第9章　流量控制

流量控制（Flow Control）的概念起源于网络通信。一个复杂的网络系统由各类设备（如交换机、路由器、核心网），与这些设备之间的连接通路组成。从数据传输的角度来看，整个网络中具有两类资源，一类是数据通路，另一类是数据缓冲。

数据通路是网络上最珍贵的资源，直接决定了数据链路可能的最大带宽；而数据缓冲是另外一个重要的资源。当一个网络上的设备从一点传送到另外一点时，需要通过网络上的若干节点才能最终到达目的地。在这些网络节点中含有缓冲区，暂存在这个节点中没有处理完毕的报文。网络设备使用这些数据缓冲，可以搭建数据传送的传送流水线，从而提高数据传送性能。最初在网络设备中只为一条链路提供了一个缓冲区，如图9-1所示。

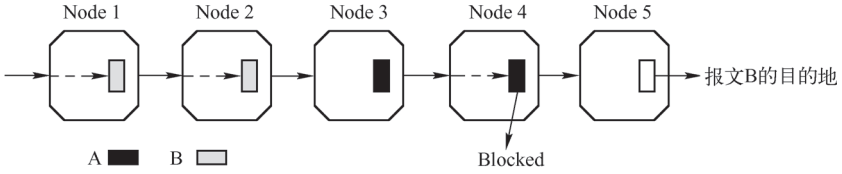

图9-1　基于单数据通路的数据传递

当网络设备使用单数据通路进行数据传递时，假设在该通路中正在传递两个数据报文，分别是A和B。其中数据报文B需要经过Node1～5才能到达最终的目的地，而数据报文B在经过Node 3时发现由于Node 3正在向Node 4发送一个数据报文A。从而数据报文B到达Node 3后，由于Node 4的接收缓存被数据报文A占用，而无法继续传递。此时虽然在整个数据通路中，Node 4和Node 5之间的通路是空闲的，但是报文B还是无法通过Node 3和4，因为在Node 4中只有一个数据缓冲，而这个数据缓冲正在被报文A使用。

使用这种数据传送规则会因为一个节点的数据缓冲被占用，而影响了后继报文的数据传递。为了解决这个问题，在现代网络节点中设置了多个虚通路VC，不同的数据报文可以使用不同的通路进行传递。从而有效解决了单数据通路带来的问题，基于多通路的数据传递如图9-2所示。

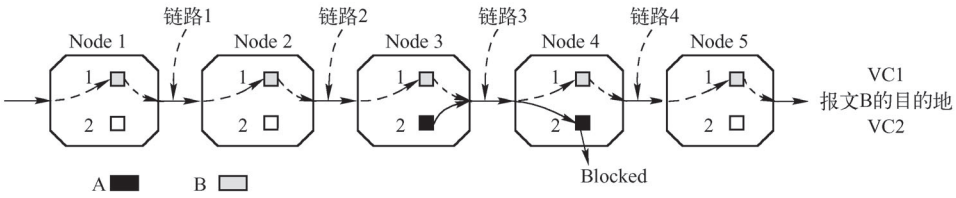

图9-2　基于双数据通路的数据传递

如上图所示，所谓多通路是指在每一个节点内设置两个以上的缓存。上例中设置了两个缓存，报文A经过Node 1～5时使用缓存2进行缓存，然后进行数据传递，而报文B使用缓存1进行缓存。因此虽然报文A因为某种原因不能继续传递，也只是将报文阻塞在缓存2中，而不影响报文B的数据传递。

所谓 VC 是指缓存到缓存之间的传递通路。如图 9-2 所示的例子中含有两个 VC，分别是 VC1 和 VC2。其中 VC1 对应节点间缓存 1 到缓存 1 的数据传递，而 VC2 对应缓存 2 到缓存 2 的数据传递。VC 间的数据传递，如 Node 1 的缓存 1/2 到 Node 2 的缓存 1/2，都要使用实际的物理链路“链路 1”，这也是将 VC 称为“虚”通路的主要原因。

在一个实际的系统中，虚通路的使用需要遵循一定的规则。如在 PCIe 总线中，将不同的数据报文根据 TC 分为 8 类，并约定这些 TC 报文可以使用哪些 VC 进行数据传递。在 PCIe 总线中使用 TC/VC 的映射表，决定 TC 与 VC 的对应关系。

PCIe 总线规定同一类型的 TC 报文只能与一条 VC 对应，当然从理论上讲，不同的 TC 报文可以与不同的 VC 对应，也可以实现一种自适应的算法根据实际情况实现 TC 报文和 VC 的对应关系。只是使用这种方法需要付出额外的硬件代价，效果也不一定明显。

下文以图 9-2 所示的数据传递为例，进一步对此说明相同类型的报文使用不同 VC 的情况。假设报文 A 和 B 属于相同种类的报文，但是报文 A 使用 VC1，而报文 B 使用 VC2。首先报文 A 传递到 Node 4 后被阻塞。而报文 B 使用的 VC 和报文 A 使用的 VC 不同，报文 B 最终也会到达 Node 4。

由于报文 A 和报文 B 属于相同类型的报文，Node 4 阻塞报文 B 的概率非常大，因为 Node 4 已经阻塞了报文 A。阻塞报文 A 的原因在很大概率上也会对报文 B 适用。此时两个虚通路都被同一种类型的报文阻塞，其他报文将无法通过。因此在实际应用中，相同类型的数据报文多使用同一个 VC 进行数据传递，而在 PCIe 总线中，一个 TC 只能对应一个 VC。

目前多通路技术的应用已经普遍应用到网络传输中，虚通路是一种防止节点拥塞的有效方法。但是在网络传送中，还存在一种不可避免的拥塞现象，即某一条链路或者某个节点是整个系统的瓶颈。

我们假设图 9-2 中 Node 4 将报文转发到 Node 5 的速度低于 Node 3 发送报文的速度。在这种情况下，Node 4 将成为整个传送路径上的瓶颈，无论 Node 4 中的缓存 1 和 2 有多大，总会被填满，从而造成节点拥塞。

当缓存填满后，如果 Node 3 继续向 Node 4 发送报文时，Node 4 将丢弃这些报文，之后 Node 3 将会择时重发这个报文，而 Node 4 仍然会继续丢弃这个报文，这种重复丢弃的行为将极大降低网络带宽的利用率，而且 Node 3 也会成为网络中新的瓶颈，从而引发连锁反应，造成整个网络的拥塞。为了避免这类事件发生，网络中的各个组成部件需要对数据传送进行一定的流量控制，合理地接收和发送报文。

如上文所述，在网络中有两类资源，一类是数据通路，另一类是数据缓冲。而流量控制的作用是合理地管理这两类资源，使这些资源能够被有效利用。

9.1 流量控制的基本原理

目前流量控制从理论到实现大多基于多通道技术，本节也仅讨论基于多通道的流量控制（FCVC，Flow-Controlled Virtual Channels）的基本原理。

流量控制的主要作用是在发送端和接收端进行数据传递时，合理地使用物理链路，避免因为接收端缓冲区容量不足而丢弃来自发送端的数据，从而要求发送端重新发送已经发送过的报文，并最终有效地利用网络带宽。

目前几乎所有流量控制算法的核心都是根据接收端缓冲区的容量，向发送端提供反馈。而发送端根据这个反馈，决定向接收端发送多少数据。这些流量控制算法都力求发送的每一个数据报文都能够被接收端正确接收，而不会被接收端因为缓冲不足而丢弃。使用流量控制机制并不能提高网络的峰值带宽，相反还会降低网络的带宽，但是可以有效减少数据报文的重新发送，从而保证网络带宽被充分利用。

流量控制的目标是在充分利用网络带宽的前提下，尽量减少数据报文因为接收端缓存容量不足而被丢弃的情况，因为此时数据发送端将会择时重新传送这些丢弃的报文，从而降低了数据通路的利用率。

为了实现流量控制，数据接收端需要及时地向发送端反馈一些信息，发送端依此决定，是否能够向接收端继续发送数据。这些反馈信息也需要占用一定的数据通路带宽。但是采用这种方法可以有效避免数据报文的丢失与重发，从而在整体上提高了数据通路的利用率。流量控制针对端到端的数据传递，目前流行的流量控制方法共有两种，分别为 Rate-Based 机制和 Credit-Based 机制。

9.1.1 Rate-Based 流量控制

Rated-Based 机制适合“可预知带宽”的数据传递方式，而 Credit-Based 机制更加适合“突发数据传送”。下面将以图 9-3 所示的实例简单介绍 Rate-Based 机制。

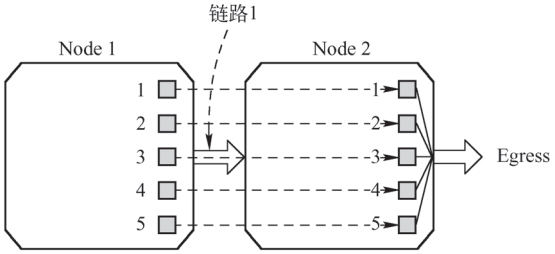

图 9-3 Rate-based 流量控制机制

假设 Node 1 与 Node 2 之间共存在 5 个 VC，即在链路 1 的两端设置了 5 组缓存。而这 5 个 VC 共享一个物理通路，即链路 1。为方便起见，假设链路 1 的带宽为 1，而在系统初始化时，将 VC1～5 可以使用的带宽 BCVC 都设为 1/4（即 Rate 值为 1/4），而且每一个 VC 使用的最大数据传送率不能超过 BCVC。

在某些情况下，由于在 Node 1 中，每个 VC 的 BCVC 最大值为 1/4，因此 Node 1 可以向 Node 2 发送的数据带宽总和为 5/4，大于链路 1 所能提供的峰值带宽，因此链路 1 将可能成为瓶颈，从而造成网络拥塞。此时来自 Node 1 的数据报文必然会阻塞在各自 VC 的发送缓冲中，并可能出现报文重传现象。

Rete-Based 机制使用“自适应”调解的办法有效防止了这种网络拥塞。Rate-Based 机制规定每一个 VC 在发送一定数量的报文后，将主动地将相应 VC 的 Rate 调整为 Rate 减去 ADR（Additive Decrease Rate），直到 Rate 等于 MCR（Minimum Cell Rate）[⊖]；当 Node 2 的

⊖ 在本节的实例中，5 × MCR ≤ 1。即所有 VC 使用的 MCR 之和小于或等于链路总带宽。

Egress端口并不拥塞时，Node 2 将向 Node 1 的对应 VC 发出正向反馈，通知该 VC 可以适当提高数据传送率，当 Node 1 的 VC 收到这个正向反馈后，将更新其 BCVC。

假设在本例中 MCR 为 x，当链路 1 严重拥塞时，Node 2 不会向 Node 1 的所有 VC 发出正向反馈，最终 Node 1 所有 VC 的 Rate 都将降为 MCR，此时 Node 1 将不会向 Node 2 发送过多的数据报文；当链路 1 并不拥塞时，Node 2 将向 Node 1 的相应 VC 发出正反馈，通知 Node 1 可以适当提高数据报文的数据传送率。

Rate-Based 流量控制机制可以使用漏桶（Leaky Bucket）算法或者令牌桶（Token Bucket）算法实现。使用令牌桶算法时，一个设备至少具有 MCR 个令牌，这个设备每发送一定数量的报文后，将令牌减少 ADR 个，但是总令牌数不低于 MCR。当这个设备收到下游设备的正反馈时，将增加令牌数。

采用 Rate-Based 流量控制机制可以有效解决“可预知带宽”的数据传递。比如 Node 1 向 Node 2 发送音频或者视频数据，这些音视频数据占用的数据带宽基本恒定，因此使用这种方法可以保证这类数据报文的流畅传递。

而对于多数长度“不可预知”的突发数据传递，该机制并不能完全适用。因为 Rate-Based 流量控制的实时性较弱，当一个 VC 需要瞬间传递大量报文时，Rate-Bsed 机制不能及时地为这条 VC 提供足够的数据传送率；而当一个 VC 拥塞时，也不能及时地降低数据传送率。因此使用 Rate-Based 机制并不能满足网络上突发数据传送的需要，此时需要使用 Credit-Based 机制对流量进行控制。

9.1.2 Credit-Based 流量控制

为便于 Credit-Based 流量控制机制的讨论，假设在网络中存在三类节点，分别是 Upstream 节点、Current 节点和 Downstream 节点，这些节点之间通过实际的物理链路互连，在每一个节点内部都使用两个 VC，其结构如图 9-4 所示。

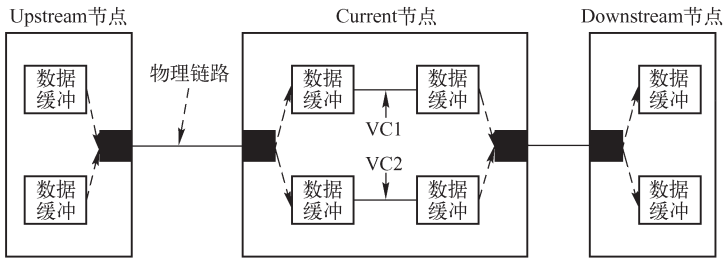

图 9-4　Upstream Current 与 Downstream 节点的关系

其中 Upstream 节点通过物理链路将数据报文发向 Current 节点，而 Current 节点通过物理链路将数据报文发向 Downstream 节点。在虚通路的设计中，每个节点的发送端口和接收端口之间具有分属于不同 VC 的数据缓冲，这些数据缓冲之间的互连组成了不同的 VC。

Current 节点首先将来自 Upstream 节点的报文暂存在数据缓冲中，然后再发送到 Downstream 节点。当 Upstream 节点通过 Current 节点，向 Downstream 节点发送数据报文时，流量控制发生在 Upstream 节点和 Current 节点、Current 节点和 Downstream 节点之间，而不是 Upstream节点到 Downstream 节点。简而言之，流量控制发生在链路的两端，基于端到端之

间，而不是基于节点到节点间。

在 Upstream 节点、Current 节点和 Downstream 节点中存在两个 VC，下文以其中的一个 VC 为例，说明如何使用 Credit-Based 机制进行数据传递。值得注意的是，Current 节点、Upstream 节点和 Downstream 节点只是一个相对概念。Current 节点也可以从 Downstream 节点接收数据报文，而向 Upstream 节点转发这些数据报文，从而组成一个双向通路。为简便起见，本章仅讨论在单向通路下，Credit-Based 流量控制机制的原理与实现。

Credit-Based 机制基于“Credit”进行数据传递，当 Upstream 节点需要发送数据报文时，需要具备足够的 Credit，才能向 Current 节点发送数据。这个 Credit 由 Current 节点提供，并与 Upstream 节点保存的 Credit 同步，为此 Current 节点需要定时向 Upstream 发送“传递 Credit”的数据报文。

下文为简便起见，假设节点间传送的数据报文，其长度固定，而且每次只能传递一个数据报文。Credit-Based 机制需要使用以下参数进行报文传递。

- Buf_Alloc 参数。该参数保存在 Current 节点中接收数据缓冲的总大小。Upstream 节点能够发送的数据报文总数不能大于该参数。
- Crd_Bal 参数。该参数是 Upstream 节点可以发送的数据报文数，Current 节点需要定时向 Upstream 节点发送 Credit 报文。Upstream 节点收到该报文后，使用该报文中的“Credit”同步 Crd_Bal 参数。Upstream 节点可以发送的数据报文数不能超过 Credit_Bal 参数。
- Tx_Cnt 参数。该参数为 Upstream 节点已经发送的数据报文数。
- Fwd_Cnt 参数。该参数为 Current 节点转发到 Downstream 节点的数据报文数。

Credit-Based 流量控制使用的各个参数之间的关系如图 9-5 所示。

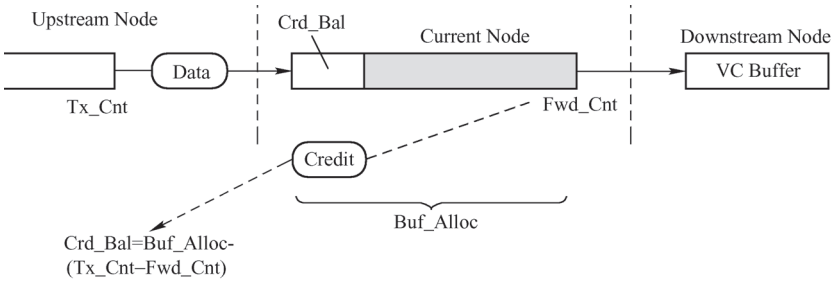

图 9-5　使用 Credit-Based 机制进行数据传递

1. Upstream 节点向 Current 节点发送报文

Upstream 节点向 Current 节点发送报文时，Current 节点必须有足够的缓冲，而且 Current 节点需要预先将其剩余的缓冲数量，即 Credit（其值为一个正整数），及时地发送给 Upstream 节点。Upstream 节点使用 Crd_Bal 参数保存这个 Credit。

Crd_Bal 参数的初始值为 Buf_Alloc，即 Current 节点能够接收的最大报文个数，该值在系统初始化时由 Current 节点发向 Upstream 节点。因此 Upstram 节点在流量控制机制初始化完毕后，Crd_Bal 参数与 Current 节点中能够存放的最大报文数相同。

Upstream 节点每成功发送一个数据报文后，Crd_Bal 值减 1，当 Crd_Bal 参数为 0 时，Upstream 节点不能向 Current 节点发送数据报文。此时 Upstream 节点必须等待 Current 节点发

送 Credit 报文，更新 Crd_Bal 参数后，才能继续发送数据报文。

2. Current 节点向 Upstream 节点发送 Credit

Current 节点收到来自 Upstream 节点的数据报文后，将会根据链路的实际情况，将这些报文转发到 Downstream 节点。

假设在一段时间之内，Current 节点共收到了 Tx_Cnt 个报文，而转发了 Fwd_Cnt 个报文，那么此时在 Current 节点中还能容纳 Buf_Aloc - (Tx_Cnt - Cx_Cnt)个报文空间。Current 节点将这个值作为 Credit，发送到 Upstream 节点。而 Upstream 节点将根据这个 Credit 的值更新 Crd_Bal 参数。

3. Current 节点将报文转发到 Downstream 节点

Current 节点接收到报文之后再将其转发出去涉及一些路由算法。常用的算法有 Cut-Through 路由和 Wormhole 路由算法。

当使用 Wormhole 路由方式时，一个报文将被分解为若干个 Flit，包括 Header Flit、Data Flit 和 Tail Flit。当数据报文到达 Current 节点后，Current 节点立即对 Header Flit 进行分析，首先根据路由算法决定发向哪个 Downstream 节点[㊀]。如果在对应的 Downstream 节点中，链路空闲且有足够的缓冲资源时，Current 节点将发送 Data Flit 直到 Tail Flit，即发送整个数据报文；如果对应的 Downstram 节点没有资源接收这个报文，数据报文将在 Current 节点中存储。

Cut-Through 路由与 Wormhole 路由类似。采用 Cut-Through 路由时，Downstream 节点必须具有接收整个报文的能力时，才能接收报文；而采用 Wormhole 路由时，Downstream 节点具有接收部分报文的能力时，就可以接收报文。采用 Wormhole 路由的优点是数据报文传送延时较短，每一个节点所需要的数据缓存相对较小。当网络发生拥塞时，采用 Wormhole 路由技术可能会使一个报文分别缓冲在 Current 节点和 Downstream 节点中，而使用 Cut-Though 路由技术数据报文最终缓冲在一个节点中。

巨型机一般使用 Wormhole 技术进行报文传递，而网络系统中多使用 Cut-Though 路由技术。有关 Wormhole 和 Cut-Though 技术的优劣分析超出了本书的范围，本书不会对此进行详细分析。巨型机应用针对的是可预知的网络拓扑结构，而网络系统的拓扑结构是变化的。在一个网络拓扑结构可预知的前提下，采用 Wormhole 技术可以在避免拥塞的前提下，降低网络报文的传递延时；而对于一个未知的网络拓扑结构，使用 Cut-Though 技术更为合理。

9.2 Credit-Based 机制使用的算法

在第 9.1.2 节中提到的 Credit-Based 机制是一个较为理想的模型，在这个模型中，没有考虑网络的延时和拥塞，也没有考虑 Current 节点何时采用哪种策略将 Credit 传送给 Upstream 节点，同时也没有考虑 Buf_alloc 缓冲是否会出现上溢出（Overrun）或者负载（Underrun）。本节将首先给出 Overrun 和 Underrun 的定义，然后讨论 Credit-Based 机制使用的各类算法以及这些算法中使用的各类参数。

- 在本章中，Overrun 指 Current 节点没有足够的缓存接收来自 Upstream 节点的数据报

㊀ 在网络系统中，Current 节点可能对应多个 Downstream 节点。

文；或者 Downstream 节点没有足够的缓存接收来自 Current 节点的数据报文。

- 而 Underrun 指 Downstream 节点有足够的缓存可以接收 Current 节点的报文，但是 Current 节点的缓存中没有需要发送的报文；或者 Current 节点有足够的缓存可以接收 Upstream 节点的报文，但是在 Upstream 节点的缓存中没有需要发送的报文。

这两种溢出情况都将导致链路带宽的浪费，在实际设计中需要尽力避免这两种溢出。此外在一个设计中，还需要考虑链路的传送延时。由于传送延时的存在，Current 节点向 Upstream 节点传送 Credit 信息时，这个 Credit 信息并不能瞬间到达，因而会造成这两个节点间，Credit 的同步问题。如果一个设计将上述这些因素考虑进去，Credit-Based 机制的实现更为复杂。为深入研究 Credit-Baed 机制所使用的算法，我们首先定义以下系统参数。

（1）R_{TT}（Round Trip Time）

该参数记载数据通路的链路延时，单位为 s（秒）。使用 Credit-Based 机制进行报文传递时，Upstream 需要获得 Credit 然后才能发送报文。在链路中存在两个延时，一个是 Current 节点向 Upstream 节点发送 Credit 报文的线路延时 T_{CU}，另一个是 Upstream 节点向 Current 节点发送数据报文的延时 T_{UC}。

如果一个物理链路的发送与接收链路速度相等，而且 Credit 报文长度等于数据报文长度时，T_{CU}将等于 T_{UC}。但是在很多情况下这两个值并不相等。本章为简化起见，假设 T_{CU} 与 T_{UC}的值相等。

而 R_{TT}是这两种延时之和，R_{TT}的值和物理链路的延时成正比。除此之外节点在发送数据报文时需要通过若干逻辑门，这也增加了传送延时。R_{TT}的值可以在链路配置时计算出来，但是在具体实现中，设计者可能使用一个事先预估的数值作为 R_{TT}。值得注意的是，该参数不能估计得过低，否则将会造成 Current 节点的 Overrun；也不能估计得过高，否则将可能引发 Current 节点的 Underrun。

（2）BLINK

该参数存放 Upstream、Current 和 Downstream 节点间数据传递的峰值带宽，即数据链路所能提供的最大物理带宽，单位为 bps（Bits Per Second）。为简便起见，本章认为这几个节点间进行数据传递时的峰值带宽相等。

（3）Packet_Size

该参数存放一个数据报文的大小，单位为 Bit。假定所有节点间进行数据通信时使用的数据报文的大小相等。值得注意的是，在 PCIe 总线中，数据报文并不等长，这为 PCIe 总线的流量控制带来了不小的麻烦。

（4）F（In-flight Data）

该参数存放在 R_{TT}时间段内，Upstream 节点和 Current 节点之间的双向链路上存在的报文，其单位为报文数。其最大值等于 $R_{TT} \times B_{LINK}$/Packet_Size。该值在链路进行远距离传送时必须要考虑。而 PCIe 链路通常在一个 PCB 内部，至多作为机箱到机箱之间的链路，因此 RTT 的值非常小，F 参数几乎可以忽略不计。

（5）B_{VC}

该参数存放源节点到目标节点的有效数据带宽，单位为 bps。在一个物理链路上除了要传递有效数据之外，还需要传递 Credit 报文，因此 $B_{VC} < B_{LINK}$。

该参数不一定是一个常数，因为在一个实际的系统中，不同时间内的带宽并不一定相

等。为了简化模型，使用 B_{VCR}参数替代 B_{VC}参数，B_{VCR}参数为一个常数。

本节力求简化流量控制的数学模型，并依此进行量化分析。有关流量控制的量化分析涉及一些相对复杂的数学推导，本章对此不做详细说明。

使用 Credit-Based 机制时，Buf_Alloc 参数可以被分解为三部分，分别由 N1、N2 和 N3 组成，Buf_alloc 参数与 N1 ~3 间的关系如图 9-6 所示。

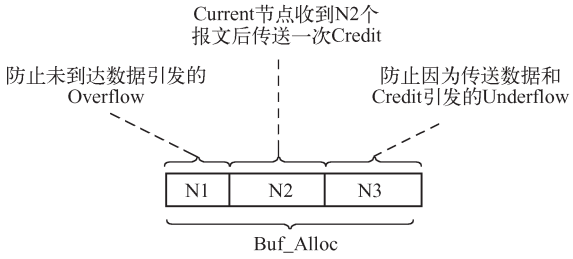

图 9-6　Buf_Alloc 的组成

（1）N1 缓冲

该参数用来防止因为线路延时而造成的 Overrun，Credit-Based 流量控制机制所采用的算法对该参数的解释有略微区别，详见下文。

（2）N2 缓冲

N2 的值与 VC 的设置有相关。当 Current 节点向 Downstream 节点每转发 N2 个报文后，将向 Upstream 节点发送一个 Credit 报文。Current 节点可以将所有 VC 的 N2 值设为相同，也可以分别设置。

N2 的值越大，Buf_Alloc 参数的值也越大，节点发送 Credit 报文的频率也越低，从而 Credit 报文占用数据通路带宽的比例越小；N2 的值越小，则 Buf_Alloc 的值也越小，向 Upstream 节点发送的 Credit 报文的频率越高，Credit 报文占用数据通路带宽的比例也越大。如果 N2 的值为 10，Current 节点每转发 10 个报文后，才向 Downstream 发送一个 Credit 报文，从而整个系统用于传送 Credit 报文所占用的带宽不会超过 10%。

（3）N3 缓冲

该参数保证 Current 节点不会出现 Underrun，即出现 Downstream 节点仍有缓存接收报文，而在 Current 节点的缓存中没有报文需要发送这种情况。

为此在进行系统设计时，需要合理设置 N3 的值。使得在理想情况下，只要 Upstream 节点向 Current 节点不断地发送报文，且 Downstream 节点有足够的缓存时，就不会在 Current 节点中出现 Underrun 的现象。

当然 Downstream 节点接收数据报文的速度足够快，而且 Current 节点未能及时地从 Upstream 节点获得报文时，Current 节点总会出现 Underrun 的现象。

假设 Downstream 节点从 Current 节点获取报文的速度为 B_{VCR}，只要 $N3 = B_{VCR} \times R_{TT}/Packet\_Size$，那么在 Downstream 节点将 N3 中的报文取完之前，Current 节点总能获得新的报文（其前提是 Upstream 节点在不断地向 Current 节点发送数据报文）。

由此可见 B_{VCR}与 N3 的容量有关，B_{VCR}越大，N3 也越大。N3 的容量可以影响数据通路的有效带宽，在流量控制机制的实现中，如果 N3 过小，那么 Current 节点将出现 Under-

run 的现象，从而影响数据通路的有效带宽。根据上述数学模型，H. T. Kuang 与 Alan Chapman 提出了三种流量控制算法，分别为 N123、N123 + 和 N23 算法。

9.2.1 N123 算法和 N123 + 算法

N123 算法的使用规则如下所示。

（1）Current 节点发送上一个 Credit 报文之后，至少需要向 Downstream 节点转发 N2 个报文后，才能发送下一个 Credit 报文。

（2）Credit 报文中存放的 Credit 为 Current 节点已经发送的报文数，其最大值为 N2 + N3，其中 N1 不参与“Credit”的计算，因为在 N123 算法中，N1 用来防止 Current 节点的溢出。当 Current 节点使用的缓冲超过 N2 + N3 时，Current 节点不再向 Upstream 节点发送 Credit 报文，即便 Current 节点已经向 Downstream 转发了 N2 个报文。

（3）Upstream 节点收到 Credit 报文后，使用 Crd_Bal 参数保存这个报文中的 Credit，当 Crd_Bal 参数不为 0 时，Upstream 节点可以发送数据报文。Upstream 节点每发送一个报文，Crd_Bal 参数将减 1，当这个参数为 0 时，Upstream 节点停止发送报文。Crd_Bal 参数的初始值为 N2 + N3。

采用以上算法时，必须保证 $N1 \leqslant R_{TT} \times B_{LINK}/Packet\_Size$，此时 Current 节点的接收缓冲才不会溢出。下文将详细解释为什么 N1 的最小值为 $R_{TT} \times B_{LINK}/Packet\_Size$，并用一个实例说明 N1 最小值的设置原因，而不进行理论分析。

假设在某一个时间点，Upstream 节点的 Crd_Bal 参数为 0，而 Current 节点的 N2 和 N3 缓冲区被完全占满，此时 Upstream 节点不能发送新的报文，直到获得新的 Credit 报文。之后 Upstream、Current 节点和 Downstream 节点按照以下步骤运行。

（1）当 Current 节点向 Downstream 节点转发 x（$x \geqslant N2$）个报文后，将通过 Credit 报文将 x 传递给 Upstream 节点。

（2）当 Upstream 节点收到这个 Credit 后，将更新 Crd_Bal 参数为 x，之后可以向 Current 节点发送 x 个报文。这些报文将通过链路不断发向 Current 节点。

（3）假设此时 Current 节点接收到的报文个数为 z1（$z1 \leqslant x$），Current 节点在接收这些报文的同时，向 Downstream 节点又转发了 y 个报文，此时 Current 节点一共需要向 Upstream 节点发送的 Credit 为（x + y - z1）。假定此时 Downstream 节点由于缓存不足，禁止接收来自 Current 节点的报文，Current 节点的空闲缓存将定格为 x + y - z1。

（4）在 Current 节点向 Upstream 节点发送 Credit（该值为 x + y - z1）的过程中，该节点又陆续收到了一些报文，其个数为 z2。值得注意的是“Current 节点发送 Credit 报文”到“Upstream 节点收到这个报文”有一段延时，该延时等于 T_{CU}。

（5）Upstream 节点收到新的 Credit 后，将 x + y - z1 个报文发向 Current 节点，除此之外在从 Upstream 节点到 Current 节点间的链路上还留有一部分报文，其个数为 z3。这段残留的报文数为在 T_{UC} 时间段内传递的数据报文。

（6）因此 Current 节点最终收到的报文个数为（x + y - z1）+ z2 + z3 个。

（7）其中 x + y - z1 个报文可以被 Current 节点中空闲缓存吸收，而多出来的 z2 + z3 个数据报文将放置到 N1 中。

Max（z2 + z3）的计算方法如公式 9-1 所示。

$$\begin{aligned}\mathrm{Max}(z2+z3) &= (T_{\mathrm{CU}} \times B_{\mathrm{LINK}} + T_{\mathrm{UC}} \times B_{\mathrm{LINK}})/Packet\_Size \\ &= R_{\mathrm{TT}} \times B_{\mathrm{LINK}}/Packet\_Size \end{aligned} \quad (9\text{-}1)$$

因此只要 N1 被设置为 $R_{TT} \times B_{LINK}$/Packet_Size，采用 N123 算法就一定不会在 Current 节点上产生 Overrun。

通过以上分析，可以发现由于物理链路传送的延时，采用 N123 算法时，Current 节点将多收到 z2 + z3 个报文，其中 z2 是 Current 节点更新 Upstream 节点 Credit 的过程中产生；而 z3 是 Upstream 节点更新完毕 Credit 后，在网络线路上残留的报文。使用 N123 算法时，需要设置 N1 缓冲处理这些因为网络延时产生的报文。

以上过程并非严格的数学证明，只是使用较为简单的推理说明，在某些情况下，在 Current 节点中的 N1 至少为 $R_{TT} \times B_{LINK}$/Packet_Size 时，才能够保证 Current 节点的接收缓冲不会溢出。希望读者认真理解 N1 缓冲在 N123 算法中的意义。

N123 + 算法基于 N123 算法，但是发送 Credit 报文的方式略有不同。N123 算法要求 Current 节点每转发 N2 个报文后，才能给 Upstream 节点发送一个 Credit 报文；而 N123 + 算法除了要求同样的规则之外，还要求 Current 节点在一个时间段 RTT 中至少发送一个 Credit 报文，即发送上一个 Credit 报文之后，即便 Current 节点没有向 Downstream 转发了 N2 报文，在一个时间段 RTT 中也至少要向 Upstream 节点发送一次 Credit 报文。

采用这种方法，因为在每一个 R_{TT}时间段里都会向 Upstream 节点发送 Credit 信息，因此 F 将小于这个 Credit，而这个 Credit 又小于 N2 + N3。为此使用这种方法时，N1 的计算方法如公式 9-2 所示。

$$N1 = \mathrm{Min}(N2 + N3, R_{TT} \times B_{LINK}/Packet\_Size) \quad (9\text{-}2)$$

使用这种方法，在(N2 + N3) < $R_{TT} \times B_{LINK}$/Packet_Size 时，Current 节点将可以使用较小的 N1 缓冲，从而节约了接收缓冲。这种算法对于网络延时较长的通信网络有所帮助，而网络延时较短时，但是采用这种方法，发送 Credit 报文所占用的带宽较大。在 PCIe 总线中，端到端的延时相对较小，因此没有必要使用这种流量控制机制。

9.2.2 N23 算法

N23 算法是流量控制中常用的算法，使用该算法的优点是 Current 节点的缓存中不包含 N1，从而降低了节点的缓存容量。该算法基于 N123 算法，区别在于使用该算法时 Crd_ Bal 参数的计算。基于该算法的实现方式如图 9-7 所示。

N23 算法的使用规则如下。

- 当系统初始化时，Crd_Bal 参数为 N2 + N3 - E，E 为在时间段 R_{TT}中，Upstream 节点向 Current 节点发送的报文数，其值等于 $B_{VCR} \times R_{TT}$/Packet_Size。而 Upstream 节点每发送一个报文，Crd_Bal 参数的值将减 1。当 Crd_Bal 参数等于 0 时，Upstream 节点停止发送报文，直到重新获得 Current 节点的 Credit 报文，更新 Crd_Bal 参数后，才能继续发送。
- 使用 N23 算法时，Crd_Bal 参数与 Current 节点发出的 Credit 并不相等，而是等于 Credit - E。
- Current 节点至少需要向 Downstream 节点发送 N2 个报文后，才能向 Upstream 节点发送 Credit 报文，与 N123 算法一致。

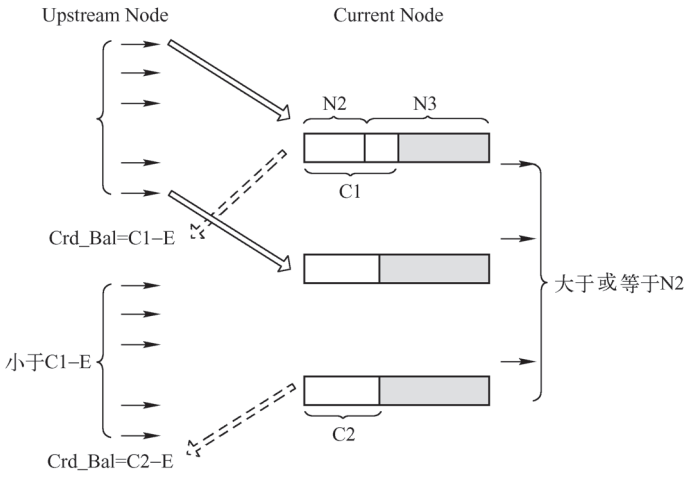

图 9-7　基于 N23 算法的流量控制

通过以上介绍，可以发现之所以采用 N23 算法，不需要设置 N1，是因为 Upstream 节点使用的 Crd_Bal 参数与 N123 算法相比少 E 个包，所以 N23 算法虽然没有使用 N1 缓冲，也不会导致在传送过程中出现 Overrun，因为采用 N23 算法，将 N1 隐含在 E 中。同时因为 N3 的存在，采用 N23 算法也不会导致在传送过程中出现 Underrun。

综上所述，使用 N23 算法进行数据报文的传递时，只要 N2 和 N3 参数设置合理，将不会发生节点的 Overrun 和 Underrun 的情况。但是还需要继续讨论 Current 节点发送 Credit 报文时会不会引发 Overrun 和 Underrun。

首先 Current 节点发送 Credit 报文不会引发 Upstream 节点的 Overrun，因为 Upstream 节点每次接收到新的 Credit 报文都会把 Crd_Bal 参数更新，不可能因为 Credit 报文过多而无法处理。但是 Current 节点发送过多的 Credit 报文将严重影响物理链路的有效利用率，在流量机制的实现中，需要合理设置发送 Credit 报文的频率。

而 Current 节点向 Upstream 节点传递 Credit 报文延时过大时，可能会引发 Current 节点的 Underrun。在这种情况下，Upstream 节点虽然有很多报文等待发送，但是由于 Crd_Bal 参数为 0，不能发送这些报文。

因此造成在 Current 节点的数据缓冲中，没有数据报文需要发向 Downstream 节点，尽管此时在 Current 节点中还有足够的缓存可以接收数据报文。在流量控制机制的设计中，需要考虑 Credit 报文的传送延时，合理设置 Current 节点中的缓冲，以保证 Upstream 节点在获得新的 Credit 之前，Current 节点的缓冲中具有一定的报文数，以避免 Underrun。

采用 N23 算法可以有效避免这种因为 Credit 报文传递不及时而引发的 Underrun（N123 算法与 N23 算法都使用了 N3 缓冲避免 Current 节点的 Underrun）。我们首先基于 N23 算法做出以下假设以简化数学模型。

（1）$N3 = B_{VCR} \times R_{TT}/Packet\_Size$。设置 N3 缓存的主要目的是保证 Current 节点不会出现 Underrun。而 $B_{VCR} \times R_{TT}/Packet\_Size$ 是 N3 缓存的最小值。

（2）Upstream、Current 和 Downstream 间 VC 的通信带宽为 B_{VCR}，其值为一个常数。

（3）Downstream 节点始终有足够的缓冲接收报文。Current 节点可以通畅地将数据报文

转发到 Downstream 节点。

Upstream 和 Current 节点间的数据交换是基于“得到 Credit，然后发送数据”这样的循环。假定在一个发送循环的起始点中，Upstream 节点的 Crd_Bal 参数为 N2，即 Upstream 节点刚收到的 Credit 为 N2 + N3，而采用 N23 算法时，Crd_Bal = Credit − E = (N2 + N3) − E = N2。

我们定格 Upstream 节点刚刚收到 Current 节点 Credit 报文，更新 Crd_Bal 参数完毕这个场景，其时间戳为 T2，而 Current 节点发送这个 Credit 报文的时间戳为 T1。假设从 T1 ~ T2 这段时间内，Current 节点收到 z2 个报文（Current 节点向 Upstream 节点发送 Credit 报文的过程中，仍然在持续地接收报文）。T1 ~ T3 的示意如图 9-8 所示。

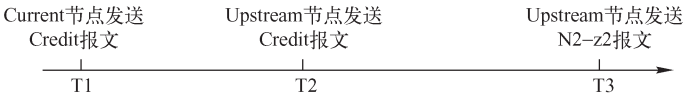

图 9-8　T1 ~ T3 的示意图

由于各节点的带宽都为 BVCR，所以 Current 节点每收到一个报文，都会发送到 Downstream 节点，因此此时 Current 节点的可用缓存始终保持为 N2 + N3。Current 节点向 Upstream 节点发送的 Credit 报文始终为 N2 + N3，而 Upstream 节点收到这个 Credit 后，其 Crd_Bal 参数将被置为 N2。

Upstream 节点收到 Credit 报文后，开始向 Current 节点发送 N2 个报文，Upstream 节点每发送一个报文，Current 节点将收到一个报文（假设此时从 Upstream 节点到 Current 节点的链路之间已经堆积了 z2 个报文，此时物理链路已经充满数据报文）。

假设 Upstream 节点向 Current 节点发送完毕 N2 − z2 个报文（数据报文离开发送端口的时间戳为 T3）。那么从 T3 ~ T1 这段时间里，Current 节点将收到 N2 个报文，同时 Current 节点也将向 Downstream 节点转发完毕 N2 个报文，此时（T3 时间戳）Current 节点将向 Upstream 节点发送 Credit 报文（数值为 N2 + N3）。

而 Upstream 节点处于 T3 这个时刻时，还能向 Current 节点发送 z2 个报文，因为之前已经发送了 N2 − z2 个报文，此时 Crd_Bal 参数为 z2。等到 Upstream 节点将 z2 个报文发送完毕，来自 Current 节点的 Credit 报文恰好到达，因为 Upstream 节点将 z2 个报文发送完毕的时间刚好等于 Current 节点向 Upstream 节点发送 Credit 报文的时间。通过以上计算，可以发现采用 N23 算法不会因为 Credit 报文的传送时间而导致 Current 节点的 Underrun。

此外采用 N23 算法时还需要处理错误报文，N23 算法规定当一个节点收到一个错误数据报文时，将丢弃这个报文，此时这个被丢弃的报文将不占用接收节点的缓存，但是发送节点的 Crd_Bal 参数仍然需要考虑这个报文。

9.2.3　流量控制机制的缓冲管理

上文讲述了基于单个 VC 的流控机制，实际上在 Upstream、Current 和 Downstream 节点中一般含有多个 VC。多个 VC 之间如何合理地使用缓存值得重点关注，在实际设计中，可以为每一个 VC 设置独立接收缓存，也可以使多个 VC 共享同一个接收缓存。在 FCVC 的实现中，可以根据实际情况选择独立缓存或者共享缓存。

其中，每一个 VC 都使用独立接收缓存的流量控制方法称为静态（Static）流量控制；

而使用共享缓存的流量控制方法称为自适应（Adaptive）流量控制。

假定在一个系统中，一共具有 n 条 VC，而且这几条 VC 都使用 N23 算法进行流量控制，那么在使用 Static 流量控制方式时，该系统一共需要的缓冲大小为（n × N2 + N3）× Packet_Size[⊖]。如果（n × N2 + N3）× Packet_Size 并不是很大时，为了使数据链路获得更大的带宽，可以使用 Static 流量控制。使用这种方法，也将极大地简化缓冲管理的设计难度。

值得注意的是，在一个系统工程的架构设计中，应当重点关注“Critical Path”的设计，需要容忍非“Critical Path”的不完美。当（n × N2 + N3）× Packet_Size 的值大到了可以容忍的范围之外时，设计者必须考虑如何减少 Current 节点的接收缓存大小。Static 流量控制是针对每一个 VC 都是按照全负荷运转的情况，在绝大多数应用中，几乎不可能出现每一条 VC 都被充分利用的情况，因为多条 VC 共享一个物理链路，不可能出现所有 VC 都在全负荷运行的情况。为此在系统设计时可以使用 Adaptive 流量控制方法。

Adaptive 流量控制的本质是 Current 节点中，所有 VC 共享一个接收缓存，从而这个缓存可以远小于（n × N2 + N3）× Packet_Size。因为在绝大多数时间内，数据链路的多条 VC 不可能都被充分使用，因此并不需要为每条 VC 都提供 N2 缓冲，而是为所有 VC 统一提供接收缓冲，从而合理使用这些接收缓冲。

目前接收缓存的分配常使用两种算法，分别是 Sender-Oriented 和 Receiver-Oriented 管理算法。这两种算法的缓冲设置如图 9-9 所示。使用 Sender-Oriented 管理算法时，Adaptive Buffer 的分配在 Upstream 节点中完成，如果系统中有多个 Upstream 节点，Current 节点需要在其接收端点处为每个 Upstream 节点准备 Adaptive Buffer，而且 Current 节点并不知道 Upstream 节点的使用情况，这为 Current 节点的缓冲管理带来了不小的困难。

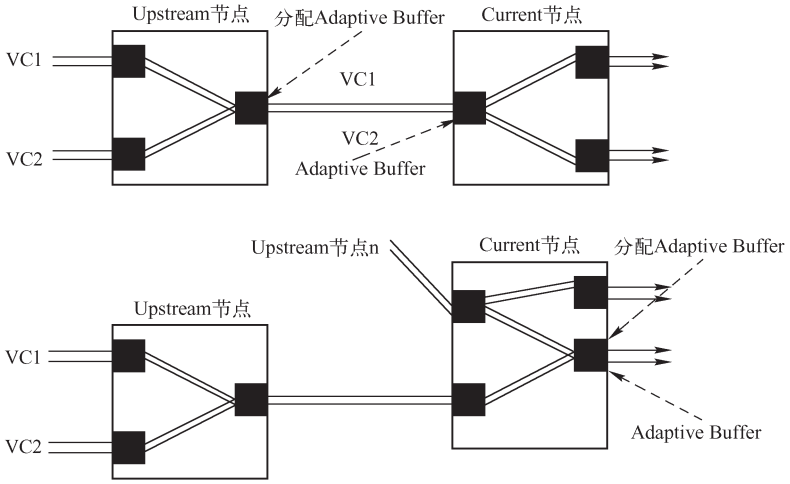

图 9-9　Sender-Oriented 和 Receiver-Oriented 管理算法

而使用 Receiver-Oriented 管理算法可以有效避免这类困难，使用这种算法时，所有来自 Upstream 节点的数据报文在 Current 节点的发送端准备一个 Adaptive Buffer，对这个 Buffer 的分

⊖　多条 VC 可以复用 N3 缓冲，因为多条 VC 对应一条物理链路。但是必须提供独立的 N2 缓冲。

配也在 Current 节点内部完成。这两种算法的具体实现都较为复杂，本节并不详细介绍这些算法，在 PCIe 总线中，RC 和 Switch 的硬件设计将会涉及这些内容，而 EP 无需关心这些问题。

图中的上半部分是 Sender-Oriented 管理算法的示意图，而下半部分是 Receiver-Oriented 管理算法的示意图。由图 9-9 可以发现，使用这两种算法时 Adaptive Buffer 都在 Current 节点中，只是位置不同。

9.3 PCIe 总线的流量控制

我们仍然使用上文的 Upstream、Current 和 Downstream 节点模型分析 PCIe 总线使用的流量控制机制。在 PCIe 总线中，Upstream 节点和 Downstream 节点可以为 RC 或者 EP，而 Current 节点只能为 Switch 或 RC[⊖]。

此外在 PCIe 总线中，允许 Upstream 节点和 Downstream 节点直接相连，而不需要经过 Current 节点，如 RC 的某个端口可以和 EP 直接相连。当然这种情况也可以理解为 Upstream 和 Current 节点直接相连，但是 Current 节点不需要与 Downstream 节点相连。

与传统的流量控制机制相比，PCIe 总线的流量控制机制有所不同。流量控制机制首先出现在互联网中，使用流量控制机制最典型的应用是基于 ATM（Asynchronous Transfer Mode）的分组交换网。在 ATM 分组交换网系统中，数据传递以报文为单位进行，每一个报文都可以独立地通过分组交换网，到达目的地。而 PCIe 总线的数据传递基于节点到节点间的数据传递，一个完整的 PCIe 总线传输需要使用多个报文，而且这些报文和报文之间还有一定的联系，如一个完整的存储器读由存储器读请求 TLP 和存储器读完成 TLP 组成。

在 PCIe 总线中，RC 端口和 EP 之间可以直接互连，而不需要中间节点，这和基于分组交换的网络有很大的不同。此外在 PCIe 总线中，报文的大小并不固定，如数据报文的大小可以为 128B、256B，只要数据报文的有效负载小于 Max_Payload_Size 参数即可。

这些长度不确定的数据报文，为 PCIe 总线的流量控制带来了不小的困难，也是因为这个原因，PCIe 总线的流量控制机制将一个 TLP 分为 TLP 头和 Payload 两部分，并分别为这两部分提供不同的接收缓冲，以合理利用 PCIe 链路的带宽。

PCIe 总线的这些特性实际上不利于流量控制的实现，为此 PCIe 总线在传统流量控制理论的基础上，做出了许多改动。在 PCIe 总线中存在多条 VC，其流量控制也是基于 FCVC 的，但是 PCIe 总线在接收缓冲的设计上与传统的流量控制机制有很大的不同。

PCIe 总线的主要应用领域在 PC 或者服务器中进行板内互连，在这个应用领域中，流量控制并不是最重要的。PCIe 总线的流控机制远非完美，这在某种程度上影响了 PCIe 总线在大规模互连结构中的使用。但是 PCIe 总线的流量控制机制仍有其闪光之处，因此读者仍有必要了解 PCIe 总线的流量控制机制。

与传统流量控制机制相比，PCIe 总线在实现流量控制机制时需要更多的接收缓存，因此 PCIe 总线实现流控的代价相对较大。值得庆幸的是，目前 PCIe 总线上的设备，包括 RC、EP 和 Switch，很少有支持两个以上 VC 通路的。一般来说 PCIe 总线上的设备，如 RC 和

⊖ 如果 RC 支持 Peer-to-Peer 传送方式。

Switch 上也只有两个 VC，多数 EP 仅支持一个 VC。

PCIe 总线的流量控制机制由事务层和数据链路层协调实现，而对系统软件透明。PCIe 总线使用 Credit-Based 流量控制机制，其中 Credit 报文以 DLLP 的形式从 Current 节点反馈到 Upstream 节点。在 PCIe 总线中，数据报文首先以 TLP 的形式通过数据链路层，而到达数据缓存时被分解为 Header 和 Data 两个部分，分别存放到不同的接收缓存队列中。

9.3.1 PCIe 总线流量控制的缓存管理

在 PCIe 总线的节点中，一个 VC 的接收缓存由 PH（Posted Header）缓存、PD（Posted Data）缓存、NPH（Non-Posted Header）缓存、NPD（Non-Posted Data）缓存、CplH（Completion Header）缓存和 CplD（Completion Data）缓存组成。

- PH 缓存存放存储器写请求 TLP 和 Message 报文使用的 TLP 头。
- PD 缓存存放存储器写请求 TLP 和 Message 报文使用的 Payload。
- NPH 缓存存放 Non-Posted 请求 TLP 使用的 TLP 头。
- NPD 缓存存放 Non-Posted 请求 TLP 使用的 Payload。在 Non-Posted 请求 TLP 中，如存储器读请求 TLP 并不含有 Payload 字段，但是 I/O 和配置写请求 TLP 使用 Payload 字段。
- CplH 缓存存放完成报文使用的 TLP 头。
- CplD 缓存存放完成报文使用的 Payload。如上文所述，完成报文分为两大类，带数据的完成报文和不带数据的完成报文。其中不带数据的完成报文不需要使用 CplD 缓存。

在 PCIe 总线中，一个 TLP 从 Upstream 节点传送到 Current 节点时，必须同时具备多个缓存的 Credit 后才能发送。如存储器写请求 TLP，需要同时具备 PH 和 PD 缓存的 Credit，才能发送；而“不带数据的”存储器读完成 TLP，仅需要具备 CplH 缓存即可。这些缓存在 PCIe 设备中的组成结构如图 9-10 所示。

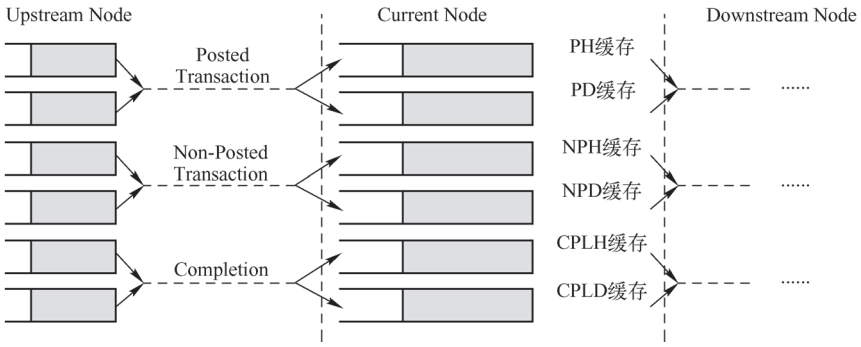

图 9-10　PCIe 总线 Current 节点的缓冲管理

如表 6-2 所示，PCIe 总线根据 Type 字段将 TLP 分为 15 种，而根据这些 TLP 的传输特性，可以将这些 TLP 分为 Posted Transaction、Non-Posted Transaction 和 Completion 三大类。在 PCIe 总线中，这三大类数据传送需要遵循各自的规则，这些 Transaction 也有各自的特点。这三大类 Transaction 在进行数据传递时需要使用不同的缓存，这些缓存由多个单元（Unit）组成。每个 Unit 的大小与缓存类型相关，如表 9-1 所示。

表 9-1　PCIe 总线缓存的单元大小

类　　型	Unit 大小
PH 缓存	5 个 DW，Posted Transaction TLP 头的最大值为 4DW，再加上一个 Digest
PD 缓存	4 个 DW
NPH 缓存	5 个 DW，Non-Posted Transaction 的 TLP 头的最大值为 4DW，再加上一个 Digest
NPD 缓存	1 个 DW。NPD 缓存用于 IO 和配置读写的数据
CplH 缓存	4 个 DW。3DW 的完成报文的头，加上一个 Digest
CplD 缓存	4 个 DW。该缓存至少能够存放一个完整的带数据的完成报文。其总单元数的最小值为 min（RCB/4B，Max_Payload_Size/4B）

PCIe 总线将 Header 和 Data 缓存分离的主要原因是，一个 TLP 的 Header 大小是固定的，如 PH 和 NPH 大小在 5DW 之内，CplH 大小在 4DW 之内；而 Data 的大小并不固定，除了 NPD 的大小为 1 个 DW 之外，其他数据报文的长度由 TLP 的 Length 字段确定，并不固定。PCIe 总线将 Header 和 Data 缓存分离有利于 Data 缓存的合理使用。

PCIe 总线规范将这些缓存使用的 Unit 统称为 FC（Flow Control）Unit，下文将以 FC Unit 简称这些对应缓存的 Unit。因为 Header 的大小固定，所以 Header 缓存能够精确地预知可以容纳几个 Header；而由于 Data 的大小并不固定，Data 缓存无法精确预知可以存放几个 Data，当然 Data 缓存也不可能将 Data 的基本单位设置为 Max_Payload_Size 参数。因为这样做不仅不合理，而且非常浪费资源。将 Header 和 Data 缓存分离，有利于 Data 缓存使用类似 Adaptive 流量控制的方法使所有 Data 共用一个缓存，从而提高了 Data 缓存的利用率。但是也造成 TLP 因为不能同时具有 Head 和 Data 缓存，而无法传送。

在 PCIe 总线中，不同的 TLP 使用对应缓存的 Unit 数量也不同，Current 节点有时需要两种缓存才能接收一个 TLP。不同 TLP 使用的缓存数目如表 9-2 所示。

表 9-2　不同 TLP 使用的缓存

TLP	缓 存 种 类
存储器、IO 和配置读请求	使用 1 个 NPH 单元。Non-Posted 数据请求的 TLP 类型通过完成报文获得数据
存储器写请求	使用 1 个 PH 单元和若干 PD 单元。存储器写请求使用的 PD 单元与 Payload 的长度相关
配置、I/O 写请求	使用 1 个 NPH 单元和 1 个 NPD 单元。Non-Posted 数据传送类型通过完成报文通知 I/O 写结束
不带数据的消息请求	使用 1 个 PH 单元
带数据的消息请求	使用 1 个 PH 单元和若干个 PD 单元
存储器读完成	使用 1 个 CplH 单元和若干 CplD 单元。读完成报文使用的 CplD 单元与 Payload 的长度相关
I/O 和配置读完成	使用 1 个 CplH 单元和 1 个 CplD 单元
配置、I/O 写完成	使用 1 个 CplH 单元

由上表所示，一个 TLP 可能需要使用两种缓存，如存储器写请求需要使用 PH 和 PD 缓存。这也意味着在 PCIe 设备的 VC 中，缓存之间存在依赖关系。当 Upstream 节点向 Current 节点进行存储器写时，发送方必须同时具有 PH 和 PD 两个缓存的 Credit 才能进行；而向 Current 节点发送读完成 TLP 时，Upstream 节点必须同时具有 CplH 和 CplD 两个缓存的 Credit 才能进行。这为 PCIe 总线的流量控制带来了额外的麻烦。

此外，在 PCIe 总线中，进行存储器写和存储器读完成 TLP 时，究竟需要多少个 PD 或者 CplD 单元是随 TLP 而变的，VC 无法预知确切的单元数量。无论 VC 采用何种缓冲分配策

略，这种“不可预知”都会给流量控制带来巨大的麻烦。

报文长度的不确定性为 PCIe 总线流量控制机制带来了许多难以解决的问题。造成这种现象的主要原因是 PCIe 总线源于 PCI 总线，其主要应用来自 PC 领域而不是通信领域。PCIe 总线为了向前兼容 PCI 总线，做出了许多功能上的牺牲。

PCIe 总线使用 Credit-Based 流控机制，Upstream 节点在发送 TLP 时，必须首先获得 Current 节点相应缓存的 Credit。如 Upstream 节点发送存储器写请求时，需要同时具有 Current 节点中 PH 缓存和 PD 缓存的 Credit，才能进行。

9.3.2 Current 节点的 Credit

PCIe 总线规范没有强行规定采用哪种算法实现 Credit-Based 流量控制机制，因此 PCIe 总线上的 Current 节点可以选择使用合理的 Credit-Based 流量控制算法，如上文提到的 N123 算法或者 N123 +算法。PCIe 总线规范没有规定，各类节点如何处理因为链路延时而产生的额外报文和在链路上残留的报文。

在 PCIe 流量管理中还存在一个问题就是 PCIe 总线上各个节点所采用的流量控制算法并不完全统一，虽然 PCIe 总线对此进行了一定程度的约定，但是这个约定较为模糊。因为在 PCIe 总线中，Upstream 节点、Current 节点和 Downstream 节点可能来自不同的生产厂商。虽然这种流量控制算法的“不统一”会为流量控制的整体效率带来影响，也可能会造成接收缓冲的浪费。

但是这种“不统一”并不会严重影响 PCIe 总线的流量控制机制。因为 PCIe 总线的流量控制基于“链路到链路”进行的，只要链路的两端采用相同的协议满足 PCIe 总线的基本约定即可。PCIe 总线上的流量控制机制以 Intel 的 RC 实现作为事实标准。

在 PCIe 总线中，Credit-Based 流量控制的数据传送规则仍然是 Upstream 节点获得 Credit，之后向 Current 节点发送数据；而 Current 节点将一定数量的报文（N2）转发到 Downstream 节点之后，将向 Upstream 节点反馈 Credit。PCIe 总线规范也将向 Upstream 节点反馈 Credit 的过程称为 Advertisement。

PCIe 总线规范并没有规定如何设置 Credit，但是规定了 Credit 在初始化之后的最小值，即 Current 节点发向 Upstream 节点的 Credit 的最小值，如表 9-3 所示。PCIe 总线中，不同的缓存使用不同的 Credit 值。

表 9-3　PCIe 总线初始化后 Credit 的最小值

Credit 的类型	Credit 的最小值
PH 缓存	Credit（PH）的最小值为 0x01，即一个 PH 单元，大小为 5DW
PD 缓存	Credit（PD）的最小值为 Max_Payload_Size/Unit(PD)
NPH 缓存	Credit（NPH）的最小值为 0x01，即一个 NPD 单元，大小为 5DW
NPD 缓存	Credit（NPD）的最小值为 0x01，即一个 NPH 单元，大小为 1DW。值得注意的是原子操作请求使用的 Credit（NPD）可以为 2DW
CplH 缓存	如果 Current 节点不是“最终”节点，则 Credit（CplH）的最小值为 0x01，即一个 CplH 单元，大小为 4DW；否则 Credit（CplH）为 0，该值为 0，表示“不限带宽（Infinite FC Unit）”，其详细解释见下文
CplD 缓存	如果 Current 节点不是“最终”节点，则 Credit（CplD）的最小值为 Max_Payload_Size/Unit(CplD)①；否则 Credit（CplH）为 0

① 注意不是 RCB/Unit（CPLD）。第一个“最终”节点的 RCB 并不相等，但是都小于 Max_Payload_Size。

“最终节点”共有两类组成，一个是 EP，因为当一个报文到达 EP 后，将不会被再次转发，此时 FC Unit 为 0，表示该节点将无条件接收来自 Upstream 节点的报文，而且保证一定不会出现 Overrun 的现象，这就是 PCIe 总线规范提出的 Infinite FC Unit 的含义。

这也意味着 EP 在发起存储器、I/O 和配置读请求时，必须为存储器、I/O 和配置读完成报文预留必要的缓存，保证这些完成报文一定能够被 EP 接收。值得注意的是，只有设备在接收完成报文时，才存在 Infinite FC Unit 的概念。

在许多应用中，“最终节点”支持这种“Infinite FC Unit”将会影响 PCIe 链路的效率。假设一个 PCIe 设备进行 DMA 读，从主存储器获得数据，如果该 PCIe 设备支持 Infinite FC Unit，其传送流程如下所示。

（1）PCIe 设备向 RC 连续发送存储器读请求 TLP。

（2）PCIe 设备每发送一个读请求 TLP 时，将从接收缓冲中为读完成 TLP 预留空间。因为如果 PCIe 设备支持 Infinite FC Unit，必须能够接收这些读完成 TLP。

（3）当 PCIe 设备的接收缓冲使用完毕后，将不能发送存储器读请求 TLP。

（4）PCIe 设备接收来自 RC 的读请求完成 TLP，并将其传送到上层，同时释放该读完成 TLP 使用的接收缓存。

（5）PCIe 设备获得可用的接收缓存后，可以继续发送存储器读请求 TLP。直到完成所有数据请求。

（6）PCIe 设备获得所有存储器读完成 TLP 后，结束整个 DMA 读过程。

由以上过程可以发现在（3）和（5）中，由于接收缓冲不足，整个 DMA 读过程无法形成连续的流水操作，从而影响 DMA 读的数据传送效率。如果 PCIe 设备的接收缓冲无限大时，可以合理地解决这个问题，但是更为合理的方法是确定接收缓冲的最小值。接收缓冲的最小值由 PCIe 链路的延时决定，有关 PCIe 总线延时的详细分析见第 12.4 节。

还有一类“最终节点”是 RC，但 RC 并非在任何情况下都是最终节点。如果多端口 RC 的各个端口之间不支持转发（即 PCIe 总线规范中的 Peer-to-Peer 传送方式）时，RC 将成为“最终节点”；如果多端口 RC 的各个端口之间支持转发，RC 也可能成为“最终节点”。数据通过多端口 RC 的流程如图 9-11 所示。

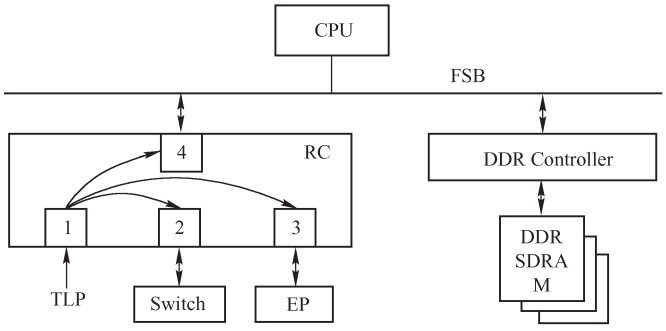

图 9-11　多端口 RC 的数据传递

假设在一个 RC 中有 3 个下游端口，分别是端口 1，2，3，这些端口与 EP 或者 Switch 相连；还有一个上游端口，端口 4，与 FSB 相连。如果 RC 支持端口转发，当到达端口 1 的

TLP 的目的地是和端口 2 或者 3 相连的某个设备时，Credit（CplH）的最小值为 1，而 Credit（CplD）的最小值为 Max_Payload_Size/Unit(CplD)，此时 RC 并不是最终节点。

如果到达端口 1 的 TLP，其目的地是端口 4 时，RC 就是“最终”节点，此时 Credit（CplH）和 Credit（CplD）的最小值为 0，流量控制机制不起作用，来自 Upstream 节点的报文可以不进行流量检测，直接进入 RC，而 RC 必须保证有足够的缓存接收这些报文。这也意味着 RC 发起读请求报文时，必须保证在 RC 中有足够的缓冲接收完成报文。

如果 RC 支持端口间的相互转发，必须设置必要的缓冲以支持流量控制机制，此时 RC 可能成为某个 TLP 的 Current 节点，因此 RC 不将 Credit（CplH）和 Credit（CplD）设为 0，此时使用流量控制机制进行报文转发。

9.3.3 VC 的初始化

PCIe 总线使用 FCP（Flow Control Packets）传递 Credit 信息，FCP 是一种 DLLP，该报文的使用与事务层的接收缓存直接相关，但是对事务层透明，该报文产生于数据链路层，终止于数据链路层。PCIe 总线定义了以下 FCP，如表 9-4 所示。

表 9-4　PCIe 总线定义的 FCP

DLLP 类型	编　　码
InitFC1-P	0b0100 $0V_2V_1V_0$ ①
InitFC1-NP	0b0101 $0V_2V_1V_0$
InitFC1-Cpl	0b0110 $0V_2V_1V_0$
InitFC2-P	0b1100 $0V_2V_1V_0$
InitFC2-NP	0b1101 $0V_2V_1V_0$
InitFC2-Cpl	0b1110 $0V_2V_1V_0$
UpdateFC-P	0b1000 $0V_2V_1V_0$
UpdateFC-NP	0b1001 $0V_2V_1V_0$
UpdateFC-Cpl	0b1010 $0V_2V_1V_0$

① $V_2V_1V_0$ 与 VC 对应，PCIe 总线规定 VC 个数的最大值为 8，因此使用 3 位存放 VC 号。

由上表所示 FCP 共分为三大类 InitFC1、InitFC2 和 UpdateFC。在这三类报文中有 3 个重要的字段。其中 HdrFC 字段存放 Header 的 Credit；DataFC 字段存放 Data 的 Credit；而 VC ID 字段存放不同的 VC 号。其报文格式如图 9-12 所示。

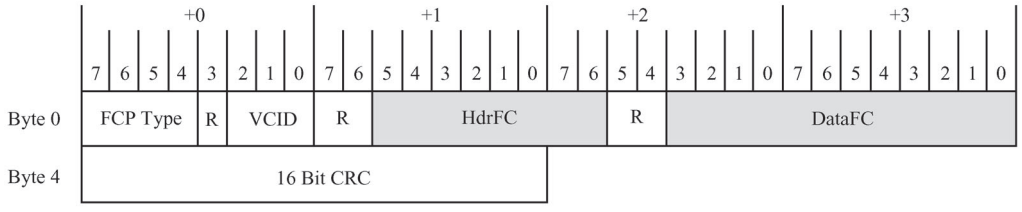

图 9-12　FCP 的格式

Current 节点使用以上报文向 Upstream 节点发送 Credit 信息，其中 InitFC1 和 InitFC2 与 VC 的初始化相关，而 UpdateFC 负责向 Upstream 节点反馈 Credit 信息。我们首先讲述 Current 节点如何使用 InitFC1 和 InitFC2 报文初始化 VC 的流量控制，并在第 9.3.4 节讲述如何使用 UpdateFC 报文实现 PCIe 总线的流量控制。

在各个节点能够正常使用之前，首先需要对 VC0 进行初始化。当 VC0 初始化完毕之后，PCIe 设备可以对 VC1 ~7 进行初始化。在 VC0 初始化完成之前，设备不能接收任何 TLP，而 VC0 初始化完毕后，PCIe 总线的相应节点在使用 VC0 接收 TLP 的同时，初始化其他 VC。在 VC 的初始化过程中存在两个状态，FC_INIT1 和 FC_INIT2。

PCIe 总线的数据链路层共有三个状态，分别为 DL_Inactive（PCIe 链路无效状态，链路不可用或者链路上没有连接有效设备）、DL_Init（PCIe 链路可用，此时将进行 VC0 的初始化）和 DL_Active（链路可以正常使用）。

当 Current 节点的数据链路层进入 DL_Init 状态时，该节点的 VC0 将进入 FC_INIT1 状态。Current 节点将在 FC_INIT1 状态首先初始化 VC0，之后才能初始化其他 VC。其他 VC 的使能位在 Current 节点的配置空间中，当系统软件打开这些使能位时，Current 节点将初始化其他 VC。

当 Current 节点的 VC 进入 FC_INIT1 状态时，事务层需要首先禁止该 VC 发送数据报文，随后 Current 节点将向 Upstream 节点依此发送 InitFC1-P、InitFC1-NP 和 InitFC1-Cpl[⊖] 报文初始化 Upstream 节点使用的 Credit。

此时 Current 节点还可能收到来自 Downstream 节点的 InitFC1-P、InitFC1-NP 和 InitFC1-Cpl 报文，并初始化 Current 节点的 Credit。当 Current 节点收到这些来自 Downstream 节点的 FCP 后，将设置对应缓冲的 FI1 状态位为 1。

当 Current 节点所有缓存的 FI1 状态位有效后，VC 将进入 FC_INIT2 状态。Current 节点进入 FC_INIT2 状态之前，Upstream 节点获得的 Credit 和 Current 节点的空余缓存大小相等，Current 节点获得的 Credit 和 Downstream 节点的空余缓存大小相等。这一点和其他流量控制机制类似，即 Crd_Bal 的初始值为 Buf_Alloc。

PCIe 总线还提供了 FC_INIT2 状态，该状态的主要功能是验证 FC_INIT1 的结果。当节点进入 FC_INIT2 状态时，与流量控制相关的缓存已经初始化完毕。PCIe 总线设置 FC_INIT1 和 FC_INIT2 这两个状态与数据链路层的状态机相关。

如第 7.1.1 节所示，当数据链路层处于 DL_Init 状态时，将初始化 PCIe 总线的流量控制机制。当 VC 处于 FC_INIT1 状态时，数据链路层通知事务层 DL_Down 状态位有效，此时事务层不能向对端设备发送 TLP，从而流量控制机制的初始化可以在一个“相对没有干扰的环境”下进行。

而当 Current 节点的 VC 进入 FC_INIT2 状态时，事务层需要首先禁止使用这条 VC 发送报文，之后 Current 节点向 Upstream 节点依此发送 InitFC2-P、InitFC2-NP 和 InitFC2-Cpl 报文初始化 Upstream 节点的发送缓冲。当 Upstream 节点收到这些报文之后，将丢弃这些报文中包含的 Credit 信息，并设置相应的 FI2 状态位。

同理 Current 节点也将收到来自 Downstream 节点的 InitFC2-P、InitFC2-NP 和 InitFC2-Cpl

⊖ PCIe 总线规定每隔 34μs 发送一组 InitFC 报文。

报文，并设置 Current 节点的 FI2 状态位。当所有数据缓存的 FI2 状态位有效后，将完成 PCIe 链路流量控制机制的初始化。最后数据链路层通知事务层 DL_Active 状态位有效，此时事务层可以使用这个 VC 发送 TLP。

9.3.4 PCIe 设备如何使用 FCP

PCIe 总线完成流量控制的初始化之后，Current 节点、Upstream 节点和 Downstream 节点通过发送 UpdateFC-P、UpdateFC-NP 和 UpdateFC-Cpl 报文进行流量控制。

1. Upstream 节点向 Current 节点发送报文

Upstream 节点向 Current 节点发送报文时，需要设置一些参数。

（1）CREDITS_CONSUMED，简称为 CC。该参数存放 Upstream 节点已经发送了多少个 FC Unit，不同数据缓存的 FC Unit 的大小见表 9-3。在 PCIe 设备初始化时该参数为 0，之后 PCIe 设备每发送一个 FC Unit，该值将加 1。PCIe 总线规定

$$CC = (CC + Increment) \bmod 2^{Field\ Size}$$

其中 Increment 指发送的 TLP 含有的 FC Unit 个数。其中 PH、NPH 和 CplH 的 Field Size 参数为 8，而 PD、NPD 和 CplD 的 Field Size 参数为 12。

（2）CREDIT_LIMIT，简称为 CL。该参数存放当前节点的 Credit，在 VC 初始化时，该参数通过收到的 InitFC1 报文赋值。此后每收到 UpdateFC 报文时，Current 节点将此参数与 UpdateFC 报文中的 Credit 比较，如果结果不等，则使用 UpdateFC 报文中的 Credit 对此参数重新赋值。

在 Upstream 节点中，设置了一个 Credit 检查逻辑（Transmitter Gating Function），用来判断在 Current 节点中是否有足够的缓存接收即将发送的 TLP。如果 Upstream 节点没有足够的 Credit 时，则不能向对端设备发送这个 TLP。

Upstream 节点检查缓存的算法如下。首先将 CUMULATIVE_CREDITS_REQUIRED 参数简称为 CR，而 CR = CC + <PTLP(即将发送 TLP 所需要的 Credit)>。当以下公式

$$(CL - CR) \bmod 2^{Field\ Size} \leqslant 2^{Field\ Size}/2$$

成立时，表示 Upstream 节点可以向 Current 节点发送 TLP 报文。

使该公式成立有一个额外需求，即每次发送的 Credit 小于 $2^{Field\ Size}/2$[⊖]，此时 CL 不可能比 CC 大 $2^{Field\ Size}/2$。当(CL-CR) Mod $2^{Field\ Size}$ 不大于 $2^{Field\ Size}/2$ 时，表示 Current 节点有足够的缓冲接收来自 Upstream 节点的报文；如果(CL-CR) Mod $2^{Field\ Size}$ 大于 $2^{Field\ Size}/2$ 时，运算结果是一个负数，表示 Current 节点没有足够的缓冲接收来自 Upstream 节点的报文。

2. Current 节点从 Upstream 节点接收报文

Current 节点接收来自 Upstream 节点发送报文时，也需要设置一些参数。

（1）CREDITS_ALLOCATED，简称为 CA。该参数存放 Current 节点一共允许 Upstream 节点发送多少个 FC Unit，不同数据缓存的 FC Unit 初始化后使用的最小值见表 9-3。在初始化时该参数为 Current 节点接收缓冲的大小，之后 Current 节点每分配一个 FC Unit，该值将加

⊖ 相当于 N23 算法的 N3 等于 $2^{[Field\ Size]}/2$，此时可以防止 Current 节点的 Underrun。但是 PCIe 总线并没有规定发送 Credit 的频率。

1。PCIe 总线规定：

$$CA = (CA + Increment) \bmod 2^{Field\ Size}$$

其中 Increment 指 Current 节点新分配的 FC Unit 个数。PH、NPH 和 CplH 的 Field Size 参数为 8，而 PD、NPD 和 CplD 的 Field Size 参数为 12。Current 节点使用 UpdateFC DLLP 将 CA 传递给 Upstream 节点，之后 Upstream 节点使用该值更新 CL 参数。

（2）CREDIT_RECEIVED，简称为 CRCV。该参数存放 Current 节点一共接收了多少个 FC Unit。在初始化时该参数为 0，之后 Current 节点每接收一个 FC Unit，该值将加 1。PCIe 总线规定：

$$CRCV = (CRCV + Increment) \bmod 2^{Field\ Size}$$

其中 Increment 指 Current 节点新接收的 FC Unit 个数。

Current 节点可以设置一个逻辑检查部件，检查来自 Upstream 节点的 TLP 报文是否会造成 CA 的溢出。这个逻辑检查部件是一个可选件，因为 Upstream 节点在发送 TLP 时，已经保证了 Current 节点的 CA 不会溢出。当以下公式

$$(CA\text{-}CRCV)\ Mod\ 2^{Field\ Size} \geqslant 2^{Field\ Size}/2$$

成立时，Current 节点认为 CA 溢出，此时 Current 节点将抛弃来自 Upstream 节点的 TLP，而并不改变 CRCV 的值，同时释放为这个 TLP 预先分配的缓存空间。

9.4 小结

本章仅使用了较少的篇幅讲述 PCIe 总线的流量控制机制，而重点讨论通用流量控制机制的基本原理。PCIe 总线由于强调与 PCI 总线的兼容，流量控制机制的设计并不完美。PCIe 总线的主要应用领域依然是 PC，在这个领域中，流量控制并不是最重要的。

第 10 章　MSI 和 MSI-X 中断机制

在 PCI 总线中，所有需要提交中断请求的设备，必须能够通过 INTx 引脚提交中断请求，而 MSI 机制是一个可选机制。而在 PCIe 总线中，PCIe 设备必须支持 MSI 或者 MSI-X 中断请求机制，而可以不支持 INTx 中断消息。

在 PCIe 总线中，MSI 和 MSI-X 中断机制使用存储器写请求 TLP 向处理器提交中断请求，下文为简便起见将传递 MSI/MSI-X 中断消息的存储器写报文简称为 MSI/MSI-X 报文。不同的处理器使用了不同的机制处理这些 MSI/MSI-X 中断请求，如 PowerPC 处理器使用 MPIC 中断控制器处理 MSI/MSI-X 中断请求，在第 10.2 节中将介绍这种处理情况；而 x86 处理器使用 FSB Interrupt Message 方式处理 MSI/MSI-X 中断请求。

不同的处理器对 PCIe 设备发出的 MSI 报文的解释并不相同。但是 PCIe 设备在提交 MSI 中断请求时，都是向 MSI/MSI-X Capability 结构中的 Message Address 的地址写 Message Data 数据，从而组成一个存储器写 TLP，向处理器提交中断请求。

有些 PCIe 设备还可以支持 Legacy 中断方式[㊀]。但是 PCIe 总线并不鼓励其设备使用 Legacy 中断方式，在绝大多数情况下，PCIe 设备使用 MSI 或者 MSI/X 方式进行中断请求。

PCIe 总线提供 Legacy 中断方式的主要原因是，在 PCIe 体系结构中，存在许多 PCI 设备，而这些设备通过 PCIe 桥连接到 PCIe 总线中。这些 PCI 设备可能并不支持 MSI/MSI-X 中断机制，因此必须使用 INTx 信号进行中断请求。

当 PCIe 桥收到 PCI 设备的 INTx 信号后，并不能将其直接转换为 MSI/MSI-X 中断报文，因为 PCI 设备使用 INTx 信号进行中断请求的机制与电平触发方式类似，而 MSI/MSI-X 中断机制与边沿触发方式类似。这两种中断触发方式不能直接进行转换。因此当 PCI 设备的 INTx 信号有效时，PCIe 桥将该信号转换为 Assert_INTx 报文，当这些 INTx 信号无效时，PCIe 桥将该信号转换为 Deassert_INTx 报文。

与 Legacy 中断方式相比，PCIe 设备使用 MSI 或者 MSI-X 中断机制，可以消除 INTx 这个边带信号，而且可以更加合理地处理 PCIe 总线的“序”。目前绝大多数 PCIe 设备使用 MSI 或者 MSI-X 中断机制提交中断请求。

MSI 和 MSI-X 机制的基本原理相同，其中 MSI 中断机制最多只能支持 32 个中断请求，而且要求中断向量连续，而 MSI-X 中断机制可以支持更多的中断请求，而并不要求中断向量连续。与 MSI 中断机制相比，MSI-X 中断机制更为合理。本章将首先介绍 MSI/MSI-X Capability 结构，之后分别以 PowerPC 处理器和 x86 处理器为例介绍 MSI 和 MSI-X 中断机制。

10.1　MSI/MSI-X Capability 结构

PCIe 设备可以使用 MSI 或者 MSI-X 报文向处理器提交中断请求，但是对于某个具体的

㊀ 通过发送 Assert_INTx 和 Deassert_INTx 消息报文进行中断请求，即虚拟中断线方式。

PCIe 设备，可能仅支持一种方式。在 PCIe 设备中含有两个 Capability 结构，一个是 MSI Capability 结构，另一个是 MSI-X Capability 结构。通常情况下一个 PCIe 设备仅包含一种结构，或者为 MSI Capability 结构，或者为 MSI-X Capability 结构。

10.1.1 MSI Capability 结构

MSI Capability 结构共有四种组成方式，分别是 32 和 64 位的 Message 结构，32 位和 64 位带中断 Masking 的结构。MSI 报文可以使用 32 位地址或者 64 位地址，而且可以使用 Masking 机制使能或者禁止某个中断源。MSI Capability 寄存器的结构如图 10-1 所示。

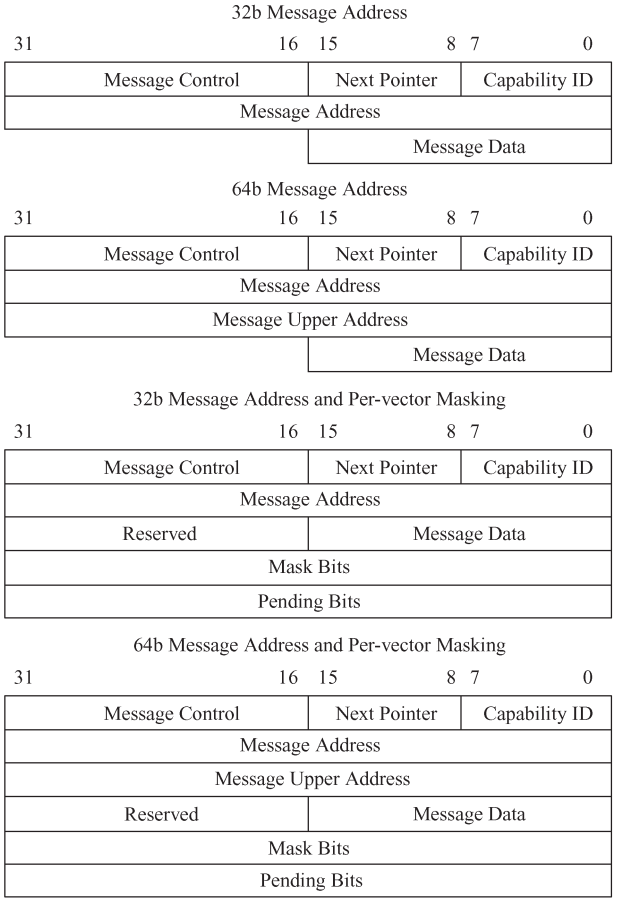

图 10-1 MSI Capability 结构

- Capability ID 字段记载 MSI Capability 结构的 ID 号，其值为 0x05。在 PCIe 设备中，每一个 Capability 结构都有唯一的 ID 号。
- Next Pointer 字段存放下一个 Capability 结构的地址。
- Message Control 字段。该字段存放当前 PCIe 设备使用 MSI 机制进行中断请求的状态与控制信息，如表 10-1 所示。

表 10-1　MSI Cabalibities 结构的 Message Control 字段

Bits	定　义	描　述
15:9	Reserved	保留位。系统软件读取该字段时将返回全零，对此字段写无意义
8	Per-vector Masking Capable	该位为 1 时，表示支持带中断 Masking 的结构；如果为 0，表示不支持带中断 Masking 的结构。该位对系统软件只读，在 PCIe 设备初始化时设置
7	64 bit Address Capable	该位为 1 时，表示支持 64 位地址结构；如果为 0，表示只能支持带 32 位地址结构。该位对系统软件只读，在 PCIe 设备初始化时设置
6:4	Multiple Message Enable	该字段可读写，表示软件分配给当前 PCIe 设备的中断向量数目。系统软件根据 Multiple Message Capable 字段的大小确定该字段的值。在系统的中断向量资源并不紧张时，Multiple Message Capable 字段和该字段的值相等；而资源紧张时，该字段的值可能小于 Multiple Message Capable 字段的值
3:1	Multiple Message Capable	该字段对系统软件只读，表示当前 PCIe 设备可以使用几个中断向量号，在不同的 PCIe 设备中该字段的值不同。当该字段为 0b000 时，表示 PCIe 设备可以使用 1 个中断向量；为 0b001、0b010、0b011、0b100 和 0b101 时，表示使用 4、8、16 和 32 个中断向量；而 0b110 和 0b111 为保留位。该字段与 Multiple Message Enable 字段的含义不同，该字段表示当前 PCIe 设备支持的中断向量个数，而 Multiple Message Enable 字段是系统软件分配给 PCIe 设备实际使用的中断向量个数
0	MSI Enable	该位可读写，是 MSI 中断机制的使能位。该位为 1 而且 MSI-X Enable 位为 0 时，当前 PCIe 设备可以使用 MSI 中断机制，此时 Legacy 中断机制被禁止。一个 PCIe 设备的 MSI Enable 和 MSI-X Enable 位都被禁止时，将使用 INTx 中断消息报文发出/结束中断请求[㊀]

- Message Address 字段。当 MSI Enable 位有效时，该字段存放 MSI 存储器写事务的目的地址的低 32 位。该字段的 31:2 字段有效，系统软件可以对该字段进行读写操作；该字段的第 1 ~0 位为 0。
- Message Upper Address 字段。如果 64 bit Address Capable 位有效，该字段存放 MSI 存储器写事务的目的地址的高 32 位。
- Message Data 字段，该字段可读写。当 MSI Enable 位有效时，该字段存放 MSI 报文使用的数据。该字段保存的数值与处理器系统相关，在 PCIe 设备进行初始化时，处理器将初始化该字段，而且不同的处理器填写该字段的规则并不相同。如果 Multiple Message Enable 字段不为 0b000 时（即该设备支持多个中断请求时），PCIe 设备可以通过改变 Message Data 字段的低位数据发送不同的中断请求。
- Mask Bits 字段。PCIe 总线规定当一个设备使用 MSI 中断机制时，最多可以使用 32 个中断向量，从而一个设备最多可以发送 32 种中断请求。Mask Bits 字段由 32 位组成，其中每一位对应一种中断请求。当相应位为 1 时表示对应的中断请求被屏蔽，为 0 时表示允许该中断请求。系统软件可读写该字段，系统初始化时该字段为全 0，表示允许所有中断请求。该字段和 Pending Bits 字段对于 MSI 中断机制是可选字段，但是 PCIe 总线规范强烈建议所有 PCIe 设备支持这两个字段。
- Pending Bits 字段。该字段对于系统软件是只读位，PCIe 设备内部逻辑可以改变该字段的值。该字段由 32 位组成，并与 PCIe 设备使用的 MSI 中断一一对应。该字段需要

㊀ 此时 PCI 设备配置空间 Command 寄存器的“Interrupt Disable”位为 1。

与 Mask Bits 字段联合使用。

当 Mask Bits 字段的相应位为 1 时，如果 PCIe 设备需要发送对应的中断请求，Pending Bits 字段的对应位将被 PCIe 设备的内部逻辑置 1，此时 PCIe 设备并不会使用 MSI 报文向中断控制器提交中断请求；当系统软件将 Mask Bits 字段的相应位从 1 改写为 0 时，PCIe 设备将发送 MSI 报文向处理器提交中断请求，同时将 Pending Bit 字段的对应位清零。在设备驱动程序的开发中，有时需要联合使用 Mask Bits 和 Pending Bits 字段防止处理器丢弃中断请求[㊀]。

10.1.2 MSI-X Capability 结构

MSI-X Capability 中断机制与 MSI Capability 的中断机制类似。PCIe 总线引出 MSI-X 机制的主要目的是为了扩展 PCIe 设备使用中断向量的个数，同时解决 MSI 中断机制要求使用中断向量号连续所带来的问题。

MSI 中断机制最多只能使用 32 个中断向量，而 MSI-X 可以使用更多的中断向量。目前 Intel 的许多 PCIe 设备支持 MSI-X 中断机制。与 MSI 中断机制相比，MSI-X 机制更为合理。首先 MSI-X 可以支持更多的中断请求，但这并不是引入 MSI-X 中断机制最重要的原因。因为对于多数 PCIe 设备，32 种中断请求已经足够了。而引入 MSI-X 中断机制的主要原因是，使用该机制不需要中断控制器分配给该设备的中断向量号连续。

如果一个 PCIe 设备需要使用 8 个中断请求且使用 MSI 机制时，Message Data 的[2:0]字段可以为 0b000 ~ 0b111，因此可以发送 8 个中断请求，但是这 8 个中断请求的 Message Data 字段必须连续。在许多中断控制器中，Message Data 字段连续也意味着中断控制器需要为这个 PCIe 设备分配 8 个连续的中断向量号。

有时在一个中断控制器中，虽然具有 8 个以上的中断向量号，但是很难保证这些中断向量号是连续的。因此中断控制器将无法为这些 PCIe 设备分配足够的中断请求，此时该设备的“Multiple Message Enable”字段将小于“Multiple Message Capable”。

而使用 MSI-X 机制可以合理解决该问题。在 MSI-X Capability 结构中，每一个中断请求都使用独立的 Message Address 字段和 Message Data 字段，从而中断控制器可以更加合理地为该设备分配中断资源。

与 MSI Capability 寄存器相比，MSI-X Capability 寄存器使用一个数组存放 Message Address 字段和 Message Data 字段，而不是将这两个字段放入 Capability 寄存器中，本书将这个数组称为 MSI-X Table。从而当 PCIe 设备使用 MSI-X 机制时，每一个中断请求可以使用独立的 Message Address 字段和 Message Data 字段。

除此之外 MSI-X 中断机制还使用了独立的 Pending Table 表，该表用来存放与每一个中断向量对应的 Pending 位。这个 Pending 位的定义与 MSI Capability 寄存器的 Pending 位类似。MSI-X Table 和 Pending Table 存放在 PCIe 设备的 BAR 空间中。MSI-X 机制必须支持这个 Pending Table，而 MSI 机制的 Pending Bits 字段是可选的。

1. MSI-X Capability 结构

MSI-X Capability 结构比 MSI Capability 结构复杂一些。在该结构中，使用 MSI-X Table 存

㊀ MSI 机制提交中断请求的方式类似于边界触发方式，而使用边界触发方式时，处理器可能会丢失某些中断请求，因此在设备驱动程序的开发过程中，可能需要使用这两个字段。

放该设备使用的所有 Message Address 和 Message Data 字段，这个表格存放在该设备的 BAR 空间中，从而 PCIe 设备可以使用 MSI-X 机制时，中断向量号可以不连续，也可以申请更多的中断向量号。MSI-X Capability 结构的组成方式如图 10-2 所示。

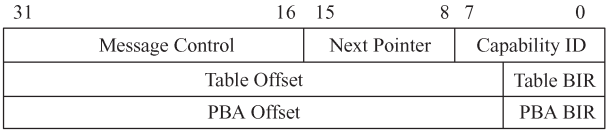

图 10-2　MSI-X Capability 结构的组成方式

上图中各字段的含义如下所示。

- Capability ID 字段记载 MSI-X Capability 结构的 ID 号，其值为 0x11。在 PCIe 设备中，每个 Capability 都有唯一的 ID 号。
- Next Pointer 字段存放下一个 Capability 结构的地址。
- Message Control 字段，该字段存放当前 PCIe 设备使用 MSI-X 机制进行中断请求的状态与控制信息，如表 10-2 所示。

表 10-2　MSI-X Capability 结构的 Message Control 字段

Bits	定　义	描　述
15	MSI-X Enable	该位可读写，是 MSI-X 中断机制的使能位，复位值为 0，表示不使能 MSI-X 中断机制。该位为 1 且 MSI Enable 位为 0 时，当前 PCIe 设备使用 MSI-X 中断机制，此时 INTx 和 MSI 中断机制被禁止。当 PCIe 设备的 MSI Enble 和 MSI-X Enable 位为 0 时，将使用 INTx 中断消息报文发出/结束中断请求
14	Function Mask	该位可读写，是中断请求的全局 Mask 位，复位值为 0。如果该位为 1，该设备所有的中断请求都将被屏蔽；如果该位为 0，则由 Per Vector Mask 位决定是否屏蔽相应的中断请求。Per Vector Mask 位在 MSI-X Table 中定义，详见下文
10:0	Table Size	MSI-X 中断机制使用 MSI-X Table 存放 Message Address 字段和 Message Data 字段。该字段用来存放 MSI-X Table 的大小，该字段对系统软件只读

- Table BIR（BAR Indicator Register）。该字段存放 MSI-X Table 所在的位置，PCIe 总线规范规定 MSI-X Table 存放在设备的 BAR 空间中。该字段表示设备使用 BAR0 ~5 寄存器中的哪个空间存放 MSI-X table。该字段由三位组成，其中 0b000 ~0b101 与 BAR0 ~5 空间一一对应。
- Table Offset 字段。该字段存放 MSI-X Table 在相应 BAR 空间中的偏移。
- PBA（Pending Bit Array）BIR 字段。该字段表示 Pending Table 存放在 PCIe 设备的哪个 BAR 空间中，0 表示 BAR0 空间，1 表示 BAR1 空间，依此类推。在通常情况下，Pending Table 和 MSI-X Table 存放在 PCIe 设备的同一个 BAR 空间中。
- PBA Offset 字段。该字段存放 Pending Table 在相应 BAR 空间中的偏移。

2. MSI-X Table

MSI-X Table 的组成结构如图 10-3 所示。

由该图可见，MSI-X Table 由多个 Entry 组成，其中每个 Entry 与一个中断请求对应。每个 Entry 中有四个参数，其含义如下所示。

DWORD 3	DWORD 2	DWORD 1	DWORD 0	
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	Entry 0
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	Entry 1
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	Entry 2
...	...	...	...	...
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	Entry N-1

图 10-3　MSI-X Table 的组成结构

- Msg Addr。当 MSI-X Enable 位有效时，该字段存放 MSI-X 存储器写事务的目的地址的低 32 位。该双字的 31:2 字段有效，系统软件可读写；1:0 字段复位时为 0，PCIe 设备可以根据需要将这个字段设为只读，或者可读写。不同的处理器填入该寄存器的数据并不相同。
- Msg Upper Addr，该字段可读写，存放 MSI-X 存储器写事务的目的地址的高 32 位。
- Msg Data，该字段可读写，存放 MSI-X 报文使用的数据。其定义与处理器系统使用的中断控制器和 PCIe 设备相关。
- Vector Control，该字段可读写。该字段只有第 0 位（即 Per Vector Mask 位）有效，其他位保留。当该位为 1 时，PCIe 设备不能使用该 Entry 提交中断请求；为 0 时可以提交中断请求。该位在复位时为 0。Per Vector Mask 位的使用方法与 MSI 机制的 Mask 位类似。

3. Pending Table

Pending Table 的组成结构如图 10-4 所示。

63	0
Pending Bits 0 Through 63	QWORD 0
Pending Bits 64 Through 127	QWORD 1
...	...
Pending Bits(N-1 div 64)×64 Through N-1	QWORD((N-1) div 64)

图 10-4　Pending Table 的组成结构

如上图所示，在 Pending Table 中，一个 Entry 由 64 位组成，其中每一位与 MSI-X Table 中的一个 Entry 对应，即 Pending Table 中的每一个 Entry 与 MSI-X Table 的 64 个 Entry 对应。与 MSI 机制类似，Pending 位需要与 Per Vector Mask 位配置使用。

当 Per Vector Mask 位为 1 时，PCIe 设备不能立即发送 MSI-X 中断请求，而是将对应的 Pending 位置 1；当系统软件将 Per Vector Mask 位清零时，PCIe 设备需要提交 MSI-X 中断请求，同时将 Pending 位清零。

10.2　PowerPC 处理器如何处理 MSI 中断请求

PowerPC 处理器使用 OpenPIC 中断控制器或者 MPIC 中断控制器，处理外部中断请求。其中 MPIC 中断控制器基于 OpenPIC 中断控制器，但是做出了许多增强，目前 Freescale 新推出的 PowerPC 处理器，其中断控制器多与 MPIC 兼容。

值得注意的是，PowerPC 处理器和 x86 处理器处理 MSI 报文的方式有较大的不同。其中

x86 处理器使用的机制比 PowerPC 处理器更为合理，但是 PowerPC 处理器的方法使用的硬件资源相对较少。本节将 MPC8572 处理器为例说明 MSI 机制的处理过程，在第 10.3 节介绍 x86 处理器如何实现 MSI 机制。

MPIC 中断控制器是 Freescale 的 PowerPC 处理器使用的通用中断控制器，目前基于 E500 内核的处理器，如 MPC854x、8572 等处理器使用这种中断控制器。目前 Freescale 使用 QorIP 架构，该架构使用的中断控制器与 MPIC 兼容。

使用 MPIC 中断控制器处理 MSI 中断时，PCIe 设备的 MSI 报文，其目的地址为 MPIC 中断控制器的 MSIIR 寄存器。当该寄存器被 PCIe 设备写入后，MPIC 中断控制器将向处理器内核提交中断请求，之后处理器再通过读取 MPIC 中断控制器的 ACK 寄存器获得中断向量号，并进行相应的中断处理。这种方式与 x86 处理器的 FSB Interrupt Message 机制相比，处理器需要读取 ACK 寄存器，从而中断处理的延时较大。

目前 Freescale 的 P4080 处理器对 MPIC 中断控制器进行了优化。在 P4080 处理器中，MPIC 中断控制器向处理器提交中断请求的同时，也向处理器内核提交中断向量，处理器内核不必读取 ACK 寄存器获得中断向量，从而缩短了中断处理延时。使用这种方法的效率与 x86 处理器使用的 FSB Interrupt Message 机制相当。

目前 Freescale 并没有完全公开 P4080 处理器的实现细节，因此本节仍以 MPC8572 处理器为例介绍 PCIe 设备的 MSI 中断请求。在 MPC8572 处理器中，MPIC 中断控制器的拓扑结构如图 10-5 所示。

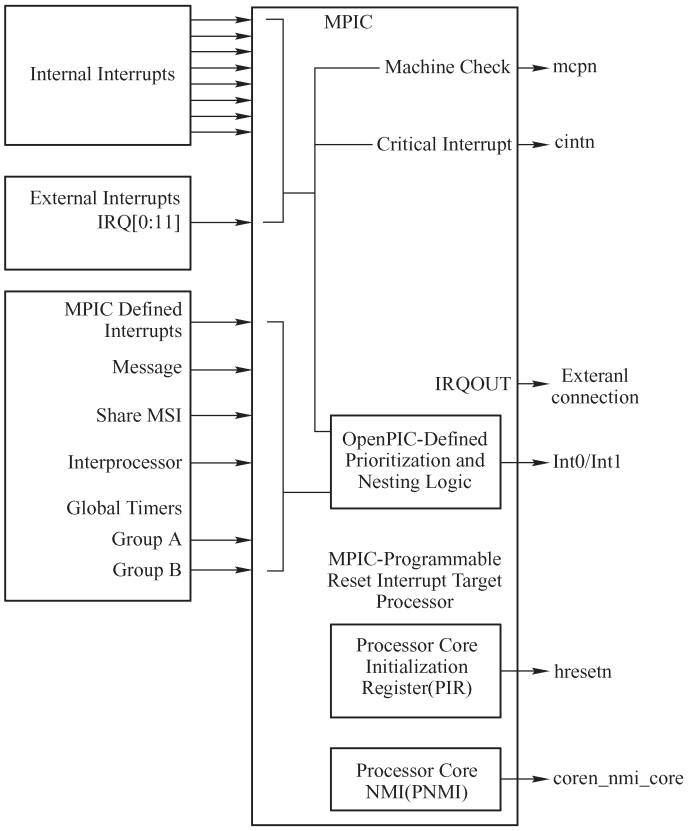

图 10-5　MPIC 中断控制器的拓扑结构

由上图所示，MPIC 中断控制器可以处理内部中断请求[㊀]、外部中断请求，Message、处理器间中断请求和 Share MSI 中断请求等。而 MPIC 中断控制器使用 Int0、Int1 等中断线向处理器提交这些中断请求。其中 Internal Interrupts 和 External Interrupts 模块处理 MPC8572 内部和外部的中断请求，而 Share MSI 处理来自 PCIe 设备的 MSI 或者 MSI-X 中断请求。

当 MPIC 中断控制器收到 MSI 报文后，将使用中断线 Int0、Int1 或者 cintn 向处理器内核提交中断请求。处理器内核被中断后，将读取 ACK 寄存器获得中断向量，然后执行相应的中断服务例程。为此 PowerPC 处理器设置了一系列寄存器，如下文所示。

10.2.1 MSI 中断机制使用的寄存器

PowerPC 处理器设置了一系列寄存器，处理来自 PCIe 设备的 MSI 报文，其中最重要的寄存器是 MSIIR 寄存器。在 PowerPC 处理器系统中，PCIe 设备 Message Address 寄存器中存放的值都为 MSIIR 寄存器的物理地址，而 Message Data 寄存器中存放的数据也与 MSIIR 寄存器相关。

在 PowerPC 处理器系统中，MSI 机制的实现过程是 PCIe 设备向 MSIIR 寄存器写入指定的数据。MPIC 中断控制器发现该寄存器被写入后，将向处理器提交中断请求。处理器收到这个中断请求后，将通过读取 MPIC 中断控制器的 ACK 寄存器确定中断向量，并依此确定中断源。为此 PowerPC 处理器还设置了其他寄存器实现 MSI 中断机制。

1. MSIIR 寄存器

在 PowerPC 处理器中，MSIIR（Shared Message Signaled Interrupt Index Register）寄存器是实现 MSI 机制的重要寄存器。

当 PCIe 设备对 MSIIR 寄存器进行写操作时，MPC8572 处理器将使能 MSIR0-MSIR7 寄存器的相应位，从而向 MPIC 中断控制器提交中断请求，而中断控制器将转发这个中断请求，由处理器进一步处理。该寄存器各字段的详细描述如表 10-3 所示。

表 10-3 MSIIR 寄存器

Bits	定　义	描　述
27 ~ 31	IBS	该字段用来选择 MSIR0 ~ MSIR7 寄存器的对应位。0b00000 对应 SH0；0b00001 对应 SH1；0b00010 对应 SH2；以此类推 0b11111 对应 SH31
24 ~ 26	SRS	该字段用来选择 MSIR0 ~ MSIR7 寄存器。0b000 对应 MSIR0；0b001 对应 MSIR1；0b010 对应 MSIR2；以此类推 0b111 对应 MSIR7
0 ~ 24		保留。

PCIe 设备通过 MSI 机制，向此寄存器写入数据时，MSIR0 ~ 7 寄存器的相应位 SH0 ~ 31 将有一位置 1。例如 PCIe 设备向 MSIIR 寄存器写入 0xFF00000 时，MSIR7 寄存器的 SH31 位将置 1（SRS 字段为 0b111 用来选择 MSIR7，而 IBS 字段为 0b11111 用来选择 SH31）。

2. MSIR 寄存器组

MSIR（Shared Message Signaled Interrupt Registers）寄存器组共由 8 个寄存器组成，分别

㊀ PowerPC 处理器中含有许多模块，如千兆以太网、ATM 等，这些模块包含在芯片内部，由这些内部模块发起的中断请求，被称为内部中断请求。

为 MSIR0 ~ MSIR7。其中每一个 MSIRx 寄存器中有 32 个有效位，分别为 SH0 ~ 31。当 PCIe 设备对 MSIIR 寄存器进行写操作时，某一个 MSIIRx 寄存器的某个 SH 位将被置为有效。系统软件通过读取该寄存器获得中断源，该寄存器读清除，对此寄存器进行写操作没有意义。

该寄存器组的大小决定了一个 PowerPC 处理器支持的 MSI 中断请求的个数。在 MPC8572 处理器中，有 8 个 MSIRx 寄存器，每个寄存器由 32 个有效位组成，因此 MPC8572 处理器最多能够处理 256 个 MSI 中断请求。该寄存器的结构如图 10-6 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SH31	SH30	SH29	SH28	SH27	SH26	SH25	SH24	SH23	SH22	SH21	SH20	SH19	SH18	SH17	SH16
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
SH15	SH14	SH13	SH12	SH11	SH10	SH9	SH8	SH7	SH6	SH5	SH4	SH3	SH2	SH1	SH0

图 10-6　MSIRx 寄存器的结构

3. MSISR 寄存器

MSISR 寄存器（Shared Message Signaled Interrupt Status Register）共由 8 个有效位组成，每一位对应一个 MSIR 寄存器。MPC8572 处理器设置该寄存器的主要目的是方便系统软件定位究竟是哪个 MSIR 寄存器中存在有效的中断请求。首先系统软件通过 MSISR 寄存器判断是哪个 MSIRx 寄存器存在有效请求，之后读取相应的 MSIRx 寄存器，该寄存器各字段的详细描述如表 10-4 所示。

表 10-4　MSISR 寄存器

Bits	定　义	描　述
0 ~ 23		保留
24 ~ 31	Sn	该字段由 8 位组成，每一位与一个 MSIR0 ~ 7 寄存器对应。该位为 0 时表示在 MSIRn 寄存器中没有有效位，即没有中断请求；该位为 1 时表示 MSIRn 寄存器中至少有一个有效位，即存在中断请求。Sn 位是 MSIRn 寄存器各个位的“与”，当 MSIRn 寄存器的相应位清除时，Sn 也将被清除

4. MSIVPR 寄存器组

MSIVPR（Shared Message Signaled Interrupt Vector/Priority Register）寄存器组由 8 个寄存器组成，分别为 MSIVPR0 ~ 7 寄存器。该组寄存器设置对应中断请求的优先级别和中断向量。其中每个 MSIVPR 寄存器对应一个 MSIR 寄存器，MSIVPR 寄存器各字段的详细解释如表 10-5 所示。

表 10-5　MSIVPR 寄存器

Bits	定　义	描　述
0	MSK	该位为 0，且 MSIR 寄存器的对应位为 1 时，则将向中断控制器提交中断请求；如果为 1 屏蔽该中断请求
1	A	该位为 0 时，表示 MPIC 中断控制器没有处理该中断请求；该位为 1 时，表示 MPIC 中断控制器正在处理该中断请求，或者该中断控制器准备处理该中断请求，这个中断请求将在 IPR（Interrupt Pending Regsiter）寄存器中排队等待处理，或者在 ISR（Interrupt Service Register）寄存器中正在被处理。该位的详细描述见 MPC8572 的数据手册
12 ~ 15	PRIORITY	OpenPIC 和 MPIC 中断控制器中为每一个中断请求设置了 0 ~ 15，共 16 个优先级。其中 1 的优先权最低，15 的优先权最高，0 表示禁止中断请求

（续）

Bits	定　义	描　述
16～31	VECTOR	该字段存放该中断的中断向量。当处理器读取 IACK 寄存器时，将获得对应中断请求的中断向量。

通过该组寄存器可以发现，在 MPC8572 处理器系统中，PCIe 设备最多可以使用 8 个中断向量，并可以共享这些中断向量。

5. MSIDR 寄存器组

MSIDR（Shared Message Signaled Interrupt Destination Registers）寄存器组共由 8 个寄存器组成，分别为 MSIDR0～7。其中每一个 MSIDRn 寄存器对应一个 MSIR 寄存器。

MPIC 中断控制器支持 Pass-through 方式，在这种方式下，PowerPC 处理器可以使用外部中断控制器处理中断请求（这种方法极少使用），而不使用内部中断控制器。MPIC 中断控制器可以使用 cint#和 int#信号提交中断请求，但是绝大多数系统软件都使用 int#信号向处理器提交中断请求。

此外在 MPC8572 处理器中有两个 CPU，分别为 CPU0 和 CPU1，MSI 机制提交的中断请求可以由 CPU0 或者 CPU1 处理。系统软件可以通过设置 MSIDRn 寄存器完成这些功能，该寄存器各字段的详细描述如表 10-6 所示。

表 10-6　MSIDRn 寄存器

Bits	定　义	描　述
0	EP	为 1 时，表示中断请求输出到 IRQ_OUT 由外部中断控制器处理；为 0 时，表示由 MPIC 中断控制器处理
1	CI0	为 1 时，表示中断控制器使用 cint#信号向 CPU0 提交中断请求
2	CI1	为 1 时，表示中断控制器使用 cint#信号向 CPU1 提交中断请求
30	P1	为 1 时，表示中断控制器使用 int#信号向 CPU0 提交中断请求
31	P0	为 1 时，表示中断控制器使用 int#信号向 CPU1 提交中断请求

10.2.2　系统软件如何初始化 PCIe 设备的 MSI Capability 结构

如果 PCIe 设备支持 MSI 机制，系统软件首先设置该设备 MSI Capability 结构的 Message Address 和 Message Data 字段。如果该 PCIe 设备支持 64 位地址空间，即 MSI Capability 寄存器的 64 bit Address Capable 位有效时，系统软件还需要设置 Message Upper Address 字段。系统软件完成这些设置后，将置 MSI Capability 结构的 MSI Enable 位有效，使能该 PCIe 设备的 MSI 机制。

其中 Message Address 字段所填写的值是 MSIIR 寄存器在 PCI 总线域中的物理地址。在 PowerPC 处理器中，PCI 总线域与存储器域地址空间独立，当 PCIe 设备访问存储器域的地址空间时，需要通过 Inbound 寄存器组将 PCI 总线域地址空间转换为存储器域地址空间。

在 PowerPC 处理器中，PCIe 设备使用 MSI 机制访问 MSIIR 寄存器时，可以不使用 Inbound 寄存器组进行 PCI 总线地址到处理器地址的转换。在 MPC8572 处理器中，专门设置了一个 PEXCSRBAR 窗口[㊀]，进行 PCI 总线域到存储器域的地址转换，使用这种方法可以节省

㊀ 该窗口的大小为 1 MB，其基地址由 PEXCSRBAR 寄存器确定。

Inbound 寄存器窗口，Linux PowerPC 使用了这种实现方式。

在 MPC8572 处理器中，MSIIR 寄存器的基地址为 CCSRBAR[㊀]（Configuration，Control，and Status Base Address Register），其偏移为 0x1740。为支持 MSI 中断机制，系统软件需要使用 PEXCSRBAR 窗口将 MSIIR 寄存器映射到 PCI 总线域地址空间，即将 CCSRBAR 寄存器空间映射到 PCI 总线域地址空间。之后 PCIe 设备就可以通过 MSIIR 寄存器在 PCI 总线域的地址访问 MSIIR 寄存器。

Linux PowerPC 使用 setup_pci_pcsrbar 函数[㊁]设置 PEXCSRBAR 窗口，该函数的源代码在 ./arch/powerpc/sysdev/fsl_pci.c 文件中，如源代码 10-1 所示，这段代码来自 Linux 2.6.30.5。

源代码 10-1　setup_pci_pcsrbar 函数

```
static void __init setup_pci_pcsrbar (struct pci_controller * hose)
{
#ifdef CONFIG_PCI_MSI
    phys_addr_t immr_base;

    immr_base = get_immrbase ();
    early_write_config_dword (hose, 0, 0, PCI_BASE_ADDRESS_0, immr_base);
#endif
}
```

系统软件除了需要设置 PCIe 设备的 Message Address 字段和 PEXCSRBAR 窗口之外，还需要设置 PCIe 设备的 Message Data 字段。PCIe 设备向 MSIIR 寄存器进行存储器写操作的数据存放在 Message Data 字段中。

系统软件在初始化 Message Data 字段之前，首先根据 Multiple Message Capable 字段预先存放的数据初始化 Multiple Message Enable 字段。一个 PCIe 设备最多可以申请 32 个中断请求，但是系统软件根据当前处理器系统的中断资源的使用情况，决定给这个 PCIe 设备提供多少个中断向量，并将这个结果存放到 Multiple Message Enable 字段。

MPC8572 处理器最多可以为 PCIe 设备提供 256 个 MSI 中断请求。但是在某些极端的情况下，可能会出现 PCIe 设备需要的中断请求超过系统所能提供的中断请求。此时某些 PCIe 设备的 Multiple Message Enable 字段可能会小于 Multiple Message Capable 字段。

如果在 PCIe 设备中，使用了多个中断请求，那么 Message Data 字段存放的是一组中断向量号，而 Message Data 字段存放这组中断向量号的基地址。MSI 机制要求“这组数据”连续，其范围在 Message Data ~ Message Data + Multiple Message Enable-1 之间。在多数情况下，MPC8572 处理器系统仅为一个 PCIe 设备分配 1 个中断向量号。

由上所述，在 MPC8572 处理器系统中，PCIe 设备使用存储器写 TLP 传送 MSI 中断报文，这个存储器写 TLP 使用的地址为 PCIe 设备 Capability 结构的 Message Address 字段，而

㊀ 在 Linux PowerPC 中使用 immr_base 变量保存该寄存器。IMMR 寄存器是 PQ2 处理器使用的寄存器，该寄存器在 PQ3 之后的处理器中升级为 CCSRBAR。

㊁ 该函数来自 Linux 2.6.30.5 内核。

数据为 Message Data ~ Message Data + Multiple Message Enable-1 之间。其中 Message Data 字段与 MSIIR 寄存器要求的格式相同。

这个特殊的存储器写 TLP 报文通过若干 Switch，并穿越 RC 后，最终将数据写入 MSIIR 寄存器中，并设置 MSIIR 寄存器的 SRS 和 IBS 字段，同时将使能 MSIR0 ~ MSIR7 寄存器的相应位，从而向中断控制器提交中断请求（如果 MSIVPR 寄存器的 MSK 位为 1）。MPIC 中断控制器获得该中断请求后，向处理器系统转发这个中断请求，并由处理器系统执行相应的中断服务例程进行中断处理。MPC8572 处理器也可以处理 PCIe 设备的 MSI-X 中断机制，本节对此不做进一步介绍。

10.3 x86 处理器如何处理 MSI-X 中断请求

PCIe 设备发出 MSI-X 中断请求的方法与发出 MSI 中断请求的方法类似，都是向 Message Address 所在的地址写 Message Data 字段包含的数据。只是 MSI-X 中断机制为了支持更多的中断请求，在 MSI-X Capablity 结构中存放了一个指向一组 Message Address 和 Message Data 字段的指针，从而一个 PCIe 设备可以支持的 MSI-X 中断请求数目大于 32 个，而且并不要求中断向量号连续。MSI-X 机制使用的这组 Message Address 和 Message Data 字段存放在 PCIe 设备的 BAR 空间中，而不是在 PCIe 设备的配置空间中，从而可以由用户决定使用 MSI-X 中断请求的数目。

当系统软件初始化 PCIe 设备时，如果该 PCIe 设备使用 MSI-X 机制传递中断请求，需要对 MSI-X Capability 结构指向的 Message Address 和 Message Data 字段进行设置，并使能 MSI-X Enable 位。x86 处理器在此处的实现与 PowerPC 处理器有较大的不同。

10.3.1 Message Address 字段和 Message Data 字段的格式

在 x86 处理器系统中，PCIe 设备也是通过向 Message Address 写入 Message Data 指定的数值实现 MSI/MSI-X 机制。在 x86 处理器系统中，PCIe 设备使用的 Message Adress 字段和 Message Data 字段与 PowerPC 处理器不同。

1. PCIe 设备使用 Message Adress 字段

在 x86 处理器系统中，PCIe 设备使用的 Message Address 字段仍然保存 PCI 总线域的地址，其格式如图 10-7 所示。

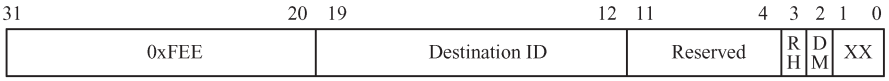

图 10-7 Message Address 字段的格式

其中第 31 ~ 20 位存放 FSB Interrupts 存储器空间的基地址，其值为 0xFEE。当 PCIe 设备对 0xFEEX-XXXX 这段“PCI 总线域”的地址空间进行写操作时，MCH/ICH 会首先进行“PCI 总线域”到“存储器域”的地址转换，之后将这个写操作翻译为 FSB 总线的 Interrupt Message 总线事务，从而向 CPU 内核提交中断请求。

x86 处理器使用 FSB Interrupt Message 总线事务转发 MSI/MSI-X 中断请求。使用这种方法的优点是向 CPU 内核提交中断请求的同时，提交 PCIe 设备使用的中断向量，从而 CPU 不

需要使用中断响应周期从寄存器中获得中断向量。FSB Interrupt Message 总线事务的详细说明见下文。

Message Address 字段其他位的含义如下所示。

- Destination ID 字段保存目标 CPU 的 ID 号，目标 CPU 的 ID 与该字段相等时，目标 CPU 将接收这个 Interrupt Message。FSB Interrupt Message 总线事务可以向不同的 CPU 提交中断请求。
- RH（Redirection Hint Indication）位为 0 时，表示 Interrupt Message 将直接发向与 Destination ID 字段相同的目标 CPU；如果 RH 为 1 时，将使能中断转发功能。
- DM（Destination Mode）位表示在传递优先权最低的中断请求时，Destination ID 字段是否被翻译为 Logical 或者 Physical APIC ID。在 x86 处理器中 APIC ID 有三种模式，分别为 Physical、Logical 和 Cluster ID 模式。
- 如果 RH 位为 1 且 DM 位为 0 时，Destination ID 字段使用 Physical 模式；如果 RH 位为 1 且 DM 位为 1，Destination ID 字段使用 Logical 模式；如果 RH 位为 0，DM 位将被忽略。

以上这些字段的描述与 x86 处理器使用的 APIC 中断控制器相关。对 APIC 的详细说明超出了本书的范围，对此部分感兴趣的读者请参阅 Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A：System Programming Guide，Part 1。

2. Message Data 字段

Message Data 字段的格式如图 10-8 所示。

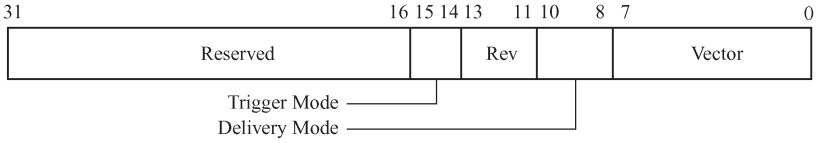

图 10-8　Message Data 字段的格式

Trigger Mode 字段为 0b0x 时，PCIe 设备使用边沿触发方式申请中断；为 0b10 时使用低电平触发方式；为 0b11 时使用高电平触发方式。MSI/MSI-X 中断请求使用边沿触发方式，但是 FSB Interrupt Message 总线事务还支持 Legacy INTx 中断请求方式，因此在 Message Data 字段中仍然支持电平触发方式。但是对于 PCIe 设备而言，该字段为 0b0x。

Vector 字段表示这个中断请求使用的中断向量。FSB Interrupt Message 总线事务在提交中断请求的同时，将中断向量也通知给处理器。因此使用 FSB Interrupt Message 总线事务时，处理器不需要使用中断响应周期通过读取中断控制器获得中断向量号。与 PowerPC 的传统方式相比，x86 处理器的这种中断请求的效率较高[㊀]。

值得注意的是，在 x86 处理器中，MSI 机制使用的 Message Data 字段与 MSI-X 机制相同。但是当一个 PCIe 设备支持多个 MSI 中断请求时，其 Message Data 字段必须是连续的，因而其使用的 Vector 字段也必须是连续的，这也是在 x86 处理器系统中，PCIe 设备支持多个 MSI 中断请求的问题所在，而使用 MSI-X 机制有效避免了该问题。

㊀ P4080 处理器也提供了一种类似于 FSB Interrupt Message 总线事务的中断请求方法。

Delivery Mode 字段表示如何处理来自 PCIe 设备的中断请求。

- 该字段为 0b000 时，表示使用“Fixed Mode”方式。此时这个中断请求将被 Destination ID 字段指定的 CPU 处理。
- 该字段为 0b001 时，表示使用“Lowest Priority”方式。此时这个中断请求将被优先权最低的 CPU 处理。当使用“Fixed Mode”和“Lowest Priority”方式时，如果 Vector 字段有效，CPU 接收到这个中断请求之后，将使用 Vector 字段指定的中断向量处理这些中断请求；而当 Delivery Mode 字段为其他值时，Message Data 字段中所包含的 Vector 字段无效。
- 该字段为 0b010 时，表示使用 SMI 方式传递中断请求，而且必须使用边沿触发，此时 Vector 字段必须为 0。这个中断请求将被 Destination ID 字段指定的 CPU 处理。
- 该字段为 0b100 时，表示使用 NMI 方式传递中断请求，而且必须使用边沿触发，此时 Vector 字段和 Trigger 字段的内容将被忽略。这个中断请求将被 Destination ID 字段指定的 CPU 处理。
- 该字段为 0b101 时，表示使用 INIT 方式传递中断请求，Vector 字段和 Trigger 字段的内容将被忽略。这个中断请求将被 Destination ID 字段指定的 CPU 处理。
- 该字段为 0b111 时，表示使用 INTR 信号传递中断请求且使用边沿触发。此时 MSI 中断信息首先传递给中断控制器，然后中断控制器通过 INTR 信号向 CPU 传递中断请求，之后 CPU 通过中断响应周期获得中断向量。上文中 PowerPC 处理器使用的方法与此方法类似。而在 x86 处理器中多使用 Interrupt Message 总线事务进行 MSI 中断信息的传递，因此这种模式很少使用。

边沿触发和电平触发是中断请求常用的两种方式。其中电平触发指外部设备使用逻辑电平 1（高电平触发）或者 0（低电平触发），提交中断请求。使用电平或者边沿方式提交中断请求时，外部设备一般通过中断线（IRQ_PIN#）与中断控制器相连，其中多个外部设备可能通过相同的中断线与中断控制器相连（线与或者与门）。

外部设备在使用低电平触发提交中断请求的过程中，首先需要将 IRQ_PIN#信号驱动为低。当中断控制器将该中断请求提交给处理器，而且处理器将这个中断请求处理完毕后，处理器将通过写外部设备的某个寄存器来清除此中断源，此时外部设备将不再驱动 IRQ_PIN#信号线，从而结束整个中断请求。

IRQ_PIN#信号线可以被多个外部设备共享，在这种情况之下，只有所有外部设备都不驱动 IRQ_PIN#信号线时，IRQ_PIN#信号才为高电平。采用电平触发方式进行中断请求的优点是不会丢失中断请求，而缺点是一个优先权较高的中断请求有可能会长期占用中断资源，从而使其他优先权较低的中断不能被及时提交。因为优先级别较高的中断源可能会持续不断地驱动 IRQ_PIN#信号。

而边沿触发使用上升沿（0 到 1）或者下降沿（1 到 0）作为触发条件，但是中断控制器并不是使用这个“边沿”作为触发条件。中断控制器使用内部时钟对 IRQ_PIN#信号进行采样，如果在前一个时钟周期，IRQ_PIN#信号为 0，而后一个时钟周期，IRQ_PIN#信号为 1，中断控制器认为外部设备提交了一个有效“上升沿”，中断控制器会锁定这个“上升沿”并向处理器发出中断请求。这也是外部设备至少需要将 IRQ_PIN#信号保持一个时钟采样周期的原因，否则中断控制器可能无法识别本次边沿触发的中断请求，从而产生 Spurious 中断

请求。

外部设备使用“上升沿”进行中断申请时，不需要持续地将 IRQ_PIN#信号驱动为 1，而只需要保证中断控制器可以进行正确采样这些中断信号即可。在处理边沿触发中断请求时，处理器不需要清除中断源。

使用边沿触发可以有效避免“优先级别”较高的中断源长期占用 IRQ_PIN#信号的情况，使用“下降沿”触发进行中断请求与“上升沿”触发类似。

但是外部设备使用边沿触发方式时，有可能会丢失一些中断请求。例如在一个处理器系统中，存在一个定时器，这个定时器使用上升沿触发方式向中断控制器定时提交中断。当处理器正在处理这个定时器的上一个中断请求时，将不会处理这个定时器发出的其他“边沿”中断请求，从而导致中断丢失。而使用电平触发方式不会出现这类问题，因为电平触发方式是一个“持续”过程，处理器只有处理完毕当前中断，并清除相应中断源之后，才会处理下一个中断源。

MSI 中断请求实际上和边沿触发方式非常类似，MSI 中断请求通过存储器写 TLP 实现，这个写动作是一个瞬间的动作，并不是一个持续请求，因此在 x86 处理器中 MSI 中断请求使用边沿触发进行中断请求。

还有一些外部设备可以通过 I/O APIC 进行中断请求[㊀]，这些 I/O APIC 接收的外部中断需要标明是使用边沿或者电平触发，I/O APIC 使用 FSB Interrupt Message 总线事务将中断请求发向 Local APIC，并由 Local APIC 向处理器提交中断请求。

10.3.2 FSB Interrupt Message 总线事务

与 MPC8572 处理器处理 MSI 中断请求不同，x86 处理器使用 FSB 的 Interrupt Message 总线事务，处理 PCIe 设备的 MSI/MSI-X 中断请求。由上文所示，MPC8572 处理器处理 MSI 中断请求时，首先由 MPIC 中断控制器截获这个 MSI 中断请求，之后由 MPIC 中断控制器向 CPU 提交中断请求，而 CPU 通过中断响应周期从 MPIC 中断控制器的 ACK 寄存器中获得中断向量。

采用这种方式的主要问题是，当一个处理器中存在多个 CPU 时，这些 CPU 都需要通过中断响应周期从 MPIC 中断控制器的 ACK 寄存器中获得中断向量。在一个中断较为密集的应用中，ACK 寄存器很可能会成为系统瓶颈。而采用 Interrupt Message 总线事务可以有效地避免这种系统瓶颈，因为使用这种方式中断信息和中断向量将同时到达指定的 CPU，而不需要使用中断响应周期获得中断向量。

x86 处理器也具有通过中断控制器提交 MSI/MSI-X 中断请求的方法，在 I/O APIC 具有一个“The IRQ Pin Assertion Register”寄存器，该寄存器地址为 0xFEC00020[㊁]，其第 4 ~ 0 位存放 IRQ Number。系统软件可以将 PCIe 设备的 Message Address 寄存器设置为 0xFEC00020，将 Meaasge Data 寄存器设置为相应的 IRQ Number。

当 PCIe 设备需要提交 MSI 中断请求时，将向 PCI 总线域的 0xFEC00020 地址写入 Message Data 寄存器中的数据。此时这个存储器写请求将数据写入 I/O APIC 的 The IRQ Pin As-

---

㊀ 与 I/O APIC 的 IRQX#引脚链接的外部设备。

㊁ 该寄存器在存储器域和 PCI 总线域中的地址都为 0xFEC00020。

sertion Register 中，并由 I/O APIC 将这个 MSI 中断请求最终发向 Local APIC，之后再由 Local APIC 通过 INTR#信号向 CPU 提交中断请求。

上述步骤与 MPC8572 处理器传递 MSI 中断的方法类似。在 x86 处理器中，这种方式基本上已被弃用。下面以图 10-9 为例，说明 x86 处理器如何使用 FSB 总线的 Interrupt Message 总线事务，向 CPU 提交 MSI/MSI-X 中断请求。

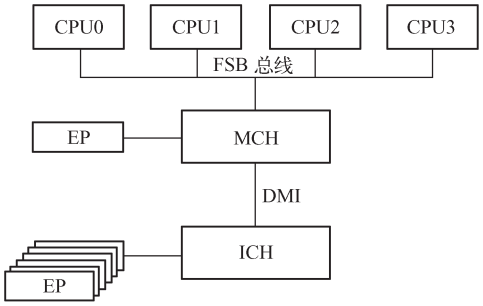

图 10-9　使用 Interrupt Message 总线事务传递 MSI 中断请求

PCIe 设备在发送 MSI/MSI-X 中断请求之前，系统软件需要合理设置 PCIe 设备 MSI/MSI-X Capability 寄存器，使 Message Address 寄存器的值为 0xFEExx00y[⊖]，同时合理地设置 Message Data 寄存器 Vector 字段。

PCIe 设备提交 MSI/MSI-X 中断请求时，需要向 0xFEExx00y 地址写 Message Data 寄存器中包含的数据，并以存储器写 TLP 的形式发送到 RC。如果 ICH 收到这个存储器写 TLP 时，将通过 DMI 接口将这个 TLP 提交到 MCH。MCH 收到这个 TLP 后，发现这个 TLP 的目的地址在 FSB Interrupts 存储器空间中，则将 PCIe 总线的存储器写请求转换为 Interrupt Message 总线事务，并在 FSB 总线上广播。

FSB 总线上的 CPU，根据 APIC ID 信息，选择是否接收这个 Interrupt Message 总线事务，并进入中断状态，之后该 CPU 将直接从这个总线事务中获得中断向量号，执行相应的中断服务例程，而不需要从 APIC 中断控制器获得中断向量。与 PowerPC 处理器的 MPIC 中断控制器相比，这种方法更具优势。

10.4　小结

本章详细描述了 MSI/MSI-X 中断机制的原理，并以 PowerPC 和 x86 两个处理器系统为例说明这两种中断机制实现机制。本章因为篇幅有限，并没有详细讲述这两个处理器使用的中断控制器。而理解这些中断控制器的实现机制是进一步理解 MSI/MSI-X 中断机制的要点。对此部分有兴趣的读者可以继续阅读 MPIC 中断控制器和 APIC 中断控制器的实现机制，以加深对 MSI/MSI-X 中断机制的理解。

设备的中断处理是局部总线的设计难点和重要组成部分，而中断处理的效率直接决定了局部总线的数据传送效率。在一个处理器系统的设计与实现中，中断处理的优化贯彻始终。

⊖ 其中 xx 表示 APIC ID，而 y 为 RH + DM。

第 11 章　PCI/PCIe 总线的序

在一个处理器系统的实现中，开发者需要注意两个问题，一个是 Cache 的一致性（Cache Coherency），另一个是数据的完整性（Data Consistency）。深入理解这两部分内容的过程贯穿处理器体系结构学习的始终。在一个处理器系统的实现过程中，由这两个问题引发的系统错误是最难定位的。因为这两种系统错误的表现形式很难与“Cache 共享一致性”与“序”联系在一起，即便最后定位是这两种原因引发的系统错误，也不容易复现这些系统错误。

本章主要讲述在 PCI/PCIe 总线中，数据传送的“序”与可能出现的死锁。在总线体系结构的设计中需要重点考虑序与死锁的问题，序和死锁一直是系统架构设计的重点所在，也是逻辑性要求较强、最容易出错、系统程序员容易忽略的内容。

所谓“序”是指数据传送的顺序，是保证数据完整性的基础。而死锁是指两个以上的设备在访问临界资源时，相互等待对方释放这些资源，而无法访问这些资源的情况。合理地安排访问“序”是解决死锁的一个有效方法。

在 PCI/PCIe 总线中，序与生产/消费者模型密切相关。生产/消费者模型是一种并发协作模型，PCI/PCIe 设备使用该模型进行数据传递。在 PCI/PCIe 总线中，访问“序”的安排必须保证生产/消费者模型的正确运转，这也意味着在 PCI/PCIe 总线中，数据的传送规则需要与生产/消费者模型一致。

合理安排数据访问的“序”对于一个系统设计是至关重要的，同时也是一个系统设计的基础。对于一个专用系统，生产消费者模型使用的数据缓冲、Flag 和 Status 位在系统中的位置相对固定，以此为基础设计合理的数据访问“序”也许并不困难。

但是对于一个通用处理器系统，数据缓冲、Flag 和 Status 位的在系统中位置并不固定，此时合理安排数据访问“序”需要异常缜密的思维。而 PCI/PCIe 总线针对的就是这样的一个通用处理器系统。

本章将在第 11.3 节介绍 PCI 总线的序，并在第 11.4 节详细介绍 PCIe 总线的序。其中 PCIe 总线的序基于 PCI 总线的序，并适当简化了 PCI 总线的强序模型，补充了 PCIe 总线的“Relaxed Ordering”和 IDO（ID-Base Ordering）模型。

11.1　生产/消费者模型

除了在 PCI/PCIe 总线中，互连网络中的数据传递以及多进程间的数据通信也经常使用生产/消费者模型。生产/消费者模型的正确运行需要一些必要条件，该模型由以下几个基本单元组成。

- 共享数据缓冲。由生产者写入，消费者读取的数据区域。
- 生产者。数据的提供方，生产者需要产生数据，并在消费者将数据读出后，再将数据写入缓冲中。

- 消费者。数据的使用方，消费者将消费数据，并在生产者将数据写入共享缓冲后，再获取数据。
- Flag 位。生产者通过对该位写 1 通知消费者，已经将数据写入缓冲中。消费者通过该位判断数据缓冲是否有效，为 1 表示在数据缓冲中的数据已经被生产者写入；为 0 表示没有被写入。该位由生产者写 1，由消费者清零。
- Status 位。消费者通过对该位写 1 通知生产者，已经将数据从缓冲读出；生产者通过该位判断数据缓冲是否有效，为 1 表示在数据缓冲中的数据已经被消费者读出；为 0 没有读出。该位由消费者写 1，由生产者清零。

生产/消费者模型需要使用两个状态位完成数据的交换。因为生产者和消费者在使用数据缓冲时，需要多种数据状态。这些数据状态如表 11-1 所示。

表 11-1　Flag 和 Status 状态位

Flag	Status	描　　述
0	0	空闲状态，此时数据缓冲为空
1	0	生产者写入数据，但是消费者没有使用该数据
0	1	消费者使用完毕该数据
1	1	正常情况不会出现该状态

11.1.1　生产/消费者的工作原理

在生产/消费者模型初始化时，Flag 和 Status 位都为 0，之后生产者和消费者通过操纵和检测 Flag 和 Status 位，进行数据传递。其中生产者负责将数据填入到数据缓冲中，而消费者负责将数据从缓冲中取出，其实现过程如表 11-2 所示。

表 11-2　生产/消费者的工作流程

	生产者工作流程	消费者工作流程
1	将数据写入缓冲中	查询 Flag 位，直到该位为 1
2	置 Flag 位为 1	置 Flag 位为 0
3	查询 Status 位，直到该位为 1	将数据从缓冲中读出
4	置 Status 位为 0。如果有数据发送，转（1）	置 Status 位为 1，转（1）

由以上描述，可以发现生产者仅轮询 Status 位，在条件允许时将该位清零，而直接对 Flag 位写 1；消费者仅轮询 Flag 位，并在条件允许时将该位清零，而直接对 Status 位写 1。

设计者在使用生产/消费者模型时，需要注意两个竞争条件。

（1）由于生产者对“Flag 位置 1”和“数据写入数据缓冲”可能没有采用相同的路径，因此这两个动作不一定同步进行。当生产者将 Flag 位置 1 时，可能数据没有完全写入到缓冲中。但是在消费者查询 Flag 位已经为 1，并将该位清零之后，数据一定要写入缓冲中，否则消费者将得到无效数据。

（2）消费者对“数据从缓冲读出”和“Status 位置 1”也可能没有采用相同的路径，这两个动作也不一定同步。因此消费者将 Status 位置 1 时，可能数据没有完全从缓冲中读出。

但是在生产者查询 Status 位已经为 1，并将该位清零之后，数据一定要从缓冲读出，否则生产者将会清除缓冲中的有效数据。

由此可见，生产/消费者模型的正确运行是有条件的。为此在一个实际的系统应用中，必须合理安排数据访问的“序”，使得该模型能够正常运转。而且这个“序”需要考虑在一个系统中存在多个生产者、多个消费者和多个数据缓冲的情况。

11.1.2 生产/消费者模型在 PCI/PCIe 总线中的实现

生产/消费者模型在 PCI/PCIe 总线中得到了充分的应用。两个 PCI 主设备之间进行数据传送，或者 PCI 主设备进行 DMA 操作时需要使用该模型。PCI 总线规定，PCI 设备必须按照生产/消费者模型提供的规则访问存储器或者 I/O 资源。采用该模型对于 PCI 设备访问存储器或者 I/O 资源时避免死锁和保证数据完整性至关重要。PCI/PCIe 总线规定了数据访问的顺序，本章在第 11.3 和第 11.4 节将讲述这些内容。

下文以两个 PCI 主设备进行数据交换说明生产/消费者模型的实现机制，假设这两个 PCI 主设备在进行数据交换时使用一个数据缓冲和 Flag 和 Status 这两个状态位。

在 PCI 总线中，数据传送只要遵循合理的顺序，无论生产者、消费者在哪一级 PCI 总线上，数据缓冲和 Flag、Status 状态位在处理器系统的什么位置，都能准确无误地发送和接收数据，保证生产/消费者模型的正确运转。

下面以图 11-1 为例，进一步说明在 PCI 总线中，生产/消费者模型的运转过程。其中消费者与数据缓冲在同一条 PCI 总线上，而 Flag 位、Status 位和生产者在另外一条 PCI 总线上。这两条 PCI 总线之间通过一个 PCI 桥进行连接。在这种处理器系统中，生产/消费者模型的详细运转过程见表 11-3。

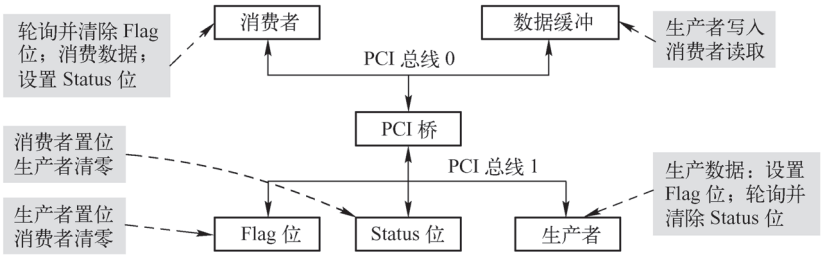

图 11-1 生产消费者模型的实例

PCI 桥的存在为生产/消费者模型在 PCI 体系结构中的实现带来了新的挑战，PCI 桥可以缓存一些报文，因此来自 PCI 设备的报文并不能立即到达目的地。其中 Flag 和 Status 位的初始值为 0。

表 11-3 基于 PCI 总线的生产/消费者模型的详细描述

步　骤	操　作	描　述
1	生产者通过 PCI 桥，使用 Posted 方式将数据写入数据缓冲中	生产者和数据缓冲不在同一条总线上，因此生产者首先使用 Posted 写周期将数据写入 PCI 桥，并不会立刻到达数据缓冲
2	生产者将 Flag 位置 1	Flag 位和生产者在同一条总线上，Flag 位会被立即置位

（续）

步　骤	操　作	描　述
3	生产者轮询 Status 状态位，判断数据缓冲中的数据是否被消费者处理完毕	Status 位和生产者在同一条总线上，生产者可以立即获得 Status 位的信息
4	消费者轮询 Flag 状态位	Flag 位和消费者不在同一条总线上，因此消费者首先将读请求发向 PCI 桥，PCI 桥将使用 Delayed 读总线请求周期获得数据，同时可能使用“Retry”周期结束消费者的读请求。为保证生产/消费者模型的正确运行，PCI 桥必须进行以下操作读取 Flag 位。 1. 为了保证读请求的正确性，PCI 桥需要首先将与读请求相同方向的 Posted 写请求发送出去，即发送到 PCI 总线 1 上。 2. PCI 桥读取 Flag 位。 3. PCI 桥将所有 Posted 写请求发送到 PCI 总线 0 上。 4. PCI 桥将 Flag 信息发送给消费者
5	如果消费者发现 Flag 位被生产者置为 1，则将 Flag 位清零	这个清零操作首先也是发向 PCI 桥，并不会立刻到达 Flag 位
6	消费者从数据缓冲中获得数据	消费者与数据缓冲在同一条总线上，可以直接获得数据
7	消费者处理完毕这些数据后，将 Status 位置 1	消费者和 Status 位不在同一条总线上，因此生产者首先使用 Posted 写周期将数据写入 PCI 桥，并不会立刻更新 Status 位
8	消费者继续轮询 Flag 位，确定数据缓冲中是否有新的数据需要处理	消费者再一次读取 Flag 位时，仍然需要跨越 PCI 桥，此时强制将 PCI 桥中的 Posted 写总线事务刷新出去
9	生产者读取 Status 位，并依此判断消费者是否处理完毕数据缓冲中的数据	生产者和 Status 位在同一条总线上，因此可以立即获得 Status 位的信息
10	如果 Status 位为 1，则生产者将 Status 位清零	生产者和 Status 位在同一条总线上，Status 位可以立即被更新
11	如果生产者继续提供数据，将重新启动整个生产/消费者模型	

由以上过程可以发现，由于 PCI 桥的存在数据并不能立即到达目的地，因此有可能造成总线死锁和数据不完整等一系列问题，最终导致生产/消费者模型不能在 PCI 总线上正确运行。为了解决这个问题，PCI 总线规定了一系列与数据传送“序”有关的规则。

如果 PCI 设备不满足这个“序”的要求，就有可能出现数据完整性的问题，从而导致数据传送失败。如果读者有机会设计基于 PCI/PCIe 总线的设备，需要认真考虑这个序的问题。值得注意的是，PCIe 总线的序与 PCI 总线的序略有区别，在第 11.4 节将详细讨论 PCIe 总线的序。

在 PCI/PCIe 总线中，使用 Posted 总线请求比 Non-Posted 总线请求传送数据的延时更短。因此在一个具体的实现中，如果 Status 位距离生产者的路径较近时，轮询该位的代价较低；同理 Flag 位距离消费者的路径较近时，轮询该位的代价也较低。

值得注意的是，在 PCI 体系结构中，无论生产者、消费者、Flag 和 Status 位在处理器系统的哪级 PCI 总线中，生产/消费者模型都可以正常运行，为此 PCI 总线详细规定了数据传送的顺序，PCI 总线的“序”对于理解 PCI 体系结构至关重要，读者需要重视这部分内容。

本书建议读者改变图 11-1 中生产者、消费者、Flag 和 Status 位的位置，按照第 11.3 节提供的规则，验证生产/消费者协议在 PCI 总线中的正确性。

11.2 PCI 总线的死锁

Tom Shanley 与 Don Anderson 对 PCI 总线的死锁进行了详细说明，本节首先通过两个实例说明在 PCI 总线中存在的死锁问题。其中第 1 个实例是因为缓冲管理不慎而导致的死锁，而第 2 个实例是因为数据传送的顺序而导致死锁。本节更侧重介绍因为数据的传送顺序而导致的死锁。

11.2.1 缓冲管理引发的死锁

如图 11-2 所示，假设在一条 PCI 总线上有两个非桥设备 A 和 B。其中 A 和 B 之间相互进行存储器写操作，这个存储器写操作需要使用设备 A 和 B 内部的缓冲区。而且设备 A/B 的发送部件和接收部件共享同一个数据缓冲。在图 11-2 的左图中 PCI 设备 A/B 的发送和接收缓冲共用了一个数据缓冲，而在右图中将这两者分离。

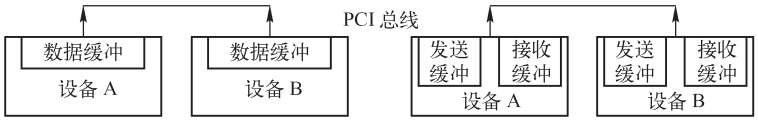

图 11-2 PCI 总线中的死锁实例 1

设备 A 和 B 进行存储器写操作的流程如下。

（1）设备 A 和 B 同时申请 PCI 总线资源，同时 A 和 B 已经将即将发送的数据写入各自的缓冲区中，准备进行数据传递。

（2）设备 A 通过总线仲裁优先使用 PCI 总线，同时将缓冲区中的数据发向设备 B。

（3）设备 B 因为在缓冲区中尚有数据需要传递，因此使用重试周期拒绝了设备 A 的数据请求，并获得 PCI 总线的使用权，将缓冲区中的数据发向设备 A。

（4）设备 A 也因为在缓冲区中尚有数据需要传递，而使用重试周期拒绝了设备 B 的数据请求。

（5）在这种情况下，设备 A 和设备 B 都因为对方的缓冲正在被使用，而无法完成存储器写，从而产生了死锁。

从这个例子中可以发现，如果 PCI 设备对缓冲区的管理不慎，极易造成死锁。我们可以采用一种简单的方法解决这类死锁问题，只要设备 A、B 将接收和发送使用的缓冲分离，这样可以保证在步骤 3 中，设备 B 可以接收来自设备 A 的数据，从而避免了死锁。

11.2.2 数据传送序引发的死锁

本节首先分析一个在 PCI 总线中死锁的实例。假设在 PCI 桥 A 中存放了一个 Posted 写请求，该请求正在准备发向 Secondary PCI 总线。而在 PCI 桥 A 还没有获得 Seconday PCI 总线的使用权时，Secondary 总线上的 PCI 设备 B，需要使用 Delayed 总线事务通过该 PCI 桥从主存储器中读取数据。如图 11-3 所示。

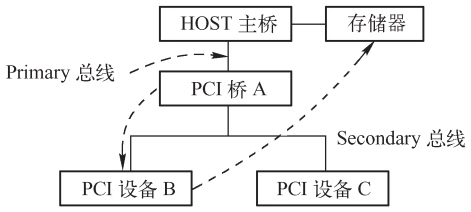

图 11-3　PCI 总线中死锁实例 2

PCI 桥 A 与 PCI 设备 B 的工作流程如下所示。

（1）首先 PCI 设备 B 发出的存储器读请求已经从存储器中获得数据，HOST 主桥将这个存储器读请求转换为存储器读完成。当这个存储器读完成穿越 PCI 桥时，要求刷新 PCI 桥中的 Posted 写请求（从 Primary 总线到 Secondary 总线的 Posted 写请求）。因此 PCI 桥首先需要重试来自 PCI 设备的读完成，然后将存储在 PCI 桥中的这个 Posted 写请求刷新出去，详细原因见第 11.3.1 节。

（2）如果这个 Posted 写请求的目的设备恰好是需要从存储器读取数据的 PCI 设备，此时将可能发生死锁。假设在 PCI 桥 A 暂存的 Posted 写的目的地恰好为 PCI 设备 B。

（3）PCI 设备 B 发现有发向自己的 Posted 写请求时，并不接收这个请求，而是使用重试周期拒绝这个写请求，因为这个 PCI 设备希望从存储器读取完数据后，才能接收这个写请求。

（4）此时在 PCI 桥 A 中暂存的 Posted 写无法完成，同时 PCI 设备 B 的读完成请求也无法穿越 PCI 桥 A，此时将产生死锁。

产生这个死锁的原因是 PCI 设备 B 需要完成存储器读之后才能接收 Posted 存储器写，如果 PCI 设备 B 可以先接收 Posted 存储器写，之后再进行存储器读完成请求，这个死锁就可以避免。产生该死锁的原因与 PCI 总线的序相关，合理地安排这些总线事务的访问序将可以避免这类死锁，下文将详细介绍 PCI 总线的序。

在 PCI 总线中，如果没有合适地处理序的问题，将产生多种类型的死锁。本节所讲述的仅是其中一个较为简单的实例。下文将继续介绍有关 PCI 总线序引发的死锁问题。PCI 总线通过安排访问顺序，可以合理地解决这类死锁问题。

11.3　PCI 总线的序

PCI 总线为满足生产/消费者模型的正确运转，设置了许多与“序”相关的规则。只要 PCI 设备满足这些序，那么无论数据缓冲、Flag 和 Status 位在 PCI 总线的什么位置，都可以保证生产/消费者模型的正常运转。

11.3.1　PCI 总线序的通用规则

PCI 总线在进行数据传递时规定了一些规则。

（1）PCI 总线仅支持 Posted 存储器写总线事务，而配置和 I/O 写总线事务只能使用“Delayed”总线写事务实现。

（2）Posted 存储器写总线事务需要按序完成。PCI 桥必须按照“先进先出”的原则，处

理 Posted 存储器写总线事务。在图 11-1 所示的实例中，如果 Flag 位在 PCI 总线 0 上时，生产者需要通过 PCI 桥传递数据，并将 Flag 位置 1，这两个操作都需要使用 Posted 存储器写总线事务。如果 PCI 不遵循“先进先出”的原则，有可能发生 Flag 位已经置 1，而数据尚未完全到达数据缓冲，从而引发数据完整性问题。

（3）双方向的数据写没有序的关系。如图 11-1 所示，生产者通过 PCI 桥向数据缓冲写入数据，与消费者通过 PCI 桥更新 Status 状态位没有序的关系。PCI 桥会为双方向的数据传递设置独立的缓冲，两者间的数据传递没有序的要求。

（4）读请求通过 PCI 桥时需要进行数据同步。当来自任何主设备的读请求通过 PCI 桥时，PCI 桥需要按照以下步骤处理这个读请求。

1）这个读请求总线事务首先被 PCI 桥暂存。该读请求可以穿越 PCI 桥，或者被 PCI 桥转换为 Delayed 读请求结束主设备的请求，并使主设备择时重试这个读请求。

2）在 PCI 桥向目标总线发起读请求之前，需要将“与这个读请求方向相同的 Posted 写事务刷新出 PCI 桥。以图 11-1 为例，消费者在读取 Flag 位时，PCI 桥中暂存的“从 PCI 总线 0 到 PCI 总线 1”的 Posted 写总线事务都将刷新到 PCI 总线 1 中。该操作可以保证主设备可以从目标设备获得最新的 Flag 信息，因为“和读请求同方向”的写操作都从 PCI 桥中刷新出去。

3）PCI 桥从目标总线上获得数据，因为在读取数据之前，已经将 Posted 写事务刷新出 PCI 桥，因此这个读操作可以获得最新的数据，从而保证了数据访问的一致性。

4）PCI 桥将“与这个读完成方向相同的”Posted 写事务刷新出 PCI 桥。以图 11-1 为例，将 Posted 写事务从桥片刷新到 PCI 总线 0，此时主设备获得 Flag 信息后，可以保证所有的数据已经到达数据缓冲，因为“和读完成同方向的写操作都从 PCI 桥中刷新出去。

5）当主设备再次发出这个读请求时，PCI 桥将传送在数据缓冲中的数据。

（5）Posted 写总线事务可以穿越 Non-Posted 总线事务。PCI 桥在等待其他 Non-Posted 总线事务完成时，能够接收 Posted 写总线事务，如果 PCI 桥不这样做，将可能引发死锁，如图 11-3 所示。这里有两个例外，一个是 PCI 桥的 Posted 写数据缓冲满时，PCI 桥可以暂时不接收这个 Posted 写总线事务；另外一个是 PCI 桥正在处理一个“Locked”总线操作时，也可以不接收 Posted 写总线事务。

11.3.2 Delayed 总线事务的传送规则

PCI 总线规定在主设备置 FRAME#信号有效后的 16 个时钟周期之内，目标设备需要置 TRDY#有效，否则 PCI 总线将出现夭折现象，因此如果一个 PCI 主设备需要使用 Non-Posted 总线事务，通过多级 PCI 桥访问最终的目的设备时，可以使用 Delayed 总线事务。此时 PCI 桥首先 Retry 当前 Non-Posted 总线事务，并将其转换为 Delayed 总线事务。该 Non-Posted 总线事务的发起者需要择时重试该总线事务，而 PCI 桥将这个 Delayed 总线事务暂存。

在处理 Delayed 总线事务时，PCI 桥可以每次只处理一个 Delayed 总线事务，当下一个 Delayed 总线事务到达时，PCI 桥可以直接拒绝此 Delayed 总线请求事务；或者在 PCI 桥中设置一个队列，依次将 Delayed 总线请求事务保存在这个队列中，当这个队列满时，再拒绝下一个 Delayed 总线事务。使用 Delayed 总线请求事务进行数据传送时，需要遵循以下规则。

（1）只有 Non-Posted 总线事务才能使用 Delayed 总线事务。

（2）主设备访问目标设备时，如果被 PCI 桥使用重试周期暂时中断时，主设备必须择时重新访问这个目标设备，因为 PCI 桥将使用 Delayed 总线请求事务继续进行数据访问，PCI 桥获得这个数据后，主设备再从 PCI 桥中获得这个数据。

（3）如果一个 Delayed 总线请求事务被要求重试，发起这个 Delayed 总线读请求事务的设备需要不断地择时重发这个数据访问，直到完成这个数据访问为止。在 Delayed 总线请求事务没有到达最终的设备之前，仅是一个数据请求，可以随时被丢弃。而这些重试操作极大浪费了 PCI 总线的带宽，这也是 Delayed 总线事务的缺点。

（4）当 Delayed 总线请求事务成功到达目的总线后，这个 Delayed 总线请求事务将被转换为 Delayed 总线完成事务。此时这个 Delayed 总线完成事务除了在以下两种情况之外，不能被随便丢弃。

1）如果主设备向一个可以预取的存储器空间进行读操作时，产生的 Delayed 总线完成事务可以被丢弃。因为对可以预取的存储器空间进行多次读操作，都不会产生任何副作用。

2）当主设备在 2^{15} 个时钟周期后，依然没有进行总线重试时，PCI 桥可以丢弃这个 Delayed 总线完成事务，而且需要通过某种机制通知主设备这个 Delayed 总线完成事务已经被丢弃，这种情况极少发生。

（5）PCI 桥在处理 Delayed 总线事务时，必须能够接收来自这个桥同一侧的 Posted 存储器写请求。

（6）Delayed 数据请求和 Delayed 数据完成之间没有序的要求。

（7）Delayed 数据完成不可以超越之前的 Posted 写总线事务。

（8）如果主设备需要“Delayed 读总线事务 A”一定要在“Delayed 读总线事务 B”之前结束，唯一的方法就是在 Delayed 读总线事务 A 完全结束后，再启动 Delayed 读总线事务 B。因为 Delayed 读总线事务 A 有可能被设备使用重试周期结束。因此尽管 PCI 设备先发送 Delayed 读总线请求 A，仍然有可能在后发送的 Delayed 读总线请求 B 完全结束后，才被处理。

11.3.3 PCI 总线事务通过 PCI 桥的顺序

在 PCI 桥中，设置了许多缓冲暂存各类总线事务，包括 Posted 存储器写（PMW），Delayed 读请求（DRR），Delayed 读完成（DRC），Delayed 写请求[⊖]（DWR）和 Delayed 写完成（DWC）。这些不同种类的总线事务在同一方向穿越 PCI 桥时，无论是从上游总线到下游总线还是从下游总线到上游总线穿越 PCI 桥时，都必须遵循一定的顺序，以满足生产/消费者模型在 PCI 总线上的正确运行。有关 PCI 桥的序如表 11-4 所示。在该表中出现的“Yes”、“No”和“Yes/No”的定义如下所示。

- “Yes/No”表示 Row 和 Column 之间的总线事务没有序的关系，如 DWC（Row E）总线事务和 PMW（Column 2）总线事务在通过 PCI 桥时没有先后顺序。
- “Yes”表示 Row 中的总线事务先于 Column 中的总线事务通过 PCI 桥。比如 PMW 总线事务可以超越 DRR 和 DWR 总线事务。
- “No”表示 Row 中的总线事务后于 Column 中的总线事务通过 PCI 桥。如 PMW 总线事务不能超越之前的 PMW 总线事务。

⊖ I/O 和配置写总线事务使用 DWR。

- 表中的“Yes”和“No”的上标对应一个序的规则，例如 No[1] 中的上标 1 对应规则 1。这些规则将在下文陆续介绍。

表 11–4　PCI 桥使用的数据访问顺序

Row pass Col?	PMW Column[2]	DRR Column[3]	DWR Column[4]	DRC Column[5]	DWC Column[6]
PMW（Row A）	No[1]	Yes[5]	Yes[5]	Yes[7]	Yes[7]
DRR（Row B）	No[2]	Yes/No			
DWR（Row C）	No[3]				
DRC（Row D）	No[4]	Yes[6]		Yes/No	
DWC（Row E）	Yes/No[4]				

1. Posted 存储器写通过 PCI 桥时需要按序完成

Posted 存储器写通过 PCI 桥时需要遵循“先进先出”的原则，否则将会引发数据完整性问题。这个要求对满足生产/消费者模型的正常运行至关重要。

对于生产者，置 Flag 位为 1 和将数据写入数据缓冲都使用 Posted 存储器写总线事务。如果 Flag 位和数据缓冲在同一条 PCI 总线上时，Posted 存储器写不按序到达，可能导致 Flag 位被生产者置 1，而数据缓冲并没有收到数据，从而在消费者使用数据缓冲中的数据时，可能会得到无效数据。

2. DRR 不能超越 PMW

PCI 桥首先将 DRR 保存在缓冲区中，之后 PCI 桥将缓存在该桥中的所有 PMW 都刷新出去后，才能执行这个 DRR，即“先写后读”，否则读入的数据有可能不是最新的。系统软件可以使用这一功能实现“读刷新”操作。PCIe 总线还支持读“0 字节”操作，其主要目的就是完成这种“读刷新”操作。

3. DWR 不能超越 PMW

DWR 首先在 PCI 桥中锁存，之后 PCI 桥将缓存的所有 PMW 发送出去后，才能执行这个 DWR。在生产/消费者模型中，生产者除了可以使用 PMW 总线事务设置 Flag 位之外，也可以使用 DWR 总线事务设置 Flag 状态位（Flag 状态位可能存放在 I/O 地址空间中，而且假设在图 11–1 中，Flag 位和数据缓冲都在 PCI 总线 0 上）。

此时 DWR 不能超越 PMW，因为生产者必须将所有数据都写入数据缓冲后，才能设置 Flag 状态位。如果 DWR 可以超越 PMW，则会出现生产者没有将数据完全写入到数据缓冲中，而 Flag 位已经置 1，从而可能引发数据完整性问题。规则 2、3 与规则 5 直接相关，DRR、DWR 和不能超越 PMW，也意味着 PMW 需要超越 DRR 和 DWR。

4. DRC 不能超越 PMW

PCI 总线使用 DRR 总线事务处理存储器和 I/O 读请求，而该总线事务在获得所读取的数据和完成状态后，将被转化为 DRC 总线事务，并在主设备重新发起读操作时，将数据传递给主设备。DRC 要求数据在穿越 PCI 桥传递给主设备之前，PCI 桥将缓存的 PMW 总线事务刷新出去。

以图 11–1 为例，如果消费者使用 DRC 总线事务获得 Flag 位，当 Flag 位为 1 时，必须要求生产者将数据全部写入数据缓冲中。如果 DRC 可以超越 PMW，则可能出现消费者通过

DRC 获得 Flag 位，而且 Flag 位为 1 时，生产者提供的数据仍在 PCI 桥中的情况。因为这些数据需要通过 PCI 桥才能达到缓冲，而 Flag 位与生产者在同一条总线上，可以立即生效。

当消费者发现 Flag 位为 1 时，会立即读取数据缓冲，在图 11-1 中，如果这个 DRC 请求能够超越 PMW，那么 PCI 桥首先将 Flag 位已经为 1 的消息传递给消费者，之后消费者开始从缓冲中读取数据，而这时生产者使用 PMW 方式写入缓冲的数据还可能在 PCI 桥中，从而造成数据完整性的问题。

在 PCI 总线中，解决该问题的方法是 DRC 不能超越 PMW，这样消费者在收到 Flag 位为 1 的信息之前，在 PCI 桥中的 PMW 一定被 DRC 刷新到 PCI 总线 0 中，从而不会引发数据完整性问题。规则 4 与规则 7 之间相关，DRC 不能超越 PMW，也意味着 PMW 需要超越 DRC。

在 PCI 总线中，DWC 与 PMW 之间没有序的要求。因为 DWC 中并不包含有效数据，仅是通知发送 DWR 的设备，该写请求已经结束。DWC 不在生产/消费者模型中出现。

5. PMW 可以超越 DRR 和 DWR

PCI 总线为了避免死锁，可以使刚进入 PCI 桥的 PMW 总线事务，超越已经在 PCI 桥中暂存的 DRR 和 DWR 总线事务，即 PMW 总线事务可以提前执行。如果 PMW 不能超越 DRR 和 DWR，则会产生死锁。实际上因为规则 2 和 3 的原因，规则 5 的引入是顺利成章的。

值得注意的是该规则的引入还解决了在 PCI 总线中使用的不同版本的 PCI 桥的问题，本小节对这种情况不做深入讨论。

6. Delayed 读写完成可以超越 Delayed 读写请求

如果 DWC 和 DRC 总线事务到达 PCI 主设备时，这个主设备正在等待 DWR 和 DRR 总线事务完成，DWC 和 DRC 必须可以优先到达 PCI 主设备，否则将引发死锁，下文以图 11-4 所示的实例说明这种死锁。

我们考虑 PCI 主设备 X 和 Y 通过 PCI 桥 A 和 B 进行数据读写，其步骤如下。

（1）PCI 主设备 X 和 Y 同时发起一个 Non-Posted 读写请求。

（2）PCI 桥 A 和 B 将这个 Non-Posted 总线读写请求转换为 Delayed 读写请求，并暂存在桥内缓冲中，分别为 DRR-X 和 DRR-Y，同时使用重试周期结束 PCI 主设备 X 和 Y 的读写请求。之后 PCI 主设备 X 和 Y 将定时重发这个 Non-Posted 总线读写请求，直到完成本次总线读写请求。

（3）PCI 桥 A 和 B 依次获得 PCI 总线 1 的使用权，并将 DRR-X 和 DRR-Y 请求发向对方。PCI 桥 B 和 A 将来自对方的 Delayed 读写请求锁存在桥内缓冲中，并发起重试周期结束来自 PCI 桥 A 和 B 的 Delayed 读写请求。

（4）PCI 桥 A 和 B 将定时重发这个 Delayed 总线读写请求，直到获得 Delayed 总线读写完成信息。

（5）PCI 桥 A 和 B 将分别获得 PCI 总线 0 和 2 的使用权，将 DRR-Y 和 DRR-X 请求发送到最终 PCI 设备。假设 PCI 桥 A 获得 PCI 总线 0 的使用权，并完成了 PCI 主设备 Y 发起的 DRR-Y 请求，此时 PCI 桥 A 将从 PCI 主设备 X 中得到 Delayed 读写完成信息，并将 DRR-Y 请求转换为 DRC-Y 请求，并将其锁存在 PCI 桥 A 的缓存中。等待 PCI 桥 B 对 PCI 桥 A 进行重试。

（6）如果此时 PCI 桥 A 在没有完成 DRR-X 请求（该请求是发向 PCI 主设备 Y）时，不能接收 DRC-Y 请求将引发死锁。因为 DRC-X 请求也会因为相同的原因不会被 PCI B 桥接收，从而 PCI 桥 A 无法完成 DRR-X 请求。

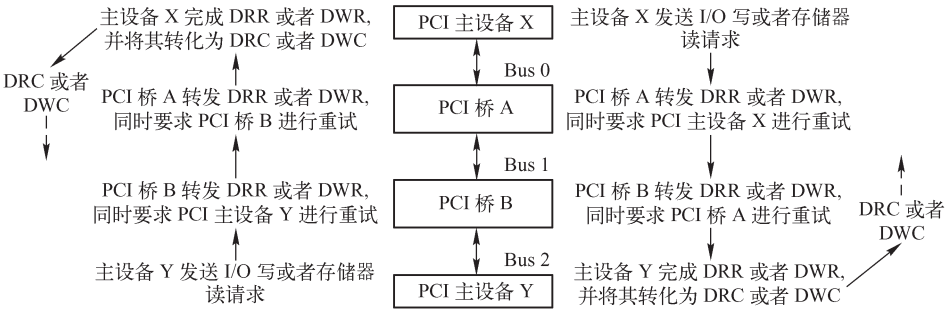

图 11-4　DWC/DRC 不能超越 DRR/DWR 而造成死锁

如果 Delayed 总线完成可以超越 Delayed 请求时，将有效地避免这类死锁。因为 Delayed 总线完成是发生在 Delayed 总线请求之后的事件，这两个请求有因果关系，希望读者认真体会规则 6。

7. PMW 总线事务可以超越 DRC/DWC 总线事务

PCI 总线规定 PMW 总线必须可以超越之前已经在 PCI 桥中暂存的 DRC/DWC 总线事务，以避免死锁。引入该规则的主要原因是在 PCI 总线层次结构中可能存在不同版本的 PCI 桥，它们的缓冲管理策略并不一致，从而有可能造成死锁。本节对这种情况不做进一步描述。

11.3.4　LOCK，Delayed 和 Posted 总线事务间的关系

PCI 桥可以将一个 LOCK 总线事务从上游传递到下游，但是不能将一个 LOCK 总线事务从下游传递到上游。PCI 桥将下游传递到上游的 LOCK 总线事务转换为普通的总线事务，即去掉这个 LOCK 标志。

当 PCI 桥从一个主设备接收一个 LOCK 总线事务之后，将这个总线事务发送到下游总线的目标设备。如果该 LOCK 总线事务是 Non-Posted 总线事务，此时 PCI 桥并不能立即完成这个 LOCK 总线事务，因为 PCI 桥还需要将“Non-Posted 总线请求”对应的总线完成信息传递给发起者之后，LOCK 总线事务才能最终完成。

在 PCI 桥没有将 LOCK 总线完成传递给发起者之前这段时间里，PCI 桥仅接收这个发起 LOCK 总线事务的主设备的总线请求，而重试其他主设备发出的总线事务。该 PCI 桥不会使用 Delayed 总线事务接收其他 Non-Posted 总线事务，也不会暂存这些总线事务。

一个 LOCK 总线事务从 PCI 桥的上游到达 PCI 桥，并在 PCI 桥将这个 LOCK 总线事务传递到下游总线这段时间里，PCI 桥需要进行以下处理。

（1）将所有同方向的 PMW 总线事务刷新到下游总线。

（2）对于 Delayed 总线事务，PCI 桥需要进行特别处理。丢弃所有暂存在 PCI 桥中的 Delayed 总线事务；允许 LOCK 总线事务超越任何读写请求。或者完成所有 Delayed 读写请求，再将 LOCK 总线事务发送给下游总线。

在 PCI 桥的下游总线接收 LOCK 总线请求之前，PCI 桥仍然可以暂存来自下游总线的数据请求；而在下游总线接收 LOCK 总线请求之后，PCI 桥不能接收任何来自下游总线的数据请求，直到发起 LOCK 总线请求的主设备解锁。

在一个最简单的 PCI 桥的实现中，一个 LOCK 总线事务在 PCI 桥的下游总线建立完毕后，PCI 桥不能接收上/下游总线的数据请求，除了来自发起 LOCK 总线事务的 PCI 主设备

的数据请求。而且 PCI 桥必须完成发向上游总线的 PMW，DRC 和 DWC 总线请求。采用这个规则可以保证使用 LOCK 总线事务时不会引发死锁。但是使用这些规则将极大影响 PCI 总线的传送性能，为此在处理器系统的设计中，最好不使用 LOCK 总线事务。

11.4 PCIe 总线的序

PCIe 总线的序基于 PCI 总线的序，并进行了许多扩展。在 PCI 总线上，仅能使用强序传送规则，而 PCIe 总线支持 Relaxed ordering 方式进行数据传递，使用这种方法时，不同的 TLP 在通过 RC 和 Switch 到达 EP 时，不一定遵循 PCI 总线的强序原则，这也意味着先发出去的 TLP 并不一定能够最先到达目的地。PCIe 总线使用 Relaxed ordering 数据传送方式，在一定程度上可以提高数据传送效率。

在 TLP 的 Attr 字段中有一个 Relaxed Ordering 位，表示该 TLP 是否支持 PCIe 总线的 Relaxed Ordering 方式，但是 TLP 是否可以使用 Relaxed Ordering 还与这个 TLP 经过的设备有关。如果一个 TLP 经过的 Switch 不支持 PCIe 的 Relaxed Ordering 数据传送方式，通过这个 Switch 的 TLP 报文依然需要使用强序方式通过这个 Switch。

系统软件可以通过使能 Device Control 寄存器中的 Enable Relaxed ordering 位，来禁止或者使能 TLP 报文的 Relaxed ordering 功能，Device Control 寄存器在 PCIe 设备的 PCI Express Capability 结构中。目前大多数 PCIe 设备不支持 Relaxed ordering 方式进行 TLP 的传递。

PCIe 总线的 Relaxed Ordering 数据传送方式是有条件的，PCIe 总线的每一个 TLP 报文都有一个唯一的 TC，而这个 TC 又和一个唯一的 VC 对应[㊀]。Relaxed Ordering 与报文使用的 VC 相关。VC 相同的 TLP 间的传送遵循 Relaxed Ordering 的原则，而 VC 不同的 TLP 间没有序的要求。在 PCIe 总线中，所有数据传送类型，如存储器、I/O、配置和 Message 总线事务都需要遵循规定的传送顺序。

11.4.1 TLP 传送的序

VC 不同的 TLP 间没有序的要求，在 PCIe 总线中，“序”是指 VC 相同的 TLP 之间的传送顺序，其关系如表 11-5 所示。

表 11-5 PCIe 总线的序

Row Pass Col?	Posted Request Col 2	Read Request Col 3	NPR with Data Col 4	Completion Col 5
Posted Request Row A	a. No b. Y/N	Yes	Yes	a. Y/N b. Yes
Read Request Row B		Y/N		
NPR with Data Row C				
Completion RowD		Yes		a. Y/N b. No

㊀ 不同的 TC 可能会共用一个 VC，而一个 TC 只能使用一个 VC。TC 和 VC 的对应关系是多对一的关系。如 TC1 和 TC2 可以都可以使用 VC1，而 TC1 不能在使用 VC0 的同时还使用 VC1。

各个表项的含义如下。

- Posted Request 由存储器写请求 TLP 或者 Message 使用。
- Read Request 由 I/O、配置和存储器读请求使用。
- NPR（Non-Posted Request）with Data 由 I/O、配置写和原子操作使用。
- a. 表示 TLP 的 RO 位为 0，即不使能 Relaxed Order 的情况。
- b. 表示 TLP 的 RO 位为 1 或者 IDO 位为 1，即使能 Relaxed Ordering 或者使能 ID-Based Ordering 的情况。不同的规则使用 a，b 子规则略有差异。
- Yes 表示 Row 中的 TLP 必须能够穿越 Col 中的 TLP。
- Y/N 表示 Row 中的 TLP 和 Col 中的 TLP 没有序的关系。
- No 表示 Row 中的 TLP 一定不能穿越 Col 中的 TLP。

下文出现的 XY a/b 中，X 与行对应，其值为 A ~ D，Y 与列对应，其值为 2 ~ 5。如 A2 a 表示“Posted Request”在 RO 位为 0 的情况下是否能够超越“Posted Request”。

通过表 11-5 与表 11-4 的比较，可以发现在 RO 位为 0 时（即不使用 Relax Ordering），PCIe 总线的序与 PCI 的序基本兼容。但是因为在一个 TLP 中有时 RO 位和 IDO 位不为 0，因此 PCIe 总线的序需要根据 a 和 b 两种情况分别进行讨论。

1. A2

A2 需要分为两种情况讨论，其中 a 对应 TLP 的 RO 和 IDO 位都为 0 情况，而 b 对应 TLP 的 RO 或者 IDO 位为 1 的情况。

A2 a 的值为 No，表示 Posted Request 报文不能超越之前的 Posted Request 报文，这与 PCI 总线中 PMW 不能超越之前的 PMW 要求相同。PCI 总线的 PMW 与 PCIe 的 Posted Request 报文基本一致，存储器写和 Message 使用这类报文。

A2 b 的值为 Y/N，该规则需要分为两种情况进行讨论，RO 位为 1 或者 IDO 位为 1。当 RO 位为 1 时，该 Posted Request 报文可以超越之前的 Posted Request 报文。在设计中应用该规则是十分危险的，该规则也意味着“写”可以超越“写”。

如在第 11.1.1 节描述的生产/消费者模型中，生产者首先将数据写入数据缓冲，然后将 Flag 位置 1。如果“将 Flag 位置 1”的写操作可以超越“写入数据缓冲”的写操作，那么消费者可能会从无效的数据缓冲中读取数据，从而出现错误。

在 Switch 和支持 Peer-to-Peer 传送的 RC 中，设置了一个寄存器位“No RO-enabled PR-PR Passing”[⊖]，当该位为 1 时，当 TLP 通过这些 Switch 和 RC 时，Posted Request 报文不能超越之前的 Posted Request 报文，即便这些 TLP 的 RO 位为 1。

当 IDO 位为 1 时，该 Posted Request 报文可以超越之前的 Posted Request 报文。使用该规则的前提是，这两个 Posted Request 报文使用的 Requester ID 号不同，即这两个 Posted Request 报文是由不同的 PCIe 设备发出的，有关 IDO 序的详细说明见第 11.4.2 节。

2. A3 和 A4

A3 和 A4 的值为 Yes，表示 Posted Request 报文可以超越之前的 Non-Posted 读和写请求。该规则与 PCI 总线的 PMW 可以超越 DRR 和 DWR 兼容，其主要目的是避免死锁。详见 PCI

⊖ 该位在 Device Capabilities 2 寄存器中。

总线序的规则 5。

3. A5

A5 需要分为两种情况讨论，a 适用于 PCIe 总线中的 RC 和 Switch；而 b 适用于 PCIe 桥。

A5 a 的值为 Y/N。在 PCIe 总线中，Posted Request 报文可以超越之前的完成报文也可以不超越，在这两种情况下，都不会造成死锁。该规则与 PCI 总线中 PMW 必须超越 DRC 和 DWC 不同（PCI 总线中的规则 7），因为在 PCI 体系结构中会出现不同版本的 PCI 桥，而在 PCIe 体系结构中不会出现这种情况。

A5 b 的值为 Yes。表示在 PCIe 桥中，PCIe 总线向 PCI 总线的方向传递报文时，Posted Request 报文必须可以超越完成报文，以避免死锁。PCIe 桥内部由多个虚拟 PCI 桥组成，参见图 4-13，因此 PCIe 总线中的 A5 b 与 PCI 总线中的规则 7 兼容。

4. B2，C2

B2 需要分为两种情况讨论，其中 a 对应 TLP 的 IDO 位为 0 情况，而 b 对应 TLP 的 IDO 位为 1 的情况。

B2 a 的值为 No，表示 Read Request 报文不能超越之前的 Posted Request 报文，这与 PCI 总线中 DRR 不能超越之前的 PMW 要求相同。PCI 总线的 DRR 与 PCIe 总线的 Read Request 报文基本一致，存储器、I/O 和配置读使用这类报文。

B2 b 对应 TLP 的 IDO 位为 1 的情况。当 IDO 位为 1 时，该 Read Request 报文可以超越之前的 Posted Request 报文，否则不能超越。使用该规则的前提是，Read Reques 报文和 Posted Request 报文使用的 Requester ID 号不同。

C2 也需要分为两种情况讨论，其中 a 对应 TLP 的 IDO 位为 0 情况，而 b 对应 TLP 的 IDO 位为 1 的情况。其实现机制与 B2 类似，本节对此不做进一步说明。

5. B3，B4，C3，C4

在 PCIe 总线中，Non-Posted Request 报文可以超越之前的 Non-Posted Request 报文，也可以不进行这种超越。该规则从 PCI 总线中继承而来，而在 PCI 总线中 DRR/DWR 可以超越之前的 DRR/DWR。PCIe 设备在实现中，需要与该规则兼容。

但是在 PCIe 总线中，存储器读操作使用 Split 方式进行传送，因此该规则的引入为 PCIe 设备的设计带来了不小的麻烦。当一个 EP 进行 DMA 读操作时，需要首先向 RC 发送存储器读请求，如 R1 ~ R4，而 RC 收到这些读请求时，将回送读完成 TLP，如 C1 ~ C4。为简便起见，我们认为每一个存储器读请求的大小为 64B，而读完成的大小也为 64B，而且不考虑对界的问题，这样 EP 发送的存储器读请求将与 RC 的读完成一一对应，我们假设 R1 对应 C1，R2 对应 C2，并以此类推 R4 对应 C4。

EP 首先按照 R1 ~ R4 的顺序发送这些存储器读请求。但是 R1 ~ R4 在通过 Switch 和 RC 之后可能出现乱序，如果 Non-Posted Request 报文可以超越之前的 Non-Posted Request 报文，RC 最终收到的存储器读请求可能是乱序的，如 R2，R4，R3 和 R1，因此 RC 发送给 EP 的读完成报文可能为 C2，C4，C3 和 C1，这个顺序与 EP 发向 RC 的存储器读请求的顺序并不相同。因此 EP 必须处理这种乱序，这为 EP 的设计带来了不小的困难。

6. B5，C5

在 PCIe 总线中，Non-Posted Request 报文与之前的完成报文没有序的要求。该规则从 PCI 总线中继承而来，在 PCI 总线中 DRR/DWR 可以超越之前的 DRC/DWC，也可以不超

越。这些报文的传递不会影响生产/消费者模型的正常运行。

7. D2

D2 需要分为两种情况进行讨论，分别是 D2 a 和 D2 b。

其中 D2 a 为 No，表示在 PCIe 总线中，CplD 报文不能超越 Posted Request 报文，该规则与 PCI 总线中的规则 4 兼容。这也是保证生产/消费者模型正常运行的必要条件。

而 D2 b 为 Y/N。如果 TLP 的 RO 位为 1 时，该 CplD 报文可以超越 Posted Request 报文。设计者需要慎重使用该规则，因为该规则的应用有可能破坏生产/消费者模型的正常运转。只有传递与生产/消费者模型无关的报文时，才能应用该规则。

此外如果 TLP 的 IDO 位为 1 时，该 CplD 报文可以超越之前的 Posted Request 报文，否则不能超越。使用该规则的前提是，CplD 报文和 Posted Request 报文使用的 Requester ID 号不同。值得注意的是，Cpl 报文由 I/O 或者配置写完成报文使用，该报文中不含有数据，仅包含完成信息。该报文的使用方法与 PCI 总线的 DWC 类似。该报文与 Posted Request 报文没有序的要求，该规则与 PCI 总线的规则 4 兼容。

8. D3，D4

在 PCIe 总线中，完成报文可以超越之前的 Non-Posted Request 报文。该规则从 PCI 总线中继承而来，与规则 6 对应。该规则的引入主要为了防止死锁。

9. D5

如果完成报文与之前的完成报文的 Transaction ID 不同时，该报文可以超越之前的完成报文；如果相同，不能进行这样的超越。

当一个 PCIe 设备向目标设备发送存储器读请求时，目标设备可能会使用一个或者多个存储器读完成报文将数据回送。如果使用多个存储器读完成报文时，这些存储器完成报文按“地址升序”顺序先后到达源设备。

如果设备 A 需要从设备 B 读取 256B[㊀]的数据，其访问的地址为 0x1000－0000～0x1000－00FF 时，设备 A 可以向设备 B 发送一个存储器读请求 TLP，而设备 B 将以 64B 为单位向设备 A 发送存储器读完成 TLP，这些完成报文必须以 C1～C4 的顺序到达设备 A，C1～C4 存储器读完成 TLP 对应的数据区域如下。

- C1 与 0x1000－0000～0x1000－003F 对应。
- C2 与 0x1000－0040～0x1000－007F 对应。
- C3 与 0x1000－0080～0x1000－00BF 对应。
- C4 与 0x1000－00C0～0x1000－00FF 对应。

如果设备 A 需要从设备 B 读取 512B 的数据，访问的地址为 0x1000－0000～0x1000－01FF 时，这段数据区域大于大于设备 A 的 Max_Read_Request_Size 参数，因此设备 A 需要向设备 B 发出两个存储器读请求，这两个存储器读请求使用两个不同的 Tag 字段进行区分，分别为 R_{T0} 和 R_{T1}。假设 R_{T0} 的 Tag 字段为 0，其请求的数据区域为 0x1000－0000～0x1000－00FF；而 RT1 的 Tag 字段为 1，其请求的数据区域为 0x1000－0100～0x1000－01FF。

假设设备 A 首先发送 R_{T0}，然后再发送 R_{T1}。但是设备 B 仍然可能首先收到 R_{T1}，然后再

㊀ 假设设备 A 的 Max_Read_Request_Size 参数为 256B，而设备 B 可以为 RC 也可以为普通的 EP。

收到 R_{T0}，因为 PCIe 总线允许存储器读请求超越存储器读请求。设备 B 收到这些存储器读请求后，向设备 A 发送存储器读完成 TLP（以 64B 为单位），分别为 $C1_{T0}$ ~ $C4_{T0}$ 和 $C1_{T1}$ ~ $C4_{T1}$。

其中 $C1_{T0}$ ~ $C4_{T0}$ 使用的 Tag 字段为 0，而 $C1_{T1}$ ~ $C4_{T1}$ 使用的 Tag 字段为 1，分别与 R_{T0} 和 R_{T1} 对应，这也意味着这两组存储器读完成使用的 Transaction ID 不同，因此可以彼此超越，这两组存储器读完成 TLP 对应的数据区域如下。

- $C1_{T0}$ 与 0x1000 – 0000 ~ 0x1000 – 003F 对应；$C2_{T0}$ 与 0x1000 – 0040 ~ 0x1000 – 007F 对应；$C3_{T0}$ 与 0x1000 – 0080 ~ 0x1000 – 00BF 对应；$C4_{T0}$ 与 0x1000 – 00C0 ~ 0x1000 – 00FF 数据区域对应。
- $C1_{T1}$ 与 0x1000 – 0100 ~ 0x1000 – 013F 对应；$C2_{T1}$ 与 0x1000 – 0140 ~ 0x1000 – 017F 对应；$C3_{T1}$ 与 0x1000 – 0180 ~ 0x1000 – 01BF 对应；$C4_{T1}$ 与 0x1000 – 01C0 ~ 0x1000 – 01FF 数据区域对应。

此时设备 A 收到的存储器读完成，有多种可能，如表 11–6 所示。

表 11–6　设备 A 收到的存储器读完成序列

序列 1	序列 2	序列 3	序列 4	序列 5	序列 6	序列 7	序列 8
$C1_{T0}$	$C1_{T0}$	$C1_{T1}$	$C1_{T1}$	$C1_{T0}$	$C1_{T0}$	$C1_{T1}$	$C1_{T1}$
$C2_{T0}$	$C1_{T1}$	$C2_{T1}$	$C2_{T1}$	$C1_{T1}$	$C2_{T0}$	$C2_{T1}$	$C1_{T0}$
$C3_{T0}$	$C2_{T0}$	$C3_{T1}$	$C1_{T0}$	$C2_{T1}$	$C3_{T0}$	$C3_{T1}$	$C2_{T0}$
$C4_{T0}$	$C2_{T1}$	$C4_{T1}$	$C2_{T0}$	$C2_{T0}$	$C1_{T1}$	$C1_{T0}$	$C2_{T1}$
$C1_{T1}$	$C3_{T0}$	$C1_{T0}$	$C3_{T1}$	$C3_{T0}$	$C2_{T1}$	$C2_{T0}$	$C3_{T0}$
$C2_{T1}$	$C3_{T1}$	$C2_{T0}$	$C4_{T1}$	$C4_{T0}$	$C3_{T1}$	$C3_{T0}$	$C4_{T0}$
$C3_{T1}$	$C4_{T0}$	$C3_{T0}$	$C3_{T0}$	$C3_{T1}$	$C4_{T1}$	$C4_{T0}$	$C3_{T1}$
$C4_{T1}$	$C4_{T1}$	$C4_{T0}$	$C4_{T0}$	$C4_{T1}$	$C4_{T0}$	$C4_{T1}$	$C4_{T1}$

上表仅列出了设备 A 可能从设备 B 中收到的存储器读完成序列。由此可见对于 Tag 字段不同的存储器完成报文，在到达设备 A 时顺序并不确定。但是对于 Tag 字段相同的存储器读完成 TLP，这些存储器完成报文是严格按照地址“升序”的顺序到达设备 A。PCIe 总线的这种乱序为 PCIe 设备的设计带来了不小的麻烦，设计者必须认真地处理这些乱序可能。

11.4.2　ID-Base Ordering

IDO 模型由 PCIe V2.1 版本引入。该模型引入了“数据流”的概念，即相同的数据源发出的 TLP 属于相同的“数据流”，而不同数据源发出的 TLP 属于不同的“数据流”。PCIe 链路可以根据“数据流”对 TLP 进行区分。IDO 模型允许分属不同“数据流”的 TLP 之间没有序的要求，即可以“自由乱序”。

IDO 模型的引入有利于 Switch 对发向不同 Egress 端口的报文进行优化处理。我们假设 Switch 的一个 Ingress 端口收到了若干个数据报文，这些报文分别发向不同的 Egress 端口，如图 11–5 所示。

其中 TLP1 ~3 分别发向 Egress 端口 1 ~3，在不使用 IDO 模型的情况下，Ingress 端口需要等待之前的报文被发送出去之后，才能发送之后的报文。TLP3、TLP2 和 TLP1 依次进入 Ingress 端口，如果不使用 IDO 模式时，TLP2 需要等待 TLP3 完全离开 Ingress 端口后，才能

被发送；同理 TLP1 需要等待 TLP2 离开 Ingress 端口才能被发送。显然这种数据传送方式并不合理。

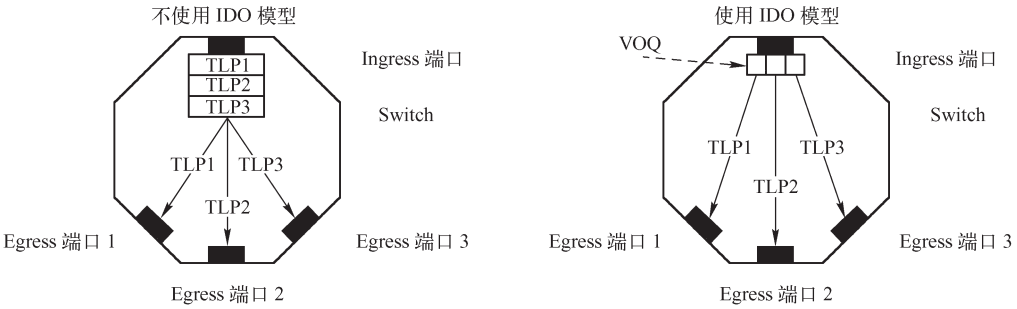

图 11-5　IDO 模型的优点

这种拥塞也被称为 HOL（Head-of-Line）Blocking。引起这种现象的主要原因是 Ingress 端口每次只能处理一个报文引起的。如果在 Ingress 端口只有一个发送部件与 Egress 端口 1 ~ 3 对应时，即便使用 IDO 模型并不会提高效率，因为这些报文依然需要通过这一个发送部件串行发送。由此可见对于这种类型的 Switch，使用 IDO 模型并不会提高效率。

但是如果 Ingress 端口中有 3 个发送部件，分别与 Egress 端口 1 ~ 3 对应时，实际上是 Ingress 端口为每一个 Egress 端口提供分离的发送缓冲，这个缓冲也被称为 VOQ（Virtual Output Queue）。此时 TLP1 ~ 3 的发送可以同时进行，而 Ingress 端口使用 IDO 模型，可以不考虑传送“序”[㊀]而同时发送 TLP1 ~ 3，从而极大地提高了 Switch 的转发效率。目前多数 Switch 的 Ingress 端口都支持 VOQ 技术。

在 PCIe V2. 1 总线规范提出之前，PLX[㊁]公司已经使用类似 IDO 模型的技术以优化 Switch 的数据传送，即 PLX-Specific Relaxed Ordering 技术。下文以 PEX8518 芯片为例说明该技术的具体实现，该芯片是 PLX 公司设计的 PCIe Switch。在 PEX8518 中，每一个 Ingress 端口都为不同的 TC 设置了一个“PLX-Specific Relaxed Ordering”使能位，当该位为 1 时，当一个 Ingress 端口收到的 TLPs 发向的 Egress 端口不同，则没有序的要求；而 Egress 端口相同的 TLPs 必须按序进行。PLX 使用的这种技术与 IDO 模型类似。

11. 4. 3　MSI 报文的序

在 PCIe 总线中，还有一种序引发的数据完整性问题需要特别注意，即由 MSI 报文引发的数据完整性问题。PCIe 设备使用 MSI 机制时，通过向中断控制器发送 MSI 报文以提交中断请求。然而对于 PCIe 体系结构而言，这个 MSI 报文与普通的存储器写报文并没有本质的区别，这个报文也可以使用不同的 TC。如果设备的数据传送使用 TC0，而 MSI 报文使用 TC1 时，将可能引发数据完整性的问题。

假设一个 PCIe 设备正在使用 DMA 写操作，将一组数据传递到主存储器，此时该设备将使用存储器写 TLP 进行数据传送，当数据传送完成后，使用 MSI 报文通知处理器 DMA 写操

㊀ TLP1 ~ 3 使用的 Requester ID 不同，因此在 IDO 模型中，没有序的要求。

㊁ PLX 是 PCIe Switch 芯片的主要提供商。

作已经结束。

如果进行数据传递的 TLP 使用 TC0，而 MSI 报文使用 TC1 时，这两种 TC 可能使用的 VC 并不相同，而不同 VC 间的数据传递并没有序的要求。因此该 PCIe 设备虽然从设计逻辑上保证，将传递数据的存储器写 TLP 发送完毕后再发送 MSI 报文，但是 RC 仍然会首先收到 MSI 报文，然后再收到传递数据的存储器写 TLP。

此时如果处理器在收到 MSI 报文后，立即在中断处理服务例程中使用该 PCIe 设备写入的数据，将可能引发数据完整性问题。

PCIe 总线规范并没有约定如何处理传递 MSI 报文而产生的数据完整性问题。在上述实例中，如果 MSI 报文使用的 TC 与数据传递使用的 TC 相同，将不会出现这个数据完整性问题。如果在设计中 MSI 报文使用的 TC 与数据传递使用的 TC 不一致，需要注意该问题。

一个可行的方法是在数据传递结束后，使用“zero-length 存储器读请求 TLP”对目标设备进行读操作，这个读操作可以将数据写入最终目的地。当该读操作结束后，即 PCIe 设备收到存储器读完成 TLP 后，再发送 MSI 报文。

如一个 PCIe 设备完成 DMA 写操作之后，再向目标地址（某个存储器地址）发送一个“zero-length 存储器读请求”报文，该报文可以保证之前的存储器写报文都被刷新到主存储器后，才能从主存储器获得应答信息，因为存储器读请求 TLP 不能超越存储器写报文。当 PCIe 设备收到与这个存储器读请求对应的存储器读完成 TLP 后，再发送 MSI 报文进行中断请求。

使用上述方法虽然可以避免因为传送 MSI 报文带来的数据完整性问题，但是将带来较大的中断延时。因为在 PCIe 体系结构中，一个设备从“发送存储器读请求 TLP”到“获得存储器读完成 TLP”的延时较长。而且使用这种方法也增加了硬件逻辑设计的难度。

目前支持多个 VC 的 PCIe 设备，通常将 MSI 报文和数据传送报文使用的 TC 设置为相同的值，以避免数据完整性问题。如在 Intel 的高精度声卡控制器中，数据传送使用的报文和 MSI 报文都只能使用 TC0。

11.5 小结

本章重点介绍 PCI/PCIe 总线的序。在局部总线中，数据传送顺序以及与其相关的数据完整性一直是系统程序员的学习重点与难点。对于系统程序员而言，这部分内容必须熟练掌握。有许多系统 Bug 仍然因为系统程序员的疏忽，或者并没有深入理解数据完整性的原理，而无意产生，并很难复现。这些 Bug 将对一个处理器系统的稳定运行产生致命影响。

在 PCI/PCIe 总线中规定了数据传送序，使用生产/消费者模型进行数据传递，合理解决了数据完整性问题。而设计者在实现 PCI/PCIe 体系结构时，必须遵循这些传送序，并且符合生产/消费者模型要求的数据传送方式。

第 12 章　PCIe 总线的应用

本章以分析一个 EP 的硬件设计原理和基于这个 EP 的 Linux 驱动程序为线索，说明 PCIe 设备和基于该设备的 Linux 驱动程序的设计流程。本章使用的 PCIe 设备基于 Xilinx 公司 Vetex-5 XC5VLX50T 内嵌的 PCI Express EP 模块，该模块也被 Xilinx 称为 LogiCORE。

LogiCORE 提供了 PCIe 设备的物理层和数据链路层，并做了一些基本的与事务层相关的工作，这使得许多设计者在并不完全清楚 PCIe 体系结构的情况下，也可以实现具有 PCIe 总线接口的设备。本章所述的 PCIe 设备基于 LogiCORE，本章将该 PCIe 设备简称为 Capric 卡。

12.1　Capric 卡的工作原理

Capric 卡的组成结构如图 12-1 所示。

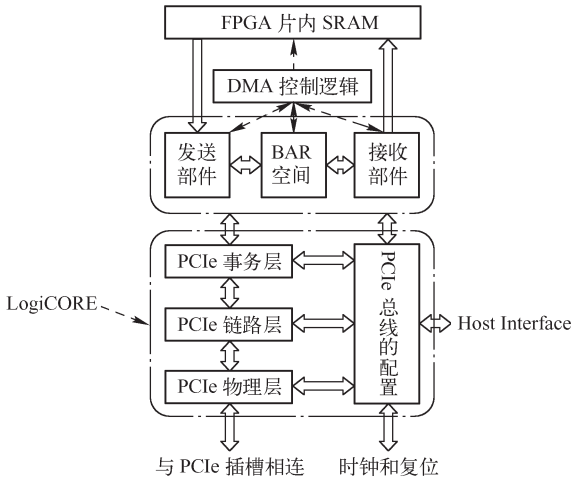

图 12-1　Capric 卡的组成结构

Capric 卡基于 PCIe 总线，主要功能是通过 DMA 读写方式与 HOST 处理器进行数据交换。Capric 卡由 LogiCORE、发送部件、接收部件、BAR 空间、DMA 控制逻辑和 FPGA 片内 SRAM 组成，其工作原理较为简单。

Capric 卡首先使用 DMA 读方式，将主存储器中的数据搬移到 FPGA 的片内 SRAM 中，然后使用 DMA 写方式，将 FPGA 的片内 SRAM 中的数据写入主存储器中。在 Capric 卡中，一次 DMA 操作可以传送的数据区域的最大值为 0x7FFB（0x2047B）。

Capric 卡的各个组成模块的功能描述如下所示。

- LogiCORE。其主要功能是处理 PCIe 设备的物理层、链路层与部分事务层的逻辑，并向外提供必要的接口。PCIe 设备配置空间的初始化，以及与配置和中断请求相关的总线事务也由 LogiCORE 完成。LogiCORE 是 PCIe 总线的接管者，其他部件通过 Log-

iCORE 与 PCIe 链路进行通信。LogiCORE 通过“Host Interface”实现 PCIe 设备的初始化配置。

- Capric 卡的发送部件负责发送 TLP 报文，包括“存储器读请求”和“存储器写请求 TLP”，但是并不包含配置和消息报文的发送。MSI 报文由发送部件通过 LogiCORE 发送。
- 接收部件负责接收“存储器读完成 TLP”。Capric 卡不支持 I/O 读写 TLP。
- DMA 控制逻辑协调发送与接收部件，以完成 DMA 写与 DMA 读操作，该逻辑的实现是 Capric 卡的设计重点。
- BAR 空间中存放了一组操纵 DMA 控制逻辑的寄存器，这组寄存器由 HOST 处理器和 Capric 卡共同读写，从而完成相应的 DMA 操作。Capric 卡仅使用了 BAR0 空间，处理器使用存储器映像寻址方式，而不是 I/O 映像寻址方式访问 BAR0 空间。

12.1.1 BAR 空间

Capric 卡仅使用 BAR0 空间，其大小为 256 B，在该空间中包含以下寄存器，这些寄存器使用小端编码方式，如表 12-1 所示。

表 12-1 Capric 卡的 BAR 空间寄存器

缩　写	偏　移	描　述
DCSR1(Device Control and Status Register 1)	0x00	设备控制和状态寄存器 1
DCSR2(Device Control and Status Register 2)	0x04	设备控制和状态寄存器 2
WR_DMA_ADR	0x08	DMA 写地址寄存器
WR_DMA_SIZE	0x0C	DMA 写传送大小寄存器
RD_DMA_ADR	0x1C	DMA 读地址寄存器
RD_DMA_SIZE	0x20	DMA 读传送大小寄存器
INT_REG	0x2C	中断状态寄存器
ERR	0x30	错误状态寄存器

（1）DCSR1 寄存器，该寄存器由 7 个有效位组成。

- init_rst_o，第 0 位，该位可读写，复位值为 0。为 1 表示 Capric 卡处于复位状态，为 0 表示 Capric 卡已经完成复位。软件通过操纵该位对 Capric 卡进行复位。其过程为首先向此位写 1，然后延时至少 5μs 后（Capric 卡内逻辑和 FPGA 的片内 SRAM 需要至少 5 μs 的复位时间），再向此位写 0。
- int_rd_enb，第 8 位，该位可读写，复位值为 0。为 1 表示当 DMA 读完成后，Capric 卡可以向处理器提交中断请求；为 0 表示 DMA 读完成后，Capric 卡不能向处理器提交中断请求。该位为 DMA 读完成中断使能位。
- int_wr_enb，第 9 位，该位可读写，复位值为 0。为 1 表示当 DMA 写完成后，Capric 卡可以向处理器提交中断请求；为 0 表示 DMA 写完成后，Capric 卡不能向处理器提交中断请求。该位为 DMA 写完成中断使能位。
- int_rd_msk，第 16 位，该位可读写，复位值为 0。为 1 表示当 DMA 读完成后，Capric

卡不能向处理器提交中断请求，而是置 int_rd_pending 位为 1；为 0 表示 DMA 读完成后，Capric 卡可以向处理器提交中断请求。该位为 DMA 读完成中断屏蔽位。

- int_rd_pending，第 17 位，该位可读写，复位值为 0。为 1 表示 Capric 卡含有未发出的 DMA 读完成中断请求，当 int_rd_msk 位由 1 变为 0 时，Capric 卡发送该中断请求；为 0 表示 Capric 卡不含有未发出的 DMA 读完成中断请求。
- int_wr_msk，第 24 位，该位可读写，复位值为 0。为 1 表示当 DMA 写完成后，Capric 卡不能向处理器提交中断请求，而是置 int_wr_pending 位为 1；为 0 表示 DMA 写完成后，Capric 卡可以向处理器提交中断请求。该位为 DMA 写完成中断屏蔽位。
- int_wr_pending，第 25 位，该位可读写，复位值为 0。为 1 表示 Capric 卡含有未发出的 DMA 写完成中断请求，当 int_wr_msk 位由 1 变为 0 时，Capric 卡发送该中断请求；为 0 表示 Capric 卡不含有未发出的 DMA 写完成中断请求。

设置 Mask 和 Pending 位的主要目的是为了防止中断丢失和产生 Spurious 中断请求，这两位与 MSI Capability 结构中的 Mask 和 Pending 位的功能相似。LogiCORE 并不支持 MSI Capability 结构中的 Mask 和 Pending 位，因此 Capric 卡引入了这两个位。

（2）DCSR2，该寄存器由 4 位组成。分别为 mwr_start，DMA 写启动位；mwr_done，DMA 写结束位；mrd_start，DMA 读启动位；mrd_done，DMA 读结束位。

- mwr_start，第 0 位，该位可读写，复位值为 0。系统软件向该位写 1 时启动 DMA 写操作，软件对此位写 0 无意义。
- wr_done_clr，第 1 位，可读，写 1 清除。当一次 DMA 写完成后，wr_done_clr 位将置 1，系统软件将 wr_done_clr 位写 1 清零后，mwr_start 位也将清零，之后系统软件可以重新启动下一次 DMA 写操作。
- mrd_start，第 16 位，该位可读写，复位值为 0。系统软件向该位写 1 时启动 DMA 读操作，软件对此位写 0 无意义。
- rd_done_clr，第 17 位，可读，写 1 清除。当一次 DMA 读完成后，rd_done_clr 位将置 1，系统软件将 rd_done_clr 位写 1 清零后，mrd_start 位也将清零，之后系统软件可以重新启动下一次 DMA 读操作。

（3）WR_DMA_ADR，DMA 写地址寄存器，该寄存器存放 DMA 写操作的目的地址，该寄存器的复位值无意义。再一次强调该寄存器存放的地址为 PCI 总线域的地址，而不是存储器域的地址，尽管在许多处理器系统中，该地址的 PCI 总线域地址与存储器域地址相等。该寄存器由 32 位组成，存放 32 位的 PCI 总线域地址。

（4）WR_DMA_SIZE，DMA 写传送大小寄存器。该寄存器存放一次 DMA 写操作的传送大小，以字节为单位。该寄存器的复位值为 0，由 32 位组成，但是只有低 11 位有效，因此 Capric 卡一次 DMA 传送的最大值为 0x7FF。该寄存器为 N 时表示一次 DMA 写操作的传送大小为 N 个字节。该寄存器为 0 时，即便 DCSR2 寄存器的 mwr_start 位为 1 时，也不能启动 DMA 写传送。当 mwr_start 位为 1，该寄存器由 0 变为其他数据时，也将启动 DMA 写传送。

（5）RD_DMA_ADR，DMA 读地址寄存器，该寄存器存放 DMA 读操作的目的地址，该寄存器的复位值无意义。该寄存器存放的地址为 PCI 总线域地址，由 32 位组成。Capric 仅支持 32 位的 PCI 总线地址。

（6）RD_DMA_SIZE，DMA 读传送大小寄存器，存放一次 DMA 读操作的传送大小，以字节为单位。该寄存器的复位值为 0，由 32 位组成，但是只有最低 11 位有效，因此 Capric 卡一次 DMA 读传送的最大值为 0x7FF。该寄存器为 N 时表示一次 DMA 读操作的传送大小为 N 个字节。该寄存器为 0 时，即便 DCSR2 寄存器的 mrd_start 位为 1 时，也不能启动 DMA 读传送。当 mrd_start 位为 1 时，该寄存器由 0 变为其他数据时，将启动 DMA 读传送。

（7）INT_REG，中断控制状态寄存器，该寄存器存放 Capric 卡的中断状态，该寄存器共由 5 个有效位组成。

- int_src_rd，第 0 位，该位只读，复位值为 0。当 Capric 卡的 DMA 读操作完成后，而且 DCSR1 寄存器的 int_rd_enb 位为 1 时，该位为 1 表示 Capric 卡已经向处理器提交了 DMA 读完成中断请求。值得注意的是当 int_rd_msk 位为 1 时，Capric 卡不能发送 DMA 读完成中断，此时 int_src_rd 位需要等待 Capric 卡置 int_rd_msk 位为 0，发送完毕 DMA 读完成中断后，才能置 1。
- int_src_wr，第 1 位，该位只读，复位值为 0。当 Capric 卡的 DMA 写操作完成后，而且 DCSR1 寄存器的 int_wr_enb 位为 1 时，该位为 1 表示 Capric 卡已经向处理器提交了 DMA 写完成中断请求。值得注意的是当 int_wr_msk 位为 1 时，Capric 卡不能发送 DMA 写完成中断，此时 int_src_wr 位需要等待 Capric 卡置 int_wr_msk 位为 0，发送完毕 DMA 写完成中断后，才能置 1。
- rd_done_clr，第 8 位，该位可读写，复位值为 0，该位为 1 表示 DMA 读操作完成。DMA 读结束后该位由硬件置 1，软件将该位清零后，硬件才能进行下一次 DMA 读操作。该位置 1 后，如果 DCSR1 寄存器的 int_rd_enb 位为 1 时，Capric 卡将向处理器提交中断请求[⊖]。

系统软件在中断服务例程中，向该位写 1 将清除该位；向该位写 0 无意义。如果 Capric 卡不使用中断方式接收数据，系统软件可以查询该位确定当前 DMA 读操作是否已经完成。此位与 DCSR2 寄存器的 rd_done_clr 位相同，软件清除 INT_REG 寄存器的该位后，DCSR2 寄存器的 rd_done_clr 位也被清零。

- wr_done_clr，第 9 位，该位可读写，复位值为 0，该位为 1 表示 DMA 写操作完成。DMA 写完成后该位由硬件置 1，软件将该位清零后，硬件才能进行下一次 DMA 写操作。该位置 1 后，如果 DCSR1 寄存器的 int_wr_enb 位为 1 时，Capric 卡将向处理器提交中断请求。

系统软件在中断服务例程中，向该位写 1 将清除该位；向该位写 0 无意义。如果 Capric 卡不使用中断方式接收数据，系统软件可以查询该位确定当前 DMA 写操作是否已经完成。此位与 DCSR2 寄存器的 wr_done_clr 位相同，软件清除 INT_REG 寄存器的该位后，DCSR2 寄存器的 wr_done_clr 位也被清零。

- int_asserted，第 31 位，该位只读，复位值为 0。当 Capric 卡向处理器提交中断请求时，该位由硬件逻辑置 1。对此位进行写操作无意义。在逻辑实现中，int_asserted = int_src_rd & int_src_wr。

⊖ Capric 卡可以使用 Legacy INTx 或者 MSI 方式进行中断请求。

12.1.2 Capric 卡的初始化

Capric 卡在初始化时需要进行配置寄存器空间和 Capric 卡硬件逻辑的初始化。其中配置寄存器空间的初始化由软硬件联合完成。

Capric 卡的设计基于 Xilinx 公司的 LogiCORE。因此 Capric 卡需要使用 Xilinx 公司提供的“CORE Generator GUI”对 LogiCORE 进行基本的初始化，并设置一些必要的参数，包括 Vendor ID、Device ID、Revision ID、Subsystem Vendor ID 和 Subsystem ID 等参数。有关该工具的使用见［LogiCORE（tm）Endpoint PIPE v1.7］，本节对此不做进一步描述。Capric 卡的配置寄存器空间的初始值如下所示。

- Vendor ID 为 0x10EE，Xilinx 使用的 Vendor ID。
- Device ID 为 0x0007，LogiCORE 使用的 Device ID。
- Revision ID 为 0x00。
- Subsystem ID 为 0x10EE。
- Device ID 为 0x0007。
- Base Class 为 0x05，表示 Capric 卡为“类存储器控制器”。
- Sub Class 和 Interface 为 0x00，进一步描述 Capric 卡为 RAM 控制器。
- Card CIS Pointer 为 0x00，表示不支持 Card Bus 接口。
- BAR0 为 0xFFFFFF00。Capric 卡仅支持 BAR0 空间，该空间采用 32 位存储器映像寻址，其大小为 256 B，而且不支持预读。在初始化时，BAR0 寄存器存放该空间所需要的存储器空间大小，该寄存器由系统软件读取后，再写入一个新的数值。这个数值为 BAR0 空间使用的基地址。
- Max_Payload_Size Supported 参数为 0b010，即 Max_Payload_Size Supported 参数的最大值为 512 B。多数 RC 支持的 Max_Payload_Size Supported 参数仅为 128 B 或者 256 B。因此 LogiCORE 支持 512 B 已经足够了。在 Capric 卡的初始化阶段，需要与对端设备进行协商，确认 Max_Payload_Size 参数的值，如果 Capric 卡与 Intel 的 Chipset 直接相连，该参数为 128 B 或者 256 B。Capric 卡需要根据协商后的 Max_Payload_Size 参数，而不是 Max_Payload_Size Supported 参数，确定存储器写 TLP 有效负载的大小。当 DMA 写的数据区域超过 Max_Payload_Size 参数时，需要进行拆包处理，详见第 12.2.1 节。
- Capric 卡不支持 Phantom 功能。即不能使用 Function 号，进一步扩展 Tag 字段。Phantom 功能的详细说明见第 4.3.2 节。
- Multiple Message Capable 参数为 0b000，即支持一个中断向量。
- Max_Read_Request_Size 参数为 0b010，即存储器读请求 TLP 一次最多能够从目标设备中读取 512 B 大小的数据。如果 DMA 读的数据区域超过 512 B 时，需要进行拆包处理，详见第 12.2.2 节。

系统软件在 Capric 卡初始化时，将分析 Capric 卡的配置空间，并填写 Capric 卡的配置寄存器空间。值得注意的是，系统软件对 Capric 卡进行配置时，Capric 卡将保留该设备在 PCI 总线树中的 Bus Number、Device Number 和 Function Number，LogiCORE 使用寄存器 cfg_bus_number[7:0]、cfg_device_number[4:0]和 cfg_function_number[2:0]存放这组数值，当 LogiCORE 发起存储器读请求 TLP 时，需要使用这组数值。

在设备驱动程序中，Capric 卡需要执行以下步骤完成硬件初始化。

（1）向 DCSR1 寄存器的 init_rst_o 位写 1。

（2）延时 5 μs。

（3）向 DCSR1 寄存器的 init_rst_o 位写 0。

（4）向 DCSR1 寄存器的 int_rd_enb 和 int_wr_enb 位写 1，使能 DMA 读写中断请求。

12.1.3 DMA 写

Capric 卡使用 DMA 写过程将 Capric 卡 SRAM 中的数据发送到 HOST 处理器。在设备驱动程序中，DMA 写过程如下所示。

（1）填写 WR_DMA_ADR 寄存器，注意填写的是 PCI 总线域的地址。

（2）填写 WR_DMA_SIZE 寄存器，以字节为单位。

（3）填写 DCSR2 寄存器的 mwr_start 位，启动 DMA 写。

（4）等待 DMA 写完成中断后，结束 DMA 写。如果系统软件屏蔽了 DMA 写完成中断，可以通过查询 INT_REG 寄存器的 wr_done_clr 位判断 DMA 写是否已经完成。在 Capric 卡中，上一次 DMA 写操作没有完成之前，不能启动下一次 DMA 写操作。

（5）最后将 wr_done_clr 位清零。

从硬件设计的角度来看，DMA 写过程较为复杂。Capric 卡需要通过 DMA 控制逻辑，组织一个或者多个存储器写 TLP，将 SRAM 中的数据进行封装然后传递给发送部件，再由发送部件将数据传送到 LogiCORE，最后由 LogiCORE 将存储器写 TLP 传递给 RC。

如果一次 DMA 写所传递的数据超过了 512 B[⊖]，那么 DMA 控制逻辑需要传递多个存储器写 TLP 给发送部件，才能完成一次完整的 DMA 写操作。而且在 DMA 操作中需要进行数据对界。其详细实现过程见第 12.2.1 节。

12.1.4 DMA 读

Capric 卡使用 DMA 读过程将主存储器中的数据读到 Capric 卡片内 SRAM 中。在设备驱动程序中，DMA 读过程如下所示。

（1）填写 RD_DMA_ADR 寄存器，注意此处填写的是 PCI 总线域的地址。

（2）填写 RD_DMA_SIZE 寄存器，以字节为单位。

（3）填写 DCSR2 寄存器的 mrd_start 位，启动 DMA 读。

（4）等待 DMA 读完成中断产生后，结束 DMA 读。如果系统软件屏蔽了 DMA 读完成中断，则系统软件可以通过查询 INT_REG 寄存器的 rd_done_clr 位判断是否 DMA 读已经完成。在 Capric 卡中，上一次 DMA 读操作没有完成之前，不能启动下一次 DMA 读操作。

（5）最后将 rd_done_clr 位清零。

从硬件设计的角度来看，DMA 读过程比 DMA 写过程复杂。PCIe 总线使用 Split 方式实现存储器读。Capric 卡的 1 次 DMA 读操作使用两种 TLP 报文，并通过发送部件和接收部件协调完成。

⊖ LogiCORE 规定 Max_Payload_Size 为 512 B。

（1）首先 DMA 控制逻辑组织一个或者多个存储器读请求 TLP，然后由发送部件将存储器读请求 TLP 传递给 RC。

（2）RC 正确接收到这个请求报文后，使用一个或者多个存储器读完成 TLP 将数据传递给 Capric 卡。

（3）Capric 卡接收逻辑从 RC 中获得这些存储器读完成 TLP 时，需要首先处理乱序，之后完成一个 DMA 读操作，并向 RC 提交中断请求。

如果一次 DMA 读请求的数据大于 512 B[㊀]时，DMA 控制逻辑需要发送多个存储器读请求 TLP 给 RC，而且在 DMA 读操作中需要进行数据对界。尤其值得注意的是这几个 TLP 的 Tag 字段不能相同，为此硬件逻辑必须正确维护存储器读请求使用的 Tag 字段。DMA 读操作的详细实现过程见第 12.2.2 节。

12.1.5 中断请求

Capric 卡使用 MSI 机制提交中断请求，并只使用了一个中断向量处理 DMA 读/写完成和错误处理。本章为简便起见，忽略了错误处理的过程，但是在一个实际的设计中，错误处理及恢复过程非常重要。

当 DCSR1 寄存器的 int_rd_enb 和 int_wr_enb 位为 1，而且 int_wr_msk 和 int_rd_msk 不为 1 时，DMA 读写完成后，Capric 卡将向处理器提交中断请求。当 DMA 读写完成后，硬件逻辑将 INT_REG 寄存器的 int_asserted 位置为 1，表示有中断请求。此时系统软件需要进一步查询 INT_REG 寄存器的 int_src_rd 和 int_src_wr 位[㊁]，判断该中断请求为 DMA 读完成还是 DMA 写完成，其步骤如下。

（1）如果 int_asserted 位为 1，表示 Capric 卡提交了一个中断请求；否则转（6）。

（2）int_src_rd 位为 1，表示 Capric 卡提交了一个 DMA 读完成中断请求，否则转（4）。此时 rd_done_clr 位也应该为 1。

（3）进行 DMA 读完成处理。向 rd_done_clr 位写 1，清除 DMA 读完成请求位，转（6）。

（4）如果 int_src_wr 位为 1，表示 Capric 卡提交了一个 DMA 写完成中断请求。此时 wr_done_clr 位也应该为 1。

（5）进行 DMA 写完成处理。向 wr_done_clr 位写 1，清除 DMA 读完成请求位。

（6）结束。

以上过程仅为一个简单的中断服务例程的执行流程，一个具体设备驱动程序在 DMA 读写完成后，将检查一些返回状态，以确定 DMA 读写是否正确结束。

12.2 Capric 卡的数据传递

本节主要介绍系统软件在启动 DMA 读写操作时，硬件逻辑的工作过程，包括软件启动 DMA 写时，Capric 卡如何向 RC 发送存储器写 TLP；软件启动 DMA 读时，Capric 卡如何向

㊀ LogiCORE 规定 Max_Read_Request_Size 为 512 B。

㊁ Capric 卡也可以使用多个 MSI 报文（Multiple Message），其中 DMA 写完成和读完成分别对应一个 MSI 报文，而无需查询，目前 Linux 系统并不支持 PCIe 设备的 Multiple Message 功能。

RC 发送存储器读请求 TLP，以及 Capric 卡如何接收来自 RC 的存储器读完成 TLP。

如果考虑 DMA 读写操作的数据对界，DMA 读写操作的实现较为复杂。为此本节定义了两种操作处理对界问题。

（1）向前 X 字节对界

向前对界操作使用 $Head_X$（Y）函数，其中 Y 参数对应某个物理地址；而 X 参数对应对界单位，其值必须为 2 的幂。该函数的计算方法如公式 12-1 所示。

$$Head_X(Y) = Y - (Y \bmod X) \tag{12-1}$$

由以上公式，可以得出 $Head_4(0x1000) = 0x1000$，而 $Head_4(0x1007) = 0x1004$。该操作非常适合硬件实现，在硬件实现中 $Head_X(Y) = Y_nY_{n-1}\ldots Y_m0_{m-1}0_{m-2}\ldots 0_10_0$。如果 Y 的长度为 32b，则 n 等于 31，而 m 等于 $Log_2(X)$。因此在硬件逻辑中，只要将 Y 的第 0～m－1 位清零即可。

（2）向后 X 字节对界

向后对界操作使用 $Tail_X(Y)$函数，其中 Y 参数对应某个物理地址；而 X 参数对应对界单位，其值必须为 2 的幂。该函数的计算方法如公式 12-2 所示。

$$Tail_X(Y) = Head_X(Y) + X - 1 \tag{12-2}$$

由以上公式，可以得出 $Tail_4(0x1000) = 0x1003$，而 $Tail_4(0x1007) = 0x1007$。该操作也非常适合硬件实现，在硬件实现中 $Tail_X(Y) = Y_nY_{n-1}\ldots Y_m1_{m-1}1_{m-2}\ldots 1_11_0$。如果 Y 的长度为32b，则n等于31，而m等于$Log_2(X)$。因此在硬件逻辑中，只要将 Y 的第0～m－1位置 1 即可。

12.2.1 DMA 写使用的 TLP

当软件启动 DMA 写过程后，DMA 控制逻辑将组织存储器写 TLP 发送给 RC。PCIe 总线使用 Posted 总线事务发送存储器写 TLP。Capric 卡使用 4DW 长度的 TLP 头，即使用 64 位地址编码格式。存储器写 TLP 由一个通用 TLP 头加上若干数据字段组成，存储器写 TLP 格式如图 12-2 所示。

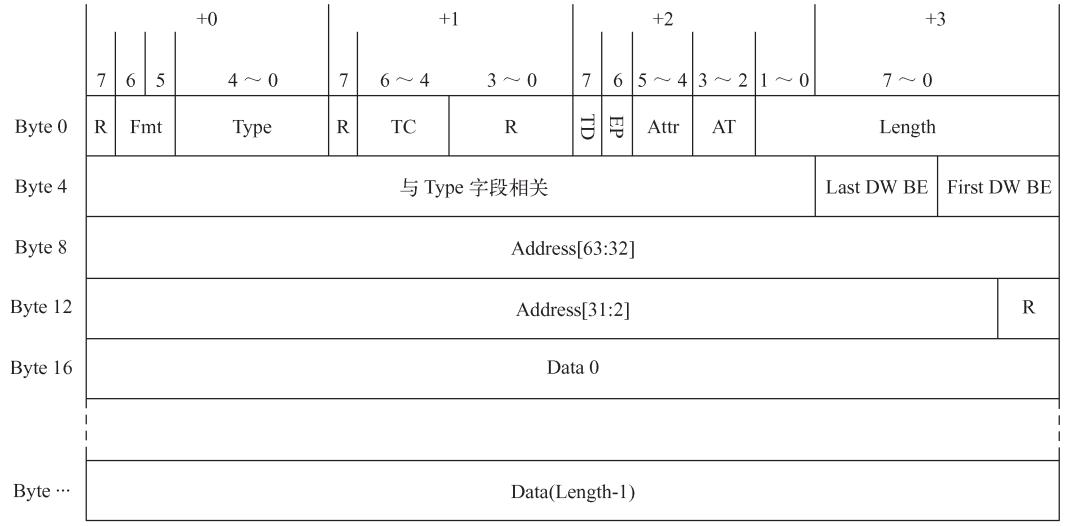

图 12-2　存储器写请求 TLP

1. DMA 写操作使用的实际长度

DMA 写逻辑首先从 WR_DMA_ADR 寄存器中获得起始地址 A($A_{31}A_{30}...A_1A_0$)，然后从 WR_DMA_SIZE 寄存器中获得传送长度 L($L_{15}L_{14}...L_1L_0$)，并由 A 和 L，通过计算获得本次 DMA 写的结束地址 B($B_{31}B_{30}...B_1B_0$)，其值如公式 12-3 所示。

$$B = A + L - 1 \tag{12-3}$$

系统软件需要保证 B 的计算结果不会出现进位，而 DMA 写逻辑由公式 12-3 获得本次数据传送的地址范围 A ~ B。在存储器读写 TLP 中，Length 字段以 DW 为单位，因此向 A ~ B 这段数据区域进行 DMA 写时，Capric 卡实际上需要向 $Head_4$(A) ~ $Tail_4$(B)这段数据区域进行 DMA 写操作，同时使用 TLP 的 First DW BE 和 Last DW BE 字段进行对界处理。

如果一次 DMA 写操作向 0xFFF0 - 0003 ~ 0xFFF0 - 0200 这段区域传送数据时，虽然这段数据区域的长度为 0x1FE 字节[⊖]，但是由于 TLP 的 Length 字段以 DW 为基本单位，因此 Capric 卡需要向 0xFFF0 - 0000 ~ 0xFFF0 - 0203 这段区域进行写操作，然后通过 First DW BE 和 Last DW BE 字段屏蔽 0xFFF0 - 0000 ~ 0xFFF0 - 0002 和 0xFFF0 - 0201 ~ 0xFFF0 - 0203 这两段数据区域。

因此在 L 中存放的长度（A ~ B 数据区域的长度）并不是该 TLP 使用的实际长度。在该 TLP 中使用的实际长度为 $Head_4$(A) ~ $Tail_4$(B)这段数据区域的长度。本节使用 M（$M_{15}M_{14}...M_1M_0$）保存 TLP 中使用的实际长度，其值如公式 12-4 所示。

$$M = (Tail_4(B) - Head_4(A) + 1) \gg 2 \tag{12-4}$$

使用以上公式可以较为方便地得出 DMA 写操作使用的实际长度，但是公式 12-3 和公式 12-4 中使用了多个 32 位的加法器，非常耗费 FPGA 的内部资源。为此本次设计使用了另外一种算法计算 M 的值，如公式 12-5 所示。

$$M' = (Head_4(00A_1A_0 + 00L_1L_0 + 3) - Head_4(00A_1A_0))$$

$$M = L \gg 2 + M' \gg 2 \tag{12-5}$$

我们可以使用公式 12-5 计算 0xFFF0 - 0003 ~ 0xFFF0 - 0200 这段区域使用的实际长度 M。在这段区域中，A 为 0xFFF0 - 0003，而 L = 0x1FE，因此 M′等于 8，而 M = 0x7F + 0x2 = 0x81。该结果与公式 12-4 计算结果相等。但是在公式 12-5 中，M′的计算仅使用 4 位加法器，其实现代价比公式 12-4 所耗费的资源少得多，更为重要的是计算速度也比公式 12-4 快得多。

在 LogiCORE 中，Max_Payload_Size Supported 参数的最大值为 512 B。但是链路两端经过协商后，实际确认的 Max_Payload_Size 参数可能小于 512 B，在多数 x86 处理器系统中，该参数为 128B，因此下文假设 Max_Payload_Size 参数为 128 B。

当 M 大于 0x20（即数据区域的实际长度超过 128 B）时，Capric 卡进行 DMA 写时需要发送多个存储器写请求 TLP，而 M 小于或等于 0x20 时仅需要发送 1 个存储器写请求 TLP。下文分别讨论这两种情况。

2. M 小于或等于 0x20

Capric 卡向数据区域［$A_{31}A_{30}...A_1\ A_0$ ~ $B_{31}B_{30}...B_1\ B_0$］进行 DMA 写操作时，如果通过公式 12-5 的计算发现 M 小于或等于 0x20，DMA 控制逻辑将组织 1 个或者 2 个存储器写

⊖ 数据区域的大小为数据尾地址 - 数据首地址 + 1，即 0xFFF0 - 0200 - 0xFFF0 - 0003 + 1 = 0x1FE。

TLP 传递给 LogiCORE。

如果这个 TLP 所传递的数据区域跨越了 4KB 边界，将组织 2 个存储器写 TLP，因为 PCIe 总线规定被传送的数据区域不能跨越 4KB 边界；如果没有跨越 4KB 边界，Capric 卡组织 1 个存储器写 TLP。我们首先讨论这段数据区域没有跨越4KB 边界的情况，此时这个存储器写 TLP 的各个字段如下所示。

- Fmt 字段为 0b10 或者 0b11，表示使用 3DW 或者 4DW 的 TLP 头，而且带有数据。在 Capric 中，Fmt 字段为 0b011。
- Type 字段为 0b00000，表示当前 TLP 为存储器写 TLP。
- TC 字段为 0b000，表示传送类型为 TC0。
- TD 位为 0b0，表示当前 TLP 不含有 ECRC 信息。
- EP 位为 0b0，表示当前 TLP 是正常的，没有出现完整性问题。
- Attr 字段为 0b00，表示当前 TLP 不使用 Relaxed Ordering，由硬件完成 Cache 一致性操作。有关 Cache 一致性的处理见第 3.3 节和第 12.3.6 节，而 Relaxed Ordering 的描述见第 11.4 节。
- AT 字段为 0b00，表示不进行地址转换。
- Length 字段由公式 12-5 计算而来，其值与 M 相等，单位为 DW，最大值为 0x20，即 128 B。假定在 Capric 卡中，Max_Payload_Size 参数为 128 B。在 x86 处理器系统中，多数 RC 的 Max_Payload_Size 参数为 128B。
- Address 字段为 $A_{31}A_{30}...A_2$。Address 字段为 DW 对界的，共由 30 位组成。
- 当 M 不等于 1 时，Last DW BE 字段与 TLP 报文的结束地址有关，更准确地讲，与 B_1 和 B_0 位有关，如下所示。

  B_1B_0 = 0b11，则 Last DW BE 字段为 0x1111。

  B_1B_0 = 0b10，则 Last DW BE 字段为 0x0111。

  B_1B_0 = 0b01，则 Last DW BE 字段为 0x0011。

  B_1B_0 = 0b00，则 Last DW BE 字段为 0x0001。
- 当 M 不等于 1 时，First DW BE 字段与 TLP 报文的起始地址有关，更准确地讲，与 A_1 和 A_0 位有关，如下所示。

  A_1A_0 = 0b00，则 First DW BE 字段为 0x1111。

  A_1A_0 = 0b01，则 First DW BE 字段为 0x1110。

  A_1A_0 = 0b10，则 First DW BE 字段为 0x1100。

  A_1A_0 = 0b11，则 First DW BE 字段为 0x1000。

当 M 等于 1 时，Last DW BE 字段必须为 0b0000；First DW BE 字段的计算与 A_1A_0 和 B_1B_0 相关，其中（First DW BE）A_1A_0 ~（First DW BE）B_1B_0 字段为 1，其他位为 0。这两个字段的关系如表 12-2 所示。

表 12-2　First DW BE 与 A1A0，B1B0 之间的关系

A_1A_0	B_1B_0	First DW BE	Last DW BE
0b00	0b00	0b0001	0b0000
0b00	0b01	0b0011	0b0000
0b00	0b10	0b0111	0b0000

（续）

A_1A_0	B_1B_0	First DW BE	Last DW BE
0b00	0b11	0b1111	0b0000
0b01	0b01	0b0010	0b0000
0b01	0b10	0b0110	0b0000
0b01	0b11	0b1110	0b0000
0b10	0b10	0b0100	0b0000
0b10	0b11	0b1100	0b0000
0b11	0b11	0b1000	0b0000

值得注意的是，当 M 小于或等于 0x20 时，TLP 所传递的报文，其数据区域依然可能会跨越 4 KB 边界。如 Capric 卡向 0xFFFF－0FFF～0xFFFF－1000 数据区域进行 DMA 写时，虽然 M 等于 0x2（实际长度只有两个字节），但是 Capric 卡需要使用两个存储器写请求 TLP，分别向 0xFFFF－0FFF～0xFFFF－00－0FFF 和 0xFFFF－1000～0xFFFF－1000 这两段数据进行写操作。

由此可以发现，当 Capric 卡对 A～B 这段数据区域[㊀]进行 DMA 写时，首先需要判断这段区域是否跨越 4 KB 边界[㊁]。如果跨越则需要向 A～Tail4096（A）和 Head4096（B）～B 这两段数据区域进行写操作，这两段数据区域一定都小于 0x20，因此采用上文描述的方法组织 TLP 报文即可。

3. M 大于 0x20

如果 M 大于 0x20，此时 Capric 卡进行一次 DMA 写操作时，LogiCORE 需要向 RC 发送多个存储器写请求 TLP。我们假设 Capric 卡需要向［$A_{31}A_{30}...A_1A_0$～$B_{31}B_{30}...B_1B_0$］这段数据区域进行 DMA 写操作，而且这段数据区域的 M 大于 0x20。此时 DMA 写逻辑需要进行拆包操作。这个拆包操作需要遵循以下原则。

（1）TLP 传递的数据区域不能跨越 4 KB 边界。

为此 Capric 卡首先需要分析 A～B 这段是否跨越 4 KB 边界。值得注意的是，在 Capric 卡中，一次 DMA 写的长度小于 2048，因此其传递的数据区域至多会跨越一个 4 KB 边界。此时需要向 A～$Tail_{4096}$（A）和 $Head_{4096}$（B）～B 这两段数据区域进行写操作，而且这两段数据区域的 M 都可能大于 0x20。下文采用的拆包方法可以保证在不进行 4KB 边界检查的情况下，保证拆分后的 TLP 不会跨越 4KB 边界。

（2）尽量减少拆分后 TLP 的总个数。

比如，可以将 0x1000～10FF 这段数据区域拆分为 0x1000～107F 和 0x1080～0x10FF，尽量利用 Max_Payload_Size 参数，而不使用更多的 TLP 进行数据传递。

（3）拆分后的 TLP 尽量不跨越 Cache 行边界。虽然 PCIe 总线规范并没有规定拆分 TLP 的方法。但是将 0x1000～10FF 这段数据区域拆分为 0x1000～0x107E，0x107F～0x108F 和 0x1090～0x10FF，从原理上讲是可行的，可是并不合理。

㊀ 这段数据区域的有效长度小于等于 0x80，即 M≤0x20。

㊁ 如果 B＞$Tail_{4096}$（A）时，表示当前数据区域超越 4 KB 边界。

在 Capric 卡中，为了简化设计，当 M 大于 0x20 时将采用以下规则进行拆包处理。

- 第一个 TLP 的起始地址必须为 $0xA_{31}A_{30}...A_2$，而其他 TLP 的起始地址必须为 0x80 字节对界。
- 最后一个 TLP 的结束地址必须为 $0xB_{31}B_{30}...B_2$，而其他 TLP 的结束地址必须为 0x80 字节对界。

根据以上规则，我们可以将 $A_{31}A_{30}...A_1\ A_0 \sim B_{31}B_{30}...B_1\ B_0$ 这段数据区域划分为多个数据区域，其中每一个区域的 Length 字段不超过 0x20，而且每段数据区域以 128 B 对界。

$A_{31}A_{30}...A_1\ A_0 \sim Tail_{128}(A)$

$Head_{128}(A+128) \sim Tail_{128}(A+128)$

…

$Head_{128}(A+n\times128) \sim Tail_{128}(A+n\times128)$

…

$Head_{128}(B) \sim B_{31}B_{30}...B_1\ B_0$

以上这些数据区域的 M 都小于或等于 0x20，而且都不会跨越 128B 边界，因此也不可能跨越 4 KB 边界。向这些数据区域传送数据时，TLP 各字段的设置参见 M 小于或等于 0x20 的情况。当 Capric 卡将这些存储器写请求 TLP 发送完毕后，可以向处理器提交中断请求。

12.2.2 DMA 读使用的 TLP

与 DMA 写模块相比，DMA 读模块的逻辑设计较为复杂。在 PCIe 总线中，存储器写 TLP 使用 Posted 总线传送方式，实现 DMA 写操作只需要使用存储器写 TLP 即可。而 PCIe 总线使用 Split 总线传送方式进行存储器读操作。因此一个 DMA 读过程由 EP 向 RC 发送“存储器读请求 TLP”，之后再由 RC 使用“存储器读完成 TLP”将数据传递给 EP。

当软件启动 DMA 读操作后，DMA 控制逻辑将根据需要读取数据区域的大小，决定发送存储器读请求 TLP 的个数，如果所读取数据区域的实际长度超过 Max_Read_Request_Size 参数[㊀]时，DMA 控制逻辑需要进行拆包处理，向 RC 发送多个存储器读请求 TLP，这些存储器读请求 TLP 将使用不同的 Tag。

当 RC 收到这些存储器读请求 TLP 后，将使用存储器读完成 TLP，将数据传递给 Capric 卡。其中一个存储器读请求 TLP（Tag 不同的报文）可能对应多个存储器读完成 TLP，而且这些存储器读完成 TLP 可以乱序到达。

在 Capric 卡的 DMA 读模块的设计中，首先需要进行拆包处理，其次需要合理地管理 Tag 资源，而最值得注意的是对存储器读完成 TLP 的乱序处理。为此在 Capric 卡中设置了一个单向循环链表 tag_queue，以便 Capric 卡发送存储器读请求 TLP，进行 Tag 资源的管理，并处理存储器读完成 TLP 的序。

1. tag_queue

Capric 卡的 DMA 读模块使用了一个单向循环队列 tag_queue，当然设计者也可以使用其他逻辑实现同样的功能。实际上对于 Capric 卡而言，设置这样的循环队列是奢侈的，因为

㊀ 在 LogiCORE 中该参数为 512 B。

Capric 卡仅实现了基本的 DMA 读写操作。该循环队列实际上是为笔者的另一个设计，即 Cornus 卡[⊖]准备的。

在 tag_queue 队列中，设置了头尾指针，分别为 tag_front 和 tag_rear，Capric 卡使用 8 位[⊖]寄存器存放这两个指针。该队列的每一个 Entry 对应一个 tag 资源，其 Entry 号与 Tag 号一一对应。DMA 读模块从 tag_rear 指针获得当前可以使用的 tag 资源（相当于将获得的 tag 字段加入 tag_queue 中），并从 tag_front 指针处释放 tag 资源（相当于从 tag_queue 的头部释放资源），在 tag_front 和 tag_rear 之间的 Entry 保存正在使用的 tag 资源。tag_queue 队列的组成结构如图 12-3 所示。

在 Capric 卡复位时，tag_front 与 tag_rear 指针同时指向 Entry 0，此时 tag_queue 为空。当 Capric 卡需要使用 tag 资源时，首先判断 tag_queue 是否为满，如公式 12-6 所示。

$$(tag\_rear + 1) \bmod 256 = tag\_front \tag{12-6}$$

当 tag_queue 队列不满时，Capric 卡可以从 tag_queue 中获得 tag 资源，其值等于 tag_rear，然后将 tag_rear 更新为（tag_rear + 1）mod 256（相当于将获得的 tag 加入到 tag_queue 队列中）。当 Capric 卡释放 tag 资源时，需要判断 tag_queue 是否为空，如公式 12-7 所示。

$$tag\_rear = tag\_front \tag{12-7}$$

在 Capric 卡中，到达的存储器读完成 TLP 因为乱序的原因，其 tag 字段不一定与 tag_front 相等。Capric 卡错误处理逻辑需要判断到达的存储器读完成 TLP 的 tag 字段是否在 tag_front 和 tag_rear 之间（在图 12-3d 中，阴影部分的 Entry 在当前 tag_queue 中有效），其判断条件如公式 12-8 所示。

$$(tag - tag\_front) \bmod 256 < (tag\_rear - tag\_front) \bmod 256 \tag{12-8}$$

在 Cornus 卡中，tag_queue 队列的 Entry 由许多字段组成，在这些字段中“L”和“U”位对于 Capric 卡有意义。其中 U 为 Used 位，当该位为 1 时，表示对应 Entry 正在被使用，为 0 时表示没有被使用；而 L 为 Last 位，当该位为 1 时，表示对应 Entry 保存 DMA 操作最后一个存储器读请求 TLP。这些位的详细解释见下文。

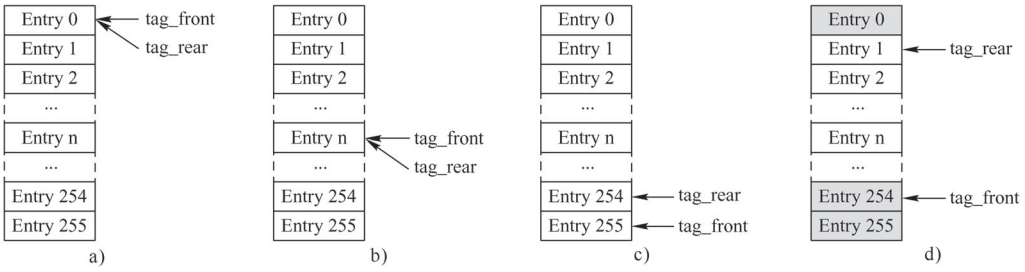

图 12-3　循环队列 tag_queue 的组成结构

a）tag_queue 的初始化　b）tag_queue 为空　c）tag_queue 为满　d）tag_queue 中的有效 Entry

2. Capric 卡发送存储器读请求 TLP

Capric 卡发送存储器读请求 TLP 与发送存储器写 TLP 的步骤较为类似，只是存储器读请求不含有 Data Payload，存储器读请求 TLP 的格式如图 6 - 8 所示。与存储器写请求 TLP 相

⊖ Cornus 卡是一个基于 PCIe 总线的以太网卡，支持多通路 DMA 读写操作。

⊖ 因为 tag_queue 队列的长度为 256。

比，存储器读报文多了两个字段分别为 Requester ID 字段和 Tag 字段，这两个字段合称为 Transaction ID 字段，该字段的结构如图 6 -9 所示。

从第 6.2.1 节中，可以获知存储器读请求 TLP 使用地址路由方式，Capric 卡将存储器读请求 TLP 发送给 RC 时，并不需要使用 Transaction ID 字段进行 ID 路由。但是存储器读完成 TLP 需要使用 ID 路由方式进行传送，因而需要使用 Capric 卡的 Transaction ID 字段将存储器读完成 TLP 发送给 Capric 卡。

在 PCIe 总线中，每一个数据传送都有唯一的 Transaction ID，Transaction ID 由 Requester ID 和 Tag 字段组成，其中 Requester ID 由 HOST 处理器在系统初始化时设置。Capric 卡需要记录这个 Requester ID，以便传送存储器读请求 TLP。

在存储器读请求 TLP 中其他字段的设置与存储器写请求 TLP 类似，这些字段的设置参见上文。其中存储器读请求 TLP 的 Fmt 字段为 0b00/0b01，表示使用 3DW/4DW 的 TLP 头，而且不带数据；而 Type 字段与存储器写请求 TLP 的 type 字段相等，都为 0b0000。在 PCIe 总线中，存储器写报文一定带有数据，而存储器读请求一定不带数据，这是 PCIe 总线区分存储器读请求 TLP 和存储器写 TLP 的方法。在设计中，Capric 卡使用 4DW 的读请求 TLP 头，因为 Capric 卡内部使用 64b 数据总线，使用 4DW 的报文头便于对界处理；而存储器读完成报文只能使用 3DW 的报文头，这为 Capric 卡的设计也带来了一些困难。

在存储器读请求 TLP 中，需要重点处理的字段是 Tag。在 PCIe 总线中，每个设备都有唯一的 Requester ID，而且每一次数据传送使用不同的 Transaction ID，在一次数据传送没有完成之前，其他数据传送不能使用相同的 Transaction ID。在 PCIe 总线中，使用 Tag 字段区分不同的 Transaction ID，因为对于同一个 PCIe 设备发出的 TLP，Requester ID 字段都是相同的，只有 Tag 字段不同。

当 Capric 卡向 RC 发送存储器读请求 TLP 时，将从 tag_queue 中选择一个未用的 Tag 资源；当 Capric 卡收齐与“存储器读请求 TLP”对应的“存储器读完成 TLP”[⊖]后，将释放这个 Tag 资源，之后其他存储器读请求 TLP 可以使用这个 Tag。

为此 Capric 卡需要使用一组数据缓冲维护这个 Tag 字段。在 PCIe 总线中，Tag 字段为 5 或者 8 位，如果使能了 Phantom 功能，一个 PCIe 设备可以使用更多的 Tag 资源，详见第 4.3.2 节。在 Capric 卡中，并没有使能 Phantom 功能，而且使用的 Tag 字段为 8 位，即 Device Capability 寄存器的 Extended Tag Field Supported 位为 1，该寄存器的详细描述见第 4.3.2 节。

Capric 卡使用 tag_rear 指针从 tag_queue 队列中获得未用的 tag 资源，并设置 tag_queue 中对应 Entry 的 L 和 U 位。Capric 卡发送存储器读请求 TLP 时，首先需要判断本次 DMA 读操作一共需要向 RC 发送几个存储器读请求 TLP。在 Capric 卡中，一次 DMA 读可以使用的最大传送单位为 2047B，该值超过 Capric 卡的 Max_Read_Request_Size 参数（512 B），因此 Capric 卡发送存储器读请求 TLP 时需要进行拆包操作。

如果 Capric 卡读取 A($A_{31}A_{30}...A_1\ A_0$) ~ B($B_{31}B_{30}...B_1\ B_0$) 这段数据区域时，需要首先计算每段数据区域的实际长度 M，该长度的计算与公式 12-4 相同。如果 M 小于或等于 512 B，需要继续检查这段数据区域是否超过 4 KB 边界，如果超过需要将这段数据区域分为 A ~

⊖ 一个存储器读请求 TLP 可能对应若干个存储器读完成 TLP。

$Tail_{4096}(A)$和 $Head_{4096}(B)\sim B$ 这两段数据区域进行读操作。

如果 M 大于 512 B，Capric 卡需要进行拆包处理，将 A ~ B 这段数据区域分割为若干个数据区域，其中每一段数据区域的 Length 字段不超过 0x80，而且为 512B 对界。Capric 卡使用的拆包方法如下所示。

第 1 个存储器读请求 TLP 对应的数据区域：$A_{31}A_{30}\ldots A_1\ A_0\sim Tail_{512}(A)$

第 2 个存储器读请求 TLP 对应的数据区域：$Head_{512}(A+512)\sim Tail_{512}(A+512)$

…

第 n 个存储器读请求 TLP 对应的数据区域：$Head_{512}(A+n\times512)\sim Tail_{512}(A+\ n\times512)$

…

最后 1 个存储器读请求 TLP 对应的数据区域：$Head_{512}(B)\sim B_{31}B_{30}\ldots B_1\ B_0$

以上这些数据区域的 M 都小于 0x80，而且都不会跨越 512 B 边界，因此也不可能超过 4 KB 边界。使用以上拆包方法，Capric 卡可以获得若干个 M 小于或等于 0x80 的数据区域，因此 Capric 卡可以使用一个存储器读请求从主存储器获得以上每段数据区域对应的数据。Capric 卡使用 4DW 的 TLP 头，其格式如图 12-4 所示。

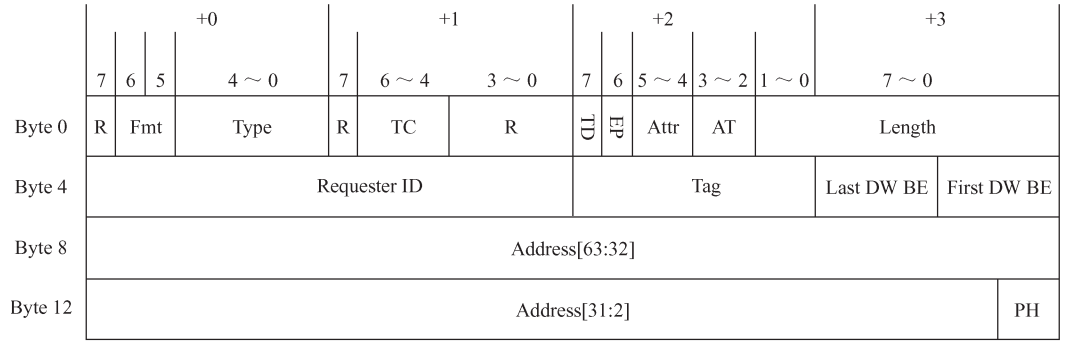

图 12-4　存储器读请求 TLP 头格式

Capric 卡向 RC 发送这些存储器读请求 TLP 的详细步骤如下。

（1）组织存储器读请求 TLP，其中 Byte 0 中的字段、First/Last DW BE 字段和 Address 字段与存储器写请求 TLP 的对应字段相同，本小节对此不做详细描述。而 Requester ID 字段由 Host 处理器在初始化时设置。

（2）Capric 卡从 tag_queue 队列中获得 tag 字段，首先通过公式 12-6 判断 tag_queue 队列是否有可用 tag 资源，如果没有则循环等待公式 12-6 成立；如果有可用 tag，则继续。在 Capric 卡中，Address[63:2]较小的存储器读请求使用的 Tag 字段也较小。

（3）当前存储器读请求使用 tag_rear 作为 tag 字段，同时置 tag_queue[tag_rear].U 为 1，表示当前 Entry 已被使用。

（4）在一次 DMA 读操作时，Capric 卡可能需要进行拆包操作。如果当前存储器读请求 TLP 是最后一个 TLP，则将 tag_queue[tag_rear].L 位置为 1，否则置为 0。

（5）将 tag_rear 赋值为（tag_rear + 1）mod 256，然后发送该存储器读请求 TLP。如果 L 位为 0 时转（1），表示与当前 DMA 操作对应的存储器读请求 TLP 还没有发送完毕；否则结束存储器读请求报文的发送。

Capric 卡发送存储器读请求 TLP 时，还需要考虑一个细节问题。在 PCIe 总线中，EP 为

CplH 和 CplD 提供的 Credit 为 0，即 Infinite Credit，详见第 9.3.2 节。这意味着 EP 每发送一个存储器读请求，必须为对应的存储器读完成的报文头和数据预留缓冲。

假设 Capric 卡连续向 RC 发送了 256 个存储器读请求 TLP，其中每个存储器读请求 TLP 访问的数据区域为 512B，而在 RC 发送的存储器读完成 TLP 中一次只能携带 64B。此时即便不考虑对界的问题，Capric 卡也需要为存储器读完成预留较大的缓冲空间，该空间由两部分组成，如下所示。

（1）预留存储器读完成 TLP 头的空间大小为 8192 B（256 ×512 ×4/64）。

（2）预留存储器读完成 TLP 数据的空间大小 128 KB（256 ×512 B）。

在硬件设计中，为了提高 DMA 读的数据传送效率，还可以使能 Phantom 功能，此时 EP 能够发送的存储器读请求 TLP 更多。如果 EP 经过若干级 Switch 后，才能到达 RC，此时 RC 可能正在处理其他 EP 的存储器读请求而不会立即处理这些存储器读请求，此时该 EP 可能长时间不能收到存储器读完成 TLP，从而无法释放预留的数据缓冲。因此 EP 可能会因为没有数据缓冲，而无法继续发送存储器读请求 TLP。

实际上，硬件为存储器读完成报文预留的数据缓冲是有限的，一般不会预留 136 KB 大小的空间，在 LogiCORE 中，为 CplH 预留的缓冲单元为 33 ~ 36 个，而为 CplD 中的数据预留的缓冲为 2176 ~ 2304 B。

由此可以发现如果 RC 没有及时地将存储器读完成 TLP 发送回来，Capric 卡最多在连续发送 33 个存储器读请求后（假设存储器读请求使用 4DW 的报文头），就因为无法为存储器读完成的报文头提供足够的缓冲而不能继续发送；此外如果每个存储器读请求所访问的数据区域都是 512 B，Capric 卡最多在连续发送 4 个这样存储器读请求后，就不能继续发送这样的存储器读请求，从而造成发送流水线的中断。

由此可以发现，在 LogiCORE 中，由于预留的缓冲有限，Capric 卡在使用 PCIe 总线要求的 Infinite Credit 机制时，将因为预留缓冲不足而造成流水线的中断。为此 LogiCORE 提供了三种流量控制机制，这三种流量控制机制在重构 LogiCORE 时选择使用，系统软件不能通过修改寄存器动态配置这些流控方式。

（1）Infinite Credit

该方式与 PCIe 总线规范兼容。如果 Capric 卡使用 Infinite Credit 方式，当 LogiCORE 内部的接收缓冲不足时，Capric 卡不能向 RC 发送存储器读请求报文。根据上文的讨论，由于 LogiCORE 内部的接收缓冲不足，因此使用该方式在某种程度上，将造成 DMA 读流水线的中断，从而影响 DMA 读的效率。

（2）One Posted/Non-Posted Header

该方式与 PCIe 总线规范不兼容，使用这种方式时，EP 每次为上游端口发送的 Posted 请求和 Non-Posted 请求提供最小的 Credit，相当于 EP 每一次只能发送一个存储器读请求 TLP，而得到与之对应的存储器读完成 TLP 后，再提交下一个存储器读请求 TLP。使用这种方法将严重影响 DMA 读的效率。LogiCORE 提供的这种方法可能是用于调试目的。在正常情况下设计者不应该使用这种方式。

（3）Non-Infinite Credit

该方式与 PCIe 总线规范不兼容，使用这种方式时，EP 并没有给上游端口提供无限量的 Credit，而是根据预留接收缓冲的实际使用情况，为上游端口提供 Credit。Capric 卡采用这种

方式，发送存储器读请求 TLP 时，并不会为存储器读完成 TLP 事先预留接收缓冲，从而在发送存储器读请求 TLP 时，并不会因为接收缓冲不足而被中断，因此提高了 Capric 卡发送存储器读请求 TLP 的效率。

LogiCORE 在接收存储器读完成报文时，将根据预留缓冲的实际大小为对端提供 Credit。虽然采用这种方法与 PCIe 总线规范要求的 Infinite Credit 并不兼容。但是使用这种方法避免了在发送存储器读请求时，因为接收缓冲不足而引发的流水线中断，从而 Capric 卡可以连续发送多个存储器读请求，无论是否具有足够的接收缓冲。

而 Capric 卡将以较快的速度从 LogiCORE 的预留缓冲中获得数据，因此在多数时间里，不会因为预留缓冲被对端设备耗尽而引发接收流水线的中断。因此在实际设计中，Capric 卡使用了这种流量控制方式。

3. Capric 卡接收存储器读完成 TLP

在 PCIe 总线中，EP 发出的存储器读请求可以超越之前的存储器读请求，而且当存储器完成报文使用的 Transaction ID 不同时，存储器读完成 TLP 也可能超越之前的存储器读完成 TLP，这将造成存储器读完成 TLP 乱序到达 Capric 卡。

Capric 卡必须注意处理这个乱序问题。下面举例说明这个序的问题，假设 Capric 卡向处理器的 0x1000 ~ 0x11FF 这段数据区域发送存储器读请求 TLP，RC 将通过存储器读完成 TLP 向 Capric 卡传递数据。如果这个 RC 使用的 RCB 为 64 B，则 RC 可以使用 4 个存储器读完成 TLP 发送这些数据，如图 12-5 所示。

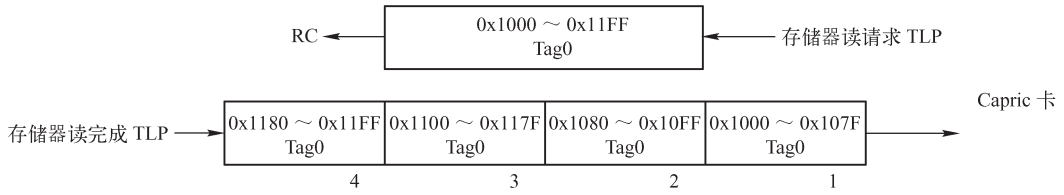

图 12-5　使用一个存储器读 TLP 对 0x100 ~ 0x11FF 进行 DMA 读操作

其中第 1 个存储器读完成 TLP 的数据来自 0x1000 ~ 0x107F 这段数据区域；第 2 个存储器读完成 TLP 的数据来自 0x1080 ~ 0x10FF 这段数据区域；第 3 个存储器读完成 TLP 的数据来自 0x1100 ~ 0x117F 这段数据区域；而第 4 个存储器读完成 TLP 的数据来自 0x1180 ~ 0x11FF 这段数据区域。在这种情况下，存储器读完成报文将按序到达 Capric 卡，从而并不会对 Capric 卡的硬件逻辑造成影响。

如果 Capric 卡使用 2 个存储器读请求 TLP 向处理器的 0x1000 ~ 0x11FF 这段数据区域发起存储器读请求。其中第 1 个存储器读请求 TLP（tag 0）向处理器的 0x1000 ~ 0x10FF 这段数据区域发起存储器读请求，而第 2 个存储器读请求 TLP（tag 1）向处理器的 0x1100 ~ 0x11FF 这段数据区域发起存储器读请求。此时来自 RC 的存储器读完成报文可能乱序到达 Capric 卡，如图 12-6 所示。

此时 RC 依然使用 4 个存储器读完成 TLP 向 Capric 卡发送这些数据，但是由于序的问题，这 4 个存储器读完成 TLP 可能以 2 种不同的顺序发向 Capric 卡。当然 RC 还可以以其他顺序向 Capric 卡发送这些 TLP，本节并不列出所有可能的顺序。

（1）第 1 种序

- 第 1 个存储器读完成 TLP 的数据来自 0x1000 ~ 0x107F 这段数据区域（tag 0）。

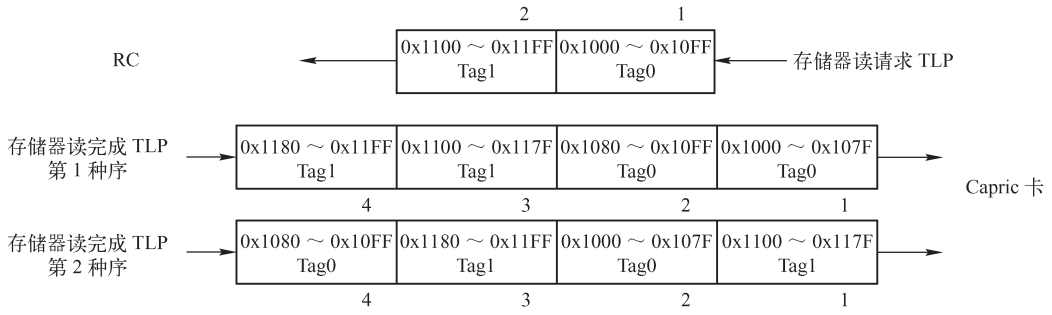

图 12-6　使用两个存储器读 TLP 对 0x100 ~ 0x11FF 进行 DMA 读操作

- 第 2 个存储器读完成 TLP 的数据来自 0x1080 ~ 0x10FF 这段数据区域（tag 0）。
- 第 3 个存储器读完成 TLP 的数据来自 0x1100 ~ 0x117F 这段数据区域（tag 1）。
- 第 4 个存储器读完成 TLP 的数据来自 0x1180 ~ 0x11FF 这段数据区域（tag 1）。

（2）第 2 种序

- 第 1 个存储器读完成 TLP 的数据来自 0x1100 ~ 0x117F 这段数据区域（tag 1）。
- 第 2 个存储器读完成 TLP 的数据来自 0x1000 ~ 0x107F 这段数据区域（tag 0）。
- 第 3 个存储器读完成 TLP 的数据来自 0x1180 ~ 0x11FF 这段数据区域（tag 1）。
- 第 4 个存储器读完成 TLP 的数据来自 0x1080 ~ 0x10FF 这段数据区域（tag 0）。

这个乱序问题为 Capric 卡的 DMA 读机制的设计带来了不小的麻烦。因为在“第 2 种序”的情况下，先发出去的存储器读请求 TLP，后接收到与之对应的存储器完成报文。不过值得庆幸的是，对于一个存储器读请求 TLP，其对应的存储器完成报文虽然也有多个，但是这些报文将以地址顺序先后到达。如向 0x1000 ~ 0x10FF 这段数据区域发送的存储器读请求，其存储器完成报文虽然被分解为两个，但一定是传送 0x1000 ~ 107F 这段区域的存储器读完成 TLP 率先到达，而传送 0x1080 ~ 0x10FF 这段区域的存储器读完成 TLP 随后到达。

在 Capric 卡的设计中必须考虑这个乱序问题，因为 Capric 卡进行 DMA 读操作时，所读取的数据区域可能超过 Max_Read_Request_Size 参数，此时 Capric 卡对这段数据区域进行 DMA 读时，必须向 RC 发出多个存储器读请求 TLP，参见上文。

与 Capric 卡发送存储器读请求 TLP 相比，Capric 卡处理存储器读完成 TLP 的过程更为复杂。当 Capric 卡收到来自 RC 的存储器完成报文后，需要进行一系列检查。存储器读完成 TLP 的格式如图 12-7 所示。

Capric 卡接收到存储器读完成 TLP 后，首先需要检查报文头。其中 Fmt 字段必须为 0b010，Type 字段必须为 0b01010。除此之外 Capric 卡还需要进行以下检查。

（1）存储器读完成 TLP 的 Requester ID 字段必须与 Capric 卡的 Requester ID 字段相等。否则该存储器读完成 TLP 被认为是“Unexpected Completion”报文，Capric 卡需要丢弃该存储器读完成 TLP，并将 ERR 寄存器的 UC 位置 1。

（2）检查存储器读完成 TLP 的 Status 字段，如果 Status 字段不为 0b000，则表示接收到的 TLP 出现错误。如果 Status 字段为 0b001 或者 0b100 时，Capric 卡需要丢弃该存储器读完成 TLP，并将 ERR 寄存器的相应位置 1。

（3）检查存储器读完成 TLP 的 Tag 字段，确认当前报文是否与已经发出的存储器读请求

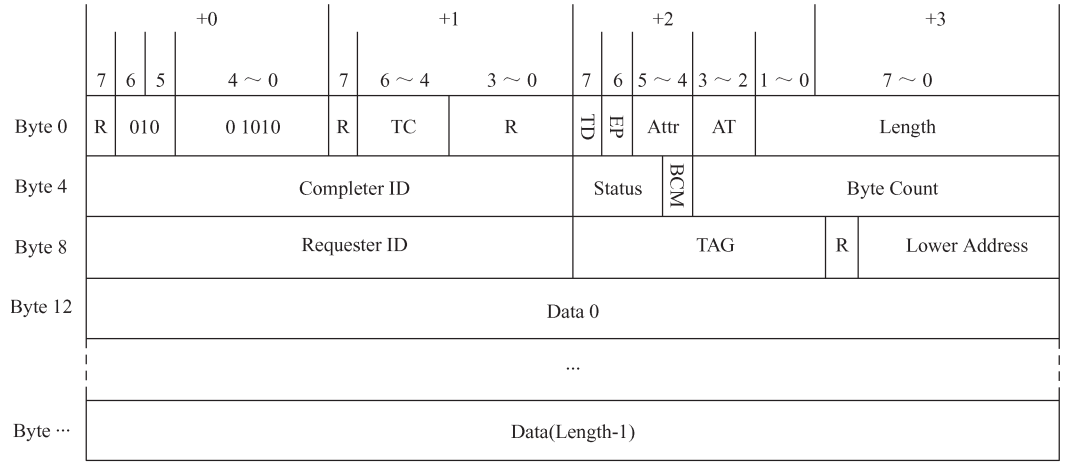

图 12-7　存储器读完成 TLP

TLP 对应，检查方法如公式 12-8 所示。

（4）此外 Capric 卡还需要检查 EP 位，TD 位、TC 字段和 Attr 字段。

Capric 卡的接收部件成功完成这些检查之后，将从存储器读完成 TLP 中获取数据。PCIe 总线规定，一个存储器读请求 TLP，可以对应多个存储器读完成 TLP。这为 Capric 卡的设计带来了一定的困难，为此 Capric 卡需要将存储器读完成 TLP 全部收齐后，才能释放相应的 Tag 资源，最后将 tag_queue 对应 Entry 的 U 位和 L 位清零。

存储器读完成报文虽然可能有多个，但是这些报文将以地址顺序先后到达。因此 Capric 卡首先需要分析 Tag 字段，从而确定当前存储器读完成 TLP 与哪个存储器读请求 TLP 对应。其中第 1 个存储器读完成 TLP 与存储器读请求 TLP 起始地址对应，之后的存储器读完成 TLP 将地址顺序依次到达。假定向［$A_{31}A_{30}...A_1\ A_0 \sim B_{31}B_{30}...B_1\ B_0$］这段数据区域发起存储器读请求时，RC 将发送多个存储器读完成 TLP，并以下列顺序到达。

［$A_{31}A_{30}...A_1A_0 \sim \ Tail_{64}(A_{31}A_{30}...A_1\ A_0)$］

［$Head_{64}(A_{31}A_{30}...A_1A_0 + 64) \sim Tail_{64}(A_{31}A_{30}...A_1\ A_0 + 64)$］

…

［$Head_{64}(A_{31}A_{30}...A_1A_0 + n * 64) \sim Tail_{64}(A_{31}A_{30}...A_1\ A_0 + n * 64)$］

…

［$Head_{64}(B_{31}B_{30}...B_1B_0) \sim B_{31}B_{30}...B_1\ B_0$］

当然 RC 也可能向 Capric 发送一个存储器读完成 TLP，传递［$A_{31}A_{30}...A_1A_0 \sim B_{31}B_{30}...B_1B_0$］数据区域中的所有数据。无论这些存储器读完成 TLP 以什么样的形式到达，Capric 卡都需要正确接收这个存储器读完成 TLP。

Capric 卡首先分析存储器读完成 TLP 的 Length 字段，在该字段中存放当前存储器读完成 TLP 的长度，值得注意的是 Length 字段所存放的长度，可能超过这个存储器完成报文的包含的有效数据长度，因为地址 $A_{31}A_{30}...A_1\ A_0$ 很可能不是 1DW 对界，而 Length 字段存放的最小数据单位为 1DW。此时 Capric 卡必须正确识别存储器读完成 TLP 中 Data0（即第一个双字）中包含的有效数据，以及 Data(Length-1)（即最后一个双字）中包含的有效数据。

在 RC 发送给 Capric 卡的多个存储器读完成 TLP 中，只有第 1 个存储器读完成 TLP 所对应的存储器区域的起始地址可能不是 DW 对界；而如果存在其他存储器读完成 TLP，那么这些报文所对应存储器区域的起始地址至少是 64 B 对界的，也可能是 128B 对界的[⊖]。

但是存储器读完成 TLP 并不含有 First DW BE 字段，此时 Capric 卡需要使用存储器读完成 TLP 中的 Lower Address 字段识别 Data0 中的有效字节。

在第 1 个存储器读完成 TLP 中，Lower Address[1:0] = A_1A_0，对于其他存储器读完成 TLP，其 Low Address[1:0] = 0b00。因此通过 Lower Address 字段，可以识别 Data0 中第一个有效数据，即 Data0[A_1A_0]为第一个有效数据。

存储器读完成 TLP 并没有设置 Last DW BE 字段，Capric 卡需要使用 Byte Count 和 Lower Address 字段联合识别 Data(Length-1)中的有效数据。如果当前存储器读完成 TLP 不是最后一个 TLP,那么其 Data(Length-1)中的数据全部有效。因为 PCIe 总线规定,如果 RC 为 1 个存储器读请求 TLP 发送多个存储器读完成 TLP,如果这个存储器读完成 TLP 不是最后一个报文,那么其结束地址必须 64 B 对界。

如果当前存储器读完成 TLP 不是第 1 个 TLP,那么其 Lower Address[1:0] = 0b00。在这两种情况下,Data(Length-1)中的有效数据较易计算。但是有一个特例情况,就是 RC 只发出了一个存储器读完成 TLP 给 Capric 卡,此时这个 TLP 既是第一个存储器读完成 TLP 也是最后一个存储器读完成 TLP。但是无论是上述哪种方式,依然存在计算 Data(Length-1)中的有效数据的通用方法,如公式 12-9 所示。

$$0bX_1X_0 = LowAddress[1:0] + ByteCount[1:0] - 0b01 \tag{12-9}$$

其中 $Data(Length-1)[X_1X_0]$为存储器读完成 TLP 中最后一个有效数据。Capric 卡计算完毕存储器读完成 TLP 的 Data0 和 Data(Length-1)中的有效数据后，还需要判断当前存储器读完成 TLP 是不是 RC 发出的最后一个与当前 Tag 对应的存储器读完成 TLP。为直观起见，以图 12-8 为例说明如何计算当前存储器读完成 TLP 是否为最后一个报文。

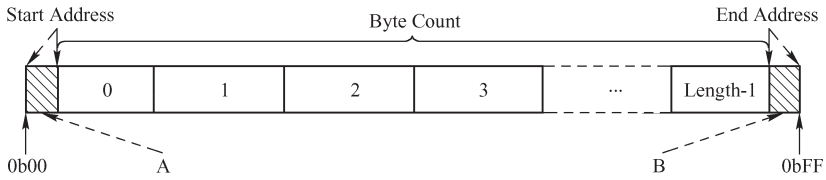

图 12-8　最后一个存储器读完成 TLP 的判断方法

如上图所示，Start Address 为存储器读完成 TLP 的起始地址，而 End Address 为存储器读完成 TLP 的结束地址。在一个存储器读完成 TLP 中，我们无法得到 Start Address 和 End Address 的确切的数值，因为存储器读完成 TLP 不包含 Address 字段，但是可以得到阴影 A 和阴影 B 的大小。其中阴影 A 的大小为 Low Address[1:0]，而阴影 B 的大小为 0b11 - 0bX_1X_0。

如果当前 TLP 的 Byte Count 字段加上阴影 A 和 B 的大小与 Length × 4 相等，即公式 12-10 成立时，该 TLP 为 RC 发给 Capric 卡的最后一个存储器读完成 TLP。

$$(Byte\ Count + Low\ Address[1:0] + 0b11 - 0bX_1X_0) = Length \ll 2 \tag{12-10}$$

⊖ 采用哪种对界方式与 RCB 参数相关，不同处理器使用的 RCB 参数并不相同。

请读者重新阅读第 6.3.2 节，深入理解 Byte Count 参数的含义，以加深对公式 12-10 的理解。在 Capric 卡的硬件设计中，需要使用该公式识别最后一个到达的存储器读完成 TLP。

在 Capric 卡接收到最后一个存储器完成 TLP 之后，将完成一次存储器读请求。当最后一个存储器读请求完成后，将完成一次 DMA 读操作。Capric 卡接收存储器读完成 TLP 的详细步骤如下所示。

（1）首先进行报文检查。如果通过这些检查后，将从存储器读完成报文中获得数据填入相应 SRAM 的对应区域中。

（2）DMA 读逻辑通过存储器读完成 TLP 的 Tag 字段在 tag_queue 中查找对应的 Entry。如果当前存储器读完成是最后一个 TLP，将该 Entry 的 U 位清零。此时如果该 Entry 的 L 位为 1，表示本次 DMA 读结束，并向处理器提交中断请求，同时清除 L 位，并置相应的中断状态寄存器。Cormus 卡支持多路并发的 DMA 读操作，因此需要在 Entry 中设置 L 位。

（3）DMA 读模块可能会更新 tag_front 指针，如果 Tag 字段不等于 tag_front 指针，读模块不能更新 tag_front，而仅是将对应 Entry 的 U 位清零；如果相同则将 tag_front 更新为（tag_front + 1）mod 256，同时将 U 位清零。

（4）之后 DMA 读模块继续判断 tag_queue[tag_front]的 U 位是否为 0。如果该位为 0，将 tag_front 更新为（tag_front + 1）mod 256，然后继续判断 Tag_queue[tag_front]的 U 位是否为 0，直到公式 12-7 成立，或者 Tag_queue[tag_front]的 U 位为 1。

12.2.3 Capric 卡的中断请求

Capric 卡支持两种中断请求方式，一种是 Legacy INTx 方式，另一种是 MSI 中断方式。Capric 卡需要向 RC 发送两个 Legacy INTx 中断消息，一个是 Assert INTx，另一个是 Deassert INTx，以实现 Legacy INTx 中断方式。第 6.3.4 节详细介绍了这种中断请求方式。这种中断请求方式虽然使用了 INTx 消息，但是其原理与电平触发方式类似，而 MSI 中断方式的工作原理与边沿触发方式类似。因此系统软件对 Capric 卡的这两种中断请求的处理并不相同。

在第 10.3 节中，我们曾详细讨论了电平触发与边沿触发的区别。其中采用电平触发不会丢失中断请求，而采用边沿触发将会丢失中断请求。Capric 卡可以保证即便使用了 MSI 中断机制，也不会丢失中断请求。

MSI 中断机制使用存储器写 TLP 实现，这个存储器写 TLP 的目的地址为 MSI Capability 结构中的 Message Address 字段，而数据为 Message Data 寄存器中的值。Message Address 字段和 Message Data 字段由系统软件在初始化时填写。在不同的处理器体系结构中，系统软件填写的这两个字段的数据并不相同，详见第 10.2 节和第 10.3 节。

LogiCORE 内部实现了 MSI 中断机制，Capric 卡仅需一些简单的组合逻辑即可实现 MSI 中断机制。Capric 卡需要根据 INT_REG 寄存器的信息，决定如何发送中断请求，这部分中断逻辑的实现较为简单，本节对此不做进一步说明。

12.3 基于 PCIe 总线的设备驱动

本节简要介绍 Capric 卡在 Linux 系统中使用的 Char 类型设备驱动程序。为便于读者理解 Linux 系统 PCIe 总线驱动程序的实现构架，本节将详细介绍 Capric 卡的初始化、DMA 读、

DMA 写、中断服务例程和关闭过程，但是并不会过多介绍和 Linux 系统相关的知识，而是重点介绍系统软件如何管理和配置 PCIe 设备。

12.3.1 Capric 卡驱动程序的加载与卸载

在 Linux 系统中，Capric 卡驱动程序的加载与卸载的过程如源代码 12-1 所示。这部分程序并不会直接操作 PCIe 设备，而是通过 pci_register_driver 函数向内核注册一个 pci_driver 结构，即 capric_drv，并由 capric_probe 函数完成 Capric 卡的初始化。

源代码 12-1　Capric 卡驱动程序的加载与卸载

```
static struct pci_device_id capric_ids[ ] = {
        {PCI_DEVICE( PCI_VENDOR_ID_XILINX, PCI_DEVICE_ID_EP_PIPE),},
        {0,}
};
…
static struct pci_driver capric_drv = {
        .name = DEV_NAME,
        .id_table = capric_ids,
        .probe = capric_probe,
        .remove = capric_remove,
};

static int __init capric_init( void)
{
        int result;

        result = pci_register_driver( &capric_drv);
        return result;
}

static void capric_exit( void)
{
        pci_unregister_driver( &capric_drv);
}

module_init( capric_init);
module_exit( capric_exit);
```

在上述源代码中，pci_register_driver 函数的主要作用是将 capric_drv 结构与 PCI 设备的 pci_dev 结构[㊀]进行绑定，并在初始化时执行 capric_probe 函数，而在结束时执行 capric_re-

㊀ 这些 pci_dev 结构在 Linux 系统对 PCI 总线枚举时建立。在加载 Capric 卡驱动程序之前，这些 pci_dev 结构已经存在。Linux 系统对 PCI 总线的枚举过程见第 14.3 节。

move 函数。这段源代码的主要作用是将 Capric 卡驱动程序使用的“软件结构 pci_driver”与“硬件结构 pci_dev”建立联系。本文并不会深入分析 pci_register_driver 和 pci_unregister_driver 函数的实现细节，而仅介绍该函数的执行顺序。对 Linux 系统有一定经验的读者，可以从中获得必要的知识。

pci_register_driver 函数首先调用__pci_register_driver→driver_register→bus_add_driver 函数。bus_add_driver 函数进行一些必要的初始化操作后，调用 driver_attach→bus_for_each_dev 函数查找 Capric 卡的 pci_dev 结构。

在 Linux 系统中，bus_for_each_dev 函数是一个重要的函数，该函数将遍历 Capric 卡所在 PCI 总线树上的所有 pci_dev 结构，并依次判断 pci_dev 结构中的 Device ID、Vendor ID 等信息是否与 capric_ids 结构中包含的对应信息相同，如果相同则调用 capric_probe 函数。bus_for_each_dev 函数调用__driver_attach 函数实现该过程。

__driver_attach 函数调用 drv→bus→match 函数（即 pci_bus_match 函数），而 pci_bus_match 函数将继续调用 pci_match_device→pci_match_id 函数，判断 capric_ids 所包含的内容是否在当前 PCI 总线树的 pci_dev 中出现。如果出现，将 capric_drv 结构与实际的 PCI 设备进行绑定。之后继续调用 driver_probe_device→really_probe 函数。

really_probe 函数将调用 dev→bus→probe 函数（即 pci_device_probe 函数），pci_device_probe 函数将调用__pci_device_probe→pci_call_probe→local_pci_probe 函数，并最终调用 Capric 卡的 probe 函数，即 capric_probe 函数。

Capric 卡的卸载过程是加载的逆过程，其调用顺序为 pci_unregister_driver 函数、driver_unregister 函数、bus_remove_driver 函数、driver_detach 函数和__device_release_driver 函数，并最终调用 capric_remove 函数。

对于 Capric 卡，初始化与结束操作是在 capric_probe 和 capric_remove 函数中完成的。在 capric_ids 结构中使用的 id 号，是联系 Capric 卡的 pci_driver 结构和 pci_dev 结构的桥梁。在该结构中的 PCI_VENDOR_ID_XILINX 和 PCI_DEVICE_ID_EP_PIPE 即为 Capric 卡的 Vendor ID 和 Device ID，分别为 0x10EE 和 0x0007。

12.3.2 Capric 卡的初始化与关闭

Capric 卡的 probe 函数完成硬件初始化和一些 Linux 系统相关的初始化操作。当 Linux PCI 在当前 PCI 总线树中，发现 Capric 卡后，由 local_pci_probe 调用 capric_probe 函数，该函数具有两个入口参数 pci_dev 和 ids，其执行过程如源代码 12-2 ~5 所示。

源代码 12-2　Capric 卡的硬件初始化片段 1

```
static struct capric_private *adapter;
…
static int capric_probe(struct pci_dev *pci_dev,
                const struct pci_device_id *ids)
{
        int result;
        resource_size_t base_addr;
```

```
        unsigned long length;
...
        adapter = kmalloc(sizeof(struct capric_private),GFP_KERNEL);

        if(unlikely(! adapter))
          return -ENOMEM;
        adapter->pci_dev = pci_dev;
...
        result = pci_enable_device(pci_dev);

        if(unlikely(result))
            goto free_adapter;
```

首先 capric_probe 函数从 local_pci_probe 函数中获得 Capric 卡对应的 pci_dev 描述符，在 Linux 系统中每一个 PCI/PCIe 设备都与唯一的 pci_dev 描述符对应。pci_dev 描述符包含 PCIe 设备的全部信息，该结构较为简单，在 ./include/linux/pci.h 文件中定义，本节并不对该结构进行详细介绍。

这段函数首先为全局指针 adapter 分配空间，在全局指针 adapter 中记录了一些 Capric 卡需要使用的私有参数，包括 Capric 卡使用的 pci_dev。这段程序为 adapter 分配完内存空间后，将调用 pci_enable_device 函数使能 PCIe 设备。pci_enable_device 函数的主要作用是修改 Capric 卡 PCI 配置空间 Command 寄存器的 I/O Space 位和 Memory Space 位。

pci_enable_device 函数最终调用 pci_enable_resources 函数，并由 pci_enable_resources 函数扫描 Capric 卡的 BAR0～5 空间，如果这些 BAR0～5 空间用到了 I/O 或者 Memory 空间，则将 I/O Space 位和 Memory Space 位置 1。pci_enable_device 函数最后调用 pcibios_enable_irq 函数分配 PCI 设备使用的中断向量号[㊀]。此后处理器可以使用存储器或者 I/O 指令与 Capric 卡通信。pci_enable_device 函数还有一个用途是置 PCI 设备的 D－State 为 D0。

Linux PowerPC[㊁]与 Linux x86 在此处的处理基本类似。只是在 PowerPC 处理器系统中，某些 PCI 设备支持存储器写并无效周期，此时 pci_enable_device 函数还需要使能 PCI 设备的 Memory Write and Invalidate 位，同时需要填写配置空间的 Cache Line Size 寄存器[㊂]。

Linux x86 的 MSI 中断机制处理过程与 Linux PowerPC 也不尽相同。在 Linux x86 中，一个 PCIe 设备使能或者不使能 MSI 中断机制时，其 pci_dev→irq 参数并不相同。

如果其他设备驱动程序再次调用 pci_enable_device 函数使能该设备时，该函数仅增加 pci_dev 的引用计数，并不会重新使能该 PCI 设备。与此对应 pci_disable_device 函数只有在 pci_dev 的引用计数为 0 之后，才能关闭 pci_dev 结构。当两个以上的设备驱动程序操纵相同的硬件时，会出现这种情况。

㊀ 如果该 PCI 设备没有使用 MSI 或者 MSI-X 机制时，才进行这种操作。

㊁ 本书分别用 Linux PowerPC 和 Linux x86 代表基于 PowerQUICC（32 位）和 x86 处理器的 Linux 系统。

㊂ 该步骤如 pmac_pci_enable_device_hook 函数所示。

源代码 12-3　Capric 卡的硬件初始化片段 2

```
        pci_set_master(pci_dev);
//      pci_try_set_mwi(pci_dev);

        result = pci_set_dma_mask(pci_dev, DMA_MASK);

        if(unlikely(result)){
            PDEBUG("can not set dma mask ... \n");
            goto disable_pci_dev;
    }
```

pci_set_master 函数将 Capric 卡 PCI 配置空间 Command 寄存器的 Bus Master 位置 1，表示 Capric 卡可以作为 PCI 总线的主设备。Capric 卡是基于 PCIe 总线的设备，而 PCIe 总线不支持存储器写并无效操作，因此这段程序不需要使用 pci_try_set_mwi 函数设置 Command 寄存器的 Memory Write and Invalidate 位。

pci_set_dma_mask 函数设置 PCIe 设备使用的 DMA 掩码。Capric 卡对一段内存进行 DMA 操作时，需要使用这段内存在 PCI 总线域的物理地址 pci_addess，如果这段内存在存储器域的物理地址 physical_address & DMA_MASK = physical_address 时，表示 Capric 卡可以对这段内存进行 DMA 操作。

x86 处理器可以使用 pci_dma_supported 函数获得最合适的 DMA_MASK 参数；而在 PowerPC 处理器中允许 PCIe 设备访问的主存储器地址范围在 Inbound 寄存器组中定义，只有在 Inbound 寄存器窗口中映射的物理地址才能被 PCIe 设备访问，有关 Inbound 寄存器组的详细说明见第 2.2 节。在 x86 和 PowerPC 处理器中，pci_set_dma_mask 函数的实现方法不同。

源代码 12-4　Capric 卡的硬件初始化片段 3

```
    result = pci_request_regions(pci_dev,DEV_NAME);
    if(unlikely(result))
        goto disable_pci_dev;

    base_addr = pci_resource_start(pci_dev,0);

    if(unlikely(! base_addr)){
        result = -EIO;
        PDEBUG("no MMIO... \n");
        goto release_regions;
    }

    if(unlikely((length = pci_resource_len(pci_dev,0) < BAR0_BYTE_SIZE)){
        result = -EIO;
        PDEBUG("MMIO is too small ... \n");
        goto release_regions;
```

```
    }

    adapter->pci_bar0 = ioremap(base_addr, length);

    if(unlikely(! adapter->pci_bar0)){
        result = -EIO;
        PDEBUG("cannot map MMIO...\n");
        goto release_regions;
    }
```

这段源代码调用 pci_request_regions 函数使 DEV_NAME 对应的驱动程序成为 pci_dev 存储器资源的拥有者。在 Linux 系统中所有存储器映射的寄存器和 I/O 映射的寄存器都使用 ioresources 进行管理。每一组存储器空间都对应一个 resource 结构，pci_request_regions 函数经过一系列函数调用，最终调用__request_region 函数，将 Capric 卡的 BAR0 空间使用的 resource 结构，其 name 参数设置为 DEV_NAME，其 flags 参数设置为 IORESOURCE_BUSY。

因此一个 PCIe 设备的驱动程序使用 pci_request_regions 函数对 pci_dev 结构进行设置，将其使用的 flags 位设置为 IORESOURCE_BUSY 之后，其他驱动程序不能再次设置这个 flags 位。在实际应用中可能存在一个硬件设备对应两个设备驱动程序的情况，此时只有一个设备驱动程序可以使用 pci_request_regions 函数对资源进行管理。

pci_resource_start 函数从 resource 结构获得 BAR0 空间的基地址，该地址为存储器域的物理地址，而不是 PCI 总线域的物理地址，该值与直接使用 pci_read_config_word 函数读取 PCI 设备 BAR 寄存器所获得的值即便相等，也没有本质的联系，因为从 resource 结构获得的是该设备 BAR 寄存器在存储器域的物理地址，而使用 pci_read_config_word 函数获得的是 PCI 总线域的物理地址。在 Linux 驱动程序中，需要使用的是存储器域的物理地址。

本书从始至终一直强调 PCI 总线域物理地址和存储器域物理地址的区别，希望读者真正理解这两个地址的区别。在 x86 处理器中，并没有显式区分这两个物理地址的区别，这在某种程度上误导了部分系统程序员。

在这段程序的最后，使用 ioremap 函数将存储器域的物理地址映射成为 Linux 系统中的虚拟地址，之后 Capric 卡的设备驱动程序可以使用 adapter→pci_bar0 指针访问 Capric 卡中存储器映射的寄存器。

除了 ioremap 函数之外，Linux 系统还提供了 ioremap_nocache 和 ioremap_cache 函数用于存储器域虚实地址的转换。其中 ioremap_nocache 函数将存储器域的物理地址空间映射到一段“不可 Cache”的虚拟地址空间，在绝大多数体系结构中，这个函数与 ioremap 函数的实现一致，因为绝大多数外部设备都需要映射到“不可 Cache”的虚拟地址空间中。由于历史原因，99% 以上的程序员已经使用了 ioremap 函数而不是 ioremap_nocache 函数进行总线地址到虚拟地址的转换，因此 ioremap_nocache 函数显得冗余。

值得注意的是，对于 PCIe 设备的 Linux 驱动程序，ioremap 函数使用的物理地址必须从 pci_resource_start 函数获得，而不能使用“通过 pci_read_config_xxxx 函数”获得的 BARx 基地址，因为 PCIe 设备的 BARx 基地址空间属于 PCI 总线域，而不是存储器域。

某些设备还可能使用“可 Cache 的”虚拟地址空间，此时需要使用 ioremap_cache 函数

将存储器域地址转换为虚拟地址，目前为止，仅有极少数外部设备需要使用这一函数，如PCIe 设备中的 ROM 空间。

Linux 系统还提供了一个 ioremap_flags 函数，使用这个函数可以自定义存储器域地址转换到哪种类型的虚拟地址，该函数提供了一个入口参数 flags，系统程序员可以使用该参数确定所申请虚拟地址空间的类型。对于 x86 处理器，flags 参数的定义见 ./arch/x86/include/asm/pgtable.h 文件；而对于 PowerPC 处理器，flags 参数的定义见 ./arch/powerpc/include/asm/ pgtable-ppc32.h 文件。

源代码 12-5　Capric 卡的硬件初始化片段 4

```
        result = register_chrdev(test_dri_major, DEV_NAME, &capric_fops);
    …
        result = pci_enable_msi(pci_dev);

        if(unlikely(result)){
            PDEBUG("can not enable msi ... \n");
            goto chrdev_unregister;
        }

        result = request_irq(pci_dev->irq, capric_interrupt,
                        0, DEV_NAME, NULL);

        if(unlikely(result)){
            PDEBUG("request interrupt failed ... \n");
            goto err_disable_msi;
        }
    …
        capric_reset();
        return 0;
    …
    }

    static const struct file_operations capric_char_fops = {
        .owner          = THIS_MODULE,
        .ioctl          = capric_ioctl,
        .open           = capric_open,
        .release        = capric_release,
        .write          = capric_write,
        .read           = capric_read,
    };
```

这段源代码首先使用 register_chrdev 函数注册一个 char 类型的设备驱动程序，包括打开、关闭、读写操作和 ioctl 函数。之后该程序调用 pci_enable_msi 函数使能 Capric 卡的 MSI

中断请求机制，该函数将在第 12.3.5 节中详细介绍。

随后这段程序使用 request_irq 函数注册 Capric 卡使用的中断服务例程 capric_interrupt，并使用 pci_dev→irq 作为这个函数的 irq 入口参数。Capric 卡的 pci_dev→irq 参数在 Linux 系统对 PCI 总线进行初始化时分配，在 x86 处理器中，如果一个 PCIe 设备支持 MSI 中断，驱动程序执行完毕 pci_enable_msi 函数后，pci_dev→irq 参数还会发生变化。因此 request_irq 函数必须在 pci_enable_msi 函数之后运行。

这段源代码的最后将调用 capric_reset 函数，对 Capric 卡进行硬件初始化，该函数执行的操作见第 12.1.2 节。

12.3.3 Capric 卡的 DMA 读写操作

Capric 卡的 DMA 读/写过程与 capric_write/capric_read 函数对应。

1. DMA 写的操作流程

Capric 卡的数据传送方法较为简单，其 DMA 读写的硬件操作流程如第 12.1.3 节所示。DMA 写的实现过程与 capric_read 函数对应，如源代码 12-6～7 所示。

源代码 12-6　Capric 卡的 DMA 写片段 1

```
static ssize_t capric_read(struct file * file,
        char __user *buff, size_t count, loff_t *f_pos)
{
    int err = -EINVAL;
    void *virt_addr = NULL;
    dma_addr_t dma_write_addr ;
...
    virt_addr = kmalloc(count, GFP_KERNEL);

    if((unlikely(! virt_addr))){
        PDEBUG("can not alloc rx memory you want ... \n");
        return -EIO;
    }

    dma_write_addr = pci_map_single(adapter->pci_dev,
                                    virt_addr, count, PCI_DMA_FROMDEVICE);

    if((unlikely(pci_dma_mapping_error(adapter->pci_dev, dma_write_addr)))){
        PDEBUG("RX DMA MAPPING FAIL ... \n");
        goto err_kmalloc;
    }
```

这段源代码首先对 count 字段进行检查，因为 Capric 卡规定一次 DMA 操作所传递的数据不超过 2KB，之后使用 kmalloc 函数分配 DMA 写使用的数据缓存。kmalloc 函数所能分配的内存大小受限于 Linux 系统中的 SLAB/SLUB 内存分配器，在系统内存紧张时，有可能失

败，因此在此必须进行参数检查。值得注意的是，在一个实际驱动程序中，很少在读写服务例程中使用 kmalloc 函数申请内存，然后使用 kfree 函数释放这段内存，因为这样做容易产生内存碎片。而且 kmalloc 函数的执行时间也相对较长，影响数据传送的效率。

随后这段代码调用 pci_map_single 函数将存储器域的虚拟地址 virt_addr 转换为 PCI 总线域的物理地址 dma_write_addr，供 Capric 卡的 DMA 控制器使用。Linux 系统提供了一组将虚拟地址转换为设备域物理地址的方法，参见第 12.3.5 节。

源代码 12-7　Capric 卡的 DMA 写片段 2

```
#ifdef CONFIG_NOT_COHERENT_CACHE㊀
        dma_sync_single(adapter->pci_dev,
            virt_addr, count, PCI_DMA_FROMDEVICE);
#endif
        capric_w32(dma_write_addr, WR_DMA_ADR);
        capric_w32(count, WR_DMA_SIZE);
        capric_w32(MWR_START,DCSR2);

        if(unlikely(interruptible_sleep_on(adapter->dma_write_wait)))
            goto err_pci_map;
…
        if((unlikely(copy_to_user(buff, virt_addr, count))))
            goto err_pci_map;

        pci_unmap_single(adapter->pci_dev, virt_addr, count, PCI_DMA_FROMDEVICE);
        kfree(virt_addr);
        return count;
…
}
```

如果当前 DMA 写操作不与 Cache 进行一致性操作，将首先执行 dma_sync_single 函数进行存储器与 Cache 的同步操作，该函数的详细说明见第 12.3.6 节。随后这段程序使用 capric_w32 函数执行第 12.1.3 节中要求的寄存器操作，之后可以使用轮询方式，或者使用中断方式唤醒这个 DMA 写进程。当进程被唤醒后，表示 DMA 写操作已经完成，此时这段程序使用 copy_to_user 函数将数据复制到用户空间。

值得注意的是，这段代码使用了 interruptible_sleep_on 函数将当前进程休眠，而在中断处理程序中使用 wake_up_interruptible 函数将其唤醒。这是一种非常糟糕的实现方式，而且存在相当大的隐患。

interruptible_sleep_on 函数的主要工作是将当前进程放入等待队列中睡眠，目前在 Linux 系统中，该函数已经逐步被 wait_event_interruptible 函数取代，但这并不是问题的关键。在源

㊀ #ifdef 和#endif 出现在函数体中并不可取，请读者参阅 Greg Kroah-Hartman 的 Proper Linux Kernel Coding Style 掌握正确的处理方法。

代码 12-7 中，即便使用 wait_event_interruptible 函数也存在同样的问题。

因为 interruptible_sleep_on 函数的执行路径较长，很可能在当前进程还没有被该函数放入 adapter->dma_write_wait 队列时，处理器已经执行中断服务例程，打断 interruptible_sleep_on 函数，并执行 wake_up_interruptible 函数。Capric 中断服务例程的详细说明见第 12.3.4 节。

wake_up_interruptible 函数将唤醒在 adapter->dma_write_wait 队列中休眠的进程，而此时当前进程可能还没有被加入到等待队列中。当该函数执行完毕退出中断处理例程之后，处理器继续执行 capric_read 函数，并完成 interruptible_sleep_on 函数的执行，将自身加入到等待队列中睡眠。此时由于中断服务例程已经被提前执行，因此当前进程不会被 wake_up_interruptible 函数唤醒，从而造成死锁。

程序员可以使用 DCSR1 寄存器的 msk 和 pending 位解决这个死锁问题。采用这种方法时，设备驱动程序需要保证当前进程进入等待队列后，再允许 Capric 卡提交中断请求。但是这种方法将产生较长的中断延时，从而极大影响 Capric 卡的 DMA 读写效率。

程序员还可以使用 interruptible_sleep_on_timeout 或者 wait_event_interruptible_timeout 函数进行超时处理，使用该方法也可以解决上述死锁问题。

以上这两种方法都不是完美的解决方案，因为产生这种死锁的主要原因是 Capric 卡的逻辑设计并不合理。Capric 卡使用的数据传送模型较为简单，系统程序员很难基于此模型写出高效的驱动程序。

2. DMA 读的操作流程

Capric 卡 DMA 读使用的函数与 DMA 写的类似，其流程如源代码 12-8 所示。

源代码 12-8　Capric 卡的 DMA 读

```
static ssize_t capric_write(struct file *file,
        const char __user * buff, size_t count, loff_t * f_pos)
{
    int err = -EINVAL;
    void *virt_addr = NULL;
    dma_addr_t dma_write_addr;

    virt_addr = kmalloc(count, GFP_KERNEL);
...
    if((unlikely(copy_from_user(virt_addr, buff, count))))
        return err;

    dma_write_addr = pci_map_single(adapter->pci_dev,
                virt_addr, count, PCI_DMA_TODEVICE);
...
#ifdef CONFIG_NOT_COHERENT_CACHE
    dma_sync_single(adapter->pci_dev, virt_addr,
                count, PCI_DMA_TODEVICE);
```

```
#endif

        capric_w32(dma_write_addr, RD_DMA_ADR);
        capric_w32(count, RD_DMA_SIZE);
        capric_w32(MRD_START, DCSR2);

        adapter->dma_read_done = 0;

        if(unlikely(interruptible_sleep_on(adapter->dma_read_wait)))
            goto err_pci_map;
…
        pci_unmap_single(adapter->pci_dev,
                                dma_write_addr, count, PCI_DMA_TODEVICE);
        kfree(virt_addr);
        return count;
…
}
```

读者如果正确理解了上文关于 DMA 写的执行过程，DMA 读的执行过程并不难理解。Capric 卡 DMA 读的硬件操作流程如第 12.1.4 节所示。上述源代码使用 capric_w32 函数完成硬件寄存器的填写，然后可以使用轮询或者中断方式确定 DMA 读是否已经完成。在 DMA 读完成之后该例程将释放使用的内存资源后返回。与 DMA 写的操作流程类似，这段程序依然存在隐患。

12.3.4 Capric 卡的中断处理

Capric 卡一共需要处理三种中断请求，分别为 DMA 写完成、DMA 读完成和错误中断请求。Capric 卡使用了一个中断服务例程处理这些中断请求，其执行流程如源代码 12-9 所示。

源代码 12-9　Capric 卡的中断服务例程

```
static irqreturn_t capric_interrupt(int irq,void * dev)
{
        unsigned int statue;

        statue = capric_r32(INT_REG);

        if(!(statue & INT_ASSERT)){
            PDEBUG("irq_none... \n");
            return IRQ_NONE;
        }

        if(statue & INT_ASSERT_R){
```

```
            capric_w32(statue,INT_REG);
            wake_up_interruptible(&adapter->dma_read_wait);
        }
        else{
            capric_w32(statue,INT_REG);
            wake_up_interruptible(&adapter->dma_write_wait);
        }

        PDEBUG("irq handled...\n");
        return IRQ_HANDLED;
    }
```

在 capric_probe 函数中，capric_interrupt 中断服务例程被 request_irq 函数注册到 Linux 系统的 irq_desc 中断描述符表中，并与 Linux 系统的外部中断处理函数 do_IRQ 挂接，当 Capric 卡通过 MSI 中断方式提交外部中断请求后，do_IRQ 函数将最终调用 capric_interrupt 函数完成相应的中断处理。Capric 卡处理中断请求的硬件操作流程如第 12.1.5 节所示。

目前 Linux 系统对 MSI 机制的支持并不理想，pci_enable_msi 函数[⊖]仅可以获得一个 irq 号，这为中断服务例程的设计带来了一定的困难。如果 pci_enable_msi 函数可以获得多个 irq 号，那么在 capric_probe 函数中，可以使用多个中断服务程序，其中 DMA 写完成、读完成和错误处理分别使用三个中断服务例程，而不必使用 capric_r32 函数读取 INT_REG 寄存器。在 Linux 系统中，将 pci_enable_msi 函数改写为支持多个 irq 号并不困难。对于许多 PCIe 设备，这种改写是必须的，因为 RC 从 PCIe 设备中读取寄存器的代价是非常昂贵的。

12.3.5 存储器地址到 PCI 总线地址的转换

在 Linux 系统中，支持一系列 API 实现存储器地址到 PCI 总线地址的转换，这些 API 的详细定义见./Documentation/DMA-API.txt 文件。本节仅以 pci_map_single 函数为例说明这种地址转换的工作原理。pci_map_single 函数在./include/asm-generic/pci-dma-compat.h 文件中，如源代码 12-10 所示。

源代码 12-10 pci_map_single 函数

```
static inline dma_addr_t
pci_map_single(struct pci_dev *hwdev, void *ptr, size_t size, int direction)
{
    return dma_map_single(hwdev == NULL ? NULL : &hwdev->dev,
        ptr, size, (enum dma_data_direction)direction);
}
```

该函数共有 4 个输入参数，其中 hwdev 参数与 PCI 设备的 pci_dev 对应，ptr 参数对应存储器域的虚拟地址，size 字段对应数据区域的大小。而 direction 参数与数据区域的使用方法

⊖ Linux 2.6.31 内核提供的 pci_enable_msi_block 函数也仅支持一个中断向量。

对应，PCI_DMA_NONE 用于调试，较少使用；PCI_DMA_TODEVICE 表示这段数据的传递方向是从存储器到 PCI 设备；PCI_DMA_FROMDEVICE 表示这段数据的传递方向是从 PCI 设备到存储器；PCI_DMA_BIDIRECTIONAL 表示方向未知。该函数的返回值为 dma_addr，即 PCI 总线域的物理地址。

pci_map_single 函数的主要作用是通过 ptr 参数，获得与之对应的 dma_addr，即进行存储器域虚拟地址到 PCI 总线域物理地址的转换。值得注意的是存储器域物理地址与 PCI 总线域物理地址的区别。

在 Linux 系统中，使用 virt_to_phys 函数将存储器域的虚拟地址转换为存储器域的物理地址，但是通过该函数仅能获得存储器域的物理地址，因此该地址不能填写到 PCI 设备中进行 DMA 操作。值得注意的是，进行 DMA 操作的地址是由 PCI 设备使用的，而且这个地址只能是 PCI 总线域的物理地址，尽管在许多处理器中，virt_to_phys 函数和 pci_map_single 函数的返回值相同。

不同的处理器使用不同的方式实现 pci_map_single 函数。起初在 x86 处理器中，存储器域物理地址到 PCI 总线域物理地址的转换非常简单，是直接相等的关系。但是 x86 处理器为了支持虚拟化技术，使用了 VT-d/IOMMU[㊀]技术，使得该函数的实现略微复杂。

同样是基于 x86 架构，AMD 处理器使用的 IOMMU 技术与 Intel 有所区别，AMD 的 x86 处理器使用 ./arch/x86/kernel/amd_iommu. c 文件中的 map_single 函数，进行存储器域地址空间到 PCI 总线域地址空间的转换；而 Intel 的 x86 处理器使用 ./drivers/pci/intel - iommu. c 文件中的 intel_map_single 函数实现存储器地址空间到 PCI 域地址空间的转换。IOMMU 技术略微有些复杂，在第 13. 1 节中将专门描述这部分内容。

在 PowerPC 处理器中，存在一组 Inbound 寄存器，通过该组寄存器可以将 PCI 总线地址转换为 PowePC 处理器规定的存储器地址，详见第 2. 2 节。这组 Inbound 寄存器也可以看作一种 IOMMU，只是该 IOMMU 机制仅支持段式映射而不支持页式映射。

Linux PowerPC 使用 dma_direct_map_page 函数实现这个地址转换，该函数的定义详见 ./arch/powerpc/kernel/dma. c。在 Linux PowerPC 中，PCI 总线域的物理地址也与存储器域的物理地址相等。

Linux PowerPC 还需要设置 Inbound 寄存器组，这段代码在 ./arch/power/sysdev/fsl_pci. c 文件的 setup_pci_atmu 函数中，如源代码 12-11 所示。目前这段代码对 Inbound 寄存器组的 Entry 2 进行设置，允许 PCIe 设备访问 0 ~ 0x7FFF - FFFF（2GB）这段存储器域物理地址空间，而且 PCI 总线地址与存储器地址一一对应而且相等。

源代码 12-11　setup_pci_atmu 函数

```
static void __init setup_pci_atmu(struct pci_controller * hose,
                struct resource * rsrc)
{
...
    /* Setup 2G inbound Memory Window @ 1 */
```

㊀ VT-d 是指 Intel 的 Virtualization Technology for Directed I/O 技术，而 AMD 将这一技术称为 IOMMU。下文将这些技术都简称为 IOMMU。

```
        out_be32(&pci->piw[2].pitar, 0x00000000);
        out_be32(&pci->piw[2].piwbar,0x00000000);
        out_be32(&pci->piw[2].piwar, PIWAR_2G);
    ...
    }
```

这段代码源于 Linux 2.6.30，在这个版本中，PCIe 设备不能访问 PowerPC 处理器 2GB 之上的物理内存。而在 Linux 2.6.31.6 中，该函数被大规模修改，以支持超过 2GB 的存储器系统，本节对 Linux 内核的这些改动不做进一步描述，对此有兴趣的读者可以参考 Linux 2.6.31.6 内核中 setup_pci_atmu 函数的最新实现。

有些支持 IOMMU 机制的 PowerPC 处理器，如 IBM 的 PowerPC 处理器系列，可以使用 dma_iommu_map_page 或者 ibmebus_map_page 函数实现 pci_map_single 函数，而 cell 处理器使用 dma_fixed_map_page 函数实现该功能。pci_map_single 函数函数在 IBM 的 PowerPC 处理器上已经移植完毕，但是 Freescale 除了 P4080 处理器之外，还没有支持 IOMMU 的处理器。目前对 P4080 处理器的支持并没有加入到 Linux PowerPC 中。

12.3.6 存储器与 Cache 的同步

Linux 系统还提供了一组 sync 函数，如 dma_sync_single、dma_sync_sg 等函数，这组 sync 函数的主要作用是为了支持“不进行 Cache 共享一致性”的 DMA 操作。

如果设备进行 DMA 操作时，不需要硬件进行 Cache 一致性操作[㊀]，那么处理器在 DMA 操作之前，需要使用软件指令将操作的数据区域与 Cache 进行同步，之后进行 DMA 操作。PCIe 设备启动 DMA 请求时，如果其 TLP 头部 Attr 字段的 No Snoop Attribute 位为 1[㊁]时，驱动程序也需要进行这种同步操作。在 PowerPC 处理器中，有一些非 PCIe 设备，如 QE（QUICC Engine）中的一些内嵌设备，这些设备可以通过设置 snoop 位决定在 DMA 传送过程中，是否需要硬件进行 Cache 一致性操作。

目前多数 RC 或者 HOST 主桥都可以通过总线监听，解决 PCI 设备进行 DMA 操作的 Cache 一致性问题。但是有些 RC，如 MPC8572 处理器的可以通过设置 Inbound 寄存器决定当前访问是否支持 Cache 一致性操作。

如果硬件不支持 Cache 共享一致性，那么 PCI 设备进行 DMA 操作时，必须使用软件指令维护存储器与 Cache 的同步，从而避免 Cache 与主存储器不一致的现象发生。

系统软件程序员使用软件指令维护 Cache 时，务必深入理解这些指令的特点和使用方法。在处理器系统的设计中，有两类错误最难被发现，一类是 Cache 与存储器系统的不一致，另一类是数据传送的序。

即使是对资深的系统程序员，也很难从这些错误表现形式中，发现是 Cache 不一致或者数据传送的序引发的系统错误。因此系统程序员需要重视在一个处理器系统中的 Cache 一致性（Cache Cohenrency）和数据完成性（Data Consistency）。

㊀ 有些处理器不支持硬件的 Cache 共享一致性，如一些低端的 ARM 处理器。

㊁ 目前大多数 PCIe 设备进行 DMA 操作时，No Snoop Attribute 位都为 0。

因此虽然对于多数 PCI 设备，Cache 一致性可以由硬件保证，本节也必须讲述如何通过软件指令维护 Cache 的一致性。Linux 系统使用 dma_sync_single 函数维护 Cache 的一致性。dma_sync_single 函数的实现如源代码 12-12 所示。

源代码 12-12　__dma_sync 函数

```
#define dma_sync_single dma_sync_single_for_cpu

static inline void
dma_sync_single_for_cpu(struct device *hwdev, dma_addr_t dma_handle,
                        size_t size, enum dma_data_direction dir)
{
        struct dma_map_ops *ops = get_dma_ops(hwdev);

        BUG_ON(!valid_dma_direction(dir));

        if (ops->sync_single_for_cpu)
                ops->sync_single_for_cpu(hwdev, dma_handle, size, dir);
        debug_dma_sync_single_for_cpu(hwdev, dma_handle, size, dir);
        flush_write_buffers();
}
```

不同的处理器系统使用不同的 ops->sync_single_for_cpu 操作函数。值得注意的是，Linux x86 并没有实现 ops->sync_single_for_cpu 函数，因为使用软件指令维护 Cache 一致性的情况在 x86 处理器系统中并不多见。而 Linux PowerPC 使用 dma_direct_sync_single_range 函数实现 ops->sync_single_for_cpus 函数，该函数最终将调用__dma_sync 函数。这两个函数的实现如源代码 12-13 所示。

源代码 12-13　__dma_sync 函数

```
static inline void dma_direct_sync_single_range(struct device *dev,
                dma_addr_t dma_handle, unsigned long offset, size_t size,
                enum dma_data_direction direction)
{
        __dma_sync(bus_to_virt(dma_handle + offset), size, direction);
}
…

/*
 * make an area consistent.
 */
void __dma_sync(void *vaddr, size_t size, int direction)
{
        unsigned long start = (unsigned long)vaddr;
        unsigned long end = start + size;
```

```
        switch (direction) {
        case DMA_NONE:
            BUG();
        case DMA_FROM_DEVICE:
            /*
             * invalidate only when cache-line aligned otherwise there is
             * the potential for discarding uncommitted data from the cache
             */
            if ((start & (L1_CACHE_BYTES - 1)) || (size & (L1_CACHE_BYTES - 1)))
                flush_dcache_range(start, end);
            else
                invalidate_dcache_range(start, end);
            break;
        case DMA_TO_DEVICE: /* writeback only */
            clean_dcache_range(start, end);
            break;
        case DMA_BIDIRECTIONAL: /* writeback and invalidate */
            flush_dcache_range(start, end);
            break;
        }
    }
    EXPORT_SYMBOL(__dma_sync);
```

在 Linux PowerPC[⊖]中，flush_dcache_range、invalidate_dcache_range 和 clean_dcache_range 函数分别使用 dcbf、dcbi 和 dcbst 指令实现。dcbi/dcbf/dcbst 指令的格式为 dcbi/dcbf/dcbst rA，rB，如源代码 12-14 所示。

源代码 12-14　dcbi/dcbf/dcbst 指令格式

```
if rA =0 then a <- 640 else a <- rA
EA <- 320 || (a + rB)32:63
InvalidateDataCacheBlock(EA)          // dcbi
FlushDataCacheBlock                   // dcbf
StoreDataCacheBlock(EA)               // dcbst
```

dcbf、dcbi 和 dcbst 指令的详细说明如下。

- dcbi 指令首先在 Cache 中检查 EA。如果 EA 在 Cache 中命中，将直接将 EA 所对应的 Cache 行的状态改变为 I，无论这个 Cache 行原来的状态是什么，都不将数据回写到存储器中。
- dcbf 指令首先在 Cache 中检查 EA。如果 EA 在 Cache 中命中，则继续检查 Cache 的状态，如果为 M，则将 Cache 行刷新到内存，然后 Invalidate 该 Cache 行，将 Cache 行的

⊖ 以 E500 V2 内核为例。

状态改变为 I；否则直接 Invalidate 该 Cache 行。

- dcbst 指令首先在 Cache 中检查 EA。如果地址在 Cache 中命中，则继续检查 Cache 的状态，如果为 M，则将 Cache 行回写到内存，然后将 Cache 行的状态改变为 E；否则不做任何操作。

在 x86 处理器系统中，也存在类似的 Cache 指令。如 INVD、WBINVD 和 CLFLUSH 指令。其中 INVD 指令的作用是 Invalidate 处理器中的内部 Cache，并通过 FSB 总线周期 Invalidate 外部 Cache；而 WBINVD 指令是在 Invalidate 内部和外部 Cache 之前，先将 Cache 中的数据回写，然后进行 Invalidate 操作。

但是这两条指令都是针对整个 Cache，而不是针对某个 Cache 行。如果需要对 Cache 行进行刷新时，x86 处理器可以使用 CLFUSH 指令操作某个 Cache 行，该指令所实现的功能与 dcbf 指令类似。单从操作 Cache 行的指令的角度上看，PowerPC 处理器比 x86 处理器好得多，因此本节以 PowerPC 处理器为例说明这些 Cache 指令在不同情况下的使用方法。

下文将分别介绍在 DMA 写和 DMA 读操作中__dma_sync 函数的工作流程。假设在一个单处理器系统[⊖]中，Cache 行长度为 512b，而且 PCI 设备进行 DMA 读写操作时，硬件不进行 Cache 一致性操作。

1. DMA 写

在外部设备进行 DMA 写操作之前，需要使用__dma_sync 函数同步 Cache 与存储器中的数据。有许多书籍包括 Linux 系统中的 DMA - API 文档，都认为在 DMA 写操作完成后，调用__dma_sync 函数 Invalidate 数据区域所对应的 Cache 行，处理器就可以使用来自设备的数据。这种说法是基于设备访问的数据区域头尾都是 Cache 行对界的情况而言的，如果数据区域并不是 Cache 行对界时，这种做法将引发系统错误。

假设在一个处理器系统中，Cache 行长度为 64B。当一个 PCI 设备通过 DMA 写操作，访问 0x1001 ~ 10FE 这段数据区域时，这段数据区域将占用 4 个 Cache 行，而且并不是 Cache 行对界的，如图 12-9 所示。

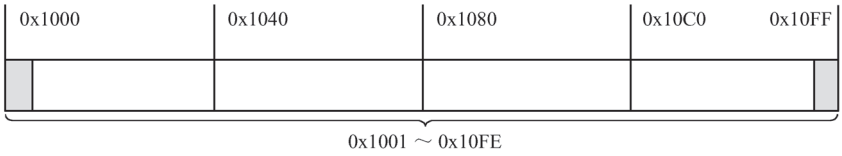

图 12-9　DMA 写访问的数据区域不对界

如果在 0x1000 ~ 0x10FF 这段数据区域中，0x1000 和 0x10FF 字节曾经被改写过，那么 0x1000 ~ 103F 和 0x10C0 ~ 10FF 这两个 Cache 行的状态为 M，因此图 12-9 中阴影部分的数据与存储器不一致，而且为处理器系统中最新的数据。

而 DMA 写结束后，0x1001 ~ 10FE 这段数据区域被 PCI 设备改写，且为处理器系统中最新的数据，此时这段数据区域对应的 Cache 行状态仍然为 M。此时如果处理器 Invalidate 0x1001 ~ 10FE 这段数据区域对应的 Cache 行，即 Invalidate 0x1000 ~ 0x10FF 这段数据区域的 Cache 行时，将会丢失 0x1000 和 0x10FF 这两个字节中保存的合法数据。如果处理器刷新

⊖ 在 SMP 系统中，Cache 行的状态转换更为复杂。

0x1001 ~ 10FE 这段数据区域对应的 Cache 行，即刷新 0x1000 ~ 0x10FF 这段数据区域的 Cache 行时，将丢失所有来自 PCI 设备的数据，采用这种方法问题更大。

通过上述分析可以发现，如果在 DMA 写完成后，再对访问的数据区域进行 Cache 同步操作，将可能引发严重的 Cache 一致性问题，从而导致整个系统异常。

为此正确的方法是在 DMA 写操作之前，将其访问的数据区域与 Cache 进行一致性操作，如源代码 12-13 中“case DMA_FROM_DEVICE”所示。但是这段源代码仍然有较大的问题，因为这段代码在处理数据区域不对界的情况时，将刷新整个数据区域对应的 Cache 行。

这种做法不会产生错误，但是会影响效率。假设与 0x1000 ~ 0x10FF 这段数据区域对应的 Cache 行的状态都为 M，那么这段程序将 0x1000 ~ 103F、0x1040 ~ 0x107F、0x1080 ~ 0x10BF 和 0x10C0 ~ 10FF 对应的 Cache 行都刷新到存储器中，然后再 Invalidate 这些 Cache 行。而实际上 0x1040 ~ 0x10C0 这段数据区域将由 PCI 设备重新填写，因而将这部分区域进行刷新然后再 Invalidate 是没有意义的。

正确的做法是刷新不对界的数据区域 0x1000 ~ 1040 和 0x10C0 ~ 10FF，即将这段数据区域的头尾刷新即可，而直接 Invalidate 中间的数据区域 0x1040 ~ 10BF。采用这种方法将有效地提高系统效率。

2. DMA 读

外部设备进行 DMA 读操作之前，处理器必须保证该设备访问的数据区域与 Cache 一致。因此需要调用__dma_sync 函数（dir 参数为 DMA_TO_DEVICE）将 Cache 中的数据回写到存储器。该函数执行完毕后，Cache 中的所有数据都与存储器一致，而之前状态位为 M 的 Cache 行将更改为 E。经过这个 Cache 同步操作后，设备进行 DMA 读操作就可以从存储器中获得正确的数据。

此处还有一个细节问题值得考虑，就是 DMA 读操作是调用 dcbst 指令回写 Cache 行，还是调用 dcbf 指令刷新 Cache 行。如上文所述 dcbf 和 dcbst 指令都将状态位为 M 的 Cache 行与存储器进行同步，只是 dcbf 将 Cache 行的 M 位更新为 I，而 dcbst 指令将 Cache 行的状态更新为 E。使用这两个指令都可以保证 DMA 读的数据不会出现一致性问题。

此时在处理器系统中，如果 DMA 读的数据将不会被处理器使用时，应该使用 dcbf 指令，Invalidate 该 Cache 行，从而该 Cache 行可以被其他进程使用；如果 DMA 读的数据将很快被处理器使用时，应该使用 dcbst 指令，回写该 Cache 行，从而处理器使用该数据时，可以从 Cache 而不是存储器中获得。

12.4 Capric 卡的延时与带宽

总线的延时与带宽是一个巨大的话题，即便本节将其局限到 PCIe 总线，局限到 Capric 卡，也并不能改变这个话题的沉重。总线的带宽和延时之间有一定的制约关系，任何一个处理器系统都希望在获得巨大总线带宽的同时，尽量缩小访问延时。

在处理器系统的设计中，必须权衡“提高带宽”与“缩小延时”之间的关系，以组建一个合理的应用系统。在处理器领域，带宽是指数据的传送速率，即 1 秒钟传送的数据大小。而延时指传送处理一个数据单元所需要的时间。

片面地追求带宽显然并不合理，过大的带宽对某些应用并不合适。我们无法忽视一个满

载光盘的火车所能提供的数据带宽，这辆火车至少能传递几十万 Pebibyte 大小的数据，而且十个小时左右就能从北京抵达上海。显然多辆火车所能提供的带宽非常巨大，但是我们仍然无法使用火车传送网络报文，因为十个小时的延时是许多系统应用无法容忍的。

延时与带宽之间存在某种联系，而不是直接对立关系，并不是带宽越大延时也越大，带宽越小延时也越小。处理器系统设计所追求的目标是提高带宽的前提下，尽可能掩盖传送延时，组成一个可实现的处理器系统。

PCIe 总线作为处理器的通用局部总线，需要权衡带宽和延时之间的关系，满足在处理器系统中多数应用的需求，而非包罗万象。PCIe 总线能够满足，绝大多数处理器系统，特别是 PC 系统中的显卡、硬盘，声卡和一些慢速设备与处理器之间的数据传送，但是对于一些延时要求较高的应用并不适用。

当一个网络设备需要以 1Gb/10Gb 的速率“线速传送”64B 大小的网络数据报文时，PCIe 总线显得力不从心，通常这种网络设备需要直接与处理器的 FSB 相连，以尽量缩短传送延时。在许多处理器系统中，如 Freescale 的 P4080 处理器，RMI 的 XLP832、Cavium 的 CN6335 处理器，网络设备与 FSB 直接相连，并在追求最大带宽的同时，尽量减少访问延时。

在 x86 处理器系统中，可以使用 QPI（QuickPath Interconnect）连接这样的高性能网卡。但是目前在 PC 领域中，高性能网卡依然使用 PCIe 总线进行连接。

在一个处理器系统中，为掩盖传送延时，通常使用“流水线”技术，当数据传送的延时增加时，“流水线”所使用的资源也随之增加，并很容易到达处理器系统所不能忍受的范围。本章讲述在 PCIe 总线中，延时与带宽间的关系，以及存在的问题。PCIe 总线基于 TLP 进行数据传递，因此本节所强调的带宽与延时与 TLP 直接相关。

在本节中，PCIe 总线的带宽指每秒钟传送的“TLP 中有效数据“的大小，即 PCIe 总线的有效带宽。该定义与 PCIe 总线的链路带宽不同。如 PCIe V2.1 总线规范链路带宽为 5GT/s，而这个带宽需要去掉物理层 8/10b 转换，以及 PCIe 总线的协议开销后才能得到 PCIe 总线的有效带宽。PCIe 总线的有效带宽与许多因素相关，包括协议开销、TLP Payload 的大小、传送延时、流量控制等因素相关，当然最重要的因素依然是 PCIe 总线的链路宽度。

PCIe 总线的延时指一个存储器请求从产生到结束的时间。值得注意的是存储器读与存储器写 TLP 的延时计算有所不同，存储器写 TLP 产生于发送端而结束于接收端，仅计算单向延时；而存储器读请求 TLP 产生于发送端，接收端将存储器读请求 TLP 转换为存储器读完成 TLP，再发送给发送端，需要计算双向延时。

12.4.1 TLP 的传送开销

在 PCIe 总线中，TLP 产生于事务层，并在通过 PCIe 总线的链路层与物理层时，加入若干前缀和后缀后，才能经由 PCIe 端口发送。PCIe 总线定义了多种 TLP，本章重点关心与存储器读写相关的 TLP，包括存储器写 TLP、存储器读请求 TLP 和存储器读完成 TLP。这些 TLP 报文的通用格式如图 12-10 所示。

1 Byte	2 Byte	3 ～ 4 DW	0 ～ 1024 DW	1 DW	1 DW	1 Byte
Start	Sequence ID	TLP Head	Data Payload	ECRC	LCRC	End

图 12-10　PCIe 总线 TLP 格式

由上图所示，一个 PCIe 设备发送 TLP 报文时，在经过数据链路层和物理层时，需要加上若干前缀和后缀。

（1）Start 和 End 前后缀由物理层添加，表示一个 TLP 的开始与结束，各由一个字节组成，用于物理层同步 TLP 发送与接收。

（2）Sequence ID 和 LCRC 前后缀由数据链路层添加，存放 TLP 的识别号和数据链路层的 CRC 校验，分别由 2 B 和 1DW 组成。

（3）TLP Head 前缀描述 TLP 的属性，由 3 ~ 4 个 DW 组成。

（4）ECRC 后缀存放 TLP 在事务层的 CRC 校验，该字段可选。

（5）而 Data Payload 是真正的有效负载，其长度在 0 ~ 1024DW 之间。由上文所述，TLP 的有效负载长度由 PCIe 设备的 Max_Payload_Size 参数确定，该字段通过上下游链路进行协商后获得。目前在多数处理器系统中，PCIe 设备使用的 Max_Payload_Size 参数为 128B 或者 256 B。

我们假设 TLP 使用 3DW 的报文头，而且不需要 ECRC 校验，根据图 12-10，在一个 TLP 中，Data Payload 所占的比例如公式 12-11 所示。

$$Payload\_Ratio = Payload/(Payload + 20) \tag{12-11}$$

假设 PCIe 设备的 Max_Payload_Size 参数为 256 B 时，根据以上公式可以得出该 PCIe 设备最大的 Payload_Ratio = 256/(256 + 20) ≈ 92.8%。但是 TLP 在 PCIe 链路中进行传送时，远不能获得 Payload_Ratio 大小的链路带宽。TLP 在传送过程中，需要通过 PCIe 总线的事务层、数据链路层和物理层，因此必须考虑这些协议所带来的开销。

1. 事务层的开销

事务层的开销需要分存储器写和存储器读两种情况讨论。在 PCIe 总线中，存储器写请求 TLP 使用 Posted 总线事务，而存储器读请求 TLP 使用 Split 总线事务。在这两种情况之下，事务层的开销不同。

PCIe 设备进行 DMA 写的过程较为简单，当 PCIe 设备将一个 4KB 大小的数据传送到存储器时，首先将 4KB 数据封装到多个存储器写请求 TLP 中，然后发向 RC，其中每个 TLP 最大的 Payload 为 Max_Payload_Size。

PCIe 设备的 Max_Payload_Size 参数由 PCIe 链路协商确定，目前 x86 处理器系统 RC 的 Max_Payload_Size 为 128 B 或者 256 B，所以与 RC 直接相连的 PCIe 设备，其 Max_Payload_Size 参数只能为 128 B 或者 256 B。由公式 12-11，可以计算出与 Max_Payload_Size 参数对应的 Payload_Ratio，并由此推算存储器写 TLP 在事务层上的开销。

PCIe 设备进行 DMA 读的过程略微复杂。首先 PCIe 设备向 RC 发送存储器读请求 TLP，当 RC 收到这个存储器读请求 TLP 后，将从存储器中获得数据，然后组成一个或者多个存储器读完成 TLP，并将其传送给 PCIe 设备。

存储器读完成 TLP 能请求的数据大小为 MRRS（Max_Read_Request_Size），该参数的大小为 128 B ~ 4096 B。如果 PCIe 设备进行 DMA 读的大小超过该参数时，将以该参数为界，向 RC 发出多个存储器读请求 TLP。而 RC 可以使用一个存储器读完成报文传递所有数据；也可以以 RCB 为边界，使用多个存储器读完成报文传递所有数据。大多数 RC 使用后一种方式传递存储器读完成 TLP，而且一次存储器读完成报文的大小也不超过 RCB。本章以这种方式为例分析存储器读 TLP 在事务层中的开销。

假设存储器读请求 TLP 头的大小为 3DW，而且报文头中不包含 ECRC 校验。PCIe 设备进行 DMA 读的大小恰好为 MRRS 时，RC 需要使用 MRRS/RCB 个存储器读完成报文传递数据。此时一次 DMA 读操作中 Data Payload 所占的比例如公式 12-12[⊖]所示。

$$Payload\_Ratio = MRRS/(MRRS + 3 \times 4 + 3 \times 4 \times MRRS/RCB) \quad (12-12)$$

当 MRRS 为 512B，而 RCB 为 64B 时，Payload_Ratio = 512/(512 + 12 + 12 * 8) ≈ 82.6%。由以上分析可以发现 PCIe 设备进行 DMA 读在事务层上的开销大于 DMA 写在事务层上的开销，这也是 PCIe 设备 DMA 写的速度略高于 DMA 读的主要原因。

除了事务层的开销之外，DMA 读操作的数据传送路径也长于 DMA 写，因而访问延时大于 DMA 写操作的访问延时，这也为 DMA 读逻辑的设计带来了不小的麻烦。从第 12.2.2 节中，也可以发现 DMA 读逻辑的设计远比 DMA 写逻辑的设计复杂。

2. 链路层的开销

由图 12-10 所示，链路层向 TLP 添加 Sequence ID 和 LCRC 前后缀，这些开销已经在上文中计算，本小节不再重复计算这些开销。本小节所关心的链路层开销由两部分组成，一个是 ACK/NAK 协议的开销，另一个是流量控制所带来的开销。如第 7.2 节所示，发送方在发送 TLP 时，首先将这些 TLP 放入到 Replay Buffer 中，直到收到接收方的 ACK 报文后，才能确认该 TLP 已经正确地被接收方接收；如果收到接收方 NAK 报文，则表示部分 TLP 没有被正确接收，需要重新发送这些 TLP。这些 ACK/NAK 报文将占用部分链路带宽。

在 PCIe 设备的实现过程中，设计者可以调整 TLP 接收个数的阈值，这个阈值的定义为接收端收到多少 TLP 后，给发送端提供一次 ACK/NAK DLLP。该阈值决定了数据接收端发送 ACK/NAK 报文的间隔。

该阈值越大，则发送端的 Replay Buffer 也将随之增大，否则发送端无法将数据及时填入 Replay Buffer，从而阻塞了发送流水，并影响 PCIe 总线的传送效率；如果该阈值越小，则接收端需要发送较多的 ACK/NAK 报文给接收端，也会影响 PCIe 总线的效率。因此接收端需要合理地设置 ACK/NAK 的阈值，以最大程度地利用 PCIe 总线的带宽。

在链路层的设计中，需要选择合适的 Replay Buffer 的大小。如果 Replay Buffer 过小，事务层无法及时地将 TLP 发送到 Replay Buffer，从而造成 TLP 发送流水线的中断。而保存在 Replay Buffer 中的报文需要得到对端设备的确认报文后，才能释放。

因此可以发现 Replay Buffer 的大小，只需要保证事务层发送 TLP 时，有足够的缓冲即可。Replay Buffer 的大小与 PCIe 链路的延时相关。在实现中，Replay Buffer 不能过大，否则将使用较多的芯片资源。

在链路层中，除了 ACK/NAK 报文的开销外，流量控制报文也需要占用 PCIe 总线的链路开销。PCIe 总线使用 Credit - Based 流量控制策略，发送端需要保证接收端有足够的缓冲之后才能发送报文，而且接收端需要按照某种策略及时使用 FC Update 报文，向发送端通知剩余的数据缓冲。因此流量控制也需要占用一些 PCIe 总线的带宽。在 PCIe 总线中，流量控制是基于“端到端”的，而且 PCIe 总线并没有规定 PCIe 设备使用的流量算法，因此流量控制对 PCIe 总线带宽的影响与设备相关，并没有一个统一的公式。

⊖ 该公式没有计算物理层和数据链路层报文头的开销。

3. 物理层的开销

在 PCIe V2.1 总线规范中，TLP 在物理层中还需要进行 8/10b 转换，这个转换将极大地降低 PCIe 总线的实际链路带宽，而且在 PCIe 总线中，这种带宽的浪费是无法避免的。在 PCIe V3.0 规范中，这个 8/10b 转换被升级为 128/130 转换。128/130 转换将极大节约 PCIe 链路带宽的浪费。但是无论如何，TLP 在发送过程中，仍然会因为这种转换浪费 PCIe 链路的一些带宽。

除了 8/10b 转换之外，物理层为了解决接收时钟与逻辑时钟间的漂移所带来的问题，每一个 Lane 需要在发送 1180～1538 个字符后，发送一个 SKIP 序列进行时钟补偿。这种定时的时钟补偿序列也将浪费 PCIe 链路的部分带宽。

12.4.2 PCIe 设备的 DMA 读写延时

上节简要介绍了影响 PCIe 设备进行数据传递的因素，无论设计者采用什么样的设计方式，TLP 在通过事务层、链路层和物理层时都会受到这些因素的影响。但是不同的设计方法依然会极大影响 PCIe 总线的使用效率。

本文中出现的 Capric 卡是一个很糟糕的设计，在这个设计中，并没有使用流水线机制来掩盖 PCIe 总线的延时，因此该卡通过 PCIe 总线进行 DMA 读时的效率并不高。

1. Capric 卡 DMA 写的效率

在 Capric 卡中，DMA 写操作由多个步骤组成，并由 Capric 卡的硬件逻辑和处理器的中断处理机制协调完成，其步骤如图 12-11 所示。

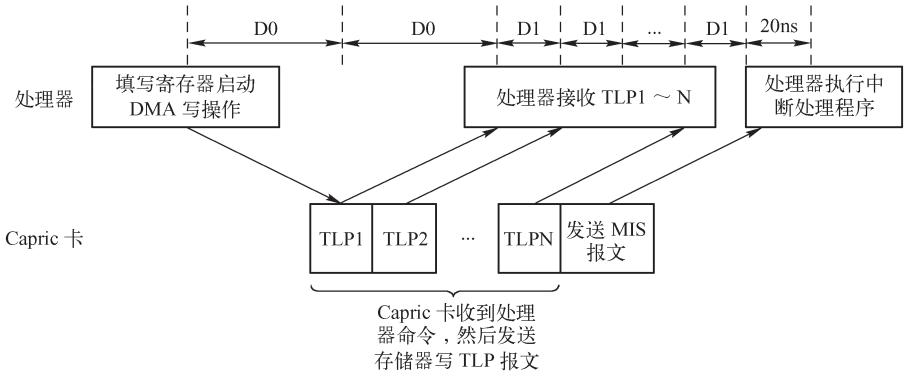

图 12-11 Capric 卡的 DMA 写过程

首先处理器填写 Capric 卡的 WR_DMA_ADR、WR_DMA_SIZE 和 DCSR2 寄存器，经过延时 D0 之后，这些命令陆续到达 Capric 卡。其中 D0 的大小与 Capric 卡连接在处理器系统中的位置相关，而与 Capric 卡的设计无关。如果 Capric 卡直接与 RC 相连接，则 D0 的值较小；如果 Capric 卡通过多级 Switch 之后再与 RC 连接，则 D0 较大。

Capric 卡收到处理器的 DMA 写请求后，将向 RC 连续发送存储器写 TLP，并由 RC 将数据写入到主存储器。Capric 卡根据 WR_DMA_SIZE 寄存器的数值，将数据分解为多个存储器写请求 TLP，其中每个存储器写请求 TLP 的有效负载不超过 Max_Payload_Size 参数，本节假设该参数的大小为 128 B。

假定 Capric 卡使用 100 MHz[㊀]的时钟，而内部总线宽度为 64 位，此时 Capric 卡内部总线的带宽可以达到 800 MB/s，该值非常接近 ×4 PCIe 链路所能提供的有效带宽；存储器写 TLP 使用 3DW 的报文头且不使用 ECRC 校验，而每个存储器写 TLP 的最大有效负载为 128B（32DW）。因此在 Capric 卡中，一个存储器写 TLP 的最大长度为 35DW，此时 Capric 卡需要使用 18（实际值为 17.5）个时钟周期才能将这个 TLP 发送出去，因此在 Capric 卡中 D1 的大小为 180 ns。

Capric 卡将存储器写 TLP 发送完毕后，将向 RC 发送 MSI 报文，MSI 报文也是一种存储器写 TLP，Capric 卡使用两个时钟周期，即 20ns 即可将该报文发送出去。RC 在等待一段延时后，将陆续收到存储器写请求 TLP1 ~ N 和 MSI 报文，这段延时为 TLP 从 EP 到 RC 的延时，约等于 D0。

处理器收到 MSI 报文后，将执行中断处理程序，Capric 卡的中断处理例程通过 RC 读取中断控制状态寄存器 INT_REG，并结束整个 DMA 写操作。HOST 处理器读取中断控制状态寄存器的开销绝对不能忽略，因为这个读取过程首先是 RC 发送存储器读请求 TLP，当 Capric 卡收到这个 TLP 后再向 RC 发送存储器读完成 TLP，其访问延时为 2 × D0。

假设 Capric 卡一次 DMA 写的大小为 X 字节[㊁]，则这次 DMA 写所需的时间 T_{dmaw} 如公式 12-13 所示。

$$T_{dmaw} = D0 + D0 + X/128 \times D1 + 20 + 2 \times D0 \qquad (12-13)$$

其中 T_{dmaw} 的值越小，Capric 卡传送 X 字节的数据所需的时间也越短。但是以上公式并没有考虑 Capric 在 DMA 写过程中，因为数据缓冲不足而暂时中断存储器写 TLP 发送流水线的情况，而且忽略了中断处理例程所需的切换与执行时间。

由以上公式，可以发现当 D0 越大 T_{dmaw} 也越大。但是当 X 越大时，D0 在 T_{dmaw} 中所占的比重越小，在一个处理器系统中，D0 的大小是 Capric 卡无法控制的，该值的大小与 Capric 卡在处理器系统的位置和处理器系统的 RC 确定，属于系统延时，我们假设该值为 250 ns[㊂]。根据以上假设，可以利用公式 12-13 获得 Capric 卡 DMA 写的有效带宽，如表 12-3 所示。

表 12-3　X 的大小与 DMA 写有效带宽的关系

Tdmaw（ns）	X	Capric 卡的最大有效带宽
1200	128 B	106.7 MB/s
1380	256 B	185.5 MB/s
1740	512 B	294.3 MB/s
2460	1 KB	416.3 MB/s
3900	2 KB	525.1 MB/s
6780	4 KB	604.1 MB/s
12540	8 KB	653.2 MB/s
24060	16 KB	681.0 MB/s

㊀ 实际上 LogiCORE 只能使用 100 MHz 或者 250 MHz 的内部时钟，但是笔者没有无法在 FPGA 内部运行 250 MHz 的时钟频率。

㊁ X 可以被 128 整除。

㊂ 在 Tylersburg EP 平台中，这个系统延时大于 250ns。

由表 12-3 所示，X 越大，Capric 卡的有效带宽也越高，因此适当提高 X 的大小将有效提高 Capric 卡 DMA 写的传送效率。

2. Capric 卡 DMA 读的效率

在 Capric 卡中，DMA 读的过程比 DMA 写过程略微复杂一些。因为 DMA 读是由两部分组成的，首先 Capric 卡向 RC 发起存储器读请求 TLP；当 RC 从存储器获得数据后，再向 Capric 卡发送存储器读完成 TLP，其实现过程如图 12-12 所示。

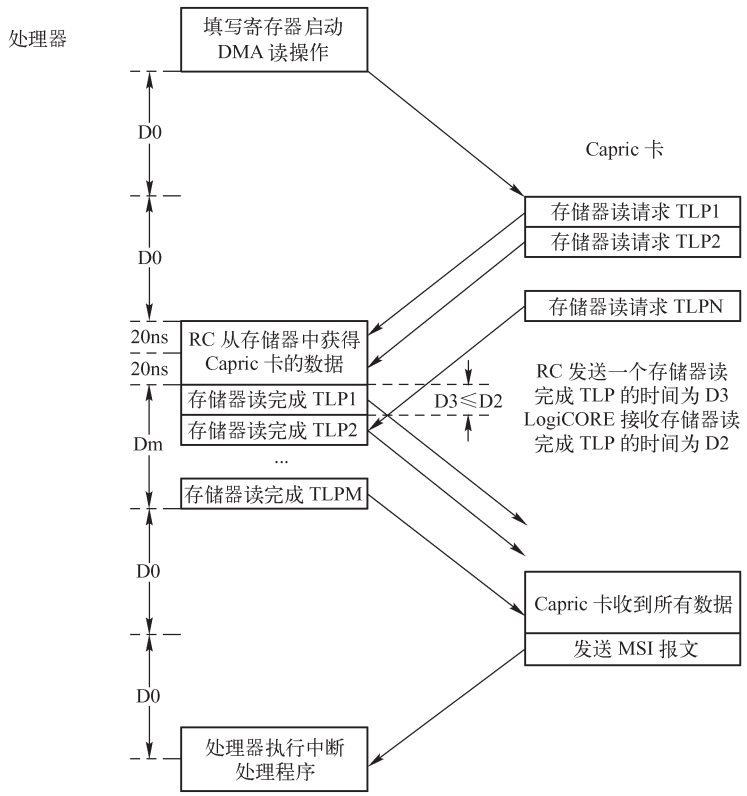

图 12-12　Capric 卡的 DMA 读过程

如上图所示，Capric 卡的 DMA 读过程如下所示。

（1）首先处理器填写 Capric 卡的寄存器启动 DMA 读。

（2）Capric 卡在等待 D0 这段延时后，收到这个命令，之后向 RC 提交存储器读请求 TLP。假设 Capric 卡一共需要 N 个存储器读请求 TLP 才能发送完成整个请求，而且需要 Dn 这段时间才能将完成这些操作。而且存储器读请求 TLP 到达 RC 的延时也为 D0。

（3）处理器在等待 D0 + D0 这段延时后，将开始接收存储器读请求 TLP，并从主存储器获得相应的数据，并组织存储器读完成报文发向 Capric 卡。RC 从开始接收存储器读请求 TLP 到接收最后一个存储器读请求 TLP 的时间延时为 Dn。其中 Dn 也与 Capric 卡发送全部存储器读请求的延时相同。

（4）RC 收到来自 Capric 卡的存储器读请求 TLP 后，开始发送存储器读完成报文，其中 Capric 卡接收存储器读请求 TLP 与 Capric 卡发送存储器读完成 TLP 可以同步进行，因此 Dn 这段延时并不会被重复计算，而仅计算发送存储器读完成报文的流水准备时间，这段时间由

两部分组成，Capric 卡发送一个存储器读请求 TLP 的延时，和 RC 从主存储器中读取数据的延时。我们假定这两段延时都为 20 ns，因此流水准备时间为 40 ns。

（5）Capric 卡经过 D0 这段延时后开始接收存储器读完成 TLP，并经过 Dm 这段延时后将接收完毕所有来自 RC 的存储器读，并获得 DMA 读的所有数据。

（6）Capric 卡发送 MSI 报文通知处理器 DMA 读完成。

（7）处理器经过 D0 这段延时后，收到 MSI 报文，并开始执行中断处理程序。如果这个中断处理程序仍然需要读取中断控制状态寄存器，则处理器需要 2 × D0 这段延时时间之后才能完成 Capric 卡的 DMA 读。

其中 D3 延时为 RC 发送存储器读完成 TLP 所需要的时间，D3 延时与 RC 的 RCB 参数相关。我们假设 RC 的 RCB 参数为 64 B，存储器读完成中的 Payload 为 64 B，而读完成报文头为 12 B；同时假设 RC 使用 128b 的数据总线，而且其总线频率为 667 MHz，此时 RC 发送的数据报文已经超过了 Capric 卡接收存储器读完成报文的速度，因此在分析中我们使用 D2 延时，其中 D2 为 Capric 接收存储器读完成报文的延时。Capric 卡接收一个长度为 84 B 的报文时间约为 110ns（实际需要 10.5 个时钟周期）。因此 Dm 的计算方法如公式 12-14 所示。

$$D_m = D2 \times X/64 = 110 \times X/64 \tag{12-14}$$

由以上分析，可以发现在 PCIe 总线中，Capric 卡 DMA 读比 DMA 写的过程略微复杂。假设 Capric 卡一次 DMA 读的大小为 X[⊖]字节，则这次 DMA 读所需的时间 T_{dmar} 如公式 12-15 所示。

$$T_{dmar} = 2 \times D0 + 40 + 110 \times X/64 + 2 \times D0 + 2 \times D0 \tag{12-15}$$

由以上公式，可以发现当 D0 越大时 T_{dmar} 也越大，但是当 X 越大时，D0 在 T_{dmar} 中所占的比重越小。假设 D0 为 250 ns 时，可以根据公式 12-13 获得 Capric 卡 DMA 读的有效带宽，如表 12-4 所示。

表 12-4　X 的大小与 DMA 读有效带宽的关系

T_{dmaw}（ns）	X	Capric 卡的最大有效带宽
1760	128 B	72.7 MB/s
1980	256 B	129.3 MB/s
2420	512 B	211.6 MB/s
3300	1 KB	310.3 MB/s
5060	2 KB	404.7 MB/s
8580	4 KB	477.4 MB/s
15620	8 KB	524.4 MB/s
29700	16 KB	551.6 MB/s

由表 12-3 和表 12-4 可以发现，Capric 卡的 DMA 写的效率略高于 DMA 读的效率。以上有关 Capric 卡 DMA 读写效率是一个粗粒度的分析，其分析结果并没有考虑数据链路层、物理层和流量控制的开销，也没有考虑发送接收流水线中断的情况，是一个较为理想的结果。

⊖ X 可以被 64 整除。

笔者的实测结果与表 12-3 和表 12-4 有些差距，因为在 x86 处理器系统中，D0 延时时间远大于 250 ns。

12.4.3 Capric 卡的优化

由上两节的分析可以发现，制约 Capric 卡 DMA 读写带宽的主要因素由两部分组成，一是 PCIe 总线的链路带宽，二是在数据传送过程中的延时。使用更宽的 PCIe 链路，显然可以增加带宽。Capric 卡可以使用 ×8 的 PCIe 链路进一步提高物理带宽。但这并不是本章所侧重的提高物理带宽的方法，因为在一个给定的处理器系统中，PCIe 链路的带宽是一定的，设计者已经充分考虑了这些带宽。本章所侧重的是通过减少系统延时以提高带宽。

1. 减少处理器对 Capric 卡的寄存器读操作

由上文的分析可以发现，处理器读取 Capric 卡的寄存器需要通过两个步骤，一是发送存储器读请求 TLP，EP 接收到这个读请求后，再向 RC 发送存储器读完成 TLP。这个读操作的操作延时为 2 × D0，这个延时是处理器系统所无法承受的，更为重要的是这个延时将严重阻塞设备进行 DMA 读写操作的流水线。

为此 Capric 卡将处理器需要读取的数据，以 DMA 写的方式预先写入到主存储器，之后处理器只需要读取主存储器即可获得相应的信息。处理器读取主存储器的延时与读取 EP 中的寄存器相比，可以忽略不计。而 Capric 卡进行 DMA 写这段延时，可以掩盖在数据传送的流水中，从而不会影响数据传送的效率。

在上文中，处理器在执行中断处理程序时，还需要读取中断控制状态寄存器。这个读取寄存器的开销也是可以避免的，因为 MSI/MSIX 中断机制支持“Multiple Message”。在 Capric 卡的实现中，发送完成、接收完成和错误处理中断请求可以与独立的中断服务例程对应。因此可以有效避免在发送完成、接收完成中断服务例程中读取中断控制状态寄存器，从而减少系统的延时。

2. 流水线技术

使用流水线技术可以有效地掩盖数据传送中的延时，从而提高带宽。在 PCIe 设备中，通常使用 Ring Buffer 技术，实现多路 DMA 读写操作的并发执行，从而显著提高 DMA 读写的效率。在 Cornus 卡中使用了 Ring Buffer 技术，与 Capric 卡相比，极大提高了 DMA 读写效率。采用 Ring Buffer 的这种方式也称为 Ring – Based DMA 机制。

目前 Ring Buffer 技术在网络设备中得到了广泛的应用，在此类网卡中，分别为发送部件和接收部件设置了一个 Ring Buffer，在 Ring Buffer 中的每一个 Entry 对应一个 DMA 描述符。使用该技术相当于在网卡中设置了多个虚拟 DMA 通路，并使用流水机制掩盖数据传送中的延时，从而提高数据传送的有效带宽。

Ring Buffer 机制还可以进一步升级为 List/Ring Buffer 机制。所谓 List/Ring Buffer 机制是为发送部件和接收部件设置多个 Ring Buffer，从而进一步提高物理链路的利用率。读者可参阅 e1000e 系列的网卡，或者 PowerPC 处理器的 TSEC 控制器，以了解 Ring Buffer 和 List/Ring Buffer 的实现机制。本节对此不做进一步描述。但是无论使用 Ring Buffer 还是 List/Ring Buffer 机制都很难进一步提高基于 PCIe 总线的网卡的数据传送率。

处理器可以根据用途分为 Control-Plane 处理器和 Data-Plane 处理器，PCIe 总线的主要功能是作为 Control-Plane 处理器的局部总线，并不是 Data-Plane 处理器的局部总线。从体系结

构的角度来看，基于 PCIe 总线的网络设备远不能与 Data-Plane 处理器的网络设备在传送效率，尤其是小报文的传送效率上，一较高低。

12.5 小结

本章通过 Capric 卡的设计实例，讲述了在 PCIe 体系结构中，事务层的硬件实现和 Linux 设备驱动程序的编写方法。其中与 Cache 一致性相关的话题最为重要，虽然这部分内容对于 PCIe 总线并不重要，但是系统程序员需要深入理解相关的概念。

本章的最后，简单分析了影响 PCIe 总线传送效率的因素，理解这些内容可以使读者在系统设计中，选择最合适的局部总线，以构建一个合理的处理器系统。值得注意的是，不同的局部总线所适用的应用领域并不相同。

第 13 章　PCIe 总线与虚拟化技术

目前虚拟化技术在处理器体系结构中，已经占据一席之地。虚拟化技术由来已久，其含义也较为广泛，多个进程共享一个 CPU，多个进程的虚拟空间共享同一个物理内存等一系列在体系结构中已经根深蒂固的概念，都可以归于虚拟化技术。

本章所强调的虚拟化技术是指在一个处理器系统[㊀]中运行多个虚拟处理器系统的技术。其中每一个虚拟处理器系统都有独立的虚拟运行环境，包括 CPU、内存和外部设备。在这个虚拟环境中运行的操作系统彼此独立，但是这些操作系统仍使用相同的物理资源。

因此处理器需要为虚拟化环境设置专门的硬件，以支持多个虚拟处理器系统在一个物理环境中的资源共享。虚拟化技术的核心是通过 VMM（Virtual Machine Monitor）集中管理物理资源，而每个虚拟处理器系统通过 VMM 访问实际的物理资源。有时为了提高虚拟机访问外部设备的效率，虚拟处理器系统也可以直接访问物理资源。

在一个处理器系统中，这些物理资源包括 CPU、主存储器、外部设备和中断。IA 处理器[㊁]使用 EPT（Extended Page Table）和 VPID 技术对主存储器进行管理，而使用虚拟中断控制器接管中断请求以实现中断的虚拟化。目前这些技术较为成熟，对这些内容感兴趣的读者可参阅《系统虚拟化——原理与实现》，本章对此不做详细分析。

本章重点关注的是 VMM 对外部设备的管理，而在外部设备中重点关注对 PCI 设备的管理。在一个处理器系统中，设置了许多专用硬件，如 IOMMU、PCIe 总线的 ATS 机制、SR-IOV（Single Root I/O Virtualization）和 MR-IOV（Multi-Root I/O Virtualization）机制，便于 VMM 对外部设备的管理。

13.1　IOMMU

在多进程环境下，处理器使用 MMU 机制，使得每一个进程都有独立的虚拟地址空间，从而各个进程运行在独立的地址空间中，互不干扰。MMU 具有两大功能，一是进行地址转换，将分属不同进程的虚拟地址转换为物理地址；二是对物理地址的访问进行权限检查，判断虚实地址转换的合理性。

在多数操作系统中，每一个进程都具有独立的页表存放虚拟地址到物理地址的映射关系和属性。但是如果进程每次访问物理内存时，都需要访问页表时，将严重影响进程的执行效率。为此处理器设置了 TLB（Translation Lookaside Buffer）作为页表的 Cache。如果进程的虚拟地址在 TLB 中命中时，则从 TLB 中直接获得物理地址，而不需要使用页表进行虚实地址转换，从而极大提高了访问存储器的效率。

从地址转换的角度来看，IOMMU 与 MMU 较为类似。只是 IOMMU 完成的是外部设备地

㊀ 包括 SMP 系统和更为复杂的 NUMA 结构处理器系统。

㊁ 本章出现的 IA 处理器是指 Intel 的 x86-64 处理器，而不是指 Itanium 处理器。

址到存储器地址的转换。我们可以将一个 PCI 设备模拟成为处理器系统的一个特殊进程，当这个进程访问存储器时使用特殊的 MMU，即 IOMMU，进行虚实地址转换，然后再访问存储器。在这个 IOMMU 中，同样存在 IO 页表存放虚实地址转换关系和访问权限，而且处理器为了加速这种虚实地址的转换，还设置了 IOTLB 作为 IO 页表的 Cache。单纯从这个角度来看，许多 HOST 主桥和 RC 也具备同样的功能，如 PowerPC 处理器的 Inbound 窗口和 Outbound 窗口，也可以完成这种特殊的地址转换。但是这些窗口仅能完成 PCI 总线域到一个存储器域的地址转换，无法实现 PCI 总线域到多个存储器域的转换。

目前设置 IOMMU 的主要作用是支持虚拟化技术，当然使用 IOMMU 也可以实现其他功能，如使“仅支持 32 位地址的 PCI 设备”访问 4GB 以上的存储器空间。IA 处理器和 AMD 处理器分别使用 VT-d”和“IOMMU”，实现外部设备的地址转换。这两种技术都可以将 PCI 总线域地址空间转换为不同的存储器域地址空间，便于虚拟化技术的设计与实现。

13.1.1 IOMMU 的工作原理

根据虚拟化的理论，假设在一个处理器系统中存在两个 Domain，其中一个为 Domain 1，而另一个为 Domain 2。这两个 Domain 分别对应不同的虚拟机，并使用独立的物理地址空间，分别为 GPA1（GPA 即 Guest Physical Address）和 GPA2 空间，其中在 Domain 1 上运行的所有进程使用 GPA1 空间，而在 Domain 2 上运行的所有进程使用 GPA2 空间。

GPA1 和 GPA2 采用独立的编码格式，其地址都可以从各自 GPA 空间的 0x0000-0000 地址开始，只是 GPA1 和 GPA2 空间在 System Memory 中占用的实际物理地址 HPA（Host Physical Address）并不相同，HPA 也被称为 MPA（Machine Physical Address），是处理器系统中真实的物理地址。而 PCI 设备依然使用 PCI 总线域地址空间，PCI 总线地址需要通过 DMA-Remapping 逻辑转换为 HPA 地址后，才能访问存储器。DMA-Remapping 逻辑的组成结构如图 13-1 所示。

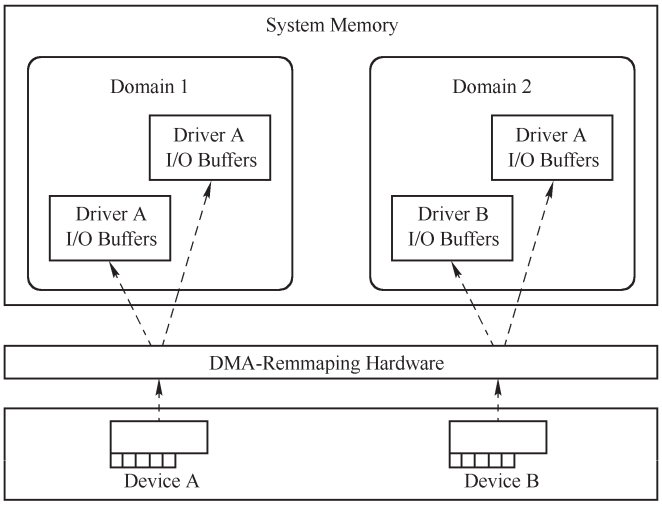

图 13-1　DMA-Remapping 的实现

在以上处理器模型中，假设存在两个外部设备 Device A 和 Device B。这两个外部设备分

属于不同的 Domain，其中 Device A 属于 Domain 1，而 Device B 属于 Domain 2。在同一段时间内，Device A 只能访问 Domain 1 的 GPA1 空间，也只能被 Domain 1 操作；而 Device B 只能访问 GPA2 空间，也只能被 Domain 2 操作。Device A 和 Device B 通过 DMA-Remmaping 机制最终访问不同 Domain 的存储器。

使用这种方法可以保证 Device A/B 访问的空间彼此独立，而且只能被指定的 Domain 访问，从而满足了虚拟化技术要求的空间隔离。这一模型远非完美，如果每个 Domain 都可以自由访问所有外部设备当然更加合理，但是单纯使用 VT-d 机制还不能实现这种访问机制。

在这种模型之下，Device A/B 进行 DMA 操作时使用的物理地址仍然属于 PCI 总线域的物理地址，Device A/B 仍然使用地址路由或者 ID 路由进行存储器读写 TLP 的传递。值得注意的是虽然在 x86 处理器系统中，这个 PCI 总线地址与 GPA 地址一一对应且相等，但是这两个地址所代表的含义仍然完全不同。

GPA 地址为存储器域的地址，隶属于不同的 Domain，而 PCI 设备使用的地址依然是 PCI 总线域的物理地址，只是在虚拟化环境下，PCI 设备与 Domain 间有明确的对应关系。当这个 PCI 设备进行 DMA 读写时，TLP 首先到达地址转换部件 TA（Translation Agent），并通过 ATPT[⊖]（Address Translation and Protection Table）后将 PCI 总线域的物理地址转换为与 GPA 地址对应的 HPA 地址，然后对主存储器进行读写操作。其转换关系如图 13-2 所示。

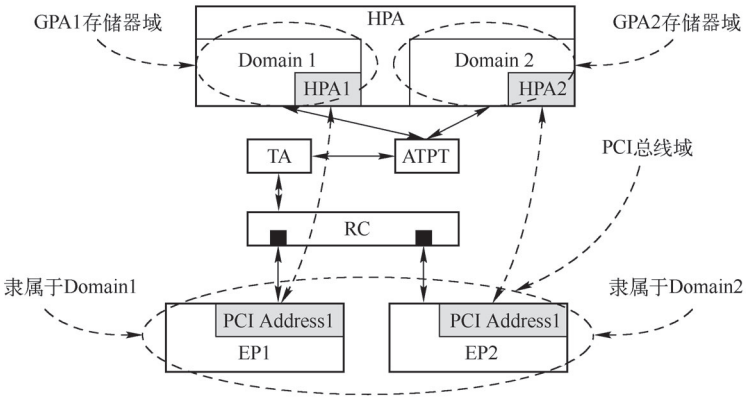

图 13-2　PCI 总线域物理地址与 HPA 的关系

在上图所示的处理器系统中，存在两个虚拟机，其使用的地址空间分别为 GPA Domain1 和 GPA Domain2。假设每个 GPA Domain 使用 1GB 大小的物理地址空间，而且 Domain 间使用的地址空间独立，其地址范围都为 0x0000-0000 ~ 0x4000-0000。其中 Domain 1 使用的 GPA 地址空间对应的 HPA 地址范围为 0x0000-0000 ~ 0x3FFF-FFFF；Domain 2 使用的 GPA 地址空间对应的 HPA 地址范围为 0x4000-0000 ~ 0x7FFF-FFFF。在一个处理器系统中，不同的虚拟机使用的物理空间是隔离的。

在这个处理器系统中存在两个 PCIe 设备，分别为 EP1 和 EP2，其中 EP1 隶属于 Domain1，而 EP2 隶属于 Domain2，即 EP1 和 EP2 进行 DMA 操作时只能访问 Domain1 和 Do-

⊖ ATPT 相当于 I/O 页表，每一个 Domin 都具有独立的 I/O 页表。

main2 对应的 HPA 空间，但是 EP1 和 EP2 作为一个 PCIe 设备，并不知道处理器系统进行的这种绑定操作，EP1 和 EP2 依然使用 PCI 总线域的地址进行正常的数据传送。因为处理器系统的这种绑定操作由 TA 和 ATPT 决定，而对 PCIe 设备透明。在 EP1 和 EP2 进行 DMA 操作时，当 TLP 到达 TA 和 ATPT，经过地址转换后，才能访问实际的存储器空间。

下面以 EP1 和 EP2 进行 DMA 写操作为例，说明在这种虚拟化环境下，不同种类地址的转换关系，其步骤如下所示。

（1）Domain1 和 Domain2 填写 EP1 和 EP2 的 DMA 写地址和长度寄存器启动 DMA 操作。

其中 EP1 最终将数据写入到 GPA1 的 0x1000-0000 ~ 0x1000-007F 这段数据区域，而 EP2 最终将数据写入到 GPA2 的 0x1000-0000 ~ 0x1000-007F 这段数据区域。然而 EP1 和 EP2 仅能识别 PCI 总线域的地址。Domain1 和 Domain2 填入 EP1 和 EP2 的 DMA 写地址为 0x1000-0000，而长度为 0x80，这些地址都是 PCI 总线地址。

在 x86 处理器系统中，这个地址与 GPA1 和 GPA2 存储器域的地址恰好相等，但是这个地址仍然是 PCI 总线域的地址，只是由于 IOMMU 的存在，相同的 PCI 总线地址，可能被映射到相同的 GPA 地址空间，然而这些 GPA 地址空间对应的 HPA 地址空间不同。这个 PCI 总线地址仍然在 RC 中被转换为存储器域地址，并由 TA 转换为合适的 HPA 地址。

（2）EP1 和 EP2 的存储器写 TLP 到达 RC。

来自 EP1 和 EP2 存储器写 TLP 经过地址路由最终到达 RC，并由 RC 将 TLP 的地址字段转发到 TA 和 ATPT，进行地址翻译。

EP1 和 EP2 使用的 I/O 页表已经事先被 VMM 设置完毕，TA 将使用 Domain1 或者 Domain2 的 I/O 页表，进行地址翻译。EP1 隶属于 Domain1，其地址 0x1000-0000（PCI 总线地址）被翻译为 0x1000-0000（HPA）；而 EP2 隶属于 Domain2，其地址 0x1000-0000（PCI 总线地址）被翻译为 0x5000-0000（HPA）。值得注意的是在 TA 中设置了 IOTLB，以加速 I/O 页表的翻译效率，因此 TA 并不会每次都从存储器中查找 I/O 页表。

（3）来自 EP1 和 EP2 存储器写 TLP 的数据将被分别写入到 0x1000-0000 ~ 0x1000-007F 和 0x5000-0000 ~ 0x5000-007F 这两段数据区域。

（4）Domain1 和 Domain2 都使用 0x1000-0000 ~ 0x1000-007F 这段 GPA 地址访问来自 EP1 和 EP2 的数据，这个 GPA 地址将转换为 HPA 地址，然后发向存储器控制器。在 IA 处理器系统中，使用 EPT 和 VPID 技术进行 GPA 地址到 HPA 地址的转换。

IA 处理器和 AMD 处理器使用不同的技术，实现 TA 和 ATPT。其中 IA 处理器使用 VT-d 技术，而 AMD 使用 IOMMU。从工作原理上看，这两种技术类似，但是在实现细节上，两者有较大区别。

13.1.2 IA 处理器的 VT-d

IA（Intel Architecture）处理器使用 VT-d 技术将 PCI 总线域的物理地址转换为 HPA 地址。这个映射过程也被称为 DMA Remapping。IA 处理器系统使用 DMA Remapping 机制可以辅助虚拟化技术对外部设备进行管理。

在 IA 处理器系统中，所有的外部设备都是 PCI 设备。每一个设备都唯一对应一个 Bus Number、Device Number 和 Function Number，为此 IA 处理器设置了一个专门的结构，即 Root

Entry Table[⊖]，管理每一棵 PCI 总线树。在这种结构下，每一个 PCI 设备根据其 Bus、Device 和 Function 号唯一确定一个 Context Entry。VT-d 将这个结构称为“Device to Domain Mapping”结构，如图 13-3 所示。

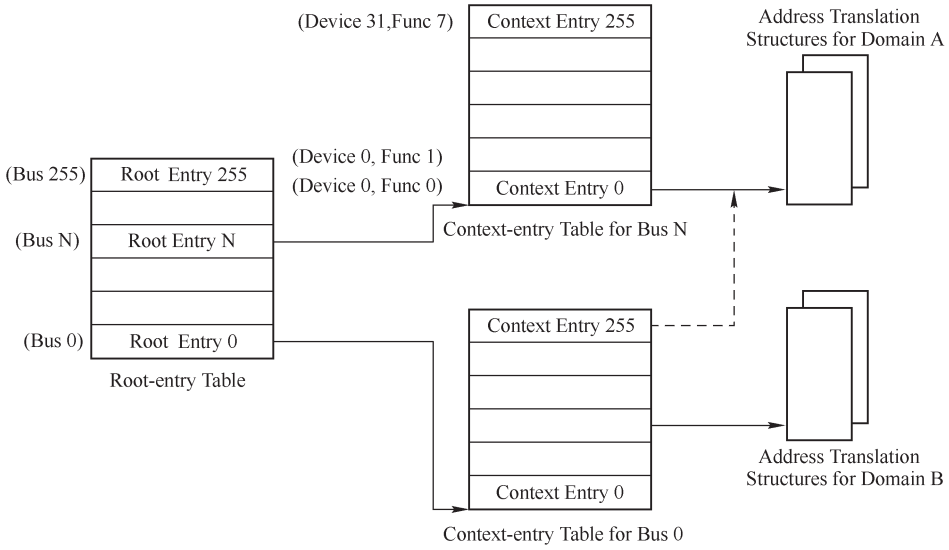

图 13-3　Device to Domain Mapping 结构

VT-d 一共设置了两种结构描述 PCI 总线树结构，分别为 Root Entry 和 Context Entry。其中 Root Entry 描述 PCI 总线，一棵 PCI 总线树最多有 256 条 PCI 总线，其中每一条 PCI 总线对应一个 Root Entry；每条 PCI 总线中最多有 32 个设备，而每个设备最多有 8 个 Function，其中每一个 Function 对应一个 Context Entry，因此每个 Context Entry 表中共有 256 表项。

在一个处理器系统中，一个指定的 PCI Function 唯一对应一个 Context Entry，这个 Context Entry 指向这个 PCI Function 使用的地址转换结构（Address Translation Structures）。当一个 PCI Function 隶属于不同的 Domain 时，将使用不同的地址转换结构，但是在一个时间段里，PCI Function 只能使用一个地址转换结构，即 Context Entry 只能指向一个 Domain 的地址转换结构。这个地址转换结构的主要功能是完成 PCI 总线域到 HPA 存储器域的地址转换。

如图 13-1 所示，当一个设备进行 DMA 操作时，Domain 使用 PCI 总线域的地址填写这个设备和与 DMA 转送相关的寄存器。当这个设备启动 DMA 操作时，将使用 PCI 总线地址，之后通过 DMA Remapping 机制将 PCI 总线地址转换为 HPA 存储器域地址，然后将数据传送到实际的物理地址空间中。而 Domain 通过处理器的 MMU 机制将 GPA 转换为 HPA，访问物理地址空间。

从图 13-3 中可以发现，每一个 Function 在每一个 Domain 中都可能有一个地址转换结构，以完成 GPA 到 HPA 的转换，因此在每一个 Domain 中最多有 256 个地址转换结构。这些结构无疑将占用部分内存，但是并不会产生较大的浪费。因为在实际设计中，同一个 Do-

⊖ 如果处理器系统有多个 PCI 总线树（Segment），则需要设备多个 Root Entry Table。

main 下的所有 PCI 设备使用的总线地址到 HPA 地址的转换结构可以相同。因此在实现中，每个 Domain 仅使用一个地址转换结构即可。

IA 处理器使能 VT-d 机制后，PCI 设备进行 DMA 操作需要根据 Bus、Device 和 Function 号确定 Context Entry，之后使用图 13-4 所示的方法完成 PCI 总线地址到 HPA 地址的转换。

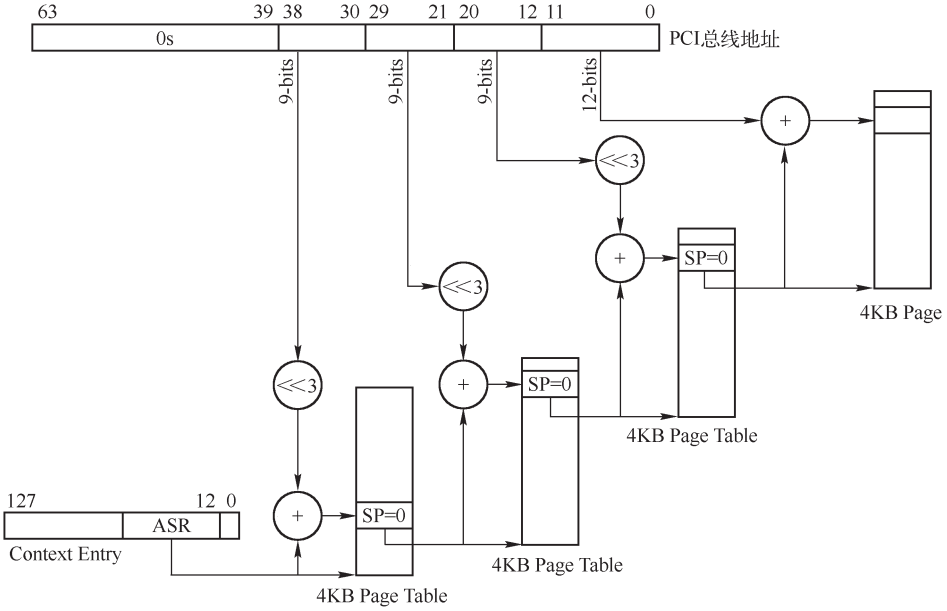

图 13-4　VT-d 使用的 PCI 总线地址到 HPA 的转换机制

在上图中，4KB Page Table 中的每个 Entry 的大小为 8B，因此在计算偏移时，需要左移 3 位，PCI 总线地址通过 3 级目录，最终找到与 HPA 所对应的 4KB 大小的页面，从而完成 PCI 总线地址到 HPA 的转换。值得注意的是，IA 处理器还支持 2MB（SP = 1）、1GB（SP = 2）、512GB（SP = 3）和 1TB（SP = 4）大小的 Super Page，而本节仅使用了 4KB 大小的页面。

为了加快 PCI 总线地址到 HPA 地址的转换速度，IA 处理器分别为 Root Entry 和 Context Entry 设置了 Context Cache 以加快 Context Entry 的获取速度，同时还设置了 IOTLB 加速 PCI 总线地址到 HPA 地址的转换速度。

IOTLB 相当于 I/O 页表的 Cache，当一个 PCI 设备进行 DMA 操作时，首先在 IOTLB 中查找 PCI 总线地址与 HPA 地址的映射关系，如果在 IOTLB 命中时，PCI 设备直接获得 HPA 地址进行 DMA 操作；如果没有在 IOTLB 命中，则需要使用图 13-4 中所示的算法进行 PCI 总线地址到 HPA 地址的转换。

Intel 并没有公开“没有在 IOTLB 命中”的实现细节，当出现这种情况时，IA 处理器可能使用内部的 Microcode 完成图 13-4 所示的算法。使用 VT-d 除了可以有效地支持虚拟化技术之外，还可以支持一些只能访问 32 位地址空间的 PCI 设备访问 4GB 之上的物理地址空间。

13.1.3　AMD 处理器的 IOMMU

AMD 处理器的 IOMMU 技术与 Intel 的 VT-d 技术类似，其完成的主要功能也类似。AMD

率先提出了 IOMMU 的概念，并发布了 IOMMU 的技术手册，但是 Intel 首先将这一技术在芯片中实现。由于 AMD 和 Intel 使用的 x86 体系结构略有不同，因此 AMD 的 IOMMU 技术在细节上与 Intel 的 VT-d 并不完全一致。

AMD 处理器使用 HT（Hyper Transport）总线连接 I/O Hub，其中每一个 I/O Hub 都含有一个 IOMMU，其结构如图 13-5 所示。

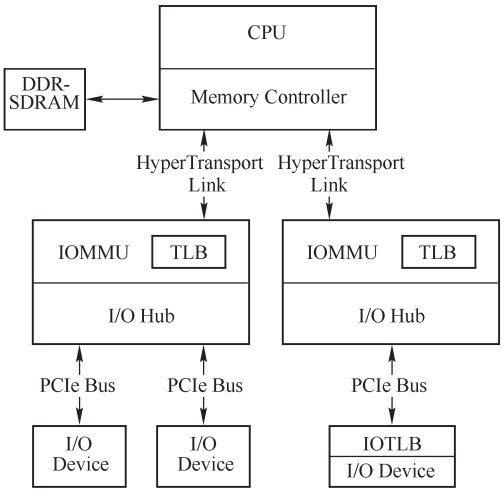

图 13-5　AMD 处理器的 IOMMU 结构

其中每一个 IOMMU 都使用一个 Device Table。AMD 处理器使用 Device Table 存放图 13-3 中的结构，Device Table 最多由 2^{16} 个 Entry 组成，其中每个 Entry 的大小为 256b，因此 Device Table 最大将占用 2 MB 内存空间，与 Intel 使用的 Root/Context Entry 结构相比，AMD 使用的这种方法容易造成内存的浪费。

在 I/O Hub 中的设备[⊖]使用 16 位的 DeviceID 在 Device Table 查找该设备所对应的 Entry，并使用这个 Entry，根据 I/O Page Table 结构最终找到 IO PTE 表，并完成 GPA 到 HPA 的转换。在 AMD 处理器中，GPA 到 HPA 的转换与图 13-4 中所示的方法有类似之处，但实现细节不同。IOMMU 使用一个新型的页表结构完成 GPA 到 HPA 的转换，这个页表结构基于 AMD64 使用的虚拟地址到物理地址的页表结构，但是做出了一些改动。AMD64 进行虚拟地址到物理地址的转换时使用 4 级页表结构，如图 13-6 所示。

与 Intel 处理器的结构类似，一个进程首先从 CR3 寄存器中获得页表的基地址指针寄存器“Page Map Level-4 Base Address”，之后通过 4 级索引最终获得 4KB 大小的物理页面，完成虚拟地址到物理地址的转换。AMD 处理器也支持大页面方式，如果使用三级索引，可以获得 2MB 大小的物理页面；使用二级索引，可以获得 1 GB 大小的页面。

IOMMU 使用的 I/O 页表结构基于以上结构，但是做出了一定的改动。在 IOMMU 中，4 级 I/O 页表指针可以直接指向 2 级 I/O 页表指针，从而越过第 3 级 I/O 页表，使用这种方法可以节省 Page Table 的空间。如图 13-7 所示。

⊖ 其中 PCI 设备使用 Bus Number、Device Number 和 Function Number 组成 16 位的 Device ID，而 HT 设备使用 HT Bus Number 和 Unit ID 组成 16 位的 Device ID。

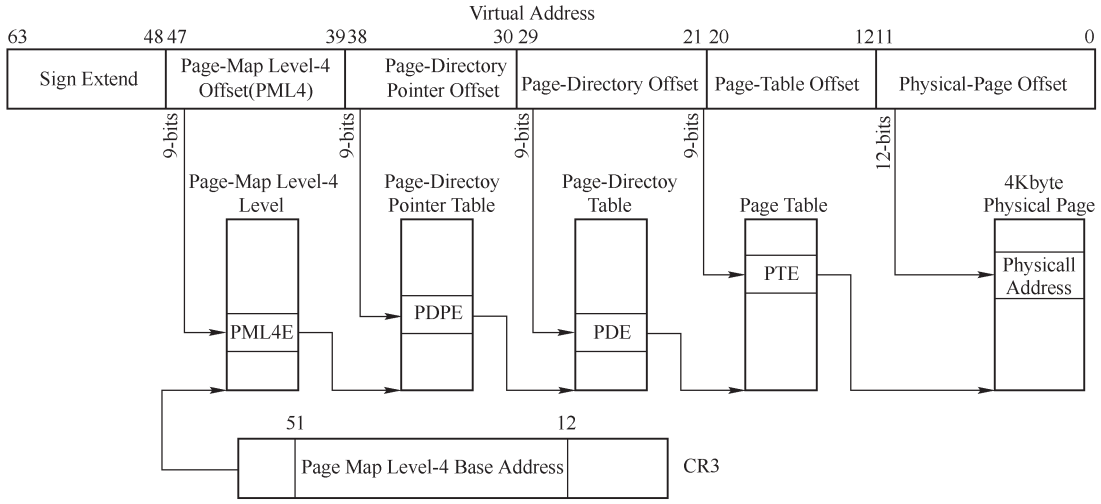

图 13-6　AMD64 虚拟地址到物理地址的页表结构

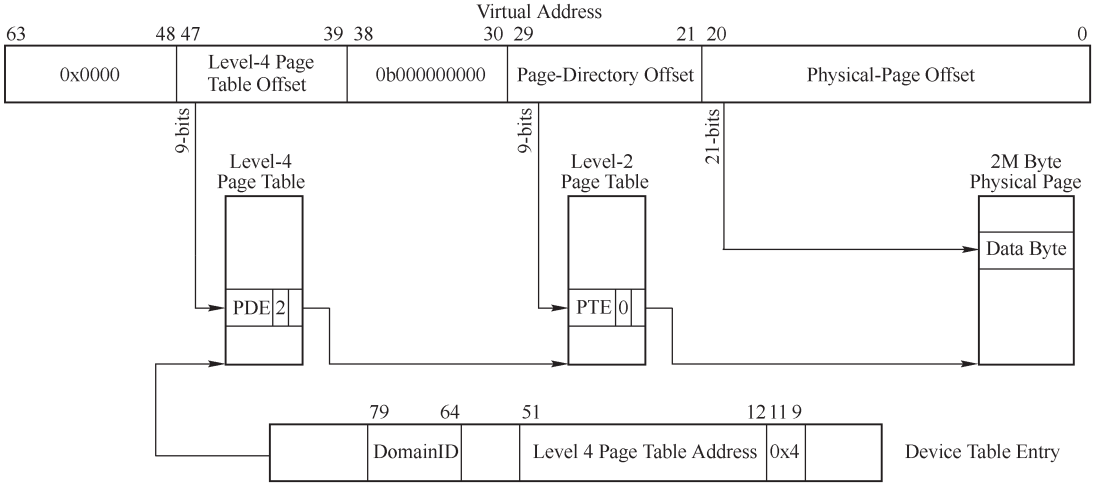

图 13-7　IOMMU 使用的 GPA 到 HPA 的转换机制

当设备进行 DMA 操作时，首先需要从相应的 Device Table 的 Entry 中获得“Level 4 Page Table Address”指针，并定位设备使用的 I/O 页表，最后使用多级页表结构，最终完成 PCI 总线地址到 HPA 地址的转换。Page Table 的 Entry 由 64 位组成，其主要字段如下所示。

- 第 51 ~ 12 位为 Next Table Address/Page Address 字段，该字段存放下一级页表或者物理页面的地址，该地址为系统物理地址，属于 HPA 空间。
- 第 11 ~ 9 位为 Next Level 字段，表示下一级页表的级数，其中在 Device Table 中存放的级数一般为 4，Level-N 级页表中存放的 Next Level 字段为 N-1 ~ 1。

如图 13-7 所示，在第 4 级页表的 Entry 中的 Next Level 字段为 2，表示第 4 级页表直接指向第 2 级页表，而忽略第 3 级页表。当该字段为 0b000 或者 0b111 时，表示下一级指针指向物理页面而不是页表。Next Level 字段为 0b000 时，表示所指向的物理页面的大小是固定的，AMD64 支持 4 KB、2 MB、1 GB、512 GB 和 1TB（SP = 4）大小固定页面；如果 Next

Level 字段为0b111 时，表示所指向的物理页面大小是浮动的。如果 Level 2 Page Table 的 Entry 中的 Next Level 字段为 0b111，表示该 Entry 指向的物理页面大小浮动，其中物理页面大小和 GPA 的第 29 ~21 位相关，如表 13-1 所示。

表 13-1　Next Level 字段为 0b111 时的页表大小

29	28	27	26	25	24	23	22	21	Page Size	Default Page Size
Page Address								0	4 MB	2 MB
Page Address							0	1	8 MB	2 MB
Page Address						0	1	1	16 MB	2 MB
Page Address					0	1	1	1	32 MB	2 MB
Page Address				0	1	1	1	1	64 MB	2 MB
Page Address			0	1	1	1	1	1	128 MB	2 MB
...		0	1	1	1	1	1	1	256 MB	2 MB
...	0	1	1	1	1	1	1	1	512 MB	2 MB

AMD64 处理器使用这种 IO 页表方式，可以方便地支持 4KB、8KB、……、4GB 大小的浮动物理页面。除了 I/O 页表外，IOMMU 也设置了 IOTLB 以加快 GPA 到 HPA 地址的转换，这部分内容与 IA 处理器的实现方式类似，本章不对此继续进行描述。对 IOMMU 感兴趣的读者可以参考 AMD I/O Virtualization Technology Specification。

13.2　ATS（Address Translation Services）

单纯使用 IOMMU 并不能充分发挥处理器系统的效率，从图 13-2 中可以发现，所有 PCI 设备在进行 DMA 操作时，都需要经过 TA 和 ATPT 进行地址翻译，然后才能访问主存储器。因而 TA 和 ATPT 很容易成为瓶颈，从而影响虚拟化系统的整体效率。

除此之外，在图 13-2 中，EP1 和 EP2 分别隶属于 Domain1 和 Domain2。在正常情况下，一个 Domain 并不能访问其他 Domain 的 PCI 设备。但是如果处理器系统中存在一个恶意的虚拟机，而且 EP1 隶属于该虚拟机（Domain1）。当 EP1 进行 DMA 写操作时，该虚拟机填写的 DMA 写地址可以与 EP2 的 BAR 地址空间重合，那么启动 DMA 写操作时，Domain1 可以将数据传递到 EP2，从而影响 Domain2 的正常运行。

解决这种异常的最合理的方法是，隶属于 Domain1 的 PCI 设备只能访问 GPA1 的空间，而仅使用 IOMMU 并不能解决该问题。解决该问题较为有效的方法是 PCI 设备进行数据传送的同时也进行地址转换，从而该 PCI 设备使用的地址是经过转换的 HPA 地址。此时再进行 DMA 写时，该数据将传递到与 Domain1 对应的 HPA 地址空间中，而不会将数据传送到 EP2。从而这个恶意的虚拟机并不会影响其他正常工作的虚拟机。

PCIe 总线使用 ATS 机制实现 PCIe 设备的地址转换。支持 ATS 机制的 PCIe 设备，内部含有 ATC（Address Translation Cache），ATC 在 PCIe 设备中的位置如图 13-8 所示。

在 ATC 中存放 ATPT 的部分内容，当 PCIe 设备使用地址路由方式发送 TLP 时，其地址首先通过 ATC 转换为 HPA 地址。如果 PCIe 设备使用的地址没有在 ATC 中命中时，PCIe 设备将通过存储器读 TLP 从 ATPT 中获得相应的地址转换信息，更新 ATC 后，再发送 TLP。与

其他 Cache 类似，ATC 还可以被 Invalidate。当 ATPT 被更改时，处理器系统将发送 Invalidate 报文，同步在不同 PCIe 设备中的 ATC。

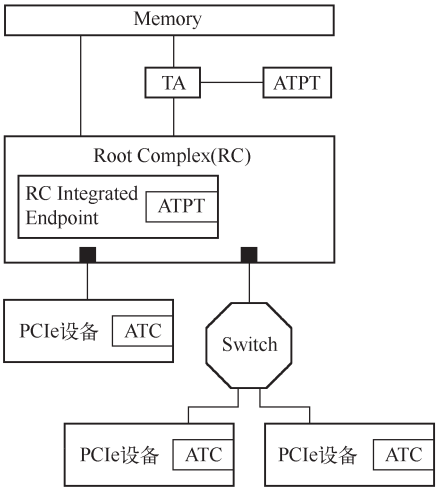

图 13-8　ATC 在 PCIe 设备中的位置

PCIe 总线在 TLP 中设置了 AT 字段以支持 ATS 机制。在 PCIe 总线中，只有与存储器相关的 TLP 支持 AT 字段。值得注意的是，只有处理器系统支持 IOMMU 时，PCIe 设备才可以使用 ATS 机制。

13.2.1　TLP 的 AT 字段

TLP 的 AT 字段与 ATS 机制直接相关。根据 AT 字段的不同，PCIe 设备可以发送三种类型的 TLP。

1. AT 字段为 0b00

当 AT 字段为 0b00 时，当前 TLP 的 Address 字段没有通过 ATC 进行转换，存放的是 PCI 总线域的物理地址。如果 PCIe 设备不支持 ATS 机制，而且处理器系统也没有使能 IOMMU 时，当前 TLP 的 Address 字段为 PCI 总线域的物理地址。PCIe 设备进行 DMA 操作时，该地址将被 RC 转换为存储器域的物理地址，然后对存储器进行读写操作。

如果 PCIe 设备不支持 ATS 机制，但是当前处理器支持 IOMMU 时，当前 TLP 的 Address 字段依然为 PCI 总线域的物理地址。PCIe 设备进行 DMA 操作时，该地址将被 TA 根据 I/O 页表的设置，转换为合适的存储器域物理地址。

如果当前处理器系统支持虚拟化技术，当前 PCIe 设备将隶属于某一个 Domain，此时该 PCIe 设备进行 DMA 操作时，数据将被传送到属于该 Domain 的存储器域中。

2. AT 字段为 0b01

当 AT 字段为 0b01 时，表示当前 TLP 报文为“Translation Request”报文。支持 ATS 机制的 PCIe 设备，必须支持这类报文。

该报文由 PCIe 设备通过存储器读请求 TLP 发出，其目的地为 TA。TA 收到该报文后，将根据 I/O 页表的设置，将合适的地址转换关系，通过存储器读完成 TLP，发送给 PCIe 设备。而 PCIe 设备收到这个地址转换关系后，将更新 ATC。

3. AT 字段为 0x10

当 AT 字段为 0x01 时，表示当前 TLP 的 Address 字段已经通过 ATC 进行地址转换。当 PCIe 设备使用存储器读写报文进行 DMA 操作，而 RC 收到这些报文时，将不再通过 TA 和 ATPT 进行地址转换，而直接将数据发送给存储器。从而减轻了 ATPT 进行地址转换的压力。

值得注意的是，经过 ATC 进行地址转换后，在 TLP 的 Address 字段中存放的依然是 PCI 总线域的物理地址，该物理地址为 HPA 地址在 PCI 总线域中的映像。

如果 TLP 中的 Address 字段没有经过 ATC 进行地址转换，而且处理器系统支持虚拟化技术，该地址为仍然对应 GPA 地址在 PCI 总线域中的映像，此时该 TLP 使用的 AT 字段为 0b00。这些地址在经过 RC 后，将被转换为存储器域的地址，然后进入 TA 和 ATPT 再次进行地址转换。由以上描述可以发现，PCIe 设备无论是否使用 ATC 机制，在 TLP 中存放的 Address 字段仍然保存的是 PCI 总线地址。

13.2.2 地址转换请求

PCIe 设备可以使用地址转换请求（Translation Requests）TLP 向 TA 提交地址转换请求。该 TLP 具有 64 位和 32 位两种地址格式。本节仅介绍 64 位地址格式，如图 13-9 所示。其中 AT 字段为 0b01 表示当前报文为地址转换请求 TLP。

该报文的格式与存储器读请求 TLP 的报文格式基本类似，但是在地址转换请求 TLP 中，一些字段的含义与存储器读请求 TLP 并不相同。该报文的作用是将 Untranslated Address 字段发送到 TA，而 TA 根据 ATPT 将 Untranslated Address 数据区域进行翻译，然后通过存储器读完成 TLP 将地址转换关系发送给 PCIe 设备。PCIe 设备收到这个存储器读完成 TLP 后将这个地址转换关系保存在 PCIe 设备的 ATC 中。

Untranslated Address 数据区域的长度由 Length 字段确定。在地址转换请求 TLP 中，Length 字段的最低位和高 5 位为 0，而且 Length 字段不能为 0b00-0000-0000，因此该地址转换请求 TLP 所访问的数据区域最小为 8B，而最大不能超过 RCB。而且该数据区域为 1DW 对界，First DW BE 与 Last DW BE 字段都为 0b1111。

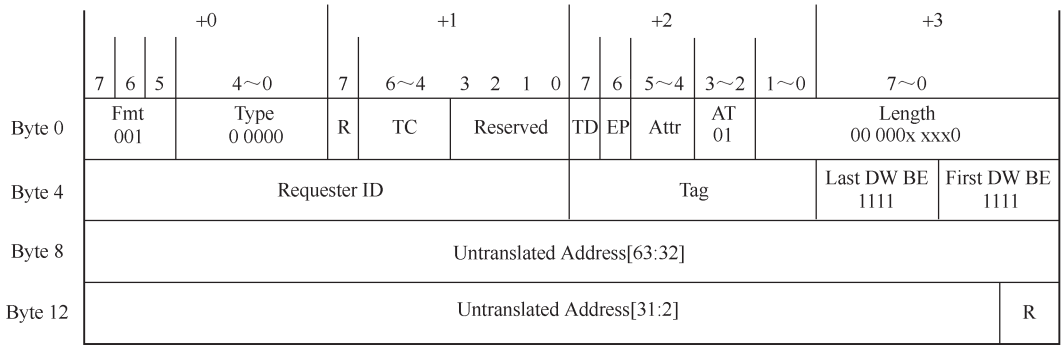

图 13-9　64 位地址转换请求 TLP 的格式

当 PCIe 设备与某个虚拟机绑定时，Untranslated Address 数据区域的 GPA 地址连续，但是其对应的 HPA 地址并不一定连续（在绝大多数虚拟机的实现中，为简化设计，GPA 所对应的 HPA 地址区域地址连续）。因此 PCIe 设备发送一个地址转换请求后，可能会从 TA 得到

多个地址转换关系，这些地址转换关系可以使用一个存储器读完成 TLP 发送给 PCIe 设备。在 PCIe 总线中，一个地址转换关系由 8B 组成，这也是地址转换请求 TLP 的 Length 字段至少为 0b10 的原因。

当 TA 收到地址转换请求 TLP 后，将查找 ATPT，然后通过存储器读完成 TLP，将转换关系发送给 PCIe 设备。如果地址转换成功时，TA 使用 CplD（带数据的存储器读完成报文）将转换关系发送给 PCIe 设备；否则使用 Cpl 将失败信息发送给 PCIe 设备。本节仅讨论 CplD 报文，其格式如图 13-10 所示。

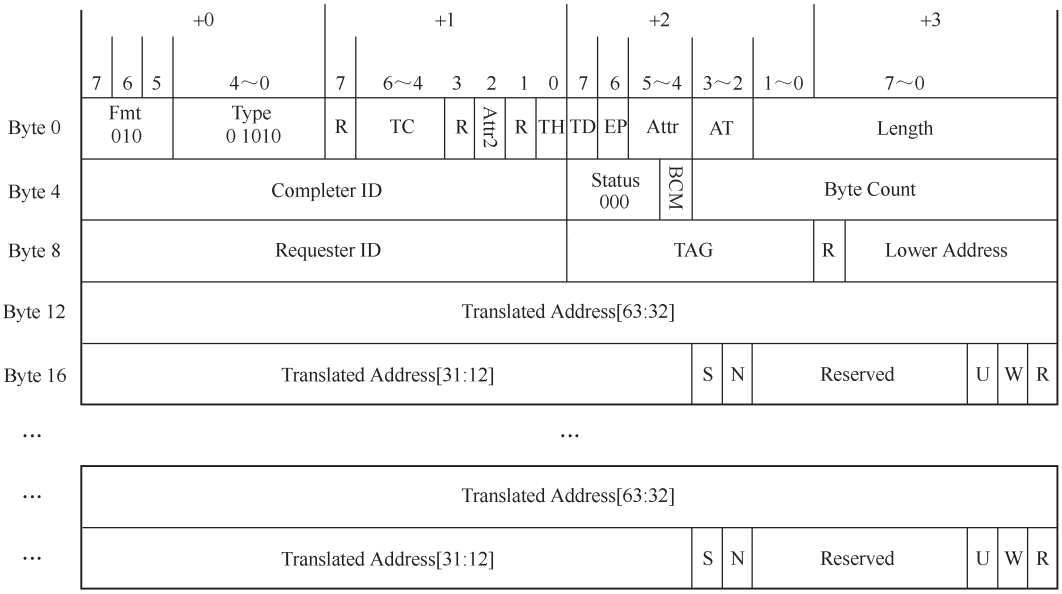

图 13-10　地址转换完成 TLP 的格式

地址转换完成 TLP 的报文头与存储器读完成报文头完全相同，而 Payload 字段由一个或者多个地址转换关系组成。一个地址转换关系由以下字段与位组成。

- Translated Address［63:12］字段保存与 Untranslated Address 对应的 HPA 地址，即经过转换的地址。
- 而 U 位为 1 时，表示这段 HPA 空间只能使用 Untranslated 地址访问，即 PCIe 设备不能使用 AT 字段等于 0b10 的存储器读写 TLP；R 位为 1，表示 HPA 地址空间可读；W 位为 1 时，表示 HPA 地址空间可写。
- N 位为 1 时，PCIe 设备访问这段数据区域时，其存储器读写 TLP 的 No Snoop 位必须为 0，表示硬件需要进行 Cache 一致性操作；如果该位为 0，PCIe 设备将使用其他方法确定 No Snoop 位是否可以为 1。
- S 位需要与 Translated Address 字段联合使用，表示该段数据区域的大小，如表 13-2 所示。

由该表所示，Translated Address 数据区域的最小值为 4KB，此时 S 位必须为 0。如果 S 不为 0，则表示这段数据区域大于 4KB。当 Address31 为 0，而 Address［30：12］和 S 位都为 1 时表示，这段区域为 4GB，但是 4GB 并不是 Translated Address 数据区域的最大值。PCIe

设置还可以使用 Address［63:32］字段，继续扩展数据区域的大小。

表 13-2　Translated Address 区域的大小

Tranlated Address																					S	页面大小/B
63:32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12		
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	4K
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	1	8K
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	1	1	16K
...																						
X	X	X	X	X	X	X	X	X	X	X	X	0	1	1	1	1	1	1	1	1	1	2M
...																						
X	X	X	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1G
X	X	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2G
X	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4G

当 PCIe 设备支持 ATS 机制并进行 DMA 操作时，首先查找当前访问的地址是否在 ATC 中命中，如果命中则直接从 ATC 中获得 Translated Address，并使用 AT 等于 0b10 的存储器读写 TLP 与主存储器交换数据。TA 收到 AT 等于 0b10 的 TLP 后，将不使用 ATPT 进行地址转换，而将报文直接发送到存储器控制器，与主存储器进行数据交换。

如果 PCIe 设备访问的地址区域没有在 ATC 中命中时，PCIe 设备有两种处理方法，一是使用 AT 等于 0b00 的 TLP，即使用 Untranslated Address 直接访问存储器，这个 Untranslated Address 到达 TA 后，TA 根据 ATPT 的设置，将 Untranslated Address 进行地址转换，然后再将报文发送到存储器控制器中。

如果处理器系统中存在恶意的虚拟机时，PCIe 设备使用这种方法时将会带来安全隐患。因为恶意的虚拟机可以直接将这个 Untranslated Address 与其他 PCIe 设备使用的 BAR 空间重合，从而该虚拟机可以破坏隶属于其他虚拟机的 PCIe 设备，干扰其他虚拟机的正常运行，这是虚拟机系统禁止的行为。

因而当 PCIe 设备访问的地址区间没有在 ATC 中命中时，应该首先进行地址转换。采用这种方式时，PCIe 设备将首先向 TA 发送地址转换请求 TLP，并从 ATPT 中获得地址转换关系后，使用 TA 等于 0b10 的存储器读写 TLP，即使用 Translated Address 与主存储器进行数据交换，从而有效避免了上文所述的安全隐患。

目前尚无支持 ATS 机制的 PCIe 设备，但是通过本节的描述可以发现，使用 ATS 机制可以有效减轻 TA 进行地址转换的负担，同时避免虚拟机中存在的安全隐患。

13.2.3　Invalidate ATC

处理器系统更改 ATPT 时，需要使用 MsgD 报文通知相应 PCIe 设备，并 Invalidate ATC 中相应的 Entry，该 MsgD 报文也被称为“Invalidate Request Message”，其格式如图 13-11 所示。其中 PCIe 设备每收到一个 MsgD 只能 Invalidate ATC 中的一个 Entry。

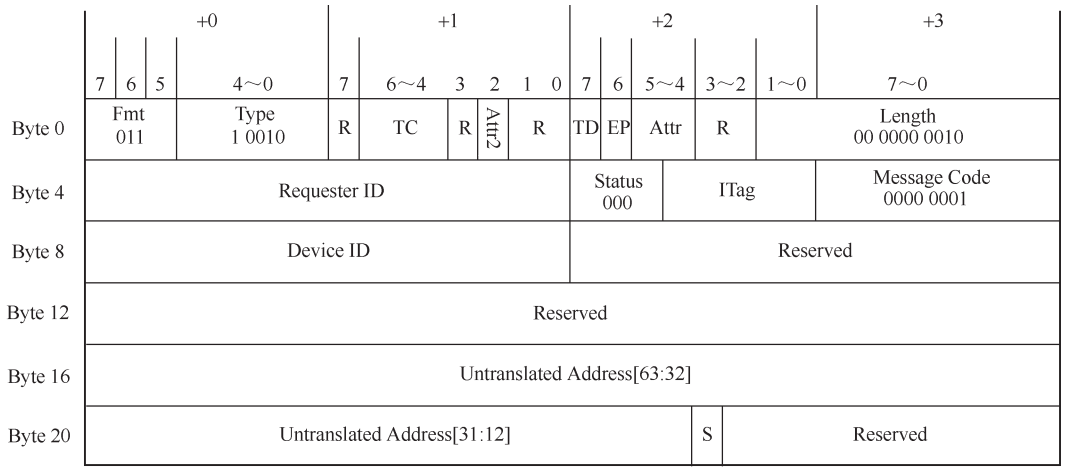

图 13-11　Invalidate Request Message 的格式

Invalidate Request Message 各个字段的描述如下所示。

- Fmt 字段为 0b011，表示报文头为 4DW，而且含有 Payload 字段。Length 字段为 0b10，表示该报文 Payload 的大小为 8B。而 TC、Attr、TD 和 EP 的含义与通用 TLP 头相同，详见第 6.1 节。
- Type 字段为 0b10010，表示该消息报文使用 ID 路由方式。其中 Requester ID 字段保存 TA 的 ID 号。Device ID 保存目标设备使用的 ID 号，即存放 ATC 的 PCIe 设备 ID，Message 报文使用该字段进行 ID 路由。
- ITag 字段与 Tag 字段的功能类似，取值范围为 0 ~ 31。当 TA 需要连续发送多个“Invalidate Request Message”报文时，使用该字段区别不同的 MsgD。

该 MsgD 报文的 Payload 字段中存放“Unstranlated Address”，而 S 位用来表示数据区域的大小，如表 13-2 所示。当 PCIe 设备收到 Invalidate Request Message 报文后，根据 Unstranlated Address 字段 Invalidate ATC 中对应的 Entry。PCIe 设备 Invalidate ATC 中对应的 Entry 之后，将向 TA 发送 Invalidate Completion Message 报文，表示已经 Invalidate ATC 中的对应 Entry，该报文的格式如图 13-12 所示。

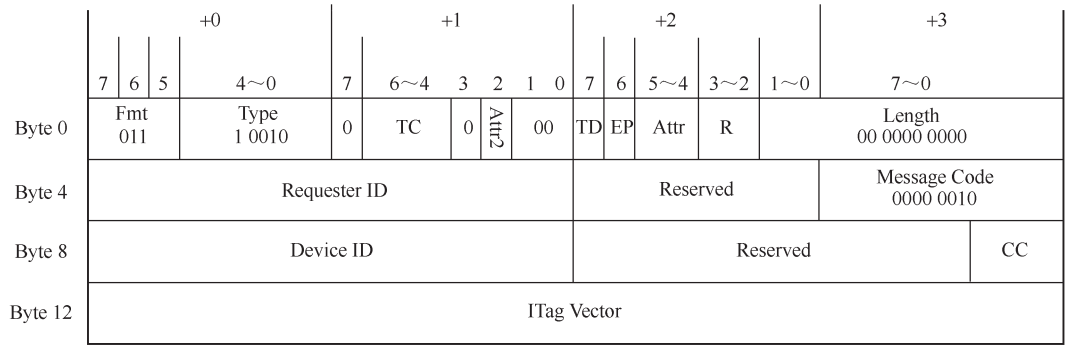

图 13-12　Invalidate Completion Message 的格式

Invalidate Completion Message 报文的各个字段的描述如下所示。

- Fmt、Type 等字段与 Invalidate Request Message 的对应字段类似。
- Requester ID 字段保存 TA 的 ID 号，而 Device ID 字段保存 PCIe 设备的 ID 号。
- CC 字段表示 PCIe 设备需要向 TA 发送 Invalidate Completion Message 报文的个数。当 CC 字段为 0 时表示需要 8 个这样的报文，为 n 时表示需要 n 个这样的报文。n 的最大值为 0x07。
- ITag Vector 字段由 32 位组成，其中每一位对应 Invalidate Request Message 报文的一个 ITag 字段，Invalidate Completion Message 通过 ITag Vector 字段可以向多个 Invalidate Request Message 报文发出回应，表示已经 Invalidate ATC 中的多个 Entry。

13.3 SR-IOV 与 MR-IOV

PCIe 总线除了提供了 ATS 机制外，还使用 SR-IOV 和 MR-IOV 机制，进一步优化虚拟化技术的实现。其中 SR-IOV 技术的主要作用是将一个物理 PCIe 设备模拟成多个虚拟设备，其中每一个虚拟设备可以与一个虚拟机绑定，从而便于不同的虚拟机访问同一个物理 PCIe 设备。在 PCIe 体系结构中，即便使用了 ATS 和 SR-IOV 技术，在处理器系统中仍然只有一个 PCIe 总线域，所有的虚拟机共享这个 PCI 总线域，这为虚拟化技术的实现带来了不小的障碍。使用 SR-IOV 技术，可以解决单个 PCIe 设备被多个虚拟机共享的问题，但是并没有对管理 PCIe 设备的 Switch 进行约束。

提出 MR-IOV 技术的主要目的是解决多个处理器系统对一个 PCI 总线域共享的问题，其本质是将一个物理 PCI 总线域，分解为多个虚拟的 PCI 总线域，多个处理器系统可以与多个 PCI 总线域对应，从而实现了不同 PCI 总线域间的隔离。MR-IOV 技术对 PCIe 总线进行了大规模的扩展，提出了 MRA（Multi-Root Aware）Switch、MRA（Multi_Root Aware）Device 和 MRA RP（Rort Port）的概念，同时对 PCIe 总线的数据链路层和流量控制进行了细微改动。

13.3.1 SR-IOV 技术

在 SR-IOV 技术没有引入之前，一个 PCIe 设备在一个指定的时间段内，只能与一个虚拟机（Domain 1）绑定，而其他虚拟机（Domain 2）访问与 Domain 1 绑定的 PCIe 设备时，需要首先向 Domain 1 发送请求，由 Domain 1 从 PCIe 设备获得数据后，再传送给 Domain 2。使用这种方法将极大增加在虚拟化环境下，虚拟机访问 PCIe 设备的延时，同时也干扰了其他虚拟机的正常运行。

而在处理器系统中并行设置多个同样的物理设备，不仅增加了系统成本，而且增加了处理器系统的规模，从而造成不必要的浪费。SR-IOV 技术在此背景下诞生。支持 SR-IOV 的 PCIe 设备，由多组虚拟子设备组成，其拓扑结构如图 13-13 所示。

由上图所示，基于 SR-IOV 的 PCIe 设备由多个物理子设备 PF（Physical Function）和多组虚拟子设备 VF（Virtual Function）组成。其中每一组 VF 与一个 PF 对应。在上图中存在 M 个 PF，分别为 PF0～M。其中“VF0，1～N1”与 PF0 对应；而“VFM，1～N2”与 PFM 对应。

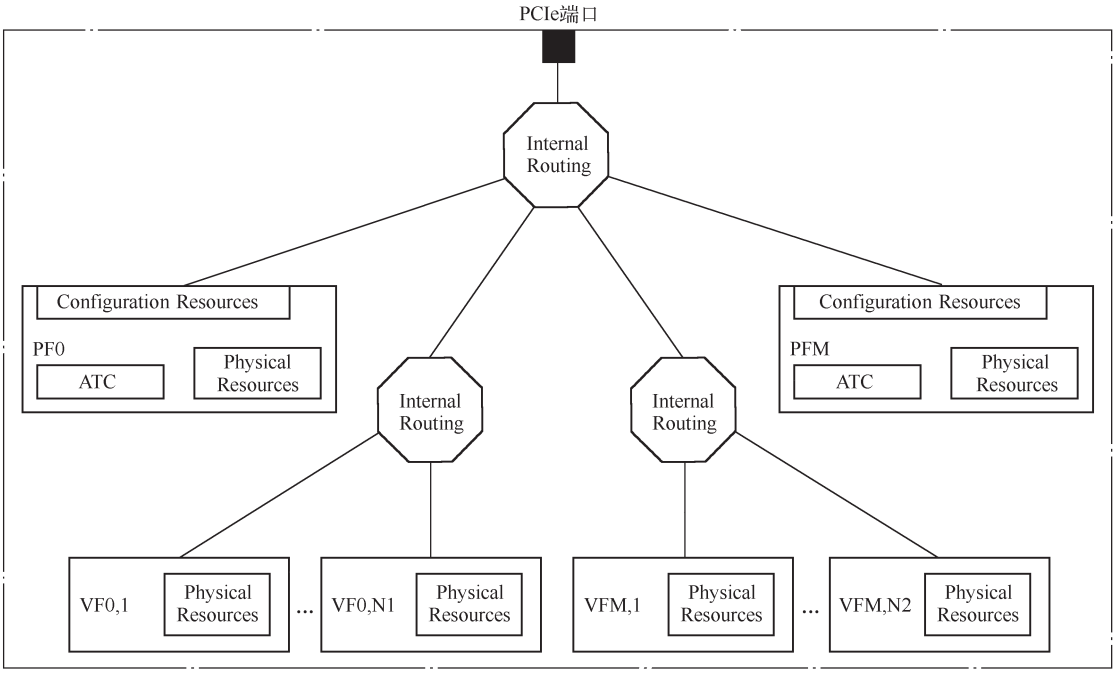

图 13-13　基于 SR-IOV 的 PCIe 设备

其中每个 PF 都有唯一的配置空间，而与 PF 对应的 VF 共享该配置空间，每一个 VF 都有独立的 BAR 空间，分别为 VF BAR0 ~ 5。从逻辑关系上看，这种做法相当于在一个 PF 中，存在多个虚拟设备。

在虚拟化环境中，每个虚拟机可以与一个 VF 绑定。假设在一个处理器系统中，网卡使用了 SR-IOV 技术，该网卡由一个 PF 和多个 VF 组成。其中每个虚拟机可以使用一个 VF，从而实现多个虚拟机使用一个物理网卡的目的。

PCIe 总线设置了 Single Root I/O Virtualization Extended Capabilities 结构以支持 SR-IOV 机制。对此感兴趣的读者可参阅 Single Root I/O Virtulization and Sharing Specification。这些细节对于非虚拟化技术的开发者并不重要。从本质上说，SR-IOV 技术与多线程处理器技术类似，只是多线程处理器技术应用于处理器领域，而 SR-IOV 将同样的概念应用于 PCIe 设备。

13.3.2　MR-IOV 技术

MR-IOV 技术的主要功能是将处理器系统的 PCI 总线域划分为多个虚拟 PCI 总线域，从而多个处理器系统可以共享同一个物理 PCI 总线域。MR-IOV 技术引入了几个新的概念，MRA RP、MRA Devices 和 MRA Switch。其中 MRA Switch 的结构如图 13-14 所示。

MRA Switch 与传统的 Switch 相比，其结构有较大不同。

- MRA Switch 由 0 个或者多个上游端口组成，如图 13-14 所示 MRA Switch 可以与多个 RP 连接，这个 RP 可以是 MRA RP 也可以是传统的 RP。在某些应用中，MRA Switch 可以作为中间节点与其他 MRA RP 相连，此时该 MRA Switch 不需要上游端口。
- MRA Switch 由 0 个或者多个下游端口组成，MRA Switch 可以与多个 MRA 设备连接，

也可以连接 SR-IOV 设备和传统的 PCIe 设备。MRA Switch 可以作为中间节点与其他 MRA Switch 相连，此时该 MRA Switch 可以不需要下游端口。

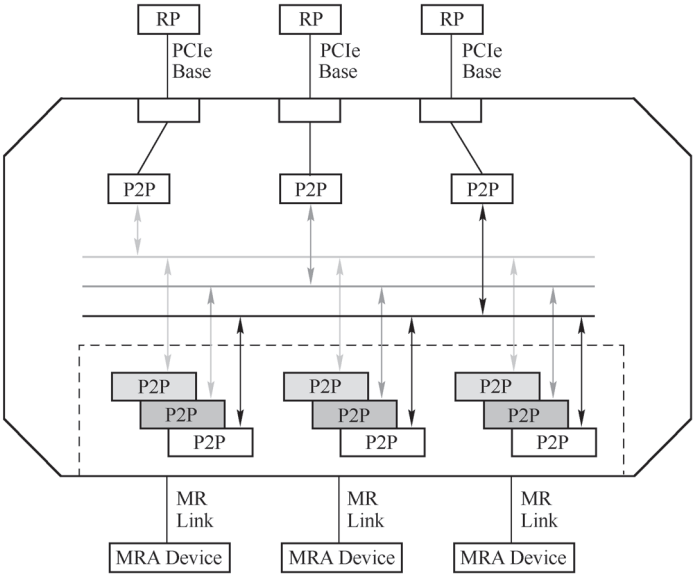

图 13-14　MRA Switch 的结构

- 使用 MRA Switch 可以组成多个虚拟 PCI 总线域 VHs（Virtual Hierarchies）。如图 13-14 所示，MRA Switch 由 3 套 P2P 桥组成，每一套 P2P 可以组成 1 个 PCI 总线域，这 3 个 PCI 总线域的地址空间独立。
- MRA Switch 可支持若干个双向端口，与其他 MRA Switch 和 MRA RP 相连。处理器系统之间可以使用这些双向端口组成复杂的服务器系统。

MRA Switch 的设计与实现较为复杂，而且该类芯片的应用范围较为有限，目前仅可能应用在支持虚拟化技术的高端服务器上。更为重要的是，Intel 并没有制作 Switch 芯片的传统，因此在短时间之内 MRA Switch 仅可能出现在 MR-IOV 规范中，而很难有实际的芯片。Intel 目前还没有支持 MRA RP 的 Chipset，但是可以预计 MRA RP 一定出现在 MRA Switch 之前，因而目前应该关注 MRA RP 与 MRA Devices 的连接拓扑结构。

MRA Devices 与 SR-IOV 设备相比略有不同，MRA Devices 略微更改了数据链路层。此外 MRA Devices 还重新定义了 SR-IOV 设备的 PF 和 VF，使得这些 PF 或者 VF 可以分属于不同的 PCI 总线域。

MRA Devices 可以与 MRA RP 或者 MRA Switch 联合使用，从而组成独立的 PCI 总线域。这些独立的 PCI 总线域可以与多个独立的虚拟机组成多个虚拟处理器系统。在这种结构下，不同的存储器域可以使用独立的 PCI 总线域，以最大限度地实现虚拟机对外部设备的隔离访问。MRA Device 的组成结构如图 13-15 所示。

在 MRA Device 中含有 1 个新的子设备 BF（Base Function），该设备存放管理 MRA Devices 的 MR-IOV 的 Capability 结构，该结构用来管理在 MRA Device 的 PCI 总线域。除此之外，在该结构中还可以存放与设备相关的寄存器，如网卡使用的 MAC 地址。BF 使用“BF 0：f”进行描述，其中“0”表示 BF 使用的 VH 号，而“f”表示 BF 使用的 Function 号，其

值在 0 ~ 255 之间。值得注意的是 BF 使用的 VH 号只能为 0。

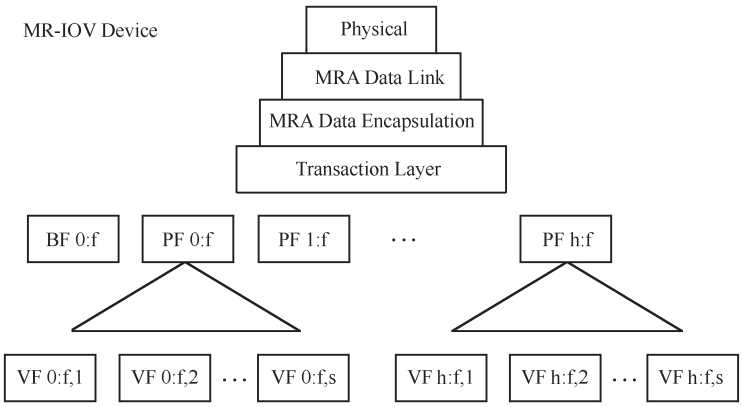

图 13-15　MRA Device 的结构

在 MRA Device 中，如果含有多个 PF 设备，这种 MRA Device 也被称为 SR-IOV/MRA Devices。这些 PF 设备使用“PF h：f”进行编码，其中“h”表示 PF 使用的 VH 号，而“f”表示 PF 使用的 Function 号，其值在 0 ~ 255 之间。

而与 SR-IOV 类似，在 MRA Device 中还存在多组 VF，其中每一组 VF 与一个 PF 对应，并使用“VF h：f，s”描述，其中“h”表示 VF 使用的 VH 号，而“f”表示 PF 使用的 Function 号，“s”表示 VF 号。

由以上描述可见，在 MRA Device 中，PF 和 VF 可以分别属于不同的虚拟 PCI 总线域 VH，并与 MRA RP 或者 MRA Switch 连接，形成一个完整的 PCI 总线域。为简便起见，本节仅介绍 MRA RP 与 SR-IOV 设备、SR-IOV/MRA Devices 和 MRA Devices 的连接关系，如图 13-16 所示。

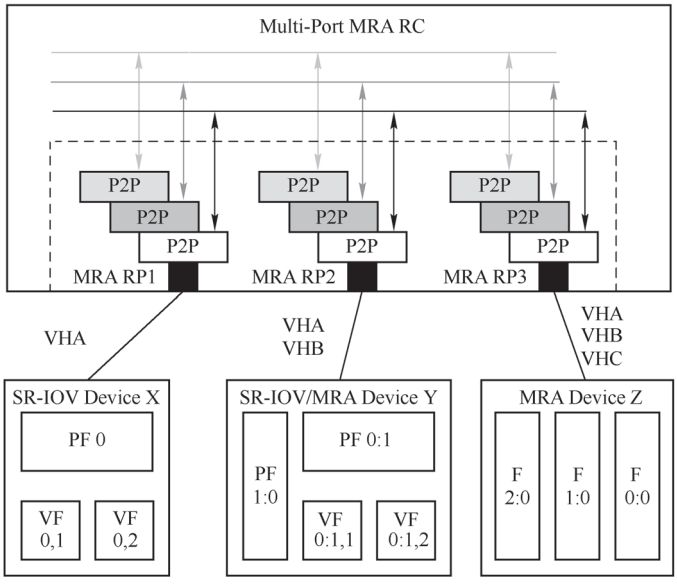

图 13-16　MRA RP 与 MRA Device 的连接

该图所示的 MRA RC 支持三个虚拟 PCI 总线域，分别为 VHA、VHB 和 VHC，并包含三

个 MRA RP，其中 RP1 与 SR-IOV Device X、RP2 与 SR-IOV/MRA Device Y、RP3 与 MRA Device Z 连接。

Device X 仅含有一个 PCI 总线域，因此只能指派给一个虚拟 PCI 总线域，假设该设备使用 VHA；Device Y 中含有 2 个 PCI 总线域，假设该设备的 VH1 与 VHB 对应，而 VH0 与 VHA 对应；Device Z 中含有 3 个 PCI 总线域，假设该设备的 VH2 与 VHC 对应，VH1 与 VHB 对应，而 VH0 与 VHA 对应。

由此可知，在当前处理器系统中，虚拟 PCI 总线域 VHA 中包含 Device X 中的全部子设备、Device Y 中的“PF0:1”、“VF 0:1，1”和“VF 0:1，2”和 Device Z 中的“F0:0”；虚拟 PCI 总线域 VHB 中包含 Device Y 的“PF1:0”和 Device Z 中的“F1:0”；而虚拟 PCI 总线域 VHC 中包含 Device Z 中的“F2:0”。

假设在处理器系统中含有 3 个虚拟机，这 3 个虚拟机可以分别使用 VHA、B 和 C 这 3 个不同的虚拟 PCI 总线域，从而实现对 PCI 设备的隔离访问。MR-IOV 技术的实现细节较为复杂，本节对此不做深入介绍。

通过以上描述，可以发现使用 MR-IOV 技术，通过为虚拟机提供独立的 PCI 总线域，较好地解决了虚拟机对外部设备的隔离访问。而且处理器系统还可以使用 MRA Switch 组成更为复杂的网络拓扑结构，从而便于实现基于多个 SMP 系统的虚拟机。但是目前尚无支持 MR-IOV 技术的 RP 和 Switch。

13.4 小结

本章简单介绍了 PCIe 总线与虚拟化技术相关的内容。读者需要获得与处理器相关的虚拟化知识后，才能进一步理解这些内容。囿于篇幅，本章没有进一步介绍虚拟化技术的实现细节。

第 II 篇的内容到此告一段落，在本篇中较为详细地介绍了 PCIe 总线的层次结构，流量控制机制，电源管理、序和死锁以及虚拟化技术等一系列内容。本篇的内容并不局限于 PCIe 总线本身，希望读者可以从本篇中了解通用总线的设计与实现过程，以及值得注意的实现细节。