# 实验四 流水线MIPS处理器设计

在实验三中已介绍过MIPS传统的五级流水线阶段，分别是取指、译码、执行、访存、回写(Instruction Fetch, Decode, Execution, Memory Request, Write Back)，五阶段。

单周期CPU虽然CPI为1，但由于时钟周期取决于时间最长的指令(如lw、sw)没有很好的性能，而多周期虽然能提升性能但仍旧无法满足当今处理器的需求。流水线能够很好地解决效率问题，通过分阶段，达到指令的并行执行。同时，在单周期的基础上，能够**很容易地使用触发器做阶段分隔**，实现流水线。

本次实验将从实验三单周期处理器过渡至五级流水，并将解决冒险(hazard)问题。

## 4.1 实验目的

（1）掌握流水线(Pipelined)处理器的思想；

（2）掌握单周期处理中执行阶段的划分；

（3）了解流水线处理器遇到的冒险；

（4）掌握数据前推、流水线暂停等冒险解决方式。

## 4.2实验设备

PC机一台；

计算机系统能力培养实践平台（MINISYS定制开发板）

或：Nexys4 DDR实验开发板；

Xilinx Vivado开发套件(2017.x版本)。

## 4.3实验项目内容

阅读实验原理实现以下模块：

1. Datapath，所有模块均可由实验三复用，需根据不同阶段，修改mux2为mux3(三选一选择器)，以及带有enable(使能)、clear(清除流水线)等信号的触发器。
2. Controller，其中main decoder与alu decoder可直接复用，另需增加触发器在不同阶段进行信号传递
3. 指令存储器inst\_mem(Single Port Rom)，数据存储器data\_mem(Single Port Ram)；同实验三一致，无需改动；
4. 参照实验原理，在单周期基础上加入每个阶段所需要的触发器，重新连接部分信号。实验给出top文件，需兼容top文件端口设定。
5. 实验给出仿真程序，最终以仿真输出结果判断是否成功实现要求指令。

## 4.4实验原理

**4.4.1 单周期改流水线原理**

实验3已完成图中单周期的部分，可以看到，**流水线处理器的主要改动，是在每个执行阶段加入触发器，使得每个周期执行一个阶段，得到的结果送往下一个周期进行执行，同时下一条指令执行一个阶段，这样能够使指令各阶段并行执行，提升效率**。



可以看到，单周期中写寄存器堆的地址信号writereg需要延迟到writeback阶段与回写数据result一起写回寄存器堆：



在此基础上，datapath的基本通路已经形成，下面加入控制器部分。控制器部分与单周期相同，仍然由main decoder和alu decoder构成，但由于改为五级流水线后，每一个阶段所需要的控制信号仅为一部分，控制器产生信号的阶段为译码阶段，产生控制信号后，依次通过触发器传到下一阶段，若当前阶段需要的信号，则不需要继续传递到下一阶段：



**4.4.2 各类型触发器的实现**

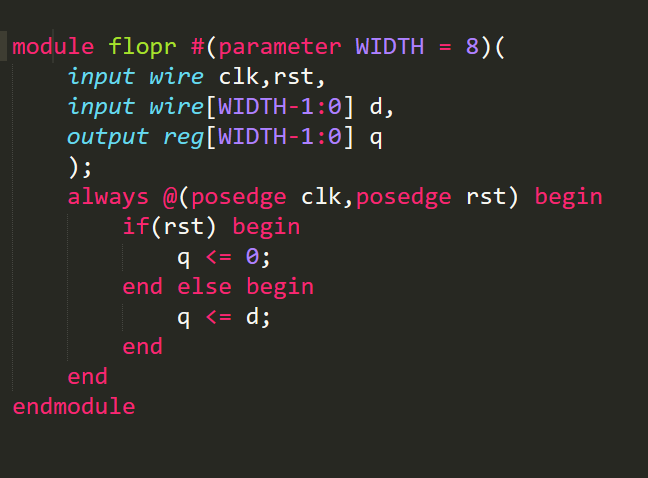
实验3中已给出触发器flopr，作为PC使用，若采用flopr，则需要在其基础上实现下列触发器：

Floprc：带有reset与clear的触发器

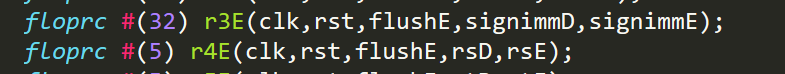
Flopenr：带有enable与reset的触发器

Flopenrc：带有enable、reset与clear的触发器

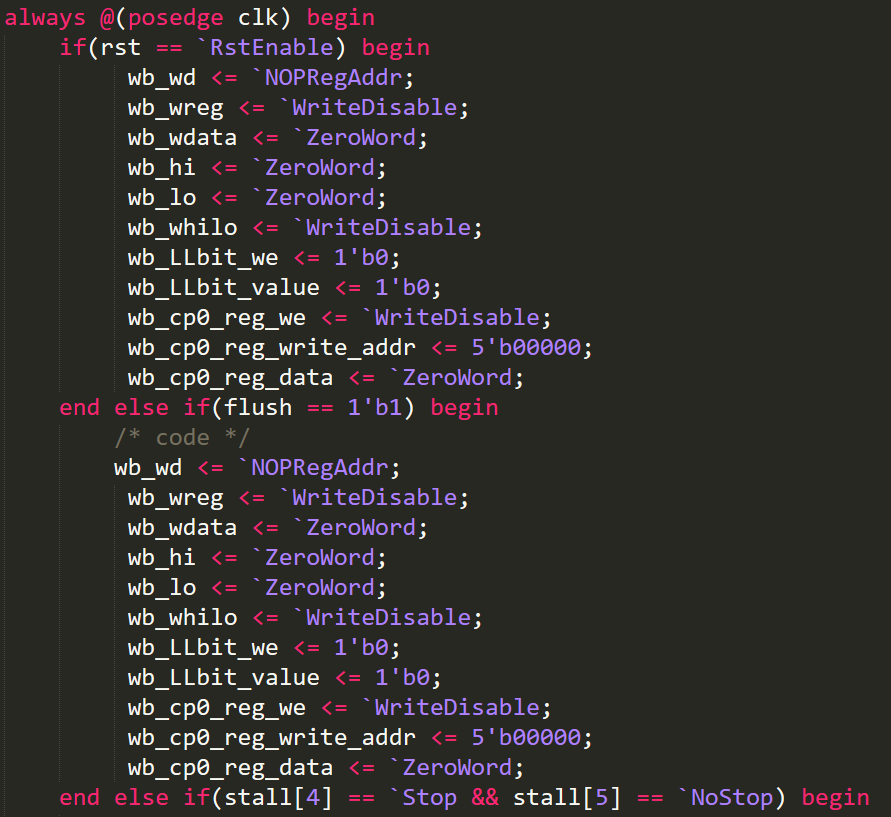
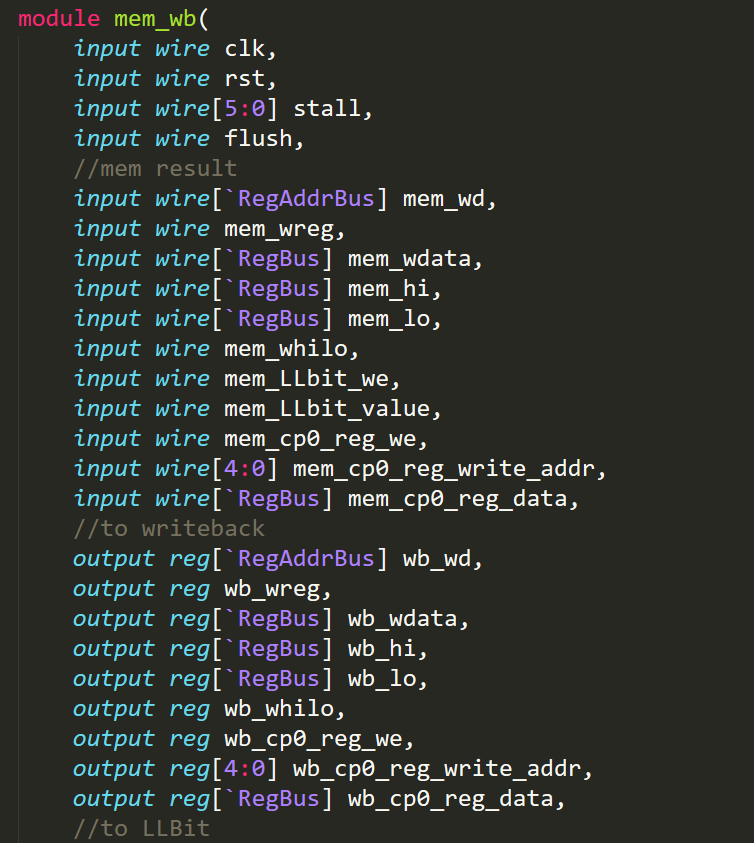
Flopr的写法如下图：



注意#(parameter WIDTH = 8) 的写法，这样写，在调用时可以指定宽度：



此类触发器的优点在于可多次复用，且每次针对单一信号，只需要定义好宽度即可。此外还有另外一种实现方式，可将全部信号写入同一个触发器当中，每个阶段同时传递信号：



这种写法的原理与单一信号的触发器无差别，可以涵盖两个阶段所需要传递的所有信号，与原理图更贴近，但需要熟练掌握所有信号的传递路径，调试阶段更加繁琐。

两者实现效果相同，根据需求自由选择。

**4.4.3 冒险(hazard)的解决**

在流水线CPU中，并不是能够完全实现并行执行。在单周期中由于每条指令执行完毕才会执行下一条指令，并不会遇到冒险问题，而在流水线处理器中，由于当前指令可能取决于前一条指令的结果，但此时前一条指令并未执行到产生结果的阶段，这时候，就产生了冒险。

冒险分为：

数据冒险：寄存器中的值还未写回到寄存器堆中，下一条指令已经需要从寄存器堆中读取数据；

控制冒险：下一条要执行的指令还未确定，就按照PC自增顺序执行了本不该执行的指令(由分支指令引起)。

**4.4.3.1 数据冒险**



分析图中指令，and、or、sub指令均需要使用$s0中的数据，然而add指令在回写阶段才能写入寄存器堆，此时后续三条指令均已经过或正在执行译码阶段，得到的结果均为错误值。

以上就是数据冒险的特点，数据冒险有以下解决方式：

1. 在编译时插入空指令；
2. 在编译时对指令执行顺序进行重排；
3. 在执行时进行数据前推；
4. 在执行时，暂停处理器当前阶段的执行，等待结果。

由于我们未进行编译层的处理，需要在运行时(run time)进行解决，故采用3、4解决方案。

**数据前推**



从图中可以看到，add指令的结果在execute阶段已经由ALU计算得到，此时可以将alu得到的结果直接推送到下一条指令的execute阶段，同理，后续所有的阶段均已有结果，可以向对应的阶段推送，而不需要等到回写后再进行读取，达到数据前推的目的。

数据前推的实现逻辑如下:

**if ((*rsE* != 0) AND (*rsE* == *WriteRegM*) AND *RegWriteM*)**

**then *ForwardAE* = 10**

**else if ((*rsE* != 0) AND (*rsE* == *WriteRegW*) AND *RegWriteW*)**

**then *ForwardAE* = 01**

**else *ForwardAE* = 00**

结合实现逻辑，观察下图。在execute阶段需要判断当前输入ALU的地址是否与其他指令在此时执行的阶段要写入寄存器堆的地址相同，如果相同，就需要将其他指令的结果直接通过多路选择器输入到ALU中。

此处需要：

1. 增加rs,rt的地址传递到execute阶段，并与冒险模块连接；
2. Memory阶段和writeback阶段要写入寄存堆的地址与冒险模块连接；
3. Memory阶段和writeback阶段的寄存器堆写使能信号regwrite与冒险模块连接；
4. 根据实现逻辑，将生成的forward信号输出，控制mux3选择器。



**流水线暂停**





多数情况下，数据前推能解决很大一部分数据冒险的问题，然而在上图中，lw指令在memory阶段才能够从数据存储器读取数据，此时and指令已经完成ALU计算，无法进行数据前推。

在这种情况下，必须使流水线暂停，等待数据读取后，再前推到execute阶段。



流水线暂停的实现逻辑如下：

***lwstall* =** **((*rsD*==*rtE*) OR (*rtD*==*rtE*)) AND *MemtoRegE***

***StallF* = *StallD* = *FlushE* = *lwstall***

结合实现逻辑，需要完成下列功能:

1. 判断decode阶段rs或rt的地址是否是lw指令要写入的地址；
2. 设置PC、fetch->decode阶段触发器的暂停信号(触发器使能端disable)；
3. Decode->exexcute阶段触发器清除(避免后续阶段的执行，等待完成后方可继续执行后续阶段)。

数据冒险解决后的通路图如下：



**4.4.3.2 控制冒险**

控制冒险是分支指令引起的冒险。在五级流水线当中，分支指令在第4阶段才能够决定是否跳转；而此时，前三个阶段已经导致三条指令进入流水线开始执行，这时需要将这三条指令产生的影响全部清除。



将分支指令的判断提前至decode阶段，此时能够减少两条指令的执行；



在regfile输出后添加一个判断相等的模块，即可提前判断beq：



此时又产生了数据冲突问题，需要增加数据前推和流水线暂停模块；



实现逻辑如下：

* **Forwarding logic:**

***ForwardAD*** = (*rsD* !=0) AND (*rsD* == *WriteRegM*) AND *RegWriteM*

***ForwardBD*** = (*rtD* !=0) AND (*rtD* == WriteRegM) AND *RegWriteM*

* **Stalling logic:**

***branchstall*** = *BranchD* AND *RegWriteE* AND

(*WriteRegE* == *rsD* OR *WriteRegE* == *rtD*)

OR *BranchD* AND *MemtoRegM* AND

(*WriteRegM* == *rsD* OR *WriteRegM* == *rtD*)

*StallF* = *StallD* = *FlushE* = *lwstall* OR *branchstall*

**附录A**

实验所附的coe文件中所有指令均包含于下表中，可供查询opcode及funct所代表的具体指令。

表3.1 MIPS的31种指令

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **助记符** | **指 令 格 式** | | | | | | **示 例** | **示例含义** | **操作及解释** |
| BIT # | 31..26 | 25..21 | 20..16 | 15..11 | 10..6 | 5..0 |  |  |  |
| **R-类型** | **op** | **rs** | **rt** | **rd** | **shamt** | **func** |  |  |  |
| add | 000000 | rs | rt | rd | 00000 | 100000 | add $1,$2,$3 | $1=$2+S3 | (rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1 |
| sub | 000000 | rs | rt | rd | 00000 | 100010 | sub $1,$2,$3 | $1=$2-S3 | (rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1 |
| and | 000000 | rs | rt | rd | 00000 | 100100 | and $1,$2,$3 | $1=$2&S3 | (rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1 |
| or | 000000 | rs | rt | rd | 00000 | 100101 | or $1,$2,$3 | $1=$2|S3 | (rd)←(rs) | (rt); rs=$2,rt=$3,rd=$1 |
| slt | 000000 | rs | rt | rd | 00000 | 101010 | slt $1,$2,$3 | if($2<$3)  $1=1 else  $1=0 | if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1 |
| **I-类型** | **op** | **rs** | **rt** | **immediate** | | |  | | |
| addi | 001000 | rs | rt | immediate | | | addi $1,$2,10 | $1=$2+10 | (rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2 |
| lw | 100011 | rs | rt | offset | | | lw $1,10($2) | $1=Memory[  $2+10] | (rt)←Memory[(rs)+(sign\_extend)offset],  rt=$1,rs=$2 |
| sw | 101011 | rs | rt | offset | | | sw $1,10($2) | Memory[  $2+10] =$1 | Memory[(rs)+(sign\_extend)offset]←(rt),  rt=$1,rs=$2 |
| beq | 000100 | rs | rt | offset | | | beq $1,$2,40 | if($1=$2)  goto PC+4+40 | if ((rt)=(rs)) then (PC)←(PC)+4+( (Sign-Extend) offset<<2), rs=$1, rt=$2 |
| **J-类型** | **op** | **address** | | | | |  | | |
| j | 000010 | address | | | | | j 10000 | goto 10000 | (PC)←( (Zero-Extend) address<<2),  address=10000/4 |

**附录B**

**参加PDF文件**