# 第六章 发射阶段

## 6.1引言

流水线的发射阶段，负责向功能单元发射指令以执行。发射方案主要有两种类型：定序和变序。前者遵循程序的顺序，而后者则在源操作数可用时立即将指令控制变序。

通常，有序方案会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将其发出[46]。

但是，大多数最新的处理器都实现了变序调度方案。实现变序执行的方法有很多种。事实上，最终实现也在很大程度上取决于为其余阶段做出的设计决定。例如，如果在分派阶段之前或之后读取指令的源操作数，则需要不同的硬件。 此外，发射方案之间存在显著差异，具体取决于它们是基于保留站 [30]、分布式发射队列 [27，33，48] 还是集中式发射队列 [39]。

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 按顺序发出逻辑

按顺序发射逻辑以提取指令相同的顺序发出执行指令。因此，说明会一直等到所有以前的指令发出。然后，只要指令的源操作数可用，并且执行所需的资源准备就绪，就会发出指令。

这种发射逻辑有时在处理器的解码阶段实现，因为使用记分牌很简单。典型的记分牌包括两个表、一个数据依赖表和一个资源表。这些表可能因处理器的实际硬件限制而异。

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。此表上的每个表项都表示寄存器值的状态。此状态的范围为"不可用"，因此，在值可用之前，在某些旁路级别或写入寄存器文件中时，无法发出指令。读者将在第 7 章中找到有关旁路的更多详细信息。

资源表跟踪执行资源（如功能单元）的可用性。 有些功能单元（如除数）无法在每个周期接受一个新的操作请求。在这种情况下，如果处理器计划了以前一个周期使用它的另一个指令，则处理器无法安排使用除数的指令。因此，问题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） 处理器实现简化的按顺序问题逻辑。 这些处理器不实现任何类型的记分牌，因为它是生成代码的软件的责任，以调度每个指令，使其输入在发出执行时可用。该软件通常是静态编译器或共同设计的虚拟机，如在Transmeta Efficeon [47]。

## 6.3变序发射

**发射逻辑是确定乱序处理器能够利用的指令级并行性数量的关键组件**。一旦功能单元的源操作数可用，它就会通过向功能单元发出指令来乱序执行。但是，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中[1]。因此，实现一个高的复杂性-有效性比的发射逻辑，能够充分提高指令级并行性而不影响周期时间，这一点非常重要。（发射逻辑好坏会影响IPC）

有许多不同的替代方法来解决涉及发射逻辑实现的多个设计决策。但是，本章的目标不是对所有可能的实现进行广泛的描述，而是显示最常见的示例，以便给出硬件特性的概念。

在本章中，我们将介绍假集中式发射队列的两个主要方案。使用集中发射队列的处理器实现单个队列，其中存储所有重命名的指令，等待执行。这不同于其他方案，如保留站或分布式发射队列，其中指令在单独的缓冲区中分配，具体取决于执行指令所需的资源类型。

第一种情况表示处理器的发射逻辑的实现，其中指令在进入发射队列（如类似P6的体系结构）之前先读取其源操作数。然后，作为第二种情况，我们描述了实现发射逻辑所需的主要更改，在这些逻辑中，在像MIPS R10000或Alpha 21264这样发射要执行的源操作数之后对其进行读取。这两种情况适用于任何不同的现有方案，以保存指令产生的值（合并寄存器文件，重命名缓冲区，重排序缓冲区等）。

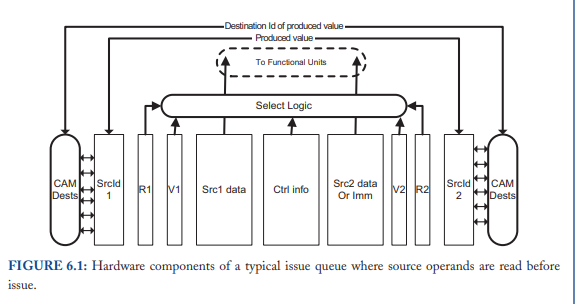
但是，由于这是一个正交的设计决策，为清楚起见，我们将为两种实现方式假定合并的寄存器文件。请注意，我们将合并的寄存器文件称为寄存器文件，该文件存储体系结构状态和推测值，如第5章中详细描述的。但是，所描述的硬件可以轻松地适用于任何其他寄存器文件方案。

本章还介绍了其他替代方法，例如分布式发射队列和保留站。由于上述方案已经涵盖了在实现中需要考虑的大多数折衷方案，因此将对这些替代方案进行更详细的说明。

最后，我们特别注意内存操作的发射逻辑的实现。与在重命名阶段检查数据相关性的其余操作相反，在存储操作计算其地址之前，无法识别存储相关性。正如我们稍后将描述的那样，此特征对这些指令的管理具有重要意义。（没法提前直到地址，进行数据依赖性检测）

## 6.3.1在发射之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的发射队列的主要特征是，它需要保存指令在执行时需要的信息以及已产生的源操作数的值。图6.1显示了用于存储此信息的典型组件的总览。图6.1中的每个块代表一个表，表中的表项与发射队列可以保存的指令数量一样多。此外，为简单起见，我们假设处理器具有类似于简化MIPS [32]的ISA，其中指令最多可以具有两个源操作数或一个源操作数，并且立即数值被编码为指令的一部分。



在我们的简单示例中，有一个名为Ctrl info的块。该块负责保存指令执行所需的所有静态信息（功能码，存储器操作的数据大小，立即数的使用等等）。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，并且已由重命名阶段生成，如第5章所述。如果由于指令对应的源操作数没有就绪，则该操作数的源ID无效。使用有效数组V1和V2标识无效的源操作数。这些块为每个表项使用1位实现，分别表示Src1Id和Src2Id是否有效。在此示例中，我们假定立即值始终为Src2。因此，在指令具有立即数的情况下，有效数组会将源标记为有效，但Src2Id将设置为0号寄存器或从未用于重命名目的的任何其他标识符。因为寄存器0通常被硬编码为0，所以它永远不是有用的目的地，因此0是一个很好的选择。

块Src1数据和Src2数据或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数据或Imm中。

最后，R1和R2的ready信号通知Src1数据和Src2数据或Imm是否已经就绪。一旦指令的所有源操作数就绪，就可以将指令转向执行。

引入主要组件后，我们将描述发射队列上发生的不同部件，以及该硬件结构如何与处理器的其余部分进行交互。这些部件是发射队列分配，指令唤醒，指令仲裁和发射队列回收。

## 6.3.1.1发射队列分配

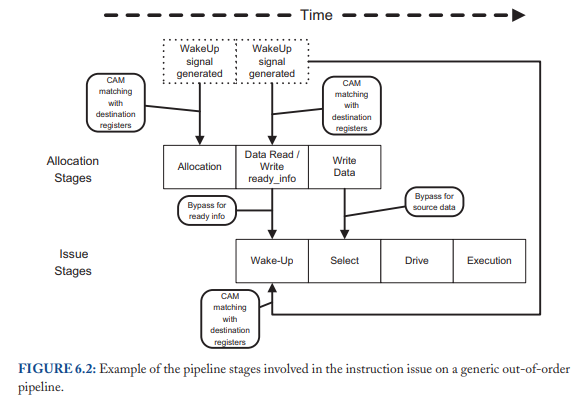
图6.2显示了发射队列在乱序处理器的通用流水中的集成示例。如我们所见，指令首先在重命名阶段（也就是分配阶段）被重命名，然后分派并输入发射队列。

如果没有可用的条目，则分派阶段将停止。请注意，由于时间限制，重命名阶段可能无法使用有关射布队列占用率的最新信息。在这种情况下，重命名可以对发射队列占用率做出保守的假设，以避免处理可能找不到发射队列条目可用的指令。（保留站满了，或者要满了，怎么通知前面的流水线？）

然后，在下一个周期中访问寄存器文件，以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，指示该源操作数是否已可用。因此，一条指令将必须从寄存器文件中读取操作数设置了可用位的输入。（会增加标记，标记哪些指令可以通过Bypass获得，可以不用读取寄存器组，这有助于减少寄存器端口竞争）

最后，与重命名的指令相关联的发射队列表项中填充了重命名信息以及从寄存器文件中读取的数据。



## 6.3.1.2指令唤醒

唤醒是通知一个操作数已准备就绪。该信号通常包括产生值的重命名ID，值本身和有效位。然后，CAM逻辑将此ID与SrcId1和SrcId2数组中的表项进行比较，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数据表项上。一旦设置了指令的有效源的就绪位，该指令就准备就绪（我们说它已被唤醒），并且可以被仲裁逻辑认为是要执行的指令。

请注意，唤醒信号仅产生一次，并且发生时，消费者指令可能不在发射队列中。因此，应该保证一旦产生一个值，它的所有使用者就会知道该数据已经可用。

这可以通过将重命名表上的上述valid位设置为1来完成。

在发射之前读取操作数的处理器通常在重命名阶段和在分派指令之间至少需要一个额外的周期，**如图6.2所示。注意，在此阶段，指令的源操作数也应与到来的唤醒ID进行比较，以防止死锁。**否则，如果他们一些的源操作数在此周期中产生， 这些指令永远不会唤醒，因为它们在前一个周期读取了available位，并且直到下一个周期才写入发射表项以执行CAM匹配。

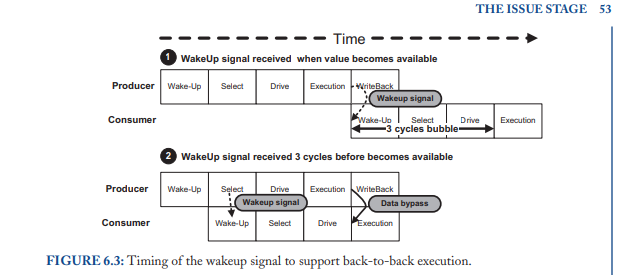
**可以减少需要CAM匹配的流水线级数量的可能解决方案是预先编写SrcId1和SrcId2。从重命名表获得此信息后**，**可以立即将其写入**。图6.2显示了在重命名之后在循环中写入此信息的情况。然后，发射队列将在唤醒部件和SrcId字段之间进行常规比较，以使ready bits在其表项发生变化时保持最新状态。从寄存器文件中读取值后，将相应地更新Src1数据和Src2数据。

另一种可以使我们不必为重命名和发射队列分配之间的所有阶段实施CAM匹配逻辑的替代方法是将availale bit实现为与重命名表分开的表(p-296姚 使用Score Board表，保存物理寄存器在一个声明周期中经过的地方)。在这种情况下，将在重命名阶段访问重命名表，但是直到指令将其SrcId分配到发射队列中之后，才可以访问availablebit表。

提前进行SrcId分配比其他方法有一些优势，因为它允许一条指令在读取源操作数的同一周期内唤醒。因此，可以考虑将整个数据写入发射队列的同一周期执行，如图6.2所示。

如前所述，唤醒信号通知一个值已经可用。但是，可以在实际产生该值之前生成此信号，以最小化生产者和消费者的执行之间的距离。（Bypass bit）可以提前通知唤醒信号，因为一条指令直到到达执行阶段（在指令唤醒之后的一个或多个周期）才需要其源操作数。例如，图6.3中的流水线实现在唤醒和执行之间的隔了两个阶段。

6.3显示了生产者和使用者执行之间的周期数，具体取决于发送唤醒信号的时刻。在方案1中，一旦执行生产者并且该值可用，就发送唤醒信号。然后，在生产者生成值之后，消费者将唤醒下一个周期并执行三个周期。如场景2所示，可以通过更早生成唤醒信号来避免这种三周期气泡。在这种情况下，在生产者生成该值之前的三个周期中发送唤醒信号。然后，消费者将在生产者生成值之后立即进入执行阶段，从而允许背对背执行。注意，在第二种情况下，该值应从功能单元的输出传输到另一个功能单元的输入。该连接称为旁路，其实现在第7章中进行了描述。此外，图6.3中所示的场景还假定可以在单个周期内完成生产者的选择和消费者的唤醒。该设计决策对于性能也至关重要，因为如果选择和唤醒不在一个周期完成，对于具有一个周期延迟的指令，不能执行背对背执行。导致性能显着下降[7]。总而言之，背对背执行对于性能至关重要[34]，因此，大多数处理器都将其实现[27,30,33,48]。（这里因该是一个周期）



有两种常见的实现方式来产生唤醒信号。一种选择是在指令执行完成之前的指令驻留三个周期的流水线阶段中生成信号（涉及到延迟广播）。注意，一条指令执行所需的周期数取决于其使用的功能单元。例如，整数加法器通常需要一个单周期才能完成，而整数乘法器或浮点功能单元可能需要更长的时间。因此，对于单周期操作，流水线应该能够从冲裁阶段生成唤醒信号（发射阶段，立即唤醒与之数据相关的指令），直到花费较长时间的功能单元结束三个周期为止

另一种选择是将有效位数组的每个表项、实现为移位寄存器加有效位。这些移位寄存器还可以实现为记分板，每个物理寄存器具有一个移位寄存器。每个移位寄存器的数值应与功能单元产生该值所需的最大周期数一样多。然后，总是在仲裁阶段产生唤醒信号（延迟唤醒），一位寄存器的值被设置为实际延迟周期数减去1.。移位寄存器每个周期都逻辑右移一位，直到最后的一位为1，这个有效信号被置位。

请注意，当指令的延迟是恒定的，并且仅取决于指令本身时，这些机制是合适的。此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Loads）的延迟取决于数据缓存或数据 TLB 中是否命中。遗憾的是，只有在发射Load、计算其地址并访问这些结构时，才会知道这一点。（这里老大坑了！，涉及到由于Load不命中导致它和后续的指令一直占用保留站，可能导致保留站饱和。另外访问cache不命中问题，需要设计相应的后续指令唤醒逻辑，推测的，还是确定性的，涉及到保留站重置问题。涉及到性能问题？）

可以通过延迟唤醒信号的生成，直到我们知道Load是否会命中高速缓存和TLB为止，以保守的方式处理Load。使用这种方法，如果访问cache命中，我们可以立即唤醒消费者-指令。如果发生未命中，我们将不会产生唤醒信号，直到解决了未命中。但是，程序中平均加载操作数约为20％[16]，并且大多数加载程序都有使用者（处于程序执相关链的开始处），**因此延迟这些操作的唤醒信号会对性能产生重大影响**。因此，一些处理器推测性地唤醒了Load结果的使用者，假设其命中了延迟时间，这种情况会付出代价在Load在访问cache不命中的情况下。推测唤醒将在本章稍后说明。

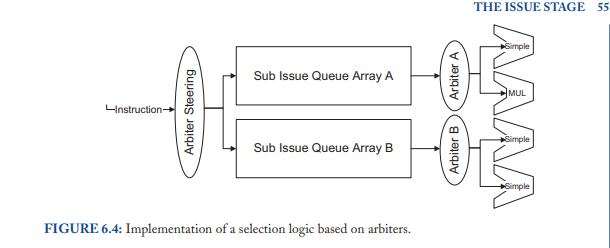
## 6.3.1.3 指令仲裁

仲裁逻辑（又名选择逻辑）负责在发射队列中选择将在给定周期中引导的就绪指令的子集。如果指令的源操作数已准备就绪且所需的执行资源可用（这个问题引出是在单个FU执行不同周期的指令，而且FU是否流水设计有关。为了避免同一个FU的两条指令同时执行完毕而导致的竞争公用总线），则可以仲裁指令。例如，如果处理器只实现一个乘法器，则不可能并行引导两个乘法指令。

仲裁逻辑的执行时间非常关键，因为它必须在唤醒逻辑之后完成，以支持单周期延迟操作的背对背执行。因此，处理器通常不实现单个仲裁逻辑，但会将其分发到称为仲裁器或调度程序的简单组件 [27，33，39]。例如，每个周期最多能引导 4 个指令的处理器将实现 2 或 4 个仲裁器，并且每个功能单元都将静态绑定到单个仲裁器。在发射队列上等待的指令也绑定到单个仲裁器。此配置允许并行化每个功能单元的仲裁逻辑。否则，仲裁器应同步，以保证它们不选择相同的指令，或者它们不会将两个不同的指令引导到同一个执行资源。

图 6.4 显示了基于仲裁器的选择逻辑的可能实现，如 Alpha 21264 [33]所示。在这个简化的示例中，我们只关注整数算术运算，最大问题宽度为 4，但最多只能是三个简单的算术运算和一个乘法。请注意，4 发射处理器通常随便发射4条指令，因为它们受实现的功能单元的约束（1、受FU实现方式2、受FU分配有关，例如一个FU可以执行3周期和2周期的指令，则不能背靠背执行，会产生总线竞争）。常见的处理器设计实现多个单元，这些单元执行简单的算术运算，但只有一个或几个单元支持复杂的算术运算。图 6.4 中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乘法器，而仲裁器 B 绑定到两个简单的功能单元。

在保留站里的指令也绑定到一个仲裁器。重命名指令后，它由仲裁器控制逻辑分配给仲裁器，并在指定仲裁器的子数组上分配一个表项。这种转向逻辑通常非常简单，并且发送指令到仲裁器，其功能单元可以执行它们，试图使每个仲裁器分配的指令量保持相同。



在每个周期中，每个仲裁器都会检查其子数组以获取可用的就绪指令，并选择要执行的指令。对于Alpha 21264，当子数组中就绪指令的数量超过发出宽度时，最早的指令始终具有优先权（older-first）。该算法在Alpha 21264之类的处理器中相对容易实现，在该处理器中，指令始终按顺序存储在保留站中。然而，其他处理器设计（例如MIPS或Intel P4）中的指令顺序未保留在发射队列中，它们实现了基于伪（older-first）算法或仅基于发射队列位置（position-based）进行优先级排序[48]。

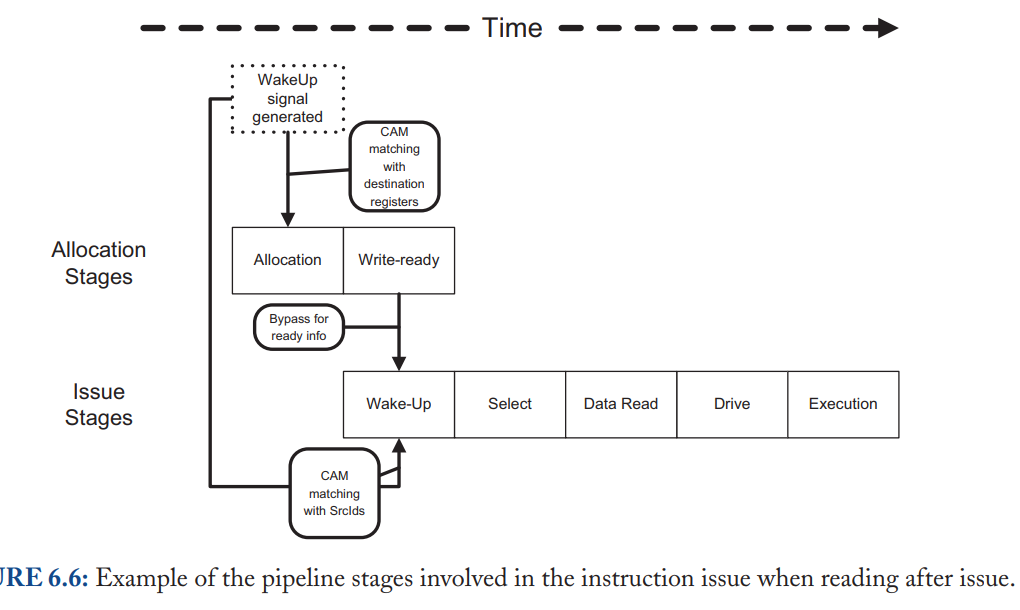
## 6.3.1.4表项回收

一旦选择了一条指令并将其数据转发到功能单元，就可以安全地收回其发射队列表项了。但是，诸如推测性唤醒之类的某些技术可能需要延迟回收，直到我们确定指令可以执行为止。（对于可能发生异常或者反例的指令，MIPS中怎么处理？不在保留站中保留指令，发生异常的时候重新取值还是保留站中的表现不清掉）（这里涉及怎么保留、怎么清除保留站表项的问题？）推测性唤醒通常用于减少加载操作与其使用者之间的等待时间。第5.2.4节介绍了此技术。

## 6.3.2发射之后读取源操作数的发射过程

本节描述了发出指令后读取源操作数的方案与先前方案之间的主要区别。发射后读取和发射前读取之间的主要区别在于发射后读取不需要发射队列来存储源操作数值。因此，唤醒信号不需要转发这些值。图6.5显示了在发射之前和发射之后读取数据时发射队列的组件之间的差异。他们中的灰色部分如果我们发射后再读，则不需要。请注意，即使我们完全删除了Src1数据，Src2数据仍然保留，尽管可能会更小。**原因是该方案仍然需要空间来保存立即值。**但是，这些值通常比寄存器值短。此外，几乎没有指令需要此字段，因此可以将其实现为具有很少表项的单独结构。在后一种实现中，需要立即数的发射队列表项将在该结构中包含立即数的偏移量。图6.6显示了此发射方案的可能流水线。**可以看出，由于尚未读取数据，因此重命名阶段与发射队列分配之间的阶段数减少了。这意味着减少了所需的CAM匹配逻辑的数量**

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了一个周期。（读取寄存器阶段）



## 6.3.2.1减少读端口

寄存器组的面积，功耗和访问等待时间随读取端口的数量而增加[7]**（面积是读端口数的平方还是啥来的？）**。因此，为了实现节能设计，最小化端口数量非常重要。

在最坏的情况下，某些处理器设计会根据需要实现尽可能多的读取端口，在这种情况下，发射宽度已得到充分利用，所有指令都从寄存器文件中读取其操作数[29,33,48]。但是，Alpha 21264将其分成两个复制的物理寄存器文件，每个寄存器的读取端口总数减半，从而减少了每个寄存器文件的读取端口数量。该解决方案减少了读取端口的数量，但代价是原本具在一个周期延迟的指令不能背靠背执行由于访问不同的寄存器组。

在文献[7]中已经表明，大多数源都是从旁路网络读取的，而不是从寄存器文件读取的。而且，发射宽度通常未得到充分利用。因此，也有可能用比最坏情况下所需端口少的端口来实现寄存器文件，并且对性能的影响最小。

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，以便计算每个所选指令将使用的读取端口数。如果所需的读取端口数超过可用的读取端口数，则某些仲裁器将取消发射过程。（更影响性能）请注意，将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此逻辑的延迟。因此，实现此读取端口的同步可能会对延迟产生影响。

被动式替代方法使仲裁器发射指令，假设永远不会达到读取端口的总数，并且在极少数情况下会做出反应。在这种替代方案中，在发射指令时分配读取端口。

然后，如果超出了可用读取端口的数量，则应取消某些已发射的指令，然后重新发射。可以使用在执行推测性唤醒的处理器中实现的任何替代方法来完成取消和重新发射。其中一些技术在5.3中进行了描述。最后，请注意，最后，请注意，反应机制可能会导致仲裁器“饥饿”甚至活锁。因此，定义公平的政策以执行取消很重要。（取消谁的问题呗？）

## 6.3.3变序发射的其他方案

对先前方案的讨论涵盖了在执行问题逻辑时需要解决的大多数主要设计决策。但是，为了完整起见，我们在本节中简要描述了可以在现代处理器中找到的两个其他问题实现：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配到执行集群中，其中每个集群都实现自己的发射队列。例如，英特尔奔腾4实现了两个带有私有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，根据操作类型将指令引导到一个发射队列，然后将其绑定到发射队列内的特定调度程序。

6.3.3.2保留站

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

实现此方案的处理器将重命名之后的指令分配到保留站，并在那里等待操作数，直到其输入值变为有效。然后，每条指令将其产生的值广播到所有功能单元的所有保留站。可以看出，此技术需要一些设计决策，类似于发射队列方案，其中数据在发出之前读取。

## 6.4对于内存访问类指令的发射逻辑

内存操作具有数据依赖性，在重命名阶段无法识别。仅在发出这些指令并计算了它们的地址后，才能检查这些内存相关性。

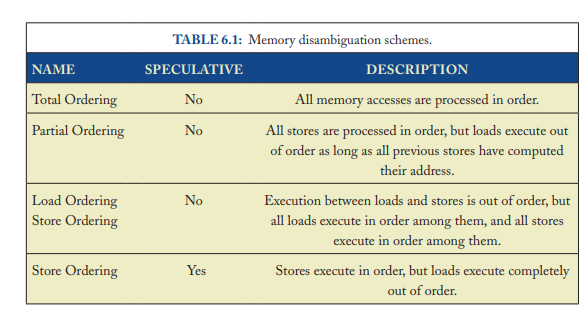
负责处理内存依赖性的机制称为内存消歧策略。不同的处理器实现完全不同的内存消歧策略。表6.1显示了在不同的微体系结构上实现的一些典型方案。

这些方案可以分为两大类：非推测性和推测性消歧策略。第一组不允许执行存储操作，直到我们确定它与先前的任何存储操作都不相关为止。相比之下，第二组预测一个存储操作是否与另一个进行中的存储操作具有依赖性。

选择合适的存储器消除歧义对于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约有30％是存储器操作。因此，实施非常保守的内存消除歧义策略可能会产生不必要的执行序列化，从而极大地限制了可以利用的指令级并行性。另一方面，非常激进的内存歧义消除策略可能最终会导致复杂的恢复机制，并且由于错误推测会大大增加功耗。

## 6.4.1 非推测内存歧义

在以前所有Store都计算其地址之前，非推理内存消除歧义策略不会发出任何Load操作。因此，可以安全地计算内存依赖性。 现有处理器实现的非推测内存消除歧义策略有三种主要类型：定序、部分乱序（Load或者Store乱序）。



总的来说，所有内存操作都按顺序执行。如今，据我们所知，没有乱序处理器可以实现全定序，因为它限制了我们可以利用的大量并行性。

相反，其余的非推测性方案允许Load操作相对于Store乱序执行。在使用Store定序Load变序的情况下，Load按顺序进行，而Store变序进行。但是，Load不必等待先前的存储访问缓存。我们可以找到在AMD K6等处理器中实现的该方案。但是，在部分排序中，Load可以乱序处理。在这种情况下，只要Load已准备好其源操作数并且所有先前的Store都已经计算了其地址，就可以发射Load。实现部分按序的处理器示例包括MIPS R10000和AMD K8。

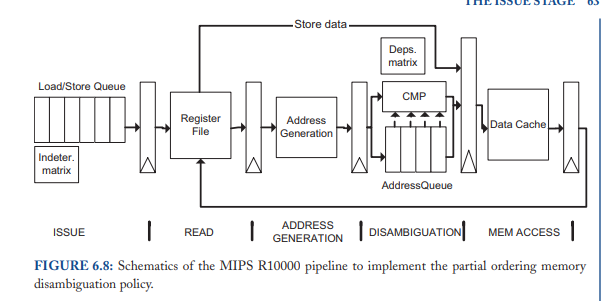
注意，一旦计算出存储器的存储器地址，就可以执行存储器歧义消除。因此，一些处理器将存储操作分为两个子任务：一个子任务计算地址，另一个子任务获取数据。然后，存储操作不必等待数据的生成者完成即可计算其地址。

在某些情况下，像HP PA8000 [29]一样，即使处理器也会尽快计算地址，并且直到存储成为最古老的运行指令后才读取数据。

在下一部分中，我们将对现有处理器中实现的非推测性内存消除歧义策略进行案例研究。示例显示了在MIPS R10000上部分排序的实现。

6.4.1.1案例研究：在MIPS R10000处理器上的部分变序。

在MIPS R10000处理器实现部分变序。因此，只要所有先前的内存操作都已计算了其地址，就可以无序执行Load。相反，Store以严格的程序顺序进行处理。消除内存歧义的流水线阶段和组件的示意图如图6.8所示。此内存流水实现以下组件以执行歧义消除：



**加载/存储队列**：这是一个 16 项队列，在重命名阶段后按顺序存储Load和Store指令。指令不会离开此队列，直到其源操作数准备就绪。

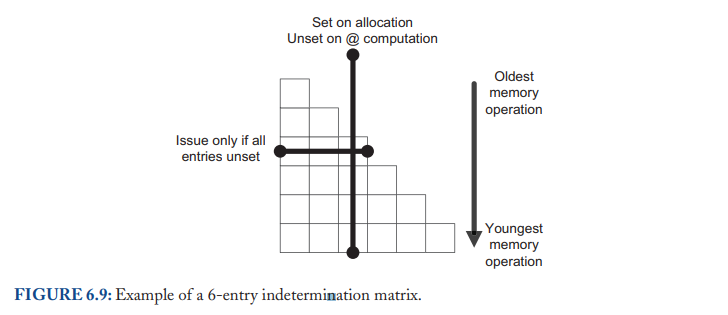
**不确定矩阵**：这是一个 16x16 矩阵的一半，其中每列和行表示load/store队列上的表项。内存操作将列上的所有表项设1，并在计算其地址时重置它们。然后，当其行的任何位置为1标识其属于较旧的内存操作时，无法发射内存操作 图 6.9 显示了 6 表项的load/store队列的确定矩阵示例。

**依赖项矩阵**：这是一个 16x16 矩阵，其中每列和行表示load/store队列上的表项。一条load依赖于以前的stores将把所有Stores列存在依赖的store行设置为1。然后，在重置**行**上的所有表项之前，它将无法恢复其执行。相比之下，store在更新内存时重置其**列上**的所有bits。图 6.10 显示了 6 个表项的load/store队列的依赖项矩阵示例。

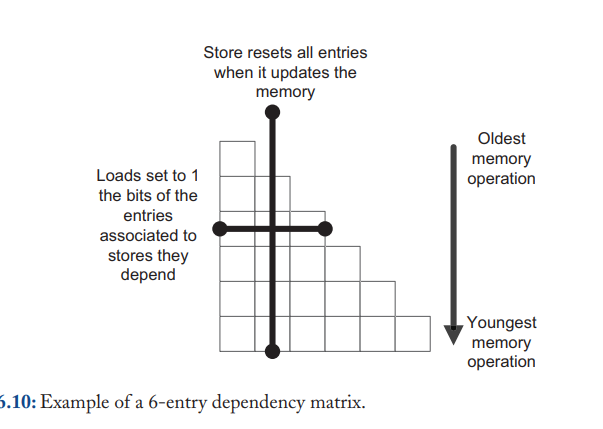
**地址生成：**这是根据源操作数计算机其访问memory地址的地址转换逻辑。

**地址队列：**此队列保留要访问缓存的load和store的地址。在loads的情况下，除了在此队列上写入其地址外，它们还会将其与所有以前存储的地址进行比较，如果匹配，**则设置依赖项矩阵上的相应表项。**

在重命名阶段，所有存储操作都会激活不确定矩阵上其**列**的位。请注意，**与指令关联的列、取决于其在load/store队列中的位置**。然后，只要它在不确定矩阵上的行不包含任何一个（所有先前的存储指令都已计算出其地址）并且其源操作数可用，就发出存储操作并在读取阶段读取其源操作数。（它这意思是检测所有的相关性，没有相关性、并且源操作数可用，就发射）（**但是这个矩阵的更新怎么实现？）**



一旦内存操作计算出其地址，它就会存储在地址队列中。如果是store，它将一直驻留在那里，直到它成pipeline中最古老的指令为止。然后，将发出重置命令以重置其在依赖矩阵上的列，并使用专用读取端口从寄存器组中读取要存储在存储器中的数据。请注意，由于存储是流水线中最古老的指令，因此所有数据都已计算出来，因此所有先前的指令均已产生结果并已淘汰。相比之下，load会将其地址与先前存储的地址进行比较，并相应地更新其在依赖矩阵上的行。load将在地址队列上等待，直到其在该矩阵上的行完全重置。



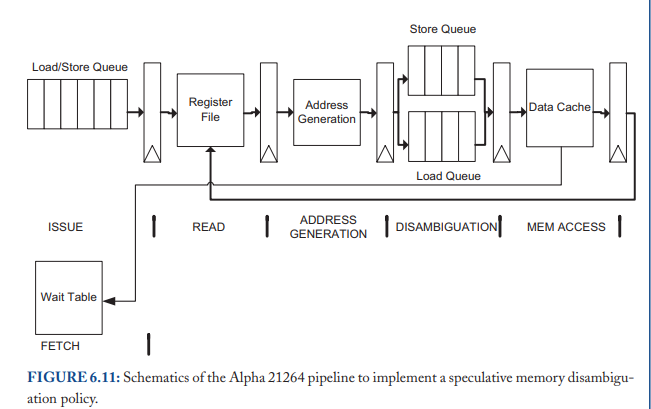
MIPS R10000实现了2路数据高速缓存，其中指令的访问是乱序的。在发生高速缓存未命中的情况下，该指令必须等待，直到未命中得到解决为止，但是其他存储器指令可以同时进行。这种情况可能会导致不希望发生的情况，称为抖动。当在高速缓存上发生内存操作未命中时，以及在未命中得到解决且在读取数据之前，另一条指令已再次逐出该行时，就会发生触发。请注意，如果流水线影响流水线中最早的指令，则可能会导致流水线跳动。 MIPS R10000通过锁定流水线中最早的内存操作将访问的集合的方式来避免浪费，直到成功读取数据为止。最古老的指令在歧义消除阶段锁定其设置的方式。

最后，如前所述，一旦允许访问地址离开地址队列，就立即加载并存储访问内存。

## 6.4.2推测性内存消除

一些最新的处理器设计（例如Alpha 21264或Intel Core架构）已经实现了推测性内存歧义消除。这些处理器通过推测性地发射load而不依赖于任何先前执行的store来提高性能。因此加载操作不必等待所有以前的存储来计算其地址。请注意，由于此方案推测内存依赖性，因此我们可能发生错误预测，最终导致应用程序执行不正确。因此，这些处理器需要特殊的硬件来识别这些错误预测和恢复执行。

我们将 Alpha 21264 将描述实现方案。



## 6.4.2.1 案例研究：阿尔法 21264。

此内存流水线实现以下组件，以便执行消除歧义

**Load/Store队列**：此队列保留内存操作，直到它们计算其源操作数，并可以发射以执行。

**Load队列：**此队列按程序顺序存储load的**物理地址**。Load在重命名阶段分配此队列上的表项，并在停用时回收该表项。 此队列实现 32 个 CAM 表项。

**Store 队列**：此队列按程序顺序存store指令的物理地址和数据。store在重命名阶段分配表项，直到停用才会回收表项。此结构还实现 32 个 CAM 表项。

**Wait Table：**此表实现由虚拟 pc 索引的 1 bit 的1024 个表项。每当我们确定load依赖于store发生时，一个store-load order trap 将会生成，并且此表将表示load的虚拟地址的表项设置为 1。提取单元读取此表，以便在load过去产生访存为例时标记load。 然后，这些load将不会再推测发射。但是，等待表每 16，384 个周期重置一次，如果不这样，最后这张表将被填满。

Load和Store在load/sotre队列上等待，直到它们的源操作数准备就绪。就store而言，我们等待计算地址和数据所需的源操作数，这与以前的案例研究相反。如果load将其wait bit位置1，则在所有先前的store都已发出之前，load将不会离开load/store队列。

然后，在下一个周期读取用于计算地址的源操作数，并在一个周期后计算访存地址。计算出其地址的load会将其保留在load队列中。

已经计算出地址的load将其保留在load队列中。此外，它们将其地址与较年轻的load的地址进行比较，如果它们匹配，则触发load-load 内存冲突trap。此trap使处理器从触发trap的load开始恢复执行。如果不需要trap，load将继续访问cache。

在内存消除歧义阶段，store还会在store队列表项上写入其内存地址。此外，他们检查load队列，寻找与其地址匹配的年轻load。如果发生这种情况，, a store-load memory violation trap occurs，并从load处恢复执行。此外，将更新wait table，以为了标记将来的这种load例子，并避免这种情况再次发生。请注意，store不会检查store-store内存访问是否存在冲突，因为即使store变序发射，它们也不会在提交之前更新cache。**由于提交是按程序顺序提交的，因此永远不会发生store-store内存冲突。**

## 6.5 load指令的推测唤醒

load操作的延迟是可变的，主要取决于load是否命中 TLB 和cache。还有其他因素可能会影响最终数据可用的周期，例如，data cache上的bank冲突、读端口与 MOB 中的其他内存操作冲突等。但是，最常见的情况是load访问cache不命中造成的延迟，如图 6.12 所示。在此图中，我们可以看到两种可能的情况，其中消费者指令读取load生成的值。可以看出，laod是否命中会在仲裁后的三个周期后产生。因此，如果我们实现一个保守的唤醒，即消费者只有在保证load将达到时才会被唤醒，我们将获得生产者和消费者之间的双周期气泡，如场景 1 所示。但是，如果我们推测触发唤醒信号，假设load将像场景 2 中那样达到，我们也将能够实现load操作的背对背执行。

但是，在缓存中load未命中或由于其他原因延迟执行的不常见情况下，消费者指令将不得不取消并重新发射。

一旦指令离开发射队列，就不能保证此指令可能在此队列上有一个空槽使其能够返回。发射队列可能会有与已经发射指令存在数据依赖关系的指令。因此，发射指令在等待发射队列上的空闲插槽以重新发射，这样可能会死锁。

有几个解决方案可以避免这种死锁，具有不同的优缺点。一种解决方案是刷新流水线中比重新发出的指令更年轻的所有指令，然后重新从这里执行。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推测中恢复。此方案的主要缺点是，如果这种情况经常发生，我们可能会出现显著的性能下降。这是 Alpha21264 实现等wait table的原因之一。

另一个降低指令重新发射性能损失的解决方案是推迟重新使用指令分配的发射表项的回收，直到我们确定不必重新发射此指令。在这种情况下，每个发射队列表项都有一个bit（issued bit），说明是否已发出此指令。然后，仲裁逻辑不考虑已发出的指令。但是，每当需要重新发出指令时，都会重置受影响的表项的issued bit，以便仲裁逻辑再次考虑这些指令以执行。与上一个机制相比，此机制减少了重新发射指令的惩罚代价，但增加了发射队列的压力。请注意，所有已发射指令分配的发射队列表项不能用于查找进一步的独立指令。

不幸的是，发射队列表项的数量通常很小，因此由于发射的指令对表项的占用，像上一个解决方案那样的解决方法可能会降低性能。 netburst体系结构实现了小发射队列，以使其适应紧凑的周期时间。此外，由于它实现了深层次的流水线，指令的周期数应该留在发射队列中，因为它们在我们确定不会被重新发行之前已经发行了，所以数量非常多。因此，netburst体系结构（如P4）实现了重发射队列。在这种情况下，指令一经执行就离开了发射队列（或调度程序），但它们在称为冲发射队列的其他fifo结构中排队。然后，指令驻留在此队列中，直到确保不再需要重新发出指令为止。但是，如果需要重新发出指令，则调度程序将优先级赋予重播队列，以便按照在此处分配指令的顺序重新发出其指令。