计算机科学与技术系列教材

计 算 机 系 统 结 构

高　辉　张玉萍　编著

武 汉 大 学 出 版 社

计算机科学与技术系列教材

编　委　会

图书在版编目(CIP)数据

计算机系统结构/高辉，张玉萍编著. —武汉：武汉大学出版社，2004.8
(计算机科学与技术系列教材)
ISBN 7-307-04246-0

Ⅰ.计… Ⅱ.①高… ②张… Ⅲ.计算机体系结构—高等学校—教材 Ⅳ.TP303

中国版本图书馆 CIP 数据核字(2004)第 047914 号

责任编辑：黄金文　　责任校对：程小宜　　版式设计：支　笛

出版发行：武汉大学出版社　(430072　武昌　珞珈山)
(电子邮件：wdp4@whu.edu.cn　网址：www.wdp.whu.edu.cn)
印刷：湖北省通山县印刷厂
开本：787×980　1/16　　印张：27　　字数：550 千字
版次：2004 年 8 月第 1 版　　2004 年 8 月第 1 次印刷
ISBN 7-307-04246-0/TP・152　　定价：39.00 元

前　言

本书作者已经多年且仍然正在从事计算机专业之“计算机系统结构”课程教学工作，这本教材是在多年教学经验的基础上，结合教学科研成果以及多种计算机课程参考书综合编著而成的。

当今的计算机系统是一个硬件和软件相结合的一个复杂系统。要全面地、正确地了解和掌握计算机系统，必须了解和掌握计算机软、硬件交界面及其上、下功能分配。本课程是在学习完《计算机组成原理》、《高级语言程序设计》、《数字逻辑》、《计算机操作系统》等专业基础课之后，再进一步学习计算机专业知识而设置的。为了使读者尽快掌握这门知识，本教材在编写上具有如下两个显著特点：一是深入浅出地介绍计算机系统的基本概念、基本原理、基本结构和基本分析方法。语言表达尽量通俗易懂，概念明确，关键之处以例题说明，达到提高学习效率的目的；二是结合当今流行的微型计算机体系结构的发展。以前只能在大、中型计算机上实现的关键技术，目前已经部分“下移”至微型计算机上。针对这一特点，教材中已在各章后部介绍了 Pentium 微处理器的体系结构的知识与特点，使读者能够有“理论联系实际”之新感受。

本教材共分为 7 章。第一章介绍计算机系统的基本概念。包括主计算机系统结构的定义，计算机系统的分类、设计方法、性能评价以及 Pentium 微处理器的体系结构。第二章介绍数据表示与指令系统。包括基本的数据表示和高级数据表示，指令的寻址方式，指令格式的优化设计和指令设计的两种风格，DLX 指令集结构。第三章介绍输入输出系统。包括输入输出原理，总线设计，中断系统，Pentium 系列微计算机的中断系统与 APIC 技术，I/O 通道与 I/O 处理机。第四章介绍存储系统。包括存储系统的基本原理，虚拟存储技术，Cache 存储器技术，Pentium 微处理器中虚拟存储技术和 Cache 存储器。第五章介绍流水和向量处理技术。包括标量流水技术，标量流水线的性能分析，标量流水中的障碍及控制，标量非线性流水线的调度，超标量与超流水技术，Pentium 微处理器中流水技术，指令的动态执行技术，向量流水处理机。第六章介绍并行处理技术。包括并行处理的基本概念，SIMD 计算机阵列基本结构，SIMD 计算机的互连网络，SIMD 计算机举例，多处理机系统。第八章介绍几种新型计算机系统。

本教材第二、三章由张玉萍编写，其他各章由高辉编写，全书由高辉统稿。

由于作者水平有限，难免有错漏之处，敬请读者批评指正。

为了帮助广大读者更好地掌握这门知识，我们随后将推出与本教材配套的《计算机系统结构学习辅导及习题解答》。

在本教材的编写过程中，得到了武汉大学教务部和武汉大学出版社的大力帮助，在此表示衷心地感谢。

作　　者

2004 年于武汉大学

目　录

第一章 计算机系统结构的设计基础

世界上第一台通用电子数字计算机 ENIAC(Electronic Numerical Integrator And Computer),于 1946 年在美国宾夕法尼亚大学建成。由于 ENIAC 输入和更换程序特别繁杂,ENIAC 课题组的顾问、著名数学家冯·诺依曼提出将程序的指令与指令所操作的数据一起存于存储器的概念。这个著名的存储式程序(stored-program)概念,成为计算机工作的基本机理。这一概念也被图灵大约同时期提出。世界上第一台存储式程序计算机是 1949 年在英国剑桥大学建成的 EDSAC(Electronic Delay Storage Automatic Calculator)计算机,它使用 3 000 只电子管,每秒钟能完成 700 次加法运算。1953 年 IBM 公司制造出第一台电子存储式程序的商用计算机。

50 多年来,计算机系统性能得到了大幅度地提高,价格却大幅度地下降。计算机的发展已经历了四次更新换代,现在正处于第五代。依据半导体技术发展水平,这五代划分应该是:

第一代:1945～1954 年,电子管和继电器;

第二代:1955～1964 年,晶体管和磁芯存储器;

第三代:1965～1974 年,中、小规模集成电路(MSI-SSl);

第四代:1975～1990 年,LSI/ VLSI 和半导体存储器;

第五代:1990 年至今,ULSI/ GSI(Giga-Scale Integration)巨大规模集成电路。

计算机系统更新换代的标志主要是两个方面:一是由于器件不断发展,使计算机系统的工作速度、功能、集成度和可靠性等指标不断提高和价格不断降低而造成的;二是得益于计算机系统结构的改进。有人统计在 1965 年到 1975 年期间,计算机系统性能提高近 100 倍,其中由于器件性能提高使其性能增加 10 倍,而另外的 10 倍,主要归功于系统结构改进。在 50 多年的发展进程中,器件在技术上的改进是比较稳定的,而系统结构的改进则有较大起伏。特别是近 20 年来,由于计算机系统的设计对集成电路技术的依赖性大为增加,从而使得在这一期间内,各类不同的计算机系统的性能增长率有了差异。

巨型机的性能增长得益于器件技术和系统结构两方面的改进;大型机的性能增长则主要靠器件工艺上的改进,因为系统结构方面的改进没有新的突破;小型机的发展,一方面是由于计算机实现方法有了较大的改进,另一方面是因为采用了许多大型机中行之有效的先进技术。然而这三类计算机,在 1970 年到 1990 年期间,每年计算机的性能平均增长率均只在 18% 左右。与之形成明显对照的是微型计算机的性能

增长非常快,每年的平均增长率约为 35%,这是因为微型计算机能从集成电路技术的进展中得到最为直接的好处。自 20 世纪 80 年代起,微处理机技术实际上已成为新系统结构和老系统结构更新时所选用的主要技术。

自 1985 年开始,一种具有新颖设计风格的系统结构,即 RISC 技术的系统结构,为计算机工业界所青睐。它将集成电路技术进展、编译技术改进和新的系统结构设计思想三者有机地结合起来,从而使得这种风格设计的计算机系统的性能以每年增长一倍的高速率加以改进。应该指出的是,这种改进的基础是通过对以往计算机如何被使用的模拟实验数据进行定量分析后获得的。有的学者将这种设计风格称为定量分析的计算机系统结构设计风格,显然这比传统的定性设计风格要精确得多,开发 RISC 技术的两位先驱者,美国加州大学伯克莱分校的 D. Patterson 教授和斯坦福大学的 J .Hennessy 教授是这种定量分析设计方法的主要倡导者。

1 .1 计算机系统结构的基本概念

1. 1. 1 计算机系统的层次结构

现代通用计算机系统是由硬件和软件组成的一个复杂系统,按其功能可划分为多级层次结构,如图 1 .1 所示。层次结构由上往下依次为应用语言机器级、高级语言机器级、汇编语言机器级、操作系统机器级、传统机器级和微程序机器级。对于一个具体的计算机系统,层次的多少会有所不同。

各机器级的实现主要靠翻译或解释,或者是这两者的结合。翻译(Translation)是先用转换程序将上一级机器级上的程序整个地变换成下一级机器级上可运行的等效程序,然后再在下一级机器级上去实现的技术。解释(Interpretation)则是在下一级机器级上用它的一串语句或指令来仿真上一级机器级上的一条语句或指令的功能,通过对上一级机器语言程序中的每条语句或指令逐条解释来实现的技术。

应用语言虚拟机器级是为了满足信息管理、人工智能、图像处理、辅助设计等专门的应用来设计的。使用面向某种应用环境的应用语言(L5)编写的程序一般是经应用程序包翻译成高级语言(L4)程序后,再逐级向下实现的。高级语言机器级上的程序可以先用编译程序整个地翻译成汇编语言(L3)程序或机器语言(L1)程序,再逐级或越级向下实现,也可以用汇编语言(L3)程序、机器语言(L1)程序,甚至微指令语言(L0)程序解释实现。对汇编语言(L3)源程序则先用汇编程序整个将其变换成等效的二进制机器语言(L1)目标程序,再在传统机器级上实现。操作系统程序虽然已发展成用高级语言(如面向编写操作系统软件的 C 语言)编写,但最终还是要用机器语言程序或微指令程序来解释的。它提供了传统机器级所没有,但为汇编语言和高级语言使用和实现所用的基本操作、命令和数据结构,例如,文件管理、存储管理、进程管理、多道程序共行、多重处理、作业控制等所用到的操作命令、语句和数据结构

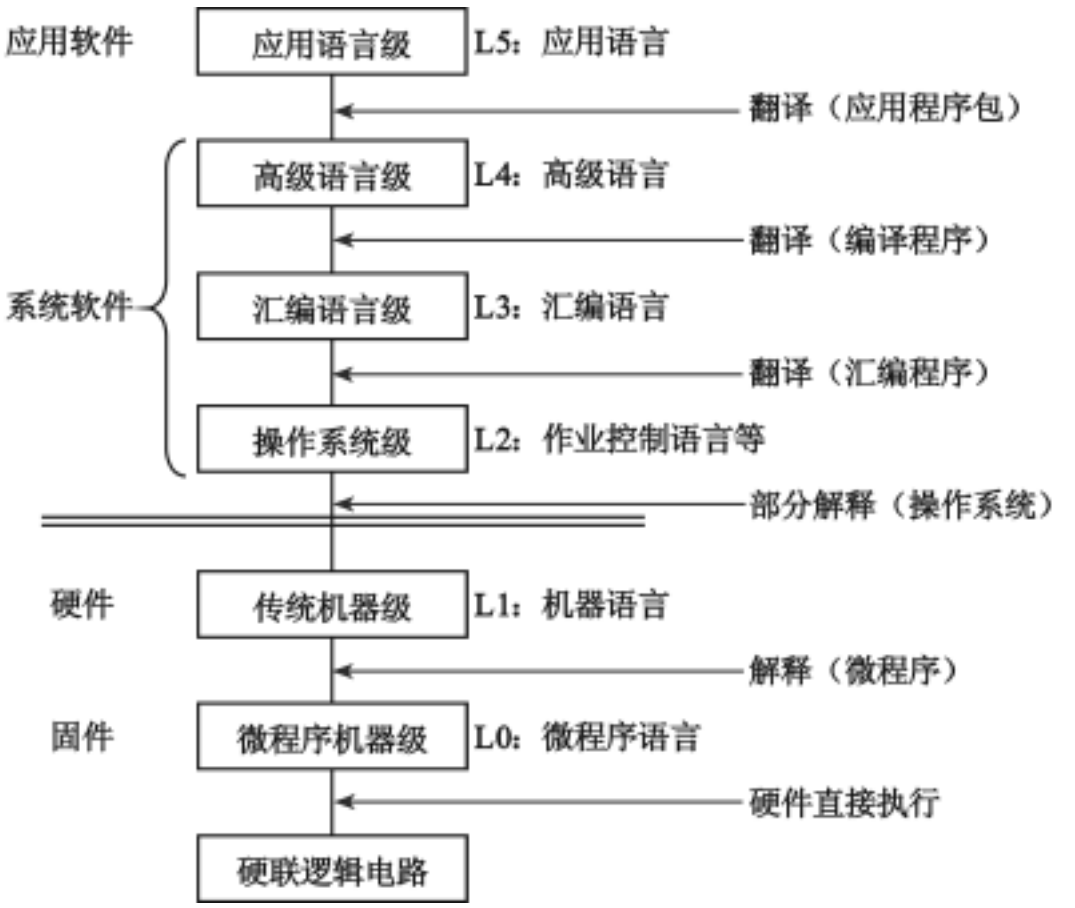


图 1.1 计算机系统的多级层次结构

等。因此,操作系统机器级放在传统机器级和汇编语言机器级之间是适宜的。传统机器级采用组合逻辑电路控制,其指令可直接用硬件来实现,也可以采用微程序控制,用微指令(L0)程序来解释实现。微指令直接控制硬件电路的动作。

就目前的状况来看,第 0 级用硬件实现,第 1 级用微程序(固件)实现,第 2 级到第 5 级大多用软件实现。我们称以软件为主实现的机器为虚拟机器,以区别于由硬件或固件实现的实际机器。虚拟机器不一定全都由软件实现,有些操作可以用固件或硬件实现。如操作系统中的某些命令可由比它低两级的微程序解释,或全部用硬件来实现。高级语言机器则直接用微程序解释或用硬件实现,没有编译软件。

计算机系统结构作为一门学科,主要研究软件、硬件功能分配和对软件、硬件界面的确定,即哪些功能由软件完成,哪些功能由硬件完成。采用何种实现方式,要从整个计算机系统的效率、速度、价格、资源状况等方面考虑,对软件、硬件、固件取舍进行综合平衡。软件和硬件在逻辑功能上是等效的。原理上,软件实现的功能完全可以用硬件或固件完成,硬件实现的功能也可以由软件的模拟来完成,只是其性能、价格、实现的难易程度等有所不同。具有相同功能的计算机系统,其软、硬件功能分配比例可以在很宽的范围内变化。

1.1.2 计算机系统结构

“计算机系统结构”这个名词来源于英文 Computer Architecture,也有译成“计

算机体系结构”的。architecture 这个词原来用于建筑领域,其意义是“建筑学”、“建筑物的设计或式样”,它是指一个系统的外貌。20 世纪 60 年代这个名词被引入计算机领域,“计算机系统结构”一词已经得到普遍应用,它研究的内容不但涉及计算机硬件,也涉及计算机软件,已成为一门学科。但对“计算机系统结构”一词的含义仍有多种说法,并无统一的定义。

计算机系统结构这个词是 Amdahl 等人在 1964 年提出的。他们把系统结构定义为由程序设计者所看到的一个计算机系统的属性,即概念性结构和功能特性。这实际上是计算机系统的外特性。按照计算机层次结构,不同程序设计者所看到的计算机有不同的属性。例如,对于使用 FORTRAN 高级语言程序员来讲,一台 IBM3090 大型机、一台 VAX11/ 780 小型机或一台 PC 微型机,看起来都是一样的,因为在这三台计算机上运行他所编制的程序,所得到的结果是一样的。但对于使用汇编语言程序的程序员来讲,由于这三台机器的汇编语言指令完全不一样,他所面对的计算机的属性就也会不一样。另外,即使对同一台机器来讲,处在不同级别的程序员,例如应用程序员、高级语言程序员、系统程序员和汇编程序员,他们所看到的计算机外特性也是完全不一样的。那么通常所讲的计算机系统结构的外特性应是处在哪一级的程序员所看到的外特性呢?比较一致的看法是机器语言程序员或编译程序编写者所看到的外特性,这种外特性是指由他们所看到的计算机基本属性,即计算机的概念性结构和功能特性,这是机器语言程序员或编译程序编写者为使其所编写、设计或生成的程序能在机器上正确运行所必须遵循的。由机器语言程序员或编译程序编写者所看到的计算机的基本属性是指传统机器级的系统结构,在传统机器级之上的功能被视为属于软件功能,而在其之下的则属于硬件和固件功能。因此,计算机系统的概念性结构和功能属性实际上是计算机系统中软、硬件之间的界面。

在计算机技术中,一种本来是存在的事物或属性,但从某种角度看似乎不存在,称为透明性现象。通常,在一个计算机系统中,低层机器级的概念性结构和功能特性,对高级语言程序员来说是透明的。由此看出,在层次结构的各个级上都有它的系统结构。

就目前的通用机来说,计算机系统结构的属性应包括以下几个方面:

- 硬件能直接识别和处理的数据类型和格式等的数据表示;
- 最小可寻址单位、寻址种类、地址计算等的寻址方式;
- 通用/ 专用寄存器的设置、数量、字长、使用约定等的寄存器组织;
- 二进制或汇编级指令的操作类型、格式、排序方式、控制机构等的指令系统;
- 内存的最小编址单位、编址方式、容量、最大可编址空间等的存储系统组织;
- 中断的分类与分级、中断处理程序功能及入口地址等的中断机构;
- 系统机器级的管态和用户态的定义和切换;
- 输出设备的连接、使用方式、流量、操作结束、出错指示等的机器级 I/ O 结构;
- 各部分的信息保护方式和保护机构等。

1.1.3 计算机组成与实现

计算机组成(Computer Organization)指的是计算机系统结构的逻辑实现,也常称为计算机组织。它包括机器级内的数据流和控制流的组成以及逻辑设计等。它着眼于机器级内各事件的排序方式与控制机构、各部件的功能及各部件间的联系。计算机组成设计要解决的问题是在所希望达到的性能和价格下,怎样最佳、最合理地把各种设备和部件组织成计算机,以实现所确定的系统结构。计算机组成设计主要是围绕提高速度,着重从提高操作的并行度、重叠度以及分散功能和设置专用功能部件来进行的。

计算机组成设计要确定的方面一般应包括:

- 数据通路宽度(在数据总线上一次并行传送的信息位数多少)。
- 专用部件的设置(设置哪些专用部件,如乘除法专用部件、浮点运算部件、字符处理部件、地址运算部件等,每种专用部件设置的数量,这些都与机器所需达到的速度、专用部件的使用频度高低及允许的价格等有关)。
- 各种操作对部件的共享程度(共享程度高,即使操作在逻辑上不相关也只能分时使用,限制了速度,但价格便宜,可以设置多个部件降低共享程度,提高操作并行度来提高速度,但价格也将提高)。
- 功能部件的并行度(功能部件的控制和处理方式是采用顺序串行,还是采用重叠、流水或分布处理)。
- 控制机构的组成方式(事件、操作的排序机构是采用硬联控制还是用微程序控制,是采用单机处理还是用多机处理或功能分布处理)。
- 缓冲和排队技术(在不同部件之间怎样设置及设置多大容量的缓冲器来弥补它们的速度差异;是采用随机方式,还是先进先出、先进后出、优先级或循环方式来安排等待处理事件的先后顺序)。
- 预估、预判技术(为优化性能和优化处理,采用什么原则来预测未来的行为)。
- 可靠性技术(采用什么样的冗余技术和容错技术来提高可靠性)。

计算机实现(Computer Implementation)指的是计算机组成的物理实现。它包括处理机、主存等部件的物理结构,器件的集成度、速度和信号,器件、模块、插件、底板的划分与连接,专用器件的设计,电源、冷却、装配等技术。

计算机系统结构、计算机组成和计算机实现是三个不同的概念。计算机系统结构是指令系统及其执行模型;计算机组成是计算机系统结构的逻辑实现;计算机实现是计算机组成的物理实现。它们各自包含不同的内容和采用不同的技术,但又有紧密的联系。在学习和理解时有两点需要注意:

一是计算机系统结构、组成和实现之间的界限变得越来越模糊了。尤其是严格区分计算机系统结构和组成已不太可能,也没有太大的实际意义。随着 VLSI 技术的进步,新器件的不断涌现,当今计算机系统结构的设计所面临的问题与 Amdahl 所

处的时期大不相同,就是与 10 年前也大不相同。例如,10 年前系统配置几十至几百 KB 的内存就很不错了,某些指令系统的设计中甚至有对存储器操作数直接进行加减的指令,不惜牺牲执行速度来珍惜宝贵、有限的内存资源。现在,存储器芯片的集成度大幅度提高而价格急剧下降,内存容量已不是计算机系统结构设计的主要问题了;如何组织存储器以提高存取速度,如何保证 CPU—内存之间的通道不致成为系统性能的瓶颈,是当代计算机系统结构设计必须考虑的问题。现在,一般已将功能模块设计移入计算机系统结构考察的范畴之内。

二是我们介绍了计算机系统结构、组成和实现三者之间的关系,但不要认为计算机系统结构设计就是硬件设计,两者不能混淆。操作系统、编译程序以及高级语言的发展都对计算机系统结构的设计有重要影响。计算机系统结构设计是在功能这一层次上考虑问题,当然要涉及到硬件,但它不是只包括硬件设计。例如,存储器管理功能可以由硬件和软件共同实现,它们之间的分工取决于当前硬件和软件的可用性、性能和价格。在 VLSI 发展的初期,存储器管理功能一般由软件实现;现在,存储器控制芯片已能实现这些存储器管理算法并维护存储器与高速缓存的一致性。因此,计算机系统结构设计的一个主要任务是研究软件、硬件功能分配和对软件、硬件界面的确定。

1.1.4 计算机系统结构的分类

研究计算机系统的分类方法有助于认识计算机系统结构的组成和特点,理解计算机系统的工作原理和性能。常用的计算机系统结构分类方法有三种:

1 .Flynn 分类法

1966 年 M J .Flynn 提出了按照指令流和数据流的多倍性概念进行分类的方法,其定义如下:

指令流(instruction stream)——机器执行的指令序列。

数据流(data stream)——由指令流调用的数据序列,包括输入数据和中间结果。

多倍性(multiplicity)——在系统最受限制的部件(瓶颈)上同时处于同一执行阶段的可并行执行的指令或数据的最大可能个数。

按照指令流和数据流的不同组织方式,把计算机系统的结构分为以下四类:

① 单指令流单数据流 SISD(Single Instruction stream Single Data stream);

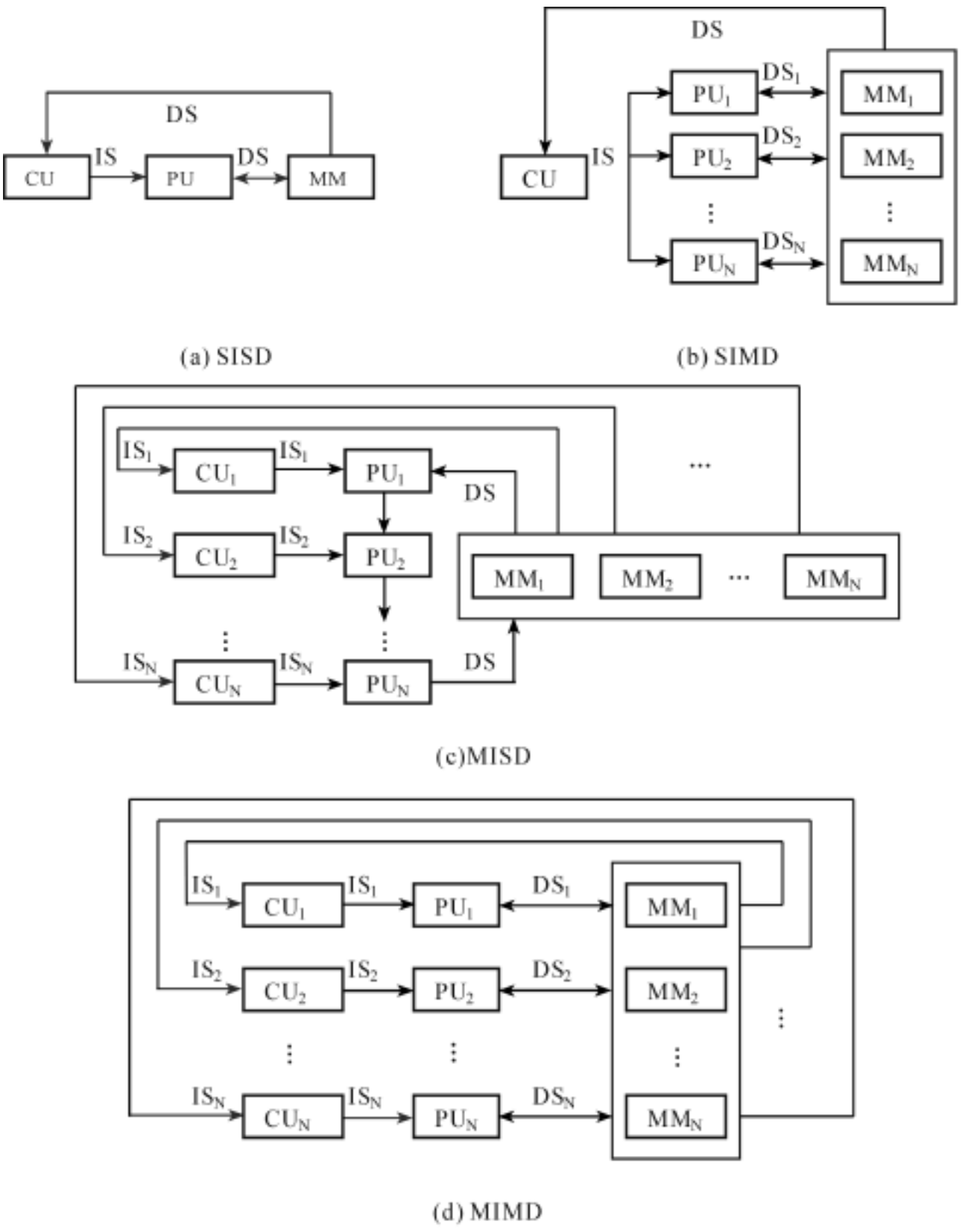
② 单指令流多数据流 SIMD(Single Instruction stream Multiple Data stream);

③ 多指令流单数据流 MISD(Muhiple Instruction stream Single Data stream);

④ 多指令流多数据流 MIMD(Muhiple Instruction stream Multiple Data stream)。

对应于这四类计算机的基本结构框图,如图 1 .2 所示。SISD 是传统的顺序处理计算机。SIMD 以阵列处理机或并行处理机为代表。MISD 在实际上代表何种计算

机,也存在着不同的看法,有的文献把流水线结构机器看做是 MISD 结构,多处理机属于 MIMD 结构。



CU:控制部件　PU:处理部件　MM:存储模块　IS:指令流　DS:数据流

图 1.2　Flynn 分类法各机器结构

2 .冯氏分类法

1972 年美籍华人冯泽云(Tse-yun Feng)教授提出了用最大并行度(Pm)来定量描述各种计算机系统特性的冯氏分类法。最大并行度 Pm 的定义为:计算机系统在单位时间内能够处理的最大的二进制位数。

图 1 .3 描述了用最大并行度对计算机系统分类的方法。用平面直角坐标系中的一个点代表一个计算机系统,横坐标代表字宽(n 位),即在一个字中同时处理的二进制位数;纵坐标代表位片宽度(m 位),即在一个位片中能同时处理的字数。

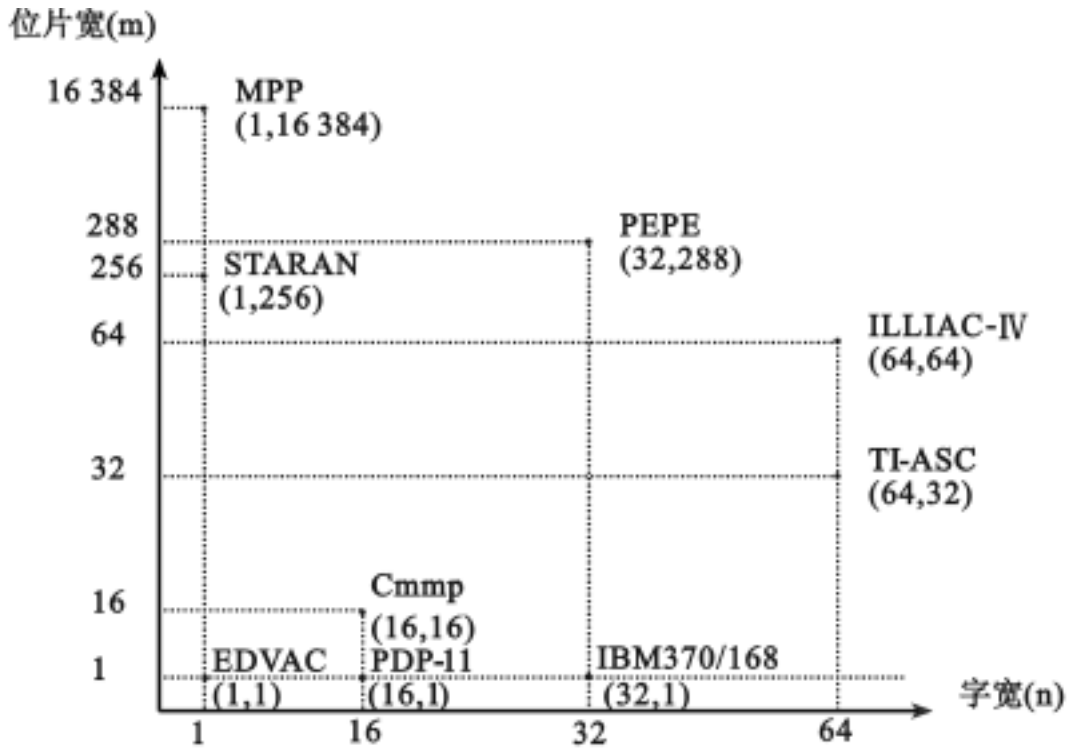


图 1 .3　按最大并行度的冯氏分类法

由图 1 .3 可得出四类不同处理方法的计算机系统结构:

①字串位串 WSBS(Word Serial and Bit Serial),其中 n = l,m = 1。这是第一代计算机发展初期的纯串行计算机。

②字并位串 WPBS(Word Parallel and Bit Serial),其中 n > 1,m = 1。这是传统并行单处理机。

③字串位并 WSBP(Word Serial and Bit Parallel),其中 n = 1,m > l。STARAN,MPP,DAP 属于这种结构。

④字并位并 WPBP(Word Parallel and Bit Parallel),其中 n > 1,m > 1。PEPE,ILLIAC Ⅳ,Cmmp 属于这种结构。

3 .Handler 分类法

1977 年 Wolfgan Handler 根据并行度和流水线提出了另一种分类法。这种分类方法把计算机的硬件结构分成三个层次,并分别考虑它们的可并行 - 流水处理程度。这三个层次是:

① 程序控制部件(PCU)的个数 k;

② 算术逻辑部件(ALU)或处理部件(PE)的个数 d;

③ 每个算术逻辑部件包含基本逻辑线路(ELC)的套数 w。

这样我们可以把一个计算机系统的结构用如下公式表示:t(系统型号) = (k, d, w)。为了进一步揭示流水线的特殊性,一个计算机系统的结构可用如下公式表示:t(系统型号) = (k×k′,d×d′,w×w′),其中:k′表示宏流水线中程序控制部件的个数,d′表示指令流水线中算术逻辑部件的个数,w′表示操作流水线中基本逻辑线路的套数。

例如,Crayl 有 1 个 CPU,12 个相当于 ALU 或 PE 的处理部件,可以最多实现 8 级流水线。字长为 64 位,可以实现 l～14 位流水线处理。所以 Crayl 的系统结构可表示为:

t(Crayl) = (1,12×8,64(1～14))

下面是用这种分类法的例子:

t(PDP11) = (1,l,16)

t(ILLIAC Ⅳ) = (1,64,64)

t(STARAN) = (1,8192,1)

t(Cmmp) = (16,1,16)

t(PEPE) = (1×3,288,32)

t(TIASC) = (1,4,64×8)

1.2 计算机系统设计技术

1.2.1 计算机系统设计原理

下面介绍计算机系统设计中经常用到的几个定量原理。

1 .加快经常性事件的速度(Make the Common Case Fast)

这是计算机设计中最重要也最广泛采用的设计准则。使经常性事件的处理速度加快能明显提高整个系统的性能。一般说来,经常性事件的处理比较简单,因此比不经常出现的事件处理要快。例如,在 CPU 中两个数进行相加运算时,相加结果可能出现溢出现象,也可能无溢出发生,显然经常出现的事件是不发生溢出的情况,而溢

出是偶然发生的事件。因此,在设计时应优化不发生溢出的情况,使这个经常性事件的处理速度尽可能快,而对溢出处理则不必过多考虑优化。因为发生溢出的概率很小,即使发生了,处理得慢一些也不会对系统性能产生很大的影响。

在计算机设计中经常会遇到上述情况,那么,如何确定经常性事件以及如何加快处理它,这就是下面介绍的 Amdahl 定律要解决的问题。

2 .Amdahl 定律

Amdahl 定律:系统中对某一部件采用某种更快的执行方式后整个系统性能的改进程度,取决于这种执行方式被使用的频率,或所占总执行时间的比例。

$$\text{系统加速比} = \frac{\text{系统改进后的性能}}{\text{系统改进前的性能}} = \frac{\text{系统改进前执行某一任务的总时间}}{\text{系统改进后执行同一任务的总时间}}$$

Amdahl 定律定义了采取改进(加速)某部分功能后可获得的性能改进或执行时间的加速比。系统加速比取决于两个因素:

① 可改进部分在原系统计算时间中所占的比例。例如,一个需运行 60s 的程序中有 20s 的运算可以加速,那么该比例就是 20/ 60。这个值用“ 可改进比例 ”表示,它总是小于或等于 1 的。

② 可改进部分改进以后的性能提高。例如,系统改进后执行程序,其中可改进部分花费的时间为 2s,而改进前该部分需花费的时间为 5s,则性能提高为 5/ 2。用“ 部件加速比 ”表示性能提高比,一般情况下它是大于 1 的。

若以 T_o 和 T_e 分别表示采取某种改进措施前后完成同一任务所需的总时间,用 f_e 表示可改进部分的比例($0 \leq f_e \leq 1$),用 r_e 表示采用改进措施比不采用改进措施可加快执行的倍数,则上述各参量与系统加速比(S_p)之间的关系可用如下表达式表示:

$$T_o = T_e \times (1 - f_e) + \frac{f_e \times T_e}{r_e} = [(1 - f_e) + \frac{f_e}{r_e}] \times T_e$$

(部件改进后,系统的总执行时间等于不可改进部分的执行时间加上可改进部分改进后的执行时间)

$$S_p = \frac{T_e}{T_o} = \frac{1}{(1 - f_e) + f_e / r_e}$$

例 1. 1　若考虑将系统中的某一功能的处理速度加快到 10 倍,该功能部件的原处理时间为整个系统运行时间的 40 %,则采用改进措施后,使整个系统的性能提高了多少 ?

解:由题意可知: $f_e = 0.4$, $r_e = 10$,由加速比公式可得

$$S_p = \frac{1}{(1 - 0.4) + 0.4 / 10} = 1.56$$

3 .程序访问的局部性规律

所谓程序访问的局部性是指程序执行中,呈现出频繁重新使用那些最近已被使

用过的数据和指令的规律。统计表明一个程序执行时间中的 90% 是花费在 10% 程序代码上。例如对 Gcc(GnuC 编译)、Spice(CAD 电路分析软件)以及 Tex(文本处理软件)三个典型测试程序的测试表明,相应的比例数分别为 13%,9.5% 和 9.3%,遵从上述的 90%/ 10% 定量规律。

程序访问局部性主要反映在时间和空间局部性两个方面,时间局部性是指程序中近期被访的信息项很可能马上将被再次访问;空间局部性是指那些在访问地址上相邻近的信息项很可能会被一起访问。

程序访问局部性规律是按层次构成存储体系的主要依据,程序中的大部分可放在容量较大,工作速度较慢及成本较低的慢速存储部件,而只需将其中的一小部分(如 10%～20%)存放在高速的存储部件中,这便是虚存、Cache 高速缓存得以实现的具体根据。

1.2.2 计算机系统设计的方法

1 .软硬件舍取的基本原则

计算机系统结构设计的一个主要任务是进行软、硬件功能分配。一般来说,提高硬件功能的比例可提高解题速度,减少程序所需存储空间,但会提高硬件成本,降低硬件利用率和计算机系统的灵活性、适应性;而提高软件功能比例可降低硬件成本,提高系统的灵活性、适应性,但解题速度下降,软件设计费用和所需存储空间增加。下面介绍软、硬件舍取的三个基本原则。

(1)在现有硬件和器件(主要是逻辑器件和存储器件)条件下,系统要有高的性价比。主要从实现费用、速度和其他性能要求来综合平衡;

(2)系统结构和组成的设计尽可能地不要过多或不合理地限制各种组成、实现技术的采用;

(3)不仅从“ 硬 ”的角度考虑如何应用组成技术的成果和便于发挥器件技术的进展,还应从“ 软 ”的角度把如何为编译和操作系统的实现以及为高级语言程序的设计提供更多更好的硬件支持。

2 .计算机系统设计者的主要任务

计算机系统的设计,不仅应该关心系统结构所包含的指令系统设计,还应考虑功能的组成、逻辑设计以及具体的物理实现,它包括了集成电路的设计、封装、电源以及冷却等。

计算机系统设计者的主要任务是:

① 要满足用户对功能上的要求以及相应的对价格和性能的要求。这里功能上的要求包括:应用领域,如专用还是通用,科学计算或是商业上使用;软件兼容级别,在程序设计语言级或是目标代码(二进制)级兼容;操作系统需求,如地址空间的大

小、存储器管理、保护及环境转换;中断和自陷以及对标准的要求,如浮点标准、I/O总线标准、操作系统标准、网络标准及程序设计语言标准等。

② 在满足功能要求基础上,进行设计的优化。优化主要是以性能价格比为衡量指标,仔细考虑某一所需的功能应该用软件还是用硬件来实现。因此在系统设计中如何很好地平衡软、硬件两者就是特别关键的了。此外在选择硬件或软件的方法时,还应考虑设计复杂性和具体实现的难易程度。

③ 设计应能适应日后发展趋势。这是指好的系统结构设计应能经受硬件技术、软件技术以及应用需求特征的变化。因此设计者应对计算机使用趋势和计算机技术的发展趋势有足够的认识。

硬件技术进展表现在IC(集成电路)逻辑技术、DRAM技术及Disk(硬盘)技术。统计资料表明:一个芯片上的晶体管数大约每年增加25%,因此每三年可增加1倍;器件的开关速度增长基本类似DRAM的密度每年增长约60%,因此每三年将增长3倍;访问存储器周期改进相应较慢,每十年约减少1/3;硬盘密度每年增加25%,每三年增加1倍,访问时间则每十年减少1/3。

软件技术的发展趋向:一是程序所要求的存储器空间增长,大约每年增长1.5～2倍;因此相应地要求地址位每年能增长0.5位到1位;这两点对设计非常重要,即必须留有足够的地址空间以待扩展。二是汇编语言将逐步为高级语言所替代,编译技术将起更大作用,因此系统结构应能更好地支持编译要求。编译程序将逐步成为用户和计算机间的主要界面。

3.计算机系统设计的基本方法

从计算机系统的多级层次结构出发,可以有“由上往下”、“由下往上”和“由中间开始”三种不同的设计思路。

①“由上往下”设计是先考虑如何满足应用要求,确定好面对使用者那级机器应有什么基本功能和特性,如基本命令、指令或语言结构、数据类型以及格式等,然后再逐级往下设计,每级都考虑怎样优化上一级实现。这样设计的计算机系统对于设计时所面向的应用必然是很好的。因此,它适合于专用机设计,但不宜用于通用机的设计。因为当应用对范围改变时,软、硬件分配会很不适应,而使系统效率急剧下降。

②“由下往上”设计是不管应用要求,只根据能拿到的器件,参照或吸收已有各种机器的特点,先设计出微程序机器级(如果采用微程序控制)及传统机器级,然后再为不同应用配多种操作系统和编译系统软件,使应用人员可根据所提供的语言种类、数据形式,采用合适的算法来满足相应的应用。这是20世纪60～70年代前常用的通用机设计思路。

由于是在硬件不能改变的情况下去被动设计软件,尽管某些硬件的设计不利于软件的实现,而且有时只需稍许改变或增加某些功能就可大大简化软件的设计,也无法加以改变。因此,由下往上设计在硬件技术飞速发展而软件发展相对缓慢的今天,

难以适应系统设计要求,所以已很少使用。

③“从中间开始设计”的“中间”指的是层次结构中的软硬交界面,目前多数是在传统机器级与操作系统机器级之间。

进行合理的软、硬件功能分配时,既要考虑能拿到的硬、器件,又要考虑可能的应用所需的算法和数据结构,先定义好这个交界面。确定哪些功能由硬件实现,哪些功能由软件实现,同时还要考虑好硬件对操作系统、编译系统的实现提供些什么支持。然后由这个中间点分别往上、往下进行软件和硬件的设计。软件人员依次设计操作系统级、汇编语言级、高级语言级和应用语言级;硬件人员依次设计传统机器级、微程序机器级、数字逻辑级。

软件和硬件并行设计,可缩短系统设计周期,设计过程中可交流协调,是一种交互式的、较好的设计方法。当然,这要求设计者应同时具备丰富的软、硬、器件和应用等方面的知识。又由于软件设计周期一般比较长,为了能在硬件研制出来之前开展软件的设计测试,还应有有效的软件设计环境和开发工具。如在某个宿主机上建立目标机的指令模拟器、系统结构分析模拟器,以及良好的调试程序、性能评价的测试程序等。

1.3 计算机系统的性能评价

衡量计算机系统性能可有各种指标,但最为关键的是时间。时间可根据计算方法给以不同的定义,如响应时间、CPU 时间等。响应时间是指在用户向计算机系统送入一个任务后,直到获得他所需要的结果所需的等待时间,其中包括了访问磁盘和访问主存储器时间、CPU 运算时间、I/O 动作时间以及操作系统工作的时间开销等。虽然这种定义比较直观,但对于多道程序,由于 CPU 可在某一程序等待 I/O 操作时转去执行其他程序,响应时间并不能区别这种情况。另一种情况是只考虑 CPU 时间,此时便可加以区别,它将不包括等待 I/O 操作的时间以及 CPU 转去运行其他程序所用的时间。当然 CPU 时间本身还可分为用户 CPU 时间和系统 CPU 时间。系统 CPU 时间的统计很难做到精确,因为这实际上是要求操作系统进行自测量。此外,当比较具有不同系统代码的机器时,由于系统 CPU 时间是不一样的,因而误差较大,故采用用户 CPU 时间作为性能衡量时间较为妥当。当然,在衡量未加载系统的性能时,采用前述的响应时间较为合适,而衡量 CPU 性能则宜采用用户 CPU 时间。下面主要讨论以用户 CPU 时间来衡量的 CPU 性能。

1.3.1 CPU 性能

绝大多数计算机都使用以固定速率运行的时钟,它的运行周期称为时钟周期(Clock cycles),它常以时间长短(以 ns 计算)或运行速率(以 MHz 计算)来表示。一个程序在 CPU 上运行所需的时间 T_{CPU},可用以下的公式表示:

$$T_{CPU} = I_N \times CPI \times T_C$$

式中，I_N 表示要执行程序中的指令总数，CPI 表示执行每条指令所需的平均时钟周期数，而 T_C 表示时钟周期的时间。

由此公式可见，用户 CPU 时间取决于三个特征：时钟周期（或速率），每条指令所需时钟周期数以及程序中总的指令数。其中，I_N 主要取决于机器指令系统和编译技术，CPI 主要与计算机组成和指令系统有关，而 T_C 则主要由硬件工艺和计算机组成决定。

每条指令平均所需时钟周期数 CPI，可由下式表示：

$$CPI = \sum_{i=1}^{n} \left(CPI_i \times \frac{I_i}{I_N} \right)$$

其中，I_i 表示第 i 类指令在程序中执行次数，CPI_i 表示执行一条第 i 类指令所需的平均时钟周期数，n 为程序中所有的指令种类数，I_i / I_N 表示第 i 种指令在程序中所占比例。

下面通过两个例子来说明上述概念和公式的应用。

例 1.2　假定在设计机器的指令系统时，对条件转移指令的设计有以下两种不同的选择：

CPU_A 采用一条比较指令来设置相应的条件码，由紧随其后的转移指令对此条件码进行测试，以确定是否要进行转移。显然为实现一次条件转移就必须使用比较和测试两条指令。

CPU_B 采用使转移指令不仅具有判别是否实现转移功能，还包含有上述方案中的比较指令的功能，这样实现一次条件转移就只需要一条指令便可完成。

假设在两个机器的指令系统中，执行条件转移指令需 2 个时钟周期，而其他的指令只需 1 个时钟周期。又假设在 CPU_A 上，要执行的指令中有 20% 是条件转移指令，由于每条转移指令都需要一条比较指令，因此，比较指令也将占据 20%。由于 CPU_B 在转移指令中包含了比较功能，因此它的时钟周期就比 CPU_A 的要慢 25%。现在要问，采用不同转移指令方案的 CPU_A 和 CPU_B，哪一个工作速度会更快些？

解：根据上述假设，可计算得 $CPI_A = 0.2 \times 2 + 0.8 \times 1 = 1.2$，因为转移指令需 2 个时钟周期，而其余的指令，包括比较指令，只需 1 个时钟周期。所以

$$T_{CPU_A} = I_{NA} \times 1.2 \times T_{CA} = 1.2\ I_{NA} \times T_{CA}$$

对于 CPU_B，由于没有比较指令，从而使转移指令由原来的占 20% 上升为 20% ÷ 80% = 25%，它也需 2 个时钟周期，而其余的 75% 指令只需 1 个时钟周期，因而有 $CPI_B = 0.25 \times 2 + 0.75 \times 1 = 1.25$。由于 CPU_B 中没有比较指令，因此 $I_{NB} = 0.8 \times I_{NA}$。此外，$T_{CB} = 1.25\ T_{CA}$，所以

$$T_{CPU_B} = I_{NB} \times CPI_B \times T_{CB} = 0.8\ I_{NA} \times 1.25 \times 1.25\ T_{CA} = 1.25\ I_{NA} \times T_{CA}$$

与 T_{CPU_A} 相比较，可见由于 CPU_A 所需时间较小，所以 CPU_A 比 CPU_B 运行得更

快些。

例 1.3 在上例中,如果 CPU_B 的时钟周期只比 CPU_A 的慢 10%,那么此时,哪一个 CPU 会工作得更快些?

解:

$T_{CPU_A} = 1.2 I_{NA} \times T_{CA}$,而此时因 $T_{CB} = 1.10 T_{CA}$,故

$T_{CPU_B} = 0.8 I_{NA} \times 1.25 \times 1.10 T_{CA} = 1.10 I_{NA} \times T_{CA}$

由于 T_{CPU_B} 所需时间较少,故 CPU_B 比 CPU_A 运行得更快些。

1.3.2 MIPS 和 MFLOPS

除了时间评估标准之外,MIPS 和 MFLOPS 也是比较常用的性能评估标准。

1.MIPS

MIPS(Million Instructions Per Second,每秒百万次指令)是一个用来描述计算机性能的尺度。对于给定的一个程序,MIPS 可表示成:

$$\text{MIPS} = \frac{I_N}{T_E \times 10^6} = \frac{I_N}{I_N \times \text{CPI} \times T_C \times 10^6} = \frac{R_C}{\text{CPI} \times 10^6}$$

其中,T_E 表示执行该程序所需时间;R_C 表示时钟速率,它是时钟周期时间 T_C 的倒数。

在使用 MIPS 时应注意它的应用范围,它只适宜于评估标量机,因为在标量机中执行一条指令,一般可得到一个运算结果,而向量机中,执行一条向量指令通常可得到多个运算结果,因此,用 MIPS 来衡量向量机是不合适的。在 MIPS 中,不仅是运算指令,所有的服务性指令,如取数、存数、转移等都计算在内,而在浮点运算中服务性指令均不予计入。

有时还用相对 $MIPS_{ref}$ 这一标准,这需要事先选择一个给定的参照计算机的性能,然后与其比较。因此有:

$$\text{MIPS}_{ref} = \frac{T_{ref}}{T_V} \times \text{MIPS}_{ref}$$

式中,T_{ref} 表示在参照机上程序的执行时间;T_V 表示相同程序在要评估机器上执行时间;$MIPS_{ref}$ 表示所约定的参照机的 MIPS 速率。在 20 世纪 80 年代,常以 DEC 公司的 VAX-11/780 计算机作为参照机,称其为 1 MIPS 机器。

2.MFLOPS

MFLOPS(Million FLoating point Operations Per Second,每秒百万次浮点运算)可用如下式子表示:

$$\text{MFLOPS} = \frac{I_{FN}}{T_E \times 10^6}$$

式中，I_{FN} 表示程序中的浮点运算次数。

MFLOPS 测量单位比较适用于衡量向量机的性能，因为一般而言，同一程序运行在不同计算机上时往往会执行不同数量的指令数，但所执行的浮点数个数常常是相同的。采用 MFLOPS 作为衡量单位时，应注意它的值不但会随整数、浮点数操作混合比例的不同发生变化，而且也会随快速和慢速浮点操作混合比例的变化而变化。例如，运行由 100%浮点加法组成的程序所得到的 MFLOPS 值将比由 100%浮点除法组成的程序要高。因此，通常对源程序中的每种实际的浮点操作乘以一个正则化值，然后再进行 MFLOPS 的求值。例如，在 Livermore 循环测试程序中，浮点加、减、乘及比较操作的正则化值为 1；浮点除法、开方操作的正则化值为 4；而浮点指数、三角函数等操作的正则化值为 8。

MFLOPS 和 MIPS 两个衡量值之间的量值关系没有统一的标准，一般认为在标量计算机中执行一次浮点运算需 2～5 条指令，平均约需 3 条指令，故有 1MFLOPS≈3MIPS。

1.3.3 基准测试程序

在进行计算机系统的评价时，除了与被评价的机器的结构、功能等特性参数有关以外，还与输入(即该计算机系统的工作负荷(Workload))有密切关系。被评价的一个计算机系统往往对某一种工作负荷表现出较高性能，而对另一种工作负荷则可能呈现较低性能。为了对计算机系统的性能进行客观的评价，就需要选取具有真实代表性的工作负荷。通常采用不同层次的基准(Benchmark)测试程序来评价系统性能。

① 采用实际应用程序。例如，C 语言的各种编译程序；Tex 正文处理软件以及 Spice 那样的 CAD 工具软件。

② 采用核心程序。这是从实际程序中抽取少量关键循环程序段，并以此来评估性能，例如 Livermore 24 Loops(24 个循环段)和 Linpack(解线性方程组)便是典型代表，但这些核心程序，只具有评价性能价值。

③ 合成测试程序。它类似于核心程序方法，但这种合成测试程序是人为编制的，较流行的合成测试程序有 Whetstone 和 Dhrystone 两种。

Whetstone 是有关整、浮点运算的合成混合，还包括了超越函数、条件转移，函数调用。早先用 Algol 60 书写，后改用 FORTRAN 书写。

Dhrystone 主要是有关整数计算，包括字符串及数组处理。应该指出的是由于这类典型测试程序是人为编制的，因此它比核心程序离实际程序更远。

1988 年以来，美国 HP，DEC，MIPS 以及 SUN 公司等发起成立了 SPEC 组织(系统性能评价合作团体)，一致同意用一组实用程序和相应输入来评价计算机系统性能。SPEC 典型测试程序由以下 10 个程序组成：GCC，Espresso，Splce2g6，DODUC，NASA7，Li，Eqntott，Matrix300，FPPPP，TOMCATV，其中 4 个用 C 语言

编写(GCC, Espresso, Li 和 Eqntott)进行整数运算,余下 6 个用 FORTRAN 语言编写,进行浮点运算,计算所得 SPECmark 的分值越大越好,它是相对于 VAX-11/ 780 的性能,1SPEC 分值约相当于 0 .2～0 .3 MFLOPS。SPEC 还可进一步分为 $SPEC_{int}$ (整数 SPEC)和 $SPEC_{fp}$ (浮点 SPEC),以及与它相对应的 1989、1992 和 1995 三个版本。

类似于 SPEC 的组织尚有 Perfect 俱乐部,它主要由大学和公司中对并行计算感兴趣的爱好者组成,选定了称为 Perfect Club 的典型测试程序。

1.3.4 性能评价结果的统计和比较

关于计算机性能的评价,通常用峰值性能(Peak Performance)及持续性能(Sustained Performance)两个指标。峰值性能是指在理想情况下计算机系统可获得的最高理论性能值,它不能真实反映系统的实际性能。实际性能又称为持续性能,它的值往往只有峰值性能的 5%到 35%(因算法而异)。

持续性能的表示常用算术平均(Arithmetic Mean),几何平均(Geometric Mean)和调和平均(Harmonic Mean)三种平均值方法,这三种性能值(运算速率)的计算公式如下:

1 .算术性能平均值 A_m

$$A_m = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{T_i} = \frac{1}{n}\left(\frac{1}{T_1} + \frac{1}{T_2} + \cdots + \frac{1}{T_n} \right)$$

(若以执行时间表示性能,则有 $A_m = \frac{1}{n}\sum_{i=1}^{n} T_i$)

2 .几何性能平均值 G_m

$$G_m = \sqrt[n]{\left(\prod_{i=1}^{n} R_i\right)} = \sqrt[n]{\left(\prod_{i=1}^{n} \frac{1}{T_i}\right)}$$

3 .调和性能平均值 H_m

$$H_m = \frac{n}{\sum_{i=1}^{n} \frac{1}{R_i}} = \frac{n}{\sum_{i=1}^{n} T_i} = \frac{n}{T_1 + T_2 + \cdots + T_n}$$

以上三个公式中, R_i 表示由 n 个程序组成的工作负荷中执行第 i 个程序的速率, T_i 表示执行第 i 个程序所需的时间,这里, $R_i = 1/ T_i$。

如果考虑工作负荷中各程序不会以相等比例出现这一情况,就需要对各程序的执行速率或执行时间加上相应权值,此时有:

$$A_m = \sum_{i=1}^{n} W_i R_i = \sum_{i=1}^{n} W_i \frac{1}{T_i}$$

$$G_m = \prod_{i=1}^{n} (R_i)^{wi} = (R_1)^{w1} \times (R_2)^{w2} \times \cdots \times (R_n)^{wn}$$

$$H_m = \frac{1}{\sum_{i=1}^{n} \frac{W_i}{R_i}} = \frac{1}{\sum_{i=1}^{n} T_i W_i}$$

从上述的三种表示方式中,可以看到只有 H_m 值是真正与运行所有典型测试程序所需的时间总和成反比关系的,若以前面提及的衡量性能指标的惟一标准是时间这一点来看,用 H_m 值来衡量计算机系统性能是较为精确的。但 G_m 表示法有一个很好的特性:

$$\frac{G_m(X_i)}{G_m(Y_i)} = G_m\left[\frac{X_i}{Y_i}\right]$$

即几何平均比与比的几何平均是相等的,因此在对各种机器性能比较而进行性能规格化(即以某台机器性能作参考标准,其他机器性能除以该参考标准所得到的比值)过程中,不论取哪一台作参考机,G_m 均能保持比较结果的一致性,A_m 和 H_m 由于没有这样的特性,因而在作比较时,就不如 G_m 那样方便。下面通过一个例子来说明这一点。

表 1.1 中示出了用两个基准测试程序对三台计算机进行测试所获得的运行时间。由表中可见,对基准测试程序比 B_1,Y 机的速度为 X 机的 2 倍;但对基准测试程序 B_2,Y 机速度仅为 X 机的一半。类似地有,对 B_1,Z 机速度仅为 X 机的一半;对 B_2,Z 机的速度则为 X 机的 2 倍。直观地看,这 3 台机器应有相同的性能。然而,如果将这些测试值以 X 机为基准进行规格化(表中括号中的值),并求出相应的 Am 值,就会发现:Y 机和 Z 机速度均比 X 机要慢 25%。

表 1.1　　以 X 机为标准规格化测试值和 A_m 值

基准测试程序	处　理　机		
	X	Y	Z
B_1	20(1.00)	10(0.50)	40(2.00)
B_2	40(1.00)	80(2.00)	20(0.50)
A_m	(1.00)	(1.25)	(1.25)

更糟糕的是,如果我们以 Y 机作为标准来规格化测试,将得到表 1.2 所示的 A_m 值。由该结果值可知,现在 Y 机的速度比 X 机快 25%,而比 Z 机要快 2 倍多,尽管 Y 机的总的基准测试程序的运行时间要比 X 机和 Z 机都要长。显然选用这种标准机来规格化 A_m,将会得到不同的结论,因此这种评价方法是不可取的。

表 1.2　　以 Y 机为标准规格化测试值和 A_m 值

基准测试程序	处 理 机		
	X	*Y*	*Z*
B_1	20(2 .00)	10(1 .00)	40(4 .00)
B_2	40(0 .50)	80(1 .00)	20(0 .25)
A_m	(1 .25)	(1 .00)	(2 .13)

对于几何平均值 G_m,由于有前面所提到的良好特性则不会发生如同 A_m 那样的错误情况。如表 1 .3 和表 1 .4 所示,分别同样以 *X* 和 *Y* 机作为标准机,对测试值进行规格化。可以看出两者得到的结果是一致的,表明这三台处理机具有相同性能这一正确结论。

表 1 .3 以 X 机为标准规格化测试值和 G_m 值

基准测试程序	处 理 机		
	X	*Y*	*Z*
B_1	20(1 .00)	10(0 .50)	40(2 .00)
B_2	40(1 .00)	80(2 .00)	20(0 .50)
G_m	(1 .00)	(1 .00)	(1 .00)

表 1 .4 以 Y 机为标准规格化测试值和 G_m 值

基准测试程序	处 理 机		
	X	*Y*	*Z*
B_1	20(2 .00)	10(1 .00)	40(4 .00)
B_2	40(0 .50)	80(1 .00)	20(0 .25)
G_m	(1 .00)	(1 .00)	(1 .00)

1. 3. 5 Intel 微处理器性能评价

微处理器发展非常迅速,种类日趋繁多,应用也日益广泛,仅仅靠 CPU 型号和主频的标识,不能充分说明处理器的性能。为此,Intel 公司制定并发展了 iCOMP 指数体系,通过它希望能给处理器性能一个较为科学、公正的评价。下面介绍 iCOMP Intel 体系的基本架构。

1 .iCOMP1 .0 和 2 .0

为向一般用户提供一个理解和比较 Intel 微处理器性能相对差异的易用工具,

Intel 公司于 1992 年推出了 iCOMP(intel COmparative Microprocessor Performance)微处理器性能比较指数体系,后来被称为 iCOMP 1.0 版。

iCOMP1 .0 考虑了那个时期桌面计算机系统的应用,综合了 9 项评估项目:适用于 DOS 应用的 16 位整数运算、适用于 CAD/ CAM 的 16 位浮点运算、适用于 UNIX 应用的 32 位整数运算和 32 位浮点运算、适用于多媒体应用的 16 位和 32 位视频运算、16 位和 32 位图形处理。每项指定了测试基准 BM(BenehMark),并分别予以不同的权值 P,如表 1 .5 所示。注意,它的视频运算和图形处理的权值为 0,Intel 当时的打算是随着多媒体和 3D 动画应用的普及,将来再逐渐改变权值分配。

表1 .5　　iCOMP 1 .0 的评估条件

评 估 项 目	测 试 基 准	权 值(P)
16 位整数运算	ZD Labs CPUmix	67%
16 位浮点运算	Whetstone	2%
16 位浮点运算	ZD Labs CPUmix	1%
32 位整数运算	SPEC int 92	25%
32 位浮点运算	SPECfp 92	5%
16 位视频运算	ZD Labs/ SPEC int 92	0%
32 位视频运算	ZD Labs/ SPEC int 92	0%
16 位图形运算	ZD Labs/ SPEC int 92	0%
32 位图形运算	ZD Labs/ SPEC int 92	0%

iCOMP 1 .0 以不含 FPU 的主频 25MHz 的 80486(486SX-25)作为性能测试的基值(Base),即它的 iCOMP 值为 100。以如下公式计算某类型处理器的 iCOMP 值:

$$iCOMP = 100 \times \left[\left(\frac{BM_1}{Base\_BM_1} \right) P_1 + \left(\frac{BM_2}{Base\_BM_2} \right) P_2 + \cdots + \left(\frac{BM_9}{Base\_BM_9} \right) P_9 \right]$$

其中,$Base\_BM_i$ 代表 486SX-25 在第 i 项下的测试值,BM_i 代表被测处理器在第 i 项下的实测值,P_i 为第 i 项的权值。

早期某些主频下 Pentium 处理器的测试,按上式加权平均计算出的 iCOMP 1 .0 值,如表 1 .6 所示。

表1 .6　　早期 Pentium 主频与性能指数

CPU 类型	主频(MHz)	外总线时钟频率(MHz)	倍频因子	iCOMP1 .0 指数

Pentium-60	60	60	1	510
Pentium-66	66	66	1	567
Pentium-75	75	50	1.5	610
Pentium-90	90	60	1.5	735
Pentium-100	100	66	1.5	815

自 iCOMP1.0 公布之后,短短几年微处理器性能和市场应用发生迅速改变。首先是向 32 位桌面操作系统和应用的快速转变,据调查到 1997 年底与新 PC 一块装机的软件的 90%是 32 位的,仅 10%是 16 位的,并推测到 1998 年底装机软件几乎全部都是 32 位的。其次是多媒体、网络通信以及 3D 图形处理的应用已形成规模。另外,测试基准程序已经修改,或者已开发出反映新兴软件中特有指令综合功能的新基准程序。这一切使得 Intel 公司将它的 iCOMP 评估体系推进到 iCOMP 2.0 版。

1996 年公布的 iCOMP 2.0,由 4 个工业标准的 32 位测试基准和一个 Intel 公司自行开发的多媒体测试基准(Intel Media Benchmark)组成。它们的应用类别和权值,如表 1.7 所示。

表 1.7 iCOMP 2.0 的评估条件

评估项目	测试基准	权 值(P)
传统商业应用	CPUmark32	40%
高端应用	Norton SI32	15%
常规用途的整数运算	SPEC base-int 95	20%
常规用途的浮点运算	SPEC base-fp 95	5%
常规多媒体、通信和视频应用	Intel Media Benchmark	20%

iCOMP 2.0 的 iCOMP 值计算公式仍同上述相同(只是 5 项加权平均),以 Pentium120(主频 120MHz,外频 60MHz)为性能测试的基值,即它的 iCOMP 值为 100。表 1.8 列出 Pentium 系列微处理器在 2.0 版下的 iCOMP 值。

表 1.8 Pentium 系列微处理器主频与性能指数

CPU 类型	主频(MHz)	外总线时钟频率(MHz)	倍频因子	iCOMP2 .0 指数
Pentium-120	120	60	2	100
Pentium-133	133	66	2	111
Pentium-150	150	60	2 .5	114
Pentium-166	166	66	2 .5	127
Pentium-200	200	66	3	142
Pentium MMX-166	166	66	2 .5	160
Pentium MMX-200	200	66	3	182
Pentium MMX-233	233	66	3 .5	203
Pentium Ⅱ-233	233	66	3 .5	267
Pentium Ⅱ-266	266	66	4	303
Pentium Ⅱ-300	300	66	4 .5	332
Pentium Ⅱ-333	333	66	5	366

2 .iCOMP 3 .0

为能反映处理器在当今迅猛增长的多媒体、3D 和 Internet 应用需求的性能指标,1999 年初 Intel 公司公布了 iCOMP 3 .0 性能评估体系。这次,Intel 将 PC 应用明确划分成 4 大领域:多媒体、3D、Internet 和 Productivity(生产率)。其中,Productivity 是指除去多媒体、3D、Internet 之外的所有 PC 应用,典型的是各种电子表格、数据库和字处理软件。iCOMP 3 .0 由 6 项工业标准的测试基准组成,它们是:

(1) CPU mark * 99(BM_1)

Zitt-Davis 开发的 CPU mark * 99 是一个 Windows 测试基准,它测试处理器、内部 Cache、外部 Cache 以及系统 RAM 的性能。

(2) Wintune * 98 Advanced CPU Integer Test(BM_2)

Wintune * 98 是一个用于 Windows 95、Windows 98 和 Windows NT 系统的诊断测试和基准程序,它完成 7 项测试,其中包括 CPU、存储器、视频和磁盘速度。测试结果可通过 Internet 访问 Windows 期刊社维护的一个中央数据库,来与其他类似的机器进行性能比较。

(3) Multimedia Mark * 99(BM_3)

它是由 Futuremark Corporation 开发的一个测试套件,主要用于测试当代 PC 在“现实世界”环境中的多媒体性能,这包括 MPEG-1 视频编码、MPEG-1 视频播放、图像处理和音频效果。

(4) 3D WinBench * 99 - 3D Lighting and Transformation Test(BM_4)

3D WinBench * 99 是一个系统级的 3D 性能测试套件,测试包括 CPU 和图形子

系统的性能。为了解 CPU 的 3D 性能,要使用此套件中的 3D Lighting and Transformation Test,它测试 3D 图形流水线中 CPU 涵盖部分的性能。

(5) WinBench * 99-FPU WinMark(BM_5)

这是由 Zitt-Davis 开发的、主要用于测试处理器浮点子系统的综合测试套件。为测量 FPU 在科学计算和 3D 图形透视计算的能力,此测试由 5 个算法组成:3D 图形操作,快速傅立叶变换,行星轨道计算,多边形面积计算和线性方程系数矩阵的高斯—约旦消元法。Zitt-Davis 为每种算法指定一个权值,然后加权平均得到单一的记分。

(6) JMark * 2 .0 Processor Test(BM_6)

它是 Zitt-Davis 开发的测试处理器 Java 性能的基准程序,着重测试 Java 虚拟机(JVM)的非图形工作负载方面的性能。

这 6 项测试基准的应用类别和权值,如表 1 .9 所示。

表1 .9 iCOMP 3 .0 的评估条件

评 估 项 目	测 试 基 准	权值(P)
Productivity	CPU mark * 99	20%
Productivity	Wintune * 98 Advanced CPU Integer Test	20%
多媒体	Multimedia Mark * 99	25%
3D	3D WinBench * 99 - 3D Lighting and Transformation Test	20%
Productivity/ 3D	WinBench * 99-FPU	5%
Internet	JMark * 2 .0 Processor Test	10%

iCOMP3 .0 值的计算公式仍类似于前面介绍的计算方法(只是 6 项加权平均)。只是它以 PentiumⅡ-350 为性能测试的底值,并且其 iCOMP 值为 1000。故 iCOMP 值计算公式为:

$$\text{iCOMP} = 1000 \times \left[\left(\frac{BM_1}{Base\_BM_1} \right) P_1 + \left(\frac{BM_2}{Base\_BM_2} \right) P_2 + \cdots + \left(\frac{BM_6}{Base\_BM_6} \right) P_6 \right]$$

PentiumⅡ-350 以 6 个测试基准程序测得的 $Base\_BM_1$, $Base\_BM_2$, …, $Base\_BM_6$ 的值分别是:27 .8, 87 .21, 883, 26 .5, 1790, 609。将这些值代入上式,即可得到 iCOMP 3 .0 下的 iCOMP 值实际计算公式:

$$\text{iCOMP} = 1000 \times \left[\frac{BM_1}{27.8} \times 20\% + \frac{BM_2}{87.21} \times 20\% + \frac{BM_3}{833} \times 25\% + \frac{BM_4}{26.5} \times 20\% + \frac{BM_5}{1790} \times 5\% + \frac{BM_6}{609} \times 10\% \right]$$

只要以 6 个测试基准程序测出某处理器的记分,代入上式即可算出该处理器3 .0

版下的 iCOMP 值。例如,以 6 个测试基准程序测试 Pentium Ⅱ - 400,得到 BM_1 ～ BM_6 的记分为 31 .6,99 25,999,30 .1,2050,974,不难算出它的 iCOMP 值为 1130。表1. 10列出 PentiumⅡ和 PentiumⅢ一些处理器的 iCOMP 值。

表 1 .10　　PentiumⅡ/ Ⅲ处理器的主频与性能指数

CPU 类型	主频(MHz)	外总线时钟频率(MHz)	倍频因子	iCOMP3 .0 指数
PentiumⅡ-350	350	100	3 .5	1000
PentiumⅡ-400	400	100	4 .0	1130
PentiumⅡ-450	450	100	4 .5	1240
PentiumⅢ-450	450	100	4 .5	1500
PentiumⅢ-500	500	100	5 .0	1650
PentiumⅢ-550	550	100	5 .5	1780

Intel 用于 iCOMP 3. 0 性能评估时的系统配置为: Intel SE440BX-2 的主板, 128MB SDRAM 的内存, 512KB 的 2 级 Cache, Adapte AHA 2940 UW2W SCSI/ PCI 的硬盘控制器/ 总线, Seagate Cheetah ST 39102LW 的硬盘驱动器, 1024×768 分辨率、16 位颜色的图形显示器, Diamond Multimedi Viper AGP(显示存储器为 16MB SDRAM)的 Video, Diamond Monster Sound M80 PCI 的 Audio。使用的操作系统是带有 Microsoft Direct x6 .1 的 Windows 98。

1 .4　计算机系统结构的发展

1. 4. 1　计算机系统结构的演变

传统的计算机主要以运算器为中心,此外还有控制器、存储器以及输入/ 输出设备。所有的输入、输出活动都必须经过运算器。存储器中存放有指令及数据。由于这一设想是美国科学家冯·诺依曼(Von Neumann)首先提出来的,故又称为冯·诺依曼计算机,如图 1 .4 所示。

冯·诺依曼型计算机的主要特点是:

① 存储程序方式。指令和数据都以字的方式存放在同一存储器中,没有区别,由机器状态(如取指令或取操作数周期)来确定从存储器读出的字是指令还是数据。指令送往控制器译码,数据送往运算器进行运算。

② 指令串行执行,并由控制器加以集中控制。

③ 单元定长的一维线性空间的存储器。

④ 使用低级机器语言,数据以二进制形式表示。

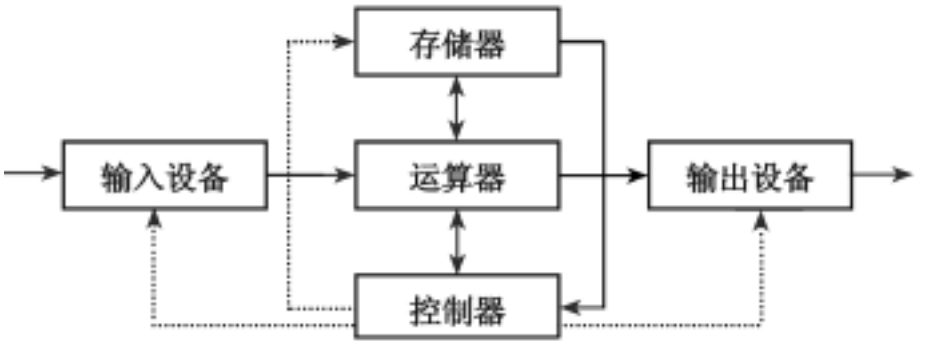


图 1.4 冯·诺依曼计算机结构

⑤ 单处理机结构,以运算器作为中心。

这是一种最为简单且容易实现的计算机结构,在当时元器件可靠性较低的情况下,也是一种很合适的结构。以后,在计算机技术发展的很长一段时期内,基本上没有离开这一结构模式。这种结构存在的主要缺点有:

(1)存在有两个主要的瓶颈。一个是物理瓶颈,即在 CPU 和存储器之间存在频繁的信息交换,而且指令中数据的访问地址通常并不是有效地址,必须经过各种变换和运算后才能得到真正的有效地址;二是智能瓶颈,即每次只能顺序地执行一条指令。

(2)低级的机器语言和高级的程序设计语言之间存在着巨大的语义差距,此差距往往要靠大量复杂的软件程序来填补。

(3)复杂的数据结构对象无法直接存放到一维线性地址空间的存储器中,必须经过地址映像。

近半个世纪以来,对冯·诺依曼型计算机结构已做了许多改进。归纳起来采用了两种方法。一种是"改良"方法,即基本上仍保留原来的工作方式,但做了许多重大改进以提高计算机系统性能,并称为改进的冯·诺依曼型计算机结构。另一种是"革命"方法,即采用一种与冯·诺依曼型计算机完全不同的方式工作。

在改进的冯·诺依曼型机中,具有以下一些重要特征,其目的是为了提高运算速度,更好地支持高级语言和结构化数据对象,它们包括:

① 增加了新的数据表示,如浮点数、字符串和十进制数的表示。

② 采用虚拟存储器,方便了高级语言编程。

③ 堆栈的引入,以支持高级语言中的过程调用、递归机制以及表达式求值等。

以上三项主要是为了更好地支持高级语言。

④ 采用变址寄存器并增加了间接寻址方式,以方便对复杂数据结构对象的访问。

这一项主要是为了支持对复杂数据结构对象的访问。

⑤ 增加 CPU 内的通用寄存器数量和增设 Cache 高速缓冲存储器,以减少 CPU 与主存储器间的过分频繁的信息交换。

⑥ 采用存储器交叉访问技术以及无冲突并行存储器,以加宽存储器带宽。

⑦ 采用流水技术,包括指令级流水和运算级流水,以加快指令及操作的执行速度。

⑧ 采用多功能部件,这样一条指令就可以对多个数据元素在不同功能部件上进行并发操作。

⑨ 采用支持处理机,如协处理机(Coprocessors)及输入/输出处理机(I/O processors),以使 CPU 能集中精力从事数值运算。

⑩ 采用自定义的数据表示,由数据中的标志符显式说明是指令还是某一种类型的数据。虽然对这一措施的优越性有争论,但至少在 LISP 那样的迟汇集(Late binding)语言中是非常重要的。

1 使程序和数据空间分开,从而增加了存储器带宽。

以上①～⑦项主要是为了提高处理速度。其中的最后两项还超出原来冯·诺依曼型机基本结构的范围。上述各种改进措施的实现使计算机系统结构从原来以运算器为中心逐步演变为以存储器为中心。

系统结构的革命性发展导致了 20 世纪 70 年代数据流计算机的出现,以及需求驱动计算机乃至初等智能计算机的出现,预计到 21 世纪将会在这方面有较大的发展和新的突破。

总的发展趋势是对计算机性能的要求越来越高,主存容量越来越大和 I/O 吞吐能力越来越强。在 20 世纪的追求目标是万亿次(Teraflops)运算速度、万亿字节存储容量(Terabyte)和每秒万亿字节的 I/O 吞吐率(Terabyte/second),简称“3T”目标。

1.4.2 软件、应用和器件对系统结构发展的影响

1.软件对系统结构发展的影响

早期的计算机系统,由于硬件比较昂贵,因此通常将许多功能用软件实现。再加上应用软件的开发,导致软件越来越多,功能越来越复杂,由于软件书写基本上依靠人力,造成软件编写及排错困难,生产效率低下,导致“软件危机”的出现,随着器件更新换代的飞速发展以及硬件价格逐渐下降,造成软件价格相对地不断上升。因此,用户就希望在新型号机出台后,原先已开发的软件仍能继续在升档换代的新型号机器上使用,即软件要求具有兼容性,或称为软件的可移植性。它是指一个软件可不经修改或只需少量修改便可由一台机器移植到另一台机器上运行,即同一软件可应用于不同环境。为了实现软件的可移植性,一般可采用如下方法:

(1)采用统一的高级语言方法

软件移植包括应用软件和系统软件两个方面的移植。软件又可用高级语言、汇编语言或机器语言来编写。由于高级语言是面向题目和算法的,与机器的具体结构

关系不大,如果能统一出一种满足各种应用需要的通用高级语言,那就很容易实现所有应用软件和部分系统软件之间的移植。

统一高级语言的方法,目前存在着三个问题。第一,目前已经存在的各种高级语言,是根据不同的应用而产生的,这些高级语言又具有不同的语法结构和语义结构。如果用一种高级语言包含所有的应用,是非常困难的。第二,计算机工作者目前对高级语言的基本结构在看法上存在不一致,如常用的 GOTO 语句。第三,习惯势力的影响。人们不愿轻易抛弃已经习惯了的高级语言和软件上的成果及经验积累。

虽然统一高级语言的方法目前实现非常困难,但仍是努力的重要方向。如果能够统一成一种或几种,对于加速软件人才的培养和软件开发意义是非常重大的。ADA 语言的出现是朝着这个方向的重大进展。

(2)采用系列机方法

所谓系列机是指在同一厂家内生产的具有相同系统结构,但具有不同组成和实现的一系列的机器。它要求预先确定好一种系统结构(软硬件界面)。然后,软件设计者依此进行系统软件设计,硬件设计者则根据不同性能、价格要求,采用各种不同的组成和物理实现技术,以向用户提供不同档次的机器。采用这样的方法后,由于机器语言程序员或是编译程序设计者所看到的这些机器的概念性结构和功能属性都是一样的,即机器语言都是一样的。因此为某一档次机器编制的软件在其他档次的机器上都可运行。

系列机方法较好地解决了硬件技术更新发展快而软件编写开发周期比较长之间的矛盾。由于系列机中的系统结构在相当长的时期内不会改变,改变的只是组成和实现技术,从而使得软件开发有一个较长的相对稳定的周期。IBM 公司首先提出了这种思想,在 1964 年推出了 IBM 360 系列机。以后又陆续推出了 IBM 370 系列机,IBM 303X,43XX,308X,309X 等系列机。DEC 公司则推出了 PDP ll 系列机,VAX ll,8000,6200,6300,6400 等系列机。这种系列机设计思想对以后问世的微型计算机以及巨型计算机都产生了影响。如 Intel 公司推出了 80X86 微机系列,Motorola 公司推出了 680X0 微机系列,CRAY 公司推出了巨型机 CRAY 系列,等等,便是例证。

在各档机器的中央处理机中,指令系统都相同,但指令的分析执行则可有顺序、重叠或流水等不同处理方式。在数据表示方面,从程序设计者所看到的各档机器的字长均为 32 位(定点数都为 16 位的半字或 32 位全字,浮点数为单、双、4 倍字长),但低、中、高档的不同型号机器,它们所采用的数据通路宽度可能分别为 8 位、16 位、32 位或 64 位。显然这种数据通信宽度对程序员来讲是透明的。此外在进行输入输出时,各档机器都有采用通道方式,但随机器档次的不同,可采用结合型方式(通道借用中央处理机中某些部件完成)或是独立型的。

在系列机中,由于机器语言、汇编语言以及编译程序在各档机器间都可通用,因此它们是软件兼容的。这种方法使得软件的开发有一个较长时间的稳定的环境,有利于计算机系统随着硬件器件技术的不断发展而升级换代,对计算机的发展起到了

很大的推动作用。但是,这种可移植性仅限于某一厂商所生产的某一系列机内部,用户不能在不同厂商的产品中进行选择。系列机的思想后来在不同厂家间生产的机器上也得到了体现,出现了兼容机。

所谓兼容机(Compatible machine)是指不同厂家能生产的具有相同系统结构,但具有不同组成和实现的一系列计算机。这样就使得一些没有软件开发能力的厂家能借用有软件开发能力的计算机厂家的已有的软件成果,在采用新的组成和实现技术后,有更好的性能价格比;另一方面还可以对原有系统结构加以某种形式扩充以使之能有更强的功能和竞争力。如 Amdahl 公司生产的与 IBM 370 兼容的 Amdahl 470,480 等,以及长城公司生产的 0520IBM-PC 兼容机,后者由于扩充了汉字处理功能,因而形成了更强的市场竞争力。

系列机方法较好地解决了软件移植的问题,但由于这种方法要求系统结构不能改变,这也就在较大程度上限制了计算机系统结构发展,而且所有的软件兼容也是有一定条件约束的。

软件兼容按性能上的高低和时间上推出的先后还可分为向上、向下和向前、向后四种兼容。

所谓向上(下)兼容,是指按某档机器编制的软件,不加修改就能运行于比它高(低)档的机器上。同一系列内的软件一般应做到向上兼容,向下兼容就不一定,特别是与机器速度有关的实时性软件向下兼容就难以做到。

所谓向前(后)兼容,是指在按某个时期投入市场的该型号机器上编制的软件,不用修改就能运行于在它之前(后)投入市场的机器上。同一系列机内的软件必须保证做到向后兼容,不一定非要向前兼容了。

(3)采用模拟和仿真方法

系列机的方法只能在系统结构相同的机器之间实现软件移植。为了实现系统结构不相同的机器之间也能实现软件移植,必须采用模拟与仿真的方法。

模拟方法是指用软件方法在一台现有的计算机上实现另一台计算机的指令系统。如在 A 机上要实现 B 机的指令系统,通常要用解释方法来完成,即对应 B 机中的每一条指令,用相应的一段 A 机(如 n 条)指令进行解释执行。这种用实际存在的机器语言解释实现软件移植的方法称为模拟。A 机常称为宿主机,B 机则称为虚拟机,因为 B 机实际上并不存在,如图 1 .5(a)所示。为了模拟 B 机系统,除了指令系统以外通常还需模拟它的系统结构环境,包括 B 机的存储系统、I/ O 子系统以及 B 机的操作系统等。对应用软件的模拟也可采用类似方法。由于模拟是采用纯软件解释执行方法,因此运行速度较慢。

另一种方法是仿真,当宿主机本身采用微程序控制时,则对 B 机指令系统每条指令的解释执行可直接由 A 机中对应的一段微程序来完成,此时 A 机仍称宿主机,但 B 机称为目标机。此外由于仿真方法中微程序是存放在控存中(模拟方法中模拟程序存放在主存中),因而实际上是有部分硬件(或固件)参与解释过程。因此仿真的

工作速度要比纯软件的模拟方法快,如图 1.5(b)所示。

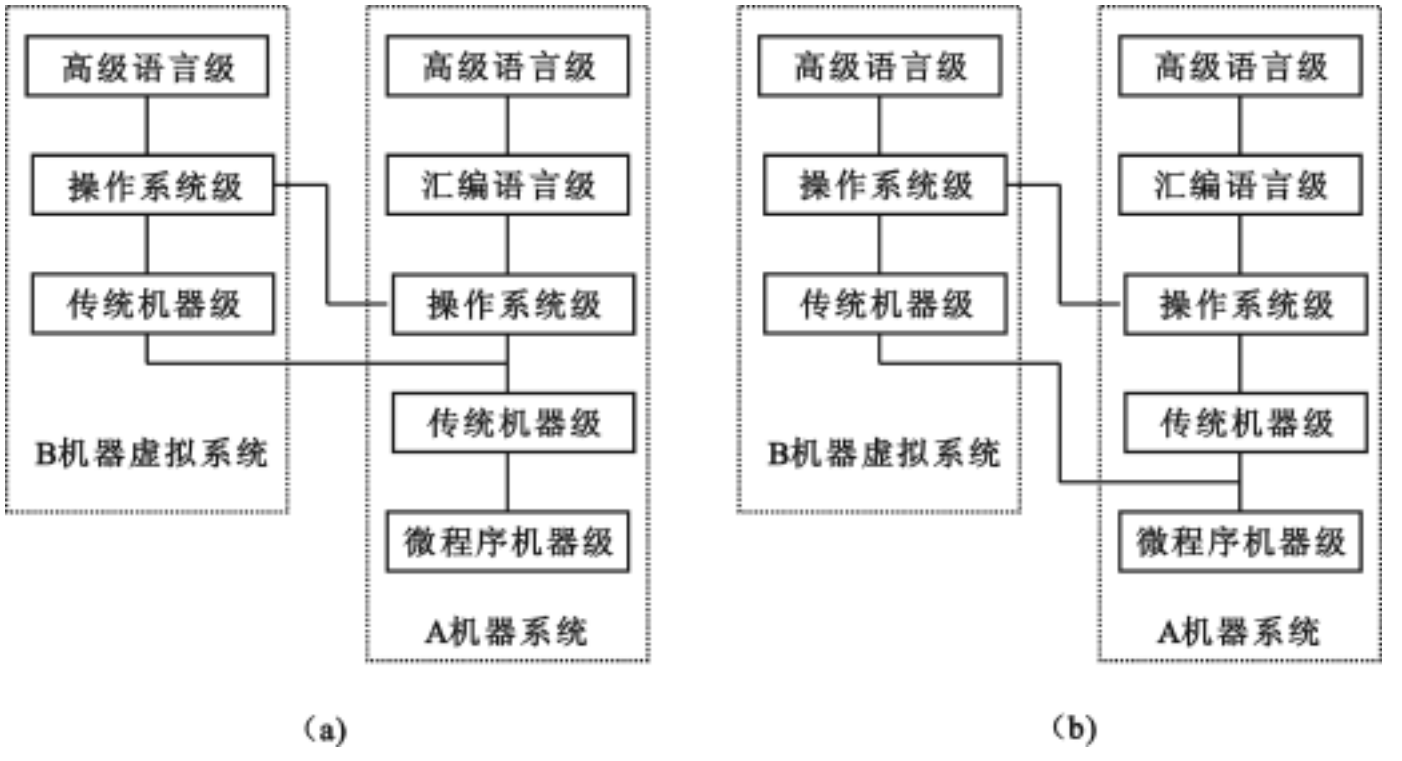


图 1.5 用模拟方法和仿真方法实现软件的移植

用微程序仿真方法实现解释执行时,由于微程序机器级结构更依赖计算机的系统结构,再加上编写仿真微程序较为费时和复杂,因此对于系统结构差别较大的机器难以完全用仿真方法来实现软件移植,所以通常将这两种方法混合使用。对于使用频率较高的指令,尽可能用仿真方法以提高运算速度,而对使用频率低且难以用仿真实现的指令(包括 I/O 指令等)则用模拟方法加以实现。

值得注意的是,在 1978 年国际标准化组织(ISO)首先提出了关于开放式系统的新概念。所谓开放式系统(Open system),是指一种独立于厂商,且遵循有关国际标准而建立的、具有系统可移植性、交互操作性,从而能允许用户自主选择具体实现技术和多厂商产品渠道的系统集成技术的系统。它是相对于原来的封闭系统或专有系统而言的。

这里的国际标准主要是指由 IEEE 的开放系统技术委员会主管制定的系统或网络中,应用软件、系统软件和硬件相互间的界面接口标准。

系统可移植性主要是指应用系统可移植性和用户(使用者)可移植性。前者指应用软件在多厂商、多硬件平台的计算机环境中,从一个平台移到另一个平台的可行度,而后者是指使用规范的一致性,这样用户就可不用改变它原来的使用习惯(或只做少许改变)便可从一个平台转移到另一个平台上工作。

系统的交互操作性主要是指在多厂商、多硬件平台的计算机环境中、一个应用软件能与另一个应用软件进行协调工作以及实现数据与资源共享的可行度,它包括了应用软件能透明地在一个系统中通过网络调用另一个系统中的数据资源及计算能

力,以及在一个系统上运行的任务可通过网络有效地分解成一些子任务并分配到其他系统上以实现这些子任务的并发操作。

开放系统还为系统的档次皆宜性或规模可伸缩性(Scalability)创造了良好条件,同一应用软件在某一开放系统的系列产品上都具有运行的能力,如从个人计算机、工作站到小型机、大型机乃至巨型机。

要达到上述目标,必须在优选的硬件和软件的平台上,对编程接口、人机接口、通信接口等加以统一,建立统一的规范和协议。

2 .应用对系统结构发展的影响

计算机应用是促使计算机系统结构发展的最根本动力。

计算机的性能是包括硬件(如主频、CPU 运算速度、字长、数据类型、主存容量、寻址空间大小、存储系统、I/O 处理能力、I/O 设备量、指令系统等)、软件(高级语言状况、操作系统功能、用户程序包等)、可靠性、可用性等多种指标的综合。20 世纪 60 年代中期的多功能通用机的概念起始于大、中型机,后来小型机和微型机也逐步实现多功能通用化了。回顾这几十年的发展,巨、大、中、小、微、亚微、微微型机的性能、价格随时间变化的趋势,如图 1 .6 所示,其中虚线为等性能线。

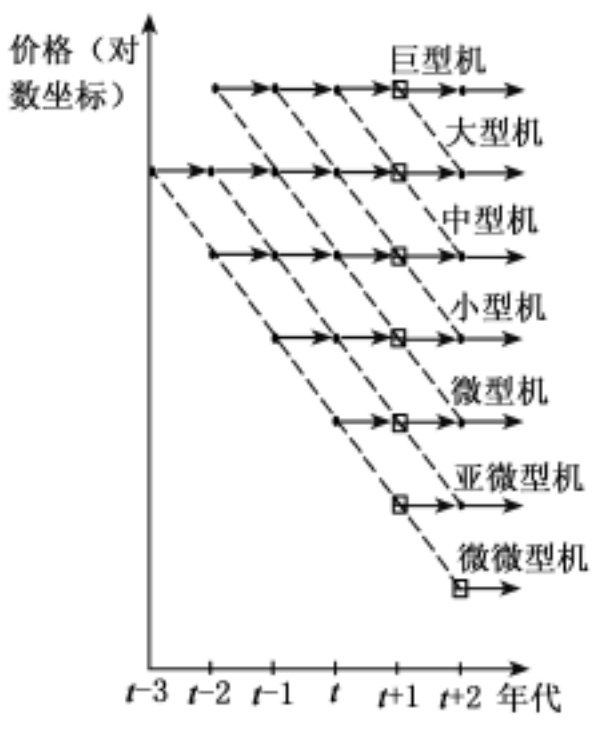


图 1 .6　计算机性能、价格变化趋势示意图

可以看出,各型机器所具备的性能是随时在动态下移的,但价格却在相当长一段时期内基本不变。因此,有些人就主张用价格来划分巨型机到微型机的不同类型。20 世纪 60 年代末(相当于图中的 t-1)问世的小型机(如 PDP-8)性能几乎与 70 年代末(对应图中的 *t*)的微型机相近,但后者的价格却下降很多。50 年代末期购置的机

器,所需费用在目前差不多可以购置一台大型机,其性能却仅接近目前的亚微型机(膝上型和笔记本型)。微型机的发展有两个趋势:一是尽量利用 VLSI 的进展,维持价格基本不变来提高性能,往小型机靠拢;二是维持性能基本不变,降低价格,研制出更低档的亚微型机及微微型(掌上型)计算机,以进一步扩大计算机的用途。

在性能上,微型机的高档机可以替代小型机以至超级小型机的低档,超级小型机的高档顶替大、中型机的低档机的情况将长期存在,这是推动大、中、小、微型机不断提高其性能的重要因素。

高端计算机应用对系统结构不断提出的基本要求是高的运算速度,大的存储容量和大的 I/O 吞吐率。

从最初的科学计算发展到应用于商业和事务处理等方面的数据处理、工业过程控制乃至日常社会生活,从而使计算机应用逐渐深入到国民经济及社会的各个领域。随着应用范围的扩大,对计算机系统的要求也就越来越高。为适应不同的应用需要,满足各种应用的多功能通用机,满足某种特殊要求的专用机,适应不同应用场合需要的大、中、小型机,需要高速运算的巨型机,以及方便灵活的微型机等的涌现都是应用需要的结果。计算机应用从最初的纯科学计算正逐步向更高级、更复杂的应用发展,经历了从数据处理、信息处理、知识处理以及智能处理这四级逐步上升的阶段,长期以来进行数据处理一直是计算机的重要任务,它所加工的数据对象相互之间是没有什么关系的。随着对数据结构研究的深入,数据处理逐步向信息处理方向发展,此时被加工的数据对象之间存在着某种语法结构构成了信息项,从而导致能有效地对信息项进行管理、增删、检索、变更、维护、查询、修改等工作的数据库的出现。当信息加入某些语义后便构成了知识,导致了对知识处理的应用,各类专家系统的出现便是这种应用日益广泛发展的结果。在 20 世纪 90 年代这种应用将会有更大发展。在知识处理基础上,使计算机能以自然形式实现人机间对话,如用语言、文字、图画、图像、文件等形式,并在引入推理功能后,就可使计算机系统具有从事定理证明、逻辑推理以及具有学习功能的能力,这就是智能处理的应用。当然在这种应用中,对处理复杂性和所需技巧也就要求较高,而可开发的并行程度显然也是较高的。在 21 世纪这种应用将逐步成为主流。与数据处理及信息处理主要面向数据世界不同,知识处理与智能处理主要是面向现实世界。在 21 世纪的高度信息化社会中,如何能够像人类大脑那样以灵活(柔性)方式来处理这种实况计算(Real world computing)的问题,将是智能处理和柔性信息处理所要研究的主要问题。

3 .器件对系统结构发展的影响

计算机所用的基本器件已经从电子管、晶体管、小规模集成电路、大规模集成电路迅速发展到超大规模集成电路,并使用或开始使用砷化镓器件、高密度组装技术和光电子集成技术。器件的发展是计算机换代的两个标志之一,也是推动系统结构和

组成前进的关键因素和主要动力。可从以下三个方面看:

(1) 从器件的功能和使用方法来看

器件的功能和使用方法发生了很大的变化,由早先使用的非用户片,发展到使用现场片和用户片。这种变化影响着系统结构和组成技术的发展。

非用户片也称通用片,其功能是在器件厂生产时确定下来的,计算机设计者只能使用,不能改变器件内部的功能。例如门、触发器、多路开关、加法器、译码器、寄存器、计数器等通用逻辑类器件。这类器件的优点是通用性好,灵活方便,但集成度难以提高,因而可靠性低。

现场片是指用户可根据需要改变器件内部的功能或内容,以适应结构和组成变化的需要。例如,可编程只读存储器 PROM,现场可编程逻辑阵列 FPLA 等。它不仅使用灵活,功能强,可取代硬联组合网络,还可构成时序网络,加上又是存储型芯片,也可以实现乘法运算和码制转换、函数计算等功能。其规整性和通用性强,适合于大规模集成。

用户片则是专门按用户要求生产的高集成度的 VLSI 器件。完全按用户要求设计的用户片称全用户片。为解决器件厂和整机厂的矛盾,发展的门阵列、门-触发器阵列等为半用户片。

通常,同一系列各档机器可以分别用通用片、现场片或用户片实现。就是同一型号的机器一般也是先用通用片或现场片实现,等到机器比较成熟并取得用户信任后,再改为半用户片或全用户片实现。至于高速机器一般一开始就用门阵列片甚至用用户片,只有这样才能发挥出单元电路的高速性。

(2) 从器件在计算机系统中的地位来看

① 器件的发展使计算机主频速度迅速提高。早先的计算时间是以 ms 为单位,1960 年左右是以 us 为单位,现在以 ns 为单位。几十年来,无论是器件的速度(门、触发器的级延迟,存储器的存储周期等)、集成度、体积、可靠性、价格等随时间都呈指数型地改进,这就使计算机的性能价格比有了显著的提高。如果没有器件集成度和速度的迅速提高,机器的主频和速度就不能发生数量级上的提高。

② 器件的发展推动了系统结构、组成的发展。如果器件的可靠性未发生数量级上的提高,是无法使用后面要讲述的流水技术的实现的。如果没有高速、廉价的半导体存储器,则能使解题速度得以迅速提高的高速缓冲存储器(Cache)及早在 20 世纪 60 年代初期就已提出的虚拟存储器就无法真正实现。没有可编程只读存储器 PROM 器件的出现,早在 20 世纪 50 年代初期就已提出的微程序技术也无法真正得到广泛应用。只有有了高速相联存储器器件,才有相联处理机这种结构的发展,才能推动向量机、数组机、数据库机器的发展。

③ 器件的发展也使系统结构的“下移”速度加快。大型机的各种数据表示、指令系统、操作系统很快出现在小型机、微型机上。器件的发展为实现多个 CPU 的分布

处理提供了基础。智能终端、智能机的出现也都表明了这点。

④ 器件的发展还促进了算法、语言和软件的发展。在硬件结构上，由数百个甚至上万个微处理器组成的 MPP 系统有着很高的性能价格比和良好的可扩展性。促使人们为它不断探索研究新的并行算法、并行语言及开发新的并行处理应用软件和控制并行操作的操作系统软件，使系统的规模和处理速度能随节点处理器数的增加而显著提高。由 1024 个有着 100 MFLOPS 性能的 RISC 微处理器构成的 MPP 系统，其最高性能可达 100 GFLOPS，大大超过巨型向量机的性能，而其造价只有巨型向量机的 1/ 5。nCUBE 公司 1992 年 6 月推出的 nCUBE2 由 8192 个微处理器组成，性能达 34 GFLOPS 和 0. 123 TIPS。到了 1995 年，推出的 nCUBE3，其性能已达 6. 5 TFLOPS，而其价格反而有了显著降低。日本制定的 RWC(真实世界计算)计划准备用 l M(1 024×1 024)个微处理器组成 MPP 系统，实现高达 125 TIPS(即 125 万亿条指令每秒)的性能。

(3) 从计算机系统的设计方法来看

器件的发展改变了逻辑设计的传统方法。传统的逻辑设计方法是逻辑化简，力图节省所用门的个数、门的输入端数及门的级数等，以节省功耗、降低成本、提高速度。而对于现今采用 VLSI 器件来说，斤斤计较省几个门只能带来设计周期的延长，组成实现的不规整，故障诊断的困难，机器产量的低下，这些都使成本反而提高。因此，应当考虑采用什么样的组成方式才能发挥 VLSI 器件技术发展所带来的好处，以及选用什么样的 VLSI 器件能使机器的性能价格比更高，应当着眼于在满足系统结构所提出的功能和速度的情况下，如何缩短设计周期、提高系统效能及能用上大批量生产的通用 VLSI 芯片。

当今流行的设计方法有计算机辅助设计(CAD)和设计自动化系统(DAS)，支持各种应用的库软件、工具软件相当丰富。

总之，系统结构设计者要密切了解器件的现状和发展趋势，经常关注和分析新器件的出现和集成度的提高会给系统结构发展带来什么样的新途径和新方向。

软件、应用、器件对系统结构的发展是有着很大影响的，反过来，系统结构的发展又会对软件、应用、器件的发展提出新的要求，促使其有更大的发展。计算机系统结构设计者不仅要了解结构、组成、实现的关系，还要充分了解掌握软件、应用、器件发展的现状、趋势和发展要求。只有这样，才能对系统的结构进行有成效的设计、研究和探索。

习　　题

1. 解释下列术语：

翻译　解释　层次结构　计算机系统结构　计算机组成　计算机实现　透明性

固件　系列机　软件兼容　兼容机　模拟　仿真　宿主机　指令流　数据流　多倍性　Amdahl 定律　CPI　MIPS　MFLOPS

2. 如有一个经解释实现的计算机,可以按功能划分成 4 级。每一级为了执行一条指令需要下一级的 N 条指令解释。若执行第 1 级的一条指令需 Kns 时间,那么执行第 2、3、4 级的一条指令各需要用多少时间(ns)?

3. 有一个计算机系统可按功能划分成 4 级,各级的指令都不相同,每一级的指令都比其下一级的指令在效能上强 M 倍,即第 i 级的一条指令能完成第 i～1 级的 M 条指令的计算量。现若需第 i 级的 N 条指令解释第 i+1 级的一条指令,而有一段第 1 级的程序需要运行 Ks,问在第 2,3,4 级上的一段等效程序各需要运行多长时间(s)?

4. 硬件和软件在什么意义上是等效的,在什么意义上又是不等效的,试举例说明。

5. 试以实例说明计算机系统结构、计算机组成与计算机实现之间的相互关系与相互影响。

6. 什么是透明性概念?对计算机系统结构,下列哪些是透明的?哪些是不透明的?

存储器的模 m 交叉存取;浮点数据表示;Ⅳ O 系统是采用通道方式还是外围处理机方式;数据总线宽度;字符行运算指令;阵列运算部件;通道是采用结合型的还是独立型的;PDP-11 系列中的单总线结构;访问方式保护;程序性中断;串行、重叠还是流水控制方式;堆栈指令;存储器最小编址单位;Cache 存储器。

7. 从机器(汇编)语言程序员看,以下哪些是透明的?

指令地址寄存器;指令缓冲器;时标发生器;条件码寄存器;乘法器;主存地址寄存器;磁盘外设;先行进位链;移位器;通用寄存器;中断字寄存器。

8. 下列哪些对系统程序员是透明的?哪些对应用程序员是透明的?

系列机各档不同的数据通路宽度;虚拟存储器;Cache 存储器;程序状态字;“启动Ⅳ O”指令;“执行”指令;指令缓冲寄存器。

9. 系列机概念对计算机发展有什么意义?系列机软件兼容的基本要求是什么?列出几个熟知的系列机。

10. 用一台 40MHz 处理机执行标准测试程序,它所含的混合指令数和相应所需的时钟周期数,如表 1.11 所示,求有效 CPI,MIPS 速率和程序的执行时间。

表 1.11

指令类型	指令数	时钟周期数
整数运算	45 000	1

数据传送	32 000	2
浮点运算	15 000	2
控制传送	8 000	2

11. 某工作站采用时钟频率为 15MHz、处理速率为 10MIPS 的处理机执行一个已知混合程序。假定每次存储器存取为 1 周期延迟,试问:

(1) 此计算机的有效 CPI 是多少?

(2)假定将处理机的时钟提高到 30MHz,但存储器子系统速率不变。这样,每次存储器存取需要 2 个时钟周期。如果 30%指令每条只需要一次存储器存取,而另外 5%每条需要两次存储器存取,还假定已知混合程序的指令数不变,并与原工作站兼容,试求改进后的处理机性能。

12. 假设在一台 40MHz 处理机上运行 200 000 条指令的目标代码,程序主要由 4 种指令组成。根据程序跟踪实验结果,已知指令混合比和每种指令所需的指令数,如表 1.12 所示。

表 1.12

指 令 类 型	CPI	指令混合百分比
算术和逻辑运算	1	60%
高速缓存命中的加载/存储	2	18%
转移	4	12%
高速缓存失效的存储器访问	8	10%

(1)计算在单处理机上用上述数据运行程序的平均 CPI。

(2)根据(1)所得 CPI,计算相应的 MIPS 速率。

13. 假设高速缓存 Cache 工作速度为主存的 5 倍,且 Cache 被访问命中的概率为 90%,则采用 Cache 后,能使整个存储系统获得多高的加速比?

14. 假定一台具有理想 Cache(无 Cache 失效)计算机的有关数据如表 1.13 所示。其中 Cache 的失效率对指令来说为 5%,对数据访问来说为 10%,并且 Cache 失效导致的失效开销为 40 个时钟周期。求出当有 Cache 失效时各种类型指令的 CPI。

表 1.13

指令类型	频率	周期数	指令访问	数据访问
ALU	43%	1	1	0

LOAD	21%	2	1	1
STORE	12%	2	1	1
BRANCH	24%	2	1	0

15. 假定利用增加向量处理模块来提高计算机的运算速度。计算机处理向量的速度比其通常的运算要快 20 倍。将可用向量处理部分所花费的时间占总时间的百分比称为可向量化百分比。

(1)求出加速比 S 和可向量化百分比 F 之间的关系式。

(2)当要得到加速比为 2 时的可向量化百分比 F 为多少？

(3)为了获得在向量模式所得到的最大加速比的一半，可向量化百分比 F 为多少？

16. 已知 4 个程序在三台计算机上的执行时间(s)，如表 1 .14 所示。

表 1 .14

程　　序	执　行　时　间		
	计算机 A	计算机 B	计算机 C
程序 1	1	10	20
程序 2	1000	100	20
程序 3	500	1000	50
程序 4	100	800	100

假设 4 个程序中每一个都有 100 000 000 条指令要执行，计算这三台计算机中每台机器上每个程序的 MIPS 速率。根据这些速率值，能否得出有关三台计算机相对性能的明确结论？能否找到一种将它们统计排序的方法？试说明理由。

17. 想在系列机中发展一种新型号机器，你认为下列哪些设想是可以考虑的，哪些则是不行的？为什么？

(1)新增加字符数据类型和若干条字符处理指令，以支持事务处理程序的编译。

(2)为增强中断处理功能，将中断分级由原来的 4 级增加到 5 级，并重新调整中断响应的优先次序。

(3)在 CPU 和主存之间增设 Cache 存储器，以克服因主存访问速率过低而造成的系统性能瓶颈。

(4)为解决计算误差较大，将机器中浮点数的下溢处理方法由原来的恒置“1”法，改为增设用只读存储器存放下溢处理结果的查表舍入法。

(5)为增加寻址灵活性和减少平均指令字长，将原来全部采用等长操作码的指令

改成有 3 类不同码长的扩展操作码,并将源操作数寻址方式由原来的操作码指明改成增加一个如 VAX-11 那样的寻址方式位字段来指明。

(6)将 CPU 与主存之间的数据通路宽度由 16 位扩展成 32 位,以加快主机内部信息的传送。

(7)为了减少使用公用总线的冲突,将单总线改为双总线。

(8)把原来的 0 号通用寄存器改为专用的堆栈指示器。

18. 设计指令存储器有两种不同方案:一种是采用价格较贵的高速存储器芯片,另一种是采用价格便宜的低速存储芯片。采用后一种方案时,用同样的经费可使存储器总线带宽加倍,从而每隔 2 个时钟周期就可取出 2 条指令(每条指令为单字长 32 位);而采用前一种方案时,每个时钟周期存储器总线仅取出 1 条单字长指令。由于访存空间局部性原理,当取出 2 个指令字时,通常这 2 个指令字都要使用,但仍有 25%的时钟周期,取出的 2 个指令字中仅有 1 个指令字是有用的。试问采用这两种实现方案所构成的存储器带宽是多少?

第二章 数据表示与指令系统

指令系统是计算机系统中软件与硬件分界面的一个主要标志,它历来是计算机体系结构设计者、系统软件设计者和硬件设计者共同关注的问题。在计算机系统的设计过程中,指令系统的设计是非常关键的,它必须由软件设计人员和硬件设计人员共同完成。

本章主要介绍指令系统及与指令系统直接相关的数据表示和寻址方式等方面的内容,重点讨论在指令系统属性上的传统机器级界面及软、硬件功能分配,以及缩小与软件之间的语义差距的改进措施。在此基础上给出一种指令集结构的实例——DLX 指令集结构。

2.1 浮点数据表示和 IEEE754 标准

2.1.1 数据表示、数据类型、数据结构的关系

计算机中所使用的数据一般可分为三类:第一类是用户定义的数据,这类数据通常是由程序设计语言所确定的;第二类是系统数据,它是程序在执行时由计算机系统蕴含生成的;第三类是指令,即被执行的程序可看成是数据的复合。这里讲的是“数据”,那么“数据类型”是什么呢?

数据类型不同于数据,它除了指一组值的集合外,还定义了可作用于这个集合上的操作集,比如有一组整数值的集合,连同定义在这个集合上可进行的加减乘除等算术操作,这个整数的集合就成为了整数数据类型。计算机中定义数据类型的目的是为了防止不同数据类型间的误操作。从系统结构的观点来看,数据类型可分为基本数据、结构数据、访问指针和抽象数据等类型。我们这里讲的主要是基本数据和结构数据。

基本数据类型一般包括二进制位及其位串、整数及自然数、实数(浮点数)、字符和布尔数等。可以说所有系统结构都支持基本数据类型。结构数据类型是一组由相互有关的数据元素复合而成的数据类型,这些数据元素可以是基本数据类型中的元素,也可以是结构数据类型本身中的元素。也就是说这些数据是有结构的,它们的结构在编译时就被确定,在执行过程中是不允许改变的。如向量和数组、字符串、堆栈、队列、记录等,结构数据类型中的元素不一定都具有相同类型。如向量和数组中的元

素都具有相同类型,而记录中的数据元素往往具有不同的类型。大多数系统结构只能部分地支持结构数据类型。

计算机系统结构中的数据表示指的就是机器硬件能直接识别和引用的数据类型。这里讲到硬件能够直接识别,就是说在系统中能够直接由硬件实现相应数据的运算,也就是系统结构中要有相应的运算指令和运算部件来完成这项任务。那么怎样表示才能让硬件识别某种数据类型?这就涉及到软硬件的交界面了。数据结构所研究的是软的方面,而数据表示考虑的是硬的方面,让计算机能够识别处理,并尽量节约存储空间。

数据结构就是指前面所讲的结构数据类型的组织方式,它反映了结构数据类型中各种数据元素或信息单元之间的结构关系,它是面向应用和软件的,如串、栈、队列、向量、树、图等线性和非线性的数据结构,它们反映了实际应用中各种数据元素或信息单元之间的联系。比如树这种数据结构,里面的元素就有根和叶的层次逻辑关系。数据结构一般是通过高级语言描述建立的,但是计算机硬件并不懂什么是根,什么是叶,它只认 0 和 1。这就需要我们确定如何在计算机系统中进行数据表示,让硬件能认识各种数据类型。所以说数据表示是数据结构的一个基础子集合,数据结构通过一定的算法变成数据表示才能在系统中处理,不同的数据表示对数据结构提供不同程度的支持,它关系到数据结构的实现效率及方便性,它们的差距是高级语言与机器语言的语义差距,数据结构课程的研究范围就是填补这一差距的算法和映像。

关键在于确定哪些数据类型用数据表示实现,哪些数据类型采用数据结构来实现。这本质上是一个软硬件取舍的问题。引入数据表示的原则主要有两个:一是看系统效率是否提高,这主要体现在程序的执行时间和目标代码所占的存储空间是否减少。二是看该数据表示的通用性和利用率是否高,否则会因为硬件成本的增加而降低性能价格比。

如要实现向量操作 $A = A + B$, A 和 B 都是 200×200 的矩阵。如果在没有向量数据表示的计算机系统上实现,一般需要 6 条指令,其中有 4 条指令要循环 40 000 次,但若在有向量数据表示的机器上实现,只需要 1 条“向量加”指令即可,这样,在访存和处理机之间的信息传送量仅取指令就减少了 $4 \times 40\ 000 = 160\ 000$ 个字。这当然会使实现时间大大减少,而且向量运算部件的流水运算也可节省时间,同时可使辅助开销时间减少(如判越界,同时使编译简单,主存空间用量节省),所以说向量指令既减少了指令存储空间,又减少了主存和处理机之间信息量的传送,加快了程序的执行。

如果引入某种数据表示以后,只对某种数据结构的实现效率很高,而对其他数据结构的实现效率很低,或者引入这种数据表示在应用中很少用到,那么为此所花的硬件代价过多却并未在性能上得到好处,必然导致性能价格比的下降,特别是对一些复杂的数据表示。例如,引入具有树形数据表示的所谓树结构式机器,增加的硬件对树数据结构的实现是高效的,但对堆栈、向量、链表等其他数据结构的实现却是低效的。

但若用指针实现树数据结构,尽管效率没有树形数据表示的那么高,但它能同时比较高效地实现树、向量、链表、栈、图等多种数据结构,花的代价不大,但实现数据结构的适应性、通用性强,有利于机器性能价格比的提高。

一般来说,一些简单的、常用的、通用的数据类型采用数据表示,如 int, float, Boolean, String, stack 等。而复杂的数据结构一般通过数据结构实现或通过软硬件联合设计实现。如 table, Graph, Tree 等。当前的发展趋势是,除基本数据表示外,还不断引入一些复杂数据表示,给予数据结构更多的支持。如在目前的计算机系统中,字符串数据表示、向量数据表示、堆栈数据表示等已经普遍使用,有些很复杂的数据表示,如图、表等数据表示也开始在某些计算机系统中出现。

另外,对于一些复杂的数据类型,如果用数据表示来实现,硬件的代价可能非常大,然而,如果用硬件给予适当的支持,或者说,用软件和硬件相结合的方法来实现,效果会很好。例如,用字节编址和字节运算指令来支持字符串数据表示,用变址寻址方式来支持向量数据表示等。

因此,在设计计算机系统时,对于数据类型,系统结构设计者首先要做的是:确定哪些数据类型全部用硬件实现,即数据表示,哪些数据类型用软件实现,即数据结构,哪些数据类型可由硬件给予适当的支持,即由软件和硬件共同实现,并确定软件与硬件的适当比例关系。

对于一些基本数据表示,也有一些问题是要精心设计的,如浮点数的尾数基值的选取,上、下溢处理方法的选择等。下面主要讨论浮点数据表示中尾数基值的选取对浮点数的影响。

2.1.2 浮点数据表示

1 .问题的提出

早期的计算机只有定点数据表示法,这种计算机的优点是硬件结构简单,但存在以下不足:

(1) 编程困难,程序设计人员要做大量的数据规格化工作。即编程人员首先要把参加运算的数据扩大或缩小某一个倍数后送入机器,等运算结果出来后再恢复到正确的数据。

(2) 数据的表示范围小,要表示两个大小相差很大的数据,需要很长的机器字长。例如,16 位字长的表示范围为[- 32768,32767],如要表示相差为 10^{61} 以上的数,即 $2^X > 10^{61}$,解为 $X > 203$ 位,如再要求精度不低于 10 进制 7 位,则有:$2^{-X} < 10^{-7}$,解为:$X > 23$ 位,这样定点数表示总共需要:203 + 23 = 230 位。

(3) 数据存储空间的利用率低。例如,为了把小数点的位置定在数据最高位前面,必须把所有参与运算的数据至少都除以这些数据中的最大数,只有这样才能把所有数据都化成纯小数,因而会造成很多数据有大量的前置零,从而浪费了许多数据存

储单元。为此,引入了浮点数据表示法。

浮点数表示方式研究的核心内容是数据字长与这种数据表示方式下的表数范围、表数精度和表数效率三者之间的关系。在数据字长确定的情况下,研究各种浮点数表示方式下的表数范围、表数精度和表数效率及它们三者之间的关系,其目的是找到具有最大表数范围、最高表数精度和最优表数效率的浮点数表示方式。

2 .浮点数的表示范围

对任意浮点数 N,可表示为: $N = m \times r_m^e$ 的形式,其中: $e = r_e^q$;

一种浮点数据表示方式需要 6 个参数来定义,分别为:

m: 尾数,多数机器中采用规格化小数表示,只有少数机器采用整数表示;

e: 阶码的值,一般采用整数、移码表示,只有少数采用补码表示;

r_m : 尾数的基,一般采用二进制、四进制、八进制、十六进制和十进制等;

r_e : 阶码的基,在目前见到的所有浮点数据表示方式中,均为 2;

p: 尾数长度(不包括符号位),不是指尾数的二进制位数,如当 $r_m = 16$ 时,每 4 个二进制位表示一位尾数;

q: 阶码长度,由于 $r_e = 2$,所以,在一般情况下, q 就是阶码部分的二进制位数。

一种浮点数表示方式,即在数据存储单元中的存放方式,如图 2 .1 所示,其中, m_f 表示尾数的符号, e_f 表示阶码的符号。

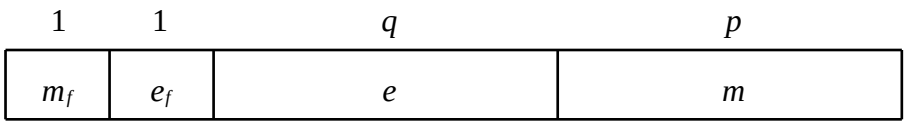


图 2 .1 浮点数的表示方式

在尾数采用原码、纯小数,阶码采用移码、整数的浮点数表示方式中,规格化浮点数 N 的表数范围,如表 2 .1 所示。

表 2 .1 尾数用原码、纯小数时规格化浮点数的表数范围

表数范围	规格化尾数	阶码	规格化浮点数
最大正数(N_{max})	$1 - r_m^{-p}$	$r_e^q - 1$	$(1 - r_m^{-p}) \times r_m^{r_e^q - 1}$
最小正数(N_{min})	r_m^{-1}	—	$r_m^{-1} \times r_m^{-r_e^q}$
最大负数($-N_{max}$)	$-r_m^{-1}$	—	$-r_m^{-1} \times r_m^{-r_e^q}$
最小负数($-N_{min}$)	$-(1 - r_m^{-p})$	$-r_e^q$	$-(1 - r_m^{-p}) \times r_m^{r_e^q - 1}$

表数范围	$r_m^{-1} \times r_m^{-r_e^q} \leq \lvert N \rvert \leq (1 - r_m^{-p}) \times r_m^{r_e^q - 1}$

当尾数用补码表示时,正数区间的表数范围与尾数采用原码时完全相同,而负数区间的表数范围为: $-r_m^{r_e^q - 1} \leq N \leq -(r_m^{-1} + r_m^{-p}) \times r_m^{-r_e^q}$,如表 2.2 所示。

表 2.2　　尾数用补码、纯小数时规格化浮点数的表数范围

表数范围	规格化尾数	阶码	规格化浮点数
最大正数(N_{max})	$1 - r_m^{-p}$	$r_e^q - 1$	$(1 - r_m^{-p}) \times r_m^{r_e^q - 1}$
最小正数(N_{min})	r_m^{-1}	—	$r_m^{-1} \times r_m^{-r_e^q}$
最大负数($-N_{max}$)	$-(r_m^{-1} + r_m^{-p})$	—	$-(r_m^{-1} + r_m^{-p}) \times r_m^{-r_e^q}$
最小负数($-N_{min}$)	-1	$-r_e^q$	$-r_m^{r_e^q - 1}$

例如,尾数用补码、纯小数表示,阶码用移码、整数表示, $p = 6$, $q = 6$, $r_m = 16$, $r_e = 2$,规格化浮点数 N 在正数区间的表数范围是:

$$\frac{1}{16}16^{-2^6} \leq N \leq (1 - 16^{-6}) \times 16^{2^6 - 1}$$

在负数区间的表数范围是: $-16^{63} \leq N \leq -(\frac{1}{16} + 16^{-6}) \times 16^{-64}$

在浮点数表示方式中尾数采用规格化小数的目的是为了在尾数中表示最多的有效数据位及数据表示的惟一性。如:

$0.003475 - - - > 0.3475 * 10^{-2}$

从上面的分析中可以看到,规格化浮点数的表数范围主要与阶码的长度 q 和尾数的基值 r_m 有关。这时,能表示的绝对值最大的浮点数可近似为:

$$\lvert N_{max} \rvert = r_m^{r_e^q} \qquad (2.1)$$

可以看出,表数范围随着 q 和 r_m 的增加而扩大。

3.浮点数的表数精度

表数精度也称为表数误差,浮点数存在表数精度的根本原因是由于浮点数的不连续性造成的。由于在上述表示方法下,浮点数能够表示数据的个数最多为: 2^{p+q+2},而实际数据(实数)的变化是连续的,即数的个数是无限的。因此,浮点数表示的仅仅是实数的一个子集,称为浮点数集 F。

令 N 为实际要表示的实数,而 M 是 F 中最接近 N 的浮点数,且被用来代替 N 的浮点数,则浮点数表数的绝对误差为:

$$\delta = | M - N |$$

而相对表数误差为:

$$\delta = \left| \frac{M - N}{N} \right|$$

产生表数误差的原因有两个,一是假设两个数 $a, b \in F$,但 a 和 b 运算后不一定在 F 集内,设 $c = a \text{ op } b$,但 $c \notin F$,因此只好用数 $c' \in F$ 来表示数 c,因此可能造成表数的绝对误差和相对误差。另一个是原始数据从外部进入计算机的过程中,通常要将数据从十进制转换成二进制、四进制、八进制等,而这一转换过程就有可能产生表数误差。如遇到循环小数时,用有限长度的尾数是不能精确表示循环小数的。

如一种浮点数表示方式的 $q = 1$, $p = 2$, $r_m = 2$, $r_e = 2$,尾数用原码、纯小数表示,阶码用移码、整数表示,则所能表示的浮点数,如表 2.3 所示。

表 2.3 一种浮点数表示方式所能表示的全部浮点数

尾数 \ 阶码		阶码($q=1$, $r_e=2$)			
		1(1 1)	0(1 0)	-1(0 1)	-2(0 0)
尾数 $p=2$ $r_m=2$	0.75(0.11)	3/2	3/4	3/8	3/16
	0.5(0.10)	1	1/2	1/4	1/8
	-0.5(1.10)	-1	-1/2	-1/4	-1/8
	-0.75(1.11)	-3/2	-3/4	-3/8	-3/16

若有两个浮点数:$a_1 = 1/2$, $b_1 = 3/4$ 都在所定义的浮点集内,而它们相加的结果 $a_1 + b_1 = 5/4$,则不在这个浮点数集内,为此必须用该浮点数集内的最靠近 5/4 的浮点数 1 或 3/2 来代替,从而产生绝对表数误差:

$$\delta = |5/4 - 3/2| = 1/4 \quad \text{或} \quad \delta = |5/4 - 1| = 1/4$$

相对表数误差为:

$$\delta = \left| \frac{5/4 - 3/2}{5/4} \right| = 1/5 \quad \text{或} \quad \delta = \left| \frac{5/4 - 1}{5/4} \right| = 1/5$$

由于浮点数的尾数有效位长度是确定的,因此,对规格化浮点数而言,相对误差是确定的,而绝对误差是不确定的。

又如把十进制数 0.1 输入二进制的计算机中,会出现循环小数,即:

$$0.1_{(10)} = 0.000110011001100\cdots_{(2)}$$
$$= 0.0121212\cdots_{(4)}$$
$$= 0.0614631\cdots_{(8)}$$

$= 0.1999\cdots_{(16)}$

由于物理存储空间的有限和数据表示的无限是矛盾的,只好采用舍入的方法,这些都会造成浮点数的表数误差。

实际上尾数的基 r_m 需要用 m' 个二进制位表示,具体如下:

$$m' = \lceil \log_2 r_m \rceil$$

例如,十进制数下的 $m' = 4$,表示一个十进制数位要用 4 个二进制位表示。因此,尾数 m 的实际数位 k 为: $k = p \times m'$,这样,浮点数的表示方法如图 2.2 所示。

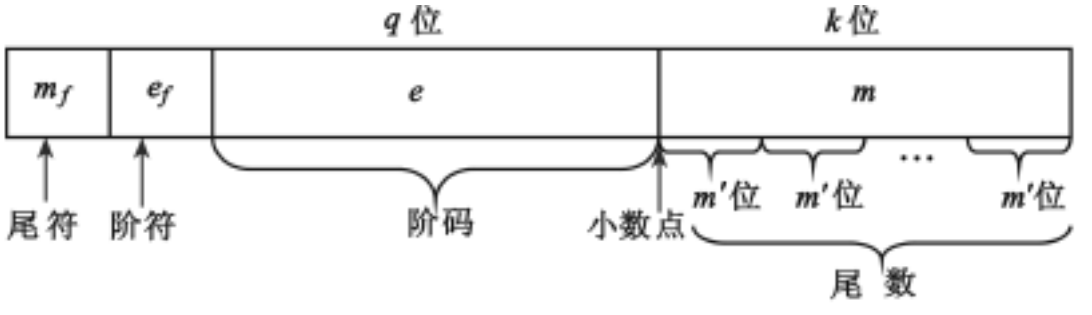


图 2.2 浮点数的实际表示方式

由以上分析可知,规格化浮点数的表数精度主要与尾数基值和尾数长度有关,在一般情况下,认为规格化尾数最后 1 位的精确度是一半,这样可以定义表数精度如(2.2)式所示:

$$\delta(r_m, p) = \frac{1}{2} r_m^{-(p-1)} \tag{2.2}$$

当 $r_m = 2$ 时,有 $\delta(2) = \frac{1}{2} \times 2^{-(p-1)} = 2^{-p}$ (2.3)

这样,可以得出,当用 p_2 个二进制位表示尾数 r_m 时, r_m 的实数位为:

$p = \left\lceil \frac{p_2}{\log_2 r_m} \right\rceil$, $r_m = 2^{\log_2 r_m}$,代入(2.2)式,有:

$$\delta(r_m) = 2^{\lceil \log_2 r_m \rceil - 1} \times 2^{-p_2} \tag{2.4}$$

比较式(2.4)和式(2.3),由于式(2.4)中的 p_2 就是式(2.3)中的 p,所以可得出如下结论:

当浮点的尾数长度(P_2 值)相同时,尾数取值 $r_m = 2$ 具有最高的表数精度。当尾数取基值 $r_m > 2$ 时,浮点数的精度和 $r_m = 2$ 相比将损失 $2^{\lceil \log_2 r_m \rceil - 1}$ 倍,即相当于尾数减少了 $\lceil \log_2 r_m \rceil - 1$ 个二进制位。

实际上,得出上述结论是很显然的,因为,当尾数基值 $r_m = 2$ 时,规格化浮点数的尾数肯定没有前置 0,全部尾数都是有效位,而当尾数基值 $r_m > 2$ 时,规格化浮点数的尾数最多可能有 $\lceil \log_2 r_m \rceil - 1$ 个前置 0,所以,当尾数 $r_m > 2$ 时,浮点数的表数精度与 $r_m = 2$ 相比将损失 $2^{\lceil \log_2 r_m \rceil - 1}$ 倍。

例如,当尾数基值 $r_m = 16$ 时,规格化浮点数的尾数可能有 3 个前置 0,因此其表

数精度与 $r_m = 2$ 时相比将损失 2^3 倍,即相当于尾数减少了 3 个二进制位。

4. 浮点数的表数效率

通常把最高位为非 0 的浮点数称为规格化浮点数。因此,规格化尾数 x 值可表示为:$1/\ r_m < |\ x| < 1$(0 除外)。因机器零的尾数和阶码都是 0,所以它也是一个规格化浮点数。

同一种浮点数表示方式中,规格化浮点数具有最长的尾数有效位数,所以,规格化浮点数的表数精度是最高的。

在采用浮点数表示方式的计算机中,一般规定存放在存储部件中的浮点数、进入运算部件的浮点数、从运算部件中输出的浮点数都必须是规格化浮点数,只有在运算过程中才可能出现非规格化的浮点数。浮点数的表数效率定义为:

$$\eta = \frac{\text{可表示的规格化浮点数}}{\text{全部浮点数个数}} = \frac{2 \times (r_m - 1) \times r_m^{p-1} \times 2 \times r_e^q + 1}{2 \times r_m^p \times 2 \times r_e^q} \approx \frac{r_m - 1}{r_m} \tag{2.5}$$

在上式的分子分母中,第一个 2 表示尾数的正负数各半,第二个 2 表示由于阶码的符号位使编码种数增加 1 倍。分子中的最后一个 1 表示机器 0。

从式(2.5)可以看出,浮点数的表数效率主要与尾数的基值有关。

例如,$r_m = 2$ 时,$\eta(2) = 50\%$

$r_m = 16$ 时,$\eta(16) = (16 - 1)/\ 16 = 93.75\%$

尾数基值 $r_m = 2$ 与 $r_m = 16$ 相比,浮点数的表数效率提高了:

$$T = \frac{\eta(16)}{\eta(2)} = 1.875(\text{倍})$$

5. 浮点数尾数基值的选取

从上面的分析中可以看到,浮点数的表数范围、表数精度和表数效率三个主要特征都与尾数的基值 r_m 有关,因此,这里主要讨论在浮点数表示方法中,尾数基值 r_m 应当如何选取。

尾数基值 r_m 的选择主要考虑两个因素:一是在机器字长一定的情况下,即 $p + q + 2$ 为定值时,如何使表数范围最大、表数精度和表数效率最高;二是在尾数基值确定之后,如何根据表数范围和精度的要求,确定 p 和 q 值。下面主要讨论第一种因素。

从式(2.1)中看到,浮点数的表数范围主要与阶码的长度 q 和尾数基值 r_m 有关,从式(2.2)中看到,表数精度主要与尾数长度 p 和尾数基值 r_m 有关。现假设两种浮点数表示方式 F1 和 F2,它们的二进制字长相同,尾数都用原码或补码、小数表示,阶码都用移码、整数表示,阶码的基值均为 2,而尾数的基值不同。

设浮点数表示方式 F1:尾数基值 $r_{m1} = 2$,尾数长度 p_1,阶码长度 q_1,二进制字

长: $L_1 = p_1 + q_1 + 2$

浮点数表示方式 F2: 尾数基值 $r_{m2} = 2^k$, 尾数长度 p_2, 阶码长度 q_2, 二进制字长: $L_2 = kp_2 + q_2 + 2$

(1)当 $L_1 = L_2$, 且 $|N_{1\max}| = |N_{2\max}|$ 时, 分析尾数基值和精度的关系:

分析: 由于 $|N_{1\max}| = |N_{2\max}|$, 且 $|N_{1\max}| = 2^{2^{q_1}}$, $|N_{2\max}| = (2^k)^{2^{q_2}} = 2^{k\times 2^{q_2}}$, 所以:

$$2^{q_1} = k \times 2^{q_2}, \text{即: } q_1 = q_2 + \log_2 k \tag{2.6}$$

由于 $L_1 = L_2$, 即 $p_1 + q_1 = kp_2 + q_2$ (2.7)

将式(2.6)代入式(2.7)得:

$$p_1 = kp_2 - \log_2 k \tag{2.8}$$

由式(2.2)可知, F1 的表数精度是:

$$\delta(2, p_1) = \frac{1}{2} \times 2^{1-p_1} \tag{2.9}$$

把式(2.8)代入式(2.9)得:

$$\delta(2, p_1) = \frac{1}{2} \times 2^{1-kp_2+\log_2 k} \tag{2.10}$$

F2 的表数精度是:

$$\delta(2^k, p_2) = \frac{1}{2} \times 2^{k(1-p_2)} \tag{2.11}$$

为了考察 F1 与 F2 的表数精度之间的关系, 取它们的比值:

$$T = \frac{\delta}{\delta} = 2^{k-\log_2 k-1} \tag{2.12}$$

由式(2.12)可以得出, 只有当 $k=1$(表示 $r_m = 2$)或 $k=2$(表示 $r_m = 4$)时, 比值 $T=1$, 否则 $T>1$。这说明:

① 当 $k>2$ 时, 即 $r_m > 4$ 时, F2 的表数精度将低于 F1。

② 在浮点数的字长和表数范围一定时, 尾数基值取 2 或 4 具有最高的表数精度

(2)分析在浮点数的字长和表数精度一定时, 尾数基值与表数范围之间的关系。即 L 和 δ 一定的情况下, r_m 和 $|N_{\max}|$ 间的关系?

由 F1 与 F2 的表数精度相同得到:

$\frac{1}{2} \times 2^{1-p_1} = \frac{1}{2} \times (2^k)^{1-p_2}$, 即:

$$p_1 = kp_2 - k + 1 \tag{2.13}$$

把式(2.13)代入式(2.7)得到:

$$q_1 = q_2 + k - 1 \tag{2.14}$$

则 F1 的表数范围: $|N_{1\max}| = 2^{2^{q_1}} = 2^{2^{q_2+k-1}} = 2^{2^{q_2}\times 2^{k-1}}$

F2 的表数范围：$\left| N_{2\max} \right| = (2^k)^{2^{q_2}} = 2^{2^{q_2} \times k}$

现假设 F2 的表数范围大于 F1 的表数范围，则 F2 的阶码的最大值要大于 F1 的阶码的最大值：

$$2^{q_2} \times k > 2^{q_2} \times 2^{k-1}$$

即：

$$k > 2^{k-1} \tag{2.15}$$

式(2.15)在整数定义域内是无解的，这表明：

① 不存在比 F1 的表数范围更大的浮点数表示方式。

② 当 $k=1(r_m=2)$或 $k=2(r_m=4)$时，$\left| N_{1\max} \right| = \left| N_{2\max} \right|$。

由以上(1)，(2)的分析，可以得出如下结论：

当浮点数字长确定后，r_m 取 2 或 4 时，具有最大的表数范围和最高的表数精度。

实际上发生这种情况的根本原因是：当浮点数的尾数基值从 $r_m=2$ 增大时，表数精度是按尾数字长的指数关系变小，而表数范围却按阶码字长的指数的指数关系变大，因此，为了保持同样的表数精度和表数范围，损失的尾数字长大于节省的阶码字长，从而造成浮点数的总字长增加。

上面的结论说明，从浮点数的表数范围和表数精度看，尾数基值取 2 或 4 时最好，但由于 $r_m=2$ 时，表数效率却最低，只有 50%，为了提高表数效率，许多计算机系统采用了隐藏位表数法。其主要原因是当 $r_m=2$，且采用规格化方法时，尾数 m 的最高位一定为 1，故可以隐藏或省去，此时表数效率 $\eta=100\%$，但 $r_m=4$ 时，就不能采用隐藏位表数法，因为尾数可从 00，01，10，11 中取值。

由上述综合分析可以得出：当 $r_m=2$ 时，且采用隐藏位表数法时，能使浮点数的表数范围最大，表数精度最高，表示效率最优。

IBM 公司的 IBM 360 系列机、IBM 370 系列机、IBM 4300 系列机等，浮点数的尾数基值是 16。Burroughs 公司的 B6700，B7700 等大型计算机，浮点数的尾数基值取的是 8。DEC 公司的 VAX-11 和 Alphs 等计算机，CDC 公司的 CDC6600，CYBER70 等大型计算机及应用最广泛的 Intel 公司的 X86 系列机等的浮点数均采用尾数基值 2。浮点数尾数基值不取 2 的原因可能是早期计算机设计人员对浮点数的研究不够深入，后来虽然弄明白了尾数基值取 2 最好，但是为了系列计算机具有兼容性，也只能如此了。

2.1.3 IEEE754 标准浮点数表示

二进制浮点数的表示，由于不同机器所选用的基值、尾数位长度和阶码位长度不同，因此对浮点数表示有较大差别，这就不利于软件在不同计算机间的移植。美国 IEEE(电气及电子工程师协会)为此提出了一个从系统结构角度支持浮点数的表示方法，称为 IEEE 标准 754(1985)，它是一种优化表示法，当今流行的计算机几乎都采

用了这一标准。

IEEE754 标准在表示浮点数时,每个浮点数均由三部分组成:符号位 S,指数部分 E 和尾数部分 M。

浮点数一般采用以下四种基本格式:

(1) 单精度格式(32 位):除去符号位 1 位后,E = 8 位,M = 23 位。

(2) 扩展单精度格式:E≥11 位,M≥31 位。

(3) 双精度格式(64 位):E = 11 位,M = 52 位。

(4) 扩展双精度格式:E≥15 位,M≥63 位。

在 IEEE754 标准中,约定小数点左边隐含一位,通常这位数就是 1,这样实际上使尾数的有效位数为 24 位(单精度),即尾数为 1 .M。指数的值在这里称为阶码,为了表示指数的正负,所以阶码部分采用移码表示,移码值为 127,阶码值即从 1 到 254 变为 - 126 至 + 127,在 IEEE754 标准中所有的数字位都得到了使用,明确地表示了无穷大和 0,并且还引进了“非规格化数”,使得绝对值较小的数得到更准确的表示。

IEEE754 标准的单精度和双精度浮点数表示格式,如图 2 3 所示。

1 位 S	指数 E(8 位)	尾数 M(23 位)

(a)单精度

1 位 S	指数 E(11 位)	尾数 M(52 位)

(b)双精度

图 2 3 IEEE754 标准浮点数表示格式

其中,阶码值 0 和 255 分别用来表示特殊数值:当阶码值为 255 时,若分数部分为 0,则表示无穷大;若分数部分不为 0,则认为这是一个非数值 NaN(Not a Number)。当阶码和尾数均为 0 时,则表示该值为 0,因为非零数的有效位总是大于等于 1,因此特殊约定其表示为 0。

概括起来,由 32 位单精度所表示的 IEEE754 标准浮点数 N 的解释,如表 2 4 所示。

表 2 4 IEEE754 单精度浮点数的表示方法

S(1 位)	E(8 位)	M(23 位)	N(共 32 位)
符号位	0	0	0
符号位	0	不等于 0	$(-1)^S \times 2^{-126} \times (0.M)$ 为非规格化数,(0 为隐藏位)
符号位	1 到 254 之间	不等于 0	$(-1)^S \times 2^{E-127} \times (1.M)$ 为规格化数,(1 为隐藏位)
符号位	255	不等于 0	NaN(非数值)
符号位	255	0	$(-1)^S \infty$(无穷大)

注意当数字 N 为非规格化数或是 0 时,隐含位是 0。

由此可见,IEEE754 标准使 0 有了精确表示,同时也明确地表示了无穷大,所以,当 a/ 0(a 不等于 0)时得到结果值为±∞;当 0/ 0 时得到结果值为 NaN。对于绝对值较小的数,为了避免下溢而损失精度,允许采用比最小规格化数还要小的数来表示,这些数称为非规格化数(Denormal number),应注意的是,非规格化数和正、负零的隐含位值是 0,而不是 1。

IEEE754 标准单、双精度浮点数的特征如表 2.5 所示。

表 2.5 IEEE754 标准单、双精度浮点数的特征

	单精度	双精度
符号位	1	1
指数位	8	11
尾数位	23	52
总位数	32	64
指数系统	移码 127	移码 1023
指数取值范围	- 126～ + 127	- 1022～ + 1023
最小规格化数	2^{-126}	2^{-1022}
最大规格化数	2^{+128}	2^{+1024}
十进制数范围	10^{-38}～ + 10^{+38}	10^{-308}～ + 10^{+308}
最小非规格化数	10^{-45}	10^{-324}

下面举两个例子来说明 IEEE754 标准浮点数的表示:

(1) N = - 1.5,它的单精度格式表示为:

1 01111111 10000000000000000000000

其中,S = 1,E = 127,M = 0.5,因此 $N = (-1)^1 \times 2^{127-127} \times (1.5) = -1.5$

(2) 以下的 32 位数所表示的单精度浮点数为:

1 10000001 01000000000000000000000

其中,S = 1, E = 129, M = 0.25, 由公式 $N = (-1)^1 \times 2^{129-129} \times (1.25) = -2^2 \times 1.25 = -5$

2.2 高级数据表示

一般可把数据表示分为两大类:基本数据表示和高级数据表示。基本数据表示包括定点数据表示、浮点数据表示、逻辑数据表示、十进制数据表示等;而把自定义数

据表示、向量数据表示、堆栈数据表示等归纳为高级数据表示一类。第一节已经介绍了浮点数这种基本数据表示方法,这节主要了解在机器中不断引入的一些高级数据表示对数据结构提供的支持等。

2.2.1 自定义数据表示

高级语言中数据运算表达式是统一的,而数据类型是用专门的类型说明语句来说明的,但在机器语言中对同一运算要针对不同的数据类型使用不同的指令。即对于目前的大多数计算机,数据存储单元(如寄存器、主存、外存等)里存放的都是纯数据,而对这些数据的类型(如定点数、浮点数、复数、字符串、逻辑数等)、进位制(如二进制、十进制、十六进制)、字长(如字、半字、双字等)、寻址方式、功能(如地址、数值、控制字、标志等)等的解释要通过指令中的操作码来进行。如 IBM 370 的加法指令就有 8 条,这些指令所用到的操作类型、字长、进位制和操作数所采用的寻址方式等均不相同。又如 Intel86X 中的乘法指令也有 Mul,Imul 两种。

为了缩小这一语义差距,减轻编译软件工件量,在有些计算机中引进带数据类型标志的自定义数据表示,这样就可以把不同类型的同种运算指令统一起来,使每种指令的种类大为减少,称为通用化指令(即操作说明与数据类型无关)。

自定义数据表示包括带标志符的数据表示和数据描述符二类。下面分别介绍这两种数据表示方式。

1.带标志符的数据表示(Tagged data representation)

带标志符的数据表示指的是用以定义某个数据的数据类型和数值的数据表示方法。格式如图 2.4 所示。

图 2.4 带标志符的数据表示方式

美国 Burroughs 公司在 20 世纪 60 年代初期生产的 B5000 大型计算机中,每个数据有一位用来区分是操作数还是描述符;在 60 年代后期生产的 B75000 大型计算机中,每个数据用三位标志符来区分 8 种数据类型。在 70 年代生产的 R-2 试验性计算机中采用了 10 位标志符,如图 2.5 所示。

图 2.5 中的最高两位用来区分操作数、指令、地址、控制字;两位陷阱位可由软件定义四种捕捉方式,为程序员跟踪程序的执行提供方便;一位封写指定数据是只读的还是可读可写的;四位类型可以在最前面两个功能位定义的基础上进一步定义是二进制、十进制、定点数、浮点数、复数、字符串、单精度、双精度等,如果最前面两位已经定义了是地址,则四个类型位可进一步说明是绝对地址、相对地址、变址地址、未连接

的地址等;最后一位是奇偶检验位。

图 2 5 R-2 计算机中带标志符的数据表示方式

由上可知,带标志符的数据表示,不仅可以定义数据类型,而且还可以用来描述机器中用到的各种有用信息。它通常由系统软件和高级语言的编译器建立,对高级程序员和计算机用户是透明的。

(1) 采用标志符数据表示方法主要有以下几方面的优点:

1) 便于实现一致性检验,硬件自动实现数据转换

在采用标识符的数据表示后,数据类型的一致性检查和转换等都用硬件完成,能快速检测出参加运算的数据是否定义错误(如一个浮点数与一个字符相乘),是否不相容(如一个浮点数与一个定点数相加,两个字长不相等的定点数相加等)。如果发现数据类型定义有错误,则进入出错中断处理程序,如果发现数据类型不相容,则由硬件自动进行数据类型和数据长度的变换,从而缩短了目标程序的长度,减少了占用的空间。因为在 Compiler 中,这种数据一致性检测程序要反复使用,如能通过硬件实现加以优化,则将大大提高程序执行的效率并减小占用空间。

2) 标志符方法减少和简化了指令系统

指令中的操作码只需指出操作种类,不必指出数据类型,指令的条数必然减少很多,而且数据的一些信息也不必再通过指令系统进行描述,因而可简化指令的地址码,简化访问主存的命令、输入输出指令和程序控制指令。

3) 简化了系统程序和编译程序的设计

自动实现数据类型的一致性检验和数据转换,简化了编译程序的设计,而且这种数据表示缩小了人与计算机之间的语义差距,从而使程序设计变得更加容易。

4) 方便程序调试和应用软件开发

通过设置陷阱位 trap 位,可以捕捉指令执行状态。陷阱位可通过软件设置,也可在调试程序时手工设置,这一功能为软件的跟踪和调试提供了极大的方便,必然会缩短软件开发的周期。

5) 支持 DBS 的实现与数据类型无关的需求

即一个软件不加修改就可适用于多种数据类型,标志符数据表示方式正好支持这种要求。在一般计算机系统中要实现软件与数据类型无关是非常困难的。

(2) 对采用带标志符数据表示的主要质疑是:

1) 当采用标志符数据表示时,整个程序的存储空间是增大还是变小了呢

由于每个数据都带有标志符,这必然要加长数据的字长,从而使数据所占用的存

储空间增加。但由于这种表示方法简化了指令系统,指令只需指出操作种类,不需要指出数据类型等,每条指令的字长可以缩短,只要设计合理,整个程序(包括指令和数据)的总存储量反而有可能减少,即使增加也不会增加得太多,如图 2.6 所示。

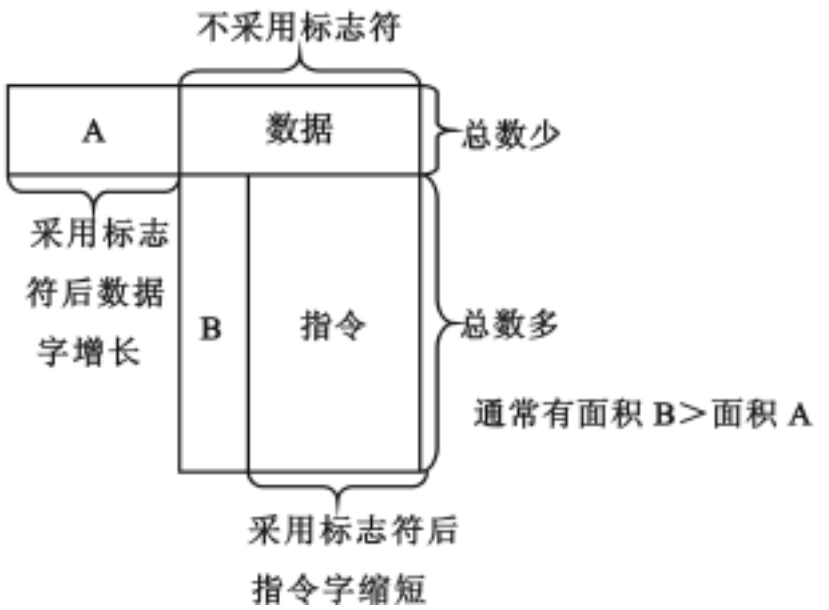


图 2.6　采用标志符表示后可能节省的程序空间

从图 2.6 中可以看出,采用标志符数据表示方法后,总的存储空间的占用是否增加,取决于面积 A 和面积 B 哪一个大。通常情况下,数据字增长所占用存储空间的增加(面积 A)是小于指令字缩短所减少占用的存储空间(面积 B)的,此外,由于带标志符数据表示使编译简化,从而使编译程序所占用的程序空间减小,所以说,总的存储空间一般情况下是减小的。

但在这种情况下,会造成数据和指令的长度可能不一致。解决的方法通常有两种:一种是把数据和指令分别存放在两个存储器中,即采用多体存储方案(第四章中详细介绍);另一种是合理设计数据字长和指令字长,一般的设计原则是指令字长向数据字长靠拢。

2) 指令的执行速度降低

这是因为在指定的执行过程中,要对每个标志符进行逐个解释,并判断数据是否相容,因此将导致指令本身的执行速度降低,但可使程序的宏观速度加快。这是因为程序的宏观执行时间是程序的设计时间、编译时间、调试时间和执行时间的总和。前三项称为宏观速度,执行时间称为微观速度。采用标志符数据表示法提高了宏观速度,而降低了微观速度。

3) 硬件设计的复杂性提高

由于要用硬件实现数据相容性、一致性测试,实现数据类型的自动转换,并解释所有标志符,因此采用标志符数据表示法的计算机系统,其硬件复杂度很大,过去仅在少数大型计算机和巨型计算机中采用,然而,随着 VLSI 技术的迅速发展,硬件复

杂度将不再成为一个大的问题。

带标志符的数据表示在商品化计算机中并不流行,但在需要支持含有动态数据类型的高级语言中(如 LISP 和 PROLOG 的系统结构)应用得较多。美国 Intel 公司生产的 8087 微处理器,日本为研制第五代计算机而生产的个人计算机中都采用了标志符数据表示方法。

综上所述,采用标志符数据表示法虽然硬件复杂度高,机器的微观速度可能降低,但是,由于它能缩短人与计算机之间的语义差距,能减少高级语言与机器语言之间的语义差距,能提高机器的宏观性能,因此这是一种很有前途的数据表示方法,目前有待研究的关键问题是如何定义好标志符,如何进一步扩大标志符的用途等。

2 .数据描述符表示法(Data descriptors)

上面提到的带标志符的数据表示,对每一个数据都需要附加标志位,但在数组中,不一定要对数组中的每个元素都附加标志位,因为数组中各个元素具有相同属性,只需对整个数组配以一个标志符就可以了,从而引出了另一种自定义数据表示,即数据描述符。所以说数据描述符是用于描述复杂和多维结构的数据类型,对于这类数据,可用一个描述符来说明一个同类数据的集合,如向量、矩阵、记录等。

数据描述符的格式如图 2 .7 所示。

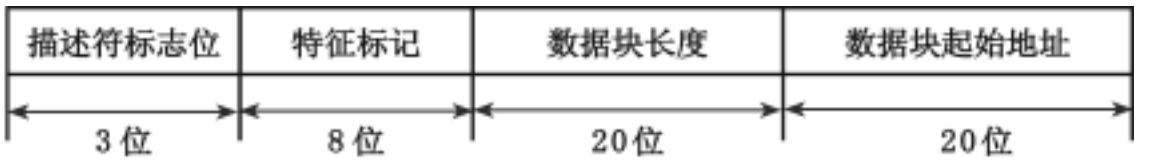


图 2 .7 数据描述符的格式举例

图 2 .7 中描述符标志位部分指明了这是一个数据描述符;特征标记部分指明了所描述数据的各种特征;长度部分指明所描述数据的长度,即数组中元素个数;起始地址部分则指明了所描述的一组数据的首址。

美国 Burroughs 公司率先在它的商品化计算机 B-6700 中采用了数据描述符,格式如图 2 .8 所示。

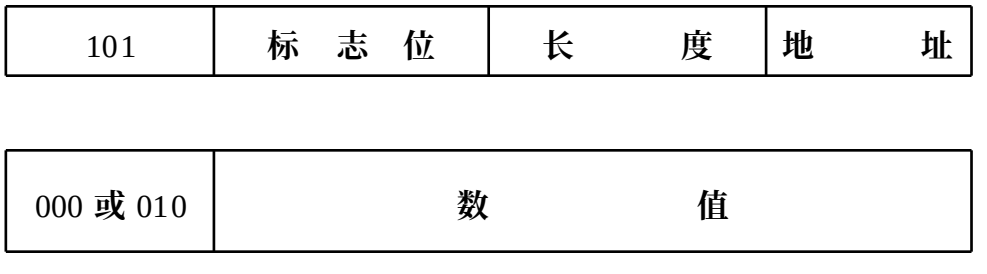


图 2 .8 B-6700 计算机的数据描述符表示法

每个 48 位长的字,附加有 3 位标志符。当标志符为 000 和 010 状态时,分别表明后面 48 位长的字为单精度和双精度数据,此时就成为 51 位长的带标志位的数据字。当标志符为 101 时,就表示该字为描述符。它的后 48 位被分成 3 个字段,前 8 位为特征标志位,指明要描述的数据块的主要属性,后面两个 20 位长的字段则分别指明数据块的长度和数据所在的首地址。应注意的是,描述符是对它所指向的数据块的共性描述,为数据块中所有数据字所共享,但对数据块中的每个数据字,则仍有 3 位附加标志符以表明它是属于哪一类型的数据字。

8 位标志位的含义:分别表示是否描述另一个数据描述符,描述的是单个数据还是数据块,是连续存放还是分段存放,是字还是串,是单精度数据还是双精度数据,是可读可写还是只读。

借助描述符访问指令所需操作数的过程,如图 2 .9 所示。按指令操作数地址 X,Y 访存,若取来的字,其前三位为“000”,就是所需的操作数,若为“101”,表明它是描述符,将它取到描述符寄存器,由它的标志位、长度和地址字段联合控制,经地址形成逻辑、操作数的地址,再访存取操作数。当然,对于数据块,访存取到寄存器的描述符,能应用于块内的所有元素,不必每次访存取元素时都去加上访存取描述符的操作。这样,只需用这样一条指令就能执行对整个数据块的运算。

采用描述符方式操作处理数据块时可以有较简单的访存过程及越界校验等操作。这种表示方法为向量、多维数组等数据结构提供了较好的硬件支持,可以用一条指令控制整个阵列的运算,采用数据描述符方法形成数据地址的速度比常规计算机中采用的变址寻址要快得多,这样有利于简化编译中的代码生成,加快处理。

例如,通过多次使用描述符访问可方便地描述多维数据结构,如要构成一个 3×2 的二维数组,可用图 2 .10 所示的方法加以描述。

采用数据描述符表示法的优点、缺点与带标志符数据表示法相同,它的计算机结构可能比标志符数据表示法更复杂。

两种自定义数据表示方法的主要区别是:

① 标志符要与每个数据相连,两者合存在一个存储单元中,而描述符则和数据分开存放。

② 要访问数据集中的元素时,必须先访问描述符,这就至少要增加一级寻址。

③ 描述符可看成是程序的一部分,而不是数据的一部分,因为它是专门用来描述要访问的数据的特性的,例如,指明是访问整块还是单个数据,访问数据块或数据元素所需要的地址,数据块的长度等。

2.2.2 向量数据表示

如何为向量数据结构的实现和快速运算提供更好的硬件支持,是系统结构设计的另一个重要问题。

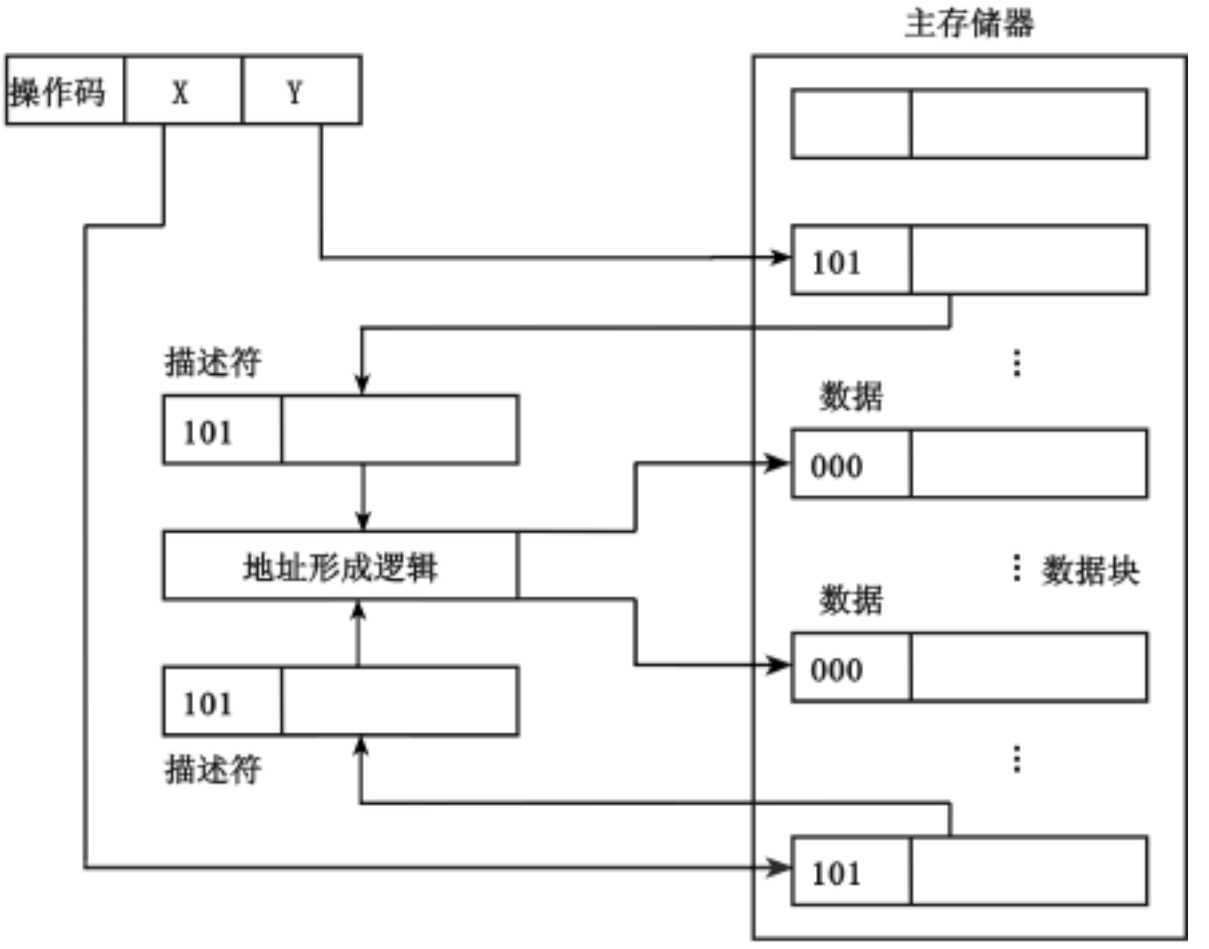


图 2.9 通过描述符取指令所需操作数的过程

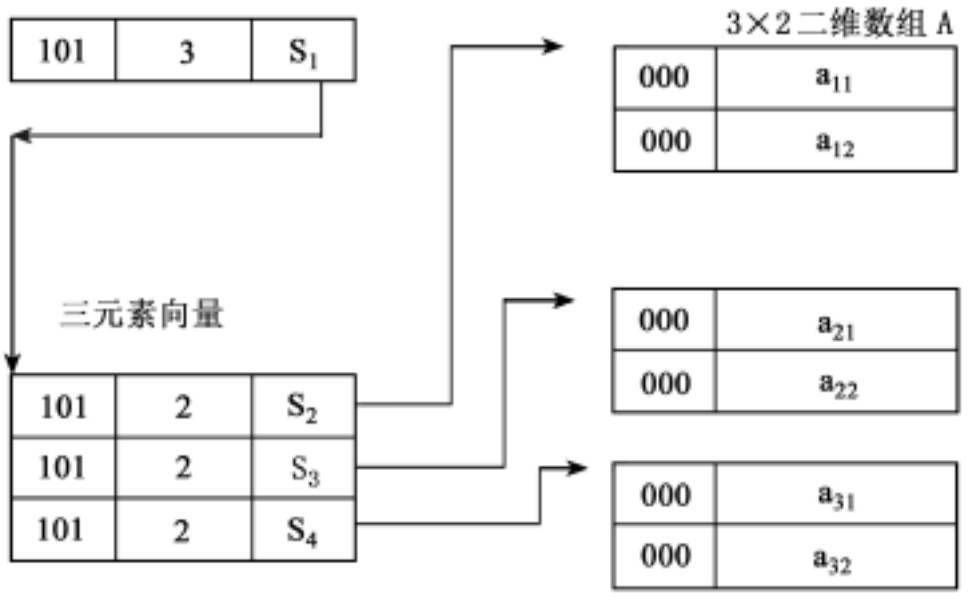


图 2.10 用描述符描述二维数组

为了较好地从硬件角度支持向量数据结构的实现和快速运算,在 20 世纪 60 年代中期就已在一些向量计算机系统结构中提供了向量数据表示,如 CDC STAR-100

计算机系统以及 TI-ASC 计算机系统。70 年代以后,出现了向量计算机,如 CRAY 公司推出的 CRAY-1 巨型计算机,更使向量数据表示得到了进一步的改进和完善。在这种计算机中,把向量数据类型直接作为一种有硬件支持的数据表示。

例如,若要计算 $c_i = a_i + b_i - 8$,$i = 4,5,\cdots,11$ 的向量加法时,用 FORTRAN 可写成如下的 DO 循环部分:

```
      DO   20   I = 4,11
20    C(I) = A(I) + B(I - 8)
```

像这种情况就很难采用数据描述符进行优化。

如果在标量机上运行时,因没有向量数据表示,故在编译后需借助变址操作实现,各条指令只能顺序执行。但若在有向量数据表示的计算机上运行,由于设置了相应的向量运算指令和快速的向量运算部件,就可大大加快这种向量运算操作。为了实现上述 DO 循环的全部功能,可设置如下的一条向量加法指令:

向量加	X	A	Y	B	Z	C

X,Y,Z 各区段表示寄存器号,分别存放源向量 A,B 和结果向量 C 的位移量,而 A,B,C 各区段分别存放源向量 A,B 和结果向量 C 的基址及长度。

图 2 .11 示出了根据上述向量加法指令中所给出的参数,对 A,B,C 三个向量进行编址的情况。图中各向量的脚标 b,d,s 和 e 分别表示相应向量的基址、位移量、起始地址和向量有效长度,其中操作向量的起始地址 = 基址 + 位移量,而向量有效长度 = 向量长度 - 位移量,若向量指令中 X,Y 和 Z 所指明的寄存器内容分别为 4, - 4 和 4,A,B 和 C 的值分别为 12,4 和 12,图 2 .11 中标出了其有关的参数值。此时仅需用一条向量指令:

$$C(4:11) = A(4:11) + B(-4:3)$$

就可代替前述的 DO 循环语句。

在向量处理中经常会遇到稀疏向量(即含有大量零元素的向量),为了节省存储空间和处理时间,通常采用压缩向量的表示方法。压缩一般要分为两步实现:第一步,先形成一个向量 Z,它是一个“位向量”,用来指明稀疏向量中各元素的状况及所在位置。第二步,根据 Z 向量的内容将稀疏向量与 Z 向量中“1”元素相对应的向量元素存入指定存储单元,转变成压缩向量,如图 2 .12 所示。这样不但可节省存储稀疏向量的空间,也可使实际的向量运算长度减少,从而可加快稀疏向量的运算。有序向量 Z 在稀疏向量的生命周期内应予以保存,这样在运算结束后仍可根据有序向量 Z 再恢复原来的稀疏向量。

2.2.3 堆栈数据表示

在计算机系统软件和程序设计技术中较为广泛应用的一种数据结构是堆栈。堆栈实际上是一种有序表,对堆栈数据处理是依据后进先出的原则,所以堆栈也称为后

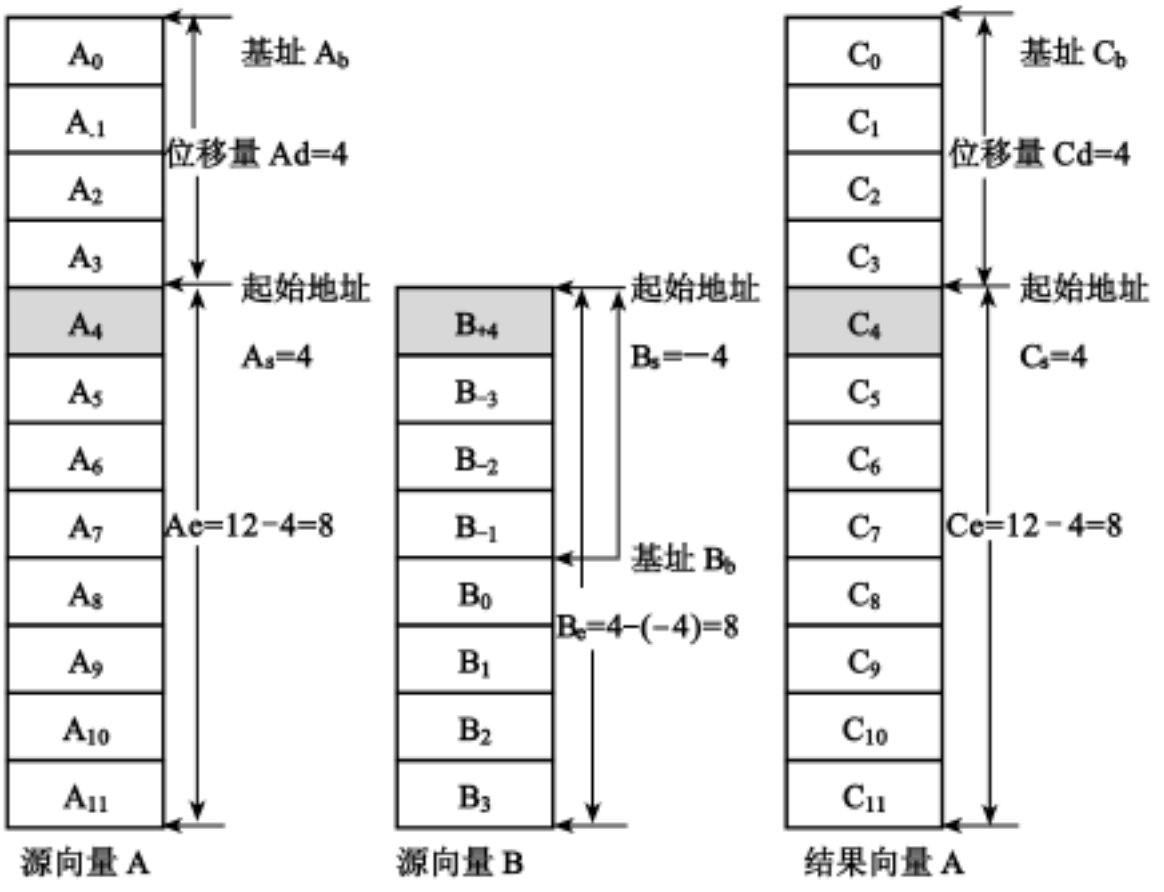


图 2 .11 向量数据表示所用的参数

进先出表。堆栈只是一个数据出入的端口,数据存入堆栈称为压入,数据从堆栈取出称为弹出。数据在堆栈中的存取过程,就像子弹在弹夹中压入与弹出的过程一样,先将堆栈内原来存放的数据依次向栈底移动一个单元,使栈顶空出来存放新的数据;当从堆栈弹出数据时,从栈顶单元取数据,并使下边所存数据依次向栈顶移动一个单元,使栈顶后的第二个单元数据填充到栈顶。可见,数据的存取只能在栈顶单元进行。

为了控制简单,便于逻辑实现,一般在压入、弹出时,栈内数据单元地址不变,而使栈顶单元改变。

堆栈数据操作的重要特点是指令只需指出进行什么样的操作,而无需指出操作数地址。但由于堆栈在主存中,取栈顶操作数和次栈顶操作数以及运算结果压入堆栈都需要访问主存,所以堆栈指令是很慢的。为了改善这种情况,在有的堆栈型计算机中,由高速寄存器组成硬件堆栈,并附加控制电路使之与主存中的堆栈联为一体,这样,指令中对栈顶和次栈顶数操作时可直接在寄存器中进行,不必访存操作,可大大加速堆栈指令的执行速度。

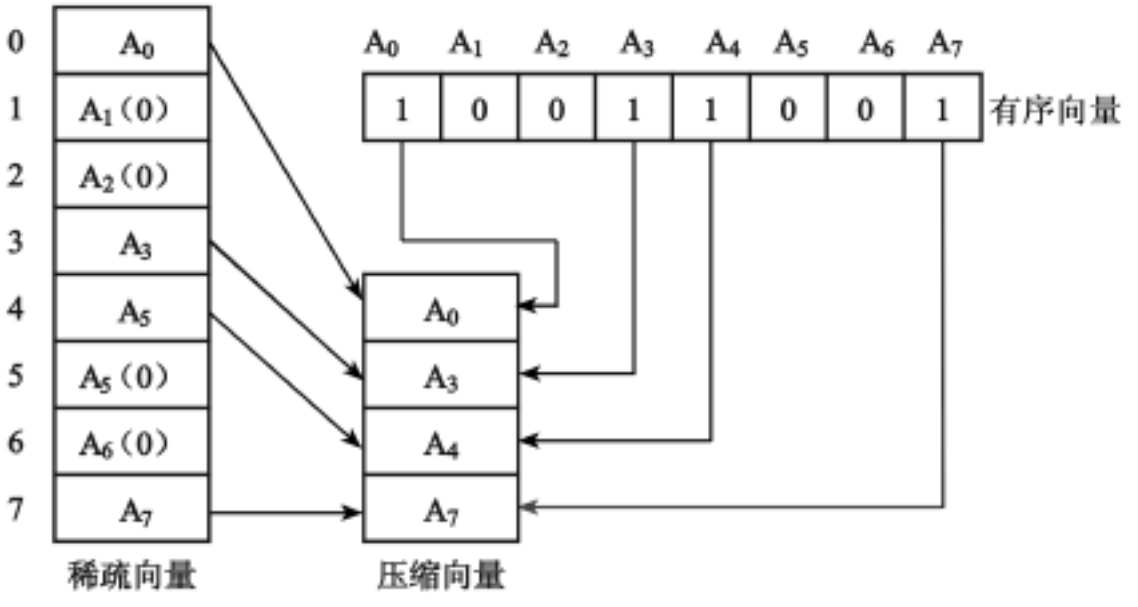


图 2.12　稀疏向量、有序向量和压缩向量

堆栈数据表示,对于复杂的算术表达式的求解是比较方便的。将算术表达式转用逆波兰表示法,在堆栈上进行计算很方便,可以缩短计算程序。

另外,堆栈数据表示支持对子程序的嵌套和递归调用,在堆栈计算机上还含有很丰富的堆栈操作类指令且功能很强,直接可对堆栈中的数据进行各种运算和处理。

堆栈结构及逆波兰表示法的特点被广泛应用于高级语言的编译技术和程序调用技术等方面,所以现代计算机系统结构中一般都设置有堆栈型数据表示。具有堆栈型数据表示且以面向堆栈寻址方式为主的计算机称为堆栈型计算机。

2.3　寻址方式与指令格式的优化设计

指令系统是软件与硬件的一个主要分界面,是计算机系统结构的一个重要组成部分。而寻址技术又是指令格式设计中要重点考虑的一个方面,本节就主要介绍常用的几种寻址方式及其优缺点、指令中操作码的几种编码方法,在此基础上给出如何进行指令格式的优化设计,并举例说明。

2.3.1　寻址方式

指令在执行过程中需要操作数,运算结果要送到存储单元中保持,寻找操作数及数据存放单元的方法称为寻址方式。

寻址方式在指令中的指明方式有两种:一是利用操作码的某些位来指明;二是在地址码部分专门设置寻址方式位来指明,如在 PDP-11 计算机中,用 3 位表示 8 种寻址方式,在 VAX-11 计算机中,用 4 位表示 16 种寻址方式。在汇编语言和计算机组成原理课程中我们已经学习了寻址方式的基本原理,本课程主要从计算机体系结构

的角度介绍寻址方式的设计思想和设计方法。

寻址方式的多样性和灵活性可使指令系统对各种应用需要(对各种数据结构的处理)提供好的支持。那么如何根据寻址方式找到指令在执行过程中所需要的操作数和存放运算结果呢？计算机中的操作数可能存放的位置有主存储器、寄存器、堆栈等,因此,计算机中的寻址方式就有面向主存储器寻址(直接、间接、基址和变址),面向寄存器寻址,面向堆栈寻址,下面介绍这几种寻址方式的设计思想和设计方法。

1 .立即寻址

操作数除了可存放在主存储器、堆栈和寄存器外,还有一种来源,就是直接在指令中给出,这种寻址方式称为立即数寻址方式,通常仅仅用来指定一些精度要求不高的整型常数。

立即寻址方式的优点是无须再去寻找操作数存储单元,指令执行速度快。但它只能用于源操作数寻址,数据长度不能太长(受指令字长度限制),大量使用会使程序的通用性降低。

也可以把这种方式归为主存储器寻址方式。

2 .寄存器寻址

以寄存器寻址方式为主的计算机称为通用寄存器型计算机。

指令在执行过程中所需要的操作数来源于寄存器,运算结果写回到寄存器中,这种寻址方式在所有的计算机中都普遍采用。这是因为目前的处理机(特别是 RISC 系统计算机)中通常都有几十个、几百个甚至几千个寄存器。

当然,对于输入输出指令和一些特殊的处理机控制指令,指令中所给出的寄存器可能是设备的控制寄存器、状态寄存器和处理机的程序计数器、堆栈指针、状态字寄存器等。

(1) 寄存器寻址可分为两种:

① 直接寻址:指令在执行过程中所需要的操作数直接取自通用寄存器,运算结果也保存在通用寄存器中,这样,运算型指令只要指定通用寄存器的编号,无须指定主存储器的地址,这就是寄存器直接寻址方式。

② 间接寻址:即在通用寄存器中存放的是操作数在主存储器中的地址,或者是操作数在主存储器中的地址的地址等(多重间址),这种寻址方式称为寄存器间接寻址。它能解决操作数地址的修改问题,可以做到在程序设计过程中对操作数所存放的主存地址进行修改,而不必去修改程序中的指令本身。

(2) 寄存器寻址方式的优点主要有:

① 指令字长短。由于通用寄存器的个数一般只有几十个,在指令中只需要很少几位就能表示一个操作数的地址。例如,在 IBM370 系列计算机中,有 16 个通用寄存器,只要用 4 个二进制位就能表示一个操作数的地址,即使是三地址指令,也只要

12 位地址码。

由于通用寄存器的字长可以比较长,如 32 位或 64 位,在采用寄存器间址方式时,在通用寄存器中可以存放字长很长的主存地址,对于数据为 64 位的计算机系统,在一个通用寄存器中不仅可以存放一个字长很长的主存地址,还可以同时存放地址的偏移量及各种标志等其他许多信息。

② 指令执行速度快。由于寄存器的速度很快,访问寄存器的时间和访问主存的时间相比几乎可以忽略不计,因此,大多数寄存器型指令都能在一个节拍内完成。对于那些要连续使用的数据,把它们存放在通用寄存器中,能够大幅度提高程序的执行速度。而且,对于普遍使用的二地址指令,必须要有通用寄存器的支持,否则执行速度将明显下降。

③ 支持向量、矩阵运算。当通用寄存器的数量比较多时,比如在 CRAY-1 向量计算机中,有 512 个寄存器,这样可以把一个向量或向量的一部分放在通用寄存器内,从而提高运算速度。

(3) 寄存器寻址方式也有明显的缺点,主要是:

① 不利于编译程序的设计。通用寄存器的分配是否合理,直接影响到编译程序的设计。这是因为通常要把那些连续使用或用得比较频繁的变量分配在通用寄存器内,这就给编译程序在寄存器的优化使用方面带来很大的麻烦,这说明,面向寄存器寻址方式为主的通用寄存器型计算机不适应高级语言的数据模型。

另外,通用寄存器使用中的一些特殊用法及某些限制也给优化编译造成很大的困难。如在 IBM370 系列机中规定:R0 寄存器不能作变址寄存器和基址寄存器使用,对于双字长指令只能用编号是偶数的通用寄存器等。

② 不利于中断和子程序的递归调用。在程序运行的过程中,当发生了中断或子程序调用时,需要保护和恢复现场状态。这时要把有关通用寄存器中的内容都保存到主存储器(简称“主存”)中,在中断返回和子程序返回时,又要全部恢复这些寄存器中的内容,寄存器的数目越大,保存和恢复所要的时间就越长。为了解决这个问题,在 RISC 计算机中现在普遍使用了重叠寄存器窗口技术,这在下一节将具体讲述。

③ 增加了硬件复杂度。在处理机中设置大量的通用寄存器,一方面使硬件成本增加,另一方面,由于这些寄存器在使用过程中的控制也相当复杂,这些都使硬件设计复杂化。如在超标量超流水线处理机中,同时要从寄存器中取多个源操作数,还要写回多个运算结果,可见其译码控制逻辑是相当复杂的。

3 .主存储器寻址

这是几乎所有计算机都有的一类寻址方式,也是最复杂的寻址方式。

指令中如果有存储器操作数,存储器操作数个数及寻址方式将直接影响到指令系统的设计和计算机系统的性能。下面我们就来简要地介绍一下存储器操作数的各种寻址方式及其优缺点。

主存储器寻址方式主要包括直接寻址、间接寻址和变址寻址三种类型。

(1)直接寻址

在指令中直接给出操作数的有效地址。这种寻址方式在早期的计算机和目前一些专用计算机中用得比较多,但随着主存容量的不断扩大及虚拟存储器的普及,这种寻址方式逐渐暴露出它的弱点,主要为:

① 地址码字段比较长。特别是在二地址和三地址指令中,字长有限的指令是无法容纳比较长的地址码的。

② 不支持程序的可再入性。如为了支持数组运算编制一段循环程序,那么在每次执行循环体时数组元素的地址是必须修改的,而采用这种寻址方式,修改数据地址就必须修改指令本身,采用这种方法编写的程序无法支持程序的再入性要求,这是现代程序设计思想所不能接受的。

③ 不适用于多任务和多用户环境。在多任务及多用户操作系统下,要求程序能够在主存中浮动,采用直接寻址方式编写的程序会给多任务多用户的操作系统作业调试带来很大的麻烦,而现在的操作系统几乎都是多任务和多用户操作系统。

鉴于以上的缺点,必须要有其他寻址方式来支持。

(2)间接寻址

和寄存器间接寻址的原理相同,只不过在采用主存储器间接寻址方式的指令中,给出的地址是主存地址而不是寄存器的编号。

间接寻址也可进一步分为一次间接寻址和多重间接寻址方式。一次间接寻址,在指令中给出的是操作数地址的地址,而在多重间接寻址方式中,指令中给出的是下一级间接寻址的地址。通常第一级间址的标志由指令给出,以后各次的间址标志要由紧接着访问主存储器所取出来的地址码给出,如果取出的地址码的间址标志位(通常是指定最高位)为“1”,则要用除去标志位后的部分作为地址码继续访问主存储器,直至取出来的地址码的间址标志位为“0”时为止,这时,除去间址标志位之后的地址即为有效地址。

通常划出主存储器最低端的一小部分存放间接地址,因此,采用间接寻址方式时,指令中需要表示的地址码的长度可以很短。

采用间接寻址方式,由于必须经过两次或两次以上的访问主存储器操作才能得到操作数,所以采用这种方式的指令执行速度不高。

(3)变址寻址

采用这种寻址方式时,需要设置一个或多个变址寄存器。变址寄存器的长度由主存储器的寻址空间决定。也可以直接采用通用寄存器作为变址寄存器来使用。变址寄存器的主要作用是用来存放数组的基地址。

指令中如采用这种寻址方式,只需要给出变址寄存器的编号和地址的偏移量,在指令的执行过程中,用一个硬件加法器,把变址寄存器中给出的基地址与指令中给出的偏移量做算术加法,相加的结果就是有效地址。如果变址寄存器只有一个,可以隐

含不表示出来。

另外,变址寻址还有两种特殊的形式。一种称为 PC 相对寻址方式,另一种称为基址寻址方式。

① 相对寻址方式:当变址寄存器就是程序计数器本身时称为 PC 相对寻址方式,这种寻址方式主要在控制转移指令中用来指定代码位置。用这种寻址方式实现分支转移或者无条件转移的优点很明显,这是因为目标通常都和当前的指令离得不远,而且使用相对偏移地址可以减少指令的长度。使用 PC 相对寻址还可以使程序的运行与加载的位置无关,可以减少程序在链接时的工作,对在执行时才链接的程序也有好处。

② 基址寻址方式:这是为了支持程序的动态再定位而引入的一种寻址方式。实际上,它也是一种变址寻址方式。在有些计算机系统中,既有变址寻址,又有基址寻址,这时,在指令执行过程中,对于每一个操作数都要进行两次变址运算,即做两次加法运算才能得到操作数的地址。

(4)间接寻址与变址寻址

间接寻址与变址寻址的目标都是为了解决操作数地址的修改问题,它们都能做到在程序设计过程中能够对操作数的地址进行修改,而不必去修改指令本身,既然它们的作用和目标一样,那么,就会提出这样一个问题:在设计一台计算机系统时,间接寻址与变址寻址方式是选择一种,还是二者兼用?又如何选取间接寻址与变址寻址方式呢?

对于这个问题,就需要对间接寻址和变址寻址方式作一个比较,比较结果如表 2.6 所示。

表 2.6　　间接寻址与变址寻址的主要差别

比较内容	间接寻址	变址寻址
有无偏移量	间接地址在主存中,无偏移量	基地址在变址寄存器中,有偏移量
实现的难易程度	实现容易,只需增加一条从主存的数据寄存器到地址寄存器的数据通路即可	需要相应的硬件支持,如需一个或多个变址寄存器,一个硬件加法器
指令的执行速度	执行速度慢,因读写一个操作数需要两次或两次以上的访存操作	执行速度较快。通过硬件加法器得到操作数有效地址,只需访问主存 1 次即可获得操作数
对数组运算的支持	对数组运算的支持较差	对数组运算的支持较好,因基地址加偏移量能够很有效地表示向量、矩阵等运算

在变址寻址方式中,偏移量是带有符号的,通常用补码表示,这样,不仅可以向前

转移,也可向后转移。偏移量的长度一般短于基地址的长度,在做加法时要进行符号扩展。

如果在一台计算机中同时设置了间址和变址这两种寻址方式,先做变址运算还是先做间址运算也需要事先定义。在指令中如给出一个变址寄存器 X 和一个相对地址 A,有效地址 EA 的计算方法有如下两种:

第一种:前变址寻址方式。EA = ((X) + A)

第二种:后变址寻址方式。EA = (X) + (A)

另外,无论采用间接寻址,还是变址寻址方式编写程序,对于数组运算,都必须有对地址进行增量的指令,为省去这些对地址进行增量的指令,在许多计算机中对于这两种寻址方式都增加了自动变址的功能。地址增量的先后关系是必须清楚的,一般有三种方式:

第一种:先用后增与先减后用方式,即(X) + , - (X)。这种方式多用于有后进先出堆栈,且堆栈指针指向栈顶元素的计算机中。第二种:先增后用与先用后减方式,即 + (X),(X) - 。这种方式多用于有后进先出堆栈,而且堆栈指针指向栈顶第一个空元素的计算机中。第三种:先增后用与先减后用方式,即 + (X), - (X)。这种方式多用于没有后进先出堆栈的计算机中。

4.堆栈寻址

操作数放在堆栈中的寻址方式称为堆栈寻址,这种方式只能对栈顶元素进行操作。

从 20 世纪 60 年代开始,出现了一批以堆栈寻址方式为主的堆栈计算机。这类计算机系统与以寄存器寻址方式和主存寻址方式为主的计算机系统相比,具有如下特点:

① 程序所占的主存空间最小。由于堆栈指令不需要地址码,指令的长度很短,与以寄存器和以主存寻址方式为主的计算机系统相比,程序的总存储量要缩短很多。

② 有力支持程序的嵌套和递归调用,支持中断处理。在程序的嵌套和递归调用过程中,要保存返回地址,保存处理机状态,保存程序现场,并向子程序传送参数,在堆栈型计算机中,可以把这些信息都压入堆栈,而不必为它们赋予地址。当从子程序返回时,可以直接从堆栈中弹出所需要的信息,这些都使辅助操作减少,加快运算速度。中断的处理过程与程序调用很相似,使用堆栈能加快中断的处理过程,简化中断程序的设计。

③ 有力地支持高级语言程序的编译。因为一般的算术表达式可以很容易地转化成逆波兰表达式,而逆波兰表达式能够直接形成由堆栈指令组成的程序,这样就缩小了高级语言和机器语言之间的差距,简化了编译程序的设计。而在以主存寻址为主的计算机中,在编译一个算术表达式时,要为每一个变量分配主存单元,如何合理为变量分配存储单元,减少中间变量的个数,是编译器的一项相当困难的工作。对于

以寄存器寻址方式为主的计算机系统,编译器必须优化寄存器的使用,减少访问主存的次数,同样,也存在如何减少中间变量、节省存储空间的问题。

堆栈型计算机的主要缺点是运算速度比较低,这是由于堆栈与处理机之间的信息传送量很大造成的。另外,也不能随机访问堆栈,从而很难生成有效代码,堆栈是整个系统的瓶颈,所以很难被高效地实现。

以上比较了面向寄存器、面向主存储器和面向堆栈这三种主要的寻址方式,这三种寻址方式各有优缺点,很难笼统地说哪一种寻址方式最好,对不同的用户程序和不同的工作阶段会得到不同的效果。而且由于用户程序的多样性及高级语言源程序从编译到运行的阶段不同反映出的工作特点也不同,所以这三种寻址方式不应相互排斥。在同一系统中,这三类面向的寻址方式都应采用,只是根据应用的特点,选择某一种面向的寻址方式为主,其他面向的寻址方式为辅,以相互取长补短。例如,在通用寄存器型机器系统结构中可增加堆栈及堆栈寻址方式,以支持高级语言程序中表达式的编译和子程序的调用。在堆栈型计算机中,可以增加面向通用寄存器的寻址方式以及除可以直接访问栈顶外,也能实现直接访问栈内任意单元的能力。为提高堆栈的运算速度,可以设置硬堆栈或者增设栈顶寄存器组。

2.3.2 程序定位技术

上面所讲的各种操作数的寻址方式,都是按地址访问的方式。如果按地址访问,必须对这些可访问的存放操作数的部件进行编址,地址的编址,通常有三种方式:第一种,按各种部件分类编址。各类部件分别从“0”开始单独编址,形成多个一维的线性地址空间。第二种,隐式编址,如堆栈和某些专用寄存器,多数采用事先约定好的编址方式隐式寻址,以加快对这些部件的访问。第三种,统一编址,即把各部件编址成一个从“0”开始的一维线性地址空间,对不同部件的访问反映在对这个空间中不同地址的访问上。但无论是哪种编址方式,在计算机中有两个地址概念应该加以区别,一个是逻辑地址,另一个是物理地址。

逻辑地址指的是程序员在编制程序时所使用的地址,包括高级语言的符号名称地址和通过编译生成的逻辑地址,它所对应的空间称为逻辑地址空间。前面所讨论的寻址方式,主要是指逻辑地址的寻址方式。一般各个程序的逻辑地址空间是独立的,都是从“0”开始的一维线性地址空间,它们独立于实际存储空间。在冯·诺依曼计算机中主存是由一个从“0”开始编址的一维线性地址来访问的,这一地址称为实际的主存物理地址,对应的空间称为主存物理地址空间。

为了把一个程序交给处理机运行,必须先把这个程序的指令和数据装入到主存储器的一个或几个区域中,一般情况下,程序所分配到的主存物理空间与程序本身的逻辑地址空间是不相同的,因此,当程序装入主存时需要进行逻辑地址空间到物理地址空间的变换,即进行程序的定位。定位技术就是主要研究程序中的指令和数据的主存物理地址在什么时间确定?采用什么方式来实现?

根据程序中指令和数据的主存物理地址的确定时间,定位方式可分为直接定位、静态再定位和动态再定位三种。

1 .直接定位方式

一般适用于单任务工作方式。

程序装入主存之前,程序中的指令和数据的主存物理地址就已经确定了的方式称为直接定位方式。

在早期的计算机系统、目前的某些小型微型计算机和一些专用计算机中,在编译程序时,就已经可以确定程序在主存储器中将要存放的实际位置,因此,能够把指令和数据的地址直接编译成主存储器的物理地址。在这种情况下,逻辑地址空间和主存物理地址空间是一致的。可以采用这种方式。但目前在多道程序、多用户等系统中,它们是不同的,在装入程序时就必须使用其他定位方式。

2 .静态再定位方式

采用这种方式的程序必须是可重定位的,即要修改地址的指令和数据需具有某种标识。静态定位要求程序在运行之前,由专门设计的定位装入程序集中一次完成逻辑地址到物理地址的变换,程序一旦进入主存储器之后,地址就不能再修改。如早先在 DOS 操作系统中的 LINK 过程,在某些单片机、专用机中采用这种定位方式。

静态定位技术是由专门设计的静态定位装入软件来完成地址变换的,不需要增加任何硬件设备,而且可对多个程序段组成的程序进行静态链接,但是在执行期间,程序的地址是不能修改的,这对提高主存的利用率不利。倘若程序所需空间超过了分配给它的主存物理空间,还必须采用覆盖技术,而且,多个用户也不能共享放在主存储器中的同一个程序,如果几个用户要使用同一个程序,则在每个用户程序中都必须包含这个程序的副本。采用动态定位技术可以解决这些问题。

3 .动态再定位方式

对于采用这种方式定位的程序,在装入主存储器时,指令和数据地址不做任何修改,只把此程序在主存中的起始地址存入与之对应的基址寄存器中。在程序执行过程中,再用地址加法器将指令中的逻辑地址和基址寄存器中的程序起始地址相加,就形成实际的主存物理地址。

采用基址寻址实质上是把原来由装入程序完成的功能,改用地址加法器和基址寄存器完成,使地址变换的速度加快了。动态再定位和静态再定位相比,具有如下优点:

① 提高了主存的利用率。由于地址变换是在程序执行的过程中完成的,程序运行之前也不必全部装入主存,这样一个程序可分配在多个不连续的主存空间内,只要为每个连续空间设定一对上、下界寄存器即可,而且这种界限寄存器的设置,还有利

于实现主存信息的保护。

② 主存中的同一个程序段可为多个程序所共享。

③ 支持虚拟存储器。在虚拟存储器中,进一步发展和改进了动态定位技术,地址的转换和程序的调度完全由系统管理程序来完成,这在第四章中会进一步讲述。

但是为实现动态再定位,需要有相应的硬件支持,而且在虚拟存储器中,实现存储管理的软件算法也比较复杂。

2.3.3 指令格式的优化与设计

在“计算机组成原理”课中已学过,指令由操作码和操作数寻址信息两部分组成。操作码指明操作种类和所用操作数的数据类型,地址码包括操作数的地址、地址的附加信息、寻址方式等。如何用较少的位数来表示指令的操作信息和地址信息,将影响到编程后的目标程序长短及执行速度,所以指令格式的优化设计是指令系统设计的一个重要问题。指令格式优化设计的主要目标有两个,一是节省程序的存储空间,二是指令格式要尽量规整,以减少硬件译码的复杂程序。下面,分别介绍指令中操作码和地址码的设计方法,再以 IBM370 指令格式为例说明整个指令格式的优化设计方法。

1.操作码优化设计

操作码优化设计的目的主要是为了缩短指令字的长度,增加指令字所能表示的操作信息和地址信息。要对操作码进行优化设计,就需知道每种操作指令在程序中出现的概率或使用频度(这一般可通过对大量已有的典型程序进行统计求得)。按信息论的观点,当各种指令的出现是相互独立的时候,操作码的信息源熵(信息源所包含的平均信息量) H 为(以二进制位数表示):

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i$$

其中, P_i 表示第 i 类指令的使用频度,共有 n 类指令。这里的 H 即为操作码可以达到的最短平均码长。

而实际编码的操作码平均码长为: $l = \sum_{i=1}^{n} l_i p_i$

式中, l_i 为第 i 类指令的编码长度, p_i 为第 i 类指令的使用频度。这种编码的信息冗余量为:

$$R = 1 - \frac{H}{l}$$

操作码的优化表示就是要使信息冗余量 R 最小。

操作码的表示方法通常有三种,等长编码、Huffman 编码法和扩展编码,下面分别加以介绍。

1.等长编码

在系统中所有指令的操作码长度都是相同的,即所占用的二进制位数相同。

一般处理机的指令条数通常为几十条到几百条之间,用一个字节(8 位)表示,指令规整,硬件译码简单,但操作码的长度增加了,信息冗余现象严重。

现设一台模型机,共有 7 条不同的指令,使用频度如表 2 .7 所示。其最短二进制平均码长为:

$$H = -\sum p_i \log_2 p_i = 0.40 \times 1.32 + 0.26 \times 1.94 + 0.15 \times 2.74 + 0.06 \times 4.06 + 0.05 \times 4.32 + 2 \times 0.04 \times 4.64 = 2.27$$

即要表示这 7 条指令,只需 2 .27 位就够了。

若用等长操作码编码方法,则至少需要 3 位二进制数来表示($\lceil \log_2 7 \rceil = 3$),平均码长 $l = 3$,信息冗余量为:

$R = 1 - 2.27/3 = 0.24$,即 24%。

表 2 .7 模型机的操作码的使用频度和三种不同编码方法

指令	使用频度(p_i)	等长编码	Huffman 编码	扩展编码
I1	0 .40	000	0	00
I2	0 .26	001	10	01
I3	0 .15	010	110	10
I4	0 .06	011	11100	1100
I5	0 .05	100	11101	1101
I6	0 .04	101	11110	1110
I7	0 .04	111	11111	1111
平均码长	2 .27	3	2 .32	2 .38
信息冗余量		24%	2%	5%
码长种类		1	4	2

2 .Huffman 编码法

利用哈夫曼压缩概念的思想,根据每类指令的使用频度,使使用频度高的指令的操作码用较短的二进制位来表示,频度较低的指令的操作码用较长的二进制位表示,使得平均二进制位数变短(与等长编码相比)。哈夫曼编码的优点是实际平均码长最短;但各类码长种类太多,实际不能采用。

求哈夫曼编码的方法如下:

① 构造哈夫曼树。将所有的使用频度值作为树的叶节点,找出两个权值最小的相加,相加后的值作为新节点的权值,放入其中再作比较,继续用两个权值最小的节点相加,形成一个新节点,重复以上过程,直至根节点,根节点的权值必定为 1 .00。

② 编码。按照二叉树左“ 1 ”右“ 0 ”或者是左“ 0 ”右“ 1 ”的原则,在树上标出,遍历所有的叶节点。然后,在根节点至叶子节点的路径上的二进制符号,即为叶子节点对应的该类指令操作码的哈夫曼编码。

如上面所讲的模型机,哈夫曼树及编码如图 2 .13 所示。由图 2 .13 可得出哈夫曼编码的平均码长为:

l= 0 .40×1 + 0 .26×2 + 0 .15×3 + 0 .06×5 + 0 .05×5 + 0 .04×5 + 0 .04×5 = 2 .32

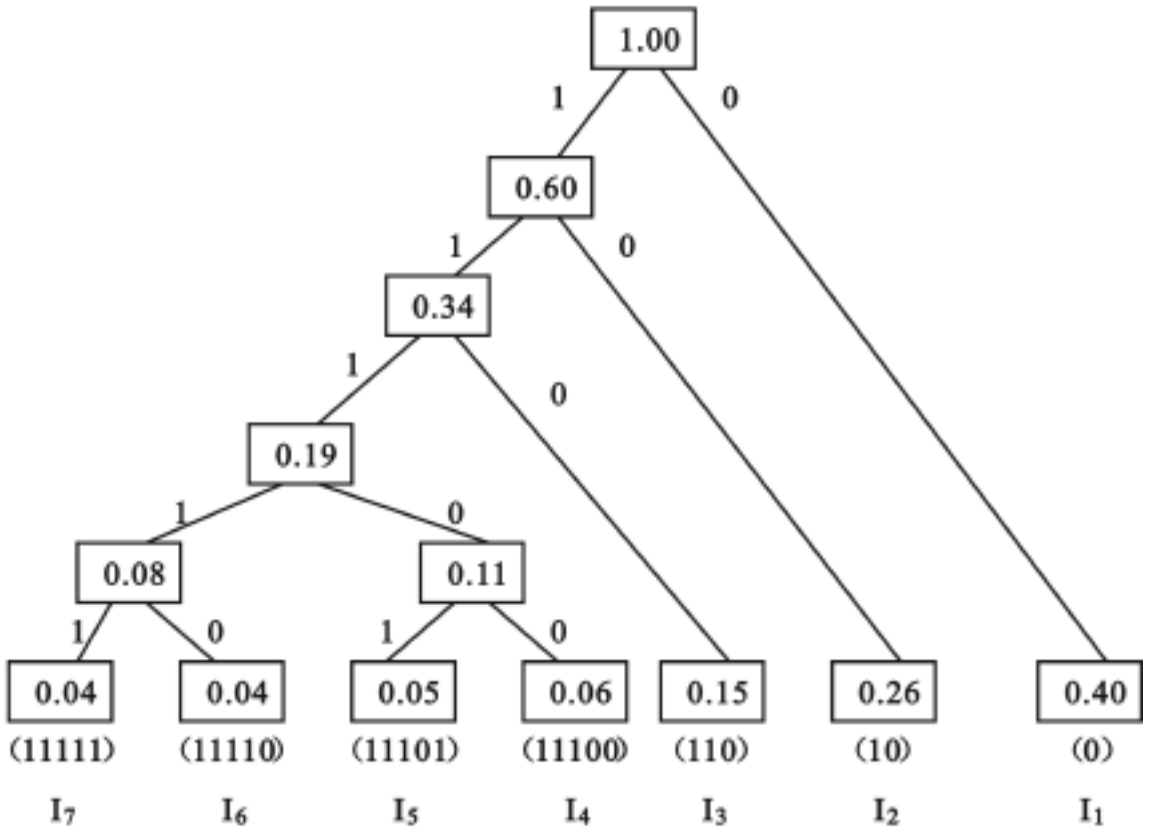


图 2 .13　利用 Huffman 树进行操作码编码

信息冗余量为:R = 1 - 2 .27/ 2 .32 = 0 .02(即 2%)。

从表 2 .7 中可看出,哈夫曼编码的平均码长最短,只比最优编码多 0 .05 位。

实际上,操作码不可能达到最优哈夫曼编码法的最短平均长度,这是因为操作码的位数必须是正整数,然而,可以利用哈夫曼思想,通过构造哈夫曼树对操作码进行编码,达到实际平均码长最短的哈夫曼编码,但这种编码方法不是惟一的,只要将任意一个二叉节点上的“0”和“1”交换,就可以得到一组新的操作码编码,无论怎样交换,操作码的平均长度是惟一的。

3 .扩展编码法

采用哈夫曼编码能够使操作码的平均长度最短,信息的冗余量最小,但这种方法形成的操作码很不规整,即全哈夫曼码是完全不等长码,如表 2 .7 中所示,7 条指令就有 4 种不同的码长,这样,既不利于硬件的译码,也不利于软件编译,还很难与地址码配合,形成有规则的指令字长。

鉴于以上两种编码方法各自的特点,在许多处理机中,采用了一种新的折中的编码方法,即哈夫曼扩展编码法,简称扩展编码。这种方法是由等长操作码与哈夫曼操作码编码方法相结合形成的。

扩展编码法的基本思想就是对哈夫曼编码,根据使用频度宏观分布,将编码长度扩展成几种长度的编码。使这种编码的平均码长接近全哈夫曼码的码长,但又保持了定长码指令的规整性。

常用的扩展方法有等长扩展和不等长扩展两种。扩展编码中选择某些特征位用于扩展。但无论采用哪一种方法,衡量的标准是操作码平均码长是否最短,即信息冗余量是否最小。

如对模型机的 7 条指令采用 2-4 等长扩展编码,使码长由全哈夫曼编的 4 种减为 2 种,最高两位二进制‘11”作为扩展标志,区分 2 位码长和 4 位码长。

为了便于实现分级译码,一般采用等长扩展法。例如,操作码按 4-8-12 位进行扩展,当然要根据实际情况,也可以采用每次扩展的位数不等的不等长扩展。如美国的 Burroughs 公司生产的 B-1700 计算机中就采用 4-6-10 扩展操作码。

对于等长和不等长扩展法,根据采用不同的扩展标志还可以有多种不同的扩展方法。如对于等长扩展 4-8-12 扩展法,有采用保留一个码点标志的 15/ 15/ 15 扩展法,采用每次保留一个标志位的 8/ 64/ 512 扩展法等。对于不等长扩展 4-6-10,可以有 15/ 3/ 16,8/ 31/ 16,4/ 32/ 256 等多种编码方法。但具体采用哪种扩展,还要根据所设计系统中各种指令出现的概率分布情况决定。

例如,假设指令系统共有 42 种指令,前 15 种使用频度平均为 0 .05,中间 13 种使用频度平均为 0 .015,最后 14 种使用频度平均为 0 .004。如何编码?

因 42 种指令的频度分布有三种,故码长可有三种;又因每段指令数基本相同,故可采用等长扩展(4-8-12 位),结果如图 2 .14 所示。

从图 2 .14 中可以看出,采用的是 15-15-15 扩展方法,最后一种编码用于扩展,每段 0000-1110 用于编码,1111 用于扩展。

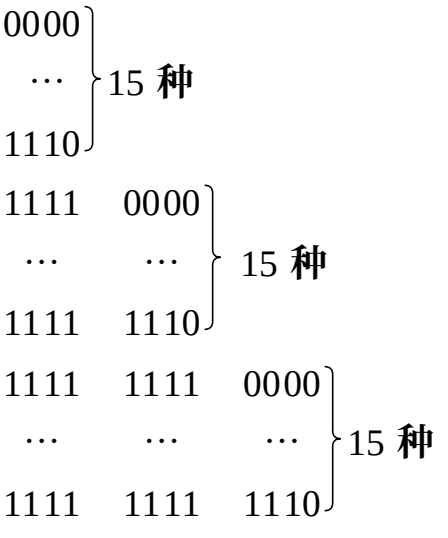


图 2 .14 一种 15-15-15 的扩展编码方式

2. 地址码的优化表示

有了操作码的优化设计,如果没有在地址码表示和寻址方式上采取相应的措施,程序所需总位数还是难以减少的,操作码的优化将没有任何意义。这是因为,现在通用的主存结构一般为定长字,且绝大多数计算机的最小可寻址单位是字节地址,计算机可定位的地址很少是二进制位。那么,为了保证只需一次访存便可取出一条指令,要求指令字遵从整数边界存储原则,实际上,为了保证所需的各种信息都只用一个存储周期访问到,信息在主存中的存放都要遵守这个原则,这就要求信息在主存中存放的地址必须是该信息宽度(字节数)的整数倍,否则可能发生信息跨主存边界存放,影响速度。即:

字节信息地址为:X…XXXX

半字信息地址为:X…XXX0

单字信息地址为:X…XX00

双字信息地址为:X…X000

这就是信息在存储器中按整数边界存储的概念。像 CDC STAR-100 那样的计算机是按位编址的,信息在存储器中的存放是用该信息的首位地址表示的,为保持信息按整数边界存储,就要求存储在主存中的各种信息的地址必须是其信息位数的整数倍。

指令字的存放也需遵守这个存储规则,在此前提下,如何使操作码优化后的表示能够发挥其优点呢?这与指令字是定长或变长有关。若指令字为定长,则在操作码优化表示后,将使指令字有效部分变长,操作码部分必须取操作码优化后的长度中最长者作为操作码部分标准。因此若地址码部分不随之作相应的长度变化,则操作码优化的效果就显示不出来。所以地址码必须与操作码相配合,完成整个指令格式字的优化设计,一般采取的办法有以下几种:

① 采用多种地址码长:长操作码与短地址码相结合,短操作码与长地址码相结合。

实际上,由于寻址方式的种类很多,每种寻址方式所占用的操作数地址码位数变化很大,如寄存器寻址和主存直接寻址,所占位数相差很大。这说明地址码长度可在很宽的范围内变化,只要恰当安排,就可与变长的操作码相配合构成定长指令字。

② 可以采用多种地址制,使多种地址制都可以在指令中使用,同一种地址制还可以采用多种地址形式和长度,如果操作码和地址码之外还有空余的码位,则设法用来存放立即操作数或常数。

如果让最常用的操作码最短,其地址码字段个数越多,采用二地址、三地址甚至多地址制,就越能使指令的功能增强,越可以从宏观上减少所需的指令条数,进一步缩短程序占用的空间,也会因减少访存次数而加快程序的执行速度。而对操作码长的指令,可以采用单地址或零地址制等。

③ 进一步,还可以考虑采用多指令字长度的指令,这比只有一种长度的定长指令字方式更能减少信息的冗余量,缩短程序的长度。

在 PDP-11 和 VAX-11 系列机中,都采用扩展操作码方式。在单字长及双字长的指令格式中,操作码的长度有 4,7,8,10,12 位和 13 位 6 种,对于不同长度的操作码,配以不同的操作码地址个数,就能充分利用非操作码部分的空间。而且,在当今的 RISC 机的指令系统中,都是定字长指令格式,为了充分利用指令字空间,通常采用扩展码形式,实际上这也是一种变相的可变操作码长度格式。

最后,以 IBM370 为例,介绍指令格式的设计。

IBM 370 系列计算机的指令长度有 16 位、32 位和 48 位 3 种,所有指令的操作码都是 8 位固定长度,操作码的最高两位用来指明指令的长度和格式。具体如下:

00 为 RR 格式,指令字长 16 位

01 为 RX 格式,指令字长 32 位

10 为 RS 或 SI 格式,指令字长 32 位

11 为 SS 格式,指令字长 48 位

这里 R 代表寄存器,S 代表存储器,X 代表变址,I 代表立即操作数。操作码的其余 6 位用来区分每种格式下的各种指令,因此每种格式最多可表示 64 种指令。这 5 种主要指令的格式如图 2 .15 所示。

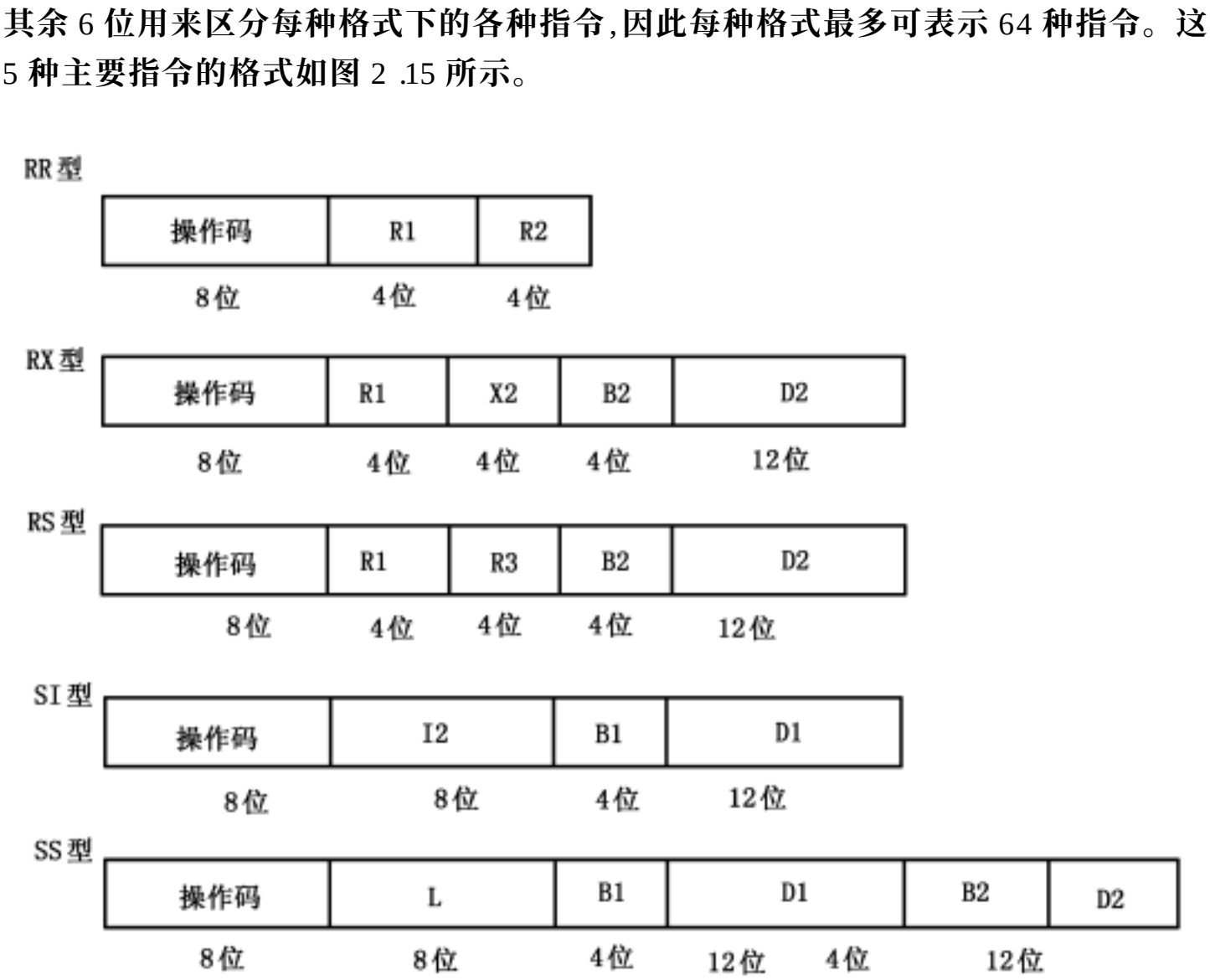


图 2 .15　IBM 370 指令的主要格式

在 IBM370 中有 16 个通用寄存器,可兼做变址寄存器和基址寄存器使用。5 种指令格式都采用两地址制,其中,RR 型指令字长只需要 16 位,两个参加运算的操作数都在通用寄存器中,运算结果也放在通用寄存器中,一个存储字可以放两条指令。RX 型指令参加运算的操作数一个来自主存储器,并采用变址寻址方式,一个来自通用寄存器,运算结果也放在通用寄存器中,指令字长 32 位,正好一个存储字。RS 型指令字长也是 32 位,它用来在通用寄存器与主存储器之间一次传送多个数据,R1 为起始通用寄存器编号,R3 为结束通用寄存器编号,只要用一条指令就能把这连续编号的通用寄存器中的所有数据存入相邻的主存单元。同样,也可以从主存储器的相邻单元读出多个数据到通用寄存器中。这种指令在程序调用和中断处理时,可以加快现场的保护和恢复。SI 型指令同样也是 32 位长的指令字,它的一个操作数来自主存储器,一个为立即数,立即数的长度为 8 位,而 SS 型指令的两个操作数都在主存储器中,并且支持“串”操作,如字符串运算,十进制运算等,可支持用来把长度 L≤256 个字节的数据块从主存储器中的一个区域传送到另一个区域中去。

2.4 指令系统设计的两种风格

指令系统是指计算机所具有的全部指令的集合,它反映了计算机所拥有的基本功能,是机器语言程序员所看到的计算机的主要属性之一。指令系统的设计主要是功能设计和指令格式的设计,上节已经详细介绍了指令格式的优化设计方法,本节主要介绍两种不同风格的指令系统:CISC 和 RISC 各自的特点、实现的关键技术等。

2.4.1 指令系统的功能设计

指令系统是机器软硬件的主要交界面,对软件和硬件设计的影响都非常大,那么,一种指令集结构中的指令到底要支持哪些类型的操作呢?这就是指令集功能设计问题,下面就从不同的方面来简要介绍指令的功能设计。

1. 指令系统的组成

各种类型的计算机由于结构不同、具体硬件构成不同、对软件的支持要求不同,所以对应的指令系统也不同。除了系列机以外,各种计算机的指令系统是不兼容的。但一般指令系统应包括下面的一些基本功能指令:

① 算术和逻辑运算类指令。计算机现在在许多应用领域的主要任务仍然是做运算,因此,运算型指令在指令集中应该占有比较大的比重,否则影响系统的性能。

② 数据传送类指令。主要是在相同或不相同的数据存储设备之间传送数据。如在 CPU 和寄存器、存储器之间的数据传送。

③ 程序控制指令。包括各种类型的转移指令、程序调用和返回、循环控制指令等。

④ 输入输出指令。这种指令在多用户或多任务环境下属于特权指令。当程序需要进行输入输出操作时,用系统调用进入 OS,由 OS 对设备统一进行管理。

⑤ 处理机控制和调试指令。处理机控制指令主要是对系统状态进行切换,系统资源进行分配和管理等。调试指令主要用于硬件和软件的调试。硬件调试指令主要有:钥匙位置的读取、开关状态的读取、内部主要寄存器和主存单元的显示等。软件调试指令主要有断点的跟踪和自陷指令等。一般处理机都设置有调试指令,但这些指令对一般用户是不公开的。

以上这 5 类指令,每一个计算机系统基本上都有,且一般是必须具备的。

2. 指令系统的扩展

各种计算机除以上基本指令外,还有扩展的指令部分,这部分扩展功能指令主要是提供一些对高级语言和操作系统的支持,如访管和访问监督指令;存储管理和保护指令(实际上也是 OS 功能);控制系统状态指令;诊断指令(计算机里,如军用计算机,还提供对系统的诊断,可用软件实现,但硬件实现更容易,性能更好);高级运算指令等。详细扩展方向在 CISC 和 RISC 中再介绍。

3. 指令保护问题

为了防止用户程序破坏系统软件,在指令系统中,把指令分成特权指令和非特权指令。用户只能使用非特权指令。特权指令只能在操作系统的管态下运行,即只有系统管理程序可以使用,主要包括处理机状态的设置和管理、系统硬件和软件资源的管理、进程的管理等。所以只有在管态下才能够使用机器所提供的全部指令,包括特权指令。在用户态下,只能使用一般指令,不能使用特权指令。

4. 指令功能设计

指令功能设计的基本思想是:计算机系统中的一些基本操作应由硬件实现还是由软件实现;某些复杂操作是由一条专用的指令实现,还是由一串基本指令实现。希望能在充分发挥硬件功能的条件下,尽可能多地对高级语言和操作系统提供支持。从性能价格比来讲,如何达到最优呢?

从目前操作系统和高级语言来说,对一种计算机,指令系统的功能设计总体上要求具有完整性、规整性和可扩充性。

① 完整性(Completeness):要包括各种基本指令类型,能处理机器所具有的各种数据表示。

② 规整性(Consistency):各寄存器和内存单元在指令系统中处于对等地位,这对将来译码、执行比较有利,主要表现在“对称性”和“均匀性”两个方面。

对称性指各种与指令系统有关的数据存储设备的使用、操作码的设置等都要对称。如所有寄存器要同等对待。这一点在目前的许多计算机系统中都没有做到,往

往隐含规定某一个或某几个通用寄存器有特殊用途。

如有: (R1)op(R2)→R1　　或　　(R2)op(R1)→R1

也应有:(R2)op(R1)→R2　　(R1)op(R2)→R2

均匀性是指对于各种不同的数据类型、字长、数据存储设备、操作种类等,指令的设置要同等对待。如某计算机有 5 种数据表示、4 种字长、8 种数据存储的有效排列,则设计加法指令时,指令种类应该有 5×4×8 = 160 种两地址加法指令,如果再考虑对称性的要求和不同的寻址方式等,指令种类还要增加很多倍,这实际上很难做到,也是不现实的。因此,在设计指令功能时,对于规整性的要求必须有所选择,在 RISC 体系结构中,规定运算型指令都在通用寄存器内进行操作,即使在 CISC 中,如果采用二地址通用寄存器结构,通常也规定,两个地址中必须有一个是通用寄存器。对于逻辑数、十进制数、字符串数据表示、双字长、半字长和字节等数据长度,可以适当减少指令种类。这样,可以把加法指令的种类压缩在 10 种之内。另外,如果采用自定义数据表示方式,每个数据只要带有几个标志位,规整性问题就不难解决了。

③ 可扩充性(Extendibility):要保留一定余量的操作码空间,为以后的扩展所用。这一点主要是考虑兼容性的要求。没有兼容性,大量的系统软件和更多的各种应用软件将无法使用,计算机也就没有了市场,所以兼容性是必须考虑的。一般来讲,后生产的机器,总要对原有指令进行扩充,以提高计算机的性能。

为了使指令系统对软件层次有较好的支持,指令集在其基本功能之上都要进行扩展和改进。当前在指令功能的设计、发展和改进上有两种截然不同的方向,一种是强化指令功能的复杂指令系统计算机 CISC;另一种是降低指令集结构的复杂性,以达到简化实现,提高性能目的的精简指令系统计算机 RISC。下面将分别从这两个方向讨论指令系统功能设计的一些问题。

2.4.2　复杂指令系统计算机(CISC)设计风格

CISC(Complex Instruction Set Computer):它的改进方向是怎样进一步增强原有指令的功能,设置复杂的功能更强的新指令代替原先由软件子程序实现的功能,进行软件功能的硬化。按照这种方向来发展,必将使指令系统越来越庞大和复杂,因此,具有这样指令集结构的计算机被称为 CISC。

早期计算机的指令系统还是比较简单的,当时受制于计算机中的硬件比较昂贵和可靠性较低的约束。随着器件技术和微组装技术的不断发展,这些不再是计算机性能不高的主要问题;另一方面,大量的系统软件和应用软件对计算机兼容性的要求,都促使计算机科学家为提高计算机的性能,满足软件发展的需要,不断增强原有指令的功能和引入一些复杂的、功能更强的指令,这就开始出现了 CISC。可以说,大约从 1964 年的 IBM360 系列机开始,逐步确立这一设计风格,后来又有 DEC 公司推出的多功能小型机 PDP-11 系列、20 世纪 70 年代末期的 VAX-11 系列,以及 70 年代后出现的各种微机,如 Intel 80X86 系列、Motorola 68020 等,基本上都是遵循这种风

格的。通过对这些计算机进行分析研究,可以看到它们增强计算机指令功能的主要改进方向为:支持目标程序的优化实现、高级语言的优化实现及操作系统的优化实现,下面就从这三个方面介绍 CISC 的设计风格。

1 .面向目标程序的指令功能的优化与改进

目标程序是由处理机直接执行的机器代码,面向目标程序来改进的目标主要是缩短程序的长度,减少程序执行过程中处理机和主存之间信息交换的次数,减少指令的执行时间。改进的方法是通过对目标程序中指令的静态使用频度和动态使用频度的统计分析,确定改进措施。

静态使用频度指的是对程序中出现的各种指令或指令串进行统计得出的百分比。动态使用频度指的是在目标程序的执行过程中对出现的各种指令和指令串进行统计得出的百分比。按静态使用频度来改进指令系统是着眼于减少目标程序所占用的存储空间,按动态使用频度来改进指令系统是着眼于减少目标程序的执行时间。按这两种频度来改进都可以减少处理机和主存之间信息的交换量。改进的具体思路就是对于那些使用频度高的指令,可以增强其功能,加快其执行速度,缩短其指令字长;对频度高的常用指令串可以增设新指令来代替,这样,不但减少了目标程序所占用的空间,而且能减少目标程序访存取指令的次数,加快目标程序的执行。

IBM 公司曾对 IBM360 上运行的 19 个典型程序统计出几种常用指令的使用频度,如表 2 .8 所示。从这个表可以看出,大部分常用指令的静态使用频度和动态使用频度非常接近,这说明可只对其中一种频度进行统计分析,根据这种频度进行改进,同样可以实现对目标程序的优化与改进。根据表 2 .8 的统计,最常用的指令其实有三种:第一种是数据传送指令,如取指令的静态使用频度是 28 .6%,动态使用频度是 27 .3%;存指令的静态使用频度是 15 .0%,动态使用频度是 9 .8%。第二种是程序控制类指令,如条件转移指令,静态使用频度为 10 .0%,动态使用频度是 13 .7%。第三种是算术和逻辑运算类指令,如表中的加减指令,比较与逻辑左移指令的使用频度也很高,因此,优化目标程序的主要途径就有以下几个方面:

表 2 .8 IBM360 指令的动、静态使用频度

指令类型	静态使用频度%	动态使用频度%
L(取)	28 .6	27 .3
ST(存)	15 .0	9 .8
BC(条件转移)	10 .0	13 .7
LA(取地址)	7 .0	6 .1
SR(减)	5 .8	4 .5
A(加)		3 .7
C(比较)		6 .2

续表

指令类型	静态使用频度%	动态使用频度%
SLL(逻辑左移)	3.6	
BAL(转移与链接)	5.3	
IC(插入字符)	3.2	4.1

(1)增强数据传送指令的功能

由于这类指令在整个指令系统中占有非常重要的地位,设计好这类指令对提高计算机系统的性能至关重要。现在在一些数据处理和通用计算机中,为方便主存储器和通用寄存器组之间、通用寄存器和通用寄存器之间、主存储器和主存储器之间的信息块的传送,一般都设置了成组传送指令。

在 IBM370 机上增设了用单条指令完成多个数据传送的功能。如“成组取”和“成组存”指令,属于 RS 型指令格式:

成组取或成组存	R1	R3	B2	D2
8 位	4 位	4 位	4 位	12 位

“成组取”指令完成从(B2) + D2 地址指明的主存地址起始的一个字向量(32 位)读到号为 R1 到 R3 的一组顺序的通用寄存器中,而“成组存”指令刚好相反,它把通用寄存器组中的一组数据传送到指定的主存储器中。另外,为完成主存储器中不同区域之间的信息传送,IBM370 还增设了一条“成组传送”指令,属于 SS 型格式:

成组传送	L	B1	D1	B2	D2
8 位	8 位	4 位	12 位	4 位	12 位

它将从(B2) + D2 指明的主存起始地址、长度为 L 的字符行或字节向量传送到从(B1) + D1 指明的主存起始地址开始的顺序单元中。这些支持向量传送和字符行传送的指令有利于缩短目标程序的长度,也便于进行汇编语言程序的设计。

由于对向量、数组等处理的需要,也要在寄存器之间或寄存器与处理部件之间设置一些一次就能传送多个数据的指令,如向量计算机中的向量传送指令等。这在向量处理机中将会详细讲述。

(2)增强程序控制指令的功能

控制类指令主要包括条件分支、无条件跳转、过程调用和过程返回。这类指令的使用频度也是比较大的。如使用基准测试程序 SPECint92 在一台 loda/ store 型指令集结构的机器上运行,对指令的使用频度进行分析,得出表 2.9 的统计结果。

表 2.9 中的数据是 SPECint92 的 5 个程序 compress, espresso, eqntott, gcc, li 运行结果的平均值。从表 2.9 中可看到,常用的指令仍然是前面所述的三种,控制流指

令中的条件分支指令的使用频度高达 20%,而其他三类,即调用和返回、无条件跳转相对低一些。同样,在一台 Load/ store 型指令集结构的机器上运行 SPECint92 和 SPECfp92 基准程序,测试控制类指令出现的相对频度如表 2 .10 所示。

根据表 2 .10 中的统计结果,控制类指令中使用最多的是条件分支指令,因此在进行条件分支指令设计时,确定如何表示分支条件就非常重要。现在常用的比较条件分支的技术有三种:第一种是条件码(Condition Code,CC),由 ALU 操作设定的某些特定位,在程序的控制下,从一条指令将分支条件信息传送到分支指令,CC 是额外状态,它限制了指令顺序,但可以自由设置分支条件。第二种是条件寄存器,根据比较结果测试条件寄存器,这种方式简单,但占用一个寄存器。第三种是比较且分支,即把比较和分支两条指令功能合在一条指令中完成,使分支指令的操作增多,这种测试条件分支的方法通常受一定的限制,比较只限于子集内部。如在 80×86 中就有一个专用的标志寄存器,标志寄存器中就有条件码 CF,ZF 等状态位。条件分支指令根据这些状态位实现相应的转移。常用的控制指令有 JNZ, JZ, JMP, JMPF, CALL, CALLF, RET, RETF, LOOP 等。

表 2 .9　　80X86 的最常用的 10 种指令

排名	80×86 指令	整数平均值(占所有执行指令的百分比)
1	取数(load)	22%
2	条件分支 (conditional branch)	20%
3	比较(compare)	16%
4	存数(store)	12%
5	加(add)	8%
6	逻辑与(and)	6%
7	减(sub)	5%
8	寄存器传送(move register - register)	4%
9	调用(call)	1%
10	返回(return)	1%
总计		96%

表 2 .10　　控制指令的使用频度

控制类指令	整型平均相对频度%	浮点平均相对频度%
调用/ 返回	13	11
跳转	6	4
条件分支	81	87

(3)增强运算型指令的功能

可通过增设强功能复合指令来取代原先由常用宏指令或子程序实现的功能。如在科学计算的计算机中,为减少程序调用的额外开销,减少子程序所占用的主存空间,加快运算型指令的速度,常设置有复杂函数运算类指令,如开平方、求三角函数、对数函数、指令函数等。在一些经常进行事务处理的计算机中,设置有二—十进制转换、编辑、翻译等指令。例如,IBM370 上增设的翻译指令,属于 SS 型指令,它可以完成码制转换,如 ASCII 码与 BCD 码之间的转换。指令实现的功能如下:

翻译	L	B1	D1	B2	D2
8 位	8 位	4 位	12 位	4 位	12 位

((B2 + D2) + ((B1) + D1 + G))→(B1) + D1 + G

其中,G = 0,1,…,L - 1。即将由(B1) + D1 形成的主存字节单元起始,连续 L 个字符,逐个通过查变换表进行变换,而指令中的(B2) + D2 指明变换表在主存中的起始地址。

用新指令替代常用指令串的办法实际上是尽量减少程序中如存、取、传送、转移、比较等不执行数据变换的非功能型指令的使用,让真正执行数据变换的加、减、乘、除等功能型指令所占的比例提高。有时也以程序中功能型指令与非功能型指令的条数比值作为衡量计算机系统结构设计好坏的一个标志,不过,这个比值也与所选定的工作负荷有很大关系,在实际使用时,要在指令的执行速度、使用频度、复杂度等多方面综合分析和权衡。

2. 面向高级语言及其编译程序的指令功能的优化与改进

今天几乎所有的程序都是用高级语言编写的,而高级语言源程序必须经相应的编译程序编译生成目标程序后才能在机器上运行,这与直接用机器语言或汇编语言编写的程序相比,时间开销和空间开销都要大一个数量级,因此,从面向高级语言及编译程序来改进指令的功能,主要是为了缩小高级语言与机器语言之间的语义差距,支持编译系统,使编译过程加快,目标程序形成效率高。它的主要改进方法有如下方面:

(1)用指令实现高级语言中的高频度语句

这种改进方法也是对语句的使用频度进行统计,和前面不同的是对各种高级语言源程序中的不同的指令进行统计,根据统计结果,对常用的指令或指令串提供支持。如 IBM370 中为优化循环时的循环控制的辅助操作而增设的“小于等于转移指令”、“计数转移指令”等。

(2)从编译系统代码生成的优化算法的要求,使指令系统更加规整和对称

编译优化应使各存储单元和寄存器在指令系统中都具有同等的地位,运算数据可以取自任意单元,存入任意单元,这样可以提高编译速度,使名称符号易于与暂存

单元对应,有利于减少名称暂存单元的冗余,减少一些附加操作程序指令。如在IBM370中就有许多不规整、不对称的地方。如通用寄存器并不真正通用,有的指令只能使用指定的几个寄存器,对寄存器使用的种种限制,造成在编译时凡指令要用到这些指定的寄存器时,必须先腾空再移入数据,增加了不必要的数据传送次数,使通用寄存器的优化管理复杂化。像存、取、加、减和乘这类指令,都有对应的全字(32位)、半字(16位)指令,但除法指令却只有全字的,这些都是IBM370不对称的地方。还有一些特殊使用也给软件的设计带来不便,如"十—二进制"指令,其溢出却是以"定点除溢出"形式出现。

(3)指令系统对高级语言的支持,应保持对各种高级语言的语义差距的一致性

高级语言和机器语言之间存在着很大的语义差距,如果试图适应某一种语言结构的特殊性质而设计指令,通常却难以使用。而且多种语言同时存在,它们之间的语法结构差别也很大。为可以对更多的高级语言提供支持,应设法提供基本原语,而不是解决方案,使指令集结构能与各种高级语言的语义差距都有共同的缩小。如IBM370中的"小于等于转移"复合指令,对各种语言的循环控制都有支持,这也符合软硬取舍的原则。

微程序的发展,特别是可读写控存的采用,可采用动态指令系统结构,分别面向不同的高级语言。在不同语言环境下,引入不同的指令系统,由可读写控存分别装入不同指令系统的解释微程序,其余的微程序放在主存中,使用时调入,这样使编译工作量减少,而代之解释过程。如1972年的B-1700就是这样一种思路。它为BASIC,FORTRAN,COBOL与RPG,SDL语言每种都设计了一个对应的系统结构,这种系统结构由微程序解释实现。其中,因COBOL与RPG的语义相近,所以这两种语言共用一种系统结构。SDL语言的系统结构是为操作系统提供支持的。各种系统结构都有自己的指令系统和数据表示,对这种高级语言来说都是最优化的,这样,B-1700就具有多种系统结构,在运行过程中根据需要动态地切换。

(4)发展高级语言机器

如果进一步缩小高级语言和机器语言之间的语义差距,不断减小编译的工作量,而增大解释的比例,最后发展出来的就是高级语言计算机,不再需要编译优化。如高级语言作为计算机的汇编语言来使用,称为间接执行高级语言计算机,如直接作为机器语言来使用,则称为直接执行高级语言计算机。20世纪70年代初出现过一些高级语言计算机,但性能价格比不理想,没有产生过巨大的商业影响。且在20世纪80年代初期,RISC思想的萌芽,使计算机系统结构的方向,开始由为语言提供高级硬件支持的方向后退。

3. 面向操作系统的指令功能的优化与改进

比起高级语言来,操作系统对系统结构的依赖性更大。如没有系统结构提供的中断响应硬件支持,实时操作系统就很难实现。在操作系统中,直接由指令系统支持

的功能主要有:

- 进程管理:进程切换、进程的生成与撤消、进程间的同步与通信;
- 存储管理:存储空间的分配、页表的管理等;
- 保护:存储保护、程序保护和数据保护等。

像以上这些“机构型”的功能(基本的、较固定的功能)适宜于硬件实现,而一些“策略型”的功能(会随不同的环境而异,而且用户应能修改的功能),如作业排队、资源管理、进程优先级的确定等,是不稳定的,会随环境不同而发生变化,它在操作系统的运行期间会不断变化,所以不适于硬件实现,而适于软件实现。具体的支持途径有三条:

(1)用硬件实现一些管理类指令

如 IBM370 的“测试与置定”指令、“比较与交换”指令,是为了支持多个进程正确使用公用区的管理而增设的专用指令。

(2)用固件实现一些使用频繁、对速度影响大的子程序(HOT SPORTS 过热点)

固件可做扩展固件,可保证与原系统的兼容。

不少机器(如 IBM370)是将操作系统中的下述过热点采用固化和硬化实现,它们是:

- 进管处理:对要求进入管态,调用 OS 的申请进行分析,并按它执行 N 向转移,进入相应的管理程序。
- Ⅰ/O 中断处理:对中断申请进行分析并转移到相应的中断处理程序。
- 通道控制字翻译程序:对通道控制字进行翻译并传送到实存。
- Ⅰ/O 工作区的管理:动态地对 Ⅰ/O 工作区进行分配。
- 虚拟存储管理:对不同容量要求的各个用户进行主存分配。
- 页面管理:在辅存与主存之间进行页面交换,等等。

如 IBN370 配上这种选件可提高 OS 速度 20%以上。

(3)用固件实现原语

可避免中断的执行干扰关键程序段,又可减少中断开、闭的辅助操作。

总之,正确的指令系统改善途径是:在尽量缩小语义差距的前提下,充分发挥软、硬件的特长,即硬件实现可提高系统执行效率和速度,减少系统管理调度的开销,而软件实现能提供较高的灵活性和较复杂常变的算法。今后指令系统功能的改进,还将进一步对应用软件提供支持,如支持数据库软件的比较、搜索、排序等处理过程,这可提高计算机对事务处理应用的支持能力。

2.4.3 精简指令系统计算机(RISC)设计风格

RISC(Reduced Instruction Set Computer),指令系统功能优化和改进的另一种方向,它和 CISC 的指导思想完全相反,是研究如何通过减少指令总数和简化指令的功能来降低硬件设计的复杂度,提高指令的执行速度。沿着这条途径和方向发展,使

计算机指令系统精练简单,因此,称采用这种途径设计成 CPU 的计算机为精简指令系统计算机 RISC。当今 RISC 的典型代表如 Hewlett Packard PA - RISC, IBM 和 Motorola PowerPC, SGI MIPS,最初由 Sun Microsystem 开发的 SPARC, DEC Alpha, Motorola 88110, Intel 80860, 80960。另外,在有些典型的 CISC 处理机中也采用了 RISC 设计思想,如 Intel 公司的 80486, Pentium, Pentium Pro, Pentium Ⅱ及以后的产品。

1. 精简指令系统的设计思想

最早的计算机,包括 UNIVAC I, EDSAC 和 IAS 计算机,都是基于累加器的计算机,这类计算机的简单性,在硬件资源十分有限的条件下成了很自然的选择。从这些简单计算机的出现,特别是 1964 年 IBM360 系列机推出之后,人们一直在改进计算机的结构,不断地改进指令的功能,不断地增强机器的功能,强调的都是如何为高级语言、操作系统、程序设计环境及应用等提供更多更好的硬件支持,缩小它们与系统结构和机器语言之间的语义差距。就像上面所讲的,这种改进和发展的计算机就是 CISC。但是到了 20 世纪 70 年代中期以后,开始感到采用这种日益庞大复杂的指令系统不但实现起来越来越困难,实际上还有可能降低整个系统的性能。因此,1975 年 IBM 公司就组织力量研究采用复杂的指令系统是否合理的问题,在 John Coche 领导下,于 1979 年研制出了 32 位的 IBM 801 小型计算机。IBM 801 计算机是为 100 万门交换机而设计的,它作为大型电话交换系统的高速控制器,当时要求探索新的设计途径,性能指标要求达到 6MIPS。在设计这台计算机时,John Coche 等人对指令系统作了精选,只选取了少量简单、使用频度高的指令来构成指令系统,并设法使每条指令都在一个周期内执行完毕,IBM 801 只有 120 条指令,指令字长固定为 32 位,采用硬联线控制,设有 32 个通用寄存器,两个 cache(指令 cache 和数据 cache),强调了流水线的实现。由于采用了这一系列的措施,尽管在当时还没有 VLSI 芯片,仍使 IBM 801 性能达到了 10 MIPS,超过了原先的 6 MIPS 的设计要求。虽然早在 1964 年, Control Data 发表的第一台超级计算机,即 CDC 6600, 该机的设计者们为了流水线的效率,采用了简单的系统结构,但 IBM 801 仍是计算机领域首先使用 RISC 思想研制出的计算机, IBM 801 是一个试验项目, IBM 从来没有把 IBM 801 直接转化成产品,而是采纳了它的设计思想,它是 1986 年正式推出的 IBM RT-PC 工作站的雏形。

1979 年,美国加州大学伯克利分校以 David Patterson 为首的研究小组进一步开展了这方面的研究工作,他们发现, CISC 并没有使计算机系统性能有明显提高,反而与软硬件的发展出现了很不协调的情况,其原因主要是 CISC 存在如下一些问题:

(1)指令系统复杂

具体表现在以下几个方面:

CISC 的指令条数一般多于 100 条,寻址方式大于 4 种,指令格式大于 4 种。如

VAX 11/ 780(1978)包括 304 条指令,24 种寻址方式,2～57 个字节不等的可变长指令格式,微代码存储器 480KB,而 1985 年公布的 Intel 386 有 111 条指令,11 种寻址方式,1～17 个字节不等的可变长指令格式。这么庞大的指令系统,多种指令格式和复杂的寻址操作,使指令译码、硬件设计复杂化,这不利于 VLSI 的设计,复杂的指令还需复杂的控制器,这使微程序执行速度也很难提高,同样延长了设计周期,降低了系统的可靠性。

(2)有些复合以后的指令,并不比几条简单的指令执行速度快

在 CISC 中,为了支持目标程序的优化,缩小高级语言与机器语言之间的语义差距,增加了许多复杂指令,用一条这样的指令代替原来的一串指令,但为了实现这些指令的功能,不仅仅是增加了硬件的复杂度,而且使指令的执行周期大大加长了。据统计,一般 CISC 处理机的指令平均执行周期都在 4 个以上,有些在 10 个以上,如 Intel 公司的 8088, Motorola 公司的 MC68010 等。

(3)复杂的指令系统使编译效率难以提高,编译程序复杂

编译程序的基本任务是完成大量的各种情况的分析,生成高效的目标代码。指令系统越复杂,选择目标指令的范围越大,则分析情况的数目就越多,分析就越困难,需要的时间就越长,而且就更加难以获得优化的目标程序。

(4)扩展的指令使用频度不很高

在 CISC 中,各种指令的使用频度相差很大,大量的统计数字表明,存在这样一个规律:约有 80%的指令只有 20%的时间用到。如在表 2 .9 中的统计结果,80×86 中常用的指令只有 10 条,占了总指令条数的 96%。所以说,扩展指令对提高整机性能价格比不利。

针对 CISC 结构存在的这些问题, Patterson 等人首先提出了 RISC 这一术语, RISC 是一种计算机系统结构的设计思想,它不是一种产品。由 Patterson 领导的研究组,根据这一思想,先后研制了 RISC-Ⅰ和 RISC-Ⅱ计算机。RISC-Ⅰ是 32 位的微处理器,1981 年研制成功,总共有 31 条指令(算术逻辑运算 12 条,访问存储器指令 8 条,程序控制指令 7 条,其他指令 4 条),3 种数据类型,2 种寻址方式(变址寻址和 PC 相对寻址)。寻址单位为字节,指令字长是 32 位,采用三地址制,有少量指令采用二地址和一地址,时钟频率 8MHz,所有指令都在一个周期(500ns)内完成,只有 LOAD/ STORE 指令可以访问存储器,其他指令的操作都在通用寄存器之间进行。设置了 78 个通用寄存器,采用重叠寄存器窗口技术(后面将讲到),使用 NMOS VLSI 电路,控制芯片的面积只占约 6%, MC68000 为 50%, Z8000 为 53%,设计错误和布线错误都只有大约 12 个,而 MC68000 的设计错误和布线错误各有 70 个, Z8000 为 60 个和 100 个。研制周期只用了 10 个月,其性能却比当时最先进的商品化微处理器 MC68000 和 Z8000 快 3～4 倍,有些方面还超过了小型机 PDP-11/ 70 和 VAX11/ 780。1983 年 RISC-Ⅱ研制成功,采用与 RISC-Ⅰ相同的 NMOS 工艺,指令系统扩充到 39 条(增加了采用变址寻址方式的取数指令 5 种和存数指令 3 种),通用

寄存器增加到 138 个,时钟频率提高到 12MHz,指令执行周期缩短到 330ns,存取指令只能使用变址寻址一种方式,设计错误大约只有 18 个,布线错误大约只有 12 个,控制部分只占 CPU 总面积的 10%。

1981 年美国的斯坦福大学在 J .Hemnessy 教授领导下研制了 MIPS(Microprocessor Without Interlocking Pipeline Stages,无互锁多级流水线处理器)的 RISC 芯片,强调流水高效实现和采用编译方法进行流水调度,使 RISC 技术设计风格得到很大补充和发展。1985 年后推出商品化的 MIPS2000RISC 机(1986),1987 年 SUN 公司推出了基于伯克利分校 RISC 机的 SPARC 系统结构,从此以后,RISC 技术在计算机工业界被广泛采用。从 1985 年以后所宣布的任何计算机,基本上都采用了 RISC 技术。那么,什么样的计算机才是 RISC 计算机呢?RISC 计算机应有哪些特点呢?主要特点归纳如下:

① 大多数指令在单周期内完成。

② 采用 Load/ Store 结构:尽量将运算的数据存放在寄存器中,从而减少访问 RAM 的次数。

③ 硬布线控制逻辑:真正由硬件完成,而不是通过 Micro program 完成。

④ 减少指令和寻址方式的种类。

⑤ 固定的指令格式。

⑥ 注重译码优化。

⑦ 面向寄存器设计指令系统。

⑧ 注重 pipeline 的效率设计。

⑨ 重视优化编译设计。

2. RISC 基本设计原则

由于 RISC 是针对 CISC 结构存在的问题而提出的另外一种完全相反的设计思想,在具体设计 RISC 处理器时,指令集的选取就应当遵循一些原则,主要包括如下几个方面:

(1)采用寄存器间的运算结构

除 LOAD 和 STORE 指令以外,其余指令都在寄存器之间进行,使编译优化,尽量减少访存的次数。

(2)选取核心的和较常用的指令

确定指令系统时,应选择高频度指令,在此基础上增加少量扩展指令,使指令条数尽量少。

(3)采用单字指令、固定操作数域

这样,90%以上的指令可在一个执行周期内完成,访存可通过寄存器间址方式扩展寻址范围。

(4)硬联控制为主,固件实现为辅

所有简单指令直接用硬件译码,由硬件提供 Cache, I/O,虚存管理、内存管理等的支持,提高指令执行速度,少数指令采用微程序实现。

(5)支持高级语言

对一些既实现简单,又对高级语言支持较大的复杂指令,如 CALL 指令用微程序来实现,再加上编译的优化及协处理器的支持,使 RISC 能更好地支持高级语言环境。

3. RISC 实现的关键技术

如上所述,RISC 之所以有如此良好的性能,除了在设计 RISC 处理机时应遵守的那些原则外,主要是由于它采用了一些特殊实现技术,目前,在 RISC 处理机中主要采用如下几种技术:

(1)重叠寄存器窗口技术(overlapping register windows)

CISC 中的一条复杂指令,在 RISC 中通常要用一段由简单指令组成的程序段来实现,因此,RISC 中的过程调用(CALL)和返回指令(RETURN)要比 CISC 中的多。据统计,在 PASCAL 语言和 C 语言中分别有 15%和 12%的 CALL 和 RETURN 操作,而它们访问存储器的信息量却占整个访存信息量的 44%和 45%,这是因为这两种指令需要传递大量的参数,因此缩短 CALL 语句和 RETURN 语句的操作时间在 RISC 结构中就非常重要。

为减少 CALL 和 RETURN 大量的访存操作,美国加州伯克利分校的 F .Baskett 提出了重叠寄存器窗口技术,它的基本思路就是在处理机中设置一个数量较大的寄存器堆,并把它分为若干个寄存器组,每组有若干个寄存器,称为寄存器窗口。每个寄存器窗口中,又分成大小固定的三个区:高区、本区和低区。每当有过程调用时,就分配一个未被使用的寄存器窗口,在这个寄存器窗口中,高区用来存放调用过程传来的参数,在本过程返回时用来存放要返回给调用者的参数。本区用来存放局部变量,只有本过程可以访问,而低区作为本过程调用其他过程时传递参数给被调用过程,被调用过程执行完时送回的结果也存放在此区中。在使用时,每一对调用和被调用过程的寄存器窗口各自的低区和高区相互重叠。一旦发生过程调用或返回,由一个窗口转换到另一窗口时,这些参数就通过两个窗口间的重叠的寄存器部分自动地被传送,而不再需要额外的传送时间。另外还要设置一个为所有过程公用的寄存器组,用来存放全局变量,所有的过程都可以直接访问。

重叠寄存器窗口技术首先在 RISC-Ⅰ上应用,后又在 RISC-Ⅱ上实现,下面就以 RISC-Ⅱ为例,说明这种技术的应用。

RISC-Ⅱ机共有 138 个 32 位的工作寄存器,编号从 0～137,其中第 0 号至第 9 号共 10 个寄存器中存放全局变量,可被所有正在系统中运行的程序或过程直接访问,故称为全局寄存器。剩下的 128 个寄存器分为 8 个寄存器窗口,每个窗口中有 22 个逻辑寄存器,编号为 R10 到 R31,它们被分为三部分:其中逻辑编号为 R10 到

R15 的 6 个寄存器为低区,存放输出参数(本过程是主调程序);编号为 R16 到 R25 的 10 个寄存器为本区,存放只有本过程访问的局部变量;R26 到 R31 的 6 个寄存器称为高区,存放输入参数(本过程是被调用程序)。这样,每个过程可以直接使用的寄存器总共有 32 个。整个系统共有 8 个窗口,窗口号是 0～7。只要调用的深度不超过 8 层,重叠寄存器窗口技术可以减少大量的访存操作,当调用层数超过 8 层时,称为寄存器溢出,这时就需将一个窗口中的内容传送到主存,以腾出一个空窗口。实验表明,调用深度一般不会超过 6 层,发生寄存器溢出的机会大约只占 1%。

图 2 .16 中所示的是 A 过程调用 B 过程、B 过程又调用 C 过程的情况。A 的低区和 B 的高区重叠,B 的低区和 C 的高区重叠,这样,不需要花费任何附加的操作时间就可以实现调用和被调用过程的参数传递,因而可以大量减少调用和返回时的访存信息量。

表 2 .11 是 RISC-Ⅱ和 VAX-11,PDP-11,MC 68000 执行调用和返回时的开销比较,可以看出,在执行时间、执行指令条数、访问存储器次数方面,采用重叠寄存器窗口技术都是非常有效的。

表 2 .11　　过程调用/ 返回的开销

计算机	执行时间(μs)	执行指令的条数	访问存储器的次数
VAX-11	26	5	19
PDP-11	22	19	15
MC 68000	19	9	12
RISC-Ⅱ	2	6	0 .2

(2)采用流水和优化延迟转移技术

RISC 中由于大部分指令都可以在一个机器周期内执行完毕,很适合流水处理。为了加快计算机中的指令执行速度,一般都采用了让本条指令的执行和下条指令的预取在时间上重叠起来的流水方式。取指令和执行还可进一步流水处理,如在指令的执行阶段,又可将从源寄存器读操作数、运算及运算结果打入目的寄存器三者之间采用流水方式实现,具体流水线的级数因计算机而异。

正常情况下,流水处理每一个机器周期就能执行完一条指令。然而,在遇到转移指令且转移成功时,流水线就要断流。在 CISC 机中,为实现灵活转移,一般将比较和转移功能分为两条指令,这样比较指令结束后才能对相应条件码进行置位,转移指令才能根据条件码进行相应的处理。RISC 中,为加快转移指令的处理,减少断流时间,一般将这两条指令合并成一条“比较转移”指令,该指令将直接对两个对象进行相等或不等比较,然后根据比较结果判别是否进行转移,若转移成功,转移目标地址也只能在这条指令执行完成后才能计算出来,这时必须使流水线停顿一段时间,直到转

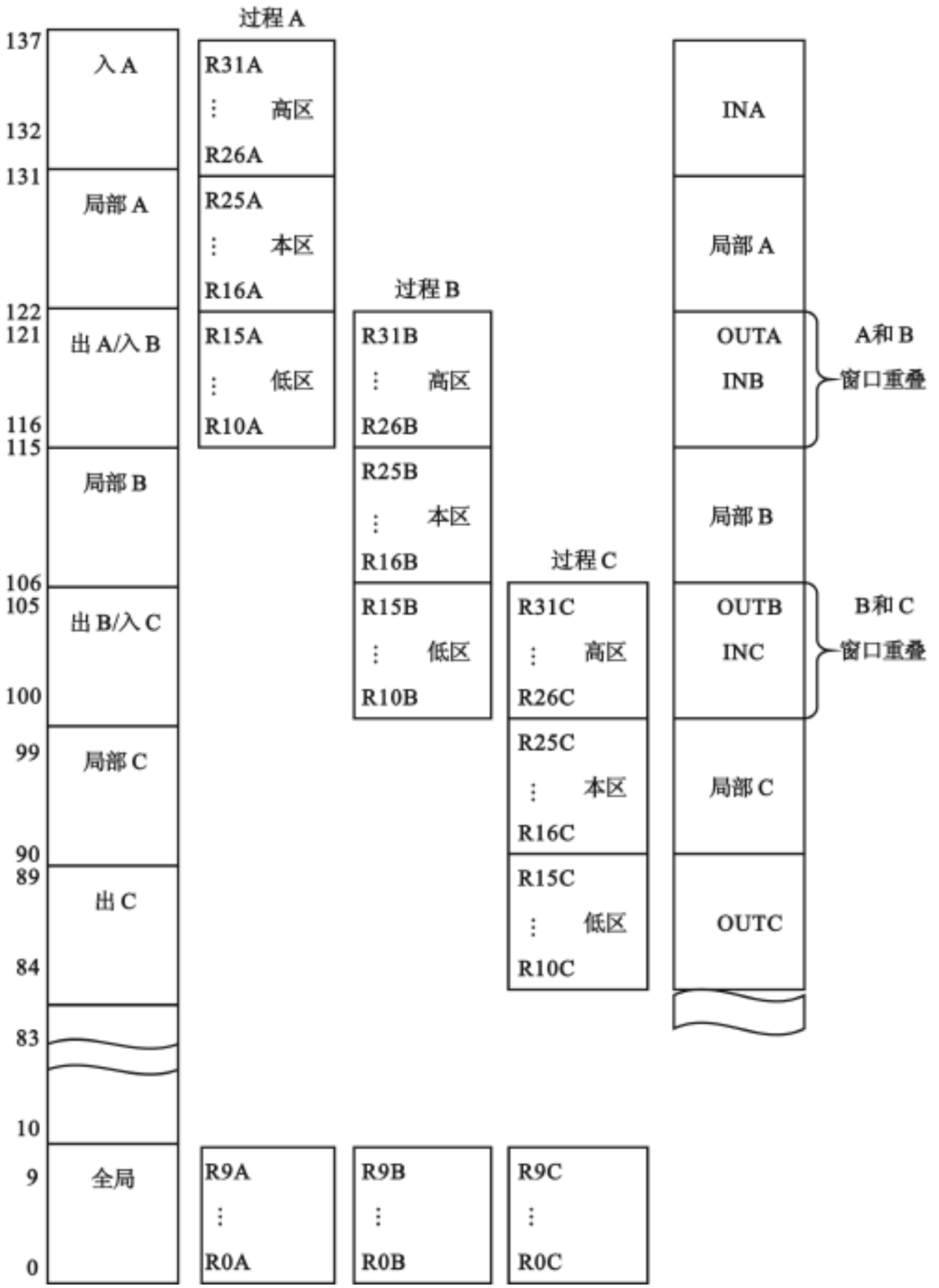


图 2.16　RISC-Ⅱ的重叠寄存器窗口

移目标地址已产生为止。在转移指令后,需延迟到后继指令进入流水线的时间段称

为转移延迟槽。转移目标地址生成得越晚,转移延迟槽就越长,因转移指令使用频度较大,为尽量减少转移指令对流水线性能的影响,减少损失,通常在延迟槽内插入一些指令,这就需要采用延迟转移技术。

延迟转移技术就是在转移延迟槽中插入空指令,减少转移方向错误的损失。虽然这种方法可避免预取的转移指令后的下一条指令作废,减少访存开销,但这同样要浪费一个机器周期,如果在转移延迟槽中插入一条有效指令,即这条指令的执行不影响转移指令的执行和转移后后续指令执行的结果,也不影响指令的流水执行,这样就可少花费一个机器周期,这种技术就称为优化延迟转移技术,如表 2 .12 所示。

表 2 .12 中示出了一小段程序,其中,R1,R2,R3 为寄存器单元,X,Y 为主存单元。表中第二列为不采用延迟转移技术的常规转移方法,如果执行到地址为 102 处的转移指令“ BRANCH 105 ”时,如果转移成功,则采用流水方式预取的下一条指令“ ADD R1,R2 ”将作废,否则将导致错误。第三列为在转移指令后插入一条空指令,这样,即使转移成功了,也可直接从目的地址重新取指令执行,不影响程序的结果。但执行空操作同样浪费了一个机器周期,为避免这种浪费,如果将 101 处指令和 102 处的转移指令的顺序对调一下,且原 101 处的指令的执行不影响转移指令的执行和其后续指令的执行结果,如第四列所示,则采用这种方法后,即使转移成功也是在其后的指令执行完毕后才发生,从而使预取的指令不作废,就可减少一个机器周期,这就是优化延迟转移技术。

表 2 .12　　延迟转移的一个实例

地址	常规转移	延迟转移	优化延迟转移
100	LOAD　X,R1	LOAD　X,R1	LOAD　X,R1
101	ADD　1,R1	ADD　1,R1	BRANCH　105
102	BRANCH　105	BRANCH　106	ADD　1,R1
103	ADD　R1,R2	NOP	ADD　R1,R2
104	SUB　R3,R1	ADD　R1,R2	SUB　R3,R2
105	STORE　R1,Y	SUB　R3,R2	STORE　R1,Y
106		STORE　R1,Y	

在优化延迟转移技术中,插入的这条有效指令如何选取呢?一般来说,如果转移指令的前一指令与转移指令不相关,则直接插入转移指令前一指令即可,如上表所示。如果转移指令前一指令与转移指令相关,则可以选取某转移方向指令插入转移延迟槽。当然,这部分工作是由编译程序自动进行的。所以说,优化延迟转移技术对应用程序员透明,对系统程序员不透明。这也说明了在 RISC 计算机中,流水属于系统结构,不像一般的流水机中流水只属于计算机组成。

模拟统计结果表明:RISC 的转移延迟槽一般为一条指令,填入转移延迟槽中的

指令,其中 50%以上的将是有效指令。

如果遇到条件转移指令时,调整指令序列非常困难,在许多情况下找不到可以用来调整的有效指令,有些 RISC 处理机中采用指令取消技术。如果指令被取消,相当于执行了一条空操作指令,不影响流水线的运行,但为了减少指令取消带来的流水线效率的降低,应该尽量少取消指令。因此,可采用如下规则:

如果转移目标地址小于当前程序计数器 PC 值的向后转移,则在转移不成功时取消下条指令,否则执行下条指令;如果转移目标地址大于当前程序计数器 PC 值的向前转移,则正好相反,在转移不成功时执行下条指令,否则,取消下条指令。

向后转移指令取消技术的例子:

```
LOOP: XXX                          XXX
      YYY                    LOOP: YYY
      …                            …
      ZZZ                          ZZZ
      COMP R1, R2, LOOP            COMP R1, R2, LOOP
      WWW                          XXX
                                   WWW
```

(a) 调整前的程序　　(b) 调整后的程序

循环体的第一条指令 XXX 经调整后安排在两个位置,第一个是进入循环体之前先要执行一次,第二个是在循环条件判断指令的下面,执行完条件判断指令后,如果转移成功,则 XXX 指令的执行有效,接着返回到 LOOP,再重新进入循环;如果转移不成功,则取消 XXX 指令的执行,接着执行 WWW 指令。

由于向后转移时,绝大多数情况下是转移成功的,只有在循环结束时转移才不成功,因此,采用这种指令取消技术能够使指令流水线在绝大多数情况下不断流,保持高的流水线效率。

向前转移的情况,主要适用于像 IF… …THEN 这样的控制结构,如下面一段程序:

```
      XXX
      …      } “IF”部分的程序代码
      YYY
      COMP R1, R2, THUR
      ZZZ
      …      } “THEN”部分的程序代码
      WWW
THUR: VVV
```

由于向前转移成功与不成功的概率通常各为 50%,因此,采用常规的指令取消技术就可以了。在上面的程序中,如比较指令 COMP 指令的转移条件成立,则取消

下条指令 ZZZ 的执行,程序转向 THUR 位置,“THEN”部分的指令全部不执行;如转移不成功,则下条指令 ZZZ 不取消,“THEN”部分的指令全部执行。

另外,还有一种情况,即在条件分支中只有一条指令的情况,可以采用隐含转移技术,例如实现下面的语句:

IF(a < b) THEN b = b + 1

用汇编语言改写为:

COMP > = , Ra, Rb; 通用寄存器 Ra, Rb 中分别存放变量 a 和 b

INC Rb

指令 COMP 如转移成功,则取消下条指令,否则,执行下条指令 INC。

这里要特别注意的是,调整指令序列绝对不能改变原来程序的数据相关关系,即被移动指令中所有的数据存储单元与移动过程中所涉及指令的数据存储单元之间不能有数据的读—写、写—读、写—写相关。而且,被调整的指令也不能破坏 CPU 的条件码。如果不能满足上述条件,则只能在转移指令后,插入一条 NOP 指令。

(3)在逻辑上采用硬件实现为主,固件实现为辅的技术

用微程序实现机器指令的主要优点是容易实现复杂指令的功能,且指令系统容易修改,增加了计算机的灵活性和适应性,但由于多次访问控制存储器取微指令要花费一定的时间,所以降低了指令的执行速度。而在 RISC 中要求绝大多数的指令都是单周期指令,因此,在 RISC 计算机中,一般采用让大多数的简单指令用硬联方式实现,少数复杂指令可用微程序解释实现,且最好采用高度水平型微指令,即用固件的方式实现。现在大多数商用的 RISC 处理机中指令系统的实现方法,都采用以硬件为主固件为辅的方法。

(4)采用优化编译技术

RISC 的处理机在采用硬件技术提高性能的同时,优化编译技术也起着非常重要的作用。优化的目的主要是对程序重新排序和调度,优化代码顺序,减少目标代码长度,提高程序运行效率。

在 RISC 中所采用的优化编译技术,除了要充分利用常规的优化技术和手段进行编译程序设计外,还要注意以下方面的优化设计:高级优化阶段中对原始程序进行优化(如用过程体代替过程调用);局部优化阶段的块内优化(如消去公共子表达式、常数传递、降低堆栈高度等);全局优化阶段的块间优化(如拷贝传递、代码移动、消去索引变量等);寄存器的分配(提高寄存器的利用率,这一步较少依赖计算机);代码生成阶段(利用计算机的特点进行优化,如降低计算量、流水线调度、分支偏移的优化,这一步需依赖计算机),主要突出了两个方面的优化调度:一是如何最佳地分配 RISC 中大量寄存器的使用,从而减少访问存储器的次数;二是设法对程序中的指令序列进行调整,在保持原来语义正确的基础上尽量减少计算机的空等时间。另外,还可设法对指令序列进行调整,消除指令中不必要的等待时间。

例如:设 A, A + 1, B, B + 1 为主存单元,则程序段

```
LOAD    R1,A
STORE    B,R1
LOAD    R1,A+1
STORE    B+1,R1
```

该程序实现的是将主存单元 A,A+1 的内容传送到 B,B+1 两个主存单元中去。因 LOAD 和 STORE 指令交替进行,都要用到同一个寄存器 R1,所以上条指令没有执行完,下条指令就无法执行,这不利于流水处理,如果将这段程序改为如下的程序段:

```
LOAD    R1,A
LOAD    R2,A+1
STORE    B,R1
STORE    B+1,R2
```

则实现的功能和上面的程序段一样,但这四条指令可流水执行,这样每隔一个机器周期就解释完一条指令,速度提高了 1 倍。

综上所述,RISC 中的优化编译技术,主要是为了在编译时就可发现可能出现的阻塞情况,由编译器通过寄存器置换、调整指令序列等手段来消除可能出现的阻塞情况,从而使流水线的运行效率达到最高。当无法消除时就填入相应的空操作,因此不需要硬件的互锁流水支持。

以上所述的这些实现技术在目前流行的 RISC 计算机中经常采用,但并不是在每一种 RISC 处理机中都必须采用以上的所有技术。实际上,在 RISC 处理机的设计上,主要有两个典型代表。一个是遵循加州大学伯克利分校的 RISC-Ⅰ,RISC-Ⅱ机的思路,侧重采用由大量寄存器组成的寄存器组,以及寄存器窗口重叠技术;另一个则是遵循斯坦福大学的 MIPS 计算机的思路,侧重采用优化编译技术。

如 RISC-Ⅱ中,采用了多寄存器和寄存器窗口重叠技术,同时也采用了优化延迟转移技术。该机器的主要设计思想在以后的 SUN SPARC 系统结构中被广泛采用,而斯坦福大学的 MIPS(Microprocessor without Interlocked Pipeline Stage,无互锁的多级流水线处理机)系列,即 MIPS R2000/ 3000/ 4000/ 4400/ 10000 等则主要侧重于借助软件手段,特别是优化编译来提高计算机性能。它们的主要特征就是不采用硬件互锁流水,而依靠优化编译器进行指令序列的调整,防止流水线中出现的相关冲突,没有采用寄存器窗口重叠技术。

2.4.4 CISC 机和 RISC 机的比较

RISC 是针对 CISC 中存在的问题而提出的一种指令系统的设计思路,和 CISC 机相比,RISC 机具有以下特点:

1. RISC 系统结构的主要特点

(1) 强调系统结构与 VLSI 实现技术之间的配合

硬件实现要求逻辑电路具有简单性、规格化和高度重复性。在 CISC 处理器中，指令译码和控制电路相当复杂，控制逻辑占用芯片面积的 50%以上。而在 RISC 处理器中，由于指令简单、规整，方便 VLSI 设计和验证，适合 VLSI 技术的实现。控制器和 CISC 相比也很简单，控制逻辑只占用芯片面积的 10%左右，这样就可以有多余的空间来加强处理器的其他功能，除了前面所讲的寄存器窗口重叠技术外，还可以广泛采用高速缓冲技术提高处理机的速度。

(2) 强调系统结构与编译器的优化配合

前面所述的延迟转移技术、延迟取数等指令流调整的优化编译技术，可充分利用寄存器的快速访问及流水处理的高速处理。具体方法在第五章介绍。

(3) 强调流水线技术及对并行度的开发

由于单周期的简单指令有利于流水处理，所以执行速度可提高很多。另外，高速缓存的采用，加快了 CPU 对内存的访问(第四章详述)，有利于流水处理。

由于 RISC 可节省空间，可加入多处理机相连的接口，如 Transputer T800 中有一 100MB/ S 的通信连接接口，可直接使处理器之间互连成多机并行系统。如 CONNECT MACHINE 是一个多达 65 536 个处理单元、系统峰值速度可达 28GFLOPS 多机并行系统。

另外还出现了几种新型结构的 RISC，如超级流水线结构、超级标量结构、超长指令字等。这将在第五章中详细介绍。

如果用 CPU 性能公式来比较这两种不同风格的处理机，则 CISC 处理机主要是减少了指令的条数，RISC 主要是减少了指令平均执行周期数，虽然就目标代码来讲，RISC 的目标代码可能比 CISC 中长 2～3 倍，但由于流水处理和优化编译技术的使用，总体上 RISC 的执行速度是 CISC 的 5～10 倍，所以总的处理速度可成倍提高。

2. RISC 系统结构存在的某些不足和问题

(1) 要设计复杂的子程序库

因为在 CISC 中的一条复杂指令，在 RISC 中要用一段子程序来实现，所以，RISC 的子程序库通常要比 CISC 的子程序库大得多。

(2) 对编译器要求较高

优化编译器必须精心安排每一个寄存器的使用，以充分发挥通用寄存器的功能，尽量减少访存次数；同时，优化编译器还要做数据和控制相关性分析，要设法调整指令序列，减少流水线断流的发生。

(3) 对浮点运算和虚拟存储器的支持虽有很大加强，但仍不够理想

由于 RISC 也存在某些不足和问题，使得在设计 CPU 时，有的机器向着采取将 RISC 和 CISC 概念和技术密切结合，互相取长补短的方向发展。如 Fairchild(仙童)公司的 CLIPPER 机就是这样一种典型的 32 位微处理机，在处理机内部有一块包括 CPU 与浮点运算单元的 CPU/ FPU。主 CPU 共有 101 条基本的硬联指令，其常用

指令的执行多数只需一个 30ns 时钟周期,设置了 32 个 32 位的通用寄存器,除存和取指令可以访存外,所有操作都在寄存器之间进行,指令处理采用了 3 级流水,所有这些都体现了 RISC 的特点。另外还包括 64 位浮点部件 FPU,有 8 个 64 位的浮点寄存器和独立的浮点运算部件 ALU,包括 67 条 CISC 型的宏指令,可识别 10 种数据类型,有 9 种寻址方式,14 种指令格式,有 16,32,48 位和 64 位共 4 种指令长度,这些都体现了 CISC 的特点。所以说,CLIPPER 机以提高整个计算机系统的性能为基点,在结构上吸收了 RISC 和 CISC 各自的优点,并相互融合在一起。

RISC 技术经过近二十几年的发展,已逐步完善和成熟。目前各大计算机公司都大力支持发展 RISC 系统,并且与 CISC 在结构上互相补充。1987 年,美国的 Phil Koopmen 综合平衡 RISC 和 CISC 概念,提出了 WISC(Witable Instruction Set Computer)结构设想,允许用户在 RISC 结构中,写入自己定义的复杂指令,这种针对用户环境而采用自己定义的复杂指令是效益最高的指令,可使整机性能价格比提高。

2.5 DLX 指令集结构

本节将讨论一种称为 DLX 的 Load/Store 型指令集结构。DLX 是一种多元未饱和型指令集结构,称之为多元未饱和型结构,是因为它不仅体现了当今多种机器(AMD29K, DEXstation 3100, HP850, IBM801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC-Ⅰ, SGI 4D/60, SPARCstation-1, Sun-4/110, Sun-4/260)指令集结构的共同特点,而且它还将会体现未来一些计算机的指令集结构的特点。这些机器的指令集结构设计思想都和 DLX 指令集结构的设计思想十分相似,它们都强调以下几点:具有简单的 Load/Store 指令集系统;设计上注重指令的流水效率;简化指令的译码;使编译器更容易产生高效的目标代码。

DLX 是一种适合于学习和研究的系统结构模型,下面就重点介绍其指令集结构和效能。

2.5.1 DLX 指令集结构

1.DLX 的寄存器

DLX 有 32 个 32 位通用寄存器(GPR),名称为 R0, R1, …, R31。寄存器 R0 的值总是为 0,稍后将会看到如何利用该寄存器由简单指令集来合成一组有用的操作。

另外,DLX 还有 32 个 32 位的浮点寄存器(FPR),名称为 F0, F1, …, F31。这些浮点寄存器可以用来保存 32 位的单精度浮点数,或者通过相邻两个浮点寄存器奇偶对 $F_i F_{i+1}$ ($i = 0, 2, 4, \cdots, 30$)来保存 64 位的双精度浮点数,这种组合而成的双精度浮点寄存器被命名为 F0, F2, …, F28, F30。

在 DLX 中还有一些特殊的寄存器,如用来保存浮点操作结果信息的浮点状态寄存器,其中有些用来和通用寄存器交换数据。DLX 也提供了在 FPR 和 GPR 之间传送数据的指令。

2 .DLX 的数据类型

DLX 提供了多种长度的整型数据和浮点数据。对整型数据而言,有 8 位、16 位和 32 位三种长度;对浮点数据而言,有 32 位单精度浮点数和 64 位双精度浮点数,遵循 IEEE 754 标准。DLX 操作是面向 32 位的整数以及 32 位或 64 位的浮点数的。字节或半字在调入 32 位的寄存器时,用零或者符号位来填充 32 位寄存器的剩余位。一旦被调入以后,它们将按照 32 位整数的方式进行计算。

3 . DLX 数据传输的寻址方式

数据寻址方式只有立即数寻址、寄存器寻址、偏移寻址和寄存器间接寻址四种方式。寄存器寻址字段的大小为 5 位,用来标识 32 个通用寄存器或浮点寄存器。

DLX 的存储器地址采用的是高端字节表示顺序,存储器按字节寻址,其地址宽度为 32 位。由于 DLX 是一种 Load/ Store 结构,所以它通过寄存器(通用寄存器和浮点寄存器)和存储器之间的数据传送操作完成对存储器的访问。

由于 DLX 支持上述数据类型,所以对通用寄存器而言,相应的存储器访问数据大小有 8 位、16 位和 32 位;而对浮点寄存器而言,相应的存储器访问数据大小有 32 位的单精度浮点数和 64 位的双精度浮点数,所有内存访问必须是对准的。

4 . DLX 指令格式

由于 DLX 只有四种寻址方式,所以它们可以编码到操作码中。为了使计算机更容易进行流水线操作和译码,所有指令都是固定的 32 位长,其中 6 位是基本操作码。图 2 .17 是 DLX 指令的格式,从图可以看出,这些指令格式比较简单,同时还为偏移寻址、立即数寻址或 PC 相对跳转地址提供了 16 位的域。

5 .DLX 操作

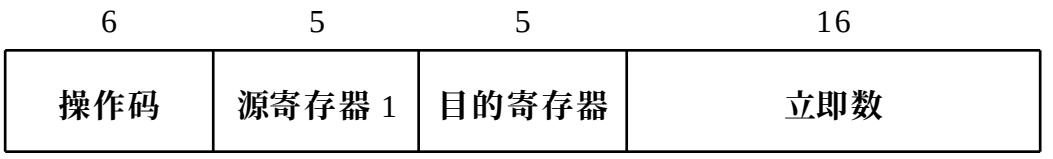
DLX 支持上面提到的一些简单操作,还有一些其他操作。DLX 指令大致可以分为四大类:加载存储、ALU 操作、分支与跳转以及浮点数操作。

在后面对上述各种操作类型进行论述的过程中,将采用 C 语言的扩充形式来表示指令的含义。所以这里首先介绍一些 C 语言扩充表示方法的约定。

① 符号“←”表示数据传送操作,其后附带一个下标 n,即“$\leftarrow_n$”表示传送一个 n 位数据。

② 符号“##”表示两个域的串联操作,可以出现在数据传送操作的任何一边。

I 型指令



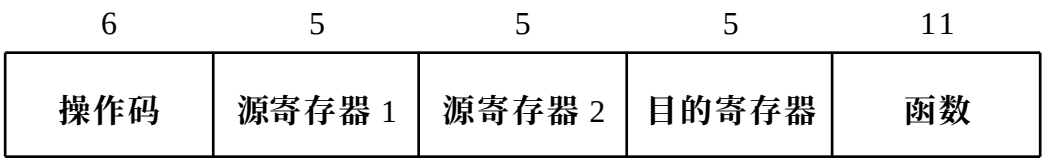
编码:对字节、半字、字等所有立即数的存/ 取

(rd←rsl op immediate)

条件转移指令(rsl 是寄存器,rd 没有使用)

跳转寄存器,跳转和链接寄存器(rd = 0, rs = 目的,immediate = 0)

R 型指令



寄存器—寄存器 ALU 操作:rd←rsl func rs2

函数操作: ADD, SUB, …

读/ 写特定寄存器和传送

J 型指令

跳转、跳转和链接

陷阱和意外返回

图 2 .17　DLX 的指令格式

③ 域的下标用来表明从该域中选择某一位。域中的位从最高位开始标记,并且起始标记为 0。下标可以是一个单独的数字,如 $Regs[R4]_0$ 表示选择寄存器 R4 中内容的符号位;下标也可以是一个范围,如 $Regs[R3]_{24..31}$ 表示选择寄存器 R3 中内容的最低一个字节。

④ 上标表示复制一个域,如 0^{24} 可以得到一个 24 位全为 0 的域。

⑤ 变量 Mem 用来表示存储器,存储器按照字节寻址,它可以传送任何数目的字节。

为了进一步说明上述约定表示方法的用途,现设 R8 和 R10 均为 32 位寄存器,那么,$Regs[R10]_{16..31} \leftarrow_{16} (Mem[Regs[R8]]_0)^8$ # # Mem[Regs[R8]]的含义是:以 R8 中的内容作为存储器地址寻址存储器相应单元的字节,将该字节的符号扩展形成

8 位的域,并和该字节的值串联形成一个 16 位的值,然后将该值传送到寄存器 R10,保存在寄存器 R10 的低 16 位,而寄存器 R10 的高 16 位保持不变。

DLX 中的四种操作类型:

(1)加载存储操作

DLX 中的所有通用寄存器与浮点寄存器都可作为加载 Load 和存储 Store 之用,惟一例外的是加载 R0。单精度浮点数占用一个单精度寄存器,而双精度浮点数占用一对,单精度与双精度之间的转换必须显式地进行。表 2 .13 给出了 DLX 的加载和存储指令的例子。

表 2 .13 DLX 的加载和存储指令

指令实例	指令名称	含 义
LW R1,30(R2)	加载整型字	Regs[R1]$\leftarrow_{32}$Mem[30 + Regs[R2]]
LW R1,1000(R0)	加载整型字	Regs[R1]$\leftarrow_{32}$Mem[1000 + 0]
LB R1,40(R3)	加载字节	Regs[R1]$\leftarrow_{32}$(Mem[40 + Regs[R3]]$_0$)^{24} # # Mem[40 + Regs[R3]]
LBU R1,40(R3)	加载无符号字节	Regs[R1]$\leftarrow_{32}0^{24}$ # # Mem[40 + Regs[R3]]
LH R1,40(R3)	加载整型半字	Regs[R1]$\leftarrow_{32}$(Mem[40 + Regs[R3]]$_0$)^{16} # # Mem[40 + Regs[R3]] # # Mem[41 + Regs[R3]]
LF F0,50(R3)	加载单精度浮点数	Regs[F0] $\leftarrow_{32}$Mem[50 + Regs[R3]]
LD F0,50(R2)	加载双精度浮点数	Regs[F0] # # Regs[F1] $\leftarrow_{64}$Mem[50 + Regs[R2]]
SW 500(R4),R3	存储整型字	Mem[500 + Regs[R4]] $\leftarrow_{32}$Regs[R3]
SF 40(R3),F0	存储单精度浮点数	Mem[40 + Regs[R3]] $\leftarrow_{32}$Regs[F0]
SD 40(R3),F0	存储双精度浮点数	Mem[40 + Regs[R3]] $\leftarrow_{32}$Regs[F0] Mem[44 + Regs[R3]] $\leftarrow_{32}$Regs[F1]
SH 502(R2),R31	存储整型半字	Mem[502 + Regs[R2]] $\leftarrow_{16}$Regs[R31]$_{16..31}$
SB 41(R3),R2	存储整型字节	Mem[41 + Regs[R3]] $\leftarrow_{8}$Regs[R2]$_{24..31}$

(2) ALU 操作

在 DLX 中,所有的 ALU 指令都是寄存器—寄存器型指令,包括简单的算术和

逻辑操作:加、减、与、或、异或和移位。所有这些指令都支持立即数寻址模式,立即数以 16 位符号扩展形式出现。LHI(加载高位立即数)操作将立即值加载到一个寄存器的高半部分,而该寄存器的低半部分设为零。这样就可以通过两条 Load 指令构造一个 32 位的常数。

正如上面所提到的,R0 主要用来合成通用操作。如加载一个常数就可以看做是一次简单的立即值加操作,其中一个源操作数是 R0;寄存器—寄存器间的数据传送可以通过其中一个源操作数是 R0 的加法来完成,这两个操作可以分别用 LI 和 MOV 表示。

还有比较两个寄存器的比较指令(= , ≠ , > , < , ≥ , ≤) ,如果比较结果为真,则比较指令将在目的寄存器中放入一个 1,否则填入 0。因为这些比较操作都设置目标寄存器,所以它们也被叫做 set-equal, set-not-equal, set-less-than 等。同时这些比较指令也具有立即数的形式。表 2 .14 给出了 ALU 指令的例子。

表 2 .14　　ALU 指令(带有或不带有立即数)

指令实例	指令名称	含　义
ADD　R1, R2, R3	加	Regs[R1]←Regs[R2] + Regs[R3]
ADDI　R1, R2, # 3	加立即数	Regs[R1]←Regs[R2] + 3
LHI　R1, # 42	加载立即数到高半字	Regs[R1]←42 # # 0^{16}
SLLI　R1, R2, # 5	逻辑左移立即数	Regs[R1]←Regs[R2] < < 5
SLT　R1, R2, R3	置小于	If(Regs[R2] < Regs[R3])Regs[R1]←1 Else Regs[R1]←0

(3)分支与跳转操作

在 DLX 中,对程序流程的控制是通过一组跳转与一组分支指令来实现的。根据目标地址的两种指定方式和是否进行链接来区分,可以将跳转操作分为四种指令类型。其中两种类型的跳转指令用带符号的 26 位偏移量加上程序计数器的值确定目的地址,另外两种类型的跳转指令则指定一个包含目的地址的寄存器确定。跳转有两种类型,一种是简单跳转,另一种是跳转并链接(用于过程调用)。后者将返回一个地址,即将下一条顺序指令地址(返回地址),放入 R31 中加以保存。

DLX 中的所有分支指令都是条件分支指令,其源操作数寄存器中包含了一个数值或某个比较结果。分支指令测试该源操作数寄存器中的值是 0 还是非 0,从而决定分支是否成功。分支目标地址由一个带符号的 26 位偏移量加上程序计数器的值来确定,分支目的地址指向下一条要执行的指令。表 2 .15 给出了一些典型的分支和

跳转指令。

表 2 .15 典型的分支和跳转指令

指令实例	指令名称	含 义
J name	跳转	PC←name; ((PC + 4) - 2^{25}) ≤name≤ ((PC + 4) + 2^{25})
JAL name	跳转并链接	Regs[31]←PC + 4;PC←name; ((PC + 4) - 2^{25}) ≤name≤ ((PC + 4) + 2^{25})
JALR R2	寄存器跳转并链接	Regs[R31]←PC + 4;PC←Regs[R2]
JR R3	寄存器跳转	PC←Regs[R3]
BEQZ R4,name	等于零时的分支	If(Regs[R4] = = 0)PC←name; ((PC + 4) - 2^{15}) ≤name≤ ((PC + 4) + 2^{15})
BNEZ R4,name	不等于零时的分支	If(Regs[R4] ! = 0)PC←name; ((PC + 4) - 2^{15}) ≤name≤ ((PC + 4) + 2^{15})

(4)浮点操作

在 DLX 中,浮点指令的操作数来源于浮点寄存器,同时该浮点指令还指明了相应的操作是单精度浮点操作还是双精度浮点操作。DLX 的浮点操作包括有:加、减、乘、除。后缀 D 代表双精度浮点操作,而后缀 F 代表单精度浮点操作(如 ADDD, ADDF, SUBD, SUBF, MULTD, MULTF, DIVD, DIVF)。值得提出的是,DLX 的浮点比较操作将设置浮点状态寄存器中的位,如果比较结果为真,则将该位设置为 1;如果比较结果为假,则将该位设置为 0。浮点分支指令 BFPT 和 BFTF 则测试该寄存器的值以决定分支是否成功。

另外,操作 MOVF 将一个单精度浮点寄存器的内容拷贝至另一个单精度浮点寄存器;MOVD 则将一个双精度浮点寄存器的内容拷贝至另一个双精度浮点寄存器;MOVFP2I 和 MOVI2FP 操作则是在一个浮点寄存器和通用寄存器之间移动数据。如果要将一个双精度浮点数移入两个通用寄存器则需要两条指令。另外,DLX 还提供了在 32 位浮点寄存器中进行整数乘除操作的指令。

表 2 .16 列出了 DLX 的所有指令及其含义。表中 SP 表示单精度,DP 表示双精度。

表 2.16 DLX 指令列表

指令类型/ 操作码	指 令 含 义
数据传输	在寄存器与内存之间传送数据,或者是在整数寄存器与浮点寄存器或特定寄存器之间传送数据;内存惟一的寻址模式是 16 位偏移 + GPR 的内容
LB, LBU, SB	加载字节,加载无符号字节,存储字节
LH, LHU, SH	加载半字,加载无符号半字,存储半字
LW, SW	加载字,加载无符号字,存储字
LF, LD, SF, SD	加载 SP 浮点数,加载 DP 浮点数,存储 SP 浮点数,存储 DP 浮点数
MOVI2S, MOVS2I	在 GPR 与特定寄存器之间传送数据
MOVF, MOVD	拷贝一个 FP 寄存器或一对 DP 寄存器内容对到另一个寄存器或寄存器对
MOVFP2I, MOVI2FP	在整数寄存器和 FP 寄存器之间传送 32 位数据
算术/ 逻辑	在 GPR 中的整数或逻辑数据的操作;有符号数在溢出时产生陷阱中断
ADD, ADDI, ADDU, ADDUI	加,加立即数(所有立即数都是 16 位),加符号数,加无符号数
SUB, SUBI, SUBU, SUBUI	减,减立即数,有符号减,无符号减
MULT, MULTU, DIV, DIVU	乘和除,有符号和无符号乘除,操作数必须是 FP 寄存器,所有的操作使用或产生 32 位的值
AND, ANDI	与,立即数与
OR, ORI, XOR, XORI	或,立即数或,异或,立即数异或
LHI	加载立即数到高半字
SLL, SRL, SRA, SLLI, SRLI, SRAI	移位:立即数形式和变量形式,逻辑左移,逻辑右移,算术右移
S_, S_I	设置条件,“_”可以是 LT, GT, LE, GE, EQ, NE
控制	条件分支或跳转,相对于 PC 或者通过寄存器指明跳转地址
BEQZ, BNEZ	GPR 为零或不为零跳转,相对于 PC + 4 的 16 位偏移
BFPT, BFPF	测试 FP 状态寄存器中的比较位并分支,相对于 PC + 4 的 16 位偏移
J, JR	跳转,相对于 PC + 4 的 26 位偏移(J)或者由寄存器指定(JR)
JAL, JALR	跳转并链接,把 PC + 4 保存在 R31,目标地址为相对于 PC(JAL)或是寄存器指定(JALR)
TRAP	将一个向量地址传输给操作系统
RFE	从例外返回用户代码,恢复用户态

续表

指令类型/ 操作码	指 令 含 义
浮点操作	DP 和 SP 格式的 FP 操作
ADDD,ADDF	加 DP,SP 浮点数
SUBD,SUBF	减 DP,SP 浮点数
MULTD,MULTF	乘 DP,SP 浮点数
DIVD,DIVF	除 DP,SP 浮点数
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F,CVTI2D	转换指令,CVTx2y 把 x 转换成 y,其中 x 和 y 的含义是 I(整数),D(双精度数),F(单精度数),所有操作数都是 FPR
- D, - F	DP 和 SP 比较,“ - ”= LT,GT,LE,GE,EQ,NE,并设置在 FP 状态寄存器中的状态

2.5.2 DLX 指令集结构效能分析

综上所述,DLX 指令集结构的指令格式、寻址方式和操作都非常简单。这些特性看起来可能会使目标代码中指令条数增多,导致程序运算时间加长,从而使这种指令集结构的机器性能不会太高,但实际情况并非如此。第一章的 CPU 性能公式提示,程序的执行时间并不仅仅是一个指令条数的函数,它还与每条指令所需的平均时钟周期数以及时钟周期这两个因素密切相关,因此不能从指令条数来考虑指令集结构的性能。

那么,为了观察指令条数的减少能否被 CPI 以及时钟周期长度的增加所抵消,需要把 DLX 与一个复杂的系统结构比较一下,这里选择出现于 20 世纪 70 年代中期的 VAX 指令集结构作为参考结构。之所以选择 VAX,是因为 VAX 的设计思想和 DLX 的设计思想截然不同,VAX 中提供了多种指令格式、大量的寻址方式,所有的寻址方式均可以适用于各种类型的指令操作,并且指令的操作也比较复杂。

VAX 采用这种设计思想的重要目标之一是希望尽可能地缩短目标代码,减少存储空间的浪费,这和当时的时代背景是一致的。在 20 世纪 70 年代中期,在设计 VAX 的时候,占主导地位的思想是创建一种与编程语言相近的指令集,从而简化编译器。另一种盛行的设计思想是使代码的长度最小化。大家知道,DRAM 的容量每 3 年提高 4 倍,因此 70 年代中期的 DRAM 芯片的容量只有当今 DRAM 芯片容量的 1/ 1000,所以对代码空间的要求非常严格。

VAX 机器的设计者后来对 VAX 和类 DLX 型计算机进行了一次定量的比较,因为它们有可比的结构。他们选择了 VAX8700 和 MIPS M2000。VAX 与 MIPS 的不同目标导致了它们完全不同的系统结构。VAX 的目标是简化编译器和减少执行代码的长度,因此它有很强的寻址模式、很强的指令、高效的指令编码以及较少的寄

存器。MIPS 的目标是通过流水线获得高性能,易于用硬件实现,以及与高度优化编译器的兼容性。这些目标导致了简单的指令、简单的寻址模式、固定长度的指令格式和大量的寄存器。图 2 .18 是这两种机器运行 SPEC89 基准程序测试结果的比较,它表明了两种机器执行的指令条数的比值、CPI 的比值和以时钟周期为单位的性能比值。

因为 VAX8700 和 MIPS M2000 的组织结构类似,所以假设它们的时钟周期时间一样。MIPS M2000 执行的指令数大约为 VAX8700 的 2 倍,而 VAX8700 的 CPI 将近是 MIPS M2000 的 6 倍,所以 MIPS M2000 在性能上将近是 VAX8700 的 3 倍。另外,由于 MIPS 结构十分简单,所以实现 MIPS 的 CPU 比实现 VAX 的 CPU 所需要的硬件要少得多。

也正是由于这种性能价格比上的差异使得过去研制 VAX 机器的公司现在已抛弃了 VAX 指令集结构,而采用了和 DLX 类似的指令集结构。

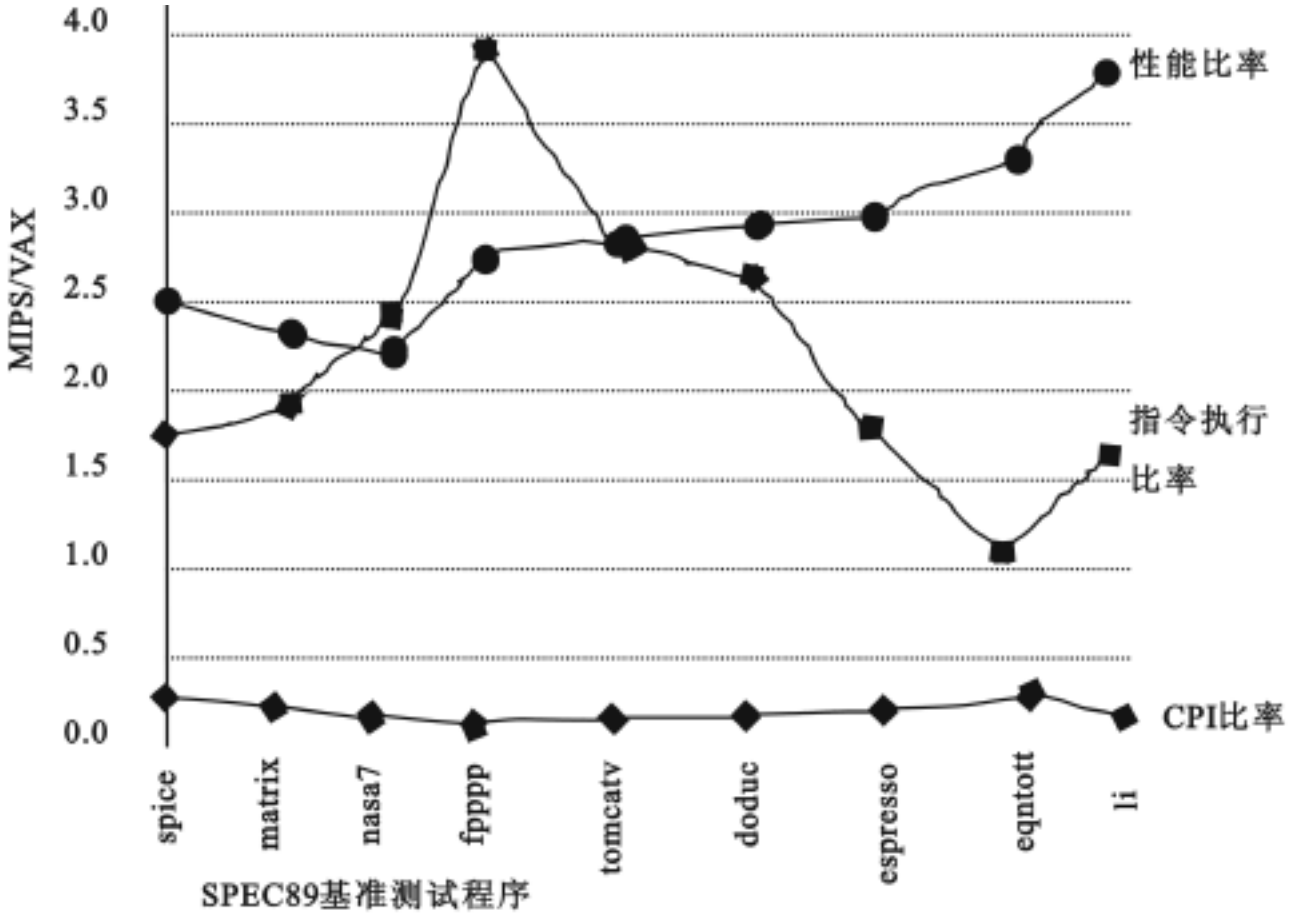


图 2 .18 VAX8700 和 MIPS M2000 的比较结果

习 题

1 .解释下列术语:

数据表示 寻址方式 逻辑地址 物理地址

静态再定位　　动态再定位　　静态使用频度　动态使用频度

CISC　　RISC

2 .数据结构和机器的数据表示之间是什么关系？确定和引入数据表示的基本原则是什么？

3 .设某机阶码 6 位、尾数 48 位,阶符和数符不在其内,当尾数分别以 2,8,16 为基值时,在非负阶、正尾数、规格化数情况下,求出其最小阶、最大阶、阶的个数、最小尾数值、最大尾数值、可表示的最小值和最大值及可表示的规格化数的总个数。

4 .请证明:在浮点数的字长和表数精度一定时,尾数基值 r_m 取 2 或 4 时,浮点数具有最大的表数范围。

5 .一台计算机系统要求浮点数的精度不低于 10^{-7},表数范围正数不小于 10^{38},且正、负数对称。尾数用原码、纯小数表示,阶码用移码、整数表示。

(1)设计这种浮点数的格式;

(2)计算(1)所设计浮点数格式实际上能够表示的最大正数、最大负数、表数精度和表数效率。

6 .将下列的 IEEE 单精度浮点数由十六进制转换成十进制:

(1)42E48000H　　(2)3F880000H

(3)00800000H　　(4)C7F00000H

7 .将下列的 IEEE 单精度数由二进制数转换成十进制数。

(1)1　1000　0011　11　0000　0000　0000　0000　0000　0

(2)0　0111　1110　10　1000　0000　0000　0000　0000　0

(3)0　1000　0000　00　0000　0000　0000　0000　0000　0

8 .将下列数转换成 IEEE 单精度浮点格式,以 8 位十六进制数表示。

(1)9　　(2)5/32　　(3) -5/32　　(4)6 .125

9 .标志符数据表示与描述符数据表示有何区别？描述符数据表示与向量数据表示对向量数据结构所提供的支持有什么不同？

10 .请用三重描述符描述 2×3×4 三维阵列 A,要求按树形连续表示。

11 .若要计算 $c_i = a_{i+8} + b_{i-8}$, $i = 2,3,\cdots,9$ 的向量加法,假设 A,B,C 三向量的基地址分别为十六进制的 1000H,2000H 和 3000H,它们的向量长度分别为 2,18 和 10,每个向量元素均为 32 位长。用图画出在进行上述操作时它们的起始地址、位移量以及有效向量长度。

12 .变址寻址和基址寻址各适用于哪些场合？设计一种只用 6 位地址码就可以指向一个大地址空间中任意 64 个地址之一的寻址机构。

13 .设计如 IBM370 那样有基地址寄存器的机器的另一种方法是,每条指令不用现在的基地址寄存器(4 位)加位移量(12 位)共 16 位作为地址,而是让每条指令都有一个 24 位的直接地址。针对下面两种情况评价一下这个方法的优、缺点:

(1)数据集中于有限几块,但这些块分布在整个存储空间;

(2)数据均匀地分布在整个地址空间。

你认为 IBM370 的设计者在实际应用中考虑这两种情况的哪一种可能性大？为什么？

14 .某模型机有 8 条指令,使用频度分别为:0 .3,0 .3,0 .2,0 .1,0 .05,0 .02,0 .02,0 .01,试分别用哈夫曼编码和扩展编码对其操作码进行编码,限定扩展编码只能有两种长度,则它们的平均编码长度各比定长操作码的平均编码长度减少多少？信息冗余量是多少？

15 .某模型机有 9 条指令,其使用频度为:

ADD(加)	30%	SUB(减)	24%
JOM(按负转移)	6%	STO(存)	7%
JMP(转移)	7%	SHR(右移)	2%
CIL(循环左移)	3%	CLA(清加)	20%
STP(停机)	1%		

要求有两种指令字长,都按双操作数指令格式编排,采用扩展操作码,并限制只能有两种操作码码长。设该机有若干个通用寄存器,主存为 16 位,按字节编址,采用整数边界存储,任何指令都在一个主存周期中取得,短指令为寄存器—寄存器型,长指令为寄存器—主存型,主存地址应能变址寻址。

(1)仅根据使用频度,不考虑其他要求,设计出全哈夫曼操作码,并计算出该操作码方式的平均码长。

(2)考虑题目其他全部要求,设计优化的实用指令操作码形式,并计算其操作码的平均码长。

(3)该机允许使用多少可编址的通用寄存器？

(4)画出该机两种指令字格式,标出各字段的位数。

(5)指出访存操作数地址寻址的最大相对位移量为多少个字节？

16 .某处理机的指令字长为 16 位,有双地址指令、单地址指令和零地址指令三类,并假设每个地址字段的长度均为 6 位。

(1)如果双地址指令有 15 条,单地址指令和零地址指令的条数基本相同,问单地址指令和零地址指令各有多少条？并且为这三类指令分配操作码。

(2)如果要求三类指令的比例大致为 1 :9 :9,问双地址指令、单地址指令和零地址指令各有多少条？并且为这三类指令分配操作码。

17 .试叙述 RISC 计算机中所采用的窗口重叠寄存器的工作原理,并列举它的主要优缺点。

18 .为什么不少计算机采用由 8 个或更多个寄存器构成的通用寄存器组？它们为什么会使编译生成更复杂？你能设想什么技术的发展会使得不必采用通用寄存器组？

19 .设计 RISC 计算机的一般原则及可采用的基本技术有哪些？

20 .简要比较 CISC 计算机和 RISC 计算机各自的结构特点,它们分别存在哪些不足和问题?为什么说今后的发展应是 CISC 和 RISC 的结合?

21 .DLX 是一个实验性的 RISC 计算机,它的指令系统中各类指令的平均 CPI 值如下:寄存器—寄存器型指令为 1 个时钟周期;取/ 存指令为 1 .4 个时钟周期;条件转移指令当转移发生或转移不发生时,分别为 2 个或 1 .5 个时钟周期;无条件转移指令为 1 .2 个时钟周期。假定有 60%的条件转移指令为转移发生,且若在某个典型测试程序中各类指令所占的比例为: ALU 指令 46%,取/ 存指令 37%,条件转移指令 16%,无条件转移指令 1%,试计算在执行该测试程序时,其平均的 CPI 值为多少?

第三章 输入输出系统

输入输出系统是计算机系统的一个重要组成部分,是计算机与外界交互的接口,其设计的好坏直接影响着计算机系统的整体性能和效率,本章着重讨论总线、中断、输入输出系统的设计及软硬件功能分配。

3.1 输入输出系统原理

在计算机系统中,通常把处理机和主存储器之外的部分统称为输入输出系统,它包括输入输出设备、输入输出接口和输入输出软件等。

输入输出系统的主要功能就是完成主机(处理机和主存储器)与外部系统的信息交换,是 Von Neumann 系统结构的四大组成部分之一。但长期以来,在计算机系统结构设计中对输入输出部分都不太重视,主要是因为通常利用 CPU 时间来评价计算机系统的性能,而输入输出系统的优劣,不能直接通过 CPU 时间体现出来,况且输入输出设备又称为"外部设备"。

实际上,输入输出系统的性能无论对系统的性能还是对系统的效率都有非常大的影响,这可以从以下几方面看到:

(1)不同系统中外部设备的差异

外部设备是计算机系统的一个重要组成部分。外部设备的控制能力和数量影响到计算机系统的整体功能,甚至是区别计算机系统的标志。如大型计算机和小型计算机的一个重要区别就在于大型机能够控制大量外部设备,工作站和小型机的主要区别在于工作站具有图形终端和图形处理功能,工作站和服务器的主要区别在于服务器的外存管理能力等。

(2)输入输出系统的设计问题

要设计一个好的输入输出系统,要对系统中连接的外部设备的种类和连接方式进行分析,包括性能、成本、容量等;要设法避免输入输出系统成为整个系统的性能瓶颈。

(3)对计算机整体性能的影响

由 Amdahl 定律可知,对系统性能影响最大的是执行时间最慢的那部分,即系统瓶颈。CPU 的速度提高得是很快的,按目前的状况,每 18 个月就提高一倍,如果输入输出部分不加以改进,系统的性能将受限于输入输出系统,再快的 CPU 都将失去

其意义。

例 3.1　假设一台计算机的输入输出处理占总执行时间的 10%,CPU 处理占总执行时间的 90%,当其输入输出性能保持不变,而在 CPU 性能改进的情况下,系统总体性能会提高多少?

① 如果 CPU 处理速度提高 10 倍;

② 如果 CPU 处理速度提高 100 倍。

解:根据 Amdahl 定律可知:

$$S_{(1)} = \frac{1}{1 - 0.9 + \frac{0.9}{10}} = \frac{1}{0.1 + 0.09} \approx 5$$

$$S_{(2)} = \frac{1}{1 - 0.9 + \frac{0.9}{100}} = \frac{1}{0.1 + 0.009} \approx 10$$

从这个例子可知,如果不对输入输出加以改进,则当 CPU 性能提高 10 倍时,系统总的性能仅提高 5 倍,CPU 性能提高 100 倍时,系统总的性能只提高了 10 倍,表示有差不多 50%和 90%的性能浪费在没有改进的输入输出系统上了。那么,如何提高输入输出系统的性能呢?首先必须了解输入输出系统的特点和其组织方式,这正是本节要讲的主要内容。

3.1.1　输入输出系统的特点

随着计算机系统的不断发展和应用领域的进一步扩大,要求输入输出的数据量在迅速增大,对数据传送的速度要求在明显地增长,外部设备的种类和数量也在日益增多,而且,现代的外部设备的品种繁多,性能、结构差异很大,它们输入输出的方式又不一样,面对这些复杂的外部设备、不同的输入输出方式和人们对输入输出的要求不尽相同,只有找出输入输出的共同特点,才能更好地分析和设计输入输出系统。

输入输出系统的特点归纳起来主要是三个方面:异步性、实时性和独立性。

1.异步性

外部设备相对于处理机通常是异步工作的。外部设备由于品种、功能、结构、工作原理等诸方面不同,速度差异是很大的,而且外部设备的工作速度和 CPU 的相差更大。如键盘操作速度取决于人的手指按键的速度,按键速度又因人而异;不同类型的打印机通常按照自身的速度,每分钟打印一定行数的字符数;高速的磁盘机,也因其输入输出的速度受电机转速限制,与 CPU 的速度相比仍相差很远。这些都说明外部设备的操作在很大程度上独立于 CPU,不能使用统一的工作节拍,各个设备按照自己的时钟工作。但另一方面,外部设备要与处理机交换数据,什么时候准备好数据,什么时候请求传送,对 CPU 来说是随机的,为了能使处理机和外部设备充分提高工作效率,保证处理机与外部设备之间、外部设备与外部设备之间能够并行工作,

要求输入输出操作异步于 CPU,把它们之间的相互牵制降低到最低限度。

2 .实时性

外部设备和处理机进行信息交换时,由于设备类型不同,信息传输的速率差异很大,传送方式也不一样,有的一次只传送一个字符,如打印机和键盘,有的按数据块或文件为单位传送,如磁盘和磁带等,这就要求处理机必须能按不同设备的传输速率、传送方式及时为设备服务,否则信息就可能丢失或造成外部设备工作错误。

实时性的要求在计算机控制领域更重要,如用于现场测试或控制的场合,信号的出现是即时的,若不及时接收和处理,就有丢失的危险,有的甚至造成巨大的损失。

对于处理机本身在运行时发生的硬件或软件错误,如电源故障、页面失效、溢出等,处理机也必须及时地给予处理。

为了能够为各种不同类型的设备提供服务,处理机必须具有与各种设备相配合的多种工作方式,这就是输入输出系统的实时性要求。

3 .独立性

各种外部设备发送和接收信息的方法不同,数据格式及物理参数差异也较大,而处理机与它们之间的控制和状态信号是有限的,接收和发送数据的格式是固定的,处理机的输入输出不可能针对某一个具体的设备来设计,应该有统一的规则,所以规定了一些独立于具体设备的标准接口。这样,输入输出与具体设备无关,具有独立性,只有这样,才能摆脱各种设备不同的要求。

各种外部设备必须根据自己的特点和要求,选择一种标准接口与处理机进行连接,设备之间的差异由设备本身的控制器通过硬件和软件来进行填补,具体的输入输出处理和设备调度由操作系统来分配和进行。这样,用户的高级语言源程序中出现的读写输入输出语句到读写操作全部完成,需要通过编译程序、操作系统软件和输入输出总线、设备控制器、设备等硬件来共同完成。输入输出系统硬件的功能对用户来讲是透明的,输入输出的功能只反映在高级语言和操作系统界面上。操作系统的工作主要是根据高级语言、控制语言的输入输出语句要求形成相应的输入输出机器指令,安排好让输入输出操作与 CPU 操作的并行执行,分配好输入输出操作所要用到的主存空间,对输入输出系统发出的控制信息进行处理,进行设备和文件管理,给应用程序员提供方便、简单的进行输入输出的使用界面等。所以,大多数计算机输入输出系统的设计应是面向操作系统,考虑怎样在操作系统与输入输出系统之间进行合理的软、硬件功能分配。

输入输出系统的这三个特点是现代计算机系统必须具备的共同特性。根据各种外部设备的不同特点要处理好这三方面的关系,这是输入输出系统组织的基本内容。一般针对异步性,采用自治控制的方法;针对实时性,采用层次结构的方法;针对独立性,采用分类处理的方法。

3.1.2 输入输出系统的基本方式

处理机与外部设备之间的信息交换应随外部设备性质的不同而采用不同的控制方式,输入输出系统的发展经历了三个阶段,对应于三种不同的控制方式:程序控制方式、直接存储器访问方式和 I/O 处理机方式。

1.程序控制方式

程序控制方式分为两种:程序查询方式和程序中断控制方式。程序查询方式是通过 CPU 对寄存器设标志决定 I/O 设备要执行的操作,由 CPU 执行驱动程序,启动外设,周期性地测试外部设备的状态位,以确定是否可以进行下一个 I/O 操作的简单接口方式。这种方式不需专门的硬件,简单的计算机系统都采用这种方式,但由于 CPU 速度比 I/O 设备速度快很多,查询方式会浪费许多 CPU 时间,使系统的性能大大降低,这种方式已很少采用。

为解决这个问题,导致了程序中断控制方式的出现。

中断驱动 I/O 已被许多系统所采纳,这种方式允许 CPU 在等待 I/O 设备时处理其他事务,在需要为 I/O 服务时,才中断 CPU 处理的事务,转去进行相应的输入输出操作。在一般的应用中,中断驱动 I/O 是实现多任务操作系统和获得快速响应时间的关键技术。

程序中断控制方式虽不像在查询方式下那样被一台外设独占,它可以同时与多台设备进行数据传送,但这种方式仍属于程序控制的输入输出方式,因在信息传送阶段,CPU 仍要执行一段程序控制,CPU 还没有完全摆脱对输入输出操作的具体管理,而且操作系统花费在每次中断事件上的代价太高。对于实时应用来讲,每秒有上百个 I/O 事件,这种代价是无法容忍的。对实时系统的解决方案是利用时钟周期性地中断 CPU,CPU 此时查询每一个 I/O 事件。

2.直接存储器访问(DMA)方式

程序中断控制方式把 CPU 从等待每个 I/O 事件中解脱出来,使设备和 CPU 在一定程度上并行工作。但是,这种方式下,仍有许多 CPU 周期花费在数据传输上,特别是对于那些配置有高速外部设备,如磁盘、磁带的计算机系统,这将使 CPU 处于频繁的中断工作状态,影响全机的效率。且像磁盘这样的外部设备,一般都是进行成块的数据传输,现在许多计算机系统都配置了直接存储器访问(DMA)硬件,使得数据的传输不再需要 CPU 的介入。

DMA 方式是一种完全由硬件进行成组信息传送的控制方式。它在 CPU 做其他工作的同时,可以在内存和 I/O 设备之间传输数据,数据的传输不再经过 CPU,传输速率也很高。DMA 作为 CPU 外部的一种特殊处理器,必须是总线的主设备。进行 DMA 传输前,CPU 首先设置 DMA 寄存器,包括主存地址以及要传输的字节数,

一旦DMA完成传输操作或在传输过程中发生错误,DMA控制器就向CPU发中断,进行传送结束时的处理或故障中断。

在一个计算机系统中,可以设置多个DMA设备,DMA控制器通常是I/O设备控制器的一部分。

3.I/O处理机方式

为了进一步减轻CPU的负担,可以增加DMA设备的智能性,采用比DMA功能更强的I/O处理机。

I/O处理机方式又可以有两种不同的形式,一种是以IBM公司为代表的通道(Channel)方式,另一种是以CDC和Burroughs公司为代表的外围处理机(Peripheral Processor Unit,PPU)方式。

在通道方式中,通道实际上可以看做是一台处理机,它有自己简单的指令系统(通道指令)和程序(通道程序),通道通过执行通道指令对外部设备进行控制,执行通道程序来完成输入输出。它还可以实现对外部设备的统一管理,并在主存和外设交换信息的过程中实现字与字节之间的装配和拆卸,能对输入输出系统出现的各种情况进行处理。这与DMA方式相比,大大提高了CPU的工作效率。

然而通道指令功能简单,通道程序又是存在和CPU公用的主存内,通道内部也只有用于数据缓冲的小容量存储器,所以,通道还不能看做是一种独立的处理机。

外围处理机是一种独立性、通用性和功能都较强的处理机,它是通道处理机的进一步发展。由于PPU基本上独立于主机工作,它的结构更接近一般处理机,甚至就可以是微小型计算机。在一些系统中,设置了多台PPU,分别承担I/O控制、通信、维护诊断等任务,从某种意义上说,这种系统已变成分布式的多处理机系统。

综上所述,程序查询方式和程序中断方式适用于数据传输率比较低的外部设备,而DMA方式、通道方式和PPU方式适用于数据传输率比较高的外部设备。目前,单片机和微型机中多采用程序查询方式、程序中断方式和DMA方式,通道方式和PPU方式大都用在中、大型计算机中。

本章主要介绍通道方式的工作原理和通道的流量分析。

由于通道总线、输入输出总线是连接处理机和外部设备的纽带,输入输出过程中的很多操作又离不开中断系统,所以这章首先讨论总线的设计和中断系统,然后再重点讨论通道方式的基本原理和流量设计,最后介绍外围处理机的原理和工作方式。

3.2 总线设计

总线技术是计算机系统的一个重要技术,它是计算机中各个子系统连接的纽带。输入输出总线既要能传送各类不同的信息,还要使不同的外部设备与CPU或主存交叉地经这些总线传送信息,所以其设计的好坏对I/O系统的性能会有较大的影

响。本节就从总线的类型、控制方式、通信技术等方面讨论总线设计的几种可供选择的方案,并简单介绍 Pentium 微处理器的总线系统,Pentium 系列的输入输出总线实例(USB 和 IEEE1394)。

3.2.1 总线的类型

总线是构成计算机系统的互连机构,是多个系统功能部件之间进行数据传送的公共通路,通过总线将 CPU、主存储器和输入输出设备连接起来。现代的一个单处理器系统中的总线,大致分为三类:第一类是内部总线,指的是 CPU 内部连接各寄存器及运算部件之间的总线,又称芯片级总线。第二类是系统总线,它是 CPU 同计算机系统的其他高速功能部件,如存储器、通道等互相连接的总线。第三类就是 I/O 总线,包括中低速 I/O 设备之间互相连接的总线,应遵循总线标准。

按数据线的宽度划分,总线一般可分为 8 位、16 位、32 位、64 位、128 位等总线。总线的宽度对总线的实现成本、可靠性、数据传输速率等方面的影响都非常大。总线的宽度越宽,其性能就越高,但是总线需要有发送电路、接收电路、传输导线或电缆、插接头等,这部分比逻辑线路的成本高得多,而且转接器往往占系统物理空间的很大比例,是降低系统可靠性的主要部分。所以说,总线的线数越多,成本越高,干扰越大、可靠性越低,但总线的宽度越宽,传输速度和流量也越大。一般来说,越是比较长的总线,信号畸变的可能性增大,干扰也增大,因此,越是长的总线,其线数应尽可能减少。如采用一位、一个字或一个全字等。总线设计的目标就是用较少的线数实现较高的传输速率。

从允许数据的传送方向来划分,总线可以有单向传输和双向传输两种。其中双向传输又有半双向和全双向的不同。半双向总线在同一时刻信息只能向其中的一个方向传送,全双向允许同时在两个方向传送,全双向的速度快,相应的造价也高,结构也比较复杂。

总线按其使用方法不同可以分成专用的和非专用的两种。

只实现一对物理部件之间的连接的总线,称为专用总线。它的数据流量高,多个部件可以同时工作,不会出现总线冲突,控制简单。任何总线的失效都只影响到该总线连接的两个部件,而这两个部件还可以通过和其他部件间的连接间接通信,提高了系统的可靠性,但这种总线需要的总线数目很大。如果有 N 个部件需要用专用总线在所有可能的路径上都相互连接的话,就需要用 $N \cdot (N-1)/2$ 组总线。当 N 很大时,总线数将几乎与部件数成平方倍关系增加,不仅增加了转接头,难以小型化和集成电路化,且成本高,所以一般用在只需要单一连接的场合,如某个外部设备和另外一个外部设备之间的直接连接。

非专用总线可以被多个部件共享,但同一时间,只允许一对部件使用,所以是一种分时共享总线。这种公用总线按其具体连接的方式或结构不同可分成单总线结构、双总线结构、三总线结构、开关矩阵联结非专用总线。

1 .单总线结构

在许多单处理器的计算机中,使用一条单一的系统总线来连接 CPU、主存和 I/O 设备,通过这条总线既可以访问主存储器,又可以进行 I/O 数据的传输,所以称为单总线结构,如图 3 .1 所示。

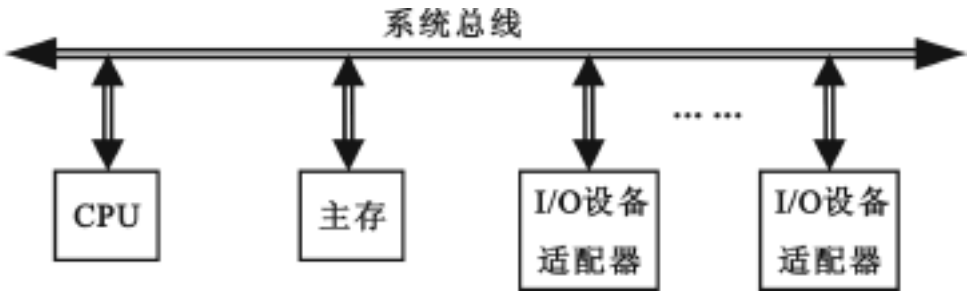


图 3 .1 单总线结构

在这种单总线结构中,对输入输出设备的操作,完全是用和访问主存一样的方法来处理,即主存和 I/O 设备统一编址,所有的主存单元及外部设备接口寄存器的地址一起构成一个连续的地址空间,访问主存或者外部设备只需由地址值来区别,CPU 可以像访问主存一样来访问外部设备。

单总线结构还可以在任意两台外部设备之间交换信息。某些外部设备也可以指定地址,如果外部设备获得总线的使用权,就可向总线发送地址,使总线上的地址线置为适当的代码状态,以便指定它将要与哪一个设备进行信息交换,如果此时指定的地址对应于一个主存单元,则主存予以响应,于是在主存和外设之间将进行 DMA 传输。

单总线结构简单,使用灵活,易扩充,可以很方便地扩展成多 CPU 系统,这只要在系统总线上挂接多个 CPU 即可。但在这种方式下,要求连接到总线上的逻辑部件必须高速运行,以便在某些设备需要使用总线时,能迅速获得总线控制权,而当不再使用总线时,能迅速放弃总线控制权,否则,时间延迟会很长。此外,由于所有的部件均通过一条总线进行通信,分时使用总线,因此,通信速度比较慢,流量和总线长度也有较大的矛盾。为解决这个问题,有的系统也采用多重单总线结构,一方面,可以用于并行传送来提高信息传送流量,另一方面也可用来防止单点失效,以提高可靠性。

通常,这种单总线结构适用做小型或微型计算机的系统总线。

2 .双总线结构

在这种结构中有两条总线:存储总线和系统总线,如图 3 .2 所示。这种总线和单总线相比,增加了 CPU 和主存之间的一组高速的存储总线。

存储总线的设置可使 CPU 通过专用总线与主存交换数据,解决了高速主存访

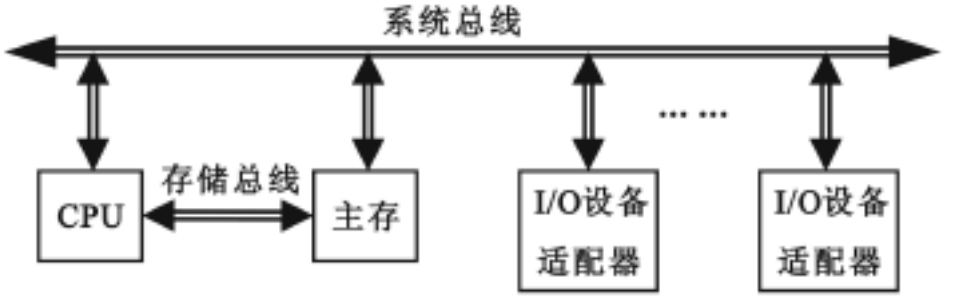


图 3 .2　双总线结构

问和慢速 I/ O 之间的差别,并减轻了系统总线的负担,同时主存仍可通过系统总线与外部设备之间实现 DMA 操作,而不必经过 CPU,当然,这和单总线相比,是以增加硬件为代价的。

3 .三总线结构

三总线结构是在双总线结构的基础上增加了 I/ O 总线形成的,如图 3 .3 所示。其中系统总线是 CPU、主存和外围处理机(IOP)或通道之间进行数据传送的公共通路,I/ O 总线是负责多个外部设备与通道之间进行数据传送的公共通路。

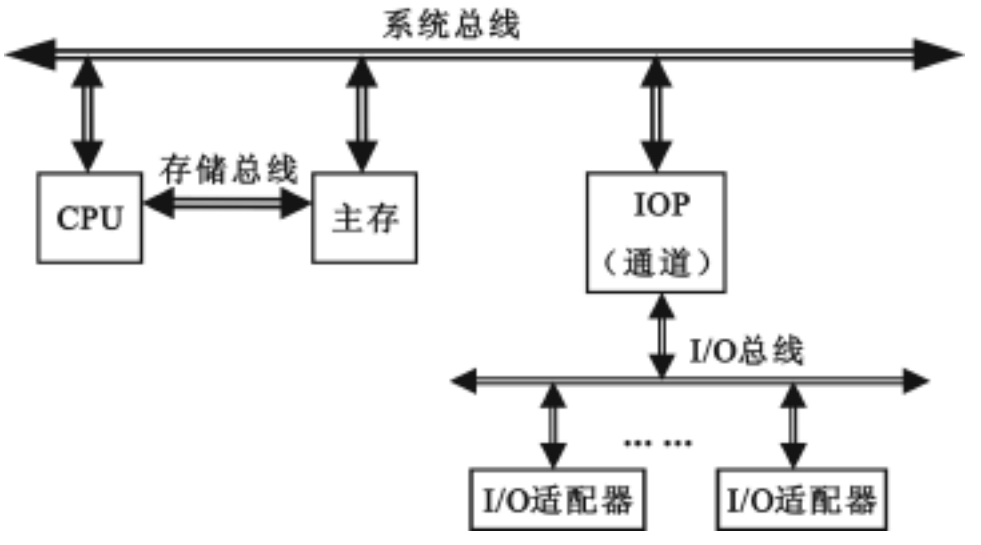


图 3 .3　三总线结构

由前面叙述可知,在 DMA 方式中,外部设备与存储器间直接交换数据而不必经过 CPU,从而减轻了 CPU 对数据输入输出的控制,而在三总线结构中,“ 通道 ”方式进一步提高了 CPU 的效率,由于增加了通道处理机,使整个系统的效率大大提高,然而这是以增加更多的硬件代价换来的。

4 .开关矩阵联结非专用总线

当各部件之间需要频繁的同时联系时,可采用开关矩阵式的联结总线,如图 3 .4

所示,它也是非专用总线的一种结构形式。

每个纵横交点处都设置有一个开关,任何时刻,只允许每行每列中一组开关可以接通,这说明,如果 K = min{M, N},则所有行中任何 K 个相异的部件,可以同时联结到所有列中对应的 K 个相异的部件,即在这种总线结构中,最多允许有 K 对部件同时进行信息传送。这实际上是一种多组非专用总线的例子,它可以减少争用总线的现象,提高系统的传输能力。但这种开关的控制是很复杂的,且开关的数目随联结的部件数近似成平方倍关系增加,同时一组开关的失效会使相应部件之间没有另外的通路。

这种非专用总线结构仅用于分布机或多处理机之间的通信,而上面所述的三种非专用总线结构主要是面向单机系统的。现有的大多数总线都是面向单机的总线,在这种总线中,访存操作是总线的主要操作,每次操作一般是传送一个数据,要求总线裁决的速率快。面向多机的总线则以传输数据块为单位,所以要求总线的数据传输速率要高,对总线裁决速度的要求则相对较低。由于每个数据块只需给出起始地址,因此一般地址线和数据线是分时复用的,如 Multibus I 和 Multibus Ⅱ等。

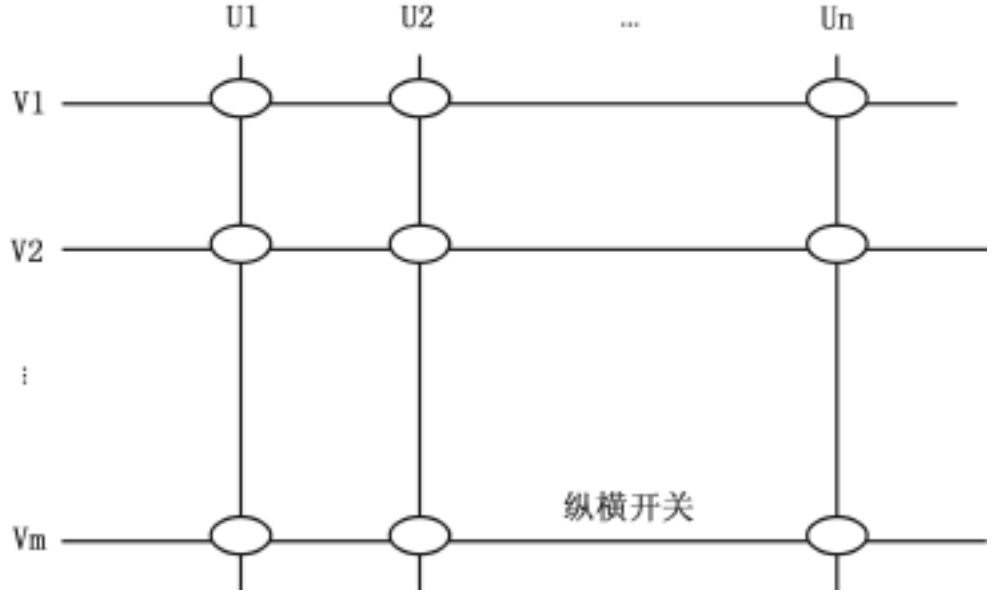


图 3.4 两组不同类型部件的纵横开关连接

非专用总线可降低成本,有较大的扩展性和结构灵活性,但系统流量小,经常出现争用总线的现象,如果处理不当,总线有可能成为系统速度性能的瓶颈,单总线结构尤其如此。其次,对共享总线的失效特别敏感,它往往会导致整个系统不能正常运行。一般为了增强可靠性,提高流量,可设置多重总线结构。无论采用哪种结构形式,对非专用总线而言,都要解决好总线争用的问题。

I/O 系统中大多数采用非专用总线。

3.2.2 总线的控制方式

在非专用总线上,连接到总线上的功能模块有主动(主方)和被动(从方)两种形态。主方可以申请使用总线,从方只能响应主方的请求。每次的总线操作,只能有一个主方占用总线控制权,但同一时间里可以有一个或更多个从方回应主方。

连接到总线上的模块,除 CPU 模块外,还有一些其他功能模块也可以提出总线请求,如某些 I/O 功能模块。为解决同一个时刻多个主设备同时竞争总线控制权的问题,必须具有总线仲裁机构,以选择一个主设备作为总线的下一次主方。

总线的控制即为总线仲裁机构应完成的功能,故又称总线控制器。总线的控制方式主要有两种:集中式控制和分布式控制。集中式控制方式将总线控制逻辑集中在一起,如设置一个单独的总线控制器或者将它放在 CPU 中,都称为集中式控制。而当总线控制逻辑分散地放在各个连接到总线的部件中去时,就为分布式总线控制,如在分布式多处理机系统和局部网络中都可以采用这种方式,在这里主要讨论集中控制方式。

目前,系统总线多采用集中式控制的方式解决总线使用权的问题。总线控制器对总线分配优先次序的确定可以有三种方式:串行链接、定时查询和独立请求。

1.串行链接方式

在串行链接方式下,总线使用权的分配通过三根控制线来实现:总线可用、总线请求和总线忙信号线,如图 3.5 所示。所有的功能部件经过一条公共的总线请求信号线向总线控制器发出要求使用总线的请求,控制器收到总线申请后,首先检查总线忙信号线,只有当总线处于空闲状态时,总线请求才能被总线控制器响应,此时,送出总线可用的回答信号,该信号串行地通过每个部件。未发出总线请求的部件在接收到总线可用信号时将其传送给下一个功能部件;发出请求的部件在收到总线可用信号后就停止传送该信号,并开始建立总线忙信号,并去除总线请求信号,开始总线操作。在数据传送期间,总线忙信号维持总线可用信号的建立。完成数据传送后,部件除去总线忙信号,总线可用信号也随之去除。此后若有总线请求,则再次开始总线分配过程。

可见,这种方式使使用总线的优先次序完全由总线可用线所接部件的物理位置来决定,离总线控制器越近的部件其获得总线使用权的优先级别越高,越远的部件优先级别越低。

串行链接方式的主要优点是总线裁决算法很简单,用于控制总线分配的线数很少,而且与挂接在总线上的部件的数量无关,易于扩充设备。但这种方式由于优先级是固定的,灵活性较差,不能由软件改变优先级,如果级别高的部件频繁使用总线时,优先级低的部件可能很久也得不到响应。又由于总线可用信号串行地通过各个部件,这限制了总线分配的速度;在总线可用信号传输的过程中,如果第 I 个部件发生

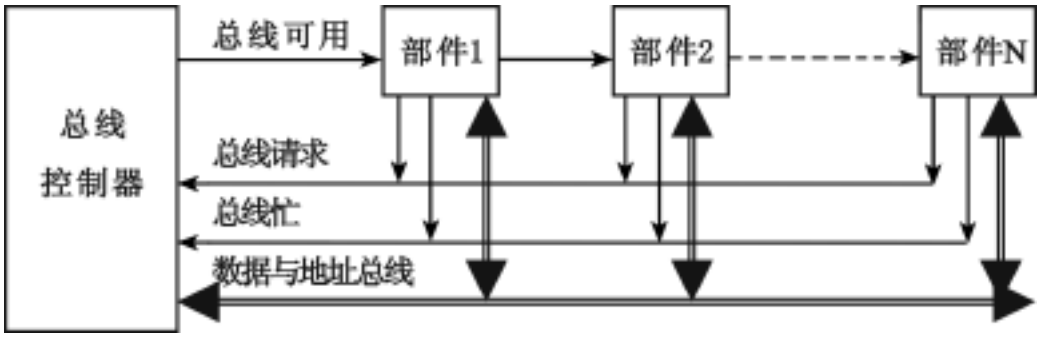


图 3 .5 串行链接式总线控制

故障,在其后的所有部件将永远得不到总线的使用权,即对硬件的失效很敏感。在总线上增加、去除或移动部件也要受总线长度的限制。

2 .定时查询方式

图 3 .6 为采用查询方式的集中式总线控制方式。查询方式的原理是在总线控制器中设置一个查询计数器。由控制器轮流地对各部件进行测试,看其是否发出总线请求。当总线控制器收到申请总线的信号后,计数器开始计数,如果申请部件编号与计数器输出一致,则计数器停止计数,该部件可以获得总线使用权,并建立总线忙信号,然后开始总线操作。使用完毕后,撤消总线忙信号,释放总线,若此时还有总线请求信号,控制器继续进行轮流查询,开始下一个总线分配过程。

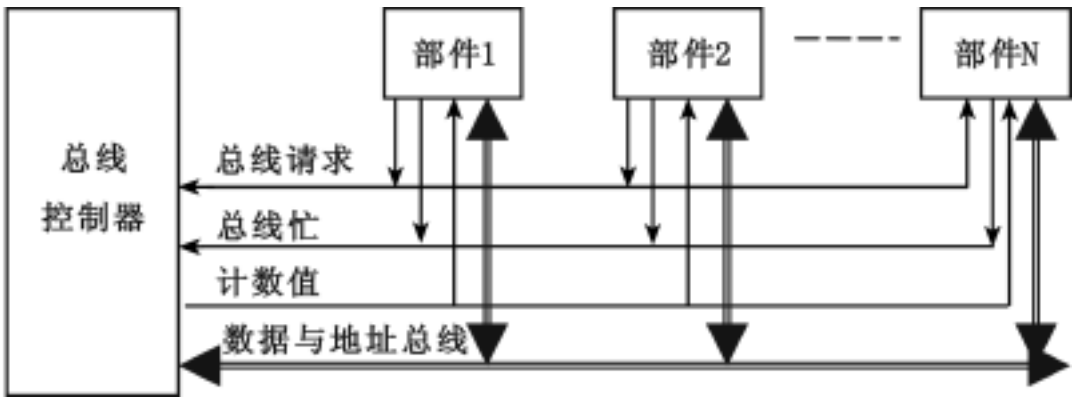


图 3 .6 定时查询式总线控制

计数器的值可以每次从“ 0 ”开始计数,这时部件的优先级类似于串行链接方式;如果计数器的值每次从上次的中止点开始计数,则是一种循环优先级,每个部件获得总线使用权的机会均相等;计数器的值还可以通过程序的方法来改变,在每次总线分配前赋予计数器一个起始值,同样,部件号也可以由程序置定,这样部件的优先级有较灵活的改变。

查询方式是用计数查询线代替了串行链接方式的总线可用信号线,这样不会因某一部件的故障而引起其他部件获得总线的使用权,故可靠性比较高。但查询线的

数目限制了总线上可挂接的部件数目,扩充性较差,而且控制较为复杂,总线的分配速度取决于计数信号的频率和部件数,速度仍然不会很高。

3 .独立请求方式

每个部件都有各自的一对总线请求和总线允许线,各部件可以独立地向控制器发出总线请求,总线已被分配信号线是所有部件公用的,如图 3 .7 所示。当部件要申请使用总线时,送总线请求信号到总线控制器,如果总线已被分配信号线还未建立,即总线空闲时,总线控制器按照某种算法对同时送来的请求进行裁决,确定响应哪个部件发来的总线请求,然后返回这个部件相应的总线允许信号。部件得到总线允许信号后,去除其请求,建立总线已被分配信号,这次的总线分配结束,直至该部件传输完数据,撤消总线已被分配信号,经总线控制器去除总线准许信号,可以接受新的申请信号,开始下一次的总线分配。

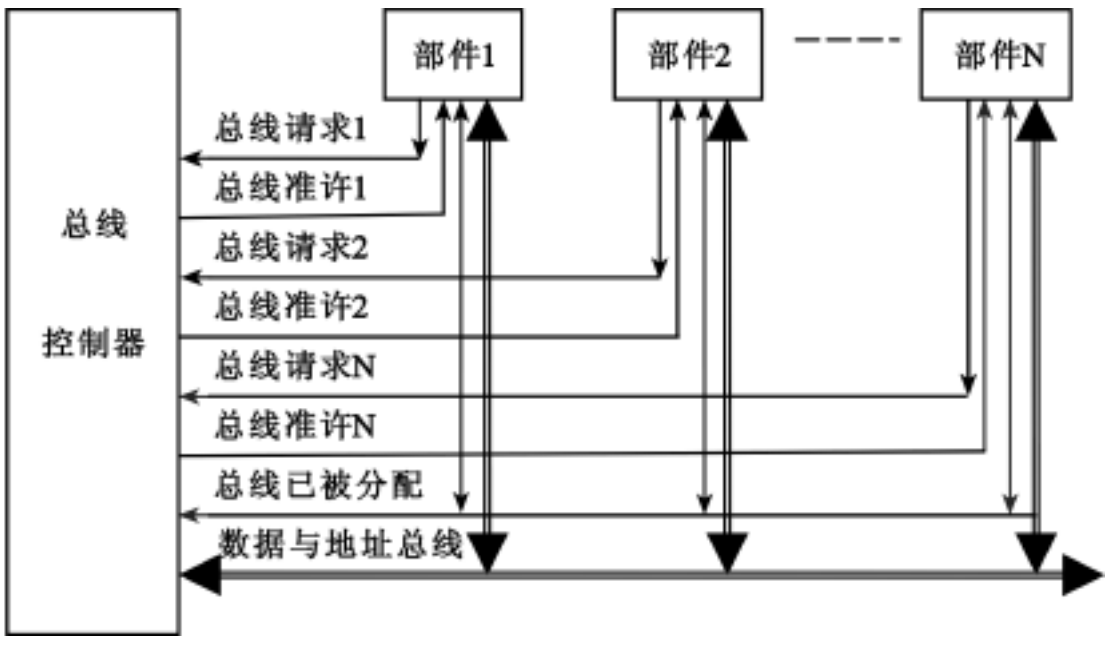


图 3 .7　独立请求式总线控制

这种方式的总线分配速度快,各模块优先级的确定灵活,既可以采用优先级固定法,也可以通过程序改变优先次序,还可以通过屏蔽禁止某个请求,也能方便地不响应来自已知失效或可能失效的部件发出的请求,但这是以增加总线控制器的复杂性和控制线的数目为代价的。

上述三种集中式控制方式都是采用优先级进行总线的分配,为使总线的裁决和总线的操作能重叠进行,要求裁决算法简单,这样易于用硬件实现。常用的总线裁决算法有静态优先级算法、固定时间片算法、动态优先级算法和先来先服务算法。

静态优先级算法是对每个总线设备请求设置固定的优先级,当多个设备同时要求使用总线时,其中优先级最高的获得总线的使用权,所以也称菊花链算法,串行链接控制方式采用的就是这种总线裁决算法。如 DEC PDP-11 的 Unibus(单种线)和

Motorola MC 68000 的总线中都采用这种方法。

如果将优先级在总线各设备之间轮转,如果轮到的设备不要求使用总线,则将使用总线的权利传递给下一个设备,这种方式就称为固定时间片算法。定时查询方式就是采用的这种算法。

在固定时间片算法中,各设备获得总线使用权的机会均等,但设备获得总线使用权的平均等待时间较长。为了解决这个问题,而又可保证设备有较平等的机会使用总线,可采用动态优先权的算法。这种算法是赋予各个设备惟一的优先级,这个优先级可以动态改变。设备优先级改变的方法主要有两种:最近最少使用算法(LRU)和轮转菊花链算法(RDC)。LRU 算法将最高优先级赋予最久未使用总线的设备,而 RDC 算法中,不需要集中控制器,总线许可信号来自各设备构成的环中的上一个设备。占用总线的设备就是总线的控制器,由它对下一个总线请求操作进行裁决。每个设备的优先级取决于它与占用总线设备的距离,这样设备的优先级可在各设备之间动态改变。这种总线控制算法比较适用于面向多机系统的总线,因为这种系统通常要求各处理机对称。

先来先服务算法就是对先发送来总线请求的设备予以优先响应,这样各设备获得总线使用权的机会均等,而且这种算法各设备的平均等待时间最少。但因为控制器必须记住各个总线请求到达的顺序,有时难以区分这种次序,因此实现比较困难。

集中式控制的三种主要方式各有优缺点,其中定时查询和独立请求方式主要用在中型机以上,而小、微型机上用得最广泛的还是串行链接方式,主要是因为它简单,易于实现,可扩充性好。至于各个设备优先级是固定的且对部件故障失效比较敏感,可以根据不同的需要通过设置具有多根有不同优先级次序的总线可用线,并用软件控制相应的总线可用线工作来弥补。

3.2.3 总线通信技术

当获得了总线的使用权后,就要开始信息的传送。为了使信息能通过总线传输,就必须对总线上信号的传输方式作统一的规定,发送端、接收端都必须遵守这个协议。

信号在总线上的传输方式基本上可分为同步和异步两种。

1.同步通信

两个通信部件之间的信息传输由定宽、定距的系统时标信号同步。每个功能部件什么时候发送或接收信息都由统一时钟规定,如图 3.8 所示。因此,同步通信数据传输速率较高,而且受总线长度的影响小,即当信号在总线上因长度而滞后时,也不会影响传输速率,但是,时标线上的干扰信号会引起错误的同步,滞后的时标也会造成同步误差,且对通信的正确性一般无法实时检验,一般在一个时间片之后,目的部件发回出错信号,申请源部件重发,这要求源部件必须保留原始数据至下一时间片完

毕。

同步方式对任何两个功能模块的通信都给予同样的时间安排,总线必须按最慢的模块设计公共时钟,当各功能模块存取时间相差很大时,会大大损失总线效率。所以同步通信适用于总线长度较短、各功能模块存取时间比较接近的情况。CPU-主存总线是典型的同步总线,如 PCI, NUBUS 和 Multibus 等都是同步总线的实例。

由于 I/O 系统中,各外部设备的数据传输速率相差较大,所以 I/O 总线一般不采用同步通信方式。

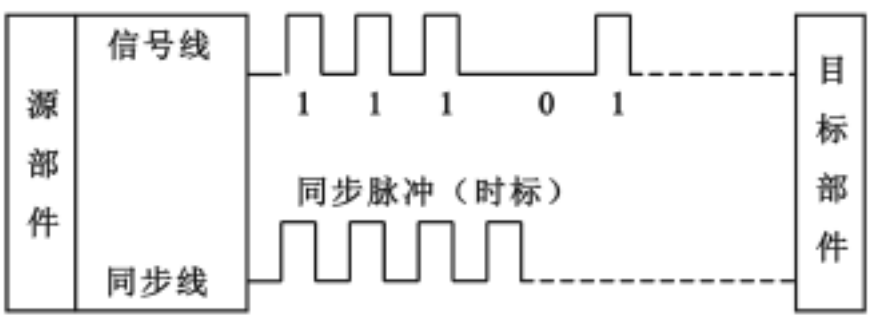


图 3.8 同步通信示意图

2.异步通信

异步通信又称应答通信,是一种建立在应答式或互锁机制基础上的通信方式。即后一事件出现在总线上的时刻取决于前一事件的出现。在这种系统中,不需要统一的公共时钟信号,总线周期的长度是可变的,不把响应时间强加到功能部件上,因而允许快速和慢速的功能部件都能连接到同一总线上,但这是以增加总线的复杂性和成本为代价的。

异步通信中根据应答信号是否互锁,即请求和回答信号的建立和撤消是否互相依赖,异步通信可分为三种类型:非互锁通信、半互锁通信和全互锁通信。图 3.9 给出了这三种方式的通信联络。图中 t_1 为数据在总线上达到稳定所需要的时间,t_{d1} 为一次总线传输的延迟时间和目的部件执行时间之和,t_{d2} 为一次总线传输延迟。

图 3.9(a)所示为非互锁通信方式。在这种方式下应答信号的建立和撤消互相没有依赖关系。假定源部件将数据放到数据总线上,经 t_1 延迟后再在控制总线上发“READY”数据准备好信号,以给目的部件作接收数据的选通信号用,延迟 t_1 是为了防止“READY”信号可能先于数据“DATA”到达目的部件而产生的定时错。目的部件经过 t_{d1} 时间后,便接收数据,并发数据接收回答信号“ACK”,源部件在 t_{d2} 时间后撤消数据,准备下一个数据的传送。

半互锁通信是指应答信号的建立与撤消存在一定的依赖关系,如图 3.9(b)所示。和非互锁方式不同的是:源部件在收到回答信号“ACK”后就撤消请求信号,即请求信号的撤消依赖回答信号的建立。然而,回答信号的撤消与请求信号无依赖关

系,因此,称为半互锁。

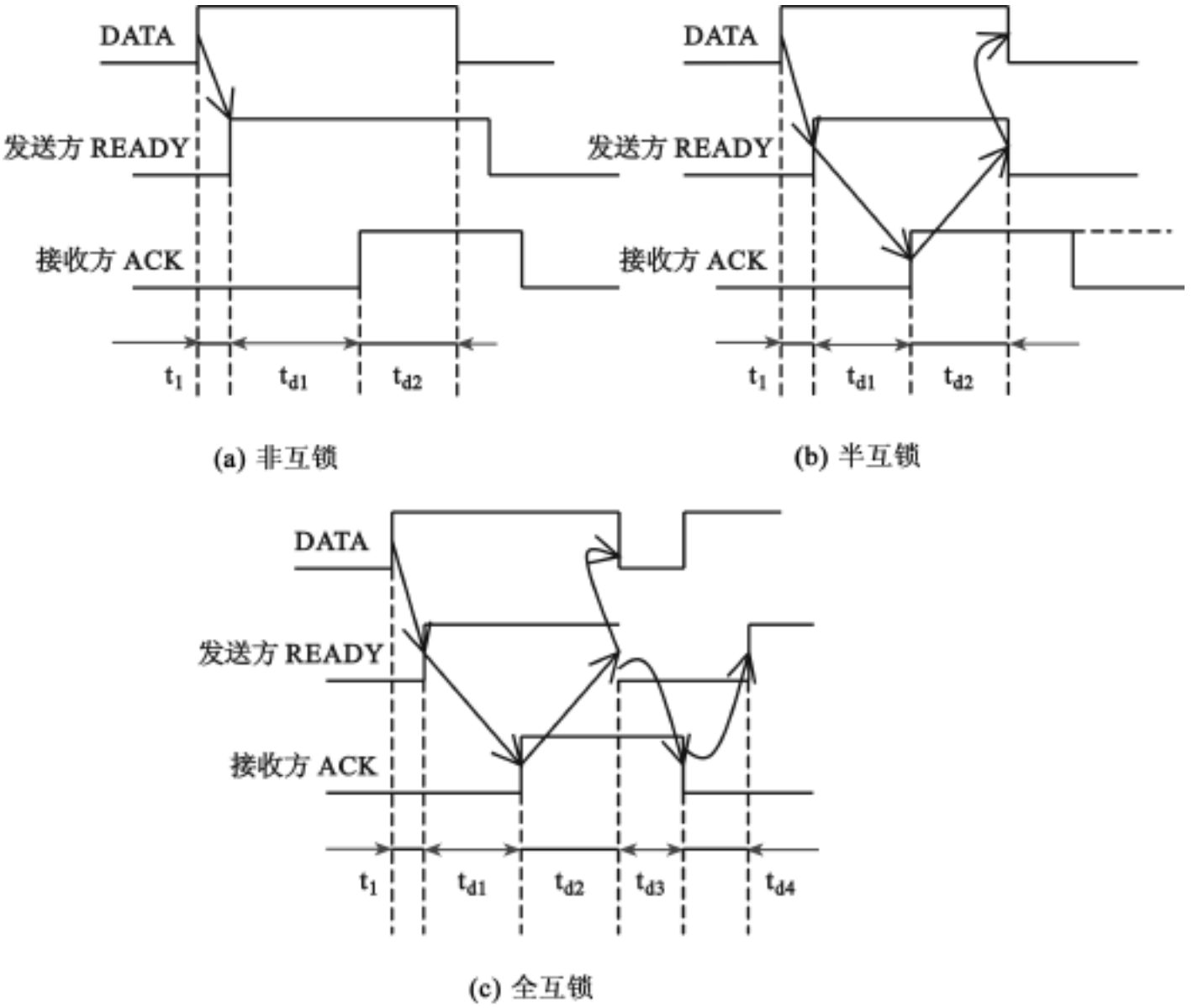


图 3.9 异步通信三种方式

半互锁方式下,如果下一个数据的数据准备好信号“READY”到来后,回答信号仍未复位,这时“READY”信号会使数据回答信号“ACK”一直处于高电平,如图中虚线所示,从而造成错误,这可以通过全互锁方式解决。

全互锁通信是指应答信号的建立与撤消存在着完全的依赖关系,如图 3.9(c)所示。源部件在收到回答信号后就撤消请求信号,请求信号的撤消又使目的部件的回答信号失效,当源部件测试到回答信号已撤消,可以再发数据准备好,即请求信号,因此,保证了一个数据的请求信号只能在一个回答信号结束后才能发出,从而大大提高了通信的可靠性。

显然,这三种通信方式中,全互锁通信可保证数据的正确发送和接收,可靠性最高,不过这增加了信号沿总线来回传送的次数,也使控制硬件稍许复杂些。但这种方式既能适应各种 I/O 设备的不同速度,保证数据的正确传送,又能有较高的数据传

输速率,因此它总是以所接源和目的部件中相对较低的速率来通信,这比同步方式中以所有部件中最低速率进行通信的效率要高,所以异步通信对不同的设备可以有不同的传输速率,所以适合于 I/O 系统总线。

3.2.4 总线设计

既然总线是各子系统间共享的通信链路,它的最大优点就是低费用和多能性。通过定义一种互连模式,一个新设备可以很容易地加到总线上去,外部设备可以通过公共总线在计算机系统间进行移动。由于总线的一组金属线可以多路共享,因此价格较低。

总线的主要缺点是它同时只允许一对部件进行通信,可能会成为通信的瓶颈,限制系统中总的 I/O 吞吐率。如在大型计算机中,由于 CPU 性能较高,所需的 I/O 频率也高,设计一个总线系统,满足处理器的需求是一个巨大的挑战。

总线设计困难的一个重要原因是总线的速度受到物理因素的严重限制,如总线的长度、所接设备的数量等,另外,高速率(低延迟)和高 I/O 吞吐率也导致与设计需求相冲突。

总线的设计有多种选择,像计算机其他部分一样,总线设计需要考虑价格和性能两个指标。一般总线设计时需考虑的因素如表 3.1 所示。

表 3.1　　总线设计要考虑的主要因素

选　项	高　性　能	低　性　能
总线宽度	独立的地址和数据总线	地址与数据总线分时复用
数据总线宽度	越宽越快(如 64 位、128 位)	越窄越便宜(如 8 位)
传输块大小	块越大总线开销越小	单字传输简单
主设备数量	多个(需要对总线使用权进行仲裁)	单个主设备
分段操作	采用(单独的请求和应答包,需较高的总线带宽)	不用(持续的连接便宜、延迟小)
通信方式	同步通信	异步通信

表 3.1 中前三项对总线设计的影响最为明显。如果追求高流量高性能,可以采用地址和数据总线分开设计的方法,同时增加数据总线的位数和每次传输的位数,当然这是以增加成本为代价的。

I/O 设备取得 I/O 总线使用权后所传送数据的总量,称为“数据宽度”,即为每次传输的块的大小。传送完这些数据后,总线要重新进行分配。它不同于数据总线宽度,图中第二项的数据总线宽度是指数据传送的物理宽度,即一个时钟同期所传送的

信息量,它直接取决于数据总线的线数。二次分配总线之间所传送的数据宽度可能要经过许多个时钟周期的分次传送来完成。数据宽度的种类有单字、定长块、可变长块、单字加定长块和单字加可变长块等。单字传输适合低速率的设备,定长块等适合于磁盘等高速设备。采用何种数据宽度和具体的总线控制方式、通信技术、设备特点等都是密切相关的。

当有多个 CPU 或有多个 I/O 设备可以启动总线时,一条总线上就有多个主设备。当有多个主设备申请使用总线时,总线可通过数据打包来提高带宽,这样就不必在整个传输过程中都占有总线,这种技术称为分段总线操作。分段总线操作的工作过程如图 3.10 所示。读操作分成包含地址的读请求操作和包含数据的存储器应答操作。CPU 和存储器通过标识来识别每次总线操作。分段操作使总线在存储器从指定地址读取信息时能够为其他主设备服务,这要求 CPU 对于总线发送来的数据和主存对总线的返回数据必须能够正确识别。因此分段操作有较高的带宽,但是它比一个主设备在整个数据传输过程中独占总线的延迟要大。

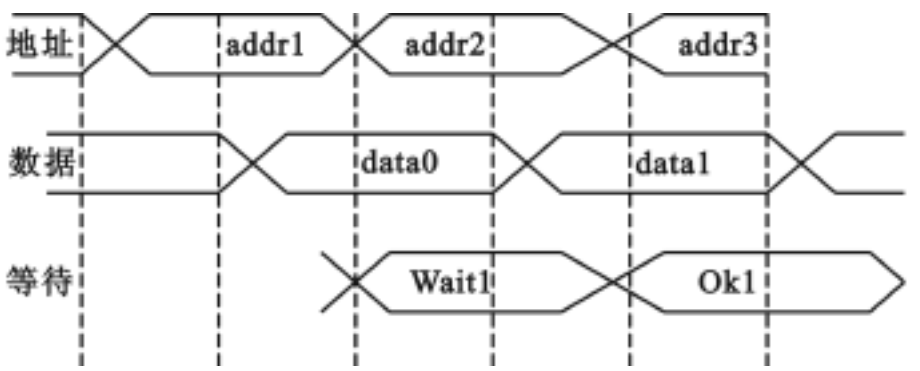


图 3.10 分段总线操作(总线上的地址对应于最近的存储器访问)

一些系统中把分段操作也称为连接/断开连接、流水总线或包交换总线。

至于选用何种通信方式,由前面内容可知,同步通信较快且费用低,但要求总线长度要短,一般适用于存储总线,而异步总线可以接受各种类型的设备,并可以加长总线长度而不需要考虑时钟扭曲和同步总量,且便于总线技术的更新,所以 I/O 总线通常是异步的。

一旦确定了总线类型、通信方式、数据宽度、控制方式等,总线的申请、使用方式及相应的规范也就确定了,所有要使用总线的设备或部件都必须遵守这个规范,这个规范就是总线的标准化。只要计算机系统设计者和 I/O 设备设计者都遵守总线标准,任何 I/O 设备都可以连接到任何计算机上。实际上,I/O 总线标准就是定义计算机如何连接设备的文件。I/O 总线的标准实例有 S 总线(32 位)、微通道(32 位)、PCI(32 位或 64 位)、SCSI2(8 到 16 位)等,CPU—存储器总线实例有 HP Summit(128 位)、SGI Challenge(256 位)、Sun XDBus(144 位)等,这三种存储器总线都采用了分段操作来提高带宽,峰值带宽分别为 960MB/s,1200MB/s,1056MB/s。

注意 I/O 总线的设计一定不能使它成为 I/O 系统的性能瓶颈。

3.2.5 Pentium 微处理器的总线系统

大多数计算机采用了分层次的多总线结构。在这种结构中,速度差异较大的设备模块使用不同速度的总线,而速度相近的设备模块使用同一类总线。显然,这种结构的优点不仅解决了总线负载过重的问题,而且使总线设计简单,并能充分发挥每类总线的效能。

图 3.11 是 Pentium 计算机主板的总线结构框图。为了便于说明,框图以使用 Pentium II-300MHz 处理器为例。可以看出,它是一个三层次的多总线结构,即有 CPU 总线、PCI 总线和 ISA 总线。

CPU 总线:也称 CPU—存储器总线,它是一个 64 位数据线和 32 位地址线的同步总线。总线时钟频率为 66.6MHz(或 60MHz),CPU 内部时钟是此时钟频率的倍频。此总线可连接 8MB～1GB 的主存。主存扩充容量是以内存条形式插入主板有关插座来实现的。主存控制器和 Cache 控制器芯片用来管理 CPU 对主存和 Cache 的存取操作。CPU 是这条总线的主控者,但必要时可放弃总线控制权。

PCI 总线:用于连接高速的 I/O 设备模块。如图形显示器、网络接口控制器、硬盘控制器等。通过"桥"芯片,上面与更高速的 CPU 总线相连,下面与低速的 ISA 总线相接。PCI 总线是一个 32 位(或 64 位)的同步总线,32 位(或 64 位)数据/地址线是同一组线,分时复用。总线时钟频率为 33.3MHz,总线带宽是 132MB/s。PCI 总线采用集中式仲裁方式,有专用的 PCI 总线仲裁器。主板上一般有 3 个 PCI 总线扩充槽。

ISA 总线:Pentium 机使用该总线与低速 I/O 设备连接。主板上一般留有 3～4 个 ISA 总线扩充槽,以便使用各种 16 位/8 位适配器卡。该总线支持 7 个 DMA 通道和 15 级可屏蔽硬件中断。另外,ISA 总线控制逻辑还通过主板上的片级总线与实时钟/日历、ROM、键盘和鼠标控制器(8042 微处理器)等芯片相连接。

CPU 总线、PCI 总线、ISA 总线通过两个"桥"芯片连成整体。桥芯片在此起到了信号速度缓冲、电平转换和控制协议的转换作用。通过桥将两类不同的总线"粘合"在一起的技术特别适合于系统的升级换代。按照摩尔定律,几乎每 18 个月微处理器芯片就升级一次。现在,每当微处理器改变时只需改变 CPU 总线和改动"北"桥芯片,全部原有外部设备则可自动继续工作。

Pentium 计算机总线系统中有一个核心逻辑芯片组,简称 PCI 芯片组,它包括主存控制器和 Cache 控制器芯片、北桥芯片和南桥芯片。这个芯片组叫 Intel 430 系列、440 系列,调整 CPU、内存、Cache、PCI 总线和其他外围总线(ISA,SCSI,PC 卡,USB,1394)等之间的流量,将多种总线"粘合"成一个整体,它们在系统中起着至关重要的作用。

从图 3.11 中可以看到,还有另外两种专用总线:

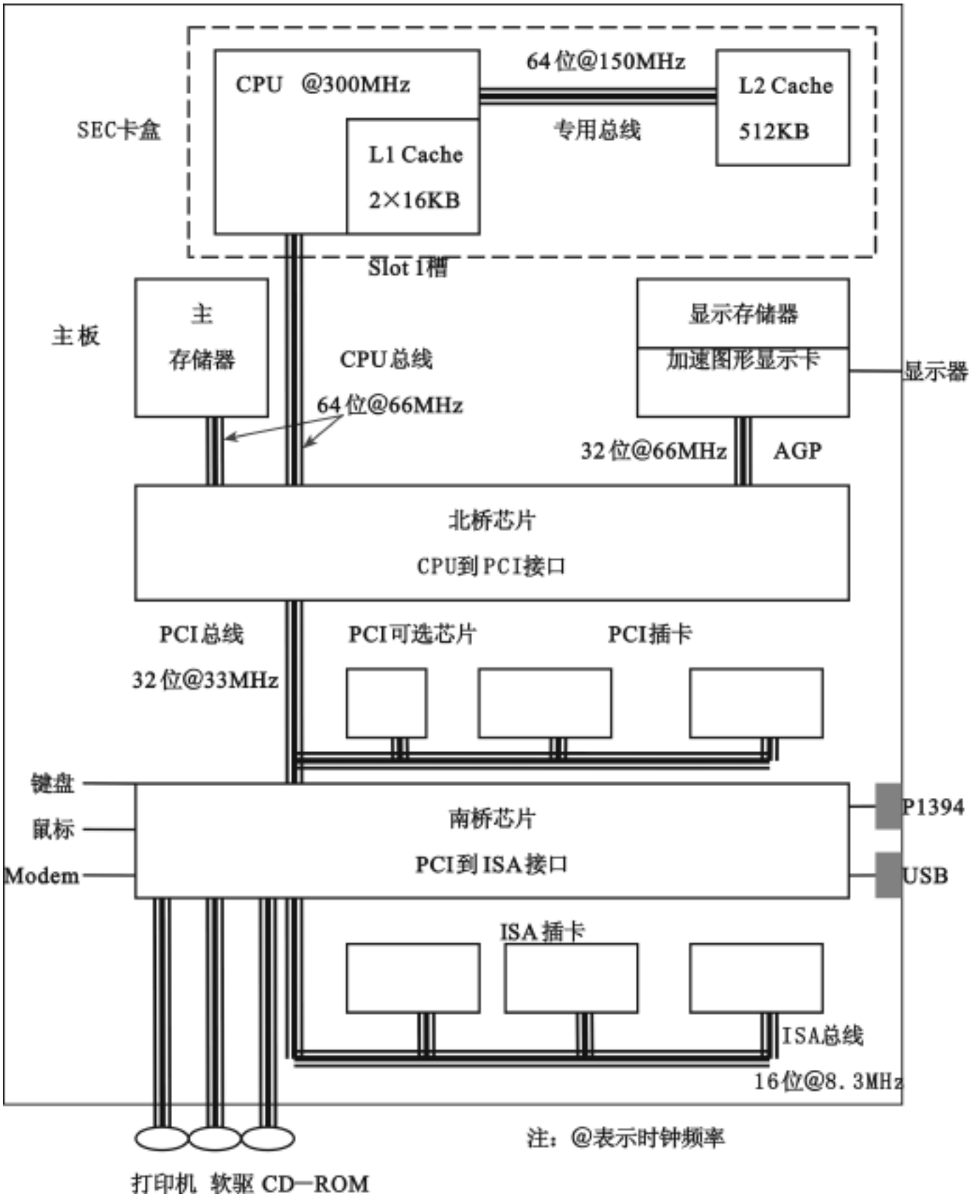


图3.11 Pentium计算机主板总线结构框图

一是L2 Cache与处理器核心相连的总线。由于Pentium II处理器采用双独立总线结构(DIBA),L2 Cache在处理器卡盒内部通过专用的后端总线(BSB)与处理器核心相连,并且以处理器核心工作频率(主频)的一半速率工作。如图3.11中,使用64位、时钟频率为150MHz的专用总线与CPU连接。监听系统其他主控者访问主

存的活动并维护 Cache 一致性的工作,由 PCI 芯片组的北桥芯片与 CPU 联手完成。这种将 Cache 总线独立出去的做法,不仅减轻了 CPU 总线的负荷,也使 L2 Cache 工作速率由 66MHz 提高到 150MHz。它的发展趋势是专用总线将以 CPU 工作时钟速率同速运行,L2 Cache 的容量进一步扩充到几 MB。

二是专用的 AGP 总线。专用的 AGP 总线将加速图形处理速度,以适应高速增长的 3D 图形变换和生动视频显示等需求。

3.2.6 Pentium 系列微计算机系统的输入输出总线(USB 和 IEEE 1394)

目前的 Pentium 中都已集成有两种标准的串行 I/O 总线,即 USB 和 IEEE 1394。它们有很多相似之处,但 USB 主要是面向低、中速外部设备,而 IEEE 1394 的速度一开始就是 USB 最高速度的 8 倍。这里先介绍 IEEE 1394,然后再介绍 USB。

1 .IEEE 1394 标准

随着处理器速度的提高,存储器容量已达到吉(G)位级,由于 PC、工作站和服务器对快速 I/O 的强烈需求,工业界期望能有一种更高速、连接更方便的 I/O 总线,对此,串行 SCSI 业已提出一些好的策略。1993 年 Apple 公司公布了一种高速串行总线 Fire Wire,希望能取代并行的 SCSI 总线。IEEE(电气与电子工程协会)接管了这项工作,在此基础上制定了 IEEE 1394 标准。

(1)IEEE 1394 性能特点

IEEE 1394 串行总线与 SCSI 等并行总线相比,有如下三个显著特点:

① 数据传送的高速性

IEEE 1394 的数据传输率分为 100Mb/s、200Mb/s、400Mb/s 三档。而 SCSI-2 的数据传输率只有 40MB/s(相当于 320Mb/s),小于 IEEE 1394 的最高数据传输率。这样的高速特性特别适合于新型高速硬盘及多媒体数据传送。IEEE 1394 之所以达到高速,一是串行传送比并行传送容易提高数据传送时钟速率;二是采用了 DS-Link 编码技术,把时钟信号的变化转变为选通信号的变化,即使在高的时钟速率下也不易引起交调失真。

② 数据传送的实时性

实时性可保证图像和声音不会出现时断时续的现象,因此对多媒体数据传送特别重要。IEEE 1394 之所以做到实时性,原因有二:一是它除了异步传送外,还提供了一种等步传送方式,数据以一系列的固定长度的包规整间隔地连续发送,端到端既有最大延时限制又有最小延时限制;二是总线仲裁除优先权仲裁外,还有均等仲裁和紧急仲裁两种方式。

③ 体积小易安装,连接也方便

IEEE 1394 使用 6 芯电缆,直径约为 6mm,插座也小。而 SCSI 使用 50 芯或 68 芯电缆,插座体积大。在当前计算机要连接的设备越来越多、主机箱的体积越来越窄

小情况下,电缆细、插座小的 IEEE 1394 是很有吸引力的,尤其对笔记本电脑一类机器更是如此。IEEE 1394 的电缆不需要与电缆阻抗匹配的终端器,而且电缆上的设备随时可从插座拔出或插入,即具有热插入能力。这种对 PnP 的支持,对用户安装和使用 IEEE 1394 设备很有利。

(2)IEEE 1394 配置结构

IEEE 1394 采用菊花链式配置,也允许树形结构配置。事实上,菊花链结构是树形结构的一种特殊情况。

IEEE 1394 总线也需要一个主适配器和系统总线相连。这个主适配器的功能逻辑在高档的 Pentium 机中集成在主板的核心芯片组的 PCI 总线到 ISA 总线的桥芯片中(即所谓的"南"桥芯片内)。机箱的背面只看到主适配器的外接端口插座。

我们将主适配器及其端口称为主端口。主端口是 IEEE 1394 总线树形配置结构的根节点。一个主端口最多可连接 63 台设备,这些设备称为节点,它们构成亲子关系。两个相邻节点之间的电缆最长为 4.5m,但两个节点之间进行通信时中间最多可经过 15 个节点的转接再驱动,因此通信的最大距离是 72m。电缆不需要终端器。

IEEE 1394 配置的实例见图 3.12。其中右侧是线性链接方式,左侧是亲子层次链接方式,整体是一个树形结构。

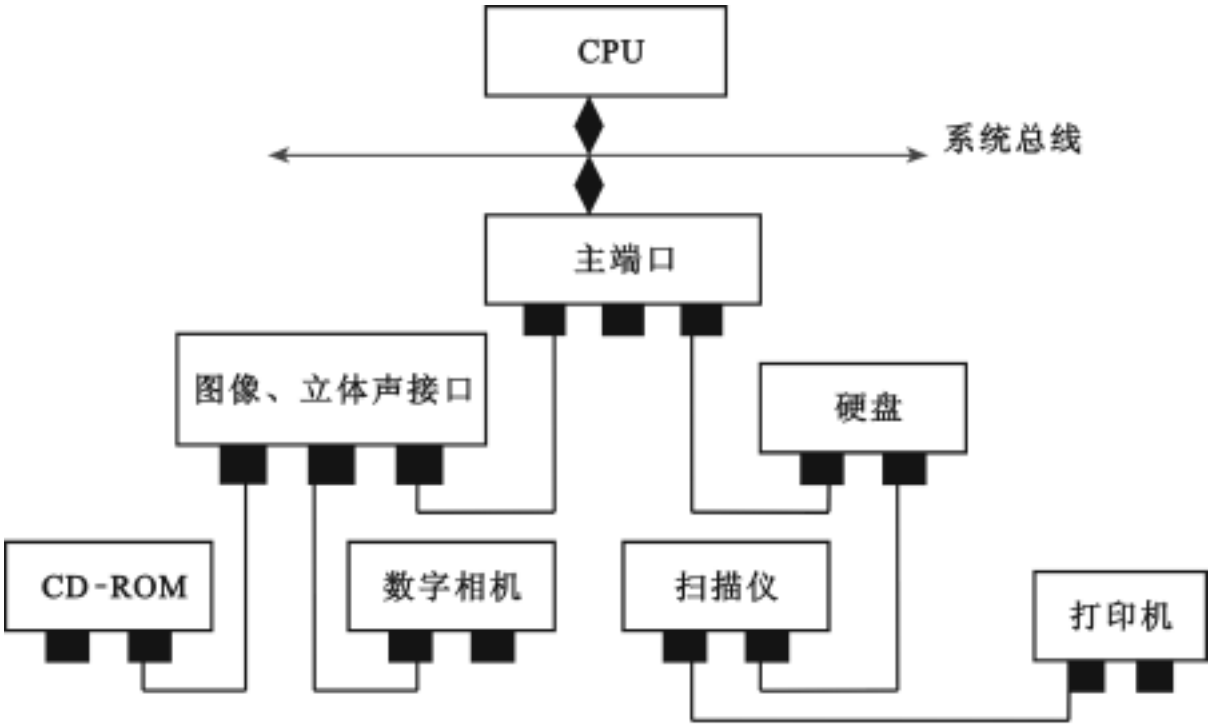


图 3.12 IEEE 1394 总线配置举例

IEEE 1394 采用集中式总线仲裁方式。中央仲裁逻辑在主端口内,并以先到先服务方法来处理节点提出的总线访问请求。在 n 个节点同时提出使用总线请求时,按照优先权进行仲裁。最靠根节点的竞争节点有高的优先权;同样靠近根节点的竞争节点,其设备标识号 ID 大的有更高优先权。IEEE 1394 具有 PnP 功能,设备标识

号是系统自动指定的,而不是用户设定的。

为了保证总线设备的对等性和数据传送的实时性,IEEE 1394 的总线仲裁增加了均等仲裁和紧急仲裁功能。

均等仲裁:是将总线时间分成均等的间隔,当间隔期间开始时,竞争的每个节点置位自己的仲裁允许标志,在间隔期内各节点可竞争总线的使用权。一旦某节点获得总线访问权,则它的仲裁允许标志被复位,在此期间它不能再去竞争总线,以此来防止具有高优先权的忙碌设备独占总线。

紧急仲裁:指对某些高优先权的节点可为其指派紧急优先权。具有紧急优先权的节点可在一个间隔期内多次获得总线控制权,允许它控制 75%的总线可用时间。

(3)IEEE 1394 协议集

IEEE 1394 的一个重要特色是:它规范了一个三层协议集,将宿主系统经由串行总线与各外部设备的交互动作标准化。

图 3 .13 为 IEEE 1394 协议集。

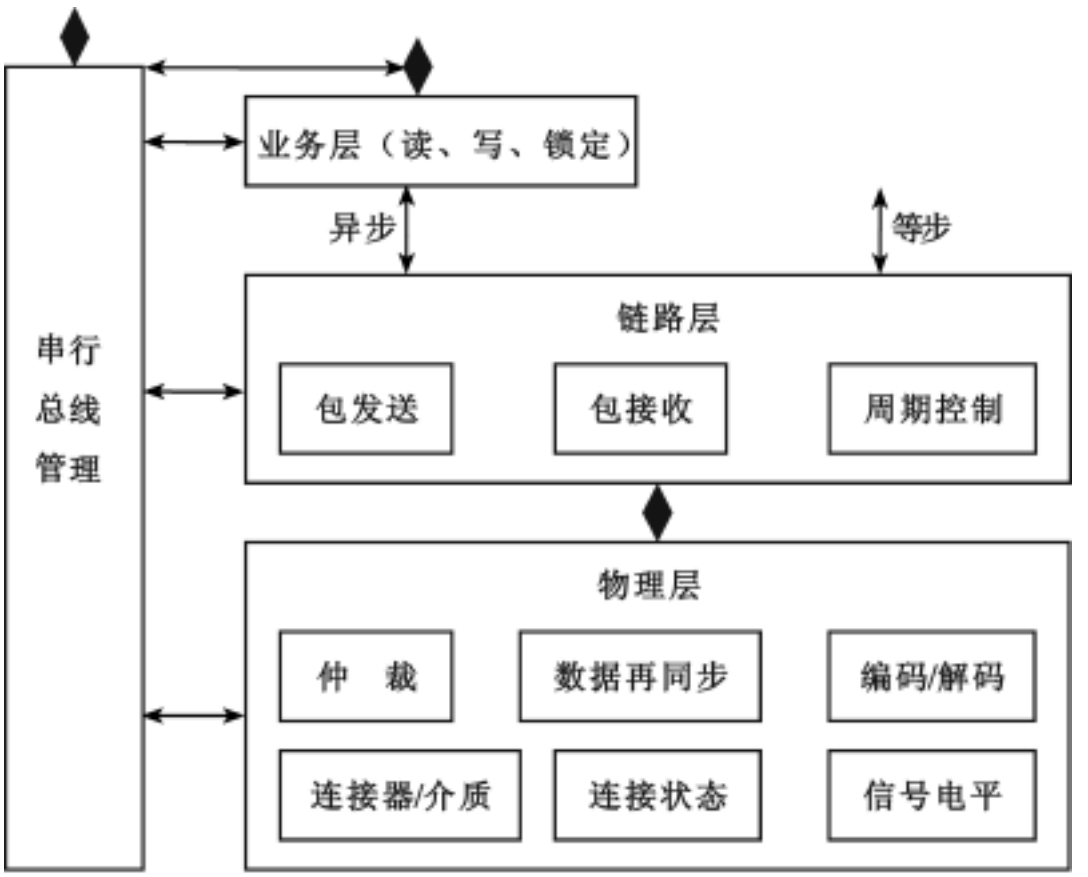


图 3 .13　IEEE 1394 协议集

业务层: 定义了一个完整的请求—响应协议实现总线传输,包括读操作、写操作和锁定操作。

链路层: 可为应用程序直接提供等步数据传送服务,提供地址化、数据校验和数据成帧服务。它支持异步和等步的包发送和接收。异步包传送是,一个可变总量的数据及业务层的几个信息字节作为一个包传送到显式地址的目标方,并要求返回一

个认可包。等步包传送是,一个可变总量的数据以一串固定大小的包按照规整间隔发送,使用简化寻址方式,不要求目标方认可。IEEE 1394 把完成一个包的递交过程称为子动作。链路层可为应用程序直接提供等步数据传送业务。

物理层:将链路层的逻辑信号根据不同的串行总线介质转换成相应的电信号,也为串行总线的接口定义了电气和机械特性。实际上,IEEE 1394 串行接口的物理拓扑结构分成‘底板环境”和‘电缆环境”两部分。总线规范并未要求特别的环境设定,所有节点可严格限定在单一底板上,也可直接连在电缆上。也可以既有在底板上的,也有在电缆上的。两种环境下的物理层也有所不同。

串行总线管理:它提供总线节点所需的标准控制、状态寄存器服务和基本控制功能。

总之,IEEE 1394 是一种高速串行 I/O 标准总线。英特尔、微软等公司联手制定的 PC98 系统设计指南中明确将 IEEE 1394 列为 1998 年以后的新一代个人机新标准。IEEE 1394 的另一个重大特点是:各被连接装置的关系是平等的,不用 PC 介入也能自成系统。如在 1998 年北京中国家用电器博览会上,SONY 公司以 IEEE 1394 为接口,将微机与电视、音响、摄录放一体机、数据存档系统、数字相机、彩色打印机、图像采集卡等连接起来,构成了一个标准的家庭网络环境,这意味着 IEEE 1394 在家电等消费类设备的连接应用方面有很好的前景。但该标准并不局限于家庭环境,同样适用于多种计算机外部设备和远程网中。

2.通用串行总线 USB

USB 是由 Intel,Compaq,DEC,IBM,Microsoft,NEC 和 Northern Telecom 7 家公司联手制定的一种串行总线标准。1994 年 11 月制定了第一个草案,1996 年 2 月公布了 USB 标准版本 1.0。现在有上百家公司支持这一标准,USB 接口已在高档 Pentium PC 中实现。

USB 与 IEEE 1394 有许多共同点,如它们都是串行 I/O 总线,设备连接方便,都实现了即插即用和热插入,都有适合传送多媒体数据的传送模式,都可由电缆向设备提供(功率有限)的电源等。它与 IEEE 1394 最大不同在于传送速度低,最高只有 12Mb/s,最低只有 1.5Mb/s。USB 主要用于键盘、鼠标器等低速设备和扫描仪、MODEM、数字相机、打印机等中速设备与 PC 的连接。下面再简要介绍 USB 的其他一些主要特点:

(1)使用集线器的树形连接

USB 使用的 4 线电缆:两根串行位传送线,采用 NRZ1(不归零翻转)编码方式,一根电源(+5V)线,一根地线。

USB 使用集线器(hub)经电缆对设备进行树形连接。必有一个主集线器在 PC 机内与主总线(如 PCI)相连接,以它为根节点最多可分成 6 层对外部设备进行树形连接,最多可接 127 个设备(地址为 1～127,0 是未初始化设备)。上下两层设备之间

是亲子关系,往上层连接的电缆插头与往下层连接的电缆插头不一样,不能插错。各设备都只能与主集线器进行通信并接受它的控制。

(2)USB 的帧格式

USB 支持四种类型的帧:控制、等步、成块、中断。控制帧用于配置设备,发布命令和查询状态。等步帧用于有实时要求的设备,以精确的时间间隔发送或接收数据,具有高度可预期的延迟,但出错不提供重发。成块帧用于无实时要求的设备(如打印机)的大量数据传送。为支持以中断处理完成操作的设备(如键盘),操作系统采用定时(如每 50ms 一次)查询方式,中断帧用这样的设备向系统报告中断。图 3 .14 给出了一个 USB 的帧序列示意图。

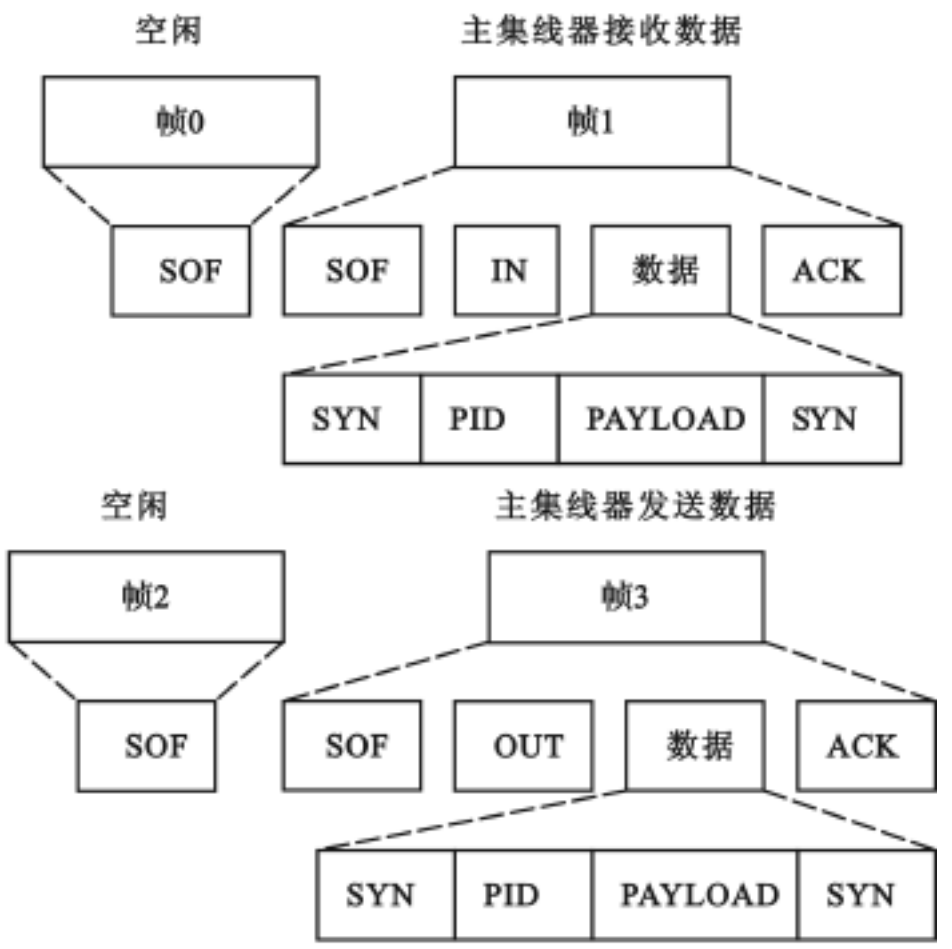


图 3 .14　USB 的帧序列

一个帧由一个或多个包组成。USB 有四种类型包:令牌、数据、握手联络和特殊。令牌包由主集线器发往设备用于系统控制,图 3 .14 中的 SOF(帧开始)包、IN 包、OUT 包都是令牌包。握手联络包有 ACK(已正确接收)、NAK(出错)和 STALL(暂停)三种类型。数据包用于传送可多达 64 字节的数据,其格式也示于图 3 .14 中,它由一个 8 位同步(SYN)域、一个 8 位包标识(PID)域、一个有效负载(PAYLOAD)域和一个 16 位循环冗余校验(CRC)域组成。值得注意的是,即使总线空闲,主集线器也要定时(如每 1 .00±0 .05ms)广播一个只含 SOF 包的帧,以使 USB 上各设备与主集线器保持同步。

总之,USB 是一个面向低、中速设备的串行总线。现在正在努力把 USB 控制功能集成到设备的 ASIC(专用 IC)中,届时设备只需要再添加成本不足 1 美元的插座就可成为 USB 设备。USB 的低成本对 PC 是很有吸引力的。

3.3 中断系统

中断技术的出现,是计算机系统结构设计中的一个重大变革。在 3.1 节中曾经提到,在集中式处理机的 I/O 系统中,中断 I/O 方式是一种重要的 I/O 管理方式,这种方式的主要优点是可以很好地适应外部设备 I/O 请求的随机性,使主机与外部设备能同时工作,提高了主机的工作效率,容易实现外部设备的分级优先管理。在现代的计算机系统中,中断功能不仅仅是 I/O,而且已成为整个计算机系统的功能。如系统监测、异常情况处理、人机联系、实时处理、目态程序和操作系统的联系、多机联系等多种机会随机出现的事件,都以中断方式来处理。本节只讨论中断系统设计过程中软件与硬件功能分配及与其相关的一些问题。

3.3.1 中断系统的分类与分级

计算机系统在运行时,出现某种非预期的事件,CPU 暂时停下现行程序,转向为该事件服务,待事件处理完毕,再从原程序被打断的地方接着往下执行,这个过程就称为中断。引起中断的各种事件称为中断源。根据中断源所处位置不同,中断分为内部中断和外部中断。

内部中断是由主机内部的某种因素引起的,它又可分为强迫中断和自愿中断两种。强迫中断产生的原因有硬件故障和软件出错。硬件故障包括由各部件中的集成电路芯片、元件、器件、印刷线路板、导线及焊点引起的故障,一般将电源故障和电源电压的下降也归于硬件故障。软件出错包括指令出错、程序出错、数据出错、地址出错等。强迫中断是 CPU 无法预料的情况的出现引起的中断,此时,CPU 不得不停下现行的工作,转去进行相应中断处理。而自愿中断则是出于计算机系统管理的需要,自发地进入中断处理。计算机系统为了方便用户调试软件、检查程序错误、调用外部设备等,设置了自中断指令、进管指令。CPU 执行程序时遇到这类指令就进入中断,在中断中调出相应的管理程序。所以自愿中断是可以预料的,即将程序重复执行,断点的位置不改变。

外部中断指的是所有由主机外部事件引起的中断。如操作员对机器干预引起的中断、由外部设备引起的中断、多机联系时引起的中断等。大量的中断是由系统配置的外部设备引起的。这类中断也属于强迫中断。

外部中断可进一步分为可屏蔽中断和不可屏蔽中断。可屏蔽中断通过处理机内部的中断许可状态产生中断,处理机可以通过设置中断屏蔽位来屏蔽一些不重要或不紧急的中断请求。不可屏蔽中断是一些最紧急最重要的中断,如掉电等。中断源

的分类如图 3 .15 所示。

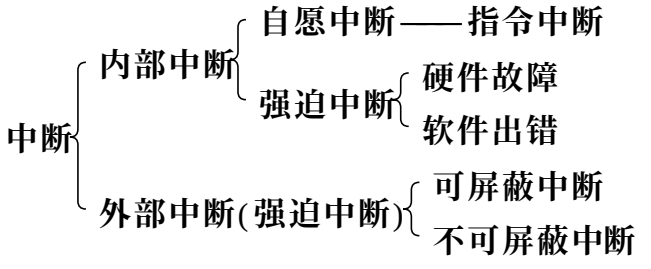


图 3 .15　中断源的分类

中断系统的复杂性就是由于中断源的多样性引起的,如上所述,中断源可以来自系统外部,也可以来自机器内部,甚至是处理器本身的错误。中断可以由硬件引起,也可以由软件引起,所以说,如何将各种各样的中断源进行分类分级,是中断系统设计中首先要做好的一件事情。

1 .中断源的分类

为处理一个中断请求,必须调用相应的中断服务程序。在中断源比较少的情况下,可以通过硬件为每一个中断源形成其相应的中断服务程序的入口,加快中断的处理过程。但现代计算机系统中,中断源的数目很多,特别是对中、大型计算机系统,中断源一般有几十到几百个。如果对每一个中断源都单独用硬件形成其对应的入口地址的话,硬件代价太高,实现也很困难,实际在中断处理上也没有这个必要。因为不少中断源的性质比较接近,可以将它们归为一类,对每一类中断源由硬件形成一个入口地址,再由软件转入对相应的中断源进行处理。

在设计中断系统时,通常根据事件的紧急程度、中断源的工作速率、中断源的性质等进行分类。如 IBM 公司的机器通常把中断源归为如下六类:

① 机器校验中断:由设备故障引起的中断类型。如电源故障、运算器误动作、主存校验错、通道处理机动作故障、处理器的其他各种硬件故障等都属于这一类。当发生这类故障时,用一个 64 位的机器校验中断码以指明故障原因和其严重程度。更为详细的中断原因和故障位置说明保存在机器校验保存区内。

机器校验中断根据对整个系统的影响程度不同,可分为紧急的和可抑制的两种。如某个通道或外设的故障,只影响到局部的输入输出,属于可抑制的机器校验中断,而像 CPU 的地址错,如不及时处理,会使机器无法正常工作,影响整个系统的正常运转,故归为紧急的机器校验中断。

② 程序性中断:由程序性错误引发的中断请求。它包括指令或数据格式出错、程序执行过程中出现的异常(执行了非法指令、目态下使用管态指令、主存访问方式保护、地址越界错误、各种运算溢出、除数为“ 0 ”、有效位为零错误等)、程序的事件记录以及监督程序对事件的检测引起的中断等。

③ 访管中断:用户程序执行“访管”指令引起的中断,通过这种中断,用户可以调用系统管理程序。访管原因由访管指令中的8位码指明。这是一种自愿中断,CPU一般不能禁止这类中断。

④ 外部中断:来自主处理机以外的中断事件。它包括各种定时器中断、外部信号中断和中断键中断。其中定时器中断主要用做计时、计费、控制等用途;外部信号中断主要用于与其他机器或系统的联系;中断键中断则用于操作员对机器的干预。

外部中断又可分为两类:一类中断是在没有得到处理机响应时继续保留,而另一类是如果处理机不响应也不再保留。

⑤ 输入输出中断:输入输出操作完成、I/O通道或设备产生故障时发出,用于实现处理机与各种外部设备及通道处理机的联系。

⑥ 重新启动中断:为操作员或另一台处理机启动一种程序所用的。在这六类中断中,只有重新启动中断不论CPU处于停止状态还是处于运行状态都可以发生,其他五类中断只发生在CPU处于运行状态时。CPU不能禁止重新启动中断。

对于这六类中断中的第二至第五类,各有一个16位的中断码,这个中断码用来区分各个具体的中断源。当处理机响应中断后,从硬件形成的入口地址进入各类中断源之后,可以通过这个中断码找到是这类中断中的哪一个中断源发出的中断请求。

在许多机器中还把中断源分为可屏蔽中断和不可屏蔽中断两大类,或称为一般中断和异常中断。当发生不可屏蔽中断时,处理机必须进行处理,不能通过中断屏蔽码进行屏蔽。

在IBM 370系列机中,把执行现行指令引起的中断划分为不可屏蔽中断,如运算结果溢出、页面失效等,这类中断必须及时予以处理,否则程序无法继续正常运行下去或者导致程序运行错误。可屏蔽中断是指那些与当前进程无关的中断事件,如外部设备的输入输出请求、定时器的中断请求等,这些中断可以被屏蔽,中断请求被保留在中断寄存器中不会丢失,当解除屏蔽后,仍然会得到响应和处理。

在异常中断中,有一类称为自陷(Trap),发生在引起异常的指令执行的末尾,经中断处理后返回到这条指令的下一条指令处继续执行原程序。另一类称为故障(Fault),发生在指令的执行过程中,经中断处理后要返回到原先发生故障的那条指令处重复执行。还有一类称为失效(Abort),也发生在执行指令的过程中,但这类中断除非强制干预或系统重新复位,否则机器无法继续正常执行程序。

2.中断的分级管理

中断请求一般是随机发生的,当中断源很多时,经常会发生多个中断源同时发出中断请求的情况,为正确处理和响应中断请求,应该为每一类中断赋予不同的优先级别。中断优先级的确定是一个涉及到计算机系统全局的问题,究竟为每一类中断赋予多高的优先级,应主要考虑以下因素:

① 中断事件的紧急程度。主要是考虑这种事件对整个系统的影响程度,如果影

响到系统的正常工作,优先级应该安排得比较高,而对一些只影响到局部的中断事件,优先级相应可以低一些。如电源故障、总线错、CPU 的地址错、数据错等,它们一旦发生,必须及时处理,否则整个系统都将无法正常运行,所以这些机器检验性错误引起的中断一般要安排在最高优先级。而像程序性错误引起的中断、外部设备引起的中断优先级相应低一些。

② 设备的工作速率。快速设备由于数据存在的时间短,为避免下一个数据产生时上一个数据还未来得及处理,其优先级应该安排得比慢速设备高一些。如在一些常用的外部设备中,优先级从高到低的顺序一般为:实时钟、磁盘存储器(包括软磁盘)、行式打印机、控制台终端输出、控制台键盘输入。

③ 数据恢复的难易程度。数据丢失后无法再恢复的设备,其优先级应高于能自动或手动恢复数据的设备。

④ 要求处理机提供的服务量。如果能够大部分时间独立工作,而较少需要处理机服务的中断事件,其优先级应当高于那些需要处理机提供大量服务的事件。如 DMA 请求只在数据传送前和完成后需要处理机干预,数据传送期间可以独立工作,而主程序则需要处理机连续为它服务。当然这是两个比较极端的例子。

根据以上这些因素,IBM 370 系列机中,把六类不同的中断共分为五级,优先级从高到低的顺序是:紧急的机器校验;访管中断、程序性中断、可抑制的机器校验;外部中断;输入输出中断;重新启动中断。

紧急的机器校验放在第一级是因为像发生掉电、数据通路错等设备故障时,影响到整个系统的正常运转。而像某个外部设备发生故障,优先级别可相应低一些。

访管中断、程序性中断、可抑制的机器校验这三类中断请求是不会同时发生的,把它们放在第二级。访管中断放在第二级是因为它是一种自愿中断,机器在执行“访管”指令时,如果未发生紧急的机器校验错误,则进入相应的管理程序。其中断响应不是靠中断级的排队电路,而是靠机器执行“访管”指令进入的,所以任一级的中断屏蔽都不能屏蔽这类中断,即它不受中断级屏蔽码的控制,这点可使各级中断中都可使用“访管”指令,嵌套进入相应的管理程序,从而给系统程序的编制带来方便。

为防止在处理外部中断和输入输出中断的过程中,由于中断处理程序本身执行过程中引起的程序性错误,产生的程序中断和主程序中原先的程序性中断混在一起,引起中断混乱,因此,程序性中断的优先级别应高于外部和输入输出中断。

外部中断由于牵涉到多机联系、人机交互,而输入输出操作是由各通道管理,中断响应晚些也不会丢失信息和带来太大的影响,和外部中断相比属于局部性问题,所以优先级安排得比外部中断要低。

重新启动中断级别一般都安排得最低,因为重新启动并不是很紧迫的事件,但是当 CPU 处于停止状态时,重新启动中断的级别一般最高。

有的计算机系统中往往还设置有 0 级中断。这是在机器因故障重叠发生或无法排除的情况下,系统完全不能正常工作,由中断系统硬件发出的机器告急中断。它或

者向操作人员报警,请求干预,或者向其他处理机发出求援,进行机器间的任务切换。这种告急状态也使机器所执行的程序被中断,但它不参加中断级排队,中断发生后也无法自行恢复,它不是真正的中断级。

3.3.2 中断系统软、硬件功能分配

中断系统的功能包括中断请求的保存和清除、优先级的确定、中断现场的保护和恢复、中断请求的分析和处理、中断返回等,这些全是由中断响应硬件和中断处理软件共同完成的。其中,有些功能必须用硬件来实现,有些功能必须用软件来实现,而中间的大部分功能用硬件或软件实现都可以,或者软、硬结合共同来实现。因此,在设计中断系统时,如何恰当地进行软、硬取舍,是设计好中断系统的最关键的一个问题。

中断响应时间和中断系统的灵活性是中断系统设计好坏的两个重要性能指标,在处理中断时如何进行软、硬功能分配就要考虑这两方面的因素。

中断响应时间指的是从某一个中断源发出中断请求到处理机响应这个中断源的中断请求,并进入中断处理程序所需的时间。

中断系统的这两个性能指标是相互矛盾的,这是因为一般情况下,用硬件实现的功能越多,响应时间越快,但灵活性越差;用软件实现的功能越多,灵活性越好,但中断响应时间势必要增加。

不同的机器在解决这个问题时的处理方式上存在很大的差别。但是,其中断处理的过程大体上是相同的,这个过程可简单地归纳为中断响应、中断识别、保存现场、中断服务、恢复现场和中断返回等,如图 3 .16 所示。

(1)中断响应

CPU 在执行完一条指令后,如果允许中断,就进入中断周期。在这个时间段内,CPU 自动关闭中断允许触发器,即关中断,禁止响应任何中断请求,因下面一段工作是不能被打断的:

① 保存断点:通常是将程序计数器 PC 中的内容压入系统堆栈中,以便在中断服务完成后能正确地返回到原来的程序中去。

② 撤消中断请求:撤消对应的中断服务请求,这是为保证在下一次开中断后不再响应此中断申请。

③ 保存硬件现场:主要指将处理机状态字 PSW 及堆栈指针 SP 中的内容压入系统堆栈或主存储器的指定单元中。

以上这些操作犹如 CPU 执行一条指令一样,功能固定不变,但实际上 CPU 不是在执行机器指令,称这些操作为执行中断隐指令,在机器指令系统中是找不到此类指令的,它是机器进入中断周期后自动执行的,不管是由哪一个中断源引起的中断,只要进入响应,这些操作必须进行,因此称这些操作为公操作。

(2)中断识别:这个阶段主要是识别中断源,并转向对应的中断服务程序入口。

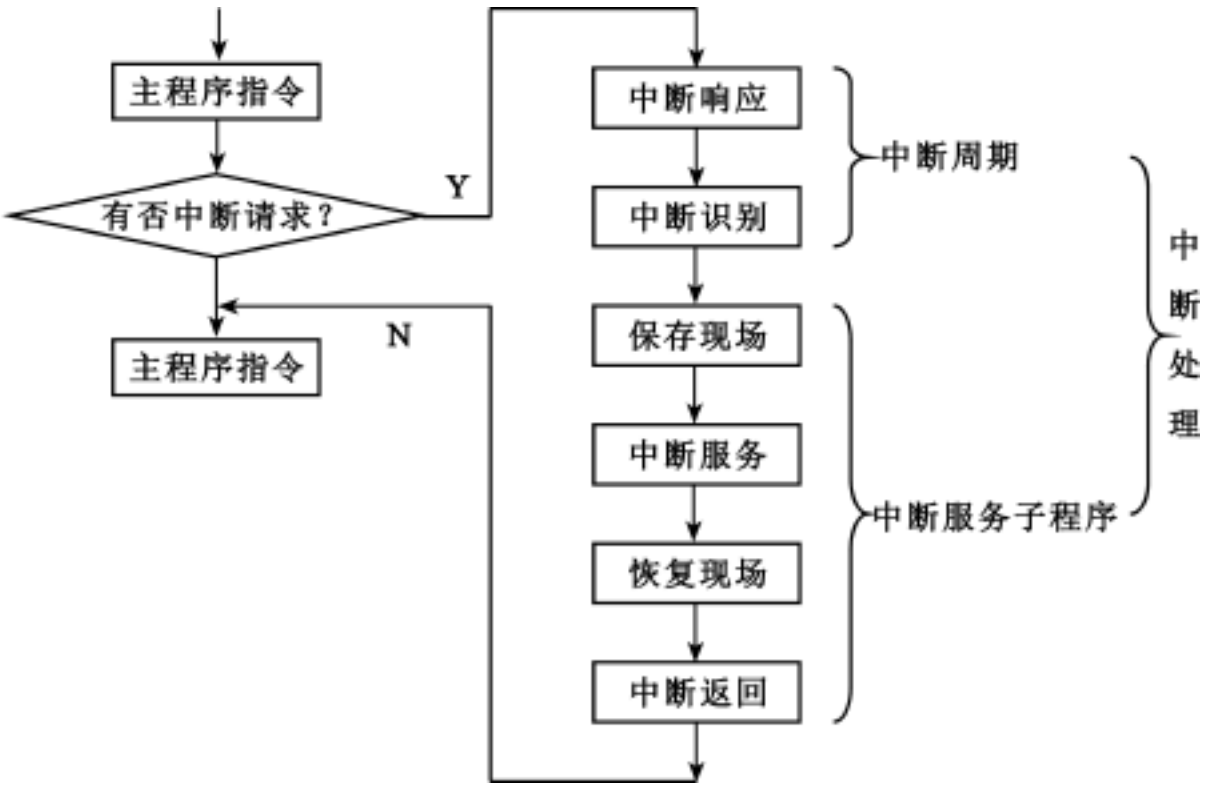


图 3.16　中断处理过程

其中识别中断源的方法常用的有程序查询法、串行中断源排队链法、中断向量法等。程序查询法是软件识别方法，在中断响应结束时，CPU 自动从固定地址单元中取出指令，并执行这些指令，用于中断识别。串行中断源排队链法是通过硬件排队线路来完成中断源的排队，类似于总线的串行链接控制方式。中断向量法实际上是在主存储器的固定区域中开辟出一个专用的中断向量区，保存对应中断事件的 PC 和 PSW（PC 和 PSW 合称为中断向量，它们在主存中的第一个单元地址称为向量地址），用硬件排队器和编码器在所有请求中断服务的中断源中，产生具有最高优先级的中断源编号，直接通过硬件将这个中断源向量地址送往 CPU，实现中断服务程序入口地址的最快转移。所以这是一种识别中断源速度更快，使用也更为广泛的方法。

经过中断识别引出中断服务子程序入口地址，中断周期结束，就进入了中断服务子程序。它一般包括三个部分，如图 3.17 所示。

前处理的工作主要是：

① 保护现场：除了 PC 和 PSW 外，如果在中断服务子程序中还要使用其他通用寄存器，则必须把这部分将被破坏的通用寄存器中的内容压入系统堆栈或切换到另外一组通用寄存器中进行保存。

② 交换屏蔽字：一般是为了屏蔽掉与本设备相同或比本设备低的中断请求，这样，在执行中断处理程序的过程中，可允许高级中断打断低级中断的执行，实现中断嵌套。

③ 开中断：在中断响应时，CPU 内的中断允许触发器是自动关闭的，其目的是在替换新老屏蔽字和保护现场时禁止一切中断，以免引起 CPU 现场混乱。所以为

了 CPU 可以响应其他更高级中断源的中断服务请求,实现中断嵌套,在前处理的最后一条指令是开中断,它将中断允许触发器置 1。

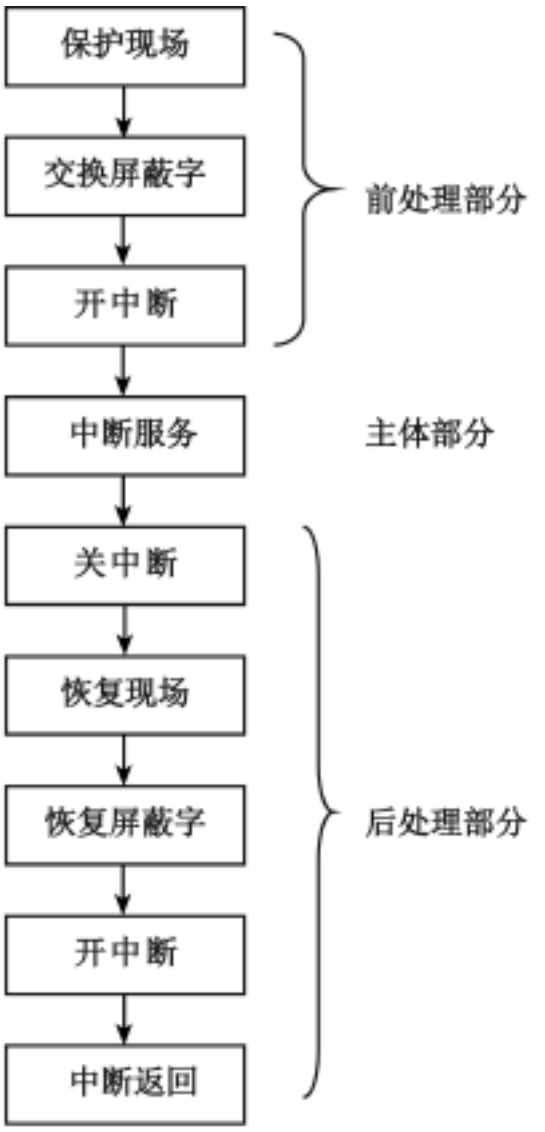


图 3 .17 中断服务子程序

中断服务程序的主体部分就是真正要进行的中断处理。有的是进行数据传送,有的是检查设备,有的是数据传送完毕后的结束处理。根据不同情况,主体部分可以是一条指令,也可以是一段程序。

后处理要进行的工作有:

① 关中断:在下一次开 CPU 中断前,正在运行的程序不允许被中断。

② 恢复现场:恢复在前处理过程中保存的通用寄存器的内容等。

③ 恢复屏蔽字:恢复在中断前的屏蔽状态。

④ 开中断:开放中断指令中的开中断操作在硬件上延时到下一条指令执行时才完成,这样可避免断点地址的混乱。

⑤ 中断返回:返回到中断点,这是中断服务程序的最后一条指令。该指令将压入堆栈中的原 PC 送回相应的寄存器,原程序从断点开始又继续执行下去。至此,中

断处理过程全部结束。

上述的中断处理过程是一个典型的过程,但并不是所有的机器都相同,各种机器的设计方案和硬件结构不同,具体的中断处理过程可能各不相同。但在中断处理过程中,只有中断响应阶段的“是否响应中断”、“关中断”和“保存断点”、中断识别阶段的“转向中断服务程序入口”这些功能是必须用硬件来实现的,同样,也只有中断服务子程序中的“主体部分”和“返回到中断点”这两个功能必须用软件实现,其他的所有功能是既可以用硬件实现,也可以用软件实现,或软硬相结合实现。这就需考虑到中断响应时间和中断系统的灵活性。

在中断系统的设计中,目前一般是将中断响应和中断识别阶段的所有功能用硬件实现,而中断服务子程序阶段的所有功能用软件实现。但中断响应阶段的保存硬件现场,如果为了节省硬件,这些与当前运行程序有关的硬件现场也可以采用软件在中断服务程序中加以保存和恢复,而在中断服务程序中的“保存现场”,把它称为软件现场。对于软件现场,大多数机器都采用软件来保存和恢复。为了加快保存和恢复软件现场的过程,在一般机器中都设置有成组存和成组取的指令,也有些机器采用硬件来保存软件现场,如前面所讲的 RISC 中的重叠寄存器窗口技术。

以上所讲的是针对所有任务都在同一个处理机上实现的集中式处理机系统而言的,对于 I/O 处理机可以最终解除 I/O 中断,所以这种分布处理的设计越来越普遍地被接受。

3.3.3 中断响应与中断屏蔽

为了缩短中断响应时间,现在的处理机一般都用硬件来完成中断请求的优先级排队,如串行排队链法、中断向量法等,这些方法识别中断源的速度很快。但是,由于中断源的中断优先级是由硬件固定的,不能由程序员通过软件来改变,因此灵活性较差。

正如前面所述,一般在处理某级中断请求的过程中,为实现中断嵌套,允许高级中断打断低级中断的执行,禁止同级中断或低级中断打断中断服务程序的执行,这样,中断实际处理完的次序是可以不同于中断的响应次序的。

根据需要,由操作系统管理软件控制改变实际的中断处理次序,很多机器都设置了中断级屏蔽位寄存器硬件,以决定是否让某级中断请求进入中断响应排队器排队。只有进入排队线路,才能让级别高的中断请求优先得到响应。中断屏蔽位可以分布在各个中断源中,也可以集中在处理机中,例如,放在处理机的程序状态字中,这是一种普遍采用的方法。如 IBM 370 系列机就采用了这种方法。如在程序状态字中设置有中断屏蔽位字段,只要操作系统对每一类中断处理程序的现行程序状态字中的中断级屏蔽位设置成不同状态,就可以实现所希望的中断处理次序。

例 3.2 假设某中断系统中共设置有四级中断 D1,D2,D3,D4,它们的中断优先级从高到低分别是 1 级、2 级、3 级、4 级。相应地每一级中断处理程序的现行程序状

态字中都设有 4 位中断级屏蔽位。如果中断级屏蔽位为“1”,表示对该级的各个中断请求都开放,允许其进入中断响应排队器进行排队;若为“0”,则表示对该级的各个中断请求都屏蔽,不让其进入中断响应排队器进行排队。正常状态下的中断屏蔽码和改变后的中断屏蔽码,如表 3.2 所示。

现假设在运行用户程序的过程中同时出现了四级中断请求,则正常中断屏蔽码状态下的实际中断处理次序如图 3.18 所示。由于用户程序是不能屏蔽任何中断请求的,所以执行用户程序时其现行 PSW 中的中断级屏蔽位均为“1”,当 D1,D2,D3,D4 四级中断请求同时到来时,均进入排队器进行排队,于是优先响应 D1 级中断请求,通过交换新旧 PSW 实现程序切换。此时 D1 级对应的中断屏蔽级屏蔽位“0000”被放置到中断屏蔽寄存器,这样,如果有 D2,D3,D4 级中断请求的话都将被屏蔽,不予响应。执行完 D1 级中断服务程序,交换 PSW,又返回到原被中断前的用户程序,此时,由于作为新 PSW 的用户程序状态字中的中断级屏蔽位全为“1”,所以,D2,D3,D4 中断请求才又进入排队器排队。同样的道理,优先响应 D2 级中断请求并进行处理,之后又返回到被中断前的用户程序,此时还有 D3 和 D4 级中断请求未被响应,进入排队器排队,再优先响应 D3 级中断请求并进行处理,之后又返回到被中断前的用户程序,只剩下 D4 级中断请求还未被响应,待响应 D4 中断请求并处理完后又返回到被中断前的用户程序继续向下执行。

表 3.2　　四个中断源的中断优先级和中断屏蔽码

中断源名称	中断优先级	正常中断屏蔽码				改变后的中断屏蔽码			
		D1	D2	D3	D4	D1	D2	D3	D4
D1	1	0	0	0	0	1	1	1	0
D2	2	1	0	0	0	1	1	0	0
D3	3	1	1	0	0	1	0	0	0
D4	4	1	1	1	0	0	0	0	0

由此可见,在正常的中断屏蔽码状态下,当某一个中断优先级的中断源占有处理机,正在执行它的中断服务程序时,只有更高一级中断级别的中断源的中断服务请求才能中断这个中断服务程序的执行,同一级的和比它低级的所有中断源的中断请求都被屏蔽掉,即不能中断当前的中断服务程序。这种情况下,中断响应的次序和实际处理完的中断服务程序的次序是完全一致的。

如果采用改变后的中断屏蔽码,处理机实际为各个中断请求服务的先后次序就会发生改变,如图 3.19 所示。

当处理机正在执行用户程序时,如果同时出现了四级中断请求,则仍然是优先响应 D1 级中断请求,但 D1 级中断服务程序的 PSW 中设置的中断级屏蔽位为

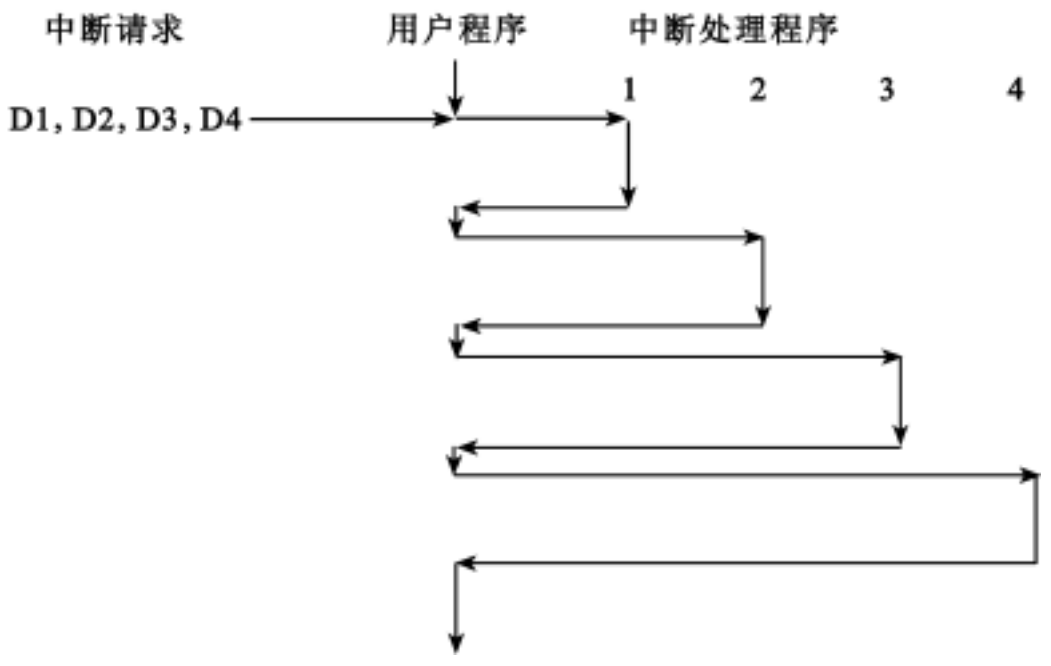


图 3 .18　正常中断屏蔽码状态下的中断处理次序

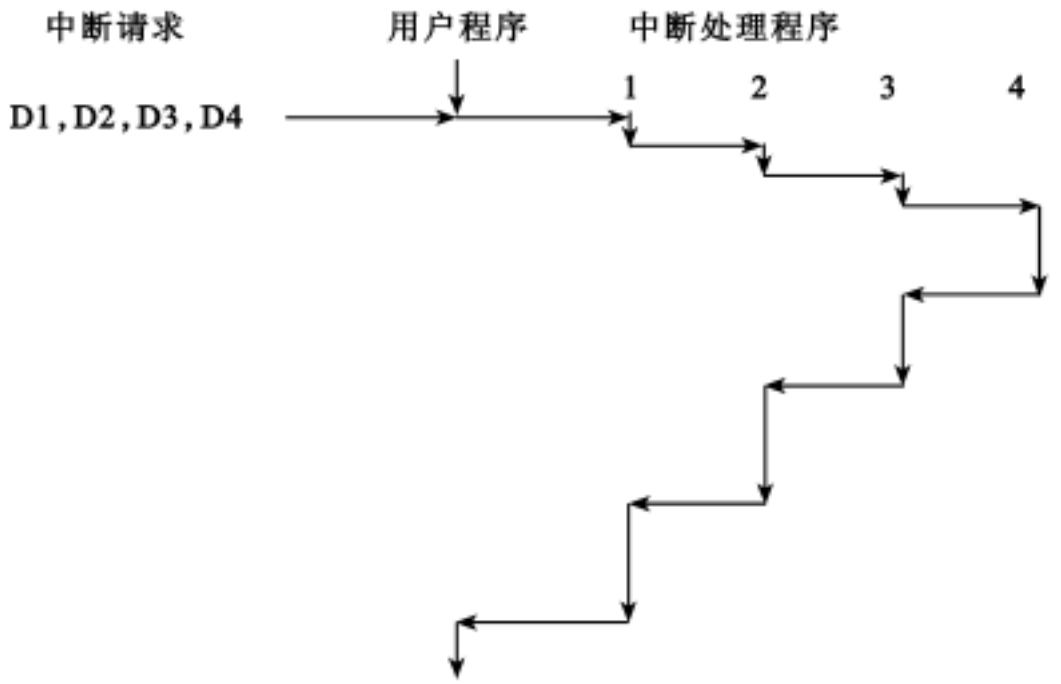


图 3 .19　按照改变后的中断屏蔽码处理机响应中断的次序和实际的中断处理次序

“1110”,所以仍然允许 D2,D3,D4 进入中断响应排队器进行排队,优先响应 D2 级中断请求。同样的道理,从 D2 级进入 D3 级,又从 D3 级进入 D4 级,D4 级中断服务程序的 PSW 对应的中断级屏蔽位为“0000”,不允许任何中断请求打断它的执行。当 D4 级中断服务程序执行完后,依次返回,执行 D3、D2、D1 级中断服务程序。待 D1 级中断服务程序执行完后,返回用户程序,继续往下执行。可见,在这种改变了的中断屏蔽码状态下,中断响应的次序仍然是 D1→D2→D3→D4,但实际中断处理的次序

却是 D4→D3→D2→D1,与中断响应的次序刚好相反。因此,通过设置中断屏蔽码,可以通过软件,由系统管理员任意改变中断源的中断服务顺序,提高中断系统的灵活性,这就是中断系统采用软、硬结合带来的一个好处。

3.3.4 Pentium 系列微计算机的中断系统

1. Pentium 的中断类型

Pentium 有两类中断源,即中断和异常。

(1)中断

通常称为外部中断,它是由 CPU 的外部硬件信号引发的,有两种情况:

① 可屏蔽中断:CPU 的 INTR 引脚收到中断请求信号后,如果 CPU 中标志寄存器 IF = 1 时,可引发中断;IF = 0 时,中断请求信号在 CPU 内部被禁止。这类中断主要是指诸如键盘、鼠标、磁盘、打印机等这类外部设备产生的中断请求,故也称为外部硬件中断。

② 非屏蔽中断:CPU 的 NMI 引脚收到的中断请求信号而引发的中断,这类中断不能被禁止。非屏蔽中断实际上是另一类由外部硬件引发的中断,用于处理必须立即响应的外部事件,如电源故障和存储器读出现奇偶错。

(2)异常

通常称为异常中断,它是由指令执行引发的,有两种情况:

① 软件中断:在 Pentium 处理器指令系统中,包括一些如 INT0,INT3,INT n 和 BOUND 这类的软件中断指令,这些软件中断指令在执行时,立即启动相应的中断服务例程,它们的中断类型号是固定的,不需要以中断识别总线周期由外部来读取。

② 内部中断和例外:CPU 在执行一条指令的过程中出现错误、故障等不正常条件引发的中断;它是自动被测试的,不受 IF 中断允许标志位的影响,而且,它的中断类型号是固定的,中断处理功能也是约定好的。表 3.3 列出了 Pentium 的中断类型,我们看到,类型号 0～31 中除类型 2～5 外,都用于(或保留)内部中断和例外。

对于这些内部中断和例外,基于出错位置的处理,可大致将这些内部中断和例外分成故障、陷阱和中止三类。故障类是错误出现在指令完成之前,在保护中断现场的过程中,将造成故障的指令的 CS:EIP 压入堆栈保存,于是在中断服务后恢复中断现场时,此发生故障的指令得以重新执行。陷阱类是错误出现在指令完成之后,保护中断现场的过程将造成故障的指令下一条指令的 CS:EIP 压入堆栈保存,于是中断返回后执行的是形成陷阱指令的下一条指令。中止类是不保存有关错误发生位置的任何消息,一般是严重的系统故障,如硬件故障、无效的系统表等,往往需要重新启动系统。

总之,Pentium 的中断系统详细分类共有 256 种中断和异常。每种中断给予一个编号,称为中断向量号(0～255),以便发生中断时,程序转向相应的中断服务子程序入口地址。

当有一个以上的异常或中断发生时,CPU 以一个预先确定的优先顺序为它们先后进行服务。中断优先级分为 5 级。异常中断的优先级高于外部中断的优先级,这是因为异常中断发生在取一条指令或译码一条指令或执行一条指令时出现故障的情况下,情况更为紧急。处理器以如下的顺序先后为它们服务:

- 内部中断和例外;
- 软件中断;
- 不可屏蔽中断;
- 外部硬件中断。

表 3.3　　Pentium 的中断类型

类型号	类型名	描　　述
0	除法错	执行除法指令时,发现除数为 0
1	单步中断	调试时设置的陷阱和故障
2	非屏蔽中断	NMI 引脚信号引发的硬件中断
3	断点中断	被单字节 INT3 指令引发
4	溢出中断	执行 INTO 指令并且 OF 标志位置位
5	越界中断	执行 BOUND 指令并且操作数越过数组边界
6	操作码无意义	执行的指令,其操作码是未定义的
7	设备无效	执行 ESC 或 WAIT 指令时由于设备不存在而失败
8	双重故障	执行同一指令时出现两次中断并且不能串行处理
9	(保留)	
10	无效的 TSS	任务切换时因任务的 TSS 不正确而发生故障
11	段不存在	因段描述符中的 P 位为 0 而引起
12	堆栈故障	访问堆栈越界或堆栈段不存在
13	一般保护错误	不引起另外异常的一般保护违约
14	页故障中断	分页模式下的缺页中断

15	(保留)	

续表

类型号	类型名	描　　述
16	浮点错误	执行浮点算术指令时出错
17	对准错误	访问地址未对准
18	机器检查错	总线周期不能成功完成或读数出现奇偶错
19～31	(保留)	
32～255	用户中断	其中一些被 INTR 信号有效时引发

2 .Pentium 的中断处理

中断服务子程序的入口地址信息存于中断向量号检索表内。实模式为中断向量表 IVT,保护模式为中断描述符表 IDT。CPU 识别中断类型取得中断向量号的途径有三种:

① 指令给出,如软件中断指令 INT n 中的 n 即为中断向量号。

② 外部提供,可屏蔽中断是在 CPU 接收到 INTR 信号时产生一个中断识别周期,接收外部中断控制器由数据总线送来的中断向量号;非屏蔽中断是在接收到 NMI 信号时中断向量号固定为 2。

③ CPU 识别错误、故障现象,根据异常和中断产生的条件自动指定向量号。

CPU 依据中断向量号获取中断服务子程序入口地址,但在实模式和保护模式下采用不同的途径。

(1)实模式下使用中断向量表

中断向量表 IVT(Interrupt Vector Table)位于内存地址 0 开始的 1KB 空间。实模式是 16 位寻址,中断服务子程序入口地址(段,偏移)的段寄存器和段内偏移量各为 16 位。它们直接登记在 IVT 表中,每个中断向量号对应一个中断服务子程序入口地址。每个入口地址占 4 字节。256 个中断向量号共占 1KB。CPU 取得向量号后自动乘以 4,作为访问 IVT 的偏移,读取 IVT 相应表项,将段地址和偏移量设置到 CS 和 IP 寄存器,从而进入相应的中断服务子程序,进入过程如图 3 20(a)所示。

(2)保护模式下使用中断描述符表

保护模式为 32 位寻址。中断描述符表 IDT(Interrupt Descriptor Table)也是每一表项对应一个中断向量号,表项称为中断门描述符、陷阱门描述符。这些门描述符为 8 字节长,对应 256 个中断向量号,IDT 表长为 2KB。由中断描述符表寄存器 IDTR 来指示 IDT 的内存地址。

以中断向量号乘以 8 作为访问 IDT 的偏移,若读取的表项是中断门/ 陷阱门的

描述符,则门描述符给出中断服务子程序入口地址(段,偏移),其中 32 位偏移量装入 EIP 寄存器,16 位的段值装入 CS 寄存器。依选择符中的 TI 位为 0 或 1,由全局描述符表 GDT 或局部描述符表 LDT 中读取此选择符所对应的段描述符,自动加载到 CS 描述符高速缓存器中。由此缓存器中的基地址字段来确定中断服务子程序入口的段位置,再结合 EIP,即为此中断服务子程序的入口地址。保护模式下入中断服务子程序的过程如图 3 .20(b)所示。

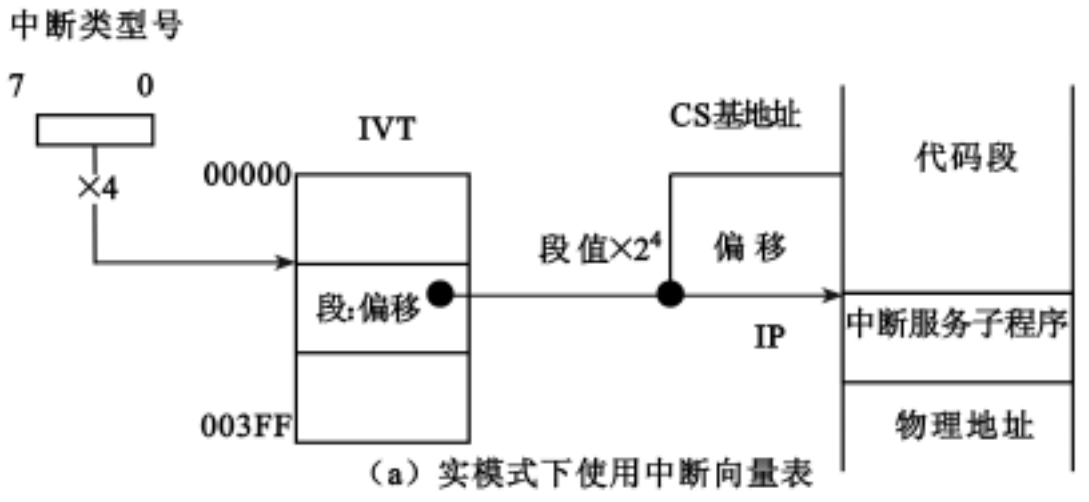


图 3 .20　中断服务子程序的进入过程

中断门和陷阱门只能设置在 IDT 内的门。两类门描述符的结构差异只在 b40 位上,中断门为 0,陷阱门为 1;功能上的差异是:中断处理通过中断门时 EFLAGS 的 IF 位被清除,而陷阱门不改变 IF 位的状态。因此,中断门是面向外部硬件中断的门,这种中断处理要求时间短,优先权大。陷阱门由于不改变 IF 位的状态,即中断处

理过程可被外部硬件中断打断,故是面向优先权小的中断。

若由 IDT 读取的表项是任务门描述符,则给出的是任务状态段 TSS(Task Status Segment)的选择符。此选择符被装入任务寄存器 TR,并以此选择符检索 GDT,取得 TSS 的描述符装入 TR 的描述符高速缓存器中。此描述符给出 TSS 的基址、段限和属性。使用任务门进行中断处理时,被中断任务的环境进栈和中断任务环境的加载全部是自动进行。它的好处是中断任务完全与被中断任务隔离,缺点是中断响应过程耗时较长。中断类型 8(双重故障)和 10(无效 TSS)必须使用任务门来进行中断处理。

当由上述两种方式获取中断服务子程序入口地址,并且处理器决定响应本次中断/ 例外请求时,依次完成下列事情:

① 当中断处理的 CPU 控制权转移涉及到特权级改变时,必须把当前的 SS 和 ESP 两个寄存器的内容压入系统堆栈予以保存。

② 标志寄存器 EFLAGS 的内容也压入堆栈。

③ 清除标志触发器 TF 和 IF(有些情况下 IF 不清除)。

④ 当前的代码段寄存器 CS 和指令指针 EIP 也压入此堆栈。

⑤ 如果中断发生伴随有错误码,则错误码也压入此堆栈。

⑥ 完成上述中断现场保护后,从中断向量号获取的中断服务子程序入口地址(段,偏移)分别装入 CS 和 EIP,开始执行中断服务子程序。

⑦ 中断服务子程序最后的 IRET 指令使中断返回。保存在堆栈中的中断现场信息被恢复,并由中断点继续执行原程序。

3.3.5 APIC 技术简介

从 8086 到 80486,处理硬件可屏蔽中断时,处理器都需要可编程中断控制器 PIC(Programmable Interrupt Controller)8259A 芯片的支持与配合。Pentium 处理器从 P54C 开始,就在处理器芯片内集成了可编程中断控制功能。若外部再有一个 I/O 模块配合,就可构成一个高级的可编程中断控制 APIC(Advanced PIC)子系统。APIC 技术主要是面向多处理环境的,因此直到 Pentium Pro,Pentium Ⅱ/ Ⅲ处理器的出现,才明显展现出它的优越性能和重要意义。在这里,重点介绍 APIC 的基本概念和工作模式。

1. APIC 技术的基本概念

高级可编程中断控制 APIC 子系统由三部分组成:Local APIC,I/O APIC 及 APIC 总线。其中,Local APIC 是集成到处理器内部的一个模块,从 Pentium P54C 开始一直到 Pentium Ⅱ/ Ⅲ都是如此。因此,只要系统板上安装 I/O APIC 模块以及设置 APIC 总线,就可实现 APIC 技术。

图 3.21 给出一个由双 Pentium Ⅱ处理器组成的系统结构框图。系统的处理器

总线、PCI 总线和 ISA 总线使用 Intel 440BX 芯片组相互连接、沟通,使系统成为一个整体。Intel 440BX 芯片组中的南桥芯片 PIIX4E 接有 I/O APIC 模块。PIIX4E 芯片将它所连接的 PCI 设备、IDE 设备、ISA 设备、串行设备等所有 I/O 设备的中断请求信号送到 I/O APIC 模块。

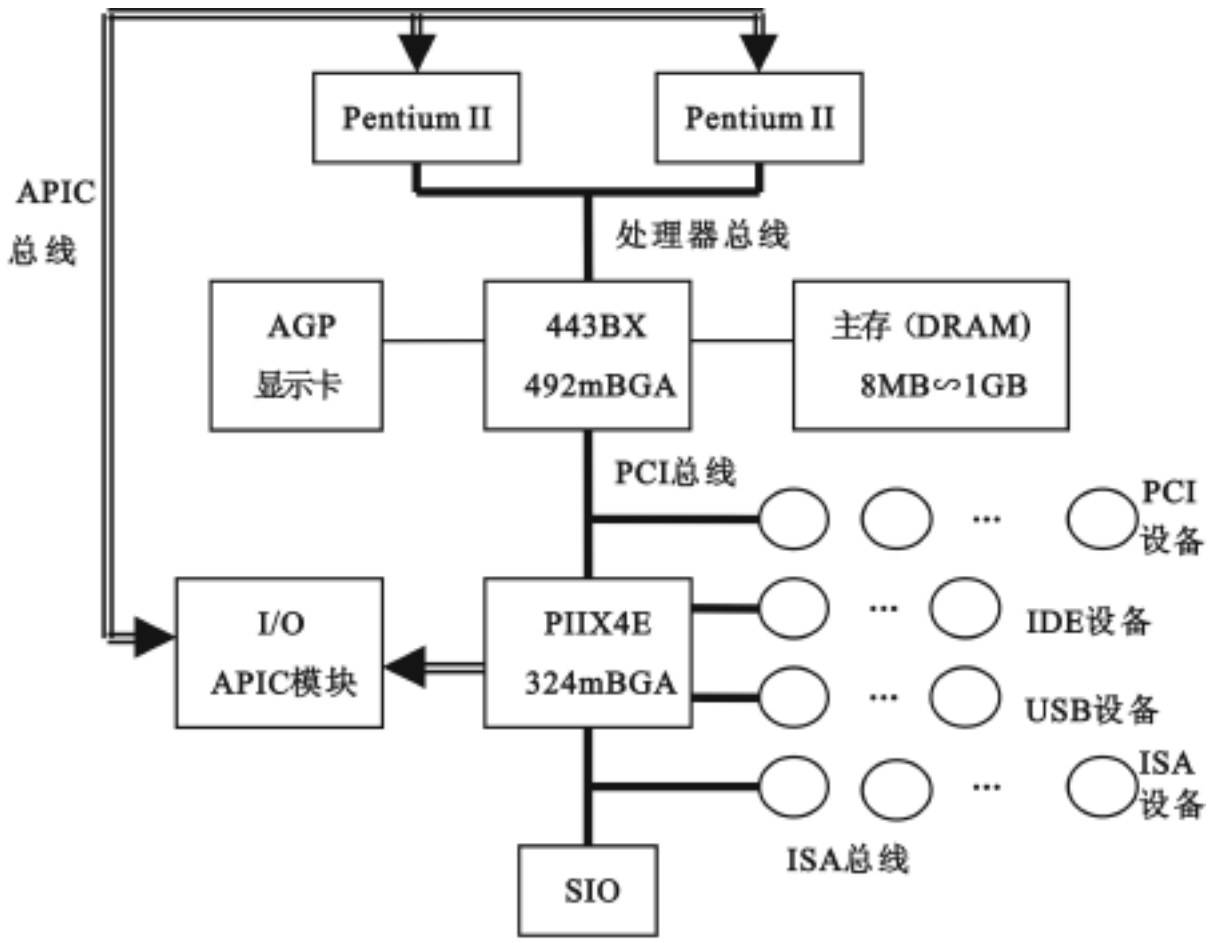


图 3.21 使用 440BX 芯片组的典型 Pentium PC 系统结构

APIC 面向多处理机,既支持对称的也支持非对称的多处理环境。而 8259PIC 只能用于在非对称的多处理环境中,即 8259 的输出只能送到运行操作系统软件、应用程序和处理中断服务的主处理器上,其他处理器完成文件管理、LAN 等 I/O 任务。APIC 技术保持了与 8259 PIC 的兼容性。不是多处理环境的工作站或服务器可以不必设置 APIC,即使置备了也可设置成"绕过"方式予以不用。当前的个人计算机大都没采用 APIC 技术。

I/O APIC 模块有 16 个中断请求输入引脚 INTIN0～INTIN15,经这些引脚送入的中断请求被称为系统中断请求,以强调它们是被系统中所有处理器所共享的。而把"外部硬件中断"(Ext INT)一术语留作专指 8259 PIC 所产生的(或与之兼容的)中断。

APIC 总线由三根线组成,它们是 PICCLK(PIC 时钟)信号线、PICD0(PIC 数据 0)和 PICD1(PIC 数据 1)线。以串行同步方式经 PIC 总线传输的信息,可以是系统中断请求、EOI 中断结束命令、处理器间中断(IPI)请求等。取决于传输信息类型的

不同,一次 APIC 总线传输可由 14～39 个 PICCLK 周期组成。PICCLK 方波信号起着同步作用,数据沿 PICD1 和 PICD0 线先高位后低位的串行传送,在每个 PICCLK 周期 PICD1 是高位、PICD0 是低位。

每个处理器内集成有 Local APIC 模块。此模块内有中断请求寄存器(IRR)、中断服务寄存器(ISR)、中断结束寄存器(EOI),以及中断优先权判决所需要的任务优先权寄存器(TPR)、处理器优先权寄存器(PPR)、仲裁优先权寄存器(APR)等。此外,Local APIC 模块中还有初始计数寄存器(ICR)、当前计数寄存器(CCR)、定时器除法配置寄存器(TDC)。由此可见,处理器内集成的 Local APIC 模块已涵盖了 PC/AT 机系统中的 8259A 芯片和 8254(定时器/计数器)芯片的功能。Local APIC 模块除接收并响应 I/O APIC 模块送来的中断请求,以及经 APIC 总线发送或响应处理器间中断(IPI)之外,还能处理本地产生的中断,这包括处理器内部产生的定时中断、错误中断、性能计数中断,和经 LINT0/INTR,LINT1/NMI 引脚直接连接到处理器的外部设备所产生的中断。图 3.22 给出了 APIC 子系统组成的结构框图。

由上可知,APIC 子系统能够处理如下三种基本类型的中断:

① 系统中断:这是指 I/O 设备经 I/O APIC 模块的 INTIN0～INTIN15 引脚送入的中断请求。I/O APIC 模块然后将这些系统中断请求提交给指定的目标处理器进行中断处理。如系统中的 PCI 设备、IDE 设备、ISA 设备都可将中断请求接至 INTIN0～INTIN15 输入到 I/O APIC 模块,产生系统中断请求。

② 本地中断:这是指 I/O 设备直接连到处理器的 LINT0、LINT1 引脚所产生的中断请求,以及处理器内部产生的定时器中断、错误中断和性能计数器中断(最后一项是 Pentium Ⅱ处理器才增加的)。本地中断直接由处理器的 Local APIC 模块处理,不使用 APIC 总线传输信息。每个 Local APIC 模块能处理 5 种本地中断:

- 本地定时器中断;
- 本地中断脚 0(LINT0)中断;
- 本地中断脚 1(LINT1)中断;
- 错误中断;
- 性能计数器中断。

其中,本地定时器中断功能是 Pentium 系列处理器将 8253/8254 定时器的基本功能集成到 Local APIC 模块中。本地错误中断是指在系统中断和处理器间中断使用 APIC 总线提交期间,Local APIC 测出错误而引发的中断。如非法寄存器地址、接收或发送非法向量、接收或发送认可错等。本地性能计数器中断用于处理器的性能监督。

③ 处理器间中断(IPI):这是指在软件控制下,以向处理器内的 Local APIC 模块中的专门寄存器写操作,可将消息以及 SMI(系统管理中断)、NMI 等中断请求,经 APIC总线递交给其他处理器,这是对多处理系统中各处理器协调完成启动初始化的有力支持。它包括了七种类型的中断,即:电平撤除的初始化消息、初始化消息、启动

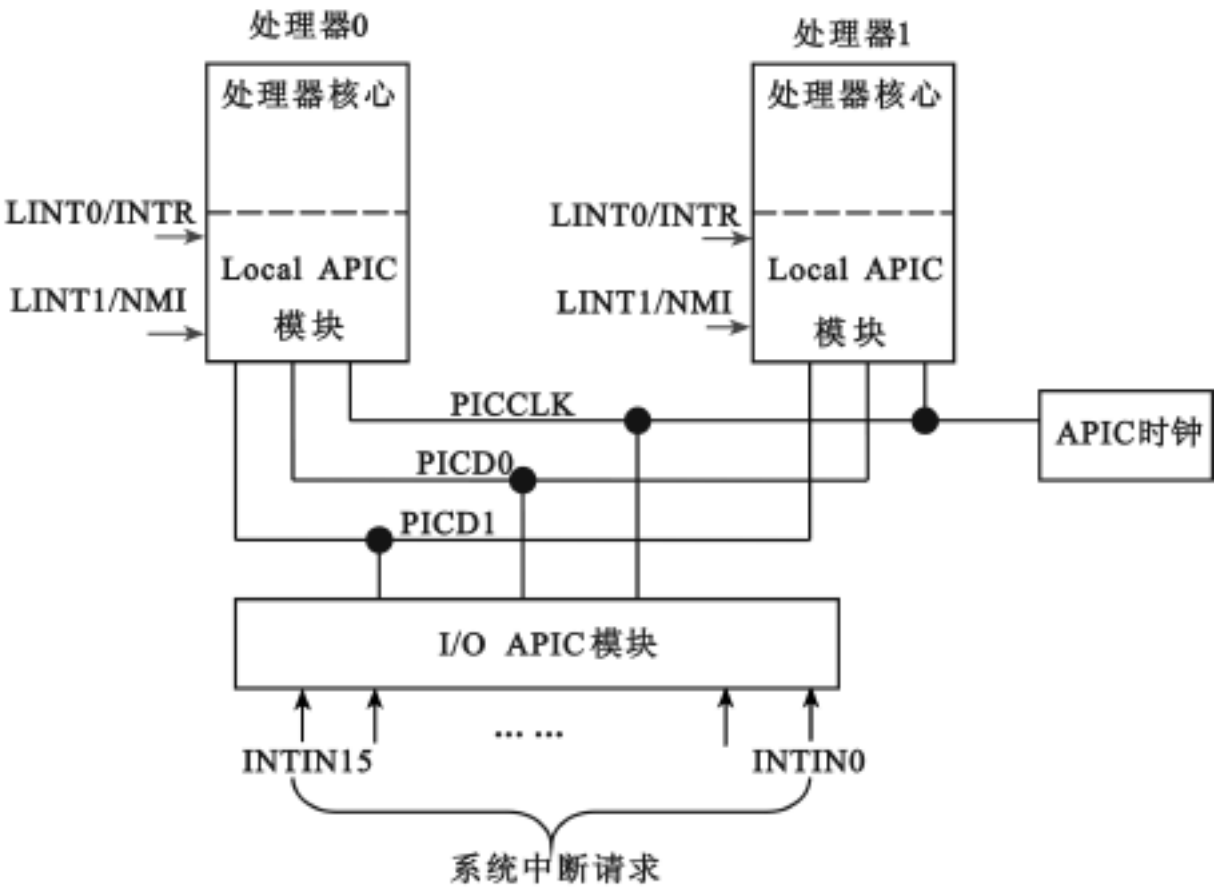


图 3.22 APIC 子系统的组成

消息、远程读消息、系统管理中断请求(SMI)、不可屏蔽中断请求(NMI)和标准的中断请求。

APIC 子系统还支持不可屏蔽中断 NMI 以及 8259PIC 产生的可屏蔽中断请求(INT),只不过 8259 兼容中断在 APIC 子系统中被称为外部硬件中断 ExtINT。在 APIC 子系统中,NMI 可以系统中断、本地中断或处理器间中断形式出现,而 ExtINT 只可以出现在系统中断或本地中断中。

总之,APIC 支持的中断类型很多,它处理中断的过程基本上可分为三个阶段(以系统中断请求处理过程为例来说明):

① 系统中断请求产生,并由 I/O APIC 模块经 APIC 总线递交给某处理器的 Local APIC 模块。

② Local APIC 模块接收中断请求,并对中断优先权进行判决,准许之后将中断请求传送给处理器核心并开始中断服务。

③ 中断服务过程中,处理器清除中断请求设备内部的中断未决位,并在中断返回之前处理器写中断结束命令(EOI),然后由它的 Local APIC 将 EOI 消息经由 APIC 总线传送给 I/O APIC 模块。

2. APIC 工作模式

前面已对 APIC 子系统的组成、工作过程以及中断类型有了初步了解,本节对

APIC的工作模式再作简单的介绍,这包括 APIC 总线仲裁、目标模式、递交模式和触发模式。

(1) APIC 总线仲裁

由图 3.22 看出,APIC 总线连接着系统中各处理器的 Local APIC 模块和系统的 I/O APIC 模块,它们中可能会有多个同时启动 APIC 总线传输中断请求或消息。因此,APIC 总线必须有仲裁机制,以使只能有一个 APIC 模块取得 APIC 总线占用权完成中断请求或消息的发送。而中断请求或消息的接收者允许同时有多个,这由目标模式和递交模式所确定。

APIC 总线仲裁是一种分布式仲裁。仲裁过程在总线传输的最初 5 个 PICCLK 时钟周期内完成,竞争的得胜者将在 PICCLK 时钟周期 6 开始有效信息的发送,直到本次传输结束放弃 APIC 总线占用权。

最初的 5 个 PICCLK 时钟周期构成的总线仲裁周期,包括两类判决:一类是传输类型,另一类是 ID 竞争仲裁。传输类型的判决发生在总线仲裁周期的第一个 PICCLK 时钟周期内。传输类型只有两种:EOI 消息(PICD1,PICD0 = 11b);包括所有其他消息及中断请求的"正常"传输类(PICD1、PICD0 = 01b)。EOI 消息的发送优先于"正常"类的发送。

两个或更多的同类消息发送者,顺利通过第一个 PICCLK 时钟周期后,进入 ID 竞争仲裁的 4 个 PICCLK 时钟周期。APIC 子系统的 I/O APIC 模块和各个 Local APIC模块中,都有一个仲裁 ID 寄存器,容纳有当前的本模块仲裁 ID 值(ID_3 ～ID_0)。仲裁 ID 值的最大值,在经过 4 个 PICCLK 的竞争后将拥有总线占用权。

仲裁 ID 最初来自模块的 APIC ID。I/O APIC 模块和各 Local APIC 模块中各有一个 4 位的 APIC ID 寄存器。APIC ID 值只使用 0000～1110,1111 为保留。因此,从理论上讲,APIC 总线上可有 1 个 I/O APIC 模块和 14 个带 Local APIC 模块的处理器。但实际上,Pentium Ⅱ/Ⅲ只有双处理器系统,Pentium Pro 出现过四处理器系统(Quad 板)。于是,系统中 APIC ID 最多只有 5 个不同值。

系统初始化后,各模块的 APIC ID 寄存器值自动拷贝到各自的仲裁 ID 寄存器中,作为第一次 APIC 总线传输竞争的仲裁 ID。APIC 总线采用循环优先权法以尽量维护各模块的平等性。即每一次总线传输之后,竞争得胜者的仲裁 ID 值复位到 0000,其他各模块(包括未参与竞争的模块)的仲裁 ID 值增 1。当某模块的仲裁 ID 为 1111 时,增 1 将为 0000 并引起其他各模块(包括刚被复位到 0000 的模块)的仲裁 ID 值再增 1。这一循环优先权策略是基于如下两点考虑:第一,刚被 APIC 总线服务完的模块应排队到队尾;第二,在连续 15 次的 APIC 总线传输中,某模块或参与竞争未得胜或未参与竞争,则认为此模块为 APIC 总线传输的不积极发起者,理应排队到队尾。

必要的时候,可由一个处理器发起电平撤除的初始化消息的广播,即以 IPI 形式令各模块的仲裁 ID 值恢复到它们的 APIC ID 值的初始情况。

(2)目标模式

目标模式 D_tM 以及目标的 APIC ID 用于确定中断请求或消息传输的接收者。但要注意,APIC 子系统中确定信息传输的目标有三个特点:

第一,EOI 消息是由处理器的 Local APIC 传送到 I/O APIC 模块,而且系统只有一个 I/O APIC 模块,故 EOI 消息中不使用 D_tM 和目标 APIC ID,即目标是隐含确定的。

第二,不是 SMI, NMI,8259 兼容中断(ExtINT)这三种特殊中断的一般中断请求的传输,要由目标模式和递交模式联合确定中断请求的接收和响应者。若递交模式选择为固定式,则由目标模式和目标 APIC ID 确定的处理器(或组)即为目标。若递交模式选择为最低优先权式,则由目标模式和目标 APIC ID 确定的处理器组只是目标的候选者,要由最低执行优先权仲裁逻辑由其中选出一个处理器为最终的目标。

第三,因此,除 EOI 之外的所有消息传输以及 SMI, NMI, ExtINT 三种中断请求的传输,都只以目标模式和目标 APIC ID 来确定目标。

为叙述方便,下面使用"以目标模式和目标 APIC ID 来确定目标"这一术语,即涵盖了上述第三种情况和第二种的固定递交模式情况,至于最低优先权模式下的目标确定,留待下面讨论。

① 物理目标模式

如果 D_tM 位 = 0,则选择物理目标模式,它只使用处理器的 4 位 APIC ID 作为目标 ID。因为每个处理器的 APIC ID 是系统初始化过程设定的,对每个处理器是惟一的而且保持不变,故一般只选定一个处理器作为目标。但是,当 I/O APIC 模块向处理器传递系统中断请求时,若它在信息的目标 ID(D_3 ～D_0)域中给出 1111 的话,则是指定 APIC 总线上的所有处理器作为本次传递的目标。

② 逻辑目标模式

如果 DtM 位 = 1,则选择逻辑目标模式,此时它使用处理器的 8 位逻辑 ID 作为目标 ID。在每个处理器的 Local APIC 模块中,除 4 位 APIC ID 寄存器外,还有一个 32 位的逻辑目标寄存器 LDR(只使用最高的 8 位)和一个 32 位的目标格式寄存器 DFR(只使用最高的 4 位)。LDR 和 DFR 格式及使用如图 3 .23 所示。

首先,要对各处理器的 DFR 和 LDR 进行编程,设定选用平展方式还是成组方式并指定各处理器的 8 位逻辑 ID 值。然后,在每次 APIC 总线传输前,信息发送者设定 8 位目标 ID(I/O APIC 模块是重定向寄存器的目标 APIC ID 字段,处理器是中断命令寄存器的目标 APIC ID 字段)。逻辑目标模式选择目标灵活,可以选定目标是单处理器或是一组处理器。特别是结合最低优先权递交模式,能在一组处理器中选择当前具有最低执行优先权的处理器作为最终目标,是 APIC 先进可编程中断控制的特色之一。

③ 目标的简化管理

目标的简化管理只在处理器间中断(IPI)中采用。若中断命令寄存器中的 D_tS-

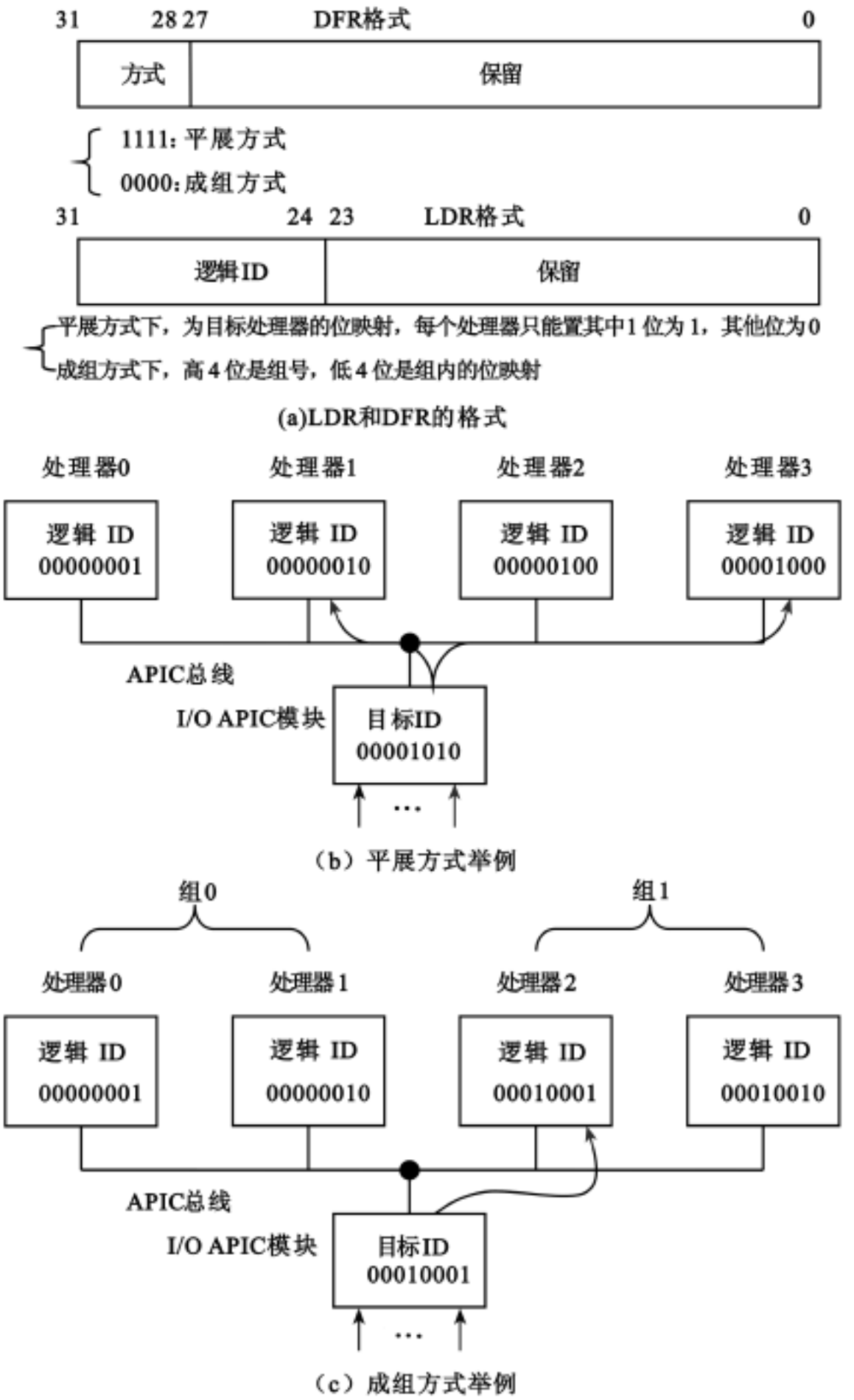


图 3 23　LDR 和 DFR 的格式及使用举例

H 字段(b_{19},b_{18})设定为非全 0,则此次中断请求或消息传输不使用 DtM 和目标 ID 来选择目标,而以 DtS-H 字段的设定来选择目标,目标可只是自身,可以是包括自身的所有处理器,或者是不包括自身的所有处理器。

(3)递交模式

APIC 子系统管理的三类中断:系统中断、本地中断和处理器间中断中,本地中断无 APIC 总线仲裁和目标模式问题,但三类中断都有递交模式设置问题。I/O APIC 模块的重定向寄存器和 Local APIC 的 LVT(本地中断向量表)表项以及中断命令寄存器中的 b_{10}～b_8 为递交模式(DM)字段,它的意义可总结成如下表示:

DM = 000　固定式
= 001　最低优先权式
= 010　系统管理中断
= 011　远程读
= 100　不可屏蔽中断
= 101　初始化
= 110　启动
= 111　8259 兼容的外部硬件中断(ExtINT)

当然,每类中断所使用的 DM 类型有所限制:系统中断只使用 DM 为 000,001,100,111 类型;本地中断的 LINT0,LINT1 只使用 000,100,111 类型,其他三种本地中断 DM 固定为 000;处理器间中断不使用 DM 为 111 类型。

我们看到递交模式的 8 种类型中,000,001 类型实际上是涉及到目标选择问题,而其他 6 种类型是指明递交的中断请求或消息的类型。因此,DM 为 010～111 的 6 种中断请求或消息的传输,目标只是固定式,即以 DtM 和目标 ID 选择目标。而一般中断请求(不是 SMI,NMI,ExtINT)的传输,目标的选择可有固定式和最低优先权两种方式。最低优先权方式是,以 DtM 和目标 ID 选择的一组处理器只是目标的候选者,要从中指出当前具有最低执行优先权的处理器作为最终目标。

最低优先权方式中的优先权指的是中断优先权,它涉及到中断向量号所反映的 16 个中断优先权级别(0～15,15 最高)。中断优先权管理方式是为了维护中断的正常嵌套,即正在执行的某优先权的代码不能被同级的或低级的优先权的中断所打断,只能被更高级优先权的中断所打断。

每个处理器的 Local APIC 模块中都有两个以 8 位中断向量号表示中断优先权的寄存器,它们是任务优先权寄存器 TPR 和处理器优先权寄存器 PPR,另外还有一个有效位也是 8 位的仲裁优先权寄存器 APR。

TPR 的值是操作系统设置的,多处理机的操作系统将某一任务分配给处理器时并在 TPR 中指明它的中断优先权级别。PPR 的值是处理器自动设定的,它总反映处理器当前正在执行的代码的中断优先权级别。当处理器没有中断发生时,PPR = TPR;当产生中断并被响应、服务时,PPR > TPR。故一般情况下是,PPR≥TPR。但

是,TPR 是软件设定的,故也存在这种情况:处理器正在执行一个中断服务子程序时,操作系统决定提升任务的中断优先权级别,此时会有 TPR > PPR。

还有一种情况是:中断请求寄存器 IRR 中的最高为 1 位的相应中断向量号(这里以 IRRV 来表示)大于 PPR 中的值,这表示一个更高中断优先权级别的中断请求已接收但还未响应和服务。

考虑到上述所有情况,一个处理器正在执行或即将执行的代码的中断优先权级别,应是 TPR,PPR 和 IRRV 中的当前最大值,以此值作为该处理器执行(或潜在执行)的优先权,参与竞争。

对于最低优先权方式来说,为和其他的优先权获得方式使用同一套总线仲裁机制,处理器将 TPR,PPR,IRRV 中的最大值,各位取反(1 的补码)后送入仲裁优先权寄存器 APR。在最低优先权的仲裁过程中,APR 由高位至低位,逐位发送到 PICD1 线并比较线上最终值。按大值得胜规则,最大的 APR 值将对应的是具有最低执行优先权的处理器。但它不同于 APIC 总线使用权的竞争过程,那时参与竞争的是处理器的 4 位 APIC ID 值,它们是惟一的,绝不会有相同的 APIC ID 值。现在,很可能两个(或多个)处理器有相同的 APR 值,若它们得胜将分不出伯仲。为此,在最低优先权中断请求的 APIC 总线传输过程中,在 8 个时钟的 APR 值判决之后紧跟的是 4 个时钟的 APIC ID 值判决,APIC ID 值大者将作为最终目标。

(4) 触发模式

触发模式控制有效中断请求的识别方式。可编程 TM = 0,为边沿触发;TM = 1,为电平触发。在 APIC 系统中,触发模式作为参量与中断向量一起,传递给目标处理器。电平触发的中断请求输入线可连接多个设备,边沿触发的中断请求输入线只连接一个设备。由于电平触发的中断存在多个设备共享的问题,故中断服务子程序的编制及中断结束处理与边沿触发中断有所不同。下面讨论电平触发模式下的共享中断问题。

在 APIC 系统中,只有系统中断和本地中断的 LINT0,LINT1 可设置中断请求为电平触发式。为叙述方便,讨论以系统中断为例,并假定 INTINi 中断请求输入线上挂接设备 A,B,C 三个设备,此请求线所对应的 I/O APIC 模块中的重定向寄存器 i 中已设定 TM = 1,中断向量为 V。

电平触发式中断必须解决如下两个问题:

第一,如何识别一个新的有效中断请求。既然不是边沿“敏感”的,那么就不能以中断请求信号从无到有的跳变来判测一个新的中断请求产生。APIC 的做法是,I/O APIC模块中的重定向寄存器中设置了一个中断请求 IR 位。若 IR 位为复位时,测到它的中断请求输入线为高电平,则认为产生一个有效的中断请求并置位 IR 位。在 IR 位置位的情况下,不再理睬中断请求输入线的电平变动。只有在此中断请求递交给目标处理器并中断服务子程序对此中断请求输入线上挂接的各设备都检查服务一遍之后,由目标处理器返回的 EOI 消息才使此 IR 位复位。于是,又可接收新的

中断请求。

I/O APIC 模块的重定向寄存器中的 IR 位,不同于 Local APIC 模块的 IRR 中位。后者是中断服务一开始,ISR 中的对应位被置位时,IRR 中的最高位即被清除。

第二,如何为共享中断的所有设备服务。一根中断请求输入线只有一个中断向量,一个入口地址的中断服务子程序如何为提出中断请求的各设备服务,而且每次提出中断请求的设备是变动的。APIC 的做法是,将各设备的服务子程序挂接到公共的中断向量上,形成一条串行服务链。具体而言,它的实现可分成设备安装的初始化和设备服务子程序的编制两个方面。

设备安装的初始化序列应按如下步骤完成:

① 将 INTINi 中断请求输入予以屏蔽。

② 检查挂接在该线上的设备,形成设备列表。

③ 按设备列表顺序,取一设备号。

④ 依此设备性质,安装该设备的驱动程序。

⑤ 安装该设备的中断服务子程序,将中断向量 V 处的原入口地址予以保存,将自己的中断服务子程序入口地址置入中断向量 V 处。

⑥ 完成此设备初始化的其他工作,之后开放该设备的中断请求产生允许标志。

⑦ 还有设备否?若有,转到③继续;否则,撤除 INTINi 中断请求输入的屏蔽,初始化完成。

编制的设备中断服务子程序应有如图 3.24 所示的结构。

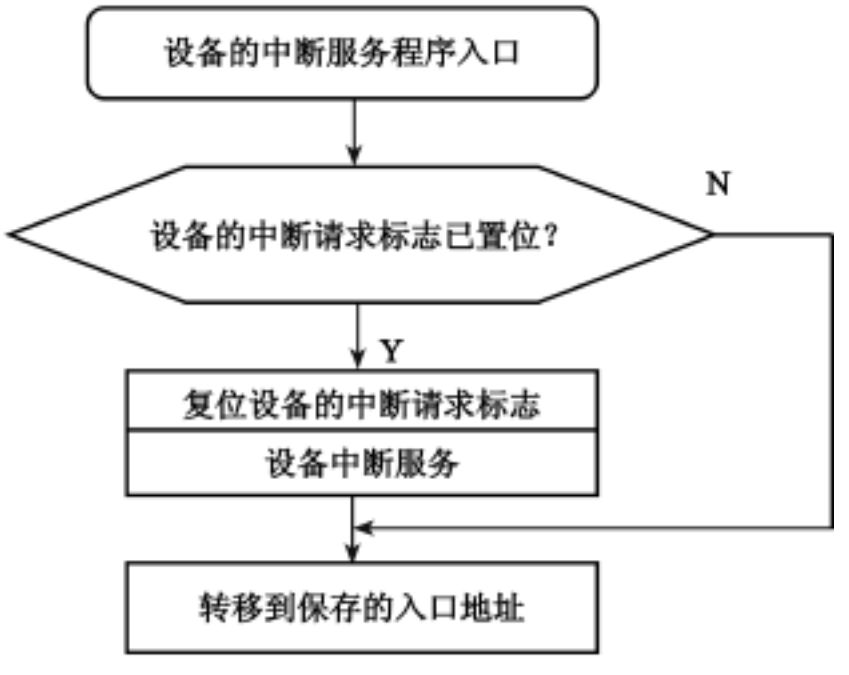


图 3.24 设备中断服务子程序结构

假定设备安装初始化之前,中断向量 V 处的入口地址的子程序,只有发布 EOI 命令和中断返回两条指令(亦称“空”设备中断服务子程序);并假定设备安装的顺序

是 A—B—C。则在设备安装初始化之后,中断向量 V 处的入口地址应是设备 C 中断服务子程序的入口地址,设备 C,B,A 的中断服务子程序串接成一条链,如图 3 .25 所示。

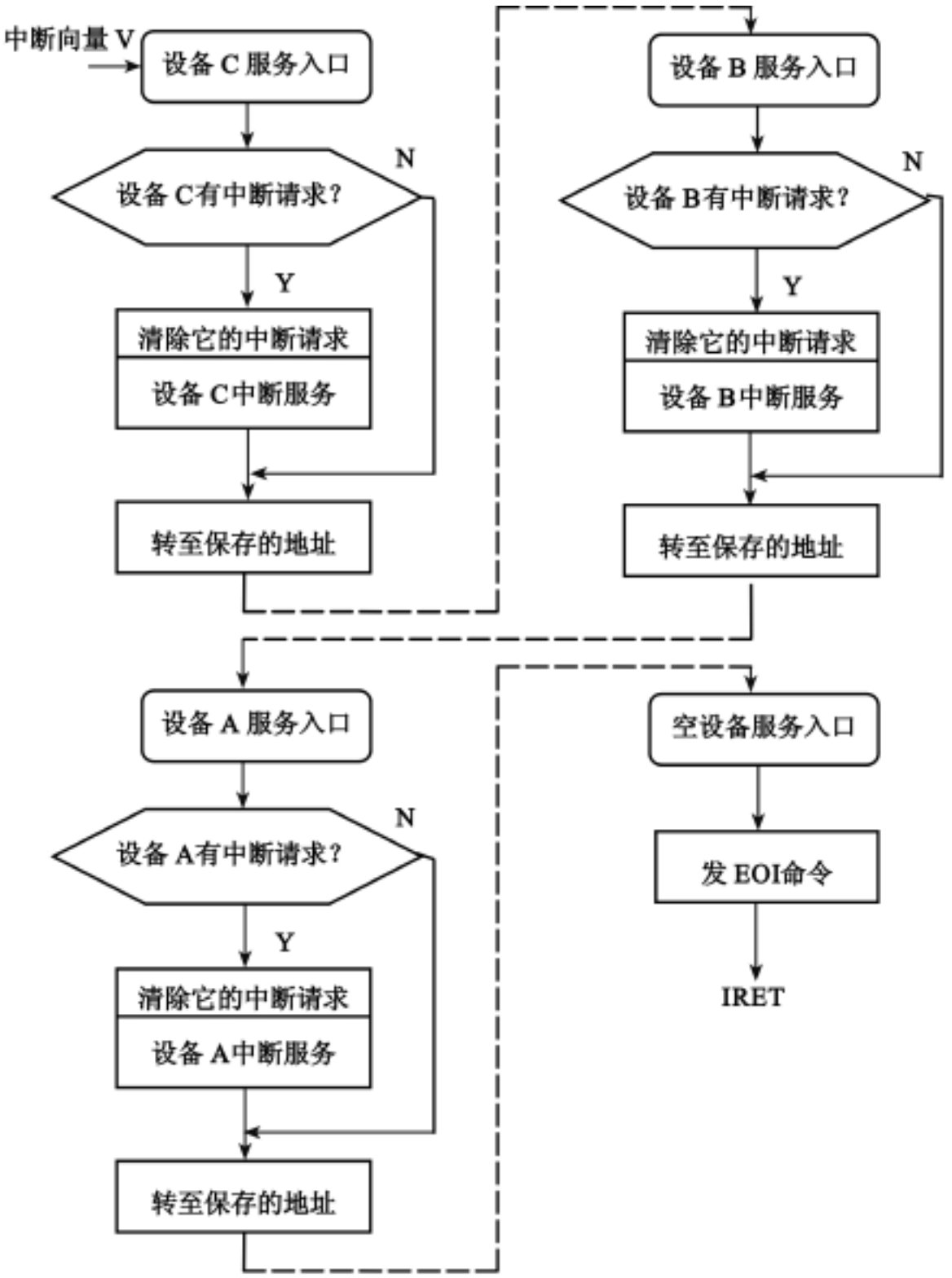


图 3 .25　共享中断的中断服务子程序结构

在对设备 C,B,A 依次查询且服务完毕后,进入空设备服务子程序。当目标处理器向 EOI 命令寄存器写入一个全 0 字时即发出中断结束命令。此时,EOI 命令不仅清除 ISR 中当前最高位,而且因 TMR 中对应位为 1(电平触发模式),清除此位将使

目标处理器发送一个 14 个 PICCLK 时钟周期的 EOI 消息给 I/O APIC 模块。它收到此消息后将依消息中给出的中断向量号找到对应的重定向寄存器,将其中的中断请求 IR 位清除。最后,一条 IRET 中断返回指令恢复处理器保存的现场,目标处理器由中断点继续执行原程序。

不难看出,在电平触发的 INTINi 中断请求输入线所对应的(重定向寄存器中的)IR 位已置位的情况下,若 INTINi 线上挂接的某设备产生中断请求,则此时并不能产生新的系统中断请求。若在中断服务子程序查询该设备之前,该设备的中断请求已置位,则该设备被服务并清除设备的中断请求;否则,当收到 EOI 消息 IR 位被清除后,该设备的中断请求立即形成新的系统中断请求。可见,共享中断的这种中断服务子程序的链式结构,不会漏掉任何设备在任何时刻提出的中断请求,能为挂接在 INTINi中断请求输入线上所有已提出中断请求的设备服务。

总之,APIC 系统是面向 SMP 多处理环境的,它的高级的可编程中断控制技术主要体现在三个方面:

第一,它可以将系统中 PCI 设备、IDE 设备、ISA 设备产生的中断请求递交给静态指定的处理器,或动态地在一组处理器中选择当前为最低执行优先权的处理器作为递交的目标。

第二,它可以在处理器之间传递初始化、启动等消息以及中断请求。

第三,它已将定时器、性能计数器等功能集成在处理器中,而将它们的中断作为本地中断。

所以说,APIC 技术保持了与 PC/AT 机原有的硬件中断如 NMI,8259 PIC 中断的兼容性,它们可以作为系统中断、本地中断或处理器间中断出现(除处理器间中断不能传递 8259 PIC 中断请求之外)。

3.4 通道处理机

在大型计算机系统中,外部设备的台数一般比较多,设备的种类、工作方式和工作速度的差别也比较大。为了使 CPU 摆脱繁重的 I/O 负担和共享 I/O 接口,IBM 公司从 IBM360 系列机开始,普遍采用了通道处理机技术。本节主要介绍通道处理机的作用、工作原理、工作方式及通道流量的计算。

3.4.1 通道的功能

1.通道的功能

正如 3.1 节所述,DMA 控制器的出现减轻了 CPU 对数据输入输出的控制,使得 CPU 的效率有显著的提高。而通道的出现则进一步提高了 CPU 的效率,特别是在大型计算机系统中,快速外部设备的数量显著增加,如果为每一台快速外部设备设

置一个 DMA 接口,付出的硬件代价很高,为解决这个问题,采用通道处理机是一种比较好的选择。

通道处理机实际上吸取了 DMA 硬件技术,并增加了软件管理,它设有专用的通道指令,尽管这些指令的功能有限,但能够负担外部设备的大部分 I/O 工作,包括管理所有按字节传输方式工作的低速和中速外部设备以及按数据块传输方式工作的高速外部设备,对 DMA 接口的初始化,设备故障的检测和处理等。虽然还不具备完整的指令系统,但是可以把它看做是一台能够执行有限 I/O 指令,并且能够被多台外部设备共享的小型 DMA 专用处理机。

在一台大型计算机系统中可以有多个通道,一个通道可以连接多个设备控制器,一个设备控制器可以管理一台或多台外部设备,这样就形成了一个非常典型的 I/O 系统的四级层次结构,如图 3.26 所示。

具有通道处理机的 I/O 系统结构一般具有四级连接:CPU 与内存\ 通道\ 设备控制器\ 外部设备。它有两种类型的总线,一种是存储总线,承担通道与内存、CPU 与内存之间的数据传送任务;另一种是 I/O 总线,即通道总线,承担外部设备与通道之间的数据传送任务。为便于通道对各设备的统一管理,对同一系列的机器,通道与设备控制器之间都有统一的标准接口,设备控制器与设备之间则根据设备要求不同而采用专用接口。

为防止通道成为限制系统性能的瓶颈,在具有通道 I/O 结构的大中型计算机系统中,一般都接有多个通道,这一方面可扩大数据的流量,另一方面对不同类型的 I/O可以进行分类管理。

通道的基本功能就是执行通道指令,组织外部设备和内存进行数据传送,按 I/O 指令要求启动外部设备,向 CPU 报告中断等,具体有以下几个功能:

① 接受 CPU 的 I/O 指令,根据指令要求选择一台指定的外部设备与通道相连接。

② 执行 CPU 为通道组织的通道程序。从主存中取出通道指令,对通道指令进行译码,并根据需要向被选中的设备控制器发送各种命令。

③ 组织外部设备和主存之间进行数据传送。主要包括给出外部设备的有关地址,即进行读/ 写操作的数据所在的位置,如磁盘存储器的柱面号、磁头号、扇区号等,并根据需要提供数据存入主存的地址和传送的数据量,统计外部设备和主存缓存区之间数据交换的个数,并依此判断数据传送工作是否结束。

④ 指定传送工作结束时要进行的操作。如将外部设备的中断请求和通道本身的中断请求,按次序及时报告 CPU 等。

⑤ 从外部设备得到设备的状态信息,是正常或故障,形成并保存通道本身的状态信息,根据要求将这些状态信息送到主存的指定单元,供 CPU 使用。

⑥ 在数据传送过程中完成必要的格式变换,如把字拆卸为字节,或者把字节装配成字等。

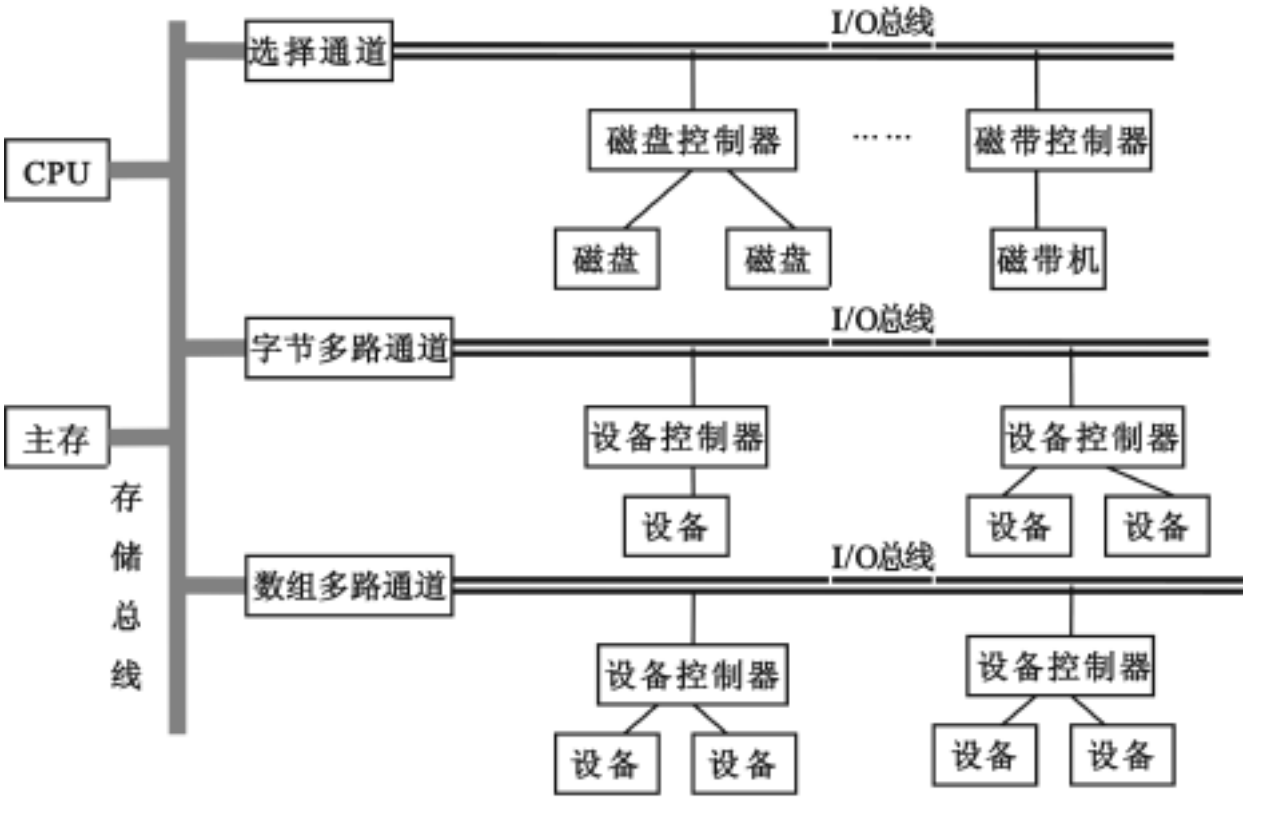


图 3.26 具有通道处理机的 I/O 系统结构

因此,通道应该能够执行一组通道指令,而且还要具有完成上述功能的硬件。通道的主要硬件包括寄存器部分和控制部分,寄存器部分主要有:数据缓冲寄存器、主存地址计数器、传送字节数计数器、通道命令字寄存器、通道状态字寄存器。控制部分有:分时控制、地址分配、数据传送、数据装配和拆卸等控制逻辑。

2.CPU 对通道的管理

CPU 通过执行 I/O 指令以及来自通道的中断请求,实现对通道进行管理。来自通道的中断有两种,一种是数据传送结束中断,另一种是故障中断。

在大、中型计算机中,I/O 指令都是特权指令,只有当 CPU 处于管态时,才能运行 I/O 指令,目态时不能运行 I/O 指令。这是因为大、中型计算机的软、硬件资源为多个用户所共享,而不是分给某个用户专用。

3.通道对设备控制器的管理

通道通过使用通道指令控制设备控制器进行数据传送操作,并以通道状态字接收设备控制器反映的外部设备的工作状态,因此,设备控制器是通道对 I/O 设备实现传输控制的执行机构。设备控制器的具体任务如下:

① 从通道接收通道指令,控制外部设备完成所要求的操作。

② 向通道反映外部设备的状态。

③ 将各种外部设备的不同信号转换成通道能够识别的标准信号。

3.4.2 通道的工作原理

通道执行输入输出操作的过程,可划分为三个阶段:准备阶段、数据传送阶段和通道操作结束阶段。完成一次输入输出的主要过程如图 3.27 所示,CPU 执行用户程序和管理程序,通道处理机输入输出主要过程的时间关系如图 3.28 所示。

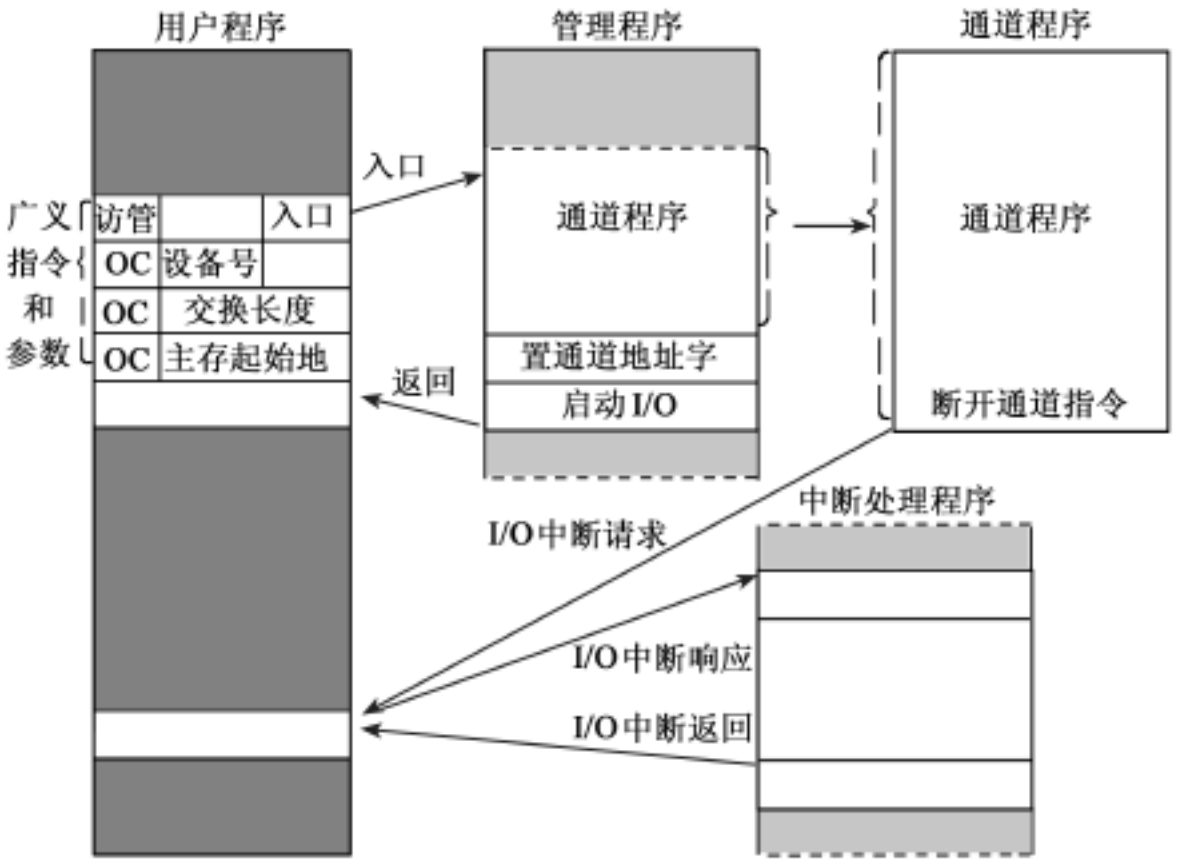


图 3.27 通道处理机输入输出的主要过程

1.准备阶段

在用户程序中如果要进行输入输出操作,执行访管指令进入管理程序,由 CPU 通过管理程序组织一个通道程序,并启动通道。

在多任务和多用户系统中,中央处理机用来控制外部设备操作用的输入输出指令被定义成管态指令,以使用户不得在目态程序中使用这些指令。这一方面可使多个用户分享 CPU、主存和外部设备资源,另一方面也可以防止用户窃取系统中不该让其读出的内容及防止用户可以自行输入而破坏其他用户程序和系统程序的考虑。这样,用户只有通过在目态程序中安排一条要求输入输出的广义指令来使用外部设备,经广义指令进入相应的管理程序来解释执行。

广义指令由访管指令和若干个参数组成,如图 3.27 所示。它的地址码实质上是对应此广义指令的管理程序入口。访管指令是目态指令,当目态程序执行到要求输入输出的访管指令后,产生自愿访管中断,如图 3.27 和图 3.28 所示。CPU 响应此

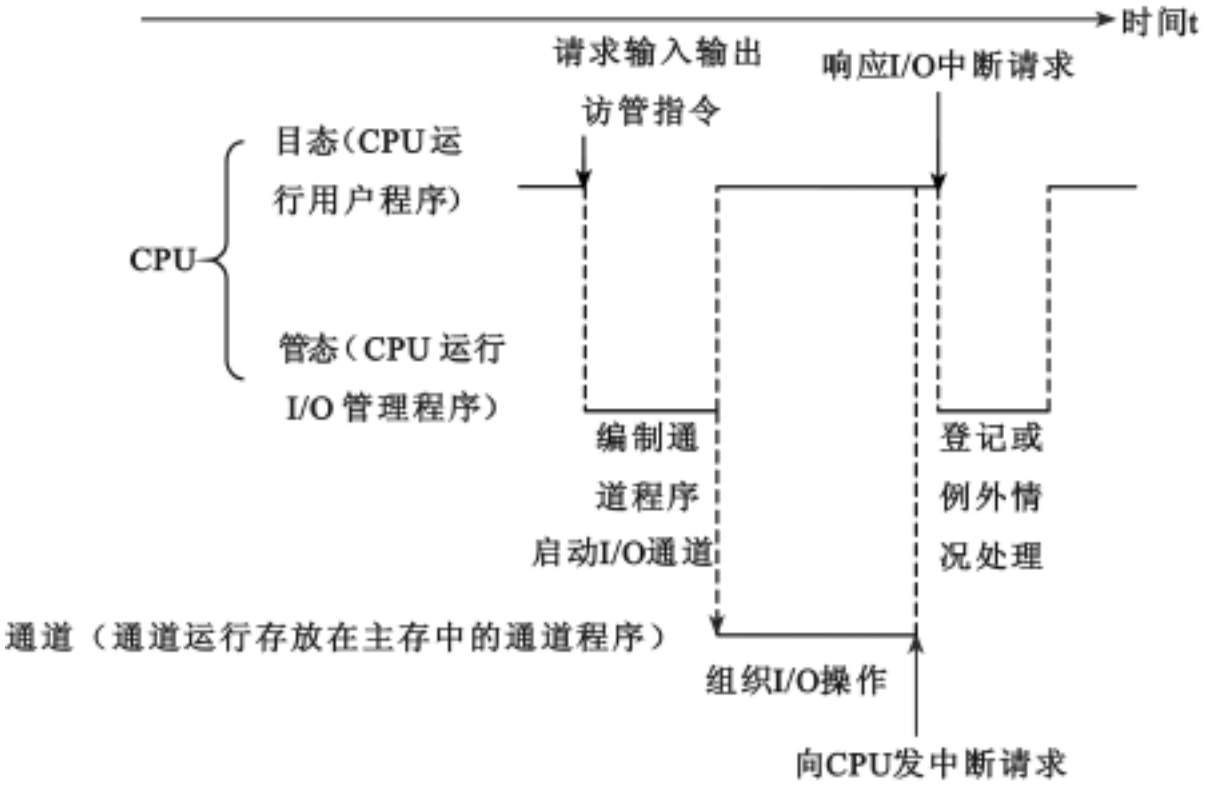


图 3.28 通道处理机输入输出主要过程的时间关系图

中断请求后,转向该管理程序入口,进入管态。

管理程序根据广义指令提供的参数,如设备号、交换信息的主存起始地址、要交换的信息长度等编制通道程序,通道程序能完成 CPU 一条输入输出指令所要求执行的许多操作。通道程序编制好后,就将它存放在主存中对应此通道的通道缓冲区中,再将通道程序的入口地址置入主存中通道地址字单元,并由管理程序指明操作方式。之后,管理程序就执行"启动 I/O"指令,开始所谓的通道开始选择设备期。

"启动 I/O"指令是主要的输入输出指令,它属于特权指令,在被访管指令调用的系统管理程序的最后一条执行。"启动 I/O"指令的工作流程为:先选择指定的通道和子通道,如果它们是在线而且是空闲的,就从主存中取出通道地址字,按照通道地址字给出的通道程序起始地址从主存的通道缓冲区中取出第一条通道指令。通道指令经过校验,如果指令格式是正确的,再选择指定的设备控制器和设备。如果被选择的设备是在线的,就向它发启动命令。设备被启动后,将向通道发回回答信号,如果设备的回答是一个全"0"字节,表示这台设备已经接受并执行了启动命令,这意味着此次启动成功,通道开始选择设备期结束。在上述过程中,如果任何一个地方不正确,表示启动输入输出设备没有成功,形成相应的条件码并结束启动过程,通道通过检测这些条件码,能够知道设备为什么没有启动成功。

2.数据传送阶段

通道被启动后,CPU 就可以退出操作系统的管理程序,返回到用户程序中继续

执行原来的程序,而通道进入通道数据传送期,开始与设备之间的数据传送。

通道和主存之间的数据交换,一般是以 DMA 方式进行的。

① 透明方式:在顺序方式中,执行指令期间通道接管主存访问总线。

② DMA 专用总线:由专门的 DMA 总线访问内存。

③ 总线接管方式:接管内存访问总线,待传送完毕后释放。

通道通过 CE 与外部设备交换数据,可以用同步方式,也可以用异步方式;可以用并行方式,也可以用串行方式,由具体的外部设备情况而定。

通道处理机是执行 CPU 为它组织的通道程序,完成指定的数据输入输出工作。从图 3 .28 中给出的时间关系可以看出,通道处理机执行通道程序是与 CPU 执行用户程序并行进行的。当通道处理机执行完通道程序的最后一条通道指令“ 断开通道指令 ”时,通道的数据传送工作就全部结束了。

3 .通道操作结束阶段

当数据传送完成,就转入通道数据传送结束期。

通道程序结束后(或设备出错,发停机中断给通道),通道向外部设备发停机命令,外部设备形成的设备状态字送通道,通道检查数据传送情况,形成对应的条件码,向 CPU 发中断请求。CPU 响应这个中断请求后,第二次进入管态,调用管理程序对输入输出中断请求进行处理。如果是正常结束,管理程序进行必要的登记计费等工作,如果是故障、错误等异常情况,则进行例外情况处理。然后,CPU 返回到目态,用户程序继续执行。

这样,每完成一次输入输出工作,CPU 只需要两次进入管态调用管理程序,大大减少了对用户程序的打扰,显著提高了 CPU 运算和外部设备操作的重叠程度。同时,系统中多个通道都可以有各自的通道程序在运行,使多种、多台外部设备可以充分并行工作。

在通道与设备之间的数据传送过程中,如果在同一个通道中有多台设备同时工作,或者是通道的“ 数据宽度 ”与要传送的信息宽度不等时,则要反复重新选择设备,即找出当前要传送数据的是哪一台设备。对于低速设备,每传送完一字节就要重新选择一次设备,而对于高速设备,通常每传送完一个数据块需重新选择一次设备。当然,如果一个通道只管理一台高速设备,那么,完成一次数据传送过程只需要做一次设备选择工作。

3.4.3 通道的类型

根据通道数据传送期中信息传送的方式不同,可将通道分为字节多路通道、选择通道和数组多路通道三种类型。

1 .字节多路通道

字节多路通道是一种简单的共享通道,主要用于连接大量的低速设备,如键盘、打印机等。这些设备的数据传输速率很低。例如数据传输率是 1000B/ s,即传送一个字节的时间是 1ms,而通道从设备接收或发送一个字节只需要几百纳秒(ns),因此通道在传送两个字节之间有很多空闲时间,字节多路通道正是利用这个空闲时间为其他设备服务。因此,其通道‘ 数据宽度 ”为单字节,采用字节交叉方式轮流为多个低速设备服务,以利于提高系统的效率。

一个字节多路通道,包括了若干个子通道,如图 3 .29 所示。每个子通道连接一

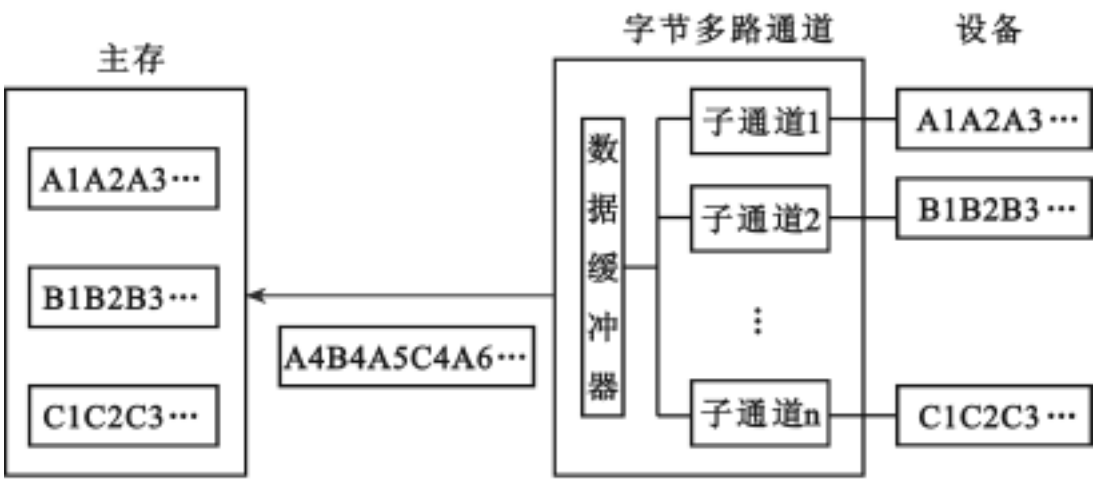


图 3 .29 字节多路通道示意图

个设备控制器,服务于一个设备。子通道的任务是提供字节缓冲、记录设备状态、传送通道命令、保存传送参数等。由于子通道的数目可以很多,例如,IBM 370 系列机的一个字节多路通道可以支持多达 256 个子通道。如果每个通道都有自己的一套硬件,必然造成总的设备量非常庞大,甚至达到不能容忍的程度。因此,目前的做法是:控制部分是公共的,由所有子通道共享,而寄存器部分每个子通道都有自己独立的一套。为了节省硬件,可以在主存储器中开辟出一个固定的区域来充当。对主存储器固定单元的访问速度完全能够满足低速和中速外部设备的要求。

每个子通道最少需要有一个字节缓冲寄存器,一个状态/ 控制寄存器以及指明固定地址的少量硬件。与各个子通道有关的参数,如主存数据缓冲区地址、交换字节个数等都存放在主存固定单元中。当通道在逻辑上与某一台设备连接时,就从主存相应的单元中把有关参数取出来,根据主存数据缓冲区地址访问主存储器,读出或写入一个字节,并将交换字节个数减 1,将主存数据缓冲区地址增量至下一个数据的地址。在这些工作都完成之后,通道控制逻辑切断与它的连接,就将通道与该设备在逻辑上断开,转为其他子通道服务。

2 .选择通道

选择通道适合于连接优先级高的磁盘这样一类的高速设备,故又称高速通道。在物理上它可以连接多个设备,但是这些设备不能同时工作,在某一段时间内通道只能选择一个设备进行工作。选择通道很像一个单道程序的处理器,在一段时间内只允许执行一个设备的通道程序,只有当这个设备的通道程序全部执行完毕后,才能执行其他设备的通道程序。所以数据传送以成块方式进行,相当于“数据宽度”为可变长块,在数据传送期内只进行一次设备选择。

选择通道的结构如图 3 .30 所示,它主要包含 5 个寄存器:数据缓冲寄存器、设备地址寄存器、主存地址计数器、交换字节数计数器和设备状态/ 控制寄存器等,另外,还有格式变换部件和通道控制部分等。

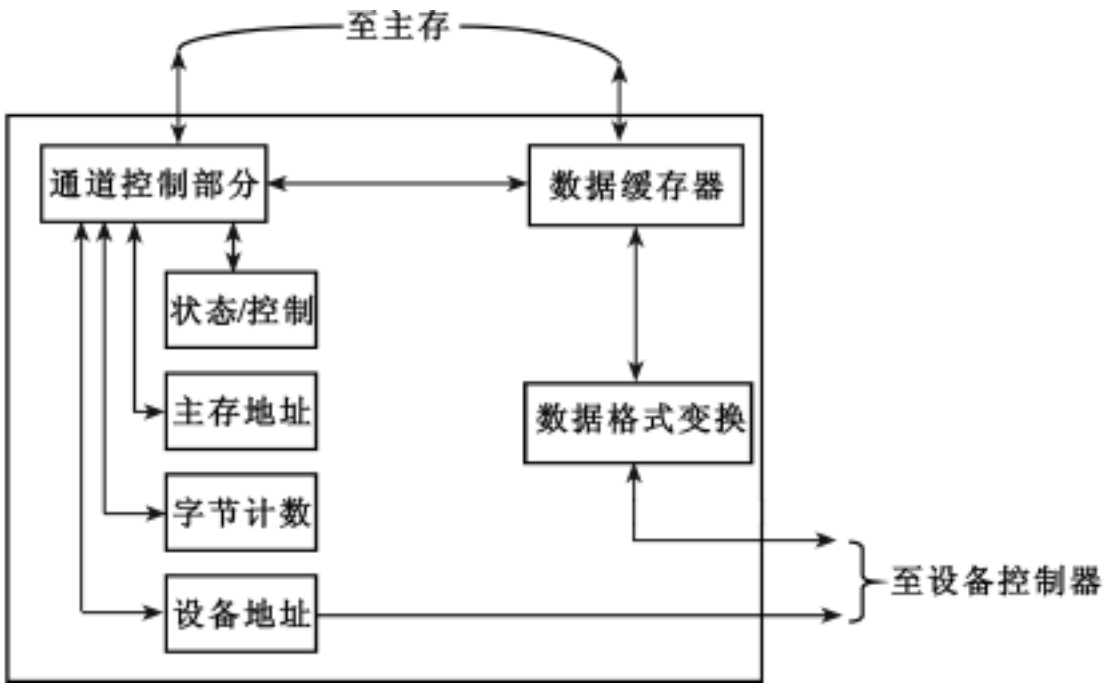


图 3 .30　选择通道的结构

因为外部设备与通道控制器之间通常是以字节为单位传送数据的,而通道与主存储器之间要以字为单位传送数据,一个字的长度一般为 32 位或 64 位。数据格式变换部件完成字到字节的拆卸及字节到字的装配。

选择通道传送数据的过程如图 3 .31 所示。

3 .数组多路通道

选择通道虽然能高速传送数据,但花费在设备辅助操作的时间不能有效地利用,如磁盘机启动后,磁头找到指定扇区的平均时间有 20～30ms,磁带机磁头定位时间更长,可达几分钟,在这样长的时间里,通道处于等待状态,为了利用这段时间,把上述字节多路通道和选择通道的特点结合起来,形成一种新的通道形式,称为数组多路

通道。

数组多路通道适合于为多台像磁盘等这样的高速设备服务。数组多路通道在向一台高速设备发出定位命令后,就立即从逻辑上与该设备断开,直到定位完成时再进行连接,发出找扇区命令后再一次断开,直到开始数据传送。其“数据宽度”为定长块,传送完定长 K 个字节数据后就重新选择下个设备进行 K 个字节数据的传送。因此,数组多路通道的实际工作方式是:通道在为一台高速设备传送数据时,有多台高速设备可以在定位或者在找扇区。它可以有多个子通道,可以同时执行多个通道程序,所有子通道能分时共享输入输出通路,但它是成组交叉传送的,既具有多路并行操作的能力,又具有很高的数据传输率。

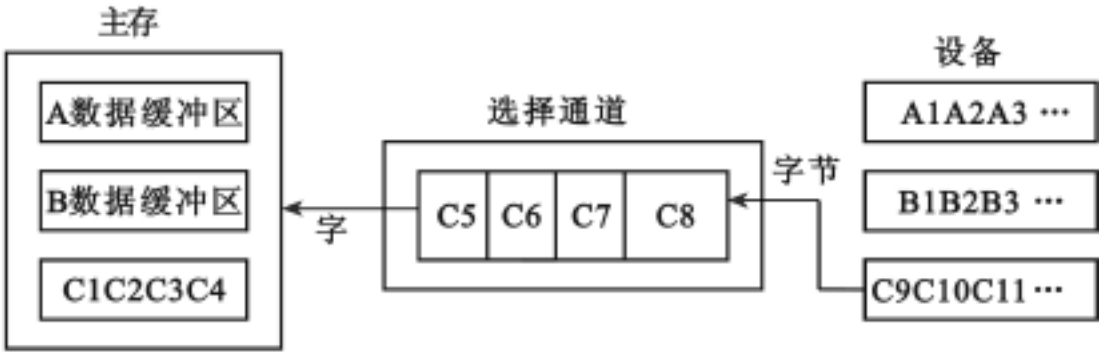


图 3 .31 选择通道传送数据的过程示意图

与选择通道相比,数组多路通道的数据传输率和通道的硬件利用都很高,但是,由于在一次输入输出过程中要多次与同一台高速外部设备连接和断开,因此,增加了控制硬件的复杂性。

目前,在大部分高性能计算机系统中均采用数组多路通道,也有一些计算机系统采用选择通道,或这两种高速通道都采用。

如在 IBM 4300 系统中连接有 1 个选择通道,1 个数组多路通道,1 个字节多路通道,通道号分别为 0,1,2。其中数组多路通道包括 8 个子通道,每个子通道可连接 8 台外部设备。字节多路通道包括 8 个子通道,通道号为 0～6 的子通道连接 8 台外部设备,通道号为 7 的连接 16 台外部设备。在选择通道下也可挂接 8 台外部设备。

3.4.4 通道流量分析

通道流量又称为通道吞吐率、通道数据传输率等,它是指通道在数据传送期内,单位时间内传送的字节数。它能达到的最大流量称通道极限流量。流量一般用字节个数来表示。一个通道能达到的极限流量与其工作方式、数据传送期内选择一次设备所用的时间 T_S 和传送一个字节所用的时间 T_D 等因素有关。它是设计系统 I/O 通道性能的主要指标。

对于字节多路通道,每选择一台设备只传送一个字节,故其通道极限流量为

$f_{max.byte} = 1/(T_S + T_D)$。数组多路通道每选择一台设备只传送 K 个字节,如果要传送 N 个字节,就得经[N/K]次传送才行,每次都要花去一个选择设备的时间 T_S,所以,其通道极限流量

$$f_{max.block} = K/(T_S + KT_D) = 1/(T_S/K + T_D)$$

选择通道每选择一台设备就把 N 个字节全部传送完,其通道极限流量

$$f_{max.select} = N/(T_S + NT_D) = 1/(T_S/N + T_D)$$

显然,如果通道的设备选通时间 T_S 和传送一个字节所需的时间 T_D 一定,且 $N > K$ 时,字节多路通道所能达到的极限流量最小,数组多路通道的流量居中,选择通道的流量最大。

根据字节多路通道的工作原理可知,设备要求通道的实际最大流量,应该是连接在这个通道上的所有设备的数据传输速率之和,即:

$$f_{byte.j} = \sum_{i=1}^{p_j} f_{i.j} \tag{3.1}$$

对于选择通道和数组多路通道,在一段时间内,一个通道只能为一台设备传送数据,而且,这时的通道流量就等于这台设备的数据传输速率,因此,无论对于这两种通道中的哪一种,其实际流量就是连接在这个通道上的所有设备中数据流量最大的那一个,即:

$$f_{block.j} = \max_{i=1}^{p_j} f_{i.j} \tag{3.2}$$

$$f_{select.j} = \max_{i=1}^{p_j} f_{i.j} \tag{3.3}$$

在式(3.1)至式(3.3)中,j 为通道的编号,$f_{i.j}$ 为第 j 通道上所挂的第 i 台设备的字节传输速率,p_j 为第 j 号通道中所接设备的台数。

为了保证通道能够正常工作,即使在所挂的设备满负荷工作的情况下都不丢失信息,必须满足设备要求通道的实际最大流量不超过通道所能达到的极限流量这一流量设计的最基本原则,因此,对上述三种类型的通道应分别满足以下关系式:

$$f_{byte.j} \leq f_{max.byte.j} \tag{3.4}$$

$$f_{block.j} \leq f_{max.block.j} \tag{3.5}$$

$$f_{select.j} \leq f_{max.select.j} \tag{3.6}$$

两边的差值越小,通道的利用率就越高,当两边相等时,通道处于满负荷工作状态。在实际设计最大通道流量时,应留有一定的余量,否则可能丢失信息。

如果 I/O 系统有 m 个通道,其中 1 至 m_1 为字节多路通道,$m_1 + 1$ 至 m_2 为数组多路通道,$m_2 + 1$ 至 m 为选择通道,则该 I/O 系统工作时的极限流量将为:

$$f_{max} = \sum_{j=1}^{m_1} f_{max.byte.j} + \sum_{j=m_1+1}^{m_2} f_{max.block.j} + \sum_{j=m_2+1}^{m} f_{max.select.j} \quad (3.7)$$

必然会满足:

$$f_{max} \geq \sum_{j=1}^{m_1} \sum_{i=1}^{p_j} f_{i.j} + \sum_{j=m_1+1}^{m_2} \max_{i=1}^{p_j} f_{i.j} + \sum_{j=m_2+1}^{m} \max_{i=1}^{p_j} f_{i.j} \quad (3.8)$$

也可以用式(3.8)左右两边的差值大小来衡量I/O系统流量利用率的高低。差值越小,I/O系统利用率越高,设计越合理。

f_{max}也是I/O系统对主存频宽的要求。但除了I/O系统外,CPU也要使用主存,因此从保证整个计算机系统各部分频带平衡的观点出发,由f_{max}根据一定的比例可以大致估算出要求达到的主存频宽。在主存频宽中,I/O系统所占比例与机器的用途密切相关。

例3.3 设有一字节多路通道,它有3个子通道,0号、1号高速印字机各占一个子通道,0号打印机、1号打印机和0号光电输入机合用一个子通道。假定数据传送期内高速印字机每隔25us发一个字节请求,低速打印机每隔150us发一个字节请求,光电输入机每隔800us发一个字节请求,则这5台设备要求通道的实际流量为:

$$f_{byte.j} = \sum_{i=1}^{p_j} f_{i,j} = \frac{1}{25} + \frac{1}{25} + \left(\frac{1}{150} + \frac{1}{150} + \frac{1}{800}\right) \approx 0.095\text{MB/s}$$

根据流量设计的基本条件,该通道的极限流量可设计成0.1MB/s,即所设计的通道工作周期为$t = \frac{1}{f_{byte.j}} = \frac{1}{T_S + T_D} = 10\text{us}$,这样各设备的请求就能及时得到响应和处理,不会丢失信息。

一般情况下总是让传输速率高的设备请求具有较高的响应优先级,设各设备请求得到响应的优先次序从高到低依次为:0号印字机、1号印字机、0号打印机、1号打印机、0号光电机。如果各设备要求传送字节数据的请求时刻如图3.32中的"↑"所示,由图可以看出,对每台设备,都是在该设备发出下一个申请之前,或最多发出申请的同时处理完上次申请,所以不会发生信息丢失。然而,各设备具体处理完每个字节请求的时间间隔却并不相同。

这里必须指出,上述流量设计的基本条件只是保证了从宏观上不丢失设备信息,并不能保证从微观上的每一个局部时刻都不丢失信息。特别是当通道的实际最大流量非常接近于通道设计所能达到的极限流量时,由于速率高的设备频繁发出请求并总是优先得到响应和处理,速率低的那些设备就可能会长期得不到通道而丢失信息。为此,常用的解决方法有三种:

第一,增加通道的最大流量,保证连接在通道上的所有设备的数据传送请求能够

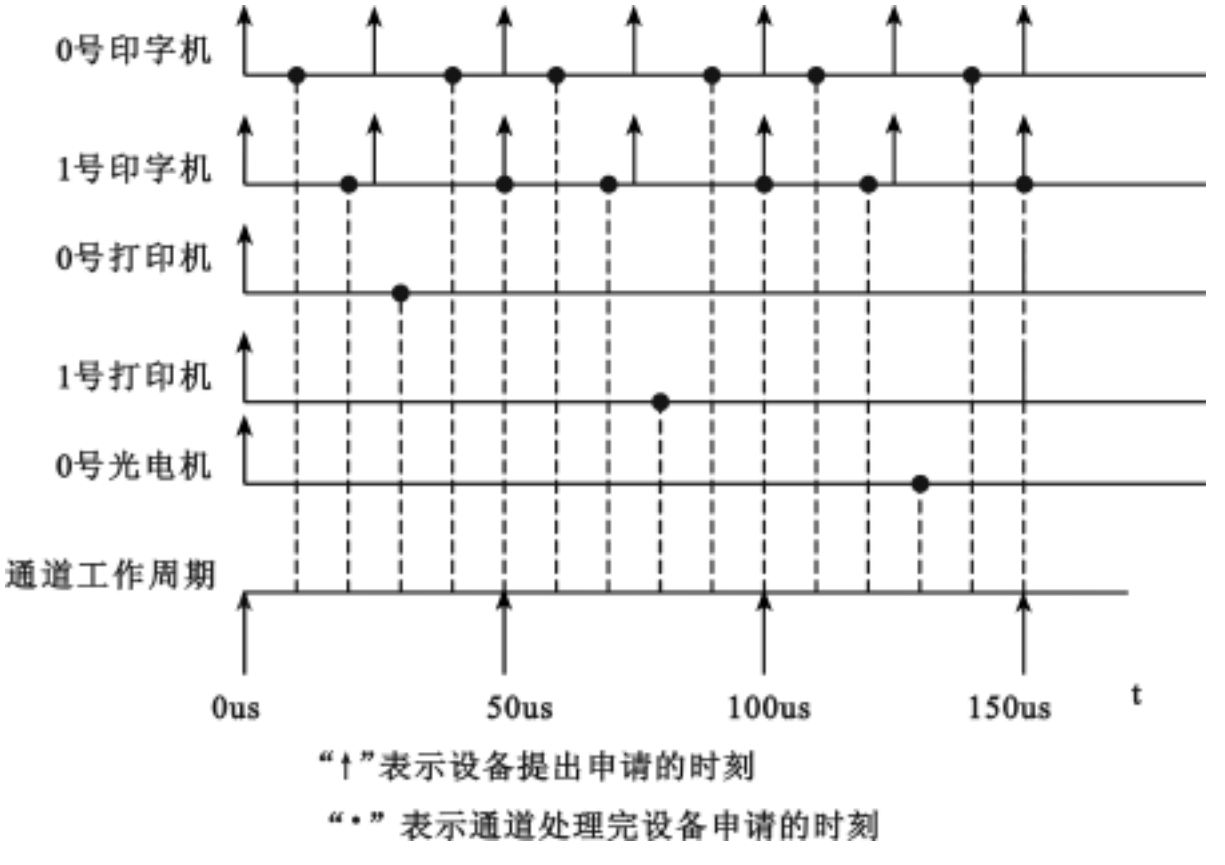


图 3 32　字节多路通道响应和处理各设备请求的时间示意图

及时得到通道的响应。

第二,动态改变设备的优先级,临时提高低速设备的响应优先级来保证从微观上也不丢失信息。当然,也可以采用临时降低设备优先级的办法,其效果是相同的。

第三,在设备或设备控制器中设置一定数量的数据缓冲器,以缓冲一时来不及处理的信息,特别是对优先级比较低的设备。

3.5　外围处理机

通道结构的进一步发展,出现了两种计算机 I/O 系统结构。

一种是通道结构的 I/O 处理器,通常称为输入输出处理器(IOP)。IOP 可以和 CPU 并行工作,提供高速的 DMA 处理能力,实现数据的高速传送。但是它不是独立于 CPU 工作的,而是主机的一个部件。有些 IOP,如 Intel 8089IOP,还提供数据的变换、搜索以及字装配/拆卸能力。这类 IOP 广泛应用于 IBM 公司系列机和一些中小型及微型计算机中。

另一种是外围处理机(PPU)。PPU 基本上是独立于主机工作的,它有自己的指令系统,完成算术/逻辑运算,读/写主存储器,与外部设备交换信息等。有的外围处理机干脆就选用已有的通用机。外围处理机主要用在除 IBM 公司以外的其他计算机公司研制的巨型、大型计算机系统中。

本节主要介绍外围处理机的工作原理和特点。

3.5.1 外围处理机的功能

通道处理机实际上并不能看成是独立的处理机,因为通道指令很简单,只有面向外部设备控制和数据传送的功能,而且没有大容量的存储器。在输入输出过程中,也还需要 CPU 承担许多工作:输入输出的前处理和后处理,设备或通道出现错误,异常后的处理,对所传送数据信息的代码和格式转换,数据块整体的正确性校验,文件管理、设备管理等操作系统的工作。中央处理机资源往往得不到充分利用,造成很大浪费。特别是对使 CPU 能高速运行所采用的流水等组成技术,常会因为遇到输入输出中断而发挥不了作用,速度严重下降。为此发展了外围处理机,让 CPU 进一步摆脱对输入输出操作的控制,使 CPU 和外围处理机各负其职,各自发挥自己的作用,这是巨型、大型计算机系统,以及一些输入输出任务繁重的计算机系统的最佳选择。

由上可知,外围处理机除了能够完成通道处理机的全部功能之外,还具有如下功能:

① 码制转换。例如,ASCII 码与 EBCDIC 码之间的转换,ASCII 码与 BCD 码之间的转换,BCD 码与二进制数之间的转换等。

② 数据块的检错和纠错。外部设备的可靠性一般都比主机低,因此,在许多外部设备中都有相当复杂的校验和校正功能。如磁盘存储器和磁带存储器中,设置 CRC 循环冗余校验码,且采用硬件实现校验和校正,而在数据传送过程中的校验与校正一般要在外围处理机中通过程序来实现。

③ 故障处理。在外围处理机中,当外部设备或处理机本身出现故障时,可以自己来处理,这是因为外围处理机设置有算术逻辑指令和程序控制指令等。而在通道处理机中,要通过中断方式报告 CPU,由 CPU 来进行处理。

④ 文件管理。文件管理是一件相当繁琐的工作,外围处理机可以代替 CPU 完成文件管理的大部分工作。在有些计算机系统中,甚至专门设置有文件处理机。

⑤ 诊断和显示系统状态。通过定时运行诊断程序,可以诊断外部设备和外围处理机是否正常。如在 CRAY-1 向量计算机中,就设置有专门的诊断维护控制处理机。

⑥ 处理人机对话。人机对话一般要通过反复调用管理程序来实现,有时把这种管理程序称为监控程序。在设置有外围处理机的计算机系统中,一般把处理人机对话的管理程序放在外围处理机上执行。

⑦ 网络或远程终端可以直接连接到外围处理机上,由外围处理机完成远程用户服务工作。

外围处理机之所以能够完成上述这些通道处理机所不能完成的工作,关键是外围处理机除了具有数据的输入输出功能外,还具有运算功能和程序控制等功能,就像

一个一般的处理机那样,有时干脆就采用一般的通用机。由于它具有较丰富的指令,功能也较强,所以还有利于简化设备控制器,甚至还可进一步承担分配给它的其他任务,如数据库和知识库的管理工作等。

总之,外围处理机通常是一台独立的处理机,具有一定的运算功能,可以承担一般的输入输出、控制操作和运算处理等任务。而且由于外围处理机具有自己的存储器,因此,不必通过主存储器就能完成与外部设备的数据交换,可以进一步提高整个计算机系统的性能。

3.5.2 外围处理机的特点

从 20 世纪 70 年代后期开始,CDC 公司首先在其研制的 6600 大型计算机系统中采用了外围处理机工作方式。下面以 CYBER 170 巨型计算机的外围处理机为例来说明它的结构特点。CYBER 170 外围处理机的结构如图 3.33 所示。

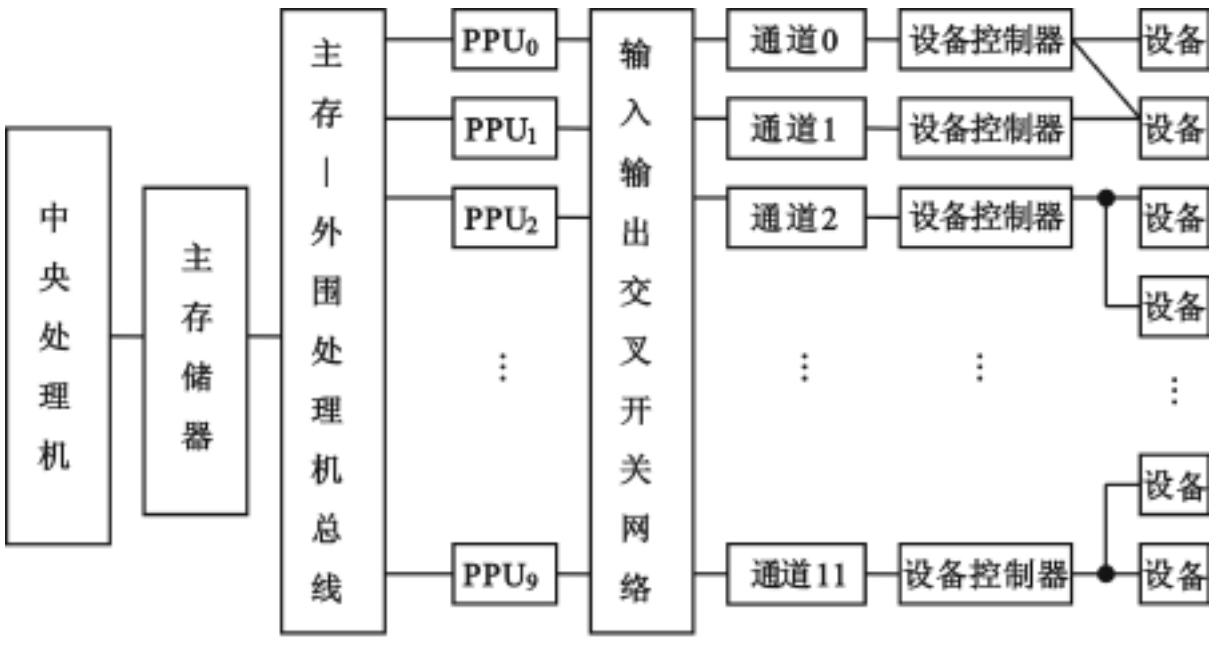


图 3.33 CYBER 170 外围处理机的结构框图

外围处理机子系统包括 10 台 PPU,即 PPU_0～PPU_9,它们通过主存—外围处理机总线分时共享主存储器,通过输入输出交叉开关网络(I/O Crossbar Switching)共享 12 个输入输出通道,每个 PPU 有一个容量为 4k×13 位(其中一位为奇偶位)的局部存储器。系统监督程序常驻在 PPU_0 的局部存储器中,控制台显示程序常驻在 PPU_1 的局部存储器中,其余 PPU 均装有各自的常驻程序。

每台 PPU 都能独立执行有关 PPU 的程序,都有相同的指令系统,共有 66 条不同的指令,包括算术/逻辑运算指令、读/写主存指令、输入输出指令及程序控制指令等。指令格式有 12 位的短指令和 24 位的长指令两种,如图 3.34 所示。用这些指令编制的 PPU 程序存放于系统主存的程序库中,可以为管理外部设备随时调用。所有 10 台 PPU 分时循环使用同一个算术/逻辑部件,每台 PPU 一次占用一个时间片,隔

10 个时间片之后又可再次占用一个时间片。因此一条 PPU 指令要经过多个大循环周期才能完成。

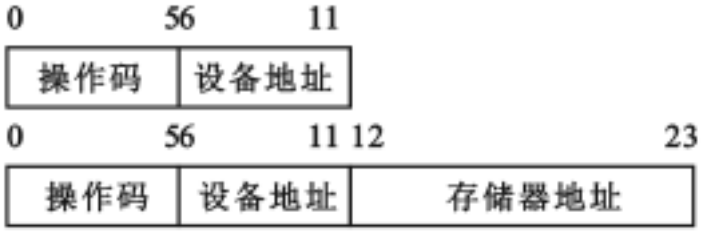


图 3.34 CYBER 170 PPU 的指令格式

中央处理机不能直接与外部设备交往,当用户程序需要调用输入输出操作时,由中央处理机发出调用 PPU 请求,即可继续执行它的用户程序,然后由外围处理机自己来完成与外部设备的通信,像在通道处理机方式中那种前处理工作就不用做了。但由于主存储器字长 60 位,PPU 局部存储器字长只有 12 位,它们之间交换信息时,需要采用桶形移位器的方法进行拆卸和装配,这种拆卸和装配所用的部件也是由 10 个 PPU 分时共享的。

每台 PPU 有 4 个寄存器,A 为累加寄存器,用于保存操作数、主存地址、输入输出字节数等,同时它也是 PPU 与其他部件之间进行通信的接口寄存器。P 为程序计数器,Q 为多功能寄存器,用于保存直接地址、间接地址、通道号或转移计数值等。K 为指令寄存器,用来保存指令操作码和指令执行周期。

通道主要由一个双向 13 位(1 位奇偶位)的通道寄存器和两个状态标志触发器组成,其中一个是通道的“忙/闲”触发器,另一个是“满/空”触发器,利用这两个状态标志触发器以及 PPU 执行相关的 I/O 指令控制 PPU、通道与外部设备的通信。不过这里的通道与上一节介绍的通道处理机有本质的不同,这里的通道只起通路连接的作用,或者可理解为一种简单的接口,它使外围处理机可以与 12 个通道中的任何一个相连接。每个通道最多可连接 8 台外部设备,用设备号来选择。

CYBER 170 的外围处理机大致工作过程如下:

当 PPU 要与外围设备交换数据时,先将对应的设备号放在该 PPU 的累加寄存器 A 中,并将有关通道的“忙/闲”和“满/空”触发器都置“1”(相当于“忙”和“满”),接着使用 PPU 指令把累加寄存器 A 的内容送往有关通道,用以选择和连接外部设备。当设备收到该信号后,返回 PPU 一个“未工作”的回答信号,再将两个触发器置成“0”状态,设备选择和连接阶段结束,开始命令发送阶段。由 PPU 依次将确定信息交换的性质和格式的命令放在累加寄存器 A 中,并将它送至外部设备控制器,然后再将数据交换个数存放到累加器 A 中,接着通过 PPU 指令启动通道,通道启动成功后,执行 PPU 指令,就转入数据传送阶段。在数据传送期间,每当有一个数据通过通道时,就将“满/空”触发器置“1”,数据被取走后,“满/空”触发器清“0”,同时将累加器 A

中的字节个数减“1”。重复上述过程,直至累加器 A 中字节个数为“0”,整个数据传送过程全部结束,这时就可以断开 PPU 与通道、通道与外部设备之间的连接。

3.5.3 外围处理机的分类

外围处理机基本上是独立于中央处理机异步工作的,它可以与中央处理机共享主存储器,也可以有自己独立的存储器,不共享主存储器。每台外围处理机可以有自己独立的运算部件和指令控制部件,也可以由多个外围处理机共享同一个运算部件和指令控制部件。

根据是否共享主存储器,可以把外围处理机分为两大类:

① 共享主存储器的外围处理机。许多早期的巨型和大型计算机系统一般采用这种方式。例如,CDC 公司的 CYBER, Texas 公司的巨型计算机 ASC, Burroughs 公司的 B- 6700 等系统都采用共享主存储器的连接方式。

这种方式的外围处理机有一个小容量的局部存储器。外围处理机要执行的管理程序一般放在主存储器中,为所有 PPU 共享,只有当需要时才通过加载或覆盖等方式把程序装入到对应 PPU 的局部存储器中。

② 不共享主存储器的外围处理机。例如,早期的 STAR-100 巨型计算机。目前的大多数并行计算机系统都采用这种不共享主存储器的连接方式,但却需要很大容量的局部存储器。各台外围处理机所运行的管理程序都存放在自己的局部存储器中,因此,这种方式可以最大限度地减少对主存储器的压力。

根据运算部件和指令控制部件是否为各个外围处理机共享,也可以把外围处理机分为两类。

① 合用同一个运算部件和指令控制部件的外围处理机。如 CYBER 和 ASC 等巨型计算机,各台 PPU 共用同一个运算部件和指令处理部件,并通过公用部件与主存通信。这可以降低外围处理机子系统的造价,但控制相对比较复杂。

② 独立运算部件和指令控制部件的外围处理机。例如,B- 6700 大型计算机和 STAR-100 巨型计算机。由于 VLSI 技术的高速发展,采用独立运算部件和指令控制部件的外围处理机已经成为主流。而且,这种外围处理机往往都有各自的大容量存储器,具有更强的独立性。

根据各种计算机系统的具体情况和不同要求,外围处理机的结构有多种组织方式:

① 在有些计算机系统中,有多个外围处理机,而且从功能上进行分工,有的专门管理外部设备,有的专门管理文件系统,有的专门管理用户的人机会话工作,有的专门管理网络和远程终端,有的专门管理数据库或知识库等。

② 在许多并行计算机和超级并行计算机系统中,以外围处理机作为主处理机,它除了担负全部输入输出任务之外,还运行操作系统,而由多个处理机或多个运算部件组成的并行处理系统仅作为运算的加速部件。

③ 在有的计算机系统中,用一台与中央处理机相同型号的处理机作为外围处理机,例如,有一种由两台 CRAY 大型计算机组成的系统,其中的一台计算机系统专门负责输入输出工作。

④ 随着集成电路技术的迅速发展,目前,在许多计算机系统中往往采用廉价的微处理机来专门承担输入输出任务,例如,Intel 公司的 8089 微处理器、80168 微处理器等经常被用来作为专用的外围处理机。

综上所述,随着微处理机价格的进一步下降,一般可采用微处理机作为外围处理机,以构成一个由中央处理机、外围处理机、主存储器、通道、设备控制器和各种外部设备相互独立的计算机系统,并且,能够根据需要通过程序动态地、灵活多变地控制它们之间的连接,具有比通道处理机方式强得多的灵活性。由于 PPU 具有一定的运算功能,可以承担一般的外围运算和操作控制任务,还可以让各台外部设备不必通过主存就可以直接交换信息,这些都进一步提高了整个计算机系统的工作效率。

如果进一步增强输入输出设备与设备控制器的“智能”,发展智能外部设备,让管理、控制操作尽可能在端点完成,使调用外部设备的过程变成是在 I/O 系统中各外围处理机之间及各缓冲存储器之间的信息传送过程,这将会继续提高 I/O 系统的数据吞吐率和让中央处理机解脱对输入输出控制管理的负担。

习　题

1.从系统结构的观点看,I/O 系统的设计对整个计算机系统有何影响?

2.设一个任务在计算机中的处理时间为 50s,CPU 处理时间为 30s,I/O 处理时间为 20s。如果将 CPU 的处理速度提高 4 位,则总的处理时间将是多少?

3.设一个任务的处理时间为 64s,CPU 在这期间始终忙于处理,I/O 处理时间为 36s。为提高系统性能,有两种方案:使 CPU 的速度增加 1 倍,或者使 CPU 和 I/O 的处理速度同时增加 1 倍。计算这两种情况下的处理时间。

4.分析和归纳集中式串行链接、定时查询和独立请求这三种总线控制方式的优缺点。

5.总线有哪些类型?总线的标准是如何制定的?

6.从一个中断源发出中断服务请求,到这个中断服务请求全部处理完成,并返回到原来的中断点所经过的过程称为中断处理过程。在一次完整的中断处理过程中,主要做了哪些工作?其中,哪些必须用硬件实现?哪些必须用软件实现?哪些可以用硬件实现也可以用软件实现?

7.有 5 个中断源 D1,D2,D3,D4 和 D5,它们的中断优先级从高到低分别是 1 级、2 级、3 级、4 级和 5 级。这些中断源的中断优先级、正常情况下的中断屏蔽码和改变后的中断屏蔽码,如表 3.4 所示。每个中断源有 5 位中断屏蔽码,其中,“1”表示该中断源被屏蔽,“0”表示该中断源开放。

表 3.4

中断源名称	中断优先级	正常的中断屏蔽码 D_1 D_2 D_3 D_4 D_5	改变后的中断屏蔽码 D_1 D_2 D_3 D_4 D_5
D_1	1	1 1 1 1 1	1 0 0 0 0
D_2	2	0 1 1 1 1	0 1 0 0 0
D_3	3	0 0 1 1 1	1 0 1 0 0
D_4	4	0 0 0 1 1	1 1 0 1 1
D_5	5	0 0 0 0 1	1 1 1 0 1

① 当使用正常的中断屏蔽码时,处理机响应各中断源的中断服务请求的先后次序是什么?实际的中断处理次序是什么?

② 当使用改变后的中断屏蔽码时,处理机响应各中断源的中断服务请求的先后次序是什么?实际上中断处理的优先次序是什么?

③ 如果采用改变后的中断屏蔽码,当 D1,D2,D3,D4 和 D5 这 5 个中断源同时请求中断服务时,画出处理机响应中断源的中断服务请求和实际运行中断服务程序过程的示意图。

④ 假设从处理机响应中断源的中断服务请求开始,到运行中断服务程序中第一次开中断所用的时间为一个单位时间,处理机运行中断服务程序的其他部分所用的时间为 4 个单位时间。当处理机执行主程序时,中断源 D3,D4 和 D5 同时发出中断服务请求,过 3 个单位时间后,中断源 D1 和 D2 同时发出中断服务请求。采用改变后的中断屏蔽码,画出处理机响应各中断源的中断服务请求和实际运行中断服务程序过程的示意图。

8.某处理机共有 4 个中断源,分别为 D1,D2,D3 和 D4。要求处理机响应中断源的中断服务请求的次序从高到低分别为 D1,D2,D3,D4,而处理机实际为各中断源服务的先后次序为 D3,D2,D4,D1。每个中断源有 4 位中断屏蔽码,其中,“1”表示该中断源被屏蔽,“0”表示该中断源开放。

① 请设计各中断源的中断优先级和中断屏蔽码。

② 如果处理机在运行主程序时,同时有 D1 和 D2 两个中断源请求中断服务,而在运行中断源 D2 的中断服务程序的过程中,中断源 D3 和 D4 又同时请求中断服务,请画出处理机响应各中断源的中断服务请求和实际运行中断服务程序过程的示意图。

9.简述字节多路、数组多路和选择型通道的数据传送方式,并回答下面的问题。

① 字节多路通道、数组多路通道、选择通道,它们一般用什么数据宽度来进行

通信？

② 如果通道在数据传送期中选择设备需 9 .8us,传送一个字节数据需 0 .2us,某低速设备每隔 500us 发出一个字节数据传送请求,问至多可接几台这种低速设备？对于如下 A～F 种高速设备,一次通信传送的字节数不少于 1024 个字节,问哪些设备可以挂在此通道上？哪些则不能？其中 A～F 设备每发一个字节数据传送请求的时间间隔,分别如表 3 .5 所示。

表3 .5

设　　备	A	B	C	D	E	F
发申请间隔(us)	0 .2	0 .25	0 .5	0 .19	0 .4	0 .21

10 .一个字节多路通道连接有 4 台外部设备,每台设备发出输入输出服务请求的时间间隔、它们的服务优先级和发出第一次服务请求的时刻如表 3 .6 所示。

表3 .6

设备名称	DEV_1	DEV_2	DEV_3	DEV_4
发服务请求间隔	10us	75us	15us	50us
服务优先级	1(最高)	4	2	3
发出第一次请求时刻	0us	70us	10us	20us

① 计算这个字节多路通道的实际流量和工作周期。

② 在数据传送期间,如果通道选择一次设备的时间为 3us,传送一个字节的时间为 2us,画出这个字节多路通道响应各设备请求和为设备服务的时间关系图。

③ 从②的时间关系图中,计算通道处理完成各设备第一次服务请求的时刻。

④ 从②画出的时间关系图中看,这个字节多路通道能否正常工作(不丢失数据)？为什么？

⑤ 在设计一个字节多路通道的工作流量时,可以采用哪些措施来保证通道能够正常工作(不丢失数据)？

11 .一台计算机系统有一个选择通道,两个数组多路通道,一个字节多路通道带有 3 个子通道。各通道的工作速度如表 3 .7 所示。

表 3.7

通道名称		连接在这个通道上的设备的数据传输速率(KB/s)
字节多路通道	子通道 1	100,50,50,25,20,5
	子通道 2	60,60,60,45,15,10
	子通道 3	100,100,80,80,80,60
数组多路通道 1		4000,4000,4000,3000,3000
数组多路通道 2		4000,4000,4000,3500,3000
选择通道 1		5000,5000,5000,4500,4000
选择通道 2		6000,6000,5000,5000,5000

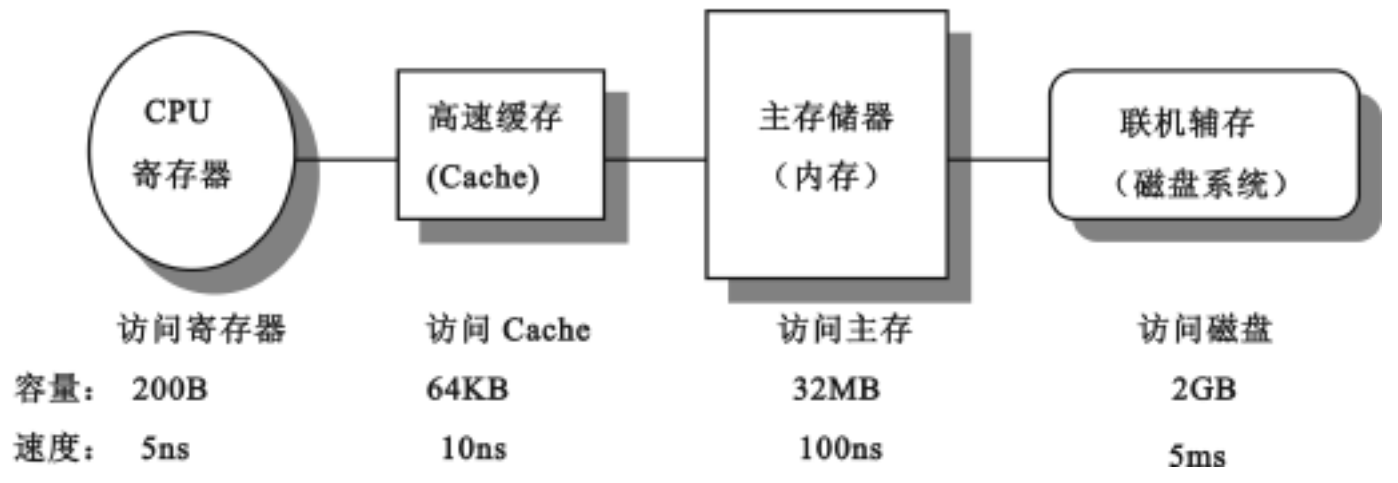
① 分别计算各通道和子通道的实际流量和工作周期。

② 假定整个 I/O 系统流量占主存流量的 1/2 时,才认为这两者速度相匹配,那么主存流量应达到多少?

第四章 存储系统

4.1 存储系统的原理

存储器是计算机的核心部件之一,其性能直接关系到整个计算机系统性能的好坏。计算机中的存储器,从结构上讲可分为多种,主存储器(内存)、高速缓冲存储器(Cache),联机辅存和脱机辅存等。计算机中常用存储器的基本层次结构如图 4.1 所示。



注:图中所标参数是 1995 年末中低档台式计算机的典型值

图 4.1 常用存储器的基本层次结构

4.1.1 存储系统的意义

1.存储器的三个主要参数

一个存储器的性能通常用三个主要参数(指标)来表示:容量、速度和每位价格。

① 存储器的容量 $S_M = W \times l \times m$。其中,$W$ 为存储器的字长(单位为位或字节),l 为存储器的体长(字数),m 为并行工作的存储体的个数(或称为模数)。容量的单位有位(bit),字节(B),千字节(KB)($1KB = 2^{10}B$),兆字节(MB)($1MB = 2^{10}KB = 2^{20}B$)和千兆字节(GB)($1GB = 2^{30}B$)等。

② 存储器的速度可用访问时间 T_A、存储周期 T_M 和频宽 B_M 来描述。

T_A 是存储器接到访存申请,完成一次 R/ W 所需的时间;

T_M 是指连续完成两次 R/ W 所需的时间间隔,一般有 $T_M > T_A$;

B_M 是指存储器在单位时间内传送的信息量,常用下式表示:

$$B_M = m \times \frac{W}{T_M}$$

B_M 单位常用 b/ S 或者 B/ S, KB/ S, MB/ S 和 GB/ S 等表示。

③ 存储器的价格是指每位价格,即 $C = \frac{\text{存储器的总价格}}{\text{存储器的总容量}}$,单位为每位多少美分($C/ bit)。

主观上,人们总希望存储器的容量越大越好,速度越快越好,而价格越便宜越好。实际上,这三个指标的提高总是相互矛盾的,那么,系统结构设计者又是如何解决这一矛盾的呢?答案是发展存储系统。

2. 存储系统的意义

存储系统又称为存储体系或存储层次,它是指两个或两个以上的速度、容量和价格不同的存储器用硬件、软件或软硬结合的方法构成的一个完整的系统。这个系统对应用程序员透明。

存储系统有两个特点:一是由多级存储器构成的这个系统,对于应用程序员是一个存储器,而且速度接近第一级存储器的速度,容量和每位价格接近最后一级存储器的容量和每位价格;二是很好地解决了存储器速度、容量和价格三者之间的主客观矛盾,为计算机中存储器性能的提高找到了一个很好的途径。

3. 存储系统的四个基本问题

存储系统的构成是基于程序访问具有局部性(时间上和空间上)规律的这一原理。在这种层次结构的系统中,每上一层存储的信息都是其下一层的一个子集,因此,与之相关联的有四个问题:

① 当把某一层中的信息块调入上一层存储器时,应该存放在什么位置?这就是地址映像(映像规则)问题。

② 如何查找(或访问)到指定的存储层次中的内容,即地址变换问题。

③ 如果访问失效,将下一层信息块调入上一层已被占用的信息块位置时,应如何替换?即替换算法及其实现问题。

④ 当对某一信息块内容进行修改时,应该进行哪些操作?即相应的写策略问题。

4.1.2 存储系统的性能指标

为了方便起见,我们以二级存储系统为例说明存储系统的性能指标。两级存储

器 M_1 和 M_2 的访问时间分别为 T_{A1} 和 T_{A2}，容量分别为 S_{M1} 和 S_{M2}，单位价格分别为 C_1 和 C_2，很显然，它们之间有 $T_{A1} < T_{A2}$，$S_{M1} < S_{M2}$，$C_1 > C_2$，如图 4.2 所示。

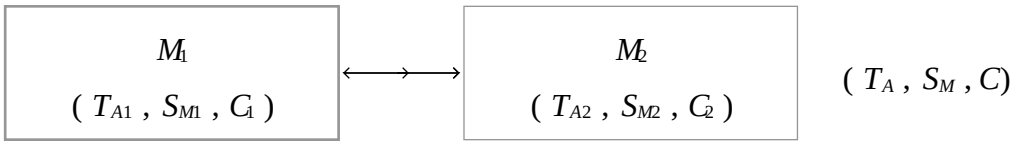


图 4.2　二级存储系统

1.每位平均价格

两级存储系统的每位平均价格可以表示为：

$$C = \frac{C_1 S_{M1} + C_2 S_{M2}}{S_{M1} + S_{M2}}$$

若希望存储系统的平均每位价格能接近于 C_2，就应使 S_{M1} n S_{M2}。但是，如果 S_{M1} 与 S_{M2} 相差太大，则会使得对 M_1 的命中率很低。同时，上式没有把由于采用存储系统所必须增加的辅助软、硬件价格计算在内，所以要使 C 接近于 C_2，还应使增加的辅助软、硬件价格只占价格中很小的比例，否则存储系统的性能价格比将会显著降低。

2.命中率

命中率可简单地定义为 CPU 访问存储系统时在存储器 M_1 中访问到指定信息的概率。命中率可用实验或模拟方法来获得，即执行或模拟一段有代表性的程序，若逻辑地址流指定的信息能在 M_1 中被访问到的次数为 N_1，在 M_1 中未被访问到的次数为 N_2，则命中率为：

$$H = \frac{N_1}{N_1 + N_2}$$

命中率 H 与程序的访存地址流、所采用的地址映像关系、M_1 中的块或页被替换的算法及 M_1 的容量都有很大的关系。显然，H 越接近于 1 越好。

为了突出反映不命中的情况，我们还经常使用不命中率或失效率 F 这个参数，它是指 CPU 访存时，在 M_1 中找不到所需信息的概率。显然：

$$F = 1 - H$$

3.平均访问时间 T_A

分两种情况来考虑 CPU 的一次访存：

① 当命中时，访问时间即为 T_{A1}。T_{A1} 常称为命中时间。

② 当不命中时，必须依据二级存储层次结构原理的不同，分别予以考虑。在大多数二级存储层次中，若访问的字不在 M_1 中，就必须从 M_2 中把包含所请求的字的信息块传送到 M_1 之后，CPU 才能在 M_1 中访问到这个字。假设传送一个信息块所需的时间为 T_B，则不命中时的访问时间为：

$$T_{A2} + T_B + T_{A1} = T_{A1} + T_M$$

其中，$T_M = T_{A2} + T_B$，它为从向 M_2 发出访问请求到把整个数据块调入 M_1 中所需的时间。T_M 常称为失效开销。

$$T_A = HT_{A1} + (1 - H)(T_{A1} + T_M) = T_{A1} + (1 - H)T_M$$

或

$$T_A = T_{A1} + FT_M$$

4 .访问效率

二级存储系统的访问效率为：

$$e = \frac{T_{A1}}{T_A} = \frac{T_{A1}}{HT_{A1} + (1 - H)T_{A2}} = \frac{1}{H + (1 - H)T_{A2}/T_{A1}}$$

可见，存储系统的访问效率主要与命中率和构成存储系统的两级存储器的访问时间比有关。

4. 1. 3 “Cache—主存”和“主存—辅存”层次

“Cache—主存”和“主存—辅存”层次是常见的两种层次结构，几乎所有当代计算机都同时具有这两种层次。我们知道，程序在执行前需先调入主存(在虚拟存储器中也是如此，只是不必一次全部调入)。因此，下面我们将从主存的角度来讨论这两个存储层次。

1 .“Cache—主存”层次

近十多年来，CPU 的性能提高得很快，在 1980～1986 年之间，CPU 以每年 35% 的速度递增，而从 1987 年开始，CPU 性能则是每年提高 55%(见图 4 .3)。但是主存储器性能的提高却慢得多。例如，DRAM 的速度每年只提高 7%。因此，CPU 和主存之间在性能上的差距越来越大。现代计算机都采用 Cache 来解决这个问题。图 4 .4(a)是“Cache—主存”层次的示意图。这是在 CPU 和主存之间增加一级速度快，但容量较小且每位价格较高的高速缓冲存储器(Cache)。它与主存构成一个有机的整体，以弥补主存速度的不足。这个层次的工作主要由硬件实现。

2 .“主存—辅存”层次

引入“主存—辅存”层次的目的是为了弥补主存容量的不足。它是在主存外面增加一个容量更大、每位价格更低、但速度更慢的存储器，称为辅存(一般由联机硬盘系统构成)。它们依靠辅助软硬件的作用，使之构成一个整体，如图 4 .4(b)所示。“主存—辅存”层次常被用来实现虚拟存储器，向编程人员提供大量的程序空间。

3 .两种存储层次的比较

表 4 .1 对“Cache—主存”和“主存—辅存”两种存储层次做了一个简单的比较。

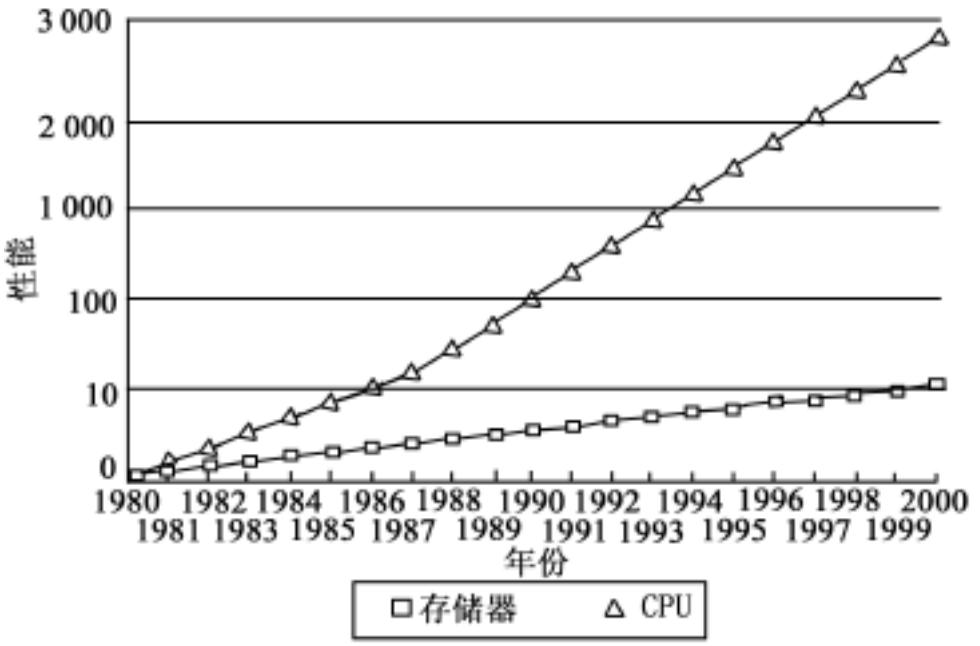
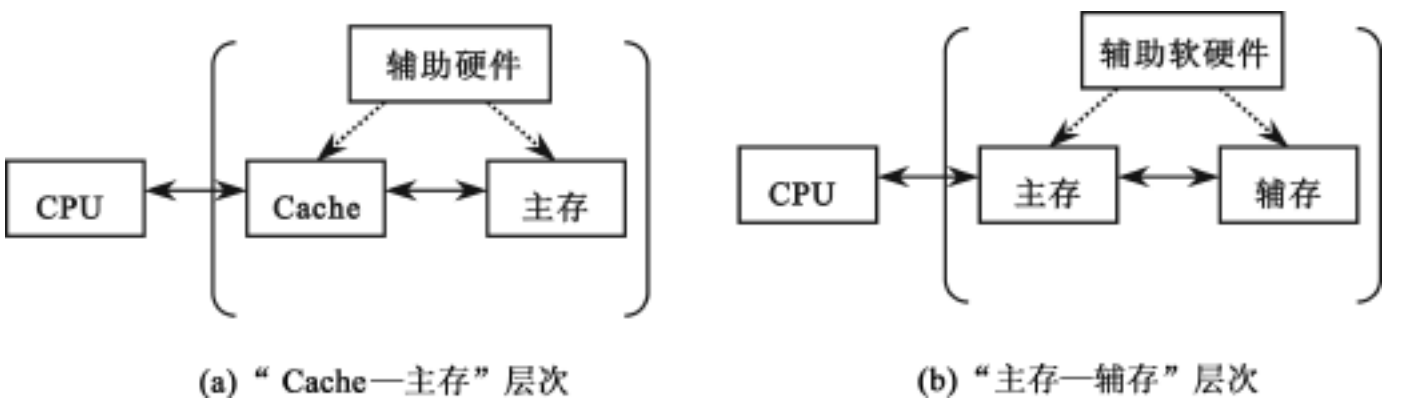


图 4 .3　1980 年以来存储器和 CPU 性能随时间而提高的情况(以 1980 年时的性能作为基础)



(a)“ Cache—主存”层次　　(b)“主存—辅存”层次

图 4 .4　两种常用的存储层次

表 4 .1　　“ Cache—主存 ”和“ 主存—辅存 ”层次的比较

比较项目 \ 存储层次	“ Cache—主存 ”层次	“ 主存—辅存 ”层次
目　　的	为了弥补主存速度的不足	为了弥补主存容量的不足
存储管理实现	主要由专用硬件实现	主要由软件实现
访问速度的比值(第一级比第二级)	几比一	几万比一
典型的块(页)大小	几十个字节	几百到几千个字节
CPU 对第二级的访问方式	可直接访问	均通过第一级
失效时 CPU 是否切换	不切换	切换到其他进程

4.1.4 主存频宽的平衡与提高

现代计算机是以存储器为中心工作的,这里所指的存储器是广义的,不仅是主存(内存),也包括所有的存储器。一般计算机中的存储器有四个访问源:CPU 取指令、CPU 读取操作数、CPU 写运算结果和各种 I/O 设备的访问。计算机系统结构设计者的一项重要工作就是使计算机系统中的各级存储器的频宽达到平衡。

假设一台普通的计算机,处理速度为 200 MIPS。那么各种访问源的频宽如下:

① CPU 读存储器取指令:200MW/S(设每条指令平均长度为 1W 长)。

② CPU 访问存储器读/写操作数和结果:400MW/S(平均每条指令访问两个操作数)。

③ 各种 I/O 设备访问:5MW/S。

以上三项相加,要求存储器的频宽不低于 605MW/S,这就是频宽平衡。

那么,如何提高存储器的频宽,以保证计算机系统的频宽,满足上述要求,达到平衡呢?一般说来有三种途径,简单介绍如下:

① 采用存储系统,特别是 Cache 存储系统,这是目前计算机中提高主存速度最有效的一种方法。有关内容在 4.3 节中介绍。

② 设置各种缓冲存储器。如取指令缓冲栈、操作数先行读数栈、运算结果后行缓冲栈等。有关内容在第五章中介绍。

③ 设计并行访问存储器。在主存储器设计中,让多个存储器并行交叉地工作,以提高其主存频宽。下面介绍几种并行访问存储器。

1.单体多字并行存储器

常规的存储器一个存储单元只能存放一个存储字 W,一个存储周期只能访问 W 位二进制信息,这种存储器称之为单体单字存储器,如图 4.5(a)所示。

如果将存储器的字长扩大 m 倍,就可以在一个存储周期内同时访问到 mW 位内容,由频宽 $B_m = m\frac{W}{T_M}$ 可知,理想值可提高到 m 倍。这种存储器称为单体多字存储器,如图 4.5(b)所示。

单体多字并行存储器的优点是实现简单,缺点是访问冲突概率大。访问冲突主要来自以下几个方面。

① 取指令冲突。单体多字并行存储器一次取出 m 个指令字,能很好地支持程序的顺序执行。但是,若一个存储字中有一条转移指令字时,那么存储字中转移指令后面被同时预取的几个指令字只能作废。

② 读操作数冲突。单体多字并行存储器一次取出的 m 个数据字不一定都是要执行的指令所需要的操作数,而当前执行指令需要的全部操作数也可能不包含在一个存储字中而不能被一次取出。因为数据存放的随机性比程序指令存放的随机性

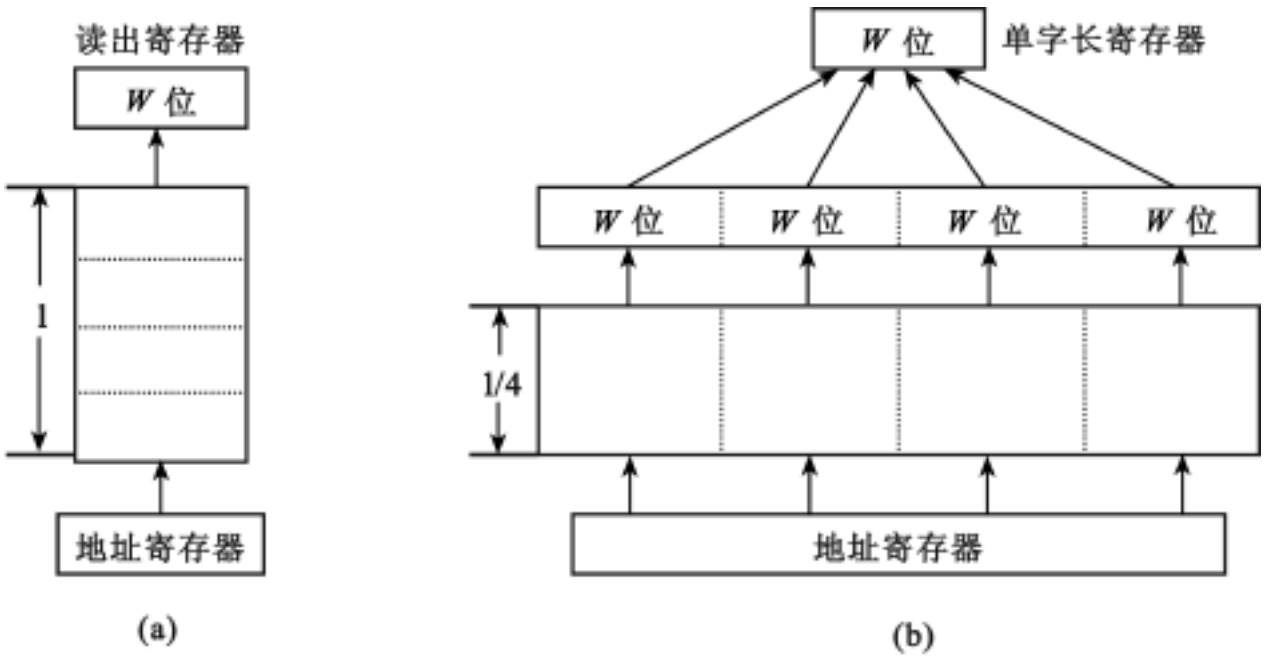


图 4 5　单体单字和单体多字存储器

大,所以读操作数冲突的概率较大。

③ 写数据冲突。单体多字并行存储器必须是凑齐了 m 个数据字之后才能作为一个存储字一次写入存储器。因此,需要先把属于一个存储字的 m 个数读到数据寄存器中,然后再把整个存储字写回存储器。

④ 读写冲突。当要读出的数据字和要写入存储器的数据字同处于一个存储字中时,读和写的操作就无法在同一存储周期中完成。

2 .交叉访问存储器

一个存储器通常对存储单元是顺序编址的。由多个存储体(存储模块)组成一个更大容量的主存时,对多个存储体的存储单元采用交叉编址方式,组成交叉访问存储器。交叉访问存储器通常有两种交叉编址方式,一是地址码的高位交叉编址,二是地址码的低位交叉编址。高位交叉编址存储器目前使用很普遍,这种编址方式能很方便地扩展常规主存的容量,但只有低位交叉编址存储器才能作为并行存储器的一种。

低位交叉访问存储器的结构如图 4 .6 所示。由 m 个存储体组成的低位交叉存储器的存储单元地址的低 $\log_2 m$ 位称为体号 k,高 $\log_2 n$ 位称为体内地址 j,存储单元地址 A 的计算公式为: $A = m \times j \times k$。若已知地址 A,可计算出对应的体号 $k = A \bmod m$,体内地址 $j = \lfloor A / m \rfloor$ 。

为了提高主存的速度,在一个存储周期内分时启动 m 个存储体,由于地址码低位交叉编址,因此对连续的地址访问将分布在不同的存储体中,避免了存储体访问冲突。理想情况下,存储器的速度可提高 m 倍。

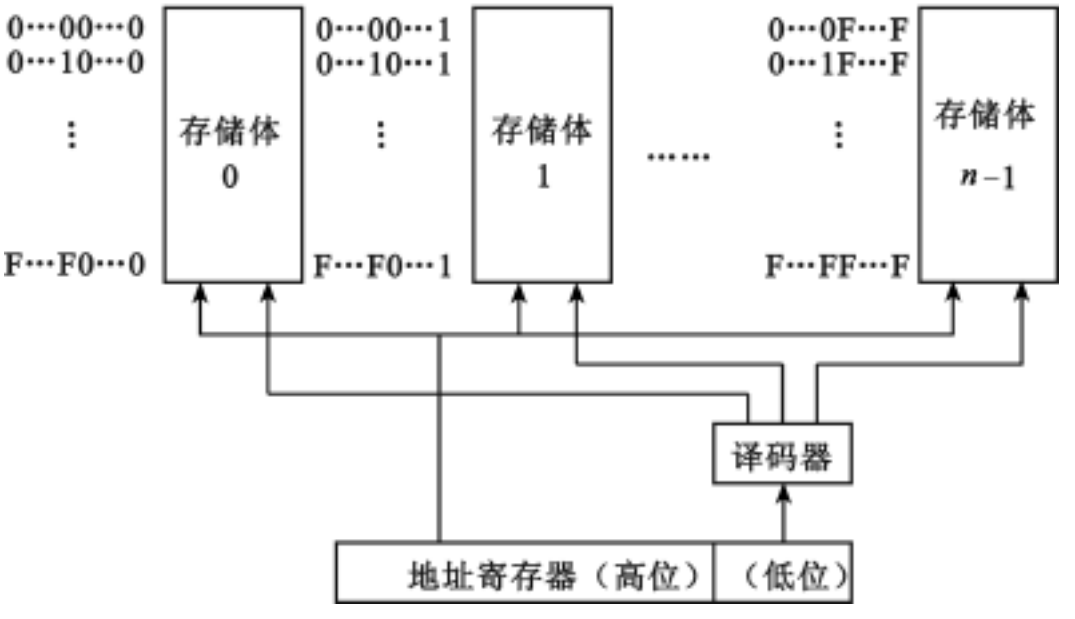


图 4.6 低位交叉访问存储器的结构

3.无访问冲突存储器

实际上,一个由 m 个存储体组成的低位交叉存储器的速度并不能提高 m 倍,其根本原因是存在存储体的访问冲突。产生访问冲突的根源主要有两个,一是程序中的转移指令,二是数据被访问的随机性,后者的影响更为严重。以一维数组和二维数组为例,介绍无冲突访问存储器。

(1)一维数组的无冲突访问

若采用低位交叉访问方式的并行存储器有 4 个存储体,交叉存放一维数组 a_0, a_1, a_2,…如图 4.7 所示。

	0 号体	1 号体	2 号体	3 号体
体内地址 0	a_0	a_1	a_2	a_3
1	a_4	a_5	a_6	a_7
2	a_8	a_9	a_{10}	a_{11}
3	…	…	…	…

图 4.7 一维数组的存储方案

如果每次都按连续地址对数组元素顺序访问,那么一个存储周期可以访问 4 个存储单元。若按位移量为 2 的变址方式访存(对下标为奇数或偶数的数组元素进行操作),则有一半的地址发生冲突,使存储器的频宽降低一半。若按位移量为 4 的变址方式访存,则情况就更糟。但若把存储体的个数 m 选为质数,变址位移量与 m 互

质,那么,一维数组的访问冲突就不存在了。

许多以向量计算为主要任务的大型计算机系统,其主存储器的存储体个数一般都是质数。例如,我国研制的银河计算机,存储体的个数为 31。美国 Burroughs 公司研制的科学处理机 BSP 的存储体个数为 17。

(2)二维数组的无冲突访问

假设一个 $n\times n$ 的二维数组存储在一个并行存储器中。现在要求对这个二维数组实现按行、按列、按对角线和按反对角线访问,并且在不同的变址位移量情况下,都能实现无冲突访问。

现以 4×4 的二维数组为例来讨论这个问题。按照地址顺序存储各行元素,一个 4×4 二维数组存储在 4 个存储体中的情况如图 4 .8 所示。显然,按行、按对角线访问都不发生访问冲突,但按列访问时,由于同一列的 4 个元素在同一个存储体内,就产生了对存储体的访问冲突,只能用 4 个存储周期顺序读取。

	0 号体	1 号体	2 号体	3 号体
体内地址 0	a_0	a_1	a_2	a_3
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}
3	a_{30}	a_{31}	a_{32}	a_{33}

图 4 .8　一维数组的存储方案

P·Budnik 和 D·J·Kuck 提出了一种能够对 $n\times n$ 二维数组实现上述要求的无冲突存储方案。

并行存储体实现 $n\times n$ 二维数组无冲突访问方案要求:

条件一:并行存储体的个数 m 大于并行访问元素个数 n,要求 m 取质数,一般取 $m=n+1$。

条件二:行、列方向的元素错位存放,即同一列相邻两元素错开的距离为 d_1,同一行相邻两元素错开的距离为 d_2,当 $m=2^{2P}+1$ 时,$d_1=2P$,$d_2=1$。

以 4×4 的二维数组为例,取大于 4 的最小质数 $m=5$ 作为并行存储体的个数,并把 m 代入关系式 $m=2^{2P}+1$,得 $p=1$,计算得:$d_1=2$,$d_2=1$。一个 4×4 的二维数组在 5 个存储体中错位存储的存储情况如图 4 .9 所示。由图 4.9 可见,二维数组按行、列、对角线和反对角线访问都不会发生存储体的访问冲突。

	0 号体	1 号体	2 号体	3 号体	4 号体
体内地址 0	a_{00}	a_{01}	a_{02}	a_{03}	…
1	a_{13}	…	a_{10}	a_{11}	a_{12}
2	a_{21}	a_{22}	a_{23}	…	a_{20}
3	…	a_{30}	a_{31}	a_{32}	a_{33}

图 4.9 二维数组无访问冲突的存储方案

$n\times n$ 二维数组中的任意元素 a_{ij} 在无冲突并行存储器中的体号和体内地址可以通过如下的一般公式来计算:

体号地址 $= (2^P \cdot i + j + k) \bmod m$

体内地址 $= i$

其中,$0 \le i \le n-1$, $0 \le j \le n-1$, k 是数组的第一个元素 a_{00} 所在存储体的体号,一般取 $k=0$;存储体的个数 m 是大于等于 n 的质数;p 是满足 $m = 2^{2p}+1$ 的任意自然数。

对于行列不相等的二维数组,如何实现无冲突访问的存储方案呢?

方案是:首先将二维数组按一维线性排列,并给出地址 a;然后分别求出每个元素存储的体号地址和体内地址。

体号地址: $k = a \bmod m$ (m 为存储体的个数)

体内地址: $j = [d \quad n]$ (n 为并行访问元素的个数)

例如,对于 4×5 的二维数组 B

$$B = \begin{Bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \end{Bmatrix}$$

取存储体个数 $m=7$,并行访问元素的个数 $n=6$,则它们之间的对应关系如表 4.2 所示:

表 4.2 数组元素与各地址的对应关系

数组元素	b_{00}	b_{01}	b_{02}	b_{03}	b_{04}	b_{10}	b_{11}	b_{12}	B_{13}	b_{14}
地址 a	0	1	2	3	4	5	6	7	8	9
体号 k	0	1	2	3	4	5	6	0	1	2
体内地址 j	0	0	0	0	0	0	1	1	1	1
数组元素	b_{20}	b_{21}	b_{22}	b_{23}	b_{24}	b_{30}	b_{31}	b_{32}	b_{33}	b_{34}

续表

地址 a	10	11	12	13	14	15	16	17	18	19
体号 k	3	4	5	6	0	1	2	3	4	5
体内地址 j	1	1	2	2	2	2	2	2	3	3

根据表 4.2 所指示元素的体内地址的体号地址,二维数组的并行无访问冲突的存储方案如图 4.10 所示。

	0 号体	1 号体	2 号体	3 号体	4 号体	5 号体	6 号体
体内地址 0	b_{00}	b_{01}	b_{02}	b_{03}	b_{04}	b_{10}	
1	b_{12}	b_{13}	b_{14}	b_{20}	b_{21}		b_{11}
2	b_{24}	b_{30}	b_{31}	b_{32}		b_{22}	b_{23}
3					b_{33}	b_{34}	

图 4.10　4×5 二维数组并行访问无冲突存储方案

4. 按内容访问的存储器

按内容访问方式与按地址访问方式不同,这种访问方式并不提供欲被访问存储单元的地址,而是给出欲被访问的内容。显然采用这种访问方式,存储器的结构形式就要作相应的变化。为了加快访问速度,必须采用并行访问方式,而相应的存储器称为联想存储器,由于这种存储器的价格比较昂贵,因此其容量通常不可能做得很大。其主要特点是以并行方式查找所需信息内容。联想存储器的基本结构,如图 4.11 所示。存储阵列有 n 个字,每个字 m 位。存储器中有比较寄存器 CR,用来存放要检索的数;屏蔽寄存器 MR,用来屏蔽不参加并行比较的位。以上两个寄存器长度均为 m 位。指示寄存器 IR(n 位)用来存放当前检索结果。当检索条件符合时,相应的 IR 的位就被置成“1”;否则仍保留初始值“0”。此外还有一个到多个暂存寄存器 TR(n 位),以存放前几次的检索结果。

要访问联想存储器时,先要在 CR 中存放好要检索的内容,由 MR 寄存器屏蔽掉那些不参加比较的部分。未经屏蔽欲进行检索的关键码与联想存储器所有单元中的相应部分进行并行比较。比较的结果可能会有好几个单元中相应内容满足比较条件,将所有符合条件的相应单元所对应的指示寄存器中的相应位置成“1”。下面通过一个例子来说明联想存储器的工作情况。假定在联想存储器中已存放了一张高校考生的登记表,如图 4.12 所示。现在要检索出所有考分大于或等于 520 分而又低于

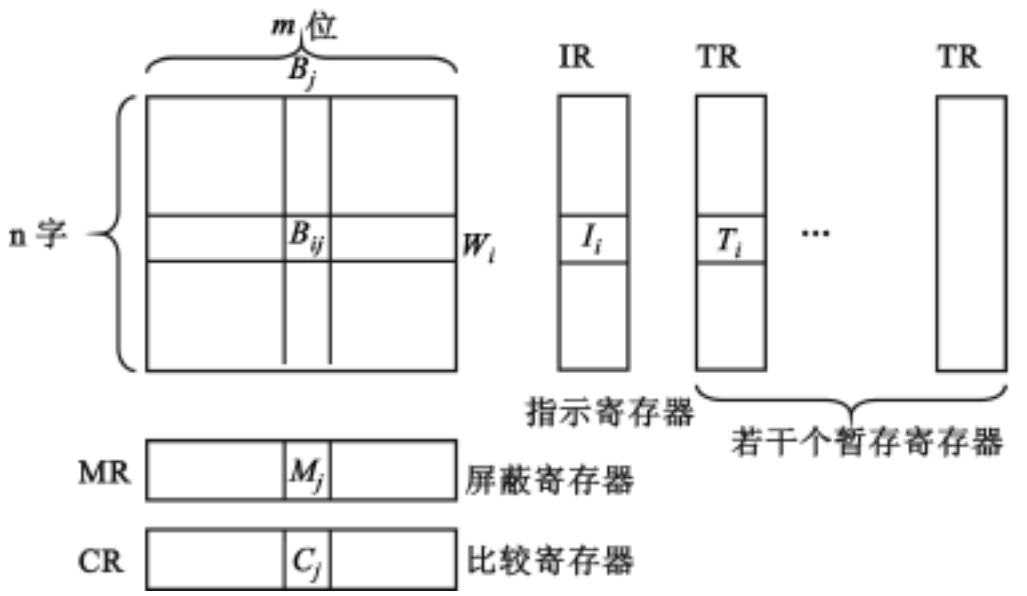
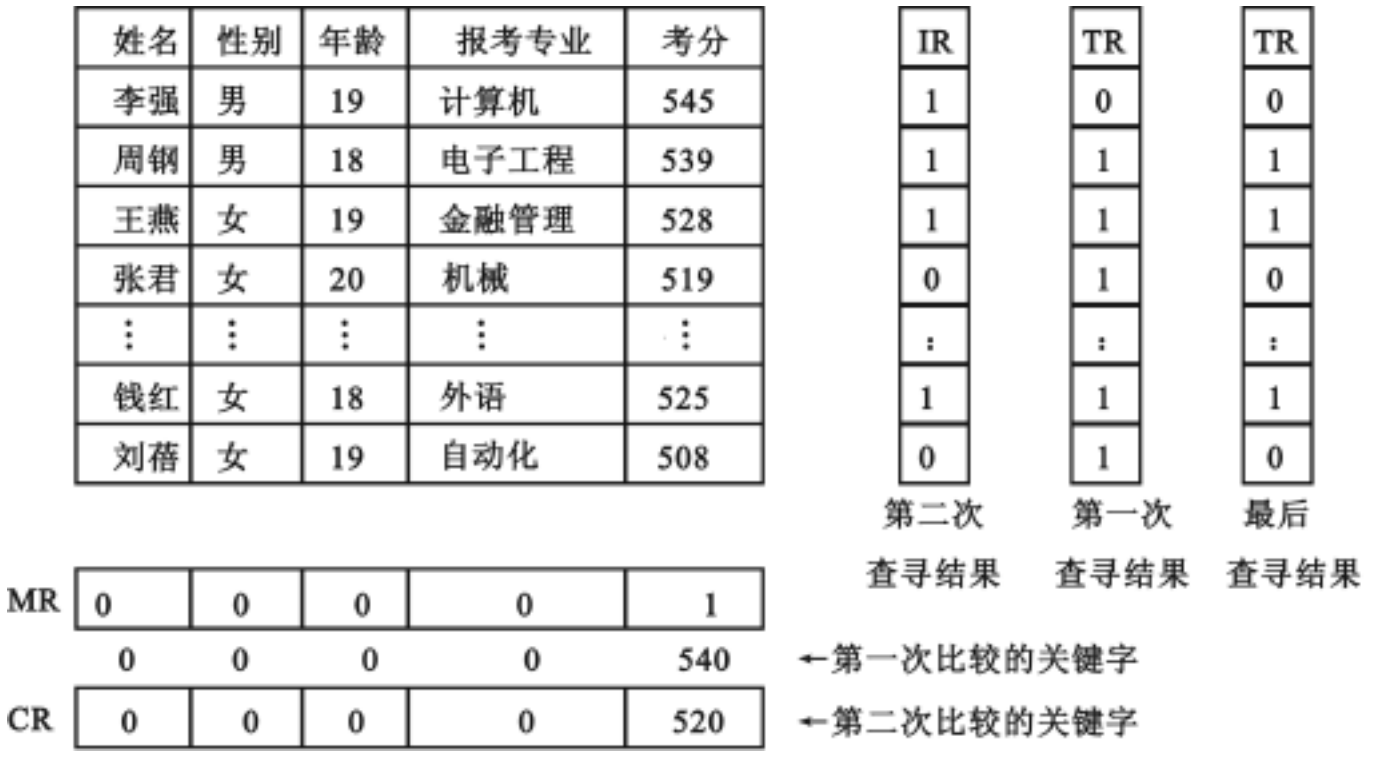


图 4.11 联想存储器的基本结构

540 分的考生名字。在进行这一特定的查询时,在 CR 中所设置的“考分”关键字是 540,与所有相应内容作小于比较,找出低于 540 分的所有考生,并在相应的 IR 寄存器的相应位置“1”,再将其送往 TR。接着进行第二次查询,将在 CR 中设置的“考分”关键字改为 520,然后作大于等于(或不小于)的查询比较,并将结果在 IR 寄存器的相应位作标志。最后把 IR 和 TR 中的相应内容作一次“与”操作,就可得到最后所需的查询结果。凡相应位带标志“1”的考生应在输出名单中(这里是周钢,王燕和钱红三位同学)。



姓名	性别	年龄	报考专业	考分
李强	男	19	计算机	545
周钢	男	18	电子工程	539
王燕	女	19	金融管理	528
张君	女	20	机械	519
⋮	⋮	⋮	⋮	⋮
钱红	女	18	外语	525
刘蓓	女	19	自动化	508

IR	TR	TR
1	0	0
1	1	1
1	1	1
0	1	0
:	:	:
1	1	1
0	1	0
第二次查寻结果	第一次查寻结果	最后查寻结果

图 4.12 联想存储器用于学生高考文档的存储和检索

在联想存储器中由于要求每个基本存储单元(图 4 .11 中的 B_{ij}),具有比较功能,因此相应的存储单元的电路及相互间的连线要比一般的存储器中的复杂得多,因此设计复杂,成本较高,特别当存储容量较大时。在早期的存储系统中联想存储器用做地址变换的旁视缓冲器(Translation Lookaside Buffer, TLB),容量都比较小。但随着 VLSI 技术的发展,联想存储器已开始得到更多的应用。如在 Intel Pentium 处理器中用来构成转移目标缓冲器(Branch Target Buffer, BTB)。

实用的联想存储器,一般除有按内容访问能力外,还有按地址访问的能力。故仍保留有地址寄存器、译码电路和读写寄存器。此外,联想存储器的每个基本单元除了有存储能力和相等比较功能外,还可实现≠, <, ≤, ≥, MAX, MIN, BETWEEN, NEXT, HIGHER, NEXT—LOWER 等比较功能。

4 .2　虚拟存储器

虚拟存储器最早是由英国曼彻斯特大学的 Kilbrn 等人提出的,并于 1961 年在该校 Atras 计算机上予以实现。20 世纪 70 年代以来,这一技术不仅被中、大型计算机采用,而且在当今的微型计算机上都普遍采用。那么,什么是虚拟存储器的技术呢?

一个高级语言程序经过编译后生成目标代码程序。目标代码程序只有装入内存,程序才能执行。我们把目标代码程序指令和数据占用的地址空间称为程序地址空间,或称为逻辑地址空间。将主存储器的存储空间称为物理地址空间,或称为实存地址空间。

虚拟存储器是主存容量的扩展,它是借助于磁盘等辅助存储器扩大主存容量,是一个容量非常大的存储器的逻辑模型,不是任何实际的物理存储器。

引入虚拟存储器的基本思想是,实现多用户软件共享宝贵的主存资源。虚拟存储器技术的实现,是基于程序访问的局部性原理的。

4. 2. 1　虚拟存储器的管理方式

在虚拟存储器中有三种地址空间:一是虚拟地址空间,它是由应用程序员编程时使用的地址空间;二是主存储器的地址空间;三是联机辅存地址空间。与这三种地址空间相对应的有三种地址,即虚拟地址(虚存地址、虚地址)、主存地址(物理地址、实地址)、辅存地址(磁盘存储器地址)。

虚拟存储器管理的主要任务是利用软、硬件的方法,实现虚拟地址空间到物理地址空间(包括辅存地址空间)的映像和变换。

地址映像就是把多用户的虚拟地址编写的程序按照某种规则装入主存储器中,并建立多用户虚地址与主存实地址之间的对应关系。

地址变换是在程序被装入主存储器之后,在实际运行时,把多用户的虚地址变换

成主存实地址(内部地址变换)或磁盘存储器地址(外部地址变换)。

根据虚拟存储器所采用的地址映像和地址变换方法不同,则有多种不同的虚拟存储器。常用的有三种,即段式虚拟存储器、页式虚拟存储器和段页式虚拟存储器。下面分别介绍这三种虚拟存储器。

1 .段式虚拟存储器

(1)分段原理

根据程序在结构上都具有相对独立的模块化这一特点,可将用户程序分成若干段,每段以零为起始地址编址。如主程序段、公共子程序段、数据段、表格段、向量段等。

(2)地址映像

在段式虚拟存储器的管理中,是利用段表(Segment Table)来建立地址映像关系的。例如某用户 A 的程序由 4 段组成,各段长度及映像如图 4 .13 所示。

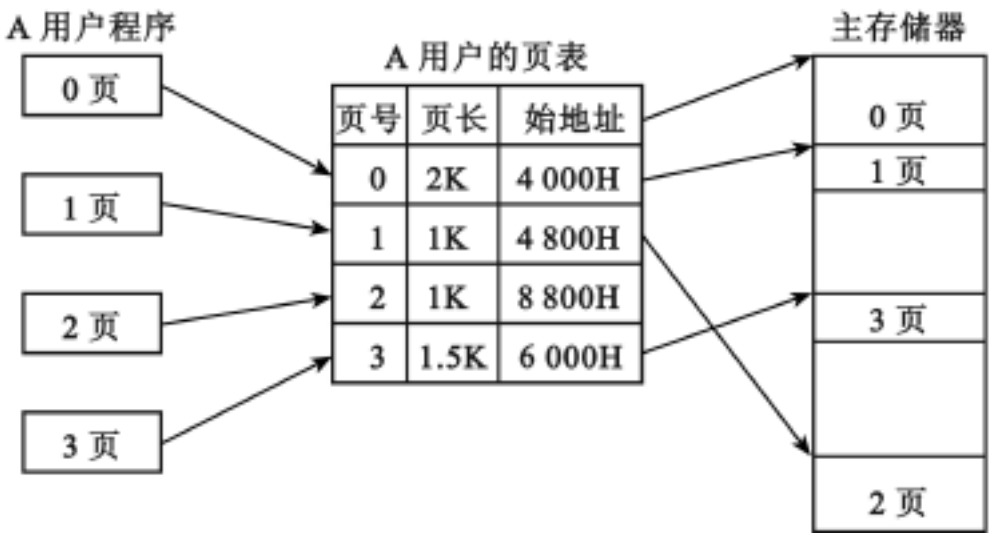


图 4 .13 段式虚拟存储器的地址映像

A 用户程序共有四个逻辑段组成,在一张段表的控制下,将它们分别映像到主存储器的各个不同区域中。实际上每个程序段可以映像到主存储器的任意位置,它们可以连续存放,也可以不连续存放,可以顺序存放,也可以前后倒置存放。

(3)地址变换

有了地址映像,可以把在虚拟地址空间中编写的程序装入主存储器中。但在程序实际执行时,还要把用来访问主存储器的多用户虚地址变换成主存实地址,才能访问已经装在主存储器中的用户程序或数据,地址变换过程如图 4 .14 所示。

一个多用户的虚地址由三部分组成,用户号 U、段号 S 和段内偏移 D。地址变换过程可描述如下:

① 查段表基址寄存器。根据多用户的虚地址中的“用户号”字段,查找段表基址

寄存器(段表基址寄存器通常设置在 CPU 内部,每个用户程序使用其中的一个基址寄存器),找到该用户程序的段表在主存中的首地址。

② 查段表。每个用户程序在主存中都设置有一个段表,段表的项(行)数等于用户程序的逻辑段数。由段表基地址加上虚地址中的“段号”,就可得到所要访问程序段的全部信息所在的地址。

③ 形成访存实地址。如果段表中的装入位为“1”,表示该段程序已经装入主存。这时,取出段表中该段程序在主存中的“起始地址”加上虚地址中的“段内偏移”,从而形成访问主存的实地址。

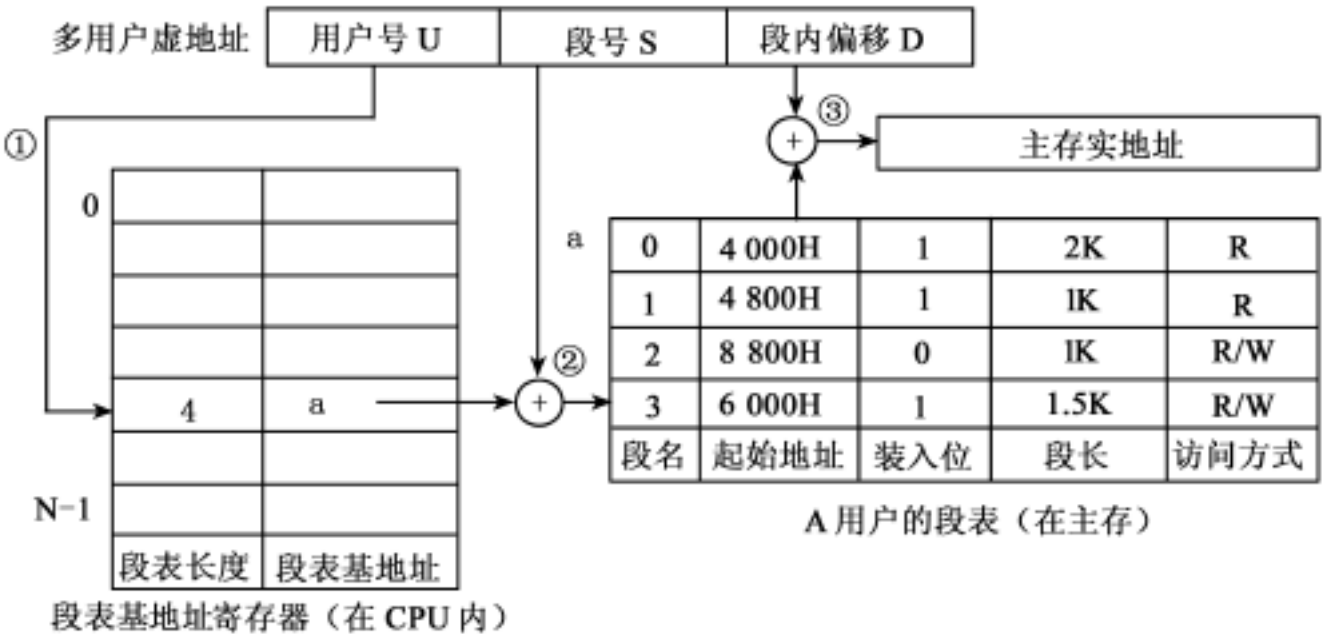


图 4 .14　段式虚拟存储器的地址变换

段表中的段长和访问方式是用来保护程序段的。可以根据程序段的起始地址和段长计算出本次访问主存储器的地址是否越界。访问方式可以指出本程序段是否需要保护和保护的级别。例如,对于子程序段,通常只能执行,不能改写;对于一些常数段或数据库中的数据段,一般用户程序只能读,不能改写,不能执行;对于有些需要保密的表格,一般用户应禁止访问等。

如果装入位给出的信息表示要访问的这个程序段不在主存储器中,则段表中的起始地址和访问方式字段等均无用。这时,可以把它们用来存放该程序段在磁盘存储器中的起始地址等信息。使用磁盘存储器的起始地址和段长就可以从磁盘存储器中把该程序段读到主存储器中。

根据需要还可以在段表中增加其他字段,例如,增加一个修改标志字段,表示本程序段是否被修改过。如果这个程序段从装入主存储器起一直没有被修改过,则在需要把它替换出主存储器时,不必把这个程序写回到外部的磁盘存储器中,只要用新调入的程序段把它覆盖掉即可。如果这个程序段被修改过,则必须先把这个程序段

全部写回到磁盘存储器中存放这个程序段的原来位置上。

段表本身也是一个段,一般常驻在主存储器中。如果段表太长,也可以把暂时不用的一部分段表放在磁盘存储器中,当需要时再把有用的段表调入主存储器。

段式虚拟存储器的主要优点如下:

① 程序的模块化性能好。对于大程序,可以划分成多个程序段,每个程序段赋予不同的名字,由多个程序员并行编写,分别编译和调试,从而可以缩短程序的编制和调试时间。

② 便于程序和数据的共享。当某个程序段需要被共享时,只要在主存储器中装入一份,同时在需要调用这个程序段的那些程序(或用户)被共享时,在需要调用这个程序段的那些程序(或用户)的段表中都使用这个程序段的主存起始地址和段长等信息,就能很方便地实现程序段的共享。

③ 便于实现信息保护。在一般情况下,一段程序是否需要保护是根据这个段程序的功能来决定的。由于段式虚拟存储器本身就是按照功能划分程序段的,因此,只要在段表中设置一个信息保护字段,就能根据需要很方便地实现对该程序段的保护。

段式虚拟存储器的主要缺点如下:

① 地址变换所花费的时间比较长。从图 4.14 中可以看到,从多用户虚地址变换到主存实地址需要查两次表,做两次加法运算。

② 主存储器的利用率往往比较低。由于每个程序段的长度是不同的,程序段在主存储器不断地调入、调出,有些程序段在执行过程中还要动态地增加长度,从而使得主存储器中有很多的空隙存在。

③ 对辅存(磁盘存储器)的管理比较困难。磁盘存储器通常是按固定大小的块来访问的,如何把不定长度的程序段映像到固定长度的磁盘存储器中,需要做一次地址变换。

2.页式虚拟存储器

(1)分页原理

页式虚拟存储器把虚拟地址空间和主存地址空间等分成大小相同的页。页是一种逻辑上的划分,它可以由系统管理软件任意指定。然而,由于磁盘存储器的物理块大小是 0.5KB,为了与外部存储器,特别是磁盘存储器相配合,虚拟存储器中页的大小通常指定为 0.5KB 的整倍数。目前在一般计算机系统中,一页的大小通常为 1KB~16KB。

(2)地址映像

在页式虚拟存储器中,虚拟地址空间的页称为虚页(页面),主存地址空间的页称为实页(页框)。将虚拟空间到主存空间的映像关系,转换为虚页与实页的对应关系。所以,在页式虚拟存储器中,映像是由“页表”来完成的。例如,某用户程序空间为 15KB,每页大小为 4KB,则虚页为 4 页,如图 4.15 所示。

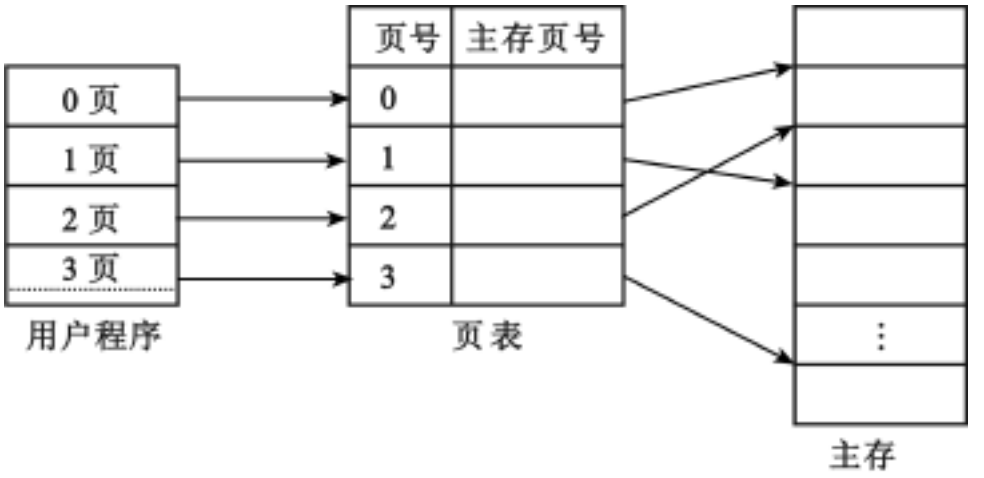


图 4 .15　页式虚拟存储器的地址映像

与段式虚拟存储器相比,由于每一页的长度是固定的,因此,不需要像段式虚拟存储器中的段长度这一字段。另外,主存地址这一字段只需要指出主存储器的页号,与段式虚拟存储器中的主存地址必须指出整个主存地址长度相比要节省很多。

(3)地址变换

在 CPU 内部有一个页表基址寄存器,用来存放多用户页表在主存中的基地址。每个用户(每道程序)使用其中一个基址寄存器。

多用户的虚地址由三部分组成:用户号 U,虚页号 P 和页内偏移 D。页式虚拟存储器的地址变换过程如图 4 .16 所示,并可描述为以下三个过程。

① 查页表基址寄存器。根据多用户的虚地址中的用户号查页表基址寄存器,从中读出这个用户程序的页表在主存中的起始地址。

② 查页表。根据页表的首地址,再加上用户虚页号,就能得到被访问页的所有信息,包括该页是否已装入主存,对应主存实页号以及其他与访问有关的信息。

③ 形成实地址。如果被访问的页已经装入主存,则根据页表中的实页号 P 与多用户虚地址中的页内偏移 D 直接拼接起来构成访问主存的实地址。

页式虚拟存储器的主要优点如下:

① 主存储器的利用率比较高。每个用户程序只有不到一页(平均为半页)的浪费,与段式虚拟存储器每两个程序段之间都有浪费相比要节省许多。

② 页表相对比较简单。它需要保存的字段数比较少,一些关键字段的长度与段式相比要短许多,因此,节省了页表的存储容量。

③ 地址变换的速度比较快。在把用户程序装入到主存储器的过程中,只要建立用户程序的虚页号与主存储器的实页号之间的对应关系即可,不必使用整个主存的地址长度,也不必考虑每页的长度等。在地址变换过程中,从图 4 .16 中可以看到,主存实地址 A 是由页表中的主存页号 P 和多用户虚地址中的页内偏移 D 直接拼接而得到的,不必经过任何运算,因此,地址变换的速度比较快。

④ 对辅存(磁盘存储器)的管理比较容易。因为页的大小一般取磁盘存储器物

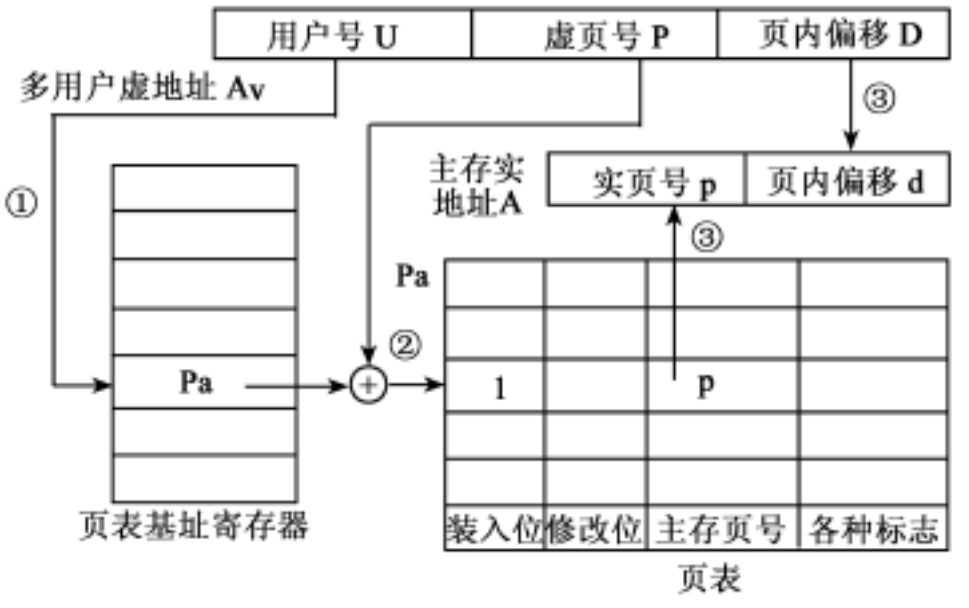


图 4 .16　页式虚拟存储器的地址变换

理块大小(512 字节)的整倍数。

页式虚拟存储器的缺点主要有两个:

① 程序的模块化性能不好。由于用户程序是强制按照固定大小的页来划分的,而程序段的实际长度一般是不固定的。因此,页式虚拟存储器中一页通常不能表示一个完整的程序功能。一页可能只是一个程序段中的一部分,也可能在一页中包含了两个或两个以上的程序段。

② 页表很长,需要占用很大的存储空间。通常,虚拟存储器中的每一页在页表中都要占有一个存储字。假设有一个页式虚拟存储器,它的虚拟存储空间大小为 4GB,每一页的大小为 1KB,则页表的容量为 4MB 存储字。如果每个页表存储字占用 4 个字节,则页表的存储容量为 16MB。

3 .段页式虚拟存储器

(1)分段分页原理

段式虚拟存储器和页式虚拟存储器各有其优点和缺点,段页式虚拟存储器是它们二者的结合。它将用户的虚拟存储空间按逻辑模块分成若干段,每段又分成大小相同的页。段的长度必须是页长的整倍数,段的起点必须是某一页的起点。每页的大小仍然是 0 .5KB 的整倍数。

一个用户程序由三个独立的程序段组成。0 号程序段的长度为 12KB,由于页的长度是 4KB,因此,正好分成 3 页。1 号程序段的长度为 10KB,也分成 3 页,其中最后一页有 2KB 是浪费的。2 号程序段的长度为 5KB,分成 2 页,其中后面一页浪费 3KB。

(2)地址映像

段页式虚拟存储器的地址映像是由段表和页表共同完成的。在图 4 .17 中,一个用户程序的三个程序段通过一张段表来控制。与段式虚拟存储器一样,每个程序段在段表中占一行。在段表中给出该程序段的页表长度和页表的起始地址,根据这两个参数就能找到这个程序段的页表。页表的长度就是这个程序段的页数,页表中给出这个程序段的每一页在主存储器中的实页号。这样就完成了用户程序的虚拟空间到主存实地址空间的映像。

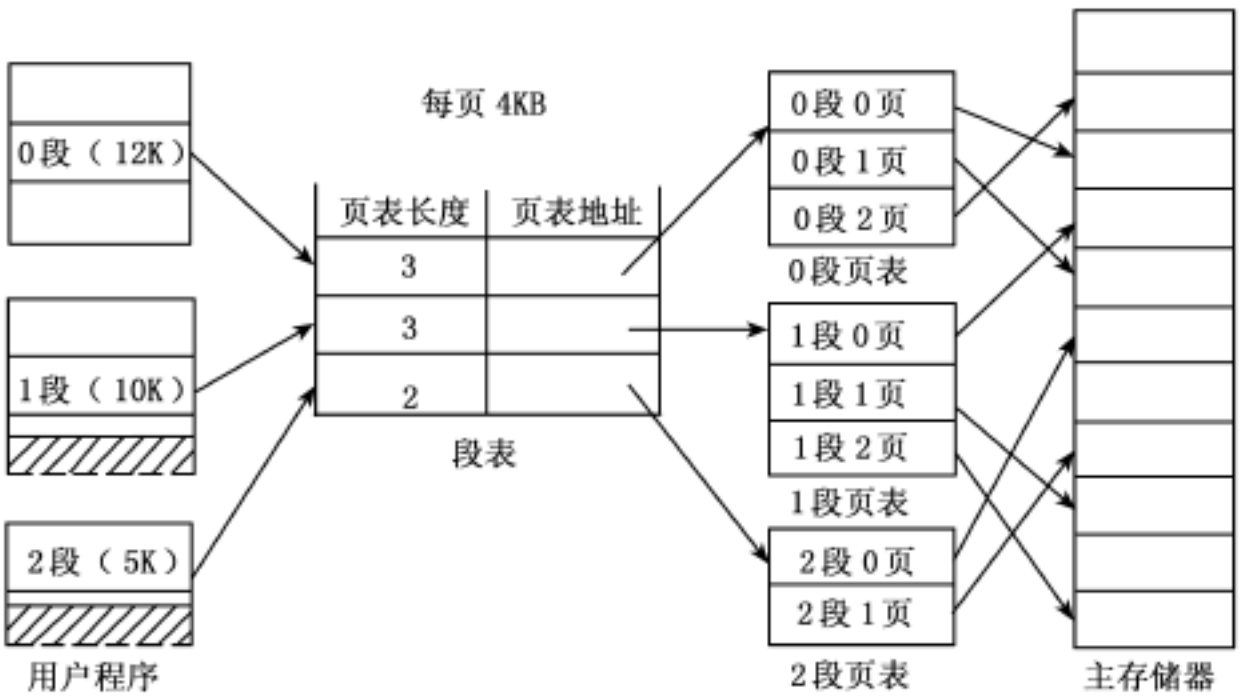


图 4 .17　段页式虚拟存储器的地址映像

(3)地址变换

在段页式虚拟存储器中,一个多用户虚拟地址由四部分组成:用户号 U、段号 S、虚页号 P 和页内偏移 D,见图 4 .18。在程序运行过程中,要把用户程序中的虚拟地址变换成实页号 p 和页内偏移 d。在程序运行过程中,要把用户程序中的虚拟地址变换成主存实地址,必须在 CPU 内部设置段表基址寄存器组,每个用户占用一个基址寄存器,同时在主存中要建立多用户段表和多用户页表。

段页式虚拟存储器的地址变换过程,如图 4 .18 所示。

① 查段表基址寄存器。根据多用户的虚地址中的用户号 U,在段表基址寄存器中找到该用户段表在主存储器中的基地址 a。

② 查段表。由段基址加上虚地址中的段号 S,形成该段程序页表在主存储器中的基地址。

③ 查页表。由页表基地址加上虚地址中的虚页号 P,找到所访问页在页表中的相应项。

④ 形成访问实地址。由页表中的实页号(设该页已经装入主存)与虚地址中的页内偏移 D 拼接形成物理地址。

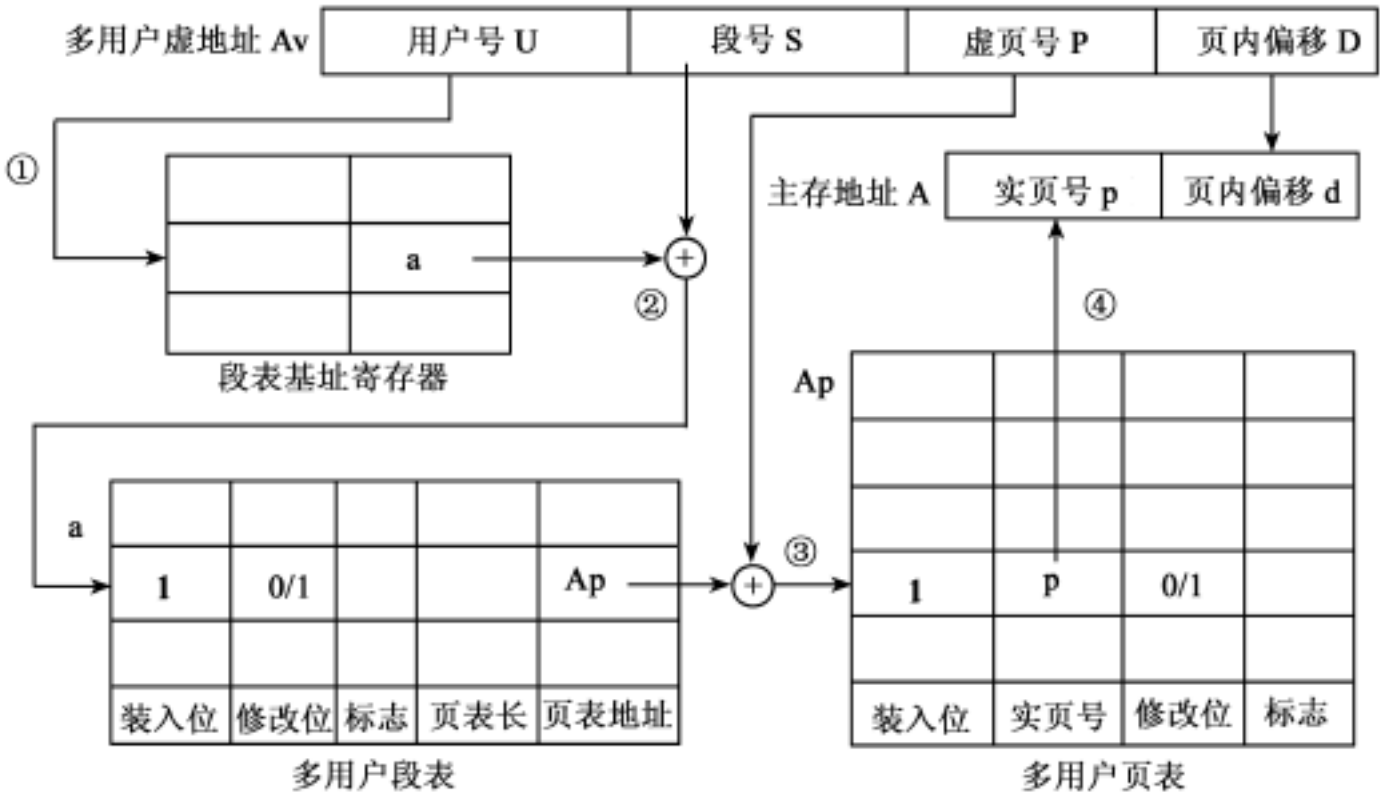


图 4 .18　段页式虚拟存储器的地址变换

由图 4 .18 可以看到,在段页式虚拟存储器中,要从主存储器中访问一个数据(取指令、读操作数或写结果),需要查两次表,一次是页表,另一次是段表。如果段表和页表都是在主存储器中,则要访问主存储器三次。对于段式虚拟存储器和页式虚拟存储器也要访问主存储器两次。因此,要想使虚拟存储器的速度接近主存储器的速度,或者说,要想使虚拟存储器能够真正实用,必须加快查表的速度。有关具体方法,将在后面作比较详细的介绍。

4. 2. 2　页式虚拟存储器的构成

页式虚拟存储器是以上三种虚拟存储器中使用最为普遍的一种,因此,对其构成按以下几个方面进行介绍。

1 .地址映像法则

页式虚拟存储器的地址映像法则是建立起多用户的虚拟程序空间到实际主存空间的映像关系,对应关系如图 4 .19 所示。

按照上述分页原理可知,由于虚拟空间与实存空间的页大小是相同的,多用户的虚地址中的‘ 页内偏移 D ”无需变换可直接形成实存地址中的页内偏移 d。关键在于将虚地址中的‘ 多用户的虚页号 ”(即用户号加虚页号部分) 变换成实地址中的实页号 p。

由于是把大的虚存空间映像到较小的主存空间,页式虚拟存储器采用的是全相

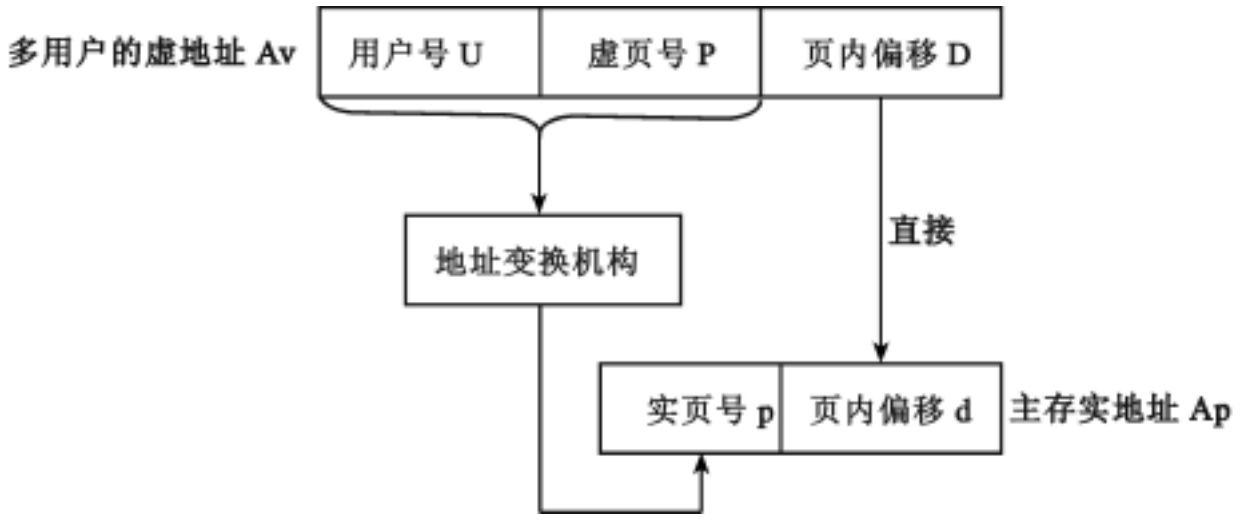


图 4.19　虚—实地址的对应关系

联映像法则。即让每道程序的任何虚页可以映像到任何实页位置。例如,某用户程序空间有 8 页,系统分配给该用户的实际主存空间只有 4 页时,全相联的映像关系如图 4.20 所示。

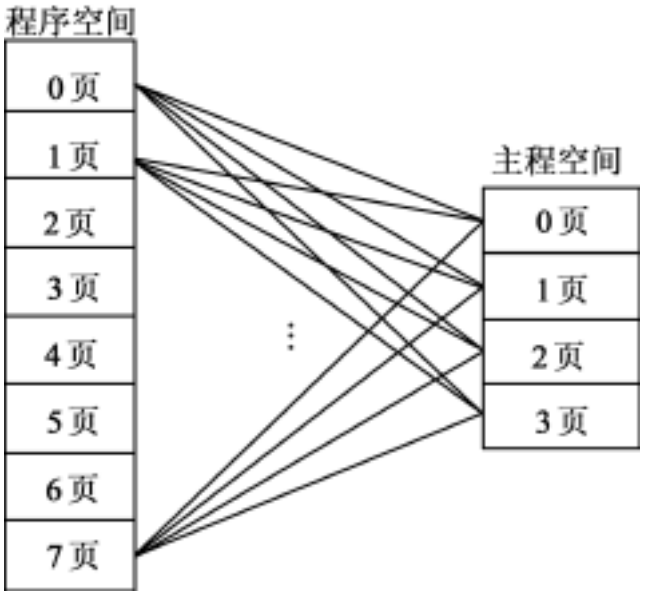


图 4.20　全相联映像

由图 4.20 可知,在任何时刻用户的程序能够装入主存的页面数最多只有 4 页。当 CPU 要访问的某一页没有装入主存时,就会产生页面失效。失效的页面 CPU 仍需要访问,系统通过通道将其调入主存后再进行访问,但由于主存已满,这就发生了两个或两个以上的页面想要进入主存中同一页位置的现象,这种现象称为页面争用或页面冲突。

例 4.1　某虚拟存储器的用户编程空间共有 32 个页面,每页 1KB,主存为 16KB。假定某时刻该用户页表已调入主存时页面的虚页号和实页号对照如表 4.3 所示。

表4.3 已装入主存的部分虚、实对照表

虚 页 号	实 页 号
0	5
1	10
2	4
8	7

问与虚地址 OA5CH,1A5CH 相对应的物理地址为多少？

解:根据前面介绍的虚实地址对应关系,可将 16 进制虚地址写成二进制形式,然后划分出虚地址中的各字段后,进而转换成主存物理地址,如图 4.21 所示。

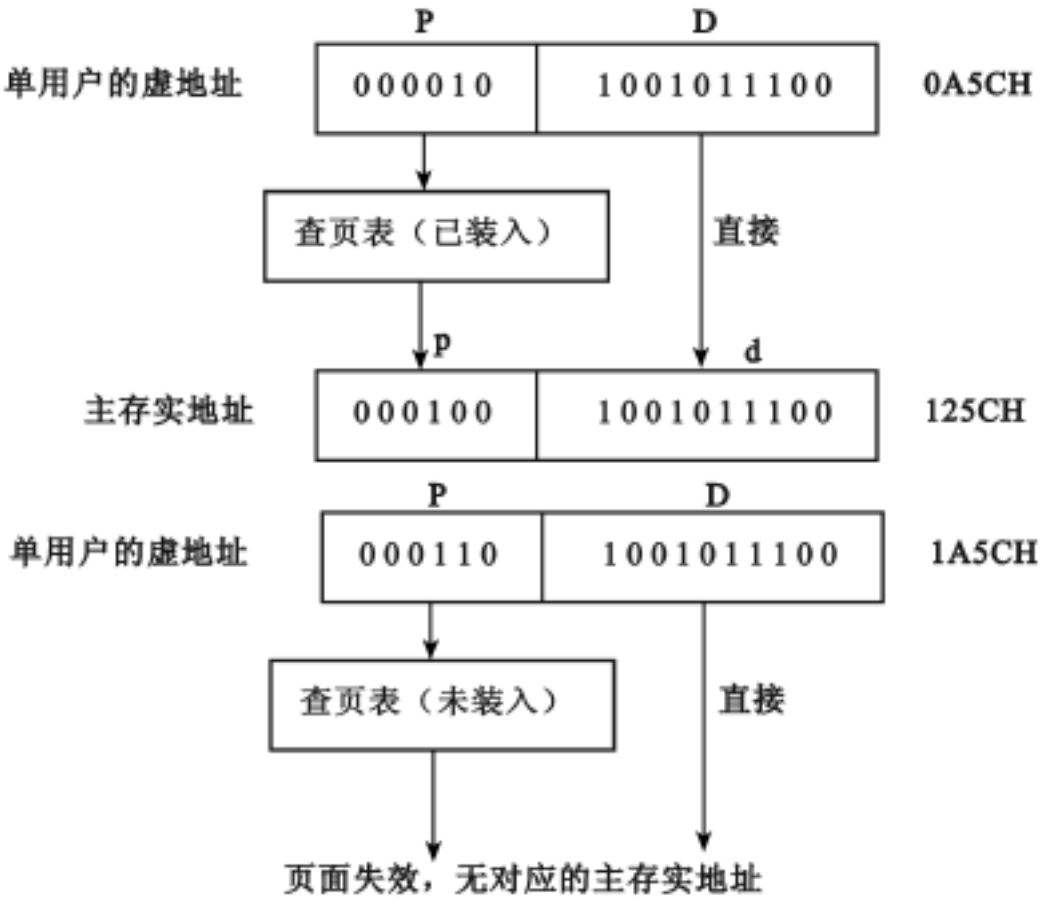


图 4.21 虚—实地址计算过程

2.外部地址变换

以上介绍了内部地址映像和变换方法,即把虚拟地址空间映像到主存物理地址空间,以及把虚拟地址变换成主存实地址的方法。当页表的有效位指示发生页面失效时,表示需要访问的那一页还没有装入到主存储器中,这时必须进行外部地址变

换。外部地址变换的目的是要找到辅存(磁盘存储器)的实地址,并且把需要访问的那一页调入到主存储器中。

磁盘存储器的地址格式如图 4.22 所示。由于一台机器可能有多个磁盘,因此,首先要给出磁盘号,然后是一个磁盘内的柱面号、磁头号和块号。磁盘存储器每一个物理块的大小是 512 字节。由于在页式虚拟存储器和段页式虚拟存储器中,每一页面的大小是固定的,通常是磁盘存储器物理块大小的整数倍,这个倍数在外部地址变换软件中是知道的。因此,在进行外部地址变换时只要给出磁盘存储器的起始地址,就能把一页或一个程序段都调入主存储器中。

磁 盘 号	柱 面 号	磁 头 号	块 号

图 4.22 磁盘存储器的地址格式

外部地址变换的过程如图 4.23 所示。由于页面失效的概率非常低,一般只有 1‰左右,因此,外部地址变换通常用软件来实现。每一个用户程序都有一张外页表(之所以称为外页表是因为它是在外部地址变换中使用的,与在内部地址变换中使用的页表被称为内页表相对应)。虚拟地址空间中的每一个页面或每一个程序段,在外页表中都有对应的一个存储字。每一个存储字中除了必须有磁盘存储器的地址外,至少还应包括一个装入位。

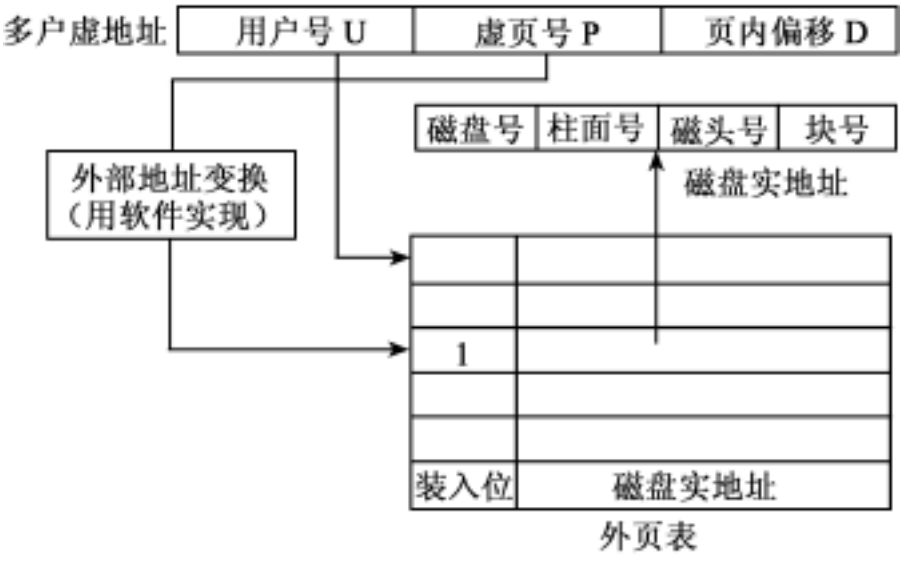


图 4.23 外部地址变换

由于每一个用户程序有一张外页表,因此,通过多用户虚地址中的用户号 U 就能够找到该用户程序的外页表起始地址。再通过虚页号 P 就能惟一确定外页表中与需要访问的页面相对应的那个存储字。如果该存储字中的装入位为“1”,则表示要访问的页面已经在磁盘存储器中,否则表示要访问的页面还不在磁盘存储器中,需要

从磁带、光盘存储器等海量存储器中调入。在段式虚拟存储器中,采用类似的方法,通过段号 S 惟一确定外页表中与需要访问的程序段相对应的那个存储字。

3 .页面替换算法

常用的页面替换算法有如下几种:

① 随机算法,即 RAND 算法(RANDom algorithm)。利用软件或硬件的随机数发生器来确定主存储器中被替换的页面。这种算法最简单,而且容易实现。但是,这种算法完全没有利用主存储器中页面使用情况的历史信息,也没有反映程序的局部性,所以命中率比较低。

② 先进先出算法,即 FIFO 算法(First-In First-Out algorithm)。这种算法选择最先调入主存储器的页面作为被替换的页面。它的优点是比较容易实现,能够利用主存储器中页面调度情况的历史信息,但是,没有反映程序的局部性。因为最先调入主存的页面,很可能也是经常要使用的页面,如公用子程序或公用数据等。

③ 近期最少使用算法,即 LRU 算法(Least Recently Used algorithm)。这种算法选择近期最少访问的页面作为被替换的页面。显然,这是一种非常合理的算法,因为到目前为止最少使用的页面,很可能也是将来最少访问的页面。该算法既充分利用了主存储器中页面调度情况的历史信息,又正确反映了程序的局部性。但是,这种算法实现起来非常困难。它要为每个页面设置一个很长的计数器,并且要选择一个固定的时钟为每个计数器定时计数。在选择被替换页面时,要从所有计数器中找出一个固定的时钟为每个计数器定时计数。在选择被替换页面时,要从所有计数器中找出一个计数值最大的计数器。因此,通常采用另外一种变通的办法,就是下面的 LFU 算法。

④ 最久没有使用算法,即 LFU 算法(Least Frequently Used algorithm)。这种算法把近期最久没有被访问过的页面作为被替换的页面。它把 LRU 算法中要记录数量上的“多”与“少”简化成判断“有”与“无”,因此,实现起来比较容易。

⑤ 最优替换算法,即 OPT 算法(OPTimal replacemant algorithm)。在时刻 t 找出主存储器中每个页将要用到的时刻 t_i,然后选择其中 $t_i - t$ 值最大的那一页作为被替换的页。

要实现 OPT 算法,惟一的办法是让程序先执行一遍,记录下实际的页地址流情况。根据这个页地址流才能找出当前要被替换的页面。显然,这样做是不现实的。因此,OPT 算法只是一种理想化的算法,然而,它也是一种很有用的算法。实际上,经常把这种算法用来作为评价其他页面替换算法好坏的标准。在其他条件相同的情况下,哪一种页面替换算法的命中率与 OPT 算法最接近,那么,它就是一种比较好的页面替换算法。

例 4 .2　一个程序共有 5 个页面,分别为 P1～P5。程序执行过程中的页地址流

(即程序执行中依次用到的页面)如下:

P1, P2, P1, P5, P4, P1, P3, P4, P2, P4

假设分配给这个程序的主存储器共有 3 个页面。图 4.24 为 FIFO, LFU 和 OPT 三种页面替换算法对这 3 页主存储器的使用情况,包括调入、替换和命中等。其中,用“*”号标记下次将要被替换掉的页面。

从图 4.24 中可以看出,FIFO 算法的命中率最低,LFU 算法的命中率与 OPT 算法很接近。这一结论具有普遍意义。因此,在实际使用中,LFU 算法是一种比较好的算法。目前,许多机器的虚拟存储器都采用 LFU 算法。

时间 t	1	2	3	4	5	6	7	8	9	10	实际命中次数
页地址流	P1	P2	P1	P5	P4	P1	P3	P4	P2	P4	
先进先出算法(FIFO 算法)	1	1	1	1 *	4	4	4 *	4 *	2	2	
		2	2	2	2 *	1	1	1	1 *	4	
				5	5	5 *	3	3	3	3 *	
	调入	调入	命中	调入	替换	替换	替换	命中	替换	替换	2 次
最久没有使用算法(LFU 算法)	1	1	1	1	1	1	1	1 *	2	3	
		2	2	2 *	4	4	4 *	4	4	4	
				5	5 *	5 *	3	3	3 *	3 *	
	调入	调入	命中	调入	替换	命中	替换	命中	替换	命中	4 次
最优替换算法(OPT 算法)	1	1	1	1	1	1 *	3 *	3 *	3	3	
		2	2	2	2	2	2	2	2	2	
				5 *	5 *	4	4	4	4	4	
	调入	调入	命中	调入	替换	命中	替换	命中	命中	命中	5 次

图 4.24　三种页面替换算法对同一个页地址流的调度过程

在上面介绍的 5 种页面替换算法中,随机算法的命中率比较低,一般仅用于必须用硬件实现,而且对命中率要求不太高的场合。LRU 算法由于其实现起来特别困难,目前很少被采用。因此,在虚拟存储器中,实际上有可能被采用的页面替换算法就只有 FIFO 和 LFU 两种。

下面说明一下堆栈型替换算法。

在主存储器页面的分配和调度过程中,影响命中率的因素很多,那么最主要的因素是什么呢?堆栈型替换算法就是研究主存储器的页面数与命中率的关系。弄清楚了这个问题,就不需要做大量的模拟工作,就可能知道如何为程序分配主存页面数。

那么,什么是堆栈型替换算法呢?它的定义如下:

以任意一个程序的页地址流作两次主存页面数分配,分别分配 m 个主存页面和 n 个主存页面,并且有 $m \leq n$。如果在任何时刻 t,主存页面数集合 Bt 都满足关系:

$$Bt(m) \quad Bt(n)$$

则这类算法称为堆栈型替换算法。

简单地说,堆栈型替换算法的基本思想是:随着分配给程序的主存页面数的增加,主存的命中率也提高,至少不下降。

很容易证明,LRU 算法和 LFU 算法是堆栈型替换算法,OPT 算法也是堆栈型替换算法。但是,FIFO 算法不是堆栈型替换算法。FIFO 算法在主存页面数增加时命中率反而下降,如图 4.25 所示。

时间 t	1	2	3	4	5	6	7	8	9	10	11	12	实际命中次数
页地址流	P1	P2	P3	P4	P1	P2	P5	P1	P2	P3	P4	P5	
主存页面数 $n=3$	1	1	1 *	4	4	4 *	5	5	5	5	5 *	5 *	
		2	2	2 *	1	1	1 *	1 *	1 *	3	3	3	
			3	3	3 *	2	2	2	2	2 *	4	4	
	调入	调入	调入	替换	替换	替换	替换	命中	命中	替换	替换	命中	3 次
主存页面数 $n=4$	1	1	1	1	1 *	1 *	5	5	5	5 *	4	4	
		2	2	2	2	2 *	2 *	1	1	1	1 *	5	
			3	3	3	3	3	3 *	2	2	2	2 *	
				4	4	4	4	4	4 *	3	3	3	
	调入	调入	调入	调入	命中	命中	替换	替换	替换	替换	替换	替换	2 次

图 4.25 FIFO 算法在主存页面数增加时命中率反而下降

由于堆栈型替换算法的命中率随分配给该程序的主存页面增加而单调上升,因此,在多道程序系统中,可以采用一种被称为页面失效频率法 PFF(Page Fault Frequency)的动态页面调度方法。具体做法是:根据各道程序在实际运行过程中页面失效率的情况,由操作系统动态高速分配给每道程序的主存页面数。当一道程序的命中率低于某个限定值时就增加分配给该道程序的主存页面数,以提高它的命中率。而当命中率高于某个限定值时就减少分配给该道程序的主存页面数,把节省出来的主存页面分配给其他程序,从而使整个系统的总的命中率和主存利用率都得到提高。

例 4.3 采用页式管理的虚拟存储器,分时运行两道程序。其中,程序 X 为

```
DO   50 I = 1,3
```

```
      B(I) = A(I) - C(I)
      IF (B(I)≤0) GOTO  40
      D(I) = 2* C(I) - A(I)
      IF  D(I) = 0  GOTO  50
40    E(I) =  0
50    CONTINUE
      DATA: A  = ( - 4, + 2, 0)
                 C  = ( - 3, 0, + 1)
```

每个数组分别放在不同的页面中;而程序 Y 在运行过程中,其数组将依次用到程序空间的第 3,5,4,2,5,3,1,3,2,5,1,3,1,5,2 页。如是采用 LRU 算法替换,实存却只有 8 页位置可供存放数组之用。试问为这两道程序的数组分配多少实页最为合理?为什么?

解:由题意可知,假若 X 程序中数组 A,B,C,D,E 分别存放在第 1,2,3,4,5 页。运行 X 程序时,出现的页地址流依次为 1,3,2,2,5,1,3,2,2,3,1,4,4,5,1,3,2,2,5 共 19 个页面地址。

使用堆栈处理法,X,Y 程序页地址流堆栈处理过程分别如图 4.26 和图 4.27 所示。其中 N 为主存页面数。

时间 t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
页地址流	1	3	2	2	5	1	3	2	2	3	1	4	4	5	1	3	2	2	5
	1	3	2	2	5	1	3	2	2	3	1	4	4	5	1	3	2	2	5
		1	3	3	2	5	1	3	3	2	3	1	1	4	5	1	3	3	2
			1	1	3	2	5	1	1	1	2	3	3	1	4	5	1	1	3
					1	3	2	5	5	5	5	2	2	3	3	4	5	5	1
												5	5	2	2	2	4	4	4
N=1				命中					命中				命中					命中	
N=2				命中					命中	命中			命中					命中	
N=3				命中					命中	命中	命中		命中		命中			命中	
N=4				命中		命中	命中	命中	命中	命中	命中		命中		命中	命中		命中	命中
N=5				命中		命中	命中	命中	命中	命中	命中		命中	命中	命中	命中	命中	命中	命中

图 4.26　X 程序页地址流堆栈处理

综合 X,Y 两道程序分析并列于表 4.4 中。由此可知,当系统分别给每个用户分配 N=4 个实页时,系统命中率达到最大值 64.95%。所以给两个程序各分配 4 个实页较为合适。

时间 t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
页地址流	3	5	4	2	5	3	1	3	2	5	1	3	1	5	2
	3	5	4	2	5	3	1	3	2	5	1	3	1	5	2
		3	5	4	2	5	3	1	3	2	5	1	3	1	5
			3	5	4	2	5	5	1	3	2	5	5	3	1
				3	3	4	2	2	5	1	3	2	2	2	3
							4	4	4	4	4	4	4	4	4
N=1															
N=2								命中					命中		
N=3					命中			命中					命中	命中	
N=4					命中	命中		命中	命中	命中	命中	命中	命中	命中	命中
N=5					命中	命中		命中	命中	命中	命中	命中	命中	命中	命中

图 4 27 Y 程序页地址流堆栈处理

表 4 .4 两道程序堆栈处理后的结果分析

X 用户程序		Y 用户程序		综合系统命中率
实页数	命中率	实页数	命中率	
3	7/ 19≈38 .8%	5	10/ 15≈66 .7%	H = 51 .75%
4	12/ 19≈63 .2%	4	10/ 15≈66 .7%	H = 64 .95%
5	14/ 19≈73 .7%	3	4/ 15≈26 .7%	H = 50 .2%

4. 页式虚拟存储器的工作全过程

页式虚拟存储器工作全过程,如图 4 .28 所示。

每当以虚地址访问主存时,都必须查内页表①,若装入位为 1 则形成主存实地址②,可访问主存③。若装入位为 0,则产生页面失效故障(异常)④,并直接经外部地址变换⑤形成辅存实地址⑥,经 I/ O 处理机调页至主存⑦。如果外页表查找失效,则需访问海量存储器⑧。发生页面失效后,由操作系统查找主存页面表,以确定调进的页存放在主存中什么位置⑨。若主存未满,根据全相联映像法则,调进的页可存放在任何空页位置⑩。否则,需通过替换算法寻找被替换掉的页面 1 , 2 。不论是哪一种情况,最后将主存页面号送 I/ O 处理机 3 ,由 I/ O 处理机完成调页⑦。替换时应考虑已经被修改过的页重新送回辅存 4 。

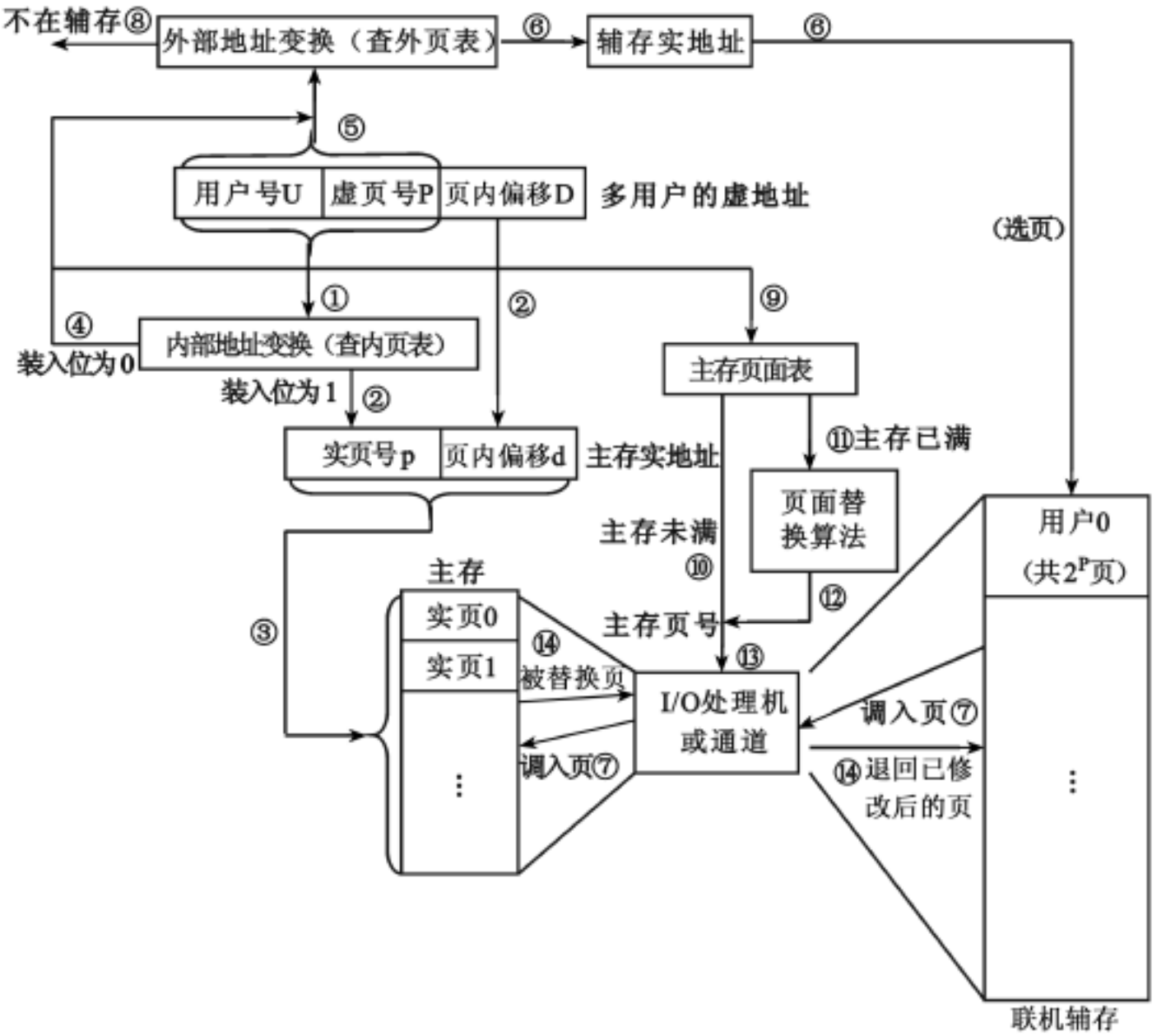


图 4 .28　页式虚拟存储器工作全过程

4. 2. 3　加快页式虚拟存储器地址变换的方法

从页式虚拟存储器的工作全过程可知,多用户的虚地址变换成主存实地址,可能性只有三种:一是被访问的页已经装入主存(命中),多用户的虚地址经查内页表(内部地址变换)转成实际主存地址;二是被访问的页不在主存(不命中),但主存仍有可调进新页的空间,多用户的虚地址经查外页表(外部地址变换),由通道或 I/O 处理机完成调页过程;三是被访问的页不在主存,且主存已经装满,则多用户的虚地址经查外页表完成调页,同时,还需根据某种替换算法,完成新旧页面的替换。

在以上三种情况中,第一种出现的概率远高于后两者之和。根据 Amdahl 定律,为了提高整个虚拟存储器地址变换速度,关键在于提高查内页表(内部地址变换)的速度。具体方法如下:

1. 目录表法

多用户的虚存空间要比实际主存空间大得多,即虚存页面数(2^{U+P} 页)。远大于实际主存页面数(2^{P} 页),内页表中装入位为“0”的项有($2^{U+P}-2^{P}$)个存储字。为此,可将内页表压缩为已经装入主存中的虚页与实页的对应关系,去掉装入位为“0”的存储字,并用相联存储器构成压缩后的内页表,页表中的每一个存储字包括虚页号、实页号、修改位和访问方式等,不再设置装入位,我们把这种相联目录表简称为目录表。

采用目录表法的虚拟存储器,其地址变换过程如图 4.29 所示。为了把多用户虚页号变换成主存实页号,要把多用户虚地址中的多用户虚页号(U 与 P 拼接起来)与相联存储器中的多用户虚页号字段逐个进行比较。如果有相等的,表示要访问的这个页面已装入到主存储器中了。这时,读出该单元中的其他字段,其中,实页号字段中存放的就是与多用户虚页号相对应的主存实页号。只要把这个实页号 p 与多用户虚地址中的页内偏移 D 直接拼接起来就成了要访问的主存实地址。如果没有相等的,则表示要访问的那个页面还没有装入到主存储器中,这时,发出页面失效请求,从磁盘存储器中把要访问的那一个页面调入主存储器。

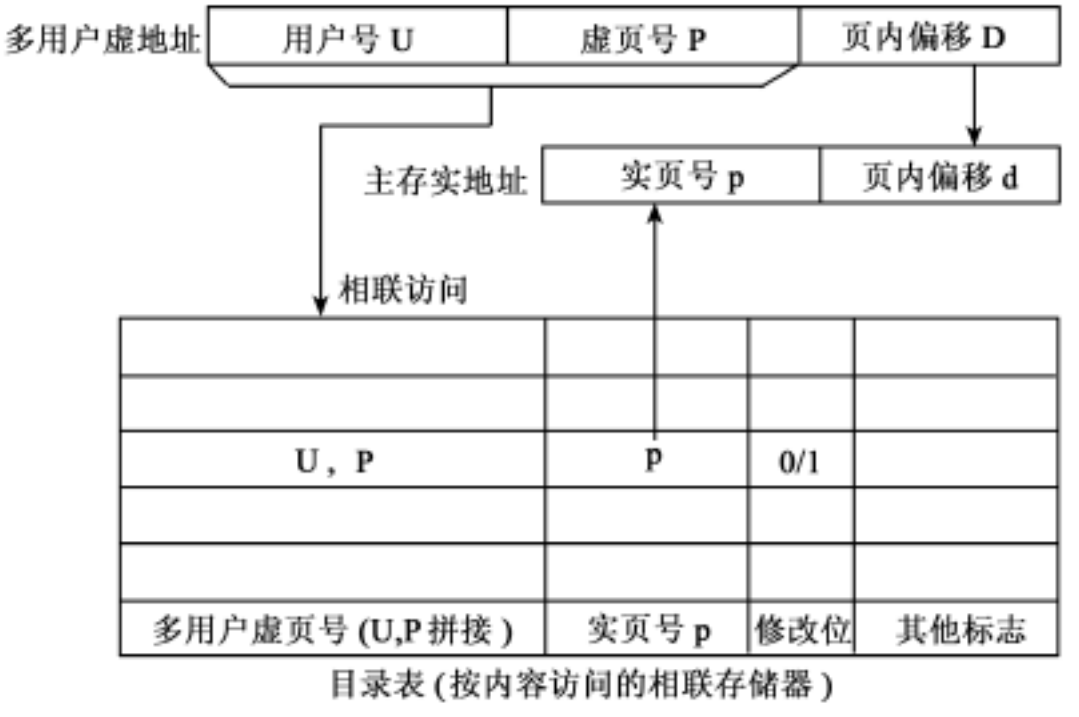


图 4.29 目录表法的地址变换过程

由于目录表是采用高速度小容量存储器实现的,与把页表放在主存储器中的方法相比,查表的速度要快得多。

然而,随着主存储器容量的增加,目录表的容量也随之增加。当主存储器的容量增加到一定数量后,目录表的造价就会很高,查表的速度也会降低。

2. 快慢表法

由于程序在执行过程中具有局部性,因此,对页表中各存储字的访问并不是完全随机的。也就是说,在一段时间内,对页表的访问只是局限在少数几个存储字内。根据这一特点,可将页表由快表和慢表共同构成。慢表是全表,仍然存放在主存储器中,快表是慢表的一个副本,一般是由十几个存储字的相联存储器构成,访问速度与CPU内部寄存器相当。

查表时快表和慢表同时进行,若快表命中,则其访问时间很短,慢表访问无效。若快表失效,则慢表访问时间即为等效查表时间。实际上快表与慢表也构成了一个由两级存储器构成的存储系统。当快表装满时,则要采用某种替换算法,替换掉某一个存储字。由快慢表法构成的虚拟存储器地址变换过程如图4.30所示。

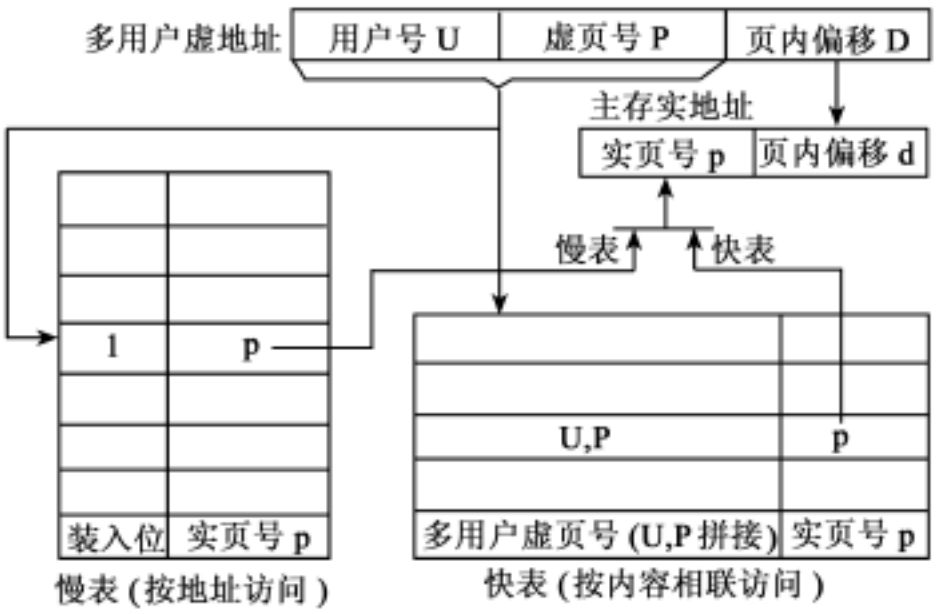


图 4.30　采用快慢表的地址变换过程

由于快表的查表速度非常快,与主存储器的一个存储周期相比几乎可以忽略不计。因此,只要快表的命中率很高,那么,虚拟存储器的访问速度就能与主存储器的工作速度很接近。

3. 散列变换法

在采用快慢表结构的虚拟存储器中,要提高快表的命中率,最直接的办法是增加快表的容量。快表的容量越大,命中率就越高。但是,由于快表是按相联方式访问的,当快表的容量增加时,它的查表速度就会降低。那么,能不能让快表不采用相联方式访问,而采用普通的按地址来访问呢?

如果要在一个按地址访问的存储器中查找一个信息,可以使用顺序查找法、对分查找法和散列查找法等。其中,散列(Hashing)查找方法的速度最快。对于快表来

说,就是要把多用户虚页号 P_V 变换成快表的地址 Ah,其函数关系是:

$$Ah = H(P_V)$$

散列函数的种类很多。由于快表中的散列函数必须用硬件来实现,因此,通常采用一些简单的函数关系。图 4.31 是一种折叠按位加散列函数,把 15 位多用户虚地址 P_V(内容)散列变换成 6 位的快表地址 Ah。

由于是把一个大的多用户虚页号 P_V 散列变换成了一个小的快表地址 Ah,因此,必然会有很多个多用户虚页号 P_V 都散列变换到相同的快表地址 Ah 中,这种现象称为散列冲突。例如,在图 4.31 中,平均会有 $2^{15} \div 2^6 = 2^9$,即 512 个多用户虚页号全部被散列变换成同一个快表地址。

为了避免因散列冲突而发生查快表时的错误,必须把多用户虚页号也加入到快表中去,并且与主存实页号存放在同一个快表存储字中。另外,还要用一个比较器,把从快表中读出来的多用户虚页号与多用户虚地址中的多用户虚页号进行比较。

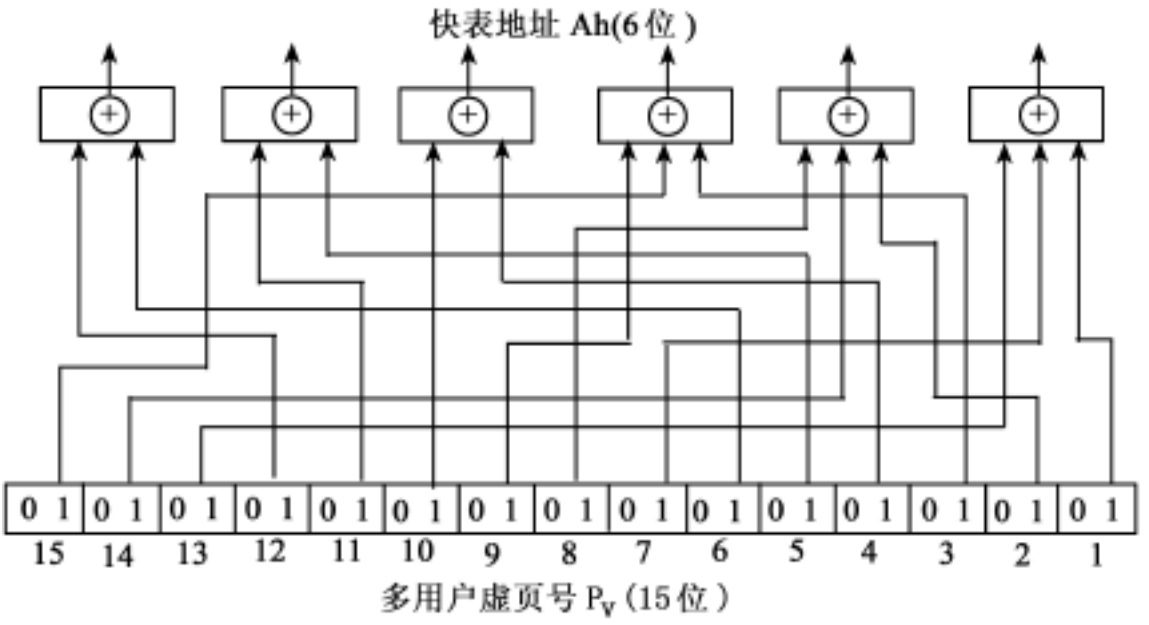


图 4.31 一种用硬件实现的散列函数

采用散列变换实现快表按地址访问的虚拟存储器如图 4.32 所示。地址变换的过程是这样的:首先把多用户虚地址中的多用户虚页号 P_V(由 U 和 P 拼接而形成)送入硬件的散列变换部件,经散列变换后得到快表的地址 Ah。然后用地址 Ah 访问快表,读出主存实页号 p 和多用户虚页号 P_V'。把主存实页号 p 送入主存储器的地址寄存器,与页内偏移 d 直接拼接起来形成主存地址,并且用这个地址立即去访问主存储器。同时,把读出的多用户虚页号 P_V' 与多用户虚地址中多用户虚页号 P_V 在相等比较器中进行比较。如果比较结果相等,就继续刚才的访问主存储器的操作,否则,立即终止访问主存储器的操作。比较结果不相等表示发生了散列冲突,这时,需要去查存放主存储器中的慢表。

由于快表按地址访问,在保证访问速度的前提下,其存储容量可以比用相联存储

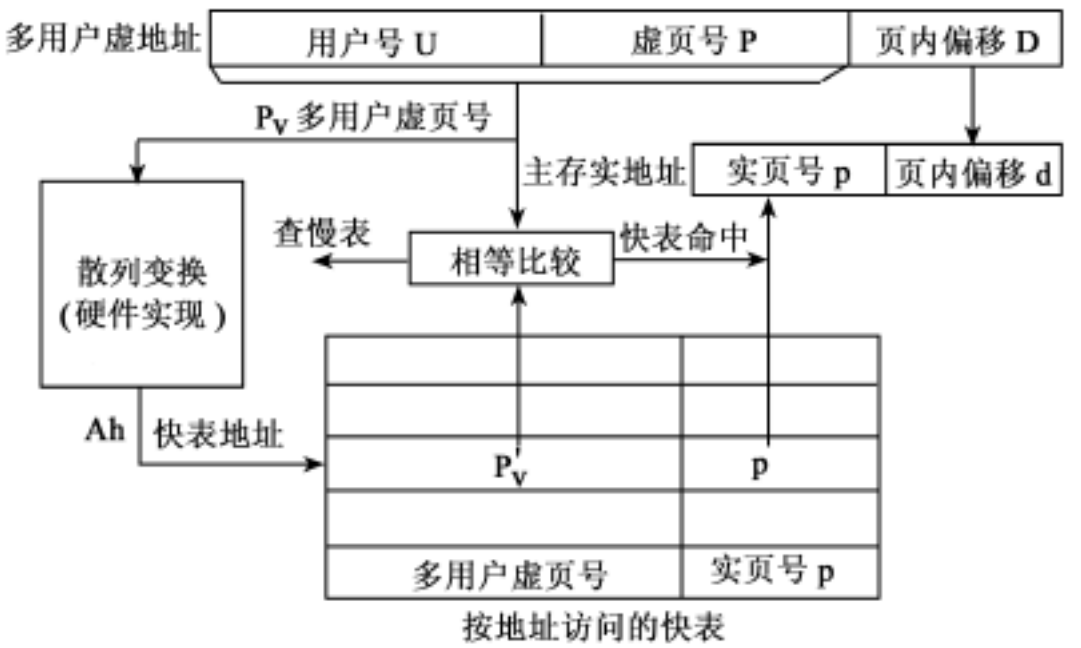


图 4 .32　采用散列变换实现快表按地址访问

器实现的快表大很多倍。虽然也有一次相等比较,但相等比较可以与访问存储器的操作同时进行。因此,这种快表按地址访问的快慢表结构,与上面介绍的采用相联存储器作快表的快慢表结构相比,快表的命中率要高很多,而且查表的速度也很快。

4. 虚拟存储器举例

IBM 370/ 168 计算机的虚拟存储器快表结构及地址变换过程,如图 4 .33 所示,虚拟地址共长 36 位。其中,页面大小为 4K 字节,占 12 位。每个用户(或每道程序)最多允许占用 4K 个页面,因此,虚页号也占 12 位。最多允许 16G 个用户(或 16G 道程序)同时上机,用户号占 24 位,但是,实际上在一段时间内同时上机的用户数一般不超过 6 个。

快表按地址访问,快表的地址由多用户虚页号经硬件的散列变换部件压缩后生成。快表地址共 6 位,因此,快表容量为 64 个存储字。

IMB 370/ 168 计算机的虚拟存储器采用了两项新的措施:第一项措施是在快表的每一个存储字中存放两对多用户虚页号与主存实页号,并采用两个相等比较器。只要其中有一个比较器的比较结果相等,就认为快表命中。这时,立即把命中的实页号 p 送到主存地址寄存器中,与地址寄存器中的页内偏移拼接成主存实地址。只有当两个比较器的比较结果都不相等时,才认为快表没有命中,需要到慢表中去查主存实页号 p。

第二项措施是用一个由 6 个寄存器组成的相联寄存器组把 24 位的用户号 U 压缩成 3 位的 ID,把这个 ID 与虚页号直接拼接起来作为散列变换部件的输入。这样做能够减少散列部件的输入与输出位数的差,从而降低散列冲突。

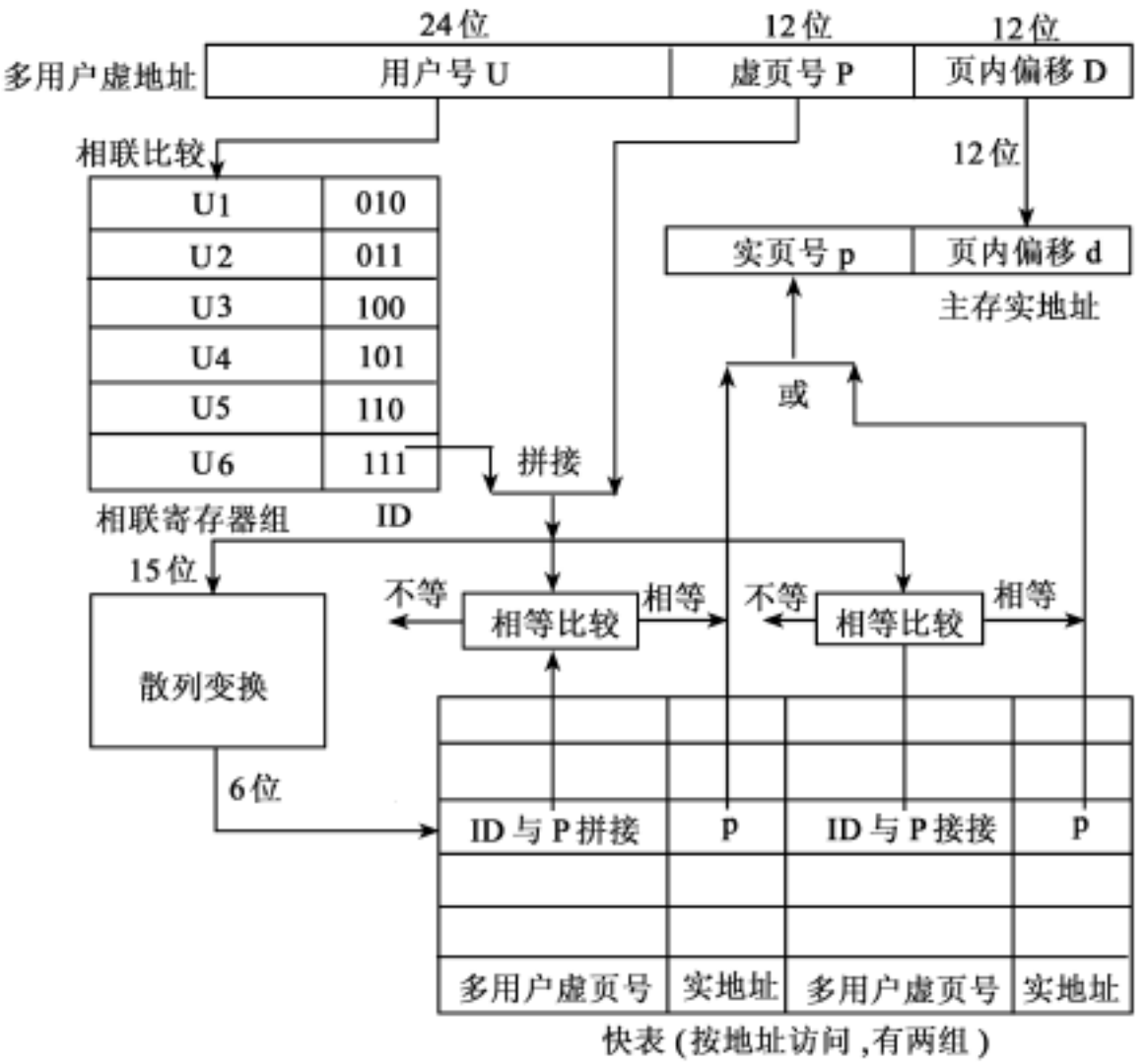


图 4.33 IBM 370/168 计算机的虚拟存储器快表结构

在地址变换开始时,把多用户虚地址中用户号 U 与相联寄存器组中的 6 个用户号 U1～U6 逐个比较。如果有相等,就读出对应的 3 位 ID,并把这 3 位 ID 与多用户虚地址中的 12 位虚页号 P 直接拼接成 15 位,用这 15 位作为散列变换部件的输入。这样,在快表中可以同时保留 6 个用户(或 6 道程序)的常用页表。在 6 个用户或程序之内切换时,仍能保持很高的快表命中率。如果用户号 U 与 6 个相联寄存器的比较结果都不相等。则表示有 6 个之外的新用户进入。这时,需要用替换算法把一个不常进入的用户替换出相联寄存器组。只要替换算法好,就能保证经常进入的用户或系统服务任务被一直保留在相联寄存器组中。

目前,许多计算机系统的虚拟存储器都采用与 IBM 370/168 计算机类似的方法,包括相联寄存器组、硬件的散列压缩、快慢表结构和多个相等比较器等,而且,所有这些措施对应用程序员来说都是透明的。

4.2.4 提高主存命中率的方法

影响主存命中率的主要因素有如下 5 个:

- 程序在执行过程中的页面地址流分布情况;
- 所采用的页面替换算法;
- 页面大小;
- 主存储器的容量;
- 所采用的页面调度方法。

在这些因素中,页面地址流的分布情况是由程序本身决定的,系统设计人员一般无能为力。页面替换算法在上一节中已经介绍过,目前,多数计算机都采用 LFU 算法,它是一种堆栈型替换算法。在当前看来,已经是一种比较好的算法了。下面,对影响主存命中率的另外三个因素作简单的分析。

1. 页面大小的选择

页面大小的选择对于主存命中率的影响分析比较复杂,它取决于程序的访存地址流以及主存空间的大小。当页面的大小 S_p 改变时,命中率 H 受到两方面因素的制约。当 S_p 比较小时,H 随 S_p 的增大而增大。然而,当 S_p 超过某一值时,H 将随 S_p 的增大而下降,如图 4.34 所示。

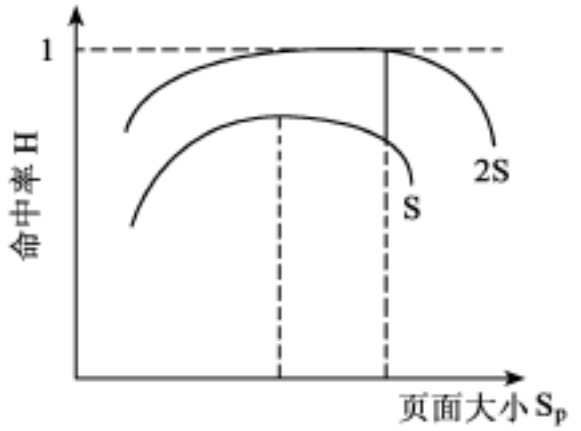


图 4.34　页面大小与主存命中率的关系

在设计一个虚拟存储器时,页面大小的选择一般要通过对典型程序的模拟实验来确定。早期的许多计算机系统,页面大小一般为 1KB。目前,大多数计算机的页面大小都取 4KB,8KB 或 16KB。

2. 主存容量

主存命中率 H 随着分配给该程序的主存容量 S 的增加而单调上升,如图 4.35 所示。在 S 比较小的时候,H 提高得非常快。随着 S 的逐渐增加,H 提高的速度逐渐降低。当 S 增加到某一个值后,H 几乎不再提高。

在页面替换算法中有这样一个结论,对于堆栈型替换算法,命中率随着分配给程序的页面数的增加而提高。当分配给程序的主存容量增加时,如果页面大小是一定

的,那么,页面数就会增加,因此,命中率也将提高。如果不是堆栈型算法,命中率虽然不会单调上升,在局部可能会有下降,但总的趋势还是上升的。

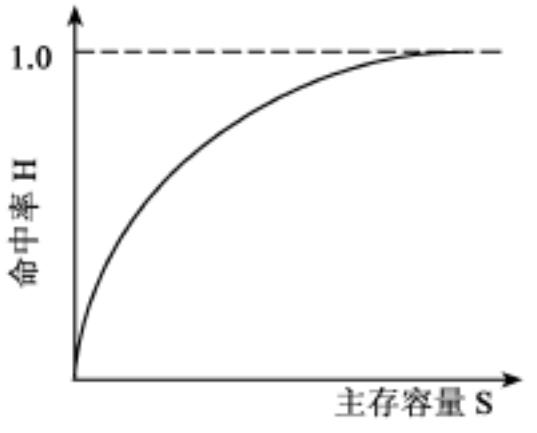


图 4 .35　主存命中率 H 与主存容量 S 的关系

从图 4 .35 中可以得到这样一个启发,在为一道程序分配主存空间时,对主存命中率的要求不能过分。当主存容量增加到某一个值之后,命中率提高得非常慢。这时,主存储器中不活跃部分所占的比例很大,主存资源的利用率就会很低。

实际上,操作系统在为程序分配主存空间时,是以页为单位分配的。因此,图 4 .35所示的不应该是一条平滑的曲线,而是台阶型的。在分配给程序的主存容量比较小的时候,台阶非常陡,随着主存容量的增加,台阶越来越平缓。

3. 页面调度方式

在操作系统中,页面调度通常有两种方式。一种是分页式,这种方式在程序装入主存储器之前对程序进行链接装配,并且要在把整个程序都调入到主存储器中之后才能开始运行。另一种方式是请求页式,这种方式只在发生页面失效时,才把要访问的页面进行链接装配,并调入到主存储器中。

前一种方式的主存命中率可以达到 100%,但是,主存利用率比较低,这是因为主存储器中不活跃部分所占的比例比较大。而且,当主存剩余空间小于程序所需要的主存空间时,这个程序将无法装入到主存储器中运行。

目前,大多数计算机采用请求页式调度方式。这种方式虽然具有主存利用率高等优点,但是,在程序执行过程中经常要发生页面失效,而且处理页面失效需要比较长的时间。特别是在程序刚开始运行时,页面失效很频繁。只有当调入的页面数比较多时,命中率才开始上升。因而就产生了一种折衷页面调度方案,即所谓的预取式调度方式。

预取式调度方式根据程序的局部性特点,在程序被挂起之后又重新开始运行之前,先把上次停止运行前一段时间内用到的页面先调入到主存储器,然后才开始运行程序。这样,在程序一开始运行时,主存储器中就已经装入了一定数量的页面,从而

可以避免在程序刚开始运行时,频繁发生页面失效的情况。

4.2.5 虚拟存储器的保护技术

以上各小节的介绍中尚未考虑多用户共享系统资源的情况,在多用户的情况下,存储系统应防止因一个用户的错误操作或者程序出错而破坏系统中的其他用户程序和数据,还要防止用一个用户不合法的访问主存中其他存储区域而对系统造成的破坏。对于多道程序系统和多用户系统这是必不可少的。

多道程序设计引出了进程的概念,进程的分时处理引出了进程切换或上、下文切换的概念,进程的切换必须保证进程的正确运行不受影响,保证进程的正确和安全运行是系统设计员和操作系统程序员共同的责任。保护的手段主要是对存储区域的保护,使一个进程的信息不被另一个进程修改。除了对存储区域的保护外,多道程序还要求进程间共享数据,使进程间进行数据通信。

存储系统的保护分存储区域的保护和访问方式的保护:

为实现区域保护,对于不是虚拟存储器的主存系统可采用界限寄存器的方式,由系统软件设置用户进程访存地址的上、下界,禁止越界访问,用户程序不能改变上、下界的值,因此也就不能破坏其他用户及系统程序的存储空间,但对于虚拟存储器系统,用户程序的访问空间映像到主存中将不是一个连续的地址空间,而将分布在主存中各个页面,因而不适合采用这种保护方式。

对虚拟存储器的保护方式有映像表保护法、键保护法和环式保护法等。

映像表保护法是段式和段页式管理利用映像表的映像关系,限制用户程序的访问地址空间,用户程序不能访问映像表上找不到的主存页面,从而起到保护作用,如在段表中的每行都设一个段长项。若访存地址超出该段的地址范围,则产生越界中断,或者将程序运行时的进程标识符作为虚地址的最高位,使不同的进程对应不同的段,以防止越段访问。但进程之间一般需要共享存储区域以实现数据通信,上述方法不支持存储区的共享。解决的方法是将进程地址空间分成系统区域和用户区域,系统区域是各进程共享的存储空间,用户区域是进程占有的存储空间,两个区域的地址分别对应不同的页表。因为各进程的系统区域是共享的,它们对应同一个主存空间,因此可以只因一个页表为系统区变换地址。在段式管理中,通常在段表中设置访问许可信息。一个段可设置为只读、只可执行或者只可由系统访问。多个进程共享的段可对每个进程都设置不同的访问许可。

键保护法是由操作系统按当时主存的分配情况,给主存的每一页分配一个键,称为存储键,它相当于一把锁,每个用户进程在主存中的各页面都有一个相同的存储键。进程访问这些页面,需要一个访问键,相当于一把钥匙。用户进程的访问键都由操作系统给定,存放在处理机的程序状态字或控制寄存器中。IBM 370 系统就采用这种方法,它的保护键有 4 位,分别赋给已调入主存的 16 个活跃进程。

环式保护法是将系统程序和用户程序按访问权限分层,如图 4.36 所示。最内的

几层是系统程序层,其外面几层是用户程序层,级别由里向外逐层降低。程序各页的环号由操作系统给定,并置入页表,每个程序都规定了一个上限环号,即可访问的最内层的环号。环式保护法既能保护用户程序的出错而不破坏系统程序的工作,也能使同一用户程序的外层部分的出错不致破坏其内层部分。

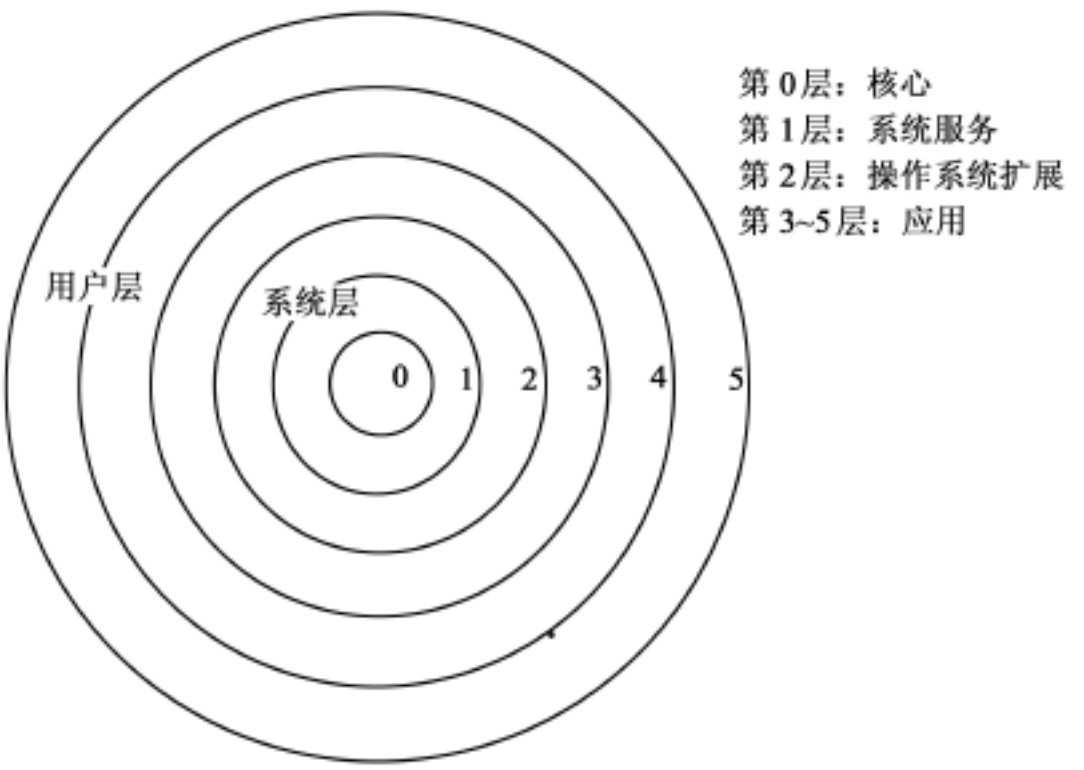


图 4.36 环式保护的分层

上述保护方法都由硬件支持,因而有较高的可靠性和速度,从系统结构上应对存储保护提供如下手段:

① 提供两种以上的工作状态,分别表示当前运行的进程是用户进程还是操作系统进程。

② 提供一些 CPU 状态,使用户进程可读但不可写。这些状态包括上、下界寄存器、用户状态/ 系统状态位、中断许可位等。

③ 提供一种在用户状态和系统状态间转换的机制,通常用户程序可通过系统调用实现状态的转换,系统调用时保存现场并使工作状态变为操作系统状态,系统状态返回用户状态通常由子程序返回的方式恢复原先状态。

4.2.6 Pentium 微处理器的虚拟存储器

作为虚拟存储器管理的典型例子,Pentium 微处理器虚拟存储器管理方式,一方面继承了过去只有在大、中型计算机系统上才能实现的管理技术,另一方面又优于这些大、中型系统最初采用的策略。因为在研制 Pentium 微处理器时,就以支持多种操作系统为其目标之一。

1. Pentium 微处理器的工作模式

Pentium 微处理器支持三种工作模式:实地址模式、受保护的虚拟地址模式和虚拟 8086(V86)模式。

(1)实模式

实模式,即为实地址模式,这是自 8086 一直延续继承下来的 16 位模式。逻辑地址形式为段、偏移,二者均是 16 位。将段名所指定的段寄存器的内容乘以 16(即左移 4 位),得到 20 位段基址,加上 16 位偏移即得 20 位物理地址。

实模式使用 A_{19} ～ A_0 的 20 根地址线,最大物理地址空间为 1MB。最高物理地址为 FFFFFH,若段基址加上偏移计算出的物理地址超过 20 位,则超出位被放弃,即出现地址环绕现象。例如 FFFF:FFFF 计算机的地址为 10FFEF,实际送出的物理地址为 0FFEF。

MS-DOS 操作系统、运行于实模式时的 Windows 3 .x 和它们的 16 位应用程序采用实模式。

(2)保护模式

保护模式即为受保护的虚拟地址模式,这是 80386 才具备并一直延续下来的 32 位模式。在这种模式下 Pentium 的存储管理部件 MMU 设有分段部件 SU 和分页部件 PU,允许 SU,PU 单独工作或同时工作。于是保护模式又细分为如下三种模式:

① 分段不分页模式(段式管理)

此时虚拟地址(或逻辑地址),由一个 16 位的选择符(Selector)和一个 32 位的偏移组成。选择符的最低 2 位与保护机构打交道,高 14 位用于指定具体的段。一个进程可拥有的最大虚拟地址空间是 $2^{14+32} = 2^{46} = 64\text{TB}$。

由 SU 将二维的分段虚拟地址转换成一维的 32 位线性地址。对于分段不分页模式,这也就是它的物理地址。分段不分页的好处是,无需访问页目录和页表,地址转换速度快。另外,对段提供的一些保护定义可以一直贯通到段的单个字节级。

② 分段分页模式(段页式管理)

这是一种在分段基础上添加分页存储管理的模式,即将 SU 转换后的 32 位线性地址看成是页目录、页表和页内偏移三个字段所组成的,由 PU 完成两级页表的查找将其转换成 32 位物理地址。此模式下一个进程可拥有的最大虚拟地址空间,同于分段不分页模式,也是 64TB。

这是一种兼顾分段分页两种优点的虚拟地址模式,受到 UNIX System V 和 OS/2 操作系统的偏爱。

③ 不分段分页模式(页式管理)

这种模式下 SU 不工作,只是 PU 工作。逻辑地址中无 16 位选择符,以寄存器提供的 32 位地址被看成是由页目录、页表和页内偏移三个字段组成。由 PU 完成虚拟地址到物理地址的转换。进程所拥有的最大虚拟地址空间是 $2^{32} = 4\text{GB}$,与上述两

种模式相比,虽然虚拟空间减小了,但也够用。

这种纯分页的虚拟地址模式,也称为平展地址模式。它也能提供保护机制,而且将虚拟存储器看成是线性分页地址空间,比分段模式具有更大的灵活性。Windows NT, Windows 95 操作系统采用了这种模式来支持 32 位应用程序的运行。

(3)虚拟 86 模式

这是一种在 32 位保护模式下支持 16 位实模式应用程序运行的特别保护模式,简称 V86 模式。自 80386 开始并一直延续到 Pentium,在这种模式下系统可建立多个 8086 虚拟机,每个虚拟机都认为自己是惟一运行的机器,安全地运行以实模式编写的 16 位应用程序。这样在 32 位保护模式的操作系统管理下,系统可同时运行 32 位应用程序和 16 位应用程序。当然,这种"同时"是由 CPU 切换完成的。CPU 中 EFLAG 寄存器的 VM(b_{17}位)即为 V86 模式位,已经工作在保护模式下的 CPU,若 VM = 1,则 CPU 运行 V86 模式,否则运行一般保护模式。这种相互切换由任务转换或中断来完成。

V86 模式是将段寄存器的内容乘以 16 作为段的基地址,再加上 16 位偏移量而得到访问存储器的线性地址,这与实模式形成物理地址的方式相同。但此时没有实模式的地址环绕现象,即允许 10FFEF 这样的线性地址出现。换句话说,V86 模式具有 1MB + 64KB 的线性存储空间。另外要注意,V86 模式是一种具有最低特权级(级3)的保护模式。

Pentium 除了上述三种主要工作模式外,还从 80486 继承下来一种称为系统管理模式 SMM(System Management Mode)的新模式。通过软件或测到某种硬件条件时,可由其他模式进入 SMM。SMM 对其他模式总是隐藏的,从而允许处理器在 SMM 模式下以软件完成某种功能,而这些对应用软件甚至对操作系统都是隐藏的。最早引进这个模式是为了笔记本电脑的电源管理模式。在 CPU 没有实质工作进行时系统降低电源功耗,此时就是以 SMM 模式来保护现场。现在 SMM 又增添了新功能,能对实际不在系统的硬件予以虚拟化。

2. 保护模式下的分段地址变换

在这种方式下的虚拟地址(逻辑地址)由 16 位段选择符和 32 位偏移量组成。选择符中的最低两位是请求特权级 RPL(b_1 b_0)其余 14 位用于索引表,相当于"段号"。

Pentium 微处理器将逻辑地址变换成线性地址,变换过程描述如图 4 .37 所示。若在不分页的情况下线性地址即为物理地址。

Pentium 微处理器将 8 字节(64 位)长的段表项称之为描述符(Descriptor),因此段表被称为描述符表。描述符的一般模式示于图 4 .37 中,它包括 32 位段线性基址,20 位的段界限,2 位的描述符特权级 DPL,1 位的出现位 P(表示该段是否已在主存),以及类型位和粒度位 G 等。20 位的段界限指定段的最大长度,G 位指定长度的单位,G = 0 表示以字节为单位,G = 1 表示以 4KB(页面大小)为单位,即段的最大长

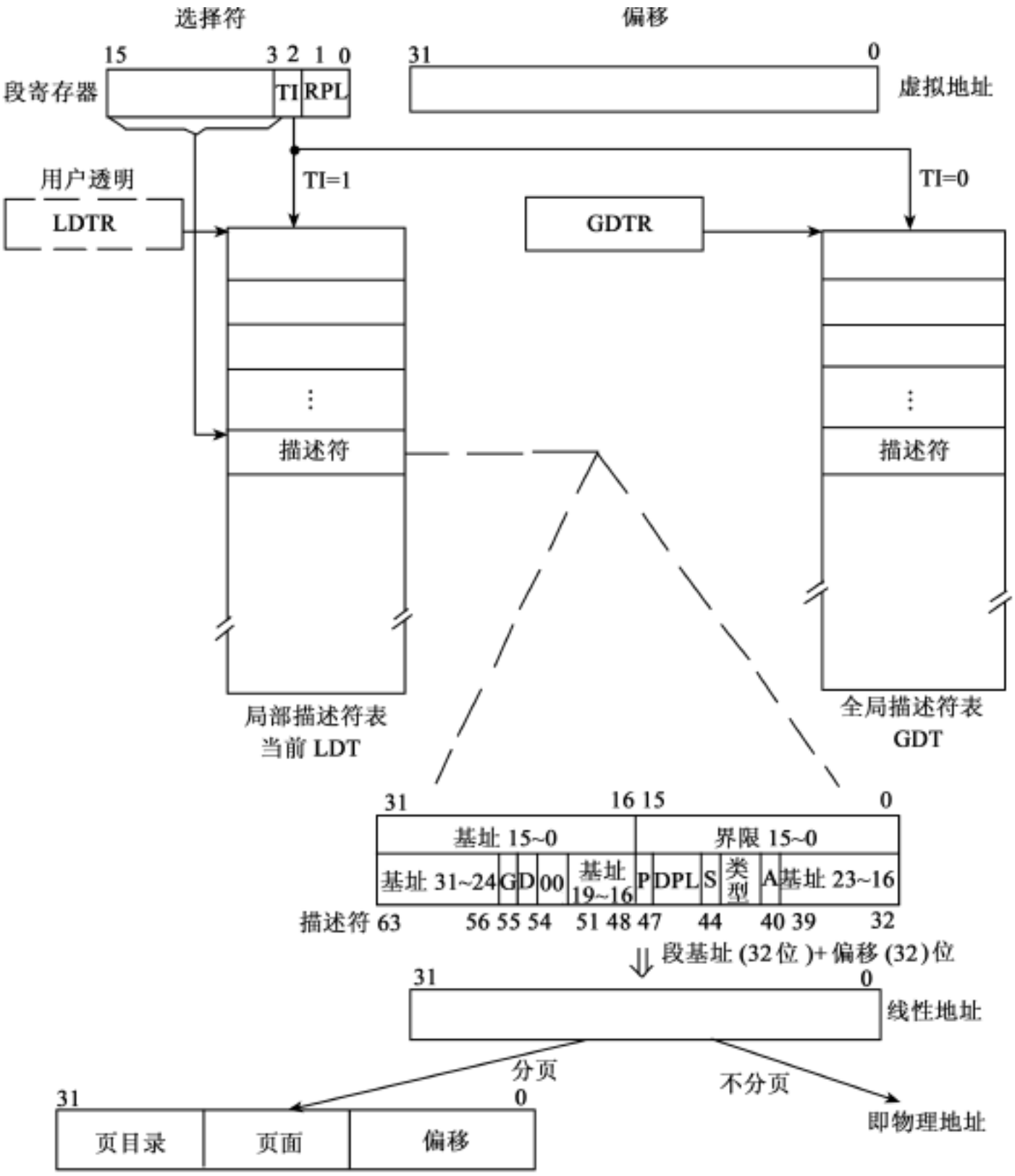


图 4.37　Pentium 的段表及其地址变换过程

度可达 4GB。

Pentium 的保护模式支持多任务环境。每个任务有自己的代码段(CS)、数据段(DS),还可有一个或多个附加数据段(ES, FS, GS)以及堆栈段(SS)等。这些段的描述符组成局部描述符表 LDT。各任务公用的代码段、数据段的描述符,以及系统段(如 LDT、任务状态段 TSS 等)的描述符组成全局描述符表 GDT。全系统只有一个

GDT,有多个 LDT。GDT 由 GDTR 指向,48 位的 GDTR 包括 32 位的 GDT 的基地址和 16 位的表界限(GDT 不超过 64KB)。当前任务使用的 LDT 由 LDTR 寄存器指向,但 LDTR 是用户不可见的。当任务发生切换时由操作系统访问 GDT 取得此任务的 LDT 描述符,将表基址、表界限、属性等装入 LDTR。

因此,一个任务可使用两级段表:一个是全系统各任务共同的全局描述符表 GDT;另一个是任务自己的局部描述符表 LDT。当前具体使用哪个段表由选择符的 TI 位指示,TI 位 = 0 使用 GDT,TI 位 = 1 使用 LDT。

需要说明的是,全系统还有一个中断描述符表 IDT。系统中所用的每类中断在 IDT 中都有一个描述符,最多可有 256 个描述符。每个描述符也是 8 字节,包含相应的中断服务程序入口地址和特性。通过 INT 指令、外部中断向量和 CPU 内部的异常来访问 IDT。

3. 保护模式下的分页地址转换

无论是分段分页模式下经 SU 送来的 32 位线性地址,还是不分段分页模式下由程序直接给出的 32 位线性地址,都经分页部件 PU 转换成 32 位物理地址。

Pentium 有两种分页方式。一种是 4KB 的页,使用页目录表、页表两级页表进行地址转换,这是从 80386/ 80486 继承下来的分页方式。另一种是 Pentium 新增加的分页方式,页为 4MB 大小,使用单级页表进行地址转换。

(1)4KB 分页方式

32 位的线性地址,基地址空间为 2^{32} = 4GB,页面大小为 4KB(2^{12}),则会有 1M 个页面。若是使用单级页表,1M 个页表项的页表规模太庞大了。Pentium 采用两级页表方式,将线性地址相邻接的 1K 个页面组成一组,使用一个 1K 个表项的页表;这一组页面,在页目录表中有一对应的页目录项,页目录表有 1K 个表项。

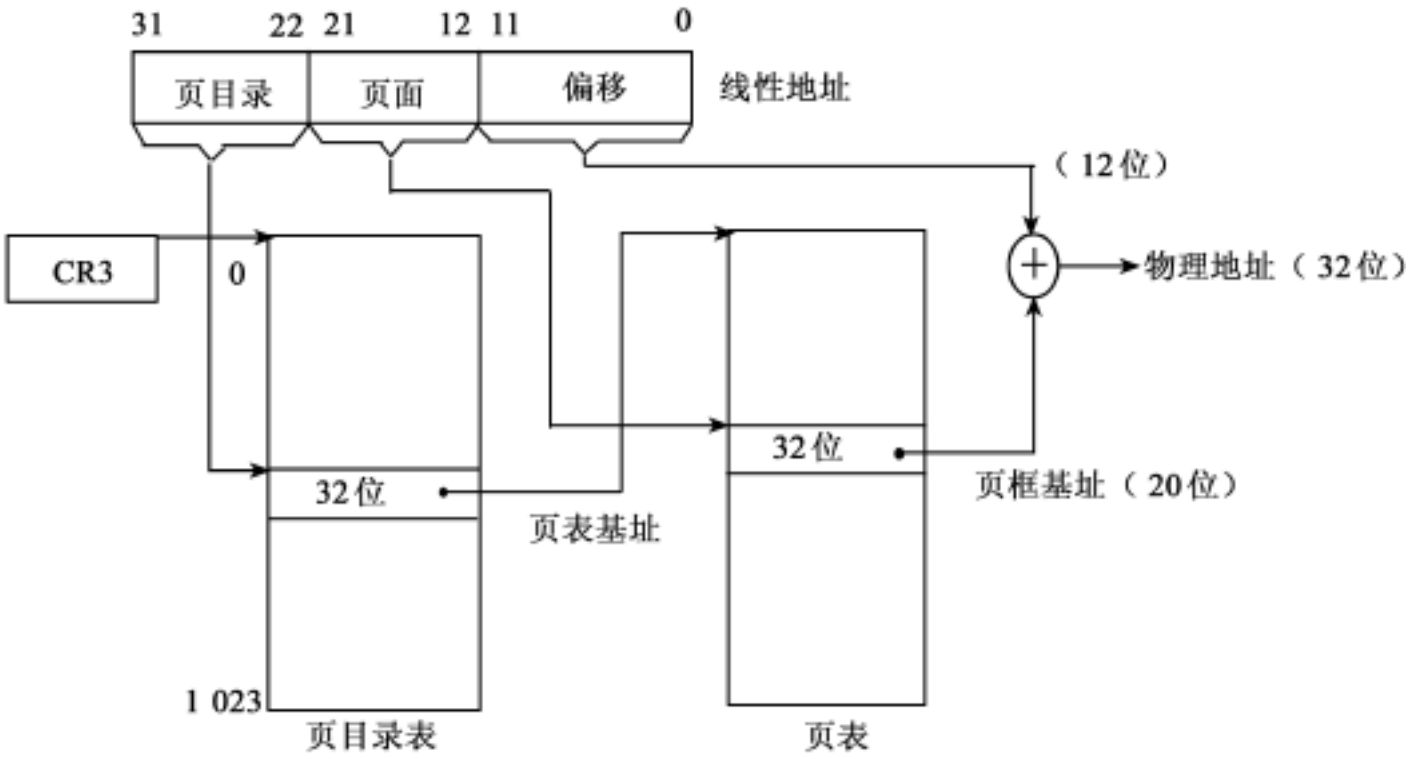
页目录表项和页表项都是 4 字节(32 位),其结构如图 4 .38 所示。这样,页目录表和页表都是 4KB 长,即页框的大小,访问主存寻找页目录表和页表都很方便。

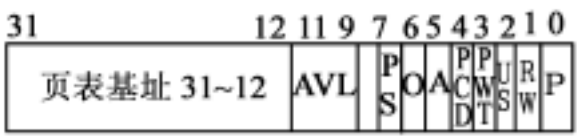
相应地,Pentium 将 32 位线性地址看成是由三个字段组成:高 10 位为页目录(号),中间 10 位为页面(号),最低 12 位为页内字节偏移。

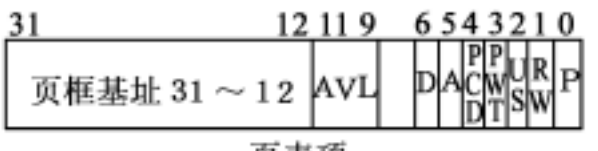
不使用物理地址扩充方式(即 32 位地址方式)时,全系统只有一个页目录表,由 CPU 控制寄存器 CR3 指向此表的起始地址。CPU 首先以线性地址的页目录号部分为索引(×4)查找页目录表,由相应的页目录表项得到其页表的基址;再以线性地址的页面号部分为索引(×4)查找该页表,由相应的页表项才得到分配给此页面的主存页框基地址。由页框基地址(20 位)与线性地址中的页内偏移(12 位)相拼接,最终得到所需要的 32 位物理地址。地址转换涉及到两次访问主存,其过程见图 4 .38。

(2)4MB 分页方式

页面大小为 4MB 的分页方式使用单级页表,减少了一次主存访问,地址转换过程加快了。此方式下,32 位线性地址分为高 10 位的页面(号)和低 22 位的页内偏移







其中：P = 出现位，US = 用户/ 监督位，PCD = 页 Cache 禁止位，D =“ 脏 ”位，RW = 读/ 写位，PWT = 页写直达位，A = 访问过位，AVL = 系统程序员可用位。

图 4 .38　Pentium 4KB 分页方式地址转换

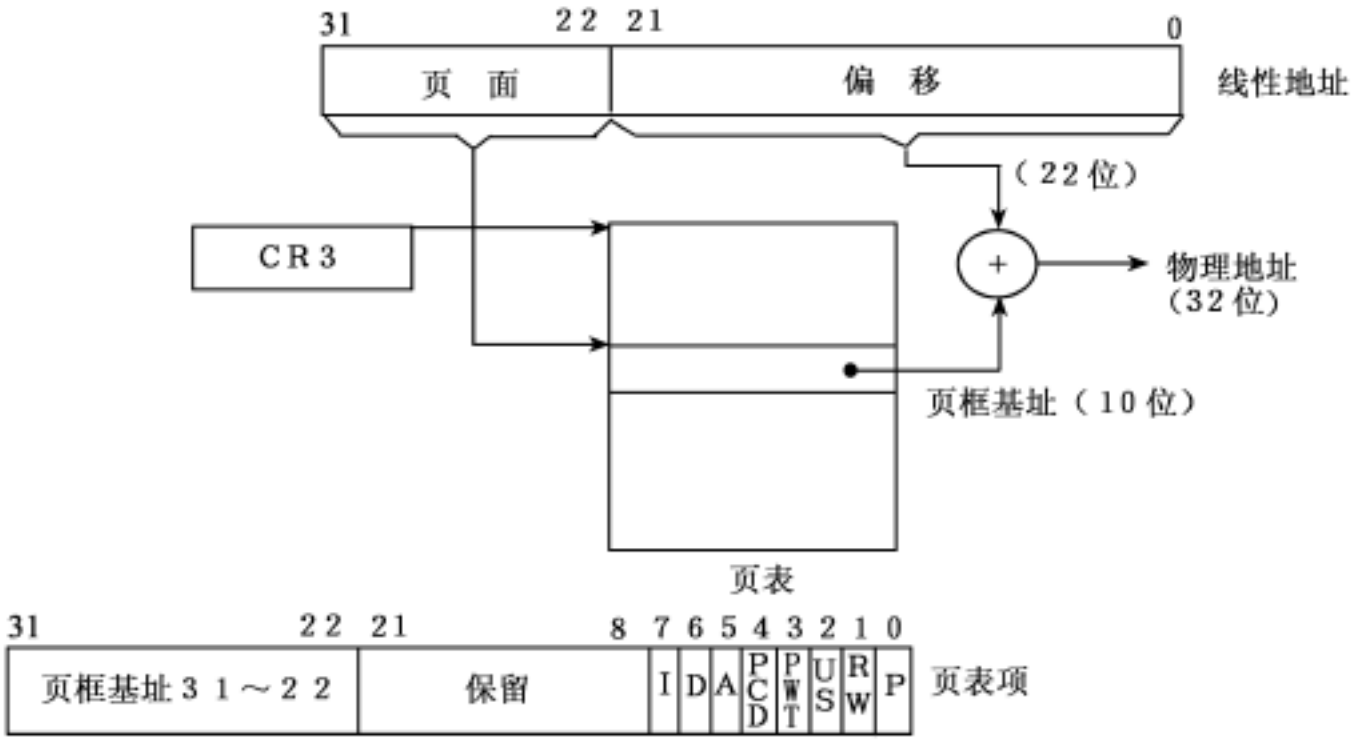
两个字段。32 位地址模式下，全系统只一张页表，由控制寄存器 CR3 指向。此页表有 1K 个表项，每项 4 字节(32 位)。

注意，此时页表项的 b_7 位为 1，此位称为页大小 PS(Page Size)位，为 0 表示是 4KB 页。CPU 以线性地址高 10 位为索引(×4)查找 CR3 指向的表时，若读取表项内容的 b_7 位为 0，知道这是一个页目录表，需要第二次访问主存才能得到 4KB 页的页框基地址。

4MB 分页方式的地址转换过程如图 4 .39 所示。

我们看到，除存取控制位之外，两种分页方式的页表项中都有 P，A，D 三位。P 位为出现位，若该位为 1，表示此页面已装入主存；若该位为 0，对此页的访问将引发缺页(或称页故障)。A 位为访问过位，此页装入主存之后被访问过，置 A 位为 1，否则该位为 0，D 位为“ 脏 ”位，即修改过位，该页被替换时，若 D 位为 1，则需将此页写回磁盘，否则弃之即可。

另外，页表项中还有 PCD(页 Cache 禁止)和 PWT(页写直达)两位，用于页级的是否禁止 Cache 以及采用的是写直达法还是写回法的控制(有关此内容请参见



其中:P = 出现位,RW = 读/写位,US = 用户/监督位,PWT = 页写直达位,PCD = 页 Cache 禁止位,A = 访问过位　D =“脏”位。

图 4.39　Pentium 4MB 分页方式地址转换

Cache 存储器部分)。Pentium 有两个输出引脚 PCD 和 PWT,其生成逻辑如图 4.40 所示。

这两个信号既用于对片外(L2)Cache 的控制,也在 CPU 内部使用,以控制片内(L1)Cache 的状态转换以及写修改方式。

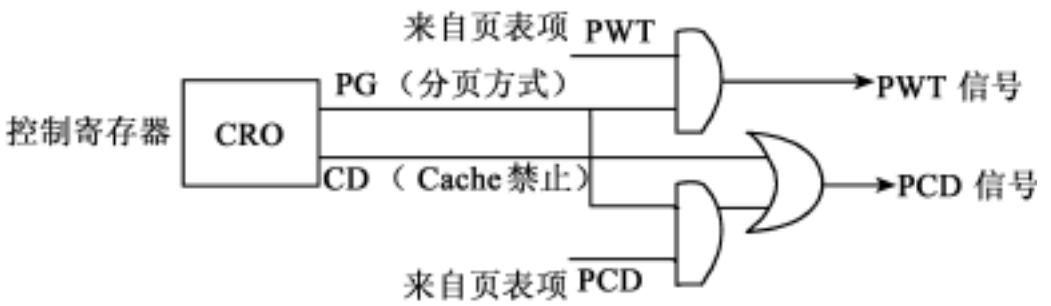


图 4.40　PCD、PWT 信号生成逻辑

4.3　高速缓冲存储器(Cache)

1967 年 Gibson 提出高速缓冲存储器(Cache)技术,1969 年首先在 IBM 360/80 计算机上得以实现。现在,这一技术不仅为大、中型计算机而且也为小型、微型计算机广泛采用。

一般的处理机中只有一级 Cache,它与主存储器构成一个两级的存储系统。一些高性能的处理机都采用两级 Cache,第一级 Cache 在 CPU 内部,容量很小,速度很

快;第二级 Cache 在主板上,容量较大,速度比第一级要低 5 倍左右。也有部分高性能的处理机采用三级 Cache,前两级在 CPU 内部。本节主要介绍由一级 Cache 与主存构成的 Cache 存储系统(以下简称 Cache)。

4.3.1 Cache 工作原理

前面已经提及 Cache 与主存储器相比容量较小,它所保存的信息只是主存内容的一个子集。特别要注意的是,CPU 与 Cache 之间的信息交换是以“字”为单位,而 Cache 与主存之间的信息交换是以“块”为单位,一个块由若干字组成,是定长的。块的大小通常取一个主存周期所访问到的字节数。例如,在 IBM370/168 计算机中,主存储器采用低位交叉存取方式,即主存共有四个存储体,每个存储体字长为 8 字节(32 位)。因而相应块的大小为 32 字节。Cache 主存结构示意图如图 4.41 所示。

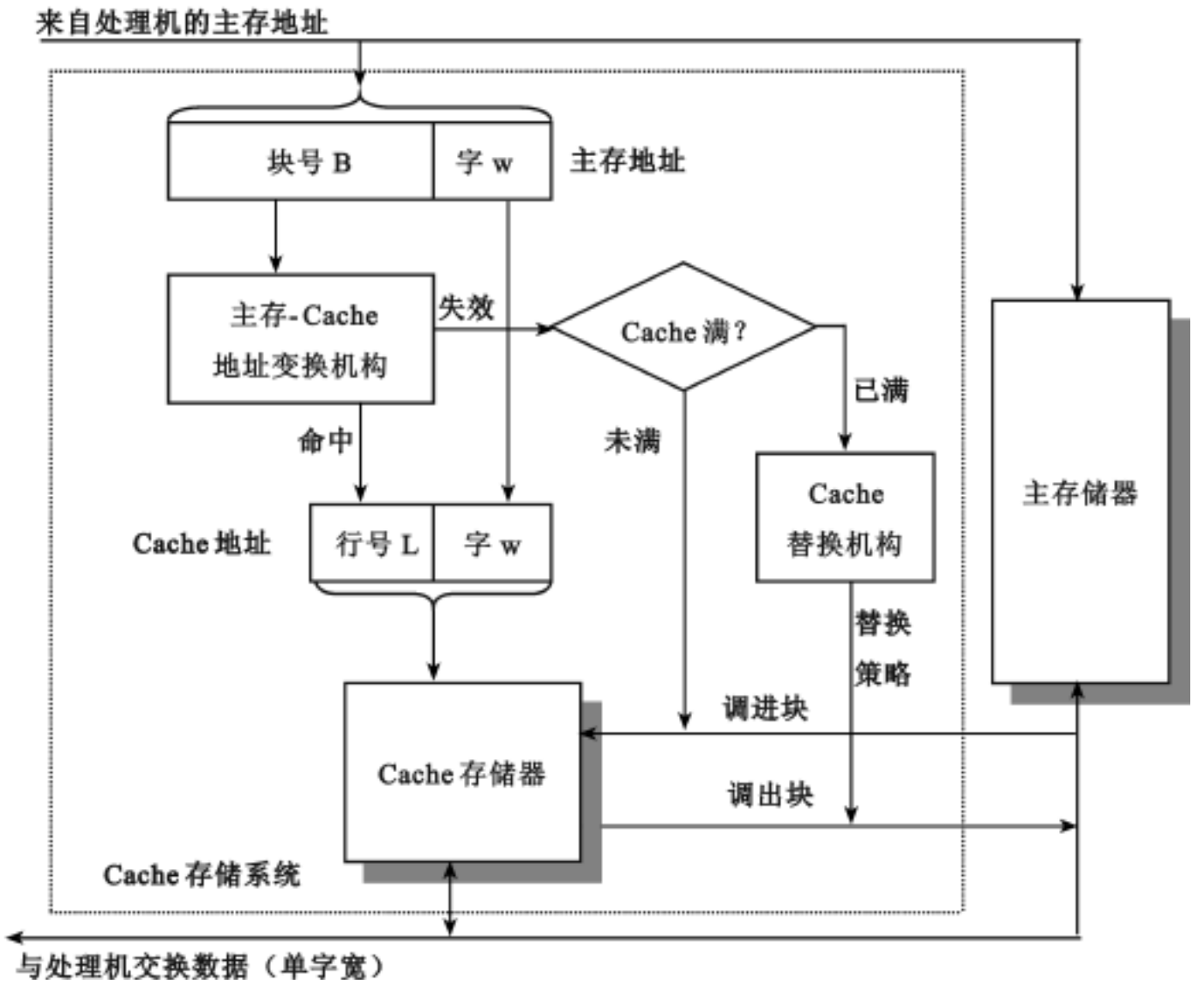


图 4.41 Cache-主存系统结构示意图

当 CPU 试图读取主存一个字时,发出此字内存地址到 Cache 和主存,此时 Cache 控制逻辑依据地址进行判断此字当前是否在 Cache 中。若是,此字立即递交给 CPU,否则,要用主存读取周期把这个字从主存读出送到 CPU,与此同时把含有

这个字的整个数据块从主存读出送到 Cache 中。由于程序的存储器访问具有局部性,当为满足一次访问需求而取来一数据块时,下面的多次访问很可能读取此块中的其他字。

4.3.2 地址映像与地址变换

与页式虚拟存储器类似,Cache 存储系统中也有地址映像与地址变换、替换算法等问题。而地址映像与地址变换是紧密相关的,采用什么样的映像规则,就必然有与之相关的地址变换方式。因此,必须将它们放在一起介绍。

与页式虚拟存储器的显著区别是,Cache 存储系统的地址映像与变换、替换算法的实现等全部由硬件实现,因此,它不仅对应用程序员透明,而且对系统程序员也透明。

在由 Cache—主存储器组成的 Cache 存储系统中,将主存储器分成 n 块,块号用 B_j 表示($j = 0,1,2,\cdots,n-1$),将 Cache 存储器分成 m 行,行号用 L_i 表示($i = 0,1,2,\cdots,m-1$)。Cache 的行大小与主存的块大小相等,即行的字节数和块的字节数等长,但是 mn n。结构图如图 4 .42 所示。图中 Cache 共有 n 行,每行可存放 k 个字,标记(Tag)用于存放对应主存块地址。

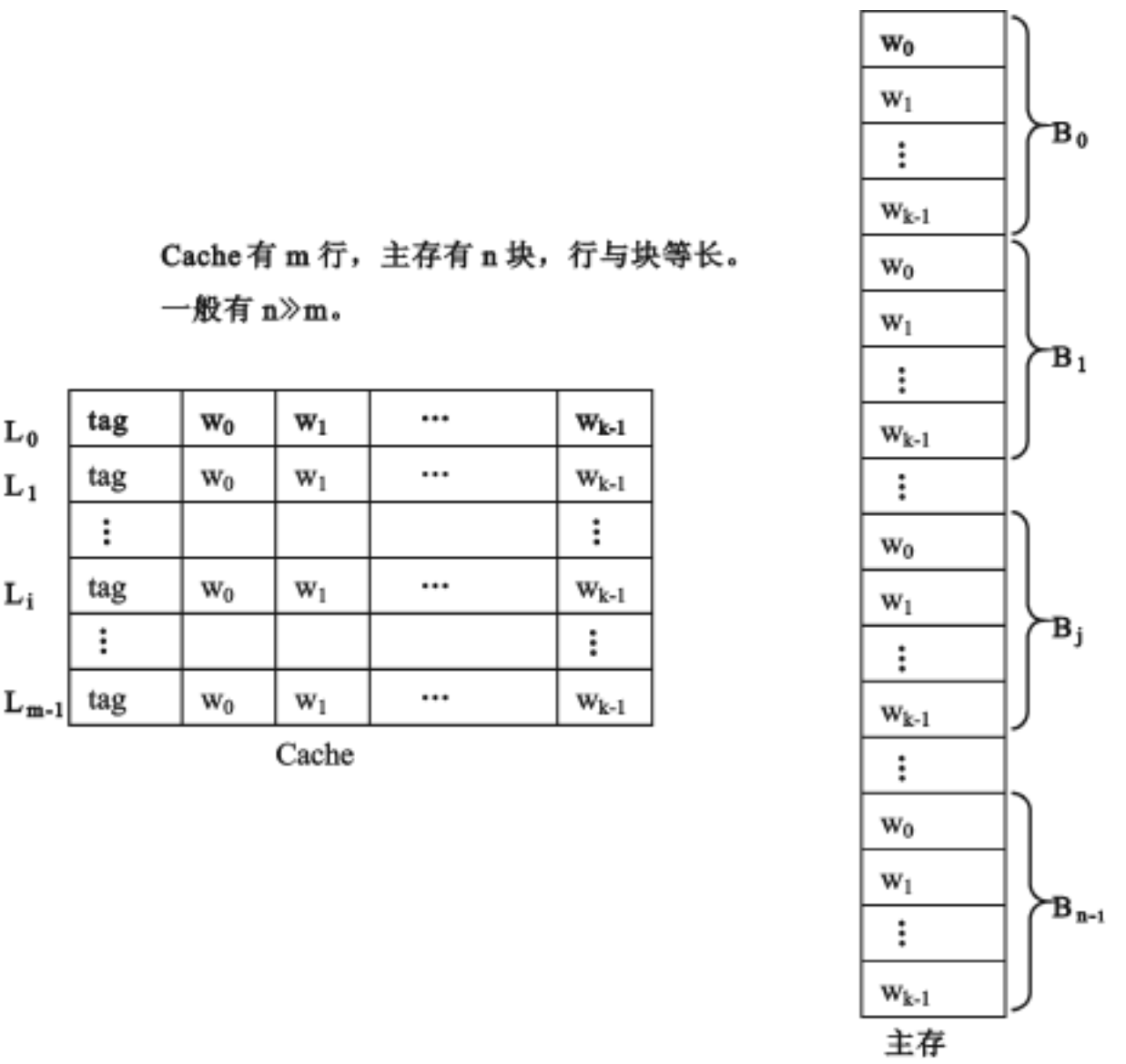


图 4 .42 Cache—主存结构示意图

下面以 Cache 大小为 8 行、主存的大小为 256 块为例,分别介绍三种主要的地址映像和地址变换过程。

1. 全相联映像与变换

全相联映像(Fully Associative-Mapping)常简称为相联映像,这种方式是将一个主存块的地址(块号)与块的内容一起都存于 Cache 行中。块地址保存于 Cache 行的标记(Tag)部分中。这种带着全部地址一起保存的做法,直接结果是一个块可以拷贝到 Cache 的任意一行上,极其灵活。全相联映像关系举例如图 4.43(a)所示。由于主存共有 256 块,故 Cache 行中的标记必须用 8 位指示。

地址变换过程如下:

1. 对于一个指定的内存地址将其块号与 Cache 所有行的标记同时比较;
2. 若某行标记与块号相等,则命中,再以字地址 W 检索到指定字单元;
3. 若所有标记都不相等,即失效,就用主存地址访问主存。

全相联映像方式的主要缺点是比较器电路难以设计与实现,尤其是 Cache 的行数较多时。一般是将全部标记用一个相联存储器实现,全部数据用一个普通 RAM 实现,即全相联 Cache 由一个相联存储器和一个 RAM 组成。相联存储器价格贵、容量小。全相联映像方式只适合于小容量 Cache 采用。

2. 直接映像与变换

直接映像(Direct-Mapping)方式虽然也是多对一的映像关系,但一个主存块只能拷贝到 Cache 的一个特定行位置上去。块号 j 与能保存此块的行号 i 有如下关系:

$$i = j \bmod m \quad (m \text{ 是 Cache 总行数})$$

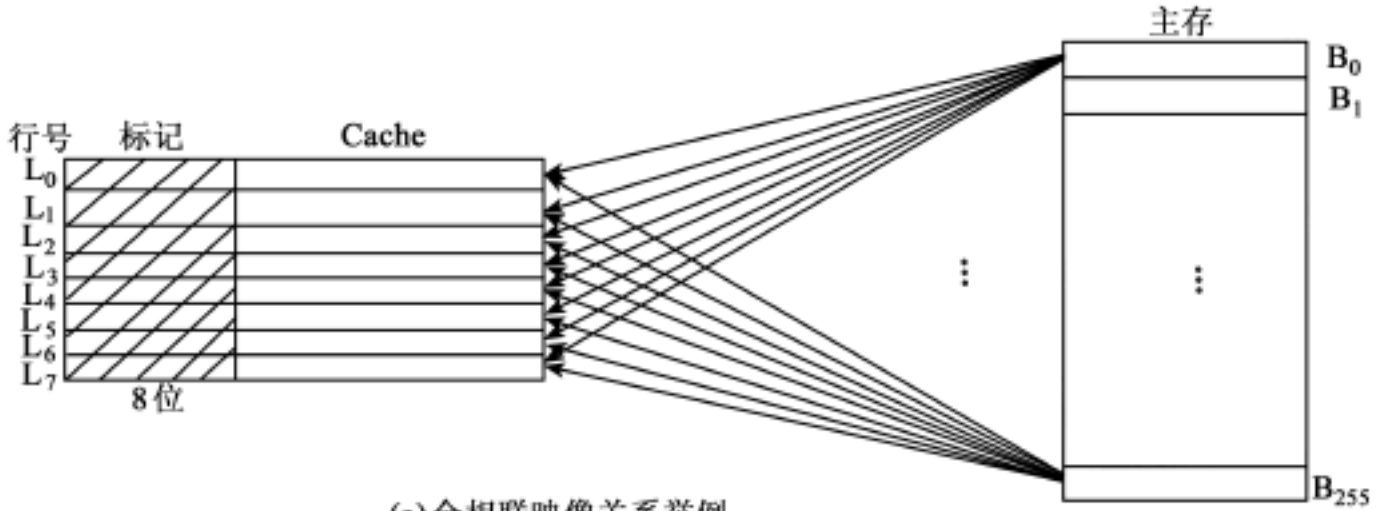
此方式是将 s 位的块地址分成两部分,低序的 r 位作为 Cache 的行地址,高序的 s-r 位作为标记(Tag)与块数据一起保存在该行。直接映像举例如图 4.44(a)所示,由于 Cache 为 8 行,主存 256 块,256 ÷ 8 = 32,所以,Cache 标记位只用于存放 32 组中的任一组号,仅用 5 位。

直接映像地址变换检索过程如下:

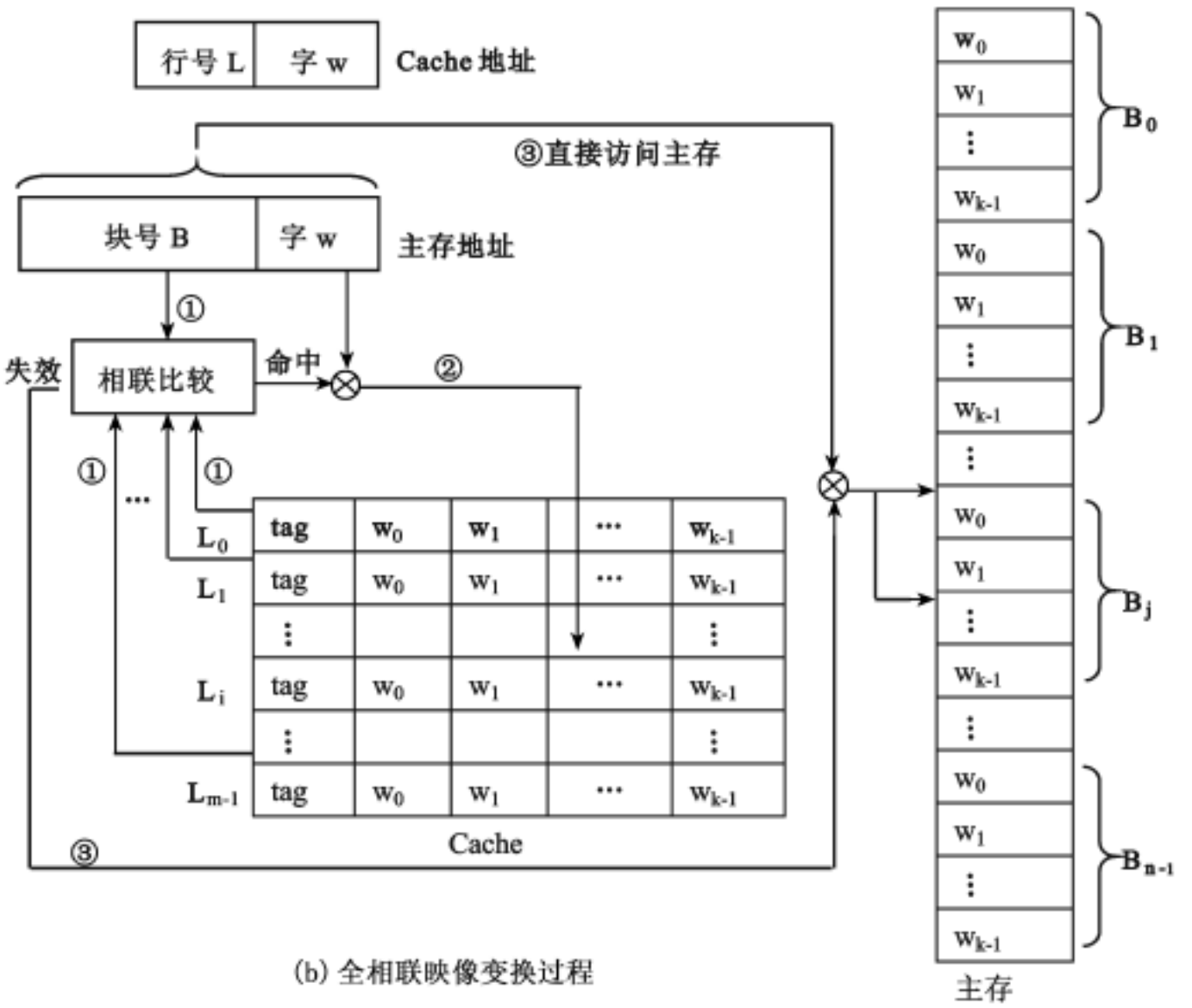
- 以主存地址中的 r 位行号,找到此 Cache 行。
- 取出 Cache 行中的标记字与主存地址中的标记作相等比较。
- 若比较相等,则命中。以主存地址中的最低 W 位查找访问指定单元内容。
- 若比较不相等,则失效。用主存地址直接访问主存。

直接映像地址检索过程如图 4.44(b)所示。

直接映像方式的优点是硬件简单,成本低;缺点是每个主存块只有一个固定的行位置可存放。如果块号相距 m 整数倍的两个块,要存于同一 Cache 行时,就要发生冲突。此时,Cache 其他位置即使有空行也用不上。发生冲突时就要将原先已存入的块换出 Cache,但很可能过一段时间又要换入,频繁换入、换出会使 Cache 效率降



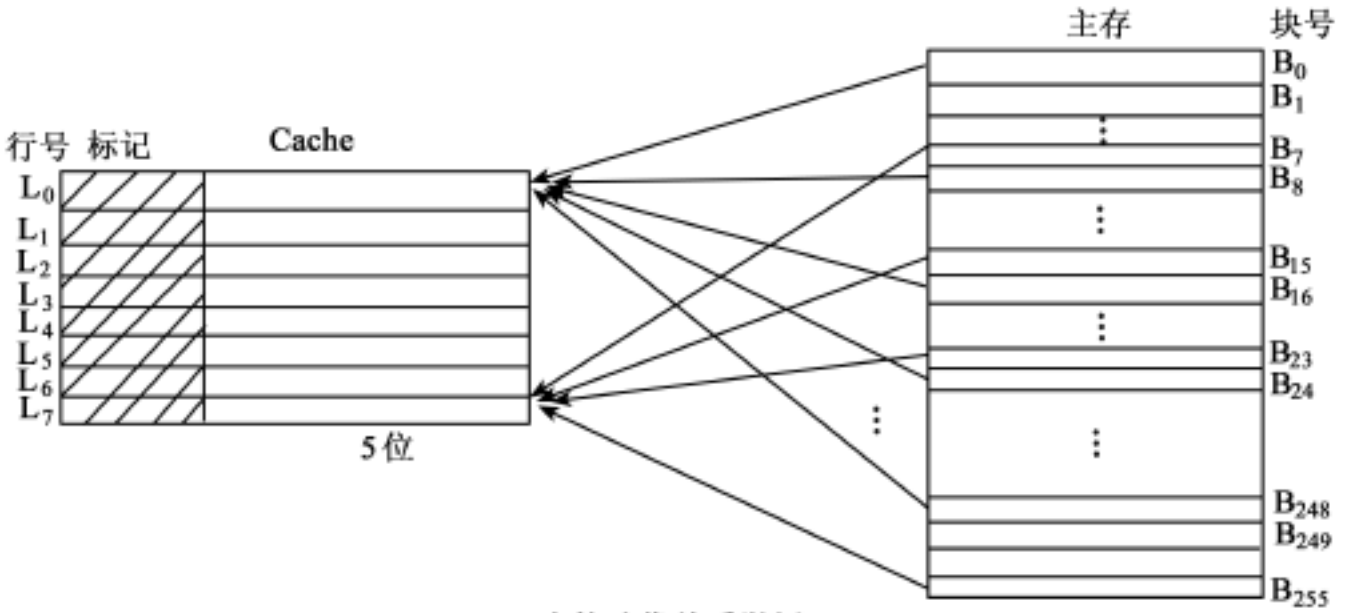
(a)全相联映像关系举例



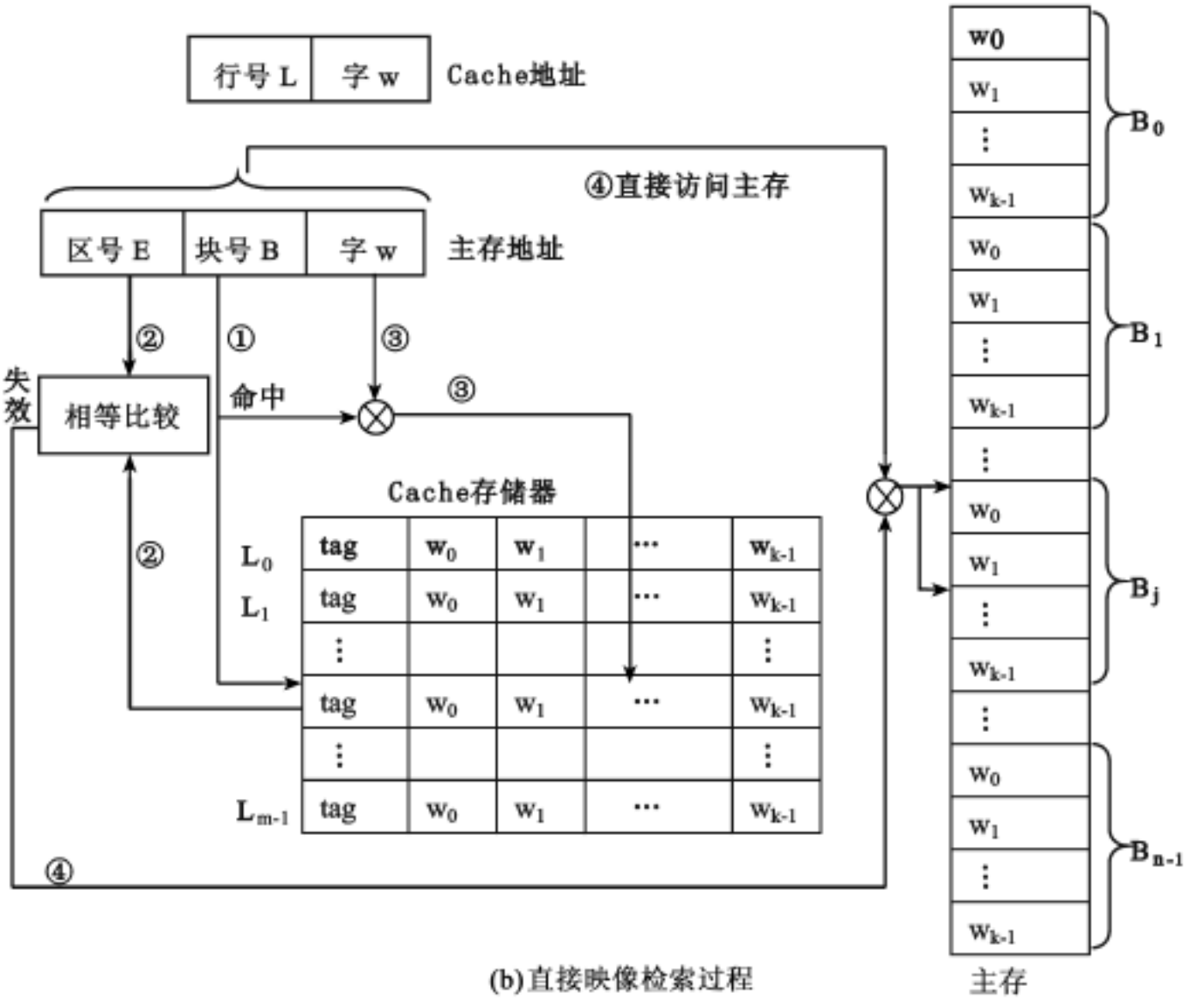
(b) 全相联映像变换过程

图 4 .43 Cache 带标记的全相联地址变换

低。直接映像 Cache 适合于需要大容量 Cache 的场合,更多的行数也可以减小冲突的机会。



(a)直接映像关系举例



(b)直接映像检索过程

图 4.44　Cache 带标记的直接映像方式地址变换

3. 组相联映像与变换

上述两种映像方式的优缺点正好相反。从存放位置的灵活性以及命中率来看，全相联映像方式要好；就比较电路简单以及硬件投资少来看，直接映像方式要优。若有一种方式能适度地兼有两者优点又尽量避免两者缺点就太好了，这个折衷方案就是组相联映像方式(Set-Associative Mapping)。

这种方式是将 Cache 分成 u 组，每组 v 行。主存块存放到哪个组是固定的，至于存放到该组哪个行是随意的。组间直接，组内相联，即有如下关系：

$$n = u \times v, \text{组号 } q = j \bmod u$$

块内存地址中 s 位块号划分成两部分，低序的 d 位($2^d = u$)用于表示 Cache 组号，高序的 s-d 位作为标记(Tag)，与块数据一起位于此组的某行中。

组相联映像方式举例如图 4.45(a)所示。其中，Cache 8 行分成 4 组，每组 2 行。即 u = 4，v = 2，对于指定的主存块号 j(0，1，…，255)，可以映像到 Cache 相应的组号(组内任意行)如表 4.5 所示。

表 4.5

主存块号 j	0	1	2	3	4	5	6	7	…	252	253	254	255
Cache 组号 q	0	1	2	3	0	1	2	3	…	0	1	2	3

组相联的地址变换(检索)过程如图 4.45(b)所示。变换过程描述如下：

① 以主存块号地址低位(d 位)检索到 Cache 相应组。

② 以主存块号地址高位(s-d)位与该组 u 行中的所有标记，同时进行比较(相联比较)。

③ 若相等，则相等行命中，以主存地址的 W 位去检索该行的指定字，即完成对指定单元的R/ W操作。

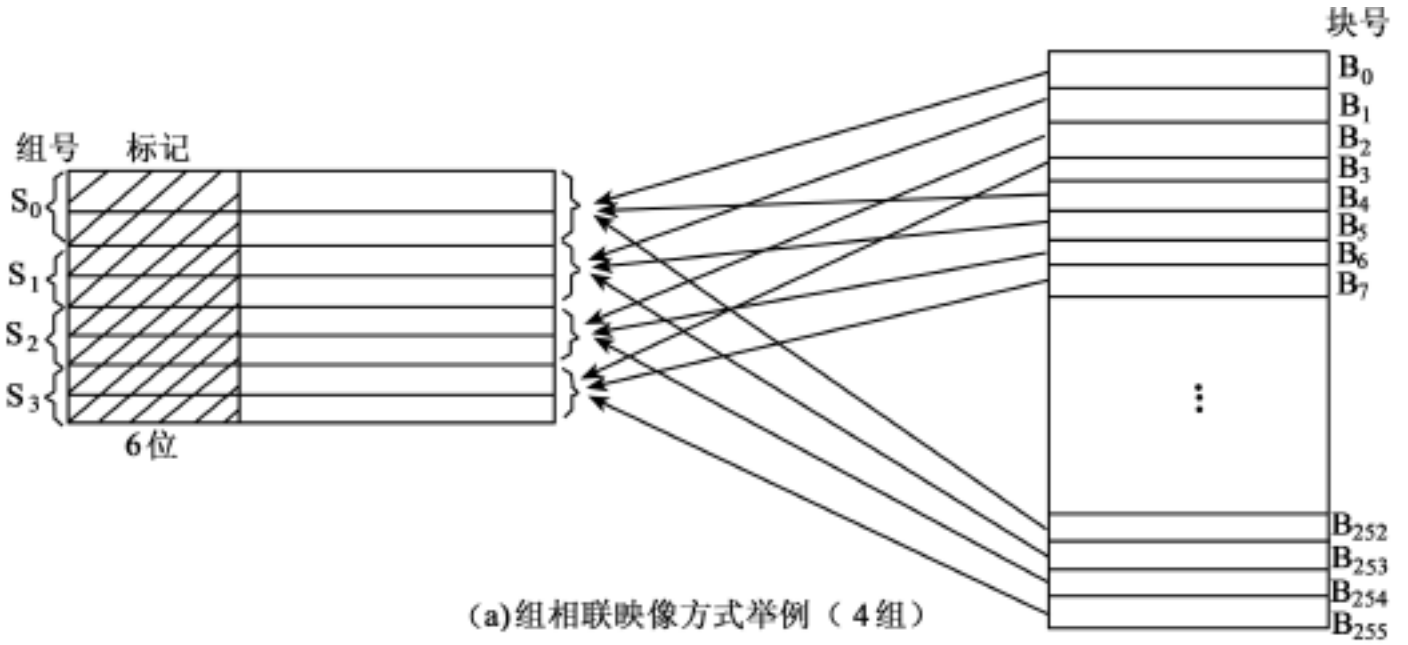
④ 若全不相等，则 Cache 访问失效。以主存地址直接访问主存储器，完成对字的访问和信息块的调入。

组相联映像方式中的每组行数比一般取值都比较小，典型值是 2，4 或 8，最大为 16。这是因为 u 值较小时在设计上容易实现，而且为了强调比较的规模和存放的灵活程度，常称为 u 路组相联 Cache。

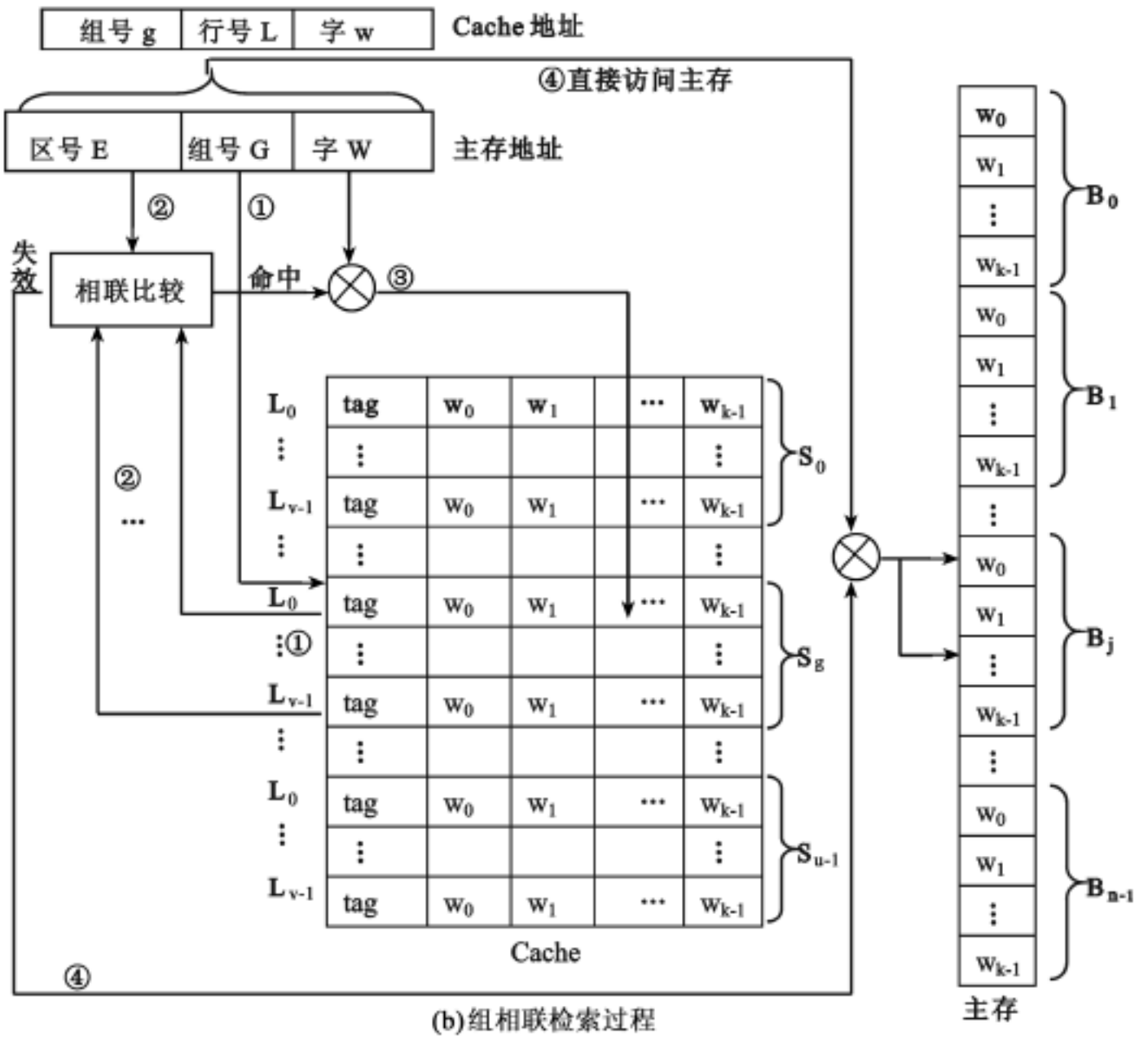
全相联和直接映像是组相联的两种极端情况。因为 m = u×v，若组数 u = 1，全部 Cache 行为一大组，即为全相联映像方式；若行数 v = 1，每组只有一行，即为直接映像。在设计中，根据具体情况，调整 u 和 v 的值，可适当兼顾两者的优点。

4.3.3 Cache 替换算法及其实现

与页式虚拟存储器相比，Cache 存储器也存在行调用和行替换及其替换算法。



(a)组相联映像方式举例（4组）



(b)组相联检索过程

图4.45 Cache带标记的u路组相联映像与变换过程

所不同的是,由于 Cache 速度很高,替换算法必须全部由硬件实现。

替换算法与 Cache 的组织方式紧密相关。对于直接映像方式的 Cache 来说,因为主存块只有一个特定的行位置可存放,所以只需把此特定行位置上的原主存块换出 Cache 即可。对于全相联和组相联的 Cache 来说,就要从允许存放新主存块的若干特定行中选取一行换出。从原理上讲,虚拟存储器中的替换算法,在 Cache 中都能适用,但由于 Cache 的特殊性,使用较多的是 LRU(LFU)替换算法。下面介绍这种算法的几种实现方法。

1. 计数器法

这种实现方法是在 Cache 的每行中设置一个计数器,计数的位数与 Cache 组内行数相关。如 Cache 每组为 4 行,则计数器有两位二进制计数。

计数器的使用及管理规则是:

① 初始化:各计数器清零。

② 调进:新调进块的行计数器清零,其余加 1。

③ 替换:被替换的行计数器清零,其余加 1。

④ 命中:被命中的行计数器清零。凡计数值小于命中行计数值则加 1,其余计数值不变。

例如在 IBM 370/ 165 机中的 Cache 采用组相联映像方式。每组 4 行,每行设置有一个 2 位计数器。在访问 Cache 的过程中,主存块的装入、命中及替换时,以及计数器的工作情况,如表 4 .6 所示。

表 4 .6　　计数器实现 LRU 过程

块地址流	主存块 1		主存块 2		主存块 3		主存块 4		主存块 5		主存块 6	
Cache 行	块号	计数器	块号	计数器	块号	计数器	块号	计数器	块号	计数器	块号	计数器
0	1	00	1	01	1	10	1	11	5	00	5	01
1		01	2	00	2	01	2	10	2	11	2	11
2		01		10	3	00	3	01	3	10	3	10
3		01		10		11	4	00	4	01	4	00
	调进		调进		调进		调进		替换		命中	

LRU 算法硬件实现不是非常困难,特别是对于两路组相联的 Cache 而言,情况会大为简化。因为一个主存块只能在一个特定组的两行中做出选择,完全不需要计数器,一组内的两行只需一个二进位即可。如果规定一组中的 A 行拷贝进新数据则置二进位。另一行(B 行)拷贝进新数据将此位置零。那么当需要替换时,只需检查

此二进位(乒乓位)的状态即可,为 0 换出 A 行,为 1 换出 B 行,实现了保护新内容的原则。后面将看到,Pentium 处理器内的数据 Cache 是一个两路组相联结构,采用的是 LRU 替换策略。

2. 堆栈法

在页式虚拟存储器中曾经介绍过,LRU 算法是堆栈型替换算法,对于 LRU 算法,堆栈 S_t 中由栈顶到栈底的各项(行)恒反映出在 t 时刻,实存中各页被访问过的近远次序,以及每访问一页,堆栈 S_t 中各项的变化过程。结果是此堆栈的栈顶恒存放近期最近访问过的页的页号,而栈底恒存放近期最久未访问过的页的页号,即准备被替换掉的页的页号。那么在 Cache 存储器中完全可按此思想组成一个实际的硬件堆栈。对于组相联映像,每组只需设置一个容量等于组内行数的堆栈,如图 4 .46 所示。每次把刚访问的 Cache 行号与堆栈中的各个行号相联比较。如果没有相符的,将此行号压入堆栈成为栈顶项,原已在堆栈中的各项都顺次下移一行,如果有相符的,将此行号从堆栈中取出再压入堆栈成新的栈顶,并使原先存放该行号的项到栈顶的那些项均下移一行,直到堆栈被全部装满(即 Cache 各行位置全被占用)后又发生失效时,栈底项存放的行号就作为被替换的 Cache 行号。

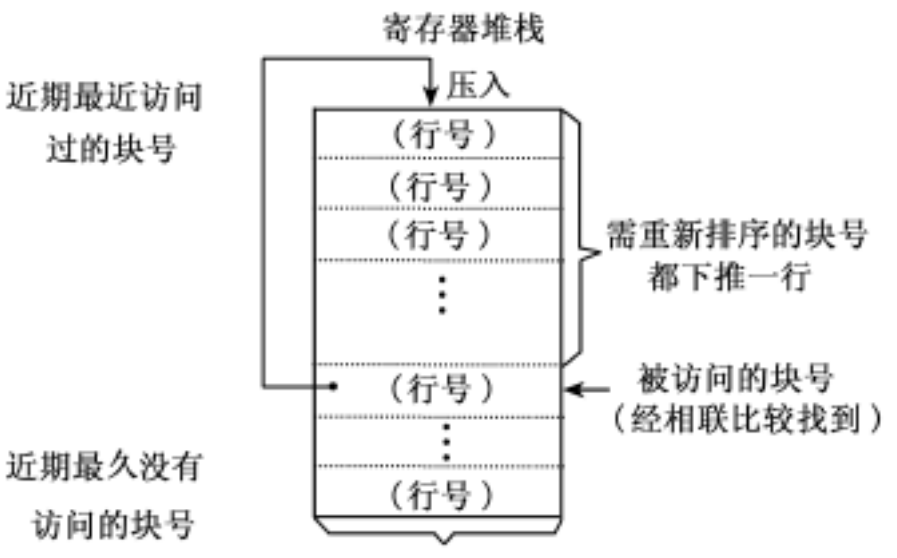


图 4 .46　LRU 算法经堆栈实现(需有相联比较功能)

由于这种硬件堆栈既要有相联比较的功能,又要有能全部下移、部分下移和从中间取出一项的功能,成本较高,因此只适用于组相联且组内块数较少的 LRU 算法替换场合。

为了避免栈中各行内容要同时下移,以节省成本,也可采用另一种变形,让存放行号的寄存器的几何位置与 Cache 中的行号对应,用寄存器存放值的大小表示该块已被访问过的远近次序。以组相联映像为例,每组均使用如图 4 .47 那样的一个寄存器组,由 2^v 个寄存器组成,每个寄存器为 v 位宽,可以存放表示从 0-2^v-1 的次序值。

如果让愈是最近访问的,其次序值愈小,则替换块的确定就只需通过相联比较求最大值,即可找到该最大值所在的寄存器号,也就是对应 Cache 中应被替换行的行号。

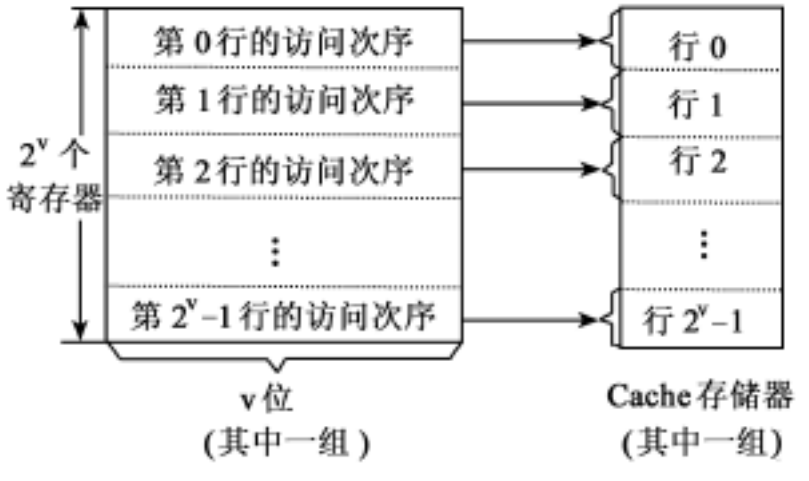


图 4.47 组相联 LRU 算法经寄存器实现
(每组一个,需有相联比较功能)

3. 比较对法

堆栈法需要硬件有相联比较的功能,因此其速度较低,也比较贵。那么,能否不用相联比较,只用一般的门、触发器来实现 LRU 替换算法呢?比较对法就是其中的一种。

比较对法的基本思路是让各行成对组合,用一个触发器的状态表示该比较对内两行访问的远近次序,再经门电路就可找到 LRU 行。如有 A,B,C 三行,组成 AB,AC,BC 三对。各对内行的访问顺序分别用"对触发器"T_{AB},T_{AC},T_{BC} 表示。T_{AB} 为"1",表示 A 比 B 更近被访问过;T_{AB} 为 0,表示 B 比 A 更近被访问过。T_{AC},T_{BC} 也类似此定义。这样,当访问过的次序为 ABC,即最近被访问过的为 A,最久未被访问的是 C,则这三个触发器状态是 $T_{AB}=1$,$T_{AC}=1$,$T_{BC}=1$。如果访问过的次序为 BAC,C 为最久未被访问过,则有 $T_{AB}=0$,$T_{AC}=1$,$T_{BC}=1$。因此 C 作为最久未被访问过的替换块,用布尔代数式必有

$$C_{LRU}=T_{AB}\cdot T_{AC}\cdot T_{BC}+T_{AB}\cdot T_{AC}\cdot T_{BC}=T_{AC}\cdot T_{BC}$$

同理可得

$$B_{LRU}=T_{AB}\cdot T_{BC}$$

$$A_{LRU}=T_{AB}\cdot T_{AC}$$

因此,完全可以用与门、触发器等硬件组合实现,如图 4.48 所示。

显然,每次访问某行时,应改变与该行有关的比较对触发器的状态。以上述 A,B,C 三行为例,每次访问 A 后需改变与 A 有关的比较对触发器的状态,置 T_{AB},T_{AC} 为 1,以反映 A 比 B 更近、A 比 C 更近访问过;访问 C 后,置 T_{AC},T_{BC} 为 0。据此可定出各个触发器的输入控制逻辑,如图 4.48 所示。

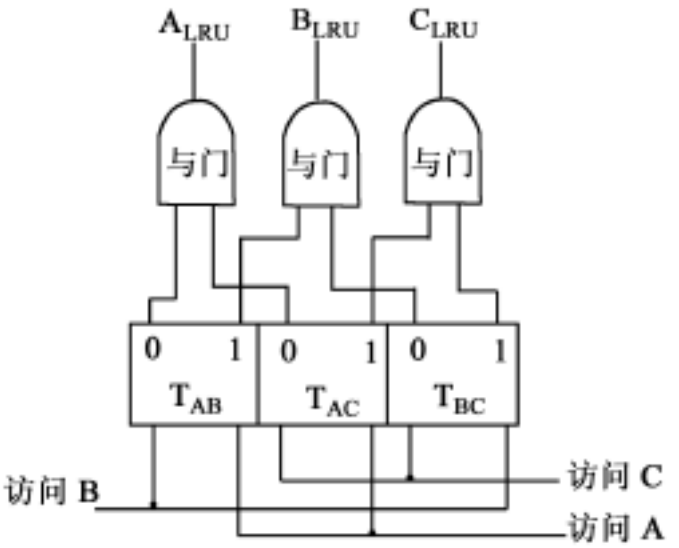


图 4 .48　用比较对法实现 LRU 算法

现在来分析比较对法所用的硬件。由于每行均可能作为 LRU 行,其信号需用与门产生,所以有多少行,就得有多少个与门;每个与门接收与它有关的对触发器来的输入,例如 A_{LRU} 与门要有从 T_{AB}, T_{AC} 来的输入, B_{LRU} 要有从 T_{AB}, T_{BC} 来的输入,而与每块有关的对触发器数为块数减 1,所以与门的输入端数是行数减 1。若 P 为行数,两两组合,比较对触发器数为 $C_P^2 = P \cdot (P - 1)/2$。表 4 .7 列出了比较对法行数 P 与门数、门的输入端数及比较对触发器数的关系。

表 4 .7　　比较对触发器数、门数、门的输入端数与行数的关系

行数	3	4	8	16	64	256	…	P
比较对触发器数	3	6	28	120	2 016	32 640	…	P(P - 1)/ 2
门数	3	4	8	16	64	256	…	P
门输入端数	2	3	7	15	63	255	…	P - 1

从 4. 7 表可以看出,比较对触发器的个数会随块数增多以极快的速度增加,门的输入端数也线性增加,这在工程实现上会带来麻烦,所以比较对法只适用于组内行数较少的组相联映像 Cache 存储器。在行数少时,它比前述堆栈法易于实现。若组内行数超过 8,则所需比较对数就多到不能承受了。不过这时也还可以用多级状态位技术来减少所用的比较对数。

4. 3. 4　Cache 一致性与写策略

在由 Cache 和主存储器共同构成的 Cache 存储系统中,从工作原理可知, CPU 只对 Cache 某行信息进行修改,而主存储器内容却没有修改,而其他的主控者(处理

器、I/O控制器等)全由主存储器得到过时的信息。同样,若其他主控者修改主存储器信息,而Cache却没有得到修改相应行的通知,CPU会由Cache得到过时的信息。这两方面都会导致Cache与主存内容不一致,也就是Cache的一致性问题。

为了维护Cache的一致性必须从两个方面入手,一方面涉及到处理器所采用的Cache写策略,另一方面涉及到处理器所采用的Cache一致性协议。

1. 写直达法(Write-Through)

这一策略的中文译名不尽相同,有的称全写法,有的称写贯通法。顾名思义,它的做法是当Cache写命中时,Cache与主存同时发生写修改。这种策略显然较好地维护了Cache与主存的内容一致性,但这并不等于说全部解决了一致性问题。例如在多处理器系统中各CPU都有自己的Cache,一个主存块若在多个Cache中都有一份拷贝的话,某个CPU以写直达法来修改它的Cache和主存时,其他Cache中的原拷贝就过时了。即使在单处理器系统中,也有I/O设备不经过Cache向主存写入的情况。总之,仍要关注一致性问题。

当Cache写失效时有写分配与写不分配之分。写分配是指完成了对主存修改后,将修改行装入Cache,即为写失效的Cache行分配一个新行。写不分配是指仅对存储器修改,而再无其他动作。

写直达法是写Cache与写主存同步进行,其优点是Cache每行无需设置一个修改位以及相应的判测逻辑。80486处理器内的Cache采用的就是写直达法,这也可从其组织结构中无修改位看出。写直达法的缺点是,Cache对CPU向主存的写操作无高速缓冲功能,降低了Cache的功效。

2. 写回法(Write-Back)

使用这种策略,当CPU对Cache写命中时,只修改Cache的内容不立即写入主存,只当此行被换出时才写回主存。这种策略使Cache在CPU-主存之间,不仅在读方向而且在写方向上都起到高速缓存作用。对一Cache行的多次写命中都在Cache中快速完成修改,只是需被替换时才写回主存,减少了访问主存的次数从而提高了效率。为支持这种策略,每个Cache行必须配置一个修改位,以反映此行是否被CPU修改过。当某行被换出时,根据此行修改位为1还是为0,决定是将该行内容写回主存还是简单地弃之而不顾。

当Cache写失效时,写回法也有写分配与写不分配。写分配是指由主存读取此行(失效行),调入Cache后再完成对此行的修改(主存对应块没有修改)。写不分配是指将处理器的写请求递交给主存,在主存中完成修改,修改后的块也不调入Cache。

一般而言,写直达法多采用不分配法,而写回法多采用写分配法。

3. 写一次法(Write-Once)

写一次法是一种基于写回法又结合了写直达法的写策略,即写命中和写未命中的处理与写回法基本相同,只是第一次写命中时要同时写入主存。这种策略主要用于某些处理器的片内 Cache,例如 Pentium 处理器的片内 Cache 采用的就是写一次法。因为片内 Cache 写命中时,写操作就在 CPU 内部高速完成,若没有内存地址及其他指示信号送出,就不便于系统中的其他 Cache 监听。采用写一次法,在第一次片内 Cache 写命中时,CPU 要在总线上启动一个存储写周期。其他 Cache 监听到此主存块地址及写信号后,即可把它们各自保存的该块拷贝及时作废(无效处理)。而后若有对片内 Cache 此行的再次或多次写命中,则按回写法处理,无需再送出信号了。这样虽然第一次写命中时花费了一个存储周期,但对维护系统全部 Cache 的一致性有利,而大多数的 Cache 写操作不涉及到片外,对指令流水执行有利。

4. MESI 协议

当代多处理机系统中,每个处理机都有自己各自的 Cache,各节点通过总线、交叉开关等互连机构连在一起,如图 4 .49 所示。同一主存的拷贝能同时存于不同的 Cache 中,若允许各个处理器各自独立地修改自己的 Cache,就会出现修改后的 Cache 与主存内容不一致的问题。解决多处理机系统中 Cache 一致性方法的协议,分为两类:

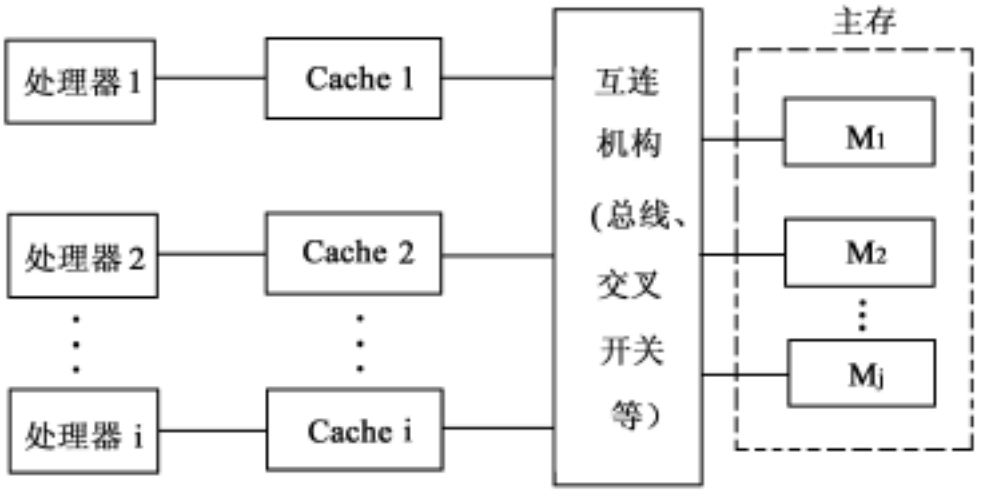


图 4 .49　多处理器系统中的 Cache/ 主存结构

目录协议(Directory Protocol):它由位于主存的目录来保存有关各个局部 Cache 的全局性状态信息,并由一个集中式的主存-Cache 控制器来维护 Cache 的一致性。

监听协议(Snop Protocol):它是将维护 Cache 一致性的责任分散到各个 Cache 控制器。每个控制器必须识别出它的 Cache 中哪些块是与其他 Cache 共享的。当修

改一个共享块时必须在系统中广播有关信息,其他 Cache 控制器监听到此信息时予以响应。根据广播的信息以及反应的不同,监听协议又分为写—修改协议和写—无效协议两种方式。

写-修改(Write-Update)协议是,某处理器要修改它的 Cache 中一个共享块时要广播具体的修改字及地址,容纳有此共享块的各个 Cache 同时予以修改。写-无效(Write-Invalidate)协议是,某处理器要修改它的 Cache 中的一个共享块时,无需广播具体的修改字,只需给出块地址和其他必要的指示信息,令其他 Cache 中的此块变为无效,然后处理器对其 Cache 的此块完成一次本地写操作。

MESI 协议状态转换规则:

MESI 协议是一种采用写—无效方式的监听协议。它要求每个 Cache 行有两个状态位,用于描述该行当前是位于修改态(M)、专用态(E)、共享态(S)或者无效态(I)中的哪种状态,从而决定它的读/ 写操作行为。这四种状态的定义是:

- 修改态(Modified),此 Cache 行已被修改过(脏行),内容已不同于主存并且为此 Cache 专有。
- 专有态(Exclusive),此 Cache 行内容同于主存,但不出现于其他 Cache 中。
- 共享态(Shared),此 Cache 行内容同于主存,但出现于其他 Cache 中。
- 无效态(Invalid),此 Cache 行内容无效(空行)。

MESI 协议适合以总线为互连机构的多处理器系统。各 Cache 控制器除负责响应自己 CPU 的内存读/ 写操作(包括读/ 写命中与未命中)外,还要负责监听总线上的其他 CPU 的内存读/ 写活动(包括读监听命中与写监听命中),并对自己的 Cache 予以相应的处理。所有这些处理过程要维护 Cache 一致性,必须符合图 4 .50 所示的 MESI 协议状态转换规则。

下面由图的四个顶点出发,介绍转换规则:

① 状态为 I 的 Cache 行是内容和标记都无效的空行,没有监听命中与否的问题,只有在 CPU 对 Cache 读/ 写未命中时用来分配给新行。当读未命中时,启动了一个存储读总线周期将读取的主存块填入特定位置的空行中并建标记,还要依据其他 Cache 返回的有/ 无监听命中指示,相应建立 S/ E 状态。当写未命中时,也要由主存读入含欲修改字的块,然后在 Cache 内完成写修改并建新状态 M。然而此时有三点要特别注意:第一,在写未命中时虽然 CPU 启动的是存储读总线周期,但它的 Cache 控制逻辑也同时送出一个“读是用于写”信号 RWITM(Read-With-Intent-To-Modify),通知其他 Cache 将此周期作为写周期来监听。第二,若其他某个 Cache 发生监听命中,而且命中行是 M 态,则这个 Cache 要暂时中止“先读后写”活动,取得总线控制权将此 M 态行写回主存,然后放弃控制权让原先的写请求重新开始先读后写操作。第三,M 态行写回主存后,其状态变为 I;但是若原先的写请求是一种猝发式,即 CPU 写入 Cache 及主存的是一个完整的新行,则此写监听命中的 M 态行没有写回主存的必要,只简单作废即可。

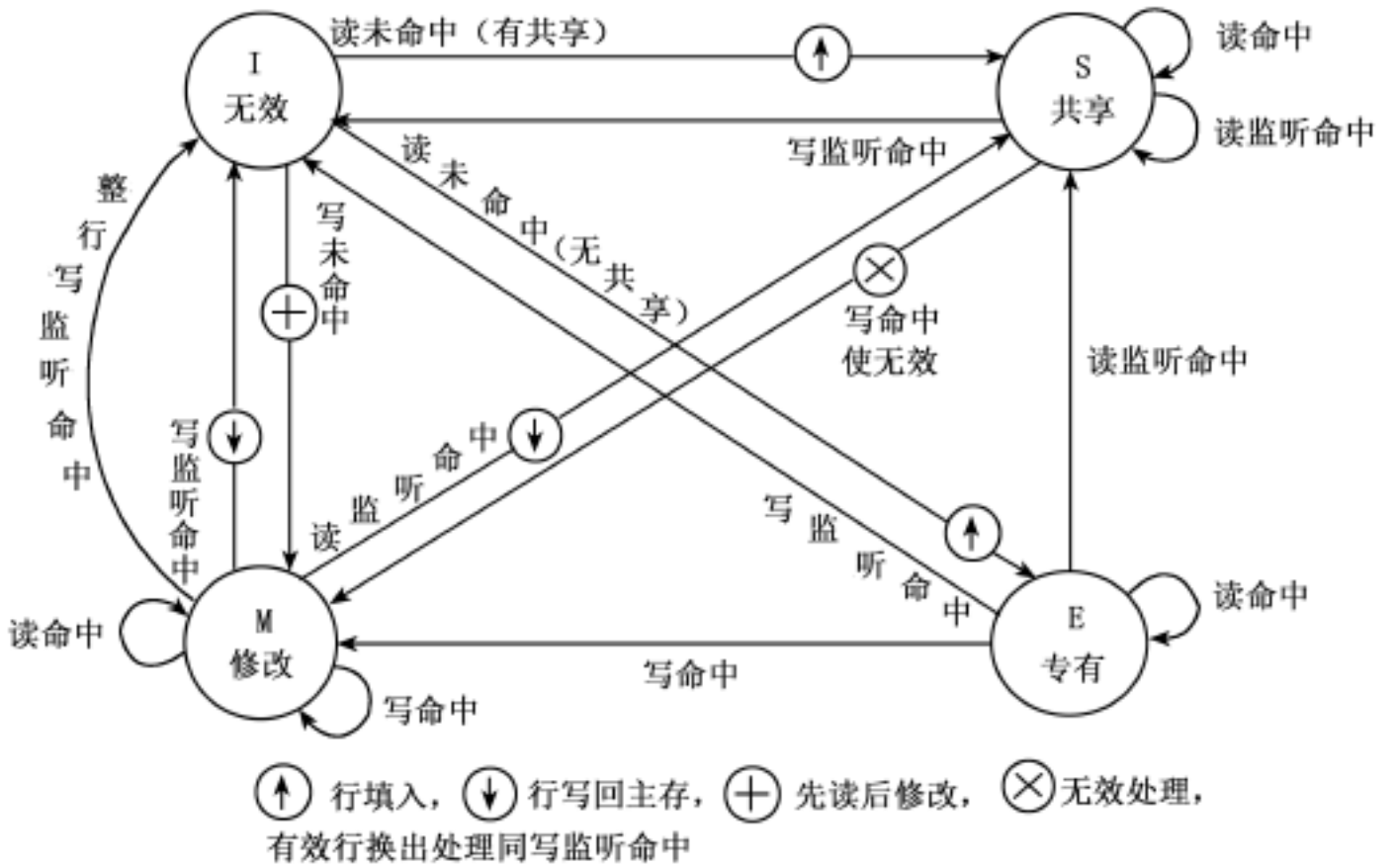


图 4.50　MESI 协议状态转换规则

② S 状态行的读命中或读监听命中都不改变其状态,只是读监听命中时要发出监听命中指示,以使其他 Cache 在读未命中时填入的新行建状态为 S。写监听命中时要将该行作废(状态变为 I)。CPU 对 S 态行的写命中,Cache 在判测的同时将此写请求通知到总线上,以使其他 Cache 同此共享行发生写监听命中而作废,这称为写无效处理,然后 CPU 对此 S 态行完成 Cache 内部写修改并进入 M 态。

③ CPU 对其 Cache E 态行的读命中不改变其状态,写命中也只是在 Cache 内部完成写修改后进入 M 态,无需它者监听。E 态行的读监听命中,表示总线上别的 Cache 正在读同地址的主存块以建立新行,此时除它的状态要改为 S 外,还要发出读监听命中指示,以使别的 Cache 也将填入新行建 S 态。E 态行的写监听命中,表示别的 Cache 由于写未命中而访问同地址的主存块,此 E 态行的内容将是过时的,故将其作废即可。

④ CPU 对其 Cache M 态行的读/写命中,在完成相应的读/写修改操作后不改变其状态,也是无信号送到总线上无需他者监听。M 态行的写监听命中处理已在①中介绍了,它最后进入 I 态被作废。注意,在收到 RWITM 信号后,有此 M 态行的 Cache 要抢占总线先行将 M 态作废。注意,在收到 RWITM 信号后,有此 M 态行的 Cache 要抢占总线先行将 M 态行的内容写回主存,因为它含有此主存块修改过的信息而且那个“先读后写”的修改字不一定能覆盖此 M 态行的全部修改过的位置。当然,若原是写整个新行(猝发式写)就没有把此 M 态行抢先写回主存的必要了。M

态行的读监听命中,也要抢先将其行内容写回主存,以使发生读未命中的 Cache 填入的新行是最新的而不是过时的。此后,它变为 S 态同时也发出一个读监听命中指示,以使他者 Cache 将填入的新行建状态为 S。

由上述分析可以看出,虽然各 Cache 控制器随时都在监听系统总线,但能监听到的只有读未命中、写未命中以及共享行写命中三种情况。读监听命中的有效行都要进入 S 态并发出监听命中指示,但 M 态行要抢先写回主存;写监听命中的有效行都要进入 I 态,但收到 RWITM 时的 M 态行要抢先写回主存。总之监控逻辑并不复杂,增添的系统总线传输开销也不大,但 MESI 协议却有力地保证了主存块脏拷贝在多 Cache 中的惟一性,并能及时写回保证 Cache-主存存取的正确性。

4.3.5 Cache 性能分析

1. Cache 系统的加速比

假设 Cache 的访问时间为 t_c,主存储器的访问时间为 t_m,Cache 存储系统的等效访问时间为 t_a,Cache 命中率为 H_c,那么 Cache 存储系统的等效访问时间为:

$$t_a = t_c \cdot H_c + (1 - H_c)\, t_m$$

则 Cache 系统的加速比 S_p 可定义为:

$$S_p = \frac{t_m}{t_a} = \frac{t_m}{t_c \cdot H_c + (1 - H_c)\, t_m} = \frac{1}{(1 - H) + H \cdot \frac{t_c}{t_m}}$$

由此可以看出,Cache 系统的加速比 S_p 是命中率 H 和主存访问时间与 Cache 访问时间比值的函数。在 Cache 系统中,主存储器和 Cache 的访问时间由于受器件条件的限制,一般为一定值。只有提高 Cache 系统的命中率 H_c(即降低失效率),才是提高加速比 S_p 的最好途径。

2. 降低 Cache 的失效率

有人对 1989～1995 年发表的论文进行过检索,对其中有关 Cache 主题 1 600 多篇文章进行综述后,提出了一个有关 Cache 优化和改进 Cache 技术的基本性能公式。

平均存储器访问时间 = 命中时间 + 失效率×失效开销

按照产生失效的原因不同,可以将失效分为三类(简称为“3C”):

- 强制性失效(Compulsory miss):当第一次访问一个块时,它不在 Cache 中,需从下一级存储器中调入 Cache,这就是强制性失效。也称为冷启动失效或首次访问失效。
- 容量失效(Capacity miss):如果程序在执行时所需的块不能全部调入 Cache 中,则当某些块被替换后,若又重新被访问,这种失效称为容量失效。
- 冲突失效(Conflict miss):在组相联或直接映像的 Cache 中,若太多的块映像

到同一组(块)中,则会出现该组中某个块被别的块替换,然后又被重新访问的情况,这就是冲突失效,也称为碰撞失效(Conllision)或干扰失效(Interference)。

下面简要介绍减少失效率的几种方法:

(1)增加 Cache 块大小

降低 Cache 失效率最简单的方法是增加块大小。在图 4.51 中给出了一组不同容量的 Cache 失效率的关系曲线图,表 4.8 中给出了具体数据,从中可以看出:

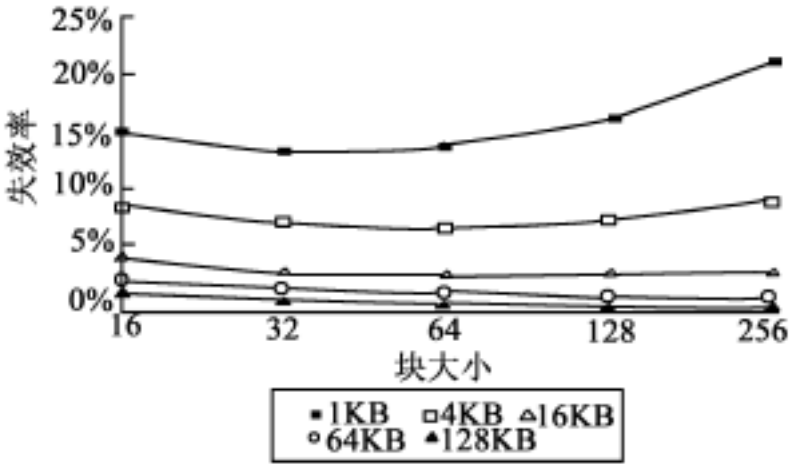


图 4.51 失效率随块大小变化的曲线

表 4.8 各种块大小情况下 Cache 的失效率

块大小	Cache 容量				
	1KB	4KB	16KB	64KB	128KB
16B	15.05%	8.57%	3.94%	2.04%	1.09%
32B	13.34%	7.24%	2.87%	1.35%	0.70%
64B	13.76%	7.00%	2.64%	1.06%	0.51%
128B	16.64%	7.78%	2.77%	1.02%	0.49%
256B	22.01%	9.51%	3.29%	1.15%	0.49%

① 对于给定的 Cache 容量,当块大小增加时,失效率开始是下降的,后来反而上升了。

② Cache 容量越大,使失效率达到最低的块大小就越大。例如在表 4.8 中块大小分别为 1KB,4KB,16KB,64KB 和 128KB 时,使失效率达到最低的块大小分别为 32B,64B,64B,128B 和 128B(或 256B)。

导致上述失效率先下降后上升的原因,在于增加块大小会产生双重作用。一方面它减少了强制性失效,因为程序的局部性原理包括时间上和空间上的局部性,增加块大小利用了空间局部性;另一方面,由于增加块大小会减少 Cache 中块的数目,所以也可能会增加冲突失效,在 Cache 容量较小时甚至会增加容量失效。刚开始增加

块大小时,由于块大小还不是很大,上述的第一种作用超过了第二种作用,从而使失效率下降,但等到块大小增加到较大时,第二种作用超过了第一种,使失效率反而下降。

(2)提高相联度

提高相联度可使失效率下降,这一点可从表 4.9 中看出。表 4.9 中所列数据是在 DEC station 5 000 上使用 SPEC92 测得的,并假设 Cache 块大小为 32 字节,采用 LRU 替换算法。

表 4.9 在不同容量不同相联度情况下,Cache 总失效率及"3C"所占的比例

Cache 块大小	相联度	总失效率	失效率组成(相对百分比)					
			强制性失效		容量失效		冲突失效	
1KB	1-路	0.133	0.002	1%	0.08	60%	0.052	39%
	2-路	0.105	0.002	2%	0.08	76%	0.023	22%
	4-路	0.095	0.002	2%	0.08	84%	0.013	14%
	8-路	0.087	0.002	2%	0.08	92%	0.005	6%
2KB	1-路	0.098	0.002	2%	0.044	45%	0.052	53%
	2-路	0.076	0.002	2%	0.044	58%	0.030	39%
	4-路	0.064	0.002	3%	0.044	69%	0.018	28%
	8-路	0.054	0.002	4%	0.044	82%	0.008	14%
4KB	1-路	0.072	0.002	3%	0.031	43%	0.039	54%
	2-路	0.057	0.002	3%	0.031	55%	0.024	42%
	4-路	0.049	0.002	4%	0.031	64%	0.016	32%
	8-路	0.039	0.002	5%	0.031	80%	0.006	15%
8KB	1-路	0.046	0.002	4%	0.023	51%	0.021	45%
	2-路	0.038	0.002	5%	0.023	61%	0.013	34%
	4-路	0.035	0.002	5%	0.023	66%	0.010	28%
	8-路	0.029	0.002	6%	0.023	79%	0.004	15%
16KB	1-路	0.029	0.002	7%	0.015	52%	0.012	42%
	2-路	0.022	0.002	9%	0.015	68%	0.005	23%
	4-路	0.020	0.002	10%	0.015	74%	0.003	17%
	8-路	0.018	0.002	10%	0.015	80%	0.002	9%
32KB	1-路	0.020	0.002	10%	0.010	52%	0.008	38%
	2-路	0.014	0.002	14%	0.010	74%	0.002	12%
	4-路	0.013	0.002	15%	0.010	79%	0.001	6%
	8-路	0.013	0.002	15%	0.010	81%	0.001	4%

续表

Cache 块大小	相联度	总失效率	失效率组成(相对百分比)					
			强制性失效		容量失效		冲突失效	
64KB	1-路	0.014	0.002	14%	0.007	50%	0.005	36%
	2-路	0.010	0.002	20%	0.007	70%	0.001	10%
	4-路	0.009	0.002	21%	0.007	75%	0.000	3%
	8-路	0.009	0.002	22%	0.007	78%	0.000	0%
128KB	1-路	0.010	0.002	20%	0.004	40%	0.004	40%
	2-路	0.007	0.002	29%	0.004	58%	0.001	14%
	4-路	0.006	0.002	31%	0.004	61%	0.000	8%
	8-路	0.006	0.002	31%	0.004	62%	0.000	7%

从表中我们可以得出两条经验规则。第一,对于表中所列出的 Cache 容量,从实际应用的角度来看,8 路组相联在降低失效率方面的作用已经和全相联一样有效。也就是说,采用相联度超过 8 的方法实际意义不大。第二,2∶1 Cache 经验规则,它是指容量为 N 的直接映像 Cache 的失效率和容量为 N/2 的两路组相联 Cache 的失效率差不多相同。

许多例子都说明,改进平均访存时间的某一方面是以损失另一方面为代价的。增加块大小的方法会在降低失效率的同时增加失效开销,而提高相联度则是以增加命中时间为代价的。Hill 曾发现,当分别采用直接映像和两路组相联时,对于 TTL 或 ECL 板级 Cache,命中时间相差 10%;而对于定制的 CMOS Cache,命中时间相差 2%。所以,为了实现很高的处理器时钟频率,就需要设计结构简单的 Cache;但时钟频率越高,失效开销就越大(所需的时钟周期数就越多)。为了减少失效开销,又要求提高相联度。

(3)牺牲者 Cache

利用增加 Cache 块大小和提高相联度来降低失效率,这是被系统结构设计者已经采用了的经典方法。

所谓牺牲者 Cache 是指在 Cache 和它的下一级存储器的数据通道之间增设一个全相联的小 Cache,称为牺牲者 Cache,如图 4.52 所示。

牺牲者 Cache 中存放由于失效而被丢弃(替换)的那些块(牺牲者)。每当发生失效时,在访问下一级存储器之前,先检查牺牲者 Cache 中是否有所需的块。如果有就将该块与 Cache 中的某个块作交换。Jouppi 于 1990 年发现,含 1～5 项牺牲者 Cache 对减少冲突失效很有效,尤其是对于那些小型的直接映像 Cache 更是如此。对于不同的程序,一个项数为 4 的牺牲者 Cache 能使一个 4KB 直接映像数据 Cache 的冲突失效减少 20%～90%。

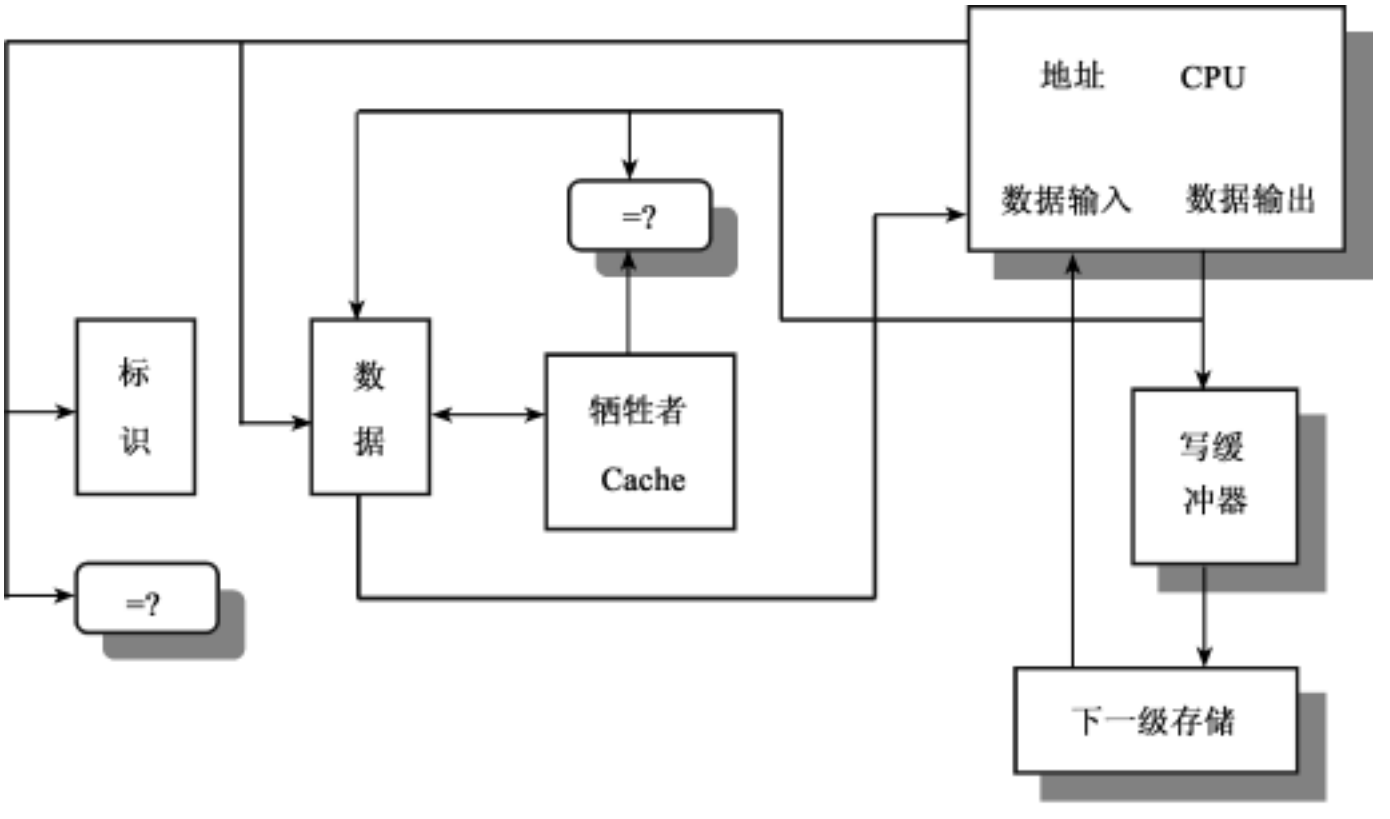


图 4 .52　牺牲者 Cache 在存储层次中的位置

(4)伪相联 Cache

伪相联又称为列相联,这种方法既能获得多路组相联 Cache 的低失效率,又能保持直接映像 Cache 的命中速度。在命中情况下,伪相联 Cache 访问过程与直接映像 Cache 访问过程相同。在失效时,也即在访问下一级存储器之前,通过检查另外一个 Cache 存储块,看看是否在那里命中。一个简单的确定方法是对索引域中的最高位求反,然后按照新的索引去寻找“伪相联组”中的对应块,如果这一块标识匹配,则发生了“伪命中”。否则,只好访问下一级存储器。

由以上分析可知,伪相联 Cache 有两种命中时间,分别对应于正常命中和伪命中的情况。图 4 .53 给出了它们的对应关系。使用伪相联 Cache 时存在一种危险:如果直接映像 Cache 里的许多快速命中在伪相联 Cache 中变成慢速命中,那么这种优化措施反而会降低整体性能。因此,要能够指出同一组中的两个块哪个为快速命中,哪个为慢速命中,这是很重要的。一种简单的方法就是交换两个块的内容。

(5)指令和数据的预取技术

牺牲者 Cache 和伪相联 Cache 都保证在失效率性能的同时,不影响处理器的时钟频率。预取技术也能实现这一点。指令和数据都可在处理器提出访问请求之前进行预取。预取内容可以直接放入 Cache,也可以放在一个访问速度比主存快的外部缓冲器中。

指令预取通常是在 Cache 之外的硬件中完成的。例如,AXP 21064 微处理器在发生指令失效的时候取两个块:被请求的指令块和与其地址相邻的下一块。被请求

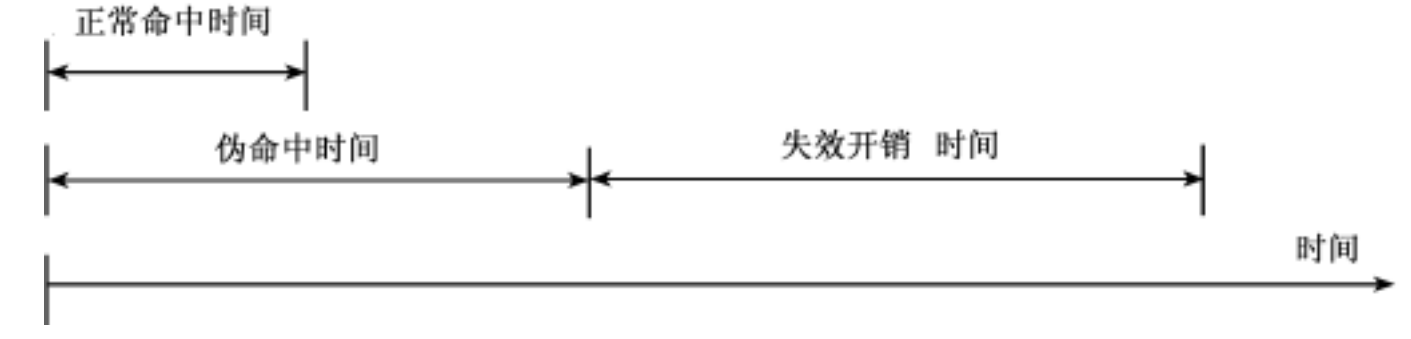


图 4.53 正常命中时间、伪命中时间和失效开销时间之间的关系

的指令块装入到指令 Cache 中,而预取的指令块被装入到指令流缓冲区中。如果被请求的块在指令流缓冲区中找到,则原 Cache 请求被取消,块被从指令流缓冲区中读入,然后下一个预取请求被发出。在 AXP 21064 指令流缓冲区中每个块的大小不超过 32 字节。Jouppi 在 1990 年研究发现,对于容量为 4KB、块大小为 16 字节的直接映像指令 Cache 来说,单个指令流缓冲区可以捕获到 15%～25%的失效。在指令流缓冲区中保存 4 个块,命中率提高大约 50%,保存 16 个块提高 72%。

相似的方案可以应用到数据访问中。Jouppi 发现单个数据流缓冲区,可以捕获到容量为 4KB 直接映像 Cache 的大约 25%的失效。在数据 Cache 外可以有多个流缓冲区,每一个都可以按不同的地址预取。Jouppi 发现 4 个数据流缓冲区可以使数据命中率提高到 43%。

另外,预取技术依赖于存储器带宽的利用。

(6)编译控制的预取技术

一个替代硬件预取指令和数据的技术是利用编译器来插入预取指令,提前发出数据请求。它可以有几种预取的方法:

- 寄存器预取(Register Prefetch):把数据预取到寄存器中。
- Cache 预取(Cache Prefetch):只将数据预取到 Cache 中,而不是寄存器中。

这两种预取技术既可以是故障性的(Faulting),也可以是非故障性的(Nonfaulting)。故障性预取是指在预取时,若出现虚地址故障或违反保护权限,就会发生异常。而非故障性预取在遇到这种情况时则不发生异常。按照这种说法,一条正常的 Load 指令应该被认为是故障性寄存器预取指令。非故障性预取如导致异常就转变为空操作。最有效的预取对程序是“语义上是不可见的”:它既不会改变指令各数据之间的各种逻辑关系或存储单元的内容,也不会造成虚拟存储器故障。本节假定 Cache 预取都是非故障性的,也叫做非绑定(Nonbinding)预取。

只有在预取数据的同时处理器还能继续执行的情况下,预取才是有意义的。这就要求 Cache 在等待预取数据返回的同时还能继续提供指令和数据。这种灵活的 Cache 称为非阻塞(Nonblocking)Cache 或非锁定(Lockup-Free)Cache。

和硬件控制的预取一样,编译器控制预取的目的也是要执行指令和读取数据能

重叠执行。循环是预取优化的主要目标,因为它们易于进行预取优化。如果失效开销较小,编译器只要简单地将循环体展开一次或两次,并调度好预取和执行的重叠。如果失效开销较大,编译器就将循环体展开多次,以便为后面较远的循环预取数据。然而,发出预取指令需要花费一条指令的开销,因此,要注意保证这种开销不超过预取所带来的收益。编译器可以通过把重点放在那些可能会导致失效的访问,使程序避免不必要的预取,从而较大程度地改善平均访存时间。

(7)编译优化技术

这种方法就是通过对软件的优化来降低失效率。这也许是硬件设计者最喜欢的解决方案。处理器和主存之间越来越大的性能差距促使编译器的设计者们更仔细地研究存储层次行为,以期能通过编译时的优化来改进性能。这项研究同样也分为减少指令失效和减少数据失效两个方面。

我们能很容易地重新组织程序而不影响程序的正确性。例如,把一个程序中的几个过程重新排序,就可能减少冲突失效,从而降低指令失效率。McFarling(1989)研究了如何使用记录信息来判断指令组之间可能发生的冲突,并将指令重新排序以减少失效。他发现,这样可将容量为2KB、块大小为4字节的直接映像指令Cache的失效率降低50%;对容量为8KB的Cache,可将失效率降低75%。他还发现,当能够使基本指令根本就不进入Cache时,可以得到最佳性能。但即使不这么做,优化后的程序在直接映像Cache中的失效率也低于未优化程序在同样大小的8路组相联Cache中的失效率。数据对存储位置的限制比指令对存储位置的限制还要少,因此更便于调整顺序。我们对数据进行变换的目的是改善数据的空间局部性和时间局部性。例如,可能把对数据的运算改为对存放在同一Cache块中的所有数据进行操作,而不是按照程序员原来随意书写的顺序访问数组元素。

① 合并数组(Merging Arrays)

合并数组主要是通过提高空间局部性来减少失效次数。一些程序按相同的下标,对多个维数相同的数组同时进行访问。这些访问会彼此影响,从而导致冲突失效。这种影响可以通过把这些独立的矩阵合并成一个复合数组来消除,使一个Cache块可以包含所需要的所有元素。

```
/ * 修改前 */
int val[SIZE];
int key[SIZE];
/ * 合并后 */
struct merge{
int val;
int key;
};
struct merge merged_[SIZE];
```

这个例子一个有趣的特性是,使用记录数组的正确编程策略,可以获得与这种优化策略相同的收益。

② 循环交换(Loop Interchange)

有些程序带有嵌套循环,它们访问存储器中的数据是非顺序性的。简单的交换嵌套循环可以使得代码按照存储顺序来访问数据。这个技术通过空间局部性来减少失效,通过代码的重新排序使得块在被替换之前能够最大限度地利用 Cache 块中的数据。

```
/ * 修改前 */
for (j = 0;j < 100;j = j + 1)
    for (i = 0;i < 5000;i = i + 1)
        x[i][j] = 2 * x[i][j];
/ * 修改后 */
for (i = 0;i < 5000;i = i + 1)
    for (j = 0;j < 100;j = j + 1)
        x[i][j] = 2 * x[i][j];
```

原先的代码以 100 个字的跨距访问存储器,而修改后的程序在访问了一个 Cache 块中的所有字后才去访问下一个 Cache 块。这种优化策略是在不影响指令数的前提下提高了 Cache 的性能。

③ 循环融合(Loop Fusion)

有些程序有分立的代码段,这些代码按照相同的循环访问相同的数组,对相同的数据进行不同的计算。通过“融合”这些代码到一个循环中,使得装入到 Cache 中的数据在被替换之前可以被重复利用。因此,同前两种技术相比较,这个优化策略的目标是通过提高时间局部性来减少失效。

```
/ * 修改前 */
for (i = 0;i < N;i = i + 1)
    for (j = 0;j < N;j = j + 1)
        a[i][j] = 1/ b[i][j] * c[i][j];
for (i = 0;i < N;i = i + 1)
    for (j = 0;j < N;j = j + 1)
        d[i][j] = a[i][j] + c[i][j];
/ * 修改后 */
for (i = 0;i < N;i = i + 1)
    for (j = 0;j < N;j = j + 1)
    {
        a[i][j] = 1/ b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
```

```
}
```

原先的代码在访问数组 a 和 c 时将发生两次失效,一次在第一个循环中,另一次是在第二个循环中。在融合的循环中,第二个语句很自然地利用了第一个语句对 Cache 的访问结果。

④ 分块

这种优化措施可能是在各种 Cache 优化策略中最著名的,它同样是通过提高时间局部性来减少失效。我们又一次要处理多个数组,有些是按行访问的,有些是按列访问的。若以行为主或以列为主存放数组并不能解决问题,因为在每一次循环中既有按行访问也有按列访问的。这种正交关系的访问意味着以前的变换(如循环交换)都是没有用的。

分块算法并不是对数组中的整行或整列进行操作,而是对子矩阵或矩阵块进行操作。目的是在被调入到 Cache 中的块被替换之前最大限度地利用它。下面是矩阵的代码例子,有助于对这种优化的理解。

```
/ * 修改前 */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        {r = 0;
        for (k = 0; k < N; k = k + 1){
           r = r + y[i][k] * z[k][j]; };
        x[i][j] = r;
};
```

两个内层循环读取了矩阵 z 的所有 N×N 个元素,并对矩阵 y 一行中的 N 个元素进行重复地访问,然后对矩阵 x 一行中的 N 个元素进行写操作。图 4 .54 给出了一个访问 3 个数组的访问情况,深色阴影部分表示一次最近的访问,浅色阴影表示一次较早的访问,白色表示还没有被访问到。

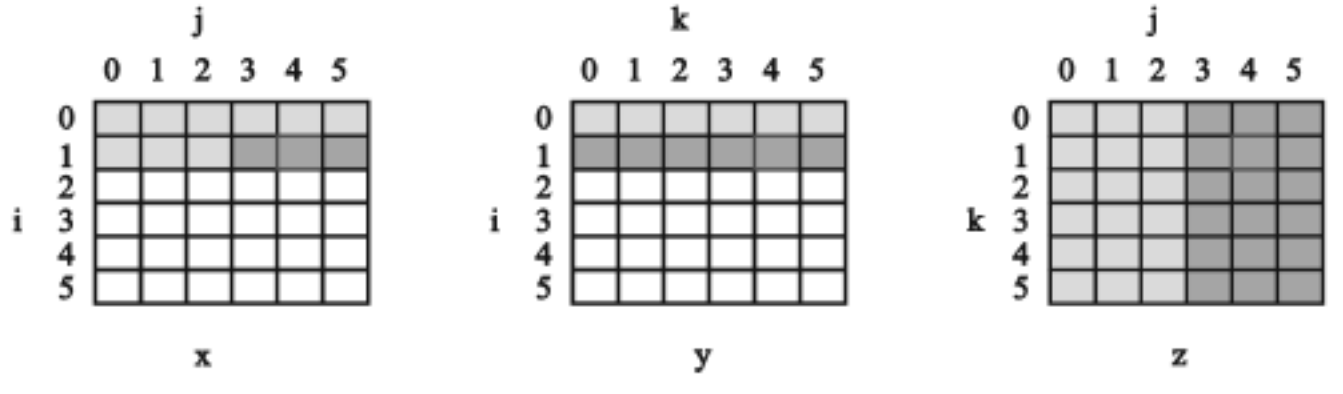


图 4 .54 当 i = 1 时,对 x,y,z 三个数组的访问情况

容量失效数显然依赖于 N 和 Cache 的容量。如果它能够包含所有 3 个 N×N

矩阵,那么情况还好,但前提是不发生冲突失效。如果 Cache 能包含一个 N×N 矩阵以及一行的 N 个元素,那么至少 y 的第 i 行以及数组 z 可以保留在 Cache 中。若 Cache 的容量比这还要小,则对于 x 和 z 都会有失效发生。在最坏的情况下,N^3 次操作会有 $2N^3 + N^2$ 次失效。

为了保证要访问的元素都能在 Cache 中命中,要对源代码进行改变,两个内部循环按步长 B 计算,而不是从 x 到 z 的开始一直到结束来计算,从而使其在一个 B×B 的子矩阵上计算。B 称为分块因子(Blocking Factor)。

```
/ * 修改后 */
for (jj = 0;jj < N;jj = jj + B)
for (kk = 0;kk < N;kk = kk + B)
for (i = 0;i < N;i = i + 1)
    for (j = jj;j < min(jj + B - 1,N);k = k + 1){
        r = 0;
         for (k = kk;k < min(kk + B - 1,N);k = k + 1){
             r = r + y[i][k] * z[k][j];
             }
        x[i][j] = x[i][j] + r;
    };
```

图 4.55 显示了用分块的方法来访问 3 个数组的情况。仅仅观察容量失效,总共访问的存储器字符为 $2N^3/B + N^2$,大约提高了一个因子 B。因此,分块充分利用了访问的空间局部性和时间局部性。y 从空间局部性中受益,而 z 从时间局部性中受益。

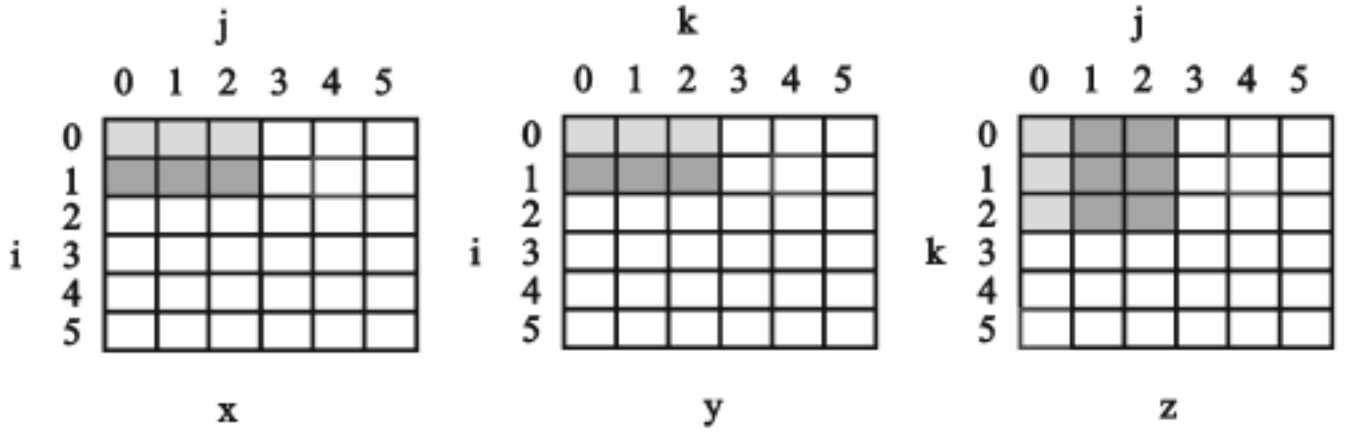


图 4.55　访问数组 x,y,z 的时间

虽然我们的目标是降低 Cache 失效率,但是分块也有助于寄存器分配。通过采用一个小的分块使得块能够被装入到寄存器中,可以使程序中取和存操作的数量最小化。

传统上,若简单地假定冲突失效很小,或者能够因 Cache 相联度高而消除,则分

块的目的是减少容量失效。因为分块减少了给定时间内在 Cache 中活动的字数,而选择比容量小的分块也可以减少冲突失效。

3. 减少 Cache 的失效开销

Cache 性能公式告诉我们,减少 Cache 失效开销与降低 Cache 失效率一样,能够带来 Cache 性能的提高。在所有的方法中,采用两级 Cache 技术最受欢迎。

CPU 和主存之间的性能差距给体系结构设计者们提出了以下问题:为了克服这个越来越大的性能差距,使存储器和 CPU 的性能匹配,是应该把 Cache 做得更快,还是应该把 Cache 做得更大?答案是:二者兼顾。通过在原有 Cache 和存储器之间增加另一级 Cache,构成两级 Cache。可以把第一级 Cache 做得足够小,使其速度和快速 CPU 的时钟周期相匹配,而把第二级 Cache 做得足够大,使它能捕获更多本来需要到主存去的访问,从而降低实际失效开销。

尽管增加一级存储层次在概念上直观、简单,但性能分析却变得复杂了。有关第二级 Cache 的定义不太好理解。用下标 L1 和 L2 分别表示第一级和第二级 Cache,原有的公式就变为:

$$\text{平均访存时间} = \text{命中时间}_{L1} + \text{失效率}_{L1} \times \text{失效开销}_{L1}$$

$$\text{失效开销}_{L1} = \text{命中时间}_{L2} + \text{失效率}_{L2} \times \text{失效开销}_{L2}$$

所以,

$$\text{平均访存时间} = \text{命中时间}_{L1} + \text{失效率}_{L1} \times (\text{命中时间}_{L2} + \text{失效率}_{L2} \times \text{失效开销}_{L2})$$

在这个公式里,第二级 Cache 的失效率是以在第一级 Cache 中不命中而到达第二级 Cache 的访存次数为分母来计算的。为避免二义性,对于第二级 Cache 系统采用以下术语:

(1)局部失效率

对于某一级 Cache 来说,局部失效率 = 该级 Cache 的失效次数/ 到达该级 Cache 的访存次数。对于第二级 Cache 来说,就是上面的失效率_{L2}。

(2)全局失效率

对于某一级 Cache 来说,全局失效率 = 该级 Cache 的失效次数/ CPU 发出的访存总次数。使用上面公式中的变量,第二级 Cache 的全局失效率就是:

$$\text{全局失效率}_{L2} = \text{失效率}_{L1} \times \text{失效率}_{L2}$$

因为访存都要经过第一级 Cache,所以它的局部失效率比较大。对于两级 Cache,全局失效率是一种比局部失效率更有用的衡量指标,它指出了在 CPU 发出的访存中,究竟有多大比例是穿过各级 Cache 最终到达存储器的。

例 4.4　假设在 1 000 次访存中,第一级 Cache 失效 40 次,第二级 Cache 失效 20 次。试问:在这种情况下,该 Cache 系统的局部失效率和全局失效率各是多少?

解:第一级 Cache 的失效率(全局和局部)是 40/ 1 000,即 4%;第二级 Cache 的局部失效率是 20/ 40,即 50%,第二级 Cache 的全局失效率是 20/ 1 000,即 2%。

请注意,上述公式是针对读写操作混合而言的,而且假设第一级 Cache 采用写回法。当采用写直达法时,第一级 Cache 将不仅把失效,而且还把所有的写访问都送往第二级 Cache,另外还会使用一个写缓冲器。

对于第二级 Cache,我们有以下结论:

① 在第二级 Cache 比第一级 Cache 大得多的情况下,两级 Cache 的全局失效率和容量与第二级 Cache 相同的单级 Cache 的失效率非常接近,这时可以利用前面关于单级 Cache 的分析和知识。

② 局部失效率不是衡量第二级 Cache 的一个好指标,因为它是第一级 Cache 失效率的函数,可以通过改变第一级 Cache 而使之变化,而且不能全面反映两级 Cache 体系的性能。因此,在评价第二级 Cache 时,应用全局失效率这个指标。

下面考虑第二级 Cache 的参数。第一级 Cache 和第二级 Cache 之间首先要区别的是第一级 Cache 的速度会影响 CPU 的时钟频率,而第二级 Cache 的速度只影响第一级 Cache 的失效开销。因此,对于第二级 Cache,在设计时可以有更多的考虑空间,许多不适合于第一级 Cache 的方案对于第二级 Cache 却可以使用。第二级 Cache 的设计只有两个问题需要权衡:它能否降低 CPI 中的平均访存时间部分?它的成本是多少?

首先,研究第二级 Cache 的容量。因为第一级 Cache 中的所有信息都会出现在第二级 Cache 中,第二级 Cache 的容量应比第一级大许多。如果第二级 Cache 只是稍大一点,局部失效率将很高。因此,第二级 Cache 的容量一般很大,和过去计算机的主存一样大!大容量意味着第二级 Cache 可能实际上没有容量失效,只剩下一些强制性失效和冲突失效。现在的问题是:相联度(组相联中的路数 n)对于第二级 Cache 的作用是否会更大?

例 4.5　给出有关第二级 Cache 的以下数据:

① 两路组相联使命中时间增加 10%×CPU 时钟周期;

② 对于直接映像,命中时间$_{L2}$ = 10 个时钟周期;

③ 对于直接映像,局部失效率$_{L2}$ = 25%;

④ 对于两路组相联,局部失效率$_{L2}$ = 20%;

⑤ 失效开销$_{L2}$ = 50 个时钟周期。

试问第二级 Cache 的相联度对失效开销的影响如何?

解:对一个直接映像的第二级 Cache 来说,第一级 Cache 的失效开销为

$$\text{失效开销}_{\text{直接映像,L1}} = 10 + 25\% \times 50 = 22.5 \text{ 个时钟周期}$$

对于两路组相联第二级 Cache 来说,命中时间增加了 10%(0.1)个时钟周期,故第一级 Cache 的失效开销为:

$$\text{失效开销}_{\text{两路组相联,L1}} = 10.1 + 20\% \times 50 = 20.1 \text{ 个时钟周期}$$

在实际机器中,第二级 Cache 几乎总是和第一级 Cache 以及 CPU 同步操作。相应地,第二级 Cache 的命中时间必须是时钟周期的整数倍。如果幸运的话,可以把该

命中时间取整为 10 个时钟周期,否则就只好取整为 11 个时钟周期,但不管怎样,都比直接映像第二级 Cache 好:

$$失效开销_{两路组相联,L1} = 10 + 20\% \times 50 = 20.0 \text{ 个时钟周期}$$

$$失效开销_{两路组相联,L1} = 11 + 20\% \times 50 = 21.0 \text{ 个时钟周期}$$

可以利用减少第二级 Cache 的失效率,从而达到减少失效开销的目的。提高相联度和伪相联方法都值得考虑,因为它们对第二级的命中时间影响很小,而且平均访存时间中很大一部分是由于第二级 Cache 失效而产生的。虽然较大容量的第二级 Cache 通过把数据分布到更多的 Cache 块中消除了一些冲突失效,但它同时也减少了容量失效,所以在直接映像的第二级 Cache 中,冲突失效所占的比例依然很大。

同样可以采用增加第二级 Cache 块大小的方法来减少失效。前面已经得出这样的结论:增加 Cache 块的大小也增加了小容量 Cache 的冲突失效次数(因为可能没有足够的位置来存放数据),导致失效率上升。但对于大容量的第二级 Cache 来说,这一点并不成为问题,而且由于访存时间相对较长,所以 64 字节、128 字节,甚至 256 字节的块大小都是第二级 Cache 经常使用的。图 4.56 给出了在存储器总线宽度为相对较窄的 32 位时,CPU 执行时间随第二级 Cache 块大小的变化而改变的情况。

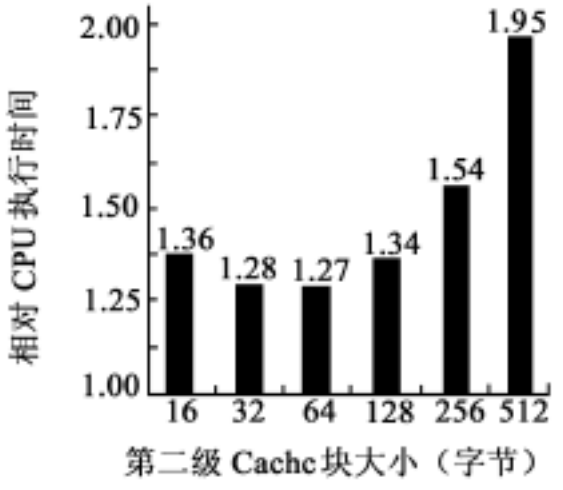


图 4.56 相对执行时间和第二级 Cache 块大小的关系

需要考虑的另一个问题是第一级 Cache 中的数据是否总是同时存在于第二级 Cache 中。如果是的话,则第二级 Cache 具有多级包容性(Multilevel Inclusion Property)。多级包容性是我们所希望的,因为它便于实现 I/O 和 Cache 之间内容一致性的检测。

为了减少平均访存时间,可以让容量较小的第一级 Cache 采用较小的块,而让容量较大的第二级 Cache 采用较大的块。在这种情况下,仍可实现包容性,但在处理第二级 Cache 失效时要做更多的工作:替换第二级 Cache 中的块时,必须作废所有映像到该块的第一级 Cache 中的块。这样不但会使第一级 Cache 失效率有所增加,而且会造成不必要的作废。如果结合使用其他一些性能优化技术(如非阻塞的第二级

Cache),包容性会进一步增加复杂度。

综合上述考虑,Cache 设计的本质是在快速命中和减少失效次数这两个方面进行权衡。大部分优化措施都是在改进一方的同时损害另一方。对于第二级 Cache 而言,由于它的命中次数比第一级 Cache 少得多,所以重点就转移到了减少失效次数上。这就导致了更大容量、更高相联度和块更大的 Cache 的出现。

4. 减少命中时间

到目前为止,我们已经讨论了通过减少失效次数和减少失效开销改进 Cache 性能的办法,现在我们来讨论减少命中时间的技术,它也是平均访存时间的三个组成部分之一。

命中时间的重要性在于它影响到处理器的时钟频率。在当今的许多机器中,往往是 Cache 的访问时间限制了处理器的时钟频率,即使在把访问 Cache 分为几个时钟周期来完成的机器中也是如此。因此减少命中时间不仅对减少平均访存时间很重要,而且对其他许多方面也是非常重要的。本节先讨论两种减少命中时间的通用技术,然后论述一个适用于写命中的优化措施。

(1)容量小、结构简单的 Cache

采用容量小而且结构简单的 Cache,可以有效地提高 Cache 的访问速度。用地址的索引部分访问标识存储器、读出标识并与地址进行比较,是 Cache 命中过程中最耗时的部分。我们在前面已经指出,硬件越简单,速度就越快。小容量 Cache 对减少命中时间当然有益,而且应使 Cache 足够小,以便可以与处理器做在同一芯片上,以避免因芯片外访问而增加时间开销,这一点是非常重要的。

某些设计采用了一种折衷方案:把 Cache 的标识放在片内,而把 Cache 的数据存储器放在片外,这样既可以实现快速标识检测,又能利用独立的存储芯片来提供更大的容量。另一个方案是要保持 Cache 结构简单,例如采用直接映像 Cache。直接映像 Cache 的主要优点是可以让标识检测和数据传送重叠进行,这样可以有效地减少命中时间。所以,为了得到高速的时钟频率,第一级 Cache 应选用容量小且结构简单的设计方案。

(2)虚拟 Cache

在采用虚拟存储器的机器中,每次访存都必须进行虚地址到实地址的变换,即将 CPU 发出的虚地址转换为物理地址。即便是容量小、结构简单的 Cache,也必须解决这个问题。

与 Cache 失效相比,Cache 命中发生的频率要高得多。按照“加快经常性事件”实现的指导思想,应在 Cache 中使用虚拟地址。这样的 Cache 称为虚拟 Cache,而物理 Cache 则是指那些使用物理地址的传统 Cache。

直接用虚拟地址访问 Cache,在命中时消除了用于地址转换的时间。然而,人们在设计时并非都采用虚拟 Cache。其原因之一是每当进行进程切换时,由于新进程

的虚拟地址所指向的物理空间有可能与原进程地址不同,故需要清空 Cache。图 4.57说明了这种清空对失效率的影响。解决这个问题的一种方法是在地址标识中增加一个进程标识符字段(PID),这样多个进程的数据可以混和存放于一个 Cache 中,由 PID 指出 Cache 中各块的数据是属于哪个程序的。为减少 PID 的位数,PID 经常是由操作系统指定。对于一个进程,操作系统从循环使用的几个数字中指定一个作为其 PID。进程切换时,仅当某个 PID 被重用(即该 PID 以前已被分配给了某个进程,现又把它分配给另一个进程)时,才需清空 Cache。

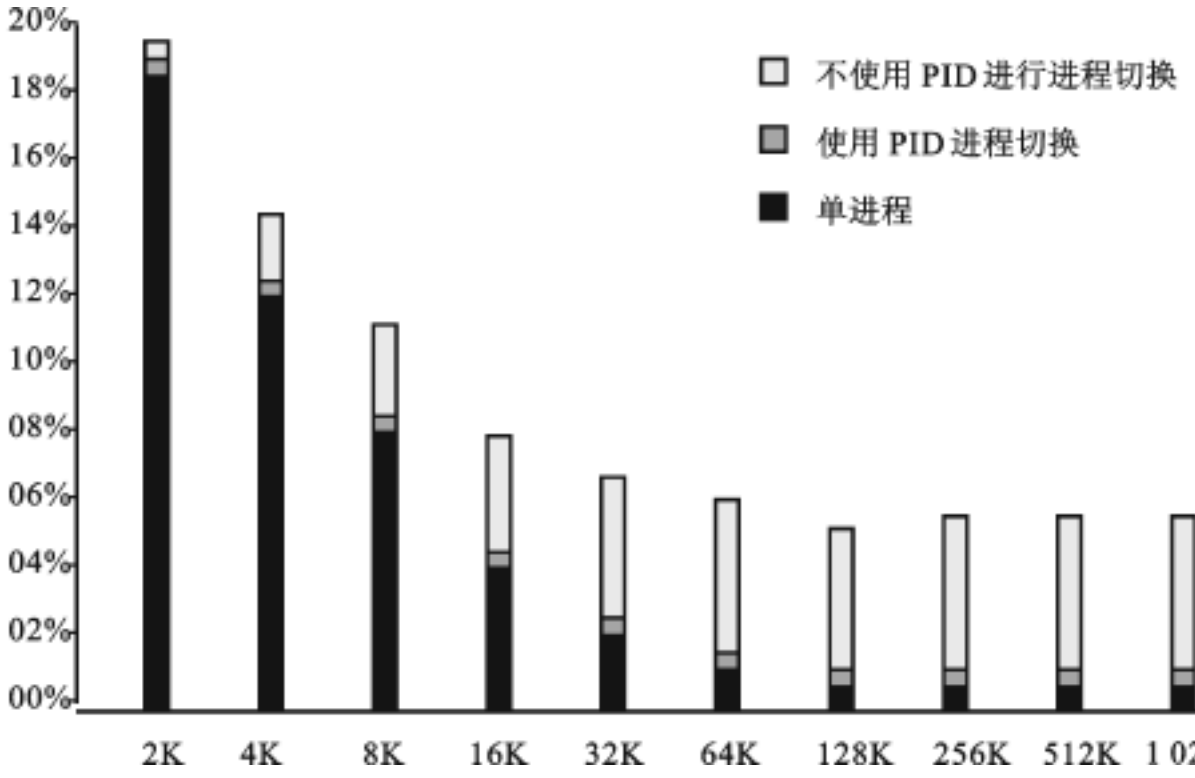


图 4.57 一个程序在三种不同方式下,虚地址 Cache 不同容量的失效率

图 4.57 说明了采用 PID 所带来的失效率上的改进,同时给出了在以下三种情况下各种容量的虚拟 Cache 的失效率:没有进程切换(单进程),允许进程切换并使用进程标识符(PIDs),允许进程切换但不使用进程标识符(purge)。从图中可看出:和单进程相比,PIDs 的绝对失效率增加 0.3%～0.6%;而和 purge 相比,PIDs 的绝对失效率减少 0.6%～4.3%。图中的数据是在 VAX 机上针对 Ultrix 操作系统统计的,并假设 Cache 采用直接映像,块大小为 16 字节。

虚拟 Cache 没有流行起来的一个原因是操作系统和用户程序对于同一个物理地址可能采用两种以上不同形式的虚拟地址来访问,这些地址称为同义(Synonym)或别名(Alias)。它们可能会导致同一个数据在虚拟 Cache 中存在两个副本。如果其中一个被修改,那么再使用另一个数据就是错误的。这种情况在物理 Cache 中是不会发生的,因为这些访问首先会把虚拟地址转换到同一物理地址,从而找到同一个物理 Cache 块。有一种用硬件解决这个问题的方法,叫做反别名法,它保证每一个 Cache 块对应于惟一的一个物理地址。

如果强行要求别名的某些地址位相同,就可以用软件很容易地解决这一问题。例如,SUN 公司的 UNIX 要求所有使用别名的地址最后 18 位都相同,这种限制被称为页着色。这一限制使得容量不超过 2^{18} 字节(256KB)的直接映像 Cache 中不可能出现一个 Cache 块有重复物理地址的情况。所有别名将被映像到同一 Cache 块位置。

对于虚拟地址,最后还应考虑 I/O。I/O 通常使用物理地址,所以为了与虚拟 Cache 打交道,需要把物理地址映像为虚拟地址。另一种实现快速命中的技术是把地址转换和访问 Cache 这两个过程分别安排到流水线的不同级中,从而使时钟周期加快,但这同时也增加了命中所需的时间。这种方法增加了访存的流水线级数,增加了分支预测错误时的开销,而且使得从 Load 指令流出到数据可用之间所需的时钟周期数增加。

还有一种方法,既能得到虚拟 Cache 的好处,又能得到物理 Cache 的优点。它直接用虚地址中的页内位移(页内位移在虚—实地址的变换中保持不变)作为访问 Cache 的索引,但标识却是物理地址。CPU 发出访存请求后,在进行虚—实地址变换的同时,可并行进行标识的读取。在完成地址变换后,再把得到的物理地址与标识进行比较。

这种虚拟索引、物理标识方法的局限性,在于直接映像 Cache 的容量不能超过页的大小。AXP21064 采用了这种方法,其 Cache 容量为 8KB,最小页为 8KB,所以可以直接从虚地址的页内位移部分中得到 8 位的索引(块大小为 32 字节)。

为了既能实现大容量的 Cache,又能使索引位数比较少,以便能直接从虚拟地址的页内位移部分得到,我们可以采用提高相联度的方法,这一点可以从下面的公式中看出:

$$2^{\text{index}} = \frac{\text{Cache 容量}}{\text{块大小} \times \text{相联度}}$$

下面举一个极端的例子——IBM3033 的 Cache。虽然研究结果已表明 8 路以上的组相联对减少失效率没多大好处,但 IBM3033 的 Cache 仍采用了 16 路组相联,其主要好处是可以采用更大的 Cache。尽管 IBM 体系结构限制页的大小为 4KB,但 16 路组相联却可以用物理索引对 64KB(16×4KB)的 Cache 进行寻址。图 4.58 给出了索引和页内位移的关系。页大小为 4 KB 意味着地址的最后 12 位不必进行转换,因此其中某些位可以用做访问 Cache 的索引。

图 4.58 IBM3033 的 Cache 中索引和页内位移的关系

我们也可以不采用提高相联度的方法,而由操作系统实现页着色。操作系统通

过使虚页地址和物理页地址的最后几位相同来实现这一点。采用这种方法时,索引的位数可以比先前的页内位移方法的索引位数多,并且仍然是对物理地址进行比较。

另一种方法是用一小块硬件来猜测虚页地址的后几位映像到什么样的物理地址。这块硬件可以是一个小表格,它对虚页地址进行散列变换。由此得到的猜测结果和地址的物理部分(页内位移)一起构成访问 Cache 的索引。用此索引读出相应的标识,并与经地址转换得到的物理地址进行比较,判断是否匹配。如果标识匹配,则发生了一次命中。如果不匹配,则要么就是数据不在 Cache 中,要么就是关于虚页地址最后几位的映像的猜测是错的。这时 Cache 一般会用正确的索引重新判断这次访存究竟是命中还是真的失效。

上述采用容量小且结构简单的 Cache 以及避免地址转换延迟的技术不仅能提高写命中的速度,而且能提高读命中的速度。

(3)写操作流水化

写命中通常比读命中花费更多的时间,因为在写入数据之前必须先检测标识,否则就有可能将数据写到错误的单元中。我们可以通过把写操作流水化来提高写命中的速度。AlphaAXP21064 和其他一些机器采用了这种技术。图 4 .59 是说明流水化

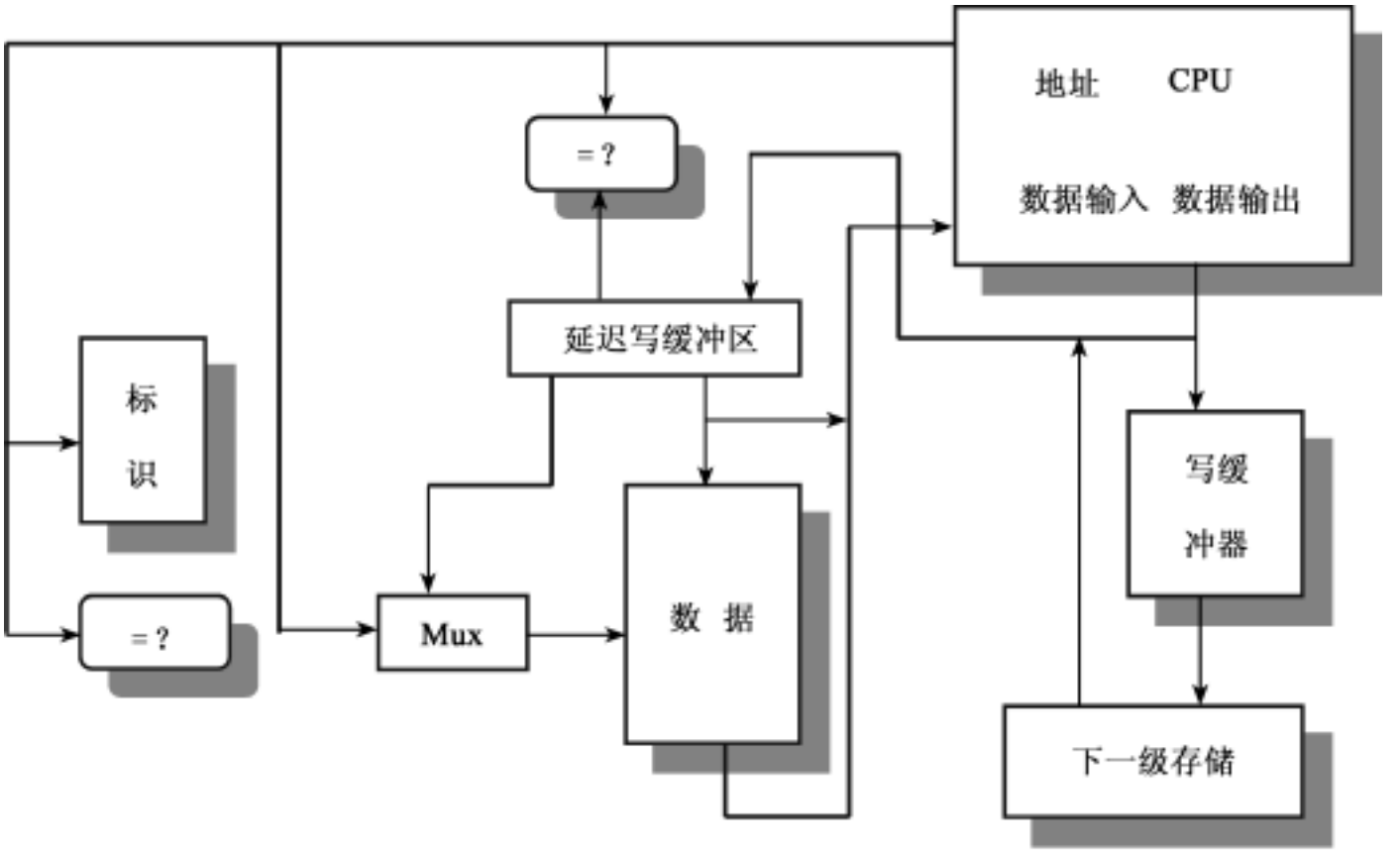


图 4 .59 流水化写的硬件组织结构

写的硬件结构框图。这里,标识和数据是分开存放的,因而能分别独立地访问。每个写操作的工作被分为两个阶段来完成:第一阶段进行标识比较,并把标识和数据存入延迟写缓冲器中;第二阶段再进行数据的写入(若命中的话)。这两个阶段按流水方

式工作。这样,当前“写”的标识比较就可以和上一个“写”的数据写入并行起来,实现每个时钟周期完成一个写操作(从 CPU 的角度)。该流水线与读操作无关,因为读操作的标识比较与数据读出本来就可以并行进行,所以无需专门的硬件支持。

5. Cache 优化技术小结

在前面讨论中论述的减少 Cache 失效率、失效开销和命中时间的技术通常会影响平均访存时间公式的其他组成部分,而且会影响存储层次的复杂性。表 4.10 对这些技术作了小结,并估计了它们对复杂性的影响。表中“+”号表示这一技术改进了相应指标,“-”号表示它使该指标变差,而空格栏则表示它对该指标无影响。从表中可以看出,没有什么技术能同时改进两项或三项指标。表中关于复杂性的衡量是主观化的,0 表示最容易,3 表示最复杂。

表 4.10　　Cache 优化技术小结

优化技术	失效率	失效开销	命中时间	硬件复杂程度	说　　明
增加块容量	+	-		0	容易实现,但意义不大;RS/6000 550 块容量为 128B
增加相联度	+		-	1	MIPS R10000 为 4 路组相联
牺牲者 Cache	+			2	HP7200 采用了类似技术
伪相联 Cache	+			2	在 MIPS R10000 的 L2 中采用
硬件预取	+			2	难以实现,仅在 AlphaAXP21064 少数计算机上使用
编译预取	+			3	需要非阻塞 Cache,部分计算机支持这一技术
编译优化	+			0	向软件提出了新要求,一些计算机通过编译提供选项
两级 Cache		+		2	硬件成本高,两级块大小不相等时实现困难
容量小结构简单的 Cache	-		+	0	容易实现,广泛使用
虚拟 Cache			+	2	Cache 容量小时意义不大,在 AlphaAXP21064 上采用。
写操作流水化			+	1	在 AlphaAXP21064 中采用

4.3.6　Pentium PC 的 Cache

Pentium PC 目前仍是一个单处理器系统,它采用两级 Cache 结构。安装在主板

上的级 2 Cache(L2),其容量是 256KB 或 512KB(PⅡ/ PⅢ 级 2 Cache 容量达到 2MB 或更高),采用的两路组相联映像方式,每行可以是 32,64 或 128 字节。集成在 Pentium 处理器内的级 1 Cache(L1)其容量是 16KB,采用的也是两路组相联映像方式,每行是 32 字节。L2 的内容是主存的子集,L1 又是 L2 的子集,从而使 L1 未命中处理时间大为缩短,为 L1 的高速使用提供了支持。

Pentium PC 的 Cache 另一特点是,它将 16KB 的 L1 分为 8KB 的指令 Cache 和数据 Cache(PⅡ/ PⅢ的 L1 分为 16KB 指令 Cache 和 16KB 数据 Cache)。实践证明将指令 Cache 与数据 Cache 分开是有好处的。指令 Cache 是只读的,单端口 256 位(32 字节)向指令预取缓冲器提供指令代码。数据 Cache 必须提供读和写操作,双端口,每端口 32 位(4 字节),向两条流水线的整数运算单元和寄存器提供数据或接收数据,两个端口还可组合成一个 64 位端口与浮点运算单元相接。两个 Cache 与 64 位数据、32 位地址的 CPU 内部总线相连。

下面分别介绍 Pentium 级 1 Cache 的组织和一致性的实现。因指令 Cache 是只读的,没有写操作也就没有一致性问题。这里只介绍数据 Cache。

1. Pentium 级 1 Cache 的组织结构

8KB 的数据 Cache 采用两路组相联的组织方式,分成 128 组,每组 2 行,每行 32 字节(8 个双字,1 个双字为 32 位)。每行有一个 20 位的标记和 2 位的 MESI 的状态位,这 22 位构成该行的目录项。采用 LRU 替换策略,一组两行共用一个 LRU 位,如规定一组中的 A 行拷贝进新数据将此位置 1,另一 B 行拷贝进新数据将此位置 0。那么当需要替换时只需检查此位状态即可,为 0 换出 A 行,为 1 则换出 B 行,实现了保护新行的原则。这样 L1 数据 Cache 呈现出如图 4 .60 所示的数据结构,只是不再是 2 位状态位而是 1 位有效位。

存取 Cache 使用 32 位物理地址。级 1 指令 Cache 和数据 Cache 每个都有一个变换后援缓冲器 TLB(Translation Lookaside Buffer),用于在虚拟存储方式下将线性地址变换为物理地址。

数据 Cache 可以在一个处理器时钟周期内存取两个数据,数据可以是字节、字(2 字节)和双字(4 字节)。这两个数据分别通过两个 32 位端口被 U,V 两条指令流水的 ALU 单元、寄存器所存取。Pentium 处理器的时钟频率在它刚推出时是 66MHz,现在一般已是 180MHz,200MHz,而新一代的 Pentium 处理器目前已有 300MHz 了。可以想见,若没有片内 Cache 的支持,高速的指令流水必将经常停滞不前,更不用说两路流水的超标量结构了。

Pentium 处理器内数据 Cache 的工作方式受控制寄存器 CR0 中的 Cache 禁止 CD(Cache Disable)位和非写直达 NW(Not-Write-through)位两位控制,如表 4 .11 所示。

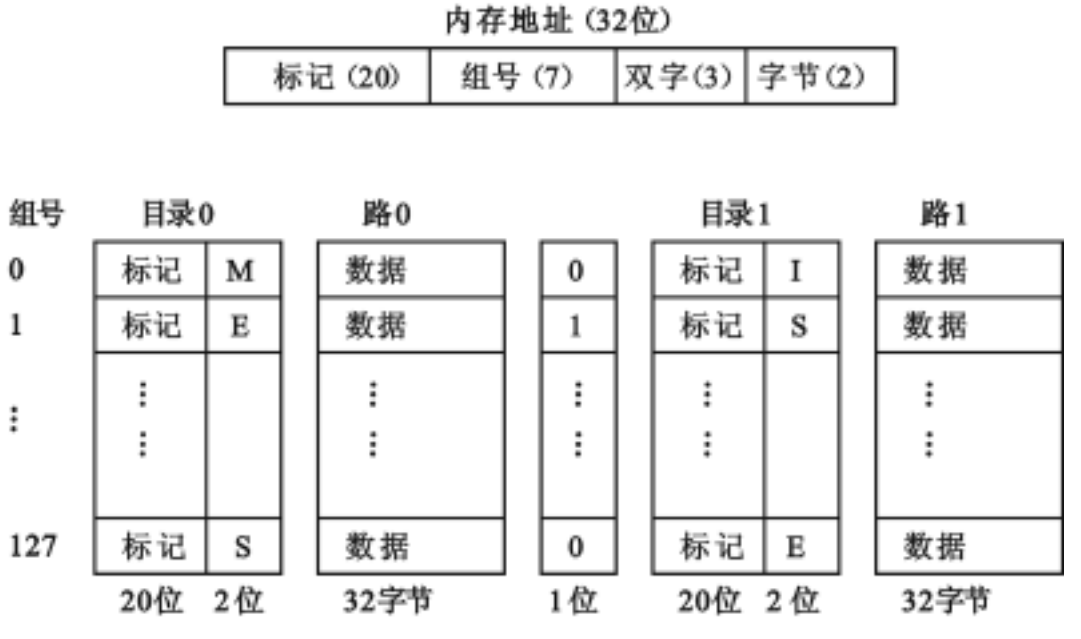


图 4 .60　Pentium 级 1 数据 Cache 结构

表 4 .11 Pentium 片内数据 Cache 工作方式

CD	NW	新行填入	写直达	使无效
0	0(最佳)	允许	允许	允许
1	0	禁止	允许	允许
1	1(复位后)	禁止	禁止	禁止

对表 4. 11 的理解应该注意,Pentium 已对 CR0 控制寄存器中的 CD 和 NW 位重新定义,不同于 80486 中的定义。CD = 1, NW = 1 是复位后的状态,而 CD = 0, NW = 0 已是最佳使用状态。这是一种在写回法的基础上允许某些情况下写直达的写策略,下面将会看到这实际上就是写一次法。

Pentium 处理器外部总线的数据宽度是 64 位。当片内 Cache 行填入或行写回时,启动的是猝发式内存读写周期(CHACE# 引脚为低电平,读时还要求 KEN# 引脚为低电平),一次完成 256 位的串传送,即一次完成整行的填入或读出。

为维护 Cache 的一致性,级 2 Cache 和级 1 数据 Cache 都采用的是 MESI 协议。鉴于级 1 数据 Cache 采用的是写一次法,其 MESI 协议做了一些简化。为支持监听片外内存访问活动,Pentium 为片内数据 Cache 提供了一个监听窗口。在监听期间,Pentium 浮起它的地址输出引脚,而允许片外输入地址,以使片内数据 Cache 能判测是否监听命中。并有如下一些处理器引脚:INV(输入,使无效),WB/ WT# (输入,写回/ 写直达),HIT# (输出,监听命中),HITM# (输出,修改状态行监听命中)等,这些信号或者协调片内 Cache MESI 协议的状态转换,或者输出监听命中指示以使片外 Cache 能判测并完成相应的状态转换。

2. Pentium 级 1 数据 Cache 的 MESI 协议

下面介绍 Pentium 级 1 数据 Cache(L1)的 MESI 协议,以及它如何与级 2 Cache (L2)相配合来维护一致性并能充分支持 Pentium 高速运行的特点。经上节介绍可给出图 4 .61 所示的 L1,L2 和主存之间工作环境的框图。

Pentium CPU 与外部数据交换的存储读写总线周期主要有两大类:一类是前面介绍的 256 位猝发式传送,用于 L1 的行填入和行写出;另一类是不经 L1 的 64 位传送,此时 CHACE# 信号为高电平,在 Intel 数据手册中称此为“不可超高速缓存”式传送。

由于 L1 位于 CPU 内部,它的内容又是 L2 内容的子集,因此 L1 与 L2 的监听对象不同。L2 监听系统总线上其他系统主控者和其他 Cache 的访问主存活动;L1 监听 L2,采样 L2 发出的 WB/ WT# 信号和 INV 信号以完成相应的行状态转换,并有监听命中信号 HIT# 和 HITM# 送至 L2。L2 可发出 AHOLD 信号,命 CPU 的地址引脚改变为输入方向,强制 CPU 的 L1 进入监听期间,Intel 数据手册中称此为询问期

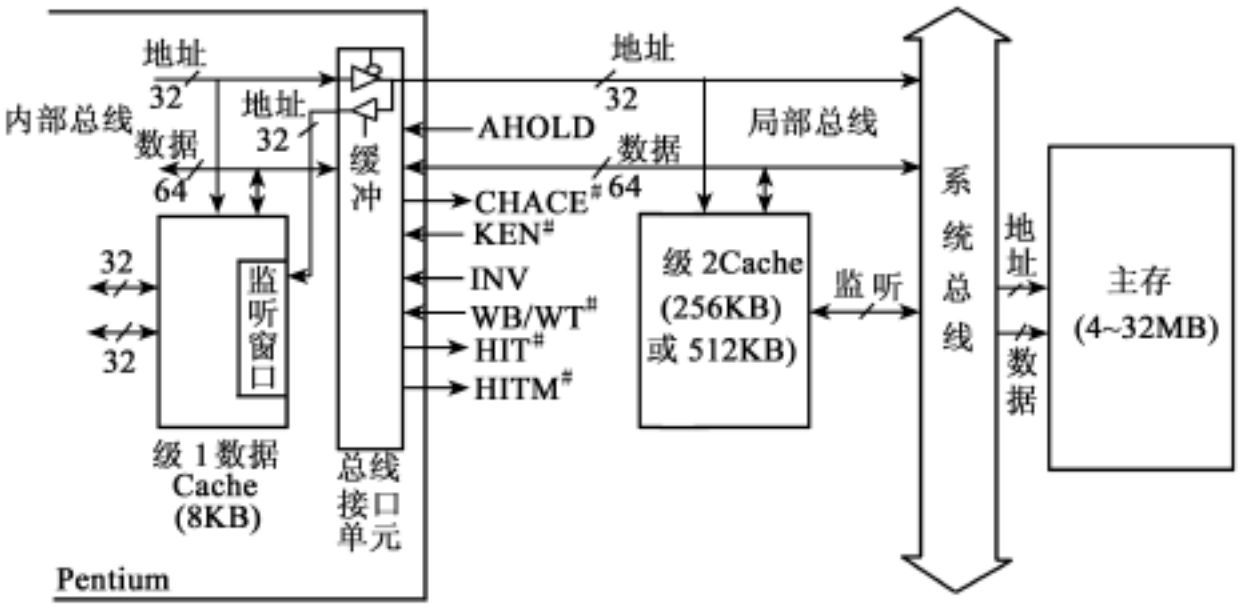


图 4 .61　Pentium 两级 Cache 工作环境

间,它的活动情况完全同于监听期间情况。由此可见,L2 负责整个系统的 Cache-主存的一致性;L1 负责响应 L2,与 L2 一起维护 L1-L2 一致性。

L2 采用的是写回法,并遵循 MESI 协议。L1 基本上是采用写回法,但在 CD = 0,NW = 0 的最佳设置情况下允许结合一些写直达操作,这里的写直达是既向片内 L1 写入又向片外启动一个存储器写总线周期。下面将会看到这种情况出现在对未修改行的第一次写操作时,故它是写一次法。而后对此修改行的再次或多次写命中采用的都是写回法策略,不向片外写,只在该行被换出或读/写监听命中时才写回。

L1 遵循 MESI 协议,但它将 E 态重新定义为第一次写命中后的修改态,实际上 L1 绝不会有同于主存而又不出现于 L2 的专有态行。图 4 .62 给出了 L1 的 MESI 协议状态转换图,该图将与 L1 有效行同内存(块)地址的 L2 有效行可能有的状态列出,以帮助理解此图。

① 当 L1 读未命中时要启动一个存储读总线周期,由片外读入一行填入空行(若没有空行,先以 LRU 算法换出一行)。此周期首先引起 L2 判测是否读命中,若读命中,L2 递交同地址的有效行并返回 WB/WT# 信号。若 L1 新行拷贝的是 L2 的 S 态行或 E 态行(WB/WT# 为低电平),新行状态为 S;若 L1 新行拷贝的是 L2 的 M 态行(WB/WT# 为高电平),新行状态为 E。若 L2 读未命中,则由主存读取此地址的数据块同时拷贝到 L2 和 L1,L2 的新行状态为 E 或 S,L1 的新行状态为 S,逻辑上仍可看做 L1 的新行是由 L2 拷贝而来。

② 对 L1 的 S 态行发生写命中,除此行在片内完成写修改并进入(特殊的)E 态外,还启动了一个片外的存储写周期,这就是前述的写一次操作。它必然引起 L2 的写命中,L2 的 E 态行或 S 态行完成写修改进入 M 态。此时注意,若是 I 上的 S 态行

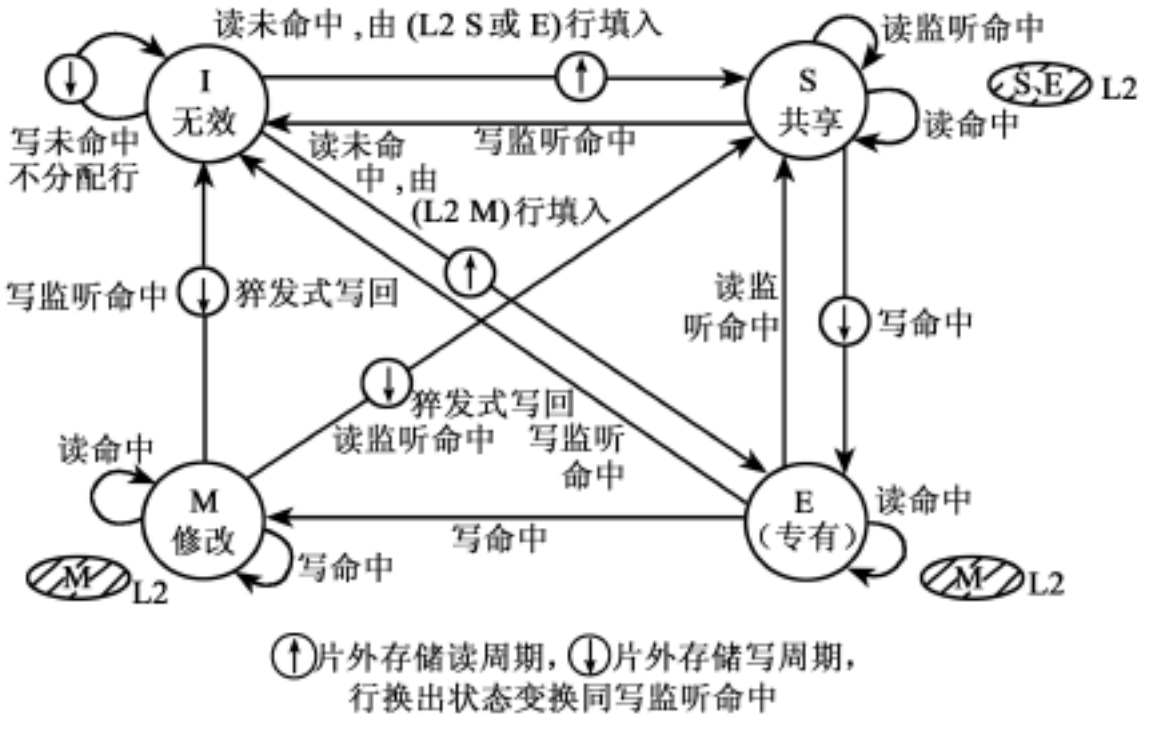


图4.62 Pentium L1的MESI状态转换图

写命中,还要引发一个无效处理过程(见图4.50),使系统中其他Cache的同此地址的共享行无效,这正是L1采用写一次法的原因。它将广播通知其他Cache无效处理的工作交由L2控制逻辑完成,也是它的巧妙之处。

③ L1写未命中时,Pentium只是将内存地址和具体修改字送出片外,即启动一个存储写总线周期,L1不分配新行。此时L2可能发生写命中,完成其有效行的写修改并进入或保持在M态;若是L2也是写未命中,则要读取内存块至L2再修改(见图4.50中的由I→M操作)。L1将这种读内存再修改的耗时工作又交给L2去完成了。当然,L1也要为此做出一点牺牲,即它不容许有这个最近存取过的拷贝,但此拷贝已在L2,下次读取时很快就可得到。这样,Pentium启动一个存储写周期后就可立即进行片内其他操作,支持了指令流水执行。

④ 前面已经说明MESI协议维护一致性的关键在于,修改过的Cache行在临界情况下及时写回主存以保证它者使用的正确性。L2监听系统总线上的其他主控者或其他Cache的访问主存活动。那么,我们来分析L2的M态行的L1同地址行有哪些状态:(a)L1M～L2M,这是L1至少两次写修改的情况,最新数据在L1M行;(b)L1E～L2M,这是L1写一次情况或是L1拷贝L2M行的情况,两行内容相同;(c)L1I～L2M,这是上面③所述的L1写未命中的情况,实际上L2的M态行此时在L1没有同地址行。换句话说,L2M态行包含了L1的M态行和E态行,而且L2的M态行与L1的M态行存在不一致性。

因此,当L2的M态行读/写监听命中时,L2要以AHOLD信号有效强行令L1

进入监听期间,来查询 L1 是否也监听命中。如果 L1 返回的是 HITM# 和 HIT# 信号都为有效,表明这是 L1M～L2M 情况,L1 的 M 态行读/写监听命中要以猝发方式将此行写到主存,然后该行相应地进入 S/I 态。如果 L1 返回的是 HITM# 无效而 HIT# 有效,表明这是 L1E～L2M 情况,L1 的 E 态行读/写监听命中只是相应地进入 S/I 态,由 L2 将它的 M 态行写回到主存(见图 4.50 的 M→S 与 M→I)。如果 L1 返回的是 HITM# 和 HIT# 都无效,表明这是 L1I～L2M 情况,一样也是由 L2 将它的 M 态行写回到主存,L1 无动作。

综上所述,L1 和 L2 的密切配合既保证了整个系统的 Cache-主存的一致性,又简化了 L1 的控制逻辑并对 CPU 的高速运行提供了有力支持。

习　　题

1. 解释下列术语:

存储器最大频宽	存储器实际频宽	模 m 交叉编址	程序局部性	虚拟存储器
段式管理	页式管理	段页式管理	地址的映像	地址的变换
堆栈型替换算法	Cache 存储器	直接映像	组相联映像	写回法
写直达法	恒预取法	不命中预取法	强制性失效	容量失效
冲突失效	牺牲者 Cache	伪相联 Cache	虚拟 Cache	环保护
访问方式保护				

2. 设二级虚拟存储器的 $TA_1 = 10^{-7}$ s, $TA_2 = 10^{-2}$ s,为使存储层次的访问效率 e 达到最大值的 80%以上,命中率 H 至少要求达到多少?实际上这样高的命中率是很难达到的,那么从存储层次上如何改进?

3. 要求主存实际频宽为 4MB/s,现设主存每个分体的存取周期为 2us,宽度为 4 字节,采用模 m 多分体交叉存取,但实际频宽只能达到最大频宽的 0.6 倍,问主存模数 m 应取多少方能使两者的速度基本匹配?(其中:m 取 2 的幂)

4. 采用页式管理的虚拟存储器中,什么叫"页面失效"?什么叫"页面争用"?什么时候,这两者不同时发生?什么时候,这两者又有可能同时发生?

5. 某虚拟存储器共 8 个页面,每页为 1 024 个字,实际主存为 4 096 个字,采用页表法进行地址映像。映像表的内容如表 4.12 所示。

(1)列出会发生页面失效的全部虚页号。

(2)按以下虚地址计算主存实地址:0,3 728,1 023,l 024,2 055,7 800,4 096,6 800。

表 4.12

实页号	3	1	2	3	2	1	0	0
装入位	1	1	0	0	1	0	1	0

6. 一个段页式虚拟存储器。虚地址有 2 位段号、2 位页号、11 位页内位移(按字编址),主存容量为 32KB 字。每段可有访问方式保护,其页表和保护位如表 4 .13 所示。

表 4 .13

段 号	段 0	段 1	段 2	段 3
访问方式	只 读	可读/ 执行	可读/ 写/ 执行	可读/ 写
虚页 0 虚页 1 虚页 2 虚页 3	实页 9 实页 3 在辅存内 实页 12	在辅存内 实页 0 实页 15 实页 8	页表不在主存内	实页 14 实页 1 实页 6 在辅存内

(1)此地址空间中共有多少个虚页?

(2)当程序中遇到如表 4 .14 所示的时:

表 4.14

方　　式	段	页	页内位移
取数	0	1	1
取数	1	1	10
取数	3	3	2 047
存数	0	1	4
存数	2	1	2
存数	1	0	14
转移至此	1	3	100
取数	0	2	50
取数	2	0	5
转移至此	3	0	60

写出由虚地址计算出的实地址。说明哪个会发生段失效、页失效或保护失效。

7. 设某程序包含 5 个虚页,其页地址为 4,5,3,2,5,1,3,2,2,5,1,3。当使用 LRU 法替换时,为获得最高的命中率,至少应分配给该程序几个实页?其可能的最高命中率为多少?

8. 设一个按位编址的虚拟存储器,它可对应 1 K 个任务,但在一段较长时间内,一般只有 4 个任务在使用,故用容量为 4 行的相联寄存器组硬件来缩短被变换的虚地址中的用户位位数;每个任务的程序空间最大可达 4 096 页,每页为 512 个字节,实主存容量为 2^{20} 位;设快表用按地址访问存储器构成,行数为 32,快表的地址是经散列技术形成的;为减少散列冲突,配有两套独立的相等比较器电路。请设计该地址变换机构,内容包括:

(1)画出其虚、实地址经快表变换的逻辑结构示意图。

(2)相联寄存器组中每个寄存器的相联比较位数。

(3)相联寄存器组中每个寄存器的总位数。

(4)散列变换硬件的输入位数和输出位数。

(5)每个相等比较器的位数。

(6)快表的总容量(以位为单位)。

9. 考虑一个 920 个字的程序,其访问虚存的地址流为 20,22,208,214,146,618,370,490,492,868,916,728。

(1)若页面大小为 200 字,主存容量为 400 字,采用 FIFO 替换算法,请按访存的各个时刻,写出其虚存地址流,计算主存的命中率。

(2)若页面大小改为 100 字,再做一遍。

(3)若页面大小改为 400 字,再做一遍。

(4)由(1),(2),(3)的结果得出什么结论？

(5)若把主存容量增加到 800 字,按第(1)小题再做一遍,由此又可得出什么结论？

10. 假设在一个采用组相联映像方式的 Cache 存储系统中,主存由 B0～B7 共 8 块组成,Cache 有 2 组,每组 1 块,每块大小为 16B。在一个程序执行过程中,访存的主存块地址流为:

B6,B2,B4,B1,B4,B6,B3,B0,B4,B5,B7,B3

(1)写出主存地址的格式,并标出各字段的长度。

(2)写出 Cache 地址的格式,并标出各字段的长度。

(3)画出主存与 Cache 之间各个块的映像关系。

(4)若 Cache 的 4 个块号为 C0,C1,C2 和 C3,列出程序执行过程中的 Cache 块地址流。

(5)若采用 FIFO 替换算法,计算 Cache 的块命中率。

(6)若采用 LRU 替换算法,计算 Cache 的块命中率。

(7)若改为全相联映像方式,再做(5)和(6)。

(8)若在程序执行过程中,每从主存装入一块到 Cache,平均要对这个块访问 16 次。请计算在这种情况下的 Cache 命中率。

11. 在某 Cache 存储器中,Cache 的容量是 2^C B,主存是由 m 个存储体组成的低位交叉访问存储器,主存总容量是 2^M B,每一个存储体的字长是 W 位。采用全相联映像方式,用相联目录表实现地址变换。

(1)画出地址变换图。

(2)写出主存地址和 Cache 地址的格式,并标出各字段的长度。

(3)说明目录表的行数、相联比较的位数和目录表的宽度。

12. 采用组相联映像的 Cache 存储器,Cache 容量为 1KB,要求 Cache 的每一块在一个主存周期内能从主存取得。主存采用 4 个低位交叉编址的存储体组成,每个存储体的字长为 32 位,主存容量为 256KB。采用按地址访问存储器存放块表实现地址变换,并采用 4 个相等比较电路。请设计此块表,求出该表的行数、总位数及每个比较电路进行相等比较的位数。

13. 有一个 Cache 存储器,主存共有 8 块(B0～B7),Cache 为 4 块(C0～C3),采用组相联映像方式,组内块数为 2 块,采用 LRU 替换算法。

(1)说明主存地址和 Cache 地址的格式及各字段的长度。

(2)画出主存块与 Cache 块之间的映像关系。

(3)在一个程序执行过程中,访存的主存块地址流为: B1,B2,B4,B1,B3,B7,B0,B1,B2,B5,B4,B6,B4,B7,B2。画出 Cache 中 C0～C3 的使用情况。

(4)对于(3),指出发生块失效且块争用的时刻,计算 Cache 的块命中率。

14. 在一个采用组相联映像方式的 Cache 存储器中,Cache 的容量为 16KB。主

存采用模 8 低位交叉编址方式访问,每个存储体的字长为 32 位,总容量为 8MB。要求 Cache 的每一块在一个主存周期取得,Cache 的每一组内共有 4 块。采用按地址访问存储器存放块表实现地址变换,并采用 8 个相等比较电路。

(1)设计主存地址和 Cache 地址的格式,并说明各字段的长度。

(2)块表的行数和宽度是多少?每个比较电路的比较位数是多少?

15. 一个采用组相联映像方式的 Cache 共有 8 块,分为 2 组。用硬件的比较对法实现 LRU 块替换算法。

(1)共需要多少个触发器和多少个与门?

(2)画出其中一组的逻辑图。

16. 在一个页式二级虚拟存储器中,采用 FIFO 算法进行页面替换,发现命中率片太低,为此有下述建议:

(1)增大辅存容量。

(2)增大主存容量(页数)。

(3)增大主、辅存的页面大小。

(4)将 FIFO 改为 LRU。

(5)将 FIFO 改为 LRU,同时增大主存容量(页数)。

(6)将 FIFO 改为 LRU,同时增大主存页面大小。

试分析上述各建议对命中率可能有什么影响?

17. 采用组相联映像的 Cache 存储器,Cache 为 1KB,要求 Cache 的每一块在一个主存周期内能从主存取得。主存采用模 4 交叉,每个分体宽度为 32 位,总容量为 256KB。采用按地址访问存储器构成的相联目录表,实现主存地址到 Cache 地址的变换,并约定采用 4 个外相等比较电路。请设计此相联目录表,求出该表的行数、总位数及每个比较电路的位数。

18. 有一个“Cache-主存”存储层次。主存共分 8 个块(0～7),Cache 为 4 个块(0～3),采用组相联映像,组内块数为 2 块,替换算法为近期最少使用法(LRU)。

(1)画出主存、Cache 存储器地址的各字段对应关系(标出位数)。

(2)画出主存、Cache 存储器空间块的映像对应关系的示意图。

(3)对于如下主存块地址流:1,2,4,1,3,7,0,l,2,5,4,6,4,7,2,如主存中内容一开始未装入 Cache 中,请列出随时间的 Cache 中各块的使用状况。

(4)对于(3),指出块失效又发生块争用的时刻。

(5)对于(3),求出此期间 Cache 的命中率。

19. 采用组相联映像、LRU 替换算法的“Cache - 主存”存储层次,发现等效访问速度不高,为此,提议:

(1)增大主存容量。

(2)增大 Cache 中的块数(块的大小不变)。

(3)增大组相联组的大小(块的大小不变)。

(4)增大块的大小(组的大小和 Cache 总容量不变)。

(5)提高 Cache 本身器件的访问速度。

试问分别采用上述措施后,对等效访问速度可能会有什么样的显著变化?其变化趋势如何?如果采取措施后并未能使等效访问速度有明显提高的话,又是什么原因?

20. 你对现有“Cache”存储层次的速度不满意,于是你申请到一批有限的经费,为了能发挥其最大的经济效益,有人建议你再去购买一些同样速度的高速缓冲存储器芯片对高速缓冲存储器容量加以扩充;而另外一些人却建议你不如干脆买更高速的缓冲存储器芯片全部更换掉现有低速的缓冲存储器芯片。你认为哪种建议可取?你如何做决定?为什么?

第五章 流水技术与向量处理

计算机系统设计者的基本任务是提高处理机指令的执行速度,而采取的主要措施是指令级的并行性,即让多条指令同时参与解释的过程。常用的有三种方法:

① 采用流水线技术,称为流水线处理机或超流水线处理机(Super Pipelining)。

② 在一个处理机中设置多个独立的功能部件,例如,在一个处理机中设置独立的定点算术逻辑部件、浮点加法部件、乘除法部件、访问存储器部件以及分支操作部件等,称为多操作部件处理机或超标量处理机(Superscalar)。也可以把超流水线技术与超标量技术结合起来,称为超标量超流水线处理机。

③ 超长指令字(Very Long Instruction Word, VLlW)技术,在一条指令中设置有多个独立的操作字段,每个字段可以分别独立地控制各个功能部件并行工作。

目前,前两种技术已经相当成熟,已经研制出了多种高性能的超标量和超流水线处理机,而超长指令字技术还在进一步研究中。

本章以介绍流水线技术为主,包括先行控制技术、流水线原理、流水线性能分析,非线性流水线的调度方法、局部数据相关和全局数据相关的处理方法,超标量处理机和超流水线处理机等,最后介绍向量流水和向量处理机。

5.1 标量流水工作原理

5.1.1 指令的重叠解释方式

一条指令的执行过程可以分为多个阶段,具体的分法要根据各种处理机的具体情况和分析过程而确定。一般把一条指令的解释过程分为 3 个(取指、分析和执行)或 5 个(取指、译码、执行、访存和写回)阶段。依据 DLX 指令流水方式的五个子过程及功能定义如表 5.1 所示。

表 5.1 DLX 指令基本流水解释过程

序号	子过程周期	代码	功能说明		
			ALU 指令	LOAD/ STORE 指令	BRANCH 指令
1	取指令	IF	按照 PC 指示的主存地址,取出一条指令送指令寄存器中		

续表

序号	子过程周期	代码	功 能 说 明		
			ALU 指令	LOAD/ STORE 指令	BRANCH 指令
2	指令译码	ID	对指令操作码译码,读寄存器堆(RF)		
3	执行	EX	执行运算	有效地址计算	目标地址计算,设置条件码
4	存储器访问	MEM	——	访存(读/ 写)	若成功,目标地址送 PC
5	写回	WB	结果写回 RF	读出的数据写入 RF	——

如果将一条指令的执行过程分为 3 个子过程,当有多条指令要在处理机中执行时,可以有多种执行方式:

1. 顺序执行方式

指令的执行过程如图 5 .1 所示。采用顺序执行方式执行 n 条指令所用的时间为:

$$T = \sum_{i=1}^{n} (t_{取指i} + t_{分析i} + t_{执行i})$$

如果取指令、分析指令和执行指令的时间都相等,每段的时间都为 t,则执行 n 条指令所用的时间为: $T = 3nt$。

按照寻址方式和地址字段中的内容形成操作数的地址,并用这个地址读取操作数(MEM)。

采用顺序执行方式的优点是控制简单,容易实现。主要缺点:一是处理机执行指令的速度慢。只有当上一条指令全部执行完之后,下一条指令才能够开始执行,即在任何时刻,处理机中只有一条指令在执行;二是功能部件的利用率很低。例如,在取指令和分析指令时,主存储器是忙碌的,但指令执行部件是空闲的,同样,在执行指令时,指令执行部件是忙碌的,但主存储器和指令分析部件等经常是空闲的。

2. 一次重叠执行方式

这是一种最简单的重叠方式,即把执行第 k 条指令与取第 k + 1 条指令同时(时间上重叠)进行。图 5 .1 中所示 3 条指令一次重叠执行过程。

如果取指令、分析指令和执行指令的时间都相等,每段的时间都为 t,则执行 n 条指令所用的时间为:

$$T = (1 + 2n)\,t$$

采用一次重叠执行方式的优点主要有两个,一是指令的执行时间与顺序执行相比缩短;二是功能部件的利用率明显提高。缺点是需要增加一些硬件,控制过程也要复杂一些。例如,为了能够在执行第 k 条指令的同时,分析第 k + 1 条指令,必须再增加一个指令寄存器。用原来的指令寄存器存放当前正在执行的第 k 条指令,而新增加的一个指令寄存器存放新取出来的第 k + 1 条指令。

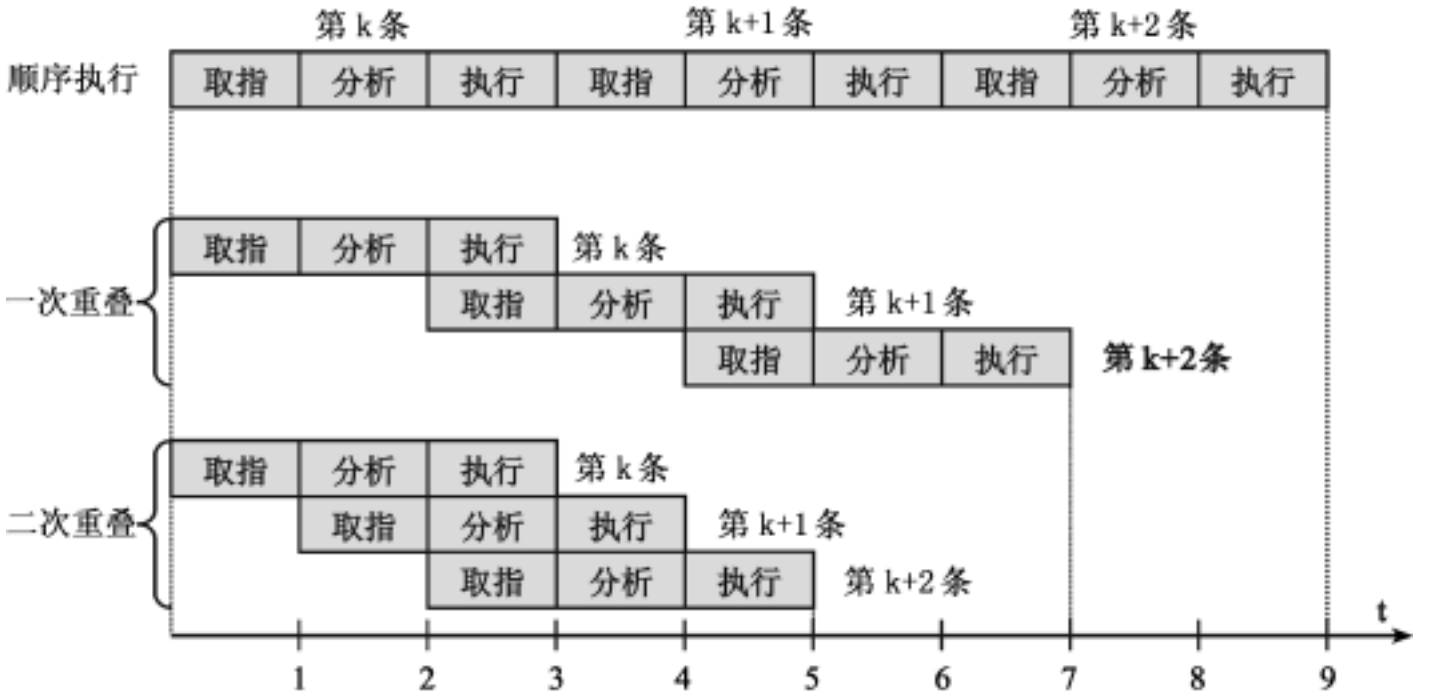


图 5 .1　三条指令解释过程比较

3. 二次重叠执行方式

为了进一步提高指令的执行速度,可以把取第 k + 1 条指令提前到分析第 k 条指令同时进行,而把分析第 k + 1 条指令与执行第 k 条指令同时进行,把取第 k + 2 条指令与分析第 k + 1 条和执行第 k 条指令同时进行。图 5 .1 中所示 3 条指令的二次重叠执行过程。

如果执行一条指令的 3 个过程的时间相等,则执行 n 条指令所用的时间为:

$$T = (2 + n)\,t$$

采用二次重叠执行方式能够使指令的执行时间更进一步缩短,这是一种理想的指令执行方式。

上面分析过程只是在非常理想情况下进行的,实际存在许多因素:资源冲突(如指令的 3 个子过程有可能同时访问主存储器而引起的与主存资源相关)、数据相关、三个功能部件的执行时间不相等(相互之间需要等待时间)以及控制流的改变(如转移指令)等,会使得重叠解释的速度降低,有关这类问题将在后面进行详细分析。

5.1.2 先行控制技术

1. 先行控制技术原理

先行控制(Look-Ahead)技术最早在 IBM 公司研制的 STRETCH 计算机中采用。目前,许多处理机中都已经采用了这种技术,包括超流水线处理机和超标量处理机等。

先行控制技术的关键是缓冲技术和预处理技术,以及这两者的结合。通过对指令流和数据流的预处理和缓冲,能够尽量使指令分析器和指令执行部件独立地工作,并始终处于忙碌状态,以提高处理器中部件的利用率。同时,先行控制技术也是解决指令重叠解释过程中,取指令、分析指令和执行指令三个部件访问主存冲突的根本办法。

缓冲技术是指在工作速度不固定的两个功能部件之间设置缓冲栈,用以平滑它们的工作速度。缓冲栈包括指令缓冲和数据缓冲,在采用先行控制的处理机中,一般要设立四个缓冲栈,它们分别是先行指令缓冲栈、先行操作栈、先行读数栈和后行写数栈。

预处理技术是把进入运算器的指令都预处理成寄存器—寄存器型指令,它与缓冲技术相结合,为进入运算器的指令准备好所需的全部操作数。

2. 先行控制技术的实现

先行控制技术的处理机结构如图 5.2 所示。

“先行指令栈”实为先行指令缓冲栈(或称指令缓冲栈),由一个指令缓冲寄存器堆和独立的控制逻辑构成。它可以把后续的指令“先行”取出,存放在缓冲栈中,从而为指令分析器分析新的指令做好准备。在处理机内部设置一定容量的指令缓冲栈,把指令分析器所需要的指令事先取到指令缓冲栈中,而不必访问主存储器。这样,就能够使取指令、分析指令和执行指令重叠起来执行。

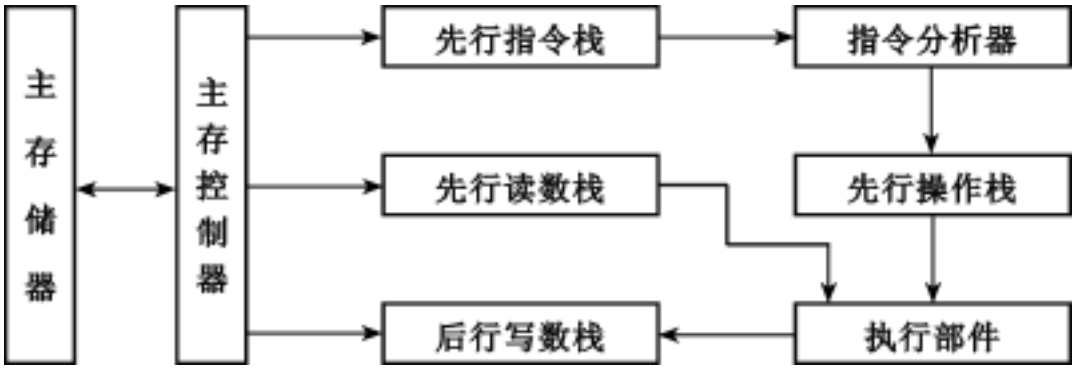


图 5.2 先行控制结构示意图

只要指令缓冲栈还没有全部充满,它就自动向存储控制器发出取指令的请求。同样,指令分析器每分析完一条指令也自动向指令缓冲栈发出取下一条指令的请求,指令取出以后就把先行指令缓冲栈中的指令作废。由于指令分析器取走指令的速度和从主存储器中取来指令的速度都是随机的,因此,指令缓冲栈中的指令数目是动态变化的。另外,在有先行指令缓冲栈的处理机中,要设置两个程序计数器,一个是先行程序计数器 PC$_1$,用来指示到主存储器中取指令,另一个是现行程序计数器,它也就是原来意义上的程序计数器 PC,用来记录指令分析器当前正在分析的指令地址。

先行操作栈是对指令分析器提供的指令进行预处理,即将所有指令转换为寄存器—寄存器型指令,以提高执行部件的处理速度。

先行读数栈和后行写数栈是两个数据缓冲栈,由若干个寄存器组成。其作用表现为两个方面:一是与先行操作栈配合,完成指令预处理过程中的操作数的读取;二是解决指令重叠解释过程中各功能部件同时访问主存而发生的冲突。

从图 5.2 中可知,主存储器的访问源有三个,即先行指令栈、先行读数栈和后行写数栈。在一般处理机中,存储控制器把这三个访问源的优先次序由高到低安排为:后行写数栈、先行读数栈、先行指令栈。因此,先行指令缓冲栈是利用主存储器的空闲时间来取指令的。只要主存储器的频带宽度足够,就能够保证先行指令缓冲栈从主存储器中取到指令。如果指令分析器每次取指令都能够在先行指令缓冲栈中得到,则取指令只需要很短的时间就能够完成,因此,可以把取指令与分析指令合并到一起,从而构成指令的一次重叠执行方式。

先行控制技术实际上是缓冲技术和预处理技术的结合,通过对指令流和数据流的先行控制,使指令分析器和指令执行部件能尽量连续地工作。这里的缓冲部件一般都是采用先进先出的工作方式,如何合理地确定缓冲部件中的寄存器个数,即缓冲深度的设计是一个关键问题。如果缓冲寄存器的个数太少,往往起不到缓冲作用,指令分析器和指令执行部件不能连续地工作。相反,如果缓冲寄存器的个数设置太多,不仅浪费设备,而且控制逻辑也复杂。

由于在缓冲器的输入和输出端,数据(或指令)流动的速度是动态变化的,要建立一个准确的数学模型来求出各个缓冲器的深度是非常困难的。只能通过一种动态分析方法,得出各个缓冲深度(D)之间的一般关系如下:

$$D_{指缓} \geq D_{操作栈} \geq D_{读数} \geq D_{写数}$$

例如,在 IBM 370/165 计算机中,$D_{指缓}=4$,$D_{操作栈}=3$,$D_{读数}=2$,$D_{写数}=1$。我国研制的两台大型计算机中,其中一台 $D_{指缓}=8$,$D_{操作栈}=D_{读数}=4$,$D_{写数}=2$,另外一台 $D_{指缓}=12$,$D_{操作栈}=D_{读数}=6$,$D_{写数}=2$。

5.1.3 标量流水工作原理

只有标量数据表示和标量指令系统的处理机称为标量处理机。标量处理机是一种最通用,也是使用最普遍的一种处理机。

1. 从重叠到流水

前述的重叠工作方式实际上是标量流水方式的原型。标量流水实际上是重叠工作方式的进一步发展(或重叠的进一步引伸)。如果把一条指令的解释过程进一步细分,例如,把取指、分析、执行 3 个过程进一步分成取指(IF)、译码(ID)、执行(EX)、访存(MEM)和写回(WB)5 个子过程,并用 5 个子部件分别处理这 5 个子过程。这样只需在上一指令的第一子过程处理完毕进入第二子过程处理时,在第一子部件中就开始对第二条指令的第一子过程进行处理,随着时间推移,这种重叠操作最后可达到 5 个子部件同时对 5 条指令的子过程进行操作。

由于这种工作方式与工厂中的装配流水线相类似,所以称这种工作方式为流水方式。由 5 个部件组成的流水线组成示意图如图 5 .3 所示。

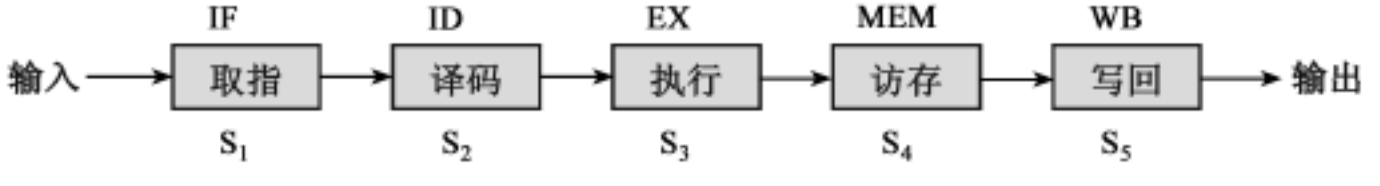


图 5 .3　流水线组成示意图

流水工作时,指令或数据从上一级向下一级流动,由流水线中的公共时钟控制,实现同步传送。在物理实现流水线时,各相邻级及输入、输出级都需有锁定(Latch)电路,以暂存欲传送的中间结果。另外,要求各个子过程(或功能段)的处理时间相等,如果各个子过程的处理时间不相等,则流水工作将引起“堵塞”或“断流”,各功能部件的作用不能充分发挥,流水线性能下降。在流水线中执行时间最长的功能段称为“瓶颈”,当遇到“瓶颈”时,必须采取办法消除,以保证流水线的畅通。

2. 流水线的时空图

在分析流水线工作过程和性能计算时经常使用时空图方法。在时空图中,常以横轴为时间轴,表示流水线完成若干个任务所经过的时间;纵轴为空间(功能段/部件)轴,表示构成流水线的功能段数。根据图 5 .3 所示 5 个功能段构成的流水线,共完成 n 个任务的流水线时空图如图 5 .4 所示。

由图 5 .4 可知,如果一个流水线中每个功能段的延迟时间都相等(Δt),完成 n 个任务的总时间为:

$$T = m\Delta t + (n - 1)\Delta t$$

该时间分为两部分,一是第一个任务经过流水线的时间,二是剩余$(n - 1)$个任务的完成时间。

另外,从流水线的工作状态来看,有三个时间阶段:建立时间、正常工作时间和排

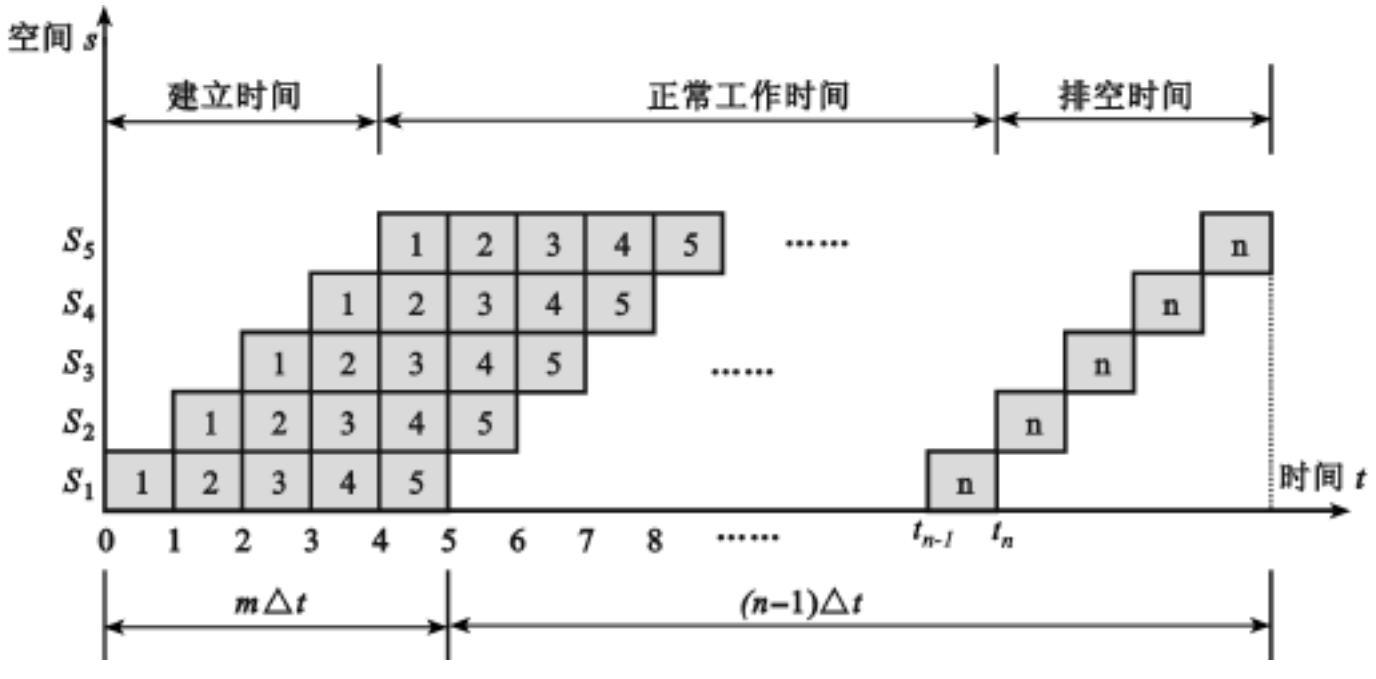


图 5 .4　流水线时空图

空时间。在建立时间和排空时间阶段,流水线中的某些功能部件是处于空闲状态的,只有在正常工作时间阶段,流水线才处于满负荷状态。

3. 流水线的特点

由上介绍可知,在流水技术中,一般具有如下特点:

① 一个流水线通常由若干个功能段组成。

② 每个流水段有专门的功能部件对指令进行某种加工。

③ 各流水段所需时间是一样的,因为各功能段之间及输入、输出级都需有锁定电路,以暂存欲传送的中间结果。

④ 流水线工作阶段可分为建立、正常工作(满载)和排空三个阶段。

⑤ 在理想情况下,当流水线正常工作后,每隔 Δt 时间将会有一个结果流出流水线。

5.1.4　标量流水线的分类

从不同的角度,按照不同的观点可以将流水线分成多种不同的种类。

1. 按照处理级别分类

按处理级别可将流水线分为指令级、操作部件级和处理机级。

指令级流水则是把一条指令解释过程分成多个子过程,如前面所提到的:取指、译码、执行、访存及写回 5 个子过程(见图 5 .3)。在采用先行控制器的处理机中,组成先行控制器的各个部件实际上也构成了一条流指令流水线,如图 5 .5 所示,在先行

控制器中,一条指令的执行过程被分解为 5 个子过程,每个子过程在一个专用的功能部件中执行。由于各种指令在同一个功能部件中执行的时间往往相差很大,因此,在每一个功能段之间要设置多个缓冲寄存器,以平滑流水线中各个功能部件的操作。

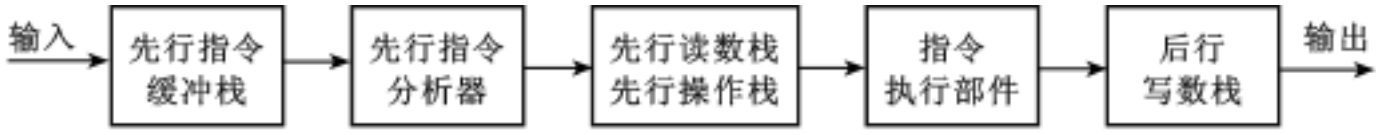


图 5 5　先行控制方式中的指令流水线

操作部件级流水是将复杂的算术逻辑运算组成流水工作方式。例如,可将浮点加法操作分成求阶差、对阶、尾数相加以及结果规格化 4 个子过程。

处理机级流水是一种宏流水,其中每个处理机完成某一专门任务。各个处理机处理所得到的结果需存放在与下一个处理机所共享的存储器中,如图 5 .6 所示。

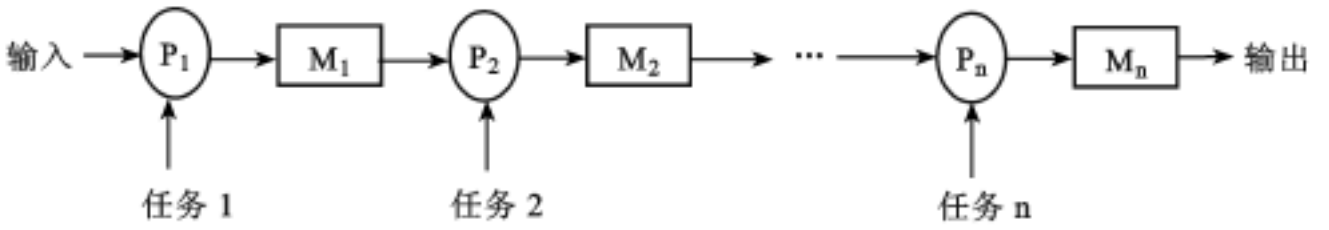


图 5 .6　处理机级流水线(P 为处理机,M 为存储器)

2. 按功能分类

流水线按功能可分成单功能流水线和多功能流水线。

单功能流水线只完成一种功能。如浮点加法或乘法流水线。

多功能流水线则可完成多种功能,它允许在不同时间,甚至同一时间内在流水线内连接不同功能段来实现不同的功能。如 TI-ASC 计算机中的一个多功能运算流水线,共有 8 个功能段,按需要它可将不同的功能段连接起来完成某一功能,如图 5 .7 所示。它可实现定点加、浮点加和定点乘等功能。这种方式虽然能充分利用功能部件,但控制也比较复杂。因此大多数流水计算机主要采用单功能流水线。

3. 按工作方式分类

流水线按这种分类方式可分为静态流水线和动态流水线。

在静态流水线中,同一时间内它只能以一种功能方式工作。它可以是单功能的,也可以是多功能的。当是多功能流水线时,即从一种功能方式变为另一种功能方式时,必须先排空流水线,然后为另一种功能设置初始条件后方可使用。显然,不希望这种功能的转换频繁地发生,否则将严重影响流水线的处理效率。

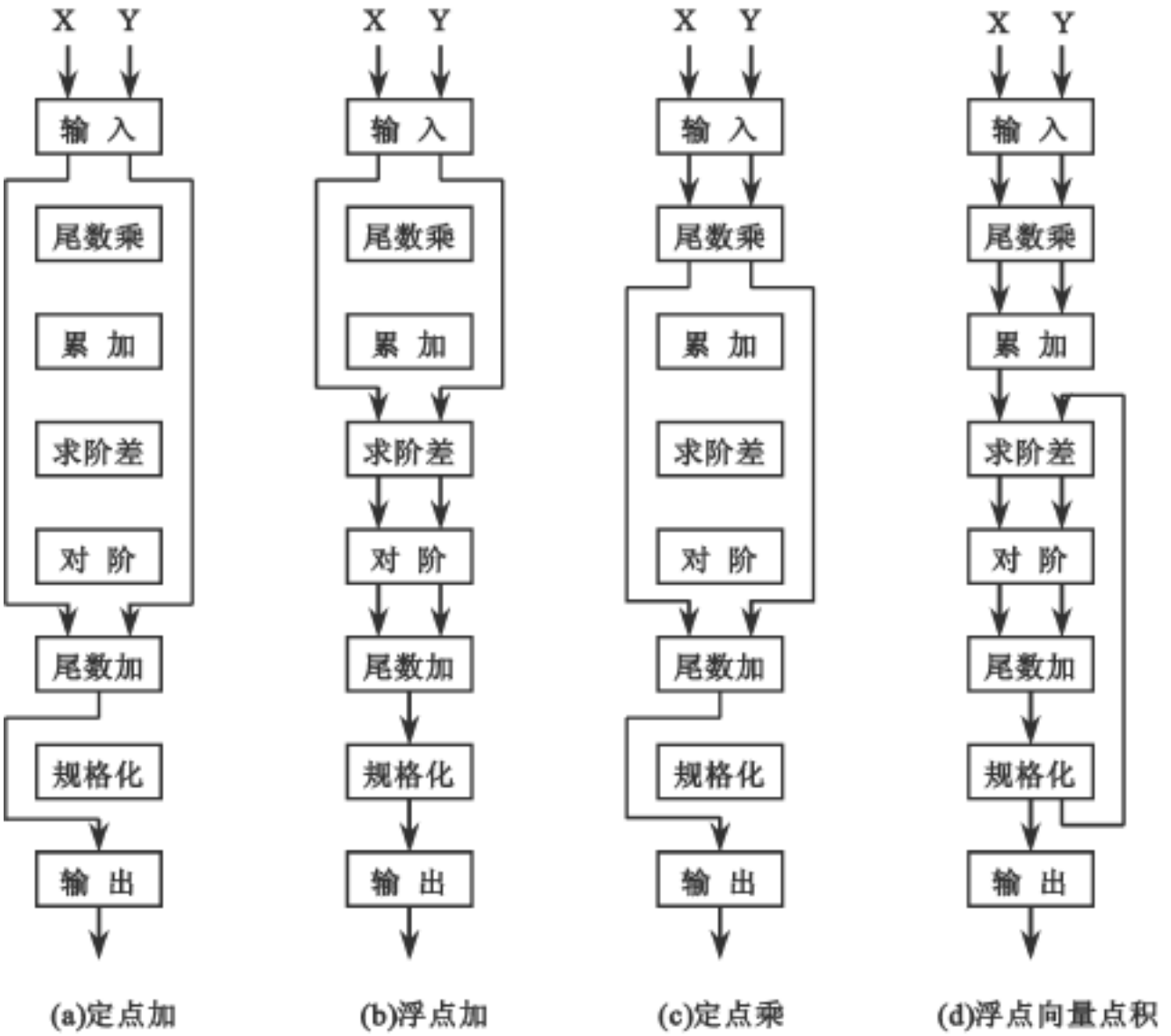


图 5 .7　TI-ASC 计算机的多功能流水线

动态流水线则允许在同一时间内将不同的功能段连接成不同的功能子集(前提条件是功能部件的使用不发生冲突),以完成不同的运算功能。显然,动态流水线必须是多功能流水线,而单功能流水线则必须是静态的。

4. 按连接方式分类

流水线按连接方式可以分为线性流水线与非线性流水线。

在线性流水线中,从输入到输出,对于一个任务每个功能段只允许经过一次,不存在反馈(或前馈)回路。一般的流水线均属这一类。

非线性流水方式中则存在反馈(或前馈)回路,因此从输入到输出过程中,一个任务将数次通过流水线中的某些功能段。这种流水线适合于进行线性递归的运算。图 5 .8 中显示出了这样的一个非线性流水线。S_3 的输出可反馈到 S_2,而 S_4 的输出可反馈到 S_1。这种非线性流水线对输入流水线加工的指令要加以较复杂控制以保证两条或多条指令不会发生对某一功能段的争用。有关调度方法将在后面讨论。

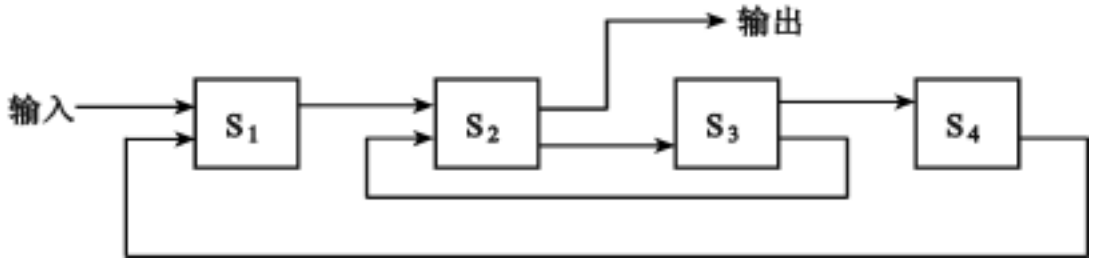


图 5.8 一种非线性流水线

5.1.5 标量流水线性能分析

衡量流水线性能的主要技术指标是:吞吐率、效率和加速比。

流水线性能指标的计算可用画时空图或直接用公式求解。

1. 吞吐率

流水线的吞吐率(Thoughput Rate,TP)是指单位时间内从流水线中流出的任务(结果)数。如前所述,由于流水线并不是在任何时刻都满负荷地工作,因此,它又分为最大吞吐率和实际吞吐率。

(1)最大吞吐率

它是指流水线达到稳定状态(正常工作时间内)可获得的吞吐率。

如果流水线中每个功能段的延迟时间都相等(均为 Δt),那么,流水线的最大吞吐率为:

$$TP_{max} = 1/\Delta t$$

也可以理解为,当流水线满负荷工作时,从它的输出端每隔一个单位时间 Δt 就可有一个任务或结果流出。

如果每个功能段的延迟时间不相等,则最大吞吐率的表达式为:

$$TP_{max} = \frac{1}{\max\limits_{i=1}^{n}\{\Delta t_i\}}$$

流水线中延迟时间最大的功能段称为流水线中的瓶颈。消除瓶颈的基本方法有两种:

一是将瓶颈子过程进一步细分成若干个子过程,使每一个子过程与其他子过程时间相等。如图 5.9(a)所示为一带有瓶颈部件的流水线,其中功能部件 3 为瓶颈。将瓶颈子过程再细分后的流水线如图 5.9(b)所示。

二是在瓶颈段,并联设置多套功能段部件,使它们轮流工作。当瓶颈子过程无法再细分时,可采取此方法消除流水线中的瓶颈,如图 5.9(c)所示。

(2)实际吞吐率

以上所述是流水线在连续流动时可达到的最大吞吐率。实际上,由于流水线在

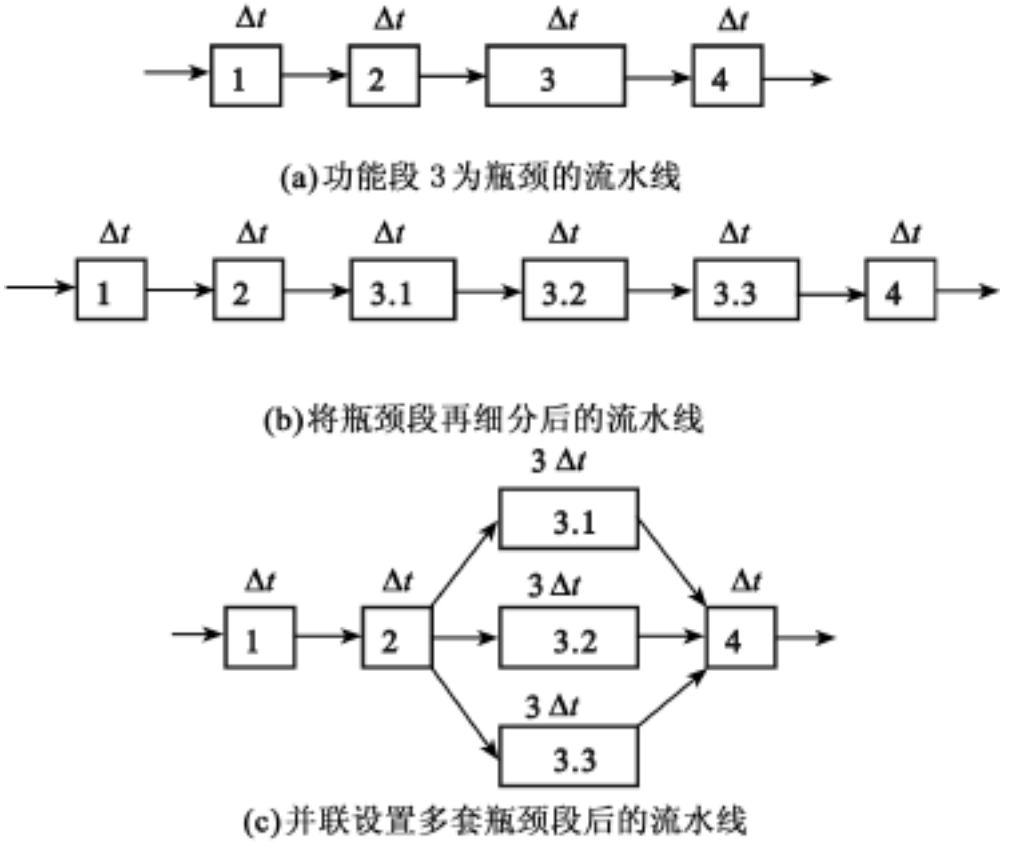


图 5 9　消除瓶颈的两种方法

开始时有一段建立时间,结束时有一段排空时间,以及由于各种相关因素使流水线无法连续流动,所以,实际吞吐率总是小于最大吞吐率。

设流水线由 m 个功能段组成,每段的延迟时间为 Δt_i。连续处理的任务数为 n,则流水线实际吞吐率的表达式为:

$$TP = \frac{\text{完成的任务数 } n}{\text{完成 } n \text{ 个任务的时间}} = \frac{\text{完成的任务数 } n}{\text{第一个任务的流出时间} + (n-1)\text{个任务的完成时间}}$$

以下分两种情况讨论流水线的实际吞吐率:

第一种情况是各功能段的延时时间相等,即 $\Delta t_i = \Delta t_j\ (i, j = 1, 2, \cdots, m)$。完成 n 个任务的时间为:

$$T = m\Delta t + (n-1)\Delta t$$

实际吞吐率为:

$$TP = \frac{n}{m\Delta t + (n-1)\Delta t} = \frac{1}{\Delta t\left(1 + \frac{m-1}{n}\right)} = \frac{TP_{max}}{1 + \frac{m-1}{n}}$$

由此可见,实际吞吐率不仅总是小于最大吞吐率,且只有当 $n \gg m$,即当输入的任务(或指令)数足够多时,实际吞吐率才能接近最大吞吐率。

第二种情况是各功能段的延迟时间不相等,即 $\Delta t_i \neq \Delta t_j\ (i, j = 1, 2, \cdots, m)$。并设瓶颈功能段的延迟时间为 Δt_j。则完成 n 个任务的时间为:

$$T = \sum_{i=1}^{m} \Delta t_i + (n - 1)\Delta t_j$$

实际吞吐率为:

$$TP = \frac{n}{\sum_{i=1}^{m} \Delta t_i + (n - 1)\Delta t_j}$$

2. 效率

流水线的效率(Efficiency)是指流水线中的各功能段(或设备)的利用率。由于流水线存在建立时间和排空时间,因此各功能段的设备不可能一直处于工作状态,总有一段空闲时间。效率的表达式一般用流水线各功能段处于工作时间的时空区与流水线中各段总的时空区之比。即为:

$$E = \frac{n \text{ 个任务占用的时空区}}{m \text{ 个功能段总的时空区}}$$

同样,对于流水线的效率也有两种情况:

当各功能段的延时时间相等时,流水线的效率为:

$$E = \frac{mn\Delta t}{mT} = \frac{n\Delta t}{T} = TP\Delta t$$

式中,分母 mT 是时空图中 m 个功能段和流水总时间 T 所围成的面积,分子 $mn\Delta t$ 是时空图中 n 个任务实际使用的面积。因此,从时空图上看(参见图 5.4),只有当 $n \gg m$ 时,E 才趋于 1。

对于线性流水且每段经过时间相等时,流水线的效率是正比于吞吐率的,当然,对于非线性流水或线性流水各段经过的时间不等时,这种成正比的关系就不存在了,此时应通过画实际工作的时空图才能求出吞吐率和效率。

当各功能段的延时时间不相等时,流水线的效率为:

$$E = \frac{n\sum_{i=1}^{m} \Delta t_i}{m\left[\sum_{i=1}^{m} \Delta t_i + (n - 1)\Delta t_j\right]}$$

3. 加速比

加速比(Speedup Ratio)是指采用流水方式后的工作速度与等效的顺序串行方式的工作速度之比。对 n 个求解任务而言,若用串行方式工作需要时间为 T_1,而用 m 段流水线来完成同一任务所需要时间为 T_2,则加速比为:

$$S_p = \frac{T_1}{T_2}$$

当各功能段的延时时间相等时,流水线的加速比为:

$$S_p = \frac{nm\Delta t}{m\Delta t + (n-1)\Delta t} = \frac{m}{1 + \frac{m-1}{n}}$$

当 $n \gg m$ 时，有 $S_p = m$，显然要获得高的加速比，流水线段数 m 应尽可能取大，即应加大流水深度。

如果各功能段的延时时间不相等，流水线的加速比为：

$$S_p = \frac{n\sum_{i=1}^{m}\Delta t_i}{\sum_{i=1}^{m}\Delta t_i + (n-1)\Delta t_j}$$

4. 流水线性能分析举例

例 5.1　带有瓶颈部件的 4 功能段流水线如图 5.10 所示。在该流水线上分别连续输入 3 条指令和 30 条指令时，求吞吐率、效率和加速比。

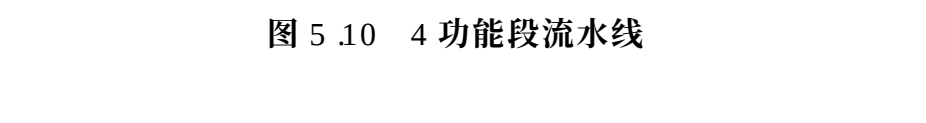


图 5.10　4 功能段流水线

求此题有两点值得注意：一是流水线中各个功能部件的执行时间不相等，即流水线中有“瓶颈”存在；二是连续输入 30 个任务时，由于任务数较多不能用画时空图的办法。下面用两种方法分别求解，即对于连续输入 3 个任务，用画时空图的方法求解，而连续输入 30 个任务时直接按照定义(即用公式法)求解。

对于连续输入 3 个任务，画出的流水线工作示意图如 5.11 所示。由图中可求出流水线实际吞吐率、效率和加速比分别如下：

$$TP = \frac{\text{完成 3 个任务}}{\text{完成 3 个任务共需用 } 14\Delta t} = \frac{3}{14\Delta t}$$

$$E = \frac{n\text{个任务占用的时空区}}{m\text{ 个功能段流水总时空区}} = \frac{3\text{ 个任务}\times\text{每个任务需 }8\Delta t}{4\text{ 个功能段}\times\text{总时间 }14\Delta t} = \frac{3\times 8\Delta t}{4\times 14\Delta t} \approx 42.86\%$$

$$S_p = \frac{\text{串行方式工作时间}}{\text{流水线方式完成同一任务时间}} = \frac{3\text{ 个任务}\times\text{每个任务需 }8\Delta t}{14\Delta t} = \frac{3\times 8\Delta t}{14\Delta t} \approx 1.71$$

当连续输入 30 个任务时，先计算出流水线完成 30 个任务的总时间为：

T = 第一个任务经过时间 + $(n-1)$ 个任务的流出(完成)时间
= 各功能部件延迟时间和 + $(n-1)\times$瓶颈部件的延迟时间
= $8\Delta t + (30-1)\times 3\Delta t$
= $95\Delta t$

实际吞吐率、效率和加速比如下：

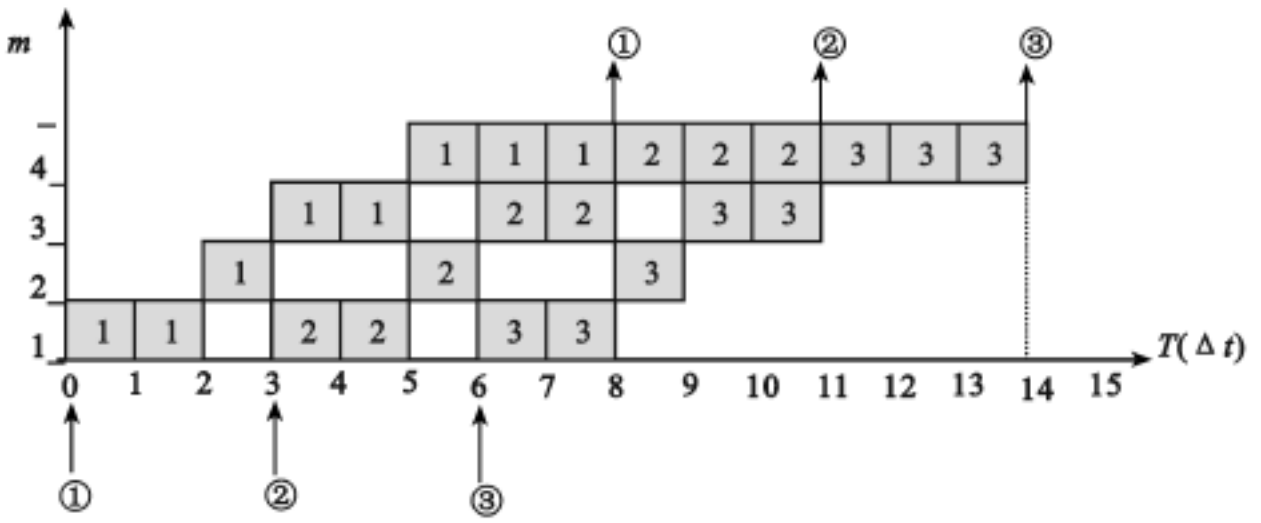


图 5.11 连续输入 3 个任务时工作示意图

$$TP=\frac{\text{完成 30 个任务}}{\text{完成 30 个任务共需总时间}}=\frac{30}{8\Delta t+29\times 3\Delta t}=\frac{30}{95\Delta t}$$

$$E=\frac{\text{完成 30 个任务}\times\text{每个任务需 }8\Delta t}{4\text{ 个功能段}\times\text{完成任务的总时间 }95\Delta t}=\frac{30\times 8\Delta t}{4\times 95\Delta t}\approx 63.16\%$$

$$S_p=\frac{\text{串行方式工作时间}}{\text{流水线方式完成同一任务时间}}=\frac{\text{30 个任务}\times\text{每个任务需 }8\Delta t}{95\Delta t}=\frac{30\times 8\Delta t}{95\Delta t}\approx 2.53$$

例 5.2 有一双功能的静态流水线如图 5.12 所示。功能段 1→5→6 组成加法流水线,1→2→3→4→6 组成乘法流水线。每功能段延时时间为 Δt。求在流水线上执行 $\prod_{i=1}^{4}(a_i+b_i)$ 的吞吐率和效率。(设数据已从主存中取出,输出数据可直接返回输入)

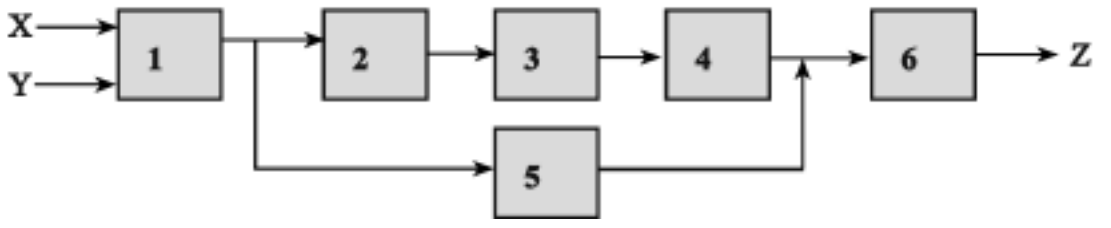


图 5.12 具有双功能的静态流水线

解题时应注意以下几个问题:

① 由于此题涉及多个问题,如流水线的功能切换,输入的任务数不是很明确,如何组织任务的输入才能使流水线的工作效率最高,等等。因此,这类题型只能用画时空图的方法求吞吐率和效率,而不能直接用公式法。

② 根据题意,必须先按 1,5,6 组成加法流水线做完全部加法,然后再将流水线功能切换到 1,2,3,4,6 做乘法(解题时功能切换时间一般忽略不计)。

③ 做加法时可连续输入 4 个任务,直至完成所有加法。而在做乘法时应注意最

后一个乘法任务必须等待前两个任务结果流出(两操作数均输出)以后,才能返回输入端而进入流水线(中间结果返回输入端时,一般不计返回时间)。

解:流水线工作时空图如下(如图 5. 13 所示):

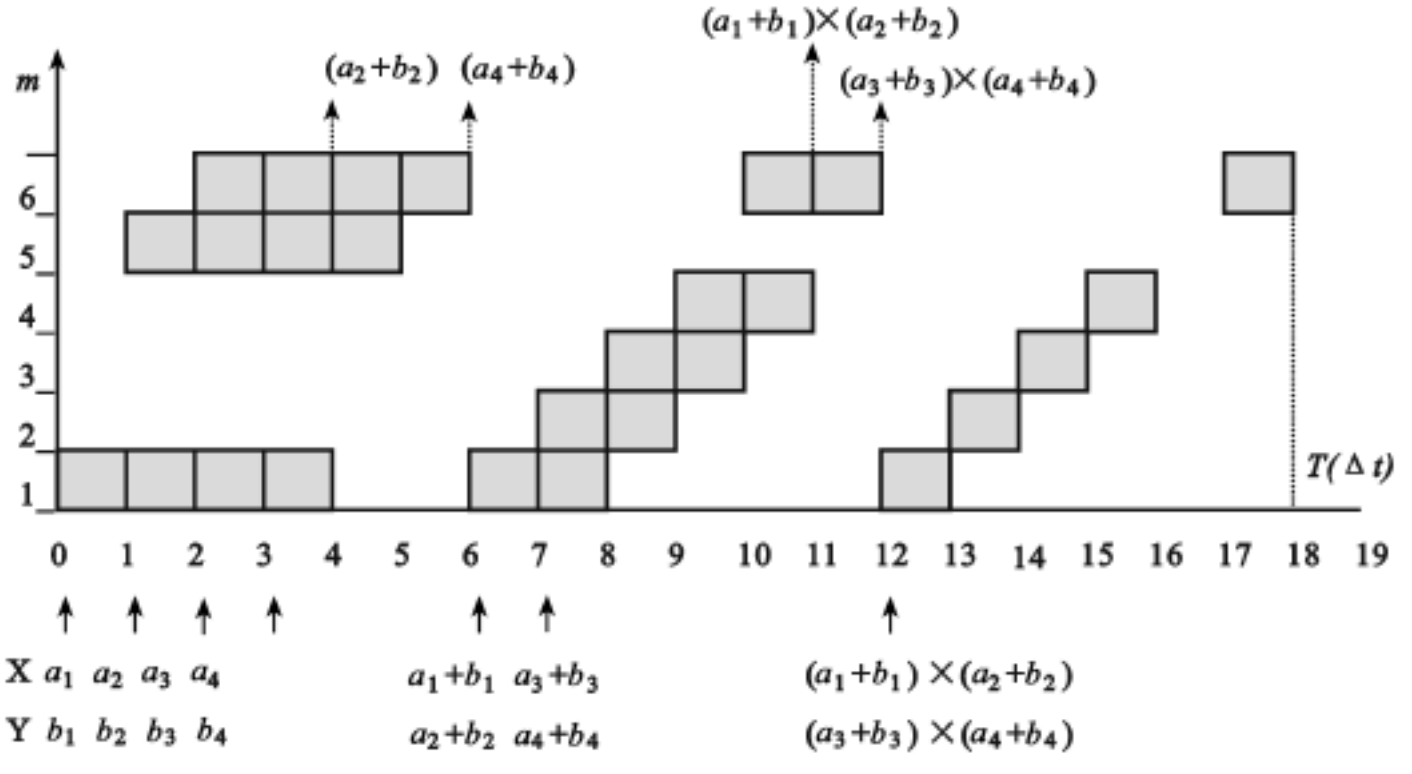


图 5 .13　双功能静态流水线工作时序图

由时空图可见,共在 17Δt 时间内输出 7 个任务(结果),因此该流水线的实际吞吐率为:

$$TP=\frac{7\text{ 个任务数}}{\text{完成 7 个任务的总时间}}=\frac{7}{17\Delta t}$$

$$E=\frac{\text{完成 7 个任务实际占用的时空区}}{6\text{ 功能段的总时空区}}=\frac{4\times3\Delta t+3\times5\Delta t}{6\times17\Delta t}\approx26\ \ 5\%$$

5 .2　标量流水中的障碍及控制

要使流水线具有较大的吞吐率、效率和加速比,必须设法使流水线能畅通流动,而不能发生断流。但在通常情况下,由于在流水过程中会出现以下的三种相关(冲突),使得流水线要实现不断流非常困难,这三种相关是资源或结构相关、数据相关和控制相关。

假定流水线由 5 个功能段组成,它们分别为取指令(IP)、指令译码(ID)、访存有效地址计算或指令执行(EX)、访存取数(MEM)以及将结果写回寄存器堆(WB)。

在执行各种不同指令时,各段所完成的操作如下:

当指令为 ALU 指令时,段 1 取出指令后,段 2 就可进行译码并由寄存器堆中读

出操作数,段 3 进行执行,段 4 不做任何操作,段 5 将运算结果写回寄存器堆。

当指令为 LOAD 或 STORE 指令时,段 1 取指令,段 2 进行译码及读寄存器堆,段 3 进行访存有效地址计算,段 4 进行访存,若为 LOAD 指令,则将自存储单元读出的操作数存入数据寄存器,若为 STORE 指令,则将数据寄存器内容写入相应的存储单元,段 5 将数据寄存器内容写入寄存器堆。

当指令为 BRANCH 时,段 1 取指令,段 2 进行译码及读寄存器堆,段 3 将生成转移目标地址,并形成比较条件码,段 4 判条件是否成立,若条件成立便将转移目标地址送往 PC,段 5 不进行操作。(参见表 5 .1)

所谓相关,是指在一个程序中的指令间存在某种关联,这种关联会影响到指令的重叠或流水方式的正常执行。按照相关影响范围的大小,可分为局部性相关和全局性相关两类。下面具体讨论这些相关对流水线工作的影响及处理方法。

5. 2. 1 局部性相关及处理

局部性相关是指相关只发生在相邻或相近的几条指令之间,影响的范围是‘局部性’的,它包括主存资源相关和寄存器数据相关。

1. 主存资源相关

主存资源相关是指,当有多条指令进入流水线后在同一机器周期内争用同一功能部件所发生的相关(冲突)。从图 5 .14 中可以看出,在时钟 4 时,第 i 条的 MEM 段和第 i + 3 条的 IF 段都要访问主存储器。通常,由于数据和指令存放在同一存储器中,且只有一个访问端口,这样便会发生这两条指令争用主存储器资源的相关冲突。

时钟 / 指令	1	2	3	4	5	6	7	8
LOAD 指令	IF	ID	EX	MEM	WB			
指令 i + 1		IF	ID	EX	MEM	WB		
指令 i + 2			IF	ID	EX	MEM	WB	
指令 i + 3				IF	ID	EX	MEM	WB
指令 i + 4					IF	ID	EX	MEM

图 5 .14 两条指令同时访问主存而引起的资源冲突相关

解决主存资源冲突的方法有:

① 将后续指令(i + 3)推迟一拍进入流水线,这样必然导致流水线的性能下降。

② 重复设置一个存储器,使指令和数据分别存放在不同的存储器中。如果不能

做到重复设置主存储器,则可以设置双 Cache 结构,即一个指令 Cache,一个数据 Cache。

③ 采用先行控制技术,或在处理器内部设置指令缓冲队列(指令缓冲栈,简称缓指)。

应该指出的是,图 5.14 中两条指令同时访存造成资源相关冲突是由于第 i 条为 LOAD 指令的缘故,若不是 LOAD 指令,则由于在 MEM 段不访存,就不会发生对存储器的争用。

2. 寄存器数据相关

这是由于流水线中的各条指令的重叠操作使得原来对操作数的访问顺序发生了变化,从而导致了数据相关的冲突。这往往发生在后继指令所需的操作数刚好是前一指令运算结果的情况下。例如,如果流水线要执行以下的两条指令:

ADD R_1, R_2, R_3 ; $(R_2)+(R_3)\rightarrow R_1$

SUB R_4, R_1, R_5 ; $(R_1)-(R_5)\rightarrow R_4$

两条指令在流水线中的执行情况见图 5.15。

从图 5.15 中可以看出,在时钟 5 时加法指令 ADD 方可将运算结果写回寄存器堆中的 R_1,但后继的减法指令 SUB 在时钟 3 时就要从 R_1 中读出数据。正常的读写顺序是,先由 ADD 指令写入 R_1,再由 SUB 指令来读 R_1。在非流水线的顺序执行时,这种写读的先后关系即先写后读,是自然维持的。在流水线中时,由于重叠操作缘故,使读写的先后关系发生了变化。如果不采取相应措施保持原来这种写读的先后关系,就会使操作的结果出错。

指令 \ 时钟	1	2	3	4	5	6
ADD	IF	ID	EX	MEM	WB	
SUB		IF	ID	EX	MEM	WB

图 5.15 两条指令发生了 R_1 的先写后读(RAW)数据相关

解决这种数据相关的方法:

① 推迟后续指令进入流水线。即遇到数据相关时,就停顿后继指令的运行,直至前面指令的结果已经生成。

② 采用定向技术,又称为旁路技术或相关专用通路技术,如图 5.16 所示。

需要说明的是,对于图 5.15 所示的两条指令,由于第一条 ADD 指令是在时钟 3 结束时才有运算结果,而第二条指令 SUB 也是在时钟 3 的一开始就需源操作数,这一内部定向传送是无法实现的。下面举出能够实现定向传送的 5 条指令解释过程,

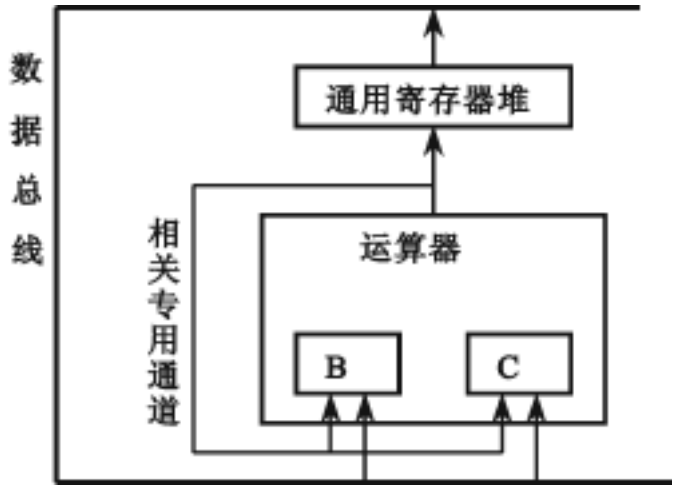


图 5 .16 采用定向技术解决寄存器数据相关

如图 5 .17 所示。

指令 \ 时钟	1	2	3	4	5	6	7	8
ADD R_1 , R_2 , R_3	IF	ID	EX	MEM	WB			
SUB R_4 , R_1 , R_5		IF	ID	EX	MEM	WB		
AND R_6 , R_1 , R_7			IF	ID	EX	MEM	WB	
OR R_8 , R_1 , R_9				IF	ID	EX	MEM	WB
XOR R_{10} , R_1 , R_{11}					IF	ID	EX	MEM

图 5 .17 定向传送 R_1 的值到 SUB, AND 和 OR 指令

由图 5 .17 可以看出,第一条指令 ADD 将向 R_1 寄存器写入操作结果,后继的 4 条指令中都要使用 R_1 中的值作为一个源操作数。显然,这样就出现了前述的 RAW 数据相关。在 WB 时钟段(时钟 5)时第一条指令才将结果写入 R_1 ,而后继的指令除 XOR 外,都需在此之前使用 R_1 中的内容。若采用后推法则会使流水线停顿 3 拍。实际上, ADD 的结果在时钟 3 的末尾处已经形成,如果设置专用通道将此产生的结果直接送往需要它的 SUB, AND 和 OR 指令的 EX 段,就可使流水线不发生停顿。但是,对于这种情况必须要对 3 条指令进行定向传输操作。为了减少这种定向传送操作的次数,可将 ID 段中的读寄存器操作安排在时钟的后半部分,而将 WB 中的写寄存器操作安排在前半部分,这样就可以将 OR 指令的定向传输操作取消了。

推迟后续指令的执行和采用定向传送技术是解决数据相关的两种基本方法。前者是以降低速度为代价,基本上不需增加设备;后者是以增加设备为代价,使指令解释速度不下降。可以看出,若相关的概率较低,就不宜采用定向技术,节省了设备,指

令解释效率也不会明显下降。

根据指令间的对同一寄存器读和写操作的先后次序关系,数据相关冲突可分为读与写(RAW)、写与读(WAR)和写与写(WAW)三种类型。例如,程序中有 i 和 j 两条指令,i 指令在前,j 指令在后,则三种不同类型的数据相关的含义为:

① 读与写(RAW)相关:若顺序指令 i(写)先于指令 j(读)对同一寄存器访问,由于异步流动可能使得指令 j 先于指令 i 之前执行。

例如:MUL R_1, R_2 ;(R_1) × (R_2) → R_1

ADD R_3, R_1 ;(R_1) + (R_3) → R_3

两条指令在寄存器 R_1 上出现了先写后读数据相关。

② 写与读(WAR)相关:若顺序指令 i(读)先于指令 j(写)对同一寄存器访问,由于异步流动可能使得指令 j 先于指令 i 之前执行。

例如:MUL R_1, R_2 ;(R_1) × (R_2) → R_1

MOV R_2, #00H ;0 → R_2

两条指令在寄存器 R_2 上出现了先写后读数据相关。

③ 写与写(WAW)相关:若顺序指令 i(写)先于指令 j(写)对同一寄存器访问,由于异步流动可能使得指令 j 先于指令 i 之前执行。

例如:MUL R_1, R_2 ;(R_1) × (R_2) → R_1

MOV R_1, #00H ;0 → R_1

两条指令之间出现了寄存器 R_1 上的写与写相关。

上述的三种数据相关,在按序流动的流水线中,只可能出现 RAW 相关。解决这种相关,通常采用上述内部定向传送方法来解决。在异序(乱序)流动流水线中,则由于允许后进入流水线的指令超过先进入流水线的指令而先流出流水线,那么既可能发生 RAW 相关,还可能发生 WAR 和 WAW 相关。

5.2.2 全局性相关及处理

全局性相关是指进入流水线的转移指令(尤其是条件转移指令)与其后续指令之间存在相关。统计表明,转移指令占总指令的 1/4 左右。因此,与数据相关相比,它会使流水线丧失更多的性能。当转移发生时,将使流水线的连续流动受到破坏。当执行转移指令时,依据是否发生转移,它可能将程序计数器 PC 内容改变成转移目标地址,也可能只是使 PC 加上一个增量,指向下一条指令的地址。由于通常要在 MEM 段末尾才会使 PC 内容发生改变,如图 5.18 所示。这样就要使流水线停顿 3 个节拍,直至 PC 中生成新的地址后才可能取出下一条指令。

由转移指令所造成的流水线性能下降程度,可通过下面的例子来加以说明。

假定所有执行指令中有 25%的指令为转移指令,进一步假设其中有 2/3 是会发生转移的,由于这些转移发生的指令,使得完成一条指令平均需要:

$$0.75 \times 1 + 0.25 \times (1/3 \times 1 + (2/3) \times (3 + 1)) = 1.5 \text{ 周期}$$

这将使流水线的性能降低 33%。

BRANCH 指令	IF	ID	EX	MEM	WB					
指令 i+1		停顿	停顿	停顿	IF	ID	EX	MEM	WB	
指令 i+2			停顿	停顿	停顿	IF	ID	EX	MEM	WB
指令 i+3				停顿	停顿	停顿	IF	ID	EX	MEM	WB

图 5.18 转移指令引起相关时使流水线产生停顿

为了减少因转移指令而引起的流水线性能下降,可采用一些如下的方法进行处理。

1. 猜测法

对于条件转移指令,其执行情况必存在两个分支,即转移成功和转移不成功分支。对于流水方式执行的指令,可采用猜测法进行处理。

设程序中第 i 条指令为条件转移指令,其一个分支是 i + 1, i + 2, …,按原来顺序执行,为转移不成功分支。另一个分支是 p, p + 1, …,为转移成功分支。如图 5.19 所示。

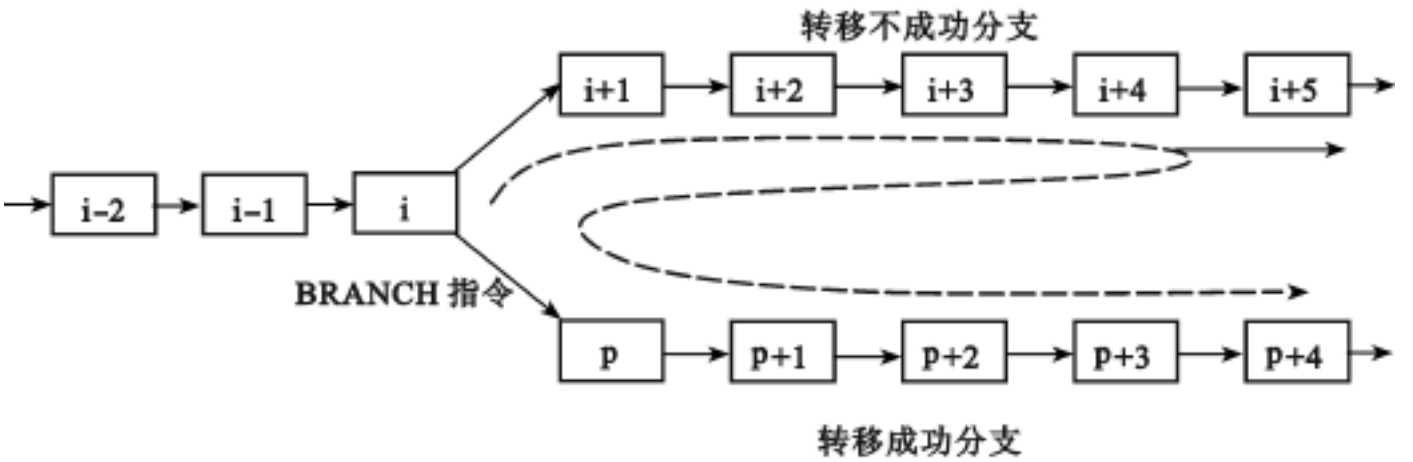


图 5.19 用猜测法处理条件转移指令

使用猜测法处理条件转移指令应注意以下几个问题:

(1)分支的选择

对于条件转移指令,猜选哪个分支好呢?如果两个分支概率相近时,宜选 i + 1 不成功转移分支。这是因为它已预取进(指令缓冲器),可以很快地从中取出进入流水线而不必等待。如果猜选成功转移分支,指令 p 很可能不在(指令缓冲器)中,需花较长时间访存去取,使流水线实际上断流。IBM 360/91 猜选的就是转移不成功分支。

现假设指令 i 所用条件码是在 i + 4 流入流水线时才建立的,若条件码是对应于

转移不成功分支就猜对了,可继续流下去;若条件码是对应于转移成功分支就猜错了,这时需对 i+1,i+2,i+3,i+4 已有的解释作废,重新回到原分支点,沿转移成功分支去解释 p,p+1,…,使流水线的吞吐率和效率都下降。但是,只要猜测法猜对的机会占大多数,流水线的吞吐率和效率就会比不用猜测法要高得多。

(2)提高猜测准确率

当转移的两个分支概率不均等时,宜猜高概率分支。那么转移概率的值如何确定呢?可采用静态及动态两种方法。静态方法不考虑转移历史,而动态方法考虑了转移历史,因而有较高的准确率。一种具有较高猜准率的动态方法是考虑以前两次转移的历史,如图 5.20 所示的那样,图中每个转移状态用 2 位二进制位表示。11 和 10 表示转移发生,00 和 01 表示转移不发生。从一个状态变到另一个状态时,若为不转移,用 0 表示;若为转移,用 1 表示。在水平方向发生状态变化时,将使低位状态位发生变化,如从 11 变为 10 或从 l0 变为 11;而在垂直方向发生状态变化时,将使高位状态位发生变化,如从 10 变为 00,或从 01 变为 11。这种方法,仅当两次连续猜错时,预测状态才会发生改变。如从 11 状态变为 10,再变为 00。这种方法猜准率经在 RISC 机上测试可高达 83%。

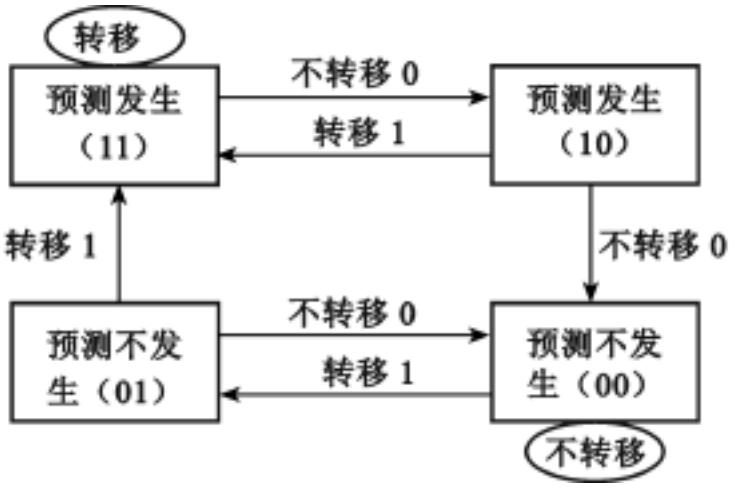


图 5.20 转移预测状态图

(3)现场恢复

采用猜测法时应能保证猜错时可恢复分支点处原先的现场,以便流水线的正常执行。处理此问题一般有三种方法。第一种方法是使计算机沿猜测分支解释指令时,应当与正常情况下的指令解释不同。例如,假设转移指令之前的某条指令是 $(R_i)+(N)\rightarrow N$,即对某存储单元 N 写入操作结果,当沿着不成功分支继续流动的 i+4 之前的某指令修改了 N 单元的内容后,流水线又返回转移成功分支执行,这时,从 N 单元读出的便是修改后的错误值。在 IBM 360/91 中采取对指令只译码和准备好操作数,在转移条件码出现之前不进行运算。第二种方法是让它运算完但不送回运算结果,有的计算机就是如此。以上介绍的是早期计算机所用的这两种方法,使用

起来不是很方便,因为若猜对后还要让这些指令继续完成余留的操作。第三种方法是采用后援寄存器,把可能被破坏的原始状态都用后援寄存器保存起来,一旦猜错,就取出后援寄存器的内容来恢复分支点的现场。这些后援寄存器实际上不是单独为流水线设置的,因为为提高系统可靠性,实现指令复执、程序卷回,本来就已设置了这些后援寄存器。一般猜对的概率要高,猜对后既不用恢复,也不用再花时间去完成余留的操作。因此,采用后援寄存器法的实现效率会更高一些。

2. 加快和提前形成条件码

提前形成条件码,以便提前知道程序流向哪个分支,会有利于流水计算机简化对条件转移的处理,这可以从两方面采取措施。

一方面是加快单条指令内部条件码的形成。例如,乘、除结果是正、负或零的条件码可在运算前形成。只要两个操作数符号位相同就是正,符号位相反就为负。相乘时有一个操作数为零或除法时被除数为零,则结果为零。由于相乘、相除操作时间较长,条件码提前形成对加速条件转移的处理大有好处。Amdahl 470V/ 6 就是在具体运算前就能将运算结果的条件码送到指令分析部件。

另一方面是在一段程序内提前形成条件码,这特别适合于循环型程序在判断循环是否继续时的转移情况。例如 FORTRAN 语言中的 DO 循环,每当执行到循环终端语句时,总要对循环次数减 1,如果结果为 0 就跳出循环,否则转回去继续执行循环体,这通常用减 1(DEC)和等于零条件转移(BE)两条指令来实现。为了使等于零条件转移指令(BE)的条件码能提前形成,可以将减 1 指令(DEC)提前到与其不相关的其他指令之前,甚至提前到循环体开始时进行。这样,执行到 BE 指令时,减 1 指令的条件码早已形成,马上就能知道是否需要转移,不至于因等待条件码形成而使流水线的吞吐率和效率下降。

3. 加快短循环程序的处理

加快短循环程序的处理是将长度小于指令缓冲器容量的短循环程序整个一次性地放入指令缓冲器内,并暂停预取指令。这样有两个好处:一是暂停预取指令后,避免执行循环时由于指令预取导致指令缓冲器中需循环执行的指令被冲掉,减少了访主存重复取指的次数;二是由于循环分支概率高,因此,让循环出口端的条件转移指令恒猜循环分支,减少因条件分支造成流水线断流的机会。例如,在 IBM 360/ 91 中设置了“向后 8 条”检查,即转向去址往回走且与条件转移指令之间相隔不超过 8 条时,将其间的指令全部移入指令缓冲器并停止预取新指令。并为上述第二点设置了“循环方式”工作状态。采取这些措施后可使循环时流水加快 1/ 3～3/ 4。

有的计算机还采取在顺序执行时,让预取的指令既放入正常使用的指令缓冲器,也放入转移目标指令缓冲器中。一旦检测出是循环时,可把转移目标指令缓冲器的内容作为短循环程序控制用,省去了第一次循环时,重新从主存中取此短循环程序中

指令的操作开销。还有的机器允许将这两种指令缓冲器连接起来使用,使更大的循环程序也能得到加快处理。

4. 采用延迟转移技术

延迟转移技术是一种有效的软件手段,以减少由于控制相关而造成的流水线性能下降。实现时不必增加硬件,在编译生成目标指令程序时,将转移指令与其前面不相关的一条或多条指令交换位置,让成功转移总是延迟到这一条或多条指令之后再执行。这样可使转移造成流水性能损失减少到 0。如图 5 .21 所示,是将转移指令之前的一条与转移指令不相关的加法指令调入到延迟槽中。

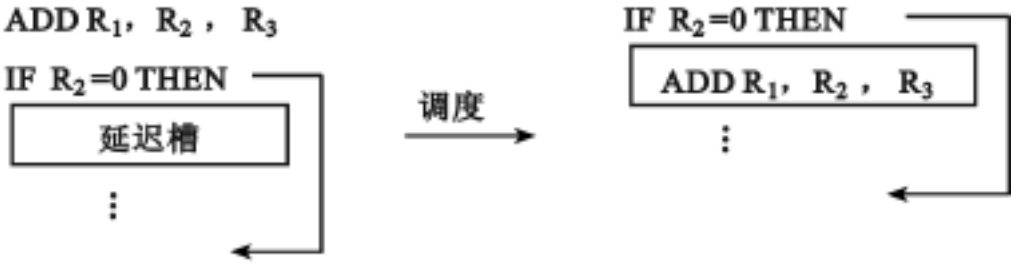


图 5 .21　转移指令延迟槽的调度方法之一

5. 2. 3　流水线的中断及处理

中断会引起流水线断流,但出现概率比条件转移的概率要低得多,且又是随机发生的。所以,流水计算机处理中断主要是如何处理好断点现场的保存和恢复,而不是如何缩短流水线的断流时间。

假若在执行指令 i 时有中断发生,断点本应是在指令 i 执行结束,指令 i + 1 尚未开始执行的地方,但流水机器是同时解释多条指令,指令 i + 1,i + 2,…,可能已进入流水线被部分解释。对于异步流动流水线,这些指令中有些可能已流到指令 i 的前面去了。

流水线中对于中断处理的方法有不精通断点和精通断点法。

所谓不精通断点法处理,是指不论指令 i 在流水线的哪一段发生中断,未进入流水线的后续指令不再进入,已在流水线的指令仍继续流完,然后才转入中断处理程序。这样,断点就不一定是 i,可能是 i + 1 或 i + 2,i + 3,…,即断点是不精确的。仅当指令 i 在第 1 段响应中断时,断点才是精确的。早期的流水机器,如 IBM 360/ 91,为简化中断处理采用不精确断点法。不精确断点法不利于编程和程序的排错。

所谓精通断点法处理,是指不论指令 i 是在流水线中哪一段响应中断,给中断处理程序的现场全都是对应 i 的,i 之后流入流水线的指令的原有现场都能恢复。精确断点法需设置很多后援寄存器,以保证流水线内各条指令的原有现场都能保存和恢复。后来的流水机器多数采用精确断点法,如 Amdahl 470V/ 6。

5.3 流水线的调度技术

本节首先介绍非线性流水线的调度方法(策略),这种方法主要是借助软件对指令执行的顺序进行调度,以减少流水线中存在功能段冲突而引起的流水线的停顿时间。这种方法也称为静态调度。然后介绍流水线的更高级调度方法,即采用硬件重新安排指令的执行顺序,以减少流水线的停顿时间。

5.3.1 非线性流水线的静态调度技术

在线性流水线中,每个任务执行时仅一次通过各流水段。当流水线中各功能段的执行时间都相等时,每拍(单位时间)都可流入一个新的任务,它们不会争用同一个流水段。但对非线性流水线来讲,由于每个功能段之间存在有前馈或反馈通路,因此一个任务在执行的过程中,可能会多次通过同一流水段,如果仍要向流水线每隔一拍送入一个新任务,就会发生几个任务同时争用同一流水段的现象,这就是功能段的使用冲突。

为了避免这种功能段冲突,必须解决以下两个问题:一是当前一个任务流入流水线后,后继任务需要间隔多少拍进入流水线,才不会引起功能部件冲突。二是由于间隔拍数可能有多个方案,如何确定最佳的送入新任务的间隔拍数,以使流水线有较高的吞吐率和效率。这就需要对流水线作适当的调度。

流水线调度采用了 1971 年 Davidson 等人提出的二维预约表进行。

若一个非线性、单功能流水线,由 S 段组成,每个任务流过流水线需 N 个时钟周期(t_1,t_2,…,t_n),以段为纵坐标,时间为横坐标,就可画出此流水线的预约表,如图 5.22所示。图中每一行代表一个段,每一列表示相应的时钟周期。若某个任务在周期 t_i 需要使用 S_i 段进行处理,则在行 S_i 和列 t_i 的相交处以“×”表示。

S \ t	1	2	3	4	5	6	7	8	9
1	√								√
2		√	√					√	
3				√					
4					√	√			
5							√	√	

图 5.22 一个 5 功能段非线性流水线预约表

根据预约表可较容易地推算出一个任务执行时,各段所需的间隔周期拍数及最

佳调度方案(策略)。下面以图 5.22 预约表为例,介绍其分析过程:

1. 根据预约表写出禁止表 F

对于功能段 1,两任务禁止间隔的拍数为 8(这只需要将同一行中相互间隔的“√”处所对应的拍数值相减即可);对于功能段 2,两任务禁止间隔的拍数为 1,5,6;对于功能段 3,两任务禁止间隔的拍数≥8;对于功能段 4,两任务禁止间隔的拍数为 1;对于功能段 5,两任务禁止间隔的拍数为 1。

将各段上所有的这种间隔汇集起来,就构成一个禁止表 F = {1,5,6,8}。

禁止表说明,要想使流入的任务不发生同时争用功能段的情况,相邻两个任务进入流水线的间隔拍数就一定不能为 1,5,6,8 拍。

2. 根据禁止表 F 写出冲突向量 C

冲突向量是用二进制数表示的,二进制的位数为禁止间隔的最大拍数。本例为 8,所以冲突向量的位数为 8 位二进制 C = (C_7 C_6 … C_1 C_0)。另外,冲突向量 C = (C_7 C_6 … C_1 C_0)的定义为:C_i = 0 表示允许间隔 i 拍流入后续任务;C_i = 1 表示禁止间隔 i 拍流入后续任务。根据上述预约表所得到的禁止表 F,可形成冲突向量 C = (10110001),称为原始冲突向量。

3. 根据原始冲突向量 C 画出状态转移图

状态转移图的求解方法是,由当前的冲突向量 C_i 求出下一冲突向量 C_j。其算符表达式为:$C_j = SHR^{(k)}(C_i) \vee C$,式中 C 为原始冲突向量,$SHR^{(k)}(C_i)$ 表示将当前冲突向量 C_i 右移 k 位,且高位补零。

例如,在本例中由 C 求 C_1:将 C = (10110001) 右移 2 位(由 C 可知第二个任务与第一个任务间隔 2 拍时可以进入),高位补 0,并得到 C_1 = (00101100) ∨ (10110001) = (10111101)。由 C_1 可知,第三个任务与第二个任务之间可以间隔的拍数分别为 2 和 7,因此,按上述方法,将 C_1 右移 2 位,高位补 0,并得到 C_2 = (00101111) ∨ (10110001) = (10111111)。由 C_2 可知,后继任务与第三个任务之间只能间隔 7 拍,将 C_2 右移 7 位,高位补 0,再与 C 做逻辑加后,得到的向量与原始向量 C 相同。

到此为止,我们仅取 C 间隔 2 拍时的分析情况。还需完成 C 分别取间隔 3,4 和 7 拍时,状态图的转移情况,分析方法与上述方法完全相同,其状态转移图如图 5.23 所示。

4. 根据状态转移图写出调度策略

状态转移图中的任何一个闭合回路即为一个调度策略,如(2,7),表示第二个与第一个间隔 2 拍,第三个与第二个间隔 7 拍,第四个与第三个间隔 2 拍,第五个与第四个间隔 7 拍,…,如此循环调度指令,便不会发生功能段冲突。还有可能的调度策

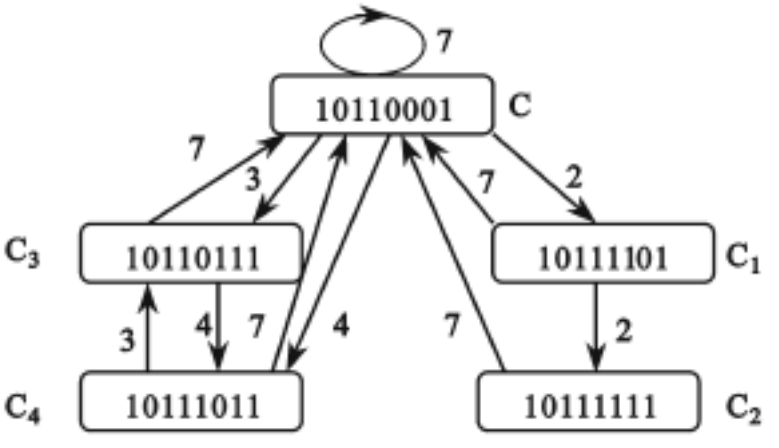


图 5 .23　非线性流水线的状态转移图

略以及它们的平均延迟拍数如表 5 .2 所示。

最后,根据列出的所有调度方案和平均延迟拍数,找出平均延迟最小的最佳调度策略。由表中可见,采用先隔 3 拍,再隔 4 拍轮流往流水线输入任务的调度法为最佳。因为,此时平均每隔 3 .5 拍即可流入一个任务,从而达到最高吞吐率。当然这是一种不等间隔的调度方案,相应的控制要复杂些。为了简化控制也可采用等间隔调度,本例中只有一种,即每隔 7 拍输入一个任务,此时吞吐率就比最佳的调度方案降低了 1/ 2。

表 5 .2　　各种调度策略及平均延迟拍数

调 度 策 略	平 均 延 迟 拍 数
(2,7)	4 .5
(2,2,7)	3 .67
(3,7)	5
(3,4)	3 .5
(3,4,3,7)	4 .25
(3,4,7)	4 .67
(4,3,7)	4 .67
(4,7)	5 .5
(7)	7

5.3.2　流水线的动态调度技术

流水线的动态调度具有如下的优点:

- 能处理某些在编译时无法知道的相关情况;
- 能简化编译程序设计;

- 使代码有可移植性。

这种方法的主要缺点是相应的硬件较为复杂。

1. 流水线的集中式动态调度

前面讨论的流水线技术在应用时必须要按序启动指令(In-order Instruction Issue),如果一条指令在流水线中发生了停顿,后继指令就不再前进,当功能部件较多时,就会使功能部件出现闲置的情况。要改善这一情况,就应允许流水线中能按无序(Out-of-Order)方式工作。

按这种无序方式工作,从前面讨论中已知将会引起更多的相关冲突。解决这一问题可采用集中式动态调度的方法。这种方法要依靠硬件在程序运行过程中对可能出现的相关情况加以检测,从而可保证流水线中的各个功能部件能最大限度地重叠工作。图 5 .24 中示出了这种集中式动态调度方式的框图。

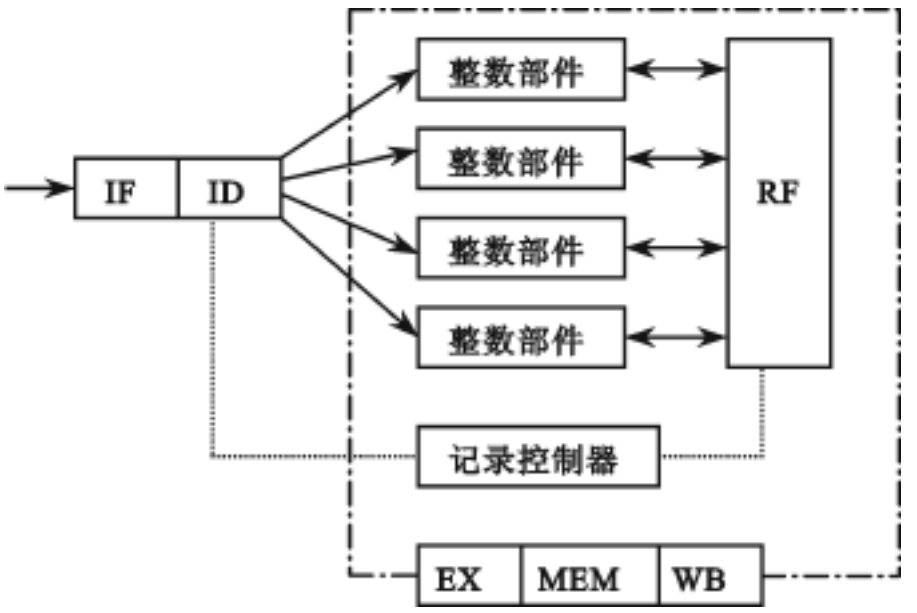


图 5 .24　集中式动态调度

这种动态调度方法主要用一个称为状态记录控制器(或记分牌)的调度部件对流水线中的各个功能部件的工作状态、进入流水线中的各条指令的工作状态、它们所使用的源寄存器和目的寄存器情况等进行集中的统一记录和调度。

在译码阶段,记录控制器根据所记录的状态决定是否将译码后的指令发送给有关功能部件进行处理。其中,主要检查该指令要使用的功能部件是否已被流水线中的其他指令占用,即检查是否有资源使用冲突。该指令的源操作数寄存器是否为其他指令的目的寄存器,或者它所要写入的目的寄存器又正好是前面其他指令所要读出的操作数,或是要写入的目的寄存器,即是检查是否有 RAW,WAR 和 WAW 的数据相关。当发现有任何一种冲突时,便不再启动此指令而将它挂起。经检查没有发现任何冲突的指令便送到相应部件去执行,由执行而引起的状态变化将被登录到记

录控制器中。此时,再由后者来判别以前已挂起的指令是否可启动等。这种集中式动态调度方法,早在 20 世纪 60 年代的 CDC 6600 计算机中采用。现在的 RISC 机中的超级标量机通常也采用与此类似的方法。

2. 流水线的分布式动态调度

在流水机 IBM 360/ 91 中,采用了另一种动态调度方法,即分布式方法。此调度方法是由日本学者 TOMASULO 在 1967 年提出的,又称为 TOMASULO 调度法。

它采用公共数据总线 CDB 来实现某些相关专用通路连接,并通过给每个浮点数寄存器 FLR 设置一个忙标志来判别指令间所用的数据是否发生数据相关,只要某些 FLR 正在使用,就将忙位置 l 表示存在数据相关,一旦使用完成就置成 0。

图 5 .25 中示出了 IBM 360/ 91 的浮点运算部分,它包括了以下主要部件:

① 运算部件。一个加法部件和一个乘除部件。

② 保存站。加法部件中有 A_1 ～ A_3 三个保存站,乘除部件有 M_1 和 M_2 两个保存站,用来保存当前参加运算的数据。当两个源操作数都到齐且运算部件空闲时,便可进行运算。这些保存站都有称为站号的地址号。它与 FLB 浮点操作数缓冲寄存器统一编址,$FLB_{1\sim6}$ 的编号为 0001～0110, M_1 和 M_2 编号为 1000 和 1001, A_1 ～ A_3 编号为 1010～l100。

③ 指令操作缓冲栈。存放经分析后由指令部件送来的浮点操作指令,译码后,产生相应的控制信号到达各个部件。

④ 浮点操作数寄存器(FLB)。存放由主存预取来的操作数,它作为源操作数使用。

⑤ 浮点数寄存器(FLR)。存放操作数寄存器,运算时,作为另一源操作数,同时也是目的操作数寄存器,即通常操作时有(FLR) + (FLB)→FLR,故常会发生数据相关。

因此,在该浮点数寄存器中,给每个寄存器附加一忙位,当它为 1 时,表明该数据已作为操作数使用,另外还有一个标记(即站号)表示数据由何处送来。FLR 由 F_0 ～ F_7 组成。

⑥ 存储数据缓冲站(SDB)。用于暂时存放将要写入主存储器的结果数据。也有站号,但单独编号。

⑦ 公用数据总线(CDB)。以上各部件间的连接总线,有 11 个部件可向 CDB 提供信息。它们是 6 个 FLB,3 个加法部件保存站和 2 个乘除部件保存站,而 CDB 则向加法部件和乘除部件的保存站、FLR 浮点寄存器以及 SDB 存储缓冲站提供数据。

下面通过一个例子来说明这些部件的使用,以及在运算过程中如何解决数据相关问题。例如,有如下的一串指令:

S_1 : LD F_0 , FLB_1 ; $(FLB_1)\rightarrow F_0$

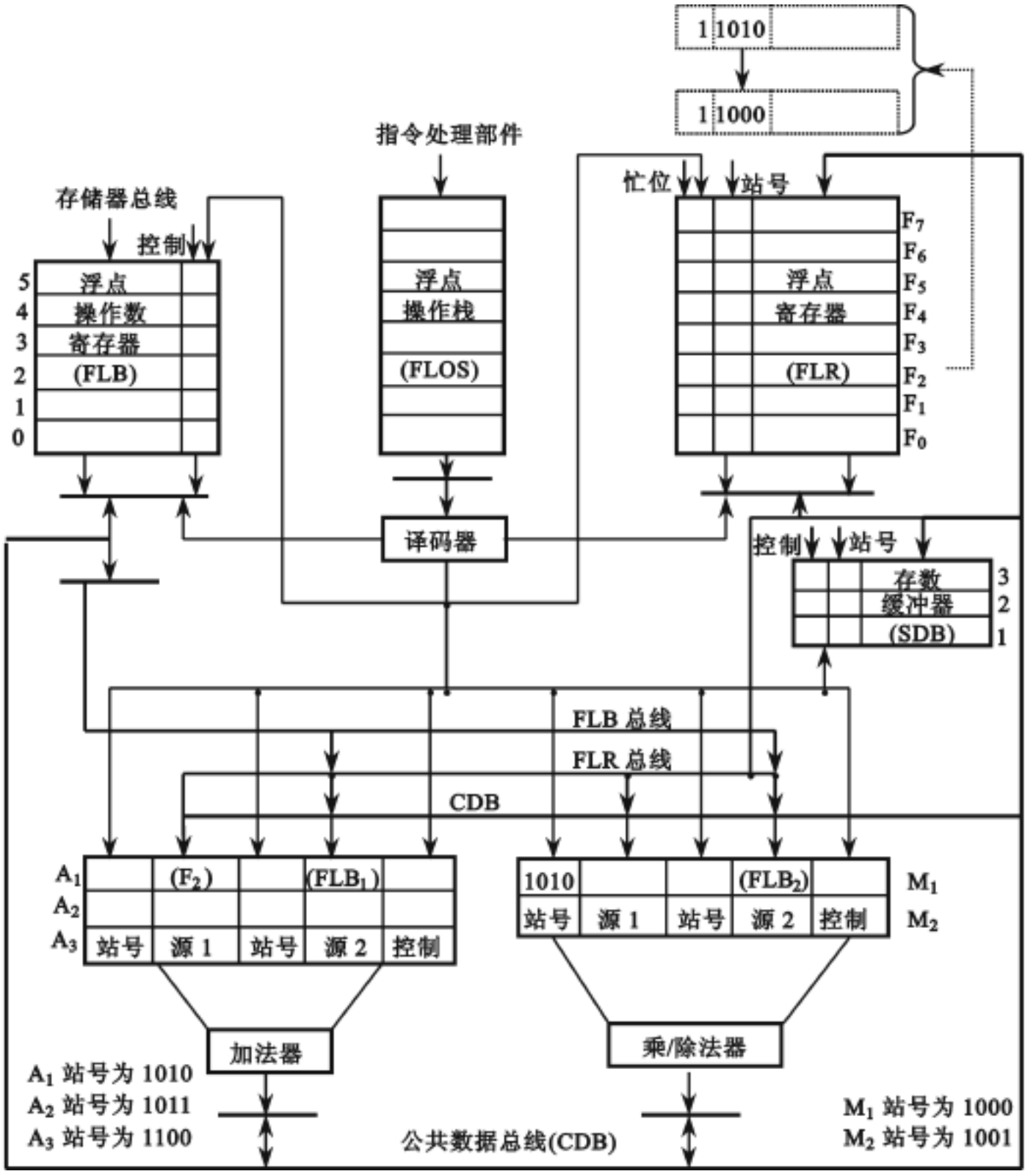


图 5 .25　IBM 360/ 91 的浮点运算部件结构框图

S_2 : MD　F_0 , FLB_2　　; $(F_0) \times (FLB_2) \rightarrow F_0$

S_3 : STD　F_0 , A　　; $(F_0) \rightarrow A$

S_4 : LD　F_0 , FLB_3　　; $(FLB_3) \rightarrow F_0$

S_5 : ADD　F_0 , FLB_4　　; $(F_0) + (FLB_4) \rightarrow F_0$

可以看到,指令间存在相关性。

S_1 译码后,将 F_0 的站号置成 0001,向 CDB 表明由 FLB_1 读出的内容应送入 F_0

中。S_2 译码后,将 F_0 的忙位置 1,表明乘法指令 MD 将使用 F_0 中的内容。同时,将 M_1 中的源 1 寄存器站号置为 0001,表明需由 FLB_1 中得到源 1 操作数,将 M_1 中的源 2 寄存器站号置为 0010,表明 FLB_2 的输出应送到这里作源 2 操作数。此外,还需将 F_0 的站号由刚才设置的 0001 改为 1000,以便站号为 1000 的 M_1 在得到乘积后经 CDB 送回 F_0。一旦结果送到 F_0 后,便将 F_0 的忙位置 0,以使其他指令可使用 F_0 中的值。S_3 译码后,将存数缓冲器 C_1 的站号置为 1000,表明由乘法产生的乘积应存放到写数缓冲器 C_1 中。S_4 译码后,将 F_0 的站号改为 0011,类似于 S_1 的动作。S_5 译码后,将 A_1 源 2 寄存器站号置成 0100,表明加法指令 ADD 将由 FLB_4 处得到源 2 操作数,将 A_1 源 1 寄存器站号置成 0011,表明由 FLB_3 处得到源 1 操作数,并将 F_0 的站号由 0011 改为 1010,表明由加法器保存站 A_1 处得到的加法结果应送回到 F_0 中。

这种调度方法的特点是:

① 由于为加法器和乘法器部件分别设置了 3 个和 2 个保存站,从而减少了资源使用冲突的机会,仅当这些保存站都处于忙碌状态时,才有可能发生资源使用冲突。

② 调度算法使用保存站,通过对寄存器重新命名(改写站号)自然地消除了 WAR 和 WAW 数据相关可能性。

③ 通过对 FLR 寄存器忙位状态的判别,来检测是否存在 RAW 数据相关。

④ 借助 CDB 公共数据总线作为专用相关通路,将有关数据直接送往所有需要它的功能部件,而不必先写入寄存器,然后再从此寄存器读出。

这种调度方法是借助 FLR 中的各寄存器的忙标志是否为 1 来判别指令间是否存在 RAW 数据相关。借助 CDB 公共数据总线作为相关专用通路,由于这种流水调度是通过分布在各个部件中的站号、忙标志以及 CDB 总线实现的,称为分布式调度。这种方式简化了同时出现的多个相关及多重相关处理,因此,比集中式的调度更加灵活。

3. 动态硬件预测转移方法

前面提及的延迟转移方法是在编译阶段完成的,因此,这是一种面向软件的静态方法。这里介绍的方法是在程序运行时,借助硬件来动态地预测转移方向。实际上,这是一种尽早生成转移目标地址的方法,它将过去发生过的转移指令地址以及它的转移目标地址存入一个由类似 Cache、称为 BTB(Branch Target Buffer)的转移目标缓冲器中。其中,转移指令地址作为标志,供检测用,它的工作原理如下:

将欲取出指令的 PC 值与 BTB 中的所有标志作相联比较,若有相符的标志时,便将该项中相应的预测转移目标地址读出,送到 PC 中。当转移条件成立时,便可确认其为有效,马上取转移目标地址处的指令;否则,便取消送到 PC 中的转移目标地址,并对 BTB 中的内容做相应更新。这是一种使用硬件支持来加快生成转移目标地址的方法,在 Intel Pentium 处理器中已被采用。

5.4 先进的流水技术

一般的标量流水处理机在每个时钟周期最快只能完成(输出)一条指令,其并行度 ILP 小于或等于 1。假设指令流水线的功能段数 k = 4,即它把一条指令的解释过程分解为取指令、分析、执行和写回运算结果 4 个阶段。指令所要执行的功能主要在多功能操作部件中,在执行这一功能段完成。多数流水线处理机的多功能操作部件采用流水线结构。有的简单指令,只要 1 个时钟周期就能够在执行功能段中完成,而比较复杂的指令往往需要多个时钟周期才能够做完。

本节介绍的几种先进的流水技术是:超标量处理机(Superscalar Processor)、超流水处理机(Superpipelinig Processor)、超标量超流水处理机(Superpipelinig Superscalar Processor)和超长指令字(Very Long Instruction Word)。它们的指令并行度 ILP 大于 1。

5.4.1 超标量流水线技术

1. 超标量流水原理

现设有 12 条无任何相关指令,要在 4 功能段(其中,段 1 为 IF,段 2 为 ID,段 3 为 EX,段 4 为 WB,)流水的一般标量处理机上执行,共需 15 个时钟周期。在每个时钟周期只能发射一条指令,如图 5.26 所示。

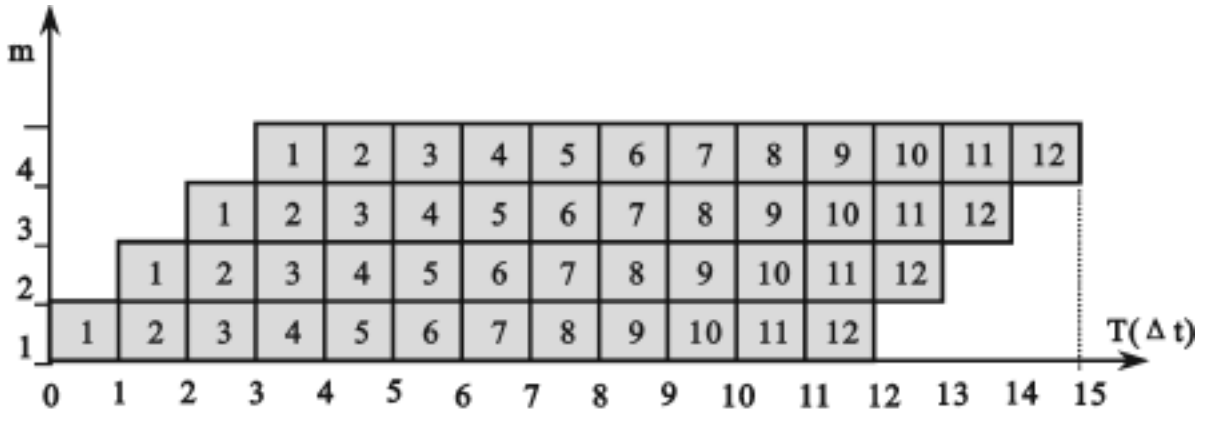


图 5.26 12 条指令在一般标量处理机上完成时空图

超标量流水处理机在每个时钟周期可同时发射两条或两条以上的指令。在这种处理机中设置有多套功能部件。如图 5.27 中,在 1 个时钟周期平均发射 3 条指令,其并行度 ILP = 3。完成 12 条无相关指令只需 7 个时钟周期。

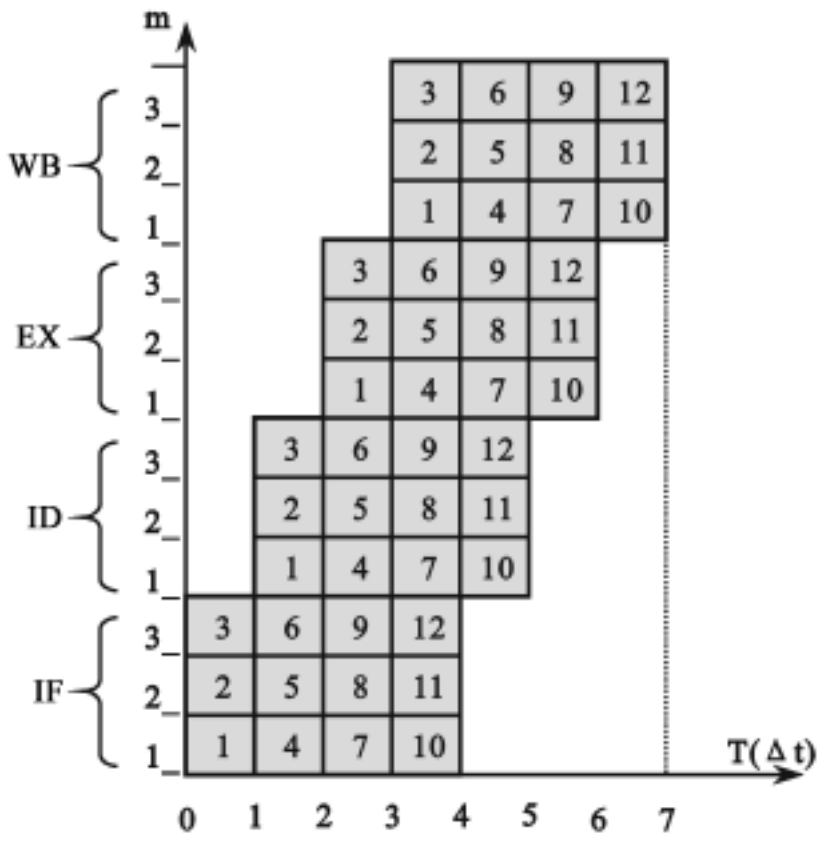


图 5 .27　12 条指令在超标量处理机上完成时空图

2. **超标量流水线处理机结构**

在有些超标量流水线(简称超标量处理机)处理机中,操作部件的个数要多于每个周期发射的指令条数。例如,在许多每个时钟周期发射两条指令的超标量处理机中,通常有 4 个或 4 个以上独立的操作部件,在有的超标量处理机中有 16 个独立的操作部件。与单发射处理机相同,多发射处理机的操作部件可以采用流水线结构,也可以不采用流水线结构,每个操作部件的时间延迟可以多于一个时钟周期。超标量处理机的一般结构如图 5 .28 所示。

在超标量处理机中,不仅需要设置多套取指令部件和指令译码部件,而且要判断指令之间确无功能部件冲突,有无数据相关和由于条件转移引起的控制相关等。另外,还要通过一套交叉开关把几个指令译码器的输出送到多个操作部件中执行。因此,超标量处理机的控制逻辑是比较复杂的。

目前,在多数超标量处理机中,每个时钟周期发射 2 条指令,通常不超过 4 条。由于存在有数据相关和条件转移等问题,采用一般的指令调度技术,理论上的最佳情况是每个时钟周期发射 3 条指令。对大量程序的模拟统计结果也表明,每个时钟周期发射 2～4 条指令比较合理。例如, Intel 公司的 i860, i960, Pentium 处理机, Motolora 公司的 MC88110 处理机, IBM 公司的 Power 6000 处理机等每个时钟周期都

发射 2 条指令;美国德州仪器公司(TI)为 SUN 公司生产 SuperSPARC 处理机每个时钟周期发射 1 条指令。

超标量处理机每个时针周期可以平均执行完成多条指令,因此,它的指令级并行度 ILP 一般都大于 1。如果一台超标量处理机每个时钟周期发射 m 条指令,则它的指令级并行度 ILP 的期望值就为 m。但是,由于数据相关、条件转移和资源冲突等原因,实际的 ILP 不可能达到 m,只会小于 m。通常有:1 < ILP < m。

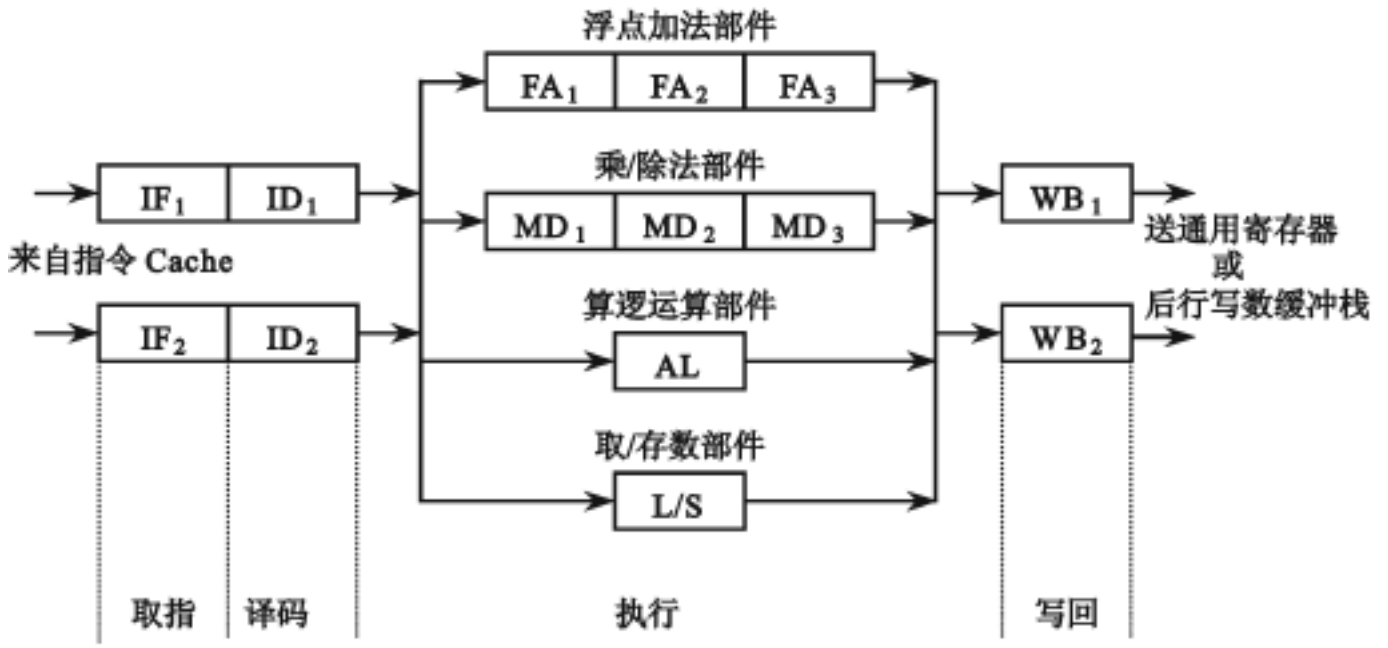


图 5.28　超标量流水线处理机结构

3. 超标量流水调度方法

在超标量处理机中,有多条指令流水线在同时工作,设置有多个能够独立工作的操作部件,因此,必须解决多流水线的调度问题和操作部件的资源冲突问题。而多条流水线的调度问题非常复杂。已经证明,多流水线实现优化调度所需要的代价很大,包括硬件代价和软件代价,通常需要软件(主要是编译器)和硬件的共同结合才能获得比较好的调度效果。

在有多条流水线同时工作时,指令的发射顺序和完成顺序对提高超标量处理机的性能非常重要。如果指令的发射顺序是按照程序中的指令排列顺序进行的,称为顺序发射(In-order Issue);否则,称为乱序发射(Out-order Issue)。同样,如果指令的完成顺序必须按照程序中的指令排列顺序进行,称为顺序完成(In-order Completion);否则,称为乱序完成(Out-order Completion)。

根据多流水线中指令发射顺序和完成顺序的不同组合,多流水线的调度主要有三种方法,即顺序发射顺序完成,顺序发射乱序完成和乱序发射乱序完成。下面通过一个具体的程序例子来介绍这三种方法。程序如下:

I_1 : LOAD R_1, A ; 主存单元 A→R_1

I_2 : FADD R_2, R_1 ; (R_1) + (R_2)→R_2

I_3 : FMUL R_3, R_4 ; (R_3)×(R_4)→R_3

I_4 : FADD R_4, R_5 ; (R_4) + (R_5)→R_4

I_5 : DEC R_6 ; (R_6) - 1→R_6

I_6 : FMUL R_6, R_7 ; (R_6)×(R_7)→R_6

在这个由 6 条指令组成的程序中,指令 I_1 和指令 I_2 之间有先写后读数据相关,指令 I_3 和指令 I_4 之间有先读后写数据相关,而指令 I_5 和指令 I_6 之间除了有先写后读数据相关之外,还有写—写数据相关。另外,在指令 I_2 和指令 I_4 之间,指令 I_3 和指令 I_6 之间有功能部件冲突。因此,在这个由 6 条指令组成的短程序中已经包含了所有可能的数据相关和功能部件冲突,这是一个很有代表性的程序。

下面,以这个典型程序的执行过程为例,分别介绍在超标量处理机中所采用的三种不同的指令调度方法。

方法一 顺序发射顺序完成

图 5.29 是采用顺序发射顺序完成的指令调度方法时,上面这个短程序的指令流水线时空图。6 条指令按照程序中的指令排列顺序从 I_1, I_2, …, I_6 分别在流水线 1 和流水线 2 中分 3 个时钟周期发射。

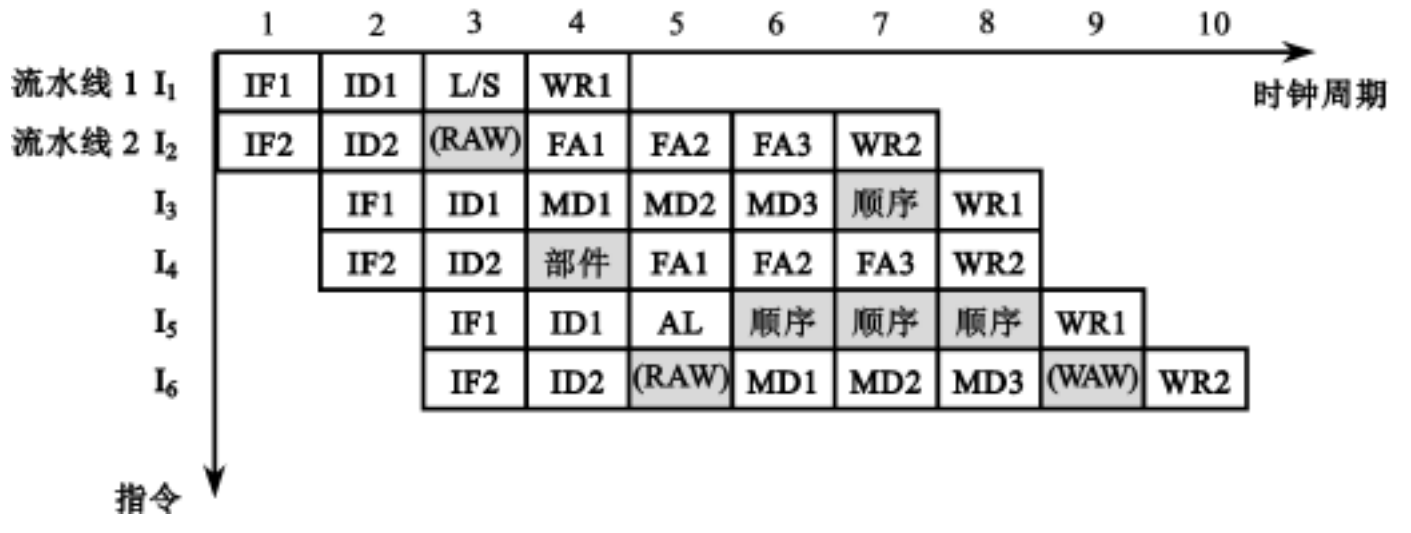


图 5.29 顺序发射顺序完成的指令流水线时空图

由于指令 I_1 与指令 I_2 之间有先写后读数据相关,指令 I_2 在流水线 2 中要等待 1 个时钟周期才能从流水线 1 中通过专用数据通路得到数据;因此,指令 I_2 在流水线 2 中译码(ID2)完成之后要等待 1 个时钟周期才能进入浮点加法部件(FA1)中执行,在图中用(RAW)表示。同样,因为指令 I_5 与指令 I_6 之间也有先写后读数据相关,因此,指令 I_6 也要等待 1 个时钟周期才能进入乘除法部件(MD1)中执行。

指令 I_4 在译码完成之后要再等待 1 个时钟周期才能进入浮点加法部件中执行,这是因为在指令 I_2 和指令 I_4 都要使用浮点加法器,它们之间有功能部件冲突。

为了维持顺序完成的要求,后发射的指令必须后进入写结果功能段。因此,指令 I_3 在乘除法部件中执行完成之后要延迟 1 个时钟周期进入写结果功能段。指令 I_5 在定点算术逻辑部件中执行完成之后要延迟 3 个时钟周期进入写结果功能段。

由于指令 I_5 和指令 I_6 之间有写—写数据相关,因此,指令 I_6 的写结果功能段要延迟 1 个时钟周期。

另外,指令 I_3 与指令 I_4 之间虽然有先读后写数据相关,由于 2 条指令在同 1 个时钟周期中发射,这种数据相关自然得到满足。

从图 5 .29 中可以看到,采用顺序发射顺序完成的调度方法,6 条指令共用了 l0 个时钟周期才完成。其中,除了流水线的装入和排空部分之外,还有 8 个空闲的时钟周期,在图中用阴影部分表示。在这 8 个空闲的时钟周期中,有 5 个时钟周期实际上是为了维持顺序完成才插入的(如果 I_5 提前完成,那么 I_6 与 I_5 之间的 WAW 相关就不存在了)。

方法二　顺序发射乱序完成

采用顺序发射乱序完成的流水线时空图如图 5 .30 所示。与图 5 .29 中的顺序发射顺序完成相比,相同的地方是 6 条指令按照程序中的指令排列顺序分别在流水线 1 和流水线 2 中分 3 个时钟周期发射,所不同的是,指令在流水线中完成的顺序是混乱的。指令的完成顺序与指令在程序中的排列顺序和在流水线中的发射顺序都无关。

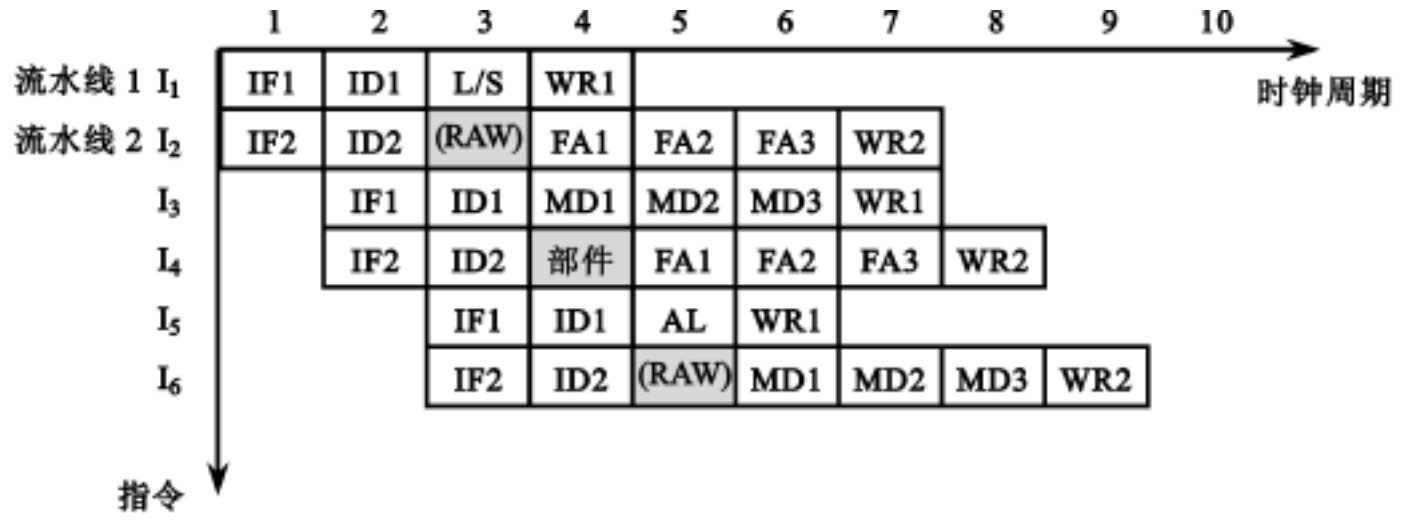


图 5 .30　顺序发射乱序完成的指令流水线时空图

从图 5 .30 中可以看到,只有 2 个先写后读数据相关和一个功能部件冲突,需要流水线空闲等待各 1 个时钟周期。与顺序发射顺序完成调度方法相比,少了 5 个空闲时钟周期。6 条指令总的执行时间为 9 个时钟周期,与顺序发射顺序完成调度方法相比节省 1 个时钟周期。因此,采用顺序发射乱序完成的指令调度方法,流水线的总的执行时间和功能部件的利用率都得到了改善。

方法三　乱序发射乱序完成

图 5 .31 是采用乱序发射乱序完成指令调度方法时的指令执行时序(指令流水线时空图)。

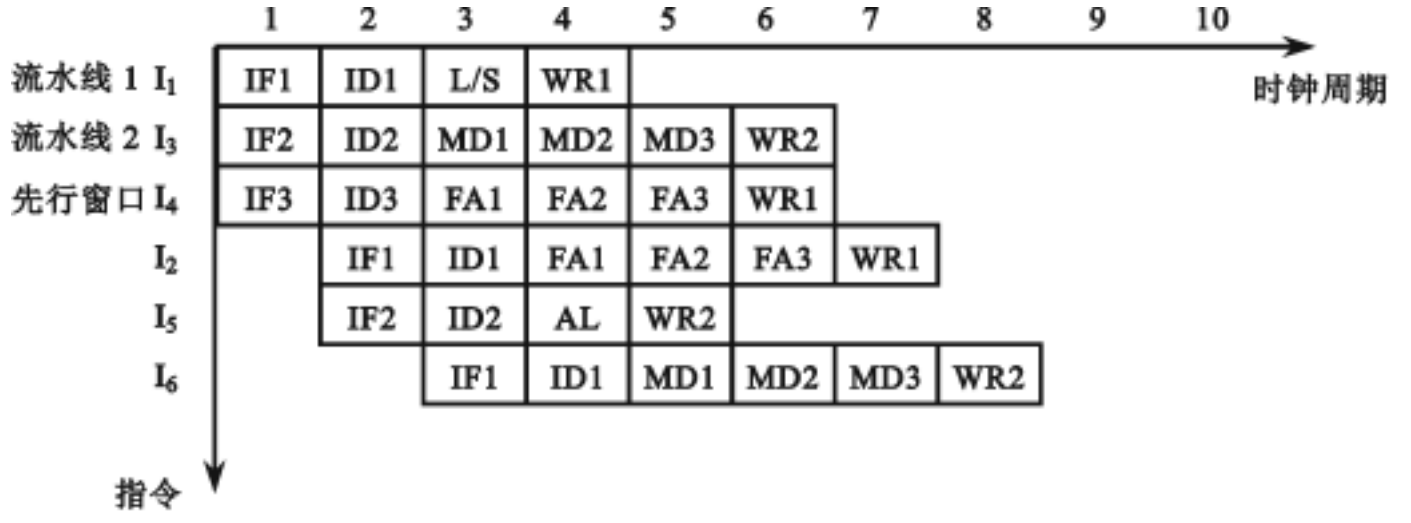


图 5 .31　乱序发射乱序完成的指令流水线时空图

由于指令 I_1 与指令 I_2 之间有先写后读数据相关,通常,指令 I_1 要早些发射;因此,指令 I_1 于第一个时钟周期在流水线 l 中发射,而指令 I_2 于第二个时钟周期也在流水线 1 中发射。指令 I_3 与指令 I_4 之间有先读后写数据相关,没有功能部件冲突,2 条指令可以同时发射。这样,先读后写数据相关也就自然消除了。指令 I_3 在流水线 2 中发射,而指令 I_4 通过先行指令窗口发射。通常,先行指令窗口除了能够做数据相关性分析和功能部件冲突的检测之外,还应该至少有一套取指令部件和一套指令译码部件。

指令 I_5 必须在指令 I_6 之前先发射,这是因为指令 I_5 与指令 I_6 之间存在有先写后读数据相关。因此,在第二个时钟周期,指令 I_2 在流水线 1 中发射,而指令 I_5 在流水线 2 中发射。先行指令窗口不发射指令。

在第三个时钟周期指令 I_6 在流水线 l 中发射。

在采用乱序发射时,指令的完成次序必然也是乱序的。从图 5 .31 中可以看出,除了流水线的装入和排空之外,已经没有空闲的时钟周期,因此,功能部件得到了充分利用。6 条指令总的执行时间缩短为 8 个时钟周期,与顺序发射顺序完成调度方法相比节省了 2 个时钟周期,与顺序发射乱序完成调度方法相比节省了 1 个时钟周期。

4. 超标量流水处理机的性能

为了便于比较,把单流水线普通标量处理机的指令级并行度记做(1,1),超标量处理机的指令级并行度记做(m,1),超流水线处理机的指令级并行度记做(1, n),而超标量超流水线处理机的指令级并行度记做(m, n)。

在理想情况下 N 条没有资源冲突、没有数据相关和控制相关的指令在单流水线

普通标量处理机上的执行时间为:

$$T(1,1) = (k + N - 1)\Delta t$$

其中,k是流水线的级数,Δt是一个时钟周期的时间长度。

如果把相同的 N 条指令在 1 台每个时钟周期发射 m 条指令的超标量处理机上执行,所需要的时间为:

$$T(m,1) = (k + \frac{N - m}{m})\Delta t$$

其中,第一项是第一批 m 条指令同时通过 m 条指令流水线所需要的执行时间,而第二项是执行其余 $N - m$ 条指令所需要的时间,这时,每一个时钟周期有 m 条指令分别通过 m 条指令流水线。

因此,超标量处理机相对于单流水线普通标量处理机的加速比为:

$$S_p(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{m(k + N - 1)}{N + m(k - 1)}$$

目前,在许多高性能超标量处理机中已经采用了乱序发射乱序完成的指令调度方法。通常设置有一个存储容量为几条指令到十几条指令的比较小的先行指令窗口,一个比较简单的数据相关性分析部件和一个功能部件冲突的检测机构,一般采用计分牌机制来表示数据相关性和功能部件的冲突。另外,通过优化编译器对指令序列进行重组来共同开发程序中指令级并行性。

5.4.2 超流水线技术

在前面介绍的一般标量流水线处理机中,通常把一条指令的执行过程分解为取指、译码、执行和写回 4 级流水线。如果把其中的每级流水线再细分,将每一级分解为两级延迟时间更短的流水线,则 1 条指令的执行过程就要经过 8 级流水线。这样,在 1 个基本时钟周期内就能够取指、译码、执行和写回各 2 条指令。这种在 1 个基本时钟周期内能够分时发射多条指令的处理机称为超流水线处理机。在有些资料上把指令流水线的级数为 8 级或超过 8 级的流水线处理机称为超流水线处理机。

超流水线处理机的工作方式与上一节中介绍的超标量处理机不同,超标量处理机是通过重复设置多个取指、译码、执行和写回部件,并让这些功能部件同时工作来提高指令的执行速度,实际上是以增加硬件资源为代价来换取处理机性能的;而超流水线处理机则不同,它只需要增加少量硬件,是通过各部分硬件的充分重叠工作来提高处理机性能的。从流水线的时空图上看,超标量处理机采用的是空间并行性,而超流水线处理机采用的是时间并行性。

超流水线处理机的指令执行时空图如图 5 .32 所示。

图中只是超流水线处理机原理上的指令执行时空图,实际上,功能段还要进一步细分,一个功能段要细分为多个流水级,每一个流水级也都有名称。在分解功能段时要根据实际情况,有些功能段分解的流水级数可多些;例如,图 5 .32 中的‘ 译码

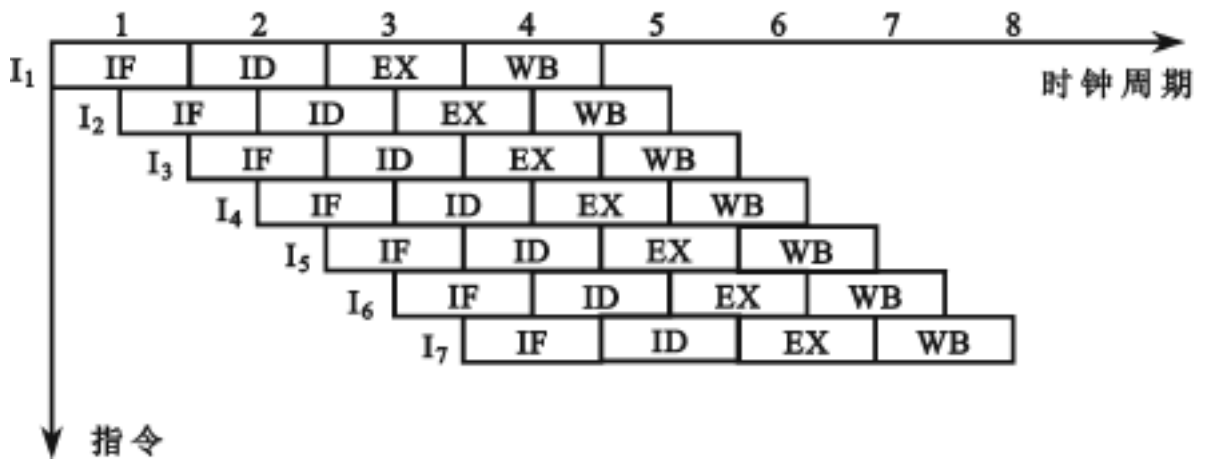


图 5 .32 超流水处理机的指令执行时空图

(ID)”功能段,可以再细分为“译码”流水级、“取第一个操作数”流水级和“取第二个操作数”流水级等;有些功能段分解的流水级数可少些,有的功能段可以不再细分,如“写回”结果功能段一般不再细分。

在早期生产的计算机中,巨型计算机 CRAY-1 和大型计算机 CDC-7600 属于超流水线处理机,其指令级并行度 ILP = 3。在目前大量使用的微处理器中,只有 SGI 公司的 MIPS(Microproccssor Without Interlocked Piped Stages)系列处理机属于超流水线处理机。MIPS 是除 Intel 公司的 X86 系列微处理器之外,生产量最大的一种微处理器。MIPS 系列的微处理器主要有 R2000, R3000, R4000, R5000 和最近刚投放市场的 R10000 等几种,其中, R4000 是典型的超流水线处理机,其流水操作示意如图 5 .33 所示。

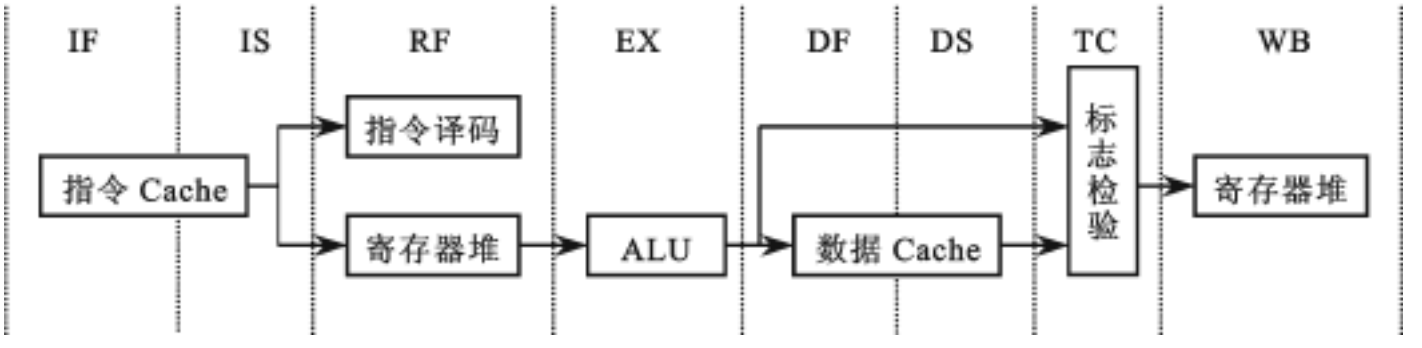


图 5 .33 MIPS R4000 处理机的超流水线示意图

图中,IF 表示取第一条指令;IS 表示取第二条指令;RF 表示读寄存器堆,指令译码;EX 表示执行指令;DF 表示取第一个操作数;DS 表示取第二个操作数;TC 表示数据标志检验;WB 表示写回结果。

在一台指令级并行度为(1, n)的超流水线处理机上,执行 N 条没有数据相关和控制相关的指令所需要的时间为:

$$T(1,n) = (k + \frac{N-1}{n})\Delta t$$

其中，k 是指令流水线的功能段数，或时钟周期数，而不是流水线级数。在一般超流水线处理机中，指令流水线的级数实际应为 kn。上式中的头一项是第一条指令通过指令流水线执行完成所需要的时间，而第二项是执行其余 $N-1$ 条指令所需要的时间，这时，每一个时钟周期有 n 条指令要在指令流水线中执行完成，也就是每一个流水线周期执行完成 1 条指令。

超流水线处理机相对于单流水线普通标量处理机的加速比为：

$$S_p(1,n) = \frac{T(1,1)}{T(1,n)} = \frac{n(k+N-1)}{nk+N-1}$$

5.4.3 超标量超流水线技术

为了进一步提高处理机的指令级并行度，可以把超标量技术与超流水线技术结合在一起，这就是超标量超流水线处理机。

超标量超流水线处理机的指令执行时空图如图 5.34 所示，它在 1 个时钟周期内要发射指令 m 次，每次发射指令 n 条，因此，超标量超流水线处理机每个时钟周期总共要发射指令 $m \times n$ 条。

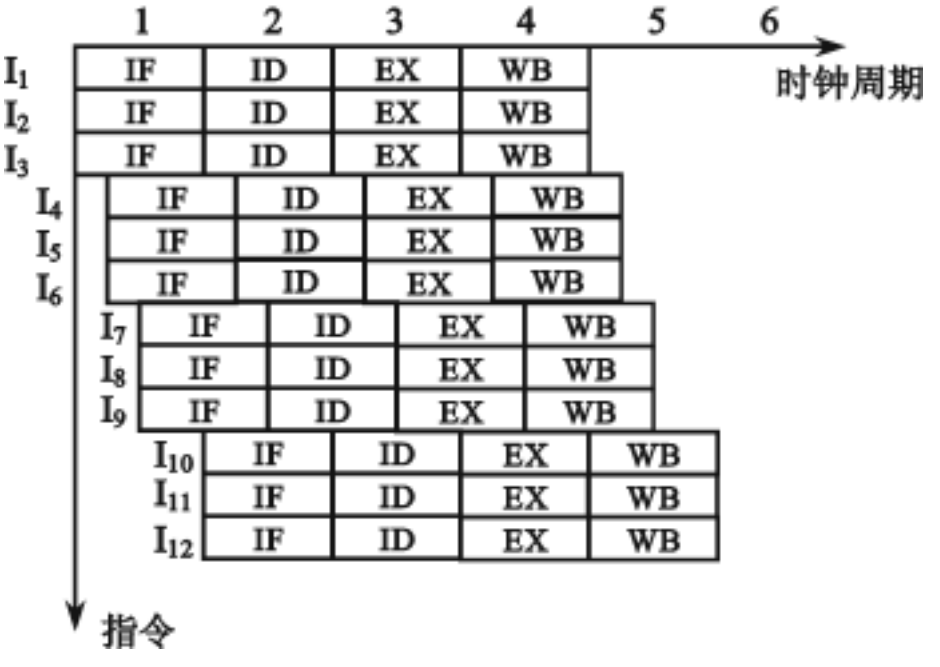


图 5.34 超标量超流水线处理机的指令执行时空图

在图中，每 1 个时钟周期分为 3 个流水线周期，每一个流水线周期发射 3 条指令，从图中可以看出，每个时钟周期能够发射并执行完成 9 条指令。因此，在理想情况下，超标量超流水线处理机执行程序的速度应该是超标量处理机和超流水线处理机执行程序速度的乘积。

在当今的主微处理器中,只有 DEC 公司的 Alpha 21064 采用的是超标量超流水线结构。

在一台指令级并行度为(m, n)的超标量超流水线处理机上,连续执行 N 条没有资源冲突、没有数据相关和控制相关的指令所需要的时间为:

$$T(m, n) = \left(k + \frac{N - 1}{mn} \right) \Delta t$$

其中,k 是指令流水线的时钟周期数,而不是流水线级数。例如,在 Alpha 21064 超标量超流水线处理机中,$k = 4$。Δt 是 1 个时钟周期的时间长度。上式中的第一项是开始 m 条指令通过指令流水线所需要的时间,第二项是执行其余 $N - 1$ 条指令所需要的时间,每一个时钟周期平均执行完成 $m \times n$ 条指令,也就是每一个流水线周期平均执行完成 n 条指令。

超标量超流水线处理机相对于单流水线标量处理机的加速比为:

$$S_p(m, n) = \frac{T(1,1)}{T(m, n)} = \frac{mn(k + N - 1)}{mnk + N - m}$$

5.4.4 超长指令字(VLIW)技术

超长指令字方法是在 1983 年由美国耶鲁大学的 Fisher 教授首先提出的,它与超级标量方法有许多类似之处,但它以 1 条长指令来实现多个操作的元并行执行,以减少对存储器的访问。这种长指令字往往达上百位,甚至上千位。并发操作主要是在流水的执行阶段进行的。

超长指令机的主要特点是:

① 单一的控制流,只有一个控制器,每个周期启动 1 条长指令。

② 超长指令字被分成多个控制字段,每个字段直接独立地控制每个功能部件。

③ 含有大量的数据通路和功能部件,由于编译器在编译时间已考虑可能出现的数据相关和资源相关,故控制硬件较简单。

④ 在编译阶段完成超长指令中多个可并行执行操作的调度。

图 5.35 中示出了一个含有两个存取部件、一个浮点加部件和一个浮点乘部件的 VLIW 机。所有功能部件均由同一时钟驱动,在同一时刻控制每个功能部件的操作字段组成一个超长指令字。指令字长度和功能部件数有关。超长指令字的生成是由编译器来完成的,由它将串行的操作序列合并为可并行执行的指令序列,以最大限度地实现操作并行性。下面通过一个例子来说明这种过程。假设要执行以下的赋值语句:

$$C = A + B,\ K = I + J,\ L = M - K,\ Q = C \times K$$

若按串行操作进行,则其所用的指令序列如图 5.36 所示。

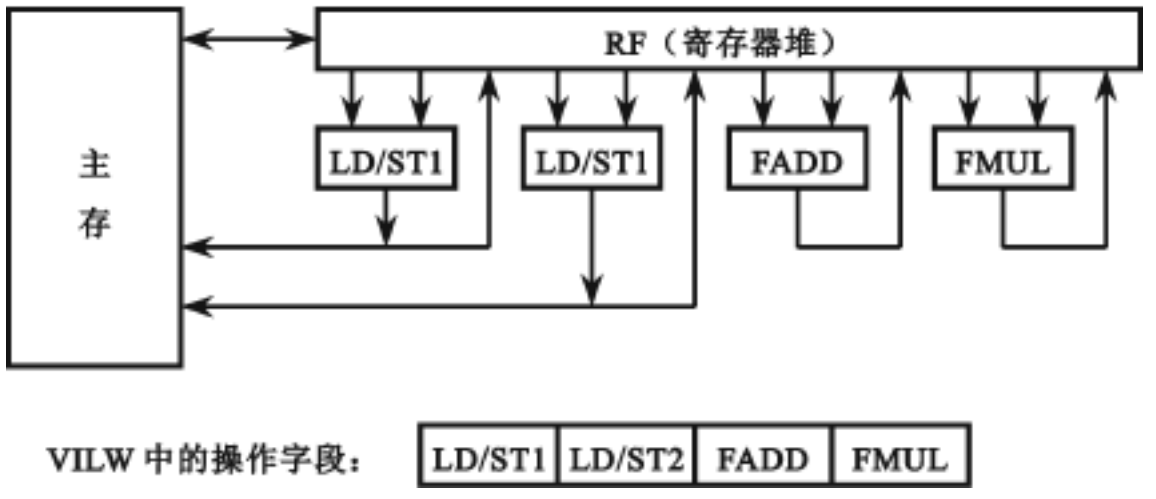


图 5 .35　VLIW 机的结构框图

源代码	操　作	所需周期
C = A + B	LOAD A	1
	LOAD B	1
	C = A + B	1
	STORE C	1
K = I + J	LOAD I	1
	LOAD J	1
	K = I + J	1
	STORE K	1
L = M - K	LOAD M	1
	L = M - K	1
	STORE L	1
Q = C×K	Q = C×K	2
	STORE Q	1

图 5 .36　串行操作指令序列及时钟周期

假设 LOAD/ STROE 指令以及 FADD 操作需 1 个周期完成,而 FMUL 需 2 个周期完成,则上述的指令串操作共需 14 个周期方可完成。

在超长指令字技术中,常采用两种压缩技术。

第一种是局部性压缩——表调度法。这种方法是在程序基本块范围内进行压缩(只有 1 个入口和 1 个出口的代码段),若采用表调度的编译方法,则可将原来的 13 条指令序列,压缩成 6 条长字指令,此时完成同样的操作,仅需 6 个周期。图 5 .37 中

示出了经压缩后(经表调度法调度后)的 VLIW 指令序列。

LOAD A	LOAD B		
LOAD I	LOAD J	C = A + B	
LOAD M	STORE C	K = I + J	
	STORE K	L = M - K	Q = C×K
	STORE L		
STORE Q			

图 5.37 经表调度法调度后的 VLIW 指令序列

第二种是全局压缩法。这种方法允许代码操作可在基本块之间移动,从而可获得更好的压缩效果。当然,这种代码操作在基本块间移动是受到一定限制的,而且为了保持程序原来的语义,某些代码操作移动后,通常还需要增加一些辅助操作。全局压缩主要有三种方法:路径调度(Trace Scheduling)、渗透调度(Percolation Scheduling)和软件流水(Software Pipelining)。

VLIW 计算机曾有过商品化机器,如美国原 Multiflow 公司生产的 TRACE 型号机器,以及 Cydrome 公司生产的 Cydra 5 型号机器。但由于性能价格比不理想等原因,这两家公司已停止生产 VLIW 计算机,而 VLIW 计算机中所采用的压缩技术已广泛流传开来。

5.5 Pentium 微处理器中的流水技术

Intel 的第五代微处理器芯片 Pentium(P54C)和 Pentium MMX(P55C)以及 Pentium Pro,具有一些 RISC 的特征,同时也具有更多的 CISC 的特征。为加深对 PentiumⅡ/Ⅲ微处理器的理解,本节先介绍 Pentium 微处理器的超标量流水线结构,然后再介绍 PentiumⅡ/Ⅲ的动态执行技术。

5.5.1 Pentium 微处理器的超标量流水线

Pentium 微处理器与它的前身 80486 有两个重要的不同点:一是它具有片内分立的 L1 级、容量各为 8KB 的指令 Cache 和数据 Cache,二是它有 2 个 32 位的 ALU 来完成整数运算和逻辑操作,因而能支持两条整数指令流水线的并行执行,这种流水结构属于超标量流水线结构,如图 5.38 所示。

两条指令流水线分别称之为 U 流水线和 V 流水线。U,V 两个字母没有别的含义,它们只是未被执行部件所引用的两个连续字母。每条指令流水线分为 5 段:预取指令(IF)段,译码 1(ID1)段。译码 2(ID2)段,执行(EX)段和写回寄存器(WB)段。

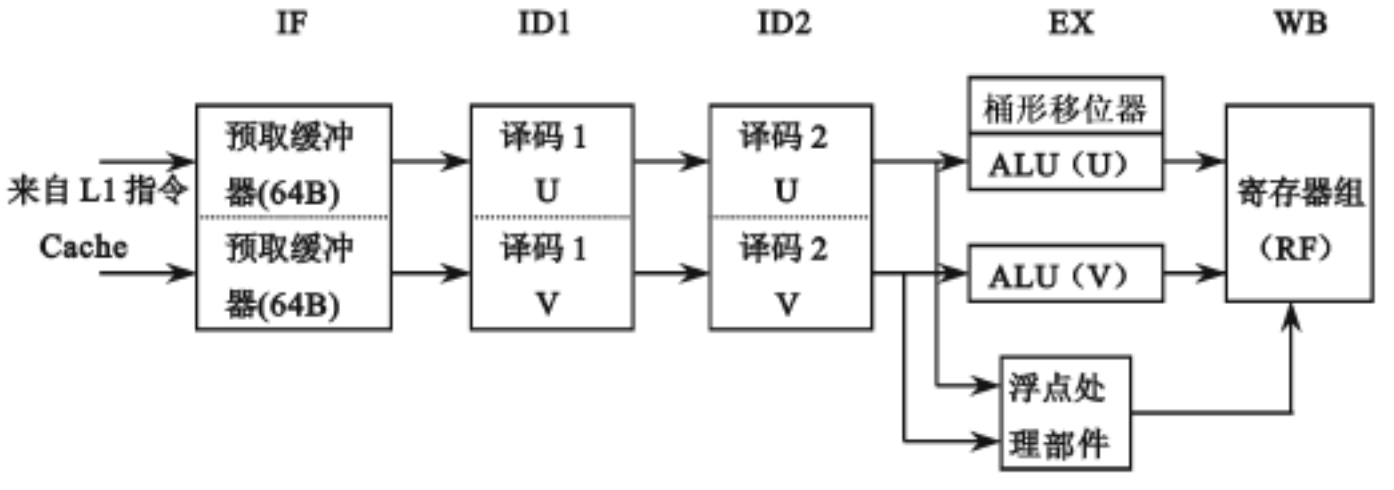


图 5.38　Pentium 微处理器中的超标量流水线结构

1. 指令预取器和预取缓冲器

虽然,U,V 流水线各有一个预取(IF)段,但实际上指令预取器和预取缓冲器对两条流水线是共有的,它们的主要功能是:

① 指令预取器总是按给定的指令地址,由 L1 指令 Cache 顺序地取指令,直到在 ID1 段遇到一条转移指令并预测它在 EX 段将发生转移时为止。此时,由转移目标缓冲器 BTB(Branch Target Buffer)提供预测发生转移的目标地址。按此地址开始又是顺序取指令,直到遇到一条转移指令并预测转移将发生时为止。指令预取器总是以这种折线式顺序由指令 Cache 取指,读取的内容整行(32 字节)装入当前活跃的预取缓冲器。有时,折线顶点给定的起始指令地址位于指令 Cache 一行的中、后部,一条指令跨两个 Cache 行。为此,指令预取器能指挥指令 Cache 完成分立行存取(Split-line Access),将两行内容装入预取缓冲器。

② 预取缓冲器有两个,每个 64 字节的容量,也常称为队列 A 和 B。两个队列任何时候只有一个是当前正在使用的,或称之为活跃的;另一队列是空闲的。例如,当前队列 A 是活跃的,只要它有 32 字节空位置,指令预取器总是由指令 Cache 取整行装入它。若在 ID1 段遇到一条转移指令并预测它在 EX 段将发生转移时,指令预取器依 BTB 提供的转移目标地址开始顺序取指令装入队列 B,而队列 A 空闲被冻结。如果此转移指令在 EX 段执行时确实发生转移,此转移目标地址处开始的指令流已进入流水线(到达 ID2 段),没有任何性能损失,队列 B 继续是活跃的。如果此转移指令在 EX 段执行时实际未发生转移,即预测错误,此时要将已进入流水线的指令清除并激活队列 A,按原转移指令下一条指令的地址顺序取指令装入流水线。队列 A 和 B 以这种乒乓方式交替工作。

③ 指令预取器不仅读取指令 Cache 整行内容装入当前活跃的预取队列,并在队列中标志每条指令的边界。我们知道,IA 指令长度是变动的,不算前缀,指令长度可

以是 1～12 字节。当译码器 1(ID1)段空时,由当前活跃的队列将连续的两条指令(i_t,i_{t+1})送入 U,V 流水线的 ID1 段,即 i_t 进入 U 的 ID1 段,i_{t+1} 进入 V 的 ID1 段。

2. 指令译码 1

指令译码在流水线的两段中出现,被称为译码 1(ID1)和译码 2(ID2)。ID1 段的主要功能是:

① 对指令操作码部分进行译码,检查是否为转移指令;若是转移指令,则将此指令的地址送往 BTB。若 BTB 命中,则根据该项的历史位状况预测此指令在 EX 段是否发生转移,并预测为发生时将该项登记的转移目标地址提交给指令预取器。若 BTB 失效,则意味着 BTB 预测逻辑没有该指令的历史。固定预测为在 EX 段该指令将不发生转移(即使是一条无条件转移指令也是如此),即不指挥预取器更改预取的指令流。注意,在将转移指令的地址提交给 BTB 后,不论预测是否发生,此转移指令都要进入 ID2 段,在 EX 段最终实际执行。

② 指令配对检查。根据 Pentium 微处理器的指令配对规则,检查进入 ID1 段的 i_t、i_{t+1} 两条指令是否可配对。若可配对,则 i_t 在 U 流水线,i_{t+1} 在 V 流水线,两指令同时离开 ID1 段进入 ID2 段,从而真正开始两指令的并行操作。若不可配对,则 U 流水线中的 i_t 指令先进入 ID2 段,然后 ID1 段 V 中的 i_{t+1} 指令移送到 ID2 段 U 中,即 i_t,i_{t+1} 两条指令都使用 U 流水线先后顺序操作。

3. 指令译码 2

指令译码 2(ID2)段的主要功能是,生成存储器操作数地址,并按保护模式的规定检查是否有保护违约,若有则产生例外。因此,ID2 段也常称为地址生成段,它使用分段部件、分页部件以及 DTLB(Data Translation Look - aside Buffer,数据转换后援缓冲器),将产生的存储器操作数的物理地址提交给 L1 数据 Cache。能同时产生两个地址,分别为 U,V 流水线服务。Pentium 的地址生成器比 80486 有所增强,即使是基址加变址的寻址方式也能在 1 个时钟周期内完成(80486 需 2 个时钟周期)。

值得注意的是,不需要存储器操作数的指令也要历经 ID2 段,而且两条配对指令要同时离开 ID2 段进入 EX 段。

另外,转移指令的目标地址计算也是在 ID2 段完成的。

4. 执行

执行段以两个 ALU 为中心,完成 U,V 流水线的两条指令的算术逻辑运算。U 流水线的 ALU 带有一个桶形移位器(Barrel Shifter),因而功能要强于 V 流水线的 ALU。执行(EX)段的主要功能有:

① 以 ID2 段提供的存储器操作数地址,在 L1 数据 Cache 中存取操作数(若 L1

数据 Cache 失效,在 L2 Cache 或主存中查找)。总之,在 EX 段前部,指令所需的存储器操作数、寄存器操作数要全部就绪,在 EX 段后部完成指令所要求的算术逻辑运算。

② U,V 两条流水线中的执令同时进入 EX 段。若 U 流水线的指令先执行完,可先行离开 EX 段进入 WB 段;但是,V 流水线的指令先执行完则必须等待 U 流水线的指令执行完,然后一起进入 WB 段。

③ 一条转移指令在 EX 段确认是否实际发生转移。若实际情况与预测相符,则除了修改 BTB 该项历史位之外,什么事情也不发生。若预测错误,则除修改该项历史位外,还要清除该指令之后已在 U、V 流水线中的全部指令并指挥指令预取器按相反方向重新取指令装入流水线。即若预测发生转移而实际不发生时,立即解冻另一指令队列,由此转移指令的下一条指令开始顺序取指令。若预测不发生转移而实际发生时,则以计算出的转移目标地址,指挥预取器由此地址开始顺序取指令(必要的话,还要修改 BTB 中此项内容)。前一种预测错误要浪费处理器 3～4 个时钟周期,后一种预测错误要浪费处理器 5～6 个时钟周期。

④ 前面已经说过,当一条转移指令在 ID1 段将指令地址提交给 BTB 而 BTB 未命中时,则 BTB 不做预测而实际上就是预测不发生,指令预取器仍顺序取指令。在 EX 段,若此指令确实没发生转移,则什么也没发生,以后遇到此转移指令时仍是作为一个"新面孔"的转移指令按上述对待。在 EX 段若此指令发生转移的话,则按上一点所说的后一种预测错误处理,还要将转移目标地址提交给 BTB,结合在 ID1 段提交的转移指令地址,在 BTB 中建立一个新项,并设定历史位为强发生(Strongly Taken)转移状态。

⑤ 同 80486 一样,Pentium 微处理器内部已有浮点运算器 FPU,但 FPU 的性能已做了很大改进。FPU 内有 8 个 80 位的浮点寄存器 FR0～FR7,内部数据总线为 80 位宽,并有分立的浮点加法器、乘法器和除法器,可同时进行 3 种不同的运算。

浮点指令流水线为 8 段,即预取指令(IF)段、指令译码 1(ID1)段、指令译码 2(ID2)段、取操作数(EX)段、执行 1(X1)段、执行 2(X2)段、结果写回(WB)段和错误报告(ER)段。前 4 段与 U,V 流水线的 IF,ID1,ID2,EX 段共享,后 4 段在 FPU 中完成。

从结构上讲,浮点指令流水线也是双指令流水线。但 1 条浮点运算指令既不能与浮点运算指令配对也不能与整数运算指令配对,只能与 FCXH 浮点交换这样的少数指令配对。于是,常常是只在 U 流水线中执行浮点运算指令,V 流水线空闲或配对执行 FCXH 指令(有浮点编程经验的人都知道,这已是性能的很大改善)。故一般而言,Pentium U,V 流水线每次只能执行 1 条浮点运算指令。

5. 写回

写回(WB)段的主要功能是以 ALU 运算结果修改 IA 寄存器,这包括对

EFLAGS 标志寄存器的修改。

Pentium 微处理器的寄存器分为系统级寄存器组和基本结构寄存器组两大类。系统级寄存器组包括 GDTR, LDTR, IDTR, IR 这类的表基地址寄存器和 CR0～CR4 控制寄存器以及用于调试和测试目的的寄存器。基本结构寄存器包括,指令指针 EIP 和标志寄存器 EFLAGS,以及 CS, DS, ES, FS, GS 和 SS 6 个段寄存器和 EAX, EBX, ECX, EDX, ESI, EDI, EBP 和 ESP 8 个通用寄存器。一般情况下,程序员大量使用的是 8 个通用寄存器,可用于编程的寄存器太少。

5.5.2 Pentium 微处理器 U,V 流水线指令配对

Pentium 微处理器比 80486 的一大进步是,指令流水线由单一流水线变为 U, V 双流水线,即是一个超标量为 2 的流水线。但是, Pentium 调度指令的能力极其有限,它只能对由预取缓冲器取来的连续两条指令进行判测。若两条指令是可配对的,则前一条指令进入 U 流水线后一条指令进入 V 流水线,两条指令可并行操作。若两条指令是不可配对的,则两条指令要先后进入 U 流水线,串行操作将不能展现超标量流水线的优势。为此,本小节讨论指令配对(Istruction Pairing)问题,并只讨论整数流水线。

1. 整数指令的配对规则

由上节知道,指令配对的检查是在 ID1 段进行的。图 5 .39(a)给出可配对的两条整数指令在 U, V 流水线并行操作的情况,图 5. 39(b)给出不可配对的两条指令在 U 流水中先后顺序操作的情况。从图中可看出,后者带来性能损失。

时钟	1	2	3	4	5
i_t	IF	ID1(U)	ID2(U)	EX(U)	WB(U)
i_{t+1}	IF	ID1(V)	ID2(V)	EX(V)	WB(V)

(a)配对的整数指令操作

时钟	1	2	3	4	5	6
i_t	IF	ID1(U)	ID2(U)	EX(U)	WB(U)	
i_{t+1}	IF	ID1(V)	ID1(U)	ID2(U)	EX(U)	WB(U)

(b)不可配对的整数指令操作

图 5 .39 Pentium U,V 流水线操作时空图

ID1 段检查 i_t , i_{t+1} 两条连续指令是否可配对,其可配对规则是:

① 两条指令都是简单指令。简单指令大多数是以硬联逻辑实现的指令,执行段

只需 1 个时钟周期。少数涉及到寄存器到存储器或存储器到寄存器的 ALU 指令，它们在执行段需要 2～3 个时钟周期，但 Pentium 包含了某些排序化硬件，允许将它们作为简单指令对待。即使两条指令都是简单指令，但由于 Pentium 的 U，V 流水线功能不对称，有的指令可配对但只能出现在 U 流水线中，有的只能出现在 V 流水线中。

② 两条指令都不同时含有立即数和偏移量。

(3)只有 i_t 指令允许带有指令前辍(Prefix)。

(4)两条指令间不存在 RAW(写与读)相关和 WAW(写与写)相关。即 i_t 的目标寄存器既不是 i_{t+1} 的源寄存器，也不是 i_{t+1} 的目标寄存器。但要注意，如 i_t 为 MOV AL，13；i_{t+1} 为 MOV AH，15；这样的两条指令仍有 WAW 相关，因为 AL，AH 是同一 32 位寄存器的两部分，而 Pentium 将所有寄存器参照都作为整个 32 位寄存器存取对待。至于第 3 种冲突即 WAR 冲突，由于 Pentium 的 U，V 流水线采用了按序发射按序完成策略而予以避免了，详见下述内容。

2. U，V 流水线按序发射按序完成策略

指令的发射和完成策略，即超标量流水线的调度，对于充分利用指令级的并行度，提高超标量处理器的性能十分重要。前面已经介绍，超标量流水线的调度策略共有三种：即顺序发射顺序完成，顺序发射乱序完成，乱序发射乱序完成。

无论采用哪种调度策略，都要保证程序运行最终结果的正确性。我们将看到，Pemium 微处理器采用的是顺序发射顺序完成策略；Pentium Ⅱ/ Ⅲ处理器采用的是按序发射乱序完成策略，而以按序回收来保证程序最终结果的正确性。

下面我们考察 Pentium 超标量流水线是如何实现顺序发射顺序完成的。图 5．40 给出了可配对的 4 对指令通过 U，V 流水线各段的时空图。

Pentium 的 U，V 流水线的 ID2 段虽名为译码 2 段，但它的主要功能是生成存储器操作数地址，实际上已属于指令执行功能范畴。故 U，V 流水线的指令发射实际上指由 ID1 段进入 ID2 段。在 ID1 段检查配对合格的一对指令，即为简单指令又无 RAW，WAW 数据冲突，则同时被发射到 U，V 流水线的 ID2 段。

图 5．40 的 I_1、I_2 两条指令用了 5 个时钟周期通过流水线，是各段都无延迟的一般情况。因为执行(EX)段总在写回(WB)段之前完成，即指令读源操作数总在写目标寄存器之前完成，WAR 冲突自然避免。在时钟 5 的 1 个周期内，I_1、I_2 指令将其执行结果写回 IA 寄存器，虽然是同时完成写回的，但两指令已无数据冲突，写回必定是正确的并符合程序顺序运行的要求。

上一小节说过，配对指令要同时进入，同时离开 ID2 段。若一条指令在 ID2 段滞留，另一条指令也必须在 ID2 段停顿。图 5．40 的 I_3，I_4 两条指令的情况是 I_4 在 ID2 段滞留，I_3 也被迫在 ID2 段停顿。

图 5．40 中 I_5，I_6 指令情况是，由于 I_3，I_4 指令在时钟 6 才离开 ID2 段，故 I_5，I_6 指

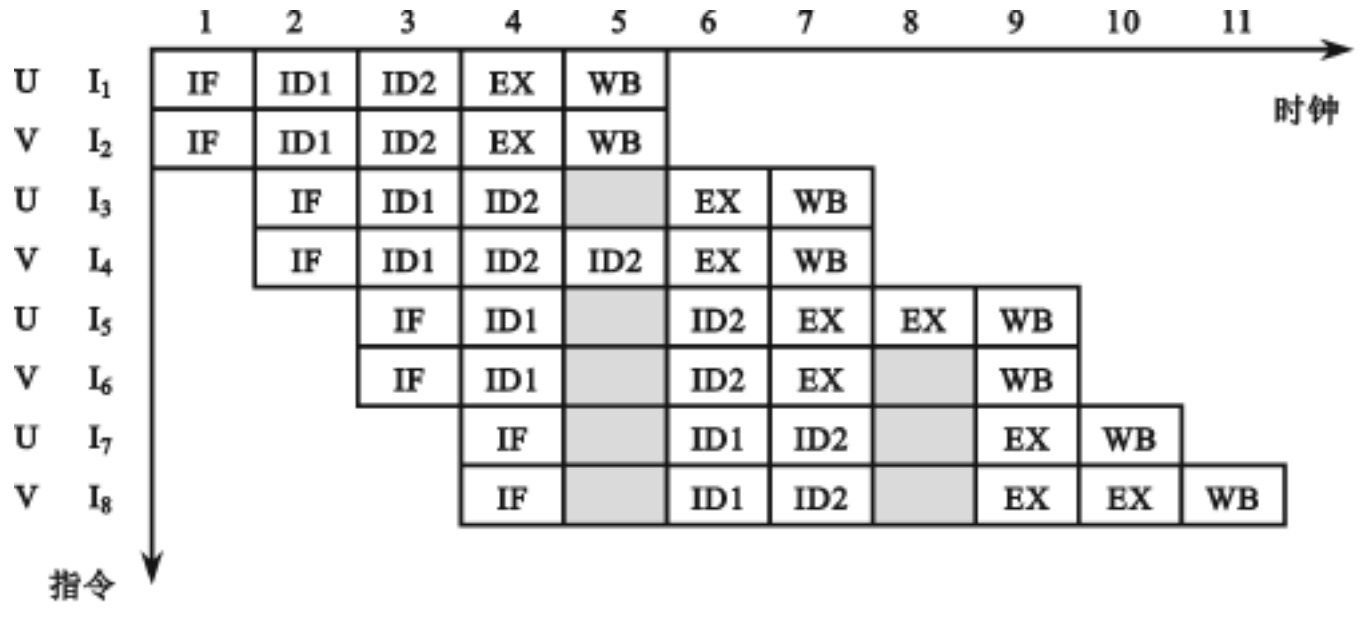


图 5.40 可配对指令通过流水线的几种情况

令在 ID1 段停顿 1 个时钟周期在时钟 6 才进入 ID2 段。在 U 流水中的 I_5 指令在执行(EX)段用了 2 个时钟周期,而 V 流水中的 I_6 指令虽实际执行只用 1 个时钟周期,但为保证按序完成它被迫在 EX 段停顿 1 个时钟周期。然后,在时钟 9 时 I_5, I_6 指令同时离开 EX 段进入 WB 段。

图 5.40 中 I_7, I_8 指令情况是,由于 I_5, I_6 的占用,它们进入 ID1 段和进入 EX 段都推迟了 1 个时钟周期。现在,U 流水线中的 I_7 在执行段只用了 1 个时钟周期,而 V 流水线中的 I_8 在执行段却用了 2 个时钟周期。即前面的指令先执行完,先进入 WB 段,后面的指令后进入 WB,这种情况是允许的。

总之,U,V 流水线可配对指令同时由 ID1 段发射到 ID2 段,并同时进入 EX 段,U 流水线中的指令可先执行完,或两条指令同时执行完。于是,实现了超标量流水线的顺序发射顺序完成。

5.5.3 Pentium 微处理器中的 BTB

转移目标缓冲器 BTB(Branch Target Buffer)。正是由于处理器指令集中有控制程序流向改变的指令,才使得程序功能丰富并编制得灵巧。这些指令包括有,跳转(JMP)指令、调用(CALL)指令、返回(RET)指令、中断(TNT)指令等,可以用转移(Branch)指令一词来代表。转移指令又可分为条件转移指令和无条件转移指令,无条件转移指令执行时肯定发生转移,条件转移指令执行时是否发生转移取决于指令所要求的条件当时是否能满足。

但是,转移指令会对流水线造成性能损失。因为转移发生时要排空流水线,由转移目标地址开始取指令重新注入流水线,而且指令流水线的段数越多和并行流水线的条数越多,转移指令造成的性能损失越大。

已有许多技术用于减少转移指令对流水线性能的影响,其中两种最常用的技术是:基于编译软件的延迟转移(Delayed Branching)技术和基于硬件实现的转移预测(Branch Prediction)技术。这里只介绍后一种技术。

转移预测技术的基本思想是,在指令实际执行之前,在流水线的指令译码段期间,即使对所遇到的转移指令进行预测,若预测它在执行段将发生转移,则现在就由转移目标处取指令,以此来减少流水线的排空量。当然,预测正确与否还要待此转移指令具体执行结果的检验。若预测正确,转移指令对流水线造成的性能损失确实减小了。若预测不正确,仍要重新注入流水线,尤其是对实际不发生转移的指令却预测为转移发生,反而凭空增加了一次性能损失。因此,转移预测技术要努力提高预测的正确率,一般要求正确率要高于 80%。

转移预测有静态、动态之分。静态预测法只依据转移指令类型来预测。例如,对某一类条件转移指令总是预测为转移发生,对另一类总是预测为转移不发生。又如,对朝前向转移的条件转移指令总是预测为转移不发生,对朝后向转移的指令总预测为发生,等等。静态预测法简单但正确率不高,只能作为其他转移处理技术的辅助手段。

动态预测法是依据一条转移指令过去的行为来预测此指令的将来行为。因为程序结构中重复或循环执行的机会众多,较好的预测算法会使动态预测法有较高的正确率,故普遍被当代处理器所采用。这里,我们只介绍 Pentium 微处理器依据 BTB 所实现的动态预测法。

图 5 .41 给出 Pcntium 微处理器的 BTB 的逻辑结构,它是一个 4 路组相联 Look-aside式 Cache,共有 256 行。以转移指令地址的低 6 位为组索引,转移指令地址的高 26 位为 Cache 行的标记字段。转移指令的转移目标地址(32 位)与历史位(2 位)及有效位(1 位)为 Cache 行的内容。

BTB 项的历史位登记该项转移指令以前的执行行为(最近两次连续发生转移的次数),用于在 ID1 段预测此指令是否发生转移。在 EX 段要根据实际是否发生转移,来修改命中项的历史位;或对于 BTB 未命中的转移指令而在 EX 段实际发生转移的情况,在 BTB 中建立新项并设定历史位为 11。图 5 .42 给出 BTB 历史位的意义及状态转换。

由图中可看出,Pentium 微处理器的BTB 对历史位意义的设定更倾向于预测转移发生。11 历史位常称为强发生(Strongly Taken)状态,10 位称为发生(Taken)状态,01 位称为弱发生(Weakly Taken)状态,3 种历史位都预测转移发生。并且,对一个新遇到的转移指令并实际发生转移的情况,则所建新项设置为强发生状态。Pentium 更倾向于预测发生的原因有两方面:一是条件转移指令发生转移的概率一般为 60%左右;二是 Pentium 的双指令队列结构,使预测为转移发生而实际上不发生时流水线蒙受的损失,要小于预测为转移不发生而实际上发生时的损失。

Pentium 微处理器的 BTB 操作分为两个阶段:指令译码(ID1)段的预测阶段和

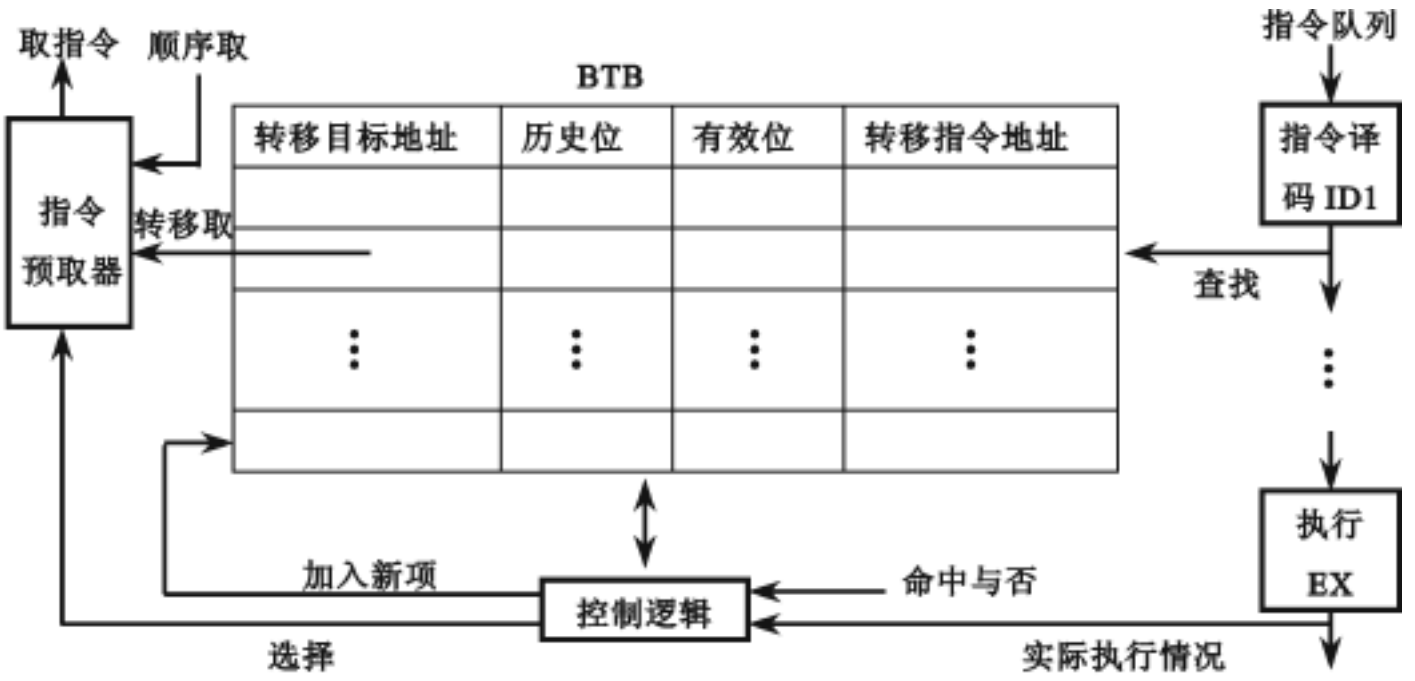


图 5.41 Pentium 微处理器的转移目标缓冲器(BTB)逻辑结构

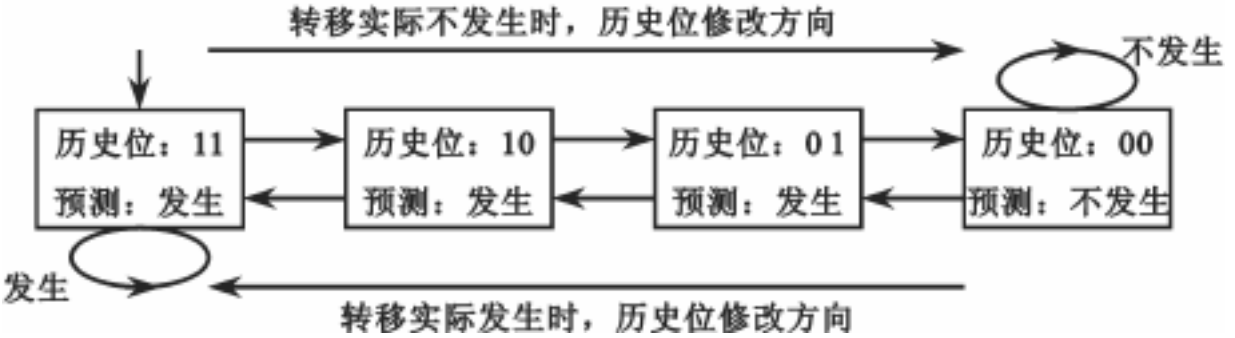


图 5.42 Pentium 微处理器 BTB 历史位状态转换

指令执行(EX)段的预测验证及修改阶段。

5.5.4 Pentium Ⅱ/ Ⅲ微处理器中动态执行技术

Intel 对 Pentium Pro, Pentium Ⅱ/ Ⅲ处理器的核心结构进行了全新设计。处理器呈现给用户的仍是 IA(Intel 结构)指令集,保持与 x86 处理器(包括 Pentium 处理器)的兼容;处理器内部,将由存储器取来的 IA 指令翻译成 RISC 指令来执行,并将最后结果写回 IA 寄存器组。这种全新设计的核心结构成功地实现了动态执行技术,全面改善了 Intel 第三代 32 位处理器的超标量超流水的指令流水线性能。

本节先简要讨论动态执行技术的要点,然后简要介绍 Pentium Ⅱ处理器的流水线结构。

1. 动态执行技术概述

动态执行(Dynamic Execution)技术也称为随机推测执行(Nandomly Speculative Execution)技术,此技术可概括为:通过预测程序流来调整指令的执行,并且分析程序的数据流来选择指令执行的最佳顺序。具体而言,它由如下三项技术组成:

(1)多路分支预测(Multiple Branch Prediction)

利用先进的转移预测技术(预测正确率高达 90%),允许程序的几个分支流向同时在处理器中进行。这样,处理器在取指令时,还会在程序中寻找未来要执行的指令,加速了向处理器传递任务的过程,并为指令执行顺序的优化提供了可调度基础。

(2)数据流分析(Dataflow Analysis)

通过分析指令之间的数据相关性,产生优化的重排序的指令调度。处理器读取软件指令并经过译码后,判断该指令能否与其他指令一道处理,然后处理器分析这些指令的数据相关性和资源可用性,以优化的执行顺序高效地处理这些指令。

(3)推测执行(Speculative Execution)

将多个程序流向的指令序列,以调度好的优化顺序送往处理器的执行部件去执行,尽量保持多端口多功能的执行部件始终为"忙",以充分发挥此部件的效能。因为程序流向是建立在转移预测基础上的,因此指令序列的执行结果也只能作为预测结果而保留。一旦证实转移预测正确,已提前建立的预测结果立即变成最终结果并修改机器的状态。显然,推测执行可保证处理器的超标量流水线始终处于忙碌状态,加快了程序执行的速度,从而全面提高了处理器的性能。

下面结合图 5.43,简要介绍 Pentium Pro, Pentium Ⅱ/Ⅲ核心结构的新特征。

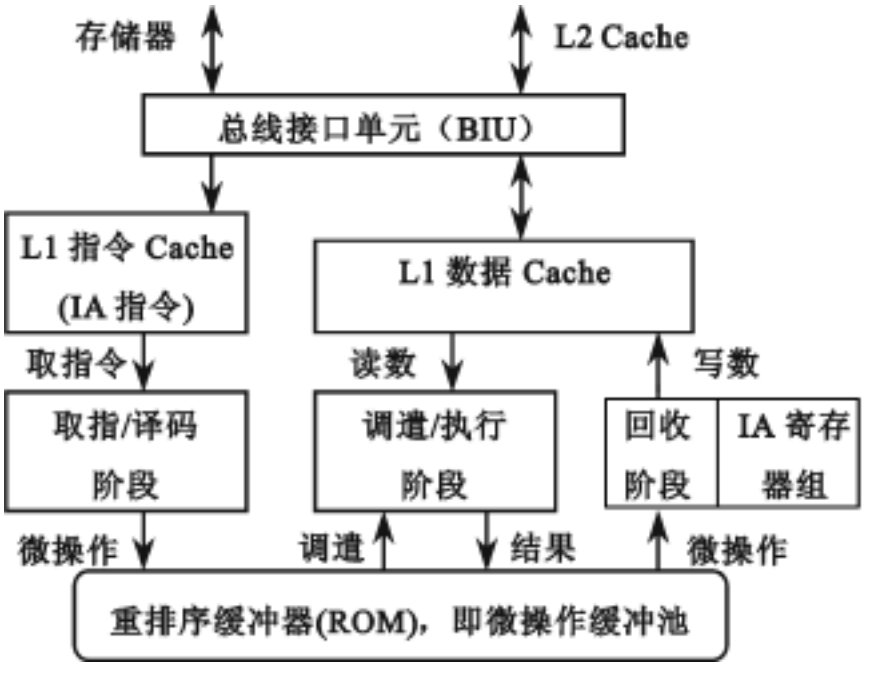


图 5.43 Pentium Pro 微处理器的核心结构

① 分立的双总线结构,即处理器的总线接口单元 BIU 通过分立的两条总线,分别与系统主存和级 2 Cache 连接。

②采用超标度为 3 的超标量结构,并将指令流水线由 5 段细分成 12 段,即它们的指令流水线是一个超标量与超流水线相结合的流水线。12 段流水线可分成三个大阶段:取指/ 译码段,调遣/ 执行段和回收段。

③ 取指/ 译码段由 L1 指令 Cache 取来的 IA 指令转换成 Intel 称为微操作(mop 或 uop)的 RISC 指令,这将大大简化以复杂著称的 IA 指令处理。

④ 核心结构以 RISC 指令缓冲池(Instruction Pool)为中心。多重程序流向的 IA 指令序列译码成 uop 后暂存于指令缓冲池,可多达 40 项。然后,以优化的重排序送往执行部件去执行,推测执行的结果也暂存于指令缓冲池,等待回收。

⑤ 采用寄存器换名(Renaming)策略。将 IA 指令使用的 IA 寄存器映射成微操作使用的 Pentium Pro 内部寄存器(数量、名称、长度,Intel 未公布),这样,可极大地消除指令的数据相关性。只有在推测执行指令序列前面的全部转移指令处理完毕,此指令序列已属正确程序顺序时,推测执行的结果才写回存储器和 IA 寄存器组,并删除缓冲池中此指令序列,这即是回收阶段。

⑥ 采用动态转移预测法和静态转移预测法相结合的两级转移预测技术。其动态转移预测法使用的 BTB 结构已增至 512 行,并采用 Yeh 的一种扩充算法,可得到 90 %以上的预测正确率。

2. Pentium Ⅱ超标量流水线结构

Pentium Ⅱ微处理器将一条指令的解释过程细分为 12 个子过程,即指令流水线由 12 个子功能段组成。各功能段的定义如图 5 .44 所示。

IFU1	IFU2	IFU3	DEC1	DEC2	RAT	ROB	DIS	EX	WB	RR	RET

图 5 .44 Pentium Ⅱ微处理器中流水线组成

下面结合图 5 .45 简要描述流水线各段的主要功能:

IFU1 取指单元段 1(Instruction Fetch Unit stage 1)

该段主要由 32 字节的“预取流式缓冲器”(Prefetch Streaming Buffer)构成。由 L1 指令 Cache 取一 32 字节的行,装入该缓冲器。

IFU2 取指单元段 2(Instruction Fetch Unit stage 2)

该段主要由“指令长度译码器”构成。预取流式缓冲器中的内容以 16 字节块向前传递,IFU2 的功能主要是在 16 字节块中标志指令边界,如果发现转移指令,即将此指令地址提交给 BTB 进行动态转移预测。

IFU3 取指单元段 3(Instruction Fetch Unit stage 3)

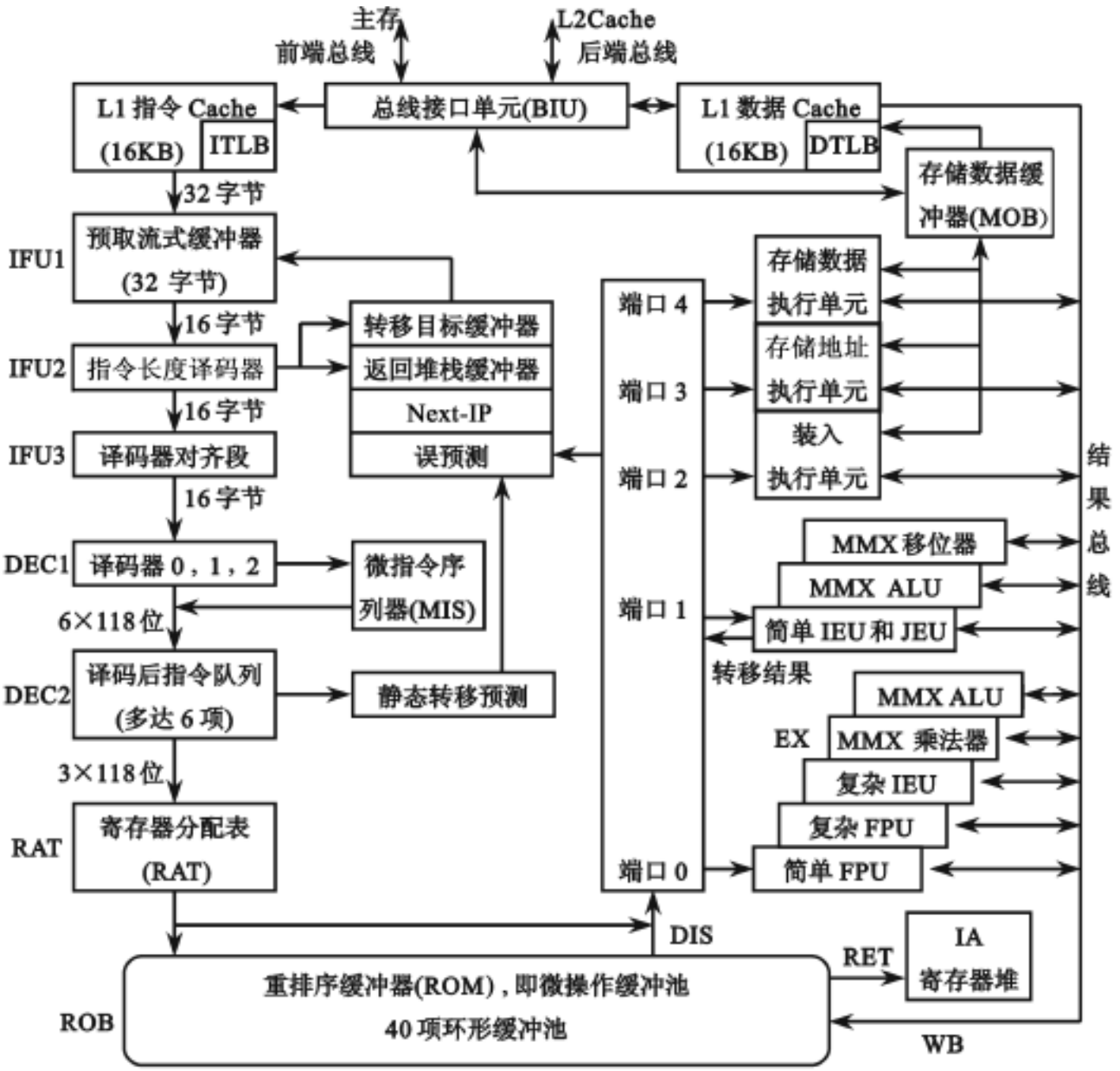


图 5.45　Pentium Ⅱ微处理器核心结构图

该段由“译码器对齐段”构成。因为下一段(DEC1)使用译码器 0(复杂)、译码器 1(简单)、译码器 2(简单)三个译码器;IFU3 的功能是旋转 16 字节块中的 3 条指令，使它们能按照复杂、简单，简单或简单、简单、复杂的次序同时递交给 DEC1。

DEC1 译码段 1(DECode stage 1)

DEC1 译码段 1 包括译码器 0、译码器 1 和译码器 2。其功能是将 IA 指令译码(或者说翻造)成 RISC 型的微操作(uop)。

译码器 0 可将一条复杂指令翻造成多达 4 个的 uop，译码器 1、2 只能各生成 1 个 uop。每个 uop 定长为 118 位(Intel 未公布 uop 的结构)。于是，DEC1 段的三个译码器可同时生成 6 个 uop。某些 IA 指令需要翻造成 5 个或更多的 uop，则 DEC1

段将它们提交给微指令序列器 MIS(Micro Instrnction Sequencer)。MIS 本质上是一个微代码 ROM。某些指令(如重复的串操作指令)经 MIS 可翻造成非常大的、重复的微操作序列。

DEC2 译码段 2(DECode stage 2)

该段是译码或翻造后的指令队列。在 DEC1 段由译码器 0,1,2 或 MIS 翻造生成的 uop 序列,每次最多 6 项送往 DEC2 段,在译码后指令队列 DIQ(Decoded Instruction Queue)中按原始程序顺序排队。并且,若在这些已排队的 uop 中发现转移型 uop 并且为 BTB 失效时,则将其提交给静态转移预测机构,这是第二级的转移预测。

RAT 寄存器别名表和分配器段(Register Alias Table and Allocator stage)

此段每次由 DIQ 按程序顺序取 3 项 uop,检查 uop 中是否使用 IA 寄存器,若有将其换名成 Pentium Pro 内部的(隐藏的)寄存器,然后将其送往微操作缓冲池 ROB。

IA 的通用寄存器只有 16 个,即整数的 EAX,EBX,ECX,EDX,ESl,EDI,EBP,ESP,和浮点数的 FR0～FR7。这样少的通用寄存器不仅编程不太方便(这种事实已无法改变),而且先后两条指令使用同一寄存器很易产生 WAR,WAW 和 RAW 数据相关,不利于超标量流水线的执行。现将 IA 寄存器换名成 Pentium Pro 内部寄存器(约有 40 个之多),极大地减少了指令间的数据相关。

ROB 重排序缓冲器段(ReOrder Buffer stage)

RAT 按程序顺序以每时钟 3 个 uop 的速率,将 uop 序列送入重排序缓冲器 ROB,即微操作缓冲池。ROB 是一个 40 项的环形队列缓冲器,它有两个指针:一个是缓冲器首指针(Start-of-Buffer Pointer),一个是缓冲器尾指针(End-of-Buffer Pointer)。尾指针是存放指针,首指针是回收指针。开始的时候首指针同于尾指针,意为缓冲器空,每存放一个 uop,尾指针即增 1,而首指针指向最"老"存入的 uop 项,等待执行完毕后的回收。

ROB 每项有状态位,登记此 uop 的当前运行状态,例如是否已调度到保留站,是否已派遣到相应执行单元端口,是否正在执行单元中执行,是否执行已完成正在将结果返回 ROB 项,是否正在检查结果(是否有误),是否标志回收就绪,是否已回收而为空项。

DIS 派遣段(Dispatch stage)

在 uop 已放入微操作缓冲池(ROB)后,保留站(RS)能以任何顺序由缓冲池拷贝多个微操作并派遣到相应的执行单元端口。这是一个乱序过程 OOO(Out-Of-Order)。调度所遵循的基本原则是,uop 的操作数已就绪并且相应的执行单元可用。其中,操作数已就绪还表示已排除指令间的 RAW 数据相关,相应的执行单元可用表示已排除指令间的结构相关(即资源冲突)。

EX 执行段(Execution stage)

该段的主要功能是执行 uop。保留站有端口 0～4。其中,端口 0 有 5 个执行单元,端口 1 有 3 个执行单元。它们完成全部的简单或复杂的整数运算、浮点运算以及多媒体扩展(MMX)功能的运算,还有一个转移执行单元(JEU)处理转移 uop。转移结果,即是否实际发生转移,除返回 ROB 外还要返回给 BTB。

端口 2 的装入执行单元负责生成读数据的存储器地址,存储器读指令只产生一个微操作。存储器写指令产生两个微操作,一个微操作送往端口 3 的存储地址执行单元用于生成写入的存储器位置,一个微操作送往端口 4 的存储数据执行单元用于生成写入数据。写数据及其存储器地址同时送往存储顺序缓冲器 MOB。然后,再以严格的程序顺序写回到 L1 数据 Cache(或 L2 Cache 以至主存)。

执行所需的处理器时钟周期数是微操作指定的,大多数微操作只需 1 个时钟周期。

WB 写回段(Write Back stage)

uop 执行结果写回 ROB 项并进行错误检查,这包括对由 L1 数据 Cache(或 L2 Cache)读入数据的 ECC 检查和修正。同 DIS,EX 段一样,WB 段也是一个乱序过程,但要避免出现 WAR 和 WAW 数据相关。

RR 回收就绪段(Retirement Ready stage)

当 uop 执行结果写回 ROB 项且结果无误时,RR 段逻辑判测它的上游的转移指令是否已全部解决。若已全部解决,则按程序顺序以 IA 指令为单位,标志它相应的 uop 已回收就绪,这是一个有序过程。

RET 回收段(RETirement stage)

按程序顺序以 IA 指令为单位,每时钟回收 3 个 uop。即将已回收就绪的 IA 指令对应 uop 的结果,最终写回 IA 寄存器集以及设置 EFLAGS 标记;并通知 MOB 将已回收就绪的存储器写指令实际完成,具体写入 L1 数据 Cache(或 L2 Cache 以至主存)。这也是一个有序过程。被回收的 uop 由 ROB 中删除,该 ROB 项变为空项,缓冲器的首指针增量。

综上所述,12 段的超流水线中的前 7 段(IFU1～ROB)是有序取指/ 译码阶段,中间 3 段(DIS,EX,WB)是乱序的调度、派遣/ 执行阶段,最后 2 段(RR,RET)是有序的回收阶段。有序的取指/ 译码阶段每次能同时译码 3 条 IA 指令,产生最多为 6 个的 RISC 型 uop,以每时钟周期 3 个 uop 的速率按序送往微操作缓冲池。乱序的调遣/ 执行阶段以优化的无结构相关、无数据相关的重排顺序执行指令,保留站的 5 个端口可同时执行 5 个 uop。最后以有序的回收阶段保证程序最终结果的正确性,RET 段以每时钟可回收 3 个 uop 的速率工作。因此,可以说 Pentium Ⅱ的指令流水线是超标度为 3 的流水线。采用动态转移预测与静态转移预测的两级预测,使预测正确率高达 90%以上,从而极大地减小了由于转移(控制相关)可能给超标量超流水线带来的性能损失。

5.6 向量流水技术

向量流水技术是向量数据表示与流水技术的结合。

在前面所讨论的流水技术中我们已经知道,为了获取高的流水处理性能,应该设法增加流水线中的段数,即记水深度。或者设法在每个时钟周期能同时启动多条指令。但在实际应用中要使标量流水机的性能有进一步的提高,通常受到以下两个因素的约束:

一是流水线工作的时钟周期不可能取得很短。加深流水深度可以使时钟周期缩短,但流水深度的增加会使流水线中出现相关性的可能性也增加,造成较大比例的断流,而为等待相关消除就需要更多的时钟周期,这导致每条指令执行所需的时钟周期数的增加。

二是取指令及译码的速率受限。即在一个时钟周期内最多只能启动一条指令,通常称为 Flynn 瓶颈。

本章所讨论的向量流水处理在一定程度上将不受上述两个因素的约束。

5.6.1 向量流水的基本概念

1. 向量流水处理的基本特点

向量流水处理具有以下基本特点:

① 在向量操作中,每个当前结果向量元素的计算与以前结果向量元素的计算是相互独立的,很少发生数据相关,这就允许向量流水线有较深的深度。

② 一条向量指令相当于一个标量循环,从而可降低对指令访问带宽的要求。此外,这也消除了由循环转移可能引起的控制相关。

③ 若向量指令所要访问的向量元素均相邻,则可以在交叉存储体中高速地依次访问它们。由于一个向量中通常含有多个元素,因此对存储器访问的延迟平均到每个元素上,其访存等待时间开销是较小的。

上述的这些特点使得对相同数量的数据项进行操作时,向量操作要比一串标量指令操作更快,此外,向量流水机还可使访存和有效地址计算流水化,高档的向量机还允许多个向量操作同时进行,从而可开发对不同元素进行多个向量操作的并行性。

2. 向量处理方式

向量处理机中对向量的各种运算可以采用不同的加工处理方式,但最为有效的加工方式应该尽量避免出现数据相关和向量操作功能的切换。下面以向量运算 D = A × (B + C)为例,讨论几种加工方式对向量运算的影响,其中,A,B,C 和 D 都有是长度为 N 的向量。

(1)横向处理方式

这种处理方式过程如图 5 .46(a)所示。

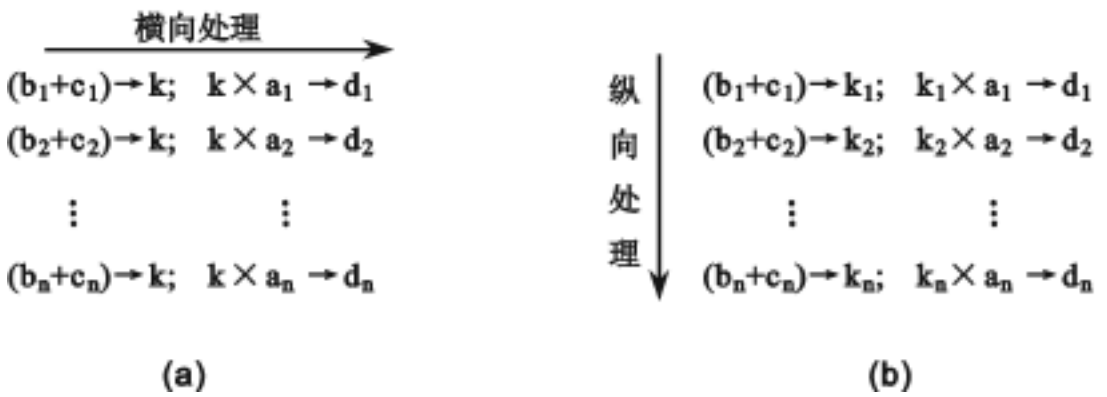


图 5 .46　向量处理方式

横向处理方式具有如下特点:

① 只需要 1 个暂存单元 k;因为每个分量的完成过程可细分为 $(b_i + c_i)\rightarrow k; k\times a_i\rightarrow d_i$。

② 每个向量元素在加、乘过程中都有数据相关,一共出现了 N 次相关。

③ 当使用静态流水线时,流水线功能需切换 2N 次。

因此,这种方式不适合于向量流水处理。

(2)纵向处理方式

这种处理方式过程如图 4 .46(b)所示。

显然,由图 5 .46(b)可见,纵向处理方式具有如下特点:

① 需要 1 个中间向量 K,其长度与运算向量相同。

② 数据相关只在两指令间出现一次,因为其运算过程可表示为 K = B + C, D = K × A。

③ 当使用静态流水线时,只有一次功能切换。

这种向量处理方式在存储器—存储器型向量处理机中实现。

(3)纵、横向处理方式

不管是存储器—存储器型向量处理机,还是寄存器—寄存器型向量处理机,它们所能表示的向量长度是有限的。当需要处理(运算)的向量长度 N 大于实际机器所能表示的向量长度时,就需要对处理向量长度 N 进行分组,然后再进行处理。

设用户需处理的向量长度为 N,向量处理机所能表示的向量长度为 n,而且有 n < N,则可将待处理的向量分成 k′组,每组元素个数为 n,则有 N = k′n + r。

向量处理过程是组内纵向处理,组间为横向处理,故称之为纵横向处理。CRAY-1 等巨型计算机都是采用这种处理方式工作的。处理过程示意如图 5 .47 所示。

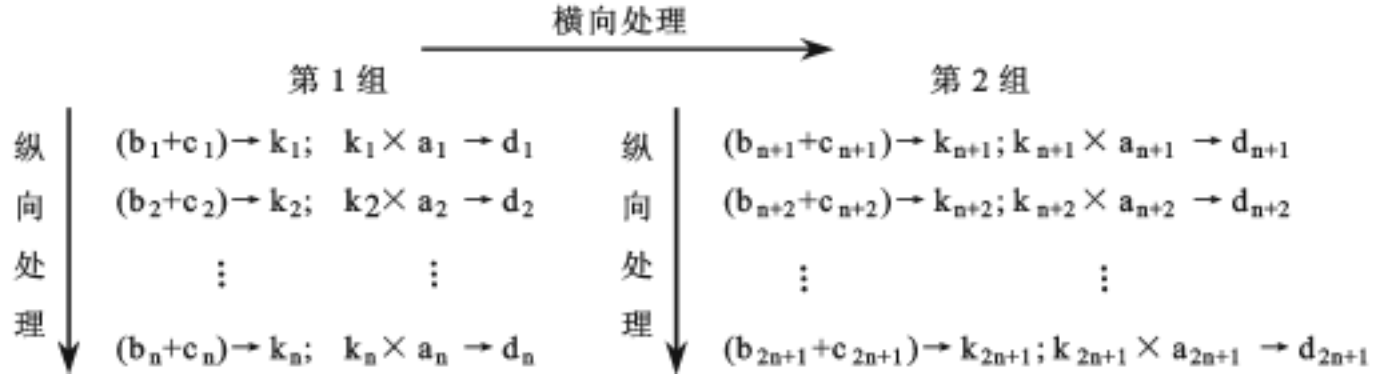


图 5.47 向量的纵、横向处理示意图(共完成 k'+ 1 组)

3. 向量处理机的分类

向量机处理机按照向量操作对象及结果主要存放在寄存器中还是存放在存储器中,可分为两大类:

(1)存储器—存储器型向量处理机

这种向量处理机的主要特点是,向量操作的原向量都取自主存,操作生成的结果向量也存放在主存中。最先的向量处理机都是存储器—存储器型的,如 TI 公司的 ASC(1972 年),CDC 公司的 STAR-100(1973 年)和 CYBER-205(1980 年)等。

(2)寄存器—寄存器型向量处理机。这种向量处理机的主要特点是,向量操作的原向量和结果向量都取自或存放到向量寄存器中。如 CRAY 公司研制生产的 CRAY-1 向量处理机(1976 年),日本富士通公司的 FACOM 系列向量机,日立公司的 S810/ 820 向量机以及我国的银河向量机等。

4. 向量访问步长及控制

连续访问的向量元素之间在存储器地址上具有相等的间隔,这个相等的间隔称为向量访问步长。

对于二维或多维向量元素,它们的存储必须通过一种映射方式将其存放在一维存储器空间中。例如,对于一个二维向量,若存储顺序是以行为主。当以行为主顺序访问时,访问顺序与存储顺序相同,即访问元素之间在地址上是连续的;当以列为主顺序访问时,而访问元素之间在地址上是不连续的,元素存储地址上是等间隔的。

那么,在向量处理机中是如何支持这种跨步访问的呢?

① 在存储器—存储器型向量处理机中,利用软件重新排序方法,将存储器中的元素按照访问顺序重新排序,以支持访问顺序。

② 在寄存器—寄存器型向量处理机中,利用硬件重组的方法,自动调整存储器中被访问元素,支持这种跨步访问。

5.6.2 CRAY-1 型向量流水处理机

向量流水处理机的结构因具体机器不同相互有所不同。CRAY-1 型向量处理机是美国 CRAY 公司于 1976 年研制完成的寄存器—寄存器型向量处理机。

1. CRAY-1 型向量处理机结构

CRAY-1 型向量处理机是由中央处理机、诊断维护控制处理机、大容量磁盘存储子系统、前端处理机组成的功能分布异构型多处理机系统。中央处理机的控制部分中有总容量为 256 个 16 位的指令缓冲器,分成 4 组,每组 64 个。图 5.48 为中央处理机向量流水处理部分结构。

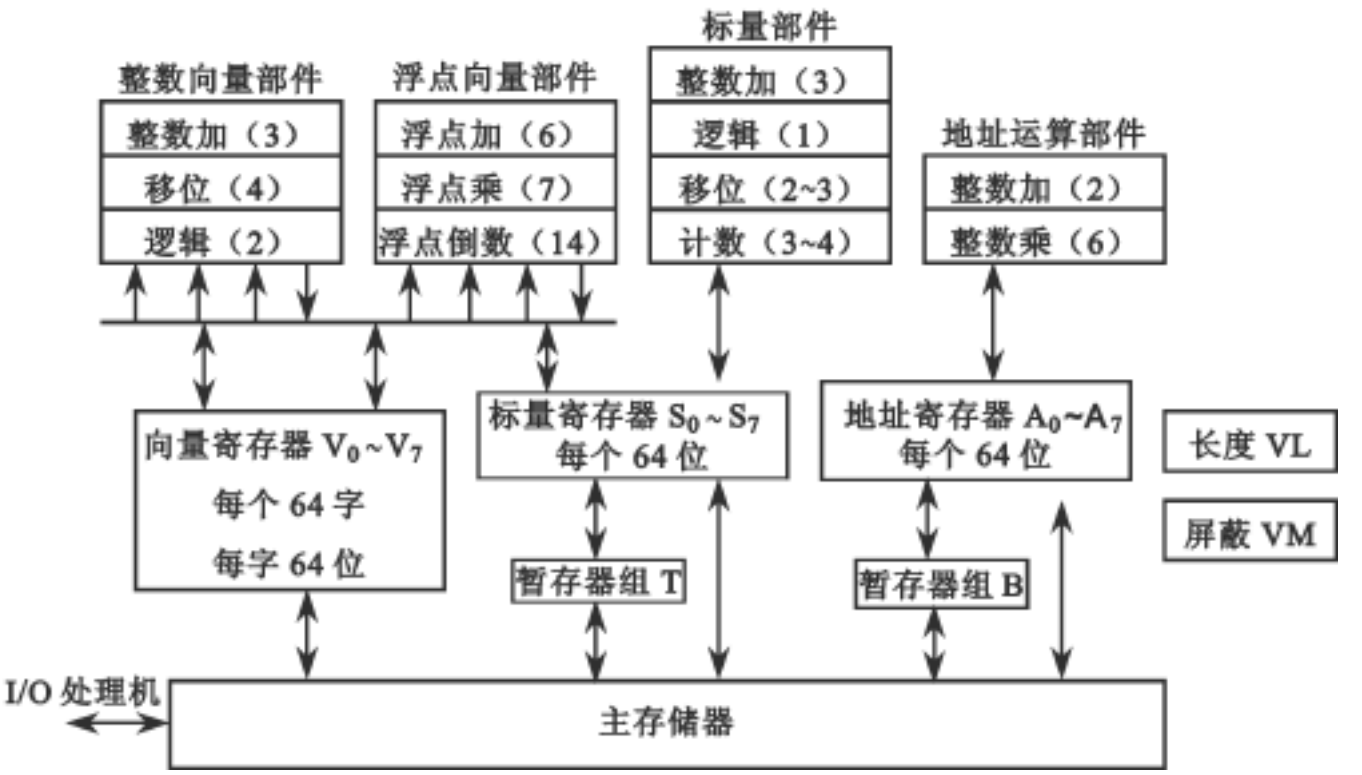


图 5.48 CRAY-1 向量处理机结构图

(1)三类基本寄存器

8 个向量寄存器 $V_0 \sim V_7$。每个向量寄存器的长度为 N = 64(每个寄存器 64 字),即每个向量寄存器可存放 64 个分量元素,每个字长度为 64 位。

$V_0 \sim V_7$ 可直接与 6 个向量运算(整数和浮点)部件、主存储器之间交换数据。$V_0 \sim V_7$ 中的任何一个寄存器都可作为向量运算中的源、目的操作数寄存器。如指令:

ADDV V_0, V_1 ; $V_0 + V_1 \rightarrow V_0$

8 个标量寄存器 $S_0 \sim S_7$。每个标量寄存器长度为 64 位,即可直接存放 64 位标量。它可为 6 个向量运算部件、4 个标量运算部件提供运算数据和存放运算结果,并

能通过中间暂存器 T 或直接与主存储器交换数据。

8 个地址寄存器 A_0～A_7。每个地址寄存器长度为 24 位,为地址运算提供操作数。通过中间暂存器或直接与主存储器之间交换数据。

(2)两组中间寄存器

中间寄存器组 T,它由 64 个 64 位的寄存器组成,为标量寄存器与主存储器之间交换数据起到缓冲作用。

中间寄存器组 B,它由 64 个 24 位的寄存器组成,在地址寄存器与主存储器之间交换数据起到缓冲作用。

(3)两个控制寄存器

向量长度寄存器 VL,它由 6 位组成,用于存放向量运算中实际参加运算元素个数。(VL)≤64,当实际参与运算的向量长度大于 64 时,必须将其分组后再运算。

向量屏蔽寄存器 VM,是一 64 位的寄存器,每一位对应于一向量元素,控制其是否实际参与向量运算。$VM_i = 0$ 时,表示屏蔽;$VM_i = 1$ 时,表示允许。

2. CRAY-1 型向量处理机的 4 类指令

CRAY-1 有标量类和向量类指令共 128 条,其中 4 种向量指令如图 5 .49 所示。

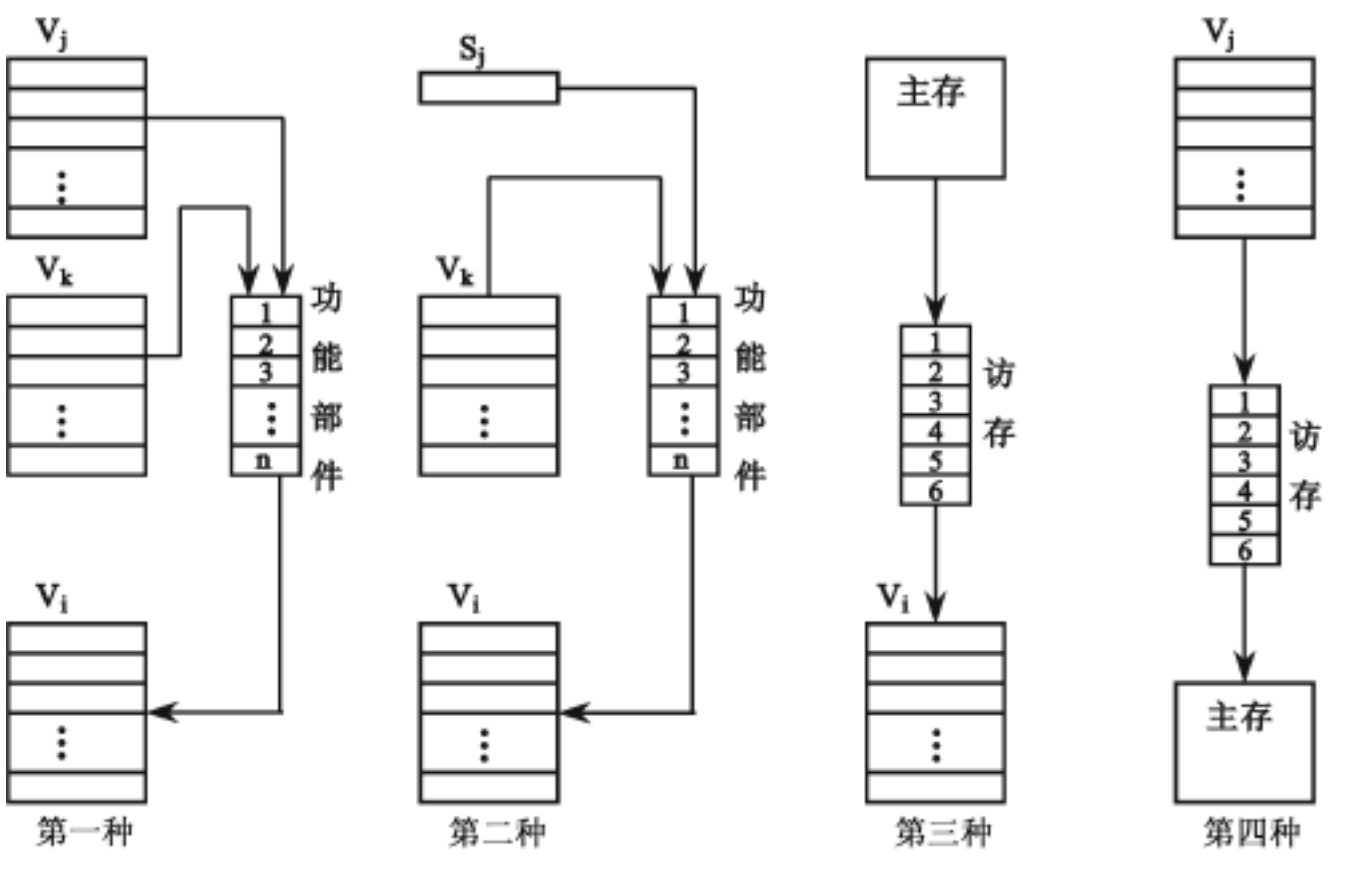


图 5 .49 CRAY-1 向量处理机的 4 种向量指令

第一种是两个参与运算的源向量分别取自两个向量寄存器 V_j, V_k,结果送入向量寄存器 V_i。如向量运算 $V_1 + V_2 \rightarrow V_3$ 属于此类,其功能部件为加法部件。

第二种是一个标量与一个向量运算,结果为一向量。它与第一种指令的差别只在于它的一个操作数取自标量寄存器 S_j。

大多数向量指令都属于这两种。由于它们不是由主存而是由向量寄存器组取得操作数,从而减轻了对主存数据流量要求高的压力,所以流水速度可以很高。这里的 n 是通过流水线功能部件的时钟数。

第三、四种用于控制主存与 V_i 向量寄存器组之间的成组数据传送。访存流水线建立时间为 6 拍。

5.6.3 增强向量处理性能的方法

本小节讨论几种改善向量处理性能的方法。第一种是采用多功能部件使它们并行工作;第二种是加快一串相关指令的操作速度,又称为链接技术,这种技术首先在 CRAY-1 型向量处理机中得到应用,目前的向量处理机都支持这种技术;最后两种主要是为了增加能以向量方式操作的循环类型,这两种方法在许多向量处理机中采用。

1. 多功能部件的并行操作

在向量处理机中,为了加快向量处理的速度,通常采用多个相对独立的功能部件并行工作。如前面所讲述的在 CRAY-1 型向量处理机中,共有四组 12 个单功能流水部件,这些功能部件都是相互独立的,只要它们满足一定的约束条件,可以并行工作。

多功能部件可并行工作的约束条件:

(1)不存在向量寄存器的使用冲突

例如,有些向量运算指令功能可描述为 $V_1 + V_2 \rightarrow V_3$ 和 $V_4 \times V_2 \rightarrow V_5$,由于这两条指令操作中都使用了同一向量寄存器 V_2,因此,这两条指令不能并行(同时)执行。这是因为两条指令中使用 V_2 时的首元素下标可能不相同,向量运算的长度也有可能不相同。

(2)不存在功能部件的使用冲突

例如,有些向量运算指令功能可描述为 $V_1 + V_2 \rightarrow V_3$ 和 $V_4 + V_4 \rightarrow V_6$,如果这两条指令的操作都是浮点运算,那么它们将使用同一浮点加法功能部件,从而引起功能部件冲突。

在一组向量运算指令中,如果这些指令既不存在向量寄存器的使用冲突,也不存在功能部件的使用冲突,则这些指令称为一个指令编队。也就是说,在一个指令编队中的所有指令,是可以并行执行的。

2. 链接技术

利用向量指令间存在的先写后读的数据相关性来加快向量指令序列执行速度的

技术称为链接技术。链接技术实质上是第四章中所讨论的标量流水定向传送方法在向量寄存器中的应用。例如,对于如下的向量加和向量乘的操作:

ADDV V_1, V_2, V_3 ; $V_3 + V_2 \rightarrow V_1$

MULTV V_4, V_1, V_5 ; $V_1 \times V_5 \rightarrow V_4$

由于这两条指令对 V_1 向量寄存器存在先写后读相关,通常必须等加法指令做完后才可开始乘法指令,但如果使向量寄存器(例中的 V_1)在同 1 时钟周期内,既接收一个功能部件送来的运算结果,又可把这一结果作为下一个向量指令运算所需的源操作数送给另一个功能部件,那就可使这两个部件链接起来进行操作。通常把这种链接称为超级向量操作。当链接进入充分流水操作状态后,在 1 个时钟周期内就可同时获取两个操作结果。在 CRAY-1 中,计算机能自动检查每一条向量指令是否可与它的前 1 条指令形成链操作。若满足链接条件时,便在前 1 条指令的第一个运算分量结果到达作为本条指令的源操作数时,立即启动本指令工作而形成链。

下面以 CRAY-1 为例,说明多功能部件并行操作和链接操作,如进行向量运算:

$D = A \times (B + C)$

假设向量长度 $N \leq 64$,且 B 和 C 已由存储器取至 V_0 和 V_1,则下面 3 条向量指令就可完成上述运算:

LD V_3, A ; $A \rightarrow V_3$

ADDV V_2, V_0, V_1 ; $V_1 + V_0 \rightarrow V_2$

MULTV V_4, V_2, V_3 ; $V_3 \times V_2 \rightarrow V_4$

第一、二条指令间因既无向量寄存器使用冲突,也无功能部件使用冲突,因而可并行执行。第三条指令与第一、二条指令均存在先写后读相关冲突,因而可将第三条指令与第一、二条指令链接执行,如图 5 .50 所示。

CRAY-1 启动访存、把元素送往功能部件、将结果存入某向量寄存器都需要有 1 拍的传送延迟,那么完成上述三条指令功能分别按三种不同的执行方式时总的时间如下:

(1)三条指令顺序(串行)执行时,总的完成时间为:

T1 = [1 + 6 + 1 + (N - 1)]{第一条访存指令的完成时间} + [1 + 6 + 1 + (N - 1)]{第二条加法指令的完成时间} + [1 + 7 + 1 + (N - 1)]{第三条乘法指令的完成时间}

= 3N + 22(拍)

(2)前两条指令并行执行完后,再执行第三条指令,完成时间为:

T2 = [1 + 6 + 1 + (N - 1)]{前两条指令并行执行时的完成时间} + [1 + 7 + 1 + (N - 1)]{第三条乘法指令的完成时间}

= 2N + 15(拍)

(3)前两条指令并行,再与第三条指令链接,完成时间为:

T3 = (1 + 6 + 1) + (1 + 7 + 1) + N - 1

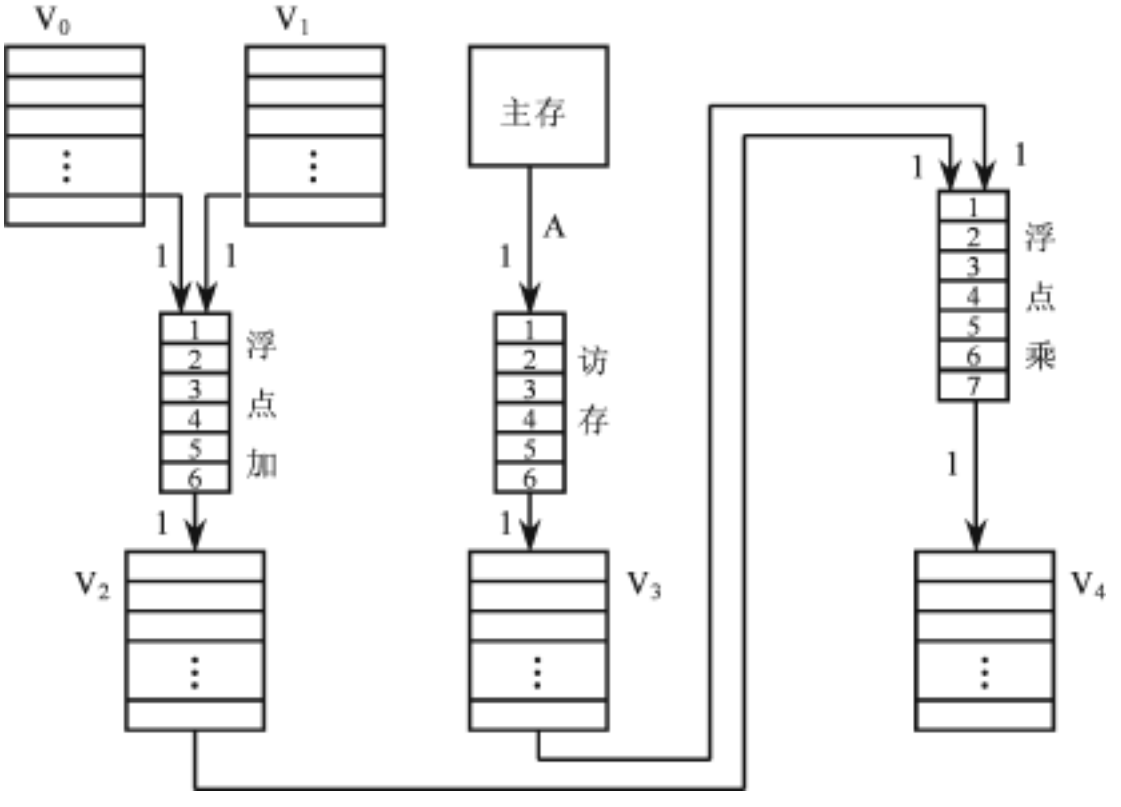


图 5.50　向量的并行和链接操作

= N + 17(拍)

可见,采用并行加链接技术的加速效率还是较高的。但要实现链接除了前述的条件之外,还应注意以下三个问题:一是链接的时间,即只有当前一指令的第一个结果分量送入结果向量寄存器的那一个时钟周期方可链接,若错过该时刻就无法进行链接,只有等前一向量指令全部执行完毕,释放向量寄存器资源后才能执行后面的指令。二是当一条向量指令的两个源操作数分别是两条先行指令的结果寄存器时,要求先行的两条指令产生运算结果的时间必须相等,即要求有关功能部件的延迟时间相等(如上例中的访存和浮点加功能部件延时均为 6 个时间单位)。三是要求两条先行向量运算指令的向量长度必须相等,否则就不可能链接。针对不同的向量机,可能对链接还有其他特殊限制。

3. 条件语句的加速处理方法

当程序中含有条件执行语句时,通常会使向量处理的优点无法发挥,如对于如下的循环:

```
do   100   i = 1, 64
if   A(i) ≠ 0   then
     A(i) = A(i) - B(i)
endif
```

```
100   continue
```

由于要执行条件语句,使循环体无法向量化,但若采用向量屏蔽控制技术,使减法只有当 A(i)不为 0 时才执行,就可使上述循环向量化。

这里的关键是要生成一个屏蔽向量,然后借助它来控制那些向量元素参加运算。屏蔽向量是通过向量测试得到的。本例中即是对 A(i)进行是否为 0 的测试,当被测试的 A(i)位元素等于 0 时,就使屏蔽向量中的对应位置 0,而 A(i)不为 0 时,就使相应位置 1。经上述测试得到的屏蔽向量,放在一个向量屏蔽寄存器 VM 中,在进行向量减法操作时,对应于屏蔽向量位为 0 的那些目的向量寄存器位的内容将不会改变。对屏蔽向量寄存器清零时,将使所有位都置为 1,因此,此后的向量指令将对所有的向量元素进行操作。下面给出实现上述循环的代码,假定向量 A 和 B 的起始地址存放在 R_a 和 R_b 中:

```
LV      V1, Ra          ;将向量 A 装入 V1
LV      V2, Rb          ;将向量 B 装入 V2
LD      F0, #0          ;将浮点 0 装入 F0
SENSV   F0, V1          ;若 V1(i)≠F0,则将 VMi 置为 1
SUBV    V1, V1, V2      ;在屏蔽向量控制下进行减法操作
CVM                     ;将屏蔽向量寄存器置为全 1
SV      Ra, V1          ;将结果存入 A
```

其中 SENSV 为屏蔽向量生成指令,CVM 为使屏蔽向量寄存器全置为 1 的指令。

屏蔽向量寄存器控制向量指令执行方法的缺点是:

① 执行时间没有减少。

② 在某些向量机中,屏蔽向量仅用来禁止向目的寄存器写入,而相应的向量操作实际仍是进行的,这就可能导致某些向量操作出现错误。如要进行的向量操作是:

```
if   A(i)≠0 then B(i) = B(i)/ A(i)
```

时,就会出现 B(i)被零除的错误。因此,比较好的实现方案是:根据屏蔽向量既禁止将结果写入目的寄存器,也禁止该操作的执行。

4. 向量归约的加速处理方法

归约(Reduction)操作一般难于向量化。因为对一个像一维数组那样的向量归约求值的结果将得到一个标量值。归约操作是递推(Recurrence)操作中的一个特例,可用如下的点积操作来说明这种归约:

```
dot = 0 .0
do 10 i = l, 64
10    dot = dot + A(i) ×B(i)
```

由于此循环存在迭代层间的数据相关,因而无法直接向量化。一个比较好的方

法是设法将此循环分成可向量化部分和递推部分,可将循环改写成如下形式:

```
do 10 i = 1,   64
10 dot(i) = A(i) × B(i)
dot l = 0 .0
do 20 i = 1,   64
20 dotl = dotl + dot(i)
```

这样已将原为标量变量的 dot 扩展成一个向量,显然第一部分循环可加以向量化,而第二部分则仍需用递推方法。对于这一递推求和部分可采用使叠加向量长度逐步缩短的方法加以完成。如首先相加两个 32 元素向量,然后再相加两个 16 元素向量等。这种技术称为递归折叠方法(Recursive Doubling)。这种方法显然比全部以标量方式的求和方法要快。例如,对于上述的第二部分递推循环,可用递归方法写成如下的代码:

```
len = 32
do 100 j = l, 6
do 10 i = 1, len
10 dot(i) = dot(i) + dot(i + len)
len = len/ 2
100 continue
```

其中 j 用来表示递推次数,对于向量长度为 32 的操作,只需递推 6 步就可在 dot(1)中得到所需的递推和,因而 j 取值由 1 到 6。当 j = 1 时,dot(1)与 dot(33)相加,dot(2)与 dot(34)相加,依次类推,直至 dot(32)与 dot(64)相加;当 j = 2 时,len 已由 32 缩短为 16,所以,dot(1)与 dot(17)相加,dot(2)与 dot(18)相加,直至 dot(16)与 dot(32)相加。当 j = 6 时,len 已折叠变为 1,因而只进行 dot(1)与 dot(2)相加,结果存入 dot(1)。这种递归折叠方法是一种加快向量归约操作的有效方法,因而在许多向量机上得到了应用。

习　　题

5 .1 解释下列术语

一次重叠　操作数相关　宏流水线　单功能流水线　多功能流水线
静态流水线　动态流水线　非线性流水　全局性相关　局部性相关
先行控制　先写后读相关　先读后写相关　写与写相关　不精确断点法
精确断点法　向量处理机　向量的流水处理
超标量处理机　超长指令字处理机　超流水线处理机

5 .2 假设一条指令的解释分为取指、分析与执行 3 段,每一段的时间为 Δt, 2Δt

和 3Δt。在下列各种情况下,分别写出连续执行 n 条指令所需要的时间表达式。

(1)顺序执行方式。

(2)仅“取指令”和“执行”重叠。

(3)“取指令”、“分析”和“执行”重叠。

(4)先行控制方式。

5.3 假设一条指令的解释分为取指、分析与执行 3 步,每步相应的时间为 $t_{取指}$、$t_{分析}$、$t_{执行}$。

(1)分别计算在下列几种情况下,执行完 100 条指令所需时间的一般关系式:

① 顺序方式;

② 仅“执行$_k$”与“取指$_{k+1}$”重叠;

③ 仅“执行$_k$”、“分析$_{k+1}$”、“取指$_{k+2}$”重叠。

(2) 分别在 $t_{取指} = t_{分析} = 2$, $t_{执行} = 1$; $t_{取指} = t_{执行} = 5$, $t_{分析} = 2$ 两种情况下,计算出上述各结果。

5.4 在一台单流水线多操作部件的处理机上执行下面的程序,取指令、指令译码各需要 1 个时钟周期,MOVE, ADDT 和 MUL 操作各需要 2 个、3 个和 4 个时钟周期。每个操作都在第 1 个时钟周期从寄存器中读取操作数,在最后 1 个时钟周期把运算结果写到通用寄存器中。

K: MOVE R_1, R_0 ; $(R_0) \rightarrow R_1$

K+1: MUL R_0, R_2, R_1 ; $(R_2) \times (R_1) \rightarrow R_0$

K+2: ADD R_0, R_2, R_3 ; $(R_2) + (R_3) \rightarrow R_0$

(1)就程序本身而言,可能有哪几种数据相关?

(2)在程序实际执行过程中,有哪几种数据相关会引起流水线停顿?

(3)画出指令执行过程的流水线时-空图,并计算执行完这 3 条指令共使用了多少个时钟周期?

5.5 某个流水线由 4 个功能部件组成,每个功能部件的延迟时间为 Δt,当输入 10 个数据后,间歇 5Δt,又输入 10 个数据,如此周期性地工作。求此时流水线的吞吐率,并画出其时-空图。

5.6 若有一个浮点乘法流水线如图 5.51(a)所示,其乘积可直接返回输入端或暂存于相应缓冲寄存器中,画出实现 A * B * C * D 的时-空图以及输入端的变化,并求出该流水线的吞吐率和效率;当流水线改为图 5.51(b)形式时,求实现同一计算时,该流水线的效率及吞吐率。

5.7 图 5.52 给出了一个非线性流水线。若 4 条指令依次间隔 2Δt 进入流水线,求出其实际的吞吐率和效率并画出其时-空图。如要加快流水,使流水线每隔 2Δt 流出一个结果,应减少哪个流水段本身经过的时间?应减少到多少,流水线方能满足要求?求出此时连续流入 4 条指令时的实际吞吐率和效率。

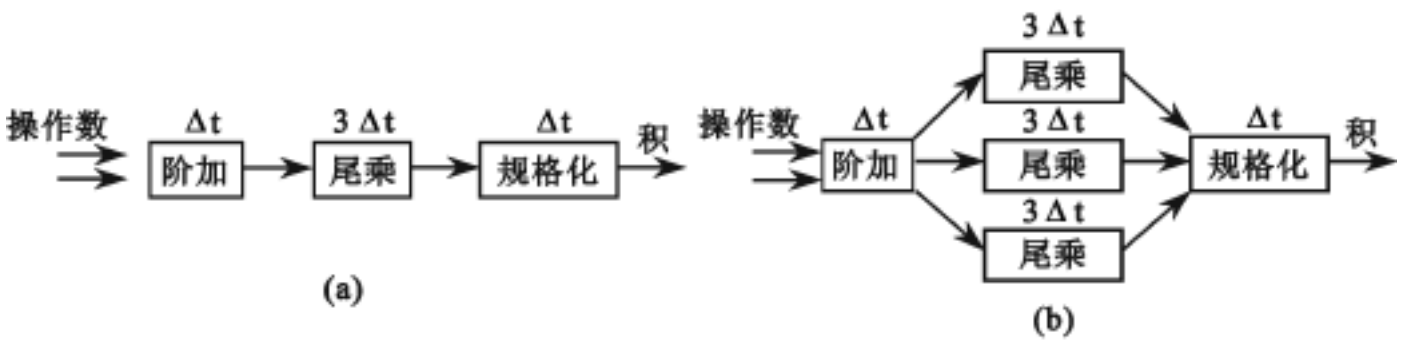


图 5 .51

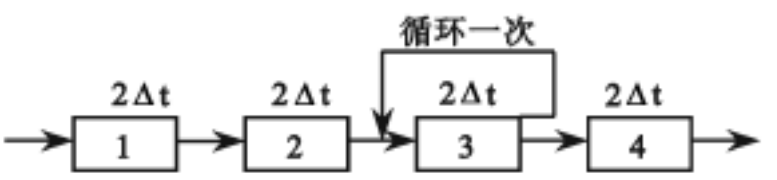


图 5 .52

5 .8 若有一静态多功能流水线分为 6 段,如图 5. 53 所示,其中乘法流水线由 1,2,3,6 段组成,加法流水线由 1,4,5,6 段组成,通过每段所需时间如图中所示。使用流水线时,要等某种功能(如加法)操作都处理完毕后才能转换成另一种功能(如乘法)。

若要计算: $A \times B = (a_1 + b_1) \times (a_2 + b_2) \times (a_3 + b_3)$

(1)在上述流水方式下,完成 A×B 需要多少时间?画出时-空图,并计算此流水线的使用效率和吞吐率。

(2)与顺序运算方式相比,加速比为多少?

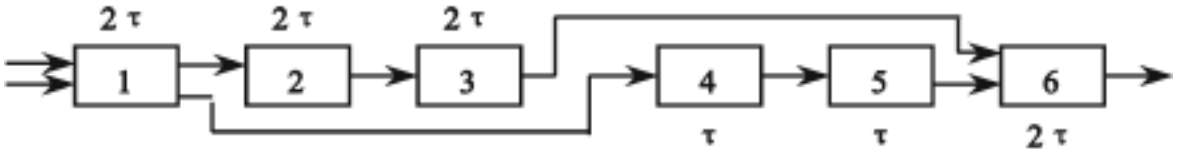


图 5 .53

5 .9 为提高流水线吞吐率,可以采用哪两种主要途径克服速度瓶颈?现有 3 段流水线,各段经过时间依次为 Δt,3Δt,Δt。为简化讨论,不考虑加“权”。

(1)分别计算在连续输入 3 条指令时和连续输入 30 条指令时的吞吐率和效率。

(2)按两种途径之一改进,画出你的流水线结构示意图,同时计算连续输入 3 条指令和连续输入 30 条指令情况下的吞吐率和效率。

(3)通过对(1),(2)两小题计算结果的比较,你得出什么有用的结论?

5.10 有一个双输入端的加/乘双功能静态流水线,由经过时间分别为 Δt,2Δt,2Δt,Δt 的 1,2,3,4 四个子过程构成。加法时按 1→2→4 连接,乘法时按 1→3→4 连接。流水线输出设有数据缓冲器,也可将数据直接回授到流水线输入端。现要执行:

$$A\times(B+C\times(D+E\times F))+G\times H$$

的运算,请对计算顺序进行变换,画出能获得尽可能高的吞吐率的流水时-空图;标出流水线输入、输出端的操作数变化情况;求出完成全部运算所需时间及此期间整个流水线的效率。如对流水线瓶颈子过程再细分,最少需多少时间可完成全部运算?若子过程 3 已无法再细分,只能采用并联方法改进,问流水线的效率为多少?

5.11 在一个 5 段的流水线处理机上需经 9 拍才能完成一个任务,其预约表如图 5.54 所示。

时间 t / 功能段	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
S_1	√								√
S_2		√	√						
S_3				√			√	√	
S_4				√	√				
S_5						√	√		

图 5.54

分别写出延迟禁止表 F、冲突向量 C;画出流水线状态图;求出最小平均延迟及流水线的最大吞吐率;此时其调度方案是什么?按此流水调度方案共输入 8 个任务,其实际吞吐率为多少?

5.12 有一个 5 段的流水线,其预约表如图 5.55 所示。

时间 t / 功能段	T_1	T_2	T_3	T_4	T_5	T_6	T_7
S_1	√						√
S_2		√			√		
S_3			√	√			
S_4				√			√
S_5					√	√	

图 5 .55

画出用冲突向量表示的流水线状态图;找出允许不等间隔调度时的最佳调度方案,其平均延迟时间是多少?若采用等间隔调度方案,其最少延迟时间是多少?

5 .13 一台非流水处理机 A 的工作时钟频率为 25MHz,它的平均 CPI 为 4。处理器 B 是 A 的改进型,它有 1 条 5 段的线性指令流水线。由于锁定电路延迟及时钟扭斜效应,它的工作时钟频率仅为 20MHz。问:

(1)若在 A 和 B 两个处理器上执行 100 条指令的程序,则处理器 B 对 A 的加速比是多少?

(2)在执行上述程序时,计算 A,B 处理器各自的 MIPS 速率是多少?

5 .14 现有向量 A 和向量 B,长度均为 8。请分别画出在下列 4 种不同结构的处理器上求点积 A×B 运行时的时-空图,并计算出获得最终结果所需时钟的最少拍数。设处理器中每个部件的输出均可直接送到任何部件的输入端或存入缓冲器中,不计其间的传送延迟,指令和源操作数均能连续提供。

(1)处理器有一个乘法部件和一个加法部件,不能同时工作,部件内部也只能以顺序方式工作,完成一次加法或乘法均需 5 拍。

(2)与(1)基本相同,只是乘法部件和加法部件可并行。

(3)处理器有一个乘、加两功能静态流水线,乘、加均由 5 个流水段构成,每段经过时间为 1 拍。

(4)处理器有乘、加两条流水线,可同时工作,各由 5 段构成,每段经过时间为 1 拍。

5 .15 若 IBM 360/ 91 浮点操作栈连续流出如下命令:

```
LD      F0 , FLB1      ; (FLB1 ) → F0
MD      F0 , FLB2      ; (FLB2 ) × ( F0 ) → F0
ADD     F0 , FLB3      ; (FLB3 ) + ( F0 ) → F0
STD     F0 , SDB1      ; ( F0 ) → DSB1
LD      F1 , FLB4      ; (FLB4 ) → F1
SUB     F1 , FLB5      ; (FLB5 ) — ( F1 ) → F1
```

STD F_1, SDB_2 ; (F_1) → SDB_2

写出 F_0, F_1, 各保存站及 SDB 的站号、忙位、数据内容的变化经过。

5.16 求向量 D = A × (B + C), 各向量元素个数均为 N, 参照 CRAY - 1 方式, 分解为下列 3 条向量指令:

① V_3 ← 存储器 {访存取 A 送入 V_3 寄存器组}

② V_2 ← V_0 + V_1 {B + C → K}

③ V_4 ← V_2 × V_3 {K × A → D}

当采用下列 3 种方式工作时, 各需多少拍才能得到全部结果?

(1)①, ②, ③串行执行;

(2)①和②并行执行完后, 再执行③;

(3) ①和②并行执行完后, 再与③链接。

5.17 在 CRAY-1 机上, 设向量长度均为 64; 所用浮点功能部件的执行时间分别为: 相加需 6 拍, 相乘需 7 拍, 求倒数近似值需 14 拍, 从存储器读数需 6 拍; 打入寄存器及启动功能部件各需 1 拍, 问下列各指令组, 组内的哪些指令可以链接? 哪些指令不可链接? 不能链接的原因是什么? 并分别计算出各指令组全部完成所需的拍数。

(1) V_0 ← 存储器
V_1 ← V_2 + V_3
V_4 ← V_5 × V_6

(2) V_2 ← V_0 × V_1
V_3 ← 存储器
V_4 ← V_2 + V_3

(3) V_0 ← 存储器
V_2 ← V_0 × V_1
V_3 ← V_2 + V_0
V_5 ← V_3 + V_4

(4) V_0 ← 存储器
V_1 ← 1/ V_0
V_3 ← V_1 × V_2
V_5 ← V_3 + V_4

5.18 设指令由取指、分析、执行 3 个子部件组成。每个子部件经过时间为 Δt, 连续执行 12 条指令。请分别画出在常规标量流水线处理机与度 m 均为 4 的超标量处理机、超长字处理机、超流水线处理机上工作的时-空图, 并分别计算出它们相对常规标量流水线处理机的加速比 Sp。

5.19 要计算两个 8 × 8 的矩阵相乘: C = A × B, 一种方案是在一个串行单处理器上进行运算, 其任何一种数学运算均在 10μs 中完成。另一种方案是在一个流水线处理器上进行运算, 该处理器有 K 个子过程, 每个子过程需经 (10/ K)μs 的延迟。对该流水线, 一条指令从流入到流出也需 10μs, 它可以进行任何一种数学运算, 在改变运算类型时, 必须让前一种运算全部完成。这两种处理器的存储、接收数据和判控制转移所花时间均可略去。置初始值采用自减方式完成, 且假定每个处理器都有足够的寄存器存放中间结果。请分别编写出适合于在串行单处理器上及在流水处理器上运行的高级语言程序。并分别计算下列各情况, 完成全部运算所需时间。

(1)串行单处理器;

(2)流水线处理器且 K = 7;

(3)**流水线处理器且** K = 5。

第六章 并行处理技术

并行处理技术是获取高性能计算的重要手段,计算机系统结构由低向高发展的过程,就是并行处理技术不断发展的过程。与器件的发展对提高计算机性能的作用相比,并行处理技术显得更重要。器件速度的改进受物理条件约束,不可能超过光速,而并行处理技术可以通过资源的重复来实现,其发展基本上是无穷尽的。因此,从这个意义上讲,它有着更为广阔的应用前景和将起到更为重要的作用。

6.1 并行处理技术的基本概念

所谓并行性是指在数值计算、数据处理、信息处理或人工智能求解过程中可能存在某些可以同时进行运算或操作的部分。开发并行性的目的是为了能进行并行处理,以提高计算机系统求解问题的效率。

并行性的双重含义是指同时性和并发性。所谓同时性是指两个或两个以上的事件在同一时刻发生的,而并发性则是指两个或两个以上的事件在同一段时间间隔内发生的。

并行处理是指一种相对于串行处理的信息处理方式,它着重开发计算过程中存在的并发事件。

在进行并行处理时,其每次处理的规模大小可能是不同的,这可用并行性粒度来表示。有关并行性粒度大小 G 可用以下公式表示(假定系统中共有 P 个处理器):

$$G = \frac{T_W}{T_C} = \frac{\sum_{i=1}^{P} t_{wi}}{\sum_{i=1}^{P} t_{ci}}$$

式中,分子为所有处理器进行计算时间的总和,而分母表示所有处理器的通信时间总和。当 T_C 较大时,G 就较小,表明处理粒度较细。当处理粒度较细时,显然此时处理器间的通信量将加大,相反,当粒度较粗时,通信量就较小。

并行性有不同的等级,而且从不同角度观察时会有不同的划分方法。程序执行过程通常可划分成以下 5 个等级:作业级、任务级、程序级(例行程序或子程序)、指令级和指令内的操作级。前面 3 级为粗粒度,主要是通过多处理机或多计算机系统实现,开发手段以软件为主,其中包括并行性算法分析、任务的调度与分配等;而后面 2

级为细粒度,主要是在单处理机中实现,开发手段以硬件为主,其中包括标量流水、超标量流水、超流水和超长指令字等。

并行性的开发途径有时间重叠、资源重复和资源共享。

时间重叠:在并行性中引入时间因素。即让多个处理过程在时间上相互错开,重叠地使用同一部件,以赢得速度,如指令的流水执行方式。

资源重复:在并行性中引入空间因素。即通过重复设置多个功能部件来提高处理性能或可靠性,如阵列处理机。

资源共享:利用软件的方法,让多个用户按一定的时间顺序轮流使用一套资源,以提高系统资源的利用率。

从计算机系统结构的角度出发,根据并行性的三个途径,从计算机系统由低性能向高性能发展过程中可以看出,并行性正从两个不同的角度向同一方向发展。

一方面在单处理机上,通过时间重叠、资源重复和资源共享,分别向异构型多处理机系统、同构型多处理机系统和分布式处理系统方向发展。

另一方面在多计算机系统中,通过功能专业化、多机互连、网络化等手段,分别向异构型多处理机系统、同构型多处理机系统和分布式多处理机系统的方向发展。

例如,在指令级内部,采用了多个子部件以流水方式执行指令,以提高单台计算机的执行速度。这一思想仍可应用于多台计算机组成的系统,如在高级语言向汇编语言转换过程中,是由编译程序完成翻译的。将翻译的全过程分为编译扫描、编译分析和目标程序生成三个子过程。为了提高效率,可由三台计算机组成流水方式执行。由于这三台计算机结构可以不一样(非对称型),故这样组成的多计算机系统称为异构型多计算机系统。

按照资源重复的思想,既然在一个计算机系统内可由多个相同的处理单元构成阵列处理机,那么用多台系统结构相同的计算机,也可以组成多计算机系统,这种系统结构称为同构型(对称型)多计算机系统。

按照资源共享的思想,单处理机采用多道程序和分时操作,并发展成为分布式多计算机系统。

6.2 SIMD 并行计算机(阵列处理机)

本节将讨论以 SIMD 方式工作、采用资源重复的并行性措施的阵列处理机。由于历史原因,习惯上也将阵列处理机(简称“阵列机”)称为并行处理机。我们将首先讨论它的基本结构,然后叙述它的主要特点,最后简短讨论适合于在阵列机上进行加工的并行算法。

6.2.1 阵列机的基本结构

阵列机通常由一个控制器(CU)、N 个处理器单元(PE)、M 个存储器模块(M)以

及一个互连网络部件(IN)所组成。由 CU 控制将指令广播给系统中的各个 PE,而所有活跃的 PE 将以同步方式执行相同的指令(单指令流),它们从相应的存储模块中取得自己所需的数据对象(多数据流)。IN 用来使各个 PE 之间或是 PE 和 M 之间实现方便的通信连接。IN 有时也称为对准(Alignment)或排列(Permutation)网络。有关互连网络的问题,将在下一节作进一步的讨论。

根据存储器模块是以分布方式存取还是集中方式存取,阵列机可分为两种基本结构:分布式存储器的阵列机和集中共享存储器的阵列机。

1. 分布式存储器的阵列机

分布式阵列机的基本结构如图 6 .1(a)所示。这种阵列机的主要结构特征如下:

(1)具有 N 个相同的处理单元 PE,它由处理器 Pi 和局部存储器 Mi 组成。只要数据分配得当,各个 Pi 主要将从自己的 Mi 中获取数据进行操作。各个 PE 将通过 IN 实现相互间必要的数据交换,因此,IN 是单向的。

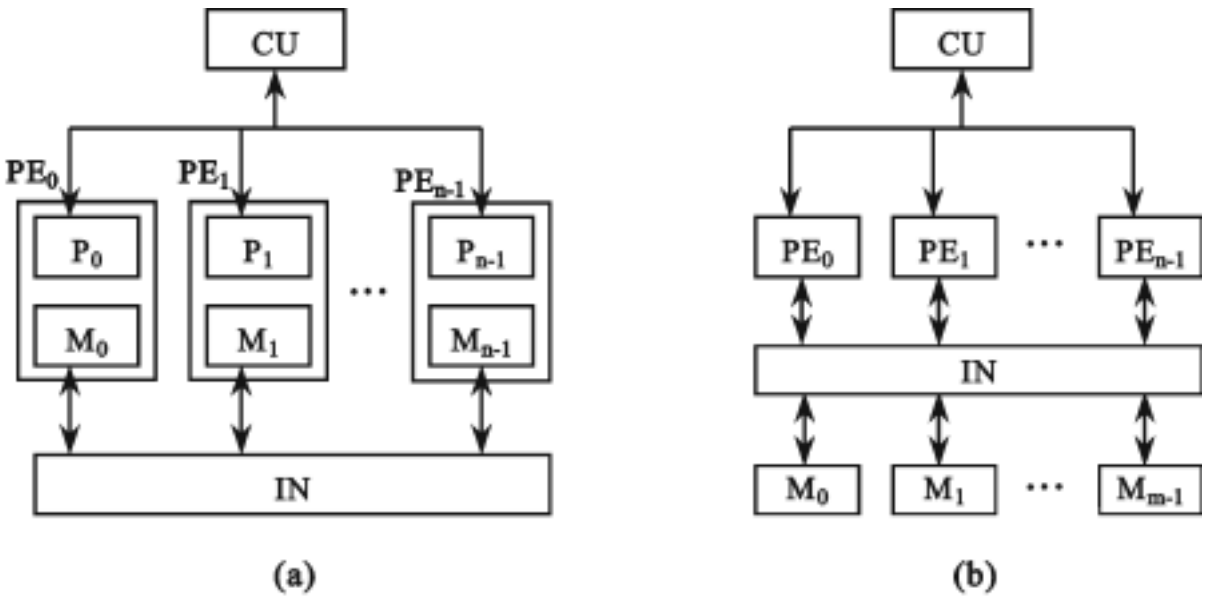


图 6 .1 阵列机的两种结构类型

(2)CU 中具有自己的存储器,以存放系统程序和用户程序,此外,它也可存放各个 PE 所需的共享数据。CU 的主要功能是对指令译码和判别它应在何处执行。对于标量或控制类指令,CU 本身中含有运算部件可以直接执行;若是向量指令,它就将此指令广播给各个 PE 去执行。

(3)各个 PE 同步执行来自 CU 的操作命令,但是并不一定每个操作非得所有 PE 都参加,CU 将对 PE 实行屏蔽控制,只有那些未被屏蔽的活跃 PE 才可参加操作。CU 还控制互连网络 IN,使各个 PE 之间通过 IN 实现相互之间必要的数据交换。当相互需要交换数据的两个 PE 不直接相连时,就需要经过它们之间的中间 PE 来完成连接。

2. 共享存储器的阵列机

图 6.2(b)中示出了这种类型的阵列机结构。它与图 6.2(a)中分布式存储器的阵列机结构的区别主要在于:

(1)每个 PE 没有局部存储器。存储器模块以集中形式为所有 PE(通过 IN)共享。

(2)互连网络受 CU 控制,用来构成 PE 和 M 之间的数据交换通路。要求互连网络具有同时连接 PE 到 M 或 M 到 PE 的双向性。系统中的一个 PE 可以与任何另一个 PE 实现数据交换(只要有任何一个存储模块同时与这两个 PE 相连接)。当两个需交换数据的 PE 之间没有共享的存储模块时,可能需要经过多次的传送之后,方可实现交换。

为了形式化地表示阵列机的特征,可用以下的参数加以描述:

$$C = (N, F, I, M)$$

式中:

N 为 PE 数;

F 为确定互连网络结构及连接拓扑的互连参数;

I 为指令集,它能进行标量、向量、数据传送通路操作和网络变换操作等;

M 为屏蔽方式集合,每一种方式将 PE 集合分为活跃和非活跃两类 PE 的子集。

这种描述模型为评价不同的阵列机结构提供了比较基础。

6.2.2 阵列机的主要特点

阵列机是以单指令流多数据流方式工作的,它具有以下特点:

① 它采用资源重复方法引入空间因素,即在系统中设置多个相同的处理单元开发并行性,这与利用时间重叠的流水线处理机是不一样的。此外,它是利用并行性中的同时性,而不是并发性,所有处理单元必须同时进行相同操作。

② 它是以某一类算法为背景的专用计算机。这是由于阵列机中通常都采用简单、规整的互连网络实现处理单元间的连接操作,从而限定了它所适用的求解算法类别。因此,对互连网络设计的研究就成为阵列机研究的重点之一。

③ 阵列机的研究必须与并行算法研究密切结合,以便使它的求解算法的适应性更强一些,应用面更广一些。

④ 从处理单元来看,由于都是相同的,因而可将阵列机看成是一个同构型并行机。但它的控制器实质上是一个标量处理机,而为了完成 I/O 操作以及操作系统的管理,尚需一个前端机,因此实际的阵列机系统是由上述三部分构成的一个异构型多处理机系统。

6.2.3 典型 SIMD 计算机举例

1. ILLIAC-Ⅳ阵列计算机(阵列机)

由美国宝来(Burroughs)公司和伊利诺斯大学 1965 年开始研制,并于 1972 年由宝来公司生产的 ILLIAC-Ⅳ阵列机是 SIMD 并行计算机的一个典型代表。它可以分为两大部分,即 ILLIAC-Ⅳ阵列和 ILLIAC-Ⅳ输入输出系统。实际上,它是由 3 种类型处理机组成的一个异构多机系统:一是专门用于数组运算的处理单元阵列;二是阵列控制器,它既是处理单元阵列的控制部分,又是一台相对独立的小型标量处理机;三是一台标准的宝来公司的 B6700 计算机,由它担负 ILLIAC-Ⅳ输入输出系统和操作系统管理功能,如图 6.2 所示。

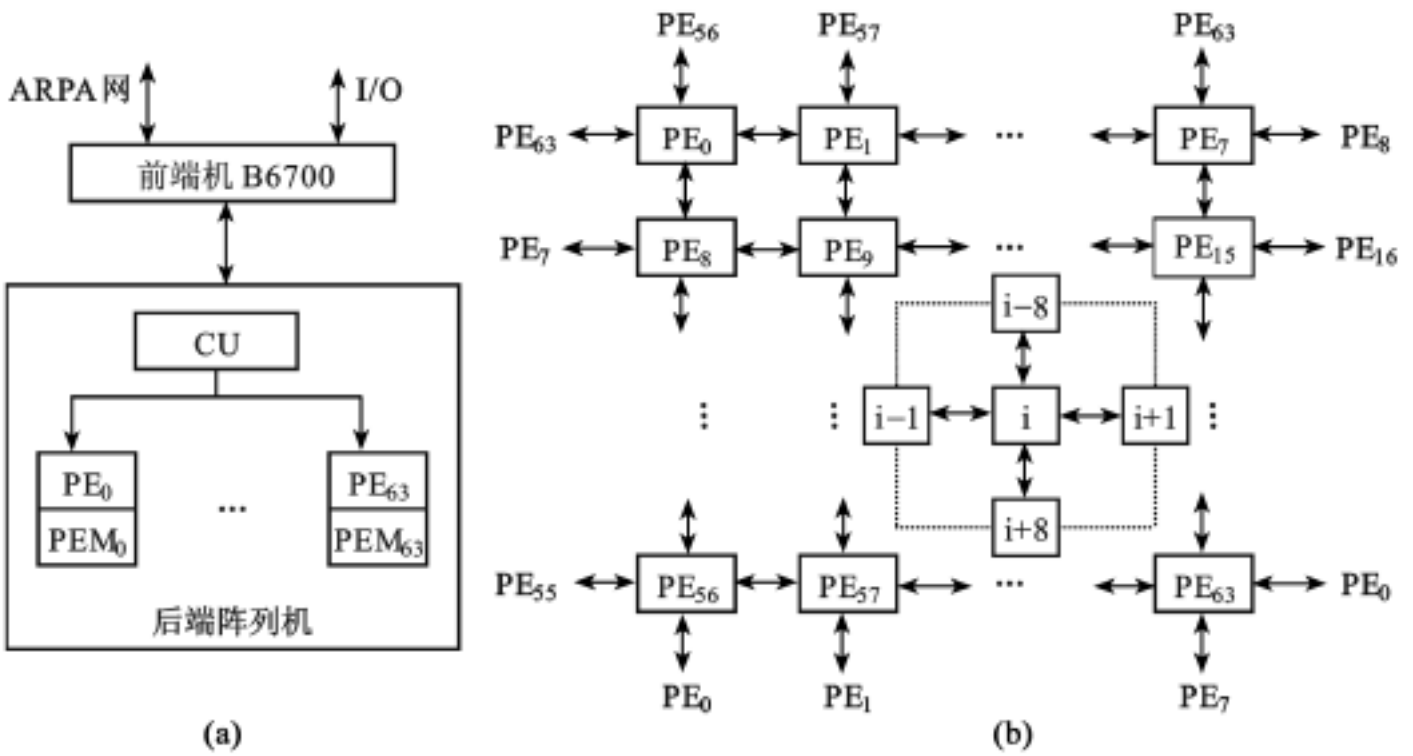


图 6.2 ILLIAC-Ⅳ阵列计算机结构

(1)ILLIAC - Ⅳ阵列

ILLIAC-Ⅳ阵列计算机共有 64 个 PE,由控制器 CU 统一控制。系统由一台 B6700 作为宿主机进行管理,每个 PE 有自己的局存 PEM,容量 2K 字,字长 64 位。每个 PE 拥有 4 个 64 位寄存器,分别用做累加器、操作数寄存器、数据路由寄存器和通用寄存器。此外,尚有 1 个 16 位变址器和 1 个 8 位方式寄存器,后者是用来存放 PE 屏蔽信息的。这是一种闭螺线阵列,每个 PE 只能与 4 个近邻的 PE 直接相连,即 PE 与 PE_{i-1}, PE_{i+1}, PE_{i-8} 和 PE_{i+8} (Mod 64)4 个近邻直接相连。因此,从一个 PE 要将数据达到另一个 PE 时,可能中间要经过若干个其他 PE 转送。传送步数 $S \leq \sqrt{N} - 1$,这里 N 为 PE 数。对于 ILLIAC-Ⅳ来讲,由于 N = 64,故 S≤7。例如,从 PE_9 到

PE_{45}的距离以下列路径为最短:

$$PE_9 \rightarrow PE_1 \rightarrow PE_{57} \rightarrow PE_{56} \rightarrow PE_{48} \rightarrow PE_{47} \rightarrow PE_{46} \rightarrow PE_{45}$$

ILLIAC-Ⅳ阵列机的每一个处理单元 PE 有 6 个可编程序寄存器 RGA, RGB, RGR, RGS, RGX 和 RGM 以及加/乘算术单元 AU、逻辑单元 LU、移位单元 SU 和地址加法器 ADA 等。

① RGA 是累加寄存器,存放第一操作数和操作结果。

② RGB 是操作数寄存器,存放加、减、乘、除等二元操作的第二操作数。

③ RGR 是被乘数寄存器,也是互连寄存器,用于 PE 与经过东、西、南、北 4 个互连通路之一的另一个处理单元之间的数据直接传送。

④ RGS 是通用寄存器,程序用来暂存中间结果。

这 4 个寄存器都是 64 位的。操作数来自以下 4 个方面: PE 本身的寄存器、PE 的处理单元存储器 PEM, CU 的公共数据总线 CDB, PE 的 4 个近邻处理单元。

⑤ RGX 是 16 位的变址寄存器,它利用地址加法器 ADA 修改指令地址,并将形成的有效地址经过存储器地址寄存器 MAR 输入存储器逻辑部件 MLU。

⑥ RGM 是 8 位的模式寄存器, RGM 中的 E 和 E1 位是"活动"标志位,用来控制 RGA, RGS 和处理单元存储器 PEM, E 位还控制 RGX。活动标志位的设立使得 64 个处理单元中的每一个处理单元都可以单独控制,只有那些处于活动状态的处理单元才执行单指令流规定的共同操作。RGM 的其他 6 位分别是: F 和 F1 位保存运算出错(上溢、下溢)标志, G, H, I, J 位保存测试结果。RGM 经常处于 CU 的监督之下,一旦出错,就发出 CU 陷阱中断。

处理单元存储器 PEM 分属于每一个处理单元,各有 2048×64 位存储容量, PEM 的取数时间不大于 350ns。64 个 PEM 联合组成阵列存储器,存放数据和指令。整个阵列存储器可以接受阵列控制器 CU 的访问,读出 8 个字的信息块到 CU 的缓冲器中。阵列存储器也可经过 1024 位的总线与 I/O 开关相连。每个 PE 只能直接访问自己的 PEM。分布在各个 PEM 中的公共数据只能先读至 CU 后,再经 CU 的公共数据总线 CDB 广播到 64 个处理单元中去。

阵列存储器的另一个特点是它具有双重变址机构:控制器 CU 实现所有处理单元的公共变址,每一个处理单元 PE 还可以单独变址。双重变址机构增加了各处理单元存储器之间数据分配的灵活性。

(2)阵列控制器

阵列控制器 CU 实际上是一台小型控制计算机。CU 除了能对阵列的处理单元实行控制以外,还能利用自己的内部资源执行一整套指令,用以完成标量的运算操作。CU 的标量操作与各 PE 的数组操作是时间重叠的。

阵列控制器 CU 同处理单元阵列之间有 4 条信息通路:

① CU 总线。处理单元存储器 PEM 经过 CU 总线把指令和数据送往阵列控制器 CU,以 8 个 64 位字为一个信息块。这里的指令是指分布存放在 PEM 中用户程

序的指令,而数据可以是处理所需要的公共数据。指令和数据先被送到 CU,再由 CU 的广播功能把它们送到各处理单元。

② 公共数据总线 CDB(Common Data Bus)。CDB 是 64 位总线,用于向 64 个处理单元同时广播公共数据的通路。例如,作为公共乘数的常数就不必在 64 个 PEM 中重复存放,可以由 CU 的某一个寄存器送往各处理单元。另外,指令的操作数和地址部分也要经过 CDB 送来。

③ 模式位线(Mode Bit Line)。每一个 PE 都可以经过模式位线把它的模式寄存器中的状态送到 CU 中来,送来的信息中包括该处理单元的"活动"状态位。从 64 个 PE 送往 CU 的模式位在 CU 的累加寄存器中拼成一个模式字,以便在 CU 内部执行一定的测试指令,对这个模式字进行测试,并根据测试结果来控制程序的转移动作。

④ 指令控制线。处理单元微操作控制信号和处理单元存储器地址、读/写控制信号都经过约 200 条指令控制线由 CU 送到阵列处理单元 PE 和存储器逻辑部件 MLU。

概括起来,阵列控制器 CU 的功能有以下 5 个方面:

① 对指令流进行控制和译码,执行标量操作指令。

② 向各处理单元发出执行数组操作指令所需的控制信号。

③ 产生和向所有处理单元广播公共的地址部分。

④ 产生和向所有处理单元广播公共的数据。

⑤ 接收和处理由各 PE(计算出错时)、系统 I/O 操作以及主机 B6700 所产生的陷阱中断信号。

(3)输入输出系统

ILLIAC-Ⅳ输入输出系统由磁盘文件系统 DFS,I/O 分系统和 B6700 组成。

磁盘文件系统 DFS 是两套大容量并行读写磁盘系统及其相应的控制器。每套磁盘系统有 13 台磁盘机,总容量为 10^9 位。每台磁盘机有 128 道,每道有一个磁头,并行读写,数据宽度为 256 位,最大传输率为 502×10^6 bit/s,平均等待时间为 19.6ms。如果 DFS 的两个通道同时发送或接收数据,则数据宽度为 512 位,最大传输率可达 10^9 bit/s。

I/O 分系统包括 3 部分,即输入输出开关 IOS、控制描述字控制器 CDC 和输入输出缓冲存储器 BIOM。

IOS 的功能有两个:一是开关功能,用以把 DFS 或可能连上的实时装置转接到阵列存储器,进行大批数据的 I/O 传送;二是作为 DFS 和 PEM 之间的缓冲,以平衡两边不同的数据宽度。

CDC 的功能是对阵列控制器 CU 的 I/O 请求进行管理。CU 提出 I/O 请求时,CDC 将使 B6700 管理计算机中断,由 B6700 响应输入/输出请求,并通过 CDC 给 CU 送回相应的响应代码,在 CU 中设置好必要的控制状态字。然后,CDC 促使 B6700

启动 PEM 的加载过程,由 DFS 向 PEM 送入程序和数据。在 PEM 加载完毕后,又由 CDC 向 CU 传送控制信号,使 CU 开始执行 ILLIAC-Ⅳ的程序。

BIOM 处在 DFS 和 B6700 之间,是为了使二者之间传送频宽能够匹配。B6700 存储器经 CPU 输送数据的频宽是 80×10^6 bit/ s,而 DFS 输送数据的频宽是 500×10^6 bit/ s,二者之间相差 6 倍多。因此,必须设立 BIOM 作为 B6700 和 DFS 之间的缓冲。BIOM 把 B6700 的 48 位字变换为 ILLIAC-Ⅳ的 64 位字,同时按双字 128 位的数据宽度输送给 DFS。实际上,BIOM 是用 4 个 PEM 做成的,总容量为 8 192×64 位。

B6700 管理计算机的基本组成是:单中央处理器(另一个 CPU 可选),32K 字内存(可扩充至 512K 字),经由多路开关控制连接的一批外围设备(包括一台容量为 1 012 位的激光外存储器以及 ARPA 网络接口)。B6700 的作用是:管理 ILLIAV-Ⅳ的全部系统资源,完成用户程序的编译或汇编,为 ILLIAC-Ⅳ进行作业调度、存储分配、产生输入/ 输出控制描述字送至 CDC、处理中断,以及提供操作系统所具备的其他服务。

2. BSP 计算机

BSP 计算机是由美国宝来公司和伊利诺斯大学于 1979 年制造的,它是共享存储器结构的 SIMD 计算机的典型代表。BSP 并不是一台独立运行的计算机,它是附属于系统管理机的一台后端处理机。

BSP 计算机及系统管理机组成框图如图 6 .3 所示。它由系统管理计算机 B7700/ B7800 和 BSP 处理机两大部分组成,前者可视为后者的前端机。系统管理机负责 BSP 程序编译、与远程终端及网络的数据通信、外围设备管理等,大多数 BSP 作业调度和操作系统活动也是在系统管理机上完成。BSP 处理机由控制处理机、文件存储器、并行存储器模块以及对准网络等组成,如图 6 .4 所示。

(1)并行处理机

并行处理机以 160ns 的时钟周期进行向量计算。所有 16 个算术单元(AE)对不同的数据组(从并行处理机控制器广播来)进行同一种指令操作。大部分的算术运算能在 2 个时钟周期(320ns)内完成。BSP 的执行速度最高可达 50MFLOPS。进行向量运算的数据存储在 17 个并行存储器模块中。每个模块的容量可达 512K 字,时钟周期为 160ns。数据在存储器模块和 AE 之间以每秒 100 兆字的速率进行传输。17 个存储器模块的组织形成了一个无冲突访问存储器,它容许对任意长度以及跳距不足 17 倍数的向量实现无冲突存取。

16 个 AE 是以 SIMD 方式在单一微序列控制下同步工作的。在每个 AE 中,只有最原始的操作才采用硬连线方式,控制字的宽度为 100 位,除实现浮点操作以外,AE 还有较强的非数值处理能力。

浮点加、减和乘都能在两个时钟周期内完成。采用两个时钟周期可使存储器频

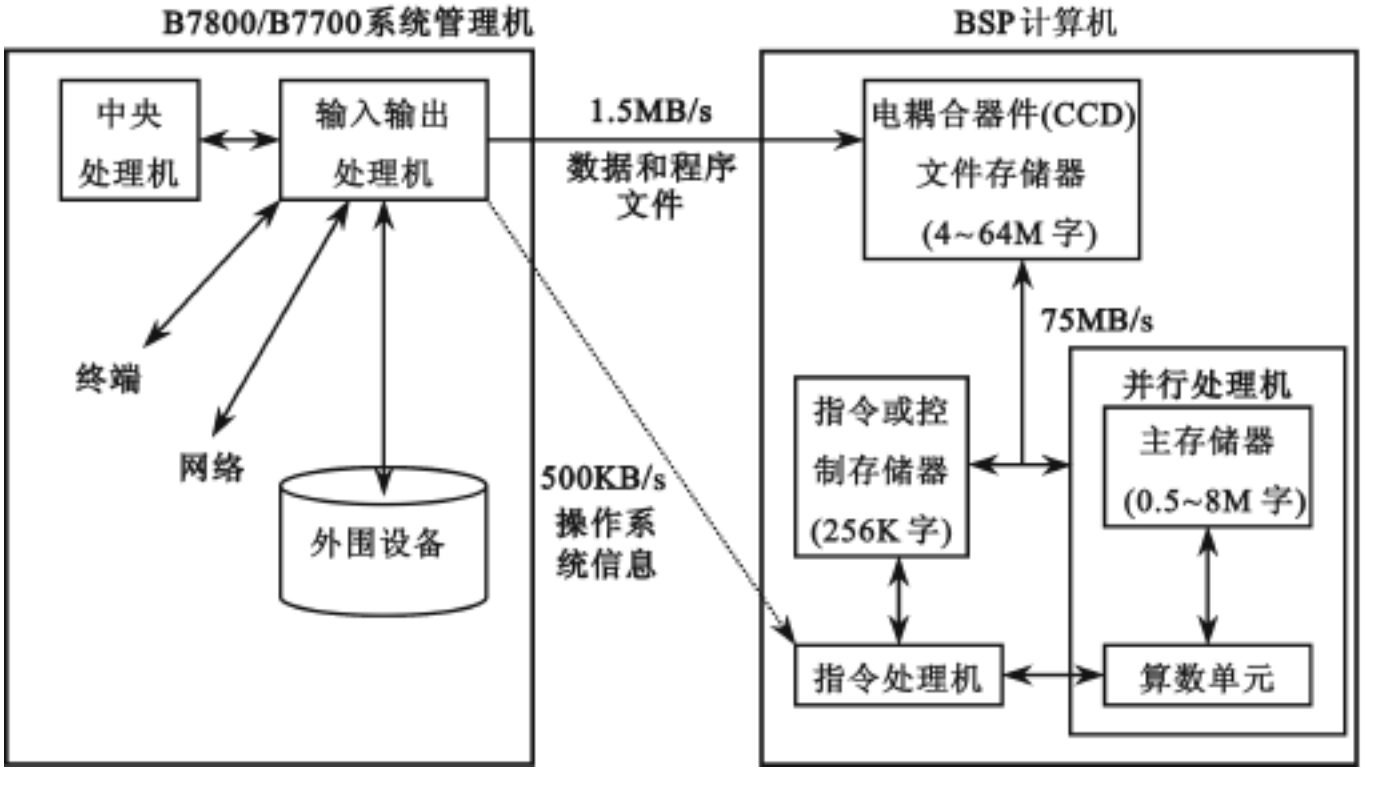


图 6.3 附属于系统管理机的 BSP 计算机

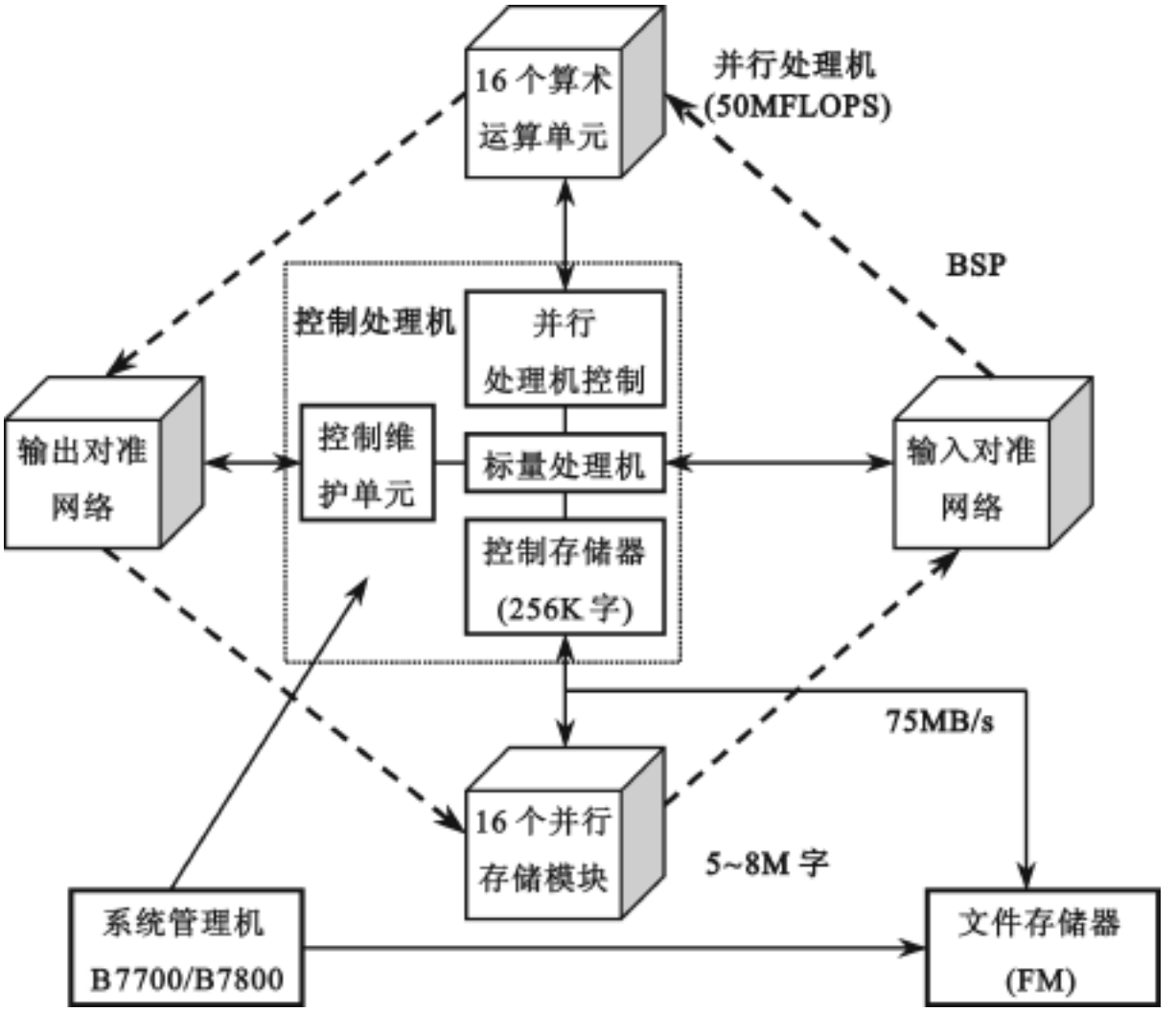


图 6.4 BSP 的流水线处理结构

宽与 AE 进行三元操作时的频宽相平衡。所谓三元操作是指由三个操作数产生一个结果的操作。浮点除要用 1 200ns,是用 Newton-Raphson 迭代算法产生倒数来实现的。在每个 AE 中设有只读存储器,以给出除法和平方根迭代的第一次近似值。浮点字长为 48 位,尾数为 36 位有效值,阶码为 10 位,以 2 为底。数的精度可达到十进制 11 位。AE 在关键部位设置了双字长累加器和双字长寄存器,这就能使双精度运算直接用硬件实现。AE 还可以用软件方法实现三倍精度的算术运算。可以估算得出来,在 BSP 中用 FORTRAN 来表达很大范围的计算问题中,其速度可达 20～40 MFLOPS。

BSP 可以对下列四类操作进行并行计算:

① 16 个算术单元实现并行运算;

② 存储器的读取/ 存取,存储器与算术运算单元间的数据传输;

③ 在并行处理机控制器中的变址值、向量长度和循环控制计算;

④ 线性向量操作描述字在标量处理机中的生成。

(2)控制处理机

控制处理机除了用以控制并行处理机以外,还提供了与系统管理机相连的接口。标量处理机则处理存储在控制存储器中的全部操作系统和用户程序的指令。它以 12 MHz 的时钟频率执行用户程序的串行或标量部分,最高速度可达 l .5 MFLOPS。全部的向量指令以及某些成组的标量指令被送给并行处理机控制器。在经过合格性检查后,控制器将它们转换为微序列,去控制 16 个 AE 操作。双极型控制存储器的容量为 256K 字,周期为 160 ns,每个字长 48 位另加 8 位奇偶校验位,提供单错校正双错检测(SECDED)的能力。控制维护单元是系统管理机与控制处理机其余部分之间的接口,用来进行初始化、监控命令通信和维护。

(3)文件存储器

文件存储器是一个半导体辅助存储器。BSP 的计算任务文件从系统管理机加载到它上面。然后对这些任务进行排队,由控制处理机加以执行。文件存储器是 BSP 直接控制下惟一的外部设备,而其他的外部设备则都由系统管理机来控制。在 BSP 程序执行过程中所产生的暂存文件和输出文件,在将它们送给系统管理机输出给用户之前是存在文件存储器中的。文件存储器的数据传输率较高,大大地缓解了 I/ O 受限问题。

(4)对准网络

对准网络包含完全交叉开关以及用来实现数据从一个源广播至几个目的地以及当几个源寻找一个目的地时能分解冲突的硬件。这就需要在算术单元阵列和存储器模块之间具备通用的互连特性。而存储器模块和对准网络的组合功能则提供了并行存储器的无冲突访问能力。算术单元也利用输出对准网络来实现一些诸如数据压缩和扩展操作以及快速傅立叶变换算法等专用功能。

在 BSP 中,存储器—存储器型的浮点运算是流水进行的。BSP 的流水线组织由

5 个功能级组成。16 个操作数先从存储器模块中取出,通过输入对准网络送给 AE 进行处理,再将结果经输出对准网络送给存储器模块存储起来。这几级的操作都是重叠进行的(参见图 6 .4)。请注意,在物理上输入对准和输出对准都是在一个实际对准网络中进行的,这里的划分只是对流水级的功能划分而已。除了在 16 个 AE 中显示出的空间并行性以及读取、对准和存储的流水线操作外,AE 中的向量运算还可以同标量处理机中的标量处理重叠起来。这就使得系统既适合于处理长向量和短向量,也能处理单独的标量,系统的功能很强也很灵活。

(5)质数存储系统

BSP 并行存储器由 17 个时钟周期为 160ns 的存储模块组成。由于每个周期存取 16 个字,因此每个字的最大有效存储周期时间为 10ns。这与算术单元完成浮点加和乘的速率很好地平衡。因为每次运算需要两个变量,算术单元中设有中间寄存器其运算速度为 320ns/ 16 次,即 20 ns/ 次。

由于程序和标量都存放在控制存储器中,因此只有数组存取(包括 I/ O)才用到并行存储器。这样,对于三元向量来说,因为在两次算术运算中需要用到 3 个变量,产生 1 个结果,共访问存储器 4 次,所以在并行存储器和浮点运算之间的频带保持完全平衡。对于长向量来说,由于中间结果都存在寄存器中,每次运算只需要一个操作数,因此并行存储器有足够的频宽留给输入和输出信息使用。

6 .3 SIMD 并行计算机算法

SIMD 并行计算机是从求解诸如有限差分、矩阵运算、信号处理、线性规划等一系列计算为背景发展起来的。下面以图像平滑算法为例,讲述该算法如何在并行机上实现。

该算法是用来平滑输入图像的灰度级,I 和 S 分别表示输入和输出图像。假定 I 和 S 均含有 512×512 个像素。I 中的每个点是一个 8 位无符号整数,用来表示 256 个灰度级。每个像素的灰度级是用来表示像素的黑色程度,以 0 表示白色,以 255 表示黑色。平滑后图像中的每个点 S(i, j)是 I(i, j)和它的 8 个最邻近的像素的灰度级的平均值。S 中的上、下、左、右边线上的像素,由于在 I 中的相应像素没有 8 个邻近像素,因而置为 0。

若用具有 1 024 个 PE 的阵列机来实现上述算法,可如图 6 .5(a)所示那样排列成32×32方阵。在每个 PE 中存储一个 16×16 的子图像块(整个 I 图像为 512×512 像素块)。PE_0 中存储行 0～15 和列 0～15 的子图像块,PE_1 中存储行 0～15 和列 16～31 的子图像块,依此类推。每个 PE 平滑自己的子图像,而所有 PE 可同时进行平滑操作。在每个 16×16 子图像的边界处,为了计算已平滑的值,数据必须在 PE 间传送。图 6 .5(b)中示出了所必须传送的数据。阵列机的每个 PE 都有类似于单处理单元的数据传送模式。要对 512×512 图像完成平滑操作,在上述的每块为 16×16

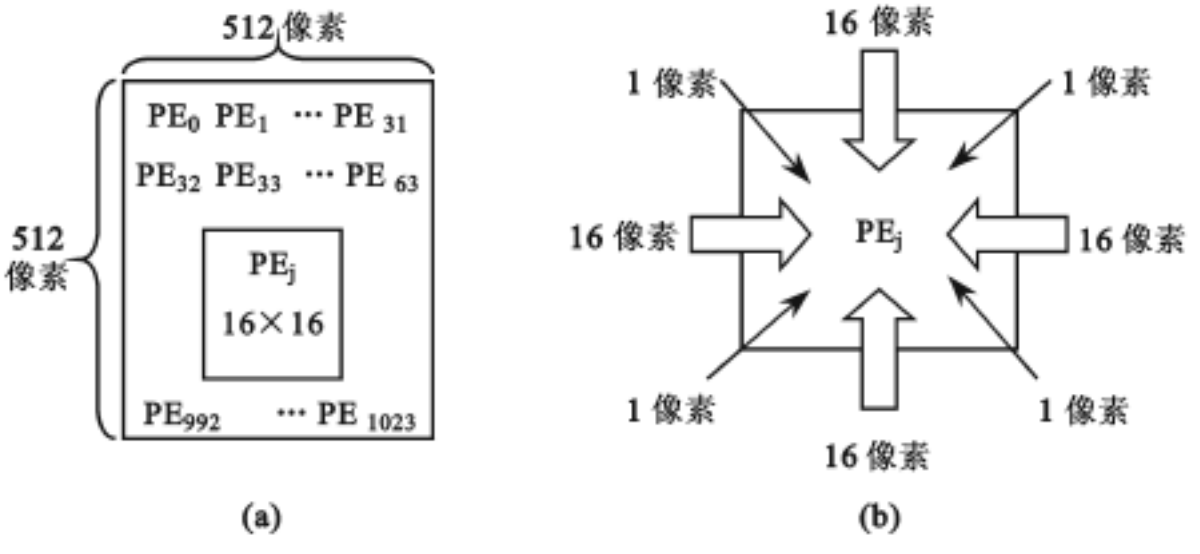


图 6.5　图像平滑算法的数据分配和 PE 间传送

大小,共 1 024 子图像块上用并行处理方法来完成平滑,则需 16×16 = 256 次并行的平滑操作。在操作过程中,总共所需的并行数据元素的传送数为 4×16 + 4 = 68(上,下,左,右)4 边每边需 16 次,而 4 个对角上每个需 1 次,参见图 6.5(b)。对同样大小的图像,若用串行算法来完成,虽然此时不需要进行 PE 间的数据传送,但总共却需要 512×512 = 262 144 次平滑操作。因此,并行算法就比串行算法快 1 024 倍。如果计入 PE 间传送所需时间,并假定每次并行数据传送时间相当于一次平滑操作时间,则改进倍数仍可达到:

$$262\,144/(256+68)=809$$

另外,就所占比例而言,数据传送所需时间几乎可以忽略不计。

6.3.1　矩阵加

阵列处理机解决矩阵加是最简单的一维情况。两个 8×8 的矩阵 A,B 相加,所得的结果矩阵 C 也是一个 8×8 的矩阵。只需把 A,B,C 居于相应位置的分量存放在同一个 PEM 内,且在全部 64 个 PEM 中,让 A,B 和 C 的各分量地址均对应取相同的地址 α,α+1 和 α+2,如图 6.6 所示。这样,实现矩阵加只需用下列三条 ILLIAC-Ⅳ汇编指令:

```
LDA     ALPHA      ;全部(α)由 PEMi 送 PEi 的累加器 RGAi
ADRN    ALPHA+1    ;全部(α+1)与(RGAi)浮点加,结果送 RGAi
STA     ALPHA+2    ;全部(RGAi)由 PEi 送 PEMi 的α+2 单元
```

其中,0≤i≤63。

从这个例子可以明显看出阵列处理机的单指令流(3 条指令顺序执行)、多数据流(64 个元素并行相加)以及数组并行中的“全并行”工作特点。由于是全部 64 个处理单元在并行操作,速度就提高为顺序处理的 64 倍。同时也可以看出,对于具有分

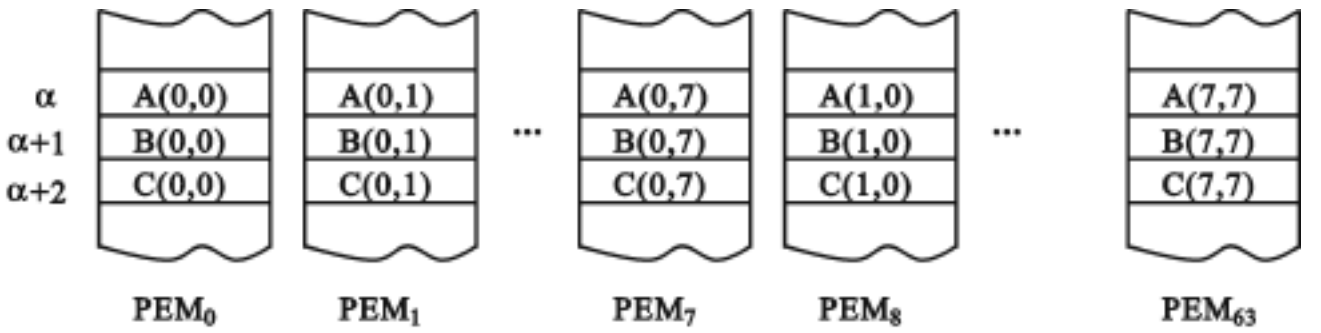


图 6.6 ILLIAC-Ⅳ 阵列机中完成矩阵加的存储器数据分配

布式存储器的阵列处理机,能否发挥其并行性与信息在存储器的分布密切相关。而信息分布算法又与系统结构及所解题目直接相关,因此,存储单元分配算法的设计比较麻烦。

6.3.2 矩阵乘

矩阵乘是二维数组运算,比矩阵加要复杂。设 A,B 和 C 为 3 个 8×8 的二维矩阵,给定 A 和 B,计算 C = A×B 的 64 个分量可用公式:

$$C_{ij} = \sum_{k=0} a_{ik} \times b_{kj}$$

其中,0≤i≤7 且 0≤j≤7。

在 SISD 计算机上求解,可执行用 FORTRAN 语言编写的下列程序:

```
DO 10   I = 0,7
DO 10   J = 0,7
C(I,J) = 0
DO 10 K = 0,7
10   C(I,J) = C(I,J) + A(I,K)×B(K,J)
```

需经 I,J,K 三重循环完成。每重循环执行 8 次,共需 512 次乘、加的时间,且每次还要包括执行循环控制判别等其他操作所需的时间。如果在 SIMD 阵列处理机上运算,可用 8 个处理单元并行计算矩阵 C(I,J)的某一行或某一列,即将 J 循环或 I 循环转化成一维的向量处理,从而消去了一重循环。以消去 J 循环为例,可执行用 FORTRAN 语言编写的下列程序:

```
DO 10 I = 0,7
C(I,J) = 0
DO 10 K = 0,7
10   C(I,J) = C(I,J) + A(I,K)×B(K,J)
```

让 J = 0～7 各部分同时在 PE_0～PE_7 上运算,这样只需 K、I 二重循环,速度可提高至 8 倍,即只需 64 次乘、加时间。其程序流程图如图 6.7 所示。

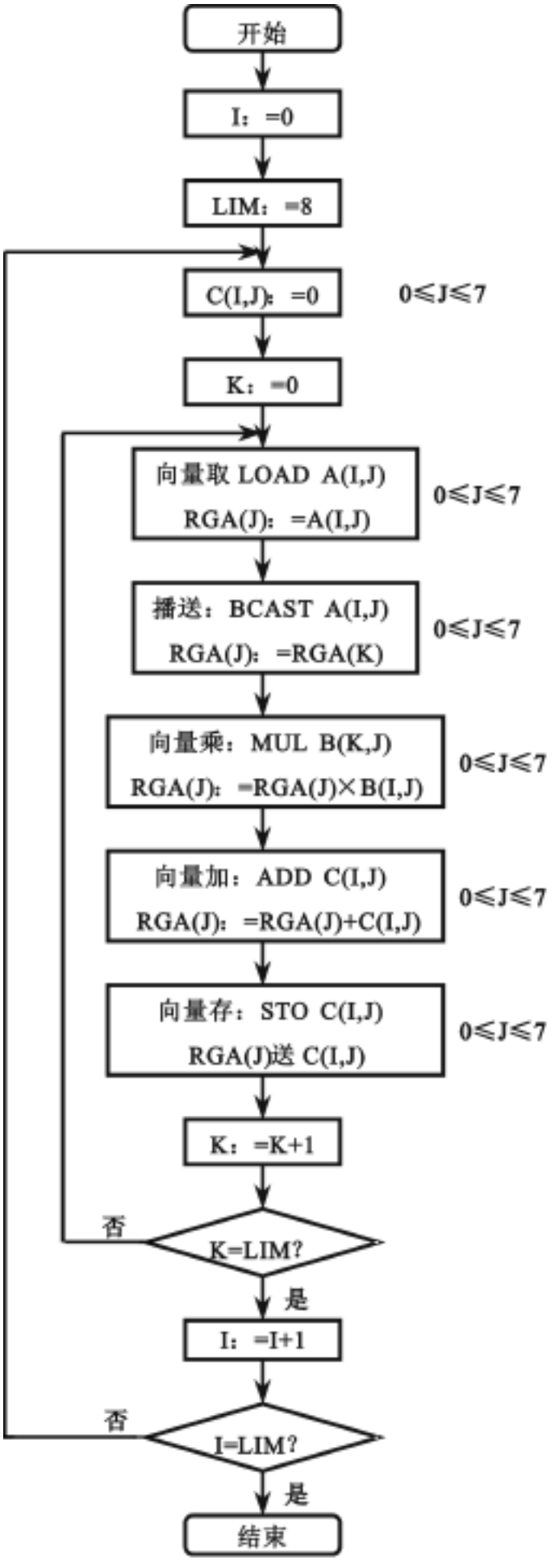


图 6 .7　矩阵乘程序流程图

需要说明的是其执行过程虽与 SISD 的类似,但实际的解决方式是不同的。每次控制部件执行的 PE 类指令表面上是标量指令,实际上已等效于向量指令,如向量取、向量存、向量加、向量乘等,是 8 个 PE 并行地执行同一条指令。每次播送时,利用阵列处理机的播送功能将处理单元 PE_K 中累加寄存器 RGA_K 的内容经控制部件 CU 播送到全部 8 个处理单元的 RGA 中去。

然而为了让各个处理单元 PE_i 尽可能只访问所带局部存储器 PEM_i,以保证高速处理,就必须要求对矩阵 A,B,C 各分量在局部存储器中的分布采用如图 6.8 所示的方案。

如果把 ILLIAC-Ⅳ的 64 个处理单元全部利用起来并行运算,即把 K 循环的运算也改为并行,还可以进一步提高速度,但需要在阵列存储器中重新恰当地分配数据。同时还要使 8 个中间积 A(I,K)×B(K,J)能够并行相加(其中,0≤K≤7),这就要用到下面的累加和并行算法。即使如此,就 K 的并行来说,速度的提高也不是 8

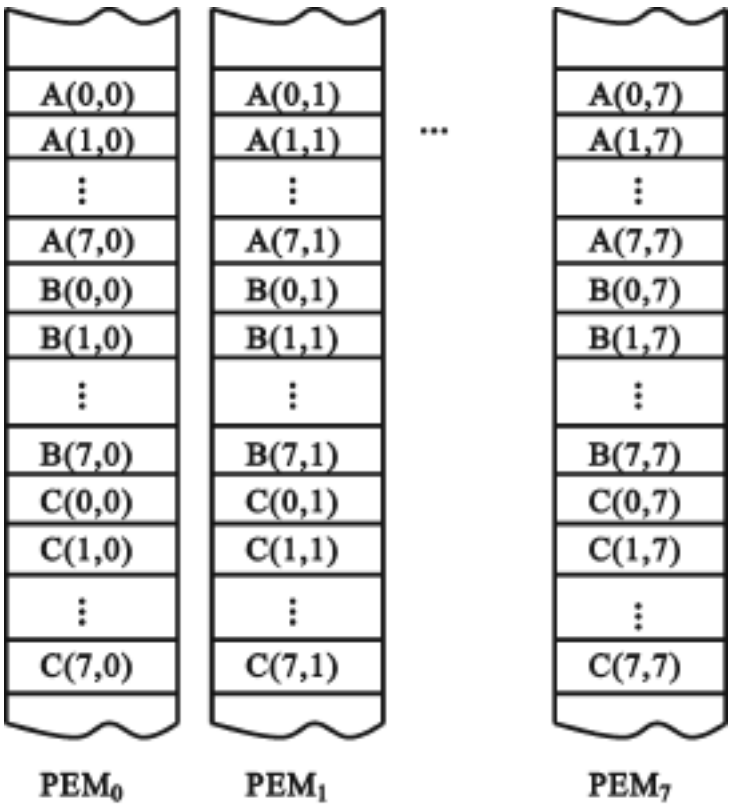


图 6.8 矩阵乘并行运算的存储器数据分配方案

倍,而只是 $8/\log_2 8$,接近于 2.7 倍。

6.3.3 累加和

这是一个将 N 个数的顺序相加转为并行相加的问题。为得到各项累加的部分和以及最后的总和,要用到处理单元中的活跃标志位。只有处于活跃状态的处理单元才能执行相应的操作。为叙述方便取 N = 8,即有 8 个数 A(I)顺序累加,其中 0≤i≤7。

在 SISD 计算机上可以写成下列 FORTRAN 程序:

```
C = 0
DO 10 I = 0,7
10   C = C + A(I)
```

这是一个串行程序,需要 8 次加法时间。

在阵列处理机上用成对递归相加算法,只需 $\log_2 8 = 3$ 次加法时间即可。首先,原始数据 A(I)分别存放在 8 个 PEM 的 α 单元中,其中 $0 \leq i \leq 7$。然后,按下面的步骤求累加和:

① 置全部 PE_i 为活跃状态,$0 \leq i \leq 7$;

② 全部 A(I)从 PEM_i 的 α 单元读到相应 PE_i 的累加寄存器 RGA_i 中,$0 \leq i \leq 7$;

③ 令 K = 0;

④ 将全部 PE_i 的(RGA_i)转送到传送寄存器 RGR_i,$0 \leq i \leq 7$;

⑤ 将全部 PE_i 的(RGR_i)经过互连网络向右传送 2^K 步距,$0 \leq i \leq 7$;

⑥ 令 $j = 2^K - 1$;

⑦ 置 PE_0 - PE_j 为不活跃状态;

⑧ 处于活跃状态的所有 PE_i 执行(RGA_i):=(RGA_i)+(RGR_i),$j \leq i \leq 7$;

⑨ K:= K + 1;

⑩ 如 K < 3,则转回④,否则往下继续执行;

⑪ 置全部 PE_i 为活跃状态,$0 \leq i \leq 7$;

⑫ 将全部 PE_i 的累加寄存器内容(RGA_i)存入相应 PEM_i 的 α + 1 单元中,$0 \leq i \leq 7$。

图 6.9 描绘了阵列处理机上累加和的计算过程。框中的数字表明各处理单元每次循环后相加的结果。图中用数字 0～7 分别代表 A(0)～A(7)。画有阴影线的处理单元表示此时不活跃。另外,图中对上述第⑤步中 PE 的(RGR_i)在右移时超出 PE_7 的内容没有表示出来,这是因为若右移步距为 $2^K \pmod 8$,应移入 PE_0～PE_i,而这些 PE 在第⑦步将要置为不活跃,所以无论它的 RGR 接受什么内容都不会对执行结果有影响。

上面的例子表明,虽然经过变换,在 ILLIAC-Ⅳ上可以实现累加和的并行运算,但由于屏蔽了部分处理单元,降低了它们的利用率,所以速度不是提高 N 倍,而只是 $N/\log_2 N$ 倍。

6.4 SIMD 计算机的互连网络

6.4.1 互连网络的设计目标

在 SIMD 计算机中,无论是处理单元之间,还是处理单元和存储模块之间,都要经互连网络来实现信息交换。因此,互连网络性能好坏对 SIMD 计算机系统的运算

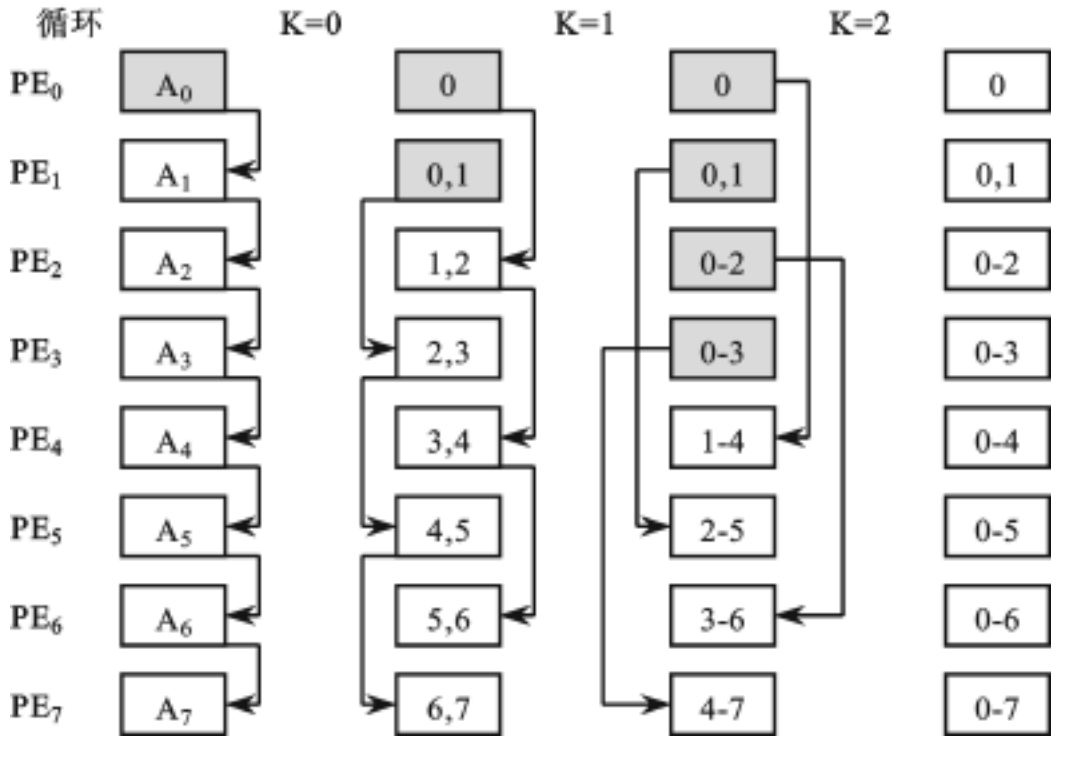


图 6 9　阵列处理机上累加和的计算过程

速度、处理单元的使用率、求解算法适应性、拓扑结构灵活性以及成本等有很大影响，所以，它是 SIMD 计算机中重要的研究课题之一。

若把处理单元或存储模块看成节点，则互连网络实际是为输入和输出两组节点之间提供数据通路或映像。对于 N 个输入和 N 个输出来讲，由于不允许有多对一的映像(因为会造成访问冲突)，由输入到输出映像共有 N^N 种，如果限定为一对一映像，则只有 N！种映像。

衡量互连网络性能好坏的主要因素是它的连接度、延时性、带宽、可靠性和成本。连接度是指一个节点与其他节点的连接程度。如果一个节点直接连接的其他节点数越多，连接度就越高，表明连接性越好。延时性是指从一个节点传送信息到任何一个节点所需的时间。通常可用节点间最大距离加以表示。

在设计互连网络时应考虑以下的四个方面的问题：

(1)通信工作方式

通信工作方式可分为同步和异步两种。在同步方式中，不论是各个 PE 对数据进行并行操作还是由控制器向处理单元广播命令，都由统一的时钟来加以同步。SIMD 并行机都采用这种方式。异步方式则不用统一时钟加以同步，各个处理单元根据需要相互建立动态连接。

(2)控制策略

控制策略分为集中和分散两种。集中式控制由一个统一控制器对各个互连开关状态加以控制，而分散式控制则由各个互连开关自身实行管理。一般的 SIMD 并行机都采用集中控制。

(3)交换方式

交换方式分为线路交换和分组交换两种。线路交换是在整个交换过程中,在源和目标节点之间建立固定的物理通路,适用于成批数据传送。分组交换则把要传送的一个信息分成多个分组,分别送入互连网络。这些分组可通过不同的路由到达目标节点。因此,较适合于短数据报文的传送。SIMD 并行机一般采用线路交换,因为处理单元间的连接比较紧密。MIMD 多机系统则往往采用分组交换方式。

(4)网络拓扑

网络拓扑分为静态和动态两种。这里的拓扑是指互连网络中的各个节点间的连接关系,通常用图来描述。静态拓扑是指在各节点间有专用的连接通路,且在运行中不能改变。动态拓扑则因设置有源开关,因而可根据需要借助控制信号对连接通路加以重新组合。静态拓扑一维的有线性阵列结构,二维的有环形、星形、树形、网格形等,三维的有立方体等,三维以上的有超立方体等。

以上四个方面的问题是设计互连网络的重要依据,设计时必须综合考虑加以合理组合。

由以上分析可知,SIMD 系统的互连网络的设计目标是:

① 结构不要过分复杂,以降低成本;

② 互连要灵活,以满足算法和应用的需要;

③ 处理单元间信息交换所需传送步数要尽可能少,以提高速度性能;

④ 能用规整单一的基本构件组合而成,或者经多次通过或者经多级连接来实现复杂的互连,模块性好,以便于用 VLSI 实现并满足系统的可扩充性。

6.4.2 互连函数

互连函数是反映互连网络连接特性的一组定义。如果将互连网络中的 N 个输入节点和 N 个输出节点的节点号用十进制 $0,1,2,\cdots,N-1$ 表示,则每一个节点的地址也可用 $n=\log_2 N$ 位二进制表示。如某一输入节点的二进制地址为 $x_{n-1}\ x_{n-2}\cdots x_1\ x_0$,则与此节点相连的输出节点的二进制地址可用互连函数 $f(x_{n-1}\ x_{n-2}\cdots x_1\ x_0)$ 表示。

下面讨论几种常用的互连函数及其图形表示。

1. 恒等置换函数

N 个节点的恒等置换函数的表达式为:$I(x_{n-1}\ x_{n-2}\cdots x_1\ x_0)=x_{n-1}\ x_{n-2}\cdots x_1\ x_0$。在此表达式中,输入节点的二进制地址与输出节点的二进制地址相同。当 $N=8$ 时,其表达式为:

$$I(x_2\ x_1\ x_0)=x_2\ x_1\ x_0$$

2. 方体置换函数

N 个节点的方体(Cube)置换函数有 $n=\log_2 N$ 个,其中第 $K(K=0,1,\cdots,n-1)$ 个函数的表达式为:

$$C_k(x_{n-1}\ x_{n-2}\cdots x_{k+1}\ x_k\ x_{k-1}\cdots x_1\ x_0)=x_{n-1}\ x_{n-2}\cdots x_{k+1}\ \overline{x_k}\ x_{k-1}\cdots x_1\ x_0$$

由上式可知,方体置换是实现第 K 位求反连接。

当 $N=8$ 时,共有 $n=3$ 个方体置换函数(K 分别取 0,1 和 2)分别为:

$$C_0(x_2\ x_1\ x_0)=x_2\ x_1\ \overline{x_0}\ 、C_1(x_2\ x_1\ x_0)=x_2\ \overline{x_1}\ x_0\ 、C_2(x_2\ x_1\ x_0)=\overline{x_2}\ x_1\ x_0$$

输入与输出节点互连形式如图 6.10 所示。

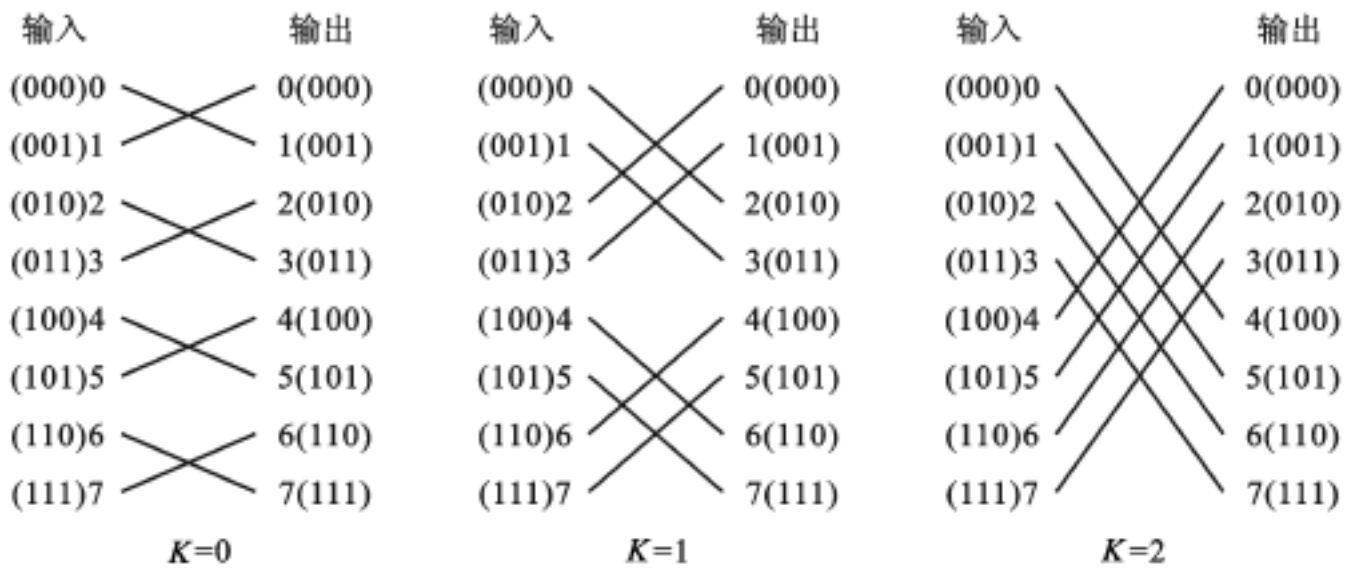


图 6.10 节点数 $N=8$ 时的方体置换(括号内为节点的二进制地址)

3. 全混洗置换函数

全混洗(Shuffle)置换又称为均匀洗牌置换,其函数表达式为:

$$S(x_{n-1}\ x_{n-2}\cdots x_1\ x_0)=x_{n-2}\cdots x_1\ x_0\ x_{n-1}$$

由上式可见,全混洗置换函数实现的是,将输入端二进制地址循环左移一位即得到输出端的二进制地址。

当 $N=8$ 时的全混洗函数式为:$S(x_2\ x_1\ x_0)=x_1\ x_0\ x_2$,输入与输出节点互连形式如图 6.11 所示。

4. 交换置换函数

N 个节点的交换(Exchange)置换函数的表达式为:

$$E(x_{n-1}\ x_{n-2}\cdots x_1\ x_0)=x_{n-1}\ x_{n-2}\cdots x_1\ \overline{x_0}$$

由上式可知,交换函数实现的是输入与输出二进制地址的最低位求反连接。

当节点数 $N=8$ 时,其交换函数的表达式为:

$$E(x_2\ x_1\ x_0) = x_2\ x_1\ \overline{x_0}$$

输入与输出节点互连形式如图 6 .12 所示。

5. 蝶式置换函数

N 个节点的蝶式(Butterfly)置换函数的表达式为:

$$B(x_{n-1}\ x_{n-2}\cdots x_1\ x_0) = x_0\ x_{n-2}\cdots x_1\ x_{n-1}$$

由上式可见,蝶式置换函数实现的是将输入端二进制地址的最高位和最低位相互交换,即可得到对应的输出端地址。

当 $N=8$ 时的蝶式函数式为:

$$S(x_2\ x_1\ x_0) = x_0\ x_1\ x_2$$

输入与输出节点互连形式如图 6 .13 所示。

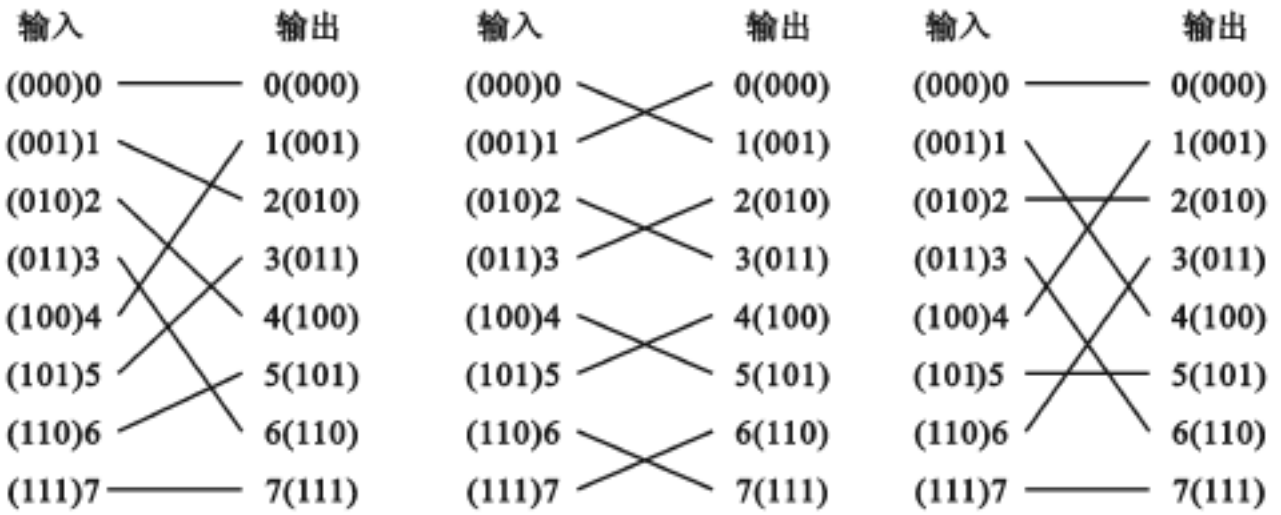


图 6 .11　全混洗置换　　图 6 .12　交换置换　　图 6 .13　蝶式置换

6. 子蝶式置换函数

与方体置换函数相类似, N 个节点的子蝶式置换函数的表达式共有 $n=\log_2 N$ 个,其中第 $K(k=0,1,\cdots,n-1)$ 个函数表达式为:

$$B_k(x_{n-1}\ x_{n-2}\cdots x_{k+1}\ x_k\ x_{k-1}\cdots x_1\ x_0) = x_{n-1}\ x_{n-2}\cdots x_{k+1}\ x_0\ x_{k-1}\cdots x_1\ x_k$$

由上式可见,第 K 个子蝶式置换函数实现的是将输入端二进制地址的第 K 位与最低位相互交换,即可得到对应的输出端地址。

当 $N=8$ 时有 3 个子蝶式函数式为:

$$B_0(x_2\ x_1\ x_0) = x_2\ x_1\ x_0,\ B_1(x_2\ x_1\ x_0) = x_2\ x_0\ x_1,\ B_2(x_2\ x_1\ x_0) = x_0\ x_1\ x_2$$

输入与输出节点互连形式如图 6 .14 所示。

7. 移数置换函数

移数置换是通过输入端数组循环的一定位置,得到输出端的地址,其表达式为:

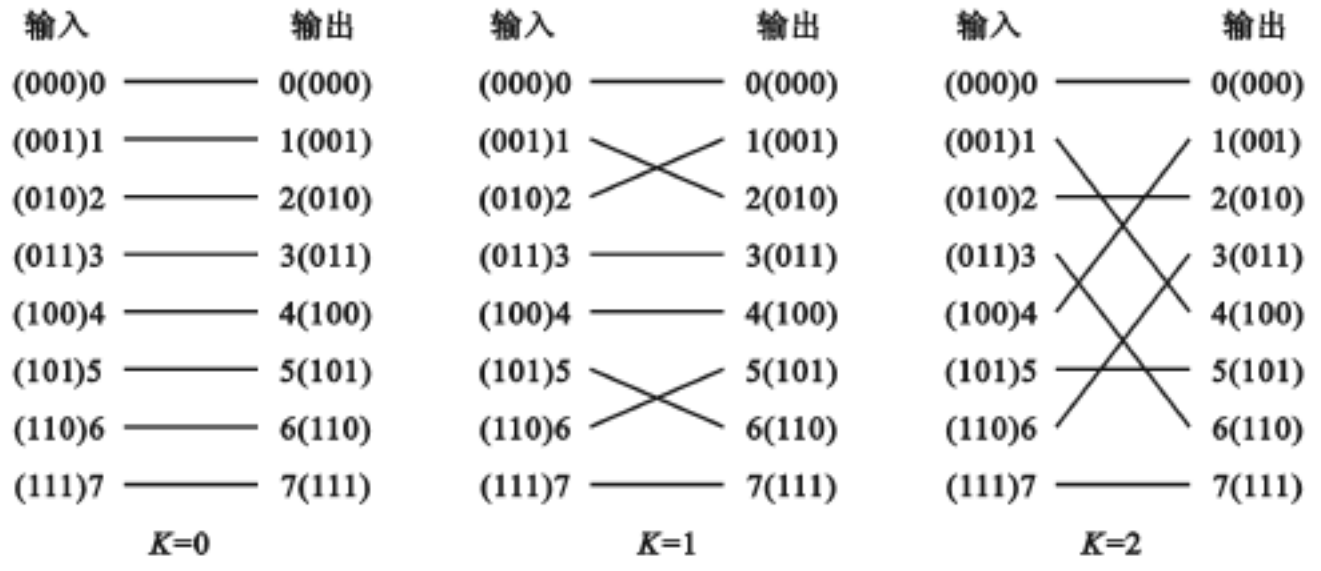


图 6 .14 $N=8$ 个节点的子蝶式置换

$$\alpha(x) = (x + K) \bmod N$$

式中,$0 \leq x \leq N-1$, K 为常数,指移过的位置。

还可把整个输入数组分成若干个子数组,在子数组内进行循环移数,称之为段内循环移数,其表达式可用两个式子表示为:

$$\alpha(x)_{(n-1):r} = (x)_{(n-1):r}$$

$$\alpha(x)_{(r-1):0} = [(x)_{(r-1):0} + K] \bmod 2^r$$

式中,下标$(n-1):r$和$(r-1):0$分别指示是从$(n-1)$位到 r 和$(r-1)$位到 0 位。 $N=8$ 个节点的移数置换和段内循环移数置换举例如图 6 .15 所示。

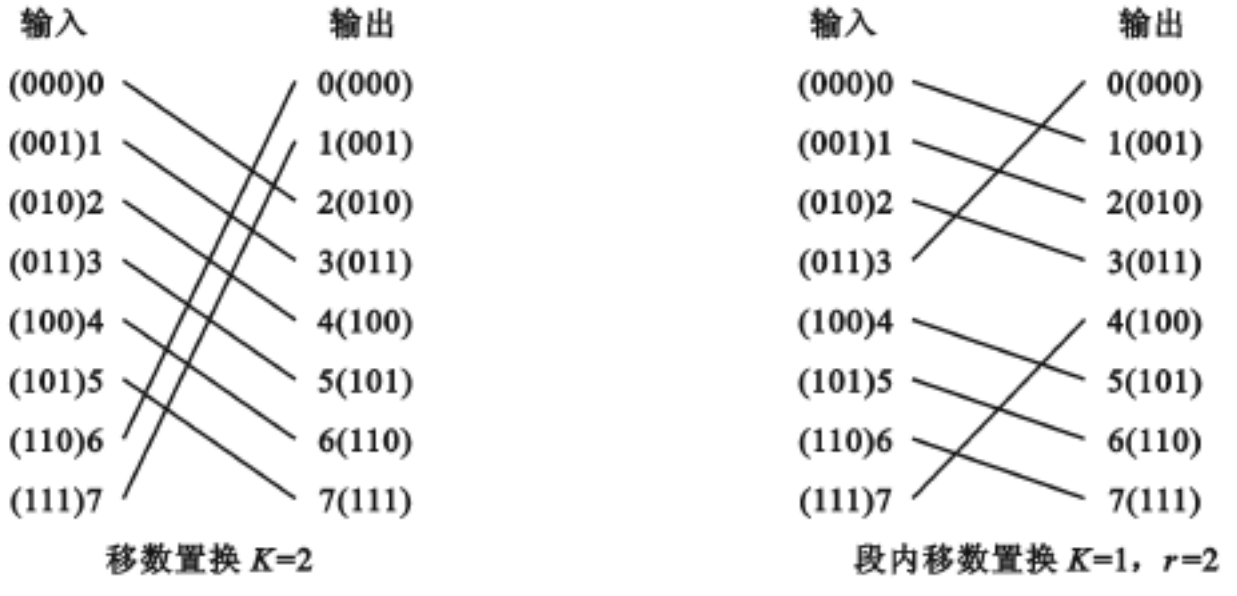


图 6 .15 $N=8$ 个节点的移数置换

8. PM2I 置换函数

N 个节点的 PM2I(Plus-Minus,加减 2^i)置换函数共有 $2n(n=\log_2 N)$ 个。其基本表达式为:

$$PM2_{+i}(j) = j + 2^i \bmod N$$

$$PM2_{-i}(j) = j - 2^i \bmod N$$

式中,$0 \leq j \leq N-1$,$0 \leq i \leq n-1$。由于 $PM2_{+(n-1)} = PM2_{-(n-1)}$,所以 PM2I 互连函数只有 $2n-1$ 种是不相同的。

当 $N=8$ 时,PM2I 互连函数有:$PM2_{+0}$,$PM2_{-0}$,$PM2_{+1}$,$PM2_{-1}$,$PM2_{+2}$、$PM2_{-2}$。其中 $PM2_{+2} = PM2_{-2}$,所以共有 5 个互不相同的函数(用循环函数表示)如下:

$PM2_{+0}$:(0 1 2 3 4 5 6 7)

$PM2_{-0}$:(7 6 5 4 3 2 1 0)

$PM2_{+1}$:(0 2 4 6)(1 3 5 7)

$PM2_{-1}$:(6 4 2 0)(7 5 3 1)

$PM2_{\pm 2}$:(0 4)(1 5)(2 6)(3 7)

其中,(0 1 2 3 4 5 6 7)表示 0 连到 1,1 连到 2,2 连到 3,…,7 连到 0。图 6.16 表示出 $N=8$ 个节点之间的部分连接情况。

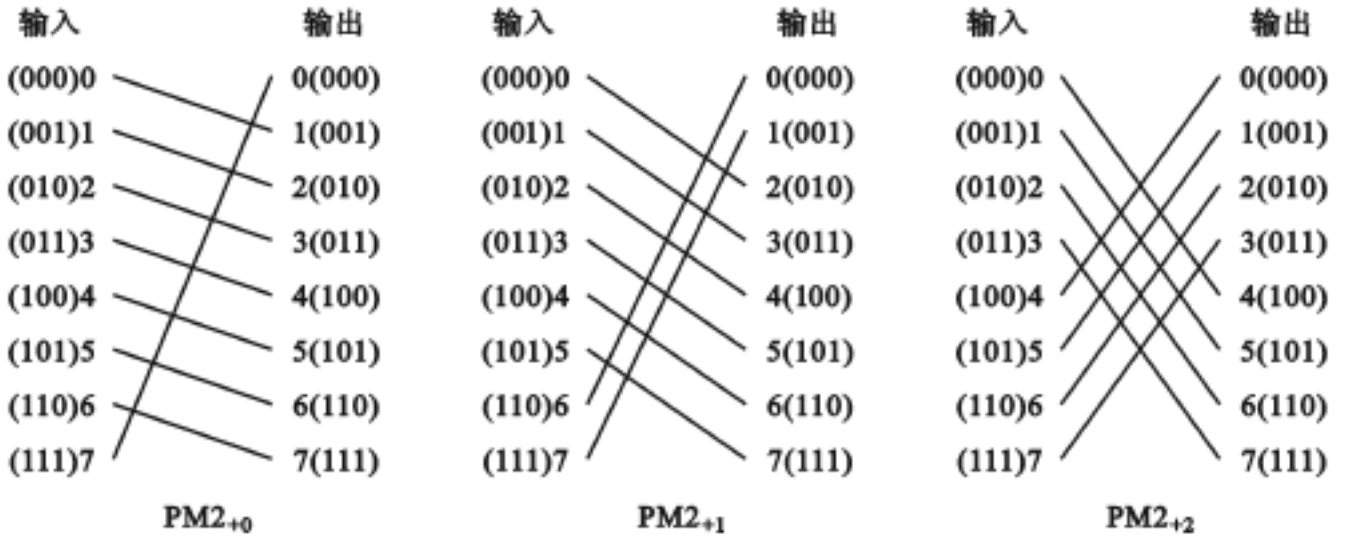


图 6.16　$N=8$ 个节点的 PM2I 置换

6.4.3　互连网络的分类和结构参数

互连网络是由开关元件按照一定的拓扑结构的控制方式构成的网络,用来实现计算机系统内部多个处理机或多个功能部件之间的相互连接。

1. 互连网络的分类

从几何形状看互连网络可分为两大类:规则网和不规则网。规则网又可分为静态网和动态网,如图 6 .17 所示。在分析这些网络时,把连接的对象(如处理机、运算部件、存储器等)称为节点,对应的单向或双向通路称为边。

静态网又叫直接互连网,又可进一步分为全互连网、星形网、总线、循环移数网、一维线性网、环形网、网格网、树形网、超立方体网、带环立方体网、m 元 n 立方体网、Paradhan 网等。

动态网又叫间接互连网,也可进一步分成交叉开关网、单级互连网、多级互连网。多级互连网又可分为阻塞网、可重排非阻塞网、非阻塞网。非阻塞网有三级 Clos 网。可重排非阻塞网有 Benes 网、字节分类网、Memphis 网。阻塞网又有 Ω 网、STAR-AN 网、间接二进制 n 方体网,基准网、δ 网、数据交换网等。

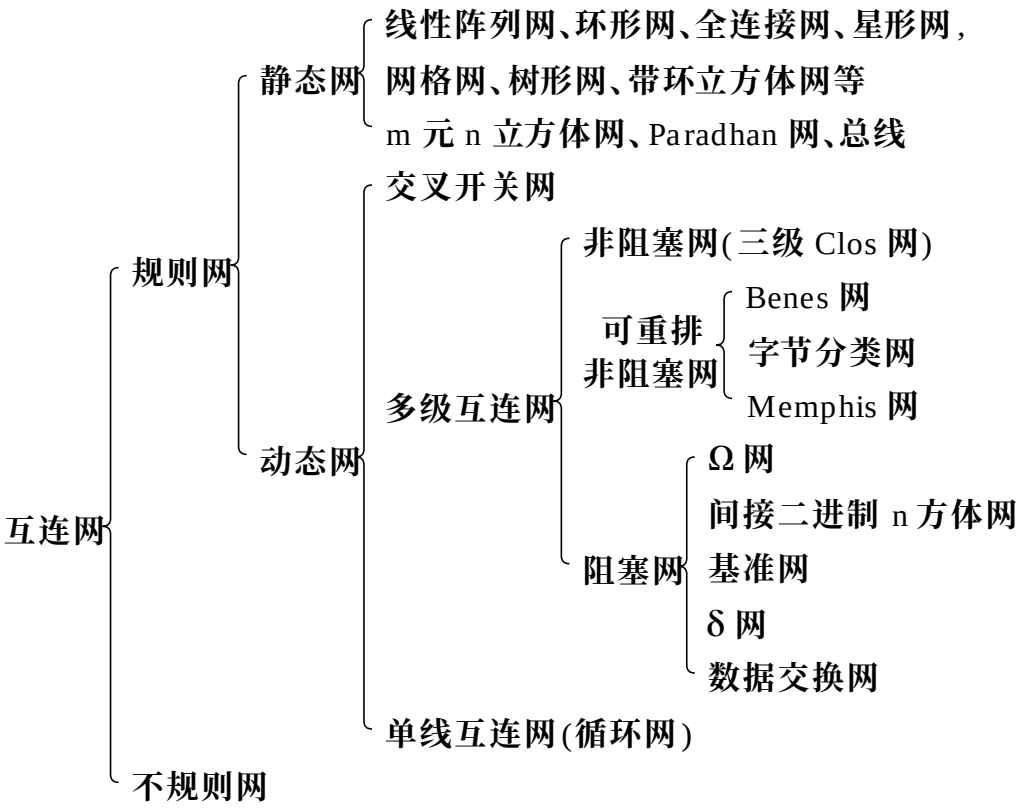


图 6 .17 互连网络的分类

2. 互连网络的结构参数

对于静态互连网络,在分析各种网络的拓扑结构前,先定义一些参数。一般网络是用有向边或无向边和连接的有限个节点组成的图来表示的。图中节点与处理机、功能部件相对应,边则与通信链路相对应。节点数称为网络的规模。

(1)度

在互连网络拓扑图中,节点所连接的边(链路或通道)数称为该节点的度 k。在

单向通道的情况下,进入节点的通道数叫入度,从节点出来的通道数叫出度。该节点的度就是二者之和。节点的度反映了节点所需 I/O 端口数,也代表了节点的价格。网络中各节点的度的最大值叫做该网络的度,记为 K。

(2)距离和直径

网络中,任意两节点间沿最短路径通信所经过的边数称为这两个节点间的距离,记为 d。任意两节点间距离的最大值称为该网络的直径,记为 D。

(3)中继量 R

由 n 个节点组成的网络,在各节点间有 $\frac{n(n-1)}{2}$ 条通信路径组合(不是边数,而是边的组合数)。在两节点之间肯定有一条最短的通信路径,也就是两节点间的距离 d。两节点间的中继次数等于这两节点间的距离减“1”。一个节点的中继量,是指除自己以外的其他节点通过自己节点的通信路径条数。网络中继量是指中继量最多的那个节点的中继量,记为 R。当然网络中继量越大,并行处理的效果就越差。如果中继量大多集中在部分节点上,就会造成这些节点上的瓶颈效应,影响整个系统并行处理的效果。所以希望平均中继量 r 能等于或接近最大中继量 R。

(4)通路的方向性

在互连网络中,通信链路可以是单工的,也可以是半双工或全双工的。单工用有向图表示,半双工或全双工用无向图表示。

(5)规则性和对称性

规则性是指网络节点互连遵循确定的规则。在规则性好的网络中,各节点连接的相邻节点数常常是相等的。任意两节点间通信的路径算法比较规则和简单。规则性还可以用对称性来描述。在全对称互连网络中,每个节点在图中的地位是等同的。环形网络就是全对称网络。半对称网络中只有部分节点在几何位置上与余下部分节点对称,但不存在全部节点地位等同的关系,树形网络就是半对称网络。

(6)广播时间

广播是指一个节点同时向其他所有节点发送相同的数据。常用于管理节点向其他作业节点发送程序或发送初始数据。广播时,各处理机节点一边中继一边接收,所以比分别向所有的节点传送数据要快得多,最好把管理节点放置在可以广播最快的位置上。从管理节点广播的数据,到达所有节点的传送次数,叫广播时间,记为 T。

以上各种参数从不同的侧面影响着网络的性能,一般我们要权衡以下因素:

① 网络的功能,是指网络所支持的互连函数、中断处理、同步、消息的组合和一致性问题,可根据系统需要选择合适的功能。

② 网络的延迟,是指单位信息通过网络时,最大的时间延迟。网络的延迟当然是越小越好,它与网络的直径、中继量及对称性有关。

③ 网络的带宽,指通过网络的最大传输率。

④ 网络的硬件复杂度,指的是开关以及连接器、仲裁及接口逻辑等的开销。这

与网络的度、功能等有关。

⑤ 网络的可扩展性,是指在网络拓扑性能保持不变的情况下,可扩充节点的能力。

此外,还要考虑网络节点的合理编址,路径的冗余度,节点是否为同构,通道是否有缓冲等各种因素。但这些因素往往是相互矛盾的,只能根据环境要求,统筹兼顾各种指标,寻求相对合理的方案。

下面我们开始分析静态互连网络和动态互连网络的拓扑结构。

6.4.4 静态互连网络

静态互连网络是节点之间直接互连。每个开关元件固定与一个节点相连,以建立该节点与邻近节点之间的被连接通路。这种网络一旦构成以后就固定不变了,比较适合于构成通信模式可预测的并行处理系统和分布计算机系统。

下面我们根据网络参数以及它们对网络通信和可扩展性的影响等几个方面来分析各种静态互连网络的拓扑结构。

1. 线性网和星形网

一维线性阵列是每个节点只与其左、右近邻节点相连(首尾例外)的最简单连接方式,如图 6.18(a)所示。

一维线性阵列网络的度 K = 2,直径 D = N - 1,等分宽度为 1,不对称。当 N 很大时,通信效率很低。

星形网络的拓扑结构如图 6.18(b)所示。一主节点处理机作为主控机位于中心位置,分别与其他所有从节点处理机相连接,这些从节点之间的通信必须通过主节点进行中继。因此,主节点为系统的瓶颈,对其工作性能要求非常高。一旦主节点处理机出现故障,整个系统都将瘫痪。

在星形网络的结构中,主节点度 k = N - 1,从节点度 k = 1。从节点间的通信距离 d = 2。

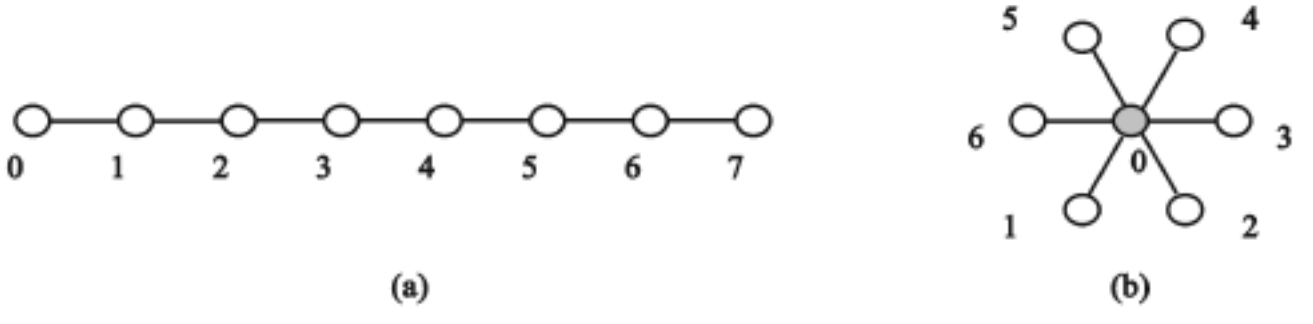


图 6.18 一维线性阵列网和星形网

2. 环网和带弦环网

如果把线性网的两端连在一起,便形成了环网,如图 6 .19(a)所示。环网缩短了线性阵列网的直径,而且结构也很简单。环网的节点度 k = 2,单向环网的直径 D = N - 1,双向环网的直径 D = N/ 2。如果要将网络的节点度提高,可用带弦环网来实现,如图 6 .19(b)所示。

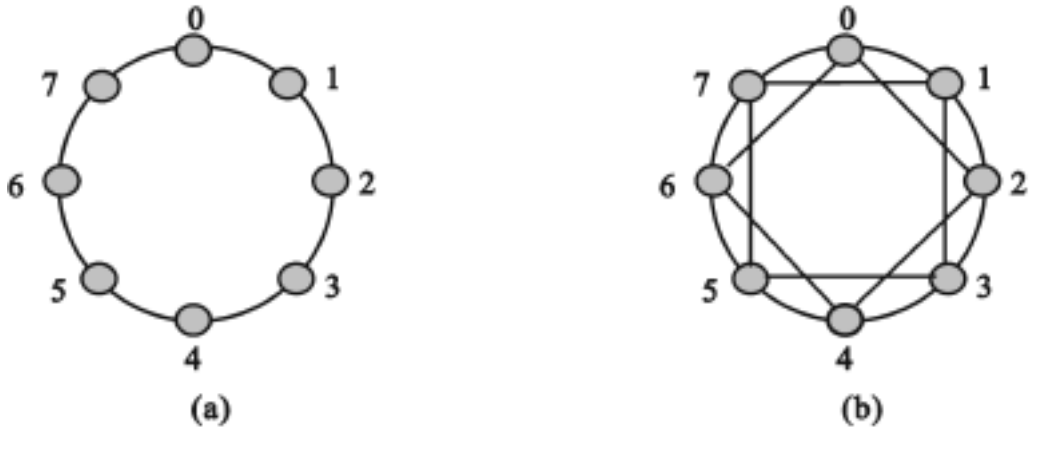


图 6 .19　环网和带弦环网

3. 循环移数网和全连接网

循环移数网是把环网上的每个节点到与其距离为 2 的整数幂的节点之间增加一条链路连接而成。图 6 .20(a)中的带弦环网也可看成是循环移数网,图中的每一节点与其相距 2^i 节点之间有一条链路。如果网络规模为 $N = 2^n$,那么网络节点度 k = 2n - 1,网络直径 D = n/ 2。

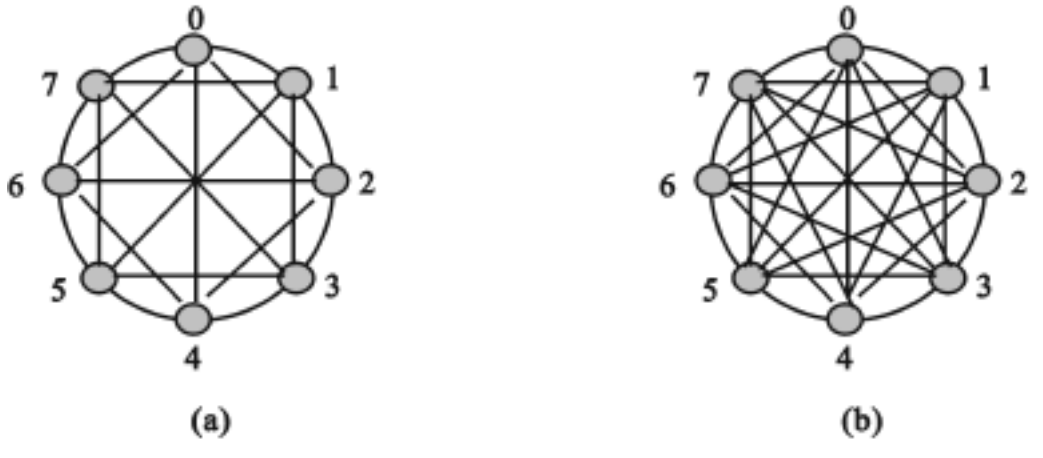


图 6 .20　循环移数网和全连接网

全连接网是指当带弦环网的节点度增加到 N - 1 时的连接网,如图 6 .20(b)所示。该网络的节点度为 N - 1,直径为 1。这种网络构成硬件开销较大,一般难以

实现。

4. 完全二叉树形网和二叉胖树形网

N 个节点的完全二叉树共有 k 层,其中,N = 2k - 1,如图 6 .21(a)所示,最大节点度为 3,网络直径是 2(k - 1)。由于节点度为一常数,因此它是一种可扩展的系统结构。

二叉胖树形网络如图 6 .21(b)所示。二叉胖树形网是在传统的完全二叉树形网基础上提出的,那是因为传统二叉树中的根节点是网络通信中的瓶颈,为了缓解这一矛盾,在越靠近根节点的地方,其链路数增加。

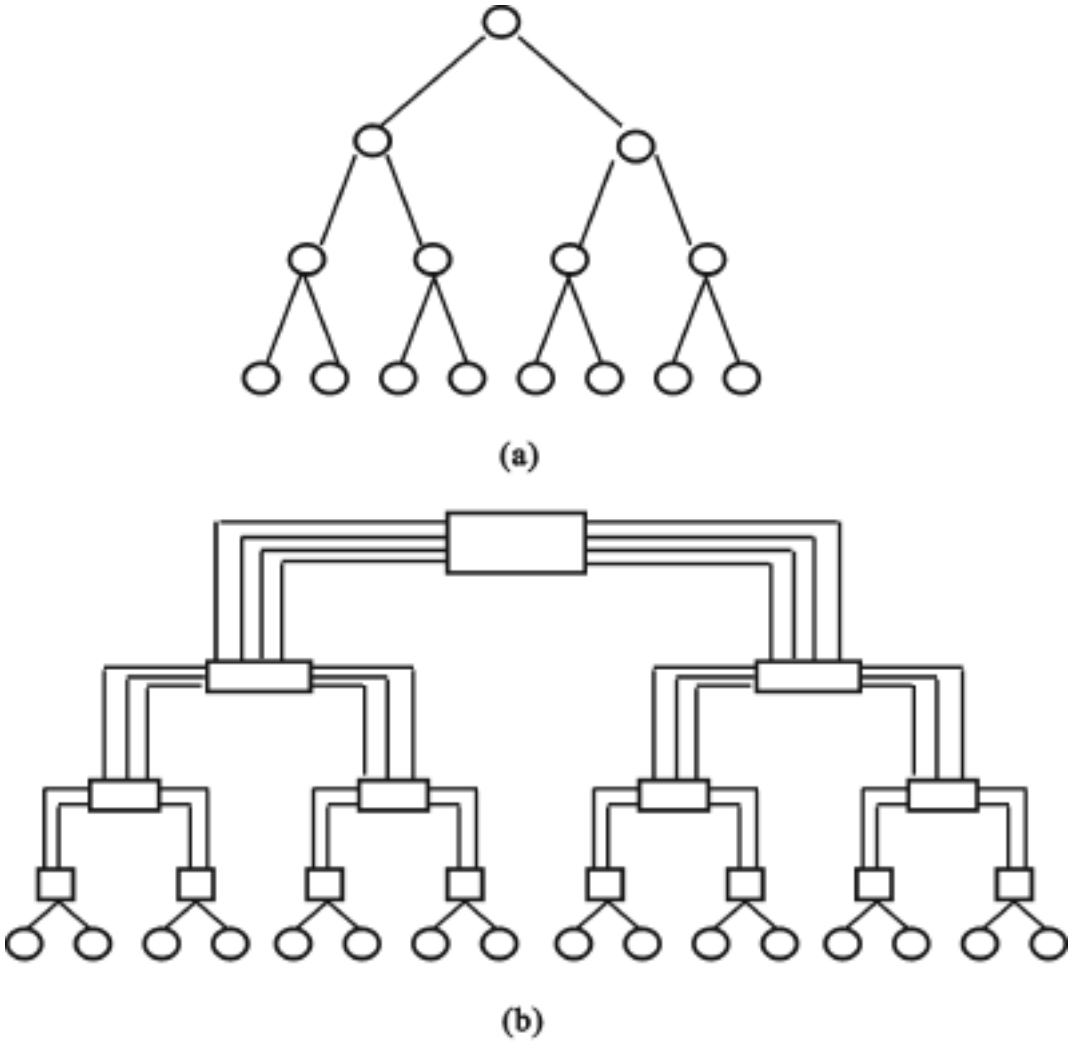


图 6 .21 完全二叉树形网和二叉胖树形网

5. 网格形网络

网格形网络是将邻近节点按一定规则的平面几何图形相互连接而成,如图 6 .22 所示。图 6 .22(a)为四边形网格,网络的度为 4、直径为 2 $\sqrt{N}$。图 6 .22(b)为六边形网格,网络的度为 3、直径为 2 $\sqrt{N}$。图 6 .22(c)为闭合螺线阵列网,由于这种网首先是在 ILLIAC-Ⅳ阵列机上实现的,故又称为 ILLIAC 网,网络的度为 4、直径为 $\sqrt{N}$ -

1。

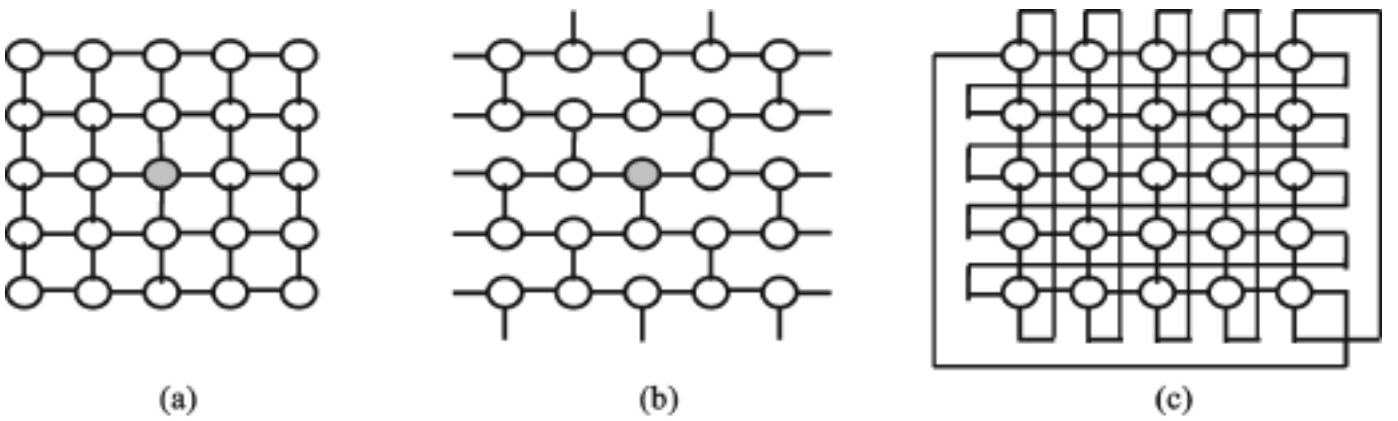


图 6 .22　四边形网、六边形网和闭合螺线阵列网

6. 立方体网络

最基本的立方体网络是二元三立方体网络,简称立方体网络或单级立方体网络。它是一种非常有用的拓扑结构,不仅在静态互连网络中使用,在动态互连网络中也广泛应用。

立方体网络可用方体互连函数来描述。

当 $N=8$ 时,共有 $n=3$ 个置换函数,它们分别为:

$$C_0(x_2\ x_1\ x_0) = x_2\ x_1\ \overline{x_0}\quad (x\text{ 方向的连接})$$

$$C_1(x_2\ x_1\ x_0) = x_2\ \overline{x_1}\ x_0\quad (y\text{ 方向的连接})$$

$$C_2(x_2\ x_1\ x_0) = \overline{x_2}\ x_1\ x_0\quad (z\text{ 方向的连接})$$

立方体网络如图 6 .23 所示。

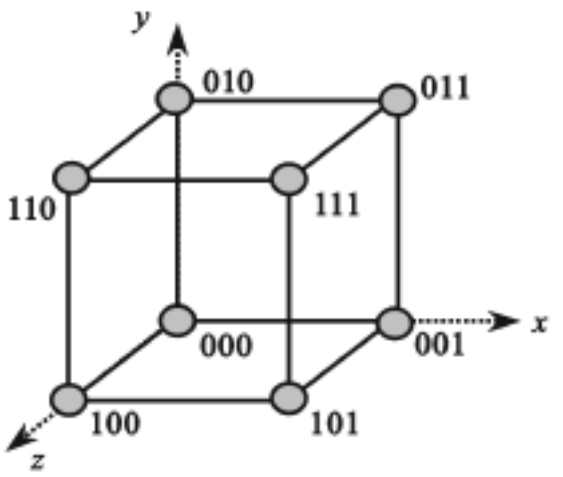


图 6 .23　立方体网络

在图 6. 23 中网络共有 8 个节点,每个节点处于一个三维立方体的顶点上。如果将这些节点的地址按一定规律编号,即用地址的二进制码相差一位的各对节点一一

相连,便构成了图中的立方体网。立方体网络的度为 3、直径也为 3。

立方体网络的寻径算法是:对源节点和目的节点的二进制地址逐位异或,然后根据所得的结果进行寻址。例如源节点地址为 111,目的节点地址为 100,二者异或的结果是 011。如果我们将节点地址的最低位定为 x 方向,中间位定为 y 方向,最高位定为 z 方向,那么在异或结果中,若某位为 0,表示在寻址过程中该位对应的方向上没有移动;若某位为 1,则表示在相应的方向上移动一步。在这个例子中,寻径过程是:从 111 节点出发,在 x, y 方向上各走一步,就到达了目的节点 100。

立方体网络的用途极广,可用于合并、FFT、置换、排序、卷积及矩阵等并行运算。其不足之处是网络的扩充性比较差,节点必须以 2^N 来增加,才能保证扩展后的网络仍是方体结构,例如超立方体网络等。

6.4.5 动态互连网络

动态互连网络是为了达到多用或通用的目的,根据程序要求实现各种通信模式的互连网络。在动态互连网络中,各节点之间不是直接固定连接的,而是在控制信号的作用下,通过网络开关的设置来建立节点之间间接的、可变的连接通路。

1. 互连网络中的三个参量

动态互连网络,特别是动态多级互连网络的结构可以用交换开关、连接模式(拓扑结构)和控制方式三个参量来描述。

(1)交换开关

动态互连网络中的交换开关是一个有源的开关元件,它可根据控制信号的不同,工作在各种不同的状态,从而实现输入端和输出端的不同连接。开关的输入端数 a 和输出端数 b 可以相等,也可以不相等。当二者相等时,一般选 $a = b = 2^k$, $k \geq 1$。常用的交换开关有 2×2,4×4,8×8 等。在这些开关中,每个输入可同时与一个或多个输出相连,但不能有两个以上的输入端与一个输出端同时相连。如果出现这种情况,就会发生冲突或争用。所谓冲突,是指要求将一个输入端连到一个输出端时,该输出端已被占用的状态。所谓争用,是指两个以上的输入端同时要求连至同一输出端的情况。当发生冲突或争用时,要采取相应的措施来解决。

现以 2×2 开关为例说明其工作状态。如图 6.24 所示,2×2 的交换开关有四种状态:

① 直连:是上输入端与上输出端相连,下输入端与下输出端相连。

② 交换:是上输入端与下输出端相连,下输入端与上输出端相连。

③ 上播:是上输入端与上、下两输出端同时相连,下输入端不用。

④ 下播:是下输入端与上、下两输出端同时相连,上输入端不用。

(2)连接模式

连接模式也称为网络中的拓扑结构,它是指多级互连网络的各级开关之间链路

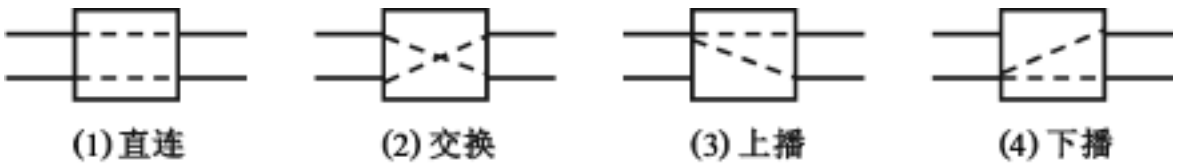


图 6.24　2×2 交换开关的四种工作状态

的互连模式。连接模式实现的互连函数不同,就可以构成各种不同的多级互连网络。

(3)控制方式

控制方式是指对网络中各级开关的控制方式,一般有三种控制方式:

① 级控方式:同级开关用一个信号来控制,所有开关都处于同一种工作状态。其优点是控制简单、实现容易,但网络能实现的互连函数少。

② 单元控制:每一个开关都有单独的控制信号,同一级的开关可以处于不同的工作状态,也可以处于相同的工作状态。其优点是网络实现的互连函数多,但控制复杂,实现较困难。

③ 部分级控制:用 i+1 个控制信号来控制第 i 级的所有开关,其中 $0 \leq i \leq n-1$,n 为级数。这种方式是介于上述两种控制方式之间的一种。

衡量和选择一个多级互连网络,一般从以下几个方面考虑:

① 连接特性好,即实现的互连函数多;

② 网络延时小;

③ 开关设备量少;

④ 控制简单;

⑤ 便于集成。

当然,以上几个方面有的是互相矛盾的,可根据自己的需要去综合权衡选择。

动态互连网络中的单级互连网络拓扑结构与静态互连网络拓扑结构相同,把静态网络的固定连接换成开关便是单级互连网络,所以有人把静态网络叫单级网络,动态网络叫多级网络。交叉开关网络既是单级网络又是动态网络。

2. 单级互连网络

单级互连网络是由一级开关组成的网络,它只能实现有限的几种基本连接,如前面所述各种静态网的连接形式,但一次通过不能实现任意节点间的互连。这里,一次通过是指从输入端到输出端能一次通过网络,无冲突地按某种置换实现数据传输。要实现任意节点间的互连有两种办法:一是循环使用同一套单级互连网络,组成循环互连网络,所以单级互连网络又叫循环网络;二是把多套相同或不同的单级互连网络串联起来,组成多级互连网络。如果把多级互连网络再循环使用,或背靠背连接,又可组成可重排非阻塞网络。

在这一小节中,我们只对组成多级互连网络的常用的几种单级互连网络进行分

析,最后介绍全连接的交叉开关网络。

(1)PM2I 单级互连网络

PM2I 网络是加减 2^i 网络的简称,又叫桶形移位器或循环移数网。当 N = 8 时,5 个互不相同的函数(用循环函数表示)如下:

$PM2_{+0}$:(0 1 2 3 4 5 6 7)

$PM2_{-0}$:(7 6 5 4 3 2 1 0)

$PM2_{+1}$:(0 2 4 6)(1 3 5 7)

$PM2_{-1}$:(6 4 2 0)(7 5 3 1)

$PM2_{\pm 2}$:(0 4)(1 5)(2 6)(3 7)

其互连网络的连接如图 6 .25 所示。

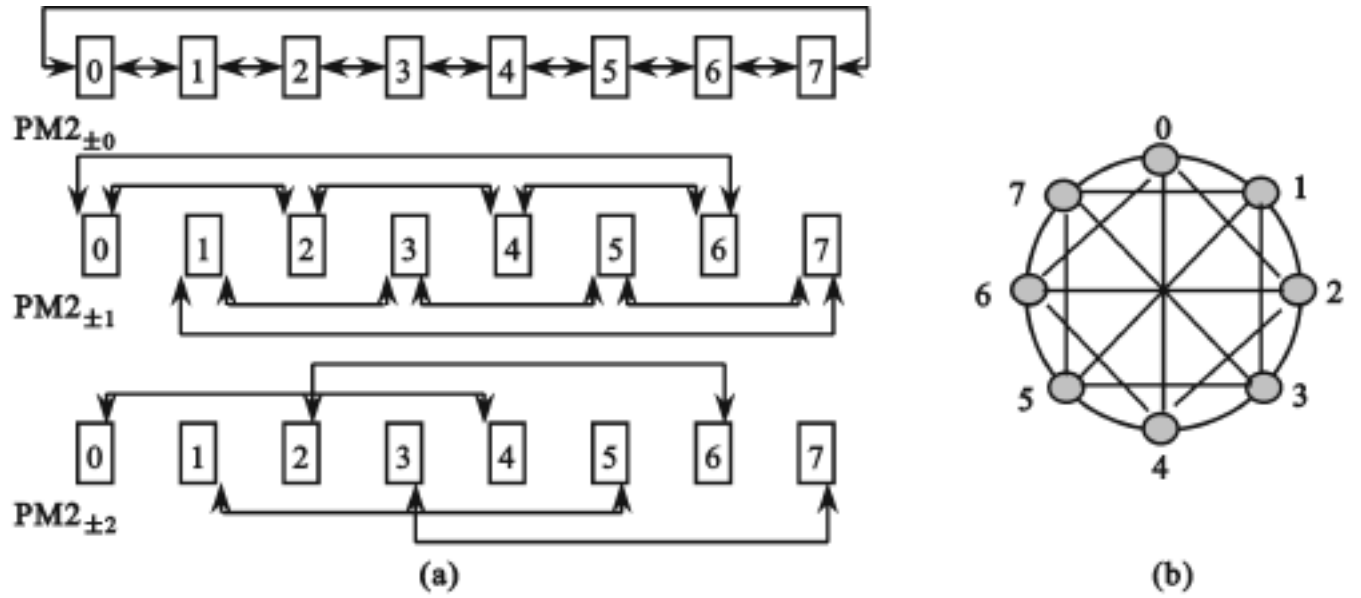


图 6 .25 N = 8 个节点的 PM2I 网络连接图

在图 6 .25(a)中,用 3 个图分别表示 $PM2_{\pm 0}$, $PM2_{\pm 1}$ 和 $PM2_{\pm 2}$ 的互连结构。在图 6 .25(b)中,用一个图综合表示 8 个节点的相互连接;各节点在圆周上的边,形成 $PM2_{\pm 0}$ 所表示的链路;由图中两个正方形构成的 8 条边,形成 $PM2_{\pm 1}$ 所表示的链路;而两个正方形的 4 条对角线,形成 $PM2_{\pm 2}$ 所表示的链路。

PM2I 网络也可以用非完全交叉开关矩阵表示为图 6 .26 的形式,其中交叉点上有标记的地方是可以相互连接的开关点。从某一源点,例如 0 节点开始,一步就可以直接到达的节点有 1,2,4,6,7。两步就可以实现任一输入节点到任一输出节点的连接。

在 ILLIAC-Ⅳ阵列处理机中,各处理单元之间的连接实际上就是采用了 PM2I 互连网络中 $PM2_{\pm 0}$ 和 $PM2_{\pm 3}$ 的 4 个互连函数,它是 PM2I 网络的特例。

(2)混洗(洗牌)交换网络

混洗交换网络是由均匀洗牌(全混洗)和交换两种互连网组成,简称 SE 网络。由于均匀洗牌网络不能实现二进制编号为全 0 或全 1 的节点与其他节点之间的连

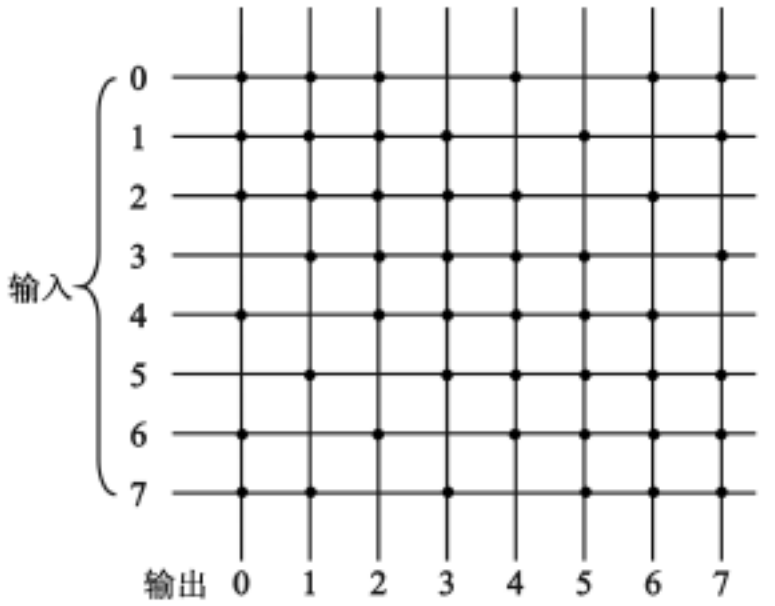


图 6.26　N = 8 的 PM2I 网络非完全交叉开关连接图

接,因此,必须增加交换连接,从而形成了混洗交换网络。

N = 8 时的混洗交换网络连接如图 6.27 所示,其中 $n = \log_2 N$。图 6.27(a)中虚线表示均匀洗牌,实线表示交换。可以看出,在混洗交换网络中,最远的两个输入输出端是全 0 和全 1。它们的连接需要 n - 1 次均匀洗牌和 n 次交换才能实现,所以其最大距离为 2n - 1。图 6.27(b)是 SE 网络的非完全交叉开关表示。

同样 SE 网络的所有节点均需使用相同的连接,即要洗牌都洗牌,要交换都交换。SE 网络也是后面的多级互连网络 Ω 网的基础。

(3)交叉开关网络

交叉开关网络的带宽和互连特性最好。一个 N×N 的交叉开关网络的连接方式有 N! 种。

N = 4 个节点的交叉开关网络如图 6.28(a)所示。在这种互连网络中,任何一个节点都有与其他节点的直接通路,所以只需经一步就可使源节点中的数据传送到网络中任一目标节点。N = 8 个节点的全交叉开关网络连接如图 6.28(b)所示,共有 N^2 个交叉开关。

如果交叉开关网络输入和输出端都连接处理机,则 N×N 的交叉开关网络一次最多只能连接 N 个输出对(即每一行和每一列只能接通一个交叉开关)。如果输入端连接处理机,输出端连接存储器模块,则为了支持并行(或交叉)存储器访问可以在一行和一列之间接通几个交叉开关。

交叉开关网络的优点是连接能力强,但硬件复杂性以 N^2 上升,造价很高,所以适合于规模小的系统。

3. 多级互连网络

前面介绍的单级互连网络(除交叉开关互连网络外)都只能实现有限几种基本连

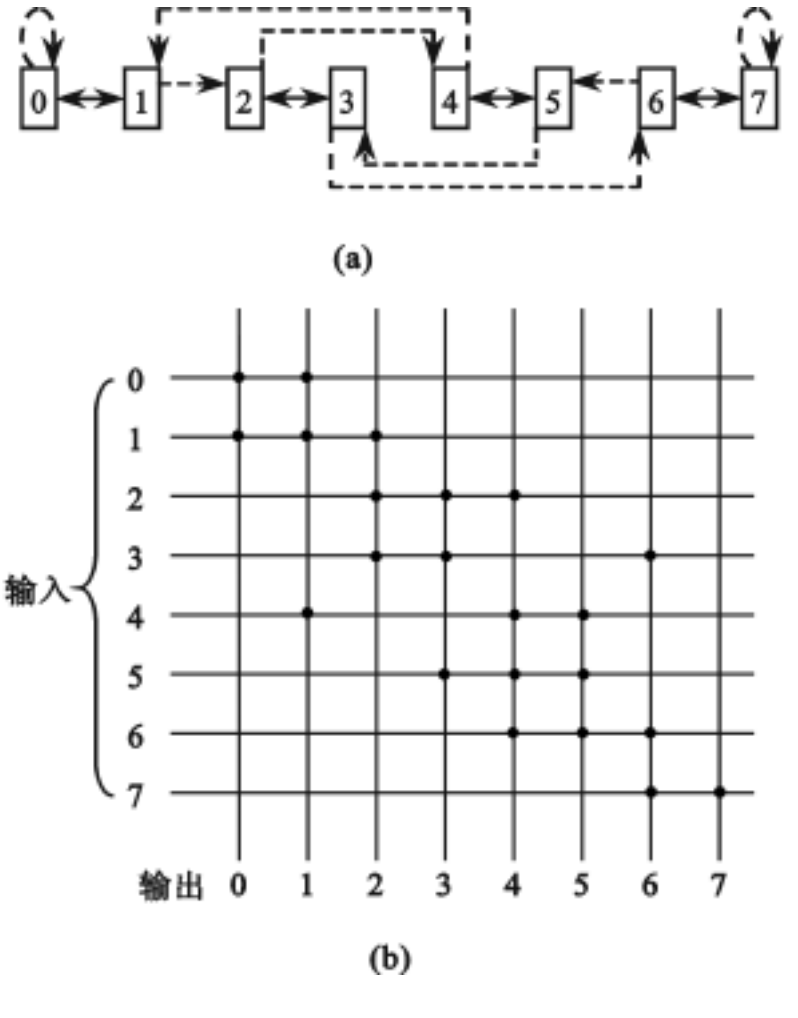


图 6.27 混洗交换网络连接图

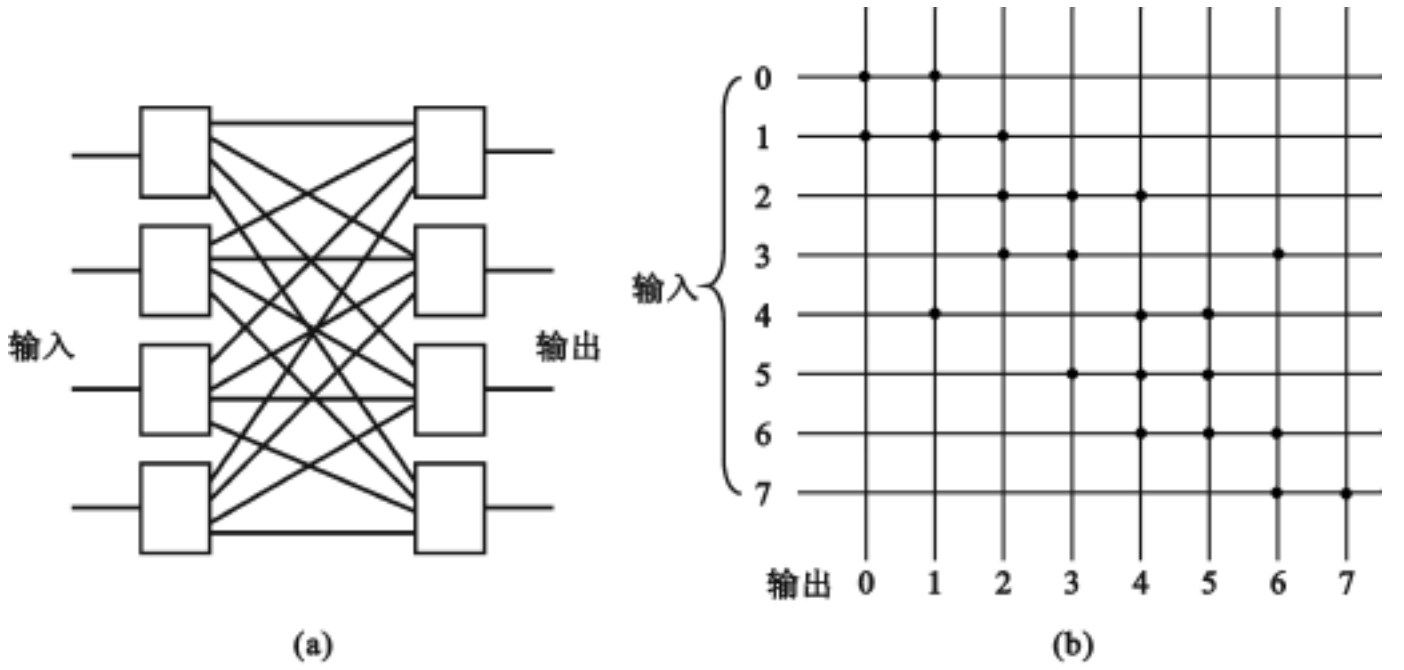


图 6.28 交叉开关网络连接图

接,不能实现任意节点对之间的连接。多级互连网络是在空间上重复设置多套单级网络,单级网络级间采用固定的模式串联,通过动态设置各级开关的状态来实现任意输入和输出节点对之间的所需连接。

在下面介绍的几种典型的多级互连网络中,分别按结构特点、控制方式和典型应用等方面进行介绍。

(1)STARAN 网络

STARAN 网络的结构特点:

① 网络规模:N 个节点的网络由 $n = \log_2 N$ 级构成,每级开关的编号从输入端到输出端依次为 $K_0, K_1, \cdots, K_{n-1}$;每级的交换开关数为 $N/2$ 个,每个交换开关都是二功能的(即直连和交换)。整个网络的开关数为 $(N/2) \times \log_2 N$ 个。

② 拓扑结构:N 个节点的网络共有 n + 1 级拓扑结构(连接模式),其编号从输入端到输出端依次为 $C_0, C_1, \cdots, C_n$;C_0 为恒等置换,$C_1 - C_n$ 为逆混洗置换。

N = 8 个节点的 STARAN 网络结构如图 6.29 所示。其中,N = 8 时的逆混洗置换函数表达式为:

$$S^{-1}(x_2\ x_1\ x_0) = x_0\ x_2\ x_1$$

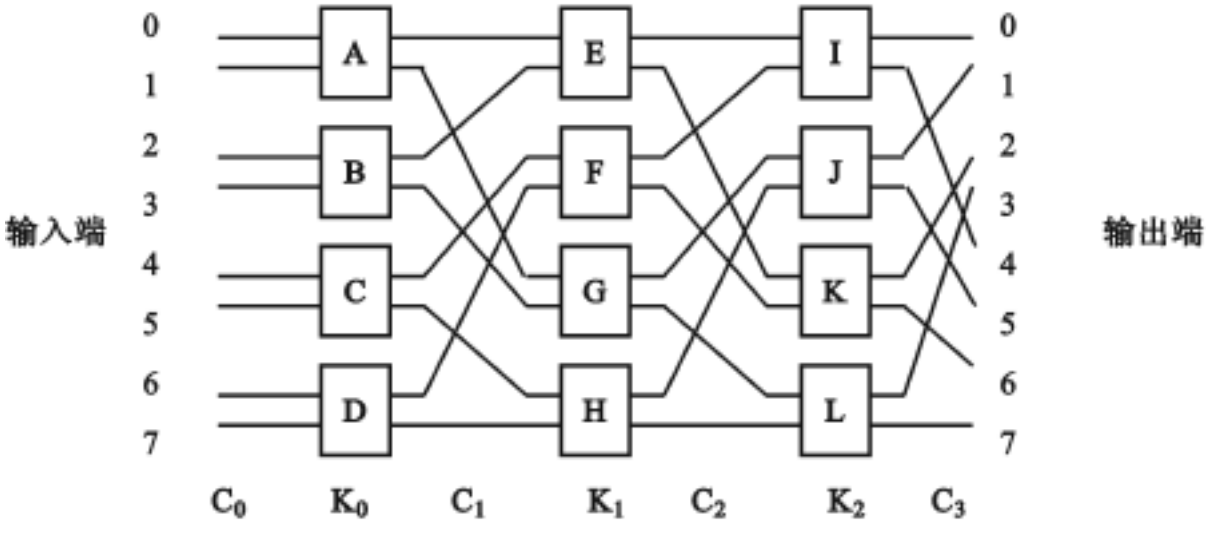


图 6.29 N = 8 个节点的 STARAN 网络

STARAN 网络的控制方式:级控和部分级控。

STARAN 网络的典型应用:交换置换和移数置换。

① 交换网络:STARAN 网络用做交换网络时,采用级控方式工作,实现的是交换函数的功能。所谓交换函数是将一组元素首尾对称地进行交换。表 6.1 列出了三级交换网络在级控信号采用各种不同组合情况下所实现的输入与输出端之间的连接。

表 6.1　　三级 STARAN 网络实现交换函数功能(k_i 为第 i 级控制信号)

		级控信号(k_2 k_1 k_0)							
		000	001	010	011	100	101	110	111
输入端号	0	0	1	2	3	4	5	6	7
	1	1	0	3	2	5	4	7	6
	2	2	3	0	1	6	7	4	5
	3	3	2	1	0	7	6	5	4
	4	4	5	6	7	0	1	2	3
	5	5	4	7	6	1	0	3	2
	6	6	7	4	5	2	3	0	1
	7	7	6	5	4	3	2	1	0
执行交换函数功能	恒等	4 组 2 元	4 组 2 元 + 2 组 4 元	2 组 4 元	2 组 4 元 + 1 组 8 元	4 组 2 元 + 2 组 4 元 + 1 组 8 元	4 组 2 元 + 1 组 8 元	1 组 8 元	
	I	C_0	C_1	$C_0 + C_1$	C_2	$C_0 + C_2$	$C_1 + C_2$	$C_0 + C_1 + C_3$	

由表 6.1 可以看出:

当控制信号为 001 时,完成对这 8 个处理单元(元素)的 4 组 2 元交换,相当于 N = 8时的方体 C_0 置换。其输入/ 输出端的对应关系为:

输入端排列:	0	1	2	3	4	5	6	7
输出端排列:	1	0	3	2	5	4	7	6

当控制信号为 010 时,完成的功能相当于先 4 组 2 元交换,后 2 元 4 组交换。其输入/ 输出端的对应关系为:

输入端排列:	0	1	2	3	4	5	6	7
先 4 组 2 元:	1	0	3	2	5	4	7	6

后 2 组 4 元:	1	0	3	2	5	4	7	6
输出端排列:	2	3	0	1	6	7	4	5

当控制信号为 111 时,实现的是全交换,也称为镜像交换。总之,不管控制信号是

什么状态,实现的都是交换函数的功能。从表中水平方向不难看出,任何输入端只要通过不同的级控信号,总可以连接到任何所需要的输出端上。

② 移数网络:当 STARAN 网络采用部分级控时,实现的是移数函数的功能,故称为移数网络。这时的控制信号和输入/输出端的连接情况列于表 6.2 中。

表 6.2　　三级 STARAN 网络实现移数函数功能

部分级控信号	2 级	K,L	0	0	1	0	0	0	0
		I	0	1	1	0	0	0	0
		J	1	1	1	0	0	0	0
	1 级	F,H	0	1	0	0	1	0	0
		E,G	1	1	0	1	1	0	0
	0 级	A,B,C,D	1	0	0	1	0	1	0
输入端号		0	1	2	4	1	2	1	0
		1	2	3	5	2	3	0	1
		2	3	4	6	3	0	3	2
		3	4	5	7	0	1	2	3
		4	5	6	0	5	6	5	4
		5	6	7	1	6	7	4	5
		6	7	0	2	7	4	7	6
		7	0	1	3	4	5	6	7
执行的移数功能			移 1 mod 8	移 2 mod 8	移 4 mod 8	移 1 mod 4	移 2 mod 4	移 1 mod 2	不移 恒等

当采用部分级控时,按照部分级控的定义,即第 i 级应有 i + 1 个控制信号。在 N = 8的 STARAN 网络中,第 0 级有 1 个控制信号,控制第 0 级上的 4 个交换开关 A,B,C 和 D;第 1 级有 2 个控制信号,分别控制 F,H 和 E,G 交换开关;第 2 级有 3 个控制信号,分别控制 K,L 和 I 及 J 交换开关。

一个 N×N 规模的移数网络,能够实现$(n^2 + n + 2)/2$种移数置换。N = 8 时的 STARAN 移数网络能实现 7 种移数置换,这 7 种移数置换在 7 组不同的控制信号下完成相对应的移数功能。

(2)间接二进制 n 方体网络

间接二进制 n 方体网络的结构特点:

① 网络规模:N 个节点的网络由 $n = \log_2 N$ 级构成,每级开关的编号从输入端到输出端依次为 $K_0, K_1, \cdots, K_{n-1}$;每级的交换开关数为 N/2 个,每个交换开关都是二功能的(即直连和交换)。整个网络的开关数为$(N/2) \times \log_2 N$个。

② 拓扑结构:N 个节点的网络共有 n + 1 级拓扑结构,其编号从输入端到输出端依次为 $C_0, C_1, \cdots, C_n$;C_0 为恒等置换,$C_1 \sim C_{n-1}$ 为子蝶式置换,C_n 为逆混洗置换。

N = 8 个节点的间接二进制 n 方体网络结构如图 6.30 所示。其中,N = 8 时的子蝶式置换函数有 3 个,表达式分别为:

$$B_0(x_2\ x_1\ x_0) = x_2\ x_1\ x_0,\ B_1(x_2\ x_1\ x_0) = x_2\ x_0\ x_1,\ B_2(x_2\ x_1\ x_0) = x_0\ x_1\ x_2$$

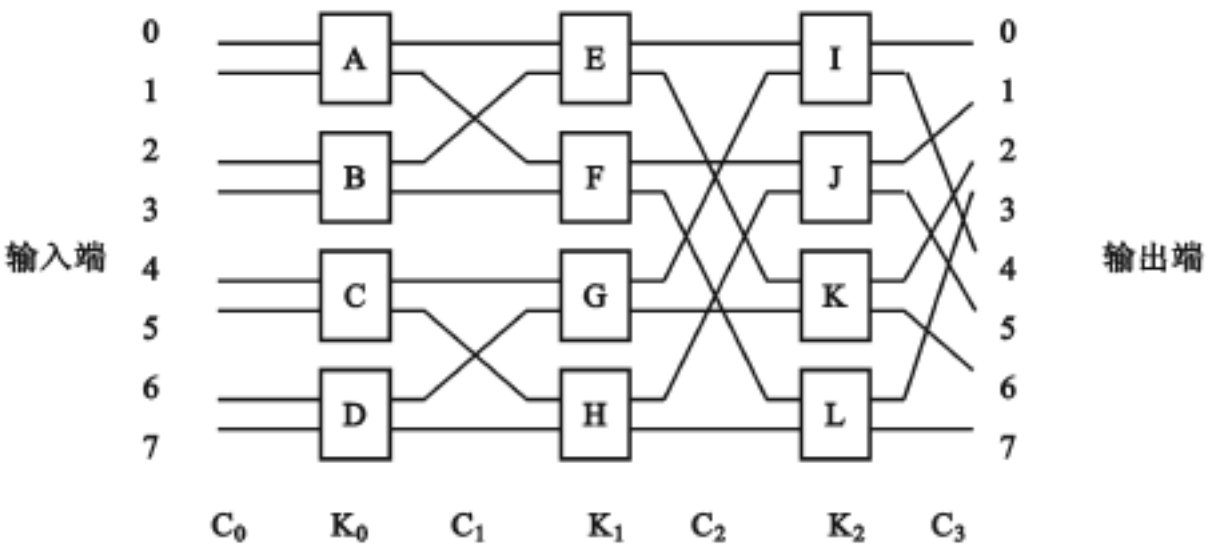


图 6.30 N = 8 个节点的间接二进制 n 方体网络

间接二进制 n 方体网络的控制方式:单元控制,即每一个交换开关有一个控制信号。

间接二进制 n 方体网络的典型应用:可实现交换置换、移数置换、子移数置换、P 序置换、逆 P 序置换、P 序加移数置换等。

(3)Ω(Omega)网络

Ω 网络的结构特点:

① 网络规模:N 个节点的网络由 $n = \log_2 N$ 级构成,每级开关的编号从输入端到输出端依次为 $K_{n-1}, K_{n-2}, \cdots, K_0$;每级的交换开关数为 N/2 个,每个交换开关都是 4 功能的(即直连、交换、上播和下播)。整个网络的开关数为 $(N/2) \times \log_2 N$ 个。

②拓扑结构:N 个节点的网络共有 n + 1 级拓扑结构,其编号从输入端到输出端依次为 $C_n, C_{n-1}, \cdots, C_0$;$C_0$ 为恒等置换,$C_1 \sim C_n$ 为全混洗置换。

N = 8 个节点的 Ω 网络结构如图 6.31 所示。

Ω 网络的控制方式:单元控制。

Ω 网络的典型应用:恒等置换、移数置换等各种函数的变形置换;可完成数组按行、列、对角线、子块等无冲突访问。

Ω 网络的寻径算法:设网络输入端地址编号为 $S = s_{n-1}s_{n-2}\cdots s_1 s_0$,输出端地址编号 $D = d_{n-1}d_{n-2}\cdots d_1 d_0$。从输入端开始,每一级开关状态由终端地址所对应位控制。若终端地址某位为 0,则对应级上开关的输入端与上输出端相连;若某位为 1,则对应

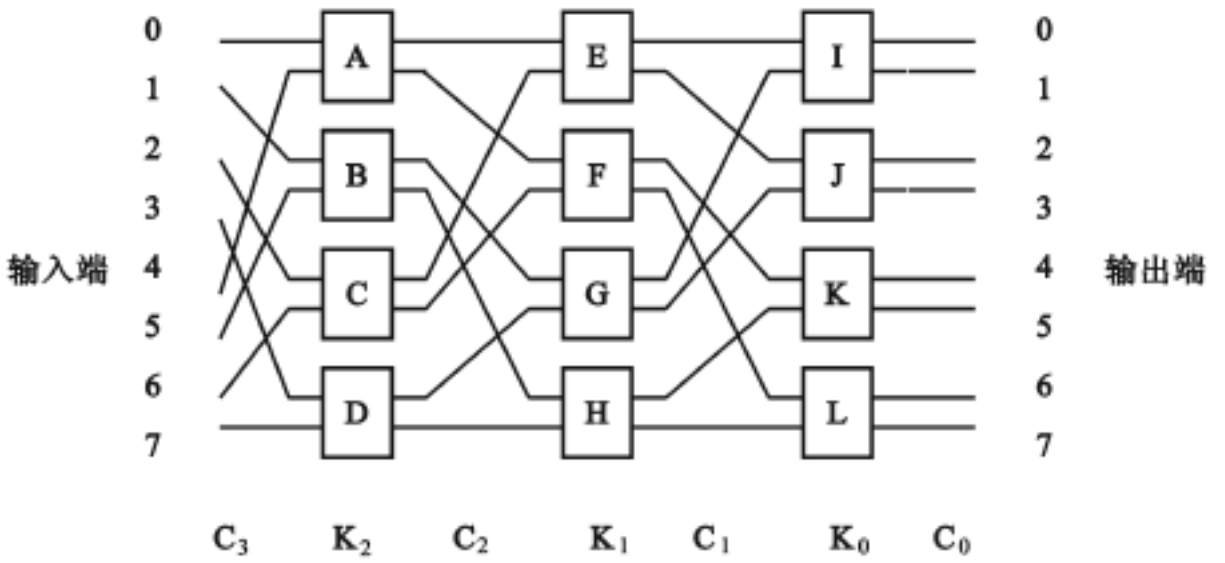


图 6.31 N = 8 个节点的 Ω 网络

级上开关的输入端与下输出端相连。

这种算法称为“终端地址控制”的寻径算法。该算法也适合于单元控制方式的间接二进制 n 方体网络中的地址寻径。

例如,在图 6.32 所示的 Ω 网络中,如果要实现输入节点 2 到输出节点 6 之间的连接,寻找路径的方法是:由于终端地址为 D = 6,即 $D = d_2 d_1 d_0 = 110$,所以用 $d_2 = 1$ 去控制 K_2 级上的交换开关,用 $d_1 = 1$ 去控制 K_1 级上的交换开关,用 $d_0 = 0$ 去控制 K_0 级上的交换开关。从源端 2 开始,开关 C 的输入端与下输出端相连($d_2 = 1$);经 C_2 级网络拓扑连至开关 F,由于 $d_1 = 1$,F 开关的输入端与下输出端相连;经 C_1 级网络拓扑连至开关 L,由于 $d_0 = 0$,L 开关的输入端与上输出端相连;经 C_0 级网络拓扑连至 6 号输出节点。这样就完成了节点对(2,6)之间的连接寻径。

对网络的源端集合到终端集合的连接,同样也可以用上述终端标记法来控制开关的状态。但由于每一个源端和终端节点对之间的连接路径是惟一的,所以不能保证所有开关的状态不发生冲突。例如,要实现(000,000),(100,010)两对同时连接,就会发生 K_2 级开关 A 的冲突,如图 6.32 所示。

同样也不能用 Ω 网络来实现均匀洗牌、蝶式和位序颠倒等变换,所以说 Ω 网络是一种阻塞网络。

(4)基准网络

基准网络的结构特点:

① 网络规模:N 个节点的网络由 $n = \log_2 N$ 级构成,每级开关的编号从输入端到输出端依次为 $K_0, K_1, \cdots, K_{n-1}$;每级的交换开关数为 $N/2$ 个,整个网络的开关数为 $(N/2) \times \log_2 N$ 个。

② 拓扑结构:N 个节点的网络共有 n + 1 级拓扑结构,其编号从输入端到输出端

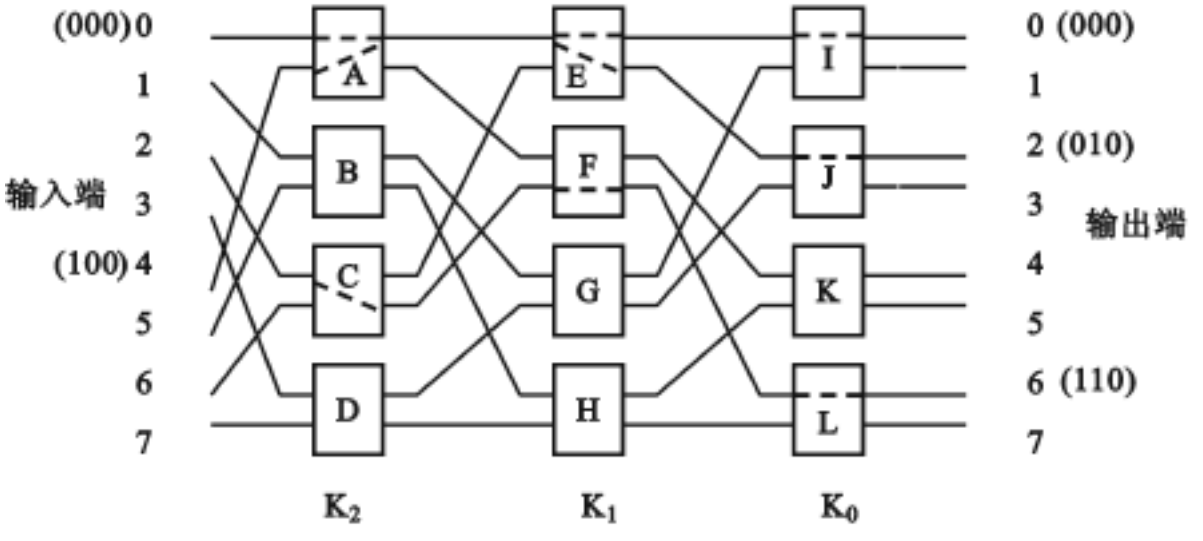


图 6.32　Ω网络上的寻径算法举例

依次为 $C_0, C_1, \cdots, C_n$；C_0 和 C_n 为恒等置换，C_1 是逆混洗置换，$C_2 \sim C_{n-1}$ 为子逆混洗置换。

N = 8 个节点的基准网络结构如图 6.33 所示。其中，N = 8 个节点时只有 C_2 级为子逆混洗置换。

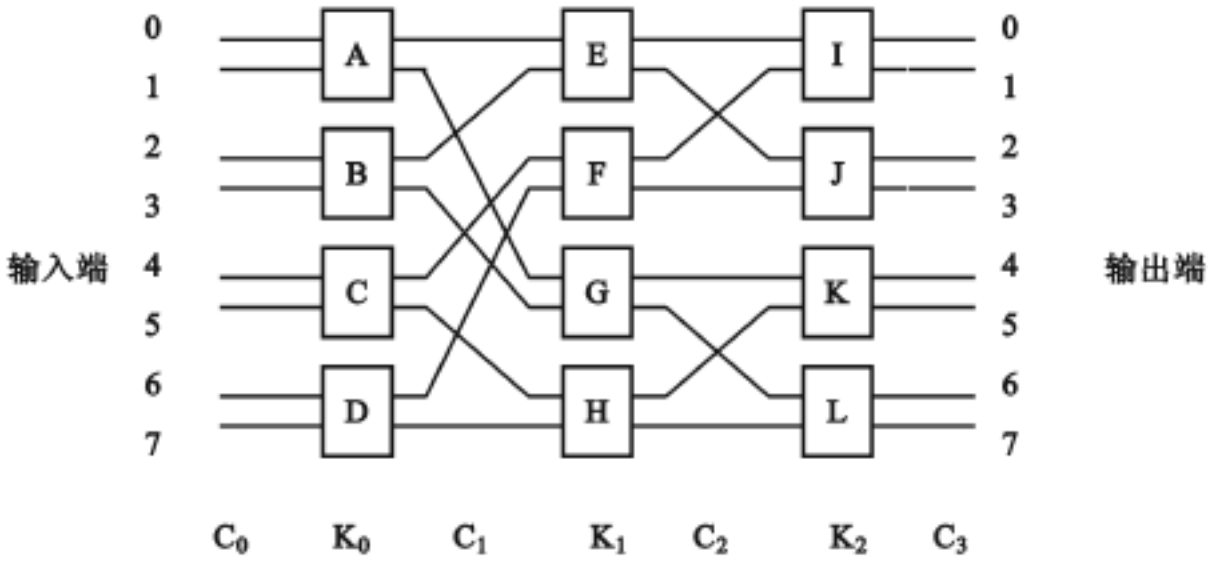


图 6.33　N = 8 个节点的基准网络

基准网络的控制方式：单元控制，可以采用终端地址标记的寻径算法。

基准网络的典型应用：一次通过基准网络可以实现位序颠倒置换，对实现 FFT 很有利，二次通过基准网络可以实现任意置换。

(5)全排列网络

前面所介绍的各种基本多级网络都能实现任意一个输入端与任意一个输出端间

的连接,但要同时实现两对或多对输入、输出端间的连接时,都有可能发生争用数据传送路径的冲突。我们称有这类性质的互连网络为阻塞式网络(Blocking Network),称无这类性质的互连网络为非阻塞式网络或全排列网络。非阻塞式网络连接的灵活性好,但连线数多、控制复杂、成本高。

如果互连网络是从 N 个输入端到 N 个输出端的一到一的映射,就可以把它看成是对此 N 个端的重新排列,因此互连网络的功能实际上就是用新排列来置换 N 个输入端原有的排列。

阻塞式网络在一次传送中不可能实现 N 个端的任意排列。大家知道 N 个端的全部排列有 N ! 种。可是,用单元控制的 $n = \log_2 N$ 级间接二进制 n 方体网络,每级有 N/2 个开关,n 级互连网络共用交换开关总数为 $(N \cdot \log_2 N)/2$。实现输入与输出端的一对一映射,每个开关只能用直连和交换二种功能,这样所有开关处于不同状态的总数最多只有 $2^{(N\log_2 N)/2}$,即 $N^{N/2}$ 种。当 N 为大于 2 的整数时,总有 $N^{N/2} < N!$,就是说它无法实现所有 N ! 种排列。以 N = 8 个节点的 3 级互连网络为例,共 12 个二功能交换开关,只有 $2^{12} = 4\ 096$ 种不同状态,最多只能控制对端子的 4 096 种排列,不可能实现全部 8 ! = 40 320 种排列,所以多对输入输出端要求同时连接时就可能发生冲突。

用两种方法可构成全排列网络:

一种方法是对某个多级互连网络通行二次,每次通行时让各开关处于不同状态就可满足对 N 个端子的全部 N ! 种排列。因为此时全部开关的总状态数有 $N^{N/2} \cdot N^{N/2} = N^N$ 种,因为 $N^N > N!$,足以满足 N ! 种不同排列的开关状态要求。这种只要经过重新排列已有输入输出端对的连接,就可完成所有可能的输入与输出端对的连接而不发生冲突的互连网络称为可重排列网络(Rearrangeable Network)。实现时,可以在上述任何一种基本多级互连网络的输出端设置锁存器,使数据在时间上顺序通行二次,这实际上就是循环互连网络的实现思路。

另一种方法是用 $\log_2 N$ 级的 N 个输入端和 N 个输出端的互连网络和它的逆网络连在一起,省去中间完全重复的一级,就可以得到总级数为 $2\log_2 N - 1$ 级的全排列网络。图 6 .34 就是以三维立方体多级网络和它的逆网络连在一起,省出中间重复的一级后构成的全排列网络,称此网络为 Benes 网络。

全排列网络的开关控制算法是改进的终端标记算法。在多级混交换(Ω)网络的终端标记寻径控制算法中规定,第 i 级的开关状态用终端地址的第 i 位 d_i 来控制。若 $d_i = 0$,则相应开关的输入端连接上输出端;若 $d_i = 1$,则相应开关的输入端连接下输出端。这样,如果同一个开关的两个输入端的终端地址 d_i 都等于 0,那么这个开关就会发生两个输入端要求同时连接一个上输出端而发生连接冲突。同样,如果同一个开关的两个输入端的终端地址 d_i 都是 1,那么这个开关就会发生两个输入端要求同时连接一个下输出端而发生连接冲突。

改进的终端地址标记控制的寻径算法只用一个输入端的控制信号 d_i 来控制开

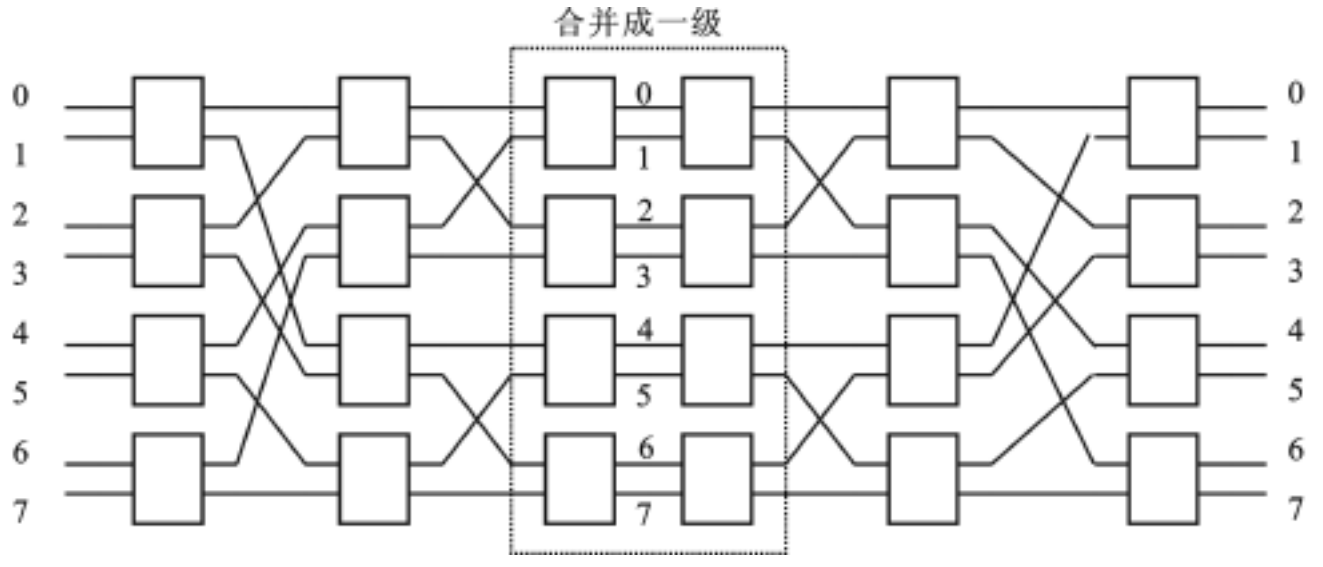


图 6 .34　多级全排列网络举例(Benes 网络)

关的工作状态。如果用上输入端的 d_i 来控制,则当 $d_i = 0$ 时,开关处于直连状态;当 $d_i = 1$ 时,开关处于交换状态;如果用开关的下输入端的 d_i 来控制,则当 $d_i = 0$ 时,开关处于交换状态,当 $d_i = 1$ 时,开关处于直连状态。值得注意的是,虽然可用开关的上、下输入端控制开关的工作状态,但在一次置换中,只能采用一种控制方案。

例如,用改进终端地址标记控制的寻径算法实现在 Benes 网络上的位序颠倒置换。现采用上输入端的控制方法分析各开关工作状态如下:

K_0 级共有 4 个交换开关:开关 1 的上输入端所要连接的终端地址 $d_0 = 0$,开关 1 为直连工作状态;开关 2 的上输入端所要连接的终端地址 $d_0 = 0$,开关 2 也为直连工作状态;开关 3 的上输入端所要连接的终端地址 $d_0 = 1$,开关 3 为交换工作状态;开关 4 的上输入端所要连接的终端地址 $d_0 = 1$,开关 4 也为交换工作状态。

K_1 级上也有 4 个交换开关:开关 5 的上输入端通过开关 1 与 0 号输入端相连,它所要连接的终端地址 $d_1 = 0$,故开关 5 为直连工作状态;开关 6 的上输入端通过开关 3 与 5 号输入端相连,它所要连接的终端地址 $d_1 = 0$,故开关 6 也为直连工作状态;开关 7 的上输入端通过开关 1 与 1 号输入端相连,它所要连接的终端地址 $d_0 = 0$,故开关 7 为直连工作状态;开关 8 的上输入端通过开关 3 与 4 号输入端相连,它所要连接的终端地址 $d_1 = 0$,故开关 8 也为直连工作状态。

同理可知,在 K_2 级上,由于开关 9 的上输入端通过开关 5 和开关 1 与 0 号输入端相连,而 0 号输入端所要连接的终端地址 $d_2 = 0$,开关 9 为直连工作状态……依此类推,开关 10 为直连工作状态;开关 11 和开关 12 为交换工作状态。K_3 级上的 4 个交换开关均为直连状态。K_4 级上的 4 个开关的工作状态分别为:17,18 为直连工作状态,19,20 为交换状态,如图 6 .35 所示。

由图可知,它可以实现输入与输出之间的位序颠倒置换。如在该网络上要实现输入端 3 与输出端 6 之间的连接,源地址的二进制为 011,而终端地址为 $d_2\ d_1\ d_0$ =

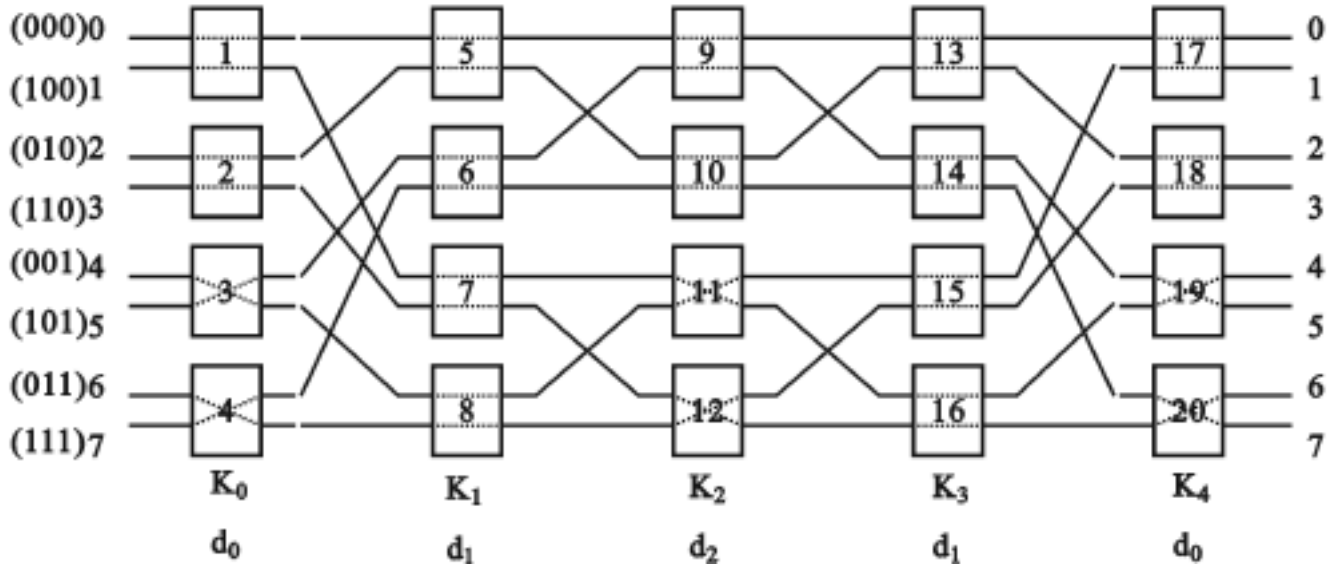


图 6.35 用改进的终端标记控制算法实现位序颠倒置换

110,它们之间的连接路径上所通过的开关为 2,7,12,16 和 20。

6.5 多处理机

多处理机是指具有两台以上的处理机,在操作系统控制下通过共享的主存或输入/输出子系统或高速通信网络进行通信。使用多处理机的目的:一是用多台处理机进行多任务处理协同求解一个大而复杂的问题来提高速度,二是依靠冗余的处理机及其重组来提高系统的可靠性、适应性和可用性。因为应用目的和结构不同,多处理机可以有同构型、异构型和分布型三种。本节只介绍为实现作业、任务间并行提高速度为目标的多处理机。

6.5.1 多处理机的特点

多处理机属多指令流多数据流(MIMD)系统,与单指令流多数据流(SIMD)系统的并行处理机相比,具有如下特点:

(1)结构灵活性

SIMD 并行处理机的结构主要是针对向量、数组处理设计的,只能开发其数据的并行性。在 MIMD 多处理机中存在有多指令流,因而在不同的处理单元上可执行不同的指令,可适应多样的算法,结构应更灵活多变,以实现复杂的机间互连,避免争用共享的硬件资源。

(2)程序并行性

SIMD 并行处理机实现操作级并行,一条指令即可对整个数组同时处理,并行性存在于指令内部,识别比较容易。MIMD 多处理机并行性存在于指令外部,表现于多个任务的并行,加上系统要求通用,使程序并行性的识别较难,必须利用算法、程序

语言、编译、操作系统以及指令、硬件等多种途径挖掘各种潜在的并行性。

(3)并行任务派生

SIMD 并行处理机由指令反映数据间能否并行计算,并启动多个处理单元并行工作。而 MIMD 多处理机需要有专门指令或语句指明程序中各程序段的并发关系,控制它们并发执行,使一个任务执行时可派生出另一些任务与它并行执行。因此,多处理机运行效率较并行处理机要高一些。

(4)进程同步

SIMD 并行处理机实现指令内数据操作的并行。所有活跃的处理单元在同一控制器控制下同时执行同一条指令,工作自然是同步的。MIMD 多处理机实现的是指令、任务、作业的并行。不同处理机执行的是不同指令,工作进度可以不一致。但若并发程序之间有数据相关或控制依赖,就要采取同步措施,使并发进程能按所需的顺序执行。

(5)资源分配和任务调度

SIMD 并行处理机主要执行向量、数组运算,处理单元数目是固定的。程序员编程时,用屏蔽手段设置处理单元的活跃状态,就可改变实际参加并行执行的处理单元数。MIMD 多处理机执行并发任务,需用处理机的个数没有固定要求,各个处理机进入或退出任务以及所需资源变化的情况要复杂得多。这就需要解决好资源分配和任务调度,让处理机的负荷均衡。尽可能提高系统硬件资源的利用率,管理和保护好各处理机、进程共享的公用单元,防止系统死锁。

6.5.2 多处理机的分类

多处理机系统在系统结构上可分为紧耦合和松耦合系统。

1. 紧耦合多处理机系统

紧耦合(Tightly Coupled)多处理机系统是通过共享主存实现处理机之间的相互通信的。在这种系统中,通常处理机的数目是有限的。这是因为主要受两个方面的约束:一是因采用共享主存进行通信,所以当处理机数目增多时,将导致访问主存资源冲突的概率增大,使系统效率下降。二是处理机与主存之间互连网络的带宽有限,当处理机数目增多后,互连网络将成为系统性能的瓶颈。

为了改善访问主存所发生的冲突,常采用如下一些方法:

① 采用多模块交叉访问的主存系统。交叉度越大,访问主存冲突概率就越小,但必须注意如何恰当地将数据分配到各个存储模块中去。

② 使每个处理机拥有一个小容量的局部存储器,用来存放频繁使用的核心代码等,以减少对主存访问次数。

③ 使每个处理机都有一个 Cache,以减少对主存访问。但必须注意 Cache 与主存之间以及各个 Cache 之间的数据的一致性。

图 6 .36 示出了一个典型的紧耦合多处理机系统。系统由 m 个存储模块, n 个处理机和 d 个 I/ O 通道组成, 由三个互连网络 PMIN(处理机—主存)、PPIN(处理机—处理机)和 PIOIN(处理机—I/ O 通道)将它们连接起来。

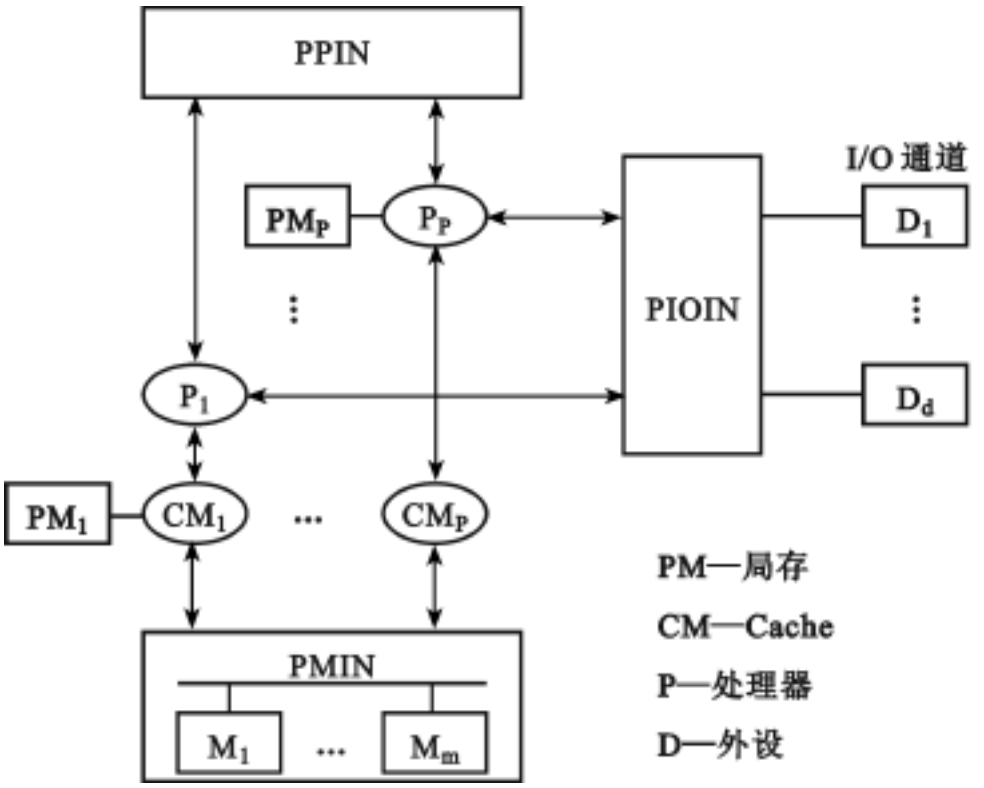


图 6 .36 紧耦合多机系统的典型结构

紧耦合系统按所用处理机类型是否相同及对称, 又可分为同构或异构以及对称或非对称形式, 常见的组合是同构对称型多处理机系统及异构非对称型多处理机系统。

(1)同构对称型多处理机系统

Sequent 公司生产的 Balance 多处理机系统就是同构对称型的, 它的结构如图 6 .37所示。CPU 模块数可在 2～32 之间选择, 存储器模块可在 1～6 之间选择。每个 CPU 模块由 80386 微处理器、Weitek 1167 浮点运算器以及 64K Cache 组成, 每个 MEM 模块由 8MB(可扩充到 40MB)及一个存储器控制器组成。上述 CPU 模块和 MEM 模块均与系统总线相连。系统总线还通过总线适配器与 Ethernet, SCSI 相连, 或是通过磁盘控制器与磁盘相连。此外, 它还借助总线 Multibus 和总线适配器与远程网相连。

我国由国家智能中心研制的“曙光一号”是一个由 4 个 CPU 组成的同构对称型多处理机系统, CPU 采用 Motorola 公司生产的 MC88100 及两个 Cache MC88200 芯片组成, 标准主存容量为 64MB(可扩展到 768MB), 用高速局部总线将 4 个 CPU 及主存连接起来。此外, 还通过总线适配器和接口与远程网(X .25)、局网(Ethernet)以及 SCSI 接口相连。

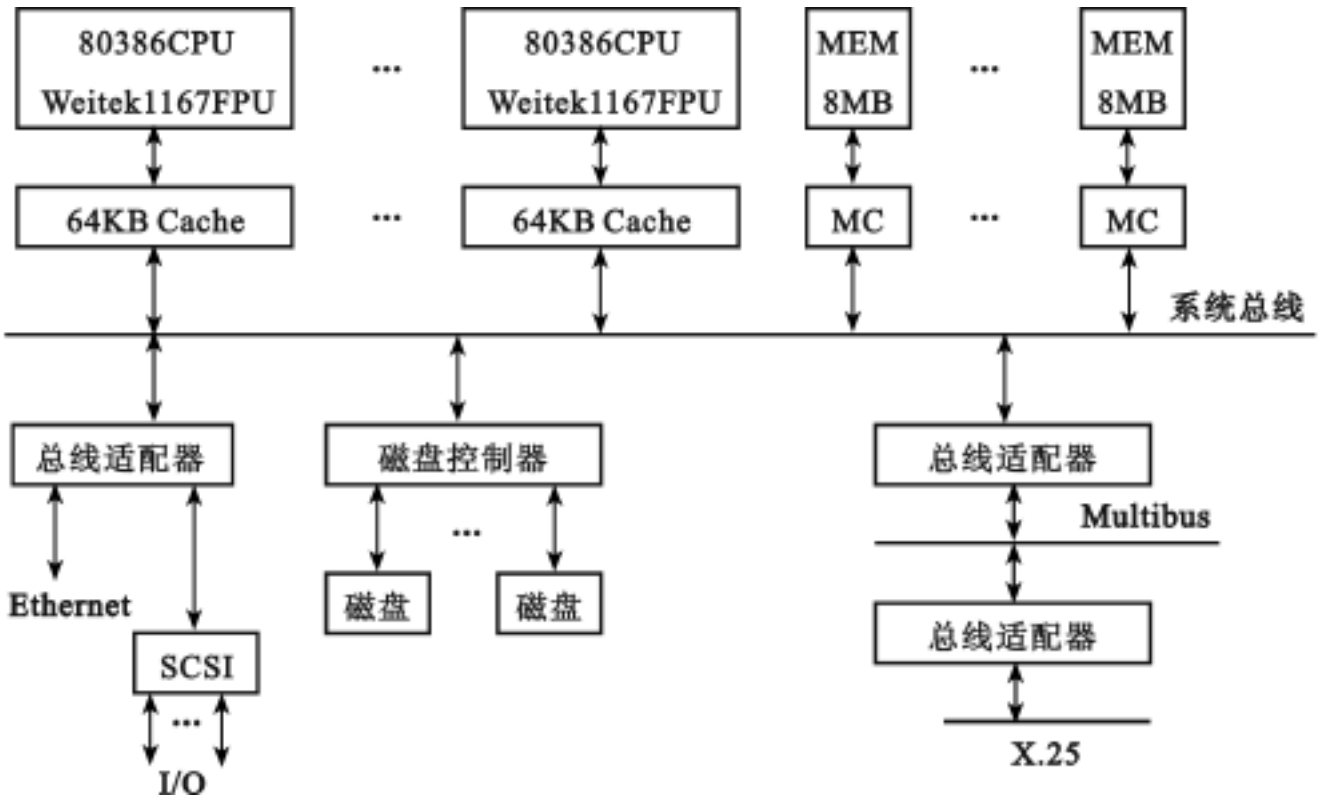


图 6 .37　Balance 同构对称型多处理机系统

(2)异构非对称型多处理机系统

图 6 .38 中示出了这种异构非对称型多处理机系统。其中主 CPU 中所用的微处理机可不同于从机中的微处理机。从机中 CIOP 处理机与字符外部设备相连,BIOP 与数组外设相连,NIOP 及 GIOP 分别为网络及图形处理机,ACOP 为向量加速处理机。

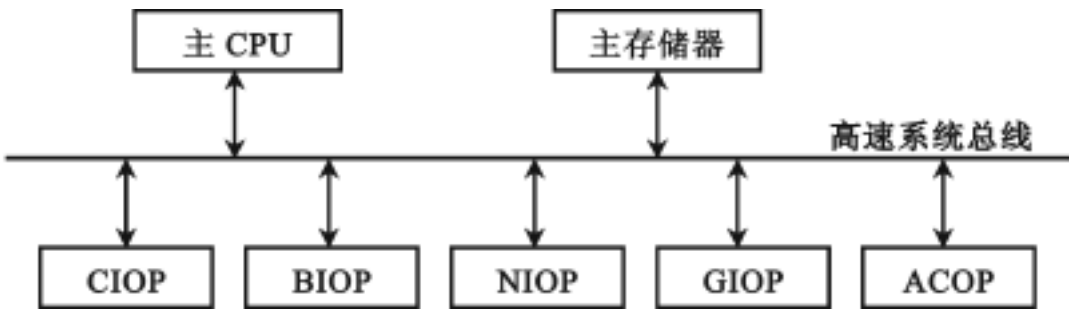


图 6 .38　异构非对称型多处理机系统

2. 松耦合多处理机系统

松耦合(Loosely Coupled)多处理机系统,它是通过消息传递方式来实现处理机间的相互通信,而每个处理机是由一个独立性较强的计算机模块组成的,该模块由处理器、局部存储器、I/O 设备以及与消息传递系统相连接的接口所组成。图 6 .39 示出了松耦合多处理机系统的框图。由于每个计算机模块中的局部存储器容量较大,

故在运算时所需的绝大部分指令和数据取自局部存储器。当不同模块上运行的进程间需要通信时,可通过节点接口及消息传递系统进行消息交换。由于这种相互间的耦合程度是很松散的,因此称为松耦合多处理机系统。在有的松耦合多处理机系统中,计算机模块本身就是一个完全独立的计算机,因此有时也将松耦合多处理机系统称为松耦合多计算机系统。

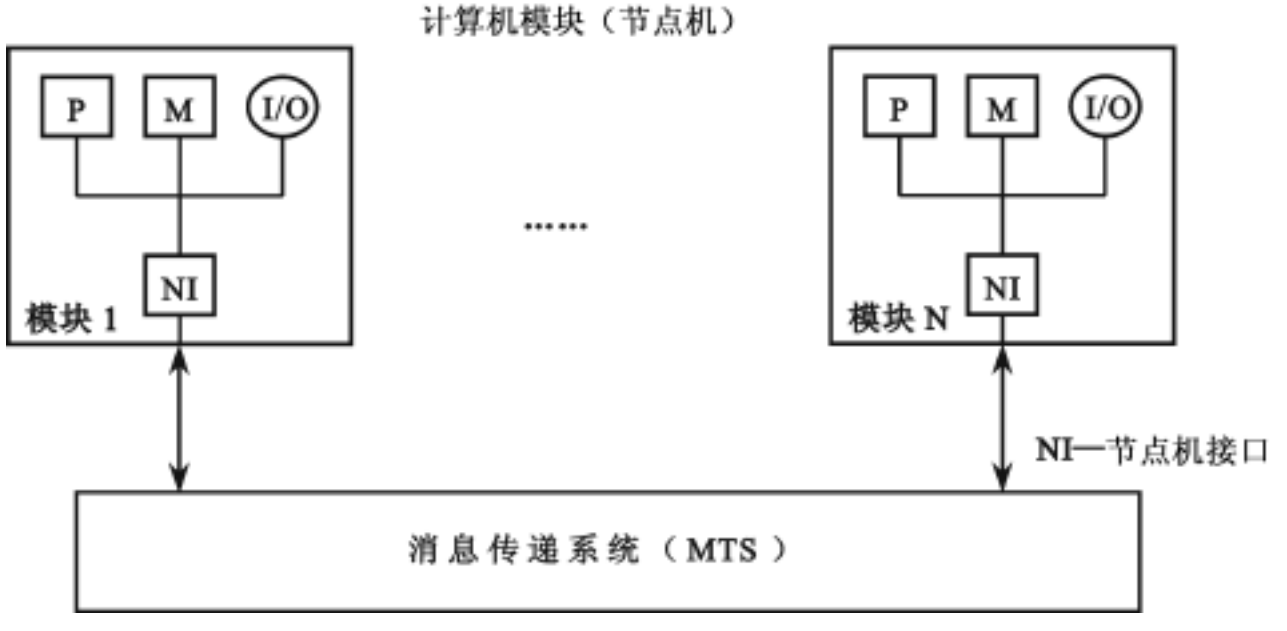


图 6 .39 松耦合多处理机系统

松耦合多处理机系统可分为非层次式和层次式两种结构。图 6 .39 中的多机系统结构即是非层次式的。其中的网络接口(NI-Network interface)通常是由通道和仲裁开关(Channel and Arbiter Switch,CAS)组成,用来对有两个或多个计算机模块同时请求访问消息传递系统(Message Transfer System,MTS)时,进行仲裁。

6.5.3 多处理机间的互连方式

多处理机系统中,对机间互连方式的要求比 SIMD 计算机要高一些,具体表现在以下三个方面:一是互连模式要灵活多样,以适应机间通信模式的多样性;二是应能适应机间通信的不规则性,实现无冲突连接;三是要求互连网络具有高带宽和低成本。

多处理机系统中常用的互连模式有以下几种:

1. 总线互连方式

总线互连方式是多机系统中实现互连的最简单的一种结构形式。多个处理机、存储器模块和 I/O 部件通过各自的接口部件与一条公用系统总线相连。欲传送的信息以分时或多路转换方式在系统中的主控器及宿控器之间传送。在多处理机系统总线中,潜在的总线主控器主要是处理机或计算机模块,宿控器可以是处理机或计算

机模块,也可以是存储器模块或 I/O 部件。

总线互连方式的优点是简单、价廉和模块可扩展性灵活。主要缺点是:可靠性较差,一旦总线失效将导致整个系统失效;带宽较窄,因此可连接到总线的部件、模块和处理机数有限,过多的总线负载将导致主控器争用总线的加剧和信息传送时间的加长,从而使整个系统性能下降。

多处理机系统总线的工作步骤为:首先,由各个潜在主控器争用总线使用权;其次,被选中的主控器及宿控器相连;最后,信息按一定规约通过总线在一对主控和宿控器间进行传送,必要时也可由一个主控器以广播方式向多个宿控器发送信息。

与单处理机总线一样,用总线带宽来衡量多处理机总线性能的好坏。为增加带宽,通常采用的方法是:用优质电缆组成总线,以提高总线速率;增加总线数,如采用双总线或多总线结构、系统总线加局部总线结构和层次式总线结构。

在总线互连方式中,为了解决多个潜在主控器同时争用总线问题,需由系统总线仲裁机构——总线仲裁器进行裁决,以确定哪个请求源可以使用系统总线。仲裁器一般由某种仲裁算法和相应的硬件构成。仲裁算法的好坏对系统性能有较大影响。

常用的仲裁算法有:

(1)静态优先级算法

它为每个连到总线上的处理机(或计算机模块)分配一个惟一的固定优先级。当多个处理机同时请求使用系统总线时,仲裁器使优先级最高的申请者使用总线。通常用串行链接方式确定优先级,因而越靠近仲裁器的处理机,它的优先级就越高。这种算法的优点是算法简单,容易实现。缺点是优先级低的处理机将很少有机会使用总线。

(2)平等算法

通常以轮转方式将总线按固定大小的时间片依次供各处理机使用。该算法的优点是算法较简单且能保证各处理机有均等机会使用总线,缺点是平均等待时间较长。此外,若所轮到的处理机不用总线时,将造成总线带宽的浪费。

(3)动态优先级算法

这是一种根据总线使用情况和相应规则,能动态地改变连接到总线上的多处理机的优先级。例如,近期最少使用 LRU 算法,它将最高的优先级分配给在最长时间间隔内未使用总线的处理机。循环菊花链(Rotating Daisy Chain, RDC)算法,则根据离最后一次使用总线的处理机所处位置的远近分配优先级。它将总线、准用线按某一方向接成闭环,刚使用总线的处理机的优先级最低,而离它越近的处理机的优先级越高。该算法的优点是兼顾了前两种算法的优点,即有较短的平均等待时间;并可使系统中的各处理机有更均等的机会使用总线。缺点是控制逻辑较为复杂。

(4)先来先服务算法

这是一种理想算法。它不是按优先级选择主控器,因而具有最好的均等性,该算法是性能最好的仲裁算法,但实现困难。该算法的作用是提供一种标准以衡量其他

算法的好坏。

上述的各种仲裁算法,通常用集中方式实现,即统一由一个仲裁器实现仲裁算法。此时,对于总线的请求和允许使用信号可采用如下三种实现方式:一是请求线共享,而允许使用线则用串行菊花链方式连接。二是使请求和允许使用线都采用分离独立的线。三是采用前两种混合方式。

上述各种仲裁算法也可采用分布方式实现。此时的仲裁硬件被分布到各个处理机中。它的工作方式如下:每个潜在总线主控器分配一个惟一的优先号,提出申请的主控器将自己的优先号送往共享请求/ 有效线进行逻辑或操作,得到一个合成优先号,然后提出申请的各个主控器将自己的优先号与合成号相比较。比合成优先号小的申请者将自动撤销申请,这样剩下获得总线使用权的必将是具有最高优先号的处理机。分布式仲裁算法的优点是有较高的可靠性。分布式仲裁器的工作过程如下:首先,通过请求线接收来自各处理机发来的使用总线请求;然后,由仲裁器加以仲裁并以串行链接或并行分离方式向选中的处理机在总线准用线上发出总线有效信号;最后,由选中处理机通过总线忙控制线向其他处理机表明总线已被占用。当主控处理机使用总线传送信息后便撤销总线忙信号,此后,仲裁器便可再去响应选择其他处理机对总线的请求。

多处理机总线的发展趋向是实行标准化,已有一些总线标准支持多处理机系统,常用的有 VME 总线和 MultibusⅡ总线等。前者以异步方式传送信息,后者则以同步方式传送信息。当用总线结构互连多处理机时,应尽量采用总线标准。如因需要必须采用专用总线时,也应通过总线适配器与标准总线相连,惟此,各种按总线标准设计的外围设备才能方便地与多处理机系统相连。

标准总线一般都带有仲裁器,如 VME 总线仲裁器采用集中式结构,而 MultibusⅡ和 Futurebus 总线仲裁器则采用分布式结构。这些标准总线一般都支持优先和均等混合仲裁算法,以适应多处理机系统需要。对外围部件使用优先仲裁算法,而对其他处理机则使用均等仲裁算法,以使各处理机有较均等机会使用系统总线。

总线互连方式虽有简单、价廉的优点,但其吞吐率是固定有限的。当有更多的处理机连到总线上时,就会使总线的工作速度大为降低。而当处理机数多到某一程度时,总线将会出现饱和状态,因此,总线互连方式不适宜连接过多的处理机。

2. 环形互连方式

为保持总线互连方式的优点,同时又能克服其不足,可以考虑构造一种逻辑总线,让各台处理机之间点点相连成环状,称环形互连,如图 6 .40 所示。在这种多处理机上,信息的传递过程是由发送进程将信息送到环上,经环形网络不断向下一台处理机传递,直到此信息又回到发送者处为止。

发送信息的处理机拥有一个惟一的令牌(Token),它是普通传送的信息中不会出现的特定标记。同时只能有一台处理机可持有这个令牌。发送者在发送信息时,

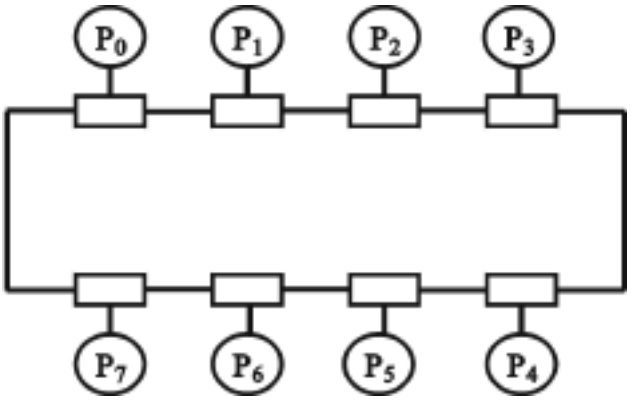


图 6.40　多处理机间采用环形互连方式

环上其他处理机都处于接收信息的状态。发送者一旦将信息发送完就向环上播送令牌。其他处理机依次传递此信息和令牌,并根据需要可接收信息。如果某台处理机想要发送信息,在收到令牌后不再将令牌传向下一台处理机,此时这台处理机就可以通过环形网络发送信息了。如果各处理机都不发送信息,令牌就在环形网络上不断循环传递,直至某台处理机需要发送信息为止。

由于环形互连是点点连接,不是总线式连接,其物理参数容易得到控制,非常适合于有高通信带宽的光纤通信。光纤通信是很难用在总线互连方式系统上的。环形互连的缺点是信息在每个接口处都会有一个单位的传输延迟,当互连的处理机数增加时,环中的信息传输延迟将增大。但与总线式互连不同的是,即使网上处理机机数很多,通信负荷很重,都不会出现像总线互连那样使系统带宽急剧下降的情况,系统的带宽仍保持一个高值。这是因为令牌环网可看成是一种周期短、延迟长的流水线。只要计算时能够保持流水线处于满负荷流动,让各处理机的计算和传输重叠进行,发送者在发送完全部信息后,不用等回收到此信息,就将其持有的令牌向下传递给新的发送者,有效带宽就可以得到最充分的利用。反之,若只有当原来的信息都不在环上传递时才能发送新的信息,那么就会与在二次不同操作之间需要排空流水线的做法类似,这样,随着环形网上处理机数的增加,网络的带宽会严重降低,造成系统效率急剧下降。

3. 交叉开关互连方式

交叉开关互连由一组二维阵列的开关组成,它将横向的 P 台处理机及纵向的模数为 m 的存储器连接起来。阵列中的总线数等于所有处理机数 P 与存储器模数 m 之和。只要 m≥P 就必然可以使每个处理机都能有一套总线与某一个存储器模块相连,从而可以大大加宽带宽。这是一种无阻塞的互连方式,与总线互连中采用时分制不同,它是采用空间分配机制。图 6.41 中示出了这种纵横交叉开关互连方式,图 6.41 中每个交叉点都是一套开关,除了有多路转换逻辑外,为了处理多个处理机同时访问某一存储器模块所发生的冲突,交叉开关互连方式需要有相应的仲裁部件。

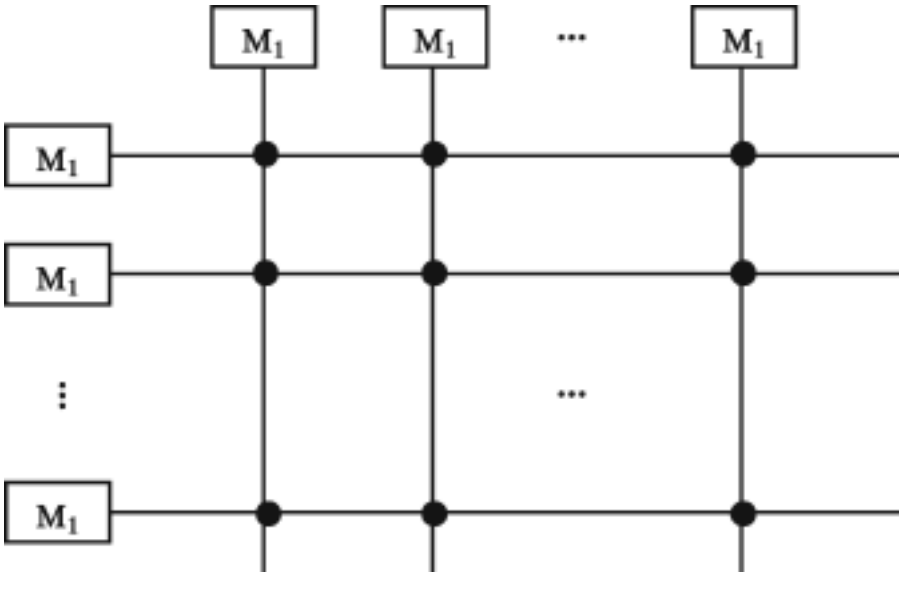


图 6 .41 多处理机的交叉开关互连方式

显然,这种互连开关阵列比较复杂,当 m,P 两者都为 n 时,阵列所需设备量将为 $O(n^2)$,受集成电路工艺水平限制,一般 n 不宜超过 16。随着 VLSI 技术的不断发展,这一上限正在不断提高。此外,对于分布存储的松耦合方式的大规模并行多处理机系统,可采用多级纵横交叉开关互连来实现众多处理机间的连接。

由于交叉开关较复杂,可通过用多个较小规模的交叉开关"串联"和"并联",构成多级交叉开关网络,以取代单级的大规模交叉开关。图 6 .42 是用 4×4 的交叉开关构成 16×16 的二级交叉开关网络,使设备量减少为单级 16×16 的一半。这实际是用 4×4 的交叉开关模块构成 $4^2 \times 4^2$ 的交叉开关网络,其中,指数 2 为互连网络的级数。

在多处理机中,经常会遇到互连网络的输入端个数和输出端个数不同的情况。此时可用 a×b 的交叉开关模块,使 a 中任一输入端与 b 中任一输出端相连。用 n 级 a×b 交叉开关模块可组成一个 $a^n \times b^n$ 的开关网络,它已被 Patel(1981 年)的多处理机采用,叫做 Delta 网。因此,SIMD 的互连网络同样也可用于多处理机间的互连。例如,多处理机用混洗交换网络互连的通信带宽和成本介于单总线互连与交叉开关互连之间,而且比较适用于处理机数较多的多处理机中。由于多处理机的通信模式不规则,因此,能实现 N ! 种排列的全排列网络同样适用于多处理机的机间互连。前面已提到,不同于 SIMD 的是开关中带有小容量缓冲存储器。

4. 多端口存储器互连方式

这种互连方式要求每个存储器模块有多个存取端口,每个端口均有控制、转接和仲裁功能,这实际上是一种将纵横交叉开关互连方式由开关阵列集中控制分散到各个端口进行分布控制。由于存储器模块的端口数不可能做得很多,因此它只适于系统中处理机数不太多的场合。

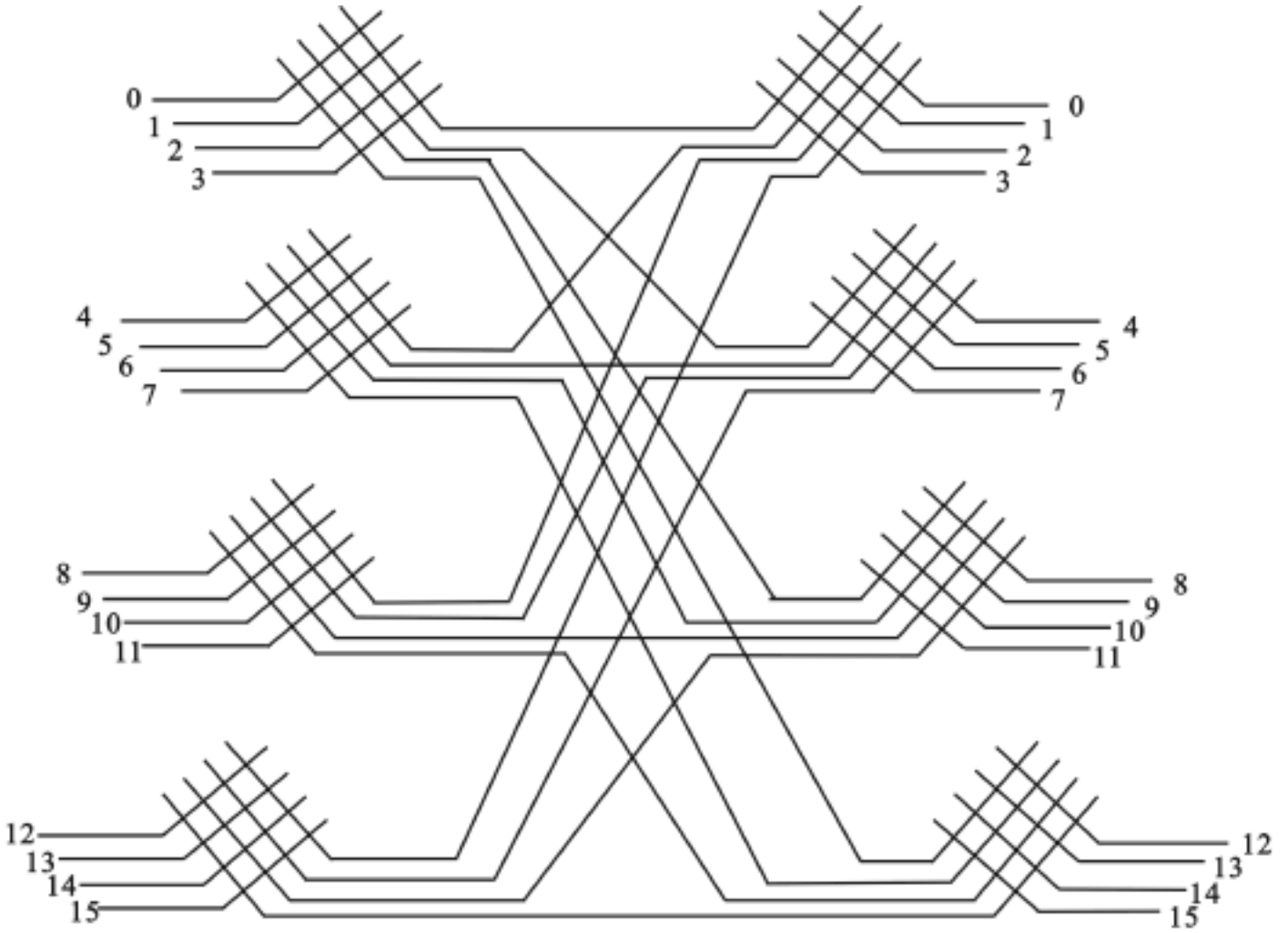


图 6.42　用 8 个 4×4 交叉开关模块构成 16×16 的二级交叉开关网络

6.5.4　多处理机系统中并行性开发

由于多处理机系统中的并行性存在于不同的层次上,因此,要充分开发它的并行性就必须从算法、语言、编译、操作系统和系统结构等各方面来统筹、协调地进行。

1. 并行算法分析

当处理机数目很多时,要把问题分解成由足够多的处理机处理的并行过程是极其困难的。为简化讨论,以算术表达式的并行运算为例来说明并行算法的研究思路。实际上,可把表达式看成是多个程序段相互作用的结果,而表达式中每一项都可看成是一个程序段的运行结果。

算法必须适应具体的计算机结构。在串行机上经常采用的循环和迭代算法往往不适合于多处理机系统,而采用直接求解法有时反倒能揭示更多的并行性。例如,表达式

$$E_1 = a + bx + cx^2 + dx^3$$

利用霍纳(Horner)法可得到

$$E_1 = a + x(b + x(c + x(d)))$$

这是在单处理机上执行的典型算法,共需 3 个乘加循环 6 级运算。但不适合于在多处理机上运行,因为它无法利用上其他的处理机。将这两式的运算过程表示为树形流程图分别如图 6.43(a)和图 6.43(b)所示。

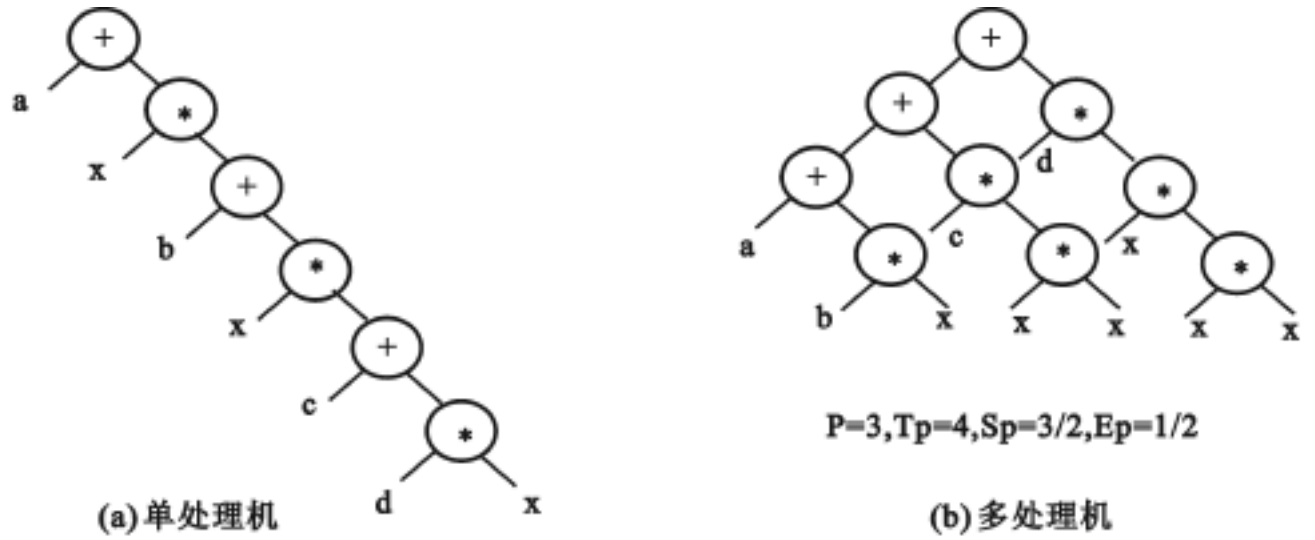


图 6.43 对于同一运算的两种表达式的树形流程图

在图 6.43 中,P 代表处理机的数目;Tp 代表运算级数,即为树的高度;Sp 为加速比,它指顺序运算的级数 T_1 与 P 台处理机运算的级数 Tp 的比值;而 Sp/P = Ep 称为效率。可见,Sp≥1 时,会使 Ep≤1,即运算的加速总是伴随着效率的下降。

既然可把运算过程表示成树形结构,那么,提高运算的并行性就是如何对树进行变换,减少运算的级数,即降低树高。树形结构可以用交换律、结合律、分配律来变换。因此,对描述运算过程的树形结构进行变换,可以作为研究并行算法的一种思路。

首先从算术表达式的最直接形式出发,利用交换律把相同的运算集中在一起。然后利用结合律把参加这些运算的操作数(称原子)配对,尽可能并行运算,从而组成树高最小的子树。最后再把这些子树结合起来。例如,表达式

$$E_2 = a + b(c + def + g) + h$$

需 7 级运算,如图 6.44(a)所示。利用交换律和结合律改写为

$$E_2 = (a + h) + b((c + g) + def)$$

则只需 5 级运算,如图 6.44(b)所示。

利用分配律进一步降低树高,在恰当平衡各子树的级数的情况下往往能收到较好的效果。在上式中的计算表达式中,如果改写成

$$E_2 = (a + h) + (bc + bg) + bdef$$

运算过程的树形结构如图 6.45 所示。

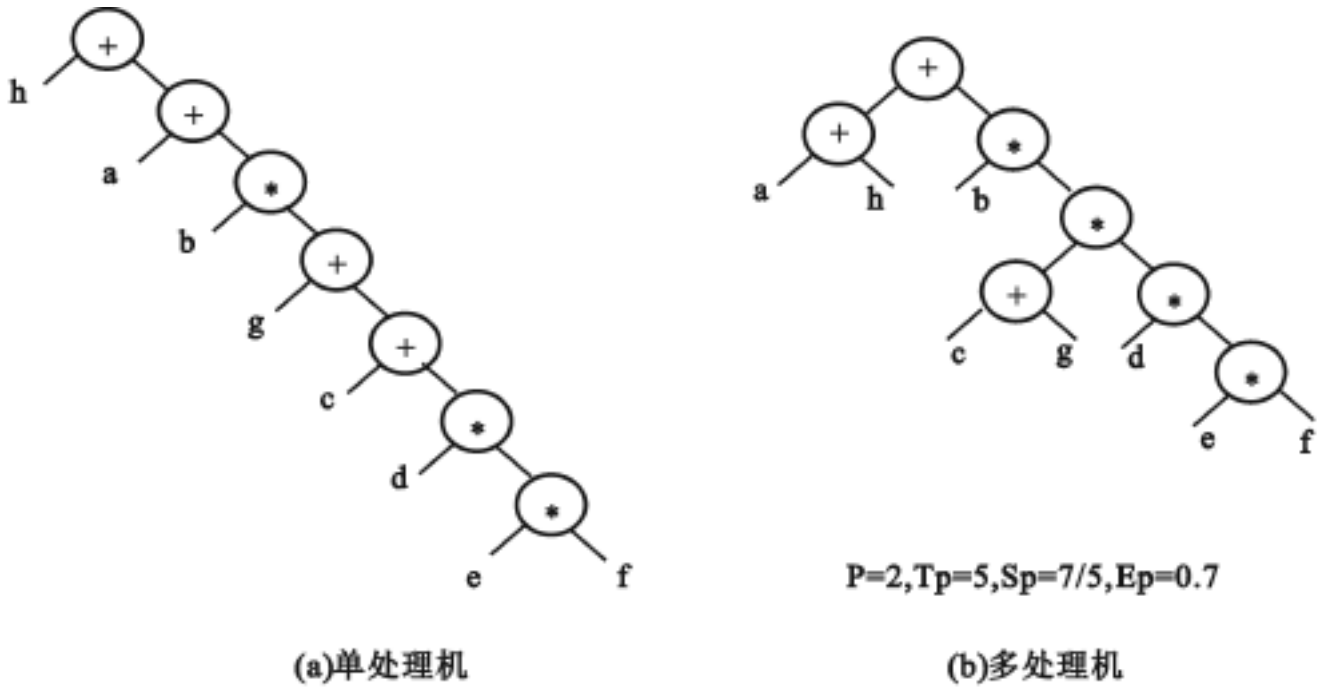


图 6 .44　利用交换律和结合律降低运算树高

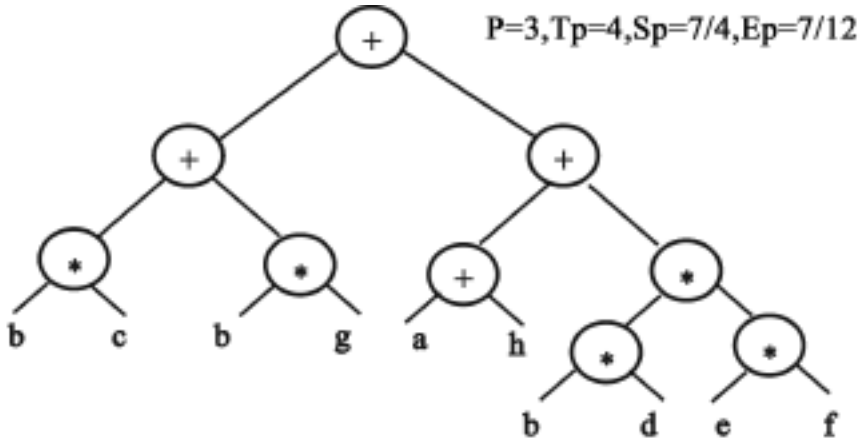


图 6 .45　利用交换律、结合律和分配律降低树高

2. 程序的并行性分析

任务间能否并行,除了算法外,在很大程度上还取决于程序的结构。程序中各类数据相关,是限制程序并行的重要因素。数据相关既可存在于指令之间,也可存在于程序段之间。下面就从程序段之间的数据相关来分析程序的并行性问题。

假定一个程序包含 P_1 , P_2 , …, P_i , …, P_j , …, P_n 等 n 个程序段,其书写的顺序反映了该程序正常执行的顺序。为便于分析,设 P_i 和 P_j 程序段都是一条语句, P_i 在 P_j 之前执行,且只讨论 P_i 和 P_j 之间数据的直接相关关系。

在第五章中曾讲过异步流动,指令之间数据相关可能有先写后读、先读后写和写—写3 种。在多处理机上,各处理机的程序段并行必然是异步的,因此,程序段之

间也必然会出现类似的 3 种数据相关。为简单起见,仅以赋值语句表示程序段为例。

(1)数据相关

如果 P_i 的左部变量在 P_j 的右部变量集内,且 P_j 必须取出 P_i 运算的结果来作为操作数,就称 P_j 数据相关于 P_i。例如:

P_i : A = B + D

P_j : C = A * E

相当于流水中发生的先写后读相关。显然,P_i 和 P_j 不能并行执行。

(2)数据反相关

如果 P_j 的左部变量在 P_i 的右部变量集内,且当 P_i 未取用其变量的值之前,是不允许被 P_j 所改变的,就称 P_i 数据反相关于 P_j。例如:

P_i : C = A + E

P_j : A = B + D

相当于流水中发生的先读后写相关。为了保证语义上的正确性,必须等 P_i 将变量 A 读出以后,P_j 才可向变量 A 进行写操作,即必须先读后写。

对于这种相关,如果能够在硬件上保证对相关单元先读后写的次序,是可以并行的,如图 6 .46 所示。即让每个处理机的操作结果先暂存于自己的局部存储器(或 Cache 存储器)中,不急于去修改原来存放于共享主存单元的内容。这样,只要控制局部存储器(或 Cache 存储器)向共享主存的写入同步进行即可。

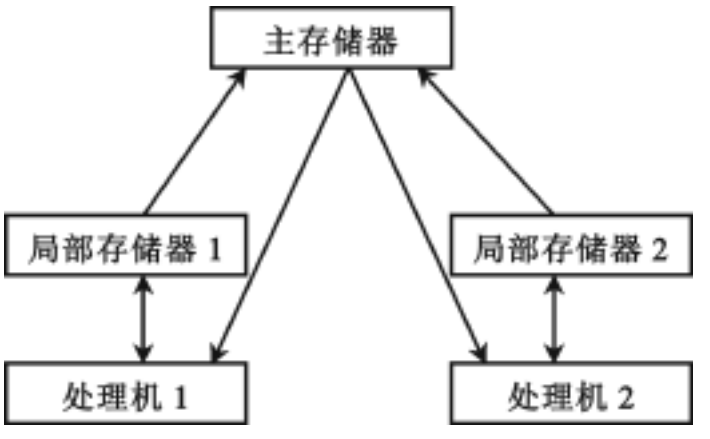


图 6 .46 能保证读—写次序的处理机结构

(3)数据输出相关

如果 P_i 的左部变量也是 P_j 的左部变量,且 P_j 存入其算得的值必须在 P_i 存入之后,则称 P_j 数据输出相关于 P_i。例如:

P_i : A = B + D

P_j : A = C + E

按原执行顺序 $A_{新}$ 应为 C + E。可见,只要同步能保证 P_i 先写入之后 P_j 再写入,这两个程序段可以并行。

除了上述 3 种相关外,如果两个程序段的输入变量互为输出变量,同时具有先写后读和先读后写两种相关,以交换数据为目的,则两者必须并行执行,既不能顺序串行,也不能交换串行。例如,两语句的左、右变量互相交换

P_i　　A = B

P_j　　B = A

必须并行执行,且需要保证读写完全同步。当然,如果在图 6 .46 的多处理机上运行这样的程序段,就能自动保证读/ 写次序,降低对同步的要求。

综上所述,两个程序段之间若有先写后读的数据相关,不能并行执行;若有先读后写的数据反相关,可以并行执行,但必须保证其写入共享主存时的先读后写次序;若有写—写的数据输出相关,可以并行执行,但同样须保证其写入的先后次序;若同时有先写后读和先读后写两种相关,以交换数据为目的时,必须并行执行,且读写要完全同步。

下面介绍由伯恩斯坦(Bernstein)提出的一种自动判别数据相关的准则。

每个程序段在执行过程中通常要使用输入和输出这两个分离的变量集。若用 I_i 表示 P_i 程序段中操作所要读取的存储单元集,用 O_i 表示要写入的存储单元集,则 P_1 和 P_2 两个程序段能并行执行的伯恩斯坦准则为:

① $I_1 \cap O_2 =$　　,即 P_1 的输入变量集与 P_2 的输出变量集不相交;

② $I_2 \cap O_1 =$　　,即 P_2 的输入变量集与 P_1 的输出变量集不相交;

③ $O_1 \cap O_2 =$　　,即 P_1 和 P_2 的输出变量集不相交。

例如,若有三个程序段 P_1 , P_2 和 P_3 分别求三个矩阵算术表达式:

$P_1 : X = (A + B) \times (A - B)$

$P_2 : Y = (C - 1) \times (C + D) - 1$

$P_3 : Z = X + Y$

其中, A, B, C, D, X 和 Y 均为 N×N 的矩阵,它们的输入、输出变量集分别为:

$I_1 = \{A, B\}$, $I_2 = \{C, D\}$, $I_3 = \{X, Y\}$, $O_1 = \{X\}$, $O_2 = \{Y\}$, $O_3 = \{Z\}$

由于 $I_1 \cap O_2 =$　　, $I_2 \cap O_1 =$　　以及 $O_1 \cap O_2 =$　　,故 P_1 和 P_2 两个程序段可以并行执行;由于 $I_3 \cap O_1 \neq$　　和 $I_3 \cap O_2 \neq$　　,故 P_3 程序段必须在 P_1 和 P_2 程序段执行完毕方可开始执行。

3. 并行程序设计语言

并行算法需要用并行程序来实现。为了加强程序并行性的识别能力,有必要在程序语言中增加能明确表示并发进程的成分,这就要使用并行程序设计语言。并行程序设计语言的产生有两种方法:一是可以在普通顺序型语言上加以扩充,增加能明确表示并行进程的成分,但每一种经扩充的语言仅能支持一种类型的并行性;二是可以通过设计全新的并行程序设计语言来支持并行处理。一般采用前一种方法比较现实。

并行程序设计语言的基本要求是:能使程序员在其程序中灵活方便地表示出各类并行性,能在各种并行/ 向量计算机系统中高效地实现。

(1)使用 FORK—JOIN 语句来表示并发性

FORK—JOIN 语句在不同机器上有不同的表示形式。现以 M .E .Conway 提出的形式为例说明。

FORK 语句的形式为 FORK m,其中 m 为新进程开始的标号。执行 FORK m 语句时,派生出标号为 m 开始的新进程,当前的进程仍继续执行。具体功能包括:

① 准备好这个新进程启动和执行所必需的信息;

② 如果现有空闲的处理机则将其分配给已派生的新进程,否则,让它们排队等待;

③ 继续在原处理机上执行 FORK 语句以后的原进程。

与 FORK 语句相配合,作为每个并发进程的终端语句 JOIN。

JOIN 语句的形式为 JOIN n,其中 n 为已派生出的并发进程个数。JOIN 语句附有一个计数器,其初始值为 0。每当执行一个 JOIN n 语句时,计数器的值加 1,并与 n 比较。若比较相等,表明这是执行的第 n 个(最后一个)并发进程经过 JOIN n 语句,于是允许该进程通过 JOIN 语句,将计数器清 0,并在其处理机上继续执行后续语句。若比较计数器的值仍小于 n,表明此进程不是并发进程中的最后一个,可让现在执行 JOIN 语句的这个进程先结束,把它所占用的处理机释放出来,分配给正在排队等待的其他进程或让该处理机空闲。

下面仍以算术表达式 Z = E + A × B × C/ D + E 的计算为例,经并行编译得到如下程序:

S_1 : G = A × B

S_2 : H = C/ D

S_3 : I = G × H

S_4 : J = E + F

S_5 : Z = I + J

如果不加并行控制语句,这个程序仍然只是一个普通的串行程序,发挥不出多处理机的作用。图 6 .47 示出了各语句间的数据相关情况。它表明 S_1 和 S_2 可以同时开始执行,但要等到 S_1 和 S_2 都完成之后,才能开始执行 S_3 ,并可并行地开始执行 S_4 ,而只有 S_4 和 S_3 汇合才能执行 S_5 。

利用 FORK—JOIN 语句实现这种派生和汇合关系,将程序改写为:

```
      FORK 20        ;派生出一个新的进程 S2
10      G = A × B    ;当前的处理机继续执行 S1
      JOIN   2       ;结束进程 S1 ,并比较计数器值是否等于 2
      GOTO 30
20      H = C/ D     ;执行进程 S2
```

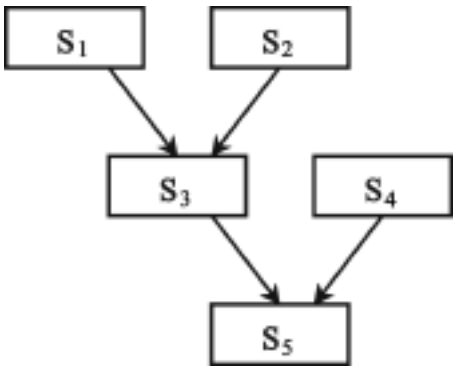


图 6.47　并行程序数据相关图

```
      JOIN   2
30      FORK 40    ;派生新进程 S4
      I = G×H      ;执行进程 S3
      JOIN 2
      GOTO 50
40      J = E + F  ;执行进程 S4
      JOIN 2
50      Z = I + J  ;执行进程 S5
```

执行这个程序可用两台处理机。假定最初的程序是在处理机 1 上运行的,遇到 FORK 20 语句时就分出一个处理机(假定是 2)去执行 S_2,而处理机 1 接着执行下面的 S_1。如果 S_1 执行时间较短,当它结束时遇到 JOIN 2 语句,S_2 尚在执行,处理机 1 从 S_1 释放,因无其他任务处于空闲。随后当 S_2 结束时,由于已与 S1 汇合,便可以通过 JOIN 语句,由处理机 2 继续执行后续的 FORK 40 语句。这条语句又派生出 S_4,分配给空闲的处理机 1,而处理机 2 接着执行 S_3。同样,等 S_4 和 S_3 都先后结束,才满足 JOIN 语句的汇合条件,经 GOTO 50 进入 S_5。其执行过程见图 6.48。

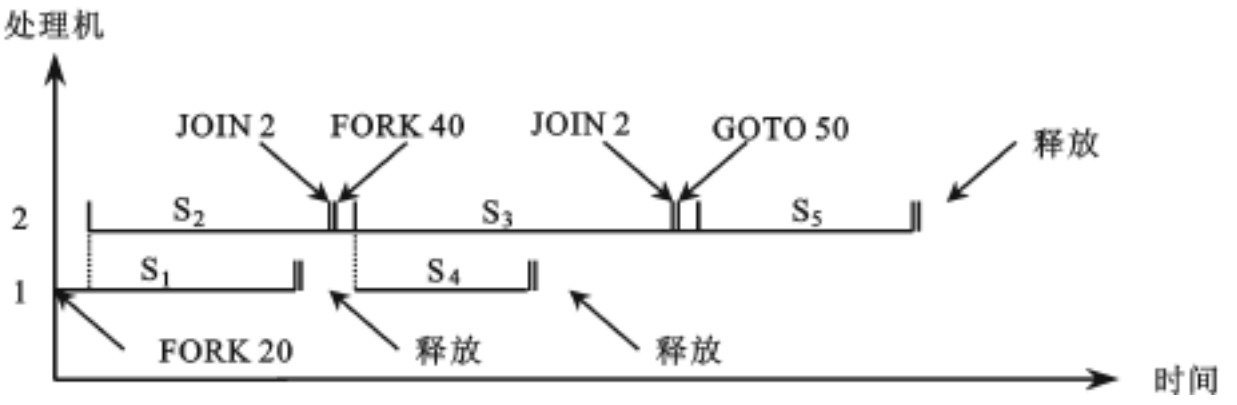


图 6.48　并行程序在多处理机上执行的时间—空间(资源)示意图

(2)使用块结构语言来表示并行性

块结构语言与 FORK—JOIN 语句在概念上是等价的,但使用更为方便。它用 cobegin—coend(或 parbegin—parend)将所有可并行执行的语句或进程前后括起来,以显式表明它们可并行执行。例如:

```
begin
    S0 ;
    cobegin S1 ;S2 ;…;coend
    Sn+1 ;
end
```

其中,S_1,S_2,…,S_n 为可并行执行的语句。而语句 S_{n+1} 则必须在 S_1,S_2 …,S_n 全部执行完后,方可开始执行。每一组的并行语句只有一个入口和出口,因此符合结构化程序设计要求。此外,并行语句中所定义的各个进程互相独立,具有不相交性,即每个语句所能修改的变量只属于该进程专有的局部变量。虽然不相交进程间可使用共享变量,但只能读,不可修改。并行语句还可任意嵌套使用。图 6.49 中示出了嵌套执行情况。例如:

```
begin
   S0 ;
   cobegin
      S1 ;
      begin
         S2 ;
         cobegin S3 ;S4 ;S5 ; coend
         S6 ;
      end
      S7 ;
   coend
   S8 ;
end
```

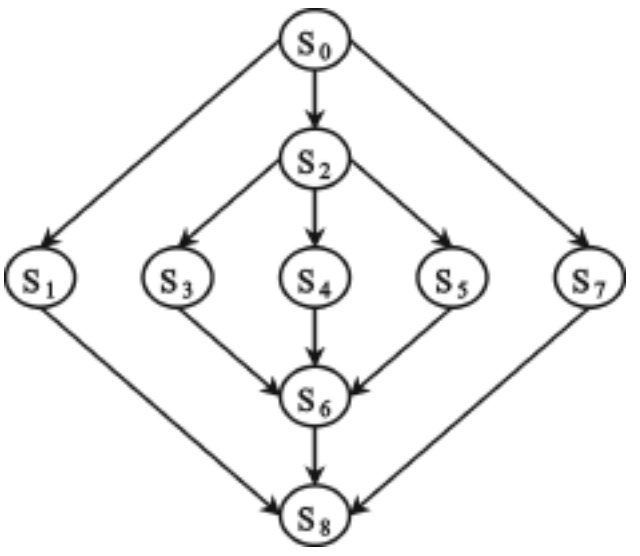


图 6.49 嵌套并行进程优先图

在循环程序中通常可找到可并行执行的语句,此时可用并行循环 parfor 语句来表示这种并行性。例如,若要进行矩阵运算 C = A×B,其中 A 为一个 n×n 的矩阵,而 B 和 C 则均为 n×l 的列向量。下面列出使用 parfor 语句的求解算法,它能派生出 p 个独立进程。假定 p 能整除 n,则有 s = n/ p:

```
parfor i = 1 until p do
begin
   for j = (i - 1)s + 1 until s×i do
      begin
         c(j) = 0;
```

```
        for k = 1 until n do
        c(j) = c(j) + A(j,k)×B(k);
    end
end
```

每个被派生的进程,根据不同的 i 值计算最外层 begin - end 之间的语句。为方便说明起见,不妨假设 n = 12, p = 4,则执行 parfor 语句将派生出四个独立进程,分别对应于 i = 1, 2, 3, 4。此后,这四个进程可并行执行最外层 begin-end 之间的语句。对i = 1的独立进程,将得到最终计算结果:C(1), C(2)和 C(3)。对 i = 2 的独立进程,将得到最终计算结果:C(4), C(5)和 C(6)。而对 i = 3 和 i = 4 的两个独立进程,则分别得到最终计算结果:C(7)～C(9)和 C(10)～C(12)。

6.5.5 多处理机操作系统

1. 多处理机操作系统的特点

从概念上讲,多处理机操作系统与单处理机的多道程序操作系统无多大差别。但当多台处理机同时工作时,操作系统的复杂性就增加了。当要求其支持异步任务并发执行时,复杂性问题就显得更加突出了。

多处理机操作系统除了要完成通常单机操作系统的资源分配和管理、存储管理和保护、死锁防止、异常进程终止或例外处理等功能外,还必须具有如下的特点和功能:

① 支持多个任务的并行执行,为此需要研究任务的分解和分派以及处理机间负载平衡问题。

② 支持处理机间同步和通信管理。

③ 提供系统结构重构能力以支持系统降级使用。

④ 自动支持硬件并行性和程序并行性的开发。

2. 多处理机操作系统

多处理机操作系统有主从型(Master—Slave Supervisor)、各自独立型(Separate Supervisor)及浮动型(Floating Supervisor)三类。

(1)主从型操作系统

主从型操作系统中的管理程序只在一台指定的处理机(主处理机)上运行。该主处理机可以是专门的执行管理功能的控制处理机,也可以是与其他从处理机相同的通用机,除执行管理功能外也能做其他方面的应用。由于主处理机是负责管理系统中所有其他处理机(从处理机)的状态及其工作的分配,只把从处理机看成是一个可调度的资源,实现对整个系统的集中控制,因此,也称为集中控制或专门控制方式。从处理机是通过访管指令或自陷软中断来请求主处理机服务。这种主从型控制方式

的系统硬件结构比较简单;整个管理程序只在一台处理机上运行,除非某些需递归调用或多重调用的公用程序,一般都不必是可再入的;只有一个处理机访问执行表,不存在系统管理控制表格的访问冲突和阻塞,简化了管理控制的实现。所有这些均使这种操作系统能最大限度地利用已有的单处理机多道程序分时操作系统的成果,只需要对它稍加扩充即可。因此,实现起来简单、经济、方便,是目前大多数多处理机操作系统所采用的方式。

然而,主从型也有许多不足之处,对主处理机的可靠性要求很高,一旦发生故障,很容易使整个系统瘫痪,这时必须要由操作员干预才行。如果主处理机不是设计成专用的,操作员可用其他处理机作为新的主处理机来重新启动系统。整个系统显得不够灵活,同时要求主处理机必须能快速执行其管理功能,提前等待请求,以便及时为从处理机分配任务,否则将使从处理机因长时间空闲而显著降低了系统的效率。即使主处理机是专门的控制处理机,如果负荷过重,也会影响整个系统的性能。特别是当大部分任务都很短时,由于频繁地要求主处理机完成大量的管理性操作,系统效率将会显著降低。

主从型操作系统适合于工作负荷固定,且从处理机能力明显低于主处理机,或由功能相差很大的处理机组成的异构型多处理机。

(2)各自独立型操作系统

与主从型操作系统不同,各自独立型操作系统将控制功能分散给多台处理机,共同完成对整个系统的控制工作。每台处理机都有一个独立的管理程序(操作系统的内核)在运行,即每台处理机都有一个内核的副本按自身的需要及分配给它的程序需要来执行各种管理功能。由于多台处理机执行管理程序,要求管理程序必须是可再入的,或对每台处理机提供专门的管理程序副本。这种方式的优点是很适应分布处理的模块化结构特点,减少对大型控制专用处理机的需求;某个处理机发生故障,不会引起整个系统瘫痪,有较高的可靠性;每台处理机都有其专用控制表格,使访问系统表格的冲突较少,也不会有许多公用的执行表,同时控制进程和用户进程一起进行调度,能取得较高的系统效率。

然而,这种方式实现复杂。尽管每台处理机有自己的专用控制表格,但仍有一些共享表格会增加共享表格的访问冲突,导致进程调度的复杂性和开销加大。某台处理机一旦发生故障,要想恢复和重新执行未完成的工作较困难。每台处理机都有自己专用的输入输出设备和文件,使整个系统的输入输出结构变换需要操作员干预。各处理机负荷的平衡比较困难。

各台处理机需要局部存储器存放管理程序副本,降低了存储器的利用率。

各自独立型操作系统适合于松耦合多处理机。

(3)浮动型操作系统

浮动型操作系统是界于主从型操作系统和各自独立型操作系统之间的一种折衷方式,其管理程序可以在处理机之间浮动。在一段较长的时间里指定某一台处理机

为控制处理机,但是具体指定哪一台处理机以及担任多长时间控制都是不固定的。主控制程序可以从一台处理机转移到另一台处理机,其他处理机中可以同时有多台处理机执行同一个管理服务子程序,因此,多数管理程序必须是可再入的。由于同一时间里可以有多台处理机处于管态,有可能发生访问表格和数据集的冲突,一般采用互斥访问方法解决。服务请求冲突可通过静态分配或动态控制高优先级方法解决。这种系统可以使各类资源做到较好的负荷平衡。一些像 I/O 中断等非专门的操作,可交由在某段时间最闲的处理机去执行。它在硬件结构和可靠性上具有分布控制的优点,而在操作系统的复杂性和经济性上则接近于主从型。如果操作系统设计得好,将不受处理机机数的多少所影响,因而具有很高的灵活性。然而,这种操作系统的设计是最困难的。

这种方式的操作系统适用于紧耦合多处理机,特别是由公用主存和 I/O 子系统的多个相同处理机组成的同构型多处理机。

6.5.6 多处理机的调度策略

在多处理机系统中,能否对资源进行有效分配是决定多处理机系统性能的关键,特别是当系统中处理机的数目很多的时候。

在多处理机系统中,资源分配要解决的主要问题是:一个进程应分配到哪个处理机上运行。同构型多处理机系统的调度策略是将各种资源,包括处理机、主存、I/O 通道等,存放在一个公共的资源池中,为所有进程共享,以求提高资源利用率和尽量使各处理机的负载平衡。对异构型多处理机系统而言,由于每个处理机所需完成的功能不同,因此调度工作主要是按处理机的功能,如控制、数据处理、输入/输出等,分配任务,使各个处理机各司其职。

衡量多处理机调度性能好坏的主要参数通常有:

完成所有任务所需的最少时间;完成所有任务所需的最少处理机数;最小平均流时间,即执行每个任务所需的平均时间;处理机的最大利用率或处理机的最小空闲时间。

图 6.50 示出了一个任务调度时空图,表示在三个处理机 P_1, P_2, P_3 上执行五个任务 $T_1 \sim T_5$ 的时空图。各个任务的执行时间分别为 7,6.5,2,2,2。完成所有任务所需总的时间为 19.5。平均流时间为 19.5/5 = 3.9。P_1, P_2, P_3 的利用率分别为 0.93,1.00 和 0.86。它们的空闲时间分别为 0.5,0.0 和 1.0。

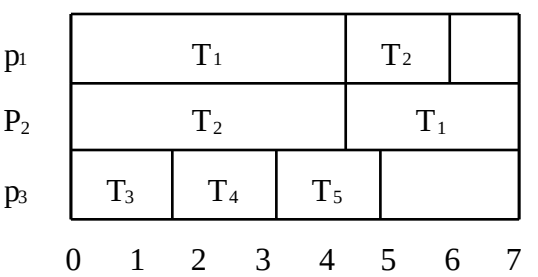


图 6.50 任务调度时空图

多处理机调度实质上就是要寻找一种调度算法,其调度目标是使用最少量的处理机在最短时间内完成全部任务。调度算法通常要依据一个模型进行,目前主要流行的调度模型有两种:静态的确定性模型和动态的随机性模型。

下面以静态的确定性调度模型为例说明。

这种调度模型要求在求解问题前就已知执行每个任务所需的时间以及系统中各任务间的关系。这种调度算法的设计较简单,但如果事先不能准确估计每个任务的执行时间及任务间关系,该调度算法的效率就不高。在确定性调度模型中,可以根据不同的调度环境采用不同的算法。为简单起见,我们只讨论由一组相同的处理机和具有确定优先关系的一组任务所组成的调度环境,并采用可抢占调度。所谓抢占调度是指一个任务在被完成之前,允许被中断,此后又可恢复执行的调度。只要这种抢占不过于频繁,那么它的效率就优于非抢占调度。

通常把一个任务集合称为任务系统,可用一个优先图来表示。图中节点表示具有独立操作且需要与其他节点联系的任务。节点集合表示了任务集合 $T=\{T_1,\cdots,T_n\}$,节点间的有向边表示了任务间存在的偏序或优先关系。任务调度必须符合以下的原则:

① 遵循偏序关系。

② 任何时候只能将一台处理机分配给一个任务。

此外,每个节点还有称为权重的另一个属性,它指明由该节点所表示的任务,在处理机上运行时所需的时间。

假定一个任务图包含节点权重为 $t_1,t_2,\cdots,t_n$ 的 n 个独立任务和 p 台处理机,则最优调度所需的最小时间为:

$$W=\max\{\max_{1\leq i\leq n}\{t_i\},(\sum_{i=1}^{n}t_i)/p\}$$

即最小执行时间不可能小于最长任务的执行时间(括号中第一项)或处理机的平均执行时间(括号中的第二项)。

为了实现优化调度,首先,要把任务图 G 中的所有节点的权值单位化成具有相同权值的图 GW,如图 6.51(a)所示。图 6.51(b)中的图 GW,权值已单位化成 W=3.5。然后,将图 GW 中的全部节点分解成若干个不相交的子集系列,使每个子集中的节点彼此独立,这样处于同一子集或同一级中的所有节点便可同时执行。对于如图 6.52(a)中的图 GW,它的最佳子集序列应为$\{T_1\},\{T_2,T_3\},\{T_5,T_6,T_7\},\{T_4,T_8\},\{T_9,T_{10}\}$和$\{T_{11}\}$。

图 6.52(a)的各级是这样划分的,结束节点(图中为 T_{11})为第一级,在此之前的单位时间内,可以执行的那些节点为第二级(图中为 T_9 和 T_{10}),依此类推直至图中的入口节点为止。

当处理机的数目为 2 时,若每一级中的任务数为偶数,则该级中所有任务都能充分地利用两台处理机,在最短时间内完成所有的任务。若任务数为奇数时,则最后三

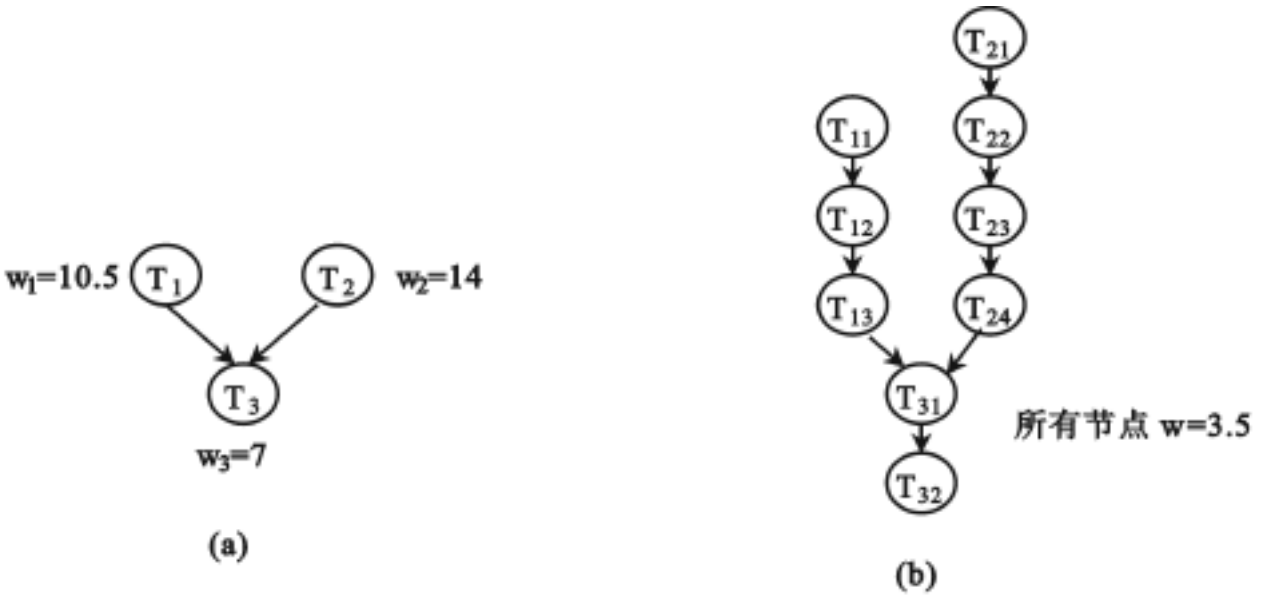


图 6 51　权值成比例的节点与权单位化的节点图

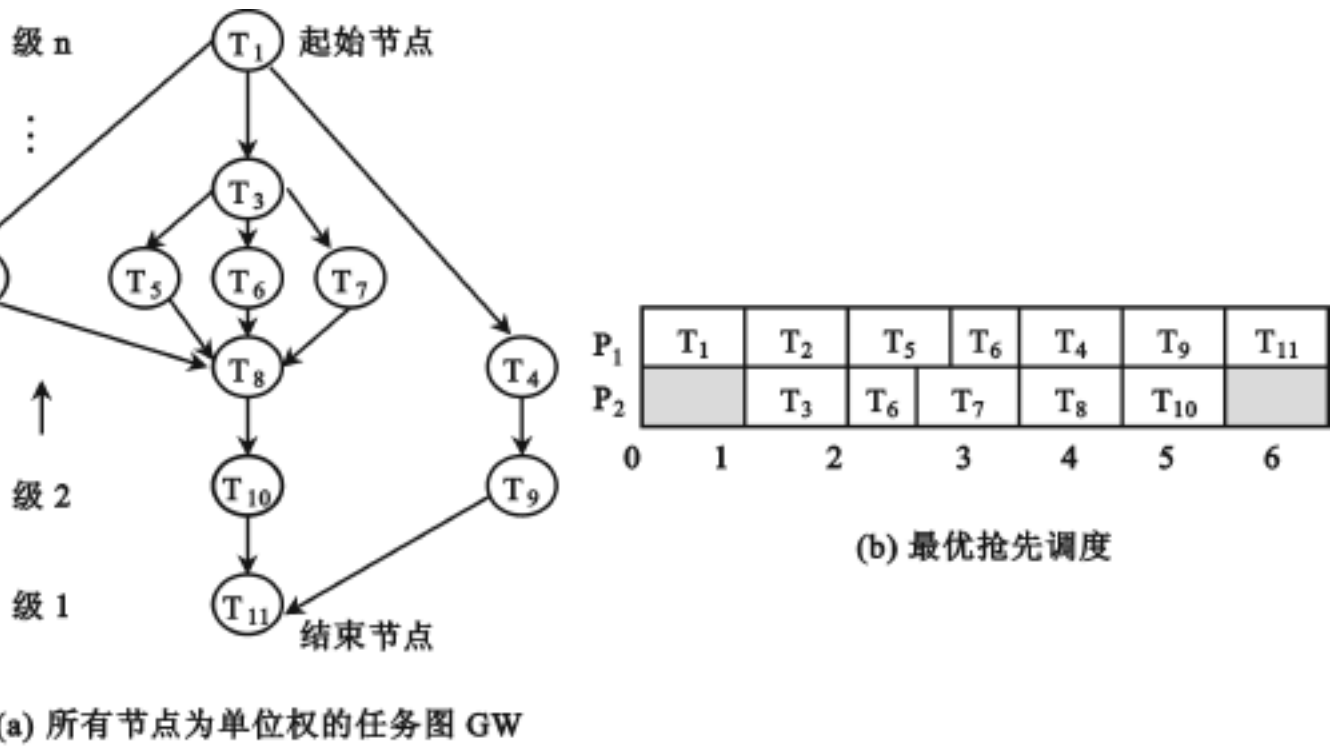


图 6 52　按独立子集顺序进行抢占的优先调度

个任务的完成时间将大于或等于 1 .5 单位时间。如图 6 .52(b) 中所示那样, T_5 , T_6 和 T_7 三个任务只需用 1 .5 单位时间便可完成, T_6 任务在 P_2 中未被完成前为 T_7 任务中断抢先, T_6 然后被迁移到 P_1 中去完成。整个的任务调度是如下进行的: 开始时因只有 T_1 任务无前趋, 因此将它调度到 P_1 处理机上去完成, 单位时间后 T_1 被完成, 此时 T_2 , T_3 和 T_4 三个任务均无前趋, 可并行执行, 但因只有两台处理机, 所以只能选其中两个任务: T_2 和 T_3 (因为这两个任务处在同一子集)。在此之后, T_5 , T_6 和

T_7 以及原来的 T_4 均无前趋,由于前三个任务处在同一子集,因此优先调度执行。

采用上述的抢占调度,只需 1.5 个单位时间便可完成。由于 T_4 和 T_8 两个任务此时均无前趋,又处在同一子集,故调度它们并行执行。此后,调度 T_9 和 T_{10},最后调度 T_{11}。

上述的最优调度方法可推广到处理机数目为任意的系统。

习　　题

1. 画出 16 个处理器仿 ILLIAC-Ⅳ的模式进行互连的互连结构图,列出 PE_0 分别只经一步、二步和三步传送能将信息传送到的各处理器号。

2. 设 16 个处理器的编号分别为 0,1,…,15,用单级互连网络互连,若互连函数为:

① $E(x)_3$

② $Cube_3$

③ $PM2_{+3}$, $PM2_{-1}$

④ Shuffle, Shuffle(Shuffle)

⑤ Butterfly

时,第 9 号处理器与哪一个处理器相连?

3. 设 128 个处理器的编号分别为 0,1,2,…,127,当复合互连函数为 Shuffle($cube_0$($PM2_{-2}$))时,第 8 号处理器将与哪个处理器相连?

4. 在有 16 个处理器的混洗交换网络中,若要使第 0 号处理器与第 15 号处理器相连需要经过多少次混洗和交换?以连接图的形式表明其变化过程。

5. 在处理器 N=8 的 OMEGA 网络中,若要实现处理器 2 与所有处理器相连,应怎样设置网络中交叉开关单元的状态?

6. 在 OMEGA 网络中,要同时实现处理器 5 到处理器 0 和处理器 7 到处理器 1 的连接是否可行?若不行,请指出在哪个交换开关单元处发生冲突?现若要同时实现处理器 0 到处理器 5 和处理器 1 到处理器 7 的连接,请作出与上述相类似的判别?

7. 编号分别为 0,1,2,…,F 的 16 个处理器之间要求按下列配对通信:(B,1),(8,2),(7,D),(6,C),(E,4),(A,0),(9,3),(5,F)。试选择所用互连网络类型、控制方式,并画出该互连网络的拓扑结构和各级交换开关状态图。

提示:注意配对节点之间的二进制地址特征。

8. 画出编号为 0,1,…,F 共 16 个处理器之间实现多级立方体互连的互连网络,当采用级控制信号为 1100(从右至左分别控制第 0 级至第 3 级)时,第 9 号处理器连向哪个处理器?

9. 若令 8×8 矩阵 $A=(a_{ij})$ 以行为主存放在主存储器中,用什么样的单级互连网络可使 A 转换成转置矩阵 A^T?总共需要传送多少步?

提示:将每个元素以 6 位地址表示,左边 3 位表示行号,右边 3 位表示列号。

10. 在三级 STARAN 移数网络中,若要实现移 4(mod 8)的功能,试给出 0 级(A,B,C,D)的级控信号,1 级的(E,G)和(F,H)的两个部分级控信号和 2 级的(K,L)以及(I)和(J)的三个部分级控信号。

11. 若一个向量有 36 个向量元素,使用 36 个处理器,采用递归折叠法求它们的累加和,则需要多少个单位时间可完成求和?假定从一个处理器传送到任何另一个处理器只需一步,且传送时间与一次加法时间相同,都为一个单位时间。与顺序求和方法相比,加速比为多少?

12. 画出 0～7 号共 8 个处理器的三级混洗交换网络,在该图上标出实现将 6 号处理器数据播送给 0～4 号,同时将 3 号处理器数据播送给其余 3 个处理器时的各有关交换开关单元的控制状态。

13. 并行处理机有 16 个处理器要实现相当于先 4 组 4 元交换,然后是 2 组 8 元交换,再次是 1 组 16 元交换的交换函数功能,请写出此时各处理器之间所实现的互连函数的一般式;画出相应多级网络的拓扑结构图,标出各级交换开关的状态。

14. 具有 $N = 2n$ 个输入端的 OMEGA 网络,采用单元控制。

(1)N 个输入总共可有多少种不同的排列;

(2)该 OMEGA 网络通过一次可以实现的置换可有多少种是不同的;

(3)若 $N = 8$,计算出一次通过能实现的置换数占全部排列数的百分比。

15. 画出 $N = 8$ 的立方体全排列多级网络,标出采用单元控制,实现 0 = 3,1 = 7,2 = 4,3 = 0,4 = 2,5 = 6,6 = 1,7 = 5 的同时传送时的各交换开关状态;说明为什么不会发生阻塞?

16. 分别画出 4×9 的一级交叉开关以及用两级 2×3 的交叉开关组成的 4×9 的 Delta 网络,比较一下交叉开关设备量的多少。

17. 说明 4×4 交叉开关组成的两级 16×16 交叉开关网络虽节省了设备,但它是一个阻塞式网络。

18. 由霍纳法则给定的表达式如下:E = a(b + c(d + e(f + gh)))利用减少树高的办法来加速运算,要求:

(1)画出树形流程图;

(2)确定 T_P,P、S_p,E_P 诸值。

19. 求 A_1,A_2,…,A_8 的累加和,有如下程序:

S_1　$A_1 = A_1 + A_2$

S_2　$A_3 = A_3 + A_4$

S_3　$A_5 = A_5 + A_6$

S_4　$A_7 = A_7 + A_8$

S_5　$A_1 = A_1 + A_3$

S_6　$A_5 = A_5 + A_7$

S_7 $A_1 = A_1 + A_5$

(1)写出用 FORK,JOIN 语句表示其并行任务的派生和汇合关系的程序,以假想使此程序能在多处理机上运行。

(2)画出该程序在有 3 台处理机的系统上运行的时间关系示意图。

(3)画出该程序在有两台处理机的系统上运行的时间关系示意图。

20. 若有下述程序:

U = A + B

V = U/ B

W = A * U

X = W - V

Y = W * U

Z = X/ Y

试用 FORK,JOIN 语句改写成可在多处理机上并行执行的程序。假设现有两台处理机,且除法速度最慢,加、减法速度最快,请画出该程序运行时的资源时间图。

21. 如图 6 .53 所示,每个任务节点的执行需一个单位时间,要求用抢占优先调度方法画出使用两台处理机 P_1 和 P_2 进行并行处理的任务调度时空图。求出完成总任务需多少个单位时间,并求出 P_1 和 P_2 两者总的利用率。

22. 如图 6 .54 所示,若用三台处理机用抢占调度法来处理,则完成此任务图的最佳完成时间(即下限值)应为多少个单位时间。

提示:只需列出下限值。

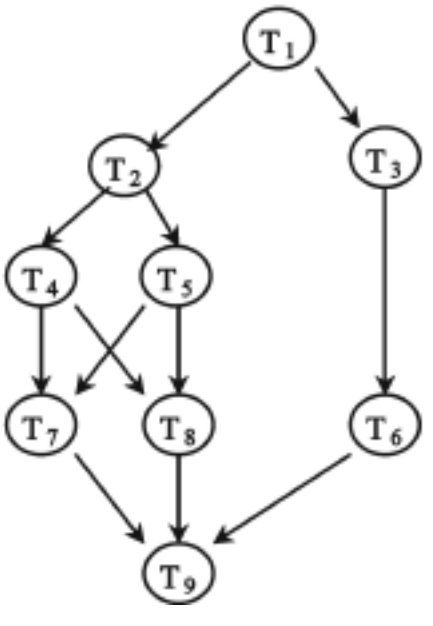


图 6 .53

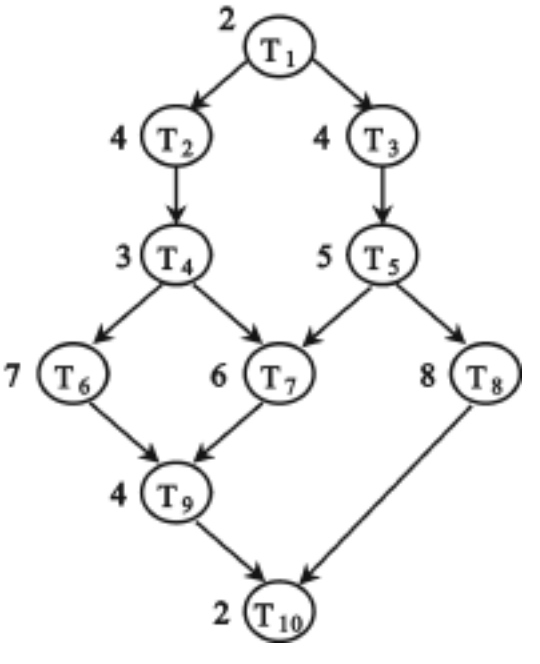


图 6 .54

23. 分别确定在下列各计算机系统中，计算点积 $S=\sum_{i=1}^{8} a_i * b_i$ 所需的时间(尽可能给出时空图示意)：

(1)通用 PE 的串行 SISD 系统；

(2)具有一个加法器和乘法器的多功能并行流水 SISD 系统；

(3)有 8 个处理器的 SIMD 系统；

(4)有 8 个处理机的 MIMD 系统。

设访存取指和取数的时间可以忽略不计；加与乘分别需要 2 拍和 4 拍；在 SIMD 和 MIMD 系统中处理器(机)之间每进行一次数据传送的时间为 1 拍，而在 SISD 的串行或流水系统中都可忽略；在 SIMD 系统中 PE 之间采用线性环形互连拓扑，即每个 PE 与其左右两个相邻的 PE 直接相连，而在 MIMD 中每个 PE 都可以和其他 PE 有直接的通路。

24. 试确定在下列 4 种计算机系统中，计算下列表达式所需时间。

$$S=\prod_{i=1}^{8}(A_i+B_i)$$

其中，加法需用 30ns，乘法需用 50ns。在 SIMD 和 MIMD 计算机中，数据由一个 PE(处理单元)传送到另一个 PE 需 10ns，而在 SISD 计算机中数据传送时间可忽略不计。在 SIMD 计算机中，PE 间以线性圆环方式互连(以单向方式传送数据)，而在 MIMD 计算机中，PE 间以全互连方式连接。

(1) 具有一个通用 PE 的 SISD 计算机系统；

(2) 具有一个加法器和一个乘法器的多功能部件的 SISD 计算机系统；

(3) 具有 8 个 PE 的 SIMD 计算机系统；

(4) 具有 8 个 PE 的 MIMD 计算机系统。

第七章 新型计算机结构

当今绝大多数的计算机都是属 Von Neumann(冯·诺依曼)类型的系统结构,它的基本特征是指令的顺序执行和数据共享,促使程序中指令执行的因素是事先由程序员编写好的指令执行顺序。20 世纪 70 年代中期出现的数据流计算机以指令执行所需的数据是否到达来驱动指令的执行,从而打破了冯·诺依曼计算机的体系结构。狭义地讲,新型计算机系统是指像数据流计算机这种完全不同于 Von Neumann 计算机工作方式的计算机。但有的学者将在体系结构上作了重大改进,而仍以 Von Neumann 计算机工作方式的计算机系统也称为新型计算机系统,显然这是广义的新型计算机系统。本章讨论几种具有代表性的计算机系统。

7.1 脉动阵列计算机

脉动阵列计算机(简称"脉动阵列机")是由一组相同的处理单元 PE 构成的阵列机。每个 PE 可完成少数基本的算术逻辑运算操作。这种阵列机的工作原理是:阵列内所有处理单元的数据锁存器都受同一个时钟控制,运算时数据在阵列结构的各个处理单元间沿各自的方向同步向前推进,就像血液受心脏有节奏地搏动在各条血管中同步向前流动一样。因此,形象地称其为脉动阵列结构。

脉动阵列机主要适用要求计算量很大的信号/图像的处理,以及某些特定计算类算法题目的求解,特别是需要对大量数据执行重复计算的运算受限类问题的求解。根据具体计算的问题不同,脉动阵列可以有一维线形、二维矩形/六边形/二叉树形/三角形等阵列互连构形。

图 7.1 给出了在一个脉动式二维阵列结构上进行两个 3×3 矩阵 A,B 相乘的例子。每个处理单元 PE 内含一个乘法器和一个加法器,可完成一个内积步运算。每经一拍,处理单元可把 3 个输入端送来的信息沿三个不同方向,即由左向右的水平方向、由下向上的垂直方向和由左下角到右上角的斜 45°方向,同时将结果传送到对应的 3 个输出端,使 $a'=a, b'=b, d=a\times b+c$。现设矩阵 A,B 和它们的乘积 C 分别为:

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad B=\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} \quad C=A\times B=\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

其中，$C_{ij}=\sum_{k=1}^{3} a_{ik}\times b_{kj}$，$1\leq i\leq 3, 1\leq j\leq 3$。在图中给出了 t_1，t_2，t_3 时刻送入阵列中的数据情况，到 t_6 时，从 45°方向上将同时输出 c_{13}，c_{12}，c_{11}，c_{21}，c_{31} 的值，t_7 时输出 c_{23}，c_{22}，c_{32} 的值，t_8 时输出 c_{33} 的值。可以看出，总共只需用 8 拍就可以完成两个 3×3 的矩阵相乘，比单处理机上循环执行所需的 27 拍，速度提高了两倍多。

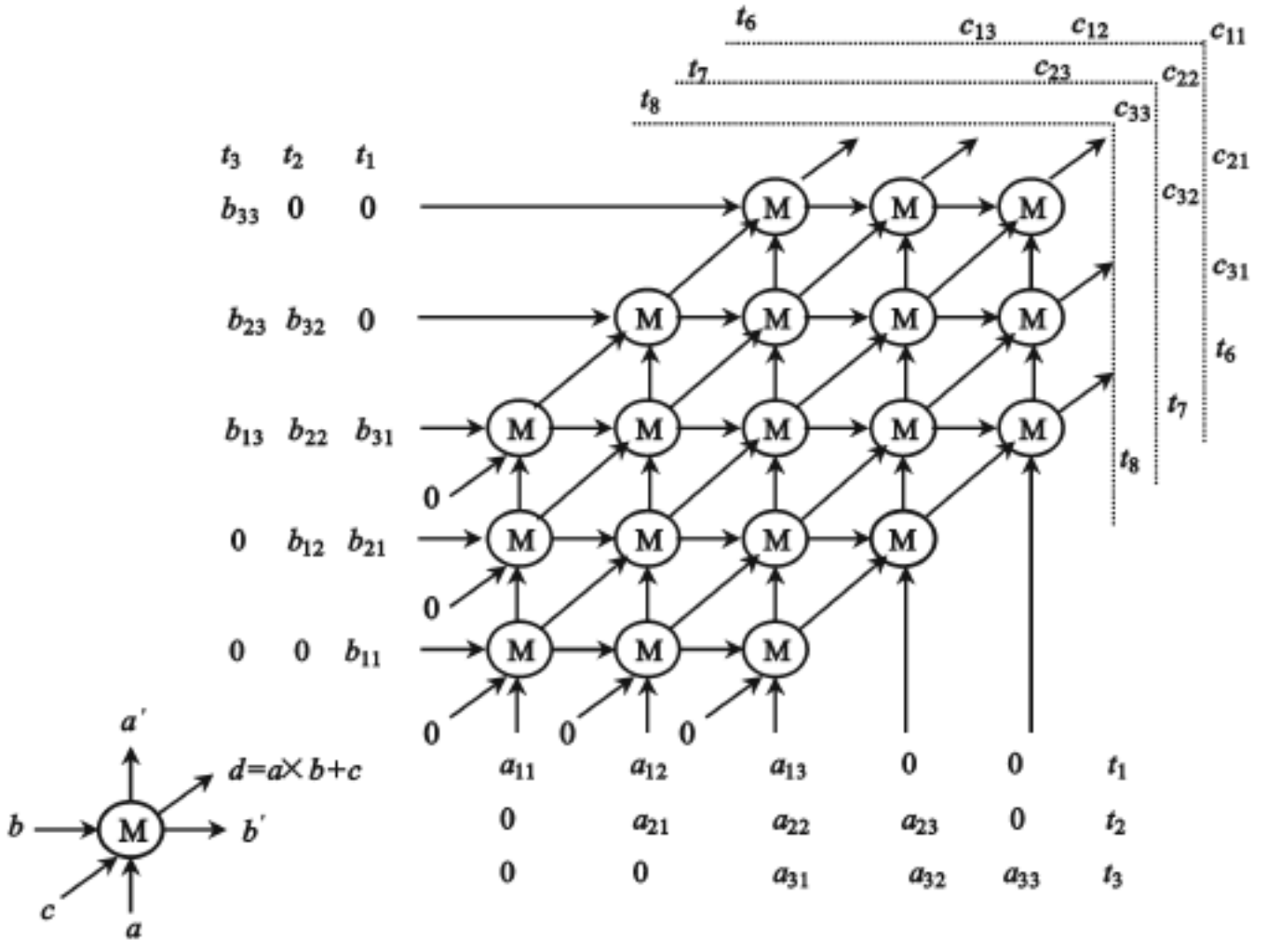


图 7.1　脉动式二维阵列流水机

7.2　数据流计算机

美国 MIT 实验室的 Jack Dennis 及其助手们于 1972 年首先提出了数据流(Data Flow)模型，并证明由此而设计的数据流计算机能够满足未来计算机理想结构的三个基本命题。这三个基本命题是：

① 要获得很高的性能价格比；

② 能跟上工艺技术进步的速度;

③ 在应用领域提供较好的可编程性。

数据流计算机的研究工作曾经被冷落过一段时间。近几年来,由于 VLSI 技术的迅速发展,为数据流计算机的发展提供了坚实的基础;因此,无论在硬件、软件,还是在理论研究方面,数据流计算机都有了一些较大的进步,并且已经出现了一批实用的数据流计算机系统。

本节首先介绍数据驱动(Data Driven)原理,着重分析数据流计算机与传统的 Von Neumann 控制驱动(Control Driven)计算机的不同;然后介绍数据流程序图和数据流语言,这是研究数据流计算机必须要了解的基础知识;最后介绍数据流计算机的典型结构,分析它们的系统结构和主要工作原理。

7.2.1 数据流计算机的基本工作原理

与传统的 Von Neumann 计算机不同,在数据流计算机中,一条指令能否执行的主要依据是:它所需要的操作数是否已经全部到达。因此,指令的执行不需要 PC 程序计数器。一条指令执行后产生的结果并不送往存储器保存起来,以供其他指令共享,而是直接流向所有需要该结果的指令,并作为这些指令的操作数,驱动这些指令的执行。因此程序执行时所需的数据是一次性生成和消费的,即时产生即时使用,操作数直接以令牌(Token)或值的记号传递而不是作为地址变量加以访问的。显然在数据流计算机中,程序中众多指令的执行是异步并行进行的,只要所需的操作数均已到达,又有可使用的计算资源,它们便可同时执行,因此最有利于计算并行性的开发。这与 Von Neumann 计算机的顺序执行、数据共享的特点是完全不同的。

在数据流计算机中,信息项以操作包和数据令牌形式出现。操作包由操作码、操作数和其后继指令所在地组成。数据“令牌”则由结果值和其去向目的地组成。众多的操作数和数据令牌在各个资源部件间传递,因而数据流计算机可视为是一种信息分组通信系统结构,具有分布式多处理机组成形式。

7.2.2 数据流程序图和数据流语言

通常用数据流程序图来表示指令级的数据流程序,它是一种特殊的有向图(Directed Graph),由多个节点和一些连接它们的弧所组成。该有向图说明了指令之间执行顺序的约束条件。数据流程序图中的节点除表示一般的算术逻辑操作外,还可表示常数产生、复制操作、判定操作和控制操作等。常用的节点种类有:

(1)算逻运算节点:

根据节点执行的操作功能可分为算术运算节点和布尔运算节点两种。常见的算术运算节点有加、减、乘、除、加 1、减 1 等,常见的布尔运算节点有与、或、异或、非等。

如果按照输入端的个数可以把节点分为单输入节点和多输入节点两种。如图 7.2 中的加 1 和非是单输入节点,而加和与是多输入节点。

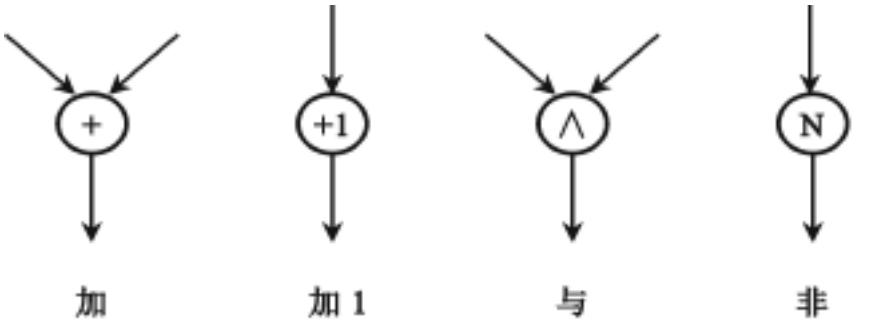


图 7 .2　算逻运算节点举例

(2)常数产生节点

常数产生节点没有输入端,只有一条输出线。这种节点的功能是用来产生一个常数。图 7 .3(a)是常数产生节点的一般示意图,图 7 .3(b)是常数 2 的节点及其执行操作后产生数据令牌的情况,该数据令牌携带有常数 2。

(3)复制操作节点

复制操作节点可以是数据也可以是控制量的多个复制。如图 7 3(c)、图 7 3(d)所示。对于数据复制节点,图中圆点和箭头用实心表示,而在控制量(布尔量)复制节点中的圆点和箭头用空心表示。

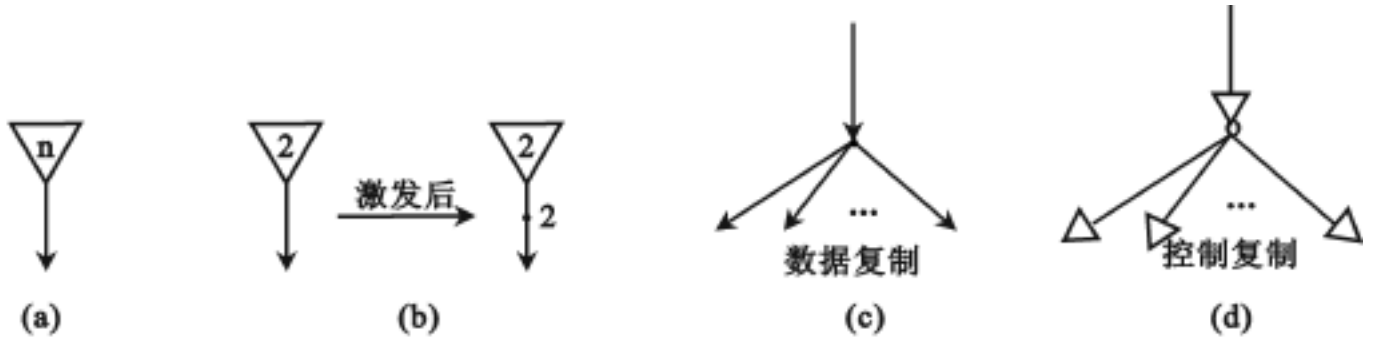


图 7 3　常数产生节点和复制节点

(4)控制操作节点

这类节点的特点是,其激发(或称点火)条件中要加上布尔控制端。常用的条件操作节点有如下四种:

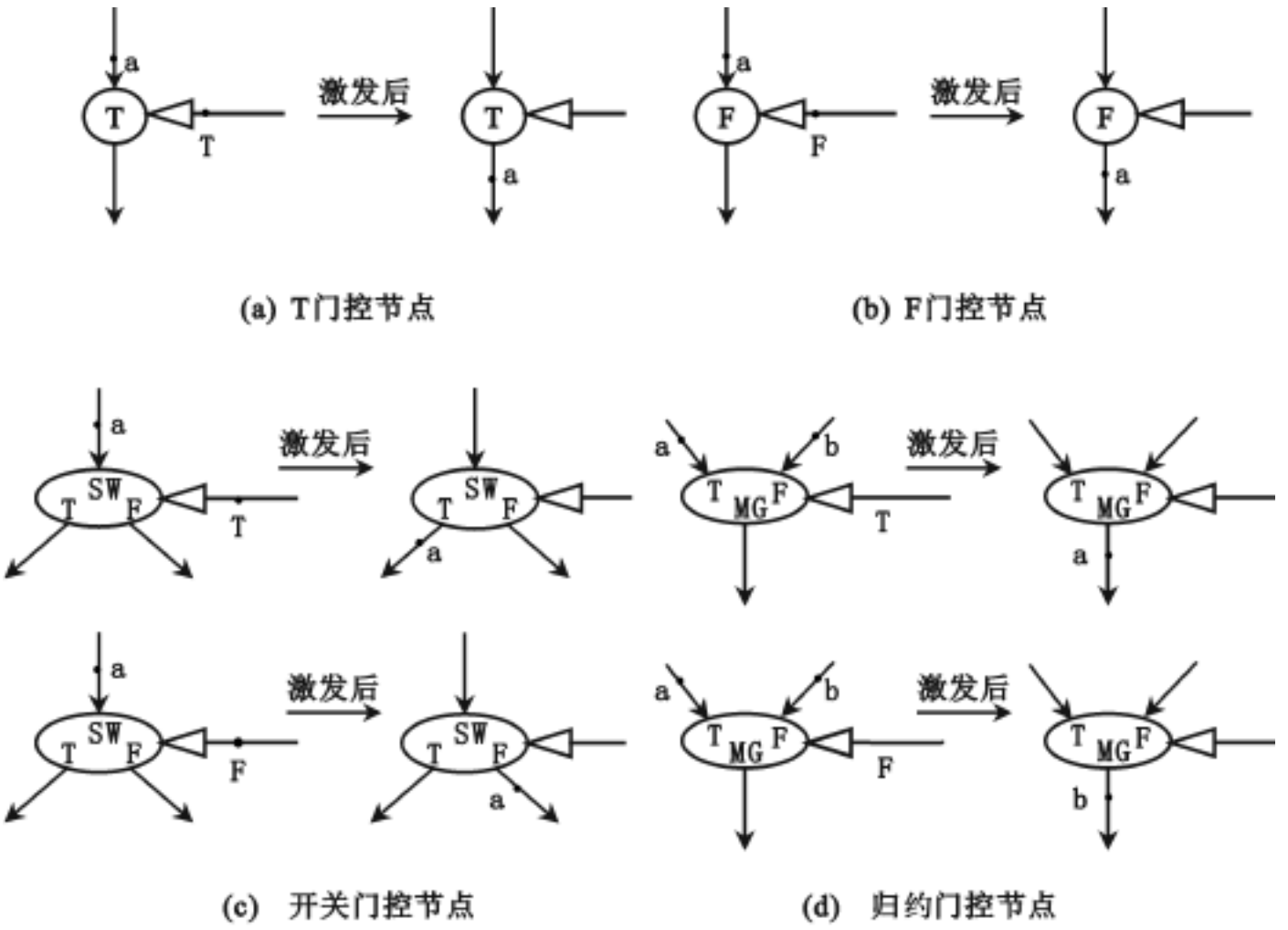
① T 门控节点:仅当布尔控制端为真,且输入端有数据令牌时才能激发,然后在输出端产生数据令牌而输入端的数据令牌消失,如图 7 .4(a)中所示。

② F 门控节点:与 T 门控节点类似,但仅当布尔控制端为假时,才能激发,如图 7 .4(b)所示。

③ 开关门控节点(SW 节点):有一个数据输入端和两个数据输出端,根据控制端值的真假确定 T 输出端或 F 输出端上带有输入端的数据令牌,如图 7 .4(c)所示。

④ 归约门控节点(MG 节点):有两个数据输入端和一个数据输出端,并受控制

端控制。激发后,根据控制端值真假在输出端上产生来自 T 输入端或 F 输入端上的数据令牌,如图 7.4(d)所示。



(a) T门控节点　　(b) F门控节点

(c) 开关门控节点　　(d) 归约门控节点

图 7.4 常用控制操作节点图及其激发规则

(5)条件判断节点

判断输入数据(通常是单个或两个)是否满足某种条件,如输入数据是否小于、等于、大于 0,两个输入数据的大小比较等。当满足条件时,将在输出端产生 T 的控制令牌,否则便产生 F 的控制令牌。数据流程序图中在空心箭头弧上流动的就是这类控制信号。图 7.5 中示出了单输入和双输入数据的判断操作工作情况。

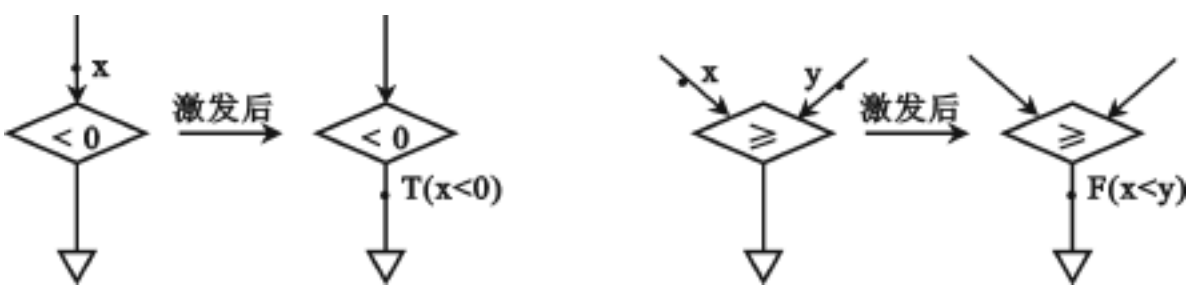


图 7.5 条件判断操作节点及其激发规则

利用上述常用节点,可以画出一些程序结构的数据流程图。例如,图 7 .6(a)是具有条件分支结构的数据流程图;图 7 .6(b)是具有循环结构的数据程序图;图 7 .7是计算 N 阶乘的数据流程序图。

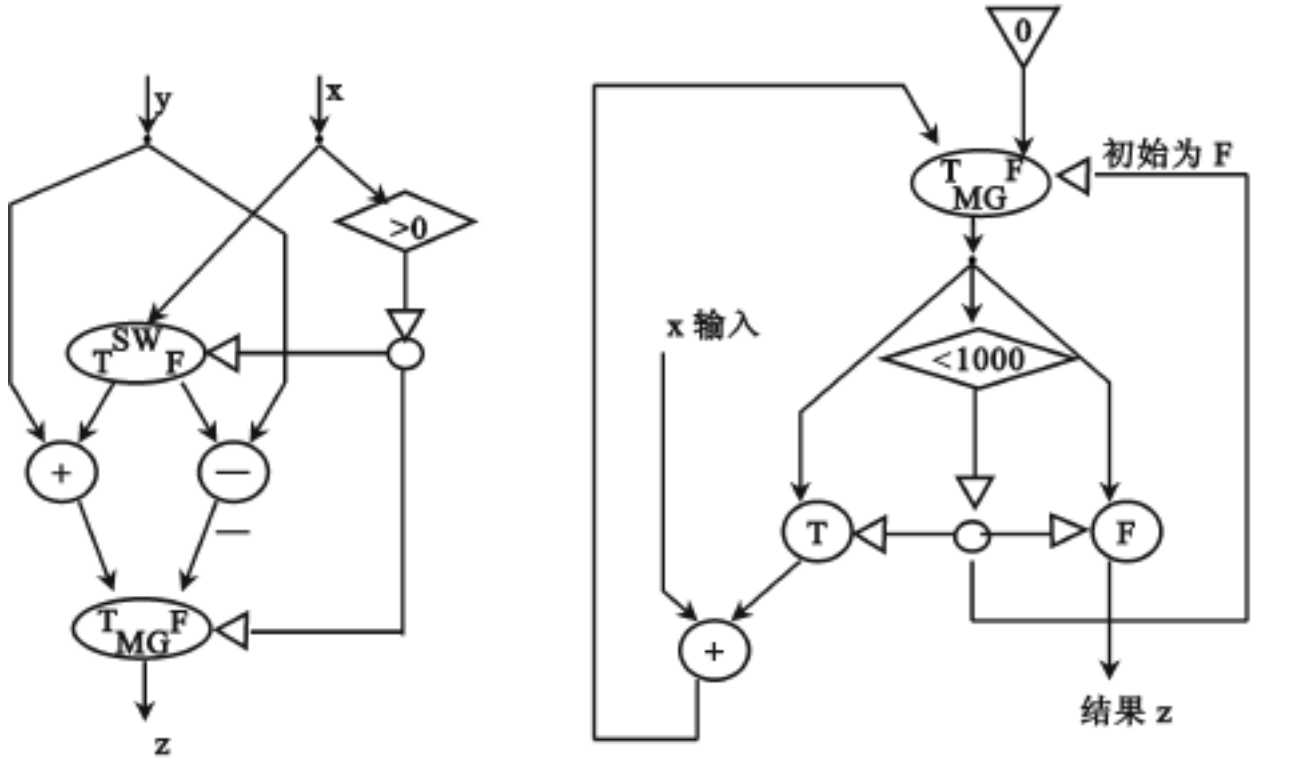


图 7 .6　具有条件分支和循环结构的数据流程图

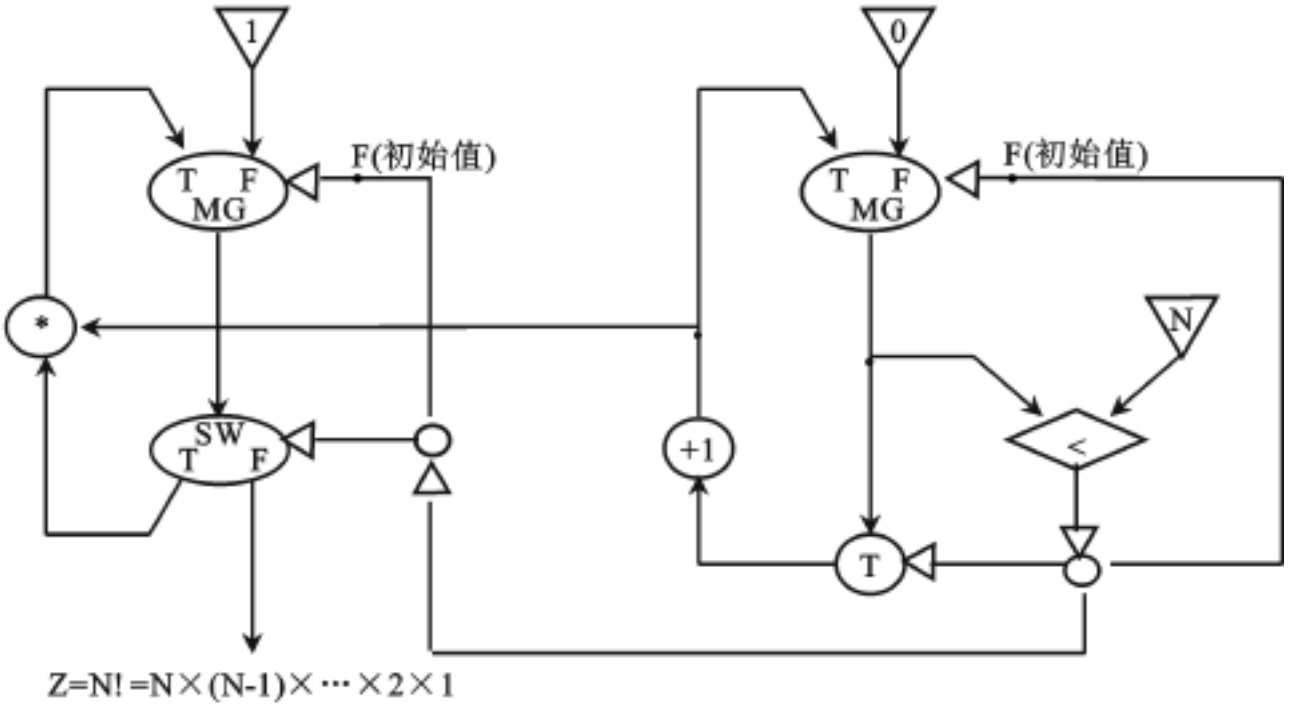


图 7 .7　计算 N 阶乘的数据流程图

在图 7. 6(a)中,当 x > 0 时,让 x 加 y,否则,就让 x 减 y。在图 7. 6(b)中,是实现

对 x 循环累加,直至其值超过 1000 为止,结果由 z 输出。

数据流程图实际上是数据流机器语言,较直观易懂,但编程效率很低。所以通常流行的是数据流语言,它的语言结构类似于命令式程序设计语言,但又要能方便地被编译成数据流程图,以在数据流计算机(简称"数据流机")上执行。

常用的数据流语言有美国的 ID 和 VAL,法国的 LAU 以及英国曼彻斯特大学的 SISAL 语言等,它们大都是单赋值语言。单赋值的含义是指,在程序中每个变量只能赋值一次,即同一变量在赋值语句的左部只允许出现一次,不允许对同一变量进行多次赋值。数据流语言一般必须具有如下的两个特点:

① 遵循单赋值规则。这有利于运算并行性的开发,同时也可防止"副作用"。所谓"副作用"是指在程序执行过程中修改了某些参数的值。

② 指令的执行次序由数据依赖关系确定,这就使指令执行规则简单地只依赖于数据的可用性。

美国的麻省理工学院(MIT)是研究数据流机的发源地,由 Jaok Dennis 教授领导的研究小组提出了 VAL 数据流语言,主要用于他们所开发的 MIT,静态数据流机。而由 Arvind 教授领导的研究小组首次提出了动态数据流机概念,并开发了相应的 ID 数据流语言。对于点积这样的操作,若用 ID 语言则可写成如下的过程:

```
procedure inner_product(a, b, n)
initial s←0
for i from 1 to n do
    new s←s + (a[i ] * b[i])
return s
```

7.2.3 数据流计算机的基本结构

依据处理数据令牌的不同,数据流计算机可分为静态和动态两大类。

1. 静态数据流机

这种数据流机的主要特点是数据令牌不带任何标号,在任何一条弧上只允许存在一个数据令牌。数据令牌沿数据流程图中的箭头方向流动,当到达某一节点时,能否激发该节点进行相应操作,要视该节点其他输入弧上是否都已有数据令牌到达。仅当该节点的所有输入弧上都出现有数据令牌且在它的任何输出弧上都没有令牌时,它才会被激发。

典型的静态数据流计算机的结构如图 7 .8 所示。指令存储部件中存放要执行的数据流程序,所有已收到全部所需数据令牌的指令将由指令部件按更新部件送来的指令地址逐个取出,送到可执行的指令队列中。此时若有空闲处理机,分派程序将按先后次序分配处理机给等待执行的指令,使它们并发执行。执行后的结果将形成新的数据令牌,它们被送到更新部件中,由后者按它们的目的地址送往指令存储部件内

相应指令的有关位置。与此同时,更新部件将已收到所有必需数据令牌的指令地址传送给取指令部件。这样便完成了一个循环流动。

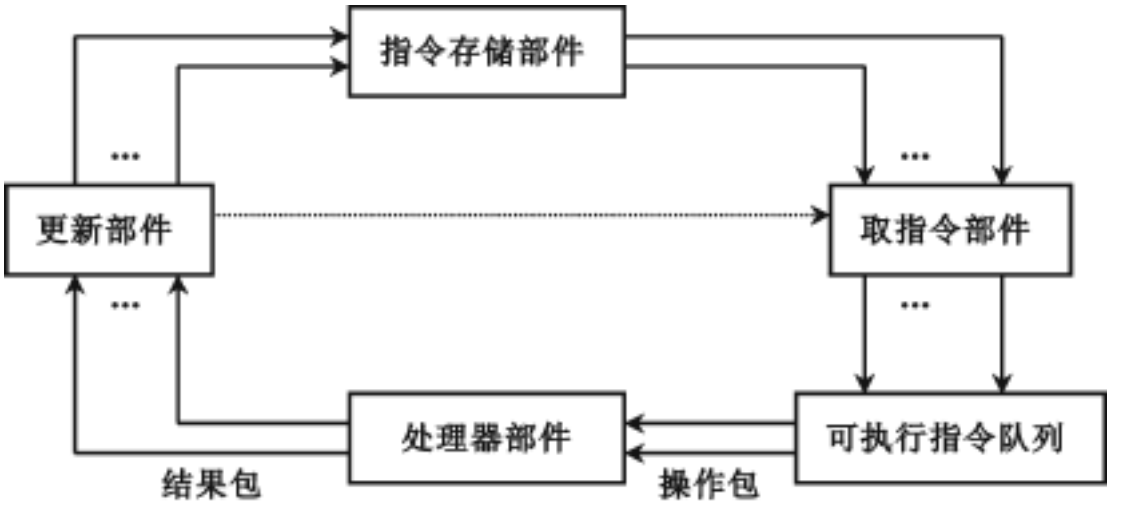


图 7 8　静态数据流计算机的结构

由于静态数据流机只允许任何一条弧线上有一个数据令牌,因此它只能处理一般循环操作。为了满足迭代要求,它必须多次重复激活某些操作节点,此外还必须设置控制令牌以识别属不同迭代层次的数据令牌,当数据令牌从输出弧上取走时,这些控制令牌就作为确认信号告诉所有输入弧上已到齐数据令牌的节点可以开始执行操作。

图 7 .9 中示出了 MIT 的静态数据流计算机结构。它由处理部件、指令存储部

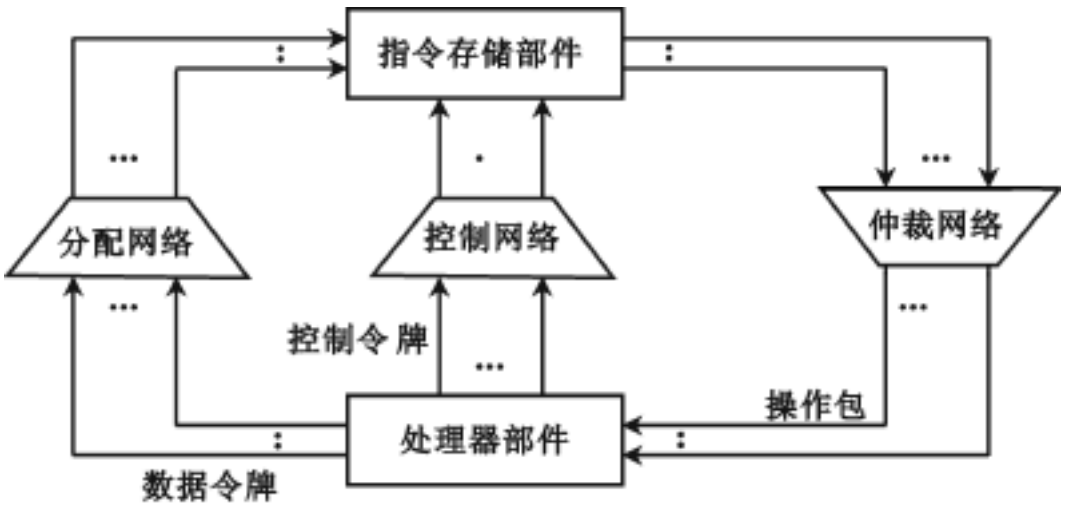


图 7 .9　MIT 静态数据流计算机结构

件、仲裁网络、分配网络和控制网络五部分组成,形成一个环状流水线。其中的仲裁网络和分配网络相应于图 7 .8 中的更新部件和取指令部件,所增加的仅是控制网络部分。指令存储部件中存放等待执行的指令,每条指令中含有相应等待接收操作数的空位,这些空位由分配网络送来的操作数填入。当全部空位被填满时,表明该指令所需的数据令牌已全部到达,此时若表示当前指令的一次执行已经结束的控制令牌

也已由控制网络送到该指令,则此指令便可被激活。该指令连同其操作数及结果送往目的地址形成一个可执行的操作包,由仲裁网络送往处理部件处理。

仲裁、分配和控制网络均以包交换方式通信,采用第六章所叙述的 Delta 多级开关网络加以实现。

2. 动态数据流机

动态数据流机的主要特点是使数据令牌带有标号。这样就可使数据流程序图中的任一条弧上同时存在带有不同标号(或称颜色)的数据令牌。此时,不需要像静态数据流机中那样依赖控制令牌来确认指令间数据令牌的传送,而只要对令牌标号进行符合比较就可加以识别。为此,需要有一个称为匹配部件的硬件机构将标号附加到数据令牌上,并完成标号的匹配工作。图 7 .10 中示出了这种动态数据流机的结构。

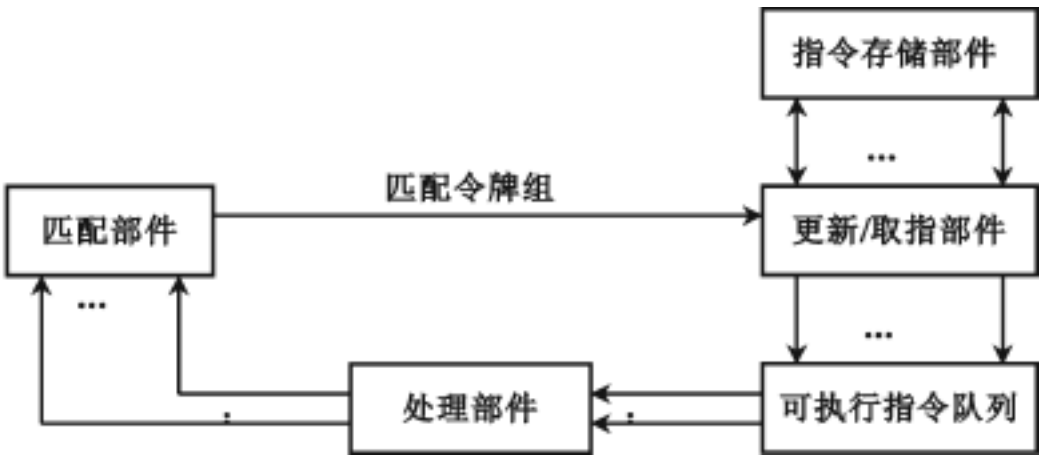


图 7 .10 动态数据流计算机结构

匹配部件将处理部件中各处理单元送来的结果数据令牌赋予相应的标号,并将流向同一指令的数据令牌匹配成对或组,然后将它们送往更新/ 取指部件,由该部件将需要它们的指令从指令存储器中取出,然后将已收到的数据令牌组和取出的指令合并成一条可执行的指令,送入可执行指令队列。由于在动态数据流机中给数据令牌赋予标号,从而可区分出不同层次的迭代执行。因此动态数据流机可最大限度地开发数据流程序图中的并行性。

动态数据流机可分为网络型和环型两种组成形式。它们的典型代表分别是 MIT 的动态数据流机和英国曼彻斯特大学的动态数据流机。图 7 .11 中示出了曼彻斯特大学的环型动态数据流机。该机进行数据流运算时所完成的主要功能是:对数据令牌进行匹配,将匹配成功的指令从指令存储器中取出和执行该指令。这三个功能分别由图 7 .11 中的匹配部件(MU)、指令存储器(IS)和处理部件(PU)完成,部件间按流水方式工作。为了缓冲同一时间由处理部件产生的多个数据令牌,在匹配部件之前设置了一个令牌队列(TQ)。每个令牌包由 4 部分组成:数据类型(5 位)、数

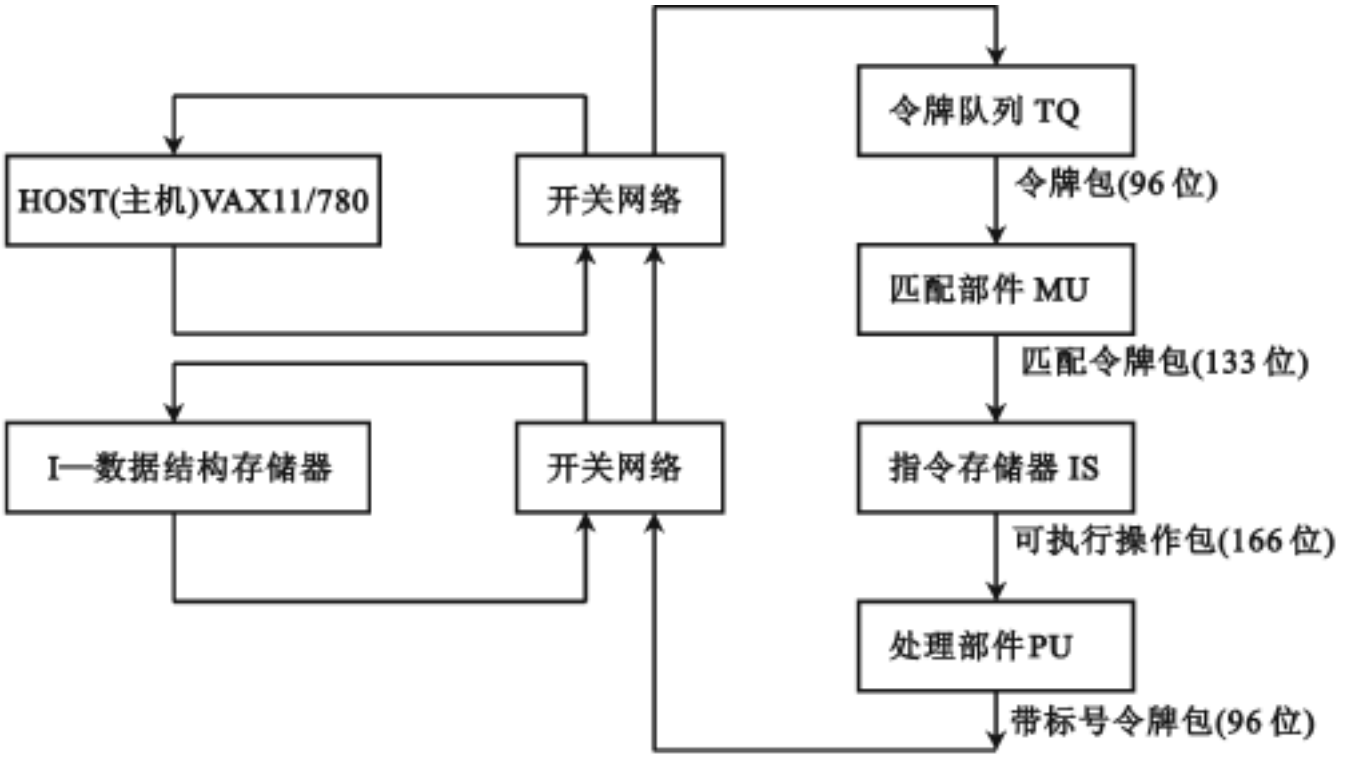


图 7 .11　曼彻斯特大学的环形数据流机的结构

据值(32 位)、标号(36 位)和目的节点的地址(23 位),总长度为 96 位。匹配后令牌包长度增至 133 位(增加了另一个数据令牌值的长度 32 位和类型位 5 位)。从指令存储器取出的操作码长度为 10 位,再加一个可选的目的地址长度 23 位,形成一个 166 位长的可执行信息包,送往处理部件。操作后输出的结果令牌包又变为 96 位长。各个部件异步工作,具有不同的时钟频率。每个 PU 中有 20 个功能部件,每个功能部件的最大处理速度为 0 .23MIPS,因此 PU 的持续性能可超过 1MIPS。TQ,MU,PU 以及指令存储器构成了一个称为 PE 的流水环,该样机最多可支持 7 个这样的流水环(通过开关网络连接到系统中)。样机的主机是一台 VAX ll/ 780,通过开关网络与流水环相接,此外开关网络还将 I—结构存储器连接到系统中。

I—结构存储器是一个数据结构存储器,这是为了提高数据流计算机在处理数据结构时的性能而设置的。如前面所述,数据流机中无共享的数据存储器,每个数据的使用都是一次性的,若要多次使用某个数据,就必须多次加以复制。对于单个标量元素,这种复制开销还不算大,但若要对整个数据结构进行多次复制,而且数据结构中包含的元素较多时,开销就很大了。例如,对于一个有 10 000 个元素的数据结构,如果欲将第 500 号元素赋予新值,则将得到一个新的数据结构。在有共享数据存储器的冯·诺依曼机中,只需用地址读出这一元素,加以修改后再写回就可以了。但在数据流机中还必须复制其余的 9999 个元素才能构成这一新的数据结构,因此时空开销都很大。采用传统共享存储器概念的 I—数据结构存储器,就是为了解决这一问题而设置的。应注意的是,数据结构本身不在系统中流动,而是以指向该数据结构的一个指针来代替。

7.2.4 数据流计算机存在的主要问题

由前面所介绍的数据流计算机的基本工作原理可以看到,数据流计算机的最大优点是能充分开发细粒度(指令级)的并行性,因此从原理上讲,由它组成的计算机系统可获取很高的性能,但它也存在以下一些问题:

① 指令级的数据驱动导致每条指令的执行有较大的时间开销,因为在运行时需要对每一个和每一次操作进行数据相关分析。

② 由于数据流程序的操作包代码长度较长,将占用较多的存储空间。此外由于数据流机没有共享的数据存储器,因此不能保存如数组那样的数据结构,当处理大型数组时就要复制大量数据,造成存储空间的很大浪费,并增加很多不必要的数据传输开销。

③ 当机器规模变大时,接到转接网络上的流水环数将增加,从而使转接网络变成系统性能的新瓶颈口。

由于上述的原因,使得数据流机至今仍不能成为商品化机器,日本在第五代计算机的研究计划中,曾试图把数据流机结构作为并行推理机的一种实现方案,并力图使之成为商品化机器,但最终并未获得成功。但数据流机的基本工作原理及其结构已在一些计算机,特别是专用计算机设计中得到了应用。

7.3 归 约 机

归约机和数据流机一样,都是基于数据流的计算模型,只是其采用的驱动方式不同。数据流机是采用数据驱动,执行的操作序列取决于输入数据的可用性;归约机则是需求驱动,执行的操作序列取决于对数据的需求,对数据的需求又来源于函数式程序设计语言对表达式的归约。

7.3.1 函数式程序设计语言

函数式程序设计语言是由所有函数表达式的集合、所有目标(也是表达式)的集合及所有由函数表达式到目标的函数集合三部分组成。函数是其基本成分,是从一批目标到另一批目标的映射。从函数程序设计的角度看,一个程序就是一个函数的表达式。通过定义一组“程序形成算符”(Program—Forming Operators),可以用简单函数(即简单程序)构成任意复杂的程序,也就是构成任意复杂函数的表达式。反过来,如果给出了一个属函数表达式集合中的复杂函数的表达式,利用提供的函数集合中的子函数经过有限次归约代换之后,总可以得到所希望的结果,即由常量构成的目标。函数表达式的每一次归约,就是一次函数的应用,或是一个子表达式(子函数式)的代换(还原)。

以表达式 z = (y - 1) × (y + x)为例,可以理解成 z = f(u),而 f(u)等价于 g(v)xh(w),

其中 g(v) = y - 1,h(w) = y + x,也就是说,函数 z = f(u)的求解可归约成求两个子函数 g(v)和 h(w)的积,而 g(v)和 h(w)又可以分别继续向下归约。

函数式程序本质上是属于解释执行方式,从函数式程序的归约来看,机器内部通常采用链表的存储结构,且依赖于动态存储分配,存储空间的大小无法预测,需要频繁地进行空白单元的回收,使空间、时间开销都较大,频繁的函数应用和参数传递,加上自变量动态取值,同样的计算往往要重复多次。所以,必须针对函数程序设计语言的特点和问题来设计支持函数式程序运行的新计算机,这就是归约机。

7.3.2 归约机的结构特点

归约机一般有如下一些结构特点:

① 归约机应当是面向函数式语言,或以函数式语言为机器语言的非 Von Neumann 型机器。其内部结构应不同于 Von Neumann 型机器。例如,应采取适合于归约的存储结构和存储器结构,要有函数定义存储器和表达式存储器,而不是 Von Neumann 型机器的程序存储器和数据存储器这种组成方式。归约机处理对象是多个运算或函数应用嵌套组合的表达式,处理器根据表达式携带的运算信息来处理表达式中的数据。因此,处理的数据和操作的信息应当合并存储,而不是像 Von Neumann 型机器那样,数据按地址存储,且数据中不含运算信息。应当有相应部件来跟踪指示表达式归约的顺序和路径,而不是 Von Neumann 型机器采用的程序计数器硬件。采用适合于归约特点的需求驱动或数据驱动的控制方式和机构,而不是 Von Neumann 型机器的按控制驱动的控制方式和机构。

② 具有大容量物理存储器并采用大虚存容量的虚拟存储器,具备高效的动态存储分配和管理的软硬件支持,满足归约机对动态存储分配及所需存储空间大的要求。

③ 处理部分应当是一种有多个处理器或多个处理机并行的结构形式,以发挥函数式程序并行处理的特长。

④ 采用适合于函数式程序运行的多处理机互连的结构,最好采用树形方式的互连结构或多层次复合的互连结构形式。

⑤ 为减少进程调度及进程间通信开销,尽量把运行进程的节点机安排成紧靠该进程所需用的数据,并使运行时需相互通信的进程所占用的处理机也靠近,并使各处理机的负荷平衡。

7.3.3 面向函数式语言的归约机

归约机按其归约模型可分为串归约(String Reduction)机和图归约(Graph Reduction)机两类。两者的区分主要是对函数表达式所使用的存储方式不同,前者以字符串形式存储而后者则以图的形式存储。

我们仍以表达式 z = (y - 1) × (y + x)为例,假定 x 和 y 分别赋以 2 和 5。

串归约方式是当提出求函数 z = f(u)的请求后,立即转化成执行由操作符“ × ”和

两个子函数 g 与 h 的作用所组成的“指令”。g 和 h 的作用又引起“指令”(-y,1)和(+y,x)的执行。于是,从存储单元中分别取出 y 和 x 的值,算出 y-1 和 y+x 的结果,然后将返回值再各自取代 g 和 h,最后求(×4,7),得结果 28。这种归约方式表示见图 7.12(a)。串归约方式实际上是一种不断地在定义表达式集合中去查找和复制的过程,而且对每次函数作用都要重复执行,因而时间和空间的辅助开销都比较大。

图归约方式与串归约方式主要的不同在于,定义表达式时设置了 z/1,z/2 等指针,如图 7.12(b)所示。这样,下一层作用的返回结果将直接取代上一层作用的自变量,省去了归约时的复制开销;同时,实现了自变量返回值的共享,不用对同一函数作用重复执行,就可以直接引用此函数求值的结果。

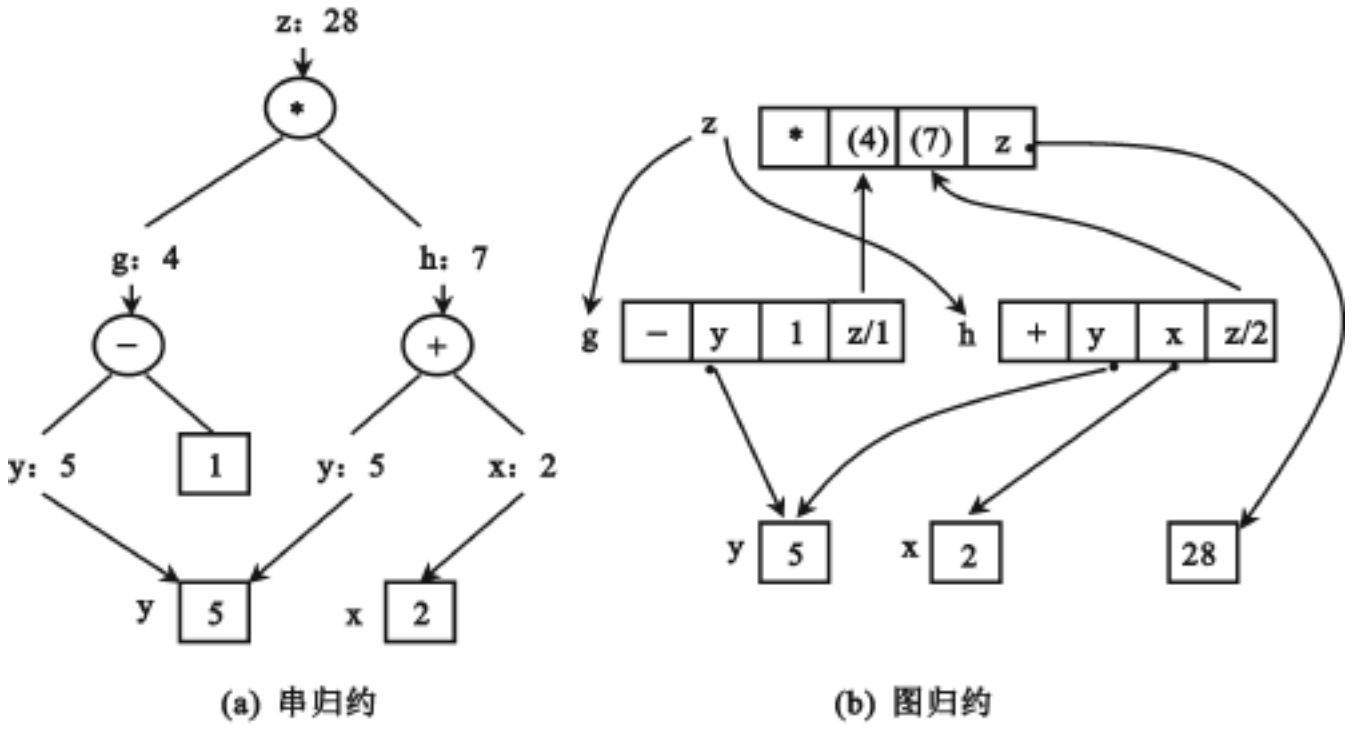


图 7.12 串归约和图归约

总之,归约方式体现了按需求驱动的思想,根据对函数求值的需求来激活相应的指令。而且,不论是采用先内后外,或先外后内,或先右后左,还是先左后右的顺序归约,也不论是采用串行归约,还是并行归约,都不影响最终结果值。

根据机器所用归约方式的不同,相应就有串归约机和图归约机两类。

7.4 人工智能计算机

近年来,人工智能技术已经在许多领域中得到了广泛的应用,例如自然语言理解、计算机视觉和机器人等。随着人工智能应用从实验室走向现实世界以及人工智能软件复杂性日益增加,计算吞吐率和成本已成为设计人工智能计算机时需要仔细

权衡的因素。由于传统的 Von Neumann 计算机主要是为顺序的和确定性数值计算而设计的,因此不适宜于人工智能方面的应用。

7.4.1 人工智能计算特征

人工智能(Artificial Intelligent, AI)计算机与传统的 Von Neumann 计算机不一样,因此两者的计算特征有很大差异。AI 的计算有如下特征:

① AI 计算的主要对象是符号而不是数值,常用的基本符号操作包括比较、选择、排序、匹配、逻辑集合运算(合、交和非)、分类以及模式检索和识别等。在更高层次上,符号操作还应包括如句子、语音、图形和图像等的模式操作。

② AI 计算是非确定计算。即很多 AI 算法具有不确定性,这是由于缺少对求解问题的完全理解和缺少相应知识造成的。因而在求解时,往往要对所有的可能性进行穷尽的枚举或是对求解空间进行有控制的搜索。

③ AI 计算是动态进行的。由于缺少完整知识和对求解过程的预见性,往往需要在求解过程中建立新的数据结构和函数。此外,由于求解问题所需的存储空间较大,无法在事先一次分配到位,因而不得不在求解问题过程中对存储空间以及其他资源进行动态的分配和回收。任务可能是动态建立的,通信拓扑也可能需要动态地变换。

④ 具有并行和分布处理的巨大潜力。在确定性算法中,往往存在有一组独立任务可以并发地进行处理,这类并行性称为 AND 并行性,由于 AI 算法具有不确定性一面,从而为并行处理提供了 OR 并行性的机会。

⑤ 知识管理问题。在减少 AI 求解问题的复杂性方面,知识起着很重要的作用,有用的知识越多意味着无用的搜索越少。由于知识量很大,因此必须加以管理。此外,所获取的知识可能有模糊成分,具有试探和不肯定特性。因而知识的表达、管理、加工以及知识学习都是很重要的问题。

需要指出的是,在许多 AI 应用中,求解时需用的知识可能是不完善的,因为在求解问题时,可能知识来源还未曾知道,或者在设计时无法预见将来的环境变化,所以 AI 系统应设计成开放的,允许不断求精和能够获取新知识。

总的来讲,改善 AI 任务求解的计算效率有两个基本方法,即使用探测知识指导搜索和使用更高速的计算机。AI 处理的基本要点是有关知识的获取、表示和智慧地加以使用。就知识获取而言,AI 系统应能从视觉、声音和书写等各种信息源获取信息。由于这些信息的来源往往是不完整、不精确,甚至是相互矛盾的,因此必须对它们进行正确的识别和理解。

知识的表示主要是对有关对象、关系、目标、动作以及处理过程的信息加以编码,形成数据结构和编写成过程。知识的表示应便于增加、修改和便于对知识进行加工。目前常用的知识表示方式有语义网络、框架结构、脚本、对象、生成系统、过程模式、谓

词逻辑以及关系数据库等。

知识的处理主要用于问题求解、逻辑演绎和情报检索等。常用的主要方法有:状态空间的遍历、问题归约、正向和反向的勾链、分辨、规划、探试搜索以及剪枝等。在这些方法中,对输入、输出的要求远大于对计算的要求,而且操作往往是在全局范围内进行的,不具有局部性。

7.4.2 AI 计算机的分类和设计方法

AI 计算机一般可分为四类:

(1)基于语言的 AI 计算机

这类 AI 计算机的设计目标是高效地执行像 LISP,PROLOG 和 FP 等面向 AI 的高级程序设计语言。在这类机器中设有专门硬件以支持有关语言的基本操作。如 LISP 机中采用面向堆栈的带标志位的系统结构。FP 计算机中采用图或链表来表示程序和数据,通常用多处理机来加以实现。PROLOG 机则需要有硬件支持模式匹配和归一的基本操作,并应有逻辑推理功能部件。

(2)基于知识的 AI 计算机

这类 AI 计算机要求能对具体所采用的知识表示方法,如语义网络、框架、规则、对象等给以支持。比较常见的有:基于规则的 AI 计算机,如 DADO 2 以及 NON—VON 计算机;基于语义网络的 AI 计算机,如 Connection machine;基于对象的 AI 计算机,如 FAIM-1,Dorado 和 SOAR。这些 AI 计算机中通常包含有大量处理器,每个处理器的功能比较简单且带有一个小容量的局部存储器。

(3)连接式(Connectionist)AI 计算机

在这种 AI 计算机中,知识不是用符号来表示的,而是直接编码成处理单元间的互连模式。这些处理单元既协同又相互竞争地求解问题。连接式 AI 计算机主要使用连接来实现信息的存储,类似于神经或细胞的互连。系统中每一个连接被赋有一定权值,由权值的模式来形成知识的表示。连接关系将大量的处理细胞元连接起来,每个细胞只完成非常简单的基本操作,如位比较等。在连接系统中的每个细胞元计算所有与它相邻细胞元状态的带权和。该细胞元的新状态是该带权值和该细胞元以前状态两者的函数。连接式 AI 计算机的最大优点是,它能就近地同时将整个知识库系统作用于求解问题,所有细胞元是并发地活跃的,所有计算直接受控于编码在网络连接中的知识库。从原则上讲,连接式系统的求解时间是固定的,不受求解问题大小的影响。由于知识被编码在整个细胞元的网络中而不像一般符号处理式的 AI 计算机是存放在特定的存储单元中,因此即使失去个别细胞元,对整个系统性能不会有很大影响,故具有较强的容错能力。连接式 AI 计算机的最大弱点是可编程性和可维护性较差,因此这种 AI 计算机仍处于实验研究阶段。

(4)带智能接口的 AI 计算机

在这种 AI 计算机中,通常在人—机之间提供语音识别、自然语言理解、图像处理和计算机视觉等智能接口。这种 AI 计算机的应用针对性和实用性较强,因此有较广阔的应用前景。应指出的是,在这种系统中不单纯进行符号处理,数值计算也占有相当比例,如对语音和图像的低层次处理主要是数值计算,仅在高层次上才对音素和图像的符号表示加以处理。

7.4.3 PROLOG 推理机

由于日本于 1982 年开始的第五代计算机研究计划中,把 PROLOG 语言作为样机的核心语言,从而引起了对 PROLOG 推理机的研究热。充分开发存在于用 PROLOG 语言书写的程序中的各种并行性,可以加快 PROLOG 程序的执行。在 PROLOG 程序中一般存在有 AND,OR 和归一三种并行性。

AND 并行性是指逻辑上相与的子句的同时执行,即要获得主目标的求解,必须先使所有子目标同时获得求解,但这种 AND 并行性操作是在所有子目标共享的变量和数据集上进行的,因此需要进行一致性检查。

OR 并行性是指对所有可能求解路径的并发搜索。由于与每条求解路径有关的变量和数据互不共享,因此不需要进行一致性检查。

归一并行性是指在 PROLOG 数据库中将所有子句进行并行匹配,并将匹配变量并发地固化为常数值。

为了支持上述三种并行性的开发,日本在第五代计算机研究计划中研制了并行推理机的样机 PIM。该系统由多个推理模块组成,每个模块有自己的局部进程池和归一部件,如图 7.13 所示。一个专用互连网络将所有推理模块连接起来,此外进程池可通过该互连网络在各推理模块间迁移。系统中还有另一个网络,它用来连接推理模块和结构存储器模块,后者主要用来存放表、向量、矩阵以及其他的结构数据。

7.4.4 AI 计算机的研究进展

日本于 1982 年到 1991 年开展了为期 10 年的第五代计算机研究计划,即人工智能计算机的研究计划。受其激发,美国、英国以及西欧各国也相继在此期间开展了人工智能计算机的研究,研制出不少样机,取得了不少阶段性成果。

日本第五代计算机的研究计划分三个阶段进行。第一阶段为 3 年,主要是对并行推理机的基本结构、并行推理机制以及知识库机结构等进行研究和模拟试验。阶段成果是推出了顺序推理机 SIM(Sequential Inference Machine)样机以及 Delta 关系数据库机。第二阶段用了 4 年时间,侧重研究并行推理机 PIM(Parallel Inference Machine)以及知识库子系统的管理软件系统。最后阶段的 3 年,主要是构成一个智能信息处理系统,它以 VLSI 技术实现的知识库机和并行推理机为硬件核心,并配以基本核心软件和应用软件。由于对人工智能机的基本问题的认识还不成熟,再加上

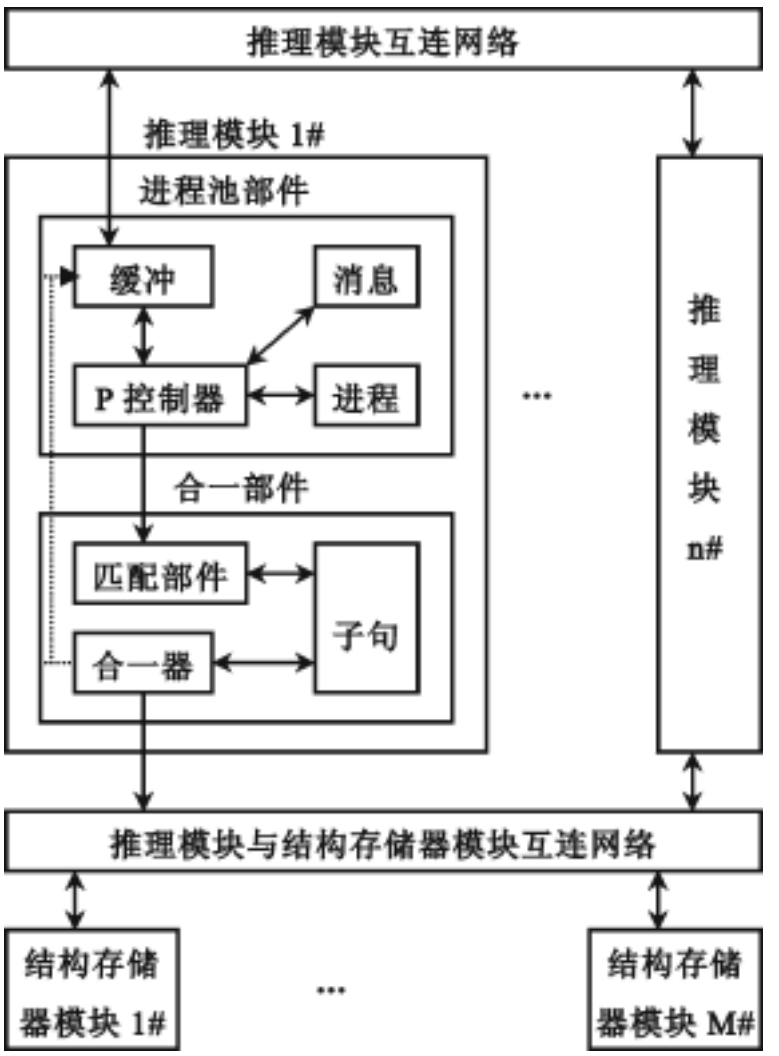


图 7.13 并行推理机 PIM 的结构

期望的研究目标定得过高,因此日本第五代计算机研究计划并不算成功。但不管如何,第五代计算机研究计划的实施,使人工智能计算机的研究前进了一大步,同时也从中获得了宝贵的经验和教训。

目前不少研究人工智能的专家认为,人工智能机研究的关键在于对人工智能软件的研究和开发,人工智能机的硬件工作平台应使用通用的高性能并行机。花费众多的人力、物力和财力去研究和开发专用的工作平台来实现人工智能机的方法并不经济有效。一般而言,用专用芯片构成的工作平台的性能仅能将速度提高 10 倍,而通用芯片的速度大约每 3 年就能提高 10 倍,这样专用机工作平台,最多能保持 3 年优势,然后就会被通用芯片赶上。此外,用专用芯片构成的工作平台,不但价格昂贵,而且换代升级很困难,因此缺乏竞争能力。

7.4.5 RWC 研究计划

日本在研制第五代计算机的基础上,自 1992 年起又推出了一个为期 10 年、称为真实(现实)世界计算机(Real World Computing, RWC)的研究计划。所谓真实世界

是指在不断产生变化的、不完全的、包含互相矛盾的、复杂相关的信息的世界。21 世纪是高度信息化社会,需要处理的信息不但量大,而且随着多媒体技术、各种传感技术的迅速发展以及计算机的应用领域日益扩展,使得要处理的信息的质的多样化也不断增大。所要处理的这些信息往往是模糊、复杂和不确定的。人类在处理这些信息时,通常是采用直觉判断方式,这种柔性(灵活)的信息处理方式,比起传统计算机以及智能推理机采用硬性(刚性)的信息处理方式要有效得多。编程困难的软件危机,促成了对智能推理机的研究,而算法设计困难的软件危机则促进了对柔性信息处理系统的研究。

RWC 研究计划是由日本通产省主持的,预期投资 700 亿日元。研究计划将分前、后两期进行,每期时间为 5 年。要求最终建成的柔性信息处理系统,使任何人都可以存取和利用网络上的大量信息资源而不感觉到计算机的存在。此外系统应具有开放性(Openness)、强壮性(Robustness)和实时性(Realtime)。

RWC 计划的主要研究内容有以下四个方面:

① 柔性信息处理的理论基础研究。也称为软逻辑基础。其中包括:信息的柔性表示、存取和调用;各种信息的综合及各种处理模块的集成;对信息及处理模块的评价;自学习和自组织能力以及优化技术。

② 面向应用的新功能的开发研究。研究内容包括:柔性识别与理解(包括柔性的图像理解、语音理解、自然语言理解、手势理解等);柔性推理与问题求解;柔性的人机界面(如虚拟现实)与模拟;柔性自主控制。

③ 超并行计算系统。主要研究内容为:超并行系统结构及超并行软件。

④ 神经系统。研究内容涉及:神经模型;其他生物功能,如人工生命、遗传算法、进化算法、生态学算法等;神经系统的硬件(VLSI 神经芯片)及软件(神经网仿真系统、神经网语言及操作系统等)。

RWC 计划中所要研制的超并行计算机系统结构将具有如下重要特征:

① 能适应不确定计算的柔性和具有高性能的计算能力。

② 具有通用性,能支持编程的多种风范。

③ 具有高安全性和高可靠性。此外需有实时处理能力。

④ 具有支持并行操作系统的功能。

其中涉及到的新技术和新概念有:减少处理机间通信开销;超线程(Super-thread);通信速率可与计算速度相比的高性能互连网络以及对超并行操作系统的支持。

RWC 计划预定在 1996 年能推出 RWC 的样机系统,有 1024 个处理单元,性能可达到 100GFLOPS。在对 RWC 样机进行系统评价后,将开发柔性信息处理系统的工作平台。这种平台系统将是含有 l 百万个以上的处理单元的通用超并行计算机,性能预期可达 125TIPS,即每秒可执行 125 万亿条指令。神经网系统也是超并行计

算机系统的原型之一,期望规模可达到 100 万个单元,速度指标为 10TCPS,即每秒可完成 10 万亿次连接更新。

当然 RWC 计划的主要目标并不单是研制一台具有柔性信息处理能力的计算机,而是在同时还要开发柔性计算所需的基本技术,包括并行系统结构、操作系统、语言以及系统支撑环境等。

习　　题

1. 简述脉动阵列结构的特点。
2. 什么叫控制驱动、数据驱动、需求驱动?
3. 用常用节点画出以下各式的数据流程序图:

① $Z = x^n$

② if (a = b) and (c < d)
then c←c - a
else c←c + a

③ $x = (-b \pm \sqrt{b^2 - 4ac})/(2a)$

4. 用节点有向图形式画出求解

$$x = \sqrt{(a+b) * d/e - e/d}$$

的数据流程序图,当 a = 4, b = 8 时,表示出该数据流程序图的执行过程。

5. 用常用节点画出下面的矩阵相乘表达式的数据流程序图:

$$C_{ij} = \sum_{k=1}^{N} A(i,k) \times B(k,j)$$

其中,k 为最内层循环中的下标变量,i 和 j 是外层循环中的下标变量。为简化起见,可用 Get 节点表示自 I—结构存储器中读取 A(i,k)和 B(k,j),可用 Store 节点表示存放 C(i,j)到 I—结构存储器中。

6. 用常用节点画出

z: = (IF X = 10 THEN X - Y ELSE (X + Y)/ Y)

的数据流程序图。

7. 画出对应于循环语句

```
WHILE i < 0 DO
new Z: = Z + X;
i: = old i + 1
END
```

迭代结构的数据流程序图。

8. 静态和动态数据流机的主要区别在哪里？

9. 人工智能计算机一般应分为哪几类？它的主要特点是什么？

主要参考文献

1 .陆鑫达 .计算机系统结构 .北京:高等教育出版社, 1996

2 .郑纬民,汤志忠 .计算机系统结构(第二版) .北京:清华大学出版社, 1998

3 .李学干 .计算机系统结构(第三版) .西安:西安电子科技大学出版社, 2000

4 .屈玉贵,梁晓雯 .并行处理系统结构 .合肥:中国科技大学出版社,1999

5 .张昆藏 .计算机系统结构——奔腾 PC .北京:科学出版社,1999

6 .张昆藏 .奔腾Ⅱ/ Ⅲ处理器系统结构 .北京:电子工业出版社,2000

7 .John L .hennessy, David A .Patterson 著,郑纬民等译 . Computer Architecture:a quantitative approach(第二版) .北京:清华大学出版社,2002

8 .John L .hennessy, David A .Patterson .Computer Architecture:a quantitative approach(英文版第三版) .北京:机械工业出版社,2003