Chapter 1 Background //背景

关于本章

为了建立理解PCI Express（PCIe）体系结构的基础，本章先回顾了先于PCIe总线出现的PCI（Peripheral Component Interface外设组件接口）总线模型，并介绍了PCI和PCI-X (PCI-eXtended) 的基本特征以及各自特点，接着讨论了从早期的并行总线模型迁移到PCIe使用的串行高速总线模型的出发点和动机。

关于下一章

下一章对PCIe体系结构进行了介绍，并打算作一个“操作执行级（executive level）”的概述，它覆盖了高层体系结构的所有基础内容。它介绍了协议规范中给出的分层的PCIe端口设计方法，并描述了每个层级的职责作用。

1.1 引言

想要理解PCIe的第一步，是要对PCIe所基于的先前的技术建立坚实的基础，而本章则将对它们的体系结构进行概述。已经对PCI比较熟悉的读者可以跳过本章去看下一章。如本章标题“背景”，本章仅意在给出一个简明的概述。对于关于PCI和PCI-X的更深更细节的内容，请参考MindShare的书籍《PCI System Architecture》以及《PCI-X System Architecture》。

这里可以通过举一个例子来说明为什么本章的背景介绍对理解PCIe有是帮助的，现在在PCIe上使用的软件（驱动）与当初在PCI上使用的软件大致相同。保持这种向后的兼容性，使得旧的设计迁移到新的设计（PCI到PCIe）时可以让软件的修改尽可能少、花费尽可能低的成本，这即是在鼓励就设计向新设计迁移。最终的结果就是，旧的支持PCI的软件在PCIe系统中的工作方式无需发生改变，而新的软件（PCIe的驱动软件）也将按照PCI一样的模型进行操作（例如对配置空间的操作）。基于这样的原因以及其他的一些原因，理解PCI以及它的操作模型将有助于促进对PCIe的理解。

1.2 PCI与PCI-X

PCI（Peripheral Component Interface外设组件接口）总线，其被开发出来的时间为1990年代初，当时人们期望用它来解决PCs（personal computer个人电脑）的外设总线的一些缺点。而那个年代的这个领域的技术标准是IBM的AT总线（Advanced Technology），它也被其他供应商称为ISA总线（Industry Standard Architecture）。ISA总线是为286 16位计算机而设计

的，它在286上也发挥了足够的性能，但随着新型的32位计算机及其外设们出现，新的需求也出现了：更高的带宽、更多更好的功能（例如即插即用）。除此之外，ISA总线使用的连接器是具有较多针脚数的大型连接器。PC供应商们认识到了需要做出一定的改变了，几种用于替代ISA总线的设计也被提出，例如IBM的MCA（Micro-Channel Architecture），EISA总线（Extended ISA，这是由IBM的竞争对手提出的），以及VESA总线（Video Electronics Standards Association，它由声卡供应商们为适配声卡而提出）。然而，所有的这些设计都具有一些使得它们无法被普及的缺点。最终，一个由PC市场的主要公司们共同联合建立的组织PCISIG（PCI Special Interest Group）开发出了PCI总线，并将其作为一种开放总线。在当时，PCI这种新型总线体系结构在性能上大大的优于ISA，而且它在每个设备内部新定义了一组寄存器，称之为配置空间（configuration space）。这些寄存器使得软件可以查看一个PCI设备内部所需的存储和IO资源，同时让软件为一个系统下的各个设备分配互不冲突的地址。这些特性：开放式设计、高速、软件可见性与可控性（software visibility and control），帮助PCI克服了限制ISA与其他总线的发展障碍，让PCI迅速地成为了PC中的标准外设总线。

几年之后，PCI-X（PCI eXtended）被开发出来，作为PCI体系结构的一种逻辑扩展（logical extension），并将总线性能提升了不少。我们将稍后再讨论PCI-X相较于PCI的改变，但是需提前一提的是PCI-X的一个主要设计目标就是保持与PCI设备的可兼容性，而且是软件和硬件的兼容性都要保证，这将使得从PCI迁移至PCI-X尽可能的简单。不久后，PCI-X 2.0版本将速率提升到了更高，原始数据速率达到了4GB/s。因为PCI-X保持了对PCI硬件上的向后兼容性，所以它依然是一个并行总线，并继承了与该总线模型相关的一些问题。有意思的事情来了，并行总线的有效带宽最终会达到一个实际的上限（paractical ceiling），然后就无法简单的继续提升。而PCISIG一直在探索如何将PCI-X的速率进一步提升，但最终这项努力还是被放弃了。这种速率上限，连同多针脚的连接方式，虽然是并行总线的缺点但是同时也提供了动力，推动了并行总线模型向新型的串行总线模型转型。

这些早期总线的定义被罗列在表 1-1，它展示了随时间发展而变得更高的带宽与频率。表中有一点很有趣的地方，那就是时钟频率和总线上插板卡的插槽数量之间的相关性。这是由PCI的低功耗信号模型带来的，这意味着更高的总线频率需要更短的板上走线以及更少的总线负载（详情可见“Reflected-Wave Signaling”）。另外有趣的一点是，随着总线频率的增加，共享总线上允许的设备数量在减少。当PCI-X 2.0被引入使用时，想要达到它的高速率需要要求总线变为一个点到点的互连（point-to-point interconnect）。

Table 1-1: Comparison of Bus Frequency, Bandwidth and Number of Slots

Bus Type	Clock Frequency	Peak Bandwidth 32-bit - 64-bit bus	Number of Card Slots per Bus
PCI	33 MHz	133 - 266 MB/s	4-5
PCI	66 MHz	266 - 533 MB/s	1-2
PCI-X 1.0	66 MHz	266 - 533 MB/s	4
PCI-X 1.0	133 MHz	533 - 1066 MB/s	1-2
PCI-X 2.0 (DDR)	133 MHz	1066 - 2132 MB/s	1 (point-to-point bus)
PCI-X 2.0 (QDR)	133 MHz	2132 - 4262 MB/s	1 (point-to-point bus)

表 1-1 总线频率、带宽、插槽数量的对比

1.3 PCI基础(PCI Basics)

1.3.1 基于PCI的系统的一些基础知识

图 1-1展示了一个基于PCI总线的老系统结构。该系统包括一个北桥（North Bridge称为“北”是因为如果将图看做一张地图，它位于中央PCI总线的北方向），北桥是处理器与PCI总线之间的接口。与北桥相连的是处理器总线（processor bus）、系统存储总线（system memory bus）、AGP图像总线（AGP graphics bus），以及PCI总线。几个设备将会共享PCI总线，它们要么直接与总线相连，要么作为插入板卡插入到连接器上。在中央PCI总线的“南方”有一个南桥（South Bridge），它用于将PCI与系统外设相连接，这里的系统外设可以是为了使用旧的遗留设备而应用发展了多年的ISA总线。南桥通常也是PCI的中心资源区（central resource），它提供了一些系统信号诸如复位、参考时钟和错误报告操作。

Figure 1-1: Legacy PCI Bus-Based Platform

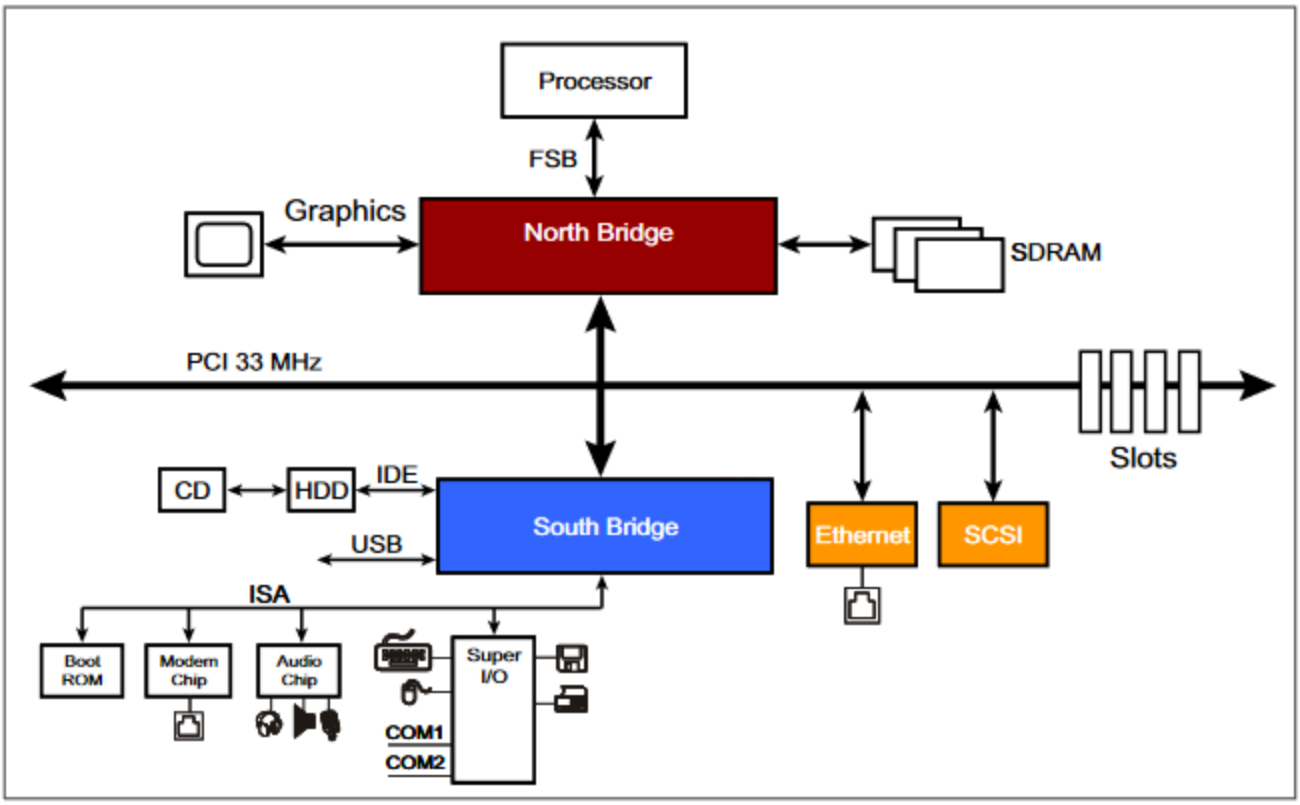

图 1-1基于旧PCI总线的平台

1.3.2 PCI总线发起方(Initiator)与目标方(Target)

在PCI层次结构中，总线上的每个设备（device）可以包含多达8个功能（function），这些功能都共享该设备的总线接口，功能编号为0到7（一个仅具有单功能的设备通常将被分配功能号0）。每个功能都能作为总线上事务（transaction）的目标方，而且它们中的大多数还可以作为事务的发起方。这样的发起方（称为总线主设备）有一对针脚（REQ#与GNT#，符号#表示他们为低有效信号），这一对信号用来处理共享PCI总线时的仲裁。如图 1-2所示，请求信号（REQ#）有效时表示主设备需要使用总线，并将此信息告知总线仲裁器来与此时所有的其他请求一起进行评估仲裁，最终将得出下一时刻由哪一个设备来占用PCI总线。仲裁器通常位于桥（bridge）中，桥是一种在层次结构中位于总线之上的结构，它可以接收总线上所有事务发起方（总线主设备）所发出的仲裁请求。仲裁器将决定谁下一个占用总线，并将这个设备的Grant（GNT#）置为有效（asserts the Grant pin for that device）。根据协议，只要总线上的上一个事务结束且总线进入空闲状态时，对于任意设备，只要在此时看到自己的GNT#信号被置为有效，那么就可以认为它是下一个占用总线的主设备，它就可以开始发起它的事务。

Figure 1-2: PCI Bus Arbitration

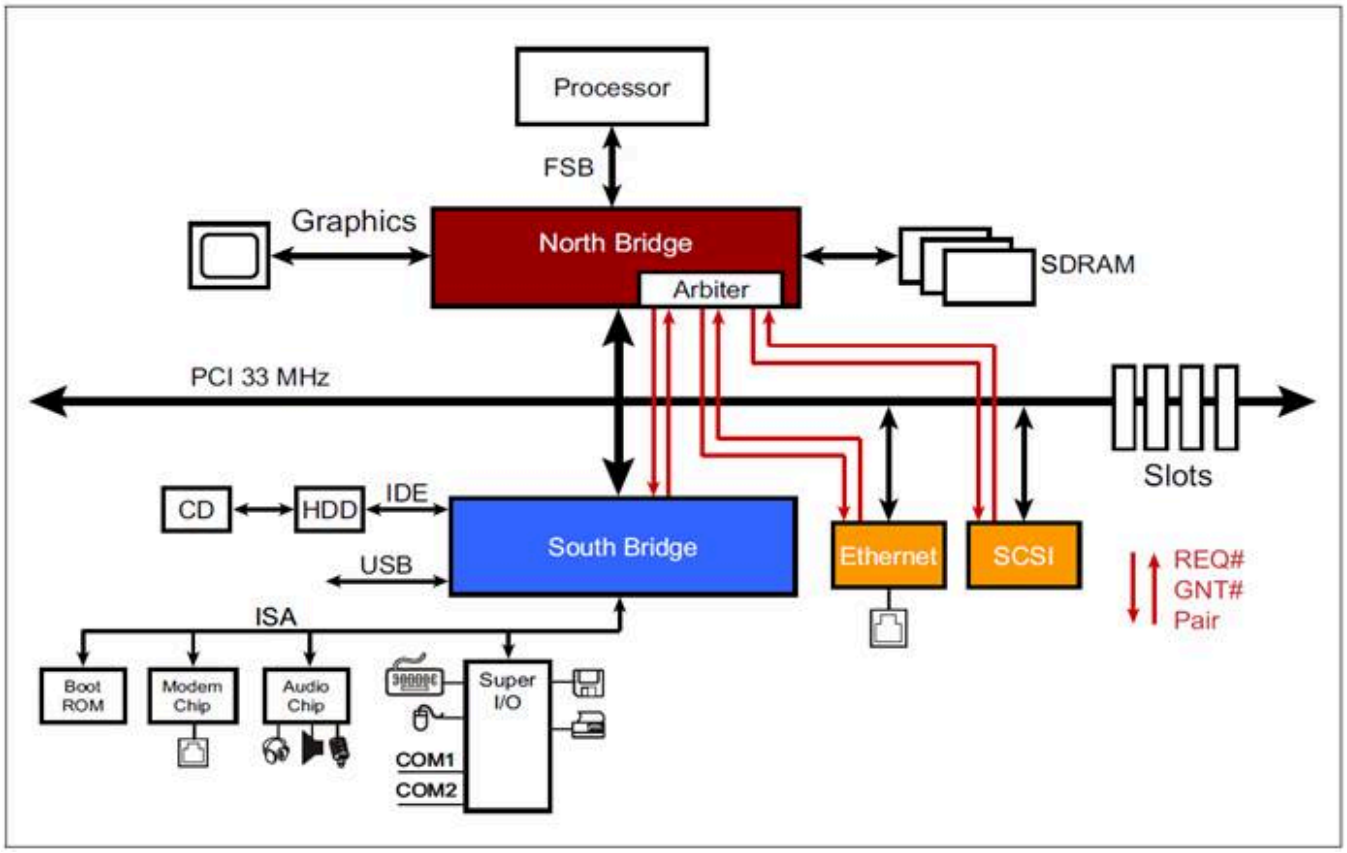

图 1-2 PCI总线仲裁

1.3.3 典型的PCI总线周期

图 1-3展示了一次典型的PCI总线周期。PCI总线是同步总线，这意味着总线上的活动都发生在时钟边沿，因此将时钟信号放在图标的顶部，并将它的上升沿都用虚线标注出来，这是因为信号正是在这些虚线所表示的时刻被发出（driven out）或被采样（sampled）。下面对总线上发生了什么进行一个简明的描述。

\1. 在时钟沿1，FRAME#信号（此信号有效时表示正在进行总线访问）和IRDY#信号（Initiator ReaDY for data，发起方准备好数据）这两个信号都处于无效状态，这表示总线处于空闲状态。与此同时，GNT#有效，这意味着总线仲裁器已经选择了该设备作为下一个事务发起方（Initiator）。

\2. 在时钟沿2，发起方将会把FRAME#信号置为有效，这表示一个新的事务已经被它发起。与此同时，它将把这个事务的地址与命令驱动到总线上。总线上其他所有的设备都将会把这些信息锁存起来，并将地址译码，查看自己的地址是否与之相匹配。

\3. 在时钟沿3，发起方将IRDY#信号置为有效，这表示它已经准备好了进行数据传输。图中AD

总线上的环形箭头符号表示这个三态总线（可以理解为inout型）正处于“转向周期”，这意味着这个信号的拥有者（三态总线上的信号此时是谁驱动的）发生了变化（这里出现“转向周期turn-around cycles”是因为这是一个读事务；发起者发出地址信息和接收数据的信号引脚是相同的）。打开目标方（Target）的缓冲buffer的时钟沿与关闭发起方缓冲buffer的时钟沿不能相同，因为我们想尽量避免两个 buffer 都在驱动同一个信号的可能性（即都往AD总线上驱动数据），哪怕只是很短的时间也不行。总线的争用将会损坏设备，因此，需要在前一个buffer关闭后的下一个时钟再打开一个新的 buffer 。在改变总线的驱动方向之前，每个共享信号都需要这样进行处理。

\4. 在时钟沿4，总线上的一个设备识别到了请求的地址与自己相匹配，于是便通过将DEVSEL#信号（device select）置为有效的方式对事务发出响应，声明参与到事务中。与此同时，它将把 TRDY# 信号（Target ReaDY）置为有效，这表示它（Target）正在发送读数据的第一部分，并将数据驱动到AD总线上（放到总线上的时间可以后延，总线允许目标方在FRAME#有效与TRDY#有效间间隔最多16个时钟周期）。由于此时 IRDY# 与 TRDY# 都处于有效状态了，数据将在此上升沿开始传输，至此完成了第一个数据阶段（first data phase）。Initiator 知道最终一共要传输多少字节的数据，但是 Target 并不知道。在事务的命令信息中并不包含字节数信息，因此每当完成一个数据阶段，Target 都必须查看 FRAME# 信号的状态，以此来得知 Initiator 是否对传输的数据量满意（即是否达到 Initiator 需要的数据量）。如果FRAME#依然有效，那么说明这并不是最后一个数据阶段，Target 仍然需要继续向 Initiator 传输数据，事务将按照相同的方式继续处理接下来传输的字节。

\5. 在时钟沿5，Target 还没有准备好继续发送下一组数据，因此它将 TRDY# 置为无效。我们将这种情况称为插入了一个“等待状态”，这使得事务被延迟了一个时钟周期。Initiator和Target都可以进行这样的操作，它们最多可以将下一次数据传输后延8个连续的时钟周期。

\6. 在时钟沿6，第二组数据被传输，并且由于传输后FRAME# 信号依然有效，Target就知道依然需要继续向Initiator传递数据。

\7. 在时钟沿7，Initiator强制总线进入了一个等待状态。在等待状态中，设备可以使当前事务暂停，并快速地将要发送的数据填充进buffer或是将接收到的缓存在buffer内的数据搬出，能进行这样的操作是因为Initiator和Target允许事务在暂停后直接恢复，而不需要进行中止和重启事务的操作。但是从另一方面来说，这样的操作将会十分低效，因为它不仅使当前事务停顿，同时也禁止其他设备访问和使用总线，尽管此时总线上没有数据在传输。

\8. 在时钟沿8，第三组数据被传输，并且此时FRAME#信号被置为无效，因此Target得知这就是最后一次数据传输。因此，在这一个时钟周期后，所有的控制信号线都关闭，总线再次进入了空闲状态。

在PCI总线上，许多信号都具有多种含义，这是为了减少引脚数量，以此来保持PCI设计中的低成本这一设计目标。数据和地址信号一起复用32bit总线，C/BE#（Command/Byte Enable）信号也共享他们的4个信号引脚。尽管减少引脚数量是一种可行的方法，并且它也是PCI使用“转向周期”这种会增加延时的方法的原因。但是这种复用引脚的做法阻止了对事务的流水线操作（在发送上一个周期的数据的同时发送下一个周期的地址）。握手信号诸如FRAME#、DEVSEL#、TRDY#、IRDY#以及STOP#被用来控制在事务（transaction）进行期间，各事件（event）何时进行。

Figure 1-3: Simple PCI Bus Transfer

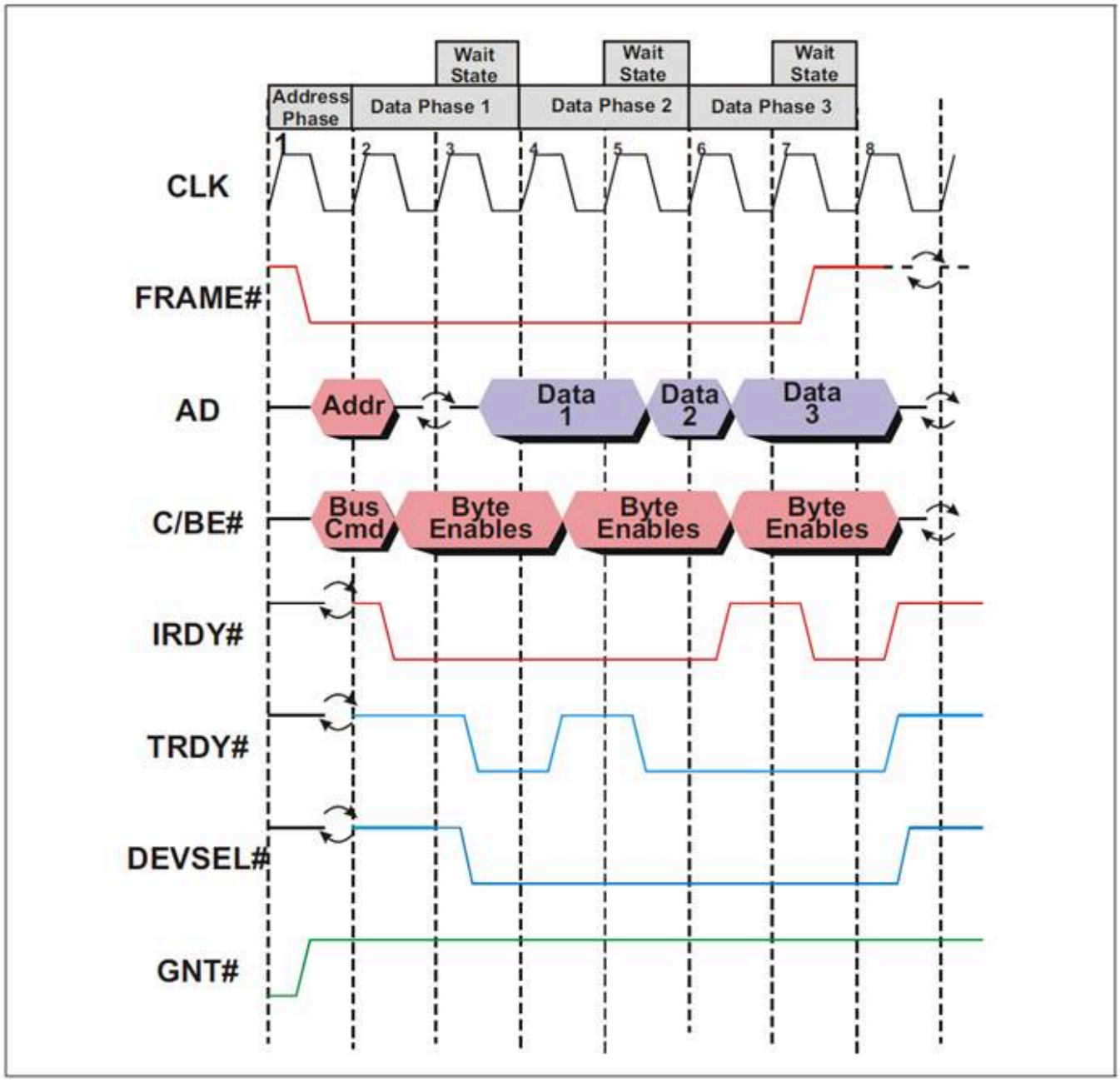

图 1-3简单的PCI总线传输

1.3.4 反射波信号传输(Reflected-Wave Signaling)

PCI体系结构支持每路总线上最多挂载32个设备，但是由于实际的电气上限要小得多，在33MHz的基频下大约可以支持10到12个电气负载。出现这种现象是由于总线使用了一种技术称为“反射波信号传输”，该技术可以降低总线上的功率损耗（如图 1-4）。在这个模型中，设备通过实现弱传输缓冲buffer（weak transmit buffer）来达到节省成本与功耗的目的，这种方法可以仅需一半的驱动电压即可完成信号电平的翻转。信号的入射波沿着传输线传输下去，直到到达传输线的末端。按照设计，传输线的末端没有终结电阻（termination）吸收回波，所以信号传输并没有就此中止，波阵面（wavefront）在遭遇传输线末端的无穷大阻抗后会被反射回来。这种反射自然是具有叠加性的，当信号返回发射方时，它会与入射信号叠加使信号增加到全电压电平。当这样的反射信号到达初始的buffer时，buffer驱动器的低输出阻抗会中止这个信号的传输以及停止其继续反射。因此，从buffer发出一个信号一直到接收方检测到有效的信号所花费的时间，就等于信号沿线路传输的时间加上反射回来的延迟时间和建立时间。所有这些都必须小于一个时钟周期。

Figure 1-4: PCI Reflected-Wave Signaling

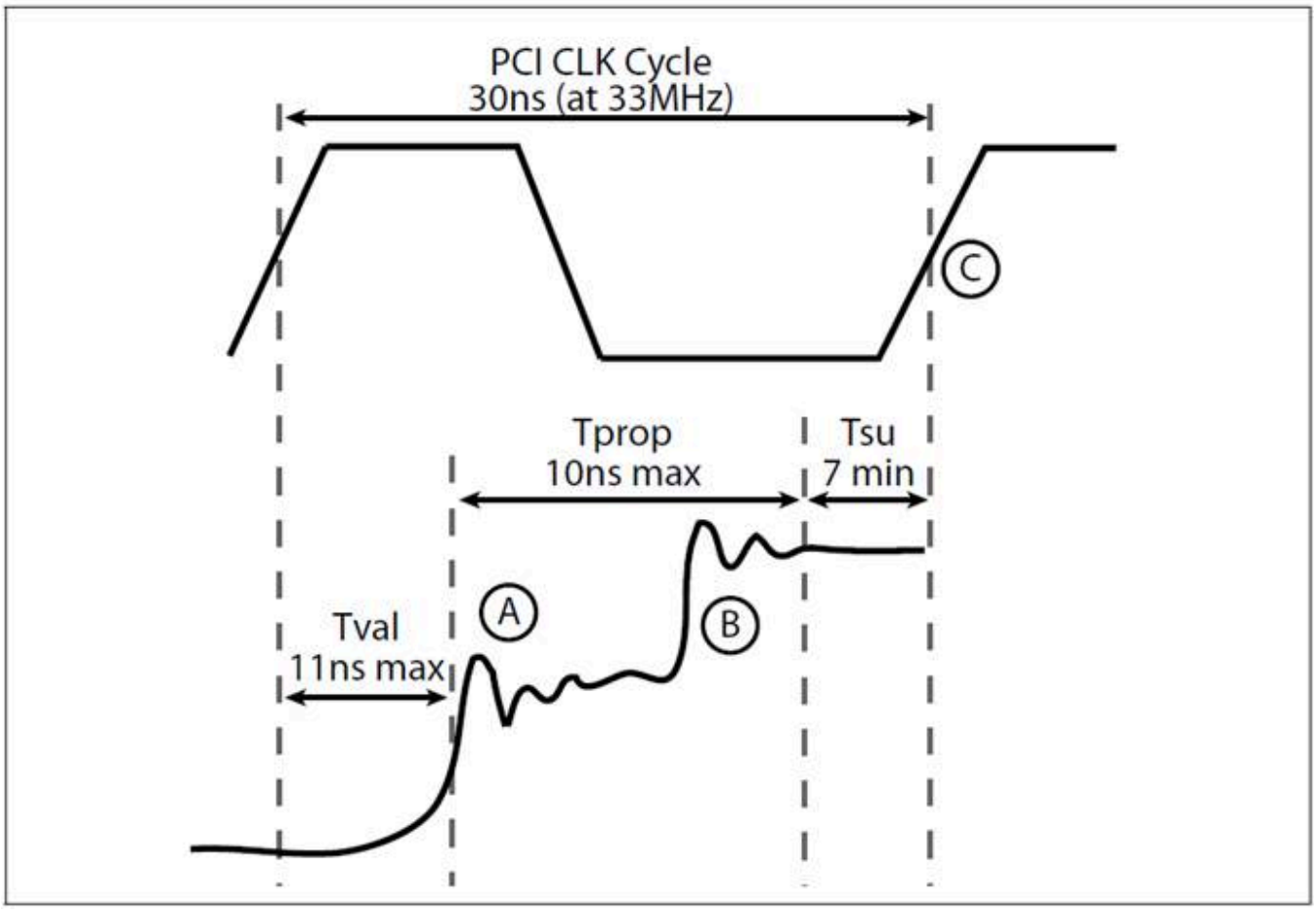

图 1-4 PCI的反射波信号传输

当走线长度和总线上的电气负载数量增加时，信号往返所需要的时间也增加。一个33MHz的PCI总线只能满足最多10-12个电气负载的信号时序要求。一个电气负载指的是系统板上安装的一个设备，但是实际上一个插了板卡的连接器插槽被算作两个电气负载。因此，如表 1-1所示的那样，一个33MHz的PCI总线最多只能支持至多4个或者5个插入板卡型设备，否则将无法保证可靠性。

为了在一个系统中接入更多的负载，需要使用PCI-to-PCI bridge，如图 1-5所示。当出现了越来越多更为先进的主芯片组后，外设的发展也十分迅速，以至于竞争访问共享PCI总线变成了限制外设性能的原因。PCI总线的速率没有跟上外设速率增长的脚步，虽然它依旧是主流的外设总线，但是已经成为了系统的性能瓶颈。这个问题的解决办法为，把PCI移到系统外设与存储器之间的主路径之外，将芯片组互连用一些专有的解决方案来替代（例如Intel的Hub Link Interface）。

PCI Bridge是拓扑结构的延伸。每个Bridge都能产生新的PCI总线，而且这种由PCI Bridge产生的PCI总线与上层PCI总线在电气上是相隔绝的，因此它也可以独立的提供额外的10-12个电气负载。PCI总线上的一些设备也可以作为Bridge，可以令较多的设备接入系统中。PCI体系结构允许在单系统中存在256路总线，每路总线下最多可以挂载32个设备。

Figure 1-5: 33 MHz PCI System, Including a PCI-to-PCI Bridge

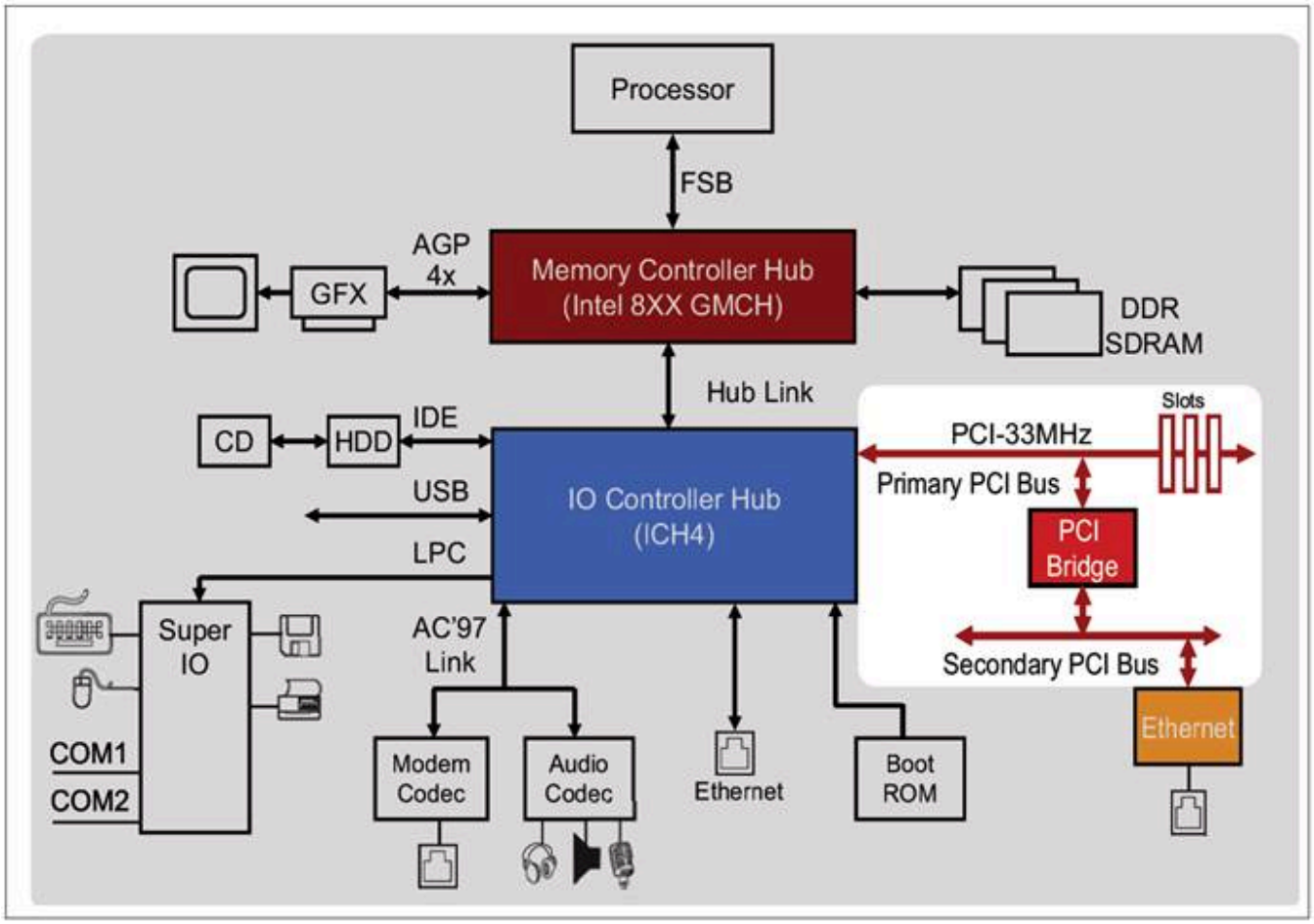

图 1-5 包含PCI-to-PCI bridge的33MHz PCI系统

1.4 PCI总线体系结构透视探究(PCI Bus Architecture Perspective)

1.4.1 PCI事务模型

PCI同先前的总线模型一样，在数据传输上使用三种模型：Programmed I/O（PIO）、Peer-to-peer、以及DMA。这些模型的图解如图 1-6所示，接下来的几个小节将对它们进行描述。

1.4.1.1 Programmed I/O(PIO)

PIO在早期PC中被广泛使用，因为设计者不愿意增加设备中由于事务管理逻辑而带来的成本开销，以及额外增加的复杂度。在当时，处理器比任何其他设备工作的都要快，所以在这个模型中，处理器包揽了所有的工作。举例来说，当一个PCI设备向CPU发出中断，并表示它需要向memory中放入数据，则由CPU最终将数据从PCI设备中读出并放入一个内部寄存器中，然后再将这个内部寄存器的值复制进memory中。反之，如果数据要从memory移动到PCI设备中，那

么软件会指示CPU将memory中的数据读出至内部寄存器中，然后再将内部寄存器的值写入到PCI设备中去。

这样的操作流程虽然可行，但是效率却比较低，其主要有两个原因。第一个原因，每一次数据传输都需要CPU开销两个总线周期。第二个原因，在数据传输过程中CPU都要忙于数据传输而不能做其他更有意义的任务。在早期，这是一种最快速的传输方式，而且单任务处理器也没有其他的任务要做。然而这种低效的方式显然不适用于更为先进的系统中，因此这种方式已不再是数据传输的常用方式，取而代之首选方法的是下一节将讲述的DMA方法。然而为了使软件与设备交互，Programmed IO仍然是一种必要的事务模型。

Figure 1-6: PCI Transaction Models

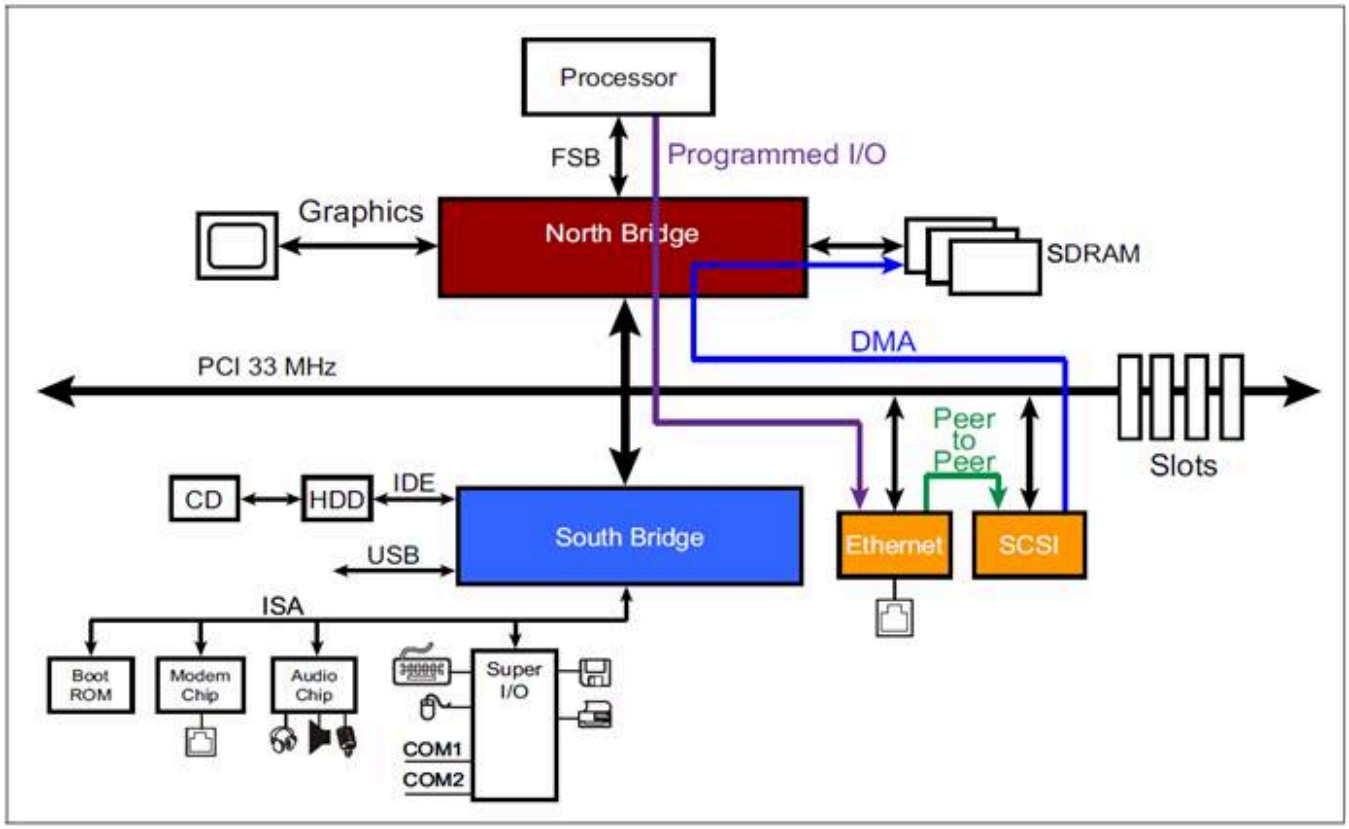

图 1-6 PCI事务模型

1.4.1.2 Direct Memory Access(DMA)

在传输数据的方法上，一种更为高效的方式叫做 DMA（Direct Memory Access，直接内存访问）。在这个模型中有另一种设备称为DMA Engine，由它来代表处理器来负责掌握从memory 向外设进行数据传输的各种细节操作，这就将这个繁琐的任务从CPU中卸载了下来。一旦CPU将起始地址（starting address）和数据字节总量（byte count）写入DMA Engine，DMA Engine将会自动处理总线协议与地址序列。这不涉及 PCI 外设的任何更改，并允许其自

身保持低成本设计。后来，经过集成性改进，外设自身可以集成入这种DMA功能，这样它们就不再需要一个外部的DMA Engine。这些设备有能力处理自己发起的总线传输，我们将它们称为总线主设备（Bus Master device）。

图 1-3 就是一个PCI总线上的总线主设备正在进行事务的过程。北桥可以对地址进行译码，以此来识别自己是否是这个事务的Target，比如北桥译码后发现地址与自己相匹配则认为自己是这个事务的Target。在总线周期中的数据传输阶段，数据在总线主设备与北桥之间传输，北桥即是数据传输的Target。北桥继而根据请求的事务内容，发起 DRAM 读写操作，与系统内存进行数据通信。在数据传输完成后，PCI外设可以产生一个中断来通知系统。DMA提升了数据传输效率，因为这种方式在搬移数据时不需要CPU的参与，那么对于CPU来说，只需要一个总线周期的开销，将起始地址和总数据量写入DMA Engine，即可完成高效的数据块搬移（move a block of data）。

1.4.1.3 Peer-to-Peer(点对点)

如果一个设备能够作为总线主设备，那么它就提供了一个有趣的作用。一个PCI总线Master可以发起针对其他PCI设备的数据传输，而对于PCI总线自身来说这整个事务都是在本地进行的，并未引入任何其他系统资源。因为这个事务是在总线上的两个设备之间进行的，且这两个设备被认为是总线中两个对等的节点，因此这个事务被称作一个“点对点”的事务。显然，这种事务非常高效，因为系统中其余的部分仍然可以自由的完成其他的任务。然而，在实际场景中点对点事务很少被使用，这是因为Initiator和Target通常不会使用相同的数据格式，除非二者都是由一个供应商制造的。因此，数据一般必须要首先发送到memory，在那里由CPU在数据传输到Target之前，对其进行格式转换，而这就阻碍并破坏了点对点传输的设计目标。

1.4.2 PCI总线仲裁

由图 1-2 可知，当今的PCI设备基本都能作为总线主设备（Bus Master device），所以它们都可以进行DMA与peer-to-peer的数据传输。在像PCI这种共享总线的体系结构中，各设备需要轮流占用总线，因此当一个设备想要发起事务时必须首先向总线仲裁器请求总线所有权（ownership）。仲裁器将查看当前所有的请求，并使用一些特定的仲裁实现算法来决定哪个Master可以下一个占用总线。PCI协议规范并没有描述这个仲裁算法，但是有声明这个仲裁必须是“公平”的，不得在访问中差别对待任何一个设备。

仲裁器可以在上一个占用总线的Master还正在进行数据传输时就决定出下一个占用总线的设备，这样总线上就不需要引入额外的时延来对下一个总线所有者进行排序。因此，总线仲裁器的仲裁作用发挥在“幕后”，被称为“隐藏”的总线仲裁，这是一种对早期总线协议的改进。

1.4.3 PCI低效的地方(PCI Inefficiencies)

1.4.3.1 PCI重试协议(PCI Retry Protocol)

当一个PCI Master发起一个事务，用于访问一个Target设备，而此时Target设备并未处于Ready状态，那么Target就会给出一个事务重试的信号，这种场景的示意图如图 1-7。

Figure 1-7: PCI Transaction Retry Mechanism

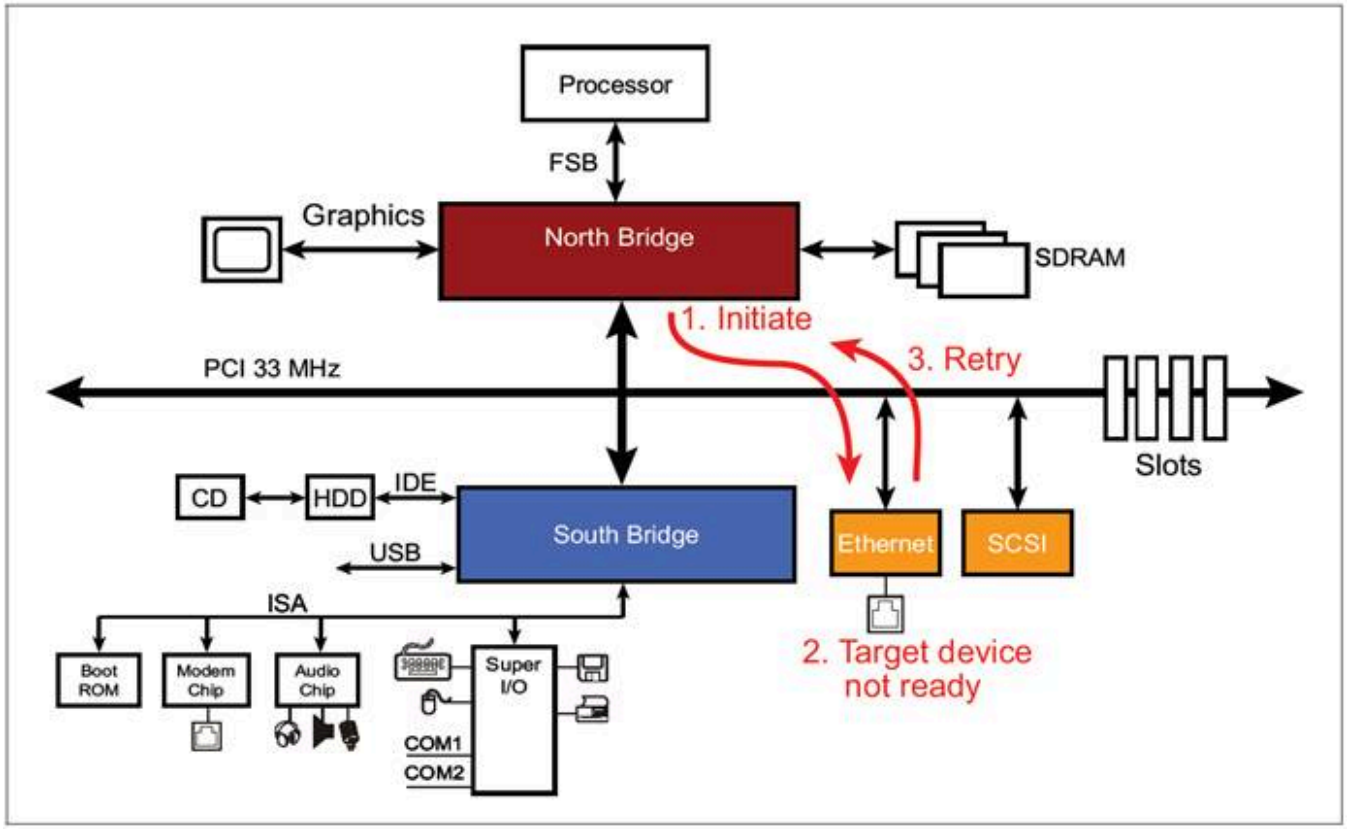

图 1-7 PCI事务重试机制

考虑接下来给出的例子：北桥发起一个读memory事务，希望从以太网设备中读取数据。这个以太网设备Target响应这个事务，请求并参与到总线周期中来。然而，这个以太网设备Target此刻并没有立即将数据返回给北桥Master。这个以太网设备此时有两个办法来推迟这次数据传输。第一个办法就是在数据传输中插入等待态（wait-state）。如果过程中仅需插入少量的等待态，那么数据的传输还能算是比较有效率。但是如果Target设备需要延迟更多的时间（从事务被发起后延迟16个时钟周期以上），那么就需要用第二种办法了，第二种办法就是Target发出一个STOP#信号来表示事务重试（retry）。Retry操作是通知Master在数据没有传输前就提前结束总线周期。这样做可以防止总线长时间处于等待状态而降低了总线效率。当Master接收到Target发来的Retry请求，Master至少等待2个时钟周期，然后必须重新向总线发起仲裁请求，当再次获得总线的使用权之后重新发起相同的总线周期。在当前总线Master正在处理Retry的这

段时间中，仲裁器可以将总线使用权授予其他的Master，以便于PCI总线能被更有效率的利用起来。当此前执行Retry的Master再次占用总线并重新启动与此前相同的总线周期时，Target可能也已经准备好了需要传输的数据，并参与到总线周期中进行数据传输。若Target仍然未准备好，那么它将会再次发起Retry，按照这样的流程循环下去直到Master成功完成了数据传输。

1.4.3.2 PCI断开连接协议(PCI Disconnect Protocol)

当一个PCI Master发起一个事务来访问一个Target，并且如果这个Target可以传输至少一个双字（doubleword，后面使用dw简写）的数据，但是它无法完成整个完整的全数据量的传输，那么它将会在它无法继续进行传输时断开与事务操作的连接。这种场景的示意图如图 1-8所示。

考虑接下来给出的例子：北桥发起一个突发读memory事务（burst memory read），希望从以太网设备中读取数据。以太网Target设备响应了请求并参与到这个总线周期中来，传输了部分数据，但是随后不久把已有的所有数据都发送光了，却依然没有达到Master需要的数据总量。以太网设备Target此时有两个选择来延迟数据的传输。第一个选择是在数据传输阶段插入等待态，然后自身也同时在等待收到新增的数据。如果过程中仅需插入少量的等待态，那么数据的传输还能算是比较有效率。但是如果Target设备需要延迟更多的时间（PCI协议规范中允许在数据传输中途最多有8个时钟周期的等待态，这里区别于上一节Retry中的16个周期，那是因为那时传输还没开始，而这8个周期针对的是传输已经开始了一段时间的传输中途等待），Target必须发出断开连接的信号。要断开连接，Target需要在总线周期运行中将STOP#置为有效，这样就能告知Master提前结束总线周期。Disconnect与Retry的一个区别就在于，Disconnect有数据已经被传输，而Retry则是数据传输根本就没开始。Disconnect这种操作也让总线避免长时间处于等待态。Master至少等待2个时钟周期，然后才能再次向总线发起仲裁请求，当再次获得总线的使用权之后就可以再次访问刚才断开连接的设备地址，继续完成此前断开未完成的总线周期。在当前总线Master正在经历Disconnect的这段时间中，仲裁器可以将总线使用权授予其他的Master，以便于PCI总线能被更有效率的利用起来。当此前经历Disconnect的Master再次占用总线并准备继续此前未完成的总线周期时，Target可能也已经准备好了需要继续传输的数据，并参与到总线周期中继续完成数据传输。否则，若Target仍然未准备好，那么它将会再次发起Retry或者Disconnect，然后按照这样的流程循环下去直到Master成功完成了所有数据的传输。

Figure 1-8: PCI Transaction Disconnect Mechanism

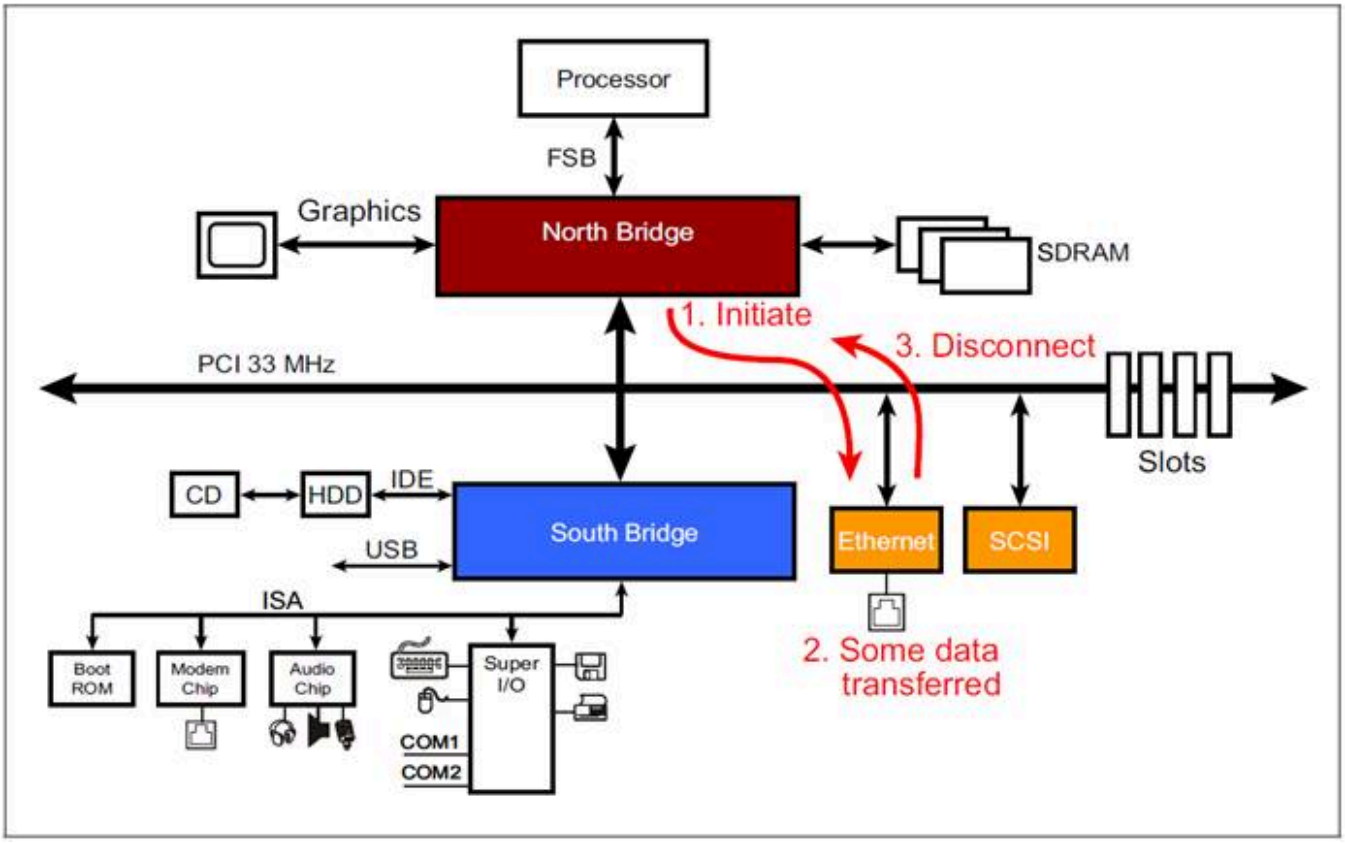

图 1-8 PCI事务断开连接机制

1.4.4 PCI中断处理(PCI Interrupt Handling)

PCI设备使用4个边带信号（sideband）作为中断信号，分别为INTA#、INTB#、INTC#、INTD#，并从中选取一个来向系统发送中断请求，即使用4个中断信号中的1个来发送中断请求。当其中一个中断引脚被置为有效时，单CPU系统的中断控制器将会对中断作出响应，相应的方式为将INTR（interrupt request）信号置为有效，将中断请求发送给CPU。后来出现的多CPU系统不再适用这种单信号线输入作为中断的方式，因此进行了改进，将中断改为了APIC（Advanced Programmable Interrupt Controller）模型，在这种模型中中断控制器将会向多CPU发送报文（message）而不是向其中一个CPU发送INTR信号。不过无论中断传输模型是什么，接收到中断的CPU都必须确认中断来源，然后为中断提供服务。传统的模型需要好几个总线周期来完成确认中断和服务中断的操作，效率不是很高。APIC模型会比传统模型好一些但是也依然存在改进的空间。

1.4.5 PCI错误处理(PCI Error Handling)

可选地，PCI设备可以在事务进行期间，检测并报告接收地址或者数据信号中的奇偶校验错误。

在事务进行期间，PCI对总线上大部分信号进行奇偶校验，并通过PAR信号表示计算的偶校验位结果。如果传输的数据或者地址信息中为逻辑电平1的比特数量是奇数，那么就将PAR信号置为1，以此来使得“电平1”的比特数量为偶数个。Target设备在接收到数据或地址时将会进行校验检查错误。奇偶校验（在这里为偶校验）的方法仅在有奇数个信号出错时才能检测到。如果设备检测到数据的奇偶校验错误，它将把PERR#（parity error）置为有效。这有可能是一个可以恢复的错误，因为例如对于memory读取来说，只需要再次发起相同的事务就可以解决问题。然而，PCI自身并不包含任何的自动的或是基于硬件的错误恢复机制，因此软件将完全负责去进行错误处理。

Figure 1-9: PCI Error Handling

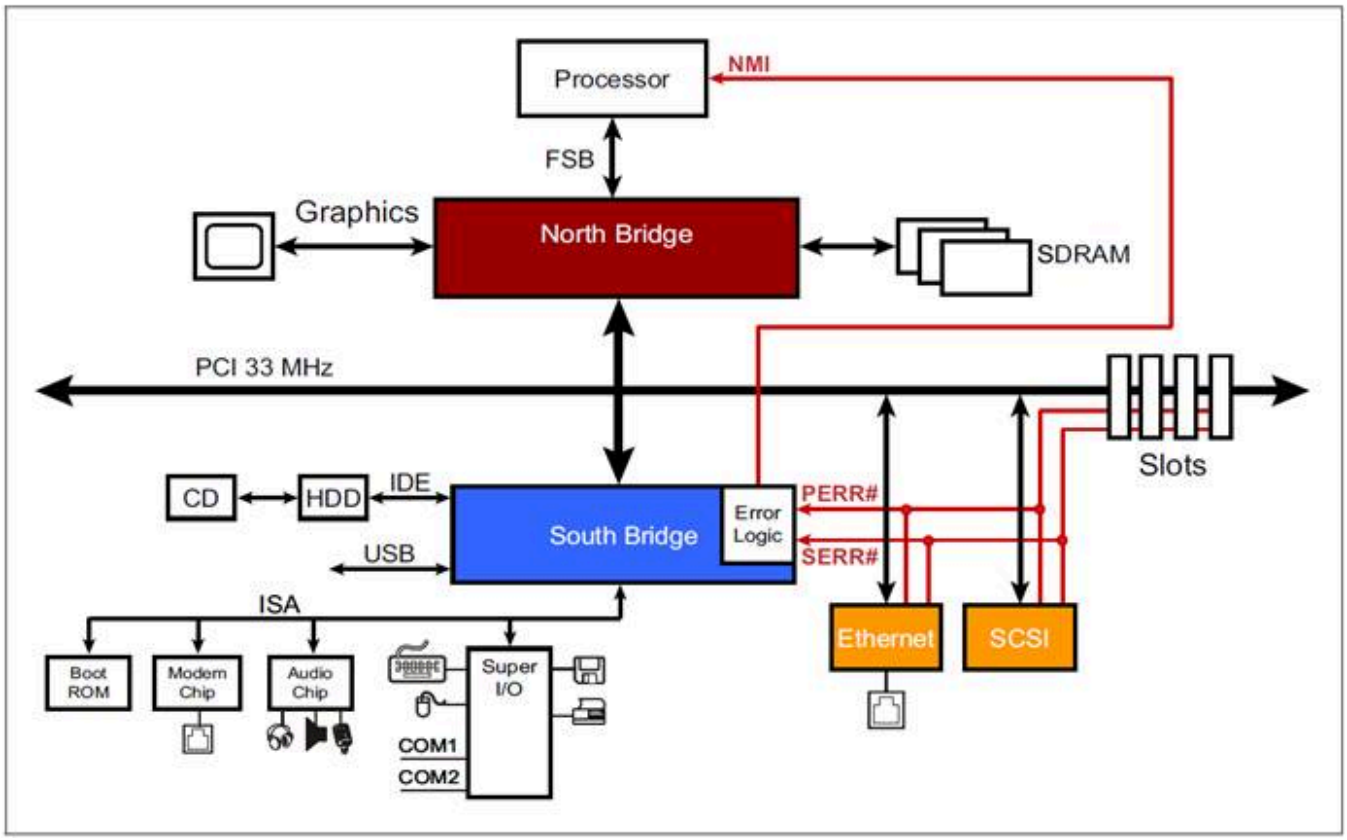

图 1-9 PCI错误处理

然而，上面所说的是数据出错的情况，若地址校验出错则情况不同。在图 1-9的例子中，地址信息损坏，这导致一个Target匹配上了这个错误的地址。我们无法得知损坏的地址信息变成了什么，也无法得知总线上哪个设备匹配上了这个错误的地址，所以对于这种情况就不存在能够简单进行错误恢复的方法。因此，这种类型的错误将会导致SERR#（system error）被置为有效，之后系统通常会调用错误处理程序。在老式机器中，为了预防引起更进一步的错误，将会强行让系统停止工作，从而导致出现“蓝屏死机”。

1.4.6 PCI地址空间映射(PCI Address Map)

PCI体系结构支持3种地址空间，如图 1-10所示，包含：Memory（内存）、I/O、Configuration Address Space（配置地址空间）。x86处理器可以直接访问Memory和I/O空间。一个PCI设备映射到处理器内存地址空间，可以支持32或64位寻址。在I/O地址空间，PCI设备可以支持32位寻址，但是因为x86 CPU仅使用16位的I/O地址空间，所以许多平台都会将I/O空间限制在64KB（对应16bit）。

Figure 1-10: Address Space Mapping

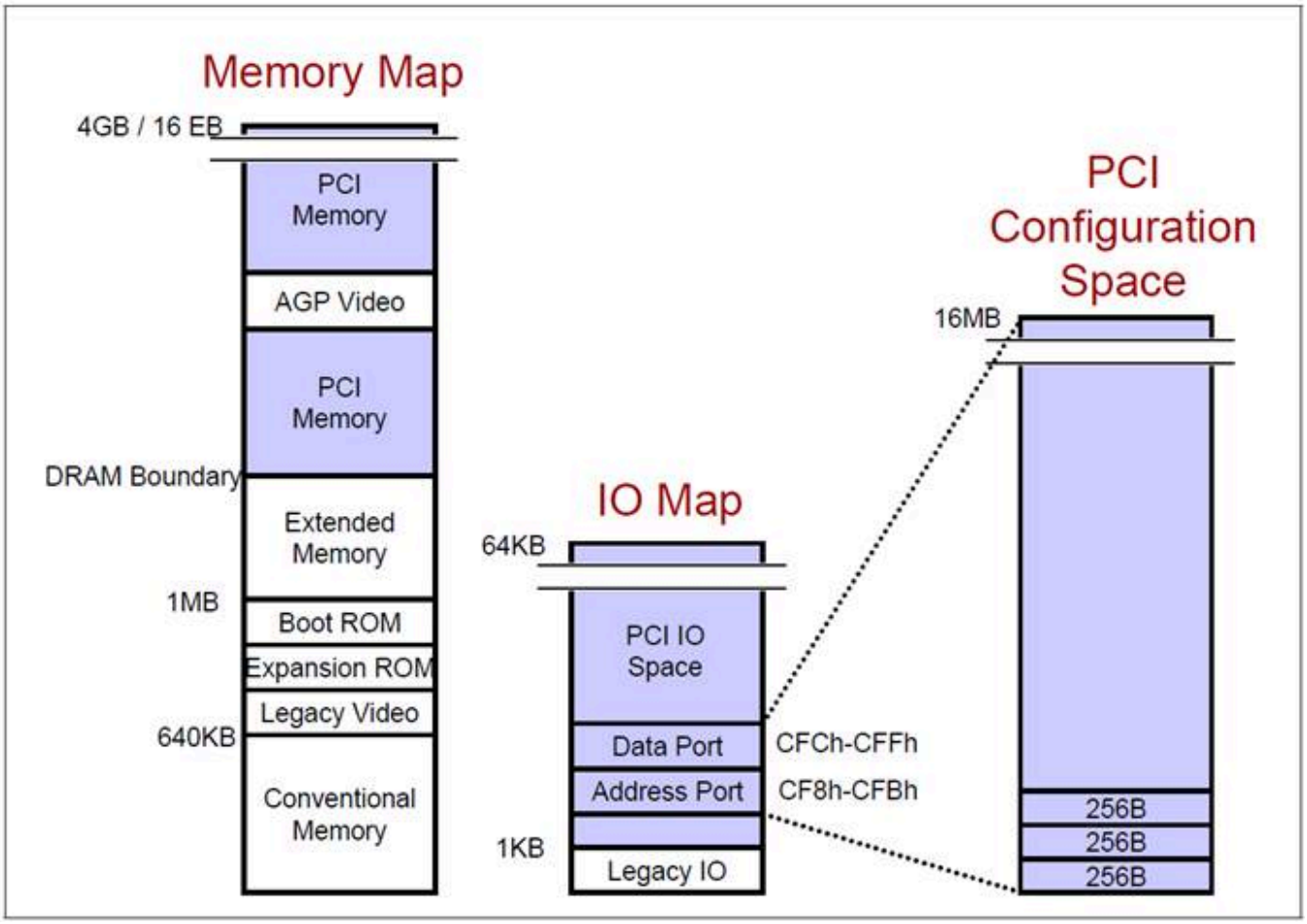

图 1-10地址空间映射

PCI还引进了第三种地址空间，称为配置空间，CPU只能对它进行间接访问而非直接访问。PCI设备中的每个Function（功能）都包含专为配置空间所准备的内部寄存器，这些寄存器对于软件来说是可见的，并且软件可以通过一个标准化的方式来控制它们的地址与资源，这为PC提供了一个真正的即插即用（plug and play）的环境。每个PCI Function最多可以有256 Bytes的配置地址空间。考虑到PCI最多可支持单个设备含有8个Function、每路总线含有32个设备、单个系统包含256路总线，那么可以得到一个系统的配置空间总量为：

256B/Function *8Function/device * 32devices/bus *256buses/system =16MB，

即一个系统的配置空间总大小为16MB。

因为x86 CPU无法直接访问配置空间，所以它必须通过IO寄存器进行索引（然而在PCI Express中引入了一种新方法来访问配置空间，这种新方法是通过将配置空间映射入内存地址空间来完成的）。在传统模型中，如图 1-10所示，使用了一种被称为配置地址端口（Configuration Address Port）的IO端口，它位于地址CF8h-CFBh；还使用了一种被称为配置数据端口（Configuration Data Port）的IO端口，它位于地址CFCh-CFFh。关于通过这种方法以及通过内存映射方法访问配置空间的细节将会在下一节进行解释。

1.4.7 PCI配置周期的生成(PCI Configuration Cycle Generation)

由于IO地址空间的大小是有限的，传统模型的设计上对地址十分保守。在IO空间中常见的做法是令一个寄存器来指向一个内部位置，然后用另一个寄存器来进行数据的读取或写入。PCI的配置过程包含如下两步。

l 第一步：CPU产生一个IO写，写入的位置为北桥中IO地址为CF8h的地址端口（Address Port），这样就给出了需要被配置的寄存器的地址，即“用一个寄存器来指向一个内部位置”。如图 1-11所示，这个地址主要由三部分组成，通过这三部分就可以定位一个PCI Function在拓扑结构中的位置，它们为：在256条总线中我们想访问哪一条总线、在该总线上的32个设备中访问哪一个、在该设备的8个Function中访问哪一个。除了这些以外，唯一还需要提供的信息是要确认访问这个Function的64dw（256Bytes）中的哪个dw。

l 第二步：CPU产生一个IO读或者IO写，操作的位置为北桥中的地址为CFCh的数据端口（Data Port），即“用另一个寄存器来进行数据的读取或写入”。在此基础上，北桥向PCI总线发起一个配置读事务（configuration read）或配置写事务（configuration write），事务要操作的地址即为步骤一中地址端口中所指定的地址。

Figure 1-11: Configuration Address Register

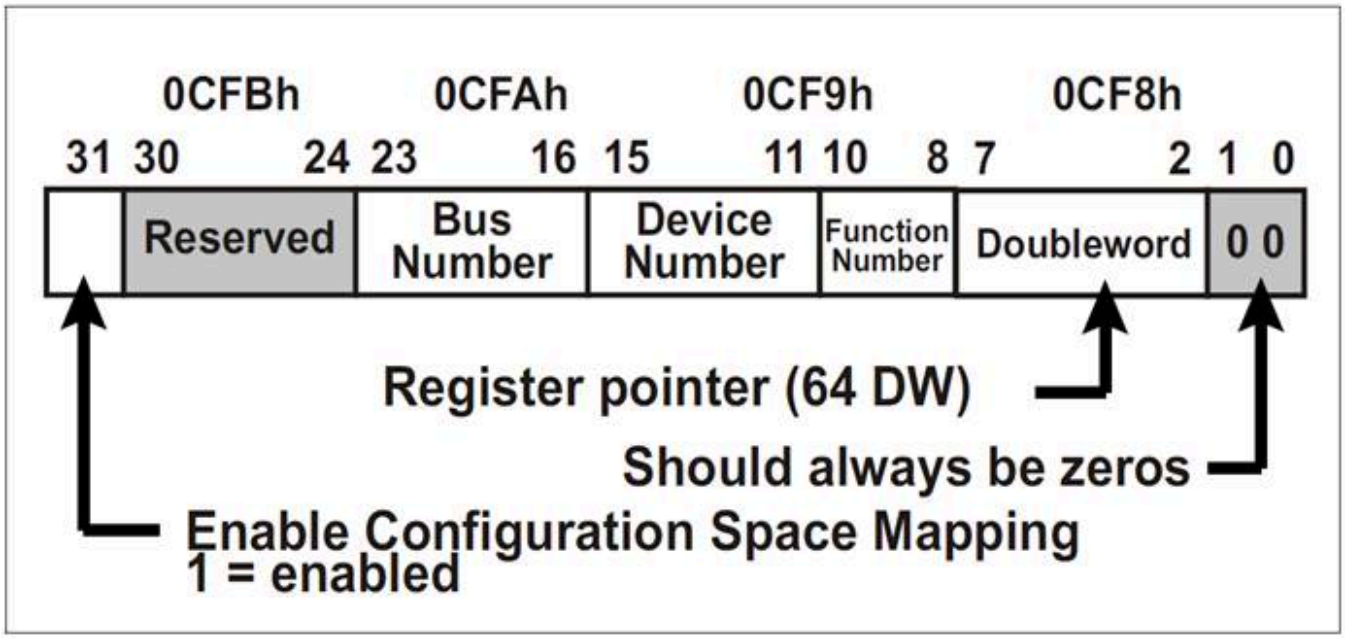

图 1-11配置地址寄存器

1.4.8 PCI Function配置寄存器空间(PCI Function Cfg Reg Space)

每个PCI Function可以包含最多256Bytes的配置空间。这个配置空间起始的64Bytes包含一个被称为Header（配置空间头）的结构，剩余的192Bytes用来支持一些其他可选功能。系统配置首先由Boot ROM固件来执行，在操作系统被加载完成后，它将重新对系统进行配置，重新进行资源分配。这也就是说系统配置的过程可能会被执行两次。

根据Header的类型，PCI Function被分为两个基本的类别。第一种Header称为Type 1 Header（类型1），它的结构如图 1-12，它用于标识这个Function是一个Bridge，Bridge将会在拓扑结构上创建另一条总线。而Type 0 Header（类型0）就是用来指示这个Function**不是**一个Bridge（如图 1-13）。关于Header类型的信息包含在dword3的字节2的同名字段中（Class Code字段），当软件在系统中发现某个Function时，第一件事就是要检查其Header的这个字段（软件发现系统中Function的过程称为枚举enumeration）。

关于配置寄存器空间以及枚举过程的更为详细的讲解将会稍后再进行。在这里我们只是希望你先熟悉一下各个部分是如何相互配合工作的。

Figure 1-12: PCI Configuration Header Type 1 (Bridge)

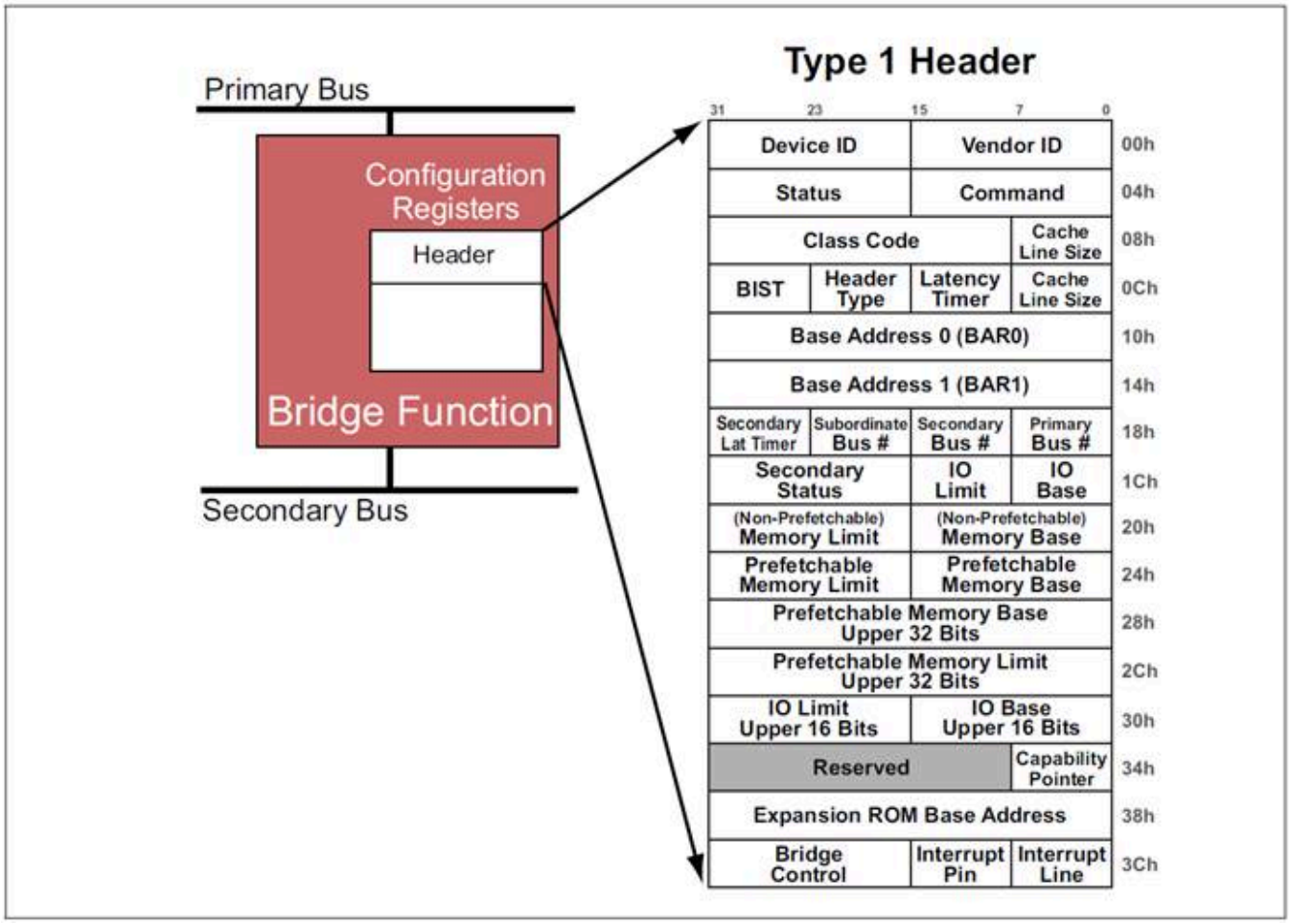

图 1-12 PCI配置Header Type 1（Bridge）

Figure 1-13: PCI Configuration Header Type 0 (not a Bridge)

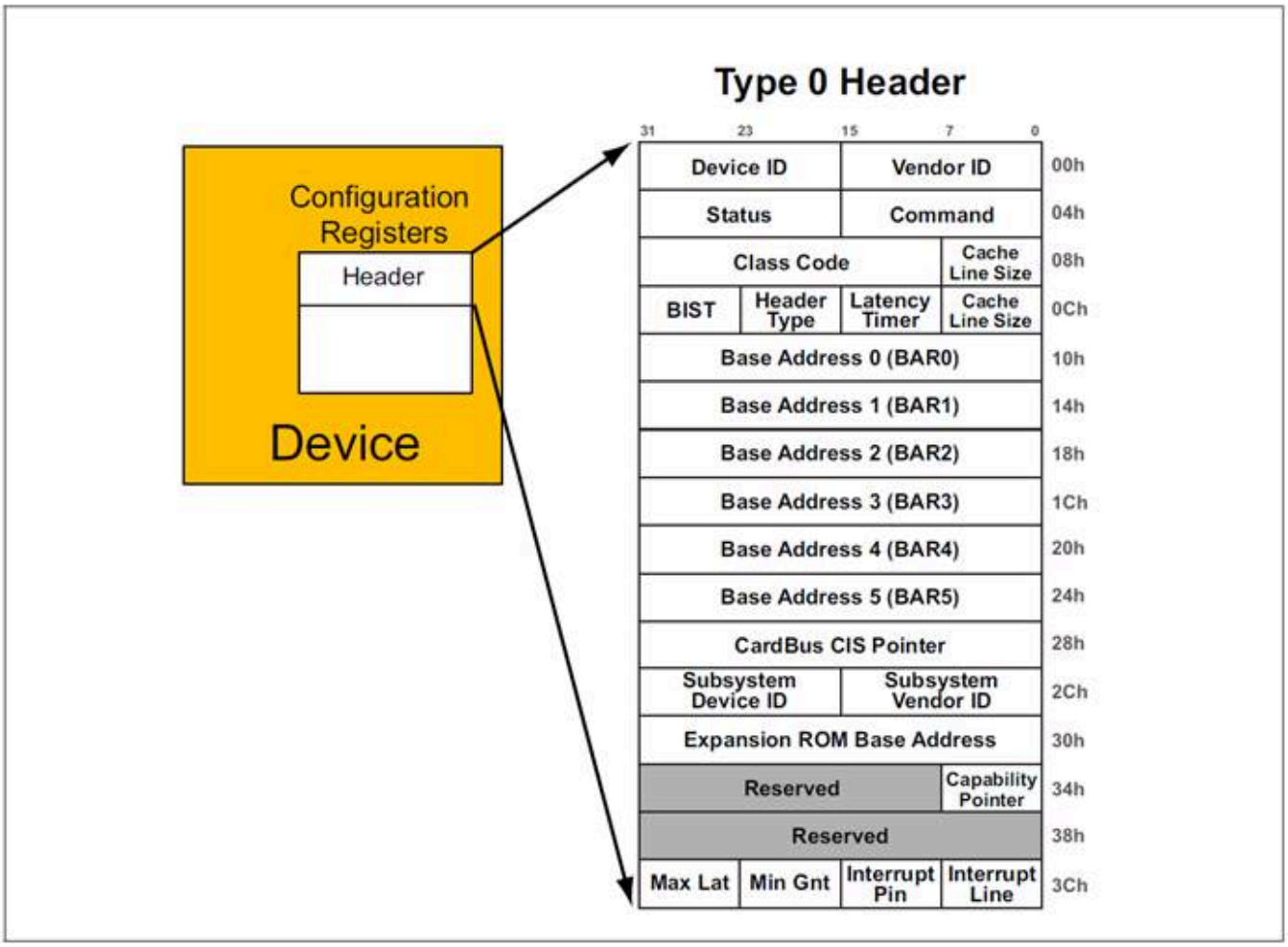

图 1-13 PCI配置Header Type 0（非Bridge）

1.4.9 更高带宽的PCI(Higher-bandwidth PCI)

为了支持更高的带宽，PCI协议规范更新成了支持位宽更宽（64bit），时钟速率更快（66MHz）的版本，使其可以达到533MB/s的速率。图 1-14展示了一个使用66MHz 64bit PCI总线的系统。

Figure 1-14: 66 MHz PCI Bus Based Platform

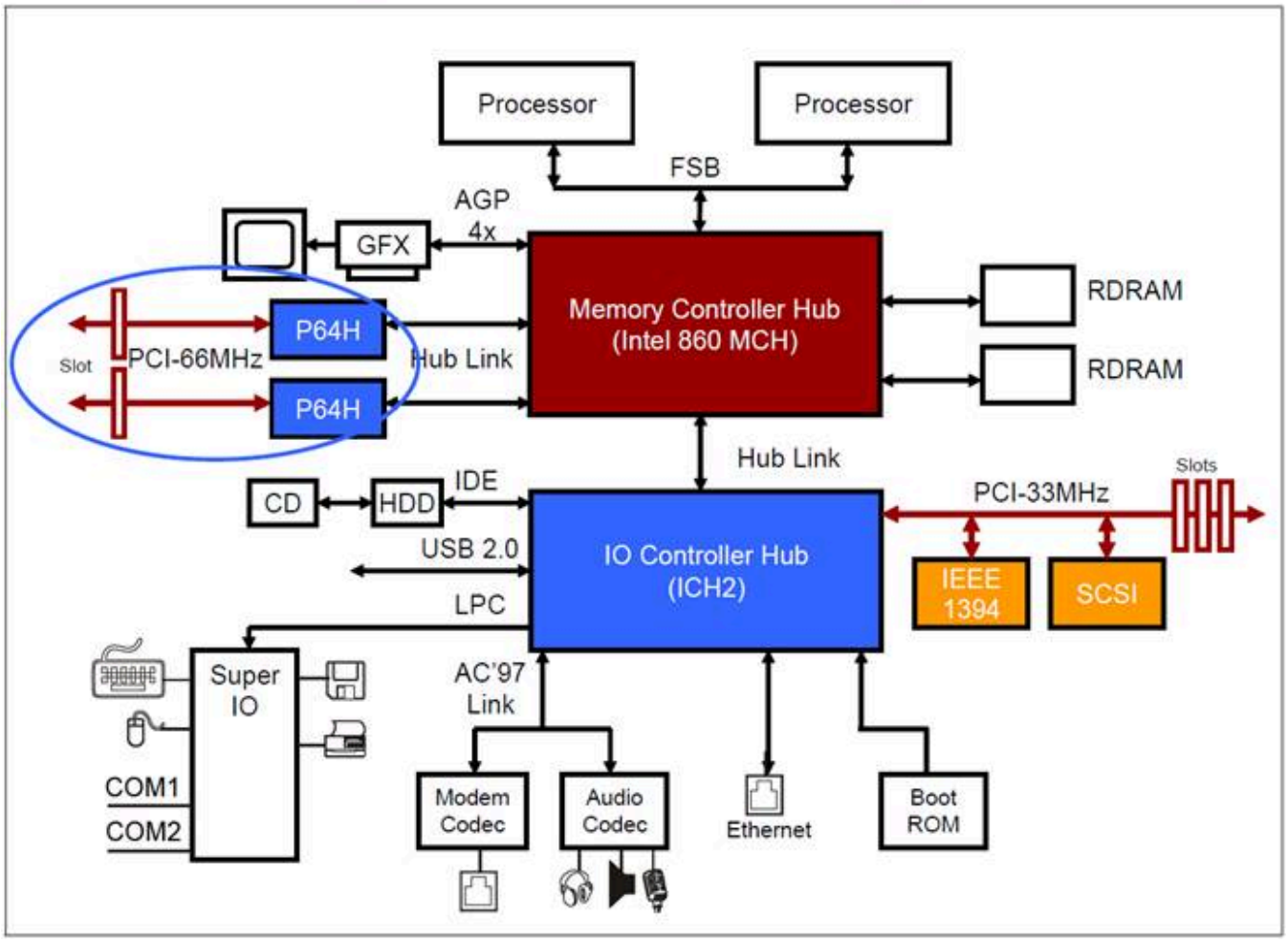

图 1-14 基于66MHz PCI总线的平台

1.4.9.1 66MHz PCI总线的限制

66MHz总线的吞吐量相较于33MHz已经翻倍，但是其存在一定的问题，图 1-14显示了它的一个主要缺点：66MHz总线与33MHz总线使用相同的反射波开关模型，使得留给在传输线上信号传输的时间减半，这将会大大的降低总线的负载能力。最终造成的结果是，一条总线上只能有一个插入板卡。增加更多的设备意味着需要增加更多的PCI Bridge来产生更多的总线，这将会增加成本以及对PCB板材的要求。同时，64bit PCI总线相较于32bit增加了引脚数，这也会增加系统成本且降低可靠性。将上述结合起来，就可以很容易看出来为什么这些因素限制了64bit或者66MHz版本的PCI总线的广泛使用。

1.4.9.2 并行PCI总线模型在66MHz以上时出现的信号时序问题

鉴于PCI总线上的实际负载以及信号渡越时间（signal flight times），PCI总线的时钟频率无法在66MHz上继续增加了。对于66MHz时钟来说，其时钟周期为15ns。分配给接收器的建立时

间(Setup time)为3ns。因为PCI使用“非寄存输入（non-registered input）”信号模型，想要将其3ns的建立时间继续减小是不现实的。剩余的12ns的时序预算（timing budget）分配给了发送器的输出延迟以及信号传输时间。如果在66MHz的基础上进一步提高频率，那么总线上传输的信号将会因为无法及时到达接收端，无法在接收端被正确采样，从而导致传输失败。

在下一节将会介绍PCI-X总线，它将所有的输入信号都先用触发器来寄存一下，然后再去使用。这样做可以将信号的建立时间降低到1ns以下。在建立时间上的优化使得PCI-X总线可以跑到一个更高的频率，例如100MHz甚至133MHz。下一节中我们将会对PCI-X总线体系结构进行简明的介绍。

1.5 PCI-X简介

PCI-X对PCI的软件和硬件都能做到向后兼容，同时PCI-X还能提供更好的性能和更高的效率。它和PCI所使用的连接器也是相同的，因此PCI-X与PCI的设备可以插入相互的插槽。除此之外，它们二者还使用相同的配置模型，因此在PCI系统中能使用的设备、操作系统、以及应用，在PCI-X系统中仍然可以使用。

为了在不改变PCI信号传输模型的基础上达到更高的速率，PCI-X使用了几个技巧来改善总线时序。首先，他们实现了PLL（锁相环）时钟发生器，利用它在内部提供相移时钟。这使得输出信号在相位上前移，而输入再在相位上延后一点进行采样，从而改善总线上的时序。同时，PCI-X的输入信号都在Target设备的输入引脚被寄存（锁存），这样就使得建立时间更小。通过这些方法节省出来的时间可以有效增加信号在总线上传输的可用时间，并使得总线可以进一步提高时钟频率。

1.5.1 PCI-X系统示例(PCI-X System Example)

如图 1-15所示，这是一个基于Intel 7500服务器芯片组的系统平台。

Figure 1-15: 66 MHz/133 MHz PCI-X Bus Based Platform

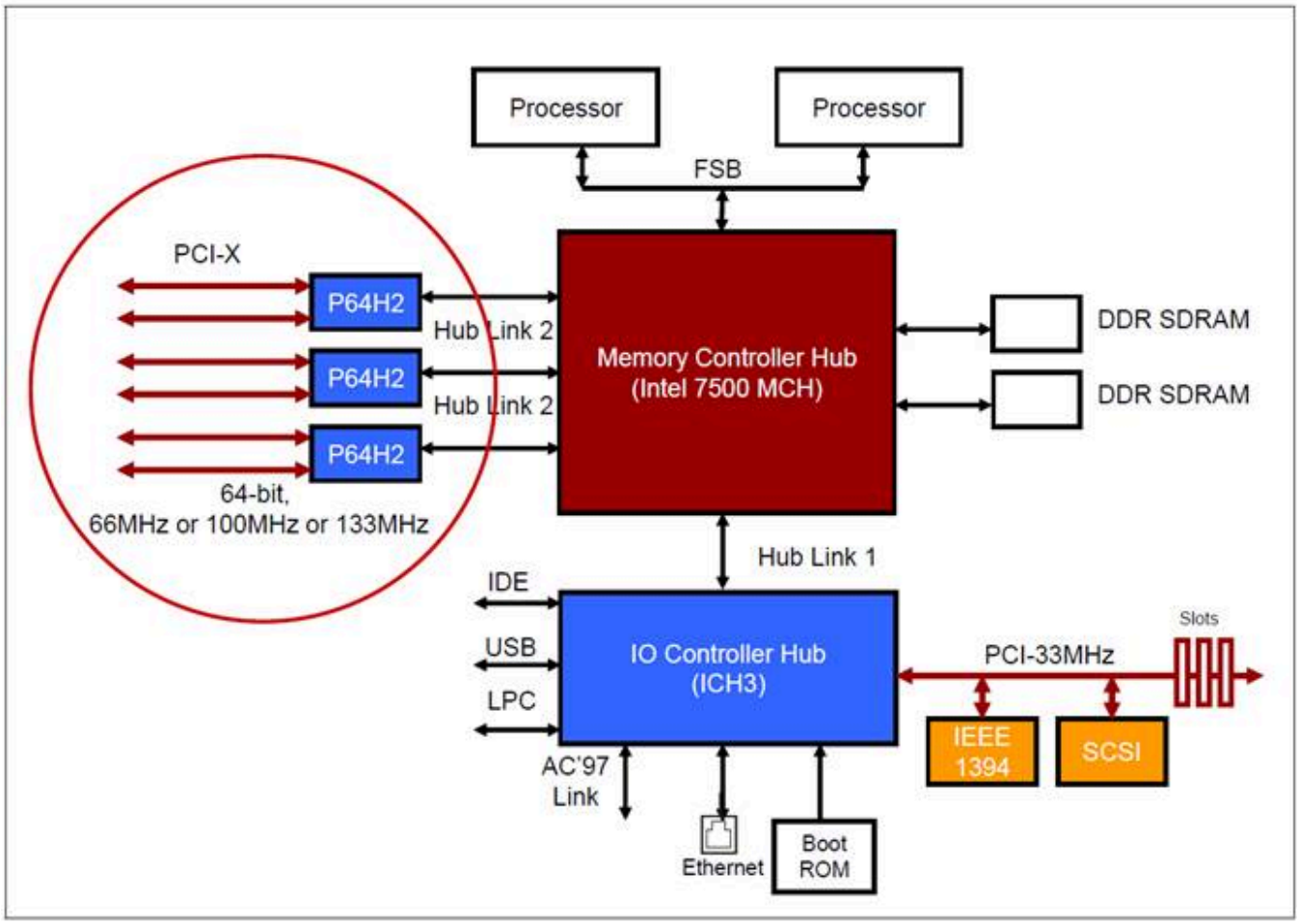

图 1-15基于66MHz/33MHz PCI-X总线的平台

MCH（Memory Controller Hub）芯片提供了3个额外的高性能的Hub-Link 2.0端口，它们各与一个PCI-X 2 Bridge（P64H2）相连接。每个Bridge支持两条PCI-X总线，总线最高时钟频率可以达到133MHz。Hub Link 2.0可以支撑PCI-X更高带宽的数据流。需要注意的是，在PCI-X上我们依然存在与当初改进66MHz PCI时遇到的一样的负载方面的问题，这使得如果我们要支持更多的设备那么就需要使用Bridge来生成更多的总线，并且这将会是一个相对成本昂贵的解决方案。不过，现在的带宽确实提高了不少。

1.5.2 PCI-X事务(PCI-X Transactions)

如图 1-16所示，这是一个PCI-X总线上Memory突发读取（burst memory read）事务。

Figure 1-16: Example PCI-X Burst Memory Read Bus Cycle

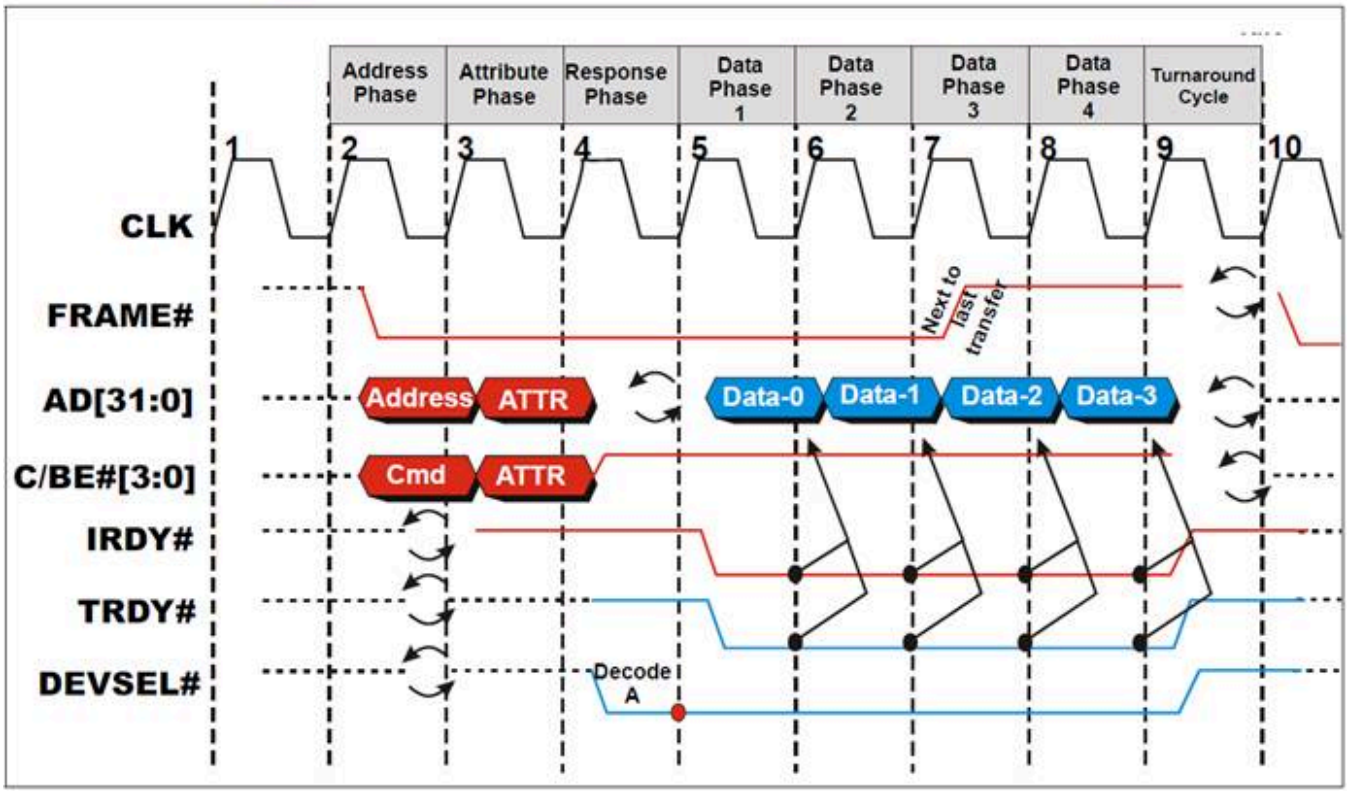

图 1-16 PCI-X Memory突发读取的总线时序周期示例

需要注意的是，PCI-X并不允许在第一个数据阶段（first data phase）后插入等待态。之所以这样做是因为PCI-X中会在事务的属性阶段（Attribute Phase），将需要传输的数据总量告诉Target设备。因此与PCI不同，Target是知道自己需要传输多少数据的。此外，多数PCI-X总线时序周期是连续的，因为使用的是突发模式，并且数据通常以128Bytes作为一个数据块（block）来进行传输。这些特性使得总线利用率提高，同时也提高了设备buffer管理的效率。

1.5.3 PCI-X特性(PCI-X Features)

1.5.3.1 拆分事务模型（Split-Transaction Model）

在传统的PCI读事务中，总线Master向总线上某个设备发起读取。如前面的内容所述，若Target设备未准备好，无法完成事务，那么它既可以选择在获取数据的同时让总线保持等待态，也可以发起Retry来推迟事务。

PCI-X则不同，它使用拆分事务的方法来处理这些情况，如图 1-17所示。

Figure 1-17: PCI-X Split Transaction Protocol

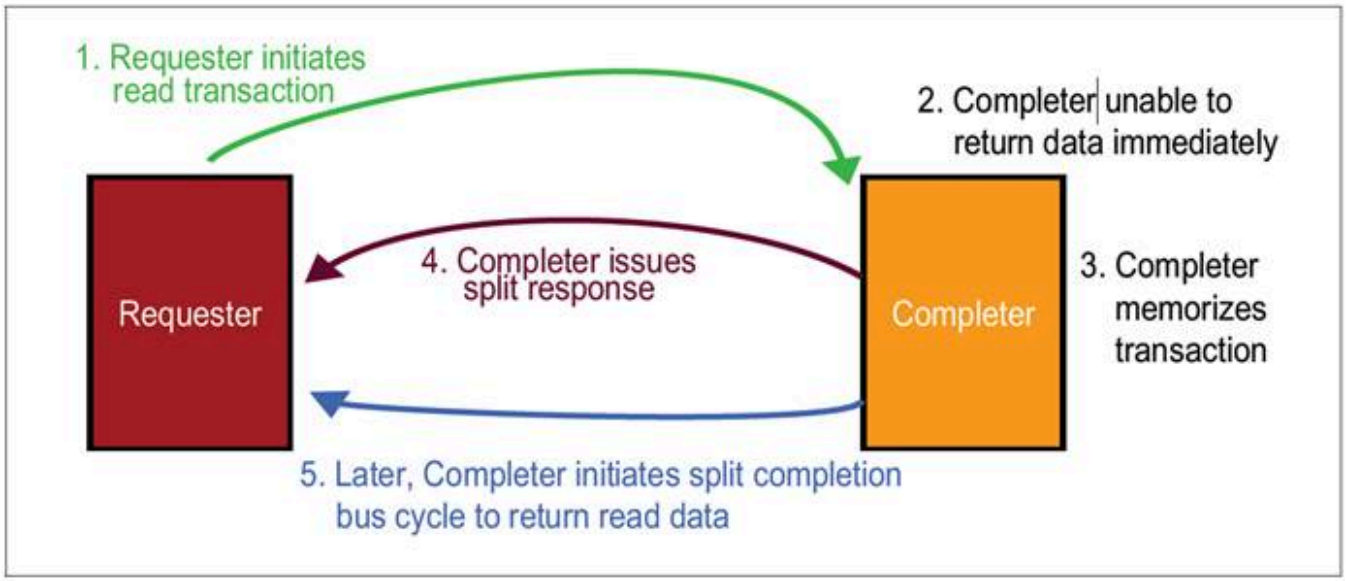

图 1-17 PCI-X 拆分事务处理协议

为了方便描述追踪每个设备正在做什么，我们现在将例子中发起读取操作的一方称为Requester（请求方），将完成读取请求提供数据的一方称为Completer（完成方）。如果Completer无法马上对请求做出相应的服务，它将把这个事务的相关信息存储起来（地址、事务类型、总数据量、Requester ID），并发出拆分响应信号。这就告诉Requester可以将这个事务先放置在队列中，并结束当前的总线时序周期，释放总线使之回到空闲状态。这样，在Completer等待所请求的数据这段时间里，总线上可以执行其他的事务。Requester在这段时间里也可以自由的做其他事情，比如发起另一个请求，甚至是向此前的那个Completer发起请求都可以。一旦Completer收集到了所请求的数据，它将会申请总线占用仲裁，当获取到总线的使用权后，它将发起一个拆分完成（Split Completion）来返回此前Requester所请求的数据。Requester将会响应并参与拆分完成这个事务的总线时序周期，在这个过程中接收来自Completer的数据。对于系统来说，拆分完成事务其实与写事务非常相似。这种拆分事务模型（Split Transaction Model）的可行性在于，在请求发起时就通过属性阶段（Attribute Phase）指出了总共需要传输多少数据，并且也告诉了Completer是谁发起了这个请求（通过提供Requester自己的Bus:Device:Function号码），这使得Completer在发起拆分完成事务时能够找到正确的目标。

对于上述的整个数据传输过程来说，需要通过两个总线事务来完成，但是读请求和拆分完成这两个事务之间的这段时间里，总线是可以执行其他任务的。Requester不需要重复的轮询设备来检查数据是否已经准备好。Completer只需要简单的申请总线占用仲裁，然后在能使用总线时将被请求的数据返回给Requester即可。就总线利用率而言，这样的操作流程使得事务模型更加高效。

到目前为止，对PCI-X所做的这些协议增强改进使得PCI-X的传输效率提高至约85%，而使用标准PCI协议则为50%-60%。

1.5.3.2 MSI消息式中断（Message Signaled Interrupts）

PCI-X设备需要支持MSI中断（Message Signaled Interrupts），在传统中断体系结构中经常见到多个设备需要共享中断，开发这种功能是为了减少或者消除这种需要。想要产生一个MSI中断请求，设备需要发起一个memory写事务，其操作地址为一段预先定义好的地址区域，在这个地址区域内的一个写入操作就会被视为是一个中断，这个信息将会被传送给一个或多个CPU，写入的数据则是发起中断的设备的中断向量，这个中断向量在系统中是唯一的，仅指向这一个设备。对于CPU来说，拥有了中断号之后就可以迅速的跳转到对应设备的中断服务程序，这样就避免了还需要花费额外的开销来查找是那个设备发起的中断。此外，这样的中断方式就不需要额外的中断引脚了。

1.5.3.3 事务属性（Transaction Attributes）

最后，我们来介绍一下PCI-X在每个事务开始时新加入的一个阶段，称之为属性阶段（如图1-16所示）。在段时间中，Requester会将所发起事务的一些信息传递给Completer，信息包括请求的总数据量大小、Requester是谁（Bus:Device:Function 号码），这些信息可以有效的让总线上的事务执行提升效率。除了这些以外，还新引入了2bit的信号来描述所发起的事务：“No Snoop（无窥探）”位与“Relaxed Ordering（宽松排序）”位。

l **No Snoop**（NS）：一般来说，当一个事务要把数据移入或者移出Memory时，CPU内部的Cache需要检查被操作的Memory区域中是否存在已经被拷贝到Cache中的部分。若存在，那么Cache需要在事务访问Memory之前就把数据写回Memory，或是把Cache内的数据按失效处理。当然，这个窥探Memory内容有无出现在cache中的过程将会消耗一定的时间，并且会增加这个请求的延迟。在有些情况下，软件是知道某个请求的Memory位置永远不会出现在Cache中（这可能是因为这个位置被系统定义为不可缓存的），因此对于这些位置是不需要进行窥探的，应该跳过这一步骤。No Snoop位正是基于这种原因而被引入进来。

l **Relaxed Ordering**（RO）：通常情况下，当事务通过Bridge的buffer时，事务间需要保持当初被放置到总线上时的顺序。这被称为强排序模型（Strongly ordered model），PCI与PCI-X一般都会遵循这种规则，除了一些个别情况。这是因为强排序模型可以解决有相互联系的事务之间的依赖问题，例如对同一个位置先进行写入再进行读取，按照强排序模型就可以使得读取操作读取到写入后的正确数据，而不是写入前的旧数据。然而实际上并不是所有的事务都存在依赖关系。如果有些事务不存在依赖关系，那么依然强制它们保持原来的顺序将会造成性能损失，这就是新添加这一属性比特位想解决的问题。如果Requester知道某个特定的事务是与此前其他事务不相关联的，那么就可以将这个事务的这一位置为有效，这样就能告诉Bridge可以在队列中向

前调度该事务，提前执行，以此带来更好的性能。

1.5.4 更高带宽的PCI-X(Higher Bandwidth PCI-X)

1.5.4.1 PCI与PCI-X 1.0并行总线模型中由于公共时钟方法所带来的问题

当尝试将PCI这样的总线升级到更高的速率时，一个问题变得越来越明显，那就是并行总线设计存在一些固有的先天局限。图 1-18可以有助于理解这个问题。这些设计通常使用公共时钟（common clock）或者是分布式时钟（distributed clock），其中，发送方在公共时钟的某一个时钟沿将数据输出，然后接收方在公共时钟的下一个时钟沿将数据锁存，也就是说用于传输的时间预算长度就是一个时钟周期。

Figure 1-18: Inherent Problems in a Parallel Design

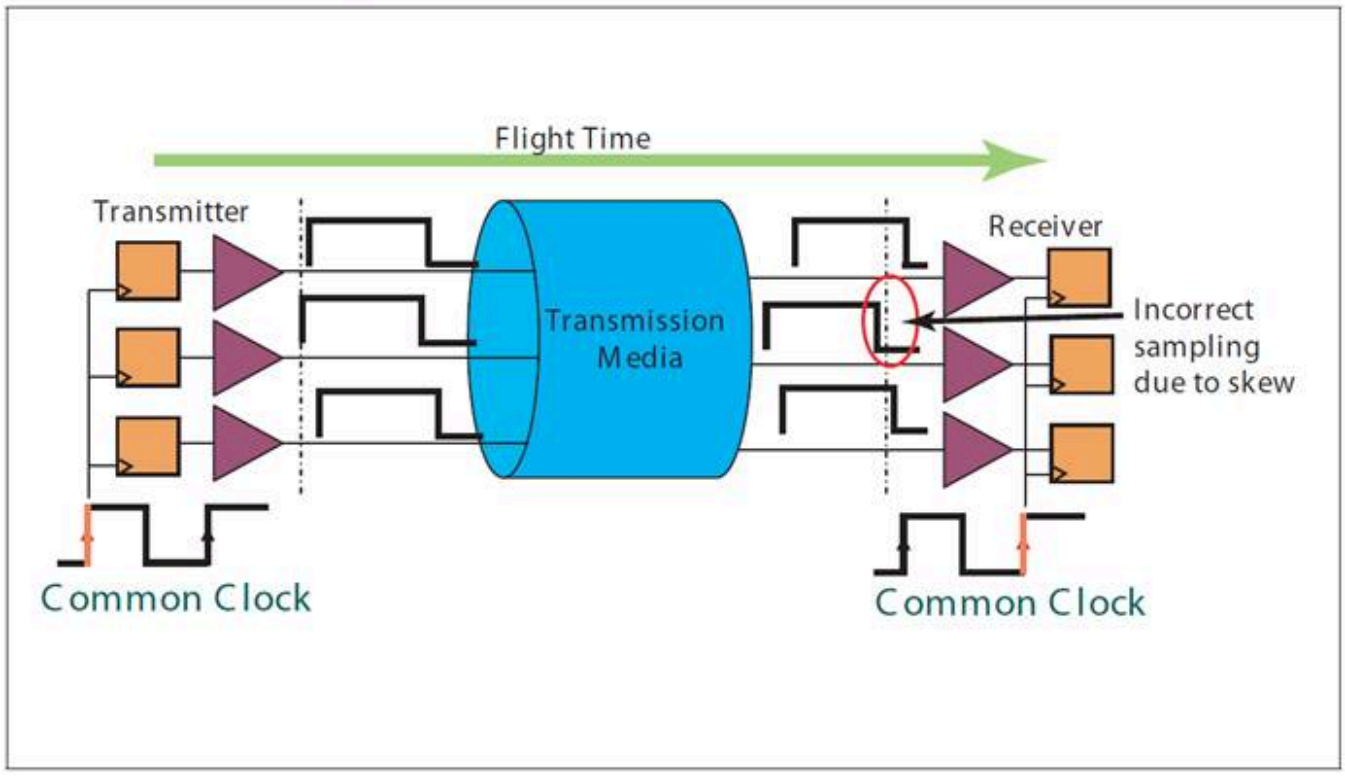

图 1-18并行设计的固有问题

需要注意的第一个问题是信号歪斜（signal skew）。当数据的多个bit在同一时刻被发出，他们所经历的延时会存在轻微的差别，这使得他们到达接收方的时刻也会存在轻微的差别。如果某些原因造成这种差别过大，那么就会出现图中所示的信号采样错误的问题。第二个问题是多个设备间的时钟歪斜问题。公共时钟到达各个设备的时间点并不是十分精确一致的，这也会大大的压缩了用于传输信号的时间预算长度。最后，第三个问题与一个信号从发送方传输到接收方所需要的时间有关，我们称这个时间为渡越时间（flight time）。时钟周期或是传输时间预算必须要大于

信号渡越时间。为了确保满足这样的条件，PCB硬件板级设计需要让信号走线尽可能短，以此来让信号传输时延小于时钟周期。在许多电路板设计中，这种短信号走线的设计并不现实。

尽管存在这些自身局限，但是为了更进一步提升性能，还是有几项技术可以使用的。首先，可以对现有的协议进行简化而提高效率。第二，可以将总线模型更改为源同步时钟模型（source synchronous clocking model），在这种时钟模型中，总线信号和时钟（strobe选通脉冲）被驱动后将经历相同的传输延迟到达接收端（这里可以对比一下common clock与其的区别）。这种方法被PCI-X 2.0所采用。

1.5.4.2 PCI-X 2.0源同步模型（PCI-X 2.0 Source-Synchronous Model）

PCI-X2.0更进一步的提高了PCI-X的带宽。如此前升级时一样，PCI-X 2.0依然保持了对PCI的软件与硬件的向后兼容性。为了达到更高的速率，总线使用了源同步传输模型，来支持DDR（双倍数据速率）或是QDR（四倍数据速率）。

“源同步”这个术语的意思是，设备在传输数据的同时，还提供了额外一个与数据信号基本传输路径相同的信号。如图 1-19所示，在PCI-X 2.0中，那个“额外”的信号被称为“strobe（脉冲选通）”，接收方使用它来锁存发来的数据。发送方可以分配data与strobe信号之间的时序关系，只要data与strobe信号的传输路径长度相近且其他会影响传输延迟的条件也相近，那么当信号到达接收方时，这种信号间的时序关系也将不会发生变化。这种特点可以用于支持更高速率的传输，这是因为在源同步时钟模型中，接收方采样所使用的strobe（类似于原来的时钟）也是数据的来源方发送的，这样就避免了公共时钟到达各设备的延时不同而造成的时钟歪斜问题，因为如上所述此时的data与strobe之间的时序关系是基本相对不变的，这样就将时钟歪斜的这段时间单独剔除了出去，不再占用原来的传输时间预算。除此之外，因为存在了strobe信号来随路指示接收方何时锁存采样数据，所以信号渡越时间不再是个问题，即使走线较长也依然能由伴随data的strobe信号来让接收方正确的采样数据，而不必纠结在下一个“时钟上升沿”必须要进行采样，现在strobe何时指示则在何时采样。

Figure 1-19: Source-Synchronous Clocking Model

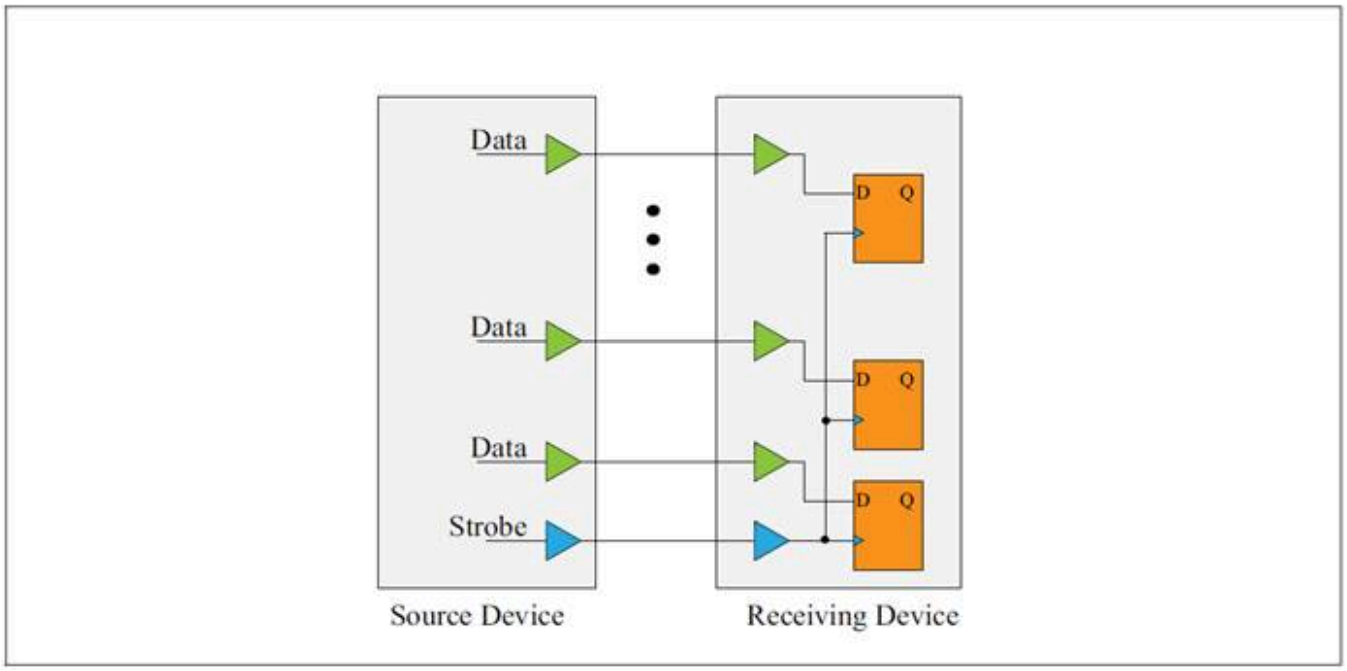

图 1-19源同步时钟模型

需要再次强调的是，这种非常高速的信号时序使得共享总线模型无法被使用，它只能被设计为点对点的传输形式。因此，想要增加更多的设备就需要更多的Bridge来产生更多的总线。为了达到这样的使用方法，可以设计一种设备，它拥有3个PCI-X 2.0接口，其内部还有一个Bridge结构以此来让他们三者进行相互通信。不过，这样的设备将会具有很高的引脚数量，而且成本也会更高，这使得PCI-X 2.0被困在了高端市场而无法大面积普及。

由于协议制定者认识到了PCI-X 2.0的昂贵解决方案仅能吸引更多的高端开发者而不是一般开发者，因此PCI-X 2.0提供了更符合高端设计的技术标准，例如支持ECC的生成与校验。PCI-X的ECC比PCI中使用的奇偶校验要更为复杂，鲁棒性（robust）也要高得多，它可以动态的自动纠正单bit错误，以及对多bit错误进行可靠的检测。这种改进后的错误处理方法增加了成本，但是对于高端平台来说，它所能增加的可靠性才是更为看中的，因此即使增加了成本这也是一个合理的选择。

尽管PCI-X 2.0将带宽、效率以及可靠性都提升了，但是并行总线的终究还是走到了它的终点，需要一种新的模型来满足人们对于更高带宽且更低成本的不断的需求。最终被选中的新型总线模型使用的是串行接口，从物理层面来看它与PCI是完全不同的总线，但是它在软件层面保留了对PCI的向后兼容性。

这个新模型，就是我们所知道的PCI Express（PCIe）。

原文： Mindshare

译者： Michael ZZY

校对： LJGibbs

欢迎参与《Mindshare PCI Express Technology 3.0 一书的中文翻译计划》

https://gitee.com/ljgibbs/chinese-translation-of-pci-express-technology