

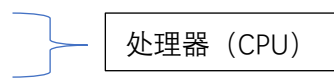
# 计算机组成原理复习概要

Edited By [SykpeWookal](#)

## Chapter 0 课程介绍与背景知识

- 计算机两个基本特征：1. 运行程序；2. 计算机由硬件+软件组成

## Chapter 1 计算机系统概论

- 软件固化：将软件的一些功能固化地保存在只读存储器中，称为**固件**
  - 功能上是软件，形态上是硬件
  - 一些系统软件的核心部分（如操作系统内核、常用软件的固定部分）常被固化在存储芯片中
- 计算机硬件的五大组成部件：
  - 运算器(数据通路)：完成算术运算和逻辑运算，暂存中间结果
  - 控制器：控制、指挥程序和数据的输入、运行以及处理运算结果
  - 存储器：存放数据和程序
  - 输入设备：将人们熟悉的信息形式转换成机器能识别的形式
  - 输出设备：将机器运算结果转换为人们熟悉的信息形式展示出来

The diagram shows a bracket on the right side of the five components (Arithmetic Logic Unit, Controller, Memory, Input Device, Output Device) pointing to a box labeled '处理器 (CPU)'.
- 冯·诺依曼机的特点：
  - 五大组成部件，以运算器为中心
  - 数据和指令用二进制数表示，**以同等地位存放于存储器中**，按地址访问
  - 指令由操作码和地址码组成，在存储器中按顺序存放
- 冯·诺依曼结构也叫做普林斯顿结构
- 哈佛结构：将指令和数据**分开存储**的结构
- 冯·诺依曼结构与哈佛结构的区别：
  - 指令和数据是否分别存储
  - 是否使用两条独立的总线，分别作为 CPU 与每个存储器间的专用通信通道，两条总线间毫无关联
  - 哈佛结构目前较多出现在嵌入式应用中
- 现代计算机的组成结构以存储器为中心，其三大组成部件为：CPU、主存储器、I/O 设备
- 运算器和控制器在逻辑关系和电路结构上联系紧密，将其集成在同一芯片上，称为中央处理器 CPU
- 运算器常称为算术逻辑运算部件 ALU
- 控制器的基本任务：
  - 按照计算机程序编排好的指令序列，从存储器中取出**一条指令**——取指过程
  - 对指令进行分析，确定要完成的**操作（操作码）**，并按寻址特征找出**操作数**的地址——分析过程
  - 根据**操作数地址**及指令的**操作码**，完成相应的操作——执行过程
- 控制器的组成：
  - 程序计数器 PC：用于存放当前将要执行的指令，具有自动加 1 的功能，能够自动形成下一条指令

的地址

- 指令寄存器 IR：存放 CPU 当前正在执行的指令内容
- 控制单元 CU：分析当前指令所要完成的操作，并发出各种微操作命令序列
- CPU 的功能：
  - 算术逻辑运算
  - 指令译码、执行
  - 数据暂存
  - 与 MEM、I/O 交换数据
  - 提供整个系统的定时和控制
  - 响应中断请求
- CPU 指令的执行过程：
  - 取指：根据 PC 访存读取当前要执行的指令；PC + 1
  - 译码：识别指令字中的操作类型，产生相应的控制信号
    - 如果遇到停机指令，结束操作
  - 取操作数：根据指令字的地址域访存
  - 执行
  - 写回
- 指令：计算机实现某种操作（控制或运算）的一条代码
  - 由**操作码**和**操作数地址**组成，以**二进制码**的形式顺序存放在存储器中
  - 解决某个问题的一串指令的序列，形成该问题的**程序**
  - “程序控制”的概念：控制器依据存储的程序来控制全机协调地完成计算任务
- 指令系统：一台计算机通常有几十种基本指令，它们构成了该计算机的指令系统，是衡量计算机性能的重要标志
- 指令格式：指令字长 16 位：操作码 6 位，地址码 10 位
  - 例：000001 0000001000
- 存储器的功能：保存或“记忆”解决问题所需的数据和解题方法步骤
- 存储器的组成：
  - 以主存储器为例，包括存储体、各种逻辑部件及控制电路等
  - 存储体 ← 存储单元 ← 存储元件
    - 存储元件，也称存储元，每个能存储一位二进制代码 0 或 1
    - 存储单元，可存储一串二进制数，称为一个存储字(字节)，其位数称为**存储字长**
    - 存储字代表的二进制码，可以是数值、字符等，也可以表示指令
  - 存储地址：对每个存储单元进行编号，其编号称为该存储单元的地址
- 存储器的组织：
  - 存储单元按**字节**或**字**寻址
  - 程序和数据顺序存放
    - 数据段
    - 代码段
  - 读写操作以**数据总线宽度**为单位
- 主存储器的工作方式：
  - 按存储单元的地址号来实现对存储字各位的写入和读出——按地址存取方式（访存）
  - MAR (Memory Address Register)，存储器地址寄存器：
    - 存放要访问的存储单元的地址，其位数对应存储单元的个数
    - 设 MAR 为 16 位，则有  $2^{16} = 64K$  个存储单元
  - MDR (Memory Data Register)，存储器数据寄存器：
    - 存放从某个存储单元取出的或准备往某个存储单元存入的内容，其位数与存储字长相等

- 常见的输入设备：键盘、鼠标、扫描仪、模数转换器等
- 常见的输出设备：显示器、打印机、绘图仪、语言语音设备等
- I/O 设备类型繁多、速度各异、数据形式不一，因此，要实现 I/O 设备与主机之间的连接和信息交换，必须经过一个数据转换和传输的设备，称为 **I/O 接口** 或者 **I/O 适配器**
- 总线：构成计算机系统的骨架，是计算机各个组成部件间进行数据传送的公用通路
  - 从连接角度看，可以分为 **内部总线**、**系统总线** 和 **I/O 总线**
  - 从功能角度看，可以分为 **数据总线**、**地址总线** 和 **控制总线**
- 计算机体系结构的 8 种属性：
  - 数据表示：硬件能直接辨识和操作的数据类型和格式
  - 寻址方式：最小可寻址单位、寻址方式的种类、地址运算
  - 寄存器组织：操作寄存器、变址寄存器、控制寄存器及专用寄存器的定义、数量和使用规则
  - 指令系统：机器指令的操作类型、格式、指令间排序和控制机构
  - 存储系统：最小编址单位、编址方式、主存容量、最大可编址空间
  - 输入输出结构：输入输出的连接方式、处理机/存储器与输入输出设备间的数据交换方式、数据交换过程的控制
  - 中断机构：中断类型、中断级别，以及中断响应方式等
  - 信息保护：信息保护方式、硬件信息保护机制
- 机器字长：CPU 一次能处理的二进制数据的位数
  - 通常与 **CPU 的寄存器位数** 有关，也反映运算部件和数据总线的位数
    - 日常所说的 32 位、64 位机器，这里的 32 和 64 就是指机器字长
  - 影响运算速度：CPU 字长较短，要运算位数较多数据时，则需多次运算才能完成
  - 影响计算精度：字长越长，数的表示范围越大，精度越高
  - 影响硬件成本：字长越长，硬件部件的成本随之增高
    - 为协调计算精度和硬件成本间的制约关系，多数计算机采用变字长运算
- 存储容量：存储器中所有存储单元的总位数
  - 分为主存容量和辅存容量
  - 存储容量 = 存储单元个数 × 存储字长
    - 存储器的地址寄存器 MAR 的位数反映了存储单元个数
    - 存储器的数据寄存器 MDR 的位数反映了存储字长
    - 假设 MAR 为 16 位、MDR 为 32 位，则表示该存储部件有  $2^{16}=65536$  个 存储单元，总的存储容量为  $2^{16} \times 32b = 2^{21}b = 2Mb = 256KB$
- 存储容量表示：
  - 通常用字节数来描述容量大小，如 B、KB、MB、GB、TB
  - 字节：一个字节 B，表示 8 位二进制数
- 数据通路宽度：数据总线一次能并行传送的信息的位数
  - 影响计算机的有效处理速度
  - 分为 CPU 内部和 CPU 外部两种情况
    - 内部数据通路宽度一般等于机器字长，即内部数据线的位数
    - 外部数据通路宽度等于系统数据总线的位数，即 CPU 与主存、I/O 设备之间一次数据传送的信息位数，即 **数据字长**
- 字的概念：
  - 不同机器其字的位数可能不一样，但一定是字节的倍数
  - 对于某一系列的计算机来说，字的长度是固定的
- 机器字长 VS. 存储字长 VS. 数据字长：三者可以相等，也可以不等，但必须是字节的整数倍
- 存储器带宽：单位时间内从存储器读出的二进制数信息量，一般用 B/s 表示
- 衡量运算速度的普通方法：完成一次加法或乘法所需的时间

- CPI (Cycle per Instruction): 指令周期, 表示执行一条指令所需的平均时钟周期数

$$CPI = \frac{\text{执行程序所需的CPU时钟周期数}}{\text{程序包含的指令条数}}$$

- MIPS (Million Instruction Per Second): 百万条指令每秒, 即单位时间内执行的指令数

$$MIPS = \frac{\text{程序的指令条数}}{\text{程序的执行时间} \times 10^6}$$

- MFLOPS (Million Floating Point Operation Per Second): 百万次浮点操作每秒, 用来衡量机器浮点运算的性能

$$MFLOPS = \frac{\text{程序的浮点操作次数}}{\text{程序的执行时间} \times 10^6}$$

➤ 程序执行的 CPU 时间:  $T_{CPU} = \frac{CLK}{f}$  其中 CLK 为总时钟周期数, f 为时钟频率, 单位为 MHz

➤ 程序执行过程中所处理的指令数, 记为 IC。CPI = CLK / IC

➤ 时钟周期  $T_{CLK} = \frac{1}{f}$

➤ 联立上面的公式, 有  $T_{CPU} = CPU \times IC \times T_{CLK}$

➤ 响应时间 (执行时间): 从事件开始到结束之间的时间。

➤ 吞吐率(Throughput): 在单位时间内所能完成的工作量 (任务)。(多用户系统)

➤ Moore 定律: 集成电路芯片上所集成的电路的数目, 每隔 18 个月就翻一番

➤ 线延迟墙(Wire delay wall): 导线变细, 阻力变大; 导线间隔变短, 导致耦合电容, 从而影响频率

➤ 存储墙 (Memory Wall): 处理器性能与存储系统性能之间存在巨大差距

➤ I/O 墙 (I/O wall): 处理器的主频与 I/O 总线时钟频率之间存在巨大差距

➤ 阿姆达尔(Amdahl)定律: 系统加速比 =  $\frac{\text{系统性能}_{\text{改进后}}}{\text{系统性能}_{\text{改进前}}} = \frac{\text{总执行时间}_{\text{改造前}}}{\text{总执行时间}_{\text{改造后}}}$

➤  $S = \frac{1}{(1-f_e) + \frac{f_e}{S_e}}$  其中  $f_e$  为并行计算部分所占比例,  $S_e$  为并行处理结点个数

- 推论: 如果只针对整个任务的一部分进行优化, 那么所获得的加速比不大于  $1/(1-f_e)$

➤ 晶片成本 = 晶圆成本 / (每块晶圆上的晶片数 × 晶片成品率)

➤ 每块晶圆上的晶片数 =  $\frac{\text{晶圆面积}}{\text{晶片面积}} - \frac{\pi \times \text{晶圆面积}}{\sqrt{2 \times \text{晶片面积}}}$

晶片成品率 = 晶圆成品率  $\times (1 + \frac{\text{疵点密度} \times \text{晶片面积}}{\alpha})^{-\alpha}$

➤ 关于图灵机的两个重要原理:

- 相似性原理: 所有计算模型的计算能力等同

- 所有合理的、功能足够强大的计算模型可以相互模拟计算, 且使用的本质相同的并行计算时间、串行计算时间和空间

- 丘奇 - 图灵论题: 可计算性等价于图灵机的可计算性

- 对偶性原理: 在并行计算模型上, 计算的时间与空间可以互换

➤ 古斯塔夫森定律: 多处理器/多核加速比  $\leq N + (1-N) \times S$ , 其中 N 为内核数量, S 为并行应用中的串行执行时间的百分数

## 思考题：

- 冯机架构中指令和数据都存储于存储器中，系统执行时如何区分？  
计算机区分指令和数据有以下 2 种方法：
  - 通过不同的时间段来区分指令和数据，即在取指令阶段取出的为指令，在执行指令阶段）取出的即为数据。
  - 通过地址来源区分，由 PC 提供存储单元地址的取出的是指令，由指令地址码部分提供存储单元地址的取出的是操作数。
- “计算机组成”与“计算机系统结构”的关系？  
计算机体系结构讲的是计算机有哪些功能（包括指令集、数据类型、存储器寻址技术、I/O 机理等等），是抽象的；计算机组成原理讲的是计算机功能是如何实现的，是具体的。计算机组成是计算机系统结构的先修课。
- 比较 Amdahl's Law 和古斯塔夫森定律  
两个定律都计算系统加速比，区别在于阿姆达尔定律侧重于部分操作并行程序的优化，古斯塔夫森定律侧重于多核情形的并行程序优化。

## Chapter 2 指令系统

- 指令：计算机执行某种操作的命令
  - 微指令：微程序级的命令，属于硬件
  - 宏指令：若干条机器指令组成的命令，属于软件
  - 机器指令：通常所说的指令，介于微指令和宏指令之前，可完成一个独立的算术运算或逻辑运算操作
- 指令系统：一台计算机所有机器指令的集合，称为该计算机的指令系统
- 指令系统的性能，决定了计算机的基本功能，与计算机硬件密切相关，也直接关系到用户的使用需要
- 完善的指令系统的四方面要求：
  - 完备性：指令系统直接提供的指令足够使用，相关功能不需用软件实现
  - 有效性：指令级程序占据的存储空间小，执行速度快，能够高效运行
  - 规整性：对称性、匀齐性、指令格式与数据格式的一致性
  - 兼容性：向上兼容—低档机器的软件能在高档机器上运行、向前兼容
- 操作码的位数反映了机器的操作种类，即机器最多允许的指令种类数
  - 操作码包含  $n$  位的机器，最多能够有  $2^n$  种指令
- 地址码：指出指令的源操作数的地址（一个或两个）、结果的地址或者下一条指令的地址
  - 这里的“地址”，可以是主存地址、寄存器地址、I/O 设备地址等
- 三地址指令：

OP码	$A_1$	$A_2$	$A_3$
-----	-------	-------	-------

  - 最典型的指令格式
  - 操作：(A1) op (A2)  $\rightarrow$  A3
  - 若指令字长 32 位，操作码 8 位，地址字段各 8 位，则三地址指令操作数的直接寻址范围为  $2^8=256$
  - 若地址字段都为存址，则完成一条三地址指令操作，共需访问 4 次主存
    - 取指令 1 次，取两个操作数 2 次，存放结果 1 次
- 二地址指令：

OP码	$A_1$	$A_2$
-----	-------	-------

  - 将指令的操作结果保存到某个操作数的地址位置
  - 操作：(A1) op (A2)  $\rightarrow$  A1 或 A2



- 4 次或 3 次访存

➤ 一地址指令：

OP码	A <sub>1</sub>
-----	----------------

- 把一个操作数隐含在累加器 ACC 中，操作结果又放回 ACC 中
- 操作：(A<sub>1</sub>) op (ACC) → ACC
- 只需 2 次访存

➤ 零地址指令：

OP码	
-----	--

- 指令字中只有操作码，没有地址码
- 零地址指令示例：
  - 空操作 NOP
  - 停机指令 HLT
  - 函数返回指令 RET、中断返回指令 IRET

➤ 扩展操作码：将一段地址码变为操作码。如操作码位数为 4，地址码位数为 6，共 16 位，那么通过扩展操作码，能将操作码扩展为 10 位和 16 位

➤ 按操作数物理位置的指令分类：

- 存储器-存储器 (SS) 型指令：
  - 操作数都存放在主存中，指令执行过程需多次访问主存
- 寄存器-寄存器 (RR) 型指令
  - 操作数都存放在 CPU 的寄存器中，指令执行过程需使用到多个通用寄存器或个别专用寄存器
  - 执行速度相对较快
  - RISC 型指令系统中，多数指令都设计为此种类型
- 寄存器-存储器 (RS) 型指令
  - 操作数可能存放在寄存器或存储器中

➤ 指令字长：一个指令字中包含的二进制代码的位数

- 取决于操作码长度、操作数地址的长度和操作数地址的个数

➤ 指令字长与机器字长的关系：

- 指令字长等于机器字长时——**单字长指令**
  - 早期机器中，存储字长也与两者相等，一次访存可以取出一条完整指令或数据
- 指令字长等于机器字长的一半——**半字长指令**
- 指令字长等于机器字长的两倍/多倍——**双/多字长指令**
  - 为了能够提供足够的地址位以解决访问内存任意单元的寻址问题
  - 需多次内存访问才能取出一条完整指令，CPU 速度降低，占用更多存储空间

➤ 按指令字长是否可变划分：

- 等长指令结构——指令系统中的所有指令字的长度都相等
  - 可以都为单字长指令或半字长指令等
  - 结构简单，控制方便
- 变长指令结构——指令系统中的各种指令字的长度因功能而异，可以采用不同位数
  - 结构灵活，充分利用指令长度，但控制较复杂
  - 为提高指令运行速度和节省存储空间，设计指令系统时，通常尽可能把常用的指令设计为单字长或半字长格式
- 一般指令字长都取 8 的整数倍

➤ 计算机中常见的操作数类型：

- **地址**：被看做一个无符号整数
  - 很多情况下，对指令中的操作数的引用必须完成某种计算，才能确定其主存中的有效地址
- **数值**：定点数、浮点数、十进制数等
- **字符**：文本或字符串

- 普遍采用 ASCII 字符编码，以 8 位的字节来存储和传送字符
- 其他的一些字符编码，如 8 位 EBCDIC（扩展 BCD 交换码）
- **逻辑数据：**
  - 存储单元中的每一位“0”或“1”都代表真或假的布尔型值，这些 0 和 1 组合的数就被看做逻辑数
  - 能够对某个具体位进行逻辑运算

➤ 边界对齐问题：多字节的数据，在存储时需要“边界对齐”

➤ 数据不做边界对齐带来的问题：读取一个数据时，可能需要两次访存操作才能取到完整的数据，降低了 CPU 的处理速度

➤ 移位操作：

- 分算术移位（有符号）、逻辑移位（无符号）、循环移位
- 算数移位常被用来代替简单的乘法和除法运算

➤ 过程调用和返回的状态保存：

- “**调用者保存**”（caller saving）方法：如果采用调用者保存策略，那么在一个调用者调用别的过程时，必须保存**调用者所要保存的寄存器**，以备调用结束返回后，能够再次访问调用者。
- “**被调用者保存**”（callee saving）方法：如果采用被调用者保存策略，那么**被调用的过程**必须保存**它要用的寄存器**，保证不会破坏过程调用者的程序执行环境，并在过程调用结束返回时，恢复这些寄存器的内容。

➤ 陷阱：一种意外事故中断，如除 0、运算溢出等

- 一般不提供给用户直接使用，而作为隐含指令由 CPU 在出现意外时自动产生并执行
- 也有机器设置供用户使用的陷阱指令，利用它来完成系统调用和程序请求

➤ 地址：操作数或指令存放在某个**存储单元**时，该存储单元的编号，称为该操作数或指令在存储器中的地址

➤ 寻址方式：确定本条指令中的**数据地址**以及下一条将要执行的**指令地址**的方法

- 与硬件结构紧密相关，直接影响指令格式和指令性能
- 分为**指令寻址**和**数据寻址**两类
  - 前者简单，后者复杂
  - 在冯·诺依曼型计算机中，指令寻址和数据寻址交替执行

➤ 指令寻址：分为顺序寻址和跳跃寻址两种

- 顺序寻址：通过程序计数器 PC 加 1，自动形成下一条指令的地址
- 跳跃寻址：
  - 由当前指令（转移类指令）的地址码域给出下一条指令的地址
  - 跳跃后按新的指令地址再顺序寻址，直到又碰到转移类指令
  - 可以实现程序转移或构成循环程序

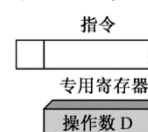
操作码	寻址特征	形式地址A
-----	------	-------

➤ 数据寻址：形成数据的有效地址的方法，称为数据的寻址方式

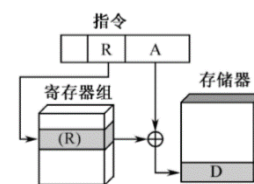
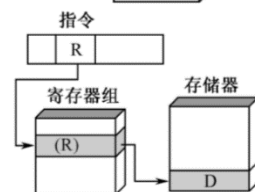
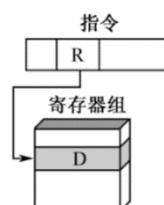
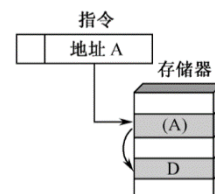
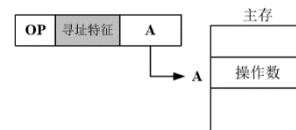
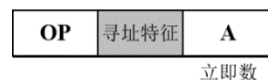
- 形式地址 A：指令的地址码字段所表示的地址，通常不代表操作数的真实地址
- 有效地址 EA：相对于形式地址而言，是操作数的真实地址，由寻址特征和形式地址共同确定
- 寻址特征字段：
  - 用于确定指令的操作数据应使用何种寻址方式
  - 间接寻址位 + 变换寻址位 + 其他寻址位
- 数据寻址过程，就是把操作数的形式地址，通过寻址特征，变换为操作数的有效地址的过程

➤ 基本的数据寻址方式：

- 隐含寻址：操作数在专用寄存器
  - 如单地址指令，第二操作数地址隐含在累加寄存器 ACC 中
- 立即寻址：操作数=A



- 指令的地址字段指出的不是操作数地址，而是操作数本身
  - 指令中的操作数又称为立即数，一般采用补码形式存放
- 取出指令就可同时获得操作数，不必访问内存
- A 的位数限制了立即数的取值范围
- 直接寻址：EA=A
  - 基本寻址方式，指令字中的形式地址 A 就是真实地址 EA
  - 优点：寻找操作数比较简单，无需专门计算操作数地址，访存一次即取得操作数
  - 缺点：A 的位数限制了操作数的寻址范围
- 间接寻址：EA=(A)→D
  - 形式地址 A 指出操作数有效地址 EA 所在的存储单元，即有效地址由形式地址间接提供
  - 优点：
    - 扩大了操作数的寻址范围
    - 便于编程
  - 缺点：需要两次或多次访存，指令执行时间变长
- 寄存器寻址：EA=R
  - 操作数不在内存中，而放在 CPU 的通用寄存器里；形式地址 A 表示的是寄存器的编号 R
  - 无须访存，减少了指令执行时间
  - 因寄存器编号较短，故可以压缩指令字，节省存储空间
- 寄存器间接寻址：EA=(R)→D
  - 形式地址 A 所表示的寄存器中，存放的不是实际操作数，而是操作数在内存中的地址
  - 需访存操作，不过相比于间接寻址，减少了一次访存
- 堆栈寻址：EA=栈顶
  - 要求计算机系统中设有堆栈才能够实现
  - 两种形式：
    - 硬堆栈，寄存器组
    - 软堆栈，主存的一部分空间
  - 可视为是一种隐含寻址，操作数地址隐含在 SP 中；也可视为寄存器间接寻址，SP 为寄存器，存放操作数有效地址
- 偏移寻址：EA=A+(R)
  - 直接寻址 + 寄存器间接寻址
    - A，内存形式地址，直接使用
    - R，某寄存器编号，间接使用
  - 根据使用的寄存器是专用或通用寄存器，可以分为隐式或显式两种
  - 常用的三种形式：相对寻址、基址寻址、变址寻址
  - 相对寻址：
    - 使用程序计数器 PC 提供基准地址，指令字中的形式地址 A 给出相对位移量即  $EA=A+(PC)$ 
      - A 称为偏移量，通常用补码表示
      - 有效地址是相对当前指令地址的一定范围内的偏移 —— 基于程序的局部性原理
    - 相对寻址方式的用处：
      - 代码模块可采用浮动地址，程序在内存中可以任意放置
      - 编程只需确定程序内部操作数与指令之间的相对距离，而无需确定操作数在主存储器中的绝对地址，这样，程序可以安排在主存的任意位置而不会影响其正确性
      - 常用于转移类指令，转移地址随 PC 值变化
  - 基址寻址 与 变址寻址：
    - 两者有效地址的形成过程极为相似，即  $EA = A + (R)$





- 使用哪个寄存器用作基址/变址寄存器，需在硬件设计时明确指定
- 基址寻址——寄存器中含有一个主存地址，指令中的形式地址 A 表示相对于该地址的偏移量
  - 基址寄存器的内容通常由操作系统或管理程序确定，在程序执行过程中不可变，可变的是形式地址 A
  - 扩大操作数的寻址范围——基址寄存器位数可设置得很长
    - ◆ 基址寄存器的位数可以大于形式地址位数，因此可以实现对主存空间的更大范围的访问
  - 用于为程序或数据分配存储空间，实现存储透明性
  - 实现段寻址
    - ◆ 将主存空间分为若干段，每段的首地址存于基址寄存器，段内偏移量由指令字中的形式地址 A 指出
- 变址寻址——指令中的形式地址 A 表示一个主存地址，寄存器中含有相对于该地址的偏移量
  - 变址寄存器的内容由用户设定，在程序执行过程中可变，但形式地址 A 的内容是不可变的
  - 常用于需要频繁修改操作数地址的处理，如数组运算、字符串操作以及循环重复等

➤ 程序的局部性原理：

- 程序的执行过程，总是趋向于使用最近使用过的数据和指令，也就是说程序执行时所使用的信息和访问的存储器地址分布不是随机的，而是相对地集中
- 这种集中包括时间和空间两个方面：
  - 程序的时间局部性——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（循环）
  - 程序的空间局部性——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（数组）

➤ “二八”定律：有 20% 的指令使用频率最大，占运行时间的 80%。也就是说，有 80% 的指令在 20% 的运行时间内才会用到

➤ RISC 指令集功能设计原则：

- 选取使用频率最高的指令，并补充一些最有用的指令
- 每条指令的功能应尽可能简单，并在一个机器周期内完成
- 所有指令长度均相同
- 只有 load 和 store 操作指令才访问存储器，其它指令操作均在寄存器之间进行

➤ MIPS 的指令格式：

R-type <i>reg-reg</i>	op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
	op(6 bits)	rs(5 bits)	rt(5 bits)	immediate(16 bits)		
I-type <i>reg-mm</i>	op(6 bits)	rs(5 bits)	rt(5 bits)	addr(16 bits)		
	op(6 bits)	rs(5 bits)	rt(5 bits)	addr(16 bits)		
J-type	op(6 bits)	addr(26 bits)				
	op(6 bits)	addr(26 bits)				

➤ MIPS 寄存器：

REGISTER	NAME	USAGE
\$0	\$zero	常量0(constant value 0)
\$1	\$at	保留给汇编器(Reserved for assembler)
\$2-\$3	\$v0-\$v1	函数调用返回值(values for results and expression evaluation)
\$4-\$7	\$a0-\$a3	函数调用参数(arguments)
\$8-\$15	\$t0-\$t7	暂时的(或随便用的)
\$16-\$23	\$s0-\$s7	保存的(或如果用，需要SAVE/RESTORE的)(saved)
\$24-\$25	\$t8-\$t9	暂时的(或随便用的)
\$28	\$gp	全局指针(Global Pointer)
\$29	\$sp	堆栈指针(Stack Pointer)
\$30	\$fp	帧指针(Frame Pointer)
\$31	\$ra	返回地址(return address)

➤ R 类型指令：

6	5	5	5	5	6
操作码	rs	rt	rd	shamt	Func

1. 寄存器—寄存器 ALU 操作：rd←rs **func** rt
2. 函数对数据的操作进行编码：加、减、...；
3. 对特殊寄存器的读/写和移动。
4. Shamt:位移量
5. Func:函数，为 Opcode 操作某个特定变体

➤ I 类型指令：

6	5	5	16
操作码	rs	rt	立即数

字节、半字、字的载入和存储；  
 $rt \leftarrow rs \text{ } \mathbf{op} \text{ 立即值。}$

➤ J 类型指令：

6	26
操作码	与 PC 相加的偏移量

**跳转，跳转并链接，从异常（exception）处自陷和返回。**

➤ MIPS 算术指令特点：

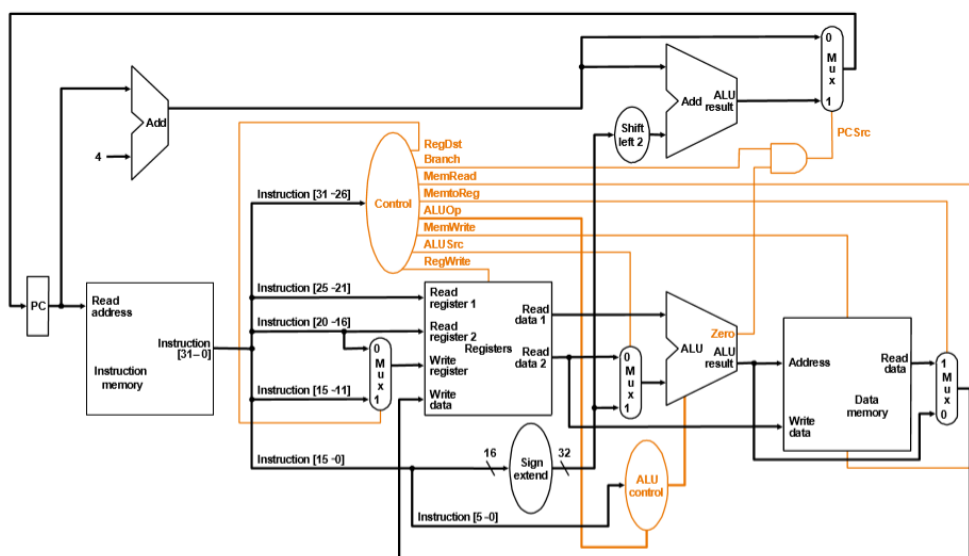
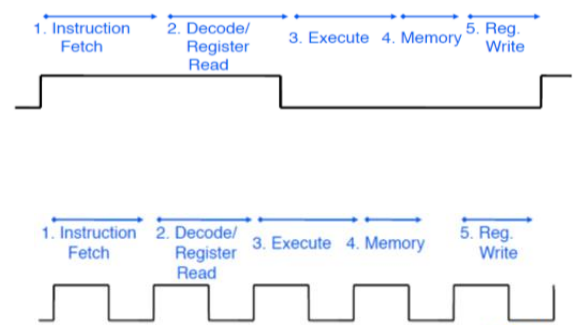
- 设计原则：simplicity favors regularity
- 操作数的次序是固定的（目标操作数在前）

## 思考题

- CPU 的 ISA 要定义哪些内容？  
基本指令集需要定义如下内容：
  - 寄存器
  - 总线大小
  - 操作
  - 操作数
  - 本文值
  - 指令格式
- Windows 系统中可执行程序的格式？  
常见的.exe 文件基本都是 PE (Portable Executable) 格式

## Chapter 3 MIPS 处理器设计

- 单周期实现：指令周期 = 1 机器周期 = 1 时钟周期
  - 指令的所有阶段都在一个很长的时钟周期内完成
  - 所需控制信号同时生成
  - 采用时钟边沿触发方式
    - 所有指令在时钟的一个边开始执行，在下一个边结束
- 多周期实现：指令周期 = n 机器周期 = n 时钟周期
  - 每个时钟周期只有一个指令级
  - 按时钟周期 (= 机器周期) 生成当前周期所需控制信号
- 单周期 CPU 的构造：



- 主要分为 PC，指令寄存器，寄存器堆，ALU 和数据存储器这几个模块，配以控制器，并连接数据通路即可
- 控制器：控制信号生成
  - op 域 (6 位) 译码产生的控制信号：8 个

- RegDst: 选择 rt 或 rd 作为写操作的目的寄存器
  - R-type 指令 (1) 与 I-type 指令二选一 (0)
- RegWrite: 寄存器写操作控制
- ALUSrc: ALU 的第二个操作数来源
  - R-type 指令 (0) 与 I-type 指令 (1)
- ALUOp: R-type 指令 (2 位)
- MemRead: 存储器读控制, load 指令
- MemWrite: 存储器写控制, store 指令
- MemtoReg: 目的寄存器数据来源
  - R-type (I 类 ALU) 指令与 load 指令二选一
- Branch: 是否分支指令 (产生 PCSrc)
- PCSrc: nPC 来源控制, 顺序与分支成功二选一
  - “是否 beq 指令 (Branch) ”& “ALU 的 Zero 状态有效”

➤ 时钟周期的确定方法

- 时钟周期由关键路径确定
- 关键路径的选择=哪种指令执行时间最长? (一般来说是 load 指令)
  - PC 时钟初始化=30ps      读内存=250ps
  - 寄存器读=150ps      寄存器写=20ps
  - ALU 计算=200ps      多个多路选择器共 25ps (4 个)
- 画出关键路径, 并计算主频

$$\begin{aligned}
 T_C &= t_{pcq-PC} + t_{mem-I} + t_{RFread} + t_{ALU} + t_{mem-D} + t_{mux} + t_{RFsetup} \\
 &= (30 + 250 + 150 + 200 + 250 + 25 + 20) \text{ ps} \\
 &= 925 \text{ ps} \\
 &= 1.08 \text{ GHz}
 \end{aligned}$$

➤ 定长单周期 or 不定长单周期?

- 程序执行时间 = 指令数 × CPI × 时钟周期时间
- 指令周期 = 机器周期 = 时钟周期
  - CPI = 1
- 设程序中 load 有 24%, store 有 12%, R-type 有 44%, beq 有 18%, jump 有 2%。假设 load, store, R, beq, jump 指令的时间分别是 8, 7, 6, 5, 2ns。
  - 平均指令执行时间 =  $8 \times 24\% + 7 \times 12\% + 6 \times 44\% + 5 \times 18\% + 2 \times 2\% = 6.3\text{ns}$
- 因此, 变长单周期实现较定长单周期实现快  $8/6.3=1.27$  倍

➤ 指令执行的多个周期:

- 共 5 个阶段 (周期)
  - 取指
  - 译码阶段、计算 beq 目标地址
  - 执行: R-type 指令执行、访存地址计算, 分支完成阶段
  - 访存: lw 读, store 和 R-type 指令完成阶段
  - 写回: lw 完成阶段
- 定长机器周期: 机器周期=时钟周期
- 不定长指令周期: 分别为 3、4、5 个机器周期
- 控制器根据机器周期标识发出控制信号





- R-type 完成：结果写回  $\text{Reg}[\text{IR}[15-11]] = \text{ALUOut}$
  - sw 完成：写入 MEM  $\text{MEM}[\text{ALUOut}] = \text{B}$
  - lw 读：  $\text{MDR} = \text{MEM}[\text{ALUOut}]$
  - 控制信号：RegWrite, RegDst, MemtoReg, lorD, MemRead, MemWrite
- lw 写阶段：
- lw 写回：  $\text{Reg}[\text{IR}[15-11]] = \text{MDR}$
  - 控制信号：RegWrite, RegDst, MemtoReg
- FSM 控制部件实现(图在实验讲义里，估计考试不会考细节的，了解一下就行了)

## 思考题

- 分析 MIPS 三种类型指令的多周期设计方案中每个周期所用到的功能部件。

阶段\指令类型	R	I	Beq	J
取指	PC, Add, Control, Mem, IR			
译码	IR, Control, regs	IR, Control, regs Sign extend		Control, PC
执行	ALU, Control	ALU, Control	ALU, Control, Branch, PC	
访存	Control	Mem, MDR, Control		
回写	Control, regs	Control, regs		

- 本书有哪些逻辑设计惯例（约定）？
- 功能部件，时钟方法（clocking methodology）
  - A、B、MDR、ALUOut 等寄存器无写控制，PC、RF、IR 有
  - 为何需要 MDR？（如果  $\text{MEM} \Rightarrow \text{RF}$ ，则 Id 可少一周期）
- 为何采用多周期？应该几个周期？
- 多周期的方式早结束的指令可以不用等待，因此可以提升性能。应该 5 个周期(load 指令)。
- 每个周期需要哪些控制信号？
- 第一周期(取指)：MemRead, IRWrite, lorD, ALUSrcA, ALUSrcB, ALUOp, PCWrite, PCSource
  - 第二周期(译码)：ALUSrcA, ALUSrcB, ALUOp
  - 第三周期(执行)：ALUSrcA, ALUSrcB, ALUOp, PCWriteCond, PCSource, PCWrite
  - 第四周期(lw 访存/其他写回)：RegWrite, RegDst, MemtoReg, lorD, MemRead, MemWrite
  - 第五周期(lw 写回)：RegWrite, RegDst, MemtoReg
- 每个周期有哪些部件空闲（做无用功）？
- 第一周期：寄存器堆、ALU
  - 第二周期：存储器
  - 第三周期：1.jump 指令：只有 PC 在工作；2.beq 指令：只有 PC 和 ALU 在工作；3.lw/sw 指令：只有 ALU 在工作；4.R 型指令：只有 ALU 在工作
  - 第四周期：1.R 型指令：寄存器堆在工作；2.sw/lw 指令：存储器在工作；
  - 第五周期：lw 指令：寄存器在工作
- PCWrite 与 PCWriteCond 何时有效？
- PCWrite 是 PC 写使能，当 PC 需要更新时有效，即取指阶段、jump 指令执行阶段时有效；PCWriteCond 是分支指令的使能，当前指令为分支指令时有效。
- 执行分支指令时，PC 执行了几次写操作？
- 其实都只执行一次写操作，只不过当分支有效时，下一次指令的 PC 地址不是+4 而是分支地址，当分

支无效时 PC 还是+4。这一次写操作都是在取指阶段执行的。

➤ 关于 beq

- 为何在 ID 计算目的地址？  
这样能减少一个周期的运算，提前确定到底下一条地址的指令是+4 还是分支。
- 在 EXE 计算地址，MEM 比较完成，ok？状态机？CPI？  
这样 beq 指令会增加一个周期，可以是可以，但是 CPI 就会增加了。
- 先比较，后计算地址，ok？

不同时进行的话相当于还是多了一个周期。由于数据通路的原因，如果条件成立，得在执行阶段才能算出地址。

➤ jump 指令在 ID 阶段完成？

译码阶段 jump 指令就可以将计算好的地址传给 PC 了，因为有一段数据通路能直接把跳转的地址传给 ALU 计算。

➤ 为何单周期中，PC 无写控制，多周期有？

单周期的时候 PC 是一个时钟周期完成一条指令的全部任务，只需要由时钟控制 PC 就可以了。但多周期的时候是多个周期完成一条指令的任务，必须得等到指令的任务完成后才能写入新的 PC，于是就有了 PC 写控制。

➤ 哪些寄存器程序员不可见？

寄存器堆里的，程序员只用给寄存器的编号就行。

## Chapter 4 MIPS 处理器设计-流水线

➤ 流水线的性质：

- 流水线中多个任务是并行处理的
- 流水线不能缩短单个任务的响应时间，但可以提高吞吐率
- 吞吐率和效率（设备利用率）成正比
- 流水线速度限制于最慢流水站的速度
- 最大加速比 = 流水站数

➤ 流水线的特点

- 流水过程由多个相关的子过程组成，这些子过程称为流水线的“级”或“段”。段的数目称为流水线的“深度”
- 每个子过程由专用的功能段实现，各功能段的时间应基本相等，通常为 1 个时钟周期（1 拍）。否则，消耗时间长的功能段将成为流水线的瓶颈，会造成流水线的阻塞或断流
- 流水线需要经过一定的填充时间才能稳定
- 流水技术适合于大量重复的时序过程
- 流水也是一种并行的手段

➤ 流水线的分类

- 单功能流水线和多功能流水线
  - 按流水线所完成的功能分类
  - 单功能流水线，是指只能完成一种固定功能的流水线
  - 多功能流水线，是指各段可以进行不同的连接，从而完成不同的功能
- 静态流水线和动态流水线
  - 按同一时间内流水段的连接方式划分
  - 静态流水线，是指在同一时间内，流水线的各段只能按同一种功能的连接方式工作
  - 动态流水线，是指在同一时间内，当某些段正在实现某种运算时，另一些段却在实现另一种

## 运算

- 部件级、处理机级及处理机间流水线

- 按流水的级别划分
- 部件级流水线，又叫运算操作流水线，是把处理机的算术逻辑部件分段，使得各种数据类型的操作能够进行流水
- 处理机级流水线，又叫指令流水线，是把解释指令的过程按照流水方式处理
- 处理机间流水线，又叫宏流水线，是由两个以上的处理机串行地对同一数据流进行处理，每个处理机完成一项任务

➤ 流水线性能分析：三项性能指标：吞吐率、加速比和效率

- 吞吐率：是衡量流水线速度的重要指标

- 吞吐率是指单位时间内流水线所完成的任务数或输出结果的数量
- 最大吞吐率：TP max 是指流水线在达到稳定状态后所得到的吞吐率
- 实际吞吐率：设流水线由 m 段组成，完成 n 个任务的吞吐率称为实际吞吐率，记作 TP
- 最大吞吐率：取决于流水线中最慢一段所需的时间，该段成为流水线的瓶颈
  - 消除瓶颈的方法：
    - 细分瓶颈段
    - 重复设置瓶颈段

- 实际吞吐率：

- 若各段时间相等(假设均为  $\Delta t_0$ )，则完成时间  $T_{流水} = m\Delta t_0 + (n-1)\Delta t_0$  (m = 深度，n = 任务数)

- $$TP = \frac{n}{T_{流水}} = \frac{n}{m\Delta t_0 + (n-1)\Delta t_0} = \frac{1}{\left(1 + \frac{m-1}{n}\right)\Delta t_0} = \frac{TP_{max}}{1 + \frac{m-1}{n}}$$

- 加速比：是指流水线速度与等功能的非流水线速度之比

- 根据定义可知，加速比  $S = T_{非流水} / T_{流水}$

- 若流水线为 m 段，每段时间均为  $\Delta t_0$ ，则  $T_{非流水} = nm\Delta t_0$ ， $T_{流水} = m\Delta t_0 + (n-1)\Delta t_0$ ， $S =$

$$\frac{mn}{m+n-1} = \frac{m}{1 + \frac{m-1}{n}}$$

- 效率：指流水线的设备利用率(部件运行时间与总时间的比率)

- 由于流水线有通过时间(填充时间)和排空时间，所以流水线的各段并非一直满负荷工作， $E < 1$

- 若各段时间相等，则各段效率也相等，即  $e_1 = e_2 = e_3 = \dots = n \Delta t / T_{流水}$

- 整个流水线效率  $E = \frac{n\Delta t_0}{T_{流水}} = \frac{n}{m+n-1} = \frac{1}{1 + \frac{m-1}{n}}$ ，当  $n \gg m$  时， $E \approx 1$

- 从时-空图上看，效率就是 n 个任务所占的时空区与 m 个段总的时空区之比

- 吞吐率、加速比和效率的关系

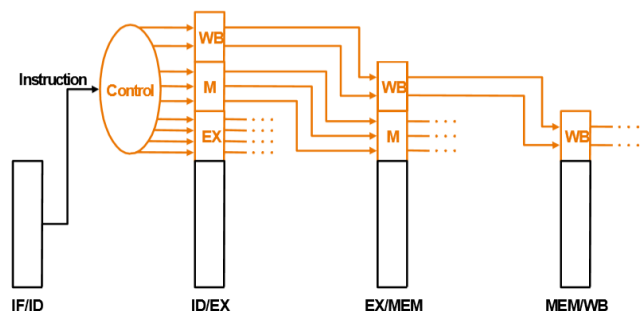
- $E = S/m$ ，效率是实际加速比 S 与最大加速比 m 之比
- $E = TP\Delta t_0$ ，当  $\Delta t_0$  不变时，流水线的效率与吞吐率呈正比。为提高效率而采取的措施，也有助于提高吞吐率

➤ 流水线控制信号

- 所有控制信号名及其功能与非流水线版相同

- 取指：读 IM，写 PC (每个周期写入一次，不需控制信号)

- 译码/寄存器读：没有控制信号
- 执行/地址计算：RegDst, ALUOp, ALUSrc
- 访存：Branch( $\Rightarrow$ PCSrc), MemRead, MemWrite
- 写回：MemtoReg, RegWrite
- 流水线段寄存器：每个周期写入一次，不需要单独的写控制



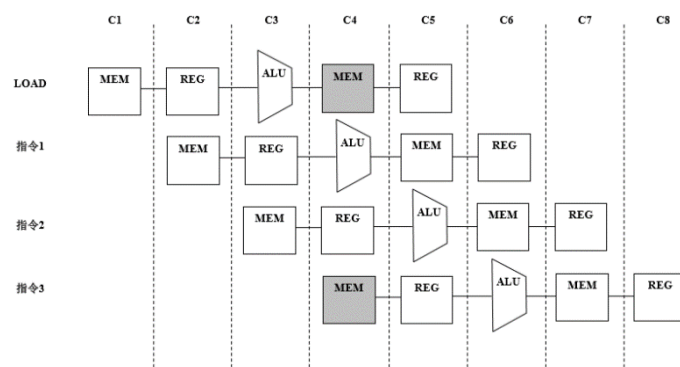
- 需要控制缓存：类似 load 指令的目的寄存器号传递

#### 流水线的“相关”：

- 结构相关：当指令在重叠执行的过程中，硬件资源满足不了指令重叠执行的要求，发生**资源冲突**时将产生结构相关
- 数据相关：当一条指令需要用到前面指令的执行结果，而这些指令均在流水线中重叠执行时，就可能引起数据相关
- 控制相关：当流水线遇到分支指令和其他会改变 PC 值的指令时，会发生控制相关

#### 结构相关：

- 典型的结构相关由访存和寄存器造成
- 解决方案 1：流水线停顿 (stall)
  - 气泡 (bubble)：流水线停顿一个周期
  - 实现：不改变 PC，重新取指
- 解决方案 2：访存结构冲突消除
  - 访存结构相关化解：哈佛结构
  - 寄存器堆读最新写入的数据时，采用前推的方式消除结构相关



#### 数据相关：

- 当指令在流水线中重叠执行时，流水线有可能改变指令读/写操作数的顺序，使之不同于它们在非流水实现时的顺序，这将导致数据相关

#### 数据相关分类：

- 写后读相关 (RAW)：j 的执行要用到 i 的计算结果，j 可能在 i 写入其计算结果之前就先对保存该结果的寄存器进行读操作
  - “读”快，“写”慢
- 写后写相关 (WAW)：j 和 i 的目的操作数一样，写入顺序错误，在目标寄存器中留下的是 i 的值而不是 j 的值
  - 后面的“写”快，前面的“写”慢
- 读后写相关 (WAR)：j 可能在 i 读取某个源寄存器的内容之前就先对该寄存器进行写操作，导致 i 读出的是错误的数据
  - 后面的“写”快，前面的“读”慢

#### 消除 RAW：

- Stall:
  - 硬件控制，动态技术（流水线互锁 Interlock）
  - 编译技术，静态技术
    - 插入 NOP
    - 调度无关指令：兼容性问题（新旧版本“延迟槽”数可能不同）
- forwarding, bypassing（定向或旁路，硬件）
  - 数据从实际存放的位置直接传到需要的位置
- 解决 LW 相关：
  - 冻结其前面的流水段：IF, ID
  - 增加两个控制信号：PCWrite 和 IF/IDWrite

- 阻止更新 PC 和 IF/ID
- 使之持续重复相同的操作
- 向 EX 流水段插入气泡
  - 将 ID/EX 的控制清零
  - 使功能部件暂停一个周期，再重新发出控制
- 控制相关：
  - 分支优化技术-降低开销
    - 延迟分支 (delayed branch)：编译调度
      - 按一定的模式向延迟槽 (delay slot) 中填入某些指令
      - 无论分支是否成功，延迟槽中的指令都将被执行
    - 延迟分支的来源
      - 使用分支前的指令填充 (从前调度)
      - 以分支目标指令填充 (从目标处调度)
      - 以分支不发生时的下一条指令填充 (从失败处调度)
  - 分支优化技术-分支预测
    - 如果能够提前知道分支是否成功，采取相应的措施，可以提高性能
    - 分支预测技术 (Prediction)
      - 静态预测：投机执行 (Speculation)
      - 动态预测：分支预测器 (Branch predictor)
- 流水线中的多发技术：通过减少流水线的停顿提升性能
  - 在一个时钟周期内流出多条指令
  - 常见的多发技术 (Multiple issue)
    - 超标量技术 (SuperScalar)
      - 指在每个时钟周期内可同时发射并执行多条指令
    - 超长指令字技术 (Very Long Instruction Word)
      - 静态指令集调度
      - VLIW 把多条能并行操作的指令组合成一条具有多个操作码字段的超长指令 (指令字长可达几百位)，由这条超长指令控制 VLIW 机中多个独立工作的功能部件，由每一个操作码字段控制一个功能部件，相当于同时执行多条指令
      - 超长指令字 (VLIW) 技术和超标量技术都是采用多条指令在多个处理部件中并行处理的体系结构，在一个时钟周期内能流出多条指令
  - 超流水线技术 (SuperPipeline)
    - 超流水线=深度流水线，在一个时钟周期内实现更细的流水段
  - SIMD 技术 (Single Instruction Multi Data)

## 思考题

- 理想流水线加速比=? IPC=?  
 $S = m(\text{流水线级数}), \quad IPC = 1$
- 为何单周期、多周期的控制信号不需要 buffer，而流水线的控制信号需要 buffer?  
 因为单周期，多周期的指令都是在一条指令完成之后才进行的，这样就不会涉及相关。而流水线的同一个周期会有多条指令进行，这样会存在指令间的相关，于是需要一个 buffer 来帮助流水线记录需要的数据并解决相关的问题。
- 哪些情形可能造成流水线 stall，有哪些解决方案?  
 三种冒险都可能造成流水线 stall。对于结构冒险，可以采用哈佛结构来解决。对于数据冒险，可以采



用前推的方式解决部分冒险问题，lw 型的部分冒险只能采用 stall 来解决。对于控制冒险，可以采用分支预测的方式解决。

➤ 影响流水线性能发挥的因素有哪些，可以采用哪些手段减小这些因素的影响？

影响因素：流水线的深度、进入流水线的指令数量、流水线的时钟周期、数据相关。

对于流水线的深度来说，根据执行操作的并行程度做适当的划分即可。

对于进入流水线的指令数量、流水线中的指令数量越多，流水线未被填充带来的影响就越小，流水线的性能就越高。

对于流水线的时钟周期来说，取决于所有阶段中耗时最长的那部分，因此需要尽可能的均匀划分每个阶段所需要的时间以减少时钟周期不均匀带来的影响。

对于数据相关来说，可以采用哈佛结构、前推、分支预测等方法来减少影响。

➤ 指令流水线中在哪个阶段会产生什么相关？

结构相关一般发生在 ID 和 WB 阶段；控制相关一般发生在 MEM 级；数据相关一般发生在 EX/MEM 级和 MEM/WB 级。

➤ 为何 MIPS 只有 I-type 指令访存？

这样可以简化数据通路，并提高性能。

➤ 典型的流水线的多发射技术有哪些？

- 超标量技术
- 超长指令字技术
- 超流水线技术
- SIMD 技术

## Chapter 5 中断与异常

➤ 中断的概念：暂停当前程序的执行，转而执行其他程序。在它们执行完成后，恢复被中断程序的执行

➤ 中断的作用：

- 异常：响应软硬件错误或故障
- I/O：响应外设的中断
  - 系统与环境交互：实时响应外部事件（I/O，人机交互）
- 并发：提高计算机的整机效率
  - 单核多任务并发：允许单处理器“同时”执行多个任务
  - 多核多任务并行：允许多处理器交互（多处理器(核)间通信）
- 服务：用户程序与 OS 间交互，Trap (陷阱)
  - 一种提供系统服务和保护的机制

➤ 中断管理：中断服务程序 ISR Interrupt Service Routines

➤ x86: Interrupt, 包含

- 外部中断：硬中断（hardware interrupt）
  - 可屏蔽中断，不可屏蔽中断 NMI(NonMaskable Interrupt)
- 内部中断：软中断（software interrupt）
  - 程序异常，系统调用 INT n，指令断点（int 3 调试）

➤ RISC: Exceptions, 包含

- 外部事件 External events(interrupts): I/O 中断
- 异常 Exceptions: 软（硬）件故障
- 陷阱 Traps: Syscall, 断点 break, 自陷 TEQ

➤ 中断的行为：

- 过程调用
- 对 CPU 控制的转移（当前程序=>中断服务程序）
- 对计算资源的并发使用
- 中断/异常发生的时机
  - I/O 中断 Interrupts（随时发生，延迟响应）
  - 异常 Exceptions（随时发生，随时处理）
  - 陷阱 Traps（专用指令，特殊处理）
- 引起中断的因素：
  - 人为设置的中断，如转管指令（与内存、IO 交互）
  - 程序异常：溢出、操作码不能识别、除法非法
  - 硬件故障
  - I/O 设备发起的中断请求
  - 外部事件：用键盘中断现行程序
- 中断请求标记 INTR：
  - 一个请求源对应一个 INTR 中断请求标记触发器
  - 多个 INTR 组成中断请求标记寄存器
- 中断判优逻辑
  - 软件实现（程序查询）：A、B、C 优先级按 降序 排列
  - 硬件实现（排队器）
    - 分散在 **各个中断源的接口电路** 中的链式排队器
    - 集中在 CPU 内的链式排队器
- 中断响应：
  - 响应中断的条件：CPU 允许中断触发器  $EINT = 1$
  - 响应中断的时间：指令执行周期结束时刻由 CPU 发查询信号
  - 中断响应-中断隐指令：
    - 中断周期完成的主要操作，通过中断隐指令完成
    - 1、保护程序断点：断点存于特定地址（如 0 号地址）内，或者在堆栈
    - 2、寻找服务程序入口地址：
      - 向量地址->PC（硬件向量法）
      - 中断识别程序入口地址 M->PC（软件查询法）
    - 3、硬件关中断
      - $EINT$  允许中断触发器  $\rightarrow 0$
      - $INT$  中断标记触发器  $\rightarrow 1$
- 中断响应的时机与条件
  - 中断发生与 CPU 响应时间
    - 外部中断（I/O）：异步（延迟响应），指令周期结束
    - 内部中断（陷阱和异常）：同步（“马上”响应）
- 中断周期(Instruction Cycle with Interrupts)
  - 指令周期结束
  - CPU 与中断控制器通信，响应外部事件(INTR,INTA)
    - 是否有中断请求
    - 识别中断源，获得中断向量
  - 动作：存 PC 和 PSW(Program Status Word, 程序状态字)，关中断，转 ISR
- 保护和恢复现场
  - 保护现场：
    - 断点  $\rightarrow$  中断隐指令完成

- 寄存器 内容→中断服务程序 ISR 完成

- 恢复现场：讲保护现场入栈的数据出栈

#### ➤ 中断服务程序入口地址的寻找

- 硬件向量法
- 软件查询法

#### ➤ 中断屏蔽技术

- 多重中断的概念(中断嵌套)

- 实现多重中断的条件：

- 提前设置开中断

- 中断隐指令响应，存断点之后关中断，中断服务 **结束之前** 开中断→中断隐指令响应，存断点之后关中断，中断服务 **开始之后** 开中断

- 优先级别高的中断源，有权中断，优先级别低的中断源

- 屏蔽技术：

- 屏蔽触发器的作用-动态优先级设定

- MASK = 0 (未被屏蔽)，INTR 被置“1”
- MASK<sub>i</sub> = 1 (被屏蔽)，INTP<sub>i</sub> = 0 (不能被排队选中)

- 屏蔽字

- 16 个中断源 1, 2, 3...16 按**降序**排列，每个对应 16 位屏蔽字

- 在 ISR 中设置屏蔽字，屏蔽对应中断源

- 新屏蔽字的设置-ISR

- 屏蔽技术可改变处理优先等级

- 响应优先级：不可改变
- 处理优先级：可改变（通过重新设置屏蔽字）
- 响应优先级 A→B→C→D 降序排列
- 处理优先级 A→D→C→B 降序排列

- 屏蔽技术的其他作用：可以**人为地屏蔽**某个中断源的请求，便于程序控制

#### ➤ MIPS 中异常的处理方式

- 由于异常是执行指令时同步发生，因此，在造成异常的指令之前执行的指令，均是有效的
- 由于 MIPS 的高度流水体系结构，在引发异常的指令执行时，后面一条指令已经完成了读取和译码的预备工作
- 当异常产生时，这些预备工作便被废弃。CPU 从异常中返回时，再重新做读取和译码的工作

#### ➤ MIPS 中的异常分类

- 外部事件
- 操作系统调用
- 算数溢出
- 非法指令
- 其他硬件错误

#### ➤ MIPS 的异常处理：

- 以两种异常作为示例：非法指令和算术溢出
- 异常处理的主要工作：
  - 断点保存：将异常执行指令的地址保存在 **EPC 寄存器** 中
  - 异常识别：根据**状态寄存器 cause** 中的异常原因分别处理异常
    - 非法指令：转到特定服务程序

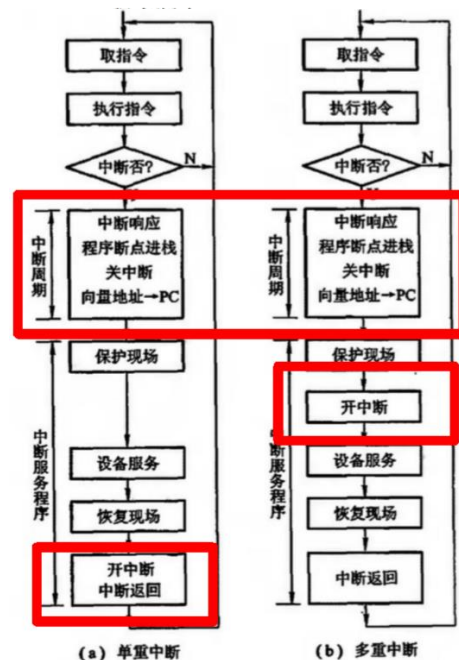
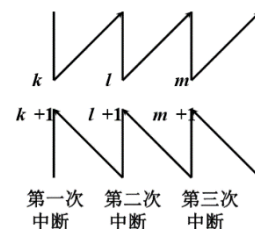


图 5.43 单重中断和多重中断服务程序流程

中断源	原屏蔽字	新屏蔽字
A	1 1 1 1	1 1 1 1
B	0 1 1 1	0 1 0 0
C	0 0 1 1	0 1 1 0
D	0 0 0 1	0 1 1 1

- 溢出：对溢出进行响应
- 跳转异常服务程序 (PC)：停止程序的执行并报告错误等
  - 未定义指令异常处理在 8000 0000hex
  - 算数溢出异常处理 8000 0180hex
- 服务程序的入口
- 多周期支持异常
  - 完成功能（非法指令、算术溢出）
    - PC 赋值(根据异常原因跳转到异常处理程序)
    - 出错 PC 保存 (PC-4 保存到寄存器)
    - 出错原因保存 (异常原因保存到寄存器)
  - 数据通路
    - PC、EPC、Cause
  - 控制信号
    - PC 写 (多路选择器)、EPC 计算写入、Cause 写入
  - RTL 代码
    - 非法指令实现、算术溢出实现
  - 状态机
    - 加入 2 个状态，在发生异常之后的一个状态加
- 中断和异常操作语义特点（非流水线）
  - 顺序语义
    - 之前的指令都已执行完成
      - 已经提交其状态
    - 之后的指令还没有启动
      - 没有改变任何机器状态
  - 断点精确：= PC/nPC
    - 外部中断：异步响应
    - 指令异常：同步响应
  - 现场简明
    - Cause
    - EPC
- 流水线中的异常与中断
  - 多个流水段，多条指令，多种异常并发
    - 异常：访存-地址错（缺页越界），译码-非法指令，ALU 溢出
    - 中断
- 流水线异常处理的挑战：“顺序执行”只是一种逻辑关系：断点和状态难以确定
  - 后续指令在产生异常指令完成之前改变了系统的部分状态
  - 异常发生顺序与指令执行顺序不一定相同
  - 转移指令和分支预测给异常处理带来了麻烦
- 非精确处理异常的方法
  - 方案 1：允许已进入流水线中的指令执行完再转去执行中断处理
    - 断点：无论在第 i 条指令的哪一流水段上发生异常，都不再允许后继指令进入流水线，断点为最后进入流水线的那条指令的地址(非精确)
      - 非精确（可能不是“当前指令”）
      - 可变（不同段发生异常，EPC 增量不同）
    - 优点：硬件比较简单
  - 方案 2：将异常指令的后续指令排空

- 处理：将异常视为一种控制相关，工作如下：
  - 暂停指令流中导致异常的指令
  - 执行完异常指令之前的所有指令
  - 清除异常指令之后的所有指令
  - 记录异常原因
  - 保存断点
  - 转异常处理程序
- 不允许后续指令继续执行
- 非精确异常的问题
  - 缺点：
    - 异常响应时间较长
    - 如果等进入流水线的指令执行结束，可能会导致程序出错
    - 程序调试不便：程序员在第 i 条指令设置断点，但程序不能准确中断在所设置的断点处。
  - 如何处理多个异常？
  - 如何非精确处理？
- MIPS 实现精确异常
  - 实现“精确”开销大
    - 要保证异常指令“可重启”
      - 安全停止流水线，并完整保存**当前状态**
      - 需要大量后援寄存器保存流水线中各指令的现场
        - 包括 RegFile、PSW、流水段寄存器（含各段的控制寄存器）!
  - MIPS 采用**提交点**技术实现“精确”异常
    - 提交点：MEM 段
      - 多个异常：先发生的异常并不立即处理，只是被标记
        - EXCn 寄存器：保存异常类型
        - 流水线中最深的指令引起的异常最优先
- 各段产生的异常及处理：MIPS 的策略
  - 保持流水线的异常标记直到**提交点**（MEM 段）
  - 如果提交点有异常，则更新 cause 和 EPC，清除所有流水段，新 PC 值到 fetch 段
  - 早期流水段的异常抑制后来的异常（flush IF/ID/EX）
  - 提交点处引入**中断处理**

## 思考题

- 中断周期要完成哪些微操作？
 

操作：存 PC 和 PSW(Program Status Word, 程序状态字)，关中断，转 ISR
- 多周期状态机中，出现溢出的指令是否将错误结果写回？
 

并不会，出现溢出后状态将变为异常状态，并对 PC 赋值（根据异常原因跳转到异常处理程序），保存出错 PC，保存出错原因，同时写入 PC、EPC 和 Cause。
- 多周期中状态机中，如何响应中断？
 

断点保存：将异常执行指令的地址保存在 EPC 寄存器中

异常识别：根据状态寄存器 cause 中的异常原因分别处理异常

跳转异常服务程序（PC）：停止程序的执行并报告错误等
- 指令顺序执行，中断“精确”；指令流水执行，中断“精确”或“非精确”可选



说的是对的，PPT 上就是这么讲的。

➤ 流水线是否存在“中断周期”？

存在，流水线也会响应中断请求进入中断周期的。

➤ 精确中断，为何提交点是 M 段？

这样能确保上一条指令完成，并且后续指令都没有写入新的数据，这样方便清空后续指令，当中断处理完毕后，重新执行提交点的指令即可。

➤ EPC 和 cause 应该在哪个段？异常检测电路？

EPC 和 Cause 都在 MEM 段。异常检测电路应该也在 MEM 段，但我没找到证据。

➤ mips 异常返回指令 eret 如何实现？

ERET 指令从异常中返回到用户态，返回时需要恢复现场，即恢复提交点的指令以及 PC 值等

➤ 异常与中断同时发生，优先级？

根据中断判优逻辑来判断吧，但一般来说中断发生时问题会更严重些，比如突然断电之类的，所以觉得中断优先级会更高一些吧。

不对，好像异常优先级高一点，因为是随时发生随时处理，谁能确定一下 www

➤ （分支）延迟槽中的指令发生异常，EPC = ？

EPC 指向分支指令（百度的）

➤ 比较中断、异常、过程调用

同步异步：中断：异步；异常：同步；系统调用：异步或同步

其他的有点复杂，麻烦帮忙整理下

## Chapter 6 存储系统

➤ 存储器：计算机系统中的记忆设备，用于存放程序和数据

➤ 存储器的分类：

- 按存储介质分类：半导体存储器、磁存储器和光盘存储器等
- 按存取方式分类：随机存储器(RAM)和顺序存储器
- 按存储内容可变性分类：只读存储器(ROM)和读写存储器
- 按信息易失性分类：易失性存储器和非易失性存储器
- 按在计算机系统中的作用分类：
  - 高速缓冲存储器
  - 主存储器
  - 辅助存储器

➤ 存储器的层次结构

- 多级存储器体系结构
  - 高速缓冲存储器 cache
    - 高速小容量半导体存储器，用于提高计算机处理速度
  - 主存储器
    - 存放计算机运行期间的大量程序和数据
  - 外存储器
    - 大容量辅助存储器满足计算机的存储需求

- 缓存-主存层次和主存-辅存层次

#### ➤ 主存储器概述

- 主存的基本组成

- 驱动器、地址译码器、读写电路都制  
作在存储芯片中
- MAR, MDR 在 CPU 中

- 字寻址与字节寻址

- 字存储单元——存放一个机器字的存储单元
- 字节存储单元——存放一个字节存储单元

- 字地址——字存储单元对应的单元地址
- 字节地址——字节存储单元对应的单元地址

- 字寻址计算机——计算机中可编址的最小单位  
是字存储单元

字节寻址计算机——计算机中可编址的最小单位是字节存储单元

- 主存储器的技术指标

- 存储容量

- 一个存储器中存储单元的总数，一般用字数(W)或字节数(B)表示

- 存取时间

- 存储器访问时间，指一次读操作命令发出到该操作完成，将数据送到数据总线上所经历的时间
- 写操作时间通常等于读操作时间

- 存储周期

- 连续两次读写操作所需的最小间隔时间
- 存储周期一般大于存取时间

- 存储器带宽

- 单位时间内存储器存取的信息量，用字/秒、字节/秒或位/秒表示
- 衡量数据传输速率的重要指标，决定了以存储器为中心的计算机系统获得信息的速度，是改善机器性能瓶颈的一个关键因素

- 提高存储器带宽的措施：

- 缩短存取时间、存储周期
- 增加存储字长，使每个存取周期可读写更多二进制位数
- 增加存储体

#### ➤ 存储芯片简介

- 半导体存储芯片的基本结构

- 采用超大规模集成电路制造工艺，在一个芯片内集成具有记忆功能的**存储矩阵**、**译码驱动电路**和**读/写电路**等

- 译码驱动

- 把地址总线送来的**地址信号**翻译成对应**存储单元的选择信号**

- 读/写电路

- 包括读出放大器和写入电路，用于完成读/写操作

- 存储芯片的连线

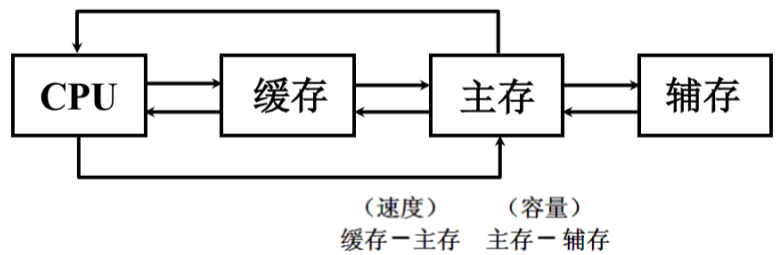
- 存储芯片通过地址总线、数据总线和控制总线与外部连接

- 地址线

- **单向输入**，其位数与芯片存储单元的数量有关

- 数据线

- **双向输入/输出**，其位数与芯片可一次读出或写入的数据位数有关



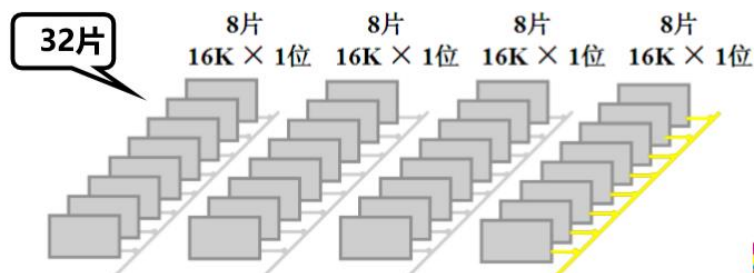
高位字节 地址为字地址      低位字节 地址为字地址

字地址	字节地址				字地址	字节地址	
0	0	1	2	3	0	1	0
4	4	5	6	7	2	3	2
8	8	9	10	11	4	5	4

- 地址线和数据线的位宽，共同反映了存储芯片的容量
- 例如，地址线 10 根，数据线 4 根，则表示存储芯片的存储单元数为  $2^{10}$ ，每个单元的数据宽度为 4 位，则其芯片容量为  $2^{10} \times 4b = 4Kb$
- 控制线：主要有读/写控制线和片选线两种
  - 读/写控制线：决定芯片进行读/写操作
    - 可以读、写分开两根线，也可以合用一根线
  - 片选线：用来在存储矩阵中选择存储芯片，可能有多根线
    - 存储芯片片选线的作用：
      - 用  $16K \times 1$  位的存储芯片组成  $64K \times 8$  位的存储器
      - 进行一次读/写时，将同时选中 8 片存储芯片，从每个存储芯片的同一地址单元取出/写入 1 位数据，组成一个 8 位的数据

#### ➤ SRAM 随机存储器

- 基本的静态存储元阵列
  - 用锁存器作为存储元—SRAM 的特征
  - 易失性存储：不断电则持久保存信息
  - 信号线
    - 地址线  $A_0 \sim A_5$
    - 数据线  $I/O_0 \sim I/O_3$
    - 控制线  $R/\bar{W}$ ，读写不会同时发生
    - 64 条地址选择线，打开每个存储元的输入与非门
- 基本的 SRAM 逻辑结构
  - 重合法双译码，便于扩展组织更大的存储容量
- SRAM 的读写周期时序
  - 读周期
    - 读出时间  $t_{AQ}$
    - 读周期时间  $t_{RC}$
  - 写周期
    - 写入时间  $t_{WD}$
    - 维持时间  $t_{\bar{M}}$
    - 写周期时间  $t_{WC}$
  - 存取周期
    - 读周期  $t_{RC} =$  写周期  $t_{WC}$
  - CS，片选信号;OE，读出使能信号;WE，读写信号



#### ➤ DRAM 随机存储器

- DRAM 芯片的逻辑结构
  - 由 MOS 晶体管（开关）和电容（信息）组成的记忆电路。电容信息存 1~2ms
  - 增加了行地址锁存器和列地址锁存器
  - 增加了刷新计数器和相应的控制电路
- DRAM 的读写周期
  - 读、写周期的界定——从行选信号  $\bar{RAS}$  的下降沿开始，到下一个  $\bar{RAS}$  信号的下降沿为止
- DRAM 的刷新
  - 过程实质：将原存储信息读出，由刷新放大器形成原信息并重新写入
  - 刷新机制：在一定时间内，使用专门的刷新电路，对 DRAM 的全部存储元按行进行一次刷新。刷新间隔一般取 2ms，即刷新周期
  - 刷新方式：集中刷新、分散刷新、异步刷新

- 集中刷新:
  - DRAM 的所有行在每一个刷新周期都被刷新, 此时必须停止所有的读/写操作
  - 以  $128 \times 128$  矩阵的存储芯片为例, 存取周期  $0.5\mu s$ ; “死区”:  $0.5\mu s \times 128 = 64\mu s$ ; “死时间率”:  $128/4000 = 3.2\%$  ( $128$  个周期, 共  $2ms/0.5\mu s = 4000$  个周期)
- 分散刷新
  - 对每一行的刷新插入在正常的读/写周期中完成
  - 不停止读/写操作, 但存取周期变长
  - 以  $128 \times 128$  矩阵的存储芯片为例,  $t_C = t_M + t_R$ , 无“死区”, 其中  $t_M$  为读写时间,  $t_R$  为刷新时间, 存取周期为  $0.5\mu s + 0.5\mu s = 1\mu s$

➤ 只读存储器 ROM (Read-Only Memory):

- 存储信息必须在工作前写入; 在工作的时候只能读出, 不能写入
- 类型: 掩膜 ROM; 可编程 ROM (PROM、EPROM、E<sup>2</sup>PROM)
- 掩膜 ROM
  - 存储内容固定, 由生产厂家提供
- 可编程 ROM
  - PROM: 实现一次性编程的 ROM
    - 熔丝断, 为 0; 熔丝不断, 为 1
  - EPROM: 可擦除可编程存储器
    - 可以对存储的信息作任意次改写
    - 擦写方式:
      - 紫外线照射: 时间相对较长, 全局性擦写
      - 电气方法 (E<sup>2</sup>PROM): 电擦除, 支持全部或局部擦写

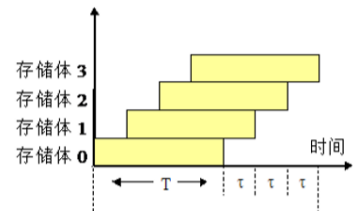
➤ 存储器与 CPU 的连接

- 存储容量的扩展
  - 单片存储芯片的容量有限, 需要将若干存储芯片连在一起组成具有足够容量的存储器
  - 位扩展: 增加存储器的横向容量
  - 字扩展: 增加存储器的纵向容量
- 位扩展:
  - 增加存储字长
  - 多个存储芯片的三组信号线关系
    - 地址线和控制线公用
    - 数据线单独分开
  - 2 片  $1K \times 4$  位的芯片组成  $1K \times 8$  位的存储器
- 字扩展:
  - 增加存储单元的数量
  - 多个存储芯片的三组信号线关系
    - 地址线和数据线公用
    - 读写控制线公用
    - 片选使能控制线独立, 通过地址的高位字段进行译码决定
  - 2 片  $1K \times 8$  位的芯片组成  $2K \times 8$  位的存储器
- 字、位扩展
  - 8 片  $1K \times 4$  位的芯片组成  $4K \times 8$  位的存储器; 12 根地址线, 8 根数据线
- 存储芯片与 CPU 芯片相连, 主要注意三种信号线的连接
- 地址线的连接
  - CPU 的地址线一般比存储芯片的地址线多
  - CPU 地址线的低位线与存储芯片的地址线相连

- CPU 地址线的高位线或用于存储扩展，或用于片选等其他用途
- 数据线的连接
  - 存储芯片的数据线应与 CPU 的数据线数量相同，若存储芯片数据线不够，需对其进行位扩展
- 读/写命令控制线的连接
  - CPU 的读/写命令线一般可以直接连到存储芯片的读/写控制端，通常高电平为读，低电平为写
  - 若读/写命令线分开，则分别连到对应的存储芯片控制端
- 片选控制线的连接
  - CPU 与存储芯片正确工作的关键
  - 片选有效信号与 CPU 的**访存控制信号** $\overline{MREQ}$ （低电平有效）有关，还与**地址线（一般为高位线）**有关
  - 通常需要用一些逻辑电路来产生片选信号，如**译码器**和**门电路**
- 合理选择存储芯片
  - 存储芯片类型（RAM 或 ROM）和数量的选择
  - ROM：通常用于存放系统程序、标准函数库和各类常数等
  - RAM：用于用户程序使用，为用户编程而设置
  - 考虑芯片数量时，应尽量使连线简单方便
  - 其他考虑：如时序配合、速度、负载等
- 提高访存速度
  - 主存的存取速度已成为计算机系统的性能瓶颈
  - CPU 不断增强，运算速度提升远快于存取速度提升
  - 结构冲突
    - 访存冲突：指令预取与数据读写
    - 总线占用：CPU 和 I/O 争抢访问主存→减少访存
  - Memory Wall
    - 处理器速度增长对系统性能的贡献将被 DRAM 性能所屏蔽
    - 提高访存速度的措施
      - 采用双端口存储器
      - 采用高速主存（SDRAM/CDRAM 等）、新型存储部件
      - 并行访问存储器：采用交叉访问：用低带宽器件构成高带宽存储总线 RAS(行地址信号)/CAS(列地址信号)轮流送给各个 bank，循环输出各个字
        - 单体多字系统
        - 多体并行系统（交叉存储器）
      - 采用层次化存储系统结构
        - 优化：隐藏访存延迟
          - 减少访存频率，减少 Cache miss
          - 会增加失效情况下的延迟
- 双端口存储器
  - 一个存储器具有两组相互独立的读写控制电路
- 单体多字系统
  - 系统中只有一个存储体
    - 由单个存储芯片或多个存储芯片构成的一个独立存储器
    - 多字：如在一个周期内，从**同一地址开始**顺序读出 4 条指令字，再逐条将指令送至 CPU 执行
- 多体并行系统
  - 系统含多个存储体
    - 每个存储体都有自己的读写线路、地址寄存器和数据寄存器



- 两种编址方式
  - 高位交叉编址：扩容，不同应用空间→顺序存储
  - 低位交叉编址：多体交叉访问→交叉存储
- 高位交叉编址-顺序存储
- 低位交叉编址-交叉存储
  - M 个模块按一定的顺序轮流启动各自的访问周期
  - 启动两个相邻模块的最小时间间隔等于单模块访问周期  $1/M$ （带宽提高 M 倍）
- 多体并行系统—高位交叉编址
  - 存储体选则
    - 高位地址
  - 顺序编址
  - 方便扩容
    - 不同内存插槽对应不同存储体
  - 支持并行访问
    - 多任务、多处理器系统，多个任务并发执行
    - 某体用于程序执行，某体用于 I/O
- 多体并行系统—低位交叉编址
  - 存储体选择
    - 低位地址
  - 交叉编址
  - 访问模式
    - 并行访问
      - 同时输出多个字
    - 交叉访问
      - 提高带宽
  - 地址依次送各个存储体，交叉访问
    - 效果：两个连续地址字的读取之间不必插入等待状态
  - 数据总线顺序输出（带宽提高 M 倍）
  - 总时间  $= T + (m-1) \times \tau$



#### ➤ 容错的解决方案：冗余

- 信息冗余
  - 海明码、CRC 码、奇偶校验码
- 时间冗余
  - 回滚
- 空间冗余
  - 复用
  - 磁盘冗余阵列

#### ➤ 奇偶编码校验

- 一个编码系统中任意两个合法编码（码字）之间不同的二进数位（bit）数叫这两个码字的码距,也称为汉明距离，用 d 表示
  - 例如码字 10010 和 01110，有 3 个位置的码元不同，所以  $d=3$
- 整个编码系统中任意两个码字的**最小距离**就是该编码系统的码距。任何一种编码是否具有**检测能力**或**纠错能力**，都与编码的最小距离有关
  - 在一个码组内为了**检测** D 个误码，需要最小码距  $L \geq D+1$
  - 在一个码组内为了**纠正** C 个误码，需要最小码距  $L \geq 2C+1$
- $L-1 \geq D+C$  且  $D \geq C$

- 即编码最小距离  $L$  越大, 则其检测错误的位数  $D$  也越大, 纠正错误位数  $C$  也越大, 且纠错能力恒小于或等于检测能力
  - 例如,  $L = 2$ , 则  $D = 1$ ,  $C = 0$ 。码距=2 才能检测 1 位错
  - 例如,  $L = 3$ , 则  $D = 2$ ,  $C = 0$ ; 或  $D = 1$ ,  $C = 1$ 。码距=3 才能检测 2 位错, 或者检测 1 位错, 纠错 1 位

#### ● 应用

- 数据通信: 奇偶校验 (串行), CRC (网络)
- 硬盘: CRC
- 内存: ECC (错误检查和纠正) 校验
- 在被传送的  $n$  位代码( $b_{n-1}b_{n-2}\dots b_1b_0$ )上增加一位校验位  $P$  (Parity), 将原数据和得到的奇 (偶) 校验位一起进行 存取或传送(即传送  $Pb_{n-1}b_{n-2}\dots b_1b_0$ )
  - 奇校验: 使“1”的个数为奇数
    - 0000 0000  $\rightarrow$  0000 0000 1
    - 0000 0001  $\rightarrow$  0000 0001 0
  - 偶校验: 使“1”的个数为偶数
    - 0000 0000  $\rightarrow$  0000 0000 0
    - 0000 0001  $\rightarrow$  0000 0001 1

#### ➤ 能够纠错一位的海明码

- 海明码需要几位校验码?
- 设有  $k$  位数据,  $r$  位校验位,  $r$  位校验位有  $2^r$  个组合
  - 若用 0 表示无差错, 则剩余  $2^r - 1$  个值表示有差错, 并指出错在第几位
- 由于差错可能发生在  $k$  个数据位中或  $r$  个校验位中, 因此有:  $2^r - 1 \geq r + k$
- 校验位和数据位是如何排列的?
  - 校验位排列在  $2^i - 1$  ( $i = 0, 1, 2, \dots$ ) 的位置上
  - 例 1: 有一个 4 位数为  $D_4D_3D_2D_1$ , 需要 3 位校验码  $P_3P_2P_1$ , 由此生成一个海明码

7	6	5	4	3	2	1	
$D_4$	$D_3$	$D_2$	$P_3$	$D_1$	$P_2$	$P_1$	
			$2^2$		$2^1$	$2^0$	

- 例 2: 有一字节的信息需生成海明码
 

12	11	10	9	8	7	6	5	4	3	2	1	
$D_8$	$D_7$	$D_6$	$D_5$	$P_4$	$D_4$	$D_3$	$D_2$	$P_3$	$D_1$	$P_2$	$P_1$	
				8				4		2	1	

- 海明码的校验位  $P_i$  和数值位  $D_i$  的关系?
  - 设  $k$  位数据,  $r$  位校验码, 把  $k+r=m$  个数据记为  $H_mH_{m-1}\dots H_2H_1$  (海明码), 每个校验位  $p_i$  在海明码中被分配在  $2^i - 1$  位置上
  - $H_i$  由多个校验位校验: 每个海明码的位号要等于参与校验它的几个检验位的位号之和
  - 分解  $i = 2^{j_1} + 2^{j_2} + \dots + 2^{j_x}$  ( $j_1 \neq j_2 \neq \dots \neq j_x$ )
  - 得:  $H_3$  由  $P_1$  和  $P_2$  校验,  $H_5$  由  $P_1$  和  $P_4$  校验,  $H_6$  由  $P_2$  和  $P_4$  校验,  $H_7$  由  $P_1$ 、 $P_2$  和  $P_4$  校验
  - 即:  $P_i$  参与第  $j_1$ 、 $j_2$ 、 $\dots$ 、 $j_x$  个校验位的计算( $P_1$  参与  $H_3$ 、 $H_5$ 、 $H_7$ 、 $\dots$ )
- 海明码的纠错原理
  - 海明码的接收端的公式:
    - $S_3 = P_3 \oplus D_4 \oplus D_3 \oplus D_2$
    - $S_2 = P_2 \oplus D_4 \oplus D_3 \oplus D_1$
    - $S_1 = P_1 \oplus D_4 \oplus D_2 \oplus D_1$
    - 假定 海明码 1010101 在传送中变成了 1000101

- $S3 = P3 \oplus D4 \oplus D3 \oplus D2 = 0 \oplus 1 \oplus 0 \oplus 0 = 1$
- $S2 = P2 \oplus D4 \oplus D3 \oplus D1 = 0 \oplus 1 \oplus 0 \oplus 1 = 0$
- $S1 = P1 \oplus D4 \oplus D2 \oplus D1 = 1 \oplus 1 \oplus 0 \oplus 1 = 1$
- 因此,由  $S3S2S1 = 101$ ,指出第 5 位错,应由 0 变 1 (因为  $101=5$ )

#### ➤ CRC 循环冗余校验码

- 基于模 2 运算: 不考虑进位和借位
  - 模 2 加减运算: 异或 (相同为“0”, 不同为“1”)
  - 模 2 乘: 按模 2 加求部分积之和
  - 模 2 除: 按模 2 减求部分余数

#### ● 模 2 运算举例

- $0 \pm 1 = 1, 0 \pm 0 = 0, 1 \pm 0 = 1, 1 \pm 1 = 0$

#### ● CRC 校验步骤

- 将  $n$  位数据  $D_{n-1}, \dots, D_0$  用  $n-1$  次多项式  $M(x)$  表示, 即  $M(x) = D_{n-1}x_{n-1} + D_{n-2}x_{n-2} + \dots + D_1x_1 + D_0x_0$
- 将  $M(x)$  左移  $k$  位 (补 0), 即得:  $M(x) * x_k (k = G(x) \text{ 位数} - 1)$
- 将  $M(x) * x_k$  除以  $k+1$  位的生成多项式  $G(x)$ , 余数即为  $k$  位的 CRC 校验位
- 将 CRC 校验位拼装在  $D_{n-1}, \dots, D_0$  之后, 成为  $n+k$  位数据, 也称  $(n+k, n)$  码

- 例:有效信息为 1100,生成多项式  $G(x)=1011$ ,将其编成 CRC 码

- $M(x) = x^3 + x^2 = 1100$
- $M(x) x^3 = x^6 + x^5 = 1100000$
- $G(x) = x^3 + x + 1 = 1011$
- $\frac{M(x) x^3}{G(x)} = \frac{1100000}{1011} = 1110 + \frac{010}{1011}$
- $M(x) x^3 + R(x) = 1100000 + 010 = 1100010$
- 编好的循环校验码称为  $(7,4)$  码,即  $n+k=7, n=4$

#### ● CRC 的译码与纠错

- 将收到的  $n+k$  位 CRC 码用约定的生成多项式  $G(x)$  去除
  - 正确, 则余数 = 0
  - 如果某一位出错,则余数不为 0。不同位出错, 余数不同
- 余数和出错位之间的对应关系不变
  - 与待测码无关, 与生成多项式有关

#### ● 余数循环

- 如果对余数补 0, 除以  $G(x)$ , 得下一余数
- 继续除下去, 各次余数将按右表顺序循环
- 特定生成多项式的余数模式固定

#### ● CRC 码的纠错方法 - 循环移位法

- 将 CRC 码进行左循环移位, 至出错位被移至最高位
  - 余数添 0 继续除法, 当余数为 101 时, 出错位被移到最高位
- 对最高位取反, 纠错
- 继续循环移位, 直至循环一周
  - 继续余数除法, 直至余数变成第一次的余数

#### ● 例 A5 出错 (生成多项式 1011)

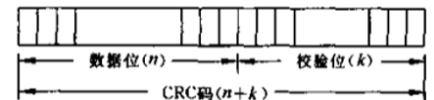
#### ● 生成多项式 $G(x)$ 应能满足下列要求:

- 任何一位发生错误都应使余数不为 0
- 不同位发生错误应当使余数不同

$$\begin{array}{r} 1010 \\ \times 101 \\ \hline 1010 \\ 0000 \\ 1010 \\ \hline 10010 \end{array} \quad \begin{array}{r} 101 \overline{) 10000} \\ \underline{101} \phantom{00} \\ 0010 \phantom{0} \\ \underline{000} \phantom{0} \\ 0100 \phantom{0} \\ \underline{101} \phantom{0} \\ 001 \phantom{0} \end{array}$$

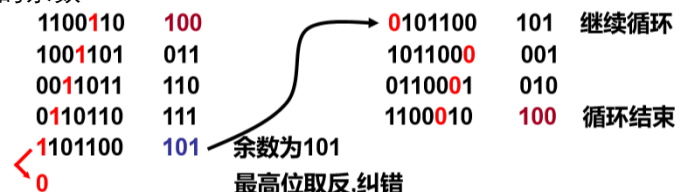
余数=001

CRC码的组成



	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>	A <sub>6</sub>	A <sub>7</sub>	余数
正确	1	1	0	0	0	1	0	000
A <sub>7</sub> 错	1	1	0	0	0	1	1	001
A <sub>6</sub> 错	1	1	0	0	0	0	0	010
A <sub>5</sub> 错	1	1	0	0	1	1	0	100
A <sub>4</sub> 错	1	1	0	1	0	1	0	011
A <sub>3</sub> 错	1	1	1	0	0	1	0	110
A <sub>2</sub> 错	1	0	0	0	0	1	0	111
A <sub>1</sub> 错	0	1	0	0	0	1	0	101

这方式真离谱啊!



- 对余数继续作模 2 除,应使余数循环

## ➤ Cache 的基本原理

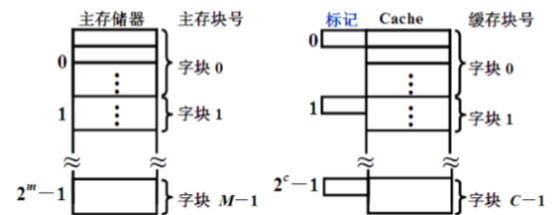
- 引入 Cache 的原因
  - 解决 CPU 和主存之间速度不匹配的问题
  - 避免 CPU 与 I/O 设备的访存冲突
    - 一旦主存需与 I/O 设备交换信息时, CPU 可以访问 Cache 获取信息

## ● Cache 的组成

- 一般使用 SRAM 构成
  - 典型大小在几百 KB 到几 MB
- 多级结构
  - CPU 内——主板上
- 全硬件实现, 对用户透明

## ● Cache-主存空间的基本结构

- 对主存和 Cache 分块, 每块中包含同样数目的存储字
  - Cache 容量远小于主存, 因此 Cache 块数远少于主存块数
- Cache 与主存之间的数据交换以块为单位



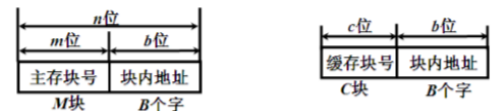
## ● Cache 的工作原理

### ● 控制逻辑

- 片外 Cache, 主存/Cache 控制器
- 片内 Cache, CPU 控制

### ● 数据交换过程

- CPU 将内存地址发给 Cache 和主存
- 控制逻辑判断该地址是否在 Cache 中
  - 1) 命中, Cache 将存储字提供给 CPU (按字传送)
  - 2) 未命中, 主存将存储字提供给 CPU, 同时将该字所在的整个块调入 Cache (按块传送) ——主存块与 Cache 块建立对应关系
  - 判断是否命中——利用 Cache 块的标记



### ● Cache 命中率

- 衡量 Cache 的性能效率的指标
- 与 Cache 的容量和块长有关
- 设程序执行过程中,  $N_c$  表示访问 Cache 完成存取的总次数,  $N_m$  表示访问主存完成存取的总次数, 则命中率  $h$  为  $h = \frac{N_c}{N_c + N_m}$

- 设  $t_c$  为命中时的 Cache 访问时间,  $t_m$  为未命中时的主存访问时间, 则 Cache/主存平均访问时间  $t_a$  为  $t_a = ht_c + (1 - h)t_m$

- 用  $e$  表示访问效率, 有  $e = \frac{t_c}{t_a} \times 100\% = \frac{t_c}{ht_c + (1-h)t_m} \times 100\%$

- $t_a$  越接近  $t_c$ ,  $h$  越接近 1, 效率越高

- 例: 假设 CPU 执行某段程序时, 共访问 Cache 命中 2000 次, 访问主存 50 次。已知 Cache 存取周期为 50ns, 主存存取周期为 200ns。求 Cache-主存系统的命中率、效率和平均访问时间。

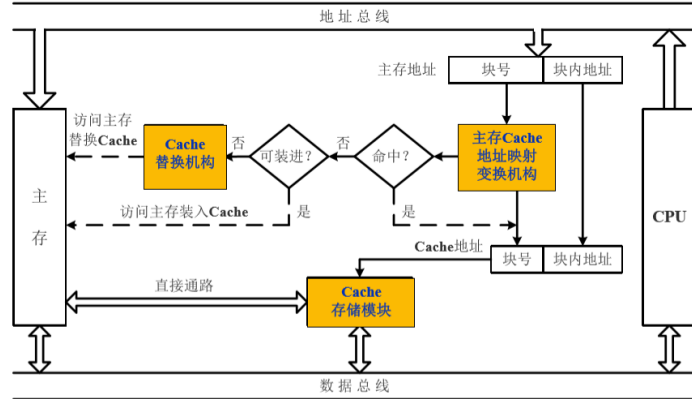
- Cache 命中率  $h = 2000 / (2000 + 50) = 0.97$ ; 平均访问时间为  $50 \text{ ns} \times 0.97 + 200 \text{ ns} \times (1 - 0.97) = 54.5 \text{ ns}$ ;  $e = \text{访问 Cache 的时间} / \text{平均访问时间} \times 100\% = 50 \text{ ns} / 54.5 \text{ ns} \times 100\% = 91.7\%$

- Cache 命中率与容量

- Cache 容量越大，命中率越高
  - 当容量达到一定值时，命中率将不再明显提高
- Cache 容量越大，成本越高
- Cache 命中率与块长
  - 取决于程序的局部特性
  - 随着块由小到大增长，命中率最初将提高，而后下降
  - 块长一般取每块 4~8 个字

#### ➤ Cache 的基本结构

- Cache 存储模块
  - 以块为单位与主存交换信息
- 地址映射变换机构
  - 将 CPU 送来的主存地址转换为 Cache 地址
  - 地址映射函数
- 替换机构
  - Cache 内容已满，无法容纳新的主存块时，确定 Cache 内哪个块移出到主存
  - 替换算法
- 单一 Cache 和多级 Cache
  - 单一 Cache
    - CPU 和主存之间只设一个 Cache
    - 随着技术发展，与 CPU 制作在同一芯片中——片内 Cache
    - 存取速度快，不占用片外系统总线，但容量受限
  - 多级 Cache
    - 在片内 Cache 基础上，再增加一（多）级片外 Cache
    - 与 CPU 之间使用独立数据通路
- 统一 Cache 和分离 Cache
  - 统一 Cache，指令和数据存放在同一 Cache 内
  - 分离 Cache，指令和数据分别存放——指令 Cache 和数据 Cache

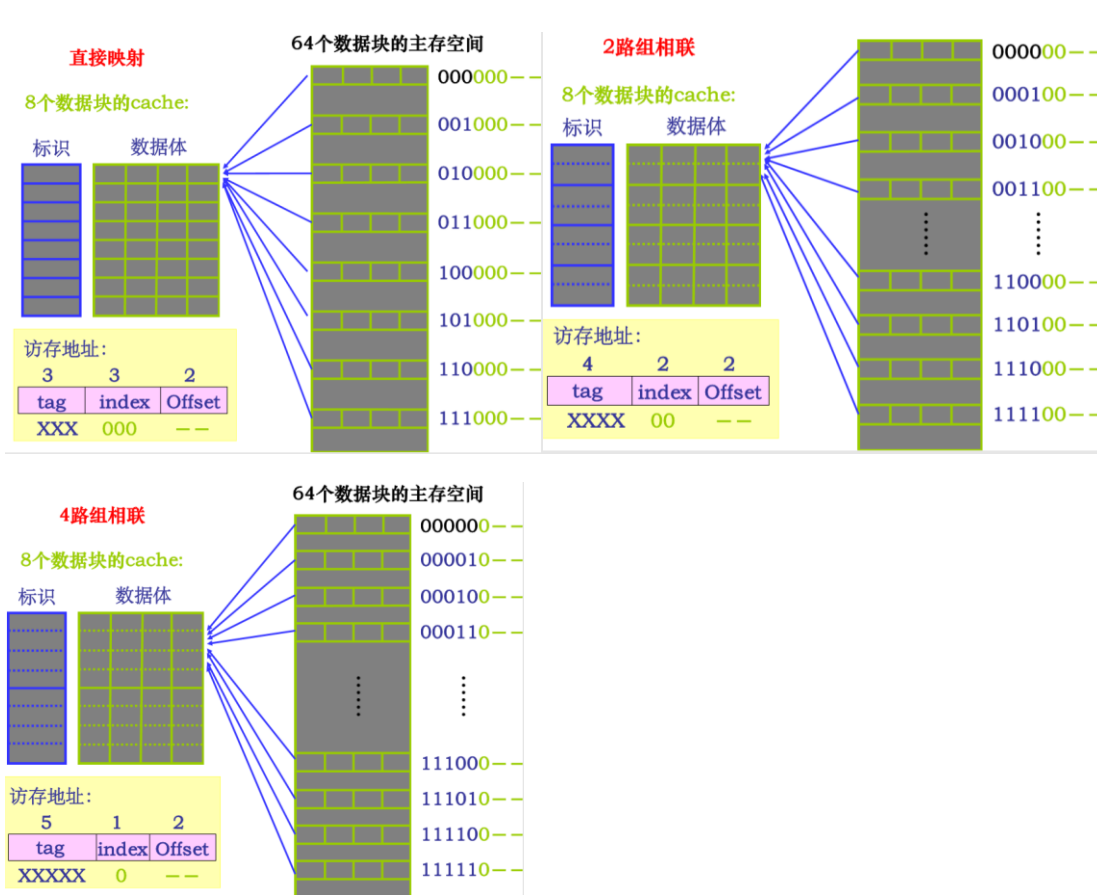


#### ➤ Cache 与主存的地址映射

- 地址映射
  - 将主存地址定位到 Cache 地址的方法
  - 地址映射的方式 (Cache 组织方式)
    - 全相联映射——灵活性大的映射关系
    - 直接映射——固定的映射关系
    - 组相联映射——前两种的折中
  - 选择映射方式需考虑的因素
    - 硬件是否容易实现
    - 地址变换的速度是否快
    - 发生冲突的概率是否低
- 全相联映射
  - 主存中的每一个字块可以映射到 Cache 中的任何一块位置上
  - 利用**标记**来记录映射关系，标记构成了一张目录表
  - 将地址分为两部分 (块号和字)，在内存块写入 Cache 时，同时写入块号标记
  - 主存地址长度 = (s+w)位，块为 s 位，主存块数为  $2^s$ ；字为 w 位，块大小为  $2^w$  个字或字节
  - 适用于小容量的 Cache
  - 为了快速检索，内存地址的块号与 Cache 中所有行的标记同时进行比较







● 地址映射方式例题:

- 假设主存容量为 512KB, Cache 容量为 4KB, 每个块为 16 个字, 每个字 32 位
  - 1) Cache 地址多少位? 可以容纳多少块?
  - 2) 主存地址多少位? 可以容纳多少块?
  - 3) 在直接映射方式下, 主存的第几块映射到 Cache 的第 5 块 (设起始字块为第 1 块) ?
  - 4) 画出直接映射方式下主存地址字段中各段的位数
- - 1) Cache 容量  $4KB = 2^{12}B$ , 地址为 12 位。块数  $m = 4KB / (16 \times 4B) = 64$  块
  - 2) 主存容量  $512KB = 2^{19}B$ , 地址为 19 位。块数  $n = 512KB / (16 \times 4B) = 8192$  块
  - 3) 主存第  $5$ 、 $2^6+5$ 、 $2 \times 2^6+5$ 、 $\dots$ 、 $2^{13}-2^6+5$  块映射到 Cache 第 5 块

主存块标记	Cache块地址	块内地址
-------	----------	------

4)           7位           6位           6位

- 某 PC 主存容量分 2048 块, 每块 512B, Cache 容量 8KB, 分为 16 块, 每块 512B
  - 1) 用直接映射时, 主存应被分几段? Cache 标记几位?
  - 2) 用全相联映射, Cache 标记几位?
  - 3) 用组相联映射, Cache 每组 2 块 (即: 两路组相联), 主存应划分为几段? 每段几块? Cache 标记几位?
- - 1) 主存容量为  $2^{20}B$ , 有 20 位的地址; cache 容量为  $2^{13}$ , 有 13 位的地址; cache 有  $16=2^4$  块, 需要 4 位表示块地址, 每块  $512=2^9B$ , 需要 9 位来表示块内地址; 综上, 主存的 20 位被分为 3 段, 分别是: 主存块标记: 7 位; Cache 块地址: 4 位; 块内地址: 9 位
  - 2) 全相联时 Cache 块地址位数应全部转给主存块标记, 此时主存块标记为 11 位, 块内地址为 9 位
  - 3) 还是划分为 3 段, 因为现在分成了 8 个组, 所以块内地址变为 3 位, 相应的主存块标记变为 8 位, 块内地址还是 9 位

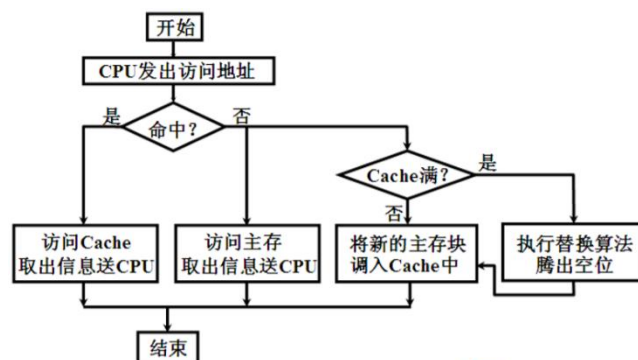
- 设有一个 cache 的容量为 2K 字，每个块为 16 字，求
  - (1) 该 cache 可容纳多少个块？
  - (2) 如果主存的容量是 256K 字，则有多少个块？
  - (3) 主存的地址有多少位？ cache 地址有多少位？
  - (4) 在直接映像方式下，主存中的第  $i$  块映像到 cache 中哪一个块中？
  - (5) 进行地址映像时，存储器的地址分成哪几段？各段分别有多少位
- 1) 块数  $= 2 \times 2^{10}W / 16W = 2^7$  块  $= 128$  块  
 2) 块数  $= 256 \times 2^{10}W / 16W = 2^{14}$  块  
 3) 主存容量为 256K 字  $= 2^{18}$  字。因此有 18 位地址； cache 容量为 2K 字  $= 2^{11}$  字，因此有 11 位地址  
 4) cache 共有 128 块，则第  $i$  块映射到 cache 的  $i \bmod 128$  块  
 5) 分成 3 段：主存块标记：7 位； cache 块标记：7 位； 块内地址：4 位

#### ➤ Cache 替换策略

- 替换产生的原因
  - Cache 容量比主存小得多，某个新的主存块要调入 Cache 时，其对应可存放的 Cache 块可能都已经被使用
- 替换策略与 Cache 组织方式密切相关
  - 直接映射，直接替换对应 Cache 块即可
  - 全相联和组相联映射，选择某一块——替换算法
- 常用的替换算法
  - 先进先出 FIFO 算法——选择最早调入 Cache 的块进行替换
  - 随机法——利用随机数产生器，随机选择被替换的块
  - 最少使用 LFU 算法——被访问的块计数器增加 1，替换时选择计数值最小的块（同时清零其计数器）缺点：不能反映 cache 近期访问情况
  - 最近最少使用 LRU 算法——被访问的块计数器置 0，其他块的计数器增加 1，替换时选择计数值最大的块，符合 cache 的工作原理

#### ➤ Cache 的读写策略

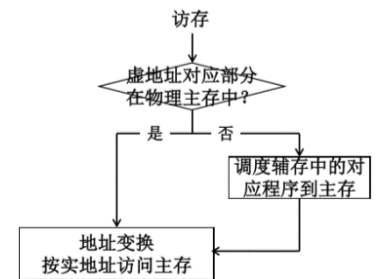
- Cache 的读操作过程
- Cache 的存储一致性问题
  - Cache 块内写入的信息，必须与被映射的主存块内的信息完全一致
- Cache 的写策略（写命中）
  - 写直达法
    - 写操作时，数据既写入 Cache，又写入主存
    - 写操作时间就是访问主存的时间
    - 优点：读操作时不涉及对主存的写操作，更新策略比较容易实现
  - 写回法
    - 写操作时，只把数据写入 Cache 而不写入主存
    - 当被写过的 Cache 块要被替换时才写回主存
    - 写操作时间就是访问 Cache 的时间，读操作 Cache 失效发生数据替换时，被替换的块需写回主存，增加了 Cache 的复杂性
- Cache 的写策略（写失效）
  - 按写分配(写时取)：写失效时，先把所写单元所在的块调入 Cache，再行写入
  - 不按写分配(绕写法)：写失效时，直接写入下一级存储器而不调块
- 写策略与调块



- 写回法 —— 按写分配
- 写直达法 —— 不按写分配
- Cache 的两种写策略（命中+失效）
  - 写回 – 按写分配：
    - 命中：只写 cache
    - 调块，只写 cache
  - 写回 – 不按写分配：
    - 命中：只写 cache
    - 失效：只写主存
  - 写直达 – 按写分配：
    - 命中：写 cache 写主存
    - 失效：调块，写 cache 写主存
  - 写直达 – 不按写分配：
    - 命中：写 cache 写主存；
    - 失效：只写主存；

#### ➤ 虚拟存储器的基本概念

- 实地址与虚地址
  - 产生虚地址的原因
    - 程序所需的存储容量超过了实际的物理内存容量
    - 多用户多任务系统中，多用户或多任务共享内存并行执行，每个程序占用的实际内存存在编程时无法确定，需运行时动态分配
  - 虚地址：也称逻辑地址，指用户编程时使用的地址
    - 对应的存储空间称为虚存空间或逻辑地址空间
  - 实地址：也称物理地址，指实际的物理主存的地址
    - 对应的存储空间称为主存空间或物理存储空间
  - 虚地址到实地址的转换过程称为程序的重定位
  - 编程时无需考虑物理主存的大小，也无需考虑程序如何存放
- 虚存访问过程
  - 用户按虚地址编程并保存在辅存中。程序执行时，分配给每个程序一定的物理主存空间，由地址转换部件完成重定位；若分配的物理主存不够，则只调入当前正运行或将要运行的程序或数据，其他部分暂存在辅存中
- 虚地址空间与实地址空间
  - 虚地址空间可以远大于实地址空间
    - 用于提高存储容量
  - 虚地址空间可以远小于实地址空间
    - 通常出现在多用户多任务系统中，单个任务不需要很大的实地址空间，使用较小的虚存空间可以缩短指令中地址字段长度
  - 虚存空间大小仅依赖于辅存的大小
  - 在主存命中率较高时，虚存的访问时间接近于主存访问时间
  - 每个程序可以拥有一个具有辅存的存储容量和近似主存访问速度的虚拟存储器



#### ➤ 虚存与 Cache

- 三级存储体系的两个层次
  - Cache——主存——辅存
  - 两个层次间分别有辅助硬件和辅助软硬件负责地址变换和管理
  - Cache 和主存构成系统内存，主存和辅存构成虚拟存储器
- 虚存与 Cache 的相同点

- 出发点相同:为了提高存储系统的性价比,利用了程序运行时的局部性原理,使其性能接近高速存储器,而价格和容量接近低速存储器
- 两个存储层次的不同点
  - 侧重点不同
    - Cache 主要解决主存与 CPU 的速度差异问题
    - 虚存主要解决主存与辅存的存储容量的问题
  - 数据通路不同
    - CPU 与 Cache 和主存之间均有直接访问通路, Cache 不命中时可以直接访问主存
    - 虚存依赖的辅存与 CPU 之间没有直接数据通路, 主存不命中时只能通过换页解决
  - 透明性不同
    - Cache 完全由硬件管理, 对系统程序员和应用程序员均透明
    - 虚存由软件(操作系统)和硬件共同管理, 对系统程序员不透明
  - 未命中时的代价不同
    - 主存未命中时的系统性能损失要远大于 Cache 未命中时的损失

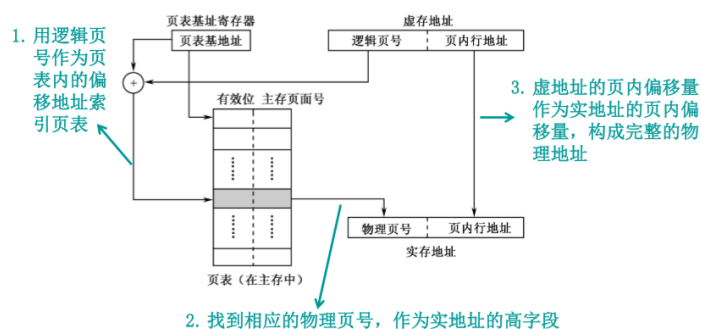
#### ➤ 虚存机制要解决的关键问题

- 地址映射问题
  - 虚地址如何变为主存物理地址(内地址变换)或辅存物理地址(外地址变换), 以便换页
  - 主存分配、存储保护和程序重定位等问题
- 替换问题
  - 决定哪些程序和数据被调出主存
- 更新问题
  - 确保主存和辅存相应数据和程序的一致性页式虚拟存储器

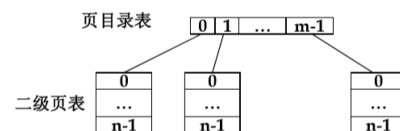
#### ➤ 页式虚拟存储器

- 虚拟存储器可以分为页式、段式、段页式存储器
- 页式虚存地址概念
  - 逻辑页: 虚地址空间被分成的等长大小的页
  - 物理页: 主存空间被分成的与逻辑页同样大小的页
  - 虚地址: 分为两个字段, 高字段为逻辑页号, 低字段为页内偏移量
  - 实地址: 分为两个字段, 高字段为物理页号, 低字段为页内偏移量
- 页表
  - 用于将虚地址(逻辑地址)转换成实地址(物理地址)
  - 每个进程对应一个页表
  - 页表的一个表项对应一个逻辑页
  - 表项的内容
    - 对应的逻辑页所在的物理页号
    - 指示该逻辑页是否已调入主存的有效位

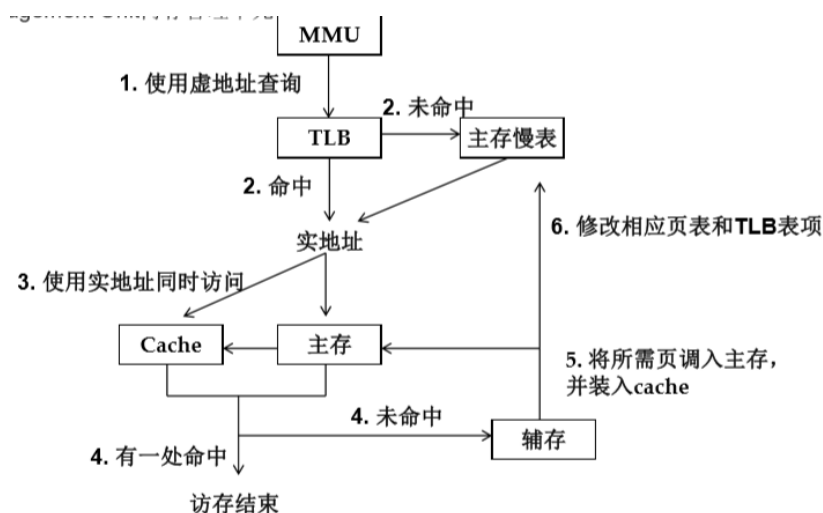
- 页式虚存地址映射
  - 现代 CPU 通常有专门的硬件支持地址映射转换
- 页表基址与页表分页
  - 页表基址
    - 将页表基址保存在页表基址寄存器中, 页表本身保存在主存中
    - 每个进程需要的页数不同, 页表长度可变
    - 地址映射: 页表基址 + 逻辑页号



- 页表的分页
  - 每个进程的页表可能很长
    - 设一个进程的虚地址空间为 2GB，每个逻辑页大小为 512B，则页表 表项数（逻辑页数）为  $2^{31} / 2^9 = 2^{22} = 4M$
  - 将页表存储在虚存中，通过对页表分页节省页表占用的主存空间
    - 进程运行时，其页表一部分在主存中，一部分在辅存中
- 二级页表
  - 设置一个页目录表，其中的每个目录表项指向一个页表
- 反向页表
  - 一般情况：逻辑页少（主存空间），物理页多（辅存空间）
  - 在页表较大的系统中，实现物理页号(少)到逻辑页号(多)的映射
  - 主存的物理页数目决定了反向页表的长度（表项数）
  - 优点：页表所占空间大大缩小
  - 缺点：反向查找匹配的时间很长，性能受限于查找算法
- 快速地址变换——快表 TLB
  - TLB 的使用（Translation Look-aside Buffer）
  - 转换检测缓冲区
  - 为什么要使用快表？
    - 页表一般保存在主存中，即使逻辑页已经在主存中，也至少需要访问两次物理主存才能完成需要的访存操作，这使得虚存的存取时间加倍
  - 为减少访存次数，对页表进行二级缓存，将页表中最活跃的部分存放在高速存储器（如 Cache）中，组成快表 TLB
    - TLB：专用于页表缓存的高速存储部件
    - 保存在主存中的完整页表称为**慢表**
- TLB 的地址映射
  - 根据逻辑页号同时查找快表和慢表
  - 按地址访问：将逻辑页号作为慢表表项的索引
  - 按内容比较：将逻辑页号作为匹配关键字在快表表项内容中进行查找
  - 根据程序局部性原理，多数虚存访问都将通过 TLB 完成，从而能够有效降低访存的时间延迟
- 完整的一次访存过程



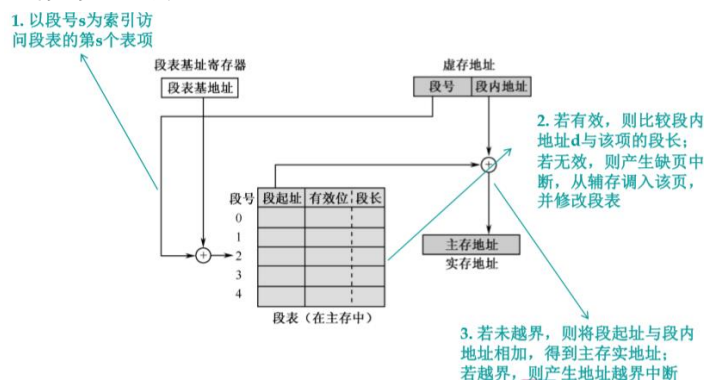
一个进程最多可以有  $m \times n$  个逻辑页



## ➤ 段式虚拟存储器

- 分页的优缺点
  - 优点：**页长固定**，便于构造页表和管理，不存在碎片

- 缺点：页长与程序逻辑大小无关，不利于编程的独立性 以及复杂的存储管理操作
- 段
  - 按程序的自然分界划分的长度可以动态改变的区域
    - 通常将代码、操作数和常数等不同类型数据划分到不同的段中
  - 每个程序可以有多个相同类型的段
  - 段的虚地址组成：段号 + 段内偏移量
  - 分段为组织程序和数据提供了方便
    - 分页对程序员不可见，而分段通常对程序员是可见的
- 段表
  - 用于虚地址到主存实地址的变换
  - 本身也是一个段，一般保存在主存中
  - 每个程序设置一个段表，其中每个表项对应一个段
  - 段表表项字段（和页表的表项页号+有效位区别）
    - 有效位——指明该段是否调入主存
    - 段起始地址——指明该段已在主存中时的首地址
    - 段长——记录该段的实际长度
    - 段长字段用于保证访问某段的地址空间时，段内地址不会超过该段长度而导致地址越界，破坏其他段
- 段式虚存的地址映射



- 段式虚存的优点
  - 段具有逻辑独立性，便于编译、管理、修改、保护和多程序共享
  - 段长可以按需动态改变，允许自由调度，能有效利用主存空间
- 段式虚存的缺点
  - 主存空间分配比较复杂
  - 容易在段间产生许多碎片，造成存储空间利用率降低
  - 段长不一定为2的整数次幂，必须进行段起始地址与段内偏移量的求和运算来得到物理地址
    - 相比于页式存储管理，段式存储管理需要更多的硬件支持
- 段式虚存和页式虚存的结合
  - 物理主存被等分为页
  - 程序先按逻辑结构分段，每个段再按照主存的页大小分页
  - 程序按页调入调出，但按段进行编程、保护和共享
- 段页式虚存管理
  - 每个程序通过一个段表和多个页表进行两级定位
  - 段表的每个表项对应一个段，其中有一个指针指向该段的页表
  - 页表指明该段各页对应的主存物理页，以及是否装入、是否修改等状态信息
- 段页式虚地址



基号N	段号S	段内逻辑页号P	页内地址偏移量D
-----	-----	---------	----------

- 多任务系统中，操作系统用来表明该程序在系统中的序号

- 虚存的替换算法
  - 替换的时机——需从辅存调页而主存已满时
  - 主要的替换算法
    - FIFO 算法：First In First Out，先入先出
    - LRU 算法：Least Recently Used，最近最少使用
    - LFU 算法：Least Frequently Used，最不经常使用
  - 虚存替换算法与 Cache 替换算法的不同
    - Cache 替换全部由硬件实现，虚存替换有操作系统的支持
    - 虚存缺页对系统性能的影响比 Cache 未命中要大很多
  - 具体算法 OS 已经复习过了，不会那是你的事情

## 思考题

- Cache 效率和局部性的度量指标？  
cache 命中率、效率、有效访问时间
- Cache miss 的原因？  
所访问的块不在 cache 上，应该从内存中调入该块到 cache 上
- 三种映射方式各自需要哪种置换算法？  
直接映射：这只能换固定的那一块，不就是 FIFO 了  
组相联映射：LRU 或 FIFO  
全相联映射：建议 LRU
- 虚存管理哪些由硬件实现，哪些由 OS 实现？MMU 中包含哪些模块？  
我不知道，而且这个也不重要
- TLB miss 和缺页异常如何处理？  
发生 TLB miss：从主存的慢表中继续搜索目标页，找到后调入 TLB 即可  
发生缺页异常：在在慢表中仍然没有找到目标页，这个时候发生缺页异常，需要从辅存中找到目标页并调入至内存中，同时再写入到 TLB 中
- 执行一条 load/store 指令需要访存几次？  
分情况讨论，如果 TLB 命中，那么直接从 TLB 中访存页即可，如果 TLBmiss，那么需要进行一次内存访问。如果发生缺页异常，那么还要从虚存中调入页，然后再进行内存访存。
- 程序的虚地址空间与磁盘空间如何映射？  
三种情况，分页，分段，段页结合，细节看上面
- Cache 与 MMU 实现机制比较？  
上面也有优缺点比较。
- 具有 1 位纠错能力的编码系统最小码距是多少？  
3
- SEC 海明码码距是多少？  
 $L-1 \geq D+C$
- $(n+k, n)$  CRC 码码距是多少？  
k

## Chapter 7 总线系统

### ➤ 总线的基本概念

- 构成计算机系统的互连结构，是连接系统中多个部件的信息传输线
- 实现计算机各个部件地址、数据和控制信息的交换，并在争用资源的基础上进行工作
  - 某一时刻，只允许有一个部件向总线发送信息，多个部件可以同时从总线上接收相同信息
- 总线的信息传送
  - 由许多传输线或通路组成，每条线可以传输一位二进制代码

### ➤ 总线的分类

- 按数据传送方式划分
  - 并行总线（又可按数据宽度细分）
  - 串行总线
- 按总线的使用范围划分
  - 计算机总线
  - 测控总线
  - 网络通信总线等
- 按时钟同步/异步划分
  - 总线上的数据与时钟同步工作的总线称为同步总线
  - 与时钟不同步工作的总线称为异步总线
- 单机系统中，按连接部件不同划分
  - 内部总线：连接 CPU 内部各寄存器及运算部件
  - 系统总线：连接 CPU 同计算机系统的其他高速功能部件，以及中低速 I/O 设备（I/O 总线）
- 系统总线
  - CPU、主存、I/O 设备各大部件之间的信息传输线，也叫板级总线
  - 系统总线按传输信息不同
    - 数据总线：用于传输各功能部件之间的数据信息，双向传输总线
    - 地址总线：用来指出数据总线上的数据在主存单元或 I/O 设备的地址，单向传输总线，由 CPU 发出
    - 控制总线：用来发出各种控制信号，单向传输线
- 常见的控制信号
  - 时钟：用来同步各种操作
  - 复位：初始化所有部件
  - 总线请求：表示某部件需获得总线使用权
  - 总线允许：表示需要获得总线使用权的部件已被允许
  - 中断请求：表示某部件提出中断请求
  - 中断响应：表示中断请求已经被接收
  - 存储器写：将数据总线的的数据写至存储器的指定地址单元
  - 存储器读：将指定存储单元中的数据读至数据总线上
  - I/O 读：从指定的 I/O 端口将数据读至数据总线上
  - I/O 写：将数据总线上的数据输出至指定的 I/O 端口

### ➤ 总线的特性和性能指标

- 总线的特性
  - 物理特性
    - 总线的物理连接方式，包括总线的根数、插头、插座形状、引脚线个数及排列方式等
  - 功能特性

- 描述总线中每一根线的功能，如地址总线、数据总线、控制总线
- 电气特性
  - 定义每一根线上信号的传递方向及有效电平范围
  - 送入 CPU 的信号称为输入信号，CPU 发出的信号称为输出信号
- 时间特性
  - 定义每根线在什么时间有效，即规定总线各信号的有效时序关系
- 总线的性能指标
  - 总线宽度：通常指**数据总线的位数**
  - 总线频率：1/传输一次数据时间
  - 总线带宽：总线的数据传输速率，即单位时间内总线传输数据的位数
    - 通常用**每秒传输信息的字节数**来衡量
  - 总线复用：一条信号线上分时传送多种信号
  - 其他指标：如负载能力、电源电压、总线宽度扩展等
- 例：(1) 某总线在一个总线周期中并行传送 4 个字节数据，假设一个总线周期等于一个总线时钟周期，总线时钟频率为 33MHz，总线带宽多少？
 

(2) 如果一个总线周期中并行传送 64 位数据，总线时钟频率升为 66MHz，总线带宽多少？
- 1)  $T = 1/33\text{MHz}$ ; 带宽  $D = 4B / T = 4 \times 33\text{MB/s} = 132\text{MB/s}$
- 2)  $D = 8B \times 66\text{MHz} = 528\text{MB/s}$

#### ➤ 总线标准

- 指系统与各功能模块、模块与模块之间的一个互连的**标准规范**
- 基于总线标准连接的两个模块，只需根据标准的要求**完成自身一方接口功能要求**，无须了解对方接口与总线的连接要求

#### ➤ 单总线结构

- 使用单一系统总线来连接 CPU、主存和 I/O 设备
- 特点：
  - 要求连接到总线上的部件必须高速运行完成操作，迅速放弃总线控制权
  - CPU 发出的地址，不仅加至主存，也同时加至总线上的所有外设
  - 对 IO 设备的操作与主存操作一样，可以指定地址
  - 易于扩展成多 CPU 系统

#### ➤ 多总线结构

- 在 CPU、主存、I/O 之间互联采用多条总线
- 双总线结构
  - 特点：将速度较低的 I/O 设备从单总线上分离，形成主存总线与 I/O 总线分开的结构
  - 多用于大、中型计算机系统
- 三总线结构
  - 特点：在双总线的基础上，进一步地将 I/O 设备按速率不同进行分类，形成多总线结构

#### ➤ 总线内部结构

- 早期总线的内部结构
  - 处理器芯片引脚的延伸，是处理器与 I/O 设备适配器的通道
  - 不足之处：总线结构与 CPU 密切相关，通用性差
- 现行总线内部结构
  - 标准总线，与结构、CPU 等无关
  - CPU 连同其 Cache 一起作为一个模块与总线相连
  - 四部分组成：
    - **数据传送总线**：数据、地址、控制
    - **仲裁总线**：包括总线请求线和总线授权线

- **中断和同步总线**：用于处理带优先级的中断操作，包括中断请求线和中断响应线
- **公用线**：包括时钟线、电源线、地线、复位线及加电/断电的时序信号线等

#### ➤ 总线仲裁（总线判优）

- 设备的主从状态
  - 连接到总线上的设备有**主动**和**被动**两种形态
  - 主设备持续占用总线的时间成为总线占用期
  - 主动方—对总线具有控制功能，可以启动一个总线周期，如 CPU 被动方—只能响应主动方的请求，如存储器
  - 每次总线操作，只能有一个**主动方**占用总线控制权，但可以同时有一个或多个**被动方**
- 总线仲裁
  - 对多个主设备提出的总线占用请求进行仲裁
  - 采用优先级或公平策略
  - 根据总线仲裁电路位置不同，分为**集中式仲裁**和**分布式仲裁**

#### ➤ 集中式仲裁

- 控制逻辑集中在一处（如 CPU 中的总线仲裁器）
- 每个设备模块有两条线连到总线仲裁器
  - 一条送往仲裁器的总线请求信号线 BR
  - 一条仲裁器送出的总线授权信号线 BG
  - 三种常见的集中式仲裁方式：
    - **链式查询**
    - **计数定时**
    - **独立请求**
- 链式查询方式
  - 总线授权信号线 **BG 串行地**从一个 I/O 接口传送到下一个 I/O 接口
    - 若 BG 到达的接口无总线请求，则继续往下查询
    - 若 BG 到达的接口有总线请求，则不再往下查询，当前 接口获得总线使用权，建立**总线忙 BS** 信号
  - 优先级仲裁——离总线仲裁器**最近**的设备具有**最高**的优先级
  - 优缺点
    - 优点：硬件连线简单，且易于扩充
    - 缺点：对电路故障敏感，优先级低的设备很难获得请求
- 计数器定时查询方式
  - 查询过程
    - 设备要使用总线时，通过 **BR 线**发出总线请求
    - 总线仲裁器接到请求信号后，在总线当前未被使用的情况下**开始计数**，并将计数值通过设备地址线发给各设备
    - 各设备接口将自身的设备地址与计数值进行比较，若一致，则该设备获得总线使用权，置 BS 线为“1”，此时中止计数查询
  - 每次计数可以从**“0”开始**，也可以从上一次的**中止值**开始
  - 特点：
    - 计数器初始值可以由程序设置，因而设备优先级**次序可以改变**
    - 对电路故障不敏感，但增加了控制线数，控制较复杂
- 独立请求查询方式
  - 每个设备都有一对**总线请求线和总线授权线**
    - 设备要使用总线时，发出该设备的请求信号
  - 总线仲裁器有一个排队电路，根据一定的优先次序决定首先响应哪个设备的请求

- 优缺点
  - 优点——响应时间快，对优先次序的控制十分灵活
  - 缺点——控制线数量多，控制更复杂
- 当代总线标准普遍采用的集中仲裁方式

#### ➤ 分布式仲裁

- 分布式仲裁不需要中央仲裁器，有三种常见的仲裁方式：
  - **自举分布式仲裁**（每个设备独立地决定是否是最优先请求者。在总线裁决期间，**每个设备将有关请求线上的信号合成后取回分析**，根据这些请求信号确定自己能否拥有总线控制权）
  - **冲突检测分布式仲裁**（每个设备独立地请求总线，多个同时使用总线的设备会发生冲突，冲突被检测到，按照**某种策略**在冲突的各方选择一个设备）
  - **并行竞争分布式仲裁**
    - 每个主设备具有专属的仲裁号和仲裁器
    - 第一个设备将自己的仲裁号写入仲裁总线
    - 仲裁过程
      - 当它们有总线请求时，把它们唯一的仲裁号发送到共享的仲裁总线上
      - 每个仲裁器将仲裁总线上得到的号与自己的号进行比较
      - 如果仲裁总线上的号大，则它的总线请求不予响应，并撤消它的仲裁号
      - 最后，获胜者的仲裁号保留在仲裁总线上
    - 基于优先级策略的仲裁方式

#### ➤ 总线操作与总线周期

- 读/写操作
  - 读操作：由**从设备**到**主设备**的数据传送
    - 地址-命令-数据
  - 写操作：由**主设备**到**从设备**的数据传送
    - 地址-数据-命令
  - 块传送操作，猝发式传送
    - 只需给出块起始地址，然后对固定块长度的数据一个接一个地读出或写入
- 写后读、读修改写操作
  - 两种组合操作：先写后读 / 先读后写
  - 只给出地址一次，读写为同一目标地址
  - 用途：
    - 先写后读一般用于**校验**目的
    - 先读后写多用于多道程序系统中对**共享存储资源的保护**
- 广播/广集操作
  - 一个主设备对多个从设备的写操作，称为**广播**
  - 多个从设备对一个主设备的读操作，称为**广集**
    - 将选定的多个从设备的数据在总线上进行与/或操作
    - 可用于检测多个中断源
- 总线周期
  - 通常指完成一次总线操作的时间
  - 一般可以分为 4 个阶段
    - **申请分配阶段**
      - 主设备提出总线使用申请，总线仲裁机构决定下一个传输周期的总线使用权归属
    - **寻址阶段**
      - 获得总线使用权的主设备发送本次要访问的从设备的地址及有关命令，启动参与本

次传送的从设备

- **传送阶段**
  - 主设备与从设备进行数据交换
- **结束阶段**
  - 主设备相关信息从总线上撤除，让出总线使用权

➤ 串行/并行传送与复用

- 传送方式
  - 串行传送
    - 只有一条传输线，采用脉冲传送
    - 位时间：每个二进制位在传输线上占用的时间长度，由同步脉冲体现
  - 并行传送
    - 使用多条传输线，同时传输多位二进制信息，采用电位传送
    - 串-并转换与并-串转换
- 分时复用
  - 总线复用，如既传数据，又传地址
  - 共享总线部件，分时使用

➤ 总线通信方式

- 总线通信控制
  - 主要解决通信双方如何获知传输开始和传输结束，以及通信双方如何协调如何配合
  - 四种方式：**同步通信**、**异步通信**、**半同步通信**、**分离式通信**
- 同步通信
  - 通信双方由统一的**时钟标准**控制数据传送
  - 时钟标准的形成
    - 通常由 **CPU 总线**控制部件发出，发送给总线上的所有设备部件
    - 也可以由各个设备部件各自的时序发生器发出，但必须由总线控制部件发出的时钟信号对它们进行**同步**
    - 优点：规定明确、统一，模块间的配合简单一致
    - 缺点：
      - **强制同步**，必须在限定的时间内完成规定操作
      - 需按**最慢**速度部件来设计公共时钟，影响总线效率，缺乏灵活性
    - 一般用于总线长度较短、各部件存取时间较一致的场合
- 异步通信
  - 没有公共的时钟标准，不要求所有部件严格统一操作时间，允许各部件速度不一致
  - 采用应答方式（握手方式）
    - 需在主、从设备间增加两条应答线
  - 异步通信应答方式：不互锁（访存）、半互锁（访问共享存储器）和全互锁（网络通信）
- 不同数据传输率的异步串行通信
  - 异步串行通信字符格式中包含起始位、终止位、校验位等若干附加位，若只考虑有效数据位，可用**比特率**来衡量数据传输速率
  - **比特率**——单位时间内传送的二进制**有效数据**的位数，单位为 bps
  - **波特率**——单位时间内传送的二进制**数据**的位数，单位为 bps
  - 为提高速度，可以去掉附加位，采用同步传送。同步传送中，数据块开始处要用同步字符 SYN 来指明
- 例：画图说明异步串行传送方式发送十六进制数据 95H。要求字符格式为：1 位起始位、8 位数据位、1 位偶校验位、1 位终止位  
95H = 1001 0101B



异步串行传送在起始位后传输数据位的**最低位**，数据位的最高位之后传输**校验位**，最后终止位。  
95H 的偶校验位为 0，波形图如下：



- 例：在异步串行传输系统中，字符格式为 1 位起始位、8 位数据位、1 位奇校验位和 1 位终止位。假设波特率为 1200bps，求相应的比特率  
比特率为单位时间内传输有效数据的位数，有效数据为 8 位，共 11 位，因此比特率为  $1200 \times (8/11)$  bps = 872.72bps
- 半同步通信
  - 保留同步通信的基本特点
    - 所有地址、命令、数据信号的发出，都严格参照系统时钟沿开始
  - 结合异步通信方式，允许设备部件以不同速度工作
    - 增设一条“**等待**”响应信号线，采用插入时钟（等待）周期的措施来协调通信双方的配合问题
  - 优点：控制方式比异步通信简单；各模块由统一时钟控制同步工作，可靠性较高
  - 缺点：等待时间不确定导致工作效率低
  - 适用于工作速度差异较大的各类设备组成的简单系统
- 上述三种通信方式的特点
  - 总线传输周期从主设备发出地址和读写命令开始，直到数据传输结束
  - 传输周期，系统总线由具有总线使用权的主设备和它选中的从设备占据
  - 总线传输周期时间主要花费在
    - 主设备通过总线向从设备**发送地址和命令**
    - 从设备按照命令**准备数据**
    - 从设备通过总线向主设备**提供数据**
- 分离式通信方式
  - 充分挖掘系统总线的潜力，提高系统性能
  - 基本思想：将一个总线周期分为两个子周期
    - 第一个子周期，主设备**获得总线使用权**后向相关从设备发送地址和命令等信息，然后**放弃总线使用权**
    - 第二个子周期，从模块准备好数据，然后**申请总线使用权**，向相应的主设备发送要求的数据信息
  - 特点
    - 两个子周期都只有单方向的信息流，每个设备其实都成了**主设备**
    - 各个设备都有权申请总线使用权
    - 采用**同步方式通信**，不等对方回答
    - 各模块**准备数据时，不占用总线**
    - 总线被占用时都在有效工作，不存在空闲等待时间
    - 总线在多个主、从设备间交叉重叠并行式传送
- 控制比较复杂，在普通微型计算机系统中很少采用，多见于大型计算机系统

## Chapter 8 I/O 系统

➤ I/O 系统的组成

- I/O 软件
  - 主要任务
    - 将用户程序或数据**输入**主机
    - 将运算结果**输出**给用户
    - 实现输入输出系统与主机工作的协调等
  - I/O 指令
    - **机器指令**的一类，反映 CPU 与 I/O 设备交换信息的各种特点 (IN/OUT)
  - 通道指令(通道控制字 Channel Control Word)
- I/O 硬件
  - 带接口的 I/O 系统，一般包括接口模块及 I/O 设备两部分
    - 接口电路一般包含数据传送通路、控制信号通路及相应逻辑电路
  - 具有通道的 I/O 系统

➤ I/O 系统的发展概况

- 早期阶段
  - I/O 设备种类少，与主存交换信息都必须通过 CPU
  - 当时 I/O 设备的特点
    - 每个 I/O 设备都必须由一套独立的逻辑电路与 CPU 相连，分散连接
    - 输入输出过程穿插在 CPU 执行过程进行，CPU 与 I/O 设备**串行工作**
- 接口模块和 DMA ( Direct Memory Access ) 阶段
  - I/O 设备通过接口模块与主机连接，系统采用**总线结构**
  - 接口提供缓冲和数据转换功能，并支持中断请求处理，I/O 设备与 CPU 可以并行工作
  - DMA: I/O 设备与主存之间使用一条直接的数据通路
- 具有通道结构的阶段
  - 在 I/O 设备繁多、数据传送频繁的情况下，DMA 方式同样带来硬件成本增加、控制复杂化和占用 CPU 时间等问题
  - 通道
    - 用来负责管理 I/O 设备以及实现主存与 I/O 设备之间交换信息的部件，可以认为是具有特殊功能的**处理器**
    - 专门的**通道指令**，构成独立地输入输出程序
    - 依赖 CPU 的 I/O 指令启动、停止或改变工作状态，是从属于 CPU 的专用处理部件
- 具有 I/O 处理机的阶段
  - 外围处理机，独立于主机工作
  - 可完成 I/O 通道要完成的 I/O 控制，以及数据处理、转换、检错纠错等操作
  - 与 CPU 工作的并行性更高

➤ I/O 设备与主机的联系

- I/O 设备编址方式
  - 统一编址
    - 将 I/O 设备地址看做存储器地址的一部分
      - 如在 64K 地址的存储空间中，划出 8K 地址作为 I/O 设备地址，凡对这 8K 地址范围的访问，就是对 I/O 设备的访问
    - I/O 指令与访存指令类似
  - 独立编址
    - I/O 地址和存储器地址分开，使用**专门的 I/O 指令**访问 I/O 设备
- 优缺点
  - 统一编址：占用存储空间，减少主存容量，但无须专用 I/O 指令

- 独立编址：不占用主存容量，但需设置专用的 I/O 指令
- 传送方式
  - 并行传送 V.S. 串行传送
- 联络方式
  - 用于 I/O 设备与主机相互了解彼此工作状态
  - 按 I/O 设备的工作速度，分为三种
    - 立即响应方式—对工作速度缓慢的 I/O 设备,无控制信号
    - 异步应答信号方式—I/O 设备与 CPU 工作速度不匹配的情况
    - 同步联络方式—I/O 设备与 CPU 工作速度完全同步
- I/O 接口概述
  - 接口：两个系统或两个部件之间的交接部分
    - 可以是两种**硬设备之间的连接电路**
    - 也可以是两个**软件之间的共同逻辑边界**
  - I/O 接口
    - 也叫**适配器**，指主机与 I/O 设备间设置的硬件电路及其相应的软件控制
  - 为什么需要 I/O 接口
    - CPU 可能连接多个不同设备号的 I/O 设备，可以通过接口实现**设备的选择**
    - 利用接口实现 I/O 设备与 CPU 的**数据缓冲**，减缓两者的速度差
    - 利用接口实现数据的**串-并转换、电平转换**
    - 通过接口传送**控制命令**
    - 通过接口监视**设备工作状态**并保存，供 CPU 查询使用
  - 接口与端口
    - 端口：指接口电路中的一些**寄存器**
      - 这些寄存器用于存放数据、控制命令、状态信息等
    - 接口与端口的关系
      - 若干端口加上相应的控制逻辑组成接口
    - CPU 对 I/O 接口（或 I/O 设备）的信息读写，实际上都是对端口的操作
- I/O 接口的功能与组成
  - 总线连接方式的 I/O 接口电路
    - 设备选择线
    - 数据线
    - 命令线
    - 状态线
  - I/O 接口的功能
    - 选址功能
      - 利用接口的设备选择电路实现
    - 传送命令的功能
      - 存放命令的命令寄存器、命令译码器等
    - 传送数据的功能
      - 设置数据缓冲寄存器，与数据线相连
    - 反映 I/O 设备状态的功能
      - 设置相关的状态触发器
        - 如中断请求触发器、中断屏蔽触发器、工作标志触发器等
      - 完成触发器 D，工作触发器 B
  - I/O 接口的基本组成
    - 数据缓冲寄存器 DBR

- 设备状态标记
- 设备选择电路
- 命令寄存器和命令译码器
- 控制逻辑电路

#### ➤ I/O 接口类型

- 按数据传送方式分类
  - 并行接口
  - 串行接口
- 按功能选择的灵活性分类
  - 可编程接口：接口功能及操作方式可通过程序来改变或选择
  - 不可编程接口：只能通过硬连线逻辑来实现不同功能
- 按通用性分类
  - 通用接口：可以供多种 I/O 设备使用
  - 专用接口：专门为某类外设或某种用途而设计
- 按数据传送的控制方式分类
  - 程序型接口：用于连接速度较慢的 I/O 设备，采用程序中断方式
  - DMA 接口：用于连接高速 I/O 设备

#### ➤ I/O 信息交换方式

- 程序查询方式、程序中断方式、DMA 方式、通道方式
- 程序查询方式
  - 程序查询方式的特点
    - 最简单的输入输出方式
    - 数据在 CPU 和外围设备之间的传送完全靠计算机程序控制
      - 在 CPU 的主动控制下进行
      - 有 I/O 操作时，CPU 暂停主程序，转去执行设备 I/O 的服务程序
    - 优缺点
      - 优点：CPU 与外设的操作能够同步，硬件结构简单
      - 缺点：CPU 循环查询，浪费 CPU 周期和资源
    - 一般用于单片机或数字信号处理 DSP 中
  - 查询通过做 I/O 指令完成
    - 对 I/O 接口的某些控制寄存器置“0”或“1”，用于控制设备进行相关工作
    - 测试设备的某些状态，如“忙”、“就绪”等，以便决定下一步操作
    - 传送数据
  - 程序查询方式的接口
    - 设备选择电路、数据缓冲寄存器、设备状态标志
- 程序中断方式
  - 中断系统
    - 计算机实现中断功能的软硬件总称
    - 一般在 CPU 中设置中断机构，在外设接口中设置中断控制器，在软件上设置相应的中断服务程序
- DMA 方式
  - DMA
    - 直接内存访问，一种完全由硬件执行 I/O 交换的工作方式
    - 主存与 I/O 设备间高速交换批量数据，传送速度快
    - 基本思想
      - 硬件 DMA 控制器从 CPU 完全接管对总线的控制，数据交换不经过 CPU，直接在主

存和 I/O 设备之间进行

- 工作过程描述
  - 在主存中要开辟连续地址的专用缓冲器，用来提供或接收传送的数据。在数据传送前和结束后 CPU 要通过**程序或中断方式**对缓冲器和 DMA 控制器进行预处理和后处理
  - 由 DMA 控制器给出当前正在传送的数据的主存地址，并统计传送数据的个数以确定一组数据的传送是否已结束
- DMA 控制器：通过大规模集成电路**硬件实现**
- DMA 的基本操作
  - 从外设发出 DMA 请求
  - CPU 响应请求，DMA 控制器接管总线控制
  - DMA 控制器对内存寻址，决定数据传送的内存单元地址及数据传送个数的计数，并执行数据传送操作
  - 向 CPU 报告 DMA 操作结束
- 在 DMA 方式中，数据传送前后的准备和处理工作，由 CPU 上的管理程序负责，DMA 控制器仅负责数据传送工作
- DMA 与主存交换数据的三种方式
  - 停止 CPU 访问内存
    - CPU 处于不工作状态或保持状态，未能发挥 CPU 对主存的利用率
    - 控制简单，适用于数据传输率很高的设备进行成组传输
  - 周期挪用（周期窃取）
    - DMA 控制器与主存间传送一个数据时，占用（窃取）一个或多个 CPU 周期。即 CPU 暂停工作一个周期，然后继续执行程序
    - I/O 设备通过 DMA 访问主存有三种可能情况
      - CPU 此时不访存——无冲突（执行复杂 ALU 指令）
      - CPU 正在访存——待访问结束后，让出总线
      - CPU 和 DMA 同时请求访存——**DMA 优先**
    - 较常采用的方法：异步、需申请总线访问，比较适合于 **I/O 读写周期大于主存周期** 的情况
  - DMA 与 CPU 交替访存
    - CPU 周期分为两部分，一部分**专用于** CPU 访存，另一部分**专用于** DMA 访存
    - 不需要申请和归还总线使用权，总线控制权的转移速度快，DMA 效率高，控制复杂
- DMA 控制器基本组成
  - 内存地址寄存器(计数器)AR：用于存放内存中要交换的数据的地址，DMA 传送时，每交换一次数据，寄存器值加“1”
  - 字计数器 WC：用于记录传送数据块长度
  - 数据缓冲寄存器 BR：用于暂存每次传送的数据
  - 设备地址寄存器 DAR：IO 设备码或辅存寻址信息
  - 控制/状态逻辑：负责管理 DMA 传送过程，用于修改相关信息和状态
- 中断机构：字计数器溢出时，表示一组数据传送完毕，触发中断
- DMA 数据传送过程
  - 分为三个阶段：预处理、数据传送、后处理
  - 预处理阶段
    - CPU 通过 I/O 指令给 DMA 控制器预置初值，取状态和设置传送需要的有关参数
    - 1) 通知 DMA 控制器传送方向
    - 2) 设备地址写入 DMA **设备地址寄存器**
    - 3) 主存地址写入**内存地址寄存器**

#### 4) 传送字数写入字计数器

- 数据传送阶段
  - 以数据块为基本单位
  - 通过循环来实现
- 后处理阶段
  - 校验送入主存的数据是否正确
  - 是否继续用 DMA
  - 由中断服务程序完成
- DMA 接口与系统的连接方式
  - 公用 DMA 请求方式
  - 独立 DMA 请求方式
- DMA 控制器的类型
  - 选择型
    - 在物理上连接多个设备，在逻辑上只允许连接一个设备
  - 多路型
    - 在物理上连接多个设备，在逻辑上允许连接多个设备同时工作
- 通道方式
  - 通道的基本概念
    - 通道：计算机系统中代替 CPU 管理控制外设的独立部件，是一种能执行有限 I/O 指令集合——通道命令的 I/O 处理部件
    - 在通道控制方式中，一个主机可以连接几个通道。每个通道又可连接多台 I/O 设备
      - 这些设备可具有不同速度，可以是不同种类
      - 这种输入输出系统增强了主机与通道操作的并行能力以及各通道之间、同一通道的各设备之间的并行操作能力
      - 为用户提供了增减外围设备的灵活性
    - 采用通道方式组织输入输出系统，多使用主机—通道—设备控制器—I/O 设备四级连接方式
    - 在 CPU 启动通道后，通道自动地去内存取出通道指令并执行。直到数据交换过程结束向 CPU 发出中断请求，进行通道结束处理工作
  - 通道的功能
    - 执行通道指令，组织外设和内存进行数据传输，按 I/O 指令要求启动外设，向 CPU 报告中断等
    - 具体 5 项任务
      - 接受 CPU 的 I/O 指令，按指令要求与指定的外围设备进行通信
      - 从内存选取属于该通道程序的通道指令，经译码后向设备控制器和设备发送各种命令
      - 组织外设和内存之间进行数据传送，并根据需要提供数据缓存的空间，以及提供数据存入内存的地址和传送的数据量
      - 从外围设备得到设备状态信息，形成并保存通道本身的状态信息，根据要求将这些状态信息送到内存的指定单元，供 CPU 使用
      - 将外围设备的中断请求和通道本身的中断请求，按次序及时报告 CPU
  - 通道的发展
    - IO 处理机 (IOP)
      - 不是一台独立的计算机，而是计算机系统中的一个部件
      - 可以和 CPU 并行工作，提供高速的 DMA 处理能力，实现数据的高速传送
      - 有些 IOP 还提供数据的变换、搜索和字装配/分拆能力



➤ I/O 设备

- 外部设备大致分三类：
  - 人机交互设备：键盘、鼠标、打印机、显示器
  - 计算机信息存储设备：磁盘、光盘、磁带
  - 通信设备：调制解调器
- 输入设备：
  - 键盘
  - 鼠标
  - 触摸屏
- 输出设备
  - 显示器
  - 打印机
- 其他
  - 模拟 / 数字（数字/模拟）转换器
  - 汉字处理

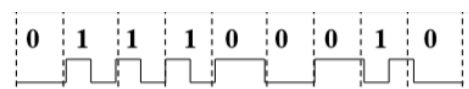
➤ 辅存概述

- 又叫外部存储器，简称外存
- 辅存特点
  - 容量大、速度慢、价格低、可脱机保存信息等
  - 不直接与 CPU 交换信息
- 辅存种类
  - 硬盘、软盘、磁带、光盘、U 盘、闪存等

➤ 磁记录设备

- 设备读写方式
  - 随机方式：RAM
  - 顺序方式：磁带（稳定，安全，速度慢，成本低）
  - 直接方式：磁盘（扇区的定位采用随机方式，依靠磁盘旋转可直接找到某一扇区，扇区内则采用顺序读写方式）
- 记录密度：道密度（磁盘）、位密度（磁盘、磁带）
  - 道密度：沿半径方向单位长度磁道数
    - 单位：道/英寸（TPI, Tracks Per Inch）P：道距
    - $D_t = \frac{1}{P}$
  - 位密度：单位长度磁道所记录的数据位数，单位为位/英寸（bpi）或位/毫米（bpm）
    - $D_b = \frac{f_t}{\pi d_{min}}$  其中  $f_t$  为每道总位数，各道相同， $d_{min}$  为同心圆最小直径
- 容量：存储的信息总量
  - 磁盘总容量  $C = n \times k \times s$ 
    - n：盘面数
    - k：每面磁道数
    - s：每道记录代码数
  - 非格式化容量：磁表面可以利用的磁化单元总数
  - 格式化容量：按某种特定的记录格式所能存储信息的总量，约为非格式化容量的 70%~80%
- 寻址时间
  - 磁盘寻址过程：直接存取
    - 随机寻道，顺序定位记录

- 寻址时间 = 寻道时间 ( $t_s$ ) + 等待时间 ( $t_n$ )
- 平均寻址时间
  - 寻道：最外、最内、相邻，各不相同
  - 等待时间：外道、内道长度不同
  - $T_a = t_{sa} + t_{wa} = \frac{t_{s\max} + t_{s\min}}{2} + \frac{t_{w\max} + t_{w\min}}{2}$
- 磁带寻址过程：顺序存取
  - 磁头不动，磁带空转到指定位置
  - 寻址时间 = 空转时间
- 传输率、误码率
  - 传输率：
    - 单位时间传输的数据量（字节、位）
    - $D_r = \text{记录密度 (D)} \times \text{介质运行速度 (V)}$
  - 误码率：
    - 读出时，出错位数/读出的总位数
    - 为了减少出错率，磁表面存储器通常采用循环冗余码 CRC 来发现并纠正错误
- 磁记录方式：又称编码方式
  - 常用的编码方式有：
    - 归零制 (NZ)
    - 不归零制 (NRZ)
    - 见 1 就翻的 NRZ1
    - 调相制-曼彻斯特编码 (PM)
    - 调频制 (FM)
    - 改进调频制 (MFM)
  - 归零制 (RZ)
    - 正脉冲电流表示“1”，负脉冲电流表示“0”
    - 不论记录“0”或“1”，在记录下一个信息前，**记录电流恢复到零电流**
    - 简单易行，记录密度低
    - 改写磁层上的记录比较困难，一般是先去磁后写入
    - **有自同步能力**（能从磁头读出信号中分离获得同步信号）
      - 自同步能力指：从单个磁道读出的脉冲序列中提取时钟信号频率的难易程度
  - 不归零制 (NRZ)
    - 磁头线圈始终有电流，电流方向“见变就翻”
    - 对连续记录的“1”和“0”，写电流的方向是不改变的
    - 无自同步能力
  - 见“1”就翻的不归零制 (NRZ1)
    - 磁头线圈始终有电流通过
    - 在记录“1”时，电流改变方向，写“0”电流保持不变
    - 不具备自同步能力，需要引用外同步信号
  - 调相制 (PM)：相位编码 (PE)、曼彻斯特码
    - 记录数据“0”时，规定磁化翻转的方向由负变为正，记录数据“1”时从正变为负
    - “0”，“1”的读出信号相位不同，抗干扰能力强
      - 磁带多用此方式
    - 具有自同步能力
  - 调频制 (FM)
    - 频率变化（“1”的频率是“0”的两倍）



- 在位与位之间的边界处都要翻转一次
- 具有自同步能力
- 用于软硬磁盘
- 改进调频制 (MFM)
  - 不是在每个位周期的起始处都翻转。当连续两个或两个以上“0”时，在位周期的起始位置翻转一次
  - 具有自同步能力



- 编码效率：位密度与磁化翻转密度的比值，用记录一位信息的最大反转次数表示
  - FM、PM：最多需反转 2 次，效率 50%
  - NRZ、NRZ1、MFM：最多只需反转 1 次，效率 100%

➤ RAID 技术：OS 复习过了，不会的话点击右上角红叉谢谢

➤ 例：磁盘存储器有 6 个盘片，最外两侧盘面不能记录，每面有 204 条磁道，每条磁道有 12 个扇段，每个扇段 512B，磁盘机以 7200rpm 的速度旋转，平均寻道时间为 8ms

1) 计算该磁盘存储器的存储容量

2) 计算该磁盘存储器的平均寻址时间

- 1) 6 个磁盘有 10 个记录面，存储容量 =  $10 \times 204 \times 12 \times 512\text{B} = 12533760\text{B}$
- 2) 平均寻址时间 = 平均寻道时间 + 平均等待时间，磁盘转一周的平均时间 =  $1/7200\text{rpm} \times 0.5 = 0.0000694\text{min} = 4.167\text{ms}$ ，平均寻址时间为  $8\text{ms} + 4.167\text{ms} = 12.167\text{ms}$

➤ 光盘存储器

- CD-ROM 光盘
  - 只读型光盘，容量约为 680MB
  - 存储原理
    - 光盘上的信息以坑点的形式分布，有坑点表示 1，无坑点表示 0
    - 信息记录的轨迹称为光道，光道划分为多个扇区。扇区是光盘的最小可寻址单位

➤ Flash 存储器

- 闪速存储器，高密度非易失性的读/写存储器
- Flash 存储元
  - 在 EPROM 存储元基础上发展而来
- 闪存基本操作
  - 编程操作（写操作）
    - 以页为单位，1 页为 512B、2KB、4KB 或者更大
  - 读取操作
    - 以页为单位，1 页为 512B、2KB、4KB 或者更大
  - 擦除操作
    - 以块为单位，1 块为 64K、256K、512KB 或者更大
  - 写入操作无法实现原地覆盖，写入之前必须进行额外的擦除

恭喜你看完了全部的文档内容，祝你考试顺利~