



中山大學  
SUN YAT-SEN UNIVERSITY

# 《计算机组成原理实验》 实验报告

(项目二)

学 院 名 称 : 计算机学院

专业 (班级) : 计算机科学与技术

学 生 姓 名 : 熊昊翔

学 号 : 23336265

时 间 : 2024 年 12 月 30 日

# 项目二：多周期CPU设计与实现

## 一.实验目的

1. 认识和掌握多周期数据通路图的构成、原理及其设计方法；
2. 掌握多周期CPU的实现方法，代码实现方法；
3. 编写一个编译器，将MIPS汇编程序编译为二进制机器码；
4. 掌握多周期CPU的测试方法；
5. 掌握多周期CPU的实现方法。

## 二.实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：

### ==>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) sub rd, rs, rt

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

完成功能：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。

### ==>逻辑运算指令

(4) and rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能：rd ← rs & rt；逻辑与运算。

(5) andi rt, rs, immediate

010001	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

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

(6) ori rt, rs, immediate

010010	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。

(7) xori rt, rs, immediate

010011	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

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

### ==>移位指令

(8) 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

(9) slti rt, rs, immediate 带符号

100110	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。

(10) slt rd, rs, rt 带符号

100111	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

(11) sw rt, immediate(rs)

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

功能:  $\text{memory}[rs + (\text{sign-extend})immediate] \leftarrow rt$ 。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

(12) lw rt, immediate(rs)

110001	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能:  $rt \leftarrow \text{memory}[rs + (\text{sign-extend})immediate]$ 。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数, 然后保存到 rt 寄存器中。

### ==>分支指令

(13) beq rs, rt, immediate (说明: immediate 从 pc+4 开始和转移到的指令之间间隔条数)

110100	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: if(rs=rt)  $pc \leftarrow pc + 4 + ((\text{sign-extend})immediate \ll 2)$  else  $pc \leftarrow pc + 4$ 。

(14) bne rs, rt, immediate (说明: immediate 从 pc+4 开始和转移到的指令之间间隔条数)

110101	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  $pc \leftarrow pc + 4$ 。

(15) bltz rs, immediate

110110	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

(16) j addr

111000	addr[27:2]
--------	------------

功能:  $pc \leftarrow \{(pc+4)[31:28], \text{addr}[27:2], 2'b00\}$ , 跳转。

说明: 由于 MIPS32 的指令代码长度占 4 个字节, 所以指令地址二进制数最低 2 位均

为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

(17) jr rs

111001	rs(5 位)	未用	未用	reserved
--------	---------	----	----	----------

功能：pc ← rs，跳转。

==>调用子程序指令

(18) jal addr

111010	addr[27:2]
--------	------------

功能：调用子程序，pc ← {(pc+4)[31:28],addr[27:2],2'b00}; \$31←pc+4，返回地址设置；子程序返回，需用指令 jr \$31。跳转地址的形成同 j addr 指令。

==>停机指令

(19) halt (停机指令)

111111	00000000000000000000000000000000(26 位)
--------	--

不改变 pc 的值，pc 保持不变。

### 三.实验原理

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

#### 1. CPU在处理指令时要经过的步骤

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。

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。

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。

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。

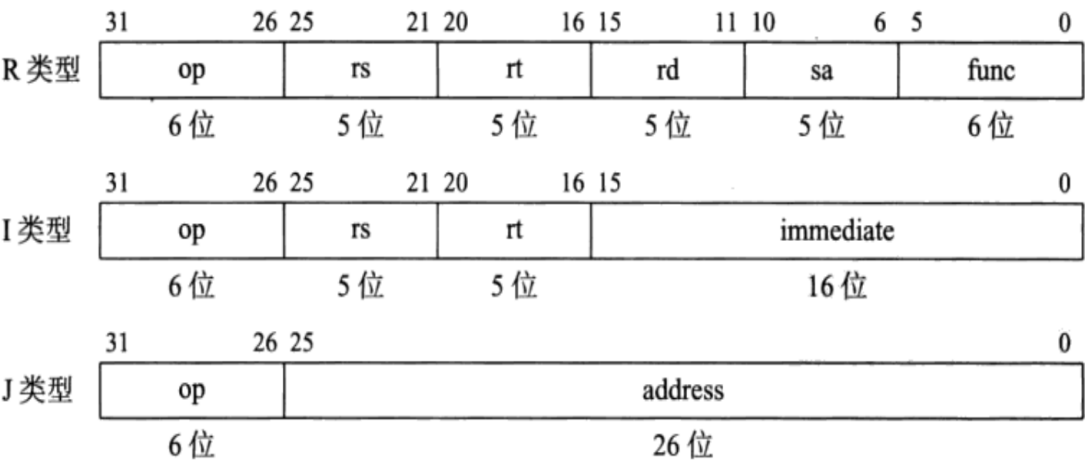
结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器

中。单周期CPU，是在一个时钟周期内完成这五个阶段的处理。



单周期CPU指令处理过程

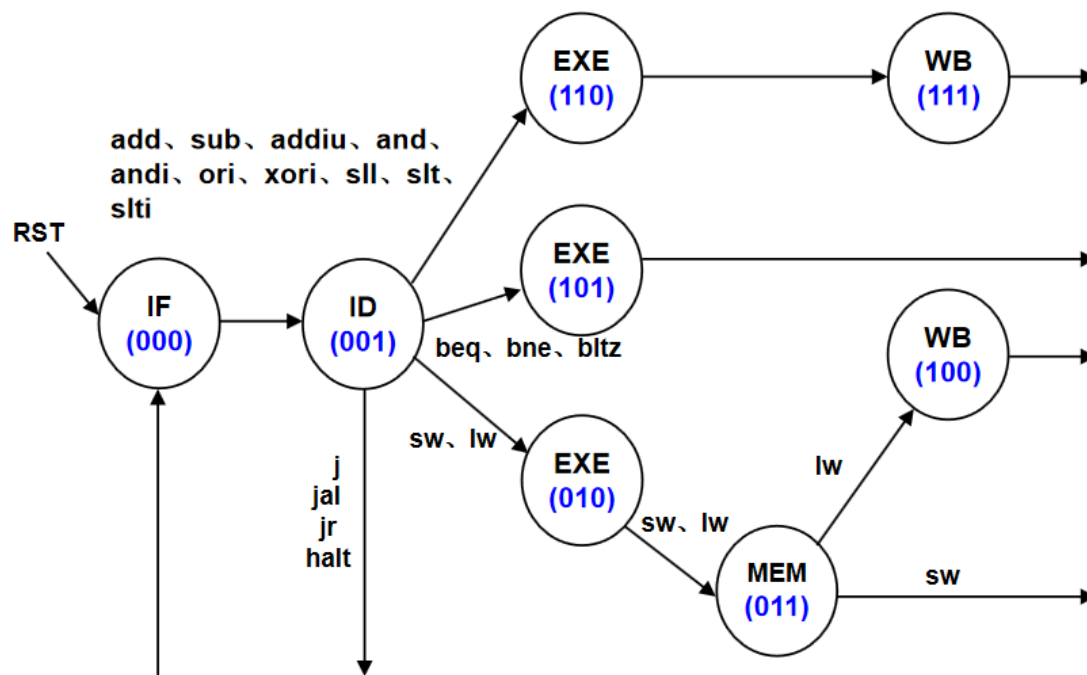
2. MIPS指令的三种格式:



- op: 操作码;
- rs: 只读。为第 1 个源操作数寄存器;
- 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;
- rd: 只写。为目的操作数寄存器;
- sa: 位移量 (shift amt) , 移位指令用于指定移多少位;
- func: 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 (Laod)/数据保存 (Store) 指令的数据地址字节偏移量和分支指令中PC的有符号偏移量;
- address: 地址

3. 多周期 CPU 状态转移

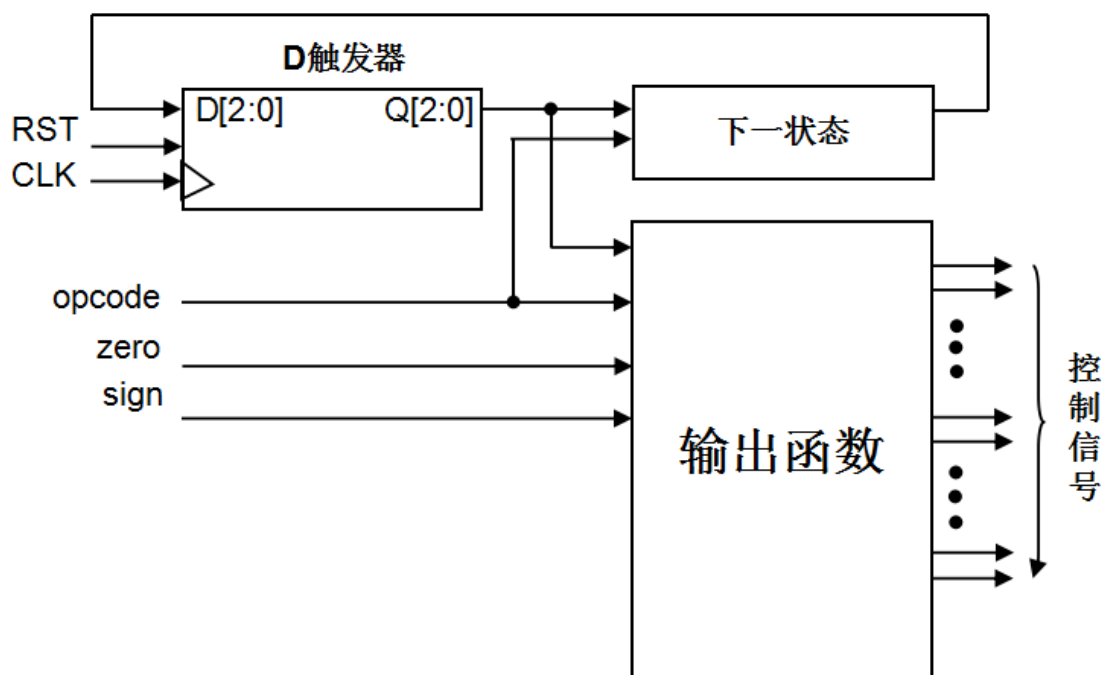
状态的转移有的是无条件的, 例如从 sIF 状态转移到 sID 就是无条件的; 有些是有条件的, 例如 sEXE 状态之后不止一个状态, 到底转向哪个状态由该指令功能, 即指令操作码决定。 每个状态代表一个时钟周期



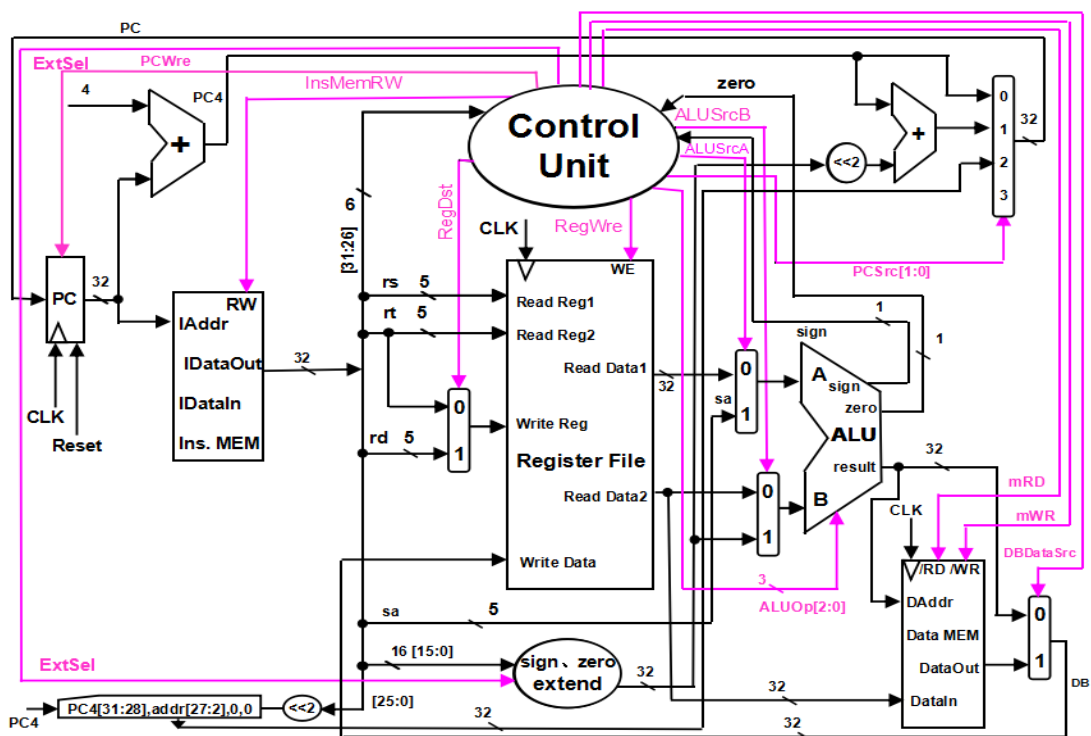
多周期 CPU 状态转移图

#### 4. 多周期 CPU 控制部件的原理结构图

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



5. 单周期 CPU 数据通路和控制线路



上图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。

图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表一

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 ,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	来自 sign 或 zero 扩展的立即数，相关指令：addi、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



<b>RegWre</b>	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
<b>InsMemRW</b>	写指令存储器	读指令存储器(Ins. Data)
<b>mRD</b>	输出高阻态	读数据存储器，相关指令：lw
<b>mWR</b>	无操作	写数据存储器，相关指令：sw
<b>RegDst</b>	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
<b>ExtSel</b>	(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
<b>PCSrc[1..0]</b>	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
<b>ALUOp[2..0]</b>	ALU 8 种运算功能选择(000-111)， 看表二	

表2

ALUOp[2..0]	功能	描述
<b>000</b>	$Y = A + B$	加
<b>001</b>	$Y = A - B$	减
<b>010</b>	$Y = B \ll A$	B 左移 A 位
<b>011</b>	$Y = A \vee B$	或
<b>100</b>	$Y = A \wedge B$	与
<b>101</b>	$Y = (A < B) ? 1 : 0$	比较 A<B 不带 符号
<b>110</b>	$Y = (((A < B) \&\& (A[31] == B[31])) \vee (((A[31] == 1 \&\& B[31] == 0))) ? 1 : 0$	比较 A<B 带符 号
<b>111</b>	$Y = A \oplus B$	异或

### 6.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 1 读

WR, 数据存储器写控制信号, 为 1 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

#### 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

## 汇编器

汇编器的作用是把汇编代码转换为机器码，只需要读取每一条指令，并且根据指令的格式构造机器码即可

```
import re
def reg_to_num(reg):
    regid = int(reg.lstrip('$'))
    result = bin(regid)[2:].rjust(5, '0')
    return result
def type_r(opcode, operands):
    rs = reg_to_num(operands[1])
    rt = reg_to_num(operands[2])
    rd = reg_to_num(operands[0])
    return opcode + rs + rt + rd + 11 * '0'
def type_i(opcode, operands):
    rs = reg_to_num(operands[1])
    rt = reg_to_num(operands[0])
    immediate = int(operands[2])
    if immediate < 0:
        immediate += 2**16
    immediate = bin(immediate)[2:].rjust(16, '1')
    else:
        immediate = bin(immediate)[2:].rjust(16, '0')
    return opcode + rs + rt + immediate
def type_j(opcode, operands):
    addr = bin(int(operands[0], 16))
    addr = str(addr[2:-2]).rjust(26, '0')
    return opcode + addr
def asm_compile(instruction, operands):
    if instruction == 'add':
        result = type_r('00000', operands)
    elif instruction == 'sub':
        result = type_r('00001', operands)
    elif instruction == 'addiu':
        result = type_i('00010', operands)
    elif instruction == 'and':
        result = type_r('01000', operands)
    elif instruction == 'andi':
        result = type_i('01001', operands)
    elif instruction == 'ori':
        result = type_i('01010', operands)
    elif instruction == 'xori':
        result = type_i('01011', operands)
```

```

elif instruction == 'sll':
opcode = '011000'
rt = reg_to_num(operands[1])
rd = reg_to_num(operands[0])
sa = bin(int(operands[2]))[2:].rjust(5, '0')
result = opcode + 5*'0' + rt + rd + sa + 6*'0'
elif instruction == 'slti':
result = type_i('100110', operands)
elif instruction == 'slt':
result = type_r('100111', operands)
elif instruction == 'sw':
opcode = '110000'
tempRegex = re.compile('([0-9])\(((.+))\')')
rs = reg_to_num(tempRegex.search(operands[1]).group(2))
rt = reg_to_num(operands[0])
immediate = int(tempRegex.search(operands[1]).group(1))
if immediate < 0:
immediate += 2 ** 16
immediate = bin(immediate)[2:].rjust(16, '1')
else:
immediate = bin(immediate)[2:].rjust(16, '0')
result = opcode + rs + rt + immediate
elif instruction == 'lw':
opcode = '110001'
tempRegex = re.compile('([0-9])\(((.+))\')')
rs = reg_to_num(tempRegex.search(operands[1]).group(2))
rt = reg_to_num(operands[0])
immediate = int(tempRegex.search(operands[1]).group(1))
if immediate < 0:
immediate += 2 ** 16
immediate = bin(immediate)[2:].rjust(16, '1')
else:
immediate = bin(immediate)[2:].rjust(16, '0')
result = opcode + rs + rt + immediate
elif instruction == 'beq':
opcode = '110100'
rs = reg_to_num(operands[0])
rt = reg_to_num(operands[1])
immediate = int(operands[2])
if immediate < 0:
immediate += 2 ** 16
immediate = bin(immediate)[2:].rjust(16, '1')
else:
immediate = bin(immediate)[2:].rjust(16, '0')

```

```

result = opcode + rs + rt + immediate
elif instruction == 'bne':
opcode = '110101'
rs = reg_to_num(operands[0])
rt = reg_to_num(operands[1])
immediate = int(operands[2])
if immediate < 0:
immediate += 2 ** 16
immediate = bin(immediate)[2:].rjust(16, '1')
else:
immediate = bin(immediate)[2:].rjust(16, '0')
result = opcode + rs + rt + immediate
elif instruction == 'bltz':
opcode = '110110'
rs = reg_to_num(operands[0])
immediate = int(operands[1])
if immediate < 0:
immediate += 2 ** 16
immediate = bin(immediate)[2:].rjust(16, '1')
else:
immediate = bin(immediate)[2:].rjust(16, '0')
result = opcode + rs + '00000' + immediate
elif instruction == 'j':
result = type_j('111000', operands)
elif instruction == 'jr':
opcode = '111001'
rs = reg_to_num(operands[0])
result = opcode + rs + '0'*21
elif instruction == 'jal':
result = type_j('111010', operands)
elif instruction == 'halt':
opcode = '111111'
return opcode + '0'*26
return result
if __name__ == '__main__':
asmFile = open('D:\\MYCPU\\计算机组成原理\\计组实验\\5.实验三:
MulticycleCPU\\test_code.asm',
mode='r', encoding='UTF-8')
regex = re.compile('([a-zA-Z]+) +(\\.+)')
count = 1
for asmCode in asmFile:
asmCode = asmCode.strip()
match = regex.search(asmCode)
if match is not None:

```

```

instruction = match.group(1)
operands = match.group(2).split(',')
else:
instruction = asmCode
operands = []
machineCode = asm_compile(instruction, operands)
#print(str(count).rjust(2), ':', end='')
print(machin

```

首先根据数据通路，用面向对象的思想可将整个CPU程序分成PC，ControlUnit，DataMemory，InstructionMemory，RegisterFile，SignZeroExtend，ALU和顶层模块SingleCircleCPU几个模块，另外，多周期CPU在单周期CPU的基础上增加了ChangeState，IR,DR和mux2to1模块，其他模块与单周期CPU几乎没有差别，所以各模块的测试正确性环节（在单周期报告中有）省略。

下面是各个模块的详细情况：

## ALU（算术逻辑单元）

Input:

```

ReadData1,      // 第一个操作数，来自寄存器文件
ReadData2,      // 第二个操作数，来自寄存器文件
Ext,            // 扩展后的立即数（用于某些指令）
Sa,            // 移位量（通常由shamt字段提供）
ALUOp,         // 指定要执行的操作类型
ALUSrcA,       // 控制信号：选择InA的来源（ReadData1或Sa）
ALUSrcB,       // 控制信号：选择InB的来源（ReadData2或Ext）

```

Output:

```

zero,          // 输出信号：当Result为0时置1
Result,        // 运算结果输出
sign           // 符号

```

实现代码：

```

`timescale 1ns / 1ps
module ALU(
    input [31:0] ReadData1,      // 第一个操作数，来自寄存器文件
    input [31:0] ReadData2,      // 第二个操作数，来自寄存器文件
    input [31:0] Ext,            // 扩展后的立即数（用于某些指令）

```

```

    input [4:0] Sa,                // 移位量 (通常由 shamt 字段提供)
    input [2:0] ALUOp,            // 指定要执行的操作类型
    input ALUSrcA,                // 控制信号: 选择 InA 的来源 (ReadData1 或
Sa)
    input ALUSrcB,                // 控制信号: 选择 InB 的来源 (ReadData2 或
Ext)

    output wire zero,             // 输出信号: 当 Result 为 0 时置 1
    output reg [31:0] Result,     // 运算结果输出
    output wire sign              // 符号
);

// 内部连接线, 用于表示实际参与运算的操作数
wire [31:0] InA;                 // 实际参与运算的第一个操作数
wire [31:0] InB;                 // 实际参与运算的第二个操作数

// 根据控制信号 ALUSrcA 选择 InA 的来源:
assign InA = ALUSrcA ? {{27{1'b0}}}, Sa : ReadData1;

// 根据控制信号 ALUSrcB 选择 InB 的来源:
assign InB = ALUSrcB ? Ext : ReadData2;

// 计算零标志位 zero: 如果 Result 为全 0, 则 zero 置 1; 否则为 0
assign zero = (Result == 32'b0) ? 1 : 0;
assign sign = Result[31];
// 组合逻辑块, 当任何输入发生变化时自动重新计算 Result
always @(*) begin
    case(ALUOp) // 根据 ALUOp 的值选择执行不同的运算功能
        3'b000 :
            Result = InA + InB;                // 加法操作
        3'b001 :
            Result = InA - InB;                // 减法操作
        3'b010 :
            Result = InB << InA[4:0];          // 左移逻辑操作
        3'b011 :
            Result = InA | InB;                // 或运算
        3'b100 :
            Result = InA & InB;                // 与运算
        3'b101 :
            Result = (InA < InB) ? 32'b1 : 32'b0; // 无符号小于比较
        3'b110 :
            Result = (((InA < InB) && (InA[31] == InB[31])) ||
                ((InA[31] == 1'b1) && (InB[31] == 1'b0))) ?
32'b1 : 32'b0; // 带符号小于比较
    endcase
end

```

```

        3'b111:
            Result = InA ^ InB;                // 异或运算
        default:
            Result = 32'h00000000;            // 默认情况, 将 Result 清零
        endcase
    end
endmodule

```

## PC（程序计数器）

Input:

CLK,           // 时钟信号

Reset,         // 复位信号: 当为低电平时将PC置零

PCWre,         // 写使能信号: 为高电平时允许更新PC值

PCSrc,         // 程序计数器源选择信号

Immediate,    // 立即数

JumpPC,        // 跳转目标地址

Output:

Address,       // 当前的PC值

nextPC,        // 下一个PC值

PC4            // 当前PC值的高4位

实现代码:

```

`timescale 1ns / 1ps

module PC(
    input CLK, // 时钟信号
    input Reset, // 复位信号: 当为低电平时将 PC 置零
    input PCWre, // 写使能信号: 为高电平时允许更新 PC 值
    input [1:0] PCSrc, // 程序计数器源选择信号
    input signed [15:0] Immediate, // 立即数
    input [31:0] JumpPC, // 跳转目标地址

    output reg signed [31:0] Address, // 当前的 PC 值
    output [31:0] nextPC, // 下一个 PC 值
    output [3:0] PC4 // 当前 PC 值的高 4 位
);

```



```

// 计算下一个 PC 值
// 如果 PCSrc[0]为 1, 则执行相对跳转: Address + 4 + (Immediate << 2)
// 如果 PCSrc[1]为 1, 则执行绝对跳转: JumpPC
// 否则顺序执行下一条指令: Address + 4
assign nextPC = (PCSrc[0]) ? Address + 4 + (Immediate << 2) :
((PCSrc[1]) ? JumpPC : Address + 4);

// 提取当前 PC 值的高 4 位
assign PC4 = Address[31:28];

// 在 CLK 或 Reset 的下降沿触发
always @(negedge CLK or negedge Reset) begin
    if(Reset == 0) begin
        // 如果 Reset 为低电平, 将 PC 置零
        Address = 0;
    end else if(PCWre) begin // 只有在 PCWre 为高电平时才允许更改 PC 值
        if(PCSrc[0]) begin
            // 如果 PCSrc[0]为 1, 执行相对跳转
            Address = Address + 4 + (Immediate << 2);
        end else if(PCSrc[1]) begin
            // 如果 PCSrc[1]为 1, 执行绝对跳转
            Address = JumpPC;
        end else begin
            // 否则, 顺序执行下一条指令
            Address = Address + 4;
        end
    end
end
endmodule

```

## InstructionMemory(指令存储器)

这里需要提前用一个txt文件来存放我们要进行读取的指令，指令文本中的内容在“测试表格”文档中

已经给出：

地址	汇编程序	指令代码					16 进制数代码	
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)			
0x00000000 0	addiu \$1,\$0,8	001001	00000	00001	00000000 00001000	=	24010008	

0x00000004	ori \$2,\$0,2	001101	00000	00010	00000000 00000010	=	34020002
0x00000008	xori \$3,\$2,8	001110	00010	00011	00000000 00001000	=	38430008
0x0000000C	sub \$4,\$3,\$1	000000	00011	00001	00100000 00100010	=	00612022
0x00000010	and \$5,\$4,\$2	000000	00100	00010	00101000 00100100	=	00822824
0x00000014	sll \$5,\$5,2	000000	00000	00101	00101000 10000000	=	00052880
0x00000018	beq \$5,\$1,-2(=,转 14)	000100	00101	00001	11111111 11111110	=	10A1FFFE
0x0000001C	jal 0x00000050	000011	00000	00000	00000000 00010100	=	0C000014
0x00000020	slt \$8,\$13,\$1	000000	01101	00001	01000000 00101010	=	01A1402A
0x00000024	addiu \$14,\$0,-2	001001	00000	01110	11111111 11111110	=	240EFFFF
0x00000028	slt \$9,\$8,\$14	000000	01000	01110	01001000 00101010	=	010E482A
0x0000002C	slti \$10,\$9,2	001010	01001	01010	00000000 00000010	=	292A0002
0x00000030	slti \$11,\$10,0	001010	01010	01011	00000000 00000000	=	294B0000
0x00000034	add \$11,\$11,\$10	000000	01011	01010	01011000 00100000	=	016A5820
0x00000038	bne \$11,\$2,-2 (≠, 转 34)	000101	01011	00010	11111111 11111110	=	1562FFFE
0x0000003C	addiu \$12,\$0,-2	001001	00000	01100	11111111 11111110	=	240CFFFF
0x00000040	addiu \$12,\$12,1	001001	01100	01100	00000000 00000001	=	258C0001
0x00000044	bltz \$12,-2 (<0, 转 40)	000001	01100	00000	11111111 11111110	=	0580FFFE
0x00000048	andi \$12,\$2,2	001100	00010	01100	00000000 00000010	=	304C0002
0x0000004C	j 0x0000005C	000010	00000	00000	00000000 00010111	=	08000017
0x00000050	sw \$2,4(\$1)	101011	00001	00010	00000000 00000100	=	AC220004
0x00000054	lw \$13,4(\$1)	100011	00001	01101	00000000 00000100	=	8C2D0004
0x00000058	jr \$31	000000	11111	00000	00000000 00001000	=	03E00008

0x0000005 C	halt	111111	00000	00000	00000000 00000000	=	FC000000
----------------	------	--------	-------	-------	-------------------	---	----------

```

00100100 00000001 00000000 00001000
00110100 00000010 00000000 00000010
00111000 01000011 00000000 00001000
00000000 01100001 00100000 00100010
00000000 10000010 00101000 00100100
00000000 00000101 00101000 10000000
00010000 10100001 11111111 11111110
00001100 00000000 00000000 00010100
00000001 10100001 01000000 00101010
00100100 00001110 11111111 11111110
00000001 00001110 01001000 00101010
00101001 00101010 00000000 00000010
00101001 01001011 00000000 00000000
00000001 01101010 01011000 00100000
00010101 01100010 11111111 11111110
00100100 00001100 11111111 11111110
00100101 10001100 00000000 00000001
00000101 10000000 11111111 11111110
00110000 01001100 00000000 00000010
00001000 00000000 00000000 00010111
10101100 00100010 00000000 00000100
10001100 00101101 00000000 00000100
00000011 11100000 00000000 00001000
11111100 00000000 00000000 00000000

```

行 6, 列 36 | 863 个字符 | 100% | Windows (CRLF) | UTF-8

上面是“Instructions.txt”文档的内容

特别注意txt文本中的指令格式应如上所示每八个比特位进行一个填充一个空格或者换行

（因为我们的实验要求指令存储器和数据存储器单元宽度一律使用8位，因此将一个32

位的指令拆成4个8位的存储器单元存储，再从文件中取出后将他们合并为32的指令）

Input:

```

PC4,          // 用于跳转指令的高4位PC值

IAddr,        // 指令地址输入

RW,           // 读写控制信号：0 表示写入，1 表示读取

```

Output:

```

op,           // 操作码

funct,        // 功能码

```

```

rs,          // 源寄存器1

rt,          // 源寄存器2/目标寄存器

rd,          // 目标寄存器（仅适用于某些指令）

Immediate,   // 立即数

Sa,          // 移位量

JumpPC,      // 跳转地址

IDataOut     // 用于保存从内存读取的32位指令

```

实现代码：

```

`timescale 1ns / 1ps
module InstructionMemory(
    input [3:0] PC4, // 用于跳转指令的高4位PC值
    input [31:0] IAddr, // 指令地址输入
    input RW, // 读写控制信号：0表示写入，1表示读取

    output [5:0] op, // 操作码
    output [5:0] funct,
    output [4:0] rs, // 源寄存器1
    output [4:0] rt, // 源寄存器2/目标寄存器
    output [4:0] rd, // 目标寄存器（仅适用于某些指令）
    output [15:0] Immediate, // 立即数
    output [4:0] Sa, // 移位量
    output [31:0] JumpPC, // 跳转地址
    output reg [31:0] IDataOut // 用于保存从内存读取的32位指令
);

// 因为实验要求指令存储器和数据存储器单元宽度一律使用8位，
// 因此将一个32位的指令拆成4个8位的存储器单元存储。
reg [7:0] Mem[0:127]; // 定义一个8位宽、128个地址空间的内存数组来存储指令
// 将IDataOut中的相应位分配给输出端口
assign op = IDataOut[31:26]; // 提取操作码（6位）
assign rs = IDataOut[25:21]; // 提取源寄存器1（5位）
assign rt = IDataOut[20:16]; // 提取源寄存器2或目标寄存器（5位）
assign rd = IDataOut[15:11]; // 提取目标寄存器（5位）
assign Immediate = IDataOut[15:0]; // 提取立即数（16位）
assign Sa = IDataOut[10:6]; // 提取移位量（5位）
assign funct = (IDataOut[31:26] == 6'b000000)? IDataOut[5:0] :
6'bxxxxxx;

// 构建跳转地址，由PC4和指令的一部分组成，并在末尾添加两个零以确保是字对齐的
assign JumpPC = {PC4, IDataOut[25:0], 2'b00};

```

```

initial begin
    $readmemb("C:/Users/Bamboo/Desktop/vivado1/MyCpu1/Instructions.txt",
Mem); // 在仿真开始时从文件 "Instructions.txt" 中加载指令集到内存
    IDataOut = 0; // 初始化指令数据为 0
end

// 当 IAddr 或 RW 为 1 触发
always @( RW or IAddr) begin
    if(RW == 1) begin // 如果是读操作
        // 根据 IAddr 从内存中读取 4 个连续的 8 位数据并组合成 32 位指令
        IDataOut[7:0] = Mem[IAddr + 3];
        IDataOut[15:8] = Mem[IAddr + 2];
        IDataOut[23:16] = Mem[IAddr + 1];
        IDataOut[31:24] = Mem[IAddr];
    end
end

endmodule

```

## RegisterFile（寄存器）

Input:

CLK, // 时钟信号

RegDst, // 寄存器目标选择信号：为1时选择rd，为0时选择rt

RegWre, // 寄存器写使能信号：为1时允许写入寄存器

DBDataSrc, // 数据源选择信号：为1时从数据存储器读取，为0时从ALU读取

Opcode, // 指令操作码（注释掉）

rs, // 源寄存器1编号

rt, // 源寄存器2编号

rd, // 目标寄存器编号

im, // 立即数（注释掉）

dataFromALU, // 来自ALU的数据

dataFromRW, // 来自数据存储器或其它源的数据

Output:

Data1, // ALU运算的第一个输入A

```
Data2,          // ALU运算的第二个输入B

writeData       // 写入寄存器的数据
```

实现代码:

```
`timescale 1ns / 1ps
module RegisterFile(
    input CLK,                // 时钟信号
    input RegDst,             // 寄存器目标选择信号: 为 1 时选择 rd, 为 0 时选择
rt
    input RegWre,             // 寄存器写使能信号: 为 1 时允许写入寄存器
    input DBDataSrc,          // 数据源选择信号: 为 1 时从数据存储器读取,
为 0 时从 ALU 读取
    input [4:0] rs,           // 源寄存器 1 编号
    input [4:0] rt,           // 源寄存器 2 编号
    input [4:0] rd,           // 目标寄存器编号
    input [31:0] dataFromALU, // 来自 ALU 的数据
    input [31:0] dataFromRW,  // 来自数据存储器或其它源的数据

    output [31:0] Data1,      // ALU 运算的第一个输入 A
    output [31:0] Data2,      // ALU 运算的第二个输入 B
    output [31:0] writeData   // 写入寄存器的数据
);

// 要写的寄存器端口
wire [4:0] writeReg;

// 根据 RegDst 选择要写入的目标寄存器
assign writeReg = RegDst ? rd : rt;

// 根据 DBDataSrc 选择写入寄存器的数据来源
assign writeData = DBDataSrc ? dataFromRW : dataFromALU;

// 初始化寄存器数组, 共 32 个 32 位寄存器
reg [31:0] register[0:31];
integer i;
initial begin
    for(i = 0; i < 32; i = i + 1)
        register[i] <= 0; // 所有寄存器初始值设为 0
end

// 输出 Data1 和 Data2 随寄存器变化而变化
// Data1 为 ALU 运算时的第一个输入 A, 其值始终为 rs 寄存器的内容
// Data2 为 ALU 运算时的第二个输入 B, 其值始终为 rt 寄存器的内容
assign Data1 = register[rs];
assign Data2 = register[rt];
```

```

// 在时钟下降沿触发，更新寄存器内容
always @ (negedge CLK) begin
    if(RegWre && writeReg != 0) // 只有在 RegWre 为高电平且不是写入 0 号寄存器时才允许写入
        register[writeReg] <= writeData;
end

endmodule

```

## DataMemory（数据存储器）

Input:

CLK,                // 时钟信号

DAddr,             // 地址输入，用于指定访问的内存位置

DataIn,            // 数据输入，当进行写操作时，此为要写入的数据

RD,                 // 读使能信号：为1时允许读操作，为0时不进行读操作（或高阻态）

WR,                 // 写使能信号：为0时允许写操作，为1时禁止写操作或输出高阻态

Output:

DataOut            // 数据输出，当进行读操作时，从此输出读取的数据

实现代码:

```

module DataMemory(
    input CLK, // 时钟信号
    input wire [31:0] DAddr, // 地址输入，用于指定访问的内存位置
    input wire [31:0] DataIn, // 数据输入，当进行写操作时，此为要写入的数据
    input RD, // 读使能信号：为 1 时允许读操作，为 0 时不进行读操作（或高阻态）
    input WR, // 写使能信号：为 0 时允许写操作，为 1 时禁止写操作或输出高阻态
    output wire [31:0] DataOut // 数据输出，当进行读操作时，输出读取的数据
);

// 定义一个 8 位宽、128 个地址空间的内存数组
reg [7:0] Memory [0:127]; // 每个存储单元为 8 位，共有 128 个这样的存储单元

// 因为一条指令由四个存储单元存储，所以地址需要左移两位以乘以 4
wire [31:0] address;
assign address = (DAddr << 2); // 将输入地址左移两位，相当于乘以 4，得到实际的字节地址

```



```

// 读操作: 如果 RD 为 0, 则从内存中读取数据; 否则输出高阻态 ('z')
assign DataOut[7:0] = (RD == 1) ? Memory[address + 3] : 8'bz; // 最低字节
assign DataOut[15:8] = (RD == 1) ? Memory[address + 2] : 8'bz; // 第二个字节
assign DataOut[23:16] = (RD == 1) ? Memory[address + 1] : 8'bz; // 第三个字节
assign DataOut[31:24] = (RD == 1) ? Memory[address] : 8'bz; // 最高字节

// 写操作: 在 CLK 的下降沿触发, 如果 WR 为 0, 则将 DataIn 中的数据写入内存
always @ (negedge CLK) begin
    if(WR == 1) begin // 如果写使能信号为 0, 则执行写操作
        Memory[address] <= DataIn[31:24]; // 写入最高字节
        Memory[address + 1] <= DataIn[23:16]; // 写入第三个字节
        Memory[address + 2] <= DataIn[15:8]; // 写入第二个字节
        Memory[address + 3] <= DataIn[7:0]; // 写入最低字节
    end
end
endmodule

```

## ZreoSignExtend (符号零扩展器)

Input:

Immediate, // 输入的16位立即数

ExtSel, // 扩展选择信号: 1为符号扩展, 0为零扩展

Output:

Out // 输出的32位扩展后的数据

实现代码:

```

`timescale 1ns / 1ps
module SignZeroExtend(
    input wire [15:0] Immediate, // 输入的 16 位立即数
    input ExtSel, // 扩展选择信号: 1 为符号扩展, 0 为零扩展
    output wire [31:0] Out // 输出的 32 位扩展后的数据
);
// 后 16 位存储立即数
// 直接将输入的 16 位立即数赋值给输出的低 16 位
assign Out[15:0] = Immediate[15:0];

```

```
// 前 16 位根据立即数进行补 1 或补 0 的操作
// 如果 ExtSel 为 1, 则进行符号扩展: 将 Immediate 的最高位 (第 15 位) 复制到高 16 位
// 如果 ExtSel 为 0, 则进行零扩展: 将高 16 位全部设为 0
assign Out[31:16] = ExtSel == 1 ? {16{Immediate[15]}} : 16'b0;

endmodule
```

## mux2to1 (二选一选择器)

input:

ALUResult,

DMOut,

DBDataSrc,

Output:

DB

实现代码如下:

```
`timescale 1ns / 1ps

module mux2to1(
    input [31:0] ALUResult,
    input [31:0] DMOut,
    input DBDataSrc,
    output [31:0] DB
);
    assign DB = (DBDataSrc == 0)? ALUResult : DMOut;
endmodule
```

## DR (数据存储器)

实现代码如下:

```
`timescale 1ns / 1ps

// 定义了一个名为 DR 的模块, 该模块包含一个时钟输入 (clk)、一个 32 位的数据输入 (in) 以及一个 32 位的寄存器输出 (out)。
module DR(
    input clk,           // 输入: 时钟信号
```

```

    input [31:0] in,      // 输入: 32 位宽的数据信号
    output reg [31:0] out // 输出: 32 位宽的寄存器类型信号, 用于保存数据
);

// 始终在时钟的上升沿执行以下过程块
always @(posedge clk) begin
    // 在每个时钟周期的上升沿, 将输入信号 'in' 的值赋给输出寄存器 'out'
    // 这意味着每当有新的时钟脉冲到来时, 输入数据会被采样并传递到输出
    out <= in;
end

endmodule

```

## IR (指令存储器)

Input :

clk, //时钟信号

IRWre, //写使能信号, 当为高电平时允许更新内部状态

instruction, //32位宽的数据信号, 表示待解析的指令

PC4, // PC + 4 的值, 通常用于跳转或分支指令计算目的地址

Output:

OpCode, // 6位操作码

func, //6位功能码, 主要用于R型指令

rs, //源寄存器1编号

rt, //源寄存器2或目标寄存器编号

rd, //目标寄存器编号, 主要用于R型指令

Immediate, // 16位立即数

sa, //移位量, 主要用于R型指令中的移位操作

JumpPC //跳转指令的目标地址

下面是代码实现:

```

`timescale 1ns / 1ps

// 32 位指令数据 (instruction)、以及 PC+4 值 (PC4) 作为输入, 并输出解析后的指令字段。

```

```

module IR(
    input clk,           // 输入：时钟信号
    input IRWre,         // 输入：写使能信号，当为高电平时允许更新内部状态
    input [31:0] instruction, // 输入：32 位宽的数据信号，表示待解析的指令
    input PC4,           // 输入：PC + 4 的值，通常用于跳转或分支指令计算目的地址

    output reg[5:0] OpCode, // 输出：6 位操作码 (OpCode)
    output reg[5:0] func,   // 输出：6 位功能码 (func)，主要用于 R 型指令
    output reg[4:0] rs,     // 输出：源寄存器 1 编号 (rs)
    output reg[4:0] rt,     // 输出：源寄存器 2 或目标寄存器编号 (rt)
    output reg[4:0] rd,     // 输出：目标寄存器编号 (rd)，主要用于 R 型指令

    output reg[15:0] Immediate, // 输出：16 位立即数 (Immediate)
    output reg[4:0] sa,         // 输出：移位量 (sa)，主要用于 R 型指令中的移位操作
    output reg[31:0] JumpPC    // 输出：跳转指令的目标地址 (JumpPC)
);

// 始终在时钟的上升沿执行以下过程块
always @(posedge clk) begin
    // 如果写使能信号 (IRWre) 有效，则进行指令解析
    if(IRWre == 1) begin
        // 解析并分配指令的不同部分到对应的输出端口

        // 操作码 (OpCode) 位于指令的最高位 (31-26 位)
        OpCode <= instruction[31:26];

        // 功能码 (func) 仅对操作码为 0x00 (R 型指令) 的指令有效，位于最低位 (5-0 位)
        // 对于非 R 型指令，设置为未知 ('x')
        func <= (instruction[31:26] == 6'b000000) ?
instruction[5:0] : 6'bxxxxxx;

        // 源寄存器 1 编号 (rs) 位于指令的第 25 至 21 位
        rs <= instruction[25:21];

        // 源寄存器 2 或目标寄存器编号 (rt) 位于指令的第 20 至 16 位
        rt <= instruction[20:16];

        // 目标寄存器编号 (rd) 位于指令的第 15 至 11 位，主要用于 R 型指令
        rd <= instruction[15:11];

        // 立即数 (Immediate) 位于指令的最低 16 位

```

```

        Immediate <= instruction[15:0];

        // 移位量 (sa) 位于指令的第 10 至 6 位，主要用于 R 型指令中的移位操作
        sa <= instruction[10:6];

        // 跳转指令的目标地址由 PC+4 值和指令的低 26 位组成，最后两位总是 0
        // 这是因为 MIPS 架构假设指令长度为 4 字节对齐
        JumpPC <= {PC4, instruction[25:0], 2'b00};
    end
end
endmodule

```

## ChangeState (状态转化单元)

Input:

opCode,

func, // 指令的操作码和功能码 (对于 R 型指令)

Reset, // 复位信号，低电平有效

clk, // 时钟信号

output “

new\_state // 输出寄存器，用于保存下一个状态

实现代码如下:

```

// 定义时间尺度为 1ns 的时间单位和 1ps 的时间精度
`timescale 1ns / 1ps

module ChangeState(
    // 输入输出端口定义
    input [5:0] opCode, func, // 指令的操作码和功能码 (对于 R 型指令)
    input Reset, // 复位信号，低电平有效
    input clk, // 时钟信号
    output reg [2:0] new_state // 输出寄存器，用于保存下一个状态
);

// 定义状态常量，使用 3 位二进制表示 7 种不同的状态
parameter [2:0] IF = 3'b000,
                ID = 3'b001,
                EXE1 = 3'b110,
                EXE2 = 3'b101,

```

```

        EXE3 = 3'b010,
        WB1  = 3'b111,
        WB2  = 3'b100,
        MEM  = 3'b011;

// 初始化状态为 IF（取指阶段）
initial begin
    new_state = IF;
end

// 状态更新逻辑，基于时钟下降沿和复位信号
always @(negedge clk or negedge Reset) begin
    if (Reset == 0) begin // 如果复位信号有效，则将状态设置为 IF
        new_state = IF;
    end else begin // 否则根据当前状态进行状态转移
        case (new_state)
            IF: new_state <= ID; // 取指后进入译码阶段
            ID: begin
                // 根据 opCode 和 func 的不同，决定下个状态
                // beq, bne, bltz 指令转到 EXE2
                if (opCode == 6'b000100 || opCode == 6'b000101 || opCode
== 6'b000001)
                    new_state <= EXE2;
                // sw, lw 指令转到 EXE3
                else if (opCode == 6'b101011 || opCode == 6'b100011)
                    new_state <= EXE3;
                // j, jal, jr, halt 指令回到 IF
                else if (opCode == 6'b000010 || opCode == 6'b000011 ||
                    (opCode == 6'b000000 && func == 6'b001000) ||
opCode == 6'b111111)
                    new_state <= IF;
                else
                    new_state <= EXE1; // 其他指令转到 EXE1
            end
            EXE1: new_state <= WB1; // 执行后进入写回阶段 WB1
            EXE2: new_state <= IF; // 跳转指令后直接回到 IF
            EXE3: new_state <= MEM; // 存储/加载指令后进入 MEM 阶段
            WB1: new_state <= IF; // 写回后回到 IF
            WB2: new_state <= IF; // 第二写回阶段后也回到 IF
            MEM: begin
                // 对于存储指令(sw)，执行后直接回到 IF
                if (opCode == 6'b101011)
                    new_state <= IF;
                else

```

```

new_state <= WB2; // 对于其他情况，转到第二写回阶段 WB2
    end
endcase
end
end
endmodule

```

## ControlUnit（控制单元）

与“单周期CPU设计与实现”实验不同，多周期CPU的ControlUnit是时序逻辑，这是因为在多周期CPU中，一条指令的执行被分为了2~5个状态分步执行。在多周期CPU中，控制信号不仅依赖于当前指令的操作码（opcode），还依赖于当前正处于的状态（state）。在所有控制信号中，有四个涉及“写入”操作的信号，分别是PCWre、IRWre、mWR和RegWre，除了这四个信号之外，其余的信号不涉及写入操作，因此可以认为它们与状态无关（即在任何状态下都不会改变）。我们先来考虑除了PCWre、IRWre、mWR和RegWre以外的控制信号。很容易地可以列出如下表示指令与控制信号之间的关系的表格：

指令	opcode	ALUSrcA	ALUSrcB	DBDataSrc	WrRegDSrc	mRD	ExtSel	PCSrc[1:0]	RegDst[1:0]	ALUOp[2:0]
add	000000	0	0	0	1	0	0	00	10	000
sub	000001	0	0	0	1	0	0	00	10	001
addiu	000010	0	1	0	1	0	1	00	01	000
and	010000	0	0	0	1	0	0	00	10	100
andi	010001	0	1	0	1	0	0	00	01	100
ori	010010	0	1	0	1	0	0	00	01	011
xori	010011	0	1	0	1	0	0	00	01	111
sll	011000	1	0	0	1	0	0	00	10	010
slti	100110	0	1	0	1	0	1	00	01	110
slt	100111	0	0	0	1	0	0	00	10	110
sw	110000	0	1	0	1	0	1	00	00	000



lw	110001	0	1	1	1	1	1	00	01	000
beq	110100	0	0	0	1	0	1	01(zero)/ 00	00	001
bne	110101	0	0	0	1	0	1	01(!zero) /00	00	001
bltz	110110	0	0	0	1	0	1	01(sign)/ 00	00	000
j	111000	0	0	0	1	0	0	11	00	000
jr	111001	0	0	0	1	0	0	10	00	000
jal	111010	0	0	0	0	0	0	11	00	000
halt	111111	0	0	0	1	0	0	00	00	000

代码实现如下：

```
`timescale 1ns / 1ps
module ControlUnit(
    //根据数据通路图定义输入和输出
    input [2:0] state,
    input [5:0] OpCode,
    input [5:0] func,
    input zero,
    input sign,

    output reg IRWre,    //IR 指令寄存器的写使能信号，0：不更改；1：接受来自指令寄存器的指令
    output reg PCWre,
    output reg ALUSrcA,
    output reg ALUSrcB,
    output reg DBDataSrc, //0：数据来自 ALU 运算结果； 1：来自 DMOut
    output reg WrRegDSrc, //0：写入寄存器的值来自 PC+4； 1：来自 ALU 结果或者 DMOut
    output reg InsMemRW,
    output reg RD,
    output reg WR,
    output reg ExtSel,    //0：零扩展； 1：符号扩展
    output reg [1:0] RegDst, //要写的寄存器的地址
    output reg [1:0] PCSrc, //决定下一条 PC 如何改变
    output reg [2:0] ALUOp, //决定运算功能
    output reg RegWre
);
    //设置常量
    parameter [2:0] IF = 3'b000,
```

```

ID = 3'b001, EXE1 = 3'b110,
EXE2 = 3'b101, EXE3 = 3'b010,
WB1 = 3'b111, WB2 = 3'b100,
MEM = 3'b011;

//每次状态改变时，信号发生改变
always @(state) begin
    DBDataSrc = (OpCode == 6'b100011) ? 1 : 0; //lw
    WrRegDSrc = (OpCode == 6'b000011) ? 0 : 1; //jal
    ExtSel=(OpCode==6'b001100 || OpCode==6'b001110 || OpCode==6'b001101)
?0:1; //andi,xori,ori
    PCSrc[1]=( (OpCode==6'b000000&&func==6'b001000) || OpCode==6'b00001
0 || OpCode==6'b000011)?1:0; //jr,j,jal
    PCSrc[0]=( (OpCode==6'b000100&&zero==1) || (OpCode==6'b000101&&zero
==0) || (OpCode==6'b000001&&sign==1) || OpCode==6'b000010 || OpCode==6'b00001
1)?1:0; //beq(zero=1)、bne(zero=0)、bltz(sign=1)、j、jal
    RegDst[1]=( (OpCode==6'b000000&&func==6'b100000) || (OpCode==6'b000
000&&func==6'b100010) || (OpCode==6'b000000&&func==6'b100100) || (OpCode==6
'b000000&&func==6'b101010) || (OpCode==6'b000000&&func==6'b000000)) ?1:0;
//add,sub,and,slt,sll
    RegDst[0]=( (OpCode==6'b000000&&func==6'b100000) || (OpCode==6'b000
000&&func==6'b100010) || (OpCode==6'b000000&&func==6'b100100) || (OpCode==6
'b000000&&func==6'b101010) || (OpCode==6'b000000&&func==6'b000000) || OpCod
e==6'b000011)?0:1; //add,sub,and,slt,sll,jal

    //EXE
    if(state==EXE1 || state==EXE2 || state==EXE3)begin
        ALUSrcA=(OpCode==6'b000000&&func==6'b000000)?1:0; //sll
        ALUSrcB=(OpCode==6'b001001 || OpCode==6'b001100 || OpCode==6'b00
1101 || OpCode==6'b001110 || OpCode==6'b001010 || OpCode==6'b101011 || OpCode==
6'b100011)?1:0; //addiu,andi,ori,xori,slti,sw,lw
        ALUOp[2]=(OpCode==6'b001100 || (OpCode==6'b000000&&func==6'b10
0100) || OpCode==6'b001010 || OpCode==6'b001110 || (OpCode==6'b000000&&func==
6'b101010)) ?1:0;
        ALUOp[1]=(OpCode == 6'b001101 || OpCode == 6'b001010 ||
(OpCode == 6'b000000 && func == 6'b100101) || (OpCode == 6'b000000 &&
func ==
6'b000000) || OpCode==6'b001110 || (OpCode==6'b000000&&func==6'b101010)) ?
1 : 0;
        ALUOp[0]=( (OpCode == 6'b000000 && func == 6'b100010) ||
OpCode == 6'b001101 || (OpCode == 6'b000000 && func == 6'b100101) ||
OpCode == 6'b000001 || OpCode == 6'b000101 || OpCode == 6'b000100
||OpCode==6'b001110 ) ? 1 : 0;
    end
end

```

```

//IF
if(state==IF)begin
    if(OpCode!=6'b111111)
        PCWre=1;
    else PCWre=0;
end
else begin
    PCWre=0;
end

//ID
InsMemRW=1;
if(state==ID)
    IRWre=1;
else IRWre=0;

//MEM
if(state==MEM)begin
    RD=(OpCode==6'b100011)?1:0; //lw
    WR=(OpCode==6'b101011)?1:0; //sw
end
else begin
    RD=0;
    WR=0;
end

//WB
if(state==WB1||state==WB2)
    RegWre=(OpCode==6'b000100||OpCode==6'b000101||OpCode==6'b000
001||OpCode==6'b000010||OpCode==6'b101011||(OpCode==6'b000000&&func==6'
b001000)||OpCode==6'b111111)?0:1; //beq,bne,bltz,j,sw,jr,halt
    else if (OpCode==6'b000011 && state==IF) //允许 jal 指令在 ID 阶
段写寄存器，而我的 state 是在时钟下降沿才改变，所以为了防止$31 写入错误，所以在
它的下一个 stats (IF 状态) 才让 RegWre 改变（这里我判断状态为 IF 才更改信号，是
我经过不断调试改出来的，可能更好的实现能让它更加规范的实现）
        RegWre=1;
    else RegWre=0;

end
endmodule

```

## multiCycleCPU（顶层模块）

顶层模块的作用是实例化各个底层模块，并使用导线将它们按照数据通路连接起来。

代码实现如下:

```
`timescale 1ns / 1ps

// 定义多周期 CPU 模块
module multiCycleCPU(
    input CLK, // 时钟信号
    input Reset, // 复位信号
    output [4:0] rs, rt, // 读取寄存器的地址
    output wire [5:0] OpCode, // 操作码
    output wire [31:0] Out1, Out2, // 寄存器文件输出
    output wire [31:0] curPC, nextPC, // 当前和下一个程序计数器的值
    output wire [31:0] Result, // ALU 运算结果
    output wire [31:0] DBData, // 数据总线输出
    output wire [31:0] Instruction // 指令存储器输出
);

    // 定义内部信号
    wire [2:0] ALUOp; // ALU 操作码
    wire [5:0] func; // 函数码
    wire [2:0] state; // 状态码
    wire [31:0] Extout, DMOut; // 扩展输出和数据存储器输出
    wire [15:0] Immediate; // 立即数
    wire [4:0] rd, sa; // 写入寄存器地址和移位量
    wire [31:0] JumpPC; // 跳转程序计数器的值
    wire zero, sign; // ALU 零标志和符号标志
    wire PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, IRWre, WrRegDSrc;

    // 控制信号
    wire InsMemRW, RD, WR, ExtSel; // 内存读写信号和扩展选择信号
    wire [1:0] RegDst, PCSrc; // 寄存器目标选择信号和程序计数器源选择信号
    wire [3:0] PC4; // 程序计数器加 4 的值
    wire [31:0] PC_add_4, drPC_add_4; // 程序计数器加 4 的值和其延迟版本
    wire [31:0] drOut1, drOut2; // 寄存器文件输出的延迟版本
    wire [31:0] DAddr; // 数据地址
    wire [31:0] DB, drDB; // 数据总线及其延迟版本

    // 状态机模块, 用于改变状态
    ChangeState ST(OpCode, func, Reset, CLK, state);

    // 控制单元模块, 用于生成控制信号
    ControlUnit CU(state, OpCode, func, zero, sign, IRWre, PCWre,
        ALUSrcA, ALUSrcB, DBDataSrc, WrRegDSrc, InsMemRW, RD, WR, ExtSel,
        RegDst, PCSrc, ALUOp, RegWre);

    // 程序计数器模块
```

```

    PC pc(CLK, Reset, PCWre, PCSrc, Immediate, Out1, JumpPC, curPC,
nextPC, PC_add_4, PC4);

    // 指令存储器模块
    InstructionMemory IM(curPC, InsMemRW, Instruction);

    // 指令寄存器模块
    IR ir(CLK, IRWre, Instruction, PC4, OpCode, func, rs, rt, rd,
Immediate, sa, JumpPC);

    // 寄存器文件模块
    RegisterFile RF(CLK, WrRegDSrc, drPC_add_4, RegDst, RegWre, rs, rt,
rd, drDB, Out1, Out2, DBData);

    // 延迟寄存器模块，用于程序计数器加 4 的值
    DR pcadd(CLK, PC_add_4, drPC_add_4); // 这个模块的必要性在于，如果没有
它，在仿真时 jal 指令能够正常在$31 号寄存器写入 PC+4，但是在实际硬件上该指令却不能正常执行

    // 延迟寄存器模块，用于 Out1 的延迟
    DR Adr(CLK, Out1, drOut1);

    // 延迟寄存器模块，用于 Out2 的延迟
    DR Bdr(CLK, Out2, drOut2);

    // 算术逻辑单元模块
    ALU alu(drOut1, drOut2, Extout, sa, ALUOp, ALUSrcA, ALUSrcB, zero,
Result, sign);

    // 延迟寄存器模块，用于 ALU 结果的延迟
    DR aluOutDr(CLK, Result, DAddr);

    // 数据存储器模块
    DataMemory DM(DAddr, drOut2, RD, WR, DMOut);

    // 2 选 1 多路选择器模块，用于选择 DBData 的数据源
    mux2to1 db(Result, DMOut, DBDataSrc, DB);

    // 延迟寄存器模块，用于 DB 的延迟
    DR dbdr(CLK, DB, drDB);

    // 符号零扩展模块
    SignZeroExtend SZE(Immediate, ExtSel, Extout);
endmodule

```

## 测试代码:

```
`timescale 1ns / 1ps

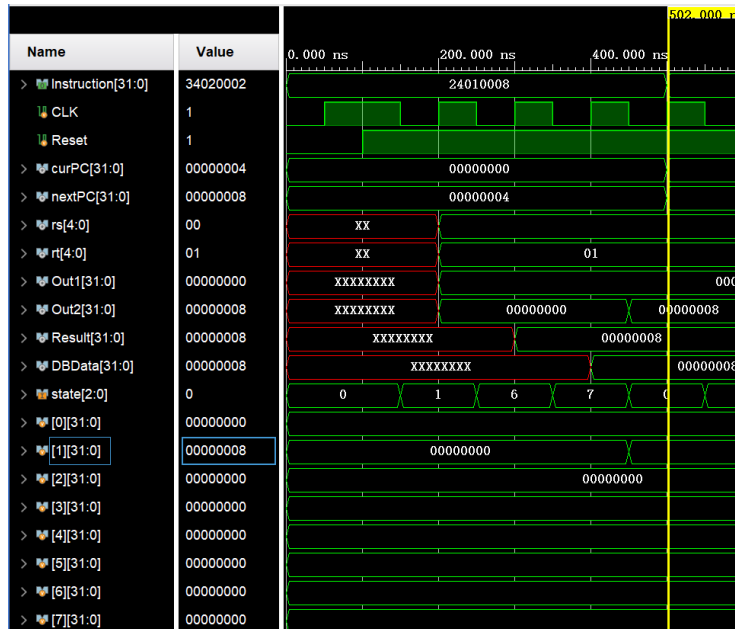
module multiCycleCPU_sim();
    //inputs
    reg CLK;
    reg Reset;
    //Outputs
    wire [31:0] curPC,nextPC;
    wire [4:0] rs, rt;
    wire [31:0] Out1, Out2;
    wire [31:0] Result,DBData;
    //instantiate the Unit Under Test
    multiCycleCPU uut(
        .CLK(CLK),
        .rs(rs),
        .rt(rt),
        .Out1(Out1),
        .Out2(Out2),
        .Reset(Reset),
        .DBData(DBData),
        .curPC(curPC),
        .nextPC(nextPC),
        .Result(Result)
    );

    initial begin
        //record
        $dumpfile("SCCPU.vcd");
        $dumpvars(0, multiCycleCPU_sim);
        //innitial inputs
        CLK = 0;
        Reset = 0; //刚开始设置 PC 为 0
        #50;
        CLK = 1;
        #50;
        Reset = 1;
        //产生时钟信号
        forever #50 begin
            CLK = !CLK;
        end
    end
end
```

endmodule

下面是对“测试表格”中的指令正确性的逐一验证：

1. addiu      100ns-500ns      状态码 0-1-6-7      \$1=8

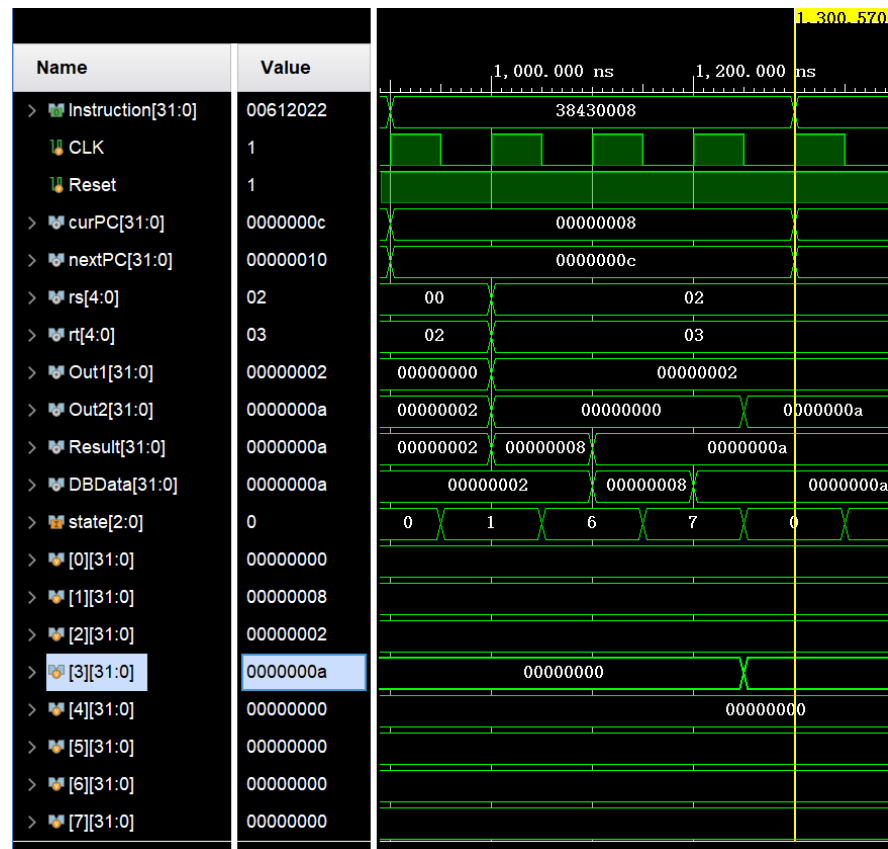


2. ori      500ns-900ns      状态码 0-1-6-7      \$2=2

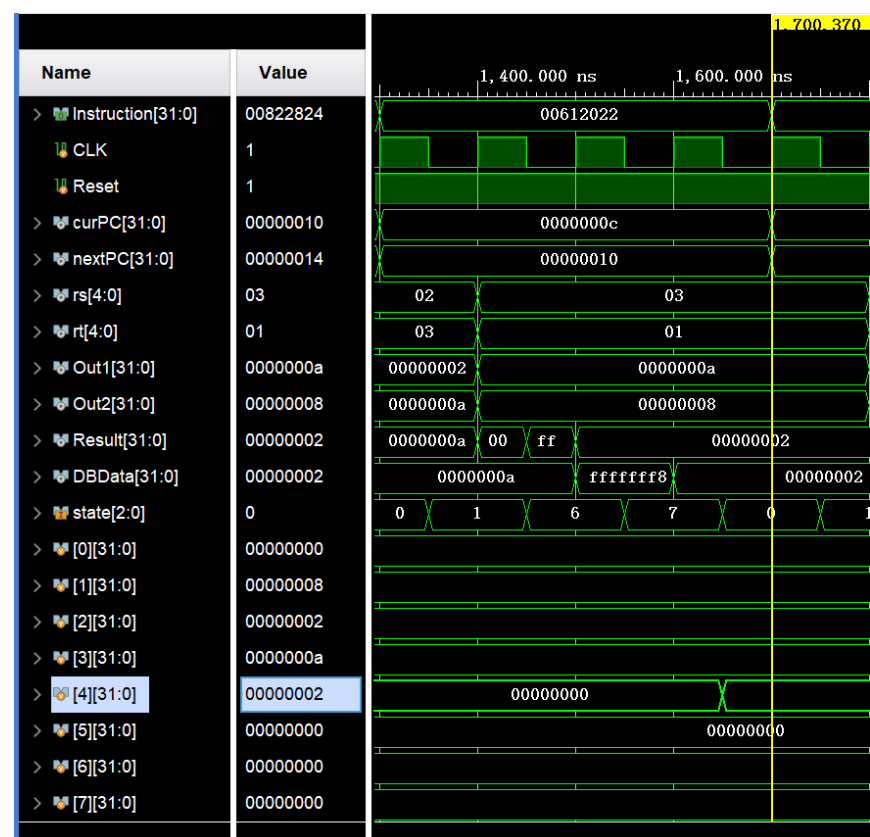




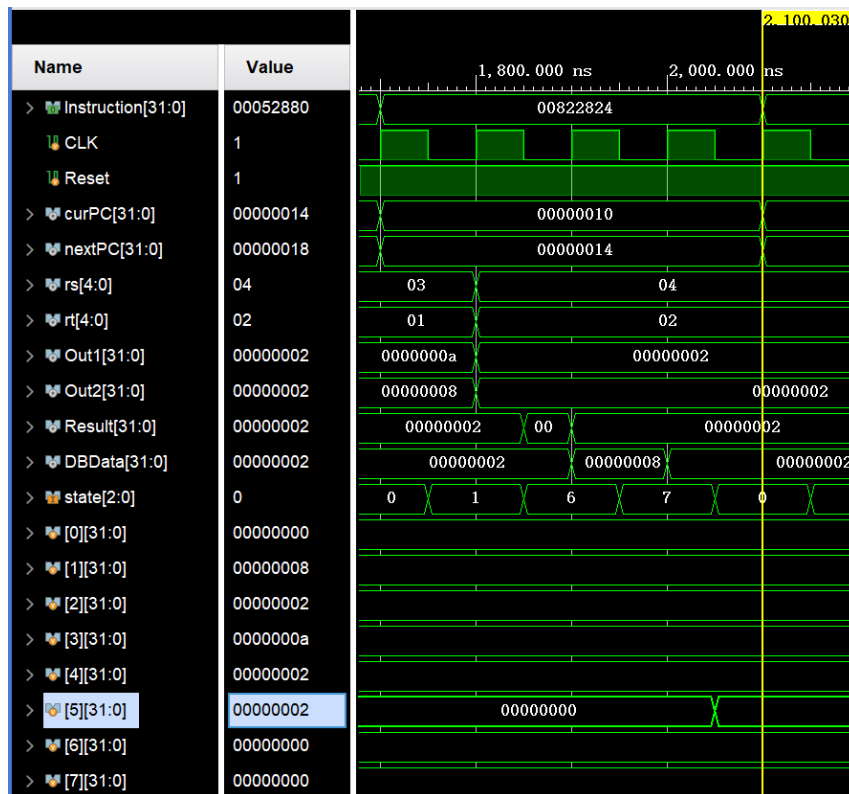
3.xori 900ns-1300ns 状态码 0-1-6-7 \$3=10



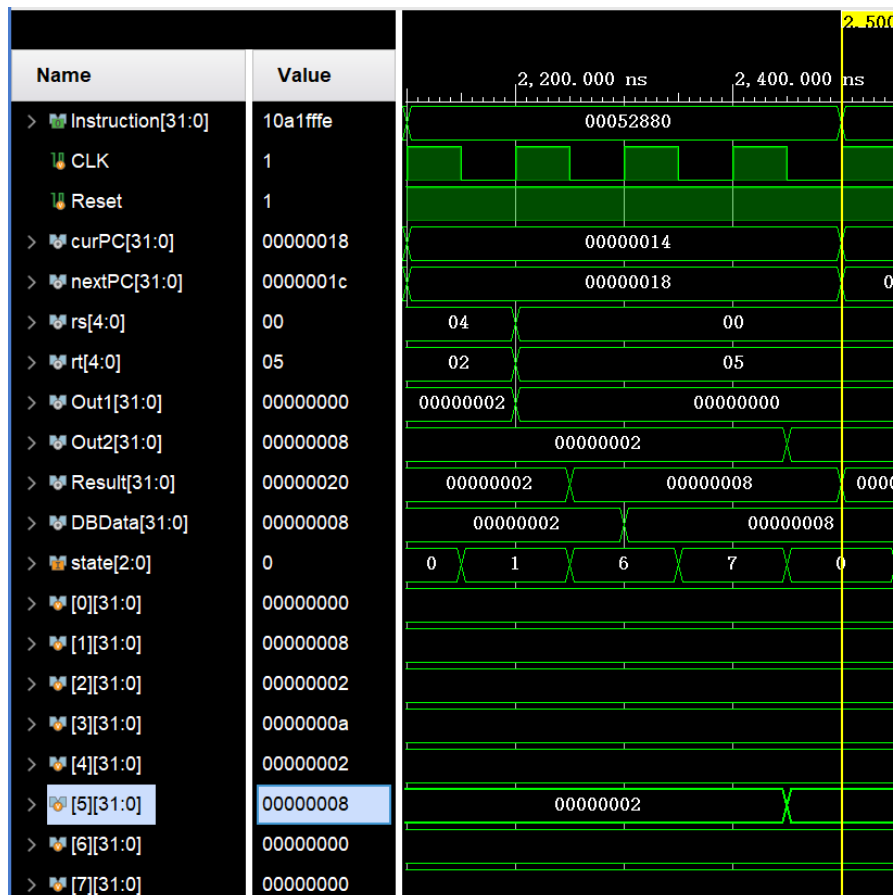
4.sub 1300ns-1700ns 状态码 0-1-6-7 \$4=2



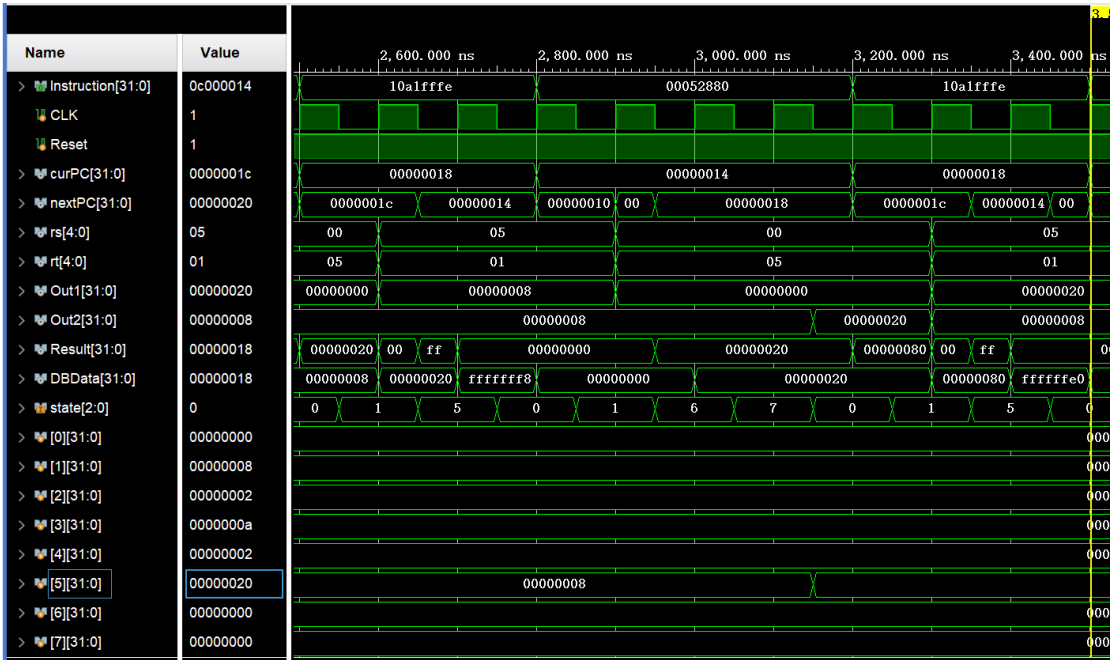
5.and 1700ns-2100ns 状态码 0-1-6-7 \$5=2



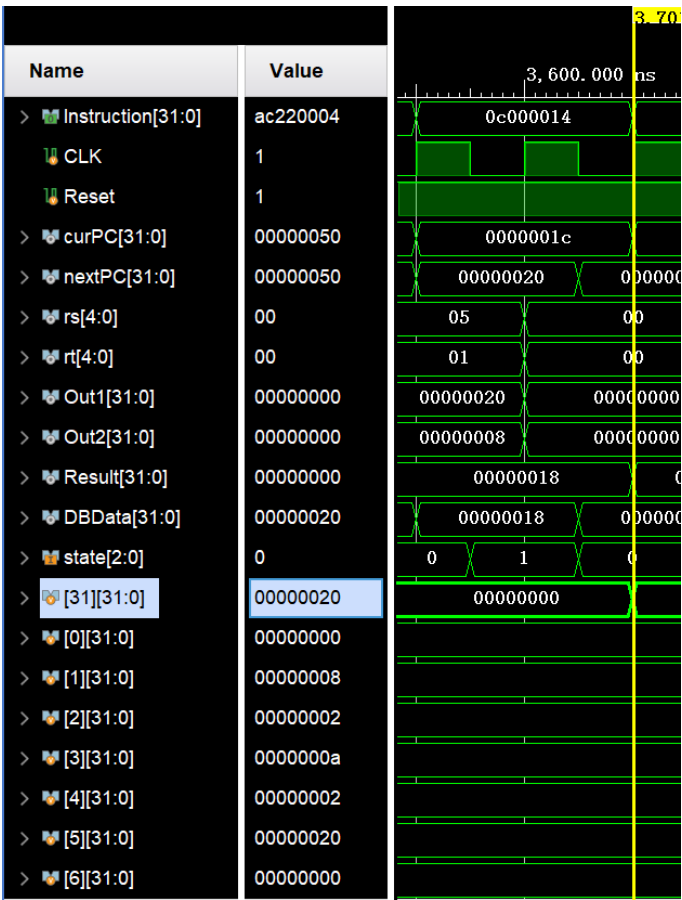
6.sll 2100ns-2500ns 状态码 0-1-6-7 \$5=8



7.beq 2500ns-2800ns 状态码 0-1-5 \$5 和\$1 相等则跳转到 00000014  
sll 2800ns-3200ns 状态码 0-1-6-7 \$5=32  
beq 3200ns-3500ns 状态码 0-1-5 \$5 和\$1 不相等，继续下一条指令



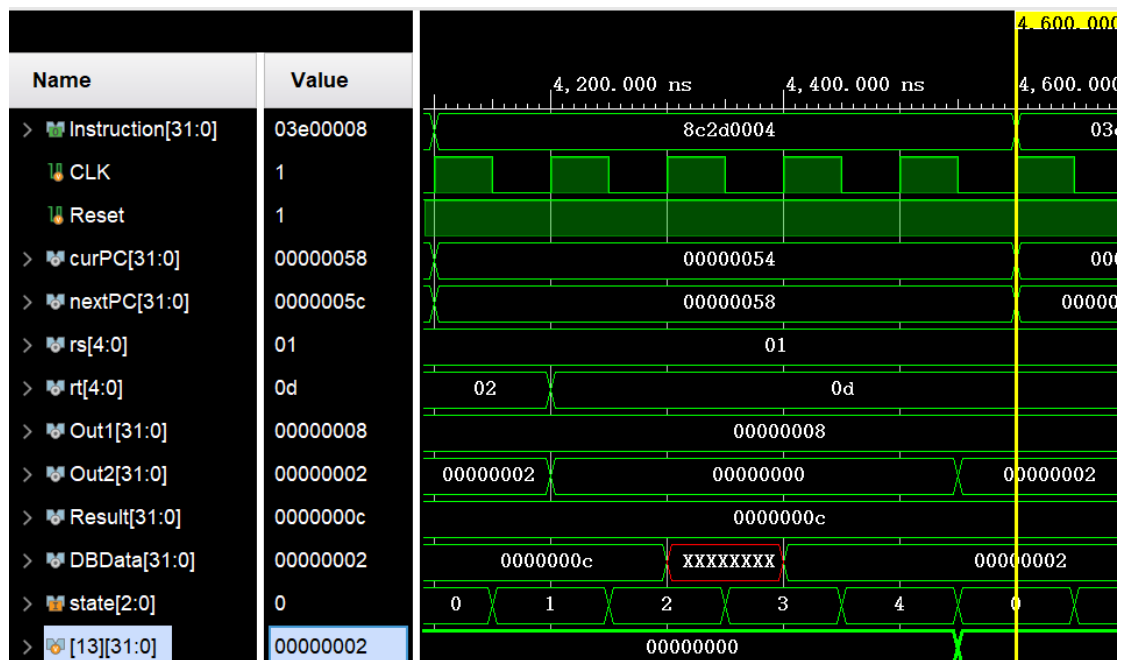
8.jal 3500ns-3700ns 状态码: 0-1 \$31=20



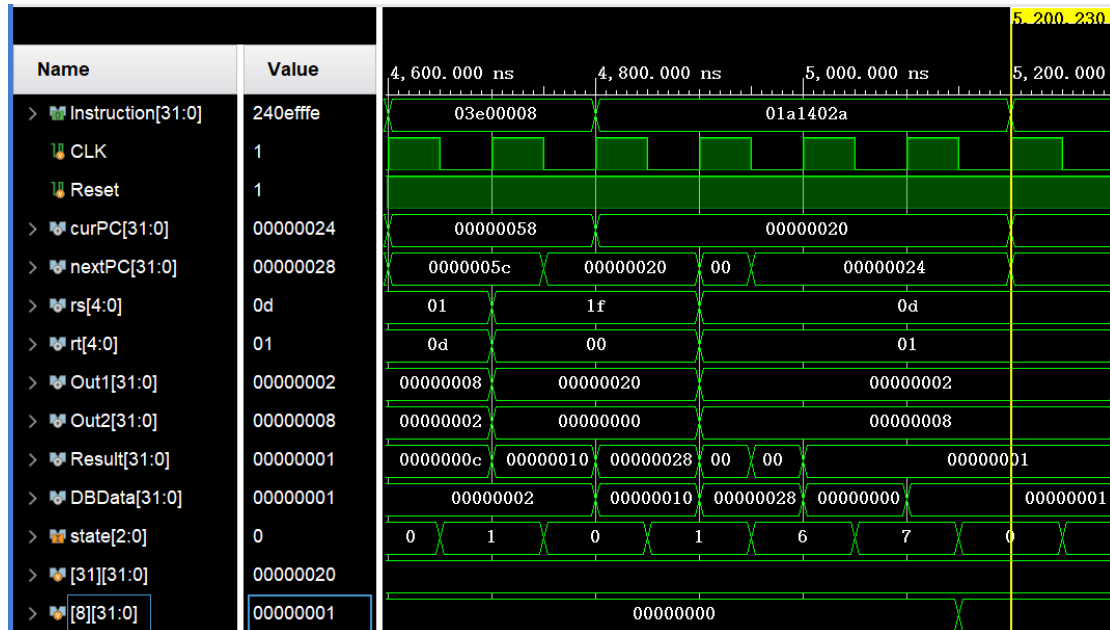
9.sw 3700ns-4100ns 状态码 0-1-2-3 DataMemory[51]=2



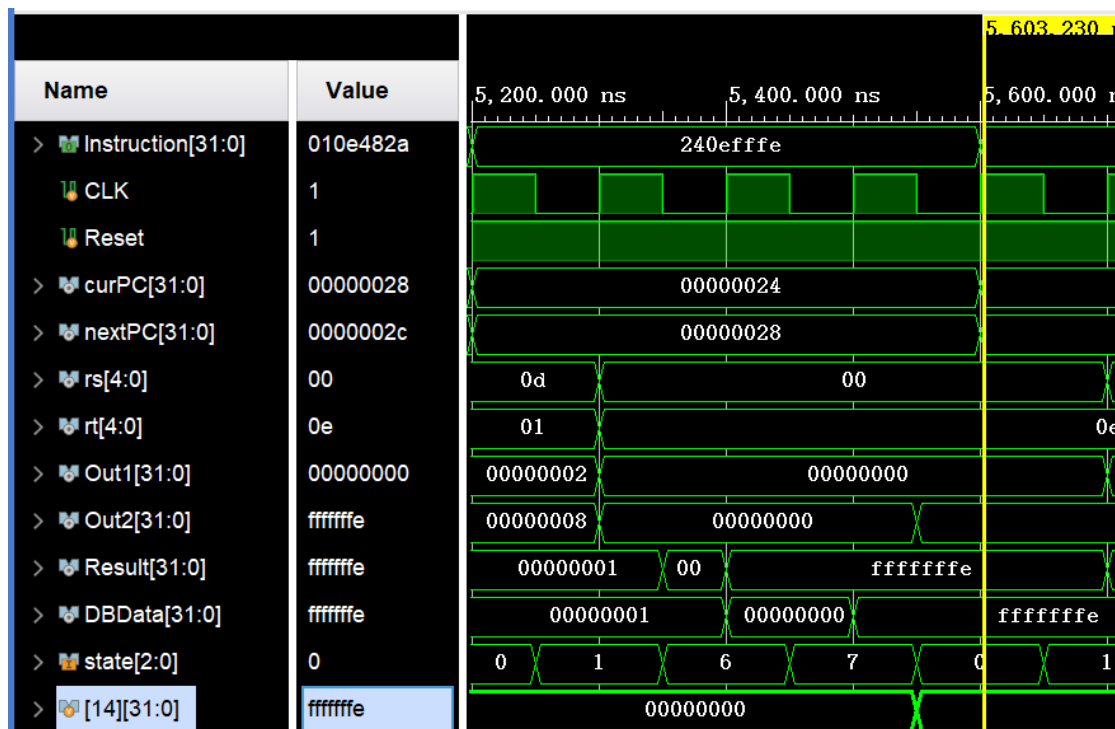
10.lw 4100ns-4600ns 状态码: 0-1-2-3-4 \$13=2



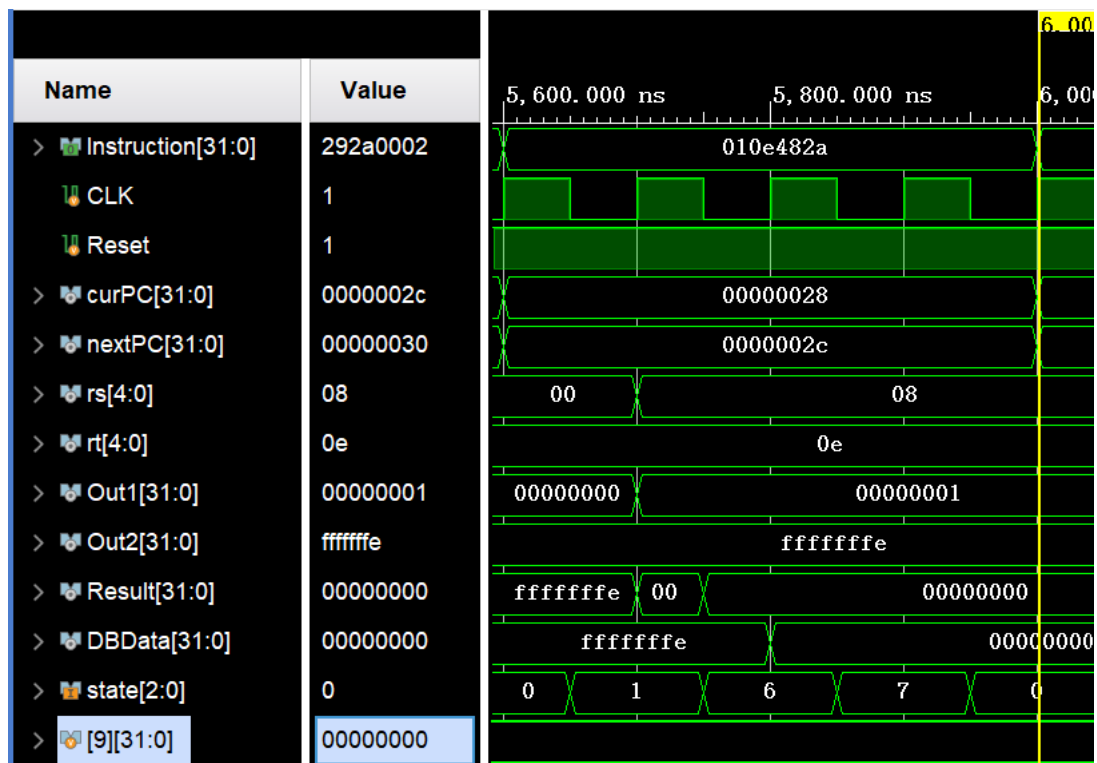
11.jal 4600ns-4800ns 状态码 0-1 00000020 跳转到指令 00000020  
 slt 4800ns-5200ns 状态码 0-1-6-7 \$8=1



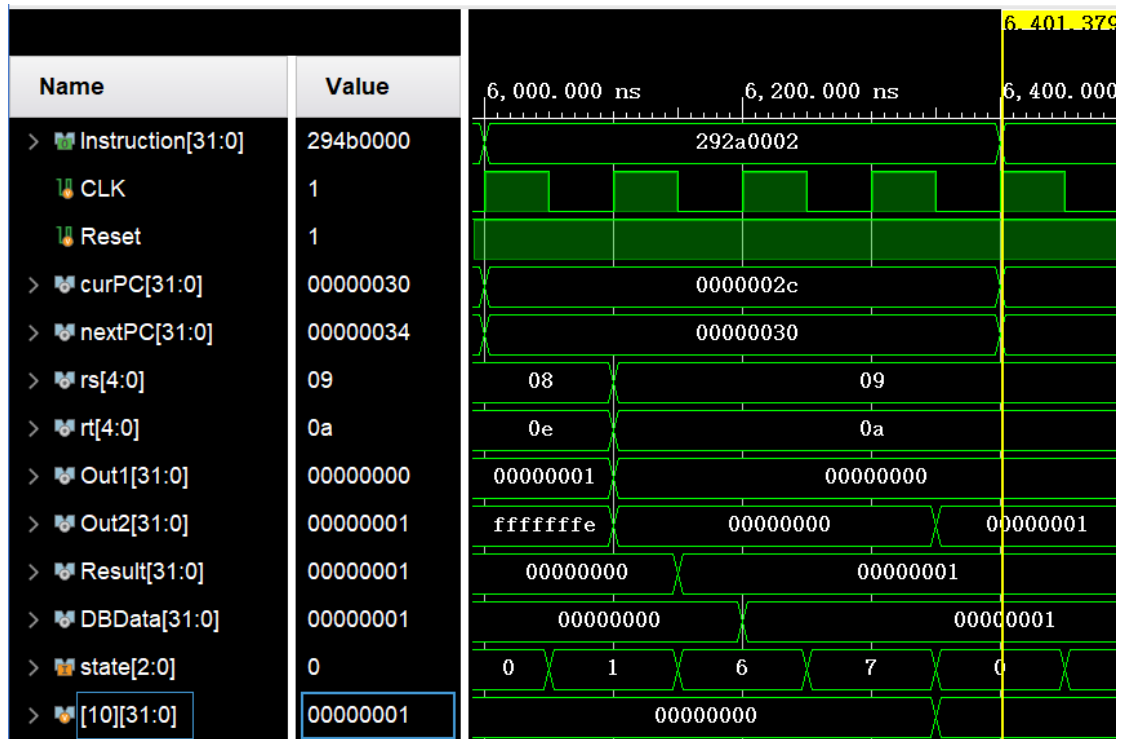
12.addiu 5200ns-5600ns 状态码: 0-1-6-7 \$14=-2



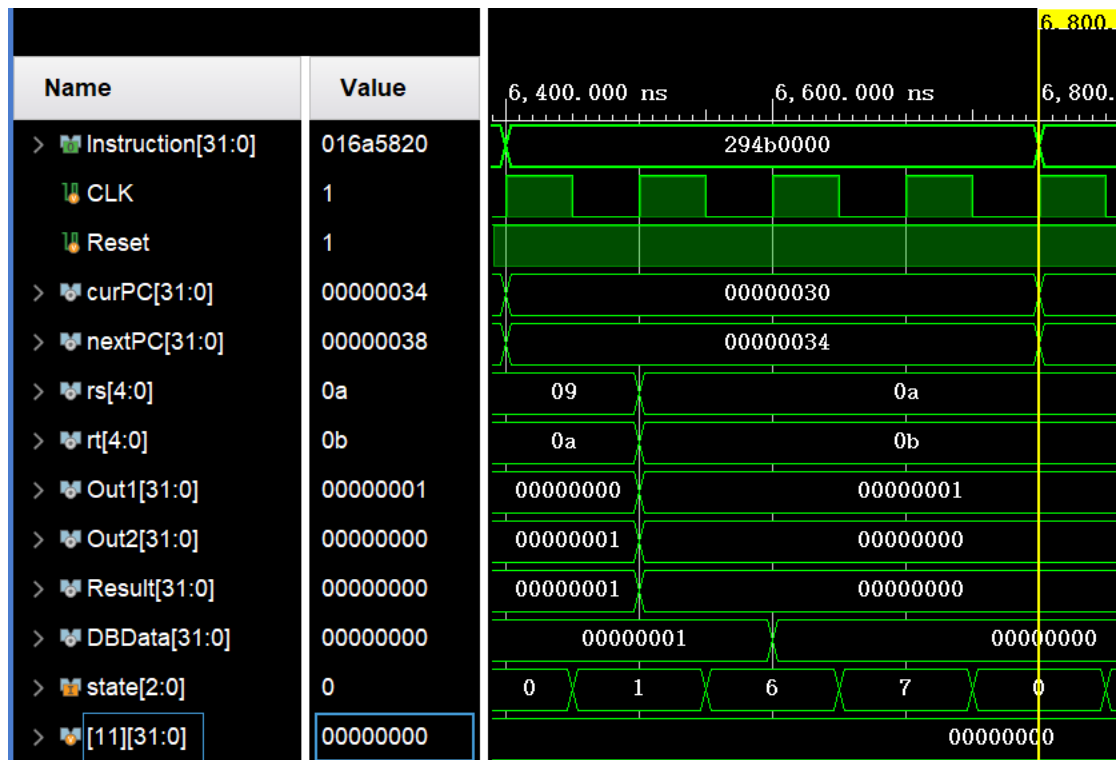
13.slt      5600ns-6000ns      状态码 0-1-6-7      \$9=0



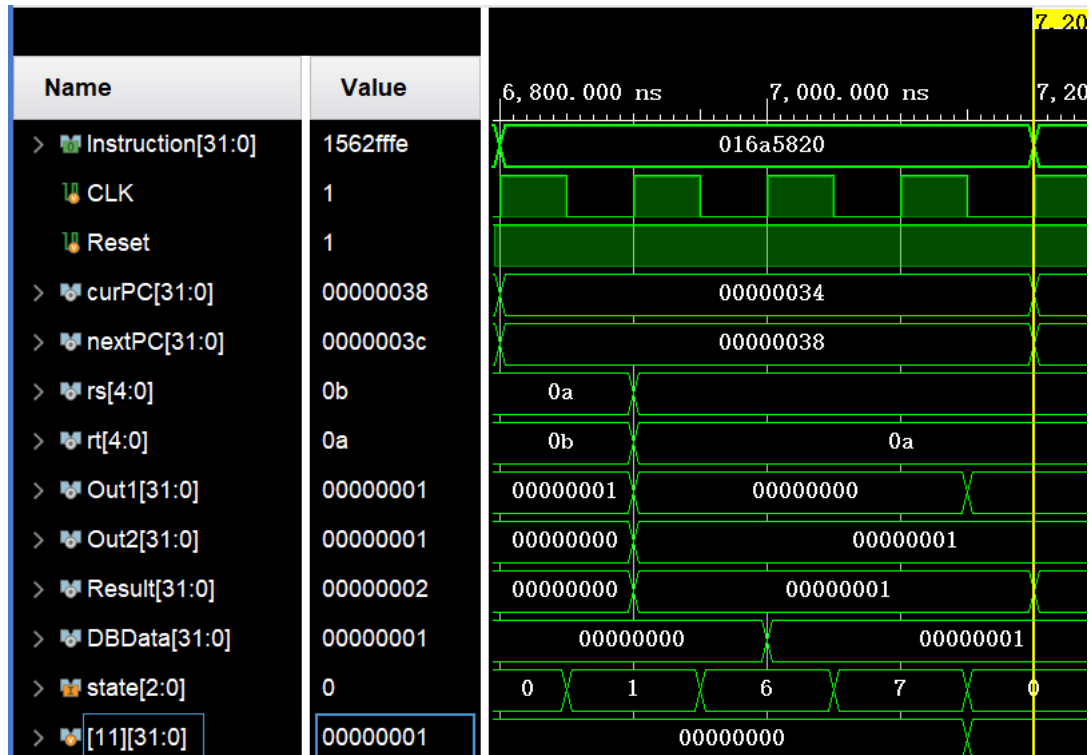
14.slti      6000ns-6400ns      状态码 0-1-6-7      \$10=1



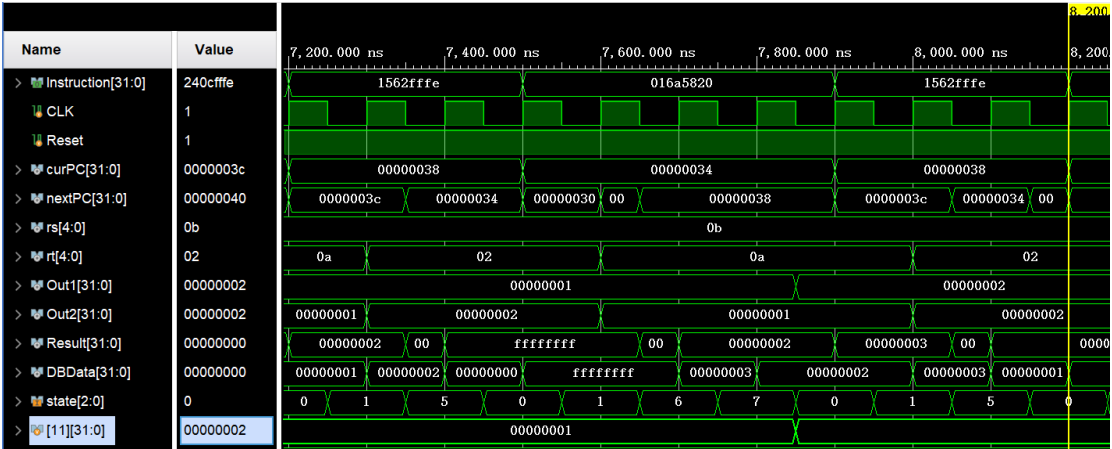
15. slti      6400ns-6800ns      状态码 0-1-6-7      \$11=0



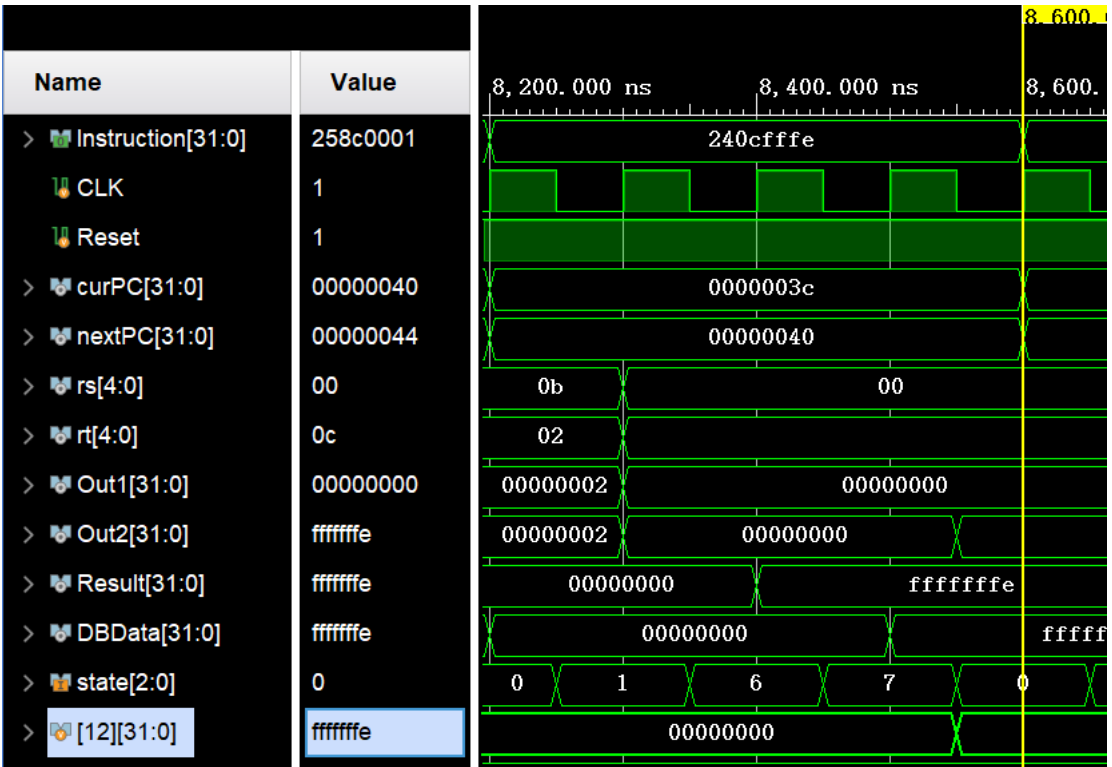
16.add      6800ns-7200ns      状态码 0-1-6-7      \$11=1



17.bne 7200ns-7500ns 状态码 0-1-5 转到 00000034  
add 7500ns-7900ns 状态码 0-1-6-7 \$11=2  
bne 7900ns-8200ns 状态码 0-1-5 继续下一条指令

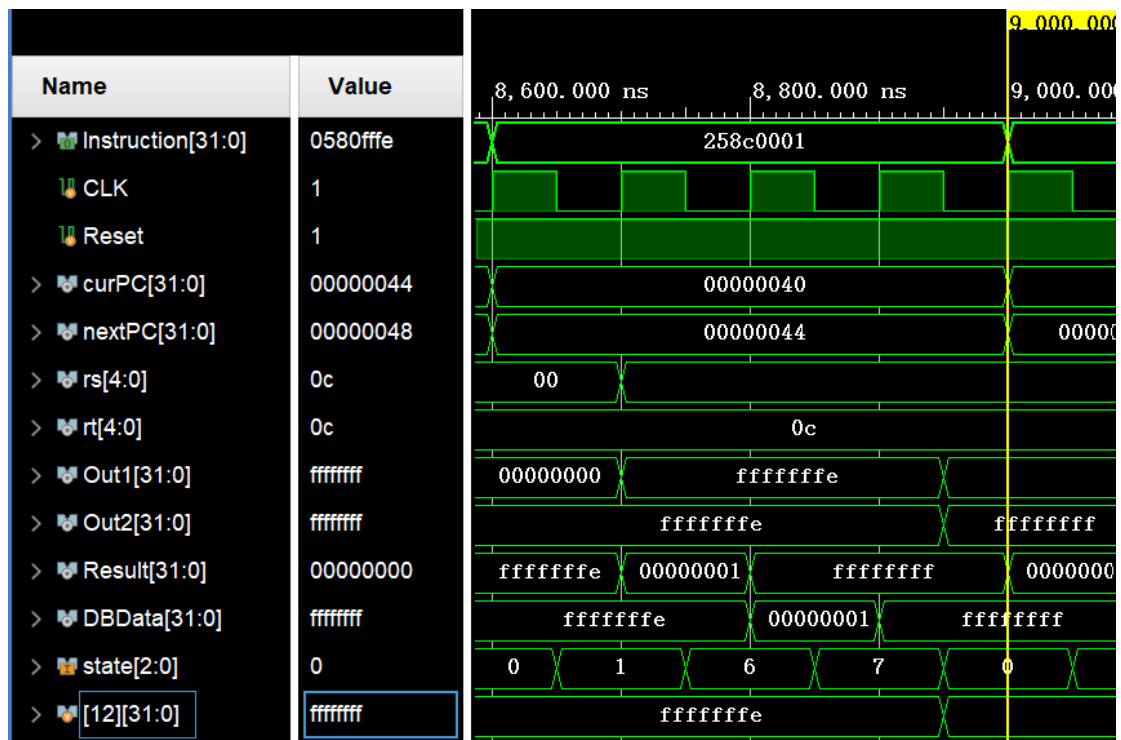


18.addiu 8200ns-8600ns 状态码 0-1-6-7 \$12=-2

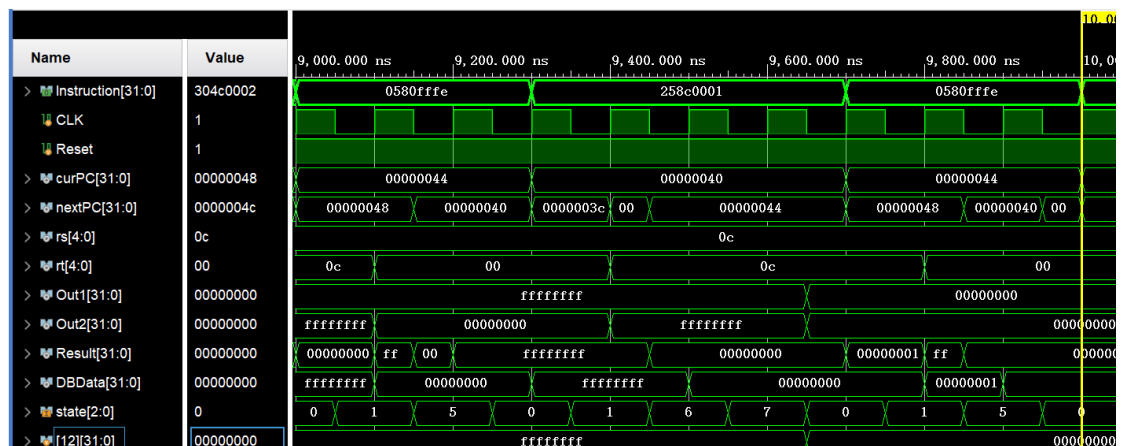




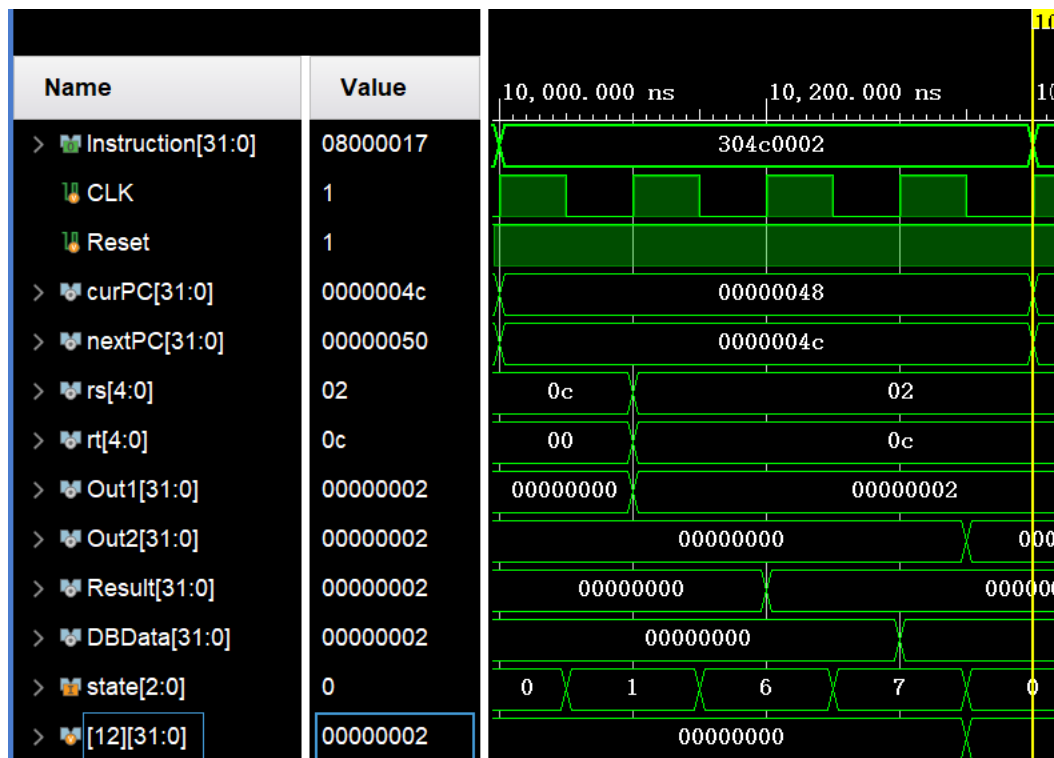
19. addiu 8600ns-9000ns 状态码 0-1-6-7 \$12=-1



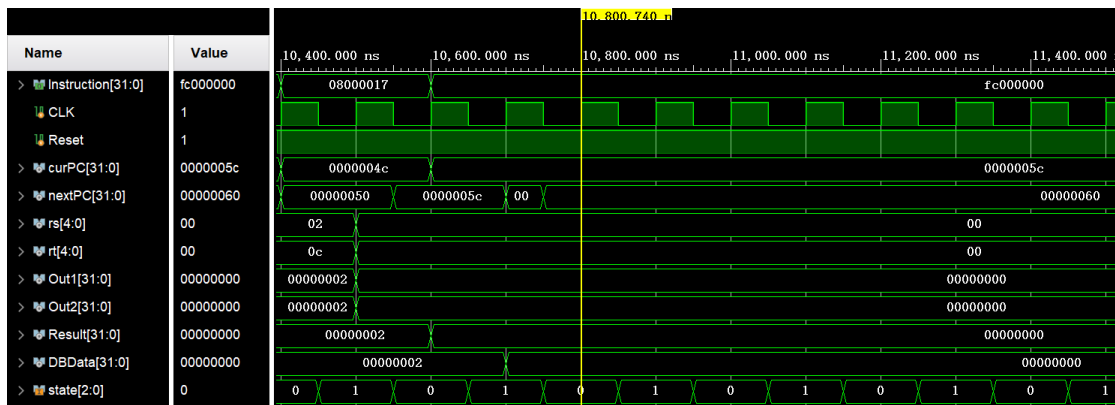
20. bltz 9000ns-9300ns 状态码 0-1-5 小于零跳转到 00000040  
 addiu 9300ns-9700ns 状态码 0-1-6-7 \$12=0  
 bltz 9700ns-10000ns 状态码 0-1-5 继续下一条指令



21.addi 10000ns-10400ns 状态码 0-1-6-7 \$12=2



22. j 10400ns-10600ns 状态码 0-1 跳转到 5C  
halt 结束



进行 Basys 烧写开发板（与单周期 CPU 一样）

设计思路：

- 1、实现 CPU 在板上运行需要两个时钟信号，CPU 工作时钟和 Basys3 板系统时钟。CPU 工作时钟即为按键，是 CPU 正常工作时钟信号，按键必须进行消抖处理；Basys3 板系统时钟即为板提供的正常工作时钟信号，即为 100MHZ。Basys3 板系统时钟信号引脚对应管脚 W5。
- 2、每个按键周期，4 个数码管都必须刷新一次。数码管位控信号 AN3-AN0 是

1110、1101、1011、0111，为0时点亮该数码管，当然，还应该为数码管各位“1gfedcba”引脚输出信号，最高位为“1”。比如，“当前PC值”低8位中的高4位和低4位，必须经下页转换后送给数码管各引脚。

3. 显示模块设计大概分为4个部分：

- (1) 对 Basys3 板系统时钟信号进行分频，分频的目的用于计数器；
- (2) 生成计数器，计数器用于产生4个数。这4数用于控制4个数码管；
- (3) 根据计数器产生的数生成数码管相应的位控信号（输出）和接收CPU来的相应数据；
- (4) 将从CPU接收到的相应数据转换为数码管显示信号，再送往数码管显示（输出）。

开关说明：（以下数据都来自CPU）（SW15、SW14、SW0为Basys3板上开关名，BTN0为按键名）

开关 SW\_in (SW15、SW14) 状态情况如下。显示格式：左边两位数码管 BB：右边两位数码管 BB。以下是数码管的显示内容。

SW\_in = 00: 显示 当前 PC 值: 下一条指令 PC 值

SW\_in = 01: 显示 RS 寄存器地址: RS 寄存器数据

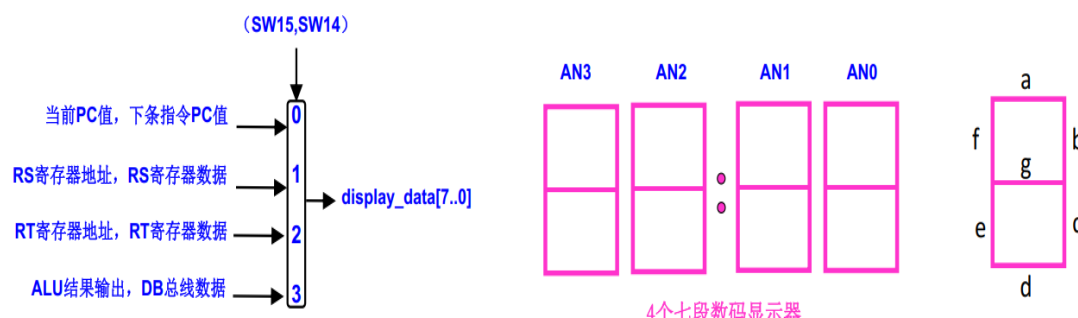
SW\_in = 10: 显示 RT 寄存器地址: RT 寄存器数据

SW\_in = 11: 显示 ALU 结果输出 : DB 总线数据。

复位信号（reset）接开关 SW0，按键（单脉冲）接按键 BTN0。

另外：

- 1、7段数码管的位控信号 AN3-AN0，每组编码中只有一位为0（亮），其余都是1（灭）。
  - 2、七段数码显示器编码与引脚对应关系为（左到右，高到低）：七段共阳极数码管->1gfedcba；七段共阴极数码管->0gfedcba。
  - 3、必须有足够的刷新频率，频率太高或太低都不成，系统时钟必须适当分频，否则效果达不到。
- 指令执行采用单步（按键控制）执行方式，由开关（SW15、SW14）控制选择查看数码管上的相关信息，地址和数据。地址或数据的输出经以下模块代码转换后接到数码管上。



综上，显示模块大概分为4个部分：

1. CLK\_slow: 对 Basys3 板系统时钟信号进行分频，分频的目的用于计数器。
2. counter: 生成计数器，计数器用于产生4个数。这4数用于控制4个数码管。根据计数器产生的数生成数码管相应的位控信号(输出)和接收CPU来的相应数据。
3. Keyboard\_slow: 按键信号消抖。
4. change: 将从CPU接收到的相应数据转换为数码管显示信号，再送往数码管显示(输出)
5. Display\_7SegLED: 数码管显示输出。

## CLK\_slow(分频)

时钟分频的原理是定义一个计数器 counter，每当原始时钟（开发板的时钟 BasysCLK）到来时加1。当计

数器满一定的值后，再将分频时钟 CLK\_slow 翻转一次，这样就将 CLK 进行了分频。

设计分析：

Input：

clk: //Basys 板子的全局高频时钟。

reset: //系统复位信号。

Output：

clk\_sys: //分频后的时钟信号。

代码实现：

```
`timescale 1ns / 1ps

module CLK_slow(
    input clk, //原时钟
    input reset, //重置时间
    output reg clk_sys //分频后的 basys 时钟
);

parameter limit = 100000;
integer counter;

always @(posedge clk or negedge reset)
    if (!reset)
        counter <= 32'b0;
    else
        begin
            counter <= counter + 1'b1;
            if (counter == limit)
                begin
                    counter <= 32'b0;
                    clk_sys <= ~clk_sys;
                end
        end
end
endmodule
```

## Keyboard\_CLK(消抖)

Basys3 板采用的是机械按键，在按下按键时按键会出现人眼无法观测但是系统会检测到的抖动变化，这可能会使短时间内电平频繁变化，导致程序接收到许多错误的触发信号而出现许多不可知的错误。

按键按下的一刻，存在一段时间的抖动，同时在释放按键的一段时间里也是存在抖动的，这就可能导致状态在识别的时候可能检测为多次的按键，因为运行过程中普通的检测一次状态 key 为 1 就执行一次按键操作。所以我们在使用按键时往往需要消抖。

设计分析

Input：

clk\_sys        //分频后的时钟  
reset         //重置信号  
key            //原始按键信号  
Input:  
debkey        //消抖后的按键信号

代码实现:

```
`timescale 1ns / 1ps

module Keyboard_CLK(
    input clk_sys,
    input reset,
    input key, //原始按键信号
    output debkey //消抖后的按键信号
);

reg key_rrr, key_rr, key_r;

always @(posedge clk_sys or negedge reset)
    if (!reset)
        begin
            key_rrr <= 1'b1;
            key_rr <= 1'b1;
            key_r <= 1'b1;
        end
    else
        begin
            key_rrr <= key_rr;
            key_rr <= key_r;
            key_r <= key;
        end

assign debkey = key_rrr & key_rr & key_r; //只有当连续三次采样的按键状态都为高电平时, debkey 才会为高电平

endmodule
```

## counter (计数)

使用计数器记录时钟脉冲, 当达到阈值时, 循环切换激活位。以实现动态刷新效果。

设计分析:

Input:

clk            //时钟信号  
Reset          //重置信号

Output:

AN                    //激活信号，用于控制数码管显示位置

代码实现:

```
`timescale 1ns / 1ps

module counter(
    input clk,
    input Reset,
    output reg[3:0] AN //控制数码管显示位置
);

reg [16:0] counter;
parameter limit = 100000;

initial begin
    counter <= 0;
    AN <= 4'b0111;
end

always @(posedge clk)
begin
    if (Reset == 0) begin
        counter <= 0;
        AN <= 4'b0000;
    end else begin
        counter <= counter + 1;
        if (counter == limit)
            begin
                counter <= 0;
            end
        case(AN)
            4'b1110 : begin
                AN <= 4'b1101;
            end
            4'b1101 : begin
                AN <= 4'b1011;
            end
            4'b1011 : begin
                AN <= 4'b0111;
            end
            4'b0111 : begin
                AN <= 4'b1110;
            end
            4'b0000 : begin
                AN <= 4'b0111;
            end
        endcase
    end
end
```

```

        end
    endcase
end
end
end
endmodule

```

## change（取值）

取数模块根据选择信号（SW）和显示位选择信号（AN）动态选择当前需要显示的数据。

设计分析：

Input：

myCLK

Reset

AN                   //数码管显示的位置决定信号

SW                   //显示的数据内容决定信号

aluResult,

curPC,

newPC,

writeData,

regRs,

regRt,

instruction

Output：

store            //当前选中的 4 位数据

代码实现：

```

`timescale 1ns / 1ps

module change(
    input myCLK,
    input Reset,
    input [3:0] AN, //数码管显示的位置决定信号
    input [2:0] SW, //显示的数据内容决定信号
    input [31:0] aluResult,
    input [31:0] curPC,
    input [31:0] newPC,
    input [31:0] writeData,
    input [31:0] regRs,
    input [31:0] regRt,
    input [31:0] instruction,
    output reg [3:0] store
);

```

```

always@ (myCLK) begin
    if (!Reset)
        store <= 4'b0000;
    else begin
        case(AN)
            4'b1110 : begin // AN0
                case(SW)
                    2'b00: store <= newPC[3:0];
                    2'b01: store <= regRs[3:0];
                    2'b10: store <= regRt[3:0];
                    2'b11: store <= writeData[3:0];
                endcase
            end
            4'b1101 : begin // AN1
                case(SW)
                    2'b00: store <= newPC[7:4];
                    2'b01: store <= regRs[7:4];
                    2'b10: store <= regRt[7:4];
                    2'b11: store <= writeData[7:4];
                endcase
            end
            4'b1011 : begin // AN2
                case(SW)
                    2'b00: store <= curPC[3:0];
                    2'b01: store <= instruction[24:21];
                    2'b10: store <= instruction[19:16];
                    2'b11: store <= aluResult[3:0];
                endcase
            end
            4'b0111 : begin // AN3
                case(SW)
                    2'b00: store <= curPC[7:4];
                    2'b01: store <= { 3'b000,instruction[25] };
                    2'b10: store <= { 3'b000,instruction[20] };
                    2'b11: store <= aluResult[7:4];
                endcase
            end
        endcase
    end
end

endmodule
//SW_in = 00: 显示 当前 PC 值:下条指令 PC 值

```



```
//SW_in = 01: 显示 RS 寄存器地址:RS 寄存器数据  
//SW_in = 10: 显示 RT 寄存器地址:RT 寄存器数据  
//SW_in = 11: 显示 ALU 结果输出 :DB 总线数据。
```

## Display\_7SegLED(显示)

该模块将 4 位二进制数据转换为数码管显示信号

设计分析:

Input:

display\_data           //需要显示的 4 位数据

Output:

dispcode               //数码管显示信号

代码实现:

```
`timescale 1ns / 1ps  
  
module Display_7SegLED(  
    input [3:0] display_data, //选择信号  
    output reg [7:0] dispcode //输出信号  
);  
  
always @ (display_data) begin  
    case(display_data)  
        4'b0000 : dispcode = 8'b1100_0000; // 0  
        4'b0001 : dispcode = 8'b1111_1001; // 1  
        4'b0010 : dispcode = 8'b1010_0100; // 2  
        4'b0011 : dispcode = 8'b1011_0000; // 3  
        4'b0100 : dispcode = 8'b1001_1001; // 4  
        4'b0101 : dispcode = 8'b1001_0010; // 5  
        4'b0110 : dispcode = 8'b1000_0010; // 6  
        4'b0111 : dispcode = 8'b1101_1000; // 7  
        4'b1000 : dispcode = 8'b1000_0000; // 8  
        4'b1001 : dispcode = 8'b1001_0000; // 9  
        4'b1010 : dispcode = 8'b1000_1000; // A  
        4'b1011 : dispcode = 8'b1000_0011; // B  
        4'b1100 : dispcode = 8'b1100_0110; // C  
        4'b1101 : dispcode = 8'b1010_0001; // D  
        4'b1110 : dispcode = 8'b1000_0110; // E  
        4'b1111 : dispcode = 8'b1000_1110; // F  
        default : dispcode = 8'b0000_0000; // 不亮  
    endcase  
end  
  
endmodule
```

## basys3（顶层模块）

在顶层模块中，通过实例化各子模块，实现信号的传递与功能的综合。

Input:

clk,                    //时钟信号  
SW,                    //选择输出信号  
Reset,                 //重置按钮  
Button,                //运行按钮

Output:

AN,                    //数码管选择信号  
Out                    //数码管输入信号

代码实现:

```
`timescale 1ns / 1ps

module basys3(
    input clk, //时钟信号
    input [1:0] SW, //选择输出信号
    input Reset, //重置按钮
    input Button, //运行按钮
    output [3:0] AN, //数码管选择信号
    output [7:0] Out //数码管输入信号
);

wire [31:0] curPC; //当前的 PC
wire [31:0] DBData; //DB 总线
wire [31:0] Instruction; //当前地址的指令
wire [31:0] regRs; //寄存器组 rs 寄存器的地址
wire [31:0] regRt; //寄存器组 rt 寄存器的地址
wire [5:0] Opcode; //指令的 OP 码
wire [31:0] Out1, Out2; //Out1 为 rs 寄存器中的值; Out2 为 rt 寄存器中的值
wire [31:0] Result; //ALU 运算后得到的结果
wire myCLK; //消抖后的信号
wire clk_sys; //分频后的信号
wire [3:0] store; //记录当前要显示位的值
wire [31:0] nextPC; //下一条 PC

multiCycleCPU my_SCPU(myCLK, Reset, regRs, regRt, Opcode, Out1, Out2,
    curPC, nextPC, Result, DBData, Instruction); //CPU
CLK_slow cd(clk, Reset, clk_sys); //分频
Keyboard_CLK my_key(clk_sys, Reset, Button, myCLK); //消抖
counter my_ct(clk, Reset, AN); //计数
Display_7SegLED led(store, Out); //显示
```

```
change my_cg(myCLK, Reset, AN, SW, Result, curPC, nextPC, DBData, Out1,
Out2, Instruction, store); //取值

endmodule
```

至此多周期 CPU 整个的代码设计到此结束，接下来要进行烧板操作：综合代码后进行引脚分配，根据上面说到的引脚和接口的对应关系来进行分配，最后生成约束文件如下：

```
set_property PACKAGE_PIN W4 [get_ports {AN[3]}]
set_property PACKAGE_PIN V4 [get_ports {AN[2]}]
set_property PACKAGE_PIN U4 [get_ports {AN[1]}]
set_property PACKAGE_PIN U2 [get_ports {AN[0]}]
set_property PACKAGE_PIN V7 [get_ports {Out[7]}]
set_property PACKAGE_PIN U7 [get_ports {Out[6]}]
set_property PACKAGE_PIN V5 [get_ports {Out[5]}]
set_property PACKAGE_PIN U5 [get_ports {Out[4]}]
set_property PACKAGE_PIN V8 [get_ports {Out[3]}]
set_property PACKAGE_PIN U8 [get_ports {Out[2]}]
set_property PACKAGE_PIN W6 [get_ports {Out[1]}]
set_property PACKAGE_PIN W7 [get_ports {Out[0]}]
set_property PACKAGE_PIN R2 [get_ports {SW[1]}]
set_property PACKAGE_PIN T1 [get_ports {SW[0]}]
set_property PACKAGE_PIN W5 [get_ports clk]
set_property PACKAGE_PIN V17 [get_ports Reset]
set_property PACKAGE_PIN U18 [get_ports Button]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports Button]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports Reset]
```



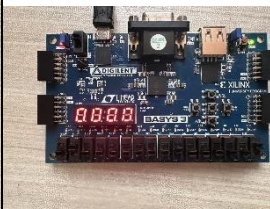

最后生成比特流文件就可以在开发板上进行显示了

SW\_in = 00: 显示 当前 PC 值:下条指令 PC 值  
SW\_in = 01: 显示 RS 寄存器地址:RS 寄存器数据  
SW\_in = 10: 显示 RT 寄存器地址:RT 寄存器数据  
SW\_in = 11: 显示 ALU 结果输出 :DB 总线数据。

逐一指令验证篇幅过长，下面只对

0x0000001C	jal 0x00000050
------------	----------------

这一跳转指令进行验证展示：

pc: nextpc	rs 地址: rs 值	Rt 地址: rt 值	Alu: DBdata
			

## 六. 实验心得

- 1.在制作单周期CPU时遇到的问题在制作多周期CPU前已经解决了，所以多周期CPU制作起来会更加快捷。
2. 多周期 CPU 设计与实现与单周期 CPU 设计与实现这两个实验有非常多的相似之处——InstructionMemory、RegisterFile、DataMemory、ALU 等底层模块是完全相同的。因此这些模块可以直接使用在项目一中已经实现完成的代码，大大简化了实验步骤。从数据通路图可以看出，ControlUnit 模块需要重写，此外还增加了多周期 CPU 特有的 IR、DR 这几个寄存器。IR、ADR、BDR、DBDR 和 ALUoutDR 都可以看做是 D 触发器，其实代码比较容易实现，而且它们几乎是相同的，因此可以写一份代码，然后实例化为多个模块即可。多周期 CPU 的 ControlUnit 新增加了时钟作为输入，这是因为 ControlUnit 内部要实现状态转移的功能。容易想到，ControlUnit 主要完成两件事——一是状态转移，根据当前指令 opcode 以及当前状态推断出下一个状态；二是输出函数，也就是根据当前指令 opcode 和当前所处状态生成各种控制信号来控制 CPU 中的其他部件。所以，实现了状态转移，多周期 CPU 就和单周期的流程区别不大。
- 3.我更加深入地理解了单周期和多周期 CPU 的工作原理，理解了 ALU 等底层模块的功能及其实现、数据通路的构建、各种寄存器的作用以及数据选择器在数据通路中的重要作用，还通过实例更清晰地认识了状态机的功能及其实现。总的来说，对计算机内部组成（主要是 CPU 的组成）有了更加全面和深入的认识。
4. Verilog 语言的调试与以往学的 C/C++、Python 高级程序设计语言的调试方法不同。在其他语言中，调试方法往往是观察特定的变量在函数中的变化情况，以及分支、循环条件的判断等。而在 Verilog 语言中有所不同。根据我自己的经验，我认为调试 Verilog 要遵从该语言的“模块化”这一特性：当发现仿真结果的某个信号不正确的时候，首先应该检查的是该信号直接出现的底层模块其次考虑底层模块更高一层的模块——是不是底层模块实例化时不小心遗漏了某个引脚？是不是搞混了相似的变量名称？如果还是没有找出错误，继续按照这样的步骤检查更高层的模块，直到顶层模块。
5. 这学期的计算机组成原理理论课和实验课让我对计算机 CPU 的认识深入了很多。相信在以后的课程学习中，甚至是更未来的科研实习和工作中都会用到我在计组实验课上得到的知识或方法。

ps: (汇编器的代码在验收时没有完成，现在在实验报告中呈现了)