



# 《计算机组成原理与接口技术实验》

## 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 软件工程二 (4) 班

学生姓名 : 刘宇庭

学号 : 16340158

时间 : 2018 年 5 月 20 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 (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： $rd \leftarrow rs + rt$ 。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： $rt \leftarrow rs + (\text{sign-extend}) \text{immediate}$ ；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： $rd \leftarrow rs - rt$

=> 逻辑运算指令

(4) ori rt , rs , immediate

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

功能： $rt \leftarrow rs | (\text{zero-extend}) \text{immediate}$ ；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： $rd \leftarrow rs \& rt$ ；逻辑与运算。

(6) or rd , rs , rt

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

功能： $rd \leftarrow rs | rt$ ；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) slti rt, rs, immediate 带符号

011011	rs(5位)	rt(5位)	immediate(16位)
--------	--------	--------	----------------

功能: if ( $rs < (\text{sign-extend immediate})$ )  $rt = 1$  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

(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( $rs \neq rt$ )  $pc \leftarrow pc + 4 + (\text{sign-extend immediate}) \ll 2$  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。

==> 跳转指令

(13) j addr

111000	addr[27..2]
--------	-------------

功能:  $pc \leftarrow \{(pc+4)[31..28], addr[27..2], 2\{0\}\}$ , 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

(14) halt

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

功能：停机；不改变 PC 的值，PC 保持不变。

### 三. 实验原理

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：

**R 类型:**

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

**I 类型:**

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

**J 类型:**

31	26 25	0
op	address	
6 位	26 位	

其中，

op: 为操作码；

rs: 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。

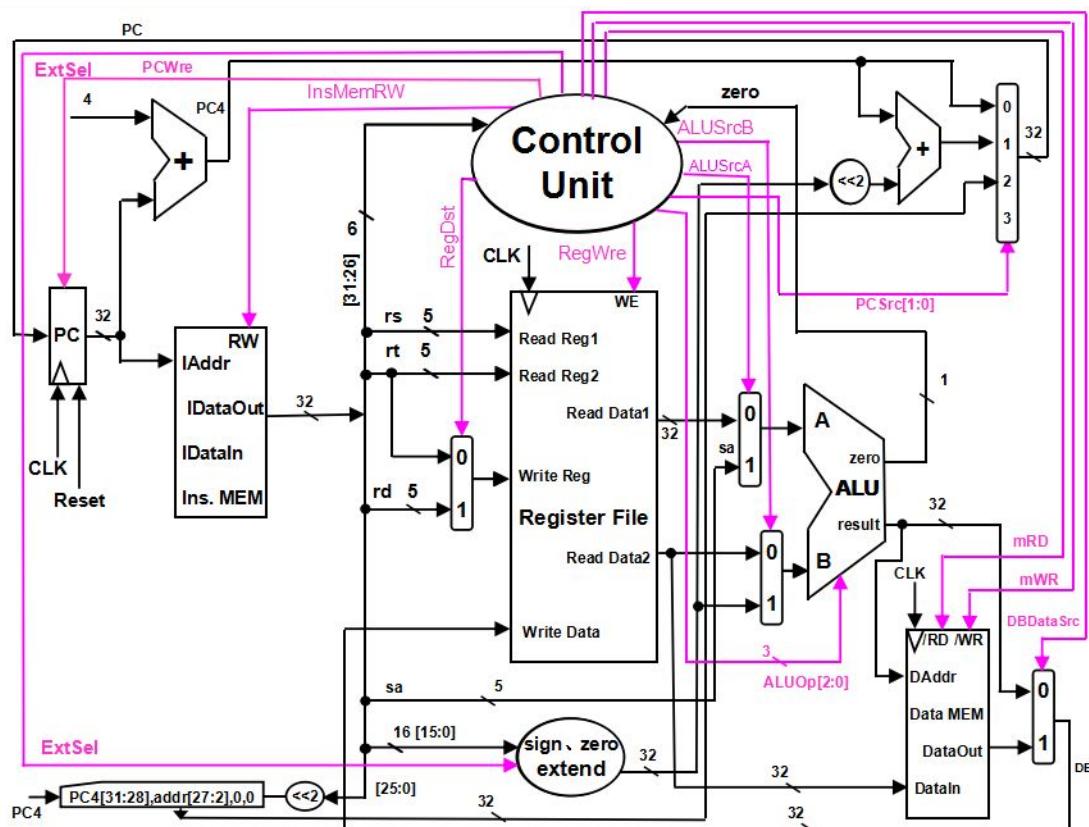


图 2 单周期 CPU 数据通路和控制线路图

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、slti、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、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、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、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、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: addi、ori、lw、slti	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: ori	(sign-extend) immediate (符号扩展) , 相关指令: addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00: pc<-pc+4, 相关指令: add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1); 01: pc<-pc+4+(sign-extend) immediate, 相关指令: beq(zero=1)、bne(zero=0); 10: pc<-{(pc+4)[31:28], addr[27:2], 2{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

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	$Y = A + B$	加

001	$Y = A - B$	减
010	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与
101	$Y = (A < B) ? 1 : 0$	比较 A 与 B 不带符号
110	$Y = (((regA < regB) \&& (regA[31] == regB[31]))    ((regA[31] == 1 \&& regB[31] == 0))) ? 1 : 0$	比较 A 与 B 带符号
111	$Y = A \oplus B$	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（当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作）。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. 实验过程与结果

设计思路以及流程：

- ① 完成控制信号与相对应指令之间相互关系的表格。

表3是依据表1控制信号的作用以及表2 ALU运算功能表完成的，某些指令无需用到部分模块，则相对应模块的使能控制信号与其无关。例如，对于跳转指令而言，其无需对数据寄存器进行读写操作，则数据寄存器相关的控制信号mRD, mWR设为0，防止修改里面的数据。部分指令执行不需要所有的模块都参与，故有些模块的控制信号与其没有直接关系，为了防止出现一些不必要的错误，统一将指令相对应的无关的使能控制信号默认设置为低电平(0)，无需ALU运算的(例如跳转指令)默认将其操作变成(000)。

指令	控制信号量											
	PCWre	ExtSel	InsMemRW	RegDst	RegWre	ALUSrcA	ALUSrcB	PCSrc(zero:0/1)	ALUOp	mRD	mWR	DBDataSrc
addi	1	1	1	0	1	0	1	00	000	0	0	0
ori	1	1	1	0	1	0	1	00	011	0	0	0
add	1	0	1	1	1	0	0	00	000	0	0	0
sub	1	1	1	1	1	0	0	00	001	0	0	0
and	1	0	1	1	1	0	0	00	100	0	0	0
or	1	0	1	1	1	0	0	00	011	0	0	0
sll	1	0	1	1	1	1	0	00	010	0	0	0
bne	1	1	1	X	0	0	0	01 / 00	001	0	0	0
slti	1	1	1	0	1	0	1	00	101	0	0	0
beq	1	1	1	X	0	0	0	00 / 01	001	0	0	0
sw	1	1	1	X	0	0	1	00	000	0	1	0
lw	1	1	1	0	1	0	1	00	000	1	0	1
j	1	0	1	X	0	X	X	10	000	0	0	0
halt	0	0	0	X	0	X	X	00	000	0	0	0

表 3 控制信号与相对应指令之间的相互关系

完成控制信号与相对应指令之间的关系以后该表后，对于如何实现单周期依旧感到很模糊，不知道相对应的信号量具体的控制意义，因此尝试结合实验原理中的图 2 单周期 CPU 数据通路和控制线路图，思考三种类型的指令，R 型、I 型、J 型指令的 CPU 处理过程。对于 R 型指令而言，主要是一些算术运算指令和逻辑运算，主要为取指令，解析指令，执行指令，将运算结果写回寄存器组，其不需要访问数据寄存器，下一条指令顺序下一条，即

$pc \leftarrow pc + 4$ , 其中的一些运算则由控制单元得到指令的操作码以后, 设置控制信号, 控制各个模块执行不同操作或者数据选择器选择相对应的输入作为输出; 对于 I 型指令, 其包含指令种类比较多, 存储器指令, 需要对存储器进行读或写的操作, 对于  $pc$  没有别的特别影响, 而分支指令则下一个  $pc$  可能不是  $pc + 4$ , 需要依据其运算结果做相对应的跳转操作或者顺序执行操作; 对于 J 型指令, 其是跳转指令, 跳转到指令中相对应的地址中, 主要对  $pc$  进行操作。不同类型的指令, 其进行的过程并非完全相同的, 不同类型指令所使用的模块并不是一样的, 所有的指令也不是都需要完整的五个处理阶段。结合 CPU 数据通路图以及指令相对应的控制信号后, 对于每种指令的数据通路有了一个比较清晰的了解, 对于每个控制信号与相对应的功能模块更加熟悉和了解, 理清了如何设计单周期 CPU, 即将其模块化, 并且在控制单元中依据指令的操作码, 对各个模块的控制信号进行一定的设定, 执行指令相对应的操作。

## ② CPU 模块划分与实现

依据图 2 单周期 CPU 数据通路和控制线路图, 将 CPU 划分为 9 个模块, 没有完全依据单周期 CPU 数据通路图进行划分, 主要依据数据通路图进行划分太冗余, 因此将一些数据选择器合并进了部分功能模块中, 实现简化。模块划分结果如图三所示。

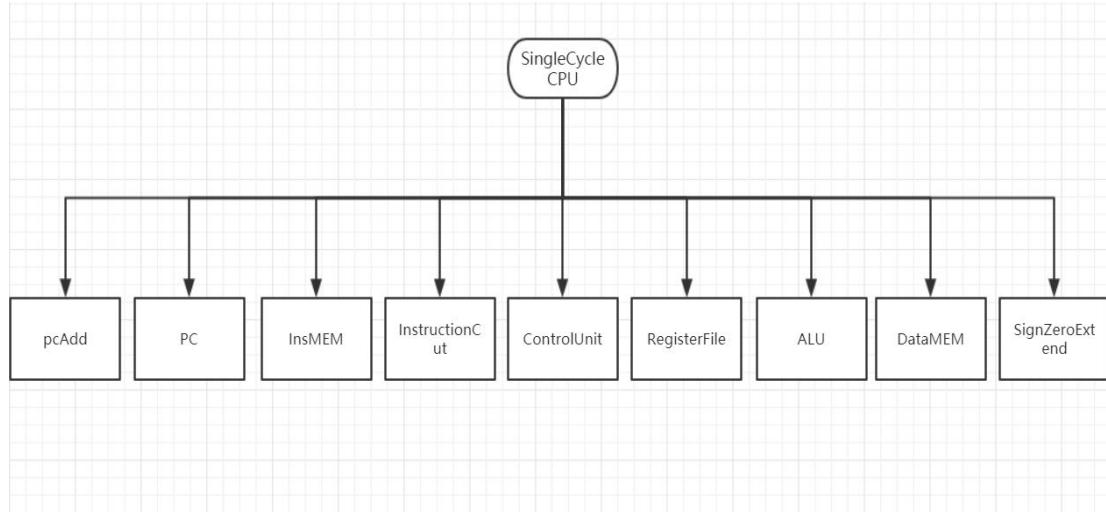


图 3 单周期 CPU 模块划分图

### 模块介绍:

#### 1. pcAdd

模块功能: 根据控制信号 PCSrc, 计算获得下一个  $pc$  以及控制信号 Reset 重置。

实现思路: 首先决定何时引起触发, 决定敏感变量, 该模块选择将时钟的下降沿以及控制信号 Reset 的下降沿作为敏感变量, 主要是为了能够确保下一条  $pc$  能够正确得到。

#### 主要实现代码:

```

always@(posedge CLK or negedge Reset)
begin
    if(!Reset) begin
        nextPC <= 0;
    end
    else begin
        pc <= curPC + 4;
        case(PCSrc)
            2'b00: nextPC <= curPC + 4;
            2'b01: nextPC <= curPC + 4 + immediate * 4;
            2'b10: nextPC <= {pc[31:28], addr, 2'b00};
            2'b11: nextPC <= nextPC;
        endcase
    end
end

```

## 2. PC

模块功能：根据控制信号PCWre，判断pc是否改变以及根据Reset信号判断是否重置

实现思路：将时钟信号的上升沿和控制信号Reset作为敏感变量，使得pc在上升沿的时候发生改变或被重置。

主要实现代码：

```

always@(posedge CLK or negedge Reset)
begin
    if(!Reset) // Reset == 0, PC = 0
        begin
            curPC <= 0;
        end
    else
        begin
            if(PCWre) // PCWre == 1
                begin
                    curPC <= nextPC;
                end
            else // PCWre == 0, halt
                begin
                    curPC <= curPC;
                end
        end
end

```

## 3. InsMEM

模块功能：依据当前pc，读取指令寄存器中，相对应地址的指令

实现思路：将pc的输入作为敏感变量，当pc发生改变的时候，则进行指令的读取，根据相关的地址，输出指令寄存器中相对应的指令

### 主要实现代码:

```

reg [7:0] rom[128:0]; // 存储器定义必须用reg类型，存储器存储单元8位长度，共128个存储单元，可以存32条指令

// 加载数据到存储器rom。注意：必须使用绝对路径
initial
begin
    $readmemb("F:\\Vivado\\SingleCycleCPU\\romData.txt", rom);
end

//大端模式
always@(IAddr or InsMemRW)
begin
    //取指令
    if(InsMemRW)
        begin
            IDataOut[7:0] = rom[IAddr + 3];
            IDataOut[15:8] = rom[IAddr + 2];
            IDataOut[23:16] = rom[IAddr + 1];
            IDataOut[31:24] = rom[IAddr];
        end
    //display("addr: %d insmemrw: %d inst: %d", IAddr, InsMemRW, IDataOut);
end

```

### 4. InstructionCut

**模块功能：**对指令进行分割，获得相对应的指令信息

**实现思路：**根据各种类型的指令结构，将指令分割，得到相对应的信息

### 主要实现代码:

```

always@(instruction)
begin
    op = instruction[31:26];
    rs = instruction[25:21];
    rt = instruction[20:16];
    rd = instruction[15:11];
    sa = instruction[10:6];
    immediate = instruction[15:0];
    addr = instruction[25:0];
end

```

### 5. ControlUnit

**模块功能：**控制单元，依据指令的操作码(op)以及标记符(ZERO)，输出PCWre、ALUSrcB等控制信号，各控制信号的作用见实验原理的控制信号作用表（表3），从而达到控制各指令的目的。

### 主要实现代码:

```
always@(op or zero)
```

```

begin
    PCWre = (op == 6'b111111) ? 0 : 1; //halt
    InsMemRW = (op == 6'b111111) ? 0 : 1;
    mWR = (op == 6'b100110) ? 1 : 0; //sw
    mRD = (op == 6'b100111) ? 1 : 0; //lw
    DBDataSrc = (op == 6'b100111) ? 1 : 0;

case(op)
    //addi
    6'b000001:
        begin
            ExtSel = 1;
            RegDst = 0;
            RegWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 1;
            PCSrc = 2'b00;
            ALUOp = 3'b000;
        end
    //ori
    6'b010000:
        begin
            ExtSel = 1;
            RegDst = 0;
            RegWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 1;
            PCSrc = 2'b00;
            ALUOp = 3'b011;
        end
    //add
    6'b000000:
        begin
            ExtSel = 0;
            RegDst = 1;
            RegWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 0;
            PCSrc = 2'b00;
            ALUOp = 3'b000;
        end
    //sub
    6'b000010:
        begin

```

```
ExtSel = 1;
RegDst = 1;
RegWre = 1;
ALUSrcA = 0;
ALUSrcB = 0;
PCSrc = 2' b00;
ALUOp = 3' b001;
begin
end
//and
6' b010001:
begin
ExtSel = 0;
RegDst = 1;
RegWre = 1;
ALUSrcA = 0;
ALUSrcB = 0;
PCSrc = 2' b00;
ALUOp = 3' b100;
end
//or
6' b010010:
begin
ExtSel = 0;
RegDst = 1;
RegWre = 1;
ALUSrcA = 0;
ALUSrcB = 0;
PCSrc = 2' b00;
ALUOp = 3' b011;
end
//sll
6' b011000:
begin
ExtSel = 0;
RegDst = 1;
RegWre = 1;
ALUSrcA = 1;
ALUSrcB = 0;
PCSrc = 2' b00;
ALUOp = 3' b010;
end
//bne
6' b110001:
begin
```

```

ExtSel = 1;
RegDst = 0;
RegWre = 0;
ALUSrcA = 0;
ALUSrcB = 0;
PCSrc = zero ? 2'b00 : 2'b01;
ALUOp = 3'b001;
end
//slti
6'b011011:
begin
    ExtSel = 1;
    RegDst = 0;
    RegWre = 1;
    ALUSrcA = 0;
    ALUSrcB = 1;
    PCSrc = 2'b00;
    ALUOp = 3'b101;
end
//beq
6'b110000:
begin
    ExtSel = 1;
    RegDst = 0;
    RegWre = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    PCSrc = zero ? 2'b01 : 2'b00;
    ALUOp = 3'b001;
end
//sw
6'b100110:
begin
    ExtSel = 1;
    RegDst = 0;
    RegWre = 0;
    ALUSrcA = 0;
    ALUSrcB = 1;
    PCSrc = 2'b00;
    ALUOp = 3'b000;
end
//lw
6'b100111:
begin

```

```

ExtSel = 1;
RegDst = 0;
RegWre = 1;
ALUSrcA = 0;
ALUSrcB = 1;
PCSrc = 2' b00;
ALUOp = 3' b000;
end
//j
6' b111000:
begin
    ExtSel = 0;
    RegDst = 0;
    RegWre = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    PCSrc = 2' b10;
    ALUOp = 3' b000;
end
//halt
6' b111111:
begin
    ExtSel = 0;
    RegDst = 0;
    RegWre = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    PCSrc = 2' b11;
    ALUOp = 3' b000;
end
endcase
end

```

## 6. RegisterFile

模块功能：寄存器组，通过控制单元输出的控制信号，进行相对应的读或写操作

主要实现代码：

```

reg [31:0] regFile[0:31]; // 寄存器定义必须用reg类型
integer i;
initial begin
    for (i = 0; i < 32; i = i + 1) regFile[i] <= 0;
end

always@(ReadReg1 or ReadReg2)
begin
    ReadData1 = regFile[ReadReg1];
    ReadData2 = regFile[ReadReg2];
    //$display("regfile %d %d\n", ReadReg1, ReadReg2);
end

always@(negedge CLK)
begin
    //0恒为0，所以写入寄存器的地址不能为0
    if(RegWre && WriteReg)
        begin
            regFile[WriteReg] <= WriteData;
        end
    end

```

## 7. ALU

**模块功能：**算术逻辑单元，对两个输入依据ALUOp进行相对应的运算

**实现思路：**依据实验原理中的ALU运算功能表(表2)完成操作码对应的操作

**主要实现代码：**

```

always@(ReadData1 or ReadData2 or ALUSrcA or ALUSrcB or ALUOp)
begin
    //定义两个输入端口
    A = (ALUSrcA == 0) ? ReadData1 : sa;
    B = (ALUSrcB == 0) ? ReadData2 : extend;
    case(ALUOp)
        3'b000: result = A + B;
        3'b001: result = A - B;
        3'b010: result = B << A;
        3'b011: result = A | B;
        3'b100: result = A & B;
        3'b101: result = (A < B) ? 1 : 0;
        3'b110: result = (((ReadData1 < ReadData2) && (ReadData1[31] == ReadData2[31])) || ((ReadData1[31] == 1 && ReadData2[31] == 0))) ? 1:0;
        3'b111: result = A ^ B;
    endcase
    zero = (result == 0) ? 1 : 0;
end

```

## 8. DataMEM

**模块功能：**数据存储器，通过控制信号，对数据寄存器进行读或者写操作，并且此处模块额外合并了输出DB的数据选择器，此模块同时输出写回寄存器组的数据DB。

**主要实现代码：**

```

reg [7:0] ram [0:31];      // 存储器定义必须用reg类型

always@ (mRD or DAddr or DBDataSrc)
begin
    //读
    DataOut[7:0] = mRD ? ram[DAddr + 3] : 8'bzz; // z 为高阻态
    DataOut[15:8] = mRD ? ram[DAddr + 2] : 8'bzz;
    DataOut[23:16] = mRD ? ram[DAddr + 1] : 8'bzz;
    DataOut[31:24] = mRD ? ram[DAddr] : 8'bzz;

    DB = (DBDataSrc == 0) ? DAddr : DataOut;
end

always@ (negedge CLK)
begin
    //写
    if (mWR)
        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
    // $display("mwr: %d $12 %d %d %d", mWR, ram[12], ram[13], ram[14], ram[15]);
end

```

## 9. SignZeroExtend

**模块功能:** 根据指令相关的控制信号ExtSel, 对立即数进行扩展。

**实现思路:** 根据控制信号ExtSel判断是0扩展还是符号扩展, 然后进行相对应的扩展

**主要实现代码:**

```

assign extendImmediate[15:0] = immediate;
assign extendImmediate[31:16] = ExtSel ? (immediate[15] ? 16'hffff : 16'h0000) : 16'h0000;

```

## 10. 顶层模块: SingleCycleCPU

**实现思路:** 在顶层模块中将各个已实现的底层模块进行实例, 并且用verilog语言将各个模块用线连接起来

**实例模块:**

```

pcAdd pcAdd(. Reset(Reset),
          . CLK(CLK),
          . PCSrc(PCSsrc),
          . immediate(extend),
          . addr(addr),
          . curPC(curPC),
          . nextPC(nextPC));

PC pc(. CLK(CLK),
      . Reset(Reset),
      . PCWre(PCWre),
      . PCSrc(PCSsrc),
      . nextPC(nextPC),
      . curPC(curPC));

InsMEM InsMEM(. IAddr(curPC),
              . InsMemRW(InsMemRW),
              . IDataOut(instruction));

InstructionCut InstructionCut(. instruction(instruction),
                             . op(op),
                             . rs(rs),
                             . rt(rt),
                             . rd(rd),
                             . sa(sa),
                             . immediate(immediate),
                             . addr(addr));

DataMEM DataMEM(. mRD(mRD),
                . mWR(mWR),
                . CLK(CLK),
                . DBDataSrc(DBDataSrc),
                . DAddr(result),
                . DataIn(B),
                . DataOut(DataOut),
                . DB(DB));

```

---

```

ControlUnit ControlUnit(. zero(zero),
                        . op(op),
                        . PCWre(PCWre),
                        . ExtSel(ExtSel),
                        . InsMemRW(InsMemRW),
                        . RegDst(RegDst),
                        . RegWre(RegWre),
                        . ALUSrcA(ALUSrcA),
                        . ALUSrcB(ALUSrcB),
                        . PCSrc(PCSsrc),
                        . ALUOp(ALUOp),
                        . mRD(mRD),
                        . mWR(mWR),
                        . DBDataSrc(DBDataSrc));

```

```

RegisterFile RegisterFile(. CLK(CLK),
                        . ReadReg1(rs),
                        . ReadReg2(rt),
                        . WriteData(DB),
                        . WriteReg(RegDst ? rd : rt),
                        . RegWre(RegWre),
                        . ReadData1(A),
                        . ReadData2(B));

```

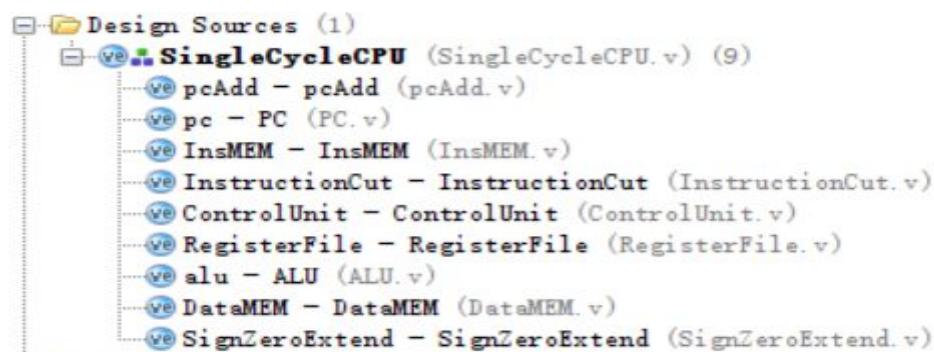
```

ALU alu(. ALUSrcA(ALUSrcA),
         . ALUSrcB(ALUSrcB),
         . ReadData1(A),
         . ReadData2(B),
         . sa(sa),
         . extend(extend),
         . ALUOp(ALUOp),
         . zero(zero),
         . result(result));

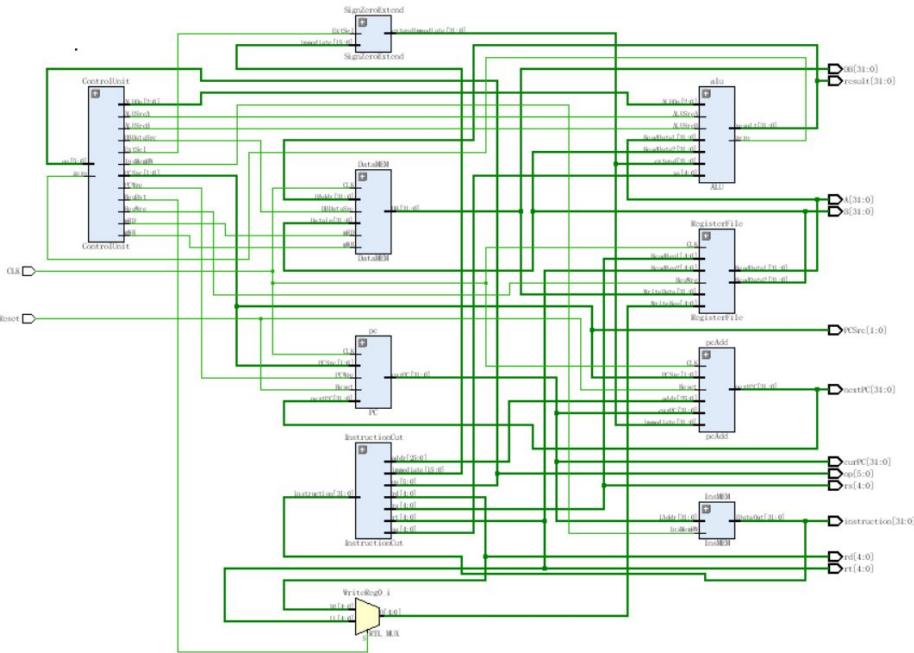
```

---

### 文件结构图:



### RTL Schematic:



### ③ CPU 正确性的验证

#### 1. 仿真文件

实例一个单周期 cpu:

```
// Instantiate the Unit Under Test (UUT)
SingleCycleCPU uut (
    .CLK(CLK),
    .Reset(Reset),
    .curPC(curPC),
    .nextPC(nextPC),
    .instruction(instruction),
    .op(op),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .DB(DB),
    .A(A),
    .B(B),
    .result(result),
    .PCSrc(PCSrc),
    .zero(zero),
    .PCWre(PCWre),
    .ExtSel(ExtSel),
    .InsMemRW(InsMemRW),
    .RegDst(RegDst),
    .RegWre(RegWre),
    .ALUSrcA(ALUSrcA),
    .ALUSrcB(ALUSrcB),
    .ALUOp(ALUOp),
    .mRD(mRD),
    .mWR(mWR),
    .DEDDatasrc(DEDDatasrc)
);
```

初始化并产生时钟信号：

```

initial begin
    // Initialize Inputs
    CLK = 1;
    Reset = 0;

    CLK = !CLK; // 下降沿，使PC先清零
    Reset = 1; // 消除保持信号
    forever #5
    begin // 产生时钟信号，周期为50s
        CLK = !CLK;
    end
end

```

## 2. 程序代码测试

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	0001 1000 0000 0000	= 00411800
0x00000000 C	sub \$5,\$3,\$2	000010	00011	00010	0010 1000 0000 0000	= 08622800
0x00000010	and \$4,\$5,\$2	010001	00101	00010	0010 0000 0000 0000	= 44a22000
0x00000014	or \$8,\$4,\$2	010010	00100	00010	0100 0000 0000 0000	= 48824000
0x00000018	sll \$8,\$8,1	011000	00000	01000	0100 0000 0100 0000	= 60084040
0x00000001 C	bne \$8,\$1,-2 (#,转 18)	110001	01000	00001	1111 1111 1111 1110	= c501ffff
0x00000020	slti \$6,\$2,8	011011	00010	00110	0000 0000 0000 1000	= 6c460008
0x00000024	slti \$7,\$6,0	011011	00110	00111	0000 0000 0000 0000	= 6cc70000
0x00000028	addi \$7,\$7,8	000001	00111	00111	0000 0000 0000 1000	= 04e70008
0x00000002 C	beq \$7,\$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	= c0e1ffff
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	j 0x00000040	111000	00000	00000	0000 0000 0001 0000	= e0000010
0x00000003 C	addi \$10,\$0,10	000001	00000	01010	0000 0000 0000 1010	= 0401000a
0x00000040	Halt	111111	00000	00000	0000 0000 0000 0000	= fc000000
0x00000044						
0x00000048						
0x00000044 C						

表 4 程序测试段

使用表 4 进行测试 CPU 正确性，将其中的指令写入一个 romData.txt 文件中。

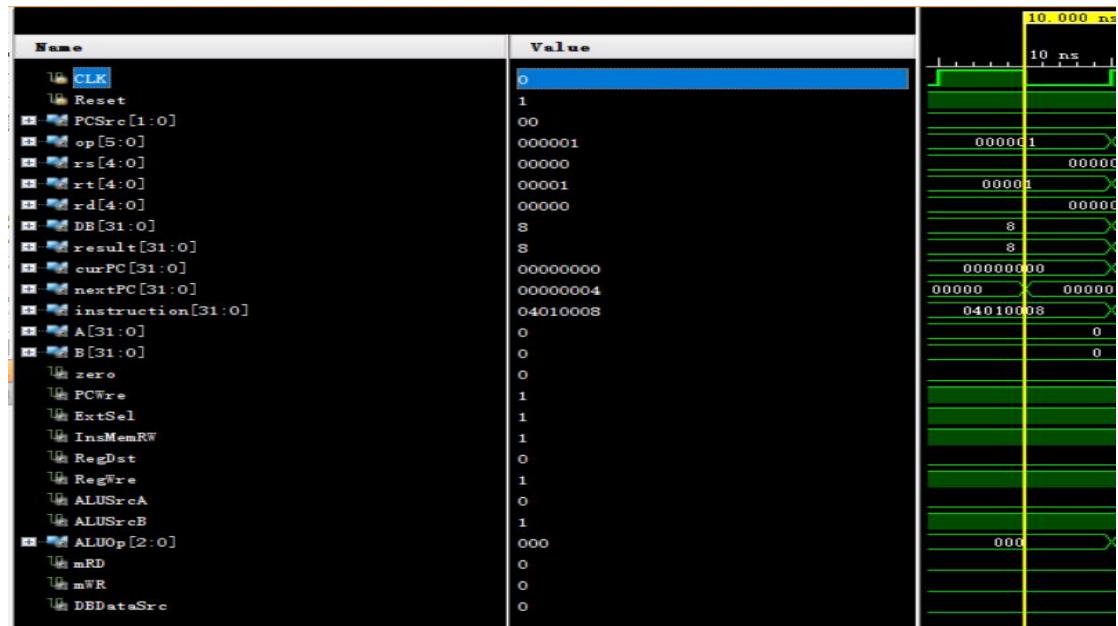
在模块 InsMEM 中进行读入（使用的路径为绝对路径）

```
$readmem("F:\\Vivado\\SingleCycleCPU\\romData.txt", rom);
```

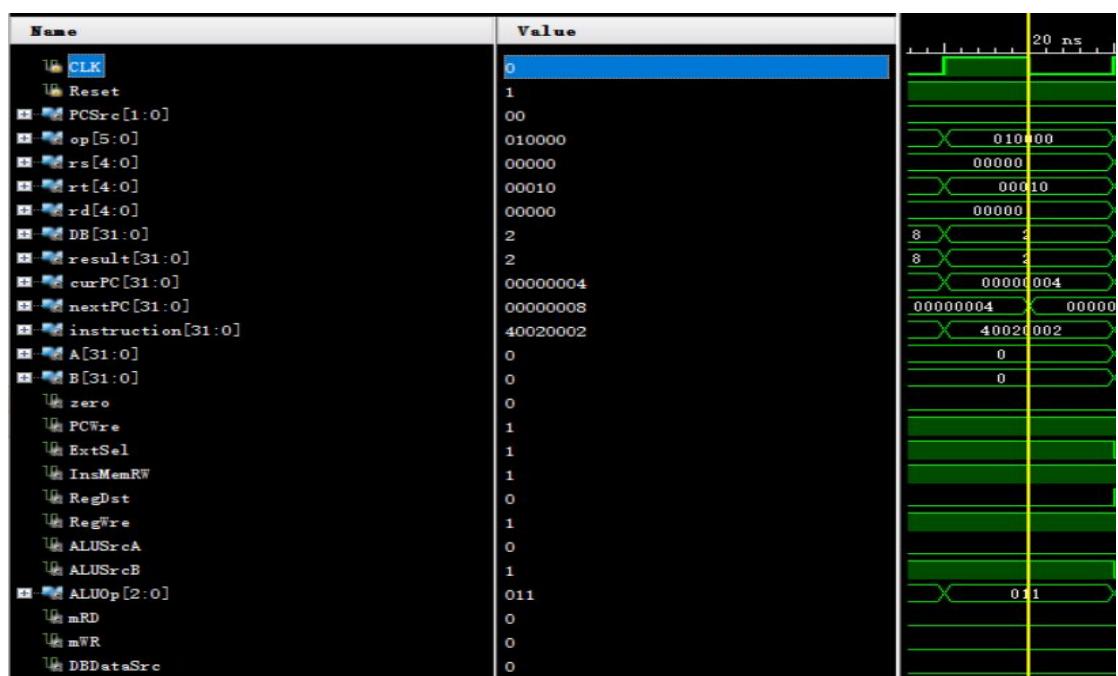
测试：

(注：仿真输出中 A、B 为寄存器组的两个输出并非 ALU 的两个输入，curPC 为当前 PC，nextPC 为下一条 PC，其余与上文中的控制信号对应，指令 instruction、curPC、nextPC 为十六进制输出，ALU 计算结果 result 和写回数据 DB 为有符号整数输出)

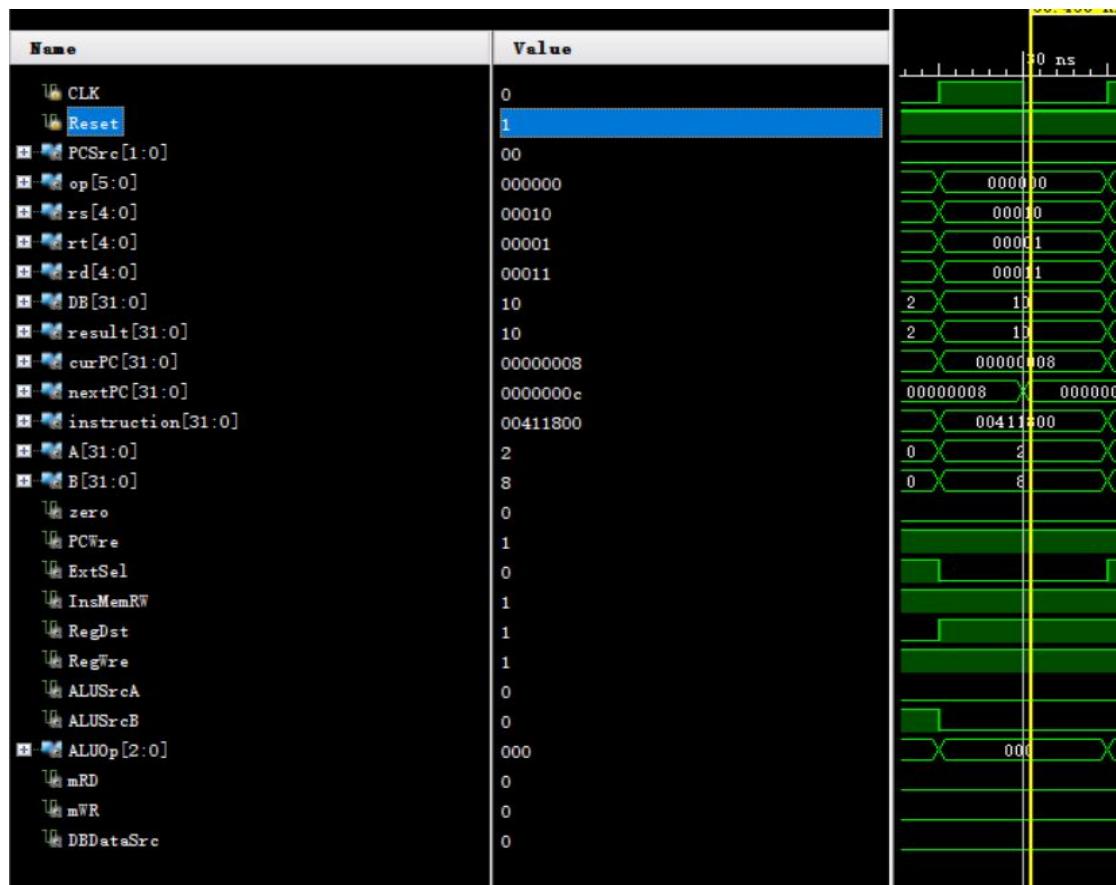
●	0x00000000	addi \$1,\$0,8	000001	00000	00001	0000 0000 0000 1000	=	04010008
---	------------	----------------	--------	-------	-------	---------------------	---	----------



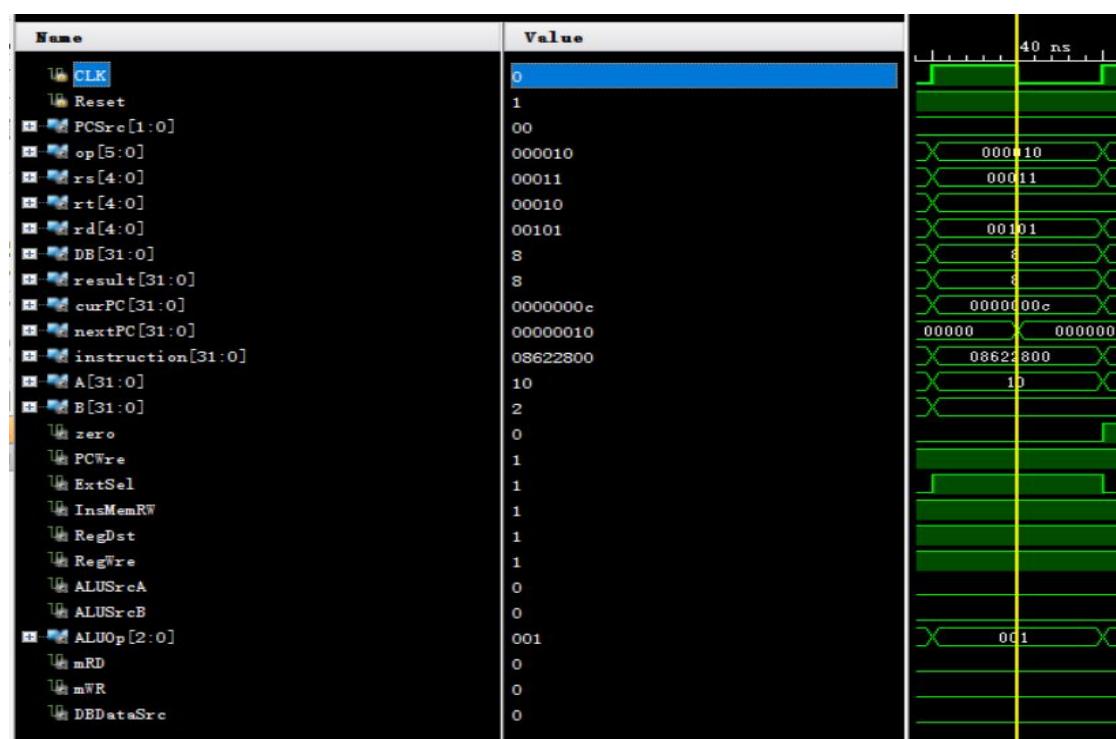
●	0x00000004	ori \$2,\$0,2	010000	00000	00010	0000 0000 0000 0010	=	40020002
---	------------	---------------	--------	-------	-------	---------------------	---	----------



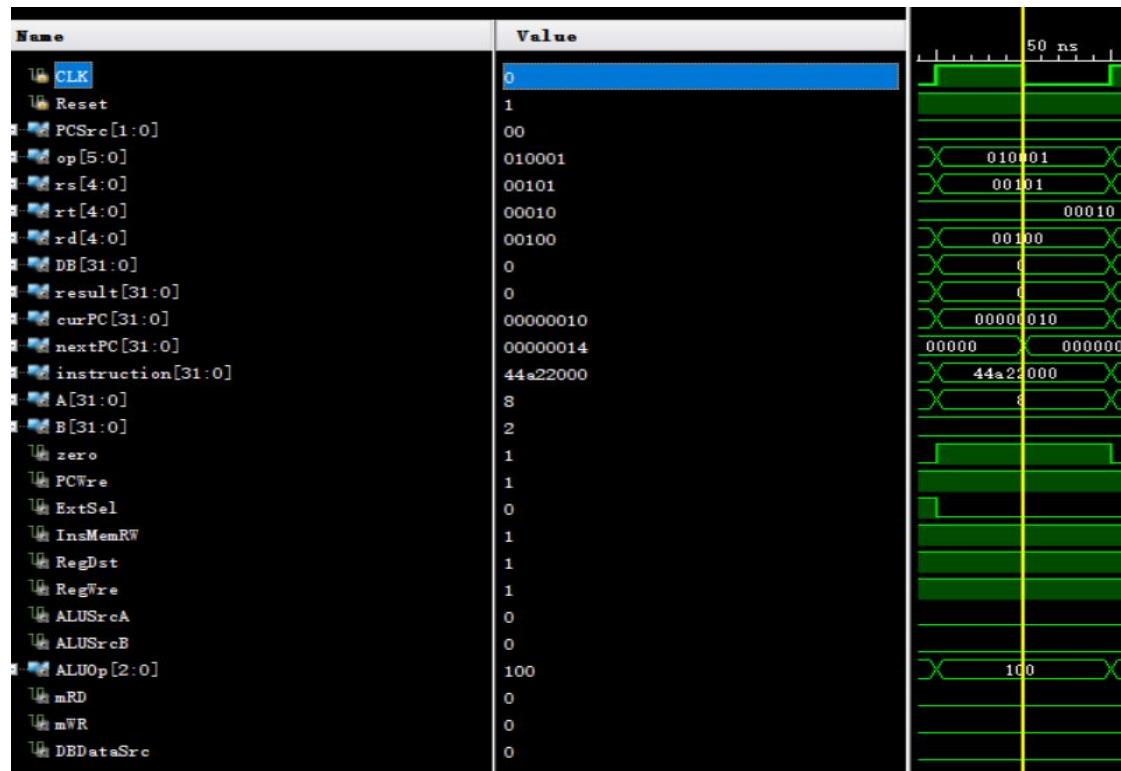
0x00000008	add \$3,\$2,\$1	000000	00010	00001	0001 1000 0000 0000	=	00411800
------------	-----------------	--------	-------	-------	---------------------	---	----------



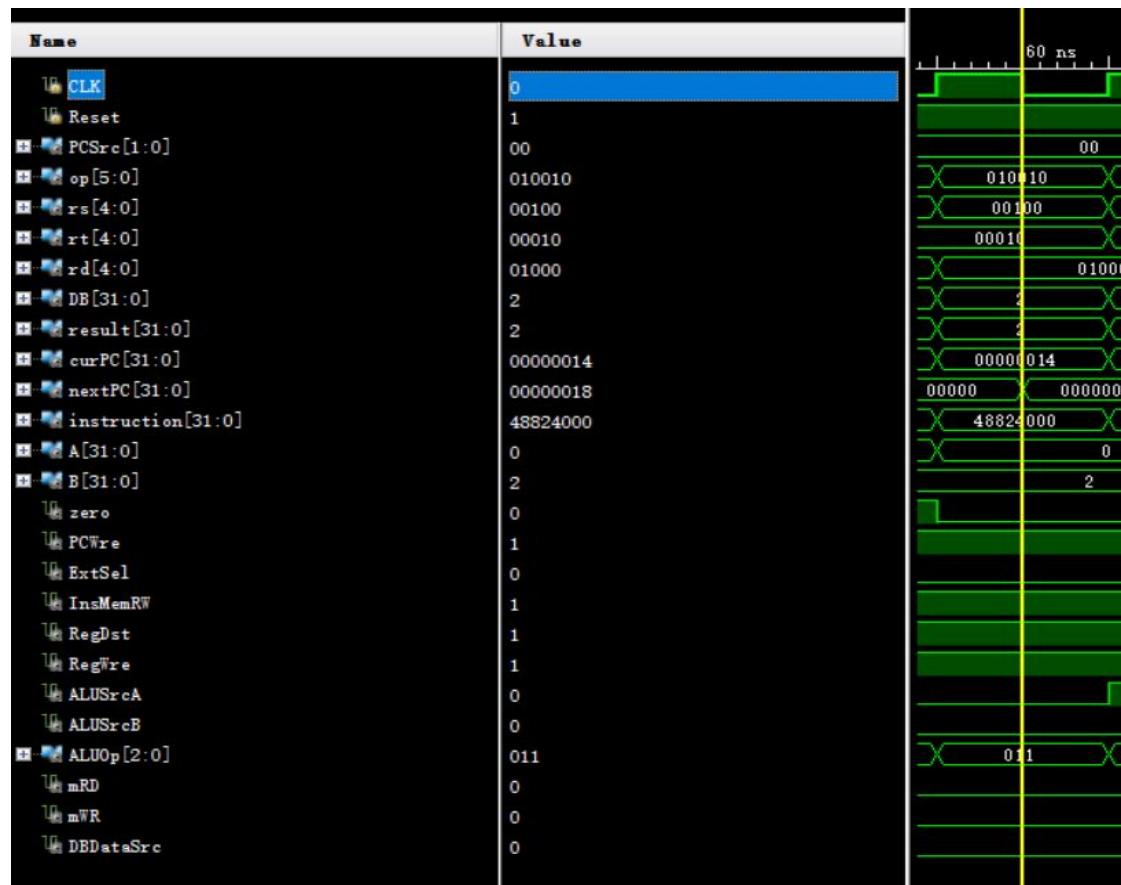
0x0000000C	sub \$5,\$3,\$2	000010	00011	00010	0010 1000 0000 0000	=	08622800
------------	-----------------	--------	-------	-------	---------------------	---	----------

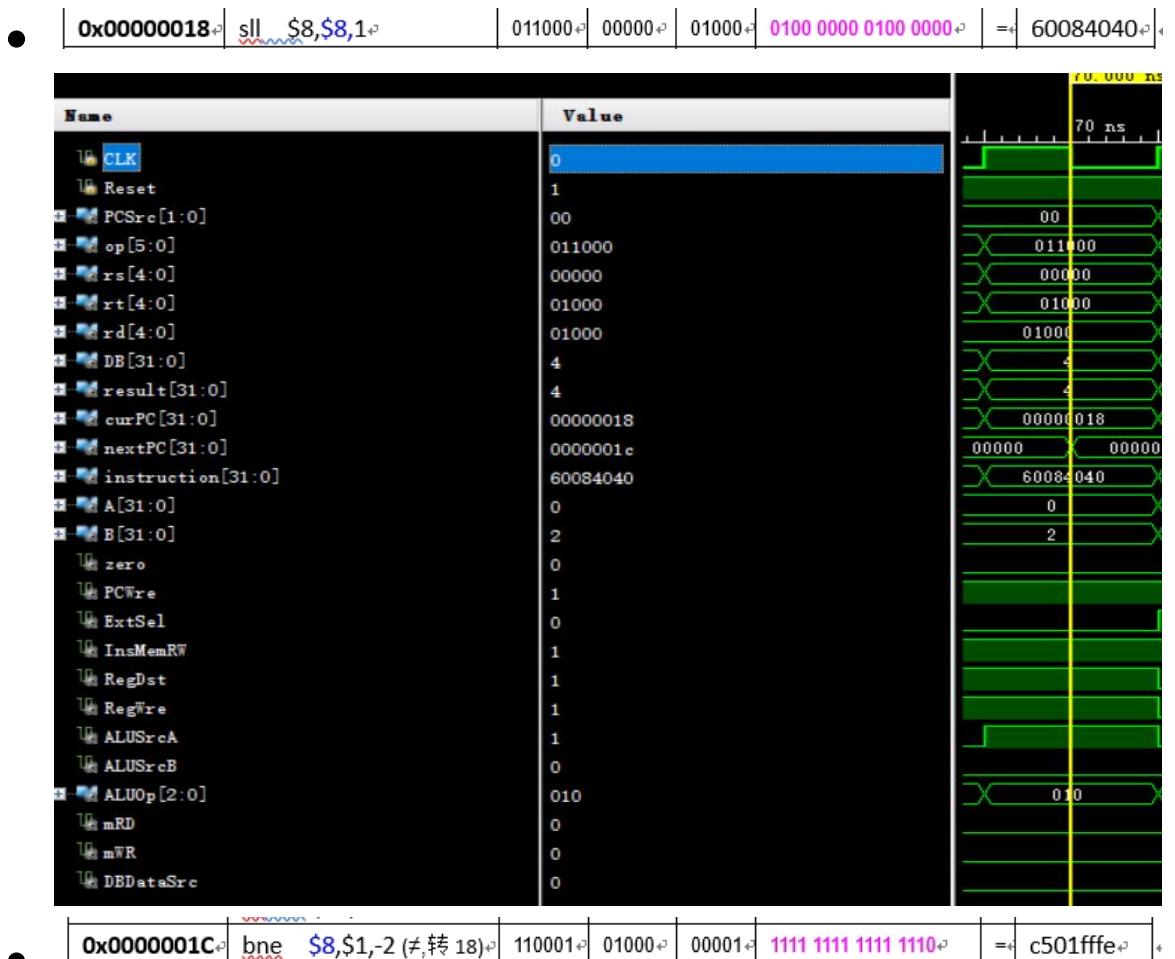


0x00000010	and \$4,\$5,\$2	010001	00101	00010	0010 0000 0000 0000	= 44a22000
------------	-----------------	--------	-------	-------	---------------------	------------

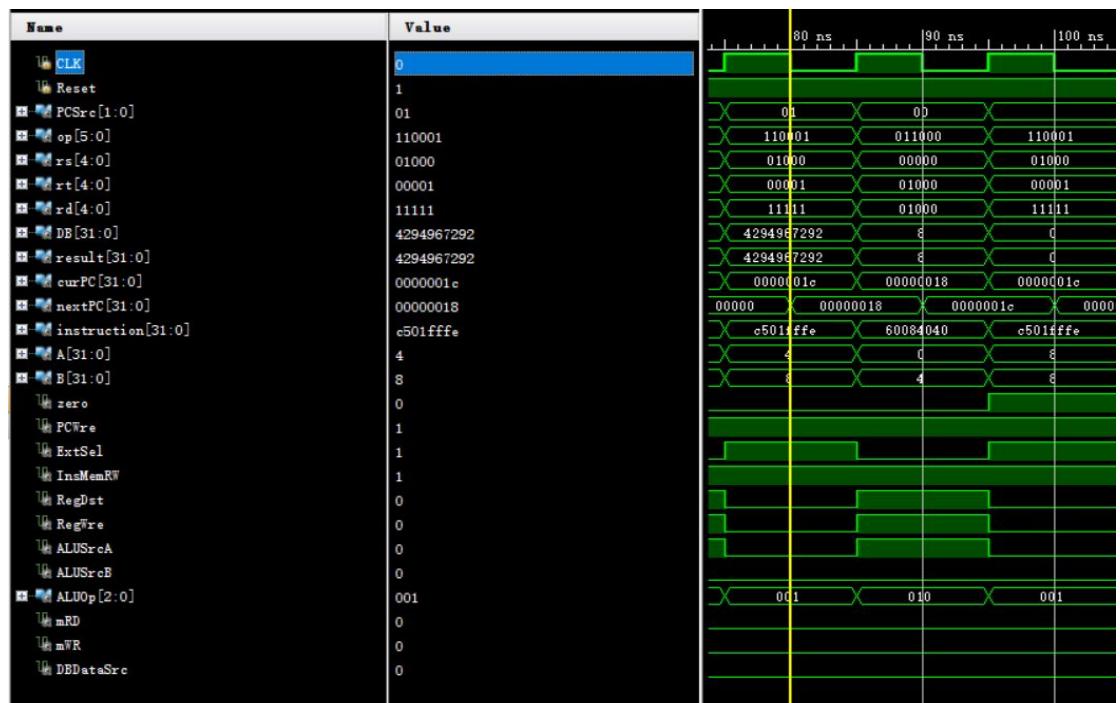


0x00000014	or \$8,\$4,\$2	010010	00100	00010	0100 0000 0000 0000	= 48824000
------------	----------------	--------	-------	-------	---------------------	------------

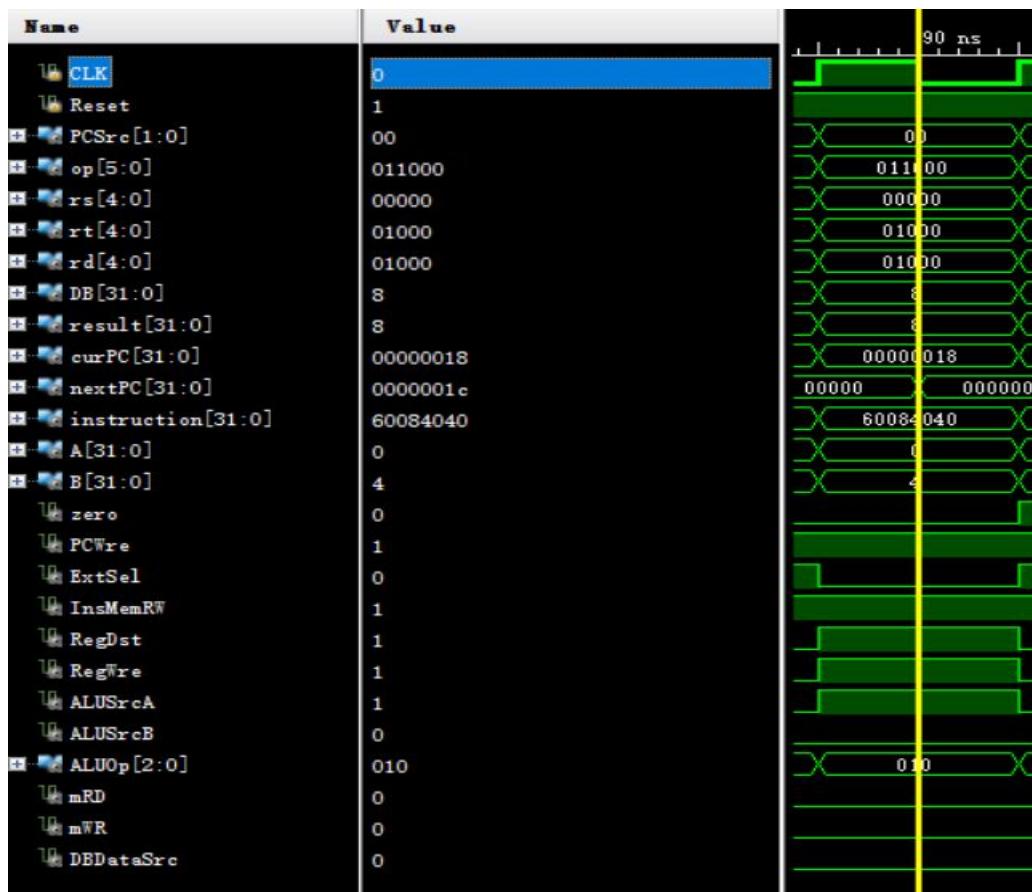




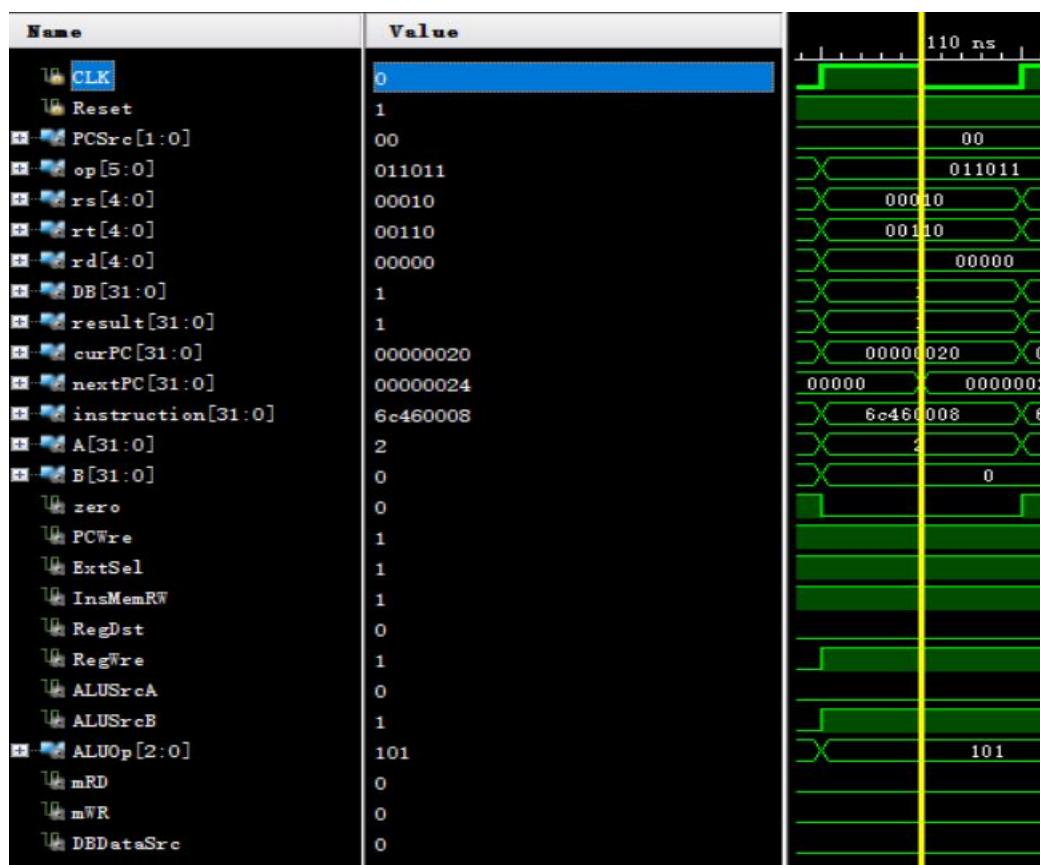
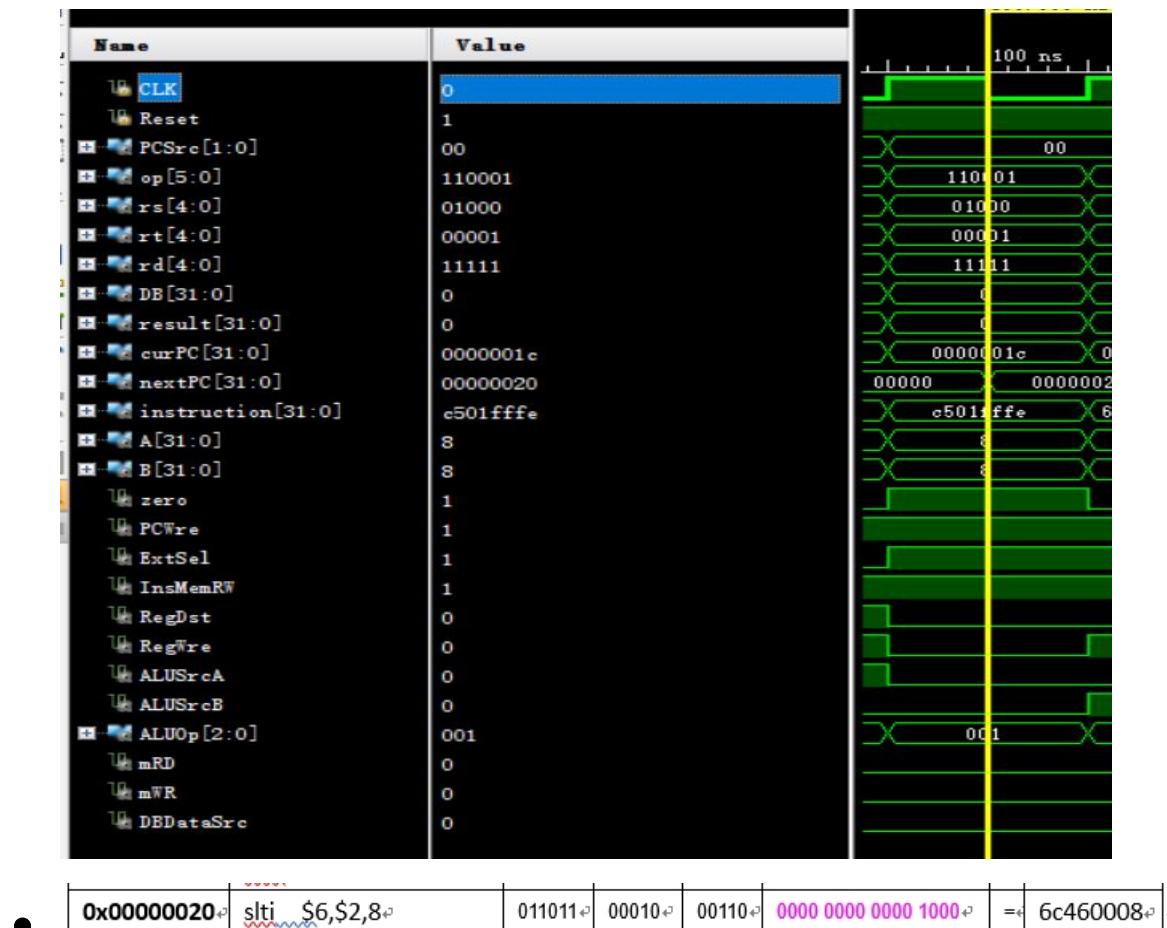
本条测试比较特殊，\$8 存的是数据 4 而\$1 寄存器存的是数据 8，二者不相等，将发生跳转，跳转到上一条指令 sll，将\$8 中的数据左移 1 位后其数据变成 8 再次执行到 bne 时，二者相等，继续顺序执行  
执行 bne \$8, \$1, -2:



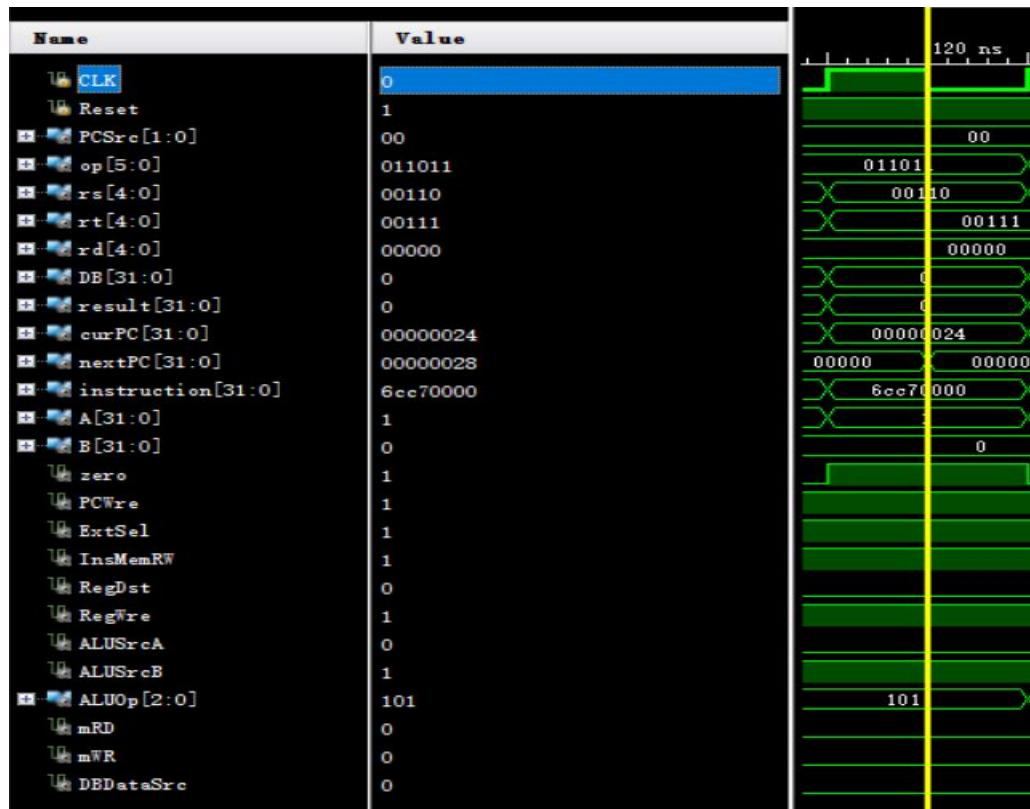
跳转回去执行 sll \$8, \$8, 1: (\$8 的数据左移一位)



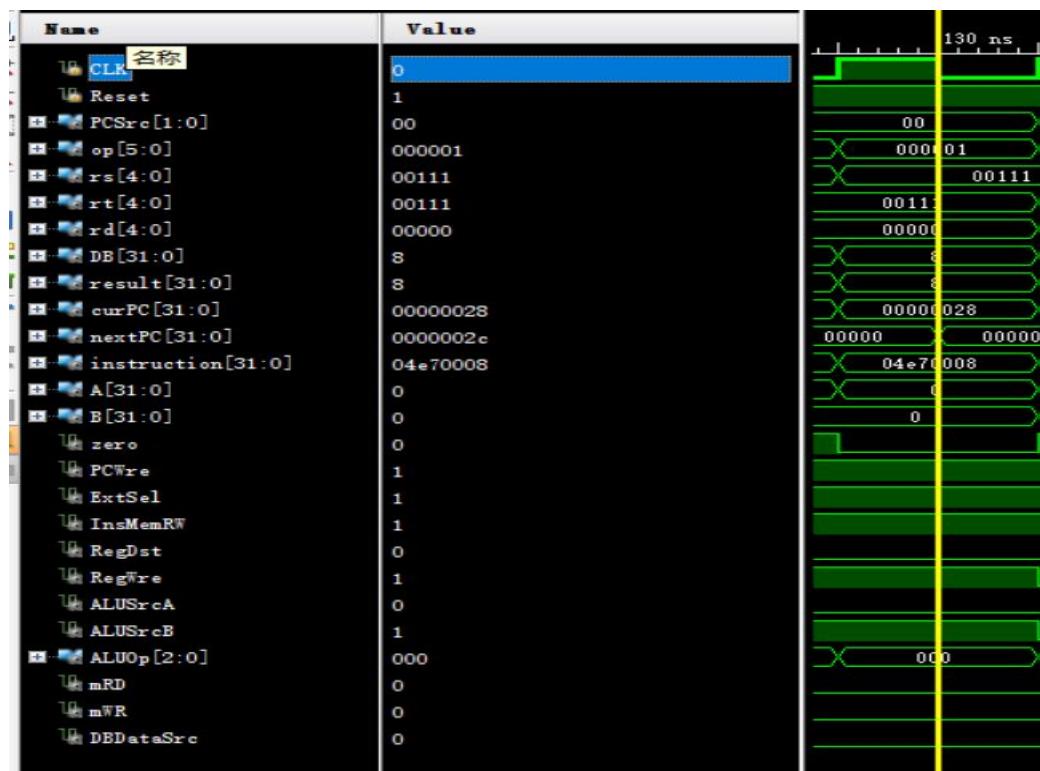
再次执行 bne \$8, \$1, -2: (二者相等, 不跳转)



●	0x00000024	slti \$7,\$6,0	011011	00110	00111	0000 0000 0000 0000	=	6cc70000
---	------------	----------------	--------	-------	-------	---------------------	---	----------

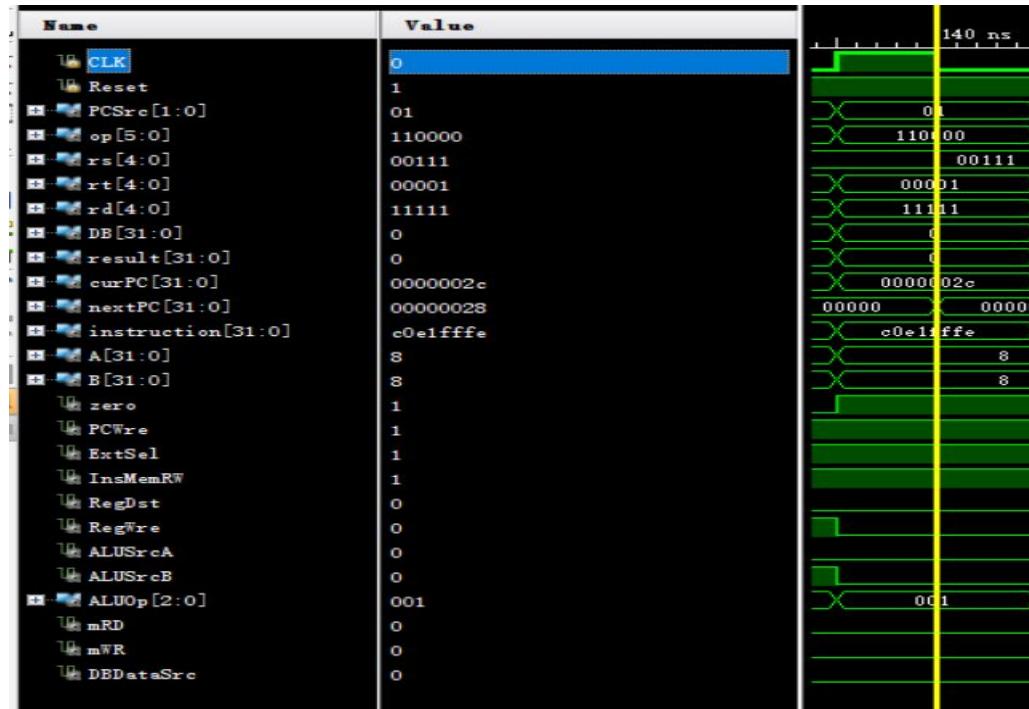


●	0x00000028	addi \$7,\$7,8	000001	00111	00111	0000 0000 0000 1000	=	04e70008
---	------------	----------------	--------	-------	-------	---------------------	---	----------

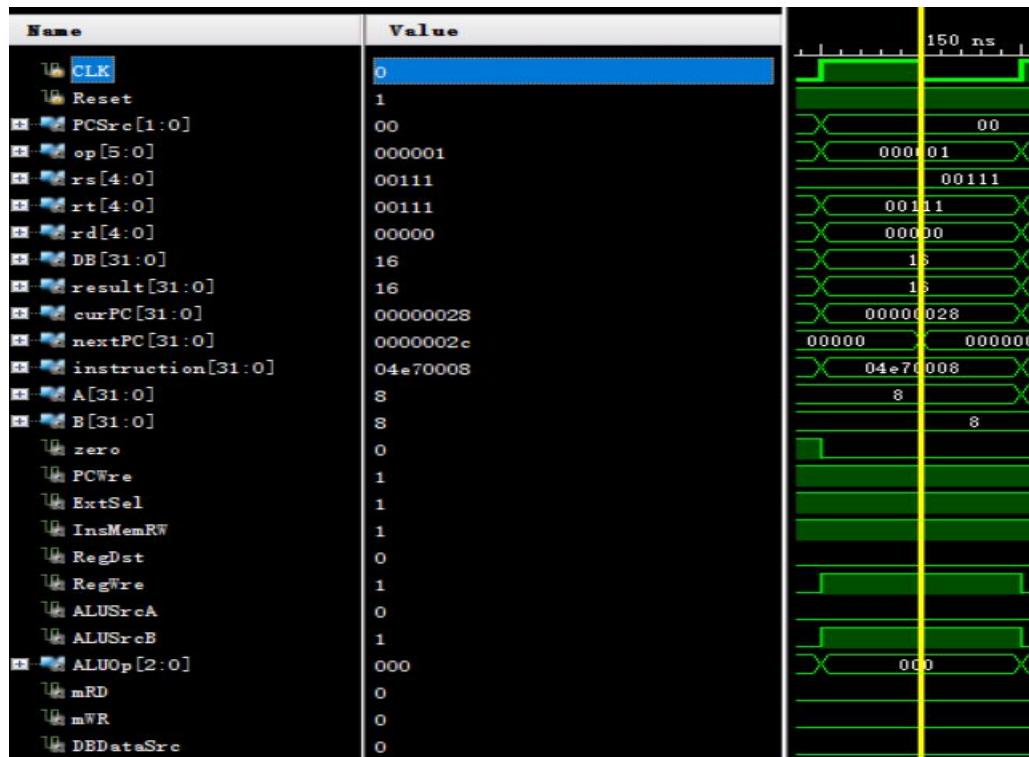


●	0x0000002C	beq \$7,\$1,-2 (= -28)	110000	00111	00001	1111 1111 1111 1110	=	c0e1fffe
---	------------	------------------------	--------	-------	-------	---------------------	---	----------

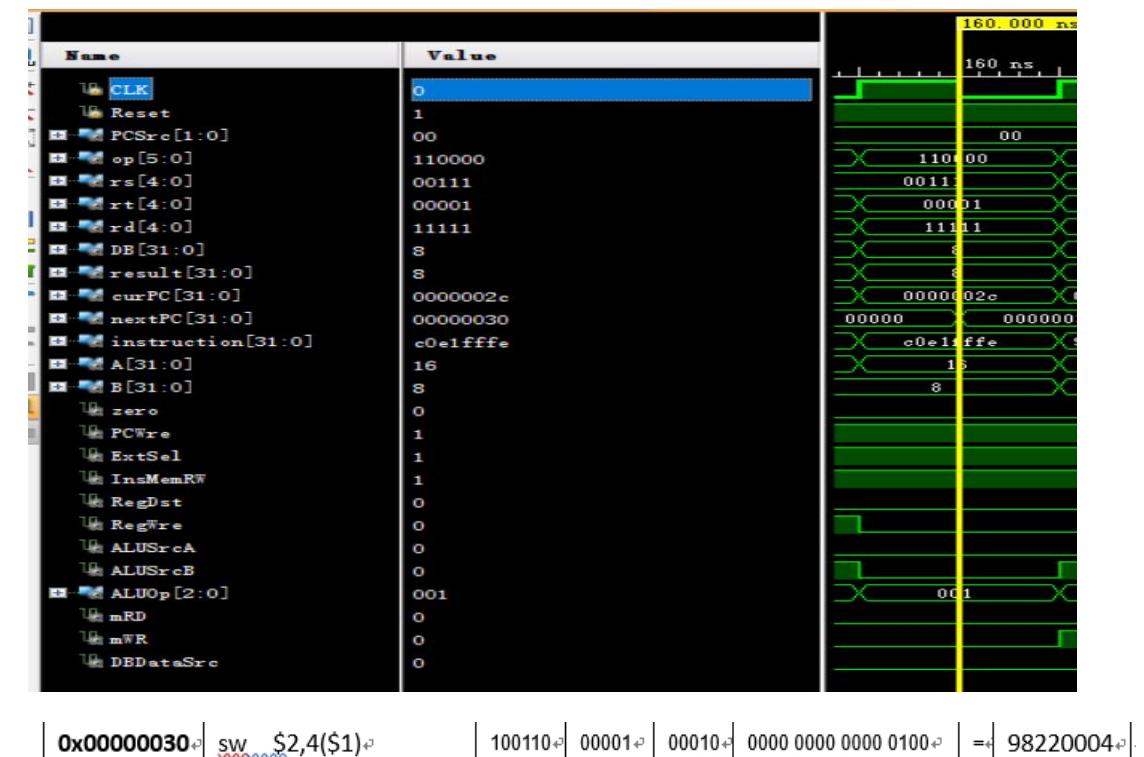
本指令中比较\$7 和\$1 的是否相等，若相等，则发生跳转，否则顺序执行，\$7 和\$1 在上述指令中其存储的数据值都为 8，故将发生跳转，跳转到当前指令的上面一条。执行 `beq $7, $1, -2:`



执行 `addi $7, $7, 8:` (\$7 存储的值被修改为了 16)

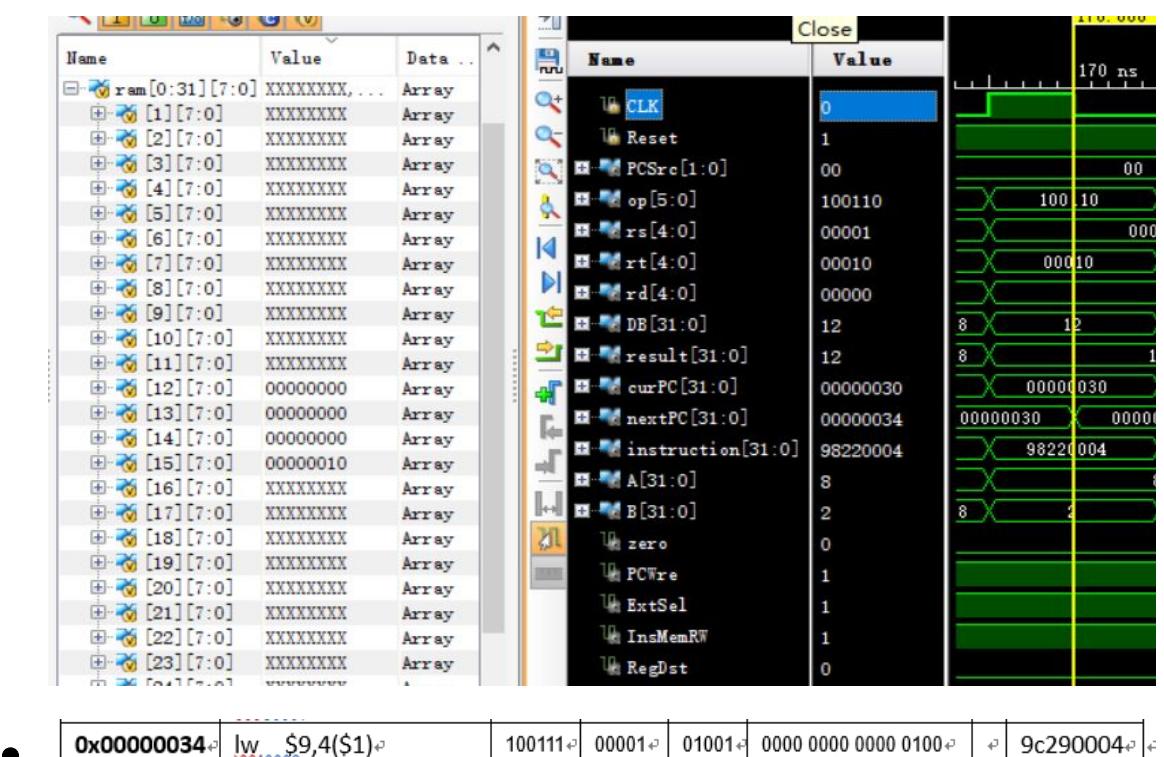


再次执行 `beq $7, $1, -2:` (此时二者不相等，不再发生跳转)

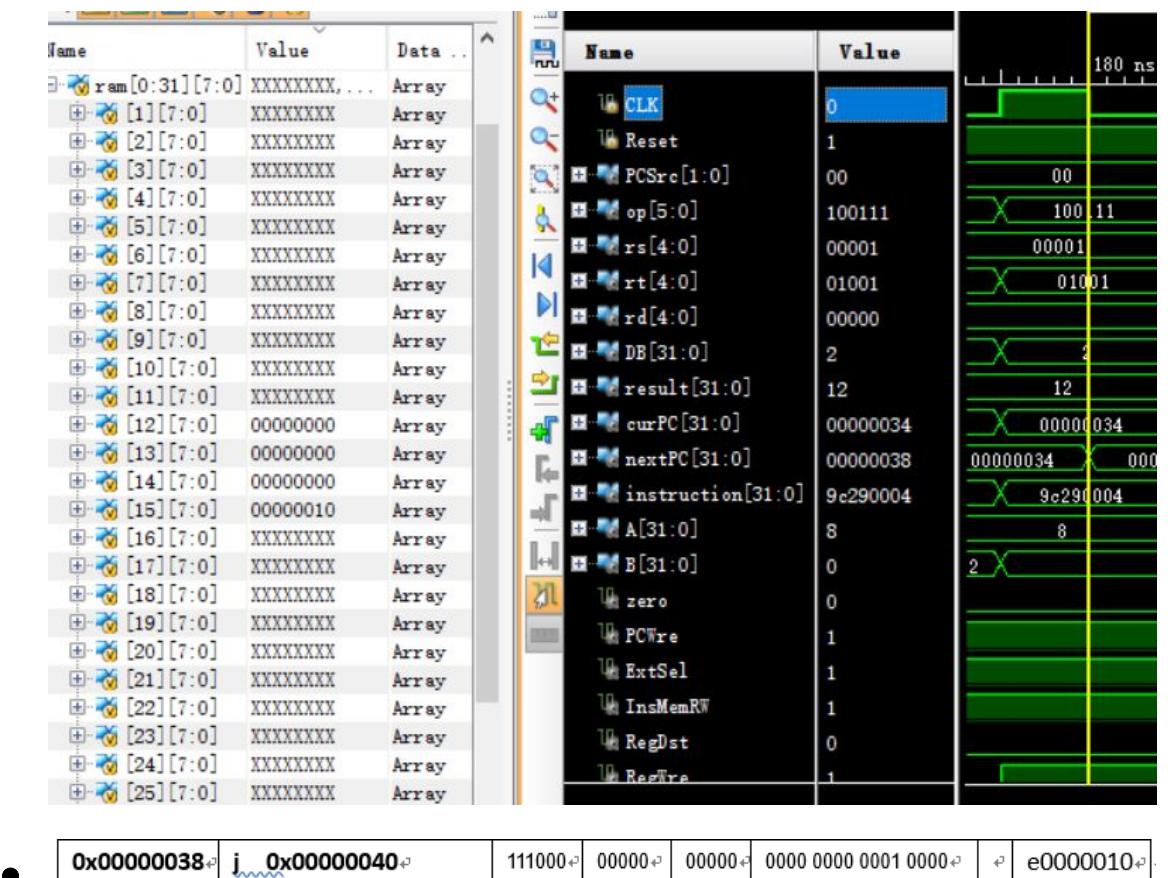


该指令执行将\$2寄存器的内容保存到\$1寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。\$2的内容为2，\$1的内容为8，即将\$2的内容存在数据寄存器中地址为12的地方，本次写入为大端。

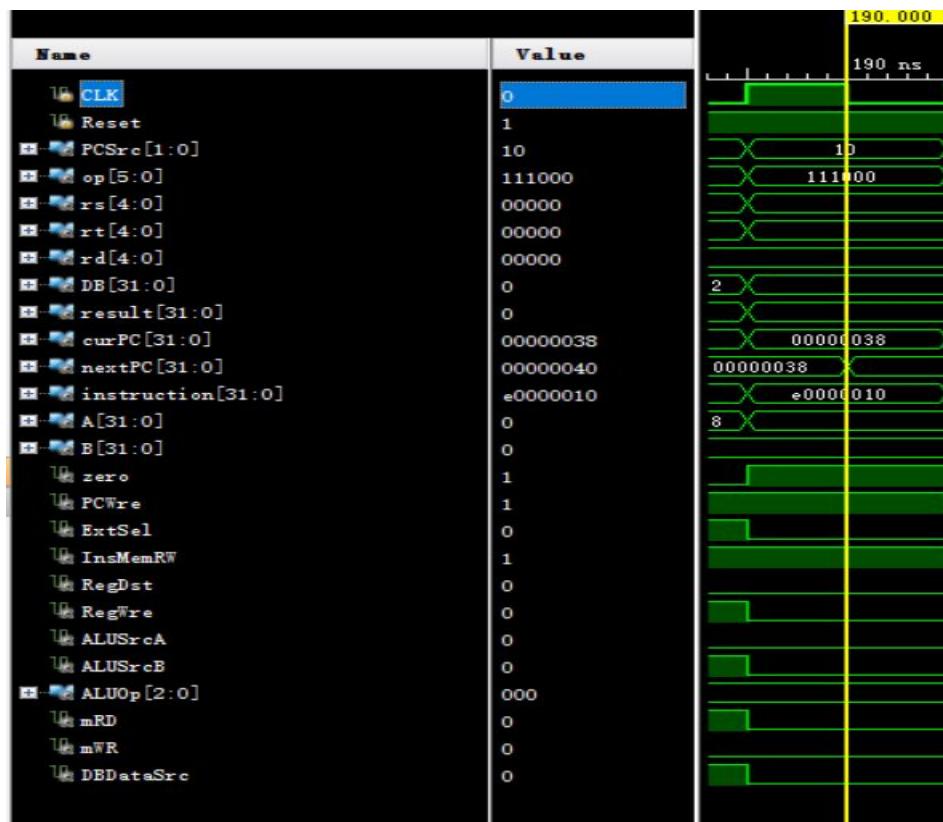
图中左半边为数据寄存器中存储的数据，可以观察到从第12到15有写入的数据，计算可以发现正是\$2的内容，验证了sw指令正确执行。



该指令执行将\$1 寄存器的内容和立即数扩展后的数相加作为地址的内存单元中的数，然后保存到\$9 中。\$1 的内容为 8，立即数，即将数据寄存器中地址为 12 开始的内容存在寄存器\$9 的地方。



本指令为跳转指令，地址为 0x00000040.



nextPC 为 0x00000040，证明地址为 0x0000003c 的被跳过，验证 j 指令是正确的。

● 0x00000040	Halt	111111	00000	00000	0000 0000 0000 0000	= fc000000
--------------	------	--------	-------	-------	---------------------	------------

停机指令。不改变 PC，PC 不发生改变。



④ 在 Basys3 板上运行所设计的 CPU

- a) 设计：额外创建一个新的 project，主要还是将所设计的单周期 cpu 实现的代码复制进去，添加了一个按键消抖的模块，同时添加一个新的顶层模块 showCPU，将消抖模块和 cpu 设计作为子模块。
- b) 文件结构：



c) 消抖模块：

- 消抖原因和目的：脉冲按键与电平按键通常采用机械式开关结构，按键信号在开关拨片与出点接触多次弹跳以后才会稳定，在按键过程中，可能出现多个脉冲，为了实现每次按键只执行一条指令，则必须对其进行消抖，防止一次产生多个脉冲，导致执行多条指令。
- 主要代码：

```

module Debounce(
    input clk,
    input key,
    output out
);

reg delay1, delay2, delay3;
assign out = delay1&delay2&delay3;
always@(posedge clk)//CLK 100M
begin
    delay1 <= key;
    delay2 <= delay1;
    delay3 <= delay2;
end
endmodule
  
```

d) 顶层模块

- 实例化 cpu 模块和消抖模块

```

Debounce Debounce(.clk(clock_100Mhz), .key(click), .out(CpuCLK));

SingleCycleCPUBasys SingleCycleCPUBasys(.CLK(CpuCLK),
                                         .Reset(Reset),
                                         .curPC(curPC),
                                         .nextPC(nextPC),
                                         .instruction(instruction),
                                         .op(op),
                                         .rs(rs),
                                         .rt(rt),
                                         .rd(rd),
                                         .DB(DB),
                                         .A(A),
                                         .B(B),
                                         .result(result),
                                         .PCSrc(PCSsrc));

```

### ii. 七段数码管显示

```

//扫描频率
always @(posedge clock_100Mhz)
begin
    refresh_counter <= refresh_counter + 1;
end
assign LED_activating_counter = refresh_counter[20:19];

//显示板块
always @(*)
begin
    case(LED_activating_counter)
        2'b00: begin
            Anode_Activate = 4'b0111;
            if(!S0 && !S1) begin
                LED_BCD = curPC[7:4];
            end
            else if(!S0 && S1) begin
                LED_BCD ={3'b000, rs[4]};
            end
            else if(S0 && S1) begin
                LED_BCD ={3'b000, rt[4]};
            end
            else begin
                LED_BCD = result[7:4];
            end
        end
    end

```

```
2'b01: begin
    Anode_Activate = 4'b1011;
    if(!S0 && !S1) begin
        LED_BCD = curPC[3:0];
    end
    else if(!S0 && S1) begin
        LED_BCD = rs[3:0];
    end
    else if(S0 && !S1) begin
        LED_BCD = rt[3:0];
    end
    else begin
        LED_BCD = result[3:0];
    end
end
2'b10: begin
    Anode_Activate = 4'b1101;
    if(!S0 && !S1) begin
        LED_BCD = nextPC[7:4];
    end
    else if(!S0 && S1) begin
        LED_BCD = A[7:4];
    end
    else if(S0 && !S1) begin
        LED_BCD = B[7:4];
    end
    else begin
        LED_BCD = DB[7:4];
    end
end
2'b11: begin
    Anode_Activate = 4'b1110;
    if(!S0 && !S1) begin
        LED_BCD = nextPC[3:0];
    end
    else if(!S0 && S1) begin
        LED_BCD = A[3:0];
    end
    else if(S0 && !S1) begin
        LED_BCD = B[3:0];
    end
    else begin
        LED_BCD = DB[3:0];
    end
end
```

```

        end
    endcase
end
// Cathode patterns of the 7-segment LED display
always @(*)
begin
    case(LED_BCD)
        4'b0000: LED_out = 7'b0000001; // "0"
        4'b0001: LED_out = 7'b1001111; // "1"
        4'b0010: LED_out = 7'b0010010; // "2"
        4'b0011: LED_out = 7'b0000110; // "3"
        4'b0100: LED_out = 7'b1001100; // "4"
        4'b0101: LED_out = 7'b0100100; // "5"
        4'b0110: LED_out = 7'b0100000; // "6"
        4'b0111: LED_out = 7'b0001111; // "7"
        4'b1000: LED_out = 7'b0000000; // "8"
        4'b1001: LED_out = 7'b0000100; // "9"
        4'b1010: LED_out = 7'b0001000; //A
        4'b1011: LED_out = 7'b1100000; //B
        4'b1100: LED_out = 7'b0110001; //C
        4'b1101: LED_out = 7'b1000010; //D
        4'b1110: LED_out = 7'b0110000; //E
        4'b1111: LED_out = 7'b0111000; //F
        default: LED_out = 7'b0000000; //不亮
    endcase
end

```

e) 添加约束文件

```

set_property PACKAGE_PIN W5 [get_ports clock_100Mhz]
set_property IOSTANDARD LVCMOS33 [get_ports clock_100Mhz]
set_property PACKAGE_PIN R2 [get_ports SO]
set_property IOSTANDARD LVCMOS33 [get_ports SO]
set_property PACKAGE_PIN T1 [get_ports S1]
set_property IOSTANDARD LVCMOS33 [get_ports S1]
set_property PACKAGE_PIN V17 [get_ports Reset]
set_property IOSTANDARD LVCMOS33 [get_ports Reset]
set_property PACKAGE_PIN U18 [get_ports click]
set_property IOSTANDARD LVCMOS33 [get_ports click]
set_property PACKAGE_PIN W7 [get_ports {LED_out[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[6]}]
set_property PACKAGE_PIN W6 [get_ports {LED_out[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[5]}]
set_property PACKAGE_PIN US [get_ports {LED_out[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[4]}]
set_property PACKAGE_PIN V8 [get_ports {LED_out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[3]}]
set_property PACKAGE_PIN U5 [get_ports {LED_out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[2]}]
set_property PACKAGE_PIN V5 [get_ports {LED_out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[1]}]
set_property PACKAGE_PIN U7 [get_ports {LED_out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED_out[0]}]
set_property PACKAGE_PIN U2 [get_ports {Anode_Activate[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[0]}]
set_property PACKAGE_PIN U4 [get_ports {Anode_Activate[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[1]}]
set_property PACKAGE_PIN V4 [get_ports {Anode_Activate[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[2]}]
set_property PACKAGE_PIN W4 [get_ports {Anode_Activate[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Anode_Activate[3]}]

```

f) 结果显示（程序测试段与仿真测试的时候为同一段）

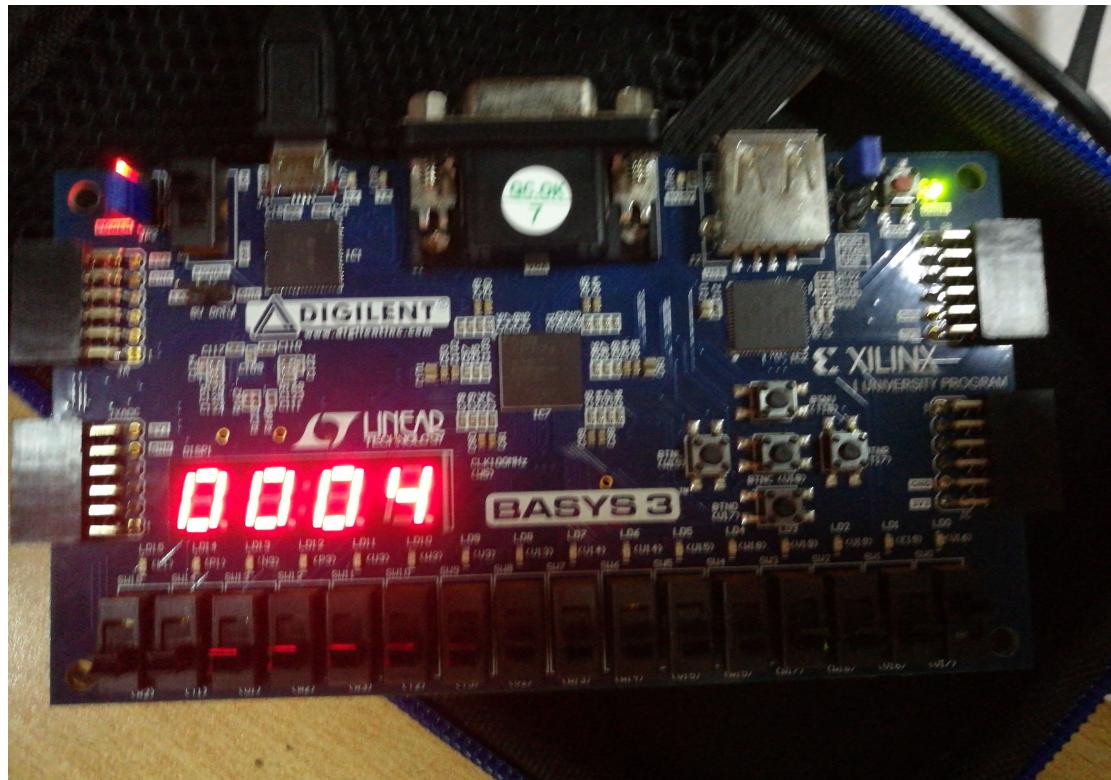
(BasyS 运行结果与仿真所得相匹配，此处就不具体上所有指令的图了)

初始时候当前 pc 和下一个 pc 都为空

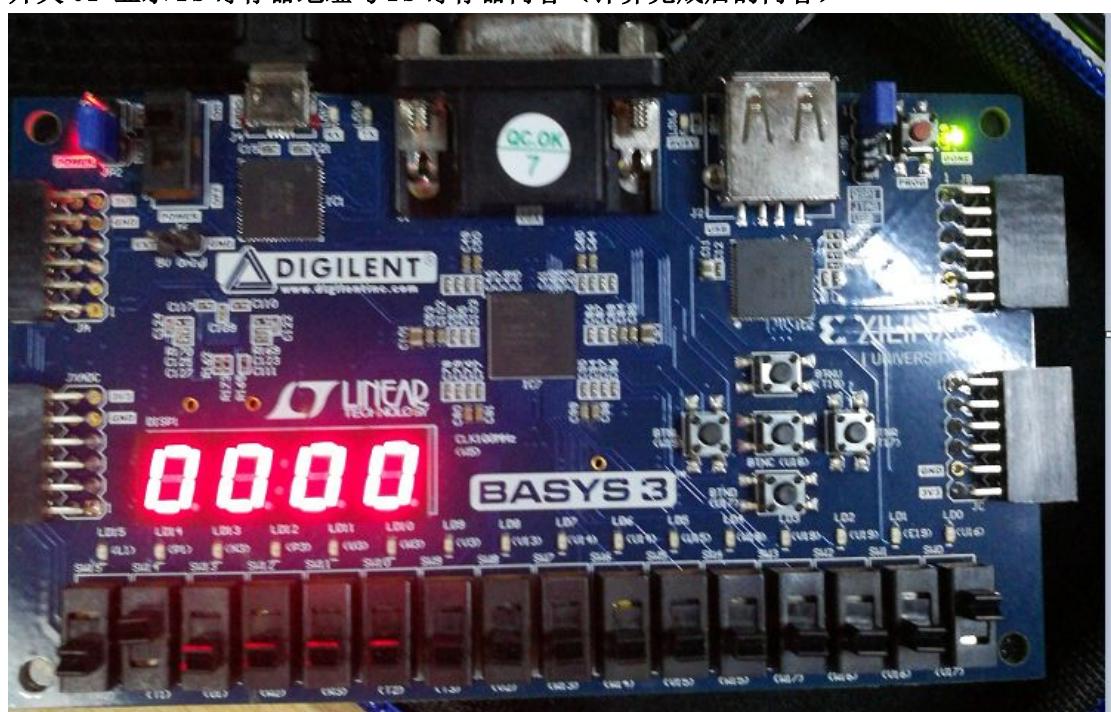
按下按键，获得当前 pc 以及下一个 pc

第一条指令：addi \$1,\$0,8

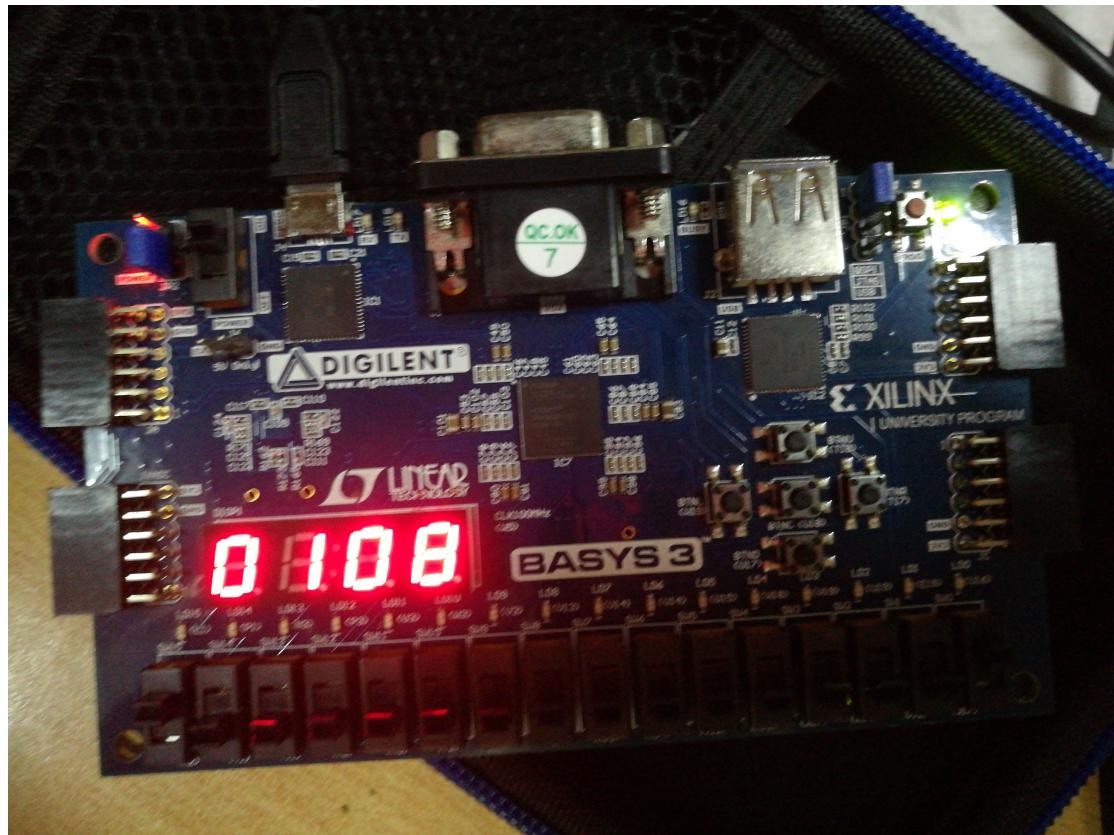
开关 00 显示当前 pc 以及下一个 pc



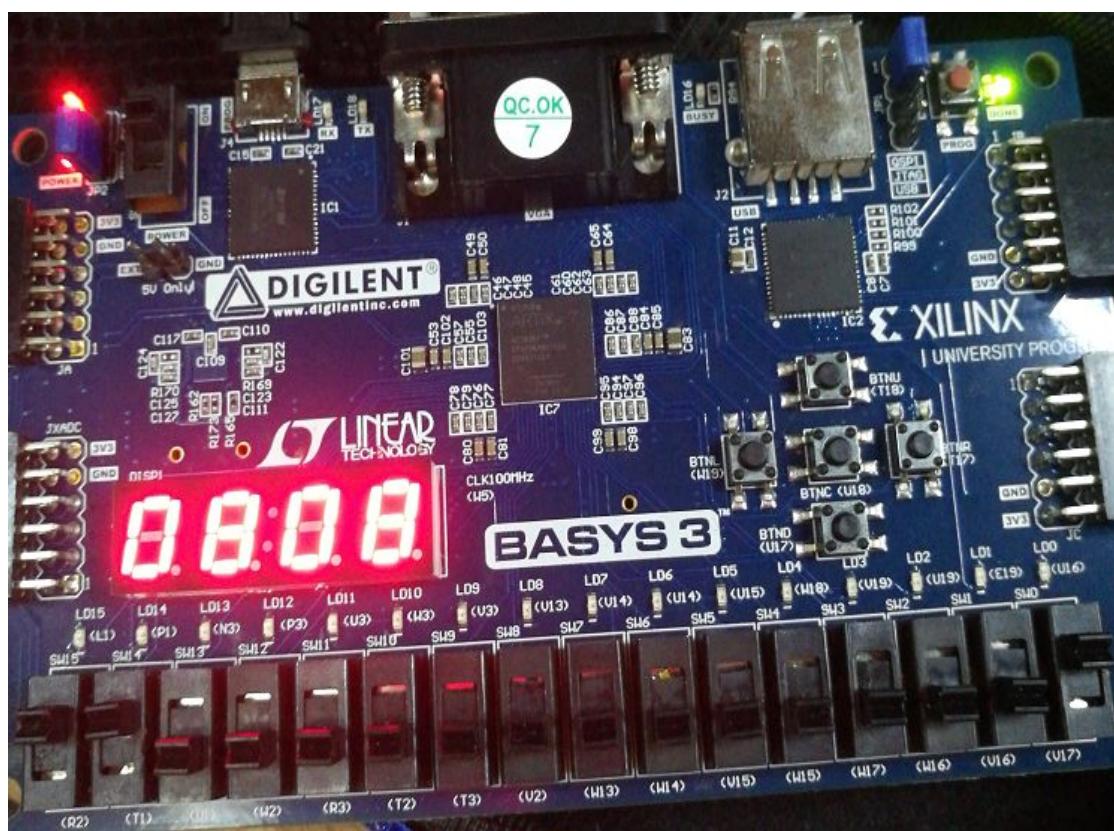
开关 01 显示 rs 寄存器地址与 rs 寄存器内容（计算完成后的内容）



开关 10 显示 rt 寄存器地址与 rt 寄存器内容（计算完成后的内容）



开关 11，显示 ALU 结果输出与 DB 总线输出 s



## 六. 实验心得

本次实验中遇到的问题比较多。首先是关于 CPU 的设计，其次就是 verilog 语言。一开始不知道如何实现，感觉无从下手。主要通过分析实验原理中的图 2 单周期 CPU 数据通路和控制线路图，分析各种指令的处理过程，学会将 CPU 内各个部分模块化，各个模块分别实现一定的功能，然后通过相对应的控制信号连接起来，这样就实现 cpu 设计。完成模块的划分以后，按照先前对每个模块功能预设进行完成，但是每个模块的敏感信号的选择还是很重要的，有些模块程序要在时钟信号上升沿触发，而有些模块要在时钟信号的下降沿触发，有些则将电平信号作为敏感信号，每个模块里面的敏感信号的选择都十分的重要，一开始没有太过注意导致出现了很多的问题，后面重新仔细的想指令的处理过程，重新规定了各个模块 always@里面的敏感信号。

其次就是 verilog 里面的 wire 和 reg 两种变量类型，感觉这是比较大的坑。一开始不了解两者的区别，导致后面一堆报错。现在大致的清楚了二者的区别，wire 主要起信号间连接的作用，例如顶层模块中，需要将各个模块连接起来，这时候只能用 wire 连接，不能使用 reg，wire 不保存状态，它的值的随时可以改变，不受时钟信号的影响，而 reg 则是寄存器的抽象表达，可以用于存储数值，例如指令寄存器和寄存器组以及数据寄存器里面的存储器必须为 reg 类型，用于保留数据。其次 wire 类型只能通过 assign 进行赋值，而 reg 类型只能在 always 里面被赋值，而涉及到 always 又有阻塞赋值和非阻塞赋值这个大坑，一开始也不知道怎么弄，就混用了，后面也是出现乱七八糟的问题，后面仔细学习了一下，敏感信号为电平信号的时候，采用阻塞赋值(=)，而敏感信号为时序信号的时候，采用非阻塞赋值(<=)。

再者就是烧板的时候的消抖问题。一开始没有进行消抖，然后总是按一下运行了几条指令，后面上网学习了一下如何消抖，顺利的解决了该问题。

还有比较疑惑的问题就是使用 vivado 进行 Implementation 的时候，有时候进行 Running place\_design 这一部分的时候就一直在此处运行，没有任何进度了，网上也没有合理的解释，然后新新建个项目，将里面的代码复制进去又可以正常的运行了，这个问题目前尚未解决。

本次单周期 CPU 设计实验，将计组理论课上所讲的指令处理过程自己重复并实现了单周期 CPU 的设计，加深了 CPU 处理指令过程理解，之前由于计组理论学的不是特别清楚，本次实验加深了印象，也更加了解每条指令的处理过程以及单周期 CPU 是如何工作的，同时本次实验也更加了解 verilog 语言，之前学的懵懵懂懂的，最重要的是学会模块化，将一项工作分成多个模块进行完成，先简化成小部分，然后再将其组合起来。