中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

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：	曹广杰
学　　号：	15352015
专业（班级）：	15 移动教学（1）班
时　　间：	2017　年　5　月　30　日

成 绩 ：

实验二 ： 多周期CPU设计基本实验

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

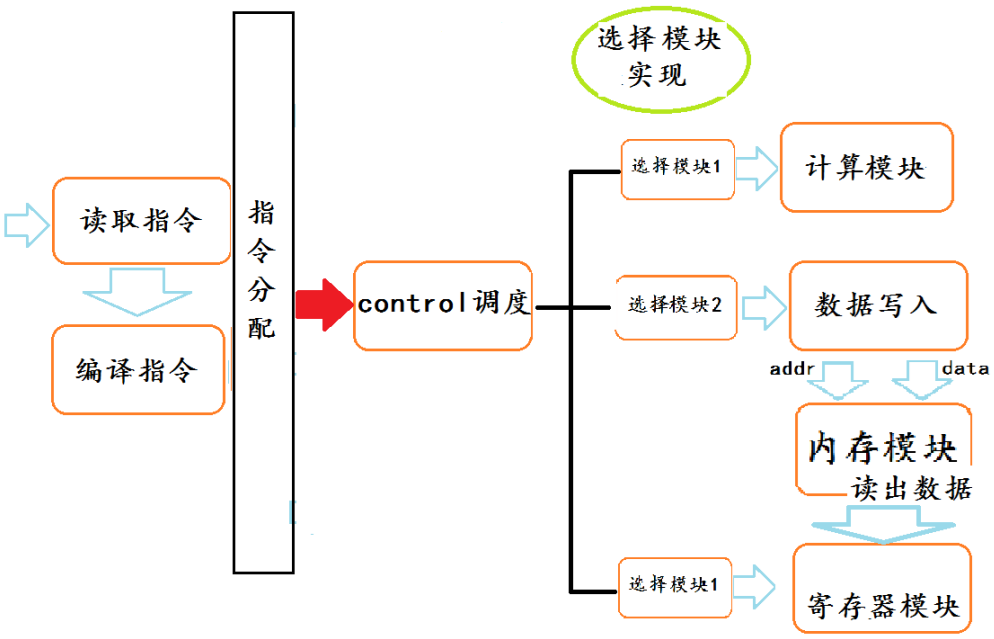
1. 设计一个多周期 CPU，该 CPU 至少能实现算数运算指令、逻辑运算指令、储存器读写指令、分支指令以及停机指令；
2. 寄存器各个组件和核心控制系统使用时钟触发；
3. 指令存储器和数据存储器存储单元宽度一律使用8位，即一个字节的存储单位。不能使用32位作为存储器存储单元宽度；
4. 控制器部分可以考虑用控制信号真值表方法（有共性部分）与用 case 语句方法逐个产生各指令其它控制信号相配合，注意：信号必须与状态配合。当然，还可以用其它方法，自己考虑；
5. 必须写一段测试用的汇编程序，而且必须包含所要求的所有指令。slt、sltu 指令必须检查两种情况：“小于”和“大于等于”；beq、bne 指令必须检查两种情况：“等”和“不 等”。这段汇编程序必须尽量优化且出现在实验报告中，同时，给出每条指令在内存中的地 址。检查实验时，必须提供。
6. 实验报告中应该有每条指令执行的波形（截图），用于说明该指令的正确性；

三. 实验原理

设计控制模块接收来自于计算机上的硬件指令并编译指令,继而向各个功能模块发出调控指令，实现计算机上txt文件的二进制指令行。目前做的多周期CPU运行机制是在同一个周期内同时运行多个指令。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。

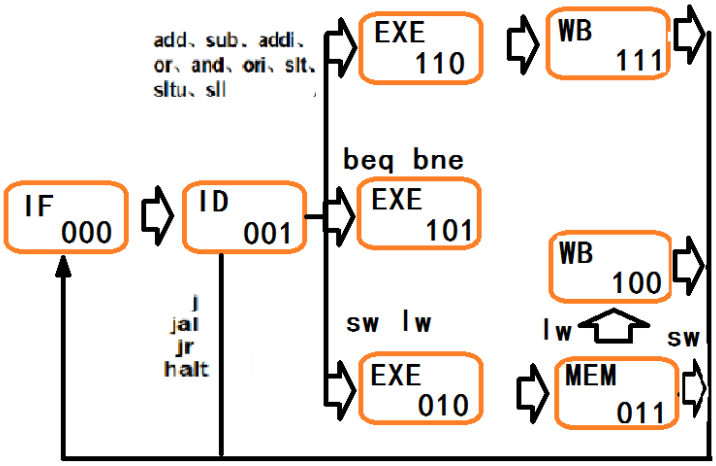
本次实验的设计基础（即实验提出的统一要求）包括输入的指令、操作译码的寄存器以及整个CPU设计的处理框架。

一、CPU的处理框架：



上图为本次多周期设计的总体框架，多周期的框架设计与单周期很类似，比如计算单元的实现以及指令的读取——这些操作与原来的操作都是完全一致的。但是，当涉及到整体CPU对于时钟信息的利用时，二者的差距就拉开了。由上图显示，本次的多周期CPU设计有很大的一个部分就是对于寄存器组件的设计以及使用。由于寄存器的存在，一个周期内同时运行多条语句成为可能。

此外，有一个非常重要的点就是对于始终信号的处理。既然要在同一个周期内部执行多条指令，势必可能引发混乱，因此，除了对于寄存器的高要求之外，各个运算单元以及寄存器单元的协调与统一也是至关重要的。所以，对于核心处理结构需要对时钟处理同时输出信号同步各个组件的信息，以此实现组件之间的协调工作。



本次实现，笔者根据要求将核心控制模块分解为三个子模块，以实现特定周期中不同阶段操作的跳变；

1. 读取指令：

   指令储存在txt文件中，通过Verilog代码readmem函数，读取txt文件中的代码，继而储存在指令寄存器中。顺便提及，储存的信号统一以字节存储，方便之后的指令随着PC地址的变化而变化，从而使CPU执行不同的指令。这里引入PC地址的概念，PC地址即在指令文件中默认分配的地址。PC地址的变动会引起操作的变化；

2. 指令寄存器：

   读取指令之后，需要在时钟信号的指引下将得到的指令信号由上升沿触发，以此实现对于信号的定时输出，以完成多周期设计的第一步；

3. 指令译码：

   某一条指令包含了一个操作需要的全部信息，但是对于不同的模块而言，需要的信息是不同的，有时候需要的信息即使是二进制数字也与指令中的排列顺序不同，这里需要设计一个编译模块，根据不同功能模块的需求将指令中的代码转化为功能模块可以直接使用的数据。这里，control unit完成了对于指令的OP位置的6位的解码。指令译码的操作涉及核心控制单元以及所有涉及到寄存器地址的位置，32位二进制码都可以直接使用；

4. 指令执行：

   Control unit模块对OP指令译码之后，将复杂的数据转化为各个功能模块已有的模式的一种选择。根据OP指令的不同，针对各个功能模块输入特定的数据，使得功能模块可以按照指令运作；

5. 数据选择：

在接收到不同的操作信息之后，功能模块会从原始指令中选择一些数据用于运算。有时候不同种类的数据之间的操作方式是一致的，这里设计选择模块，可以根据不同的情况对数据进行不同的选择，并将选择后的数据传入用于实现操作的模块中。有效地避免了因为不同种类数据之间的组合而使得运算模块增加的情况，而且避免了各个运算模块之间的操作重复，减少代码量与电路复杂性。本次实验中的数据选择模块统一由ControlUnit模块输出的信号调节控制，数据选择模块与时钟之间不再有任何关系——由于统一调控的存在，实现协调性与统一性的共存；

6. 数据计算：

根据输入的数据以及有control unit返回的模式选择变量对输入的数据进行不同的运算。并将计算结果输出，输出的结果可能被用于储存在储存器中，也可能被用于写入寄存器中以便于下一次运算。本次运算的基础结构还是基于上一次实现，即单周期CPU设计的实验设计，多数的计算语句也都是已经在Verilog语言中存在的运算结构，符号之类的操作直接调用即可。有所更新的是，本次实验添加了sll（移位语句），以及有符号比较语句，这些语句在Verilog中实现起来会有一点困难，不过也就是多一些条件选择语句，不会影响本次实现中ALU的整体格局；

7. 读写操作：

读写操作的本质是根据输入的地址，返回或者写入相应的数值。本次实验中需要读写操作的结构为寄存器和储存器。这一点与之前的单周期CPU完全一致，不需要任何的更改；

二、输入指令：

输入指令格式固定，分为3种R类型、I类型以及J类型：

R类型（统一为32位）：

指令OP	寄存器rs	寄存器rt	寄存器rd	位移量sa	保留码fu
6	5	5	5	5	6

此种指令用于寄存器数值之间的读取以及运算，指令中的5位二进制数字表示寄存器储存的地址，该指令被指令译码读入之后，指令译码会根据地址读取相应的数字根据OP指令在运算模块中运算。本次实验中除了对于寄存器rs、rt、rd的使用，这种使用在ALU计算的过程中表现尤甚。还添加了对于sa变量的使用，与之对应的算术操作是sll，移位操作，具体实现比较简单，在ALU内部调用移位运算符即可。

I类型（统一为32位）：

指令OP	寄存器rs	寄存器rt	立即数imm
6	5	5	16

I类型的指令主要用于处理寄存器数值与立即数之间的运算，依然由指令译码模块对指令进行译码，读出数据与立即数根据OP指令在运算模块中运算。这种指令在通常的算术操作中比较常见，主要内容还是rs与立即数相加，结果储存在rt寄存器内。

J类型（统一为32位）：

指令OP	地址addr
6	26

J型指令主要用于跳转，根据给出的地址，进行PC地址的运算并将下一个PC值直接赋值为该地址处的位置，本次实验中与之相关的指令是j、jr以及jal，主要的用途就是对于指定地址的跳转操作。

寄存单元操作译码：

所有的寄存器或者储存器在本次实验中都被要求使用8位的储存单元,8位的储存单元储存32位的指令或者数据就需要分割为4个字节，并声明更大的集合，以确定能够储存我们需要计算的数据或者地址；

四. 实验器材

电脑一台、Xilinx ISE 软件一套。

五. 实验分析与设计

1. 实验分析：

1)设计多周期CPU，笔者还是在单周期CPU的基础上进行修改。本次实验的目的在于设计一个多周期CPU，能够执行要求中所给出的命令行语句的功能。汇编语句中最主要的成分是计算与储存，所以主要的计算实现模块也应该是计算（由算数语句推理得知）和储存结构，要求使用8位的储存方式储存为多个字节。本次实验中需要考虑到储存方式的结构为寄存器（由运算语句推理得）与储存器（由写入语句推理得）。考虑到以上几点，得到本次实验的实现需要从ALU算数单元、寄存器与储存器入手；

2)；

2. 实验设计：

本次实验的设计顺序如下：

设计顺序

从设计MCPU的角度考虑，从ALU设计最容易入手。本次实验中，在寄存器的设计上也是非常清晰明朗的，但是寄存器终究只能作为在计算运转过程中的一个工具，尤其编写的过程中，会显得编写的目的很不明朗。而在由简单到复杂的过程中，一个目的明确的开发过程对于持续性强的开发是至关重要的，鉴于此，笔者还是从ALU计算单元入手。本次实验中，计算单元添加了两个语句sll（移位计算语句）和（bne不等式判决语句）。

1)sll语句：

在sll语句的设计中，传入参量与单周期CPU的传入参量有所改变，笔者将会在ALU中添加一个运算模式，该运算模式采用输出数据为一个数据度另一个数据移位的操作。强调:ALU计算单元仅仅用于计算，这种操作是笔者在单周期CPU的设计过程中就已经强调过的了。至于对sa数据的导入，笔者这里还是使用选择器以实现该操作。

2)bne判决：

bne判决语句与beq判决语句何其相同，二者的操作几乎完全类似，如果有什么不同就是对于输出数据的操作，一个是被比较两者相同时为1，另一个则是不相同时为1，这种语句的增加在CPU设计中添加一个取非符号即可；

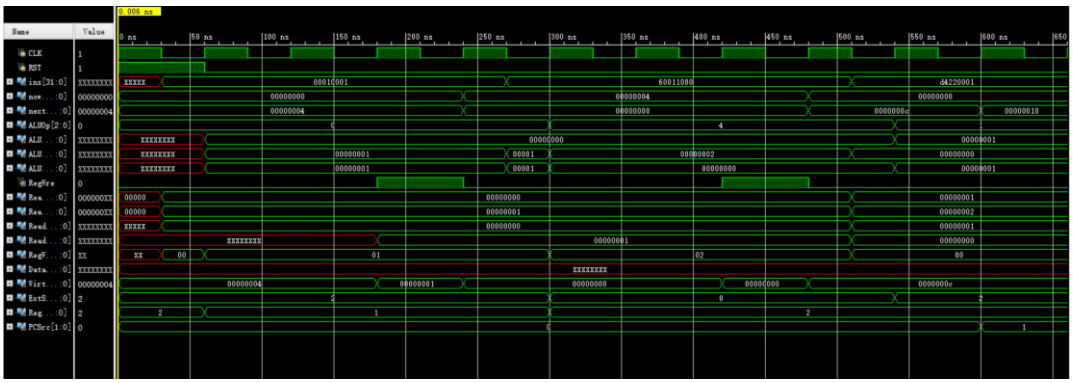
ALU设计之后。ALU设计之后，就可以直接面对多周期的根本设计了。多周期之所以为多周期，就是因为设计中对于寄存器多个组件的设计以及使用——由于诸多寄存器的存在，一个周期内同时运行多条语句成为可能。所以此时可以先设计ControlUnit模块，如此先构建一个MCPU的框架，再在内部添加寄存器模块，使得整个框架可以正常运作，严丝合缝。但是笔者对于vivado的报错机制实在不敢恭维，所以这里选择了先设计寄存器与选择器模块。在此之前，笔者先在单周期的框架上，完成对于已有的计算语句的测试，确保所有计算语句以及跳转语句运作正常无误之后，才开始进一步的设计。

寄存器与选择器的设计。此处笔者所指出的寄存器并非寄存器组，而是小单元的寄存器模块。由于实验要求中已经给出寄存器模块的所有出入口，笔者只需要拟定存在的控制信号，

并在控制信号的基础上实现已有的信号的输出即可。

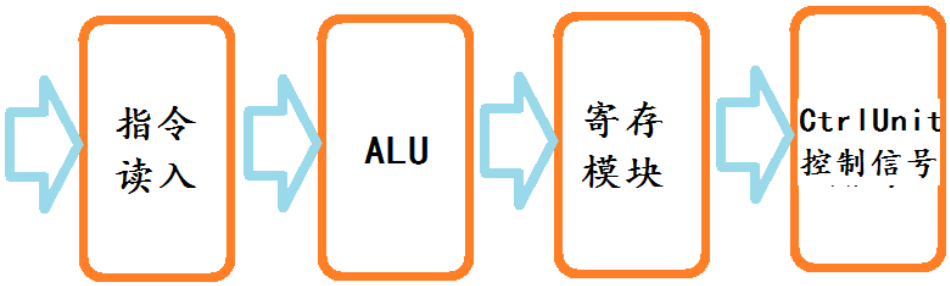
ControlUnit模块的设计。；

3. 实验结果分析：



首先，绿色显示该测试中所有的数据都有非常明确的数据值，并且这些数据值或用于显示输出结果或用于检查错误位置，都是有意义的信息，不存在XXXX类型的情况，是测试合理的第一步；

ALU运算单元的巨大价值。接下来考虑实验结果中真正重要的是什么，除了跳变语句的运行，其他语句都需要ALU运算单元的运算，更甚者，ALU的运算几乎是所有的寄存器与选择器的晴雨表——原因很简单，在一条语句中，最少使用4个寄存器与3个选择器。而在运行多个连续的计算语句夹杂着写入语句的时候,这种对于寄存器与选择器的大幅度占用是非常可观的。笔者在检测的过程中将寄存器的写入语句与计算语句多次混合交叉排入，这里我们考虑一下我们之前设计CPU的过程，由于选择器是依据ALU而生的，在多次运算中，尤其是设计好的运算中，选择器哪怕有一点点的失误，计算结果与写入结果都会有不合逻辑之处。与之对应，如果ALU的计算单元如果计算无失误，就已经意味着选择器是没有问题的，如果出现问题，再使用回溯法查找选择器的问题。以此类推我们得到我们检查结果时候最重要的几点——寄存器、ALU以及指令读入系统（这纯粹是为了标明我们检查的是哪一条语句，当然，检查数据读入是否顺利也是非常重要的，故而此处将其作为一个非常重要的检查指标吧,由于数据写入模块笔者使用的是单周期的代码，此处就免于检查），因为以上几个模块都是相互独立的，是每一个阶段工作的里程碑：



以此顺序，检查整个多周期CPU的设计是否符合笔者的预期。

检查指令读入模块：

在看到前两个指令分别为0与4的时候，不出意外的，整个mcpu的指令读入系统就已经是符合预期的了。因为对于指令读入而言，所有的指令都是同一种操作，无差别。而我们通过图中数据可知，curPC以及nextPC都符合当前值，这说明指令读取成功了。

观察笔者添加的前两条运算指令：

PC的值		测试指令	结果	op(6)
0x00000000	addiu	rt = rs+ 2	rt = 2 ($1 = 2)	000010
0x00000004	sll	rd = rt << sa	rd = 4 ($2 = 4)	011000
0x00000008	bne	comp	rs != rt pc+=1(to 10	110101
0x0000000c	ori	rt = rs | 10	rt = 10	010010
0x00000010	sltu	comp	rs < rt (s3 = 1)	100111
0x00000014	beq	$1, $2, 3 (nto14)	rs==rt pc += 1(to 1c	110100
0x00000018	addu	rd = rs + rt	rd = 18	000000

另外，ins的内容与笔者所有二进制数字的汇编指令转化为16进制后的数值也是符合的：

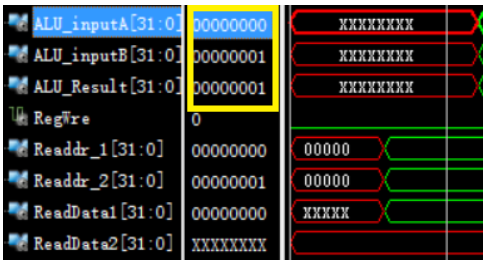
测试汇编指令（16进制）
0801 0001
6001 1080

都是0801 0001，至此可知，笔者的读入模块已经经受得起检查了。因为对于多个指令而言，读入的操作都是一致的。

由此我们得出结论：读入模块可信。

检查寄存模块：

接下来，就是对于寄存模块的讨论，第一条语句意思很明显，就是把0+1的数值储存到ALU内部，那么输入的数值也应该是0和1，而我们发现ALU的input内容刚好符合，另外，看到读入寄存器的地址也是如此，分别是0和1，立即数是1：



这些都符合汇编指令的储存数据，现在可以说明，ALU之前的寄存器与选择器数据的读出是没有问题的，接下来检测ALU模块之后的计算语句以确定之后的寄存器与选择器读出数据也是符合预期的。

但是读入模块还没有测试到，因为读入寄存器可能来自于ALU的运算，也可能来自于内存数据的读取。

得出结论：**寄存模块是可信的。**并且在ALU之前的寄存模块与选择模块都是可信的。（这需要对于所有已知的信息进行统一的检查，对于相同的运算不必进行如此繁杂的检查）。

检查ALU模块：

笔者根据前两次的实验结果，得出结论，在加法运算与移位运算中，ALU模块表现稳定。因此，笔者断定，在已知的传入信号准确的情况下，ALU模块可以计算得到正确的答案，至于ControlUnit模块信号的准确度则需要根据最后对于计算单元与寄存器单元的综合考量得出结果——这里需要注意到，ControlUnit模块由于运算冗杂，非常有可能出现意想不到的小错误，但是如果在不影响计算单元与寄存器单元的前提下，这种错误是可以忽略的——因为我们认为其没有造成实际的影响。

所以我们认为：在输入控制信号准确的前提下，ALU运算单元是可信的。而对输入信号的检查需要在后续的检测中予以验证；

检查数据储存模块。数据储存模块的读入包括读入数据与读入地址，因为读入数据的地址是确定的，始终是rt寄存器，笔者这里就没有添加测试信息，因为在这个环节没有寄

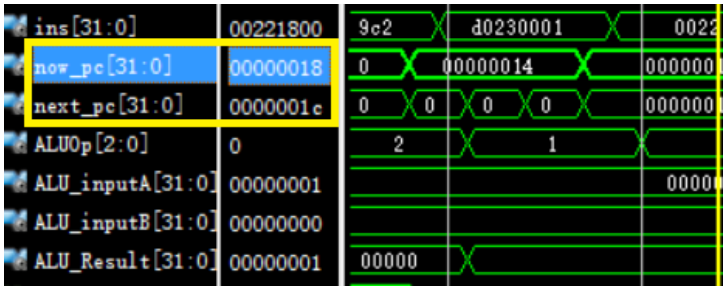
存器，测试意义不大。在这个过程中尚且没有写入内存的必要，故而不知道数据储存模块的可信度如何。但是根据总图可以看出，输入内存的数据是一直在接口处存在的，但并没有执行读入操作，因为内存的输出信号一直都是不确定的，只有在sw语句执行的时候，该信号才会从红色转化为绿色。于是得出结论，内存的写入能力没有问题，且相对应的选择器系统也没有问题。

其他计算操作的检查：

0x00000010	sltu	comp
0x00000014	beq	$1, $2, 3 (nto14)
0x00000018	addu	rd = rs + rt
0x0000001c	subu	rd = rs - rt
0x00000020	jal	
0x00000024	slt	comp

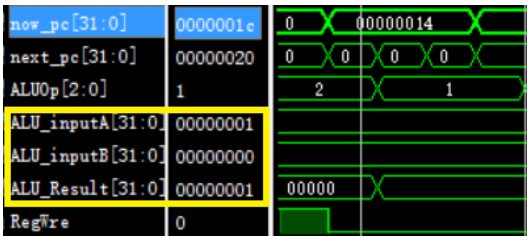
两条语句，加减——分别是第18与第1c处。

检查ALU运作是否正常：



在pc为18处，所做的依然是1+0=1的操作，而ALU的运算在加法上并没有出现纰漏。

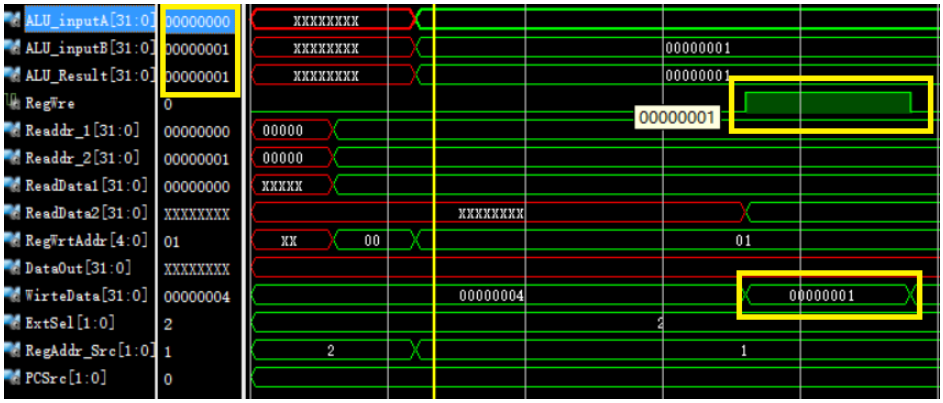
检查减法：



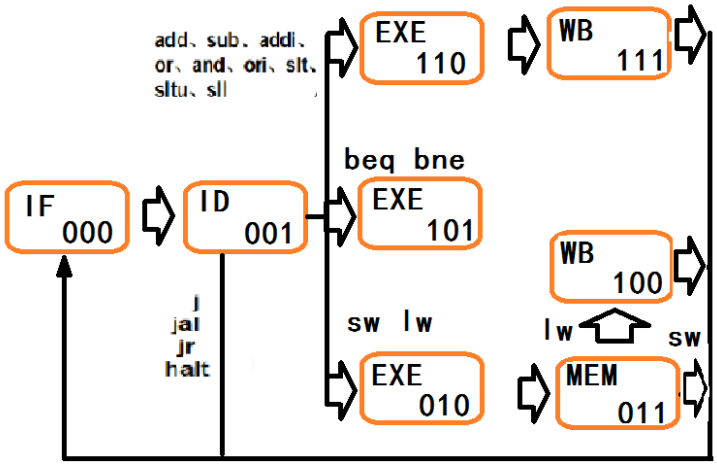
此处，传入的寄存器内部数值为1，期望为1；

至此，**考虑寄存器组模块与跳变信号的准确度：**

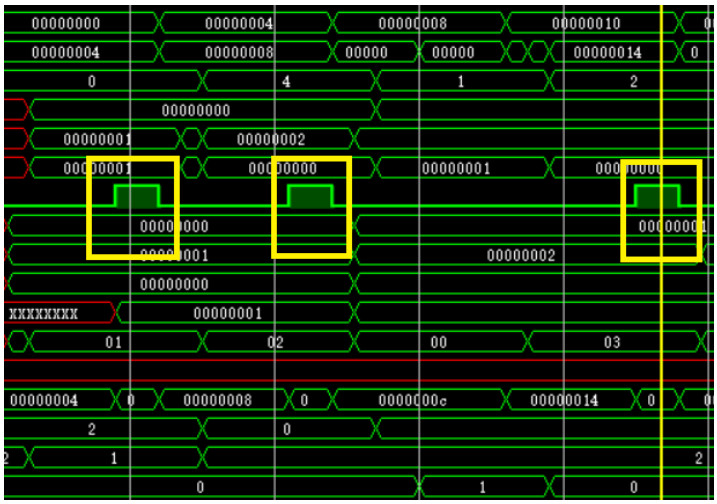
首先，对于寄存器组模块。寄存器组模块有控制信号，来源于ControlUnit模块，在需要寄存器组写入的时候，该信号会显示为1，否则为0：



可以看到，上图右侧的信号为1，此处正是信号应该写入的时刻，这种时刻只有在特定的时钟周期才可以触发，比如说计算过一次ALU的结果，获得了准确的数据之后，会根据传入指令的前6位转移到写入指令的操作中。此后便是需要使得ALU算数单元可以获得准确的信息，输出的信息也储存到相应的位置上，设置相应的地址信息。涉及到地址信息与数值信息之间的联系，并且多次计算中需要调取之前的数据。而调用RegWre的时刻，在ControlUnit模块中则显示为一个特定的部分，这个部分由时钟信号与指令信号同时影响以至于跳变。其实这三种状态也是非常清晰的：



即111、100与001的位置，当且仅当以上三个位置的时候，写入信号才会触发，那么笔者就去检查已经调用以上三个指令的位置，观察信息是否符合预期：

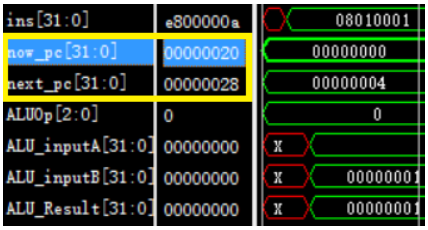


可以看到，在以上的三个位置，该信号中的高电平信息都非常准时地出现在了特定的时钟周期内。至此，可以知道，寄存器组的设计中对于控制信号的处理是符合预期的。而由于其他的主要操作与单周期的设计一致，笔者此处默认为是合理的；

跳变信号的检查：

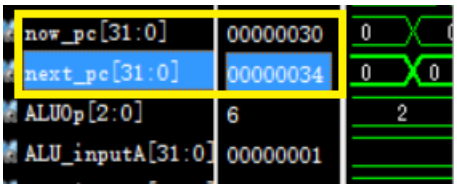
0x00000020	jal		to 28	111010
0x00000024	slt	comp	rs < rt (s3 = 1)	100110
0x00000028	addiu	rt = rs + 30	rt = 34;($6 = 34)	000010
0x0000002c	and	rd = rs & rt	rd = 0 ($4 = 0)	010001
0x00000030	jr	to rs($6)	to 34	111001
0x00000034	or	rd = rs | rt	rd = 6	010000
0x00000038	sw	$2,5($1)	mem[s1+5]=s2=10	110000
0x0000003c	lw	$1,5($1)	s1=mem[s1+5]=10	110001
0x00000040	j		to the end	111000
0x00000044	halt			111111

在笔者规定的语句中，第20、 30以及第40条语句都是用于测试跳变函数的，以上三者的跳变处理，可以通过对下一点的pc值的观测得出：



可以看到，在当前PC为20时，下一条PC值是28，而笔者的预期正是28。此处，由于笔者的设置择址跳变，ALU不参与运算，所以可以看到上图中的ALU三者都处于闲置状态。ALU的模式也处于默认值状态。

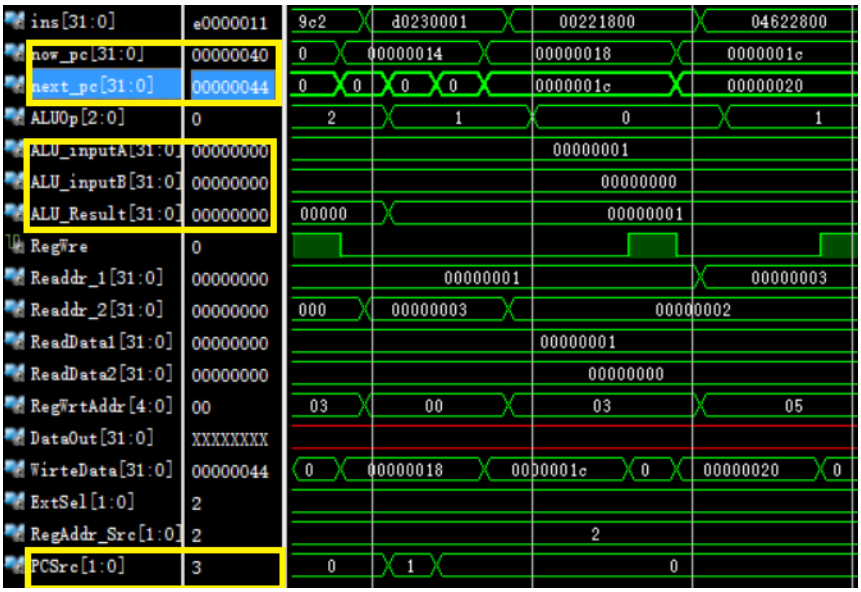
继而检查第30条语句：



第30条语句也涉及了PC地址的跳转，这里由于笔者限定了测试的语句数目，所以为了使用更少的语句测试更多的内容，笔者要求其跳转到下一个位置，虽然，可能不太明显。

检查第40条语句：

笔者设计第40条语句，意在要求其直接跳转到最后一条指令，即停机指令。这时候，最明显的现象就是停机操作，虽然也是临近PC的跳转，但是我们可以根据Pcwre的信号确定跳转操作是否进行，而且ALU设计也可以作为辅证：



此处包含3个信息，PCSrc不是在1或者2状态，这说明PC跳变不是属于正常跳变，而ALU运算单元也处于闲置状态，这一点辅证了j的运作是符合预期的。其次这就是看next-pc，其值是确实符合预期的。

综上，在跳变语句的实现中，CPU设计符合预期。；

六. 实验心得

多周期的框架设计与单周期很类似，比如计算单元的实现以及指令的读取——这些操作与原来的操作都是完全一致的。但是，当涉及到整体CPU对于时钟信息的利用时，二者的差距就拉开了。本次的多周期CPU设计有很大的一个部分就是对于寄存器组件的设计以及使用。由于寄存器的存在，一个周期内同时运行多条语句成为可能。

时钟信号的处理是本次实验的灵魂。既然要在同一个周期内部执行多条指令，势必可能

引发混乱，因此，除了对于寄存器的高要求之外，各个运算单元以及寄存器单元的协调与统一也是至关重要的。所以，对于核心处理结构需要对时钟处理同时输出信号同步各个组件的信息，以此实现组件之间的协调工作。

个人觉得有几点是非常重要的：

a) 多周期CPU的设计主干。多周期CPU在设计的过程中，对于时钟信号的掌控理应是本次实验的主干，时钟信号牵扯出核心控制模块内部的状态转移操作，每一个时钟周期又有不同的状态转移行为，不同的行为控制着信号向整个CPU的所有组件中发射信号，使得整个CPU计算出合理的结果。毫无疑问，由一个时钟信号控制的CPU可以很容易地实现各个位置的同步处理。反思单周期CPU，其实使用这种操作方式也未尝不可，在更少的时钟信号的基础上，状态的操作转移可以在很大的程度上减轻设计的复杂度与工作量；

b) 寄存器组件与多周期的形成。何为多周期，笔者一开始设计的时候最疑惑的就是这个过程。笔者于是又计算单元ALU开始，使用迭代的方法由混沌迷糊的状态一点点地实现迭代开发。这个问题一直到最后的寄存器组设计的时候才恍然大悟。多周期的形成与时钟周期联系紧密，一个时钟周期的过程同时运行多条指令，那么CPU的吞吐量必然会增大，但是与此同时的，减少了时间消耗，同样的就需要使用空间代价来补偿。这种代价最直观的就是各个小型的寄存器的形成。多个小寄存器的设计都有时钟触发的因素，这可以说是一种特权，即便是PC组件尚且受control组件的调控，寄存器直接受命于clk，就是因为寄存器需要作为指令运行的里程碑,上升沿触发,则该指令只能在寄存器这里停留，产生的另一个影响就是其他指令运作的时候，由于数据的保存，保证了所有的组件都能够正常运作，而且每一个组件都处在忙碌状态，充分地利用了计算资源；

c) Vivado软件报错的诸多错误机制。笔者在一次测试的过程中，使用核心控制模块对某一个信号量做了赋值为1的处理，但是最后的波形图显现却不尽如人意，该波形始终为0，笔者后续花了很大的力气才发现原来是某处数据转化不对，由5线的数据转化为了32线的数据，但是这与那个信号毫无关系，但是最后的模拟操作显示的错误就是如此。这种报错机制使得检验过程非常困难，这种设计可以说是非常的不人性化甚至是弱智。数据量之间的关联处理无效化大大增大了开发的难度；

d) 导师给的错误的参考信息。笔者在阅读老师给出的参考信息的时候，发现了几处非常明显的矛盾，毫无疑问，老师是完全有能力发现这种问题的，只是恐怕老师对于给出的材料甚至都没有看过。比如PC模块的设计，PC模块的设计本质是由核心控制单元操作的，但是老师却一意孤行地要求使用clk实现触发，但是这实际上并不是必须的，甚至有可

能出问题，使用核心控制单元可以很容易地实现协调。笔者不得已添加了clk参数传入，然而内部实现与clk并无多大关联，只是希望任务发布者能够更加负责更加切实，不要沉迷于形式主义与个人存在感的幻想中而过多地浪费笔者的时间；

e) 读入数据与指令跳转的不同步性。读入的指令数据经过转化输出的操作，受到两个关键因素的制约，一个是初始化延时，另一个是寄存器延时。这两个因素的存在使得读入数据的延时是必须的。因此，无论最后我们设计的时候，指令更新是上升沿触发还是下降沿触发都是一样要浪费一点时间，最起码半个时钟周期。这是不可消除不可省略的半个周期；

f) 开发模式的不同导致思维结构不同。在实验分析的过程中，笔者对于实验的那几条语句不是非常理解，笔者的第一反应就是这几条指令通过C语言是非常容易实现的，但是对设计CPU时候与之有什么不同并未察觉到，当时只意识到Verilog语言与C语言何其相似。事后反省过来才意识到开发模式根本不同；

g) Debug的资料搜索。百度上对于vivado的开发内容实在不敢恭维，各个博客互相抄袭，真正有用的东西凤毛麟角。笔者不得不自己看报错信息，并且询问同学，互通有无。这实在是比较低效的过程，不过信息的交流当然是非常重要啦；

h) 信息的整合。为了进一步分析实验，笔者收集了老师提供的所有资料并进行合并，最后梳理出的文件只有一个代码合集、要求合集的text文档以及测试方法的PDF文档，在保留压缩包后，对于多余的原始文档一次删除，思路清晰了很多，最起码明白应该先从ALU模块入手。ALU模块实现使得与之相类似的寄存器模块和储存器模块都有了入手的基础。所以没有这一点，此后的工作简直寸步难行；

i) 最后检查项目的实现情况也是需要仔细理解题目的。笔者亲眼看到自己的同学对着指令表一行一行地对照核实，实在是非常浪费时间的方式，尤其是对于选择器的信号检查，这是最繁琐最耗时最低效的检查项目之一，但是该环节与其他环节之间联系紧密。免除选择器的检查环节可以大大简化整个项目后续实现过程中的工作量；

体会和建议

本学期的课程从汇编语言开始，继而是单周期与多周期CPU设计。这些实验都是在计算机的基础角度设计的，笔者从以上实验中清晰地了解了计算机基层语言的设计方法以及组成原理。

笔者在这一学期的学习中了解到的计算机组成的基础原理使得笔者在对于计算机乃至

人类的当前科技的发展有了更加清晰的认识。从汇编语言开始，汇编语言的设计的特点就是每一个位置每一个语句都需要考虑储存空间的因素，这一点比C语言有过之而无不及。如果C语言是高级语言编程的鼻祖，那么C语言尚且没有涉及到计算机储存的物理地址，没有涉及到计算机内部的筋骨与脏器。但是汇编语言就切实地触及到了计算机的根本，并且时刻游弋与计算机的灵魂。此后设计的CPU实验，使得这种复杂的设计基础在Verilog的基础上得到了综合与升华。如果说，这就是诸多人类科技的发展方向，是将来足以带领人类走向一个新的纪元，那么是基于以上的逻辑单元的设计操作，恐怕不足以带领人类领略一个新的时空。毫无疑问，这种设计的基础过于复杂，人类涉及到的领域可以使用其进行代劳，但是除此之外，更多的事情，仅仅是基于以上的二进制的操作组合恐怕不足以完成目前来看对于未来的美好设想。原因很简单，由于这种设计单元的单一性，以及单元之间组合的复杂性，在人类将来面对更多的甚至更加抽象的模糊的问题时，一个模型的构建就越发困难，而与此直接对应的，就是组合体系的逐渐庞大与人类计算算量的极限。人类的这种设计终究会在更加庞大复杂的模型中捉襟见肘，未来的更新，不再是发展这一种简单逻辑单元的复杂度组合。渺茫的希望，将会寄予新的材料的开发，使人类摆脱当前的认知受限的情况。而更大的潜在，应该在我们不能理解的量子领域，在那里实现理解与数学的纠缠，实现模糊与清晰的飞扬。