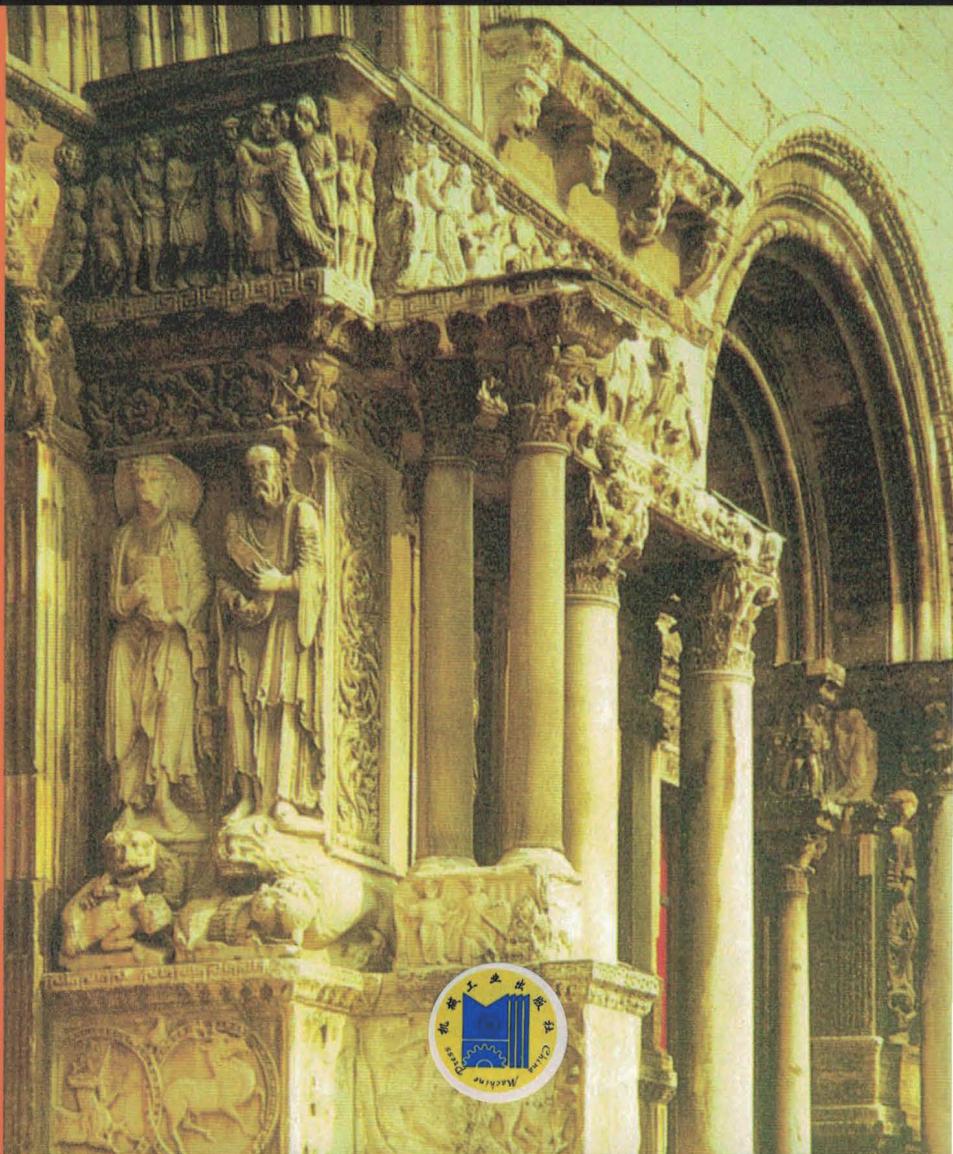


PCI Express 体系结构导读

■ 王齐 编著



PCI Express 体系结构导读

王齐 编著



机械工业出版社

本书讲述了与 PCI 及 PCI Express 总线相关的最为基础的内容，并介绍了一些必要的、与 PCI 总线相关的处理器体系结构知识，这也是本书的重点所在。深入理解处理器体系结构是理解 PCI 与 PCI Express 总线的重要基础。

读者通过对本书的学习，可超越 PCI 与 PCI Express 总线自身的内容，理解在一个通用处理器系统中局部总线的设计思路与实现方法，从而理解其他处理器系统使用的局部总线。本书适用于希望多了解一些硬件的软件工程师，以及希望多了解一些软件的硬件工程师，也可供电子工程和计算机类的研究生自学参考。

图书在版编目(CIP)数据

PCI Express 体系结构导读/王齐编著.—北京:机械工业出版社, 2010.3 (2016.6 重印)
ISBN 978-7-111-29822-9

I. ①P… II. ①王… III. ①总线 - 结构 IV. ①TP336

中国版本图书馆 CIP 数据核字(2010)第 028735 号

机械工业出版社(北京市百万庄大街 22 号 邮政编码 100037)

责任编辑：车 忱

责任印制：李 洋

三河市宏达印刷有限公司印刷

2016 年 6 月第 1 版 · 第 4 次印刷

184mm × 260mm · 28.5 印张 · 704 千字

7301-8500 册

标准书号：ISBN 978-7-111-29822-9

定价：75.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

网络服务

服务咨询热线：(010) 88361066

机工官网：www.cmpbook.com

读者购书热线：(010) 68326294

机工官博：weibo.com/cmp1952

(010) 88379023

教育服务网：www.cmpedu.com

封面无防伪标均为盗版

金书网：www.golden-book.com

前　　言

PCI 与 PCI Express (PCIe) 总线在处理器系统中得到了大规模应用。PCISIG 也制定了一系列 PCI 与 PCI Express 总线相关的规范，这些规范所涉及的内容庞杂广泛。对于已经理解了 PCI 与 PCI Express 总线的工程师，这些规范便于他们进一步获得必要的细节知识。对于刚刚接触 PCI 与 PCI Express 总线的工程师，这些规范性的文档并不适合阅读。在阅读这些规范时，工程师还需要具备一些与体系结构相关的基础知识，这恰是规范并不涉及的内容。对于多数工程师，规范文档适于查阅，而不便于学习。

本书将以处理器体系结构为主线介绍 PCI Express 总线的组成，以便读者进一步理解 PCI Express 总线协议。本书并不是关于 PCI 和 PCI Express 总线的百科全书，因为读者完全可以通过阅读 PCI 和 PCI Express 总线规范获得细节信息。本书侧重的是 PCI 和 PCI Express 总线中与处理器体系结构相关的内容。

本书不会对 PCI 总线的相关规范进行简单重复，部分内容并不在 PCI 总线规范定义的范围内，例如 HOST 主桥和 RC。PCI 总线规范并没有规定处理器厂商如何实现 HOST 主桥和 RC，不同的处理器厂商实现的 HOST 主桥和 RC 有较大差异，而这些内容正是本书所讨论的重点。此外本书还讲述了一些在 PCI 总线规范中提及，但是容易被忽略的一些重要概念。

本书共由三篇组成。第 I 篇（第 1~3 章）介绍 PCI 总线的基础知识。第 II 篇（第 4~13 章）介绍 PCI Express 总线的相关概念。第 II 篇的内容以第 I 篇为基础。从系统软件的角度来看，PCI Express 总线向前兼容 PCI 总线，理解 PCI Express 总线必须建立在深刻理解 PCI 总线的基础之上。读者需要按照顺序阅读这两篇。

第 1 章主要说明 PCI 总线涉及的一些基本知识。有些知识稍显过时，但是在 PCI 总线中出现的一些数据传送方式，如 Posted、Non-Posted 和 Split 数据传送方式，依然非常重要，也是读者需要掌握的。

第 2 章重点介绍 PCI 桥。PCI 桥是 PCI 及 PCI Express 体系结构的精华所在，本章还使用了一定篇幅介绍了非透明桥。非透明桥不是 PCI 总线定义的标准桥片，但是在处理器系统之间的互联中得到了广泛的应用。

第 3 章详细阐述 PCI 总线的数据传送方式，与 Cache 相关的内容和预读机制是本章的重点。目前 PCI 与 PCI Express 对预读机制的支持并不理想。但是在可以预见的将来，PCI Express 总线将充分使用智能预读机制进一步提高总线的利用率。

第 4 章是 PCI Express 篇的综述。第 5 章以 Intel 的笔记本平台 Montevina 为例说明 RC 的各个组成模块。实际上 RC 这个概念，只有在 x86 处理器平台中才真正存在。其他处理器系统中，并不存在严格意义上的 RC。

第 6、7 章分别介绍 PCI Express 总线的事务层、数据链路层和物理层。物理层是 PCI Express 总线的真正核心，也是中国工程师最没有机会接触的内容。这是我们这一代工程师的遗憾与无奈。第 8 章简要说明了 PCI Express 总线的链路训练与电源管理。

第 9 章主要讨论的是通用流量控制的管理方法与策略。PCI Express 总线的流量控制机

制仍需完善，其中不等长的报文长度也是限制 PCI Express 总线流量控制进一步提高的重要因素。

第 10 章重点介绍 MSI 和 MSI-X 中断机制。MSI 中断机制在 PCI 总线中率先提出，但是在 PCI Express 总线中才得到大规模普及。目前 x86 架构多使用 MSI-X 中断机制，而在许多嵌入式处理器中仍然使用 MSI 中断机制。

第 11 章的篇幅很短，重点介绍 PCI 和 PCI Express 总线中的序。有志于学习处理器体系结构的工程师务必掌握这部分内容。在处理器体系结构中有关 Cache 和数据传送序的内容非常复杂，掌握这些内容也是系统工程师进阶所必须的。

第 12 章讲述了笔者的一个实际设计——Capric 卡，简单介绍了 Linux 设备驱动程序的实现过程，并对 PCI Express 总线的延时与带宽进行了简要分析。

第 13 章介绍 PCI 总线与虚拟化相关的一些内容。虚拟化技术已崭露头角，与虚拟化相关的一系列内容将对处理器体系结构产生深远的影响。目前虚拟化技术已经在 x86 处理器中得到了广泛的应用。

第Ⅲ篇以 Linux 系统为实例说明 PCI 总线在处理器系统中的使用方法，也许有许多读者对这一篇有着浓厚的兴趣。Linux 无疑是一个非常优秀的操作系统。但是需要提醒系统工程师，Linux 系统仅是一个完全开源的操作系统。对于有志于学习处理器体系结构的工程师，学习 Linux 系统是必要的，但是仅靠学习 Linux 系统并不足够。

通常说来，理解处理器体系结构至少需要了解两三种处理器，并了解它们在不同操作系统上的实现。尺有所短，寸有所长。不同的处理器和操作系统所应用的领域并不完全相同。也是因为这个原因，本书以 PowerPC 和 x86 处理器为基础对 PCI 和 PCI Express 总线进行说明。

本书在写作过程中得到了我的同事和在处理器及操作系统行业奋战多年的朋友们的帮助。在 Linux 系统中许多与处理器和 PCI 总线相关的模块，都有着他们的辛勤付出。刘建国和郭超宏先生审阅了本书的第Ⅰ篇。马明辉先生审阅了本书的第Ⅱ篇。张巍、余珂与刘劲松先生审阅了第 13 章。吴晓川、王勇、丁建峰、李力与吴强先生共同审阅了全书。

本书第 12 章中出现的 Capric 和 Cornus 卡由郭冠军和高健协助完成。看着他们通过对 PCI Express 总线理解的逐渐深入，最终设计出一个具有较高性能的 Cornus 卡，备感欣慰。此外杨强浩先生也参与了 Capric 和 Cornus 卡的原始设计与方案制定，在此对他及他的团队在这个过程中给予的帮助表示感谢，我们也一道通过这两块卡的制作进一步领略了 PCI Express 总线的技术之美。

一个优秀的协议，从制定到广大技术人员理解其精妙之处，再到协议应用到一个个优秀产品中，需要更多的人参与、投入、实践，这也是编写此书最大的动力源泉。本书的完成与我的妻子范淑琴的激励直接相关，Capricornus 也是她的星座。还需要感谢本书的编辑车忱与策划时静，正是他们的努力使得本书提前问世。

对本书尚留疑问的读者，可通过我的邮箱 sailing.w@gmail.com 与我联系。最后希望这本书对您有所帮助。

作 者

目 录

前言

第 I 篇 PCI 体系结构概述

第 1 章 PCI 总线的基本知识	3	第 2 章 PCI 总线的桥与配置.....	28
1.1 PCI 总线的组成结构	5	2.1 存储器域与 PCI 总线域	28
1.1.1 HOST 主桥	6	2.1.1 CPU 域、DRAM 域与存储器域	29
1.1.2 PCI 总线	7	2.1.2 PCI 总线域.....	30
1.1.3 PCI 设备	7	2.1.3 处理器域	30
1.1.4 HOST 处理器	8	2.2 HOST 主桥	32
1.1.5 PCI 总线的负载	8	2.2.1 PCI 设备配置空间的访问机制	33
1.2 PCI 总线的信号定义	9	2.2.2 存储器域地址空间到 PCI 总线	
1.2.1 地址和数据信号	9	域地址空间的转换	35
1.2.2 接口控制信号	10	2.2.3 PCI 总线域地址空间到存储器	
1.2.3 仲裁信号	12	域地址空间的转换	37
1.2.4 中断请求等其他信号	12	2.2.4 x86 处理器的 HOST 主桥	40
1.3 PCI 总线的存储器读写总线事务	13	2.3 PCI 桥与 PCI 设备的配置空间	42
1.3.1 PCI 总线事务的时序	14	2.3.1 PCI 桥	42
1.3.2 Posted 和 Non-Posted 传送方式.....	15	2.3.2 PCI Agent 设备的配置空间	44
1.3.3 HOST 处理器访问 PCI 设备.....	16	2.3.3 PCI 桥的配置空间	50
1.3.4 PCI 设备读写主存储器	18	2.4 PCI 总线的配置	53
1.3.5 Delayed 传送方式	19	2.4.1 Type 01h 和 Type 00h 配置请求	53
1.4 PCI 总线的中断机制	21	2.4.2 PCI 总线配置请求的转换原则	55
1.4.1 中断信号与中断控制器的连接		2.4.3 PCI 总线树 Bus 号的初始化.....	57
关系	21	2.4.4 PCI 总线 Device 号的分配	59
1.4.2 中断信号与 PCI 总线的连接		2.5 非透明 PCI 桥	60
关系	22	2.5.1 Intel 21555 中的配置寄存器	62
1.4.3 中断请求的同步.....	23	2.5.2 通过非透明桥片进行数据传递	63
1.5 PCI-X 总线简介	25	2.6 小结	65
1.5.1 Split 总线事务	25	第 3 章 PCI 总线的数据交换.....	66
1.5.2 总线传送协议	26	3.1 PCI 设备 BAR 空间的初始化	66
1.5.3 基于数据块的突发传送	26	3.1.1 存储器地址与 PCI 总线地址	
1.6 小结	27	的转换	66

3.1.2 PCI 设备 BAR 寄存器和 PCI 桥	空间进行 DMA 读写	81
Base、Limit 寄存器的初始化		67
3.2 PCI 设备的数据传递	3.3.4 PCI 设备进行 DMA 写时发生	69
3.2.1 PCI 设备的正向译码与负向	Cache 命中	82
译码		85
3.2.2 处理器到 PCI 设备的数据传送	3.3.5 DMA 写时发生 Cache 命中	71
3.2.3 PCI 设备的 DMA 操作	的优化	72
3.2.4 PCI 桥的 Combining、Merging	3.4 预读机制	73
和 Collapsing	3.4.1 指令 Fetch	86
3.3 与 Cache 相关的 PCI 总线事务	3.4.2 数据预读	89
3.3.1 Cache 一致性的基本概念	3.4.3 软件预读	91
3.3.2 PCI 设备对不可 Cache 的存储器	3.4.4 硬件预读	93
空间进行 DMA 读写	3.4.5 PCI 总线的预读机制	94
3.3.3 PCI 设备对可 Cache 的存储器	3.5 小结	98

第 II 篇 PCI Express 体系结构概述

第 4 章 PCIe 总线概述	101
4.1 PCIe 总线的基础知识	101
4.1.1 端到端的数据传递	101
4.1.2 PCIe 总线使用的信号	103
4.1.3 PCIe 总线的层次结构	107
4.1.4 PCIe 链路的扩展	108
4.1.5 PCIe 设备的初始化	110
4.2 PCIe 体系结构的组成部件	112
4.2.1 基于 PCIe 架构的处理器系统	112
4.2.2 RC 的组成结构	117
4.2.3 Switch	118
4.2.4 VC 和端口仲裁	120
4.2.5 PCIe-to-PCI/PCI-X 桥片	122
4.3 PCIe 设备的扩展配置空间	123
4.3.1 Power Management Capability	
结构	124
4.3.2 PCI Express Capability 结构	127
4.3.3 PCI Express Extended Capabilities	
结构	133
4.4 小结	139
第 5 章 Montevina 的 MCH 和 ICH	140
5.1 PCI 总线 0 的 Device 0 设备	141
5.1.1 EPBAR 寄存器	144
5.1.2 MCHBAR 寄存器	144
5.1.3 其他寄存器	144
5.2 Montevina 平台的存储器空间的	
组成结构	145
5.2.1 Legacy 地址空间	147
5.2.2 DRAM 域	147
5.2.3 存储器域	148
5.3 存储器域的 PCI 总线地址	
空间	150
5.3.1 PCI 设备使用的地址空间	150
5.3.2 PCIe 总线的配置空间	151
5.4 小结	154
第 6 章 PCIe 总线的事务层	155
6.1 TLP 的格式	155
6.1.1 通用 TLP 头的 Fmt 字段和 Type	
字段	156
6.1.2 TC 字段	158
6.1.3 Attr 字段	159
6.1.4 通用 TLP 头中的其他字段	160
6.2 TLP 的路由	161
6.2.1 基于地址的路由	161

6.2.2 基于 ID 的路由	164	8.1.3 Receiver Detect 识别逻辑	217
6.2.3 隐式路由	166	8.2 LTSSM 状态机	218
6.3 存储器、I/O 和配置读写		8.2.1 Detect 状态	220
请求 TLP	167	8.2.2 Polling 状态	221
6.3.1 存储器读写请求 TLP	168	8.2.3 Configuration 状态	223
6.3.2 完成报文	172	8.2.4 Recovery 状态	228
6.3.3 配置读写请求 TLP	174	8.2.5 LTSSM 的其他状态	231
6.3.4 消息请求报文	175	8.3 PCIe 总线的 ASPM	232
6.3.5 PCIe 总线的原子操作	177	8.3.1 与电源管理相关的链路状态	232
6.3.6 TLP Processing Hint	178	8.3.2 L0 状态	233
6.4 TLP 中与数据负载相关的		8.3.3 L0s 状态	234
参数	181	8.3.4 L1 状态	235
6.4.1 Max_Payload_Size 参数	181	8.3.5 L2 状态	236
6.4.2 Max_Read_Request_Size 参数	182	8.4 PCI PM 机制	237
6.4.3 RCB 参数	183	8.4.1 PCIe 设备的 D-State	237
6.5 小结	184	8.4.2 D-State 的状态迁移	238
第 7 章 PCIe 总线的数据链路层与物理层	185	8.5 小结	240
7.1 数据链路层的组成结构	185	第 9 章 流量控制	241
7.1.1 数据链路层的状态	186	9.1 流量控制的基本原理	242
7.1.2 事务层如何处理 DL_Down 和		9.1.1 Rate-Based 流量控制	243
DL_Up 状态	189	9.1.2 Credit-Based 流量控制	244
7.1.3 DLLP 的格式	189	9.2 Credit-Based 机制使用的算法	246
7.2 ACK/NAK 协议	191	9.2.1 N123 算法和 N123 + 算法	249
7.2.1 发送端如何使用 ACK/NAK		9.2.2 N23 算法	250
协议	192	9.2.3 流量控制机制的缓冲管理	252
7.2.2 接收端如何使用 ACK/NAK		9.3 PCIe 总线的流量控制	254
协议	195	9.3.1 PCIe 总线流量控制的缓存	
7.2.3 数据链路层发送报文的顺序	199	管理	255
7.3 物理层简介	199	9.3.2 Current 节点的 Credit	257
7.3.1 PCIe 链路的差分信号	200	9.3.3 VC 的初始化	259
7.3.2 物理层的组成结构	204	9.3.4 PCIe 设备如何使用 FCP	261
7.3.3 8/10b 编码与解码	206	9.4 小结	262
7.4 小结	210	第 10 章 MSI 和 MSI-X 中断机制	263
第 8 章 PCIe 总线的链路训练与电源管理	211	10.1 MSI/MSI-X Capability 结构	263
8.1 PCIe 链路训练简介	211	10.1.1 MSI Capability 结构	264
8.1.1 链路训练使用的字符序列	213	10.1.2 MSI-X Capability 结构	266
8.1.2 Electrical Idle 状态	216	10.2 PowerPC 处理器如何处理	
		MSI 中断请求	268
		10.2.1 MSI 中断机制使用的寄存器	270

10.2.2 系统软件如何初始化 PCIe 设备的 MSI Capability 结构	272	12.1.4 DMA 读	302
10.3 x86 处理器如何处理 MSI-X 中断请求	274	12.1.5 中断请求	303
10.3.1 Message Address 字段和 Message Data 字段的格式	274	12.2 Capric 卡的数据传递	303
10.3.2 FSB Interrupt Message 总线事务	277	12.2.1 DMA 写使用的 TLP	304
10.4 小结	278	12.2.2 DMA 读使用的 TLP	308
第 11 章 PCI/PCIe 总线的序	279	12.2.3 Capric 卡的中断请求	317
11.1 生产/消费者模型	279	12.3 基于 PCIe 总线的设备驱动	317
11.1.1 生产/消费者的工作原理	280	12.3.1 Capric 卡驱动程序的加载与卸载	318
11.1.2 生产/消费者模型在 PCI/PCIe 总线中的实现	281	12.3.2 Capric 卡的初始化与关闭	319
11.2 PCI 总线的死锁	283	12.3.3 Capric 卡的 DMA 读写操作	324
11.2.1 缓冲管理引发的死锁	283	12.3.4 Capric 卡的中断处理	327
11.2.2 数据传送序引发的死锁	283	12.3.5 存储器地址到 PCI 总线地址的转换	328
11.3 PCI 总线的序	284	12.3.6 存储器与 Cache 的同步	330
11.3.1 PCI 总线序的通用规则	284	12.4 Capric 卡的延时与带宽	334
11.3.2 Delayed 总线事务的传送规则	285	12.4.1 TLP 的传送开销	335
11.3.3 PCI 总线事务通过 PCI 桥的顺序	286	12.4.2 PCIe 设备的 DMA 读写延时	338
11.3.4 LOCK, Delayed 和 Posted 总线事务间的关系	289	12.4.3 Capric 卡的优化	342
11.4 PCIe 总线的序	290	12.5 小结	343
11.4.1 TLP 传送的序	290	第 13 章 PCIe 总线与虚拟化技术	344
11.4.2 ID-Base Ordering	294	13.1 IOMMU	344
11.4.3 MSI 报文的序	295	13.1.1 IOMMU 的工作原理	345
11.5 小结	296	13.1.2 IA 处理器的 VT-d	347
第 12 章 PCIe 总线的应用	297	13.1.3 AMD 处理器的 IOMMU	349
12.1 Capric 卡的工作原理	297	13.2 ATS (Address Translation Services)	352
12.1.1 BAR 空间	298	13.2.1 TLP 的 AT 字段	353
12.1.2 Capric 卡的初始化	301	13.2.2 地址转换请求	354
12.1.3 DMA 写	302	13.2.3 Invalidate ATC	356

第 III 篇 Linux 与 PCI 总线

第 14 章 Linux PCI 的初始化过程	365	13.3 SR-IOV 与 MR-IOV	358
14.1 Linux x86 对 PCI 总线的		13.3.1 SR-IOV 技术	358
		13.3.2 MR-IOV 技术	359

14.1.1 pcibus_class_init 与 pci_driver_init		13.4 小结	362
--	--	---------	-----

函数	368	BAR 寄存器	407
14.1.2 pci_arch_init 函数	369	14.4 Linux PowerPC 如何初始化 PCI	
14.1.3 pci_slot_init 和 pci_subsys_init		总线树	412
函数	372	14.5 小结	416
14.1.4 与 PCI 总线初始化相关的其他		第 15 章 Linux PCI 的中断处理	417
函数	373	15.1 PCI 总线的中断路由	417
14.2 x86 处理器的 ACPI	374	15.1.1 PCI 设备如何获取 irq 号	419
14.2.1 ACPI 驱动程序与 AML		15.1.2 PCI 中断路由表	426
解释器	377	15.1.3 PCI 插槽使用的 irq 号	428
14.2.2 ACPI 表	380	15.2 使用 MSI/MSIX 中断机制	
14.2.3 ACPI 表的使用实例	382	申请中断向量	432
14.3 基于 ACPI 机制的 Linux PCI 的		15.2.1 Linux 如何使能 MSI 中断	
初始化	388	机制	432
14.3.1 基本的准备工作	388	15.2.2 Linux 如何使能 MSI-X 中断	
14.3.2 Linux PCI 初始化 PCI 总线号 ...	393	机制	437
14.3.3 Linux PCI 检查 PCI 设备使用的		15.3 小结	440
BAR 空间	404	参考文献	441
14.3.4 Linux PCI 分配 PCI 设备使用的			

第 I 篇 PCI 体系结构概述

PCI (Peripheral Component Interconnect) 总线的诞生与 PC (Personal Computer) 的蓬勃发展密切相关。在处理器体系结构中，PCI 总线属于局部总线 (Local Bus)。局部总线作为系统总线的延伸，其主要功能是连接外部设备。

处理器主频的不断提升，要求速度更快、带宽更高的局部总线。起初 PC 使用 8 位的 XT 总线作为局部总线，很快升级到 16 位的 ISA (Industry Standard Architecture) 总线，并逐步发展到 32 位的 EISA (Extended Industry Standard Architecture)、VESA (Video Electronics Standards Association) 和 MCA (Micro Channel Architecture) 总线。

PCI 总线规范在 20 世纪 90 年代提出。这条总线推出之后，很快得到了各大主流半导体厂商的认同，并迅速统一了当时并存的各类局部总线，EISA、VESA 等其他 32 位总线很快就被 PCI 总线淘汰了。从那时起，PCI 总线一直在处理器体系结构中占有重要地位。

在此后相当长的一段时间里，处理器系统的大多数外部设备都是直接或者间接地与 PCI 总线相连。即使目前 PCI Express 总线逐步取代 PCI 总线成为 PC 局部总线的主流，也不能掩盖 PCI 总线的光芒。从软件层面上看，PCI Express 总线与 PCI 总线基本兼容；从硬件层面上看，PCI Express 总线在很大程度上继承了 PCI 总线的设计思路。因此 PCI 总线依然是软硬件工程师在进行处理器系统的开发与设计时必须掌握的一条局部总线。

PCI 总线规范由 Intel 的 IAL (Intel Architecture Lab) 提出，其 V1.0 规范在 1992 年 6 月 22 日正式发布。IAL 是 Intel 的一个重要实验室，USB (Universal Serial Bus)、AGP (Accelerated Graphics Port)、PCI Express 总线规范和 PC 的南北桥结构都是由这个实验室提出的。

IAL 起初的研究领域包括硬件和软件，但是 IAL 在软件领域的研究遭到了 Microsoft 的抵触，IAL 提出的许多软件规范并不被 Microsoft 认可，于是 IAL 更专注硬件领域，并在 PC 体系架构上取得了一个又一个突破。IAL 是现代 PC 体系架构的重要奠基者。2001 年，IAL 由于其创始人的离去而临时解散。2005 年，Intel 重建了这个实验室。

PCI 总线 V1.0 规范仅针对在一个 PCB (Printed Circuit Board) 环境内的器件之间的互连，而 1993 年 4 月 30 日发布的 V2.0 规范增加了对 PCI 插槽的支持。1995 年 6 月 1 日，PCI V2.1 总线规范发布，这个规范具有里程碑意义。正是这个规范使得 PCI 总线大规模普及，至此 PCI 总线完成了对(E)ISA 和 MCA 总线的替换。

至 1996 年，VESA 总线也逐渐离开了人们的视线，当然 PCI 总线并不能完全提供显卡所需要的带宽，真正替代 VESA 总线的是 AGP 总线。随后 PCISIG (PCI Special Interest Group) 陆续发布了 PCI 总线 V2.2、V2.3 规范，并最终将 PCI 总线规范定格在 V3.0。

除了 PCI 总线规范外，PCISIG 还定义了一些与 PCI 总线相关的规范，如 PCMCIA（Personal Computer Memory Card International Association）规范和 MiniPCI 规范。其中 PCMCIA 规范主要针对 Laptop 应用，后来 PCMCIA 升级为 PC Card（Cardbus）规范，而 PC Card 又升级为 ExpressCard 规范。

PC Card 规范基于 32 位、33MHz 的 PCI 总线；而 ExpressCard 规范基于 PCI Express 和 USB 2.0。这两个规范都在 Laptop 领域中获得了成功。除了 PCMCIA 规范外，Mini PCI 总线也非常流行，与标准 PCI 插槽相比，Mini PCI 插槽占用面积较小，适用于一些对尺寸有要求的应用。

除了以上规范之外，PCISIG 还推出了一系列和 PCI 总线直接相关的规范。如 PCI-to-PCI 桥规范、PCI 电源管理规范、PCI 热插拔规范和 CompactPCI 总线规范。其中 PCI-to-PCI 桥规范最为重要，理解 PCI-to-PCI 桥是理解 PCI 体系结构的基础；而 CompactPCI 总线规范多用于具有背板结构的大型系统，并支持热插拔。

PCISIG 在 PCI 总线规范的基础上，进一步提出 PCI-X 规范。与 PCI 总线相比，PCI-X 总线规范可以支持 133MHz、266MHz 和 533MHz 的总线频率，并在传送规则上做了一些改动。虽然 PCI-X 总线还没有得到大规模普及就被 PCI Express 总线替代，但是在 PCI-X 总线中提出的许多设计思想仍然被 PCI Express 总线继承。

PCI 总线规范是 Intel 对 PC 领域做出的一个巨大贡献。Intel 也在 PCI 总线规范中留下了深深的印记，PCI 总线规范的许多内容都与基于 IA（Intel Architecture）架构的 x86 处理器密切相关。但是这并不妨碍其他处理器系统使用 PCI 总线，事实上 PCI 总线在非 x86 处理器系统上也取得了巨大的成功。目前绝大多数处理器系统都使用 PCI/PCI Express 总线连接外部设备，特别是一些通用外设。

随着时间的推移，PCI 和 PCI-X 总线逐步遇到瓶颈。PCI 和 PCI-X 总线使用单端并行信号进行数据传递，由于单端信号容易被外部系统干扰，其总线频率很难进一步提高。目前，为了获得更高的总线频率以提高总线带宽，高速串行总线逐步替代了并行总线。PCI Express 总线也逐渐替代 PCI 总线成为主流。但是从系统软件的角度上看，PCI Express 总线仍然基于 PCI 总线。理解 PCI Express 总线的一个基础是深入理解 PCI 总线，同时 PCI Express 总线也继承了 PCI 总线的许多概念。本篇将详细介绍与处理器体系结构相关的一些必备的 PCI 总线知识。

为简化起见，本篇主要介绍 PCI 总线的 32 位地址模式。在实际应用中，使用 64 位地址模式的 PCI 设备非常少。而且在 PCI Express 总线逐渐取代 PCI 总线的大趋势之下，将来也很难会有更多的使用 64 位地址的 PCI 设备。如果读者需要掌握 PCI 总线的 64 位地址模式，请自行阅读 PCI 总线的相关规范。实际上，如果读者真正掌握了 PCI 总线的 32 位地址模式之后，理解 64 位地址模式并不困难。

为节省篇幅，下文将 PCI Express 总线简称为 PCIe 总线，PCI-to-PCI 桥简称为 PCI 桥，PCI Express-to-PCI 桥简称为 PCIe 桥，Host-to-PCI 主桥简称为 HOST 主桥。值得注意的是许多书籍将 HOST 主桥称为 PCI 主桥或者 PCI 总线控制器。

第1章 PCI总线的基本知识

PCI总线作为处理器系统的局部总线，其主要目的是为了连接外部设备，而不是作为处理器的系统总线连接Cache和主存储器。但是PCI总线、系统总线和处理器体系结构之间依然存在着紧密的联系。

PCI总线作为系统总线的延伸，其设计考虑了许多与处理器相关的内容，如处理器的Cache共享一致性和数据完整性（Data Consistency，也称为Memory Consistency），以及如何与处理器进行数据交换等一系列内容。其中Cache共享一致性和数据完整性是现代处理器局部总线的设计的重点和难点，也是本书将重点讲述的主题之一。

孤立地研究PCI总线并不可取，因为PCI总线仅是处理器系统的一个部分。深入理解PCI总线需要了解一些与处理器体系结构相关的知识。这些知识是本书侧重描述的，同时也是PCI总线规范忽略的内容。脱离实际的处理器系统，不容易也不可能深入理解PCI总线规范。

对于今天的读者来说，PCI总线提出的许多概念略显过时，也有许多不足之处。但是在当年，PCI总线与之前存在的其他并行局部总线如ISA、EISA和MCA总线相比，具有许多突出的优点，是一个全新的设计。

（1）PCI总线空间与处理器空间隔离

PCI设备具有独立的地址空间，即PCI总线地址空间，该空间与存储器地址空间通过HOST主桥隔离。处理器需要通过HOST主桥才能访问PCI设备，而PCI设备需要通过HOST主桥才能访问主存储器。在HOST主桥中含有许多缓冲，这些缓冲使得处理器总线与PCI总线工作在各自的时钟频率中，互不干扰。HOST主桥的存在也使得PCI设备和处理器可以方便地共享主存储器资源。

处理器访问PCI设备时，必须通过HOST主桥进行地址转换；而PCI设备访问主存储器时，也需要通过HOST主桥进行地址转换。HOST主桥的一个重要作用就是将处理器访问的存储器地址转换为PCI总线地址。PCI设备使用的地址空间是属于PCI总线域的，这与存储器地址空间不同。

x86处理器对PCI总线域与存储器域的划分并不明晰，这也使得许多程序员并没有准确地区分PCI总线域地址空间与存储器域地址空间。而本书将反复强调存储器地址和PCI总线地址的区别，因为这是理解PCI体系结构的重要内容。

PCI规范并没有对HOST主桥的设计进行约束。每一个处理器厂商使用的HOST主桥，其设计都不尽相同。HOST主桥是联系PCI总线与处理器的核心部件，掌握HOST主桥的实现机制是深入理解PCI体系结构的前提。

本书将以Freescale的PowerPC处理器和Intel的x86处理器为例，说明各自HOST主桥的实现方式，值得注意的是本书涉及的PowerPC处理器仅针对Freescale的PowerPC处理器，而不包含IBM的Power和AMCC的PowerPC处理器。而且如果没有特别说明，本书中涉及的x86处理器特指Intel的处理器，而不是其他厂商的x86处理器。

(2) 可扩展性

PCI 总线具有很强的扩展性。在 PCI 总线中，HOST 主桥可以直接推出一条 PCI 总线，这条总线也是该 HOST 主桥管理的第一条 PCI 总线，该总线还可以通过 PCI 桥扩展出一系列 PCI 总线，并以 HOST 主桥为根节点，形成 1 棵 PCI 总线树。这些 PCI 总线都可以连接 PCI 设备，但是在 1 棵 PCI 总线树上，最多只能挂接 256 个 PCI 设备（包括 PCI 桥）。

在同一条 PCI 总线上的设备间可以直接通信，而并不会影响其他 PCI 总线上设备间的数据通信。隶属于同一棵 PCI 总线树上的 PCI 设备，也可以直接通信，但是需要通过 PCI 桥进行数据转发。

PCI 桥是 PCI 总线的一个重要组成部件，该部件的存在使得 PCI 总线极具扩展性。PCI 桥也是有别于其他局部总线的一个重要部件。在“以 HOST 主桥为根节点”的 PCI 总线树中，每一个 PCI 桥下也可以连接一个 PCI 总线子树，PCI 桥下的 PCI 总线仍然可以使用 PCI 桥继续进行总线扩展。

PCI 桥可以管理这个 PCI 总线子树，PCI 桥的配置空间含有一系列管理 PCI 总线子树的配置寄存器。在 PCI 桥的两端，分别连接了两条总线，分别是上游总线（Primary Bus）和下游总线（Secondary Bus）。其中与处理器距离较近的总线被称为上游总线，另一条被称为下游总线。这两条总线间的通信需要通过 PCI 桥进行。PCI 桥中的许多概念被 PCIe 总线采纳，理解 PCI 桥也是理解 PCIe 体系结构的基础。

(3) 动态配置机制

PCI 设备使用的地址可以根据需要由系统软件动态分配。PCI 总线使用这种方式合理地解决了设备间的地址冲突，从而实现了“即插即用”功能。因此 PCI 总线不需要使用 ISA 或者 EISA 接口卡为解决地址冲突而使用的硬件跳线。

每一个 PCI 设备都有独立的配置空间，在配置空间中含有该设备在 PCI 总线中使用的基址，系统软件可以动态配置这个基址，从而保证每一个 PCI 设备使用的物理地址并不相同。PCI 桥的配置空间中含有其下 PCI 子树所能使用的地址范围。

(4) 总线带宽

PCI 总线与之前的局部总线相比，极大提高了数据传送带宽，32 位/33 MHz 的 PCI 总线可以提供 132 MB/s 的峰值带宽，而 64 位/66 MHz 的 PCI 总线可以提供的峰值带宽为 532 MB/s。虽然 PCI 总线所能提供的峰值带宽远不能和 PCIe 总线相比，但是与之前的局部总线 ISA、EISA 和 MCA 总线相比，仍然具有极大的优势。

ISA 总线的最高主频为 8 MHz，位宽为 16，其峰值带宽为 16 MB/s；EISA 总线的最高主频为 8.33 MHz，位宽为 32，其峰值带宽为 33 MB/s；而 MCA 总线的最高主频为 10 MHz，最高位宽为 32，其峰值带宽为 40 MB/s。PCI 总线提供的峰值带宽远高于这些总线。

(5) 共享总线机制

PCI 设备通过仲裁获得 PCI 总线的使用权后，才能进行数据传送，在 PCI 总线上进行数据传送，并不需要处理器进行干预。

PCI 总线仲裁器不在 PCI 总线规范定义的范围内，也不一定是 HOST 主桥和 PCI 桥的一部分。虽然绝大多数 HOST 主桥和 PCI 桥都包含 PCI 总线仲裁器，但是在某些处理器系统的设计中也可以使用独立的 PCI 总线仲裁器。如在 PowerPC 处理器的 HOST 主桥中含有 PCI 总线仲裁器，但是用户可以关闭这个总线仲裁器，而使用独立的 PCI 总线仲裁器。

PCI 设备使用共享总线方式进行数据传递，在同一条总线上，所有 PCI 设备共享同一总线带宽，这将极大地影响 PCI 总线的利用率。这种机制显然不如 PCIe 总线采用的交换结构，但是在 PCI 总线盛行的年代，半导体的工艺、设计能力和制作成本决定了采用共享总线方式是当时的最优选择。

(6) 中断机制

PCI 总线上的设备可以通过四根中断请求信号 INTA ~ D# 向处理器提交中断请求。与 ISA 总线上的设备不同，PCI 总线上的设备可以共享这些中断请求信号，不同的 PCI 设备可以将这些中断请求信号“线与”后，与中断控制器的中断请求引脚连接。PCI 设备的配置空间记录了该设备使用这四根中断请求信号的信息。

PCI 总线还进一步提出了 MSI (Message Signal Interrupt) 机制，该机制使用存储器写总线事务传递中断请求，并可以使用 x86 处理器 FSB (Front Side Bus) 总线提供的 Interrupt Message 总线事务，从而提高了 PCI 设备的中断请求效率。

虽然从现代总线技术的角度来看，PCI 总线仍有许多不足之处，但也不能否认 PCI 总线已经获得了巨大的成功。不仅 x86 处理器将 PCI 总线作为标准的局部总线连接各类外部设备，PowerPC、MIPS 和 ARM^①处理器也将 PCI 总线作为标准局部总线。除此之外，基于 PCI 总线的外部设备，如以太网控制器、声卡、硬盘控制器等，也已经成为主流。

1.1 PCI 总线的组成结构

如上文所述，PCI 总线作为处理器系统的局部总线，是处理器系统的一个组成部件，讲述 PCI 总线的组成结构不能离开处理器系统这个大环境。在一个处理器系统中，与 PCI 总线相关的模块如图 1-1 所示。

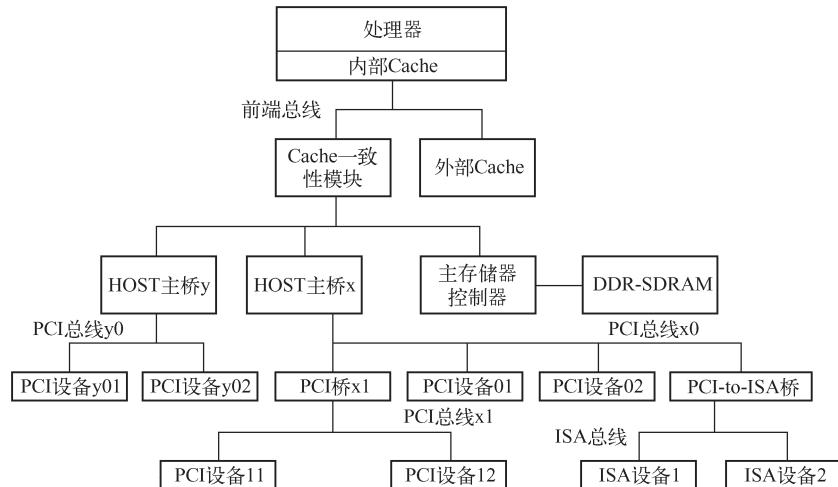


图 1-1 基于 PCI 总线的处理器系统

^① 在 ARM 处理器中，使用 SoC 平台总线，即 AMBA 总线，连接片内设备。但是某些 ARM 生产厂商，依然使用 AMBA-to-PCI 桥推出 PCI 总线，以连接 PCI 设备。

图中与 PCI 总线相关的模块包括：HOST 主桥、PCI 总线、PCI 桥和 PCI 设备。PCI 总线由 HOST 主桥和 PCI 桥推出，HOST 主桥与主存储器控制器在同一级总线上，因此 PCI 设备可以方便地通过 HOST 主桥访问主存储器，即进行 DMA 操作。

值得注意的是，PCI 设备的 DMA 操作需要与处理器系统的 Cache 进行一致性操作，当 PCI 设备通过 HOST 主桥访问主存储器时，Cache 一致性模块将进行地址监听，并根据监听的结果改变 Cache 的状态。

在一些简单的处理器系统中，可能不含有 PCI 桥，此时所有 PCI 设备都是连接在 HOST 主桥推出的 PCI 总线上。此外在一些处理器系统中可能含有多个 HOST 主桥，如图 1-1 所示的处理器系统中含有 HOST 主桥 x 和 HOST 主桥 Y。

1.1.1 HOST 主桥

HOST 主桥是一个很特别的桥片，其主要功能是隔离处理器系统的存储器域与处理器系统的 PCI 总线域，管理 PCI 总线域，并完成处理器与 PCI 设备间的数据交换。处理器与 PCI 设备间的数据交换主要由“处理器访问 PCI 设备的地址空间”和“PCI 设备使用 DMA 机制访问主存储器”这两部分组成。

为简便起见，下面将处理器系统的存储器域简称为存储器域，而将处理器系统的 PCI 总线域称为 PCI 总线域，存储器域和 PCI 总线域的详细介绍见第 2.1 节。值得注意的是，在一个处理器系统中，有几个 HOST 主桥，就有几个 PCI 总线域。

HOST 主桥在处理器系统中的位置并不相同，如 PowerPC 处理器将 HOST 主桥与处理器集成在一个芯片中。而有些处理器不进行这种集成，如 x86 处理器使用南北桥结构，处理器内核在一个芯片中，而 HOST 主桥在北桥中。但是从处理器体系结构的角度看，这些集成方式并不重要。

PCI 设备通过 HOST 主桥访问主存储器时，需要与处理器的 Cache 进行一致性操作，因此在设计 HOST 主桥时需要重点考虑 Cache 一致性操作。在 HOST 主桥中，还含有许多数据缓冲，以支持 PCI 总线的预读机制。

HOST 主桥是联系处理器与 PCI 设备的桥梁。在一个处理器系统中，每一个 HOST 主桥都管理了一棵 PCI 总线树，在同一棵 PCI 总线树上的所有 PCI 设备属于同一个 PCI 总线域。如图 1-1 所示，HOST 主桥 x 之下的 PCI 设备属于 PCI 总线 x 域，而 HOST 主桥 y 之下的 PCI 设备属于 PCI 总线 y 域。在这棵总线树上的所有 PCI 设备的配置空间都由 HOST 主桥通过配置读写总线周期访问。

如果 HOST 主桥支持 PCI V3.0 规范的 Peer-to-Peer 数据传送方式，那么分属不同 PCI 总线域的 PCI 设备可以直接进行数据交换。如图 1-1 所示，如果 HOST 主桥 y 支持 Peer-to-Peer 数据传送方式，PCI 设备 y01 可以直接访问 PCI 设备 01 或者 PCI 设备 11，而不需要通过处理器的参与。但是这种跨越总线域的数据传送方式在 PC 架构中并不常用，在 PC 架构中，重点考虑的是 PCI 设备与主存储器之间的数据交换，而不是 PCI 设备之间的数据交换。此外在 PC 架构中，具有两个 HOST 主桥的处理器系统也并不多见。

在 PowerPC 处理器中，HOST 主桥可以通过设置 Inbound 寄存器，使得分属于不同 PCI 总线域的设备可以直接通信。许多 PowerPC 处理器都具有多个 HOST 主桥，有关 PowerPC 处理器使用的 HOST 主桥详见第 2.2 节。

1.1.2 PCI 总线

在处理器系统中，含有 PCI 总线和 PCI 总线树这两个概念。这两个概念并不相同，在一棵 PCI 总线树中可能具有多条 PCI 总线，而具有血缘关系的 PCI 总线组成一棵 PCI 总线树。如在图 1-1 所示的处理器系统中，PCI 总线 x 树具有两条 PCI 总线，分别为 PCI 总线 x0 和 PCI 总线 x1。而 PCI 总线 y 树中仅有一条 PCI 总线。

PCI 总线由 HOST 主桥或者 PCI 桥管理，用来连接各类设备，如声卡、网卡和 IDE 接口卡等。在一个处理器系统中，可以通过 PCI 桥扩展 PCI 总线，并形成具有血缘关系的多级 PCI 总线，从而形成 PCI 总线树型结构。在处理器系统中有几个 HOST 主桥，就有几棵这样的 PCI 总线树，而每一棵 PCI 总线树都与一个 PCI 总线域对应。

与 HOST 主桥直接连接的 PCI 总线通常被命名为 PCI 总线 0。考虑到在一个处理器系统中可能有多个主桥，图 1-1 将 HOST 主桥 x 推出的 PCI 总线命名为 x0 总线，而将 PCI 桥 x1 扩展出的 PCI 总线称为 x1 总线，将 HOST 主桥 y 推出的 PCI 总线称为 y0 ~ yn。分属不同 PCI 总线树的设备，其使用的 PCI 总线地址空间分属不同的 PCI 总线域空间。

1.1.3 PCI 设备

在 PCI 总线中有三类设备：PCI 主设备、PCI 从设备和桥设备。其中 PCI 从设备只能被动地接收来自 HOST 主桥或者其他 PCI 设备的读写请求；而 PCI 主设备可以通过总线仲裁获得 PCI 总线的使用权，主动地向其他 PCI 设备或者主存储器发起存储器读写请求。而桥设备的主要作用是管理下游的 PCI 总线，并转发上下游总线之间的总线事务。

一个 PCI 设备可以既是主设备也是从设备，但是在同一个时刻，这个 PCI 设备或者为主设备或者为从设备。PCI 总线规范将 PCI 主从设备统称为 PCI Agent 设备。在处理器系统中常见的 PCI 网卡、显卡、声卡等设备都属于 PCI Agent 设备。

在 PCI 总线中，HOST 主桥是一个特殊的 PCI 设备，该设备可以获取 PCI 总线的控制权访问 PCI 设备，也可以被 PCI 设备访问。但是 HOST 主桥并不是 PCI 设备。PCI 规范也没有规定如何设计 HOST 主桥。

在 PCI 总线中，还有一类特殊的设备，即桥设备。它包括 PCI 桥、PCI-to-(E)ISA 桥和 PCI-to-Cardbus 桥。本书重点介绍 PCI 桥，而不介绍其他桥设备的实现原理。PCI 桥的存在使 PCI 总线极具扩展性，处理器系统可以使用 PCI 桥进一步扩展 PCI 总线。

PCI 桥的出现使得采用 PCI 总线进行大规模系统互连成为可能。但是在目前已经实现的大规模处理器系统中，并没有使用 PCI 总线进行处理器系统与处理器系统之间的大规模互连。因为 PCI 总线是一个以 HOST 主桥为根的树型结构，使用主从架构，因而不易实现多处理器系统间的对等互连。

即便如此 PCI 桥仍然是 PCI 总线规范的精华所在，掌握 PCI 桥是深入理解 PCI 体系结构的基础。PCI 桥可以连接两条 PCI 总线，上游 PCI 总线和下游 PCI 总线，这两个 PCI 总线属于同一个 PCI 总线域，使用 PCI 桥扩展的所有 PCI 总线都同属于一个 PCI 总线域。

其中对 PCI 设备配置空间的访问可以从上游总线转发到下游总线，而数据传送可以双方向进行。在 PCI 总线中，还存在一种非透明 PCI 桥，该桥片不是 PCI 总线规范定义的标准桥

片，但是适用于某些特殊应用，在第 2.5 节中将详细介绍这种桥片。在本书中，如不特别强调，PCI 桥是指透明桥，透明桥也是 PCI 总线规范定义的标准桥片。

PCI-to-(E)ISA 桥和 PCI-to-Cardbus 桥的主要作用是通过 PCI 总线扩展(E)ISA 和 Cardbus 总线。在 PCI 总线推出之后，(E)ISA 总线并没有在处理器系统中立即消失，此时需要使用 PCI-(E)ISA 桥扩展(E)ISA 总线，而使用 PCI-to-Cardbus 桥用来扩展 Cardbus 总线。本书并不关心(E)ISA 和 Cardbus 总线的设计与实现。

1.1.4 HOST 处理器

PCI 总线规定在同一时刻内，在一棵 PCI 总线树上有且只有一个 HOST 处理器。这个 HOST 处理器可以通过 HOST 主桥，发起 PCI 总线的配置请求总线事务，并对 PCI 总线上的设备和桥片进行配置。

在 PCI 总线中，HOST 处理器是一个较为模糊的概念。在 SMP (symmetric multiprocessing) 处理器系统中，所有 CPU 都可以通过 HOST 主桥访问其下的 PCI 总线树，这些 CPU 都可以作为 HOST 处理器。但是值得注意的是，PCI 总线树的实际管理者是 HOST 主桥，而不是 HOST 处理器。

在 HOST 主桥中，设置了许多寄存器，HOST 处理器通过操作这些寄存器来管理 PCI 设备。如在 x86 处理器的 HOST 主桥中设置了 0xCF8 和 0xCFC 这两个 I/O 端口访问 PCI 设备的配置空间，而 PowerPC 处理器的 HOST 主桥设置了 CFG_ADDR 和 CFG_DATA 寄存器访问 PCI 设备的配置空间。值得注意的是，在 PowerPC 处理器中并没有 I/O 端口，因此使用存储器映像寻址方式访问外部设备的寄存器空间。

1.1.5 PCI 总线的负载

PCI 总线能挂接的负载与总线频率相关，其中总线频率越高，能挂接的负载越少。下面以 PCI 总线和 PCI-X 总线为例说明总线频率、峰值带宽和负载能力之间的关系，如表 1-1 所示。

表 1-1 PCI 总线频率、带宽与负载之间的关系

总线类型	总线频率/MHz	峰值带宽/(MB/s)	负载能力
PCI	33	133	4~5 个插槽
	66	266	1~2 个插槽
PCI-X	66	266	4 个插槽
	133	533	2 个插槽
	266	1066	1 个插槽
	533	2131	1 个插槽

由表 1-1 可知，PCI 总线频率越高，能挂接的负载越少，但是整条总线能提供的带宽越大。值得注意的是，PCI-X 总线与 PCI 总线的传送协议略有不同，因此 66 MHz 的 PCI-X 总线的负载数较大。PCI-X 总线的详细说明见第 1.5 节。当 PCI-X 总线频率为 266 MHz 和 533 MHz 时，该总线只能挂接一个 PCI-X 插槽。在 PCI 总线中，一个插槽相当于两个负载，接插件和插卡各算为一个负载。在表 1-1 中，33 MHz 的 PCI 总线可以挂接 4~5 个插槽，相当于

直接挂接 8 ~ 10 个负载。

1.2 PCI 总线的信号定义

PCI 总线是一条共享总线，在一条 PCI 总线上可以挂接多个 PCI 设备。这些 PCI 设备通过一系列信号与 PCI 总线相连，这些信号由地址/数据信号、控制信号、仲裁信号、中断信号等多种信号组成。

PCI 总线是一个同步总线，每一个设备都具有一个 CLK 信号，其发送设备与接收设备使用这个 CLK 信号进行同步数据传递。PCI 总线可以使用 33 MHz 或者 66 MHz 的时钟频率，而 PCI-X 总线可以使用 133 MHz、266 MHz 或者 533 MHz 的时钟频率。

除了 RST#、INTA ~ D#、PME# 和 CLKRUN# 等信号之外，PCI 设备使用的绝大多数信号都使用这个 CLK 信号进行同步。其中 RST# 是复位信号，而 PCI 设备使用 INTA ~ D# 信号进行中断请求。本书并不详细介绍 PME# 和 CLKRUN# 信号。

1.2.1 地址和数据信号

在 PCI 总线中，与地址和数据相关的信号如下所示。

(1) AD[31:0] 信号

PCI 总线复用地址与数据信号。PCI 总线事务在启动后的第一个时钟周期传送地址，这个地址是 PCI 总线域的存储器地址或者 I/O 地址；而在下一个时钟周期传送数据^①。传送地址的时钟周期也被称为地址周期，而传送数据的时钟周期也被称为数据周期。PCI 总线支持突发传送，即在一个地址周期之后，可以紧跟多个数据周期。

(2) PAR 信号

PCI 总线使用奇偶校验机制，保证地址和数据信号在进行数据传递时的正确性。PAR 信号是 AD[31:0] 和 C/BE[3:0] 的奇偶校验信号。PCI 主设备在地址周期和数据周期中，使用该信号为地址和数据信号线提供奇偶校验位。

(3) C/BE[3:0]# 信号

PCI 总线复用命令与字节选通引脚。在地址周期中，C/BE[3:0]# 信号表示 PCI 总线的命令。而在数据周期中，C/BE[3:0]# 引脚输出字节选通信号，其中 C/BE3#、C/BE2#、C/BE1# 和 C/BE0# 与数据的字节 3、2、1 和 0 对应。使用这组信号可以对 PCI 设备进行单个字节、字和双字访问。PCI 总线通过 C/BE[3:0]# 信号定义了多个总线事务，这些总线事务如表 1-2 所示。

表 1-2 PCI 总线事务

C/BE[3:0]#	命 令 类 型	说 明
0000	Interrupt Acknowledge	中断响应总线事务读取当前挂接在 PCI 总线上的中断控制器的中断向量号。目前大多数处理器系统的中断控制器都不挂接在 PCI 总线上，因此这种总线事务很少使用

① 双地址周期在第一、二个时钟周期，都传送地址。

(续)

C/BE[3:0]#	命 令 类 型	说 明
0001	Special Cycle	HOST 主桥可以使用 Special Cycle 事务在 PCI 总线上进行信息广播
0010	I/O Read	HOST 主桥可以使用该总线事务对 PCI 设备的 I/O 地址空间进行读操作。目前多数 PCI 设备都不支持 I/O 地址空间，而仅支持存储器地址空间，但是仍有部分 PCI 设备同时包含 I/O 地址空间和存储器地址空间
0011	I/O Write	对 PCI 总线的 I/O 地址空间进行写操作
0100	Reserved	保留
0101	Reserved	保留
0110	Memory Read	HOST 主桥可以使用该总线事务对 PCI 设备的存储器空间进行读操作。PCI 设备也可以使用该总线事务读取处理器的存储器空间
0111	Memory Write	HOST 主桥可以使用该总线事务对 PCI 设备的存储器空间进行写操作。PCI 设备也可以使用该总线事务向处理器的存储器空间进行写操作
1000	Reserved	保留
1001	Reserved	保留
1010	Configuration Read	HOST 主桥可以对 PCI 设备的配置空间进行读操作。每一个 PCI 设备都有独立的配置空间。在多功能 PCI 设备中，每一个子设备（Function）也有一个独立的配置空间。该总线事务只能由 HOST 主桥发出，PCI 桥可以转发该总线事务
1011	Configuration Write	HOST 主桥对 PCI 设备的配置空间进行写操作
1100	Memory Read Multiple	HOST 主桥可以使用该总线事务对 PCI 设备的存储器空间进行多行读操作，这种操作并不多见。该总线事务的主要用途是供 PCI 设备使用，读取主存储器。这个读操作与 Memory Read 操作（C/BE [3: 0] 为 0x0110 时）略有不同，详见第 3.4.5 节
1101	Dual Address Cycle	PCI 总线支持 64 位地址，处理器或者其他 PCI 设备访问 64 位 PCI 总线地址时，必须使用双地址周期产生 64 位的 PCI 总线地址。PCI 设备使用 DMA 读写方式访问 64 位的存储器地址时，也可以使用该总线事务
1110	Memory Read Line	HOST 主桥可以使用该总线事务对 PCI 设备的存储器空间进行单行读操作，这种操作并不多见。该总线事务的主要用途是供 PCI 设备使用，读取主存储器。详见第 3.4.5 节
1111	Memory Write and Invalidate	存储器写并无效操作，与存储器写不同，PCI 设备可以使用该总线事务对主存储器空间进行写操作。该总线事务将数据写入主存储器的同时，将对应 Cache 行中的数据“使无效”，详见第 3.3.4 节

1.2.2 接口控制信号

在 PCI 总线中，接口控制信号的主要作用是保证数据的正常传递，并根据 PCI 主从设备的状态，暂停、终止或者正常完成当前总线事务，其主要信号如下。

(1) FRAME#信号

该信号指示一个 PCI 总线事务的开始与结束。当 PCI 设备获得总线的使用权后，将置该信号有效，即置为低，启动 PCI 总线事务，当结束总线事务时，将置该信号无效，即置为高。PCI 设备（包括 HOST 主桥）只有通过仲裁获得当前 PCI 总线的使用权后，才能驱动该信号。

(2) IRDY#信号

该信号由 PCI 主设备（包括 HOST 主桥）驱动，该信号有效时表示 PCI 主设备的数据已经准备完毕。如果当前 PCI 总线事务为写事务，表示数据已经在 AD[31:0] 上有效；如果为读事务，表示 PCI 目标设备已经准备好接收缓冲，目标设备可以将数据发送到 AD[31:0] 上。

(3) TRDY#信号

该信号由目标设备驱动，该信号有效时表示目标设备已经将数据准备完毕。如果当前 PCI 总线事务为写事务，表示目标设备已经准备好接收缓冲，可以将 AD[31:0] 上的数据写入目标设备；如果为读事务，表示 PCI 设备需要的数据已经在 AD[31:0] 上有效。

该信号可以和 IRDY# 信号联合使用，在 PCI 总线事务上插入等待周期，对 PCI 总线的数据传送进行控制。

(4) STOP#信号

该信号有效时表示目标设备请求主设备停止当前 PCI 总线事务。一个 PCI 总线事务除了可以正常结束外，目标设备还可以使用该信号终止当前 PCI 总线事务。目标设备可以根据不同的情况，要求主设备对当前 PCI 总线事务进行重试（Retry）、断连（Disconnect），也可以向主设备报告目标设备夭折（Target Abort）。

目标设备要求主设备 Retry 和 Disconnect 并不意味着当前 PCI 总线事务出现错误。当目标设备没有将数据准备好时，可以使用 Retry 周期使主设备重试当前 PCI 总线事务。有时目标设备不能接收来自主设备较长的 Burst 操作时，可以使用 Disconnect 周期，将一个较长的 Burst 操作分解为多个 Burst 操作。当主设备访问的地址越界时，目标设备可以使用 Disconnect 周期，终止主设备的越界访问。

而 Target Abort 表示在数据传送中出现错误。处理器系统必须对这种情况进行处理。在 PCI 总线中，出现 Abort 一般意味着当前 PCI 总线域出现了较为严重的错误。

(5) IDSEL 信号

PCI 总线在进行配置读写总线事务时，使用该信号选择 PCI 目标设备。配置读写总线事务与存储器读写总线事务在实现上略有不同。在 PCI 总线中，存储器读写总线事务使用地址译码方式访问外部设备。而配置读写总线事务使用“ID 译码方式”访问 PCI 设备，即通过 PCI 设备的总线号、设备号和寄存器号访问 PCI 设备的配置空间。

IDSEL 信号与 PCI 设备的设备号相关，相当于 PCI 设备配置空间的片选信号，这部分内容将在第 2.4.4 节中详细介绍。

(6) DEVSEL#信号

该信号有效时表示 PCI 总线的目标设备准备好，该信号与 TRDY# 信号的不同之处在于该信号有效仅表示目标设备已经完成了地址译码。目标设备使用该信号通知 PCI 主设备，其访问对象在当前 PCI 总线上，但是并不表示目标设备可以与主设备进行数据交换。而 TRDY# 信号表示数据有效，PCI 主设备可以向目标设备写入或者从目标设备读取数据。

PCI 总线规范根据设备的译码速度，将 PCI 设备分为快速、中速和慢速三种。在 PCI 总线上还有一种特殊的设备，即负向译码设备，在一条 PCI 总线上当快速、中速和慢速三种设备都不能响应 PCI 总线事务的地址时，负向译码设备将被动地接收这个 PCI 总线事务。如果在 PCI 主设备访问的 PCI 总线上，没有任何设备可以置 DEVSEL#信号为有效，主设备将使用 Master Abort 周期结束当前总线事务。

(7) LOCK#信号

PCI 主设备可以使用该信号，将目标设备的某个存储器或者 I/O 资源锁定，以禁止其他 PCI 主设备访问此资源，直到锁定这个资源的主设备将其释放。PCI 总线使用 LOCK#信号实现 LOCK 总线事务，只有 HOST 主桥、PCI 桥或者其他桥片可以使用 LOCK#信号。在 PCI 总线的早期版本中，PCI Agent 设备也可以使用 LOCK#信号，而目前 PCI 总线使用 LOCK#信号仅是为防止死锁和向前兼容。LOCK 总线事务将严重影响 PCI 总线的传送效率，在实际应用中，设计者应当尽量避免使用该总线事务。

1.2.3 仲裁信号

PCI 设备使用该组信号进行总线仲裁，并获得 PCI 总线的使用权。只有 PCI 主设备需要使用该组信号，而 PCI 从设备可以不使用总线仲裁信号。这组信号由 REQ#和 GNT#组成。其中 PCI 主设备的 REQ#和 GNT#信号与 PCI 总线的仲裁器直接相连。

PCI 主设备的总线仲裁信号与 PCI 总线仲裁器的连接关系如图 1-2 所示。值得注意的是，每一个 PCI 主设备都具有独立的总线仲裁信号，并与 PCI 总线仲裁器一一相连。而总线仲裁器需要保证在同一个时间段内，只有一个 PCI 设备可以使用当前总线。

在一个处理器系统中，一条 PCI 总线可以挂接 PCI 主设备的数目，除了与负载能力相关之外，还与 PCI 总线仲裁器能够提供的仲裁信号数目直接相关。

在一棵 PCI 总线树中，每一条 PCI 总线上都有一个总线仲裁器。一个处理器系统可以使用 PCI 桥扩展出一条新的 PCI 总线，这条新的 PCI 总线也需要一个总线仲裁器，通常在 PCI 桥中集成了这个总线仲裁器。多数 HOST 主桥也集成了一个 PCI 总线仲裁器，但是 PCI 总线也可以使用独立的 PCI 总线仲裁器。

PCI 主设备使用 PCI 总线进行数据传递时，需要首先置 REQ#信号有效，向 PCI 总线仲裁器发出总线申请，当 PCI 总线仲裁器允许 PCI 主设备获得 PCI 总线的使用权后，将置 GNT#信号为有效，并将其发送给指定的 PCI 主设备。而 PCI 主设备在获得总线使用权之后，将可以置 FRAME#信号有效，与 PCI 从设备进行数据通信。

1.2.4 中断请求等其他信号

PCI 总线提供了 INTA#、INTB#、INTC#和 INTD#四个中断请求信号，PCI 设备借助这些中断请求信号，使用电平触发方式向处理器提交中断请求。当这些中断请求信号为低时，PCI 设备将向处理器提交中断请求；当处理器执行中断服务程序清除 PCI 设备的中断请求

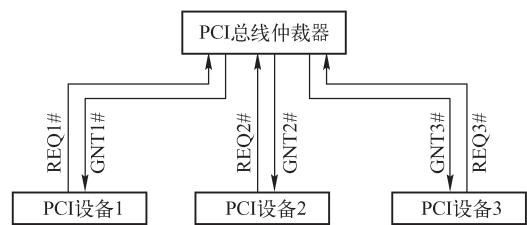


图 1-2 PCI 设备与总线仲裁器的连接关系

后，PCI设备将该信号置高^①，结束当前中断请求。

PCI总线规定单功能设备只能使用INTA#信号，而多功能设备才能使用INTB#/C#/D#信号。PCI设备的这些中断请求信号可以通过某种规则进行线与，之后与中断控制器的中断请求信号线相连。而处理器系统需要预先知道这个规则，以便正确处理来自不同PCI设备的中断请求，这个规则也被称为中断路由表，有关中断路由表的详细描述见第1.4.2节。

PCI总线在进行数据传递过程时，难免会出现各种各样的错误，因此PCI总线提供了一些错误信号，如PERR#和SERR#信号。其中当PERR#信号有效时，表示数据传送过程中出现奇偶校验错（Special Cycle周期除外）；而当SERR#信号有效时，表示当前处理器系统出现了三种错误可能，分别为地址奇偶校验错、在Special Cycle周期中出现数据奇偶校验错、系统出现其他严重错误。

如果PCI总线支持64位模式，还需要提供AD[63:32]、C/BE[7:4]、REQ64、ACK64和PAR64这些信号。此外PCI总线还有一些与JTAG、SMBCLK以及66MHz使能相关的信号，本章并不介绍这些信号。

1.3 PCI总线的存储器读写总线事务

总线的基本任务是实现数据传送，将一组数据从一个设备传送到另一个设备，当然总线也可以将一个设备的数据广播到多个设备。在处理器系统中，这些数据传送都要依赖一定的规则，PCI总线也不例外。

PCI总线使用单端并行数据线，采用地址译码方式进行数据传递，而采用ID译码方式进行配置信息的传递。其中地址译码方式使用地址信号，而ID译码方式使用PCI设备的ID号，包括Bus Number、Device Number、Function Number和Register Number。下面将以图1-1中的处理器系统为例，简要介绍PCI总线支持的总线事务及其传送方式。

由表1-2可知，PCI总线支持多种总线事务。本节重点介绍存储器读写总线事务与I/O读写总线事务，并在第2.4节详细介绍配置读写总线事务。值得注意的是，PCI设备只有在系统软件初始化配置空间之后，才能够被其他主设备访问。

当PCI设备的配置空间被初始化之后，该设备在当前的PCI总线树上将拥有一个独立的PCI总线地址空间，即BAR（Base Address Register）寄存器所描述的空间，有关BAR寄存器的详细说明见第2.3.2节。

处理器与PCI设备进行数据交换，或者PCI设备之间进行存储器数据交换时，都将通过PCI总线地址完成。而PCI设备与主存储器进行DMA操作时，使用的也是PCI总线域的地址，而不是存储器域的地址，此时HOST主桥将完成PCI总线地址到存储器域地址的转换，不同的HOST主桥进行地址转换时使用的方法并不相同。

PCI总线的配置读写总线事务与HOST主桥与PCI桥相关，因此读者需要了解HOST主桥和PCI桥的详细实现机制之后，才能深入理解这部分内容。在第2.4节将详细介绍这些内容。在下文中，假定所使用的PCI设备的配置空间已经被系统软件初始化。

PCI总线支持以下几类存储器读写总线事务。

^① INTx#这组信号为开漏输出，当所有的驱动源不驱动该信号时，该信号由上拉电阻驱动为高。

(1) HOST 处理器对 PCI 设备的 BAR 空间进行数据读写，BAR 空间可以使用存储器或者 I/O 译码方式。HOST 处理器使用 PCI 总线的存储器读写总线事务和 I/O 读写总线事务访问 PCI 设备的 BAR 空间。

(2) PCI 设备之间的数据传递。在 PCI 总线上的两个设备可以直接通信，如一个 PCI 设备可以访问另外一个设备的 BAR 空间。不过这种数据传递在 PC 处理器系统中较少使用。

(3) PCI 设备对主存储器进行读写，即 DMA 读写操作。DMA 读写操作在所有处理器系统中都较为常用，也是 PCI 总线数据传送的重点。在多数情况下，DMA 读写操作结束后将伴随着中断的产生。PCI 设备可以使用 INTA#、INTB#、INTC# 和 INTD# 信号提交中断请求，也可以使用 MSI 机制提交中断请求。

1.3.1 PCI 总线事务的时序

PCI 总线使用第 1.2 节所述的信号进行数据和配置信息的传递，一个 PCI 总线事务的基本访问时序如图 1-3 所示，与 PCI 总线事务相关的控制信号有 FRAME#、IRDY#、TRDY#、DEVSEL# 等其他信号。

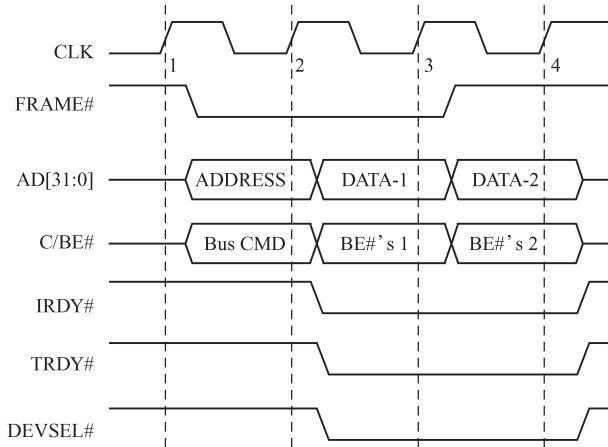


图 1-3 PCI 总线事务的时序

当一个 PCI 主设备需要使用 PCI 总线时，首先需要发送 REQ# 信号，通过总线仲裁获得总线使用权，即 GNT# 信号有效后，使用以下步骤完成一个完整 PCI 总线事务，对目标设备进行存储器或者 I/O 地址空间的读写访问。

(1) 当 PCI 主设备获得总线使用权之后，将在 CLK1 的上升沿置 FRAME# 信号有效，启动 PCI 总线事务。当 PCI 总线事务结束后，FRAME# 信号将被置为无效。

(2) PCI 总线周期的第一个时钟周期（CLK1 的上升沿到 CLK2 的上升沿之间）为地址周期。在地址周期中，PCI 主设备将访问的目的地址和总线命令分别驱动到 AD[31:0] 和 C/BE# 信号上。如果当前总线命令是配置读写，那么 IDSEL 信号线也被置为有效，IDSEL 信号与 PCI 总线的 AD[31:11] 相连，详见第 2.4.4 节。

(3) 当 IRDY#、TRDY# 和 DEVSEL# 信号都有效后，总线事务将使用数据周期进行数据传递。当 IRDY# 和 TRDY# 信号没有同时有效时，PCI 总线不能进行数据传递，PCI 总线使用这两个信号进行传送控制。

(4) PCI 总线支持突发周期，因此在地址周期之后可以有多个数据周期，可以传送多组数据。而目标设备并不知道突发周期的长度，如果目标设备不能继续接收数据时，可以 disconnect（断连）当前总线事务。值得注意的是，只有存储器读写总线事务可以使用突发周期。

一个完整的 PCI 总线事务远比上述过程复杂得多，因为 PCI 总线还支持许多传送方式，如双地址周期、fast back-to-back（快速背靠背）、插入等待状态、重试和断连、总线上的错误处理等一系列总线事务。本书不一一介绍这些传送方式。

1.3.2 Posted 和 Non-Posted 传送方式

PCI 总线规定了两类数据传送方式，分别是 Posted 和 Non-Posted 数据传送方式。其中使用 Posted 数据传送方式的总线事务也被称为 Posted 总线事务；而使用 Non-Posted 数据传送方式的总线事务也被称为 Non-Posted 总线事务。

其中 Posted 总线事务指 PCI 主设备向 PCI 目标设备进行数据传递时，当数据到达 PCI 桥后，即由 PCI 桥接管来自上游总线的总线事务，并将其转发到下游总线。采用这种数据传送方式，在数据还没有到达最终的目的地之前，PCI 总线就可以结束当前总线事务，从而在一定程度上解决了 PCI 总线的拥塞问题。

而 Non-Posted 总线事务是指 PCI 主设备向 PCI 目标设备进行数据传递时，数据必须到达最终目的地之后，才能结束当前总线事务的一种数据传递方式。

显然采用 Posted 传送方式，当这个 Posted 总线事务通过某条 PCI 总线后，就可以释放 PCI 总线的资源；而采用 Non-Posted 传送方式，PCI 总线在没有结束当前总线事务时必须等待。这种等待将严重阻塞当前 PCI 总线上的其他数据传送，因此 PCI 总线使用 Delayed 总线事务处理 Non-Posted 数据请求，使用 Delayed 总线事务可以相对缓解 PCI 总线的拥塞。Delayed 总线事务的详细介绍见第 1.3.5 节。

PCI 总线规定只有存储器写请求（包括存储器写并无效请求）可以采用 Posted 总线事务，下文将 Posted 存储器写请求简称为 PMW（Posted Memory Write），而存储器读请求、I/O 读写请求、配置读写请求只能采用 Non-Posted 总线事务。

下面以图 1-1 的处理器系统中的 PCI 设备 11 向存储器进行 DMA 写操作为例，说明 Posted 传送方式的实现过程。PCI 设备 11 进行 DMA 写操作时使用存储器写总线事务，当 PCI 设备 11 获得 PCI 总线 x1 的使用权后，将发送存储器写总线事务到 PCI 总线 x1。当 PCI 桥 1 发现这个总线事务的地址不在该桥管理的地址范围内将首先接收这个总线事务，并结束 PCI 总线 x1 的总线事务。

此时 PCI 总线 x1 使用的资源已被释放，PCI 设备 11 和 PCI 设备 12 可以使用 PCI 总线 x1 进行通信。PCI 桥 1 获得 PCI 总线 x0 的使用权后，将转发这个存储器写总线事务到 PCI 总线 x0，之后 HOST 主桥 x 将接收这个存储器写总线事务，并最终将数据写入主存储器。

由以上过程可以发现，Posted 数据请求在通过 PCI 总线之后，将逐级释放总线资源，因此 PCI 总线的利用率较高。而使用 Non-Posted 方式进行数据传送的处理过程与此不同，Non-Posted 数据请求在通过 PCI 总线时，并不会及时释放总线资源，从而在某种程度上影响 PCI 总线的使用效率和传送带宽。

1.3.3 HOST 处理器访问 PCI 设备

HOST 处理器对 PCI 设备的数据访问主要包含两方面内容，一方面是处理器向 PCI 设备发起存储器和 I/O 读写请求；另一方面是处理器对 PCI 设备进行配置读写。

在 PCI 设备的配置空间中，共有 6 个 BAR 寄存器。每一个 BAR 寄存器都与 PCI 设备使用的一组 PCI 总线地址空间对应，BAR 寄存器记录这组地址空间的基址。本书将与 BAR 寄存器对应的 PCI 总线地址空间称为 BAR 空间，在 BAR 空间中可以存放 I/O 地址空间，也可以存放存储器地址空间。

PCI 设备可以根据需要，有选择地使用这些 BAR 空间。值得注意的是，在 BAR 寄存器中存放的是 PCI 设备使用的“PCI 总线域”的物理地址，而不是“存储器域”的物理地址，有关 BAR 寄存器的详细介绍见第 2.3.2 节。

HOST 处理器访问 PCI 设备 I/O 地址空间的过程，与访问存储器地址空间略有不同。有些处理器，如 x86 处理器，具有独立的 I/O 地址空间。x86 处理器可以将 PCI 设备使用的 I/O 地址映射到存储器域的 I/O 地址空间中，之后处理器可以使用 IN、OUT 等指令对存储器域的 I/O 地址进行访问，然后通过 HOST 主桥将存储器域的 I/O 地址转换为 PCI 总线域的 I/O 地址，最后使用 PCI 总线的 I/O 总线事务对 PCI 设备的 I/O 地址进行读写访问。在 x86 处理器中，存储器域的 I/O 地址与 PCI 总线域的 I/O 地址相同。

对于有些没有独立 I/O 地址空间的处理器，如 PowerPC 处理器，需要在 HOST 主桥初始化时，将 PCI 设备使用的 I/O 地址空间映射为处理器的存储器地址空间。PowerPC 处理器对这段“存储器域”的存储器空间进行读写访问时，HOST 主桥将存储器域的这段存储器地址转换为 PCI 总线域的 I/O 地址，然后通过 PCI 总线的 I/O 总线事务对 PCI 设备的 I/O 地址进行读写操作。

在 PCI 总线中，存储器读写事务与 I/O 读写事务的实现较为类似。首先 HOST 处理器在初始化时，需要将 PCI 设备使用的 BAR 空间映射到“存储器域”的存储器地址空间。之后处理器通过存储器读写指令访问“存储器域”的存储器地址空间，HOST 主桥将“存储器域”的读写请求翻译为 PCI 总线的存储器读写总线事务之后，再发送给目标设备。

值得注意的是存储器域和 PCI 总线域的概念。PCI 设备能够直接使用的地址是 PCI 总线域的地址，在 PCI 总线事务中出现的地址也是 PCI 总线域的地址；而处理器能够直接使用的地址是存储器域的地址。理解存储器域与 PCI 总线域的区别对于理解 PCI 总线至关重要，在第 2.1 节将专门讨论这两个概念。

以上对 PCI 总线的存储器与 I/O 总线事务的介绍并没有考虑 PCI 桥的存在，如果将 PCI 桥考虑进来，情况将略微复杂。下面将以图 1-1 为例说明处理器如何通过 HOST 主桥和 PCI 桥 1 对 PCI 设备 11 进行存储器读写操作。当处理器对 PCI 设备 11 进行存储器写操作时，这些数据需要通过 HOST 主桥 x 和 PCI 桥 x1，最终到达 PCI 设备 11，其访问步骤如下。值得注意的是，以下步骤忽略 PCI 总线的仲裁过程。

(1) 首先处理器将要传递的数据放入通用寄存器中，之后向 PCI 设备 11 映射到的存储器域的地址进行写操作。值得注意的是，处理器并不能直接访问 PCI 设备 11 的 PCI 总线地址空间，因为这些地址空间是属于 PCI 总线域的，处理器所能直接访问的空间是存储器域的地址空间。处理器必须通过 HOST 主桥将存储器域的数据访问转换为 PCI 总线事务才能对

PCI 总线地址空间进行访问。

(2) HOST 主桥 x 接收来自处理器的存储器写请求，之后处理器结束当前存储器写操作，释放系统总线。HOST 主桥 x 将存储器域的存储器地址转换为 PCI 总线域的 PCI 总线地址。并向 PCI 总线 x0 发起 PCI 写请求总线事务。值得注意的是，虽然在许多处理器系统中，存储器地址和 PCI 总线地址完全相等，但其含义并不相同。

(3) PCI 总线 x0 上的 PCI 设备 01、PCI 设备 02 和 PCI 桥 1 将同时监听这个 PCI 写总线事务。最后 PCI 桥 x1 接收这个写总线事务，并结束来自 PCI 总线 x0 的 PCI 总线事务。之后 PCI 桥 x1 向 PCI 总线 x1 发起新的 PCI 总线写总线事务。

(4) PCI 总线 x1 上的 PCI 设备 11 和 PCI 设备 12 同时监听这个 PCI 写总线事务。最后 PCI 设备 11 通过地址译码方式接收这个写总线事务，并结束来自 PCI 总线 x1 上的 PCI 总线事务。

由以上过程可以发现，由于存储器写总线事务使用 Posted 传送方式，因此数据通过 PCI 桥后都将结束上一级总线的 PCI 总线事务，从而上一级 PCI 总线可以被其他 PCI 设备使用。如果使用 Non-Posted 传送方式，直到数据发送到 PCI 设备 11 之后，PCI 总线 x1 和 x0 才能依次释放，从而在某种程度上将造成 PCI 总线的拥塞。

处理器对 PCI 设备 11 进行 I/O 读写操作和存储器读操作时只能采用 Non-Posted 方式进行，与 Posted 方式相比，使用 Non-Posted 方式，当数据到达目标设备后，目标设备需要向主设备发出“回应^①”，当主设备收到这个“回应”后才能结束整个总线事务。本节不再讲述处理器如何对 PCI 设备进行 I/O 写操作，请读者思考这个过程。

处理器对 PCI 设备 11 进行存储器读时，这个读请求需要首先通过 HOST 主桥 x 和 PCI 桥 x1 到达 PCI 设备，之后 PCI 设备将读取的数据再次通过 PCI 桥 x1 和 HOST 主桥 x 传递给 HOST 处理器，其步骤如下所示。我们首先假设 PCI 总线没有使用 Delayed 传送方式处理 Non-Posted 总线事务，而是使用纯粹的 Non-Posted 方式。

(1) 首先处理器准备接收数据使用的通用寄存器，之后向 PCI 设备 11 映射到的存储器域的地址进行读操作。

(2) HOST 主桥 x 接收来自处理器的存储器读请求。HOST 主桥 x 进行存储器地址到 PCI 总线地址的转换，之后向 PCI 总线 x0 发起存储器读总线事务。

(3) PCI 总线 x0 上的 PCI 设备 01、PCI 设备 02 和 PCI 桥 x1 将监听这个存储器读请求，之后 PCI 桥 1 接收这个存储器读请求。然后 PCI 桥 x1 向 PCI 总线 x1 发起新的 PCI 总线读请求。

(4) PCI 总线 x1 上的 PCI 设备 11 和 PCI 设备 12 监听这个 PCI 读请求总线事务。最后 PCI 设备 11 接收这个存储器读请求总线事务，并将这个读请求总线事务转换为存储器读完成总线事务之后，将数据传送到 PCI 桥 x1，并结束来自 PCI 总线 x1 上的 PCI 总线事务。

(5) PCI 桥 x1 将接收到的数据通过 PCI 总线 x0，继续上传到 HOST 主桥 x，并结束 PCI 总线 x0 上的 PCI 总线事务。

① 如果是存储器、I/O 读或者配置读总线事务，这个回应包含数据；如果是 I/O 写或者配置写，这个回应不包含数据。

(6) HOST 主桥 x 将数据传递给处理器，最终结束处理器的存储器读操作。

显然这种方式与 Posted 传送方式相比，PCI 总线的利用率较低。因为只要 HOST 处理器没有收到来自目标设备的“回应”，那么 HOST 处理器到目标设备的传送路径上使用的所有 PCI 总线都将被阻塞。因而 PCI 总线 x0 和 x1 并没有被充分利用。

由以上例子，可以发现只有“读完成”依次通过 PCI 总线 x1 和 x0 之后，存储器读总线事务才不继续占用 PCI 总线 x1 和 x0 的资源，显然这种数据传送方式并不合理。因此 PCI 总线使用 Delayed 传送方式解决这个总线拥塞问题，有关 Delayed 传送方式的实现机制见第 1.3.5 节。

1.3.4 PCI 设备读写主存储器

PCI 设备与存储器直接进行数据交换的过程也被称为 DMA。与其他总线的 DMA 过程类似，PCI 设备进行 DMA 操作时，需要获得数据传送的目的地址和传送大小。支持 DMA 传递的 PCI 设备可以在其 BAR 空间中设置两个寄存器，分别保存这个目标地址和传送大小。这两个寄存器也是 PCI 设备 DMA 控制器的组成部件。

值得注意的是，PCI 设备进行 DMA 操作时，使用的目的地址是 PCI 总线域的物理地址，而不是存储器域的物理地址，因为 PCI 设备并不能识别存储器域的物理地址，而仅能识别 PCI 总线域的物理地址。

HOST 主桥负责完成 PCI 总线地址到存储器域地址的转换。HOST 主桥需要进行合理设置，将存储器的地址空间映射到 PCI 总线之后，PCI 设备才能对这段存储器空间进行 DMA 操作。PCI 设备不能直接访问没有经过主桥映射的存储器空间。

许多处理器允许 PCI 设备访问所有存储器域地址空间，但是有些处理器可以设置 PCI 设备所能访问的存储器域地址空间，从而对存储器域地址空间进行保护。例如 PowerPC 处理器的 HOST 主桥可以使用 Inbound 寄存器组，设置 PCI 设备访问的存储器地址范围和属性，只有在 Inbound 寄存器组映射的存储器空间才能被 PCI 设备访问，在第 2.2 节将详细介绍 PowerPC 处理器的这组寄存器。

综上所述，在一个处理器系统中，并不是所有存储器空间都可以被 PCI 设备访问，只有在 PCI 总线域中有映像的存储器空间才能被 PCI 设备访问。经过 HOST 主桥映射的存储器，具有两个“地址”，一个是在存储器域的地址，一个是在 PCI 总线域的 PCI 总线地址。当处理器访问这段存储器空间时，使用存储器地址；而 PCI 设备访问这段内存时，使用 PCI 总线地址。在多数处理器系统中，存储器地址与 PCI 总线地址相同，但是系统程序员需要正确理解这两个地址的区别。

下面以 PCI 设备 11 向主存储器写数据为例，说明 PCI 设备如何进行 DMA 写操作。

(1) 首先 PCI 设备 11 将存储器写请求发向 PCI 总线 x1，注意这个写请求使用的地址是 PCI 总线域的地址。

(2) PCI 总线 x1 上的所有设备监听这个请求，因为 PCI 设备 11 是向处理器的存储器写数据，所以 PCI 总线 x1 上的 PCI Agent 设备都不会接收这个数据请求。

(3) PCI 桥 x1 发现当前总线事务使用的 PCI 总线地址不是其下游设备使用的 PCI 总线地址，则接收这个数据请求，有关 PCI 桥的 Secondary 总线接收数据的过程见第 3.2.1 节。此时 PCI 桥 x1 将结束来自 PCI 设备 11 的 Posted 存储器写请求，并将这个数据请求推到上游

PCI 总线上，即 PCI 总线 x0 上。

(4) PCI 总线 x0 上的所有 PCI 设备包括 HOST 主桥将监听这个请求。PCI 总线 x0 上的 PCI Agent 设备也不会接收这个数据请求，此时这个数据请求将由 HOST 主桥 x 接收，并结束 PCI 桥 x1 的 Posted 存储器写请求。

(5) HOST 主桥 x 发现这个数据请求发向存储器，则将来自 PCI 总线 x0 的 PCI 总线地址转换为存储器地址，之后通过存储器控制器将数据写入存储器，完成 PCI 设备的 DMA 写操作。

PCI 设备进行 DMA 读过程与 DMA 写过程较为类似。不过 PCI 总线的存储器读总线事务只能使用 Non-Posted 总线事务，其过程如下。

(1) 首先 PCI 设备 11 将存储器读请求发向 PCI 总线 x1。

(2) PCI 总线 x1 上的所有设备监听这个请求，因为 PCI 设备 11 是从存储器中读取数据，所以 PCI 总线 x1 上的设备，如 PCI 设备 12，不会接收这个数据请求。PCI 桥 x1 发现下游 PCI 总线没有设备接收这个数据请求，则接收这个数据请求，并将这个数据请求推到上游 PCI 总线上，即 PCI 总线 x0 上。

(3) PCI 总线 x0 上的设备将监听这个请求。PCI 总线 x0 上的设备也不会接收这个数据请求，最后这个数据请求将由 HOST 主桥 x 接收。

(4) HOST 主桥 x 发现这个数据请求是发向主存储器的，则将来自 PCI 总线 x0 的 PCI 总线地址转换为存储器地址，之后通过存储器控制器将数据读出，并转发到 HOST 主桥 x。

(5) HOST 主桥 x 将数据经由 PCI 桥 x1 传递到 PCI 设备 11，PCI 设备 11 接收到这个数据后结束 DMA 读。

以上过程仅是 PCI 设备向存储器读写数据的一个简单流程。如果考虑处理器中的 Cache，这些存储器读写过程较为复杂。

PCI 总线还允许 PCI 设备之间进行数据传递，PCI 设备间的数据交换较为简单。在实际应用中，PCI 设备间的数据交换并不常见。下面以图 1-1 为例，简要介绍 PCI 设备 11 将数据写入 PCI 设备 01 的过程；请读者自行考虑 PCI 设备 11 从 PCI 设备 01 读取数据的过程。

(1) 首先 PCI 设备 11 将 PCI 写总线事务发向 PCI 总线 x1 上。PCI 桥 x1 和 PCI 设备 12 同时监听这个写总线事务。

(2) PCI 桥 x1 将接收这个 PCI 写请求总线事务，并将这个 PCI 写总线事务上推到 PCI 总线 x0。

(3) PCI 总线 x0 上的所有设备将监听这个 PCI 写总线事务，最后由 PCI 设备 01 接收这个数据请求，并完成 PCI 写事务。

1.3.5 Delayed 传送方式

如上所述，当处理器使用 Non-Posted 总线周期对 PCI 设备进行读操作，或者 PCI 设备使用 Non-Posted 总线事务对存储器进行读操作时，如果数据没有到达目的地，那么在这个读操作路径上的所有 PCI 总线都不能被释放，这将严重影响 PCI 总线的使用效率。

为此 PCI 桥需要对 Non-Posted 总线事务进行优化处理，并使用 Delayed 总线事务处理这些 Non-Posted 总线事务，PCI 总线规定只有 Non-Posted 总线事务可以使用 Delayed 总线事务。PCI 总线的 Delay 总线事务由 Delay 读写请求和 Delay 读写完成总线事务组成，当 Delay 读写

请求到达目的地后，将被转换为 Delay 读写完成总线事务。基于 Delay 总线请求的数据交换如图 1-4 所示。

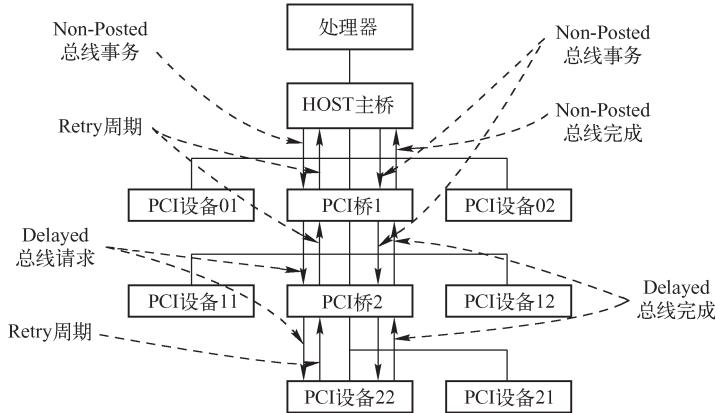


图 1-4 基于 Delayed 总线请求的数据交换

假设处理器通过存储器读、I/O 读写或者配置读写访问 PCI 设备 22 时，首先经过 HOST 主桥进行存储器域与 PCI 总线域的地址转换，并由 HOST 主桥发起 PCI 总线事务，然后通过 PCI 桥 1、2，最终到达 PCI 设备 22。其详细步骤如下。

- (1) HOST 主桥完成存储器域到 PCI 总线域的转换，然后启动 PCI 读总线事务。
- (2) PCI 桥 1 接收这个读总线事务，并首先使用 Retry 周期，使 HOST 主桥择时重新发起相同的总线周期。此时 PCI 桥 1 的上游 PCI 总线将被释放。值得注意的是 PCI 桥并不会每一次都使用 Retry 周期，使上游设备择时进行重试操作。在 PCI 总线中，有一个“16 Clock”原则，即 FRAME#信号有效后，必须在 16 个时钟周期内置为无效，如果 PCI 桥发现来自上游设备的读总线事务不能在 16 个时钟周期内结束时，则使用 Retry 周期终止该总线事务。
- (3) PCI 桥 1 使用 Delayed 总线请求继续访问 PCI 设备 22。
- (4) PCI 桥 2 接收这个总线请求，并将这个 Delayed 总线请求继续传递。此时 PCI 桥 2 也将首先使用 Retry 周期，使 PCI 桥 1 择时重新发起相同的总线周期。此时 PCI 桥 2 的上游 PCI 总线被释放。
- (5) 这个数据请求最终到达 PCI 设备 22，如果 PCI 设备 22 没有将数据准备好时，也可以使用 Retry 周期，使 PCI 桥 2 择时重新发起相同的总线周期；如果数据已经准备好，PCI 设备 22 将接收这个数据请求，并将这个 Delayed 总线请求转换为 Delayed 总线完成事务。如果 Delayed 总线请求是读请求，则 Delayed 总线完成事务中含有数据，否则只有完成信息，而不包含数据。
- (6) Delayed 总线完成事务将“数据或者完成信息”传递给 PCI 桥 2，当 PCI 桥 1 重新发出 Non-Posted 总线请求时，PCI 桥 2 将这个“数据或者完成信息”传递给 PCI 桥 1。
- (7) HOST 主桥重新发出存储器读总线事务时，PCI 桥 1 将“数据或者完成信息”传递给 HOST 主桥，最终完成整个 PCI 总线事务。

由以上分析可知，Delayed 总线周期由 Delayed 总线请求和 Delayed 总线完成两部分组成。下面将 Delayed 读请求总线事务简称为 DRR (Delayed Read Request)，Delayed 读完成总线事务简称为 DRC (Delayed Read Completion)；而将 Delayed 写请求总线事务简称为 DWR

(Delayed Write Request), Delayed 写完成总线事务简称为 DWC (Delayed Write Completion)。

PCI 总线使用 Delayed 总线事务，在一定程度上可以提高 PCI 总线的利用率。因为在进行 Non-Posted 总线事务时，Non-Posted 请求在通过 PCI 桥之后，可以暂时释放 PCI 总线，但是采用这种方式，HOST/PCI 桥将会择时进行重试操作。在许多情况下，使用 Delayed 总线事务，并不能取得理想的效果，因为过多的重试周期也将大量消耗 PCI 总线的带宽。

为了进一步提高 Non-Posted 总线事务的执行效率，PCI-X 总线将 PCI 总线使用的 Delayed 总线事务，升级为 Split 总线事务。采用 Split 总线事务可以有效解决 HOST/PCI 桥的这些重试操作。Split 总线事务的基本思想是发送端首先将 Non-Posted 总线请求发送给接收端，然后再由接收端主动地将数据传递给发送端。

除了 PCI-X 总线可以使用 Split 总线事务进行数据传送之外，有些处理器，如 x86 和 PowerPC 处理器的 FSB (Front Side Bus) 总线也支持这种 Split 总线事务，因此这些 HOST 主桥也可以发起这种 Split 总线事务。在 PCIe 总线中，Non-Posted 数据传送都使用 Split 总线事务完成，而不再使用 Delayed 总线事务。在第 1.5.1 节将简要介绍 Split 总线事务和 PCI-X 总线对 PCI 总线的一些功能上的增强。

1.4 PCI 总线的中断机制

PCI 总线使用 INTA#、INTB#、INTC# 和 INTD# 信号向处理器发出中断请求。这些中断请求信号为低电平有效，并与处理器的中断控制器连接。在 PCI 体系结构中，这些中断信号属于边带信号 (Sideband Signals)，PCI 总线规范并没有明确规定在一个处理器系统中如何使用这些信号，因为这些信号对于 PCI 总线是可选信号。PCI 设备还可以使用 MSI 机制向处理器提交中断请求，而不使用这组中断信号。有关 MSI 机制的详细说明见第 10 章。

1.4.1 中断信号与中断控制器的连接关系

不同的处理器使用的中断控制器不同，如 x86 处理器使用 APIC (Advanced Programmable Interrupt Controller) 中断控制器，而 PowerPC 处理器使用 MPIC (Multiprocessor Interrupt Controller) 中断控制器。这些中断控制器都提供了一些外部中断请求引脚 IRQ_PINx#。外部设备，包括 PCI 设备可以使用这些引脚向处理器提交中断请求。

但是 PCI 总线规范没有规定 PCI 设备的 INTx 信号如何与中断控制器的 IRQ_PINx# 信号相连，这为系统软件的设计带来了一定的困难，为此系统软件使用中断路由表存放 PCI 设备的 INTx 信号与中断控制器的连接关系。在 x86 处理器系统中，BIOS 可以提供这个中断路由表，而在 PowerPC 处理器中 Firmware 也可以提供这个中断路由表。

在一些简单的嵌入式处理器系统中，Firmware 并没有提供中断路由表，系统软件开发者需要事先了解 PCI 设备的 INTx 信号与中断控制器的连接关系。此时外部设备与中断控制器的连接关系由硬件设计人员指定。

假设在一个处理器系统中，共有 3 个 PCI 插槽（分别为 PCI 插槽 A、B 和 C），这些 PCI 插槽与中断控制器的 IRQ_PINx 引脚（分别为 IRQW#、IRQX#、IRQY# 和 IRQZ#）可以按照图 1-5 所示的拓扑结构进行连接。

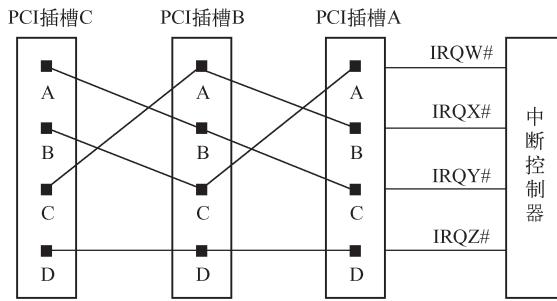


图 1-5 PCI 插槽与中断控制器连接拓扑图

采用图 1-5 所示的拓扑结构时，PCI 插槽 A、B、C 的 INTA#、INTB# 和 INTC# 信号将分散连接到中断控制器的 IRQW#、IRQX# 和 IRQY# 信号，而所有 INTD# 信号将共享一个 IRQZ# 信号。采用这种连接方式时，整个处理器系统使用的中断请求信号，其负载较为均衡。而且这种连接方式保证了每一个插槽的 INTA# 信号都与一根独立的 IRQx# 信号对应，从而提高了 PCI 插槽中断请求的效率。

在一个处理器系统中，多数 PCI 设备仅使用 INTA# 信号，很少使用 INTB# 和 INTC# 信号，而 INTD# 信号更是极少使用。在 PCI 总线中，PCI 设备配置空间的 Interrupt Pin 寄存器记录该设备究竟使用哪个 INTx 信号，该寄存器的详细介绍见第 2.3.2 节。

1.4.2 中断信号与 PCI 总线的连接关系

在 PCI 总线中，INTx 信号属于边带信号。所谓边带信号是指这些信号在 PCI 总线中是可选信号，而且只能在一个处理器系统的内部使用，并不能离开这个处理器环境。PCI 桥也不会处理这些边带信号。这给 PCI 设备将中断请求发向处理器带来了一些困难，特别是给挂接在 PCI 桥之下的 PCI 设备进行中断请求带来了一些麻烦。

在一些嵌入式处理器系统中，这个问题较易解决。因为嵌入式处理器系统很清楚在当前系统中存在多少个 PCI 设备，这些 PCI 设备使用了哪些中断资源。在多数嵌入式处理器系统中，PCI 设备的数量小于中断控制器提供的外部中断请求引脚数，而且在嵌入式系统中，多数 PCI 设备仅使用 INTA# 信号提交中断请求。

在这类处理器系统中，可能并不含有 PCI 桥，因而 PCI 设备的中断请求信号与中断控制器的连接关系较易确定。即便存在 PCI 桥，来自 PCI 桥之下的 PCI 设备的中断请求也较易处理。

在多数情况下，嵌入式处理器系统使用的 PCI 设备仅使用 INTA# 信号进行中断请求，所以只要将这些 INTA# 信号挂接到中断控制器的独立 IRQ_PIN# 引脚上即可。这样每一个 PCI 设备都可以独占一个单独的中断引脚。

而在 x86 处理器系统中，这个问题需要 BIOS 参与来解决。在 x86 处理器系统中，有许多 PCI 插槽，处理器系统并不知道在这些插槽上将要挂接哪些 PCI 设备，也并不知道这些 PCI 设备是否需要使用所有的 INTx# 信号线。因此 x86 处理器系统必须对各种情况进行处理。

x86 处理器系统还经常使用 PCI 桥进行 PCI 总线扩展，扩展出来的 PCI 总线还可能挂接一些 PCI 插槽，这些插槽上的 INTx# 信号仍然需要处理。PCI 桥规范并没有要求桥片传递其下 PCI 设备的中断请求。事实上多数 PCI 桥也没有为下游 PCI 总线提供中断引脚 INTx#，管

理其下游总线的 PCI 设备。但是 PCI 桥规范推荐使用表 1-3 建立下游 PCI 设备的 INTx 信号与上游 PCI 总线 INTx 信号之间的映射关系。

表 1-3 PCI 设备 INTx#信号与 PCI 总线 INTx#信号的映射关系

设备号	PCI 设备的 INTx#信号	PCI 总线的 INTx#信号
0, 4, 8, 12, 16, 20, 24, 28	INTA#	INTA#
	INTB#	INTB#
	INTC#	INTC#
	INTD#	INTD#
1, 5, 9, 13, 17, 21, 25, 29	INTA#	INTB#
	INTB#	INTC#
	INTC#	INTD#
	INTD#	INTA#
2, 6, 10, 14, 18, 22, 26, 30	INTA#	INTC#
	INTB#	INTD#
	INTC#	INTA#
	INTD#	INTB#
3, 7, 11, 15, 19, 23, 27, 31	INTA#	INTD#
	INTB#	INTA#
	INTC#	INTB#
	INTD#	INTC#

下面举例说明该表的含义。在 PCI 桥下游总线上的 PCI 设备，如果其设备号为 0，那么这个设备的 INTA#引脚将和 PCI 总线的 INTA#引脚相连；如果其设备号为 1，其 INTA#引脚将和 PCI 总线的 INTB#引脚相连；如果其设备号为 2，其 INTA#引脚将和 PCI 总线的 INTC#引脚相连；如果其设备号为 3，其 INTA#引脚将和 PCI 总线的 INTD#引脚相连。

在 x86 处理器系统中，由 BIOS 或者 APCI 表记录 PCI 总线的 INTA ~ D#信号与中断控制器之间的映射关系，保存这个映射关系的数据结构也被称为中断路由表。大多数 BIOS 使用表 1-3 中的映射关系，这也是绝大多数 BIOS 支持的方式。如果在一个 x86 处理器系统中，PCI 桥下游总线的 PCI 设备使用的中断映射关系与此不同，那么系统软件程序员需要改动 BIOS 中的中断路由表。

BIOS 初始化代码根据中断路由表中的信息，可以将 PCI 设备使用的中断向量号写入到该 PCI 设备配置空间的 Interrupt Line register 寄存器中，该寄存器将在第 2.3.2 节中介绍。

1.4.3 中断请求的同步

在 PCI 总线中，INTx 信号是一个异步信号。所谓异步是指 INTx 信号的传递并不与 PCI 总线的数据传送同步，即 INTx 信号的传递与 PCI 设备使用的 CLK#信号无关。这个“异步”信号给系统软件的设计带来了一定的麻烦。

系统软件程序员需要注意“异步”这种事件，因为几乎所有“异步”事件都会带来系统的“同步”问题。以图 1-1 为例，当 PCI 设备 11 使用 DMA 写方式，将一组数据写入存

储器，该设备在最后一个数据离开 PCI 设备 11 的发送 FIFO 时，会认为 DMA 写操作已经完成。此时这个设备将通过 INTx 信号，通知处理器 DMA 写操作完成。

此时处理器（驱动程序的中断服务例程）需要注意，因为 INTx 信号是一个异步信号，当处理器收到 INTx 信号时，并不意味着 PCI 设备 11 已经将数据写入存储器中，因为 PCI 设备 11 的数据传递需要通过 PCI 桥 1 和 HOST 主桥，最终才能到达存储器控制器。

而 INTx 信号是“异步”发送给处理器的，PCI 总线并不知道这个“异步”事件何时被处理。很有可能处理器已经接收到 INTx 信号，开始执行中断处理程序时，该 PCI 设备还没有完全将数据写入存储器。

因为“PCI 设备向处理器提交中断请求”与“将数据写入存储器”分别使用了两个不同的路径，处理器系统无法保证哪个信息率先到达。从而在处理器系统中存在“中断同步”的问题，PCI 总线提供了以下两种方法解决这个同步问题。

(1) PCI 设备保证在数据到达目的地之后，再提交中断请求。

显然这种方法不仅加大了硬件的开销，而且也不容易实现。如果 PCI 设备采用 Posted 写总线事务，PCI 设备无法单纯通过硬件逻辑判断数据什么时候写入到存储器。此时为了保证数据到达目的地后，PCI 设备才能提交中断请求，PCI 设备需要使用“读刷新”的方法保证数据可以到达目的地，其方法如下。

PCI 设备在提交中断请求之前，向 DMA 写的数据区域发出一个读请求，这个读请求总线事务将被 PCI 设备转换为读完成总线事务，当 PCI 设备收到这个读完成总线事务后，再向处理器提交中断请求。PCI 总线的“序”机制保证这个存储器读请求，会将 DMA 数据最终写入存储器，有关 PCI 序的详细说明见第 11.3 节。

PCI 总线规范要求 HOST 主桥和 PCI 桥必须保证这种读操作可以刷新写操作。但问题是，没有多少芯片设计者愿意提供这种机制，因为这将极大地增加他们的设计难度。除此之外，使用这种方法也将增加中断请求的延时。

(2) 中断服务例程使用“读刷新”方法。

中断服务例程在使用“PCI 设备写入存储器”的这些数据之前，需要对这个 PCI 设备进行读操作。这个读操作也可以强制将数据最终写入存储器，实际上是将数据写到存储器控制器中。这种方法利用了 PCI 总线的传送序规则，与第 1 种方法基本相同，只是这种方法使用软件方式，而第 1 种方式使用硬件方式。第 11.3 节将详细介绍这个读操作如何将数据刷新到存储器中。

第 2 种方法也是绝大多数处理器系统采用的方法。程序员在编写中断服务例程时，往往都是先读取 PCI 设备的中断状态寄存器，判断中断产生原因之后，才对 PCI 设备写入的数据进行操作。这个读取中断状态寄存器的过程，一方面可以获得设备的中断状态，另一方面可以保证 DMA 写的数据最终到达存储器。如果驱动程序不这样做，就可能产生数据完整性问题。产生这种数据完整性问题的原因是 INTx 这个异步信号。

这里也再次提醒系统程序员注意 PCI 总线的“异步”中断所带来的数据完整性问题。在一个操作系统中，即便中断处理程序没有首先读取 PCI 设备的寄存器，也多半不会出现问题，因为在操作系统中，一个 PCI 设备从提交中断到处理器开始执行设备的中断服务例程，所需要的时间较长，处理器系统基本上可以保证此时数据已经写入存储器。

但是如果系统程序员不这样做，这个驱动程序依然有 Bug，尽管这些 Bug 因为各种机缘

巧合，始终不能够暴露出来，而一旦这些 Bug 被暴露出来将难以定位。为此系统程序员务必重视设计中的每一个实现细节，当然仅凭小心谨慎是远远不够的，因为重视细节的前提是充分理解这些细节。

PCI 总线 V2.2 规范还定义了一种新的中断机制，即 MSI 中断机制。MSI 中断机制采用存储器写总线事务向处理器系统提交中断请求，其实现机制是向 HOST 处理器指定的一个存储器地址写指定的数据。这个存储器地址一般是中断控制器规定的某段存储器地址范围，而且数据也是事先安排好的数据，通常含有中断向量号。

HOST 主桥会将 MSI 这个特殊的存储器写总线事务进一步翻译为中断请求，提交给处理器。目前 PCIe 和 PCI-X 设备必须支持 MSI 中断机制，但是 PCI 设备并不一定都支持 MSI 中断机制。

目前 MSI 中断机制虽然在 PCIe 总线上已经成为主流，但是在 PCI 设备中并不常用。即便是支持 MSI 中断机制的 PCI 设备，在设备驱动程序的实现中也很少使用这种机制。首先 PCI 设备具有 INTx#信号可以传递中断，而且这种中断传送方式在 PCI 总线中根深蒂固。其次 PCI 总线是一个共享总线，传递 MSI 中断需要占用 PCI 总线的带宽，需要进行总线仲裁等一系列过程，远没有使用 INTx#信号线直接。

但是使用 MSI 中断机制可以取消 PCI 总线这个 INTx#边带信号，可以解决使用 INTx 中断机制所带来的数据完整性问题。而更为重要的是，PCI 设备使用 MSI 中断机制，向处理器系统提交中断请求时，还可以通知处理器系统产生该中断的原因，即通过不同中断向量号表示中断请求的来源。当处理器系统执行中断服务例程时，不需要读取 PCI 设备的中断状态寄存器，获得中断请求的来源，从而在一定程度上提高了中断处理的效率。本书将在第 10 章详细介绍 MSI 中断机制。

1.5 PCI-X 总线简介

PCI-X 总线仍采用并行总线技术。PCI-X 总线使用的大多数总线事务基于 PCI 总线，但是在实现细节上略有不同。PCI-X 总线将工作频率提高到 533 MHz，并首先引入了 PME (Power Management Event) 机制。除此之外，PCI-X 总线还提出了许多新的特性。

1.5.1 Split 总线事务

Split 总线事务是 PCI-X 总线的一个重要特性。该总线事务替代了 PCI 总线的 Delayed 数据传送方式，从而提高了 Non-Posted 总线事务的传送效率。下面以存储器读为例，说明 PCI-X 设备如何使用 Split 总线事务。

PCI-X 总线在进行存储器读总线事务时，总线事务的发起方 (Requester) 使用 Split 总线事务与总线事务接收端 (Completer) 进行数据交换，其步骤如下。

- (1) Requester 向 Completer 发起存储器读请求总线事务。
- (2) 这个存储器读请求在到达 Completer 之前，可能会经过多级 PCI-X 桥。这些 PCI-X 桥使用 Split Response 周期结束当前总线事务，释放上游 PCI 总线。之后继续转发这个存储器读请求，直到 Completer 认领这个存储器读请求总线事务。
- (3) Completer 认领存储器读请求总线事务后，会记录 Requester 的 ID 号，并使用 Split

Response 周期结束存储器读请求总线事务。

(4) Completer 准备好数据后，将重新申请总线，并使用存储器读完成总线事务主动地将数据传送给 Requester。在这个完成报文中包含 Requester 的 ID 号，因为完成报文使用 ID 路由而不是地址路由。

(5) 这些完成报文根据 ID 路由方式，最终到达 Requester。Requester 从完成报文中接收数据并完成整个存储器读请求。

与 Delayed 总线事务相比，Requester 获得的数据是 Completer 将数据完全准备好后，由 Completer 主动传递的，而不是通过 Requester 通过多次重试获得的，因此能够提高 PCI-X 总线的使用效率。PCI-X 总线提出的 Split 总线事务被 PCIe 总线继承。

1.5.2 总线传送协议

PCI-X 总线改变了 PCI 总线使用的传送协议。目标设备可以将主设备发送的命令锁存，然后在下一个时钟周期进行译码操作。与 PCI 总线事务相比，PCI-X 总线采用的这种方式，虽然在总线时序中多使用了一个时钟周期，但是可以有效提高 PCI-X 总线的运行频率。

因为主设备通过数据线将命令发送到目标设备需要一定的延时。如果 PCI 总线频率较高，目标设备很难在一个时钟周期内接收完毕总线命令，并同时完成译码工作。而如果目标设备能够将主设备发出的命令先进行锁存，然后在下一个时钟周期进行译码则可以有效解决这个译码时间 Margin 不足的问题，从而提高 PCI-X 总线的频率。PCI-X 1.0 总线可以使用的最高总线频率为 133 MHz，而 PCI-X 2.0 总线可以使用的最高总线频率为 533 MHz，远比 PCI 总线使用的总线频率高。

除了信号传送协议外，PCI-X 总线在进行 DMA 读写时，可以不进行 Cache 共享一致性操作，而 PCI 总线进行 DMA 读写时必须进行 Cache 一致性操作。在某些特殊情况下，DMA 读写时进行 Cache 共享一致性不但不能提高总线传送效率，反而会降低。第 3.3 节将详细讨论与 Cache 一致性相关的 PCI 总线事务。

此外 PCI-X 总线还支持乱序总线事务，即 Relaxed Ordering，该总线事务被 PCIe 总线继承。对于某些应用，PCI-X 设备使用 Relaxed ordering 方式，可以有效地提高数据传送效率。但是支持 Relaxed Ordering 的设备，需要较多的数据缓存和硬件逻辑处理这些乱序，这为 PCI-X 设备的设计带来了不小的困难。

1.5.3 基于数据块的突发传送

在 PCI 总线中，一次突发传送的大小为 2 个以上的双字。一次突发传送所携带的数据越多，突发传送的总线利用率也越高。

而 PCI 总线的突发传送仍然存在缺陷。在 PCI 总线中，数据发送端知道究竟需要发送多少字节的数据，但是接收端并不清楚到底需要接收多少数据。这种不确定性，为接收端的缓冲管理带来了较大的挑战。

为此 PCI-X 总线使用基于数据块的突发传送方式，发送端以 ADB (Allowable Disconnect Boundary) 为单位，将数据发送给接收端，一次突发读写为一个以上的 ADB。采用这种方式，接收端可以事先预知是否有足够的接收缓冲，接收来自发送端的数据，从而可以及时断连当前总线周期，以节约 PCI-X 总线的带宽。在 PCI-X 总线中，ADB 的大小为 128 B。

由于 ADB 的引入，PCI 总线与 Cache 相关的总线事务如 Memory Read Line、Memory Read Multiline 和 Memory Write and Invalidate，都被 PCI-X 总线使用与 ADB 相关的总线事务替代。因为通过 ADB，PCI-X 桥（HOST 主桥）可以准确地预知即将访问的数据在 Cache 中的分布情况。

PCI-X 总线还增加了一些其他特性，如在总线事务中增加传送字节计数，限制等待状态等机制，并增强了奇偶校验的管理方式。但是 PCI-X 总线还没有普及，就被 PCIe 总线替代。因此在 PC 领域和嵌入式领域很少有基于 PCI-X 总线的设备，PCI-X 设备仅在一些高端服务器上出现。因此本节不对 PCI-X 总线做进一步描述。事实上，PCI-X 总线的许多特性都被 PCIe 总线继承。

1.6 小结

本章主要介绍了 PCI 总线的基本组成部件，PCI 设备如何提交中断请求，以及 PCI-X 总线对 PCI 总线的功能增强。本章的重点在于 PCI 总线的 Posted 和 Non-Posted 总线事务，以及 PCI 总线如何使用 Delayed 传送方式处理 Non-Posted 总线事务，请读者务必深入理解这两种总线事务的不同。

第2章 PCI总线的桥与配置

在PCI体系结构中，含有两类桥，一类是HOST主桥，另一类是PCI桥。在每一个PCI设备中（包括PCI桥）都含有一个配置空间。这个配置空间由HOST主桥管理，而PCI桥可以转发来自HOST主桥的配置访问。在PCI总线中，PCI Agent设备使用的配置空间与PCI桥使用的配置空间有些差别，但这些配置空间都是由处理器通过HOST主桥管理的。

2.1 存储器域与PCI总线域

HOST主桥的实现因处理器系统而异。PowerPC处理器和x86处理器的HOST主桥除了集成方式不同之外，其实现机制也有较大差异。但是这些HOST主桥所完成的最基本功能依然是分离存储器域与PCI总线域，完成PCI总线域到存储器域，存储器域到PCI总线域之间的数据传递，并管理PCI设备的配置空间。

之前曾经多次提到，在一个处理器系统中，存在PCI总线域与存储器域，深入理解这两个域的区别是理解HOST主桥的关键所在。在一个处理器系统中，存储器域、PCI总线域与HOST主桥的关系如图2-1所示。

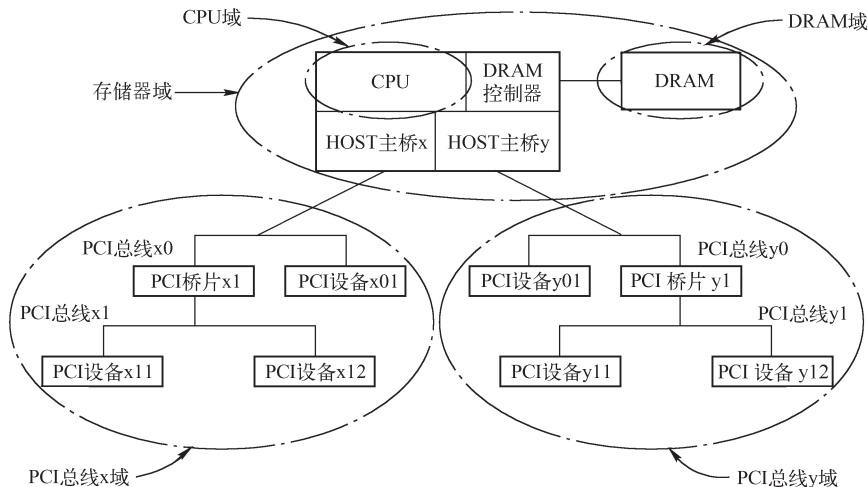


图2-1 存储器域与PCI总线域的划分

图中的处理器系统由一个CPU、一个DRAM控制器和两个HOST主桥组成。在这个处理器系统中，包含CPU域、DRAM域、存储器域和PCI总线域地址空间。其中HOST主桥x和HOST主桥y分别管理PCI总线x域与PCI总线y域。PCI设备访问存储器域时，也需要通过HOST主桥，并由HOST主桥进行PCI总线域到存储器域的地址转换；CPU访问PCI设备时，同样需要通过HOST主桥进行存储器域到PCI总线域的地址转换。

如果 HOST 主桥支持 Peer-to-Peer 传送机制，PCI 总线 x 域上的设备可以与 PCI 总线 y 域上的设备直接通信，如 PCI 设备 x11 可以直接与 PCI 设备 y11 通信。为简化模型，在本书中，PCI 总线仅使用 32 位地址空间。

2.1.1 CPU 域、DRAM 域与存储器域

CPU 域地址空间指 CPU 所能直接访问的地址空间集合。在本书中，CPU、处理器与处理器系统的概念不同。如 MPC8548 处理器的内核是 E500 V2[⊖]，本书将这个处理器内核称为 CPU；处理器由一个或者多个 CPU、外部 Cache、中断控制器和 DRAM 控制器组成；而处理器系统由一个或者多个处理器和外部设备组成。

在 CPU 域中有一个重要概念，即 CPU 域边界。所谓 CPU 域边界，即 CPU 所能控制的数据完整性边界。CPU 域的边界由 Memory Fence 指令[⊖]的作用范围确定，CPU 域边界的划分对数据完整性（Data Consistency）非常重要。与 CPU 域相关的数据完整性知识较为复杂，可以独立成书，因此本书对数据完整性不做进一步介绍。

严格地讲，CPU 域仅在 CPU 内核中有效。CPU 访问主存储器时，首先将读写命令放入读写指令缓冲中，然后将这个命令发送到 DRAM 控制器或者 HOST 主桥。DRAM 控制器或者 HOST 主桥将 CPU 地址转换为 DRAM 或者 PCI 总线地址，分别进入 DRAM 域或者 PCI 总线域后，再访问相应的地址空间。

DRAM 域地址空间指 DRAM 控制器所能访问的地址空间集合。目前处理器系统的 DRAM 一般由 DDR – SDRAM 组成，有的书籍也将这部分内存称为主存储器。在有些处理器系统中，DRAM 控制器能够访问的地址空间，并不能被处理器访问，因此在这类处理器系统中，CPU 域与 DRAM 域地址空间并不等同。

比如有些 CPU 可以支持 36 位的物理地址，而有些 DRAM 控制器仅支持 32 位的物理地址，此时 CPU 域包含的地址空间大于 DRAM 域地址空间。但是这并不意味着 DRAM 域一定包含在 CPU 域中，在某些处理器系统中，CPU 并不能访问在 DRAM 域中的某些数据区域。而 CPU 域中除了包含 DRAM 域外，还包含外部设备空间。

在多数处理器系统中，DRAM 域空间是 CPU 域空间的一部分，但是也有例外。比如显卡控制器可能会借用一部分主存储器空间，这些被借用的空间不能被 CPU 访问，而只能被 DRAM 控制器，更为准确地说是显卡通过 DRAM 控制器访问，因此这段空间不属于 CPU 域，严格地讲，这段空间属于外部设备域。

本书使用存储器域统称 CPU 域与 DRAM 域。存储器域包括 CPU 内部的通用寄存器、存储器映像寻址的寄存器、主存储器空间和外部设备空间。在 Intel 的 x86 处理器系统中，外部设备空间与 PCI 总线域地址空间等效，因为在 x86 处理器系统中，使用 PCI 总线统一管理全部外部设备。为简化起见，本书使用 PCI 总线域替代外部设备域。

值得注意的是，存储器域的外部设备空间，在 PCI 总线域中还有一个地址映射。当处理器访问 PCI 设备时，首先访问的是这个设备在存储器域上的 PCI 设备空间，之后 HOST 主桥

[⊖] MPC8548 处理器基于 E500 V2 内核。目前 E500 内核包括 V1、V2 和 mc（MultiCore）三个版本。

[⊖] x86 处理器的 Memory Fence 指令为 MFENCE、LFENCE 和 SFENCE，而 PowerPC 处理器的 Memory Fence 指令为 msync 和 mbar。

将这个存储器域的 PCI 总线地址转换为 PCI 总线域的物理地址[⊖]，然后通过 PCI 总线事务访问 PCI 总线域的地址空间。

2.1.2 PCI 总线域

在 x86 处理器系统中，PCI 总线域是外部设备域的重要组成部分。实际上在 Intel 的 x86 处理器系统中，所有的外部设备都使用 PCI 总线管理。而 AMD 的 x86 处理器系统中还存在一条 HT (HyperTransport) 总线，HT 总线的主要目的是替代 FSB 总线，但是也可以作为局部总线连接一些高速设备即 HT 设备，因此在 AMD 的 x86 处理器系统中还存在 HT 总线域。本书对 HT 总线不做进一步介绍。

PCI 总线域 (PCI Segment) 由 PCI 设备所能直接访问的地址空间组成。在一个处理器系统中，可能存在多个 HOST 主桥，因此也存在多个 PCI 总线域。如在图 2-1 所示的处理器系统中，具有两个 HOST 主桥，因而在一个处理器系统中存在 PCI 总线 x 和 y 域。

在多数处理器系统中，分属于两个 PCI 总线域的 PCI 设备并不能直接进行数据交换，而需要通过 FSB 进行数据交换。值得注意的是，如果某些处理器的 HOST 主桥支持 Peer-to-Peer 数据传送，那么这个 HOST 主桥可以支持不同 PCI 总线域间的数据传送。

PowerPC 处理器使用了 OCeaN 技术连接两个 HOST 主桥，OCeaN 可以将属于 x 域的 PCI 数据请求转发到 y 域，OCeaN 支持 PCI 总线的 Peer-to-Peer 数据传送。有关 OCeaN 技术的详细说明见第 2.2 节。

2.1.3 处理器域

处理器域是指一个处理器系统能够访问的地址空间集合。处理器系统能够访问的地址空间由存储器域和外部设备域组成。其中存储器域地址空间较为简单，而在不同的处理器系统中，外部设备域的组成结构并不相同。如在 x86 处理器系统中，外部设备域主要由 PCI 总线域组成，因为大多数外部设备都是挂接在 PCI 总线[⊖]上的，而在 PowerPC 处理器和其他处理器系统中，有相当多的设备与 FSB 直接相连，而不与 PCI 总线相连。

本书仅介绍 PCI 总线域而不对其他外部设备域进行说明。其中存储器域与 PCI 总线域之间由 HOST 主桥联系在一起。深入理解这些域的关系是深入理解 PCI 体系结构的关键所在，实际上这也是理解处理器体系结构的基础。

通过 HOST 主桥，处理器系统可以将处理器域划分为存储器域与 PCI 总线域。其中存储器域与 PCI 总线域彼此独立，并通过 HOST 主桥进行数据交换。HOST 主桥是联系存储器域与 PCI 总线域的桥梁，是 PCI 总线域实际的管理者。

有些书籍认为 HOST 处理器是 PCI 总线域的管理者，这种说法并不准确。假设在一个 SMP 处理器系统中，存在 4 个 CPU 而只有一个 HOST 主桥，这 4 个 CPU 将无法判断究竟谁是 HOST 处理器。不过究竟是哪个处理器作为 HOST 处理器并不重要，因为在每一个处理器系统中，是 HOST 主桥管理 PCI 总线域，而不是 HOST 处理器。当一个处理器系统中含有多个 CPU 时，如果每个 CPU 都可以访问 HOST 主桥，那么这些 CPU 都可以作为这个 HOST 主桥

[⊖] PCI 总线域只含有物理地址，因此下文将直接使用 PCI 总线地址，而不使用 PCI 总线物理地址。

[⊖] AMD 的 x86 处理器中的某些外部设备，可能是基于 HT 总线，而不使用 PCI 总线。

所管理 PCI 总线树的 HOST 处理器。

在一个处理器系统中，CPU 所能访问的 PCI 总线地址一定在存储器域中具有地址映射；而 PCI 设备能访问的存储器域的地址也一定在 PCI 总线域中具有地址映射。当 CPU 访问 PCI 域地址空间时，首先访问存储器域的地址空间，然后经过 HOST 主桥转换为 PCI 总线域的地址，再通过 PCI 总线事务进行数据访问。而当 PCI 设备访问主存储器时，首先通过 PCI 总线事务访问 PCI 总线域的地址空间，然后经过 HOST 主桥转换为存储器域的地址后，再对这些空间进行数据访问。

由此可见，存储器域与 PCI 总线域的转换关系由 HOST 主桥统一进行管理。有些处理器提供了一些寄存器进行这种地址映射，如 PowerPC 处理器使用 Inbound 和 Outbound 寄存器组保存存储器域与 PCI 总线域的地址映射关系；而有些处理器并没有提供这些寄存器，但是存储器域到 PCI 总线域的转换关系依然存在。

HOST 主桥进行不同地址域间的数据交换时，需要遵循以下规则。为区别存储器域到 PCI 总线域的地址映射，下面将 PCI 总线域到存储器域的地址映射称为反向映射。

(1) 处理器访问 PCI 总线域地址空间时，首先需要访问存储器域的地址空间，再通过 HOST 主桥将存储器地址转换为 PCI 总线地址，之后才能进入 PCI 总线域进行数据交换。PCI 设备使用的地址空间保存在各自的 PCI 配置寄存器中，即 BAR 寄存器中。这些 PCI 总线地址空间需要在初始化时映射成为存储器域的存储器地址空间，之后处理器才能访问这些地址空间。在有些处理器的 HOST 主桥中，具有独立的寄存器保存这个地址映射规则，如 PowerPC 处理器的 Outbound 寄存器组；而有些处理器，如在 x86 处理器中，虽然没有这样的寄存器组，但是在 HOST 主桥的硬件逻辑中仍然存在这个地址转换的概念。

(2) PCI 设备访问存储器域时，首先需要访问 PCI 总线域的地址空间，再通过 HOST 主桥将 PCI 总线地址转换为存储器地址，才能穿越 HOST 主桥进行数据交换。为此处理器需要通过 HOST 主桥将这个 PCI 总线地址反向映射为存储器地址。PCI 设备不能访问在 PCI 总线域中没有进行这种反向映射的存储器域地址空间。PowerPC 处理器使用 Inbound 寄存器组存放 PCI 设备所能访问的存储器空间，而在 x86 处理器中并没有这样的寄存器组，但是依然存在这个地址转换的概念。

(3) 如果 HOST 主桥不支持 Peer-to-Peer 传送方式，那么分属不同 PCI 总线域的 PCI 设备间不能直接进行数据交换。在 32 位的 PCI 总线中，每一个 PCI 总线域的地址范围都是 0x0000-0000 ~ 0xFFFF-FFFF，但是这些地址没有直接联系。PCI 总线 x 域上的 PCI 总线地址 0x0000-0000 与 PCI 总线 y 域上的 PCI 总线地址 0x0000-0000 并不相同，而且这两个 PCI 总线地址经过 HOST 主桥反向映射后，得到的存储器地址也不相同。

在第 2.2 节中，将主要以 PowerPC 处理器为例说明 HOST 主桥的实现机制，并在第 2.2.4 节简要说明 x86 处理器中的南北桥构架。尽管部分读者对 PowerPC 处理器并不感兴趣，笔者仍然强烈建议读者仔细阅读第 2.2 节的全部内容。

在 PowerPC 处理器中，HOST 主桥的实现比较完整，尤其是 PCI 总线域与存储器域的映射关系比较明晰，便于读者准确掌握这个重要的概念。而 x86 处理器由于考虑向前兼容 (backward compatibility)，设计中包含了太多的不得已。x86 处理器有时不得不保留原设计中的不完美，向前兼容是 Intel 的重要成就，也是一个沉重的十字架。

2.2 HOST 主桥

本节以 MPC8548 处理器为例说明 HOST 主桥在 PowerPC 处理器中的实现机制，并简要介绍 x86 处理器系统使用的 HOST 主桥。

MPC8548 处理器是 Freescale 基于 E500 V2 内核的一个 PowerPC 处理器，该处理器中集成了 DDR 控制器、多个 eTSEC（Enhanced Three-Speed Ethernet Controller）、PCI/PCI-X 和 PCIe 总线控制器等一系列接口。MPC8548 处理器的拓扑结构如图 2-2 所示。

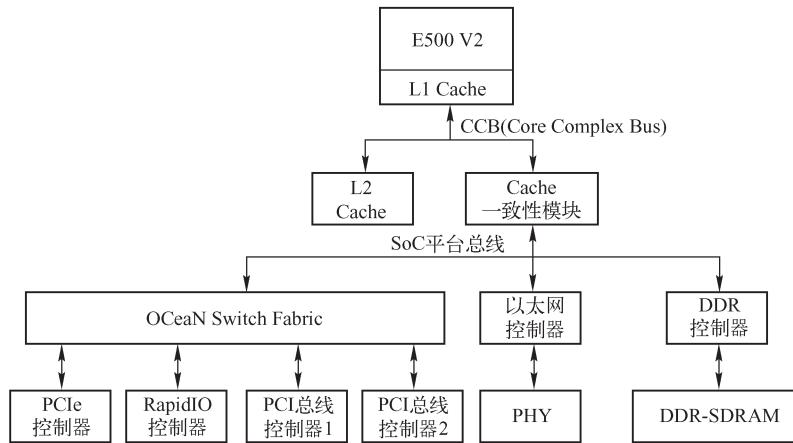


图 2-2 MPC8548 处理器的拓扑结构

图中，MPC8548 处理器的 L1 Cache 在 E500 V2 内核中，而 L2 Cache 与 FSB^①直接相连，不属于 E500 内核。值得注意的是有些高端 PowerPC 处理器的 L2 Cache 也在 CPU 中，而 L3 Cache 与 CCB 总线直接相连。

在 MPC8548 处理器中，所有外部设备，如以太网控制器、DDR 控制器和 OCeaN 连接的总线控制器都与 SoC 平台总线^②直接连接。而 SoC 平台总线通过 Cache 共享一致性模块与 FSB 连接。

在 MPC8548 处理器中，具有一个 32 位的 PCI 总线控制器、一个 64 位的 PCI/PCI-X 总线控制器，还有多个 PCIe 总线控制器。MPC8548 处理器使用 OCeaN 连接这些 PCI、PCI-X 和 PCIe 总线控制器。在 MPC8548 处理器系统中，PCI 设备进行 DMA 操作时，首先通过 OCeaN，之后经过 SoC 平台总线到达 DDR 控制器。

OCeaN 是 MPC8548 处理器中连接快速外设使用的交叉互连总线，不仅可以连接 PCI、PCI-X 和 PCIe 总线控制器，而且可以连接 RapidIO^③总线控制器。使用 OCeaN 进行互连的总线控制器可以直接通信，而不需要通过 SoC 平台总线。

① MPC8548 也将 FSB 称为 CCB (Core Complex Bus)。

② PowerPC 处理器并没有公开其 SoC 平台总线的设计规范。ARM 提出的 AMBA 总线是一条典型的 SoC 平台总线。

③ RapidIO 总线是由 Mercury Computer System 和 Motorola Semiconductor（目前的 Freescale）共同提出，用于解决背板互连的一条外部总线。

如来自 HOST 主桥 1 的数据报文可以通过 OCean 直接发向 HOST 主桥 2，而不需要将数据通过 SoC 平台总线，再进行转发，从而减轻了 SoC 平台总线的负担。OCean 部件的拓扑结构如图 2-3 所示。

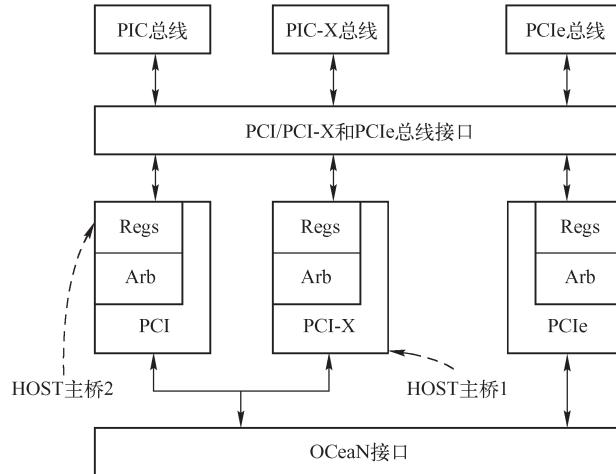


图 2-3 MPC8548 处理器的 HOST 主桥

在 MPC8548 处理器中，有两个 HOST 主桥，分别是 HOST 主桥 1 和 HOST 主桥 2，其中 HOST 主桥 1 可以支持 PCI-X 总线，而 HOST 主桥 2 只能支持 PCI 总线。此外该处理器还含有多个 PCIe 总线控制器。

本节仅介绍 HOST 主桥，即 MPC8548 处理器中的 PCI 总线控制器，而不介绍该处理器的 PCIe 总线控制器。因为从软件层面上看，MPC8548 处理器的 PCIe 总线控制器与 PCI/PCI-X 总线控制器功能类似。

MPC8548 处理器既可以作为 PCI 总线的 HOST 处理器，也可以作为 PCI 总线的从设备，本节仅讲述 MPC8548 处理器如何作为 PCI 总线的 HOST 处理器管理 PCI 总线树，而并不关心 MPC8548 处理器作为从设备的情况。

在 MPC8548 处理器的 HOST 主桥中，定义了一系列与系统软件相关的寄存器。本节将通过介绍这些寄存器，说明这个 HOST 主桥的功能。囿于篇幅，本节仅介绍与 HOST 主桥 1 相关的寄存器，HOST 主桥 2 使用的寄存器与 HOST 主桥 1 使用的寄存器类似。

2.2.1 PCI 设备配置空间的访问机制

PCI 总线规定访问配置空间的总线事务，即配置读写总线事务，使用 ID 号进行寻址。PCI 设备的 ID 号由总线号 (Bus Number)、设备号 (Device Number) 和功能号 (Function Number) 组成。

其中总线号在 HOST 主桥遍历 PCI 总线树时确定。PCI 总线可以使用 PCI 桥扩展 PCI 总线，并形成一棵 PCI 总线树。在一棵 PCI 总线树上，有几个 PCI 桥（包括 HOST 主桥），就有几条 PCI 总线。在一棵 PCI 总线树中，总线号由系统软件决定，通常与 HOST 主桥直接相连的 PCI 总线编号为 0，系统软件使用 DFS (Depth-First Search) 算法扫描 PCI 总线树上的所有 PCI 总线，并依次进行编号。

一条 PCI 总线的设备号由 PCI 设备的 IDSEL 信号与 PCI 总线地址线的连接关系确定，而

功能号与 PCI 设备的具体设计相关。在一个 PCI 设备中最多有 8 个功能设备，而且每一个功能设备都有各自的 PCI 配置空间，而在绝大多数 PCI 设备中只有一个功能设备。HOST 主桥使用寄存器号，访问 PCI 设备配置空间的某个寄存器。

在 MPC8548 处理器的 HOST 主桥中，与 PCI 设备配置空间相关的寄存器由 CFG_ADDR、CFG_DATA 和 INT_ACK 寄存器组成。系统软件使用 CFG_ADDR 和 CFG_DATA 寄存器访问 PCI 设备的配置空间，而使用 INT_ACK 寄存器访问挂接在 PCI 总线上的中断控制器的中断向量，HOST 主桥这 3 个寄存器的地址偏移和属性如表 2-1 所示。

表 2-1 PCI 总线配置寄存器

Offset	寄 存 器	属 性	复 位 值
0x0_8000	CFG_ADDR	可读写	0x0000-0000
0x0_8004	CFG_DATA	可读写	0x0000-0000
0x0_8008	INT_ACK	只读	0x0000-0000

在 MPC8548 处理器中，所有内部寄存器都使用存储器映射方式进行寻址，并存放在以 BASE_ADDR^①变量为起始地址的“1 MB 连续的物理地址空间”中。PowerPC 处理器可以通过 BASE_ADDR + Offset 的方式访问表 2-1 中的寄存器。

MPC8548 处理器使用 CFG_ADDR 寄存器和 CFG_DATA 寄存器访问 PCI 设备的配置空间，其中用 CFG_ADDR 寄存器保存 PCI 设备的 ID 号和寄存器号，该寄存器的各个字段的详细说明如下所示。

- Enable 位。当该位为 1 时，HOST 主桥使能对 PCI 设备配置空间的访问，当 HOST 处理器对 CFG_DATA 寄存器进行访问时，HOST 主桥将对这个寄存器的访问转换为 PCI 配置读写总线事务并发送到 PCI 总线上。
- Bus Number 字段记录 PCI 设备所在的总线号。
- Device Number 字段记录 PCI 设备的设备号。
- Function Number 字段记录 PCI 设备的功能号。
- Register Number 字段记录 PCI 设备的配置寄存器号。

MPC8548 处理器访问 PCI 设备的配置空间时，首先需要在 CFG_ADDR 寄存器中设置这个 PCI 设备对应的总线号、设备号、功能号和寄存器号，然后使能 Enable 位。之后当 MPC8548 处理器对 CFG_DATA 寄存器进行读写访问时，HOST 主桥将这个存储器读写访问转换为 PCI 配置读写请求，并发送到 PCI 总线上。如果 Enable 位没有使能，处理器对 CFG_DATA 的访问不过是一个普通的 I/O 访问，HOST 主桥并不能将其转换为 PCI 配置读写请求。

HOST 主桥根据 CFG_ADDR 寄存器中的 ID 号，生成 PCI 配置读写总线事务，并将这个读写总线事务，通过 ID 译码方式发送到指定的 PCI 设备。PCI 设备将接收来自配置写总线事务的数据，或者为配置读总线事务提供数据。

值得注意的是，在 PowerPC 处理器中，在 CFG_DATA 寄存器中保存的数据采用大端方式进行编址，而 PCI 设备的配置寄存器采用小端编址，因此 HOST 主桥需要进行端模式转换。下面以源代码 2-1 为例说明 PowerPC 处理器如何访问 PCI 配置空间。

① 在 MPC8548 处理器中，BASE_ADDR 存放在 CCSRBAR 寄存器中。

源代码 2-1 PowerPC 处理器访问 PCI 配置空间

```
stw r0,0(r1)
ld r3,0(r2)
```

首先假设寄存器 r1 的初始值为 BASE_ADDR + 0x0_8000 (即 CFG_ADDR 寄存器的地址), 寄存器 r0 的初始值为 0x8000-0008, 寄存器 r2 的初始值为 BASE_ADDR + 0x0_8004 (即 CFG_DATA 寄存器的地址), 而指定 PCI 设备 (总线号、设备号、功能号都为 0) 的配置寄存器的 0x0B ~ 0x08 中的值为 0x9988-7766。

这段源代码的执行步骤如下。

(1) 将 r0 寄存器赋值到 r1 寄存器所指向的地址空间中, 即初始化 CFG_ADDR 寄存器为 0x8000-0008。

(2) 从 r2 寄存器所指向的地址空间中读取数据到 r3 寄存器中, 即从 CFG_DATA 寄存器中读取数据到 r3 寄存器。

在 MPC8548 处理器中, 源代码 2-1 执行完毕后, 寄存器 r3 保存的值为 0x6677-8899, 而不是 0x9988-6677。系统程序员在使用这个返回值时, 一定要注意大小端模式的转换。值得注意的是, 源代码 2-1 可以使用 lwbxx 指令进行优化, 该指令可以在读取数据的同时, 进行大小端模式的转换。

处理器读取 INT_ACK 寄存器时, HOST 主桥将这个读操作转换为 PCI 总线中断响应事务。PCI 总线中断响应事务的作用是通过 PCI 总线读取中断控制器的中断向量号, 这样做的前提是中断控制器需要连接在 PCI 总线上。

PowerPC 处理器使用的 MPIC 中断控制器不是挂接在 PCI 总线上, 而是挂接在 SoC 平台总线上, 因此 PCI 总线提供的中断应答事务在这个处理器系统中并没有太大用途。但是并不排除某些 PowerPC 处理器系统使用了挂接在 PCI 总线上的中断控制器, 比如 PCI 南桥芯片, 此时 PowerPC 处理器系统需要使用中断应答事务读取 PCI 南桥中的中断控制器, 以获取中断向量号。

2.2.2 存储器域地址空间到 PCI 总线域地址空间的转换

MPC8548 处理器使用 ATMU (Address Translation and Mapping Unit) 寄存器组进行存储器域到 PCI 总线域, 以及 PCI 总线域到存储器域的地址映射。ATMU 寄存器组由两大组寄存器组成, 分别为 Outbound 和 Inbound 寄存器组。其中 Outbound 寄存器组将存储器域的地址转换为 PCI 总线域的地址, 而 Inbound 寄存器组将 PCI 总线域的地址转换为存储器域的地址。

在 MPC8548 处理器中, 只有当 CPU 读写访问的地址范围在 Outbound 寄存器组管理的地址空间之内时, HOST 主桥才能接收 CPU 的读写访问, 并将 CPU 在存储器域上的读写访问转换为 PCI 总线域上的读写访问, 然后才能对 PCI 设备进行读写操作。

如图 2-2 所示, CPU 对存储器域的地址访问, 首先使用 CCB 总线事务, 如果所访问的地址在 Cache 中命中, 则从 Cache 中直接获得数据, 否则将从存储器域中获取数据。而在绝大多数情况下, 外部设备使用的地址空间是不可 Cache[⊖] 的, 所以发向 PCI 设备的 CCB 总线

[⊖] PCI 设备使用的 ROM 空间可以是“可 Cache”的地址空间。

事务通常不会与 Cache 进行数据交换。

如果 CCB 总线事务使用的地址在 HOST 主桥的 Outbound 寄存器窗口中命中，HOST 主桥将接收这个 CCB 总线事务，并将其转换为 PCI 总线事务之后，再发送到 PCI 总线上。MPC8548 处理器的每一个 HOST 主桥都提供了 5 个 Outbound 寄存器窗口来实现存储器域地址到 PCI 总线域地址的映射，其映射过程如图 2-4 所示。

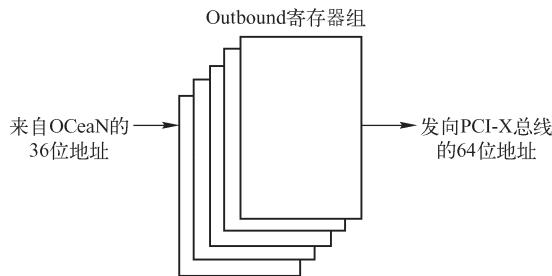


图 2-4 MPC8548 处理器存储器域到 PCI 域的转换

在介绍 MPC8548 处理器如何使用 Outbound 寄存器组进行存储器域地址空间到 PCI 总线域地址空间的转换之前，本节将首先介绍 Outbound 寄存器组中的相应寄存器。Outbound 寄存器组的地址偏移、属性和复位值如表 2-2 所示。

表 2-2 PCI/X ATMU Outbound 寄存器组

地址偏移	寄存器名	属性	复位值
0x0_8C00/20/40/60/80	POTARn	可读写	0x0000-0000
0x0_8C04/24/44/64/84	POTEARN	可读写	0x0000-0000
0x0_8C28/48/68/88	POWBArn	可读写	0x0000-0000
0x0_8C30/50/70/90	POWARn	可读写	0x0000-0000

1. POTARn 和 POTEARN 寄存器

在 POTARn 和 POTEARN 寄存器中保存当前 Outbound 窗口在 PCI 总线域中的 64 位地址空间的基址。这两个寄存器的主要字段如下。

- POTARn 寄存器的 TEA 字段，第 0 ~ 11 位，保存 PCI 总线地址空间的 43 ~ 32 位。
- POTARn 寄存器的 TA 字段，第 12 ~ 31 位，保存 PCI 总线地址空间的 31 ~ 12 位^①。
- POTEARN 寄存器的 TEA 字段，第 12 ~ 31 位，保存 PCI 总线地址空间的 63 ~ 44 位。

2. POWBArn 和 POWARn 寄存器

而 POWBArn 寄存器保存当前 Outbound 窗口在存储器域中的 36 位地址空间的基址^②，其主要字段如下。

- WBEA 字段保存存储器域地址的第 0 ~ 3 位。
- WBA 字段保存存储器域地址的第 4 ~ 23^③位。

① POTARn 寄存器没有保存 PCI 总线的 11 ~ 0 位，因为 Outbound 窗口大小至少为 4 KB。

② MPC8548 处理器的物理地址为 36 位。注意在 PowerPC 处理器中，第 0 位是地址的最高位。

③ WBA 字段并没有保存存储器域的第 24 ~ 35 位地址，因为 Outbound 窗口大小至少为 4 KB 的整数倍。

POWARn 寄存器描述 Outbound 窗口的属性，其主要字段如下。

- EN 位，第 0 位。该位是 Outbound 窗口的使能位，为 1 表示当前 Outbound 寄存器组描述的存储器地址空间到 PCI 总线地址空间的映射关系有效；为 0 表示无效。
- RTT 字段，第 12 ~ 15 位，该字段描述当前窗口的读传送类型，为 0b0100 表示存储器读，为 0b1000 表示 I/O 读。
- WTT 字段，第 16 ~ 19 位，该字段描述当前窗口的写传送类型，为 0b0100 表示存储器写，为 0b1000 表示 I/O 写。在 PCIe 总线控制器中，RTT 字段和 WTT 字段还可以支持对配置空间的读写操作。
- OWS 字段，第 26 ~ 31 位，该字段描述当前窗口的大小，Outbound 窗口的大小在 4 KB ~ 64 GB 之间，其值为 2^{OWS+1} 。

3. 使用 Outbound 寄存器访问 PCI 总线地址空间

MPC8548 处理器使用 Outbound 寄存器组访问 PCI 总线地址空间的步骤如下。

(1) 首先 MPC8548 处理器需要将程序使用的 32 位有效地址 EA (Effective Address) 转换为 41 位的虚拟地址 VA (Virtual Address)。E500 V2 内核不能关闭 MMU (Memory Management Unit)，因此不能直接访问物理地址。

(2) MPC8548 处理器通过 MMU 将 41 位的虚拟地址转换为 36 位的物理地址。在 E500 V2 内核中，物理地址是 36 位（缺省是 32 位，需要使能）。

(3) 检查 LAWBAR 和 LAWAR 寄存器，判断当前 36 位的物理地址是否属于 PCI 总线空间。在 MPC8548 中定义了一组 LAWBAR 和 LAWAR 寄存器对，每一对寄存器描述当前物理空间是与 PCI 总线、PCIe 总线、DDR 还是 RapidIO 空间对应。该组寄存器的详细说明见 MPC8548 PowerQUICC III™ Integrated Host Processor Family Reference Manual。如果 CPU 访问的空间为 PCI 总线空间，则执行第 (4) 步，否则处理器将不会访问 PCI 地址空间。

(4) 判断当前 36 位物理地址是否在 POWBARn 寄存器 1 ~ 4 描述的窗口中，如果在则将 36 位的处理器物理地址通过寄存器 POTARn 和 POTEARn 转换为 64 位的 PCI 总线地址，然后 HOST 主桥将来自处理器的读写请求发送到 PCI 总线上；如果不在 POWBARn 寄存器 1 ~ 4 描述的窗口中，POWBAR0 寄存器作为缺省窗口，接管这个存储器访问，并使用寄存器 POTAR0 和 POTEAR0，将处理器物理地址转换为 PCI 总线地址，当然在正常设计中很少出现这种情况。

许多系统软件，将 Outbound 窗口两边的寄存器使用“直接相等”的方法进行映射，将存储器域的地址与 PCI 总线地址设为相同的值。但是系统软件程序员务必注意这个存储器地址与 PCI 总线地址是分属于存储器域与 PCI 总线域的，这两个值虽然相等，但是所代表的地址并不相同，一个属于存储器域，而另一个属于 PCI 总线域。

2.2.3 PCI 总线域地址空间到存储器域地址空间的转换

MPC8548 处理器使用 Inbound 寄存器组将 PCI 总线域地址转换为存储器域的地址。PCI 设备进行 DMA 读写时，只有访问的地址在 Inbound 窗口中时，HOST 主桥才能接收这些读写请求，并将其转发到存储器控制器。MPC8548 处理器提供了 3 组 Inbound 寄存器，即提供 3 个 Inbound 寄存器窗口，实现 PCI 总线地址到存储器地址的反向映射。

从 PCI 设备的角度来看，PCI 设备访问存储器域的地址空间时，首先需要通过 Inbound 窗口将 PCI 总线地址转换为存储器域的地址；而从处理器的角度来看，处理器必须将存储器地址通过 Inbound 寄存器组反向映射为 PCI 总线地址空间，才能被 PCI 设备访问。

PCI 设备只能使用 PCI 总线地址访问 PCI 总线域的地址空间。HOST 主桥将这段地址空间通过 Inbound 窗口转换为存储器域的地址之后，PCI 设备才能访问存储器域地址空间。这个地址转换过程如图 2-5 所示。

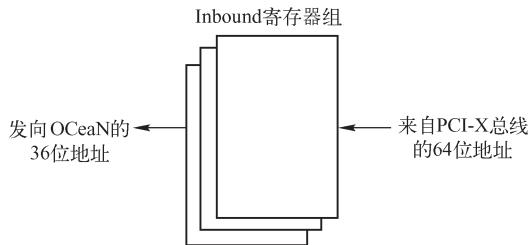


图 2-5 MPC8548 处理器 PCI 域到存储器域的转换

在介绍 MPC8548 处理器如何使用 Inbound 寄存器组进行 PCI 总线域地址空间到存储器域地址空间的转换之前，我们首先简要介绍 Inbound 寄存器组中的相应寄存器。该组寄存器的地址偏移、属性和复位值如表 2-3 所示。

表 2-3 PCI/X ATMU Inbound 寄存器组

Offset	寄存器名	属性	复位值
0x0_8DA0/C0/E0	PITARn	可读写	0x0000-0000
0x0_8DA8/C8/E8	PIWBArn	可读写	0x0000-0000
0x0_8DAC/CC	PIWBearn	可读写	0x0000-0000
0x0_8DB0/D0/F0	PIWARn	可读写	0x0000-0000

值得注意的是，Inbound 寄存器组除了可以进行 PCI 总线地址空间到存储器域地址空间的转换之外，还可以转换分属不同 PCI 总线域的地址空间，以支持 PCI 总线的 Peer-to-Peer 数据传送方式。

1. PITARn 寄存器

PITARn 寄存器保存当前 Inbound 窗口在存储器域中的 36 位地址空间的基址，其地址窗口的大小至少为 4 KB，因此在该寄存器中仅存放存储器域地址的第 0 ~ 23 位，该寄存器的主要字段如下所示。

- TEA 字段存放存储器地址空间的第 0 ~ 3 位。
- TA 字段存放存储器地址空间的第 4 ~ 23 位。

2. PIWBArn 和 PIWBearn 寄存器

PIWBArn 和 PIWBearn 寄存器保存当前 Inbound 窗口在 PCI 总线域中的 64 位地址空间的基址的第 63 ~ 12 位，Inbound 窗口使用的最小地址空间为 4 KB，因此在这两个寄存器中不含有 PCI 总线地址空间的第 11 ~ 0 位。这两个寄存器的主要字段如下所示。

- PIWBArn 寄存器的 BEA 字段存放 PCI 总线地址空间的第 43 ~ 32 位。
- PIWBArn 寄存器的 BA 字段存放 PCI 总线地址空间的第 31 ~ 12 位。

- PIWBARN 寄存器的 BEA 字段存放 PCI 总线地址空间的第 63 ~ 44 位。

3. PIWARn 寄存器

PIWARn 寄存器描述当前 Inbound 窗口的属性，该寄存器由以下位和字段组成。

- EN 位，第 0 位。该位是 Inbound 窗口的使能位，为 1 表示当前 Inbound 寄存器组描述的存储器地址空间到 PCI 总线地址空间的映射关系有效；为 0 表示无效。
- PF 位，第 2 位。该位为 1 表示当前 Inbound 窗口描述的存储区域支持预读；为 0 表示不支持预读。
- TGI 字段，第 8 ~ 11 位。该字段为 0b0010 表示当前 Inbound 窗口描述的存储区域属于 PCIe 总线域地址空间；为 0b1100 表示当前 Inbound 窗口描述的存储区域属于 RapidIO 总线域地址空间。该字段对于 OCean 实现不同域间的报文转发非常重要，如果当前 Inbound 窗口的 TGI 字段为 0b0010，此时 PCI 总线上的设备可以使用该 Inbound 窗口，通过 OCean 直接读取 PCIe 总线的地址空间，而不需要经过 SoC 平台总线。如果 TGI 字段为 0b1111 表示 Inbound 窗口描述的存储器区域属于主存储器地址空间，这也是最常用的方式。使用该字段可以实现 HOST 主桥的 Peer-to-Peer 数据传送方式。
- RTT 字段和 WTT 字段，分别为该寄存器的第 12 ~ 15 位和第 16 ~ 19 位。Inbound 窗口的 RTT/WTT 字段的含义与 Outbound 窗口的 RTT/WTT 字段基本类似。只是在 Inbound 窗口中可以规定 PCI 设备访问主存储器时，是否需要进行 Cache 一致性操作（Cache Lock and Allocate），在进行 DMA 写操作时，数据是否可以直接进入 Cache。该字段是 PowerPC 处理器对 PCI 总线规范的有效补充，由于该字段的存在，PowerPC 处理器的 PCI 设备可以将数据直接写入 Cache，也可以视情况决定 DMA 操作是否需要进行 Cache 共享一致性操作。
- IWS 字段，第 26 ~ 31 位。该字段描述当前窗口的大小，Inbound 窗口的大小在 4 KB ~ 16 GB 之间，其值为 2^{IWS+1} 。

4 使用 Inbound 寄存器组进行 DMA 操作

PCI 设备使用 DMA 操作访问主存储器空间，或者访问其他 PCI 总线域地址空间时，需要通过 Inbound 窗口，其步骤如下。

(1) PCI 设备在访问主存储器空间时，将首先检查当前 PCI 总线地址是否在 PIWBARN 和 PIWBARN 寄存器描述的窗口中。如果在这个窗口中，则将这个 PCI 总线地址通过 PITARn 寄存器转换为存储器域的地址或者其他 PCI 总线域的地址；如果不在将禁止本次访问。

(2) 如果 PCI 设备访问的是存储器地址空间，HOST 主桥将来自 PCI 总线的读写请求发送到存储器空间，进行存储器读写操作，并根据 Inbound 寄存器组的 RTT/WTT 位决定是否需要进行 Cache 一致性操作，或者将数据直接写入到 Cache 中。

结合 Outbound 寄存器组，可以发现 PCI 总线地址空间与存储器地址空间是有一定联系的。如果存储器域地址空间被 Inbound 寄存器组反向映射到 PCI 空间，这个存储器地址具有两个地址，一个是在存储器域的地址，一个是在 PCI 总线域的地址；同理 PCI 总线空间的地址如果使用 Outbound 寄存器映射到寄存器地址空间，这个 PCI 总线地址也具有两个地址，一个是在 PCI 总线域的地址，一个是在存储器域的地址。

能够被处理器和 PCI 总线同时访问的地址空间，一定在 PCI 总线域和存储器域中都存在

地址映射。再次强调，绝大多数操作系统将同一个空间的 PCI 总线域地址和存储器地址设为相同的值，但是这两个相同的值所代表的含义不同。

由此可以看出，如果 MPC8548 处理器的某段存储器区域没有在 Inbound 窗口中定义时，PCI 设备将不能使用 DMA 机制访问这段存储器空间；同理如果 PCI 设备的空间不在 Outbound 窗口，HOST 处理器也不能访问这段 PCI 地址空间。

在绝大多数 PowerPC 处理器系统中，PCI 设备地址空间都在 HOST 主桥的 Outbound 窗口中建立了映射；而 MPC8548 处理器可以选择将哪些主存储器空间共享给 PCI 设备，从而对主存储器空间进行保护。

2.2.4 x86 处理器的 HOST 主桥

x86 处理器使用南北桥结构连接 CPU 和 PCI 设备。其中北桥（North Bridge）连接快速设备，如显卡和内存条，并推出 PCI 总线，HOST 主桥包含在北桥中。而南桥（South Bridge）连接慢速设备。x86 处理器使用的南北桥结构如图 2-6 所示。

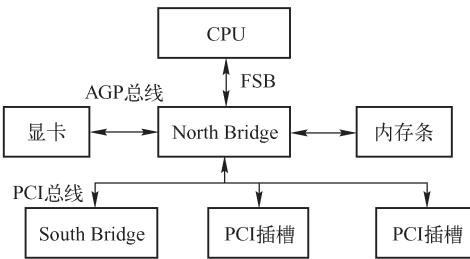


图 2-6 x86 处理器的南北桥结构

Intel 使用南北桥概念统一 PC 架构。但是从体系结构的角度上看，南北桥架构并不重要，北桥中存放的主要部件不过是存储器控制器、显卡控制器和 HOST 主桥而已，而南桥存放的是一些慢速设备，如 ISA 总线和中断控制器等。

不同的处理器系统集成这些组成部件的方式并不相同，如 PowerPC、MIPS 和 ARM 处理器系统通常将 CPU 和主要外部设备都集成到一颗芯片中，组成一颗基于 SoC 架构的处理器系统。这些集成方式并不重要，每一个处理器系统都有其针对的应用领域，不同应用领域的需求对处理器系统的集成方式有较大的影响。Intel 采用的南北桥架构针对 x86 处理器的应用领域而设计，并不能说采用这种结构一定比 MPC8548 处理器中既含有 HOST-to-PCI 主桥也含有 HOST-to-PCIe 主桥更为合理。

在许多嵌入式处理器系统中，既含有 PCI 设备也含有 PCIe 设备，为此 MPC8548 处理器同时提供了 PCI 总线和 PCIe 总线接口，在这个处理器系统中，PCI 设备可以与 PCI 总线直接相连，而 PCIe 设备可以与 PCIe 总线直接相连，因此并不需要使用 PCIe 桥扩展 PCI 总线，从而在一定程度上简化了嵌入式系统的设计。

嵌入式系统所面对的应用千变万化，进行芯片设计时所要考虑的因素相对较多，因而在某种程度上为设计带来了一些难度。而 x86 处理器系统所面对的应用领域针对个人 PC 和服务器，向前兼容和通用性显得更加重要。在多数情况下，一个通用处理器系统的设计难度超过专用处理器系统的设计，Intel 为此付出了极大的代价。

在一些相对较老的北桥中，如 Intel 440 系列芯片组中包含了 HOST 主桥，从系统软件的角度上看 HOST-to-PCI 主桥实现的功能与 HOST-to-PCIe 主桥实现的功能相近。本节仅简单介绍 Intel 的 HOST-to-PCI 主桥如何产生 PCI 的配置周期，有关 Intel HOST-to-PCIe 主桥[⊖]的详细信息参见第 5 章。

x86 处理器定义了两个 I/O 端口寄存器，分别为 CONFIG_ADDRESS 和 CONFIG_DATA 寄存器，其地址为 0xCF8 和 0xCFC。x86 处理器使用这两个 I/O 端口访问 PCI 设备的配置空间。PCI 总线规范也以这两个寄存器为例，说明处理器如何访问 PCI 设备的配置空间。其中 CONFIG_ADDRESS 寄存器存放 PCI 设备的 ID 号，而 CONFIG_DATA 寄存器存放进行配置读写的数据。

CONFIG_ADDRESS 寄存器与 PowerPC 处理器中的 CFG_ADDR 寄存器的使用方法类似，而 CONFIG_DATA 寄存器与 PowerPC 处理器中的 CFG_DATA 寄存器的使用方法类似。CONFIG_ADDRESS 寄存器的结构如图 2-7 所示。

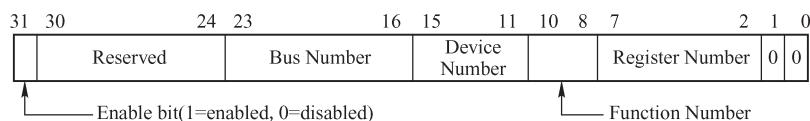


图 2-7 CONFIG_ADDRESS 寄存器的结构

CONFIG_ADDRESS 寄存器的各个字段和位的说明如下所示。

- Enable 位，第 31 位。该位为 1 时，对 CONFIG_DATA 寄存器进行读写时将引发 PCI 总线的配置周期。
- Bus Number 字段，第 23 ~ 16 位，记录 PCI 设备的总线号。
- Device Number 字段，第 15 ~ 11 位，记录 PCI 设备的设备号。
- Function Number 字段，第 10 ~ 8 位，记录 PCI 设备的功能号。
- Register Number 字段，第 7 ~ 2 位，记录 PCI 设备的寄存器号。

当 x86 处理器对 CONFIG_DATA 寄存器进行 I/O 读写访问，且 CONFIG_ADDR 寄存器的 Enable 位为 1 时，HOST 主桥将这个 I/O 读写访问转换为 PCI 配置读写总线事务，然后发送到 PCI 总线上，PCI 总线根据保存在 CONFIG_ADDR 寄存器中的 ID 号，将 PCI 配置读写请求发送到指定 PCI 设备的指定配置寄存器中。

x86 处理器使用小端地址模式，因此从 CONFIG_DATA 寄存器中读出的数据不需要进行模式转换，这点和 PowerPC 处理器不同，此外 x86 处理器的 HOST 主桥也实现了存储器域到 PCI 总线域的地址转换，但是这个概念在 x86 处理器中并不明晰。

本书将在第 5 章以 HOST-to-PCIe 主桥为例，详细介绍 Intel 处理器的存储器地址与 PCI 总线地址的转换关系，而在本节不对 x86 处理器的 HOST 主桥做进一步说明。x86 处理器系统的升级速度较快，目前在 x86 的处理器体系结构中，已很难发现 HOST 主桥的身影。

目前 Intel 对南北桥架构进行了升级，其中北桥被升级为 MCH (Memory Controller Hub)，而南桥被升级为 ICH (I/O Controller Hub)。x86 处理器系统在 MCH 中集成了存储器控制器、

[⊖] 这个 HOST-to-PCIe 主桥也是 RC (Root Complex) 的一部分。

显卡芯片和 HOST-to-PCIe 主桥，并通过 Hub Link 与 ICH 相连；而在 ICH 中集成了一些相对低速总线接口，如 AC'97、LPC（Low Pin Count）、IDE 和 USB 总线，当然也包括一些低带宽的 PCIe 总线接口。

在 Intel 最新的 Nehalem[⊖]处理器系统中，MCH 被一分为二，存储器控制器和图形控制器已经与 CPU 内核集成在一个 DIE 中，而 MCH 剩余的部分与 ICH 合并成为 PCH（Peripheral Controller Hub）。但是从体系结构的角度上看，这些升级与整合并不重要。

目前 Intel 在 Menlow[⊖]平台基础上，计划推出基于 SoC 架构的 x86 处理器，以进军手持设备市场。在基于 SoC 构架的 x86 处理器中将逐渐淡化 Chipset 的概念，其拓扑结构与典型的 SoC 处理器，如 ARM 和 PowerPC 处理器，较为类似。

2.3 PCI 桥与 PCI 设备的配置空间

PCI 设备都有独立的配置空间，HOST 主桥通过配置读写总线事务访问这段空间。PCI 总线规定了三种类型的 PCI 配置空间，分别是 PCI Agent 设备使用的配置空间，PCI 桥使用的配置空间和 Cardbus 桥片使用的配置空间。

本节重点介绍 PCI Agent 和 PCI 桥使用的配置空间，而并不介绍 Cardbus 桥片使用的配置空间。值得注意的是，在 PCI 设备配置空间中出现的地址都是 PCI 总线地址，属于 PCI 总线域地址空间。

2.3.1 PCI 桥

PCI 桥的引入使 PCI 总线极具扩展性，也极大地增加了 PCI 总线的复杂度。PCI 总线的电气特性决定了在一条 PCI 总线上挂接的负载有限，当 PCI 总线需要连接多个 PCI 设备时，需要使用 PCI 桥进行总线扩展，扩展出的 PCI 总线可以连接其他 PCI 设备，包括 PCI 桥。在一棵 PCI 总线树上，最多可以挂接 256 个 PCI 设备，包括 PCI 桥。PCI 桥在 PCI 总线树中的位置如图 2-8 所示。

PCI 桥作为一个特殊的 PCI 设备，具有独立的配置空间。但是 PCI 桥配置空间的定义与 PCI Agent 设备有所不同。PCI 桥的配置空间可以管理其下 PCI 总线子树的 PCI 设备，并可以优化这些 PCI 设备通过 PCI 桥的数据访问。PCI 桥的配置空间在系统软件遍历 PCI 总线树时进行配置，系统软件不需要专门的驱动程序设置 PCI 桥的使用方法，这也是 PCI 桥被称为透明桥的主要原因。

在某些处理器系统中，还有一类 PCI 桥，叫做非透明桥。非透明桥不是 PCI 总线定义的标准桥片，但是在使用 PCI 总线挂接另外一个处理器系统时非常有用，非透明桥片的主要作用是连接两个不同的 PCI 总线域，进而连接两个处理器系统，本章将在第 2.5 节中详细介绍 PCI 非透明桥。

使用 PCI 桥可以扩展出新的 PCI 总线，在这条 PCI 总线上还可以继续挂接多个 PCI 设

[⊖] Nehalem 处理器也称为 Core i7 处理器。

[⊖] Menlow 平台于 2008 年 3 月发布，其目标应用为 MID（Mobile Internet Device）设备。Menlow 平台基于低功耗处理器内核 Atom。

备。PCI 桥跨接在两个 PCI 总线之间，其中距离 HOST 主桥较近的 PCI 总线被称为该桥片的上游总线（Primary Bus），距离 HOST 主桥较远的 PCI 总线被称为该桥片的下游总线（Secondary Bus）。如图 2-8 所示，PCI 桥 1 的上游总线为 PCI 总线 x0，而 PCI 桥 1 的下游总线为 PCI 总线 x1。这两条总线间的数据通信需要通过 PCI 桥 1。

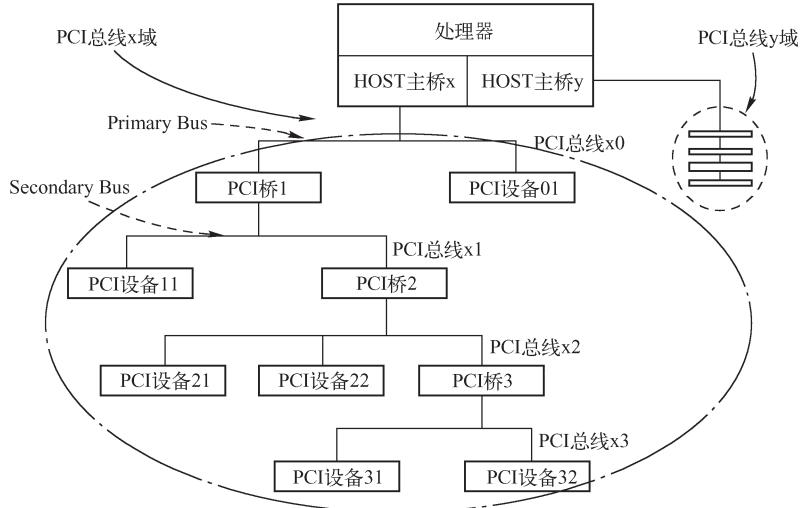


图 2-8 使用 PCI 桥扩展 PCI 总线

通过 PCI 桥连接的 PCI 总线属于同一个 PCI 总线域，在图 2-8 中，PCI 桥 1、2 和 3 连接的 PCI 总线都属于 PCI 总线 x 域。在这些 PCI 总线域上的设备可以通过 PCI 桥直接进行数据交换而不需要进行地址转换；而分属不同 PCI 总线域的设备间的通信需要进行地址转换，如与 PCI 非透明桥两端连接的设备之间的通信。

如图 2-8 所示，每一个 PCI 总线的下方都可以挂接一个到多个 PCI 桥，每一个 PCI 桥都可以推出一条新的 PCI 总线。在同一条 PCI 总线上的设备之间的数据交换不会影响其他 PCI 总线。如 PCI 设备 21 与 PCI 设备 22 之间的数据通信仅占用 PCI 总线 x2 的带宽，而不会影响 PCI 总线 x0、x1 与 x3，这也是引入 PCI 桥的一个重要原因。

由图 2-8 还可以发现 PCI 总线可以通过 PCI 桥组成一个胖树结构，其中每一个桥片都是父节点，而 PCI Agent 设备只能是子节点。当 PCI 桥出现故障时，其下的设备不能将数据传递给上游总线，但是并不影响 PCI 桥下游设备间的通信。当 PCI 桥 1 出现故障时，PCI 设备 11、PCI 设备 21 和 PCI 设备 22 将不能与 PCI 设备 01 和存储器进行通信，但是 PCI 设备 21 和 PCI 设备 22 之间的通信可以正常进行。

使用 PCI 桥可以扩展一条新的 PCI 总线，但是不能扩展新的 PCI 总线域。如果当前系统使用 32 位的 PCI 总线地址，那么这个系统的 PCI 总线域的地址空间为 4 GB 大小，在这个总线域上的所有设备将共享这个 4GB 大小的空间。如在 PCI 总线 x 域上的 PCI 桥 1、PCI 设备 01、PCI 设备 11、PCI 桥 2、PCI 设备 21 和 PCI 设备 22 等都将共享一个 4 GB 大小的空间。再次强调这个 4 GB 空间是 PCI 总线 x 域的“PCI 总线地址空间”，和存储器域地址空间和 PCI 总线 y 域没有直接联系。

处理器系统可以通过 HOST 主桥扩展出新的 PCI 总线域，如 MPC8548 处理器的 HOST 主

桥 x 和 y 可以扩展出两个 PCI 总线域 x 和 y。这两个 PCI 总线域 x 和 y 之间的 PCI 空间在正常情况下不能直接进行数据交换，但是 PowerPC 处理器可以通过设置 PIWARn 寄存器的 TGI 字段使得不同 PCI 总线域的设备直接通信，详见第 2.2.3 节。

许多处理器系统使用的 PCI 设备较少，因而并不需要使用 PCI 桥。因此在这些处理器系统中，PCI 设备都是直接挂接在 HOST 主桥上，而不需要使用 PCI 桥扩展新的 PCI 总线。即便如此读者也需要深入理解 PCI 桥的知识。

PCI 桥对于理解 PCI 和 PCIe 总线都非常重要。在 PCIe 总线中，虽然在物理结构上并不含有 PCI 桥，但是与 PCI 桥相关的知识在 PCIe 总线中无处不在，比如在 PCIe 总线的 Switch 中，每一个端口都与一个虚拟 PCI 桥对应，Switch 使用这个虚拟 PCI 桥管理其下 PCI 总线子树的地址空间。

2.3.2 PCI Agent 设备的配置空间

在一个具体的处理器应用中，PCI 设备通常将 PCI 配置信息存放在 E²PROM 中。PCI 设备进行上电初始化时，将 E²PROM 中的信息读到 PCI 设备的配置空间中作为初始值。这个过程由硬件逻辑完成，绝大多数 PCI 设备使用这种方式初始化其配置空间。

读者可能会对这种机制产生一个疑问，如果系统软件在 PCI 设备将 E²PROM 中的信息读到配置空间之前，就开始操作配置空间，会不会带来问题？因为此时 PCI 设备的初始值并不“正确”，仅仅是 PCI 设备使用的复位值。

读者的这种担心是多余的，因为 PCI 设备在配置寄存器没有初始化完毕之前，即 E²PROM 中的内容没有导入 PCI 设备的配置空间之前，可以使用 PCI 总线规定的“Retry”周期使 HOST 主桥在合适的时机重新发起配置读写请求。

在 x86 处理器中，系统软件使用 CONFIG_ADDR 和 CONFIG_DATA 寄存器，读取 PCI 设备配置空间的这些初始化信息，然后根据处理器系统的实际情况使用 DFS 算法，初始化处理器系统中所有 PCI 设备的配置空间。

在 PCI Agent 设备的配置空间中包含了许多寄存器，这些寄存器决定了该设备在 PCI 总线中的使用方法，本节不会全部介绍这些寄存器，因为系统软件只对部分配置寄存器感兴趣。PCI Agent 设备使用的配置空间如图 2-9 所示。

在 PCI Agent 设备配置空间中包含的寄存器如下所示。

(1) Device ID 和 Vendor ID 寄存器

这两个寄存器的值由 PCISIG 分配，只读。其中 Vendor ID 代表 PCI 设备的生产厂商，而 Device ID 代表这个厂商所生产的具体设备。如 Intel 公司的基于 82571EB 芯片的系列网卡，其 Vendor ID 为 0x8086^①，而 Device ID 为 0x105E^②。

其中 0x8086 代表 Intel，0x105E 代表 82571EB 网卡芯片。Intel 将 0x10xx 作为 LAN 设备的 Device ID。Intel 在 PCISIG 上注册了多如牛毛的 Device ID，这些 Device ID 放在一起，几页纸也列不完。不过 16 位的 Device ID 即便对于 Intel 这样大的公司，也基本没有用完的可能。当 Vendor ID 寄存器为 0xFFFF 时，表示为无效 Vendor ID。

① PCI SIG 分配给 Intel 的 Vendor ID 号是 0x8086，8086 处理器也是 Intel 设计的第一个 PC 处理器。

② 这仅是 Intel 为 82571 的 Copper 口分配的 Vendor ID。

31	24 ~ 23	16 ~ 15	8 ~ 7	0	
	Device ID		Vendor ID		0x00
	Status		Command		0x04
	Class Code		Revision ID		0x08
BIST	Header Type	Latency Timer	Cache Line Size		0x0C
	Base Address Register0				0x10
	Base Address Register1				0x14
	Base Address Register2				0x18
	Base Address Register3				0x1C
	Base Address Register4				0x20
	Base Address Register5				0x24
	Cardbus CIS Pointer				0x28
	Subsystem ID		Subsystem Vendor ID		0x2C
	Expansion ROM Base Address				0x30
	Reserved		Capabilities Pointer		0x34
	Reserved				0x38
MAX_Lat	Min_Gnt	Interrupt Pin	Interrupt Line		0x3C

图 2-9 PCI Agent 设备的配置空间

(2) Revision ID 和 Class Code 寄存器

这两个寄存器只读。其中 Revision ID 寄存器记载 PCI 设备的版本号。该寄存器可以被认为是 Device ID 寄存器的扩展。

而 Class Code 寄存器记载 PCI 设备的分类，该寄存器由三个字段组成，分别是 Base Class Code、Sub Class Code 和 Interface。其中 Base Class Code 将 PCI 设备分类为显卡、网卡、PCI 桥等设备；Sub Class Code 对这些设备进一步细分；而 Interface 定义编程接口。Class Code 寄存器可供系统软件识别当前 PCI 设备的分类。

除此之外硬件逻辑设计也需要使用该寄存器识别不同的设备。当 Base Class Code 寄存器为 0x06，Sub Class Code 寄存器为 0x04 时，如果 Interface 寄存器为 0x00，表示当前 PCI 设备为一个标准的 PCI 桥；如果 Interface 寄存器为 0x01，表示当前 PCI 设备为一个使用“负向译码”的 PCI 桥。

硬件逻辑需要根据这些寄存器判断当前 PCI 桥的使用方式，许多 PCI 桥既可以支持“正向”译码，也可以支持“负向”译码，系统软件必须合理设置 Class Code 寄存器。有关正向译码与负向译码的详细说明见第 3.2.1 节。

(3) Header Type 寄存器

该寄存器只读，由 8 位组成。

- 第 7 位为 1 表示当前 PCI 设备是多功能设备，为 0 表示为单功能设备。
- 第 6 ~ 0 位表示当前配置空间的类型，为 0 表示该设备使用 PCI Agent 设备的配置空间，普通 PCI 设备都使用这种配置头；为 1 表示使用 PCI 桥的配置空间，PCI 桥使用这种配置头；为 2 表示使用 Cardbus 桥片的配置空间，Card Bus 桥片使用这种配置头，本书对这类配置头不作详解。

系统软件需要使用该寄存器区分不同类型的 PCI 配置空间，该寄存器的初始化必须与 PCI 设备的实际情况对应，而且必须为一个合法值。

(4) Cache Line Size 寄存器

该寄存器记录 HOST 处理器使用的 Cache 行长度。在 PCI 总线中和 Cache 相关的总线事务，如存储器写并无效和 Cache 多行读等总线事务需要使用这个寄存器。值得注意的是，该寄存器由系统软件设置，但是在 PCI 设备的运行过程中，只有其硬件逻辑才会使用该寄存器，比如 PCI 设备的硬件逻辑需要得知处理器系统 Cache 行的大小，才能进行存储器写并无效总线事务，单行读和多行读总线事务。

如果 PCI 设备不支持与 Cache 相关的总线事务，系统软件可以不设置该寄存器，此时该寄存器为初始值 0x00。对于 PCIe 设备，该寄存器的值无意义，因为 PCIe 设备在进行数据传送时，在其报文中含有一条数据传送的大小，PCIe 总线控制器可以使用这个“大小”，判断数据区域与 Cache 行的对应关系。

(5) Subsystem ID 和 Subsystem Vendor ID 寄存器

这两个寄存器和 Device ID 及 Vendor ID 类似，也是记录 PCI 设备的生产厂商和设备名称。但是这两个寄存器和 Device ID 及 Vendor ID 寄存器略有不同。下面以一个实例说明 Subsystem ID 和 Subsystem Vendor ID 的用途。

Xilinx 公司在 FPGA 中集成了一个 PCIe 总线接口的 IP 核，即 LogiCORE。用户可以使用 LogiCORE 设计各种各样基于 PCIe 总线的设备，但是这些设备的 Device ID 都是 0x10EE，而 Vendor ID 为 0x0007^①。

因此仅使用 Device ID 和 Vendor ID 寄存器无法区分这些设备。此时必须使用 Subsystem ID 和 Subsystem Vendor ID。如果 Intel 也使用 LogiCORE 设计一款网卡适配器，那么这个基于 LogiCORE 的网卡适配器的 Subsystem Vendor ID 寄存器为 0x8086，而 Subsystem ID 寄存器将是 0x10xx。

(6) Expansion ROM base address 寄存器

有些 PCI 设备在处理器还没有运行操作系统之前，就需要完成基本的初始化设置，比如显卡、键盘和硬盘等设备。为了实现这个“预先执行”功能，PCI 设备需要提供一段 ROM 程序，而处理器在初始化过程中将运行这段 ROM 程序，初始化这些 PCI 设备。Expansion ROM base address 记载这段 ROM 程序的基地址。

(7) Capabilities Pointer 寄存器

在 PCI 设备中，该寄存器是可选的，但是在 PCI-X 和 PCIe 设备中必须支持这个寄存器，Capabilities Pointer 寄存器存放 Capabilities 寄存器组的基地址，PCI 设备使用 Capabilities 寄存器组存放一些与 PCI 设备相关的扩展配置信息。该组寄存器的详细说明见第 4.3 节。

(8) Interrupt Line 寄存器

这个寄存器是系统软件对 PCI 设备进行配置时写入的，该寄存器记录当前 PCI 设备使用的中断向量号，设备驱动程序可以通过这个寄存器，判断当前 PCI 设备使用处理器系统中的哪个中断向量号，并将驱动程序的中断服务例程注册到操作系统中^②。

该寄存器由系统软件初始化，其保存的值与 8259A 中断控制器相关，该寄存器的值也是由 PCI 设备与 8259A 中断控制器的连接关系决定的。如果在一个处理器系统中，没有使

^① Xilinx 使用的 Device ID 号为 0x10EE，而 LogiCORE 的 Vendor ID 号为 0x0007。

^② Linux 系统使用 request_irq 函数注册一个设备的中断服务例程。

用 8259A 中断控制器管理 PCI 设备的中断，则该寄存器中的数据并没有意义。

在多数 PowerPC 处理器系统中，并不使用 8259A 中断控制器管理 PCI 设备的中断请求，因此该寄存器没有意义。即使在 x86 处理器系统中，如果使用 I/O APIC 中断控制器，该寄存器保存的内容仍然无效。目前在绝大多数处理器系统中，并没有使用该寄存器存放 PCI 设备使用的中断向量号。

(9) Interrupt Pin 寄存器

这个寄存器保存 PCI 设备使用的中断引脚。PCI 总线提供了四个中断引脚：INTA#、INTB#、INTC#和 INTD#。Interrupt Pin 寄存器为 1 时表示使用 INTA#引脚向中断控制器提交中断请求，为 2 表示使用 INTB#，为 3 表示使用 INTC#，为 4 表示使用 INTD#。

如果 PCI 设备只有一个子设备时，该设备只能使用 INTA#；如果有多个子设备时，可以使用 INTB ~ D#信号。如果 PCI 设备不使用这些中断引脚，向处理器提交中断请求时，该寄存器的值必须为 0。值得注意的是，虽然在 PCIe 设备中并不含有 INTA ~ D#信号，但是依然可以使用该寄存器，因为 PCIe 设备可以使用 INTx 中断消息，模拟 PCI 设备的 INTA ~ D#信号，详见第 6.3.4 节。

(10) Base Address Register 0 ~ 5 寄存器

该组寄存器简称为 BAR 寄存器，BAR 寄存器保存 PCI 设备使用的地址空间的基址址，该基址址保存的是该设备在 PCI 总线域中的地址。其中每一个设备最多可以有 6 个基址空间，但多数设备不会使用这么多组地址空间。

在 PCI 设备复位之后，该寄存器将存放 PCI 设备需要使用的基址空间大小，这段空间是 I/O 空间还是存储器空间[⊖]，如果是存储器空间该空间是否可预取，有关 PCI 总线预读机制的详细说明见第 3.4.5 节。

系统软件对 PCI 总线进行配置时，首先获得 BAR 寄存器中的初始化信息，之后根据处理器系统的配置，将合理的基址写入相应的 BAR 寄存器中。系统软件还可以使用该寄存器，获得 PCI 设备使用的 BAR 空间的长度，其方法是向 BAR 寄存器写入 0xFFFF-FFFF，之后再读取该寄存器。Linux 系统使用 __pci_read_base 函数获得 BAR 寄存器的长度，其步骤详见第 14.3.2 节。

处理器访问 PCI 设备的 BAR 空间时，需要使用 BAR 寄存器提供的基址址。值得注意的是，处理器使用存储器域的地址，而 BAR 寄存器存放 PCI 总线域的地址。因此处理器系统并不能直接使用“BAR 寄存器 + 偏移”的方式访问 PCI 设备的寄存器空间，而需要将 PCI 总线域的地址转换为存储器域的地址。在 Linux 系统中，一个处理器系统使用 BAR 空间的正确方式如源代码 2-2 所示。

源代码 2-2 Linux 系统使用 BAR 空间的正确方法

```
pciaddr = pci_resource_start(pdev, 1);

if (!pciaddr) {
    rc = -EIO;
    dev_err(&pdev->dev, "no MMIO resource\n");
```

[⊖] 一般来说 PCI 设备使用 E²PROM 保存 BAR 寄存器的初始值。

```

        goto err_out_res;
    }

    ...

regs = ioremap( pciaddr, CP_REGS_SIZE );

```

在 Linux 系统中，使用 `pci_dev->resource[bar]. start` 参数保存 BAR 寄存器在存储器域的地址。在编写 Linux 设备驱动程序时，必须使用 `pci_resource_start` 函数获得 BAR 空间对应的存储器域的物理地址，而不能使用从 BAR 寄存器中读出的地址。

当驱动程序获得 BAR 空间在存储器域的物理地址后，再使用 `ioremap` 函数将这个物理地址转换为虚拟地址。Linux 系统直接使用 BAR 空间的方法是不正确的，如源代码 2-3 所示。

源代码 2-3 Linux 系统使用 BAR 空间的错误方法

```

ret = pci_read_config_dword( pdev, 1, &pciaddr );

if ( ! pciaddr ) {
    rc = -EIO;
    dev_err( &pdev -> dev, "no MMIO resource\n" );
    goto err_out_res;
}

...

regs = ioremap( pciaddr, BAR_SIZE );

```

在 Linux 系统中，使用 `pci_read_config_dword` 函数获得的是 PCI 总线域的物理地址，在许多处理器系统中，如 Alpha 和 PowerPC 处理器系统，PCI 总线域的物理地址与存储器域的物理地址并不相等。

如果 x86 处理器系统使能了 IOMMU 后，这两个地址也并不一定相等，因此处理器系统直接使用这个 PCI 总线域的物理地址，并不能确保访问 PCI 设备的 BAR 空间的正确性。除此之外在 Linux 系统中，`ioremap` 函数的输入参数为存储器域的物理地址，而不能使用 PCI 总线域的物理地址。

而在 `pci_dev->resource[bar]. start` 参数中保存的地址已经经过 PCI 总线域到存储器域的地址转换，因此在编写 Linux 系统的设备驱动程序时，需要使用 `pci_dev->resource[bar]. start` 参数中的物理地址，再用 `ioremap` 函数将物理地址转换为“存储器域”的虚拟地址。

(11) Command 寄存器

该寄存器为 PCI 设备的命令寄存器，在初始化时，其值为 0，此时这个 PCI 设备除了能够接收配置请求总线事务之外，不能接收任何存储器或者 I/O 请求。系统软件需要合理设置该寄存器之后，才能访问该设备的存储器或者 I/O 空间。在 Linux 系统中，设备驱动程序调用 `pci_enable_device` 函数[⊖]，使能该寄存器的 I/O 和 Memory Space 位之后，才能访问该设备的存储器或者 I/O 地址空间。Command 寄存器的各位的含义如表 2-4 所示。

[⊖] `pci_enable_device` 函数的详细说明第 12.3.2 节。

表 2-4 Command 寄存器

位	描述
0	I/O Space 位，该位表示 PCI 设备是否响应 I/O 请求，为 1 时响应，为 0 时不响应。如果 PCI 设备支持 I/O 地址空间，系统软件会将该位置 1。复位值为 0
1	Memory Space 位，该位表示 PCI 设备是否响应存储器请求，为 1 时响应，为 0 时不响应。如果 PCI 设备支持存储器地址空间，系统软件会将该位置 1。复位值为 0
2	Bus Master 位。该位表示 PCI 设备是否可以作为主设备，为 1 时 PCI 设备可以作为主设备，为 0 时不能。复位值为 0
3	Special Cycle 位，该位表示 PCI 设备是否响应 Special 总线事务，为 1 时响应，为 0 时不响应。PCI 设备可以使用 Special 总线事务，将一些信息广播发送到多个目标设备，Special 总线事务不能穿越 PCI 桥。如果一个 PCI 设备需要将 Special 总线事务发送到 PCI 桥之下的总线时，必须使用 Type 01h 配置周期。PCI 桥可以将 Type 01h 配置周期转换为 Special 周期。该位的复位值为 0
4	Memory Write and Invalidate 位，该位表示 PCI 设备是否支持 Memory Write and Invalidate 总线事务，为 1 时支持，为 0 时不支持。许多低端的 PCI 设备不支持这种总线事务。该位对 PCIe 设备无意义
5	VGA Palette Snoop 位。该位为 1 时支持 Palette Snoop 功能，为 0 时不支持
6	Parity Error Response 位，复位值为 0。该位为 1，而且 PCI 设备在传送过程中出现奇偶校验错误时，PCI 设备将 PERR#信号设置为 1；该位为 0 时，即便出现奇偶检验错误，PCI 设备也仅会将 Status 寄存器的“Detected Parity Error”位置 1
8	SERR# Enable 位，复位值为 0。该位为 1，而且 PCI 设备出现错误时，将使用 SERR#信号，将这个错误信息发送给 HOST 主桥，为 0 时，不能使用 SERR#信号
9	Fast Back-to-Back 位。该位为 1 时，PCI 设备使用 Fast Back-to-Back（快速背靠背）总线周期，这种周期是一种提高传送效率的方法。但并不是所有的 PCI 设备都支持 Fast Back-to-Back 传送周期。该位的复位值为 0
10	Interrupt Disable 位，复位值为 0。该位为 1 时，PCI 设备不能通过 INTx 信号向 HOST 主桥提交中断请求，为 0 时可以使用 INTx 信号提交中断请求。当 PCI 设备使用 MSI 中断方式提交中断请求时，该位将被置为 1

(12) Status 寄存器

该寄存器的绝大多数位都是只读位，保存 PCI 设备的状态，其含义如表 2-5 所示。

表 2-5 Status 寄存器

位	描述
3	Interrupt Status 位，该位只读。该位为 1 且 Command 寄存器的 Interrupt Disable 位为 0 时，表示 PCI 设备已经使用 INTx 信号向处理器提交了中断请求。在多数 PCI 设备中的 BAR 空间，存在自定义的中断状态寄存器，因此设备驱动程序很少使用该位判断 PCI 设备是否提交了中断请求
4	Capabilities List 位，该位只读。该位为 1 时 Capability Pointer 寄存器中的值有效。本书在第 4.3 节详细介绍 PCI 设备的 Capability Pointer 寄存器
5	66MHz Capability 位，该位只读。为 1 时表示此设备支持 66 MHz 的 PCI 总线
7	Fast Back-to-Back Capable 位。该位只读，该位为 1 表示此设备支持快速背靠背总线周期
8	Master Data Parity Error 位。PCI 总线的 PERR#信号有效时将置该位为 1；当 PCI 总线出现数据传送错误时置此位为 1；当 Command 寄存器的 Parity Error Response 位为 1 时，此位为 1
9 ~ 10	DEVSEL timing 字段。该字段为 0b00 时表示 PCI 设备为快速设备；为 0b01 时表示 PCI 设备为中速设备；为 0b10 时表示 PCI 设备为慢速设备。快速设备要求 PCI 总线主设备置 FRAME#信号有效的一个时钟周期后，置 DEVSEL#信号有效；中速设备要求 PCI 总线主设备置 FRAME#信号有效的两个时钟周期后，置 DEVSEL#信号有效；慢速设备要求 PCI 总线主设备置 FRAME#信号有效的三个时钟周期后，置 DEVSEL#信号有效

(续)

位	描述
9 ~ 10	在一条 PCI 总线上，如果快速设备、中速设备和慢速设备都没有使用 DEVSEL#信号响应当前总线事务时，这条总线上的负向译码设备，将被动地接收这个总线事务。如果在这条总线上没有负向译码设备，主设备在 FRAME#信号有效后的第 4 个时钟周期，使用主设备夭折时序，结束当前总线事务。有关负向译码设备的详细说明见第 3.2.1 节。 值得注意的是，在 PCI-X 总线中，该字段的含义与 PCI 总线有所不同
11	Signaled Target Abort 位。该位由 PCI 目标设备设置，当目标设备使用目标设备夭折（Target Abort）时序，结束当前总线周期时，PCI 设备将置该位为 1
12	Received Target Abort 位。该位由 PCI 主设备设置，当发生目标设备夭折时序时，该位被置为 1
13	Received Master Abort 位。该位由 PCI 主设备设置，当发生主设备夭折时序，该位被置为 1。当以上几个 Abort 位有效时，表示 PCI 总线的数据传送通路出现了较为严重的问题
14	Signaled System Error 位。当设备置 SERR#信号有效时，该位被置 1
15	Detected Parity Error 位。当设备发现奇偶校验错时，该位被置 1

(13) Latency Timer 寄存器

在 PCI 总线中，多个设备共享同一条总线带宽。该寄存器用来控制 PCI 设备占用 PCI 总线的时间，当 PCI 设备获得总线使用权，并使能 Frame#信号后，Latency Timer 寄存器将递减，当该寄存器归零后，该设备将使用超时机制停止^①对当前总线的使用。

如果当前总线事务为 Memory Write and Invalidate 时，需要保证对一个完整 Cache 行的操作结束后才能停止当前总线事务。对于多数 PCI 设备而言，该寄存器的值为 32 或者 64，以保证一次突发传送的基本单位为一个 Cache 行。

PCIe 设备不需要使用该寄存器，该寄存器的值必须为 0。因为 PCIe 总线的仲裁方法与 PCI 总线不同，使用的连接方法也与 PCI 总线不同。

2.3.3 PCI 桥的配置空间

PCI 桥使用的配置空间的寄存器如图 2-10 所示。PCI 桥作为一个 PCI 设备，使用的许多配置寄存器与 PCI Agent 的寄存器是类似的，如 Device ID、Vendor ID、Status、Command、Interrupt Pin、Interrupt Line 寄存器等，本节不再重复介绍这些寄存器。下面将重点介绍在 PCI 桥中与 PCI Agent 的配置空间不相同的寄存器。

与 PCI Agent 设备不同，在 PCI 桥中只含有两组 BAR 寄存器，即 Base Address Register 0 ~ 1 寄存器。这两组寄存器与 PCI Agent 设备配置空间的对应寄存器的含义一致。但是在 PCI 桥中，这两组寄存器是可选的。如果在 PCI 桥中不存在私有寄存器，那么可以不使用这两组寄存器设置 BAR 空间。

在大多数 PCI 桥中都不存在私有寄存器，操作系统也不需要为 PCI 桥提供专门的驱动程序，这也是这类桥被称为透明桥的原因。如果在 PCI 桥中不存在私有空间时，PCI 桥将这两个 BAR 寄存器初始化为 0。在 PCI 桥的配置空间中使用两个 BAR 寄存器的原因是这两个 32 位的寄存器可以组成一个 64 位地址空间。

在 PCI 桥的配置空间中，有许多寄存器是 PCI 桥所特有的。PCI 桥除了作为 PCI 设备之

^① 此时 GNT#信号为无效。为提高仲裁效率，PCI 设备在进行数据传送时，GNT#信号可能已经无效。

外，还需要管理其下连接的 PCI 总线子树使用的各类资源，即 Secondary Bus 所连接 PCI 总线子树使用的资源。这些资源包括存储器、I/O 地址空间和总线号。

31	24 23	16 15	8 7	0			
Device ID	Vendor ID			0x00			
Status	Command			0x04			
Class Code		Revision ID		0x08			
BIST	Header Type	Primary Latency Timer	Cache Line Size	0x0C			
Base Address Register0							
Base Address Register1							
Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number	0x10			
Secondary Status	I/O Limit		I/O Base	0x14			
Memory Limit	Memory Base			0x18			
Prefetchable Memory Limit	Prefetchable Memory Base			0x1C			
Prefetchable Base Upper 32 Bits							
Prefetchable Limit Upper 32 Bits							
I/O Limit Upper 16 bits	I/O Base Upper 16 bits			0x20			
Reserved	Capabilities Pointer			0x24			
Expansion ROM Base Address							
Bridge Control	Interrupt Pin	Interrupt Line		0x28			
				0x2C			
				0x30			
				0x34			
				0x38V			
				0x3C			

图 2-10 PCI 桥的配置空间

在 PCI 桥中，与 Secondary bus 相关的寄存器包括两大类。一类寄存器管理 Secondary Bus 之下 PCI 子树的总线号，如 Secondary 和 Subordinate Bus Number 寄存器；另一类寄存器管理下游 PCI 总线的 I/O 和存储器地址空间，如 I/O 和 Memory Limit、I/O 和 Memory Base 寄存器。在 PCI 桥中还使用 Primary Bus 寄存器保存上游的 PCI 总线号。

其中存储器地址空间还分为可预读空间和不可预读空间，Prefetchable Memory Limit 和 Prefetchable Memory Base 寄存器管理可预读空间，而 Memory Limit、Memory Base 管理不可预读空间。在 PCI 体系结构中，除了 ROM 地址空间之外，PCI 设备使用的地址空间大多都是不可预读的。

(1) Subordinate Bus Number、Secondary Bus Number 和 Primary Bus Number 寄存器

PCI 桥可以管理其下的 PCI 总线子树。其中 Subordinate Bus Number 寄存器存放当前 PCI 子树中，编号最大的 PCI 总线号。而 Secondary Bus Number 寄存器存放当前 PCI 桥 Secondary Bus 使用的总线号，这个 PCI 总线号也是该 PCI 桥管理的 PCI 子树中编号最小的 PCI 总线号。因此一个 PCI 桥能够管理的 PCI 总线号在 Secondary Bus Number ~ Subordinate Bus Number 之间。这两个寄存器的值由系统软件遍历 PCI 总线树时设置。

Primary Bus Number 寄存器存放该 PCI 桥上游的 PCI 总线号，该寄存器可读写。Primary Bus Number、Subordinate Bus Number 和 Secondary Bus Number 寄存器在初始化时必须为 0，系统软件将根据这几个寄存器是否为 0，判断 PCI 桥是否被配置过。

不同的操作系统使用不同的 Bootloader 引导，有的 Bootloader 可能会对 PCI 总线树进行遍历，此时操作系统不必重新遍历 PCI 总线树。在 x86 处理器系统中，BIOS 会遍历处理器系统中的所有 PCI 总线树，操作系统可以直接使用 BIOS 的结果，也可以重新遍历 PCI 总线树。而 PowerPC 处理器系统中的 Bootloader，如 U - Boot 并没有完全遍历 PCI 总线树，此时操作系统必须重新遍历 PCI 总线树。本书将在第 14 章以 Linux 系统为例说明 PCI 总线树的遍

历过程。

(2) Secondary Status 寄存器

该寄存器的含义与 PCI Agent 配置空间的 Status 寄存器的含义相近，PCI 桥的 Secondary Status 寄存器记录 Secondary Bus 的状态，而不是 PCI 桥作为 PCI 设备时使用的状态。在 PCI 桥配置空间中还存在一个 Status 寄存器，该寄存器保存 PCI 桥作为 PCI 设备时的状态。

(3) Secondary Latency Timer 寄存器

该寄存器的含义与 PCI Agent 配置空间的 Latency Timer 寄存器的含义相近，PCI 桥的 Secondary Latency Timer 寄存器管理 Secondary Bus 的超时机制，即 PCI 桥发向下游的总线事务；在 PCI 桥配置空间中还存在一个 Latency Timer 寄存器，该寄存器管理 PCI 桥发向上游的总线事务。

(4) I/O Limit 和 I/O Base 寄存器

在 PCI 桥管理的 PCI 子树中包含许多 PCI 设备，而这些 PCI 设备可能会使用 I/O 地址空间。PCI 桥使用这两个寄存器，存放 PCI 子树中所有设备使用的 I/O 地址空间集合的基地址和大小。

(5) Memory Limit 和 Memory Base 寄存器

在 PCI 桥管理的 PCI 子树中有许多 PCI 设备，这些 PCI 设备可能会使用存储器地址空间。这两个寄存器存放所有这些 PCI 设备使用的存储器地址空间集合的基地址和大小，PCI 桥规定这个空间的大小至少为 1MB。

(6) Prefetchable Memory Limit 和 Prefetchable Memory Base 寄存器

在 PCI 桥管理的 PCI 子树中有许多 PCI 设备，如果这些 PCI 设备支持预读，则需要从 PCI 桥的可预读空间中获取地址空间。PCI 桥的这两个寄存器存放这些 PCI 设备使用的可预取存储器空间的基地址和大小。

如果 PCI 桥不支持预读，则其下支持预读的 PCI 设备需要从 Memory Base 寄存器为基地址的存储器空间中获取地址空间。如果 PCI 桥支持预读，其下的 PCI 设备需要根据情况，决定使用可预读空间还是不可预读空间。PCI 总线建议 PCI 设备支持预读，但是支持预读的 PCI 设备并不多见。

(7) I/O Base Upper 16 Bits and I/O Limit Upper 16 寄存器

如果 PCI 桥仅支持 16 位的 I/O 端口，这组寄存器只读，且其值为 0。如果 PCI 桥支持 32 位 I/O 端口，这组寄存器可以提供 I/O 端口的高 16 位地址。

(8) Bridge Control Register

该寄存器用来管理 PCI 桥的 Secondary Bus，其主要位的描述如下。

- Secondary Bus Reset 位，第 6 位，可读写。当该位为 1 时，将使用下游总线提供的 RST#信号复位与 PCI 桥的下游总线连接的 PCI 设备。通常情况下与 PCI 桥下游总线连接的 PCI 设备，其复位信号需要与 PCI 桥提供的 RST#信号连接，而不能与 HOST 主桥提供的 RST#信号连接。
- Primary Discard Timer 位，第 8 位，可读写。PCI 桥支持 Delayed 传送方式，当 PCI 桥的 Primary 总线上的主设备使用 Delayed 方式进行数据传递时，PCI 桥使用 Retry 周期结束 Primary 总线的 Non-Posted 数据请求，并将这个 Non-Posted 数据请求转换为 Delayed 数据请求，之后主设备需要择时重试相同的 Non-Posted 数据请求。当该位为 1

时，表示在 Primary Bus 上的主设备需要在 2^{10} 个时钟周期之内重试这个数据请求，为 0 时，表示主设备需要在 2^{15} 个时钟周期之内重试这个数据请求，否则 PCI 桥将丢弃 Delayed 数据请求。

- Secondary Discard Timer 位，第 9 位，可读写。当该位为 1 时，表示在 Secondary Bus 上的主设备需要在 2^{10} 个时钟周期之内重试这个数据请求，为 0 时，表示主设备需要在 2^{15} 个时钟周期之内重试这个数据请求，如果主设备在规定的时间内没有进行重试时，PCI 桥将丢弃 Delayed 数据请求。

2.4 PCI 总线的配置

PCI 总线定义了两类配置请求，一类是 Type 00h 配置请求，另一类是 Type 01h 配置请求。PCI 总线使用这些配置请求访问 PCI 总线树上的设备配置空间，包括 PCI 桥和 PCI Agent 设备的配置空间。

其中 HOST 主桥或者 PCI 桥使用 Type 00h 配置请求，访问与 HOST 主桥或者 PCI 桥直接相连的 PCI Agent 设备或者 PCI 桥^①；而 HOST 主桥或者 PCI 桥使用 Type 01h 配置请求，需要至少穿越一个 PCI 桥，访问没有与其直接相连的 PCI Agent 设备或者 PCI 桥。如图 2-8 所示，HOST 主桥可以使用 Type 00h 配置请求访问 PCI 设备 01，而使用 Type 01h 配置请求通过 PCI 桥 1、2 或者 3 转换为 Type 00h 配置请求之后，访问 PCI 总线树上的 PCI 设备 11、21、22、31 和 32^②。

当 x86 处理器对 CONFIG_DATA 寄存器进行读写操作时，HOST 主桥将决定向 PCI 总线发送 Type 00h 配置请求还是 Type 01h 配置请求。在 PCI 总线事务的地址周期中，这两种配置请求总线事务的不同反映在 PCI 总线的 AD [31:0] 信号线上。

值得注意的是，PCIe 总线还可以使用 ECAM (Enhanced Configuration Access Mechanism) 机制访问 PCIe 设备的扩展配置空间，使用这种方式可以访问 PCIe 设备 256 B ~ 4 KB 之间的扩展配置空间。但是本节仅介绍如何使用 CONFIG_ADDRESS 和 CONFIG_FATA 寄存器产生 Type 00h 和 Type 01h 配置请求。有关 ECAM 机制的详细说明见第 5.3.2 节。

处理器首先将目标 PCI 设备的 ID 号保存在 CONFIG_ADDRESS 寄存器中，之后 HOST 主桥根据该寄存器的 Bus Number 字段，决定是产生 Type 00h 配置请求，还是 Type 01h 配置请求。当 Bus Number 字段为 0 时，将产生 Type 00h 配置请求，因为与 HOST 主桥直接相连的总线号为 0；大于 0 时，将产生 Type 01h 配置请求。

2.4.1 Type 01h 和 Type 00h 配置请求

本节首先介绍 Type 01h 配置请求，并从 PCI 总线使用的信号线的角度上，讲述 HOST 主桥如何生成 Type 01 配置请求。在 PCI 总线中，只有 PCI 桥能够接收 Type 01h 配置请求。Type 01h 配置请求不能直接发向最终的 PCI Agent 设备，而只能由 PCI 桥将其转换为 Type 01h 继续发向其他 PCI 桥，或者转换为 Type 00h 配置请求发向 PCI Agent 设备。PCI 桥还可

① 此时 PCI 桥作为一个 PCI 设备，接收访问其配置空间的读写请求。

② 最终 Type 01h 配置请求将会被转换为 Type 00h 配置请求，然后访问 PCI Agent 设备。

以将 Type 01h 配置请求转换为 Special Cycle 总线事务（HOST 主桥也可以实现该功能），本节对这种情况不做介绍。

在地址周期中，HOST 主桥使用配置读写总线事务，将 CONFIG_ADDRESS 寄存器的内容复制到 PCI 总线的 AD [31:0] 信号线上。CONFIG_ADDRESS 寄存器与 Type 01h 配置请求的对应关系如图 2-11 所示。

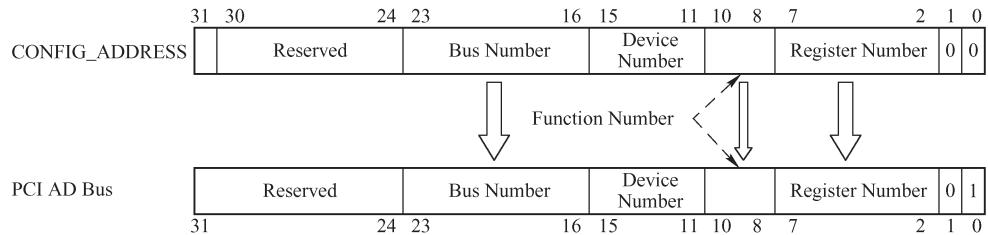


图 2-11 CONFIG_ADDRESS 寄存器与 Type 01h 配置请求的对应关系

从图 2-11 中可以发现，CONFIG_ADDRESS 寄存器的内容基本上是原封不动地复制到 PCI 总线的 AD [31:0] 信号线上的^①。其中 CONFIG_ADDRESS 的 Enable 位不被复制，而 AD 总线的第 0 位必须为 1，表示当前配置请求是 Type 01h。

当 PCI 总线接收到 Type 01 配置请求时，将寻找合适的 PCI 桥^②接收这个配置信息。如果这个配置请求是直接发向 PCI 桥下的 PCI 设备时，PCI 桥将接收这个 Type 01 配置请求，并将其转换为 Type 00h 配置请求；否则 PCI 桥将当前 Type 01h 配置请求原封不动地传递给下一级 PCI 总线。

如果 HOST 主桥或者 PCI 桥发起的是 Type 00h 配置请求，CONFIG_ADDRESS 寄存器与 AD [31:0] 的转换如图 2-12 所示。

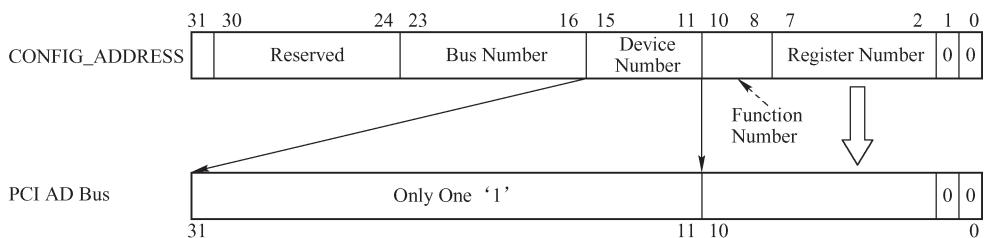


图 2-12 CONFIG_ADDRESS 寄存器与 Type 00h 配置请求的对应关系

此时处理器对 CONFIG_DATA 寄存器进行读写时，处理器将 CONFIG_ADDRESS 寄存器中的 Function Number 和 Register Number 字段复制到 PCI 的 AD 总线的第 10~2 位；将 AD 总线的第 1~0 位赋值为 0b00。PCI 总线在配置请求总线事务的地址周期根据 AD [1:0] 判断当前配置请求是 Type 00h 还是 Type 01h，如果 AD [1:0] 等于 0b00 表示是 Type 00h 配置请求，如果 AD [1:0] 等于 0b01 表示是 Type 01h 配置请求。

而 AD [31:11] 与 CONFIG_ADDRESS 的 Device Number 字段有关，在 Type 00h 配置请

^① Type 01h 配置头信息存在于 PCI 总线事务的地址周期中。

^② PCI 桥根据 Subordinate Bus Number 和 Secondary Bus Number 寄存器，决定是否接收当前配置请求。

求的地址周期中，AD [31:11] 位有且只有一位为 1，其中 AD [31:11] 的每一位选通一个 PCI 设备的配置空间。如第 1.2.2 节所述，PCI 设备配置空间的片选信号是 IDSEL，因此 AD [31:11] 将与 PCI 设备的 IDSEL 信号对应相连。

当以下两种请求之一满足时，HOST 主桥或者 PCI 桥将生成 Type 00h 配置头，并将其发送到指定的 PCI 总线上。

(1) CONFIG_ADDRESS 寄存器的 Bus Number 字段为 0 时，处理器访问 CONFIG_DATA 寄存器时，HOST 主桥将直接向 PCI 总线 0 发出 Type 00h 配置请求。因为与 HOST 主桥直接相连的 PCI 总线号为 0，此时表示 HOST 主桥需要访问与其直接相连的 PCI 设备。

(2) 当 PCI 桥收到 Type 01h 配置头时，将检查 Type 01 配置头的 Bus Number 字段，如果这个 Bus Number 与 PCI 桥的 Secondary Bus Number 相同，则将这个 Type 01 配置头转换为 Type 00h 配置头，并发送到该 PCI 桥的 Secondary 总线上。

2.4.2 PCI 总线配置请求的转换原则

当 CONFIG_ADDRESS 寄存器的 Enable 位为 1，系统软件访问 CONFIG_DATA 寄存器时，HOST 主桥可以产生两类 PCI 总线配置读写总线事务，分别为 Type 00h 和 Type 01h 配置读写总线事务。在配置读写总线事务的地址周期和数据周期中，CONFIG_ADDRESS 和 CONFIG_DATA 寄存器中的数据将被放置到 PCI 总线的 AD 总线上。其中 Type 00h 和 Type 01h 配置读写总线事务映射到 AD 总线的数据并不相同。

其中 Type 00h 配置请求可以直接读取 PCI Agent 设备的配置空间，而 Type 01h 配置请求在通过 PCI 桥时，最终将被转换为 Type 00h 配置请求，并读取 PCI Agent 设备的配置寄存器。本节重点讲述 PCI 桥如何将 Type 01h 配置请求转换为 Type 00h 配置请求。

首先 Type 00h 配置请求不会被转换成 Type 01h 配置请求，因为 Type 00h 配置请求是发向最终 PCI Agent 设备，这些 PCI Agent 设备不会转发这些配置请求。

当 CONFIG_ADDRESS 寄存器的 Bus Number 字段为 0 时，处理器对 CONFIG_DATA 寄存器操作时，HOST 主桥将直接产生 Type 00h 配置请求，挂接在 PCI 总线 0 上的某个设备将通过 ID 译码接收这个 Type 00h 配置请求，并对配置寄存器进行读写操作。如果 PCI 总线上没有设备接收这个 Type 00h 配置请求，将引发 Master Abort，详情见 PCI 总线规范，本节对此不做进一步说明。

如果 CONFIG_ADDRESS 寄存器的 Bus Number 字段为 n ($n \neq 0$)，即访问的 PCI 设备不是直接挂接在 PCI 总线 0 上的，此时 HOST 主桥对 CONFIG_DATA 寄存器操作时，将产生 Type 01h 配置请求，PCI 总线 0 将遍历所有在这条总线上的 PCI 桥，确定由哪个 PCI 桥接收这个 Type 01h 配置请求。

如果 n 大于或等于某个 PCI 桥的 Secondary Bus Number 寄存器，而且小于或等于 Subordinate Bus number 寄存器，那么这个 PCI 桥将接收在当前 PCI 总线上的 Type 01 配置请求，并采用以下规则进行递归处理。

- (1) 开始。
- (2) 遍历当前 PCI 总线的所有 PCI 桥。
- (3) 如果 n 等于某个 PCI 桥的 Secondary Bus Number 寄存器，说明这个 Type 01 配置请求的目标设备直接连接在该 PCI 桥的 Secondary bus 上。此时 PCI 桥将 Type 01 配置请求转

换为 Type 00h 配置请求，并将这个配置请求发送到 PCI 桥的 Secondary Bus 上，Secondary Bus 上的某个设备将响应这个 Type 00h 配置请求，并与 HOST 主桥进行配置信息的交换，转 (5)。

(4) 如果 n 大于 PCI 桥的 Secondary Bus Number 寄存器，而且小于或等于 PCI 桥的 Subordinate Bus number 寄存器，说明这个 Type 01 配置请求的目标设备不与该 PCI 桥的 Secondary Bus 直接相连，但是由这个 PCI 桥下游总线上的某个 PCI 桥管理。此时 PCI 桥将首先认领这个 Type 01 配置请求，并将其转发到 Secondary Bus，转 (2)。

(5) 结束。

下面将举例说明 PCI 总线配置请求的转换原则，并以图 2-8 为例说明处理器如何访问 PCI 设备 01 和 PCI 设备 31 的配置空间。PCI 设备 01 直接与 HOST 主桥相连，因此 HOST 主桥可以使用 Type 00h 配置请求访问该设备。

而 HOST 主桥需要经过多级 PCI 桥才能访问 PCI 设备 31，因此 HOST 主桥需要首先使用 Type 01h 配置请求，之后通过 PCI 桥 1、2 和 3 将 Type 01h 配置请求转换为 Type 00h 配置请求，最终访问 PCI 设备 31。

1. PCI 设备 01

这种情况较易处理，当 HOST 处理器访问 PCI 设备 01 的配置空间时，发现 PCI 设备 01 与 HOST 主桥直接相连，所以将直接使用 Type 00h 配置请求访问该设备的配置空间，具体步骤如下。

首先 HOST 处理器将 CONFIG_ADDRESS 寄存器的 Enabled 位置 1，Bus Number 号置为 0，并对该寄存器的 Device、Function 和 Register Number 字段赋值。当处理器对 CONFIG_DATA 寄存器访问时，HOST 主桥将存放在 CONFIG_ADDRESS 寄存器中的数值，转换为 Type 00h 配置请求，并发送到 PCI 总线 0 上，PCI 设备 01 将接收这个 Type 00h 配置请求，并与处理器进行配置信息交换。

2. PCI 设备 31

HOST 处理器对 PCI 设备 31 进行配置读写时，需要通过 HOST 主桥、PCI 桥 1、2 和 3，最终到达 PCI 设备 31。

当处理器访问 PCI 设备 31 时，首先将 CONFIG_ADDRESS 寄存器的 Enabled 位置 1，Bus Number 字段置为 3，并对 Device、Function 和 Register Number 字段赋值。之后当处理器对 CONFIG_DATA 寄存器进行读写访问时，HOST 主桥、PCI 桥 1、2 和 3 将按照以下步骤进行处理，最后 PCI 设备 31 将接收这个配置请求。

(1) HOST 主桥发现 Bus Number 字段的值为 3，该总线号并不是与 HOST 主桥直接相连的 PCI 总线的 Bus Number，所以 HOST 主桥将处理器对 CONFIG_DATA 寄存器的读写访问直接转换为 Type 01h 配置请求，并将这个配置请求发送到 PCI 总线 0 上。PCI 总线规定 Type 01h 配置请求只能由 PCI 桥负责处理。

(2) 在 PCI 总线 0 上，PCI 桥 1 的 Secondary Bus Number 为 1 而 Subordinate Bus Number 为 3。而 $1 < \text{Bus Number} \leq 3$ ，所以 PCI 桥 1 将接收来自 PCI 总线 0 的 Type 01h 配置请求，并将这个配置请求直接下推到 PCI 总线 1。

(3) 在 PCI 总线 1 上，PCI 桥 2 的 Secondary Bus Number 为 2 而 Subordinate Bus Number 为 3。而 $1 < \text{Bus Number} \leq 3$ ，所以 PCI 桥 2 将接收来自 PCI 总线 0 的 Type 01h 配置请求，并

将这个配置请求直接下推到 PCI 总线 2。

(4) 在 PCI 总线 2 上, PCI 桥 3 的 Secondary Bus Number 为 3, 因此 PCI 桥 3 将“来自 PCI 总线 2 的 Type 01h 配置请求”转换为 Type 00h 配置请求, 并将其下推到 PCI 总线 3。PCI 总线规定, 如果 PCI 桥的 Secondary Bus Number 与 Type 01h 配置请求中包含的 Bus Number 相同时, 该 PCI 桥将接收的 Type 01h 配置请求转换为 Type 00h 配置请求, 然后再发向其 Secondary Bus。

(5) 在 PCI 总线 3 上, 有两个设备: PCI 设备 31 和 PCI 设备 32。在这两个设备中, 必然有一个设备将要响应这个 Type 00h 配置请求, 从而完成整个配置请求周期。在第 2.4.1 节中, 讨论了究竟是 PCI 设备 31 还是 PCI 设备 32 接收这个配置请求, 这个问题涉及 PCI 总线如何分配 PCI 设备使用的设备号。

2.4.3 PCI 总线树 Bus 号的初始化

在一个处理器系统中, 每一个 HOST 主桥都推出一棵 PCI 总线树。在一棵 PCI 总线树中有多少个 PCI 桥 (包括 HOST 主桥), 就含有多少条 PCI 总线。系统软件在遍历当前 PCI 总线树时, 需要首先对这些 PCI 总线进行编号, 即初始化 PCI 桥的 Primary、Secondary 和 Subordinate Bus Number 寄存器。

在一个处理器系统中, 一般将与 HOST 主桥直接相连的 PCI 总线命名为 PCI 总线 0。然后系统软件使用 DFS (Depth First Search) 算法, 依次对其他 PCI 总线进行编号。值得注意的是, 与 HOST 主桥直接相连的 PCI 总线, 其编号都为 0, 因此当处理器系统中存在多个 HOST 主桥时, 将有多个编号为 0 的 PCI 总线, 但是这些编号为 0 的 PCI 总线分属不同的 PCI 总线域, 其含义并不相同。

在一个处理器系统中, 假设 PCI 总线树的结构如图 2-13 所示。当然在一个实际的处理器系统中, 很少会出现这样复杂的 PCI 总线树结构, 本节采用这个结构的目的是便于说明 PCI 总线号的分配过程。

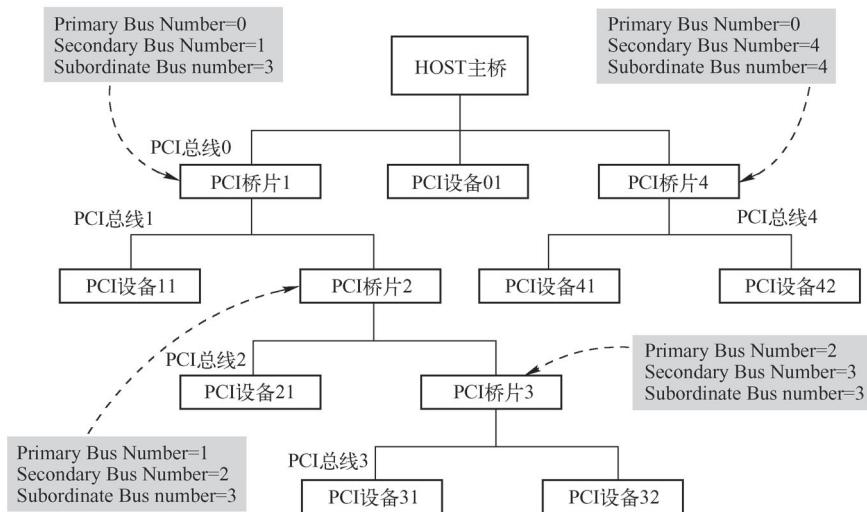


图 2-13 PCI 总线树结构

在 PCI 总线中，系统软件使用深度优先 DFS 算法对 PCI 总线树进行遍历，DFS 算法和广度优先 BFS（Breadth First Search）算法是遍历树型结构的常用算法。与 BFS 算法相比，DFS 算法的空间复杂度较低，因此绝大多数系统在遍历 PCI 总线树时，都使用 DFS 算法而不是 BFS 算法。

DFS 是搜索算法的一种，其实现机制是沿着一棵树的深度遍历各个节点，并尽可能深地搜索树的分支，DFS 的算法为线性时间复杂度，适合对拓扑结构未知的树进行遍历。在一个处理器系统的初始化阶段，PCI 总线树的拓扑结构是未知的，适合使用 DFS 算法进行遍历。下面以图 2-13 为例，说明系统软件如何使用 DFS 算法，分配 PCI 总线号，并初始化 PCI 桥中的 Primary Bus Number、Secondary Bus Number 和 Subordinate Bus number 寄存器。所谓 DFS 算法是指按照深度优先的原则遍历 PCI 胖树，其步骤如下。

(1) HOST 主桥扫描 PCI 总线 0 上的设备。系统软件首先忽略这条总线上的所有 PCI Agent 设备，因为在这些设备之下不会挂接新的 PCI 总线。例如 PCI 设备 01 下不可能挂接新的 PCI 总线。

(2) HOST 主桥首先发现 PCI 桥 1，并将 PCI 桥 1 的 Secondary Bus 命名为 PCI 总线 1。系统软件将初始化 PCI 桥 1 的配置空间，将 PCI 桥 1 的 Primary Bus Number 寄存器赋值为 0，而将 Secondary Bus Number 寄存器赋值为 1，即 PCI 桥 1 的上游 PCI 总线号为 0，而下游 PCI 总线号为 1。

(3) 扫描 PCI 总线 1，发现 PCI 桥 2，并将 PCI 桥 2 的 Secondary Bus 命名为 PCI 总线 2。系统软件将初始化 PCI 桥 2 的配置空间，将 PCI 桥 2 的 Primary Bus Number 寄存器赋值为 1，而将 Secondary Bus Number 寄存器赋值为 2。

(4) 扫描 PCI 总线 2，发现 PCI 桥 3，并将 PCI 桥 3 的 Secondary Bus 命名为 PCI 总线 3。系统软件将初始化 PCI 桥 3 的配置空间，将 PCI 桥 3 的 Primary Bus Number 寄存器赋值为 2，而将 Secondary Bus Number 寄存器赋值为 3。

(5) 扫描 PCI 总线 3，没有发现任何 PCI 桥，这表示 PCI 总线 3 下不可能有新的总线，此时系统软件将 PCI 桥 3 的 Subordinate Bus number 寄存器赋值为 3。系统软件在完成 PCI 总线 3 的扫描后，将回退到 PCI 总线 3 的上一级总线，即 PCI 总线 2，继续进行扫描。

(6) 在重新扫描 PCI 总线 2 时，系统软件发现 PCI 总线 2 上除了 PCI 桥 3 之外没有发现新的 PCI 桥，而 PCI 桥 3 之下的所有设备已经完成了扫描过程，此时系统软件将 PCI 桥 2 的 Subordinate Bus number 寄存器赋值为 3。继续回退到 PCI 总线 1。

(7) PCI 总线 1 上除了 PCI 桥 2 外，没有其他桥片，于是继续回退到 PCI 总线 0，并将 PCI 桥 1 的 Subordinate Bus number 寄存器赋值为 3。

(8) 在 PCI 总线 0 上，系统软件扫描到 PCI 桥 4，则首先将 PCI 桥 4 的 Primary Bus Number 寄存器赋值为 0，而将 Secondary Bus Number 寄存器赋值为 4，即 PCI 桥 1 的上游 PCI 总线号为 0，而下游 PCI 总线号为 4。

(9) 系统软件发现 PCI 总线 4 上没有任何 PCI 桥，将结束对 PCI 总线 4 的扫描，并将 PCI 桥 4 的 Subordinate Bus number 寄存器赋值为 4，之后回退到 PCI 总线 4 的上游总线，即 PCI 总线 0 继续进行扫描。

(10) 系统软件发现在 PCI 总线 0 上的两个桥片 PCI 总线 0 和 PCI 总线 4 都已完成扫描后，将结束对 PCI 总线的 DFS 遍历全过程。

从以上算法可以看出，PCI 桥的 Primary Bus 和 Secondary Bus 号的分配在遍历 PCI 总线树的过程中从上向下分配，而 Subordinate Bus 号是从下向上分配的，因为只有确定了一个 PCI 桥之下究竟有多少条 PCI 总线后，才能初始化该 PCI 桥的 Subordinate Bus 号。

2.4.4 PCI 总线 Device 号的分配

一条 PCI 总线会挂接各种各样的 PCI 设备，而每一个 PCI 设备在 PCI 总线下具有唯一的设备号。系统软件通过总线号和设备号定位一个 PCI 设备之后，才能访问这个 PCI 设备的配置寄存器。值得注意的是，系统软件使用“地址寻址方式”访问 PCI 设备的存储器和 I/O 地址空间，这与访问配置空间使用的“ID 寻址方式”不同。

PCI 设备的 IDSEL 信号与 PCI 总线的 AD [31:0] 信号的连接关系决定了该设备在这条 PCI 总线的设备号。如上文所述，每一个 PCI 设备都使用独立的 IDSEL 信号，该信号将与 PCI 总线的 AD [31:0] 信号连接，IDSEL 信号的含义见第 1.2.2 节。

在此我们简要回顾 PCI 的配置读写事务使用的时序。如图 1-3 所示，PCI 总线事务由一个地址周期加若干个数据周期组成。在进行配置读写请求总线事务时，C/BE#信号线的值在地址周期中为 0x1010 或者为 0x1011，表示当前总线事务为配置读或者配置写请求。此时出现在 AD [31:0] 总线上的值并不是目标设备的 PCI 总线地址，而是目标设备的 ID 号，这与 PCI 总线进行 I/O 或者存储器请求时不同，因为 PCI 总线使用 ID 号而不是 PCI 总线地址对配置空间进行访问。

如图 2-12 所示，在配置读写总线事务的地址周期中，AD [10:0] 信号已经被 Function Number 和 Register Number 使用，因此 PCI 设备的 IDSEL 只能与 AD [31:11] 信号连接。

认真的读者一定可以发现在 CONFIG_ADDRESS 寄存器中 Device Number 字段一共有 5 位可以表示 32 个设备，而 AD [31:11] 只有 21 位，显然在这两者之间无法建立一一对应的映射关系。因此在一条 PCI 总线上如果有 21 个以上的 PCI 设备，那么总是有几个设备无法与 AD [31:11] 信号线连接，从而 PCI 总线无法访问这些设备。因为 PCI 总线在配置请求的地址周期中，只能使用第 31 ~ 11 这些 AD 信号，所以在一条总线上最多也只能挂接 21 个 PCI 设备。这 21 个设备可能是从 0 到 20，也可能是从 11 到 31 排列。从而系统软件在遍历 PCI 总线时，还是需要从 0 到 31 遍历整条 PCI 总线。

在实际的应用中，一条 PCI 总线能够挂接 21 个设备已经足够了，实际上由于 PCI 总线的负载能力有限，即便在总线频率为 33 MHz 的情况下，在一条 PCI 总线中最多也只能挂接 10 个负载，一条 PCI 总线所能挂接的负载详见表 1-1。AD 信号线与 PCI 设备 IDSEL 线的连接关系如图 2-14 所示。

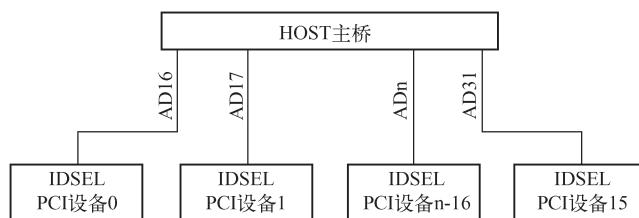


图 2-14 PCI 总线设备号的分配

PCI 总线推荐了一种 Device Number 字段与 AD [31:16] 之间的映射关系。其中 PCI 设备 0 与 Device Number 字段的 0b00000 对应；PCI 设备 1 与 Device Number 字段的 0b00001 对应，并以此类推，PCI 设备 15 与 Device Number 字段的 0b01111 对应。

在这种映射关系之下，一条 PCI 总线上，与信号线 AD16 相连的 PCI 设备的设备号为 0；与信号线 AD17 相连的 PCI 设备的设备号为 1；以此类推，与信号线 AD31 相连的 PCI 设备的设备号为 15。在 Type 00h 配置请求中，设备号并没有像 Function Number 和 Register Number 那样以编码的形式出现在 AD 总线上，而是与 AD 信号一一对应，如图 2-12 所示。

这里有一个原则需要读者注意，就是对 PCI 设备的配置寄存器进行访问时，一定要有确定的 Bus Number、Device Number、Function Number 和 Register Number，这“四元组”缺一不可。在 Type 00h 配置请求中，Device Number 由 AD [31:11] 信号线与 PCI 设备 IDSEL 信号的连接关系确定；Function Number 保存在 AD [10:8] 字段中；而 Register Number 保存在 AD [7:0] 字段中；在 Type 01h 配置请求中，也有完整的四元组信息。

在一个处理器系统的设计中，如果在一条 PCI 总线上使用的 PCI 插槽少于 4 个时，笔者建议优先使用 AD [17:20] 信号与 PCI 设备的 IDSEL 信号连接。因为 PCI-X 总线规范建议使用 AD17 连接 PCI 设备 1、AD18 连接 PCI 设备 2、AD19 连接 PCI 设备 3、AD20 连接 PCI 设备 4，采用这种方法便于实现 PCI 总线与 PCI-X 总线的兼容。

2.5 非透明 PCI 桥

PCI 桥规范定义了透明桥的实现规则，在第 2.3.1 节中详细介绍了这种桥片。通过透明桥，处理器系统可以以 HOST 主桥为根节点，建立一颗 PCI 总线树，在这个树上的 PCI 设备共享同一个 PCI 总线域上的地址空间。

但是在某些场合下 PCI 透明桥并不适用。在图 2-15 所示的处理器系统中存在两个处理器，此时使用 PCI 桥 1 连接处理器 2 并不利于整个处理器系统的配置与管理。假定 PCI 总线使用 32 位地址空间，而处理器 1 和处理器 2 所使用的存储器大小都为 2GB，同时假定处理器 1 和处理器 2 使用的存储器都可以被 PCI 设备访问。

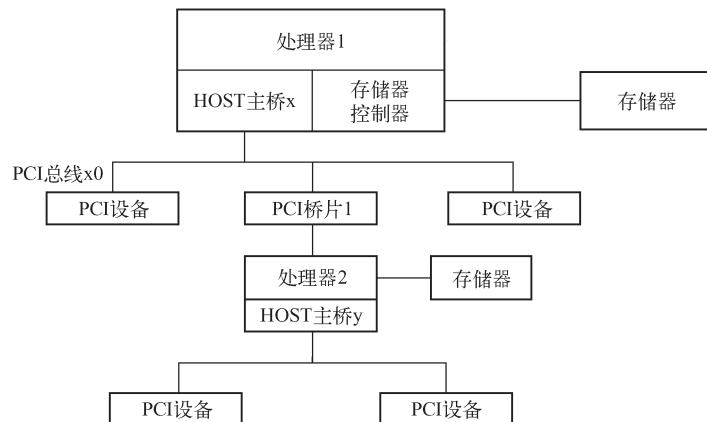


图 2-15 不适合 PCI 透明桥的处理器系统互连方式

此时处理器 1 和 2 使用的存储器空间必须映射到 PCI 总线的地址空间中，而 32 位的 PCI 总线只能提供 4GB 地址空间，此时 PCI 总线 x0 的地址空间将全部被处理器 1 和 2 的存储器空间占用，而没有额外的空间分配给 PCI 设备。

此外有些处理器不能作为 PCI Agent 设备，因此不能直接连接到 PCI 桥上，比如 x86 处理器就无法作为 PCI Agent 设备，因此使用 PCI 透明桥无法将两个 x86 处理器直接相连。如果处理器 2 有两个以上的 PCI 接口，其中一个可以与 PCI 桥 1 相连（此时处理器 2 将作为 PCI Agent 设备），而另一个作为 HOST 主桥 y 连接 PCI 设备。此时 HOST 主桥 y 挂接的 PCI 设备将无法被处理器 1 直接访问。

使用透明桥也不便于解决处理器 1 与处理器 2 间的地址冲突。对于图 2-15 所示的处理器系统，如果处理器 1 和 2 都将各自的存储器映射到 PCI 总线地址空间中，有可能出现地址冲突。虽然 PowerPC 处理器可以使用 Inbound 寄存器，将存储器地址空间映射到不同的 PCI 总线地址空间中，但并非所有的处理器都具有这种映射机制。许多处理器的存储器地址与 PCI 总线地址使用了“简单相等”这种映射方法，如果 PCI 总线连接了两个这样的处理器，将不可避免地出现 PCI 总线地址的映射冲突。

采用非透明桥将有效解决以上这些问题，非透明桥并不是 PCI 总线定义的标准桥片，但是这类桥片在连接两个处理器系统中得到了广泛的应用。一个使用非透明桥连接两个处理器系统的实例如图 2-16 所示。

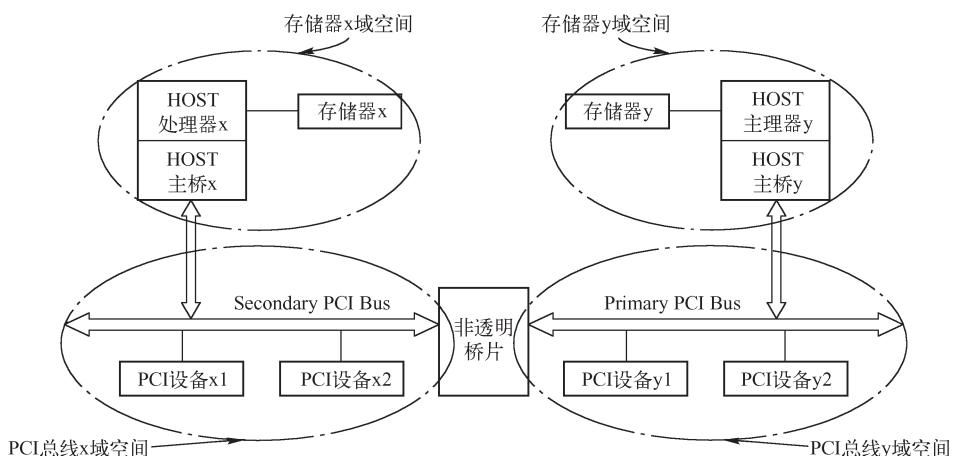


图 2-16 使用 PCI 非透明桥连接两个处理器系统

使用非透明 PCI 桥可以方便地连接两个处理器系统。从图 2-16 中我们可以发现非透明桥可以将 PCI 总线 x 域与 PCI 总线 y 域进行隔离。值得注意的是，非透明 PCI 桥的作用是对不同 PCI 总线域地址空间进行隔离，而不是隔离存储器域地址空间。而 HOST 主桥的作用才是将存储器域与 PCI 总线域进行隔离。

非透明 PCI 桥可以连接两条独立的 PCI 总线，一条被称为 Secondary PCI 总线，另一条被称为 Primary PCI 总线，但是这两条总线没有从属关系，两边是对等的^①。从处理器 x 的角

^① 有些非透明桥，如 DEC21554 的两边并不是完全对等的，尤其是在处理 64 位地址空间时，本文对此不做详细说明。

度来看，与非透明 PCI 桥右边连接的总线叫 Secondary PCI 总线；而从处理器 y 的角度来看，非透明 PCI 桥左边连接的总线叫 Secondary PCI 总线。

HOST 处理器 x 和 PCI 设备可以通过非透明 PCI 桥直接访问 PCI 总线 y 域的地址空间，并通过 HOST 主桥 y 访问存储器 y；HOST 处理器 y 和 PCI 设备也可以通过非透明 PCI 桥，直接访问 PCI 总线 x 域的地址空间，并通过 HOST 主桥 x 访问存储器 x。为此非透明 PCI 桥需要对分属不同 PCI 总线域的地址空间进行转换。

目前有许多厂商可以提供非透明 PCI 桥的芯片，在具体实现上各有差异，但是其基本原理类似，下面以 Intel 21555 为例说明非透明 PCI 桥。值得注意的是，在 PCIe 体系结构中，也存在非透明 PCI 桥的概念。

2.5.1 Intel 21555 中的配置寄存器

Intel 21555 非透明 PCI 桥源于 DEC21554[⊖]，并在此基础上做了一些改动。Intel 21555 桥片与其他透明桥在系统中的位置相同。如图 2-16 所示，这个桥片一边与 Primary PCI 总线相连，另一边与 Secondary PCI 总线相连。

在 Intel 21555 桥片中，包含两个 PCI 设备配置空间，分别是 Primary PCI 总线配置空间和 Secondary PCI 总线配置空间，处理器可以使用 Type 00h 配置请求访问这些配置空间。在大多数情况之下，在 Primary PCI 总线上的 HOST 处理器管理 Primary PCI 配置空间；在 Secondary PCI 总线上的 HOST 处理器管理 Secondary PCI 配置空间[⊖]。

在 Intel 21555 桥片中，还有一组私有寄存器 CSR (Control and Status Register)，系统软件使用这组寄存器对非透明桥进行管理并获得桥片的一些信息，这组寄存器可以被映射成为 PCI 总线的存储器地址空间或者 I/O 地址空间。

本章仅介绍 Primary PCI 总线这一边的配置寄存器，Secondary PCI 总线的配置寄存器虽然与 Primary PCI 总线的这些寄存器略有不同，但是基本对等，因此本节对此不做介绍。Primary PCI 总线的主要寄存器如表 2-6 所示。

表 2-6 Primary PCI 总线的配置寄存器

Offset	寄 存 器	PCI 配置寄存器	复 位 值
0x13 ~ 0x10	Primary CSR and Memory 0 BAR	BAR0	0x0000-0000
0x17 ~ 0x14	Primary CSR I/O BAR	BAR1	0x0000-0001
0x1B ~ 0x18	Downstream Memory 1 BAR	BAR2	0x0000-0000
0x1F ~ 0x1C	Downstream Memory 2 BAR	BAR3	0x0000-0000
0x23 ~ 0x20	Downstream Memory 3 BAR	BAR4	0x0000-0000
0x27 ~ 0x24	Downstream Memory 3 Upper 32 Bits	BAR5	0x0000-0000
0x97 ~ 0x94	Downstream Memory 0 Translated Base	None	不确定
0x9B ~ 0x98	Downstream I/O or Memory 1 Translated Base	None	不确定
0x9F ~ 0x9C	Downstream Memory 2 Translated Base	None	不确定
0xA3 ~ 0xA0	Downstream Memory 3 Translated Base	None	不确定

[⊖] DEC21554 是 Digital 公司的产品。

[⊖] Intel 21555 非透明桥片两边的 HOST 处理器都可以访问 Primary 和 Secondary 总线的配置寄存器。

从表 2-6 中,我们可以发现 Primary PCI 总线的这些配置寄存器共分为两组,一组寄存器与 PCI 设备的配置寄存器的 BAR0 ~ 5 对应,这些寄存器与标准 PCI 配置寄存器 BAR0 ~ 5 的功能相同;另一组寄存器是 Translated Base 寄存器,这组寄存器的主要作用是将来自 Primary PCI 总线的数据访问转换到 Secondary PCI 总线。

其中 BAR0 ~ 5 寄存器在系统初始化时由 Primary PCI 总线上的 HOST 处理器进行配置，配置过程与 PCI 总线上的普通设备完全相同。只是 Intel 21555 规定，BAR0 只能映射为 32 位存储器空间。

CSR 寄存器可以根据需要映射在 BAR0 空间中,此时 BAR0 空间最小为 4 KB。CSR 寄存器也可以根据需要使用 BAR1 寄存器映射为 I/O 地址空间,同时 BAR1 寄存器还可以映射其他 I/O 空间;BAR2 ~ 3 只能映射为 32 位存储器地址空间;而 BAR4 ~ 5 用来映射 64 位的存储器地址空间。

对于 Primary PCI 总线,所有 BAR0 ~ 5 寄存器映射的地址空间都将占用 Primary PCI 总线域,然而这些地址空间中所对应的数据并不在 Primary PCI 总线域中,而是在 Secondary PCI 总线域中。Translated Base 寄存器实现不同 PCI 总线域地址空间的转换,Intel 21555 将不同 PCI 总线域地址空间的转换过程称为“地址翻译”。

Intel 21555 支持两种地址翻译方法,一种是直接地址翻译,另一种是查表翻译。Primary PCI 总线的 BAR 空间只支持直接地址翻译,而 Secondary PCI 总线的 Memory 2 BAR 空间支持查表翻译,本节仅介绍直接地址翻译方法,对查表翻译有兴趣的读者请阅读 Intel 21555 的数据手册^⑨。直接地址翻译过程如图 2-17 所示。

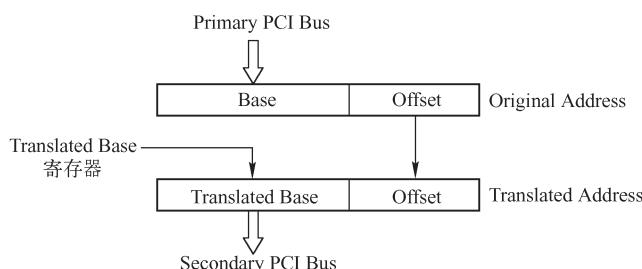


图 2-17 21555 的直接地址翻译过程

当 Primary PCI 总线对非透明桥 21555 的 BAR0 ~ 5 地址空间进行数据请求时,这个数据请求将被转换为对 Secondary PCI 总线的数据请求。Translated Base 寄存器将完成这个地址翻译过程,下节将结合实例说明这个直接地址翻译过程。

2.5.2 通过非透明桥片进行数据传递

下面以图 2-16 中处理器 x 访问处理器 y 存储器地址空间的实例,说明非透明桥 21555 如何将 PCI 总线 x 域与 PCI 总线 y 域联系在一起。

处理器 x 在访问处理器 y 的存储器空间之前，需要做一些必要的准备工作。

- ② 多数半导体厂商提供两类芯片手册，分别是 Datasheet 和 User manual。其中 Datasheet 偏重硬件电气特性，User Manual 偏重芯片使用原理。

(1) 首先确定由哪一个 BAR 寄存器空间映射处理器 y 的存储器地址空间。本节假定使用 BAR2 寄存器映射处理器 y 的存储器地址空间。

(2) BAR2 寄存器使用 Downstream Memory 2 Translated Base 寄存器, 将来自 Primary PCI 总线的访问转换为对 Secondary PCI 总线地址空间的访问。其中 Downstream Memory 2 Translated Base 寄存器可以由处理器 x 或者处理器 y 根据需要进行设置。

假定处理器 x 和 y 的 HOST 主桥使用“直接相等”策略, 建立存储器域与 PCI 总线域间的映射; 而处理器 x 使用 BAR2 地址空间访问处理器 y 存储器空间 $0x1000 - 000 \sim 0x1FFF - FFFF$; 处理器 x 的系统软件事先将 BAR2 寄存器设置完毕。处理器 x 访问处理器 y 的这段存储器空间的步骤如下, 读者可参考图 2-18 理解以下步骤。

(1) 首先处理器 x 访问在处理器 x 域中, 且与非透明桥的 BAR2 空间相对应的存储器地址空间。

(2) HOST 主桥将进行存储器域到 PCI 总线域的转换, 并将这个请求发送到 Primary PCI 总线上。

(3) 非透明桥发现这个数据请求发向 BAR2 地址空间, 则接收这个数据请求, 并在桥片中暂存这个数据请求。

(4) 非透明桥根据 Downstream Memory 2 Translated Base 寄存器的内容, 按照图 2-17 所示的规则进行地址转换。假设 Downstream Memory 2 Translated Base 寄存器的基址被预先设置为 $0x1000 - 0000$, 大小为 256MB(这个物理地址属于处理器 y 的主存储器地址空间)。

(5) 经过非透明桥的转换后, 这个数据请求将穿越非透明桥, 从 Primary PCI 总线域进入 Secondary PCI 总线域, 然后访问处理器 y 的基址为 $0x1000 - 0000$ 的存储器区域。

(6) 处理器 y 的 HOST 主桥接收这个存储器访问请求, 并最终将数据请求发向处理器 y 的存储器中。

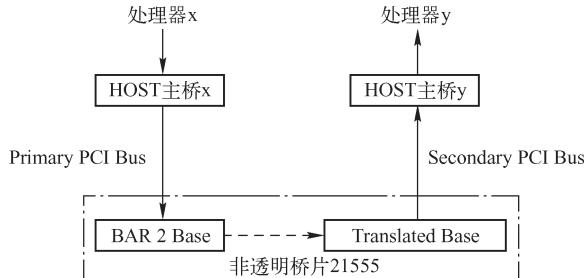


图 2-18 通过非透明桥 21555 进行数据传递

非透明桥 21555 除了可以支持存储器到存储器之间的数据传递, 还支持 PCI 总线域到存储器域, 以及 PCI 总线域之间的数据传递, 此外非透明桥 21555 还可以通过 I²O 和 Doorbell 寄存器进行 Primary PCI 总线与 Secondary PCI 总线之间的中断信号传递。本节对这部分内容不做进一步介绍。

非透明桥有效解决了使用 PCI 总线连接两个处理器存在的问题, 因而得到了广泛的应用。在 PCIe 体系结构中, 也存在非透明 PCI 桥的概念。如在 PLX 的 Switch 芯片中, 各个端口都可以设置为非透明模式。

2.6 小结

本章介绍了在 PCI 总线中使用的桥,包括 HOST 主桥和 PCI 桥,并较详细地介绍了如何使用这些桥访问 PCI 设备的配置空间。

其中 HOST 主桥并不在 PCI 总线规范的约束范围内,不同的处理器可以根据需要设计出不同的 HOST 主桥。本章更加侧重介绍 PowerPC 处理器使用的 HOST 主桥,在该主桥的设计中,提出了许多新的概念,并极大促进了 PCI 总线的发展,在这个桥片中出现的许多新的思想被 PCI V3.0 总线规范采纳。

在 PowerPC 处理器的 HOST 主桥中,明确了存储器域与 PCI 总线域的概念。而区分存储器域与 PCI 总线域也是本章的书写重点,本书将始终强调这两个域的不同。有些处理器系统并没有明确区分这两个域的差别,因此许多读者忽略了 PCI 总线域的存在,并错误地认为 PCI 总线域是存储器域的一部分。

本章还重点介绍了 PCI 桥的实现机制。在许多较简单的处理器系统中,并不包含 PCI 桥,但是读者仍然需要深入理解 PCI 桥这一重要概念。深入理解 PCI 桥的运行机制,是理解 PCI 体系结构的重要基础。

第3章 PCI总线的数据交换

PCI Agent 设备之间以及 HOST 处理器和 PCI Agent 设备之间可以使用存储器读写和 I/O 读写等总线事务进行数据传送。在大多数情况下，PCI 桥不直接与 PCI 设备或者 HOST 主桥进行数据交换，而仅转发来自 PCI Agent 设备或者 HOST 主桥的数据。

PCI Agent 设备间的数据交换并不是本章讨论的重点。本章更侧重讲述 PCI Agent 设备使用 DMA 机制读写主存储器的数据，以及 HOST 处理器如何访问 PCI 设备的 BAR 空间。本章还将使用一定的篇幅讨论在 PCI 总线中与 Cache 相关的总线事务，并在最后介绍预读机制。

3.1 PCI设备BAR空间的初始化

在 PCI Agent 设备进行数据传送之前，系统软件需要初始化 PCI Agent 设备的 BAR0 ~ 5 寄存器和 PCI 桥的 Base、Limit 寄存器。系统软件使用 DFS 算法对 PCI 总线进行遍历时，完成这些寄存器的初始化，即分配这些设备在 PCI 总线域的地址空间。当这些寄存器初始化完毕后，PCI 设备可以使用 PCI 总线地址进行数据传递。

值得注意的是，PCI Agent 设备的 BAR0 ~ 5 寄存器和 PCI 桥的 Base 寄存器保存的地址都是 PCI 总线地址。而这些地址在处理器系统的存储器域中具有映像，如果一个 PCI 设备的 BAR 空间在存储器域中没有映像，处理器将不能访问该 PCI 设备的 BAR 空间。

如上文所述，处理器通过 HOST 主桥将 PCI 总线域与存储器域隔离。当处理器访问 PCI 设备的地址空间时，需要首先访问该设备在存储器域中的地址空间，并通过 HOST 主桥将这个存储器域的地址空间转换为 PCI 总线域的地址空间之后，再使用 PCI 总线事务将数据发送到指定的 PCI 设备中。

PCI 设备访问存储器域的地址空间，即进行 DMA 操作时，也是首先访问该存储器地址空间所对应的 PCI 总线地址空间，之后通过 HOST 主桥将这个 PCI 总线地址空间转换为存储器地址空间，再由 DDR 控制器对存储器进行读写访问。

不同的处理器系统采用不同的机制实现存储器域和 PCI 总线域的转换。如 PowerPC 处理器使用 Outbound 寄存器组实现存储器域到 PCI 总线域间的转换，并使用 Inbound 寄存器组实现 PCI 总线域到存储器域间的转换。

而 x86 处理器没有这种地址空间域的转换机制，因此从 PCI 设备的角度来看，PCI 设备可以直接访问存储器地址；从处理器的角度来看，处理器可以直接访问 PCI 总线地址空间。但是读者需要注意，在 x86 处理器的 HOST 主桥中仍然有存储器域与 PCI 总线域这个概念。只是在 x86 处理器的 HOST 主桥中，存储器域的存储器地址与 PCI 总线地址相等，这种“简单相等”也是一种映射关系。

3.1.1 存储器地址与 PCI 总线地址的转换

下面根据 PowerPC 和 x86 处理器的主桥，抽象出一个虚拟的 HOST 主桥，并以此为例讲述 PCI Agent 设备之间以及 PCI Agent 设备与主存储器间的数据传送过程。

假设在一个 32 位处理器中，存储器域的 0xF000-0000 ~ 0xF7FF-FFFF（共 128MB）这段物理地址空间与 PCI 总线的地址空间存在映射关系。

当处理器访问这段存储器地址空间时，HOST 主桥会认领这个存储器访问，并将该存储器访问使用的物理地址空间转换为 PCI 总线地址空间，并与 0x7000-0000 ~ 0x77FF-FFFF 这段 PCI 总线地址空间对应。

为简化起见，假定在存储器域中只映射了 PCI 设备的存储器地址空间，而不映射 PCI 设备的 I/O 地址空间。而 PCI 设备的 BAR 空间使用 0x7000-0000 ~ 0x77FF-FFFF 这段 PCI 总线域的存储器地址空间。

在这个 HOST 主桥中，存储器域与 PCI 总线域的对应关系如图 3-1 所示。

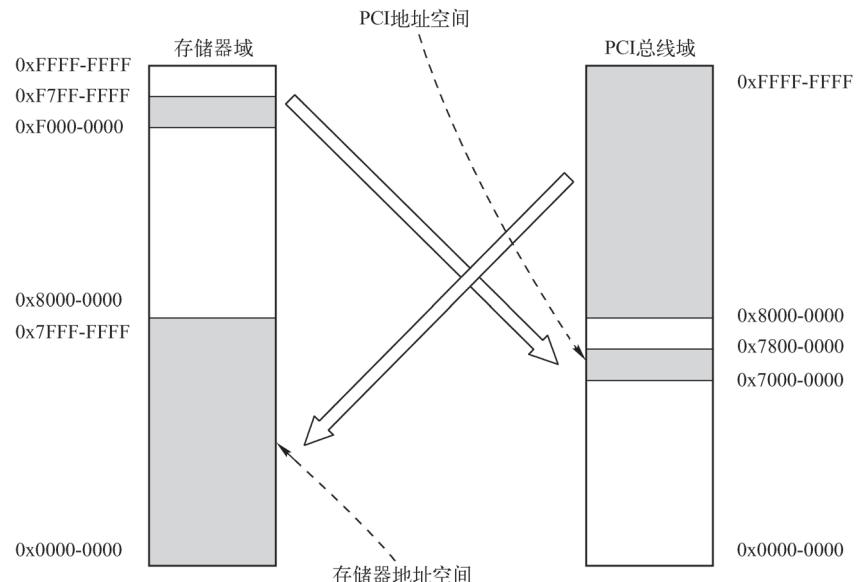


图 3-1 存储器域与 PCI 总线域的映射关系

当 PCI 设备使用 DMA 机制访问存储器域地址空间时，处理器系统同样需要将存储器域的地址空间反向映射到 PCI 总线地址空间。假设在一个处理器系统中，主存储器大小为 2 GB，其在存储器域的地址范围为 0x0000-0000 ~ 0x7FFF-FFFF，而这段地址在 PCI 总线域中对应的“PCI 总线地址空间”为 0x8000-0000 ~ 0xFFFF-FFFF。

PCI 设备进行 DMA 操作时，必须使用 0x8000-0000 ~ 0xFFFF-FFFF 这段 PCI 总线域的地址，HOST 主桥才能认领这个 PCI 总线事务，并将这个总线事务使用的 PCI 总线地址转换为存储器地址，并与 0x0000-0000 ~ 0x7FFF-FFFF 这段存储器区域进行数据传递。

在一个实际的处理器系统中，很少有系统软件采用这样的方法，实现存储器域与 PCI 总线域之间的映射，“简单相等”还是最常用的映射方法。本章采用图 3-1 的映射关系，虽然增加了映射复杂度，却便于读者深入理解存储器域到 PCI 总线域之间的映射关系。下面将以这种映射关系为例，详细讲述 PCI 设备 BAR0 ~ 5 寄存器的初始化。

3.1.2 PCI 设备 BAR 寄存器和 PCI 桥 Base、Limit 寄存器的初始化

PCI 桥的 Base、Limit 寄存器保存“该桥所管理的 PCI 子树”的存储器或者 I/O 空间的

基地址和长度。值得注意的是，PCI 桥也是 PCI 总线上的一个设备，在其配置空间中也有 BAR 寄存器，本节不对 PCI 桥 BAR 寄存器进行说明，因为在多数情况下透明桥并不使用其内部的 BAR 寄存器。下面以图 3-2 所示的处理器系统为例说明上述寄存器的初始化过程，该处理器系统使用的存储器域与 PCI 总线域的映射关系如图 3-1 所示。

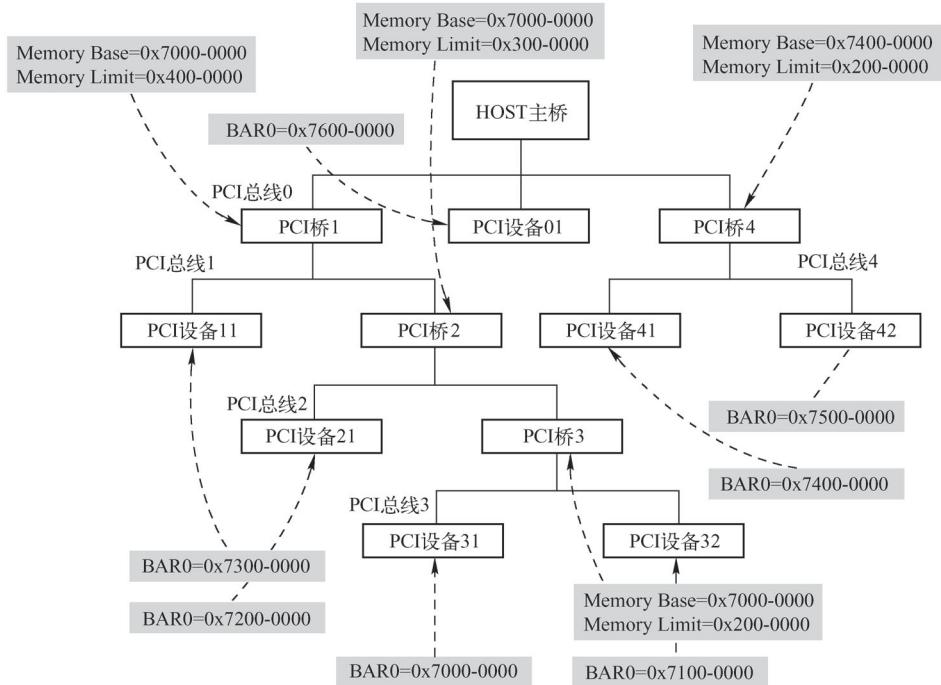


图 3-2 BAR 寄存器的初始化

在 PCI 设备的 BAR 寄存器中，包含该设备使用的 PCI 总线域的地址范围。在 PCI 设备的配置空间中共有 6 个 BAR 寄存器，因此一个 PCI 设备最多可以使用 6 组 32 位的 PCI 总线地址空间，或者 3 组 64 位的 PCI 总线地址空间。这些 BAR 空间可以保存 PCI 总线域的存储器地址空间或者 I/O 地址空间，目前多数 PCI 设备仅使用存储器地址空间。而在通常情况下，一个 PCI 设备使用 2 到 3 个 BAR 寄存器就足够了。

为简化起见，首先假定在图 3-2 中所示的 PCI 总线树中，所有 PCI Agent 设备只使用了 BAR0 寄存器，其申请的数据空间大小为 16MB（即 0x1000000B）而且不可预读，而且 PCI 桥不占用 PCI 总线地址空间，即 PCI 桥不含有 BAR 空间。并且假定当前 HOST 主桥已经完成了对 PCI 总线树的编号。

根据以上假设，该 PCI 总线树的遍历过程如下所示。

(1) 系统软件根据 DFS 算法，首先寻找到第一组 PCI 设备，分别为 PCI 设备 31 和 PCI 设备 32[⊖]，并根据这两个 PCI 设备需要的 PCI 空间大小，从 PCI 总线地址空间中 (0x7000-

[⊖] HOST 主桥下的第一个桥片是 PCI 桥片 1，PCI 桥片 1 下的第一个桥片是 PCI 桥片 2，而 PCI 桥片 2 下的第一个桥片是 PCI 桥片 3，因而第一组 PCI 设备为 PCI 总线 3 下的 PCI 设备。不同的系统软件查找第一组 PCI 设备的方法不同，Linux 认为第一组 PCI 设备为 PCI 总线 0 下的 PCI 设备。

0000 ~ 0x77FF-FFFF) 为这两个 PCI 设备的 BAR0 寄存器分配基地址，分别为 0x7000-0000 和 0x7100-0000。

(2) 当系统软件完成 PCI 总线 3 下所有设备的 BAR 空间的分配后，将初始化 PCI 桥 3 的配置空间。这个桥片的 Memory Base 寄存器保存其下所有 PCI 设备使用的“PCI 总线域地址空间的基址”，而 Memory Limit 寄存器保存其下 PCI 设备使用的“PCI 总线域地址空间的大小”。系统软件将 Memory Base 寄存器赋值为 0x7000-0000，而将 Memory Limit 寄存器赋值为 0x200-0000。

(3) 系统软件回溯到 PCI 总线 2，并找到 PCI 总线 2 上的 PCI 设备 21，并将 PCI 设备 21 的 BAR0 寄存器赋值为 0x7200-0000。

(4) 完成 PCI 总线 2 的遍历后，系统软件初始化 PCI 桥 2 的配置寄存器，将 Memory Base 寄存器赋值为 0x7000-0000，Memory Limit 寄存器赋值为 0x300-0000。

(5) 系统软件回溯到 PCI 总线 1，并找到 PCI 设备 11，并将这个设备的 BAR0 寄存器赋值为 0x7300-0000。并将 PCI 桥 1 的 Memory Base 寄存器赋值为 0x7000-0000，Memory Limit 寄存器赋值为 0x400-0000。

(6) 系统软件回溯到 PCI 总线 0，并在这条总线上发现另外一个 PCI 桥，即 PCI 桥 4。并使用 DFS 算法继续遍历 PCI 桥 4。首先系统软件将遍历 PCI 总线 4，并发现 PCI 设备 41 和 PCI 设备 42，并将这两个 PCI 设备的 BAR0 寄存器分别赋值为 0x7400-0000 和 0x7500-0000。

(7) 系统软件初始化 PCI 桥 4 的配置寄存器，将 Memory Base 寄存器赋值为 0x7400-0000，Memory Limit 寄存器赋值为 0x200-0000。系统软件再次回到 PCI 总线 0，这一次系统软件没有发现新的 PCI 桥，于是将初始化这条总线上的所有 PCI 设备。

(8) PCI 总线 0 上只有一个 PCI 设备，即 PCI 设备 01。系统软件将这个设备的 BAR0 寄存器赋值为 0x7600-0000，并结束整个 DFS 遍历过程。

3.2 PCI 设备的数据传递

PCI 设备的数据传递使用地址译码方式，当一个存储器读写总线事务到达 PCI 总线时，在这条总线上的所有 PCI 设备将进行地址译码，如果当前总线事务使用的地址在某个 PCI 设备的 BAR 空间中时，该 PCI 设备将使能 DEVSEL#信号，认领这个总线事务。

如果 PCI 总线上的所有设备都不能通过地址译码，认领这个总线事务时，这条总线的“负向译码”设备将认领这个总线事务，如果在这条 PCI 总线上没有“负向译码”设备，该总线事务的发起者将使用 Master Abort 总线周期结束当前 PCI 总线事务。

3.2.1 PCI 设备的正向译码与负向译码

如上文所述，PCI 设备使用“地址译码”方式接收存储器读写总线请求。在 PCI 总线中定义了两种“地址译码”方式，一种是正向译码，另一种是负向译码。

下面仍以图 3-2 所示的处理器系统为例，说明数据传送使用的寻址方法。当 HOST 主桥通过存储器或者 I/O 读写总线事务访问其下所有 PCI 设备时，PCI 总线 0 下的所有 PCI 设备都将对出现在地址周期中的 PCI 总线地址进行译码。如果这个地址在某个 PCI 设备的 BAR 空间中命中时，这个 PCI 设备将接收这个 PCI 总线请求。这个过程也被称为 PCI 总线的正向

译码，这种方式也是大多数 PCI 设备所采用的译码方式。

但是在 PCI 总线上的某些设备，如 PCI-to-(E)ISA 桥并不使用正向译码接收来自 PCI 总线的请求，PCI-to-ISA 桥在处理器系统中的位置如图 1-1 所示。PCI 总线 0 上的总线事务在三个时钟周期后，没有得到任何 PCI 设备响应时（即总线请求的 PCI 总线地址不在这些设备的 BAR 空间中），PCI-to-ISA 桥将被动地接收这个数据请求。这个过程被称为 PCI 总线的负向译码。可以进行负向译码的设备也被称为负向译码设备。

在 PCI 总线中，除了 PCI-to-(E)ISA 桥可以作为负向译码设备，PCI 桥也可以作为负向译码设备，但是 PCI 桥并不是在任何时候都可以作为负向译码设备。在绝大多数情况下，PCI 桥无论是处理“来自上游总线”，还是处理“来自下游总线”的总线事务时，都使用正向译码方式，但是在某些特殊应用中，PCI 桥也可以作为负向译码设备。

笔记本在连接 Dock 插座时，也使用了 PCI 桥。因为在大多数情况下，笔记本与 Dock 插座是分离使用的，而且 Dock 插座上连接的设备多为慢速设备，此时用于连接 Dock 插座的 PCI 桥使用负向译码。Dock 插座在笔记本系统中的位置如图 3-3 所示。

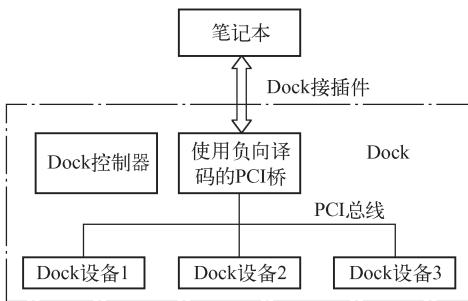


图 3-3 Dock 插座与笔记本的连接关系

当笔记本与 Dock 建立连接之后，如果处理器需要访问 Dock 中的外部设备时，Dock 中的 PCI 桥将首先使用负向译码方式接收 PCI 总线事务，之后将这个 PCI 总线事务转发到 Dock 的 PCI 总线中，再访问相应的 PCI 设备。

在 Dock 中使用负向译码 PCI 桥的优点是，该桥管理的设备并不参与处理器系统对 PCI 总线的枚举过程。当笔记本插入到 Dock 之后，系统软件并不需要重新枚举 Dock 中的设备并为这些设备分配系统资源，而仅需要使用负向译码 PCI 桥管理好其下的设备即可，从而极大降低了 Dock 对系统软件的影响。

当 HOST 处理器访问 Dock 中的设备时，负向译码 PCI 桥将首先接管这些存储器读写总线事务，然后发送到 Dock 设备中。值得注意的是，在许多笔记本的 Dock 实现中，并没有使用负向译码 PCI 桥，而使用 PCI-to-ISA 桥。

PCI 总线规定使用负向译码的 PCI 桥，其 Base Class Code 寄存器为 0x06，Sub Class Code 寄存器为 0x04，而 Interface 寄存器为 0x01；使用正向译码方式的 PCI 桥的 Interface 寄存器为 0x00。系统软件（E²PROM）在初始化 Interface 寄存器时务必注意这个细节。

综上所述，在 PCI 总线中有两种负向译码设备，PCI-to-E(ISA)桥和 PCI 桥。但 PCI 桥并非在任何时候都是负向译码设备，只有 PCI 桥连接 Dock 插座时，PCI 桥的 Primary Bus 才使用负向译码方式。而这个 PCI 桥的 Secondary Bus 在接收 Dock 设备的请求时仍然使用正向译码方式。

PCI 桥使用的正向译码方式与 PCI 设备使用的正向译码方式有所不同。如图 3-4 所示，当一个总线事务是从 PCI 桥的 Primary Bus 到 Secondary Bus 时，PCI 桥使用的正向译码方式与 PCI 设备使用的方式类似。如果该总线事务使用的地址在 PCI 桥任意一个 Memory Base 窗口^①命中时，该 PCI 桥将使用正向译码方式接收该总线事务，并根据实际情况决定是否将这个总线事务转发到 Secondary Bus。

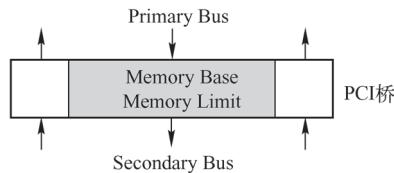


图 3-4 PCI 桥使用的正向译码方式

当一个总线事务是从 PCI 桥的 Secondary Bus 到 Primary Bus 时，如果该总线事务使用的地址没有在 PCI 桥所有的 Memory Base 窗口命中，表明当前总线事务不是访问该 PCI 桥管理的 PCI 子树中的设备，因此 PCI 桥将接收当前总线事务，并根据实际情况决定是否将这个总线事务转发到 Primary Bus。

以图 3-2 为例，当 PCI 设备 11 访问主存储器空间时，首先将存储器读写总线事务发送到 PCI 总线 1 上，而这个存储器地址显然不会在 PCI 总线 1 的任何 PCI 设备的 BAR 空间中，此时 PCI 桥 1 将认领这个 PCI 总线的数据请求，并将这个总线事务转发到 PCI 总线 0 上。最后 HOST 主桥将接收这个总线事务，并将 PCI 总线地址转换为存储器域的地址，与主存储器进行读写操作。

值得注意的是，PCI 总线并没有规定 HOST 主桥使用正向还是负向译码方式接收这个存储器读写总线事务，但是绝大多数 HOST 主桥使用正向译码方式接收来自下游的存储器读写总线事务。在 PowerPC 处理器中，如果当前存储器读写总线事务使用的地址在 Inbound 窗口内，HOST 主桥将接收这个总线事务，并将其转换为存储器域的读写总线事务，与主存储器进行数据交换。

3.2.2 处理器到 PCI 设备的数据传送

下面以图 3-2 所示的处理器系统为例，说明处理器向 PCI 设备 11 进行存储器写的数据传送过程。处理器向 PCI 设备进行读过程与写过程略有区别，因为存储器写使用 Posted 方式，而存储器读使用 Non-Posted 方式，但是存储器读使用的地址译码方式与存储器写类似，因此本节对处理器向 PCI 设备进行存储器读的过程不做进一步介绍。

PCI 设备 11 在 PCI 总线域的地址范围是 0x7300-0000 ~ 0x73FF-FFFF。这段空间在存储器域中对应的地址范围是 0xF300-0000 ~ 0xF3FF-FFFF。下面假设处理器使用存储器写指令，访问 0xF300-0008 这个存储器地址，其步骤如下。

(1) 存储器域将 0xF300-0008 这个地址发向 HOST 主桥，0xF000-0000 ~ 0xF7FF-FFFF 这

^① PCI 桥除了具有 Memory Base 窗口外，还有 I/O Base 窗口和 Prefetchable Memory Base 窗口。

段地址已经由 HOST 主桥映射到 PCI 总线域地址空间，所以 HOST 主桥认为这是一个对 PCI 设备的访问。因此 HOST 主桥将首先接管这个存储器写请求。

(2) HOST 主桥将存储器域的地址 0xF300-0008 转换为 PCI 总线域的地址 0x7300-0008，并通过总线仲裁获得 PCI 总线 0 的使用权，启动 PCI 存储器写周期，并将这个存储器写总线事务发送到 PCI 总线 0 上。值得注意的是，这个存储器读写总线事务使用的地址为 0x7300-0008，而不是 0xF300-0008。

(3) PCI 总线 0 的 PCI 桥 1 发现 0x7300-0008 在自己管理的地址范围内，于是接管这个存储器写请求，并通过总线仲裁逻辑获得 PCI 总线 1 的使用权，并将这个请求转发到 PCI 总线 1 上。

(4) PCI 总线 1 的 PCI 设备 11 发现 0x7300-0008 在自己的 BAR0 寄存器中命中，于是接收这个 PCI 写请求，并完成存储器写总线事务。

3.2.3 PCI 设备的 DMA 操作

下面以图 3-2 所示的处理器系统为例，说明 PCI 设备 11 向存储器进行 DMA 写的数据传送过程。PCI 设备的 DMA 写使用 Posted 方式而 DMA 读使用 Non - Posted 方式。本节不介绍 PCI 设备进行 DMA 读的过程，而将这部分内容留给读者分析。

假定 PCI 设备 11 需要将一组数据发送到 0x1000-0000 ~ 0x1000-FFFF 这段存储器域的地址空间中。由上文所述，存储器域的 0x0000-0000 ~ 0x7FFF-FFFF 这段存储器空间与 PCI 总线域的 0x8000-0000 ~ 0xFFFF-FFFF 这段 PCI 总线地址空间对应。

PCI 设备 11 并不能直接操作 0x1000-0000 ~ 0x1000-FFFF 这段存储器域的地址空间，PCI 设备 11 需要对 PCI 总线域的地址空间 0x9000-0000 ~ 0x9000-FFFF 进行写操作，因为 PCI 总线地址空间 0x9000-0000 ~ 0x9000-FFFF 已经被 HOST 主桥映射到 0x1000-0000 ~ 0x1000-FFFF 这段存储器域。这个 DMA 写具体的操作流程如下。

(1) 首先 PCI 设备 11 通过总线仲裁逻辑获得 PCI 总线 1 的使用权，之后将存储器写总线事务发送到 PCI 总线 1 上。值得注意的是，这个存储器写总线事务的目的地址是 PCI 总线域的地址空间 0x9000-0000 ~ 0x9000-FFFF，这个地址是主存储器在 PCI 总线域的地址映像。

(2) PCI 总线 1 上的设备将进行地址译码，确定这个写请求是不是发送到自己的 BAR 空间，在 PCI 总线 1 上的设备除了 PCI 设备 11 之外，还有 PCI 桥 2 和 PCI 桥 1。

(3) 首先 PCI 桥 1、2 和 PCI 设备 11 对这个地址同时进行正向译码。PCI 桥 1 发现这个 PCI 地址并不在自己管理的 PCI 总线地址范围之内，因为 PCI 桥片 1 所管理的 PCI 总线地址空间为 0x7000-0000 ~ 0x73FF-FFFF。此时 PCI 桥 1 将接收这个存储器写总线事务，因为 PCI 桥 1 所管理的 PCI 总线地址范围并不包含当前存储器写总线事务的地址，所以其下所有 PCI 设备都不可能接收这个存储器写总线事务。

(4) PCI 桥 1 发现自己并不能处理当前这个存储器写总线事务，则将这个存储器写总线事务转发到上游总线。PCI 桥 1 首先通过总线仲裁逻辑获得 PCI 总线 0 的使用权后，然后将这个总线事务转发到 PCI 总线 0。

(5) HOST 主桥发现 0x9000-0000 ~ 0x9000-FFFF 这段 PCI 总线地址空间与存储器域的存储器地址空间 0x1000-0000 ~ 0x1000-FFFF 对应，于是将这段 PCI 总线地址空间转换成为存储器域的存储器地址空间，并完成对这段存储器的写操作。

(6) 存储器控制器将从 HOST 主桥接收数据，并将其写入到主存储器。

PCI 设备间的数据传递与 PCI 设备到存储器的数据传递大体类似。我们以 PCI 设备 11 将数据传递到 PCI 设备 42 为例说明这个传递过程。我们假定 PCI 设备 11 将一组数据发送到 PCI 设备 42 的 PCI 总线地址 0x7500-0000 ~ 0x7500-FFFF 这段地址空间中。这个过程与 PCI 设备 11 将数据发送到存储器的第 1 ~ 5 步基本类似，只是第 5、6 步不同。PCI 设备 11 将数据发送到 PCI 设备 42 的第 5、6 步如下所示。

(5) PCI 总线 0 发现其下的设备 PCI 桥 4 能够处理来自 PCI 总线 0 的数据请求，则 PCI 桥 4 将接管这个 PCI 写请求，并通过总线仲裁逻辑获得 PCI 总线 4 的使用权，之后将这个存储器写请求发向 PCI 总线 4。此时 HOST 主桥不会接收当前存储器写总线事务，因为 0x7500-0000 ~ 0x7500-FFFF 这段地址空间并不是 HOST 主桥管理的地址范围。

(6) PCI 总线 4 的 PCI 设备 42 将接收这个存储器写请求，并完成这个 PCI 存储器写请求总线事务。

PCI 总线树内的数据传送始终都在 PCI 总线域中进行，不存在不同域之间的地址转换，因此 PCI 设备 11 向 PCI 设备 42 进行数据传递时，并不会进行 PCI 总线地址空间到存储器地址空间的转换。

3.2.4 PCI 桥的 Combining、Merging 和 Collapsing

由上所述，PCI 设备间的数据传递有时将通过 PCI 桥。在某些情况下，PCI 桥可以合并一些数据传递，以提高数据传递的效率。PCI 桥可以采用 Combining、Merging 和 Collapsing 三种方式，优化数据通过 PCI 桥的效率。

1. Combining

PCI 桥可以将接收到的多个存储器写总线事务合并为一个突发存储器写总线事务。PCI 桥进行这种 Combining 操作时需要注意数据传送的“顺序”。当 PCI 桥接收到一组物理地址连续的存储器写访问时，如对 PCI 设备的某段空间的 DW1、2 和 4 进行存储器写访问时，PCI 桥可以将这组访问转化为一个对 DW1 ~ 4 的突发存储器写访问，并使用字节使能信号 C/BE[3:0]#进行控制，其过程如下所示。

PCI 桥将在数据周期 1 中，置 C/BE[3:0]#信号为有效表示传递数据 DW1；在数据周期 2 中，置 C/BE[3:0]#信号为有效表示传递数据 DW2；在数据周期 3 中，置 C/BE[3:0]#信号为无效表示在这个周期中所传递的数据无效，从而跳过 DW3；并在数据周期 4 中，置 C/BE[3:0]#信号为有效表示传递数据 DW4。

目标设备将最终按照发送端的顺序，接收 DW1、DW2 和 DW4，采用这种方法在不改变传送序的前提下，提高了数据的传送效率。值得注意的是，有些 HOST 主桥也提供这种 Combining 方式，合并多次数据访问。如果目标设备不支持突发传送方式，该设备可以使用 Disconnect 周期，终止突发传送，此时 PCI 桥/HOST 主桥可以使用多个存储器写总线事务分别传送 DW1、DW2 和 DW4，而不会影响数据传送。

如果 PCI 桥收到“乱序”的存储器写访问，如对 PCI 设备的某段空间的 DW4、3 和 1 进行存储器写访问时，PCI 桥不能将这组访问转化为一个对 DW1 ~ 4 的突发存储器写访问，此时 PCI 桥必须使用三个存储器写总线事务转发这些存储器写访问。

2. Merge

PCI 桥可以将收到的多个对同一个 DW 地址的 Byte、Word 进行的存储器写总线事务，合并为一个对这个 DW 地址的存储器写总线事务。PCI 规范并没有要求这些对 Byte、Word 进行的存储器写在一个 DW 的边界之内，但是建议 PCI 桥仅处理这种情况。本节也仅介绍在这种情况下，PCI 桥的处理过程。

PCI 规范允许 PCI 桥进行 Merge 操作的存储器区域必须是可预读的，而可预读的存储器区域必须支持这种 Merge 操作。Merge 操作可以不考虑访问顺序，可以将对 Byte0、Byte1、Byte3 的存储器访问合并为一个 DW，也可以将对 Byte3、Byte1、Byte0 的存储器访问合并为一个 DW。在这种情况下，PCI 总线事务仅需屏蔽与 Byte2 相关的字节使能信号 C/BE2#即可。

如果 PCI 设备对 Byte1 进行存储器写、然后再对 Byte1、Byte2、Byte3 进行存储器写时，PCI 桥不能将这两组存储器写合并为一次对 DW 进行存储器写操作。但是 PCI 桥可以合并后一组存储器写，即首先对 Byte1 进行存储器写，然后合并后一组存储器写（Byte1、Byte2 和 Byte3）为一个 DW 写，并屏蔽相应的 C/BE0#信号。Combining 与 Merge 操作之间没有直接联系，PCI 桥可以同时支持这两种方式，也可以支持任何一种方式。

3. Collapsing

Collapsing 指 PCI 桥可以将对同一个地址进行的 Byte、Word 和 DW 存储器写总线事务合并为一个存储器写操作。使用 PCI 桥的 Collapsing 方式是，具有某些条件限制，在多数情况下，PCI 桥不能使用 Collapsing 方式合并多个存储器写总线事务。

当 PCI 桥收到一个对“DW 地址 X”的 Byte3 进行的存储器写总线事务，之后又收到一个对“DW 地址 X”的 Byte、Word 或者 DW 存储器写总线事务，而且后一个对 DW 地址 X 进行的存储器写仍然包含 Byte3 时，如果 PCI 桥支持 Collapsing 方式，就可以将这两个存储器写合并为一个存储器写。

PCI 桥在绝大多数情况下不能支持这种方式，因为很少有 PCI 设备支持这种数据合并方式。通常情况下，对 PCI 设备的同一地址的两次写操作代表不同的含义，因此 PCI 桥不能使用 Collapsing 方式将这两次写操作合并。PCI 规范仅是提出了 Collapsing 方式的概念，几乎没有 PCI 桥支持这种数据合并方式。

3.3 与 Cache 相关的 PCI 总线事务

PCI 总线规范定义了一系列与 Cache 相关的总线事务，以提高 PCI 设备与主存储器进行数据交换的效率，即 DMA 读写的效率。当 PCI 设备使用 DMA 方式向存储器进行读写操作时，一定需要经过 HOST 主桥，而 HOST 主桥通过 FSB 总线^①向存储器控制器进行读写操作时，需要进行 Cache 共享一致性操作。

PCI 设备与主存储器进行的 Cache 共享一致性增加了 HOST 主桥的设计复杂度。在高性能处理器中 Cache 状态机的转换模型十分复杂。而 HOST 主桥是 FSB 上的一个设备，需要按照 FSB 规定的协议处理这个 Cache 一致性，而多级 Cache 的一致性和状态转换模型一直是高

^① 在许多处理器中，HOST 主桥与 FSB 之间还存在 SoC 平台总线。

性能处理器设计中的难点。

不同的 HOST 主桥处理 PCI 设备进行的 DMA 操作时，使用的 Cache 一致性的方法并不相同。因为 Cache 一致性操作不仅与 HOST 主桥的设计相关，而且主要与处理器和 Cache Memory 系统设计密切相关。

PowerPC 和 x86 处理器可以对 PCI 设备所访问的存储器进行设置，其设置方法并不相同。其中 PowerPC 处理器，如 MPC8548 处理器，可以使用 Inbound 寄存器的 RTT 字段和 WTT 字段，设置在 PCI 设备进行 DMA 操作时，是否需要进行 Cache 一致性操作，是否可以将数据直接写入 Cache 中。RTT 字段和 WTT 字段的详细说明见第 2.2.3 节。

而 x86 处理器可以使用 MTRR（Memory Type Range Registers）设置物理存储器区间的属性是否为可 Cache 空间。下文分别讨论在 PowerPC 与 x86 处理器中，PCI 设备进行 DMA 写操作时，如何进行 Cache 一致性操作。

但是与 PowerPC 处理器相比，x86 处理器在处理 PCI 设备的 Cache 一致性上略有不足，特别是网络设备与存储器系统进行数据交换的效率。因为 x86 处理器重点优化的是 PCIe 设备，目前 x86 处理器使用的 IOAT（I/O Acceleration Technology）技术，显著提高了 PCIe 设备与主存储器进行数据通信的效率。

3.3.1 Cache 一致性的基本概念

PCI 设备对可 Cache 的存储器空间进行 DMA 读写操作的过程较为复杂，有关 Cache Memory 的话题可以独立成书。而不同的处理器系统使用的 Cache Memory 的层次结构和访问机制有较大的差异，这部分内容也是现代处理器系统设计的重中之重。

本节仅介绍在 Cache Memory 系统中与 PCI 设备进行 DMA 操作相关的一些最为基础的概念。在多数处理器系统中，使用了以下概念描述 Cache 一致性的实现过程。

1. Cache 一致性协议

多数 SMP 处理器系统使用了 MESI 协议处理多个处理器之间的 Cache 一致性。该协议也称为 Illinois protocol，在 SMP 处理器系统中得到了广泛的应用。MESI 协议使用四个状态位描述每一个 Cache 行。

- M (Modified) 位。M 位为 1 时表示当前 Cache 行中包含的数据与存储器中的数据不一致，而且它仅在本 CPU 的 Cache 中有效，不在其他 CPU 的 Cache 中存在副本，在这个 Cache 行的数据是当前处理器系统中最新的数据副本。当 CPU 对这个 Cache 行进行替换操作时，必然会引发系统总线的写周期，将 Cache 行中数据与内存中的数据同步。
- E (Exclusive) 位。E 位为 1 时表示当前 Cache 行中包含的数据有效，而且该数据仅在当前 CPU 的 Cache 中有效，而在其他 CPU 的 Cache 中存在副本。在该 Cache 行中的数据是当前处理器系统中最新的数据副本，而且与存储器中的数据一致。
- S (Shared) 位。S 位为 1 表示 Cache 行中包含的数据有效，而且在当前 CPU 和至少其他一个 CPU 中具有副本。在该 Cache 行中的数据是当前处理器系统中最新的数据副本，而且与存储器中的数据一致。

- I (Invalid) 位。I 位为 1 表示当前 Cache 行中没有有效数据或者该 Cache 行没有使能。

MESI 协议在进行 Cache 行替换时，将优先使用 I 位为 1 的 Cache 行。

MESI 协议还存在一些变种，如 MOESI 协议和 MESIF 协议。基于 MOESI 协议的 Cache 一致性模型如图 3-5 所示。AMD 处理器就使用 MOESI 协议。不同的处理器在实现 MOESI 协议时，状态机的转换原理类似，但是在处理上仍有较大的区别。

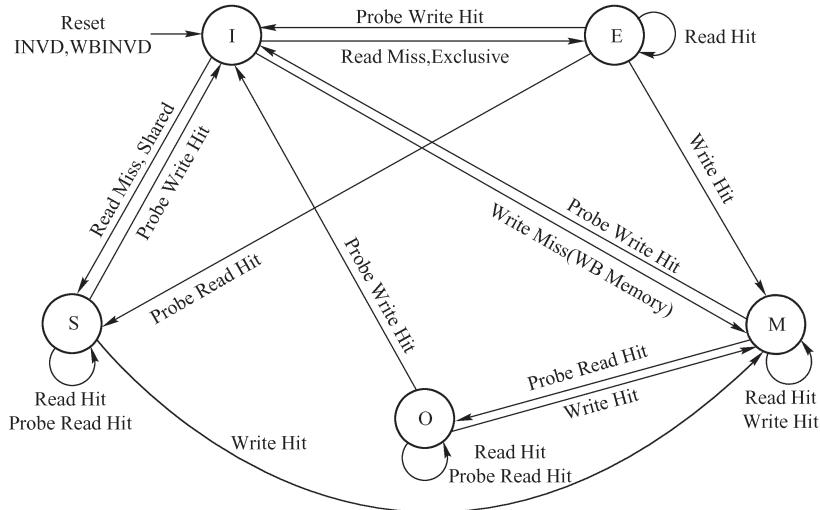


图 3-5 基于 MOESI 协议的 Cache 一致性模型

MOESI 协议引入了一个 O (Owned) 状态，并在 MESI 协议的基础上，重新定义了 S 状态，而 E、M 和 I 状态和 MESI 协议的对应状态相同。

- O 位。O 位为 1 表示在当前 Cache 行中包含的数据是当前处理器系统最新的数据副本，而且在其他 CPU 中一定具有该 Cache 行的副本，其他 CPU 的 Cache 行状态为 S。如果主存储器的数据在多个 CPU 的 Cache 中都具有副本时，有且仅有一个 CPU 的 Cache 行状态为 O，其他 CPU 的 Cache 行状态只能为 S。与 MESI 协议中的 S 状态不同，状态为 O 的 Cache 行中的数据与存储器中的数据并不一致。
- S 位。在 MOESI 协议中，S 状态的定义发生了细微的变化。当一个 Cache 行状态为 S 时，其包含的数据并不一定与存储器一致。如果在其他 CPU 的 Cache 中不存在状态为 O 的副本时，该 Cache 行中的数据与存储器一致；如果在其他 CPU 的 Cache 中存在状态为 O 的副本时，Cache 行中的数据与存储器不一致。

在一个处理器系统中，主设备（CPU 或者外部设备）进行存储器访问时，将试图从存储器系统（主存储器或者其他 CPU 的 Cache）中获得最新的数据副本。如果该主设备访问的数据没有在本地命中时，将从其他 CPU 的 Cache 中获取数据，如果这些数据仍然没有在其他 CPU 的 Cache 中命中，主存储器将提供数据。外设设备进行存储器访问时，也需要进行 Cache 共享一致性。

在 MOESI 模型中，“Probe Read” 表示主设备从其他 CPU 中获取数据副本的目的是为了读取数据；而“Probe Write” 表示主设备从其他 CPU 中获取数据副本的目的是为了写入数据；“Read Hit” 和 “Write Hit” 表示主设备在本地 Cache 中获得数据副本；“Read Miss” 和

“Write Miss” 表示主设备没有在本地 Cache 中获得数据副本；“Probe Read Hit” 和 “Probe Write Hit” 表示主设备在其他 CPU 的 Cache 中获得数据副本。

本节为简便起见，仅介绍 CPU 进行存储器写和与 O 状态相关的 Cache 行状态迁移，CPU 进行存储器读的情况相对较为简单，请读者自行分析这个过程。

当 CPU 对一段存储器进行写操作时，如果这些数据在本地 Cache 中命中时，其状态可能为 E、S、M 或者 O。

- 状态为 E 或者 M 时，数据将直接写入到 Cache 中，并将状态改为 M。
- 状态为 S 时，数据将直接写入到 Cache 中，并将状态改为 M，同时其他 CPU 保存该数据副本的 Cache 行状态将从 S 或者 O 迁移到 I (Probe Write Hit)。
- 状态为 O 时，数据将直接写入到 Cache 中，并将状态改为 M，同时其他 CPU 保存该数据副本的 Cache 行状态将从 S 迁移到 I (Probe Write Hit)。

当 CPU A 对一段存储器进行写操作时，如果这些数据没有在本地 Cache 中命中时，而在其他 CPU，如 CPU B 的 Cache 中命中时，其状态可能为 E、S、M 或者 O。其中 CPU A 使用 CPU B 在同一个 Cache 共享域中。

- Cache 行状态为 E 时，CPU B 将该 Cache 行状态改为 I；而 CPU A 将从本地申请一个新的 Cache 行，将数据写入，并该 Cache 行状态更新为 M。
- Cache 行状态为 S 时，CPU B 将该 Cache 行状态改为 I，而且具有同样副本的其他 CPU 的 Cache 行也需要将状态改为 I；而 CPU A 将从本地申请一个 Cache 行，将数据写入，并该 Cache 行状态更新为 M。
- Cache 行状态为 M 时，CPU B 将原 Cache 行中的数据回写到主存储器，并将该 Cache 行状态改为 I；而 CPU A 将从本地申请一个 Cache 行，将数据写入，并该 Cache 行状态更新为 M。
- Cache 行状态为 O 时，CPU B 将原 Cache 行中的数据回写到主存储器，并将该 Cache 行状态改为 I，具有同样数据副本的其他 CPU 的 Cache 行也需要将状态从 S 更改为 I；CPU A 将从本地申请一个 Cache 行，将数据写入，并该 Cache 行状态更新为 M。

Cache 行状态可以从 M 迁移到 O。例如当 CPU A 读取的数据从 CPU B 中命中时，如果在 CPU B 中 Cache 行的状态为 M 时，将迁移到 O，同时 CPU B 将数据传送给 CPU A 新申请的 Cache 行中，而且 CPU A 的 Cache 行状态将被更改为 S。

当 CPU 读取的数据在本地 Cache 中命中，而且 Cache 行状态为 O 时，数据将从本地 Cache 获得，并不会改变 Cache 行状态。如果 CPU A 读取的数据在其他 Cache 中命中，如在 CPU B 的 Cache 中命中而且其状态为 O 时，CPU B 将该 Cache 行状态保持为 O，同时 CPU B 将数据传送给 CPU A 新申请的 Cache 行中，而且 CPU A 的 Cache 行状态将被更改为 S。

在某些应用场合，使用 MOESI 协议将极大提高 Cache 的利用率，因为该协议引入了 O 状态，从而在发送 Read Hit 的情况时，不必将状态为 M 的 Cache 回写到主存储器，而是直接从一个 CPU 的 Cache 将数据传递到另外一个 CPU。目前 MOESI 协议在 AMD 和 RMI 公司的处理器中得到了广泛的应用。

Intel 提出了另外一种 MESI 协议的变种，即 MESIF 协议，该协议与 MOESI 协议有较大的不同，也远比 MOESI 协议复杂，该协议由 Intel 的 QPI (QuickPath Interconnect) 技术引入，其主要目的是解决“基于点到点的全互连处理器系统”的 Cache 共享一致性问题，而

不是“基于共享总线的处理器系统”的 Cache 共享一致性问题。

在基于点到点互连的 NUMA (Non-Uniform Memory Architecture) 处理器系统中，包含多个子处理器系统，这些子处理器系统由多个 CPU 组成。如果这个处理器系统需要进行全机 Cache 共享一致性，该处理器系统也被称为 ccNUMA (Cache Coherent NUMA) 处理器系统。MESIF 协议主要解决 ccNUMA 处理器结构的 Cache 共享一致性问题，这种结构通常使用目录表，而不使用总线监听处理 Cache 的共享一致性。

MESIF 协议引入了一个 F (Forward) 状态。在 ccNUMA 处理器系统中，可能在多个处理器的 Cache 中存在相同的数据副本，在这些数据副本中，只有一个 Cache 行的状态为 F，其他 Cache 行的状态都为 S。Cache 行的状态位为 F 时，Cache 中的数据与存储器一致。

当一个数据请求方读取这个数据副本时，只有状态为 F 的 Cache 行，可以将数据副本转发给数据请求方，而状态位为 S 的 Cache 不能转发数据副本。从而 MESIF 协议有效解决了在 ccNUMA 处理器结构中，所有状态位为 S 的 Cache 同时转发数据副本给数据请求方，而造成的数据拥塞。

在 ccNUMA 处理器系统中，如果状态位为 F 的数据副本，被其他 CPU 复制时，F 状态位将被迁移，新建的数据副本的状态位将为 F，而老的数据副本的状态位将改变为 S。当状态位为 F 的 Cache 行被改写后，ccNUMA 处理器系统需要首先 Invalidate 状态位为 S 其他的 Cache 行，之后将 Cache 行的状态更新为 M。

独立地研究 MESIF 协议并没有太大意义，该协议由 Boxboro-EX 处理器系统^①引入，目前 Intel 并没有公开 Boxboro-EX 处理器系统的详细设计文档。MESIF 协议仅是解决该处理器系统中 Cache 一致性的一个功能，该功能的详细实现与 QPI 的 Protocol Layer 相关，QPI 由多个层次组成，而 Protocol Layer 是 QPI 的最高层。

对 MESIF 协议 QPI 互连技术有兴趣的读者，可以在深入理解“基于目录表的 Cache 一致性协议”的基础上，阅读 Robert A. Maddox, Gurbir Singh and Robert J. Safranek 合著的书籍“Weaving High Performance Multiprocessor Fabric”以了解该协议的实现过程和与 QPI 互连技术相关的背景知识。

值得注意的是，MESIF 协议解决主要的问题是 ccNUMA 架构中 SMP 子系统与 SMP 子系统之间 Cache 一致性。而在 SMP 处理器系统中，依然需要使用传统的 MESI 协议。Nehalem EX 处理器也可以使用 MOESI 协议进一步优化 SMP 系统使用的 Cache 一致性协议，但是并没有使用该协议。

为简化起见，本章假设处理器系统使用 MESI 协议进行 Cache 共享一致性，而不是 MOESI 协议或者 MESIF 协议。

2. HIT#和 HITM#信号

在 SMP 处理器系统中，每一个 CPU 都使用 HIT# 和 HITM# 信号反映 HOST 主桥访问的地址是否在各自的 Cache 中命中。当 HOST 主桥访问存储器时，CPU 将驱动 HITM# 和 HIT# 信号，其描述如表 3-1 所示。

^① Boxboro-EX 处理器系统由多个 Nehalem EX 处理器组成，而 Nehalem EX 处理器由两个 SMP 处理器系统组成，一个 SMP 处理器系统由 4 个 CPU 组成，而每一个 CPU 具有 2 个线程。其中 SMP 处理器系统之间使用 QPI 进行连接，而在一个 SMP 处理器内部的各个 CPU 仍然使用 FSB 连接。

表 3-1 HITM#和 HIT#信号的含义

HITM#	HIT#	描述
1	1	表示 HOST 主桥访问的地址没有在 CPU 的 Cache 中命中
1	0	表示 HOST 主桥访问的地址在 CPU 的 Cache 中命中，而且 Cache 的状态为 S (Shared) 或者 E (Exclusive)，即 Cache 中的数据与存储器的数据一致
0	1	表示 HOST 主桥访问的地址在 CPU 的 Cache 中命中，而且 Cache 的状态为 M (Modified)，即 Cache 中的数据与存储器的数据不一致，在 Cache 中保存最新的数据副本
0	0	MESI 协议规定这种情况不允许出现，但是在有些处理器系统中仍然使用了这种状态，表示暂时没有获得是否在 Cache 中命中的信息，需要等待几拍后重试

HIT#和 HITM#信号是 FSB 中非常重要的两个信号，各个 CPU 的 HIT#和 HITM#信号通过“线与”方式直接相连[⊖]。而在一个实际 FSB 中，还包括许多信号，本节并不会详细介绍这些信号。

3. Cache 一致性协议中使用的 Agent

在处理器系统中，与 Cache 一致性相关的 Agent 如下所示。

- Request Agent。FSB 总线事务的发起设备。在本节中，Request Agent 特指 HOST 主桥。实际上在 FSB 总线上的其他设备也可以成为 Request Agent，但这些 Request Agent 并不是本节的研究重点。Request Agent 需要进行总线仲裁后，才能使用 FSB，在多数处理器的 FSB 中，需要对地址总线与数据总线分别进行仲裁。
- Snoop Agents。FSB 总线事务的监听设备。Snoop Agents 为 CPU，在一个 SMP 处理器系统中，有多个 CPU 共享同一个 FSB，此时这些 CPU 都是这条 FSB 上的 Snoop Agents。Snoop Agents 监听 FSB 上的存储器读写事务，并判断这些总线事务访问的地址是否在 Cache 中命中。Snoop Agents 通过 HIT#和 HITM#信号向 FSB 通知 Cache 命中的结果。在某些情况下，Snoop Agents 需要将 Cache 中的数据回写到存储器，同时为 Request Agent 提供数据。
- Response Agent。FSB 总线事务的目标设备。在本节中，Response Agent 特指存储器控制器。Response Agent 根据 Snoop Agents 提供的监听结果，决定如何接收数据或者向 Request Agent 设备提供数据。在多数情况下，当前数据访问没有在 Snoop Agents 中命中时，Response Agent 需要提供数据，此外 Snoop Agents 有时需要将数据回写到 Response Agent 中。

4. FSB 的总线事务

一个 FSB 的总线事务由多个阶段组成，包括 Request Phase、Snoop Phase、Response Phase 和 Data Phase。目前在多数高端处理器中，FSB 支持流水操作，即在同一个时间段内，不同的阶段可以重叠，如图 3-6 所示。

在一个实际的 FSB 中，一个总线事务还可能包含 Arbitration Phase 和 Error Phase。而本节仅讲述图 3-6 中所示的 4 个基本阶段。

- Request Phase。Request Agent 在获得 FSB 的地址总线的使用权后，在该阶段将访问数据区域的地址和总线事务类型发送到 FSB 上。

[⊖] HIT#和 HITM#信号是 Open Drain (开漏) 信号，Open Drain 信号可以直接相连，而不用使用逻辑门。

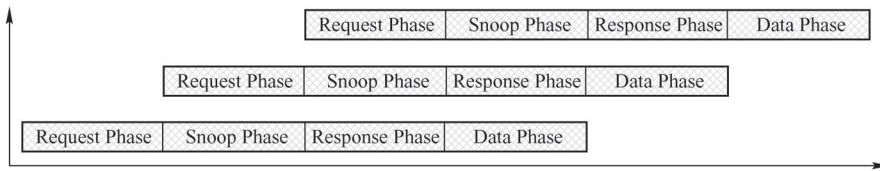


图 3-6 FSB 的流水操作

- Snoop Phase。Snoop Agents 根据访问数据区域在 Cache 中的命中情况，使用 HIT# 和 HITM# 信号，向其他 Agents 通知 Cache 一致性的结果。有时 Snoop Agent 需要将数据回写到存储器。
- Response Phase。Response Agent 根据 Request 和 Snoop Phase 提供的信号，可以要求 Request Agent 重试（Retry），或者 Response Agent 延时处理（Defer）当前总线事务。在 FSB 总线事务的各个阶段中，该步骤的处理过程最为复杂。本章将在下文结合 PCI 设备的 DMA 读写执行过程，说明该阶段的实现原理。
- Data Phase。一些不传递数据的 FSB 总线事务不包含该阶段。该阶段用来进行数据传递，包括 Request Agent 向 Response Agent 写入数据；Response Agent 为 Request Agent 提供数据；和 Snoop Agent 将数据回写到 Response Agent。

下面将使用本小节中的概念，描述在 PCI 总线上，与 Cache 相关的总线事务，并讲述相关的 FSB 的操作流程。

3.3.2 PCI 设备对不可 Cache 的存储器空间进行 DMA 读写

在 x86 处理器和 PowerPC 处理器中，PCI 设备对“不可 Cache 的存储器空间”进行 DMA 读写的过程并不相同。其中 PowerPC 处理器对“不可 Cache 的存储器空间”进行 DMA 读写进行了专门的处理，而 x86 处理器在对这类空间操作时，效率相对较低。

1. x86 处理器

x86 处理器使用 MTRR（Memory Type Range Register）寄存器设置存储器空间的属性，如果存储器空间为“可 Cache 空间”，x86 处理器还可以进一步设置这段空间为“Write Through”、“Write Combining”、“Write Protect” 和 “Write Back”。但是这些设置与 PCI 设备进行 DMA 操作时，是否进行 Cache 一致性操作并没有直接关系。

在 x86 处理器系统中，一个 PCI 设备进行 DMA 写操作，可以将数据从 PCI 设备写入到主存储器中。这个数据首先需要通过 HOST 主桥，然后经过 FSB 发送到存储器控制器。虽然在 x86 处理器系统中，CPU 知道这个存储器区域是否为“可 Cache 的”，但是 HOST 主桥并不知道 PCI 设备访问的存储器地址是否为“可 Cache 的”，因此都需要使用“Cache 一致”的 FSB 总线传送事务[⊖]进行存储器写操作，从而数据在发向 FSB 时，CPU 必须要进行总线监听，通知 FSB 总线这段空间是“不可 Cache 的”。

在 x86 处理器中，PCI 设备向不可 Cache 的存储器空间进行读操作时，CPU 也必须进行

[⊖] FSB 总线定义了许多总线事务，有的 FSB 总线提供了一个 Snoop 信号，该信号为 1 时表示当前 FSB 的总线事务需要进行 Cache 共享一致性，为 0 时不需要进行 Cache 共享一致性。

Cache 共享一致性操作，而这种没有必要的 Cache 共享一致性操作将影响 PCI 总线的传送效率。当 PCI 设备所访问的存储器空间没有在 CPU 的 Cache 命中时，CPU 会通知 FSB，数据没有在 Cache 中命中，此时 PCI 设备访问的数据将从存储器中直接读出。

x86 处理器在前端总线上进行 Cache 共享一致性操作时，需要使用 Snoop Phase，如果 PCI 设备能事先得知所访问的存储器是“不可 Cache 的”，就不必在前端总线上进行 Cache 共享一致性操作，即 FSB 总线事务不必包含 Snoop Phase，从而可以提高前端总线的使用效率。但是 x86 处理器并不支持这种方式。

在 x86 处理器系统中，无论 PCI 设备访问的存储器空间是否为“不可 Cache 的”，都需要进行 Cache 共享一致性操作。这也是 PCI 总线在 x86 处理器使用中的一个问题。而 PCIe 总线通过在数据报文中设置“Snooping”位解决了这个问题，有关 PCIe 总线 Snooping 位的内容参见第 6.1.3 节。

2. PowerPC 处理器

在 MPC8548 处理器中，HOST 主桥可以通过 PIWARn 寄存器[⊖]的 RTT 字段和 WTT 字段预知 PCI 设备访问的存储器空间是否为可 Cache 空间。当 HOST 主桥访问“不可 Cache 空间时”，可以使用 FSB 总线的“不进行 Cache 一致性”的总线事务。

此时 PowerPC 处理器不会在 FSB 总线中进行 Cache 一致性操作，即忽略 FSB 总线事务的 Snoop Phase。PCI 设备进行 DMA 写时，数据将直接进入主存储器，而 PCI 设备进行 DMA 读所读取的数据将直接从主存储器获得。与 x86 处理器相比，PowerPC 处理器可以忽略 CPU 进行总线监听的动作，从而提高了 FSB 传送效率。

3.3.3 PCI 设备对可 Cache 的存储器空间进行 DMA 读写

PCI 设备向“可 Cache 的存储器空间”进行读操作的过程相对简单。对于 x86 处理器或者 PowerPC 处理器，如果访问的数据在 Cache 中命中，CPU 会通知 FSB 总线，PCI 设备所访问的数据在 Cache 中。

首先 HOST 主桥发起存储器读总线事务，并在 Request Phase 中提供地址。Snoop Agent 在 Snoop Phase 进行总线监听，并通过 HIT# 和 HITM# 信号将监听结果通知给 Response Agent。如果 Cache 行的状态为 E 时，Response Agent 将提供数据，而 CPU 不必改变 Cache 行状态。如果 Snoop Agent 可以直接将数据提供给 HOST 主桥，无疑数据访问的延时更短，但是采用这种方法无疑会极大地提高 Cache Memory 系统的设计难度，因此采用这种数据传送方式的处理器[⊖]并不多。

如果 Cache 行的状态为 M 时，Response Agent 在 Response Phase 阶段，要求 Snoop Agent 将 Cache 中的数据回写到存储器，并将 Cache 行状态更改为 E。Snoop Agent 在 Data Phase 将 Cache 中的数据回写给存储器控制器，同时为 HOST 主桥提供数据。Snoop Agent 也可以直接将数据提供给 HOST 主桥，不需要进行数据回写过程，也不更改 Cache 行状态，但是采用这种方法会提高 Cache Memory 系统的设计难度。

⊖ 该寄存器在 Inbound 寄存器组中，详见第 2.2.3 节。

⊖ 目前 Cortex A8/A9 和 Intel 的 Sandy Bridge 处理器支持这种方式。

如果 PCI 设备访问的数据没有在 Cache 中命中，Snoop Agents 会通知 FSB 总线，PCI 设备所访问的数据不在 Cache 中，此时存储器控制器（Response Agent）将在 Data Phase 向 HOST 主桥提供数据。

PCI 设备向“可 Cache 的”存储器区域进行写操作，无论对于 PowerPC 处理器还是 x86 处理器，都较为复杂。当 HOST 主桥通过 FSB 将数据发送给存储器控制器时，在这个系统总线上的所有 CPU 都需要对这个 PCI 写操作进行监听，并根据监听结果，合理地改动 Cache 行状态，并将数据写入存储器。

下面以图 3-7 所示的 SMP 处理器系统为例，说明 PCI 设备对“可 Cache 的存储器空间”进行 DMA 写的实现过程。

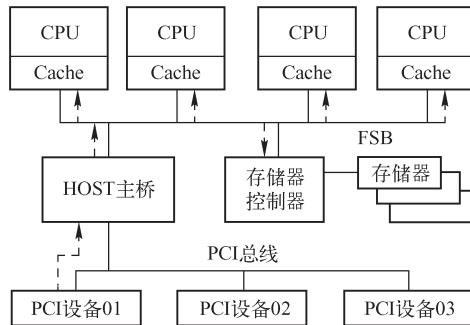


图 3-7 PCI 设备向可 Cache 的存储器空间进行写操作

在图 3-7 所示的处理器系统中，存在 4 个 CPU，这 4 个 CPU 通过一条 FSB 连接在一起，CPU 之间使用 MESI 协议进行 Cache 一致性处理，而 HOST 主桥和存储器控制器与 FSB 直接相连。HOST 主桥向存储器控制器传递数据时，需要处理 Cache 的一致性。

在这个处理器系统中，当 PCI 设备，如 PCI 设备 01，进行 DMA 写操作时，数据将首先到达 HOST 主桥，而 HOST 主桥将首先接管该 PCI 设备数据访问并将其转换为 FSB 总线事务，并在 Request Phase 中，提供本次 FSB 总线事务的地址。CPU 将在 Snoop Phase 对这个地址进行监听，判断当前地址在 Cache 中的命中情况。

当 HOST 主桥访问的地址不在 Cache 中命中时，此时在处理器系统中，所有 CPU 都没有驱动 HIT# 和 HITM# 信号，HIT# 和 HITM# 信号都为 1，表示 HOST 主桥访问的地址没有在 CPU 的 Cache 中命中，HOST 主桥可以简单地将数据写入存储器。当 HOST 主桥访问的存储器地址在 Cache 中命中时，Cache 行的状态可以为 S、E 或者为 M，此时处理器系统的处理过程相对较为复杂，下一节将专门讨论这种情况。

3.3.4 PCI 设备进行 DMA 写时发生 Cache 命中

如果 PCI 设备访问的地址在某个 CPU 的 Cache 行中命中时，可能会出现三种情况。

第一种情况是命中的 Cache 行其状态为 E，即 Cache 行中的数据与存储器中的数据一致；而第二种情况是命中的 Cache 行其状态为 S。其中 E 位为 1 表示该数据在 SMP 处理器系统中，有且仅有一个 CPU 的 Cache 中具有数据副本；而 S 位为 1 表示在 SMP 处理器系统中，该数据至少在两个以上 CPU 的 Cache 中具有数据副本。

当 Cache 行状态为 E 时，这种情况比较容易处理。因为 PCI 设备（通过 HOST 主桥）写

入存储器的信息比 Cache 行中的数据新，而且 PCI 设备在进行 DMA 写操作之前，存储器与 Cache 中数据一致，此时 CPU 仅需要在 Snoop Phase 使无效（Invalidate）这个 Cache 行，然后 FSB 总线事务将数据写入存储器即可。当然如果 FSB 总线事务可以将数据直接写入 Cache，并将 Cache 行的状态更改为 M，也可提高 DMA 写的效率，这种方式的实现难度较大，第 3.3.5 节将介绍这种优化方式。

Cache 行状态为 S 时的处理情况与状态为 E 时的处理情况大同小异，PCI 设备在进行写操作时也将数据直接写入主存储器，并使无效状态为 S 的 Cache 行。

第三种情况是命中的 Cache 行其状态为 M（Modified），即 Cache 行中的数据与存储器的数据不一致，Cache 行中保存最新的数据副本，主存储器中的部分数据无效。对于 SMP 系统，此时有且仅有一个 CPU 中的 Cache 行的状态为 M，因为 MESI 协议规定存储器中的数据不能在多个 CPU 的 Cache 行中的状态为 M。

我们假定一个处理器的 Cache 行长度为 32B，即 256b。当这个 Cache 行的状态为 M 时，表示这个 Cache 行的某个字节、双字、几个双字、或者整个 Cache 行中的数据比主存储器中含有的数据新。

假设 HOST 主桥访问的地址，在 Snoop Phase，通过 CPU 进行总线监听后，发现其对应的 Cache 行状态为 M。此时 HOST 主桥进行存储器写操作时，处理情况较为复杂，此时这些状态为 M 的数据需要回写到主存储器。

下面考虑如图 3-8 所示的实例。假定处理器的 Cache 使用回写（Write-Back）策略进行更新。在这个实例中，HOST 主桥对存储器的某个地址进行写操作，而所有 CPU 通过 FSB 总线进行总线监听时发现，HOST 主桥使用的这个目的地址在某个 CPU 的 Cache 行命中，此时这个 CPU 将置 HIT#信号为 0，并置 HIT#信号为 1，表示当前 Cache 行中含有的数据比存储器中含有的数据更新。

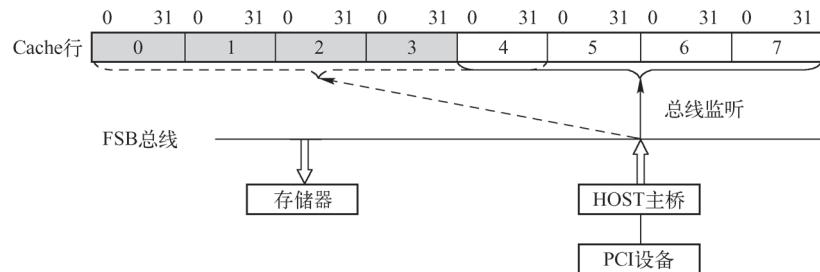


图 3-8 PCI 设备向 Cache 行状态为 M 的存储器进行写操作

假设此时在 Cache 行中，阴影部分的数据比存储器中的数据新，而其他数据与存储器保持一致，即在这个 Cache 行中第 0 ~ 3 个双字的数据是当前处理器系统中最新的数据，而第 4 ~ 7 个双字中的数据与存储器保持一致。

如果 PCI 设备向存储器写的数据区域可以完全覆盖这些阴影部分，如对第 0 ~ 5 个双字进行写操作时，这种情况不难处理。此时 CPU 只需在总线监听阶段，将这个 Cache 行使无效，然后将数据写入存储器即可。因为完成这个存储器写操作之后，PCI 设备写入的数据是最新的，而且这个最新的数据将完全覆盖在 Cache 行中阴影部分的数据，所以 CPU 只需要简单地将这个 Cache 行使无效即可。

然而 PCI 设备（HOST 主桥）无法预先知道这些 Cache 行中的数据哪些是有效的，哪些是无效的，而仅知道命中了一个“被修改过”的 Cache 行，从而 PCI 设备（HOST 主桥）无法保证对 Cache 行中有效数据进行覆盖。因此 PCI 设备对存储器进行写操作时，不能简单地使无效（Invalid）状态位为 M 的 Cache 行。

仍然以图 3-8 为例，考虑一个 PCI 设备将 4 个双字（第 4 ~ 7 个双字）的数据写入到一个存储器中，这 4 个双字所访问的数据在某个 CPU 的 Cache 行中命中，而且该 Cache 行的状态为 M，而且这个 Cache 行的前 4 个双字曾被处理器修改过。

此时 CPU 对 FSB 总线监听时，不能简单将当前 Cache 行使无效，因为这个使无效操作将丢失阴影部分的有效数据。这个阴影部分中的有效数据并没有被 PCI 设备重新写入，因此在整个处理器系统中，这个阴影部分仍然包含最新的数据。将最新的数据丢弃显然是一种错误做法，会导致处理器系统的崩溃。

为此 HOST 主桥需要专门处理这种情况，不同的 HOST 主桥采用了不同的方法处理这种情况，但无外乎以下三种方法。

(1) CPU 进行总线监听后发现，HOST 主桥访问的数据命中了一个状态位为 M 的 Cache 行，此时存储器控制器将通知 HOST 主桥重试或者延时处理，并暂时停止 HOST 主桥发起的这次存储器写操作。随后 CPU 将状态位为 M 的 Cache 行与存储器进行同步后，再使无效这个 Cache 行。之后 HOST 主桥在合适的时机，重新发起被 HOST 主桥要求重试的总线事务，此时 CPU 再次进行总线监听时不会再次出现 Cache 命中的情况，因此 HOST 主桥可以直接将数据写入存储器。许多 HOST 主桥使用这种方法处理 PCI 设备的存储器写总线事务。

(2) 首先 HOST 主桥将接收 PCI 设备进行 DMA 写的数据，并将这些数据放入存储器控制器的一个缓冲区中，同时结束 PCI 设备的存储器写总线事务。之后 CPU 进行总线监听，如果 CPU 发现 HOST 主桥访问的数据命中了一个状态位为 M 的 Cache 行时，则这个 Cache 行放入存储器控制器的另一个缓冲区后，使无效这个 Cache 行。最后存储器控制器将这两个缓冲区的数据合并然后统一写入到存储器中。

(3) HOST 主桥并不结束当前 PCI 总线周期，而直接进行总线监听，如果 CPU 进行总线监听发现 HOST 主桥访问的数据命中了一个状态位为 M 的 Cache 行时，则将这个 Cache 行整体写入存储器控制器的缓冲区后使无效这个 Cache 行，之后 HOST 主桥开始从 PCI 设备接收数据，并将这些数据直接写入这个缓冲区中。最后 HOST 主桥结束 PCI 设备的存储器写总线周期，同时存储器控制器将这个缓冲区内的数据写入存储器。

以上这几种情况是 PCI 设备进行存储器写时，HOST 主桥可能的处理情况，其中第 1 种方法最常用。而 x86 处理器使用的 implicit writeback 方式，与第 2 种方法基本类似。第 3 种方法与第 2 种方法并没有本质不同。

但是如果 PCI 设备对一个或者多个完整 Cache 行的存储器区域进行写操作时，上述过程显得多余。对完整 Cache 行进行写操作，可以保证将 Cache 行对应的存储器区域完全覆盖，此时 Cache 行中的数据在 PCI 设备完成这样的操作后，在处理器系统中将不再是最新的。PCI 设备进行这样的存储器写操作时，可以直接将数据写入存储器，同时直接使无效状态为 M 的 Cache 行。

PCI 总线使用存储器写并无效（Memory Write and Invalidate）总线事务，支持这种对一个完整 Cache 行进行的存储器写总线事务。PCI 设备使用这种总线事务时，必须事先知道当

前处理器系统中 CPU 使用的 Cache 行大小，使用这种总线事务时，一次总线事务传递数据的大小必须以 Cache 行为单位对界。为此 PCI 设备必须使用配置寄存器 Cache Line Size 保存当前 Cache 行的大小，Cache Line Size 寄存器在 PCI 配置空间的位置见图 2-9。

存储器读（Memory Read）、存储器多行读（Memory Read Multiple）和存储器单行读（Memory Read Line）总线事务也是 PCI 总线中的重要总线事务，这些总线事务不仅和 Cache 有关，还和 PCI 总线的预读机制有关，在第 3.4.5 节中将重点介绍这些总线事务。

3.3.5 DMA 写时发生 Cache 命中的优化

在许多高性能处理器中，还提出了一些新的概念，以加速外设到存储器的 DMA 写过程。如 Freescale 的 I/O Stashing 和 Intel 的 IOAT 技术。

如图 3-8 所示，当设备进行存储器写时，如果可以对 Cache 直接进行写操作，即便这个存储器写命中了一个状态为 M 的 Cache 行，也不必将该 Cache 行的数据回写到存储器中，而是直接将数据写入 Cache，之后该 Cache 行的状态依然为 M。采用这种方法可以有效提高设备对存储器进行写操作的效率。采用直接向 Cache 行写的方法，PCI 设备对存储器写命中一个状态为 M 的 Cache 行时，将执行以下操作。

- (1) HOST 主桥将对存储器的写请求发送到 FSB 总线上。
- (2) CPU 通过对 FSB 监听，发现该写请求在某个 Cache 行中命中，而且该 Cache 行的状态为 M。
- (3) HOST 主桥将数据直接写入到 Cache 行中，并保持 Cache 行的状态为 M。注意此时设备不需要将数据写入存储器中。

从原理上看，这种方法并没有奇特之处，只要 Cache 能够提供一个接口，使外部设备能够直接写入即可。但是从具体实现上看，设备直接将数据写入 Cache 中，还是有相当大的难度。特别是考虑在一个处理器中，可能存在多级 Cache，当 CPU 进行总线监听时，可能是在 L1、L2 或者 L3 Cache 中命中，此时的情况较为复杂，多级 Cache 间的协议状态机远比 FSB 总线协议复杂得多。

在一个处理器系统中，如果 FSB 总线事务在“与 FSB 直接相连的 Cache”中命中时，这种情况相对容易处理；但是在与 BSB（Back-Side Bus）直接相连的 Cache 命中时，这种情况较难处理。下面分别对这两种情况进行讨论，在一个处理器中，采用 FSB 和 BSB 连接 Cache 的拓扑如图 3-9 所示。

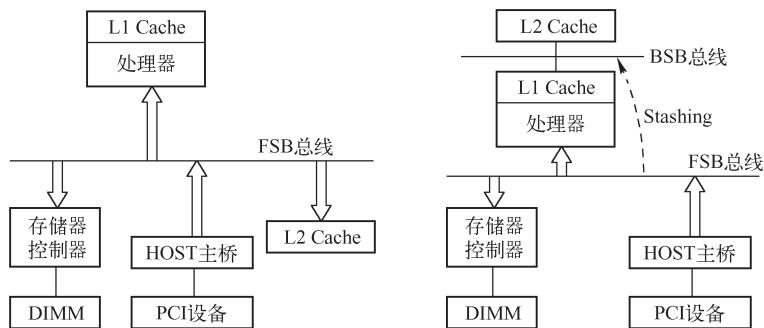


图 3-9 采用 FSB/BSB 进行 Cache 连接

当采用 FSB 总线连接 L2 Cache 时，L2 Cache 直接连接到 FSB 总线上，设备通过 FSB 总线向 L2 Cache 进行写操作并不难实现，MPC8548 处理器就是采用了这种结构将 L2 Cache 直接连接到 FSB 总线上。

但是由于 FSB 总线的频率低于 BSB 总线频率，因此采用这种结构将影响 L2 Cache 的访问速度，为此高端处理器多采用 BSB 总线连接 L2 Cache，x86 处理器在 Pentium Pro 之后的高性能处理器都使用 BSB 总线连接 L2 Cache，Freescale 的 G4 系列处理器和最新的 P4080 处理器也使用 BSB 总线连接 L2 Cache。

当 L2 Cache 没有直接连接到 FSB 上时，来自外部设备的数据不容易到达 BSB 总线。除了需要考虑 Cache 连接在 BSB 总线的情况下，在外部设备进行 DMA 操作时，还需要考虑多处理器系统的 Cache 共享一致性协议。设计一个专用通道，将数据从外部设备直接写入到处理器的 Cache 中并不容易实现。Intel 的 IOAT 和 Freescale 的 I/O Stashing 可能使用了这种专用通道技术，直接对 L1 和 L2 Cache 进行写操作，并在极大增加了设计复杂度的前提下，提高了处理器系统的整体效率。

以上对 Cache 进行直接写操作，仅是 Intel 的 IOAT 和 Freescale 的 I/O Stashing 技术的一个子集。目前 Intel 和 Freescale 没有公开这些技术的具体实现细节。在一个处理器系统中，可能存在多级 Cache，这些 Cache 的层次组成结构和状态机模型异常复杂，本章对这些内容不做进一步说明。

3.4 预读机制

随着处理器制造工艺的进步，处理器主频越来越高，存储器和外部设备的访问速度虽然也得到极大的提升，但是依然不与处理器主频的提升速度成正比，从而处理器的运行速度和外部设备的访问速度之间的差距越来越大，存储器瓶颈问题愈发严重。虽然 Cache 的使用有效缓解了存储器瓶颈问题，但是仅使用 Cache 远远不够。

因为无论 Cache 的命中率有多高，总有发生 Cache 行 Miss 的情况。一旦 Cache 行出现 Miss，处理器必须启动存储器周期，将需要的数据从存储器重新填入 Cache 中，这在某种程度上增加了存储器访问的开销。

使用预读机制可以在一定程度上降低 Cache 行失效所带来的影响。处理器系统可以使用的预读机制，包括指令 Fetch、数据预读、外部设备的预读队列和操作系统提供的预读策略。本章将简要介绍指令 Fetch，并重点介绍 CPU 如何对主存储器和外部设备进行数据预读。并以此为基础，详细说明 PCI 总线使用的预读机制。

3.4.1 指令 Fetch

指令预读是 CPU 指令流水的一个阶段。

在一段程序中，存在大量的分支预测指令，因而在某种程度上增加了指令 Fetch 的难度。因此如何判断程序的执行路径是指令流水首先需要解决的问题。

在 CPU 中通常设置了分支预测单元（Branch Predictor），在分支指令执行之前，分支预测单元需要预判分支指令的执行路径，从而进行指令 Fetch。但是分支预测单元并不会每次

都能正确判断分支指令的执行路径，这为指令 Fetch 制造了不小的麻烦，在这个背景下许多分支预测策略应运而生。

这些分支预测策略主要分为静态预测和动态预测两种方法。静态预测方法的主要实现原理是利用 Profiling 工具，静态分析程序的架构，并为编译器提供一些反馈信息，从而对程序的分支指令进行优化。如在 PowerPC 处理器的转移指令中存在一个“at”字段，该字段可以向 CPU 提供该转移指令是否 Taken[⊖] 的静态信息。在 PowerPC 处理器中，条件转移指令“be”表示 Taken；而“be-”表示 Not Taken。

CPU 的分支预测单元在分析转移指令时可以预先得知该指令的转移结果。目前在多数 CPU 中提供了动态预测机制，而且动态预测的结果较为准确。因此在实现中，许多 PowerPC 内核并不支持静态预测机制，如 E500 内核。

CPU 使用的动态预测机制是本节研究的重点。而在不同的处理器中，分支预测单元使用的动态预测算法并不相同。在一些功能较弱的处理器，如 8 b/16 b 微控制器中，分支指令的动态预测机制较为简单。在这些处理器中，分支预测单元常使用以下几种方法动态预测分支指令的执行。

(1) 分支预测单元每一次都将转移指令预测为 Taken，采用这种方法无疑非常简单，而且命中率在 50% 以上，因为无条件转移指令都是 Taken，当然使用这种方法的缺点也是显而易见的。

(2) 分支预测单元将向上跳转的指令预测为 Taken，而将向下跳转的指令预测为 Not Taken。分支预测单元使用的这种预测方式与多数程序的执行风格类似，但是这种实现方式并不理想。

(3) 一条转移指令被预测为 Taken，而之后这条转移指令的预测值与上一次转移指令的执行结果相同。

当采用以上几种方法时，分支预测单元的硬件实现代价较低，但是使用这些算法时，预测成功的概率较低。因此在高性能处理器中，如 PowerPC 和 x86 处理器并不会采用以上这 3 种方法实现分支预测单元。

目前在高性能处理器中，常使用 BTB (Branch Target Buffer) 管理分支预测指令。在 BTB 中含有多个 Entry，这些 Entry 由转移指令的地址低位进行索引，而这个 Entry 的 Tag 字段存放转移指令的地址高位。BTB 的功能相当于存放转移指令的 Cache，其状态机转换也与 Cache 类似。当分支预测单元第一次分析一条分支指令时，将在 BTB 中为该指令分配一个 Entry，同时也可能会淘汰 BTB 中的 Entry。目前多数处理器使用 LRU (Least recently used) 算法淘汰 BTB 中的 Entry。

在 BTB 的每个 Entry 中存在一个 Saturating Counter。该计数器也被称为 Bimodal Predictor，由两位组成，可以表示 4 种状态，为 0b11 时为“Strongly Taken”；为 0b10 时为“Weakly Taken”；为 0b01 时为“Weakly not Taken”；为 0b00 时为“Strongly not Taken”。

当 CPU 第一次预测一条转移指令的执行时，其结果为 Strongly Taken，此时 CPU 将在 BTB 中为该指令申请一个 Entry，并置该 Entry 的 Saturating Counter 为 0b11。此后该指令将按

[⊖] 为简便起见，下文将转移指令成功进行转移称为“Taken”；而将不进行转移称为“Not Taken”。

照 Saturating Counter 的值，预测执行，如果预测结果与实际执行结果不同时，将 Saturating Counter 的值减 1，直到其值为 0b00；如果相同时，将 Saturating Counter 的值加 1，直到其值为 0b11。目前 Power E500 内核和 Pentium 处理器使用这种算法进行分支预测。

使用 Saturating Counter 方法在处理转移指令的执行结果为 1111011111 或者 0000001000 时的效果较好，即执行结果大多数为 0 或者 1 时的预测结果较好。然而如果一条转移指令的执行结果具有某种规律，如为 010101010101 或者 001001001001 时，使用 Saturating Counter 并不会取得理想的预测效果。

在程序的执行过程中，一条转移指令在执行过程中出现这样有规律的结果较为常见，因为程序就是按照某种规则编写的，按照某种规则完成指定的任务。为此 Two-Level 分支预测方法应运而生。

Two-Level 分支预测方法使用了两种数据结构，一种是 BHR (Branch History Register)；而另一种是 PHT (Pattern History Table)。其中 BHR 由 k 位组成，用来记录每一条转移指令的历史状态，而 PHT 表含有 2^k 个 Entry，每个 Entry 由两位 Saturating Counter 组成。BHR 和 PHT 的关系如图 3-10 所示。

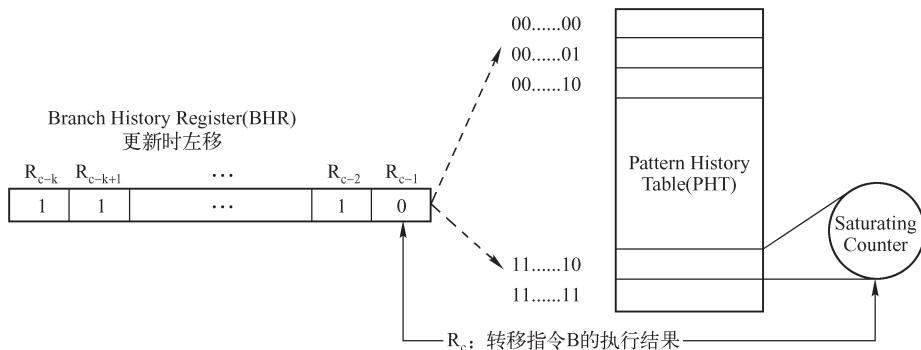


图 3-10 BHR 和 PHT 之间的关系

假设分支预测单元在使用 Two-Level 分支预测方法时，设置了一个 PBHT 表 (Per-address Branch History Table) 存放不同指令所对应的 BHR。在 PBHT 表中所有 BHR 的初始值为全 1，而在 PHT 表中所有 Entry 的初始值值为 0b11。BHR 在 PBHT 表中的使用方法与替换机制与 Cache 类似。

当分支预测单元分析预测转移指令 B 的执行时，将首先从 PBHT 中获得与转移指令 B 对应的 BHR，此时 BHR 为全 1，因此 CPU 将从 PHT 的第 11…11 个 Entry 中获得预测结果 0b11，即 Strongly Taken。转移指令 B 执行完毕后，将实际执行结果 R_c 更新到 BHR 寄存器中，并同时更新 PHT 中对应的 Entry。

当 CPU 再次预测转移指令 B 的执行时，仍将根据 BHR 索引 PHT 表，并从对应 Entry 中获得预测结果。而当指令 B 再次执行完毕后，将继续更新 BHR 和 PHT 表中对应的 Entry。当转移指令的执行结果具有某种规律 (Pattern) 时，使用这种方法可以有效提高预测精度。如果转移指令 B 的实际执行结果为 001001001…001，而且 k 等于 4 时，CPU 将以 0010-0100-1001 这样的循环访问 BHR，因此 CPU 将分别从 PHT 表中的第 0010、0100 和 1001 个 Entry 中获得准确的预测结果。

由以上描述可以发现，Two-Level 分支预测法具有学习功能，并可以根据转移指令的历史记录产生的模式，在 PHT 表中查找预测结果。该算法由 T. Y. Yeh 与 Y. N. Patt 在 1991 年提出，并在高性能处理器中得到了大规模应用。

Two-Level 分支预测法具有许多变种。目前 x86 处理器主要使用“Local Branch Prediction”和“Global Branch Prediction”两种算法。

在“Local Branch Prediction”算法中，每一个 BHR 使用不同的 PHT 表，Pentium II 和 Pentium III 处理器使用这种算法。该算法的主要问题是当 PBHT 表的 Entry 数目增加时，PHT 表将以指数速度增长，而且不能利用其他转移指令的历史信息进行分支预测。而在“Global Branch Prediction”算法中，所有 BHR 共享 PHT 表，Pentium M、Pentium Core 和 Core 2 处理器使用这种算法。

在高性能处理器中，分支预测单元对一些特殊的分支指令如“Loop”和“Indirect 跳转指令”设置了“Loop Prediction”和“Indirect Prediction”部件优化这两种分支指令的预测。此外分支预测单元，还设置了 RSB（Return Stack Buffer），当 CPU 调用一个函数时，RSB 将记录该函数的返回地址，当函数返回时，将从 RSB 中获得返回地址，而不必从堆栈中获得返回地址，从而提高了函数返回的效率。

目前在高性能处理器中，动态分支预测的主要实现机制是 CPU 通过学习以往历史信息，并进行预测，因而 Neural branch predictors 机制被引入，并取得了较为理想的效果，本节对这种分支预测技术不做进一步说明。目前指令的动态分支预测技术较为成熟，在高性能计算机中，分支预测的成功概率在 95% ~ 98% 之间，而且很难进一步提高。

3.4.2 数据预读

数据预读是指在处理器进行运算时，提前通知存储器系统将运算过程中需要的数据准备好，而当处理器需要这些数据时，可以直接从这些预读缓冲（通常指 Cache）中获得这些数据。Steven P. Vanderwiel 与 David J. Lilja 总结了最近出现的各类数据预读机制，下面将以图 3-11 为例进一步探讨这些数据预读机制。

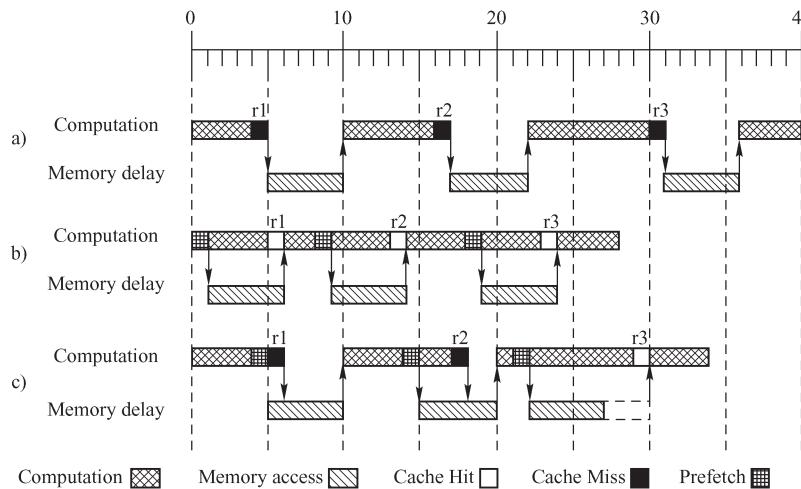


图 3-11 数据预读机制示意图

图 3-11 列举了三个实例说明数据预读的作用。其中实例 a 没有使用预读机制；实例 b 是一个采用预读机制的理想情况；而实例 c 是一个采用预读机制的次理想情况。我们假设处理器执行某个任务需要经历四个阶段，每个阶段都由处理器执行运算指令和存储指令组成。而处理器一次存储器访问需要 5 个时钟周期。其中每一个阶段的定义如下所示。

- (1) 处理器执行 4 个时钟周期后需要访问存储器。
- (2) 处理器执行 6 个时钟周期后需要访问存储器。
- (3) 处理器执行 8 个时钟周期后需要访问存储器。
- (4) 处理器执行 4 个时钟周期后完成。

实例 a 由于没有使用预读机制，因此在运算过程中需要使用存储器中的数据时，不可避免地出现 Cache Miss。实例 a 执行上述任务的过程如下。

- (1) 执行第一阶段任务的 4 个时钟周期，之后访问存储器，此时将发生 Cache Miss。
- (2) Cache Miss 需要使用一个时钟周期^①，然后在第 5 个时钟周期启动存储器读操作。
- (3) 在第 10 个周期，处理器从存储器获得数据，继续执行第二阶段任务的 6 个时钟周期，之后访问存储器，此时也将发生 Cache Miss。
- (4) 处理器在第 17 ~ 22 时钟周期从存储器读取数据，并在第 22 个时钟周期继续执行第三阶段任务的 8 个时钟周期，之后访问存储器，此时也将发生 Cache Miss。
- (5) 处理器在第 31 ~ 36 时钟周期从存储器读取数据，并在第 36 个时钟周期继续执行第四阶段任务的 4 个时钟周期，完成整个任务的执行。

采用这种机制执行上述任务共需 40 个时钟周期。而使用预读机制，可以有效缩短整个执行过程，如图 3-11 中的实例 b 所示。在实例 b 中在执行过程中，都会提前进行预读操作，虽然这些预读操作也会占用一个时钟周期，但是这些预读操作是值得的。合理使用这些数据预读，完成同样的任务 CPU 仅需要 28 个时钟周期，从而极大提高了程序的执行效率，其执行过程如下。

- (1) 首先使用预读指令对即将使用的存储器进行预读^②，然后执行第一阶段任务的 4 个时钟周期。当处理器进行存储器读时，数据已经准备好，处理器将在 Cache 中获得这个数据然后继续执行^③。
- (2) 处理器在执行第二阶段的任务时，先执行 2 个时钟周期之后进行预读操作，最后执行剩余的 4 个时钟周期。当处理器进行存储器读时，数据已经准备好，处理器将在 Cache 中获得这个数据然后继续执行。
- (3) 处理器执行第三阶段的任务时，先执行 4 个时钟周期之后进行预读操作，最后执行剩余的 4 个时钟周期。当处理器进行存储器读时，数据已经准备好，处理器将在 Cache 中获得这个数据然后继续执行。
- (4) 处理器执行第四阶段的任务，执行完 4 个时钟周期后，完成整个任务的执行。

当然这种情况是非常理想的，处理器在执行整个任务时，从始至终是连贯的，处理器执行和存储器访存完全并行，然而这种理想情况并不多见。

首先在一个任务的执行过程中，并不易确定最佳的预读时机；其次采用预读所获得数据

^① 假定从访问 Cache 到发现 Cache Miss 需要一个时钟周期。

^② PowerPC 处理器使用 dcbt 指令，而 x86 处理器使用 PREFETCHh 指令，实现这种软件预读。

^③ 假定从 Cache 中获得数据需要一个时钟周期。

并不一定能够被及时利用，因为在程序执行过程中可能会出现各种各样的分支选择，有时预读的数据并没有被及时使用。

在图 3-11 所示的实例 c 中，预读机制没有完全发挥作用，所以处理器在执行任务时，Cache Miss 时有发生，从而降低了整个任务的执行效率。即便这样，实例 c 也比完全没有使用预读的实例 a 的任务执行效率高一些。在实例 c 中，执行完毕图 3-11 中所示的任务共需要 34 个时钟周期。

但是在某些特殊情况下，采用预读机制有可能会降低效率。首先在一个较为复杂的应用中，很有可能预读的数据没有被充分利用，一个程序可能会按照不同的分支执行，而执行每一个分支所使用的数据并不相同。其次预读的数据即使是有效的，这些预读的数据也会污染整个 Cache 资源，在大规模并行任务的执行过程中，有可能引发 Cache 颠簸，从而极大地降低系统效率。

什么时候采用预读机制，关系到处理器系统结构的每一个环节，需要结合软硬件资源统筹考虑，并不能一概而论。处理器提供了必备的软件和硬件资源用以实现预读，而如何“合理”使用预读机制是系统程序员考虑的一个细节问题。数据预读可以使用软件预读或者硬件预读两种方式实现，下文将详细介绍这两种实现方式。

3.4.3 软件预读

软件预读机制由来已久，首先实现预读指令的处理器是 Motorola 的 88110 处理器，这颗处理器首先实现了“touch load”指令，这条指令是 PowerPC 处理器 dcbt 指令^①的雏形。88110 处理器是 Motorola 第一颗 RISC 处理器，具有里程碑意义。这颗处理器从内核到外部总线的设计都具有许多亮点，是 Motorola 对 PowerPC 架构做出的巨大贡献。PowerPC 架构中著名的 60X 总线也源于 88110 处理器。

后来绝大多数处理器都采用这类指令进行软件预读，Intel 在 i486 处理器中提出了 Dummy Read 指令，这条指令也是后来 x86 处理器中 PREFETCHh 指令^②的雏形。

这些软件预读指令都有一个共同的特点，就是在处理器真正需要数据之前，向存储器发出预读请求，这个预读请求不需要等待数据真正到达存储器之后，就可以执行完毕^③。从而处理器可以继续执行其他指令，以实现存储器访问与处理器运算同步进行，从而提高了程序的整体执行效率。由此可见，处理器采用软件预读可以有效提高程序的执行效率。下面考虑源代码 3-1 所示的实例。

源代码 3-1 源代码 3-1 没有采用软件预读机制的程序

```
int ip, a[N], b[N];  
  
for (i = 0; i < N; i++)  
    ip = ip + a[i] * b[i];
```

这个例子在对数组进行操作时被经常使用，这段源代码的作用是将 int 类型的数组 a 和

① dcbt 指令是 PowerPC 处理器的一条存储器预读指令，该指令可以将内存中的数据预读到 L1 或者 L2 Cache 中。

② PREFETCHh 指令是 x86 处理器的一条存储器预读指令。

③ 预读指令在一个时钟周期内就可以执行完毕。

数组 b 的每一项进行相乘，然后赋值给 ip，其中数组 a 和 b 都是 Cache 行对界的。源代码 3-1 中的程序并没有使用预读机制进行优化，因此这段程序在执行时会因为 a[i] 和 b[i] 中的数据不在处理器的 Cache 中，而必须启动存储器读操作。因此在这段程序在执行过程中，必须要等待存储器中的数据后才能继续，从而降低了程序的执行效率。为此将程序进行改动，如源代码 3-2 所示。

源代码 3-2 采用软件预读机制的程序

```
int ip, a[N], b[N];

for (i=0; i<N; i++) {
    fetch(&a[i+1]);
    fetch(&b[i+1]);
    ip = ip + a[i] * b[i];
}
```

以上程序对变量 ip 赋值之前，首先预读数组 a 和 b，当对变量 ip 赋值时，数组 a 和 b 中的数据已经在 Cache 中，因而不需要进行再次进行存储器操作，从而在一定程度上提高了代码的执行效率。以上代码仍然并不完美，首先，ip、a[0] 和 b[0] 并没有被预读，其次，在一个处理器中预读是以 Cache 行为单位进行的，因此对 a[0]、a[1] 进行预读时都是对同一个 Cache 行进行预读[⊖]，从而这段代码对同一个 Cache 行进行了多次预读，结果影响了执行效率。为此再次改动程序，如源代码 3-3 所示。

源代码 3-3 软件预读机制的改进程序

```
int ip, a[N], b[N];

fetch(&ip);
fetch(&a[0]);
fetch(&b[0]);

for (i=0; i<N-4; i+=4) {
    fetch(&a[i+4]);
    fetch(&b[i+4]);
    ip = ip + a[i] * b[i];
    ip = ip + a[i+1] * b[i+1];
    ip = ip + a[i+2] * b[i+2];
    ip = ip + a[i+3] * b[i+3];
}

for ( ; i<N; i++)
    ip = ip + a[i] * b[i];
```

[⊖] 假定这个处理器系统的 Cache 行长度为 4 个双字，即 128 位。

对于以上这个例子，采用这种预读方法可以有效提高执行效率，对此有兴趣的读者可以对以上几个程序进行简单的对比测试。但是提醒读者注意，有些较为先进的编译器，可以自动加入这些预读语句，程序员不必手工加入这些预读指令。实际上源代码 3-3 中的程序还可以进一步优化。这段程序的最终优化如源代码 3-4 所示。

源代码 3-4 软件预读机制的改进程序

```
int ip, a[N], b[N];  
  
fetch( &ip );  
  
for ( i = 0; i < 12; i += 4 ) {  
    fetch( &a[i] );  
    fetch( &b[i] );  
}  
  
for ( i = 0; i < N - 12; i += 4 ) {  
    fetch( &a[i + 12] );  
    fetch( &b[i + 12] );  
    ip = ip + a[i] * b[i];  
    ip = ip + a[i + 1] * b[i + 1];  
    ip = ip + a[i + 2] * b[i + 2];  
    ip = ip + a[i + 3] * b[i + 3];  
}  
  
for ( ; i < N; i ++ )  
    ip = ip + a[i] * b[i];
```

还可以对 ip、数据 a 和 b 进行充分预读之后，再一边预读数据，一边计算 ip 的值，最后计算 ip 的最终结果。使用这种方法可以使数据预读和计算充分并行，从而优化了整个任务的执行时间。

由以上程序可以发现，采用软件预读机制可以有效地对矩阵运算进行优化，因为矩阵运算进行数据访问时非常有规律，便于程序员或编译器进行优化，但是并不是所有程序都能如此方便地使用软件预读机制。此外预读指令本身也需要占用一个机器周期，在某些情况下，采用硬件预读机制更为合理。

3.4.4 硬件预读

采用硬件预读的优点是不需要软件进行干预，也不需要浪费一条预读指令来进行预读。但硬件预读的缺点是预读结果有时并不准确，有时预读的数据并不是程序执行所需要的。在许多处理器中这种硬件预读通常与指令预读协调工作。硬件预读机制的历史比软件预读更为久远，在 IBM 370/168 处理器系统中就已经支持硬件预读机制。

大多数硬件预读仅支持存储器到 Cache 的预读，并在程序执行过程中，利用数据的局部

性原理进行硬件预读。其中最为简单的硬件预读机制是 OBL (One Block Lookahead) 机制，采用这种机制，当程序对数据块 b 进行读取出现 Cache Miss 时，将数据块 b 从存储器更新到 Cache 中，同时对数据块 b + 1 也进行预读并将其放入 Cache 中；如果数据块 b + 1 已经在 Cache 中，则不进行预读。

这种 OBL 机制有很多问题，一个程序可能只使用数据块 b 中的数据，而不使用数据块 b + 1 中的数据，在这种情况下，采用 OBL 预读机制没有任何意义。而且使用这种预读机制时，每次预读都可能伴随着 Cache Miss，这将极大地影响效率。有时预读的数据块 b + 1 会将 Cache 中可能有用的数据替换出去，从而造成 Cache 污染。有时仅预读数据块 b + 1 可能并不足够，有可能程序下一个使用的数据块来自数据块 b + 2。

为了解决 OBL 机制存在的问题，有许多新的预读方法涌现出来，如“tagged 预读机制”。采用这种机制，将设置一个“tag 位”，处理器访问数据块 b 时，如果数据块 b 没有在 Cache 中命中，则将数据块 b 从存储器更新到 Cache 中，同时对数据块 b + 1 进行预读并将其放入 Cache 中；如果数据块 b 已经在 Cache 中，但是这个数据块 b 首次被处理器使用，此时也将数据块 b + 1 预读到 Cache 中；如果数据块 b 已经在 Cache 中，但是这个数据块 b 已经被处理器使用过，此时不将数据块 b + 1 预读到 Cache 中。

这种“tagged 预读机制”还有许多衍生机制，比如可以将数据块 b + 1, b + 2 都预读到 Cache 中，还可以根据程序的执行信息，将数据块 b - 1, b - 2 预读到 Cache 中。

但是这些方法都无法避免因为预读而造成的 Cache 污染问题，于是出现了 Stream buffer 机制。采用该机制，处理器可以将预读的数据块放入 Stream Buffer 中，如果处理器使用的数据没有在 Cache 中，则首先在 Stream Buffer 中查找，采用这种方法可以消除预读对 Cache 的污染，但是增加了系统设计的复杂性。

与软件预读机制相比，硬件预读机制可以根据程序执行的实际情况进行预读操作，是一种动态预读方法；而软件预读机制需要对程序进行静态分析，并由编译器自动或者由程序员手工加入软件预读指令来实现。

3.4.5 PCI 总线的预读机制

在一个处理器系统中，预读的目标设备并不仅限于存储器，程序员还可以根据实际需要对外部设备进行预读。但并不是所有的外部设备都支持预读，只有“well-behavior”存储器支持预读。处理器使用的内部存储器，如基于 SDRAM、DDR-SDRAM 或者 SRAM 的主存储器是“well-behavior”存储器，有些外部设备也是“well-behavior”存储器。这些 well-behavior 存储器具有以下特点。

(1) 对这些存储器设备进行读操作时不会改变存储器的内容。显然主存储器具有这种性质。如果一个主存储器的一个数据为 0，那么读取这个数据 100 次也不会将这个结果变为 1。但是在外部设备中，一些使用存储器映像寻址的寄存器具有读清除的功能。比如某些中断状态寄存器^①。当设备含有未处理的中断请求时，该寄存器的中断状态位为 1，对此寄存器进行读操作时，硬件将自动地把该中断位清零，这类采用存储映像寻址的寄存器就不是 well-behavior 存储器。

① 假设中断状态寄存器支持读清除功能。

(2) 对“well-behavior”存储器的多次读操作，可以合并为一次读操作。如向这个设备的地址 n , $n+4$, $n+8$ 和 $n+12$ 地址处进行四个双字的读操作，可以合并为对 n 地址的一次突发读操作（大小为 4 个双字）。

(3) 对“well-behavior”存储器的多次写操作，可以合并为一次写操作。如向这个设备的地址 n , $n+4$, $n+8$ 和 $n+12$ 地址处进行四个双字的写操作，可以合并为对 n 地址的一次突发写操作。对于主存储器，进行这种操作不会产生副作用，但是对于有些外部设备，不能进行这种操作。

(4) 对“well-behavior”的存储器写操作，可以合并为一次写操作。向这个设备的地址 n , $n+1$, $n+2$ 和 $n+3$ 地址处进行四个单字的写操作，可以合并为对 n 地址的一次 DW 写操作。对主存储器进行这种操作不会出现错误，但是对于有些外部设备，不能进行这种操作。

如果外部设备满足以上四个条件，该外部设备被称为“well-behavior”。PCI 配置空间的 BAR 寄存器中有一个“Prefetchable”位，该位为 1 时表示这个 BAR 寄存器所对应的存储器空间支持预读。PCI 总线的预读机制需要 HOST 主桥、PCI 桥和 PCI 设备的共同参与。在 PCI 总线上，预读机制需要分两种情况进行讨论，一个是 HOST 处理器通过 HOST 主桥和 PCI 桥访问最终的 PCI 设备；另一个是 PCI 设备使用 DMA 机制访问存储器。

PCI 总线预读机制的拓扑结构如图 3-12 所示。

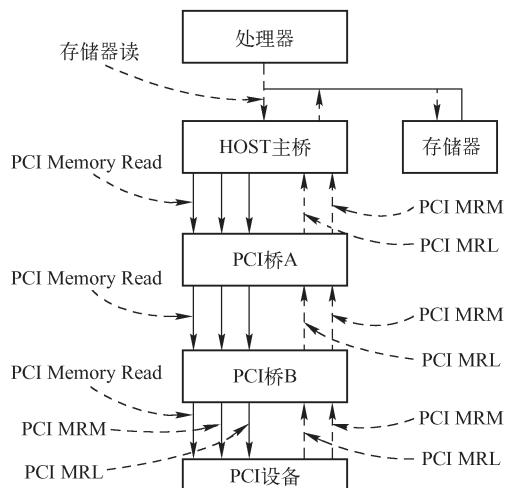


图 3-12 PCI 总线的预读

由上图所示，HOST 处理器预读 PCI 设备时，需要经过 HOST 主桥，并可能通过多级 PCI 桥，最终到达 PCI 设备，在这个数据传送路径上，有的 PCI 桥支持预读，有的不支持预读。而 PCI 设备对主存储器进行预读时也将经过多级 PCI 桥。PCI 设备除了可以对主存储器进行预读之外，还可以预读其他 PCI 设备，但是这种情况在实际应用中极少出现，本节仅介绍 PCI 设备预读主存储器这种情况。

1. HOST 处理器预读 PCI 设备

PCI 设备的 BAR 寄存器可以设置预读位，首先支持预读的 BAR 寄存器空间必须是一个 Well-behavior 的存储器空间，其次 PCI 设备必须能够接收来自 PCI 桥和 HOST 主桥的 MRM

(Memory Read Multiple) 和 MRL (Memory Read Line) 总线事务。

如果 PCI 设备支持预读，那么当处理器对这个 PCI 设备进行读操作时，可以通过 PCI 桥启动预读机制（该 PCI 桥也需要支持预读），使用 MRM 和 MRL 总线事务，对 PCI 设备进行预读，并将预读的数据暂时存放在 PCI 桥的预读缓冲中。

之后当 PCI 主设备继续读取 PCI 设备的 BAR 空间时，如果访问的数据在 PCI 桥的预读缓冲中，PCI 桥可以不对 PCI 设备发起存储器读总线事务，而是直接从预读缓冲中获取数据，并将其传递给 PCI 主设备。当 PCI 主设备完成读总线事务后，PCI 桥必须丢弃预读的数据以保证数据的完整性。此外当 PCI 桥预读的地址空间超越了 PCI 设备可预读 BAR 空间边界时，PCI 设备需要“disconnect”该总线事务。

如果 PCI 桥支持“可预读”的存储器空间，而且其下挂接的 PCI 设备 BAR 空间也支持预读时，系统软件需要从 PCI 桥“可预读”的存储器空间中为该 PCI 设备分配空间。此时 PCI 桥可以将从 PCI 设备预读的数据暂存在 PCI 桥的预读缓冲中。

PCI 总线规定，如果下游 PCI 桥地址空间支持预读，则其上游 PCI 桥地址空间既可以支持也可以不支持预读机制。如图 3-12 所示，如果 PCI 桥 B 管理的 PCI 子树使用了可预读空间时，PCI 桥 A 可以不支持可预读空间，此时 PCI 桥 A 只能使用存储器读总线事务读取 PCI 设备，而 PCI 桥 B 可以将这个存储器读总线事务转换为 MRL 或者 MRM 总线事务，预读 PCI 设备的 BAR 空间（如果 PCI 设备的 BAR 空间支持预读），并将预读的数据保存在 PCI 桥 B 的数据缓冲中。

但是 PCI 总线不允许 PCI 桥 A 从其“可预读”的地址空间中，为 PCI 桥 B 的“不可预读”区域预留空间，因为这种情况将影响数据的完整性。

大多数 HOST 主桥并不支持对 PCI 设备的预读，这些 HOST 主桥并不能向 PCI 设备发出 MRL 或者 MRM 总线事务。由于在许多处理器系统中，PCI 设备是直接挂接到 HOST 主桥上的，如果连 HOST 主桥也不支持这种预读，即便 PCI 设备支持了预读机制也没有实际作用。而且如果 PCI 设备支持预读机制，硬件上需要增加额外的开销，这也是多数 PCI 设备不支持预读机制的原因。

尽管如此本节仍需要对 HOST 处理器预读 PCI 设备进行探讨。假设在图 3-12 所示的处理器系统中，HOST 主桥和 PCI 桥 A 不支持预读，而 PCI 桥 B 支持预读，而且处理器的 Cache 行长度为 32B (0x20)。

如果 HOST 处理器对 PCI 设备的 0x8000-0000 ~ 0x8000-0003 这段地址空间进行读操作时。HOST 主桥将使用存储器读总线事务读取 PCI 设备的“0x8000-0000 ~ 0x8000-0003 这段地址空间”，这个存储器读请求首先到达 PCI 桥 A，并由 PCI 桥 A 转发给 PCI 桥 B。

PCI 桥 B 发现“0x8000-0000 ~ 0x8000-0003 这段地址空间”属于自己的可预读存储器区域，即该地址区域在该桥的 Prefetchable Memory Base 定义的范围内，则将该存储器读请求转换为 MRL 总线事务，并使用该总线事务从 PCI 设备^①中读取 0x8000-0000 ~ 0x8000-001F 这段数据，并将该数据存放到 PCI 桥 B 的预读缓冲中。MRL 总线事务将从需要访问的 PCI 设备的起始地址开始，一直读到当前 Cache 行边界。

① 此时 PCI 设备的这段区域一定是可预读的存储器区域。

之后当 HOST 处理器读取 0x8000-0004 ~ 0x8000-001F 这段 PCI 总线地址空间的数据时，将从 PCI 桥 B 的预读缓冲中直接获取数据，而不必对 PCI 设备进行读取。

2. PCI 设备读取存储器

PCI 设备预读存储器地址空间时，需要使用 MRL 或者 MRM 总线事务。与 MRL 总线周期不同，MRM 总线事务将从需要访问的存储器起始地址开始，一直读到下一个 Cache 行边界为止。

对于一个 Cache 行长度为 32B (0x20) 的处理器系统，如果一个 PCI 设备对主存储器的 0x1000-0000 ~ 0x1000-0007 这段存储器地址空间进行读操作时，由于这段空间没有跨越 Cache 行边界，此时 PCI 设备将使用 MRL 总线事务对 0x1000-0000 ~ 0x1000-001F 这段地址区域发起存储器读请求。

如果一个 PCI 设备对主存储器的 0x1000-001C ~ 0x1000-0024 这段存储器地址空间进行读操作时，由于这段空间跨越了 Cache 行边界，此时 PCI 设备将使用 MRM 总线事务对 0x1000-001C ~ 0x1000-002F 这段地址空间发起存储器读请求。

在图 3-12 所示的例子中，PCI 设备读取 0x1000-001C ~ 0x1000-0024 这段存储器地址空间时，首先将使用 MRM 总线事务发起读请求，该请求将通过 PCI 桥 B 和 A 最终到达 HOST 主桥。HOST 主桥将从主存储器中读取 0x1000-001C ~ 0x1000-002F 这段地址空间的数据^①。如果 PCI 桥 A 也支持下游总线到上游总线的预读，这段数据将传递给 PCI 桥 A；如果 PCI 桥 A 和 B 都支持这种预读，这段数据将到达 PCI 桥 B 的预读缓冲。

如果 PCI 桥 A 和 B 都不支持预读，0x1000-0024 ~ 0x1000-002F 这段数据将缓存在 HOST 主桥中，HOST 主桥仅将 0x1000-001C ~ 0x1000-0024 这段数据通过 PCI 桥 A 和 B 传递给 PCI 设备。之后当 PCI 设备需要读取 0x1000-0024 ~ 0x1000-002F 这段数据时，该设备将根据不同情况，从 HOST 主桥、PCI 桥 A 或者 B 中获取数据而不必读取主存储器。值得注意的是，PCI 设备在完成一次数据传送后，暂存在 HOST 主桥中的预读数据将被清除。PCI 设备采用这种预读方式，可以极大提高访问主存储器的效率。

PCI 总线规范有一个缺陷，就是目标设备并不知道源设备究竟需要读取或者写入多少个数据。例如 PCI 设备使用 DMA 读方式从存储器中读取 4 KB 大小的数据时，只能通过 PCI 突发读方式，一次读取一个或者多个 Cache 行。

假定 PCI 总线一次突发读写只能读取 32 B 大小的数据，此时 PCI 设备读取 4KB 大小的数据，需要使用 128 次突发周期才能完成全部数据传送。而 HOST 主桥只能一个一个地处理这些突发传送，从而存储器控制器并不能准确预知何时 PCI 设备将停止读取数据。在这种情况下，合理地使用预读机制可以有效地提高 PCI 总线的数据传送效率。

我们首先假定 PCI 设备一次只能读取一个 Cache 行大小的数据，然后释放总线，之后再读取一个 Cache 行大小的数据。如果使用预读机制，虽然 PCI 设备在一个总线周期内只能获得一个 Cache 行大小的数据，但是 HOST 主桥仍然可以从存储器获得 2 个 Cache 行以上的数据，并将这个数据暂存在 HOST 主桥的缓冲中，之后 PCI 设备再发起突发周期时，HOST 主桥可以不从存储器，而是从缓冲中直接将数据传递给 PCI 设备，从而降低了 PCI 设备对存储

^① 假设 HOST 主桥读取存储器时支持预读，多数 HOST 主桥都支持这种预读。

器访问的次数，提高了整个处理器系统的效率。

以上描述仅是实现 PCI 总线预读的一个例子，而且仅仅是理论上的探讨。实际上绝大多数半导体厂商都没有公开 HOST 主桥预读存储器系统的细节，在多数处理器中，HOST 主桥以 Cache 行为单位读取主存储器的内容，而且为了支持 PCI 设备的预读功能 HOST 主桥需要设置必要的缓冲部件，这些缓冲的管理策略较为复杂。

目前 PCI 总线已经逐渐退出历史舞台，进一步深入研究 PCI 桥和 HOST 主桥，意义并不太大，不过读者依然可以通过学习 PCI 体系结构，获得处理器系统中有关外部设备的必要知识，并以此为基础，学习 PCIe 体系结构。

3.5 小结

本章重点介绍了 PCI 总线的数据交换。其中最重要的内容是与 Cache 相关的 PCI 总线事务和预读机制。虽然与 Cache 相关的 PCI 总线事务并不多见，但是理解这些内容对于理解 PCI 和处理器体系结构，非常重要。

第 I 篇的主体是以 PCI 总线为例，说明一个局部总线在处理器系统中的作用，这也是笔者写作本书的初衷。PCI 总线作为一个局部总线，在设计思路上，与其他局部总线并没有本质的不同。在本篇中，最重要的内容是局部总线的设计与实现方法，希望读者阅读本书时，不要仅仅将目光锁定在 PCI 总线本身。

本书的第 II 篇内容与第 I 篇密切相关，希望读者在真正理解第 I 篇内容的基础上阅读第 II 篇。PCIe 总线在继承 PCI 总线部分内容的基础上做出了许多重大调整。但是从处理器体系结构的角度来看，PCIe 总线依然是局部总线，这条局部总线与 PCI 总线以及其他平台的局部总线相比，并不存在本质的不同。而理解这些局部总线的关键，仍然是深入理解处理器的体系结构。