# MIPS 流水线处理器实验报告

张作柏

17300240035

2019 年 5 月 26 日

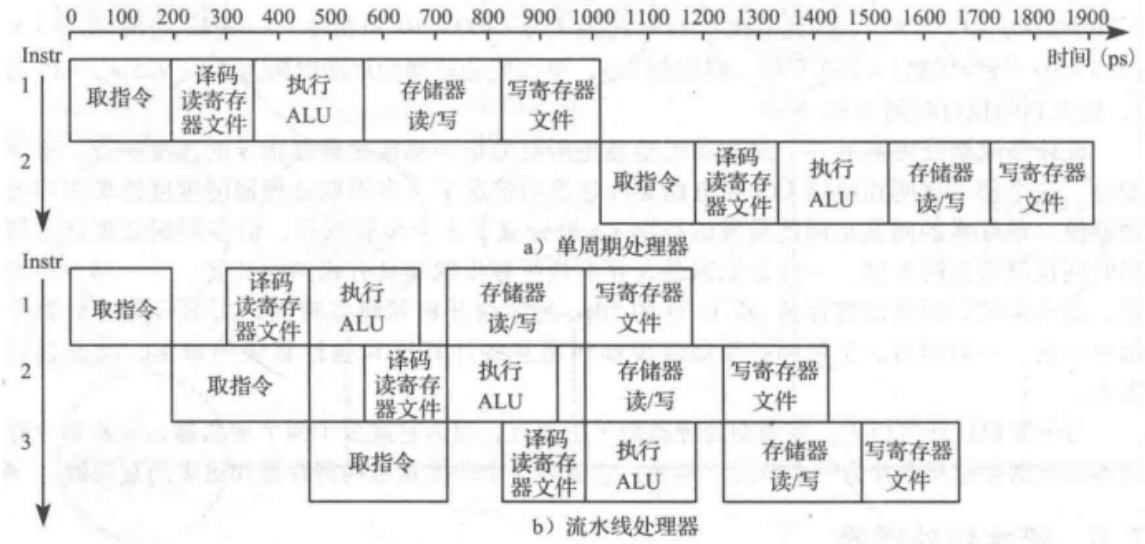
## 目录

# 1 流水线处理器简介

流水线技术是提高数字系统吞吐量的有效手段。通过将单周期处理器分解成 5 个流水线阶段来构成流水线处理器。因此，可以在流水线中同时执行 5 条指令，时钟频率几乎可以提高 5 倍。

流水线被划分为五个阶段，每个阶段完成一个操作：取指令 (Fetch)，译码 (Decode)，执行 (Execute)，存储器 (Memory) 和写回 (Writeback)。

- **取指**：处理器从指令存储器中读取指令。
- **译码**：处理器从寄存器文件中读取源操作数并对指令译码以便产生控制信号。
- **执行**：处理器使用 ALU 执行计算。
- **存储器**：处理器读或写数据存储器。
- **写回**：处理器将结果协会到寄存器文件。




上图对比了单周期处理器与流水线处理器的时序图[1]。可以看出，流水线的吞吐量明显高于单周期，也说明了流水线处理器的高效性。。

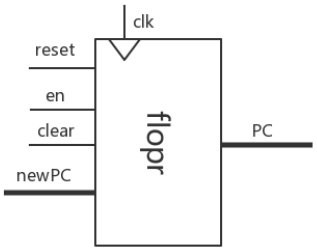
流水线系统中的核心问题是冲突 (Hazard)，即当后一条指令需要前一条指令的计算结果，而前一条指令还没有执行完时就会发生冲突。在这里，我们可以使用重定向、阻塞和刷新三种处理方法。

[1]图源于教材 256 页

# 2 部件分析

多周期处理器中大部分器件都与单周期相似，在此只列出新添加或发生改动的部件，其余请参考我的单周期报告。

## 2.1 触发器 flopr

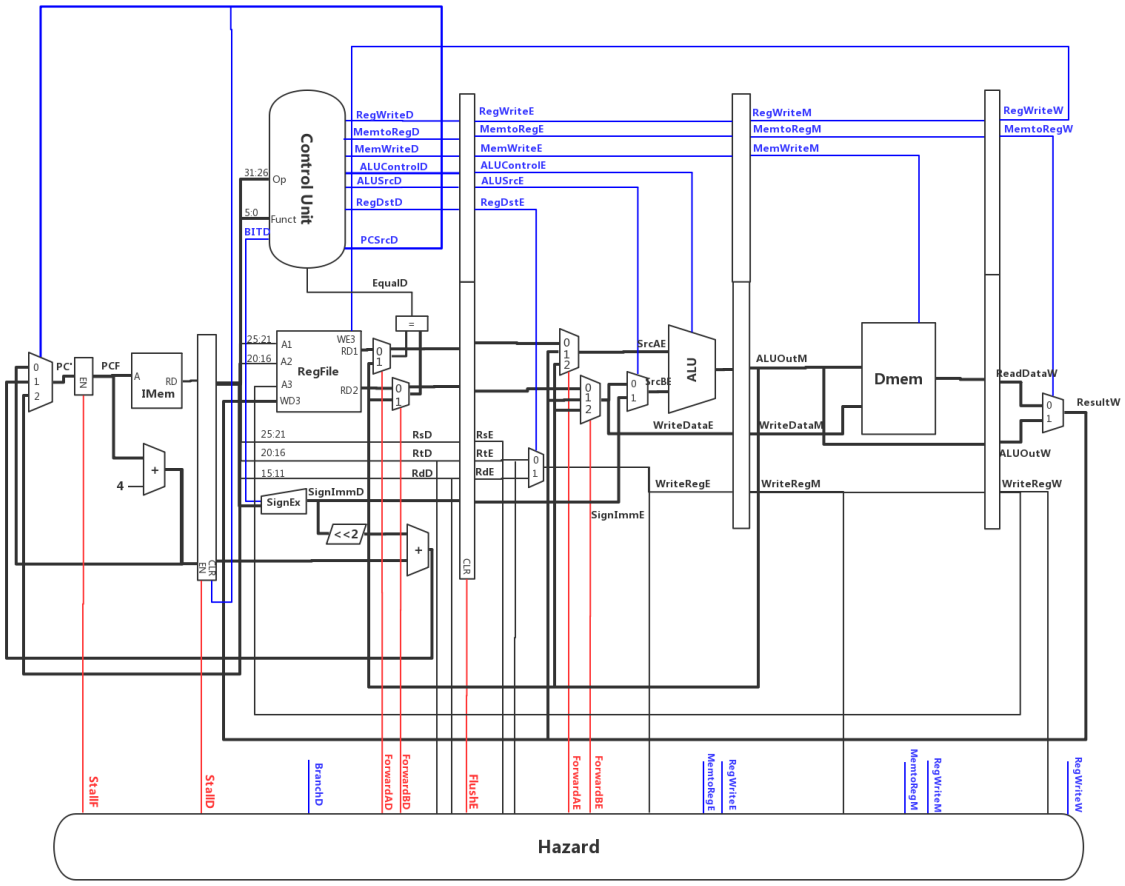



流水线中，除了本来 datapath 中的寄存器，我们还需要添加两两阶段之间的状态寄存器。而为了支持状态的阻塞和刷新操作，我们需要给寄存器加入同步清零功能。所以，寄存器共有异步清零 reset、使能 en 和同步清零 clear 三个控制口。

## 2.2 比较器 eqcmp


为了提前预测 beq 和 bne 是否跳转，我们需要在 decode 阶段预判两个操作数是否相等，所以需要添加一个比较器。A 与 B 都是 32 位数字，当 A==B 时，eq=1，当 A!=B 时，eq=0。

## 2.3 数据通路 Datapath




**功能说明**：因为与单周期相比很多部件都没有改变，所以这里只说明变化的部分。

- 左侧 PC 寄存器，添加了 PCEn 信号，控制寄存器写入逻辑，只在需要更新 PC 值的状态才将 PCEn 设置为 1。
- 左侧二路选择器，因为 Dmem 与 Imem 合二为一了，所以要通过 IorD 信号选择读取指令还是读取数据。
- 左侧 Instr 寄存器，只有当需要读指令时，才将 IRWrite 信号设为 1，其余状态则需要存储原先的指令。
- 中间 A、B 两个寄存器用于存从 RegFile 中读出来的值，因为这些值可能会延迟一个状态使用，所以需要用寄存器储存。
- 中间 SrcA 是 ALU 的 A 参数，由 ALUSrcA 信号控制：当 ALUSrcA 为 0 时，选择 PC 值作为运算对象 (常用作计算下一指令的 PC 值)；当 ALUSrcB 为 1 时，选择 A 寄存器的值作为运算对象，用于 R 型和部分 I 型指令。
- 中间 SrcB 是 ALU 的 B 参数，由 ALUSrcB 信号控制：当 ALUSrcB 为 00 时，选择 B 寄存器的值作为运算对象，用于 R 型指令；当 ALUSrcB 为 01 时，选择 4 作为

运算对象，用于计算下一条指令的 PC；当 ALUSrcB 为 10 时，选择移位前的立即数作为运算对象，用于 I 型指令；当 ALUSrcB 为 11 时，选择移位后的立即数作为运算对象，用于计算 jal、bne、beq 指令跳转的目标地址。

- 右侧 ALUOut 寄存器用于存储 ALU 运算的结果，以供下一周期使用。
- 右侧四路选择器用于选择 PC 的值，由 PCSrc 信号控制，共四种选择：ALUResult(下一指令 PC)，ALUOut(jal、bne、beq 指令的跳转地址)、PCJump(j 指令的跳转地址)、A(jr 指令的跳转地址)。

## 2.4 控制模块 Controller

### 2.4.1 控制信号说明

### 2.4.2 实现细节

最后，在每个时钟上升沿到来时，用 nextstate 更新 state：

## 2.5 冲突处理模块 Hazard

# 3 冲突处理与额外指令分析

## 3.1 冲突处理

图中未标出的控制变量均默认为 0。

## 3.2 额外指令分析

### 3.2.1 移位指令 sll

注意到，这里没有为移位指令单独设计状态，而是将其作为普通的 R-Type 指令来处理。因为我改造了 ALU，把移位数传到了 ALU 中，这样就 ALU 就可以直接根据 funct 进行移位操作了。过程与其它 R-Type 指令完全一样，无需增加新的状态。

### 3.2.2 函数调用指令 jal

我为 jal 指令新增加了一个状态 17：JALExecute，在这个状态中我们要将返回地址写入寄存器 $ra 中。那么，

### 3.2.3 跳转寄存器指令 jr

跳转寄存器指令也比较好实现，只需要为 PC 的来源增加一个选项为寄存器值即可。然后增加一个状态 JRExecute，选择 PCSrc=3，并设置 PCWrite 为 1。

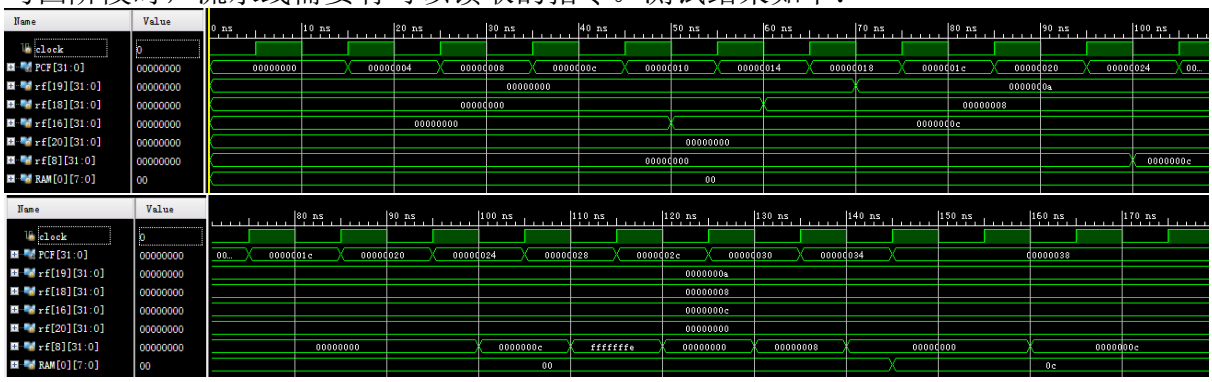
# 4 测试样例与结果

与单周期相同的测试样例均已通过，这里只列举几个关键的和新加入的测试样例。

## 4.1 all.in

```
 0x0 : addi $s0, $0, 12      | 2010000c
 0x4 : andi $s2, $s0, -8     | 3212fff8
 0x8 : ori $s3, $s1, 10      | 3633000a
 0xc : slti $s4, $s2, 5      | 2a540005
0x10 : nop                   | 00000000
0x14 : add $t0, $s0, $s1     | 02114020
0x18 : sub $t0, $s2, $s3     | 02534022
0x1c : and $t0, $s3, $s1     | 02714024
0x20 : or $t0, $s1, $s2      | 02324025
0x24 : slt $t0, $s0, $s2     | 0212402a
0x28 : sw $s0, 0($0)         | ac100000
0x2c : lw $t0, 0($0)         | 8c080000
0x30 : nop                   | 00000000
0x34 : nop                   |
```

注意，这里需要在所有指令结束后添加 nop 指令，因为当最后一条指令在存储器和写回阶段时，流水线需要有可以读取的指令。测试结果如下：




上图中依次给出了寄存器 $s3, $s2, $s0, $s4 和 $t0，以及内存地址 0 处的值。每条指令的结果依次应为：

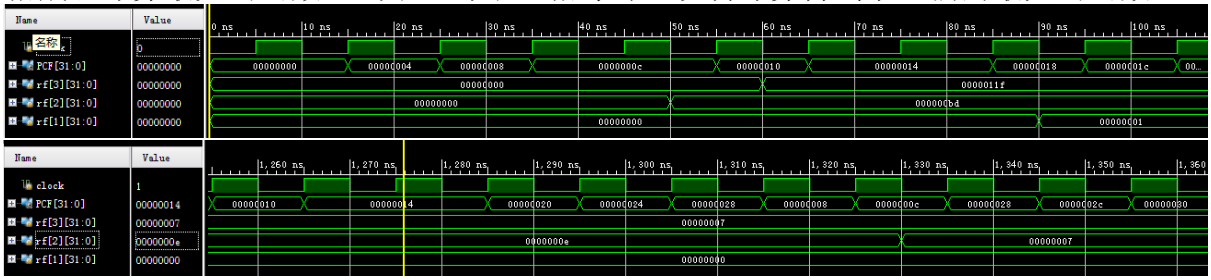
$s0 = 12 → $s2 = 8 → $s3 = 10 → $s4 = 0 → nop → $t0 = 12 → $t0 = -2 → $t0 = 0 → $t0 = 8 → $t0 = 0 → 0($0) = 12 → $t0 = 12

## 4.2 gcd.in

```
 0x0 : addi $v0,$0,189       | 200200bd
 0x4 : addi $v1,$0,287       | 2003011f
 0x8 : main:                 |
 0x8 : beq $v0,$v1,end       | 10620007
```

```
 0xc : slt $at,$v0,$v1       | 0043082a
0x10 : beq $at,$0,run        | 10010003
0x14 : add $at,$v0,$0        | 00400820
0x18 : add $v0,$v1,$0        | 00601020
0x1c : add $v1,$at,$0        | 00201820
0x20 : run:                  |
0x20 : sub $v0,$v0,$v1       | 00431022
0x24 : j main                | 08000002
0x28 : end:                  |
0x28 : add $t3, $0, $0       | 00005820
```

这个测试样例是从黄文皓同学那里借来的，通过循环计算两个数的最大公约数。开始时，我们把需要计算的两个数字 189 和 287 分别赋值给 $v0 和 $v1。之后，通过辗转相减法计算最大公约数。当 $v0 与 $v1 相等时，说明计算得到了它们的最大公约数。



这里的两幅图分别展示了程序开始与结束时的计算结果。从第一幅图可以看到程序正常的执行循环，后一幅图则说明程序成功地计算出两数的最大公约数为 7。

## 4.3 quick_multiply.in

```
 0x0 : addi $t0, $0, 99      | 20080063
 0x4 : addi $t1, $0, 37      | 20090025
 0x8 : addi $s0, $0, 0       | 20100000
 0xc :                       |
 0xc : while:                |
 0xc : beq $t1, $0, done     | 10090006
0x10 : andi $t2, $t1, 1      | 312a0001
0x14 : beq $t2, $0, target   | 100a0001
0x18 : add $s0, $s0, $t0     | 02088020
0x1c : target:               |
0x1c : sll $t0, $t0, 1       | 00084040
0x20 : srl $t1, $t1, 1       | 00094842
0x24 : j while               | 08000003
0x28 :                       |
0x28 : done:                 |
0x28 : add $t3, $0, $0       | 00005820
```

这个代码曾在单周期处理器中演示过，用于计算两数的乘积，方法类似快速幂。首先将 99 与 37 写入 $t0 和 $t1 寄存器，然后进入循环，每次根据 $t1 的末位是否为 1，来

决定是否向 $s0 进行累加，然后每次 $t0 乘 2，$t1 右移一位。最终，计算结果存在 $s0 中，应为 0xe4f。这个测试样例测试了跳转、移位等指令，测试结果如下：

图中依次展示的是寄存器 $s0,$t2,$t1,$t0 的值，可以看到程序开始时，寄存器 $t0 与 $t1 分别被赋值为 0x63 与 0x25，分别对应 99 与 37。循环过程中，$t0 每次乘 2，$t1 每次除以 2，最终 $s0 变为 0xe4f，结果正确，说明程序正确执行。

# 5 注意事项

## 5.1 遇到的问题

想了一下，好像没啥问题。。。

## 5.2 显示模块

这次的显示模块和上次一模一样，不需要做任何改动。不过为了方便演示，我加入了一个暂停功能：

以 sel 信号作为调整时钟快慢的信号，以 stop 信号来控制是否暂停。当 stop 为 1 时，直接将 0 作为时钟信号传入即可。

# 6 申 A 理由

- 实现了所有的基本指令
- 增加了移位指令 sll,srl,sra
- 增加了与函数调用相关的跳转指令 jal,jr
- 增加了乘法指令 mul
- 为乘法指令和函数调用指令添加了相应的测试样例