原书第十四章

第十六章

指令集并行性和超标量处理器

16.1概述

超标量与超流水线的对比

限制

16.2设计考虑

指令级并行性和机器并行性

指令发射策略

寄存器重命名

机器并行性分析

分支预测

超标量执行

超标量实现

16.3 Intel 酷睿微架构

前端

乱序执行逻辑

整数和浮点执行单元

16.4 ARMCORTEX-A8

指令提取单元

指令译码单元

整数执行单元

SIMD和浮点流水线

16.5 ARM CORTEX-M3

流水线结构

处理分支问题

16.6关键词、思考题和习题

学习目标

学习本章后，你应该能够：

阐述超标量和超流水线方法之间的区别。

定义指令级并行性。

讨论指令并行性的限制，即相关性和资源冲突。

概述指令并行性中涉及的设计问题。

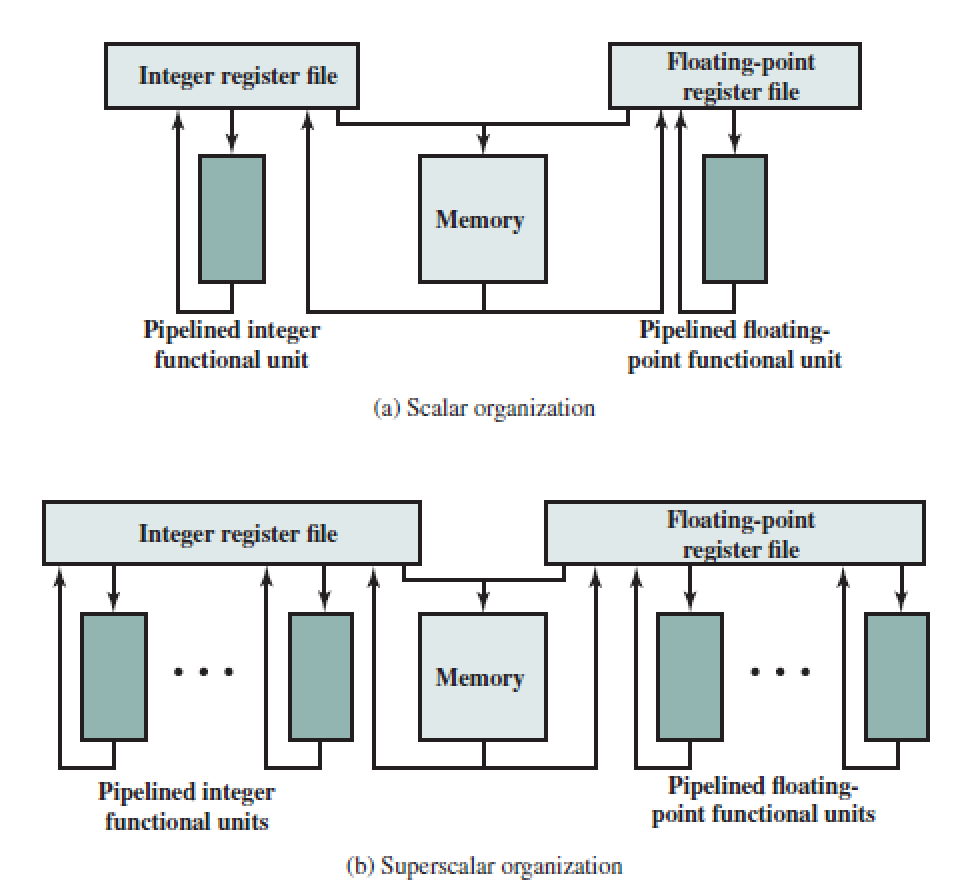
比较提高RISC机器和超标量机器中流水线性能的技术的不同。

超标量实现的处理器结构是指，在这样的结构中，整数和浮点运算、装载、保存以及条件分支之类的普通指令，能同时启动并独立执行。这种实现引出了几个复杂的涉及指令流水线的设计问题。

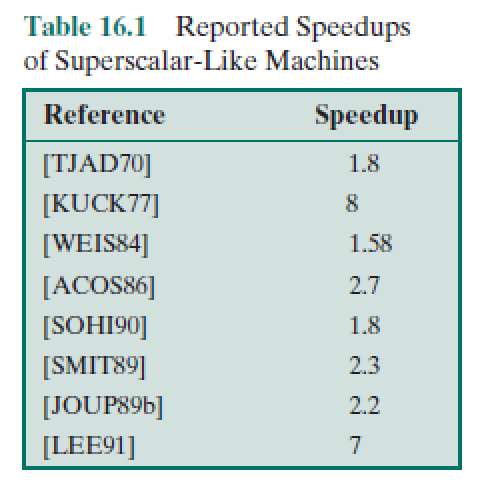
超标量设计紧跟RISC体系结构的脚步，虽然RISC机器精简指令集体系结构自身已倾向于应用超标量技术，但超标量方法既能用于RISC也能用于CISC体系结构。

其实，以IBM 801和Berkeley RISC I开始的RISC研究到RISC商品机的推出，其孕育期长达7-8年，而最初成为商业可用的超标量机器只是超标量这个概念提出后一两年的事。超标量方法现在已成为实现高性能微处理器的标准方法  
 本章先是概述超标量方法，将它与超级流水线对照。接着提出与超标量实现相关的主要设计考虑，然后考察几个最具代表性的超标量处理器实例。  
  
**16.1概述**

超标量（superscalar）这一术语最早是在1987年提出的[AGER87]，它指的是为改善标量指令执行性能而设计的机器。在大多数应用中，大量操作都是对标量进行的。因而超标量方法代表了高性能通用处理器的进一步发展。  
 超标量方法的本质是一种在不同流水线中可独立执行指令的能力。此概念可进一步发展为允许指令以不同于原程序顺序的次序来执行。图16-1比较了标量和超标量方法的通用模式。在传统的标量构架中，有一个用于整数运算的流水线功能单元和一个用于浮点运算的功能单元。并通过使多个指令一次处于流水线的不同阶段来实现并行性。在超标量构架中，有多个功能单元，每个功能单元都以流水线的形式被实现。每个单独的功能单元凭借其流水线结构提供一定程度的并行性。 处理器使用多个功能单元便能够并行执行指令流，每个流水线一个流。硬件与编译器一起负责确保并行执行的操作不违反程序的意图。



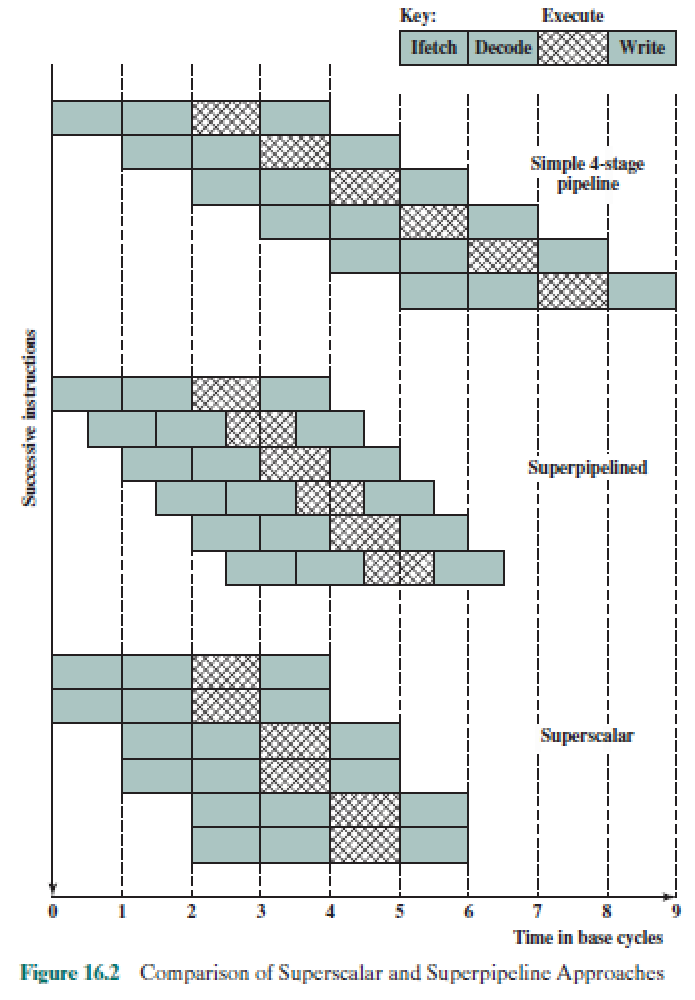
不少研究人员考察了类超标量的处理器，他们的研究指出这些处理器可以有某种程度的性能改善。表16-1列出了所报告的性能改进，结果的不同起因于被模拟机器硬件设备模拟应用方面的不同。



**16.1.1超标量与超级流水线的对比**

实现更高性能的另一种方法是超级流水线（super pipelining），这一术语最早提出是在1988年[JOUP88]。超级流水线是利用了如下事实：多数流水阶段所完成的任务只需要比小于时钟周期一半的时间。于是，双倍的内部时钟速率允许在一个外部时钟周期内完成两个任务。我们已看过这个方法的一个例子，即MIPS R4000。

图16-2比较了这两种方法。图的上部显示了一个普通的流水线，用作比较的基础。它是每个时钟周期发出一条指令，并能每时钟周期完成一个流水段。此流水段有4段：取指令、操作译码、操作执行和结果写回。为清楚起见，执行段以阴影表示，注意虽有几条指令并执行，但任何时刻只有一条指令处于执行段。



图的中间部分表示一种超级流水线实现，该实现能每个时钟周期完成两个流水阶段。观察它的另一种方式是将每个流水阶段所完成的任务分成两个不重叠的部分，并且每个部分能在半个时钟周期内执行完。这种样式的超级流水线被称作级别为2。图的最下部表示的是一种超标量实现，它能并行执行每阶段的两个实例。自然，更高程度的超级流水线和超标量实现也完全是可能的。

图16-2所描述的超级流水线和超标量实现，二者在稳定状态下，具有相同的指令数同时在执行。在程序开始和每次转移到目标时超级流水线处理器落后于超标量处理器  
  
**16.1.2限制**

超标量方法依赖于并行执行多条指令的能力。**指令并行性**（instruction-level parallelism）指的是程序指令能并行执行的程度。硬件技术与编译器优化技术的结合能够达到最大程度的指令并行性。在考察超标量机器用于提高指令级并行性所采用的设计技术之前，我们需要查看并行性的基本限制，这些限制是系统必须认真对待的。参考文献[JOHN91]中列出五种限制：

·真实数据相关性（true data dependency）

·过程相关性（procedure dependency）

·资源冲突（resource conflict）

·输出相关性（output dependency）

·反相关性（antidependency）

本节先考察前三个限制，后两个限制留在下一节讨论。

**1. 真实数据相关性**

考虑如下指令序列[[1]](#footnote-2)：

ADD EAX, ECX ; 将寄存器EAX的内容和ECX的内容相加，结果保存到EAX寄存器中。

MOV EBX, EAX ; 将寄存器EAX的内容保存到EBX寄存器中

第二条指令能取址并译码，但直到第一条指令执行完成之前不能被执行。原因在于第二条指令需要第一条指令产生的数据，这种情况称为真实数据相关性，也称为流相关性（flow dependency）或写后读相关性（read after write dependency，RAW）。

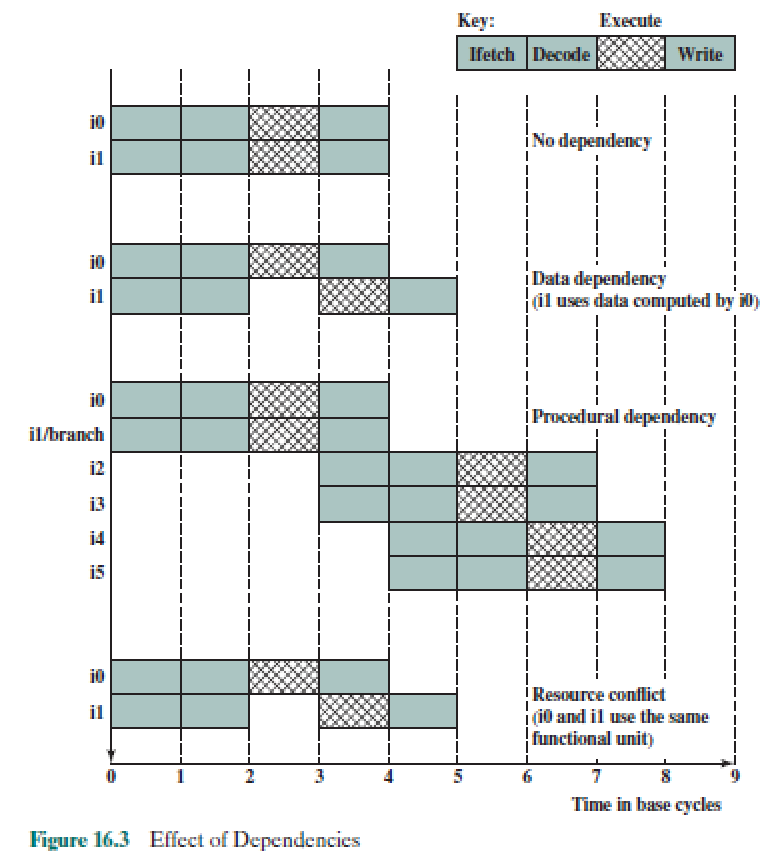


图16-3说明了级别为2的超标量机器中的这种相关性。若没有相关性，两条指令能并行的取指和执行。若第一、第二条指令有数据相关性存在，则第二条指令要延迟一定时钟周期以待相关性消除。通常，任何指令在直到它的所有输入值都已产生之前必须被延迟。

一个简单的流水线中，例如图16-2上部所示的流水线，上述指令序列可能不会引起延迟，然而考虑如下指令序列，其中一条指令从内存装载一个数，而不是从寄存器读取一个数：

MOV EAX, eff ; 将有效存储器地址eff处的内容装到寄存器EAX

MOV EBX, EAX ; 将EAX的内容传送到寄存器EBX

一个典型的RISC处理器要用2个或更多时钟周期来完成由存储器取出的操作，这是在高速缓存（cache）命中的情况下。由于片外存储器访问的延迟，所有高速缓存级别的高速缓存丢失可能需要几十甚至数百个周期才能完成操作。补偿这种延迟的一种方法是编译器重排指令顺序，让不必等待存储器装载数据的一条或多条后续指令开始进入流水线。这种策略在超标量流水线情况下不太有效。在装载期间执行的相关指令很可能在装载的第一个周期执行完毕，留下处理器无事可做直到装载完成。

**2. 过程相关性**

正如第14章所讨论的，指令序列中出现分支指令把流水操作弄复杂了。分支（发生或不发生转移）之后的指令有对分支指令的过程相关性。而且直到分支指令被执行之前它们都不能去执行。图16-3说明了分支对级别为2的超标流水线的影响。

我们已经见识过过，这类相关过程的相关性也会影响简单的流水线。但结果对于超标量流水来说要更严重，因为每个延迟会丢失更多的机会。

若使用变长指令，会出现另一类过程相关性。因为任何指令的具体长度不是事先已知的，在取后续指令之前，它必须至少部分地被译码。这就妨碍了超标量流水所要求的指令同时取指。这也是超标量技术更适合用于RISC和类RISC结构的理由之一，因为它们的指令长度固定。

**3. 资源冲突**

资源冲突是指两个或多个指令同时竞争同一资源。资源的例子包括存储器、缓存、总线、寄存器组端口和功能单元（如ALU加法器）。

对于流水线而言，资源冲突是类似数据相关性的行为（参见图16-3）。然而也有些不同，资源冲突可以通过复制资源来克服，而真实数据相关性是不能被消除的。还有，当操作需要较长时间来完成时，通过将相应的功能单元流水化可减轻资源冲突。

**16.2 设计考虑**

**16.2.1指令级并行性和机器并行性**

文献[JOUP89a]指出了指令级并行性和机器并行性这两个相关概念的一个重要区别。当指令序列中的指令是独立的，并在此基础上能通过重叠来并行执行时，则存在**指令级并行性**（instruction-level parallelism）。

作为说明指令级并行性概念的一个例子，考虑如下两个代码片段[JOUP89b]：

Load R1 d R2 Add R3 d R3, “1”

Add R3 d R3, “1” Add R4 d R3, R2

Add R4 d R4, R2 Store [R4] d R0

左边的三条指令是独立的，并且从理论上讲这三条是可以并行执行的。相对照，右边的三条指令则不能并行执行，因为第二条指令使用了第一条的结果，第三条指令又使用了第二条的结果。

代码中的真实数据相关性和过程相关性出现的频繁程度决定了指令级的并行性。这些因素本身又取决于指令级体系结构和应用程序。指令级并行性也可由操作延迟时间（operation latency）所确定[JOUP89a]。操作延迟时间是指等到一条指令的结果可作为后续指令的操作数使用时，所需的等待时间。它确定了一个数据或过程的相关性将引起多长的延迟。

机器并行性（machine parallelism）是指处理器获取指令级并行性好处的能力指标。机器并行性由下面这些因素决定：它能同时取指和执行的指令数（并行流水线数），以及处理器用于找出独立指令所使用结构的速度及精巧程度。

指令级并行性和机器并行性都是提高性能的重要因素。一个不具有充分指令级并行性的程序也能取得机器并行性的全部好处。像RISC那样使用固定长度的指令集结构，增强了指令集并行性。从另一方面讲，有限的机器并行性又将限制程序的性能，无论其什么性质。

**16.2.2.指令发射策略**

正如所提到过的，机器并行性并不只是使每个流水段能容纳多条指令这样简单的事情。处理器必须能识别出指令级并行性，并指挥流水线去取指、译码和执行。文献[JOHN91]使用了术语**指令发射**（instruction issue），它是指启动指令去处理器功能单元执行的过程，并用指令发射策略（instruction-issue policy）这个术语来表示启动指令执行时所采用的协议。通常，我们说指令发射是在指令从流水线的译码阶段向流水线的执行阶段前进时发生的。

实际上，指令发射就是处理器试图在当前执行点之前查找能进入流水线并执行的指令。因此，三种类型的排序是重要的：

·取指令的顺序。

·指令执行的顺序。

·指令改变寄存器和存储器位置内容的秩序。

处理器越精巧，对这些顺序间严格关系的限制就越少。对那些在严格顺序执行中所见到的次序，处理器可能需要更改一个或多个指令的次序以要求做到各个流水线部件的最大化利用。对此的唯一限制是，处理器必须保证结果是正确的。于是处理器必须协调先前所讨论过的各种相关性和冲突。

通常，我们能把超标量指令发射策略分为下面这几种：

·按序发射按序完成（in-order issue with in-order completion）

·按序发射乱序完成（in-order issue with our-of-order completion）

·乱序发射乱序完成（out-of-order issue with our-of-order completion）

1. **按序发射按序完成**

最简单的发射策略是，严格地按照顺序执行的那个顺序来发射指令（按序发射）。并以同样的顺序写入结果（按序完成）。即使标量流水线也不遵循这种简单方式的策略。然而将它作为更复杂指令发射方法的一个比较基准还是有用的。

图16-6a给出了这种策略的一个例子。假定超标量流水线一次能取并译码两条指令，有三个分立的功能单元（如整数算术、浮点算术等），有两个流水写回段的部件。该例展示了一个6条指令的代码片段，并假定有如下限制：

·I1执行要求两个执行周期。

·I3和I4为使用同一功能单元而发生冲突。

·I5依赖于I4产生的值。

·I5和I6为使用同一功能单元而发生冲突。

以上是一次取两条指令并传送到译码单元。因为指令是成对取的，所以后两条指令必须等待，直到译码流水段已完成上次所取指令的译码。为保证按序完成，当有功能单元冲突或功能单元产生结果需要不止一个周期时，指令发射必须停止。

在这个例子中，由译码第一条指令写回最后结果总共花费的时间是8个时钟周期。

1. **按序发射乱序完成**

  乱序完成在标量RISC机器中用来改善需要执行多个时钟周期指令的性能。图16-4b说明了它在超标量处理器上的使用。指令I2被允许先于I1完成。这就允许I3也能更早完成，从而节省了一个时钟周期。

采用乱序完成的话，在任何时候都可能有多条指令在流水线执行阶段运行，最大数目取决于各个功能单元之间的最大机器并行度。如果发生资源冲突、出现数据相关性或过程相关性，指令发射将被迫停顿。

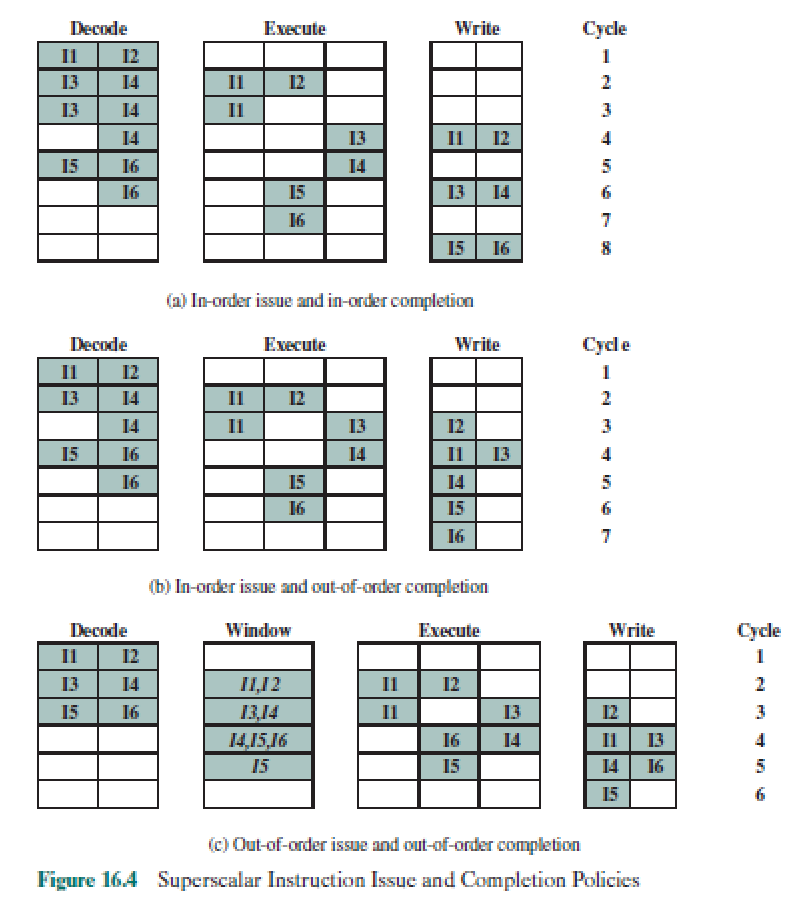
除开上面的限制，一种新的相关性。前面曾称为输出相关性，也称为“写后写”相关性（write after write dependency，WAW）出现了。以下代码片段说明了这种相关性（其中op表示任何 一种操作）。

I1: R3 ← R3 op R5

I2: R4 ← R3 + 1

I3: R3 ← R5 + 1

I4: R7 ← R3 op R4



指令I2不能先于I1执行，因为I2需要I1在R3后产生的结果。这是16-1节所描述过的真实数据相关性的例子。类似地，I4必须等待I3，因为它使用I3产生的结果。那么I1和I3之间有什么关系呢？这里没有我们已定义的那种数据相关性。然而若I3的执行先于I1完成，则R3内容的错误值将被I4的执行所取用。于是I3必须在I1之后完成，以产生正确的结果。如果它的结果可能会被一条需要较长时间完成的较早指令改写的话，为保证结果正确，第3条指令的发射必须停止。

乱序完成比按序完成要求更复杂的指令发射逻辑。另外，其在处理中断和异常时也更困难。当一个中断出现时，当前点的指令执行被挂起，中断处理后再恢复。处理器必须保证这个恢复操作已考虑到下述情况：中断发生时，那些位于引起此中断的指令之后的指令可能已经先行完成。

1. **乱序发射乱序完成**

如果按序发射，那么处理器对指令进行译码时，遇到相关点或冲突点即停顿，这期间没有另外的指令被译码，直到冲突解决。于是，处理器不能向前查看冲突点的后续指令，而这些后续指令可能独立于已在流水线中的指令，因而这点可被加以利用，引入流水线中。

为允许乱序发射，有必要解耦流水线的译码段和执行段。这是通过使用一个称为**指令窗口**（instruction window）的缓冲器来完成的。在这种组织方式下，处理器译完一条指令就把它放入指令窗口，只要缓冲器未满，处理器就继续取指和译码新指令。当执行阶段中的功能单元变成可用时，需要此功能单元的指令就会由指令窗口发射到执行段。只要（1）指令所需的具体功能单元是可用的，以及（2）没有冲突或相关性阻塞这条指令，那任何指令都可以被发射。

这种组织方式的结果是处理器要有先行查找的能力，这样允许它识别那些能放入执行段的独立指令。指令由指令窗口发射出去的次序则很少遵循它们原来的程序顺序。同前面一样，唯一的限制是程序执行结果得是正确的。

图16-4c说明了这种策略。每周期里，两条指令被取入译码段。由于缓冲器大小的限制，每周期两条指令由译码段进入窗口。在这个例子中，指令I6先于I5被发射时可能的（回想一下，I5依赖于I4，但I6则并非如此。）于是，执行和写回两段都节省了一个周期，与图16-4b相比，端到端是节省了一个周期的。

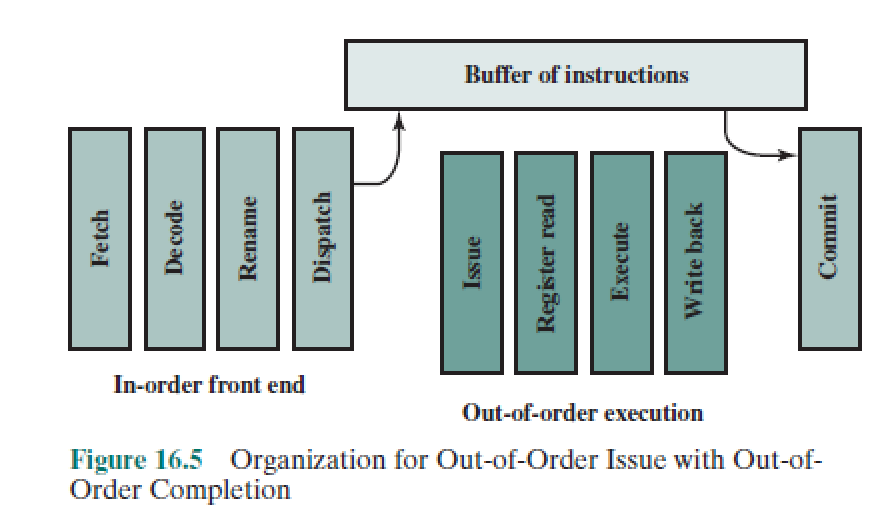


图16-4c中所表示的指令窗口仅在于说明它的作用。注意，它并不是一个附加的流水线阶段。一条指令位于窗口中简单地意味着，处理器具有关于那条指令应何时发射的充足信息。

乱序发射乱序完成策略也要服从前面所描述过的限制。若一条指令违背相关性或冲突，那它不能进行发射。不同之处在于有更多的指令可被用来发射，降低了流水线阶段不得不停顿的发生概率。另外，一种前面称为反相关性（也称为“读后写”相关性，WAR）的问题出现了。前面考察过的代码段可说明这种相关性。

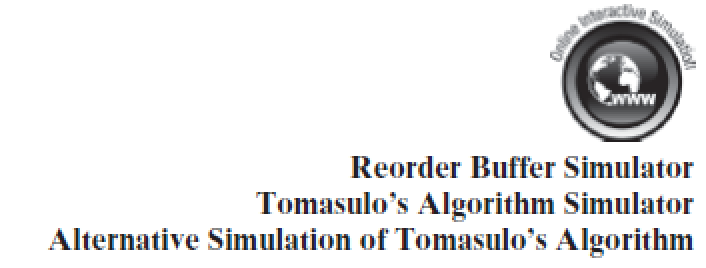
I1: R3 ← R3 op R5

I2: R4 ← R3 + 1

I3: R3 ← R5 + 1

I4: R7 ← R3 op R4

在指令I2开始执行并已取得它的操作数之前，指令I3不能完成执行。这里是因为I3修改寄存器R3，而R3是I2的源操作数。这里使用术语反相关性（antidependency）是因为这一限制与真实数据相关性类似，但正好相反。真实数据相关性是前一条指令产生的值会被后一条指令使用，而反相关性是后一条指令破坏前一条指令所使用的数据值。



**16.2.3寄存器重命名**

当允许乱序指令发射和/或乱序指令完成时，我们发现会有产生输出相关性（写后写相关性，WAW）和反相关性（读后写相关性，WAR）的可能。这些相关性不同于真实数据相关性和资源冲突，后者反映了通过程序的数据流和执行的顺序，而输出相关性和反相关性的出现，从另一方面看，是因为寄存器的值可能不再反映被程序流指定的值顺序。

当指令顺序发射顺序完成时，在程序的每个执行点上是可以确定每个寄存器的内容的。当采用乱序技术时，仅考虑程序指定的指令顺序，则每点上的寄存器值不能完全已知。实际上，值对于寄存器的使用是存在冲突的，处理器必须偶尔停顿一个流水线阶段来解决这些冲突。

反相关性和输出相关性都是寄存器存储冲突的例子。多个指令为使用同一寄存器位置而竞争，产生了妨碍性能的流水限制。当采用寄存器优化技术被时问题会恶化，因为这些编译器技术力图最大限度地使用寄存器，相应也使得寄存器存储冲突最大化。

对付这种类型的存储的一种方法是基于传统的资源冲突解决方法：资源复制。在现在的语境中，此技术称为**寄存器重命名**（register renaming）。本质上，寄存器由处理器硬件动态分配，并且它们与各时间点指令所需值相关。当一个新寄存器产生时（即当一条以寄存器为目标操作数的指令执行时），一个新寄存器被分配给那个值。作为源操作数访问那个寄存器值的后续指令则必须历经一个重命名过程：这些指令中的寄存器引用部分必须修改成对含有所需值寄存器的引用。故若准备使用不同值，不同指令中的同样原始寄存器引用可能引用到不同的实际寄存器。

让我们考虑寄存器重命名如何用于已考察过的那个代码段上。

I1: R3b ← R3a op R5a

I2: R4b ← R3b + 1

I3: R3c ← R5a + 1

I4: R7b ← R3c op R4b

不带下标的寄存器引用指的是指令中找到的逻辑寄存器。带下标的寄存器引用指向被分配用来保存新值的硬件寄存器。当一对具体逻辑寄存器进行新的分配后，作为源操作数访问那个逻辑寄存器的后续指令要修改成对最近被分配的硬件寄存器的引用（最近是依据程序的指令顺序而定）。

在这个例子中，指令I3中的寄存器R3c的生成则避免了对第二条指令的反相关性和对第一条指令的输出相关性，而且它不影响正被I4访问的正确值。结果是I3能立即被发射；没有重命名，直到第一条指令完成和第二条指令发射完毕之前，I3不能被发射。

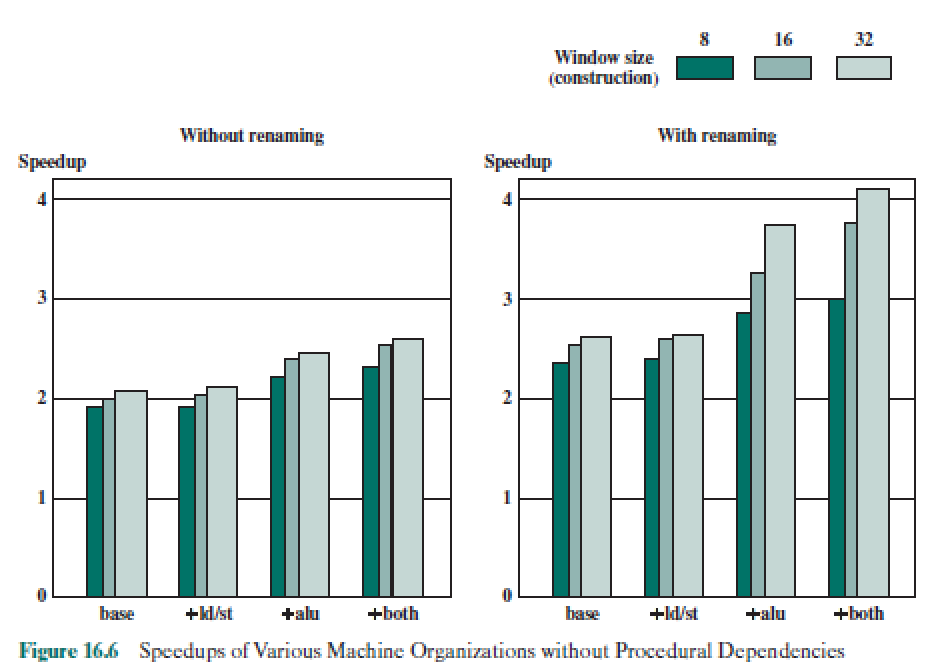


不同于寄存器从重命名的另一种允许指令乱序发射的技术是记分牌（scoreboarding）技术。本质上讲，记分牌是一种寄存器使用登记技术，该技术允许指令乱序执行，只要指令不依赖于前面的指令，且不存在结构冒险时该指令就可以被发射。请参考附录N对记分牌技术的介绍。

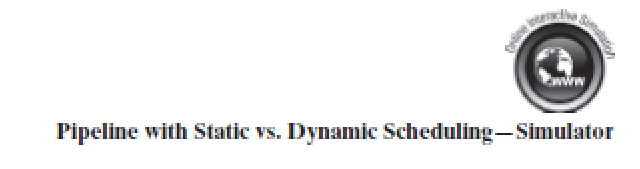
**16.2.4机器并行性**

前面已查看了能用在超标量处理器中提高性能的三种硬件技术：资源复制、乱序发射和重命名。[SMIT89]中的研究报告说明了这些技术之间的相互关系。这个研究是在有MIPS R2000特征的模拟器上进行的，并带有各种超标量特性的增强。研究者对几种不同的程序指令序列进行了模拟。

图16-6表示其结果。在每个图中，纵轴表示超标量机器与标量机器相对比的加速程度。横轴对应着4种不同的处理器组织。基本（base）机器不复制任何功能单元，但它可以乱序地发射指令。第二种配置（+装入/存储）是复制了访问数据高速缓存的装载/保存功能单元。第三种配置（+ALU）复制了ALU。第四种配置（+两者）是装载/保存和ALU都复制。每个图中的三个图柱分别对应的是8、16和32条指令的指令窗口，它们指出处理器能先行查找的指令总量。左右两图不同之处在于，右图允许寄存器重命名。这等于说左图反映的机器受限于所有相关性，而右图所对应的机器只受限于真实相关性。



结合两图能得出些重要的结论。首先，没有寄存器重命名而添加功能单元可能不会有太大意义。此时会有些许的性能改善，但要付出增加硬件复杂性的代价。使用寄存器重命名，消除了反相关性和输出相关性，通过添加更多的功能单元，能实现显著的加速。还要注意，在实现加速的总量方面，使用8指令的指令窗口与更大指令窗口之间也有明显不同，这意味着若指令窗口太小，数据相关性将妨碍额外功能单元的有效利用；处理器必须有能力更快更超前地找出独立的指令，才能更全面地利用硬件。



**16.2.5分支预测**

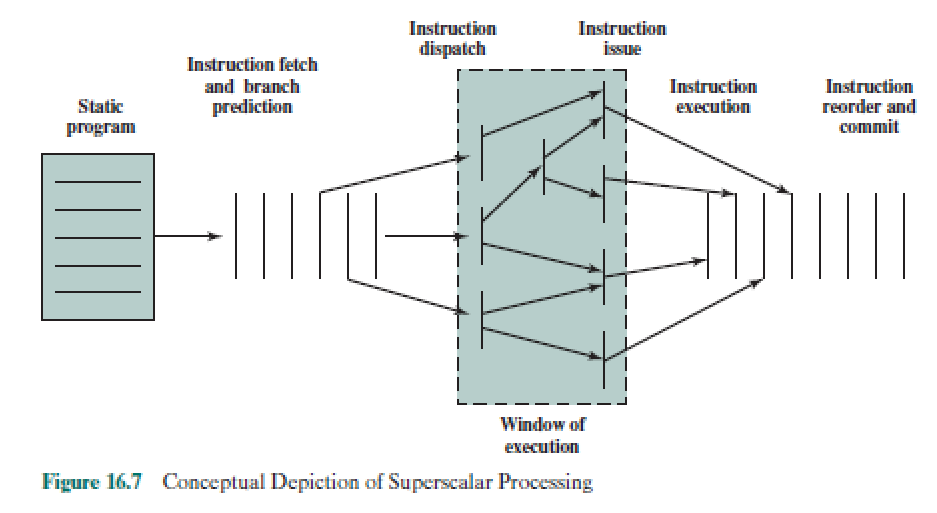
任何高性能的流水式机器必须解决分支处理问题。例如，Intel 80486解决这个问题的方法是，既读取位于分支指令之后的下一顺序指令，又推测地读取转移目标处的指令。然而由于取指和执行之间有两个流水段，当转移发生时这种策略要导致两个周期的延迟。

基于RISC机器的先进性，可采用延迟分支策略。这允许处理器在预取一些无用指令之前，先计算条件分支指令的结果。通过这种方法，处理器总是执行紧跟在分支指令之后的那条指令。这样，在处理器读取新的指令流的同时可保持流水线满载。

随着超标量机器的发展，延迟分支策略反而被较少采用了。原因在于多条指令需要在延迟槽中进行，这会引起一些指令相关性问题。于是超标量机器又转回RISC出现以前使用的分支预测技术。某些机器，像PowerPC 601，采用简单的静态分支预测技术。更为复杂的机器，像PowerPC 620和Pentium4，采用基于转移历史分析的动态分支预测技术。

**16.2.6超标量执行**

现在，我们可对超标量的执行提供一个概述，如图16-7所示。将被执行的程序由一个线性指令序列组成，这是程序员编写的或编译器生成的静态程序。包括分支预测在内的取指令过程用来形成一个动态的指令流。对此指令进行相关性检查后，处理器会解除某些人为的相关性。然后处理器派发指令进入执行窗口，在此窗口中指令不再是顺序流，而是根据它们的真实数据相关性来排序。处理器以真数据相关性和资源可用性所确定的顺序来完成每条指令的执行阶段，最后，指令的结果被登记，从概念上讲它们是被放回到原顺序序列。



上面所提到的最后一步称为**提交**（commiting）或回收（retiring）指令，需要此步有如下理由：首先，由于使用并行的多条流水线，指令会以不同于静态程序的顺序来完成；其次，预测分支和推测执行的使用，意味着某些指令已完成执行但其结果需要放弃，因为它们所在的分支没有真正发生。于是当一条指令执行完，不能立即修改固有存储位置和程序可见的寄存器，而应将结果暂存到一个相关指令使用的临时存储位置中，当确认顺序模型执行指令时，再使其结果永久化。

**16.2.7超标量实现**

依上述的全面讨论，我们能对超标量方式所需要的处理器硬件予以某些一般性评论。在[SMIT95]中列出了如下关键部件：

·同时取多条指令的取指策略，经常要有预测条件分支指令结果和超前取指的功能，这要求使用多个取指和译码流水线段，以及分支预测逻辑。

·确定有关寄存器值真相关性的逻辑，以及执行期间把这些值与需要他们的位置之间相互联系起来的机制。

·并行启动或发射多条指令的机制。

·多条指令并行执行所需的资源，包括多个流水的功能单元，以及为多个存储器访问同时提供服务的存储器层次结构。

·以正确顺序提交处理器状态的机制

**16.3 Intel酷睿微架构**

虽然超标令设计这一概念通常是与RISC体系结构联系在一起的，但是同样的超标量原则也能应用到CISC机器上。或许这方面的最著名的例子要属Intel x86架构了。考察超标量概念产品在Intel产品系列中发展情况是有益的。386是一个传统的非流水CISC机器。486是x86系列处理器中第一个引入流水线的，从而使得整数操作的平均延迟从2~4个时钟周期减少到了1个时钟周期。不过486仍然被限制为一个时钟周期只能执行一条指令，没有超标量的部件。最初的Pentium有了一定的超标量能力，它使用了两个分立的整数执行单元。Pentium Pro引入了全面的超标量设计理念。后续的x86模型的处理器具有更精进、功能更强大的超标量设计。

图16-8给出x86流水线的普通框图。Intel将这样的流水线架构称为微架构（microarchitecture）。微体系结构是实现机器指令集体系结构的基础，被称为Intel酷睿微体系结构。它是在Intel 酷睿2和英特尔Xeon处理器家族中的每个处理器核心上实现的。也还有一个增强的Intel酷睿微架构。两个微架构之间的一个关键区别在于增强型Intel酷睿微架构提供了第三级缓存。

表16-2显示了缓存体系结构的一些参数和性能特征。所有缓存使用写回更新策略。当指令从存储器位置读取数据时，处理器按照以下顺序在缓存和主存储器中查找包含该数据的缓存行：

（1）启动核心的L1数据缓存

（2）其他内核和L2缓存的L1数据缓存

（3）系统内存

只有在修改了另一核的L1数据缓存时，才从其中获取缓存行，从而忽略L2缓存中的缓存行可用性或状态。表16-2b显示从内存集群中获取不同位置的前四个字节的特性。延迟列提供访问延迟的估计。然而实际的延迟可能会根据缓存、内存组件及其参数的负载而变化。

Intel酷睿微架构的流水线包含：

1. 从存储器中提取指令流的有序发布前端，其具有四个指令译码器，用于将译码后的指令提供给无序执行核心。每条指令被译成一个或多个定长的RISC指令，称为**微操作**（micro-operation，micro-op）。
2. 乱序超标量执行核心，每周期最多可发射6个微操作并重新排序之，以便在源准备好并且执行资源可用时立即执行。
3. 一种顺序退休单元，用于确保微操作的执行结果得到处理，并且根据原始程序顺序更新体系结构状态和处理器的寄存器集。

从效果上看，Intel酷睿微架构由外层的CISC壳和内部的RISC核所组成。内部的RISC微操作通过至少有14段（stage）的流水线。在某些情况下，微操作要求多个执行段，导致流水线更长。这可与早期Intel x86处理器和Pentium上使用的5段流水线（见图14-21）做一对照。

**16.3.1前端**

前端需要提供译码指令（micro-ops），并将流维持到6个问题宽度的乱序引擎。它主要由三个部分组成：分支预测单元（branch prediction unit，BPU）、指令提取与预编码单元、指令队列与译码单元。

**1. 分支预测单元**

这个单元通过预测各种分支类型：条件、间接、直接、调用和返回，帮助指令获取单元获取最有可能被执行的指令。分支预测单元使用每个分支类型的专用硬件。分支预测使处理器能够在决定分支结果之前很久就开始执行指令。

微架构的使用基于分支指令最近执行的历史的动态分支预测策略。一个分支目标缓存区（branch target buffer，BTB）会被维护，它会缓存最近遇到的分支指令的信息。只要在指令流中遇到分支指令，就检查BTB。如果BTB中已经存在条目，那么在确定是否采用分支时，指令单元由该条目的历史信息引导。如果预测到分支，则与此条目关联的分支目的地址用于预取分支目标指令。

一旦指令被执行，正确的条目的历史部分被更新以反映分支指令的结果。如果在BTB中没有表示该指令，则将该指令的地址加载到BTB中的条目中；如有必要，则删除较旧的条目。

前面两段的描述大体上符合原始Pentium模型以及后续使用的分支预测策略Pentium模型，包括目前的Intel模型。然而，在Pentium系列中则采用相对简单的2位历史方案。后续模型具有更长的流水线（Intel酷睿微架构包含14个阶段，而Pentium则包含5个阶段），因此错误预测带来的惩罚更大。故后面的模型使用具有更多历史位的更精细的分支预测方案来降低误预测率。

在BTB中无历史记录的条件分支指令，采用静态预测算法。转移与否根据如下规则来预测：

·对于转移地址不是IP相对寻址的条件分支指令，如果该分支指令是一个返回，则也测发生，否则预测不发生。

·对于IP相对寻址的后向条件分支指令，预测转移发生。这个负责反映了典型的循环行为。

·对于IP相对寻址的前向条件分支指令，预测转移不发生。

**2. 指令提取和预编码单元**

指令提取单元包括指令翻译前瞻缓冲器（instruction translation lookaside buffer，ITLB）、指令预取器、指令高速缓存和预编码逻辑。

指令提取是从L1指令缓存执行的。当发生L1缓存丢失时，按序前端一次从L2缓存64字节向L1缓存提供新指令。默认情况下，指令是顺序获取的，因此每个L2缓存行获取内容都包括下一个要获取的指令。通过分支预测单元的分支预测可以改变该顺序提取操作。ITLB将给定的线性IP地址转换为访问L2缓存所需的物理地址。前端的静态分支预测用于确定接下来要获取哪些指令。

预编码单元接受来自指令缓存或预取缓冲器的16个字节，并执行以下任务：

·确定指令的长度。

·译码所有与指令相关的前缀。

·为译码器标记指令的各种属性（例如，“is分支指令”）。

预编码单元每个周期最多可以写入6个指令到指令队列中。如果一个提取含超过六个指令，那么预编译器每周期继续译码多达六个指令，直到提取操作中的所有指令都被写入指令队列。后续的获取只能在当前获取完成之后再进入预编码。

**3. 指令队列和译码单元**

获取的指令被放置于指令队列中。译码单元从那里扫描字节以确定指令边界；这是必要的操作，因为x86指令长度可变。译码器将每条机器指令转换为1至4个微操作，每个微操作都是一条118位的RISC指令。注意，为了进行比较，大多数纯RISC机器的指令长度只有32位。为了适应更复杂的x86指令则需要更长的微操作长度。尽管如此，微操作比派生它们的原始指令更容易管理。

一些指令需要超过4个微操作。这些指令被传送到微码ROM，其中包含与复杂机器指令相关联的一系列微操作（5个及以上）。例如，字符串指令可以译成一个非常大的（甚至数百个）、重复的微操作序列。因此，微码ROM在第六部分讨论过的意义基础上是一个微程序控制单元。

得到的微操作序列被传递到重命名/分配器模块。

**16.3.2乱序执行逻辑**

处理器的这部分将重排序微操作，以允许它们一旦输入操作数就绪即可快速被执行。

**1. 分配**

流水线的分配（allocate）段为微操作的执行分配资源。它完成如下功能：

·每时钟周期有三个微操作到达分配器。如果其中某个微操作所需的寄存器这类资源不可用，则分配器停顿流水线，直到三者所需资源都可用。

·分配器要为微操作在重排序缓冲器（reorder buffer，ROB）中分配一项。此ROB共有126项，每项跟踪一个操作执行过程中的完成状况。[[2]](#footnote-3)

·分配器要为微操作的结果数据在128个整数或浮点寄存器组中分配一项，以及可能为流水线中的装载（可多达48个）和保存（可多达24个）微操作分配一个装载或保存缓冲器。

·分配器在指令调度器前面的两个微操作队列中的一个中分配条目。

ROB是一个环形缓冲器，能保持多达126个的微操作，并含有128个硬件寄存器。每个缓冲器由下列字段组成。

·**状态**（State）：指示此微操作是否已被调度、派发、完成执行、回收（retirement）就绪等。

·**存储器地址**（Memory Address）：产生此微操作的Pentium 指令地址。

·**微操作**（Micro-op）：实际的操作。

·**别名寄存器**（Alias Register）：若微操作引用了机器体系结构16个寄存器的某一个，则此字段将该引用重定向到128硬件寄存器的某一个。

微操作按序进入ROB，然后微操作从ROB被无序地分发到调度/执行单元。调度的标准是合适的执行单元和这个微操作所需的所有必要数据项都可用。最后，微操作从ROB中按序回收。为了实现按序回收，已完成的微操作项打上回收就绪标志，并由时间最早的微操作开始，顺序回收这些已标记的微操作。

**2. 寄存器重命名**

在寄存器重命名（register renaming）段将对16个体系结构寄存器（8个浮点寄存器加上EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP）的引用重新映射到128个物理寄存器。这样，就解除了由体系结构寄存器数量有限引起的虚假数据相关性，与此同时仍保留了真实数据相关性（写后读，RAW）。

**3. 微操作排队**

在资源分配和寄存器重命名之后，微操作被放置在两个微操作队列之一中，在那里一直保持存放，直到调度器中有空间。两个队列一个用于存储器操作（装载和保存），一个用于不涉及存储器访问的其他微操作。每个队列均遵循先进先出（FIFO）规则，但队列间不维护一个次序。也就是说，一个微操作是否出队与另一队列的微操作没有次序关系，这给调度器提供了更大的灵活性。

**4. 微操作调度和派发**

调度器负责由队列取出微操作并派发它们去执行。调度器查找那些其状态指明已具备自己全部操作数的微操作，若它所需的执行单元可用，则调度器取出此微操作，并将它派发到相应的执行单元。每周期能派发多达6个的微操作。如果多个微操作要使用同一个执行单元，调度器将按队列顺序逐个派发它们。这也是一种FIFO规则，偏向按序执行；但此时指令流已被相关性和分支重新排列了，实际上它已是乱序了。

调度器有4个端口与执行单元连接。端口0用于整数和浮点运算，但简单整数运算不在其内，端口1用于简单整数运算和分支预测失误处理。另外，几个MMX执行单元有的在端口0，有的在端口1，余下的两个端口分别用于存储器装载和保存。

**16.3.3整数和浮点执行单元**

  整数和浮点寄存器组是执行单元待完成操作的数据源之一）。执行单元由寄存器组以及L1数据高速缓存取出所需的值。一个单独的流水线阶段专门用于计算标志（如零、负等），这些值一般都是分支指令所需要使用的值。

下一个流水线阶段完成分支检查，它将分支的实际结果与预测进行比较。在最后的驱动（drive）阶段期间实现分支检查结果）。如果预测是错的，那么在各个阶段正在进行的微操作必须从流水线中清除掉。正确的目标地址被提供给分支预测器（branch predictor），从而由新的目标地址重新启动整个流水线。

**16.4 ARM CORTEX-A8**

ARM体系结构最近的实现已经开始在指令流水线中采用超标量技术。在本节中，我们将几种讨论ARM Cortex-A8，它提供了一个基于RISC的超标量设计的很好实例。

Cortex-A8在ARM系列处理器中被称为是应用处理器。ARM公司的应用处理器是指那些运行复杂操作系统的嵌入式处理器，它们主要的应用是无线通信，消费电子以及图像处理等。Cortex-A8的目标定位于各种移动和消费电子应用，包括手机、机顶盒、游戏机以及汽车导航/娱乐系统。

图16-9显示了Cortex-A8处理器的逻辑结构，并突出了其中各个功能单元之间的指令流。主要的指令流在3个功能单元之间，这3个功能单元实现了一个有13个阶段，按序发射的双流水线。Cortex设计人员决定采用按序发射是为了使所需的功耗保持最低。乱序发射和回收需要大量的逻辑电路来实现，从而消耗更多的电能。

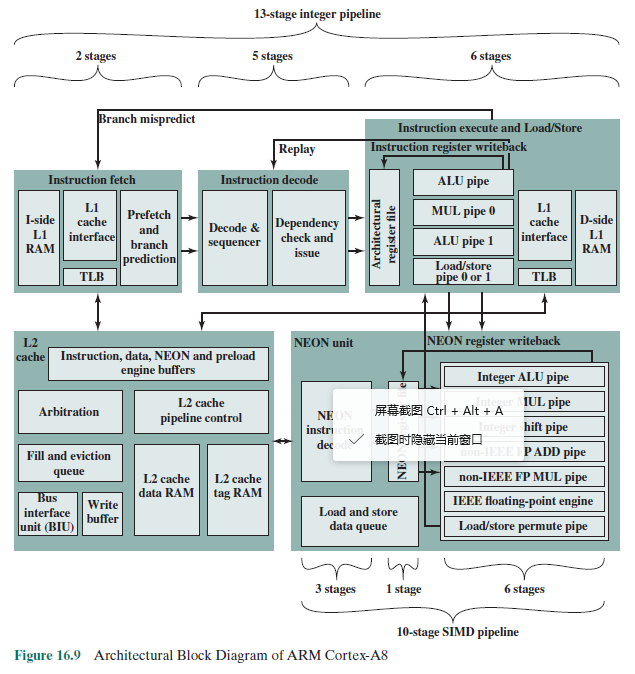
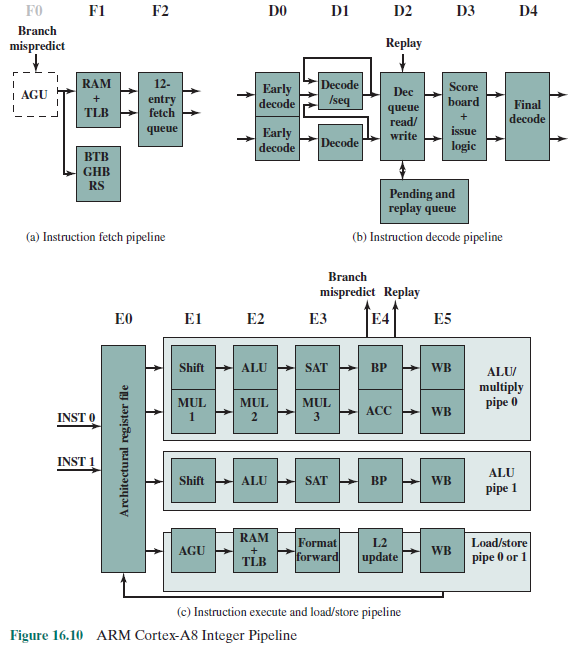


图16-10示了Cortex-A8主流水线的详细结构。Cortex-A8另外还有一个单独的SIMD（单指令多数据，single-instruction-mutiple-data）单元。该单元被实现为一条10阶段的流水线。

**16.4.1 指令取指单元**

指令取指单元（instruction fetch unit）会测指令流，从L1指令高速缓存中取指，并把取来的指令放到缓冲器中，以便译码流水线对指令进行译码。L1指令高速缓存包含在指令取指单元中。由于流水线中允许有若干未确定的分支指令，因此指令取指是推测性的。这意味着取来的指令不一定会被执行。代码流中的分支指令和发生异常的指令会导致流水线清空，丢弃当前取来的指令。指令取指单元每周期可以取来多达4条指令，取指操作经过如下的这些阶段：



**F0** 地址生成单元（Address Generation Unit，AGU）生成一个新的虚拟地址。通常这个地址是上一个地址的顺序后继地址。它也可以是分支转移目标地址，该地址由分支预测器对前一条指令的预测而产生。F0不作为一个阶段计入13阶段流水线，因为ARM处理器传统上把指令高速缓存访问当作流水线的第一阶段。

**F1** 计算得到的地址用于从L1指令高速缓存中取指。与此同时，该地址也用于访问分支预测阵列，以便确定下一个取指地址是否应该基于分支预测产生。

**F3** 取来的指令被放到指令队列中。如果一条指令引起了分支预测动作，那么新的目标地址会被送到地址生成单元。

为尽量减少由较深流水线带来的较大转移开销，Cortex-A8处理器实现了一个两级全局预测器。该预测器由转移目标缓冲器（Branch Target Buffer，BTB）和全局历史缓冲器（Global History Buffer，GHB）组成。这些数据结构在指令取指的同时被并行访问。BTB会出当前取指地址是否会是一个分配指令，并给出对应的分支转移目标地址。BTB包含512项。如果取指地址与其中一项匹配，就触发一个分支预测动作，并将使用到GHB。GHB包含4096个2位的计数器。这些计数器记录了分支转移的方向及强度信息。GHB使用最近10次分支转移方向的一个10位历史记录和PC中的4位一起作为索引。除了动态分支预测器之外，取指单元还使用了一个返回栈，来预测子过程返回地址。返回栈有8个32位的项，每项保存了一个连接寄存器r14中的值，以及调用函数的ARM指令或压缩（Thumb）状态。当一个返回类型指令被预测为要发生转移时，返回栈就提供最后被压入栈的地址和状态。

指令取指单元可以取指和入队多达12条指令，并能在同一时间发射两条指令到译码单元。指令队列使得指令取指单元能够先于整数流水线其他阶段去取指令，形成一批积压的指令等待译码。

**16.4.2 指令译码单元**

指令译码单元对所有的ARM指令和压缩指令进行译码并排序。译码单元有一个双流水线结构，称为流水线0和流水线1。这样在同一时间，可以有两条指令通过译码单元。指令译码单元发出两条指令时，流水线0总是包含了程序顺序中靠前的那条指令。这意味着如果流水线0中的指令不能发射的话，那么流水线1中的指令也不会发射。一旦发射，所有被发射指令按序进入到执行流水线，并在执行流水线末尾把结果写入到寄存器组中。这种按序发射，按序完成的方式避免了WAR冒险，同时能直接记录WAW冒险，也能从流水线清空条件中恢复。这样以来，指令译码单元主要的考虑就是如何避免RAW冒险了。

每条指令将通过如下5个阶段的操作。

**D0** 压缩指令被解压缩到32位的ARM指令。初始的译码功能被执行。

**D1** 继续完成指令译码功能。

**D2** 这一阶段把译码后的指令写入等待/重放指令队列，并从等待/重放队列中读出指令送往下一阶段。

**D3** 这一阶段包含了指令调度逻辑。其中一个记分牌根据静态调度技术[[3]](#footnote-4)预测寄存器的可用性。本阶段同时检查各种情况的冒险。

**D4** 完成最后的译码，产生整数执行及装载/保存单元需要的所有控制信号。

在最开始的两个阶段，将确定指令类型、源操作数和目的操作数，以及指令的资源需求。ARM指令中有一些不常使用的指令，称为多周期指令。D1指令会把这些指令分开成为多个指令操作码，这些指令操作码将被分别排序通过执行流水线。

等待（pending）队列起到两个作用。首先，它避免来自D3阶段的流水线停顿信号进一步扩散从而影响流水线的运行。第二，通过缓冲指令，双流水线应该总是有两个可用的指令。在仅发出一条指令的情况下，挂起队列使两条指令能够一起沿着流水线进行下去，即使它们最初是以不同周期从取出单元发送的。

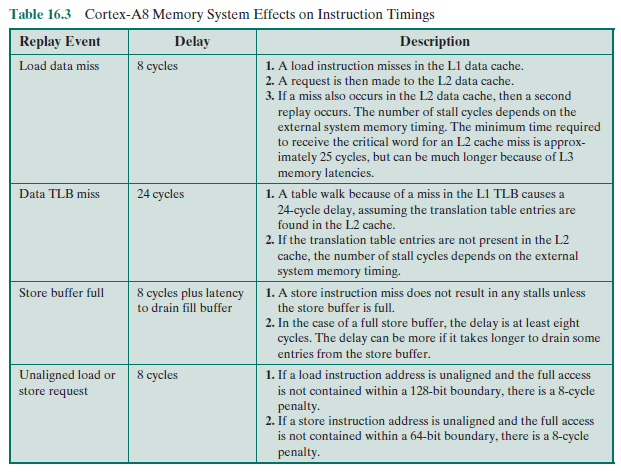
重放（replay）操作是为了处理存储器系统对指令定时的影响而设计的。指令基于源操作数何时可用的预测，在D3阶段进行静态调度。存储系统的任何停顿都会导致一个不少于8个周期的延迟。这个最小8周期的延迟对应于L1装载缺失时，从L2告诉缓存接收数据所需的最少可能周期数。表16-3给出了由于内存系统停止而导致指令重放的最常见情况。

为了处理这些停顿，指令译码单元使用了一个恢复机制，该机制先清空执行流水线中所有的后续指令，然后重新发射（重放）它们。为支持重放，指令在发射之前会被拷贝到重放队列中，直到它们写回了它们的执行结果并完成，才会从重放队列中删除。如果重放信号被置位，指令就从重放队列中被读出，并重新放入流水线。

译码单元并行地发射两条指令到执行单元，除非译码单元碰到了发射限制情况。表16-4显示了最常见的限制情况。

**16.4.3 整数执行单元**

指令执行单元由两个对称的算术逻辑单元（ALU）流水线组成，其中一个服务于装载/保存指令的地址生成器，另一个是乘法流水线。执行流水线也进行寄存器写回操作。下面是指令执行单元的功能：



·执行所有的整数ALU和称法操作，包括标志的生成。

·为装载和保存指令生成虚拟地址。如有需要，生成写回基值（base write-back value）。

·为保存指令提供格式化后的数据，并转发数据及标志。

·处理分支以及其他对指令流的改变，并计算指令条件码。

对于ALU指令，可以使用两条流水线的任何一条，其执行包括如下阶段：

**E0** 访问寄存器组。对于两条指令，最多需要从寄存器组读出6个寄存器值。

**E1** 如果需要的话，桶移位器（见图14-25）执行移位操作。

**E2** ALU（见图）执行算术逻辑运算。

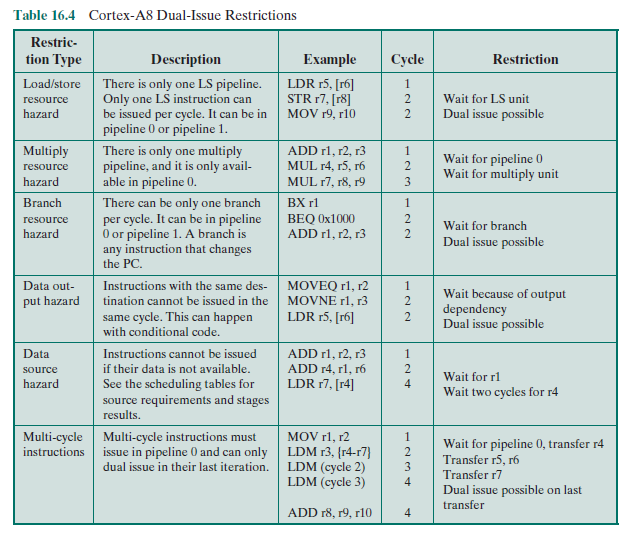
**E3** 如果需要的话，该阶段完成某些ARM数据处理指令所使用的饱和运算。

**E4** 如果控制流发生任何改变，包括分支预测失误、异常以及存储系统重放，那么该阶段保证这些情况被优先处理。

**E5** ARM指令的执行结果被写回到寄存器组。

使用乘法单元（见图14-25）的指令被放入到流水线0中处理。乘法操作在E1阶段到E3阶段进行，乘积累加在E4阶段执行。

装载/保存流水线与整数流水线并行运行。装载/保存流水线包括如下阶段：



**E1** 存储地址从基址和变址寄存器产生。

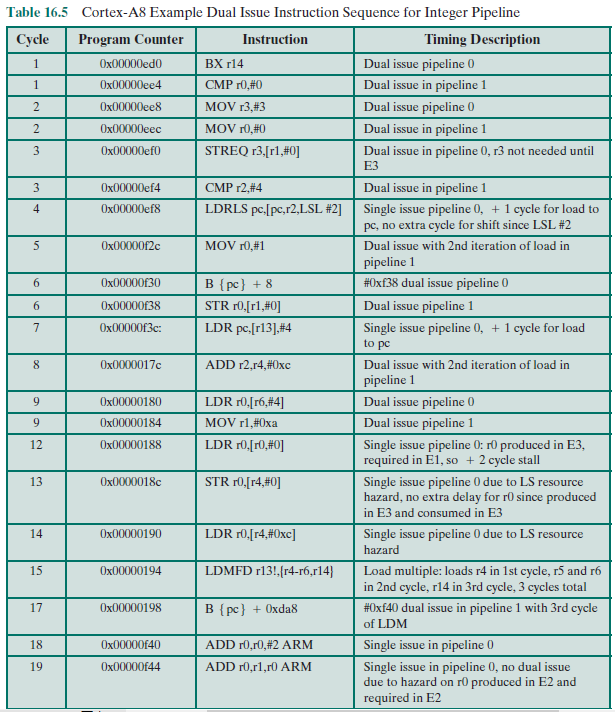
**E2** 地址被用于高速缓存阵列的访问。

**E3** 对于装载指令，数据被返回并被格式化，以便转发给ALU或乘法单元。对于保存指令，数据被格式化，以便写入到高速缓存中。

**E4** 如果需要，对L2高速缓存进行更新。

**E5** ARM指令的执行结果被写回到寄存器组。

表16-5显示了一个示例代码片段，并指出了处理器可能会如何来调度它。



**16.4.4 SIMD和浮点流水线**

所有的SIMD和浮点指令都通过整数流水线并由一个单独的10阶段流水线处理（见图16-11）。这个称为NEON的单元，能处理压缩的（packed）SIMD指令，并提供了两种类型的浮点处理支持。根据实际实现，该单元可能带有一个向量浮点（vector floating-point，VFP）协处理器，负责完成遵循IEEE 754标准的浮点操作。如果实际实现中不带这个协处理器，那么会有一条单独的乘法和加法流水线来实现浮点运算。

![C:\Users\Grantaire\AppData\Roaming\Tencent\Users\122695959\QQ\WinTemp\RichOle\5D_039](AXVH{@6H8JDR]GC.png](data:image/png;base64,)

**16.5 ARM CORTEX-M3**

上一节介绍了应用程序处理器Cortex-A8相当复杂的流水线组织结构。作为有用的对比，本节研究相当简单的Cortex-M3的流水线组织。Cortex-M系列是专为微控制器领域设计的。因此该系列的处理器需要尽可能简单和有效。

图16-12提供了Cortex-M3处理器的框图概述。该图提供了比图1.16所示的更详细的信息。关键要素包括：

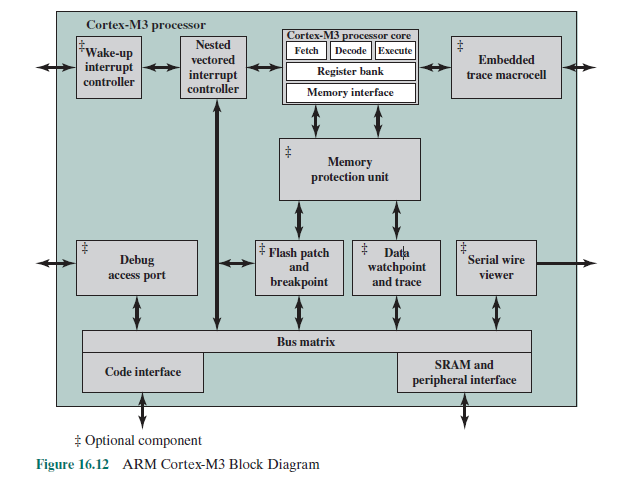
**·处理器核心**（Processor Core）：包括三级流水线、寄存器库和存储器接口。

**·内存保护单元**（Memory protection unit）：保护操作系统使用的关键数据不受用户应用程序的影响，通过禁止对彼此数据的访问、禁用对内存区域的访问、允许将内存区域定义为只读、以及检测可能会破坏内存区域的对系统的意外内存访问。

**·嵌套矢量中断控制器**（Nested vectored interrupt controller，NVIC）：向处理器提供可配置的中断处理能力。它有助于实现低延迟异常和中断处理，并控制电源管理。

**·唤醒中断控制器**（Nested vectored interrupt controller，NVIC）：向处理器提供可配置的中断处理能力。它有助于实现低延迟异常和中断处理，并控制电源管理。

**·Flash补丁和断点单元**（Flash patch and breakpoint unit）：实现断点和代码修补程序。



**·数据监视点和跟踪**（Data watchpoint and trace，DWT）：实现监视点、数据跟踪和系统概要分析。

**·串行线观察器**（Serial wire viewer）：可以通过单个管脚导出软件生成的消息流、数据跟踪和分析信息。

**·调试访问端口**（Debug access port）：提供用于外部调试访问处理器的接口。

**·嵌入式跟踪宏单元**（Embedded trace macrocell）：是一个应用程序驱动的跟踪源，它支持printf()样式的调试以跟踪操作系统和应用程序事件，并生成诊断系统信息。

**·总线矩阵**（Bus matrix）：将核心和调试接口连接到微控制器上的外部总线。

**16.5.1 流水线结构**

CORTEX-M3流水线结构有三个阶段（图16-12）。我们将依次检查这三个阶段。

在取指阶段，一次性获取一个32位的字，并将其加载到一个3字长的缓冲器。32位字可包括：

·两个压缩（Thumb）的指令；

·一个字对齐的Thumb-2指令，或；

·半字对齐的Thumb-2指令的上/下半字与

—一个Thumb指令，或

—另一个半字对齐的Thumb-2指令的下半字/上半字。

从核心获取的所有地址都是字对齐的。如果一条Thumb-2指令是半字对齐的，则需要两个取指来获取Thumb-2指令。无论如何，三项预取缓冲区确保了仅获取的第一个半字Thumb-2指令需要中断周期。

该译码阶段执行三个关键功能：

·**指令译码和寄存器读取**（Instruction decode and register read）：译码压缩指令和Thumb-2指令。

·**地址生成**（Address Generation）：地址生成单元(Address generation unit，AGU)为装载/保存单元生成主存储器地址。

·**分支**（Branch）：在分支指令中基于立即偏移执行分支，或者基于链接寄存器(寄存器R14)的内容执行返回。

在指令执行的最后只有一个执行阶段，包括ALU、加载/存储和分支指令。

**16.5.2 处理分支问题**

为了使处理器尽可能地保持简单，Cortex-M3处理器不使用分支预测，而是使用分支转发和分支推测的简单技术，其定义如下：

·**分支转发**（Branch forwarding）：转发（fowarding）这个术语指的是呈现要从内存中取出的指令地址。处理器转发某些分支类型，通过这些类型，分支的内存事务比操作码到达执行时提前至少一个周期被呈现。分支转发提高了核心的性能，因为分支是嵌入式控制器应用程序的重要组成部分。受影响的分支是具有即时偏移的PC，或者使用链接寄存器（LR）作为目标寄存器。

·**分支投机**（Branch speculation）：对于条件分支，推测性地预先发送指令地址，以便在知道指令是否将被执行之前从存储器中取出指令。

Cortex-M3处理器使用预取指令操作，在执行之前将指令取出缓冲区，还从分支目标地址进行推测性预取指。具体一些，即当遇到条件分支指令时，译码阶段还包括推测性指令获取，这可使执行速度更快。处理器在译码阶段期间自行获取分支目标地址指令。然后在执行阶段中，解析该分支，明确接下来要执行哪个指令。

如果不采取分支，则下一个顺序指令已经备好可用。如果要进行分支指令，则在作出决定的同时使分支指令可用，从而将空闲时间限制为一个周期。

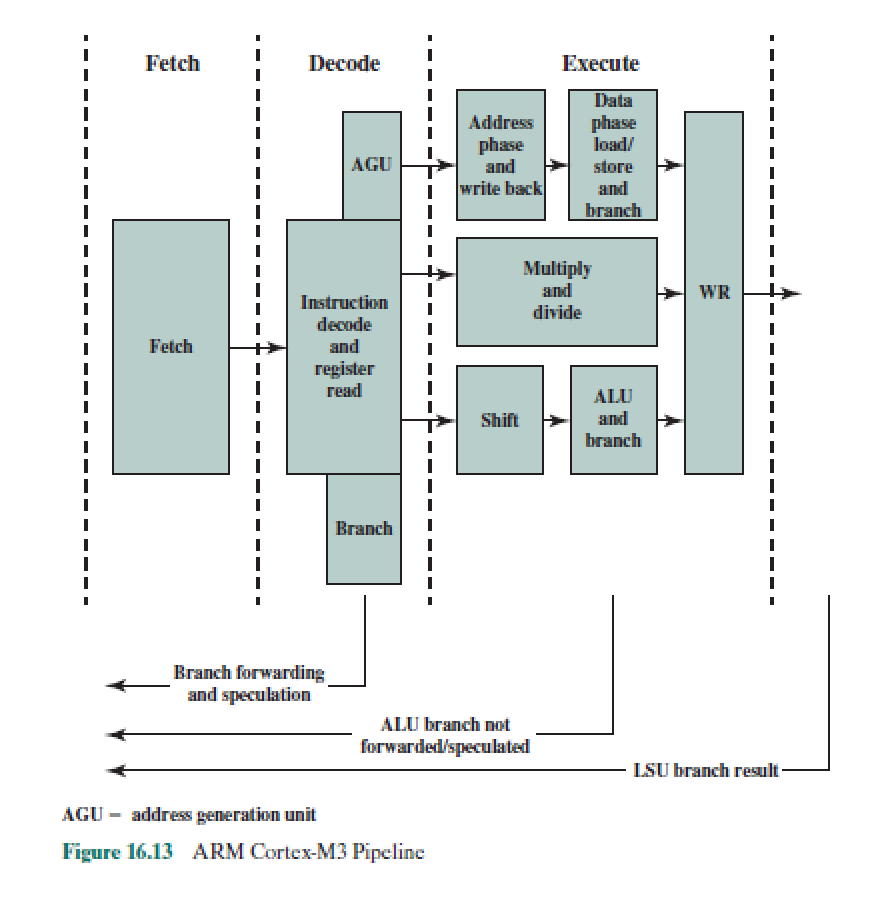
图16-13阐明了分支的处理方式，其描述如下：

1. 译码阶段由无条件分支节点转发地址，并在有可能推算出地址时从条件分支推测性转发地址。

2. 如果ALU确定没有采用分支，则反馈此信息以清空指令缓存。

3. 对程序计数器的加载指令导致要转发用于获取的分支地址。

可以看出， Cortex-M处理分支条件的方式比Cortex-A要简单得多，需要更少的处理器逻辑和处理过程。



**16.6关键词、思考题和习题**

**关键词**

antidependency：反相关性

branch prediction ：分支预测

commit：托收（或提交）

flow dependency：流相关性

in-order issue：按序发射

in-order completion：按序完成

instruction issue：指令发射

instruction-level parallelism：指令级并行性

instruction window：指令窗口

machine parallelism：机器并行性

micro-operations：微操作

micro-ops：微操作

out-of-order completion：乱序完成

out-of-order issue：乱序发射

output dependency：输出相关性

procedural dependency：过程相关性

read-write dependency：读写相关性

register renaming：寄存器重命名

resource conflict：资源冲突

retire：回收

superpiplelined：超级流水线式

superscalar：超标量

true data dependency：真数据相关性

write-read dependency：写读相关性

write-write dependency：写写相关性

**思考题**

16.1 处理器超标量设计方法的本质特征是什么？

16.2 超标量与超级流水线的区别是什么？

16.3 什么是指令级并行性？

16.4 简要定义如下术语：

·真数据相关性

·过程相关性

·资源冲突

·输出相关性

·反相关性

16.5 指令级并行性与机器并行性有何区别？

16.6 列出并简要定义超标量指令的三种发射策略。

16.7 指令窗口的用途是什么？

16.8 什么是寄存器重命名？它的目的何在？

16.9 超标量机器组织的关键部件是什么？

**习题**

16.1 当超标量处理器采用乱序完成时，中断处理后的恢复执行被复杂化了，因为被检测到异常的条件可能会是一条乱序完成指令的结果。程序不能以异常指令之后的顺序指令来重新启动，因为该后续指令可能已经完成，这样做会导致该指令被执行两次。请提出一种机制以处理这种情况。

16.2 考虑如下指令序列，它的句法是：操作码之后是一个目标寄存器，再其后是一个或两个源寄存器：

0 ADD R3, R1, R2

1 LOAD R6, [R3]

2 AND R7, R5, 3

3 ADD R1, R6, R7

4 SRL R7, R0, 8

5 OR R2, R4, R7

6 SUB R5, R3, R4

7 ADD R0, R1, 10

8 LOAD R6, [R5]

9 SUB R2, R1, R6

10 AND R3, R7, 15

假设使用4阶段流水线：取指、译码/发射、执行、写回，并假设除了执行阶段以外，所有流水段都花费1一个时钟周期。对于简单的整数算术和逻辑指令，执行段花费1个时钟周期，但是对于由存储器装载（LOAD），执行段要花费5个时钟周期。

若此简单的标量流水线具有乱序执行能力，那么我们可以为前7条指令的执行构造下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令编号 | 取指 | 译码 | 执行 | 写回 |
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 4 | 9 |
| 2 | 2 | 3 | 5 | 6 |
| 3 | 3 | 4 | 10 | 11 |
| 4 | 4 | 5 | 6 | 7 |
| 5 | 5 | 6 | 8 | 10 |
| 6 | 6 | 7 | 9 | 12 |

表中4个流水线阶段下的项，指示每条指令在每个阶段开始的时钟周期。在这个程序中，第二个ADD指令（指令3的一个操作数r6依赖于LOAD指令（指令1））。因为LOAD指令需要5个时钟周期来执行，且发射逻辑在2个时钟周期后遇到这条相关的ADD指令时，它必须延迟三个时钟周期再发射ADD指令去执行。利用其乱序完成能力，处理器可以在时钟周期4停止指令3发射时，转而发射以下三条独立的指令，并使它们分别以进入时钟6、8和9进入执行段。LOAD在时钟9完成执行后，相关的ADD指令就可以在时钟10被发射去执行了。

（1）完成前面的表格。

（2）假设没有乱序完成的能力，重做此表；乱序完成的能力节省了多少时间？

（3）重做该表，假设一个超标量实现可以在每个流水段同时处理两条指令。

16.3 考虑如下汇编语言程序：

I1: LOAD R2, (R4) /R2 ← Memory (R4)/

I2: BLE R0 R2 L2 /Branch if (R2) > (R0)/

I3: ADD R2, R2, 6 /R2 ← (R2) + 6/

I4: MOVE R1, R2 /R1 ← (R2)/

I5: LOAD R1, (R5) /R1 ← Memory (R5)/

该程序包括了写后写（WAW）、写后读（RAW）、读后写（WAR）相关性，请指明。

16.4 （a）在下面的指令序列中找到写后读相关、写后写相关和读后写相关：

I1: R1 = 100

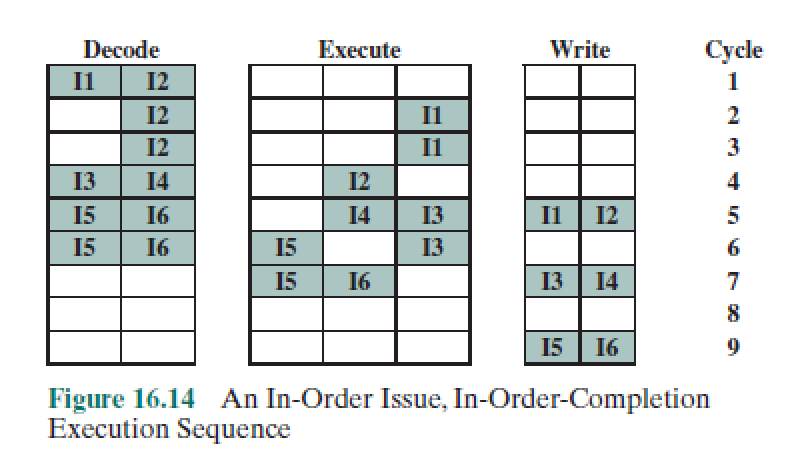
I2: R1 = R2 + R4

I3: R2 = r4 - 25

I4: R4 = R1 + R3

I5: R1 = R1 + 30

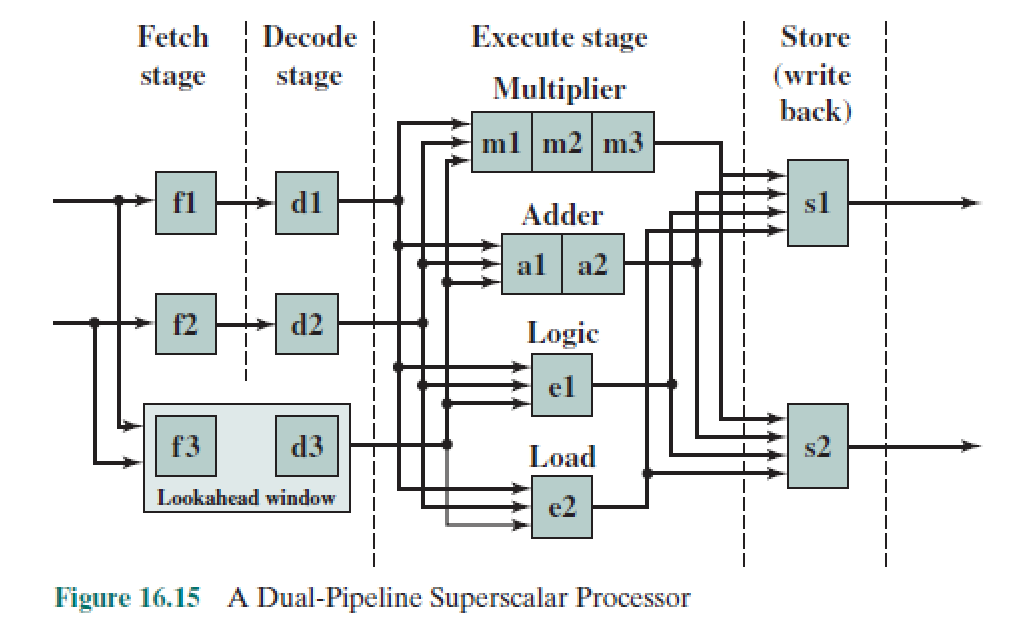
（b）重命名以上（a）中寄存器，消除相关性的问题。对于原始寄存器值的引用，在寄存器引用中加以下表“a”进行标注。



16.5 考虑图16-14中所示的“按序发射/按序完成”执行序列：

（a）指出指令I2不能在第4个周期前进入执行阶段最可能的原因。若采用“按序发射/乱序完成”或“乱序发射/乱序完成”策略，能解决此问题吗？如果能，哪个策略解决了这个问题？

（b）指出指令I6在第9个周期之前不能进入写入阶段的原因。若采用“按序发射/乱序完成”或“乱序发射/乱序完成”策略，能解决此问题吗？如果能，哪个策略解决了这个问题？



16.6 图16-15显示了超标量处理器组织的示例。如果没有资源冲突和数据相关问题，处理器每个周期可以发出两条指令。这里基本上有两条流水线，每条流水线又有4段（取指、译码、执行和保存），并都有自己取指、译码和保存单元。4个功能单元（乘法器、加法器、逻辑单元和装载单元）可用于执行阶段，且可由两个流水线动态共享。两个存储单元能被两条流水线动态使用，取决于具体周期时的可用性。这里还有一个先行窗口（lookahead window）它有自己的取指和译码逻辑。此窗口用于乱序发射指令的先行查找。

考虑在此处理器上执行下面的程序：

I1: Load R1, A /R1← Memory (A)/

I2: Add R2, R1 /R2 ← (R2) + R(1)/

I3: Add R3, R4 /R3 ← (R3) + R(4)/

I4: Mul R4, R5 /R4 ← (R4) + R(5)/

I5: Comp R6 /R6 ← (R6)/

I6: Mul R6, R7 /R6 ← (R6) \* R(7)/

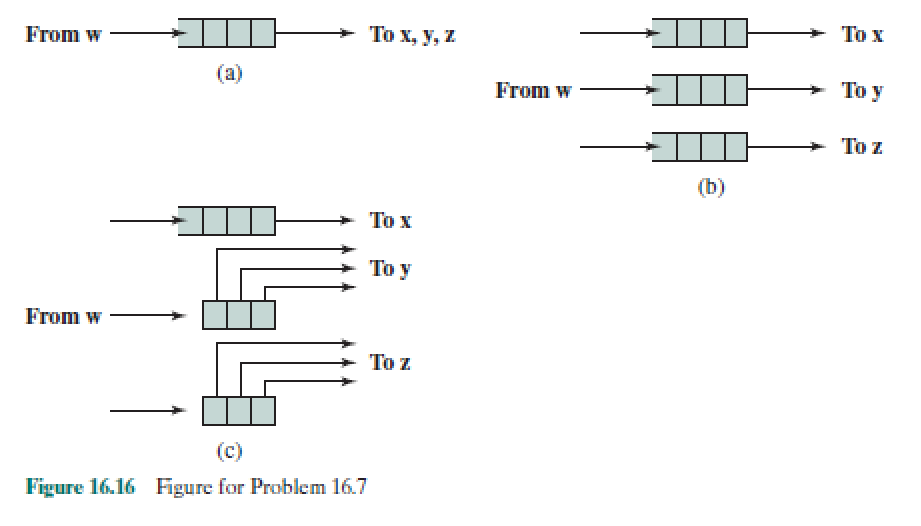
（a）程序中存在什么相关性？

（b）请给出这个程序在图16-15的处理器上运行时的流水线操作，使用类似于图16-2的表示法。首先考虑采用按序发射、乱序完成策略。

（c）再考虑采用按序发射、乱序完成策略。

（d）最后，考虑采用乱序发射、乱序完成策略。

16.7 图16-16摘自一篇超标量设计方面的论文，请解释图中的3个部分，并定义w、x、y和z。

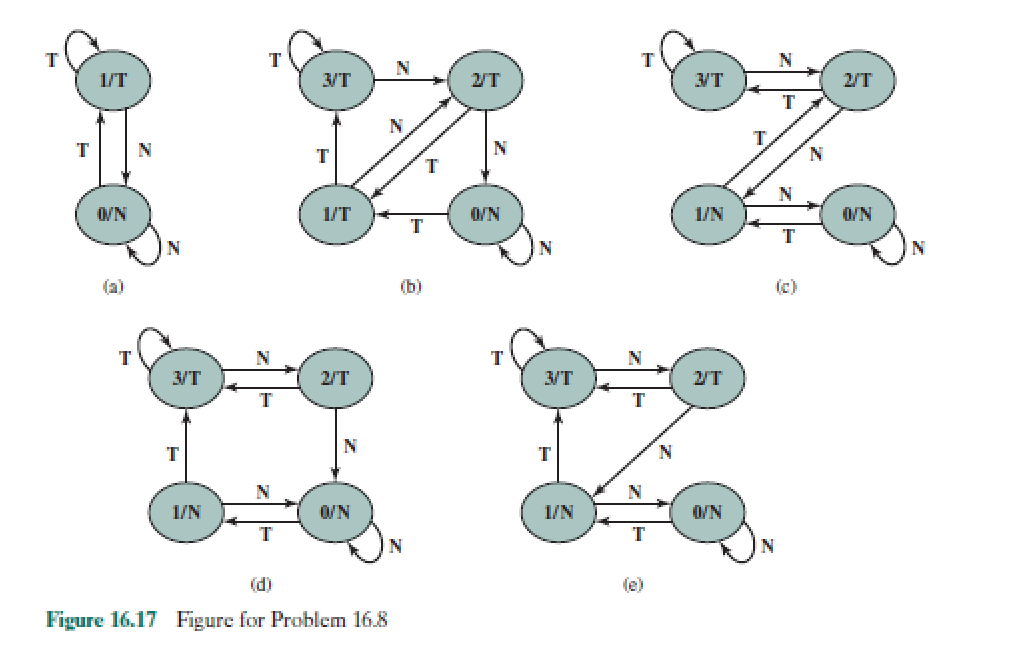


16.8 用于Pentium 4上的Yeh动态分支预测算法是一种两级分支预测算法。第一级是最后*n*次转移的历史。

第二级是此*n*此转移最后出现s次独特样式的转移行为，最后一次出现的分支模式是唯一的。程序中的每个条件分支指令在分支历史表（Branch History Table，BHT）中都有一对应项。每项由*n*位组成，相应于该分支之林最后*n*次的执行；如果分支被执行，则置1，如果分支未执行，则置0。每个BHT项可索引到一个模式表（Pattern Table，PT）中，该模式表有2n个项，每种可能的*n*位样式有一项。每个PT项由*s*位用于分支预测的位组成，如第14章（例如，图14-19）所述。在指令提取和译码期间遇到条件分支时，此指令的地址用取出一个相对应的BHT项，该项表示该指令的历史信息。然后，该BHT项用于取出相对应的PT项，以进行分支预测。在执行此分支指令后，更新此BHT项，然后相对应的PT项亦更新。

（a）在测试这个方案的性能时，Yeh曾尝试了5种不同的预测方案，如图16-17所示。请指出这之中哪三个方案与图14-19和14-28中所示的方案一致，并请描述其余两种方案。

（b）依照这种算法，预测并不只基于某特定分支指令的历史，还要基于匹配该指令BHT表项*n*位样式的所有转移样式的最近历史。请说明这种策略的原理。



1. 对于Intel x86汇编语言来说，每行的分号后为注释。 [↑](#footnote-ref-2)
2. 关于重排序缓冲器的介绍请见附录N [↑](#footnote-ref-3)
3. 有关记分牌技术的讨论请见附录N。 [↑](#footnote-ref-4)