中山大學
SUN YAT-SEN UNIVERSITY

《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）：	18 软件工程 2 班
学 生 姓 名 ：	鲁睿、罗炜乐
学　　号 ：	18342067、18342069
时　　间 ：	2019　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。

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。**reserved 为预留部分，即未用，一般填“0”。**

（2）sub rd , rs , rt

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

功能：rd←rs - rt。

（3）addiu rt , rs ,**immediate**

000010	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。

==> 逻辑运算指令

（4）andi rt , rs ,**immediate**

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

功能：rt←rs & (sign-extend)**immediate；immediate** 做符号扩展再参加“与”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）ori rt , rs ,**immediate**

010010	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

（7）or rd , rs , rt

010011	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（8）sll rd, rt,sa

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

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

==>比较指令

（9） slti rt, rs,**immediate** 带符号数

011100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if (rs< (sign-extend)**immediate**) rt =1 else rt=0, 具体请看表 2 ALU 运算功能表，带符号。

==> 存储器读/写指令

（10）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。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) 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

（12）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”。

（13）bne rs,rt,**immediate**

110001	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：与 beq 不同点是，不等时转移，相等时顺序执行。

（14）bltz rs,**immediate**

110010	rs(5 位)	00000	**immediate**(16 位)

功能：if(rs<$zero) pc←pc + 4 + (sign-extend)**immediate** <<2 else pc ←pc + 4。

==>跳转指令

（15）j　addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，无条件跳转。

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

==> 停机指令

（16）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。**）

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

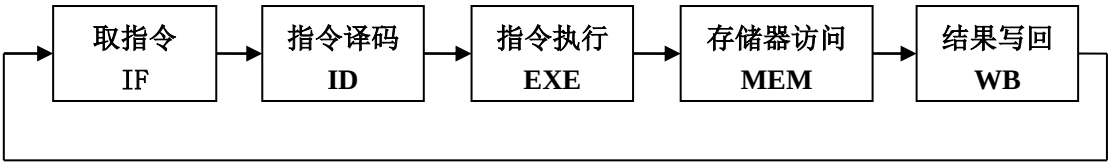


图 1　单周期 CPU 指令处理过程

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。

0MIPS 指令的三种格式：

R 类型：

31	2625	2120	1615	1110	65	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型：

31	2625	2120	1615	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型：

31	2625	0
op	address	
6 位	26 位	

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（Laod）/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。

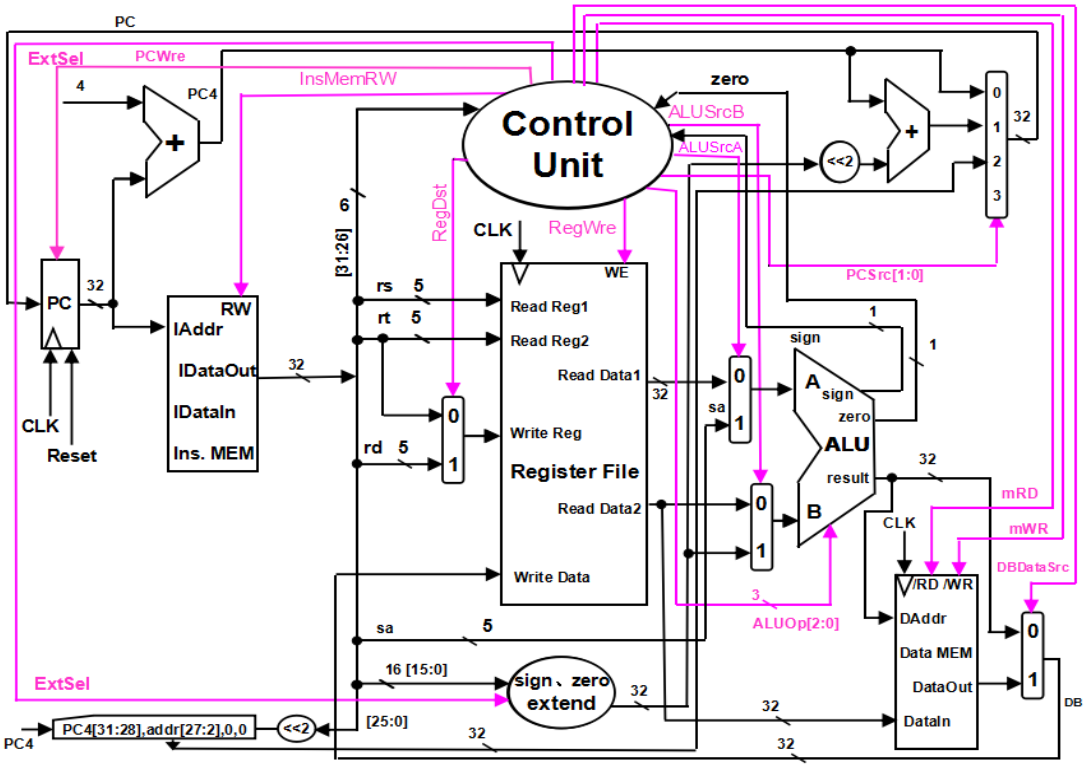


图 2 单周期 CPU 数据通路和控制线路图

上图(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。

而其中控制信号作用如下表1所示，表2为ALU计算功能表，表3为控制信号与指令关系

相关部件及引脚说明：

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、addiu、or、and、andi、ori、slti、beq、bne、bltz、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、bne、bltz、sll	来自 sign 或 zero 扩展的立即数，相关指令：addiu、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、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、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	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**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addiu、andi、ori、slti、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、sll
ExtSel	(zero-extend)**immediate(0 扩展)**，相关指令：andi、ori	(sign-extend)**immediate (符号扩展)**，相关指令：addiu、slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)； 01：pc<－pc+4+(sign-extend)**immediate<<2**，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1)； 10：pc<－{(pc+4)[31:28],addr[27:2],2'b00}，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	

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

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<B 不带符号
110	Y=(((A<B)&&(A[31] == B[31])) ||((A[31]==1&& B[31] == 0))) ? 1:0	比较 A<B 带符号
111	Y = A ⊕ B	异或

表2 ALU运算功能表

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(表 3)，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。

表3 控制信号与指令关系表

指令	Reset	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW
add	x	1	0	0	0	1	1
sub	x	1	0	0	0	1	1
addiu	x	1	0	1	0	1	1
andi	x	1	0	1	0	1	1
and	x	1	0	0	0	1	1
ori	x	1	0	1	0	1	1
or	x	1	0	0	0	1	1
sll	x	1	1	0	0	1	1
slti	x	1	0	1	0	1	1
sw	x	1	0	1	0	0	1
lw	x	1	0	1	1	1	1
beq	x	1	0	0	0	0	1
bne	x	1	0	0	0	0	1
bltz	x	1	0	0	0	0	1
j	x	1	0	0	0	1	1
halt	x	0	0	0	0	0	1

接下表

指令	mRD	mWR	RegDst	ExtSel	PCSrc[1..0]	ALUOp[2..0]
add	0	0	1	1	00	000
sub	0	0	1	1	00	001
addiu	0	0	0	1	00	000
andi	0	0	0	0	00	100
and	0	0	1	1	00	100
ori	0	0	0	0	00	011
or	0	0	1	1	00	011
sll	0	0	1	1	00	010
slti	0	0	0	1	00	110
sw	0	1	1	1	00	000
lw	1	0	0	1	00	000
beq	0	0	1	1	0(zero==0?0:1)	111
bne	0	0	1	1	0(zero==1?0:1)	111
bltz	0	0	1	1	0(sign==0?0:1)	000
j	0	0	1	1	10	000
halt	0	0	1	1	xx	000

根据上表(表3)得出表达式为：

表4 控制信号逻辑表达式

PCWre = op != halt;
ALUSrcA = op == sll;
ALUSrcB = op == addiu || op == andi|| op == ori || op == slti || op == sw || op == lw;
DBDataSrc = op == lw;
RegWre = op != beq && op != bne && op != bltz && op != sw && op != halt;
InsMemRW = 1;
mRD = op == lw;
mWR = op == sw;
RegDst = op != addiu && op != andi && op != ori && op != slti && op != lw;
ExtSel = op != andi && op != ori;
PCSrc[0] = op == beq && zero == 1 || op == bne && zero == 0 || op == bltz && sign == 1;
PCSrc[1] = op == j;
ALUOp[0] = (op == sub || op == or || op == ori|| op== beq || op == bne) ? 1 : 0;
ALUOp[1] = (op == sll||op == or||op == slti||op == ori||op == beq||op == bne) ? 1 : 0;
ALUOp[2] = (op == and| op == slti || op == andi || op == beq || op == bne) ? 1 : 0;

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

1、 设计思想

在设计的过程中使用模块化设计的思想，逐个模块设计，并使某些模块可以共用，从而减少工作量。

2、 CPU设计过程

在设计CPU过程中，我们首先要搞清楚单周期CPU的各个模块的关系，只有搞清楚这部

分内容，然后才能开始实现各个模块，这样能够帮助我们更好地理解CPU的功能，以及更加高效地实现整个CPU的设计过程。

如下图(图3)，就是各个模块的关系树状图：

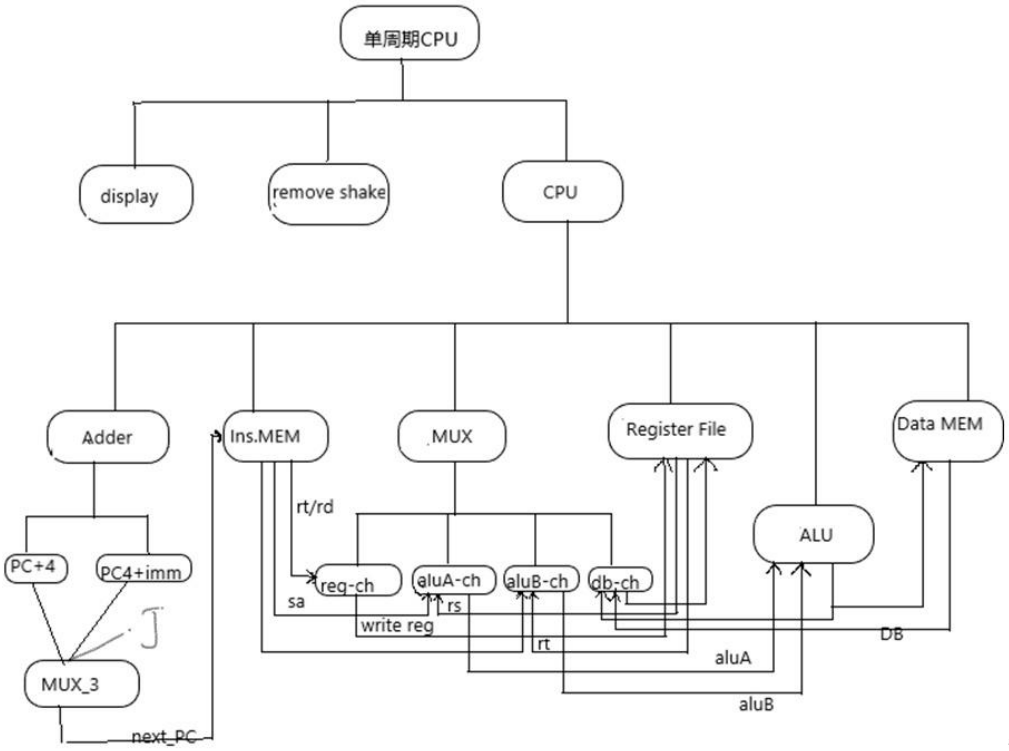


图3 模块关系树状图

图中不带箭头的线都表示关系，带箭头的线表示数据的转移

Hard_CPU.v

顶层文件，用于连接display、remove_shake、CPU三个模块，并能调节显示模式。

```
1.  `timescale 1ns / 1ps
2.  module Hard_CPU(
3.      input[1:0] display_mode,
4.      input CLK, Reset, Button,
5.      output[3:0] AN, //数码管位选择信号
6.      output[7:0] Out
7.  );
8.
9.      wire[31:0] ALU_Out, CurPC, WriteData, Reg1Out, Reg2Out, instcode, newAdd
    ress;
10.     wire myCLK, myReset;
11.     reg[3:0] store; //记录当前要显示位的值
12.
```

```
13.     CPU cpu(myCLK, myReset, CurPC, newAddress, instcode, Reg1Out, Reg2Out, A
    LU_Out, WriteData);
14.     remove_shake remove_shake_clk(CLK, Button, myCLK);
15.     remove_shake remove_shake_reset(CLK, Reset, myReset);
16.     clk_slow slowclk(CLK, myReset, AN);
17.     display show_in_7segment(store, myReset, Out);
18.
19.     always@(myCLK)begin
20.        case(AN)
21.             4'b1110:begin
22.                 case(display_mode)
23.                     2'b00: store <= newAddress[3:0];
24.                     2'b01: store <= Reg1Out[3:0];
25.                     2'b10: store <= Reg2Out[3:0];
26.                     2'b11: store <= WriteData[3:0];
27.                 endcase
28.             end
29.             4'b1101:begin
30.                 case(display_mode)
31.                     2'b00: store <= newAddress[7:4];
32.                     2'b01: store <= Reg1Out[7:4];
33.                     2'b10: store <= Reg2Out[7:4];
34.                     2'b11: store <= WriteData[7:4];
35.                 endcase
36.             end
37.             4'b1011:begin
38.                 case(display_mode)
39.                     2'b00: store <= CurPC[3:0];
40.                     2'b01: store <= instcode[24:21];
41.                     2'b10: store <= instcode[19:16];
42.                     2'b11: store <= ALU_Out[3:0];
43.                 endcase
44.             end
45.             4'b0111:begin
46.                 case(display_mode)
47.                     2'b00: store<= CurPC[7:4];
48.                     2'b01: store <= {3'b000,instcode[25]};
49.                     2'b10: store <= {3'b000,instcode[20]};
50.                     2'b11: store <= ALU_Out[7:4];
51.                 endcase
52.             end
53.         endcase
54.     end
55. endmodule
```

CPU.v

CPU的各个功能模块的顶层文件，用于连接各个功能模块。

```
1.  `timescale 1ns / 1ps
2.  module CPU(
3.      input CLK,
4.      input Reset,
5.      output[31:0] CurPC, nextAddr, instcode, Reg1Out, Reg2Out, ALU_Out, Write
    Data
6.  );
7.      wire ExtSel, PCWre, InsMemRW, RegDst, RegWre, ALUSrcA, ALUSrcB, RD, WR,
    DBDataSrc, zero, sign;
8.      wire[2:0] ALUOp;
9.      wire[1:0] PCSrc;
10.     wire[4:0] WriteRegAddr;//可添加至 output 辅助仿真
11.     wire[31:0] MemOut, Ext_Imm, PC4, InsSrcImm, InsSrc1, InsSrc2, ALU_Input_
    A, ALU_Input_B;//ALU 两个输入可以添加到本模块输出用以检查仿真
12.
13.     PC pc(CLK, Reset, PCWre, nextAddr, CurPC);
14.     InsRegister insreg(CurPC, InsMemRW, instcode);
15.     Adder pc4_adder(CurPC, 32'b00000000000000000000000000000100, PC4);
16.     Mux_2 reg_w_choose(RegDst, instcode[20:16],instcode[15:11], WriteRegAddr
    );
17.     RegisterFile regfile(RegWre, CLK, instcode[25:21], instcode[20:16], Writ
    eRegAddr, WriteData, Reg1Out, Reg2Out);
18.     Extend extend(ExtSel, instcode[15:0], Ext_Imm);
19.     Mux_2 alua_choose(ALUSrcA, Reg1Out, {27'b000000000000000000000000000,ins
    tcode[10:6]}, ALU_Input_A);
20.     Mux_2 alub_choose(ALUSrcB, Reg2Out, Ext_Imm, ALU_Input_B);
21.     ALU alu(ALUOp, ALU_Input_A, ALU_Input_B, ALU_Out, zero, sign);
22.     DataMem datamem(CLK, RD, WR, ALU_Out, Reg2Out, MemOut);
23.     Mux_2 db_choose(DBDataSrc, ALU_Out, MemOut, WriteData);
24.     LeftShift_2 after_imm_extend(Ext_Imm, InsSrcImm);
25.     LeftShift_2 addr_shift({2'b00, PC4[31:28], instcode[25:0]}, InsSrc2);
26.     Adder next_pc1(InsSrcImm, PC4, InsSrc1);
27.     Mux_3 nextpc_choose(PCSrc, PC4, InsSrc1, InsSrc2, nextAddr);
28.     ControlUnit control_unit(instcode[31:26], zero, sign, ExtSel, PCWre, Ins
    MemRW, RegDst, RegWre, ALUSrcA, ALUSrcB, PCSrc, ALUOp, RD, WR, DBDataSrc);
29. endmodule
```

PC.v

CLK 上升沿触发，更改指令地址。由于指令地址存储在寄存器里，一开始需要赋 currentAddress 为 0。Reset 是重置信号，当为 1 时，指令寄存器地址重置。PCWre 的作用为保留现场，如果 PCWre 为 0，指令地址不变。
PC 程序计数器用于存放当前指令的地址，当 PC 的值发生改变的时候，CPU 会根据程序计数器 PC 中新得到的指令地址，从指令存储器中取出对应地址的指令。在单周期 CPU 的运行周期中，PC 值的变化是最先的，而且是根据 PCSrc 控制信号的值选择指令地址是要进行 PC+4 或者跳转等操作。若 PC 程序计数器检测到 Reset 输入信号为 0 时，则对程序计数器存储的当前指令地址进行清零处理。

```
1.  `timescale 1ns / 1ps
2.  module PC(
3.      input CLK, Reset, PCWre,
4.      input [31:0] newAddr,
5.      output reg [31:0] PCAddr
6.      );
7.      initial begin
8.          PCAddr = 0;
9.      end
10.     always@(posedge CLK or negedge Reset) begin
11.         if(Reset==0) PCAddr = 0;
12.         else if(PCWre) PCAddr = newAddr;
13.     end
14. endmodule
```

InsRegister.v

该部分为指令寄存器，通过一个 256 大小的 8 位寄存器数组来保存从文件输入的全部指令。然后通过输入的地址，找到相应的指令，输出到 IDataOut。 指令存储器的功能是存储读入的所有 32-bit 位宽的指令，根据程序计数器 PC 中的指令地址进行取指令操作并对指令类型进行分析，通过指令类型对所取指令的各字段进行区分识别，最后将对应部分传递给其他模块进行后续处理。 指令存储器中每个单元的位宽为 8-bit，也就是存储每条 32-bit 位宽的指令都需要占据 4 个单元，所以第 n（n 大于或等于 0）条指令所对应的起始地址为 4n，且占据第 4n，4n+1，4n+2，4n+3 这四个单元。取出指令就是将这四个单元分别取出，因为指令的存储服从高位指令存储在低位地址的规则，所以 4n 单元中的字段是该条指令的最高 8 位，后面以此类推，并通过左移操作将指令的四个单元部分移动到相对应的位置，以此来得到所存指令。

```
1.  `timescale 1ns / 1ps
2.
3.  module InsRegister(
4.      input [31:0] IAddr,
5.      input RW,
6.      output reg [31:0] ins
```

```
    );
    reg[7:0] mem[255:0];
    initial begin
        $readmemb("D:/instruction.txt", mem);
    end

    always@(IAddr or RW) begin
        if (RW) ins = {mem[IAddr], mem[IAddr + 1], mem[IAddr + 2], mem[IAddr + 3]};
    end

endmodule
```

Adder.v

该模块用于计算下一个 PC 值

```
`timescale 1ns / 1ps

module Adder(
    input[31:0] old1,
    input[31:0] old2,
    output[31:0] res//reg?address?
    );
    assign res = old1 + old2;
endmodule
```

Mux_2.v

该模块共有 4 个位置被使用到，分别是用于 Write reg 信号选择、aluA 数据选择、aluB 数据选择、以及 DB 数据选择。

```
`timescale 1ns / 1ps

module Mux_2(
    input Select,
    input[31:0] in1,
    input[31:0] in2,
    output[31:0] out
    );
    assign out = Select ? in2 : in1;
endmodule
```

RegisterFile.v

该部分为寄存器读写单元，储存寄存器组，并根据地址对寄存器组进行读写。WE 的作用是控制寄存器是否写入。同上，通过一个 32 大小的 32 位寄存器数组来模拟寄存器，开始时全部置 0。通过访问寄存器的地址，来获取寄存器里面的值，并进行操作。（由于 $0 恒为 0，所以写入寄存器的地址不能为 0） 寄存器组中的每个寄存器位宽 32-bit,是存放 ALU 计算所需要的临时数据的,与数据存储器不同，可能会在程序执行的过程中被多次覆盖，而数据存储器内的数据一般只有 sw 指令才能进行修改覆盖。寄存器组会根据操作码 opCode 与 rs，rt 字段相应的地址读取数据,同时将 rs，rt 寄存器的地址和其中的数据输出,在 CLK 的下降沿到来时将数据存放到 rd 或者 rt 字段的相应地址的寄存器内。

```
1.  `timescale 1ns / 1ps
2.
3.  module RegisterFile(
4.      input WE,
5.      input CLK,
6.      input[4:0] ReadReg1,
7.      input[4:0] ReadReg2,
8.      input[4:0] WriteReg,
9.      input[31:0] WriteData,
10.     output[31:0] ReadData1,
11.     output[31:0] ReadData2
12.     );
13.     reg[31:0] registers[0:31];
14.     integer i;
15.     initial begin
16.         for (i = 0; i < 32; i = i + 1) registers[i] <= 0;
17.     end
18.     assign ReadData1 = ReadReg1 ? registers[ReadReg1] : 0;
19.     assign ReadData2 = ReadReg2 ? registers[ReadReg2] : 0;
20.     always@(negedge CLK) begin
21.         if (WE && WriteReg) registers[WriteReg] = WriteData;
22.     end
23. endmodule
```

Extend.v

该模块用于立即数的符号位扩展，当 ExtSel 信号为 0 时做 0 扩展，为 1 时做 1 扩展

```
1.  `timescale 1ns / 1ps
2.
3.  module Extend(
4.      input ExtSel,
5.      input [15:0] immed,
6.      output [31:0] extendImmed
7.      );
```

```
    assign extendImmed = {ExtSel && immed[15] ? 16'hffff : 16'h0000, immed};

endmodule
```

ALU.v

该部分为算术逻辑单元，用于逻辑指令计算和跳转指令比较。ALUOp 用于控制算数的类型，A、B 为输入数，result 为运算结果，zero、sign 主要用于 beq、bne、bltz 等指令的判断。

ALU 算术逻辑单元的功能是根据控制信号从输入的数据中选取对应的操作数，根据操作码进行运算并输出结果与零标志位。具体的操作逻辑见***表 2***

```
`timescale 1ns / 1ps

module ALU(
    input [2:0] ALUopcode,
    input [31:0] rega,
    input [31:0] regb,
    output reg [31:0] result,
    output zero,
    output sign
    );

    assign zero = (result == 0) ? 1 : 0;
    assign sign = result[31];
    always @( ALUopcode or rega or regb ) begin
        case (ALUopcode)
            3'b000 : result = rega + regb;
            3'b001 : result = rega - regb;
            3'b010 : result = regb << rega;
            3'b011 : result = rega | regb;
            3'b100 : result = rega & regb;
            3'b101 : result = (rega < regb)?1:0; // 不带符号比较
            3'b110 : begin // 带符号比较
                        if (rega<regb &&(( rega[31] == 0 && regb[31]==0) || (rega[31] == 1 && regb[31]==1))) result = 1;
                        else if (rega[31] == 0 && regb[31]==1) result = 0;
                        else if ( rega[31] == 1 && regb[31]==0) result = 1;

                        else result = 0;
                    end
            3'b111 : result = rega ^ regb;    //异或
            default : result = 8'h00000000;
        endcase
    end
endmodule
```

DataMem.v

该部分控制内存存储，用于内存存储、读写。用 255 大小的 8 位寄存器数组模拟内存，采用小端模式。DataMenRW 控制内存读写。由于指令为真实地址，所以不需要左移 2 位。

```
1.  `timescale 1ns / 1ps
2.
3.  module DataMem(
4.      input CLK,
5.      input mRD,
6.      input mWR,
7.      input[31:0] DAddr,
8.      input[31:0] DataIn,
9.      output reg[31:0] DataOut
10.     );
11.
12.     reg[7:0] dataMemory[255:0];
13.     integer i;
14.     initial begin
15.         for (i = 0; i < 256; i = i + 1) dataMemory[i] <= 0;//没有自增运算符
16.     end
17.
18.     always@(mRD or DAddr) begin
19.         if (mRD) begin
20.             DataOut[31:24] <= dataMemory[DAddr];
21.             DataOut[23:16] <= dataMemory[DAddr+1];
22.             DataOut[15:8] <= dataMemory[DAddr+2];
23.             DataOut[7:0] <= dataMemory[DAddr+3];
24.         end
25.     end
26.
27.     always@(negedge CLK) begin
28.         if (mWR) begin
29.             dataMemory[DAddr] <= DataIn[31:24];
30.             dataMemory[DAddr+1] <= DataIn[23:16];
31.             dataMemory[DAddr+2] <= DataIn[15:8];
32.             dataMemory[DAddr+3] <= DataIn[7:0];
33.         end
34.     end
35. endmodule
```

LeftShift.v

该模块共用于两个位置，分别是 j 类指令的立即数左移运算、I 类指令的立即数左移运算。

```
1.  `timescale 1ns / 1ps
2.
3.  module LeftShift_2(
4.      input[31:0] in,
5.      output[31:0] out
6.      );
7.      assign out = in << 2;
8.  endmodule
```

ControlUnit.v

控制单元通过输入的 zero 零标志位与当前指令中对应的指令部分来确定当前整个 CPU 程序中各模块的工作和协作情况，根据 CPU 运行逻辑，事先对整个 CPU 中控制信号的控制，以此来达到指挥各个模块协同工作的目的。

控制单元的各个控制信号逻辑表达式是根据***表 4*** 得到的。

```
1.  `timescale 1ns / 1ps
2.
3.  module ControlUnit(
4.      input[5:0] op,
5.      input zero,
6.      input sign,
7.      output ExtSel,
8.      output PCWre,
9.      output InsMemRW,
10.     output RegDst,
11.     output RegWre,
12.     output ALUSrcA,
13.     output ALUSrcB,
14.     output[1:0] PCSrc,
15.     output[2:0] ALUOp,
16.     output mRD,
17.     output mWR,
18.     output DBDataSrc
19.     );
20.     //对指令和 ALU 的操作码定义常量
21.     parameter INS_ADD = 6'b000000;
22.     parameter INS_SUB = 6'b000001;
23.     parameter INS_ADDIU = 6'b000010;
24.     parameter INS_ANDI = 6'b010000;
25.     parameter INS_AND = 6'b010001;
26.     parameter INS_ORI = 6'b010010;
27.     parameter INS_OR = 6'b010011;
28.     parameter INS_SLL = 6'b011000;
29.     parameter INS_SLTI = 6'b011100;
```

```
30.     parameter INS_SW = 6'b100110;
31.     parameter INS_LW = 6'b100111;
32.     parameter INS_BEQ = 6'b110000;
33.     parameter INS_BNE = 6'b110001;
34.     parameter INS_BLTZ = 6'b110010;
35.     parameter INS_J = 6'b111000;
36.     parameter INS_HALT = 6'b111111;
37. //    parameter ALU_ADD = 3'b000;
38. //    parameter ALU_SUB = 3'b001;
39. //    parameter ALU_SLL = 3'b010;
40. //    parameter ALU_OR = 3'b011;
41. //    parameter ALU_AND = 3'b100;
42. //    parameter ALU_SLTU = 3'b101;
43. //    parameter ALU_SLT = 3'b110;
44. //    parameter ALU_XOR = 3'b111;
45.     assign PCWre = op != INS_HALT;
46.     assign ALUSrcA = op == INS_SLL;
47.     assign ALUSrcB = op == INS_ADDIU || op == INS_ANDI|| op == INS_ORI || op
     == INS_SLTI || op == INS_SW || op == INS_LW;
48.     assign DBDataSrc = op == INS_LW;
49.     assign RegWre = op != INS_BEQ && op != INS_BNE && op != INS_BLTZ && op !
    = INS_SW && op != INS_HALT;
50.     assign InsMemRW = 1;
51.     assign mRD = op == INS_LW;
52.     assign mWR = op == INS_SW;
53.     assign RegDst = op != INS_ADDIU && op != INS_ANDI && op != INS_ORI && op
     != INS_SLTI && op != INS_LW;
54.     assign ExtSel = op != INS_ANDI && op != INS_ORI;
55.     assign PCSrc[0] = op == INS_BEQ && zero == 1 || op == INS_BNE && zero ==
     0 || op == INS_BLTZ && sign == 1;
56.     assign PCSrc[1] = op == INS_J;
57.     assign ALUOp[2] = (op == INS_AND || op == INS_SLTI || op == INS_ANDI ||
    op == INS_BEQ || op == INS_BNE) ? 1 : 0;
58.     assign ALUOp[1] = (op == INS_SLL || op == INS_OR || op == INS_SLTI || op
     == INS_ORI || op == INS_BEQ || op == INS_BNE) ? 1 : 0;
59.     assign ALUOp[0] = (op == INS_SUB || op == INS_OR || op == INS_ORI|| op==
     INS_BEQ || op == INS_BNE) ? 1 : 0;
60. endmodule
```

3、 验证所实现的CPU的正确性

首先，将测试用的指令读入，然后进行仿真检验。

测试指令转换如下表：

地址	汇编程序	指令代码					
		op（6）	**rs(5)**	**rt(5)**	**rd(5)/immediate (16)**	**16 进制数代码**	
0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000	=	00411800
0x0000000c	sub $5,$3,$2	000001	00011	00010	0010 1000 0000 0000	=	04622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000	=	44a22000
0x00000014	or $8,$4,$2	010011	00100	00010	0100 0000 0000 0000	=	4c824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040
0x0000001c	**bne $8,$1,-2 (≠,转 18)**	110001	01000	00001	1111 1111 1111 1110	=	C501fffe
0x00000020	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100	=	70460004
0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000	=	70c70000
0x00000028	addiu $7,$7,8	000010	00111	00111	0000 0000 0000 1000	=	08e70008
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	addiu $10,$0,-2	000010	00000	01010	1111 1111 1111 1110	=	080afffe
0x0000003c	addiu $10,$10,1	000010	01010	01010	0000 0000 0000 0001	=	094a0001
0x00000040	**bltz $10,-2(<0,转 3C)**	110010	01010	00000	1111 1111 1111 1110	=	C940fffe
0x00000044	**andi $11,$2,2**	010000	00010	01011	0000 0000 0000 0010	=	404b0002
0x00000048	**j 0x00000050**	111000	00000	00000	0000 0000 0001 0100	=	E0000014
0x0000004c	or $8,$4,$2	010011	00100	00010	0010 0100 0000 0000	=	4c824000
0x00000050	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000

表5 测试指令转换表

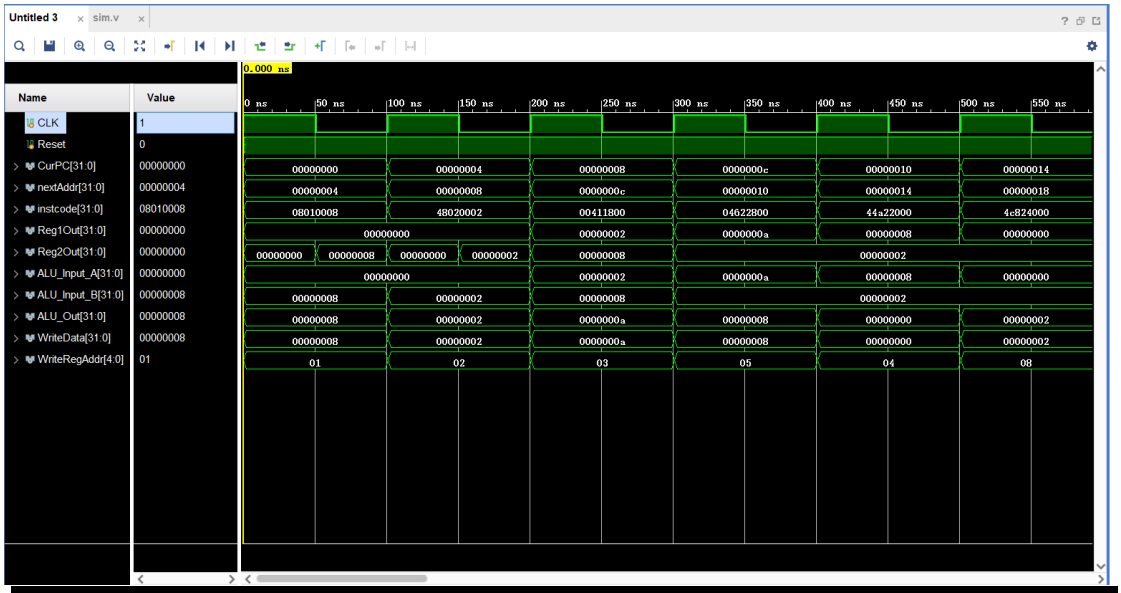
sim.v

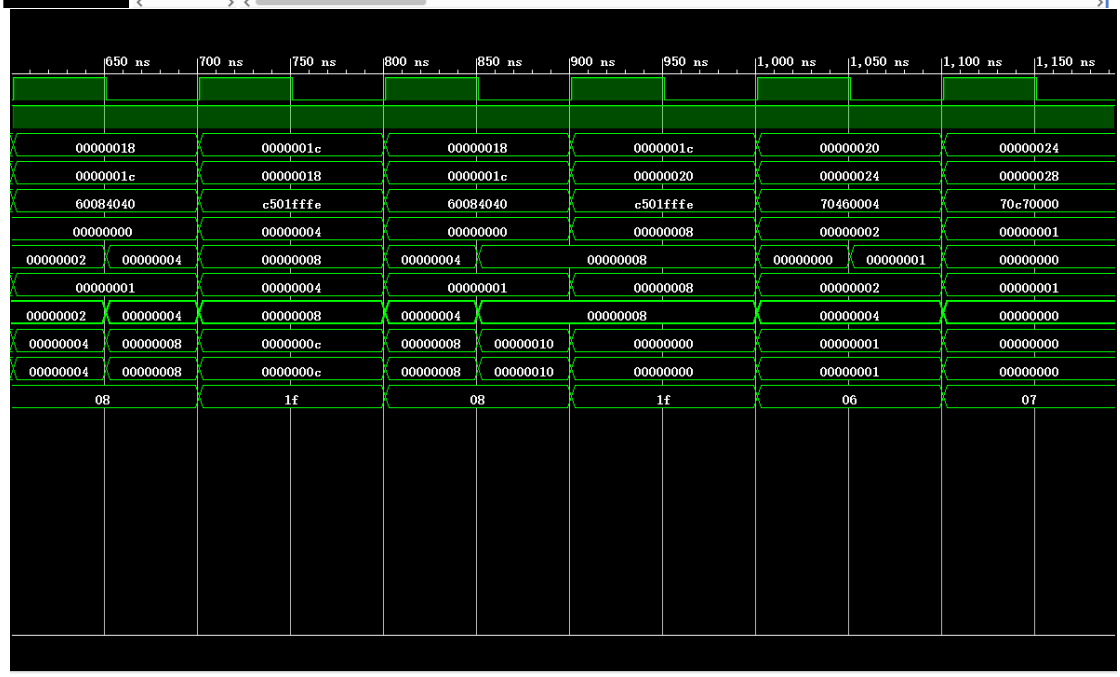
仿真模块

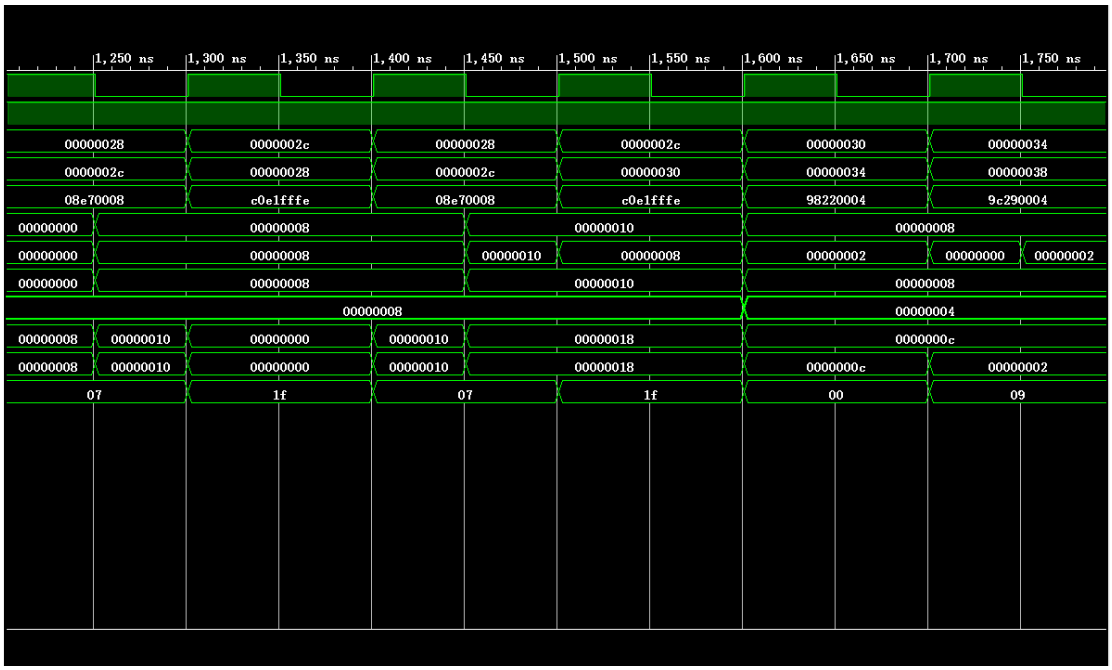
```
1.  `timescale 1ns / 1ps
2.
3.  module sim;
4.      reg CLK;
5.      reg Reset;
6.      wire[31:0] CurPC, nextAddr, instcode, Reg1Out, Reg2Out,ALU_Input_A, ALU_
    Input_B,  ALU_Out, WriteData;
7.      wire[4:0] WriteRegAddr;
```

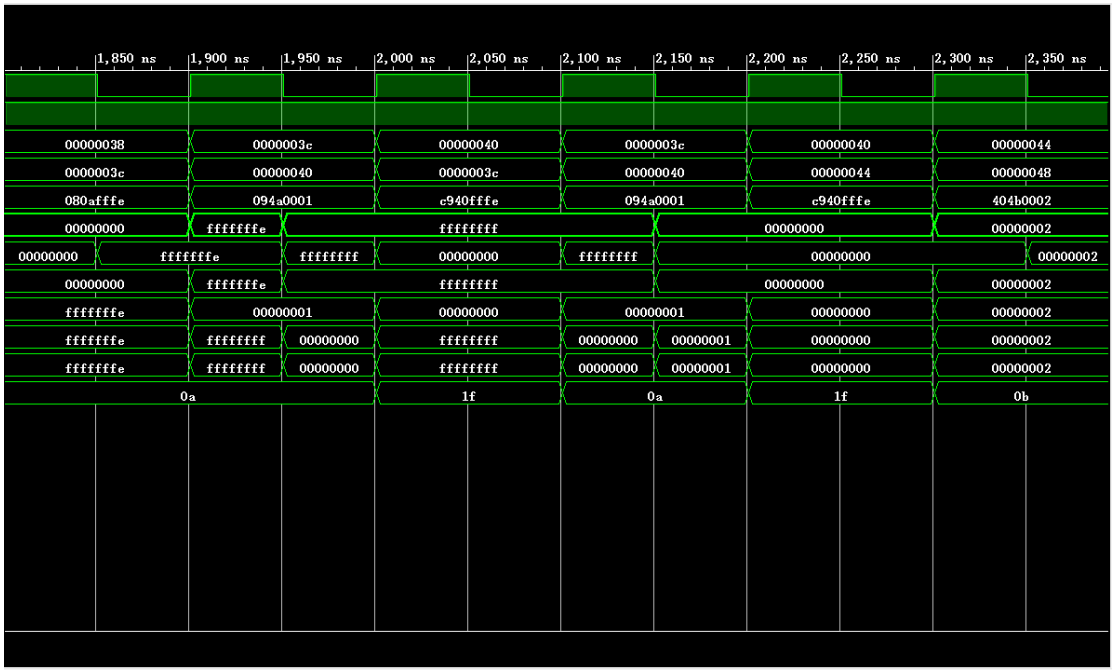
```
8.      CPU singleCPU(CLK, Reset, CurPC, nextAddr, instcode, Reg1Out, Reg2Out, A
    LU_Input_A, ALU_Input_B, ALU_Out, WriteData, WriteRegAddr);
9.      initial begin
10.         CLK = 1;
11.         Reset = 0;
12.         #1;
13.         Reset = 1;
14.         forever #50 CLK = !CLK;
15.     end
16. endmodule
```

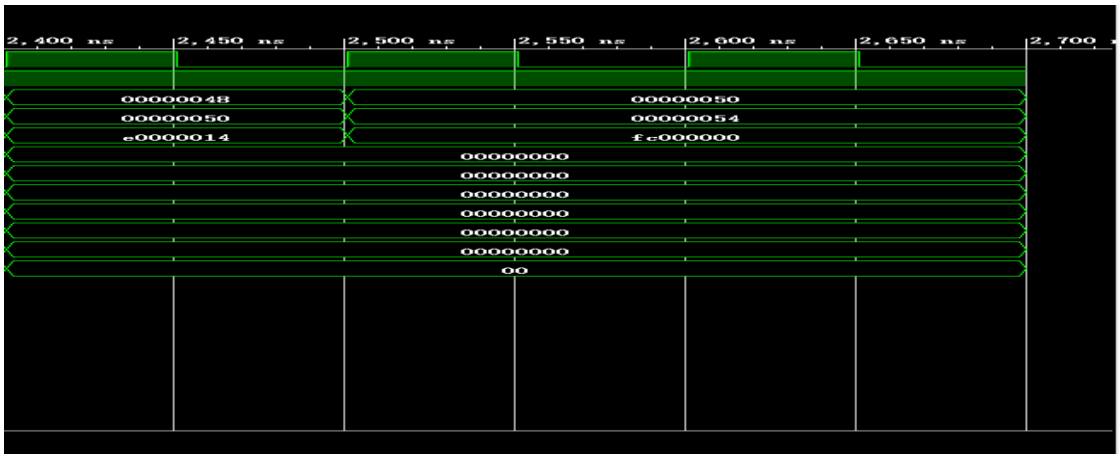
得到仿真波形如下图所示：





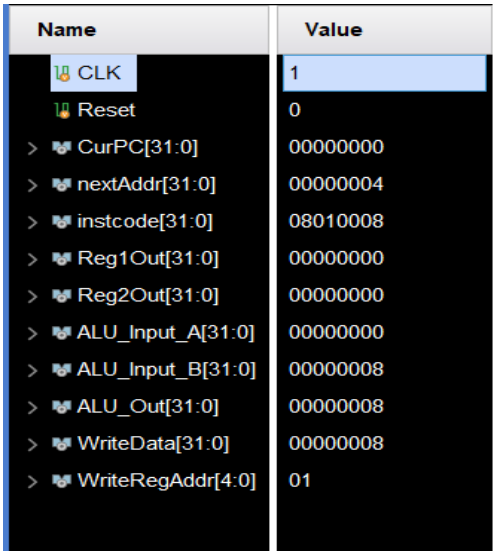

1,250 ns
1,300 ns
1,350 ns
1,400 ns
1,450 ns
1,500 ns
1,550 ns
1,600 ns
1,650 ns
1,700 ns
1,750 ns
00000028 0000002c 00000028 0000002c 00000030 00000034
0000002c 00000028 0000002c 00000030 00000034 00000038
08e70008 c0e1fffe 08e70008 c0e1fffe 98220004 9c290004
00000000 00000008 00000010 00000008
00000000 00000008 00000010 00000008 00000002 00000000 00000002
00000000 00000008 00000010 00000008
00000008 00000004
00000008 00000010 00000000 00000010 00000018 0000000c
00000008 00000010 00000000 00000010 00000018 0000000c 00000002
07 1f 07 1f 00 09


1,850 ns
1,900 ns
1,950 ns
2,000 ns
2,050 ns
2,100 ns
2,150 ns
2,200 ns
2,250 ns
2,300 ns
2,350 ns
00000038 0000003c 00000040 0000003c 00000040 00000044
0000003c 00000040 0000003c 00000040 00000044 00000048
080afffe 094a0001 c940fffe 094a0001 c940fffe 404b0002
00000000 fffffffe ffffffff 00000000 00000002
00000000 fffffffe ffffffff 00000000 ffffffff 00000000 00000002
00000000 fffffffe ffffffff 00000000 00000002
fffffffe 00000001 00000000 00000001 00000000 00000002
fffffffe ffffffff 00000000 ffffffff 00000000 00000001 00000000 00000002
fffffffe ffffffff 00000000 ffffffff 00000000 00000001 00000000 00000002
0a 1f 0a 1f 0b


2,400 ns
2,450 ns
2,500 ns
2,550 ns
2,600 ns
2,650 ns
2,700
00000048 00000050
00000050 00000054
e0000014 fc000000
00000000
00000000
00000000
00000000
00000000
00000000
00

以下所有指令均来自***表5***，指令存放的寄存器为WriteRegAddr[4:0],存储的数据为WriteData, nextAddr为即将执行的地址。

addiu　$1,$0,8



在左侧图中，CurPC表示当前PC值为00；

nextAdder表示下一条指令地址为04；

rs寄存器为00，寄存器值为00；

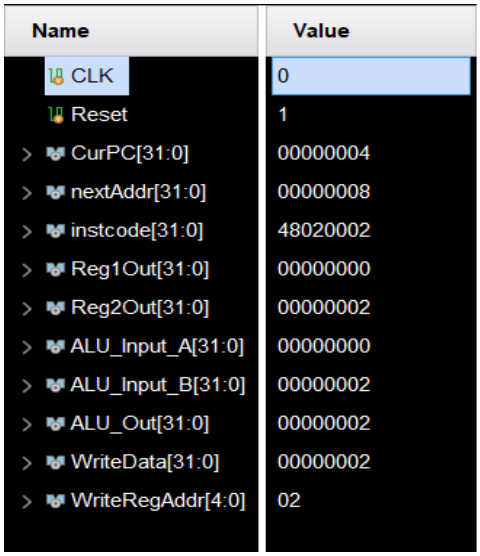
rt寄存器为01，寄存器值为08；

ALU运算结果为08，DB数据为08；

写回寄存器为01.

Rt <= rs + (符号位扩展)imm

ori　$2,$0,2



在左侧图中，CurPC表示当前PC值为04；

nextAdder表示下一条指令地址为08；

rs寄存器为00，寄存器值为00；

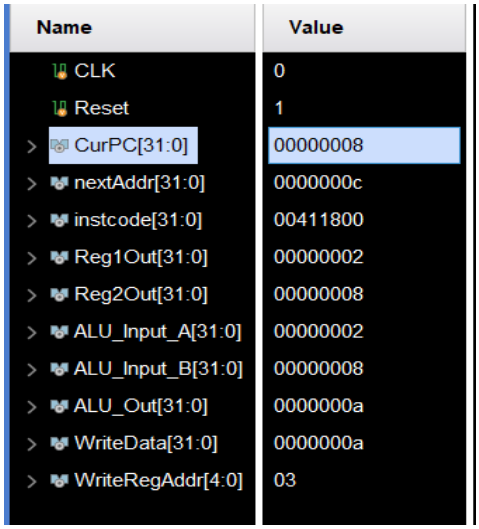
rt寄存器为02，寄存器值为02；

ALU运算结果为02，DB数据为02；

写回寄存器为02.

Rt <= rs + (符号位扩展)imm

add $3,$2,$1



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000008
nextAddr[31:0]	0000000c
instcode[31:0]	00411800
Reg1Out[31:0]	00000002
Reg2Out[31:0]	00000008
ALU_Input_A[31:0]	00000002
ALU_Input_B[31:0]	00000008
ALU_Out[31:0]	0000000a
WriteData[31:0]	0000000a
WriteRegAddr[4:0]	03

在左侧图中，CurPC表示当前PC值为08；

nextAdder表示下一条指令地址为0c；

rs寄存器为02，寄存器值为02；

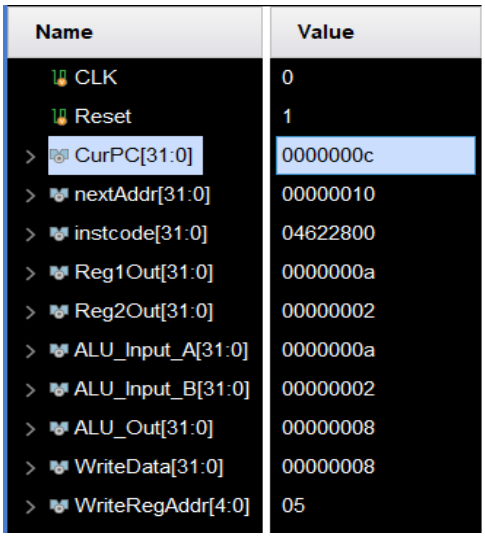
rt寄存器为01，寄存器值为08；

ALU运算结果为0a，DB数据为0a；

写回寄存器为03.

Rt <= rs + rd

sub $5,$3,$2



Name	Value
CLK	0
Reset	1
CurPC[31:0]	0000000c
nextAddr[31:0]	00000010
instcode[31:0]	04622800
Reg1Out[31:0]	0000000a
Reg2Out[31:0]	00000002
ALU_Input_A[31:0]	0000000a
ALU_Input_B[31:0]	00000002
ALU_Out[31:0]	00000008
WriteData[31:0]	00000008
WriteRegAddr[4:0]	05

在左侧图中，CurPC表示当前PC值为0c；

nextAdder表示下一条指令地址为10；

rs寄存器为02，寄存器值为02；

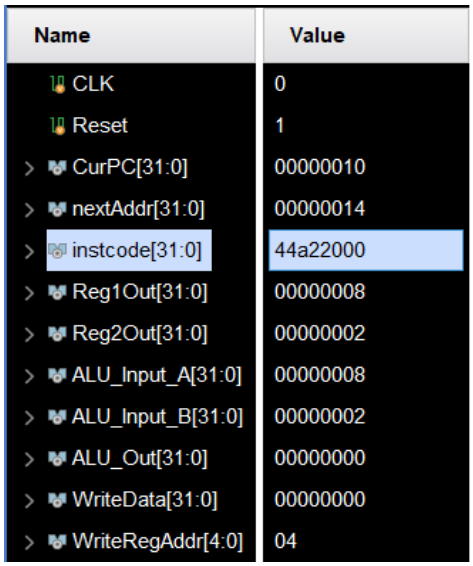
rt寄存器为01，寄存器值为08；

ALU运算结果为0a，DB数据为0a；

写回寄存器为03.

Rt <= rs + rd

and $4,$5,$2



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000010
nextAddr[31:0]	00000014
instcode[31:0]	44a22000
Reg1Out[31:0]	00000008
Reg2Out[31:0]	00000002
ALU_Input_A[31:0]	00000008
ALU_Input_B[31:0]	00000002
ALU_Out[31:0]	00000000
WriteData[31:0]	00000000
WriteRegAddr[4:0]	04

左侧图中，CurPC表示当前PC值为10；

nextAdder表示下一条指令地址为14；

rs寄存器为05，寄存器值为08；

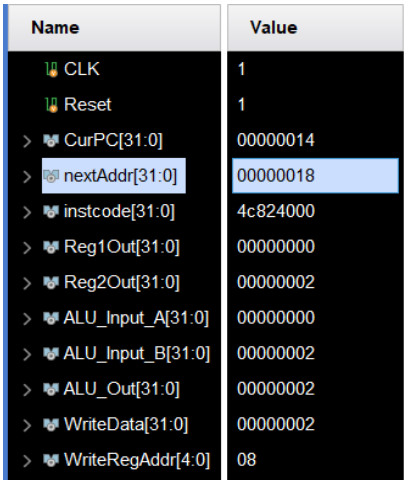
rt寄存器为02，寄存器值为02；

ALU运算结果为00，DB数据为00；

写回寄存器为04.

Rt <= rs + rd

or $8,$4,$2



在左侧图中，CurPC表示当前PC值为14；

nextAdder表示下一条指令地址为18；

rs寄存器为04，寄存器值为00；

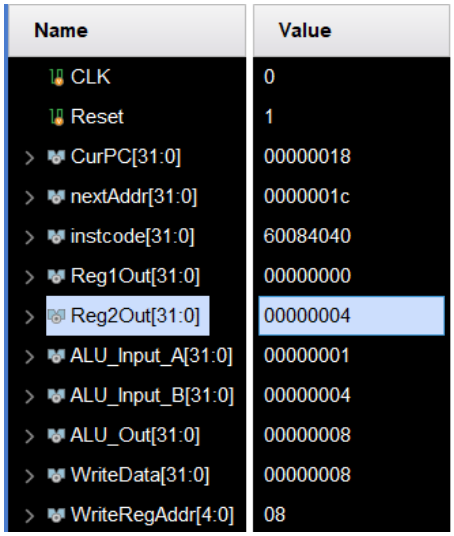
rt寄存器为02，寄存器值为02；

ALU运算结果为02，DB数据为02；

写回寄存器为08.

Rt <= rs + rd

sll $8,$8,1



在左侧图中，CurPC表示当前PC值为18；

nextAdder表示下一条指令地址为1c；

rd寄存器为08，寄存器值为08；

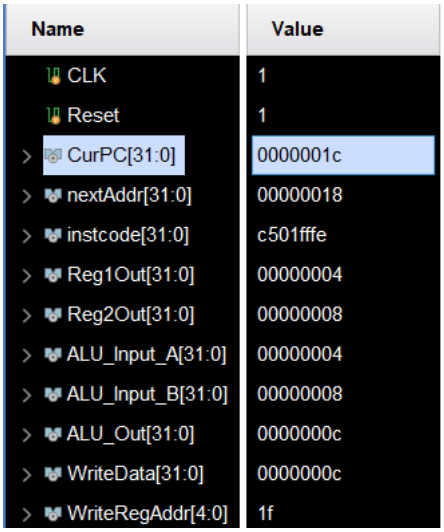
rt寄存器为08，寄存器值为04；

ALU运算结果为08，DB数据为08；

写回寄存器为08.

Rd <= rt << (符号位扩展)sa

bne $8,$1,-2



在左侧图中，CurPC表示当前PC值为1c；

nextAdder表示下一条指令地址为18；

rs寄存器为08，寄存器值为08；

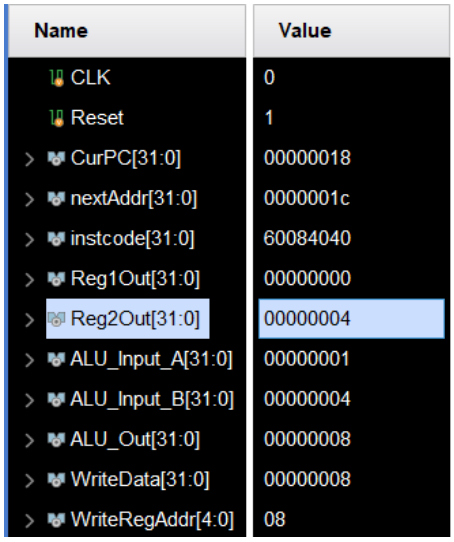
rt寄存器为01，寄存器值为04；

ALU运算结果为0c，DB数据为0c；

写回寄存器为1f.

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

sll　$8,$8,1



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000018
nextAddr[31:0]	0000001c
instcode[31:0]	60084040
Reg1Out[31:0]	00000000
Reg2Out[31:0]	00000004
ALU_Input_A[31:0]	00000001
ALU_Input_B[31:0]	00000004
ALU_Out[31:0]	00000008
WriteData[31:0]	00000008
WriteRegAddr[4:0]	08

在左侧图中，CurPC表示当前PC值为18；

nextAdder表示下一条指令地址为1c；

rd寄存器为08，寄存器值为08；

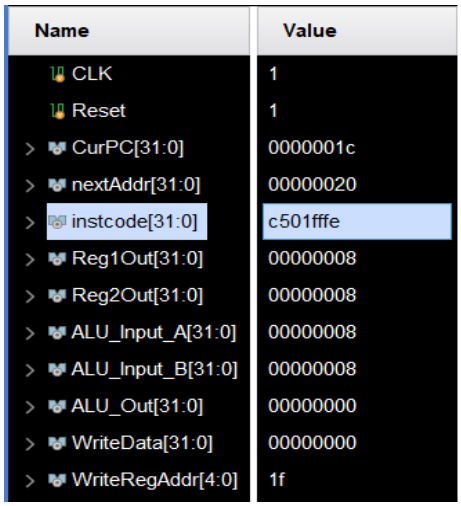
rt寄存器为08，寄存器值为04；

ALU运算结果为08，DB数据为08；

写回寄存器为08.

Rd <= rt　<<　(符号位扩展)sa

bne $8,$1,-2



Name	Value
CLK	1
Reset	1
CurPC[31:0]	0000001c
nextAddr[31:0]	00000020
instcode[31:0]	c501fffe
Reg1Out[31:0]	00000008
Reg2Out[31:0]	00000008
ALU_Input_A[31:0]	00000008
ALU_Input_B[31:0]	00000008
ALU_Out[31:0]	00000000
WriteData[31:0]	00000000
WriteRegAddr[4:0]	1f

在左侧图中，CurPC表示当前PC值为1c；

nextAdder表示下一条指令地址为20；

rs寄存器为08，寄存器值为08；

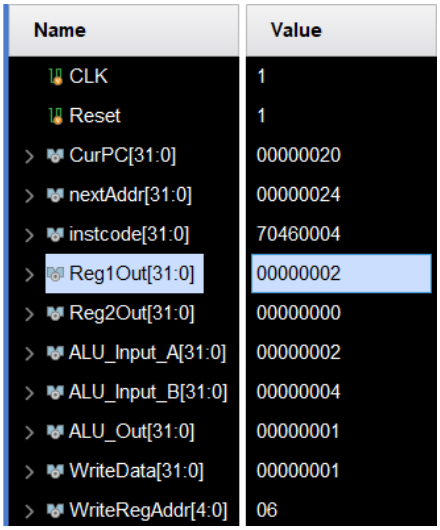
rt寄存器为01，寄存器值为08；

ALU运算结果为00，DB数据为00；

写回寄存器为1f.

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

slti　$6,$2,4



Name	Value
CLK	1
Reset	1
CurPC[31:0]	00000020
nextAddr[31:0]	00000024
instcode[31:0]	70460004
Reg1Out[31:0]	00000002
Reg2Out[31:0]	00000000
ALU_Input_A[31:0]	00000002
ALU_Input_B[31:0]	00000004
ALU_Out[31:0]	00000001
WriteData[31:0]	00000001
WriteRegAddr[4:0]	06

在左侧图中，CurPC表示当前PC值为20；

nextAdder表示下一条指令地址为24；

rs寄存器为02，寄存器值为02；

sa为04；

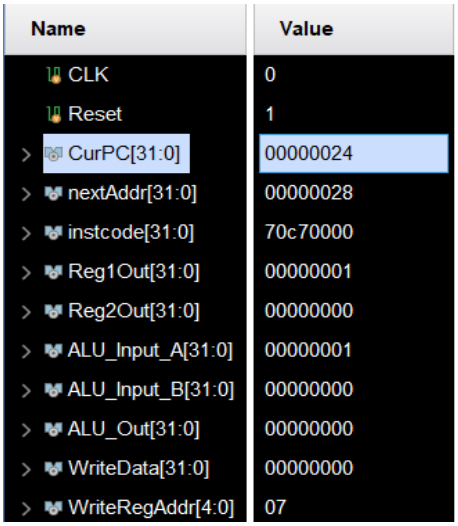
ALU运算结果为01，DB数据为01；

写回寄存器为06.

if (rs< (sign-extend)**immediate**)　rt =1

else　rt=0,

slti $7,$6,0



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000024
nextAddr[31:0]	00000028
instcode[31:0]	70c70000
Reg1Out[31:0]	00000001
Reg2Out[31:0]	00000000
ALU_Input_A[31:0]	00000001
ALU_Input_B[31:0]	00000000
ALU_Out[31:0]	00000000
WriteData[31:0]	00000000
WriteRegAddr[4:0]	07

在左侧图中，CurPC表示当前PC值为20；

nextAdder表示下一条指令地址为24；

rs寄存器为06，寄存器值为01；

sa为00；

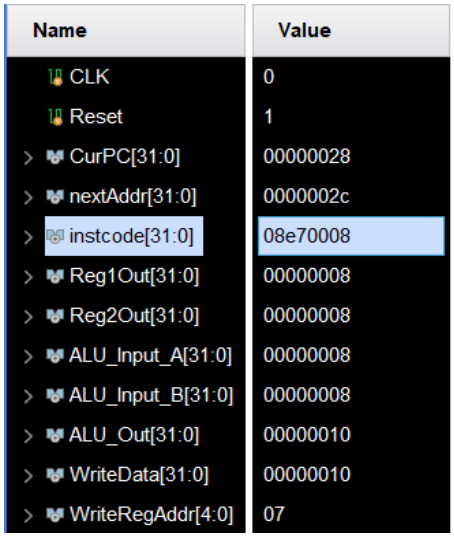
ALU运算结果为00，DB数据为00；

写回寄存器为07.

if (rs< (sign-extend)**immediate**)　rt =1

else　rt=0,

addiu　$7,$7,8



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000028
nextAddr[31:0]	0000002c
instcode[31:0]	08e70008
Reg1Out[31:0]	00000008
Reg2Out[31:0]	00000008
ALU_Input_A[31:0]	00000008
ALU_Input_B[31:0]	00000008
ALU_Out[31:0]	00000010
WriteData[31:0]	00000010
WriteRegAddr[4:0]	07

在左侧图中，CurPC表示当前PC值为28；

nextAdder表示下一条指令地址为2c；

rs寄存器为07，寄存器值为08；

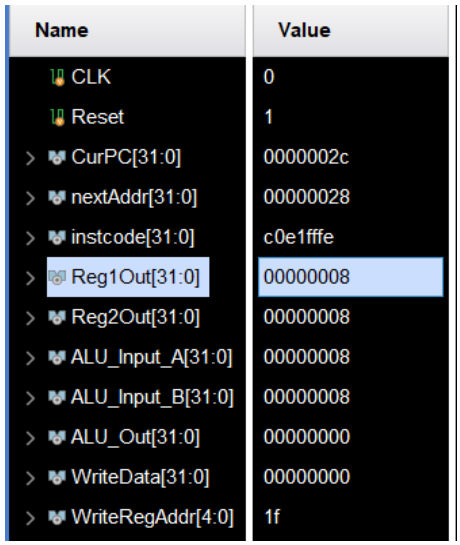
rt寄存器为07，寄存器值为08；

ALU运算结果为10，DB数据为10；

写回寄存器为07.

Rt <= rs + (符号位扩展)imm

beq $7,$1,-2



Name	Value
CLK	0
Reset	1
CurPC[31:0]	0000002c
nextAddr[31:0]	00000028
instcode[31:0]	c0e1fffe
Reg1Out[31:0]	00000008
Reg2Out[31:0]	00000008
ALU_Input_A[31:0]	00000008
ALU_Input_B[31:0]	00000008
ALU_Out[31:0]	00000000
WriteData[31:0]	00000000
WriteRegAddr[4:0]	1f

在左侧图中，CurPC表示当前PC值为2c；

nextAdder表示下一条指令地址为28；

rs寄存器为01，寄存器值为08；

rt寄存器为07，寄存器值为08；

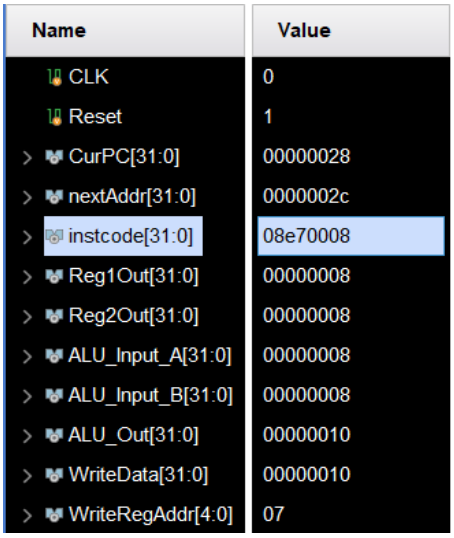
ALU运算结果为00，DB数据为00；

写回寄存器为1f.

if(rs==rt)PC<=PC+4+(sign-extend)im

mediate <<2　else pc ←pc + 4

addiu　$7,$7,8



在左侧图中，CurPC表示当前PC值为28；

nextAdder表示下一条指令地址为2c；

rs寄存器为07，寄存器值为08；

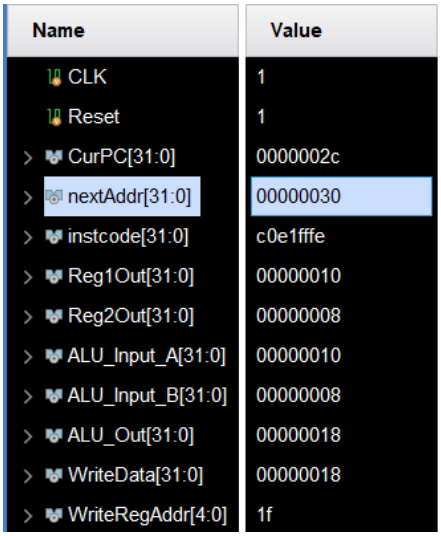
rt寄存器为07，寄存器值为08；

ALU运算结果为10，DB数据为10；

写回寄存器为07.

Rt <= rs + (符号位扩展)imm

beq $7,$1,-2



在左侧图中，CurPC表示当前PC值为2c；

nextAdder表示下一条指令地址为30；

rs寄存器为01，寄存器值为10；

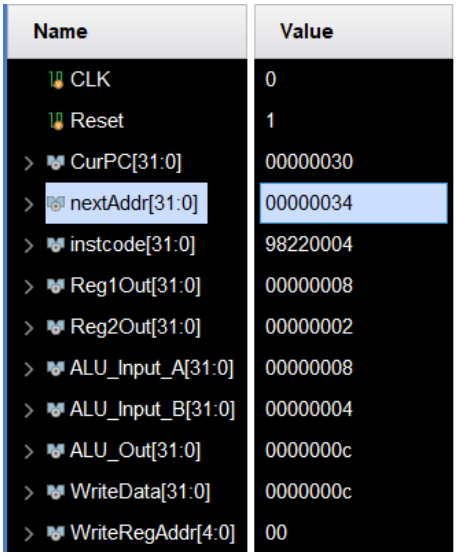
rt寄存器为07，寄存器值为08；

ALU运算结果为18，DB数据为18；

写回寄存器为1f.

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

sw　$2,4($1)



在左侧图中，CurPC表示当前PC值为30；

nextAdder表示下一条指令地址为34；

rs寄存器为01，寄存器值为08；

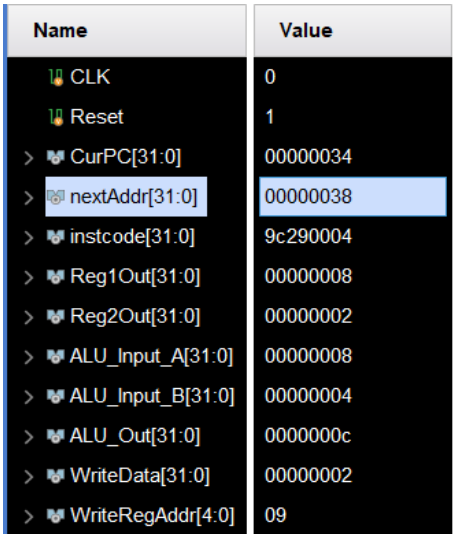
rt寄存器为02，寄存器值为02；

ALU运算结果为0c，DB数据为0c；

写回寄存器为1f.

memory[rs+(sign-extend)**immediate**] ←rt;

lw $9,4($1)



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000034
nextAddr[31:0]	00000038
instcode[31:0]	9c290004
Reg1Out[31:0]	00000008
Reg2Out[31:0]	00000002
ALU_Input_A[31:0]	00000008
ALU_Input_B[31:0]	00000004
ALU_Out[31:0]	0000000c
WriteData[31:0]	00000002
WriteRegAddr[4:0]	09

在左侧图中，CurPC表示当前PC值为34；

nextAdder表示下一条指令地址为38；

rs寄存器为01，寄存器值为08；

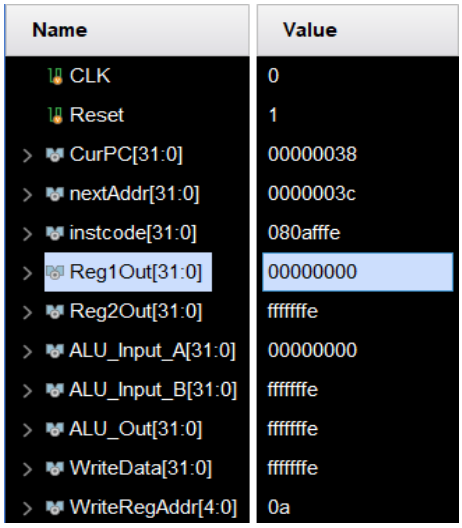
rt寄存器为09，寄存器值为02；

ALU运算结果为0c，DB数据为02；

写回寄存器为09.

rt ← memory[rs + (sign-extend)**immediate**]

addiu $10,$0,-2



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000038
nextAddr[31:0]	0000003c
instcode[31:0]	080afffe
Reg1Out[31:0]	00000000
Reg2Out[31:0]	fffffffe
ALU_Input_A[31:0]	00000000
ALU_Input_B[31:0]	fffffffe
ALU_Out[31:0]	fffffffe
WriteData[31:0]	fffffffe
WriteRegAddr[4:0]	0a

在左侧图中，CurPC表示当前PC值为38；

nextAdder表示下一条指令地址为3c；

rs寄存器为00，寄存器值为00；

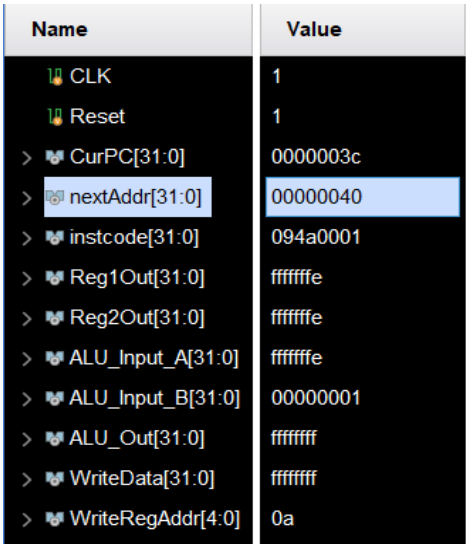
rt寄存器为10，寄存器值为-2；

ALU运算结果为-2，DB数据为-2；

写回寄存器为0a.

Rt <= rs + (符号位扩展)imm

addiu $10,$10,1



Name	Value
CLK	1
Reset	1
CurPC[31:0]	0000003c
nextAddr[31:0]	00000040
instcode[31:0]	094a0001
Reg1Out[31:0]	fffffffe
Reg2Out[31:0]	fffffffe
ALU_Input_A[31:0]	fffffffe
ALU_Input_B[31:0]	00000001
ALU_Out[31:0]	ffffffff
WriteData[31:0]	ffffffff
WriteRegAddr[4:0]	0a

在左侧图中，CurPC表示当前PC值为3c；

nextAdder表示下一条指令地址为40；

rs寄存器为00，寄存器值为00；

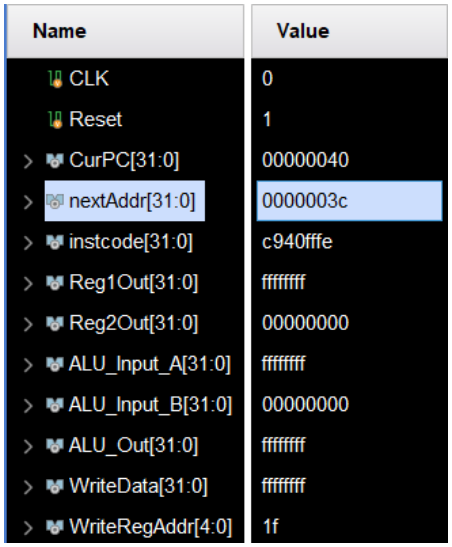
rt寄存器为10，寄存器值为-2；

ALU运算结果为-2，DB数据为-2；

写回寄存器为0a.

Rt <= rs + (符号位扩展)imm

bltz $10,-2



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000040
nextAddr[31:0]	0000003c
instcode[31:0]	c940fffe
Reg1Out[31:0]	ffffffff
Reg2Out[31:0]	00000000
ALU_Input_A[31:0]	ffffffff
ALU_Input_B[31:0]	00000000
ALU_Out[31:0]	ffffffff
WriteData[31:0]	ffffffff
WriteRegAddr[4:0]	1f

在左侧图中，CurPC表示当前PC值为40；

nextAdder表示下一条指令地址为3c；

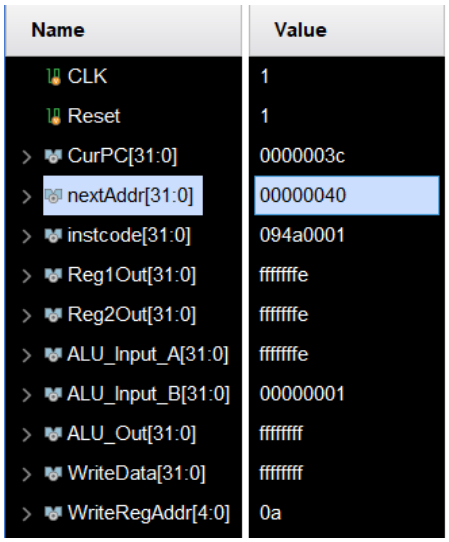
rs寄存器为10，寄存器值为-2；

ALU运算结果为-1，DB数据为-1；

写回寄存器为1f.

if(rs<$zero)　pc ←　pc　+　4　+ (sign-extend)**immediate** <<2　else pc ←pc + 4

addiu　$10,$10,1



Name	Value
CLK	1
Reset	1
CurPC[31:0]	0000003c
nextAddr[31:0]	00000040
instcode[31:0]	094a0001
Reg1Out[31:0]	fffffffe
Reg2Out[31:0]	fffffffe
ALU_Input_A[31:0]	fffffffe
ALU_Input_B[31:0]	00000001
ALU_Out[31:0]	ffffffff
WriteData[31:0]	ffffffff
WriteRegAddr[4:0]	0a

在左侧图中，CurPC表示当前PC值为3c；

nextAdder表示下一条指令地址为40；

rs寄存器为10，寄存器值为fffffffe；

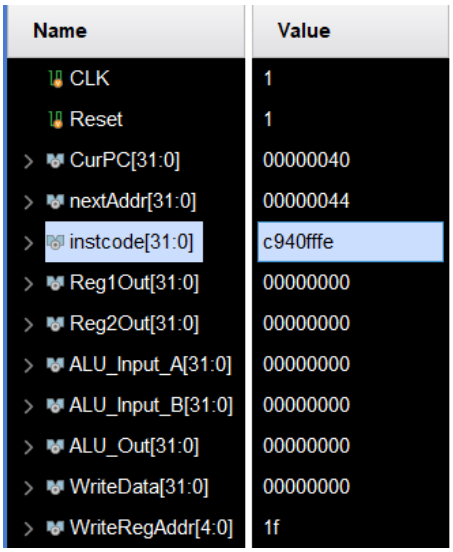
rt寄存器为10，寄存器值为fffffffe；

ALU运算结果为ffffffff，DB数据为ffffffff；

写回寄存器为0a.

Rt <= rs + (符号位扩展)imm

bltz $10,-2



Name	Value
CLK	1
Reset	1
CurPC[31:0]	00000040
nextAddr[31:0]	00000044
instcode[31:0]	c940fffe
Reg1Out[31:0]	00000000
Reg2Out[31:0]	00000000
ALU_Input_A[31:0]	00000000
ALU_Input_B[31:0]	00000000
ALU_Out[31:0]	00000000
WriteData[31:0]	00000000
WriteRegAddr[4:0]	1f

在左侧图中，CurPC表示当前PC值为40；

nextAdder表示下一条指令地址为44；

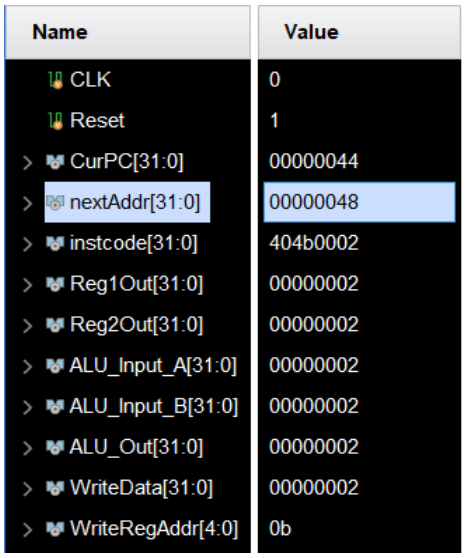
rs寄存器为10，寄存器值为00；

ALU运算结果为00，DB数据为00；

写回寄存器为1f.

if(rs<$zero)pc<=pc+4+(sign-extend)**immediate** <<2　else pc ←pc + 4

andi　$11,$2,2



Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000044
nextAddr[31:0]	00000048
instcode[31:0]	404b0002
Reg1Out[31:0]	00000002
Reg2Out[31:0]	00000002
ALU_Input_A[31:0]	00000002
ALU_Input_B[31:0]	00000002
ALU_Out[31:0]	00000002
WriteData[31:0]	00000002
WriteRegAddr[4:0]	0b

在左侧图中，CurPC表示当前PC值为44；

nextAdder表示下一条指令地址为48；

rs寄存器为02，寄存器值为02；

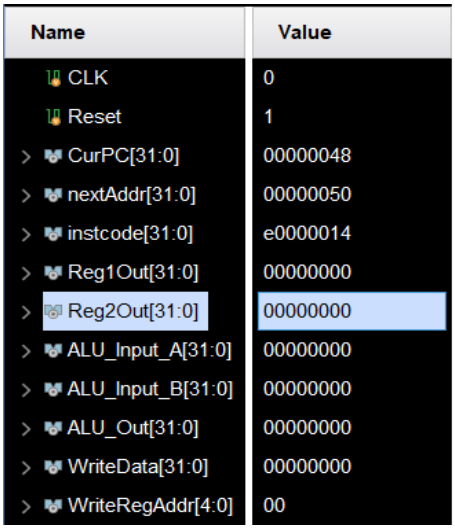
rt寄存器为11，寄存器值为02；

ALU运算结果为02，DB数据为02；

写回寄存器为0b.

rt←rs & (sign-extend)immediate

j 0x00000050



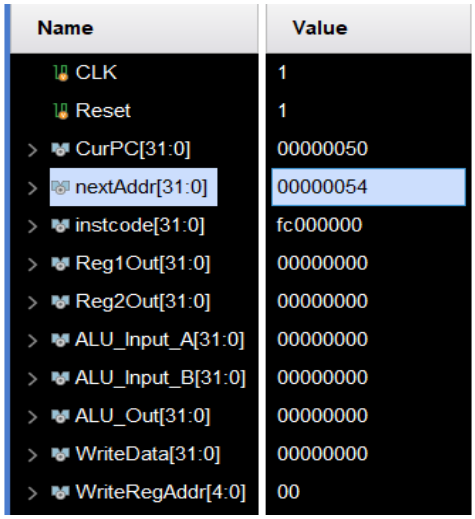
Name	Value
CLK	0
Reset	1
CurPC[31:0]	00000048
nextAddr[31:0]	00000050
instcode[31:0]	e0000014
Reg1Out[31:0]	00000000
Reg2Out[31:0]	00000000
ALU_Input_A[31:0]	00000000
ALU_Input_B[31:0]	00000000
ALU_Out[31:0]	00000000
WriteData[31:0]	00000000
WriteRegAddr[4:0]	00

在左侧图中，CurPC表示当前PC值为48；

nextAdder表示下一条指令地址为50；

跳转至50

halt



Name	Value
CLK	1
Reset	1
CurPC[31:0]	00000050
nextAddr[31:0]	00000054
instcode[31:0]	fc000000
Reg1Out[31:0]	00000000
Reg2Out[31:0]	00000000
ALU_Input_A[31:0]	00000000
ALU_Input_B[31:0]	00000000
ALU_Out[31:0]	00000000
WriteData[31:0]	00000000
WriteRegAddr[4:0]	00

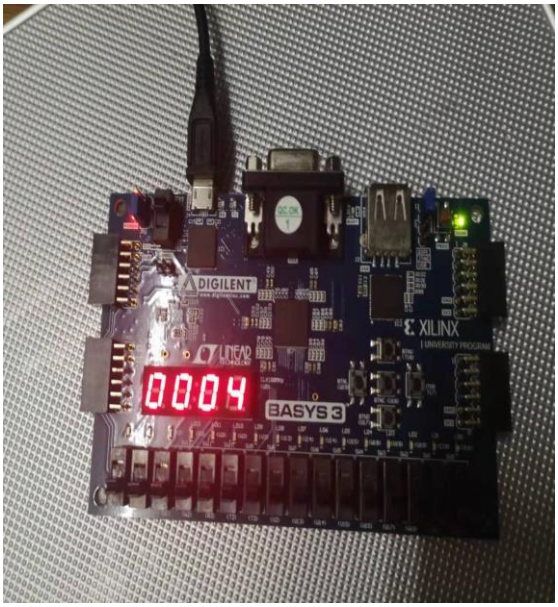
停机；

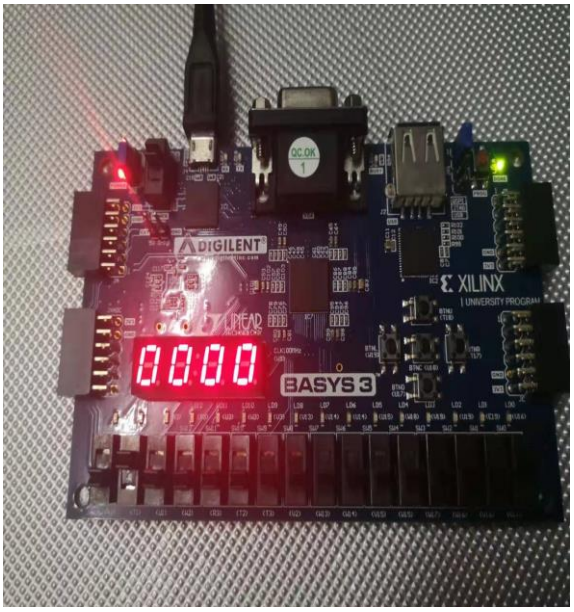
不改变 PC 的值，PC 保持不变。

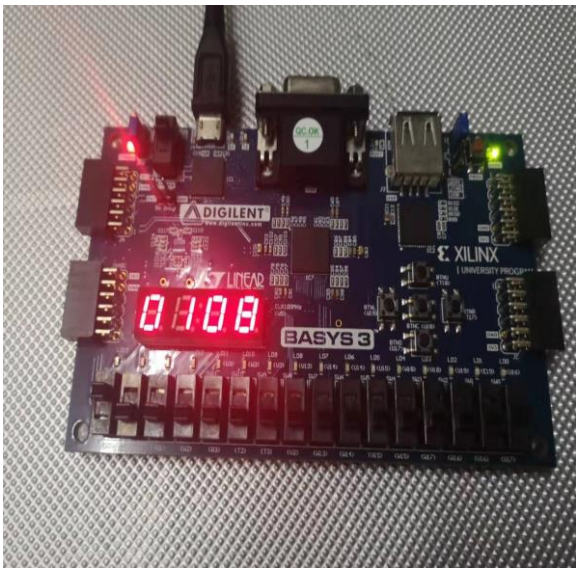
4、 实现

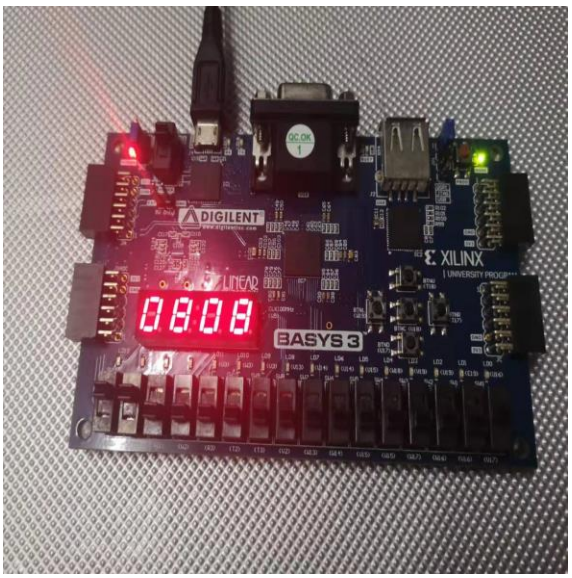
在BASY3板上实现CPU功能，展示效果如下：

addiu　$1,$0,8









从上到下依次为：

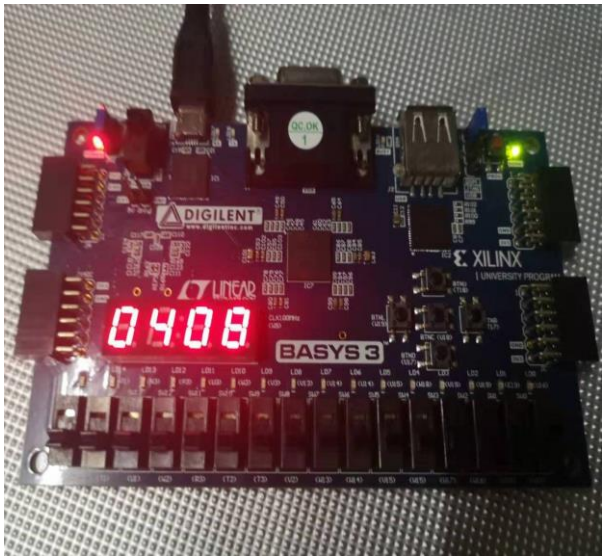
当前PC为00，下条PC为04；

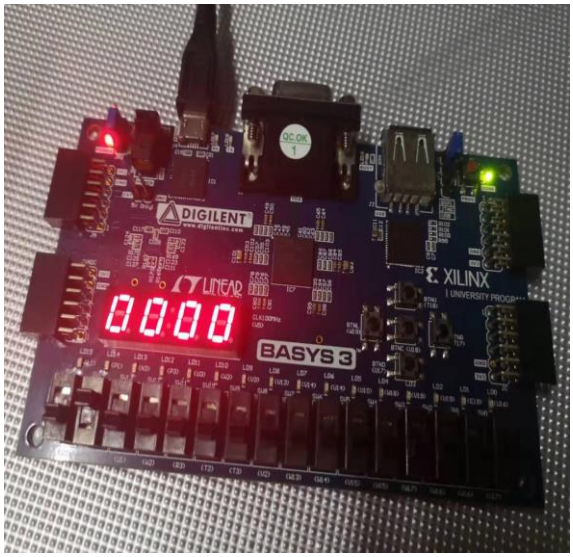
当前rs寄存器为00，寄存器值为00；

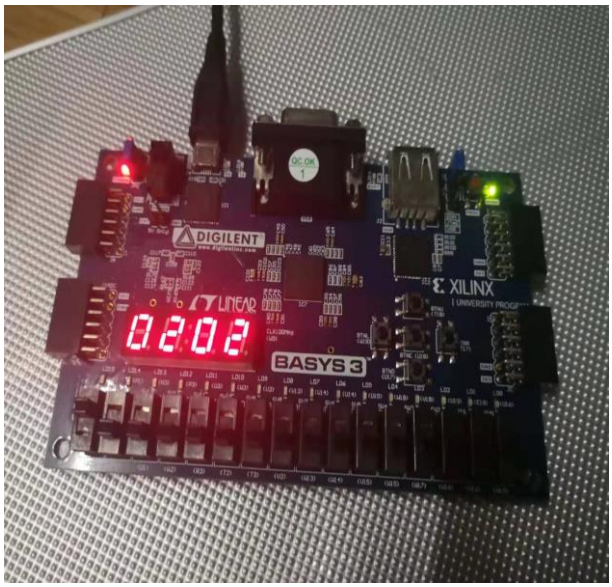
当前rt寄存器为01，寄存器值为08；

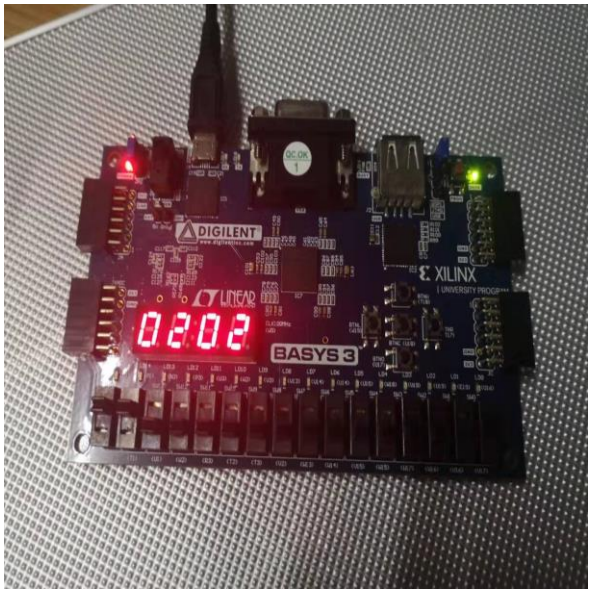
ALU结果为08，DB总线数据为08.

ori $2,$0,2









从上到下依次为：

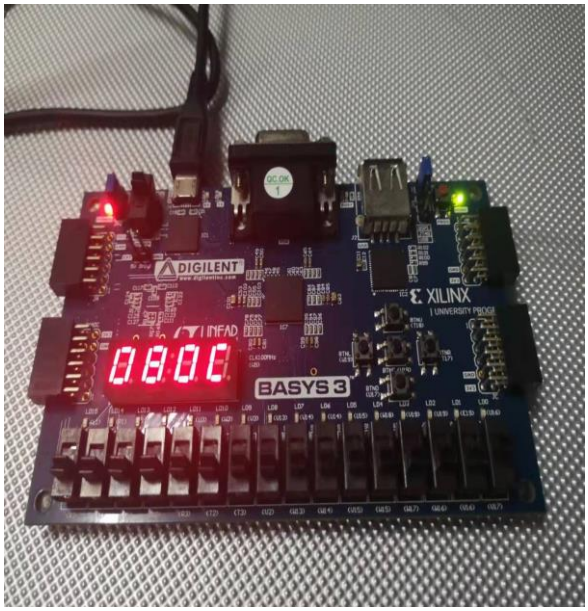
当前PC为04，下条PC为08；

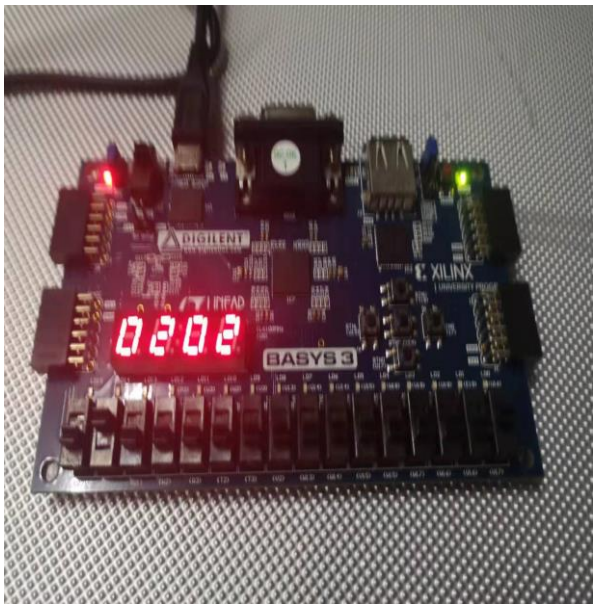
当前rs寄存器为00，寄存器值为00；

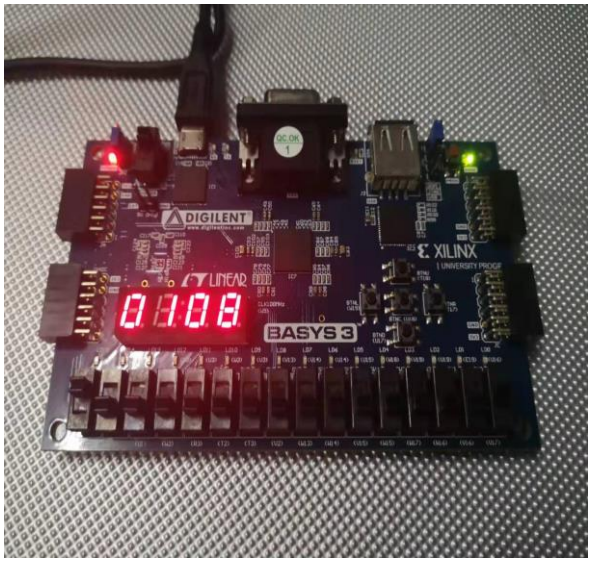
当前rt寄存器为02，寄存器值为02；

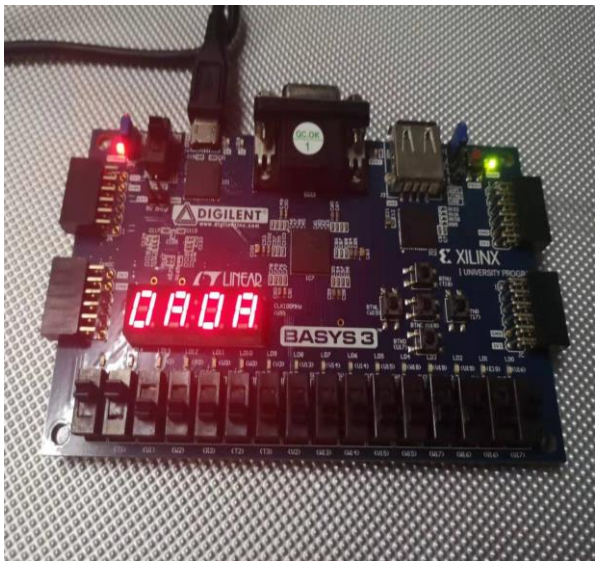
ALU结果为02，DB总线数据为02.

add $3,$2,$1









从上到下依次为：

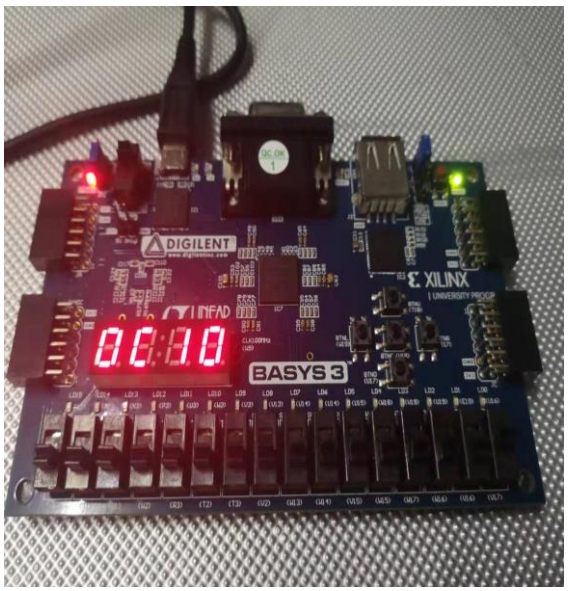
当前PC为08，下条PC为0C；

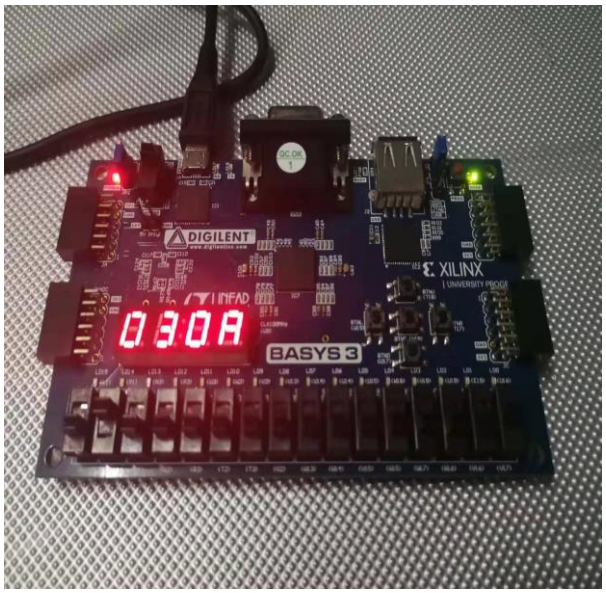
当前rs寄存器为02，寄存器值为02；

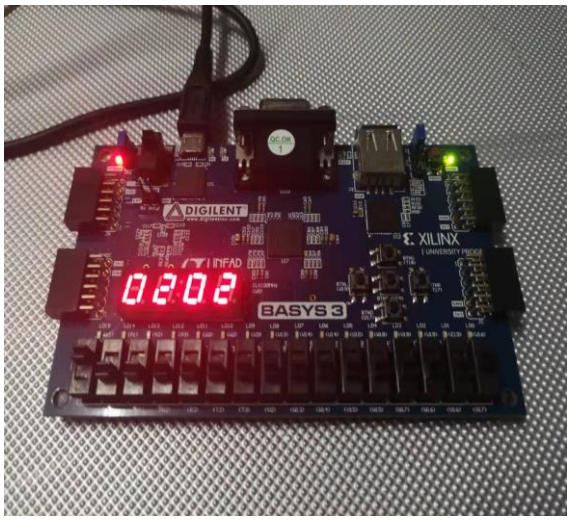
当前rt寄存器为01，寄存器值为08；

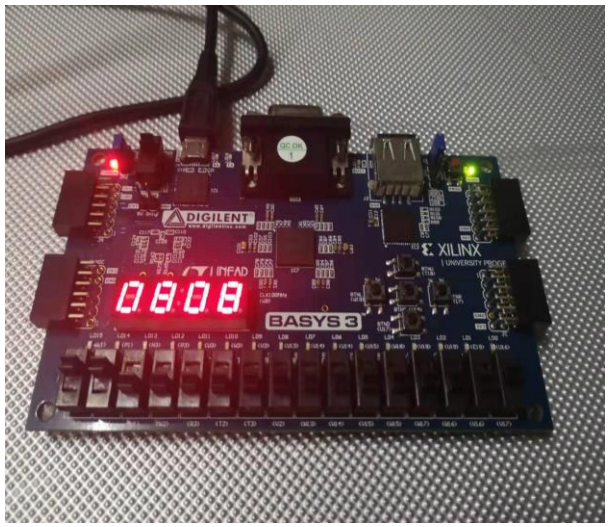
ALU结果为0A，DB总线数据为0A.

sub $5,$3,$2









从上到下依次为：

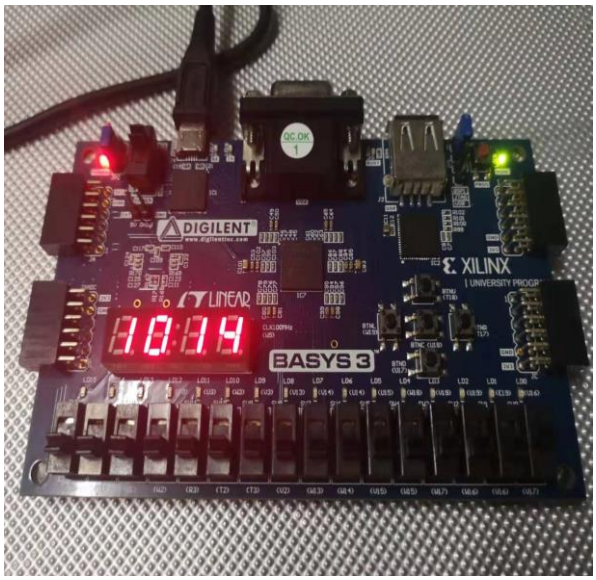
当前PC为0C，下条PC为10；

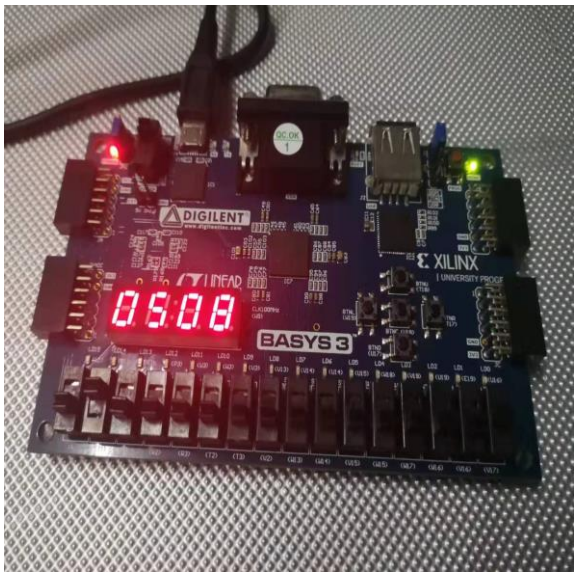
当前rs寄存器为03，寄存器值为0A；

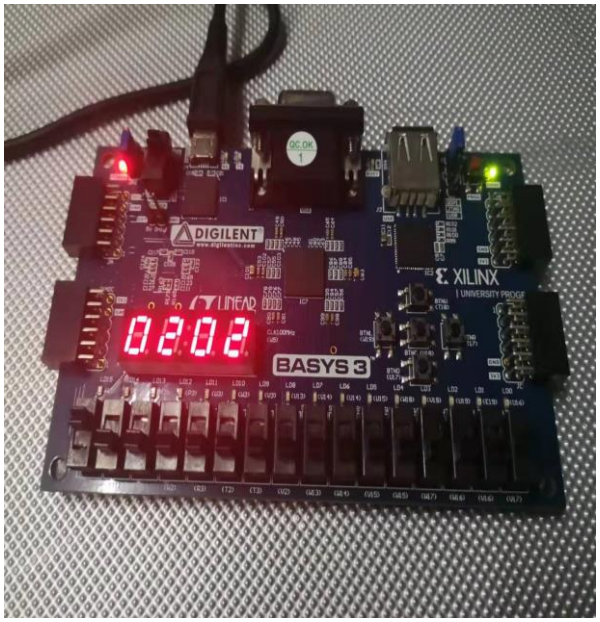
当前rt寄存器为02，寄存器值为02；

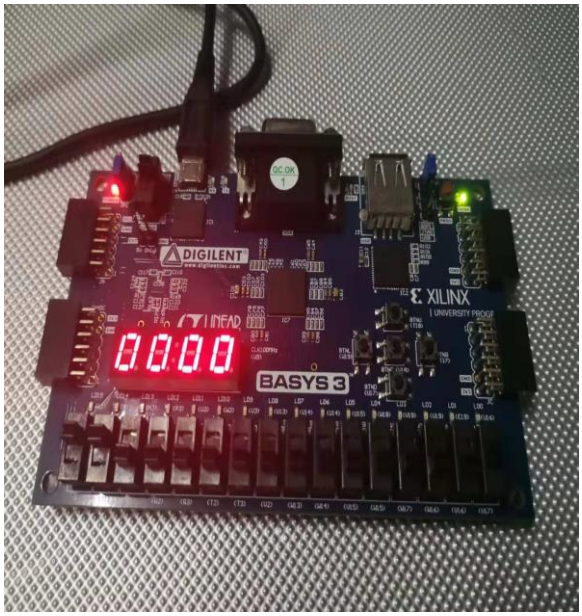
ALU结果为08，DB总线数据为08.

and $4,$5,$2









从上到下依次为：

当前PC为10，下条PC为14；

当前rs寄存器为05，寄存器值为08；

当前rt寄存器为02，寄存器值为02；

ALU结果为00，DB总线数据为00.

六. 实验心得

体会和建议：

在本次实验中，我遇到的问题有很多，除了一些语法上的错误之外，还有不少要注意的地方，现在我就把它们一一列出，谈一下我的体会。

第一个就是在InsRegister.v文件中，在readmemb的时候地址的斜杠和操作系统的斜杠是反向的，这导致程序无法读入测试用的指令，这个bug实在是让人心力憔悴，这是一个非常值得记住的问题。

还有就是在case语句里不能使用assign,否则将会报错 “the design is empty”,所以要换一种实现方法来替代这一功能。

最后，一定要多注意变量的大小写。这次的单周期CPU实验中每个.v文件里都有很多变量，而且不少变量还与其它.v文件之间有联系，所以如果写错了大小写，会有很多莫名其妙的bug，然后花很久都无法解决。

总而言之，在实验过程中一定要细心再细心，遇到问题要多上网搜索解决方法，并且在自己实在无法下手时，学会参考已有的代码(虽然很多都有问题)，这样才能高效完成实验。