流水线 Mips 处理器设计

实验报告

姓　　名：邵晨扬

班　　级：无 04

学　　号：2020010733

完成时间：2022/07/11

目录

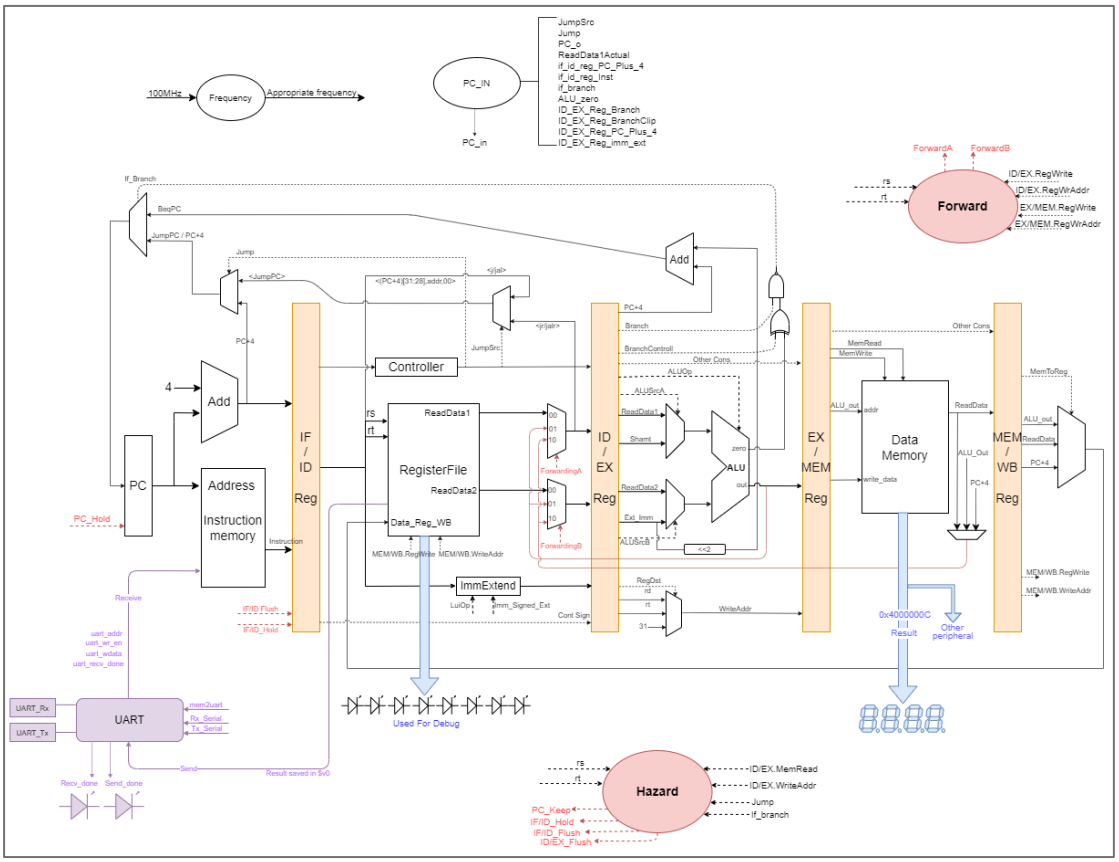
一、实验内容

将春季学期实验四设计的多周期 MIPS 处理器改进为流水线结构，并利用此处理器完成字符串搜索算法，并使用软件译码方式，将匹配结果结果输出到数码管上。

二、具体设计方案：

1、总体设计：

实验整体的数据通路图如下：（这是在写代码之前使用的数据通路，具体写代码的时候可能有些差别，但是总体都一致。）



具体每个模块的实现以及细节处都将在下面详细讲解。

2、支持指令：

R 型	Add、addu、sub、subu、and、or、xor、nor、slt、sltu、sll、srl、sra
I 型算术指令	Lui、addi、addiu、andi、sltiu、ori
分支指令	Beq、bne、blez、bgtz、bltz
跳转指令	J、jal、jr、jalr
存储访问指令	Lw、sw、Lb

因为 Mips 编辑器译码得到的指令中存在大量的 ori 和 lb 指令，前者是由于当立即数比较大的时候（16 位的立即数已经不足以储存的时候），Mips Editor 就会使用 lui 和 ori 指令结合以产生较大的数字，后者是因为所使用的字符串搜索

算法是按照 Byte 为基本单位进行读取和比较的，而非按照 word 比较，所以我增加了对这两条指令的支持。Ori 指令的支持很容易，只是一条很普通的 I 型指令。而 Lb 指令的支持将在后面具体阐述。

3、IF 阶段：

IF 阶段是第一个阶段。在 IF 阶段需要完成的功能（或者说操作）有如下 3 个：

1、以 PC 为址从 Instruction Memory 中读取指令。

2、计算$PC+4$，后面很多环节都会用到$PC+4$的值。

3、需要计算出 PC 写入端PC_In的值。这一块也是比较复杂的，因为需要考虑分支、跳转的情况，接线也比较复杂。所以在最终的代码中，我把PC_In做成了一个单独的模块，以使主程序结构清晰。

4、ID 阶段及数据转发：

ID 阶段，流水线第二级，需要完成如下 4 个操作：

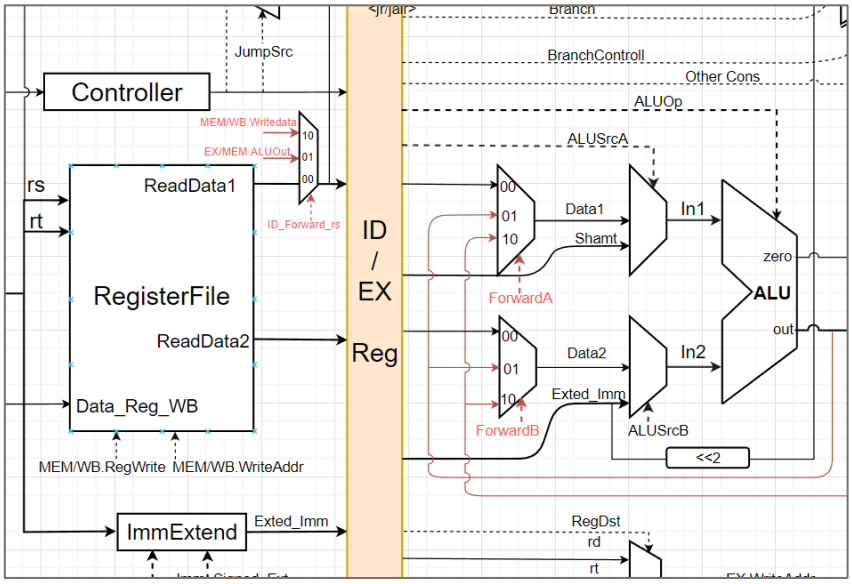
1、寄存器堆读取。

2、控制信号生成。

3、立即数扩展。

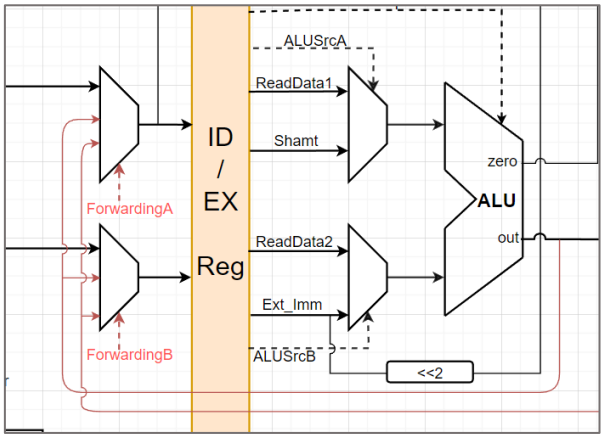
4、数据转发。

本来我以为 ID 阶段是不用转发的，在数逻课上也确实没有涉及到 ID 阶段的转发，但是写的时候我才发现其实是需要的。比如 jump 指令是在 ID 阶段进行判断，这也就意味着 jr 与 jalr 指令在 ID 阶段就会用到寄存器，这就可能产生数据冒险，所以我在 ID 阶段也添加一个转发的功能。所以一开始，我有 ID Forwarding 和 EX Forwarding 两个转发模块，就像下图这样：



但是写完的时候我就思考，是否需要两个转发单元？因为这两个转发单元实际上是连着的，一个是在 ID 阶段后期，一个是在 EX 阶段初期，这两个阶段的数据并不会有什么区别（需要算的还没算，需要读的还没读，所以转发的数据不会变，而 ID/EX 寄存器也只是起到了传递的作用，所以寄存器读的数据也不会变），那么何不把这两个转发单元合并到一块去？考虑到 ALU 的计算过程可能挺需要耗费时间，所以我决定把转发都合并到 ID 阶段末期。也就变成了下图这

样：

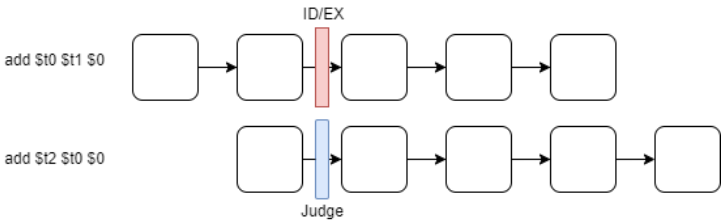


经过测试，这样也是行得通的。具体的判断方式如下：

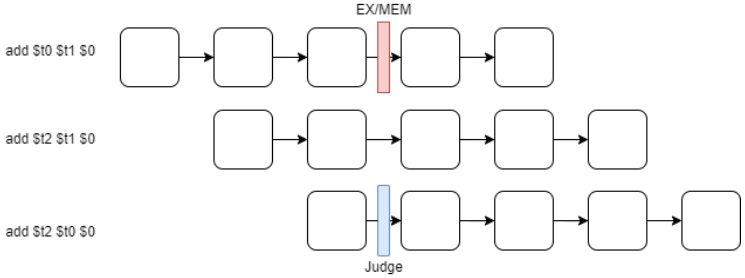
```
assign ForwardA = (ID_EX_RegWrite && ID_EX_WriteAddr != 0 && ID_EX_WriteAddr == rs) ? 2'b01
            : (EX_MEM_RegWrite && EX_MEM_WriteAddr != 0 && EX_MEM_WriteAddr == rs) ? 2'b10
            : 2'b00;

assign ForwardB = (ID_EX_RegWrite && ID_EX_WriteAddr != 0 && ID_EX_WriteAddr == rt) ? 2'b01
            : (EX_MEM_RegWrite && EX_MEM_WriteAddr != 0 && EX_MEM_WriteAddr == rt) ? 2'b10
            : 2'b00;
```

当 Forward 信号是 00 的时候，就代表没有数据冒险，就选择寄存器堆读取的数据，当 Forward 信号是 01 的时候，就转发 ALU 计算的数据；当寄存器的信号是 10 的时候，就转发 MEM 阶段的数据（这又是三选一：DM 中读的数据，PC+4，以及 ALU_Out）。

下面来解释转发的正确性：比如下面是一个典型的数据冒险，在第二条指令的 ID 阶段，ID/EX 寄存器中储存的仍然是第一条指令的参数，所以判断第一条指令是否有写寄存器的行为，是否写到 0 寄存器中去，写入的寄存器是不是现在第二条指令需要读取的那一个，如果都满足，就转发第一条指令的 ALU 的计算结果。



再举例说明 MEM 阶段的转发过程：



第三条指令需要转发第一条指令的数据。在第三条指令的 ID 阶段的后期，

EX/MEM 寄存器中还存储的是第一条指令的数据。所以这时候检测 EX/MEM 中还存留的寄存器写入信号是否为 1，寄存器写入地址是否为 0 以及是否是当前要读的寄存器，就可以正确把第一条指令的 MEM 阶段的数据转发过来。这个转发的数据需要先经过一个 3 选一的 MUX，代码如下：

```
assign dataForward = EX_MEM_Reg_MemtoReg == 2'b00 ? EX_MEM_Reg_ALU_out
                   : EX_MEM_Reg_MemtoReg == 2'b01 ? DataMemReadData
                   : EX_MEM_Reg_PC_Plus_4;
```

如上，就可以使用一个 ID 阶段的 Forwarding 操作来正确处理数据的转发。

5、EX 阶段：

EX 阶段，流水线第三级，需要完成如下 4 个操作：
1、ALU 操作数选择。
2、ALU 计算。
3、根据计算结果判断分支指令。
4、决定需要写入的寄存器端口。

EX 阶段最关键的就是 ALU 的操作。这一块我的改动还比较多。

ALU 输入端为两个 32bit 数据，支持位运算（与、或、异或、或非、左移、逻辑右移、代数右移）、加法、减法、有符号小于比较、无符号小于比较、大于 0 比较，共 12 种功能。在之前的多周期作业中，先从 Controller 中生成 ALUOp 信号，再把 ALUOp 信号传到 ALUController，然后再在 ALUController 中生成 ALUConf 来指导 ALU 工作。其实这完全没有必要，4 位的 ALUOp 完全可以区分所有的运算模式，所以可以直接在 Controller 中生成与 ALU 对接的 ALUOp，直接用 ALUOp 来控制 ALU 工作。当时在多周期中由于程序要求没有采纳这样的设计，在本次流水线中采用。

我的 ALUOp 设置如下：

```
parameter Add      = 4'h0;     // add & addu
parameter Sub      = 4'h1;     // sub & subu
parameter And      = 4'h3;     // and
parameter Or       = 4'h4;     // or
parameter Xor      = 4'h5;     // xor
parameter Nor      = 4'h6;     // nor
parameter Ult      = 4'h7;     // unsigned less than, for SLTU
parameter Slt      = 4'h8;     // signed less than, for SLT
parameter Sll      = 4'h9;     // sll
parameter Srl      = 4'hA;     // srl
parameter Sra      = 4'hB;     // sra
parameter Gtz      = 4'hC;     // greater than zero, for bgtz\bltz
```

所以，就算考虑符号位的区分，一共也只有 12 种模式，完全可以直接用 ALUOp 来控制 ALU，就省去了 Controller 的部分。

6、MEM 阶段及 Lb 指令的支持：

MEM 阶段，流水线第四级，需要完成如下 2 个操作：
1、数据存储器读/写。
2、联动外设。

这一阶段主要就是存储器的操作。主要代码根据多周期代码改编。

因为本次字符串匹配程序是按照 Byte 寻址的，需要用到大量的 Lb 指令，所以特别增加一个 Lb 指令的支持。其实 Lb 和 Lw 指令的实现方式总体差不太多，只是在 MEM 阶段读的数据上有所区别。在具体实现中，我增加了一个 LwLb 控制信号，传递到 MEM 阶段的 Data Memory 中，用来区别是 lw 指令还是 lb 指令。lw 指令时设为 0，lb 指令时设为 1。因为传到 Data Mem 中的地址是 32 位的。那么如果是 lw 指令，就直接按照地址的[31:2]位寻址，也就是按字寻址，返回值。如果是 lb 指令，就先用地址的[31:2]位找到 Byte 所在的 word，然后再利用后两位决定是这个 word 的哪个 byte，代码如下：

```
assign ReadData = MemRead == 0 ? 0 :
                  LwLb == 0 ?
                  (addr[31:2] < MEM_SIZE ? data[addr[31:2]]
                  : addr == 32'h4000000C ? {24'h0, leds}
                  : addr == 32'h40000010 ? {20'h0, AN, BCD}
                  : addr == 32'h40000014 ? system_clocks
                  : 0)
                  :
                  (tail==2'b00 ? {24'h0, data[addr[31:2]][31:24]}
                  :tail==2'b01 ? {24'h0, data[addr[31:2]][23:16]}
                  :tail==2'b10 ? {24'h0, data[addr[31:2]][15:8]}
                  :tail==2'b11 ? {24'h0, data[addr[31:2]][7:0]}
                  :0);
```

这样就可以很好地支持 lb 指令的运行，而不需要在软件层面上做出修改。

其实既然可以增加对 lb 指令的支持，也就应该能够增加对 sb 指令的支持。但是由于本程序并没有用到 sb 指令，所以就没有去做。但在原理上应该也是相似的。

7、WB 阶段：

WB 阶段，流水线第五级，其实这一阶段就没什么操作了，只需要正确地把写入寄存器堆的信号传递到寄存器的写入端就可以了。在下一个时钟周期就可以写入。在前面的数据通路中，MEM/WB 寄存器后面的部分只有一个受 MemToReg 信号控制的多路选择器。

8、控制信号：

不算 ALUOp，Controller 根据 OpCode 和 Funct 一共要生成 14 个控制信号，它们是：

RegWr、Branch、BranchControll、Jump、MemRead、MemWrite、MemtoReg、JumpSrc、ALUSrcA、ALUSrcB、RegDst、LuiOp、SignedOp、LwLb。

为了减小代码规模，不采用对每个指令分别赋控制信号的书写方式，而是采取针对每个控制信号，分不同指令加以赋值。具体代码在 Controller.v 中呈现。

根据前面提到的 ALU 控制信号的设计。在 Controller.v 中直接生成 ALU 的功能控制信号，而无须借助 ALUController.v 进一步转换。

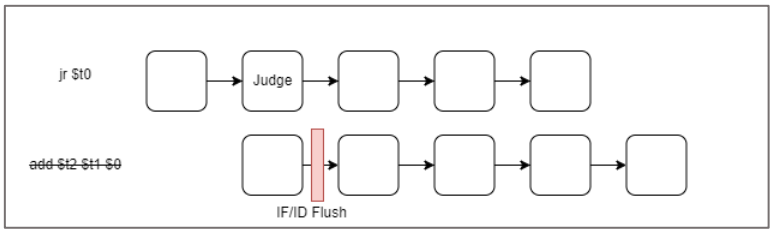
其中 BranchController 信号值得一提。因为需要支持 5 个分支指令：Beq、bne、blez、bgtz、bltz，一开始我担心 ALUOp 支持的运算种类可能会不够，就想用基本指令来代替这些指令。后来发现 ALUOp 其实够用：其中 beq 和 bne 用减法判断结果是否为零，而 blez 和 bgtz 又是互补的关系，一个是≤0，一个是＞0。而 bltz 判断是否小于 0，可以用 slt 代为实现。所以只需要增加一个是否大于零的运

算操作就可以了。然后有一次数逻的作业题也给了我启发，就是可以添加一个信号来把两种“互补”的指令分开，而无须分成两种运算。就比如 beq 和 bne，判断出来结果是否等于 0 之后，如果结果是 0，而且是 beq，那么就需要分支跳转，如果结果是 0，但指令是 bne，就不分支跳转，那么增加一个信号，当 bne 的时候取 1，beq 的时候取 0，把这个信号与 zero 的结果做不进位加，得到的信号就是最终是否要进行分支跳转的信号。感觉这样去区分运算是比较方便高效的。

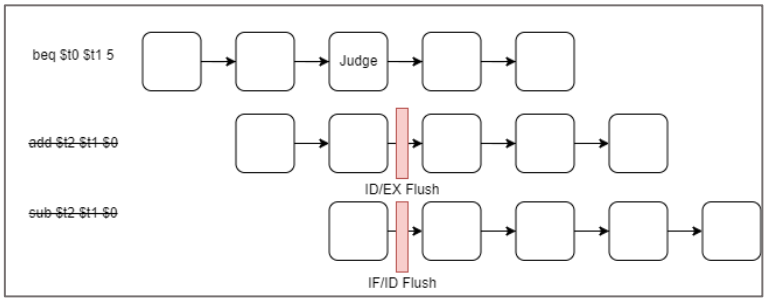
9、分支与跳转

分支跳转会涉及到一个很重要的部分——控制冒险。而控制冒险带来的实际影响就是流水线的 flush。

具体而言，跳转指令在 ID 阶段进行是否跳转的判断，如果确实要跳转，就要把下一条指令清除，也就是执行 IF/ID Flush 操作：



而本次实验的分支指令是在 EX 阶段判断，那么就要把后两条指令给清除，等价于执行 IF/ID Flush 和 ID/EX Flush 操作，也就是这样：



在代码中，我是把所有 stall、Flush 的判断和操作都整合在了 Hazard 模块里，这里面除了分支跳转的判断之外，还有 Load Use 冒险，因为 Load Use 冒险也涉及到流水线的阻塞和清除操作。最终的代码呈现将在下个部分一起给出。

10、冒险的解决

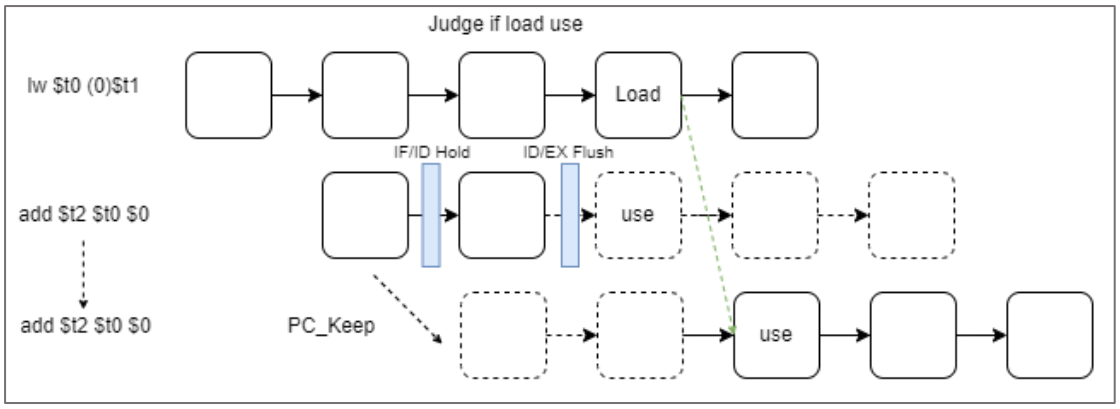
一般的数据冒险采用转发来解决。之前也说过，本程序采用的转发都是转发到 ID 阶段末期。主要有以下两条转发途径：

1、EX/MEM 转发到 ID。

2、MEM/WB 转发到 ID。

转发信号的控制专门整合成 ID_Forward 模块来进行。具体的转发过程在前面的 ID 阶段的解释中已经阐述过。

这里主要解释一下$Load-Use$的处理，$Load-Use$冒险不仅需要采用转发，还要结合对流水线的阻塞和清除操作（需要阻塞一个周期）。而阻塞的实现包括三个操作：PC 保持不变、IF/ID 寄存器保持不变、ID/EX 寄存器清 0，图示如下：



所以结合上一小节的分支跳转的冒险，以及$Load-Use$对流水线的影响，就有了 Hazard 这个 module，统一控制整个流水线的阻塞、清除操作，关键代码如下：

```
assign Load_Use = ID_EX_MemRead
            && (ID_EX_WriteAddr != 0)
            && (ID_EX_WriteAddr == rs || ID_EX_WriteAddr == rt);
assign PC_Keep = Load_Use;
assign IF_ID_Hold = Load_Use;


assign IF_ID_Flush = Jump || if_branch;
assign ID_EX_Flush = if_branch ||  Load_Use;
```

这里的 Jump 代表跳转指令，而 if_Branch 是经过验证，确定要实现分支的一个指示信号。

11、外设与软件译码。

本次实验要求实现的外设有如下几个：8 个 LED 灯，4 位数码管显示最终字符串匹配结果。LED 并没有安排具体的用处。数码管要求采用软件译码的方式来显示最终的字符串匹配的结果。

软件译码是通过 Mips 的指令来实现的。需要将字符串匹配的结果，用 4 位十六机制来表示。然后分别对每位数字，按照 dp、g、f、e、d、c、b、a 的顺序转换成对应的 8 位数码管编码，前面再加上位置对应的 4 位 AN 编码，就得到了 12 位编码，把这个 12 位编码用 lw 写入到 0x40000010 位置，在 DataMem 中把 BCD 和 AN 设置为 output reg 类型，然后在有写入操作的时候进行判断，一旦写入到 0x40000010，就更新 BCD 和 AN 寄存器的状态就可以了。

具体在代码上实现还有一定的技巧。具体的译码部分的代码我放在了 ASMTest 文件下的 decoder.asm 里面。我具体的实现过程拆解如下：

1、结果在$v0 中，把 32 位的$v0 的后十六位，拆解成 4 个 4 位：分别存储在$s3,$s2,$s1,$s0 中：

```
sll $s3 $v0 16
srl $s3 $v0 28 # s3:[15:12]
sll $s2 $v0 20
srl $s2 $s2 28 # s2: [11:8]
sll $s1 $v0 24
srl $s1 $s1 28 # s1: [7:4]
sll $s0 $v0 28
srl $s0 $s0 28 # s0: [3:0]
```

只需要使用 sll 和 srl 指令就可以得到。这 4 个 4bit 数就是需要显示出来的 4 个十六进制数字。

2、把 0 到 f 对应的 8 位数码管编码存在一段连续的内存中，也就是用数组去存这些 16 个数字的译码结果：

```
# 0 00111111   63
# 1 00000110    6
# 2 01011011   91
# 3 01001111   79
# 4 01100110  102
# 5 01101101  109
# 6 01111101  125
# 7 00000111    7
# 8 01111111  127
# 9 01101111  111
# 10 01011111  95
# 11 01111100 124
# 12 00111001  57
# 13 01011110  94
# 14 01111011 123
# 15 01110001 113
```

上图是我自己手动将这 16 个数字进行转换的结果，最后一列是转化成了 10 进制，以便后面的代码书写。

存储到内存中的代码是这样的：

```
addi $a0 $0 268501248
addi $t0 $0 63
sw $t0 0($a0)
addi $t0 $0 6
sw $t0 4($a0)
addi $t0 $0 91
sw $t0 8($a0)
addi $t0 $0 79
sw $t0 12($a0)
```

这里仅列出一部分的代码，这 16 个数字都用 sw 指令存到 DataMem 中，相当于存了一个码表，只需要通过适当的方式找出对应的位置就可以取出译码结果。

3、因为这 16 个数字是存在一段连续的内存中的，而且起始的地址我是知道的。这样就能用非常方便的方式找出数字对应的编码，只需要将数字乘以 4，然后加上数组的起始地址，就得到了对应的译码结果所在的 word，直接 load 就行了。

举个例子：

```
sll $s0 $s0 2
add $s0 $s0 $a0
lw $s0 0($s0)
```

$a0 中存储的是 16 个数字的码表的起始地址，之前将匹配结果的最后 4 位数字放到了$s0 寄存器中（也就是显示数字的最后一位），这里将$s0 乘以 4，再加上起始地址，就得到了$s0 的数字对应的译码在码表（内存）中的位置，再用 lw 就可以获得译码结果了。

这样获得的结果只是后 8 位，没有 AN 信号，实际上 AN 信号是非常容易添加的，比如$s0 是最后一位，AN 应该是 0001，那么我就让之前的译码结果加上 0001,0000,0000（十进制为 256）就可以了，所以最后一位数字的完整译码过程如下：

```
sll $s0 $s0 2
add $s0 $s0 $a0
lw $s0 0($s0)
addi $s0 $s0 256
```

译码结果仍然存储在$s0 寄存器中。

分别对每一位数字逐个进行操作即可。

4、构造循环显示：

需要逐个把译码结果用 sw 写到内存中，那么就可以采用两层循环：外层大循环是个死循环（可以用 j 指令实现），保证显示是一直进行的，在大循环中有四个小循环，每个循环控制运行时间为 1ms 左右，然后在每个小循环之前把需要显示的数字的译码结果用 sw 写到内存中就可以了。

将相应位置的数字写到内存中并开始一轮长达 1ms 的小循环的代码如下：

```
sw $s0 0x40000010($0)
addi $t0 $0 0
for1:
addi $t0 $t0 1
bne $t0 10000 for1

sw $s1 0x40000010($0)
addi $t0 $0 0
for2:
addi $t0 $t0 1
bne $t0 10000 for2
```

总共有 4 个小循环。分别对应 4 个数字的显示，外层再套一个大循环即可。以上就是数码管外设和软件译码的全部过程。

12、指令如何翻译成机器码

这一块还是很值得记录的。我一开始以为指令译码是非常简单的事情，只需要把之前写好的 Mips 指令在 Mars 编辑器中翻译成 32bit 机器码，再导入到 InstMem 中就可以了。但真正做起来，才发现很多原本不知道的事情。有如下几点：

1、数逻课程的 Mips 程序包含文件的读取操作，运算结果的控制台输出操作，存储空间的申请和释放操作。而这些操作都需要使用 syscall 指令。但是 syscall 是 Mars 独有的系统函数，不可能加以实现，而且在我的流水线中也不太可能实现文件的读写操作，所以这些部分都要修改。

2、待匹配的字符串和模式字符串如何导入程序。在原来的 Mips 程序中是通过文件读写的方式进行的。我采用如下方法：将待匹配字符串和模式字符串都手动转换为 ASCII 码值并按顺序相连，每个字符对应 8 位二进制码，4 个字符组成一个 word，如有空位补 0，把全部字符串转换为 word 形式，再用 sw 指令写入到 Data Memory 中，用到时直接从 Data Memory 中取数即可。

3、由于我的指令存储器中是以 0 作为起始下标的，如果是 j 型指令的话跳转的值和 Mars 中的值不同。需要手动修改一下。

13、提高要求：UART 串口的设计

串口部分的代码基于数逻实验三的代码修改实现。

我的具体设计在数据通路中也有体现。因为我的数据加载是通过指令来进行的。所以不需要对 Data Memory 部分进行数据传递，而只需要对 Inst Memory 部分进行数据导入就可以了。又因为程序的执行结果是存储在一个固定的寄存器里的，所以直接从 Register File 中把寄存器的值牵出来接到发送模块就可以了。

添加串口后运行的流程是这样：

1、将比特流烧入板子中，此时板子上没有任何反应，数码管也不会亮。

2、打开串口调试助手，将指令发送。

3、串口接收完毕，会亮起 K3 灯，而我在程序中设置一旦接收完毕就开始执行代码，由于程序运行极快，所以数码管几乎立刻就显示最终结果（我没有把数码管显示给取消）。

4、然后将 SW0 推上去，表示开始发送。然后串口调试助手就会接收到最终的结果。(以一个 word 的形式)。

三、测试验证和性能分析

1、Mips 指令数统计：

我的测试和验证过程都是基于字符串匹配的 kmp 算法：

我的测试数据如下：

待匹配串：abababababababababababababababab

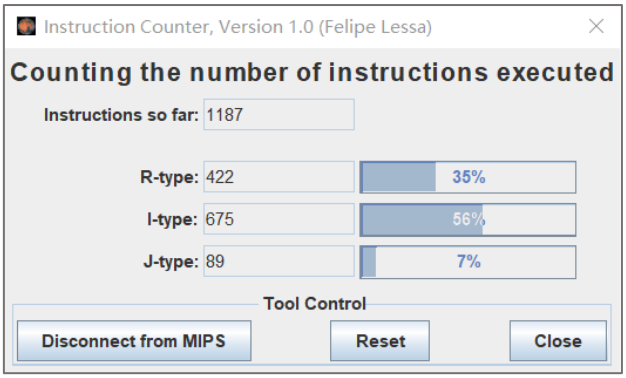
模式串：　abab

正确答案应该是 15，在 Mars 仿真器中运行正确：

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x0000000f
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0000001d
$t1	9	0x00000004
$t2	10	0x0000000f

$t2 是程序中记录匹配次数的计数器，而$v0 存储最终的结果。两者的值都是 f。

在 Mars 中统计指令数（不包括软件译码实现扫描的指令）：



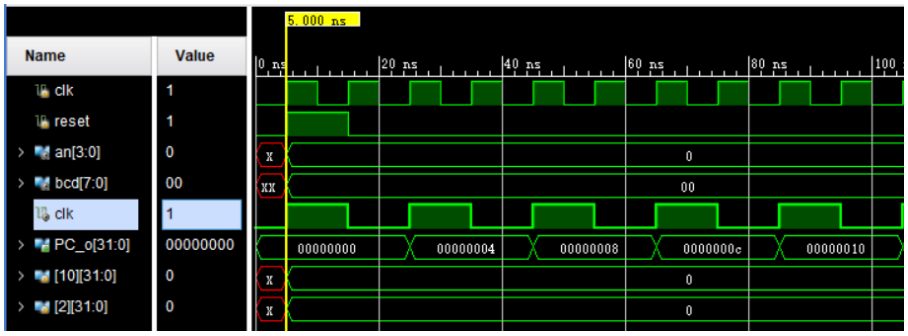
根据统计，运行完整个程序，一共有 1187 条指令，其中 I 型指令占比最大，J 型指令占比最低。在我的程序中，确实也频繁地涉及到立即数操作。所以这个结果也很合理。

2、Verilog 代码仿真确定完成字符串搜索算法所消耗的时钟周期

我的 Mips 代码的最后一个指令，是把计数器$t0的最终值，也就是总共的匹配结果存到$v0中去，所以当$v0 的值改变成最终结果时，认为程序执行完毕。已知$t0是 10 号寄存器，而$v0是 2 号寄存器。且在仿真的时候保留时钟分频模块：

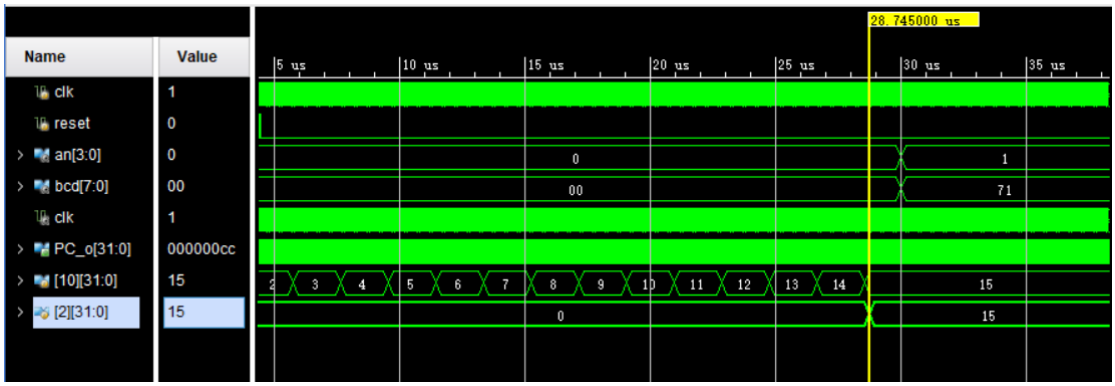
```
always @(posedge system_clk)    //100MHz->50MHz
begin
        clk <= ~clk;
end
```

这个时钟分频模块将把 100MHz 的时钟频率降低一半，变成 50MHz。仿真的开始情况如下：



认为 5ns 为程序开始执行的时间。

仿真的最终阶段情况如下：



如图所示，在 28.745us 时，$v0寄存器的值变为最终结果 15。表明程序运行完毕，testbench 中模拟出的原始时钟周期为 10ns 一个周期（也就是 100MHz），经过分频为后 50MHz。

所以仿真中运行的周期数为：

$$\frac{28745-5}{2\times 10}=1437$$

3、CPI（Cycle Per Instruction）的计算：

由 Mips 中统计出的指令数目和仿真出的执行周期数，可计算出本流水线 CPU 的 CPI 为：

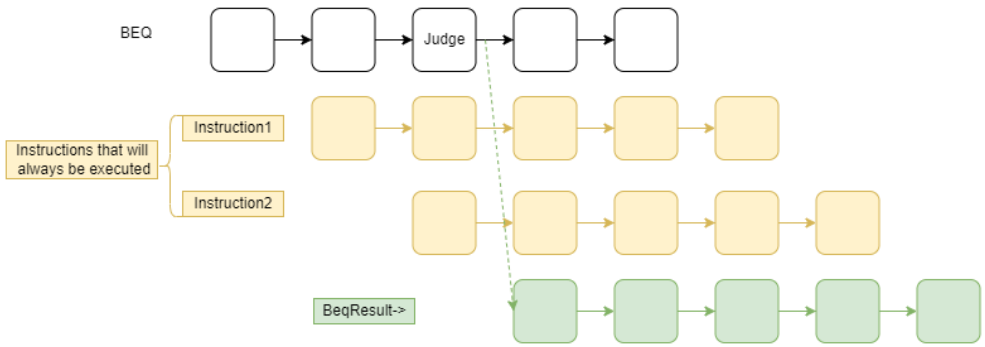
$$CPI=\frac{1437}{1187}=1.2106$$

CPI 的值还算可以。但是仍然有较大的进步空间。因为本 CPU 的分支判断是在 EX 阶段进行的，如果提前到 ID 阶段会更好一些。

4、延迟槽与动态分支预测的尝试与实现：

在数逻课上讲过针对分支指令的延迟槽技术，也就是在编译程序时，在分支指令后插入适当的指令，这条指令无论之前是否分支，总是被 CPU 执行。所以在本次数逻实验中也来尝试一下。

由于本次实验的分支判断是在 EX 阶段进行的，就存在两个延迟等待周期，那么就需要在分支指令后插入两条必须要执行的指令，大意如下图所示：



我也考虑过在硬件中实现延迟槽，但是发现这相当复杂：需要提前读取分支指令周围的多条指令，进行判断：哪些指令是必须要执行的，哪些指令可以被放

到延迟槽中，然后再在 pc 中自动决定执行哪条 PC 对应的指令。这所花的代价太大了，于是，我决定手动调整代码的顺序。

实际上手的时候发现，延迟槽技术能够调整的指令少之又少，满足条件的并不多，像下面这样的代码块只有一两处：

```
1.  add $t0 $s1 1
2.  lb $t0, 0($t0)
3.  bne $t3 1 endfor2
4.  add $t3 $t0 $t1
5.  add $t3, $t3, $s0
```

这部分可以被调整为

```
1.  bne $t3 1 endfor2
2.  add $t0 $s1 1
3.  lb $t0, 0($t0)
4.  add $t3 $t0 $t1
5.  add $t3, $t3, $s0
```

但是在大多数分支指令的应用场景中，都是对一个逻辑段的最终结果做出判断，前面的指令环环相扣，根本无法进行良好的调整，就像这样：

```
1.          slt $t3, $t1, $s3       # $t0 = ($s0 < $s1) ? 1 : 0
2.          bne $t3 1 endfor2
3.          add $t3 $t0 $t1  # i+j
4.          add $t3, $t3, $s0 # str[i+j]
5.          lb $t3, 0($t3)
6.          move $t4 $t1
7.          add $t4, $t4, $s1 # pattern[j]
8.          lb $t4, 0($t4)
9.          bne $t3 $t4 endfor2 #if(str[i + j] == pattern[j])--> for
10.         addi $t1, $t1, 1            # j++
11.         j for2
```

最终调整之后的代码仿真出的周期数只少了约三十个周期，效果很小。而且延迟槽需要经过编译器调整，在 FPGA 板子也并没有很普适的应用空间。

动态分支预测是我尝试的第二项技术，在数逻课上也重点介绍过。我的实现方式是：根据上一次是否成功分支的结果来决定这一次是否进行分支，如果上一次分支了，这一次就提前分支，如果上一次没分支，这一次就提前不分支。

但是仿真结果也并不明显，总的周期数减小了约七十个周期，还让整体的时钟变慢了好多。我觉得这是采用的字符串匹配的算法导致的，因为在这个算法中，是采用暴力手段逐位对字符串进行比对的，beq 就分布在一个个比对操作之后，beq 的结果并不具有时间上的稳定性，换句话说，beq 在绝大部分情况下都是一会儿跳转，一会儿不跳转，这就使得分支预测几乎失效。

通过测试和验证，延迟槽与动态分支预测这两项技术都没有很好的效果。但是通过这次的动手尝试，我也对这两项技术适用的场景、算法都有了更深的体会，还是收获很大的。

5、流水线 CPU 的时序性能：

仿真时序性能时，把分频模块删去，并调整相关代码。

我在约束文件中，设置周期为 20ns 的时钟进行时序仿真：

```
27 create_clock -period 20 -name system_clk -waveform {0.000 10.000} -add [get_ports system_clk]
```

在仿真过程中，我发现综合（synthesis）之后的时序分析和实现（implementation）之后的时序分析有一定差距。综合之后的时序裕度可以达到 10.3ns（在周期为 20ns 的情况下），也就是最高时钟频率达到了 103MHz，这个值还是比较好的。比当时的单周期（80MHz 左右）快很多，也比当时的多周期快一点（多周期是 101MHz 的样子，虽然快不了多少）。

但是实现（implementation）之后的时序裕度就要差一些，下面这张图是实现后的时序分析报告：

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.639 ns	Worst Hold Slack (WHS):	0.080 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	38525	Total Number of Endpoints:	38525	Total Number of Endpoints:	18459

All user specified timing constraints are met.

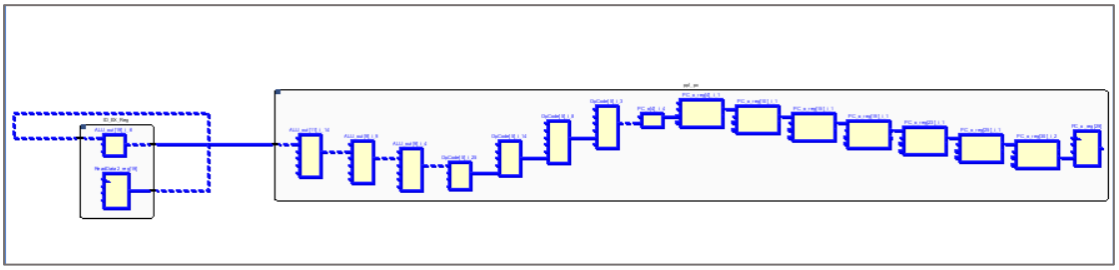
可以计算出最高时钟频率是

$$\frac{1}{20-8.639}\times 10^9 = 88.02MHz$$

这个值比当时数逻写的单周期 CPU 快一些，但没有多周期快。

周围的很多人都遇到了这样的问题，在实现之后的时钟频率相比于综合之后的时钟频率降低了。为了探究时钟频率降低的原因，我比较了综合和实现分别做时序分析的关键路径：

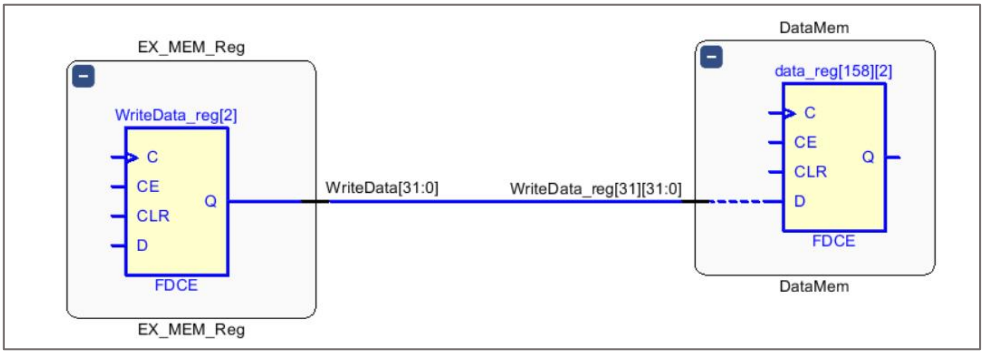
综合后，时序分析的关键路径如下：



From	To
ID_EX_Reg/ReadData2_reg[19]/C	ppl_pc/PC_o_reg[29]/D

这条路径是 PC 的更新路径，其中涉及到 ID/EX Reg 和 PC 两个模块。还是比较合理的。

再来看下实现(implemention)之后的最长路径：



From	To
EX_MEM_Reg/WriteData_reg[2]/C	DataMem/data_reg[158][2]/D

Vivado 显示得很清楚，关键路径是在 DataMem 的写操作上。这其实也挺合理的，因为 DataMem 有 512 个字，需要根据地址逐个比对来找出读取和写入的位置，确实是一个很费时间的过程。之前数逻课上，老师也说过 DataMem 的读和写都是相当耗费时间的，这和之前的理论认知也是相符的。

而且遗憾的是，我目前也没找到比较简便的方法来优化这一路径。(ip 核似乎可以解决这样的问题，但是改动较大，故没有去做。)

同时这也使我思考，为什么综合和实现之后的时序性能有这么大的差异？

此外，在增加串口接收和发送的功能之前，我也对当时的 CPU 做过时序仿真，当时实现之后的最高时钟频率为 89.66MHz，可见加了串口之后并不会对时钟频率造成本质的影响。

6、由时序性能差异思考 synthesis 和 implementation 的本质区别：

在之前的学习过程中，我对综合和实现的认识并不充分，这次 CPU 在综合和实现之后的性能的巨大差异，引起了我对这两者的思考。下面简要记录一下我查阅相关资料之后的所得。

关于综合：

1、综合是将高级抽象层次的电路描述转化成较低层次的描述。也就是将语言描述的电路逻辑转换成与或非门和触发器等基本逻辑单元的互联关系。
2、所谓“可综合”，就是说这段代码可以被翻译成门级电路。
3、综合不但能够翻译电路，还能够优化我们的电路。可实现去除冗余、结构复用等效果。

关于实现：

1、 实现是一个布局布线的过程。
2、 综合之后的门级网表只揭示了门与门之间的虚拟的连接关系。并未规定门的位置以及连线的长度。所谓布局布线，就是讲门级网表中的门的位置和连线信息确定下来的过程。

3、 FPGA 可重复编程的基础是拥有巨量的可配置逻辑块（CLB）、丰富的布线资源以及其他资源。在此基础上，“布局”的过程就是将门级网表中的每一个门安置到 CLB 中的过程；而布线是利用 FPGA 中丰富的布线资源将 CLB 根据逻辑关系连接在一起的过程。而布局布线又有着具体的算法策略。

所以，综合之后的时序分析并不能说明真正的运行性能。因为综合后还有很多关系和位置没有确定，但是实现之后，时序分许就能够真正代表实际的性能。所以按照我的实现的结果，需要对 100MHz 的时钟降频，我采取直接把频率降低一半到 50MHz 的方法，这样就完全足够 CPU 的正常运行了。之前写好的分频模块恰好能够满足这样的需求。但有趣的是，在板子上运行的时候，我发现不加分频模块也能够正确运行。可能是没有遇到最长的路径的原因。

7、时序性能尝试优化：

根据之前的时序报告，发现 Data Memory 那块很慢。于是想办法进行优化。一个想法是，是不是在 DataMemory 中同时实现 lw 和 lb 导致连线和查找表太多了，因为那块的逻辑对于电路而言确实是比较复杂，需要在存储 512 个字的寄存器中找到对应的字，再根据是否是 lb 来决定最终返回的是字还是字节，所以我尝试将这块简化一下，只保留 lb 指令的操作，因为在本程序中，其实用不太到 lw 指令（除了一开始的字符串初始化）用到了其实也可以用 lb 来替代，也不会耗费太多额外的时钟周期。尝试效果如下：

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0.888 ns	Worst Hold Slack (WHS):	0.080 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	-13.426 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	28	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	38496	Total Number of Endpoints:	38496	Total Number of Endpoints:	18460

在 10ns 的时钟周期下是裕度是-0.888ns，也就是最高时钟频率为 91.84MHz，稍有提升，但提升并不大。

进一步地，我觉得还可能是我的模块分得太散，像 PC 的计算以及数据转发信号的计算等等操作，其实本质上是不需要单独列成一个模块的，写代码的时候我为了思路清晰和可读性把他们分开了，这样可能会造成连线的延时比较长。而且因为代码书写的习惯，我用了大量的 wire 将前后的变量相连，其间可能也会额外占用查找表的资源。于是我尝试把能合并的模块合并到一块，简化一下变量之间的连线，看看效果：

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.728 ns	Worst Hold Slack (WHS):	0.080 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	38537	Total Number of Endpoints:	38537	Total Number of Endpoints:	18465

All user specified timing constraints are met.

时钟约束中还是 10ns 的周期，但是裕度达到了 0.728ns，也就是最高时钟频率有 107.85MHz，这样的效果还是很不错的，虽然没有很大幅度的提高，也很令人欣慰了。不过经过调整之后的代码，在可读性上就差了很多，因为两三个模块都被放进 CPU 总模块里了。这提醒我，在硬件编程的时候，要注意一些代码的习惯，既不能完全为了思路的顺畅而把 module 划分得太散，也不能一股脑把模块都揉到一块去。

8、资源占用：

代码实现之后的资源占用情况如下：

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)
PipelineCPU	9336	18053	3397	1680	7193	8984
DataMem (DataMem)	7288	16456	3264	1616	6603	7288
EX_MEM_Reg (EX_ME...	0	74	0	0	35	0
ID_EX_Reg (ID_EX_R...	468	158	4	0	204	468
if_id_reg (IF_ID_Reg)	11	90	0	0	31	11
MEM_WB_Reg (MEM_...	5	71	0	0	37	5
ppl_instMem (InstMem)	455	0	0	0	134	103
ppl_pc (PC)	183	32	1	0	118	183
ppl_UART_MEM (UAR...	117	180	0	0	79	117
RegFile (RegFile)	807	992	128	64	469	807

总共使用了 9336 个查找表，和 18053 个寄存器。

下图是我多周期的资源占用情况：

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (210)	BUFGCTRL (32)
MultiCycleCPU	4085	9506	1288	512	2	1
alu (ALU)	25	0	0	0	0	0
alu_out (RegTemp_2)	307	32	0	0	0	0
controller (Controller)	727	24	4	0	0	0
instAndDataMemory (I...	2208	8192	1024	512	0	0
instreg (InstReg)	136	34	4	0	0	0
Mem_Data_Reg (Reg...	0	32	0	0	0	0
pc (PC)	2	32	0	0	0	0
RegisterFile (Register...	680	992	256	0	0	0
RF_Read_data_A_Re...	0	32	0	0	0	0
RF_Read_data_B_Re...	0	136	0	0	0	0

下图是我单周期的资源占用情况：单周期的资源占用小于多周期。

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (210)	BUFGCTRL (32)
CPU	3991	8758	1266	566	66	2
RF (RegisterFile)	1285	480	235	54	0	0
dataMemory (DataMe...	2245	8192	1024	512	0	0
alucontrol (ALUControl)	113	5	7	0	0	0
alu (ALU)	345	49	0	0	0	0

可见，流水线的资源占用情况几乎是多周期的两倍（查找表是 2.28 倍，寄存器是 1.899 倍）。流水线耗费的资源还是相当多的。可能是由于各级次之间寄存器的存在（IF/ID Reg）使得寄存器耗费比较多。而寄存器的各种转发、级次之前的连接等等也都加大了资源消耗。当然，资源占用的巨头还是 Data Memory 模块，一个 Data Memory 模块就占了 78%的查找表和 91.15%的寄存器。这也是意料之中的，因为 Data Memory 不仅要写，还要读，既要支持 lw 也要支持 lb，确实资源消耗也比较多。

四、实验总结：

写到这里，回顾最后一个实验的整个历程，感到真不容易。还记得第一次写完代码，打开 vivado 软件开始验证，然后对着一点都不对的仿真图像疯狂抓头。从 PC 的正确更迭，到冒险处理与数据转发，从指令的译码，再到最后的串口，一边写代码，一边微调数据通路，一边看仿真，一边 debug。终于，当我在仿真窗口看到正确的运行结果被显示时，当我在板子上看到数码管正确显示的时候，不禁长舒一口气。

从这次实验中，我学习到很多有用的技巧：比如在写代码之前，就要进行充分的考虑和思考，数据通路就是一种很好的形式，思考的深度和成熟度决定了后面写代码和 debug 的速度；还比如在写代码的时候，因为转发以及各种数据交叉的原因，还没写到后面的模块，却已经需要用到后面模块的寄存器或者输出。这个时候，可以先声明 wire，再等到这个模块被声明的时候，用 wire 把他们连接起来。这样写代码的思路就比较顺（虽然可能会造成额外的资源占用，在写完之后最好再手动优化一下，在不影响代码阅读的情况下尽量合并）。我还学会了在一般情况下不要乱改端口或者 wire 的名字，能够用相同的就用相同的名字，否则当变量太多之后梳理不清。等等等等，在完成这一整个实验之后，体会确实很深刻。

通过本次实验，我对流水线的运行原理有了更深层次的体会。很多在理论课上的知识，像延迟槽、动态分支预测、数据转发等等等等，都在代码中和 FPGA 板子上被运用和证实，也获得了对知识的更深的理解，这大概是我最大的收获。

最后，感谢孙老师一学期的辛勤指导！以及助教一学期的付出！感谢！

五、代码文件清单：（全部代码放在 Codes 文件夹下）

```
├─ASMTest 文件夹
│       Brute_Force.asm          采用的 Mips 程序
│       decoder.asm              数码管显示 Mips 程序
│       decoder.txt
│       test.asm                 测试用 asm 程序
│
├─Constraint 文件夹
│       PipelineCPU.xdc          约束文件
│
├─Cplus_Generate-InstMem 文件夹
│       IN.txt
│       out.txt
│       源. cpp                  在有串口之前将指令转换格式为 InstMem 中的代码
│
├─PipeLineRegs 文件夹            各级流水线之间的寄存器
│       EX_MEM_Reg.v
│       ID_EX_Reg.v
│       IF_ID_Reg.v
│       MEM_WB_Reg.v
│
├─source 文件夹                  流水线中的各个模块
│       ALU.v
│       Controller.v
│       DataMem.v
│       Frequency.v
│       Hazard.v
│       ID_Forward.v
│       ImmProcess.v
│       InstMem.v
│       PC.v
│       PC_IN.v
│       PipelineCPU.v
│       RegFile.v
│
├─TestBench 文件夹               仿真文件
│       TestCPU.v
│
└─UART 文件夹                    提高要求：串口收发代码
        UART_MEM.v
        uart_rx.v
        uart_tx.v
        指令发送.txt
```