计算机体系结构复习笔记

系笔者自行总结，仅供参考

1计算机体系结构的基本概念

1.1 计算机体系结构的概念

存储程序计算机

存储程序计算机又称为**冯诺依曼结构**的计算机，由四个部分组成：控制器、运算器、存储器、输入输出设备

有以下特点：

1. 以运算器为核心（现在计算机体系结构以存储器为核心，以解放cpu性能）
2. 采用存储程序原理。指令和数据被一视同仁地存放在存储器。
3. 存储器按地址访问、线性编址。
4. 控制流由指令流产生。
5. 指令由操作码和地址码组成。
6. 数据以二进制编码表示。

本书中一条指令的操作分解为取值、译码、执行、访存、写回五个部分。

1.1.2 计算机体系结构、组成和实现

计算机体系结构的概念于1964年由阿姆道尔提出，他将其定义为：计算机体系结构是程序员所看到的计算机的属性，即概念性结构与功能特性。这些属性主要包括：

1. 数据表示
2. 寻址规则
3. 寄存器定义
4. 指令系统
5. 中断系统
6. 机器工作状态的定义和切换
7. 存储系统
8. 信息保护
9. I/O结构

这些属性由硬件或固件完成提供，因此，经典的计算机体系结构概念的实质是计算机系统中软硬件的分界面。

本书中体系结构的概念主要包括计算机指令系统、计算机组成原理和计算机硬件实现三个方面。

1.1.3 计算机系统中的层次概念

计算机系统按功能可划分为以下层次结构：

1.1.4 系列机和兼容

兼容分为以下四种：

向上/下兼容：当前机器上编制的程序能运行在更高/低档的机器上。

向前/后兼容：当前机器上编制的程序能运行在之前/后投入市场的机器上。

向后兼容是最重要的。

1.2 计算机体系结构的发展

1.2.1 并行处理技术的发展

从执行程序的角度，并行性由低到高分为：

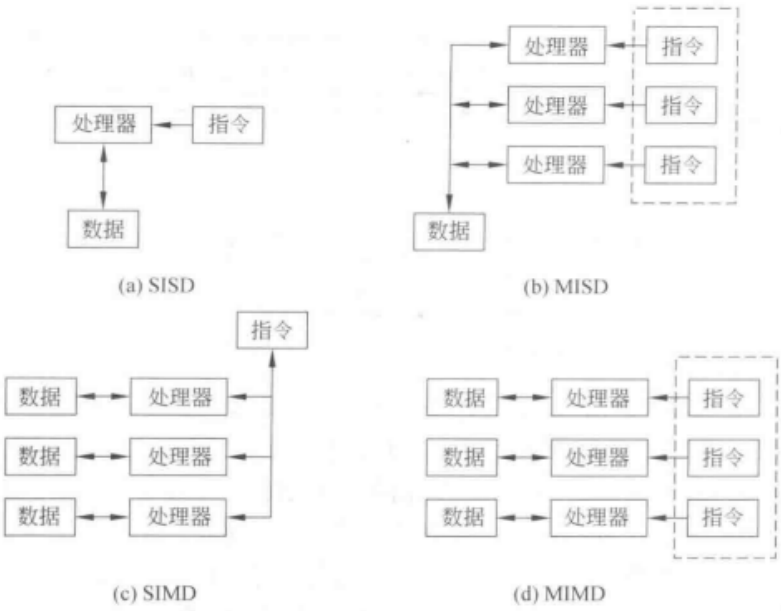
1. 指令内部并行
2. 指令级并行
3. 线程级并行
4. 任务级或过程级并行
5. 作业或程序级并行

从数据处理的角度，并学习由低到高分为：

1. 字串位串
2. 字串位并
3. 字并位串
4. 全并行

Flynn分类法：

1. SISD：指令和数据都不存在并行
2. SIMD：一个指令流多多个（组）数据进行相同操作。如GPU
3. MISD：多个指令流在一个数据流上进行操作。可用于容错计算，如三模冗余。
4. MIMD：多个独立的处理器在不同的数据上执行不同的操作，这是常见的并行处理模型，如多核处理器。



提高并行性的三种途径：

1. 时间重叠：流水
2. 资源重复：堆硬件资源
3. 资源共享：分时复用、多道程序

1.3.3 量化设计的基本原则

1. 大概率事件优先原则
2. Amdahl定律

   定义系统加速比如下：

$$\text{系统加速比} = \frac{\text{系统性能}_{\text{改进后}}}{\text{系统性能}_{\text{改进前}}} = \frac{\text{总执行时间}_{\text{改进前}}}{\text{总执行时间}_{\text{改进后}}}$$

   系统加速比等于2就是说加速后速度是原来的两倍或者说运行时间是原来的二分之一。

   同理可以定义部件的加速比。

   设系统加速比为 α，部件加速比为 β，可改进比例为 k，改进前总执行时间为$T_{前}$，改进后总执行时间为$T_{后}$。

   那么有：

$$\alpha = \frac{T_{前}}{T_{后}} = \frac{T_{前}}{(1-k)T_{前} + kT_{前}/\beta} = \frac{1}{1-k+\frac{k}{\beta}}$$

   Amdahl定律表明：

   - 加快某部件执行速度所获得的系统性能加速比取决于该部件在系统中所占的重要性。
   - 仅仅改进系统的某个部分，收益是递减的，这是因为

$$\frac{d\alpha}{d\beta} = \frac{k}{((1-k)\beta + k)^2}$$

     关于 β 单调递减，趋向于0。

     显然 α 的极限是 $1/(1-k)$。

- 前两条的推论是：理想情况下，想要获得最好的收益，应该加速系统中的瓶颈部分，直到该部分快到不再是系统的瓶颈，此时就应该去加速另外一个成为瓶颈的部分了。一个理想的计算机系统应该是 “没有短板” 的。

3. 程序的局部性原理
4. CPU的性能

   IC指一个程序的指令条数。

   指令时钟数（Cycle Per Instruction）是指指令需要的时钟周期数。不同指令需要的时钟周期可能不同，所以程序的指令时钟数一般是对所有类型的指令加权平均的（指令频次为权），即

$$CPI = \sum(CPI_i \times IC_i)/IC$$

   总CPU时间T可以写成

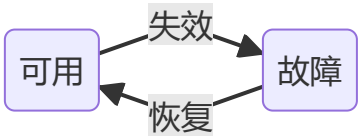
$$T = CPI \times IC \times T$$

   其中 T 是CPU的时钟周期。

1.4 可靠性问题

1.4.1 基本的可靠性模型

系统又两个状态：可用和故障。两个状态可以相互转化



1.4.2 常用指标

MTTF（Mean Time To Failure，平均无故障时间）：平均而言处于可用状态多久会发生一次失效。MTTF 的倒数就是系统的失效率。如果系统中每个模块的生存期服从指数分布，则系统整体的失效率是各部件的失效率之和。

$$\lambda = \frac{1}{MTTF}$$

MTTR（Mean Time To Repair，平均修复时间）：平均而言处于故障状态多久会发生一次恢复，即系统维修所需时间。

MTBF（Mean Time Between Failure，平均失效间隔时间）：平均而言两次失效之间的间隔时间

$$MTBF = MTTF + MTTR$$

一般 $MTTF >> MTTR$，因而 $MTBF \approx MTTF$。

FIT（Failure In Time）：10亿小时中系统的故障次数。

系统可用性（Availability）：系统可用时间的比率

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$$

2 指令系统

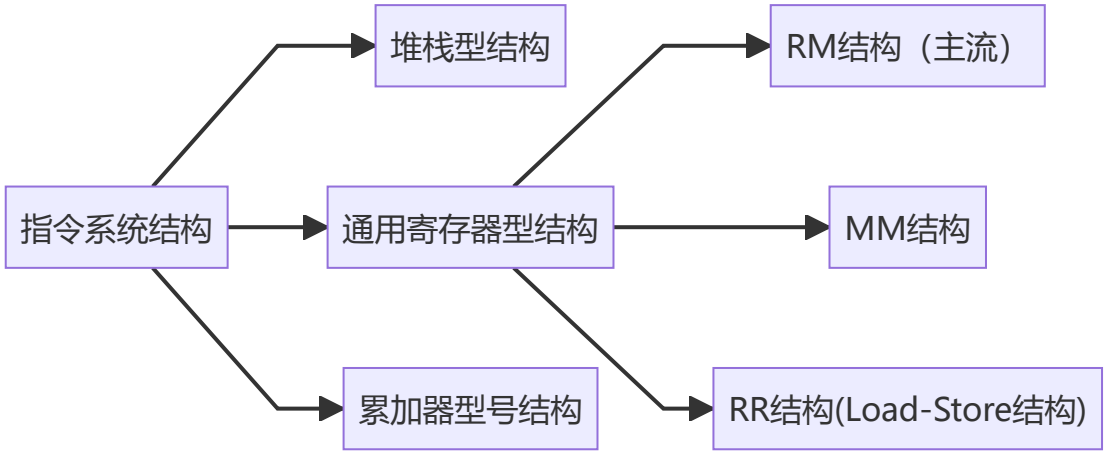
2.0 本书的指令语法

- 符号说明：
  1. ← 表示赋值
  2. Mem 表示存储器
  3. Regs 表示通用寄存器组
  4. `[]` 表示内容。比如 Mem[120] 就表示地址处120存储的内容， Regs[R1] 就表示 R1 寄存器存储的内容。
- 运算指令 `OP R1,R2` 一般是 `Regs[R1] <-- Regs[R1] OP Regs[R2]`

  运算指令 `OP R1,R2,R3` 一般是 `Regs[R1] <-- Regs[R2] OP Regs[R3]`
- `#x` 表示立即数 x
- `(R1)` 表示 `Mem[Regs[R1]]` 。`120(R2)` 表示 `Mem[120+Regs[R2]]`

2.1 指令系统结构的分类

CPU 中用来存放操作数的存储单元有三类：堆栈、累加器和通用寄存器组。据此可把指令系统分为**堆栈型结构**、**累加器型结构**和**通用寄存器型结构**。可以依指令操作数的个数以及来自存储器的操作数的个数对通用寄存器型结构进一步细分为**寄存器-存储器型结构（RM结构）**、**寄存器-寄存器型结构（RR结构）**、**存储器-存储器型结构（MM结构）** 。RR结构有且仅有 Load 和 Store 指令能访存(因此也叫**Load-Store 结构**，运算操作的源操作数和目的操作数只能来自/送往寄存器。



堆栈型结构的两个操作数分别为栈顶和次栈顶，运算后的结果写入栈顶，三者都不用显式给出；累加器型结构的一个操作数来自累加器，另一个操作数显式给出，结果写入累加器；RM结构一个操作数来自存储器，另一个操作数来自寄存器，结果写回寄存器；RR结构两个操作数都来自寄存器，结果写回寄存器。

优缺点分析

堆栈型结构和累加器型结构的优点：指令字短、占用空间小。

堆栈型和累加器型的缺点：

- 堆栈型结构
  1. 不能随机访问堆栈，难以生成有效代码

2. 对栈顶的访问会成为性能瓶颈

- 累加器型结构

  只有一个暂存器，因此需要频繁地访存

RM结构和MM结构的优点：目标代码紧凑

RM结构和MM结构的缺点：指令字长变化大（硬件设计复杂）、指令的执行周期变化大（不利于流水）、频繁访存会成为性能瓶颈

RR结构的优点：

1. 寄存器的访问速度比存储器快很多
2. 编译器能够灵活地使用寄存器，以产生更为高效的代码。比如调度代码以为充分发挥流水线性能。[[C4 指令集并行]]

如果硬要说缺点的话，RR结构的指令条数多，目标代码不够紧凑、

2.2 寻址方式

寻址方式指指令系统产生要访问的数据的地址的方法。

2.2.1 常用寻址方式

寻址方式	指令实例	含义
寄存器寻址	ADD R1,R2	Regs[R1]<--Regs[R1]+Regs[R2]
立即数寻址	ADD R3,#6	Regs[R3]<--Regs[R3]+6
寄存器间接寻址	ADD R3,(R2)	Regs[R3]<--Regs[R3]+Mem[Regs[R2]]
偏移寻址	ADD R3,120(R2)	Regs[R3]<--Regs[R3]+Mem[120+Regs[R2]]
索引寻址	ADD R4,(R2+R3)	Regs[R4]<--Regs[R4]+Mem[Regs[R2]+Regs[R3]]
直接寻址或绝对寻址	ADD R3,(1010)	Regs[R3]<--Regs[R3]+Mem[1010]
存储器间接寻址	ADD R3,@(R2)	Regs[R3]<--Regs[R3]+Mem[Mem[Regs[R2]]]
自增寻址	ADD R3,(R2)+	Regs[R3]<--Regs[R3]+Mem[Regs[R2]]；Regs[R2]<--Regs[R2]+d
自减寻址	ADD R3,(R2)-	Regs[R2]<--Regs[R2]-d；Regs[R3]<--Regs[R3]+Mem[Regs[R2]]
缩放寻址	ADD R4,80(R2)[R3]	Regs[R4]<--Regs[R4]+Mem[80+Regs[R2]+Regs[R3]*d]

- 除此之外，还有PC相对寻址，这将在2.3.2讨论。
- 偏移寻址如果偏移量为0，就退化为了寄存器间接寻址；如果寄存器中的值为0，就退化为了直接寻址。

2.2.2 采用多种寻址方式的优缺点

优：减少程序的指令条数

缺：增加计算机的实现复杂度；增加 CPI

2.2.3 寻址方式的表示

1. 隐含在指令的操作码中
2. 在指令中设置专门的寻址字段，用以直接指出寻址方式

2.2.4 信息存储的整数边界

如果一台机器：

1. 能同时存放不同宽度的信息，如字节、半字（2字节）、单字（4字节）、双字（8字节）
2. 按字节编址，各类信息都用该信息的首字节地址来寻址

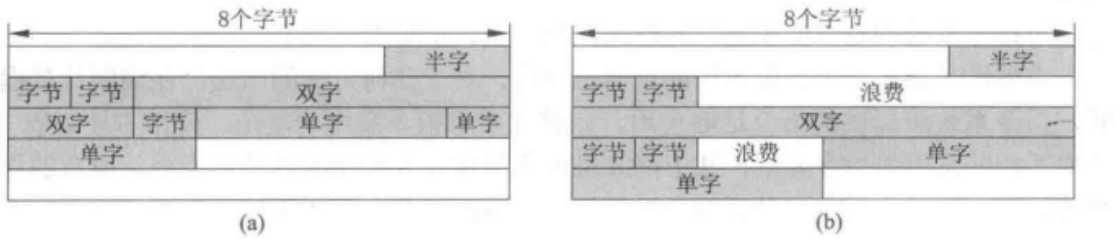
则可能出现一个信息跨存储字边界而存储于两个存储单元的情况。此时读出信息需要花费两个存储周期，不可接受。

解决方案：要求宽度不超过主存宽度的信息必须存放在一个存储字内，为此信息在主存中存放的起始地址必须是该信息宽度（字节数）的整数倍。即满足以下条件：

- 字节信息的起始地址为：x...xxxx
- 半字信息的起始地址为：x...xxx0
- 单字信息的起始地址为：x...xx00
- 双字信息的起始地址为：x...x000

这就是**信息存储的整数边界**的概念

假设主存带宽为8字节。以下是没对齐（a）和对齐（b）的情况：



显然对齐后可能会出现储存空间的浪费，所以对齐是在速度和占用空间之间权衡后的选择。

2.3 指令系统的设计和优化

指令系统的设计包括指令功能的设计和指令格式的设计。

2.3.1 指令系统设计的基本原则

1. **完整性**：指令系统的功能全、使用方便。完整性要求指令系统提供的指令足以编制任何解决可解问题的计算机程序。某些面向特定任务的计算机可能提供一些专用指令，这是一种硬件优化。

操作类型	实例
算术和逻辑运算	整数的算术和逻辑操作：加、减、乘、除、与、或等
数据传输	Load，Store
控制	分支、跳转、过程调用和返回、自陷等
系统	操作系统调用、虚拟存储器管理等
浮点	浮点操作：加、减、乘、除、比较等
十进制	十进制加、十进制乘、十进制到字符串的转换等
字符串	字符串移动、字符串比较、字符串搜索等
图形	像素操作、压缩/解压操作等

2. **规整性**：规整性又包括对称性和均匀性。

   对称性指储存单元的使用、操作码的设计是对称的。比如所有通用寄存器要同等对待；如果设置了A-B的操作码，就要设置B-A的操作码。

   均匀性指对应各种不同的操作数类型、字长和数据储存单元，指令的设置要同等对待。比如机器有5种数据表示，4种字长、两种存储单元，就要设置 $5 \times 4 \times 2 = 40$ 种同一操作的指令。

   一般这么做太复杂，不太现实，所以只实现有限的规整性。

3. **正交性**：指指令种各个不同含义的字段，如操作数类型字段、数据类型字段、寻址方式字段等，在编码时应互不相关、相互独立。

4. **高效率**：指指令的执行速度快、使用频度高。比如RISC中大多数指令都能在一个节拍完成，且只设置使用频度高的指令。

5. **兼容性**：主要实现向后兼容。指令系统可以增加新的指令，但不能删除指令或更改指令功能。

2.3.2 控制指令

本书称无条件改变控制流的控制指令为**跳转（jump）指令**，称有条件改变控制流的控制指令为**分支（branch）指令**（频度最高）。除此之外，过程**调用（call）**和**返回（return）**指令也能改变控制流。

分支条件的主要表示方法

名称	检测分支条件的方法	优点	缺点
条件码（CC）	检测由ALU操作设置的一些特殊的位	可以自由设置分支条件	额外硬件
条件寄存器	比较结果放入任何一个通用寄存器中，检测的时候就检测该寄存器	简单	占用一个寄存器
比较与分支一起实现	用一条指令实现比较和分支操作	省一条指令	该指令的操作可能太多，无法在一拍内做完，不利于流水

PC相对寻址

绝大多数情况下，指令显式地给出目标地址。但过程返回指令是个例外。因为可能在代码的很多地方都调用了该过程，而该过程究竟应该返回到哪不能在编译阶段确定。这时就要用到PC相等寻址，由一个偏移量和PC的值相加得出目标地址。除此之外，PC相对寻址还有如下优点：

1. 转移的目标地址一般离当前指令很近，用相当于当前PC值的偏移量来确定目标地址可以减少表示该目标地址所需的位数。

2. PC相对寻址允许代码被装载到主存的任意位置执行，这一特性叫做**位置无关**，能减少程序链接的工作量，且有利于在执行过程中动态链接的程序的实现。

2.3.3 指令操作码的优化

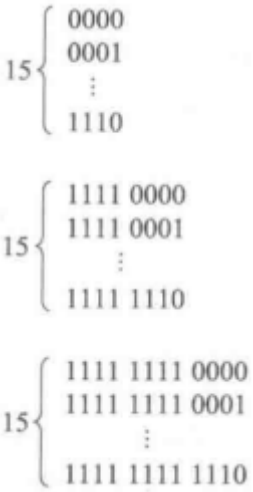
指令一般由操作码和地址码两部分构成，所以指令格式设计也包括对这两部分的设计。不同的指令格式一会影响二进制代码的长度，二会影响译码效率。

1. 等长扩展码

等长扩展码是一种变长的操作码技术。这里"等长"的含义是每次扩展的长度相等。常用的有 4-8-12 位。意思是操作码的长度可以是4位、8位或者12位。这又可以分为两种方法：

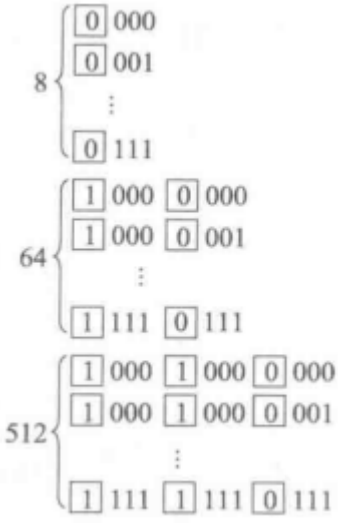
- 15/15/15法

  在4位的16个码点中，使用前15个表示最常用的15种指令，剩下的一个码点用以扩展到下一个4位，二第二个4位的16个码点也是按相同的方法分配的。



- 8/64/512 法

  在4位的16个码点中，只使用 0xxx 这8个表示8种最常用的指令。接着，用 1xxx 0xxx 表示64种指令；接着，用 1xxx 1xxx 0xxx 表示128种指令。



扩展码的设计可以有多种方案，只要满足短码不能是长码的前缀即可。选用哪种方法取决于哪种方法能使平均码长最短。

2. 定长操作码

定长操作码即所有指令的操作码都是统一的长度。

优点：译码复杂度低、速度快

缺点：平均码长较长

由于存储器空间日益增大，定长操作码成为主流，特别是在 RISC 结构的计算机上。

2.4 指令系统的发展和改进

在设计系统时，有两种截然不同的设计策略：**CISC**和**RISC**。CISC旨在增强指令功能，把越来越多的功能交给硬件实现；RISC旨在简化指令系统。

2.4.1 沿CISR方向发展和改进指令系统

1. 面向目标程序增强指令功能

找出频度高、执行时间长的指令或指令串，对指令齐进行硬件加速，而指令串可以用一条新的指令替代。

2. 面向高级语言的优化实现来改进指令系统

改进指令系统，增加对高级语言和编译器的支持，缩小语义差距，能提高计算机系统的性能。

3. 面向操作系统的优化实现来改进指令系统

指令系统对操作系统的支持主要有：

1. 处理机工作状态和访问方式的切换
2. 进程的管理和切换
3. 存储管理和信息保护
4. 进程的同步与互斥、信号灯的管理等

支持操作系统的有些指令属于特权指令，一般用户程序不能使用。

2.4.2 沿RISC方向发展和改进指令系统

CISC指令集结构的缺点：

1. 各种指令的使用频度相差悬殊。28定律：20%的指令使用频度占80%，剩下80%的指令使用频度只占20%。
2. 指令系统庞大，指令多，指令功能复杂，这使得控制器硬件设计很复杂。
    1. 占用了大量的芯片面积，给超大规模集成电路（VLSI）设计造成很大困难
    2. 增加了研制时间和成本，容易造成设计错误
3. 许多指令CPI较大，且不同指令CPI不同，不利于流水，影响执行速度。

而RISC指令集结构了上述缺点，它一般遵循：

1. 指令条数少、功能简单。只设计必要的指令。
2. 采用简单而统一的指令格式，减少寻址方法。
3. 指令的执行在单周期内完成（采用流水线技术后）
4. 采用 Load-Store 结构。
5. 大多数指令都采用硬连线逻辑来实现（而不是微程序）
6. 强调优化编译器，以生成高效的代码
7. 充分利用流水线技术提高性能。

2.5 操作数的类型和大小

2.5.1 数据表示和数据结构

数据表示指计算机硬件能够直接识别、指令系统可以直接调用的数据类型，是由硬件实现的数据类型。

数据结构指软件实现的数据类型。

确定数据表示是一个软硬间直接取舍和折中的问题。

2.5.2 表示操作数类型的方法

1. 由操作码指定。这种方法最常用。
2. 给数据加上标识，由数据本身给出操作数类型。需要在执行时动态检测标识符，开销大，少见。

2.5.3 操作数大小和类型

典型的大小有字节（8位）、半字（16位）、字（32位）、双字（64位）

字符一般用 ASCII 表示，一个字节。

整数一般用二进制补码，大小可以时字节、半字、字等。

浮点数可以分为单精度（字）和双精度（双字）。现在一般都采用 IEEE 754 标准。

压缩十进制或**二进制编码十进制（BCD）**：用4位二进制编码表示数据0~9，将两个十进制数合并到一个字节中存储。如果将十进制数直接用字符串来表示，就叫非压缩十进制。

2.6 MIPS 指令系统结构

MIPS 是典型的 RISC 指令系统。机器字长为32位。

2.6.1 MIPS的寄存器

MIPS有32个32位的通用寄存器R0,R1,...,R31。它们被简称为 GPRs（General-Purpose Registers）或整数寄存器。R0的内容恒为0。R31被用作存放返回指令。

此外，还有32个32位单精度浮点寄存器F0,F1,...,F31。它们被简称为FPRs（Floating-Point Registers）。它们既可以用来存储32个单精度浮点数，又可以用来存储16个双精度浮点数。当存放双精度浮点数时，用F0,F2,...,F30访问。MIPS提供了单精度和双精度操作的指令，还提供了在GPRs和FPRs之间传送数据的指令。

2.6.2 MIPS的数据表示

MIPS支持：

1. 整数：字节、半字、字
2. 浮点数：单精度、双精度

字节或半字的整数在经过零扩展或符号扩展后装入32位的通用寄存器，并按照32位整数的方式运算。

2.6.3 MIPS的数据寻址方式

MIPS支持立即数寻址和偏移寻址。

立即数和偏移量字段都是16位。

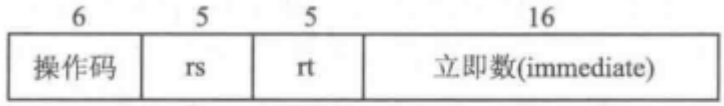
偏移寻址如果偏移量为0，就退化为了寄存器间接寻址；如果寄存器中的值为0（使用R0），就退化为了直接寻址。

寻址方式编码在操作码中。

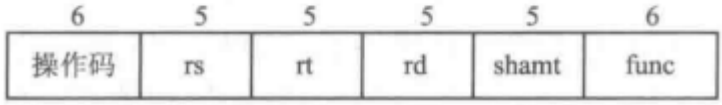
按字节寻址，地址为32位，边界对齐。

2.6.4 MIPS的指令格式

所有指令都是32位。操作码占6位。按不同类型的指令设置3种不同格式。



(a) I类指令



(b) R类指令

(c) J类指令

1. I类指令

Load 和 Store指令、寄存器0立即数型ALU指令、分支指令、寄存器跳转指令、寄存器间接跳转指令。

2. R类指令

寄存器-寄存器型ALU指令、专用寄存器读、写指令、move指令。

3. J类指令

跳转指令、跳转并链接指令、自陷指令、异常返回指令。

跳转并链接会将返回指令（NPC）放入R31。

2.6.5 MIPS的操作

略

3 流水线技术

注：本章讨论的 MIPS 指令集指[[处理器设计与实践实验指导书.pdf]]中的指令集，这其实是MIPS 指令集的一个子集

3.1 概述

3.1.1 基本概念

核心思想：将一个重复的时序过程分解成若干个子过程，每个子过程用专门的部件上执行。不同部件之间并行工作。

显然为了最大化流水线效率，至少需要满足：

1. 各部件执行时间基本一致
2. 输入端能尽量不间断提供任务

3.1.2 分类

1. 单功能流水线和多功能流水线

多功能流水线可以改变部件之间的连接方式从而实现不同的操作。

2. 静态流水线和动态流水线

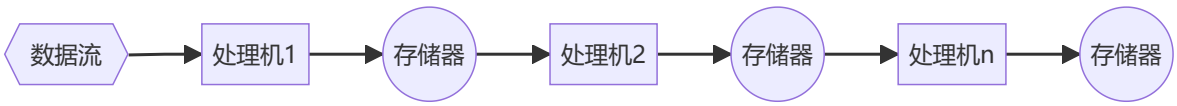
对多功能流水线进一步讨论。如果多功能流水线的流水段中不能同时执行两种不同的操作，就是静态流水线；否则就是动态流水线。如果操作序列是不同操作的交替，那静态流水线的效率就远不如动态流水线。动态流水线的控制会更复杂。

3. 部件级、处理机级和处理器间流水线

部件级流水线又叫运算操作流水线。它将一个运算过程分为多个子过程做成流水。如将浮点运算分为输入、减阶、对阶移位、相加、规格化、输出。

处理机级流水线又叫指令流水线。它将一条指令的执行过程分为多个子过程做成流水。这是本章主要讨论的流水线技术。

处理机间流水线又叫宏流水线，由两个以上的处理机串行地对同一数据流进行处理，每个处理机完成一项任务。这一般属于异构型多处理机系统。



4. 标量流水处理机和向量流水处理机

向量流水处理机就是向量数据表示技术和流水线技术的结合。

5. 线性流水线和非线性流水线

非线性流水线是指流水线中除了串行连接的通路外还有反馈回路。反馈回路的存在允许一个输入在流过流水时多次通过同一个部件。
![[Pasted image 20231002234259.png]]

非线性流水线的一个重要问题是什么时候向流水线引入新的输入，使之不与先前操作的反馈数据在流水线中发生冲突。此即**流水线的调度问题**。

6. 顺序流动流水线和异步流动流水线
异步流动流水线又称乱序流水线。它输出端任务的流出顺序与输入端任务的流入顺序可以不同。

3.2 MIPS的基本流水线

3.2.1 MIPS的一种简单实现

将一条指令的执行分为五段完成：

1. 取指令周期（IF）：从指令储存器中取指令到 IR，PC+4 送 NPC
2. 指令译码/读寄存器周期（ID）：从 IR 中读出操作数和立即数
3. 执行/有效地址计算周期（EX）：视指令类型：进行运算、设置操作码 Cond、计算访存指令的访存地址。
4. 存储器访问/分支完成周期（MEM）：分支的操作：`PC<--Cond?ALUOutput:NPC`
5. 写回周期（WB）：视操作类型将 ALUOutput 或 LMD 写回通用寄存器。

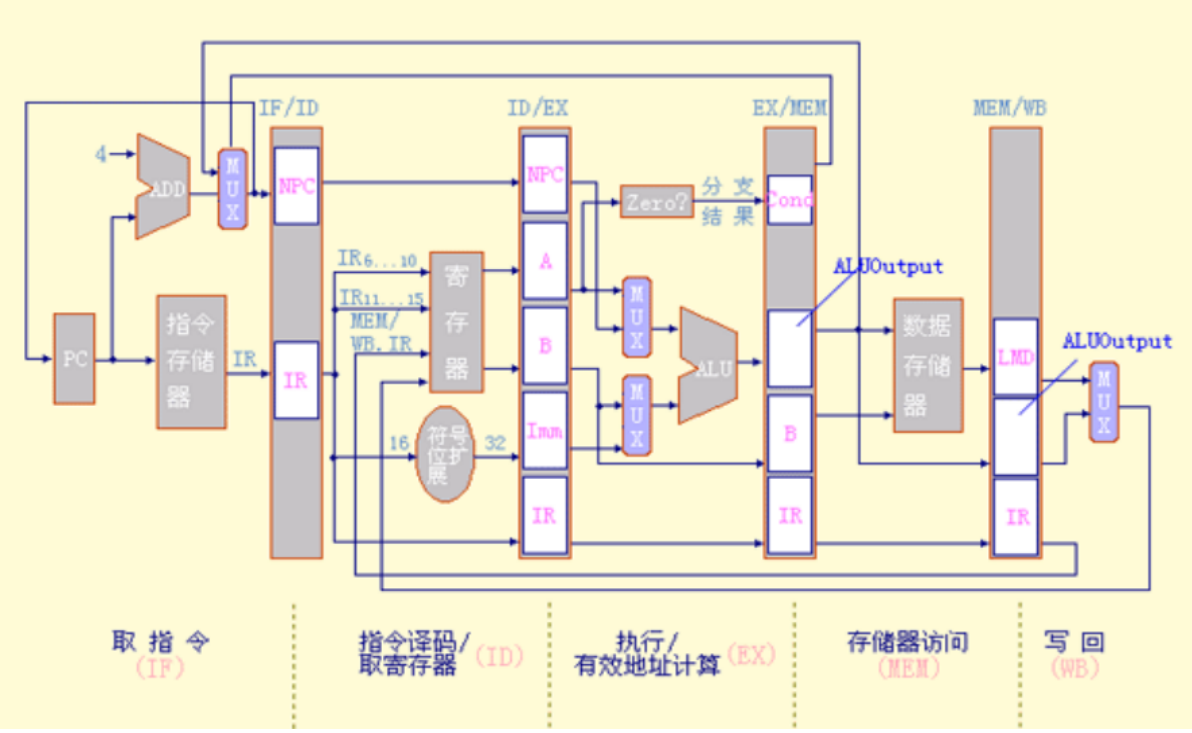
如果不使用流水线，为了提高CPI，ALU 指令可以跳过 MEM 阶段，分支指令可以跳过 WB 阶段。

3.2.2 基本的MIPS流水线

理想情况下MIPS流水线实现的时空图：
![[Pasted image 20231003124704.png]]
然而想要充分发挥流水线的效率，存在许多问题需要解决。这将在[[C3 流水线技术#3.3 流水线中的相关]]详细讨论。

如果在流水过程中仅使用之前非流水数据通路中的寄存器，那么当这些寄存器中保存的临时值还在为流水线中某条指令所用时，就可能被流水线中其他指令所重写。为解决这个问题，必须添加一些 **段间寄存器** 暂存数据和控制信息。段间寄存器的存在会引入额外的延迟。A 段和 B段之间的段寄存器命名为 `A/B.寄存器名`。

数据通路如下：



3.2.3 流水线的性能分析

1. 吞吐率

吞吐率（Throughout Rate） 指流水线在单位时间内完成的指令条数。

1）最大吞吐率 TP_{max}
最大吞吐率 TP_{max} 指流水线在连续流动达到稳定状态后所得到的吞吐率。假设流水线各段时间都是 Δt，则

$$TP_{max} = \frac{1}{\Delta t}$$

假设流水线各段时间不等，则

$$TP_{max} = \frac{1}{max\{\Delta t_i\}}$$

可见最慢的段时流水线性能的瓶颈。为此有两个解决方案：

1. 将成为瓶颈的段进一步细分。
2. 重复设置瓶颈段，使其并行工作。但是这种做法并行段之间的任务分配和同步比较复杂。

2）实际吞吐率
假设流水线有m段，有n个任务流过流水线，实际吞吐率为

$$TP = \frac{n}{(m+n-1)\Delta t}$$

显然只有 $n \gg m$ 时才有 $TP \approx TP_{max}$。

2. 加速比

加速比（Speedup Ratio） 指 m 段流水线的速度与等功能的非流水线的速度之比。

假设流水线有m段，有n个任务流过流水线，加速比为：

$$S = \frac{T_{非流水}}{T_{流水}} = \frac{mn\Delta t}{(m+n-1)\Delta t} = \frac{mn}{m+n-1}$$

显然只有 $n \gg m$ 时才有 $S \approx m$。

3. 效率
效率（efficiency） 指流水线的设备利用率，可以简单地理解为时空图中 n 个任务占用的时空区和 m 个段的总时空区之比。从而效率为：

$$E = \frac{nm\Delta t}{mT_{流水}} = \frac{n}{m+n-1}$$

显然只有 $n \gg m$ 时才有 $E \approx 1$。

3.3 流水线中的相关

相关的万能解决方案就是 **暂停** 。暂停周期一般也被称为流水线气泡或简称为 **气泡** 。当一条指令被暂停，在该指令之后发出的所有指令都要被暂停，在该指令之前的指令继续执行，流水线不会取新的指令。

3.3.1 结构相关

指流水线各段在同一时间竞争使用相同的资源。常用解决方案有流水化功能单元或资源重复。

1. 同一时刻，ALU既需要为一条指令完成减法操作，又需要为另一条指令计算有效地址。这种情况在这个简单的MIPS指令集实现中不会出现。
2. IF 和 MEM 段同时需要访问存储器。如果同时有一个指令缓存和一个数据缓存就可以解决这个冲突。这就是资源重复。
3. ID 和 WB 都要访问通用寄存器。按照[[Verilog进阶#基于 `LUT` 。]]的实现可以解决这个问题。因为这种寄存器实现是周期写、组合逻辑方式读的。这里还有一个问题，如果 ID 段读操作和 WB 段写操作是对同一个寄存器进行的。这就属于数据相关了。

3.3.2 数据相关

3.3.2.1 数据相关的分类

以下讨论考虑流水线中的两条指令 i 和 j，i 在 j 之间进入流水线。以下的数据相关都是对寄存器访问的讨论。

1. **写后读相关（Read After Write，RAW）**：j 的执行要用到 i 的计算结果，但是在流水线中 j 可能在 i 将结果写入之前就进行了读操作，从而读出了错误的值。
2. **写后写相关（Wirte After Write，WAW）**：j 的写入如果在 i 之间发生就会产生 WAW 相关。这在 MIPS 的流水线实现中不会发生。
3. **读后写相关（Write After Read，WAR）**：i 的读取操作如果发生在 j 的写入之前，就会读取 j 写入的结果。这在 MIPS 的流水线实现中不会发生。

如果 ALU 只占4个时钟周期（省去MEM），而MEM 需要两个时钟周期，则可能出现 WAW 相关和 WAR 相关。

LW R1,0(R2)	IF	ID	EX	MEM1	MEM2	WB
ADD R1,R2,R3		IF	ID	EX	WB	

以上情况会出现 WAW 相关，ADD 的 WB 错误地在 LW 的 MEM2 前写入了数据。

读后读（Read After Read，RAR） 不存在数据相关问题。

实际上在访问存储器时也可能出现数据相关问题，这可能时由于访问 Cache 失效产生的。这一章规定出现 Cache 访问失效时直接暂停。不需要暂停的实现方法将在第四章讨论。

总而言之，这一章需要解决的数据相关问题只有 RAW 相关。

3.3.2.2 解决方案

1. 如果需要在同一时刻对一个寄存器进行读操作和写操作，可以规定在上升沿写，下降沿读，这样就能避免此时的 RAW 问题。这种技术在MIPS 中不会出现 WAR 问题。这种技术的实现时简单的。
2. **定向技术**：将一个结果直接传送到需要它的功能单元。
3. 暂停：有些数据相关不能通过定向技术解决，就必须暂停。暂停通过称为 **流水线锁（Pipeline Interlock）** 的功能部件实现。
4. **流水线调度（Pipeling Scheduling）** 或 **指令调度（Instruction Scheduling）**：通过编译器重新组织生成的机器代码顺序（但不改变代码执行结果）来避免相关。

3.3.2.3 暂停和流水线锁

第一个问题是检测哪些指令序列需要暂停。很明显如果在流水线中一个寄存器数据的产生在其使用之后，就必须暂停。在 MIPS 中只有在 LW 指令的下一条指令的执行阶段需要使用 LW 指令加载的寄存器时，才会出现这种情况。

具体地，需要启动流水线锁的情况如下：

ID/EX 的操作码域（ID/EX.IR$_{31...26}$）	**IF/ID的操作码域（IF/ID.IR$_{31...26}$）**	**匹配操作数域**
Load	ALU,分支	ID/EX.IR$_{20...16}$=IF/ID.IR$_{25...21}$
Load	ALU,分支	ID/EX.IR$_{20...16}$=IF/ID.IR$_{20...16}$
Load	Load, Store	ID/EX.IR$_{20...16}$=IF/ID.IR$_{25...21}$

第二个问题时暂停如何实现。当硬件检测到一个相关，只需将 ID/EX 流水线寄存器组中的控制寄存器改为全0即可，全0表示不进行任何操作。另外，还必须暂停向前传送 IF/ID 寄存器组的内容，使得流水线能够保持被暂停的指令。

3.3.2.4 定向技术的实现

源	目的	源的操作码域	目的的操作码域	比较的操作数
EX/MEM.ALUOutput	A	EX/MEM.IR[31:26]=ALU	ID/EX.IR[31:26]=ALU|SW|LW|BNE	EX/MEM.IR[15:11]=ID/EX.IR[25:21]
EX/MEM.ALUOutput	B	EX/MEM.IR[31:26]=ALU	ID/EX.IR[31:26]=ALU|BNE	EX/MEM.IR[15:11]=ID/EX.IR[20:16]
MEM/WB.ALUOutput	A	MEM/WB.IR[31:26]=ALU	ID/EX.IR[31:26]=ALU|SW|LW|BNE	MEM/WB.IR[15:11]=ID/EX.IR[25:21]
MEM/WB.ALUOutput	B	MEM/WB.IR[31:26]=ALU	ID/EX.IR[31:26]=ALU|BNE	MEM/WB.IR[15:11]=ID/EX.IR[20:16]
MEM/WB.LMD	A	MEM/WB.IR[31:26]=LW	ID/EX.IR[31:26]=ALU|SW|LW|BNE	MEM/WB.IR[20:16]=ID/EX.IR[25:21]
MEM/WB.LMD	B	MEM/WB.IR[31:26]=LW	ID/EX.IR[31:26]=ALU|BNE	MEM/WB.IR[20:16]=ID/EX.IR[20:16]

3.3.3 流水线的控制相关

按[[3.2.2]]的数据通路，MEM段才确定分支转移地址。

V1

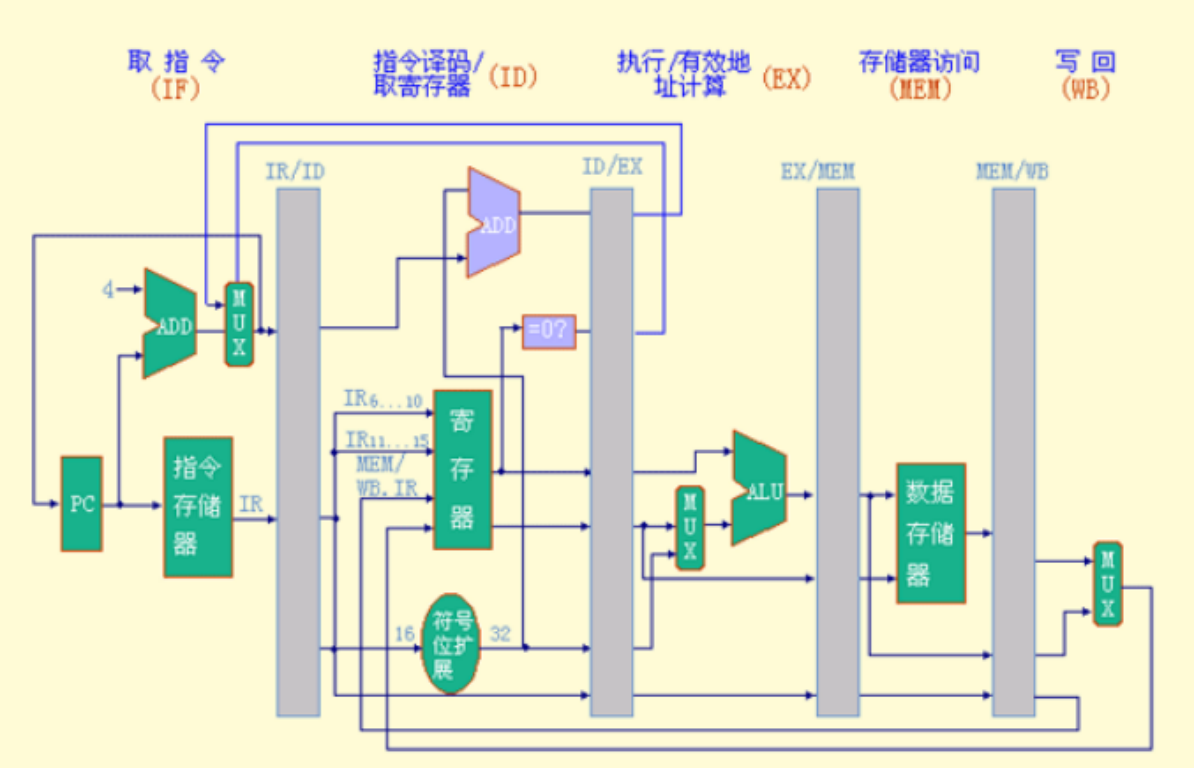
在ID段确定了指令为分支指令后，暂停流水线，直到分支指令在MEM段确定了分支转移地址。

分支指令	IF	ID	EX	MEM	WB					
分支后继指令		**IF**	**stall**	**stall**	IF	ID	EX	MEM	WB	
分支后继指令+1						IF	ID	EX	MEM	WB
分支后继指令+2							IF	ID	EX	MEM
分支后继指令+3								IF	ID	EX
分支后继指令+4									IF	ID
分支后继指令+5										IF

注意，分支后继指令必须重新进行一次IF，因为可能分支转移成功，使用的是不同的pc。这种方法分支指令稳定给流水线带来三个时钟周期的暂停。

V2.1

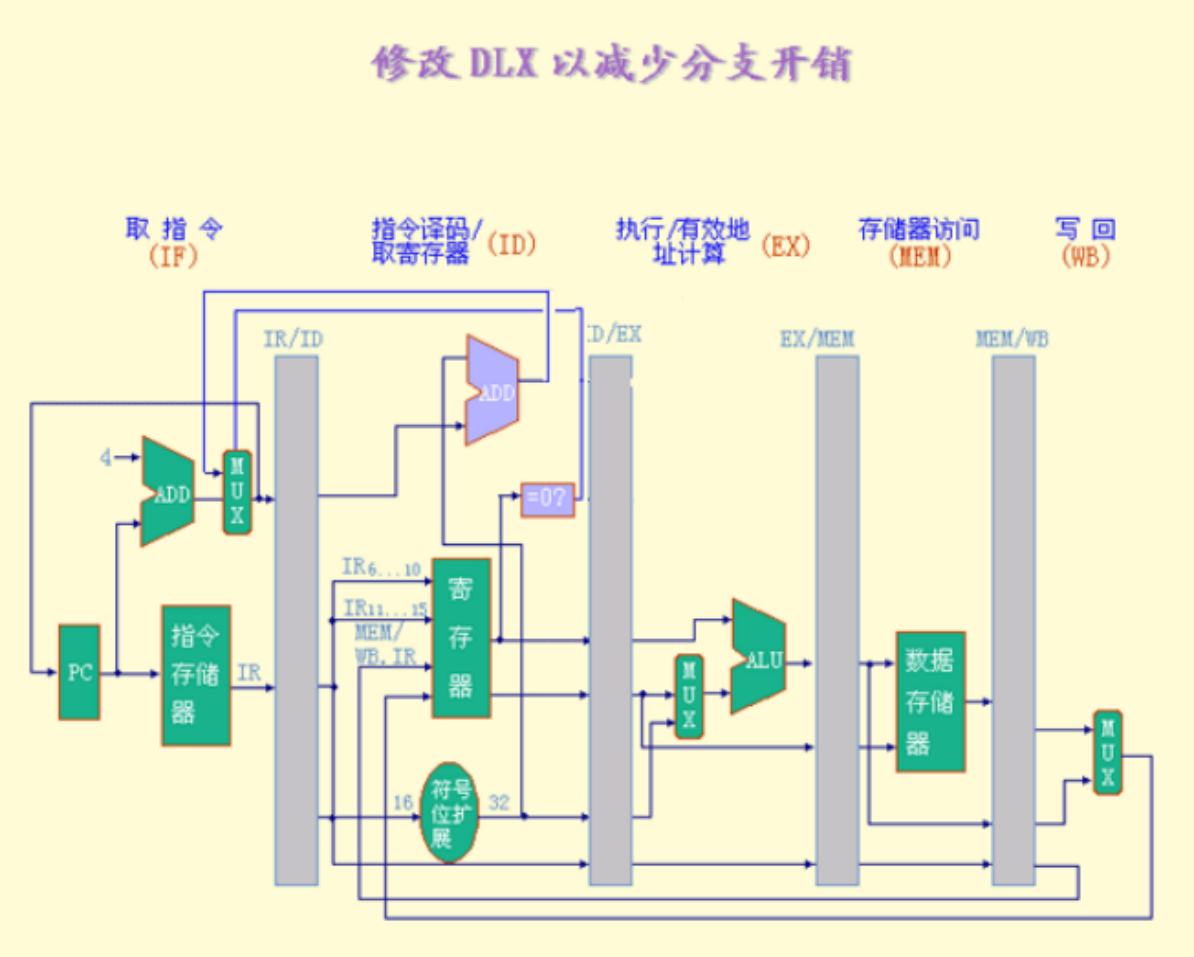
把测试分支寄存器的操作移到ID段完成，从而使得在ID段末就完成分支转移成功与否的检测。此外，将分支转移成功的pc的计算也提前到ID段，为此增设一个加法器。修改后的数据通路图：



这种方法分支指令稳定给流水线带来两个时钟周期的暂停。

V2.2

更激进的做法



这种方法分支指令稳定给流水线带来一个时钟周期的暂停。

这会带来新的冲突：如果分支指令使用了上一条指令的结果，会存在相关，需要暂停一个周期。

V3

预测分支失败，流水线在分支指令的IF执行后照常进行下一条指令的IF，就好像分支一定会失败那样。如果在分支指令的ID发现分支其实是成功的，重新在分支目标地址出IF即可。这样方法只有在分支成功时给流水线带来一个时钟周期的暂停，在分支失败时则不会带来任何暂停。

失败的分支指令	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	WB

成功的分支指令	IF	ID	EX	MEM	WB				
指令 i+1		IF	**idle**	**idle**	**idle**	**idle**			
分支目标			IF	ID	EX	MEM	WB		
分支目标+1				IF	ID	EX	MEM	WB	
分支目标+2					IF	ID	EX	MEM	WB

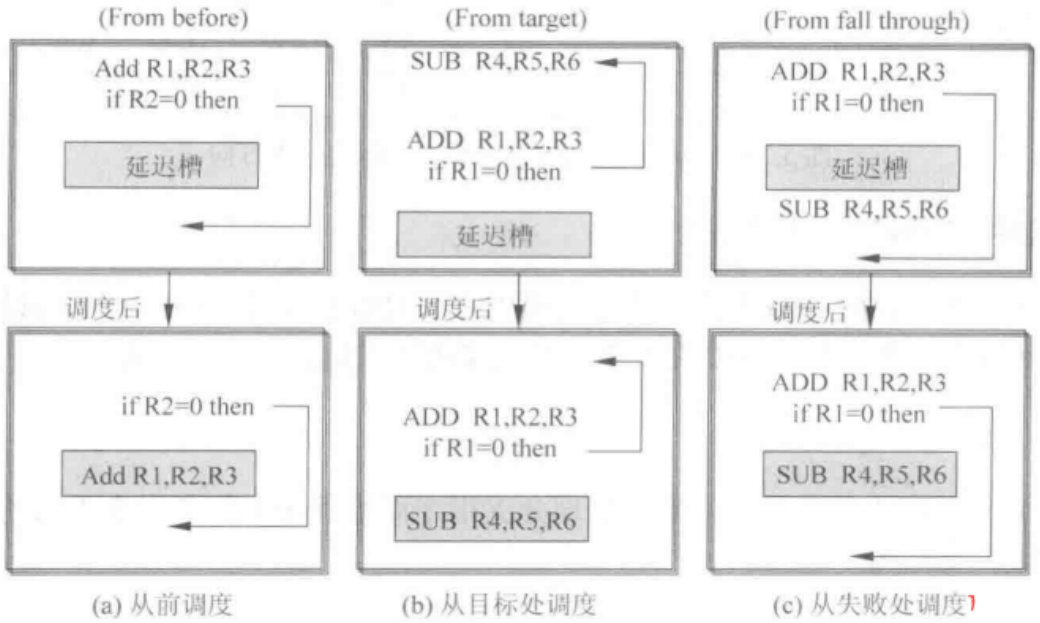
V4

采用**延迟分支技术**，这需要一个所谓的**分支延迟槽**，将分支成功与否都需要执行的指令填入分支延迟槽作为分支指令执行后的下一条指令。这种调度不应该改变指令执行的行为，所以需要考虑相关性的问题。理想情况下充分的利用分支延迟槽能完全消除控制相关，但实际上经常会难以找到可以填充到延迟槽的指令，或者填到延迟槽的指令咋在某种情况下（分支成功或分支失败）需要被作废，这会带来控制相关的暂停。

失败的分支指令	IF	ID	EX	MEM	WB				
延迟分支指令(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	WB

成功的分支指令	IF	ID	EX	MEM	WB				
延迟分支指令(i+1)		IF	ID	EX	MEM	WB			
分支目标指令 j			IF	ID	EX	MEM	WB		
分支目标指令 j+1				IF	ID	EX	MEM	WB	
分支目标指令 j+2					IF	ID	EX	MEM	WB

调度分支延迟指令的三种方法：



调度策略	对调度的要求	对流水线性能改善的影响
从前调度	分支必须不依赖于被调度的指令	总是可以有效提高流水线性能
从目标处调度	如果分支转移失败，必须保证被调度的指令对程序的执行没有影响，可能需要复制被调度指令	分支转移成功时，可以提高流水线性能。但由于复制指令，可能加大程序空间
从失败处调度	如果分支转移成功，必须保证被调度的指令对程序的执行没有影响	分支转移失败时，可以提高流水线性能

总结

1）冻结或排空流水线

2）预测分支失败

3）预测分支成功

这种技术只有在分支目标地址产生于分支结果前的处理器中才能带来好处

4）延迟分支技术

各种处理分支方法的性能

假设流水线的理想CPI为1，那么具有分支损失的流水线加速比为

$$S = \frac{D}{1 + C}$$

D是流水线深度，C是处理分支指令给程序中每条指令带来的平均暂停时钟周期数。

其中

$$C = f \times P_{分支}$$

f是程序中分支指令的频率，$P_{分支}$是处理分支指令的平均暂停周期数。

3.5 向量处理机

3.5.1 向量处理方式和向量处理机

任务：

$$D = A * (B + C)$$

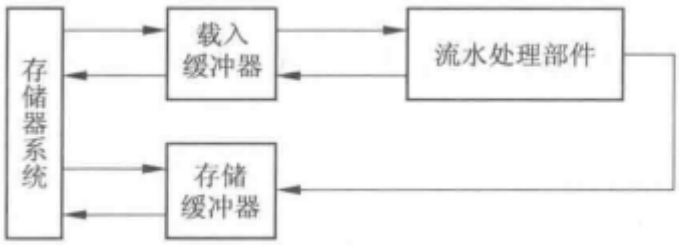
A、B、C是长度为N的向量。

1. 水平处理方式

cpu经典处理。

2. 垂直处理方式

将整个向量按相同运算处理。向量长度N不受限制。这种方式使用存储器-存储器型操作的运算流水线，这对存储系统和处理机之间的通信带宽要求高。

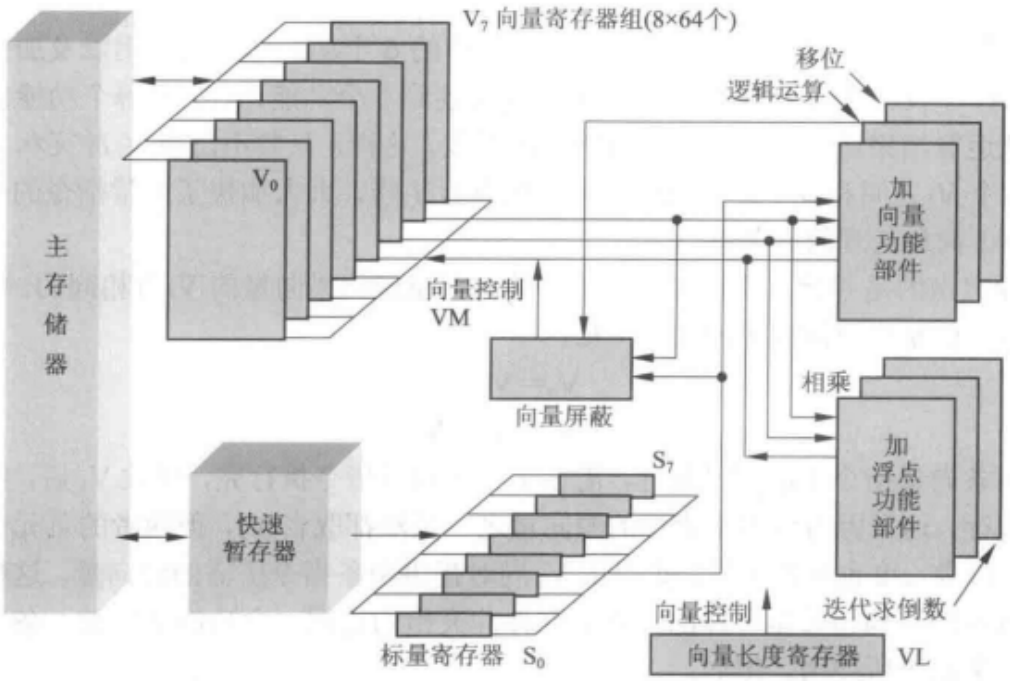


3. 分组处理方式

将长度为N的向量分层s+1组，组内按垂直方式并行，组间按水平方式串行。其中

$$N = sn + r$$

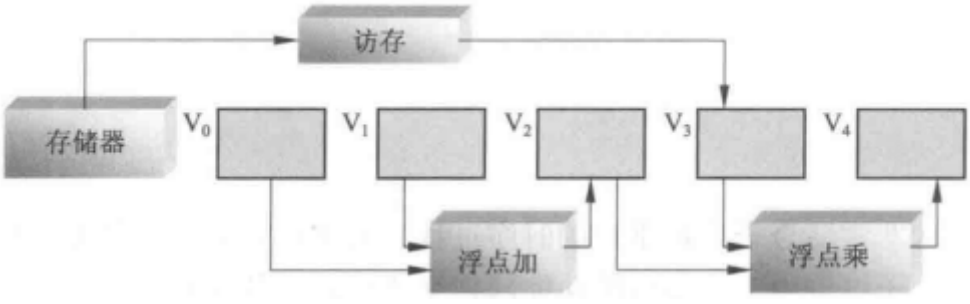
可设置长度为n的向量寄存器，构成寄存器-寄存器型操作的运算流水线。



向量处理机比起MIPS，更常用MFLOPS作为性能的量化评价单位。

如无冲突，不同功能部件可并行处理不同向量寄存器的运算。冲突分为源向量冲突和功能部件冲突。除此之外的相关可以通过链接结构解决。如

$$V_2 \leftarrow V_0 + V_1$$
$$V_4 \leftarrow V_2 * V_3$$



C4 指令级并行

本章使用的浮点流水线的延迟如下：

产生结果指令	使用结果指令	延迟时钟周期数
浮点计算	另外的浮点计算	3
浮点计算	浮点数据存操作（SD）	2
浮点数据取操作（LD）	浮点计算	1
浮点数据取操作（LD）	浮点数据存操作（SD）	0

取数之后可以通过专用通路送到数据存部件，所以延迟为0。

如果分支指令使用上一条指令的结果作为分支条件，要暂停一周期；有一个周期的分支延迟槽。

4.1 指令级并行的概念

流水线的实际CPI公式：

$$CPI_{流水线} = CPI_{理想} + 停顿_{结构相关} + 停顿_{写后读} + 停顿_{读后写} + 停顿_{写后写} + 停顿_{控制相关}$$

4.1.1 指令调度和循环展开

指令调度：改变指令在程序中的位置(但不能改变程序的运行结果)，将相关指令之间的距离加大到不小于指令执行延迟的时钟数。

循环展开：展开循环，这可以减少循环控制语句的数量，并且有利于指令调度。

指令调度和循环展开一般一起用。

```
for(i=1;i<=1000;i++)
{
    x[i]+=s;
}
```

汇编代码如下，R1用作循环计数器，F2保存常数s

```
loop:   LD      F0,0(R1)    ;读取x[i]
        ADDD    F4,F0,F2    ;x[i]+=s
        SD      0(R1),F4    ;保存计算结果
        SUBI    R1,R1,#8    ;i-=1
        BNEZ    R1,loop     ;循环控制
```

注意，汇编代码语义上是`i=1000;i>=1;i--`。

程序执行的实际时钟情况：

```
# V1
loop:   LD      F0,0(R1)    ;
        noop                ;load 后的浮点要延迟一个周期
        ADDD    F4,F0,F2    ;
        noop                ;浮点后的SD要延迟两个周期
        noop
        SD      0(R1),F4    ;
        SUBI    R1,R1,#8    ;
        noop                ;分支指令使用了上一条指令的结果
        BNEZ    R1,loop     ;
        noop                ;分支延迟槽（未装入指令）
```

共10个周期，5个是暂停。

进行指令调度：

```
# V2
loop:   LD      F0,0(R1)    ;
        SUBI    R1,R1,#8    ;
        ADDD    F4,F0,F2    ;
        noop                ;浮点后的SD要延迟两个周期
        BNEZ    R1,loop     ;
        SD      8(R1),F4    ;
```

调度后共6个周期，只有1个是暂停。

注意这里SD的偏移量为8而非0，因为R1被提前减8了。

对V1进行循环展开：

```
# V3
loop:   # 第一个循环体
        LD      F0,0(R1)    ;
        noop                ;
        ADDD    F4,F0,F2    ;
        noop                ;
        noop
        SD      0(R1),F4    ;
        # 第二个
        LD      F6,-8(R1)   ;
        noop                ;
        ADDD    F8,F6,F2    ;
        noop                ;
        noop                ;
        SD      -8(R1),F8   ;
        # 第三个
        LD      F10,-16(R1) ;
        noop                ;
        ADDD    F12,F0,F2   ;
        noop                ;
        noop                ;
        SD      -16(R1),F8  ;
```

```
        # 第四个
        LD      F14,-24(R1) ;
        noop                ;
        ADDD    F16,F14,F2  ;
        noop                ;
        noop                ;
        SD      -24(R1),F8  ;
        # 循环控制
        SUBI    R1,R1,#32   ;
        noop                ;
        BNEZ    R1,loop     ;
        noop                ;
```

每个循环体6个周期，循环控制4个周期，共4*6+4=28个周期，平均每个元素用7个周期。

注意：1. 每个循环体使用的是不同寄存器，这里面有个寄存器分配的问题；2. 每个循环的LD和SD的偏移量不同，最后的循环控制语句的偏移量也要相应修改。

循环展开+指令调度：

```
# v4
loop:   LD      F0,0(R1)    ;
        LD      F6,-8(R1)   ;
        LD      F10,-16(R1) ;
        LD      F14,-24(R1) ;
        ADDD    F4,F0,F2    ;
        ADDD    F8,F6,F2    ;
        ADDD    F12,F0,F2   ;
        ADDD    F16,F14,F2  ;
        SD      0(R1),F4    ;
        SD      -8(R1),F8   ;
        SUBI    R1,R1,#32   ;
        SD      -16(R1),F8  ;
        BNEZ    R1,loop     ;
        SD      8(R1),F8    ;
```

这个调用的逻辑也很简单。至此消除了所有暂停。

每个循环体3个周期，循环控制2个周期，共3*4+2=14个周期，平均每个元素用3.5个周期。

4.1.2 相关性

这里对[[C3 运输层]]的数据相关又重新定义了一遍。。。

1. 数据相关：即RAW。补充两点：

RAW具有传递性。

注意，对存储器上的数据相关的判断是软件判断是困难的，因为可能通过不同方式访问同一个存储器（比如0(R1)和8(R3)指向相同的存储器地址）。存储器相关本书没有细致讨论。

2. 名相关

分两种：

1. 反相关：即WAR。

2. 输出相关：即WAW。

改变指令中寄存器的名就可以消除名相关。这就是**换名技术**。

3. 控制相关

4.2 指令的动态调度

静态调度是编译器在编译时对指令进行的调度，**动态调度**是硬件在指令执行时进行的调度。动态调度并不能真正消除数据相关，但能在出现数据相关时尽量避免处理器空转。

4.2.1 动态调度的原理

在顺序流出的流水线中，如果指令i和j相关，那么j要等i，而j后面的指令，即使与i、j不存在相关性，也会被阻塞。

乱序流水线就是为了解决这个问题。

为了允许乱序执行，将基本流水线的译码阶段再分为两个阶段：

1. **流出（Issue，IS）**：指令译码，检查是否存在结构阻塞。
2. **读操作数（Read Operands，RO）**：当没有数据相关引发的阻塞时就读操作数。

指令流出之前先被取至指令队列中，一旦满足流出条件，指令就从队列中流出。这意味着取指段和流出段之间需要有一个指令缓存。

在动态调度流水线中，，但在读操作数解决允许乱序执行。

乱序执行的一大难点是异常处理。这个问题在后面会通过前瞻技术解决。

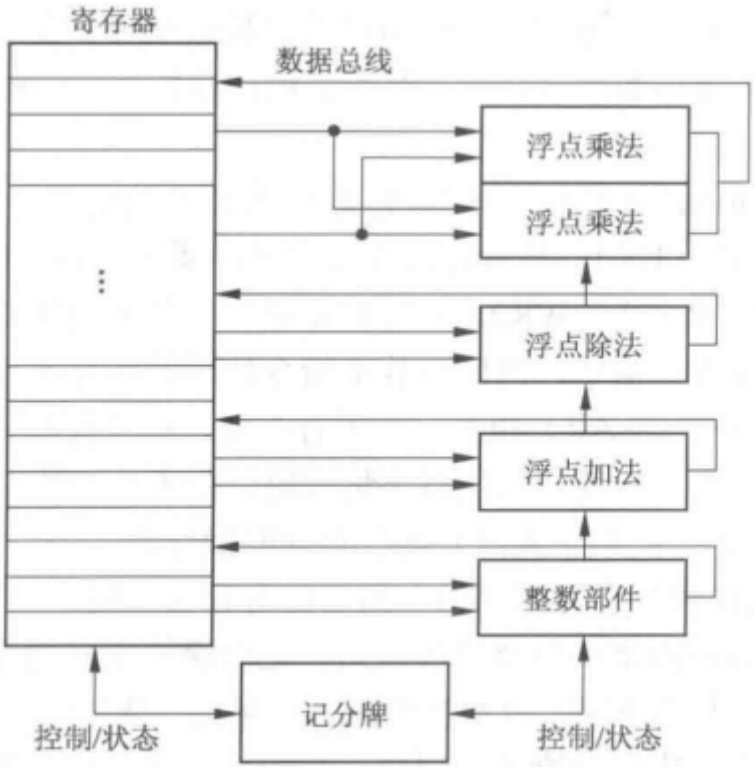
4.2.2 动态调度算法之一：记分牌

1. 流水线的四段

先讨论读后写相关。如果不同指令所用周期数不同，或者指令是乱序的，就可能存在读后写相关引起的阻塞。

记分牌技术的目标是在资源充足时，尽可能早地执行没有数据阻塞的指令。如果某条指令由于相关而被暂停，但后面的指令与流水线正在执行和被暂停的指令无相关，那么后面的指令可以继续流出并执行下去。

要发挥指令乱序执行的好处，就必须有多条指令同时处于执行阶段，这就要求有多个功能部件或功能部件流水化或二者兼有。这一章假设处理器采用多个功能部件：4个浮点部件（乘法2，除法1，加法1）、一个整数部件（处理整数运算、访存、分支操作）。在MIPS中，将记分牌技术主要用于浮点部件，其他部件操作延迟小。由于考虑浮点操作，所以不涉及访存，指令的执行可分为流出、读操作数、执行和写结果4段。



1. 流出：所有指令在流出阶段是顺序的。如果本指令所需的功能部件有空闲，且其他正在执行的指令使用的目的寄存器与本指令使用的目的寄存器不同，记分牌就向功能部件流出本指令，并修改记分牌内部的数据。如果存在结构相关或写后写相关指令就不会流出，并且后面的指令也不会流出。流出受阻可能会使取指段和流出段之间的缓存满，此时就要暂停取指。
2. 读操作数：监测操作数是否有效。如果前面已流出的还在运行的指令不对本指令的源操作数进行写操作，或者一个正在工作的功能部件已经完成了对这个寄存器的写操作，那么此操作数有效。这解决了写后读相关。
3. 执行
4. 写回：如果目标寄存器空闲，就将结果写入目标寄存器中，然后释放本指令使用的所有资源。如果前面的某条指令（按顺序流出）还没有读取到操作数，且读的源寄存器和本指令的目的寄存器相同，则暂停写。这解决了读后写相关。

注意：

1. RO和WB段可能由于功能部件到寄存器文件的总线宽度限制出现结构相关
2. 记分牌不采用旁路技术

总结：

阶段	解决的相关
流出	结构相关和写后写相关
读操作数	写后读相关
写回	读后写相关

2. 记分牌具体记录的信息

1. 指令状态表：记录正在执行的各条指令已经进入流水线的哪一段
2. 功能部件状态表：记录各个功能部件的状态。这个表又有9个域：

   1）Busy：指示功能部件是否正在工作

2）Op：功能部件当前执行的操作

3）Fi：目的寄存器编号

4）Fj，Fk：源寄存器编号

5）Qj，Qk：向Fj，Fk中写结果的功能部件

6）Rj，Rk：yes表示Fj/k已经准备就绪且正在使用，no表示Fj/k未准备就绪或已经使用完了

3. 结果寄存器状态表：每个寄存器在表中有一个域，用于记录写入本寄存器的功能部件。如果当前正在运行的功能部件没有需要写入本寄存器的，置空。

3. 例子

```
LD      F6,34(R2)
LD      F2,45(R3)
MULTD   F0,F2,F4
SUBD    F8,F6,F2
DIVD    F10,F0,F6
ADDD    F6,F8,F2
```

当第一条LD指令已经WB结束时，第二条LD指令已经完成MEM。MULTD和LD存在写后读相关，暂停；SUBD和LD存在写后读相关，暂停；DIVD和MULTD存在写后读相关，暂停（可见相关的传递性）；ADDD和SUBD存在结果相关（都需要用加法器），暂停。此外，DIVD和ADDD直接还存在着读后写相关。由是指令状态表可以给出：

指　令	指令状态表			
	IS	RO	EX	WR
LD F6,34(R2)	√	√	√	√
LD F2,45(R3)	√	√	√	
MULTD F0,F2,F4	√			
SUBD F8,F6,F2	√			
DIVD F10,F0,F6	√			
ADDD F6,F8,F2				

注意：第二条LD此时MEM已经完成，但是该表不记录MEM。

功能部件表：

部件名称	功能部件状态表								
	Busy	Op	F*i*	F*j*	F*k*	Q*j*	Q*k*	R*j*	R*k*
整数	yes	LD	F2	R3				no	
乘法 1	yes	MULTD	F0	F2	F4	整数		no	yes
乘法 2	no								
加法	yes	SUBD	F8	F6	F2		整数	yes	no
除法	yes	DIVD	F10	F0	F6	乘法 1		no	yes

寄存器状态表其实是功能部件表Fi这一列的冗余。

	结果寄存器状态表							
	F0	F2	F4	F6	F8	F10	…	F30
部件名称	乘法 1	整数			加法	除法		

4. 具体实现

符号说明

FU：当前指令使用的功能部件

D：当前指令使用的目的寄存器

S1，S2：当前指令使用的源寄存器

Op：当前指令要进行的操作

result：结果寄存器

1. 流出

   1）进入条件

```
not Busy(FU) and not result('D');   // 判断结构阻塞和写后写
```

   2）记分牌内容记录

```
Busy(FU)<-yes;
Op(FU)<-Op;
Fi(FU)<-'D';
Fj(FU)<-'S1';
Fk(Fu)<-'S2';
Qj<-result('S1');
Qk<-result('S2');
Rj<-not Qj;              // 有部件等待产生S1，就将Qj赋为no
Rk<-not Qk;
result('D')<-FU;
```

2. 读操作数

   1）进入条件

```
Rj,Rk;           //源操作数就绪
```

   2）记分牌内容更新

```
Rj<-no;
Rk<-no;
Qj<-no;
Qk<-no;
```

3. 执行

4. 写结果

   1）进入条件

```
// 任何部件要读的寄存器都不等于当前指令准备写的寄存器
forall f (Fj(f) != Fi(FU) or Rj(f)=no) and Fk(f) != Fi(FU) or Rk(f)=no))
```

为什么上述语句能判断任何部件要读的寄存器都不等于当前指令准备写的寄存器呢。

设forall内的语句确定的部件集为I。要读的寄存器不等于当前指令准备写的寄存器的部件集为 J.证明 I=J.

首先,对于已经完成读操作数的部件x,显然有$x \in I \Leftrightarrow x \in J$.

$\forall x \in J$, 如果该部件在等待某个寄存器,如果该部件在等待两个源寄存器值,同样有$Rj(f) = no$且$Rk(f) = no$,从而$x \in I$. 如果该部件只在等待一个寄存器的值, 不失一般性设在等待Fj,那么根据I的定义它等待的寄存器值不会是$Fi(FU)$,从而$x \in I$.

另一方面, $\forall x \in I$,. 如果x没有完成读操作: 对于就绪的寄存器, 不失一般性设在等待Fj, 那么必然有$Fj(f) \neq Fi(FU)$, 从而等待的寄存器不是当前指令准备写的寄存器. 对于未就绪的寄存器，它等待的一定是另外一条指令的目的寄存器, 而另外一条指令的目的寄存器不可能是$Fi(FU)$, 这是因为流水线在流出阶段就会检查写后写相关, 不可能在有一条指令未完成对$Fi(FU)$时当前指令就流入流水线. 从而一定有$x \in J$.

2）记分牌内容更新

```
\forall f (if Qj(f)=FU then Rj(f)<-yes);
\forall f (if Qk(f)=FU then Rk(f)<-yes);
result(Fi(FU))<-0;
busy(FU)=no;
```

5. 尾巴

1. 如果只在一个基本块中动态调度，效果肯定不好。基本块中的指令相关程度很高，而且大多数基本块只有六到七条指令。

4.2.3 动态调度算法之二：Tomasulo算法

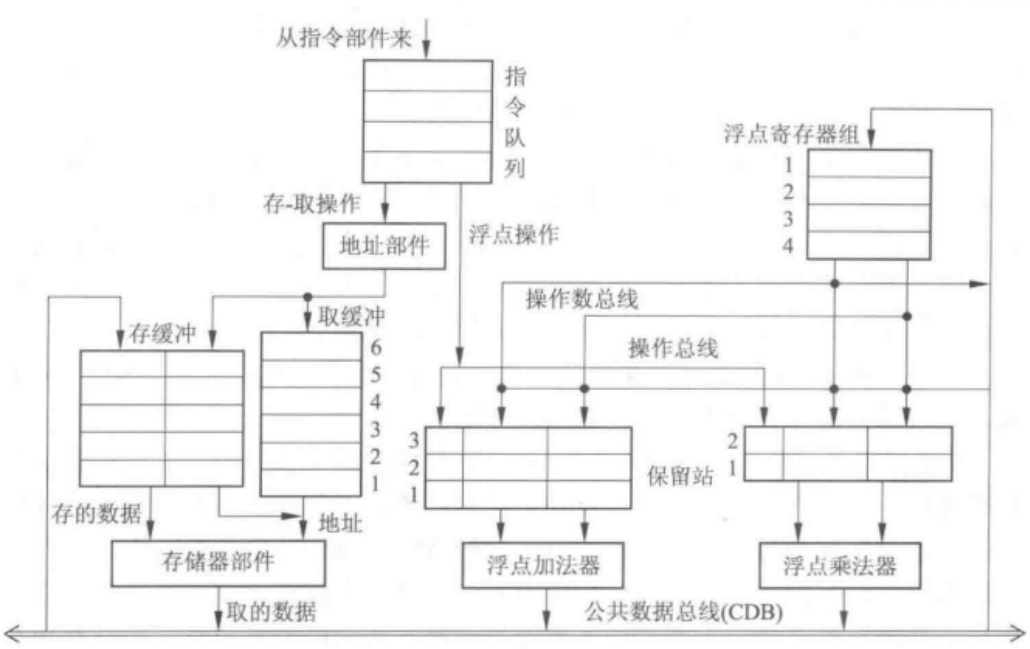
Tomasulo算法将记分牌的关键部分和寄存器换名技术结合。

Tomasulo算法通过**保留站**实现寄存器换名。保留站保存等待流出和正在流出指令所需要的操作数。

1. 流水线的三段

1. 流出：从浮点操作队列中取一条指令。如果是浮点操作并且有空的保留站就流出；如果是访存指令，只要有空的缓冲，指令就流出。当某个操作数还没有就绪，则本保留站中记录产生该操作数的保留站。
2. 执行：如果有某个操作数还未被计算出来，本保留站监视公共数据总线，等待计算的结果。一旦发现公共数据总线有所需的数据，就取到保留站中。当两个操作数都就绪后，本保留站就开始执行指令。
3. 写结果：功能部件计算完毕后，就将结果连同产生本结果的保留站号一起送到公共数据总线上。根据指令流出时所做的寄存器换名记录，所有等待本保留站计算结果的保留站、存缓冲和寄存器将同时从公共数据总线上获得所需的数据。

浮点部件结构如图：



2. 保留站等具体记录的信息

保留站有六个域：

Op：对源操作数S1和S2所进行的操作

Qj，Qk：产生结果的保留站号。等于0表示操作数在Vj或Vk中或者不需要操作数。

Vj，Vk：两个源操作数的值。V和Q域最多只有一个有效。

Busy：标识本保留站和相应的功能部件是否空闲

寄存器有：

Qi：结果要存入本寄存器的保留站号

存缓冲有：

Qj：结果要存入存缓冲的保留站号

Busy：标识存缓冲是否空闲

A：用于记录存操作的存储器地址

Vj：要存入存储器的数据

取缓冲有：

Busy：标识取缓冲是否空闲
A：用于记录取操作的存储器地址

3. 例子

```
LD      F6,34(R2)
LD      F2,45(R3)
MULTD   F0,F2,F4
SUBD    F8,F6,F2
DIVD    F10,F0,F6
ADDD    F6,F8,F2
```

图中所有的指令均已流出，但只有第一条LD指令执行完毕并将结果写到公共数据总线上。

指令		指令状态表		
		流出	执行	写结果
LD	F6,34(R2)	√	√	√
LD	F2,45(F3)	√	√	
MULTD	F0,F2,F4	√		
SUBD	F8,F2,F6	√		
DIVD	F10,F0,F6	√		
ADDD	F6,F8,F2	√		

名称	保留站						
	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	LD					45+Regs[R3]
Add1	yes	SUBD		MEM[34+REGS[R2]]	Load2		
Add2	yes	ADDD			Add1	Load2	
Add3	no						
Mult1	yes	MULTD		REG[F4]	Load2		
Mult2	yes	DIVD		MEM[34+REGS[R2]]	Mult1		

域	寄存器 状态表							
	F0	F2	F4	F6	F8	F10	…	F30
Qi	Mult1	Load2		Add2	Add1	Mult2		

4. 具体实现

符号：

rs、rt：操作数源寄存器号；rd：操作数目的寄存器号；Imm：符号拓展的立即数

Regs：寄存器文件；

r：分配给当前指令的保留站或缓冲

RS：保留站表；RegisterStat：寄存器状态表

1）指令流出

进入条件：

对于浮点操作：有空闲保留站r

对于访存操作：有空闲缓冲r

记录内容：

对于浮点操作：

```
if (RegisterStat[rs].Qi !=0)
{
    RS[r].Qj<-RegisterStat[rs].Qi;
    RS[r].Vj<-=0;
}
else
```

```
{
    RS[r].Qj<-0;
    RS[r].Vj<-Regs[rs];
}

if (RegisterStat[rt].Qi !=0)
{
    RS[r].Qk<-RegisterStat[rt].Qi;
    RS[r].Vk<-=0;
}
else
{
    RS[r].Qk<-0;
    RS[r].Vk<-Regs[rt];
}

RS[r].Busy<-yes;
RS[r].Op<-Op;
RegusterStat[rd].Qi<-r;
```

对于存取操作:

```
if(RegisterStat.Qi != 0)
{
    RS[r].Vj<-0;
    RS[r].Qj<-RegisterStat[rs].Qi;
}
else
{
    RS[r].Vj<-Regs[rs];
    RS[r].Qj<-0;
}
RS[r].Busy<-yes;
rS[r].A<-Imm;
```

对于取操作:

```
RegisterStat[rd].Qi<-r;
```

对于存操作:

```
if(RegisterStat[rt].Qi != 0)
{
    RS[r].Qk<-RegisterStat[rt].Qi;
    RS[r].Vk<-0;
}
else
{
    RS[r].Vk<-Regs[rt];
    RS[r].Qk<-0;
}
```

2）执行

进入条件：

对于浮点操作：

```
(RS[r].Qj=0) and (RS[r].Qk=0)
```

对于存/取操作第一步：

```
(RS[r].Qj=0) and (r到达存/取缓冲队列的头部)
```

3）写结果

进入条件：

对于浮点操作或取操作：保留站r执行结束且公共数据总线空闲

对于存操作：保留站r执行结束且RS[r].Rk=0

记录内容：

对于浮点操作或取操作

```
forall x(if(RegsterStat[x].Qi=r))
{
    Regs[x]<-result;
    RegsterStat[x].Qi<-0;
}
forall x(if(RS[x].Qj)=r)
{
    RS[x].Vj<-result;
    RS[x].Qj<-0;
}
forall x(if(RS[x].Qk)=r)
{
    RS[x].Vk<-result;
    RS[x].Qk<-0;
}
RS[r].Busy<=no;
```

对于存操作：

```
RS[r].Busy<-0;
```

5. 尾巴

一旦有一条分支指令还没有执行完毕，其后的指令是不允许进入执行段的。

Tomasulo算法还可以做到动态存储器地址判别，解决存储器的数据相关问题。

4.3 控制相关的动态解决技术

分支预测的目的是尽可能早猜测分支以后的指令地址，从而避免由控制相关导致的流水线停顿。只有预测并修改 PC 所需的时间小于分支延迟时，分支预测才可能有意义。分支的最终延迟取决于：

1. 流水线的结构
2. 预测的方法（准确度）

3. 预测错误后恢复所采取的策略（影响分支预测不正确时的时间开销）

4.3.1 分支预测缓冲

动态分支预测是一种基于分支操作历史的预测技术。两个关键问题：

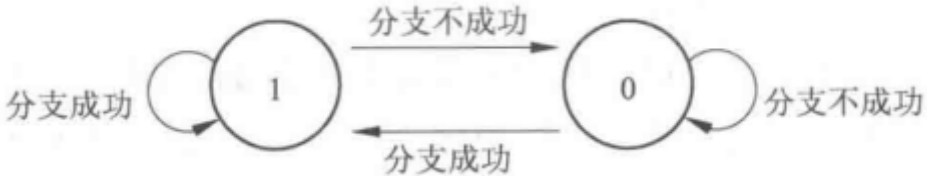
1. 如何记录分支操作的历史
2. 如果基于历史记录预测分支

记录分支操作历史的方法有：

1. 仅仅记录最近一次或几次分支是否成功
2. 记录分支成功的目标地址
3. 结合1和2，同时记录分支是否成功和分支目标地址
4. 记录分支目标地址的一条或若干条指令

最简单的分支预测技术是 **分支预测缓冲技术（Branch Prediction Buffer 或 Branch History Table，BPB 或 BHT）**。它仅仅用一片存储区域记录最近一次或几次分支是否成功。

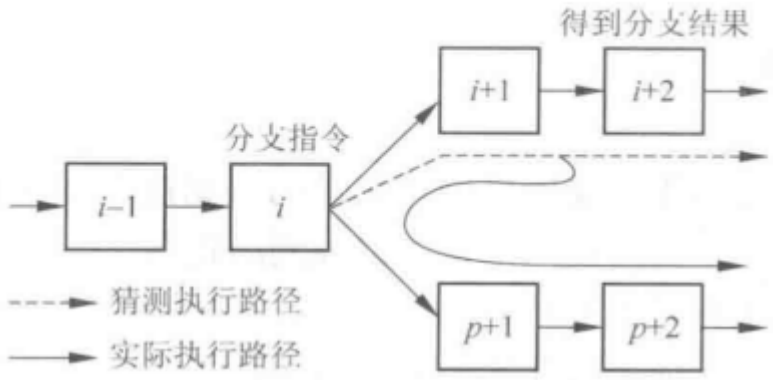
最简单的为一个预测位的 BPB，状态转换图如下：



状态 1 预测分支成功，状态 0 预测分支失败。

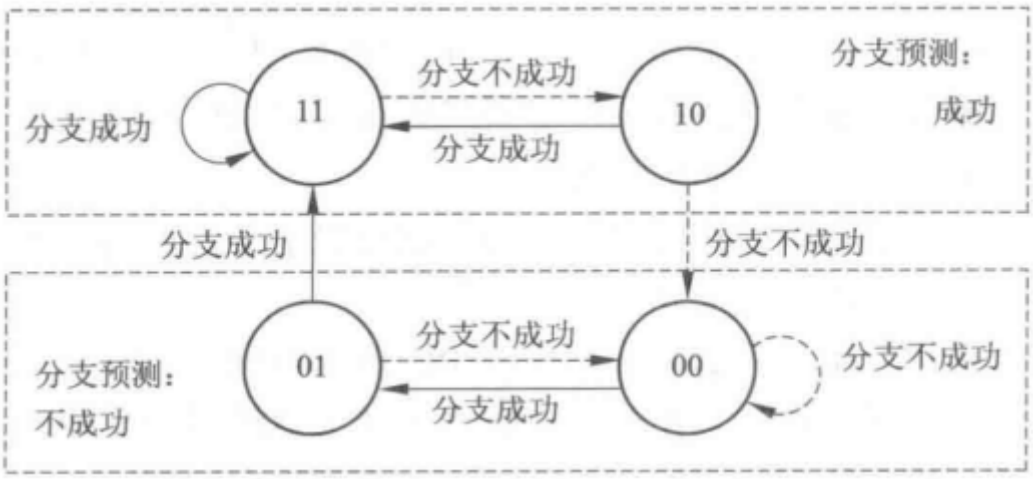
如果预测错误，就修改预测位，并恢复现场，使得程序从分支指令处重新执行。从预测到发现预测错误，机器一般会流出1~2条指令，恢复现场就是作废这1~2条指令。

分支预测不成功的执行过程：



假设一个一层循环执行了 N 次，那么预测正确的次数是 $\frac{N-2}{N}$。第一次预测是错误的，因为上一次循环最后一次分支是不成功的，最后一次预测显然也是错误的。

为提高预测准确率，可以采用两个预测位。再两个预测位的分支预测中，更改对分支的预测必须有两次连续的预测错误（或预测成功）。状态转换图如下：



进一步，可以采用 n 位的分支预测缓冲。采用 n 位计数器，当计数器的值大于 2^{n-1} 时预测下一次分支成功，否则预测下一次分支不成功。当分支成功时计数器的值加一，不成功时减 1 。研究表明，n 位分支预测的性能和两位分支预测差不多，因而大多数处理器都只采用两位分支预测。

4.3.2 分支目标缓冲

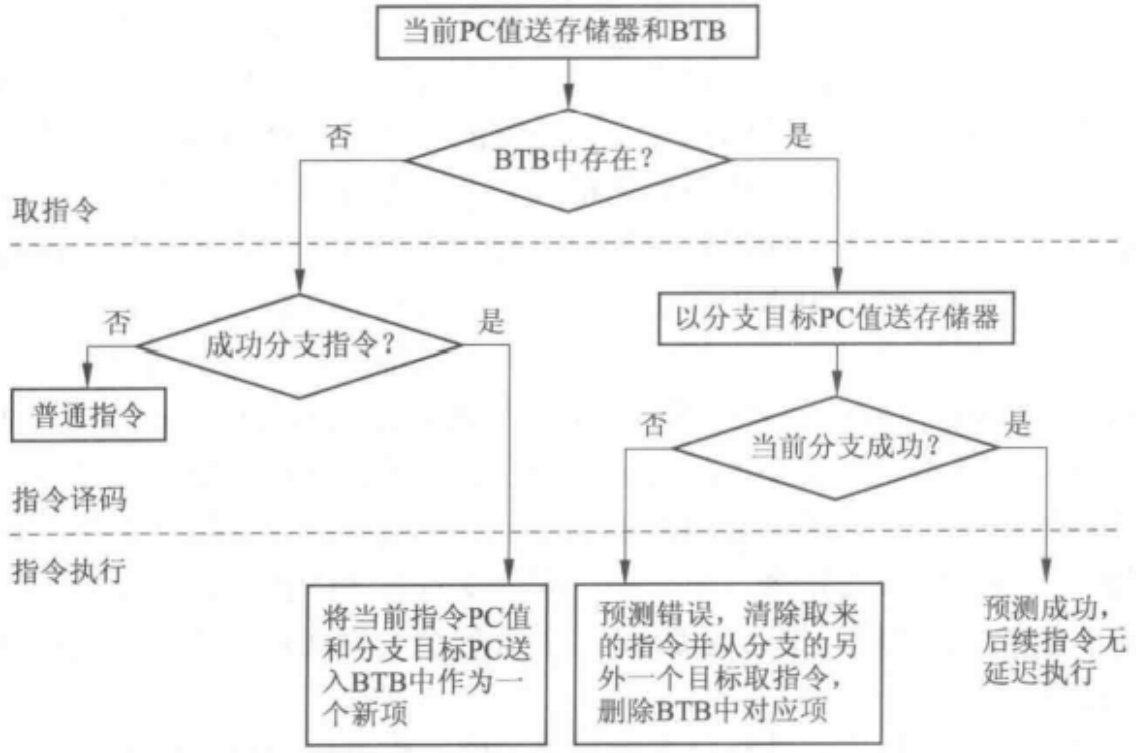
要进一步减少类似 MIPS 结构的流水线的分支延迟，就需要在新 PC 形成之前，即取值阶段（IF）后期就知道在什么地址取下一条指令。因此必须知道还没译码的指令是否是分支指令，如果是，还要知道可能的分支目标指令的地址。如果下一条指令是分支指令而且已知它的目的地址，则分支的开销可以降为0 。

实现：将分支成功的分支指令的地址和它的分支目标地址都放在一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。取指阶段，将指令地址与保存的标识作比较，如果相同，就认为本指令是分支指令，且认为它转移成功，并且它的分支目标地址就是保存在缓冲区的分支目标地址。这个缓冲区就是**分支目标缓冲区（Brachs-Target Buffrt，BTB）**。

- 如果当前指令的地址与缓冲区的标识匹配，那么此指令为上一次分支成功的分支指令，并预测此次分支也成功，且下一条指令的 PC 值在分支目标缓冲的分支目标 PC 域中，因此在本指令译码阶段（ID）开始从预测的分支目标 PC 处取下一条指令。
- 如果没有匹配而指令当前又成功分支，则分支目标地址会在指令译码阶段（ID）末知道，然后将该条指令本身的地址和目的地址加入缓冲区。
- 如果在分支目标缓冲中找到了当前指令地址，而指令当前分支不成功（即预测是错误的），则将此项从缓冲区中删去。

如果是分支指令并且预测正确，将不会有任何延时；如果预测错误，则取错误的指令会浪费一个时钟周期，下一次时钟周期可以重新取出正确的指令。如果转移分支指令缓冲不命中，耗费延迟的大小取决于指令是否成功转移。

实际实现中，不命中或预测错误时的延迟会更大，因为分支目标缓冲必须更新。解决预测错误或不命中的延迟是一个具有挑战性的问题，因为通常情况下，重新缓冲区时将停止流水线。



4.3.3 基于硬件的前瞻执行

1 . 简介

将Tomusola算法加以扩展就可支持前瞻执行。

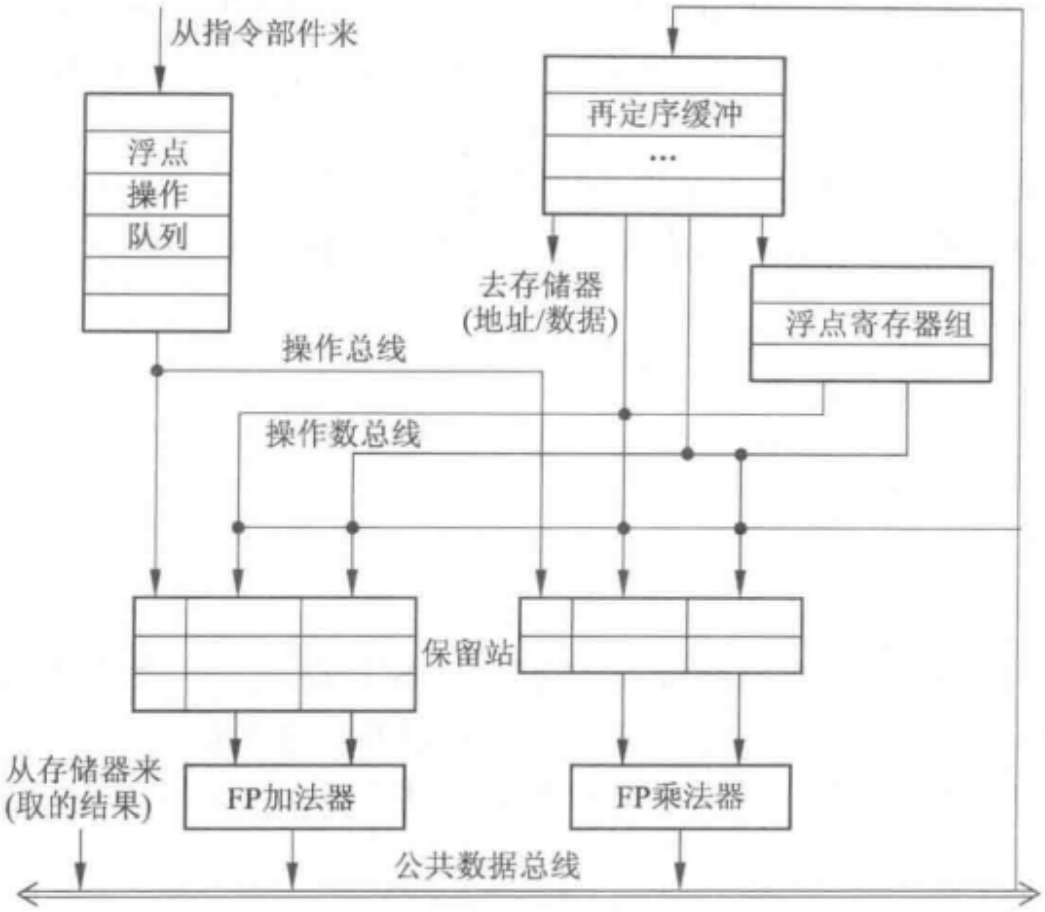
指令的确认：前瞻执行的结果供给其他指令使用的时候要明确这些结果不是实际完成的结果，使用这些结果的其他指令其实也是在前瞻执行。这些前瞻执行的指令产生的结果要一直等到指令处于非前瞻执行状态时才能确定为最终结果，才允许最终写到寄存器或存储器中去。将指令由前瞻转化为非前瞻这一步骤加到执行阶段后，称为指令的确认。

实现前瞻的关键思想是：允许指令乱序执行，但必须顺序确认。只有确认以后的结果才是最终的结果，从而避免不可恢复的行为，如更新机器状态或执行过程发生异常。在简单的单流出MPS流水线中，将写结果阶段放在流水线的最后一段，保证在顺序地检测完指令可能引发的异常情况之后再确认指令结果。加入前瞻后，需要将指令的执行和指令的确认区分开，允许指令在确认之前早就执行完毕。所以，加入指令确认阶段需要一套额外的硬件缓冲，来保存那些执行完毕但未经确认的指令及其结果。这种硬件的缓冲称为**再定序缓冲(Reorder Buffer,ROB)**,它同时还用来在前瞻执行的指令之间传送结果。

再定序缓冲和Tomasulo算法中的保留站一样，提供了额外的虚拟寄存器，扩充了寄存器的容量。再定序缓冲保存指令执行完毕到指令得到确认之间的所有指令及其结果，所以再定序缓冲像Tomasulo算法中的保留站一样是后续指令操作数的来源之一。主要的不同点是在Tomasulo算法中，一旦指令结果写到目的寄存器，下面的指令就会从寄存器文件中得到数据。而对于前瞻执行，直到指令确认后，即明确地知道指令应该执行以后，才最终更新寄存器文件，因此在指令执行完毕和确认之间的这段时间里，由再定序缓冲提供所有其他指令需要的作为操作数的数据。再定序缓冲和Tomasulo算法中的存操作缓冲不一样，为了简单起见可以将存缓冲的功能集成到再定序缓冲。因为结果在写入寄存器之前由再定序缓冲保存，所以再定序缓冲区还可以取代取缓冲区。

2. 再定序缓冲

Tomasulo算法+前瞻执行的部件结构：



再定序缓冲的每个项包含五个域：

1）Busy：该项是否空闲

2）指令的类型。包括分支、存操作、寄存器操作（取操作和ALU操作）

3）该项缓存的指令目前处于的阶段

4）目的地址：寄存器地址（寄存器操作）或存储器地址（存操作）

5）值域：指令前瞻执行的结果

再定序缓冲取代了存储器的取和存缓冲。

每条指令在确认之前在再定序缓冲中都占有一项，所以用再定序缓冲项的编号而非保留站的编号来标识。即保留站登记的是分配给该指令的再定序缓冲的编号。

3. 指令执行过程

1）流出

2）执行

3）写结果

4）确认

4. 尾巴

通过再定序技术，可以在进行精确的异常处理的同时进行动态指令调度。

4.4 多指令流出技术

超标量既可以静态调度，又可以动态调度。**超长指令字**一般只通过编译技术静态调度。**超流水**是将每个功能部件流水化。

4.4.1 静态超标量技术

考虑一条整数指令流水一条浮点流水的两路超标量处理器。

允许以下代码；

```
loop:   LD      F0,0(R1)    ;读取x[i]
        ADDD    F4,F0,F2    ;x[i]+=s
        SD      0(R1),F4    ;保存计算结果
        SUBI    R1,R1,#8    ;i-=1
        BNEZ    R1,loop     ;循环控制
```

经过5次循环展开，并经过调度的指令序列如下：

	整数指令		浮点指令		时钟周期
Loop:	LD	F0(R1)			1
	LD	F6, –8(R1)			2
	LD	F10, –16(R1)	ADDD	F4,F0,F2	3
	LD	F14, –24(R1)	ADDD	F8,F6,F2	4
	LD	F18, –32(R1)	ADDD	F12,F10,F2	5
	SD	0(R1), F4	ADDD	F16,F14,F2	6
	SD	–8(R1),F8	ADDD	F20,F18,F2	7
	SD	–16(R1),F12			8
	SD	–24(R1),F16			9
	SUBI	R1,R1,#40			10
	BNEZ	R1,Loop			11
	SD	8(R1),F20			12

每次循环12个时钟周期，平均每个元素2.4个时钟周期。

4.4.2 动态超标量技术

多条指令流出技术，也可用记分牌技术或Tomasulo算法进行动态调度处理。下面拓展Tomasulo算法支持两路超标量。区别是要求指令按顺序流向保留站，否则太复杂。

4.4.3 超长指令字技术

超长指令字（Very Long Instruction Word，VLIW）采用多个独立的功能部件，但它并不是将多条指令流出到各个功能部件，而是将多条指令的操作组装成固定格式的指令包，形成一条非常长的指令。每个操作字段称为一个**操作槽**。

在超长指令字处理器上，选择同时可流出的多条指令及其相关处理的任务都由编译器完成，所以超长指令字机器可以节省硬件。

还是一样的向量与标量加代码，忽略分支指令的延迟槽，指令序列如下：

访存指令 1	访存指令 2	浮点指令 1	浮点指令 2	整数/转移指令
LD F0,0(R1)	LD F6, –8(R1)			
LD F10,–16(R1)	LD F14, –24(R1)			
LD F18, –32(R1)		ADDD F4,F0,F2	ADDD F8,F6,F2	
		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2		
SD 0(R1),F4	SD –8(R1),F8			
SD –16(R1),F12	SD –24(R1),F16			SUBI R1,R1,#40
SD 8(R1),F20				BNEZ R1,Loop

每次循环8个时钟周期，平均处理每个元素1.6个时钟周期。可见功能部件使用效率不高。

4.4.4 多流出处理器受到的限制

1）程序内在的指令级并行性有限。

粗略地，所需要的无关指令数等于流水线深度乘以可并行工作的功能部件数。

2）硬件实现的困难

3）超标量和超长指令字处理器固有的技术限制。

5 多级存储层次

5.1 存储器的层次结构

5.1.1 多级存储系统

用户对存储器的三大需求：容量大、速度快、价格低。价格低主要是要求每位价格低。这是因为：一方面，应用程序越来越大；另一方面，CPU性能提升得很快。

考虑具体实现技术，这三个需求很难都满足。解决方案是采用多级存储器技术，构成多级存储系统。

常见的多级存储模型是：寄存器——Cache——主存——辅存。分别用 M1, M2, M3,M4 表示。从 M1 到 M4，离 CPU 越来越远，速度越来越慢，容量越来越大，每位价格越来越低。

Cache 主要由 SRAM 实现，主存主要由 DRAM 实现。

在这种存储模型下，对存储器的需求可以表述为：存储系统在速度上接近 M1，而在 容量和每位价格上接近 M4。

之所以能做到这一点，是因为 CPU 对代码和数据的访问都遵循局部性原理。即：时间局部性，当前时刻访问的代码很可能不久后会再次访问；空间局部性，接下来访问的代码很可能在当前代码的附近。

在多级存储系统中，任何一层存储器中的数据一般都是其下一级（离 CPU 更远的一级）存储器中数据的子集。CPU 访存会从上倒下访问每一级，直到找到需要的数据。每次数据被访问都会被调入上一级存储层次。（通常是调入包含该数据的整个块或页面）

5.1.2 存储层次的性能参数

对于只有 M1 和 M2 两个存储层次的多级存储系统，假设 M1 的容量、访问时间、每位价格分别为 S_1、T_{A1}、C_1，M2 的分别为 S_2、T_{A2}、C_2 。

平均每位价格 C

$$C = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}$$

当 $S_1 \ll S_2$ 时 $C \approx C_2$。

命中率 H

这里指 CPU 访存时在 M1 中找到所需信息的概率。假设运行一组代表性程序，访问 M1 和 M2 的次数分别为 N1 和 N2 。则

$$H = \frac{N_1}{N_1 + N_2}$$

失效率 F 定义为

$$F = 1 - H$$

平均访问时间 T_A

若访问在 M1 命中，则访问时间为 T_{A1}，称之为命中时间（HitTime）。若访问在 M1 不命中，则需要向 M2 访存。大多数存储系统会把 M2 中包含所请求的字的数据块传送到 M1，然后 CPU 再访问这个字。假设一个数据块从 M2 传送到 M1 用时为 T_B，则不命中访问时间为 $T_{A1} + T_{A2} + T_B$ 。定义 $T_M = T_{A2} + T_B$ 为失效开销。失效开销是从向 M2 发出访问请求到把整个数据块调入 M1 所需的时间。显然有

$$T_A = T_{A1} + FT_M$$

5.1.3 两种存储层次关系

重点研究Cache——主存层次关系和主存——辅存层次关系。前者主要解决存储体系速度问题，后者主要解决存储体系容量问题。

5.1.4 存储层次的 4 个问题

1. 映像规则：当把一个数据块调到上一级存储层次中时，应该放到哪个位置上
2. 查找算法：如何在不同层次找到数据块
3. 替换算法：当上一级存储层次已满时，新调入的数据块应该替换掉哪个旧的块
4. 写策略：如何在多级存储系统中写入新数据并保持数据的同步

5.2 Cache 基本知识

5.2.1 映像规则

主存向 Cache 的映射是一个多对一或多对多的映射。为正确在 Cache 中找到缓存的主存的副本，对主存的地址作如下解释：

主存的块内位移就是 Cache 的块内位移。显然整个快地址空间对应的所有块无法都缓存在 Cache 中。

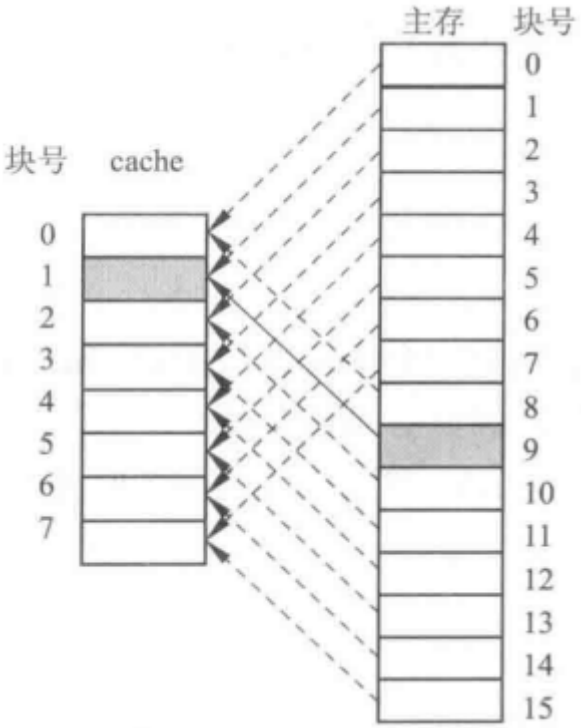
主要有三种映射规则：

1. 直接映射

最简单。假设 Cache 一共有 M 块。则块地址为 i 的主存块会被分配到块地址为 j 的 Cache块，满足：

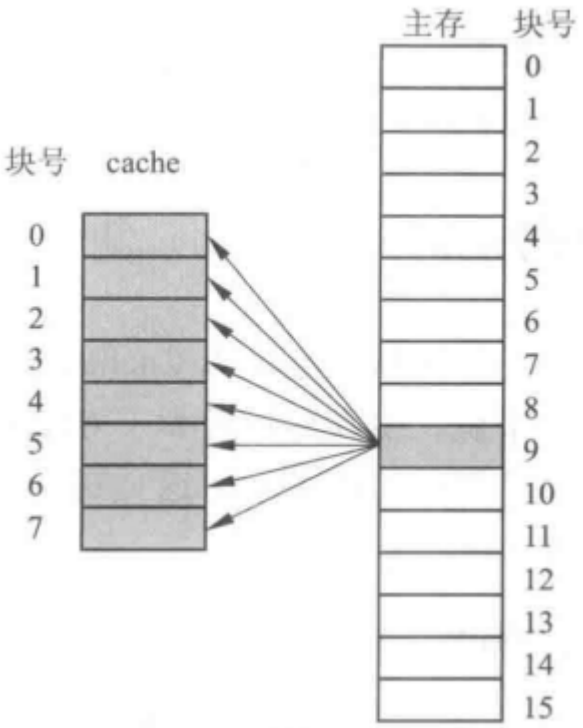
$$j = i \bmod M$$

设 $M = 2^m$ 显然 Cache 块地址 j 实际就是主存块地址 i 的低 m 位。



2. 全相联映射

也很简单。主存的任意块可以放置到 Cache 的任意块。



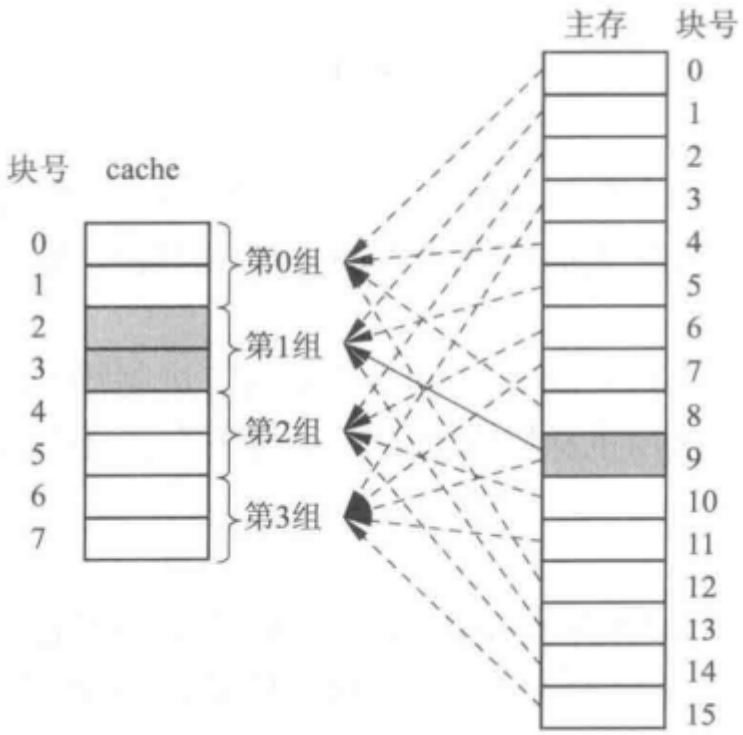
3. 组相联映射

最麻烦。有点像直接映射和全相联映射的折中。Cache 被分为若干组，每组若干块。假设 Cache 共 $M = 2^m$ 块，被分为了 $G = 2^g$ 组，那么每组有 $N = \frac{M}{G} = 2^{m-g}$ 块。主存块地址 i 到 Cache 块地址 j 的映射为

$$j = i \, mod \, G$$

显然 Cache 块地址 j 实际就是主存块地址 i 的低 g 位。

每个内存块可以选择缓存到组内的任意空闲块。



每组有 N 个块的组相联映射称为 N 路组相联映射。直接映射就是 1 路组相联映射，全相联映射就是 M 路组相联映射。

5.2.2 查找算法

进一步将主存块地址切分为**索引**（低 g 位）和**标识**（高 n 位）。对于直接映射，g=m，对于全相联映射，g=0 。索引对对应 Cache 的组号。标识区分那些会被映射到同一组的主存块。

Cache 的每个块有一个标识，共 M 个标识，存放在目录表中。一般还在目录表中给每一项设置一个有效位。

5.2.3 替换算法

直接映射因为每个主存块只有唯一的 Cache 块可以选择，所以必定是新块替换该位置的旧块。

组相联映射和全相联映射可以选择如下替换算法之一：

1. 随机法

见名知意。

2. 先进先出

先进先出（FIFO）选择最早调入的块最为被替换的块。实现简单，但可能并不能正确反映局部性原理。因为最先进入的块可能正好是最常要用到的块。

3. 最近最少使用

最近最少使用（LRU）选择近期最久没有被访问过的块作为被替换的块。这种方法实现时需要记录一段时间内各块的被访问次序以确定最久没被访问的块。

4. 最不常用

最不常用（LFU）选择近期访问次数最少的块作为被替换的块。这种方法需要记录一段时间内各块被访问的次数，实现代价较大。

5.2.4 写策略

写一般比读更费时。

写策略要解决的两个问题：

1. 如果被写的块不在 Cache 中（写失效）应该如何更新。
2. 如果被写的块在 Cache 中（写命中），是否应该同时更新主存以避免二者不一致。

1 的两种解决方案：
1）按写分配法：先把所写单元所在的块调入 Cache，然后再进行更新。这种方法也称为写时取。然后，问题 1 就转化为了问题 2。
2）不按写分配法：直接写入主存而不将相应的块调入 Cache。这种方法也成为绕写。

2 的两种解决方案：
1）写直达法：不仅把信息写入 Cache 中相应的块，而且也写入主存。
2）写回法：只把信息写入 Cache 中相应的块，只有在该块被替换时，才写回主存。

5.2.5 Cache 结构

5.2.6 Cache 性能分析

CPU时间

$$CPU\text{时间} = (CPU\text{执行周期数} + \text{存储器停顿周期数}) \times \text{时钟周期}$$

这里假设所有存储器停顿都是由 Cache 失效引起。

$$\begin{aligned}\text{存储器停顿周期数} = &\text{读的次数} \times \text{读失效率} \times \text{读失效开销} + \\ &\text{写的次数} \times \text{写失效率} \times \text{写失效开销}\end{aligned}$$

将读和写合并讨论，可将上式简化为

$$\text{存储器停顿周期数} = \text{访存次数} \times \text{失效率} \times \text{失效开销}$$

由于读和写的失效率和失效开销通常不等，所有这只是一个粗略的近似。

从而，

$$CPU\text{时间} = IC \times \left(CPI_{\text{执行}} + \frac{\text{访存次数}}{\text{指令数}} \times \text{失效率} \times \text{失效开销}\right) \times \text{时钟周期}$$

5.2.7 改进 Cache 性能

由平均访存时间的公式可知，改进 Cache 性能主要从以下三个方面着手：

1. 降低失效率
2. 减少失效开销
3. 减少 Cache 命中时间

5.3 降低 Cache 失效率的方法

Cache 失效可细分为如下三种情况：

1. **强制性失效**：第一次访问一个块时这个块显然不在 Cache 中，从而出现失效。也称为冷启动失效或首次访问失效

2. **容量失效**：只考虑全相联映射的情形。由于 Cache 容量有限，新块会替换旧块，如果此后又重新访问旧块，就会出现容量失效。
3. **冲突失效**：由于组相联映射或全相联映射的多个内存块映射到相同的 Cache 导致即使有其他空闲块，也会替换非空闲块，从而导致失效的情况。

5.3.1 调节 Cache 块大小

1. 对于给定 Cache 容量，增大块大小，失效率先降低后升高。先降低时因为更大的 Cache 块会有更少的强制性失效；后升高是因为随着块大小增大，块数减少，会出现更多的容量失效和冲突失效。
2. Cache 容量越大，使得 Cache 失效率最低的块大小就越大。
3. 更大的 Cache 块同时也会带来更大的失效开销。

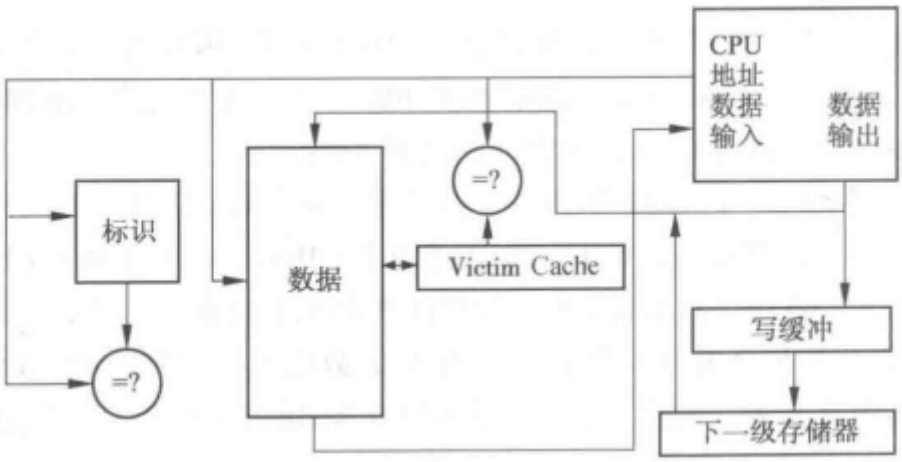
5.3.2 提高相联度

提高相联度显然会降低失效率，但同时因为提高相联度而增加的多路选择器延迟会增加命中时间。

5.3.3 Victim Cache

Victim Cache 是指：在 Cache 和它的下一级存储器直接的数据通路之间增设一个全相联的小 Cache。当 Cache 中一个块被替换时，不是将被替换的块丢弃，而是调入 Victim Cache 中。每当发生失效，在访问主存前，先检查 Victim Cache 中是否有需要的块。

注意：如果把Victim Cache 看作cache系统的扩展，则这是一种减小失效率的方法；如果将其看作下一级存储器的一部分，则这是一种减小失效开销的做法。



5.3.4 伪相联

先采用直接映射的方法访问cache，若失效，则会查找伪相联位置处的块，可以将索引字段的最高位取反作为伪相联的块。在伪相联位置的命中称为伪命中。从下级存储器取来的块会被放在快命中的位置，而原来在此位置的块交换到慢命中的位置。

5.3.5 硬件预取

硬件预取技术是指指令和数据在处理器提出访问请求之前进行预取。预取的内容可以直接放入cache，也可以放入一个访问速度比下一级存储器快的缓冲器。预取利用了存储器的空闲带宽，但如果影响了对正常失效的处理，就可能降低性能。利用编译器的支持，可以减少不必要的预取。

5.3.6 编译器控制的预取

编译时加入预取指令。包括

1. 寄存器预取：把数据取到寄存器中
2. cache预取：把数据取到cache中，不放入寄存器

只有**非阻塞cache**的预取是有意义的。

bad：1. 发出预取指令和预取指令的执行有开销。2. 在支持虚拟存储机制的系统中，预取的虚地址可能会引发异常。

这一段没看懂

5.3.7 编译器优化

编译器优化不需要额外的硬件资源。可以分为减少指令失效和减少数据失效两部分。减少指令失效的工作没有使用高相联度来得有效。以下讨论减少数据失效的手段。

1. **内外循环交换**

   修改前

```
for(j=0;j<100;j++)
{
    for(i=0;i<5000;i++)
    {
        x[i][j]=2*x[i][j];
    }
}
```

   修改后

```
for(i=0;1<5000;i++)
{
    for(j=0;j<100;j++)
    {
        x[i][j]=2*x[i][j];
    }
}
```

   假设cache一个块放数组的100项，共100块。

   修改前：

   当j=0时内层每循环一次，都需要把x的一行刚好放入cache的一个块中。由于cache只有100块，当i=101时会有一个块被替换。从此开始，内层循环每进行一次都有一个块被替换。从而，当j=0时，就会发生5000次失效。当k=1后，当前cache中存放的是x的第4900到4999行，而此刻依次需要访问x的第0，1，2，...行,从而，以后的每次访问皆失效。总之，这样一来每次访问都是失效的，一共失效100*5000=500000次。

   修改后：

   当i=0时依次访问x的第0行的100个数据，这100个数据都在一共cache块中，只有第一次冷不命中失效，从此i取每个值的情况都是一样的。因此一共失效5000次。只有修改前的1/100。

2. **循环融合**

几段程序都对数组进行相同模式的循环访问，写在一起。

修改前：

```
for(i=0;i<5000;i++)
{
    for(j=0;j<100;j++)
    {
        a[i][j]=b[i][j]*c[i][j];
    }
}
for(i=0;i<5000;i++)
{
    for(j=0;j<100;j++)
    {
        d[i][j]=a[i][j]+c[i][j];
    }
}
```

修改后：

```
for(i=0;i<5000;i++)
{
    for(j=0;j<100;j++)
    {
        a[i][j]=b[i][j]*c[i][j];
        d[i][j]=a[i][j]+c[i][j];
    }
}
```

假设cache一个块放数组的100项，共100块。修改前一共失效30000次，分析同1。修改后一共失效20000次，为原来的2/3。

3. 数组合并

数组合并主要是避免了相互干扰的访问导致的冲突失效。

修改前

```
int a[N];
int b[N];
...
for(i=0;i<N;i++)
    c[i]=a[i]+b[i];
```

修改后

```
struct merge{
    int a;
    int b;
}
struct merge merged_array[N];
...
for(i=0;i<N;i++)
    c[i]=merged+array.a[i]+merged+array.b[i];
```

假如数组a和数组b映射到了相同的cache块。修改前a和b来回替换，存在剧烈抖动，失效率很高，而修改后就没有了这个问题。

4. 分块

这是最复杂的一种。

修改前

```
// 矩阵相乘
for(i=0;i<N;i++)
{
    for(j=0;j<N;j++)
    {
        r=0
        for(k=0;k<N;k++)
        {
            r+=y[i][k]*z[k][j];
        }
        x[i][j]=r;
    }
}
```

注：这里临时变量r的使用是为了避免频繁访存。（现在编译器一般会主动做这样的优化）

当i固定时，第二次循环依次计算x的第i行的第j个元素，每次计算都会遍历y的第i行和z的第j列。所以第二层循环执行完一轮会使得y的第i行被遍历N次，z的所有N×N个元素都被访问一次。在极端的情况下，对z的每一个访问都是失效的，这主要是因为cache没有足够多的块。

访存的次数一共是$2N^3+N^2$。

修改后

```
for(jj=0;jj<N;jj+=B)
{
    for(kk=0;kk<N;kk+=B)
    {
        for(i=0;i<N;i++)
        {
            for(j=jj;j<min(jj+N-1,N);j++)
            {
                r=0;
                for(k=kk;k<min(kk+N-1,N);k++)
                {
                    r+=y[i][k]*z[k][j];
                }
                x[i][j]+=r;
            }
        }
    }
}
```

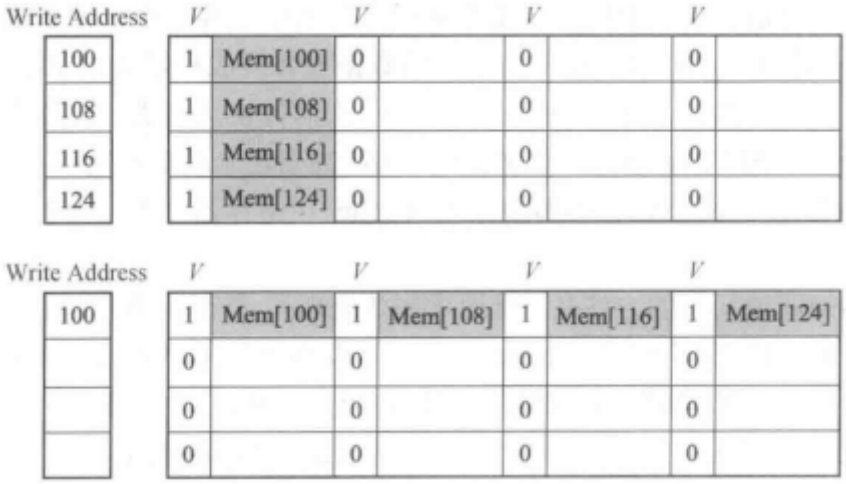
这里B被称为分块因子。首先代码的功能是正确的。再者z被切分成一些大小为N^2/B^2的小块，这些小块上能获得更好的时间局部性。一个小块被使用完后再也不会被再次使用。

5.4 减少cache失效开销

5.4.1 写缓存及写合并

cache将数据写入写缓冲比写入cache快，当数据写入写缓冲，cache就认为写入完成，转而去做其他工作。写缓冲并行地将数据真正地写入主存。假设写缓冲也能存储一定数量的块。

这里有个问题，如果写缓冲中已满，但是同时cache又需要向主存写入新的数据，应该怎么办。采用**写合并**优化这种情境。写合并会将新写入的块和写缓冲中已经存在的块的地址逐一对比，如果有地址匹配的项，则在写缓冲中将新旧块合并。这里书上的描述本身就是含混的。如果没有匹配的项，则等待。以下是一个简单的示例，每个块存放4个64位字



5.4.2 让读失效优先于写

写缓冲还会导致一个问题：读失效时，所读单元的最新值可能在写缓冲中（而非cache（cache已经被替换了）和主存（没来得及写回去））。

暴力解决：总是推迟读失效的处理直至写缓冲空，极大增加了读失效开销。

让读失效优先于写：在读失效时检查写缓冲器的内容，如果没有冲突而且存储器可访问，就继续处理读失效，否则等待缓冲区空。

5.4.3 请求字处理

当从存储器向cpu调入一个块时，往往只有块中的一个字是cpu当前需要的，称这个字为**请求字**。

请求字处理指提前传输请求字到cpu的技术。具体又有两种方案：

1. **尽早重启动**：按顺序访问存储器数据块的字，在请求字没有到达时，cpu等待；一旦请求字到达，立马发送给cpu，让等待的cpu尽早重启动。
2. **请求字优先**：掉块时，首先向存储器请求cpu所请求的字，请求字一旦到达，立即送往cpu，接下来再传输块中剩下的字。请求字优先也称为**回绕读取**或**关键字优先**。

这里有个小细节，请求字回传后cache上的块并不能被看作有效块，因为除请求字外的字还没完成传输。

5.4.4 多级cache

cache一方面要小，小更快；另一方面要大，大会有更低的失效率。多级cache是一种二者的折中。多级cache性能公式：

$$
\begin{aligned}
\text{平均访存时间} &= \text{命中时间}_{L_1} + \text{失效率}_{L_1} \times \text{失效开销}_{L_1} \\
&= \text{命中时间}_{L_1} + \text{失效率}_{L_1} \times \left(\text{命中时间}_{L_2} + \text{失效率}_{L_2} \times \text{失效开销}_{L_2}\right) \\
&= \cdots
\end{aligned}
$$

其中

$$
\text{失效率}_{L_n} = \frac{\text{第}n\text{级的失效次数}}{\text{到达第}n\text{级}cache\text{的访存次数}}
$$

在一级cache的角度来看，多级cache是一种降低失效开销的做法。但如果将所有cache看作一个整体，多级cache就是一种降低失效率的做法。

称失效开销_{L_i}为局部失效率。将所有cache看作一个整体，有

$$
\text{平均访存时间} = \text{命中时间}_{\text{全局}} + \text{失效率}_{\text{全局}} \times \text{失效开销}_{\text{全局}}
$$

这里$\text{失效率}_{\text{全局}} = n\text{级}cache\text{的失效次数}/cpu\text{发出的访存总次数} = \prod \text{失效率}_{L_i}$。

多级相容性：如果i级别cache总是i+1级cache1的一个子集，则称其具有多级相容性。good：它便于实现I/O和cache直接内容一致性的检测。bad：如果i+1级别cahce块比i级大，那么会有多个i级块对应到i+1级的一个块，i级块之一被替换会导致其他对应到同一个i+1级块的i级块也需要被替换。

5.4.5 非阻塞cache

非阻塞cache允许cache在处理失效时继续服务其他访问。

失效下命中：当cache失效时，不是完全拒绝cpu的访问，而是处理部分命中访问。

多重失效下的命中和失效下失效：当cache失效时，不仅处理新的命中访问，还处理新的失效访问。多重失效需要主存能同时送多个数据到cpu才有意义。

5.5 减少命中时间

命中时间会影响cpu的主频。

5.5.1 使用容量小、结构简单的 cache

一级cache应该尽可能满足该条。其他级别的cache可以折中。

5.5.2 虚拟化 cache

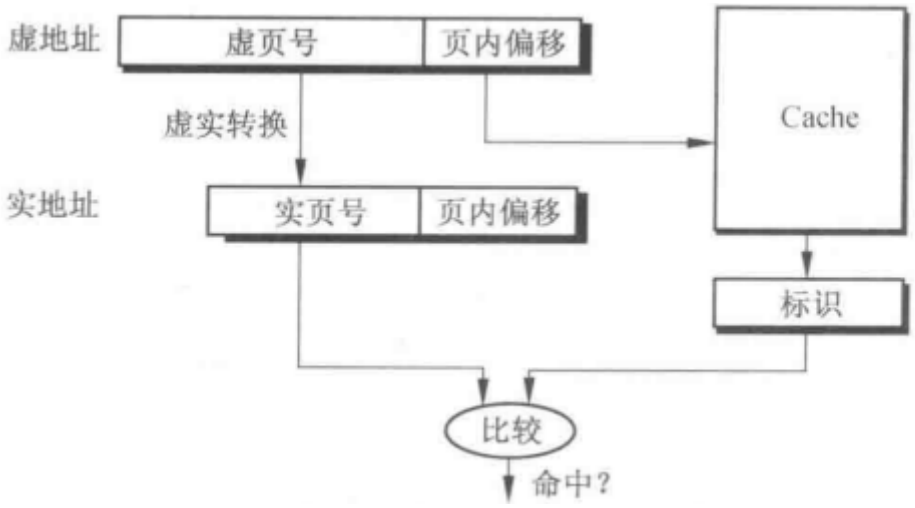
在采用虚存的机器中，每次实际访存都要把虚地址映射成实地址。

访问cache前，要进行虚地址到实地址的映射，增加了命中时间。解决技术1：地址转换单独用一级流水。解决技术2：**虚拟cache**，即使用虚地址映射并访问cache。解决技术3：采用**虚索引+实标识**。

采用虚拟cache会有三个难点：

1. 不同的进程可能使用相同的虚地址来指向不同的物理地址。解决方法是在进程切换时清空cache或在cache地址中增加一个进程标识符字段。
2. 操作系统和用户程序为了共享某些空间，对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。这会使同一块数据在cache中存在两个副本，一个被修改另一个可能未被修改。这些地址称为**同义**或**别名**硬件解决方法是检查并作废可能出现相同物理地址的副本，软件解决办法是强制别名地址的某些地址位相同，如强制最后18位相同，这样它们会被映射到同一个块。
3. I/O通常使用物理地址，如果I/O数据也在虚拟cache缓冲，则需要把物理地址映射为虚拟地址，增加了处理开销。

虚索引+实标识：虚地址的页内偏移和实地址的一样，如果cache容量小于等于页大小，那么页内偏移的某些位正好就是访问cache的索引，而标识位的匹配用的是实地址的页号，好处是块的查找和虚实页号的转换可以并行。



5.5.3 访问流水化

n级cache流水的一级cache几乎使整体的平均命中时间变为1/n（但是不改变单个访问的命中时间）

bad：增加了分支预测失败的代价，也增加了从load指令发射到使用该数据间的时间间隔。

5.5.4 多体 cache

多体cache一般把块地址按顺序分布在各个块。如分4个体，地址模4余i的数据块放在第i个体。

5.5.5 路预测

每一组cache保存一些预测位，标识下一次cache访问本组应该命中的cache块，在预测正确时多路选择的时间被省下，预测错误时再与其他块的tag进行匹配，并修改预测位。

5.5.6 trace cache

传统指令Cache存储静态代码块，而**Trace Cache**存储的基本单元是动态的指令流，可以有效减少取错指令的情况。

bad：相同的指令序列可能被不同的条件分支当作不同的总计而在cache中重复存储这构成一个缺点吗？；很复杂，实现代价大。

5.6 主存

主要性能指标：延迟和带宽

本节的假设：

1. 送地址需 4 个时钟周期
2. 每个字的访问时间为 24 个时钟周期
3. 传送一个字（4 字节）的数据要 4 个时钟周期

如果 cache 大小为 N 个字，则：

失效开销=N(4+24+4)=32N

带宽=4N/32N=0.125 字节/周期

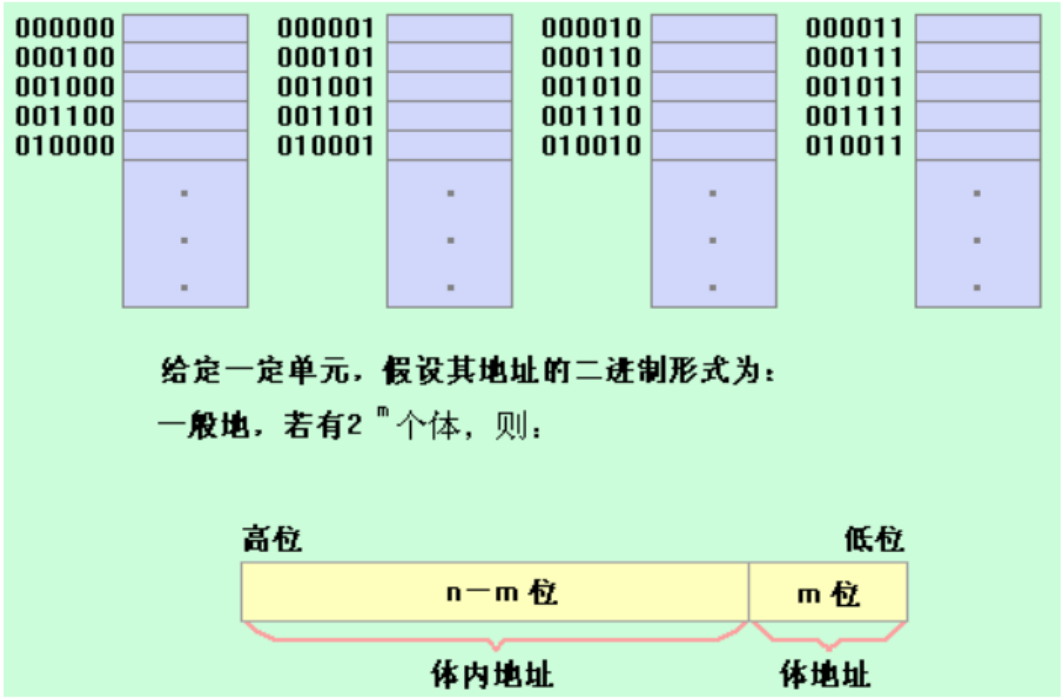
5.6.1 存储器组织技术

1. 增加存储器的带宽

如果把主存和cache带宽增加为原来的N倍，那么失效开销变为1*(4+24+4)=32，带宽变为0.125N字节/周期。bad：1.实现高带宽的存储器总线代价更大；2.具有检错机制的存储器在写入数据时会产生更高的检错代价。

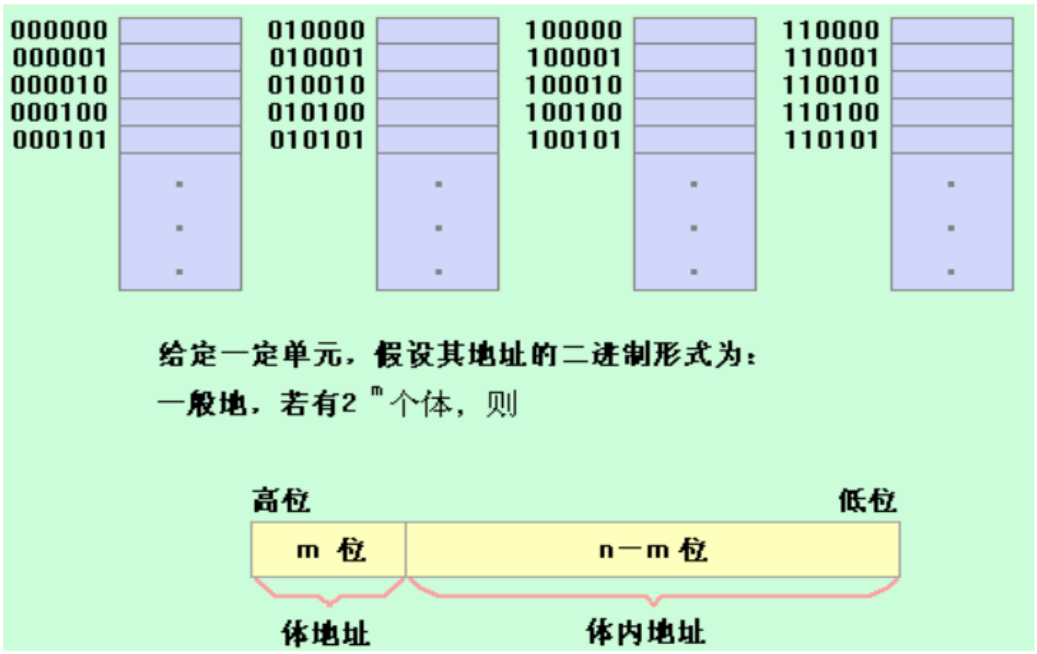
2. 多体交叉

1. 低位交叉编址



低位交叉编址，如果有N个体，如果cache的大小为N个字，那么失效开销为$4+24+4N$。因为字的传送还是需要串行的（除非增加带宽）。

2. 高位交叉编址



3.独立存储体

在包含多个独立存储体的存储器中，每个体都拥有独立的存储控制器、独立的地址线等，支持多个独立的访存（而不仅仅是顺序访存）。这给非阻塞cache失效下的失效、DMA、I/O多处理机等技术提供了很好的支持。例如：cache读操作使用一个存控、I/O设备使用另一个、cache写操作使用另一个。

当采用低位交叉的独立存储器时，可能会出现访存冲突。解决方法：

1. 采用更多的体
2. 软件方法：编译器交换循环；改变数组大小使之不是2的幂次方
3. 硬件素数模法：使体数为素数。当存储体为素数且为2的幂次减1时，体内地址可用

$$\text{体内地址} = \text{地址} mod \text{存储体字数}$$

来计算，这只要取体内地址的高位即可，硬件代价小。体号仍用

$$\text{体号} = \text{地址} mod \text{体数}$$

来计算。

5.6.2 存储器芯片技术

访问时间：从发出读请求到所需的数据到达的时间。

存储周期：两次相邻访存请求之间的最小时间间隔。

存储周期大于访问时间，原因之一是在进行下一次访存之前存储器要等待地址线进入稳定状态。

1. DRAM

DRAM一般做主存。

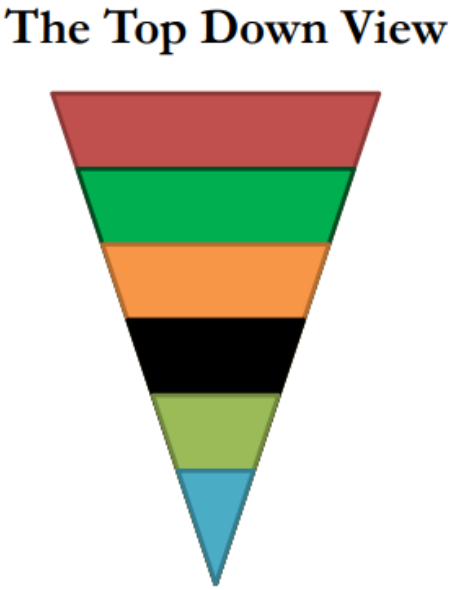
1）DRAM被组织成阵列，每次访问，先发行地址，进行行选通，再发列地址，进行列选通。这可以将芯片的地址引脚数减少一半。

2）DRAM一般用多体存储阵列实现。

3）DRAM需要动态刷新。DRAM优点是只要一个晶体管就能存储一位信息，但位必须被定期刷新。通过读取某行，就可以刷新该行所有位。存储系统必须定时刷新。刷新又有多种不同策略。

4）多个DRAM芯片经常被组装在称为**DIMM条**的小型板上出售。一个存储系统可以有多个存储通道，一个储存通道可有多个DIMM，一个DDIM有两个rank，即正反两面，一个rank又有多个芯片，一个芯片上有多个体，一个体又有多个行/列，一个行/列有多个储存单元。

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column
- Cell



2. SRAM

SRAM一般做cache。

1）SRAM的一个存储单元需要6个晶体管，不需要刷新。

3. DRAM的芯片优化技术

1. **快页模式**：缓冲行数据，下次使用相同数据就省去了行访问的时间。
2. **同步DRAM**：在DRAM接口增加一个时钟信号，使针对一个请求连续同步地传输多个数据时不需要同步开销。
3. **DDR**：在DRAM的上升沿和下降沿都进行数据传输，从而可把数据传输率提高一倍。DDR到DDR2到DDR3，标准电压降低，工作频率升高。

5.7 虚拟存储器

在系统软件和辅助硬件的管理下，主存和辅存构成了逻辑上单一的、可直接访问的大容量主存储器。应用程序员就好像拥有整个地址码宽度的储存空间一样，而不必考虑实际主存空间的大小和使用情况。

5.7.1 基本原理

1. 映射规则

处于辅存的一个页要调入主存，一般允许放在主存的任一位置，即采用全相联映射，这是因为这一级的失效开销非常大，应该尽可能降低失效率。

2. 查找算法

段页表。见操作系统。

3. 替换算法

页式虚存的失效称为**页故障**，几乎所有的操作系统都采用LRU等具有较低失效率的替换算法。

4. 写策略

总是采用写回策略、按序写分配。

5.7.2 快表

访问页表会带来额外的访存开销，为减少此开销，可以把页表进行高速缓存，以减少地址转换时间。计算机常设置一个专门用于地址转换的高速缓存部件，称为**TLB**或**快表**。

5.8 虚存保护和虚存实例

进程：进程是支持程序执行的机制，一个正在运行的程序加上它继续执行所需的任何状态（称为上下文）就是进程。在现代操作系统中，用户态的程序以进程的形式占用系统资源。

5.8.1 进程保护技术

见操作系统

5.8.2 页式虚存举例：64位Opteron

5.8.3 虚拟机保护

略

C 6 输入输出系统

I/O 系统按照主要功能可分为存储 I/O 系统和通信I/O系统。本章主要研究前者。

I/O处理对计算机总体性能的影响

分时操作系统可以在每个进程等待I/O处理时，令CPU去处理其他进程，从而最大化CPU利用率。这似乎使得站在整个系统的角度来看I/O性能不会影响系统性能。但是I/O性能仍然很重要，原因如下：

1. 站在某个进程的角度，I/O处理对速度的影响没有被屏蔽，这尤其对某些对实时性要求较强的进程来说不可接受。
2. 站在整个系统的角度，可能待处理的进程不是很多，如果I/O处理太慢，可能所有进程都处于等待的状态。
3. 一方面，Almdal定律表明系统的整体性能取决于瓶颈处；另一方面，CPU性能提升的速度很快。如果I/O性能没有改进，系统将越来越受限于I/O系统。

6.2 外部存储设备

1. 磁盘设备：包括软盘和硬盘，软盘已被淘汰。
2. Flash存储器
3. 固态硬盘SSD
4. 磁带设备
5. 光盘设备

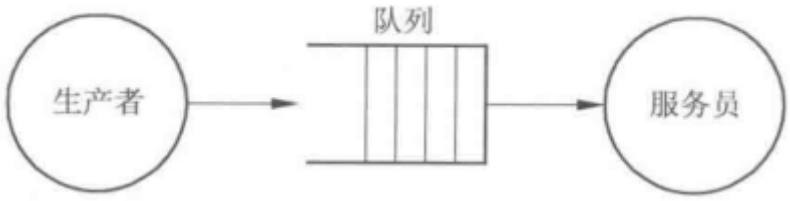
6.3 I/O系统性能分析与评测

1. 连接特性：哪些I/O设备可以和计算机系统相连。
2. I/O 系统的容量：即I/O系统可以容纳的I/O设备的数量
3. 响应时间（也叫响应延迟）和吞吐率（也叫I/O带宽）。

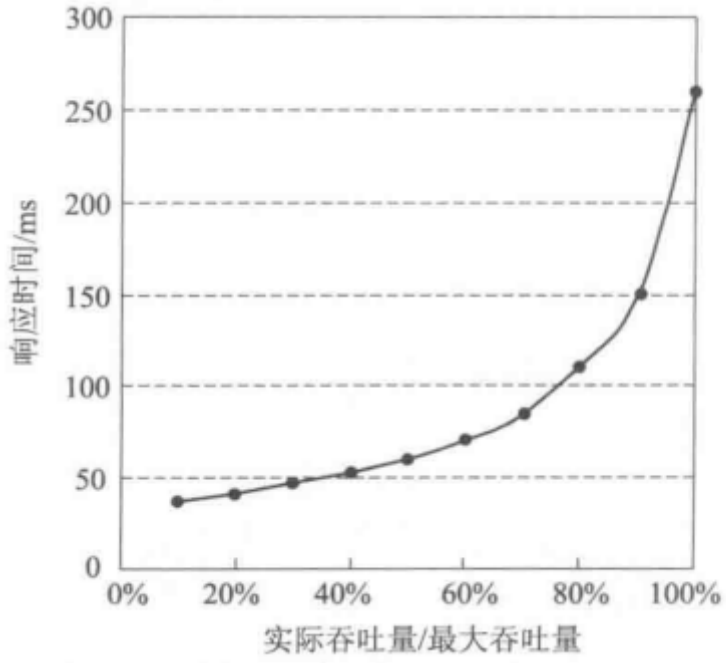
4. I/O操作对 CPU 的打扰情况：即当某个进程执行时，由于其他进程的I/O操作，使得该进程的执行时间增加了多少。
5. I/O系统的可靠性、可用性和可信性。[[6.4]]

6.3.1 响应时间与吞吐率

这是一对矛盾。以生产服务模型建模：



生产者不断产生任务放入队列，服务员按照FIFO的规则取出任务加以服务。响应时间定义为一个任务被放入队列到其被服务完毕的时间，包括排队时间（等待之间的任务被服务）和该任务自己的被服务时间。吞吐率定义为单位时间内服务员完成的任务数。



6.3.2 Little 定律

对I/O系统性能的分析用到的主要数学工具是排队论。

Little 定律适用于黑箱系统，其满足如下条件：

1. 有几个独立的请求源
2. 黑箱由足够的服务能力使得系统的输入速率与输出速率相等
3. 黑箱系统内部不会创建或消灭任务

则：

$$\text{系统中的平均任务数} = \text{到达率} \times \text{平均响应时间}$$

6.3.3 M/M/1 排队系统

略

6.3.4 M/M/m 排队系统

略

6.3.5 I/O基准测试程序

可以使用I/O基准测试程序来反映响应时间和吞吐率之间的平衡关系。事务处理委员会（TPC）制定了许多基准测试程序及规范。

TPC的特点：

1. 测试结果中给出了系统的价格因素
2. TPC模拟的是实际系统
3. 结果经TPC审核后才能发布
4. 吞吐率指标受到响应时间的限制。例如TPC-C的新事务的响应时间必须小于5s。
5. 通过独立的机构来维护

6.4 系统的可靠性、可用性和可信性

数据的丢失或受损的后果可能无法弥补，因此存储设备的指导思想是无条件保护好用户数据。

6.4.1 故障、错误和失效

失效：系统提供的功能层面，指系统实际提供的服务偏离了指定的服务。比如火箭系统的功能是升空，但是火箭爆炸了，就是失效。

错误：系统模块中的缺陷，失效的原因。

故障：错误的原因。

总之，故障产生的错误会引起系统的失效

故障的分类：暂时性、间歇性、永久性。

故障产生的原因：硬件、设计（软件编程等）、人的操作失误、环境。

6.4.2 可靠性和可用性

可靠性靠 MTTF 衡量，见[[C1]]。可用性也见[[C1]]

想提高可靠性，要么避免故障，要么设置容错部件。以下方法可以提高可靠性

1. 故障避免技术：通过合理构建系统来避免故障
2. 故障容忍技术：采取冗余措施
3. 错误消除技术：通过验证，最大限度地减少潜在的错误
4. 错误预报技术：通过分析，预报错误的出现，以便提前采取应对措施

6.4.3 可信性

可信性指的是服务的质量，即在多大程度上可以认为提供的服务是正确的。可信性不可被量化。

可靠性用 MTTF 衡量，可用性是系统正常工作的时间在连续两次正常服务间隔时间中所占比率。见[[C1 计算机体系结构的基本概念#1.4.1 基本的可靠性模型]]。可信性暂没有统一的量化标准。

提高系统组成部件可靠性的方法：

1. 有效构建方法：消除故障隐患。
2. 纠错方法：采用容错的方法，比如三模冗余。

6.5 廉价磁盘冗余阵列

6.5.1 磁盘阵列

磁盘阵列（DA）：通过使用多个磁盘构成的阵列代替一个大容量的磁盘来提高整体性能。

优点：磁盘阵列提高了并行性。一方面，同时发生的多个请求可以由多个盘并行处理，减少了I/O请求的排队等待时间；另一方面，如果一个请求访问多个块，也可以由多个盘并行处理，提高单个请求的数据传输率。

缺点：可靠性降低。如果磁盘阵列用了N个盘，则系统可靠性为单个磁盘的1/N。

缺点的解决方法：增加冗余信息来容错。这种磁盘阵列称为**廉价磁盘冗余阵列（RAID）**。

6.5.2 廉价磁盘冗余阵列的特点

1. RAID由一组物理磁盘磁盘组成，操作系统视之为一个逻辑磁盘。
2. RAID能以**条带（Strip）**的形式在这组物理磁盘上分布数据。不同实现的条带宽度不同。
3. 冗余信息存储在冗余磁盘空间中，在发生磁盘失效时可以恢复数据。

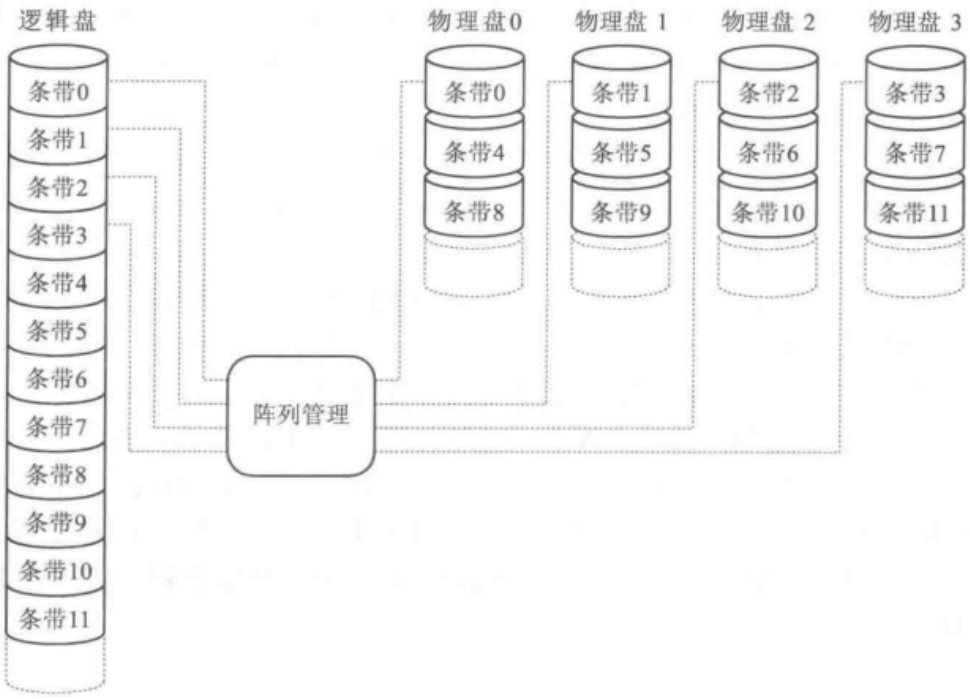
RAID0不支持特点3。

热备份盘：平时待机，如果某个磁盘失效，就代替其工作。如果该过程自动，将显著减少MTTR。

热切换技术：允许系统在不关机的情况下更换设备的技术。

6.5.3 RAID0

RAID0采用**数据分块技术**，把数据分布在多个磁盘上，无冗余信息。



优点：

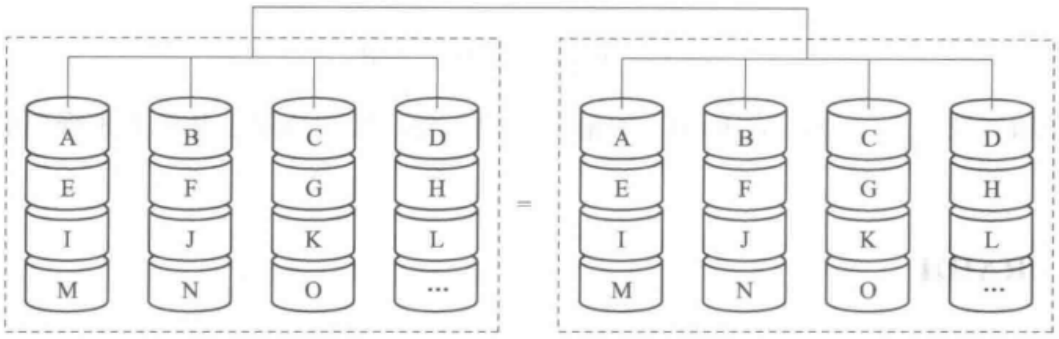
1. 如果单个I/O请求由多个相邻的条带组成，则该请求需要访问的多个条带可以并行处理，提高了数据传输率。
2. 设计和实现十分简单。

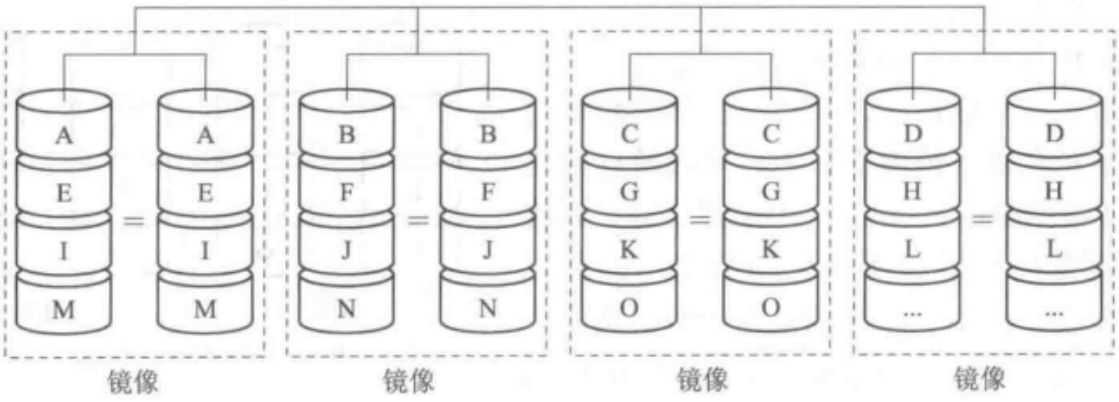
缺点：没有采用冗余技术改善可靠性。

6.5.4 RAID1

RAID1采用**镜像盘技术。**

如果同时采用分块技术和镜像盘技术：若先将数据分布到4个磁盘，再将这4个磁盘镜像到另一组的4个磁盘，则称为RAID0+1或RAID01；若先用镜像盘技术构建4对磁盘，再将数据分布到这4对磁盘，则称为RAID1+0或RAID10。





优点：

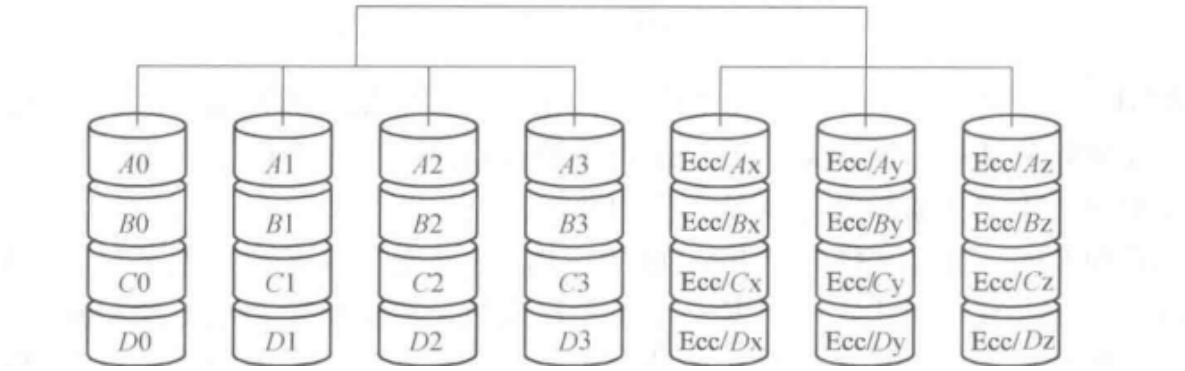
1. 读操作较快。读时间由最快的那个盘决定。这会使得RAID1在处理大批量的读请求时比RAID0更快，因为它节省了一些寻道和旋转时间。
2. 写操作较快。写时间由最慢的那个盘决定。但是由于没有检验等操作，相对以后更级别的RAID来说RAID1较快。
3. 可靠性高。只要不是数据盘和镜像盘同时失效，RAID1都是可以容忍的。

缺点：昂贵。

RAID1不能检测数据位的错误，可以在一个盘罢工后不令系统停摆

6.5.5 RAID2

RAID2为**位交叉式汉明编码阵列**。位交叉指数据按位存放在不同的磁盘中。假设数据字体为4位，RAID2将数据字的每一位分布存放在不同的数据盘上，并将这4位数据字的3位汉明码存放在另外三个校验（Ecc）盘上。汉明码可以检两位错，纠一位错。



优点：

1. 使用汉明码进行及时的检错和纠错
2. 可以实现高速数据传输

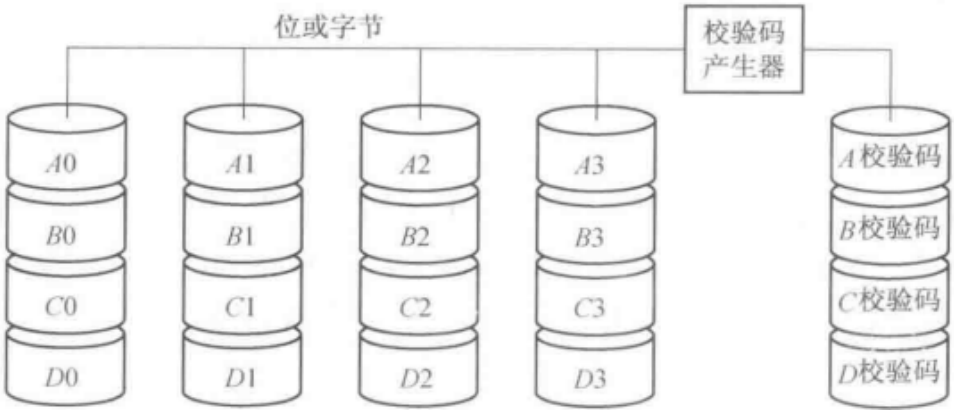
缺点：

1. 需要多个校验盘，冗余磁盘数量与数据磁盘数的对数成正比
2. 需要额外的数据校验时间
3. 阵列控制器设计复杂

RAID2可以检测数据位的错误并修正，不能在一个盘罢工后使系统继续运行

6.5.6 RAID3

RAID3为**位交叉式奇偶校验阵列**。无论数据盘有多少，校验盘只有一个。



RAID3将各个数据盘的数据位加起来，将模2和放入校验盘。如果一个数据盘失效，就可以反推出失效盘上的数据。

优点：

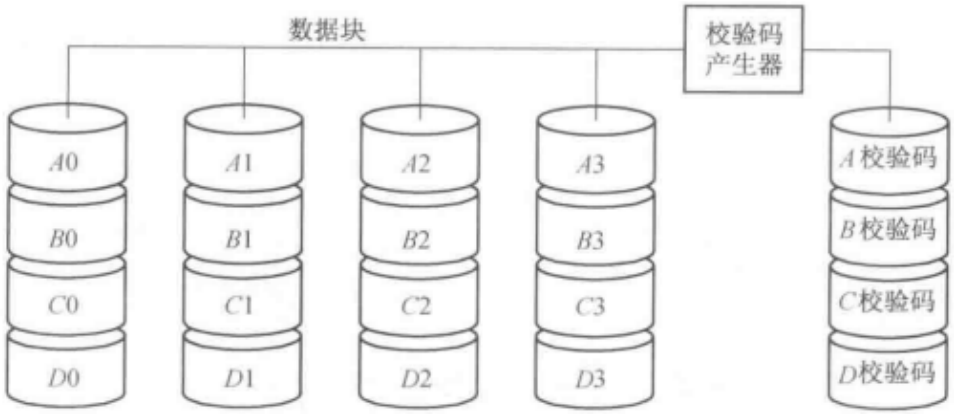
1. 冗余代价较低
2. 数据传输率高

缺点：

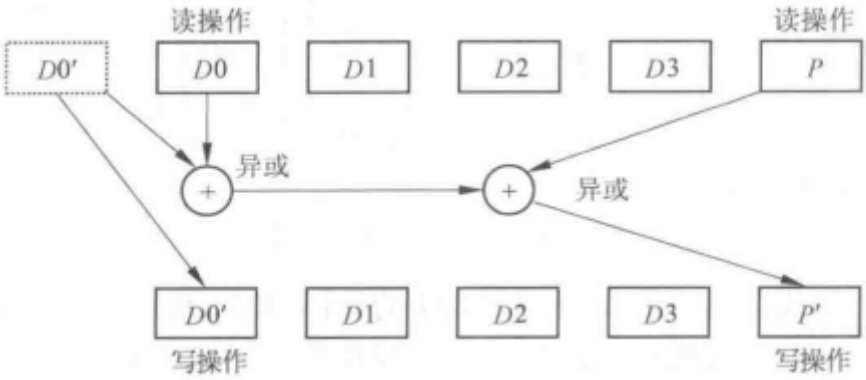
1. 阵列控制器设计复杂

6.5.7 RAID4

RAID4为**块交叉奇偶校验阵列**。数据以块（大小可变）交叉的方式存于各盘，冗余的奇偶校验信息放在校验盘。



RAID4与RAID3相比，数据传输率低了，但是能并行处理多个I/O请求。在RAID4中，读一个数据块需要对一个数据盘和校验盘同时进行操作，读可以并行；写一个数据块（例如D0）需要读出D0、校验盘P，然后改写二者，如下：



优点：

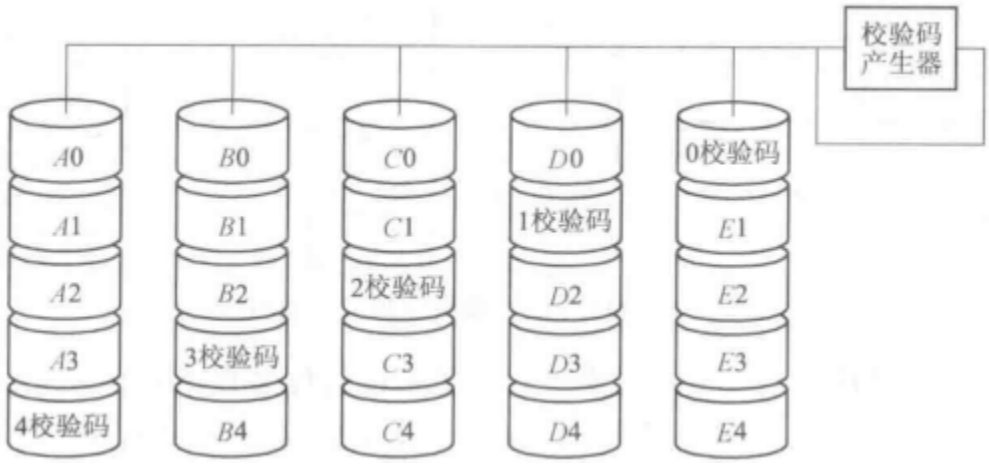
1. 冗余代价低
2. 能同时处理多个读操作

缺点：

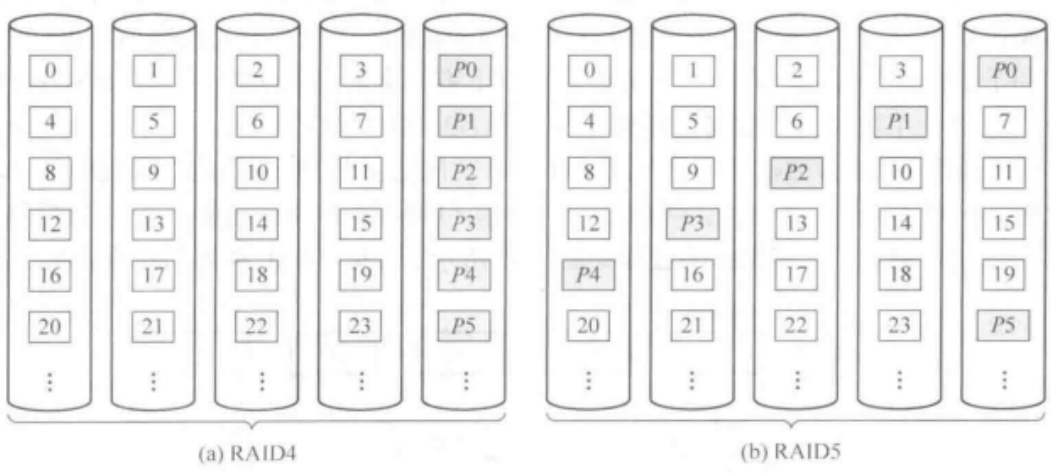
1. 阵列控制器设计复杂
2. 写操作必须对冗余盘中的校验信息进行修改，写不能并行

6.5.8 RAID5

RAID5为**块交叉分布式奇偶校验盘阵列**，又称为旋转奇偶校验独立存取阵列。校验块以交叉的方式存于各盘，没有专用的冗余盘。



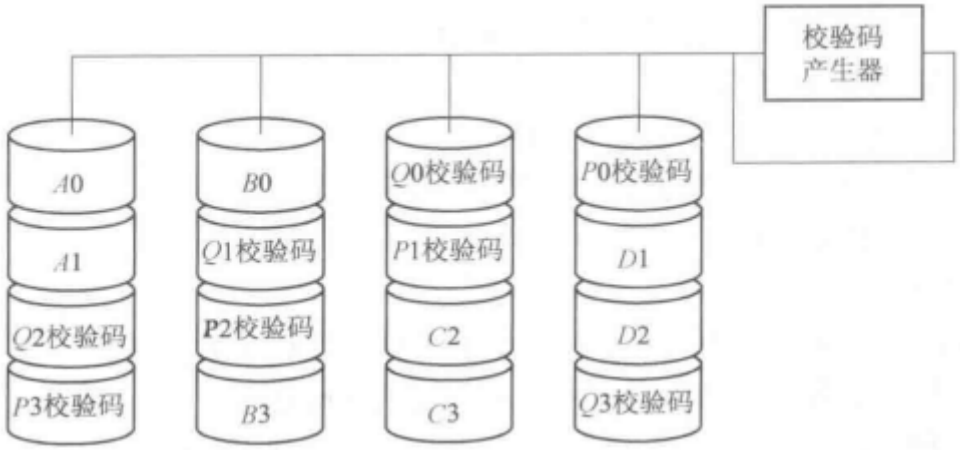
一下是RAID4和RAID5的对比：



RAID5能同时写0和4两个块，因为二者及二者的校验信息都在不同的盘。克服了RAID4写不能并行的缺点。

6.5.9 RAID6

RAID6为**双维奇偶校验独立存取盘阵列**，又称为P+Q双检验磁盘阵列。数据以块交叉的方式存于各盘，冗余的检错、纠错信息均匀地分布在所有磁盘上。



RAID6是RAID5的扩展，它在RAID5的基础上另增一组校验码。由于每次读写数据都要访问一个数据盘和两个包含冗余信息的磁盘，因此RAID6可以容忍双盘出错。但是RAID6的校验信息存储开销是RAID5的两倍。

优点：容错性能好

缺点：阵列控制器设计复杂、冗余信息计算复杂

6.5.10 RAID的实现与发展

磁盘阵列的主要实现方式：

1. 软件方式：成本低，但占用过多主机时间且带宽低。
2. 阵列卡方式：把RAID管理软件固化在I/O控制卡上，从而可不占用主机时间。
3. 子系统方式：这是一种基于通用接口的开放式平台，可用于各种主机平台和网络系统。

现在的RAID产品具有的一些业界标准功能特性：

1. 在线优化调整，例如在线扩容、在线RAID级别转换以及在线条带宽度调整等。
2. 可配置热备份盘，可对多个备份盘进行有效管理。
3. 磁盘热插拔及自动重构。
4. 在多个物理磁盘上创建多个RAID等。

主要热点问题：

1. 新型阵列体系结构
2. RAID结构与所记录文件特性的关系
3. 在RAID冗余设计中，综合平衡性能、可靠性和开销的问题
4. 超大型磁盘阵列在物理上如何构造和连接的问题
5. 如何利用新型存储部件（如SSD）构建RAID

6.6 I/O设备与CPU/存储器的连接——总线

总线是各子系统之间共享的通信链路。总线的优点是低成本和多样性，各子系统和计算机设备之间可以通过总线方便地互联。总线的缺点是它必须被独占使用，从而可能成为瓶颈。

6.6.1 总线设计时的考虑因素

选　择	高　性　能	低　代　价
总线复用方式	独立的地址和数据总线	数据和地址总线分时复用
数据总线宽度	宽（例如：64 位）	窄（例如：8 位）
传输块大小	块越大总线开销越小	每次传送单字
总线主设备	多个（需要仲裁）	单个（无须仲裁）
分离事务总线	采用	不采用
定时方式	同步	异步

当总线上连接多个设备时，就需要由**总线仲裁**机制来确定哪个总线主设备来控制使用总线。

分离事务又称为流水事务，基本思想是将总线事务分为请求和应答两部分，使得总线可以在某个事务的请求与应答之间的空闲时间处理其他事务。分离事务能获得较高带宽，但一般会增加数据传送延迟。

异步总线没有统一的参考时钟，因此主存设备之间需要采用握手协议。异步处理可以满足大量不同速度设备的连接，不需要考虑时钟扭曲问题（时钟通过长距离传输后相位漂移）。**同步总线**由于没有握手协议的开销，一般更快。

6.7 通道

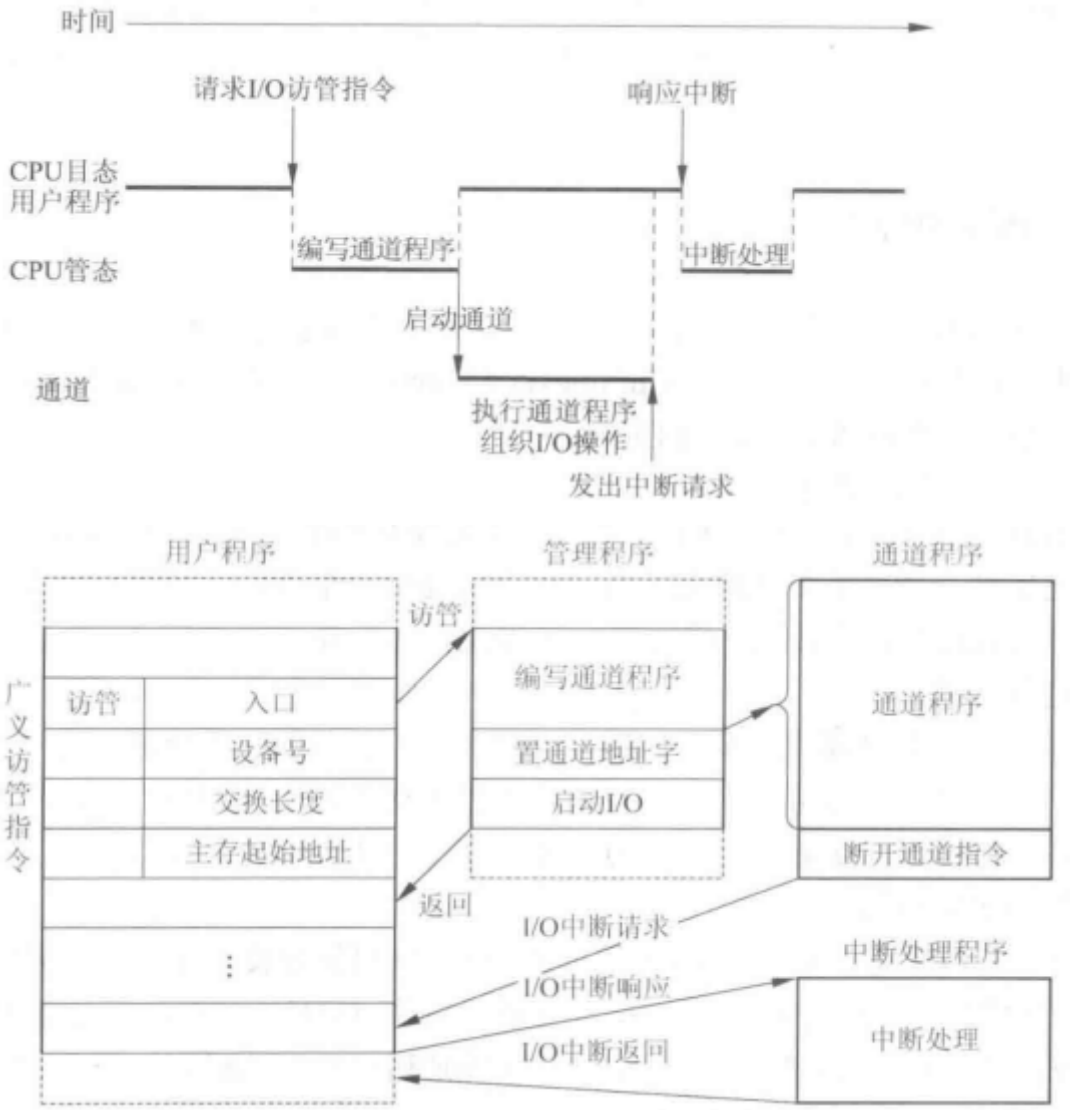
通道可以分担全部或大部分的I/O工作，使CPU从对外设的管理中解放。

6.7.1 通道的功能

1. 接受来着CPU的I/O指令，并根据指令要求选择指定的外设与通道连接。
2. 执行CPU为通道组织的通道程序，从主存中取出通道指令，对通道指令译码，并根据需要向被选中的设备控制器发出各种操作命令。
3. 为主存和外设设置传输控制信息
4. 指定传送工作结束时要进行的操作
5. 检查外设的工作状态是否正常
6. 在数据传送过程中完成必要的格式变换

6.7.2 通道的工作过程

1. 在用户程序中使用访管指令进入管理程序，由管理程序组织一个通道程序，并启动通道
2. 通道执行CPU为它组织的通道程序，完成指定的数据I/O工作
3. 通道程序结束后向CPU发中断请求



6.7.3 通道的种类

1. **字节多路通道**

   分时工作。当多台设备连接到一个字节多路通道上时，通道每连接一个外设，只传送一个字节，然后又与另一台设备连接，并传送另一个字节，如此循环。

2. **选择通道**

   在一段时间内只单独为一台高速外设服务，当这台设备的数据传送工作全部完成后，通道才能为另一台设备服务。

3. **数组多路通道**

   使得数据传输与其他外设的辅助操作可以并行。具体做法为：通道每连接一个外设就传送一个数据块，传送完成后又与另一台高速设备连接，并传送另一个数据块，如此循环。

6.8 I/O与操作系统

充分发挥储存性能需要借助操作系统，同时需要操作系统提供数据保护。

6.8.1 DMA和虚拟存储器

6.8.2 I/O和Cache的数据一致性

同时使用了cache和虚存，那一个数据就可能有三个副本：cache、主存、辅存。需要想办法保证其一致性。

如果将I/O总线直接与Cache相连，问题就解决了，但会产生两个新问题：

1. CPU与I/O设备竞争使用Cache，影响CPU'效率
2. 传入主存的数据不一定是CPU当下需要访问的，这些数据如果要经过cache就削弱了cache作为缓冲区的作用。

因此一般将I/O直接连到存储器上。这会产生两个方面的数据不一致问题：

1. I/O从存储器取数据，但数据的最新值在cache中，还未被写回主存。采用写回而非写直达的策略会出现这个问题。解决方法：操作系统根据I/O使用的存储器地址来查找cache中是否有对应的块，如果有并且修改过，则先写回再取。
2. I/O向主存写数据，这样cache中的备份可能就是旧的值。解决方法：操作系统根据I/O使用的存储器地址来查找cache中是否有对应的块，如果有，就置0有效位作废该块。