

# 体系结构期末

比较简单，PPT里有的，中文的会给出英文，回答中英文都可以看改试卷的人

翻译有问题，考试提问监考人员

十一章没有加optional那部分 前半部分

五级流水

填空，判断，选择（50），简答（50）

90分来自PPT，10分来自课堂讲的东西（？）

## ch1

选择判断填空 考一个填空或选择

- 计算机体系结构：设计、分析、选择和互联硬件组件以创建符合功能、性能和成本目标的计算机的科学和艺术
- 集成电路：集成电路便是把许许多多的电路元件集成在一个小小的硅片上面
- 摩尔定律：  
集成电路上的晶体管数目每隔约两年就会翻一倍

**Moore's law** is the observation that the number of transistors in an [integrated circuit](#) (IC) doubles about every two years. Moore's law is an [observation](#) and [projection](#) of a historical trend. Rather than a [law of physics](#), it is an [empirical relationship](#) linked to [gains from experience](#) in production.

- 晶体管结构
  - NMOS switch closes when switch control input is high 高电平闭合(高闭 所以negative)
  - PMOS switch closes when switch control input is low 低电平闭合
- Scalability
  - Scale Out(horizontal scaling): Add more components to a system (e.g. more nodes)
  - Scale Up(vertical scaling): Add resources to a single component in a system (e.g. upgrade memory)

## ch2

5-10分左右

简答题：考五级流水的图（阐述五级流水架构及功能 作用） 五级流水图要会（简答题） p35

- Aspects of Computer Design
  - Architecture (instruction set architecture): 指令级, 用户怎么去使用
  - Implementation (micro-architecture): 这个CPU (微架构) 是怎么构成的
  - Physical Design (chip realization): 在芯片上如何实现
- Instruction Set Architecture (ISA): 定义了数据流和控制流
  - 比如定义了怎么存数据 (内存寻址) 以及怎么算数据 (各种算数指令)
  - 定义了指令的格式和语义, 且对于同一个ISA会有多种CPU的implementation
- Complex Instruction Set Computer (CISC): 复杂指令集计算机(e.g. x86)
  - 编译器友好
  - (stack-oriented)用栈来传参, 节省程序计数器 (显式的Push Pop指令)
  - 寄存器-内存架构: 计算指令可以直接访问内存
  - 用condition code作为运算和逻辑指令的side effect
- Reduced Instruction Set Computer (RISC): 简单指令集计算机(e.g. MIPS)
  - 更少更简单的指令
  - (register-oriented)拥有更多的寄存器来传参
  - 只有load/store指令可以访存
  - 没有condition code
- User ISA and System ISA
  - User ISA:
    - 用于完成应用程序工作的部分
    - 数据流, ALU操作, 控制流
  - System ISA:
    - 对OS可见, 管理 (共享的) 资源
    - 优先级, 控制寄存器, 控制处理器, 内存, IO等重要资源的指令
- 5-Stage Pipeline
  - 流水线: 指令分为多个阶段 多条指令重叠执行的一种执行技术
  - 指令级并行: 利用指令之间的并行性 (统计和动态)
  - 五个阶段
    - i. **Instruction Fetch:** 从 Instruction Memory 中读取相关指令;

ii. **Instruction Decode** 解码获取到的指令，输出相应的控制信号;;从 Register File 从读取源操作数

iii. **Execute**: 利用 ALU 进行相关计算操作;

iv. **Memory Access**: 读写 Memory;

v. **Writeback**: 如果需要的话，将计算结果写回 Register File

- Speedup: 速度提升来自吞吐量的增加，指令延迟不会减少

- 一个理想的流水线

- i. 均匀的子计算 => 平衡流水线状态

- 1. 均匀的子计算表示在流水线中各个阶段的计算任务分布均匀。为了实现最佳性能，需要平衡流水线各个状态，以确保每个阶段都能够充分发挥作用。

- ii. 相同的计算 => 统一指令类型

- 1. 相同的计算表示在流水线中执行的计算任务相似或相同。为了提高效率，需要统一指令类型，以便在流水线中执行相似的操作，减少指令切换的开销。

- iii. 独立的计算 => 最小化流水线停滞

- 1. 独立的计算表示流水线中的各个阶段的计算任务之间没有依赖关系。为了减少流水线的停滞，需要最小化计算之间的依赖关系，确保每个阶段都能够独立执行，而不需要等待其他阶段的结果。

## ch3

Hazard/依赖 (哪三类讲清楚)

scoreboard的结构 (画结构, 给定example, 填写前五个cycle, P19, 21)

判断题考scoreboard局限性, 引出 tomasolu(48页)

Tomasulo Example考, 重命名, 前五个循环 tomasulo(和scoreboard不一样, 寄存器重命名, 给定同一个例子用tomasulo来执行)(58页)

Tomasulo与记分板 (IBM 360/91与CDC 6600)要考填空题, 挑战

tomasulo与scoreboard的差异(选择题)

Tomasulo的一些特点 (会考一些选择题小题) p79

会考指令的简单意思是什么, 比如ADD LD什么之类的 (没学过不懂)

## 依赖:

- data:
  - 直接: 比如j指令需要i的结果, 要等i先计算结束

- 间接: k需要j的结果, j需要i的结果
- RAW
- WAR
- WRW
- name: 两个指令用了同一个reg或者是memory的地址 (名称相关不是真正的相关)
  - Anti-dependence: 指令j写入一个寄存器或内存位置, 而指令i读取了这个位置的值。
  - Output dependence: 指令i和指令j都写入相同的寄存器或内存位置
- control: 由条件分支指令 (conditional branch) 所引起, 表现为一个基本块 (basic block) 是否可以确定被执行依赖于条件的计算结果 (evaluation of condition)

## 冒险

- 数据冒险 (Data Hazard): 如果一条指令的执行依赖于另一条未完成 的指令的运行结果
  - RAW 写后读
  - WAR 读后写 Anti-Dependence
  - WAW 写后写 Output-Dependence
  - 解决数据冒险
    - Forward/Bypassing: 当某个阶段的指令产生的数据可以被后续阶段的指令直接使用时, 可以通过旁路或转发的方式将数据直接传递给需要使用它的阶段, 而无需等待数据被写回到内存或寄存器。
    - pipeline interlock: 冻结较早的阶段, 直到所需数据变得可用为止
- 控制冒险 (Control Hazard): 如果在处理器还没有得出下一条要运行的指令的地址的情况下, 第一级流水线阶段就读取了可能有误的下一条指令
- 结构冒险: 当一条指令需要的硬件部件还在为之前的指令工作, 而无法为这条指令提供服务, 那就导致了结构冒险。
  - 解决: flush

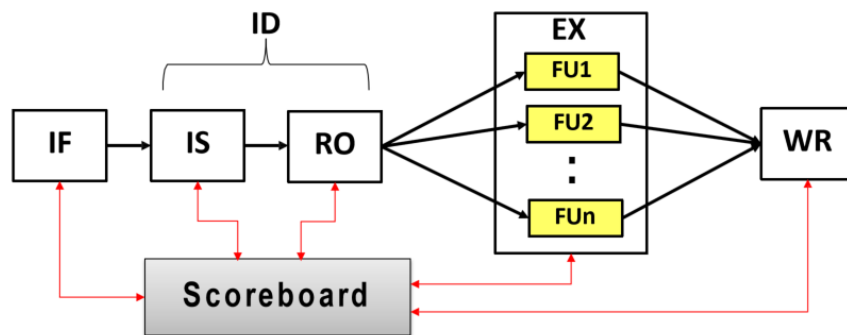
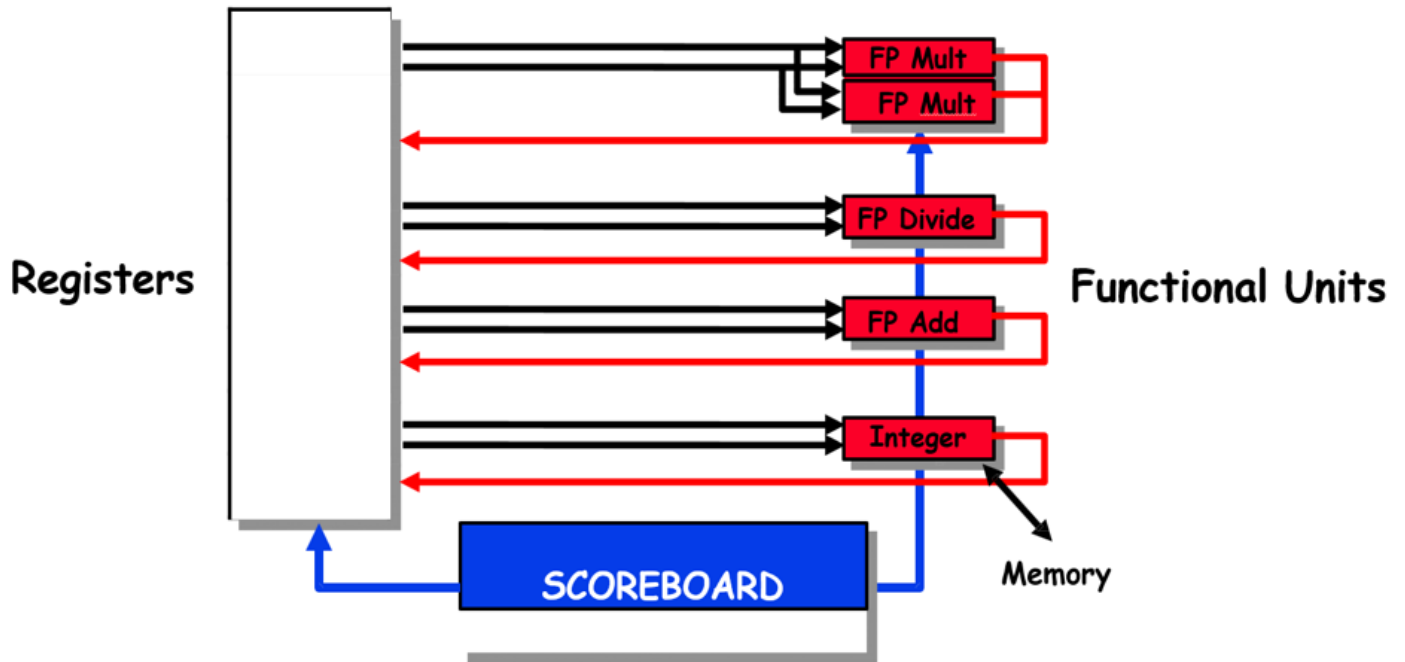
## 乱序执行

打乱指令顺序主要有两种方法:

- 静态调度: 在编译阶段静态的发现指令级并行, 再重新排序和优化指令
- 动态调度: 在硬件执行指令时动态的发现指令级的并行, 再重新排序指令
  - Control-centric: Scoreboarding
  - Data-centric: Tomasulo's algorithm

# Scoreboarding

目 体系结构笔记



优点：实现了指令的乱序执行，实现了指令的数据流式运行

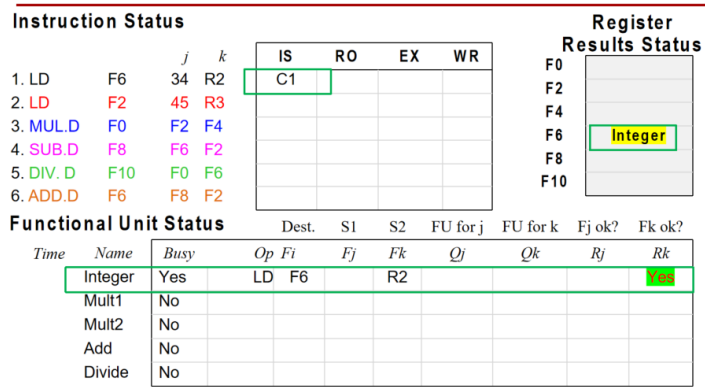
局限性

- 没有前递硬件
- 指令调度局限于基本块内(指令窗口小)
- 功能部件少（结构冒险），特别是integer/load store部件
- 存在结构冒险，就暂停发射指令，且一旦产生阻塞，后续相同类型的指令就没办法继续发射
- 等待到WAR冒险解决，停滞写回
- 防止WAW冒险，停滞发射

五个cycle

部件是integer

### Scoreboard Example: Cycle 1

**Functional Unit Status**

	Dest.	S1	S2	FU for j	FU for k	Fj ok?	Fk ok?
Integer	Yes	LD	F6	R2			Yes
Mult1	No						
Mult2	No						
Add	No						
Divide	No						

- Op : Operation to perform in the unit
- Fi : Destination register number
- Fj, Fk : Source register number

22

## Tomasulo:

关键技术：寄存器换名技术，寄存器被数值或者指针代替

- 消除WAR、WAW冒险
- 保留站比实际寄存器多，优化一些编译器不能优化的工作

区别：

Tomasulo	Scoreboard
(6 load, 3 store, 3 +, 2 x/÷) (1 load/store, 1 +, 2 x, 1 ÷)	≤ 5 指令
窗口大小: ≤ 14 指令	相同
结构冒险暂停发射	相同
WAR: 通过换名避免	暂停完成
WAW: 通过换名避免	暂停发射
从FU广播结果	写/读 寄存器
控制: 保留站	集中控制的记分板

缺点：

- 复杂
- 需要大量高速的相联存储(associative buffer)
- 公共数据总线 将成为 制约性能增长的瓶颈
  - 每个CDB必须广播到多个功能部件单元-->大容量、写操作密集

特点：

- 保留站来实现分布式控制
- 公共数据总线(CDB)广播计算结果
- 使用标签 (tags) 来标识数据值

和SB不同

- RS 分布式控制

- 结果转发给功能单元
- 数据总线广播结果

## Ch4

第四个ppt就考填空判断选择

简答题：指令集并行方法有哪些（超标量）

- 超标量自己看一看
- 分支预测有点难（是不考的意思吧）

主要填空判断

Tomasulo + ROB考给个例子前五个循环

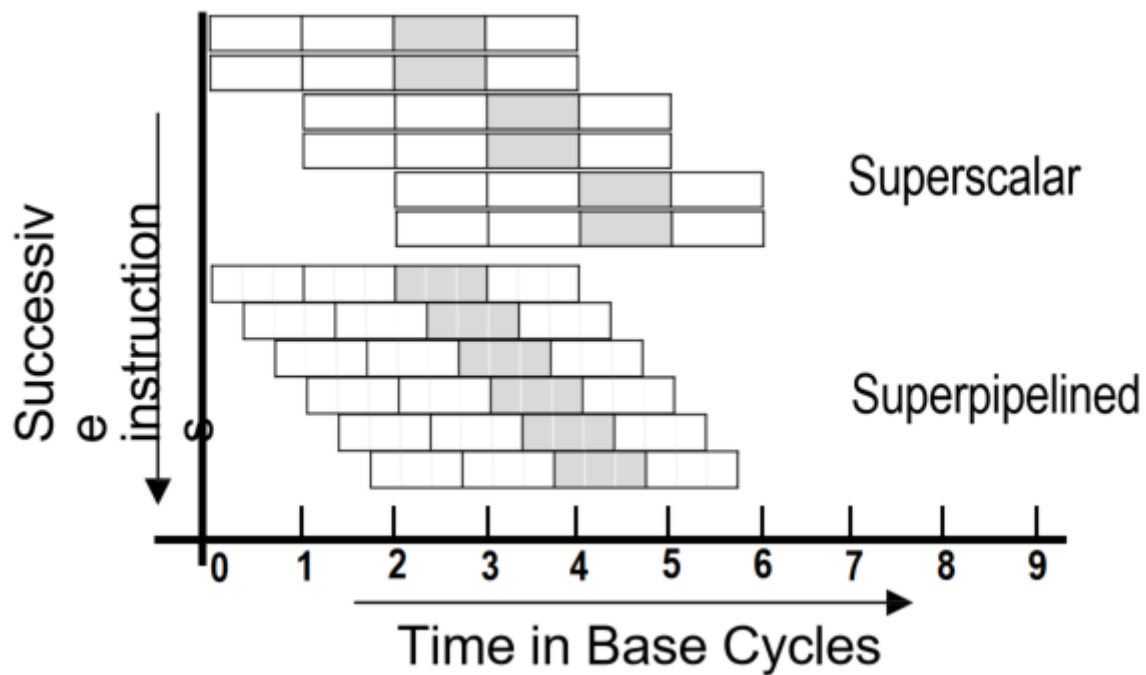
主要可能考填空题之类的（VLIW的简写）

### 指令级并行方法ILP:

- 流水线：将功能分为多个单独的硬件或者功能单元
  - 超级流水线：增加流水线级数的方法来缩短机器周期，相同的时间内超级流水线执行了更多的机器指令
- 多发射(multiple issue)：可以同时启动多个指令
  - Superscalar processors 超标量：同一周期内获取并处理多条指令并且每时钟周期内可以完成一条以上的指令
  - VLIW (very long instruction word) processors：将多个相互无依赖的指令封装到一条超长的指令字中
- 预测 (speculation)：尽早发现会使流水线停顿的事件
  - 分支预测
    - 静态：编译时预测分支行为，每个分支都有一个静态预测/固定的预测方向/基于profile
    - 动态：基于程序每个分支的运行时行为，在程序的生命周期内动态调整
    - 基于历史
    - 2-bit 状态机
    - Branch Target Buffer：存指令地址以及跳转地址
  - 值预测：指令的结果

Superscalar vs. Superpipelined:

- 超级流水线：增加流水线级数，时间上（depth）
- 超标量：同一时钟周期内获取并处理多个指令，从而在空间上展现了并行性（width）



## Reorder Buffer (ROB):

- 目的：让乱序执行的指令被顺序地提交
- ROB在这里就像是一个FIFO队列，指令在发射时入队，在提交时出队。
- <https://zhuanlan.zhihu.com/p/501631371>

## VLIW

VLIW: (Very Long Instruction Word) Processors

- 非常长指令字处理器

## EPIC (Explicitly Parallel Instruction Computing)

- 显示并行指令计算
- VLIW的一个延申，指令之间可以存在依赖关系

## ch5

memory 的结构 P4: 冯诺依曼+寄存器+L1 L2 L3cache+memory

cache层次，cache作用

Where to Put Data: Logical Organization of Cache

必考 cache优化方法 cache的优化方式：要知道了解

cache组织方式 cache的映射方式 P11肯定要考的

可以画出cache的结构（长什么样）



优化原理原则 优化的原则有哪些（选择填空）

Reducing Miss Rate 3Cmodel 3c model 考选填

Locality Principles考

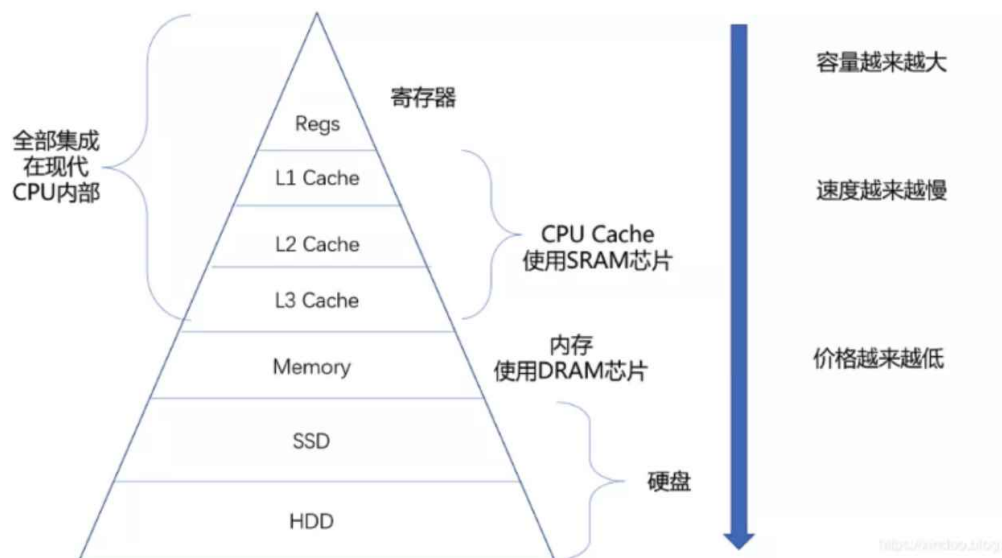
Discussion: Miss Caching vs Victim Caching 优劣对比

Overview of DRAM System 整个架构表示

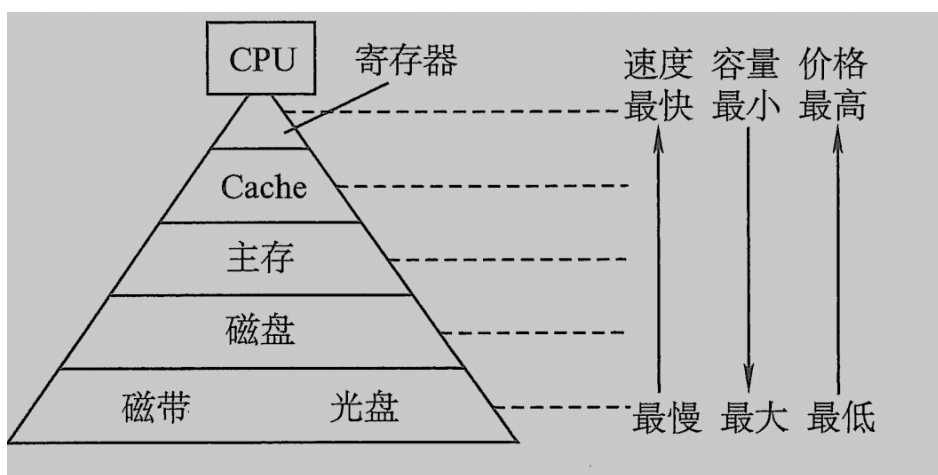
Drive Interface 这个图填空，IObus memory bus ,Drive Interface干嘛用的

Memory Hierarchy: 存储器层次结构的主要思想是上一层的存储器作为低一层存储器的高速缓存

- Registers
- Cache (SRAM): 高速缓冲存储器，主存和CPU之间
- Main Memory (DRAM): 主存/内存，存放计算机运行期间需要的程序和数据，CPU可以直接访问
- Secondary Storage (Disk)
- Tertiary Storage (Tape)



每一个现代处理器（CPU）都配置高速缓存（Cache）。目前CPU高速缓存级别主要分为L1/L2/L3三个级别；三个级别的高速缓存，缓存大小逐级增加同时访问速度逐次降低。



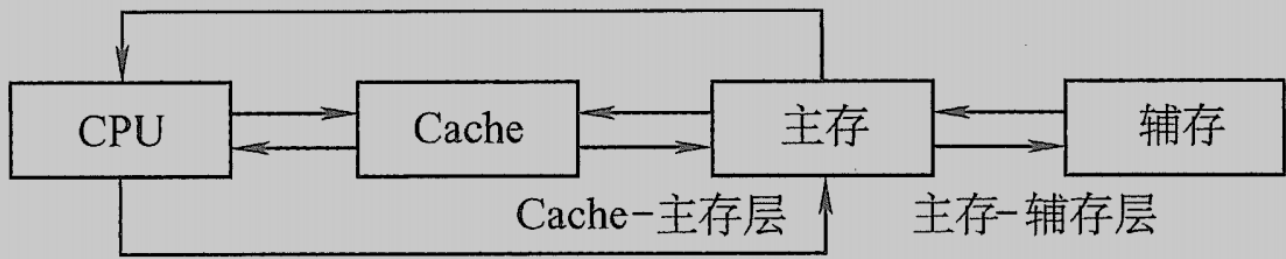


图 3.3 三级存储系统的层次结构及其构成

Cache-主存: CPU与主存在处理速度上不匹配问题

主存-辅存: 存储系统容量问题

高速缓存存在的主要原因是解决CPU与主存在处理速度上不匹配问题，从而极大提高CPU使用效率。

高速缓存均是依托于静态随机访问存储器（SRAM），而主存一般是依托于动态随机访问存储器（DRAM）

SRAM和DRAM:

- SRAM的访问速度要优于DRAM
- SRAM由CMOS技术和晶体管制成，而DRAM则使用电容器和晶体管
- SRAM不必刷新，效率更高，DRAM需要不断刷新（由于漏电）

## 高速缓存Cache

高速缓存：利用局部性原理，把程序正在使用的部分存放在一个高速的，容量较小的Cache中，使CPU访存操作大多数针对Cache进行，大大提高程序执行速度

局部性：

- 时间局部性：访问了某个数据项，有很大可能在接下来的一段时间内再次访问同一数据项；数据的短期重复利用
- 空间局部性：程序执行期间，相邻的内存位置很可能被频繁地访问
- 算法局部性：程序反复访问分布在内存空间中广泛区域的数据项或执行代码块时出现的局部性 | 程序在执行时呈现出局部性规律，即在一段时间内，整个程序的执行仅限于程序中的某一部分

为便于 Cache 和主存之间交换信息Cache 和主存都被划分为相等的块，Cache 块又称 Cache行，每块由若干字节组成，块的长度称为块长(Cache 行长)。

由于 Cache 的容量远小于主存的容量，所以 Cache 中的块数要远少于主存中的块数，它仅保存主存中最活跃的若干块的副本。因此Cache 按照某种策略，预测 CPU在未来段时间内欲访存的数据，将其装入 Cache。

当 CPU 发出读请求时

- 若访存地址在 Cache 中命中，就将此地址转换成 Cache 地址，直接对 Cache 进行读操作，与主存无关；
- 若Cache 不命中，则仍需访问主存，并把此字所在的块一次性地从主存调入 Cache。
  - 若此时 Cache 已满，则需根据某种替换算法，用这个块替换 Cache 中原来的某块信息。（值得注意的是，CPU 与Cache 之间的数据交换以字为单位，而 Cache 与主存之间的数据交换则以 Cache 块为单位）

## 映射方式：用于地址变换

- 直接映射：主存中每一块只能装入Cache中唯一位置，如果这个位置已有内容，产生冲突，直接替换
  - 主存地址结构：标记：块号：块内地址（t：c：b）
- 全相连映射：主存中每一块可以装入Cache的任何位置，每行的标志指明该块取自主存的哪一块
  - 主存地址结构：标记：块内地址
- 组级联映射：Cache分为大小相同的组，主存数据可以在一个组的任何位置，即组间直接映射，组内全相联映射
  - 主存地址结构：标记：Cache组号：块内地址
  - [标记：Cache组号] 是主存块号

## Cache Miss 3C model:

- Compulsory miss:冷启动缺失，第一次访问存储块时，由于该块不在cache中，所以必须首先将引块从主存取入cache.
- Capacity miss:容量缺失：cache不能容纳所有存储块
- Conflict miss: 当多个数据块映射到缓存的同一组（set）中，而缓存的这个组只能容纳其中的一部分数据块时，可能会发生冲突缺失。

## Cache Write Policies

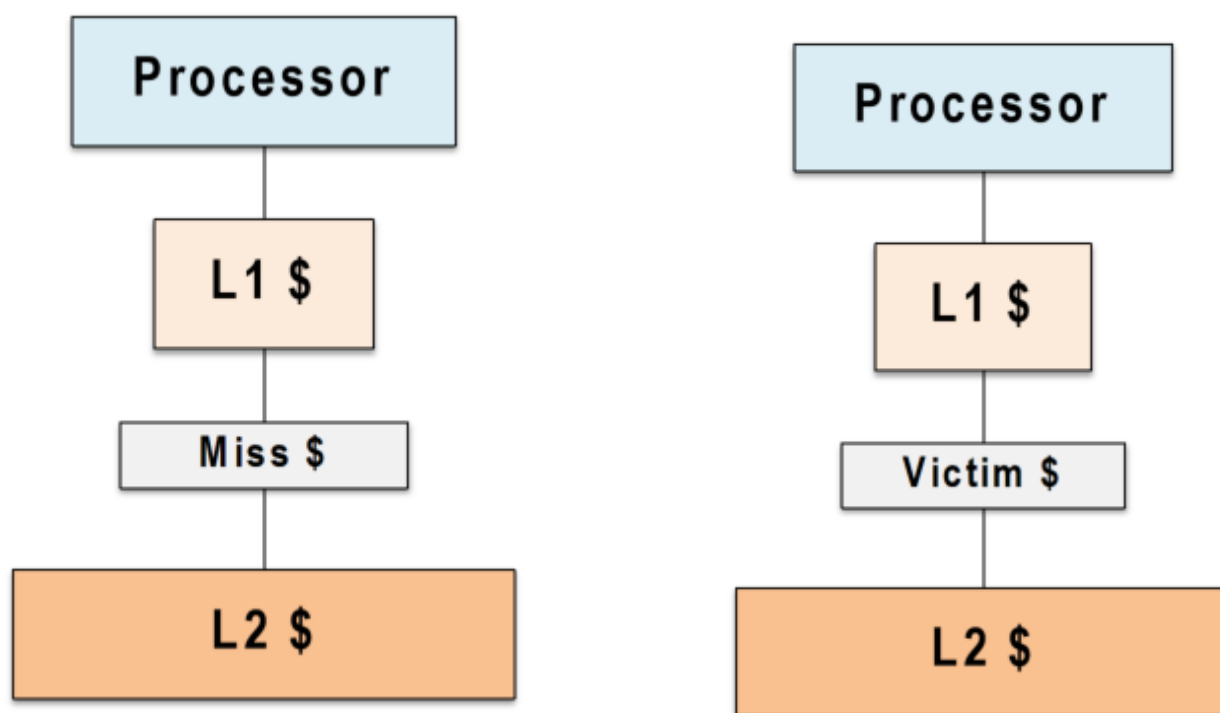
- cache hit的时候写的policy
  - Write-back（写回）：lazy的方式，modify暂时只写到cache上，在替换的时候才把脏块写到memory里
  - Write-through（写直达/写穿）：对cache和对memory一起写。这样实现简单，保持缓存的干净状态，但可能导致更多的主存访问。

- cache miss的时候写的policy
  - Write allocate: 将主存的block带到cache上并且更新
  - no-write allocate: write miss不影响cache, 直接写主存

## Cache 优化方式

<https://zhuanlan.zhihu.com/p/147661339>

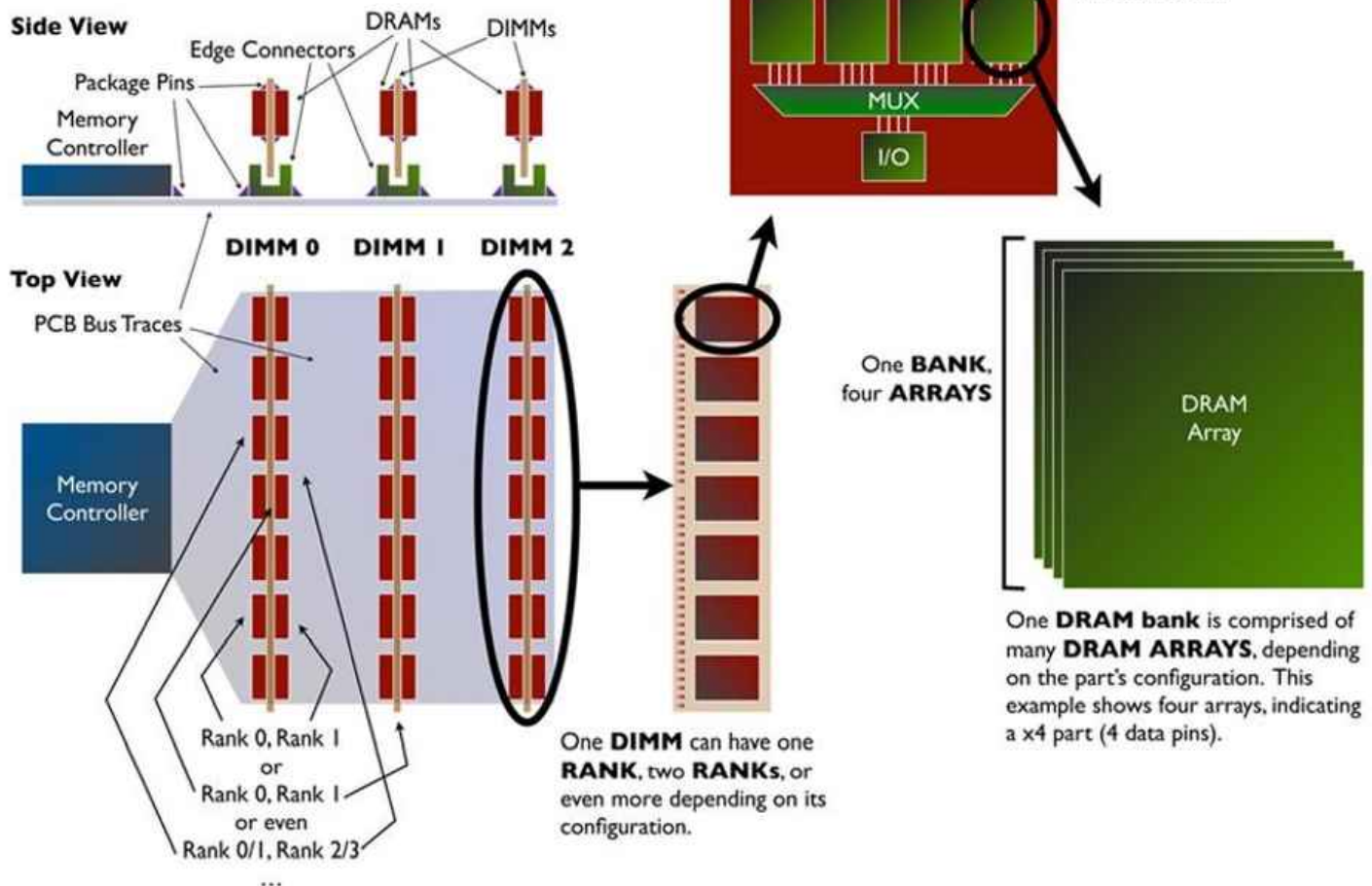
- Miss Caching:
  - L1和L2之间插入全相联Cache
  - L1 miss在Miss块里面找, 找到了取给L1; 否则在L2里面找, 找到了取给L1和Miss块
- Victim Caching: L1
  - 一般采用全相联结构, 容量小 (4-16个数据), 与cache为互斥关系
  - L1 miss 在Victim里面找, 找到了取给L1, Victim同时存下L1被替换出来的; 找不到在L2里面找



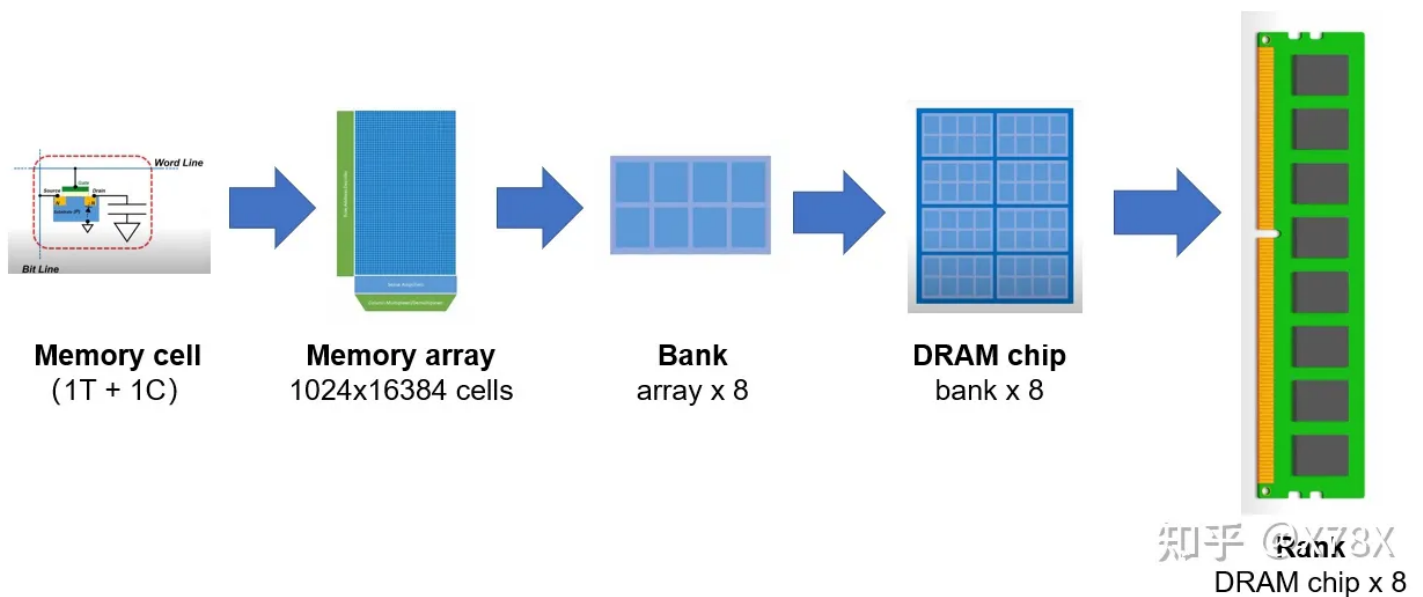
- 对比优劣
  - Victim Caching更小, 1行都能有用 性能更好

## DRAM 的结构图

# Overview of DRAM System



- DIMM (Dual in-line memory module) 层次架构
  - 每一个DIMM有一个或多个rank
    - 每一个rank有一组DRAM Devices (rank包含许多DRAM芯片)
      - 每一个DRAM device有一个或多个bank
        - 每一个bank有一些memory array(每个 Bank 包含一个独立的内存数组)
- Channel: MC和DRAM module之间的通道, 也就是平时说的多少通道内存
- Dram chips上的xN表示DRAM至少有N个memory array在一个bank上, 并且每一个列的宽度都是Nbits (一次传输N bits data)



## 虚拟存储

页表：an array of page table entries (PTEs) that maps virtual pages to physical pages

## ch6 Data Storage and I/O

IO BUS MEM BUS P11 3-5pt

11页内容以填写的形式考简答

IO Bus 和 Memory Bus 填图

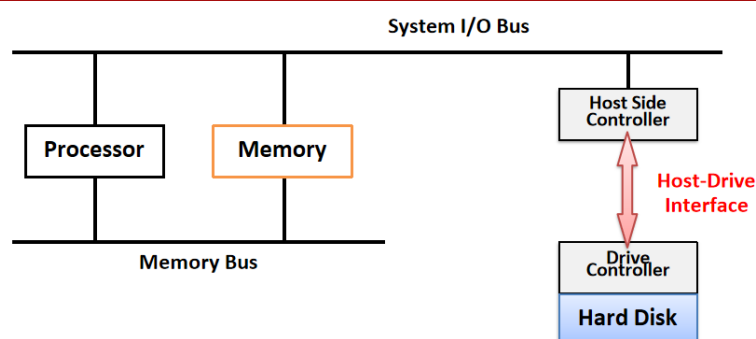
IO Bus 的图，drive interface 是什么

反正得搞明白这个架构

**Memory bus**：处理器和memory之间的

**I/O bus**：处理器和磁盘之间

### Drive Interface



- The drive interface is a bridge between host and the disk
  - The communication channel for I/O requests
  - Allows data transfers for reading and writing

## Drive Interface:

驱动接口是主机和硬盘之间的桥梁

I/O requests 通信

为数据读写传输数据

## ch7

列举出熟悉的architecture simulation，说出其作用是什么（Gsim GPGPU\_SIM 什么的要了解 简要阐述）；

选择题说明下列哪些不是模拟器；

简答：模拟器简要阐述

## Architecture Simulation

- Functional Simulation
  - 指令集模拟器：它模拟了计算机系统的指令执行，但不涉及具体的硬件时序或性能特性
  - 可以用来生成指令和地址trace，用来给其它模拟工具做输入
  - 主要关注系统是否按照设计规范执行指令，而不是关心系统在特定条件下的性能表现。
- Trace-Driven Simulation
  - 把指令和地址的trace放入一个微架构timing simulator里面模拟
  - 把功能simulation和计时simulation分离开
  - 缺点：需要trace file，以及在mis-predicted paths上不能有效做预测
- Execution-Driven Simulation
  - 目前通用的simulation
  - 把功能simulation和时间simulation结合在一起，并且准确度比Trace-Driven更高
    - 缺点是要更多的开发/evaluation时间
    - Example: SimpleScalar GEMS Simis M5 PTLSim GPGPU-Sim

**SimpleScalar:** 超标量、5级流水的RISC体系结构模拟器，提供了从最简单到超标量乱序发射的不同的模拟程序。

**GPGPU-SIM**模拟器的工作原理是让CUDA/OpenCL程序运行在模拟器上而不是GPU硬件，会用模拟器库代替真实的硬件运行库，从而用软件实现代替GPU硬件运行机制以达到模拟GPU行为。

**GEM5:** 包括系统级架构以及处理器微架构，一个模块化离散事件驱动的计算机系统模拟器平台

**gem5-gpu:** 结合gem5和GPGPU-Sim，实现对CPU-GPU异构系统的模拟。



PTLsim：基于 Xen 半虚拟化环境，运行 PTLsim 模拟器的操作系统要进行定制化修改

M5：全系统时钟精确模拟器 提供了高度可配置的**模拟**框架

## ch8 Multiprocessor and TLP

p4 并行架构的分类（选择填空）

P6 一些代表性的级别的并行性 几种类型的并行

p8 把图给出来，问你是什么架构 UMA 和 NUMA 架构是什么

p9 两个代表性的多处理器

p10 看一看（可能有判断）

p11 DSM 可能有判断

p12可能会考

cache一致性问题：要讲清楚什么是一致性问题 一致性和一惯性

p26 简单考下概念

保持一致性的协议

为什么要保持一致性，要讲清楚（ppt中有，什么结构不一致）

### FLYNN：

- SISD: Single Instruction Single Data 单指令流单数据流
  - 单指令多数据流
  - 单处理器
- SIMD: Single Instruction Multiple Data 单指令流多数据流
  - 多个数据执行相同的指令
  - 矩阵处理器
  - 向量处理器
  - 超标量
  - GPU
- MISD: Multiple Instruction Single Data 多指令流单数据流
  - 无
- MIMD: Multiple Instruction Multiple Data 多指令流多数据流
  - 多个指令流在多个数据流上操作
  - 多核计算机/超级计算机(集群)

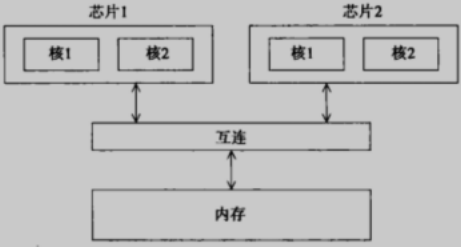
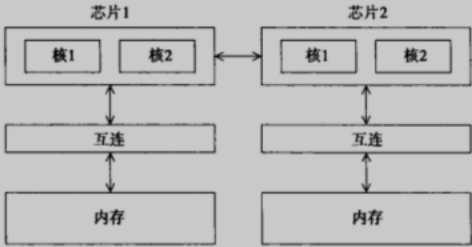


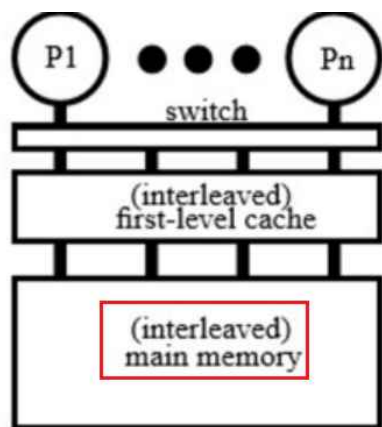
## 并行：

- 指令级并行：多个处理器组件或功能单元同时执行指令来提高处理器性能
  - 流水线 (pipelining)：将功能分为多个单独的硬件或者功能单元，将功能单元分阶段安排
  - 多发射 (multiple issue)：可以同时启动（发射）多个指令，复制功能单元来同时执行程序中的不同指令
    - 超标量：superscalar支持动态多发射的处理器
- 线程级并行：通过同时执行不同的线程来提供并行性
  - 对于多处理器的架构，在多个处理器上做并行（比如OpenMP等）
  - Single chip multi cores 多核处理器
  - 多处理器线程并行
- 数据级并行：在多核的加速卡（如GPU）上做并行（比如CUDA）
  - Many cores
- 任务级并行：在集群中有多个任务一起处理

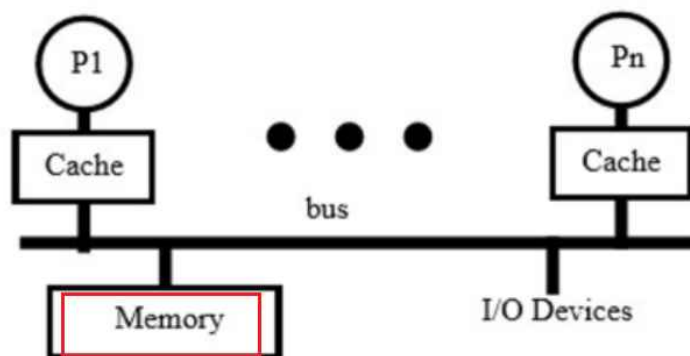
## Multiprocessors内存架构：

- UMA (Uniform memory access)：
  - 名称：所有处理器访问存储器延迟一致
  - 多个处理器共享同一个物理内存（图里面一个内存）
  - Centralized Shared-Memory / SMP (symmetric multiprocessors)
- NUMA Non-uniform memory access: d图
  - Distributed Shared-Memory
  - 物理内存分开

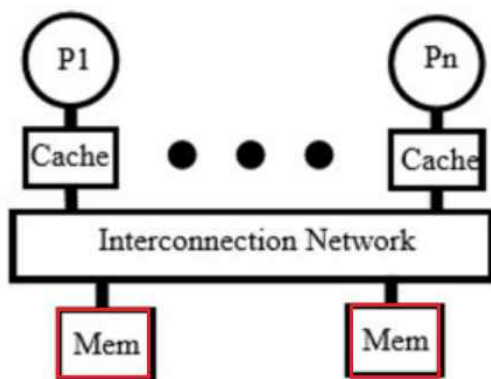
UMA一致内存访问系统	NUMA非一致内存访问系统
 <p>Diagram of UMA system: Two chips (芯片1, 芯片2) each containing two cores (核1, 核2) are connected via a central interconnect (互连) to a shared main memory (内存).</p>	 <p>Diagram of NUMA system: Two chips (芯片1, 芯片2) each containing two cores (核1, 核2) are connected via local interconnects (互连) to their own local main memory (内存). A horizontal arrow indicates communication between the chips.</p>
所有处理器之间连接到主存	每个处理器连接一块内存,处理器内置硬件可以访问内存的其他块
访问任何区域时间相同	访问之间相连的区域快很多



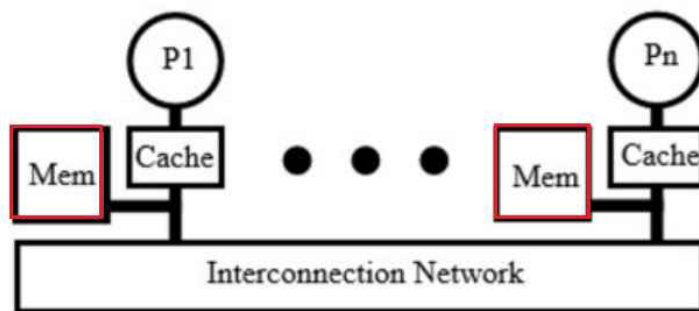
(a) Shared Cache



(b) Bus-based Shared Memory

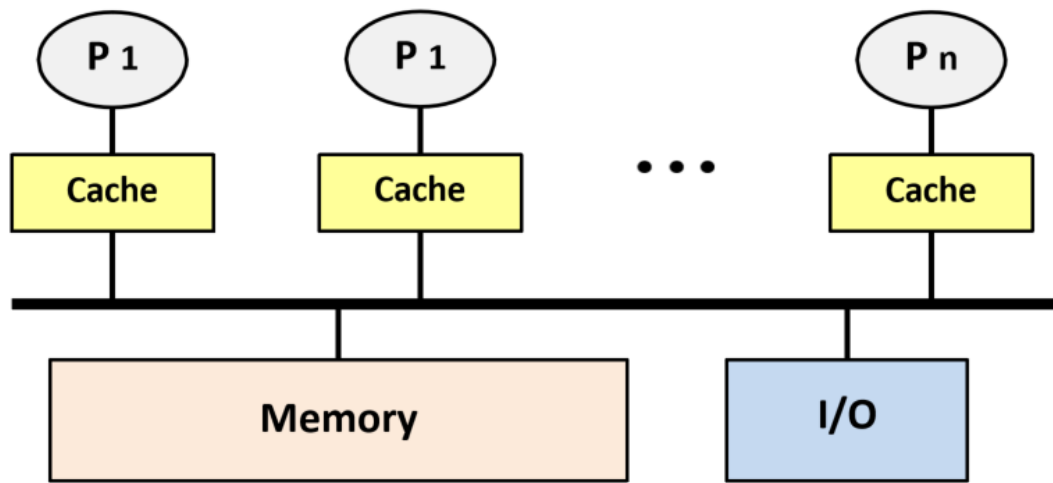


(c) Dance-hall

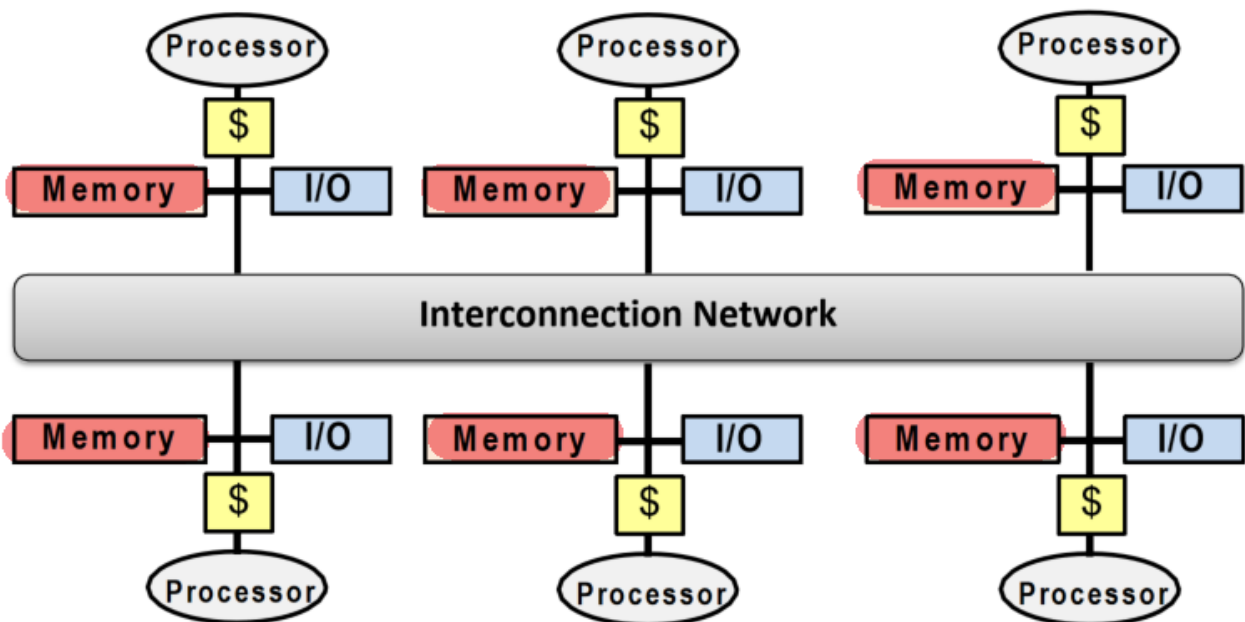


(d) Distributed Memory

- Centralized Shared Memory 集中式共享存储
  - 又称symmetric multiprocessors (SMPs)
  - 共享一个集中式存储,所有处理器平等地访问(对称地由来)



- Distributed Shared Memory (DSM): 物理分布式存储，每个处理器都有自己私有的内存空间



- Two major benefits
  - Scalable memory bandwidth
  - Low-latency local access
- Key disadvantages
  - Complex communication
  - Higher node-node latency

## 不同MIMD系统的比较

**Multiprocessors**：多处理器（包括共享内存多处理器SMP和分布式共享内存DSM）

- Tightly coupled architecture (紧密耦合结构): 处理器之间的连接较为紧密，通常通过总线或互连网络相连接
- Processors connected via bus or interconnect network: 处理器通过总线或互连网络相连接
- Consists of a few processors (2 ~ dozens): 典型的多处理器系统通常包含几个到几十个处理器
- A single shared address space: 共享地址空间 单一共享地址空间的结构，所有处理器共享相同的内存地址
- Communicate data implicitly via load and store: 处理器之间通过隐式的内存加载（load）和存储（store）来隐式地进行数据通信
- Thread-level parallelism (线程级并行)

## Multicomputers, Clusters 多计算机系统、集群

- Loosely coupled architecture (松散耦合结构): 处理节点之间的连接较为松散，通常通过局域网连接。
- Individual computers connected on a local area network (通过局域网连接的个体计算机):
- Consists of a large number of nodes (包含大量节点): 通常包含大量节点，可能达到数百或数千个。
- 私有地址空间
- Explicitly passing messages among the processors (在处理器之间显式地传递消息): 处理节点之间通过显式地传递消息来实现通信，而不是通过共享内存。
- Request/Task level parallelism (请求/任务级并行): 通过请求或任务级别的并行来实现并行计算，各节点独立执行任务，通过消息传递进行协同。

## Cache Coherence

单处理器:

多处理器:

- 共享数据在多个缓存中存在多个拷贝，并且由不同的处理器进行操作。当不同的处理器看到相同内存位置的不同数值时，就会出现缓存一致性问题。
- 主要是由于多处理器的共享存储（如主存）中保存着全局状态，而不同的处理器又各自拥有本地缓存，处理器对本地缓存的操作没有及时传播到其他处理器的本地缓存或共享存储，从而导致本地缓存中的数据可能与共享存储中同一地址对应的数据出现不一致（）。

一致性:

---

- 强调的是读操作返回的值的一致性（最新写入值）
- 关注的是多个处理器或设备之间，对于同一内存位置的读操作，系统是如何保证得到一致结果的

### 一贯性：

- 强调的是写操作何时被读取（强调读的时间概念）例如写入和传播的延迟可能造成写入值不能马上被读取到）
- 一致性更关注的是对于读写操作的时序概念，即写入和读取之间的关系。

### 保持一致性：

- **监听cache一致性**
  - 更新时广播通知其他核包含x的cache已经更新(其实是更新cache行)
  - 通常要求有足够的带宽，因为需要监视系统上的所有事务。
  - 每个cache block对应着看一个共享的状态，系统中所有的cache控制器通过共享总线进行互联，每个cache控制器都监控着共享总线上的操作，根据共享总线上的命令来更新自己的共享状态。
- **基于目录的cache一致性协议**
  - 在某个单一的位置（directory）对某个cache line的共享状态进行跟踪。
  - 当变量需要更新，会查询目录，将包含变量的行高速缓存标记非法
  - 优点:当一个Cache变量更新,只需要与存储这个变量的核交涉
  - 缺点:需要额外的空间存储

### 目录跟踪cache line 状态

为物理缓存块专门设立一个保存共享状态的目录，缓存去查询目录得到块的共享状态；DSM多采用分布式的目录一致性协议

## ch9 CMP and Multicore System

3-5pt 无大题

线程级并行：线程在干嘛

多线程类型（判断各个图）（细粒度、粗粒度、同步多线程）

- p9 可能会给判断，判断哪个是哪个

多核+线程级并行

Style of Multithreading

Impacts of SMT on Utilization

CMP vs. SMT

## MIMD

### 线程级并行(Thread-level parallelism, TLP)：

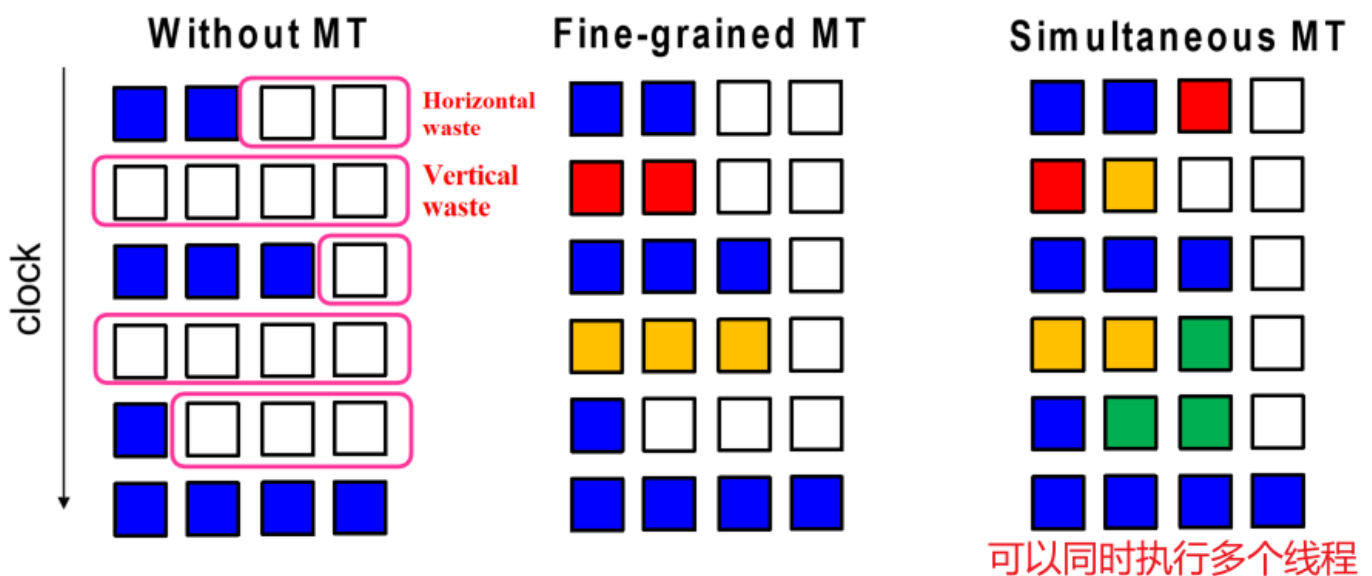
**线程：** 处理器利用的基本单元

- 具备执行所需的所有状态
- 在同一进程中的线程共享代码和数据

**线程级并行：** 通过同时执行不同的线程来提供并行性

**分类：**

- 细粒度多线程 (Fine-grained multithreading)
  - 每个时钟周期在线程之间切换，轮询，跳过停顿地线程
- 粗粒度多线程 (Coarse-grained multithreading)
  - 只在发生停顿时间长时才切换
- 同步多线程 (Simultaneous multithreading, SMT) (同时)
  - 一个周期发射多个
  - 是细粒度多线程的变体，通过允许多个线程同时使用多个功能单元来利用超标量处理器的性能
  - SMT 可以充分利用独立程序的并行性，也可以利用单个程序内部的并行性。这意味着它能够同时执行多个线程



### CMP (Chip Multi-Processor)

- 单芯片多处理器，多个处理核心 (cores) 被集成到一个处理器插槽 (socket) 上

- 也被称为多核处理器 multicore processors
- 通常是共享内存多处理器（Shared-memory multiprocessor）和多指令多数据（Multiple Instruction Multiple Data, MIMD）

### Why multicore:

Single core SMT: 单核

- 主要利用指令级并行性
- 为了提高任务执行速度，唯一的选择是增加时钟频率，但这会极大地增加功耗和热量散发，并导致设计变得更加耗时且难以验证。

Multicore solution:

- 更好地利用了线程级并行性
- 在同一芯片上集成两个或更多个核心，充分利用各核心以有效频率运行
- 使用经过验证的处理器设计，制造成本较低

### Multicore vs Manycore

Manycore:

- 具有大量核心的体系结构
- 实现更高层次的并行性，旨在提高整体吞吐量
- 通常伴随着单线程性能的降低，因为系统更注重同时处理多个任务而不是优化单个任务的执行

Multicore:

- 核心数量较少的体系结构
- 优化以同时支持并行和串行代码，并利用乱序执行（Out-of-Order, OOO）、深度流水线等技术
- 更注重单线程性能，即对于单个任务的优化执行

### CMP vs SMT

CMP: chip multi-processing (芯片多处理):

- 单芯片多处理器：同一芯片上集成多个物理核心
- 每个物理核心都拥有独立的资源，例如L1缓存（L1 cache）、转换后备缓冲（TLB）、程序计数器（PC）和通用寄存器（GPR）。而L2缓存（L2 cache）可能是共享的。

SMT: simultaneous multithreading (同步多线程):

- 同一物理核心上支持多个线程
- 不同于CMP，SMT中的多个线程共享所有处理器资源，包括缓存和转换后备缓冲。程序计数器和通用寄存器对每个线程是独立的。

HT: hyper threading (超线程):

---

- HT是英特尔的SMT技术。它是SMT的一种实现，使得单个物理核心能够同时支持多个线程。HT允许多个线程共享同一物理核心的资源，提高了处理器的整体并行性。

## ch10 Data-Level Parallelism and GPGPU

数据级别并行

根据图分辨什么是SISD和SIMD

P5图片填空 向量处理器优势

向量指令干嘛 向量处理器的意义、有什么优势（三个） 优缺点

Vector Architecture两类，代表性有哪些

P12图片 向量指令的意思+汇编指令的意思

Case Study: VMIPS 的图 VMIPS的五大部件\要把部件能全部画出来、作用；

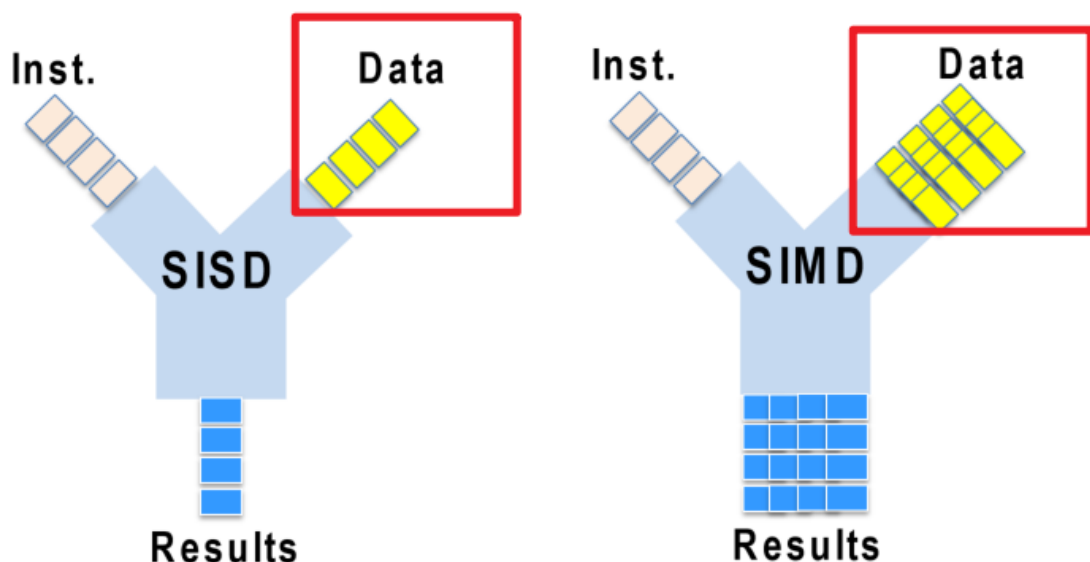
p13 考一些简单的指令是什么意思

- 向量指令的意思+汇编指令的意思

register bypass缓解冲突依赖 旁路：缓解冲突依赖（P17: vector chaining）

### 数据级并行(Data-Level Parallelism,DLP)

- 处理器能够同时处理多条数据



- 向量处理器 / GPU



# 向量处理器 Vector Processor

传统CPU处理单独数据单元/标量，向量处理器处理一组向量

**Vector Instructions (向量指令):** 操作数据项序列的指令。这种指令设计的目标是对数据序列执行相同的操作，以便高效地进行并行计算。

优点:

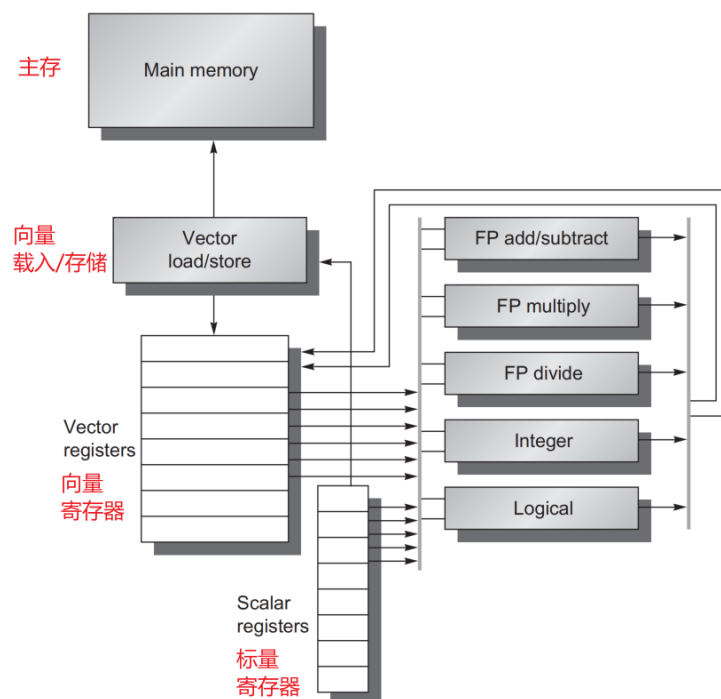
- Reduced instruction fetch bandwidth (减少指令获取带宽): 可以通过单个指令来执行大量的工作
- Less data hazard checking hardware (减少数据冒险检测硬件): 相邻元素之间的操作是独立的，因此减少了数据冒险检测硬件的需求。
- Memory access latency is amortized (内存访问延迟分摊): 因为一次指令执行中可以操作多个数据元素，从而隐藏了一部分内存访问的延迟。

两类Vector Architecture:

- memory-memory vector processors: All vector operations are memory to memory
  - 有矢量运算的操作数都存储在内存中，而不是寄存器中
  - CDC STAR-100
- vector-register processors: Vector equivalent of load-store architectures
  - load-store architectures
  - Cray、Convex

## VMIPS

结构



- 向量寄存器：
  - 长度数组，保存向量
- 向量功能单元：向量中的每个元素需要相同的操作(SIMD)
  - 5个
  - 检测冒险
- 向量载入/存储 单元：
  - 载入/存储 向量
  - 每个时钟一个字

指令解释： 操作向量而不是标量

- 向量+向量： `ADDVV.D V1, V2, V3` // add elements of V2 and V3, then put each result in V1
- 向量+标量： `ADDVS.D V1, V2, F0` // add F0 to each elements of V2 , then put each result in V1
- 载入向量： `LV V1, R1` // Load vector register V1 from memory starting at address R1
- 存储向量： `SV V1, R1` // Store vector register V1 from memory starting at address R1

## Vector Chaining

(register bypassing) 的向量化版本

允许数据可用的时候立即执行

## GPGPU

GPU: Graphics processing unit：图形处理单元， 一种大规模并行架构

GPGPU: general-purpose computation on GPU 在GPU上进行通用计算任务，GPGPU的概念允许开发者充分利用GPU的大规模并行性进行通用计算任务

提供高级编程接口，允许开发者使用高级编程语言和工具来利用底层硬件的大规模多线程处理能力

CUDA / CTM

Data Parallel Workloads： 多个数据输入上进行相同且独立的计算

- Multiprocessor Approach (MIMD)： **将独立的任务分配给多个中央处理单元（CPUs）**
- SPMD Approach (Single Program Multiple Data)： 多个处理单元上执行相同的程序，但每个处理单元处理不同的数据集。
- SIMD/Vector Approach (Single Instruction Multiple Data): 相同且独立的任务分配给多个执行单元

# ch11 Interconnection Networks

## Switch Network Topologies 交换网络拓扑结构

- 换网络视为一个图
  - 顶点:节点/交换机
  - 边: 通信路径
- Structure of the Network Graph (网络图的结构):
  - Direct network 直接网络: host节点连接到每个交换机, a.k.a. distributed switch
  - Indirect network间接网络: (主机仅连接到特定的交换机)

Bus (总线): 一种将所有输入连接到所有输出的网络拓扑结构, 使用单一的开关进行连接。

- No Fault Tolerance: Single Point of Failure (无容错性: 单点故障)
- Low Performance: Bandwidth is  $O(1)$  (性能较低: 带宽为 $O(1)$ ): 任何给定时刻只能有一个事务在进行
- Bus: the Interconnect is Mostly Just Wires (总线: 连接主要是电缆):

交叉 (Crossbar) 结构:

- 全连接网络, 每个节点都与所有其他节点直接连接, 形成了一个高度互连的网络
- $O(N)$  Bandwidth (带宽为 $O(N)$ ): 由于每个节点都可以直接与所有其他节点通信, 交叉开关的带宽是与节点数量成正比的, 即 $O(N)$
- Cost of Interconnect:  $O(N^2)$  (互连成本):
- Good for Small Number of Nodes (适用于节点数较少): 互连成本

Linear Arrays and Rings:

- Array: 由双向连接的节点组成的网络。每个节点与相邻节点直接连接。
- Ring: 环是在线性数组的基础上通过在两端添加直接连接而形成的。这使得首尾相连, 形成一个环形结构。

Multidimensional Topology:

- 2D Mesh (二维网格):
  - 节点以网格的形式相互连接, 形成一个平面的结构。每个节点通过水平和垂直方向的链接与其相邻节点相连。

- Multiple Routes Between a Pair of Nodes (节点对之间存在多条路径): 由于节点以网格形式连接, 从一个节点到另一个节点通常存在多条路径, 可以选择不同的路由。
- Cost of Interconnect:  $O(N)$  (互连成本:  $O(N)$ ): 二维网格的互连成本是与节点数量 $N$ 成正比的。
- 2D Torus 二维环形网络
  - 二维环形网络是对二维网格的扩展, 其中额外添加了首尾相连的直接连接, 使得网络呈现环形结构。
  - 通常的环形网络具有长的环绕连接, 即节点之间的连接会绕过网络的边界, 形成环形结构。这些环绕连接允许更多的路径选择。
  - 可能具有稍微较低的延迟, 但与此同时, 由于额外的连接, 网络的成本可能较高。