第1章：计算机系统结构的基础知识

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 ☆

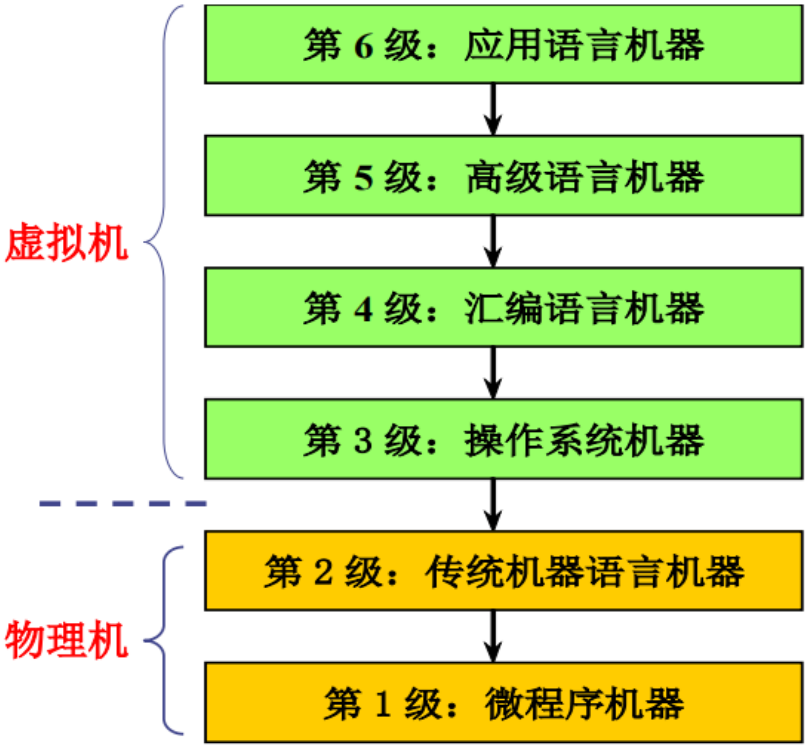
计算机系统=硬件+软件

- **硬件**：
  包括**控制器、存储器、系统结构、运算器、显示器、键盘、鼠标、打印机、磁盘、网络、外设**等。
- **软件**：
  包括**编译器、解释器、运行环境、调试器、测试程序**等。

计算机的语言是从**低级向高级**进行发展的：

高一级语言的语句相对于低一级语言来说功能更强，更便于应用，但又都**以低级语言为基础**

从计算机语言的角度，把计算机系统按功能划分成 多级层次结构，每一层以一种语言为特征



- **物理机**：用硬件/固件实现的机器（最下面的两级机器）（固件是指：具有一定软件功能的硬件）
- **虚拟机**：由软件实现的机器

六个层次结构：

- **微程序机器**
  - 微程序机器级的机器语言是微指令集

    - 由该语言编写的微程序一般是直接由硬件解释实现的
- **传统机器语言机器**
    - 该机的语言是指该计算机的**指令集**，比如不同的cpu对应的不同的指令集
    - 机器指令程序可以由微程序进行**解释**，从而可以实现多种指令集
- **操作系统机器**
    - 直接管理计算机的软硬件资源
- **汇编语言机器**
    - 用汇编语言编写的程序，首先**翻译**或**解释**成第 3 级和第 2 级语言，然后再由相应的机器执行
    - 汇编程序：完成汇编语言翻译的程序
- **高级语言机器**
    - 用这些语言所编写的程序一般是由称为编译程序翻译到第 4 级或第 3 级上，然后再被计算机执行
    - 也就是说，当编写完高级语言之后，需要对其进行**翻译或者解释**，然后再由计算机执行
- **应用语言机器**
    - 为使计算机满足某种用途而专门设计的：比如：SQL语言等
    - 应用语言编写的程序一般由应用程序包翻译到第5级语言上再去由它执行

各机器级的实现主要靠翻译或解释，或两者的结合

注意：
（1）翻译：
先用转换程序把高一级机器上的程序转换为低一级机器上等效的程序
然后再在低一级机器上运行，实现程序的功能

（2）解释：
对于高一级机器上的程序中的每一条语句或指令，是转去执行低一级机器上的一段等效程序。
执行完后，再去高一级机器取下一条语句或指令，再进行解释执行，如此反复，直到解释执行完整个程序

解释执行比编译后再执行所花的时间多，但占用的存储空间较少

1.1.2 计算机系统结构的定义 ☆

（1）定义：

传统机器程序员所看到的计算机属性，即概念性结构与功能特性，因为按照计算机系统的多级层次结构，不同级程序员所看到的计算机具有不同的属性
比如：**数据表示、寻址规则、寄存器定义、指令系统、中断系统、机器工作状态的定义和切换、存储系统、信息保护、I/O 结构**

（2）计算机系统结构的实质：

确定计算机系统中软硬件的界面，界面之上是软件实现的功能，界面之下是硬件和固件实现的功能

1.1.3 计算机组成和计算机实现 ☆

计算机系统结构VS计算机组成VS计算机实现

1. **计算机系统结构**：
    - 计算机系统的软、硬件的界面，即机器语言程序员所看到的传统机器级所具有的属性
2. **计算机组成**：
    - 计算机系统结构的**逻辑实现**，包含物理机器级中的数据流和控制流的组成以及逻辑设计等。

- 着眼于：物理机器级内各事件的排序方式与控制方式、各部件的功能以及各部件之间的联系。

3. **计算机实现**：
   - 计算机组成的物理实现
   - 包括处理机、主存等部件的物理结构，器件的集成度和速度，模块、插件、底板的划分与连接，信号传输，电源、冷却及整机装配技术等
   - 着眼于：**器件技术**（起主导作用）、微组装技术

具有相同系统结构的计算机可以采用不同的计算机组成。

同一种计算机组成又可以采用多种不同的计算机实现。

1.1.4 计算机系统结构的分类 ☆

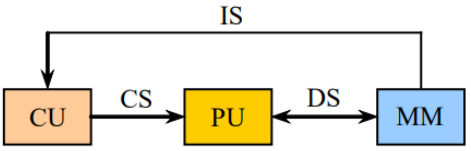
Flynn分类法

按照指令流和数据流的多倍性进行分类

- **指令流**
  计算机执行的指令序列
- **数据流**
  由指令流调用的数据序列
- **多倍性**
  在系统最受限的部件上，同时处于同一执行阶段的指令或数据的最大数目

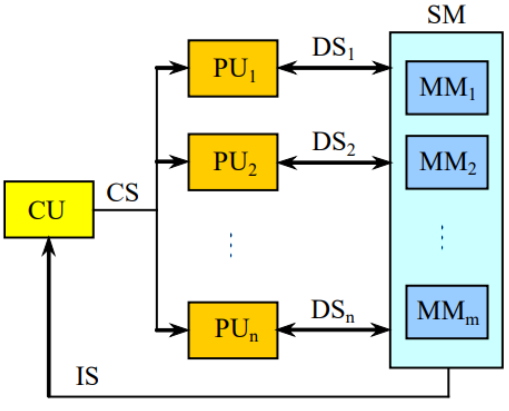
可以把计算机系统结构分为四类：

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



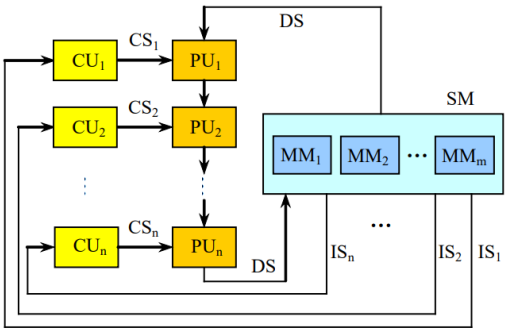
（a）SISD 计算机

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



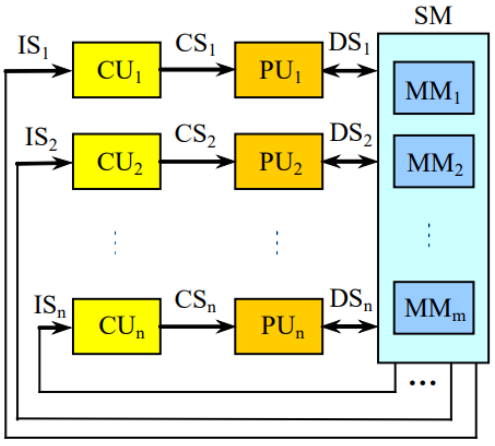
（b）SIMD 计算机

3. 多指令流单数据流**MISD**（Multiple Instruction stream Single Data stream）
（人为的划分,并没有实际机器的实现）



（c）MISD 计算机

4. 多指令流多数据流**MIMD**（Multiple Instruction stream Multiple Data stream）



（d）MIMD 计算机

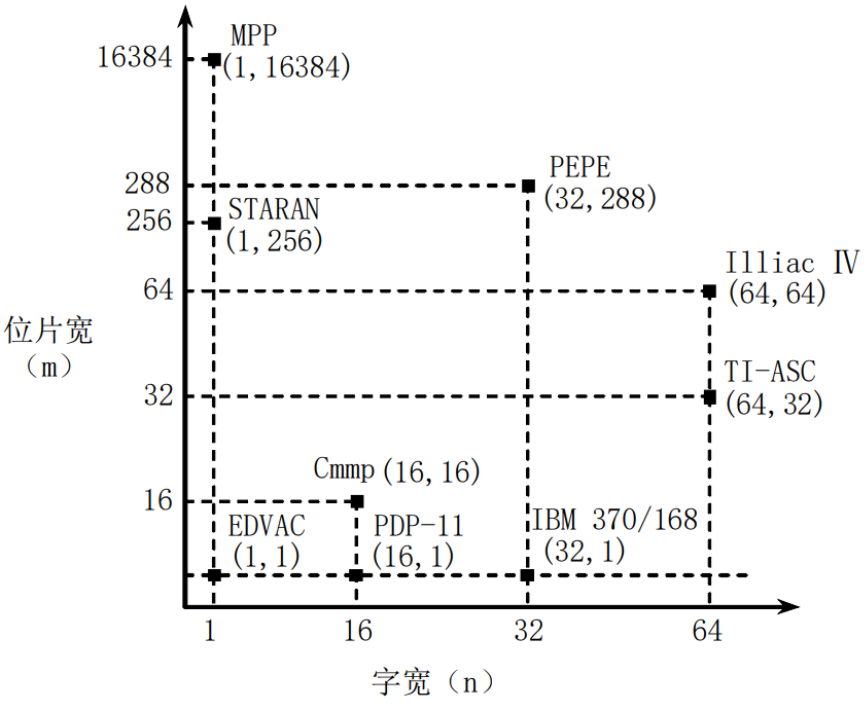
注意：IS：指令流，DS：数据流，CS：控制流，CU：控制部件，PU：处理部件，MM和SM：存储器

冯氏分类法

用系统的最大并行度对计算机进行分类

最大并行度：计算机系统在单位时间内能够处理的最大的二进制位数

用平面直角坐标系中的一个点代表一个计算机系统，其横坐标表示字宽（n位），纵坐标表示一次能同时处理的字数（m字）。**m×n就表示了其最大并行度**。



1. 字串位串（**WSBS**）：n = 1，m = 1。(第一代计算机发展初期的纯串行计算机)
2. 字串位并（**WSBP**）： n > 1，m = 1。这是传统的单处理机，同时处理单个字的多个位，如16位、32位等。
3. 字并位串（**WPBS**）：n = 1，m > 1。同时处理多个字的同一位（位片）。
4. 字并位并（**WPBP**）：n > 1，m > 1。同时处理多个字的多个位。

平均并行度：

与最大并行度密切相关的一个指标。取决于系统的运用程度，与应用程序有关。

假设每个时钟周期内能同时处理的二进制位数为P_i，则T个时钟周期内的平均并行度为：

$$P_a = \frac{\sum_{i=1}^{T} P_i}{T}$$

系统在T个时钟周期内的平均利用率定义为：

$$\mu = \frac{P_a}{P_m} = \frac{\sum_{i=1}^{T} P_i}{TP_m}$$

Handler分类法

根据**并行度和流水线**对计算机进行分类

把计算机的硬件结构分成3个层次

- 程序控制部件（PCU）的个数**k**
- 算术逻辑部件（ALU）或处理部件（PE）的个数**d**
- 每个算术逻辑部件包含基本逻辑线路(ELC)的套数**w**

$$t（系统型号）=（k，d，w）$$

改进：

$$t（系统型号）=（k \times k', d \times d', w \times w'）$$

- k'：宏流水线中程序控制部件的个数
- d'：指令流水线中算术逻辑部件的个数
- w'：操作流水线中基本逻辑线路的套数

例如：Cray-1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1～14位流水线处理。所以Cray-1系统结构可表示为：

t（Cray-1）=(1，12×8，64×（1～14）)

1.2 计算机系统的设计

1.2.1 定量原理（4个）

大概率事件优先原则（以经常性事件为重点）

对于大概率事件（最常见的事件），赋予它优先的处理权和资源使用权，**以获得全局的最优结果**

也就是常说的**经常性事件**：对经常发生的情况采用优化方法的原则进行选择，以得到更多的总体上的改进，**优化**是指分配更多的资源、达到更高的性能或者分配更多的电能等。

Amdahl定律 ☆

当我们对系统的某个部分加速时，其对系统整体性能的影响取决于该部分的重要性和加速程度。

假设我们对机器（部件）进行某种改进，那么机器系统（部件）的**加速比**就是

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

$$性能 = \frac{同一过程}{执行时间}$$

核心概念：**时间**

系统加速比依赖于两个因素

- 可改进比例
  - 可改进部分在原系统计算时间所占的比例
  - 它总是小于或等于1
- 部件加速比
  - 可改进部分改进以后的性能提高了多少，一般情况是大于1的，比如，设部件加速比为K,那么，现在该部分执行同一个操作所用时间就为：$T_{new} = \frac{T_{old}}{K}$

所以，我们可以设：可改进比例为α,部件加速比为K,所以，改进后的总执行时间为：

$$T_{new} = (1-\alpha) \times T_{old} + \frac{\alpha \times T_{old}}{K}$$

$$T_{new} = T_{old} \times ((1-\alpha) + \frac{\alpha}{K})$$

所以，系统加速比即为：

$$系统加速比 = \frac{1}{(1-\alpha)+\frac{\alpha}{K}}$$

要想显著加速整个系统，必须提升全系统中相当大的部分的速度

定律观点：

- **性能增加具有递减规则**
  仅仅对计算机中的一部分做性能改进，则改进的越多，系统获得的效果越小
- 针对整个任务的一部分进行优化，则最大加速比不大于：$\frac{1}{1-可改进比例}$
- **衡量好的计算机系统**
  好的计算机系统是一个**带宽平衡**的系统，而不是看它使用的某些部件的性能

所以：这个地方可能会出**计算题**：

- **例1**：
  将计算机系统中某一功能处理速度加快15倍，该功能的处理时间仅占系统运行时间的40%，则采用此改进方法后，系统的性能提高多少？

设之前系统所用时间为t

则加速后所用时间为t'

$$t' = (1-0.4)t + 0.4t \cdot \frac{1}{15}$$
$$= 0.6t + \frac{2}{75}t$$
$$= \frac{3}{5}t + \frac{2}{75}t = \frac{45+2}{75}t = \frac{47}{75}t$$

∴性能提高比例为：

$$Sn = \frac{\frac{1}{t'}}{\frac{1}{t}} = \frac{t}{t'} = \frac{75}{47} \approx 1.6$$

- **例2**：
  某计算机系统采用浮点运算部件后，使浮点运算速度提高到原来的25倍，而系统运行某一程序的整体性能提高到原来的4倍，试计算该程序中浮点操作所占的比例。

设所占比例为x则

$$t' = (1-x)t + x \cdot t \cdot \frac{1}{25}$$
$$= t - xt + \frac{x}{25}t$$
$$= (1-\frac{24}{25}x)t$$

又∵
$$Sn = \frac{\frac{1}{t'}}{\frac{1}{t}} = \frac{t}{t'} = \frac{1}{1-\frac{24}{25}x} = 4$$

解得 $x = \frac{25}{32} \approx 78.1\%$

程序的局部性原理 ☆

- 时间局部性
  时间局部性是指如果程序中的某条指令一旦执行，则不久之后该指令可能再次被执行；
  如果某数据被访问，则不久之后该数据可能再次被访问。
- 空间局部性
  空间局部性是指一旦程序访问了某个存储单元，则不久之后，其附近的存储单元也将被访问。

CPU性能公式 ☆

CPU时间： 执行一个程序所需要的cpu时间

$$CPU时间 = 执行程序所需的时钟周期数 \times 时钟周期时间$$

其中：时钟周期时间是系统时钟频率的倒数

CPI：每条指令执行的平均时钟周期数

$$CPI = \frac{执行程序所需的时钟周期数}{该程序所具有的指令条数} = \frac{CLK}{IC}$$

IC：程序所具有的指令条数 **Instruction Count**

所以，程序执行的CPU时间可以写为：

$$CPU时间 = IC \times CPI \times 时钟周期时间$$

所以，cpu的性能取决于三个参数：

- **时钟周期时间**：取决于硬件实现技术和计算机组成；
- **CPI**：取决于计算机组成和指令系统的结构；
- **IC**：取决于指令系统的结构和编译技术。

CPU性能公式的进一步优化 ☆

假设:计算机系统有n种指令

CPI_i ：第i种指令的时钟周期数

$\mathrm{IC_i}$:在程序中第i种指令出现的次数则:

$$CPU时钟周期数 = \sum_{i=1}^{n}(\mathrm{CPI_i} \times \mathrm{IC_i})$$

$$CPU时间 = 执行程序所需的时钟周期数 \times 时钟周期时间 = \sum_{i=1}^{n}（CPIi \times ICi）\times 时钟周期时间$$

$$\mathrm{CPI} = \frac{时钟周期数}{IC} = \frac{\sum_{i=1}^{n}(\mathrm{CP_i} \times \mathrm{IC_i})}{IC} = \sum_{\mathrm{i}=1}^{n}\left(\mathrm{CPI_i} \times \frac{IC_\mathrm{i}}{IC}\right)$$

$\frac{IC_\mathrm{i}}{IC}$ 反映了第i种指令在程序中所占的比例

例题：

例 1.3　假设 FP 指令的比例为 25%，其中，FPSQR 占全部指令的比例为 2%，FP 操作的 CPI 为 4，FPSQR 操作的 CPI 为 20，其他指令的平均 CPI 为 1.33。现有两种改进方案，第一种是把 FPSQR 操作的 CPI 减至 2，第二种是把所有的 FP 操作的 CPI 减至 2，试比较两种方案对系统性能的提高程度。

没有改进之前，每条指令的平时时钟周期CPI为：

$$CPI = \sum \left(CPI_i \times \frac{IC_i}{IC} \right)$$

$$= 4 \times 0.25 + 1.33 \times 0.75$$

$$\approx 2$$

第一种方案改进：

$$CPI = 2 - (20-2) \times 0.02$$

$$= 1.64$$

第二种方案改进：

$$CPI = 2 - (4-2) \times 0.25$$

$$= 1.5$$

∴ 第二种方案要优于第一种方案

1.2.2 计算机系统设计者的主要任务

目标：设计出能满足用户的功能需求、有较长的生命周期、且又具有很高的性能价格比的系统。

主要任务：

1. 指令系统的设计
2. 数据表示的设计
3. 功能的组织
4. 逻辑设计以及物理实现

设计计算机系统所要完成的3个方面的工作：

1. **确定用户对计算机系统的功能、价格和性能的要求**
   功能需求：根据市场的需要以及所设计系统的应用领域来确定具体可以考虑以下功能需求
   1. 应用领域

2. 软件兼容

   软件兼容是指一台计算机上的程序不加修改就可以搬到另一台计算机上正常运行

3. 操作系统需求

   包括地址空间大小、存储管理、保护等。从系统结构上对操作系统的需求提供支持，是很重要的一点。

4. 标准

   确定系统中哪些方面要采用标准以及采用什么标准。如：浮点数标准、I/O总线标准、程序设计语言标准等。

2. **软硬件功能分配**

   软件和硬件在实现功能上是 等价的

   - 用软件实现

     优点：设计容易、修改简单，而且可以减少硬件成本。

     缺点：所实现的功能的速度较慢。

   - 用硬件实现

     优点：速度快、性能高，

     缺点：修改困难，灵活性差

3. **设计出生命周期长的系统结构**

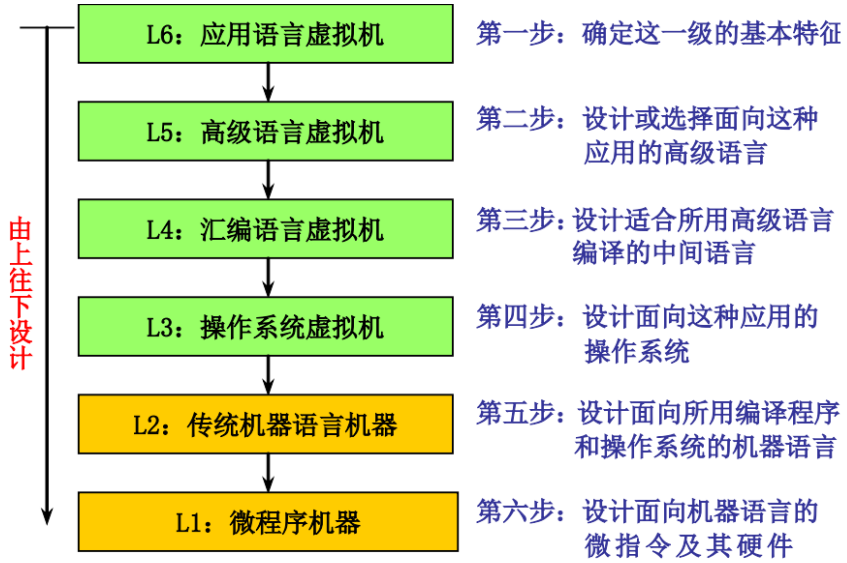
   - 特别注意**计算机应用**和**计算机技术**的发展趋势
   - 设计出具有一定前瞻性的系统结构，以使得它具有较长的生命周期

1.2.3 计算机系统设计的主要方法

由上往下的设计（top-down）

适合于专用机的设计，而不适合通用机的设计。

从层次结构中的最上面一级开始，逐层往下设计各层的机器



首先确定面对使用者的那级机器的基本特征、数据类型和格式、基本命令等

然后再逐级往下设计，每级都考虑如何优化上一级的实现

由下往上的设计（bottom-up）

在早期被采用得比较多，现在已经很少被采用了

从层次结构的最下面一级开始，逐层往上设计各层的机器。

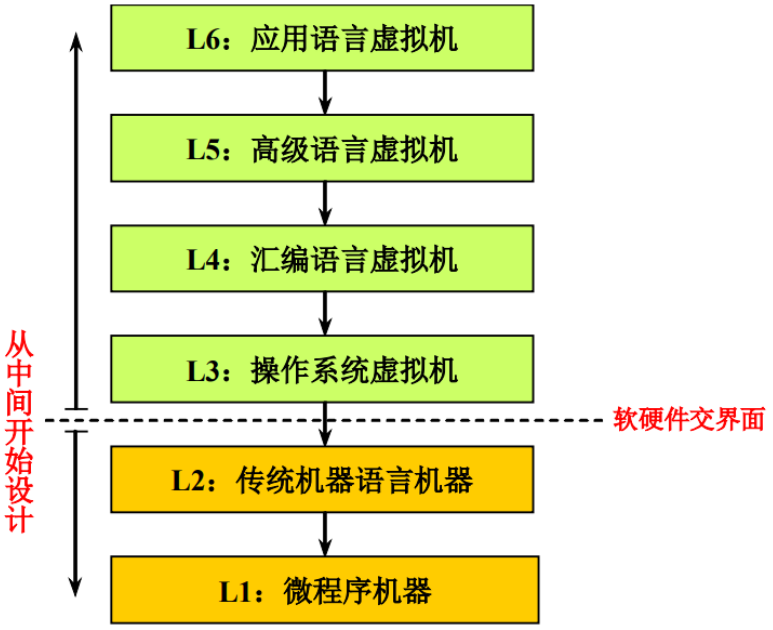
采用这种方法时，软件技术完全处于被动状态，这会造成软件和硬件的脱节

使整个系统的效率降低

从中间开始设计（middle-out）

“由上往下”和“由下往上”设计方法的主要缺点----**软、硬件设计分离和脱节**

解决方法：综合考虑软、硬件的分工，从中间开始设计。



“中间”：层次结构中的软硬件的交界面，目前一般是在**传统机器语言机器级**与**操作系统机器级**之间。

1.3 计算机系统的性能评测

执行时间和吞吐率

1. 单个程序的执行时间（执行单个程序所花的时间很少）
2. 吞吐率（在单位时间里能够完成的任务很多 ）

执行时间

计算机完成某一任务所花费的全部时间，包括磁盘访问、存储器访问、输入/输出、操作系统开销等。

CPU时间： CPU执行所给定的程序所花费的时间，不包含I/O等待时间以及运行其它程序的时间。
用户CPU时间:用户程序所耗费的CPU时间。
系统CPU时间：用户程序运行期间操作系统耗费的CPU时间。

基准测试程序

测试程序的分类

- 真实程序

- 修正的（脚本化）应用程序
- 核心程序
- 小测试程序
- 合成测试程序

测试程序包

参考网站：测试程序包

选择一组各个方面有代表性的测试程序组成尽可能全面地测试了一个计算机系统的性能

- **SPEC测试程序包** 最成功和最常见的测试程序套件
- TPC-x
- EEMBC测试程序包

1.4 计算机系统结构的发展

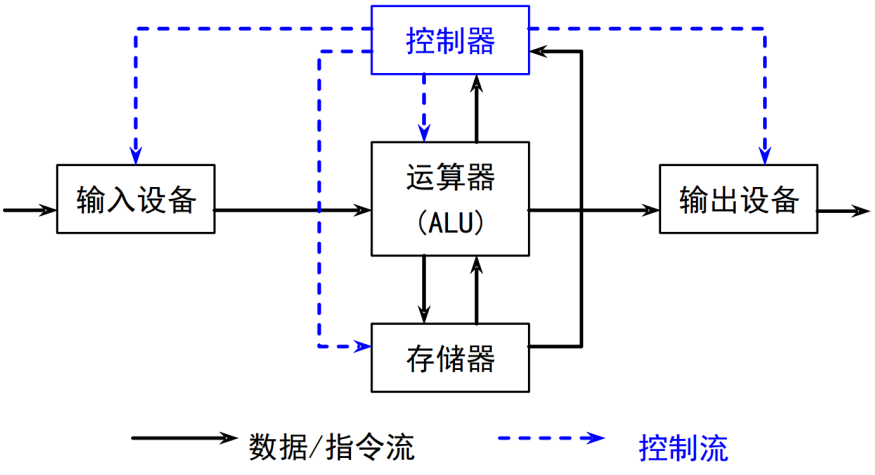
存储程序的计算机

冯诺依曼计算机的四个组成部分 ☆

- **运算器**：计算机的主要部件，负责数值计算
- **存储器**：用于存储 数据和程序
- **输入/输出设备**：完成计算机和外部的数据交互
- **控制器**：根据程序形成控制序列（例如：指令、命令），指导硬件去完成对数据的运算

存储程序计算机的主要特点

1. 机器以运算器为**中心**：
2. 采用**存储程序**的原理（也就是将程序也看成数据存储在存储器中）
3. 存储器按**地址**访问线性编址的地址空间来获取程序或数据
4. 控制流由指令流产生（所以，解题的算法是具有顺序性的）
5. 指令由**操作码**和**地址码**组成
6. 数据采用二进制编码



程序的执行过程

1. 分解程序指令，从而形成控制计算机四个主要部分进行工作的**控制流**
2. 在控制流的控制下，对数据进行加工运算形成数据流
3. 不断循环产生指令流与数据流
4. 直到得到程序的结果

一个机器周期里面安排的操作序列

- **取指**：
  指令取指是指将指令从存储器中读取出来的过程。
- **译码**：
  指令译码是指将存储器中取出的指令进行翻译的过程。
  将操作码和地址码分离出来，以便指令执行。
- **执行**：
  指令译码之后所需要进行的计算类型都已得知，并且已经从通用寄存器组中读取出了所需的操作数，那么接下来便进行指令执行。
  指令执行是指对指令进行真正运算的过程。
  在“执行”阶段的最常见部件为**算术逻辑部件运算器ALU**，作为实施具体运算的硬件功能单元。
- **访存**：
  访存是指存储器访问指令将数据从存储器中读出，或者写入存储器的过程。
- **写回**：
  写回是指将指令执行的结果写回通用寄存器组的过程。
  如果是普通运算指令，该结果值来自于“执行”阶段计算的结果；如果是存储器读指令，该结果来自于“访存”阶段从存储器中读取出来的数据。

软件对系统结构的影响 ☆

系列机

系列机（ family machine ）是**具有相同体系结构**，但**组成和实现不同**的一系列不同型号的计算机系统

各计算机厂家仍按系列机研发产品

现代计算机不但系统系列化，其构成部件和软件也系列化

也可以这样定义：一种指令集结构可以有多种组成。同样，一种组成可以有多种物理实现。系列机就是指在一个厂家生产的具有**相同的指令集结构**，但**具有不同组成和实现的一系列不同型号的机器**。

计算机	时间	处理器	字宽	主要 I/O 总线	存储空间
PC 和 PC XT	1981	8088	16 位	PC 总线	20 位
PC AT	1982	80286	16 位	AT （ ISA ）	24 位
80386 PC	1985	80386	32 位	ISA/EISA	32 位
80486 PC	1989	80486	32 位	ISA+VL	32 位
Pentium PC	1993	Pentium	32 位	ISA+PCI	32 位
Pentium II PC	1997	Pentium II	32 位	ISA+PCI+AGP	32 位
Pentium III PC	1999	Pentium III	32 位	PCI+AGP +USB	32 位
Pentium 4 PC	2000	Pentium 4	32 位	PCI-X+AGP +USB	32 位

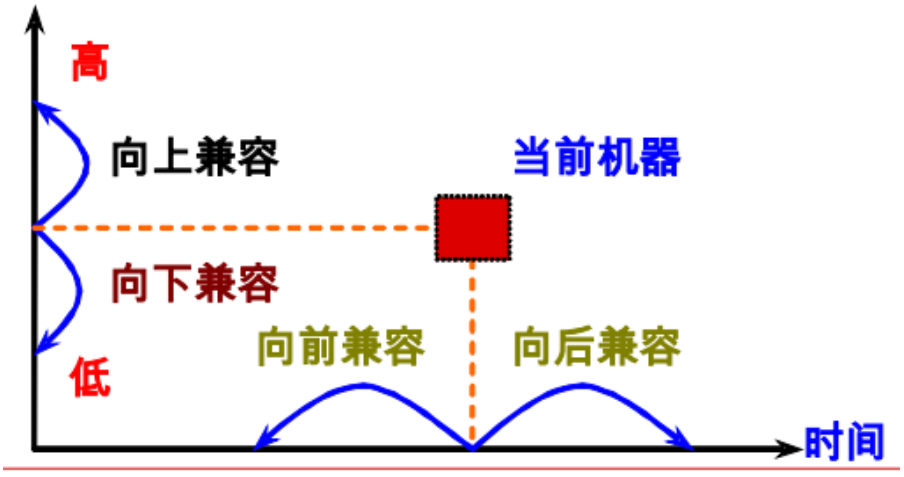
软件兼容

系列机具有相同的体系结构，软件可以在该种系列计算机的各档机器上运行，也就是说同一个软件可以不加修改地运行于**体系结构相同**的各档机器，而且它们所**获得的结果一样**，差别只在于有不同的运行时间

兼容机

不同厂家生产的具有相同体系结构的计算机，计算机厂家为了能利用大的计算机厂家的开发成果，采用新的计算机组织和实现技术，研制一些软件兼容的产品，具有更加低廉的价格

- **向上（下）兼容**
  - 指的是按某档机器编制的程序，不加修改的就能运行于比它高（低）档的机器
- **向前（后）兼容**
  - 指的是按某个时期投入市场的某种型号机器编制的程序，不加修改地就能运行于在它之前（后）投入市场的机器



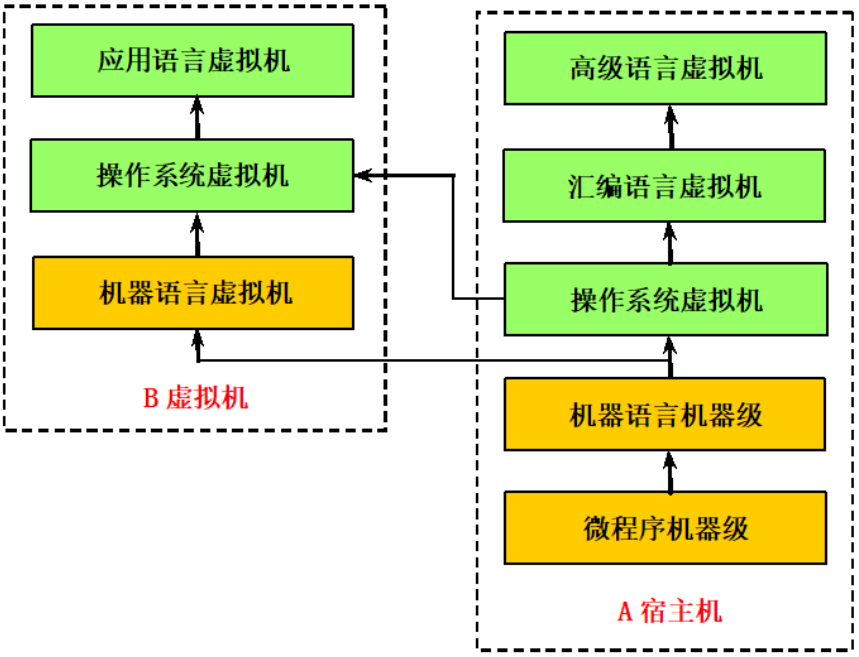
向后兼容是才是软件兼容的根本特征，也是系列机的根本特征

模拟与仿真 ☆

模拟

使软件能在具有**不同系统结构的机器之间相互移植。**

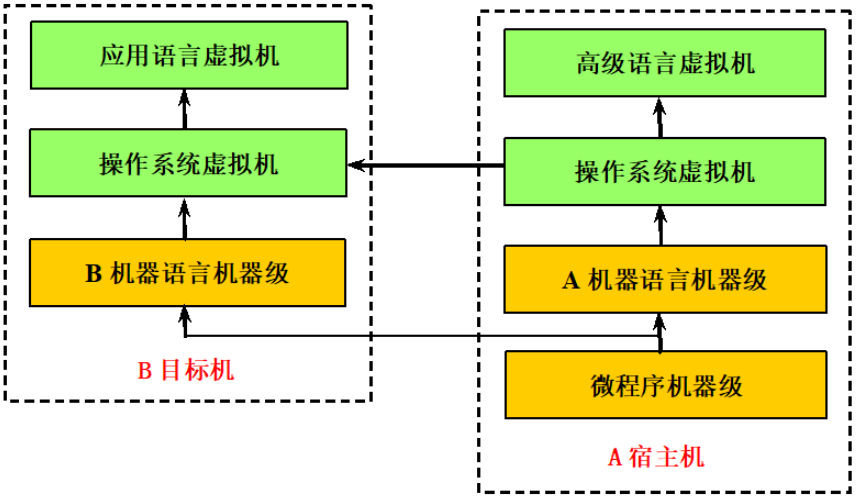
- 在一种系统结构上实现另一种系统结构。
- 从指令集的角度来看，就是要在一种机器上实现另一种机器的指令集。



仿真

用一台现有机器（宿主机）上的**微程序去解释**实现另一台机器（目标机）的指令集。

- 运行速度比模拟方法的**快**
- 仿真只能在**系统结构差距不大**的机器之间使用

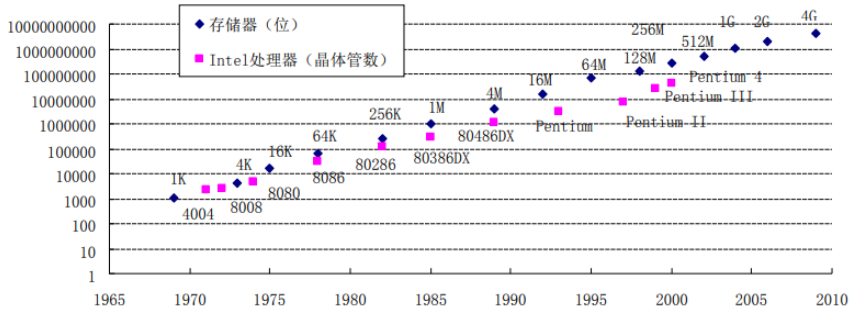


	模拟	仿真
所处层次	机器语言机器级	微程序机器级
执行方式	主存中的机器语言程序解释执行	控制存储器中的微程序解释执行
实行速度	慢	快
适用范围	–	系统结构差距不大的计算机之间

器件发展对系统结构的影响

集成电路的发展和摩尔定律

集成电路密度大约每两年翻一番



应用对系统结构的影响

- 不同的应用对计算机系统结构的设计提出了不同的要求。
- 应用需求是促使计算机系统结构发展的**最根本的动力**。
- 一些特殊领域：需要高性能的系统结构

1.5 计算机系统结构中并行性的发展 ☆

并行性概念

计算机系统在同一时刻或者同一时间间隔内进行多种运算或操作。

从数据处理的角度看并行的等级

- **字串位串**：每次只对一个字的一位进行处理。
  最基本的串行处理方式，不存在并行性。

- **字串位并**：同时对一个字的全部位进行处理，不同字之间是串行的。开始出现并行性。
- **字并位串**：同时对许多字的同一位（称为位片）进行处理。具有较高的并行性。
- **全并行**：同时对许多字的全部位或部分位进行处理。最高一级的并行。

从执行程序的角度看并行的等级

- **指令内部并行**：单条指令中各微操作之间的并行。
- **指令级并行**：并行执行两条或两条以上的指令。
- **线程级并行**：并行执行两个或两个以上的线程。通常是以一个进程内派生的多个线程为调度单位。
- **任务级或过程级并行**：并行执行两个或两个以上的过程或任务（程序段）以子程序或进程为调度单元。
- **作业或程序级并行**：并行执行两个或两个以上的作业或程序。

提高并行性的方法 ☆

1. 时间重叠
   让多个处理过程在时间上相互错开，轮流重叠地使用同一套硬件设备的各个部分
2. 资源重复
   通过重复设置硬件资源，大幅度地提高计算机系统的性能
3. 资源共享
   这是一种**软件方法**，它使多个任务按一定时间顺序轮流使用同一套硬件设备。

并行性的发展

单机系统-并行性的发展

在发展高性能单处理机过程中，起主导作用的是 时间重叠原理。

实现时间重叠的基础：**部件功能专用化**

还有资源重复利用的原理：

- 多体存储器
- 多操作部件
- 阵列处理机

在单处理机中，资源共享的概念实质上是用**单处理机模拟多处理机的功能**，形成所谓虚拟机的概念----分时系统

多机系统中并行性的发展

多机系统遵循**时间重叠**、**资源重复**、**资源共享原理**，发展为3种不同的多处理机：**同构型多处理机**、**异构型多处理机**、**分布式系统**

耦合度

反映多机系统中各机器之间物理连接的紧密程度和交互作用能力的强弱

- 紧密耦合系统（直接耦合系统）
  - 一般是通过总线或高速开关互连，可以共享主存。

- 松散耦合系统（间接耦合系统）
  - 一般是通过通
    道或通信线路实现计算机之间的互连，可以共享外存设备（磁盘、磁带等）。机器之间的相互作用是在**文件或数据集**一级上进行。

功能专用化

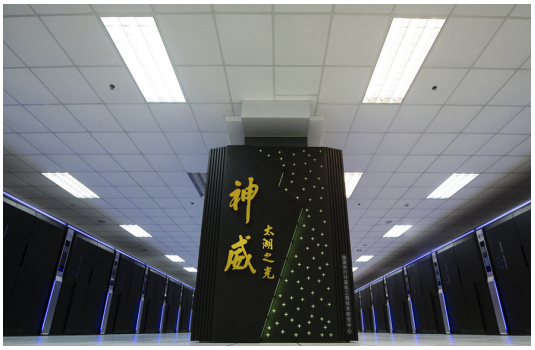
- 专用外围处理机
  例如：输入/输出功能的分离
- 专用处理机
  如数组运算、高级语言翻译、数据库管理等，分离出来。
- 异构型多处理机系统
  由**多个不同类型、至少担负不同功能的处理机**组成，它们按照作业要求的顺序，利用时间重叠原理，依次对它们的多个任务进行加工，各自完成规定的功能动作。

机间互连

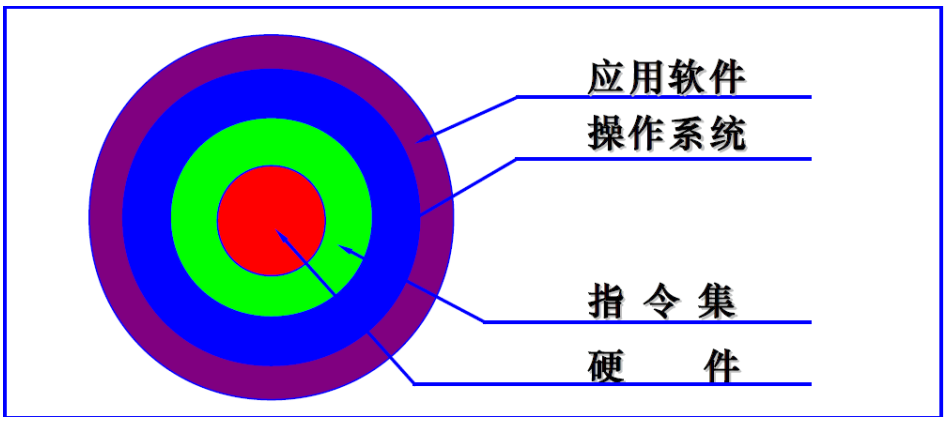
- 容错系统
- 可重构系统
  高带宽、低延迟、低开销的机间互连网络是高效**实现程序或任务一级并行处理**的前提条件。
- 同构型多处理机系统
  由**多个同类型或至少担负同等功能的处理机**组成，它们同时处理同一作业中能并行执行的多个任务。

并行机的发展

1. 并行机的萌芽阶段（1964年 ~ 1975年）
   在处理器中使用**流水线和重复设置功能单元**
2. 向量机的发展和鼎盛阶段（1976年 ~ 1990年）
   向量计算机的发展呈两大趋势
   - 提高单处理器的速度
   - 研制多处理器系统
3. MPP出现和蓬勃发展阶段（1990年 ~ 1995年）
4. 各种体系结构并存阶段（1995年 ~ 2000年）
   PVP（并行向量处理机）、MPP、SMP、DSM（分布式共享存储多处理机）、COW（工作站机群）等各种体系结构进入并存发展的阶段。
5. 机群蓬勃发展阶段（2000年以后）
   机群系统：将一群工作站或高档微机用某种结构的互连网络互连起来，充分利用其中各计算机的资源，统一调度、协调处理，以达到很高的峰值性能，并实现高效的并行计算。



指令系统是计算机系统结构的主要内容，是软硬件交界面的主要部分。



2.1 指令系统结构的分类

区别不同指令系统结构的主要因素：CPU中用来存储操作数的数据单元类型

存储操作数的存储单元类型

1. **堆栈**
2. **累加器**
3. **通用寄存器组**

指令系统的结构类型 ☆

1. 堆栈结构

堆　栈
push X
push Y
add
pop　Z

2. 累加器结构

累加器
load　X
add　　Y
store　Z

3. 通用寄存器结构

寄存器（RM型）	寄存器（RR型）
load R1, X	load R1, X
add R1, Y	load R2, Y
store R1, Z	add R3, R1, R2
	store R3, Z

根据操作数的来源不同，又可以进一步分为:

1. 寄存器-存储器结构（RM结构）
    - 操作数可以来自存储器
2. 寄存器-寄存器结构（RR结构）
    - 所有的操作数都是来自通用寄存器组
    - 该结构也被称为**load-store结构**，只有load指令和store指令才能够访问存储器，其它指令无法直接从存储器中获得操作数，只能从通用寄存器组中获得操作数。

显式给出vs隐式给出

对于不同类型的结构，操作数的位置、个数以及操作数的**给出方式**（显式或隐式）也会不同

显式给出

用 指令字中的操作数字段 给出

比如，直接给出操作数的值

隐式给出

使用事先约定好的单元

- **堆栈型结构中，操作数隐式给出**，栈顶两个元素被弹出参与运算，计算结果被压入栈中
- **累加器型结构中，累加器内的操作数是隐式的，另一个操作数是显式给出的**
  比如：add X 其中累加器中的数据是隐式的，X是显式给出的。
- 在**通用寄存器型结构**中，所有的操作数都是**显式**给出的

比如：表达式$Z = X + Y$在4种类型指令系统结构上的代码。假设：X、Y、Z均保存在存储器单元中，并且不能破坏X和Y的值。

堆 栈	累加器	寄存器（RM型）	寄存器（RR型）
push X	load X	load R1, X	load R1, X
push Y	add Y	add R1, Y	load R2, Y
add	store Z	store R1, Z	add R3, R1, R2
pop Z			store R3, Z

由上面可以看出，堆栈型加法已经事先知道了操作数的位置，也就是栈顶前两个元素，所以加法指令就是ADD，因此，该操作数就是隐式给出。对其它类型的结构，分析也是如此

堆栈型VS累加器型VS通用寄存器型

- **堆栈型**
  - 优点：指令字比较短，程序占用的空间比较小
  - 缺点：不能随机访问堆栈，难以生成有效的代码
- **累加器型**
  - 优点：指令字比较短，程序占用的空间比较小
  - 缺点：因为只有一个中间结果暂存器（累加器），因此需要频繁的访问存储器，降低了效率
- **通用寄存器型----现代指令系统结构的主流**
  - 在灵活性和提高性能方面有明显的优势：
    1. 寄存器的访问速度比存储器快
    2. 对编译器而言，能更加容易、有效地分配和使用寄存器
    3. 寄存器可以用来存放变量从而可以：
       - 减少对存储器的访问，加快程序的执行速度；
       - 用更少的地址位来对寄存器进行寻址，从而减少程序的目标代码的大小。（利用寄存器地址代替存储器地址，找到数据所需要的地址信息更少）

利用ALU指令的操作数的两个特征对通用寄存器结构进一步细分

ALU指令的操作数个数

- **3个操作数的指令**
  两个源操作数和一个目标操作数
- **2个操作数的指令**
  其中一个操作数既可以作为源操作数也可以作为目的操作数

ALU指令中存储器操作数的个数

可以是0 ~ 3 中的某一个，为0表示没有存储器操作数。

ALU指令中操作数个数和存储器操作数个数的典型组合

ALU指令中存储器操作数的个数	ALU指令中操作数的最多个数	结构类型	机器实例
0	3	RR	MIPS，SPARC，Alpha，PowerPC，ARM
1	2	RM	IBM 360/370，Intel 80x86，Motorola 68000
	3	RM	IBM 360/370
2	2	MM	VAX
3	3	MM	VAX

通用寄存器型结构进一步细分为3种类型

- 寄存器 - 寄存器型（RR型）
- 寄存器 - 存储器型（RM型）
- 存储器 - 存储器型（MM型）

优缺点对比：（m,n）表示指令的n个操作数中有m个存储器操作数

- **寄存器 - 寄存器型（RR型）**
  - 典型代表：RISC 精简指令集
  - （0，3）
  - **优点**：指令字长固定，指令结构简洁，是一种简单的代码生成模型，各种指令的执行时钟周期数相近。
  - **缺点**：与指令中含存储器操作数的指令系统结构相比，指令条数多，目标代码不够紧凑，因而程序占用的空间比较大。
- **寄存器 - 存储器型（RM型）**
  - 典型代表：CISC 复杂指令集
  - （1，2）
  - **优点**：可以在ALU指令中直接对存储器操作数进行引用，而不必先用load指令进行加载。容易对指令进行编码，目标代码比较紧凑。
  - **缺点**：一个操作数的内容将被改写，因此指令中的两个操作数不对称。
    在一条指令中同时对寄存器操作数和存储器操作数进行编码，有可能限制指令所能够表示的寄存器个数。指令的执行时钟周期数因操作数的来源（R/M）不同而差别比较大。
- **存储器-存储器型（MM型）**
  - （2，2）/（3，3）
  - **优点**：目标代码最紧凑，不需要设置寄存器来保存变量。
  - **缺点**：指令字长变化很大，特别是3操作数指令。而且每条指令完成的工作也差别很大。对存储器的频繁访问会使存储器成为瓶颈。**这种类型的指令系统结构现在已不用了。**

2.2 寻址方式

2.2.1 概念

寻址方式：指令系统中如何形成要访问的数据的地址。

在通用寄存器型指令集结构中，一般是利用**寻址方式**指明指令中的操作数是一个**常数**、一个**寄存器操作数**、或者是一个**存储器操作数**。

- 寻址实际上是从形式地址到实际地址的转换。
- **形式地址由指令描述，实际地址也称为有效地址。**
- 有效地址指明的是存储器单元的地址或寄存器地址。

2.2.2 一些操作数的寻址方式

说明

- ←：赋值操作
- Mem：存储器
- Regs：寄存器组
- 方括号：表示内容

例如：

- Mem[ ]：存储器的内容
- Regs[ ]：寄存器的内容
- Mem[Regs[R1]]：以寄存器R1中的内容作为地址的存储器单元中的内容

各种寻址方式举例

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
偏移寻址	ADD R3 , 120(R2)	Regs[R3]←Regs[R3] + Mem[120+Regs[R2]]
寄存器间接寻址	ADD R4 , (R2)	Regs[R4]←Regs[R4] + Mem[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 R4 , (1010)	Regs[R4]←Regs[R4] + Mem[1010]
存储器间接寻址	ADD R2 , @(R4)	Regs[R2]←Regs[R2] + Mem[Mem[Regs[R4]]]
自增寻址	ADD R1 , (R2)+	Regs[R1]←Regs[R1] + Mem[Regs[R2]] Regs[R2]←Regs[R2] + d
自减寻址	ADD R1, -(R2)	Regs[R2]←Regs[R2] - d Regs[R1]←Regs[R1]+Mem[Regs[R2]]
缩放寻址	ADD R1 , 80(R2)[R3]	Regs[R1]←Regs[R1] + Mem[80 + Regs[R2] + Regs[R3]*d]

采用多种寻址方式可以显著地减少程序的指令条数，但可能增加计算机的实现复杂度以及指令的CPI。

各种寻址方式的使用情况统计结果

只需要注意：

- **立即数寻址方式**和**偏移寻址方式**的使用频率最高
- 大约1/4的**load指令**和**ALU指令**采用了立即数寻址

两种表示寻址方式的方法

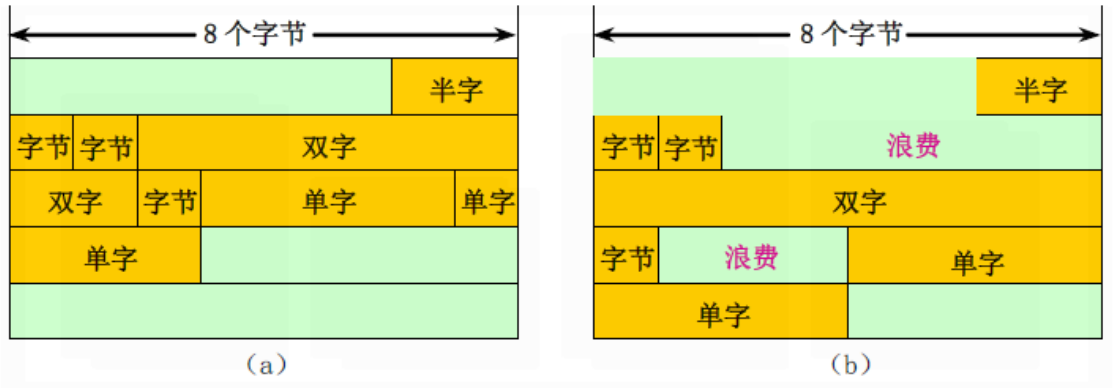
1. 将**寻址方式编码于操作码**中，由操作码描述相应操作的寻址方式。
    - 比如： 寻址方式只有很少的几种的结构：load-store结构
2. 在**指令字中设置专门的寻址字段**，用以直接指出寻址方式。
    - 处理机具有多种寻址方式，且指令有多个操作数。

物理地址空间的信息如何存放？ ☆

信息宽度不超过主存宽度的信息必须存放在一个存储字内，不能跨边界。

也就是说：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**

虽然会存在存储空间的浪费，但是可以保证访问的速度



满足以下条件

- 字节信息的起始地址为：×…××××
- 半字信息的起始地址为：×…×××0
- 单字信息的起始地址为：×…××00
- 双字信息的起始地址为：×…×000

2.3 指令系统的设计和优化

2.3.1 指令系统的设计

首先考虑所应实现的基本功能，确定哪些基本功能应该由硬件实现，哪些功能由软件实现比较合适。

- 指令的功能设计
- 指令格式的设计

2.3.2 对指令系统的基本要求 ☆

1. **完整性**
   - 在一个有限可用的存储空间内，对于任何可解的问题，编制计算程序时，**指令系统所提供的指令足够使用**。要求指令系统功能齐全、使用方便
   - 指令类型

操作类型	实　例
算术和逻辑运算	算术运算和逻辑操作：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

   其中，算数和逻辑运算、数据传输、控制、系统这4种类型属于通用计算机系统的基本指令
2. **规整性**
   1. 对称性：所有与指令系统有关的存储单元的使用、操作码的设置等都是对称的。
   2. 均匀性：指对于各种不同的操作数类型、字长、操作种类和数据存储单元，指令的设置都要同等对待。

      例如：如果某机器有5种数据表示，4种字长，2种存储单元，则要设置5×4×2=40种同一操作的指令。（一般情况下，都是实现有限的规整性，不会都去实现）
3. **正交性**
   - 指令在编码时，应该是**互相独立，互不相关**的
4. **高效率**
   - 指令执行速度快，使用频率高
5. **兼容性**
   - 主要实现 向后兼容 指令系统可以增加新指令，但不能删除指令或者更改指令

2.3.3 指令系统的两种设计策略

- **CISC（复杂指令系统计算机）**
  - 增强指令功能，把越来越多的功能交由硬件来实现，并且指令的数量也是越来越多。（但Intel在奔4之后就开始采取微内核架构，将内核设置为精简指令集）
- **RISC（精简指令系统计算机）**
  - 尽可能地把指令系统简化，不仅指令的条数少，而且指令的功能也比较简单。

2.3.4 控制指令 ☆

1. 定义：控制指令是用来改变控制流的
   - **跳转**：当指令是**无条件**改变控制流时，称之为**跳转指令**。

- **分支**：当控制指令是**有条件**改变控制流时，则称之为**分支指令**。

2. 能够改变控制流的指令
    1. 分支（branch）
    2. 跳转（jump）
    3. 过程调用（call）
    4. 过程返回（return）

    注意：改变控制流的大部分指令是**分支指令（条件转移）**
3. 分支（branch）条件的方法及其优缺点

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（即CC）	可以自由设置分支条件	条件码是增设的状态。而且它限制了指令的执行顺序，因为要保证条件码能顺利地传送给分支指令。
条件寄存器	比较指令把比较结果放入任何一个寄存器，检测时就检测该寄存器。	简单	占用了一个寄存器
比较与分支	比较操作是分支指令的一部分，通常这种比较是受到一定限制的。	用一条指令（而不是两条）就能实现分支	当采用流水方式时，该指令的操作可能太多，在一拍内做不完。

4. 转移目标地址的表示
    - 通常**在指令中显示给出目标地址**
    - 过程返回指令不同，在编译时不知道返回地址（不知道程序被加载到哪里）。
    - 所以，在指令中提供一个偏移量，由该偏移量和程序计数器（PC）的值相加而得出目标地址。（**基于PC相对寻址**）
        - 优点：

          有效地减少表示该目标地址所需要的位数（离PC近）

          位置无关（代码可被装载到主存的任意位置执行）
        - 缺点：

          模拟结果表明：采用**4 ~ 8位的偏移量字段**（以指令字为单位）就能表示大多数控制指令的转移目标地址了。
5. 过程调用和返回

- 改变控制流，保存机器状态，保存返回地址
- 过去有些指令系统结构提供了**专门的保存机制**来保存许多寄存器的内容
- 现在较新的指令系统结构则要求由**编译器生成load和store指令来保存或恢复寄存器的内容**。

2.3.5 指令操作码的优化

$$指令 = 操作码 + 地址码$$

指令格式的设计：确定指令字的编码方式，包括操作码字段和地址码字段的编码和表示方式。

指令格式的优化：如何用**最短的位数**来表示指令的操作信息和地址信息。

1. 哈夫曼编码 ☆

- **基本思想**
  - 当各种事件发生的概率不均等时，可以**对发生概率最高的事件用最短的位数（时间）来表示（处理）**，而对于出现概率较低的事件，则可以用较长的位数（时间）来表示（处理），从而使总的平均位数（时间）缩短。
- **构造哈夫曼树的方法**
  1. 将各事件按其使用频度**从小到大**依次排列
  2. 每次从中选择两个频度值最小的结点，将其合并成一个新的结点，并把新结点画在所选结点的上面，然后用两条边把新结点分别与那两个结点相连。（注意：新结点的频度值是所选两个结点的频度值的和）
  3. 把新结点与其他剩余未结合的结点一起，再以上面的步骤进行处理，反复进行，直到全部结点都结合完毕、形成根结点为止。
- **信息熵**
  - 用来描述操作码的优化程度，越小越好。
  - 也就是说，当信息熵越大时，表示所含信息混乱程度就越大，为了使得用最短的位数表示指令的操作信息和地址信息，所以熵值越小越好
- **代码的优化程度**：H
  - $$H = -\sum_{i=1}^{n} p_i \log_2 p_i$$
  - 其中H表示为信息熵，p为每个指令的概率，n为指令总数。
  - 也可以表示为：理论上的编码最短码长
- **有关习题**
  - 例题：

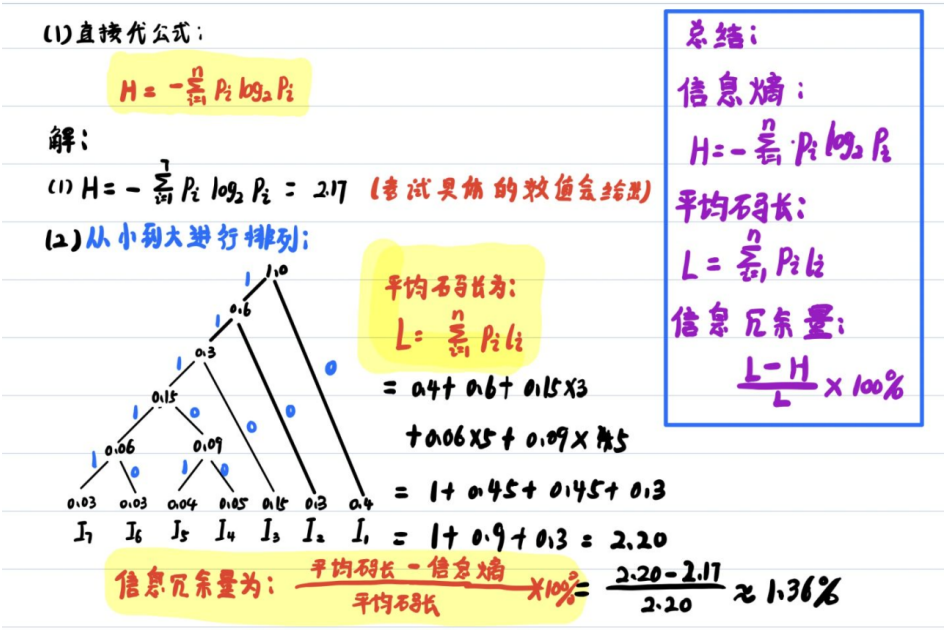
例2.1 假设某模型机有7条指令，这些指令的使用频度如表左边所示。

（1）计算这7条指令的操作码编码的最短平均码长；

（2）画出哈夫曼树，写出这7条指令的哈夫曼编码，并计算该编码的平均码长和信息冗余量。

指令	频度p_i	操作码使用哈夫曼编码	操作码长度l_i	利用哈夫曼概念的扩展操作码	操作码长度l_i
I_1	0.40	0	1	0 0	2
I_2	0.30	1 0	2	0 1	2
I_3	0.15	1 1 0	3	1 0	2
I_4	0.05	1 1 1 0 0	5	1 1 0 0	4
I_5	0.04	1 1 1 0 1	5	1 1 0 1	4
I_6	0.03	1 1 1 1 0	5	1 1 1 0	4
I_7	0.03	1 1 1 1 1	5	1 1 1 1	4

- 解题步骤：



  约定编码方法：左边是1 右边是0

- 哈夫曼编码的优缺点
  - 优点：可以减少操作码的平均位数
  - 缺点：所获得的编码是变长度的，不规整，不利于硬件处理。
- 扩展操作码
  - 位于**定长二进制编码和哈夫曼编码之间**的一种编码方案
  - 扩展操作码：

操作码使用哈夫曼编码	操作码长度l_i	利用哈夫曼概念的扩展操作码	操作码长度l_i
0	1	0 0	2
1 0	2	0 1	2
1 1 0	3	1 0	2
1 1 1 0 0	5	1 1 0 0	4
1 1 1 0 1	5	1 1 0 1	4
1 1 1 1 0	5	1 1 1 0	4
1 1 1 1 1	5	1 1 1 1	4

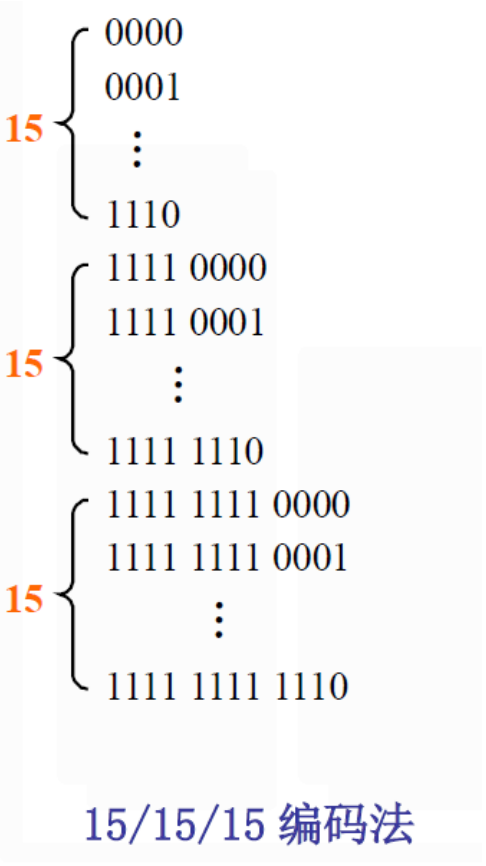
  - 这种方法的出来的结果要比哈夫曼编码大，但是编码长度较为工整，而且比定长的三位编码要小很多。（定长3位：3//哈夫曼编码：2.20//扩展操作码：2.30）

2. 等长扩展码

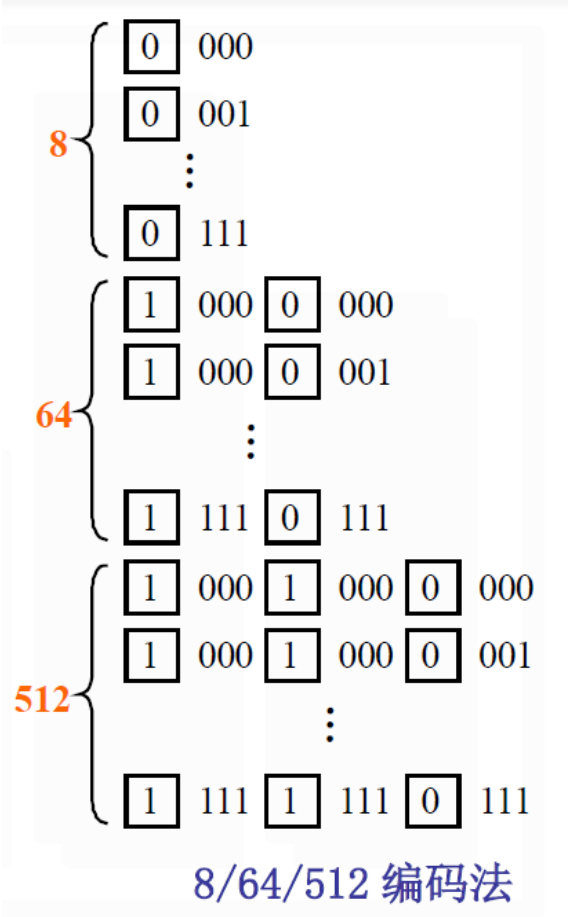
为了便于分级译码，一般都采用等长扩展码。

选用哪种编码法取决于指令使用频度pi的分布。

- **15**/15/15法
  若在**头15种**指令中pi 的值都比较大，但在后30种指令后急剧减少，则应选择15/15/15法；



- **8**/64/512法
  若pi的值在**头8种**指令中较大，之后的64种指令的pi值也不太低，则应选择8/64/512法。



衡量标准：看哪种编码法能使平均码长最短。

3. 定长操作码

固定长度的操作码：所有指令的操作码都是同一的长度（如8位）。

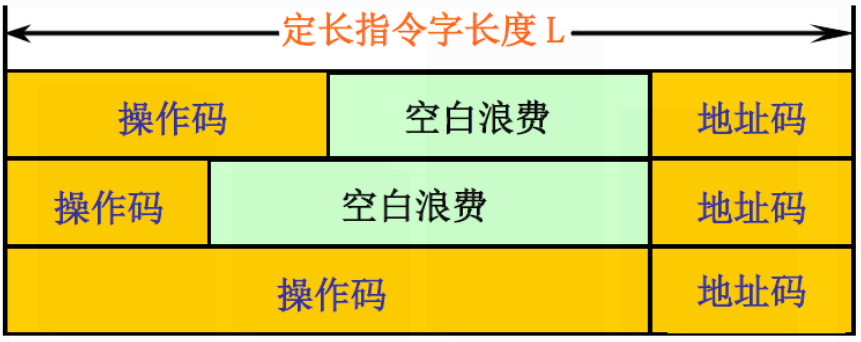
许多计算机都采用（特别是 RISC 结构的计算机） 精简指令集计算机

以程序的存储空间为代价来换取硬件实现上的好处。

2.3.6 指令字格式的优化

存在的问题：

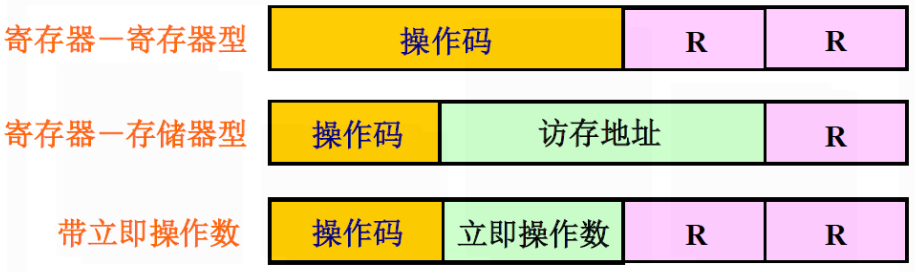
如果指令字的宽度固定，地址码的长度和个数固定，则操作码的缩短并不能带来好处，只是使指令字中出现空白浪费。



采用地址个数可变和/或地址码长度可变的方案

最常用的操作码最短，其地址字段个数最多。

能够使指令的功能增强，从总体上减少所需的指令条数。

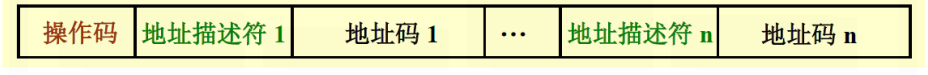


寄存器的个数以及**寻址方式**对指令字长也有很大的影响

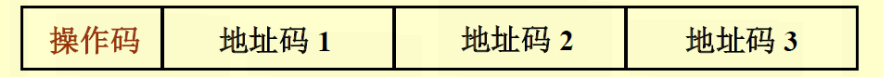
因此，在指令系统进行设计的过程中，要在指令字中选择合适的指令字长和寄存器的个数。

--------**>指令系统的三种编码格式(在操作码长度固定的情况下)** ☆

1. **可变长度编码**
   - 当指令系统的寻址方式和操作种类很多时，这种编码格式是最好的。
   - 用**最少的二进制位**来表示目标代码。
   - 可能会使各条指令的字长和执行时间**相差很大**
   - 比如：



2. **固定长度编码格式**
   - 将操作类型和寻址方式一起编码到操作码中
   - 当寻址方式和操作类型非常少时，这种编码格式非常好。
   - 可以有效地降低译码的复杂度，提高译码的速度。
   - 大部分 RISC的指令系统 均采用这种编码格式



3. **混合型编码格式**
   - 提供若干种固定的指令字长。
   - 以期达到既能够减少目标代码长度又能降低译码复杂度的目标。
   - 例如：

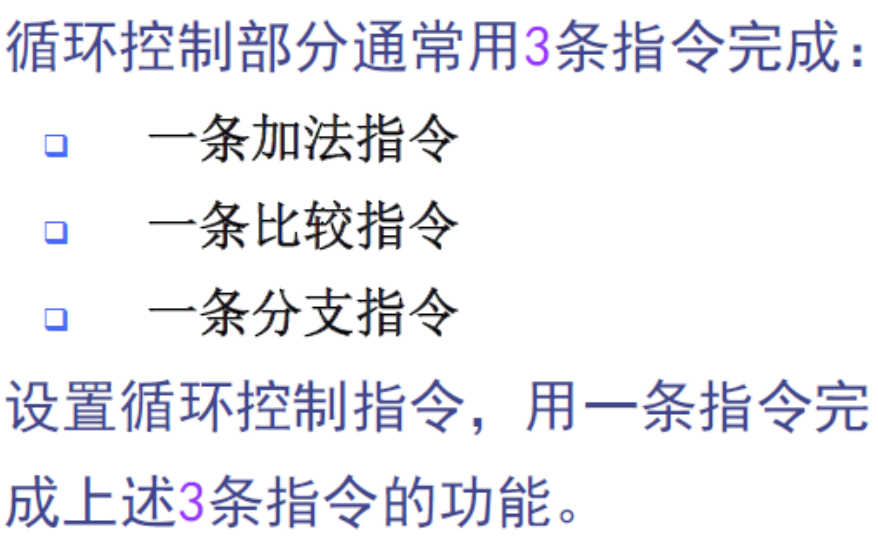
2.4 指令系统的发展和改进

2.4.1 沿CISC方向发展和改进指令系统 ☆

CISC指令系统的特点：指令数量多，功能多样

增强指令功能

1. 面向目标程序增强指令功能----根据应用优化
   - 对大量的目标程序及其执行情况进行统计分析，**找出那些使用频度高、执行时间长的指令或指令串。**
   - 对于使用频度高的**指令**，用**硬件加快**其执行；
   - 对于使用频度高的**指令串**，用一条**新的指令来替代。**
   - 改进方面：
     - 增强运算型指令的功能----处理
     - 增强数据传送指令的功能----I/O
     - 增强程序可控制指令的功能----控制
   - 举例：
     循环在程序中占有相当大的比例，所以可以在指令设计上提供专门的支持



2. 面向高级语言的优化实现来改进指令系统----编译过程优化
    - 缩小高级语言与机器语言的语义差距
    - 高级语言与一般的机器语言的语义差距非常大，为高级语言程序的编译带来了一些问题。
    - **增强对高级语言和编译器的支持**
        - 对高级语言中使用频度高、执行时间长的语句，增强有关指令的功能，加快这些指令的执行速度
        - 增加专门的指令，可以达到减少目标程序的执行时间和减少目标程序长度的目的。
        - 增强系统结构的规整性，减少系统结构中的各种例外情况。
    - **高级语言计算机（一种比较激进的方法）**
        - 间接执行高级语言机器
            - 高级语言作为机器的汇编语言。
        - 直接执行高级语言的机器
            - 直接把高级语言作为机器语言。
    - 比较简单的系统结构＋软件
        - 能够在较低成本和复杂度的前提下，提供更高的性能和灵活性。
3. 面向操作系统的优化实现改进指令系统
   操作系统和计算机系统结构是紧密联系的，**操作系统的实现在很大程度上取决于系统结构的支持。**
    - 指令系统对操作系统的支持主要有：
        - 处理机工作状态和访问方式的切换；
        - 进程的管理和切换；
        - 存储管理和信息保护；
        - 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，一般用户程序是不能使用的。

2.4.2 沿RISC方向发展和改进指令系统 ☆

CISC指令系统结构存在的问题

1. 各种指令的使用频率相差悬殊，许多指令很少使用
   使用频度高的指令也是最简单的指令。

Intel 80x86最常用的10条指令

执行频度排序	80x86指令	指令执行频度（占执行指令总数的百分比）
1	load	22%
2	条件分支	20%
3	比较	16%
4	store	12%
5	加	8%
6	与	6%
7	减	5%
8	寄存器－寄存器间数据移动	4%
9	调用子程序	1%
10	返回	1%
合 计		95%

2. 指令系统庞大，指令条数很多，许多指令的功能又很复杂，使得控制器硬件非常复杂。
3. 许多指令由于操作繁杂，其CPI值比较大，执行速度慢。采用这些复杂指令有可能使整个程序的执行时间反而增加。
4. 由于指令功能复杂，规整性不好，不利于采用流水技术来提高性能。

RISC机器遵循的原则

- **指令条数少、指令功能简单**
  - 只选取使用频度很高的指令，在此基础上补充一些最有用的指令
- 采用**简单而又统一的指令格式**，并减少寻址方式；
  - 指令字长都为32位或64位
- 指令的执行在**单个机器周期内完成**；
  - (采用流水线机制)
- **只有load和store指令才能访问存储器**
  - 其它指令的操作都是在寄存器之间进行
  - 采用load-store结构
- 大多数指令都采用硬连逻辑来实现
- 强调优化编译器的作用，为高级语言程序生成优化的代码
- **充分利用流水技术**来提高性能。

2.5 操作数的类型和大小

明确概念：

- **数据表示**：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
- **数据结构**：由软件进行处理和实现的各种数据类型

系统结构设计者要解决的问题： **如何确定数据表示**

表示操作数类型的方法

- 由指令中的**操作码指定操作数的类型**
- 给**数据加上标识**，由数据本身给出操作数类型（采用该方案的机器很少见）

操作数的大小

操作数的位数或字节数

主要大小：**字节**、**半字**、**字**、**双字**

1. 字符：**用ASCII码表示**，一个字节
2. 整数：用**二进制补码**表示，一个字节、半字、单字
3. 浮点数：**单精度浮点数**----单字、**双精度浮点数**----双字
    - 一般都采用IEEE 754浮点标准
4. 十进制操作数类型
    - 压缩十进制或二进制编码十进制（BCD码）
        - 用4位二进制编码表示数字0 ~ 9，并将两个十进制数字合并到一个字节中存储。
    - 非压缩十进制
        - 将十进制数直接用字符串来表示。

访问不同操作数大小的频率

spec基准程序对于单字和双字的数据访问具有较高的频度

一台32位的机器应该支持8、16、32位整型操作数以及32位和64位的IEEE 754标准的浮点操作数。

2.6 MIPS指令系统结构

“无内部互锁流水级的微处理器”(Microprocessor without interlocked piped stages)

MIPS的寄存器

- 32个64位通用寄存器
    - R0，R1，…，R31
    - 也称为整数寄存器
    - R0的值永远是0
- 32个64位浮点数寄存器
    - F0，F1，…，F31
    - 用来存放32个单精度浮点数（32位），也可以用来存放32个双精度浮点数（64位）。
    - 存储单精度浮点数（32位）时，只用到FPR的一半，其另一半没用。
- 一些特殊的寄存器

    - 它们可以与通用寄存器交换数据
    - 例如浮点状态寄存器：用来保存有关浮点操作结果的信息。

MIPS的数据表示

数据表示的类型

- 整数
    - 字节
    - 半字
    - 字
    - 双字
- 浮点数
    - 单精度浮点数32位
    - 双精度浮点数64位

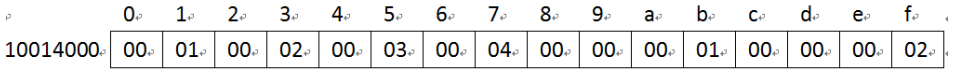
注意：

字节、半字或者字在装入64位寄存器时，用**零扩展或者用符号位扩展**来填充该寄存器的剩余部分。

装入以后，对它们将**按照64位整数**的方式进行运算。

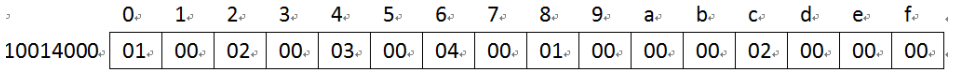
存储方式

- 大端存储



	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
10014000	00	01	00	02	00	03	00	04	00	00	00	01	00	00	00	02

- 小端存储



	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
10014000	01	00	02	00	03	00	04	00	01	00	00	00	02	00	00	00

寻址方式

1. 立即数寻址
2. 寄存器寻址
3. 偏移量寻址
    - 地址 = 立即数 + 偏移量
    - 立即数字段和偏移量字段都是16位的

- 寻址方式是编写到操作码中的
- MIPS的存储器是按字节寻址，地址为64位
- 所有存储器的访问都必须是边界对齐的

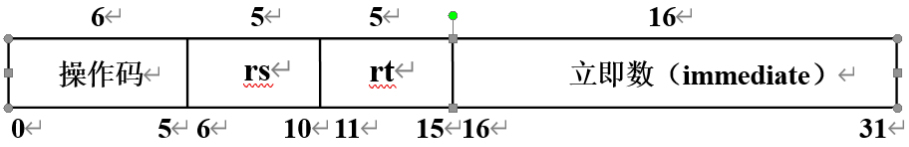
MIPS的指令格式 ☆

寻址方式编码到操作码中，同时所有的指令都是32位的，其中操作码占6位

分为3种指令格式：在3种指令格式中，同名字段的位置固定不变

1. **I类指令（指示类）**

- 包括所有的load和store指令，立即数指令，分支指令，寄存器跳转指令，寄存器链接跳转指令
- 立即数字段为16位，用于提供立即数或偏移量
- 比如：



  - 操作码：用于指定指令的操作类型（但没有funct域）
  - rs：指定第一个源操作数所在的寄存器编号
  - rt，指定用于目的操作数（保存运算结果）的寄存器编号
  - Immediate：16-bit的立即数
- 举例：

load指令

访存有效地址：Regs[rs]＋immediate

从存储器取来的数据放入寄存器rt

store指令

访存有效地址：Regs[rs]＋immediate

要存入存储器的数据放在寄存器rt中

立即数指令

Regs[rt] ← Regs[rs] op immediate

分支指令

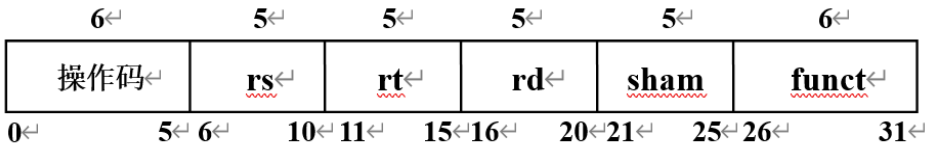
转移目标地址：Regs[rs]＋immediate，rt无用

寄存器跳转、寄存器跳转并链接

转移目标地址为Regs[rs]

2. **R类指令（运行类）**

- 包括ALU指令，专用寄存器读写指令，move指令等
- ALU指令
- 比如：



  - 操作码：用于指定指令的类型。对于R型指令，该域的值均为0
  - rs：第一个源操作数所在的寄存器编号；
  - rt：第二个源操作数所在的寄存器编号；
  - rd：目的操作数（保存运算结果）的寄存器编号；

- sham：移位指令进行移位操作的位数。
- funct:具体的运算操作编码,与操作码域组合，精确地指定指令的类型。

3. **J类指令**

- 包括跳转指令，跳转并链接指令，自陷指令，异常返回指令。
- 在这类指令中，指令字的低26位是偏移量，它与PC值相加形成跳转的地址。
- 比如：

MIPS的操作

分类

1. load和store
2. ALU操作
3. 分支与跳转
4. 浮点操作

符号的意义

$x \leftarrow_n y$：从y传送n位到x

$x, y \leftarrow z$：把z传送到x和y

- 下标：表示字段中具体的位；
  - 对于指令和数据，按从最高位到最低位（即从左到右）的顺序依次进行编号，最高位为第0位，次高位为第1位，依此类推。
  - 下标可以是一个数字，也可以是一个范围。

$Regs[R4]_0$：寄存器R4的符号位

$Regs[R4]_{56-63}$：R4的最低字节

- Mem :表示主存；
  按字节寻址，可以传输任意个字节。
- 上标:用于表示对字段进行复制的次数。
  - 例如：0^{32}：一个32位长的全0字段
- 符号##：用于两个字段的拼接，并且可以出现在数据传送的任何一边。
  -

举例：R8、R10：64位的寄存器，则

$Regs[R8]_{32-63} \leftarrow_{32} (Mem\ [Regs[R6]]_0)^{24}\ \#\#\ Mem\ [Regs[R6]]$

表示的意义是：

以R6的内容作为地址访问内存，得到的字节按符号位扩展为32位后存入R8的低32位，R8的高32位（即$Regs[R8]_{0-31}$）不变。

指令举例

1. load和store指令

指令举例	指令名称	含 义
LD R2，20(R3)	装入双字	$Regs[R2] \leftarrow_{64} Mem[20+Regs[R3]]$
LW R2，40(R3)	装入字	$Regs[R2] \leftarrow_{64} (Mem[40+Regs[R3]]_0)^{32}$ ## $Mem[40+Regs[R3]]$
LB R2，30(R3)	装入字节	$Regs[R2] \leftarrow_{64} (Mem[30+Regs[R3]]_0)^{56}$ ## $Mem[30+Regs[R3]]$
LBU R2，40(R3)	装入无符号字节	$Regs[R2] \leftarrow_{64} 0^{56}$ ## $Mem[40+Regs[R3]]$
LH R2，30(R3)	装入半字	$Regs[R2] \leftarrow_{64} (Mem[30+Regs[R3]]_0)^{48}$ ## $Mem[30+Regs[R3]]$## $Mem[31+Regs[R3]]$
L.S F2，60(R4)	装入单精度浮点数	$Regs[F2] \leftarrow_{64} Mem[60+Regs[R4]]$ ## 0^{32}
L.D F2，40(R3)	装入双精度浮点数	$Regs[F2] \leftarrow_{64} Mem[40+Regs[R3]]$
SD R4，300(R5)	保存双字	$Mem[300+Regs[R5]] \leftarrow_{64} Regs[R4]$
SW R4，300(R5)	保存字	$Mem[300+Regs[R5]] \leftarrow_{32} Regs[R4]$
S.S F2，40(R2)	保存单精度浮点数	$Mem[40+Regs[R2]] \leftarrow_{32} Regs[F2]_{0..31}$
SH R5，502(R4)	保存半字	$Mem[502+Regs[R4]] \leftarrow_{16} Regs[R5]_{48..}$

2. ALU指令
   - 寄存器 - 寄存器型（RR型）指令或立即数型
   - 算术和逻辑操作：加、减、与、或、异或和移位等

指令举例	指令名称	含义
DADDU R1，R2，R3	无符号加	$Regs[R1] \leftarrow Regs[R2] + Regs[R3]$
DADDIU R4，R5，#6	加无符号立即数	$Regs[R4] \leftarrow Regs[R5] + 6$
LUI R1，#4	把立即数装入到一个字的高16位	$Regs[R1] \leftarrow 0^{32}$ ## 4 ## 0^{16}
DSLL R1，R2，#5	逻辑左移	$Regs[R1] \leftarrow Regs[R2] << 5$
DSLT R1，R2，R3	置小于	If($Regs[R2] < Regs[R3]$) $Regs[R1] \leftarrow 1$ else $Regs[R1] \leftarrow 0$

- 注意：R0的值永远是0，它可以用来合成一些常用的操作

```
DADDIU R1，R0，#100
```

给寄存器R1装入常数100

```
DADD R1，R0，R2
```

把寄存器R2中的数据传送到寄存器R1

3. 控制指令

- 跳转：无条件跳转
  - 4种：跳转、跳转并链接、间接跳转、间接跳转并链接跳转
  - 目标地址由26位偏移量左移两位与PC相加间接跳转目标地址由指定寄存器给出的.
- 分支：条件转移
  - 分支条件由指令确定
  - 分支的目标地址由16位带符号偏移量左移两位后与pc相加
- 举例：

指令举例	指令名称	含义
J name	跳转	PC $_{36\cdots63}$← name<<2
JAL name	跳转并链接	Regs[R31]←PC+4；PC $_{36\cdots63}$←name<<2； ((PC+4)－2^{27})≤name＜((PC+4)+2^{27})
JALR R3	寄存器跳转并链接	Regs[R31]←PC+4；PC← Regs[R3]
JR R5	寄存器跳转	PC← Regs[R5]
BEQZ R4，name	等于零时分支	if(Regs[R4]== 0) PC←name ； ((PC+4)－2^{17})≤name＜((PC+4)+2^{17})
BNE R3，R4，name	不相等时分支	if(Regs[R3]!= Regs[R4]) PC←name ((PC+4)－2^{17})≤name＜((PC+4)+2^{17})
MOVZ R1，R2，R3	等于零时移动	if(Regs[R3]==0) Regs[R1]← Regs[R2]

4. MIPS的浮点操作
   - 由操作码指出操作数是单精度（SP）或是双精度（DP）
     - 后缀S：表示操作数是单精度浮点数
     - 后缀D：表示是双精度浮点数
   - 浮点操作
     - 包括：加减乘除分别有单精度和双精度指令
   - 浮点数比较指令

     根据比较结果设置浮点状态寄存器中的某一位，以便于后面的分支指令BC1T（若真则分支）或BC1F（若假则分支）测试该位，以决定是否进行分支。

3.1 流水线的基本概念

3.1.1 什么是流水线技术

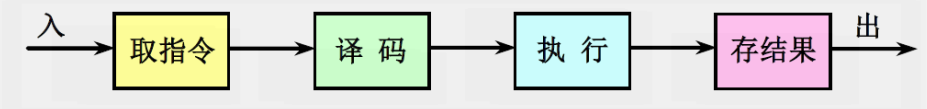
流水线技术概念 ☆

1. 把一个重复的过程分解为若干个子过程，每个子过程由专门的功能部件来实现。
2. 把多个处理过程在 时间上错开，依次通过各功能段，这样，每个子过程就可以与其它的子过程并行进行。

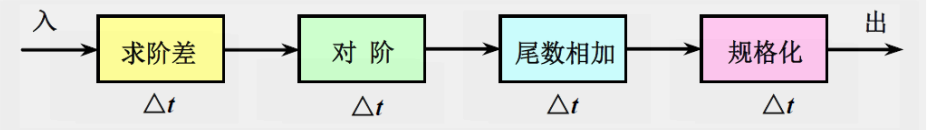
以上就是计算机中的流水线技术

名词解释

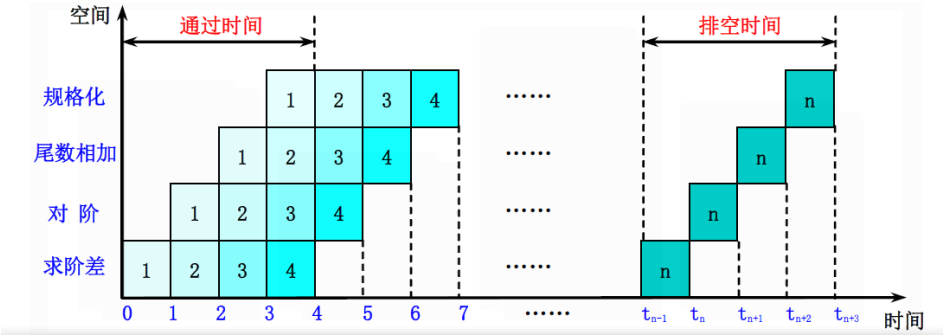
- **流水线的级或段**：流水线中的每个子过程及其功能部件
  段与段相互连接形成流水线
- **流水线的深度**：流水线的段数
- 指令流水线：
  - 把流水线技术**应用于指令的解释执行过程**，就形成了指令流水线。
  - 如果将指令分解为**分析**和**执行**两个子过程，并让这两个子过程分别用独立的分析部件和执行部件来实现：理想情况下，指令速度提高一倍
  - 四段指令流水线：



- 浮点加法流水线
  - 把流水线技术**应用于运算的执行过程**，就形成了**运算操作流水线**，也称为**部件级流水线**。
  - 把浮点加法的全过程分解为求阶差、对阶、尾数、相加、规格化四个子过程。**理想情况：速度提高3倍**



- 时空图
  - 时 - 空图从时间和空间两个方面描述了流水线的工作过程。
  - 时 - 空图中，横坐标代表时间，纵坐标代表流水线的各个段。
  - 例如：浮点加法流水线的时空图



- **通过时间**：第一个任务从进入流水线到流出结果所需的时间。
- **排空时间**：最后一个任务从进入流水线到流出结果所需的时间。

流水线技术特点

- 流水线把一个处理过程分解为若干个子过程（段），每个子过程由一个专门的功能部件来实现。
- 流水线中各段的时间应尽可能相等，否则将引起流水线堵塞、断流。
  - 时间最长的段将成为**流水线的瓶颈**。
- 流水线**每一个段的后面都要有一个缓冲寄存器（锁存器）**，称为流水寄存器。
  - 作用：在相邻的两段之间传送数据，以保证提供后面要用到的信息，并把各段的处理工作相互隔离。
- **流水技术适合于大量重复的时序过程**，只有在输入端不断地提供任务，才能充分发挥流水线的效率。
- 流水线需要有通过时间和排空时间 ☆
  - **通过时间**：第一个任务从进入流水线到流出结果所需的时间。
  - **排空时间**：最后一个任务从进入流水线到流出结果所需的时间。

3.1.2 流水线的分类

部件级、处理机级及系统级流水线

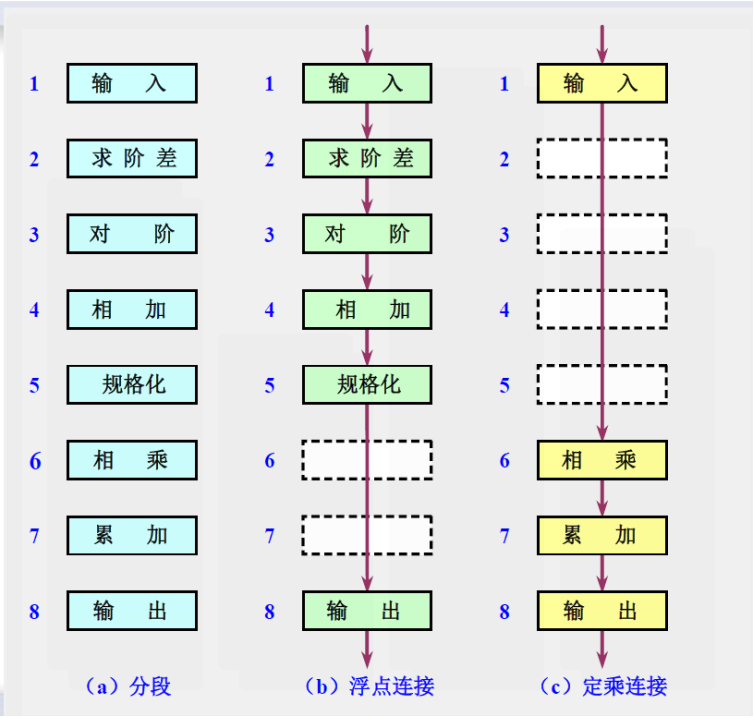
（按照流水技术用于计算机系统的等级不同）

- **部件级流水线（运算操作流水线）**：把处理机中的部件分段，再把这些分段相互连接起来，使得各种类型的运算操作能够按流水方式进行。
- **处理机级流水线（指令流水线）**：把指令的执行过程按照流水方式处理。把一条指令的执行过程分解为若干个子过程，每个子过程在独立的功能部件中执行。
- **系统级流水线（宏流水线）**：把多台处理机串行连接起来，对同一数据流进行处理，每个处理机完成整个任务中的一部分

单功能流水线与多功能流水线

（按照流水线所完成的功能来分类）

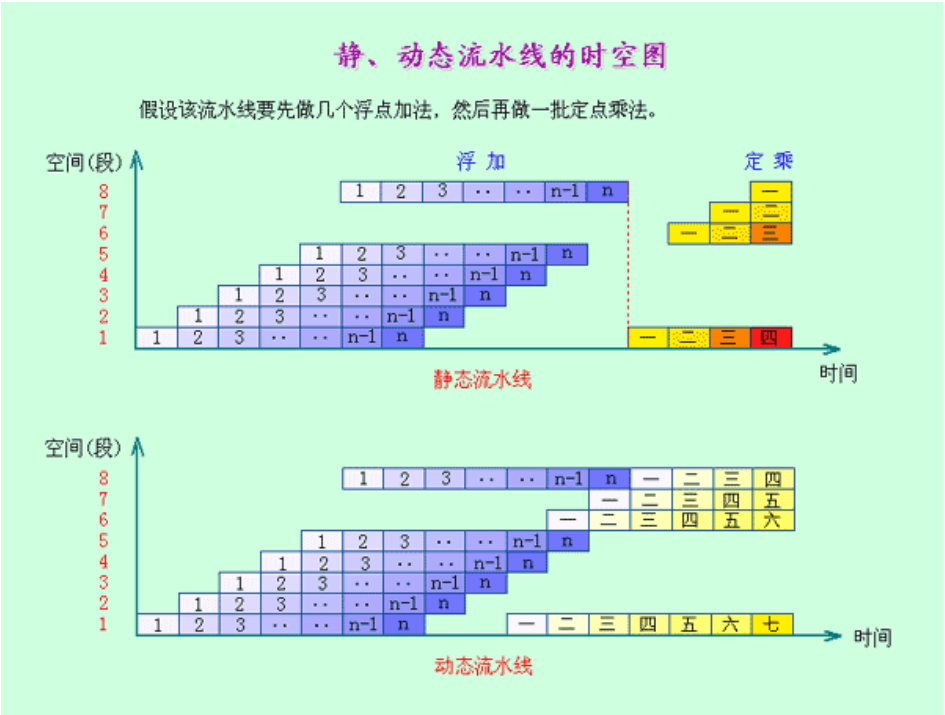
- **单功能流水线**：只能完成一种固定功能的流水线。
- **多功能流水线**：流水线的各段可以进行不同的连接，以实现不同的功能。
  - 例：ASC处理机的多功能流水线
    - 具有8个功能段，按不同链接，可以实现浮点加减法和定点乘法运算



静态流水线与动态流水线

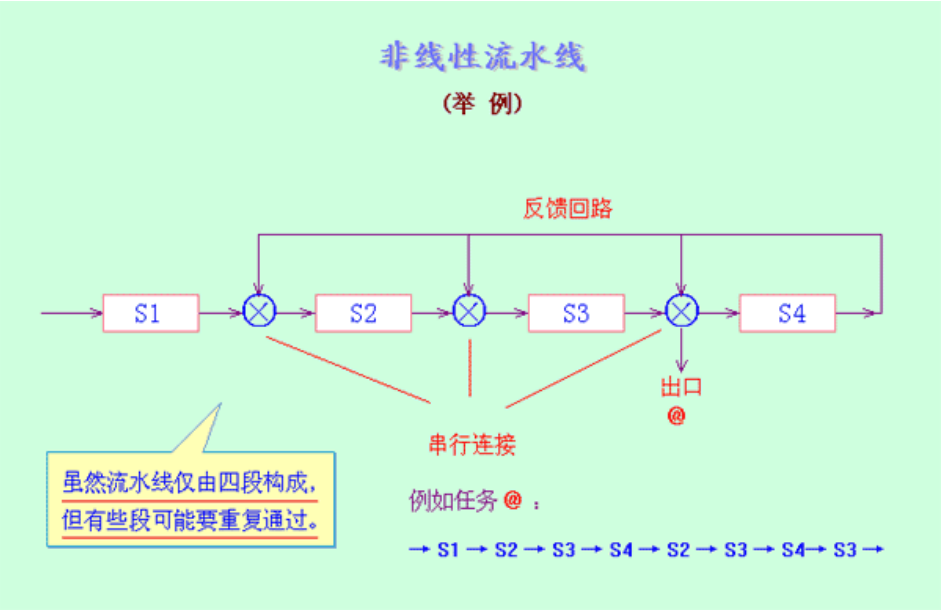
（按照同一时间内各段之间的连接方式对多功能流水线作
进一步的分类）

- **静态流水线**：在同一时间内，多功能流水线中的各段只能按同一种功能的连接方式工作。
  - 对于静态流水线来说，只有当输入的是一串相同的运算任务时，流水的效率才能得到充分的发挥
- **动态流水线**：在同一时间内，多功能流水线中的各段可以按照不同的方式连接，同时执行多种功能。
  - 优点 ： 灵活，能够提高流水线各段的使用率，从而提高处理速度。
  - 缺点：控制复杂。
- 对比：



线性流水线与非线性流水线

（按照流水线中是否有反馈回路来进行分类）

- **线性流水线**：流水线的各段串行连接，没有反馈回路。数据通过流水线中的各段时，每一个段最多只流过一次。
- **非线性流水线**：流水线中除了有串行的连接外，还有反馈回路。
  - 非线性流水线的调度问题
    - 确定什么时候向流水线引进新的任务，才能使该任务不会与先前进入流水线的任务发生冲突——争用流水段。
  - 

顺序流水线与乱序流水线

（根据任务流入和流出的顺序是否相同来进行分类）

- **顺序流水线**：流水线输出端任务流出的顺序与输入端任务流入的顺序完全相同。每一个任务在流水线的各段中是一个跟着一个顺序流动的。
  -**乱序流水线**：流水线输出端任务流出的顺序与输入端任务流入的顺序可以不同，允许后进入流水线的任务先完成（从输出端流出）。也称为无序流水线、错序流水线、异步流水线

标量处理机与向量流水处理机

把指令执行部件中采用了流水线的处理机称为流水线处理机。

- **标量处理机**：处理机不具有向量数据表示和向量指令，仅对标量数据进行流水处理。
- **向量流水处理机**：具有向量数据表示和向量指令的流水线处理机，简称向量机。
  - 向量数据表示和流水技术的结合。

3.2 流水线的性能指标

3.2.1 吞吐率（TP：ThroughPut） ☆

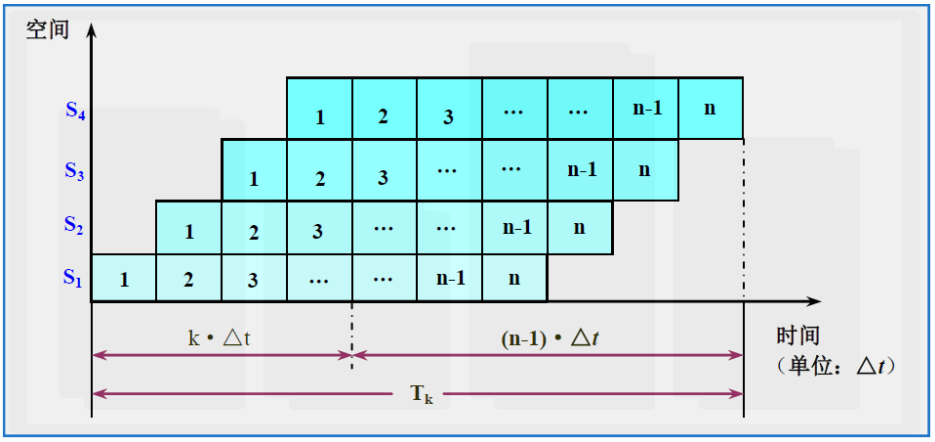
在单位时间内流水线所完成的任务数量或输出结果的数量。

$$TP = \frac{n}{T_k}$$

n: 流水线中的任务数量

T_k:处理完成n个任务所用的时间

各段时间均相等的流水线



（假设一条k段线性流水线）

完成第一个任务所需要的时间：$k * \Delta t$
完成第二个任务到第n个任务结束所需要的时间：$(n-1) * \Delta t$

流水线完成n个连续任务所需要的总时间为：

$$T_k = k * \Delta t + (n-1) * \Delta t = (k+n-1) * \Delta t$$

流水线的实际吞吐率

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

最大吞吐率 ☆

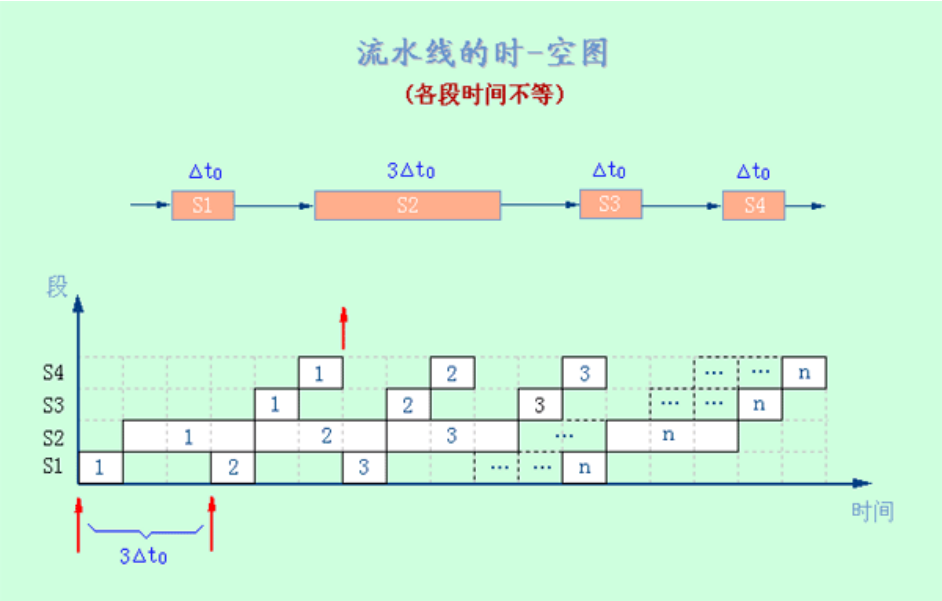
$$TP_{\max} = \lim_{n\to\infty} \frac{n}{(k+n-1)\Delta t} = \frac{1}{\Delta t}$$

最大吞吐率与实际吞吐率的关系

$$TP = \frac{n}{k+n-1} TP_{\max}$$

流水线的实际吞吐率小于最大吞吐率, 它除了与每个段的时间有关外, 还与流水线的段数 k以及输入到流水线中的任务数 n等有关。
只有当 $n > k$ 时, 才有 $TP \approx \mathrm{TP}_{\max}$ 。

各段时间完全不相等的流水线



一条4段的流水线，S1，S3，S4各段的时间：Δt，S2的时间：3Δt

流水线中这种时间最长的段称为流水线的**瓶颈段**。

完成第一个任务所需要的时间：Δt+3Δt+Δt+Δt = 6Δt
完成第二个任务到第n个任务结束所需要的时间：(n-1)3Δt

各段时间不等的流水线的实际吞吐率为:

(Δt_i为第 i 段的时间, 共有 k个段)

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\max\left(\Delta t_1, \Delta t_2, \cdots, \Delta t_k\right)}$$

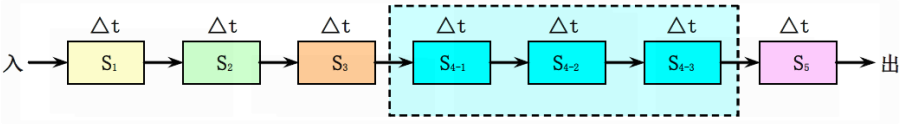
流水线的最大吞吐率为:

$$TP_{\max} = \frac{1}{\max\left(\Delta t_1, \Delta t_2, \cdots \Delta t_k\right)}$$

取决于**瓶颈段**的时间

解决流水线瓶颈问题的常用方法

1. 细分瓶颈段
   - 例如：对前面的5段流水线
     把瓶颈段S_4细分为3个子流水线段：S_{4-1}，S_{4-2}，S_{4-3}



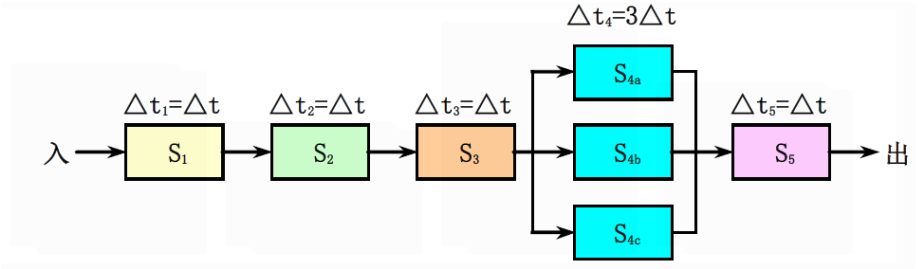
改进后的流水线的吞吐率 ：$TP_{\max} = \frac{1}{\Delta t}$

2. 重复设置瓶颈段
   - 重复设置的段并行工作，时间上依次错开处理任务。

- 缺点：控制逻辑比较复杂，所需的硬件增加了。

例如：对前面的5段流水线

重复设置瓶颈段S_4：S_{4a}，S_{4b}，S_{4c}



3.2.2 流水线的加速比 ☆

加速比：完成同样一批任务，不使用流水线所用的时间与使用流水线所用的时间之比。

假设：不使用流水线（即顺序执行）所用的时 间为 $\mathrm{T_s}$ ，使用流水线后所用的时间为 $\mathrm{T_k}$ ，则该 流水线的加速比为:

$$S = \frac{T_s}{T_k}$$

流水线各段时间相等：

一条k段流水线完成n个连续任务

所需要的时间为:

$$\mathrm{T_k = (k + n - 1)\Delta t}$$

顺序执行n个任务

所需要的时间: $T_s = nk\Delta t$

流水线的实际加速比为:

$$S = \frac{nk}{k + n - 1}$$

最大加速比：

$$S_{\max} = \frac{nk}{k + n - 1} = k$$

当$n >> k$时，$S \approx k$

流水线的各段时间不完全相等时

一条 k 段流水线完成 n 个连续任务的实际加速比为:

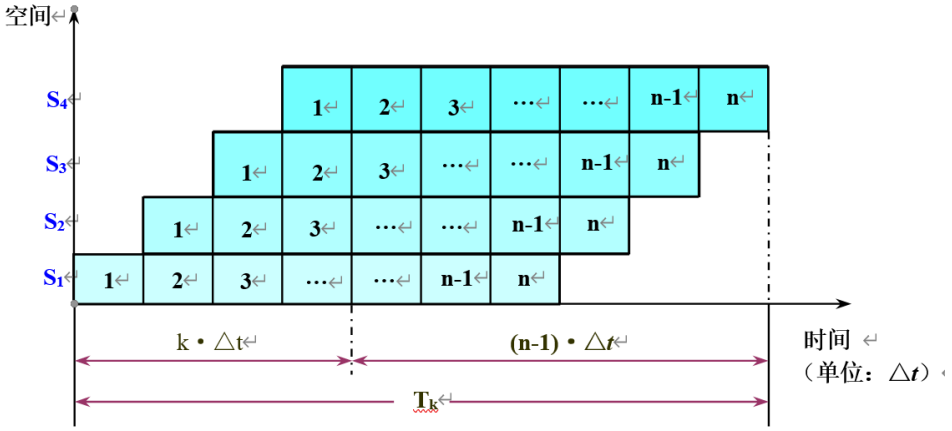
$$S=\frac{n\sum_{i=1}^{k}\Delta t_i}{\sum_{i=1}^{k}\Delta t_i+(n-1)\max\left(\Delta t_1,\Delta t_2,\cdots,\Delta t_k\right)}$$

3.2.3 流水线的效率 ☆

流水线的效率：流水线中的设备实际使用时间与整个运行时间的比值，即**流水线设备的利用率**。

由于流水线有通过时间和排空时间，所以在连续完成n个任务的时间内，各段并不是满负荷地工作。

各段时间相等



- 各段的效率e_i相同

$$e_1=e_2=\cdots=e_k=\frac{n\Delta t}{T_k}=\frac{n}{k+n-1}$$

- 整条流水线的效率为：

$$E=\frac{e_{1+}e_2+\cdots+e_k}{k}=\frac{ke_1}{k}=\frac{kn\Delta t}{kT_k}$$

  可以写成:

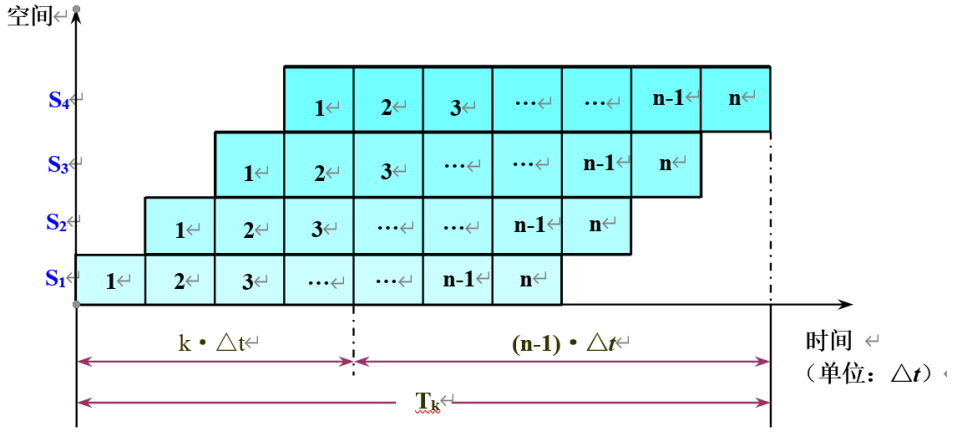
$$E=\frac{n}{k+n-1}$$

- 最高效率为:

$$E_{\max}=\lim_{n\to\infty}\frac{n}{k+n-1}=1$$

$$当n>>k时,\mathrm{E}\approx 1$$

- 当流水线各段时间相等时，流水线的效率与吞吐率成正比。
  - $$E=TP\cdot\Delta t$$
  - $$E=\frac{n}{k+n-1}\quad TP=\frac{n}{(k+n-1)\Delta t}$$
- 流水线的效率是流水线的实际加速比 S 与它的最大加速 比 k 的比值。

- $$E=\frac{S}{k} \quad S=\frac{nk}{k+n-1}$$
  - 当 $\mathrm{E}=1$ 时, $\mathrm{S}=\mathrm{k}$, 实际加速比达到最大。
- 从时空图上看，效率就是n个任务占用的时空面积和k个段总的时空面积之比。
  - $$E=\frac{n个任务实际占用的时空去}{k个段中的时空区}$$
  - 

各段时间不相等时

$$E=\frac{n \cdot \sum_{i=1}^{k} \Delta t_i}{k\left[\sum_{i=1}^{k} \Delta t_i+(n-1) \cdot \max \left(\Delta t_1, \Delta t_2, \cdots, \Delta t_k\right)\right]}$$

3.2.5 流水线设计中的若干问题

瓶颈问题 ☆

- 理想情况下，流水线在工作时，其中的任务是同步地每一个时钟周期往前流动一段。
- 当流水线各段不均匀时，机器的时钟周期取决于瓶颈段的延迟时间。
- 在设计流水线时，要尽可能使各段时间相等。

流水线的额外开销 ☆

流水寄存器延迟 时钟偏移开销

- 流水寄存器需要建立时间和传输延迟
  - 建立时间：在触发写操作的时钟信号到达之前，寄存器输入必须保持稳定的时间。
  - 传输延迟：时钟信号到达后到寄存器输出可用的时间。
- 时钟偏移开销
  流水线中，时钟到达各流水寄存器的最大差值时间。（时钟到达各流水寄存器的时间不是完全相同）
- 注:
  - 流水线并不能减少（而且一般是增加）单条指令的执行时间，但却能提高吞吐率。
  - 增加流水线的深度（段数）可以提高流水线的性能。
  - 流水线的深度受限于流水线的额外开销。

- 当时钟周期小到与额外开销相同时，流水已没意义。因为这时在每一个时钟周期中已没有时间来做有用的工作。

冲突问题---流水线设计中要解决的重要问题之一 ☆

- 运算操作的数据准备
- 指令操作的相互关联

3.3 非线性流水线的调度

在非线性流水线中,存在反馈回路,当一个任务在流水线中流过时,可能要多次经过某些段

流水线调度要解决的问题:
应按什么样的时间间隔向流水线输入新任务，才能既不发生功能段使用冲突，又能使流水线有较高的吞吐率和效率?

3.3.1 单功能非线性流水线的最优调度

非线性流水线的启动距离:向一条非线性流水线的输入端连续输入两个**任务**之间的时间间隔称为非线性流水线的启动距离

禁用启动距离:会引起非线性流水线功能段使用冲突的启动距离则称为禁用启动距离

启动距离和禁用启动距离一般都用时钟周期数来表示，是一个正整数。

预约表:

戴文森提出使用预约表来对流水线的任务进行优化调整和控制。

横向（向右）：时间（一般用时钟周期表示）

纵向（向下）：流水线的段

例：一个5功能段非线性流水线预约表

时间 / 功能段	1	2	3	4	5	6	7	8	9
S1	√								√
S2		√	√					√	
S3				√					
S4					√	√			
S5							√	√	

如果在第n个时钟周期使用第k段，则在第k行和第n列的交叉处的格子里有一个√。
如果在第k行和第n列的交叉处的格子里有一个√，则表示在第n个时钟周期要使用第k段。

根据预约表求得最佳调度方案的方法 ☆

1. **根据预约表写出禁止表F**
   - 禁止表:一个由**禁用启动距离**构成的集合
   - 具体方法：对于预约表的每一行的任何一对✔，用它们所在的列号相减（大的减小的），列出各种可能的差值，然后删除相同的，剩下的就是禁止表的元素
   - 在该图中为:

例：一个5功能段非线性流水线预约表

功能段 \ 时间	1	2	3	4	5	6	7	8	9
S1	√								√
S2		√	√					√	
S3				√					
S4					√	√			
S5							√	√	

     - 第一行的差值只有一个：8；
     - 第二行的差值有3个：1，5，6；
     - 第3行只有一个√，没有差值；
     - 第4和第5行的差值都只有一个：1；
     - **其禁止表是：F={1，5，6，8}**
2. **根据禁止表F写出初始冲突向量C_0**

   进行从一个集合到一个二进制位串的变换
   - 冲突向量:一个N位的二进制位串
   - 设$C_0=(c_N,c_{N-1},\cdots,c_2,c_1)$,则当$i$在禁止表中时,$c_i=1$否则为0,即为:
     - $c_i=0$:允许间隔i个时钟周期后送入后续任务
     - $c_i=1$:不允许间隔i个时钟周期后送入后续任务
   - **对于上面的例子:F={1，5，6，8},所以$C_0=(10110001)$**
3. **根据初始冲突向量C_0画出状态转换图**
   - 当第一个任务流入流水线后，**初始冲突向量C0决定了下一个任务需间隔多少个时钟周期才可以流入。**
   - 在第二个任务流入后，新的冲突向量为:假设第二个任务是在与第一个任务间隔j个时钟周期流入，这时，由于第一个任务已经在流水线中前进了j个时钟周期，其相应的禁止表中各元素的值都应该减去j，并丢弃小于等于0的值。-----**也就是:对冲突向量来说，就是逻辑右移j位（左边补0）。** 在冲突向量上，就是对它们的冲突向量**进行“或”运算**

$$SHR^{(j)}(C_0)\vee C_0$$

其中:$SHR^{(j)}$表示逻辑右移j位

   - 更一般的情况

推广到更一般的情况

假设：C_k：当前的冲突向量

j：允许的时间间隔

则新的冲突向量为：

$$SHR^{(j)}(C_k) \vee C_0$$

- 对于所有允许的时间间隔都按上述步骤求出其新的冲突向量，并且把新的冲突向量作为当前冲突向量，反复使用上述步骤，直到不再产生新的冲突向量为止。
- 具体方法:从初始冲突向量C0出发，反复应用上述步骤，可以求得所有的冲突向量以及产生这些向量所对应的时间间隔。由此可以画出**用冲突向量表示的流水线状态转移图**。
  - 有向弧:表示状态转移的方向
  - 弧上的数字:表示引入后续任务(从而产生新的冲突向量)所用的时间间隔(时钟周期数)
- 对于上面的例子有:
  - （1） C_0=（10110001）

    引入后续任务可用的时间间隔为：2、3、4、7个时钟周期

    如果采用2，则新的冲突向量为：

    （00101100）∨（10110001）= （10111101）

    如果采用3，则新的冲突向量为：

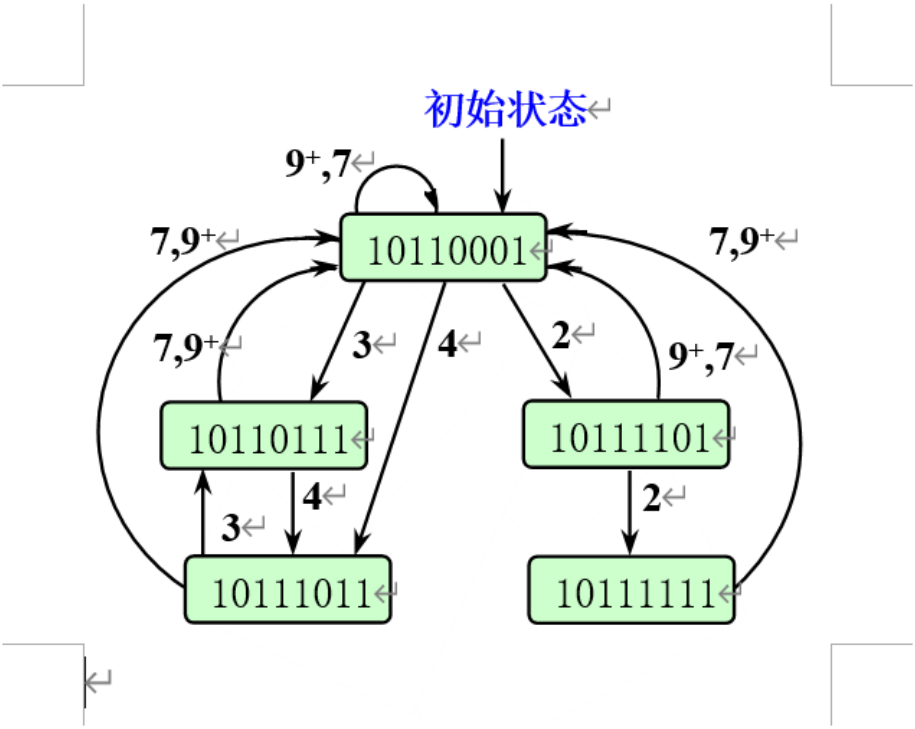
    （00010110）∨（10110001）= （10110111）

    如果采用4，则新的冲突向量为：

    （00001011）∨（10110001）= （10111011）

    如果采用7，则新的冲突向量为：

    （00000001）∨（10110001）= （10110001）
  - （2）对于新向量（10111101），其可用的时间间隔为2个和7个时钟周期。用类似上面的方法，可以求出其后续的冲突向量分别为（10111111）和（10110001）。
  - 对于其他新向量，也照此处理。
  -



- 对于9^+的解释为:对于每个状态来说,只要等待9个时钟周期,那么它的码就变为000000000,这时与初始状态进行或运算,那么得到的自然也就是初始状态

4. **根据状态转换图写出最优调度方案**
   - 根据流水线状态图，**由初始状态出发，任何一个闭合回路即为一种调度方案。**
   - 列出所有可能的调度方案，计算出每种方案的平均时间间隔，从中找出其最小者即为最优调度方案。
   -

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

   - **所以,最短的就是(3+4)/2 = 3.5**
   - 方案（3，4）是一种**不等时间间隔**的调度方案，与**等间隔的调度方案**相比，在控制上要复杂得多。为了简化控制，也可以采用等间隔时间的调度方案，但吞吐率和效率往往会下降不少。

3.4 流水线的相关与冲突

3.4.1 一条经典的5段流水线

以RISC流水线为例

一条指令的执行过程分为以下5个周期 ☆

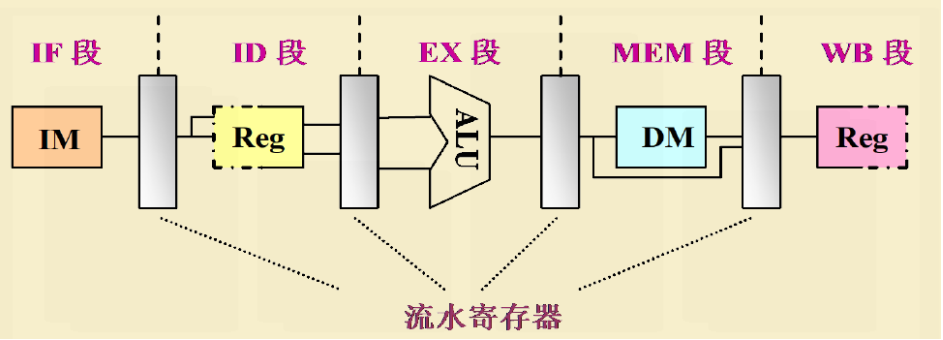
1. **取指周期**(IF:Instruction Fetch)
   - 以程序计数器PC中的内容作为地址，从存储器中取出指令并放入指令寄存器IR
   - 同时PC值加4（假设每条指令占4个字节），指向顺序的下一条指令
2. **指令译码**/读寄存器周期(ID:Instruction Decode)
   - 将IR中的指令拆分成操作码和操作数，并将操作数放入寄存器
   - 对指令进行译码，并用指令寄存器中的寄存器地址去访问通用寄存器组，读出所需的操作数。
3. **执行**/有效地址计算周期（EX:Execute）
   - **load和store指令**：ALU把指令中所指定的寄存器的内容与偏移量相加，形成访存有效地址。
   - **寄存器 - 寄存器ALU指令**：ALU按照操作码指定的操作对从通用寄存器组中读出的数据进行运算。
   - **寄存器 - 立即数ALU指令**:ALU按照操作码指定的操作对从通用寄存器组中读出的操作数和指令中给出的立即数进行运算。
   - **分支指令**:ALU把指令中给出的偏移量与PC值相加，形成转移目标的地址。同时，对在前一个周期读出的操作数进行判断，确定分支是否成功。
4. **存储器访问** / 分支完成周期（MEM:Memory Access）
   - 该周期处理的指令**只有load、store和分支指令**。其它类型的指令在此周期不做任何操作。
   - load和store指令
     - load指令：用上一个周期计算出的有效地址从存储器中读出相应的数据
     - store指令：把指定的数据写入这个有效地址所指出的存储器单元。
   - 分支指令:分支“成功”，就把转移目标地址送入PC。分支指令执行完成。
5. 写回周期（WB:WriteBack）
   - ALU运算指令和load指令在这个周期把结果数据写入通用寄存器组。
     - ALU运算指令：结果数据来自ALU。
     - load指令：结果数据来自存储器。

在这个实现方案中：

- 分支指令需要4个时钟周期（如果把分支指令的执行提前到ID周期，则只需要2个周期）；
- store指令需要4个周期；
- 其它指令需要5个周期才能完成。

流水线实现

- 每一个周期作为一个流水段,在各段之间加上锁存器

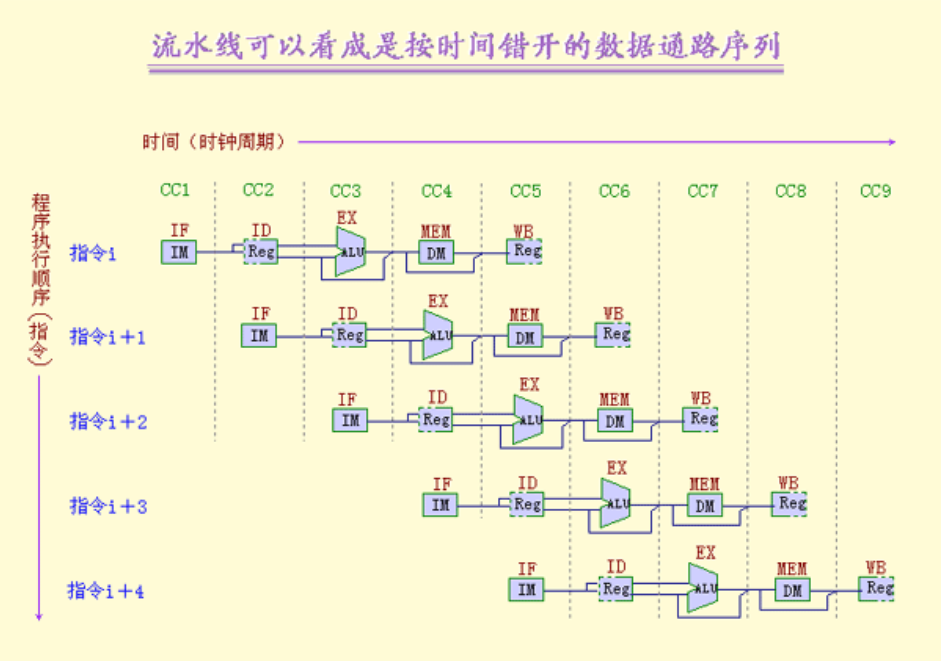


- 采用流水线方式实现时,应该注意的问题
  1. 要保证不会在同一时钟周期要求同一个功能段做两件不同的工作。
  2. 避免IF段的访存（取指令）与MEM段的访存（读/写数据）发生冲突。
     - 可以采用分离的指令存储器和数据存储器；一般采用分离的指令Cache和数据Cache。
  3. ID段和WB段都要访问同一寄存器文件
     - 解决方法:把**写操作**安排在时钟周期的**前半拍**完成，把**读操作**安排在**后半拍**完成。
  4. 考虑PC的问题
     - 流水线为了能够每个时钟周期启动一条新的指令，就必须在**每个时钟周期进行PC值的加4**操作，并保留新的PC值。**这种操作必须在IF段完成**，以便为取下一条指令做好准备。----->设置专门的加法器
     - 但分支指令也可能改变PC的值，而且是在MEM段进行，这会导致冲突。

5段流水线的两种描述方式

- 第一种:

- 第二种:



3.4.2 相关与流水线冲突

相关 ☆

相关：两条指令之间存在某种依赖关系。
如果两条指令相关，则它们就有可能不能在流水线中重叠执行或者只能部分重叠执行。

相关的3种类型:

- 数据相关
- 名相关
- 控制相关

数据相关

对于两条指令i（在前）和j（在后），如果下述条件之一成立，则称指令j与指令i数据相关。

- 指令j使用指令i产生的结果
- 指令j与指令k数据相关，而指令k又与指令i数据相关(数据相关具有传递性)
- 数据相关反映了数据的流动关系，即如何从其产生者流动到其消费者。

例如:

例如：下面这一段代码存在数据相关。

当数据的流动是经过寄存器时，相关的检测比较直观和容易。

当数据的流动是经过存储器时，检测比较复杂。

- 相同形式的地址其有效地址未必相同（10（R5））
- 形式不同的地址其有效地址却可能相同

名相关

名：指令所访问的寄存器或存储器单元的名称

如果两条指令使用相同的名，但是它们之间并没有数据流动，则称这两条指令存在名相关。

指令j与指令i之间的名相关有两种:

- **反相关**：如果指令j写的名与指令i读的名相同，则称指令i和j发生了反相关。
  指令j写的名 = 指令i读的名
- **输出相关**：如果指令j和指令i写相同的名，则称指令i和j发生了输出相关。
  指令j写的名 = 指令i写的名

名相关的两条指令之间并没有数据的传送。

如果一条指令中的名改变了，并不影响另外一条指令的执行。.

换名技术:通过改变指令中操作数的名来消除名相关。

对于寄存器操作数进行换名称为寄存器换名

寄存器换名既可以用编译器静态实现，也可以用硬件动态完成.

例如：考虑下述代码：

```
DIV.D    F2，F8，F4
ADD.D    F8，F0，F12
SUB.D    F10，F8，F14
```

DIV.D和ADD.D存在反相关。

进行寄存器换名（F8换成S）后，变成：

```
DIV.D    F2，F8，F4
ADD.D    S，F0，F12
SUB.D    F10，S，F14
```

这里面存在数据相关也存在名相关

控制相关

控制相关是指由分支指令引起的相关。

为了保证程序应有的执行顺序，必须严格按控制相关确定的顺序执行。

```
if p1 {
        S1;
    };
S;
if p2 {
        S2;
    };
```

- 控制相关带来了以下两个限制：
  - 与一条分支指令控制相关的指令不能被移到该分支之前。否则这些指令就不受该分支控制了。
    - 对于上述的例子，then部分中的指令不能移到if语句之前。
  - 如果一条指令与某分支指令不存在控制相关，就不能把该指令移到该分支之后。
    - 对于上述的例子，不能把S移到if语句的then部分中

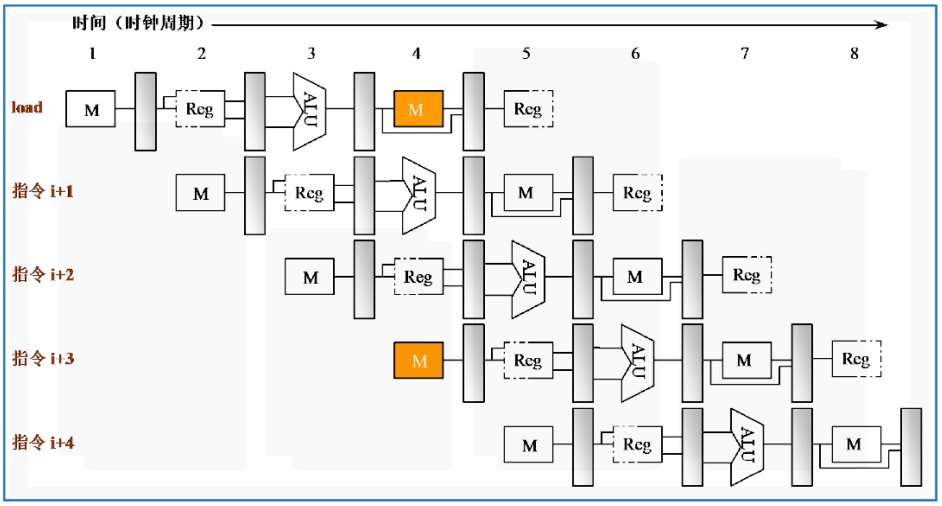
冲突 ☆

流水线冲突是指对于具体的流水线来说，由于相关的存在，使得指令流中的下一条指令不能在指定的时钟周期执行。

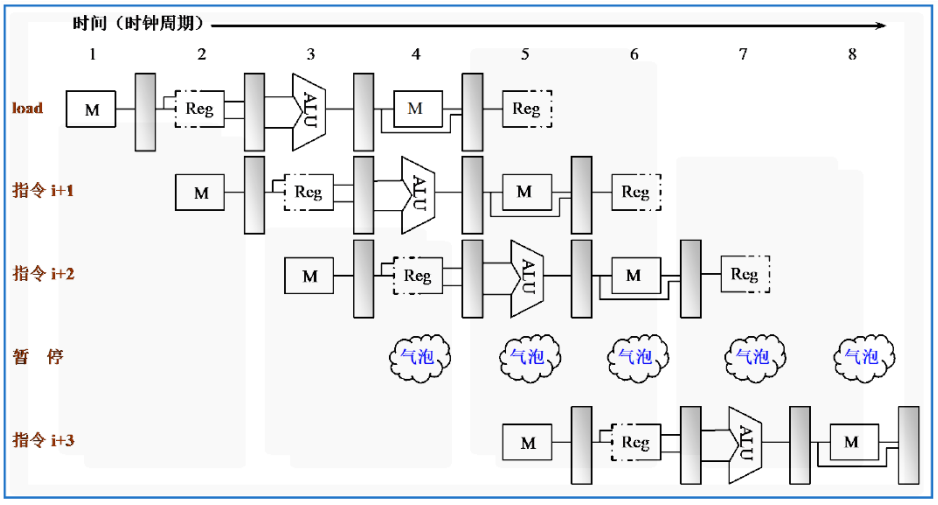
流水线冲突有3种类型：

结构冲突

因硬件资源满足不了指令重叠执行的要求而发生的冲突。

- 在流水线处理机中，为了能够使各种组合的指令都能顺利地重叠执行，需要对功能部件进行流水或重复设置资源。
- **如果某种指令组合因为资源冲突而不能正常执行，则称该处理机有结构冲突。**
- 常见的导致结构冲突的原因
  - 功能部件不是完全流水
  - 资源份数不够
- 结构冲突举例：**访存冲突**
  - 

    由于访问同一个存储器而引起的结构冲突
  - 有些流水线处理机只有一个存储器，将**数据和指令放在一起**，访存指令会导致访存冲突。

- 解决方法:
  - 解决办法 I：插入暂停周期（“流水线气泡”或“气泡”）
    - 

    - 引入暂停后的时空图

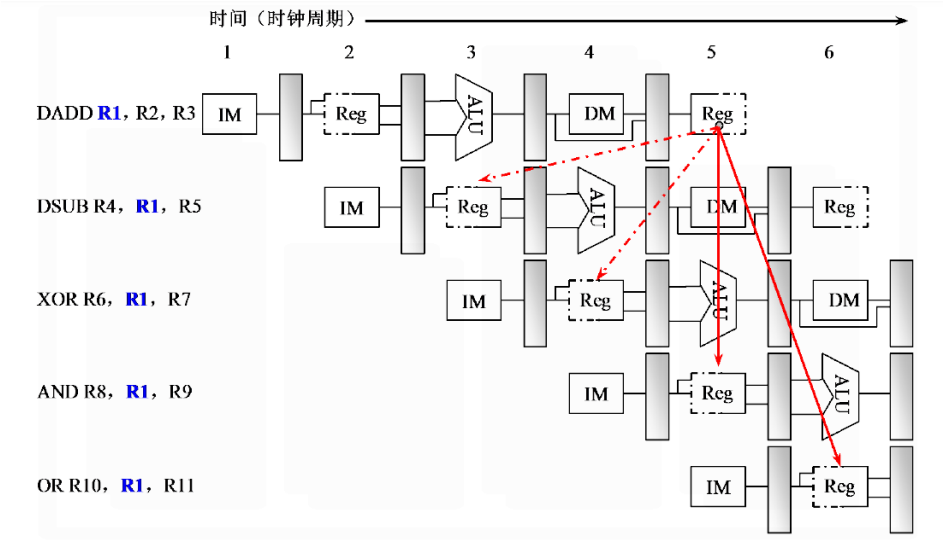
指令编号	时钟周期									
	1	2	3	4	5	6	7	8	9	10
指令i	IF	ID	EX	MEM	WB					
指令i+1		IF	ID	EX	MEM	WB				
指令i+2			IF	ID	EX	MEM	WB			
指令i+3				stall	IF	ID	EX	MEM	WB	
指令i+4						IF	ID	EX	MEM	WB
指令i+5							IF	ID	EX	MEM

  - 解决方法 II：设置相互独立的指令存储器和数据存储器或设置相互独立的指令Cache和数据Cache。
- 有时流水线设计者允许结构冲突的存在(**减少成本**)
  - 如果把流水线中的所有功能单元完全流水化，或者重复设置足够份数，那么所花费的成本将相当高。

数据冲突

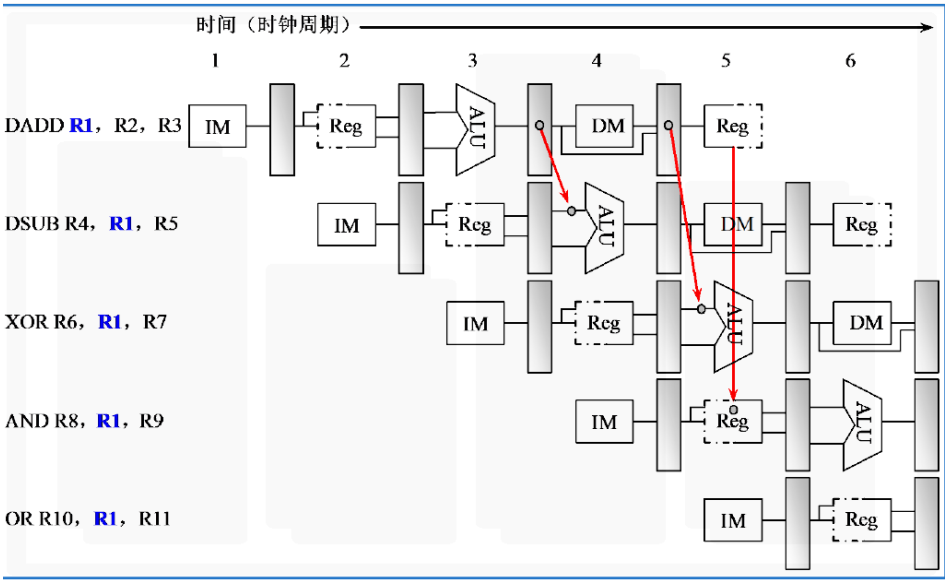
当指令在流水线中重叠执行时，因需要用到前面指令的执行结果而发生的冲突。

- 当相关的指令靠得足够近时，它们在流水线中的重叠执行或者重新排序会改变指令读/写操作数的顺序，使之不同于它们串行执行时的顺序，则发生了数据冲突。
-



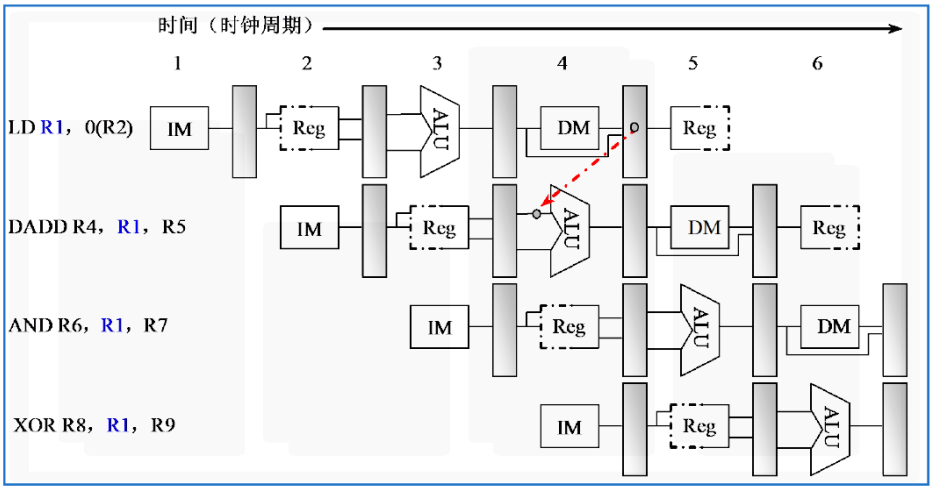
流水线的数据冲突举例

- 数据冲突三种分类:
  1. 写后读冲突(RAM)
     - 在i写入之前，j先去读。j 读出的内容是错误的。
     - 这是最常见的一种数据冲突，它对应于真数据相关。
  2. 写后写冲突（WAW）
     - 在i写入之前，j先写入。最后写入的内容是i,这个结果是错误的。
     - 这种冲突对应于输出相关
     - 写后写冲突仅发生在这样的流水线中：
       - 流水线中不只一个段可以进行写操作；
       - 指令被重新排序了。
     - 前面介绍的5段流水线不会发生写后写冲突。因为只有在WB段才会写回寄存器,所以不会发生
  3. 读后写冲突(WAR)
     - 在i读之前，j先写。i读出的内容是错误的!
     - 由反相关引起。
     - 这种冲突仅发生在这样的情况下
       - 有些指令的写结果操作提前了，而且有些指令的读操作滞后了；
       - 指令被重新排序了。
- 定向技术:减少写后读数据冲突引起的停顿:(旁路/短路)
  - 在计算结果尚未出来之前，后面等待使用该结果的指令并不一定立即需要该计算结果，如果能够将该计算结果从其产生的地方直接送到其它指令需要它的地方，那么就可以避免停顿。
  -



- 实现原理:
  - EX段和MEM段之间的流水寄存器中保存的ALU运算结果总是回送到ALU的入口。
  - 当定向硬件检测到前一个ALU运算结果写入的寄存器就是当前ALU操作的源寄存器时，那么控制逻辑就选择定向的数据作为ALU的输入，而不采用从通用寄存器组读出的数据。
- 结果数据不仅可以从某一功能部件的输出定向到其自身的输入，而且还可以定向到其它功能部件的输入。
- 需要停顿的数据冲突:
  - 举例:

```
LD    R1, 0（R2）
DADD  R4, R1, R5
AND   R6, R1, R7
XOR   R8, R1, R9
```

  - 因为load指令需要在访存阶段才能够拿到这个数据，而下一条指令在执行阶段就要用到,此时无法传递过去,必须要停顿
  - 

    无法将LD指令的结果定向到DADD指令
  - 解决措施:

        - 增加流水线互锁机制，插入“暂停”。作用：检测发现数据冲突，并使流水线停顿，直至冲突消失。
- 指令调度/流水线调度
    - 依靠编译器解决数据冲突
    - 让编译器重新组织指令顺序来消除冲突，这种技术称为指令调度或流水线调度。
    - 举例：

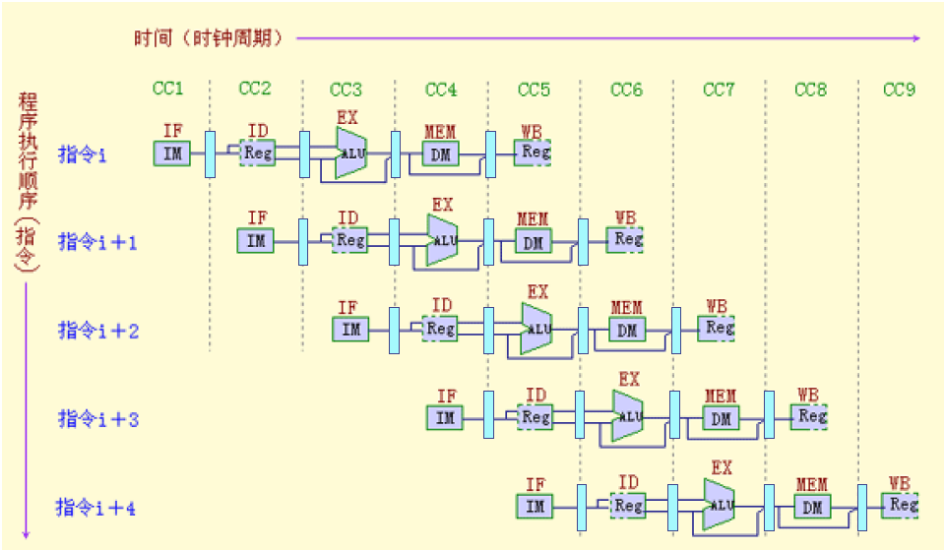
请为下列表达式生成没有暂停的指令序列：

A＝B＋C ；

D＝E－F ；

假设载入延迟为1个时钟周期。

调度前的代码		调度后的代码	
LD	Rb，B	LD	Rb，B
LD	Rc，C	LD	Rc，C
DADD	Ra，Rb，Rc	LD	Re，E
SD	Ra，A	DADD	Ra，Rb，Rc
LD	Re，E	LD	Rf，F
LD	Rf，F	SD	Ra，A
DSUB	Rd，Re，Rf	DSUB	Rd，Re，Rf
SD	Rd，D	SD	Rd，D

    - 到考试的时候一定要在草纸上画一下这个图:
        - 

控制冲突

流水线遇到分支指令和其它会改变PC值的指令所引起的冲突。

- 执行分支指令的结果有两种:
    - 分支成功:PC值改变为分支转移的目标地址
    - 不成功或失败:PC值保持正常递增,指向顺序的下一条指令

- 处理分支指令**最简单**的方法:
  - 冻结/排空流水线
  - 简单处理分支指令：分支成功的情况

分支指令	IF	ID	EX	MEM	WB					
分支目标指令		IF	stall	stall	IF	ID	EX	MEM	WB	
分支目标指令+1						IF	ID	EX	MEM	WB
分支目标指令+2							IF	ID	EX	MEM
分支目标指令+3								IF	ID	EX

- 由分支指令引起的延迟成为分支延迟
- 减少分支延迟的措施:
  - 在流水线中尽早判断出分支转移是否成功
  - 尽早计算出分支目标地址

假设分支指令是在ID段的末尾进行执行的,所带来的分支延迟为一个时钟周期,则有:

- 3种 **通过软件(编译器)** 来减少分支延迟的方法
  - 共同点:
    - 对分支的处理方法在程序的执行过程中始终是不变的，是静态的
    - 要么总是预测分支成功，要么总是预测分支失败

1. **预测分支失败**
   - 允许分支指令后的指令继续在流水线中流动，就好象什么都没发生似的;
   - **若确定分支失败，将分支指令看作是一条普通指令，流水线正常流动**;
   - **若确定分支成功，流水线就把在分支指令之后取出的所有指令转化为空操作，并按分支目地重新取指令执行。**
   - 要保证：分支结果出来之前不能改变处理机的状态，以便一旦猜错时，处理机能够回退到原先的状态。
   -

分支失败	分支指令i	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

分支成功	分支指令i	IF	ID	EX	MEM	WB				
	指令i+1		IF	idle	idle	idle	idle			
	分支目标指令i			IF	ID	EX	MEM	WB		
	分支目标指令i+3				IF	ID	EX	MEM	WB	
	分支目标指令i+4					IF	ID	EX	MEM	WB

2. **预测分支成功**
   - 假设分支转移成功，并从分支目标地址处取指令执行。
   - 起作用的前题：先知道分支目标地址，后知道分支是否成功。
   - 注:此方法在之前的5段流水线中没有任何好处

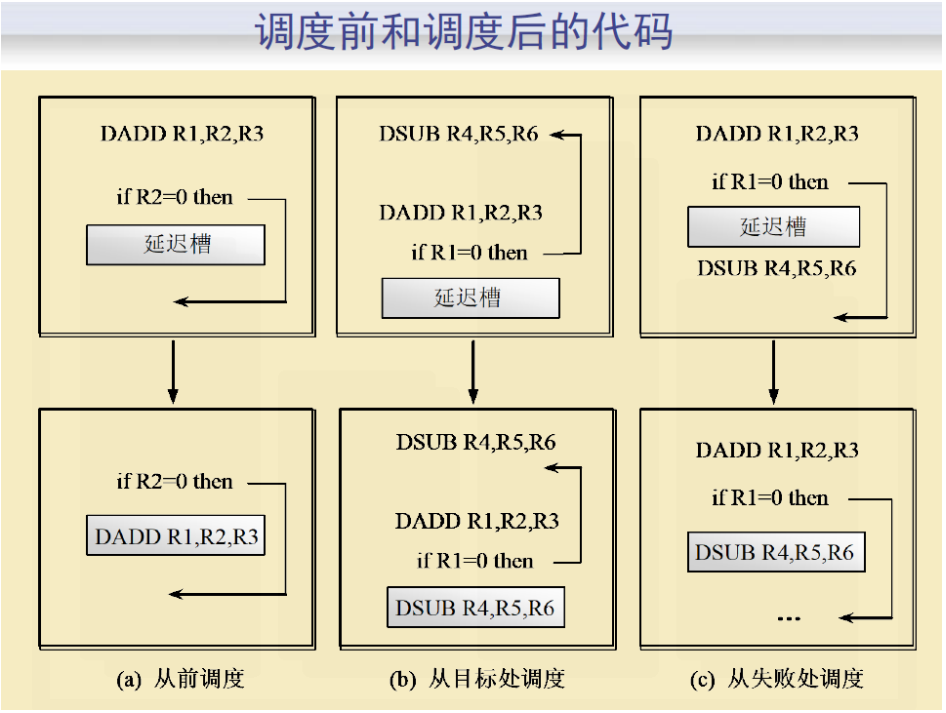
3. **延迟分支**

- 主要思想:从逻辑上“延长”分支指令的执行时间。把延迟分支看成是由原来的分支指令和若干个延迟槽构成，不管分支是否成功，都要按顺序执行延迟槽中的指令。
- 具有一个分支延迟槽的流水线的执行过程

分支失败	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令 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

分支成功	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令 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

- **分支延迟槽中的指令“掩盖”了流水线原来必需插入的暂停周期。**
- 分支延迟指令的调度
  - 任务：在延迟槽中放入有用的指令由编译器完成。能否带来好处取决于编译器能否把有用的指令调度到延迟槽中。
  - 三种调度方法:
    - 从前调度
    - 从目标处调度
    - 从失败处调度
  - 调度前和调度后的代码



三种方法的要求及效果

调 度 策 略	对调度的要求	什么情况下起作用？
从 前 调 度	被调度的指令必须与分支无关	任何情况
从目标处调度	必须保证在分支失败时执行被调度的指令不会导致错误。有可能需要复制指令。	分支成功时（但由于复制指令，有可能会增大程序空间）
从失败处调度	必须保证在分支成功时执行被调度的指令不会导致错误。	分支失败时

- 分支延迟受到两个方面的限制:
  - 可以被放入延迟槽中的指令要满足一定的条件；
  - 编译器预测分支转移方向的能力。
- 分支取消机制
  - 分支指令隐含了预测的分支执行方向，当分支的实际执行方向和事先所预测的一样时，执行分支延迟槽中的指令，否则就将分支延迟槽中的指令转化成一个空操作。
  -

分支失败	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令 i+1		IF	idle	idle	idle	idle			
	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

  -

分支成功	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令 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

3.5 流水线的实现

不做重点,不讲

向量处理机简介

向量由一组有序、具有相同类型和位数的元素组成。

在流水线处理机中，设置了向量数据表示和相应的向量指令的，称为**向量处理机**。

不具有向量数据表示和相应的向量指令的流水线处理机，称为**标量处理机**

典型的向量处理机:

- 1976年**Cray-1超级计算机**浮点运算速度达到了每秒1亿次
- CDC Cyber 205，Cray Y-MP，NEC SX-X/44，Fujitsu VP2600等性能达到了每秒几十亿 ~ 几百亿次浮点运算

4.1 向量的处理方式 ☆☆

```
//以计算表达式D=A×（B - C）为例
A、B、C、D—— 长度为N 的向量
main()
{   // 设A、B、C、D都是长度为n的数组
    for(int i=0;i<n;i++)
    {
        D[i]=A[i]*(B[i]-C[i]);
    }
}
```

横向(水平)处理方式 ☆

- 向量计算是按行的方式从左到右横向的进行的
  - $$\begin{aligned} &\text{先计算：} \quad d_1 \leftarrow a_1 \times (b_1 - c_1) \\ &\text{再计算：} \quad d_2 \leftarrow a_2 \times (b_2 - c_2) \\ &\qquad\qquad \cdots\cdots \\ &\text{最后计算：} \quad d_N \leftarrow a_N \times (b_N - c_N) \end{aligned}$$
- 组成循环程序进行处理
  - $$q_i \leftarrow b_i - c_i$$

    $$d_i \leftarrow q_i \times a_i$$

- 这样以来:
  - 有数据相关N次
  - 功能切换(也就是加法和乘法功能切换):2N次
- 所以,该种方法并不适合向量处理机的**并行处理模式**

纵向(垂直)处理方式☆☆

- 向量计算是按列的方式从上到下纵向的进行
- $$\text{先计算}\begin{cases} q_1 \leftarrow b_1 - c_1 \\ \cdots\cdots \\ q_N \leftarrow b_N - c_N \end{cases} \qquad \text{再计算}\begin{cases} d_1 \leftarrow q_1 \times a_1 \\ \cdots\cdots \\ d_N \leftarrow q_N \times a_N \end{cases}$$
- 表示成向量指令为:
  - $$Q = B - C$$
    $$D = Q \times A$$
- 这样以来:
  - 没有数据相关
  - 功能切换仅仅1次

纵横(分组)处理方式☆☆

- 把向量分成若干组,**组内按纵向方式处理**,依次处理各组
- $$N = S \times n + r$$
  - 其中N为向量长度，S为组数，n为每组的长度，r为余数。
  - 若余下的r个数也作为一组处理，则共有S+1组。
- 计算过程为:
  - 先算第1组：
    $$Q_{1\sim n} \leftarrow B_{1\sim n} - C_{1\sim n}$$
    $$D_{1\sim n} \leftarrow Q_{1\sim n} \times A_{1\sim n}$$
  -

- 再算第2组：

$$Q_{(n+1)\sim 2n} \leftarrow B_{(n+1)\sim 2n} - C_{(n+1)\sim 2n}$$

$$D_{(n+1)\sim 2n} \leftarrow Q_{(n+1)\sim 2n} \times A_{(n+1)\sim 2n}$$

- 依次进行下去，直到最后一组：第$S+1$组。
- 每组内各用两条向量指令。

4.2 向量处理机的结构

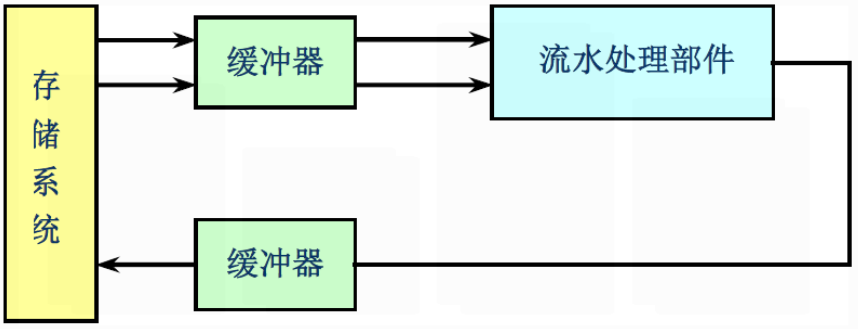
向量处理机的结构因具体机器不同而不同。

由所采用的向量处理方式决定。

有两种典型的结构

4.2.1 存储器-存储器型结构(纵向处理方式采用)

- 采用**纵向处理方式**的向量处理机对处理机结构的要求：存储器 - 存储器结构
  - 向量指令的源向量和目的向量都是存放在存储器中，运算的中间结果需要送回存储器。
  - **流水线运算部件的输入和输出端都直接（或经过缓冲器）与存储器相联**，从而构成存储器-存储器型操作的运算流水线。
  - 举例:STAR-100、CYBER-205
- “存储器 - 存储器”型操作的运算流水线
  -



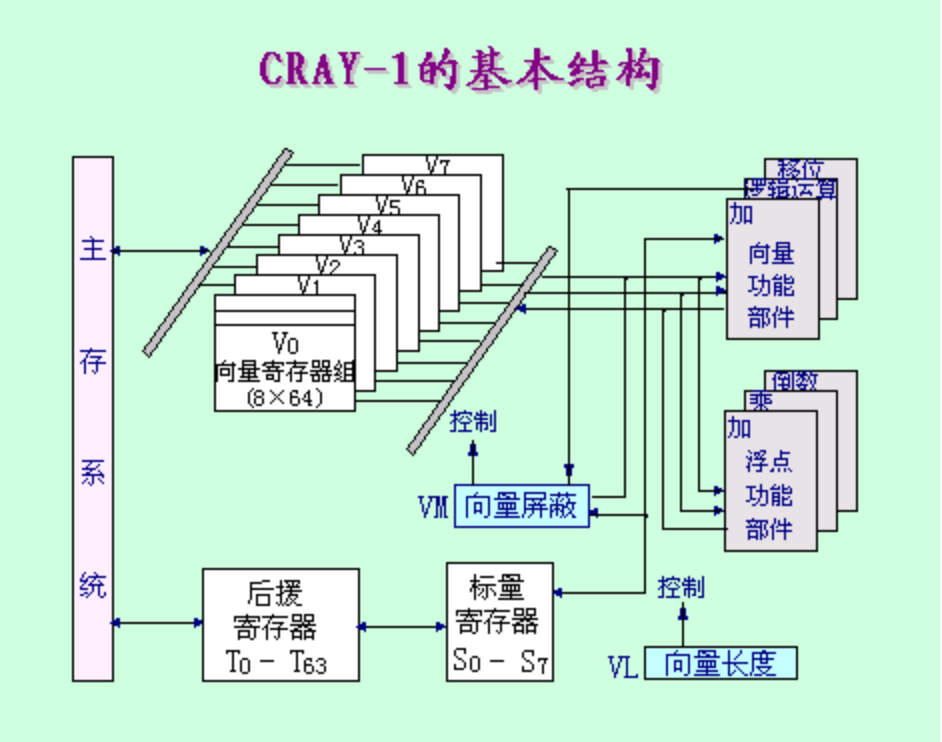
“存储器－存储器”型操作的运算流水线

- 要充分发挥这种结构的流水线效率，**存储器要不断地提供源操作数，并不断地从运算部件接收结果。**
  - 每拍从存储器读取两个数据，并向存储器写回一个结果）
  - 对存储器的带宽以及存储器与处理部件的通信带宽提出了非常高的要求。
  - 解决方法：一般是通过**采用多体交叉并行存储器**和**缓冲器技术。**
    -

例如，70年代初问世的Star 100

- 存储器：32个体交叉
- 每个体的数据宽度：8个字（字长64位）
- 最大数据流量：每秒2亿字

4.2.2 寄存器-寄存器型结构(分组处理方式采用)

- 在向量的分组处理方式中，对向量长度N没有限制，但**组的长度n却是固定不变的。**
  - 对处理机结构的要求：**寄存器－寄存器结构**
  - 设置能快速访问的向量寄存器，用于存放源向量、目的向量及中间结果。**让运算部件的输入、输出端都与向量寄存器相联**，就构成了“寄存器－寄存器”型操作的运算流水线。
  - 典型的寄存器－寄存器结构的向量处理机:美国的CRAY-1、我国的YH-1巨型机
- 以CRAY-1为例进行分析:
  - 基本结构
    - 功能部件:共有12条可并行工作的单功能流水线，可分别流水地进行地址、向量、标量的各种运算。
    - 

    - 6个单功能流水部件：进行向量运算
      - 整数加（3拍）
      - 逻辑运算（2拍）
      - 移位（4拍）
      - 浮点加（6拍）
      - 浮点乘（7拍）
      - 浮点迭代求倒数（14拍）
      - 括号中的数字为其流水经过的时间，每拍为一个时钟周期，即12.5ns。
    - 向量寄存组V

        - 由512个64位的寄存器组成，分成8组。
          组编号：$V_0 \sim V_7$
          每一个组称为一个向量寄存器，可存放一个长度（即元素个数）不超过64的向量。
          每个向量寄存器可以每拍向功能部件提供一个数据元素，或者每拍接收一个从功能部件来的结果元素。
    - 标量寄存器S和快速暂存器T
        - 标量寄存器有8个：$S_0 \sim S_7$　64位
          快速暂存器T用于在标量寄存器和存储器之间提供缓冲。
    - 向量屏蔽寄存器VM
        - 64位，每一位对应于向量寄存器的一个单元。
          作用：用于向量的归并、压缩、还原和测试操作、对向量某些元素的单独运算等。
- CRAY-1向量处理的一个显著特点
    - 每个向量寄存器Vi都有连到6个向量功能部件的单独总线。
    - 每个向量功能部件也都有把运算结果送回向量寄存器组的总线。
    - **只要不出现Vi冲突和功能部件冲突，各Vi之间和各功能部件之间都能并行工作**，大大加快了向量指令的处理。
        - **Vi冲突**：并行工作的各向量指令的源向量或结果向量使用了相同的Vi。
            - 例如：源向量相同
              $V_3 \leftarrow V_1 + V_2$
              $V_5 \leftarrow V_4 \wedge V_1$
        - **功能部件冲突**：并行工作的各向量指令要使用同一个功能部件。

4.3 提高向量处理机性能的常用技术

提高向量处理机性能的方法

设置多个功能部件，使它们并行工作；

设置多个独立的功能部件。这些部件能并行工作，并各自按流水方式工作，从而形成了多条并行工作的运算操作流水线

例如：CRAY-1向量处理机有4组12个单功能流水部件：

- 向量部件：向量加，移位，逻辑运算
- 浮点部件：浮点加，浮点乘，浮点求倒数
- 标量部件：标量加，移位，逻辑运算，数“1”/计数
- 地址运算部件：整数加，整数乘

采用链接技术，加快一串向量指令的执行；

- 两条向量指令占用功能流水线和向量寄存器的4种情况
  1. 指令不相关
  2. 功能部件冲突
  3. 源寄存器冲突
  4. 结果寄存器冲突
- 当前一条指令的结果寄存器是后一条指令的源寄存器、且不存在任何其他冲突时，就可以用链接技术来提高性能。
  - 例如：V3←V1＋V2

    V6←V3*V4
  - **向量流水线链接：**具有先写后读相关的两条指令，在不出现功能部件冲突和源向量冲突的情况下，可以把功能部件链接起来进行流水处理，以达到加快执行的目的。
  - Cray-1向量处理的一个显著特点
    - 链接特性的实质

      把流水线定向的思想引入到向量执行过程的结果。
  - 由于同步的需求，链接时，Cray-1中把向量数据元素送往向量功能部件以及把结果存入向量寄存器都需要一拍时间，从存储器中把数据送入访存功能部件也需要一拍时间。
- 进行向量链接的要求
  - 保证：无向量寄存器使用冲突和无功能部件使用冲突
  - 只有在前一条指令的第一个结果元素送入结果向量寄存器的那一个时钟周期才可以进行链接。
  - 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通过时间相等。
  - 要进行链接执行的向量指令的向量长度必须相等，否则无法进行链接。

采用分段开采技术，加快循环处理；

处理向量的长度大于向量寄存器的长度的情况

当向量的长度大于向量寄存器的长度时，必须把长向量分成长度固定的段，然后循环分段处理，每一次循环只处理一个向量段。这种技术称为**分段开采技术**。

由**系统硬件**和**软件控制**完成，对程序员是透明的

采用多处理机系统，进一步提高性能。

许多新型向量处理机系统采用了多处理机系统结构。例如：

- CRAY-2
  - 包含了4个向量处理机
  - 浮点运算速度最高可达1800MFLOPS
- CRAY Y-MP、C90
  - 最多可包含16个向量处理机

4.4 向量处理机的性能评价

向量指令的处理时间T_{vp}

- 一条向量指令的处理时间T_{vp}
  - 执行一条向量长度为 n 的向量指令所需的时间为:

$$T_{vp} = T_s + T_e + (n-1)T_c$$

T_s：向量处理部件流水线的建立时间为了使处理部件流水线能开始工作（即开始流入数 据）所需要的准备时间。
T_e：向量流水线的通过时间第一对向量元素通过流水线并产生第一个结果所花的时间。
T_c：流水线的时钟周期时间

  - 把上式中的参数都折算成时钟周期个数：

$$T_{vp} = [s + e + (n-1)]T_c$$

s: T_s 所对应的时钟周期数
e: $\mathrm{T_e}$ 所对应的时钟周期数不考虑 T_s,并令 $T_{\text{start}} = e - 1$

$$T_{vp} = (T_{start} + n)\, T_c$$

$\mathrm{T_{start}}$ ： 从一条向量指令开始执行到还差一个时钟周期就产生第一个结果所需的时钟周期数。可称之为 该向量指令的启动时间 。此后, **便是每个时钟周期 流出一个结果**, 共有 n个结果。

- 一组向量指令的处理时间

- 对于一组向量指令而言，其执行时间主要取决于三个因素：
  - 向量的长度
  - 向量操作之间是否存在流水功能部件的使用冲突
  - 数据的相关性
- 把能在**同一个时钟周期内一起开始执行的几条向量指令**称为一个编队。
  - 可以看出：同一个编队中的向量指令之间一定不存在流水向量功能部件的冲突和数据的冲突。
- 编队后，这个向量指令序列的总的执行时间为各编队的执行时间的和。

$$T_{all} = \sum_{i=1}^{m} T_{vp}^{(i)}$$

$\mathrm{T}^{(\mathrm{i})}\mathrm{vp}$: 第 i 个编队的执行时间
m ： 编队的个数

- 当一个编队是由若干条指令组成时，**其执行时间就应该由该编队中各指令的执行时间的 最大值 来确定。**
  - $T(i)_{\text{start}}$ ： 第i编队中各指令的启动时间的最大值
  - $T_{\text{all}} = \sum_{i=1}^{m} T_{vp}^{(i)} = \sum_{i=1}^{m} \left(T_{\text{start}}^{(i)} + n\right) T_c = \left(\sum_{i=1}^{m} T_{\text{start}}^{(i)} + mn\right) T_c = \left(T_{\text{start}} + mn\right) T_c$
  - $T_{\text{start}} = \sum_{i=1}^{m} T_{\text{start}}^{(i)}$ 该组指令总的启动时间（时钟周期个数）
  - 表示成时钟周期个数

$$\mathrm{T_{all}} = \mathrm{T_{start}} + \mathbf{mn}$$

- 分段开采时,一组向量指令的总执行时间
  - 当向量长度n大于向量寄存器长度MVL时，需要分段开采。
  - 引入一些客外的处理操作(假设: 这些操作所引入的额外时间为 $\mathrm{T_{loop}}$ 个时钟周期) 口 设 $\left\lfloor \frac{n}{MVL} \right\rfloor = p$ q ： 余数
    共有 $\mathbf{m}$个编队
    对于最后一次循环来说, 所需要的时间为:

$$\mathrm{T_{last}} = \mathrm{T_{start}} + \mathrm{T_{loop}} + \mathbf{m} \times \mathbf{q}$$

  - 其他的每一次循环所要花费的时间为:

$$T_{\text{step}} = T_{\text{start}} + T_{\text{loop}} + m \times MVL$$

总的执行时间为:

$$\begin{aligned}
\mathrm{T_{all}} &= \mathrm{T_{step}} \times \mathrm{p} + \mathrm{T_{last}} \\
&= (\mathbf{T_{start}} + \mathbf{T_{loop}} + \mathbf{m} \times \mathbf{MVL}) \times \mathbf{p} + (\mathbf{T_{start}} + \mathbf{T_{loop}} + \mathbf{m} \times \mathbf{q}) \\
&= (\mathbf{p} + \mathbf{1}) \times (\mathbf{T_{start}} + \mathbf{T_{loop}}) + \mathbf{m}(\mathbf{MVL} \times \mathbf{p} + \mathbf{q}) \\
&= \left[\frac{n}{MVL}\right] \times (T_{\text{start}} + T_{\text{loop}}) + mn
\end{aligned}$$

最大性能R_∞ 和半性能向量长度 $\mathrm{n}_{1/2}$

- 向量处理机的峰值性能 R_∞
  R_∞ 表示当向量长度为无穷大时, 向量处理机的 最高性能, 也称为峰值性能。

$$R_\infty = \lim_{n\to\infty} \frac{\text{向量指令序列中浮点运算次数} \times \text{时钟频率}}{\text{向量指令序列执行所需的时钟周期数}}$$

- 半性能向量长度$\mathrm{n}_{1/2}$
  - 半性能向量长度$n_{1/2}$是指向量处理机的性能为其最大性能的一半时所需的向量长度。
  - 评价向量流水线的建立时间对性能影响的重要参数。

向量长度临界值n_v

- **向量长度临界值n_v是指**：对于某一计算任务而言，向量方式的处理速度优于标量串行方式处理速度时所需的最小向量长度。

指令级并行 ☆☆

指令级并行：指指令之间存在的一种并行性，利用它，计算机可以并行执行两条或两条以上的指令。（**ILP**：Instruction-Level Parallelism）

开发ILP的途径有两种:

- **资源重复**，重复设置多个处理部件，让它们同时执行相邻或相近的多条指令；
- **采用流水线技术，使指令重叠并行执行。**

本章研究：如何利用**各种技术来开发更多的指令级并行（硬件的方法）**

5.1指令级并行的概念

- **开发ILP的方法可以分为两大类**
  - 主要基于硬件的动态开发方法
  - 基于软件的静态开发方法
- **流水线处理机的实际CPI**
  - 实际CPI等于理想流水线的CPI加上各类停顿的时钟周期数
    - $$\text{CPI}_{\text{流水线}} = \text{CPI}_{\text{理想}} + \text{停顿}_{\text{结构冲突}} + \text{停顿}_{\text{数据冲突}} + \text{停顿}_{\text{控制冲突}}$$
  - 理想CPI是衡量流水线最高性能的一个指标。
  - IPC：Instructions Per Cycle（每个时钟周期完成的指令条数）

    -基本程序块
  - 基本程序块:一串连续的代码除了入口和出口以外，没有其他的分支指令和转入点。
  - 程序平均每4 ~ 7条指令就会有一个分支。
- **循环级并行**：使一个循环中的不同循环体并行执行。
  - 开发循环的不同迭代之间存在的并行性
  - 是指令级并行研究的重点之一
  - 例如，考虑下述语句：

    ```
    for （i=1； i<=500； i++）
    a[i]+=s;
    ```

    - 每一次循环都可以与其它的循环重叠并行执行；
    - 在每一次循环的内部，却没有任何的并行性。
- 最基本的开发循环级并行的技术
  - 循环展开（loop unrolling）技术
  - 采用向量指令和向量数据表示

5.2相关与指令级并行

相关与流水线冲突

- 相关的类型有三种:
  - 数据相关
  - 名相关
  - 控制相关
- 流水线冲突是指对于具体的流水线来说，**由相关的存在，使得指令流中的下一条指令不能在指的时钟周期执行。**
  - 流水线冲突的类型有三种:
    - 结构冲突
    - 数据冲突
    - 控制冲突
- 相关是程序固有的一种属性，它反映了程序中指令之间的相互依赖关系。
- 具体的一次相关是否会导致实际冲突的发生以及该冲突会带来多长的停顿，则是流水线的属性。

可以从两个方面来解决相关问题

- 保持相关,但避免发生冲突
  - 指令调度（编译器或硬件）
- 通过代码调换,消除相关
  - 只有在可能会导致错误的情况下，才保持程序顺序。
    - 程序顺序（Program Order）：由原来程序确定的在完全串行方式下指令的执行顺序。
  - 控制相关并不是一个必须严格保持的关键属性。
    - 对于正确地执行程序来说，必须保持的最关键的两个属性是：**数据流**和**异常行为**。
      - **保持异常行为**是指：无论怎么改变指令的执行顺序，都不能改变程序中异常的发生情况。此条件通常被弱化为：**指令执行顺序的改变不能导致程序中发生新的异常**。
      - **数据流**：指数据值从其产生者指令到其消费者指令的实际流动。
        - 分支指令使得数据流具有动态性，因为一条指令有可能数据相关于多条先前的指令。
        - 分支指令的执行结果决定了哪条指令真正是所需数据的产生者。
    - 有时，不遵守控制相关既不影响异常行为，也不改变数据流。
      - 可以大胆地进行指令调度，把失败分支中的指令调度到分支指令之前。

5.3指令的动态调度 ☆☆

- **静态调度**
  - 依靠编译器对代码进行静态调度，以减少相关和冲突。
  - 它不是在程序执行的过程中、而是在**编译期间**进行代码调度和优化。
  - 通过把相关的指令拉开距离来减少可能产生的停顿。

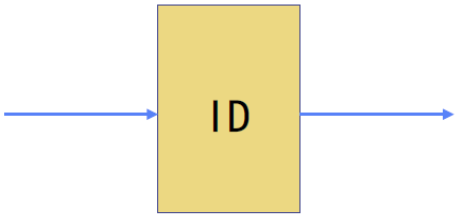
- **动态调度** ☆☆
  - **在程序的执行过程中，依靠专门硬件**对代码进行调度，减少数据相关导致的停顿。
  - 优点:
    - 能够处理一些在编译时情况不明的相关（比如涉及到存储器访问的相关），并简化了编译器；
    - 能够使本来是面向某一流水线优化编译的代码在其它的流水线（动态调度）上也能高效地执行。
  - 以硬件复杂性的显著增加为代价

5.3.1 动态调度的基本思想

到目前为止我们所使用流水线的最大的局限性:指令是按序流出和按序执行的

```
//考虑下面一段代码:
DIV.D   F4，F0，F2
ADD.D   F10，F4，F6
SUB.D   F12，F6，F14
//ADD.D指令与DIV.D指令关于F4相关，导致流水线停顿。
//SUB.D指令与流水线中的任何指令都没有关系，但也因此受阻。
```

在前面的基本流水线中：



检测结构冲突

检测数据冲突

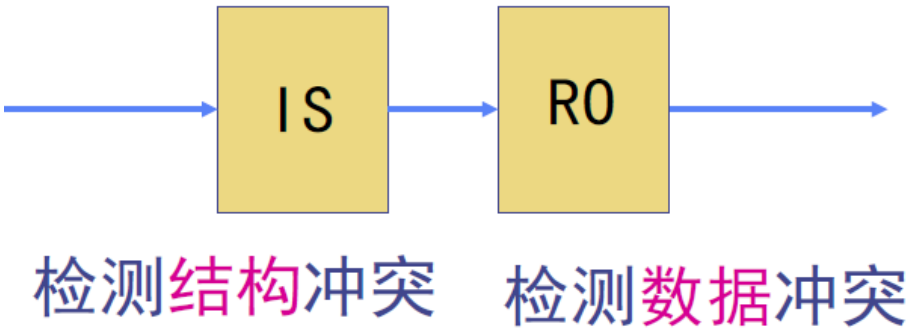
一旦一条指令受阻，其后的指令都将停顿。

为了使上述指令序列中的SUB.D指令能继续执行下去，必须把指令流出的工作拆分为两步：

- 检测结构冲突
- 等待数据冲突消失
  - 只要检测到没有结构冲突，就可以让指令流出。并且流出后的指令一旦其操作数就绪就可以立即执行。

修改后的流水线是乱序执行的

- **乱序执行**
  - 指令的执行顺序与程序顺序不相同
  - 指令的完成也是乱序完成的
    - 即指令的完成顺序与程序顺序不相同。
- 为了支持乱序执行，我们将5段流水线的译码阶段再分为两个阶段：

    - **流出**（Issue，IS）：指令译码，检查是否存在结构冲突。（in-order issue)
    - **读操作数**（Read Operands，RO）：**等待数据冲突消失**，然后读操作数(out of order execution)
    - 

    - 在前述5段流水线中，是不会发生WAR冲突和WAW冲突的。但乱序执行就使得它们可能发生了。
- **动态调度的流水线支持多条指令同时处于执行当中。**
    - 要求：具有多个功能部件、或者功能部件流水化、或者兼而有之。
    - 指令乱序完成带来的最大问题:**异常处理比较复杂**
    - 动态调度的处理机要保持正确的异常行为:对于一条会产生异常的指令来说，只有当处理机确切地知道该指令将被执行时，才允许它产生异常。
    - 即使保持了正确的异常行为，动态调度处理机仍可能发生不精确异常。
        - **不精确异常**：当执行指令i导致发生异常时，处理机的现场（状态）与严格按程序顺序执行时指令i的现场不同。
            - 发生不精确异常的两个原因：因为当发生异常（设为指令i）时：
                - 流水线可能已经执行完按程序顺序是位于指令i之后的指令；
                - 流水线可能还没完成按程序顺序时指令i之前的指令。
            - 不精确异常使得在异常处理后难以接着继续执行程序。
        - **精确异常**：如果发生异常时，处理机的现场跟严格按程序顺序执行时指令i的现场相同。
- **记分牌算法**和**Tomasulo算法**是两种比较典型的动态调度算法。

5.3.2 记分牌动态调度算法

- 记分牌的目标：**在没有结构冲突时，尽可能早地执行没有数据冲突的指令，实现每个时钟周期执行一条指令。**
- **记分牌负责和管理指令的流出和执行，也包括检测所有的冲突。**
- 要发挥指令乱序执行的好处，必须有多条指令同时处于执行阶段。
    - 比如:

CDC 6600具有16个独立的功能部件

- 4个浮点部件
- 5个访存部件
- 7个整数操作部件

- 采用了记分牌的MIPS处理器的基本结构
    - 每条指令都要经过记分牌
    - 记分牌负责相关检测并控制指令的流出和执行。

指令在采用记分牌的流水线中的处理步骤

每条指令的执行过程分为4段（主要考虑**浮点操作**）
（记分牌主要用于浮点部件才有意义，因为其他部件的操作延迟都很小）

1. **流出**
   - 如果当前流出指令所需的功能部件空闲，并且所有其他正在执行的指令的目的寄存器与该指令的不同，记分牌就向功能部件流出该指令，并修改记分牌内部的记录表。
   - 解决了WAW冲突
2. **读操作数**
   - 记分牌监测源操作数的可用性，如果数据可用，它就通知功能部件从寄存器中读出源操作数并开始执行。
   - 动态地解决了RAW冲突，并导致指令可能乱序开始。
   - 流出和读操作数段合起来完成了5段流水线的ID的功能。
3. 执行
   - 取到操作数后，功能部件开始执行。当产生出结果后，就通知记分牌它已经完成执行。
   - 在浮点流水线中，这一段可能要占用多个时钟周期。
   - 这步相当于5段流水线中的EX段
4. 写结果
   - 记分牌一旦知道执行部件完成了执行，就检测是否存在WAR冲突。如果不存在，或者原有的WAR冲突已消失，记分牌就通知功能部件把结果写入目的寄存器，并释放该指令使用的所有资源。
   - 这步这步相当于5段流水线中的WB段
   - 如果检测到WAR冲突，就不允许该指令将结果写到目的寄存器。这发生在以下情况：
     - 前面的某条指令（按顺序流出）还没有读取操作数；
     - 而且：其中某个源操作数寄存器与本指令的目的寄存器相同。
   - 在这种情况下，记分牌必须等待，直到该冲突消失。

记分牌中记录信息的构成

- **指令状态表**：记录正在执行的各条指令已经进入到了哪一段。
- **功能部件状态表**：记录各个功能部件的状态。每个功能部件有一项，每一项由以下9个字段组成：
  - Busy：忙标志，指出功能部件是否忙。初值为“no；
  - Op：该功能部件正在执行或将要执行的操作；
  - Fi：目的寄存器编号；
  - Fj，Fk：源寄存器编号；
  - Qj，Qk：指出向源寄存器Fj、Fk写数据的功能部件；
  - Rj，Rk：标志位，为“yes”表示Fj，Fk中的操作数就绪且还未被取走。否则就被置为“no”。
- **结果寄存器状态表Result**：每个寄存器在该表中有一项，用于指出哪个功能部件（编号）将把结果写入该寄存器。
  - 如果当前正在运行的指令都不以它为目的寄存器，则其相应项置为“no”。
  - Result各项的初值为“no”（全0）。

记分牌的性能受限于以下几个方面：

- 程序代码中可开发的并行性，即是否存在可以并行执行的不相关的指令。
- 记分牌的容量
  - 记分牌的容量决定了流水线能在多大范围内寻找不相关指令。流水线中可以同时容纳的指令数量称为指令窗口。
- 功能部件的数目和种类
  - 功能部件的总数决定了结构冲突的严重程度。
- 反相关和输出相关。
  - 它们引起计分牌中WAR和WAW冲突。

5.3.3 Tomasulo算法

基本思想

- 记录和检测指令相关，**操作数一旦就绪就立即执行**，把发生RAW冲突的可能性减少到最小；
- 通过**寄存器换名**来消除WAR冲突和WAW冲突

基于Tomasulo算法的MIPS处理器浮点部件的基本结构

- **保留站**（reservation station）
  - 每个保留站中**保存一条已经流出并等待到本功能部件执行的指令**（相关信息）。
  - 包括：操作码、操作数以及用于检测和解决冲突的信息。
    - 在一条指令流出到保留站的时候，如果该指令的源操作数已经在寄存器中就绪，则将之取到该保留站中。
    - 如果操作数还没有计算出来，则在该保留站中记录将产生这个操作数的保留站的标识。
  - 浮点加法器有3个保留站：ADD1，ADD2，ADD3
  - 浮点乘法器有两个保留站：MULT1，MULT2
  - 每个保留站都有一个标识字段，唯一地标识了该保留站。
- **公共数据总线CDB** -（一条重要的数据通路）
  - 所有功能部件的计算结果都是送到CDB上，由它把这些结果直接送到（播送到）各个需要该结果的地方。
  - 在具有多个执行部件且采用多流出（即每个时钟周期流出多条指令）的流水线中，需要采用多条CDB。
- **load缓冲器和store缓冲器**
  - 存放读/写存储器的数据或地址
  - load缓冲器的作用有3个：
    - 存放用于计算有效地址的分量；
    - 记录正在进行的load访存，等待存储器的响应；
    - 保存已经完成了的load的结果（即从存储器取来的数据），等待CDB传输。
  - store缓冲器的作用有3个：
    - 存放用于计算有效地址的分量；
    - 保存正在进行的store访存的目标地址，该store正在等待存储数据的到达；
    - 保存该store的地址和数据，直到存储部件接收。
- **浮点寄存器FP**

    - 共有16个浮点寄存器：F0，F2，F4，…，F30。
    - 它们通过一对总线连接到功能部件，并通过CDB连接到store缓冲器。
- **指令队列**
    - 指令部件送来的指令放入指令队列
    - 指令队列中的指令按先进先出的顺序流出
- **运算部件**
    - 浮点加法器完成加法和减法操作
    - 浮点乘法器完成乘法和除法操作

在Tomasulo算法中，**寄存器换名**是通过**保留站**和**流出逻辑**来共同完成的。

- 当指令流出时，如果其操作数还没有计算出来，则将该指令中相应的寄存器号换名为将产生这个操作数的保留站的标识。
- 指令流出到保留站后，其操作数寄存器号或者换成了数据本身（如果该数据已经就绪），或者换成了保留站的标识，不再与寄存器有关系。

算法特点

- 冲突检测和指令执行控制是分布的。
    - 每个功能部件的保留站中的信息决定了什么时候指令可以在该功能部件开始执行。
- 计算结果通过CDB直接从产生它的保留站传送到所有需要它的功能部件，而不用经过寄存器。

指令执行的步骤

使用Tomasulo算法的流水线需3段：

1. **流出**：从指令队列的头部取一条指令。
    - 如果该指令的操作所要求的保留站有空闲的，就把该指令送到该保留站（设为r）。
        - 如果其操作数在寄存器中已经就绪，就将这些操作数送入保留站r。
        - 如果其操作数还没有就绪，就把将产生该操作数的保留站的标识送入保留站r。
        - 一旦被记录的保留站完成计算，它将直接把数据送给保留站r。
        - （寄存器换名和对操作数进行缓冲，消除WAR冲突）
    - 完成对目标寄存器的预约工作
        - （消除了WAW冲突）
    - 如果没有空闲的保留站，指令就不能流出。
        - （发生了结构冲突）
2. **执行**
    - 当两个操作数都就绪后，本保留站就用相应的功能部件开始执行指令规定的操作。
    - load和store指令的执行需要两个步骤：
        - 计算有效地址（要等到基地址寄存器就绪）
        - 把有效地址放入load或store缓冲器
3. **写结果**
    - 功能部件计算完毕后，就将计算结果放到CDB上，所有等待该计算结果的寄存器和保留站（包括store缓冲器）都同时从CDB上获得所需要的数据。

5.4动态分支预测技术

所开发的ILP越多，控制相关的制约就越大，分支预测就要有更高的准确度。

本节中介绍的方法对于每个时钟周期流出多条指令（若为n条，就称为n流出）的处理机来说非常重要。

- 在n-流出的处理机中，遇到分支指令的可能性增加了n倍。
- 要给处理器连续提供指令，就需要准确地预测分支。

动态分支预测：在程序运行时，根据分支指令过去的表现来预测其将来的行为。

- 如果分支行为发生了变化，预测结果也跟着改变。
- 相比静态预测，有更好的预测准确度和适应性。

分支预测的有效性取决于：

- 预测的准确性
- 预测正确和不正确两种情况下的分支开销
    - 决定分支开销的因素：
        - 流水线的结构
        - 预测的方法
        - 预测错误时的恢复策略等

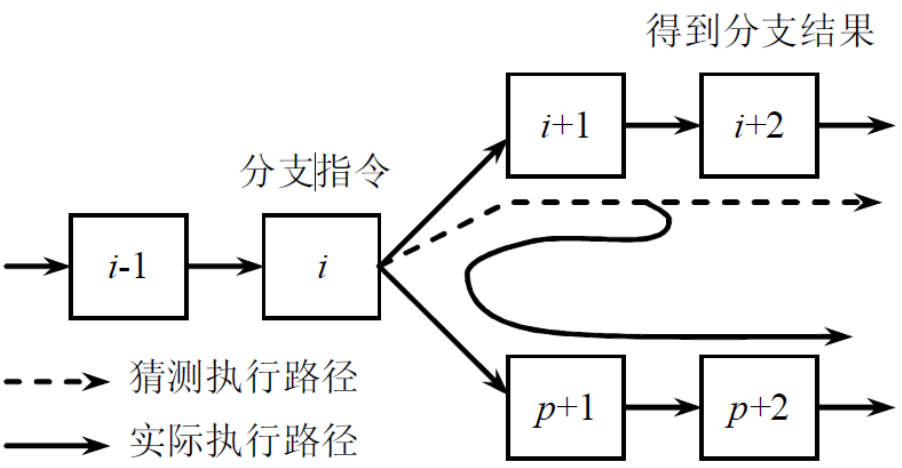
采用动态分支预测技术的目的

- 预测分支是否成功
- 尽快找到分支目标地址（或指令）
    - （避免控制相关造成流水线停顿）

需要解决的关键问题:

- 如何记录分支的历史信息，要记录哪些信息
- 如何根据这些信息来预测分支的去向，甚至提前
  取出分支目标处的指令

在预测错误时，要作废已经预取和分析的指令，恢复现场，并从另一条分支路径重新取指令。



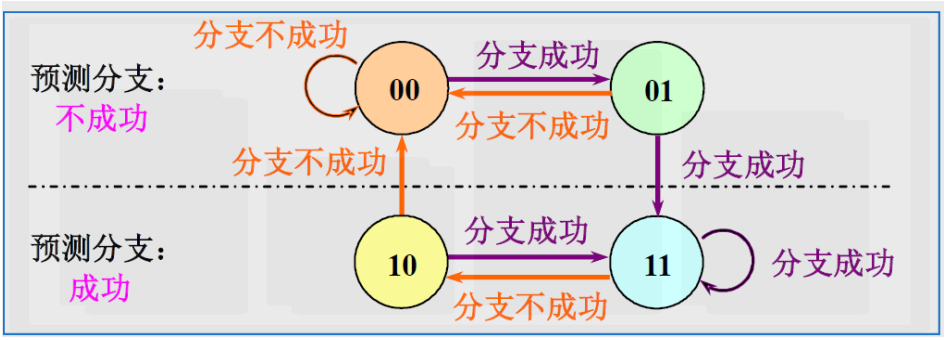
采用分支历史表BHT

分支历史表BHT（Branch History Table）

- 最简单的动态分支预测方法
- 用BHT来记录分支指令最近一次或几次的执行情况（成功还是失败），并据此进行预测

分类

1. 只有1个预测位的分支预测
   - 记录分支指令最近一次的历史，BHT中只需要1位二进制位。
2. 采用**两位二进制位**来记录历史
   - 提高预测的准确度
   - 研究结果表明：两位分支预测的性能与n位（n>2）分支预测的性能差不多。
   - 两位分支预测的状态转换如下所示：



   - 步骤:
     - **分支预测**
       - 当分支指令到达译码段（ID）时，根据从BHT读出的信息进行分支预测
       - 若预测正确，就继续处理后续的指令，流水线没有断流。否则，就要作废已经预取和分析的指令，恢复现场，并从另一条分支路径重新取指令
     - **状态修改**

BHT方法

- BHT方法中，**只对分支是否成功进行预测，对分支目标地址没有提供支持**
  - 适用于：在判定分支是否成功所需的时间大于确定分支目标地址所需的时间。
- 研究结果表明：对于SPEC89测试程序来说，具有大小为4KB的BHT的预测准确率为82% ~ 99%。
  - 一般来说，采用4KB的BHT就可以了。
- BHT可以跟分支指令一起存放在指令Cache中，也可以用一块专门的硬件来实现。

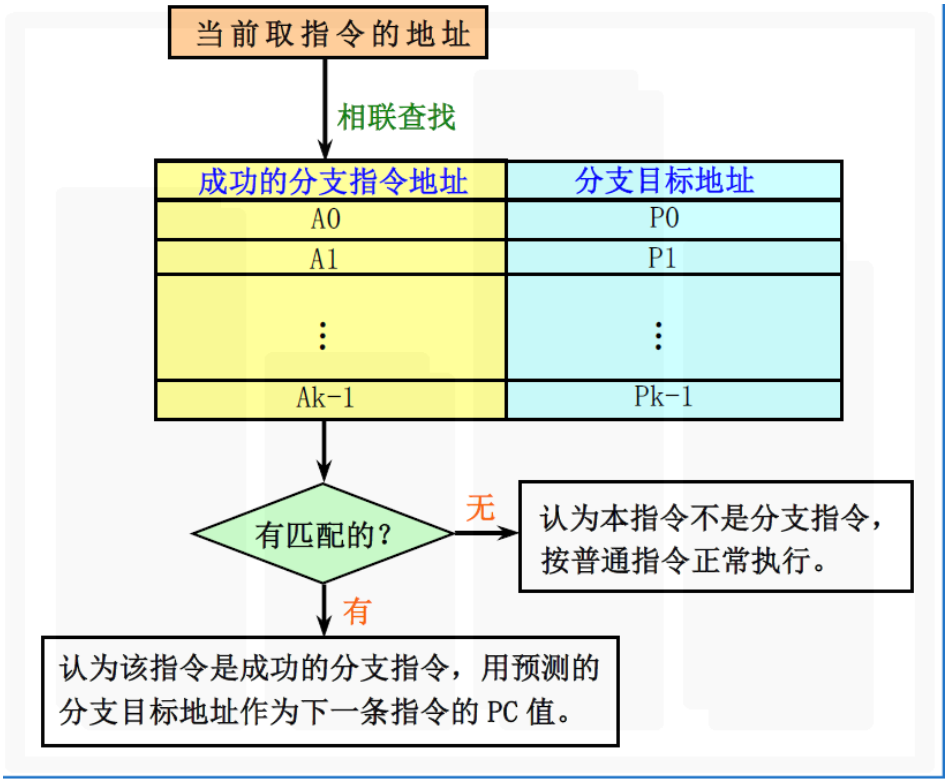
采用分支目标缓冲器BTB

目标：将分支的开销降为0

方法：分支目标缓冲

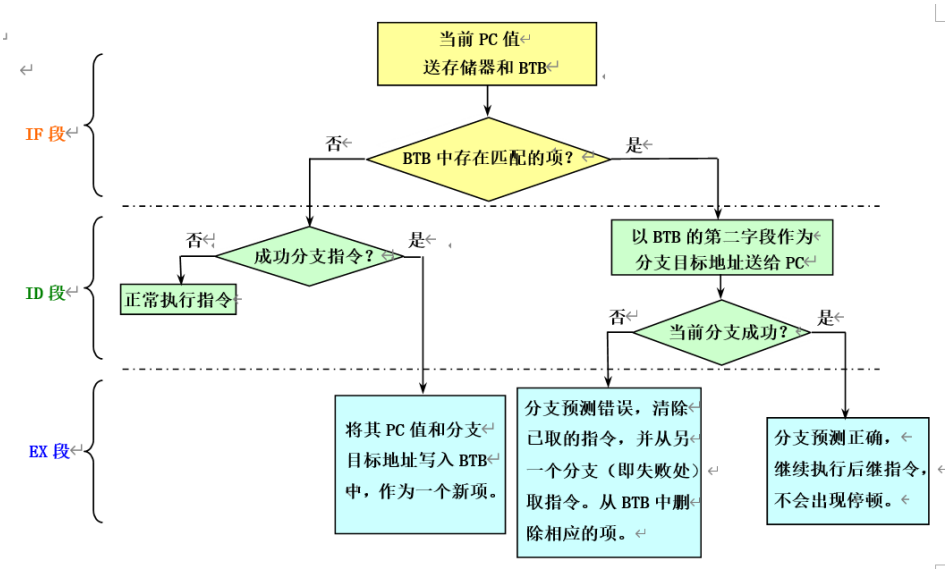
- 将分支成功的分支指令的地址和它的分支目标地址都放到一个缓冲区中保存起来，缓冲区以分支指令的地址作为标识。
- 这个缓冲区就是分支目标缓冲器（Branch-Target Buffer，简记为BTB，或者分支目标Cache（Branch-Target Cache））。

BTB的结构



- 看成是用专门的硬件实现的一张表格
- 表格中的每一项至少有两个字段：
  - 执行过的成功分支指令的地址（作为该表的匹配标识）
  - 预测的分支目标地址。

采用BTB后，在流水线各个阶段所进行的相关操作：

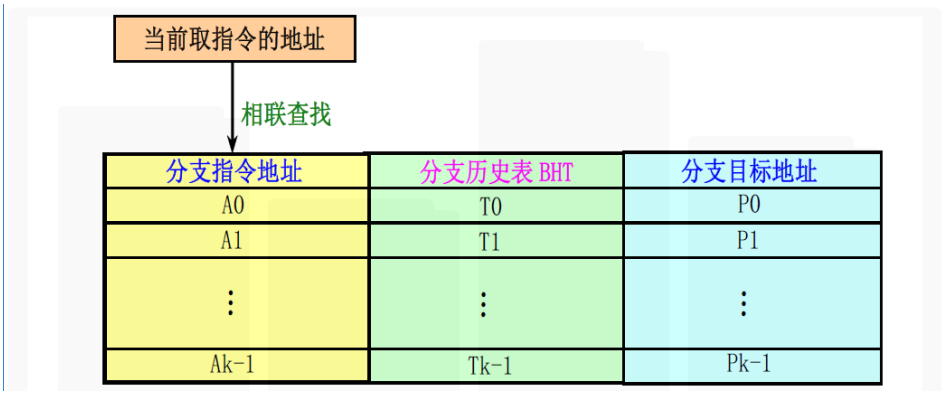


采用BTB后，各种可能情况下的延迟 ：

指令在BTB中？	预测	实际情况	延迟周期
是	成功	成功	0
是	成功	不成功	2
不是		成功	2
不是		不成功	0

BTB的另一种形式

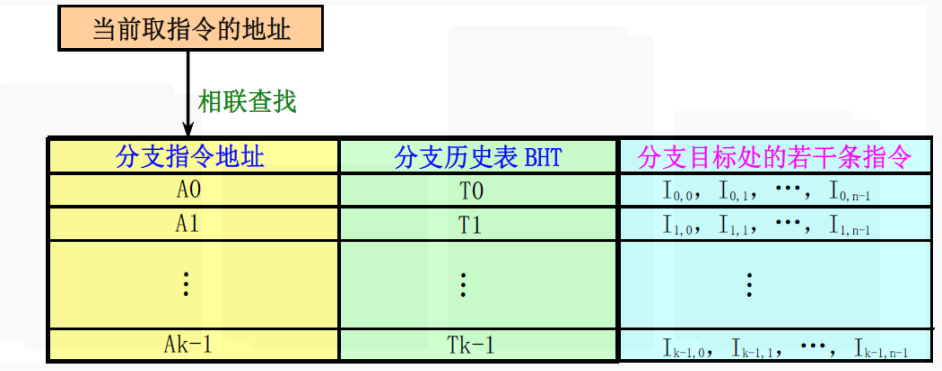
在分支目标缓冲器中增设一个至少是两位的“分支历史表”字段



- 分支历史表是用来预测转移方向
- 这种方法是BTB和BHT的结合

另一种形式

更进一步，在表中对于每条分支指令都存放若干条分支目标处的指令，就形成了**分支目标指令缓冲器**。



基于硬件的前瞻执行

前瞻执行（speculation）的基本思想：对分支指令的结果进行猜测，并假设这个猜测总是对的，然后按这个猜测结果继续取、流出和执行后续的指令。

只是执行指令的结果不是写回到寄存器或存储器，而是写入一个称为**再定序缓冲器ROB（ReOrder Buffer）**中。

等到相应的指令得到“确认”（commit）（即确实是应该执行的）之后，才将结果写入寄存器或存储器。

基于硬件的前瞻执行结合了3种思想

- 动态分支预测。用来选择后续执行的指令。
- 在控制相关的结果尚未出来之前，前瞻地执行后续指令。
- 用动态调度对基本块的各种组合进行跨基本块的调度。

对Tomasulo算法加以扩充，就可以支持前瞻执行

把Tomasulo算法的写结果和指令完成加以区分，分成两个不同的段：

- **写结果段**
  - 把前瞻执行的结果写到ROB中；
  - 通过CDB在指令之间传送结果，供需要用到这些结果的指令使用。
- **指令确认段**
  - 在分支指令的结果出来后，对相应指令的前瞻执行给予确认。
  - 如果前面所做的猜测是对的，把在ROB中的结果写到寄存器或存储器。
  - 如果发现前面对分支结果的猜测是错误的，那就不予以确认，并从那条分支指令的另一条路径开始重新执行。

实现前瞻的关键思想：

允许指令乱序执行，但必须顺序确认。

在指令被确认之前，不允许它进行不可恢复的操作。

ROB中每一项的组成字段

- 指令类型
  - 指出该指令是分支指令、store指令或寄存器操作指令。
- 目标地址
  - 给出指令执行结果应写入的目标寄存器号（如果是load和ALU指令）或存储器单元的地址（如果是store指令）。
- 数据值字段
  - 用来保存指令前瞻执行的结果，直到指令得到确认。
- 就绪字段
  - 指出指令是否已经完成执行并且数据已就绪。

Tomasulo算法中保留站的**换名功能**是由**ROB**来完成的。

采用前瞻执行机制后，指令的执行步骤：

（在Tomasulo算法的基础上改造的）

- **流出**
  - 从浮点指令队列的头部取一条指令。
  - 如果有空闲的保留站（设为r）且有空闲的ROB项（设为b），就流出该指令，并把相应的信息放入保留站r和ROB项b。
  - 如果保留站或ROB全满，便停止流出指令，直到它们都有空闲的项。
- **执行**
  - 如果有操作数尚未就绪，就等待，并不断地监测CDB。(检测RAW冲突)
  - 当两个操作数都已在保留站中就绪后，就可以执行该指令的操作。
    - load指令操作分两步：有效地址计算，读取数据
    - store指令进行有效地址计算
- **写结果**
  - 当结果产生后，将该结果连同本指令在流出段所分配到的ROB项的编号放到CDB上，经CDB写到ROB以及所有等待该结果的保留站。
  - 释放产生该结果的保留站。
  - store指令在本阶段完成，其操作为:
    - 如果要写入存储器的数据已经就绪，就把该数据写入分配给该store指令的ROB项。
    - 否则，就监测CDB，直到那个数据在CDB上播送出来，才将之写入分配给该store指令的ROB项。
- **确认**
  - 对分支指令、store指令以及其它指令的处理不同：
  - **其它指令**（除分支指令和store指令）
    - 当该指令到达ROB队列的头部而且其结果已经就绪时，就把该结果写入该指令的目的寄存器，并从ROB中删除该指令。
  - **store指令**
    - 处理与上面的类似，只是它把结果写入存储器。
  - **分支指令**
    - 当预测错误的分支指令到达ROB队列的头部时，清空ROB，并从分支指令的另一个分支重新开始执行。
      - （错误的前瞻执行）
    - 当预测正确的分支指令到达ROB队列的头部时，该指令执行完毕。

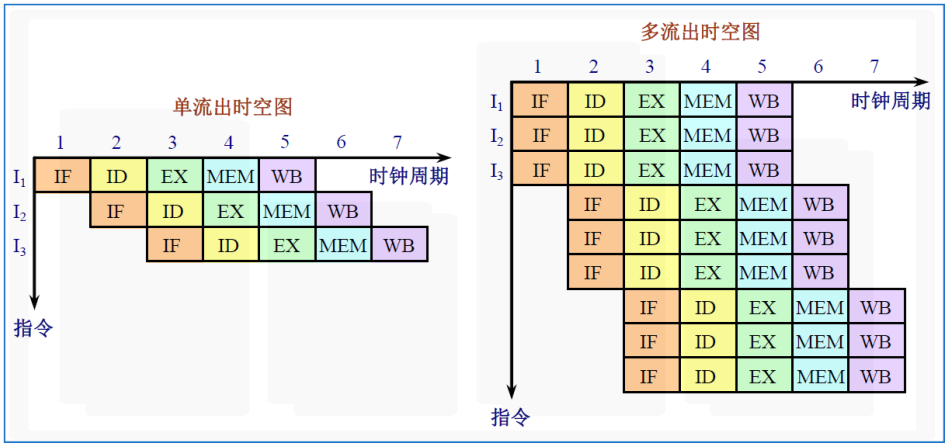
前瞻执行

- 通过ROB实现了指令的顺序完成。
- 能够实现精确异常。
- 很容易地推广到整数寄存器和整数功能单元上。
- 主要缺点：所需的硬件太复杂。

5.5多指令流出技术

在每个时钟周期内流出多条指令，CPI < 1。

单流出和多流出处理机执行指令的时空图对比



单流出和多流出处理机执行指令的时空图

多流出处理机有两种基本风格

- **超标量**
  - 在每个时钟周期流出的指令条数不固定，依代码的具体情况而定。（有上限）
  - 设这个上限为n，就称该处理机为n-流出。
  - 可以通过编译器进行静态调度，也可以基于Tomasulo算法进行动态调度。
- **超长指令字VLIM**
  - 在每个时钟周期流出的指令条数是固定的，这些指令构成一条长指令或者一个指令包。
  - 指令包中，指令之间的并行性是通过指令显式地表示出来的。
  - 指令调度是由编译器静态完成的。

超标量处理机与VLIW处理机相比有两个优点：

- 超标量结构对程序员是透明的，处理机能自己检测下一条指令能否流出，不需要由编译器或专门的变换程序对程序中的指令进行重新排列；
- 即使是没有经过编译器针对超标量结构进行调度优化的代码或是旧的编译器生成的代码也可以运行，当然运行的效果不会很好。

基于静态调度的多流出技术

- 在典型的超标量处理器中，每个时钟周期可流出1到8条指令。
- 指令按序流出，在流出时进行冲突检测。
  - 由硬件检测当前流出的指令之间是否存在冲突以及当前流出的指令与正在执行的指令是否有冲突。

举例：一个4-流出的静态调度超标量处理机

- **在取指令阶段，流水线将从取指令部件收到1 ~ 4条指令**（称为流出包）。
  - 在一个时钟周期内，这些指令有可能是全部都能流出，也可能是只有一部分能流出。
- **流出部件检测结构冲突或者数据冲突。**
  - 一般分两阶段实现：
  - 第一段：进行流出包内的冲突检测，选出初步判定可以流出的指令；

- 第二段：检测所选出的指令与正在执行的指令是否有冲突。

MIPS处理机实现超标量

假设：每个时钟周期流出两条指令：

1条整数型指令 + 1条浮点操作指令

其中：把load指令、store指令、分支指令归类为整数型指令。

1. 要求：
   - 同时取两条指令（64位），译码两条指令（64位）
2. 对指令的处理包括以下步骤：
   1. 从Cache中取两条指令；
   2. 确定哪几条指令可以流出（0 ~ 2条指令）
   3. 把它们发送到相应的功能部件
3. 双流出超标量流水线中指令执行的时空图
   - 假设：所有的浮点指令都是加法指令，其执行时间为两个时钟周期
   - 为简单起见，图中总是把整数指令放在浮点指令的前面
   -

指令类型	流水线工作情况							
整数指令	IF	ID	EX	MEM	WB			
浮点指令	IF	ID	EX	EX	MEM	WB		
整数指令		IF	ID	EX	MEM	WB		
浮点指令		IF	ID	EX	EX	MEM	WB	
整数指令			IF	ID	EX	MEM	WB	
浮点指令			IF	ID	EX	EX	MEM	WB
整数指令				IF	ID	EX	MEM	WB
浮点指令				IF	ID	EX	EX	MEM

双流出超标量处理机特点

1. 采用“**1条整数型指令 + 1条浮点指令**”并行流出的方式，需要增加的硬件很少。
2. 浮点load或浮点store指令将使用整数部件，会增加对浮点寄存器的访问冲突。
   - 增设一个浮点寄存器的读/写端口。
3. 由于流水线中的指令多了一倍，定向路径也要增加。

5.5.3 超长指令字流水技术

- 把能并行执行的多条指令组装成一条很长的指令（100多位到几百位）
- 设置多个功能部件；
- 指令字被分割成一些字段，每个字段称为一个操作槽，直接独立地控制一个功能部件；
- 在超长指令字处理机中，在指令流出时不需要进行复杂的冲突检测，而是依靠编译器全部安排好了。

存在的问题:

- **程序代码长度增加了**
  - 提高并行性而进行的大量的循环展开；
  - 指令字中的操作槽并非总能填满
    - 解决：采用指令共享立即数字段的方法，或者采用指令压缩存储、调入Cache或译码时展开的方法。
- **采用了锁步机制**
  - 任何一个操作部件出现停顿时，整个处理机都要停顿。
  - 可以设置适当的硬件动态检测机制，允许指令流出后非同步执行。
- **机器代码的不兼容性**
  - 可以采用机器代码翻译或仿真的方法。

5.5.4 多流出处理器收到的限制

- 程序所固有的指令级并行性
- 硬件实现上的困难
- 超标量和超长指令字处理器固有的技术限制

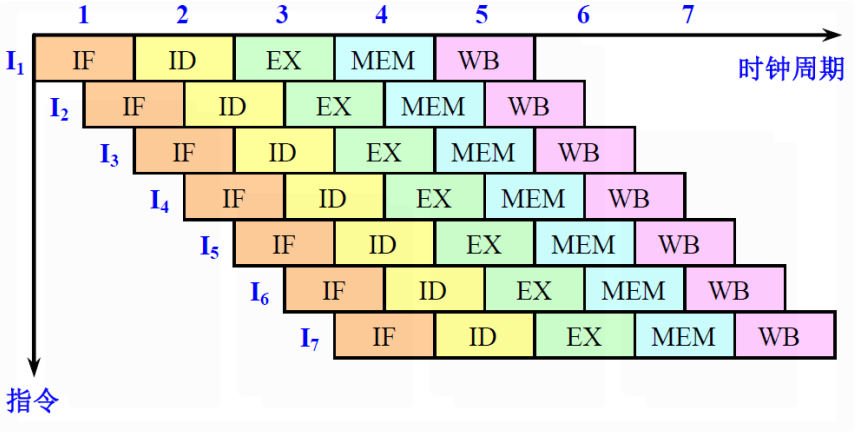
5.5.5 超流水线处理机

将每个流水段进一步细分，这样在一个时钟周期内能够分时流出多条指令。这种处理机称为**超流水线处理机。**

对于一台每个时钟周期能流出n条指令的超流水线计算机来说，这n条指令不是同时流出的，而是每隔1/n个时钟周期流出一条指令。

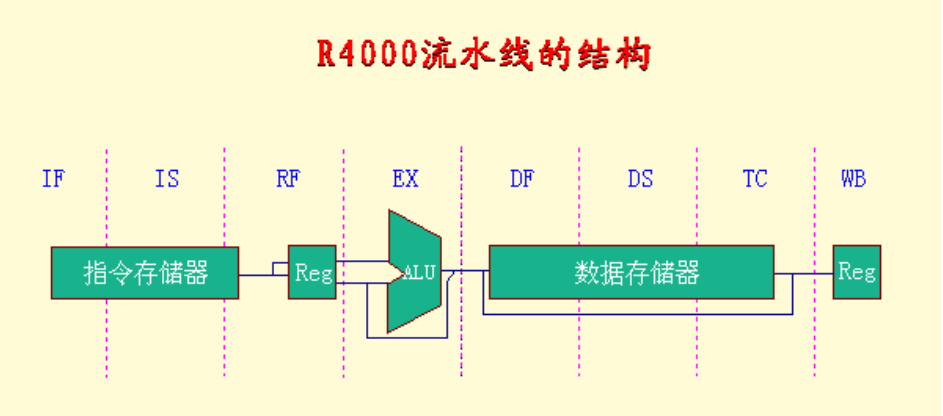
- 实际上该超流水线计算机的流水线周期为1/n个时钟周期。

一台每个时钟周期分时流出两条指令的超流水线计算机的时空图。



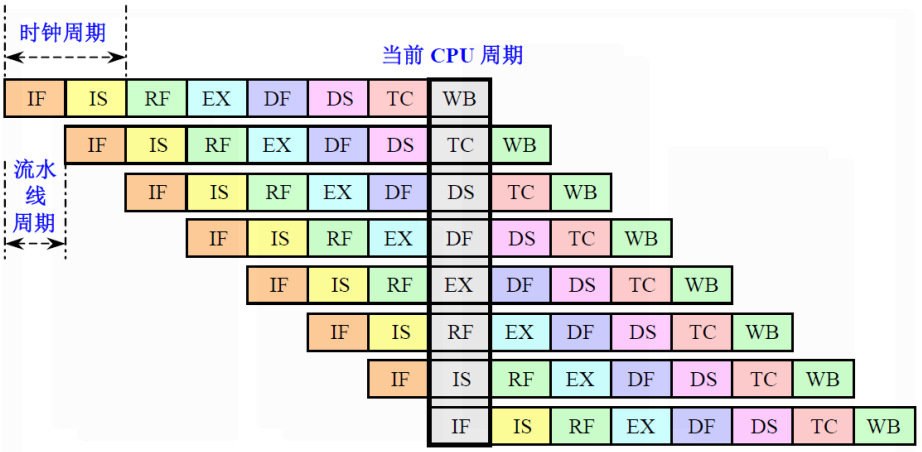
在有的资料上，把指令流水线级数为**8或8以上**的流水线处理机称为超流水线处理机。

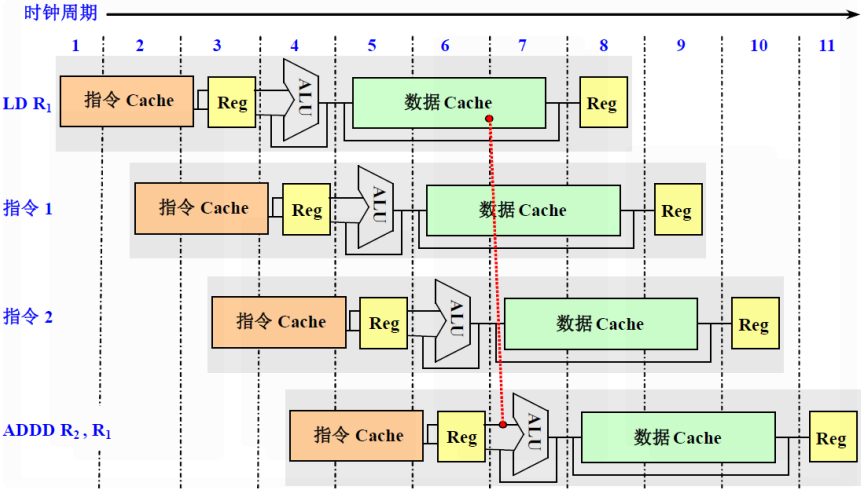
典型的超流水线处理器：SGI公司的MIPS系列R4000

- R4000微处理器芯片内有2个Cache：
  - 指令Cache和数据Cache
  - 容量都是8KB
  - 每个Cache的数据宽度为64位
- R4000的核心处理部件：整数部件
  - 一个32×32位的通用寄存器组
  - 一个算术逻辑部件（ALU）
  - 一个专用的乘法/除法部件
- 浮点部件
  - 一个执行部件 --它们可以并行工作
    - 浮点乘法部件
    - 浮点除法部件
    - 浮点加法/转换/求平方根部件
  - 一个16×64位的浮点通用寄存器组。浮点通用寄存器组也可以设置成32个32位的浮点寄存器
- R4000的指令流水线有8级
  - 

  - ➢ 各级的功能
    - IF：取指令的前半步，根据PC值去启动对指令Cache的访问。
    - IS：取指令的后半步，在这一级完成对指令Cache的访问。
    - RF：指令译码，访问寄存器组读取操作数，冲突检测，并判断指令Cache是否命中。
    - EX：指令执行。包括：有效地址计算，ALU操作，分支目标地址计算，条件码测试。
    - DF：取数据的前半步，启动对数据Cache的访问。
    - DS：取数据的后半步，在这一级完成对数据Cache的访问。
    - TC：标识比较，判断对数据Cache的访问是否命中。
    - WB：load指令或运算型指令把结果写回寄存器组。
  -

➢ MIPS R4000指令流水线时空图





7.1 存储系统的基本知识

7.1.1 存储系统的层次结构

满足对三个指标的要求:

- 容量大
- 速度快
- 价格低

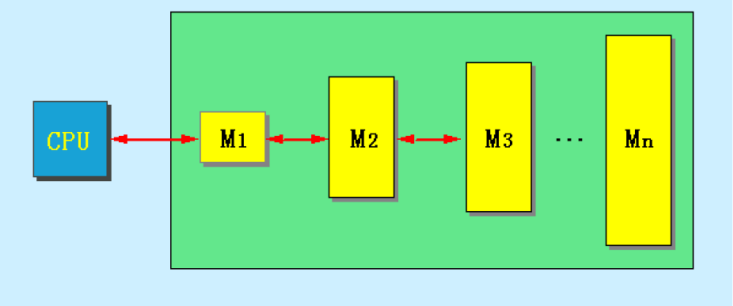
采用多级存储技术,构成多级存储结构层次

理论依据----程序的局部性原理

对于绝大多数程序来说，程序所访问的指令和数据在地址上不是均匀分布的，而是相对簇聚的。

- **时间局部性**：程序马上将要用到的信息很可能就是现在正在使用的信息。
- **空间局部性**：程序马上将要用到的信息很可能与现在正在使用的信息在存储空间上是相邻的。

存储系统的多级层次结构



多级存储层次

假设第 i 个存储器 $\mathrm{M_i}$ 的访问时间为 $\mathrm{T_i}$, 容量为 S_i , 平均每位价格为 $\mathrm{C_i}$, 则
访问时间: $\mathrm{T_1 < T_2 < \ldots < T_n}$
容量: $\mathrm{S_1 < S_2 < \ldots < S_n}$
平均每位价格: $\mathrm{C_1 > C_2 > \ldots > C_n}$
整个存储系统要达到的目标：**从CPU来看, 该存 储系统的速度接近于 M_1 的, 而容量和每位价格都 拉近于 M_n的。**
存储器越靠近CPU, 则 CPU对它的访问频度越高, 而且最好大多数的访问都能在 $\mathrm{M_1}$ 完成。

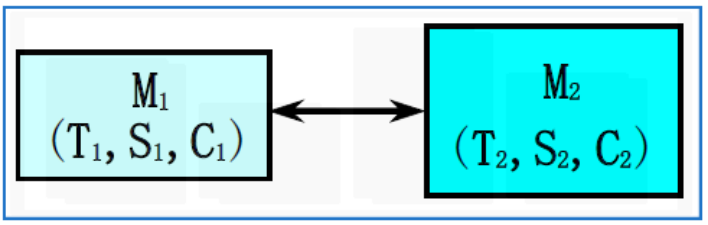
根据局部性原理，我们只要 把近期内CPU访问的程序和数据放在M1中 ， 就能使CPU对存储系统的**绝大多数访问都能在M1中命中。**

存储器之间满足**包容关系**:

- 任何一层存储器中的内容都是其下一层存储器中的内容的子集。
- CPU在访存时，首先访问M_1，如果在M1中找不到所要的数据，就访问M_2，将包含所需数据的块或页调入M_1，如果在M_2中还找不到，就要访问M_3，以此类推。

7.1.2 存储层次的性能参数

假设仅仅有两级存储层次:



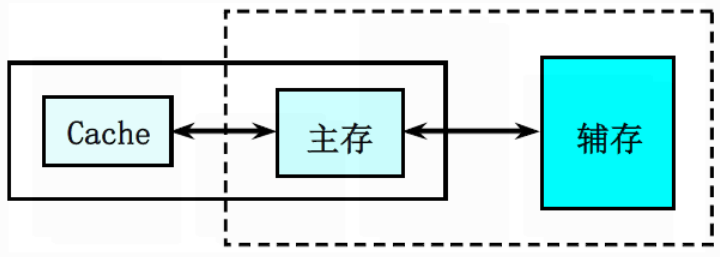
1. **存储容量S**
   - 一般来说，整个存储系统的容量即是第二级存储器M_2的容量，即$S = S_2$
2. **平均每位价格C**
   - $$C = \frac{C_1S_1 + C_2S_2}{S_1 + S_2}$$
   - $$当S_1 << S_2时, C \approx C_2$$
3. **命中率**
   - CPU访问存储系统时,在M_1中找到所需信息的概率
   - $$H = \frac{N_1}{N_1 + N_2}$$
     - N_1为访问M_1的次数
     - N_2为访问M_2的次数
   - **不命中率**：$F = 1 - H$
4. 平均访问时间T_A
   - 分两种情况来考虑CPU的一次访存
   - 当命中时，访问时间即为T_1（**命中时间**）
   - 当不命中时,访问时间为:
     - $$T_2 + T_B + T_1 = T_1 + T_M$$
     - $$T_M = T_2 + T_B$$
     - 其中:T_M为不命中开销:也就是从M_2发送请求到把整个数据块调入M_1中所需要的时间
     - 传送一个信息块所需要的时间为T_B
   - 所以存储系统的平均访问时间为:
     - $$\begin{aligned} \mathrm{T_A} &= \mathrm{HT_1} + (1 - \mathrm{H})(\mathrm{T_1} + \mathrm{T_M}) \\ &= \mathrm{T_1} + (1 - \mathrm{H})\mathrm{T_M} \\ 或\ \mathrm{T_A} &= \mathrm{T_1} + \mathrm{FT_M} \end{aligned}$$

7.1.3 三级存储系统 ☆☆

三级存储系统分别为:

- **Cache(高速缓冲存储器)**
- **主存储器**
- **磁盘存储器(辅存)**

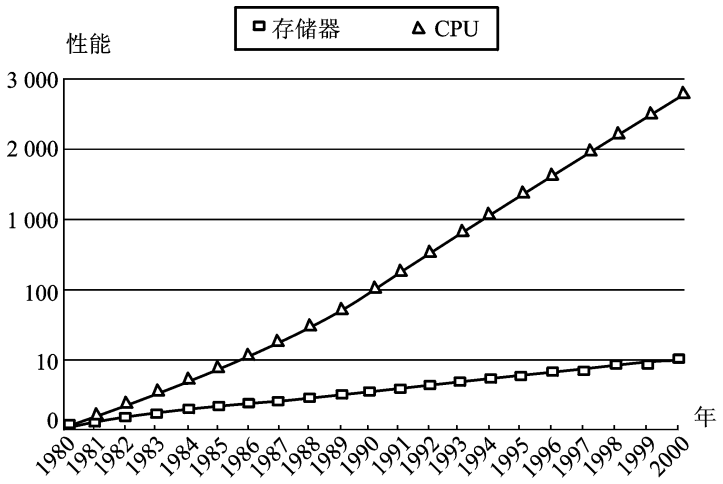
可以看成是由“**Cache—主存**”层次和“**主存—辅存**”层次构成的系统

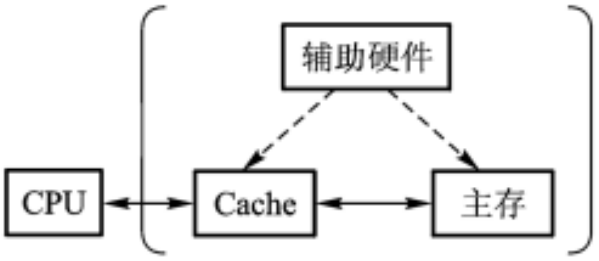


从主存的角度来看:

- cache-主存:弥补主存速度的不足
- 主存-辅存:弥补主存容量的不足

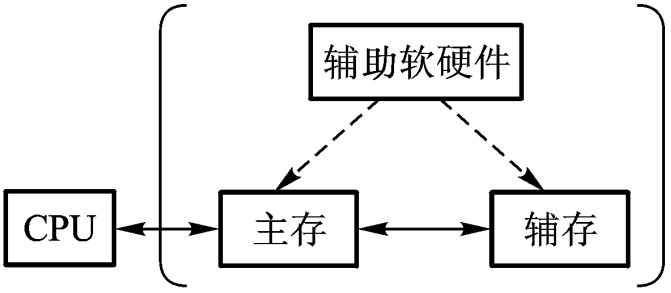
cache-主存层次 ☆

- 主存与CPU的速度差距
  - 

- 借助于**硬件**，**Cache与主存构成一个有机的整体**，以弥补主存速度的不足。
  - 这个层次的工作由 硬件 实现。
  - 

(a) “Cachc-主存”层次

主存和辅存层次 ☆

- 目的：为了弥补主存容量的不足
- 在主存外面增设一个容量更大每位价格更低、速度更慢的存储器---->**辅存，一般是硬盘**
  - 

(b) “主存-辅存”层次

“Cache - 主存”与“主存 - 辅存”层次的区别 ☆☆

存储层次 / 比较项目	“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(第一级和第二级)	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
不命中时CPU是否切换	不切换	切换到其他进程

7.1.4 存储层次的四个问题

1. 当把一个块调入高一层(靠近CPU)存储器时，可以放在哪些位置上?
    - （**映象规则**）
2. 当所要访问的块在高一层存储器中时，如何找到该块?
    - （**查找算法**）
3. 当发生不命中时，应替换哪一块?
    - （**替换算法**）
4. 当进行写访问时，应进行哪些操作?
    - （**写策略**）

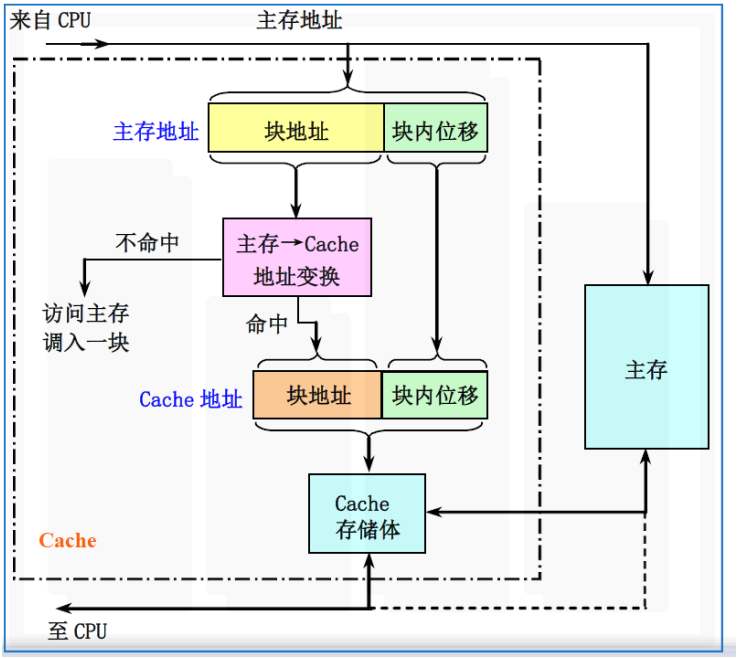
下面将逐渐展开问题的解决办法.....

7.2 Cache基本知识

7.2.1 基本结构和原理

- Cache和主存分块
    - **Cache是按块进行管理的。**Cache和主存均被分割成**大小相同**的块。信息**以块为单位**调入Cache。
        - **主存块地址(块号)** 用于查找该块在主存中的地址,当将主存地址进行变换后,可以用于查找该块在cache中的位置。
        - **块内位移**用于确定所访问的数据在该块中的位置。
        - 

- Cache的基本工作原理示意图
    -



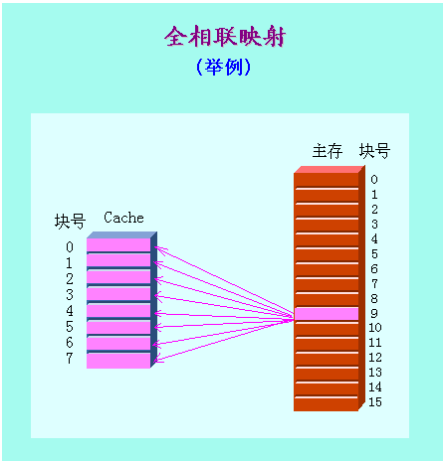
7.2.2 映像规则 ☆☆

---->块从主存调入cache的位置选择

这个和在计算机组成原理中学习的知识是一样的!

全相联映射

- **全相联**：主存中的任一块可以被放置到Cache中的**任意一个位置。**
- 对比：阅览室位置—— 随便坐
- 特点：空间利用率最高，冲突概率最低，实现最复杂。



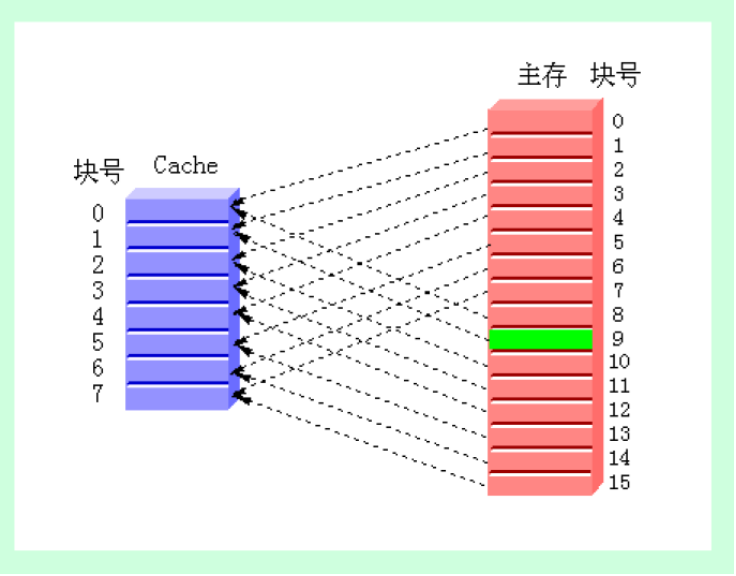
也就是说,主存当中的块可以对应cache中的任意一个cache块

直接映射

- **直接映象**：主存中的每一块只能被放置到Cache中**唯一的一个位置。**
- 对比：阅览室位置—— 一个位置固定学号**尾数为特定数值**的学生才能坐。
- 特点：空间利用率最低，冲突概率最高，实现最简单。
- 对于主存的第i块，若它映象到Cache的第j块，则:
  - $$j = i \mod (M)$$（M为$Cache$的块数）

比如:设$M = 2^m$，则当表示为二进制数时，j实际上就是i的低m位:

$j = i \mod (M)$（M为$Cache$的块数）

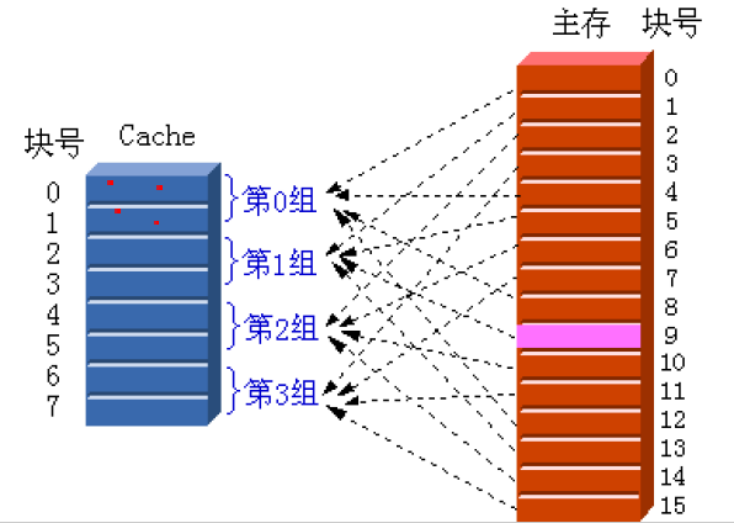


当cache块数为8时,也就是2^3,即主存块号变为二进制后的后三位组成的十进制数,对应于cache中的块号

比如:主存块号为9:(1001),后三位为:001---> 十进制为1,也就是映射到cache中块号为1

组相联映射

- **组相联**：主存中的每一块可以被放置到Cache中唯一的一个组中的任何一个位置。
- 对比：阅览室位置—— 一个区域只允许一个专业的学生（随意）坐。
- 组相联是直接映象和全相联的一种折衷



只不过将上面那个直接相连的M换为组数而已,然后,凡是可以落到这个组里的块在组内是可以进行任意放置的

- 组的选择常采用位选择算法
  - 若主存第i块映象到第k组，则: $k = i \mod （G）$（G为Cache的组数）
  - 设$G = 2^g$，则当表示为二进制数时，k 实际上就是i 的低g 位:
  - 

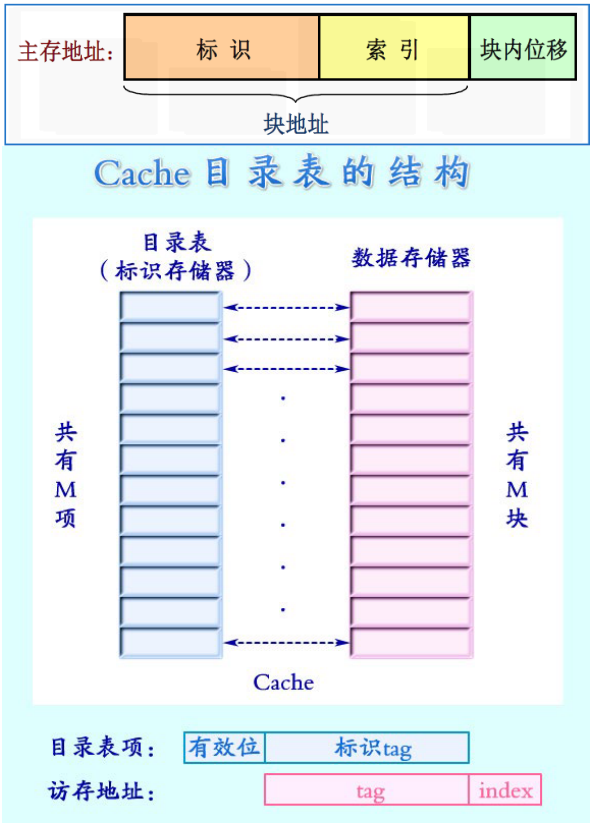
  - 低g位以及直接映象中的低m位通常称为索引。
- **n路组相联**：**每组中有n个块**(n = M/G)，称该映像规则为n路组相联。
  - n 称为相联度
  - 相联度越高，Cache空间的利用率就越高，块冲突概率就越低，不命中率也就越低。

- 绝大多数计算机的Cache:**n ≤4**

7.2.3 查找算法

通过查找目录表来实现

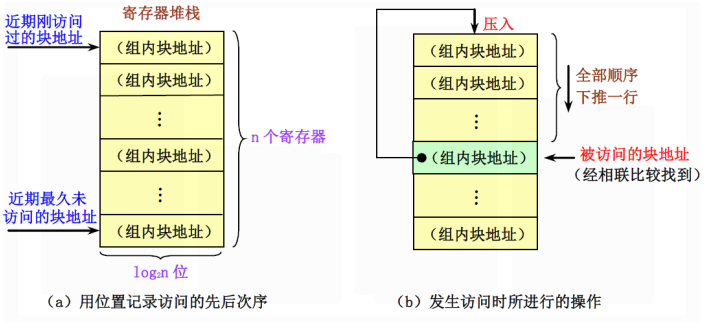
- 目录表的结构
  - 主存块的块地址的高位部分，称为标识（tag）
  - 每个主存块能唯一地由其标识来确定



- 只需查找候选位置所对应的目录表项

7.2.5 替换算法 ☆☆

- 当新调入一块，而Cache又已被占满时，替换哪一块?
  - 直接映象Cache中的替换很简单
    - 因为只有一个块，别无选择。
  - 在组相联和全相联Cache中，则有多个块供选择
    - **随机法** ☆
      - 优点：实现简单
    - **先进先出法FIFO** ☆
      - 类似于队列
    - **最近最少使用法LRU（Least Recently Used）** ☆
      - 选择近期最少被访问的块作为被替换的块。
      - 实际上：选择**最久没有被访问过的块**作为被替换的块。.
    - **LRU**和**随机法**分别因其不命中率低和实现简单而被广泛采用
- LRU算法的硬件实现
  - 用一个堆栈来记录组相联Cache的同一组中各块被访问的先后次序。
  - 用**堆栈元素的物理位置来反映先后次序**
    - 栈底记录的是该组中最早被访问过的块，次栈底记录的是该组中第二个被访问过的块，…，**栈顶记录的是刚访问过的块。**
    - 当需要替换时，从栈底得到应该被替换的块（块地址）
  -



- 堆栈中的内容必须动态更新
  - 当Cache访问命中时，通过用块地址进行相联查找，在堆栈中找到相应的元素，然后把该元素的上面的所有元素下压一个位置，同时把本次访问的块地址抽出来，从最上面压入栈顶。而该元素下面的所有元素则保持不动。
  - 如果Cache访问不命中，则把本次访问的块地址从最上面压入栈顶，堆栈中所有原来的元素都下移一个位置。如果Cache中该组已经没有空闲块，就要替换一个块。这时从栈底被挤出去的块地址就是需要被替换的块的块地址。
- 堆栈法所需要的硬件
  - 需要为每一组都设置一个项数与相联度相同的小堆栈，每一项的位数为log2n位。
- 硬件堆栈所需的功能
  - 相联比较
  - 能全部下移、部分下移和从中间取出一项的功能
- 速度较低，成本较高（只适用于相联度较小的LRU算法）

7.2.6 写策略

“写”操作必须在确认是命中后才可进行

Cache与主存内容的一致性：

- Cache中的内容是主存部分内容的一个副本，
- “写”访问有可能导致Cache和主存内容的不一致

两种写策略 ☆☆

写策略是区分不同Cache设计方案的一个重要标志。

- **写直达法**（也称为存直达法）
  - 执行“写”操作时，不仅写入Cache，而且也写入下一级存储器。
  - 写直达法的优点：易于实现，一致性好。
  - 采用写直达法时，若在**进行“写”操作的过程中CPU必须等待**，直到“写”操作结束，则称**CPU写停顿**。
    - 减少写停顿的一种常用的优化技术：采用写缓冲器
  - **按写分配(写时取)**
    - 写不命中时，先把所写单元所在的块调入Cache，再进行写入。
- **写回法**（也称为拷回法）
  - 执行“写”操作时，只写入Cache。
  - 仅当Cache中相应的块被替换时，才写回主存。
    - (设置“修改位”)
  - 写回法的优点：速度快，所使用的存储器带宽较低。
  - **不按写分配(绕写法):**
    - 写不命中时，直接写入下一级存储器而不调块

7.2.7 Cache的性能分析

- 不命中率
  - 与硬件速度无关
  - 容易产生一些误导
- **平均访存时间** ☆☆

- $$平均访存时间 = 命中时间 + 不命中率 \times 不命中开销$$

- 程序执行时间
  - CPU时间 = （CPU执行周期数+存储器停顿周期数）×时钟周期时间
    - 存储器停顿时钟周期数 = “读”的次数×读不命中率×读不命中开销 + “写”的次数×写不命中率×写不命中开销
    - 存储器停顿时钟周期数 = 访存次数×不命中率×不命中开销

7.2.8 改进Cache的性能

平均访存时间＝命中时间＋不命中率×不命中开销

- 降低不命中率(8种)
- 减少不命中开销(5种)
- 减少Cache命中时间(4种)

7.3 降低Cache不命中率

三种类型的不命中

1. **强制性不命中**(Compulsory miss)
   - 当第一次访问一个块时，该块不在Cache中，需从下一级存储器中调入Cache，这就是强制性不命中。(冷启动不命中，首次访问不命中）
2. **容量不命中**(Capacity miss)
   - 如果**程序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，就会发生不命中。这种不命中称为容量不命中。
3. **冲突不命中**(Conflict miss)
   - 在组相联或直接映象Cache中，**若太多的块映象到同一组(块)中**，则会出现该组中某个块被别的块替换(即使别的组或块有空闲位置)，然后又被重新访问的情况。这就是发生了冲突不命中。
   - (也被称为碰撞不命中，干扰不命中)

- 相联度越高，冲突不命中就越少；
- 强制性不命中和容量不命中不受相联度的影响；
- 强制性不命中不受Cache容量的影响，但容量不命中却随着容量的增加而减少。

减少三种不命中的方法:

- 强制不命中: 增加块大小,预取
  - 本身很少
- 容量不命中:增加容量
  - 抖动现象
- 冲突不命中:提高相联度
  - 全相联映射是最理想的情况

许多降低不命中率的方法会增加命中时间或不命中开销

7.3.1 增加Cache块的大小

当Cache的容量不变时,增加Cache块的大小,不命中率与块大小的关系

- 对于给定的Cache容量，当块大小增加时，不命中率开始是下降，后来反而上升了。
  - 一方面它减少了强制性不命中；()
  - 另一方面，由于增加块大小会减少Cache中块的数目，所以有可能会增加冲突不命中。

Cache容量越大，使不命中率达到最低的块大小就越大。

增加块大小会增加不命中开销

7.3.2 增加Cache的容量

最直接的方法是增加Cache的容量

缺点:

- 增加成本
- 可能增加命中时间

7.3.3 提高相联度

采用相联度超过8的方案的实际意义不大。

2:1 Cache经验规则

容量为N的直接映象Cache的不命中率和容量为N/2的两路组相联Cache的不命中率差不多相同。

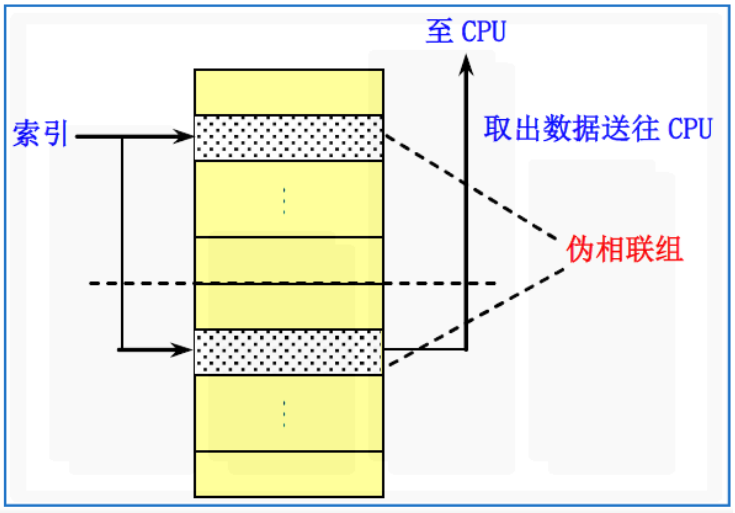
7.3.4 伪相联Cache(列相联Cache)

多路组相联的低不命中率和直接映像的命中速度关系

	优　点	缺　点
直接映像	命中时间小	不命中率高
组相联	不命中率低	命中时间大

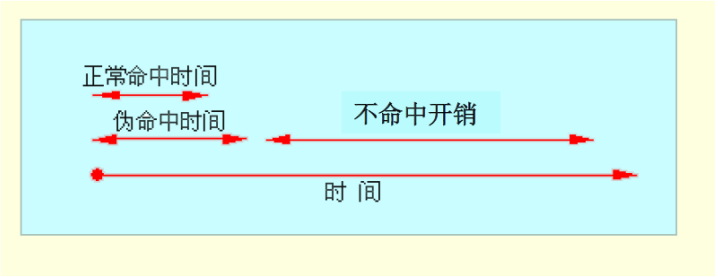
基本思想及工作原理

- 在逻辑上把直接映象Cache的空间上下平分为两个区。
- 对于任何一次访问，**伪相联Cache先按直接映象Cache**的方式去处理。
- 若命中，则其访问过程与直接映象Cache的情况一样。
- 若不命中，则再到另一区相应的位置去查找。
  - 若找到，则发生了伪命中，
  - 否则就只好访问下一级存储器。



快速命中与慢速命中

要保证绝大多数命中都是快速命中。



7.3.5 硬件预取

指令和数据都可以预取，预取内容既可放入Cache，也可放在外缓冲器中。

指令预取通常由Cache之外的**硬件**完成。

预取应利用存储器的空闲带宽，不能影响对正常不命中的处理，否则可能会降低性能。

7.3.6 编译器控制预取

在编译时加入预取指令，在数据被用到之前发出预取请求。

按照预取数据所放的位置，可把预取分为两种类型：

- 寄存器预取：把数据取到寄存器中。
- Cache预取：只将数据取到Cache中

按照预取的处理方式不同，可把预取分为：

- 故障性预取：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。
- 非故障性预取：在遇到这种情况时则不会发生异常，因为这时它会放弃预取，转变为空操作。

在预取数据的同时，处理器应能继续执行。

- 只有这样，预取才有意义。
- 非阻塞Cache (非锁定Cache)：Cache在等待预取数据返回的同时，还能继续为CPU服务。

编译器控制预取的目的:**使执行指令和读取数据能重叠执行。**

7.3.7 编译优化

- 基本思想：通过对软件进行优化来降低不命中率。
- （特色：无需对硬件做任何改动）

程序代码和数据重组

- 可以重新组织程序而不影响程序的正确性
  - 把一个程序中的过程重新排序，就可能会减少冲突不命中，从而降低指令不命中率。
  - 把基本块对齐，使得程序的入口点与Cache块的起始位置对齐，就可以减少顺序代码执行时所发生的Cache（强制）不命中的可能性。
- 如果编译器知道一个分支指令很可能会成功转移，那么它就可以通过以下两步来改善空间局部性
  - 将转移目标处的基本块和紧跟着该分支指令后的基本块进行对调
  - 把该分支指令换为操作语义相反的分支指令
- 相比代码，数据对存储位置的限制更少，更便于调整顺序（用顺序数组代替随机链表）

编译优化技术

- 数组合并

    - 将本来相互独立的多个数组合并成为一个复合数组，以提高访问它们的局部性。
- 内外循环变换
- 循环融合
    - 将若干个独立的循环融合为单个的循环。这些循环访问同样的数组，对相同的数据作不同的运算。这样能使得读入Cache的数据在被替换出去之前，能得到反复的使用。
- 分块---->大量数据打散

7.3.9 牺牲Cache

牺牲Cache -----> 一种能减少冲突不命中次数而又不影响时钟频率的方法。

- 基本思想:
    - 在Cache和它从下一级存储器调数据的通路之间**设置一个全相联的小Cache**，称为“**牺牲”Cache**（Victim Cache）。**用于存放被替换出去的块(称为牺牲者)，以备重用。**
- 作用
    - 对于减小冲突不命中很有效，特别是对于小容量的直接映象数据Cache，作用尤其明显。
    - 例如:项数为4的Victim Cache:能使4KBCache的冲突不命中减少20% ~ 90%

7.4 减少Cache不命中开销

7.4.1 采用两级Cache

- 第一级Cache(L1)小而快
- 第二级Cache(L2)容量大

局部不命中率与全局不命中率 ☆☆

- **局部不命中率** = 该级Cache的不命中次数/到达该级Cache的访问次数
- **全局不命中率** = 该级Cache的不命中次数/CPU发出的访存的总次数

$$全局不命中率_{L2} = 不命中率_{L1} \times 不命中率_{L2}$$

评价第二级Cache时，应使用**全局不命中率**这个指标。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。

采用**两级Cache**时，每条指令的平均访存停顿时间：

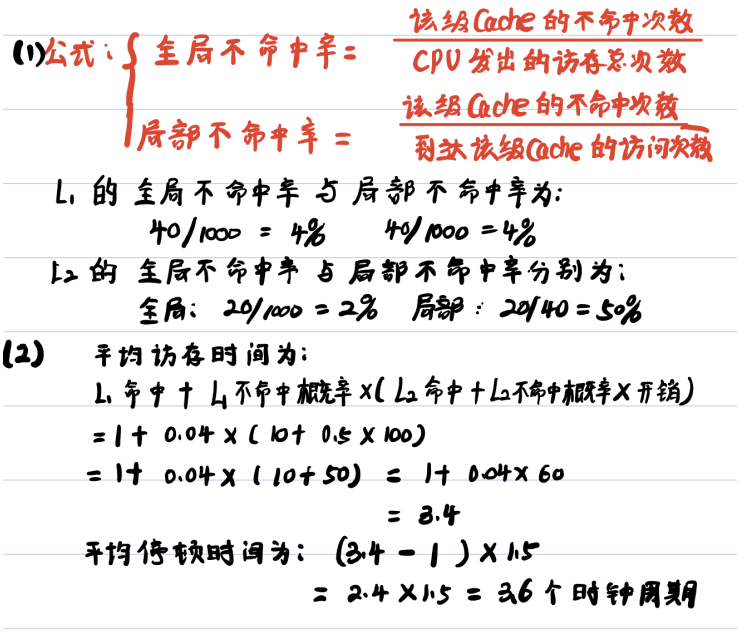
$$\begin{aligned}&每条指令的平均访存停顿时间\\&= 每条指令的平均不命中次数_{L1} \times 命中时间_{L2}\\&+ 每条指令的平均不命中次数_{L2} \times 不命中开销_{L2}\end{aligned}$$

例题:

例7.3 考虑某一两级Cache：第一级Cache为L1，第二级Cache为L2。

（1）假设在1000次访存中，L1的不命中是40次，L2的不命中是20次。求各种局部不命中率和全局不命中率。

（2）假设L2的命中时间是10个时钟周期，L2的不命中开销是100时钟周期，L1的命中时间是1个时钟周期，平均每条指令访存1.5次，不考虑写操作的影响。问：平均访存时间是多少？每条指令的平均停顿时间是多少个时钟周期？



停顿时间,也就是除了第一次cache命中的时间,其它的算是停顿时间

访存时间:一次访问数据所需要的时间(数据可能在cache也可能在主存,也可能在硬盘)

每条指令的访存时间:访存时间*每条指令的访存次数

每条指令的停顿时间:每条指令的访存次数*(停顿时间 = 访存时间-在cache中获得数据的时间)

对于第二级Cache，我们有以下结论：

- 在第二级Cache比第一级Cache大得多的情况下，两级Cache的全局不命中率和容量与第二级Cache相同的单级Cache的不命中率非常接近
- 局部不命中率不是衡量第二级Cache的一个好指标，因此，在评价第二级Cache时，应用全局不命中率这个指标。
- **第二级Cache不会影响CPU的时钟频率**，因此其设计有更大的考虑空间。

第二级cache的参数

- 容量
  - 第二级Cache的容量一般比第一级的大许多。
  - 大容量意味着第二级Cache可能实际上没有容量不命中，只剩下一些强制性不命中和冲突不命中
- 相联度
  - 第二级Cache可采用较高的相联度或伪相联方法
- 块大小
  - 第二级Cache可采用较大的块
  - 为减少平均访存时间，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。
  - **多级包容性**
    - 第一级Cache中的数据总是同时存在于第二级Cache中
    - 能够很方便实现I/O和Cache之间数据一致性

7.4.2 让读不命中优先于写

- Cache中的写缓冲器导致对存储器访问的复杂化
  - 在读不命中时，所读单元的最新值有可能还在写缓冲器中，尚未写入主存
- 解决问题的方法(读不命中的处理)
  - 推迟对读不命中的处理（被动等）
  - 检查写缓冲器中的内容（主动检查）

7.4.3 写缓冲合并

- **提高写缓冲器的效率**

- **写直达Cache**
  - 依靠写缓冲来减少对下一级存储器写操作的时间。
  - 如果写缓冲器为空，就把数据和相应地址写入该缓冲器
    - 从CPU的角度来看，该写操作就算是完成了。
  - 如果写缓冲器中已经有了待写入的数据，就要把这次的写入地址与写缓冲器中已有的所有地址进行比较，看是否有匹配的项。
    - 如果有地址匹配而对应的位置又是空闲的，就把这次要写入的数据与该项合并，这就叫写缓冲合并。
    - 如果写缓冲器满且又没有能进行写合并的项，就必须等待。
  - **使用写合并能够提高写缓冲器的空间利用率，而且还能减少因写缓冲器满而要进行的等待时间。**

7.4.4 请求字处理技术

- **请求字**
  - 从下一级存储器调入Cache的块中，只有一个字是立即需要的。这个字称为请求字。
- 应尽早把请求字发送给CPU
  - **尽早重启动**：调块时，从块的起始位置开始读起。一旦请求字到达，就立即发送给CPU，让CPU继续执行。
  - **请求字优先**：调块时，从请求字所在的位置读起。这样，第一个读出的字便是请求字。将之立即发送给CPU。
- 这种技术在以下情况下效果不大：
  - Cache块较小
  - 下一条指令正好访问同一Cache块的另一部分

7.4.5 非阻塞Cache技术

非阻塞Cache：Cache不命中时仍允许CPU进行其它的命中访问。即允许“不命中下命中”。

- 不命中重叠：
  - 多重不命中下命中
  - 不命中下不命中
    - 存储器必须能够处理多个不命中

可以同时处理的不命中次数越多，所能带来的性能上的提高就越大。但并非不命中次数越多越好。

不足：非阻塞Cache增加了Cache控制器的复杂度，尤其是多重叠非阻塞Cache

7.5 减少命中时间

命中时间直接影响到处理器的时钟频率。在当今的许多计算机中，往往是**Cache的访问时间限制了处理器的时钟频率。**

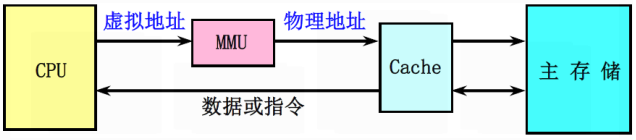
7.5.1 容量小,结构简单的Cache

- 硬件越简单，速度就越快；
- 应使Cache足够小，以便可以与CPU一起放在同一块芯片上。

7.5.2 虚拟Cache

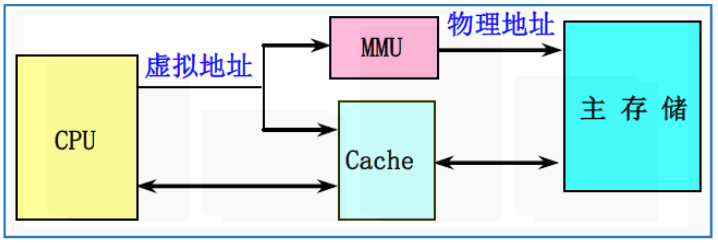
物理Cache VS 虚拟Cache

- **物理Cache**
  - 使用物理地址进行访问的传统Cache
  - 标识存储器中存放的是物理地址，进行地址检测也是用物理地址
  -



物理Cache存储系统

➢ 缺点：地址转换和访问Cache串行进行，访问速度很慢。

- **虚拟Cache**
  - **可以直接用虚拟地址进行访问的Cache**。标识存储器中存放的是虚拟地址，进行地址检测用的也是虚拟地址。
  - 

  - 优点：**在命中时不需要地址转换，省去了地址转换的时间**。即使不命中，地址转换和访问Cache也是并行进行的，其速度比物理Cache快很多。
  - 问题：

    并非所有计算机都采用虚拟Cache。
    - 当进程切换时需要清空Cache。
    - 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。
    - I/O问题，需要把物理地址映像为虚拟地址。

7.5.3 Cache访问流水线

对第一级Cache的访问按照流水的方式进行组织，将Cache的访问分解为多个时钟周期。

目的：提高时钟频率

：Intel的Pentium访问Cache需要1个时钟周期，Pentium Pro和奔三需要2个，奔四则需要4个时钟周期。

单位时间做的事情可以少一点，分成几步去做；步数的增加可以使用流水的方式来抵消。

即并没有直接减少Cache的命中时间，而是提高了单位时间内Cache的输出数据量，及提高了访问Cache的带宽。

7.5.4 跟踪Cache

开发指令级并行性所遇到的一个挑战是：

- 当要每个时钟周期流出超过4条指令时，要提供足够多条彼此互不相关的指令是很困难的

一个解决方法：**采用踪迹Cache**

指令Cache存放CPU所执行的动态指令序列

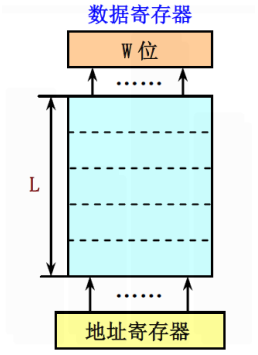
- 包含了由分支预测展开的指令，该分支预测是否正确需要在取到该指令时进行确认

7.6 并行主存系统

主存的主要性能指标：**延迟**和**带宽**

并行主存系统是**在一个访存周期内能并行访问多个存储字的存储器。**

一个单体单字宽的存储器

- 字长与CPU的字长相同。
- 每一次只能访问一个存储字。假设该存储器的访问周期是T_M，字长为W位，则其带宽为：
  - $$B_M = \frac{W}{T_M}$$
- 

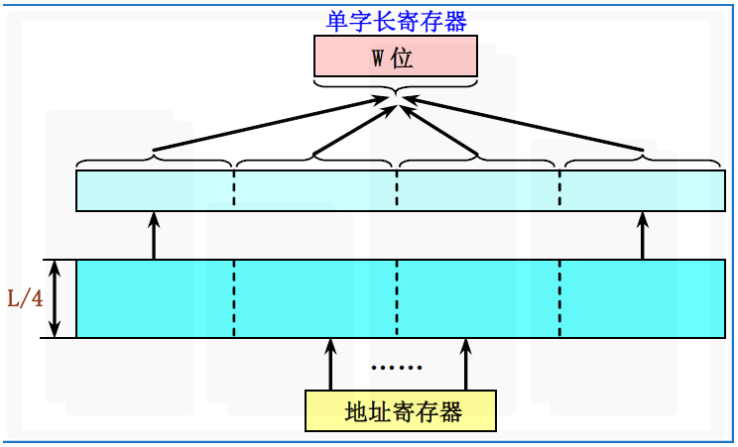
普通存储器

在相同的器件条件（**即TM相同**）下，可以采用两种并行存储器结构来提高主存的带宽：

- 单体多字存储器 (就是一个存储器)
- 多体交叉存储器

7.6.1 单体多字存储器

一个单体m字（这里m=4）存储器



- 存储器能够每个存储周期读出m个CPU字。因此其最大带宽提高到原来的m倍。
  - $$B_M = m \times \frac{W}{T_M}$$
  - 单体多字存储器的实际带宽比最大带宽小
- 优缺点:
  - 优点:实现简单
  - 缺点:访存效率不高
    - 如果一次读取的m个指令字中有分支指令，而且分支成功，那么该分支指令之后的指令是无用的。
    - **一次取出的m个数据不一定都是有用的。**另一方面，当前执行指令所需要的多个操作数也不一定正好都存放在同一个长存储字中。
    - 写入有可能变得复杂。
    - 当要读出的数据字和要写入的数据字处于同一个长存储字内时，读和写的操作就无法在同一个存储周期内完成。

7.6.2 多体交叉存储器

多体交叉存储器：由多个单字存储体构成，每个体都有自己的地址寄存器以及地址译码和读/写驱动等电路

编址方法:

- 高位交叉编址（对存储单元矩阵按列优先编址）
- 低位交叉编址（对存储单元矩阵按行优先编址）

在计算机组成原理里面有详细介绍,体系结构并不过多关注于这个

7.6.3 避免存储体冲突

- 体冲突：两个请求要访问同一个存储体。
- 减少体冲突次数的一种方法：**采用多体**

7.7 虚拟存储器

- 虚拟存储器是“主存—辅存”层次进一步发展的结果
- 虚拟存储器可以分为两类：**页式和段式**
  - 页式虚拟存储器把空间划分为大小相同的块。(页面)
  - 段式虚拟存储器则把空间划分为可变长的块。(段)
  - 页面是对空间的机械划分，而段则往往是按程序的逻辑意义进行划分。
- Cache和虚拟存储器的参数取值范围
  -

参数	第一级Cache	虚拟存储器
块（页）大小	16-128字节	4096-65,536字节
命中时间	1-3个时钟周期	100-200个时钟周期
不命中开销	8-200个时钟周期	1,000,000-10,000,000个时钟周期
（访问时间）	（6-160个时钟周期）	（800,000-8,000,000个时钟周期）
（传输时间）	（2-40个时钟周期）	（200,000-2,000,000个时钟周期）
不命中率	0.1-10%	0.00001-0.001%
地址映象	25-45位物理地址到14-20位Cache地址	32-64位虚拟地址到25-45位物理地址

7.8 Cache优化技术总结 ☆☆

总结

- “+”号：表示改进了相应指标。
- “-”号：表示它使该指标变差。
- 空格栏：表示它对该指标无影响。
- 复杂性：0表示最容易，3表示最复杂。

优化技术	不命中率	不命中开销	命中时间	硬件复杂程度	说明
增加块大小	+	–		0	实现容易,Pentium 4的第二级Cache采用128字节的块
增加Cache容量	+			1	被广泛采用，特别是第二级Cache
提高相联度	+		-	1	被广泛采用
牺牲Cache	+		-	2	AMD Athlon采用了8个项的Victim Cache

优化技术	不命中率	不命中开销	命中时间	硬件复杂程度	说明
伪相联Cache	+			2	MIPS R10000的第二级Cache采用
硬件预取指令和数据	+			2-3	许多机器预取指令，UltraSPARC III预取数据
编译器控制的预取	+			3	需同时采用非阻塞Cache； 有几种微处理器提供了这种预取的支持
用编译技术减少Cache不命中次数	+			0	向软件提出了新要求； 有些机器提供了编译器选项
使读不命中优先于写		+	-	1	在单处理机上实现容易， 被广泛采用
写缓冲合并		+		1	与写直达合用，广泛应用， 例如21164，UltraSPARC III
尽早重启动和关键字优先		+		2	被广泛采用
非阻塞Cache		+		3	在支持乱序执行的CPU中使用
两级Cache		+		2	硬件代价大； 两级Cache的块大小不同时实现困难；被广泛采用
容量小且结构简单的Cache	-		+	0	容易实现,被广泛采用
对Cache进行索引时不必进行地址变换			+	2	对于小容量Cache来说实现容易， 已被Alpha21164和UltraSPARC III采用
流水化Cache访问			+	1	被
踪迹Cache			+	3	Pentium 4 采用

计算题

$$\text{平均访存时间} = \text{命中时间} + \text{不命中率} \times \text{不命中开销}$$

- **cache\主存系统的平均访问时间**
  - 若 $\mathbf{t}_{\mathrm{c}}$ 表示命中时的cache访问时间, $\mathrm{t_m}$ 表示未命中时的主存访问时间, $1-h$表示未命中率, 则cache/主存系统的平均访问时间 $\mathbf{t}_{\mathrm{a}}$:
  - $$t_a = ht_c + (1-h)t_m$$
- 访问效率
  - 访问效率 e:
  - $$e = \frac{t_c}{t_a}$$

例题:

CPU执行一段程序时, cache完成存取的次数为 1900 次, 主存完成存取的次数为 100 次

已知cache存取周期为 $50\ \mathrm{ns}$, 主存存取周期为 $250\ \mathrm{ns}$

求cache/主存系统的效率和平均访问时间

(1) 平均访问时间:

总次数: $1900+100=2000$

命中率 h: $\frac{1900}{2000}=95\%$

平均访问时间 $t_a = 0.95\times 50 + 0.05\times 250$

$= 47.5 + 12.5$

$= 600 = 60\ ns$

(2) 效率为 e :

$e = \frac{50}{60}\times 100\%$

$= 83.3\%$

8.1 I/O系统的性能

简述

输入/输出系统简称I/O系统

它包括：

- I/O设备
- I/O设备与处理机的连接

I/O系统是计算机系统中的一个重要组成部分

- 完成计算机与外界的信息交换
- 给计算机提供大容量的外部存储器
- I/O系统的性能对CPU的性能有很大影响

分类

按照主要完成的工作进行分类：

- 存储I/O系统 本章的主要内容
- 通信I/O系统

系统响应时间

系统的响应时间(衡量计算机系统的一个更好的指标)

从用户输入命令开始，到得到结果所花费的时间。

由两部分构成：

- I/O系统的响应时间
- CPU的处理时间

多进程技术只能够提高系统吞吐率，并不能够减少系统响应时间。

评价I/O系统性能

- 连接特性
  - （哪些I/O设备可以和计算机系统相连接）
- I/O系统的容量
  - （I/O系统可以容纳的I/O设备数）
- 响应时间和吞吐率等

另一种衡量I/O系统性能的方法：

考虑I/O操作对CPU的打扰情况
即考查某个进程在执行时，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。

8.2 I/O系统的可靠性、可用性和可信性

系统的可靠性 ☆☆

系统的可靠性:系统从某个初始参考点开始一直连续提供服务的能力

用**平均无故障时间MTTF**(mean time to failure)来衡量

系统中断服务的时间用**平均修复时间MTTR**(mean time to repair)来衡量。

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**。

系统的可用性

可用性: 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的比率

$$可用性 = \frac{MTTF}{MTTF + MTTR}$$

$MTTF + MTTR$：平均失效间隔时间$MTBF$

系统的可信性

系统的可信性：服务的质量。即在多大程度上可以合理地认为服务是可靠的。（**偏主观，不可以度量**）

例题:☆☆

例8.1 假设磁盘子系统的组成部件和它们的MTTF如下：

（1）磁盘子系统由10个磁盘构成，每个磁盘的MTTF为1 000 000小时；

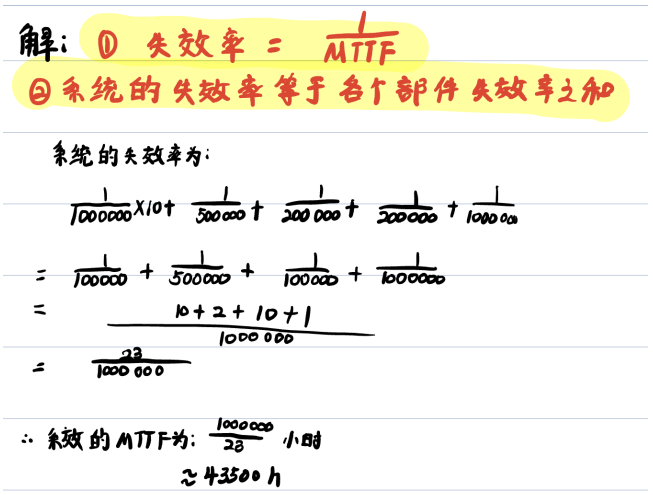
（2）1个SCSI控制器，其MTTF为500 000小时；

（3）1个不间断电源，其MTTF为200 000小时；

（4）1个风扇，其MTTF为200 000小时；

（5）1根SCSI连线，其MTTF为1 000 000小时。

假定每个部件的生存期服从指数分布，同时假定各部件的故障是相互独立的，求整个系统的MTTF。



提高系统组成部件可靠性的方法

- **有效构建方法**（valid construction）
  - 在构建系统的过程中消除故障隐患，这样建立起来的系统就不会出现故障。
- **纠错方法**（error correction）
  - 在系统构建中采用容错的方法。这样即使出现故障，也可以通过容错信息保证系统正常工作。
  - 为保证冗余信息在出现错误时不失效，通常将其存放在与出错部件不同的部件中，典型应用就是磁盘冗余阵列。

8.3 廉价磁盘冗余阵列RAID

简介

磁盘阵列DA（Disk Array）：使用多个磁盘（包括驱动器）的组合来代替一个大容量的磁盘。

- 多个磁盘并行工作。
- 以条带为单位把数据均匀地分布到多个磁盘上。
  - （交叉存放）
- 条带存放可以使多个数据读/写请求并行地被处理，从而提高总的I/O性能。
  - 多个独立的请求可以由多个盘来并行地处理
    - 减少了I/O请求的排队等待时间

- 如果一个请求访问多个块，就可以由多个盘合作来并行处理。
  - 提高了单个请求的数据传输率

阵列中磁盘**数量的增加**会导致**磁盘阵列可靠性**的下降。

- 如果使用了N个磁盘构成磁盘阵列，那么整个阵列的可靠性将降低为单个磁盘的1/N。由上面的系统失效率以及MTTF可以很容易推断出来
- 解决方法：**在磁盘阵列中设置冗余信息盘**
  - 当单个磁盘失效时，丢失的信息可以通过冗余盘中的信息重新构建。

廉价磁盘冗余阵列RAID:Redundant Arrays of Inexpensive Disks

- 独立磁盘冗余阵列

大多数磁盘阵列的组成可以由以下两个特征来区分

- **数据交叉存放的粒度**
  - 细粒度磁盘阵列是在概念上把数据分割成相对较小的单位交叉存放。
    - 优点：所有I/O请求都能够获得很高的数据传输率。
    - 缺点：在任何时间，都只有一个逻辑上的I/O在处理当中，而且所有的磁盘都会因为为每个请求进行定位而浪费时间。
  - 粗粒度磁盘阵列是把数据以相对较大的单位交叉存放。
    - 多个较小规模的请求可以同时得到处理。
    - 对于较大规模的请求又能获得较高的传输率

在磁盘阵列中设置冗余需要解决以下两个问题

- 冗余信息的计算
  - 大多都是采用**奇偶校验码**；
  - 也有采用海明码（Hamming code）或RS码的
- 如何将冗余信息分布到磁盘阵列中的各个盘里
  - 把冗余信息集中存放在少数的几个盘中。
  - 把冗余信息均匀地存放到所有的盘中。----（能避免出现热点问题）

RAID的分级及其特性

RAID级别	可以容忍的故障个数以及当数据盘为8个时，所需的检测盘的个数	优点	缺点	公司产品
0 非冗余，条带存放	0个故障；0个检测盘	没有空间开销	没有纠错能力	广泛应用
1 镜像	1个故障；8个检测盘	不需要计算奇偶校验， 数据恢复快,读数据快。 而且其小规模写操作比更高级别的RAID快	检测空间开销最大 （即需要的检测盘最多）	EMC，HP（Tandem），IBM
2 存储器式ECC	1个故障；4个检测盘	不依靠故障盘进行自诊断	检测空间开销的级别是log2m级（m为数据盘的个数）	没有
3 位交叉奇偶校验	1个故障；1个检测盘	检测空间开销小 （即需要的检测盘少）， 大规模读写， 操作的带宽高	对小规模、 随机的读写操作没有提供特别的支持	外存概念
4 块交叉奇偶校验	1个故障；1个检测盘	检测空间开销小， 小规模的读操作带宽更高	校验盘是小规模写的瓶颈	网络设备
5块交叉分布奇偶校验	1个故障；1个检测盘	检测空间开销小， 小规模的读写操作带宽更高	小规模写操作需要访问磁盘4次	广泛应用
6 P + Q双奇偶校验	2个故障；2个检测盘	具有容忍2个故障的能力	小规模写操作需要访问磁盘6次， 检测空间开销加倍（与RAID3、4、5比较）	网络设备

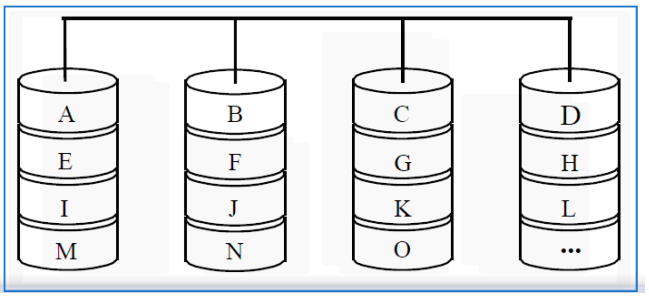
有关RAID的几个问题

- **关键问题：如何发现磁盘的故障**
  - 在磁盘扇区中除了保存数据信息外，还保存有用于发现该扇区错误的检测信息。
- 设计的另一个问题:如何减少平均修复时间MTTR
  - 典型的做法：在系统中增加热备份盘

- 热切换技术
  - 与热备份盘相关的一种技术

8.3.1 RAID 0

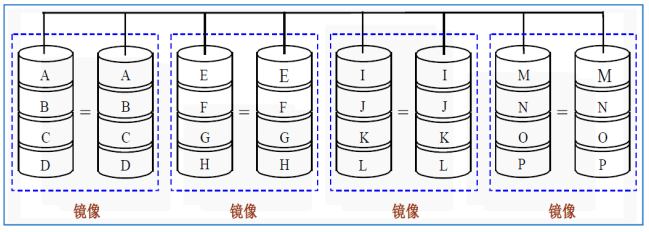
- **非冗余磁盘阵列，无冗余信息**。严格地说，它不属于RAID系列。
- RAID中最简单的一种，实现成本也最低
- **把数据切分成条带，以条带为单位交叉地分布存放到多个磁盘中。**
-



- RAID0不提供数据冗余，因此一旦数据被破坏，将无法得到恢复。
- 任何一块磁盘出现故障，整个系统将无法正常工作
  - （多个磁盘的系统可靠性比采用单个大容量磁盘的可靠性要低很多）
- 适用于需要**高带宽磁盘访问**的场合。

8.3.1 RAID 1

- **镜像磁盘**:对所有的磁盘数据提供一份冗余的备份。
- 每当把数据写入磁盘时，将该数据也写入其镜像盘。
- 在系统中**所有的数据都有两份**。
-



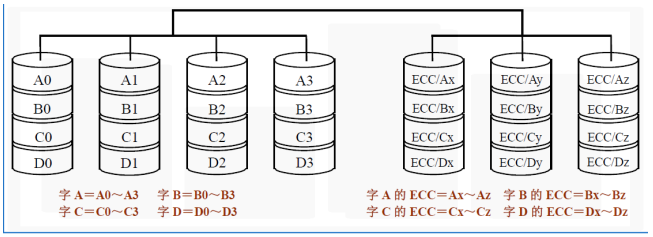
RAID 1的特点

- 读取数据时，**镜像的两个磁盘**可以独立地同时工作，能实现快速的读取操作。
- 对于写入操作，镜像的两个磁盘都要写入。但可并行进行，而且**不需要计算校验信息**，所以其速度比级别更高的RAID都快。
- 可靠性很高，数据的恢复很简单。
- 实现成本最高。

8.3.3 RAID 2

存储器式的磁盘阵列（**按海明纠错码的思路构建**）

含4个数据盘的RAID2

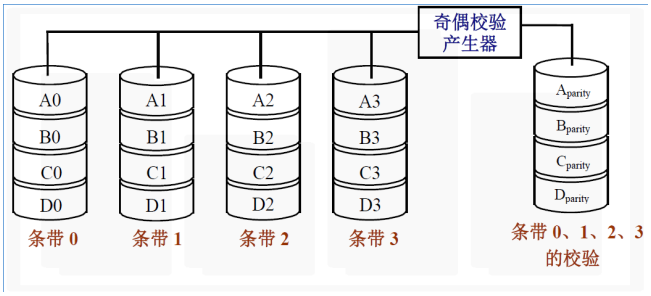


RAID 2 的特点

- 每个数据盘存放所有数据字的一位
  - （位交叉存放）
- 各个数据盘上的相应位计算海明校验码，编码位被存放在多个校验（ECC）磁盘的对应位上。
- 冗余盘是用来存放海明码的，其个数为**log2m级**。
  - m：数据盘的个数（也就是数据字的位数）
- 并未被广泛应用，目前还没有商业化产品。

8.3.4 RAID 3

位交叉奇偶校验磁盘阵列



特点:

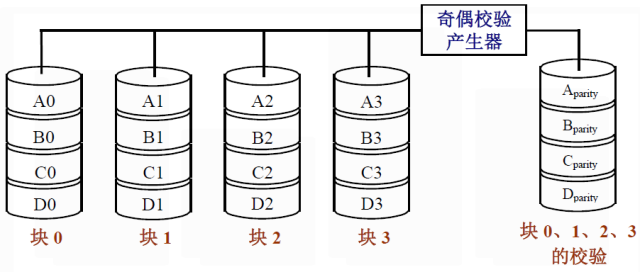
- **采用奇偶校验**
  - **写数据**时：为**每行数据形成奇偶校验位**并写入校验盘
  - **读出数据**时：如果控制器发现某个磁盘出故障，就可以根据故障盘以外的所有其他盘中的正确信息恢复故障盘中的数据。（通过**异或运算**实现）
- **细粒度的磁盘阵列，即采用的条带宽度较小。**
  - （可以是1个字节或1位）
  - 能够获得很高的数据传输率，这种磁盘阵列对大数据量的读写具有很大的优越性。
  - 不能同时进行多个I/O请求的处理。
- **只需要一个校验盘，校验空间开销比较小**

8.3.5 RAID 4

块交叉奇偶校验磁盘阵列

采用比较大的条带，**以块为单位**进行交叉存放和计算奇偶校验。

实现目标：**能同时处理多个小规模访问请求**



RAID4读写特点:

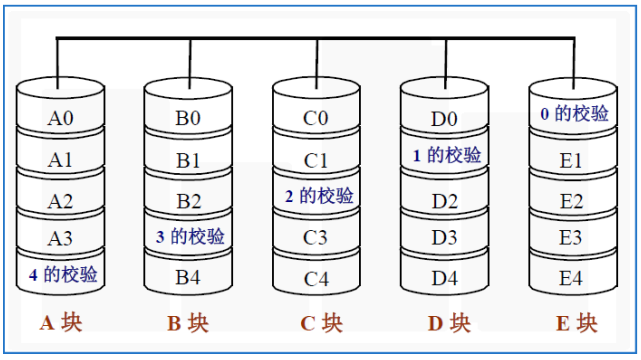
- 读取操作
  - 每次只需访问数据所在的磁盘。
  - 仅在该磁盘出现故障时，才会去读校验盘，并进行数据的重建。
- 写入操作
  - 原始方法：新数据写入，需要把所有的数据全部读出，重新计算校验值，即使没有修改其他硬盘的内容。
  - 假定:有4个数据盘和一个冗余盘。写数据需要2次磁盘读和2次磁盘写操作。

RAID4能有效地处理小规模访问，快速处理大规模访问，校验空间开销比较小。但其控制比较复杂。

8.3.6 RAID5

块交叉分布奇偶校验磁盘阵列

数据以块交叉的方式存于各盘，**无专用冗余盘**，奇偶校验信息**均匀分布在所有磁盘上**。



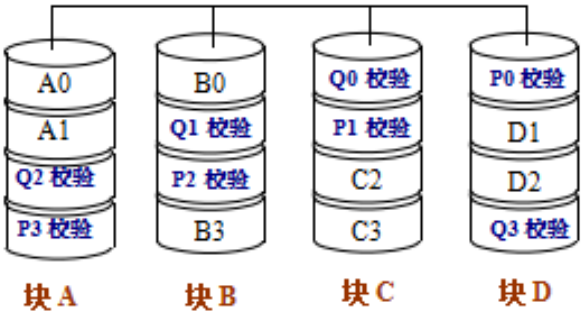
- 优点
  - 校验空间开销小
  - 快速处理大规模的访问
  - 快速处理小规模读和写的操作
- 缺点
  - 控制器最复杂

8.3.7 RAID6

P + Q双校验磁盘阵列

特点

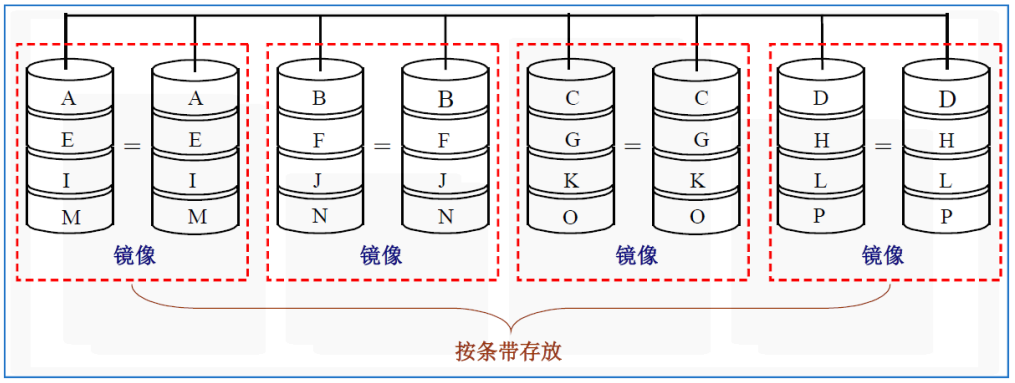
- 校验空间开销是RAID5的两倍
- 容忍两个磁盘出错
- 适合重要数据的保存



RAID10与RAID01

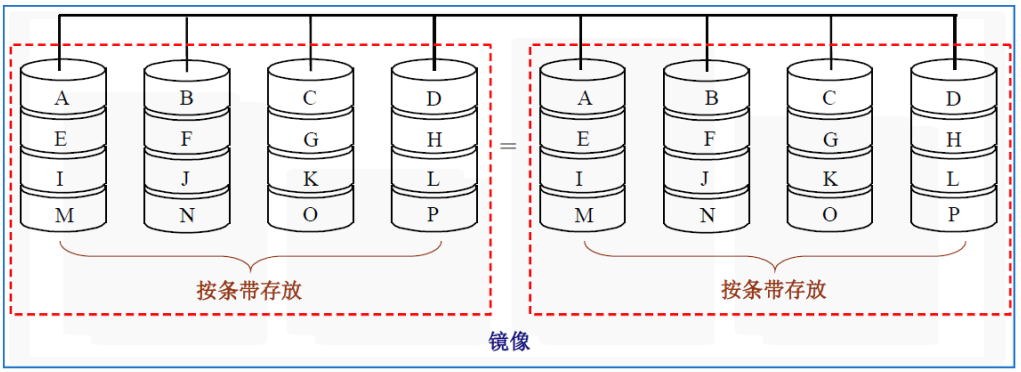
RAID10又称为RAID1+0

先进行镜像（RAID1），再进行条带存放（RAID0）



RAID01又称为RAID0+1

先进行条带存放（RAID0），再进行镜像（RAID1）



性能上，RAID 0+1比RAID 1+0有着更快的读写速度。可靠性上，当RAID 1+0有一个硬盘受损，其余三个硬盘会继续运作。RAID 0+1 只要有一个硬盘受损，同组RAID 0的硬盘亦会停止运作，可靠性较低。因此，RAID 10远较RAID 01常用，

零售主板绝大部份支持**RAID 0/1/5/10**，但不支持RAID 01。

8.3.9 RAID的实现与发展

实现盘阵列的方式主要有三种：

- **软件方式：阵列管理软件由主机来实现。**
  - 优点：成本低
  - 缺点：过多地占用主机时间，且带宽指标上不去。
- **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，一般用于工作站和PC机。
- **子系统方式**：一种基于通用接口总线的开放式平台，可用于各种主机平台和网络系统。

8.4 总线

总线:在计算机系统中，各子系统之间可以通过总线互相连接。

优点：成本低、简单
主要缺点：它是由不同的外设分时共享的，形成了信息交换的瓶颈，从而限制了系统中总的I/O吞吐量。

简介

构成计算机系统的互联机构，是多个系统功能部件之间进行数据传送的公共通路。

借助与总线连接，计算机在各系统功能部件之间实现**地址、数据和控制信息的交换**，并在征用资源的基础上进行工作。

一个单处理器系统中的总线，大致分为三类

- **内部总线**
  - CPU内部连接各寄存器及其运算部件之间的总线
- **系统总线**
  - CPU同计算机系统的其他高速功能部件，比如存储器，通道等互相连接的总线
- **I/O总线**
  - 中、低速I/O设备之间相互连接的总线

8.4.1 总线的设计

总线设计存在很多技术难点

一个重要原因：**总线上信息传送的速度极大地受限于各种物理因素。**

另外，我们一方面要求I/O操作响应快，另一方面又要求高吞吐量，这可能造成设计需求上的冲突。

特性	高性能	低价格
总线宽度	独立的地址和数据总线	数据和地址分时 共用同一套总线
数据总线宽度	越宽越快（例如：64位）	越窄越便宜（例如：8位）
传输块大小	块越大总线开销越小	单字传送更简单
总线主设备	多个（需要仲裁）	单个（无需仲裁）
分离事务	采用，分离的请求包和回答包能提高总线带宽	不采用，持续连接成本更低，而且延迟更小
定时方式	同步	异步

- 分离事务总线
  - （又称：流水总线、悬挂总线、包交换总线）
  - 在有多个主设备时，可以通过包交换来提高总线带宽。
  - 基本思想
    - 将总线事务分成请求和应答两部分
    - 在请求和应答之间的空闲时间内，总线可以供其他的I/O使用，这样就不必在整个I/O过程中都独占总线。
- 同步总线
  - **同步总线的控制线中包含一个时钟，总线上所有设备的所有的通信操作都以该时钟为基准**
  - 优点：速度快、成本低
  - 缺点:
    - 由于时钟通过长距离传输后会扭曲，因而同步总线不能用于长距离的连接。特别是对于高速同步总线来说，更是如此。
    - 总线上的所有设备都必须以同样的时钟频率工作。
  - CPU-存储器总线通常是采用同步总线。
- 异步总线
  - 没有统一的参考时钟，每个设备都有各自的定时方法。
  - 采用**握手协议**
  - 不存在时钟扭曲和同步的问题，传输距离可以比较长。
  - 很多I/O总线都采用异步总线。
  - **同步总线通常比异步总线快。**

8.4.2 总线标准和实例

I/O总线标准：定义如何将设备与计算机进行连接的文档。

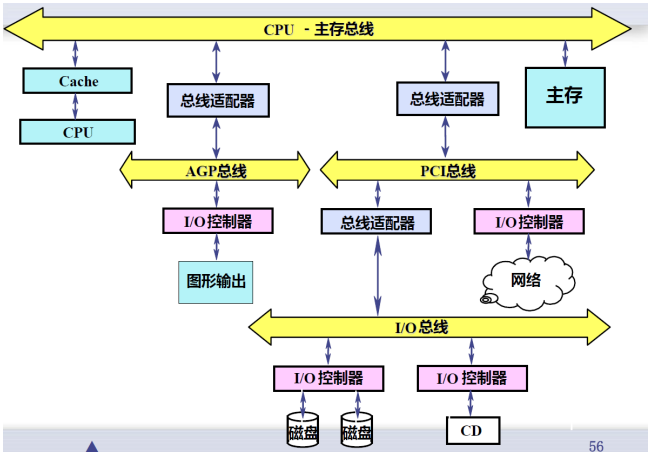
几种常用并行I/O总线

	IDE / Ultra ATA	SCSI	PCI	PCI-X
数据宽度（b）	16	8/16	32/64	32/64
时钟频率（MHz）	最高100	10（Fast） 20（Ultra） 40（Ultra2） 80（Ultra3） 160（Ultra4）	33/66	66/100/133
总线主设备数量	1个	多个	多个	多个
峰值带宽（MBps）	200	320	533	1066
同步方式	异步	异步	同步	同步
标准	无	ANSI X3.131	无	无

8.4.3 与CPU的连接

I/O总线的物理连接方式有两种选择

- 连接到存储器上
  - 更常见
- 连接到Cache上



CPU对I/O设备的编址有两种方式

- **存储器映射I/O**（也称为I/O设备统一编址）
  - 将一部分存储器地址空间分配给I/O设备，用load指令和store指令对这些地址进行读写将引起I/O设备的数据传输。
  - 将一部分存储空间留出用于设备控制，对这一部分地址空间进行读写就是向设备发出控制命令。
- **给I/O设备独立编址**
  - 需要在CPU中设置专用的I/O指令来访问I/O设备。
  - CPU需要发出一个标志信号来表示所访问的地址是I/O设备的地址

CPU与外部设备进行输入/输出的方式可分为4种

- **程序查询**
  - 程序查询方式是早期计算机中使用的一种方式。
  - 数据在CPU和外围设备之间的传送完全靠计算机程序控制
  - 优点是CPU的操作和外围设备的操作能够同步，而且硬件结构比较简单
  - 问题是，外围设备动作很慢，程序进入查询循环时将白白浪费掉CPU很多时间。CPU此时只能等待，不能处理其他业务
  - 当前，**除单片机外，很少使用程序查询方式**
- **中断**
  - 中断是外围设备用来“主动”通知CPU，准备送出输入数据或者接收输出数据的一种方式。
  - 通常，当发生中断时，CPU暂停现行程序，转向中断处理程序，从而可以输入或输出一个数据。当中断处理完毕后，CPU有返回到原来的任务，并从停止的地方开始执行程序。
  - 优点：节省CPU的时间，管理I/O的一个较为有效方法。
  - 适用于随机出现的服务，一旦提出要求，应立即响应。
  - 缺点：硬件结构相对复杂，服务开销较大。
- **DMA**
  - MA方式是一种**完全由硬件执行**的I/O交换工作方式。
  - DMA控制器从CPU完全接管对总线的控制，数据交换不经过CPU，而直接在内存和外围设备之间进行。
  - 优点：数据传送速度很高，传送速率仅受到内存访问时间的限制。
  - 缺点：需要更多的硬件。
  - **适用于内存和高速外设之间大批数据交换的场合。**
- **通道**
  - DMA方式的出现已经减轻了CPU对I/O操作的控制，使得CPU的效率有了显著提高，而通道的出现则进一步提高了CPU的效率
  - CPU将部分权利下放给通道。通道是一个具有特殊功能的处理器，某些应用中成为输入输出处理器（IOP），它可以实现对外设的统一管理和外设与内存之间的数据传送。
  - 优点：大大提高了CPU的工作效率
  - 以花费更多硬件为代价
- 外围处理机(PPU)方式
  - 外围处理机（Peripheral Processing Unit，PPU），是通道方式的进一步发展
  - 由于PPU基本上独立于主机工作，其结构更接近于一般意义上的处理机，甚至就是微小型计算机。
  - 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。
  - 从某种意义上来说，这种系统已经变成分布式的多机系统

8.5 I/O与操作系统

操作系统的作用之一是在多进程之间进行进程保护，这种保护包括存储器访问和I/O操作两个方面。

/O操作主要是在外设和存储器之间进行，所以操作系统必须保证这些I/O操作的安全性。

8.5.1 DMA和虚拟存储器

DMA使用虚拟地址还是物理地址两种方式进行数据传输

- 使用物理地址进行DMA传输，存在以下两个问题：
  - 问题
    - 对于超过一页的数据缓冲区，由于缓冲区使用的页面在物理存储器中不一定是连续的，所以传输可能会发生问题。
    - 如果DMA正在存储器和缓冲区之间传输数据时，操作系统从存储器中移出（或重定位）一些页面，那么，DMA将会在存储器中错误的物理页面上进行数据传输。
  - 解决这些问题的方法
    - 使操作系统在I/O的传输过程中确保DMA设备所访问的页面都位于物理存储器中，这些页面被称为是钉在了主存中。
    - **“虚拟DMA”技术**:
      - 允许DMA设备直接使用虚拟地址，并在DMA期间由硬件将虚拟地址转换为物理地址。
      - 在采用虚拟DMA的情况下，如果进程在内存中被移动，操作系统应该能够及时地修改相应的DMA地址表。

8.5.2 I/O和Cache数据一致性

Cache会使一个数据出现两个副本:一个在Cache中，另一个在主存中。

I/O设备可以修改存储器中的内容:

- **把I/O连接到存储器**上会出现以下情况：
  - 对于写回法，CPU修改了Cache的内容后，由于存储器的内容跟不上Cache内容的变化，I/O系统进行输出操作时所看到的数据是旧值。（写直达Cache没有这样的问题）
  - I/O系统进行输入操作后，存储器的内容发生了变化，但CPU在Cache中所看到的内容依然是旧值。
- **把I/O直接连接到Cache上**
  - 不会产生由I/O导致的数据不一致的问题
    - 所有I/O设备和CPU都能在Cache中看到最新的数据
  - I/O会跟CPU竞争访问Cache，在进行I/O时，会造成CPU的停顿
  - I/O还可能会破坏Cache中CPU访问的内容，因为I/O操作可能导致一些新数据被加入Cache，而这些新数据可能在近期内并不会被CPU访问
  - 许多计算机还是选择把I/O直接连接到存储器上，把存储器的一片区域作为I/O缓冲器，减少对CPU的打扰

解决内容一致性问题的方法:

- **软件的方法**
  - 设法保证I/O缓冲器中的所有各块都不在Cache中。
  - 具体做法有两种
    - 把I/O缓冲器的页面设置为不可进入Cache的，在进行输入操作时，操作系统总是把输入的数据放到该页面上。
    - 在进行输入操作之前，操作系统先把Cache中与I/O缓冲器相关的数据“赶出”Cache，即把相应的数据块设置为“无效”状态。
- **硬件的方法**
  - 在进行输入操作时，检查相应的I/O地址（I/O缓冲器中的单元）是否在Cache中（即是否有数据副本）。
  - 如果发现I/O地址在Cache中有匹配的项，就把相应的Cache块设置为“无效”。