



# 《计算机组成原理与接口技术实验》

## 实验报告

(实验三)

学 院 名 称 : 数据科学与计算机学院

学 生 姓 名 : 罗剑杰

学 号 : 15331229

专业 (班级) : 15 软件工程三 (6) 班

时 间 : 2017 年 05 月 18 日

成绩：

## 实验三：多周期CPU设计

## 一. 实验目的

- (1) 认识和掌握多周期数据通路原理及其设计方法；
- (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

## 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：**(说明：操作码按照以下规定使用，都给每类指令预留扩展空间，后续实验相同。)**

## ==&gt;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) addi rt, rs, **immediate**

000010	rs(5 位)	rt(5 位)	<b>immediate(16 位)</b>
--------	---------	---------	------------------------

功能：rt ← -rs + (sign-extend)**immediate**

## ==&gt;逻辑运算指令

(4) or rd, rs, rt

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

功能：rd ← -rs | rt

(5) and rd, rs, rt

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

功能：rd ← -rs &amp; rt

(6) ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	<b>immediate(16 位)</b>
--------	---------	---------	------------------------

功能：rt ← -rs | (zero-extend)**immediate**

## ==&gt;移位指令

(7) sll rd, rt, sa

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

功能：rd ← -rt &lt;&lt; (zero-extend)sa, 左移 sa 位, (zero-extend)sa

## ==&gt;比较指令

(8) slt rd, rs, rt 带符号

100110	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

(9) slti rt, rs,immediate 带符号

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

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

### ==>存储器读写指令

(10) sw rt, immediate(rs)

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

功能: memory[rs+ (sign-extend)immediate]<-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。

(11) lw rt, immediate(rs)

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

功能: rt <- memory[rs + (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

(12) 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 <- pc + 4 + (sign-extend)immediate <<2 else pc <- pc + 4

### ==>跳转指令

(13) j addr

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

功能: pc <- {(pc+4)[31..28],addr[27..2],0,0}, 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。

(14) jr rs

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

功能: pc <- rs, 跳转。

### ==>调用子程序指令

(15) jal addr

111010	addr[27..2]			
--------	-------------	--	--	--

功能: 调用子程序, pc <- {(pc+4)[31..28],addr[27..2],0,0}; \$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**

(16) 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。CPU 在处理指令时，一般需要经过以下几个阶段：

(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。

实验中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况怎样？要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的 CPU。

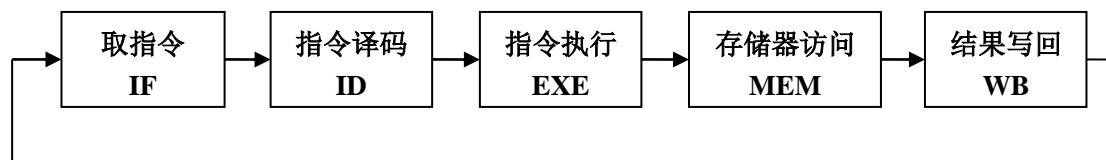
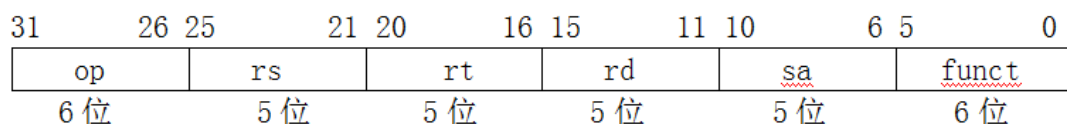
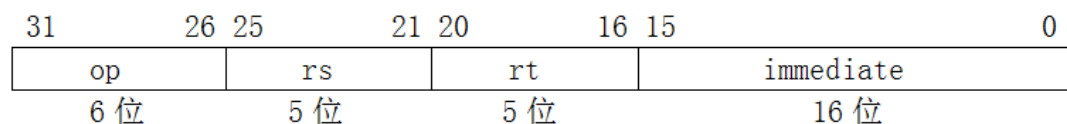
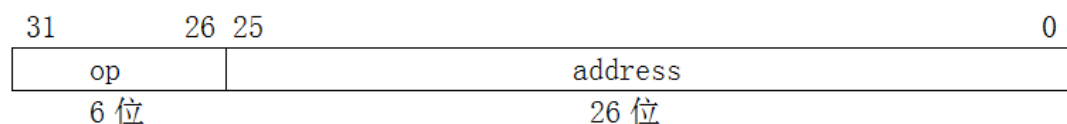


图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式：

**R 类型：****I 类型：****J 类型：**

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。

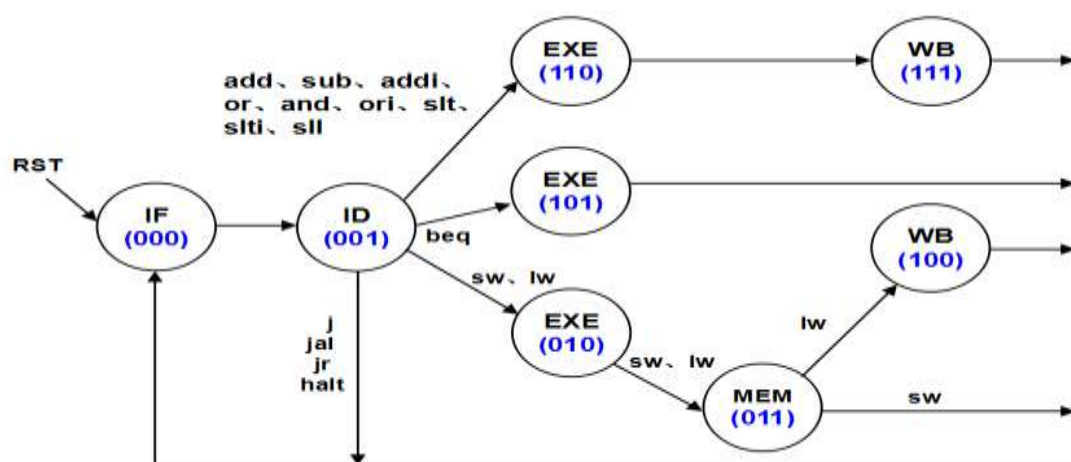


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 IF 状态转移到 ID 就是无条件的；有些是有条件的，例如 ID 或 EXE 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。

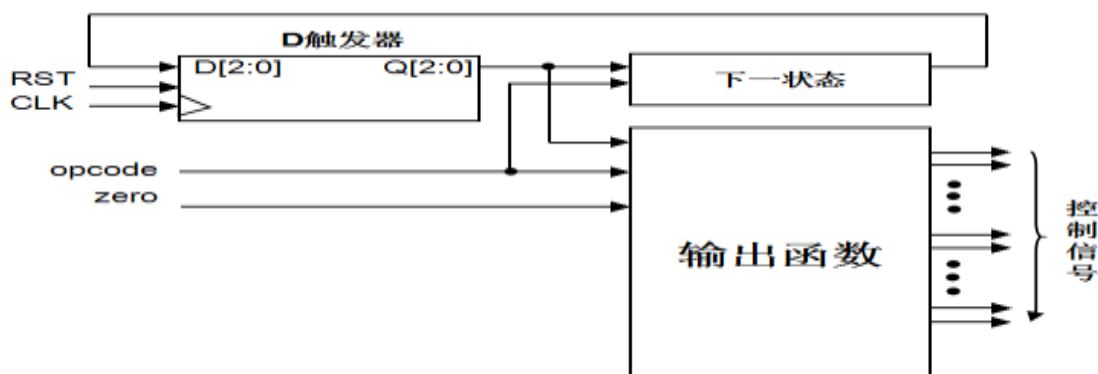


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，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。

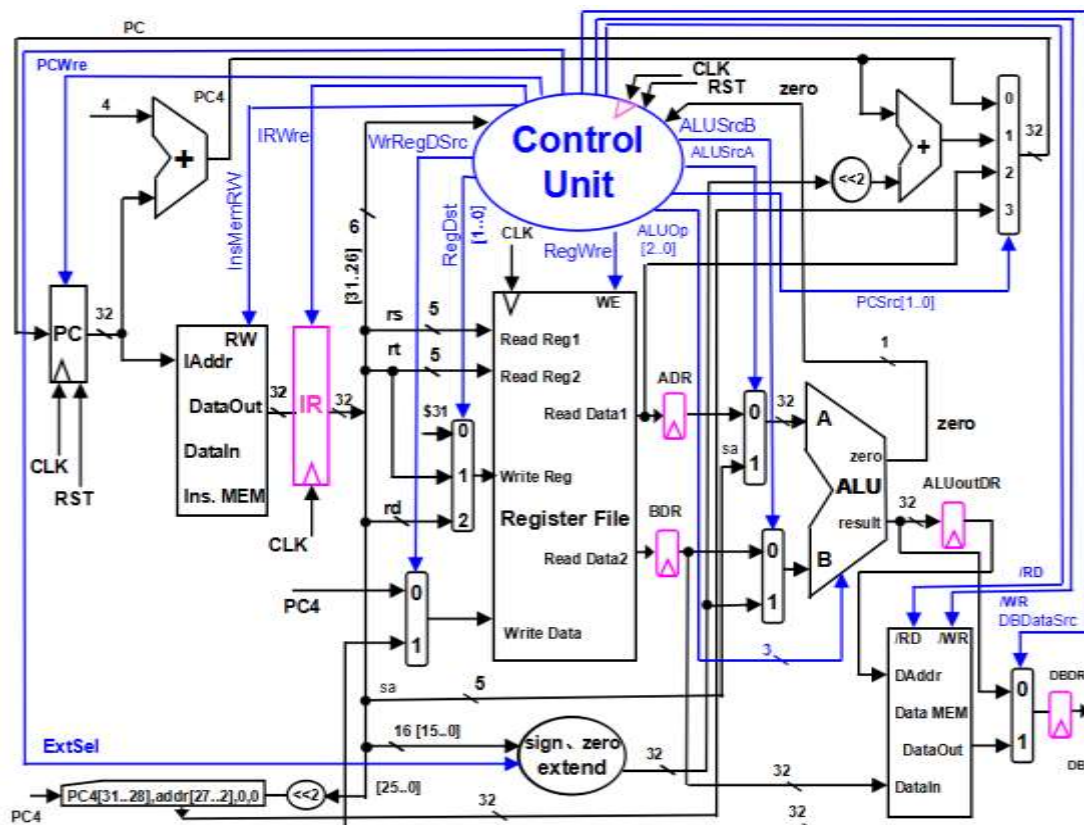


图 4 多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 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, 先给出地址, 然后由读或写信号控制操作。对于寄存器组, 读操作时, 给出寄存器地址 (编号), 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, 在时钟边沿触发写入。图中控制信号功能如表 1 所示, 表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号**，其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

### 表 1 控制信号作用

控制信号名	状态 “0”	状态 “1”
RST	对于 PC, 初始化 PC 为程序首地址	对于 PC, PC 接收下一条指令地址
PCWre	PC 不更改, 相关指令: halt, 另外, 除 ‘000’ 状态之外, 其余状态慎改 PC 的值。	PC 更改, 相关指令: 除指令 halt 外, 另外, 在 ‘000’ 状态时, 修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出, 相关指令: add、sub、addi、or、and、ori、beq、slt、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、beq、slt、sll	来自 sign 或 zero 扩展的立即数, 相关指令: addi、slti、ori、lw、sw

<b>DBDataSrc</b>	来自 ALU 运算结果的输出,相关指令: add、sub、addi、or、and、ori、slt、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、j、sw、jr、halt	寄存器组寄存器写使能, 相关指令: add、sub、addi、or、and、ori、slt、slti、sll、lw、jal
<b>WrRegDSrc</b>	写入寄存器组寄存器的数据来自 pc+4(pc4) , 相关指令: jal, 写 \$31	写入寄存器组寄存器的数据来自存储器、寄存器组寄存器和 ALU 运算结果, 相关指令: add、addi、sub、or、and、ori、slt、slti、sll、lw
<b>InsMemRW</b>	写指令存储器	读指令存储器(Ins. Data)
<b>/RD</b>	读数据存储器, 相关指令: lw	输出高阻态
<b>/WR</b>	写数据存储器, 相关指令: sw	无操作
<b>IRWre</b>	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后, 这个信号也接着发出, 在时钟上升沿, IR 接收从指令存储器送来的指令代码。与每条指令都相关。
<b>ExtSel</b>	(zero-extend) <b>immediate</b> , 相关指令: ori;	(sign-extend) <b>immediate</b> , 相关指令: addi、slti、lw、sw、beq;
<b>PCSrc[1..0]</b>	00: pc←-pc+4, 相关指令: add、addi、sub、or、ori、and、slt、slti、sll、sw、lw、beq(zero=0); 01: pc←-pc+4+(sign-extend) <b>immediate</b> , 相关指令: beq(zero=1); 10: pc←-rs, 相关指令: jr; 11: pc←-{pc(31..28),addr[27..2],0,0}, 相关指令: j、jal;	
<b>RegDst[1..0]</b>	写寄存器组寄存器的地址, 来自: 00: 0x1F(\$31), 相关指令: jal, 用于保存返回地址 (\$31←-pc+4) ; 01: rt 字段, 相关指令: addi、slti、ori、lw; 10: rd 字段, 相关指令: add、sub、or、and、slt、sll; 11: 未用;	
<b>ALUOp[2..0]</b>	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, 指令地址输入端口

DataIn, 存储器数据输入端口

DataOut, 存储器数据输出端口

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, 数据地址输入端口

DataIn, 存储器数据输入端口

DataOut, 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, 其地址输入端口 (rt、rd)

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, 在时钟上升沿写入

**IR: 指令寄存器**, 用于存放正在执行的指令代码**ALU: 算术逻辑单元**

result, ALU 运算结果

zero, 运算结果标志, 结果为 0 输出 1, 否则输出 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	if (rega<regb &&(( rega[31] == 0 && regb[31]==0)    (rega[31] == 1 && regb[31]==1))) result = 1; else if (rega[31] == 0 && regb[31]==1) result = 0; else if ( rega[31] == 1 && regb[31]==0) result = 1; else result = 0;	比较 A 与 B 带符号
011	$Y = (A < B) ? 1 : 0$	比较 A 与 B 不带符号
100	$Y = B \ll A$	B 左移 A 位
101	$Y = A \vee B$	或
110	$Y = A \wedge B$	与
111	$Y = A \oplus B$	异或

值得注意的问题, 设计时, 用模块化的思想方法设计, 关于 Control Unit 设计、ALU 设计、存储器设计、寄存器组设计等等, 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。

**五. 实验分析与设计****1. 实验总体过程:**

- 1) 阅读相关资料, 总体上把握多周期cpu和单周期cpu的异同。
- 2) 将一些有特定功能的子模块按照新的要求进行重构 (如ALU32模块), 保留单



周期cpu中可以复用的模块（如extend）构建一些新的子模块（如DataReg）。

- 3) 自己思考写出一个测试程序，按照要求翻译成机器01码，以供测试。
- 4) 构建ContorlUnit模块，主要关注点在于OutputFun子模块，根据不同的阶段生成相应的控制信号。
- 5) 构造顶层模块，按照新的连线要求来构建，然后进行模拟。
- 6) 在模拟的过程中不断发现问题，然后通过单步执行和设置断点的方式进行调试，逐渐修正一些一开始控制信号的产生的疏漏，还有测试程序的问题，进一步加深对多周期cpu的理解。
- 7) 调试至可以成功运行，检查寄存器内值与预期相符合，在从一开始以时间周期为步长，一步步模拟，检查程序运行的指令顺序直到符合预期。
- 8) 完成调试，书写实验报告

## 2. 总体思路

- 1) 多周期CPU与单周期CPU的区别在于，多周期CPU在一个时间周期内只执行一条指令中的一个阶段，而单周期CPU是在一个时间周期内执行一条完整的指令。由于并不是所有的指令都需要执行最多的5个步骤，所以多周期CPU的执行时间会比单周期CPU的略快，思考的重点就是如何实现数据内容的缓存，如何在时钟周期更替的时候使得控制信号量和相关寄存器里面的值符合预期的要求。
- 2) 多周期CPU与流水线CPU的区别在于，多周期CPU指令的执行也还是顺序执行的，而流水线CPU的执行时属于并行执行的，明白了这一点后，就可以明白在考虑多周期CPU运行过程的时候，集中点都是在一条特定的指令上，不用考虑冒险。
- 3) 增加了j、jr、jal等跳转指令了之后，对于pc的设计复杂了一点在addr的处理的时候，要明白实际存储的地址要向左移两位（即乘以4）来得到真实的地址，原因是因为MIPS架构指令的32位bit定长和存储寄存器以8位bit位定长相互结合导致。

## 3. 提供的测试程序

地址	汇编程序	指令代码				寄存器应有的结果
		op(6)	rs(5)	rt(5)	rd(5)(+sa(5))/immediate(16)	
0x00000000	addi \$1,\$0,6	000010	00000	00001	0000 0000 0000 0110	\$1=6
0x00000004	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010	\$2=2
0x00000008	add \$3,\$2,\$1	000000	00010	00001	00011 xxxxxxxxxxxx	\$3=8
0x0000000C	sub \$4,\$3,\$2	000001	00011	00010	00100 xxxxxxxxxxxx	\$4=6
0x00000010	and \$5,\$4,\$2	010001	00100	00010	00101 xxxxxxxxxxxx	\$5=2
0x00000014	or \$6,\$5,\$1	010000	00101	00001	00110 xxxxxxxxxxxx	\$6=6
0x00000018	slti \$7,\$6,-2	100111	00110	00111	1111 1111 1111 1110	\$7=0
0x0000001C	slti \$8,\$7,2	100111	00111	01000	0000 0000 0000 0010	\$8=1
0x00000020	jal 0x0000003C	111010	00 0000 0000 0000 0000 000011 11			\$31=0x00000024
0x00000024	slt \$9,\$11,\$0	100110	01011	00000	01001 xxxxxxxxxxxx	\$9=0
0x00000028	slt \$10,\$9,\$1	100110	01001	00001	01010 xxxxxxxxxxxx	\$10=1
0x0000002C	slt \$12,\$9,\$1	100110	01001	00001	01100 xxxxxxxxxxxx	\$12=1
0x00000030	sll \$12,\$12,1	011000	xxxxx	01100	01100 00001 xxxxxx	\$12=2
0x00000034	beq \$12,\$2,-2 (=, 转 30)	110100	01100	00010	1111 1111 1111 1110	(回去了一次, 最后\$12=4)
0x00000038	j 0x00000048	111000	00 0000 0000 0000 0000 0100 0010			
0x0000003C	sw \$8,4(\$1)	110000	00001	01000	0000 0000 0000 0100	看 dm 对应位置 (40)
0x00000040	lw \$11,4(\$1)	110001	00001	01011	0000 0000 0000 0100	\$11=1
0x00000044	jr \$31	111001	11111	xxxxx	xxxx xxxx xxxx xxxx	
0x00000048	halt	111111	00000	00000	00 0 0000 0000	

## 4. 主要控制信号的思考

和单周期CPU不同，多周期CPU没有办法只以一条指令为单位来进行信号量的思考出发点，原因是多周期CPU的每个时间周期内做的只是一条指令中的一个阶段，所以如果想要把每一条指令的每一个阶段的所有的信号量的值再罗列出来，显然工作量是巨大的。在多次尝试后我发现，实际上每个信号量的可取值范围只有可能是2种或者多于2种，如果把关注的主体从单个指令转移到当前信号，对于可取值范围多于两种的信号量用case来进行处理；对于可取值范围只有两种的信号量，分析取其中一个取值所满足的所有条件情况（通常取相对简单的那种情况）并用if语句来进行判断，那么另一个复杂的条件取值只需要使用一个else就可以实现了。而可以成为条件的变量也就是只有当前阶段（currState）和当前操作（opcode）这两个变量。

沿用了部分单周期CPU的设计技巧，将不同的op通过位的运算来构造不同指令的fingerprint，这样大大简化了下面的条件判断的内容。参考表1给出的信号量的取值意义可以得出下面关于所有信号量的思考情况以及实现：

- 1) **PCWre**: 只有当IF阶段的时候才可以为1，其他情况下为0，如果当前是halt指令，那么下一次IF阶段的时候PCWre也应该为0，停止取指令。

```
if (currState == IF) begin
    PCWre = 1;
    if (halt) PCWre = 0;
end
else PCWre = 0;
```

- 2) **ALUSrcA**: 在sll指令的执行阶段（EXE1）的时候才需要为1，其余情况都为0。

```
if (currState == EXE1 && sll) ALUSrcA = 1;
else ALUSrcA = 0;
```

- 3) **ALUSrcB**: 在执行阶段EXE1或者EXE3时，如果正在执行的指令是addi、slli、ori、lw或者sw的时候才为1，其余的时候为0。

```
if (currState == EXE1 || currState == EXE3) begin
    if (addi || slli || ori || lw || sw) ALUSrcB = 1;
    else ALUSrcB = 0;
end
else ALUSrcB = 0;
```

- 4) **DBDataSrc**: 当在MEM阶段而且执行的指令是lw的时候才为1，其余的情况为0。

```
if (currState == MEM && lw) DBDataSrc = 1;
else DBDataSrc = 0;
```

- 5) **RegWre**: 普通情况下在阶段WB1或者WB2的时候才会考虑是否需要写回寄存器组，这种情况下如果执行beq、j、sw、jr、halt指令的话不需写回，此时信号量为0，其余情况为1；特殊情况是当运行jal指令的ID阶段的时候，因为jal指令没有WB阶段，所以在jal指令的ID阶段，也需要写回，此时RegWre为1。因此和WrRegDSrc信号量一起设置。

- 6) **WrRegDSrc**: 当运行jal指令并且currState是ID的时候为0（此时RegWre为1），其余时刻为0。和RegWre信号一起设置。

```
// Here is the question to deal with the jal
if (currState == WB1 || currState == WB2 || (jal && currState == ID)) begin
    if (jal && currState == ID) begin
        WrRegDSrc = 0;
    end
    else WrRegDSrc = 1;

    if (beq || j || sw || jr || halt) RegWre = 0;
    else RegWre = 1;
end
else RegWre = 0;
```

7) **InsMemRw**: 持续赋值为1。

8) **RD**: 只有在MEM阶段同时执行的指令是lw的时候才为0，其余时刻为1。

```
RD = (currState == MEM && lw) ? 0 : 1;
```

9) **WR**: 只有在MEM阶段同时执行的指令是sw的时候才为0，其余时刻为1。

```
WR = (currState == MEM && sw) ? 0 : 1;
```

10) **IRWre**: 只有在IF阶段的时候才会把指令缓存在寄存器中，此时值为1，否则值为0。

```
IRWre = (currState == IF) ? 1 : 0;
```

11) **ExtSel**: 在ID阶段解码的时候才有可能用到这个信号量，在这个大前提下正在执行的指令是ori的时候信号量为0进行无符号扩展，其余情况为1。

```
ExtSel = (currState == ID && ori) ? 0 : 1;
```

12) **PCSrc**: 根据表1的说明对不同的执行情况进行分类处理：

```
if (beq && zero == 1) PCSrc = 2'b01;
else if (jr) PCSrc = 2'b10;
else if (j || jal) PCSrc = 2'b11;
else PCSrc = 2'b00;
```

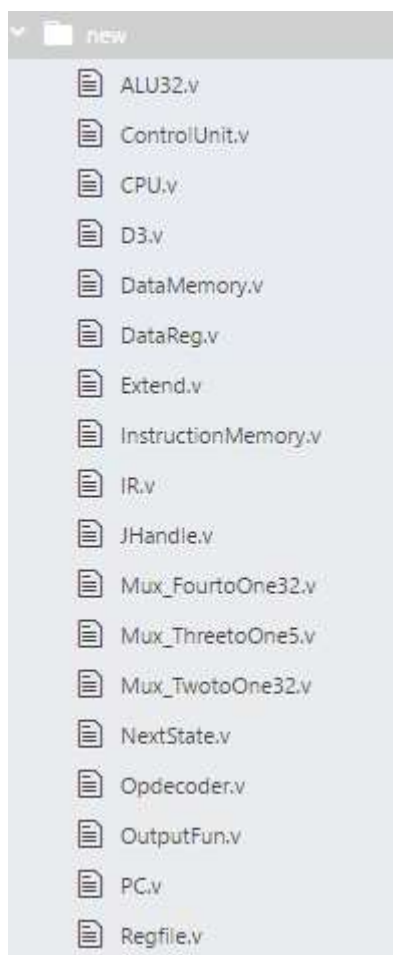
13) **RegDst**: 根据表1的说明对不同的执行情况进行分类处理。

```
if (jal) RegDst = 2'b00;
else if (addi || slti || ori || lw) RegDst = 2'b01;
else if (add || sub || or_ || and_ || slt || sll) RegDst = 2'b10;
```

- 14) **ALUOp**: 当op发生变化的时候, 使用case来判断进行赋值改变。由于太长在此处不截图。

#### 4. 相关模块的设计:

本次设计的所有模块如下图所示:



大体上的模块对应功能和单周期CPU的基本一致,

- 1) **ALU32**: 拿到两个操作数reg<sub>a</sub>, reg<sub>b</sub>, 通过case判断ALUopcode来执行相关逻辑, 得出result, result又会影响zero的输出反馈回ControlUnit。
- 2) **DataMemory**: 里面有一个单位长度为8的reg的数组, 数组长度为60 (足够使用就好), 表示数据内存的大小。通过RD的值来对输出DataOut持续赋值, 在时钟下降沿的时候通过WR来判断是否需要将DataIn的数据存入到相应的内存位置中。(而只有在WB1或者WB2阶段或者jal的ID阶段CU才会把WR改成要存入的状态)。

3) **Extend:** 通过控制信号ExtSel, 来进行相应的符号位扩展。

ExtSel	imm_16最高位	扩展内容
0	0	0
0	1	0
1	0	0
1	1	1

从真值表中不难看出, 当 `ExtSel && imm_16[15] == 1` 为真的时候, 扩展1; 否则扩展0。

4) **InstructionMemory:** 初始化的时候调用系统调用\$readmemb读入已经写好的指令文件(相对路径为”../ ../ ../ins.txt”), 因为指令存储器和数据存储器存储单元宽度一律使用8位, 所以通过RW控制信号来判断是否从内存中连续读取4条记录构建出32位的指令, 指令输出到Opdecoder里面进行解码, Opdecoder再把解码的内容送入到各个需要的模块。

5) **Mux\_ThreetoOne5:** 5位的三选一多路选择器, 通过信号sel来判断应该输出哪一个数据。

6) **Mux\_TwotoOne32:** 同Mux\_ThreetoOne5, 只是输入输出数据为32位。

7) **Mux\_FourtoOne32:** 同Mux\_TwotoOne32, 只是选择的数据有4个。

8) **PC:** 为了保证只有在IF的阶段才更新curPC检测的条件有单周期时候简单的时钟上升沿变成当PCWre和rst改变时, 输出curPC进行改变, 若信号rst为0, 则初始化把起始指令地址设为0, 若rst=1, 则将来自四选一模块的nextPC赋给curPC, 实现pc的更新。当curPC更新时同步更新PC4, 当输入的偏移量更新时同步更新新的pc偏移地址, 将这两个变量output出去以作为其他模块的使用。

9) **Regfile:** 寄存器存储器, 32位作为存储器存储单元宽度, 总共给32个寄存器, 其中0号寄存器不可写, 所有寄存器的在初始化的时候都初始化为0。通过RdReg1addr和RdReg2addr给reg1dataOut和reg2dataOut进行持续的赋值, 当clk下降沿的时候通过信号RegWre进行判断要不要写寄存器。

10) **ControlUnit:** 输入来自Opdecoder送来的opcode, 通过时钟来控制, 来产生所有的信号。ControlUnit模块的设置分成3个子模块, 最大的好处是将关注点分离, 控制信号的产生主要放在OutputFun模块中, NextState模块只进行当前阶段和下一阶段的转换处理, D触发器用来根据时钟信号控制ControlUnit的行为。

11) **CPU:** 顶层模块, 把相关以上的所有的子模块实例化并且连接起来。

12) **仿真模块:** 实例化一个CPU, 把clk和rst传入, 整个单周期CPU就可以开始工作。

## 5. 测试代码运行的实验结果

- 1) 整个程序运行完成后的所有用到的寄存器的值和测试程序的预期值相比较，发现可以完全符合，datamem的存储情况也和预期的一致。从而我们可以知道整个运行的结果是正确的。

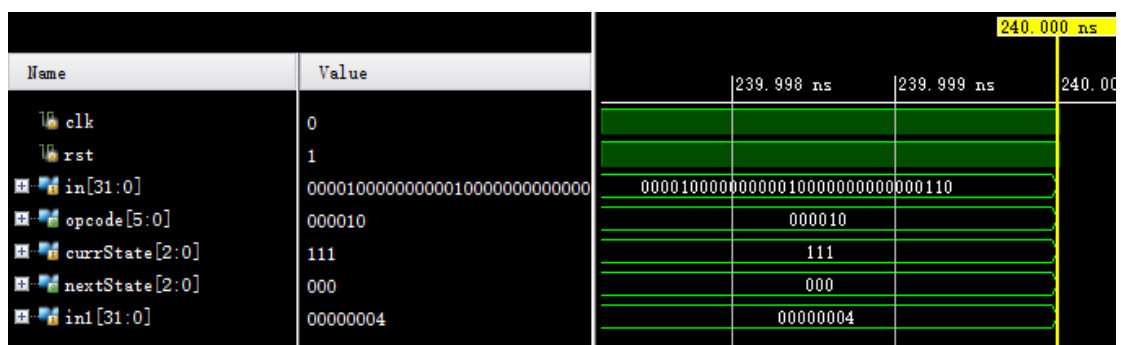
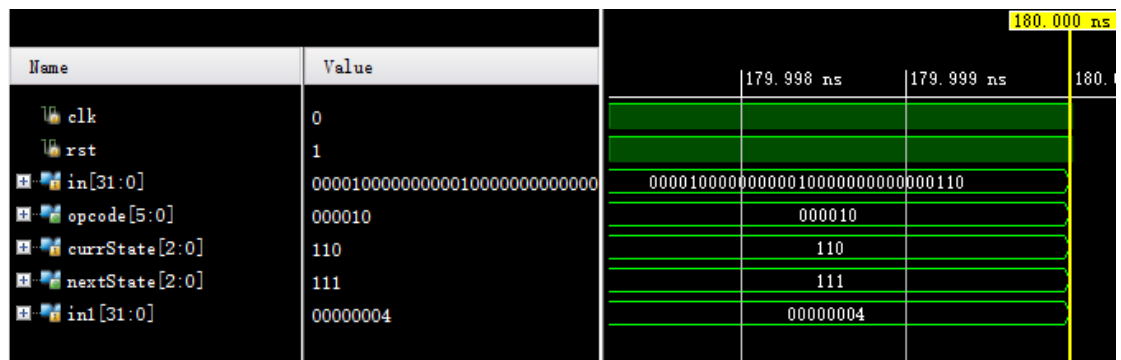
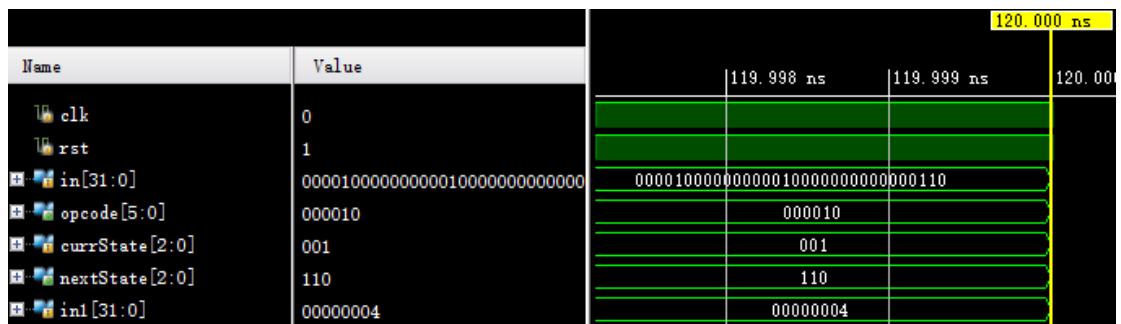
register[1:31][31:0]	00000000000000000000000000000000	00000000000000000000000000000000	110,000
[1][31:0]	00000006	00000006	
[2][31:0]	00000002	00000002	
[3][31:0]	00000008	00000008	
[4][31:0]	00000006	00000006	
[5][31:0]	00000002	00000002	
[6][31:0]	00000006	00000006	
[7][31:0]	00000000	00000000	
[8][31:0]	00000001	00000001	
[9][31:0]	00000000	00000000	
[10][31:0]	00000001	00000001	
[11][31:0]	00000001	00000001	
[12][31:0]	00000004	00000004	
[31][31:0]	00000024	00000024	

- 2) 我们把register的结果数组从观测列表中移出，然后观察每一条指令的opcode及相关的状态的输出，看每一步的指令解析是否正确：

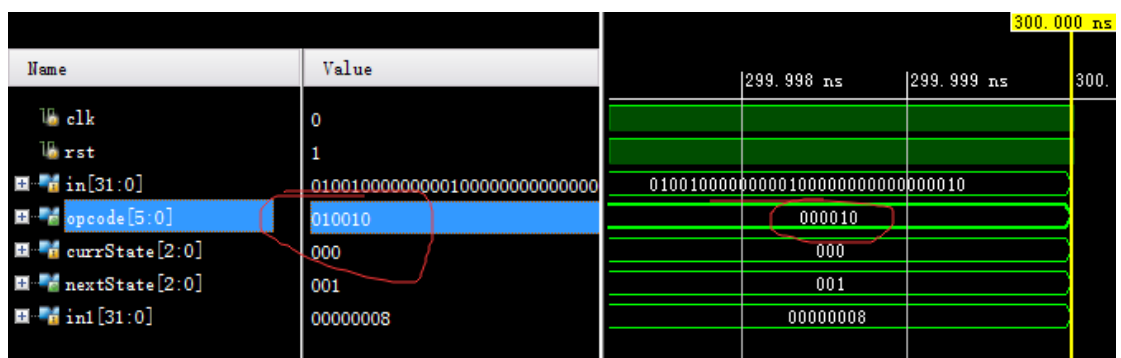
- a) IF->ID->EXE1->WB1->IF指令（以addi为例）：

Name	Value	59.998 ns	59.999 ns	60.000 ns
clk	0			
rst	1			
in[31:0]	00001000000000000100000000000000	000010000000000001000000000000110		
opcode[5:0]	000010	XXXXXX		
currState[2:0]	000	000		
nextState[2:0]	001	001		
in1[31:0]	00000004	00000004		

（IF阶段还没有解码，opcode还处于不定状态，到了ID阶段才会进行解码）



新一轮的IF:



(注意此处opcode在300ns的时刻已经变化，但是在波形图里面因为刚好走到变化的时刻所以变化没有在波形图上显示出来，但是中间的value栏可以看到opcode已经发生了改变，到下个阶段也就是ID阶段的时候波形图上的opcode才会跟着改变。这种情况在下面的截图中不再重复说明)

(上图中的in1变量为pc4的指令地址，下面将改回pc4为展示变量)

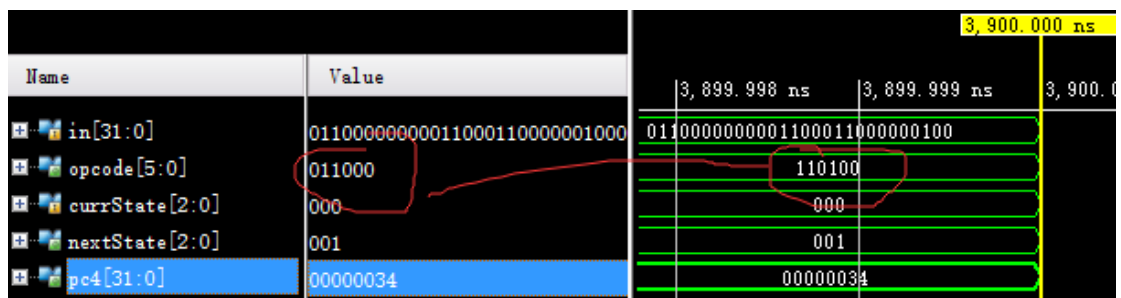


b) IF→ID→EXE2→IF指令（在此cpu中只有beq指令）

第一次beq（true）



新一轮的IF:



（判断正确， $\$12=2 == \$2$ ，指令执行成功回退）

第二次beq (false)

Name	Value			4, 140.000 ns
in[31:0]	110100011000000101111111111111	4, 139.998 ns	4, 139.999 ns	4, 140
opcode[5:0]	110100	11010001100000010111111111	11010001100000010111111111	
currState[2:0]	000	011000	011000	
nextState[2:0]	001	000	000	
pc4[31:0]	000000038	001	001	
		000000038	000000038	

Name	Value			4, 200.000 ns
in[31:0]	110100011000000101111111111111	4, 199.998 ns	4, 199.999 ns	4, 200
opcode[5:0]	110100	11010001100000010111111111	11010001100000010111111111	
currState[2:0]	001	110100	110100	
nextState[2:0]	101	001	001	
pc4[31:0]	000000038	101	101	
		000000038	000000038	

Name	Value			4, 260.000 ns
in[31:0]	110100011000000101111111111111	4, 259.998 ns	4, 259.999 ns	4, 260
opcode[5:0]	110100	11010001100000010111111111	11010001100000010111111111	
currState[2:0]	101	110100	110100	
nextState[2:0]	000	101	101	
pc4[31:0]	000000038	000	000	
		000000038	000000038	

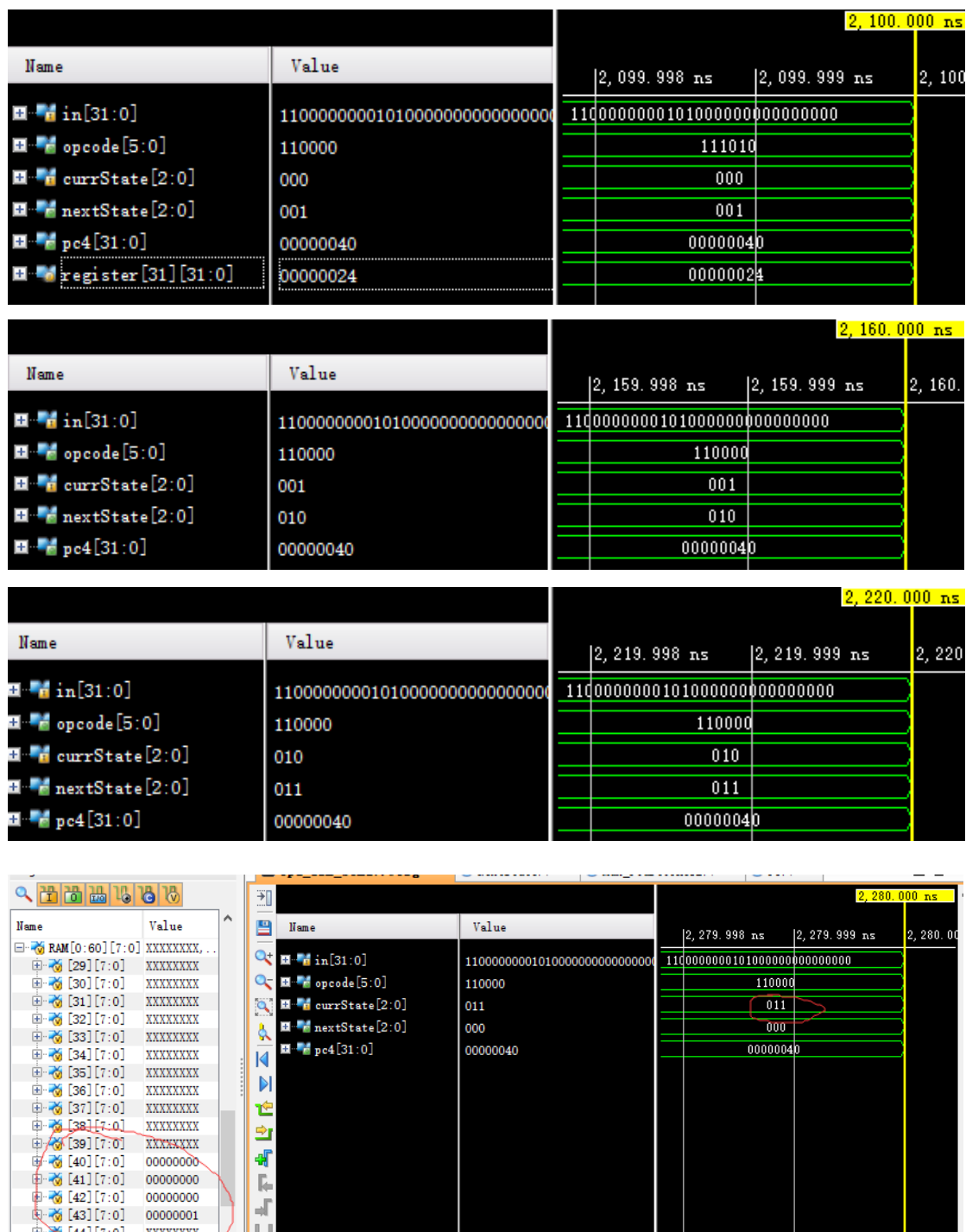
新一轮的IF:

Name	Value			4, 320.000 ns
in[31:0]	11100000000000000000000000000010	4, 319.998 ns	4, 319.999 ns	4, 320
opcode[5:0]	111000	11100000000000000000000000000010	11100000000000000000000000000010	
currState[2:0]	000	111000	111000	
nextState[2:0]	001	000	000	
pc4[31:0]	00000003e	001	001	
		00000003e	00000003e	

可以看到走到了opcode为111000的语句，就是判断为false直接往下走的那条语句。

c) IF→ID→EXE3→MEM→WB2→IF指令（以1w为例）

SW:



（写回状态的时候，datamem里面的正确位置存储了正确的值）

lw:

		2,340.000 ns		
Name	Value	2,339.998 ns	2,339.999 ns	2,340.000 ns
in[31:0]	11000100001010110000000000000000	11000100001010110000000000000000		
opcode[5:0]	110001	110001		
currState[2:0]	000	000		
nextState[2:0]	001	001		
pc4[31:0]	00000044	00000044		

		2,400.000 ns		
Name	Value	2,399.998 ns	2,399.999 ns	2,400.000 ns
in[31:0]	11000100001010110000000000000000	11000100001010110000000000000000		
opcode[5:0]	110001	110001		
currState[2:0]	001	001		
nextState[2:0]	010	010		
pc4[31:0]	00000044	00000044		

		2,520.000 ns		
Name	Value	2,519.998 ns	2,519.999 ns	2,520.000 ns
in[31:0]	11000100001010110000000000000000	11000100001010110000000000000000		
opcode[5:0]	110001	110001		
currState[2:0]	011	011		
nextState[2:0]	100	100		
pc4[31:0]	00000044	00000044		

		2,460.000 ns		
Name	Value	2,459.998 ns	2,459.999 ns	2,460.000 ns
in[31:0]	11000100001010110000000000000000	11000100001010110000000000000000		
opcode[5:0]	110001	110001		
currState[2:0]	010	010		
nextState[2:0]	011	011		
pc4[31:0]	00000044	00000044		

		2,520.000 ns		
Name	Value	2,519.998 ns	2,519.999 ns	2,520.000 ns
in[31:0]	11000100001010110000000000000000	11000100001010110000000000000000		
opcode[5:0]	110001	110001		
currState[2:0]	011	011		
nextState[2:0]	100	100		
pc4[31:0]	00000044	00000044		

Name	Value	2,579.998 ns	2,579.999 ns	2,580.000 ns
in[31:0]	11000100001010110000000000000000	11000100001010110000000000000000		
opcode[5:0]	110001	110001		
currState[2:0]	100	100		
nextState[2:0]	000	000		
pc4[31:0]	00000044	00000044		

新一轮IF（这个时候在上一轮时钟周期的下降沿成功写入\$11寄存器，lw指令运行正确）：

Name	Value	2,639.998 ns	2,639.999 ns	2,640.000 ns
in[31:0]	11100111111000000000000000000000	11100111111000000000000000000000		
opcode[5:0]	111001	111001		
currState[2:0]	000	000		
nextState[2:0]	001	001		
pc4[31:0]	00000048	00000048		
register[11][31:0]	00000001	00000001		

d) IF→ID→IF指令（同时展示jal, jr, j指令）

jal:

Name	Value	1,979.998 ns	1,979.999 ns	1,980.000 ns
in[31:0]	11101000000000000000000000000000	11101000000000000000000000000000		
opcode[5:0]	111010	100111		
currState[2:0]	000	000		
nextState[2:0]	001	001		
pc4[31:0]	00000024	00000024		

Name	Value	2,039.998 ns	2,039.999 ns	2,040.000 ns
in[31:0]	11101000000000000000000000000000	11101000000000000000000000000000		
opcode[5:0]	111010	111010		
currState[2:0]	001	001		
nextState[2:0]	000	000		
pc4[31:0]	00000024	00000024		

新一轮IF:

Name	Value			
in[31:0]	11000000001010000000000000000000	2,099.998 ns	2,099.999 ns	2,100.000 ns
opcode[5:0]	110000	11000000001010000000000000000000	11000000001010000000000000000000	
currState[2:0]	000	111010	000	
nextState[2:0]	001	000	001	
pc4[31:0]	00000040	00000040	00000040	
register[31][31:0]	00000024	00000024	00000024	

(观察, register的第31个寄存器已经存好了上一轮的pc4也就是0x00000024, 指令执行正常)

jr:

Name	Value			
in[31:0]	11100111111000000000000000000000	2,639.998 ns	2,639.999 ns	2,640.000 ns
opcode[5:0]	111001	11100111111000000000000000000000	11100111111000000000000000000000	
currState[2:0]	000	110001	000	
nextState[2:0]	001	000	001	
pc4[31:0]	00000048	00000048	00000048	

Name	Value			
in[31:0]	11100111111000000000000000000000	2,699.998 ns	2,699.999 ns	2,700.000 ns
opcode[5:0]	111001	11100111111000000000000000000000	11100111111000000000000000000000	
currState[2:0]	001	111001	001	
nextState[2:0]	000	000	000	
pc4[31:0]	00000048	00000048	00000048	

新一轮IF (观察此时的pc4已经发生改变, 成功跳回了0x00000024的地址):

Name	Value			
in[31:0]	10011001011000000100100000000000	2,759.998 ns	2,759.999 ns	2,760.000 ns
opcode[5:0]	100110	10011001011000000100100000000000	10011001011000000100100000000000	
currState[2:0]	000	111001	000	
nextState[2:0]	001	000	001	
pc4[31:0]	00000028	00000028	00000028	

j:

Name	Value			
in[31:0]	11100000000000000000000000000010	4,319.998 ns	4,319.999 ns	4,320.000 ns
opcode[5:0]	111000	11100000000000000000000000000001	11100000000000000000000000000001	
currState[2:0]	000	110100	000	
nextState[2:0]	001	001	001	
pc4[31:0]	0000003e	0000003e	0000003e	

Name		Value		
		4,380.000 ns		
		4,379.998 ns	4,379.999 ns	4,380.000 ns
in[31:0]	11100000000000000000000000000000	11100000000000000000000000000000	11100000000000000000000000000000	11100000000000000000000000000000
opcode[5:0]	111000	111000	111000	111000
currState[2:0]	001	001	001	001
nextState[2:0]	000	000	000	000
pc4[31:0]	0000003c	0000003c	0000003c	0000003c

新一轮IF:

Name		Value		
		4,440.000 ns		
		4,439.998 ns	4,439.999 ns	4,440.000 ns
in[31:0]	11111100000000000000000000000000	11111100000000000000000000000000	11111100000000000000000000000000	11111100000000000000000000000000
opcode[5:0]	111111	111111	111111	111111
currState[2:0]	000	000	000	000
nextState[2:0]	001	001	001	001
pc4[31:0]	0000004c	0000004c	0000004c	0000004c

观察pc4可知地址成功跳转。

halt (PCWre为0, 在循环执行halt, 停止运行):

Name		Value		
		4,939.998 ns		
		4,939.998 ns	4,939.999 ns	4,940.000 ns
in[31:0]	11111100000000000000000000000000	11111100000000000000000000000000	11111100000000000000000000000000	11111100000000000000000000000000
opcode[5:0]	111111	111111	111111	111111
currState[2:0]	000	000	000	000
nextState[2:0]	001	001	001	001
pc4[31:0]	0000004c	0000004c	0000004c	0000004c
PCWre	0	0	0	0

Name		Value		
		5,000.000 ns		
		4,999.998 ns	4,999.999 ns	5,000.000 ns
in[31:0]	11111100000000000000000000000000	11111100000000000000000000000000	11111100000000000000000000000000	11111100000000000000000000000000
opcode[5:0]	111111	111111	111111	111111
currState[2:0]	001	001	001	001
nextState[2:0]	000	000	000	000
pc4[31:0]	0000004c	0000004c	0000004c	0000004c
PCWre	0	0	0	0

整个仿真模拟结束。

## 六、实验心得

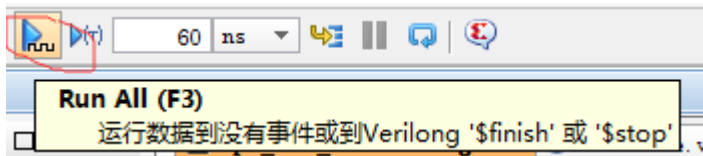
1. 总体了解了多周期CPU的实现流程，弄清楚了单周期CPU，多周期CPU，流水线CPU的辨析：**多周期CPU与单周期CPU的区别在于**，多周期CPU在一个时间周期内只执行一条指令中的一个阶段，而单周期CPU是在一个时间周期内执行一条完整的指令。由于并不是所有的指令都需要执行最多的5个步骤，所以多周期CPU的执行时间会比单周期CPU的略快，思考的重点就是如何实现数据内容的缓存，如何在时钟周期更替的时候使得控制信号量和相关寄存器里面的值符合预期的要求；**多周期CPU与流水线CPU的区别在于**，多周期CPU指令的执行也还是顺序执行的，而流水线CPU的执行时属于并行执行的，明白了这一点后，就可以明白在考虑多周期CPU运行过程的时候，集中点都是在一条特定的指令上，不用考虑冒险。

2. 在控制信号的产生方面，想了有三种表示的方法，其中被我淘汰的两种方法分别是：

1) **以单条指令为纵轴，以信号量为横轴，归纳出所有情况的信号量**：一开始的错误是沿用了单周期CPU的思路，想通过fingerprint来对所有信号进行持续赋值，但是实际上发现没有考虑到执行的阶段，所以错误。改正的方法是，对于所有的指令，再考虑各个阶段的所有信号量，这样下来工作量极大，极易出错且调试成本巨大，故放弃。

2) **以执行阶段为纵轴，以信号量为横轴，归纳出所有情况的信号量**：此方法成本开销较方法1少，但是还是容易混淆，后来想到了以信号量为纵轴中心的思考方法后，此方法也被舍弃。

3. 在调试的过程中设置的断点如果没有办法触发，那么使用



这个按钮的时候将会陷入死循环里面，如果点击取消需要等待很久，开了任务管理器杀掉进程后，编译运行文件会出现错误无法再次模拟，解决方法是把目录中的'multicycle\_cpu.sim'目录下的文件删除，重新模拟生成新的文件，即可解决问题。

### 4. 调试问题

将初步的模块都写出来可以跑起来之后，就需要一步步执行进行调试和修改自己第一次写的代码的疏漏的地方了。下面主要说一下自己调试过程中遇到的主要几个问题，相关思考还有解决方法。



### 1) 指令文件读取改用绝对路径为用相对路径

在做单周期cpu的时候，存放指令的文本文件的读取是使用绝对路径的，这样的一个是代码的扩展性很差，每次更换电脑或者移动项目后都需要重新配置路径，使用相对路径才是比较好的做法。在解决了心得3出现的问题后，我意识到了代码的可执行文件应该是在项目根目录下的'\multicycle\_cpu.sim\sim\_1\behav\'文件夹内，这样的话，我的指令文件放在项目的根目录下，于是尝试使用相对路径为' ../../../../ins.txt'，实际运行后成功。

### 2) state的转移正常，但是opcode没有改变

思考，opcode没有改变，说明inst的读入出现了问题，往上查找，检查IR模块正常，但是发现IR模块的输入也没有改变，然后检查InstructionMemory模块后，发现问题：控制信号Insmem的定义单周期CPU和多周期CPU的定义不同，单周期CPU的时候是0的时候读取指令，本次实验则是在Insmem为1的时候读取指令，一开始以为模块可以复用所以就没有检查到。解决方法时修改InstructionMemory模块中的一个判断条件即可。

### 3) IR没有被读取的时候，里面的值还是在不断地被写。

思考，重新理解了多周期CPU后，明白并不是一个时钟周期就读一个指令，所以在不是取指令（IF）阶段的时候，curPC应该不能够被改变。查看pc模块代码

```
module PC( clk, rst, PCWre, nextPC, shiftline_num, curPC, pc4, shiftNextPC);
    input clk, rst, PCWre;
    input [31:0] nextPC;
    input [31:0] shiftline_num;

    output reg [31:0] curPC;
    output reg [31:0] pc4;
    output reg [31:0] shiftNextPC;

    always@( posedge clk ) begin
        if( rst == 0 ) curPC = 0;
        else begin
            if (PCWre) curPC = nextPC;
        end
    end

    always@(curPC) begin
        pc4 = curPC + 4;
        shiftNextPC = pc4 + {shiftline_num[29:0], 2'b00};
    end
end
```

发现因为每次都在上升沿的时候尝试改变curPC，而信号量PCWre值的生成是在IF阶段中进行，所以在ID阶段的开始，curPC就总是会再加4，导致CPU内部的运

行情况错误。解决方法是把检测赋值的条件从时钟周期上升沿调整为

```
always@( PCWre or negedge rst ) begin
    if( rst == 0 ) curPC = 0;
    else begin
        if (PCWre) curPC = nextPC;
    end
end
```

这样一改果然pc4的值就正常了。

但是这样子在后期模拟的时候会出现另外一个问题，就是beq的跳转问题。这个问题在后面陈述。

#### 4) beq判断成功后跳转地址错误

思考，通过设置断点，在beq将要赋值shiftNextPC的时候查看变量内容，发现beq的转移的偏移量并不是想要的立即数，而是上一条代码的无效区域，思考后发现，一开始代码的实现是

```
always@(curPC) begin
    pc4 = curPC + 4;
    shiftNextPC = pc4 + {shiftline_num[29:0], 2'b00};
end
```

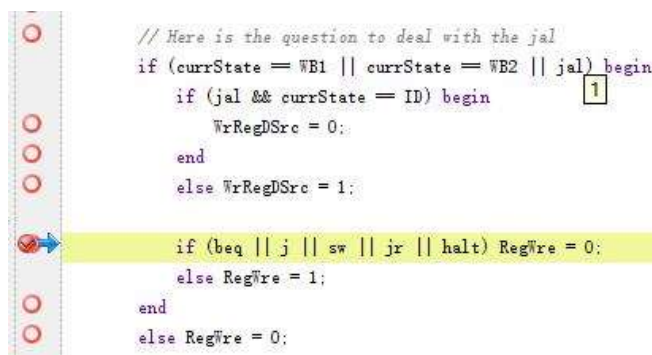
IF阶段取了新的指令后curPC就会改变，于是pc4和shiftNextPC就会刷新，但是这个时候因为还没有进入到ID阶段，beq指令还未被解码，导致shiftline\_num是上一条指令中的无用的区域，所以出错。通过将shiftNextPC分开来更新，即可以解决问题。

```
always@(curPC) begin
    pc4 = curPC + 4;
end

always @ ( shiftline_num ) begin
    shiftNextPC = pc4 + {shiftline_num[29:0], 2'b00};
end
```

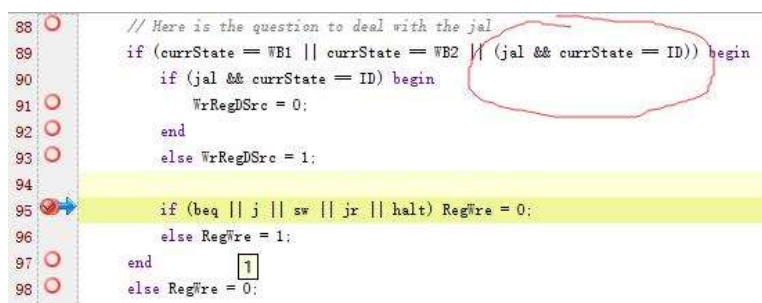
## 5) jal的地址记录\$31只成功写入一次后被其他值覆盖

思考，通过断点调试，发现在jal后的指令后寄存器的写使能端还是处于可写状态，为何会如此呢？看一下旧的代码：



```
// Here is the question to deal with the jal
if (currState == WB1 || currState == WB2 || jal) begin
  if (jal && currState == ID) begin 1
    WrRegDSrc = 0;
  end
  else WrRegDSrc = 1;
end
if (beq || j || sw || jr || halt) RegWre = 0;
else RegWre = 1;
end
else RegWre = 0;
```

在jal下一条指令的时候，jal的fingerprint还是为1，所以进入了if语句中，使能端还是为1，在IF这个时钟周期的下降沿还是可以写，所以就写到了下一条语句的无用位。主要原因是因为，IF阶段还未进入ID阶段，opcode未被刷新导致fingerprint没有更新从而造成了错误。重新规范了if的条件，加上jal的同时是处于ID阶段才能进入if语句，解决了问题。



```
88 // Here is the question to deal with the jal
89 if (currState == WB1 || currState == WB2 || (jal && currState == ID)) begin
90   if (jal && currState == ID) begin
91     WrRegDSrc = 0;
92   end
93   else WrRegDSrc = 1;
94 end
95 if (beq || j || sw || jr || halt) RegWre = 0;
96 else RegWre = 1;
97 end 1
98 else RegWre = 0;
```

5. 关于写回的问题，采用的是时钟下降沿尝试写回，总体来说没有加入太多的逻辑判断，那么为了保证结果的正确性，对于信号RegWre和WrRegDSrc的控制就需要做严格保证，只有在该写的情况下才会产生可写的使能端信号，于是就是如心得4（5）的问题就可以完善解决。

6. 再次体会到了模块化设计思想的重要性，分离了逻辑，整个实验的难度在于OutputFun中信号的产生，但是实际上单周期CPU实验里面的模块可直接重用的模块非常少，在调试的时候基本上每个模块都要重新修改一下，极大地提升了我对多周期CPU的理解。而且实验报告的书写也十分重要，当我认真地把指令运行的流程一步一步地截图说明的时候，为了合理解释我理解了很多在写实验报告前还很含糊的地方。