单周期CPU的大致原理。

相关实现代码说明:

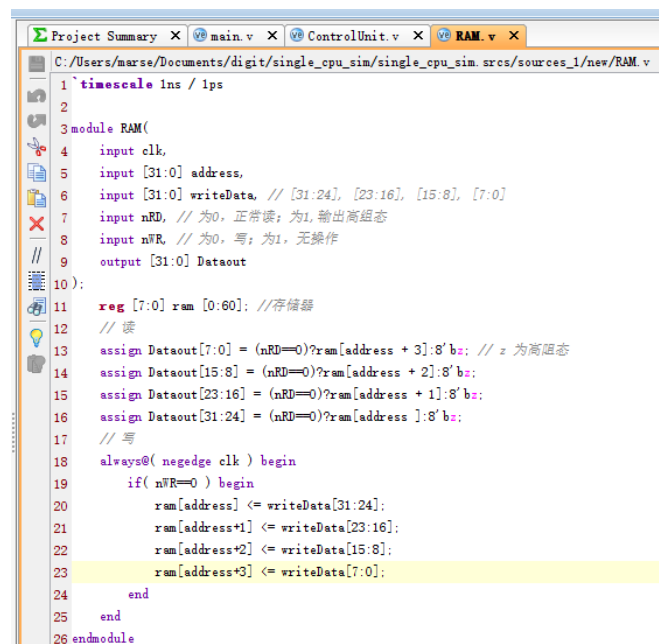
(1) 控制器模块 (Control Unit)

```
Project Summary X main.v X ControlUnit.v X
C:/Users/nurse/Documents/digit/single_cpu_sim/single_cpu_sim.srcs/sources_1/new/ControlUnit.v
1 timescale 1ns / 1ps
2
3 module ControlUnit(
4     input [5:0] Op,
5     input zero,
6     input sign,
7     output reg ALUSrcA,
8     output reg ALUSrcB,
9     output reg [1:0] PCSrc,
10    output reg ExtSel,
11    output reg RegFire,
12    output reg RegDst,
13    output reg [2:0] ALUOp,
14    output reg RD,
15    output reg WR,
16    output reg DBDataSrc
17);
18    always @ (Op or zero or sign) begin
19        if (Op == 6'b000000) begin
20            ALUSrcA = 0;
21            ALUSrcB = 0;
22            PCSrc = 0;
23            ExtSel = 0;
24            RegFire = 1;
25            RegDst = 1;
26            ALUOp = 3'b000;
27            RD = 1;
28            WR = 1;
29            DBDataSrc = 0;
30        end
31    else
32        if (Op == .....
33    end
34 endmodule
```

在设计好控制信号与指令之间的关系表（见表3）之后，使用Verilog的条件分支语句即可实现控制器模块。

控制器模块是一个组合单元，不需要使用时钟触发。

(2) 数据存储模块（RAM）

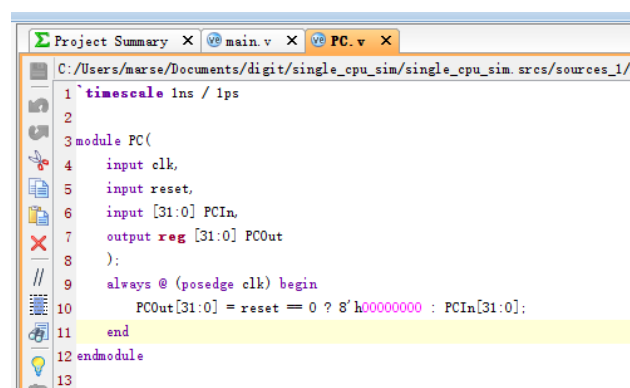


数据存储模块使用的是老师给的代码。

读存储器不需要时钟。

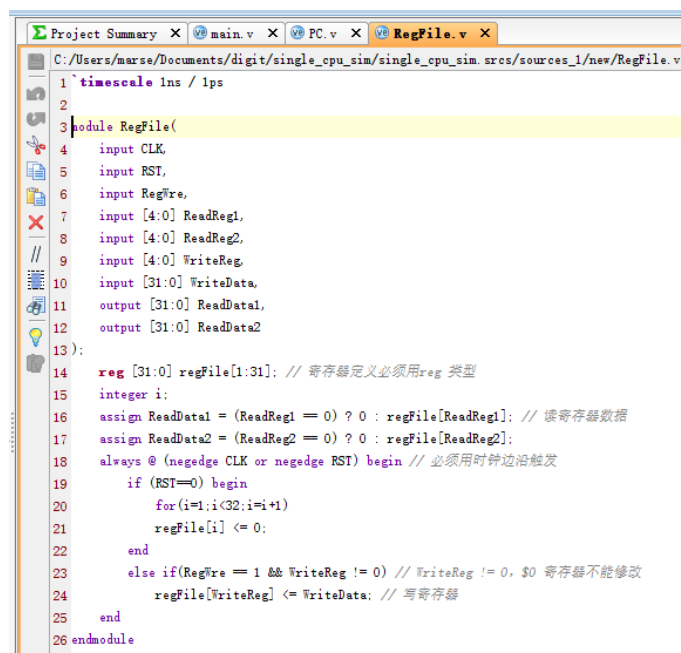
写存储器时，在时钟的下降沿触发，与PC的上升沿区别开，避免竞争。

(3) 程序计数器模块（PC）



PC模块是上升沿触发。若重置信号为低电平，则在时钟的上升沿重置PC为零；否则，在时钟的上升沿，将新地址写入PC。

(4) 寄存器模块（RegFile）



```

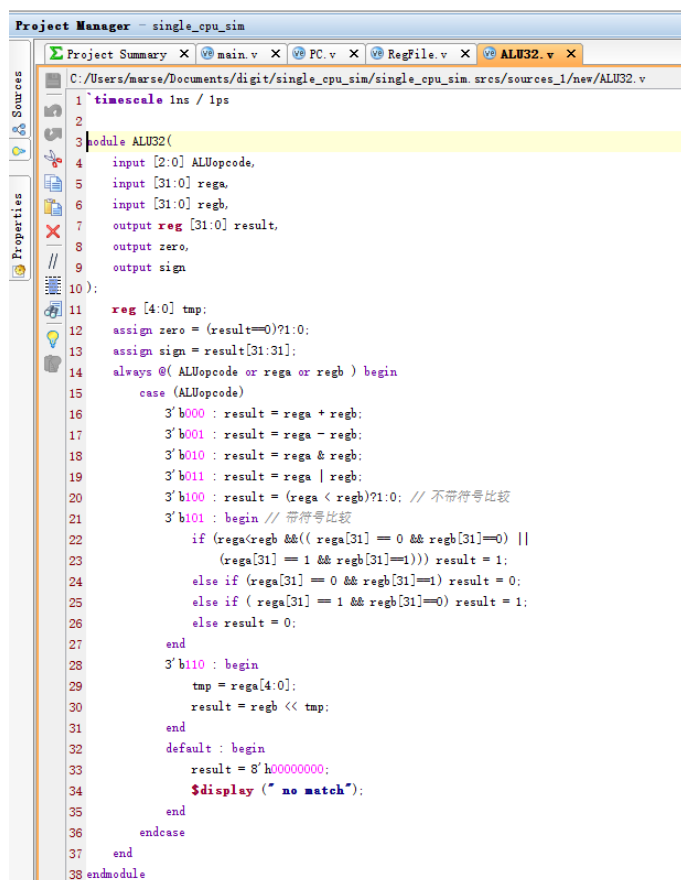
1 `timescale 1ns / 1ps
2
3 module RegFile(
4     input CLK,
5     input RST,
6     input RegWe,
7     input [4:0] ReadReg1,
8     input [4:0] ReadReg2,
9     input [4:0] WriteReg,
10    input [31:0] WriteData,
11    output [31:0] ReadData1,
12    output [31:0] ReadData2
13);
14    reg [31:0] regFile[1:31]; // 寄存器定义必须用reg 类型
15    integer i;
16    assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1]; // 读寄存器数据
17    assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
18    always @ (negedge CLK or negedge RST) begin // 必须用时钟边沿触发
19        if (RST==0) begin
20            for(i=1;i<32;i=i+1)
21                regFile[i] <= 0;
22        end
23        else if (RegWe == 1 && WriteReg != 0) // WriteReg != 0, $0 寄存器不能修改
24            regFile[WriteReg] <= WriteData; // 写寄存器
25        end
26    endmodule

```

寄存器模块用的是老师给的代码。

和数据存储器类似，读寄存器不需要时钟，但写寄存器需要在时钟的下降沿触发。

(5) 算逻运算单元模块 (ALU)



```

1 `timescale 1ns / 1ps
2
3 module ALU32(
4     input [2:0] ALUopcode,
5     input [31:0] rega,
6     input [31:0] regb,
7     output reg [31:0] result,
8     output zero,
9     output sign
10);
11    reg [4:0] tmp;
12    assign zero = (result==0)?1:0;
13    assign sign = result[31];
14    always @ (ALUopcode or rega or regb) begin
15        case (ALUopcode)
16            3'b000 : result = rega + regb;
17            3'b001 : result = rega - regb;
18            3'b010 : result = rega & regb;
19            3'b011 : result = rega | regb;
20            3'b100 : result = (rega < regb)?1:0; // 不带符号比较
21            3'b101 : begin // 带符号比较
22                if (rega<regb && ((rega[31] == 0 && regb[31]==0) ||
23                    (rega[31] == 1 && regb[31]==1))) result = 1;
24                else if (rega[31] == 0 && regb[31]==1) result = 0;
25                else if (rega[31] == 1 && regb[31]==0) result = 1;
26                else result = 0;
27            end
28            3'b110 : begin
29                tmp = rega[4:0];
30                result = regb << tmp;
31            end
32            default : begin
33                result = 8'h00000000;
34                $display (" no match");
35            end
36        endcase
37    end
38 endmodule

```

ALU模块用的是老师给的代码。

ALU不需要时钟触发。

2、验证我设计的CPU正确性:

表 4 测试程序段

地址	汇编程序	指令代码					
		op(6)	rs(5)	rt(5)	rd(5)/immediate(16)	16 进制数代码	
0x00000000	addi \$1,\$0,8	000001	00000	00001	0000 0000 0000 1000	=	0x04010008
0x00000004	ori \$2,\$0,2	010000	00000	00010	0000 0000 0000 0010	=	0x40020002
0x00000008	add \$3,\$2,\$1	000000	00010	00001	0001 1000 0000 0000	=	0x00411800
0x0000000C	sub \$5,\$3,\$2	000010	00011	00010	0010 1000 0000 0000	=	0x08622800
0x00000010	and \$4,\$5,\$2	010001	00101	00010	0010 0000 0000 0000	=	0x44A22000
0x00000014	or \$8,\$4,\$2	010010	00100	00010	0100 0000 0000 0000	=	0x48824000
0x00000018	sll \$8,\$8,1	011000	00000	01000	0100 0000 0100 0000	=	0x60084040
0x0000001C	bne \$8,\$1,-2 (≠,转 18)	110001	01000	00001	1111 1111 1111 1110	=	0xC501FFFE
0x00000020	slt \$6,\$2,\$1	011100	00010	00001	0011 0000 0000 0000	=	0x70413000
0x00000024	slt \$7,\$6,\$0	011100	00110	00000	0011 1000 0000 0000	=	0x70C03800
0x00000028	addi \$7,\$7,8	000001	00111	00111	0000 0000 0000 1000	=	0x04E70008
0x0000002C	beq \$7,\$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	=	0xC0E1FFFE
0x00000030	sw \$2,4(\$1)	100110	00001	00010	0000 0000 0000 0100	=	0x98220004
0x00000034	lw \$9,4(\$1)	100111	00001	01001	0000 0000 0000 0100	=	0x9C290004
0x00000038	bgtz \$9,1 (>0,转 40)	110010	01001	00000	0000 0000 0000 0001	=	0xC9200001
0x0000003C	halt	111111	00000	00000	0000 0000 0000 0000	=	0xFC000000
0x00000040	addi \$9,\$0,-1	000001	00000	01001	1111 1111 1111 1111	=	0x0409FFFF
0x00000044	j 0x00000038	111000	0000 0000 0000 0000 0000 0011 10			=	0xE000000E
0x00000048							
0x0000004C							

仿真文件：

```

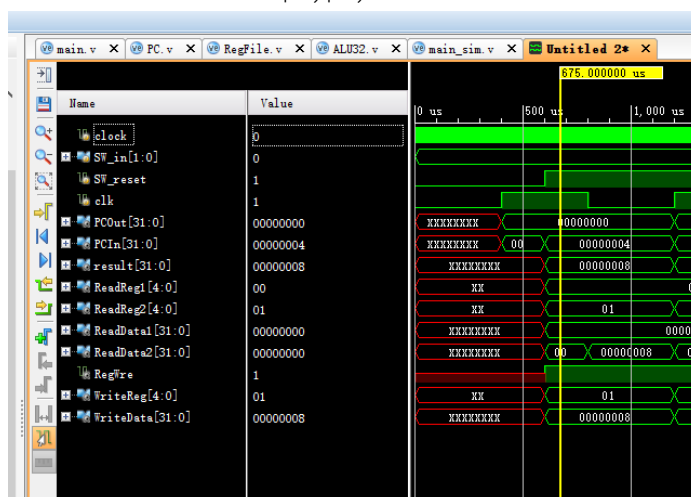
1 timescale 1ns / 1ps
2
3 module main_sim(
4
5 );
6     reg clock;
7     reg [1:0] SW_in;
8     reg SW_reset;
9     reg clk;
10
11     main uut(
12         .CLK(clock),
13         .SW_in(SW_in),
14         .SW_reset(SW_reset),
15         .button(clk)
16     );
17
18     always #1 clock = ~clock;
19     always #400000 clk = ~clk;
20     initial begin
21         clock = 0;
22         clk = 0;
23         SW_in = 0;
24         SW_reset = 0;
25         #600000;
26         SW_reset = 1;
27     end
28 endmodule
29

```

设置工作时钟间隔为400微秒。程序启动600微秒后，开始执行测试程序段。

波形图：

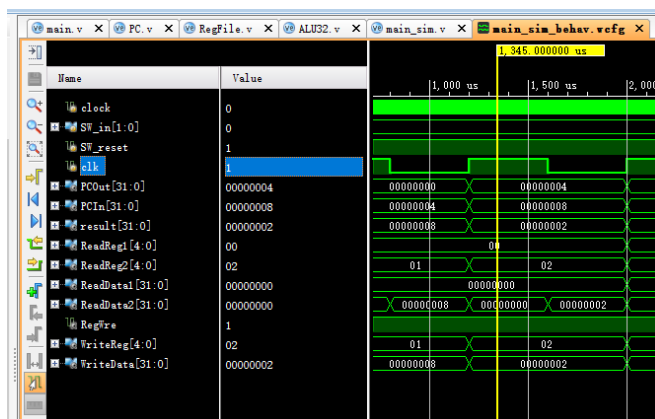
(1) 0x00000000 addi \$1,\$0,8



ReadReg1表示RS寄存器地址（\$0），ReadReg2表示RT寄存器地址（\$1）。
ReadData1表示RS寄存器数据（0），ReadData2表示RT寄存器数据（0）。
RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址（\$1），
WriteData表示写入寄存器的数据（8）。

PCOut表示当前PC值（0x0），PCIn表示新地址（0x4）。

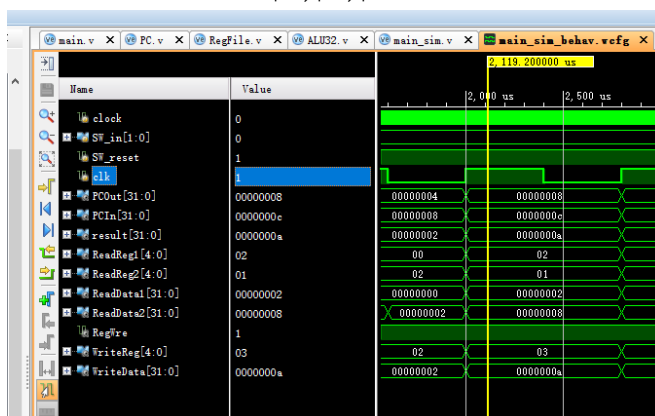
(2) 0x00000004 ori \$2,\$0,2



ReadReg1表示RS寄存器地址 (\$0)，ReadReg2表示RT寄存器地址 (\$2)。
ReadData1表示RS寄存器数据 (0)，ReadData2表示RT寄存器数据 (0)。
RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$2)，
WriteData表示写入寄存器的数据 (2)。

PCOut表示当前PC值 (0x4)，PCIn表示新地址 (0x8)。

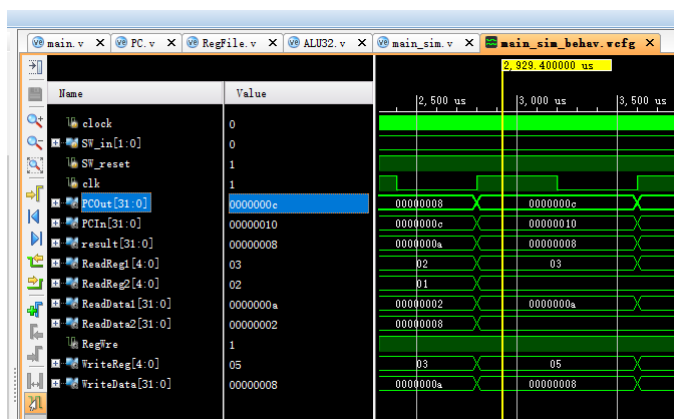
(3) 0x00000008 add \$3,\$2,\$1



ReadReg1表示RS寄存器地址 (\$2)，ReadReg2表示RT寄存器地址 (\$1)。
ReadData1表示RS寄存器数据 (2)，ReadData2表示RT寄存器数据 (8)。
RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$3)，
WriteData表示写入寄存器的数据 (10)。

PCOut表示当前PC值 (0x8)，PCIn表示新地址 (0xC)。

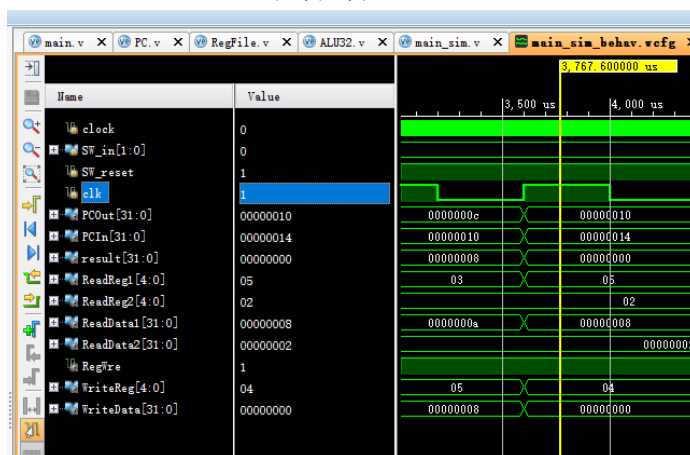
(4) 0x0000000C sub \$5,\$3,\$2



ReadReg1表示RS寄存器地址 (\$3)，ReadReg2表示RT寄存器地址 (\$2)。
ReadData1表示RS寄存器数据 (10)，ReadData2表示RT寄存器数据 (2)。
RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$5)，
WriteData表示写入寄存器的数据 (8)。

PCOut表示当前PC值 (0xC)，PCIn表示新地址 (0x10)。

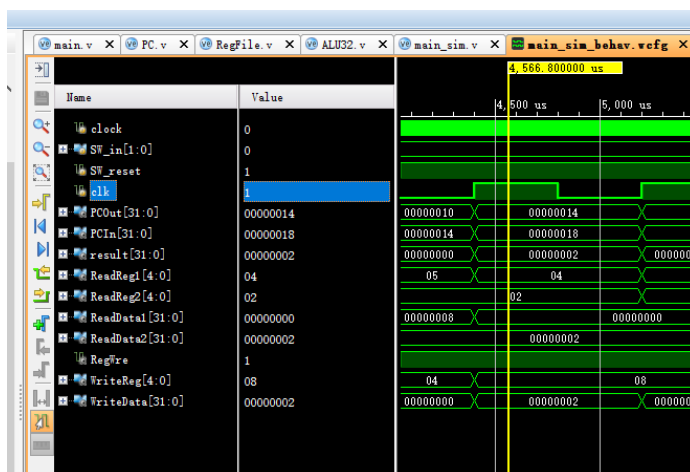
(5) 0x00000010 and \$4,\$5,\$2



ReadReg1表示RS寄存器地址 (\$5)，ReadReg2表示RT寄存器地址 (\$2)。
ReadData1表示RS寄存器数据 (8)，ReadData2表示RT寄存器数据 (2)。
RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$4)，
WriteData表示写入寄存器的数据 (0)。

PCOut表示当前PC值 (0x10)，PCIn表示新地址 (0x14)。

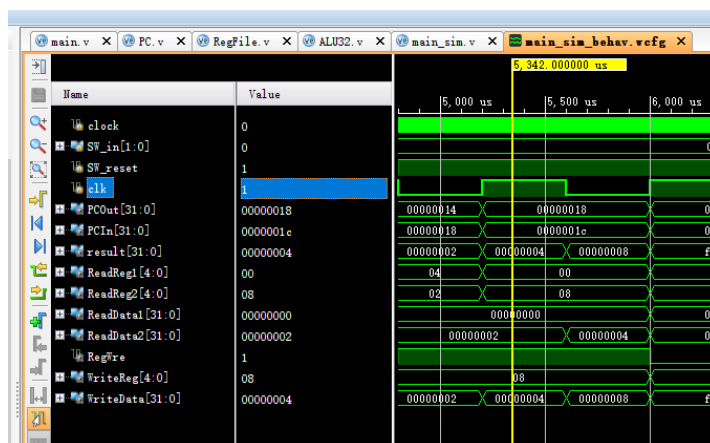
(6) 0x00000014 or \$8,\$4,\$2



ReadReg1表示RS寄存器地址 (\$4)，ReadReg2表示RT寄存器地址 (\$2)。
ReadData1表示RS寄存器数据 (0)，ReadData2表示RT寄存器数据 (2)。
RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$8)，
WriteData表示写入寄存器的数据 (2)。

PCOut表示当前PC值 (0x14)，PCIn表示新地址 (0x18)。

(7) 0x00000018 sll \$8,\$8,1



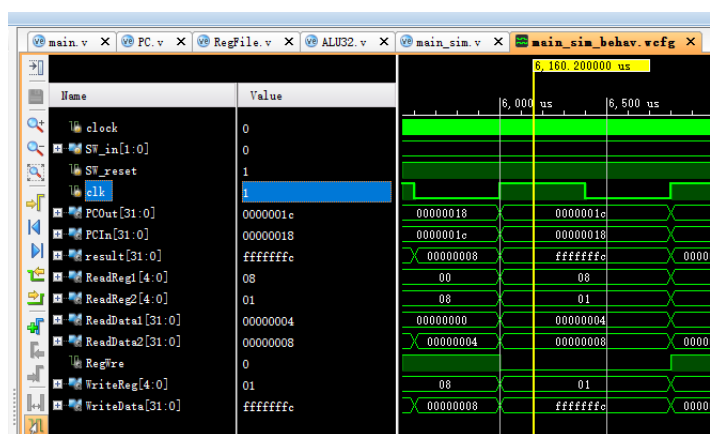
ReadReg2表示RT寄存器地址 (\$8)。

ReadData2表示RT寄存器数据 (2)。

RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$8)，WriteData表示写入寄存器的数据 (4)。

PCOut表示当前PC值 (0x18)，PCIn表示新地址 (0x1C)。

(8) 0x0000001C bne \$8,\$1,-2 (\neq ,转18)



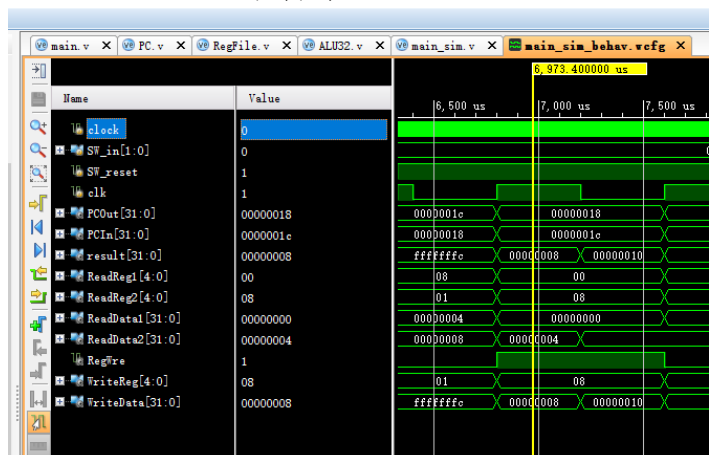
ReadReg1表示RS寄存器地址 (\$8)，ReadReg2表示RT寄存器地址 (\$1)。

ReadData1表示RS寄存器数据 (4)，ReadData2表示RT寄存器数据 (8)。

RegWre表示寄存器写入使能端 (无效)。

PCOut表示当前PC值 (0x1C)，PCIn表示新地址 (0x18)。

(9) 0x00000018 sll \$8,\$8,1



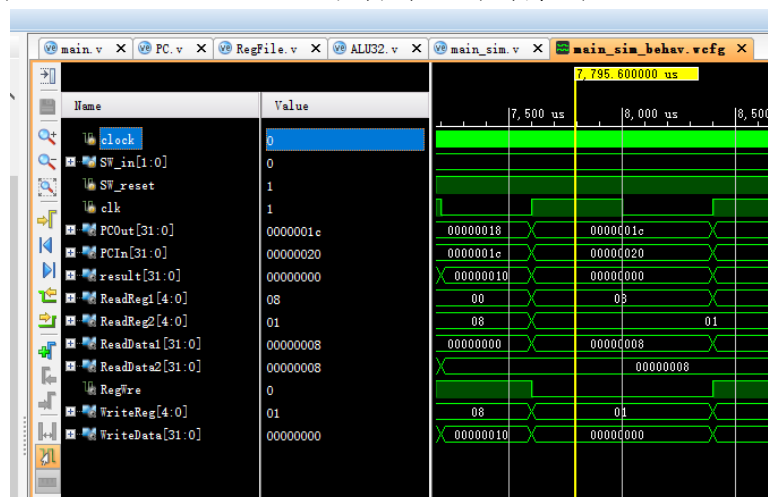
ReadReg2表示RT寄存器地址 (\$8)。

ReadData2表示RT寄存器数据 (4)。

RegWre表示寄存器写入使能端, WriteReg表示写入寄存器的地址 (\$8),
WriteData表示写入寄存器的数据 (8)。

PCOut表示当前PC值 (0x18), PCIn表示新地址 (0x1C)。

(10) 0x0000001C bne \$8,\$1,-2 (≠,转18)



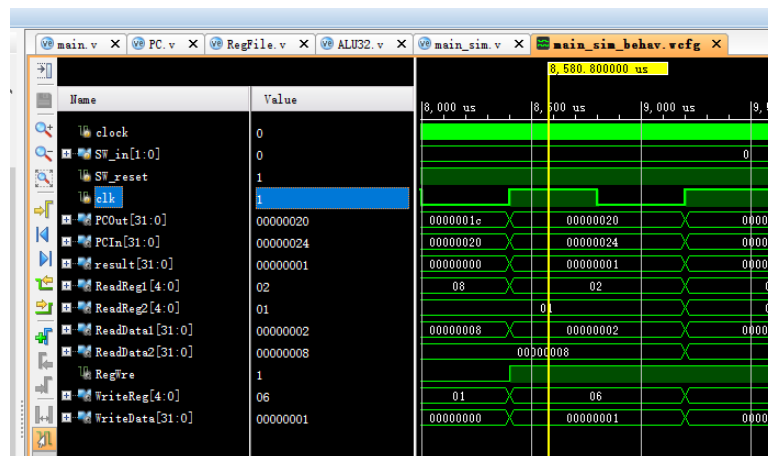
ReadReg1表示RS寄存器地址 (\$8), ReadReg2表示RT寄存器地址 (\$1)。

ReadData1表示RS寄存器数据 (8), ReadData2表示RT寄存器数据 (8)。

RegWre表示寄存器写入使能端 (无效)。

PCOut表示当前PC值 (0x1C), PCIn表示新地址 (0x20)。

(11) 0x00000020 slt \$6,\$2,\$1



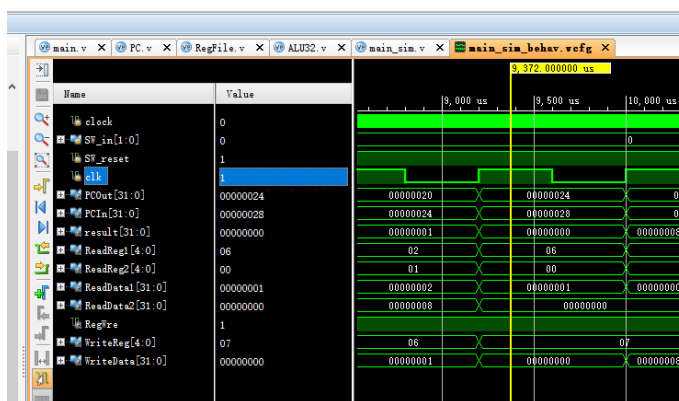
ReadReg1表示RS寄存器地址 (\$2), ReadReg2表示RT寄存器地址 (\$1)。

ReadData1表示RS寄存器数据 (2), ReadData2表示RT寄存器数据 (8)。

RegWre表示寄存器写入使能端, WriteReg表示写入寄存器的地址 (\$6),
WriteData表示写入寄存器的数据 (1)。

PCOut表示当前PC值 (0x20), PCIn表示新地址 (0x24)。

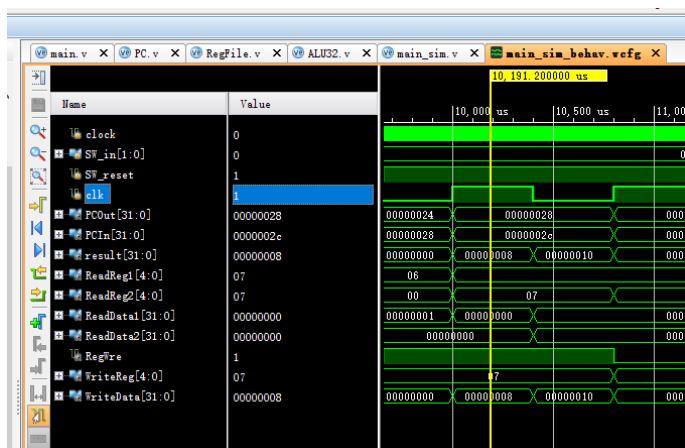
(12) 0x00000024 slt \$7,\$6,\$0



ReadReg1表示RS寄存器地址 (\$6)，ReadReg2表示RT寄存器地址 (\$0)。
 ReadData1表示RS寄存器数据 (1)，ReadData2表示RT寄存器数据 (0)。
 RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$7)，
 WriteData表示写入寄存器的数据 (0)。

PCOut表示当前PC值 (0x24)，PCIn表示新地址 (0x28)。

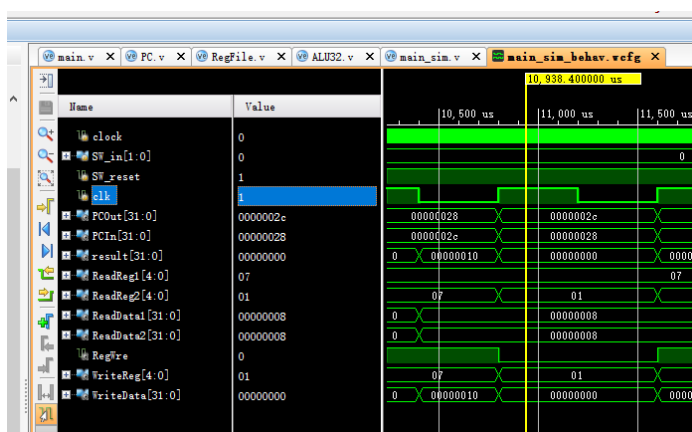
(13) 0x00000028 addi \$7,\$7,8



ReadReg1表示RS寄存器地址 (\$7)，ReadReg2表示RT寄存器地址 (\$7)。
 ReadData1表示RS寄存器数据 (0)，ReadData2表示RT寄存器数据 (0)。
 RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$7)，
 WriteData表示写入寄存器的数据 (8)。

PCOut表示当前PC值 (0x28)，PCIn表示新地址 (0x2C)。

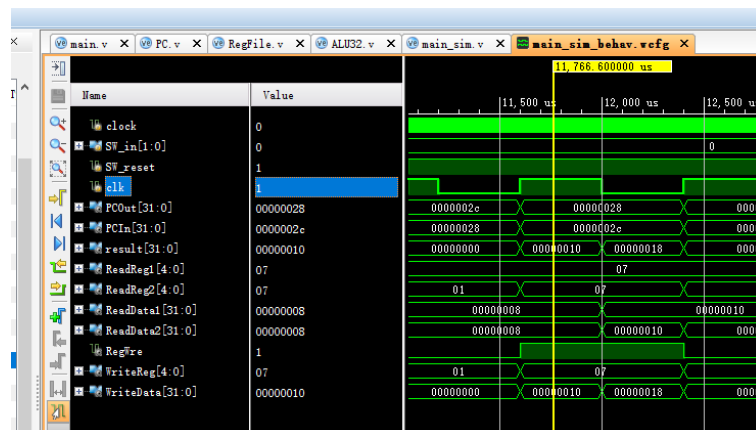
(14) 0x0000002C beq \$7,\$1,-2 (=,转28)



ReadReg1表示RS寄存器地址 (\$7) , ReadReg2表示RT寄存器地址 (\$1) 。
ReadData1表示RS寄存器数据 (8) , ReadData2表示RT寄存器数据 (8) 。
RegWre表示寄存器写入使能端 (无效) 。

PCOut表示当前PC值 (0x2C) , PCIn表示新地址 (0x28) 。

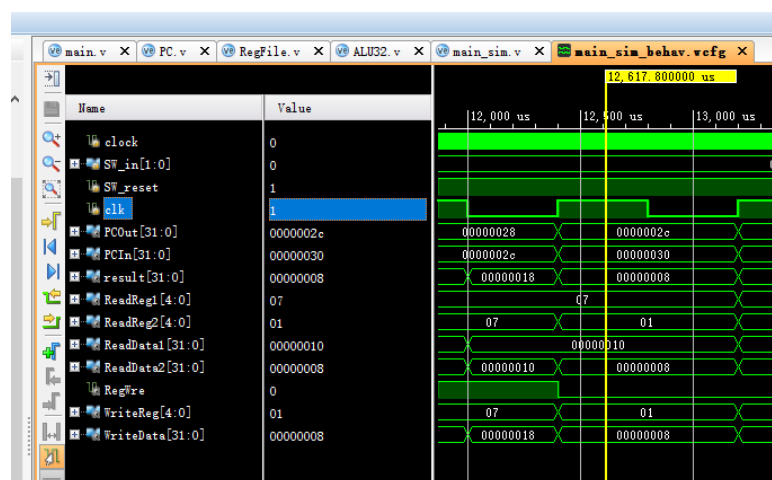
(15) 0x00000028 addi \$7,\$7,8



ReadReg1表示RS寄存器地址 (\$7) , ReadReg2表示RT寄存器地址 (\$7) 。
ReadData1表示RS寄存器数据 (8) , ReadData2表示RT寄存器数据 (8) 。
RegWre表示寄存器写入使能端, WriteReg表示写入寄存器的地址 (\$7) ,
WriteData表示写入寄存器的数据 (16) 。

PCOut表示当前PC值 (0x28) , PCIn表示新地址 (0x2C) 。

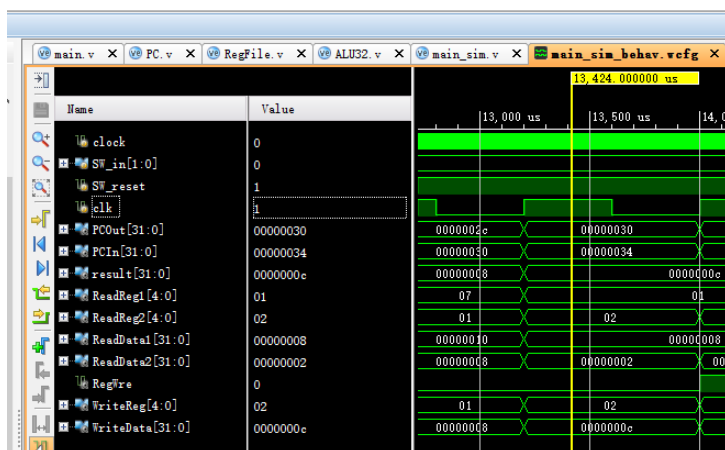
(16) 0x0000002C beq \$7,\$1,-2 (=,转28)



ReadReg1表示RS寄存器地址 (\$7) , ReadReg2表示RT寄存器地址 (\$1) 。
ReadData1表示RS寄存器数据 (16) , ReadData2表示RT寄存器数据 (8) 。
RegWre表示寄存器写入使能端 (无效) 。

PCOut表示当前PC值 (0x2C) , PCIn表示新地址 (0x30) 。

(17) 0x00000030 sw \$2,4(\$1)



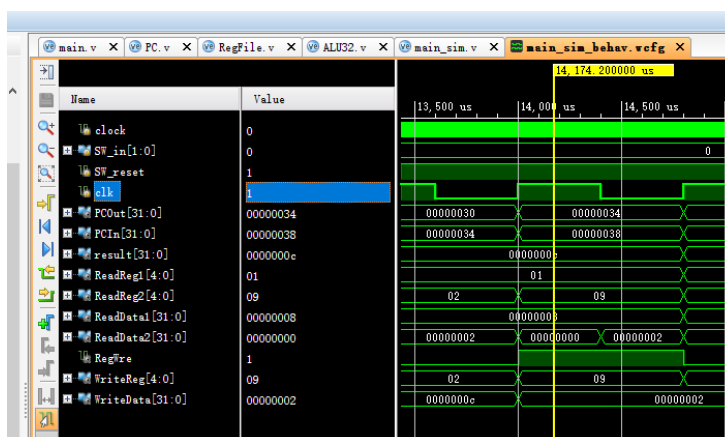
ReadReg1表示RS寄存器地址 (\$1)，ReadReg2表示RT寄存器地址 (\$2)。

ReadData1表示RS寄存器数据 (8)，ReadData2表示RT寄存器数据 (2)。

RegWre表示寄存器写入使能端 (无效)。

PCOut表示当前PC值 (0x30)，PCIn表示新地址 (0x34)。

(18) 0x00000034 lw \$9,4(\$1)



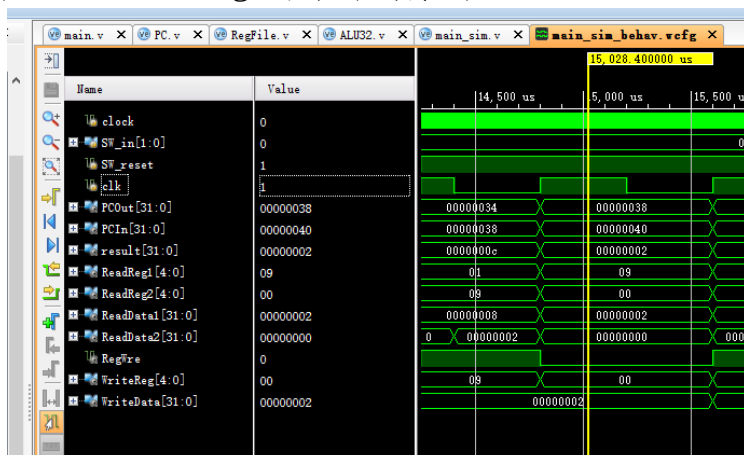
ReadReg1表示RS寄存器地址 (\$1)，ReadReg2表示RT寄存器地址 (\$9)。

ReadData1表示RS寄存器数据 (8)，ReadData2表示RT寄存器数据 (0)。

RegWre表示寄存器写入使能端，WriteReg表示写入寄存器的地址 (\$9)，WriteData表示写入寄存器的数据 (2)。

PCOut表示当前PC值 (0x34)，PCIn表示新地址 (0x38)。

(19) 0x00000038 bgtz \$9,1 (>0,转40)



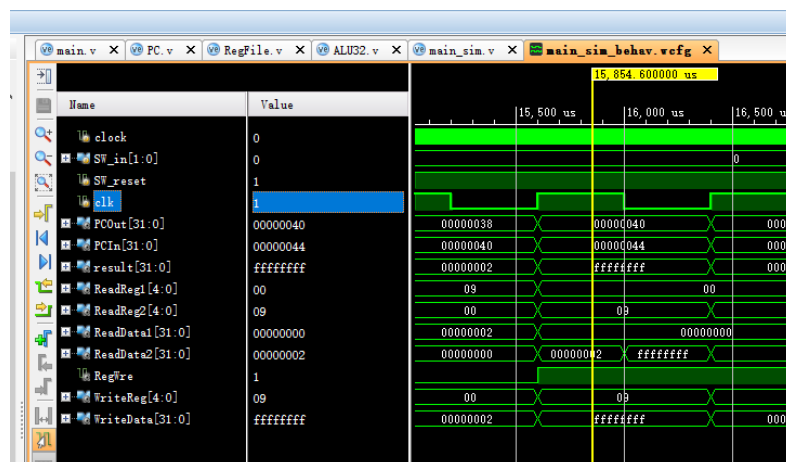
ReadReg1表示RS寄存器地址 (\$9) 。

ReadData1表示RS寄存器数据 (9) 。

RegWre表示寄存器写入使能端 (无效) 。

PCOut表示当前PC值 (0x38) , PCIn表示新地址 (0x40) 。

(20) 0x00000040 addi \$9,\$0,-1



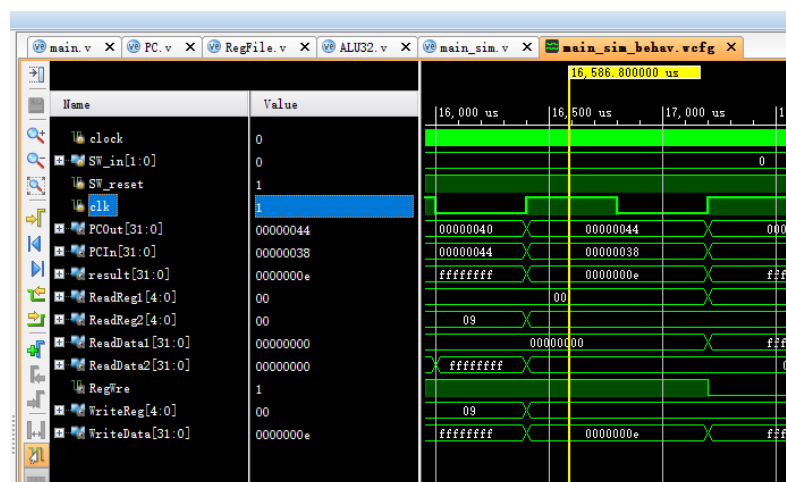
ReadReg1表示RS寄存器地址 (\$0) , ReadReg2表示RT寄存器地址 (\$9) 。

ReadData1表示RS寄存器数据 (0) , ReadData2表示RT寄存器数据 (2) 。

RegWre表示寄存器写入使能端, WriteReg表示写入寄存器的地址 (\$9) , WriteData表示写入寄存器的数据 (-1) 。

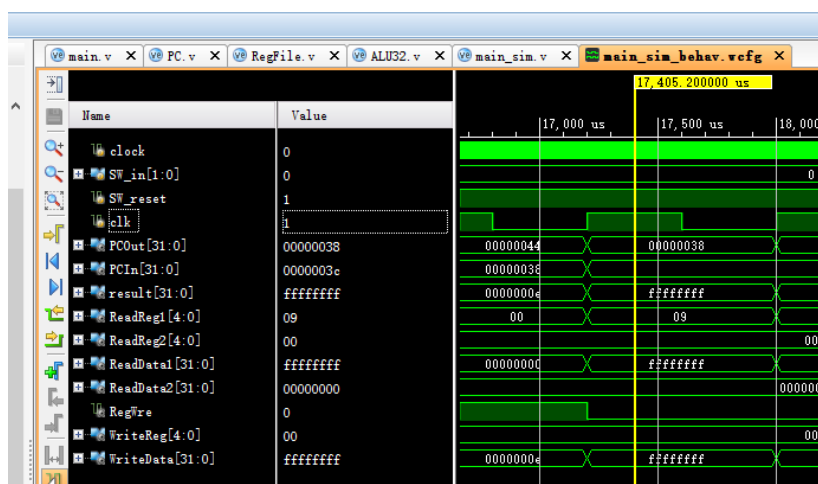
PCOut表示当前PC值 (0x40) , PCIn表示新地址 (0x44) 。

(21) 0x00000044 j 0x00000038



PCOut表示当前PC值 (0x44) , PCIn表示新地址 (0x38) 。

(22) 0x00000038 bgtz \$9,1 (>0,转40)



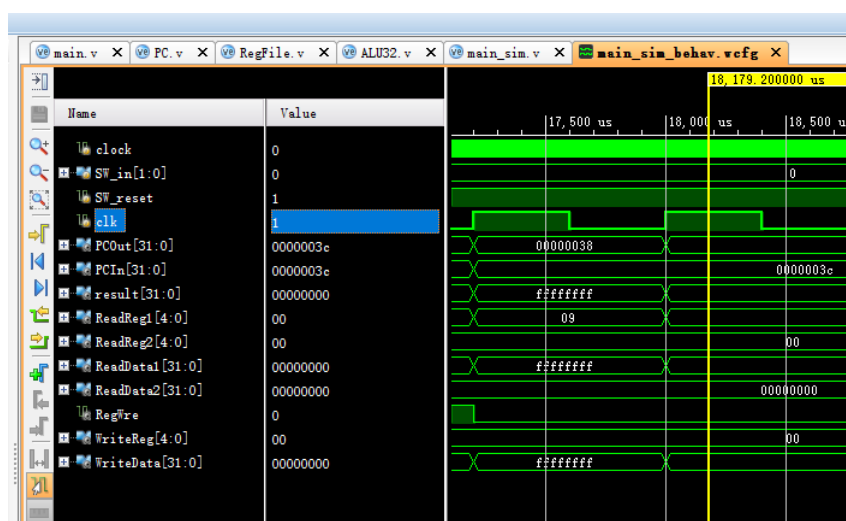
ReadReg1表示RS寄存器地址（\$9）。

ReadData1表示RS寄存器数据（-1）。

RegWre表示寄存器写入使能端（无效）。

PCOut表示当前PC值（0x38），PCIn表示新地址（0x3C）。

(23) 0x0000003C halt



PCOut表示当前PC值（0x3C），PCIn表示新地址（0x3C）。

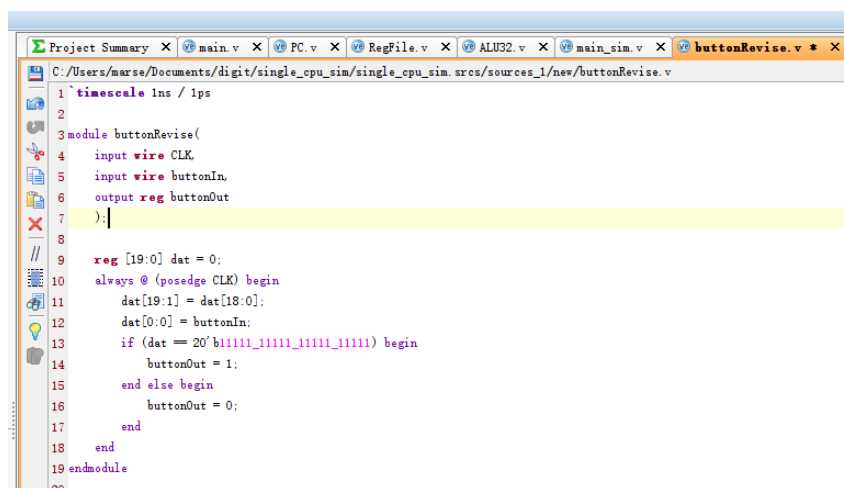
3、在Basys板上实现CPU：

在Basys实验板实现CPU还需要实现下面的模块：

- (1) 数码管显示模块
- (2) 按键消抖模块

其中，数码管显示模块在数字电路的课程已经实现过了，将代码复制过来即可。

按键消抖模块可以使用移位寄存器来做。注意传进来的CLK一定要是分频过的时钟。



```

1 `timescale 1ns / 1ps
2
3 module buttonRevise(
4     input wire CLK,
5     input wire buttonIn,
6     output reg buttonOut
7 );
8
9     reg [19:0] dat = 0;
10    always @ (posedge CLK) begin
11        dat[19:1] = dat[18:0];
12        dat[0:0] = buttonIn;
13        if (dat == 20'b11111_11111_11111_11111) begin
14            buttonOut = 1;
15        end else begin
16            buttonOut = 0;
17        end
18    end
19 endmodule
20

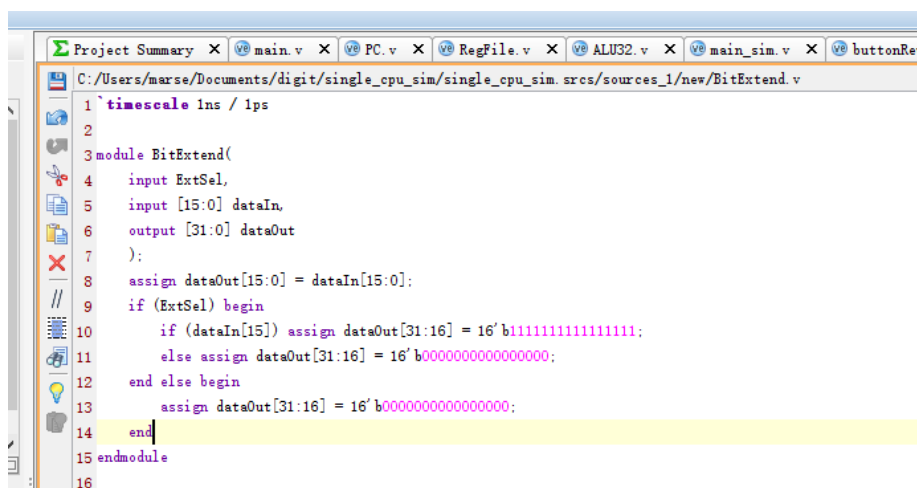
```

在Basys实验板上，我的测试程序能正确运行。PC:PC+4输出正确，RS地址:RS数据输出正确，RT地址:RT数据输出正确。

六. 实验心得

在本实验中,最重要的一步就是画数据通路图以及画控制信号表。一旦数据通路图画好,控制信号表做好,整个实验就成功了一半。因为做好这些,思路就很清晰,写代码就不容易错。

我首先在编译的过程遇到了麻烦。本来是写成这样的:

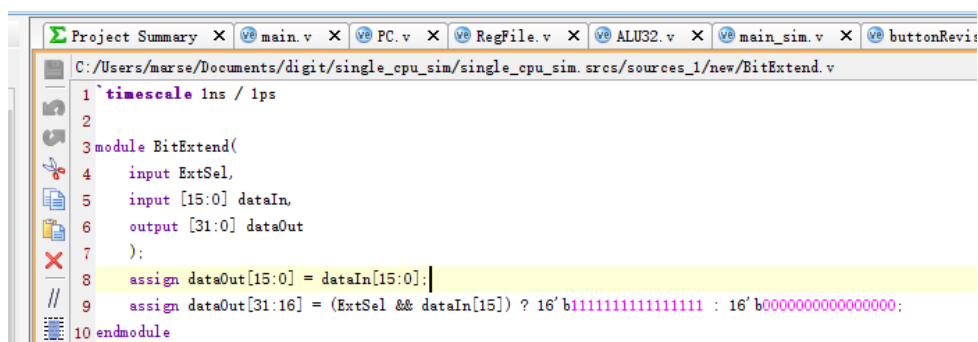


```

1 `timescale 1ns / 1ps
2
3 module BitExtend(
4     input ExtSel,
5     input [15:0] dataIn,
6     output [31:0] dataOut
7 );
8     assign dataOut[15:0] = dataIn[15:0];
9     if (ExtSel) begin
10         if (dataIn[15]) assign dataOut[31:16] = 16'b1111111111111111;
11         else assign dataOut[31:16] = 16'b0000000000000000;
12     end else begin
13         assign dataOut[31:16] = 16'b0000000000000000;
14     end
15 endmodule
16

```

上面这种情况犯的一个错误是把本该放在always/initial语句里的条件分支语句单独拿出来使用,编译当然不能通过。于是我把犯过的此类错误改用下面这种简单的写法,或者将其放入always语句中,编译就能通过了。



```

1 `timescale 1ns / 1ps
2
3 module BitExtend(
4     input ExtSel,
5     input [15:0] dataIn,
6     output [31:0] dataOut
7 );
8     always @(*) begin
9         assign dataOut[15:0] = dataIn[15:0];
10        assign dataOut[31:16] = (ExtSel && dataIn[15]) ? 16'b1111111111111111 : 16'b0000000000000000;
11    end
12 endmodule
13

```

在仿真时，我也遇到了麻烦。仿真时遇到了很多X（不确定态）和很多Z（高阻态）。于是我把正常的变量挑出来，观察正常的变量与不确定态变量的关系。最后发现是变量名打错。把所有的变量名修改正确后，程序就能通过仿真了。

在实现时，Vivado提示我有一个错误：ERROR:[Place 30-574] Poor placement for routing between an IO pin and BUFG. If this sub optimal condition is acceptable for this design, you may use the CLOCK_DEDICATED_ROUTE constraint in the .xdc file to demote this message to a WARNING. However, the use of this override is highly discouraged. These examples can be used directly in the .xdc file to override this clock rule. 虽然不知道为什么报错，但是我按照Vivado的提示，在约束文件中加入了set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SW_reset_IBUF]这句话就可以通过实现了。

烧到板上去之后，又出现问题了，执行几步后查看\$1寄存器，发现竟然是0（正常情况下是8，因为第一条命令是addi \$1, \$0, 8）。试了半天没试出来，后来想想，可能是上升沿/下降沿触发的问题。原来是因为PC是上升沿触发，寄存器是下降沿触发，而我开机之后按按钮，未将数据写入寄存器的情况下就改变了PC。于是我在按住按钮的情况下打开了开关，原来的问题消失了。通过改变按钮的行为，从原来的按下输出高电平改为按下输出低电平，问题得到完美解决。

通过本实验，我更加深入地了解了单周期CPU的工作原理，并能亲手做成一个，烧到板上去。我为自己感到高兴！