处理器微体系结构——一个实现角度

处理器微体系结构——一个实现视角

摘要

本文主要介绍当代微处理器微体系结构的研究。重点是实现方面，并讨论它们在性能、功耗和最先进设计成本方面的影响。本文首先概述了不同类型的微处理器，并回顾了cache存储器的微体系结构。然后，它描述了取指单元的实现，其中特别强调了对转移预测所需的支持。接下来的一节专门介绍指令译码，重点介绍对x86指令译码的特殊支持。下一章介绍了分配阶段，并特别关注寄存器重命名的实施。然后，对发射阶段进行了研究。在这里，对内存和非内存指令实现乱序发射的逻辑进行了详细描述。接下来的一章重点介绍指令执行，并描述当代微处理器中可以找到的不同功能部件，以及对性能有重要影响的旁路网络的实现。最后，本文以提交阶段结束，在该阶段中，它描述了在出现异常或错误推测时如何更新和恢复体系结构状态

本文旨在提供一门关于计算机体系结构的高级课程，适合于研究生或想要专门研究计算机体系结构领域的高年级本科生。它也适用于微处理器设计领域的行业从业人员。本书假设读者熟悉有关流水线、乱序执行、cache存储和虚拟内存的主要概念。

关键字

处理器微体系结构、cache存储、指令获取、寄存器重命名、指令译码、指令发射、指令执行、错误推测恢复

processor microarchitecture, cache memories, instructions fetching, register renaming, instruction decoding, instruction issuing, instruction execution, misspeculation recovery

---

第一章 介绍

计算机是当今大多数活动的核心。处理器是计算机的核心部件，但如今，我们可以在许多其他部件中找到嵌入式处理器，比如游戏机、消费电子设备和汽车，仅举几例。

本文对当代微处理器的微体系结构进行了深入研究。重点是实现方面，讨论替代方法及其在性能、功耗和成本方面的影响。

处理器的微体系结构经历了不断的演变。例如，英特尔最近大约每年都会推出一款新的微处理器。这种演变主要由两类因素推动：（1）工艺扩展和（2）工作负载演变。

工艺扩展通常被称为摩尔定律，它基本上表明晶体管密度大约每两年翻一番。每一代技术都提供更小、更快、能耗更低的晶体管。这使得设计人员能够提高处理器的性能，即使不增加其面积和功耗。

另一方面，处理器调整其功能，以便更好地利用用户应用程序的特性，这些特性会随着时间的推移而变化。例如，近年来，我们见证了多媒体应用程序使用的惊人增长，这导致处理器中的功能越来越多，以更好地支持它们。

1.1 微体系结构的分类

处理器微体系结构可以沿着多个正交维度进行分类。在这里，我们将介绍最常见的。

1.1.1 流水线/非流水线处理器

流水线处理器将每条指令的执行分为多个阶段，并允许在不同阶段同时处理不同的指令。流水线提高了指令级并行性（ILP），并且由于其成本效益，它实际上被当今所有处理器使用。

1.1.2 按序/乱序处理器

按序处理器按照指令在二进制文件中出现的顺序（根据指令的顺序语义）处理指令，而乱序处理器按照与二进制文件中指令不同（通常是不同）的顺序处理指令。乱序执行指令的目的是通过为硬件提供更多自由来选择在每个周期中处理哪些指令，从而增加ILP的量。显然，乱序处理器比按序处理器需要更复杂的硬件。

1.1.3 标量\超标量处理器

标量处理器是指在至少一个流水线级中不能执行超过1条指令的处理器。换言之，对于任何代码，标量处理器的吞吐量都不能超过每周期1条指令。非标量的处理器称为超标量处理器。请注意，超标量处理器可以在所有流水线级中同时执行多条指令，因此对于某些代码，其吞吐量可以高于每周期1条指令。

超长指令字（VLIW）处理器是超标量处理器的一种特殊情况。这些处理器可以在所有流水线阶段处理多条指令，因此它们符合超标量的定义。使超标量处理器成为VLIW的是以下特性：（a）它是按序处理器，（b）二进制代码指示哪些指令将并行执行，（c）许多执行延迟暴露给程序员，并成为指令系统体系结构的一部分，因此，代码必须遵守一些关于特定类型指令之间距离的约束，以确保正确执行。这些约束的目的是简化硬件设计，因为它们避免了在运行时检查某些操作数的可用性并决定在每个周期中发射哪些指令的硬件机制。

例如，在每个周期执行4条指令的VLIW处理器中，代码由4条指令的数据包组成，每一条指令都必须是特定类型的。此外，如果一个给定的操作需要三个周期，则代码生成器有责任确保接下来的两个数据包不使用该结果。

换句话说，在非VLIW处理器中，代码的语义仅由指令的顺序决定，而在VLIW处理器中，如果不知道硬件的某些特定特征（通常是功能单元的延迟），就无法完全推导代码的语义。通过公开一些硬件特性作为体系结构定义的一部分，VLIW处理器可以有一个更简单的设计，但另一方面，使代码依赖于实现，因此，它可能无法从一个实现兼容到另一个实现。

1.1.4 向量处理器

向量处理器是指在其ISA（指令系统体系结构）中包含大量能够对向量进行操作的指令的处理器。传统上，向量处理器的指令操作相对较长的向量。最近，大多数微处理器都包含一组丰富的指令，这些指令在相对较小的向量上运行（例如，英特尔AVX扩展中最多有8个单精度FP元素）。这些指令通常被称

为SIMD（单指令多数据）指令。根据这一定义，如今的许多处理器都是向量处理器，尽管它们对向量指令的支持程度在不同处理器之间有很大差异。

1.1.5 多核处理器

处理器可以由一个或多个内核组成。核心是一个可以处理连续代码段（通常称为线程）的单元。传统处理器过去只有一个内核，但现在大多数处理器都有多个内核。多核处理器可以同时处理多个线程，每个线程使用不同的硬件资源，并支持允许这些线程在程序员的控制下同步和通信。这种支持通常包括内核之间的某种类型的互连，以及通过这种互连进行通信和通常用于共享数据并一致地维护它们的一些原语。

1.1.6 多线程处理器

多线程处理器是一种可以在某些内核上同时执行多个线程的处理器。请注意，多核和多线程处理器都可以同时执行多个线程，但关键的区别在于，在多核的情况下，线程使用的硬件资源大部分不同，而它们共享多线程处理器中的大部分硬件资源。

多核和多线程是两个正交的概念，因此它们可以同时使用。例如，Intel Core i7处理器有多个内核，每个内核都是两路多线程的

1.2 细分市场的分类

处理器也有不同的特点，这取决于它们所针对的细分市场。最常见的细分市场分类如下：

- 服务器：这部分指的是数据中心中功能强大的系统，通常由许多用户共享，通常有大量处理器。在这一部分中，计算能力和功耗是用户最重要的参数。
- 桌面：这个术语指的是在家里或办公室使用的计算机，通常由不超过一个用户同时使用。在这些系统中，计算能力和散热方案的噪声通常是用户最重要的参数。
- 移动：指的是笔记本电脑，也被称为notebooks，其主要功能是移动性，因此，它们大部分时间使用电池工作。在这些系统中，能耗是用户最重要的参数，因为它会影响电池寿命，但计算能力也非常重要。
- 超级移动：如手机。在这些系统中，能耗对用户来说至关重要。计算能力很重要，但相对能耗而言是次要的。这些系统通常非常小，以最大限度地提高其便携性。
- 嵌入式：这部分指的是除了计算机之外，我们现在使用的许多系统中嵌入的处理器。这些嵌入式处理器几乎无处不在：在汽车、消费类电子产品、医疗器械等中。它们的特性因其嵌入的特定系统而异。在某些情况下，它们的计算能力要求可能很重要（例如，在机顶盒中），而在许多其他情况下，成本是最重要的参数，因为它们的计算要求最低，这一切都是为了将它们对产品总成本的影响降到最低。一些嵌入式处理器在移动系统中，在这种情况下，能耗也至关重要。

1.3 处理器概述

图1.1显示了处理器主要组件的高级框图，该框图适用于当今大多数处理器，特别是代表最常见组成的乱序超标量处理器。它还描述了每一条指令按顺序执行的主要阶段。请注意，这些阶段不一定对应于

流水线阶段；一个特定的实现可以将它们中的每一个分割成多个阶段，或者可以将其中的几个组合成同一个阶段。

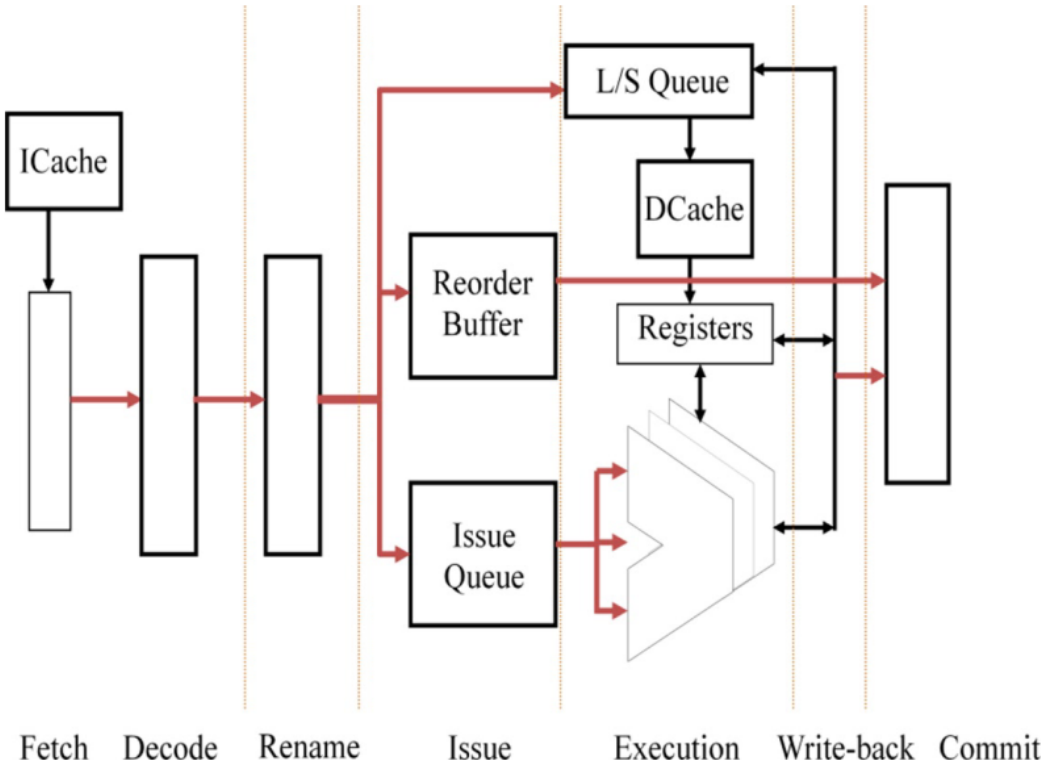

FIGURE 1.1: High-level block diagram of a microprocessor.

指令首先从指令cache中获取。然后对它们进行译码以理解它们的语义。之后，大多数处理器对寄存器操作数应用某种类型的重命名，以消除错误依赖，并增加可利用的ILP量。然后，根据指令的类型，指令被派遣到不同的缓冲区。非内存指令被派遣到发射队列和重排序缓冲区，而内存指令则被派遣到load/store队列，以及前两个队列。指令将保留在发射队列中，直到它们被发射执行。在执行指令之前必须读取操作数，但这可以通过多种方式完成，将在第5章和第6章中介绍。然后，将结果写回寄存器堆，最后提交指令。指令保留在重排序缓冲区（ROB）中，直到提交。ROB的目标是存储有关指令的信息，这些信息对指令的执行非常有用，但在必要时也可以撤销指令。

内存操作是以一种特殊的方式处理的。它们需要计算有效地址，这通常与算术指令的过程相同。然而，除了访问数据cache外，它们可能还需要检查它们与其他在执行的内存指令的潜在依赖关系。load/store队列存储为此所需的信息，相关逻辑负责确定何时以及以何种顺序执行内存指令。

在按序处理器中，指令按程序顺序流经这些阶段。这意味着，如果一条指令由于某种原因（例如，一个不可用的操作数）被暂停，后面更迟的指令可能不会超过它，因此它们可能也需要暂停。

在超标量处理器中，上述每个组件都具有同时处理多条指令的能力。此外，在一些流水线级之间添加缓冲区以将它们解耦是很正常的，并且以这种方式允许处理器隐藏由于不同类型的事件（如cache失效、操作数未就绪等）而导致的一些暂停。这些缓冲区在取指和译码、译码和重命名以及派遣和发射之间非常常见。

1.3.1 流水线概述

本节概述了流水线的主要组成部分。下面几章将对它们进行详细描述。

流水线的第一部分负责获取指令。这部分流水线的主要组成部分是（a）存储指令的指令cache，以及（b）确定下一次提取操作地址的转移预测器。

下一部分是指令译码。本部分的主要组成部分是译码器、ROMs和专门电路，其主要目的是识别指令的主要属性，如类型（如控制流）和所需资源（如寄存器端口、功能单元）。

之后，指令流到分配阶段，在分配阶段执行的两个主要操作是寄存器重命名和派遣。寄存器重命名需要更改寄存器操作数的名称，以消除所有错误依赖，从而最大限度地提高处理器可以利用的指令级并行性。这通常是通过一组表来完成的，这些表包含有关当前逻辑名称到物理名称的映射的信息，以及此时未使用的名称，以及一些逻辑，以分析同时重命名的多条指令之间的依赖关系，因为生产者-消费者对的目的和源寄存器必须以一致的方式重命名。指令派遣包括保留指令将使用的不同资源，这些资源包括重排序缓冲区、发射队列和load/store缓冲区中的表项。如果资源不可用，相应的指令将暂停，直到其他指令释放所需的资源。

流水线的下一个阶段将致力于指令发射。指令位于发射队列中，直到发射逻辑确定它们可以开始执行。对于按序处理器，发射逻辑相对简单。它基本上由一个记分板和一个简单的逻辑组成，记分板指示哪些操作数可用，简单的逻辑检查位于队列头部的指令是否已准备好所有操作数。对于乱序处理器，这种逻辑相当复杂，因为它需要在每个周期分析队列中的所有指令，以检查它们的操作数是否准备就绪，以及它们执行所需的资源是否可用。

发射指令后，指令进入流水线的执行部分。这里有各种不同类型操作的执行单元，通常包括整数、浮点和逻辑操作。如今，包含用于SIMD操作（单指令、多数据，也称为向量操作）的特殊单元也很常见。执行流水线的另一个重要组成部分是旁路逻辑。它基本上包括可以将结果从一个单元移动到其他单元的输入的导线，以及确定结果是否应该使用旁路而不是使用来自寄存器堆的数据的相关逻辑。在大多数处理器中，旁路网络的设计是至关重要的，因为线延迟的扩展速度与门延迟的扩展速度不同，因此它们对处理器的时钟周期大小有重要的贡献。

最后，指令进入提交阶段。流水线这一部分的主要目的是提供顺序执行的外观（即相同的结果），即使指令以不同的顺序发射和/或完成。与流水线这一部分相关的逻辑通常包括检查重排序缓冲区中最旧的指令，以查看它们是否已完成。一旦完成，指令就会从流水线中移出，释放资源并进行一些记账。

流水线中影响多个组件的另一部分是恢复逻辑。有时，由于一些误判（典型的情况是转移预测错误），处理器执行的活动必须撤消。发生这种情况时，必须刷新指令，并且必须将某些存储（例如寄存器堆）重置为以前的状态。

本文档的其余部分详细描述了当代处理器不同组件的设计，描述了最先进的设计，有时还概述了一些替代方案。

第二章 Caches

Caches存储最近访问的程序数据和指令，希望在不久的将来需要它们，这是基于程序内存访问显示出显著的空间和时间局部性的观察。通过在一个小而快的结构中缓存（buffering）这种频繁访问的数据，处理器可以给应用程序一种错觉，即对主存的访问是按照几个周期的量级进行的。

数据cache通常按1到3级的层次结构组织，因此我们可以讨论一级cache、二级cache等。级数越小，cache位置越靠近处理器。一级数据cache通常具有较低的相联度，存储几十(KB)千字节的数据，这些数据排列在每个大约64字节的cache块中，并且可以在几个周期内访问它们（通常在1到4之间）。正常地，处理器有两个一级cache，一个用于程序数据（数据cache），另一个用于程序指令（指令cache）。二级和三级cache的大小通常在几百KB到几MB之间，具有非常高的关联性，需要几十个周期才能访问。这些cache通常同时保存程序数据和指令。此外，尽管多核处理器中的每个核心都有自己的（专用）一级cache，但更高级别的内存层次结构通常在多个核心之间共享。在本节中，将重点讨论第一级数据cache。

正常地，程序数据和指令的地址是虚拟地址。加载和存储指令以及取指引擎必须执行地址转换，以将虚拟地址转换为物理地址。cache可以用虚拟地址或物理地址索引。在前一种情况下，cache访问可以提前启动，因为地址转换可以与cache访问并行执行。为了处理潜在的别名（aliasing）问题，标记通常是从物理地址生成的。

加载和存储指令必须采取几个步骤才能访问数据cache。首先，必须在地址生成单元（AGU）中计算内存访问的虚拟地址，这将在第7章中讨论。对于stores，存储队列将使用此地址更新，对于loads，存储队列将进行消歧检查（第6章）。当消歧逻辑决定加载可以继续时（或者当store达到提交阶段时），数据cache被访问。

指令cache访问更简单，因为不允许存储到指令cache，所以不需要消除歧义。唯一的要求是计算要获取的指令的虚拟地址。这个过程通常包括预测下一个指令地址，这一过程将在第3章中描述。

本章的其余部分组织如下。首先，解释如何将虚拟地址转换为物理地址。然后，讨论cache的结构，并描述了几种cache设计方案。讨论重点将放在数据cache上，因为与指令cache相比，它有一个更复杂的流水线。最后，简要回顾了几种可选的cache设计，讨论了它们对指令cache的影响。

2.1 地址转换

物理地址空间定义为处理器可以在其总线上生成的地址范围。虚拟地址空间是应用程序可以使用的地址范围。虚拟化程序地址有两个主要目的。首先，它允许每个程序在安装了不同数量物理内存的机器上或在多个应用程序之间共享物理内存的多任务系统上不加修改地运行。其次，通过隔离不同程序的虚拟地址空间，可以在多任务系统上保护应用程序。

线性地址空间的虚拟化是通过处理器的分页机制来处理的。使用分页时，地址空间被划分为多个页面（通常大小为4–8KB）。一个页面既可以驻留在主存中，也可以驻留在磁盘中（交换出的页面）。操作系统通过称为页表的结构维护虚拟页到物理页的映射。页表通常存储在主存中。

两个虚拟页面可以映射到同一个物理页面。例如，在共享内存多线程应用程序中，这是典型的。出现这种情况时，两个虚拟页面是同一物理实体的“别名”，因此称为虚拟别名（*virtual aliasing*）。在实现数据cache、load/store缓冲区和所有相关管理机制时，能够正确处理虚拟别名非常重要。

当程序发射load或store指令或指令获取时，必须参考页面映射，将线性地址转换为物理地址，然后才能执行内存访问。线性地址分为两部分：页内偏移和页号。页内偏移对应于地址的*N*个最低有效位，其中*N=log2（PAGE_SIZE）*，并标识页面内的字节地址。剩余地址位的页号标识地址空间内的页面位置。

在一个简单的实现中，load指令（或指令获取）必须执行多个内存访问，才能将线性地址转换为物理地址（页表位于主存中）。由于这是一个关键操作，所有现代处理器都实现了一个页表cache，称为转换查找缓冲区（TLB）。TLB是一种小型硬件cache结构，只缓存页表条目。

在一些处理器中，例如Alpha系列处理器，TLB完全由软件控制。这意味着操作系统可以完全自由地在主存中组织页面映射，还可以完全控制要缓存到TLB的页面映射（有专门的说明可以从TLB中添加\删除条目）。

在x86体系结构中，TLB由硬件控制，对操作系统几乎是透明的。在这种体系结构中，页面映射具有硬件能够理解的特定格式。操作系统负责在内存中硬件能够找到的某个位置以正确的格式创建页面映射，以便硬件能够对其进行解析。

TLB通常包含几十到几百个条目。关联性可能会有所不同，但因为它对性能至关重要，所以它的访问时间通常是一个周期。TLB总是根据虚拟地址的页号索引，并返回相应的物理页号和该页面的一些信息。这些信息包括页面的访问权限（是否可以读取或写入，或者是否可执行），以及是否映射到主存页面，或者是否在永久（非易失性）存储器中有备份。

2.2 CACHE结构组织

Cache由两个主要块组成：标记（tag）数组和数据数组。数据数组存储应用程序数据或指令，而处理器使用标记数组将应用程序地址匹配到数据数组条目中。在图2.1中，可以图形化地看到cache逻辑组织。

数据数组在逻辑上被组织为一组集合（sets）。每个集合是一系列块。集合中的块数称为cache的相联度。我们还说，关联度为N的cache是N路相联cache。第i个cache路定义为cache中所有集合的第i个块的集合。关联度为1的情况称为直接映射cache。

内存地址分为三部分。地址的*K*个最低有效位用于标识我们要访问的cache块中的哪些字节。地址的这一部分称为块内偏移。假设块大小是*Q*字节，那么*K=log2(Q)*。地址的下一部分称为索引（index）。正如其名称所示，索引用于标识集合在数据数组中的位置。对于S集的数据cache，需要*M=log2(S)*个索引位。

不同的地址可以映射到数据cache中的同一个集合（它们具有相同的索引），因此需要一种机制将索引反向映射到地址。标记数组就是为了这个目的。标记数组与数据数组具有相同的逻辑组织（相同的集合数和相联度）。对于数据数组中的每个块，标记数组保存一些元数据：剩余的地址位和块的状态（有效性等）

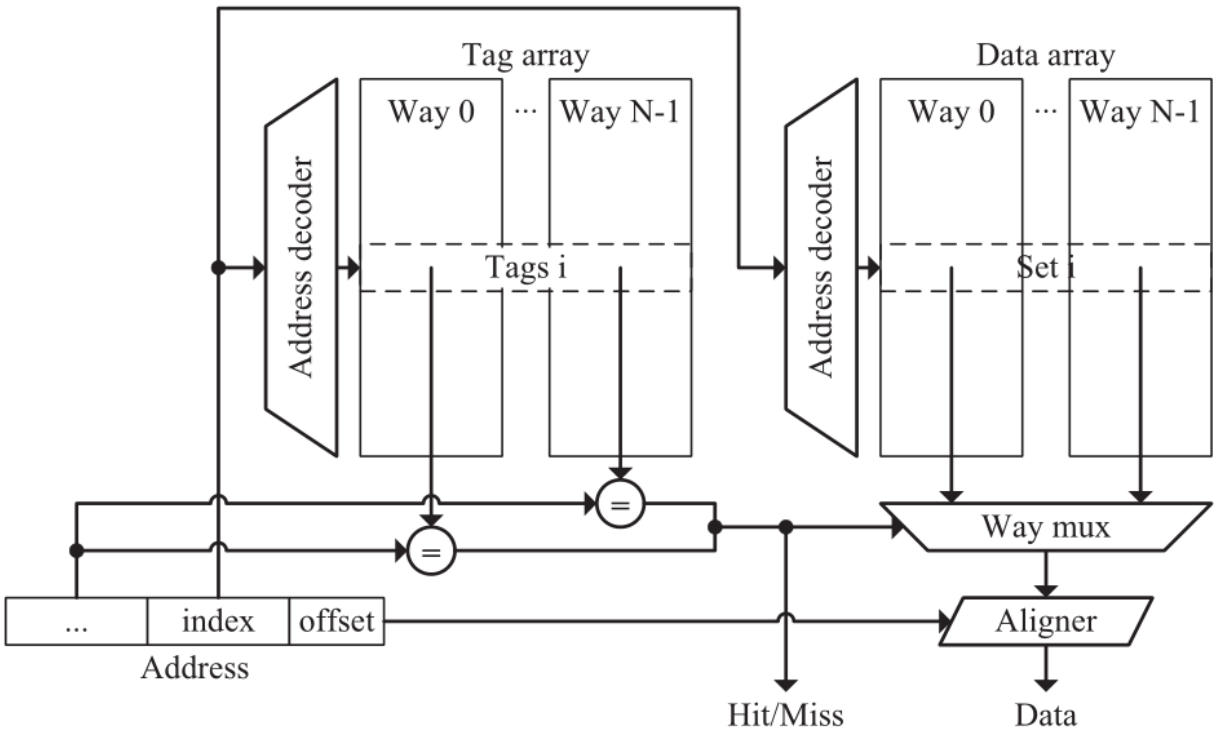

FIGURE 2.1: High-level logical cache organization.

内存请求使用地址的索引部分访问数据和标记数组，但为了知道访问的块是否对应于给定地址，它必须将其余地址位与标记位匹配。如果访问集的第i个块的标记位匹配，则正确的数据位于相应数据数组集的第i个块中（这称为cache命中）。如果集合中没有标记与传入地址匹配，则请求的数据不在cache中（这是cache失效），必须向更高级别的内存层次结构发出请求，并等待数据上提到cache中，然后才能继续访问。

第三章 指令获取单元

指令获取单元负责向处理器提供要执行的指令，因此，它是处理指令的第一个块。取指单元主要由指令cache和计算取指地址所需的逻辑组成。

高性能处理器可以支持每周期一次取指操作，这意味着每个周期必须计算一个新的取指地址。这意味着下一次取指地址计算必须与cache访问并行进行。然而，分支指令（包括条件转移、跳转、子例程调用和子例程返回）带来了额外的复杂性，因为在执行分支之前无法计算正确的取指地址。

出于这个原因，高性能处理器可以预测下一个取指地址。这一预测分为两部分。第一种部分是预测分支的方向，即是否转移。该预测通常由所谓的转移预测器单元执行。第二部分是预测分支的目标地址。该预测由通常称为分支目标缓冲区（BTB）的单元执行。一些处理器将子例程的返回视为特殊情况，并使用所谓的返回地址堆栈（RAS）单元来预测它们。

在图3.1中，我们可以看到指令取指单元的高级框图。有许多可选的取指单元组织（具有多周期预测器、多个预测级别等），但我们认为，图3.1中所示的是一个简单的设计，展示了高性能和高频取指单元的最重要原理，能够在每个周期开始新的取指。

如第2章所述，图3.1所示的指令cache并行访问数据和标记数组。还可以看到指令TLB与指令cache并行访问。在这种设计中，假设cache阵列是用虚拟地址索引的，但标记匹配是用物理地址完成的。并行

访问标记、数据和TLB可以减少总的取指流水线深度，这对性能很重要（它可以降低转移预测失效后重新启动取指的成本）。

图3.1还显示了在第一阶段并行访问的所有相关next-fetch-address预测器。我们表明，选择用于取指地址的预测器输出需要再花一个周期，以强调这个流水线是为高频设计的。尽管这种特定的设计需要四个周期来获取一个指令块，因为它是完全流水化的，所以它可以在每个周期启动一个新的取指，从而实现高吞吐量。

在下面的章节中，将更详细地描述现代微处理器中使用的指令cache和预测器的不同替代设计。

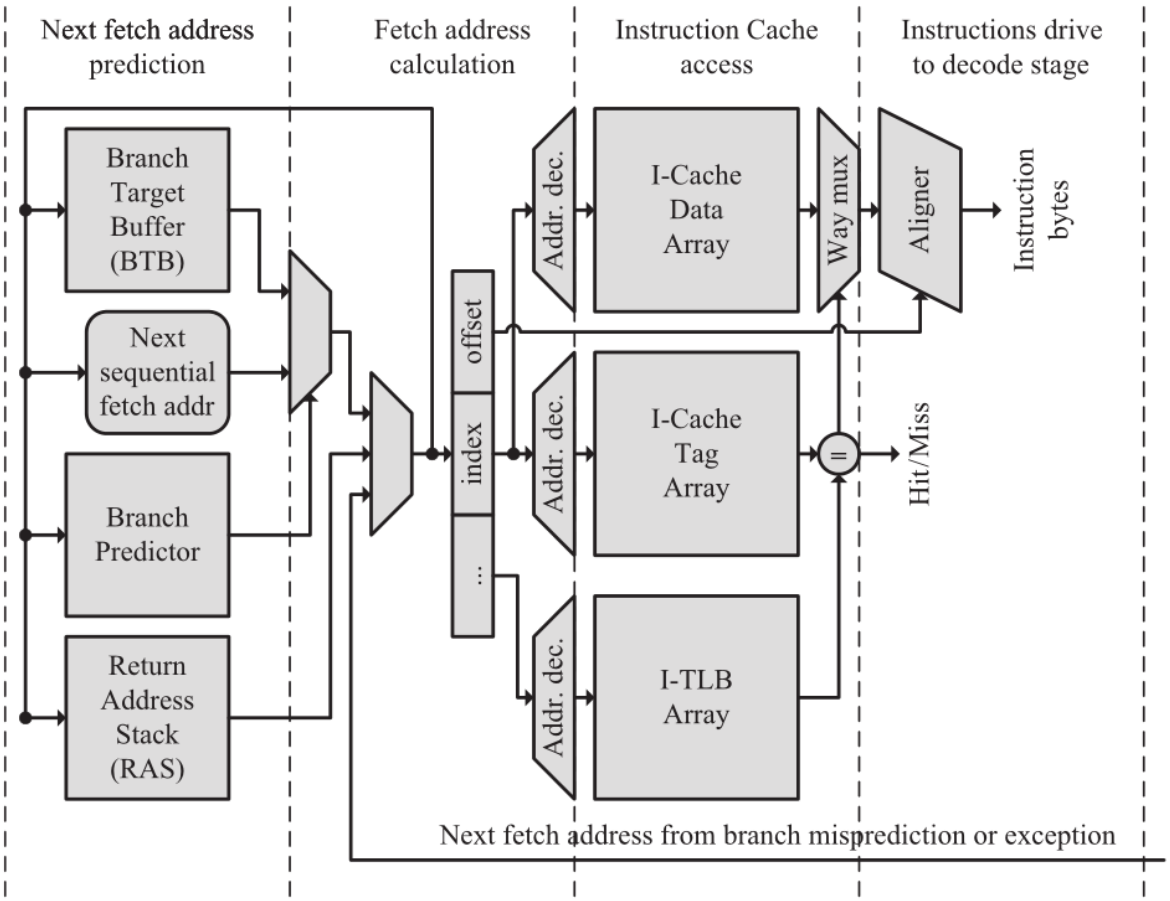

FIGURE 3.1: Example fetch pipeline.

3.1 指令Cache

指令cache存储近期可能需要的一些指令。它通常是组相联的，存储几十(KB)千字节的数据，这些数据排列在每个大约64字节的cache行中。cache可以用虚拟地址或物理地址索引。在前一种情况下，由于地址转换可以与cache访问并行执行，因此可以更早地启动访问。为了处理潜在的别名问题，tag标记通常是从物理地址生成的。

在超标量处理器中，每个周期必须获取多条指令。这通常是通过从cache中读取属于同一cache行的连续字节来实现的。通过这种方式，单个内存端口可以提供所需的带宽。一旦读取了字节串，就必须将其划分为指令。如果指令的大小是固定的，这不费吹灰之力，如果指令的大小是可变的，则需要一些译码。

3.1.1 Trace Cache

传统的cache存储指令的顺序同指令在二进制文件中出现的顺序（静态顺序）相同。然而，有一种替代组织以动态顺序存储指令，称为trace cache。图3.2说明了两个组织之间的关键区别。这两个组织之间有两个关键区别：数据复制和每个内存端口的有效带宽。在传统cache中，二进制的任何指令最多出现一次，而在trace cache中，它可能出现多次，因为它可能是多个不同trace的一部分。另一方面，传统cache的每个内存端口的最大带宽受到分支频率的限制，在某些整数程序中，分支频率可能相当高。

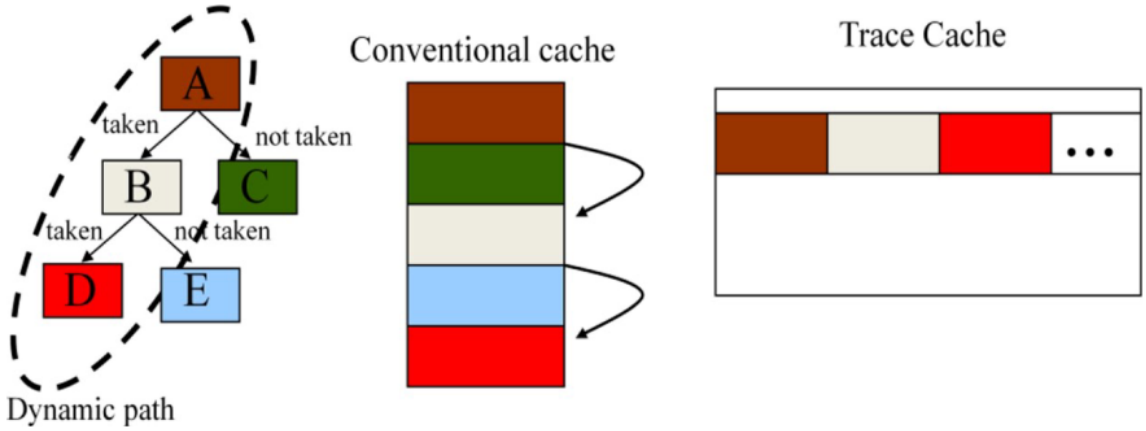

FIGURE 3.2: Conventional instruction cache and trace cache overview.

3.2 分支目标缓冲区

可以使用一个硬件表来预测指令是否是分支，该硬件表以当前取指地址为索引，并且对于每个表项，其元素与取指块中的指令一样多。每个元素一位就足够了。该位指示相应的指令是否应为分支。通过添加更多位，还可以预测分支的类型（条件、调用、返回等）。一旦取指块可用，就会检查预测，如果预测失效，就会更新表。

为了预测分支的结果，分支单元必须预测其目标地址，如果是有条件的分支，还需预测是否转移。

大多数分支指令通过偏移量对其目标地址相对于其位置（又称程序计数器相对，简称PC相对）进行编码。这通常适用于与程序中的条件结构和循环结构相对应的分支，它们负责大多数分支。这些分支有两个重要特征：第一，它们的目标地址总是相同的，并且距离分支指令不远。因此，PC相对编码是非常合适的。对于这些分支，可以在取指它们后的周期中计算目标地址，而不是预测它们。这只需要在前端有一个专用的加法器，但主要缺点是，它会为每个转移的分支引入一个1周期的气泡，这对于高性能处理器可能是一个重要的惩罚。

为了避免这样的气泡，必须预测目标地址。这通常是通过一个硬件表来实现的，该硬件表使用取指地址进行索引，并且取指块中的每条指令都有一个条目。如果该条目对应于分支，则该条目包含预测的目标地址。预测只是上一次执行分支时的目标地址。这保证对于PC相对分支是正确的，对于计算分支（即，目标地址在编译时未知，在运行时计算的分支）也是非常准确的。请注意，此表可以与上面描述的表组合，用于预测指令是否为分支，组合表通常称为分支目标缓冲区（BTB）。图3.3显示了其主要结构。

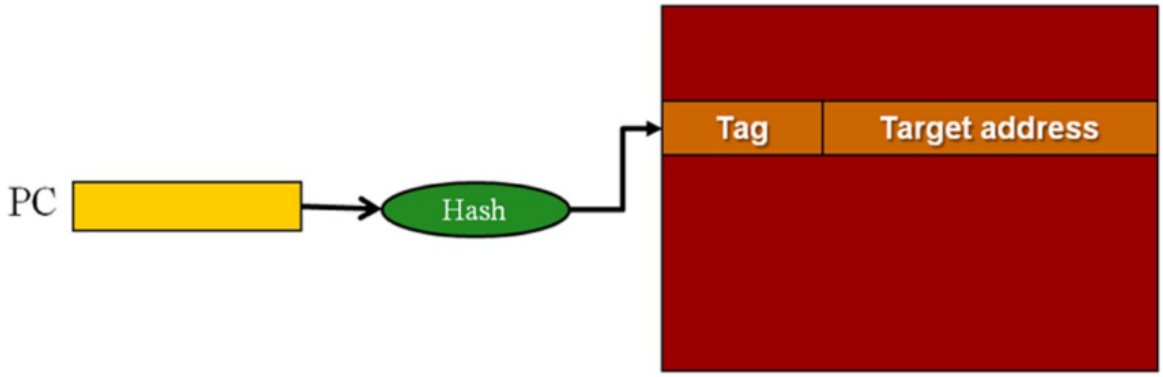

FIGURE 3.3: The branch target buffer.

3.3 返回地址栈

有一种特殊类型的计算分支值得特别注意。这些是从子程序返回的指令。这些指令具有可变的目标地址，这取决于对子例程进行相应调用的位置。BTB可以相对准确地预测它们，因为给定的子例程通常在同一位置的一行中被多次调用（例如，当调用在循环中时）。然而，有一种特殊的机制非常简单，甚至更精确。它被称为返回地址栈（RAS）。

RAS是一种硬件后进先出结构，处理器每次取指子例程调用时，下一条指令的地址都会被推入。当一条返回指令被提取（或预测将由BTB提取）时，RAS的最近条目会弹出，并用作返回指令的目标地址预测。如果RAS有无限数量的条目，它将能够正确预测几乎所有的返回指令（如果返回地址在子例程中没有显式更改，完全没受到良好编程实践的影响，在典型工作负载中非常罕见）。实际上，RAS的条目数量相对较少（例如几十条）。当call指令发现它已满时，最老的条目将丢失，这将导致未来store中的预测失效。然而，这种情况在许多程序中非常罕见，因为它只发生在子程序嵌套级别高于RAS条目数时，并且实验观察到嵌套级别很少高于几十个。显然，递归子程序并非如此。

3.4 条件转移预测

关于是否执行分支，对于条件转移，预测在任何处理器中都是必须的，因为条件取决于运行时数据，在流水线的执行阶段之前无法计算。从获取分支到计算分支，在许多微处理器中可能需要10个以上的周期，因此，在几乎任何处理器中，等待其计算来取指下一个块都不是一个选项。

转移条件预测可以静态地（即由编译器/程序员）进行，也可以动态地或两者结合进行。

3.4.1 静态预测

静态预测可以通过分析信息来完成，方法是收集特定运行的每个分支的最频繁结果，并将其用作预测。在没有分析信息的情况下，可以相对准确地预测循环闭合分支将被执行转移。对于其余的分支（例如，与条件结构相对应的分支），通常很难先验地知道它们将走向哪个方向。

从硬件的角度来看，条件分支的静态预测非常简单；它只需要指令中的一些位（一位可能就足够了），这样编译器就可以对预测进行编码。静态预测的最简形式是预测所有分支是相同结果（即全部转移或未转移），这避免了指令中需要任何额外的位。另一方面，动态预测需要更复杂的硬件，但一般来说，它要有效得多，因此几乎所有的处理器中都存在动态预测。动态预测器的优势在于，它们使用正在运行的应用程序的实际数据，并且可以更改同一静态分支的每个动态实例的预测。

3.4.2 动态预测

动态预测基于一些硬件，这些硬件存储运行中应用程序的过去信息，并使用这些信息预测每个分支。过去，一个简单且非常常用的预测器由一个表组成，该表包含2n个条目，每个条目2bits（见图3.4）。该表以分支指令的地址（PC）为索引，例如，使用n个最低有效位（如果指令大小固定，通常不考虑代表字节偏移量的少数最低有效位）。相应的条目用于预测分支条件，并在分支条件结果可用时用其更新，以反映该分支的最近历史。2-bit条目实现了一个有限状态机，用于进行预测和存储最近的历史。图3.4所示的有限状态机通常被称为2-bit饱和计数器。

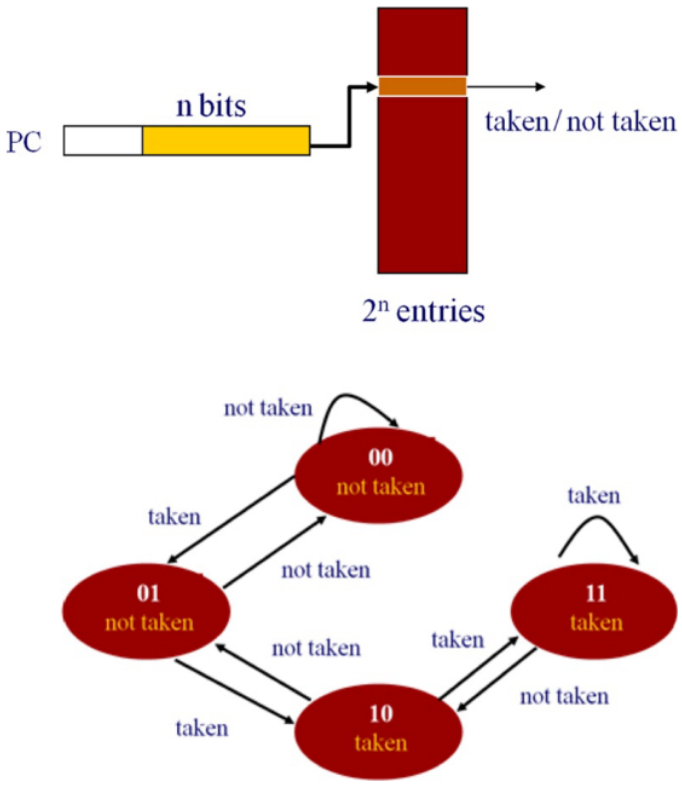

FIGURE 3.4: A local branch predictor.

这个预测器被称为局部转移预测器，因为如果忽略混叠的影响，每个分支的预测都是使用同一分支的历史进行的。该预测器设计用于处理高度偏差的分支。几乎总是转移（taken）的分支将倾向于处于“11”状态，而几乎总是不转移的分支将倾向于处于“00”状态。如果一个分支不时地改变其偏差，只要它在一段时间内保持一个特定的偏差，这个预测器也很好。

由于表的条目数量有限，有时两个不同的分支会使用同一个条目。这被称为混叠，通常会降低预测器的准确性。特别地，对于该预测器，如果两个混叠分支具有相同的偏差，则该混叠的影响最小，但如果它们具有相反的偏差，则其预测的准确性会显著降低。

2位局部转移预测器的准确率通常高于80%，对于某些程序，其准确率可高达99%。对于某些微处理器来说，这可能足够了，但对于当前的高性能处理器来说，10%的预测失效是一个重要的惩罚。这主要是因为每一次预测失效都会受到很高的惩罚。如第8章所述，转移预测失效会导致流水线刷新，取指单元被重定向到正确的路径。这意味着，来自正确路径的指令将无法到达执行阶段，直到它们遍历从取指到执行的整个流水线。在当前的高性能处理器中，这通常超过10个周期，因此对于分支非常频繁的程序（例如，每10条指令中就有1条分支指令），每次预测失效都会引入超过10个周期的气泡，这是一种严重的惩罚。前端气泡的惩罚取决于特定的微体系结构（前端的带宽、后端的带宽等）和代码的特

性，因此在没有周期级模拟器（或实际硬件）的情况下无法计算惩罚。然而，请注意，当前的超标量处理器的前端带宽大约为每周期4条指令，因此前端10个周期的气泡意味着失去了获取40条指令的机会。

为了进一步减少转移预测失效的惩罚，当前的微处理器通常包括一个相关预测器，也称为两级转移预测器。相关预测器不仅使用分支本身的历史，而且还使用其他“邻居”分支的历史，对给定分支进行预测。

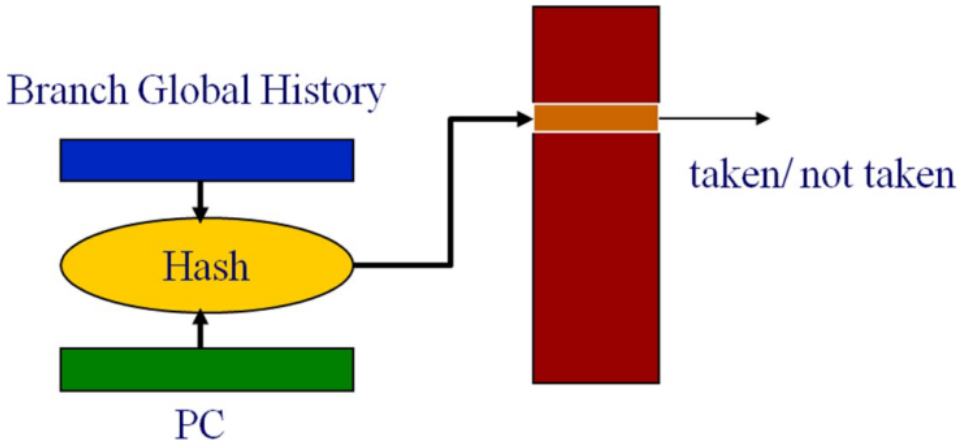

FIGURE 3.5: A gshare predictor.

图3.5显示了一种简单有效的方法来构建相关预测器。有一个名为“分支全局历史”的寄存器，用于存储最近分支的结果（1bit表示转移或未转移）。大约10到20bits历史就足够了。该历史通过哈希函数与分支的PC相结合，生成包含2位饱和计数器的表的索引。该条目用于进行预测，并以与上述局部预测器相同的方式用分支的结果更新。这个预测器被称为gshare。

gshare和所有相关预测器背后的基本思想是，尝试对静态分支和历史的每个不同组合使用不同的有限状态机。这样，预测就是基于这个特定分支过去做了什么，以及它的“邻居”分支做了什么。当然，从成本的角度来看，不为PC和历史的每一个潜在组合提供一个条目，并容忍某种程度的混淆，更具成本效益。实验证明，分支全局历史和PC的最低有效位之间的位异或是一个简单有效的哈希函数，可以最小化混叠。

相关预测器在全局历史的使用方式以及如何与预测分支的身份（即PC）相结合方面有所不同。例如，我们可能有多个全局分支历史寄存器，如图3.6所示。一个特定的分支基于其PC使用一个具体的历史寄存器，该具体寄存器与PC进行哈希运算，以获得将用于预测的特定有限状态机的索引。

FIGURE 3.6: A correlating predictor with multiple global branch history registers.

相关预测器的准确性取决于使用的全局历史记录的数量、有限状态机（即两位饱和计数器）的数量，以及如何使用全局历史和分支PC来确定每种情况下使用的具体有限状态机。没有任何特定的配置适合所有代码。例如，有时最好有更多的全局历史，而在其他情况下，全局历史没有帮助，甚至可能导致准确性下降。因此，一些处理器使用混合转移预测器。

混合转移预测器由上述描述的多个预测器和一个选择器组成（见图3.7）。选择器负责决定每个转移预测中哪个预测器更可靠。当混合预测器由两个单独的预测器组成时，选择器类似于另一个转移预测器，因为它必须预测布尔值。它可以通过一个由两位饱和计数器组成的表来实现，该表由PC和一些全局历史（或两者中的一个）的组合索引。计数器的最高位表示首选的预测器。一旦知道分支的实际结果，条目将按以下方式更新。如果两个预测值都是正确的，或者都是错误的，则不修改计数器。如果第一个预测器正确，第二个预测器错误，计数器将增加，而如果第一个预测器错误，第二个预测器正确，计数器将减少。

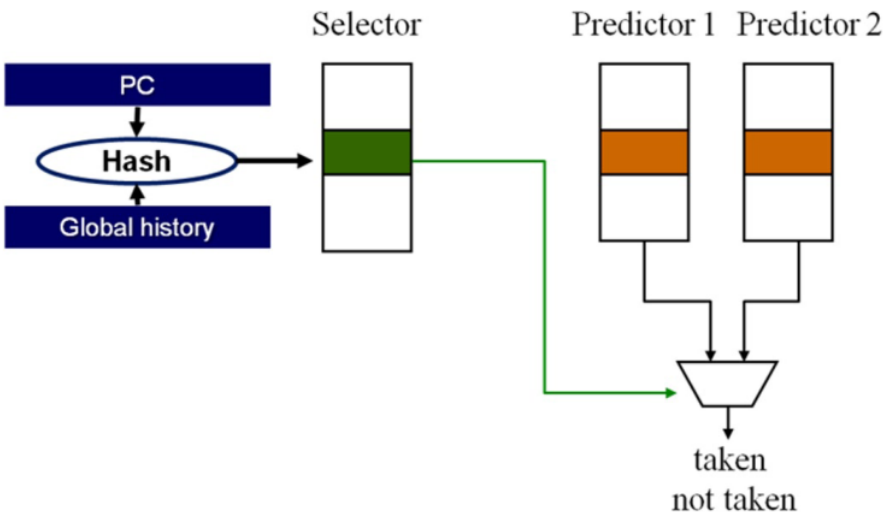

FIGURE 3.7: Hybrid branch predictors.

混合预测器很有趣，不仅因为不同类型的代码可能更适合不同的预测器，还因为预测器的预热时间不同。我们将预测器的预热时间称为从其不同表和寄存器包含不相关信息（例如，来自另一个应用程序）的初始状态到预测精度稳定的状态所需的时间。局部预测器的预热时间很短；只要一个偏差分支被执行几次，它就会被正确预测。另一方面，相关预测器的预热时间要长得多，因为给定的静态分支使用多个有限状态机，每个有限状态机对应一个不同的全局历史。为了达到其稳定状态，所有这些有限状态机都必须偏向其适当的值，这就要求该分支按照全局历史的不同值，成比例执行多次。考虑到每次有上下文切换时，预测器的状态与新过程无关，在多道程序系统中，预热时间可能会产生重要影响。因此，可以考虑使用由局部预测器和相关预测器组成的混合预测器。在每次上下文切换之后，局部预测器通常会更好，因为它的预热速度更快，但在一段时间后，相关预测器将开始比局部预测器更好地预测。

第四章 译码

指令译码阶段的目的是理解指令的语义，并定义处理器应如何执行该指令。在这个阶段，处理器识别：

- 这是什么类型的指令：控制、内存、算术等。
- 指令应该执行什么操作，例如，它是否是算术操作，应该执行什么ALU操作，它是否为条件转移，应该评估什么条件等。
- 这条指令需要什么资源，例如，对于算术指令，哪些寄存器将被读取，哪些寄存器将被写入。

通常，译码阶段的输入是包含要译码的指令的原始字节流。然后，译码单元必须首先通过识别指令边界将字节流拆分为有效指令，然后为每条有效指令的流水线生成一系列控制信号。译码单元的复杂性在很大程度上取决于ISA和我们想要并行译码的指令数。

在本章的第一节中，将简要解释在RISC机器中译码是如何工作的。在下一节中，将重新讨论x86指令的编码，并对每个ISA功能如何影响译码复杂性进行评论。接下来，我们将讨论现代x86处理器使用的动态翻译技术，将x86指令翻译为简单的RISC类操作。最后，我们将描述一个现代乱序x86处理器的译码流水线，该处理器将x86指令动态翻译为一组类似RISC的内部指令。

4.1 RISC译码

典型的RISC译码流水线如图4.1所示。在图中，我们展示了一个超标量RISC机器，它可以并行译码4条指令。在本节其余部分的讨论中，我们将假设都是图4.1的体系结构。

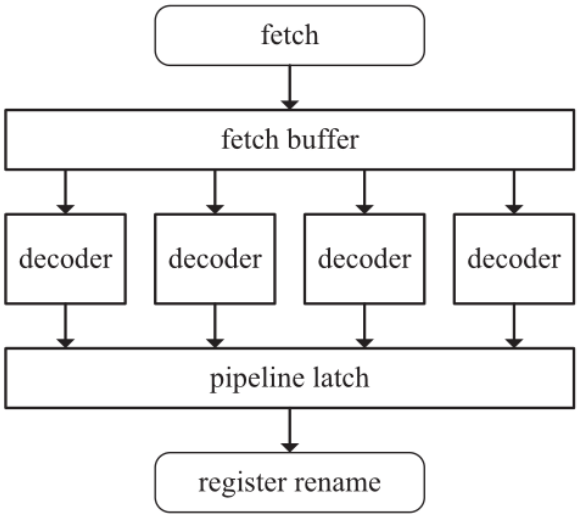

FIGURE 4.1: Typical RISC decode pipeline.

通常，RISC指令很容易译码。大多数RISC处理器都有固定的指令长度，这使得在取指缓冲区中查找指令的边界并将指令的原始位传递给译码器变得很简单。我们唯一需要的是第一条指令的取指缓冲区内的索引（它可能不与缓冲区的开头对齐），这很容易从PC的低阶位获得。

此外，RISC ISAs的编码格式非常少，这意味着指令中的操作码和操作数的位置几乎没有变化。这一点，再加上RISC指令是“简单”的，简单意味着它们为流水线生成的控制信号很少，使得译码器相对简单。

RISC指令的简单性使高性能处理器实现具有单周期译码，使用简单的PLA电路和/或小型查找表。当然，这是RISC最初的目标之一：允许轻松译码和简单的执行控制，促进高性能实现。

4.2 X86 ISA

x86是一个可变长度的CISC指令系统。x86指令的格式如图4.2所示，借用自《英特尔体系结构手册》。x86指令包含最多四个前缀字节（可选）、一个1到3字节的强制操作码，以及一个可选的寻址说明符，该说明符由ModR/M字节和或许SIB（scale-index-base）字节组成。有些指令可能还需要移位（最多4字节）或立即数字段（也 最多4字节）。

指令前缀有多种用途。例如，前缀可以修改指令操作数：段覆盖前缀更改地址的段寄存器，而操作数大小覆盖前缀在16位和32位寄存器标识符之间更改。

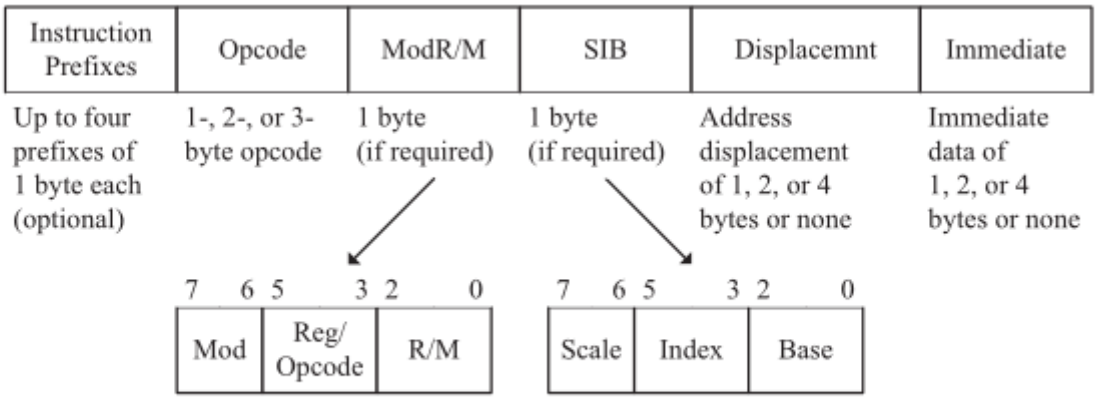

FIGURE 4.2: The x86 instruction format.

如果想并行译码多条指令，必须知道每条指令的起始位置。由于指令长度可变，这项任务具有顺序性：我们必须先知道指令i的长度，然后才能知道指令i+1在当前取指块中的起始位置。因此，能够快速计算指令长度对性能至关重要。

x86译码器面临的第一个复杂问题是识别指令长度。只有在译码操作码后，如果存在ModR/M字节，这才可能实现。操作码定义是否存在ModR/M字节，如果存在，则ModR/M定义是否存在SIB字节。移位或立即数的存在也在操作码中定义。

在译码x86指令的操作码时，有两个问题。首先，操作码与指令开头的偏移量并不总是相同的。它可以从指令的前5个字节的任何地方开始，因为它前面最多有4个字节的前缀。第二个问题是，操作码本身的大小可变，主操作码的大小可达3字节，有时，ModR/M的3到5位被用作操作码扩展。

x86译码器面临的第二个复杂问题是识别指令的操作数。例如，在寄存器对寄存器操作的简单情况下，操作数可以在操作码或ModR/M字节编码。ModR/M字节反过来可以对2或1寄存器操作数进行编码，具体取决于操作码和ModR/M的6到7位。

在寄存器到寄存器的示例中，3位操作数定义了一个通用寄存器，但要知道是哪一个，我们需要来自操作码、当前执行模式的信息，在某些情况下还需要来自前缀（如果有操作数大小覆盖前缀）的信息。这是因为3bit，只能对8个通用寄存器进行编码，但x86中有更多的体系结构寄存器。因此，在32位模式下，值为0的操作数可以解释为AL、AX、EAX、MM0或XMM0。

从上面的讨论中可以明显看出，x86译码绝非小事（trivial）。在现代x86微处理器中，译码需要几个周期，这是显著设计复杂性的一个来源。在以下几节中，将讨论高性能、乱序的超标量微处理器可能的译码单元实现。

4.3 动态翻译

x86指令包含大量语义信息，可能需要执行核心执行多个操作。例如，“add [eax], ebx”x86指令编码了一个加法操作，用EAX指定的地址处的内存值和寄存器EBX相加，并将结果写回地址EAX处的内存。这需要处理器：

1. 使用EAX和数据段寄存器DS计算内存操作数的地址。
2. 将内存位置的值放入内核（core），并将寄存器EBX的值与其相加。
3. 将加法结果存储到步骤1中计算的内存位置。

试图执行该指令的乱序执行引擎需要大量的控制状态和信号来跟踪指令在每个时间点的执行阶段（这将在第6章中更详细）。微处理器必须保证该指令的执行阶段的顺序正确，并保证与其他指令的依赖性。如果我们想要高性能，还需要将指令执行的某些部分与其他指令“并行化”。例如，地址计算不应依赖于前一条为寄存器EBX生成正确值的指令，而在加法阶段应该依赖。因此，显然，在乱序执行内核上高效地执行如此复杂的指令并不是一件容易的任务。

另一方面，RISC ISA的编译器会将这个复杂的操作分解为三条简单的指令。以下代码序列对应于这样的代码序列（对于虚构的RISC ISA）。在这里，使用x86寄存器名以便于比较这两种情况。此外，假设指令的目标操作数是最左边的：

```
1 load r0, ds:[eax]
2 add r1, r0, ebx
3 store ds:[eax], r1
```

这些指令可以由乱序执行引擎轻松处理。每个指令将执行一个且仅一个操作，每个指令的依赖关系都很明确，执行引擎可以自由地将这些操作与其他非依赖操作混合，等等。

由于x86 CISC指令在早期给执行引擎带来的复杂性，x86处理器工程师决定在处理器的译码单元中将x86指令流动态翻译为类似RISC的指令。遵循这种设计的第一批实现是AMD K5和Intel P6。这样，就保持了与x86 ISA的二进制兼容性，同时大大简化了类似于RISC机器的执行引擎。

如今，所有现代乱序的x86微处理器都会将x86指令动态翻译为类似RISC的内部指令格式。特别是，英特尔称这些内部指令为微操作，简称μops。P6 μops具有固定长度（118位）和常规格式，编码一个操作（即操作码）和三个操作数（两个源和一个目的）。P6 μops使用load/store模式。在上面的x86-vs.-RISC示例中，P6译码器将生成一个μops序列，非常类似于RISC的情况。

从P6 μops的大小，我们可以得出，实际上，μops与其说是对应于RISC指令，不如说是对应于译码的RISC指令，也就是说，对应于简单的类似RISC操作的流水线控制信号。当然，从本质上讲，现代微处理器的μop与P6的μop不同，但我们相信它们仍然遵循RISC哲学。

4.4 高性能X86译码

图4.3显示了英特尔Nehalem体系结构译码单元的高级框图。可以看出，x86译码是一种多周期操作。在这个特定的实现中，这个过程被分为两个解耦的阶段：指令长度译码器（“预译码”阶段，ILD）和动态翻译到μops阶段（“译码”阶段）。ILD阶段的目的是将原始字节流分离为有效的x86指令序列，

并将这些指令传递到第二个译码阶段。动态翻译阶段接收x86指令流作为输入，并生成功能等效的μops流。这两个阶段通过指令队列（IQ）进行解耦。这样做的原因是一个简单的锁存器是为了隐藏ILD中可能出现的气泡，当复杂的x86编码出现时，也允许ILD在需要从动态翻译单元进行复杂翻译时继续进行。

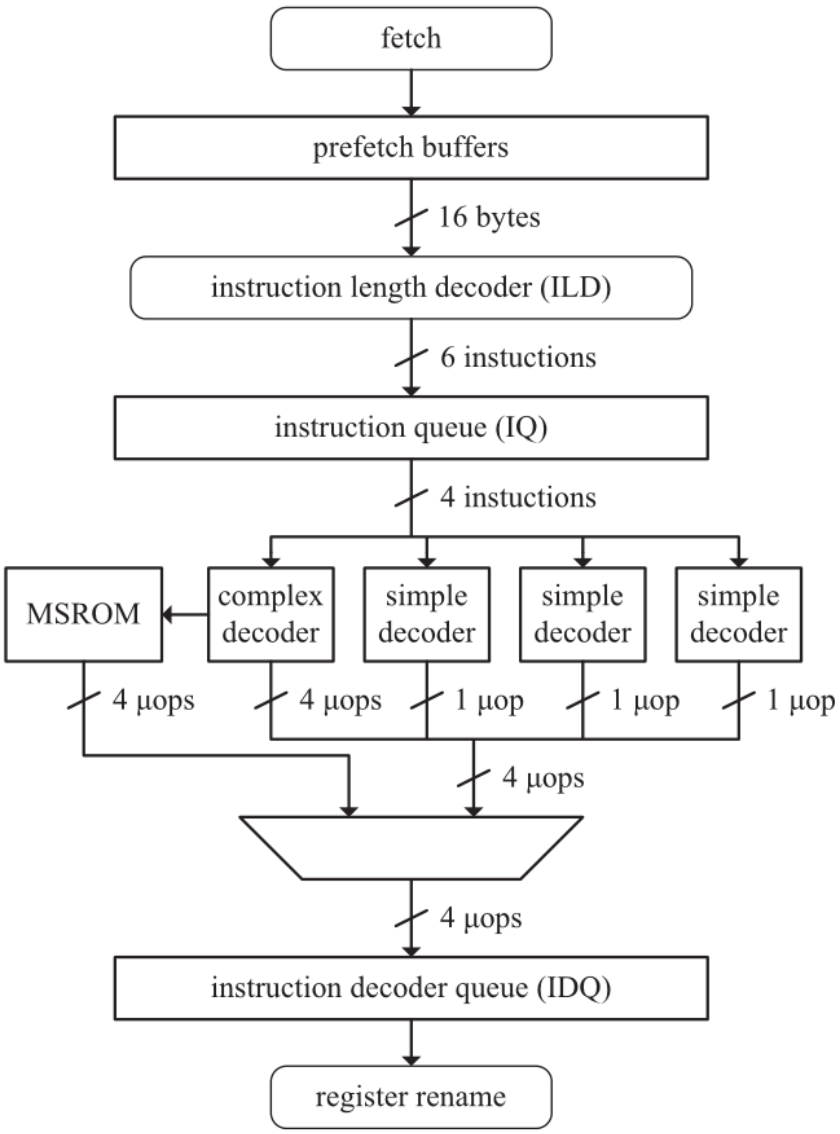

FIGURE 4.3: The Intel Nehalem decoding pipeline.

4.4.1 指令长度译码器

ILD单元从取指缓冲区接收16个对齐字节，并执行一些基本的预译码，以便于动态翻译阶段。ILD决定每条指令的长度，对其所有前缀进行译码，并标记指令的各种属性，这将有助于第二阶段译码。指令长度译码本质上是顺序的，所以如果我们想在高频下预译码许多指令，就必须尽可能快。

ILD可以在一个周期内处理最常见的指令编码。在Intel Core和Core 2微体系结构中（可能在Nehalem中也是如此），以下两种情况无法通过正常路径处理，并且使用了缓慢的六周期路径：

- 一种操作数大小覆盖前缀，在指令前面加一个字立即数。
- 一种地址大小覆盖前缀，位于一条带有ModR/M字节的指令之前。

这两个前缀被称为长度变化前缀（LCP）。要使用LCP确定指令的长度，ILD单元必须对指令进行一些复杂的译码，包括操作数，而不仅仅是正常ILD路径中的操作码。

4.4.2 动态翻译单元

在这个阶段，从指令队列中读取指令，并将其翻译为μops。许多x86指令，尤其是寄存器到寄存器的指令，被翻译为单个μop。一些具有内存操作数或使用复杂寻址模式的x86指令被翻译为多个μops。

图4.3的设计实现了三个“简单”译码器，用于处理翻译为单个μop的指令，而只有一个“复杂”译码器，用于处理最多可翻译为四个μops的指令。这种安排节省了功耗并降低了复杂性，并且假设大部分指令属于“简单”类别，则不会牺牲译码带宽。

有些x86指令，例如字符串指令，需要四个以上的μops。这些指令被发送到复杂译码器，然后复杂译码器停止正常的译码流水线，并将控制权传递给微序列器（MSROM）单元。MSROM包括序列器电路和ROM阵列。微序列器输出一个微码程序来仿真复杂的x86指令。这个程序只不过是一个预编程的正常μops序列（类似于其他译码器输出的那些）。

第五章 分配（Allocation）

在这个流水线阶段执行两个主要活动：寄存器重命名和指令派遣。前者的目的是消除由于寄存器重用而产生的错误依赖，而后者则包括保留执行指令所需的一些资源。

派遣包括保留指令将来将使用的一些资源，这些资源通常包括发射队列、重排序缓冲区和加载/存储队列中的条目。如果所需的任何资源不可用，指令将暂停，直到它们可用为止。有时，这些资源被划分为多个单元，每个单元都与某些特定的资源相关联，因此分配也会产生一个副作用，即确定指令稍后将使用哪些资源来执行。例如，可能存在与每个功能单元相关联的不同发射队列。在这种情况下，分配还确定指令将在哪个功能单元中执行。

寄存器重命名通常只在乱序处理器中进行。乱序处理器执行程序指令的顺序与编译器或程序员生成的程序顺序不同，但语义相同。指令被重新排序以增加被开发的指令级并行量。指令重新排序受指令之间的依赖关系约束。两条指令之间的依赖迫使它们之间有特定的顺序。依赖可以有两种类型（见图5.1）：数据依赖和名称依赖。前者发生在一条指令产生另一条指令使用的数据元素时。显然，在这种情况下，生产者指令必须在消费者面前被执行。

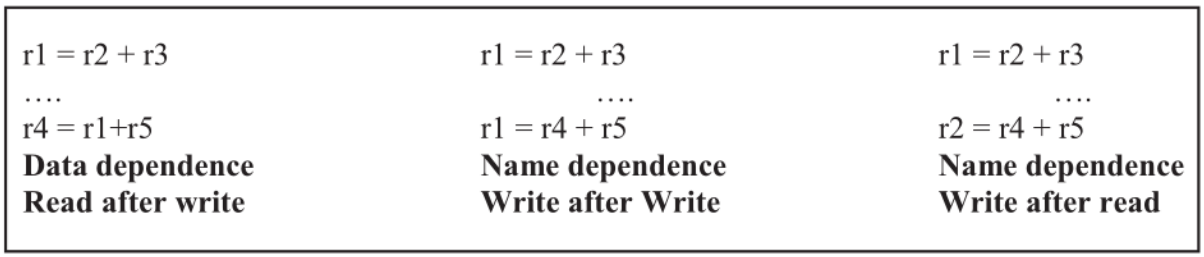

FIGURE 5.1: Instruction dependences.

另一方面，名称依赖是由存储位置的重用引起的，它们不涉及互相之间的任何特定数据传输。名称依赖项可以有两种类型：写后写和读后写。

名字的依赖可能是人为引入的；这不是因为算法，而是因为存储位置被重用。通过将每条指令写入不同的存储位置，我们可以消除所有名称依赖。这对于许多应用程序/系统来说可能是不可行的，因为它需要超出当今系统可提供的存储位置（例如，在单个内核中运行一小时的应用程序可能会执行超过1012条指令）。 然而，即使有足够的内存，这样做也可能不是一个好主意，因为它会因为失去局部性而对性能造成巨大影响。

乱序处理器的目标要小得多。它们动态地消除了所有名称依赖，但仅限于正在执行的指令。由于典型的指令窗口大小约为100条指令，因此为它们提供不同的存储位置是可以承受的。在本章中，我们将重点介绍寄存器操作数。在讨论内存指令发射时，重命名也适用于内存操作数，如第6章所述。

大多数处理器的体系结构寄存器数量非常少（例如32 整数和32 浮点），因此，通过寄存器的名称依赖非常常见，在乱序处理器中消除它们的好处是巨大的。

寄存器重命名最早于上世纪60年代由Tomasulo在其著名的IBM 360/91浮点单元乱序执行方案中提出。在该方案中，使用将产生目的操作数的保留站的标识符重命名目标操作数。当前的微处理器不使用这种方案，因为它要求保留站被指令占用，直到其执行完成。如第6章所述，当前的微处理器在发射后立即释放发射队列条目（Tomasulo术语中的保留站），这在效率方面更有效。

现代微处理器使用三种可选的重命名方案。我们将它们称为通过重排序缓冲区重命名、通过重命名缓冲区重命名和合并寄存器堆重命名。

5.1 通过重排序缓冲区重命名

在该方案中，寄存器值存储在重排序缓冲区和体系结构寄存器堆中。重排序缓冲区（ROB）存储未提交指令的结果，而体系结构寄存器堆存储每个体系结构寄存器的最新提交值。有一个重命名表，指示每个体系结构寄存器的最新定义是在ROB还是体系结构寄存器堆中。为了便于访问ROB中的操作数，重命名表在前一种情况下，还包含一个额外的字段，指示ROB中操作数所在的位置（见图5.2）。

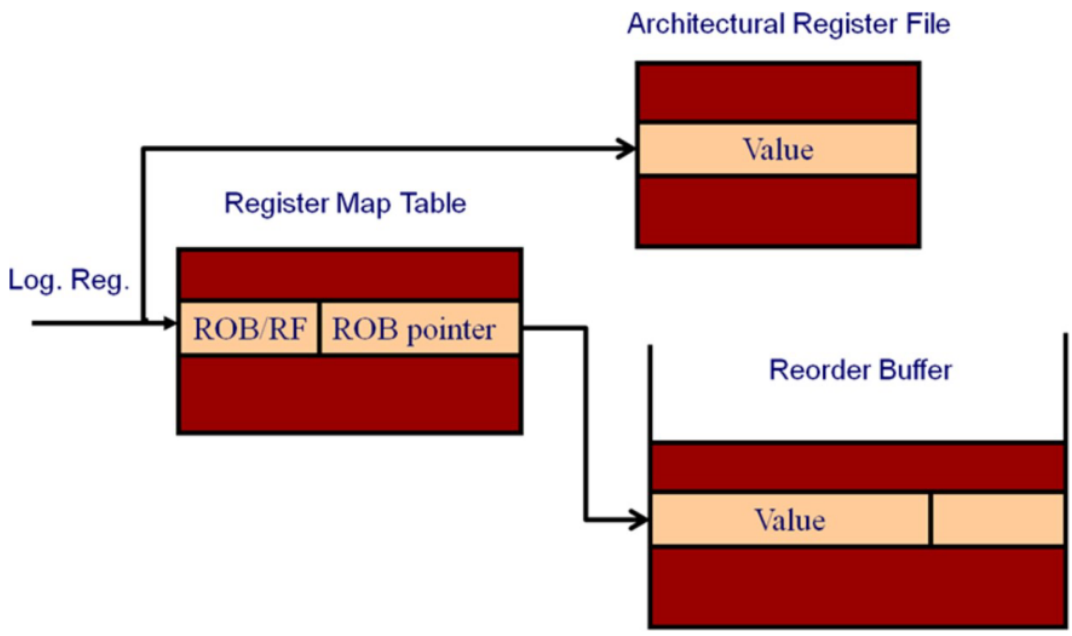

FIGURE 5.2: Renaming through the reorder buffer.

指令执行时，其值存储在ROB中。稍后提交时，该值将从ROB复制到体系结构寄存器堆。请注意，给定的操作数在其生存期内可能位于两个不同的位置。这可能会给读取操作数的方案带来一些额外的复杂性，如下所述。

这是一些微处理器使用的方案，如英特尔酷睿2。

5.2 通过重命名缓冲区重命名

这个方案是前一个方案的一个小变种。其动机是一个重要的事实，即执行的指令很大一部分（大约三分之一，尽管在不同的应用程序和ISAs中差异很大）中没有产生任何寄存器结果。在之前的重命名方案中，ROB中的每个条目都有一个字段来存储寄存器结果，这意味着大约三分之一的存储被浪费了。重命名缓冲区方案的思想是为正在运行的（即未提交的）指令的结果提供一个单独的结构。这样，只有产生结果的指令才会占用存储位置。与重排序缓冲区方案一样，结果首先存储在重命名缓冲区中，并在指令提交时移动到体系结构寄存器堆中。重命名指令时，如果指令需要重命名缓冲区条目，但没有可用的条目，则该指令将暂停，直到条目变为可用（由于正在运行的指令更旧，不会依赖于暂停的条目，因此不会发生死锁，因此最终，它们将提交并释放分配给它们的重命名缓冲区条目）。

IBM Power3处理器和其他处理器使用此方案。

5.3 合并寄存器堆重命名

在这个方案中，有一个寄存器堆存储推测值和提交值。因此，该文件的大小大于体系结构寄存器的数量。每个寄存器要么是空闲的，要么是已分配的。空闲寄存器被记录在空闲列表中。已分配的寄存器正在使用中，可能包含提交值、推测值或根本没有值（在已分配但尚未生成结果的情况下）。此外，还有一个寄存器映射表，用于存储每个体系结构寄存器的最新分配（物理寄存器标识符）（见图5.3）。

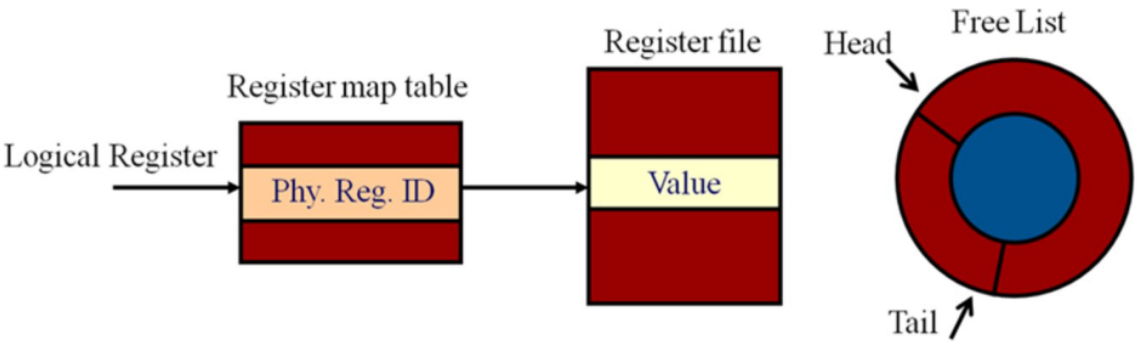

FIGURE 5.3: Merged register file.

例如，可以通过存储所有空闲物理寄存器的标识符的循环缓冲区来实现空闲列表。

重命名指令时，将查找重命名映射表，以找出其源操作数。此外，如果产生寄存器结果，则从空闲列表中分配一个空闲物理寄存器。如果没有可用的空闲寄存器，则指令将暂停，直到较旧的指令提交并释放寄存器（有关何时释放寄存器的讨论，请参阅下文）。目标操作数将重命名为此空闲寄存器，重命名映射表将更新以反映此映射。

当处理器可以保证物理寄存器失效时（即不再有指令使用它们），物理寄存器被标记为空闲。理想情况下，这可以在使用该寄存器的最后一条指令提交时完成。然而，识别寄存器的最后一次使用对处理器来说并不简单（编译器可能知道它，但当前的ISAs通常没有办法将此信息传递给硬件）。因此，处理器采用以下安全、保守的方法： 当接下来使用相同的体系结构目标寄存器的指令提交时，物理寄存器失效。如图5.4所示。在本例中，当指令（2）提交时，寄存器p1被释放。请注意，在（2）提交之前等待指令（2）被取指并等待p1的所有消费者是不够的，因为指令（2）可能会被撤销（例如，由于分支预测失效），而p1的其他消费者可能会在稍后找到。

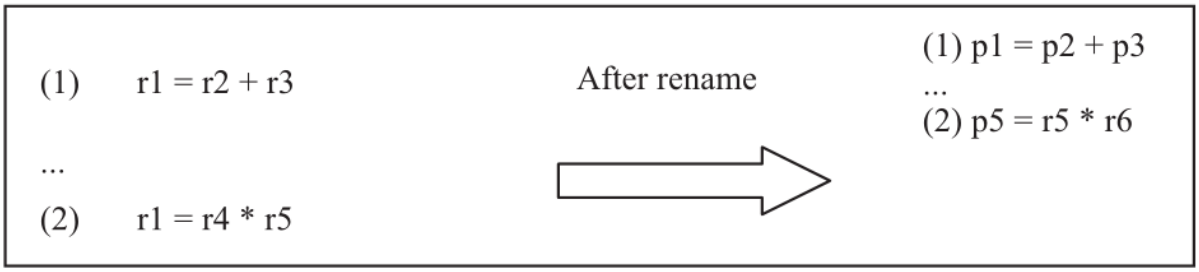

FIGURE 5.4: Releasing a physical register.

这是Alpha 21264、MIPS R12000和奔腾4等处理器使用的方案。

5.4 寄存器堆读取

另一个需要考虑的重要方面是何时读取寄存器值，这对设计的几个关键部分具有重要意义。有两种选择：在发射之前读和在发射之后读。

在前一种情况下，在发射之前读取，在指令被分派到发射队列之前读取寄存器堆，并将值存储在发射队列中。显然，当时并非所有操作数都可用，因此只读取可用的操作数，其余操作数在发射队列中标记为不可用。之后通过旁路网络获得不可用的操作数，并且不再访问寄存器堆。这种方案的一个优点是，寄存器堆可能需要较少的端口数，因为它只提供一部分操作数。另一方面，发射队列需要存储操作数，这在某种程度上类似于寄存器堆，因此在面积方面非常昂贵。此外，一些源操作数需要读两次，写一次：从寄存器堆读取并写入发射队列，从发射队列读取并发送到执行单元。这种活动消耗能源，这是当今处理器中非常宝贵的资产。

在后一种情况下，即发射后读取，发射队列存储寄存器源操作数的标识符，而操作数实际上是在发射要执行的指令后读取的。刚刚产生的，不能写入寄存器堆的操作数，通过旁路网络获得。该方案要求寄存器堆中有更多端口，因为它提供了更多的操作数，但另一方面，源操作数只读取一次，不必复制到任何地方（除了阶段锁存器）。

从理论上讲，这两种方案与正在使用的特定重命名方案是正交的，但有重要的协同作用，使得某些特定的组合非常常见。特别地，对于基于重排序缓冲区和重命名缓冲区的重命名方案，在发射之前读取具有重要优势，如下一节所述，通常是首选。

###5.5 推测失效情况下的恢复

由于多种原因（例如分支预测失效、异常），正在运行的指令有时必须被撤销。如果这些指令经过了分配阶段，那么它们保留的资源就必须被释放。此外，必须撤销这些指令在重命名表中所做的修改，以便它们反映出如果这些指令从未执行过的状态。第8章讨论了如何进行恢复。

5.6 三种方案的比较

关于物理寄存器的分配和释放，基于ROB的方案非常简单。由于物理寄存器是ROB条目分配的一部分，后者基本上是FIFO结构，因此无需保留空闲列表。对于每一条新指令，都会分配FIFO的尾部，当指令提交时，FIFO的头部会被释放。重命名缓冲区方案也是如此，因为重命名缓冲区也作为FIFO结构进行管理。另一方面，合并的寄存器堆具有更复杂的管理方案。它需要一个物理寄存器的空闲列表。当需要新的物理寄存器时，使用列表的开头。另一方面，为了释放物理寄存器，每条指令必须在重排序缓冲区中保留物理寄存器的标识符，该标识符在重命名该指令之前映射到目标寄存器。例如，在图11的示例代码中，指令（2）将在重排序缓冲区中存储p1的标识符，以便在（2）提交时释放该标识符。

另一方面，合并寄存器堆有两个主要优点。首先，寄存器值只写入一次，从不移动，而对于其他两个方案，它们写入两次，首先在重排序缓冲区或重命名缓冲区中，然后在体系结构寄存器堆中。这种额外的活动意味着额外的功耗。其次，在合并寄存器堆中，所有源操作数都来自一个位置，而在其他两种方案中，它们可能来自两个不同的位置（体系结构寄存器堆或重排序缓冲区/重命名缓冲区）。单个位置可以减少所需的互连量，并且在寄存器堆和功能单元之间的互连所花费的面积方面可能是有益的。

最后，合并寄存器堆方案可以与上述两种读取方法（发射前读取和发射后读取）一起使用，在实现方面没有显著差异。另一方面，重排序缓冲区和重命名缓冲区方案更适合于发射前读取，如果要将它们与发射后读取结合使用，则会带来一些挑战。特别是，挑战来自寄存器值最终从一个位置（重排序缓冲区或重命名缓冲区）移动到另一个位置（体系结构寄存器堆）。在发射后读取方案中，发射队列存储源操作数的标识符。如果重命名指令时，源操作数位于重新排序缓冲区或重命名缓冲区中，则发射队列将存储指向该位置的指针。如果生成此源操作数的指令在使用者读取操作数之前提交，则该值将被移动到体系结构寄存器堆，并且存储在发射队列中的指针将不再正确，因为此表项可能由其他指令分配。为了纠正它，有必要在发射队列中对每个提交的指令进行关联搜索，以检查是否有任何条目指向其目的寄存器。如果是这种情况，指针应更改为相应的体系结构寄存器堆条目。所有这些在硬件上都非常复杂。关联搜索类似于后面描述的唤醒逻辑，除此之外，还需要额外的写入端口来存储新指针。因此，通过重排序缓冲区或通过重命名缓冲区使用重命名的处理器通常选择先读后发射方案。

第六章 发射阶段

6.1 介绍

发射是负责向功能单元发射执行指令的流水线阶段。发射方案主要有两种类型：按序发射和乱序发射。前者按程序顺序控制指令，后者则在源操作数可用时立即乱序控制指令。

一般来说，按序方案检查最早的未发射指令，并在其源操作数及其执行所需的资源可用时发射该指令。

然而，大多数最新处理器都实现了乱序方案。有许多不同的方法来实现乱序发射。实际上，最终的实现也非常依赖于其他阶段的设计决策。例如，如果在派遣阶段之前或之后读取指令的源操作数，则不

需要相同的硬件。此外，发射方案之间存在显著差异，这取决于它们是基于保留站、分布式发射队列还是基于统一发射队列。

在本章中，我们将介绍在现有处理器中实现的最常见发射方案。

6.2 按序发射逻辑

按序发射逻辑以获取指令的相同顺序发射执行指令。因此，指令会一直等到之前的所有指令都已发射。然后，只要源操作数可用且执行所需的资源就绪，就立即发射指令。

由于使用记分板的简单性，这种发射逻辑有时在处理器的译码阶段实现。一个典型的记分板包括两个表，一个数据依赖表和一个资源表。根据处理器的实际硬件约束，这些表可能会有所不同。

使用要发射的指令的源寄存器标识符对数据依赖表进行索引。此表中的每个条目都表示寄存器值的状态。该状态的范围从“不可用”，指令还不能发射，到“值可用”，要么在某个旁路级，要么写入寄存器堆。读者将在第7章中找到关于旁路的更多细节。

资源表跟踪功能单元等执行资源的可用性。有些功能单元，比如除法器，不能在每个周期接受一个新的操作请求。在这种情况下，如果处理器在一个周期之前调度了另一条使用除法器的指令，则无法调度使用除法器的指令。因此，发射逻辑使用此表来检查给定的执行资源在当前周期中是否可用。

超长指令字（VLIW）处理器实现了一种简化的按序发射逻辑。这些处理器不执行任何类型的记分板，因为生成代码的软件负责将每条指令安排在距离生产者足够远的地方，以便在被发射去执行时，其输入可用。该软件通常是一个静态编译器或一个代码设计的虚拟机，如Transmeta Efficeon。

6.3 乱序发射逻辑

发射逻辑是一个关键组件，它决定了乱序处理器能够利用的指令级并行量。一旦源操作数可用，它就向功能单元发射指令，从而允许乱序执行。然而，发射过程中涉及的硬件组件位于处理器流水线的关键路径。因此，在不影响周期时间的情况下，实现一个能够利用指令级并行性的良好复杂度-有效的发射逻辑是非常重要的。

有许多不同的替代方案来解决涉及发射逻辑实现的多个设计决策。然而，本章的目的不是对所有可能的实现进行广泛的描述，而是展示最常见的示例，目的是了解硬件的特性。

在本章中，我们覆盖的两个主要场景，假设一个统一的发射队列。使用统一发射队列的处理器实现一个队列，其中存储所有重命名的指令，等待执行。这与其他方案不同，如保留站，分布式发射队列，其中指令根据执行所需的资源类型分配在单独的缓冲区中。

第一个场景代表处理器发射逻辑的实现，其中指令在进入发射队列之前读取其源操作数，如P6类体系结构。然后，作为第二个场景，我们描述了实现发射逻辑所需的主要更改，其中源操作数被发射执行后再读取，在MIPS R10000或Alpha 21264中是这样的。这两种方案适用于任何不同的现有方案，以保存指令生成的值（合并寄存器堆、重命名缓冲区、重排序缓冲区等）。

然而，由于这是一个正交设计决策，为了清楚起见，我们将为两种实现假设是一个合并寄存器堆方案。请注意，我们将一个合并寄存器堆称为一个寄存器堆，该文件存储了体系结构状态和推测值，如第5章中详细描述的。然而，所描述的硬件可以容易地适用于任何其他寄存器堆方案。

本章还介绍了其他替代方案，如分布式发射队列和保留站。这些替代方案将不太详细地解释，因为在实现中需要考虑的大多数权衡已经包含在上述场景中。

最后，我们特别关注内存操作的发射逻辑的实现。与在重命名阶段检查数据依赖性的其余操作相反，在内存操作计算其地址之前，无法识别内存依赖性。这一特性对这些指令的管理有重要影响，我们将在后面描述。

####6.3.1 发射前读取源操作数时的发射过程

在发射阶段之前读取操作数，发射队列的主要特征是，它需要保存执行发射的指令中的信息以及已生成的源操作数中的值。图6.1显示了用于存储此信息的典型组件的概述。图6.1中的每个块都表示一个表，表中的条目数与发射队列可以容纳的指令数相同。此外，为了简单起见，我们假设处理器具有类似于简化MIPS的ISA，其中指令最多可以有两个源操作数或一个源操作数以及作为指令一部分编码的立即数。

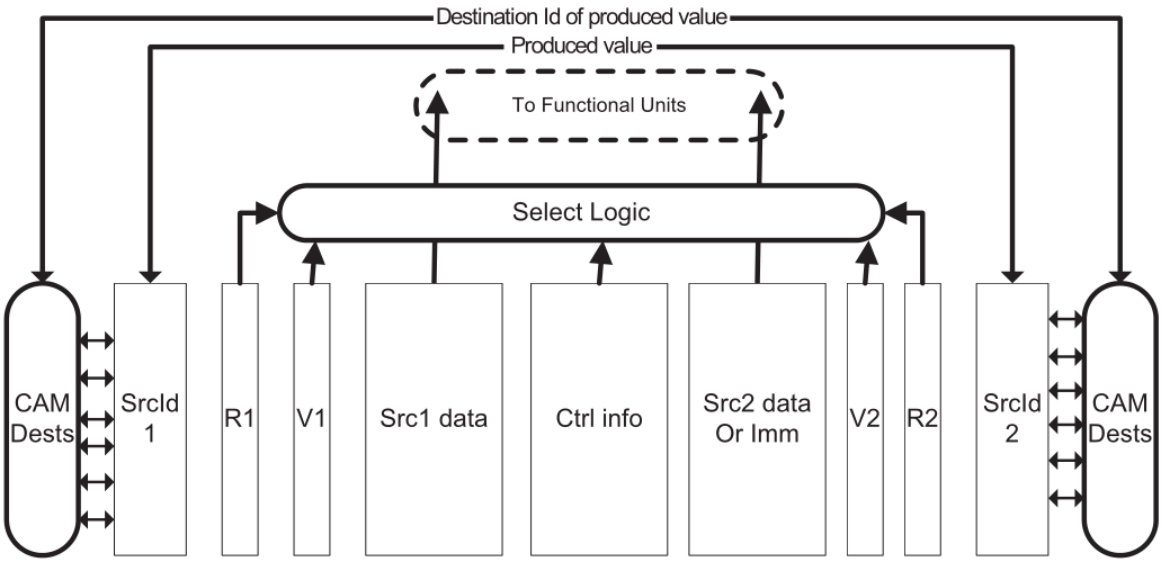

FIGURE 6.1: Hardware components of a typical issue queue where source operands are read before issue.

在我们的简单示例中，有一个名为*Ctrl info*的块。该块负责保存执行指令所需的所有静态信息（所需ALU的类型、内存操作的数据大小、立即操作数的使用等）。然后，有两个对称数组称为*Src1Id*和*Src2Id*。这些数组分别存储源1和源2的源操作数标识符。这些标识符在处理器中是唯一的，由重命名阶段生成，如第5章所述。如果由于指令没有两个源操作数而未使用其中一个源标识符，则该操作数的源ID将无效。使用有效数组*V1*和*V2*标识无效的源操作数。这些块每个条目实现1位，分别表示*Src1Id*和*Src2Id*是否有效。在本例中，我们假设立即值总是给到*Src2*。因此，如果指令有立即数，有效数组会将源标记为有效，但*Src2Id*会设置为0或任何其他从未用于重命名的标识符。0是一个很好的替代方案，因为寄存器0通常硬编码为0，所以它永远不是有用的目标。

块*Src1 data*和*Src2 data or Imm*负责存储输入值。如果指令具有立即数，则将其存储在块*Src2 data or Imm*中。

最后，就绪位（ready bit）数组*R1*和*R2*通知*Src1 data*和*Src2 data or Imm*是否已经生成了它们的值。一旦为指令的所有有效源设置了就绪位，就可以引导指令执行。

在介绍了主要组件之后，我们将描述发射队列上发生的不同事件，以及该硬件结构如何与处理器的其余部分交互。这些事件包括发射队列分配、指令唤醒、指令选择和发射队列回收。

6.3.1.1 发射队列分配

图6.2显示了在乱序处理器的通用流水线中发射队列集成的示例。如我们所见，指令首先在重命名阶段（也称为分配阶段）重命名，然后分配并进入发射队列。如果没有可用条目，分配阶段将暂停。请注意，由于时序限制，重命名阶段可能无法使用有关发射队列占用率的最新信息。重命名可能会对发射队列占用情况做出保守假设，以避免处理可能找不到可用发射队列条目的指令。

然后在下一个周期中访问寄存器堆，以便读取已经可用的源操作数。请注意，重命名表不仅存储每个源操作数的寄存器ID，而且还存储一个可用位（*available* bit），用于说明这是否已经可用。因此，指令必须从寄存器堆中读取设置了可用位的输入。

最后，与重命名指令关联的发射队列条目将填充重命名信息和从寄存器堆读取的数据。

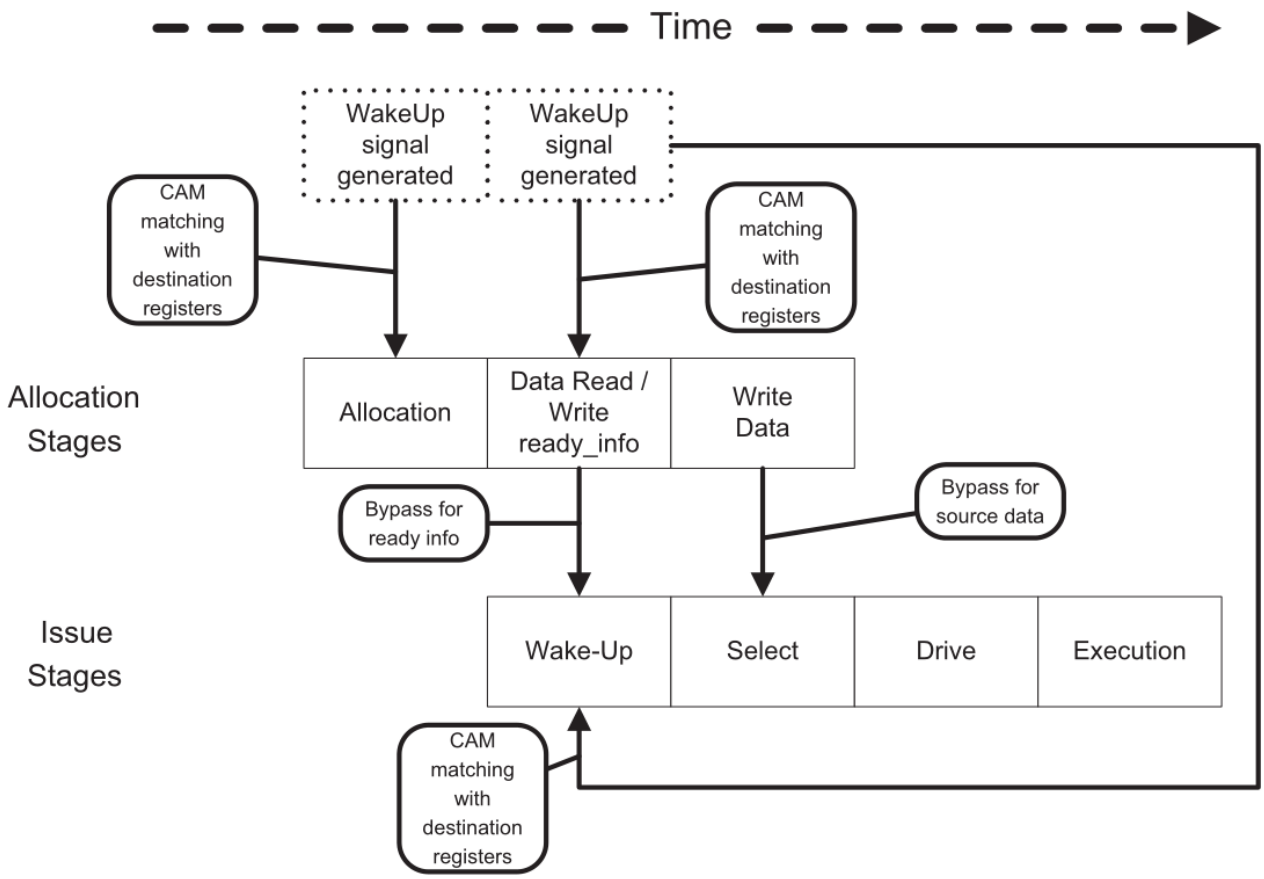

FIGURE 6.2: Example of the pipeline stages involved in the instruction issue on a generic out-of-order pipeline.

6.3.1.2 指令唤醒

唤醒是通知已生成一个源操作数的事件。该信号通常包括生成值的重命名ID、值本身和有效位。然后，CAM逻辑将该ID与*SrcId1*和*SrcId2*数组中的条目进行比较，如图6.1所示。如果匹配，则设置相应的就绪位，并将值复制到相应的*Src Data*条目上。一旦一条指令的有效源的就绪位被设置好，该指令就变得就绪（我们说它被唤醒），并且可以被选择逻辑视为被引导执行（steered for execution）。

请注意，唤醒信号只产生一次，出现时，使用者指令可能不在发射队列中。因此，应该保证一旦产生了值，所有使用者都会知道数据已经可用。这可以通过在重命名表中将上述提到的可用位设置为1来实现。

在发射前读取操作数的处理器通常在重命名阶段和将指令分配到发射队列之间至少需要一个额外的周期，如图6.2所示。请注意，此阶段的指令源操作数还应与即将到来的唤醒ID进行比较，以防止死锁。否则，如果它们的一些源操作数在此周期中生成，这些指令将永远不会被唤醒，因为它们在前一个周期读取可用位，并且在下一个周期之前不会写入发射条目以执行CAM匹配。

减少需要CAM匹配的流水线级数量的一个可能解决方案是提前写入*SrcId1*和*SrcId2*。从重命名表中获取此信息后，可以立即写入此信息。图6.2显示了一个场景，其中该信息在重命名后的周期中写入。然后，发射队列将在唤醒事件和*SrcId*字段之间进行常规比较，以使就绪位与其余条目保持最新。从寄存器堆读取值后，*Src1 data*和*Src2 data*将相应更新。

另一种可以避免我们在重命名和发射队列分配之间的所有阶段实现CAM匹配逻辑的方法是，将可用位作为重命名表中的一个单独表来实现。在这种情况下，将在重命名阶段访问重命名表，但在指令将其*SrcIds*分配到发射队列之前，不会访问可用位表。

与其他替代方法相比，提前*SrcId*分配具有一些优势，因为它允许指令在读取源操作数的同一周期内唤醒。因此，可以考虑在将整个数据写入发射队列的同一周期内执行，如图6.2所示。

如前所述，唤醒信号通知一个值已经可用。然而，为了最小化生产者和消费者之间的执行距离，可以在实际产生值之前生成该信号。可以提前通知唤醒信号，因为指令在到达执行阶段（即指令唤醒后的一个或多个周期）之前不需要其源操作数。例如，图6.3中的流水线在唤醒和执行之间实现了两个阶段。

图6.3显示了根据唤醒信号发送的时刻，生产者和消费者之间的执行周期数。在场景1中，只要生产者被执行并且值可用，就立即发送唤醒信号。然后，消费者在下一个周期被唤醒，并在生产者产生值后执行三个周期。如场景2所示，可以通过提前生成唤醒信号来避免这三个周期的气泡。在这种情况下，唤醒信号在生产者产生值三个周期前发送。然后，消费者将在生产者产生值后立即进入执行阶段，允许背靠背执行。请注意，在第二种情况下，值应该从功能单元的输出传输到另一个功能单元的输入。这种连接称为旁路，其实现在第7章中进行了描述。此外，图6.3所示的场景还假设生产者的选择和消费者的唤醒可以在一个周期内完成。这一设计决策对性能也至关重要，因为如果选择和唤醒不能在一个周期内完成，对于单周期延迟导致性能显著下降的指令，无法执行背靠背执行。总之，背靠背执行对性能至关重要，因此，大多数处理器都实现了它。

有两种常见的实现方式来生成唤醒信号。一种替代方法是在流水线阶段生成信号，指令在执行完成前三个周期驻留在该流水线阶段。请注意，指令执行所需的周期数取决于它使用的功能单元。例如，整数加法器通常需要一个周期完成，而整数乘法器或浮点功能单元可能需要更长的周期。因此，对于单周期操作，流水线应该能够从选择阶段生成唤醒信号，直到需要更长时间的功能单元结束前的三个周期。

另一种方法是将有效位数组的每个条目实现为移位寄存器加上有效位。这些移位寄存器也可以实现为记分板，每个物理寄存器有一个移位寄存器。每个移位寄存器的位数应与功能单元产生值所需的最大

周期数相同。然后，总是在选择阶段生成唤醒信号，并将移位寄存器的位设置为1，其位置等于功能单元的延迟减去1。移位寄存器每个周期移位，一旦移位寄存器的位0变为1，相应的有效位就被设置。

请注意，这些机制适用于指令延迟恒定且仅取决于指令本身的情况。这个假设适用于所有算术运算，但不适用于内存操作。内存操作（例如，load）的延迟取决于它在数据cache或数据TLB中命中还是未命中。不幸的是，只有在发射load、计算其地址并访问这些结构时才知道这一点。

可以通过延迟唤醒信号的生成，直到我们知道load是否会命中cache和TLB，以保守的方式处理load。然后，如果命中，我们可以立即唤醒消费者。如果未命中，在未解决未命中问题之前，我们不会生成唤醒信号。然而，一个程序中加载操作的平均数量约为20%，其中大多数都有消费者，因此延迟这些操作的唤醒信号将对性能产生重大影响。因此，一些处理器会在假设命中延迟的情况下推测式地唤醒load消费者，并在cache中load未命中的罕见情况下付出惩罚。本章后面将解释推测式唤醒。

6.3.1.3 指令选择

选择逻辑（又称select logic）负责选择发射队列中准备就绪的指令子集，这些指令将在给定的周期内被引导。

如果一条指令的源操作数已准备就绪且所需的执行资源可用，则可以选择该指令。例如，如果处理器只实现一个乘法器，就不可能并行控制两个乘法指令。

选择逻辑的时序非常关键，因为它必须在唤醒逻辑之后进行，以支持单周期延迟操作的背靠背执行。因此，处理器通常不会实现单一的选择逻辑，但它们将其分配到称为仲裁器或调度器的简单组件中。例如，一个处理器每周期最多可以控制4条指令，它可以实现2个或4个仲裁器，并且每个功能单元将静态绑定到一个仲裁器。等待在发射队列上的指令也绑定到一个仲裁器。这种配置允许对每个功能单元的选择逻辑进行并行化。否则，仲裁器应该同步，以确保它们不会选择同一条指令，或者不会将两条不同的指令引导到同一执行资源。

图6.4显示了基于仲裁器的选择逻辑的可能实现，和Alpha 21264一样。在这个简化的示例中，我们只关注整数算术运算，最大发射宽度为4，但限制为最多三个简单算术运算和一个乘法运算。请注意，4位宽发射处理器通常不允许发出任何类型的4条指令，它们受到所实现的功能单元的限制。通用处理器设计实现了几个执行简单算术运算的单元，但其中只有一个或几个支持复杂算术运算。图6.4所示的示例将发射宽度拆分为两个仲裁器*A*和*B*，其中仲裁器*A*绑定到一个简单的功能单元和一个乘法器，而仲裁器*B*绑定到两个简单的功能单元。

发射队列中的指令也绑定到仲裁器。指令重命名后，将由仲裁器操纵逻辑分配给仲裁器，并在指定仲裁器的子阵列上分配一个条目。这种操纵逻辑通常非常简单，并向仲裁器发送指令，其功能单元可以执行它们，试图保持每个仲裁器分配的指令量相同。

每个周期，每个仲裁器都会检查其子阵列中是否有可用的就绪指令，并选择要引导执行的指令。在Alpha 21264中，当子阵列中的就绪指令数超过发射宽度时，最早的指令始终具有优先级。该算法相对容易在Alpha 21264等处理器中实现，在这些处理器中，指令总是按顺序存储在发射队列子阵列中。然而，其他处理器设计，如MIPS或Intel P4，如果指令顺序不保留在发射队列中，则要么实施伪基于年龄的算法，要么只是根据发射队列中的位置简单地划分优先级。

6.3.1.4 表项回收

一旦选择了一条指令并将其数据转发给功能单元，就可以安全地回收其发射队列条目。然而，一些技术，比如推测式唤醒，可能需要延迟回收，直到我们确定指令可以执行为止。推测式唤醒通常用于减少load操作与其使用者之间的延迟。第5.2.4节介绍了这种技术。

6.3.2 发射后读取源操作数时的发射过程

本节介绍在发射指令后读取源操作数的方案与之前的方案之间的主要区别。

发射后读取和发射前读取的关键区别在于，发射后读取不需要发射队列存储源值。因此，唤醒信号不需要转发这些值。图6.5显示了在发射前和发射后读取数据时，发射队列组件之间的差异。如果我们在发射后读操作数，灰色部分则不需要。请注意，即使我们完全删除了*Src1 data*，*Src2 data*仍然保留，尽管它可能更小。原因是这个方案仍然需要空间来容纳立即数值。然而，这些值通常比寄存器值短。此外，很少有指令需要此字段，因此可以将其实现为一个单独的结构，只需很少的条目。在后一种实现中，需要立即数的发射队列条目将包括立即数所在结构内的偏移量。

图6.6显示了该发射方案的可能流水线。可以看出，由于尚未读取数据，重命名阶段和发射队列分配之间的阶段数量减少。这意味着所需CAM匹配逻辑的数量减少。

但是，为了读取源操作数，唤醒和执行阶段之间的周期数增加一个周期。

发射前后读取的另一个显著区别在于寄存器堆中所需的读取端口数。在发射之前读取时，寄存器堆中所需的读取端口数由机器宽度决定，机器宽度通常介于3和4之间，而在发射之后读取时，读取端口数由发射宽度决定。虽然看起来可能有悖常理，但发射宽度有时比机器宽度更宽。原因是发射仲裁器不是通用的，但它们在某些类型的指令中是专用的。例如，一个具有算术和内存操作专用仲裁器的体系结构，能够执行多达4个整数算术操作或4个内存操作，将需要8的发射宽度。我们可以在Netburst架构上找到一个例子，它是一台3宽的机器，每个周期最多可以发射6条指令。

6.3.2.1 读取端口减少

寄存器堆的面积、功耗和访问延迟随着读取端口的数量增加而增加。因此，重要的是尽量减少端口数量，以实现节能设计。

假设在最坏的情况下，发射宽度得到充分利用，并且所有指令都从寄存器堆中读取操作数，一些处理器设计会根据需要实现尽可能多的读取端口。然而，Alpha 21264将每个寄存器堆拆分为两个复制的物理寄存器堆，每个文件的读取端口总数为总读取端口数的一半，从而减少了每个寄存器堆的读取端口数。这种解决方案减少了读取端口的数量，代价是惩罚，一个周期内访问不同寄存器堆的生产者和消费者之间的背靠背执行。

文献表明，大多数数据源都是从旁路网络而不是从寄存器堆中读取的。此外，发射宽度通常没有得到充分利用。因此，在对性能影响最小的最坏情况下，也可能实现一个端口数少于所需端口数的寄存器堆。

有两种可能的替代方法：主动和被动。主动替代方案包括同步仲裁器，以便计算每个选定指令将使用的读取端口数。如果所需的读取端口数超过了可用端口数，一些仲裁程序将取消发射过程。请注意，发射逻辑分布在仲裁器中的原因是通过完全并行化来减少该逻辑的延迟。因此，实现此读取端口的同步可能会影响延迟。

被动替代方案使仲裁器发射指令，假设永远无法达到读取端口的总量，并在罕见的情况下发生反应。在这个替代方案中，在发射指令时分配读取端口。然后，如果超过了可用读取端口的数量，则应取消某些已发射的指令，并重新发射。取消和重新发射可以使用在执行推测式唤醒的处理器中实现的任何替代方案来完成。5.3中描述了其中一些技术。

最后，请注意，反应机制可能会导致仲裁者饥饿，甚至活锁（live locks）。因此，定义公平的政策以执行取消非常重要。

####6.3.3 针对乱序发射的其他实现

前面场景的讨论涵盖了在发射逻辑的实现上需要解决的大多数主要设计决策。然而，为了完整起见，我们在本节中简要介绍了现代处理器中可以找到的另外两种发射实现：分布式发射队列和保留站。

6.3.3.1 分布式发射队列

实现此方案的处理器将功能单元分布在执行集群中，每个集群实现自己的发射队列。例如，英特尔奔腾4实现了两个带有专用发射队列的执行集群：一个用于内存操作，另一个用于非内存操作。在这种情况下，指令将根据操作类型导向其中一个发射队列，然后将其绑定到发射队列中的特定调度器。

#####6.3.3.1 保留站

保留站是每个功能单元的专用缓冲区，用于存储将在特定功能单元上执行的指令及其输入值。该方案由R.Tomasulo于1967年为IBM 360/91提出，是现代超标量处理器的基础。

实现此方案的处理器在重命名到特定功能单元的缓冲区后引导指令，在缓冲区中等待输入值生效。然后，每条指令向所有功能单元的所有保留站广播其产生的值。可以看出，这种技术需要一些设计决策，类似于发射队列场景，即在发射之前读取数据。

6.4 内存操作的发射逻辑

内存操作通过内存具有数据依赖性，在重命名阶段无法识别。只有在发射这些指令并计算其地址后，才能检查这些内存相关性。

负责处理内存依赖关系的机制称为内存消歧策略。不同的处理器实现完全不同的内存消歧策略。表6.1显示了在不同微体系结构上实现的一些典型方案。这些方案可分为两组：非推测消除歧义策略和推测消除歧义策略。第一组不允许执行内存操作，直到我们确定它与以前的任何内存操作都没有依赖关系。相比之下，第二组预测一个内存操作是否会与另一个执行中的内存操作有依赖性。

选择适当的内存消歧对处理器设计的性能和复杂性至关重要。处理器执行的指令中大约有30%是内存操作。因此，实现非常保守的内存消歧策略可能会产生不必要的执行序列化，这可能会显著限制可以利用的指令级并行性。另一方面，非常积极的内存消歧策略可能最终导致复杂的恢复机制和由于错误推测而导致的显著功耗增加。

6.4.1 非推测消除歧义策略

非推测消除歧义策略不会发射任何内存操作，直到所有以前的stores都计算了它们的地址。因此，可以安全地计算内存依赖关系。现有处理器实现的非推测消除歧义策略主要有三种类型：全排序、带store排序的load排序和部分排序。

TABLE 6.1: Memory disambiguation schemes.

NAME	SPECULATIVE	DESCRIPTION
Total Ordering	No	All memory accesses are processed in order.
Partial Ordering	No	All stores are processed in order, but loads execute out of order as long as all previous stores have computed their address.
Load Ordering Store Ordering	No	Execution between loads and stores is out of order, but all loads execute in order among them, and all stores execute in order among them.
Store Ordering	Yes	Stores execute in order, but loads execute completely out of order.

在全排序中，所有内存操作都是按顺序执行的。目前，据我们所知，没有乱序处理器实现全排序，因为它限制了我们可以利用的大量并行性。

相比之下，其余的非推测方案允许加载操作相对于存储乱序执行。在使用存储排序方案进行加载排序的情况下，加载按顺序进行，存储按顺序进行。但是，加载不必等待以前的存储访问缓存。我们可以在AMD K6等处理器中找到此方案。然而，在部分排序中，加载可以乱序处理。在这种情况下，只要load的源操作数就绪，并且之前的所有存储都已经计算了其地址，就可以发出load。实现部分排序的处理器示例有MIPS R10000和AMD K8。

注意，只要计算出store的内存地址，就可以执行内存消歧。因此，一些处理器将存储操作分为两个子任务：一个子任务计算地址，另一个子任务获取数据。然后，存储操作不必等待数据的生产者完成以计算其地址。在某些情况下，甚至处理器也会尽快计算地址，并且在存储成为最旧的执行中指令之前不会读取数据，如HP PA8000中所述。

在接下来的部分中，我们将介绍两个在现有处理器中实现的非推测内存消歧策略的案例研究。第一个示例是AMD K6处理器，它实现了加载排序、存储排序方案。第二个示例显示了在MIPS R10000上实现偏序。

6.4.1.1 案例研究1：AMD K6处理器上的加载排序和存储排序

AMD K6处理器为加载和存储操作实现了两条单独的管道。这两条管道都通过某种程度的通信进行了解耦，但指令在每条管道内按严格的顺序流动。这两条管道的简化示例见图6.7。此内存管道实现以下组件以执行消歧：

- 加载队列：此队列按程序顺序存储加载操作。重载在重命名后插入到此队列中，并驻留在此队列中，直到它们成为队列中最旧的且其源操作数就绪。地址生成：这是负责根据源操作数计算内存操作的访问地址的逻辑。存储队列：此队列按程序顺序存储存储操作。存储指令位于此处，因为它们已被重命名，直到它们成为队列中最旧的指令；计算地址所需的源操作数可用。

- 存储缓冲区：该缓冲区保持存储操作的程序顺序，直到它们成为处理器中最古老的飞行指令，然后继续更新内存。

负载流经管道的上部，如图6.7所示，而通过管道的流量存储在底部。这些操作在每个管道阶段上处理如下。加载和存储在其源操作数就绪并分别成为加载和存储队列中最旧的指令后，将在发出阶段发出。请注意，在存储的情况下，问题逻辑不等待存储在内存中的数据准备就绪，而只等待计算存储地址所需的源操作数。然后，在读取阶段加载并存储从寄存器堆计算地址所需的读取源操作数。这些数据可以从寄存器堆读取，也可以从旁路逻辑获取。在地址生成阶段，计算内存操作将访问的地址。如果是存储，则读取要存储在内存中的值。如果此值不可用，则存储操作的整个管道将暂停。一旦计算出地址，操作就进入消歧阶段。对于存储，数据和地址都存储在存储缓冲区中。

该信息将驻留在存储缓冲区中，直到该操作成为处理器中最旧的指令。然后，缓存将在内存级上相应地更新。对于加载，加载会将其内存地址与存储缓冲区中早于它的存储地址进行比较。此外，如果该存储较旧，加载还将与地址生成阶段上的存储进行部分比较。此比较是部分比较，因为在进行比较之前，此存储没有时间完全计算其地址。因此，只比较少数位，如果这些位匹配，则认为负载取决于此存储。最后，负载检查调度程序，以确保没有尚未计算其地址的旧存储。然后，如果加载与任何以前的存储发生冲突，或者问题队列中有旧的存储，则加载和整个加载管道将暂停。我们可能认为按顺序处理负载是一个不必要的约束，因为负载不会修改内存，并且它们之间没有依赖关系。然而，这是实现x86参考手册中指定的处理器一致性的一种简单方法：存储必须按顺序可见，加载必须按顺序执行。不过，可以在不序列化加载执行的情况下支持处理器一致性。例如，AMD在AMD K8L上改进了其内存消歧方案，允许负载在某些情况下超过以前的负载，并且仍然保持对处理器一致性的支持。另一个例子是Intel Core架构。在这种情况下，在推测性内存消歧模型的基础上还支持处理器一致性，该模型无序执行加载，甚至超过了尚未计算其地址的存储。在无序执行内存操作的处理器中，对保证处理器一致性的机制的描述超出了本章的范围。

6.4.1.2 案例研究2：MIPS R10000处理器上的部分排序

MIPS R10000处理器实现部分排序。因此，只要以前的所有内存操作都计算了它们的地址，就可以乱序执行加载。相反，存储是按照严格的程序顺序处理的。图6.8显示了内存消歧过程中涉及的流水线级和组件的示意图。

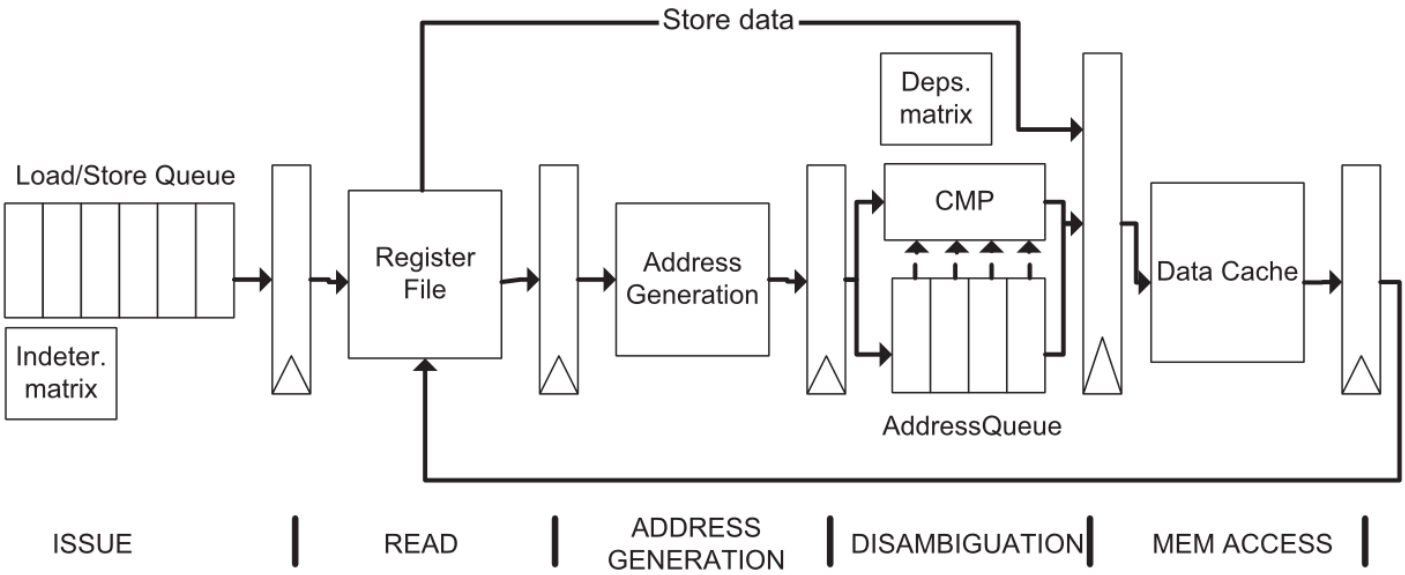

FIGURE 6.8: Schematics of the MIPS R10000 pipeline to implement the partial ordering memory disambiguation policy.

此内存流水线实现以下组件以执行消歧：

- 加载/存储队列：这是一个16个条目的队列，其中加载和存储指令在重命名阶段后按顺序存储。在源操作数就绪之前，指令不会离开此队列。
- 不确定矩阵：这是一个16x16的半矩阵，其中每一列和每一行表示加载/存储队列上的一个条目。内存操作在其列上将所有条目设置为1并且在计算其地址时重置它们。然后，当属于旧内存操作的行的任何位置上都有1时，无法发射内存操作。图6.9显示了一个6条目加载/存储队列的不确定矩阵示例。
- 依赖关系矩阵：这是一个16x16矩阵，其中每一列和每一行表示加载/存储队列上的一个条目。一种依赖于以前存储的加载，对于属于它所依赖的存储的所有列，该加载将其行上的所有条目设置为1。然后，在重置其行上的所有条目之前，它将无法恢复执行。相反，存储在更新内存时会重置其列上的所有位。图6.10显示了一个6条目加载/存储队列的依赖关系矩阵示例。
- 地址生成：这是负责根据源操作数计算内存操作的访问地址的逻辑。
- 地址队列：此队列保留要访问缓存的加载和存储的内存地址。在加载的情况下，除了将其地址写入此队列外，他们还将其与以前所有存储的地址进行比较，如果匹配，则在依赖关系矩阵上设置相应的条目。

在重命名阶段，所有内存操作都会激活其列在不确定矩阵上的位。请注意，与指令关联的列取决于它在加载/存储队列中所处的位置。然后，只要它在不定矩阵上的行不包含任何一个（所有以前的内存指令都计算了它的地址），并且它的源操作数可用，就会发出内存操作，并在读取阶段读取它的源操作数。

一旦内存操作计算出其地址，它就会存储在地址队列中。对于存储，它将驻留在那里，直到它成为流水线中最古老的指令。然后，将发射重置其在依赖关系矩阵上的列的命令，并使用专用读取端口从寄存器堆读取要存储在内存中的数据。请注意，已经计算了这些数据，因为存储是流水线中最古老的指令，所以以前的所有指令都已生成结果并已失效。相反，加载会将其地址与以前存储的地址进行比

较，并相应地更新依赖关系矩阵中的行。加载将在地址队列上等待，直到其在此矩阵上的行完全重置。

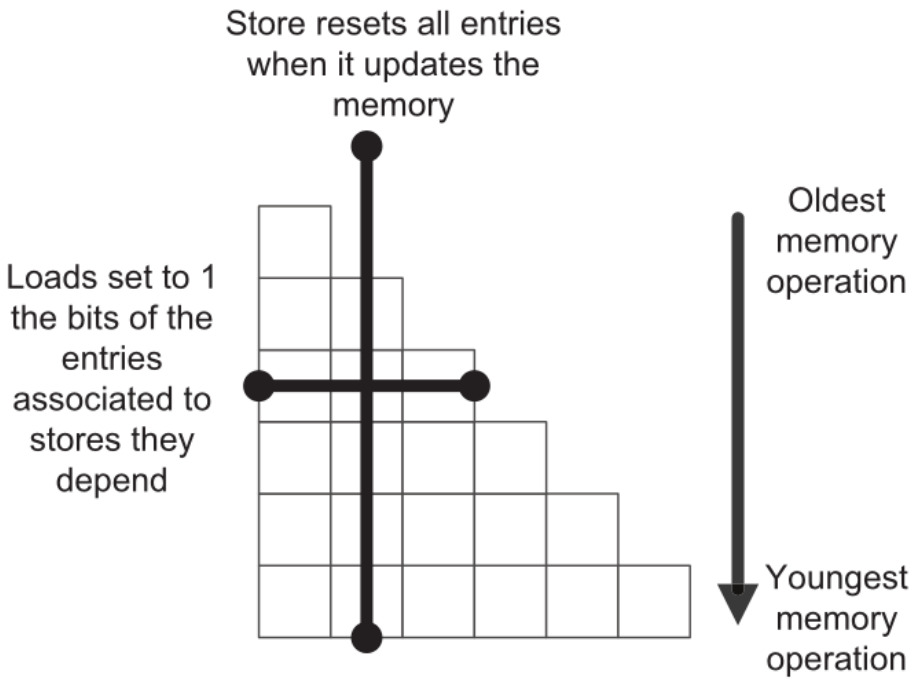

FIGURE 6.10: Example of a 6-entry dependency matrix.

MIPS R10000实现了一个2路数据缓存，其中指令乱序访问。如果缓存未命中，指令必须等待，直到未命中得到解决，但其他内存指令可以同时继续。这种情况可能发生在一种称为抖动（thrashing）的不良情况下。当缓存上的内存操作未命中时，会发生抖动，即，当未命中得到解决时，在读取数据之前，另一条指令已再次逐出该行。请注意，如果影响流水线中最旧的指令，则可能会在活锁上发生抖动。MIPS R10000通过锁定流水线中最旧的内存操作将访问的一路set，直到它成功读取数据为止的方式来避免抖动。最古老的指令在消歧阶段锁定其集合的一路。

最后，如前所述，一旦允许离开地址队列，加载和存储指令就访问内存。

6.4.2 推测式内存消歧

一些最新的处理器设计，如Alpha 21264或Intel Core架构，已经实现了推测性内存消歧。这些处理器通过推测性地发射预计不依赖于任何以前的执行中存储操作的加载指令来提高性能。因此，加载操作不必等待所有以前的存储操作来计算其地址。请注意，由于此方案在内存依赖性上推测，因此可能会发生预测失误，最终导致应用程序执行错误。因此，这些处理器需要特殊的硬件来识别这些预测失误并恢复执行。

我们将Alpha 21264的内存消歧流水线描述为该策略的一个示例。

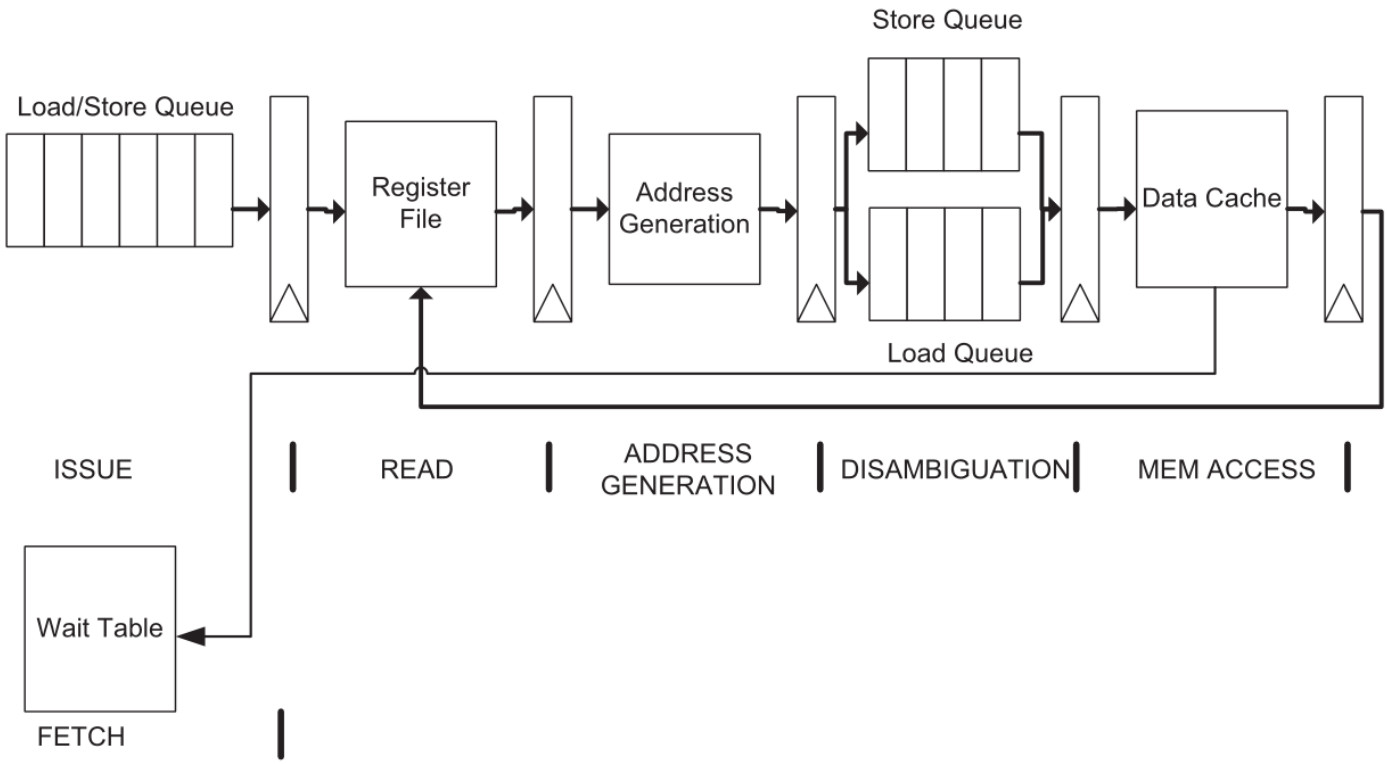

FIGURE 6.11: Schematics of the Alpha 21264 pipeline to implement a speculative memory disambiguation policy.

#####6.4.2.1 案例研究：Alpha 21264。此内存流水线实现以下组件以执行消歧：

- 加载/存储队列：此队列保存内存操作，直到它们计算出源操作数并可以发射以供执行。
- 加载队列：此队列按程序顺序存储加载的物理地址。加载指令在重命名阶段在此队列上分配一个条目，并在其失效时回收该条目。此队列实现32个CAM条目。
- 存储队列：此队列按程序顺序存储存储指令的物理地址及其数据。存储指令在重命名阶段分配一个条目，直到提交后才会回收该条目。此结构还实现了32个CAM条目。
- 等待表：此表实现1024个由虚拟pc索引的1位条目。每当我们确定某个加载依赖于它所超越的存储时，就会生成存储加载顺序陷阱，并将此表更新，将表示加载虚拟地址的条目设置为1。取指单元读取此表，以便标记加载指令，以防它们在过去产生内存冲突。然后，这些加载将不再是推测性的发射。然而，等待表每16384个周期重置一次，因为否则我们会得到一个全是1的表。

加载和存储在加载/存储队列上等待，直到其源操作数就绪。在存储的情况下，我们等待计算地址和数据所需的源操作数，与之前的案例研究相反。如果加载设置了等待位，则在所有以前的存储都已发射之前，加载不会离开加载/存储队列。

然后，在下一个周期中读取用于计算地址的源操作数，并在一个周期后计算内存地址。

已计算其地址的加载将其保留在加载队列中。此外，他们将其地址与较年轻加载的地址进行比较，如果它们匹配，则会触发加载-加载内存冲突陷阱。此陷阱使处理器从触发陷阱的加载开始恢复执行。如果不需要陷阱，加载将继续访问缓存。

在内存消歧阶段，存储还将其内存地址写入存储队列条目。此外，他们检查加载队列，寻找与其地址匹配的更年轻的加载。如果发生这种情况，将发生存储-加载内存冲突陷阱，并从加载恢复执行。此外，更新等待表是为了标记此加载的未来实例，并避免这种情况再次发生。请注意，存储不会检查存

储-存储内存冲突，因为即使存储乱序发射，它们在提交前也不会更新缓存。由于失效是按程序顺序发生的，所以不会发生存储-存储内存冲突。

6.5 加载消费者的推测式唤醒

加载操作的延迟是可变的，主要取决于加载是否命中TLB和数据缓存。还有其他因素可能会影响数据可用的最终周期，例如，数据缓存上的bank冲突、读取端口与MOB中的其他内存操作冲突等。然而，最常见的情况是加载提供的数据具有命中的延迟，如图6.12所示。在该图中，我们可以看到两种可能的情况，其中使用者指令读取由加载生成的值。可以看到，加载在选择三个周期后计算是否命中缓存。因此，如果我们实施保守的唤醒，只有在保证加载会命中的情况下才能唤醒消费者，那么我们将在生产者和消费者之间获得两个周期的气泡，如场景1所示。然而，如果我们假设加载会命中，那么我们可以推测地触发唤醒信号，像场景2中那样，那么我们也可以为加载操作实现背靠背执行。

然而，在缓存中加载未命中或其执行因其他原因延迟的罕见情况下，必须取消并重新发射使用者指令。

指令一离开发射队列，就不能保证此指令在此队列上有一个空槽以便返回。发射队列可能充满了指令，这些指令实际上可能依赖于重新发射的指令。因此，让指令等待发射队列上的空闲插槽以便重新发射可能会导致死锁。

有几种解决方案可以避免这种死锁，各有优缺点。一种解决方案是刷新流水线中比要重新发射的指令年轻的所有指令，并从那里恢复执行。此解决方案类似于在Alpha21264上实现的上述机制，用于从内存消除歧义错误中恢复。这种方案的主要缺点是，如果这种情况经常发生，我们可能会有显著的性能下降。这就是Alpha21264实现等待表的原因之一。

、、另一种减少重新发射指令的性能损失的解决方案是推迟回收由指令分配的发射条目，直到我们确定不必重新发射此指令为止。在这种情况下，每个发射队列条目都有一个位（已发射位），表示是否已发射此指令。然后，选择逻辑不考虑已发射的指令。然而，每当需要重新发射指令时，受影响条目的已发射位就会重置，以便选择逻辑再次考虑执行这些指令。与前一种机制相比，这种机制减少了重新发射指令的代价，但增加了发射队列的压力。请注意，由已发射的指令分配的所有发射队列条目不能用于查找进一步的独立指令。

不幸的是，发射队列条目的数量通常很小，因此像前一种解决方案一样的解决方案可能会由于发射的指令使用条目而降低性能。netburst体系结构实现了小型发射队列，以便将它们适应其紧凑的周期时间。此外，由于它实现了一个深层流水线，因此，自从指令被发射就应该保持在发射队列上，直到我们确定它们不会被重新发射为止，这样的周期数量会很大。因此，像P4这样的netburst架构实现了重播队列。在这种情况下，指令一旦被引导执行，就会离开发射队列（或调度程序），但它们在一个称为重播队列的附加fifo结构中排队。然后，指令驻留在此队列中，直到保证它们不必再次发射为止。但是，如果需要重新发射指令，调度程序将优先考虑重播队列，以便按照在那里分配的顺序重新发射其指令。

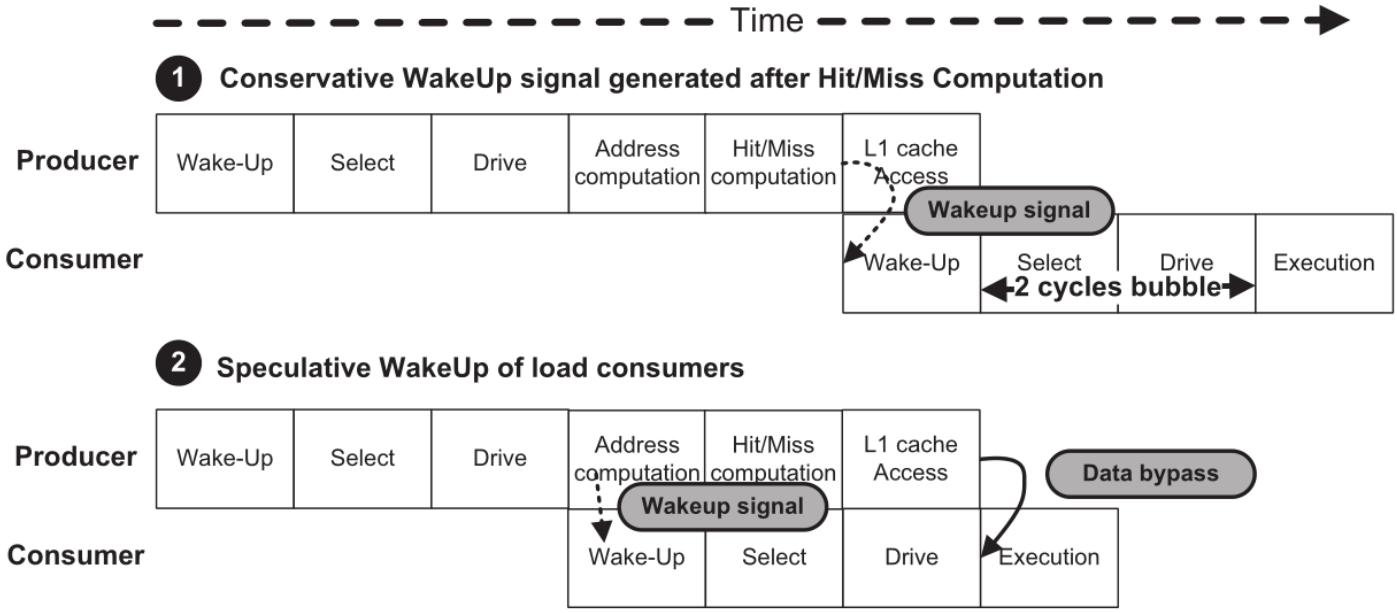

FIGURE 6.12: Load pipeline and consumer pipeline assuming (1) conservative wakeup and (2) speculative wakeup.

第七章 执行

在执行阶段计算程序结果。在此阶段，指令的输入操作数（也称为源操作数）与指令中编码的操作一起发送到处理器的计算单元。处理器对指令源操作数进行操作并产生计算结果。

处理器可以在执行阶段执行多种类型的操作。最常见的是算术运算（加法、乘法等）。内存指令通过将数据从内存加载到寄存器或将数据从寄存器存储到内存来对数据进行操作。控制流指令更改程序计数器（PC）寄存器的值。更为罕见的是，专用指令可以通过操作控制寄存器（定义处理器行为的特殊寄存器）来更改机器状态。

当然，不同类型的操作具有不同的复杂性，因此，延迟也不同。因此，在当代微处理器中，执行阶段不是单个流水线阶段，而是多个流水线阶段。此外，处理器流水线中通常有多条不同的路径，指令到达执行阶段时可以遵循这些路径。最明显的是整数路径、内存路径和浮点路径，它们具有不同的延迟。当操作结果生成并写入机器寄存器时，所有这些路径在回写阶段合并。

大多数通用乱序处理器共享图7.1所示的执行单元组织。图中灰色阴影区域显示处理器的功能单元（FUs）。功能单元对应于处理器的实际计算资源。在图中，我们可以看到四种不同类型的单元。正如其名称所示，浮点单元（FPUs）对浮点值执行算术运算。算术和逻辑单元（ALUs）是执行整数算术运算和布尔逻辑运算的单元。地址生成单元（AGUs）计算加载和存储指令的内存地址。最后，分支单元计算控制流指令的结果PC值。

图7.1中的寄存器堆对应于与指令执行相关的处理器的所有体系结构和执行中寄存器存储元素。有关该状态可能的组织（即合并的寄存器堆，重排序缓冲区中带值的体系结构寄存器堆等）。在本章中，为了简单起见，我们将假设一个合并的寄存器堆组织，它同时保存体系结构和执行中的值。

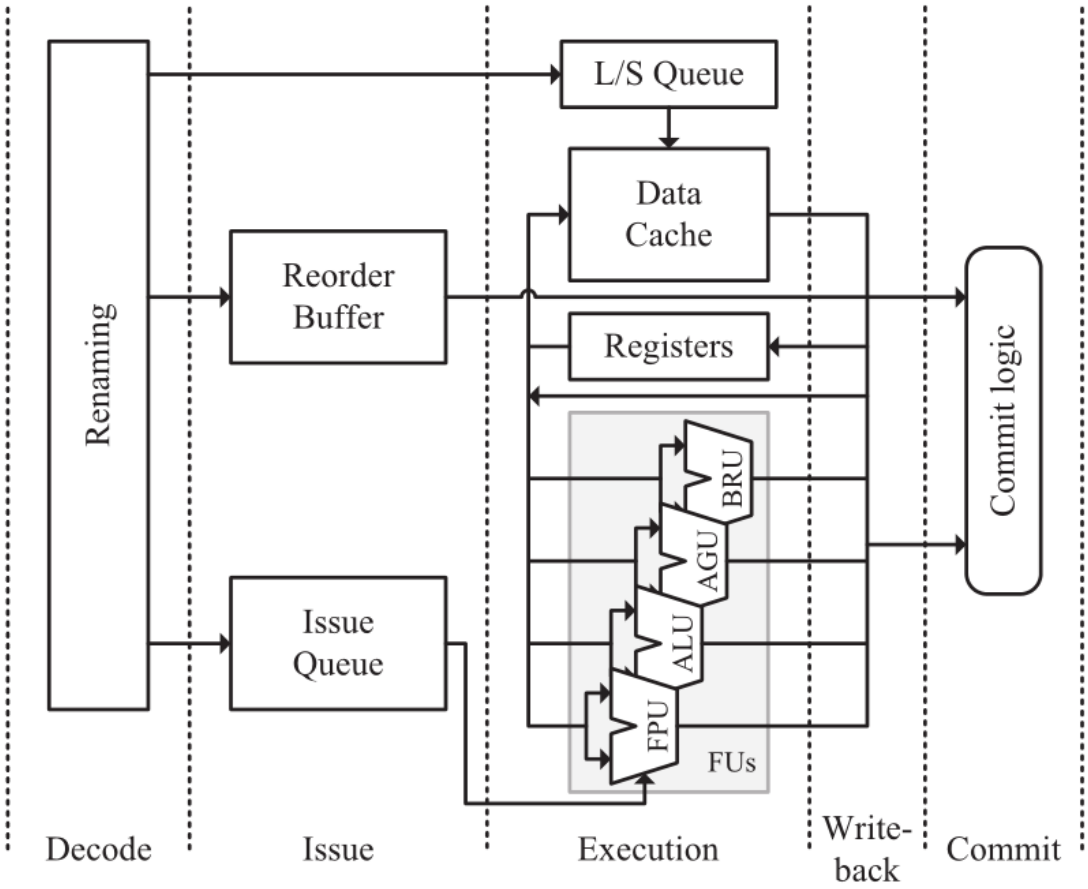

FIGURE 7.1: Processor execution units.

数据缓存是处理器执行单元中的另一个重要部分（参见第2章）。缓存用于提供对内存中常用数据的快速访问，是加载/存储执行流水线的组成部分。加载和存储指令的执行也意味着地址转换单元的使用（图7.1中未显示）。地址转换单元负责将加载/存储指令中编码的虚拟内存地址转换为操作系统为程序分配的物理内存地址。

执行阶段的另一个重要方面是旁路网络。这是负责在各个功能单元、数据缓存和寄存器堆之间移动源和计算结果的网络。在现代微处理器中，如果我们想提供相关指令的背靠背执行，则需要某种形式的旁路。由于其对性能的重要性和复杂性，旁路网络是执行阶段的关键组件之一。

接下来，我们将描述现代处理器中常见的功能单元类型，特别强调用于多媒体支持的功能单元。然后，我们描述了几种旁路网络组织：对于简单的乱序机器、广泛的乱序机器和按序机器。最后，我们研究了集群组织的设计，这些组织已在一些微处理器中使用，以减少功耗、面积和布线延迟的影响。

7.1 功能单元

现代微处理器上的功能单元可以根据其执行的操作类型和操作的数据类型进行分类。在本节中，我们确定了不同类型的功能单元，并对每个单元可以执行的操作进行了描述。

7.1.1 整数算术逻辑单元

该单元对来自通用寄存器堆或内存的两个整数值进行操作，并生成一个整数结果。整数算术逻辑单元（ALU）执行算术运算，如整数加减。根据ALU的不同，它还可以执行整数乘法和整数除法。ALU还执行位逻辑AND、OR、NOT和XOR操作。ALU还可以提供逐位NAND、NOR和XNOR操作，以及第二个操

作数反转的逻辑操作，如ANDN、ORN和XORN。最后，ALU执行数据转换操作，例如左移或右移和旋转，以及对其两个操作数之一的位进行转置（例如字节交换）。

某些指令集（如Intel x86和IBM POWER）实现条件代码或标志。例如，在x86 ISA中，有六个算术标志：符号、奇偶校验、调整、零、溢出和进位。这些标志是算术或逻辑运算的结果，即ALU中执行的任何操作。因此，通常x86兼容处理器的ALU具有计算算术标志以及每次计算结果的功能。

7.1.2 整数乘除

整数乘法和除法虽然对整数值进行操作，但ALU不支持。由于这些操作所需电路的高度复杂性和面积成本，它们在处理器中被构建为单独的执行单元（我们在这里称它们为IMUL和IDIV单元）。

此外，为了节省面积和功耗，许多处理器没有实现这些单元。相反，他们使用浮点单元（FPU）来执行整数乘法和除法。Intel Atom处理器就是这样一种处理器。要以这种方式执行整数乘法，首先将整数源转换为浮点值，然后将两个数字相乘，最后将结果返回到整数以生成最终的指令结果。与使用IMUL单元相比，此过程意味着乘法的延迟更高，但取决于应用程序，它可能值得，因为节省了功耗和面积（典型应用程序只有很少的整数乘法和除法指令）。

7.1.3 地址生成单元

内存指令通常将要操作的内存地址表示为多个源操作数的函数。地址生成单元（AGU）的目的是从内存指令的操作数生成指向程序地址空间的直接指针。AGU的操作在很大程度上取决于机器支持的内存模型。目前微处理器中有两种常用的内存模型：平面内存（flat）和分段内存（segmented）。

、、在平面内存模型中，内存对程序来说是一个单一的连续地址空间。我们把这个空间称为线性地址空间，平面内存地址称为线性地址。

在分段内存模型中，程序将内存视为独立地址空间（段）的集合。段定义从段基址开始的单个连续地址空间。在这个模型中，程序发出逻辑地址来访问内存。逻辑地址由段标识符和段内的偏移量组成。微处理器的内存系统内部只使用线性地址，因此程序逻辑地址必须由处理器的AGU转换为线性地址。这是通过将段基址和逻辑地址的段偏移部分相加来实现的。

在这两种内存模型中，地址的线性部分（即平面模型中的线性地址和分段模型中的逻辑地址的偏移部分）称为有效地址，并表示为一个或多个指令操作数的函数。指令的寻址模式定义如何组合源操作数以生成有效地址。

具有最复杂AGU需求的ISA之一是x86 ISA。x86遵循分段内存模型，有六种不同的寻址模式。在x86中，有效地址计算由以下组件组成：

- 位移：在指令位中编码的立即数。
- 基址：通用寄存器的值。
- 索引：通用寄存器的值。
- 比例：指令位中编码的常数1、2、4或8。比例值乘以索引值。

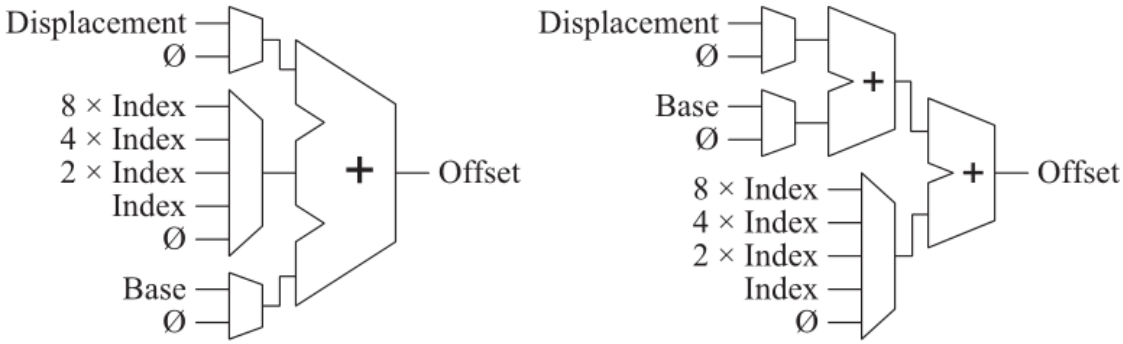

FIGURE 7.2: An x86 AGU offset calculation. The left circuit uses a three-input adder, while the right one uses a chain of two adders.

Wake-up	Select	Data read	Drive	Calculate offset / read segment info	Calculate linear address / check segment limit

FIGURE 7.3: Address calculation pipeline for x86.

六种x86寻址模式允许以下表达式的所有组合：

偏移=基础+（索引×刻度）+位移

图7.2显示了x86 AGU偏移计算电路的两种不同实现。计算偏移量只是AGU地址计算的一部分。如前所述，AGU必须通过向计算出的偏移量添加段基来生成内存访问的线性地址。此外，AGU必须对偏移量进行限制检查；也就是说，它必须检查偏移是否在线段边界内。

从上面的讨论可以看出，计算x86处理器的线性地址是一项复杂的操作。以高频工作的现代微处理器无法在一个周期内执行如此复杂的操作。图7.3显示了可能实现此类AGU的执行管道阶段。这是一个两个周期的过程。在第一个周期中，计算偏移量，并从段寄存器文件读取段基址和极限信息。

在第二个周期中，会发生三个动作。首先，将基址添加到计算的偏移量中，以形成最终的线性地址。其次，将偏移与线段限制进行比较，以检查偏移是否指向线段内部。第三，检查段的访问权限（即，如果加载操作对段定义的内存区域具有读取权限）。

另一种可能的实现方式是将地址计算拆分为多个μops（例如，一个用于执行base+scaled index，另一个用于添加段base和执行限制检查）。在这种情况下，AGU变得更简单，因为它可以用一个简单的加法器实现，但内存操作会产生多个μOp，这可能会对性能产生一些影响：我们牺牲了一些问题带宽，并且我们不能将简单解码器（见第4章）用于某些加载操作。

####7.1.4 分支单位

分支单元负责执行控制流指令（分支、跳转和函数调用/返回）并生成正确的下一个指令地址（我们这里简称为程序计数器或PC）。

控制流指令可以是有条件的或无条件的。条件控制流指令（例如分支）根据测试结果更改程序流（例如，如果两个寄存器值相等或设置了条件代码）如果测试失败，则要执行的下一条指令是内存中的下一条指令。无条件控制流指令（如跳跃）总是会中断程序流。

控制流指令的目标PC可通过以下方式定义：

- 直接绝对：指令明确定义下一个PC值。
- 直接PC相对：指令将下一个PC值定义为与当前PC（即控制流指令的PC）的偏移量。
- 间接：该指令定义一个整数寄存器，其中包含下一个PC值。

因此，分支单位必须能够计算上述所有情况下的下一个PC。图7.4显示了分支单元的下一个PC计算部分。

7.1.5 浮点单位

该单元对来自浮点寄存器文件或内存的两个浮点值进行操作，并生成浮点结果。浮点单元（FPU）执行加法、减法和乘法等算术运算。根据实现的不同，它还可以执行除法、平方根和其他复杂运算（三角函数、指数等）。

通常，浮点和通用状态保存在单独的寄存器文件中。根据体系结构的不同，可能会有指令将浮点值转换为整数，反之亦然（并将转换后的值从一个寄存器文件传输到另一个寄存器文件）。转换操作也在浮点单元中实现。

IEEE 754-1985规定了四种表示浮点值的格式：（a）以32位编码值的单精度格式，（b）以64位编码值的双精度格式，（c） 单扩展精度，将值编码为43位或更多，（d）双扩展精度格式，将值编码为79位或更多。

实际上，大多数处理器在硬件中实现（a）和（b）。Intel®x86处理器也使用80位编码来实现（d）。浮点数的x86扩展名也称为x87指令集，因为它们首次出现在英特尔8087数学协处理器中。

FPU是一个非常复杂的单元，通常比整数单元大几倍。例如，在奔腾Pro上，FPU面积与2个AGU、1个ALU、1个IMUL和1个IDIV单元的总面积相同

第八章 提交阶段

8.1 介绍

当前大多数处理器都基于基于指令序列的执行模型，其中一条指令在前一条指令完成后立即执行。因此，处理器的行为就像是按照原始的顺序一个接一个地执行指令。然而，无论是按序处理器还是乱序处理器都不会在前一条指令完成后立即开始执行指令。当前的处理器是流水线的，因此它们在执行的不同阶段总是有多条指令在运行。因此，指令可以以不同于序列顺序的顺序修改处理器状态。例如，如果我们执行包含指令A和B的序列程序，其中A比B早，即使在顺序处理器中，如果A在其执行的最后阶段触发异常，B可能已经修改了一些寄存器值，如果它有时间到达写回阶段。在这种情况下，处理器将无法在A之后和B之前提供体系结构状态来处理异常。

为了模拟指令的顺序执行，现有处理器最常见的解决方案是在流水线末端实现一个称为提交的附加阶段。指令以原始程序顺序流经此阶段。然后， 在之前的流水线阶段所做的任何更改指令都被认为是推测性的，并且在到达提交阶段之前不会成为体系结构状态的一部分。此时，我们说指令提交。

在上一个示例中，A触发的异常将在A提交时处理。此外，由于指令是按顺序提交的，所以B还没有提交，因此其更改不会成为体系结构状态的一部分。通过这种方式，可以处理A产生的异常，就好像从未

执行过任何新于A的指令一样。我们说，如果处理器像执行触发异常的指令之前那样提供了正确的体系结构状态，那么它就支持精确的异常。

处理器以两种不同的状态运行：体系结构状态和推测状态。体系结构状态在提交时更新，就像处理器将按顺序执行指令一样。与之对比，推测状态意味着体系结构状态加上处理器中正在运行的指令所执行的修改。后一种状态之所以称为推测状态，是因为不能保证这些修改将成为体系结构状态的一部分。请注意，传统处理器依赖于推测性技术，如分支预测或推测性内存消歧，以保持指令的执行。因此，如果其中一些推测失败或发生异常，推测状态将无效，并且它永远不会变成体系结构状态。

与Alpha或MIPS等RISC处理器相比，一些CISC处理器（如最新的x86处理器（英特尔奔腾3、英特尔奔腾4、英特尔奔腾M或英特尔酷睿体系结构））将x86指令拆分为简单的微操作，以便于乱序引擎的实现。在这种情况下，当所有属于x86指令的微操作都已成功完成执行时，x86指令提交并更新处理器的体系结构状态。此规则的唯一例外是那些被拆分为大量微操作（如内存复制指令）的x86指令。在这种情况下，指令会在特定的执行点定期进行部分提交，因为x86语义可以接受这一点，详见《英特尔参考手册》。

最后，由于提交是指令执行的最后一个阶段，因此这是指令分配的执行硬件资源（如重排序缓冲区（ROB）项、内存顺序缓冲区（MOB）项或物理寄存器）被回收的地方。请注意，指令应该只回收那些不再使用的资源。因此，对于那些指令将其结果写入物理寄存器的配置，应该在我们确定不再需要寄存器的内容时完成该物理寄存器的回收。因此，在回收寄存器之前，我们需要确保将来可能需要该寄存器值的所有指令都已读取该寄存器，或者它们将能够从其他位置读取该寄存器。

在本章的下一节中，我们将介绍一些替代方案，以更新体系结构状态。此外，我们还解释了几种机制，以便在发生分支预测失效或异常时恢复推测状态并恢复执行。

8.2 体系结构状态管理

体系结构状态包括内存状态加上每个逻辑寄存器的值。

作为体系结构状态的一部分，在指令提交之前，无法修改内存。原因是我们无法将内存更新传播到系统的其余部分（设备、其他内核等），直到我们确定更新是正确的。因此，所有存储操作在提交之前不会更新内存状态。同时，它们与它们修改的内存地址、修改的大小以及提交时要存储在内存中的值一起驻留在存储缓冲区的条目中。有些处理器（如PA8000或MIPS R10000）不将数据存储在存储缓冲区中，但它们在提交时使用专用端口从寄存器堆中读取数据。

因此，所有加载操作都应检查存储缓冲区中是否存在更新其读取的内存空间的更旧的存储操作。如果加载找到与要读取的内存地址相匹配的旧存储，则加载应该从此存储操作而不是从缓存中获取数据，或者等待存储更新缓存。有关内存管理的更多详细信息，请参阅第6章。

有几种方法可以跟踪逻辑寄存器的最新体系结构状态。这些方法还取决于处理器上实现的分配方案。在本节中，我们将介绍两种方法：基于重排序缓冲区（ROB）和失效寄存器堆（RRF）的体系结构状态管理，如P6或Intel Core中所述，以及基于保存推测值和体系结构值的合并寄存器堆的管理。后一种方法用于英特尔奔腾4、Alpha 21264或MIPS R10000等处理器。

8.2.1 基于失效寄存器堆的体系结构状态

像P6这样的处理器实现了一个重排序缓冲区（ROB），其中的指令存储生成的值，直到它们失效并成为体系结构状态的一部分。然后，将这些值复制到一个寄存器堆中，其中包含与可用逻辑寄存器数量一样多的条目。该寄存器堆存储每个逻辑寄存器的体系结构状态，通常称为失效寄存器文件（RRF）。

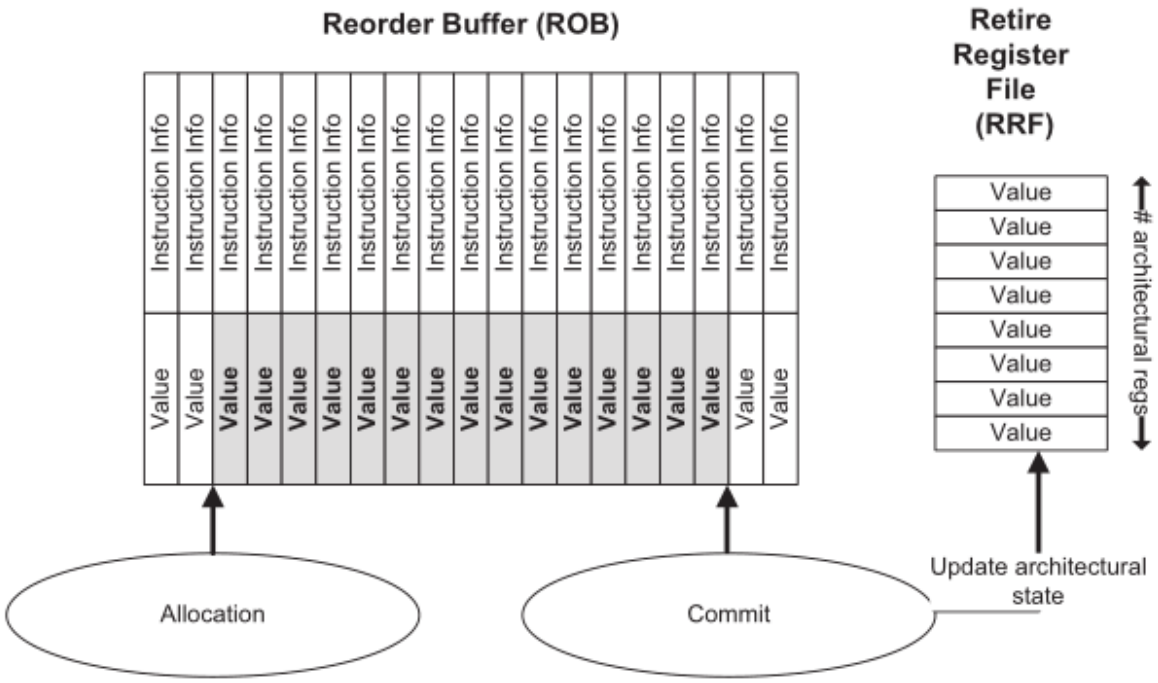

FIGURE 8.1: Reorder buffer and retire register file.

图8.1显示了该方案。ROB是一种循环FIFO，其中指令在分配阶段（又称重命名阶段）分配一个新条目，一旦指令提交或由于分支预测失误或异常等原因被撤销，就会回收该条目。然后ROB中的每个条目都包括两个主要部分：指令信息和生成值。指令信息包含有关指令类型、执行状态及其生成的体系结构寄存器标识的数据。另一方面，生成值字段保存指令生成的值。因此，执行中的指令产生的值（图中阴影部分）代表机器的推测状态。一旦指令提交，由该指令修改的体系结构寄存器将在RRF中使用ROB中存储的值进行更新，并回收其ROB条目。因此，RRF始终持有处理器体系结构状态的权威副本。

但是，请注意，此方案中存储值的位置在其生存期内有所不同。首先，该值存储在ROB中，以便消费者指令从那里读取。但是，一旦生成值的指令提交，就会回收其ROB条目，并将值写入RRF。当该值移动到RRF时，所有尚未读取该值的消费者都应该知道该值不再在ROB上可用，而是在RRF上可用。这一事实可能会使某些流水线阶段的实现复杂化，具体取决于微体系结构的特性，如下文所述。

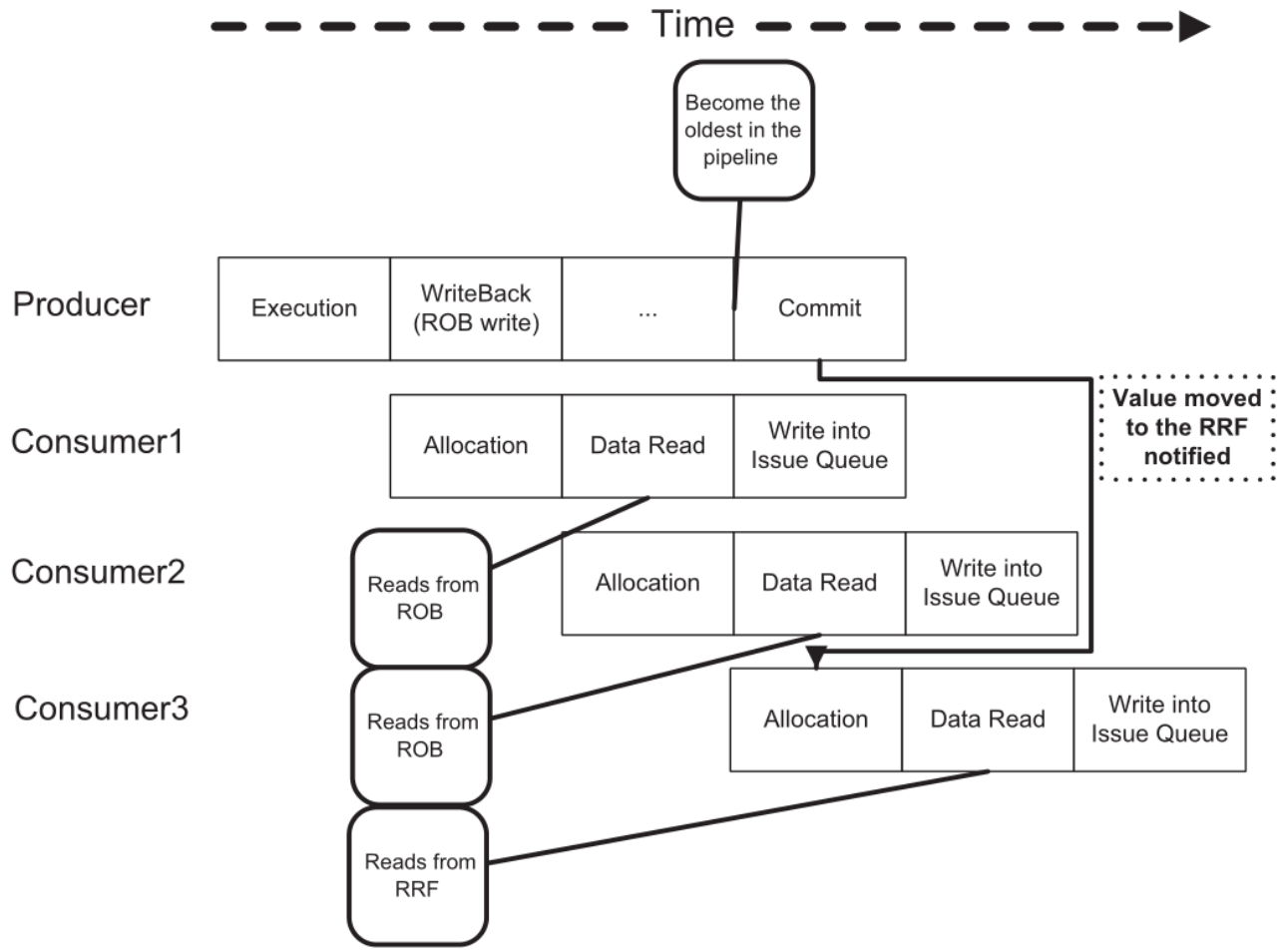

FIGURE 8.2: Notification of produced value transferred to the RRF.

大多数基于ROB的体系结构在发出指令之前读取操作数，就在重命名阶段之后。这种情况如图8.2所示。可以看出，所有消费者都读来自ROB的生成值，直到生产者指令提交。在此周期中，该值仍然驻留在ROB条目上，因为该条目尚未被重用。因此，此时读取其操作数的指令仍然可以从ROB获得它。但是，应通知分配阶段，以便更新重命名表并将此值标记为在RRF中可用。但请注意，只有当重命名表的任何条目仍指向此值时，才会受此更改的影响。一旦更新了重命名表，此后重命名的所有消费者指令都将知道该值在RRF中而不是ROB中可用。

然而，使用RRF实现基于ROB的体系结构，其中使用者在发射后读取其源操作数会使此通知复杂化。在这种情况下，提交逻辑不仅必须向分配阶段通知ROB条目的无效，还必须向位于分配和发射之间的流水线阶段中的所有指令通知ROB条目的无效，包括发射队列中的指令。然后，每个指示都应检查通知是否影响其源操作数，并相应更新其信息，以便在发射源操作数时从正确的位置读取其来源

8.2.2 基于合并寄存器堆的体系结构状态

像MIPS R10000、Alpha 21264或Intel Pentium 4等实现合并寄存器堆的处理器对属于体系结构状态的值和推测值使用相同的寄存器堆。基本上，一条指令分配一个物理寄存器来存储其产生的值，即使指令提交，该寄存器也会保留该值，直到不再需要它为止。与基于ROB的方案相比，该特性提供了三个主要优势。

值在提交时不会更改位置。因此，当值成为体系结构状态的一部分时，不需要通知尚未读取其源操作数的重命名表和正在运行的指令。这使得此方案适用于指令发出后读取其源操作数的处理器。

基于ROB的实现需要空间来容纳与ROB支持的执行中的指令数量一样多的生成值。然而，在典型应用程序中，大约25%的指令通常是存储操作和不产生任何值的分支。寄存器堆通过实现更节能的硬件结构来利用此特性，其中写入生成值的寄存器数量低于体系结构支持的执行中指令数量。

ROB是一种集中式结构，所有需要读取源操作数的指令都必须访问它。因此，如果在指令管理仍然集中的重命名阶段之后，指令读取其源操作数，则这种设计是合适的。然而，ROB会使解耦体系结构复杂化，其中指令在发射后读取其源操作数，像在英特尔奔腾4、MIPS R10000和Alpha 21264。这些处理器根据执行所需的资源类型，将指令引导到不同的执行集群。在这种情况下，不建议在分散的硬件（如执行集群）应该访问的地方实现集中式结构（如ROB）。基于寄存器堆的体系结构更适合后一种情况，即可以在每个执行集群上实现单独的寄存器堆。

第一个优点是有关一个事实：推测状态和体系结构状态共享相同的硬件结构。然而，其他两个优点来自这样一个事实，即我们使用物理寄存器堆来存储值，而不是ROB。因此，我们可以观察到使用其他方案跟踪体系结构和推测状态（如未来文件）的类似优势。

相比之下，与基于ROB的体系结构相比，与依赖寄存器堆的任何其他硬件方案一样，合并寄存器堆会使重命名复杂化。简而言之，基于ROB的体系结构利用了按顺序分配的ROB条目来存储生成的值的优势，而基于寄存器堆的体系结构则需要一个额外的重命名列表来存储可用的（物理）寄存器堆标识符。第5章对重命名阶段的影响进行了更全面的描述。

该方案的资源回收也更加复杂。虽然ROB条目在指令提交后立即按顺序回收，但合并寄存器堆无法回收任何物理寄存器，直到它保证不再需要它所持有的值。实现合并寄存器堆的处理器使用保守的方法回收物理寄存器。通常，当比A年轻的另一条指令B写入与A提交的相同的逻辑寄存器时，处理器会回收指令A分配的物理寄存器。

在下一节中，我们将描述恢复推测状态的方式，以及在发生分支预测失效或异常时恢复执行的方式。

8.3 推测状态的恢复

执行中的指令没有最终提交的原因之一是，由于分支预测失效或较早的指令引发异常，它们在错误的路径中获取。

在这两种情况中的任何一种情况下，都应该恢复推测状态，以撤消不会提交的指令重命名所产生的修改。接下来的两部分描述了在分支预测失效和异常情况下使用的典型恢复机制。这些方案取决于处理器设计是基于具有单独RRF的ROB还是基于合并寄存器堆。

8.3.1 从分支预测失误中恢复

在分支预测失误的情况下，机器的推测状态不正确，因为它从错误的路径提取、重命名和执行指令。因此，当我们识别分支预测失误，推测处理器状态和程序计数器应恢复到我们开始处理来自错误路径的指令的位置。

分支预测失误后的处理器恢复通常分为两个单独的任务：前端恢复和后端恢复。前端恢复通常比后端恢复简单。通常，恢复前端意味着刷新从错误路径提取的指令等待重命名的所有中间缓冲区，恢复转移预测器的历史记录，并更新程序计数器以恢复从正确路径提取指令。与之对比，恢复后端意味着删除位于任何缓冲区（如内存顺序缓冲区、发射队列、重排序缓冲区等）上属于错误路径的所有指令。

此外，还应恢复重命名表，以便从正确路径正确重命名指令。最后，还应该回收由错误路径指令分配的物理寄存器或发射队列条目等后端资源。

图8.3显示了假设的x86流水线在发生分支预测失误时的恢复过程。如前所述，前端的恢复时间早于后端，因此它可以尽早开始从正确的路径获取指令。然后，后端恢复与从正确路径提取第一条指令重叠。这些指令可以流经前端流水线，直到分配阶段。请注意，在重命名表完全恢复，并且回收错误路径指令分配的资源之前，分配阶段无法正确处理这些新指令。因此，如果前端流水线比后端恢复短，则在重命名阶段之前缓冲这些指令，直到后端恢复结束。

8.3.1.1 使用RRF在基于ROB的体系结构上处理分支预测失误

实现失效寄存器堆（RRF）的基于ROB的体系结构可以实现我们将为合并寄存器堆描述的任何恢复技术。然而，对于这些体系结构来说，实现类似于英特尔奔腾Pro的机制也是很常见的。

在英特尔奔腾Pro中遇到分支预测失误时，处理器不会恢复推测状态，直到提交了预测失误分支之前的所有指令以及此分支。此时，可以保证RRF中的体系结构状态表示执行预测失误分支后的应用程序状态。然后，通过使其所有条目指向RRF中的值来恢复分配阶段的重命名表，以便开始从正确路径重命名指令。

8.3.1.2 处理合并寄存器堆上的分支预测失误

实现合并寄存器堆的处理器通常不会等待预测失误的分支提交以恢复推测状态。

这些处理器记录重命名指令时重命名表的修改方式以及该指令分配的资源的日志。然后，如果发生分支预测失误，将遍历此日志，以便在重命名分支时恢复正确的状态。

此日志通常每个重命名指令有一个条目，其中每个条目都包含以下字段：指令覆盖的逻辑寄存器以及分配给此指令的物理寄存器标识符或分配给同一逻辑寄存器的前一个写入指令的物理寄存器标识符。日志包括分配给此指令的物理寄存器标识符或同一逻辑寄存器的前一个写入指令，具体取决于遍历是向前还是向后进行。

如果有许多指令需要执行，则遍历此日志可能需要很长时间。因此，像MIPS R10000或Alpha 21264这样的处理器依赖于检查点机制，以减少开始遍历的点与预测失误的分支之间的距离。这些处理器定期对重命名表的内容进行快照，这样就不必完全遍历日志，但遍历将从执行检查点的指令开始。

例如，对于MIPS R10000处理器，将比分支早的第一个检查点复制到重命名表中，并向后遍历重命名日志，直到找到预测失误的分支。日志上的每个条目都包括重命名指令重写的逻辑寄存器的先前映射。然后，将基于此信息还原重命名表，以反映在预测失误的分支重命名时它所具有的重命名映射。如果在预测失误的分支之前没有任何有效的检查点可用，则遍历将从最早的重命名指令开始，并更新现有的重命名表。

除了重命名表之外，还应更新其他信息，例如可用物理寄存器标识符的列表，以包括由错误路径中的指令分配的寄存器。一些处理器（如Alpha 21264）将空闲物理寄存器列表作为检查点的一部分实现。然后，恢复此列表，从检查点开始日志遍历，方法与重命名表相同。

8.3.2 从异常中恢复

异常通常在提交时处理。原因有两个：首先，我们需要确保触发异常的指令不是推测性的；例如，它不属于错误的路径。其次，我们需要提供体系结构状态，就像该指令之前的所有指令都是按原始顺序执行的一样。然后，将刷新所有正在运行的指令，因为应该在继续执行应用程序之前处理异常。此时，使用上一节中解释的机制之一恢复推测状态。最后，前端被重定向，以开始从异常处理程序获取指令。