附录C：指令集设计原则-笔记（C1-C4）

1. Introduction
2. Pipelining

流水线技术是一种利用指令间操作的并行性来将多指令进行重叠执行的工具，目前大多数的处理器都采用了此类流水策略来减少指令的平均执行时间。流水线的每一级是用于完成并行指令的不同组成部分，这些步骤被称作Pipe Stage或是Pipe Segment。

指令流水线的吞吐量（Throughput）是指的“单位时间内流水线完成的任务数或输出的结果数”，类似于汽车装备生产线，指令流水线的吞吐量是由一个指令在流水线上存在的时间所决定的。Processor Cycle是指的指令在流水线上进行移动的最小时间分辨率。由于所有的流水线级在流水线运行一段时间后几乎会处于同时工作的状态，所以Processor Cycle是由其中处理速度最慢的环节来决定的，通常是对应到一个时钟周期（有时候是两个时钟周期，极少数情况下是两个时钟周期以上）。

设计流水线需要平衡流水线每一级的时间开销，在理想情况下，如果流水的每一级都很好地进行了均衡设计，那么在流水线处理器中，每一个指令的时间开销为：



据此可知，流水线的提速倍数就等于流水线级数。但实际上很难均衡，而且还存在

其他的开销，会进一步导致流水延时的扩大。流水线技术是基于指令串行的并行性来实 现指令执行加速即CPI的大幅降低，对于后续编程可见，较为友好。

1. The Basics of a RISC V Instruction Set   
    RISC V是一套load-store架构（chapter 1），有着以下特点：
2. 所有对于数据的操作都是对寄存器数据进行操作，通常会改变整个寄存器（32/64位）；
3. 只有load和store指令会改变内存的数据，分别将数据从内存移入寄存器以及从寄存器移到内存中。一般都是load或者store比寄存器空间要小的数据（一个字节/16位数/32位数）。
4. 指令的格式较少，通常所有指令都是相同大小。在RISC V中，rs1，rs2和rd通常都是在同一处来简化控制。

这些特点极大简化了流水线的实现，这也是RSIC的设计初衷。

1. A Simple Implementation of a RISC Instruction Set

以下是利用基于一个RISC架构整型子集（load-store word，branch，以及integer ALU operations）的流水线，来说明pipelined version是如何提高CPI的。所有在RISC指令子集中的指令都可用5个时钟周期来执行，分别是：

* **取指**（Instruction fetch cycle，IF）

该阶段主要是将程序计数器（Computer counter, PC）送至内存，并从其中取出当前指令。由于指令都是4个字节长度，所以需要通过给PC加4来将PC更新到下一个指令的执行。

* **译码**（Instruction decode/register fetch cycle (ID)）

具体而言是：a. 需要对指令进行解码，根据寄存器文件的register source specifier读取寄存器，解码和读取寄存器内容是同时进行的（register specifier在RISC架构中的位置是确定的），这也被称作” fixed-field decoding”。不过此类设计是会导致无效的寄存器访问，一般不会造成性能的损失或者是浪费能量，但在power-sensitive的设计中也许需要避免；b. 在读取寄存器内容的同时为**潜在的分支**进行equality test；c. 对指令的offset区域进行符号位扩展，以备不时之需；d. 通过对PC加入扩展符号补偿来计算可能的分支目标地址；e. 对于load和ALU直接操作，immediate field会是同一处，对其可以进行轻松的扩展；

* 执行（Execution/effective address cycle, EX）

在译码阶段进行准备的ALU操作通常是执行1到3个函数功能，取决于指令类型，如下所示。

内存引用（Memory Reference）：ALU结合基址寄存器和偏移量来获得有效的地址；寄存器-寄存器ALU指令（Register-Register ALU instruction）：ALU对从寄存器文件中读取的值进行ALU操作码对应的操作；寄存器-立即数ALU指令（Register-Immediate ALU instruction）ALU根据从寄存器文件中读到的第一个值以及有符号扩展的立即数，来进行执行相应的操作码；条件转移（Conditional branch）：判别当前条件是否为真。

在load-store架构中，鉴于不存在指令同时计算一个数据地址和对该数据进行操作，所以**有效地址**和**执行周期**可以合并为单个**时钟周期**。

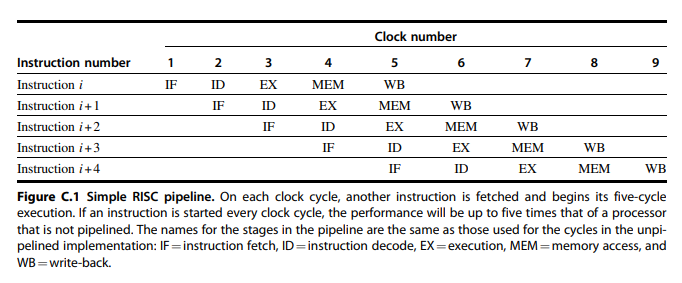
* 内存访问（Memory access, MEM）

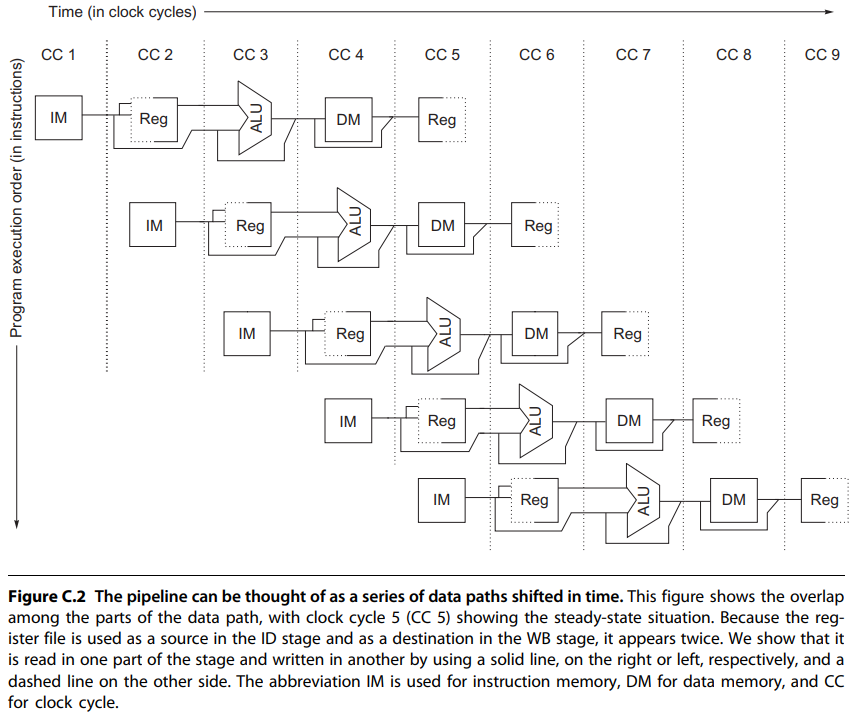
Load指令则对应利用前一周期计算得到的有效地址进行数据的读取，如果是Store指令那么内存会利用有效地址，记录第二个从寄存器栈中读到的寄存器的数据。

* 写回（Write-back cycle, WB）

此阶段对应的是Register-Register ALU instruction 或者load instruction，将来自内存的结果（load instruction）或者是来自ALU的结果协会到寄存器栈中。

其中，branch指令需要三个周期，store指令需要四个周期，其他指令需要五个周期。假设分支有12%的概率出现，store有10%的概率，一个较为经典的指令分布就会得到整体为4.66的CPI。但此时的设置并未达到最优，无论是性能还是给定性能后的硬件开销，需要进一步思考如何pipeline进行均衡设计。

1. The Classic Five-Stage Pipeline for a RISC Processor  
   

表格C.1显示了典型的流水线模式。为了实现这些流水设计，需要确保在同一时刻不会出现对于同一数据通路的两个不同操作，例如ALU不可以同时计算有效地址和执行剪发运算。所以需要保证在流水线中的指令重叠不会造成此类冲突，而RISC的简洁性使得资源分析变得相对简单。  


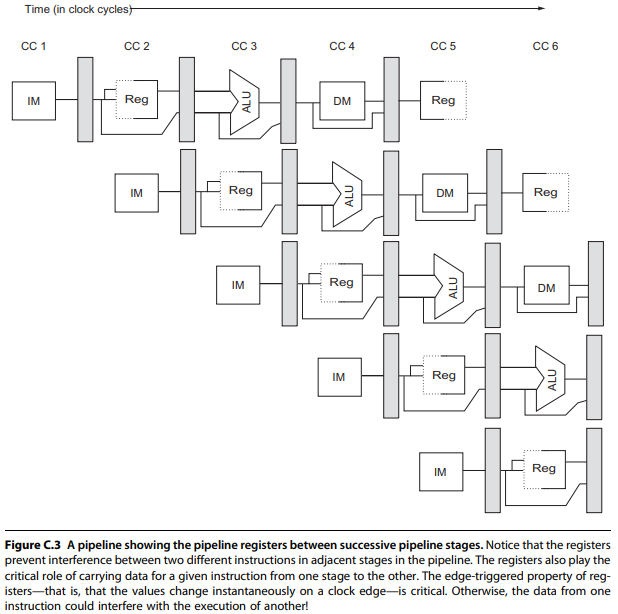
C2表示的是简化后的流水线式RISC数据通路，主要的功能部件都是用在不同时钟周期，因此引入的冲突较少。

其中主要利用了不同的指令和数据存储即数据缓存。分立的cache消除了单个内存存在的**取指和数据访存之间的冲突**。注意到如果该流水线处理器的时钟和非流水线式的一致，那么由于高性能的需求，内存将会需要五倍于非流水式的带宽。

其次，需要寄存器栈是用于两级：一级是在ID阶段的读数（作为源寄存器，图中的实线），一级在WB阶段的写数（作为目的寄存器，图中的虚线）。为了解决同一个寄存器的读写问题，在前半个时钟周期进行寄存器写，后半个周期进行寄存器读。

此外，需要在C2的IF阶段加入对PC的操作，为下一条指令做准备。此外也需要用一个加法器在ID阶段去计算潜在的分支目标（potential branch target），但在ID阶段前都不会对PC操作，会造成一定问题。

最后，也需要确保在不同流水级的指令不会互相干扰，这是通过在不同级间引入流水寄存器（pipeline register）来实现的，在一个时钟末尾，一级操作后的所有结果都会放入该寄存器中，作为下一级的输入，如图C3所示。尤其是当源寄存器和目的寄存器之间不直接相邻时。



例如，在store指令期间需要存储的数据在ID阶段才读取，但一直到MEM阶段才会进行使用。该数据需要经过两级流水寄存器才能到达data memory；在EX阶段的ALU指令结果到WB才进行存储，也是需要经过两级流水寄存器。通常流水寄存器的命名方式是按照：“前一流水级名称/后一流水级名称”。

1. Basic Performance Issues in Pipelining

流水线可以带来CPU指令吞吐量的大幅提升，整体来看系统的**程序执行效率**得到了飞升，但由于引入了控制逻辑的开销，实际上每一个指令的用时反而还增加少许，从而限制了流水线的深度（the depth of pipeline）。

由于流水线的最高频率不能高出流水线中最慢的一级，所以其不均衡负载会导致系统频率瓶颈，而且流水线开销来自于流水寄存器延时和时钟偏移（clock skew，即为时钟到达任意两个寄存器时钟端口的时差）；其中流水寄存器会增加建立时间（setup time）,而且会存在时钟在流水线上的传输延时，这些都是造成时钟偏移的重要元素。一旦时钟周期小于时钟偏移和锁存开销，此时流水线已达到瓶颈，系统中不存在可以利用的时间裕度了。【可参考Kunkel and Smith (1986)】

如果改变时钟大小无法减小additional overheads to clock，那么根据**Amdahl定律**可知该overhead限制了系统的加速比。

1. The Major Hurdle of Pipelining—Pipeline Hazards  
    在某些情况下，称为hazards，会阻止指令流中的下一条指令在其指定的时钟 周期内执行。相关会降低通过流水线获得的理想加速性能。

冲突分为三类：

当硬件无法在重叠执行中同时支持所有可能的指令组合时，资源冲突就会引起 Structural hazards。在现代处理器中，结构性相关主要发生在不常用的特殊功能单

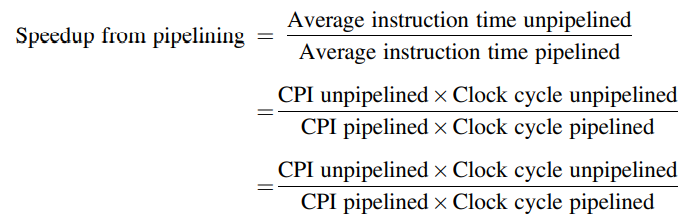
元中（例如浮点除法或其他复杂的长期运行指令），但它们并不是主要的性能因素。

我们没有更多的时间花在这种罕见的情况上，而是将重点放在了其他两种更为

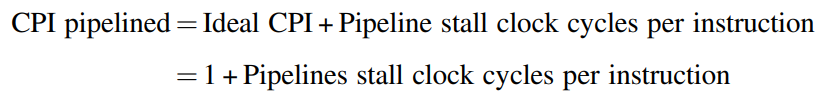
常见的相关上。即Data hazards和Control hazards。前者是当指令以流水线中指 令重叠的方式暴露于前一条指令的结果，数据的错误导致冲突；后者是源自分支流 水和其他改变程序计数器的指令。  
 流水线中的冲突可能会使流水停顿。为避免冲突，通常需要流水线中的某些 指令继续执行，而延迟其他指令。

* 1. Performance of Pipelines with Stalls

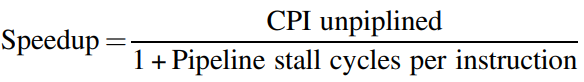
停顿会导致流水线性能从理想性能下降。 让我们看一个简单的方程式，从上一节的公式开始，从流水线中查找实际的加速比：



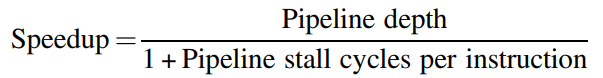
流水线本质在于减少CPI或时钟周期时间。 由于使用CPI来比较流水线是传统做法，因此从该假设开始分析。流水线处理器上的理想CPI几乎始终为1。因此，我们可以计算出流水线CPI：



如果忽略流水线的周期时间开销，并假设阶段是完美平衡的，那么两个处理器的周期时间可以相等，从而导致：



在一种重要的简化情形——所有指令占用相同的周期数，这些周期也必须等于流水线级数（也称为流水线深度）中，未流水的CPI等于流水线的深度，导致：



如果没有流水停顿，直观上会有这样的结论：可以通过流水深度来提高流水线性能。

* 1. Data Hazards

流水线的主要作用是通过重叠执行来改变指令的相对时序。这种重叠会带来数据和控制相关。当流水线更改对操作对象的读/写访问的顺序，从而使该顺序不同于非流水处理器顺序指令所见顺序时，就会发生数据冲突。假设指令i在指令j之前以程序顺序出现，并且两个指令都使用寄存器x，那么在i和j之间可能发生三种不同类型的相关：

* 写后读相关（. Read After Write (**RAW**) hazard）

最常见的情况：在指令i写入寄存器x之前，指令j读取寄存器x的值。如果无法避免这种相关，则指令j将使用错误的x值。

* 读后写相关（Write After Read (**WAR**) hazard）

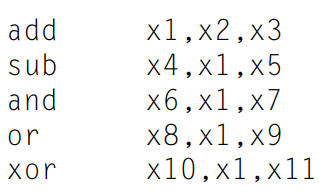
当指令i写入寄存器x后发生指令i读取寄存器x时，会发生这种相关。 在这种情况下，指令i将使用错误的x值。在五指令简单整型流水中，WAR相关不可能发发生，但当指令重新排序时除外，如C.65所示

* 写后写相关（Write After Write (**WAW**) hazard）

当指令i写入寄存器x之后，指令i写入寄存器x时，就会发生这种相关。 此情况下，寄存器x传递的值是错误的。在五指令简单整型流水中，WAR相关同样不可能的，但是当重新排序指令或运行时间变化时就会发生WAR相关。

* 一个例子

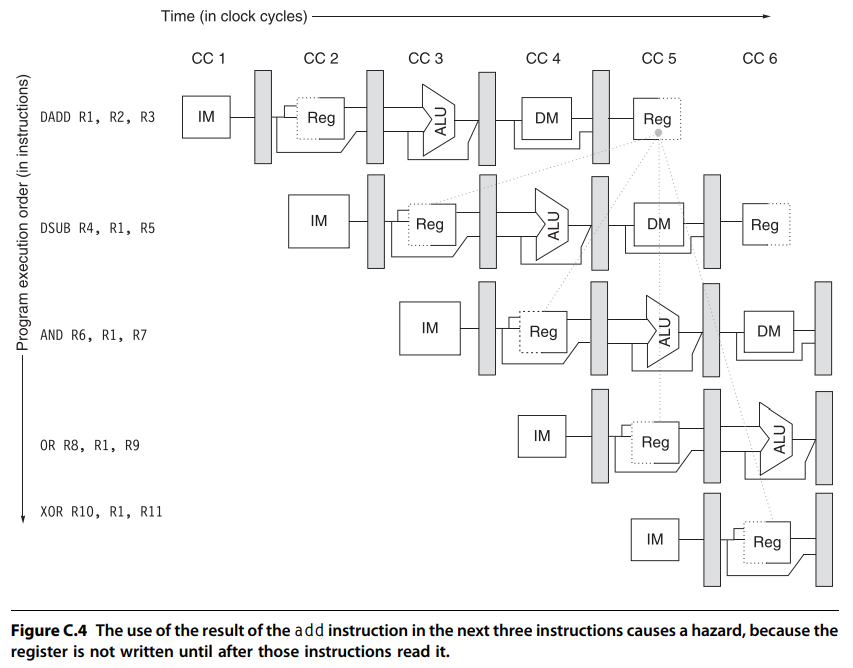
考虑以下指令的执行：



加法之后的所有指令均使用加法指令的结果。如图C.4所示，加指令在WB阶段写入x1的值，而子指令在其ID阶段读取x1的值，这会导致RAW相关。除非采取预防措施，否则子指令将读取错误的值并尝试使用它。实际上，子指令使用的值甚至不是确定性的：如果在添加指令和子指令之间发生中断，则添加的WB阶段将完成，此时x1的值将是添加的结果。这种不可预测的行为显然不可取。

and指令还可能造成RAW相关。 从图C.4中可以看到，对x1的写入要到时钟周期5结束时才能完成。因此，在第4个时钟周期内读取寄存器的and指令将获得错误结果。

xor指令正确运行，因为其寄存器读取是在寄存器写入后的第六个时钟周期中。or指令的运行也不会造成相关，因为在周期的后半部分执行**寄存器文件读取**，而在前半部分执行**写操作**。请注意，xor指令仍取决于加法，但不再产生相关。【详见chapter3】



* 1. Minimizing Data Hazard Stalls by Forwarding

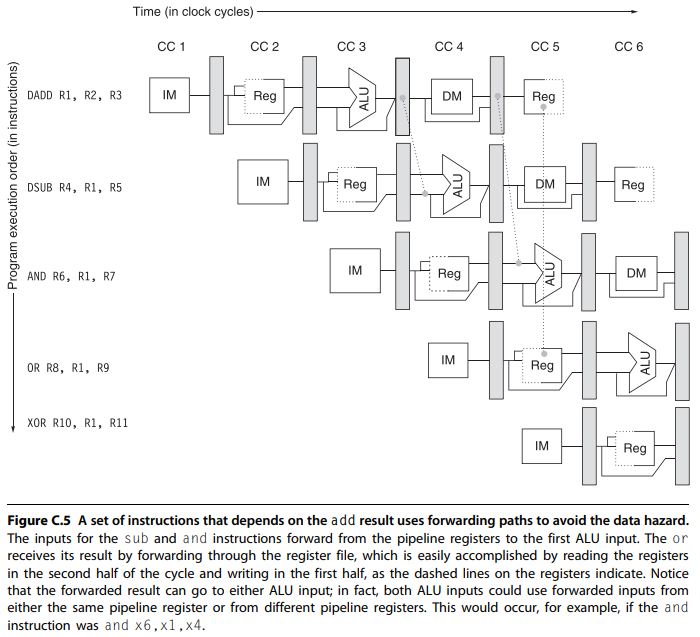
本节将会讲到消除涉及sub和and指令的延时。

图C.4中提出的问题可以通过称为转发（**forwarding**，也称为旁路**bypassing**，有时也称为短路）的简单硬件技术来解决。转发的核心思想在于，直到添加真正产生结果之后，子程序才真正需要结果。如果可以将结果从添加存储它的流水线寄存器移动到子程序需要它的位置，那么可以避免停顿的需要。据此，转发的工作方式如下：

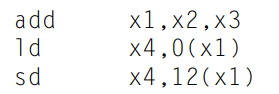
* 来自EX / MEM和MEM / WB流水寄存器的ALU结果始终反馈到ALU输入。
* 如果转发硬件检测到先前的ALU操作已写入与当前ALU操作的源相对应的寄存器，则控制逻辑会选择转发的结果作为ALU输入，而不是从寄存器文件中读取的值。

对于转发，如果子程序停滞，则添加将完成，并且旁路不会激活。对于两指令之间的中断，这种关系也适用。

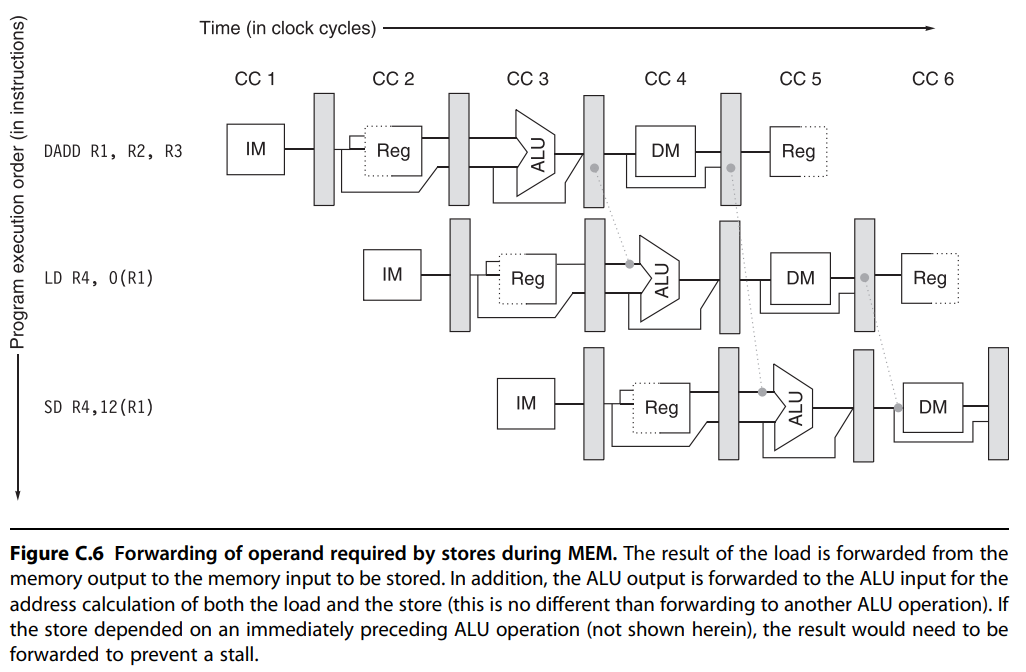
如图C.4中的示例所示，不仅需要转发紧接在前一条指令的结果，而且还可能需要转发从两个周期之前开始的一条指令的结果。图C.5显示了我们的示例，其中有旁路路径，并突出显示了寄存器读写的时序。该代码序列可以不停顿地执行。



转发可以概括为包括将结果直接传递到需要它的功能单元：将结果从与一个单元的输出相对应的流水线寄存器转发到另一单元的输入，而不仅仅是从一个单元的结果转发到另一单元的输入。同一单元的输入。以下面的序列为例：

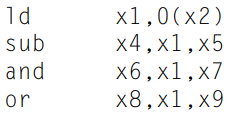


为避免此顺序中的停顿，需要将ALU输出和存储单元输出的值从流水线寄存器转发到ALU和数据存储输入。图C.6显示了此示例的所有转发路径。



* 1. Data Hazards Requiring Stalls

然而并非所有潜在的数据相关都可以通过绕过来处理。可以考虑以下指令序列：

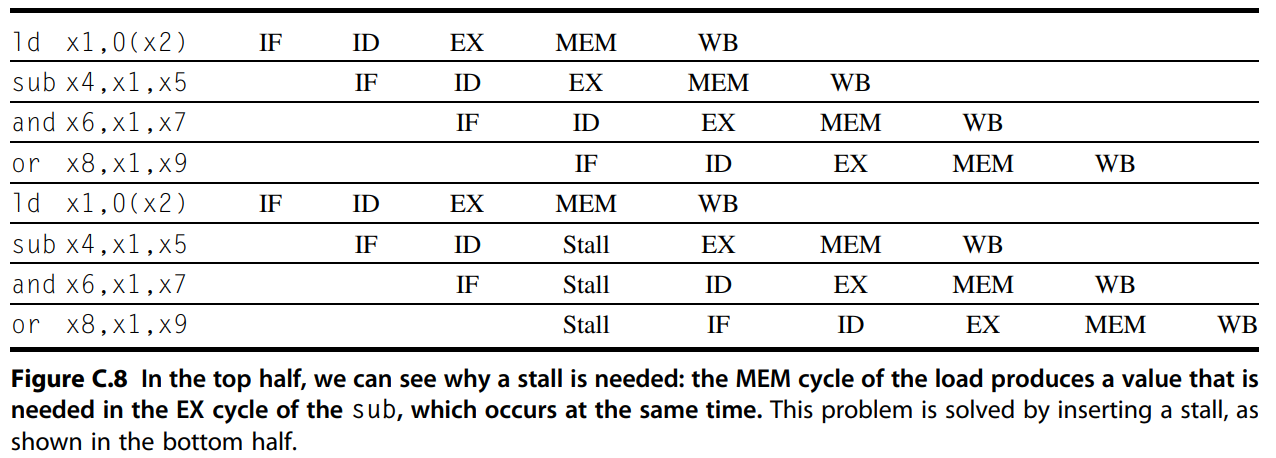


此示例的带有旁路路径的流水线数据路径如图C.7所示。这种情况与背靠背ALU操作的情况不同。ld指令直到第四个时钟周期结束（其MEM周期），才具有数据，而sub指令需要在该时钟周期的开始时具备数据。因此，使用简单的硬件无法完全消除使用加载指令带来的数据相关。

如图C.7所示，这样的转发路径必须在时间上向后运行（不过没法实现），但实际上我们可以将结果立即从流水线寄存器转发到ALU，以用于and运算（从2个时钟开始）加载后循环。同样，or指令也没有问题，因为它通过寄存器文件接收值。对于子指令，转发结果到达时太晚了——在一个时钟周期的末尾，但当开始时就需要它了。

加载指令具有无法通过单独转发消除的延迟或延迟。相反，我们需要添加称为流水线互锁的硬件，以保留正确的执行模式。通常，流水线互锁会检测到相关，并使流水线停顿，直到清除相关为止。在这种情况下，互锁使流水线停滞，从要使用数据的指令开始，直到源指令产生数据为止。正如结构相关一样，这种流水线互锁会导致停转或气泡。 停顿指令的CPI增加了停顿的长度（在这种情况下为1个时钟周期）。

图C.8使用流水线阶段的名称显示了停顿前后的流水线。由于停顿导致从子对象开始的指令在时间上较晚移动一个周期，因此现在转发到and指令是通过寄存器文件，而or指令根本不需要转发。气泡的插入导致完成此序列的循环数增加一。 在第四个时钟周期内没有任何指令开始（第六个周期内没有任何指令完成）。



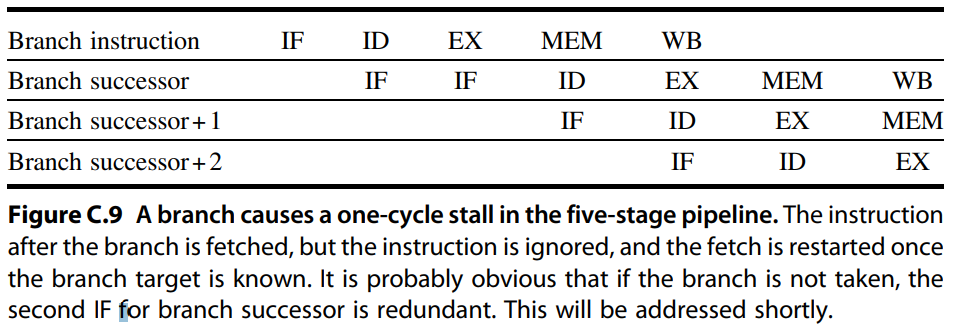
* 1. Branch Hazards

与数据相关相比，控制相关会给我们的RISC V流水线造成更大的性能损失。 当执行分支时，它可能会或可能不会将PC更改为当前值加4以外的值。回想一下，如果分支将PC更改为其目标地址，则它是一个采用的分支；如果通过，则不采取。

如果指令i是执行的分支，那么在完成地址计算和比较之后，通常直到ID阶段完成时才对PC进行更改。

图C.9显示了处理分支的最简单方法是，一旦我们在ID（解码指令）期间检测到分支，就在分支之后重做对指令的获取。第一个IF周期实质上是一个停顿，因为它从不执行有用的工作。如果未采用该分支，那么就不需要重复IF阶段，因为正确的指令的确已成功被取到。

每个分支一个停顿周期会导致10％到30％的性能损失，具体取决于分支频率，因此需要研究一些技术来解决这种损失。



* 1. Reducing Pipeline Branch Penalties

处理分支延迟引起的流水线停顿的方法有很多。在本小节中，我们讨论了四个简单的编译时间方案。在这四个计划分支的动作是静态的，在每个分支期间它们在整个执行过程都是固定的。该软件通过硬件方案和分支行为经验，尝试使用以下方法来最小化分支惩罚。动态预测分支行为是基于硬件的方案，第3章研究是针对更强大的动态分支预测的硬件技术。

处理分支的最简单方案是冻结或刷新流水线，在分支之后保留或删除任何指令，直到知道分支目标为止。该解决方案的吸引力主要在于它对应的简洁硬软件。图C.9所示的是流水线中较早使用的解决方案。在这种情况下，分支惩罚是固定的，不能通过软件设计来降低。一种性能更高且仅稍微复杂一点的方案是将每个分支都视为未采用，仅允许硬件像未执行分支一样继续运行。除非明确知道分支结果，否则不要更改处理器状态。该方案的复杂性是来源于必须明晰何时一条指令可以更改状态以及如何“撤消”这种更改。

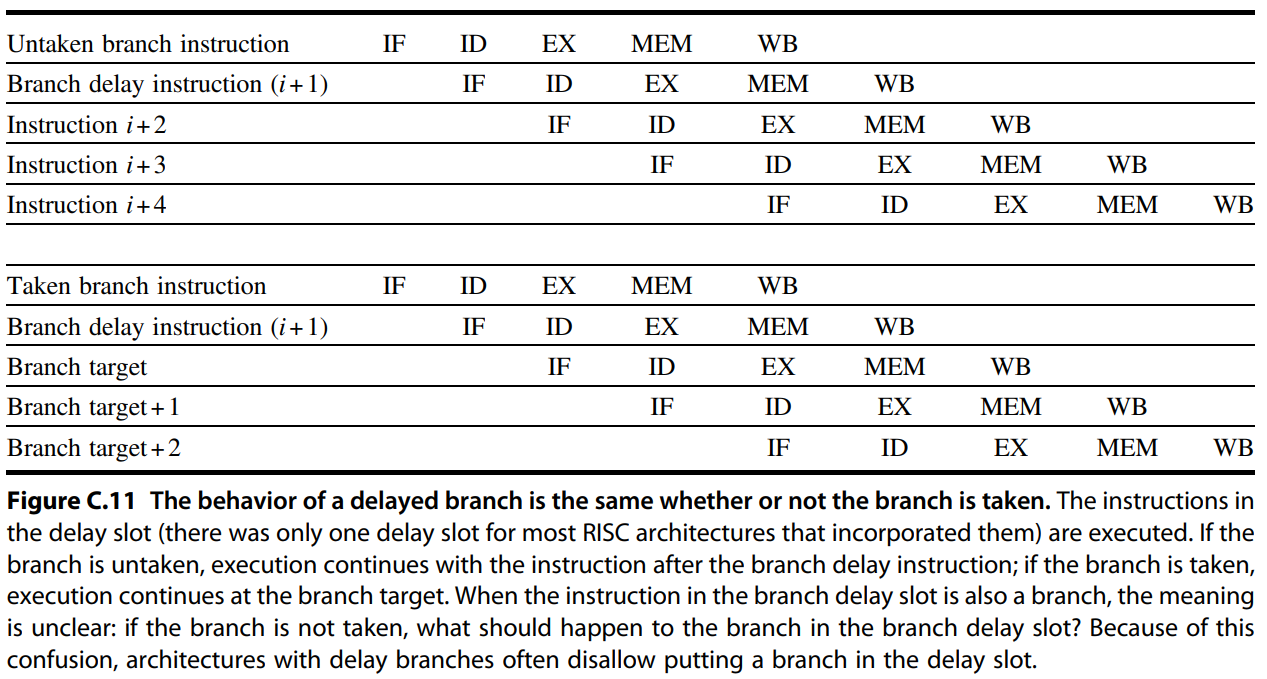
在简单的五阶段流水线中，此预测未采用方案是通过继续提取指令来实现的，就好像分支是普通指令一样。流水线看起来好像什么都没有发生。但是，如果采用分支，则需要将获取的指令转换为无操作，然后在目标地址处重新开始获取。图C.10显示了两种情况。

另一种方案是将每个分支视为已采用。一旦分支被解码并计算了目标地址，我们就假定要采用该分支，并开始在目标处进行获取和执行。执行分支相当于一个单周期的改进，因为已知ID末尾的目标地址，即在知道ALU阶段是否满足分支条件之前的一个周期。无论采用预测采用还是未采用预测方案，编译器都可以通过组织代码来提高性能，以便最频繁的路径与硬件选择相匹配。

在早期RISC处理器中大量使用的第四个方案称为延迟分支。在延迟分支中，分支延迟为1的执行周期有：

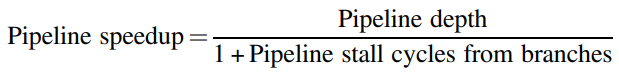
顺序后继者在分支延迟时隙中。无论是否采用分支，都会执行该指令。具有分支延迟的五级流水线的流水行为如图C.11所示。尽管分支延迟可能长于一个，但实际上几乎所有具有延迟分支的处理器都具有单个指令延迟。如果流水线具有更长的潜在分支代价，则可以使用其他技术。编译器的工作是使后续指令有效且有用。

尽管在硬件预测过于昂贵的时候，延迟的分支对于短的简单流水线很有用，但是当存在动态分支预测时，该技术会使实现变得复杂。因此，RISC V适当地省略了延迟分支。



* 1. Performance of Branch Schemes

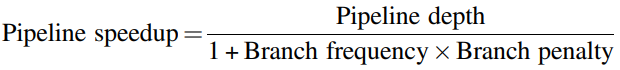
本节是关于每个方案的有效表现。假设理想CPI为1，则有效的流水线加速比与分支惩罚是：



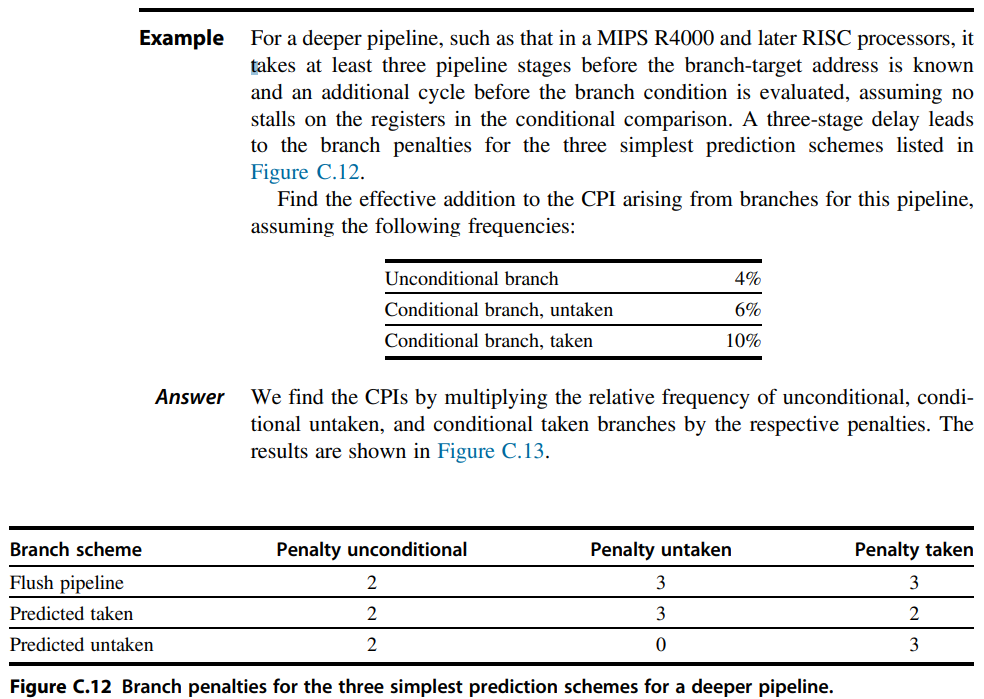
由于下式：



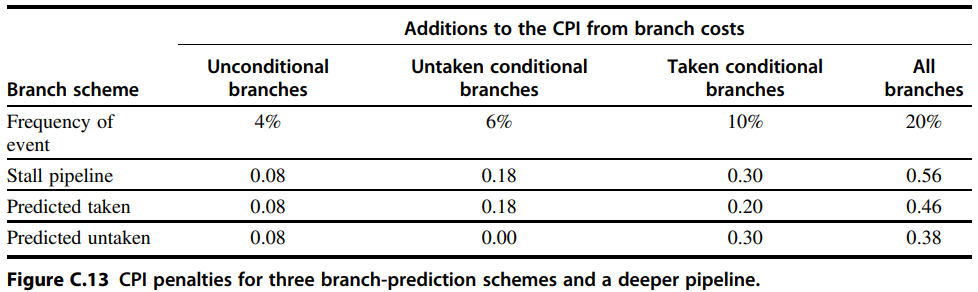
因此可得：



分支频率和分支惩罚可以具有来自无条件分支和有条件分支的分量。 但后者占主导地位，因为其出现频繁更高。



方案之间的差异随着此较长的延迟而大大增加。如果基本CPI为1，分支是唯一的停顿来源，那么理想的流水线将比使用停滞流水线方案的流水线快1.56倍。在相同的假设下，预测未采用方案将比停滞流水线方案好1.13倍。

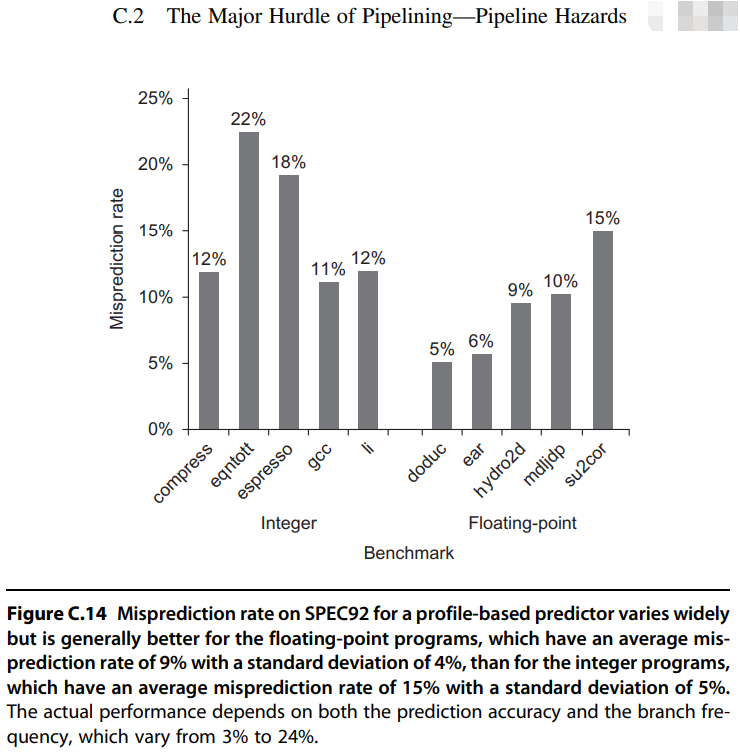


* 1. Reducing the Cost of Branches Through Prediction

随着流水线越来越深，分支的潜在代价增加，使用延迟分支和类似方案变得不够。 相反，我们需要使用更具侵略性的手段来预测分支。 此类方案分为两类：低成本 静态方案，其依赖于编译时可用的信息时间和基于程序行为动态预测分支的策略。 我们在这里讨论这两种方法。

* 1. Static Branch Prediction  
      改进编译时分支预测的关键方法是使用从较早运行中收集的配置文件信息。 使这一点有价值的主要观察结果是分支的行为通常是双峰分布的。也就是说，单个分支通常高度偏向已采用或未采用。图C.14显示了使用该策略的分支预测成功。 相同的输入数据用于运行和收集配置文件。其他研究表明，更改输入以使配置文件用于不同的运行会导致基于配置文件的预测准确性仅发生很小的变化。

任何分支预测方案的有效性都取决于方案的准确性和条件分支的频率，这在SPEC中从3％到24％不等。 整数程序的误预测率较高并且这样的程序通常具有较高的分支频率这一事实是静态分支预测的主要限制。下一步将考虑动态分支预测器，最近的处理器已经采用了这种预测器。



* 1. Dynamic Branch Prediction and Branch-Prediction Buffers

最简单的动态分支预测方案是分支预测缓冲区或分支历史记录表。 分支预测缓冲区是由分支指令地址的下部索引的小存储器。内存中包含一些信息，说明分支是否最近使用。这种方案是最简单的缓冲区。它没有标签，仅在比计算可能的目标PC所需的时间长时才可用于减少分支延迟。

有了这样的缓冲区，但实际上不确定预测是否正确，它可能已经由具有相同低位地址位的另一个分支放在了那里。该预测是用来预示假定正确，并且沿预测的方向开始取指。如果提示错误，则将预测位取反并存储回去。

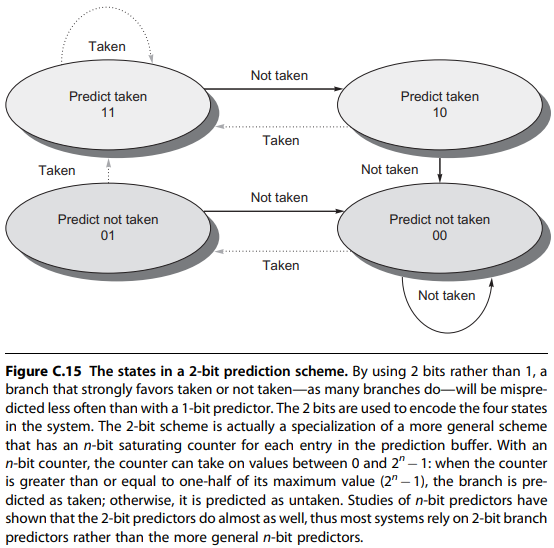
该缓冲区实际上是每次访问都命中的高速缓存，并且，正如我们将看到的，缓冲区的性能取决于所关注分支的预测频率和匹配时预测的准确性。 在分析性能之前，有必要对分支预测方案的准确性进行小而重要的改进。

这种简单的1位预测方案存在性能缺陷：即使几乎总是采用分支，由于不正确的预测会导致预测位被翻转，因此，即使不采用分支，也可能会两次错误地进行预测。

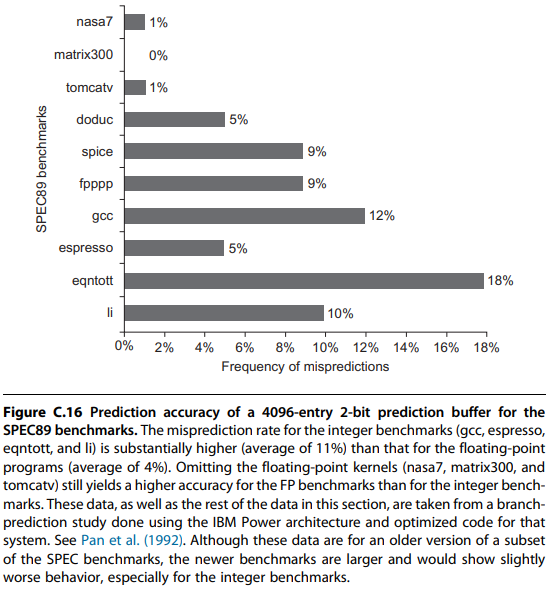
为了弥补这一缺点，经常使用2位预测方案。 在2位方案中，预测必须更改两次才能错过。 图C.15显示了用于2位预测方案的有限状态处理器。

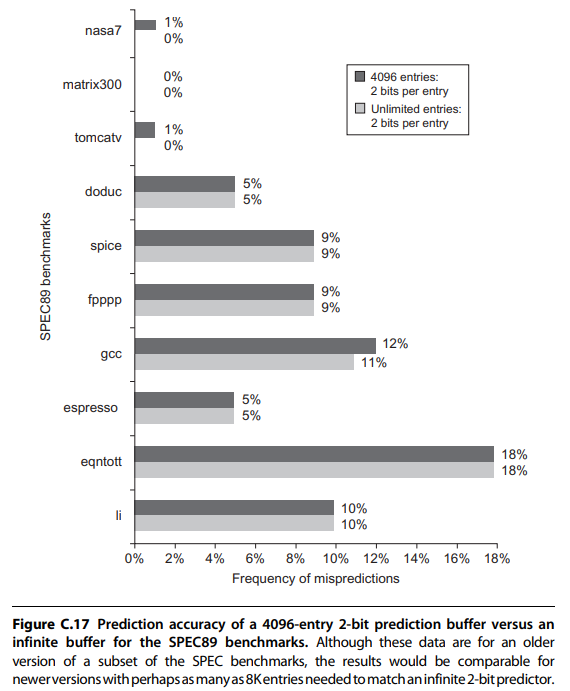
分支预测缓冲区可以实现为在IF流水线阶段通过指令地址访问的小型特殊“缓存”，也可以实现为一对附加到指令缓存中每个块并随指令提取的位。

如果指令被解码为分支，并且如果分支被预测为已采用，则只要知道PC，就从目标开始提取。否则，顺序提取和执行将继续。 如图C.15所示，如果预测结果错误，则更改预测位。在实际应用中，使用每项2位的分支预测缓冲区的精度如图C.16显示，对于SPEC89基准测试，具有4096个条目的分支预测缓冲区可提高预测精度范围从99％到82％，或者错误预测率为1％–18％。 像用于这些结果的4K输入缓冲区一样，在2017年被认为很小，而较大的缓冲区可能会产生更好的结果。



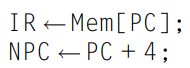
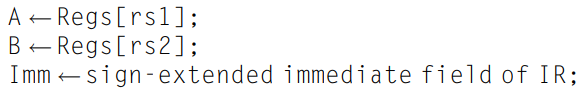
随着我们尝试利用更多的ILP，分支预测的准确性变得至关重要。 如图C.16所示，整数程序（通常也具有较高的分支频率）的预测器的准确性低于循环密集型科学程序的预测器的准确性。 我们可以通过两种方式来解决这个问题：通过增加缓冲区的大小以及通过提高方案的准确性，用于每个预测。但如图C.17所示，具有4K条目的缓冲区的性能与无限缓冲区相当，至少对于SPEC中的基准而言是如此。图C.17中的数据清楚地表明，缓冲区的命中率不是主要的限制因素。如前所述，仅增加每个预测变量的位数而不更改预测变量的结构也几乎没有影响。相反，我们需要研究如何提高每个预测变量的准确性，就像我们在第3章中将要提到的那样。





1. How Is Pipelining Implemented?
   1. A Simple Implementation of RISC V

每个RISC V指令最多可以在5个时钟周期内实现。5个时钟周期如下：

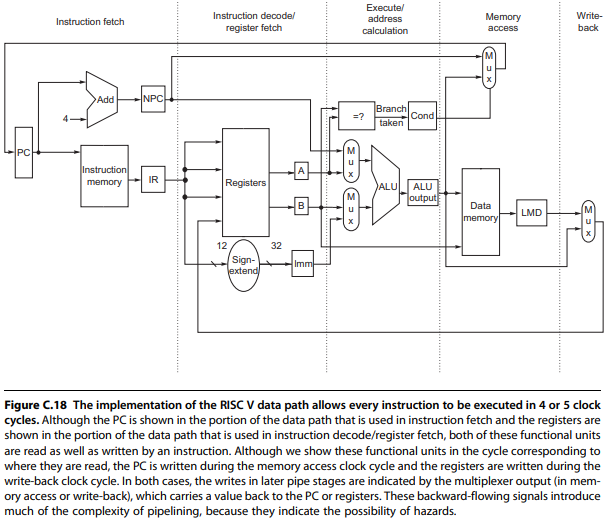
* Instruction fetch cycle (IF)  
  
* Instruction decode/register fetch cycle (ID)  
  
* Execution/effective address cycle (EX)
  + Memory reference  
    
  + Register-register ALU instruction  
    
  + Branch  
    
* Memory access/branch completion cycle (MEM)  
  PC的更新是对于全体指令而言的：PC 🡨 NPC；
  + Memory reference:  
    
  + Branch:  
    
* Write-back cycle (WB)、
  + Register-register or Register-immediate ALU instruction  
    
  + Load instruction:  
    

图C.18显示了指令如何流经数据路径。 在每个时钟周期结束时，在该时钟周期内计算出并在随后的时钟周期（无论是该指令还是下一条指令）所需的每个值都被写入存储设备，该设备可以是存储器，通用寄存器，PC 或临时寄存器（即LMD，Imm，A，B，IR，NPC，ALUOutput或Cond）。 临时寄存器在一个指令的时钟周期之间保存值，而其他存储元件是状态的可见部分，并在连续的指令之间保存值。

尽管当今所有处理器都已流水线化，但这种多周期实现可以合理地近似于大多数处理器在更早的实现的方式。用一个遵循此处显示的五周期结构的简单有限状态机可以用来实现该控制。对于更复杂的处理器，可以使用微码控制。无论哪种情况，都可实现像本节中所述的指令序列都将确定控件的结构。

在此多周期实施中，可以消除一些硬件冗余。例如，有两个ALU：一个用于递增PC，另一个用于有效地址和ALU计算。因为在同一时钟周期上不需要它们，所以可以通过添加其他多路复用器，并共享相同的ALU来合并它们。同样，由于数据和指令的访问发生在不同的时钟周期上，指令和数据可以存储在同一存储器中。

与其优化这种简单的实现，不如保留图C.18中的设计，因为这为流水线实现提供了更好的基础。



* 1. A Basic Pipeline for RISC V

可以通过在每个时钟周期上启动一条新指令来几乎不做任何更改地流水处理图C.18的数据路径。 因为每个流水线级在每个时钟周期都处于活动状态，所以流水线级中的所有操作必须在1个时钟周期内完成，并且任何操作组合都必须能够一次发生。

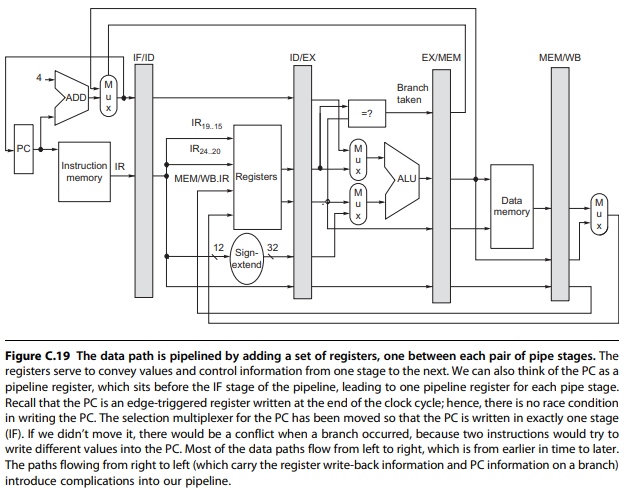
此外，对数据路径进行流水处理需要将从一个流水线级传递到下一个流水线级的值放置在寄存器中。 图C.19显示了RISC V流水线，在每个流水线级之间具有适当的寄存器，称为流水线寄存器或流水线锁存器。 寄存器标有它们连接的阶段的名称。图C.19是通过流水线寄存器从一级到另一级的连接绘制的。

在一条指令的时钟周期之间临时保存值所需的所有寄存器都包含在这些流水线寄存器中。指令寄存器（IR）的字段（是IF / ID寄存器的一部分）在用于提供寄存器名称时被标记。流水线寄存器同时承载数据和

从一个流水线阶段到下一个阶段的控制。后续流水线阶段所需的任何值都必须放置在此类寄存器中，并从一个流水线寄存器复制到下一个流水线寄存器，直到不再需要它为止。如果我们尝试仅使用我们先前的非流水线数据路径中的临时寄存器，则在所有使用完成之前，值可能会被覆盖。例如，用于加载或ALU操作写操作的寄存器操作数的字段是从MEM / WB流水线寄存器而不是IF / ID寄存器提供的。这是因为我们希望装入或ALU操作写入该操作指定的寄存器，而不是当前从IF到ID转换的指令的寄存器字段！此目标寄存器字段只是从一个流水线寄存器复制到下一个流水线寄存器，直到在WB阶段需要它为止。

任何一条指令一次都在流水线的一个阶段中处于活动状态。因此，代表指令采取的任何操作都会在一对流水线寄存器之间发生。因此，我们还可以通过检查取决于指令类型的任何流水线阶段发生的事情来查看流水线的活动。图C.20显示了此视图。命名流水线寄存器的字段是为了显示从一个阶段到下一个阶段的数据流。注意，前两个阶段的动作与当前指令类型无关。它们必须是独立的，因为直到ID阶段结束才对指令进行解码。 IF活动取决于EX / MEM中的指令是否为采用分支。如果是这样，则EX / MEM中分支指令的分支目标地址在IF的末尾写入PC；

否则，增量的PC将被写回。（如前所述，分支的影响导致我们在接下来的几节中处理的流水线中变得复杂。）寄存器源操作数的固定位置编码对于允许在ID期间获取寄存器至关重要。



为了控制这个简单的流水线，我们只需要确定如何为图C.19的数据路径中的四个多路复用器设置控件。根据指令类型设置ALU级中的两个多路复用器，该指令类型由ID / EX寄存器的IR字段决定。通过指令是否为分支来设置顶部ALU输入多路复用器，并且通过指令是寄存器-寄存器ALU操作还是任何其他类型的操作来设置底部多路复用器。

IF级的多路复用器选择使用增量PC的值还是EX / MEM.ALUOutput（分支目标）的值写入PC。该多路复用器由字段EX / MEM.cond控制。第四级多路复用器由WB级中的指令是加载还是ALU操作来控制。除了这四个多路复用器外，还需要一个额外的多路复用器，它未在图C.19中绘制，但从ALU操作的WB阶段看，其存在就很明显。目标寄存器字段位于两个不同的位置之一

取决于指令类型（寄存器-寄存器ALU与立即ALU或加载ALU）。因此，假设指令写入一个寄存器，需要一个多路复用器来选择MEM / WB寄存器中IR的正确部分，以指定寄存器目标字段。

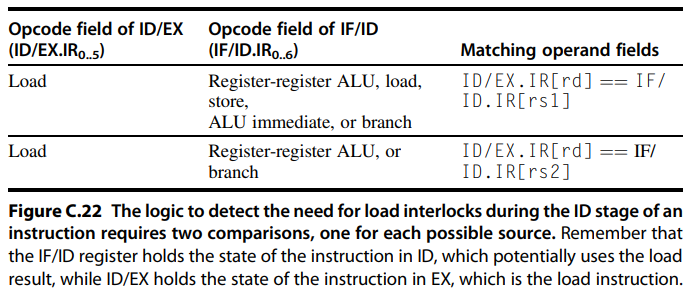
* 1. Implementing the Control for the RISC V Pipeline

让指令从该流水线的指令解码阶段（ID）进入执行阶段（EX）的过程通常称为指令发布；据说已经发出了执行此步骤的指令。对于RISC V整数流水线，可以在流水线的ID阶段检查所有数据相关。如果存在数据相关，则在发出指令之前将其停顿。同样，我们可以确定在ID期间需要进行哪些转发，然后设置适当的控件。在流水线的早期检测联锁会降低硬件的复杂性，因为除非硬件停滞不前，否则硬件永远不必挂起更新处理器状态的指令。或者，我们可以在使用操作数（此流水线的EX和MEM）的时钟周期开始时检测到相关或转发。为了说明这两种方法的差异，我们将说明如何通过签入ID来实现写后读取（RAW）相关与来自加载指令的源的互锁（称为加载互锁），而实现可以在EX期间完成到ALU输入的转发路径的确定。图C.21列出了我们必须处理的各种情况。

从实现负载互锁开始。 如果源指令正在加载而存在RAW相关，则当需要加载数据的指令位于ID阶段时，加载指令将处于EX阶段。因此，我们可以用一张小表描述所有可能的相关情况，该表可以直接转换为实现。图C.22显示了一个表，该表在使用加载结果的指令位于ID阶段时检测所有加载互锁。

一旦发现相关，控制单元必须插入流水线堵转装置，并阻止IF和ID阶段中的指令前进。 如前所述，所有控制信息都在流水线寄存器中承载。（随便执行该指令就足够了，因为所有控制都是从该指令派生的。）因此，当我们检测到相关时，只需更改ID / EX流水线的控制部分

寄存器全为0，恰好是空操作（一条不执行任何操作的指令，例如加x0，x0，x0）。 另外，我们只是简单地重新循环IF / ID寄存器的内容来保存停顿的指令。在具有更复杂相关的流水线中，将应用相同的想法：我们可以通过比较一组流水线寄存器并移入无操作以防止错误执行来检测相关。



尽管还有更多情况需要考虑，但实现转发逻辑的方法类似。 实现转发逻辑所需的关键观察是流水线寄存器既包含要转发的数据，又包含源寄存器和目标寄存器字段。 从逻辑上讲，所有转发都从ALU或数据存储器输出到ALU输入，数据存储器输入或零检测单元。 因此，我们可以通过将EX / MEM和MEM / WB级中包含的IR的目标寄存器与ID / EX和EX / MEM寄存器中包含的IR的源寄存器进行比较来实现转发。 图C.23显示了比较和可能的转发操作，其中转发结果的目的地是EX中当前指令的ALU输入。

除了必须确定何时需要启用转发路径的比较器和组合逻辑之外，我们还必须扩大ALU输入处的多路复用器，并添加用于转发结果的流水线寄存器的连接。 图C.24显示了流水线数据路径的相关段，并带有附加的多路复用器和连接。

对于RISC V，相关检测和转发硬件相当简单。 我们将看到，当我们扩展此流水线以处理浮点时，事情变得有些复杂。 在此之前，我们需要处理分支。

* 1. Dealing With Branches in the Pipeline

在RISC V中，条件分支取决于比较两个寄存器值（我们假设它们在EX周期内发生），并将ALU用于此功能。 我们还需要计算分支目标地址。 因为测试分支条件并确定下一台PC将确定分支代价是多少，所以我们想计算两个可能的PC并在选择之前选择正确的PC

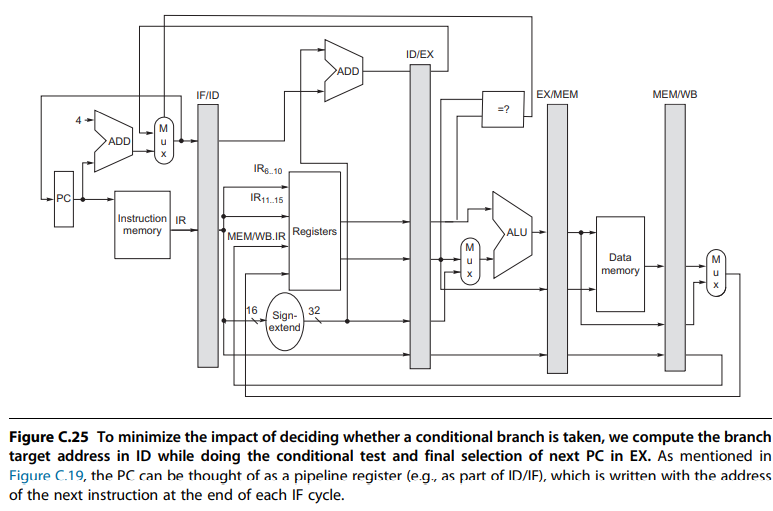
EX周期结束。 我们可以通过添加一个单独的加法器来实现此目的，该加法器在ID期间计算分支目标地址。 由于指令尚未解码，因此我们将像每个指令都是一个分支一样计算可能的目标。这可能比在EX中计算目标和评估条件要快，但确实消耗了更多能量。

图C.25显示了一条流水线数据路径，其中假定ID中的加法器和EX中的分支条件评估，这是流水线结构的微小变化。 该流水线将对分支机构造成两个周期的罚款。 在某些早期的RISC处理器（例如MIPS）中，分支条件测试被限制为允许在ID中进行测试，从而将分支延迟减少到一个周期。 当然，这意味着对寄存器的ALU操作后，基于该寄存器的条件分支会引起数据相关，如果在EX中评估分支条件，则不会发生数据相关。

随着流水线深度的增加，分支延迟增加，这使得动态分支预测成为必要。 例如，具有单独的解码和寄存器提取阶段的处理器可能会具有至少至少1个时钟周期的分支延迟。 除非处理延迟，否则分支延迟会变成分支惩罚。

许多实现更复杂指令集的较旧处理器的分支延迟为4个时钟周期或更多，而大型的深度流水线处理器通常具有6或7的分支惩罚。积极的高端超标量，例如所讨论的Intel i7。在第3章中，可能会有10–15个周期的分支惩罚。通常，流水线越深，时钟周期中的分支代价越差，并且准确预测分支就越关键。

1. What Makes Pipelining Hard to Implement?



* 1. Dealing with Exceptions

在流水线处理器中，异常情况更难处理，因为指令的重叠使得更难知道指令是否可以安全地更改处理器的状态。在流水线处理器中，一条指令是逐段执行的，并且在几个时钟周期内不会完成。但流水线中的其他指令会引发异常，这些异常可能会迫使处理器在流水线中的指令完成之前中止它们。在详细讨论这些问题及其解决方案之前，需要了解会出现什么类型的情况以及存在哪些支持它们的体系结构要求。

* + 1. Types of Exceptions and Requirements

用于描述例外情况的术语在处理器之间会改变正常的指令执行顺序。 尽管没有以一致的方式使用术语中断，故障和异常。 我们使用“例外”一词来涵盖所有这些机制，包括以下内容：

■ I/O device request

■ Invoking an operating system service from a user program

■ Tracing instruction execution

■ Breakpoint (programmer-requested interrupt)

■ Integer arithmetic overflow

■ FP arithmetic anomaly

■ Page fault (not in main memory)

■ Misaligned memory accesses (if alignment is required)

■ Memory protection violation

■ Using an undefined or unimplemented instruction

■ Hardware malfunctions

■ Power failure

当我们希望引用此类异常的某些特定类时，我们将使用更长的名称，例如I / O中断，浮点异常或页面错误。 尽管我们使用异常一词来涵盖所有这些事件，但是单个事件具有重要的特征，这些特征决定了硬件中需要采取什么措施。 可以在五个半独立轴上描述对异常的要求：

1. 同步与异步
2. 用户要求与强制
3. 用户可屏蔽与用户不可屏蔽
4. 指令内与指令间
5. 恢复与终止

图C.26根据这五个类别对前面的示例进行了分类。难点在于实现必须在恢复指令的指令内发生的中断。实现此类异常要求必须调用另一个程序以保存正在执行的程序的状态，更正异常的原因，然后恢复程序的状态，然后才能再次尝试导致异常的指令。该过程必须对执行程序实际上是不可见的。 如果流水线为处理器提供了处理异常，保存状态并重新启动而不影响程序执行的能力，则称流水线或处理器是可重新启动的。尽管早期的超级计算机和微处理器通常缺乏此属性，但是今天几乎所有的处理器都支持此属性，至少对于整数流水线支持，因为实现虚拟内存是必需的【阅第2章】。

* + 1. Stopping and Restarting Execution

发生异常时，流水线控制可以采取以下步骤安全地保存流水线状态：

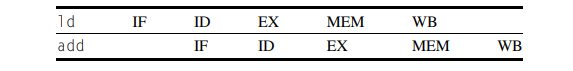
1.强制将陷阱指令插入下一个IF的流水线中。

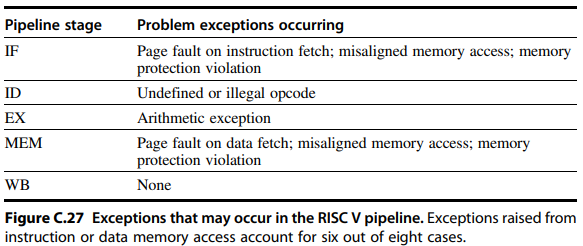
2.在捕获陷阱之前，请关闭故障指令和流水线中所有后续指令的所有写操作。 这可以通过将零放入流水线中所有指令的流水线锁存器中来完成，首先从生成异常的指令开始，而不是在该指令之前的那些例外。这样可以防止在处理异常之前无法完成的指令状态更改。

3.操作系统中的异常处理例程获得控制后，它将立即保存PC的故障指令。 此值将在以后用于从异常返回。

* + 1. Exceptions in RISC V

图C.27显示了RISC V流水线阶段以及在每个阶段中可能发生的问题异常。 使用流水线时，因为有多个指令正在执行，所以在同一时钟周期中可能会发生多个异常。 例如，考虑以下指令序列：





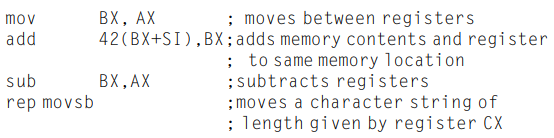
* 1. Instruction Set Complications

没有RISC V指令具有多个结果，并且RISC V流水线仅在指令执行结束时才写入该结果。当保证完成一条指令时，将其称为提交。在RISC V整数流水线中，所有指令到达MEM阶段的末尾（或WB的开始）时都会提交，并且在该阶段之前没有指令会更新状态。因此，精确的例外很简单。某些处理器的指令会在指令执行过程中更改状态，然后再保证指令及其前身完成操作。例如，IA-32体系结构中的自动递增寻址模式会导致在指令执行过程中更新寄存器。在这种情况下，如果指令由于异常而中止，它将使处理器状态改变。尽管知道是哪条指令导致了该异常，但是如果没有额外的硬件支持，则该异常将是不精确的，因为该指令将完成一半。在这种不精确的异常之后很难重新启动指令流。或者，可以避免在指令提交之前更新状态，但这可能很困难或成本很高，因为可能依赖于更新后的状态：考虑一个VAX指令，该指令可以多次自动递增相同的寄存器。因此，为了维护精确的异常模型，大多数具有此类指令的处理器都具有撤消在提交该指令之前所做的任何状态更改的能力。如果发生异常，处理器将使用此功能将处理器的状态重置为其值，然后再开始执行中断的指令。

一个相关的困难根源来自于在执行期间更新内存状态的指令，例如Intel体系结构或IBM 360上的字符串复制操作【参阅附录K】。为了能够中断和重新启动这些指令，这些指令被定义为使用通用寄存器作为工作寄存器。因此，部分完成的指令的状态始终在寄存器中，这些寄存器在发生异常时保存并在异常后恢复，从而允许指令继续执行。

状态的奇数位会带来不同的困难，这可能会造成额外的流水线危害或可能需要额外的硬件来保存和恢复。条件代码就是一个很好的例子。许多处理器隐式设置条件代码作为指令的一部分。这种方法具有优势，因为条件代码使条件的评估与实际分支脱钩。但是，隐式设置的条件代码会导致在设置条件代码和分支之间安排任何流水线延迟时遇到困难，因为大多数指令都设置了条件代码，并且无法在条件评估和分支之间的延迟时隙中使用。此外，在具有条件代码的处理器中，处理器必须决定何时确定分支条件。这涉及找出分支之前最后一次设置条件代码的时间。在大多数具有隐式设置条件代码的处理器中，这是通过延迟分支条件评估直到所有先前的指令都有机会设置条件代码来完成的。当然，具有显式设置条件代码的体系结构允许对条件测试和分支之间的延迟进行调度。但是，流水线控制仍必须跟踪设置条件代码的最后一条指令，以了解何时确定分支条件。实际上，必须将条件代码视为需要对带有分支的RAW危害进行危害检测的操作数，就像RISC V必须在寄存器中所做的一样。

以下是以流水线处理x86指令为例子来说明**流水线的多周期问题**：



尽管这些指令都不是特别长（x86指令最多可包含15个字节），但它们在所需的时钟周期数上确实有很大的不同，从低至一个时钟周期到数百个时钟周期。这些指令还需要不同数量的数据存储器访问，从零到可能的数百次。数据危害非常复杂，并且在指令之间和指令内部都发生（这不能阻止movsb的源和目标重叠！）。使所有指令执行相同数量的时钟周期的简单解决方案是不可接受的，因为它会引入大量的相关和旁路条件，并且流水线非常长。在指令级别上流水线x86很难，但是找到了一个聪明的解决方案，类似于用于VAX的解决方案。他们通过流水线传递微指令执行；微指令是按顺序使用以实现更复杂指令集的简单指令。因为微指令很简单（它们看起来很像RISC V），所以流水线控制要容易得多。自1995年以来，所有英特尔IA-32微处理器都采用了这种策略，将IA-32指令转换为微操作，然后对微操作进行流水线处理。实际上，这种方法甚至用于ARM体系结构中的一些更复杂的指令。

相比之下，负载存储处理器的简单操作具有相似的工作量和流水线。RISC V流水线如何处理长时间运行的指令，特别是浮点运算，是很重要的。多年来，人们认为指令集和实现之间的交互作用很小，并且实现问题并不是设计指令集的主要重点。在1980年代，很明显，流水线的难度和效率低下都可能因指令集的复杂性而增加。 在1990年代，所有公司都转向了更简单的指令集，目的是降低积极实施的复杂性。

1. Extending the RISC V Integer Pipeline to Handle Multicycle Operations