《计算机组成原理与接口技术实验》

实验报告

学 院 名 称：数据科学与计算机学院

学 生 姓 名：黄羽盼

学　　号：14331106

专业（班级）：14 软件工程三（6）班

合 作 者：蒋子涵

时　　间：2016 年 5 月 15 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。**）

==>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)	**immediate**(16 位)

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

==>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 & rt

（6）ori rt, rs, **immediate**

010010	rs(5 位)	rt(5 位)	**immediate**

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

==>移位指令

（7）sll rd, rs,sa

011000	rs(5 位)	未用	rd(5 位)	sa	reserved

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

==>传送指令

（8）move　rd, rs

100000	rs(5 位)	00000	rd(5 位)	reserved

功能：rd<－rs + $0

==>比较指令

（9）　slt　rd, rs, rt

100111	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：如果（rs<rt），则 rd=1;　否则　rd=0

==>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

（11）lw　rt, **immediate**(rs)

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

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

==>分支指令

（12）beq　rs,rt, **immediate** (说明：**immediate 是从 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

==>跳转指令

（13）j　addr

111000	addr[27..2]

功能：pc <{pc[31..28],addr[27..2],0,0}，转移

（14）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，转移

==>调用子程序指令

（15）jal　addr

111010	addr[27..2]

功能：调用子程序，pc <－ {pc[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。

==>停机指令

（16）halt (停机指令)

111111	00000000000000000000000000(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：

⑴ 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。

⑵ 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。

⑶ 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。

⑷ 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。

⑸ 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。

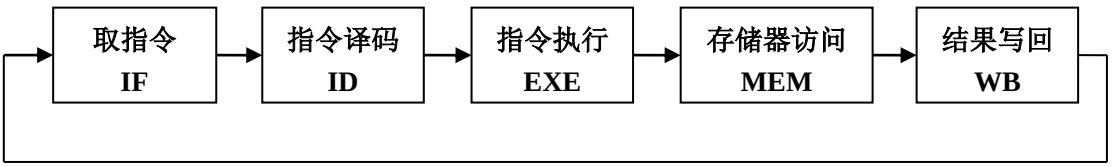
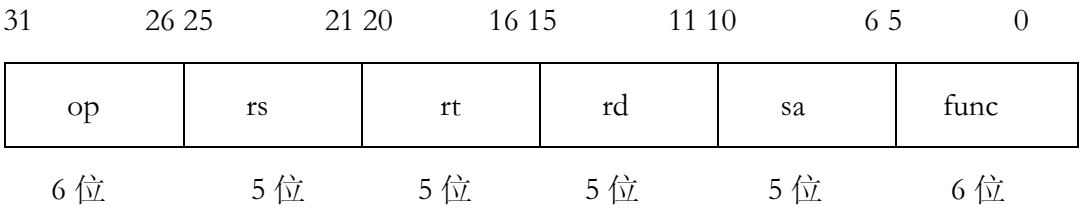


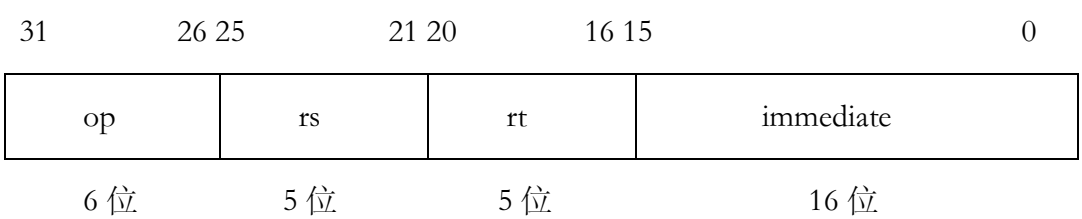
图 1 多周期 CPU 指令处理过程

MIPS32 的指令的三种格式：

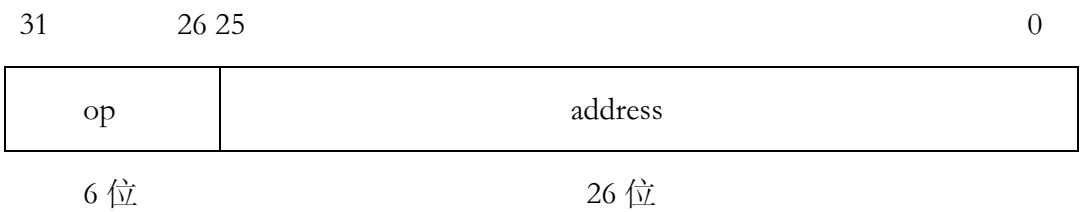
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

rs：为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：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：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；

func：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；

immediate：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载(Laod)/数据保存(Store)指令的数据地址字节偏移量和分支指令中相对程序计数器(PC)的有符号偏移量；

address：为地址。

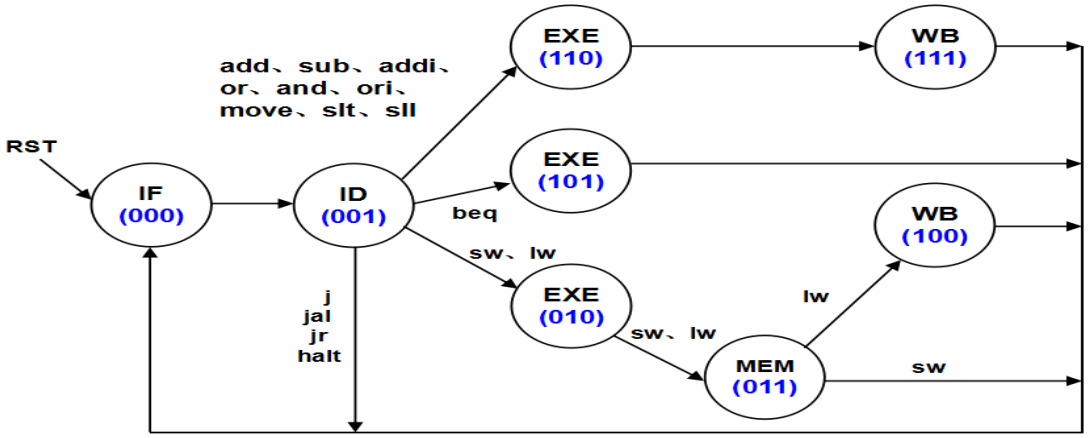


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 IF 状态转移到 ID 和 EXE 状态就是无条件的；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。

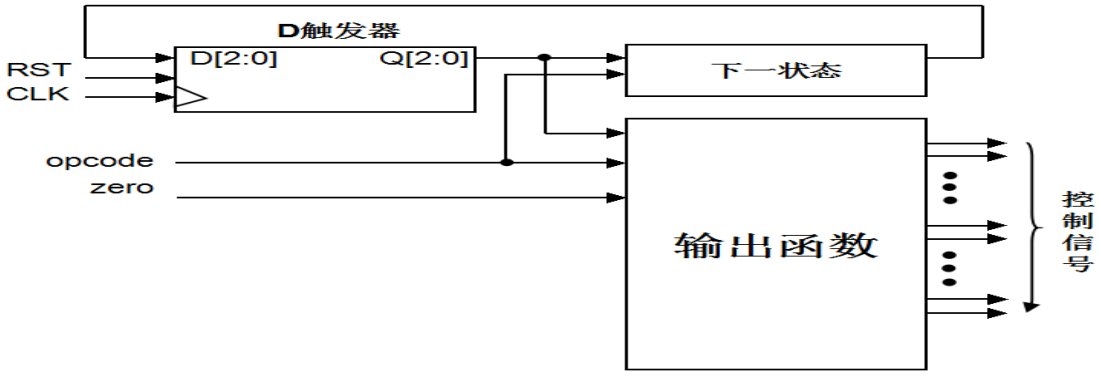


图 3 多周期 CPU 控制部件的原理结构图

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 标志等。

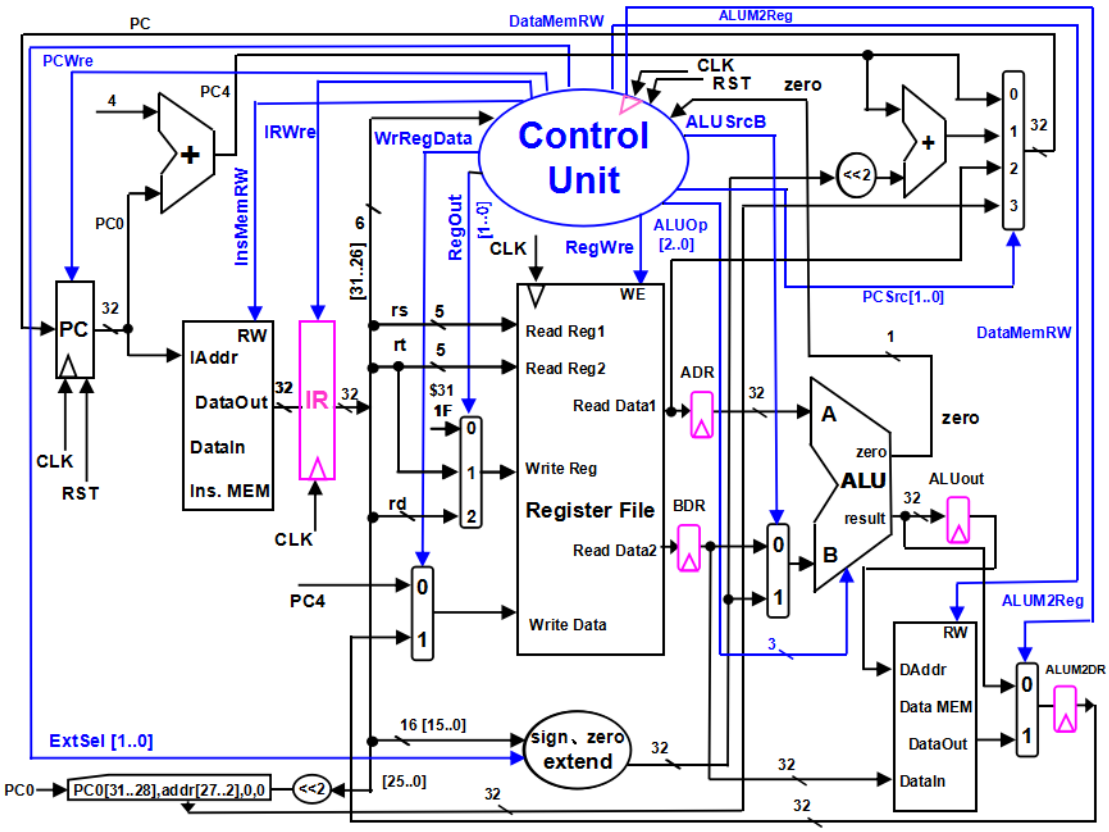


图 4 多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在单周期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/写信号控制（1-写，0-读。当然，也可以由时钟信号控制，但必须在图上画出来）。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，还有 pc 增加写使能控制信号 pcWre，也是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUout、ALUM2DR 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，大延时变为多个分段小延时。

表 1 控制信号作用

控制信号名	状态“0”	状态“1”
PCWre	PC 不更改，**相关指令**：halt，**另外，除‘000’状态之外，其余状态也不能改变 PC 的值。**	PC 更改，**相关指令**：**除指令 halt 外，另外，只有在‘000’状态时，才能改 PC 的值。**
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、addi、or、and、ori、move、beq、slt	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、lw、sw、sll
ALUM2Reg	来自 ALU 运算结果的输出，相关指令：add、sub、addi、or、and、ori、slt、sll、move	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、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addi、or、and、ori、move、slt、sll、lw、jal
WrRegData	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、sll、move、lw
InsMemRW	读指令存储器(Ins. Data)，**初始化为 0**	写指令存储器
DataMemRW	读数据存储器（Data MEM），相关指令：lw	写数据存储器，相关指令：sw
IRWre	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。
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、move、slt、sll、sw、lw、beq(zero=0) 01：pc<－pc+4+(sign-extend)**immediate**，同时 zero=1，相关指令：beq 10：pc<－rs，相关指令：jr 11：pc<－pc(31..28],addr,0,0 ，相关指令：j、jal	
RegOut[1..0]	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、ori、lw 10：rd 字段，相关指令：add、sub、or、and、move、slt、sll 11：未用	
ExtSel[1..0]	00：(zero-extend)sa，相关指令：sll 01：(zero-extend)**immediate**，相关指令：ori 10：(sign-extend)**immediate**，相关指令：addi、lw、sw、beq 11：未用	

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 1 写，为 0 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

RW，数据存储器读写控制信号，为 1 写，为 0 读

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：

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 （A<B） Y = 1; else Y = 0;	比较 A 与 B
011	Y = A>>B	A 右移 B 位
100	Y = A<<B	A 左移 B 位
101	Y = A ∨ B	或
110	Y = A ∧ B	与
111	Y = A ⊕ B	异或

四. 实验器材

电脑一台、Xilinx ISE 软件一套。

五. 实验分析与设计

1. 实验步骤

（1） 阅读课件及提供的资料，了解本次实验的内容。

（2） 对实验进行分析（见2. 分析）

（3） 对实验进行设计（见3. 设计）

（4） 编写各子模块代码

（5） 编写测试代码（见4. 测试单周期CPU）

（6） debug（见心得和体会部分的 1. 实验过程中遇到的问题及解决的办法）

（7） 反思总结写报告

2. 分析

中央处理器（CPU）是计算机取指令和执行指令的部件，它由算术逻辑单元（ALU）、寄存器和控制器组成,是计算机系统的核心部件。

CPU设计的第一步应当根据指令系统来建立数据路径，再定义各个部件的控制信号，确定时钟周期，完成控制器的设计。进而可以进行数字设计、电路设计，最后完成物理实现。本实验重点是多周期数据路径的建立和组合逻辑实现的控制器。

实验原理中已经详细阐述了本实验的主要原理依据，用我自己的话来分析，这个实验需要设计一个硬件电路,实现从存储器中读取指定的指令并解析指令从而调动各个模块执行相应的操作。与单周期CPU不同的是，这些指令的执行不是在一个时钟周期内完成，而是将整个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。也就是说要根据指令的具体内容，分配不同个数的时钟周期，从而达到“因材施教”、“物尽其用”的效果（执行不同指令需要的时间不同，有针对性的分配时间可以提高执行效率），这就是多周期CPU的含义。

要实现这个设计，需要知道CPU是怎样处理指令的。和单周期CPU一样，一条指令的执行最长需要五个（小）时钟周期：首先是取指令（IF:PC,存储器），然后需要对指令译码(ID)，即剖析对应二进制数代表的操作和含义，此时需要产生相应的操作控制信号来告诉各个模块要做什么（控制单元），然后执行(EXE)，还可能涉及到对存储器的访问(MEM)和结果写回(WB)。

多周期CPU设计和单周期最大的不同在于对指令译码阶段的分析和操作。由于多周期要对不同的指令分配不同的时钟周期,需要在译码阶段分析具体情况并给出相应操作。首先根据不同的指令画出多周期CPU状态转移图，其中EXE和WB阶段分别有3种和两种情况，加上后缀stages以示区别。

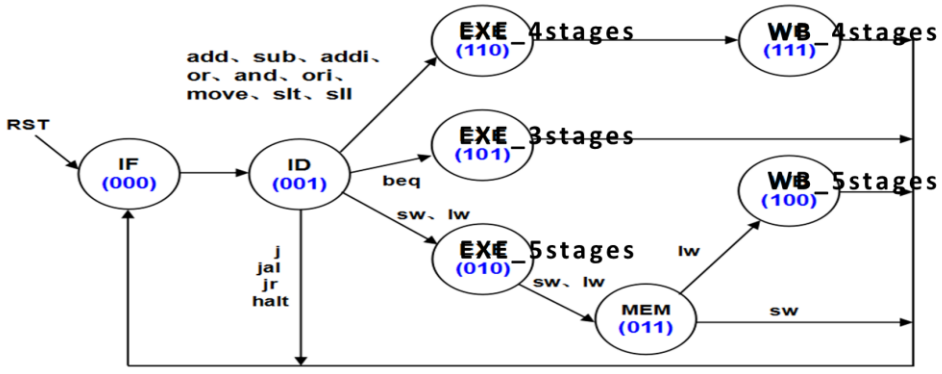


图 5 多周期 CPU 状态转移图（2）

由特定指令（指令操作码）所处的特定阶段（当前状态）可以唯一确定它的下一阶段（下个状态），可以用一个NextState模块来实现这个逻辑功能。另外和单周期CPU不同，控制单元增加了两个输入信号：时钟信号和重置信号，时钟信号用于控制在特定指令进入特定的下一个小阶段即控制状态的转换，重置信号用于初始化状态“000”即取指阶段。因此控制单元由D触发器、下一状态转换、控制信号发出这三个小模块组成，如下图。

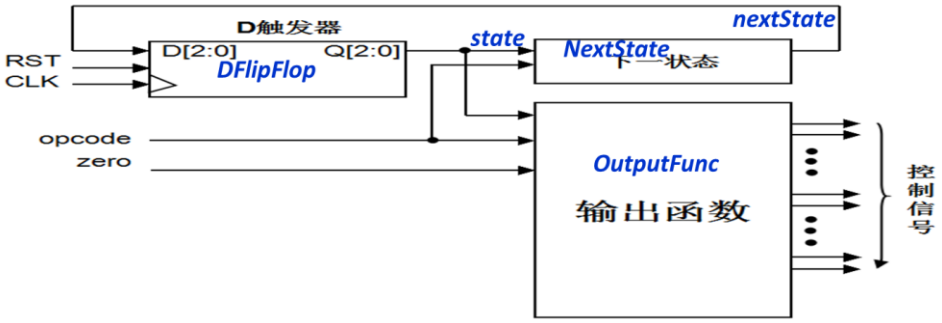


图 6 多周期 CPU 控制部件的原理结构图（2）

整个过程中需要对数据进行操作，而数据在寄存器或存储器中，不同的指令对数据的处理不一样，有的要读有的要写。因此需要分析不同类型指令的特点和异同导致的数据传输路径和控制信号的不同，可以先画出它们各自的数据通路图，再结合起来。数据通路指CPU中处理数据和地址的流动路径。各子模块的数据通路已在单周期CPU设计中进行了详尽的分析，不再赘述。

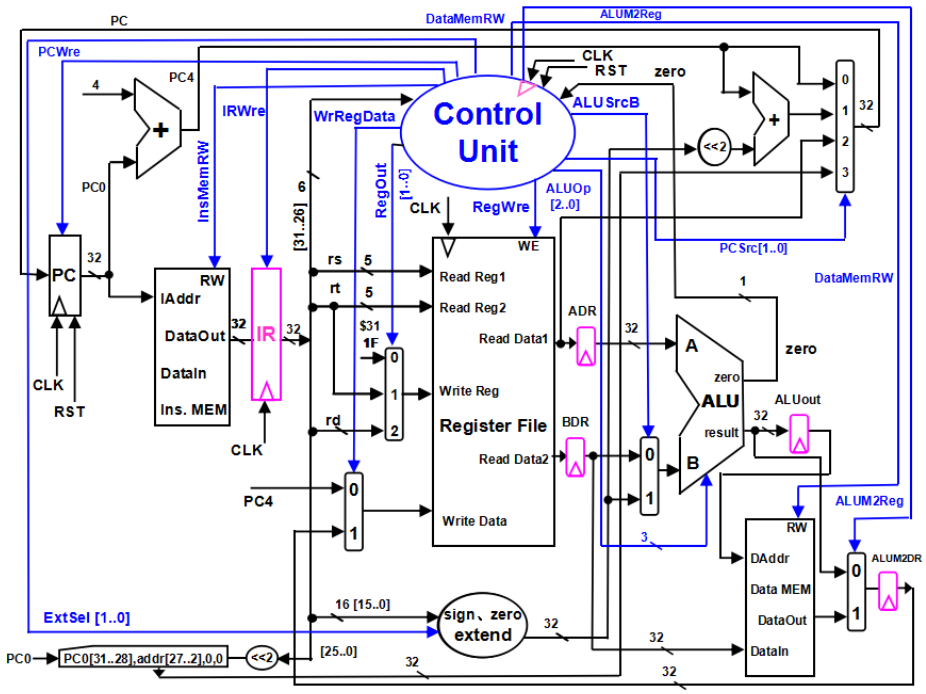


图 7 多周期 CPU 数据通路和控制线路图（2）

不同的是，单周期数据通路模型可以概括为PC->组合逻辑->寄存器文件，只有一层寄存器与组合逻辑就实现了五个阶段的功能，由于关键路径的存在，每条指令延迟均相同，无法利用不同指令具有不同执行需求的潜在特性。

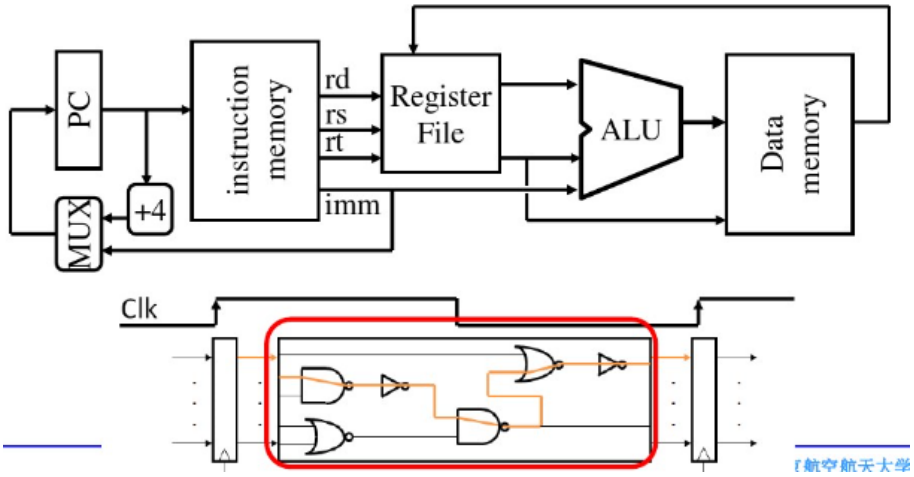


图 8 单周期数据通路模型

单周期数据通路可以将PC取指和register file看做同一阶段。多周期数据通路要在组合逻辑中插入寄存器，以切分数据通路，这样一来，单一组合逻辑被切分为若干小组和逻辑，单一大延迟变为多个分段小延迟，从而实现不同指令执行占用不同的功能单元，不必五个环节都走完。分割成五个阶段有助于性能的提高。加入的寄存器用于存上一阶段的输出值，作为下一阶段的输入值，放入其中进行缓存可以使得在控制单元发出控制信号后才进行数据的获取，保证了五个阶段互不干

扰，都受控制单元控制。

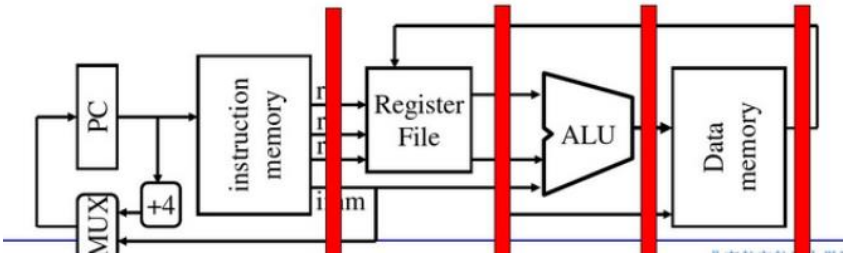
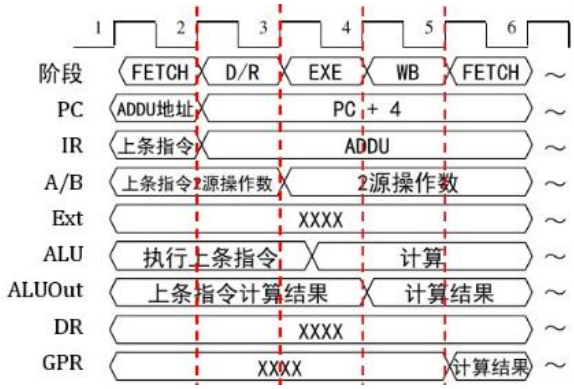


图 9 多周期数据通路构思



	FETCH	DCD/RF	EXE	WB	FETCH
PC	ADDU地址	PC+4	PC+4	PC+4	PC+4
IM	输出ADDU	指令[PC+4]	指令[PC+4]	指令[PC+4]	指令[PC+4]
IR	旧指令	ADDU	ADDU	ADDU	ADDU
A/B	旧操作数	旧操作数	新操作数	新操作数	新操作数
Ext	无效扩展	无效扩展	无效扩展	无效扩展	无效扩展
ALU	执行前指令	执行前指令	计算	计算	计算
ALUOut	前结果	前结果	前结果	计算结果	计算结果
DM	无效数据	无效数据	无效数据	无效数据	无效数据
DR	无效数据	无效数据	无效数据	无效数据	无效数据
GPR	旧值输出	新值输出	新值输出	新值输出	输出/更新

图 10 多周期 CPU 指令执行过程

具体来说，多周期CPU数据通路和控制线路图相比于单周期的增加了IR指令寄存器用于存放正在执行的指令代码，目的是使指令代码保持稳定。因为多周期时PC不是每个时钟周期都读指令，不同指令会在经过不同个数的时钟周期后进入读指令的下一阶段（在图中表现为有多个箭头指向IF），PC需要增加写使能控制信号确保PC在正确的时机更改。ADR、BDR、ALUout、ALUM2DR四个寄存器对应上图右边的红色部分，用于上面提到的切分数据通路作用，类似于“缓冲带”。它们不需要写使能信号，这是因为它们的前一阶段是固定的（在图中表现为只有一个箭头指向这个阶段），不需要分情况来读取数据。

另外，由于增加了j,jal,jr跳转指令和sll,slt指令的设计，控制信号需要进行相应的更改。具体来说，PCSrc控制信号变为两位，增加了jr指令（10）情况下pc取rs即第31号寄存器存储的地址和j,jal指令（11）情况下直接跳转。

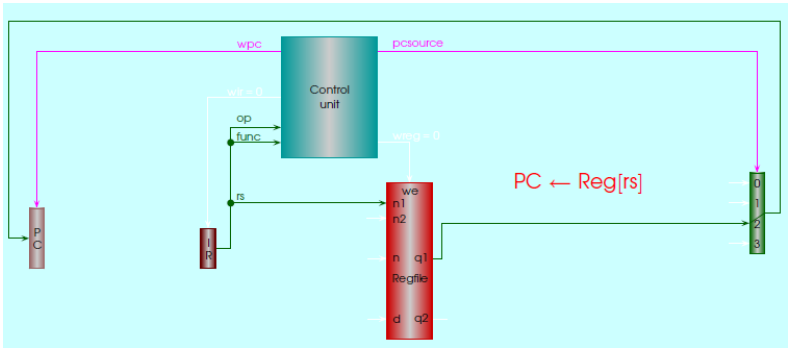


图 11 ID 状态-jr 指令

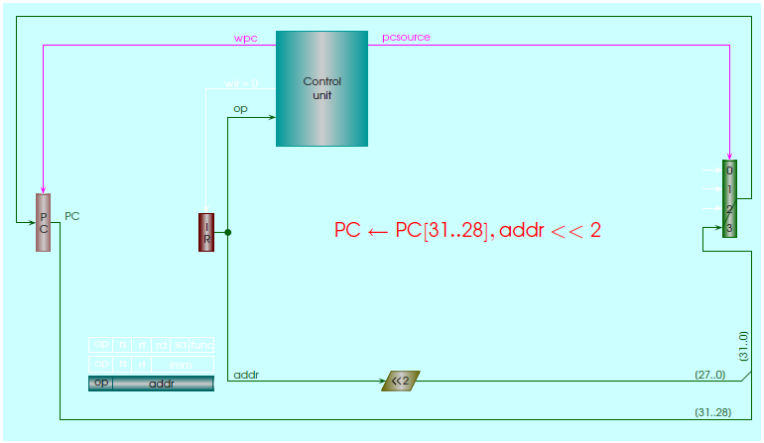


图 12 ID 状态-j 指令

而写寄存器组寄存器的地址来自RegOut，也变成了两位，增加了一个状态（00），用于jal指令下保存返回地址，即$31<-pc+4.

控制位扩展的ExtSel信号也需要增加一个状态（00），用于sll指令零扩展sa从5位到32位。

具体参见表1（控制信号作用）。由此表可以写出OutputFunc模块的逻辑组合。而由这些控制信号控制的其他模块也就很容易分析进行相应的更改。例如由PCSrc控制的PC模块的输入由1个增加成了3个，分别是readData1,pcIn,bits_26，这都跟前面所述的PCSrc在不同状态下对应指令不同阶段操作有关。

3. 设计

用模块化的思想方法设计。

查阅相关资料（Verilog模块概念和实例化[1]）可知，模块（module）是verilog最基本的概念，是v设计中的基本单元，每个v设计的系统中都由若干module组成。具体来说，模块有以下性质：

1） 模块在语言形式上是以关键词module开始，以关键词endmodule结束的一段程

[1] http://jason0214.lofter.com/post/30cbe4_12a8f72

序。

2） 模块的实际意义是代表硬件电路上的逻辑实体。

3） 每个模块都实现特定的功能。

4） 模块的描述方式有行为建模和结构建模之分。

5） 模块之间是并行运行的。

6） 模块是分层的，高层模块通过调用、连接低层模块的实例来实现复杂的功能。

7） 各模块连接完成整个系统需要一个顶层模块（top-module）。

由于单周期CPU不是一个非常简单的系统，而无论多么复杂的系统，总能划分成多个功能模块。因此我按照下面四个步骤进行单周期CPU的设计：

1） 把系统划分成子模块；

由于单周期CPU的设计涉及到了很多小部件，如加法器、二选一数据选择器、有无符号数扩展等，为了使设计起来简洁明了、打代码时思路简单，我选择将系统尽可能分解成小模块。比如单独定义二选一数据选择器的模块而不是将它们直接通过分析逻辑关系和旁边的部件组合起来。但是由于PC的操作是逻辑清晰而连贯的，模块PC可以整合PC+4及遇到跳转指令时pc←pc + 4 + (sign-extend)immediate <<2的功能，而不再细分左移、加法器等模块。具体分出的模块及各模块的名称见下面两个图。

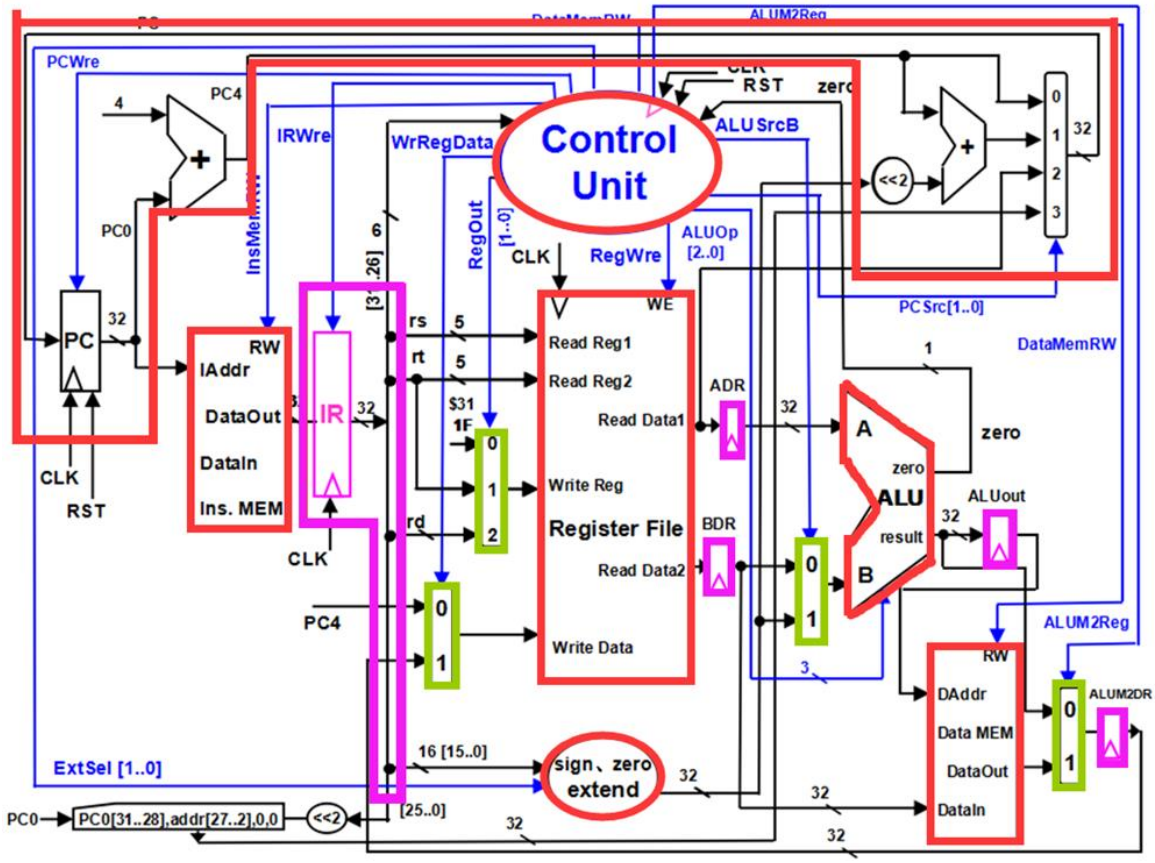


图 13 数据通路模块划分

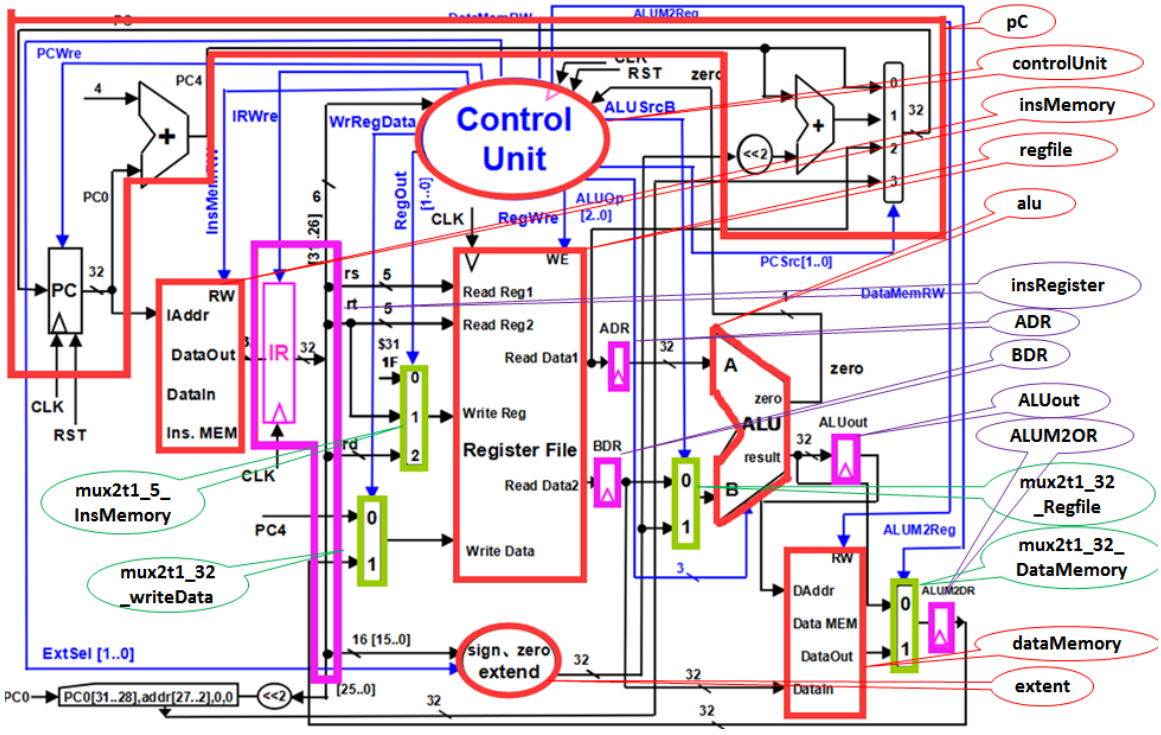


图 14 数据通路模块划分（含名称）

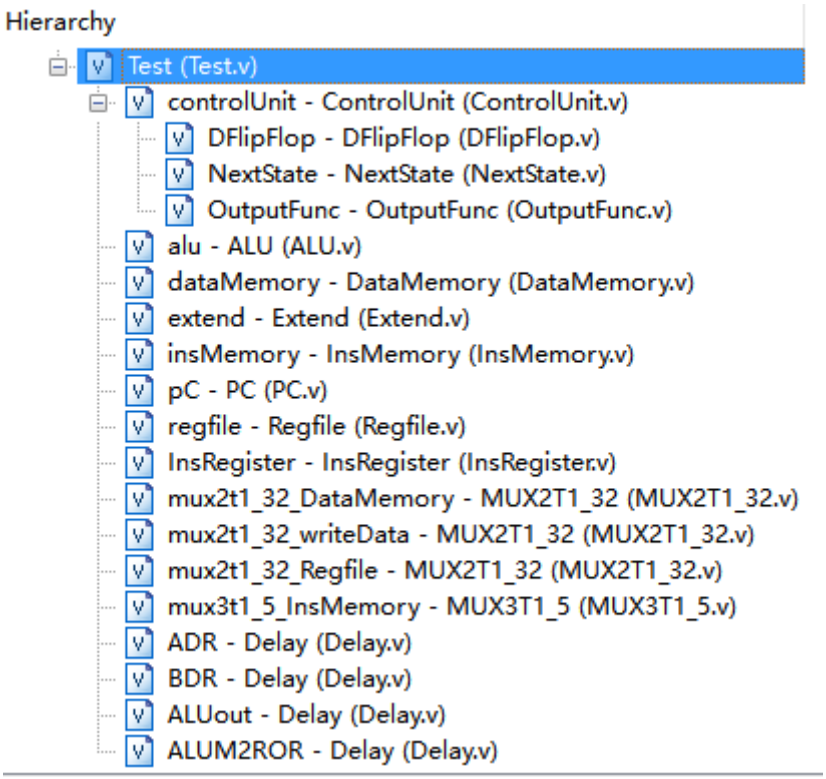


图 15 子模块名称

其中，模块MUX2T1_32是32位的二选一数据选择器，MUX3T1_5则是5位的。由于ADR、BDR、ALUout、ALUM2DR四个寄存器的实现方法相同，只是位置、输入输出不同，它们可以用同一个Delay模块实现。而模块InsRegister是结合了IR并将指令寄存器输出32位的指令分解为6位的Opcode，5位的rs,rt,rd，16位的immediate和26位的bits_26。

2) 规定各模块的接口；

各模块接口直接采用数据通路图中的名称，统一规范。

3) 进行子模块设计

a) 控制单元的设计

如前所述，我将控制单元分为三个子模块。由图很容易写出各连接接口。D触发器模块用于状态重置和转到下一指令。下一状态模块对每个当前状态分析给出下一状态。例如当前状态是IF时，下一状态一定是ID，而当前状态是ID时，则要根据Opcode分情况讨论，以此类推。此次我采用了不同的思路来实现控制信号输出模块，单周期CPU设计时，由于输出的控制信号只和指令Opcode有关，因此对每条指令讨论，给出每个控制信号的值比较方便，但多周期CPU的设计涉及到指令所处阶段，一一对指令讨论不同

阶段的情况很繁琐，且由于一个控制信号一般只在某几种情况下是不同的值，如果将控制信号作为讨论的对象将会非常方便。例如PCWre只在当前状态是IF且指令不是halt时等于1，其他情况下都等于0.这样只用一句话就说清了PCWre的值，不必对每个指令都讨论一次。另外，先将二进制表示的阶段和指令存储用parameter声明可以使代码清晰明了，不易出错。

```
parameter [2:0] IF = 3'b000,
                ID = 3'b001,
                EXE_3stages = 3'b101,
                EXE_4stages = 3'b110,
                EXE_5stages = 3'b010,
                MEM = 3'b011,
                WB_4stages = 3'b111,
                WB_5stages = 3'b100;

parameter [5:0] add = 6'b000000,
                addi = 6'b000010,
                sub = 6'b000001,
                ori = 6'b010010,
                And = 6'b010001,
                Or = 6'b010000,
                move = 6'b100000,
                sw = 6'b110000,
                lw = 6'b110001,
                beq = 6'b110100,
                halt = 6'b111111,
                sll = 6'b011000,
                slt = 6'b100111,
                j = 6'b111000,
                jr = 6'b111001,
                jal = 6'b111010;
```

b) ALU的设计

在表2（ALU运算功能表列出的8个功能中，本实验只用到了前7个，因此只写前7个。通过输入的ALUOp来判断应当执行哪个功能。最后还要判断结果是否等于0，从而为标志位Zero赋值，输出Zero作为控制单元的输入用来控制beq是否发生跳转。

c) 存储器设计

存储单元是字节宽度，一律不使用32位。我采用大端规则存储

数据存储器：考虑读和写。

```
// 大端规则存储
always @(DAddr or DataIn or RW) begin
   if (RW == 1) begin // write
      memory[DAddr] = DataIn[31:24];
      memory[DAddr + 1] = DataIn[23:16];
      memory[DAddr + 2] = DataIn[15:8];
      memory[DAddr + 3] = DataIn[7:0];
   end
   else begin // read
      DataOut={memory[DAddr],memory[DAddr+1],
      memory[DAddr+2],memory[DAddr+3]};
   end
end
```

指令存储器：本设计只考虑读不考虑写。

它们内部均有一个数组存储数据，每次读或写都进行更新。

d) PC设计

增加了两个输入，并增加了一个输出PC4=pc+4，用于作为寄存器组其中一个可选的写入的数据。

e) 寄存器组设计

寄存器组有32个寄存器，每个寄存器用32位的D触发器实现。和单周期的实现类似。

f) 指令寄存器设计

用时钟触发，且有写使能信号IRWre，增加输出pc地址的后26位，用于计算j和jal的跳转地址。

g) ADR等四个小寄存器的设计

时钟触发，实现相同的寄存器Delay

4) 对模块编程

```
//////////////////////////////////////////////////////////////////////////////////
// 控制单元大模块
module ControlUnit(
    Opcode,
    Zero,
    RegWre,
    PCWre,
    ALUSrcB,
    ALUOp,
    ALUM2Reg,
    RegOut,
    DataMemRW,
```

```
        PCSrc,
        ExtSel,
        WrRegData,
        //IRWre,
        CLK,
        RST,
        state
        );
        input[5:0] Opcode;
        input Zero, CLK, RST;
        output RegWre, PCWre, ALUSrcB, ALUM2Reg, DataMemRW, WrRegData/*,
IRWre*/;
        output [2:0] ALUOp;
        output [1:0] ExtSel;
        output [1:0] PCSrc;
        output [1:0] RegOut;
        output [2:0] state;
        wire [2:0] nextState;

        parameter [2:0] IF = 3'b000,
                        ID = 3'b001,
                        EXE_3stages = 3'b101,
                        EXE_4stages = 3'b110,
                        EXE_5stages = 3'b010,
                        MEM = 3'b011,
                        WB_4stages = 3'b111,
                        WB_5stages = 3'b100;


        parameter [5:0] add = 6'b000000,
                        addi = 6'b000010,
                        sub = 6'b000001,
                        ori = 6'b010010,
                        And = 6'b010001,
                        Or = 6'b010000,
                        move = 6'b100000,
                        sw = 6'b110000,
                        lw = 6'b110001,
                        beq = 6'b110100,
                        halt = 6'b111111,
                        sll = 6'b011000,
                        slt = 6'b100111,
                        j = 6'b111000,
                        jr = 6'b111001,
```

```
                    jal = 6'b111010;

    DFlipFlop DFlipFlop(nextState, RST, CLK, state);
    NextState NextState(state, Opcode, nextState);
    OutputFunc OutputFunc(state, Opcode, Zero,
        RegWre, PCWre, ALUSrcB, ALUM2Reg, DataMemRW, WrRegData,/*
IRWre,*/
        ALUOp, ExtSel, PCSrc, RegOut);

endmodule

// 小模块：D触发器
module DFlipFlop(
    nextState, RST, CLK, state
    );
    input [2:0] nextState;
    input RST, CLK;
    output reg [2:0] state;
    always @(posedge CLK or negedge RST) begin
        if (RST==0) begin
            // reset
            state = 3'b000;
        end
        else begin
            #1;
            state = nextState;
        end
    end
endmodule

// 小模块：下一状态
module NextState(
    state, Opcode, nextState
    );
    input [2:0] state;
    input [5:0] Opcode;
    output reg [2:0] nextState;
    parameter [2:0] IF = 3'b000,
                    ID = 3'b001,
                    EXE_3stages = 3'b101,
                    EXE_4stages = 3'b110,
                    EXE_5stages = 3'b010,
                    MEM = 3'b011,
                    WB_4stages = 3'b111,
```

```
                WB_5stages = 3'b100;

    always @(state or Opcode) begin
        case(state)
            IF: nextState = ID;
            ID: begin
                case(Opcode)
                    6'b111000, 6'b111010, 6'b111001, 6'b111111: //
j,jal,jr,halt
                        nextState = IF;
                    6'b110100: // beq
                        nextState =  EXE_3stages;
                    6'b110000, 6'b110001: // sw, lw
                        nextState =  EXE_5stages;
                    default:
                        nextState = EXE_4stages;
                endcase
            end
            EXE_4stages: nextState = WB_4stages;
            EXE_5stages: nextState = MEM;
            MEM: begin
                case(Opcode)
                    6'b110001: nextState = WB_5stages; // lw
                    default: nextState = IF; // sw
                endcase
            end
            default: nextState = IF; // EXE_3stages: WB_4stages: WB_5stages
        endcase
    end

endmodule

// 小模块：输出控制信号
module OutputFunc(
        state, Opcode, Zero,
        RegWre, PCWre, ALUSrcB, ALUM2Reg, DataMemRW, WrRegData,/*
IRWre,*/
        ALUOp, ExtSel, PCSrc, RegOut
    );
    input [2:0] state;
    input [5:0] Opcode;
    input Zero;
    output reg RegWre, PCWre, ALUSrcB, ALUM2Reg, DataMemRW,
WrRegData/*, IRWre*/;
```

```
output reg[2:0] ALUOp;
output reg[1:0] ExtSel;
output reg[1:0] PCSrc;
output reg[1:0] RegOut;

parameter [2:0] IF = 3'b000,
                ID = 3'b001,
                EXE_3stages = 3'b101,
                EXE_4stages = 3'b110,
                EXE_5stages = 3'b010,
                MEM = 3'b011,
                WB_4stages = 3'b111,
                WB_5stages = 3'b100;


parameter [5:0] add = 6'b000000,
                addi = 6'b000010,
                sub = 6'b000001,
                ori = 6'b010010,
                And = 6'b010001,
                Or = 6'b010000,
                move = 6'b100000,
                sw = 6'b110000,
                lw = 6'b110001,
                beq = 6'b110100,
                halt = 6'b111111,
                sll = 6'b011000,
                slt = 6'b100111,
                j = 6'b111000,
                jr = 6'b111001,
                jal = 6'b111010;
always @(state or Opcode) begin
    RegWre = 0;
    PCWre = 0;
    ALUSrcB = 0;
    ALUM2Reg = 0;
    DataMemRW = 0;
    WrRegData = 0;
    //IRWre = 0;
    ALUOp = 3'b000;
    ExtSel = 2'b00;
    PCSrc = 2'b00;
    RegOut = 2'b00;
```

```
            if(state == IF && Opcode != halt) PCWre = 1;

        if(Opcode==addi||Opcode==ori||Opcode==lw||Opcode==sw||Opcode==sll)
ALUSrcB=1;
            if(state==WB_5stages) ALUM2Reg=1;
           if(state==WB_4stages||state==WB_5stages||(state!=IF&&Opcode==jal))
RegWre=1;
            if(state==WB_4stages||state==WB_5stages) WrRegData=1;
            if(state==MEM&&Opcode==sw) DataMemRW=1;
            //if(state==IF) IRWre=1;
            case(Opcode)
                sub, beq: ALUOp=3'b001;
                Or, ori: ALUOp=3'b101;
                And: ALUOp=3'b110;
                sll: ALUOp=3'b100;
                slt: ALUOp=3'b010;
                default: ALUOp=3'b000;
            endcase
            case(Opcode)
                beq: if(Zero==1) PCSrc = 2'b01;
                jr: PCSrc = 2'b10;
                j, jal: PCSrc = 2'b11;
                default: PCSrc = 2'b00;
            endcase
            case(Opcode)
                jal: RegOut = 2'b00;
                addi, ori, lw: RegOut = 2'b01;
                add, sub, Or, And, move, slt, sll: RegOut = 2'b10;
                default: RegOut = 2'b11;
            endcase
            case(Opcode)
                sll: ExtSel = 2'b00;
                ori: ExtSel = 2'b01;
                addi, lw, sw, beq: ExtSel = 2'b10;
                default: ExtSel = 2'b11;
            endcase
        end
endmodule

//////////////////////////////////////////////////////////////////////////////////
module ALU(A, B, Zero, result, ALUOp);
        input [31:0] A, B;
        input [2:0] ALUOp;
        output reg Zero;
```

```
    output reg [31:0] result;

    always @(A or B or ALUOp or result) begin
        case(ALUOp)
            3'b000:result = A + B;
            3'b001:result = A - B;
            3'b010:result = (A < B ? 1 : 0);
            3'b011:result = A >> B;
            3'b100:result = A << B;
            3'b101:result = A | B;
            3'b110:result = A & B;

        endcase
        if (result == 32'b0) Zero = 1;
        else Zero = 0;
    end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module DataMemory(RW, DAddr, DataIn, DataOut);
    input [31:0] DAddr, DataIn;
    input RW;
    output reg[31:0] DataOut;
    reg [7:0] memory[0:400];
    // 数据存储器的存储单元必须是字节宽度，一律不能使用32位

    // 大端规则存储
    always @(DAddr or DataIn or RW) begin
        if (RW == 1) begin // write
            memory[DAddr] = DataIn[31:24];
            memory[DAddr + 1] = DataIn[23:16];
            memory[DAddr + 2] = DataIn[15:8];
            memory[DAddr + 3] = DataIn[7:0];
        end
        else begin // read

    DataOut={memory[DAddr],memory[DAddr+1],memory[DAddr+2],memory[D
Addr+3]};
        end
    end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module Extend(bits_16, ExtSel, bits_32);
    input [15:0] bits_16;
    input [1:0] ExtSel;
```

```
    output reg [31:0] bits_32;

    always@(ExtSel) begin
        // zero extent-sa
        if(ExtSel == 2'b00) assign bits_32 = {27'h00000, bits_16[10:6]};
        // zero extent-immediate
        if(ExtSel == 2'b01) assign bits_32 = {16'h0000, bits_16};
        // sign extent-immediate
        if(ExtSel == 2'b10) assign bits_32 = bits_16[15] ? {16'hffff, bits_16} :
{16'h0000, bits_16};
        // 未用

    end
endmodule
////////////////////////////////////////////////////////////////////////////////
module InsMemory(IAddr, IDataOut);
    // 本设计中指令寄存器只读不写，RW=InsMemRW=0
    input [31:0] IAddr;
    output reg[31:0] IDataOut;
    reg [7:0] memory[0:400];
    initial
            begin
                memory[248] = 8'he0;
                memory[249] = 8'h00;
                memory[250] = 8'h00;
                memory[251] = 8'h40;

                memory[256] = 8'h48;
                memory[257] = 8'h01;
                memory[258] = 8'h00;
                memory[259] = 8'h02;

                memory[260] = 8'h08;
                memory[261] = 8'h02;
                memory[262] = 8'h00;
                memory[263] = 8'h03;

                memory[264] = 8'h04;
                memory[265] = 8'h41;
                memory[266] = 8'h18;
                memory[267] = 8'h00;

                memory[268] = 8'h40;
                memory[269] = 8'h22;
```

```
memory[270] = 8'h20;
memory[271] = 8'h00;

memory[272] = 8'h44;
memory[273] = 8'h61;
memory[274] = 8'h28;
memory[275] = 8'h00;

memory[276] = 8'h80;
memory[277] = 8'h80;
memory[278] = 8'h30;
memory[279] = 8'h00;

memory[280] = 8'h44;
memory[281] = 8'h62;
memory[282] = 8'h38;
memory[283] = 8'h00;

memory[284] = 8'he8;
memory[285] = 8'h00;
memory[286] = 8'h00;
memory[287] = 8'h4d;

memory[288] = 8'h9c;
memory[289] = 8'h41;
memory[290] = 8'h48;
memory[291] = 8'h00;

memory[292] = 8'h9c;
memory[293] = 8'h22;
memory[294] = 8'h50;
memory[295] = 8'h00;

memory[296] = 8'h60;
memory[297] = 8'h60;
memory[298] = 8'h18;
memory[299] = 8'h40;

memory[300] = 8'hd0;
memory[301] = 8'h23;
memory[302] = 8'hff;
memory[303] = 8'hfe;

memory[304] = 8'hfc;
```

```
                memory[305] = 8'h00;
                memory[306] = 8'h00;
                memory[307] = 8'h00;

                memory[308] = 8'hc0;
                memory[309] = 8'h26;
                memory[310] = 8'h00;
                memory[311] = 8'h02;

                memory[312] = 8'hc4;
                memory[313] = 8'h28;
                memory[314] = 8'h00;
                memory[315] = 8'h02;

                memory[316] = 8'he7;
                memory[317] = 8'he0;
                memory[318] = 8'h00;
                memory[319] = 8'h00;
            end
        always @ (IAddr)
            begin
                IDataOut = {memory[IAddr],memory[IAddr+1],
memory[IAddr+2],memory[IAddr+3]};
            end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module PC(pcIn, CLK, Reset, PCSrc, PCWre, pcOut, readData1, bits_26, PC4);
    input [31:0] pcIn;
    input CLK;
    input Reset;
    input [1:0] PCSrc;
    input PCWre;
    input [31:0] readData1;
    input [25:0] bits_26;
    output reg [31:0] pcOut;
    output reg [31:0] PC4;
    reg [31:0] pc;
    always @(posedge CLK or negedge Reset) begin
        if (Reset == 0) begin
            // reset
            pcOut = 32'hF8;
            pc = 32'hF8;
        end
        else if(PCWre == 1) begin
```

```
                if(PCSrc == 2'b00) pcOut = pc + 4;
                else if(PCSrc == 2'b01) pcOut = pc + 4 + (pcIn << 2); // beq跳转
                else if(PCSrc == 2'b10) pcOut = readData1; // jr的pc <- rs
               else pcOut = {4'b0000, bits_26, 2'b00}; // j,jal
                pc = pcOut;
            end
            PC4 = pc + 4;
        end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module Regfile(
    ReadReg1, ReadReg2, WriteData, WriteReg, WE, CLK, clrn, ReadData1,
ReadData2);
    input [4:0] ReadReg1, ReadReg2, WriteReg;
    input [31:0] WriteData;
    input WE, CLK, clrn;
    output [31:0] ReadData1, ReadData2;
    reg [31:0] register[1:31]; //r1-r31
    integer i; // 注意要在外面声明
    assign ReadData1 = (ReadReg1 == 0) ? 0 : register[ReadReg1]; //read
    assign ReadData2 = (ReadReg2 == 0) ? 0 : register[ReadReg2]; //read
    always @(posedge CLK or negedge clrn) begin
        if (clrn == 0) begin
            // reset
            for(i=1;i<31;i=i+1)
            register[i] = 0;

        end
        else begin
            // write
            if((WriteReg!=0) && (WE == 1))
            register[WriteReg] = WriteData;
        end
    end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module InsRegister(iDataOut, /*IRWre,*/ Opcode, rs, rt, rd, immediate, bits_26,
CLK);
    input [31:0] iDataOut;
    input CLK/*, IRWre*/;
    output reg [5:0] Opcode;
    output reg [4:0] rs, rt, rd;
    output reg [15:0] immediate;
    output reg [25:0] bits_26;
```

```
	always @(negedge CLK) begin
		Opcode <= iDataOut[31:26];
		rs <= iDataOut[25:21];
		rt <= iDataOut[20:16];
		rd <= iDataOut[15:11];
		immediate <= iDataOut[15:0];
		bits_26 <= iDataOut[25:0];
	end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module MUX2T1_32(i0, i1, out, select);
	input [31:0] i0;
	input [31:0] i1;
	output reg [31:0] out;
	input select;

	always @(i0 or i1 or select) begin
		if(select == 0) out = i0;
		else out = i1;
	end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module MUX3T1_5(i0, i1, out, select);
	input [4:0] i0;
	input [4:0] i1;
	output reg [4:0] out;
	input [1:0] select;

	always @(i0 or i1 or select) begin
		if(select == 2'b00) out = 5'b11111;
		if(select == 2'b01) out = i0;
		if(select == 2'b10) out = i1;
	end
endmodule
//////////////////////////////////////////////////////////////////////////////////
module Delay(
	in,
	out,
	CLK
	);
	input [31:0] in;
	input CLK;
	output reg [31:0] out;
	always @(posedge CLK) begin
```

```
        out = in;
    end
endmodule
//////////////////////////////////////////////////////////////////////////////////
```

5) 连接各模块完成系统设计。

verilog是通过模块调用或称为模块实例化的方式来实现这些子模块与高层模块的连接。调用模块实例的一般形式为：<模块名><参数列表><实例名>（<端口列表>）；信号端口可通过名称关联即.PortName(port_expr)。要注意的是引用时要严格按照模块定义的端口顺序来连接,不用标明原模块定义时规定的端口名：

Design u_1(u_1的端口1，u_1的端口2，u_1的端口3，u_1的端口……)；

用’.’符号来标明原模块定义时规定的端口名：

```
Design u_2(.(端口1(u_1的端口1),
           .(端口2(u_1的端口2),
           .(端口3(u_1的端口3),
                    ……  );
```

各子模块是通过线连接起的，由于每根线两端连接两个模块，不为单独一个模块所有，我采用为衔接的线另外命名（下图橙色的线）。但控制单元模块输出的十个控制信号（下图蓝色）及输入的Opcode和zero,clk,rst等含义明显，不再单独命名。所有线都写做wire [size - 1:0] wireName;两个子模块的两个端口通过一根线连接，则分别写做.PortName1(port_expr)和.PortName2(port_expr)。由此就可以轻松表示各模块之间的连接关系了。

具体见下图：

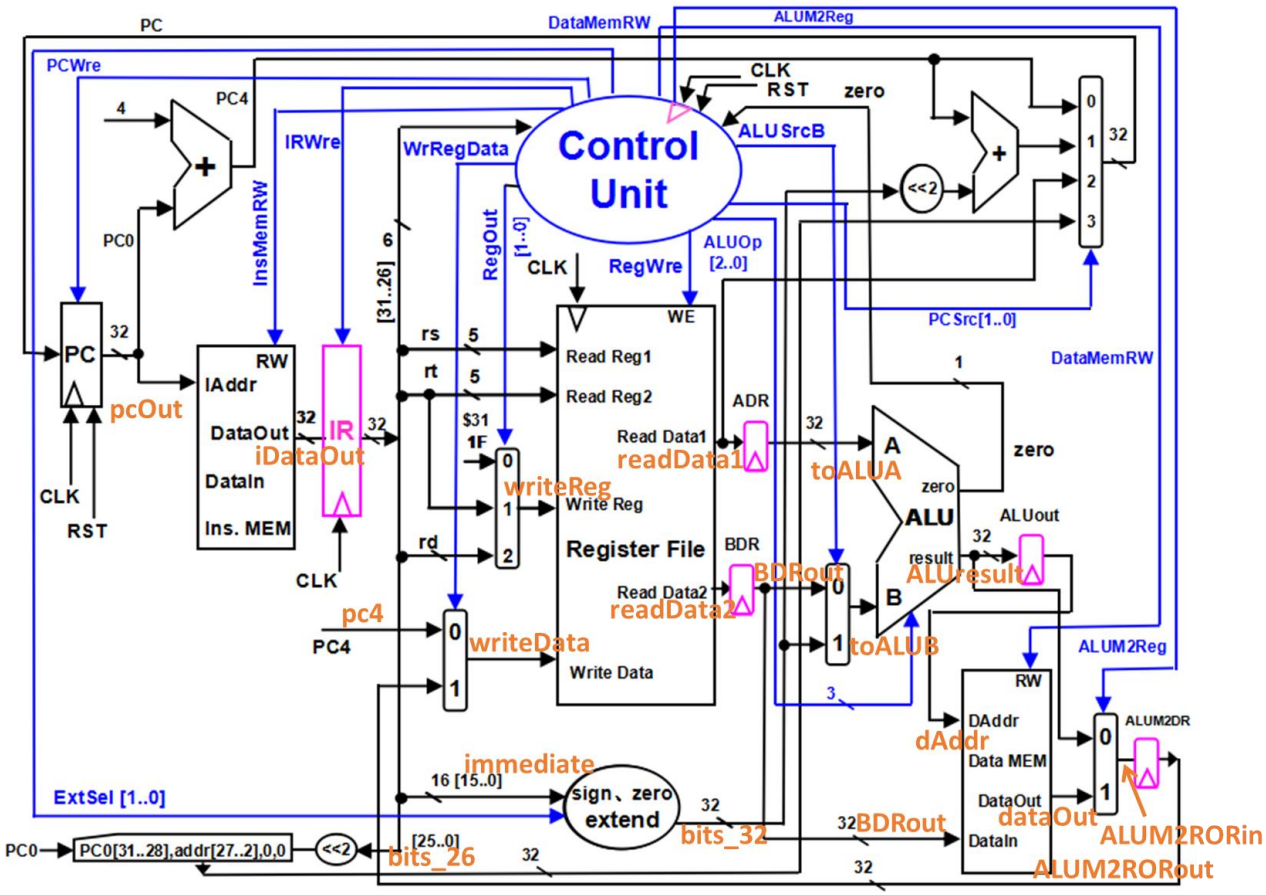


图 16　子模块的连接

代码如下：

```
module Test;

    // Inputs
    reg clk, reset;

    // Outputs

    // ControlUnit
    wire RegWre;
    wire PCWre;
    wire ALUSrcB;
    wire [2:0] ALUOp;
    wire ALUM2Reg;
    wire [1:0] RegOut;
    wire DataMemRW;
    wire [1:0] PCSrc;
    wire [1:0] ExtSel;
    wire WrRegData;
    //wire IRWre;
    wire [2:0] state;

    // ALU
    wire [31:0] toALUA;
    wire [31:0] toALUB;
    wire [31:0] ALUresult;
    wire Zero;

    // DataMemory
    wire [31:0] dataOut;
    wire [31:0] dAddr;
    wire [31:0] ALUM2RORin;
    wire [31:0] ALUM2RORout;

    // Extend
    wire [31:0] bits_32;

    // InsMemory
    wire [31:0] pcOut;
    wire [31:0] iDataOut;

    // Regfile
```

```
    wire [4:0] writeReg;
    wire [31:0] writeData;
    wire [31:0] readData1;
    wire [31:0] readData2;

    // InsRegister
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [5:0] Opcode;
    wire [15:0] immediate;

    // PC
    wire [31:0] pc4;
    wire [25:0] bits_26;

    // Delay
    wire [31:0] BDRout;


    // Instantiate the Unit Under
Test (UUT)
    ControlUnit controlUnit (
        .Opcode(Opcode),
        .Zero(Zero),
        .RegWre(RegWre),
        .PCWre(PCWre),
        .ALUSrcB(ALUSrcB),
        .ALUOp(ALUOp),

    .ALUM2Reg(ALUM2Reg),
        .RegOut(RegOut),

    .DataMemRW(DataMemRW)
,
        .PCSrc(PCSrc),
        .ExtSel(ExtSel),

    .WrRegData(WrRegData),
        //.IRWre(IRWre),
        .CLK(clk),
        .RST(reset),
        .state(state)
    );

ALU alu (
    .A(toALUA),
    .B(toALUB),
    .Zero(Zero),
    .result(ALUresult),
    .ALUOp(ALUOp)
);

DataMemory dataMemory (
    .RW(DataMemRW),
    .DAddr(dAddr),
    .DataIn(BDRout),
    .DataOut(dataOut)
);

Extend extend (
    .bits_16(immediate),
    .ExtSel(ExtSel),
    .bits_32(bits_32)
);

InsMemory insMemory (
    .IAddr(pcOut),
    .IDataOut(iDataOut)
);

PC pC (
    .pcIn(bits_32),
    .CLK(clk),
    .Reset(reset),
    .PCSrc(PCSrc),
    .PCWre(PCWre),
    .pcOut(pcOut),
    .readData1(readData1),
    .bits_26(bits_26),
    .PC4(pc4)
);

Regfile regfile (
    .ReadReg1(rs),
    .ReadReg2(rt),
    .WriteData(writeData),
    .WriteReg(writeReg),
```

```
        .WE(RegWre),
        .CLK(clk),
        .clrn(reset),
        .ReadData1(readData1),
        .ReadData2(readData2)
    );

    InsRegister InsRegister(
        .iDataOut(iDataOut),
        //.IRWre(IRWre),
        .Opcode(Opcode),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .immediate(immediate),
        .bits_26(bits_26),
        .CLK(clk)
    );

    MUX2T1_32
mux2t1_32_DataMemory (
        .i0(ALUresult),
        .i1(dataOut),
        .out(ALUM2RORin),
        .select(ALUM2Reg)
    );

    MUX2T1_32
mux2t1_32_writeData (
        .i0(pc4),
        .i1(ALUM2RORout),
        .out(writeData),
        .select(WrRegData)
    );

    MUX2T1_32
mux2t1_32_Regfile (
        .i0(BDRout),
        .i1(bits_32),
        .out(toALUB),
        .select(ALUSrcB)
    );

    MUX3T1_5
mux3t1_5_InsMemory (
        .i0(rt),
        .i1(rd),
        .out(writeReg),
        .select(RegOut)
    );

    Delay ADR (
        .in(readData1),
        .out(toALUA),
        .CLK(clk)
    );

    Delay BDR (
        .in(readData2),
        .out(BDRout),
        .CLK(clk)
    );

    Delay ALUout (
        .in(ALUresult),
        .out(dAddr),
        .CLK(clk)
    );

    Delay ALUM2ROR (
        .in(ALUM2RORin),
        .out(ALUM2RORout),
        .CLK(clk)
    );


    initial begin
        // Initialize Inputs
        clk = 0;
        reset = 0;
        #5;
        reset = 1;
        forever #5 clk = ~clk;
    end

endmodule
```

4. 测试单周期CPU

（1） 测试程序段

表 3 测试程序段

地址	汇编程序	指令代码				
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x000000F8	j 0x00000100	111000	00000	00000	0000 0000 0100 0000	e0000040
0x000000FC	……					
0x00000100	ori $1, $0, 2 ($1 = 2)	010010	00000	00001	0000 0000 0000 0010	48010002
0x00000104	addi $2,$0,3 ($2 = 3)	000010	00000	00010	0000 0000 0000 0011	08020003
0x00000108	sub $3,$2,$1 ($3 = 1)	000001	00010	00001	0001 1000 0000 0000	04411800
0x0000010C	or $4,$1,$2 ($4 = 3)	010000	00001	00010	0010 0000 0000 0000	40222000
0x00000110	and $5,$3,$1($5 = 0)	010001	00011	00001	0010 1000 0000 0000	44612800
0x00000114	move $6, $4 ($6 = 3)	100000	00100	00000	0011 0000 0000 0000	80803000
0x00000118	and $7,$3,$2 ($7 = 1)	010001	00011	00010	0011 1000 0000 0000	44623800
0x0000011C	jal 0x00000134	111010	00000	00000	0000 0000 0100 1101	e800004d
0x00000120	slt $9,$2,$1 ($9 = 0)	100111	00010	00001	0100 1000 0000 0000	9c414800
0x00000124	slt $10,$1,$2 ($10 =1)	100111	00001	00010	0101 0000 0000 0000	9c225000
0x00000128	sll $3,$3,1 ($3 = 2, 4)	011000	00011	00000	0001 1000 0100 0000	60601840
0x0000012C	beq $3,$1,-2 转 128	110100	00001	00011	1111 1111 1111 1110	d023fffe
0x00000130	halt	111111	00000	00000	0000 0000 0000 0000	fc000000
0x00000134	sw $6,2($1)(MEM[4] = 3)	110000	00001	00110	0000 0000 0000 0010	c0260002
0x00000138	lw $8,2($1) ($8 = 3)	110001	00001	01000	0000 0000 0000 0010	c4280002
0x0000013C	jr $31	111001	11111	00000	0000 0000 0000 0000	e7e00000

（2） 将指令代码初始化到指令存储器。

```
module InsMemory(IAddr, IDataOut);
   // 本设计中指令寄存器只读不写，RW=InsMemRW=0
   input [31:0] IAddr;
   output reg[31:0] IDataOut;
   reg [7:0] memory[0:400];
   initial
         begin
            memory[248] = 8'he0;
            memory[249] = 8'h00;
            memory[250] = 8'h00;
            memory[251] = 8'h40;

            memory[256] = 8'h48;
            memory[257] = 8'h01;
            memory[258] = 8'h00;
            memory[259] = 8'h02;

            memory[260] = 8'h08;
            memory[261] = 8'h02;
            memory[262] = 8'h00;
            memory[263] = 8'h03;
```

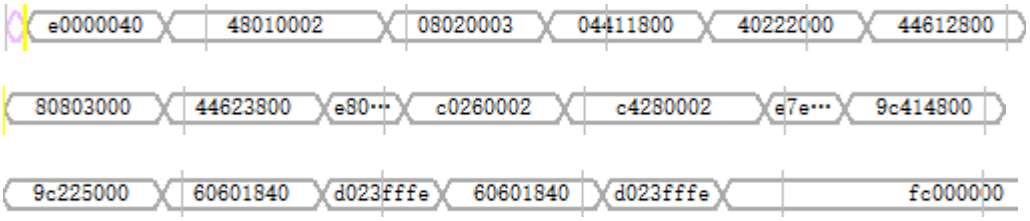
以此类推。

（3） 初始化PC的值，也就是以上程序段首地址PC=0x000000F8，假设以上程序段从0x000000F8地址开始存放。

```
if (Reset == 0) begin
    // reset
    pcOut = 32'hF8;
    pc = 32'hF8;
end
```

（4） 运行Xilinx ISE进行仿真，看波形。

a.指令代码为iDataOut，将其转为十六进制，以方便对比。本测试代码的仿真结果如下:

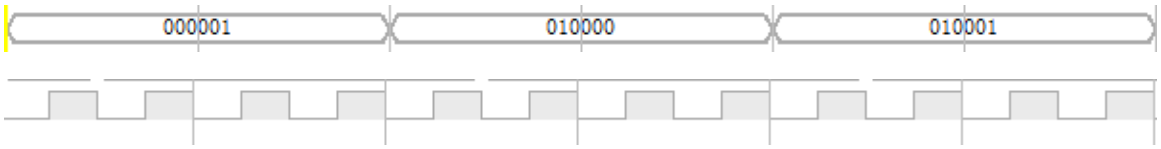


对比步骤1中的测试表，可以看到，每个时钟周期都正确获得了相应的指令，第一个指令j到对应指令，接着顺序执行，到jal指令时跳转到0x00000134地址，再从0x00000134地址对应的sw指令顺序执行到jr指令，接着转到jal的下一条指令slt，顺序执行到beq，判等成立，跳转到128，再次判等不成立，顺序执行到最后一条指令halt后停止。

b.下面每两三行中的第一行为0pcode，第二行为时钟周期，第三行是前两行对应的指令和时钟周期数。

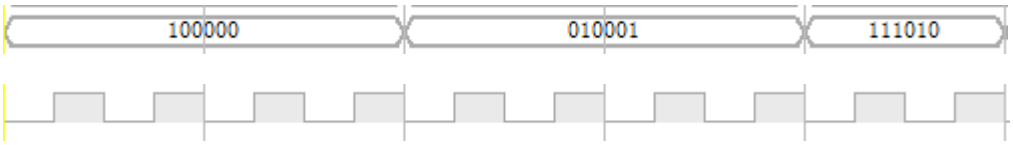
j,2 |ori,4 |addi,4

//////



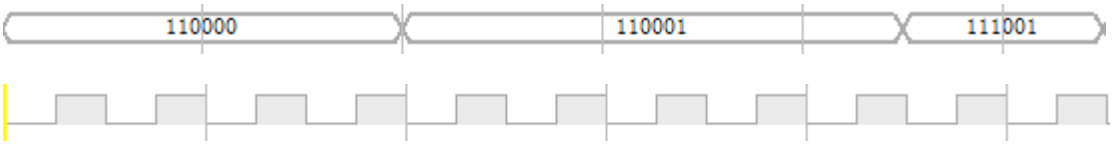
sub,4 |or,4 |and,4

//////



move,4 |and,4 |jal,2

//////

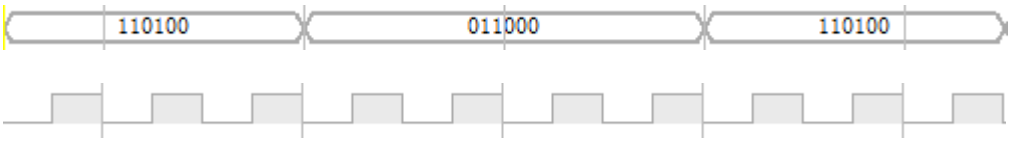


sw,4 |lw,5 |jr,2

//////

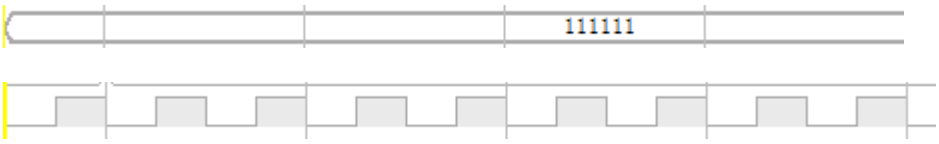
slt,4 |slt,4 |sll,4

//////



beq,3 |sll,4 |beq,3

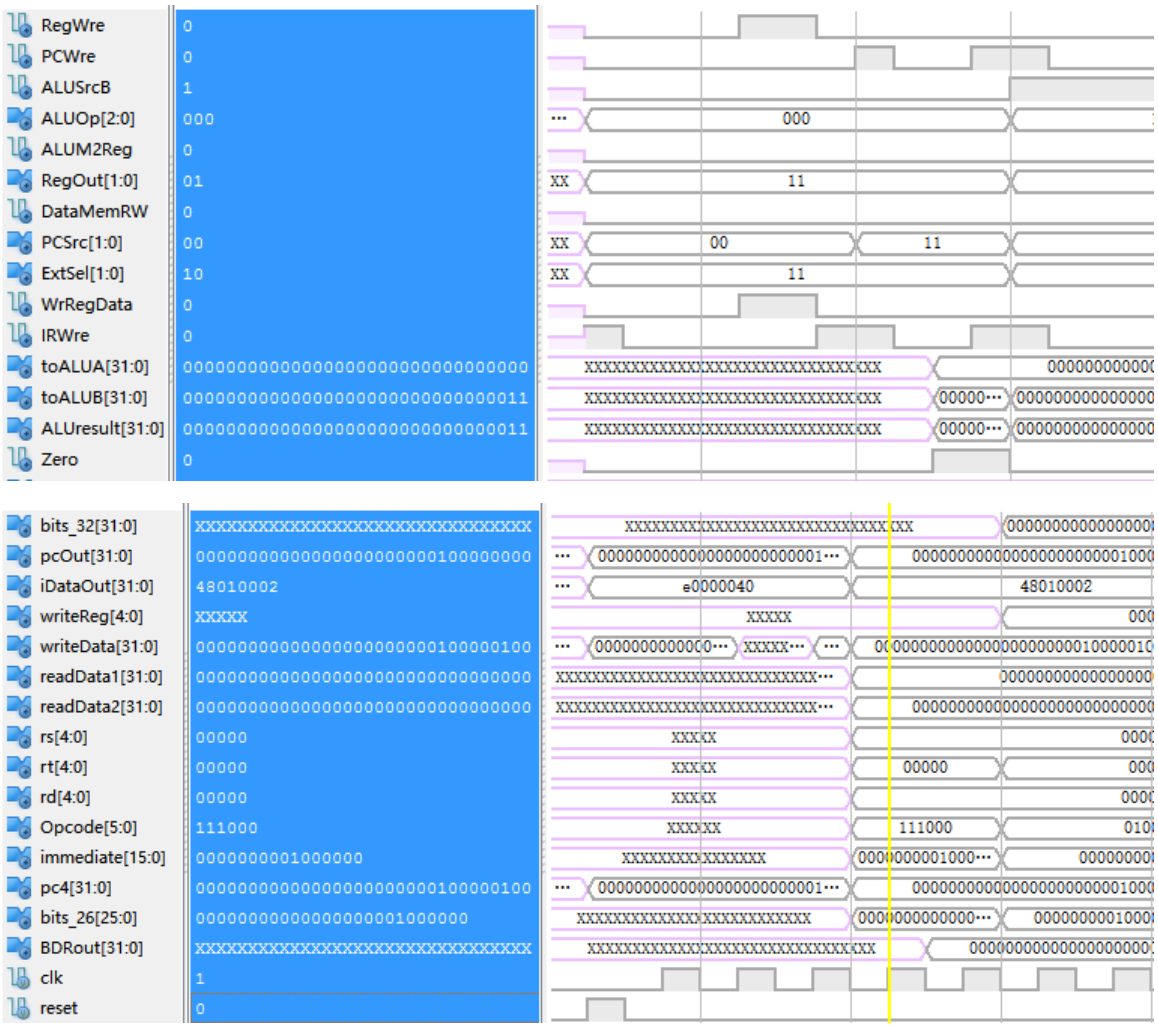
//////



halt

可以看到指令按照正确的顺序执行并且对应的时钟周期也正确分配。

c. 其它变量



另外也可以看到其他的变量是否正确变化。如rs,rt,rd,Opcode是否正确分离。控制单元输出信号是否正确，检查发现是正确的。

d. 寄存器内的结果如下：

	0	1	2	3
0x1	00000002	00000003	00000004	00000003
0x5	00000000	00000003	00000001	00000003
0x9	00000000	00000001	00000000	00000000
0xD	00000000	00000000	00000000	00000000
0x11	00000000	00000000	00000000	00000000
0x15	00000000	00000000	00000000	00000000
0x19	00000000	00000000	00000000	00000000
0x1D	00000000	00000000	00000120	

e. 数据存储器内的结果如下：

	0	1	2	3	4	5	6	7
0x0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	00000000	00000000	00000000	00000011
0x8	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x10	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x18	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x20	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x28	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x30	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x38	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x40	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

f. 指令存储器内的结果如下：

	0	1	2	3	4	5	6	7
0xF8	11100000	00000000	00000000	01000000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x100	01001000	00000001	00000000	00000010	00001000	00000010	00000000	00000011
0x108	00000100	01000001	00011000	00000000	01000000	00100010	00100000	00000000
0x110	01000100	01100001	00101000	00000000	10000000	10000000	00110000	00000000
0x118	01000100	01100010	00111000	00000000	11101000	00000000	00000000	01001101
0x120	10011100	01000001	01001000	00000000	10011100	00100010	01010000	00000000
0x128	01100000	01100000	00011000	01000000	11010000	00100011	11111111	11111110
0x130	11111100	00000000	00000000	00000000	11000000	00100110	00000000	00000010
0x138	11000100	00101000	00000000	00000010	11100111	11100000	00000000	00000000
0x140	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

从0xF8开始，4个字节对应一个字。均和分析设计一致。

5. 实验结果及分析

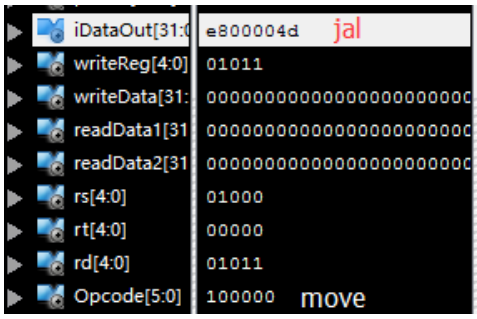
通过以上自己编写的测试代码执行的结果，加上我同时还测试了老师所提供的测试指令，可以判断设计实现的CPU是正确的。实验结果均和期望一致。

六. 实验心得

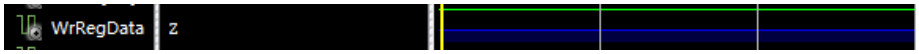
1. 实验过程中遇到的问题及解决的办法

1) jal指令没有将pc+4存入31号寄存器

发现是没有给出对应的控制信号，如regOut应为00却显示11，且只有regOut变量和Opcode不同步改变。尝试将ControlUnit的OutputFunc的检测变化always@(state)改为always@(state or Opcode)，在状态或Opcode改变时均为各控制信号重新赋值，可以成功将Opcode和控制信号对应改变。但是会导致显示的Opcode比正在执行的指令晚一个时钟周期：



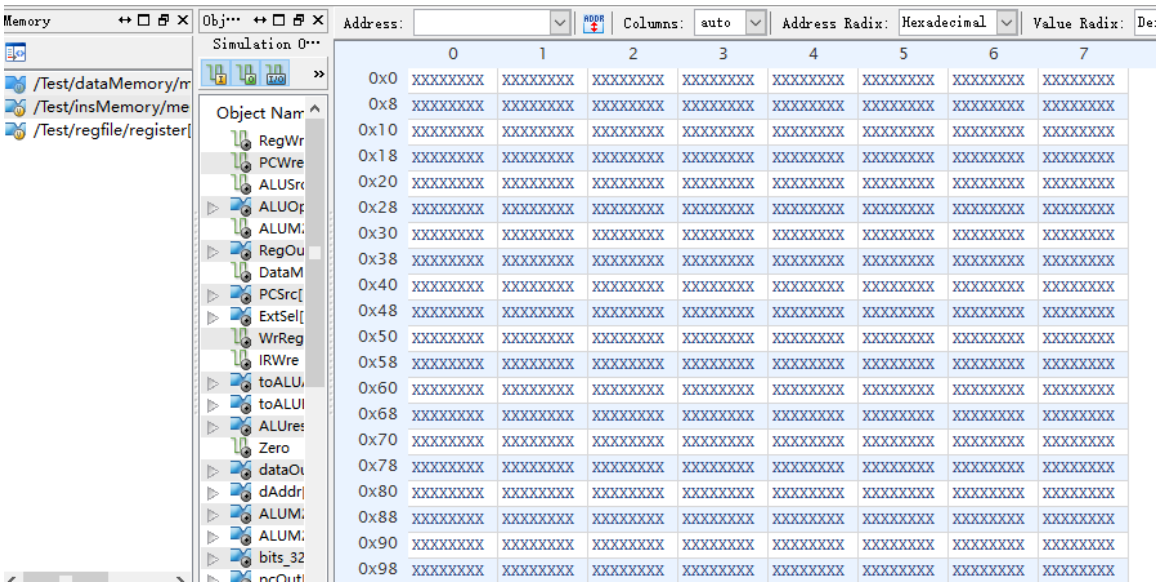
观察仿真发现WrRegData值始终为z，



检查发现test.v中忘记为ControlUnit模块加入该变量，修改如下：

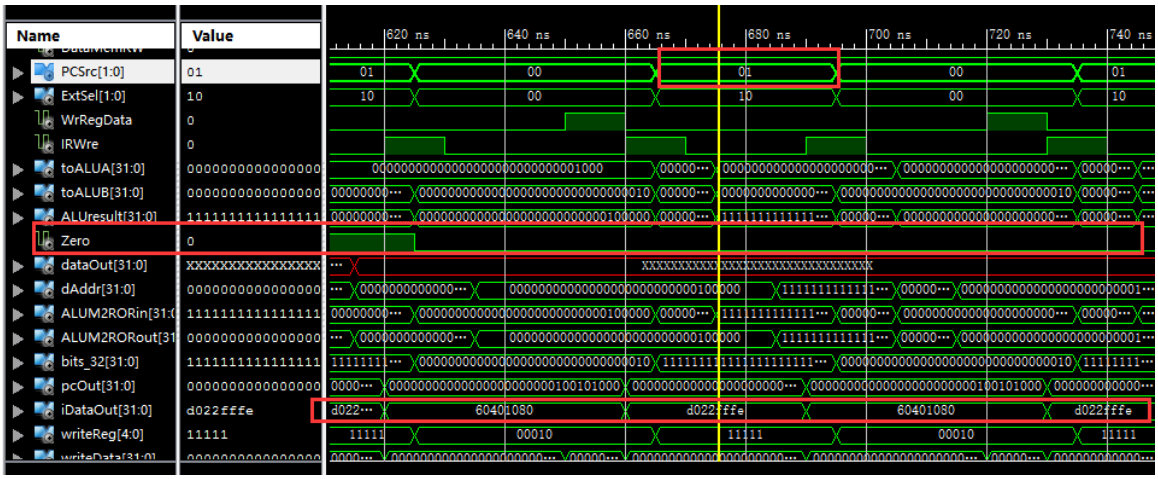
```
// Instantiate the Unit Under Test (UUT)
ControlUnit controlUnit (
    .Opcode(Opcode),
    .Zero(Zero),
    .RegWre(RegWre),
    .PCWre(PCWre),
    .ALUSrcB(ALUSrcB),
    .ALUOp(ALUOp),
    .ALUM2Reg(ALUM2Reg),
    .RegOut(RegOut),
    .DataMemRW(DataMemRW),
    .PCSrc(PCSrc),
    .ExtSel(ExtSel),
    .WrRegData(WrRegData),
    .IRWre(IRWre),
    .CLK(clk),
    .RST(reset)
);
```

2） 数据寄存器内没有内容



这是测试指令没有成功进行到sw指令，也就没有对数据存储器的写操作，因此没有被赋值。

3） 指令beq始终跳转，即使当条件不成立时页跳转，导致最后一直在测试代码的beq和sll之间循环，没有halt。



观察发现，Zero变量是正确判断了，当beq条件不成立即不相等时Zero=0,但是可以看到在beq指令处PCSrc=01，所以才执行了pc+4+immediate的跳转。于是是错误的。

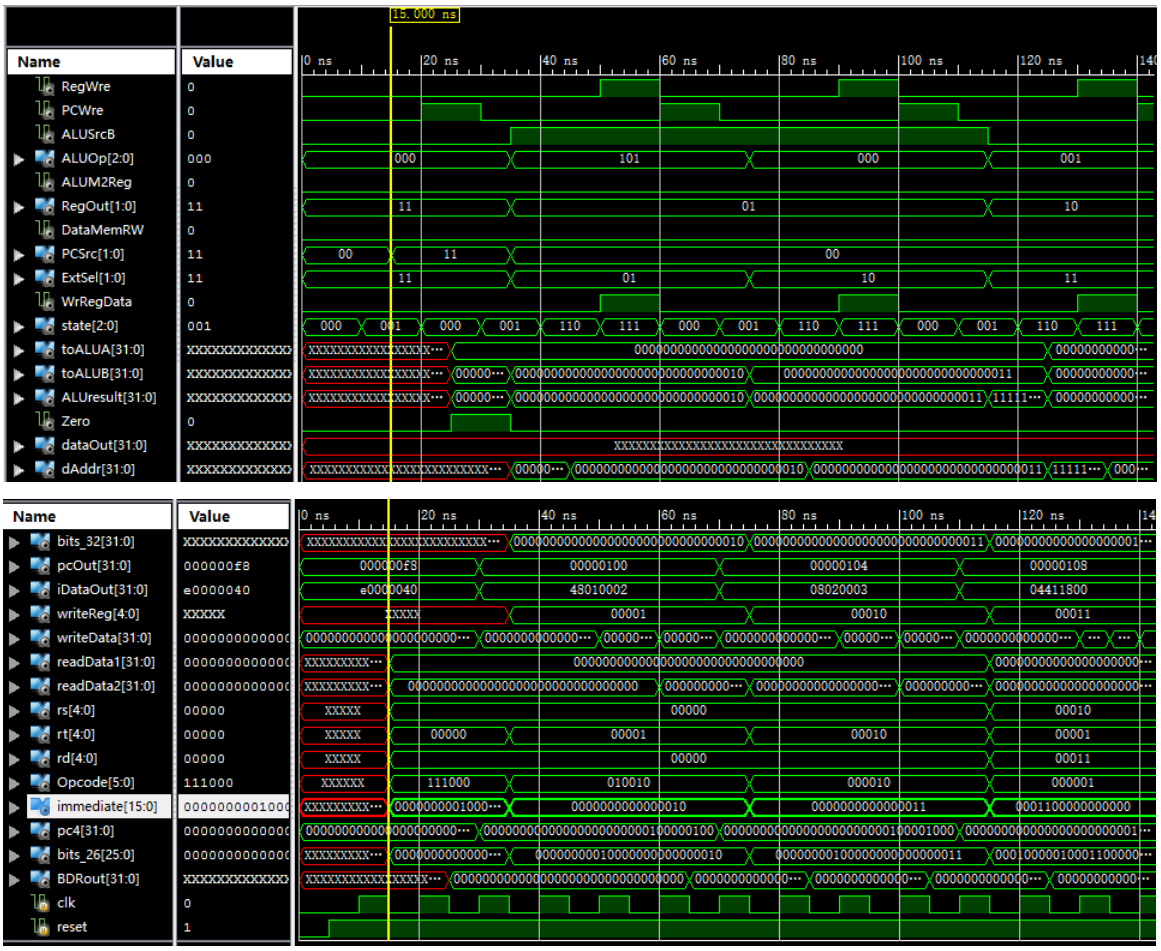
```
case(Opcode)
   beq: PCSrc = 2'b01;
   jr: PCSrc = 2'b10;
   j, jal: PCSrc = 2'b11;
   default: PCSrc = 2'b00;
endcase
```

4) 能成功仿真后遇到的问题主要是指令没有按照预想顺序执行，有段时间卡在了这个问题上，不知道怎么找原因，看变量的值有某个变量是不对应的，但由于多周期的延迟问题，我一开始没有搞清楚是什么地方进行了延迟，执行到一个特定位置的时候对应各变量是当前值还是上一阶段的值，这很重要。实践过程中，我从一个地方找到了突破点。即检查指令执行的时钟周期。因为时钟周期的正确与否反映了控制单元是否正确进行阶段转移，也可以将控制单元的时钟信号做基准去看别的变量的先后变化。由此我找到我有一个指令的时钟周期数错为5，再去检查代码，发现这条指令写错了。改正后得到了正确的结果。

5) 在实验的最后发现了一个bug，即所有指令都是正常执行的，时钟周期也对，但是第一个指令在初始的三个时钟周期后才开始执行，表现为延迟现象。我几乎纠结了一天这是怎么回事。最后通过观察clk,reset,opcode的关系，发现是我原来用reset==1时表示重置，而非用negedge reset且reset==0，这会导致在初始化reset=1后重置了一次，接着设置reset=0表示不再重置，reset从1到0发生了变化，会再次进入pc，判断if(reset)…else…并进入else的分支，导致多进行了一次执行。如果用reset的下降沿来触发就不同了，虽然reset变化前后有两个不同

的状态，但是只有一次触发，这样就不会有其他的延时问题。在改正了这个错误之后，延时问题解决了，仍然有其他问题，都是和clk的触发条件有关，在不同模块里，clk的上升沿或下降沿触发情况不同，要区别对待。经过多次试验，我采用ControlUnit,PC,regfile,Delay用时钟上升沿触发，而InsRegister用下降沿触发，得到了正确结果。

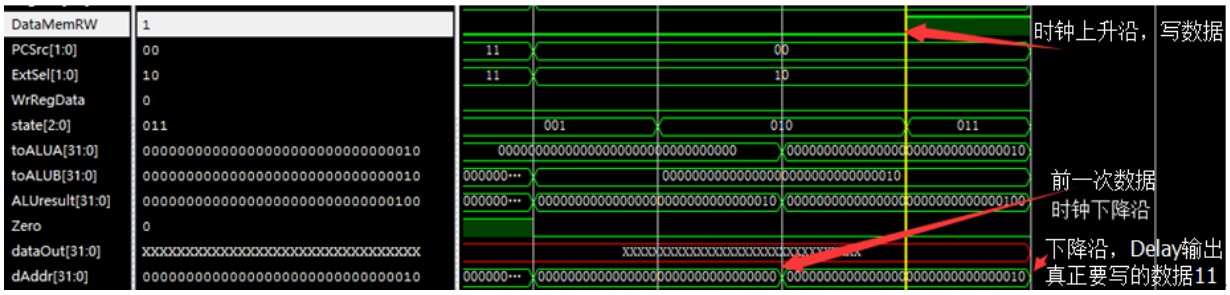
这提示我要注意实验的细节，对实验结果仔细分析发现问题，然后积极寻找错误原因并想办法解决。



6) 解决了上一个问题后发现dataMemory存储有问题，不应该是六个而应是四个单元存储了内容。

	0	1	2	3	4	5	6	7
0x0	XXXXXXXX	XXXXXXXX	00000000	00000000	00000000	00000000	00000000	00000011
0x8	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x10	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x18	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x20	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x28	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x30	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x38	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
0x40	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

我在dataMemory设置了断点进行调试，程序首先停在了下图位置：



上图中“数据”指的是代表地址的数据。将数据11写入了一个旧地址10后，DataMemRW仍为1。数据存储器对应的内容是：

	0	1	2	3	4	5	6	7
0x0	xxxxxxxx	xxxxxxxx	00000000	00000000	00000000	00000011	xxxxxxxx	xxxxxxxx
0x8	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x10	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x18	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x20	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x28	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x30	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x38	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x40	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x48	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x50	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x58	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x60	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

再次写入，这次写入的是真正要写的地址11，

```
always @(DAddr or DataIn or RW) begin
    if (RW == 1) begin // write
        memory[DAddr] = DataIn[31:24];
        memory[DAddr + 1] = DataIn[23:16];
        memory[DAdd
        memory[DAdd
    end
    else begin //
```

/Test/dataMemory/DAddr
Value: 00000000000000000000000000000100
Time: 355,000 ps

从而产生了旧数据覆盖在新数据上的错误

	0	1	2	3	4	5	6	7
0x0	xxxxxxxx	xxxxxxxx	00000000	00000000	00000000	00000000	00000000	00000011
0x8	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x10	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x18	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x20	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x28	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x30	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x38	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x40	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x48	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x50	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

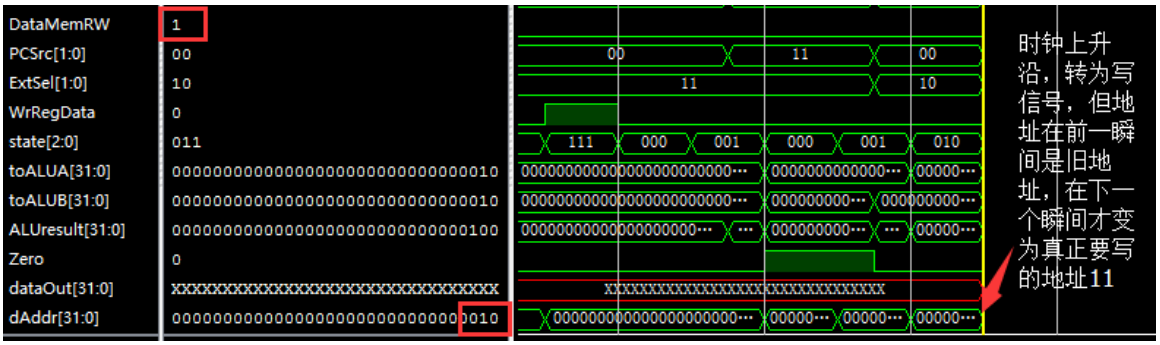
找到对应的代码：

```
module Delay(
    in,
    out,
    CLK
     );
    input [31:0] in;
    input CLK;
    output reg [31:0] out;
    always @(negedge CLK) begin
       out <= in;
    end

endmodule

module DFlipFlop(
    nextState, RST, CLK, state
     );
    input [2:0] nextState;
    input RST, CLK;
    output reg [2:0] state;
    always @ posedge CLK or negedge RST) begin
```

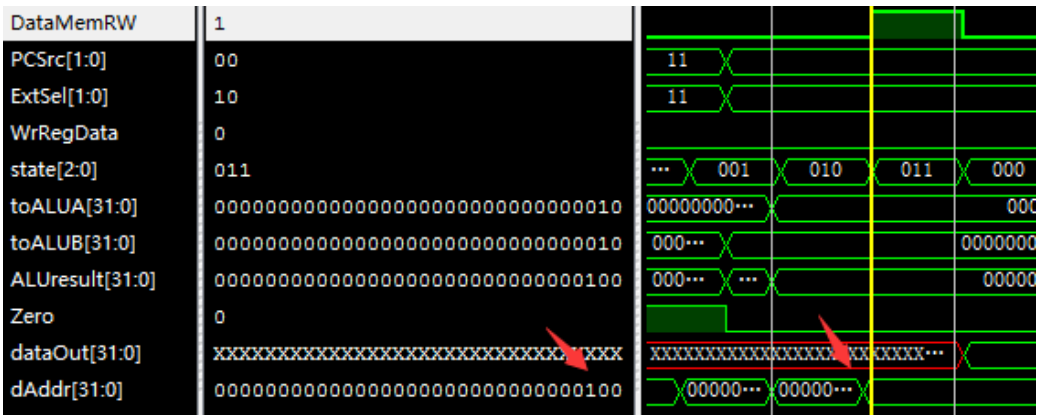
很容易想到应将上两个模块都用上升沿触发。将Delay改为上升沿触发后，出现了新的问题：



为了让DataMemRW=1时读取到的是新的地址，需要延迟DataMemRW的改变或提前dAddr的改变，显然前者容易实现，我在ControlUnit的DFlipFlop模块加了#1的延迟，使得控制信号稍微晚一点发出。

```
#1;
state = nextState;
```

可以看到下图中写数据的信号稍晚于新数据出现，从而达到了目的：



	0	1	2	3	4	5	6	7
0x0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	00000000	00000000	00000000	00000011
0x8	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x10	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x18	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x20	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x28	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x30	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x38	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0x40	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx

小结：这个bug让我熟悉了断点调试的功能并能一步步根据问题分析原因，从而进行改正。正是一点点摸索的过程，让我对多周期CPU设计中的数据切分有了更深刻的理解，也感受到每一个模块尤其是时钟触发的设计举足轻重，一不留神就错了。只有在理解的基础上才能做好。

2. 体会

本次实验具有很强的系统性，需要进行统筹考虑，从分析、设计到实现，每一步骤都必不可少。有上一次单周期CPU设计实验的铺垫，这次实验在基础知识理解上的障碍小了很多。有上一次实验的代码基础，这次编程量也小了很多。但是多周期CPU的设计显然比单周期更为复杂，要考虑的东西更多，核心模块的设计也更难。

看了资料，对多周期CPU设计有了基本认识后我先比较了两次实验的不同，直接在上一次代码的基础上改。我选择从小的方面切入，比如先编写增加的小模块IR、ALUout等，再修改有的模块如PC，Register File用到的数据选择器，它们的输入数据变了。接着我照葫芦画瓢，对照表采用上一次实验的方式修改了控制单元各控制信号。再修改测试程序，增加了一些线和接口。接着进行仿真。然而虽然没有报错，但是却没有结果，各变量都没有成功赋值。

我反思这是我设计的问题，重新看了参考文档，才发现我对多周期CPU控制单元的理解存在偏差。我仍将之理解为和单周期CPU的控制单元功能实现相仿，实

际不然。为了实现多周期CPU，控制单元加入了CLK和RST，内部分成了三个小单元，这是完全不一样的。

我开始重新考虑控制单元的实现，最后采用模块化的设计并采用了更简便的实现方法（具体见分析部分）。

当然思路正确了仍然少不了出错debug的过程，由于多周期需要综合考虑不同指令，还存在延时问题，单单通过看波形很难查错。更为重要的是要分析逻辑，深刻理解指令执行过程即相应变量的变化，再检查代码。这个过程让我感受到多周期CPU设计的难点在于对切分数据通路，加入寄存器将大延迟变为小延迟的理解和运用。

单周期CPU设计实验写代码的过程总体来说比较顺利，可以举一反三，写实验报告总结归纳的过程比较艰难，在一点点写报告的过程中才真正理解了实验的精髓。这次多周期CPU设计实验却反之，写代码的过程比较艰难，多次在bug中不知所措，但真正解决了开始写报告时就很顺利了，知识已经在前面的过程中吸收理解。

理论和实验是相辅相成的。做实验可以深入理论的细节、验证理论，学理论可以知其所以然、剖析实验。从任一方开始进行摸索，同时结合另一方是最好的学习方式。

也可以看到，多周期CPU的设计方式多样，比如子模块的划分方法、接口的定义、子模块的实现都不尽相同，对应的代码量、逻辑强度、性能也各有千秋，需要在多次实验中对比学习、多思考、多总结，找到更规范、更科学、更适合自己的方法。