期末复习

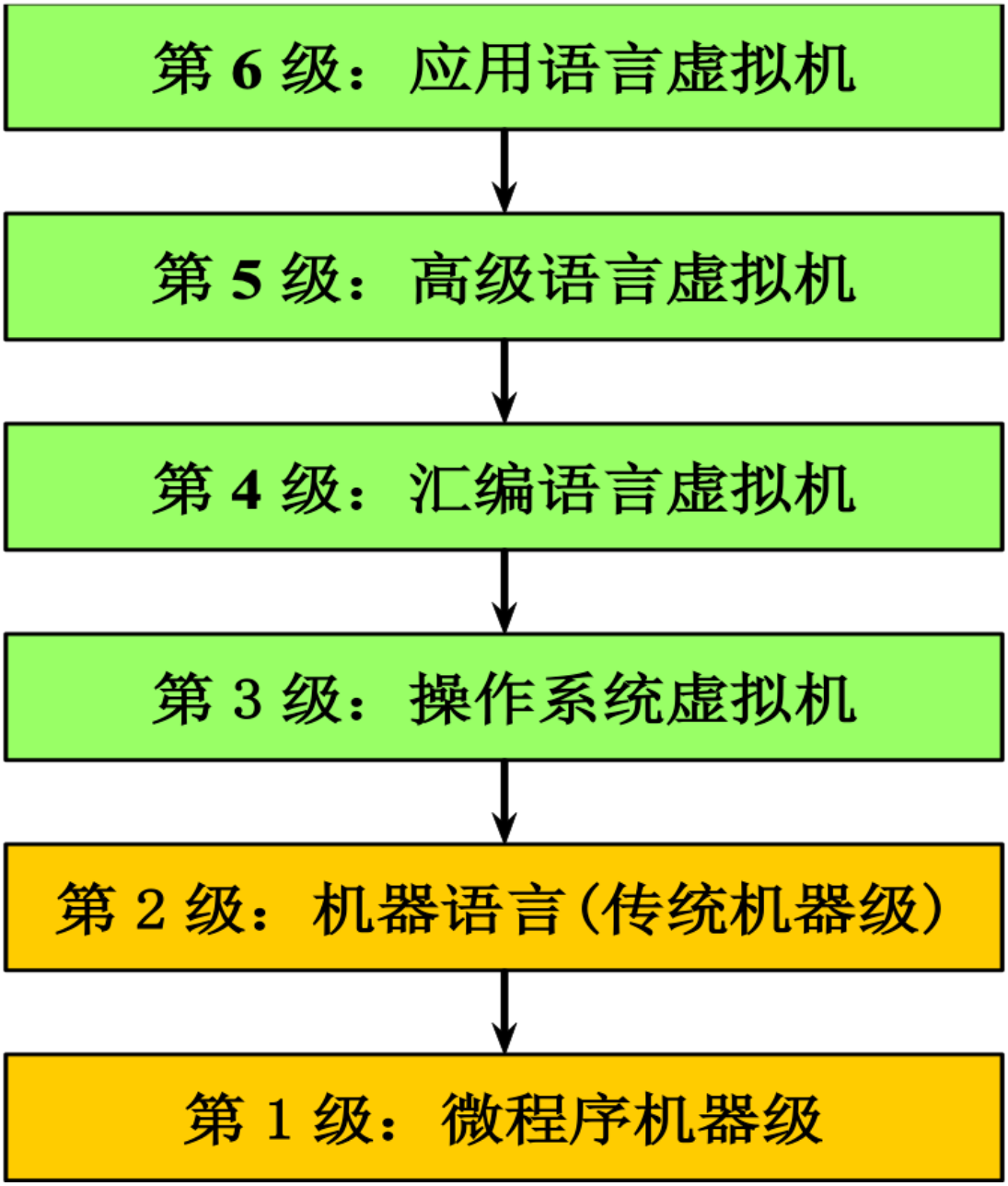
第一章计算机系统结构的基本概念

1.1 引言

- 系统结构的重大转折
  - 从单纯依靠**指令级并行**转向开发**线程级并行**和**数据级并行**

1.2 计算机系统的概念

- 计算机系统的层次结构
  - 计算机语言从低级向高级发展(**微 机 操 汇 高 应**)
    - 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。



- 第1级和第2级是硬件或固件
- 往上是软件

- L1微程序机器级
  - 机器语言是微指令集，直接由固件/硬件来实现
- L2传统机器级
  - 机器语言是传统的机器指令集，由L1的微程序进行解释执行，过程又可以称为仿真
  - 有的计算机没有采用微程序技术，指令集由硬连逻辑解释执行
- L3操作系统虚拟机
  - 传统机器级指令
  - 操作系统级指令
  - 虚拟机是指：由软件实现的机器
  - 语言实现的两种基本技术
    - 翻译
    - 解释
    - 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
- L4汇编语言虚拟机
- L5高级语言虚拟机

    - L6应用语言虚拟机
- 计算机系统结构的定义
    - 计算机系统结构的**经典定义**
        - 传统机器级程序员所看到的计算机属性，即**概念性结构**与**功能特性**
    - 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**
        - **透明性**：在计算机技术中，把这种本来存在的事物或属性、但从某种角度看又好像不存在的概念称为透明性
    - Amdahl提出的系统结构：**传统机器语言级程序员所看到的计算机属性**
    - **广义**的系统结构定义
        - **指令集结构**
        - **组成**
        - **硬件**
    - 对于通用寄存器型机器来说，这些属性主要是指
        - 指令系统
        - 数据表示
        - 寻址规则
        - 寄存器定义
        - 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
- 计算机系统结构概念的**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
- 计算机组成和计算机实现
    - 计算机系统结构：计算机系统的软、硬件的界面
        - 机器语言程序员所看到的传统机器级所具有的属性
    - 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**
    - 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
        - 着眼于：器件技术(起主导作用)、微组装技术
        - **一种体系结构可以有多种组成**
        - **一种组成可以有多种物理实现**
    - 系列机
        - 由**同一厂家**生产的具有相同体系结构、但具有**不同组成和实现**的一系列**不同型号**的计算机
        - 系列机的软件兼容有4种
            - 向上兼容
            - 向下兼容
            - 向前兼容
            - 向后兼容
        - **向后兼容**是肯定要做到的，是**系列机**的**根本特征**
- 计算机系统结构的分类
    - 常见的计算机系统结构分类法有两种
        - Flynn分类法
        - 冯氏分类法
    - 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
    - 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
            - 在系统受限的部件上，同时处于统一执行阶段的指令或数据的最大数目
    - Flynn分类法把计算机系统的结构分为4类
        - 单指令单数据流(SISD)
        - 多指令流多数据流(SIMD)
        - 多指令流单数据流(MISD)
        - 多指令流多数据流(MIMD)
    - 四类计算机的基本结构

1.3定量分析技术

计算机系统设计的定量原理(4个)(经 A C 程)

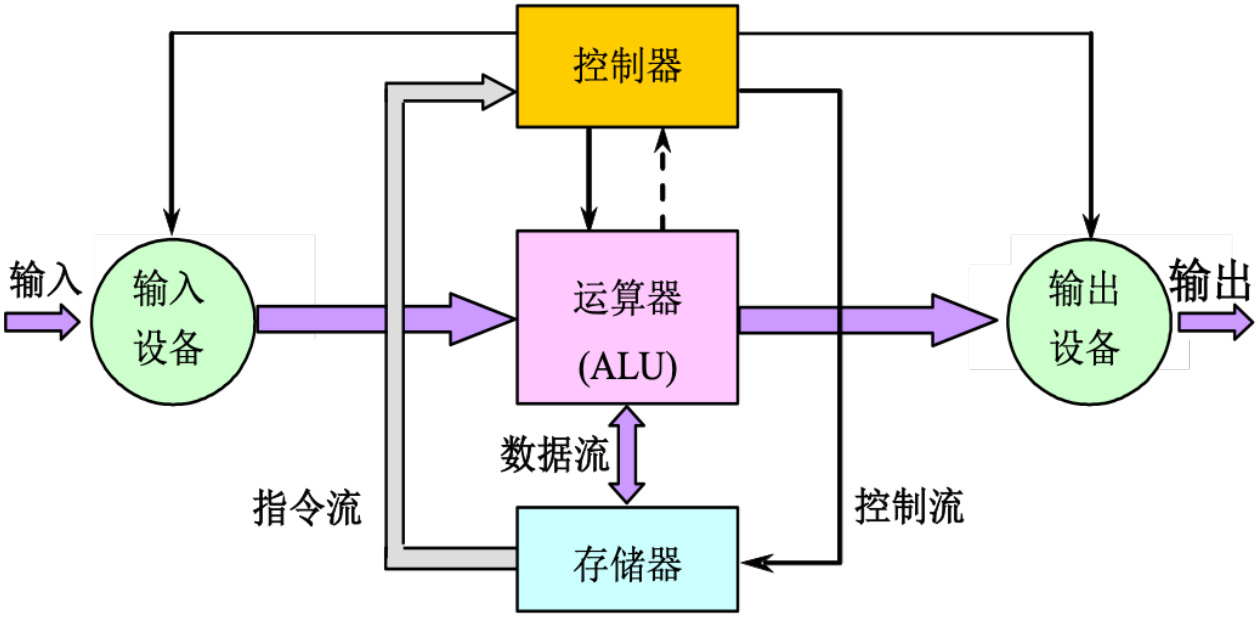
- 以经常性事件为重点
    - 计算机系统的设计中，对经常发生的情况，赋予它优先的处理权和资源使用权，以得到更多的总体上的改进
- Amdahl定律
- CPU性能公式
- 程序的局部性原理
    - 是指程序执行时所访问的存储器地址分布不是随机的，而是相对簇聚。数据访问也具有局部性，不过弱于代码访问的局部性
    - 包括时间局部性和空间局部性
    - 时间局部性
        - 程序即将用到的信息很可能就是目前正在使用的信息
    - 空间局部性
        - 程序即将用到的信息很可能与目前正在使用的信息在空间上相邻或者临近
    - 根据程序的局部性原理，可以根据程序最近的访问情况来比较准确地预测将要访问的指令和数据

计算机系统的性能评测

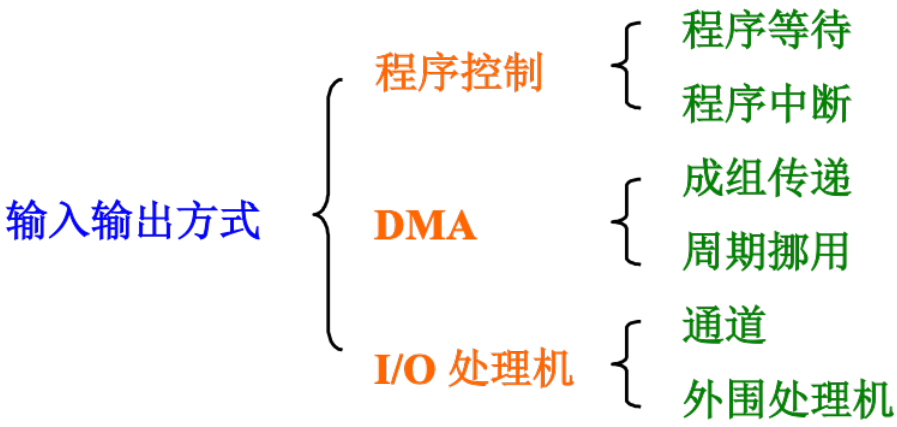
- 吞吐率
    - 单位时间内完成的任务数量
- CPU时间
    - CPU执行给定程序所花费的时间

1.4 计算机系统结构的发展

- 冯·诺依曼结构(又称为 **存储程序计算机的结构**)



- 存储程序原理的基本点：指令驱动
  - 程序预先存放在计算机存储器中
- 冯诺依曼结构的主要特点
  - 以运算器为中心
  - 在存储器中，指令和数据同等对待
  - 存储器是按地址访问、按顺序线性编址的一维结构，每个单元的位数是固定的
  - 指令的执行是顺序的
  - 指令由操作码和地址码组成
  - 指令和数据均以二进制编码表示，采用二进制运算
- 后来计算机对系统结构的改进(出 入 并 存 指)
  - 对输入输出方式的改进

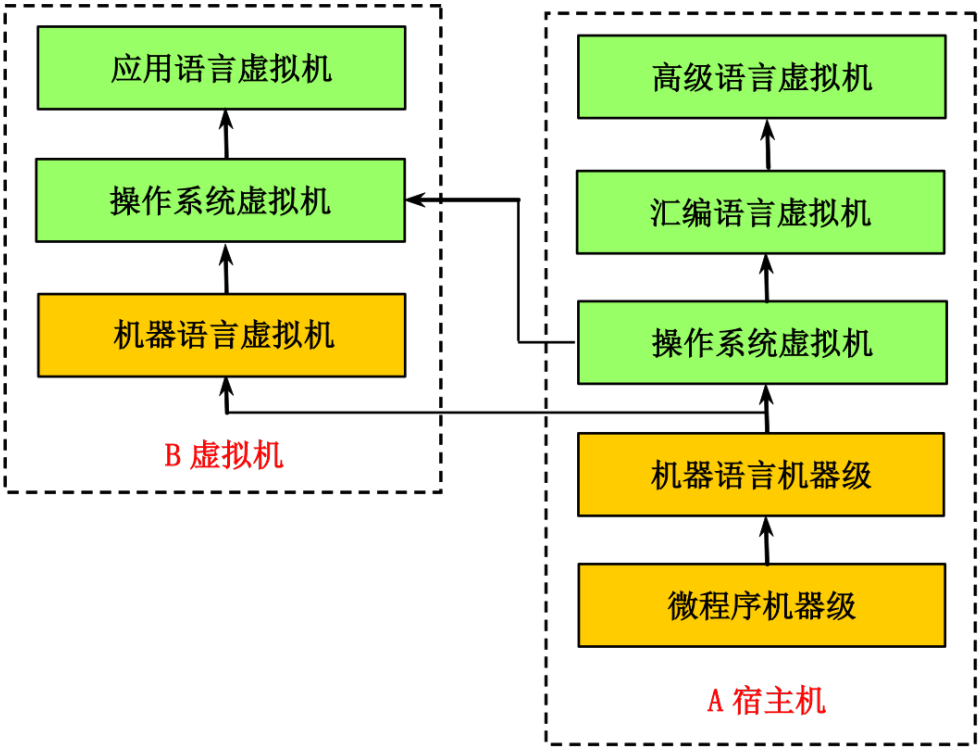


  - 采用并行处理技术
    - 在不同级别中采用并行技术

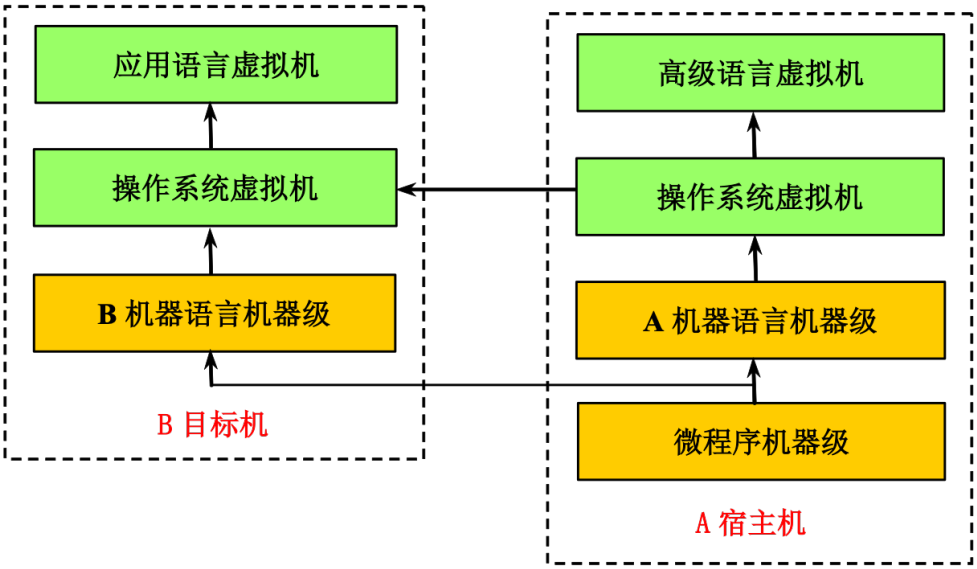
        - 微操作级
        - 指令级
        - 线程级
        - 进程级
        - 任务级
    - 存储器组织结构的发展
        - 相联存储器和相联处理机
        - 通用寄存器组
        - 高速缓冲存储器Cache
    - 指令集的发展
        - 两个发展方向
            - 复杂指令集计算机(CISC) complex
            - 精简指令集计算机(RISC) reduced

软件对系统结构的影响

- 软件的可移植性
    - 一个软件不经过修改或者只需要少量修改就可以由一台计算机移植到另一台计算机上运行，称两台计算机是软件兼容的。
- 实现可移植性的常用方法
    - 采用系列机
    - 模拟与仿真
    - 统一高级语言
- 系列机
    - 由同一厂家生产的具有相同的系统结构，但具有不同组成和实现的一系列不同型号的机器
        - 软件兼容
            - 向上(下)兼容
            - 向前(后)兼容
        - 向后兼容是系列机的根本特征
    - 兼容机
        - 由不同公司厂家生产的具有相同系统结构的计算机
- 模拟和仿真
    - 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
    - 模拟
        - 用软件的方法在一台现有的机器(称为宿主机)上实现另一台机器(称为虚拟机)的指令集
            - 通常用解释的方法来实现
            - 运行速度慢，性能差



- 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快
    - 只能在系统结构差距不大的机器之间使用



- 统一高级语言
  - 实现软件移植的一种理想方法
  - 较难实现

器件发展对系统结构的影响

- 摩尔定律
  - 集成电路芯片上所集成的晶体管数目每隔18个月就翻一番
- 计算机的分代主要以器件作为划分标准(电 晶 微 半 高)

    - 电子管
    - 晶体管
    - 微程序
    - 半导体存储器
    - 高性能微处理器

    - 在器件、系统结构和软件技术等方面都有各自的特征
        - SMP: 对称式共享存储器多处理机
        - MMP: 大规模并行处理机
- 应用对系统结构的影响
    - 应用需求是促使计算机系统结构发展的最根本动力

计算机系统结构中并行性的概念

- 并行性的概念
    - 并行性(计算机系统在同一时刻或同一时间间隔内进行的多种运算或操作)
        - 同时性(两个或两个以上的事件在同一时刻发生)
        - 并发性(两个或两个以上的事件在同一时间间隔内发生)
- 并行性等级
    - 处理数据的角度(从低到高)
        - 字串位串
        - 字串位并
        - 字并位串
        - 全并行
    - 从执行程序的角度(从低到高)
        - 指令内部并行
        - 指令级并行
        - 线程级并行
        - 任务级或过程级并行
        - 作业或程序级并行
- 单处理机系统
    - 并行性升到某一级别之后，并行性通过软件实现
- 多处理机系统
    - 并行性通过硬件实现
- 并行处理领域
    - 当并行性提高到一定级别时，则称之为进入并行处理领域。
    - 例如处理数据的并行性达到字并位串级，或者执行程序的并行级达到线程级、任务级或过程级，即可认为进入并行处理领域
- 提高并行性的技术途径
    - 时间重叠
        - 流水线技术是时间重叠的典型实例
    - 资源重复
    - 资源共享
- 单机系统中并行性的发展
    - 起主导作用的是时间重叠原理，实现时间重叠的基础是部件功能专用化
    - 资源重复原理的运用也已经十分普遍
        - 多体存储器
        - 多操作部件

        - 指令级并行
        - 阵列处理机
    - 单处理机的资源共享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
- 多机系统中并行性的发展
- 三种不同的多处理机
    - 异构型多处理机
    - 同构型多处理机
    - 分布式系统
    - 多机器系统的耦合度
        - 紧密耦合系统(直接耦合系统)
            - 通过总线或高速开关的互连，共享主存
        - 松散耦合系统
            - 通过通道或者通信线路实现计算机之间的互连，可以共享外存设备
            - 两种形式
                - 多台计算机和共享的外存设备连接
                - 计算机网络
    - 功能专用化(实现时间重叠)
        - 专用外围处理机
        - 专用处理机
        - 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
        - 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

第二章计算机指令集结构

指令集结构分类

- 区别不同指令集结构的主要因素：CPU中用来存储操作数的存储单元的类型
- CPU中用来存储操作数的存储单元的主要类型
    - 堆栈
    - 累加器
    - 通用寄存器组
- 指令集结构分为三类
    - 堆栈结构
    - 累加器结构
    - 通用寄存器结构(根据操作数来源不同，进一步分类)
        - 寄存器–存储器结构(RM结构)

        - 寄存器–寄存器结构(RR结构， load–store结构，只有load指令和store指令能够访问存储器)
- 不同类型的指令集结构，操作数的给出方式
    - 显式给出
    - 隐式给出
- 通用寄存器结构
    - 现代**指令集结构的主流**
    - 优势
        - 寄存器的访问速度比存储器快
        - 对编译器而言，能更加容易、有效地分配和使用寄存器
        - 寄存器可以用来存放变量
    - 根据ALU指令的操作数的两个特征对通用寄存器型指令集结构进一步细分
        - 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）
    - 3种通用寄存器型指令集结构的优缺点(**(m，n)表示指令的n个操作数中有m个存储器操作数**)

寻址方式

- 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
- **立即数寻址方式**和**偏移寻址方式**使用频率最高

指令集结构的功能设计

- **指令集结构的功能设计**就是确定**软、硬件功能分配**，即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
- 确定哪些基本功能用硬件来实现，考虑三个因素(**速 成 灵**)
    - 速度
    - 成本
    - 灵活性
- 对指令集的基本要求(**完 规 高 兼**)
    - 完整性
    - 规整性
        - 对称性
        - 均匀性
    - 高效率
        - 执行速度快
        - 使用频度高
    - 兼容性
- 设计指令集结构时，两种设计策略
    - CISC（复杂指令集计算机）
    - RISC（精简指令集计算机）
- CISC结构追求的目标
    - **强化指令功能**(目标程序优化 高级语言优化 操作系统优化)
        - 面向目标程序增强指令功能
        - 面向高级语言的优化实现来改进指令集
            - 增强对高级语言和编译器的支持
            - 高级语言计算机

            - 间接执行高级语言机器
            - 直接执行高级语言的机器
        - 面向操作系统的优化实现改进指令集
    - 减少程序的指令条数
    - 以达到提高性能的目的
- RISC指令集结构的功能设计
    - CISC指令集结构存在的问题
        - 各种指令的使用频度相差悬殊
        - 指令集庞大，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
    - 设计RISC机器遵循的原则
        - 指令条数少而简单
        - 采用简单而又统一的指令格式，并减少寻址方式
        - 指令的执行在单个机器周期内完成
        - 只有load和store指令才能访问存储器，其他指令的操作都是在寄存器之间进行
        - 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
- 控制指令
    - 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
    - 能够改变控制流的指令
        - 分支
        - 跳转
        - 过程调用
        - 过程返回
    - 改变控制流的大部分指令是分支指令（条件转移）。
    - 转移目标地址的表示
        - PC相对寻址
            - 关键：确定偏移量字段的长度
    - 过程调用和返回
        - 改变控制流
        - 保存机器状态
        - 保存返回地址

操作数的类型和大小

- 数据表示
    - 计算机硬件能够直接识别、指令集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
- 表示操作数类型的方法有两种
    - 指令中的操作码指定操作数的类型
    - 带标志符的数据表示。给数据加上标识，由数据本身给出操作数类型
        - 优点：简化指令集，可由硬件自动实现一致性检查和类型转换，缩小了机器语言与高级语言的语义差距，简化编译器等
        - 缺点：由于需要在执行过程中动态检测标志符，动态开销比较大，所以采用这种方案的机器很少见
- 操作数的大小：操作数的位数或字节数
    - 字节：8位
    - 半字：16位
    - 字：32位

    - 双字：64位

指令格式的设计

- 指令由两部分组成
    - **操作码**
    - **地址码**
- 指令格式设计
    - 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
- 操作码的编码
    - Huffman编码法
    - 固定长度的操作码
- 两种寻址方法的表示
    - 寻址方式编码于**操作码**中
    - 设置专门的**地址描述符**，由地址描述符表示相应操作数的寻址方式
- 指令集的3种编码格式(**优缺点**)
    - 可变长度编码格式
        - 寻址方式和操作数类型很多时最好
        - 各条指令的字长和执行时间差别很大
    - 固定长度编码格式
        - 寻址方式和操作数类型少时比较实用
        - 降低译码复杂度
    - 混合型编码格式
        - 减少目标代码的长度
        - 降低译码复杂度

第三章

流水线的基本概念

- 流水线技术
    - 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
    - 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他的子过程并行进行
- 流水线中的每个**子过程**及其**功能部件**称为**流水线的级或段**，段与段相互连接形成流水线。流水线的段数称为**流水线的深度**
- 指令流水线
    - 把指令的解释过程分解为**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
- 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4个子过程
- 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**
- 流水技术的特点
    - 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
        - 时间长的段将成为**流水线的瓶颈**

    - 流水线每一个功能部件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数据，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
- 流水线的分类
    - **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
    - 按照同一时间内**各段之间的连接方式**对多功能流水线做进一步的分类
        - 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
        - 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
    - **按照流水的级别来进行分类**
        - 部件级流水线（运算操作流水线）：把处理机的算术逻辑运算部件分段，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）：把指令的解释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
        - 处理机间流水线（宏流水线）：它是由两个或者两个以上的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
    - **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
    - **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）
    - 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处理机

流水线的性能指标

- **吞吐率**：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
    - 解决流水线瓶颈问题的常用方法
        - 细分瓶颈段
        - 重复设置瓶颈段
- **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
- **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

流水线设计中的若干问题

- 瓶颈问题

- 流水线的额外开销
  - 流水寄存器延迟
    - 建立时间
    - 传输延迟
  - 时钟偏移开销
- 冲突问题

流水线的相关与冲突

- 5段RISC流水线
  - 取指令周期(IF)
    - IR ← Mem[PC]
    - PC值加4(假设每条指令占4个字节)
  - 指令译码/读寄存器周期（ID）
    - 译码
    - 用IR中的寄存器编号去访问通用寄存器组，读出所需的操作数
  - 执行/有效地址计算周期（EX）
    - 不同指令进行的操作不同
      - 存储器访问指令：ALU把所指定的寄存器的内容与偏移量相加，形成用于访存的有效地址
      - 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取的数据进行运算
      - 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取的第一操作数和立即数进行运算
      - 分支指令：ALU把偏移量与PC值相加，形成转移目标的地址。同时，对前一个周期读出的操作数进行判断，确定分支是否成功。
  - 存储器访问/分支完成周期(MEM)
    - 该周期处理的指令只有load, store和分支指令
    - 其他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
  - 写回周期(WB)
    - 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系统
  - 总结
    - 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
    - 其他指令需要5个周期

- 采用流水线方式实现时，应解决以下几个问题
  - 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
  - 避免IF段的访存与MEM段的访存发生冲突
    - 分离的指令存储器和数据存储器
    - 分离的指令Cache和数据Cache
  - ID段和WB段都要访问同一寄存器文件

        - 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
    - 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
        - 设置一个专门的**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

- 相关与流水线冲突
    - 相关：两条指令之间存在某种依赖关系
        - 数据相关(也称真数据相关)
            - 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
        - 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
            - 名相关有两种
                - 反相关
                - 输出相关
            - **换名技术**：通过改变指令中操作数的名字来消除名相关
                - 对于寄存器操作数进行换名称称为**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
        - 控制相关
            - 控制相关是指由分支指令引起的相关
    - **流水线冲突**
        - 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
        - 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
                - 解决方法
                    - 插入**暂停周期**
                    - 设置独立的指令存储器和数据存储器
                    - 或设置独立的指令Cache和数据Cache
            - 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
                - 三种类型
                    - 写后读冲突(RAW),对应于真数据相关
                    - 写后写冲突(WAW)，对应于输出相关
                    - 读后写冲突(WAR),对应于反相关
                - **定向技术**减少数据冲突引起的停顿
                    - 关键思想：在某条指令产生计算结果之前，其他指令并不真正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他指令需要它的地方，那么就可以避免停顿
                - 需要停顿的数据冲突
                    - 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
                - 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，称为**指令调度**或者**流水线调度**。

- 控制冲突
  - 流水线遇到分支指令和其他会改变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
    - 不成功或者失败：PC的值保持正常递增，指向顺序的下一条指令
  - 处理分支指令最简单的方法
    - 冻结 或者 排空 流水线
    - 把由分支指令引起的延迟称为分支延迟
    - 可以采用两种措施来减少分支延迟
      - 在流水线中尽早判断分支转移是否成功
      - 尽早计算出分支目标地址
      - 这两步工作被提前到ID段完成，即分支指令是在ID段的末尾执行完成，所带来的分支延迟为一个时钟周期
    - 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
      - 预测分支失败
      - 预测分支成功
      - 延迟分支
        - 在延迟槽中放入有用的指令
        - 三种调度方法
          - 从前调度
          - 从目标处调度
          - 从失败处调度
        - 进一步改进
          - 分支取消机制

流水线的实现

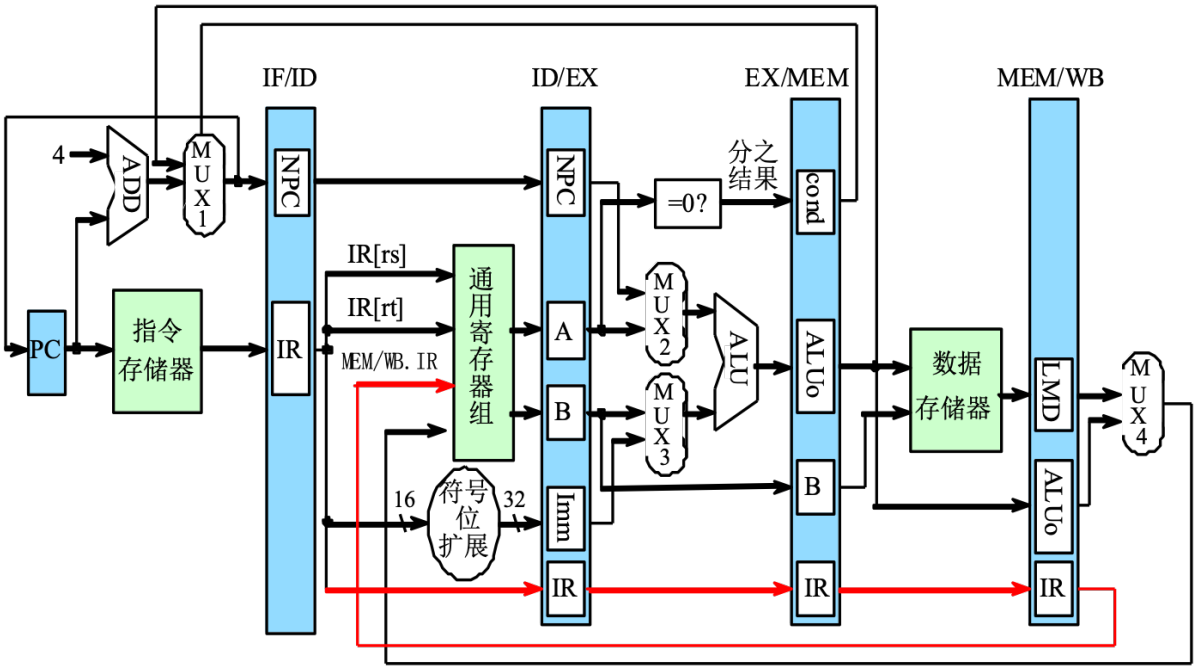
- 5个时钟周期的详细过程需要看书
- 取指令周期(IF)
  - IR <– Mem[PC]
  - NPC <– PC + 4
- 指令译码/读寄存器周期（ID）
  - $A \leftarrow Regs[rs]$
  - $B \leftarrow Regs[rt]$
  - $Imm \leftarrow ((IR_{16})^{16} \#\# IR_{16..31})$
  - 指令的译码操作和读寄存器操作是并行进行的
    - 因为在MIPS指令格式中，操作码字段以及rs、rt字段都是在固定的位置，这种技术称为固定字段译码技术
- 执行/有效地址计算周期(EX)

- 不同指令所进行的操作不同
  - 存储器访问指令：ALUo ← A + Imm
  - 寄存器－寄存器ALU指令： ALUo <– A func B
  - 寄存器－立即值ALU指令: ALUo <– A op Imm
  - 分支指令
    - ALUo←NPC+（Imm<<2）
    - cond←（A = = 0）
- 将有效地址计算周期和执行周期合并为一个时钟周期，这是因为MIPS指令集采用load/store结构，没有任何指令需要同时进行数据有效地址的计算、转移目标地址的计算和对数据进行运算。

- 存储器访问/分支完成周期（MEM）
  - 所有指令都要在该周期对PC进行更新。除了分支指令，其他指令都是做PC <– NPC
  - 在该周期的指令仅有load、store和分支指令
    - 存储器访问指令：LMD←Mem[ALUo]，或者Mem[ALUo] <– B
    - 分支指令：if （cond）PC ←ALUo else PC←NPC
- 写回周期(WB)
  - 不同的指令在写回周期完成的工作也不一样
    - 寄存器－寄存器ALU指令
      - Regs[rd]←ALUo
    - 寄存器－立即数ALU指令
      - Regs[rt]←ALUo
    - load指令
      - Regs[rt]←LMD
- 不采用单周期实现方案的主要原因
  - 对于大多数CPU来说，单周期实现效率很低，因为不同的指令所需完成的操作差别相当大，因而所需要的时钟周期时间也大不一样
  - 单周期实现时，需要重复设置某些功能部件，而在多周期实现方案中，这些部件是可以共享的

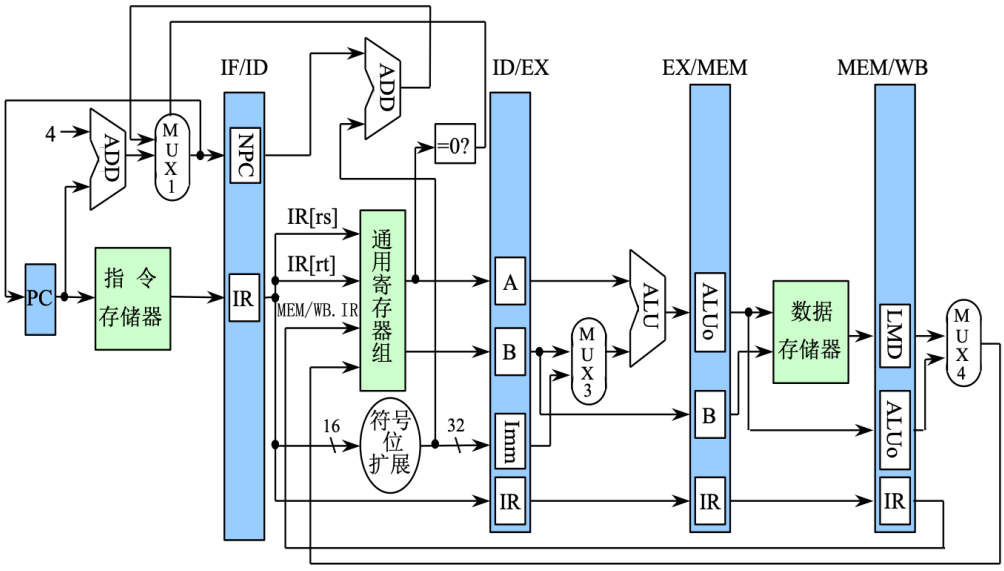
- 基本的MIPS流水线

  每一个时钟周期完成的工作看作是流水线的一段，每个时钟周期启动一条新的指令



  - 流水实现的数据通路

        - 段与段之间设置了流水寄存器
            - 流水寄存器的作用
                - 将各段的工作隔开，使得它们不会互相干扰
                - 保存相应段的处理结果
                - 向后传递将要用到的数据或控制信息
        - 增加了向后传递IR和从MEM/WB.IR回送到通用寄存器组的连接
        - 将对PC的修改移到了IF段，以便PC能及时地加4，为取下一条指令做好准备。
- 流水线的控制(主要是控制4个多路选择器)
    - MUX2：若ID/EX.IR中的指令是分支指令，则选择ID/EX.NPC，否则选ID/EX.A
    - MUX3：若ID/EX.IR中的指令是寄存器－寄存器型ALU指令，则选ID/EX.B，否则选ID/EX.Imm
    - MUX1：若EX/MEM.IR中的指令是分支指令，而且EX/MEM.cond为真，则选EX/MEM.ALUo，即分支目标地址，否则选PC+4
    - MUX4：若MEM/WB.IR中的指令是load指令，则选MEM/WB.LMD，否则选MEM/WB.ALUo
    - 第5个多路器：从MEM/WB回传至通用寄存器组的写入地址应该是从MEM/WB.IR[rd]和MEM/WB.IR[rt]中选一个
        - 寄存器－寄存器型ALU指令：选择MEM/WB.IR[rd]
        - 寄存器－立即数型ALU指令和load指令：选择MEM/WB.IR[rt]
    - 解决数据冲突的问题
    - 所有的数据冲突均可以在ID段检测到。如果存在数据冲突，就在相应的指令流出ID段之前将之暂停。完成该工作的硬件称为流水线的互锁机制
        - 在ID段确定需要什么样的定向，并设置相应的控制
        - 也可以在使用操作数哪个时钟周期的开始检测冲突和确定必需的定向
        - 检测冲突是通过比较寄存器地址是否相等来实现的
        - 举例：load互锁
            - 由于使用load的结果而引起的流水线互锁称为load互锁
            - 详情看书
    - 控制冲突
        - 分支指令的条件测试和分支目标地址的计算在EX段完成，对PC的修改在MEM段完成
        - 他所带来的分支延迟是3个时钟周期
        - 减少分支延迟(把上述工作提前到ID段进行)
            - 在ID段增设一个加法器，用于计算分支目标地址
            - 把条件测试“=0？”的逻辑电路移到ID段
            - 这些结果直接回送到IF段的MUX1
            - 改进后的流水线对分支指令的处理



第四章 指令级并行

指令集并行

- 指令集并行：几乎所有的处理机都利用流水线来使指令重叠并行执行，以达到提高性能的目的。这种指令之间存在的潜在并行性称为**指令级并行（ILP）**
  - 开发的方法主要有两大类
    - 基于**硬件**的**动态开发**
    - 基于**软件**的**静态开发方法**
- 流水线处理机的实际CPI

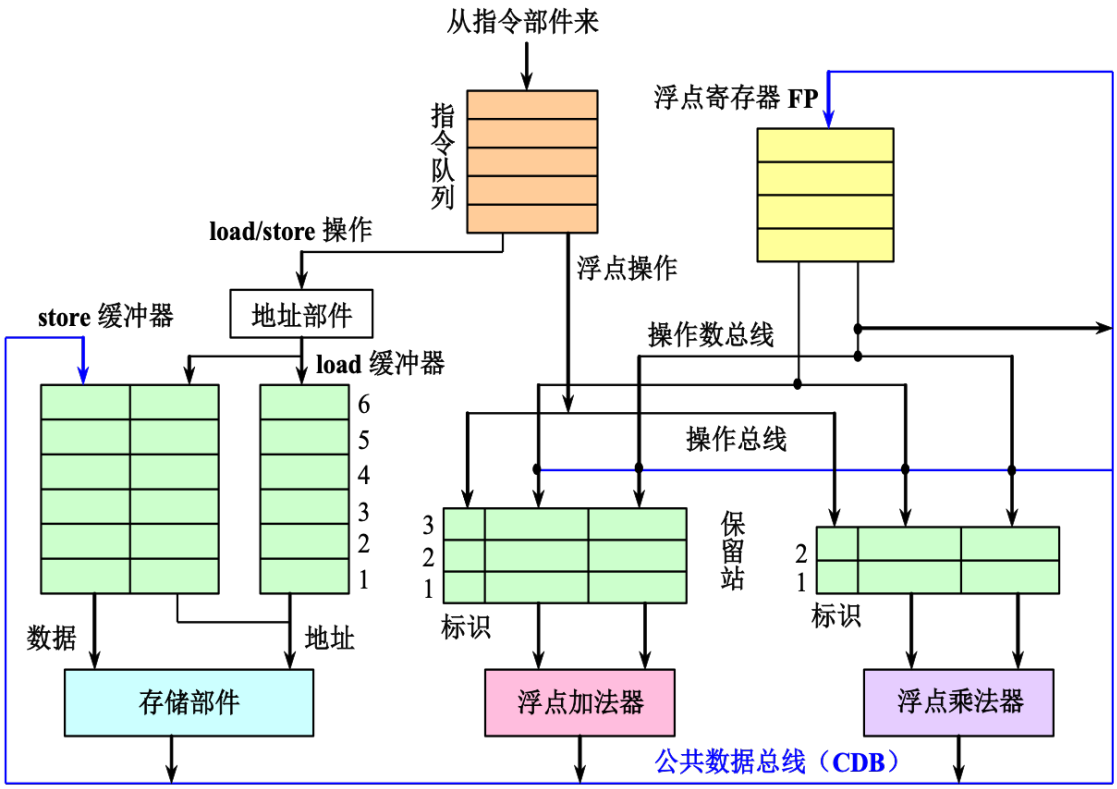
$$CPI_{流水线} = CPI_{理想} + 停顿_{结构冲突} + 停顿_{数据冲突} + 停顿_{控制冲突}$$

  - IPC：Instructions Per Cycle
    - 每个时钟周期完成的指令条数
- **基本程序块**：一段除了入口和出口以外不包含其他分支的线性代码段
- 循环级并行性：使**一个循环**中的**不同循环体**并行执行
  - 最常见、最基本
  - 指令级并行研究的重点之一
- 最基本的开发循环级并行的技术
  - 循环展开技术
  - 采用**向量指令**和**向量数据表示**
- 相关与流水线冲突
  - 相关
    - 数据相关
    - 名相关
    - 控制相关
  - 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**不能再指定的时钟周期**执行)
    - 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
- 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
    - 通过代码变换，消除相关
- 程序顺序：由源程序确定的在完全串行方式下指令的执行顺序
    - 只有在可能会导致错误的情况下，才保持程序顺序
- 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
        - 即原来程序中是怎么发生的，改变执行顺序后还是怎么发生
        - 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
    - 如果我们能做到保持程序的数据相关和控制相关，就能保持程序的数据流和异常行为
    - 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

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
    - 是在编译期间进行过代码调度和优化
    - 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他的流水线（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
- 动态调度的基本思想
    - 为了允许乱序执行，我们将5段流水线的译码阶段再分为两个阶段
        - 流出：指令译码，检查是否存在结构冲突
        - 读操作数：等待数据冲突消失，然后读操作数
    - 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
    - Tomasulo算法可以通过使用寄存器重命名来消除
    - 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、或者流水功能部件、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
            - 不精确异常
            - 精确异常
- Tomasulo算法
    - 核心思想
        - 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
        - 通过寄存器重命名来消除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
- 公共数据总线CDB
- load缓冲器和store缓冲器
- 浮点寄存器FP
- 指令队列
- 运算部件

- 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
- Tomasulo算法具有以下两个特点
  - 冲突检测和指令执行控制是分布的
  - 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
  - 流出
  - 执行
  - 写结果
- 每个保留站有以下几个字段
  - Op：要对源操作数进行的操作
  - Qj，Qk：将产生源操作数的保留站号
    - 等于0表示操作数已经就绪且在Vj或Vk中，或者不需要操作数
  - Vj，Vk：源操作数的值
    - 对于每一个操作数来说，V或Q字段只有一个有效
    - 对于load来说，Vk字段用于保存偏移量
  - Busy:为“yes”表示本保留站或缓冲单元“忙”
  - A: 仅load和store缓冲器有该字段。开始是存放指令中的立即数字段，地址计算后存放有效地址
- 寄存器状态表
  - Qi
    - 每个寄存器在该表中有对应的一项，用于存放将把结果写入该寄存器的保留站的站号
    - 为0表示当前没有正在执行的指令要写入该寄存器，也即该寄存器中的内容就绪
- Tomasulo算法具有两个主要的优点
  - 冲突检测逻辑是分布的(通过保留站和CDB实现)
  - 消除了WAW冲突和WAR冲突导致的停顿

        - 使用保留站进行寄存器换名，并且操作数一旦就绪就将之放入保留站
    - Tomasulo算法具体过程

动态分支预测技术

- 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
- 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
- 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
- 分支历史表BHT
    - 基本思想：用BHT来记录分支指令最近一次或几次的执行情况（成功或不成功），并据此进行预测
    - 两个步骤
        - 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
    - 优点和缺点
        - 只有以下情况才有用
            - 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
            - 前述5段经典流水线：由于判定分支是否成功和计算分支目标地址都是在ID段完成，所以BHT方法不会给该流水线带来好处
        - BHT可以放在指令Cache中，也可以用一个专门的硬件来实现
- 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）
    - BTB的另一种形式
        - 在分支目标缓冲器中存放一条或者多条分支目标处的指令
            - 三个潜在的好处
                - 更快地获得分支目标处的指令
                - 可以一次提供分支目标处的多条指令，这对于多流出处理器是很有必要的
                - 使我们可以进行称为分支折叠（branch folding）的优化
                - 实现零延迟无条件分支，甚至有时还可以做到零延迟条件分支
- 基于硬件的前瞻执行
    - 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，而是放到一个称为ROB（ReOrder Buffer）的缓冲器中。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

- 结合了三种思想
  - 动态分支预测
  - 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
- ROB由4个字段组成
  - 指令类型
  - 目标地址
  - 数据值字段
  - 就绪字段
- Tomasulo算法中保留站的换名功能是由ROB来完成的
- 优点
  - 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
- 缺点
  - 所需硬件太复杂

多指令流出技术

- 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是不固定的，依代码的具体情况而定
    - 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
    - 每个时钟周期流出的指令条数是固定的，构成一条长指令或者指令包
    - 指令包中，指令之间的并行性是通过指令显示地表示出来的
    - 指令调度是由编译器静态完成的
  - 超流水线处理机
    - 在一个时钟周期内能够分时流出多条指令的处理机称为超流水线处理机

循环展开和指令调度

- 所谓循环展开，是指把循环体的代码复制多次并按顺序排放，然后相应调整循环的结束条件。

第五章

存储系统的基本知识

- 存储系统的层次结构
  - 三个指标
    - 容量大
    - 速度快

        - 价格低
    - 解决方法：采用多种存储器技术，构成多级存储层次结构
        - 程序访问的局部性原理：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
        - 包含两个方面
            - 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
- 三级存储系统
    - 三级存储系统
        - Cache(高速缓冲存储器)
        - 主存储器
        - 磁盘存储器
    - Cache–主存层次和主存–辅存层次（目 实 速 大 访 切）

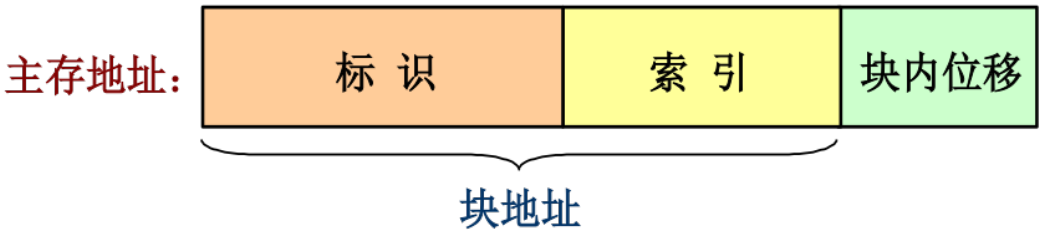
比较项目 \ 存储层次	“Cache －主存”层次	“主存－辅存”层次
目　　的	为了弥补主存速度的不足	为了弥补主存容量的不足
存储管理实现	主要由专用硬件实现	主要由软件实现
访问速度的比值(第一级和第二级)	几比一	几万比一
典型的块(页)大小	几十个字节	几百到几千个字节
CPU对第二级的访问方式	可直接访问	均通过第一级
不命中时CPU是否切换	不切换	切换到其他进程

- 存储层次的四个问题
    - 映像规则
    - 查找算法
    - 替换算法
    - 写策略

Cache基本知识

- 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Cache中的位置
        - 块内位移：确定所访问的数据在该块中的位置
- 映像规则
    - 全相联映像
        - 全相连：主存中的任一块可以被放置到Cache中的任意一个位置
        - 特点：空间利用率高，冲突概率最低，实现最复杂
    - 直接映像
        - 直接映像：主存中的每一块只能被放置到Cache中唯一的一个位置
        - 特点：空间利用率低，冲突概率最高，实现最简单
    - 组相联映像

        - 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
- 查找算法
    - 通过查找目录表来实现
        - Cache中设有一个目录表，每一个Cache块在该表都有唯一的一项。目录表存放的只是标识



    - 并行查找与顺序查找
        - 并行查找实现方法
            - 相联存储器
                - 目录由2g个相联存储区构成，每个相联存储区的大小为n×（h+log2n）位
            - 单体多字的按地址访问的存储器＋比较器
                - 优点
                    - 不必采用相联存储器，而是用按地址访问的存储器来实现
- 替换算法
    - 随机法
    - 先进先出法
    - 最近最少使用法(LRU)
        - 实际上选择最久没有被访问过的块
        - LRU算法的硬件实现
            - 堆栈法
            - 比较对法
- 写策略
    - 写直达法：执行"写"操作时，不仅把数据写入cache中相应的块，而且也写入下一级存储器
        - 采用写直达法时，"写"操作过程中 CPU必须等待，称为CPU写停顿。减少写停顿的方法是采用写缓冲器。
    - 写回法：只把数据写入cache相应的块，不写入下一级存储器，只有在相应的块被替换时，才写入下一级存储器
    - "写"操作时的调块
        - 按写分配
            - 写不命中时，先把所写单元所在的块调入cache,再行写入
        - 不按写分配
            - 写不命中是，直接写入下一级存储器而不是调块
- 写策略与调块
    - 写回法 –> 按写分配
    - 写直达法 –> 不按写分配

改进Cache性能

- 降低不命中率(大 容 相 伪 预， 预 优 牺)
    - 三种类型不命中
        - 强制性不命中

    - 容量不命中
    - 冲突不命中
- 1 增加Cache块大小
    - 现象：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
        - 原因
            - 增加了空间局部性，减少了强制性不命中
            - 减少了Cache中块的数目，增加了冲突不命中
            - 增加不命中开销
        - Cache容量越大，使不命中率达到最低的块大小就越大
- 2 增加Cache容量
    - 减少容量不命中
    - 缺点
        - 增加成本
        - 增加命中开销
    - 在片外cache中用的比较多
- 3 提高相联度
    - 减少冲突不命中
    - 增加命中时间
    - 采用相联度超过8的方案的实际意义不大
    - 2：1经验规则：容量为N的直接映像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
- 4 伪相联cache
    - 优点
        - 命中时间少
        - 不命中率低
    - 思想
        - 首先按与直接映像相同的访问方式进行访问，如果命中，就从相应的块中取出数据送给CPU，访问结束
        - 如果是不命中，伪相联Cache会检查另一个块，看是否匹配。确定另一个块的位置的方法是将索引的最高位取反，然后按照新的索引去寻找"伪相联组"中的对应块。如果这一块的标识匹配，则称发生了伪命中，否则，只能访问下一级存储器。

$$平均访存时间_{伪相联} = 命中时间_{1路} + （失效率_{1路} - 失效率_{2路}） \times 伪命中的额外开销 + 失效率_{2路} \times 失效开销_{1路}$$

- 5 硬件预取
    - 思想
        - 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
        - 发生不命中时取两个块
            - 被请求的指令块和顺序的下一指令块
            - 被请求的指令块放入Cache,预取的下一指令块放在缓冲器
    - 缺点
        - 有可能会降低正常的不命中时间
- 6 编译器控制的预取
    - 思想
        - 编译器在程序中加入预取指令来实现预取，这些指令在数据被用到之前就将它们取到寄存器或者Cache中
    - 分为两种(预取之后数据所放的位置)

        - 寄存器预取
        - Cache预取
    - 分为两种(预取的处理方式)
        - 故障性预取：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：预取时，如出现虚地址故障或违反保护权限，不发生异常，而是放弃预取，转变为空操作
    - 非阻塞Cache
        - Cache在等待预取数据返回时，还能提供数据和指令
  - 7 编译优化
    - 通过对软件进行优化来降低不命中率
        - 程序代码和数据重组
        - 编译优化技术包括
            - 数组合并
            - 内外循环交换
            - 循环融合
            - 分块
  - 8 牺牲Cache(减少冲突不命中)
    - 基本思想
        - 在Cache和其下一级存储器的数据通路上增设一个全相联的小Cache，用于存放Cache因冲突被替换出去的块。

- 减少不命中开销(两 读 合 请 塞)
  - 1 采用两级Cache
    - 在原有Cache和存储器之间再增设一级Cache
    - 第一级Cache做的小而快
    - 第二级Cache做的足够大
    - 第二级Cache捕获更多本来需要到主存去的访问，从而降低实际不命中开销
  - 2 让读不命中优先于写
    - 在写直达Cache中，设置一个大小适中的写缓冲器，在读不命中时检查写缓冲器的内容，如果没有冲突而存储器可以访问，就可继续处理读不命中。即让读不命中优先于写
    - 写回法中页可以利用写缓冲器来提高性能
  - 3 写缓冲合并
    - 写直达Cache中依靠写缓冲器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
        - 如果写缓冲器中已经有了待写入的数据，就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位置又是空闲的，就把这次要写入的数据与该项合并。
    - 优点：提高了写缓冲器的空间利用率，减少因写缓冲器满而要进行的等待时间
  - 4 请求字处理技术
    - 当从存储器向CPU调入一块时，块中往往只有一个字是CPU立即需要的，这个字称为请求字。
    - 思想
        - 当CPU所请求的字到达后，不等整个块调入Cache，就可把该字发送给CPU并重启CPU继续执行。
            - 尽早重启动
            - 请求字优先
  - 5 非阻塞Cache技术
    - 思想：Cache不命中时仍允许CPU进行其他的命中访问。即允许“不命中下命中”
    - 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
    - 缺点：大大增加了Cache控制器的复杂度
- 减少命中时间(小 虚 流 踪)
  - 1 容量小、结构简单的Cache

        - 思想：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
        - 直接映像的主要优点是：可以让标识检测和数据传送同时进行，从而有效减少命中时间
    - 2 虚拟Cache
        - 可以直接使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
    - 3 Cache访问流水化
        - 对第一级Cache的访问按流水方式组织
        - 访问Cache需要多个时钟周期才可以完成
    - 4 踪迹Cache
        - 踪迹Cache中存放的是CPU所执行过的动态指令序列，其中包含了由分支预测展开了的指令。该分支预测是否正确需要在取到该分支指令时进行确认。

记忆：

大 容 相 伪 预，预 优 牺

两 读 合 请 塞

小 虚 流 踪

并行主存系统

- 并行主存系统
    - 主存的性能衡量
        - 延迟
        - 带宽
    - 概念：在一个访存周期内能并行访问多个存储字的存储器
        - 有效提高存储器的带宽
    - 两种并行存储器结构
        - 单体多字存储器
            - 每个存储周期能读出m个CPU字
        - 多体交叉存储器
            - 由多个单字存储体构成
                - 两种编址方法
                    - 高位交叉编址
                    - 低位交叉编址

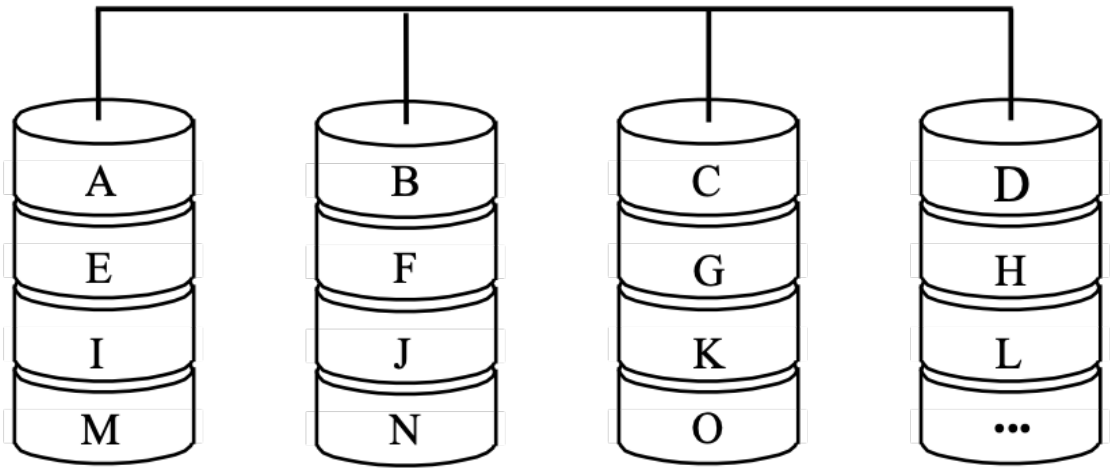
第六章 输入输出系统

- 输入/输出系统简称 I/O系统，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
- 按照主要完成的工作分类
    - 存储I/O系统
    - 通信I/O系统
- 系统的响应时间
    - 从用户输入命令开始，到得到结果所花费的时间

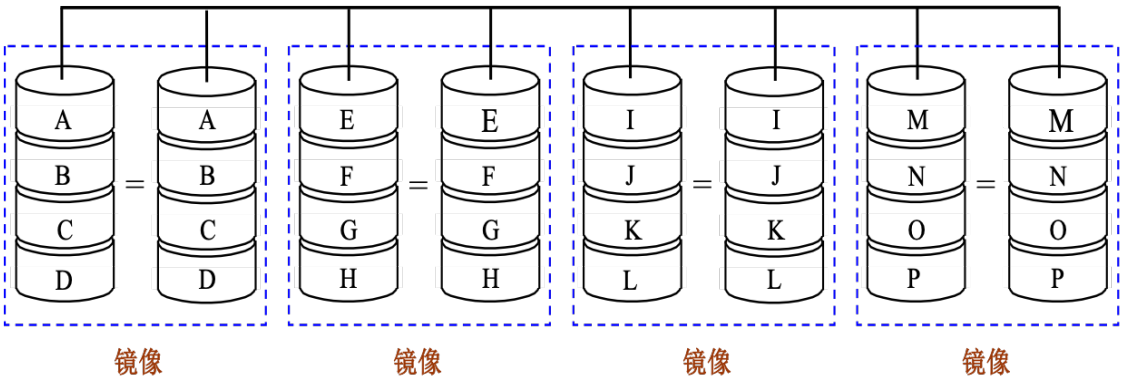
    - 两部分构成
        - I/O系统的响应时间
        - CPU的处理时间
- 评价I/O系统性能的参数主要有：
    - **连接特性**（哪些I/O设备可以和计算机系统相连接）
    - **I/O系统的容量**（I/O系统可以容纳的I/O设备数）
    - **响应时间和吞吐率**
- 另一种衡量I/O系统性能的方法：
    - 考虑I/O操作对CPU的打扰情况，即考查某个进程在执行时，由于其他进程的I/O操 作，使得该进程的执行时间增加了多少。
- 反映输入/输出系统可靠性的参数
    - 可靠性
        - 系统从某个**初始参考点**开始一直**连续 提供服务**的能力
        - 用平均无故障时间MTTF来衡量。（Mean Time To Failure）
    - 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 = MTTF/(MTTF + MTTR)
        - MTTF+MTTR：平均失效间隔时间MTBF （Mean Time Between Failure）
    - 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
- 廉价磁盘冗余阵列RAID
    - 磁盘阵列
        - 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
    - 这里并行性有两方面的含义
        - 多个独立的请求可以由多个盘来并行地处理
        - 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
    - 阵列中磁盘数量的增加会导致磁盘阵列可靠性 的下降
    - 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，丢失的信息可以利用冗余盘中的信息重新构建。只有在这个失效磁盘被恢复之前，又发生了第二个磁盘的失效时，磁盘阵列才不能正常工作。
    - 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
            - 粗粒度磁盘阵列
        - 冗余数据的计算方法以及在磁盘阵列中的存放方式

- RAID的分级及其特性

- RAID0(非冗余磁盘阵列)

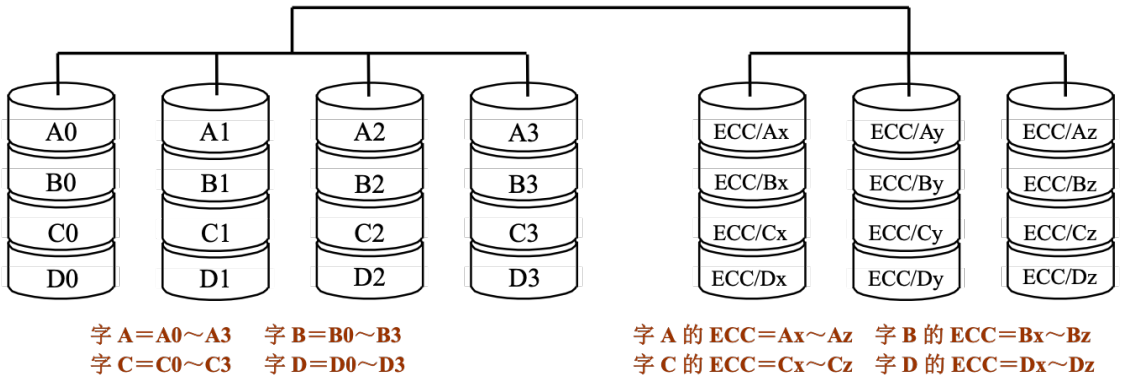


  - 非冗余阵列，无冗余信息
  - 把数据切分成条带，以条带为单位交叉地分布存放到多个磁盘中
- RAID1(镜像磁盘)

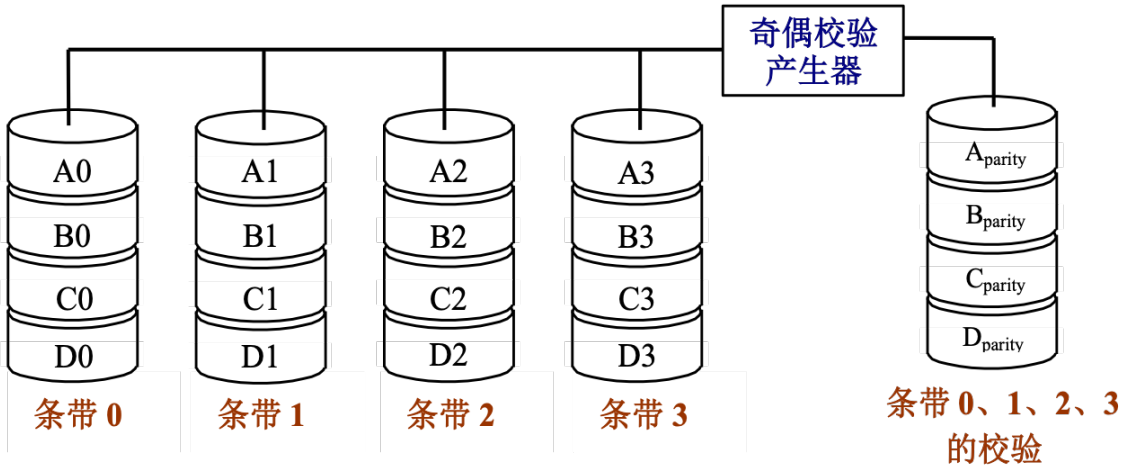


  - 核心思想
    - 对所有磁盘数据提供一份冗余的备份
  - 写操作
    - 每当把数据写入磁盘时，都要将该数据也写入其镜像盘。
  - 读操作
    - 当从该磁盘阵列读取数据时，磁盘及其镜像可独立同时工作，由最先读出数据的磁盘提供数据。所以RAID1能够实现快速的读取操作。
  - 纠错
    - 数据的恢复也很简单，由其镜像盘提供数据，系统仍能继续工作
  - 能实现快速的读取操作
  - 写性能由写性能最差的磁盘决定。相对以后各级 RAID来说，RAID1的写速度较快
  - 可靠性很高，数据的恢复很简单
  - 最昂贵的解决方法，物理磁盘空间是逻辑磁盘空间的两倍

- RAID2(存储器式的磁盘阵列)



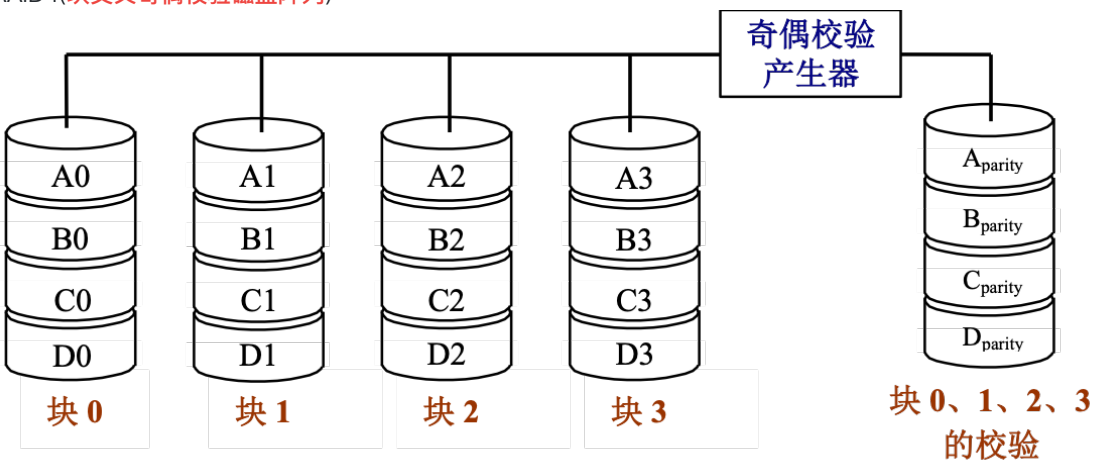
  - 因为按汉明纠错码的思路构建，所以有这样的名称
  - 核心思想
    - 每个数据盘存放所有数据字的一位（位交叉存放）
    - 各个数据盘上的相应位计算汉明校验码，编码位 被存放在多个校验（ECC）磁盘的对应位上
    - 冗余盘是用来存放汉明码的，其个数为log 2 m级，汉明码具有纠正1位错误和检测两位错误的能力。
  - 写数据
    - 每当往数据盘写入数据时，就为之形成Hamming码。
  - 读数据
    - 每当从数据盘读出数据时，就把其Hamming码也读出来，用于判断数据是否有错。Hamming码具有纠正1位错误和检测两位错误的能力。如果出现了1位错误，则可以立即加以纠正。
  - 缺点
    - 当数据盘较多时，系统的规模会变得很大。磁盘大多在自己内部已有校验码
- RAID3(位交叉奇偶校验盘阵列)



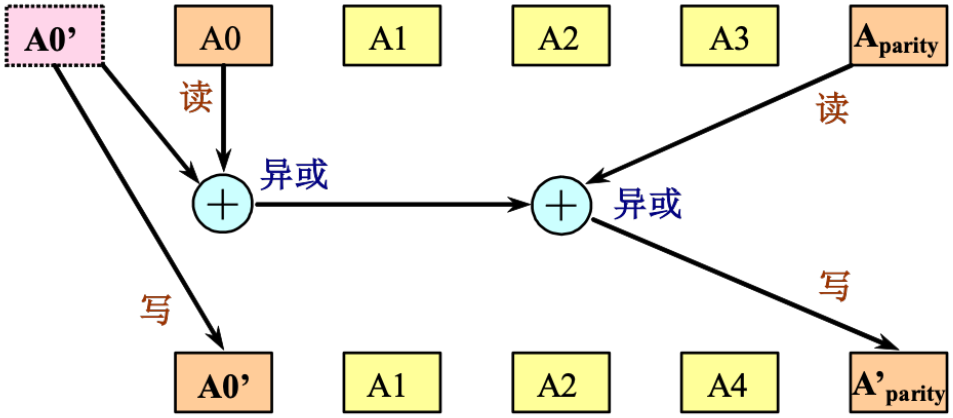
  - 核心思想
    - 采用奇偶校验，每个数据盘存放所有数据字的一位（位交叉存放），
  - 写数据
    - 在数据写入磁盘时，为每行数据形成奇偶校验并写入校验盘。
  - 读数据
    - 在读出数据时，如果控制器发现某个磁盘出故障，就可以根据故障盘以外的其他盘中的正确信息恢复故障盘中的数据。(通过异或运算实现)
  - 优点
    - 是一种细粒度的磁盘阵列，对于绝大多数的I/O请求，都需要磁盘阵列中的所有磁盘为之服务，因而能够获得很高的数据传输率，对于大数据量的读写具有很大优越性。
    - 只需要一个校验盘，校验空间开销比较小
  - 缺点

- 不能同时进行多个I/O请求的处理，对多个小规模的I/O请求来说表现较差

- RAID4(块交叉奇偶校验磁盘阵列)

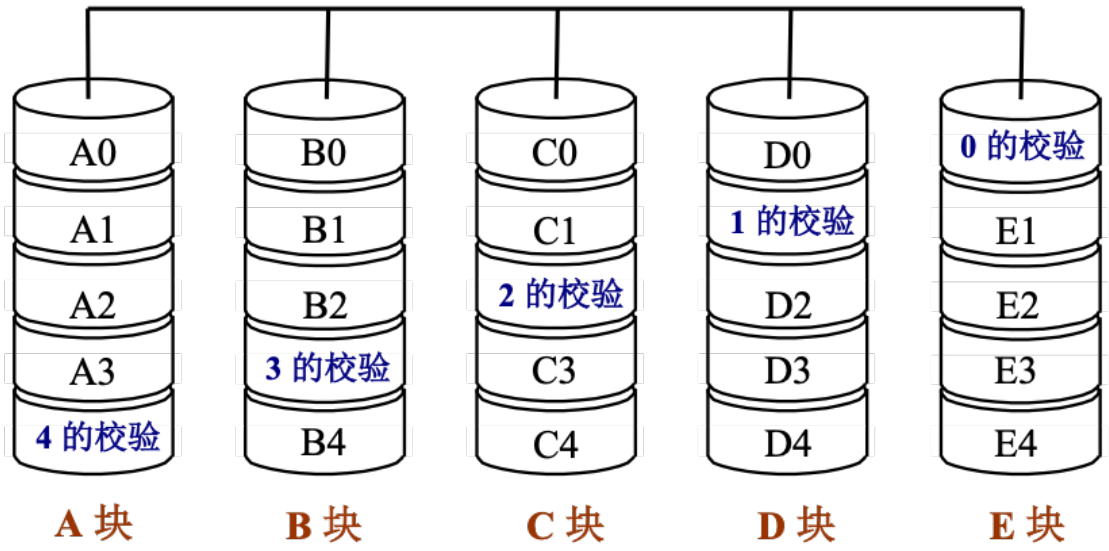


  - 核心思想
    - 采用比较大的条带，以块为单位进行交叉存放和计算奇偶校验。
  - 实现目标
    - 能同时处理多个小规模访问请求
  - 读取操作
    - 每次只需访问数据所在的磁盘
    - 仅在该磁盘出现故障时，才会去读校验盘，并进行数据的重建
  - 写入操作
    - 要重新计算校验码，要进行两次读和两次写。

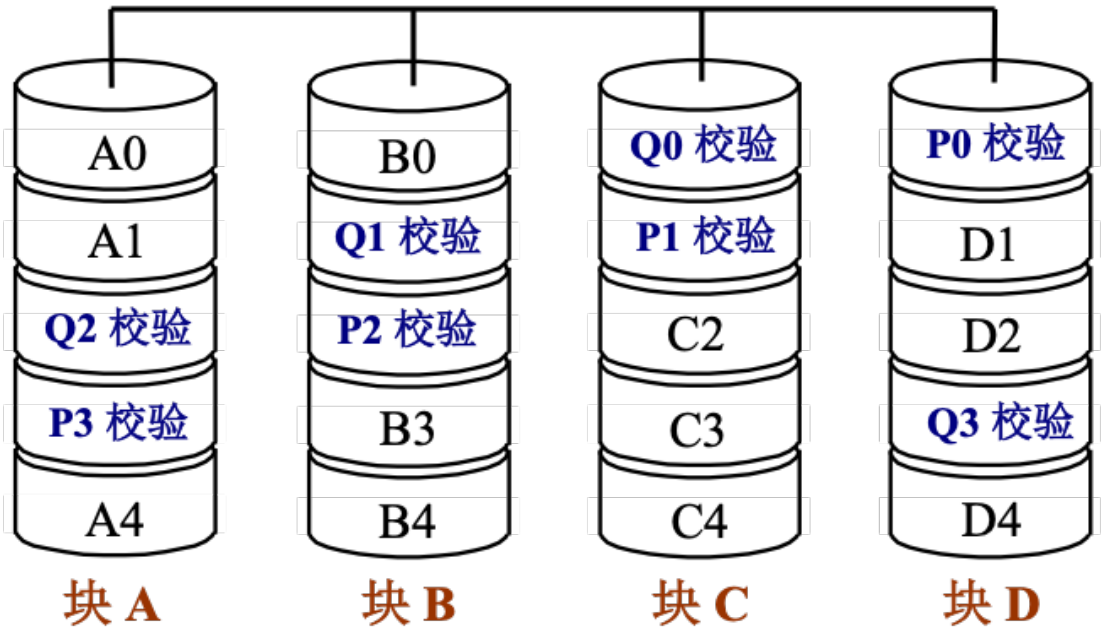


  - 优点
    - 除了能有效地处理小规模访问，也能跟RAID3一样快速处理大规模访问，校验空间也比较小
  - 缺点
    - 所有的写入操作都必须读和写校验盘，当同时处理多个小规模写访问时，它们必须都访问校验盘，而系统中校验盘只有一个，很容易称为瓶颈。

- RAID5(块交叉分布奇偶校验磁盘阵列)

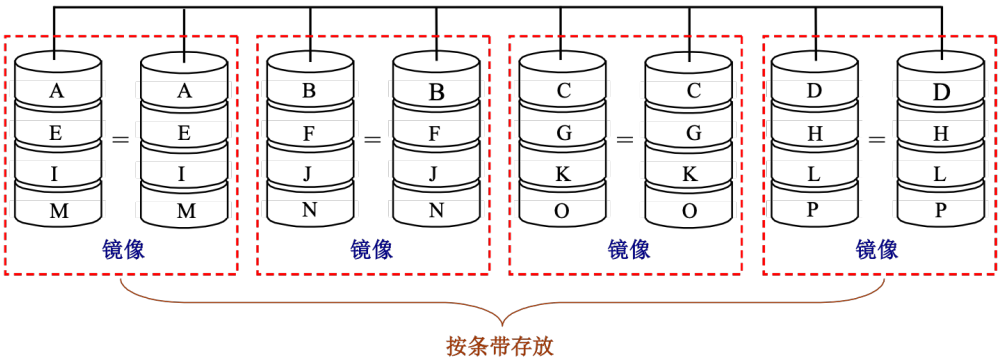


  - 核心思想
    - 数据以块交叉的方式存于各盘，无专用冗余盘，奇偶校验信息均匀分布在所有磁盘上
  - 优点
    - 校验空间较小
    - 和RAID3一样块地处理大规模访问
    - 跟RAID4一样快地处理小规模读操作
    - 还能比它们都更快地处理小规模写操作
  - 缺点
    - 控制器是(RAID1~RAID5)中最复杂的
- RAID6(P+Q双校验磁盘阵列)

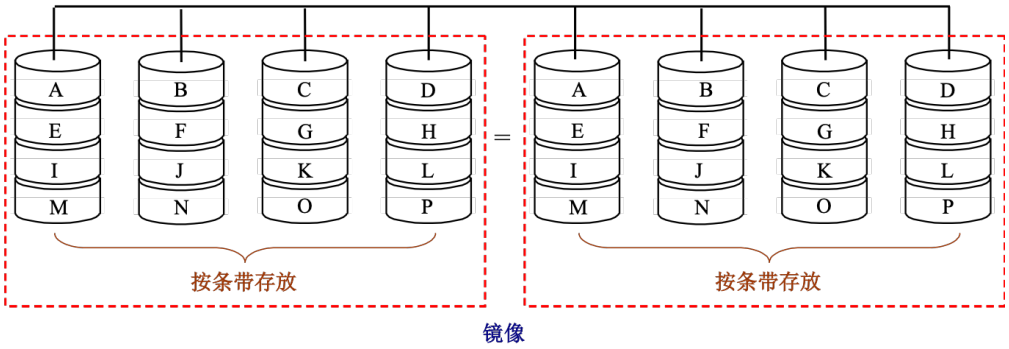


  - 核心思想
    - 在RAID5的基础上增加了一个独立的校验信息，放在另一个校验盘中
  - 特点
    - 能够容忍两个磁盘出错

        - 校验空间是RAID5的两倍
        - 适合重要数据的保存
    - 写操作
        - 要计算两个校验：P和Q，访盘的次数从4次增加到6次
- RAID 10
    - 特点
        - 先进行镜像，再进行条带存放



- RAID 01
    - 特点
        - 先进行条带存放（RAID0），再进行镜像（RAID1）



- 实现盘阵列的方式主要有三种
    - **软件方式**
    - **阵列卡方式**
    - **子系统方式**

- **通道**(通道处理机)
    - 输入输出系统的构成
        - CPU
        - 通道
        - 设备控制器
        - 外设
    - 专门负责整个计算机系统的输入/输出工作。通道处理机只能执行有限的一 组输入/输出指令
    - 通道的功能
        - 接收CPU发来的I/O指令，并根据指令要求选择指定的外设与通道相连接
        - 执行通道程序
        - 给出外设中要进行读/写操作的数据所在的地址
        - 给出主存缓冲区的首地址
        - 控制外设与主存缓冲区之间的数据传送的长度
        - 指定传送工作结束时要进行的操作
        - 检查外设的工作状态是否正常，并将该状态信息 送往主存指定单元保存
        - 在数据传输过程中完成必要的格式变换
    - 通道的主要硬件

        - 寄存器
        - 控制逻辑
    - 通道对外设的控制通过**输入/输出接口**和**设备控制器**进行
    - 通道的工作过程
        - 在**用户程序**中使用**访管指令**进入**管理程序**，由管理程序生成一个**通道程序**，并**启动**通道
        - **通道处理机**执行通道程序，完成指定的数据输入 /输出工作
        - 通道**程序结束**后向CPU发中断请求
    - 通道的类型
        - 字节多路通道
            - 用于连接多台低速或者中速设备
        - 选择通道
            - 为多台高速设备请求服务
        - 数组多路通道
            - 以数据块为单位、分时轮流地为多台高速设备提供服务。
    - 通道流量
        - 一个通道在数据传送期间，单位时间内能够传送的数据量。也称为通道吞吐率。

第八章 多处理机

并行计算机系统结构的分类

- Flynn分类法
    - SISD
    - SIMD
    - MISD
    - MIMD
- MIMD已经成为通用多处理机系统结构的选择
    - MIMD具有灵活性
    - MIMD可以充分利用商品化微处理器在性能价格比方面的优势
        - 计算机机群系统是一类广泛被采用的MIMD机器
- 根据**存储器的组织结构**，把现有的MIMD机器分为两类，每一类代表了一种存储器的结构和互连策略
    - **集中式共享存储器结构**
        - 最多由几十个处理器构成
        - 各处理器共享一个集中式的物理存储器
        - 这类机器有时被称为
            - 对称式共享存储器多处理机 SMP机器(Symmetric shared-memory MultiProcessor)
            - UMA机器（Uniform Memory Access）
    - **分布式存储器多处理机**
        - 存储器在物理上是分布的
        - 每个结点包含
            - 处理器
            - 存储器

        - I/O
        - 互联网络接口
    - 存储器分布到各结点有两个优点
        - 如果大多数的访问时针对本结点的局部存储器，则可降低对存储器和互联网络的带宽要求
        - 对本地存储器的访问延迟时间小
    - 缺点
        - 处理器之间的通信较为复杂，各处理器之间的访问延迟较大
    - 簇：超级结点
        - 每个结点内包含个数较少（例如2～8）的处理器
        - 处理器之间可采用另一种互连技术（例如总线）相互连接形成簇

- 存储器系统结构和通信机制
    - 两种存储器系统结构
        - 分布式共享存储器系统(DSM: Distributed Shared-Memory)(NUMA: Non-Uniform Memory Access)
            - 共享地址空间
                - 物理上分离的所有存储器作为一个统一的共享逻辑空间进行编址
                - 任何一个处理器可以访问该共享空间中的任何一个单元（如果它具有访问权），而且不同处理器上的同一个物理地址指向的是同一个存储单元
        - 把每个节点中的存储器编址为一个独立的地址空间，不同结点中的地址空间之间是相互独立的
            - 整个系统的地址空间是由多个独立的地址空间构成
            - 每个节点中的存储器只能由本地的处理器访问，远程的处理器不能对其直接访问
            - 每一个处理器-存储器模块实际上是一台单独的计算机
            - 这种机器多以集群的形式存在
    - 通信机制
        - 共享存储器通信机制
            - 共享地址空间的计算机采用
            - 处理器之间通过load和store指令对相同存储器地址进行读/写操作来实现的
        - 消息传递通信机制
            - 多个独立地址空间的计算机采用
            - 通过处理器间显式地传递消息来完成
            - 这些消息请求进行某些操作或者传送数据
            - 例如：一个处理器要对远程存储器上的数据进行访问或操作
                - 发送消息，请求传递数据或对数据进行操作
                    - 远程进程调用(RPC, Remote Process Call)
                - 目的处理器接收到消息以后，执行相应的操作或代替远程处理器进行访问，并发送一个应答消息将结果返回
            - 同步消息传递
                - 请求处理器发送一个消息后一直要等到应答结果才继续运行
            - 异步消息传递
                - 数据发送方知道别的处理器需要数据，通信也可以从数据发送方来开始，数据可以不经请求就直接送往数据接受方
        - 共享存储器通信的主要优点
            - 与常用的对称式多处理机使用的通信机制兼容
            - 易于编程，同时在简化编译器设计方面也占有优势
            - 采用大家所熟悉的共享存储器模型开发应用程序，而把重点放到解决对性能影响较大的数据访问上
            - 当通信数据量较小时，通信开销较低，带宽利用较好
            - 可以通过采用Cache技术来减少远程通信的频度，减少了通信延迟以及对共享数据的访问冲突
        - 消息传递通信机制的主要优点

        - 硬件较简单
        - 通信是显示的，因此更容易搞清楚何时发生通信以及通信开销是多少
        - 显式通信可以让编程者重点注意并行计算的主要通信开销，使之有可能开发出结构更好、性能更高的并行程序
        - 同步很自然地与发送消息相关联，能减少不当的同步带来错误的可能性
- 并行处理面临的挑战
    - 两个重要挑战
        - 程序中的并行性有限
        - 相对较大的通信开销

对称式共享存储器系统结构

- 多个处理器共享一个存储器
- 处理机规模较小时，十分经济
- 支持对共享数据和私有数据的Cache缓存
    - 私有数据
        - 供一个单独的处理器使用
    - 共享数据
        - 供多个处理器使用
- Cache一致性问题
    - 允许共享数据进入Cache，就可能出现多个处理器的Cache中都有同一存储块的副本
    - 当其中某个处理器对其Cache中的数据进行修改后，就会使得其Cache中的数据与其他Cache中的数据不一致
- 实现一致性的基本方案
    - 在一致的多处理机中，Cache提供两种功能
        - 共享数据的迁移
            - 把远程的共享数据复制一份，迁入本地Cache供本处理器使用，
                - 减少了对远程共享数据的访问延迟
                - 减少了对共享存储器带宽的要求
        - 共享数据的复制
            - 把多个处理器需要同时读取的共享数据在这些处理器的本地Cache中各放一个副本
                - 减少了访问共享数据的延迟
                - 减少了访问共享数据所产生的冲突
- Cache一致性协议
    - 在多个处理器中用来维护一致性的协议
        - 关键：跟踪记录共享数据块的状态
        - 两类协议
            - 目录式协议
                - 物理存储器中数据块的共享状态被保存在一个称为目录的地方
            - 监听式协议
                - 每个Cache除了包含物理存储器中块的数据拷贝之外，也保存着各个块的共享状态信息
                - Cache通常连在共享存储器的总线上，当某个Cache需要访问存储器时，他会把请求放到总线上广播出去，其他各个Cache控制器通过监听总线来判断它们是否有总线上请求的数据块。如果有，就进行相应的操作。
        - 两种方法来解决Cache一致性问题

        - 写作废协议
            - 在处理器对某个数据项进行写入之前，保证它拥有对该数据项的唯一的访问权。(作废其他的副本)
        - 写更新协议
            - 当一个处理器对某数据项进行写入时，通过广播使其他Cache中所有对应于该数据项的副本进行更新
        - 性能上的差别
            - 在对同一个数据进行多次写操作而中间无读操作的情况下，写更新协议需进行多次写广播操作，而写作废协议只需一次作废操作
            - 在对同一Cache块的多个字进行写操作的情况下，写更新协议对于每一个写操作都要进行一次广播，而写作废协议仅在对该块的第一次写时进行作废操作即可
                - 写作废是针对Cache块进行操作，而写更新则是针对字(或字节)进行
            - 考虑从一个处理器A进行写操作后到另一个处理器B能读到该写入数据之间的延迟时间
                - 写更新的延迟时间较小
- 监听协议的实现
    - 监听协议的基本实现技术(3个方面)
        - 处理器之间通过一个可以实现广播的互连机制相连。
            - 通常采用的是总线
        - 当一个处理器的Cache响应本地CPU的访问时，如果它涉及全局操作，其Cache控制器就要在获得总线的控制权后，在总线上发出相应的消息
        - 所有处理器都一直在监听总线，它们检测总线上的地址在它们的Cache中是否有副本。若有，则响应该消息，并进行相应的操作
    - 写操作的串行化：由总线实现
        - 获取总线控制权的顺序性