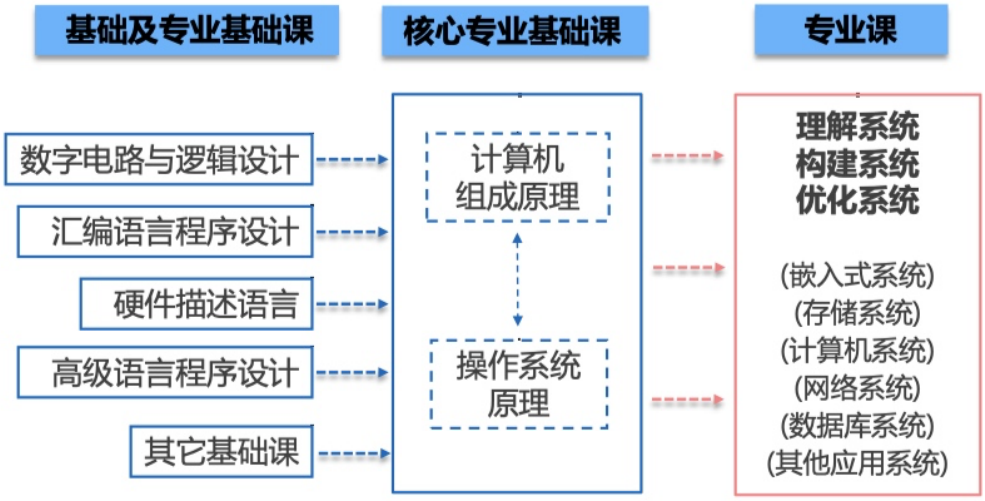
第一章 计算机系统概论

1.1 课程导学

1.1.1 为什么要学习计组?

核心专业基础课，在课程体系中承上启下的作用。

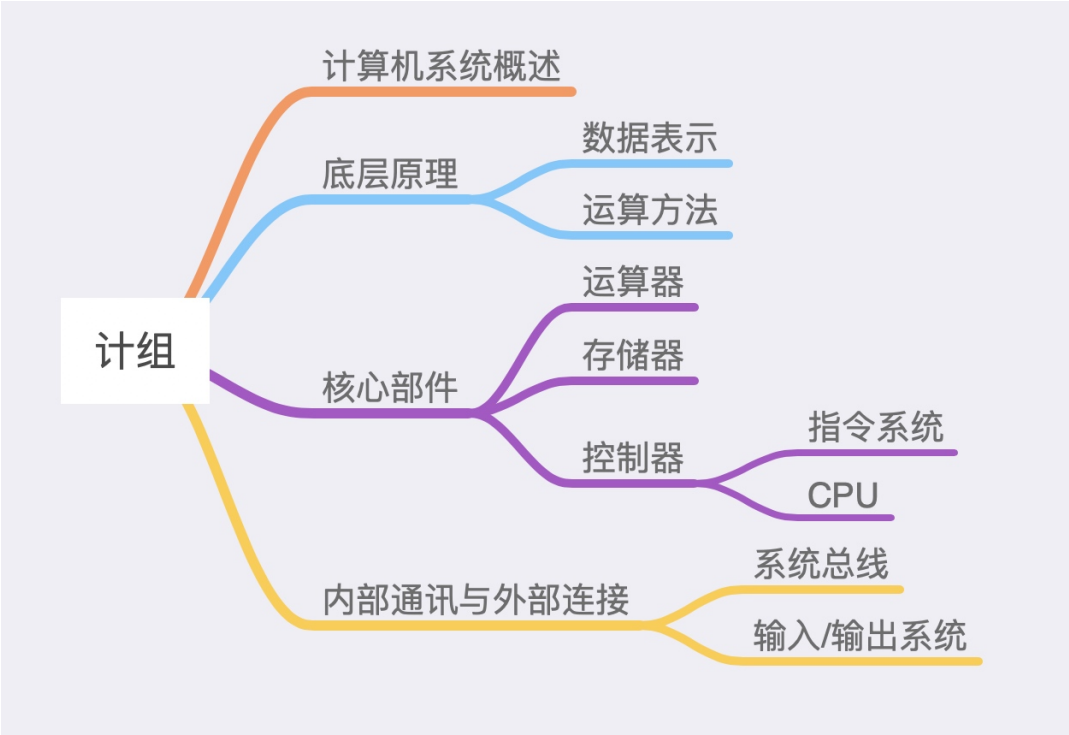
课程主要内容：介绍运算器、控制器、存储器的结构、工作原理、设计方法及互连构成整机的有关技术。（设计方法是课程关键）

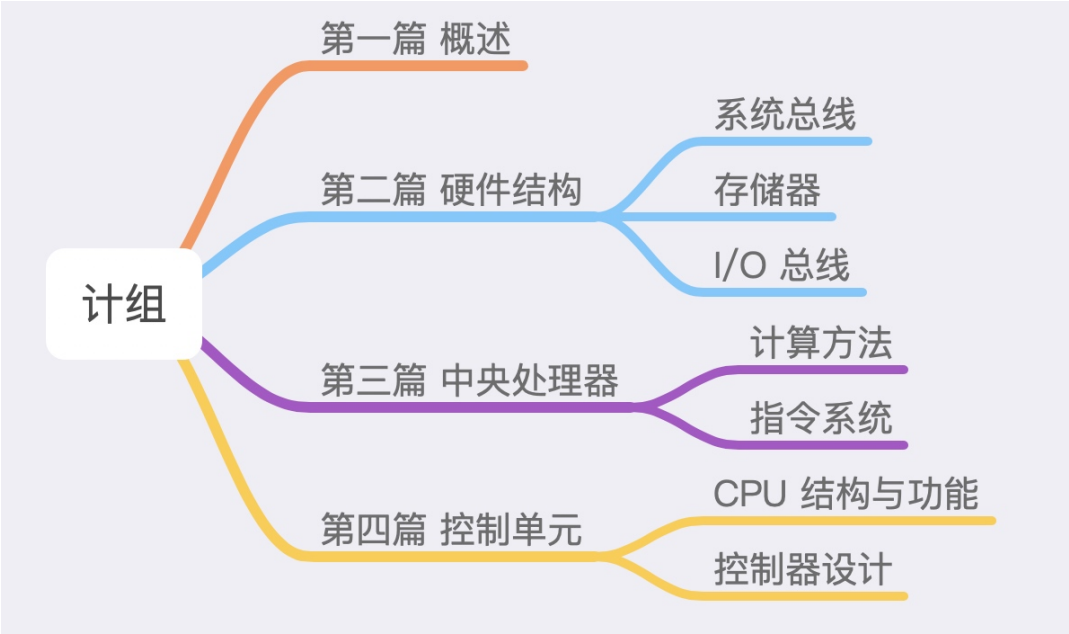


1.1.2 如何才能学好计组?

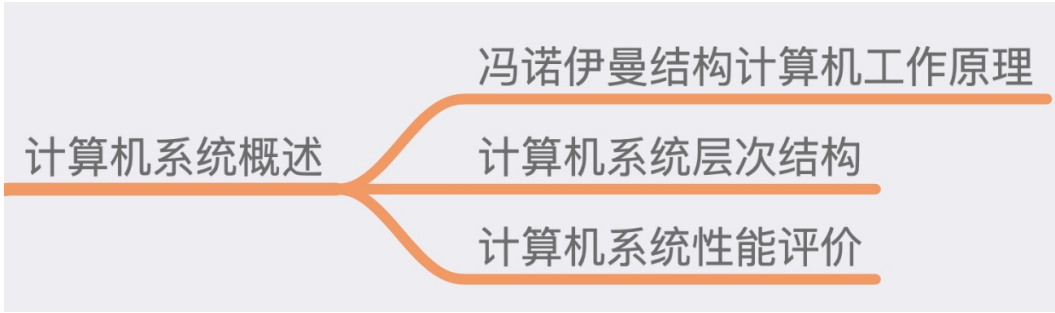
1. 构造观 —— 掌握设计方法（基本编码/解码器、运算器、控制器、存储器，简单计算机系统）
2. 系统观 —— 各部分之间的相互影响，牵一发而动全身
3. 工程观 —— 系统实现视角（成本、节能、可靠）

1.1.3 全书知识点概括





1.1.4 本章知识点概括



1.1.4 书本推荐

《深入理解计算机系统》—— 大学毕业前一定要认真读一遍

1.2 冯·诺伊曼计算机结构

1.2.1 冯·诺伊曼

1945年，冯·诺伊曼发表了《关于EDVAC的报告草案》，提出了“离散变量自动电子计算机方案（EDVAC）”，即冯·诺伊曼结构。并因此结构被称为“计算机之父”，该结构也沿用至今。

1.2.2 冯·诺伊曼计算机的工作原理

两个主要部分，分别是存储程序和程序控制。

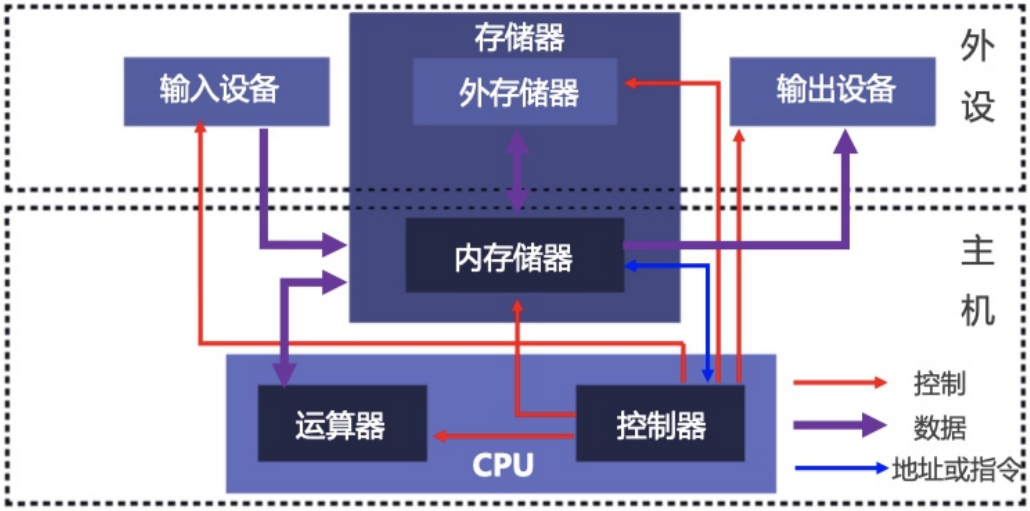
> 存储程序

将程序存放在计算机存储器中，只有放入存储器中的程序才能被执行。

> 程序控制

按指令地址访问存储器并取出指令，经译码依次产生指令执行所需的控制信号，实现对计算的控制，完成指令的功能。

1.2.3 冯·诺伊曼计算机的硬件结构



硬件结构主要分为三个部分：

1. 主机：CPU（运算器+控制器）、内存
2. 外设：输入设备、输出设备、外存储器
3. 总线：地址线、数据线、控制线

五大器件的功能：

1. 运算器：算术运算与逻辑运算，且能保存运算结果。
2. 控制器：产生指令执行过程所需要的所有控制信号，控制相关功能部件执行相应操作。（电平信号、脉冲信号）
3. 存储器：存储原程序、原数据、运算中间结果。
4. 输入设备：向计算机输入数据（键盘、鼠标、网卡、扫描仪等）
5. 输出设备：输出处理结果（显示器、声卡、网卡、打印机等）

从中也可以看出部分设备既可能是输入设备，也可能是输出设备。

1.2.4 冯·诺伊曼计算机的软件结构

对于软件的理解：

1. 软件本质：可运行的思想（算法）和内容（图像、声音、文字）的数字化
2. 软件的表现形式：程序和数据（以二进制表示的信息）
3. 软件的核心：算法

软件系统的分类：

1. 系统软件（操作系统、网络系统和编译系统等）
2. 支持软件（开发工具、界面工具等）
3. 应用软件（字处理软件、游戏软件等）

软件和硬件的关系：

1. 相互依存：硬件是软件运行基础，软件运行时硬件发挥作用的重要途径。
2. 逻辑等效性：某些功能既可由硬件实现，也可由软件实现。
3. 协同发展：软件随硬件发展而发展，软件的发展也会带来硬件的更新。

1.2.5 冯·诺伊曼计算机层次结构

在层次结构中，指令集架构层为软硬件的分界线，其上为软件，其下为硬件，其自身则为指令操作硬件的入口。

在层次结构中，有两个关键性概念，一个是透明性，一个是系统观。

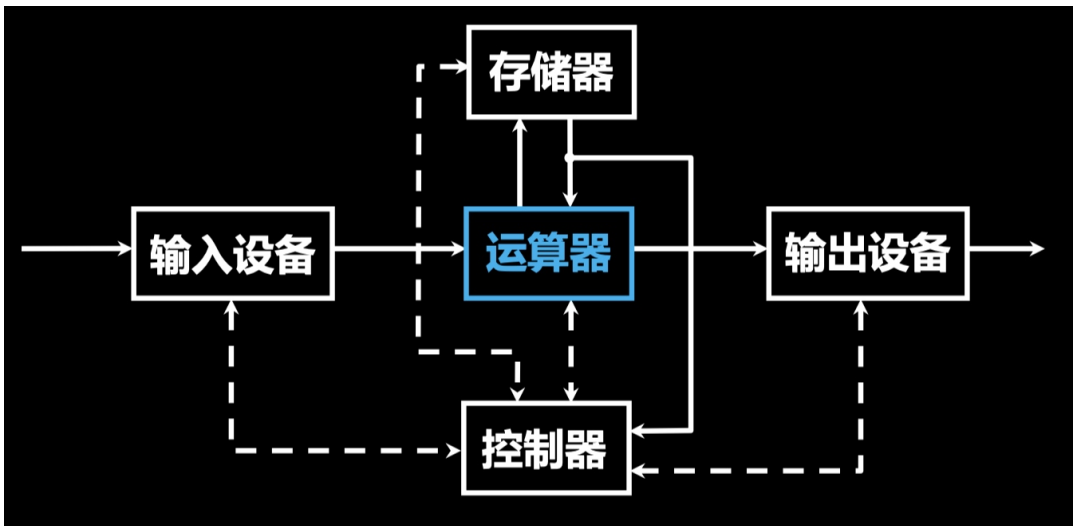
> 透明性

- 本来存在的事物或属性，从某个角度去看，却好像不存在。
- 如硬件特性对C语言程序设计者而言具有透明性。

> 系统观

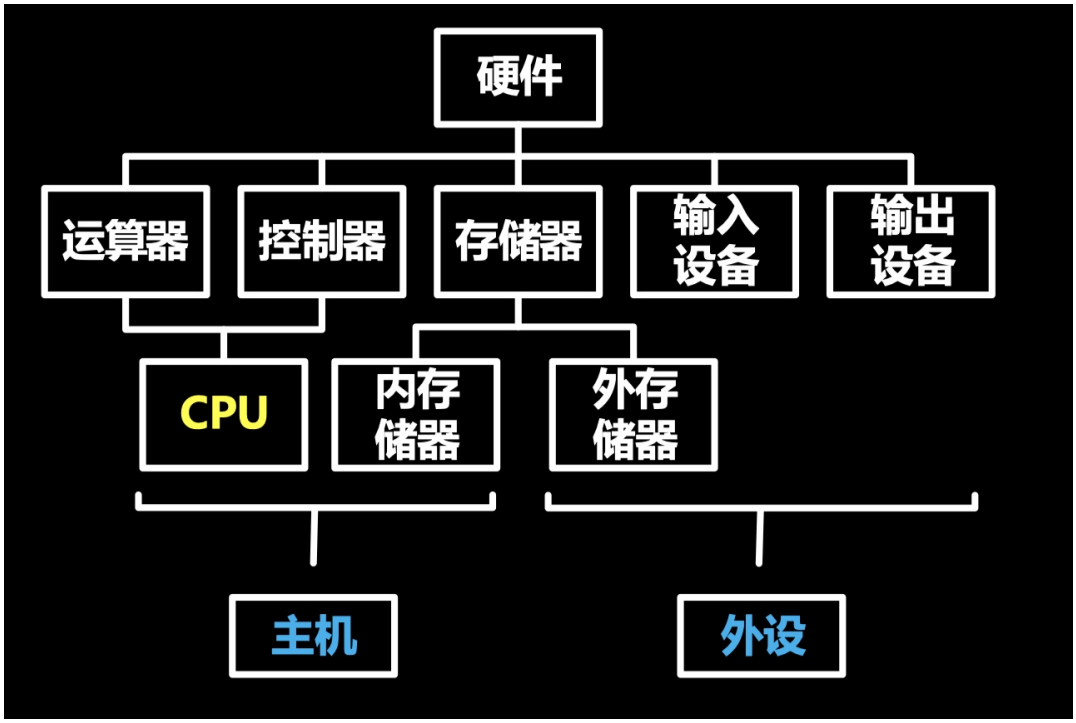
- 当硬件结构发生变化时要想到可能对软件产生的影响。

1.2.6 冯·诺伊曼计算机的工作过程

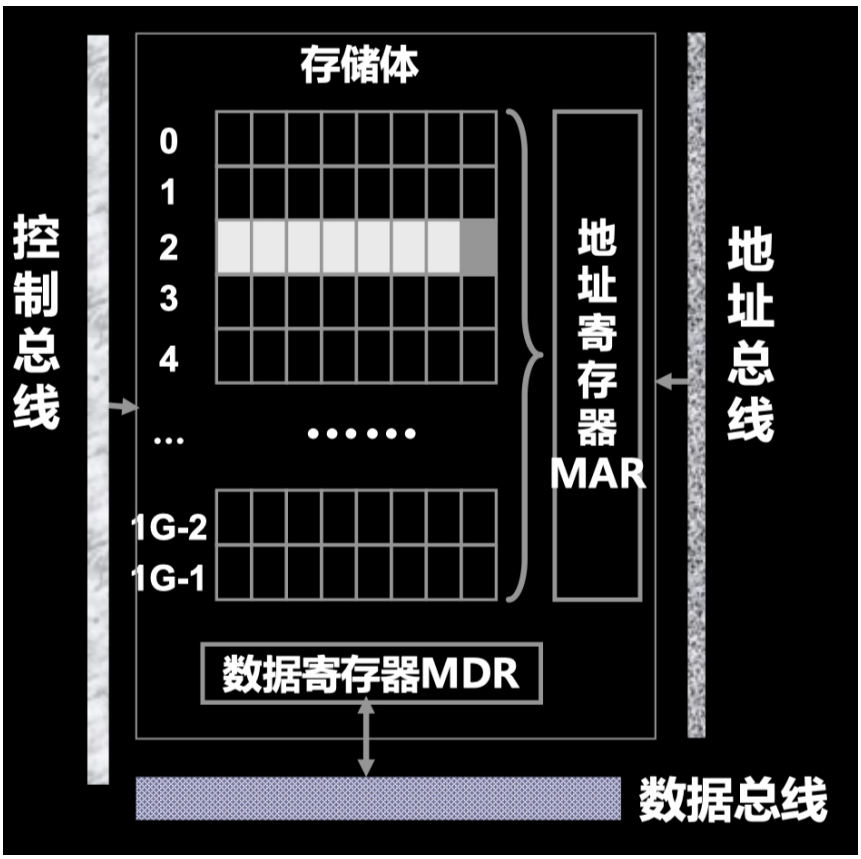


1. 控制器的控制下，输入设备输入程序和数据，并放入存储器。
2. 执行程序时，控制器到存储器中取指令，并执行这条指令。
3. 执行过程中如果涉及计算，则会从存储器中读计算所需数据，并将数据送到运算器中进行计算。
4. 计算的结果送回存储器，或者送到输出设备上进行显示或打印。

> 五大硬件之间的关系



1.2.7 主存储器的构成



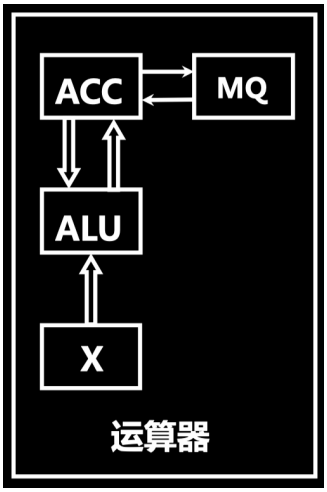
> 主存四级结构

主存储器 - 存储体 - 存储字 - 存储字长

> 主存工作过程

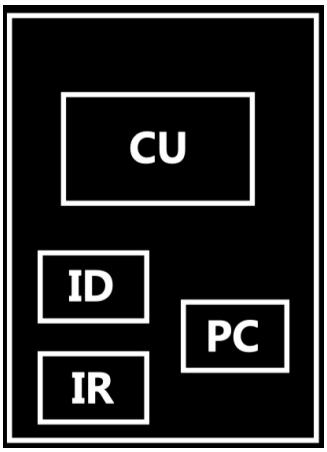
1. 地址通过地址总线发送到 MAR 中，经过地址译码后，找到地址实际单元。
2. 接下来有读、写两种方式访问地址
    - 读操作：主存数据 - MDR - 数据总线
    - 写操作：数据总线 - MDR - 主存数据
3. 读、写方式由控制总线上 read、write 两条信号线所控制。

1.2.8 运算器的组成



ALU：算术逻辑单元，支持算术运算与逻辑运算
ACC：累加器
MQ：乘商寄存器
X：操作数寄存器

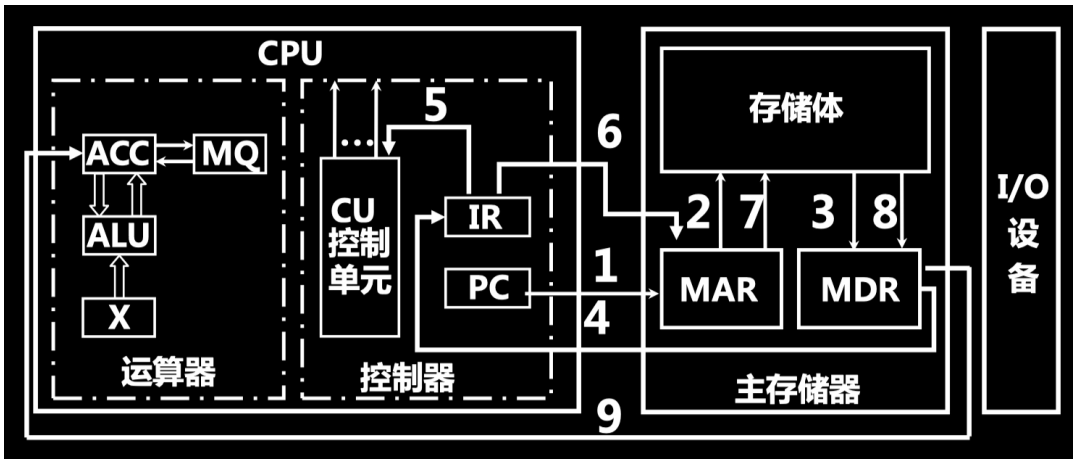
1.2.9 控制器的组成



CU：控制单元，核心部件，功能为根据指令要求发出控制信号
PC：程序计数器（指针），存放指令地址，自动 “+1”
IR：指令寄存器，存放欲执行的指令
ID：指令译码器，分析当前指令的功能

1.2.10 指令执行举例

> 取数指令

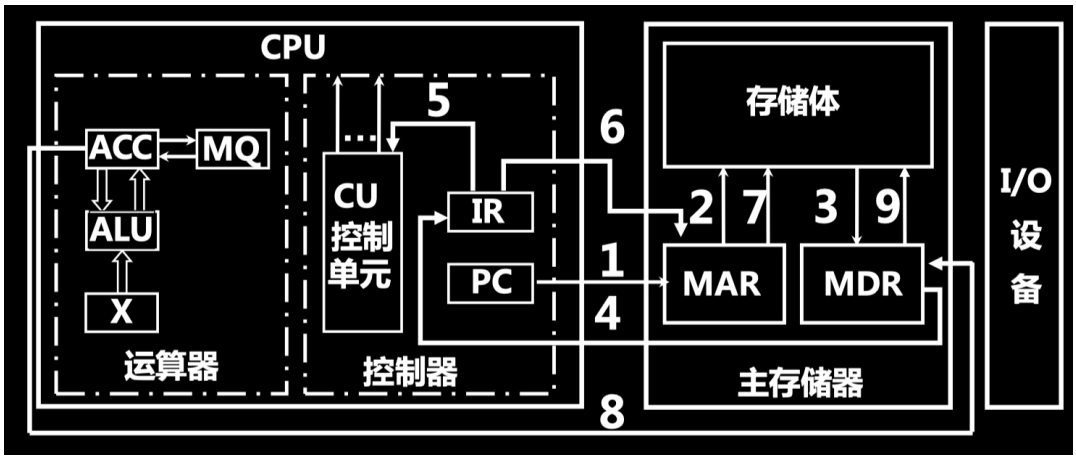


功能：从内存中取得一个数据，并送入CPU累加器中
工作过程：（内存读指令 -> 译码分析 -> 取数）

1. 程序计数器发出取址地址，将地址交给MAR
2. 经过地址译码，找到对应指令位置
3. 把指令取出交给MDR

4. 通过数据总线，把数据从MDR传到IR
5. 把指令操作码部分交给控制单元 ---------- 取指令
6. PC把取数地址给MAR
7. 在存储体中寻址
8. 将数据交给MDR
9. 通过数据总线，MDR把数据给ACC --------- 存数

> 存数指令



7: 在存储体中找到 MAR 对应的存储单元
8: 将 ACC 中的数据通过数据总线送到 MDR 中
9: 将 MDR 中的数据送到存储体的对应位置中

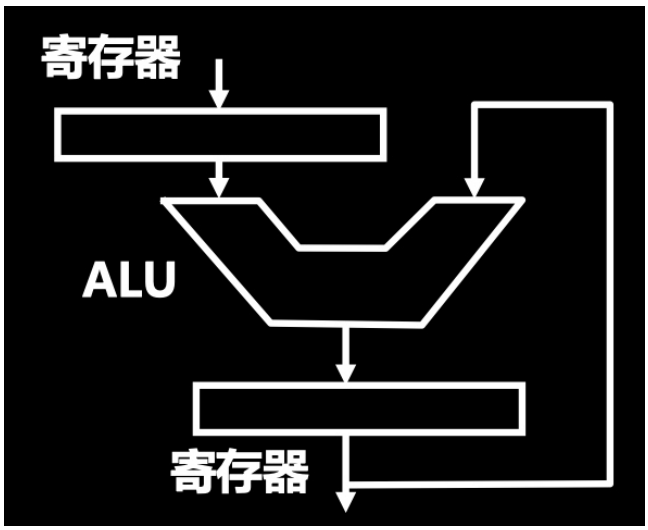
1.3 计算机系统性能指标

1.3.1 机器字长

> 概念

CPU一次能处理数据的位数，通常为 CPU 寄存器的长度。

> 特点



字长越长，数的表示范围越大，精度越高，速度越快，价格越高。

字长可以看成是并行位计算。

1.3.2 存储容量

存储容量分为主存与辅存，其中主存拔电丢失，但辅存不会。

> 主存容量计算

MAR —— 地址线位数，即寻址范围，存储单元个数

MDR —— 单个存储单元位数，即字长

存储单元个数 × 存储字长

MAR	MDR	容量
10	8	1k×8位
16	32	64k×32位

1.3.3 速度

> 速度衡量方法

1. 主频
2. 执行一次加法或乘法所需时间
    - 计算机发展早期所使用的方法
3. 吉普森法
    - $$T_M = \sum_{i=1}^{n} f_i t_i$$
    - T_M：机器的运行速度
    - n：该计算机指令系统中指令的总数
    - f_i：程序在实际执行过程中执行到第 i 条指令的概率
    - t_i：执行第 i 条指令所需时间
4. MIPS：每秒执行的百万条指令数
    - 一般指定点指令的数目
5. FLOPS：每秒执行的浮点指令数

6. CPI：平均每条指令所需时钟周期数
    - CPI 越小，运行速度越快

第二章 系统总线

2.1 总线概述

2.1.1 总线的基本概念

> 采用总线的原因

1. 用总线连接代替分散连接，线路更简单，成本更低
2. 采用同一总线标准，使电路规模更易扩展

> 总线概念

总线是$key^{(1)}$连接多个部件的信息传输线，是各部件共享的传输介质。总线由$key^{(2)}$许多传输线或通道构成，每条线可以$key^{(3)}$传送一个二进制位。

> 总线特点

总线上有超过一个信息接收部件，且在任一时刻，只允许有一个部件向总线发送信息。

- 一个人说多个人听

> 常见总线分类

片内总线、系统总线、通讯总线

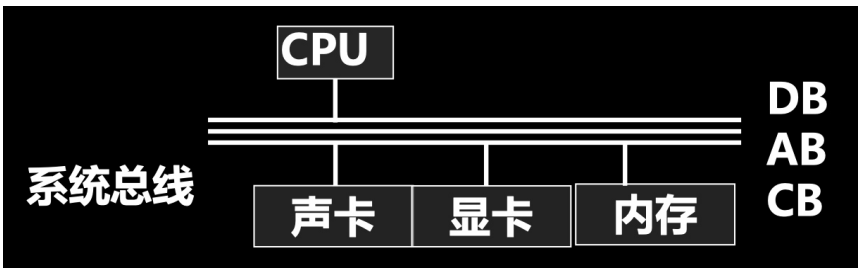
2.1.2 系统总线

> 概念

系统总线主要指处理器与主存、I/O设备等部件之间的信息传输线

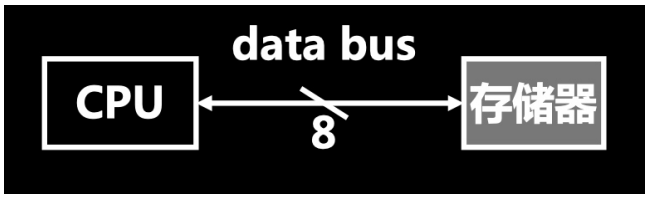
> 三总线结构

数据总线（DB）、地址总线（AB）、控制总线（CB）



> 数据总线

- 数据总线的位数称为数据总线宽度，即为机器位数，与机器字长、计算机位数、数据总线宽度、存储字宽度、寄存器长度均相同。
- 数据总线是双向传输的总线。可读可写。



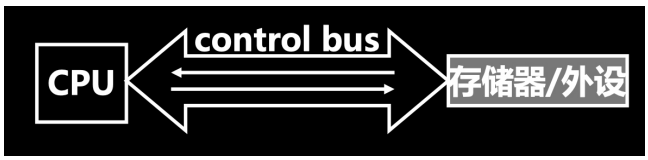
> 地址总线

- 地址总线用来标识主存或 I/O 设备上存储单元的位置
- 地址总线是单向总线。总是由 CPU 发出，标识存储器上的一个地址。

- 地址总线的位数与存储单元的个数有关，与存储单元的二进制位长度无关。如地址总线有 n 条，则最多可以寻址到 2^n 个存储单元。

> 控制总线

- 控制总线是用来发出各种控制信号的传输线。
- 单一控制线通常是单向的，但控制总线总体来说是双向总线。
    - 写、读信号线均为单向，但组合成的控制线是双向的
- 典型控制线有复位、时钟、中断相关、总线请求、存储器读写、I/O 读写、忙闲检测等。可以说，在计算机中，除了数据总线和地址总线之外的其他所有信号线均属于控制总线。



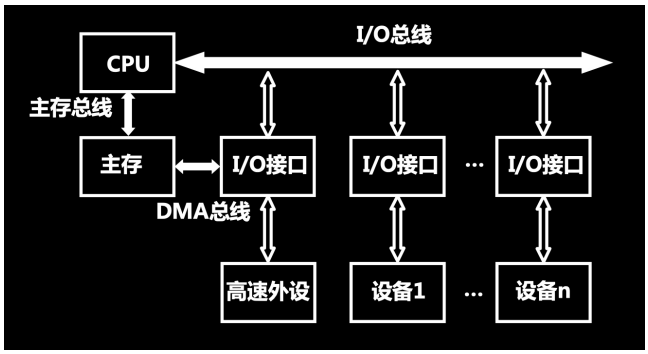
> 典型系统总线结构

DMA 总线：

- 没有 DMA 总线时，设备向主存传输数据，需要通过 I/O 总线到CPU，再到达主存。这个过程中，CPU不得不停下来处理I/O，因此非常浪费资源。
- 因此专门建立一条 DMA 总线（直接存储器存取），通过硬件方式进行数据块传输，不再需要 CPU 的过多干预。

DMA 控制器工作过程：

- CPU 通知 DMA 控制器，你到哪一个外设的什么位置去取多长的一个数据块，然后把它送到内存的什么位置。

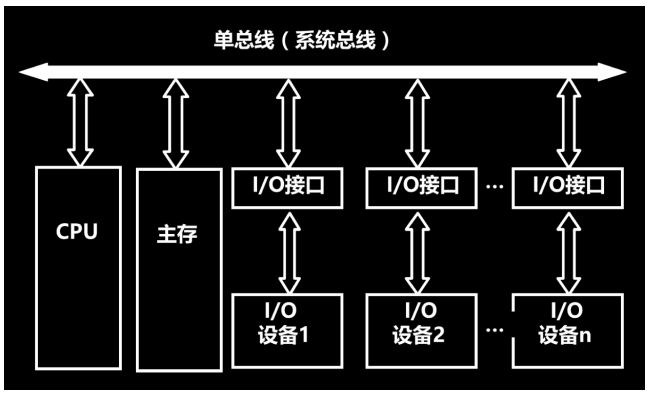


2.2 总线结构

2.2.1 单总线结构

I/O设备与主存交换信息原则上不会影响CPU，便于提高CPU效率。但是，所有部件连到单一总线，提高冲突率，影响整机性能。（PDP-11、DJS183）

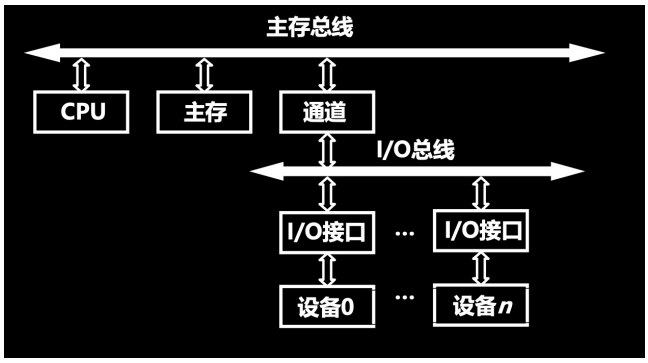
优缺点：结构简单、便于扩充、传输速率低、容易形成瓶颈。



2.2.2 双总线结构

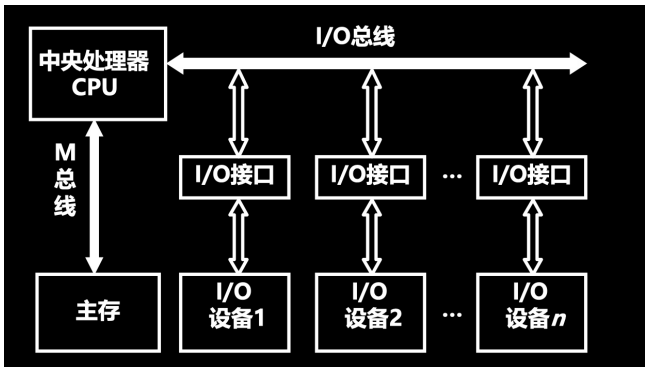
> 双总线结构

将速度较低的设备从主存总线上分离出来，形成主存总线与I/O总线分开的结构。通道是一个具有特殊功能的处理器，负责对I/O统一管理。



> 面向 CPU 的双总线结构

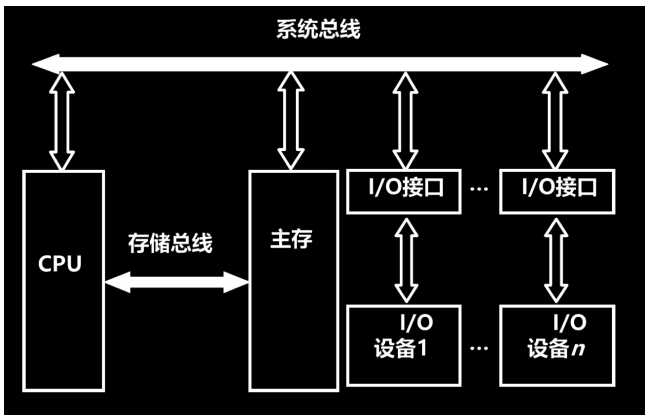
M 总线：存储总线
I/O 总线：输入输出总线



优点：增加I/O设备方便
缺点：I/O设备与主存交换信息需要通过CPU，会影响CPU效率

> 面向存储器的双总线结构

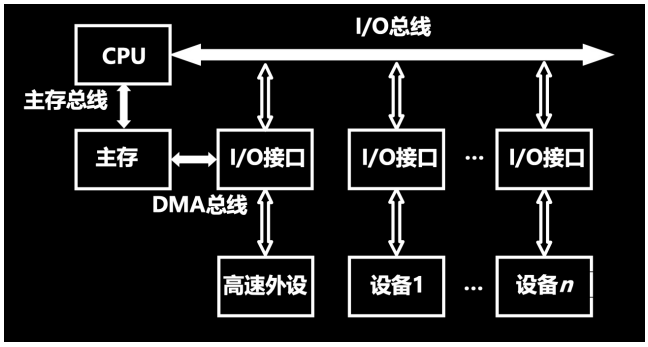
增加CPU和主存之间专用的存储总线，提高传输速度，减轻系统总线负担，保留了主存与I/O直接交换信息的特点。



2.2.3 三总线结构

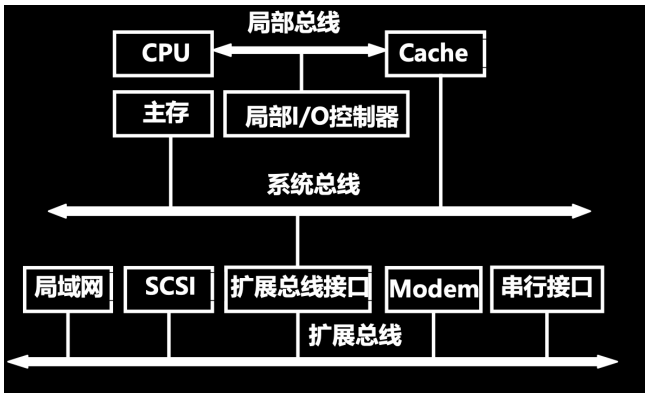
> 三总线结构 1

主存总线用于 CPU 和主存交换信息，I/O总线用于CPU和I/O设备间传送信息，DMA总线负责在主存和I/O设备间直接传递信息。



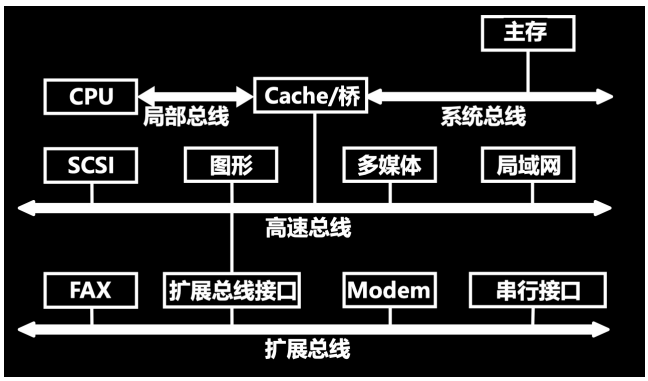
> 三总线结构 2

处理器采用局部总线连到 cache，再利用系统总线连接主存，通过扩展总线接口连接其他接口。



2.2.4 四总线结构

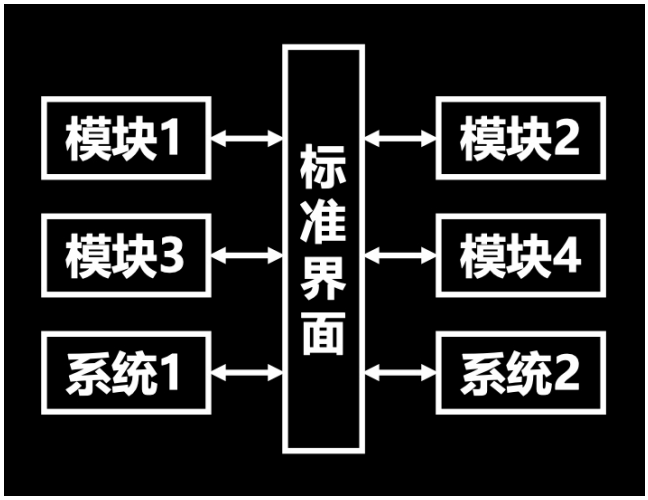
系统把设备总线分成两个层次，高速设备连接高速总线，低速设备连接扩展总线，各总线直接由桥连接。



2.3 总线标准

2.3.1 概念

总线标准，即系统与模块、模块与模块之间的一个互连的标准界面，能够隐藏符合标准的部件内部的操作细节。



模块、系统之间无需知道对方的实现细节。

2.3.2 常见总线标准

- ISA/AT 总线
- EISA 总线
- VESA（VL-BUS）总线
- PCI 总线
- AGP 总线
- RS-232C 总线
- USB 总线

2.4 总线分类

2.4.1 片内总线

芯片内部总线，例如运算器和 cache 之间的总线。

2.4.2 系统总线

系统总线也称为板级总线。处理器与主存、I/O等部件之间的信息传输线。

系统总线中常见的三总线结构：数据总线、地址总线、控制总线

2.4.3 通信总线

> 概念

计算机系统之间，或计算机与其他设备之间的信息传输线。

> 特点

类别繁杂，连接规格、传输距离、速度、工作模式各不相同。传输速度和距离成反比。

> 串行通信

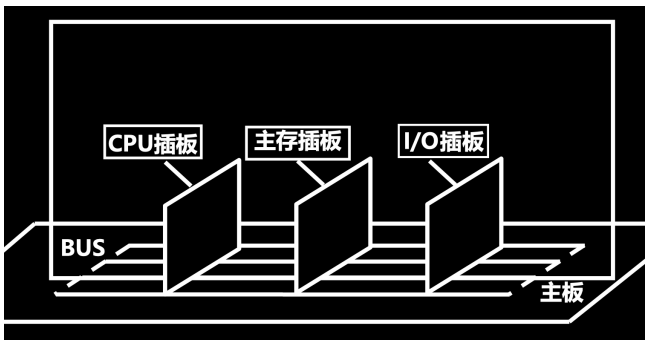
- 数据在单条 1 位宽的传输线上一位一位按顺序依次传送。
- 适宜远距离数据传送，可从几米到几千千米，成本低。

> 并行通信

- 数据在多条 1 位宽的传输线上并行传送，同时由源传送到目的地。
- 适宜近距离的数据传送，通常小于 30 米。
- 短距离低时钟频率情况下，传输速度远快于串行方式。

2.5 总线特性及性能指标

2.5.1 总线物理实现



2.5.2 总线特性

> 机械特性

物理尺寸、插头形状、管脚数、排列顺序

> 电气特性

信号线的电平范围。逻辑 “1”，逻辑 “0”。

TTL电平、CMOS电平、RS-232C

> 功能特性

每根传输线的功能：数据、地址、控制

> 时间特性

信号的前后时序关系。

2.5.3 总线性能指标

- 总线宽度：数据线宽度
- 总线带宽：数据传输速率
- 时钟同步方式：同步、异步
- 总线复用：地址、数据、控制线复用
- 信号线数：三总线所有信号线总数
- 总线控制方式：突发工作、仲裁方式
- 其他指标：带载能力、电源电压

2.6 总线判优逻辑

2.6.1 基础概念

> 总线事务

在总线上一对设备之间的一次信息交换过程，称为一个 “总线事务”。

总线事务一般包括地址阶段和数据阶段。

> 主设备

对总线有控制权，可以发起总线事务的设备。

> 从设备

只能响应总线上命令的设备。

> 典型的总线事务

1. 存储器的读写
2. I/O读写
3. 中断响应

> 突发传送事务

一个地址阶段和多个数据阶段。

- 给定 1 个地址，让 10 个元素从该地址往后依次排列，即可实现 1 个地址，存入 10 个数据。

2.6.2 总线判优

> 概念

当多个主设备同时申请使用总线时，总线判优逻辑电路按照一定的优先级顺序来确定哪个主设备可以使用总线。

> 总线判优的三个因素

1. 等级性
    - 各设备应该有优先级
2. 公平性
    - 不能有设备饿死
3. 响应速度
    - 设备提出总线需求时，应尽快给予满足

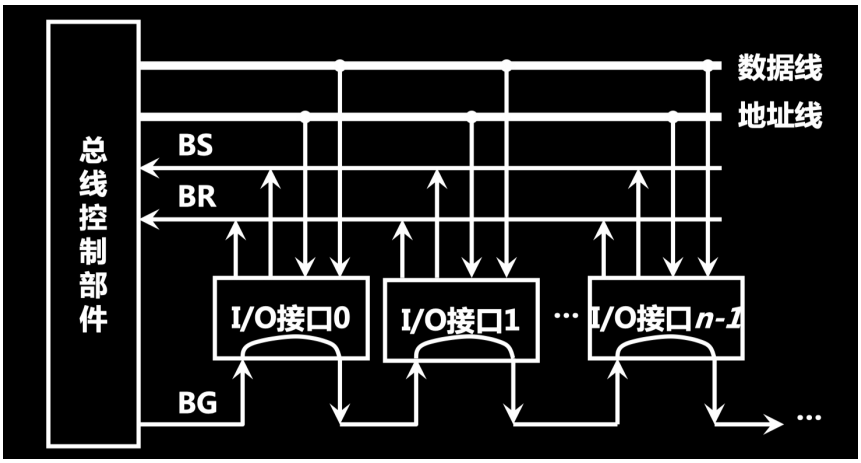
> 判优逻辑分类

1. 分布式
    - 总线判优逻辑分散在各个设备的接口电路中
    - 包括：自举分布式、冲突检测分布式、并行竞争分布式

2. 集中式
    - 总线判优逻辑是集中在一起的，一般在CPU内部
    - 包括：链式查询、计数器定时查询、独立请求方式

2.6.3 链式查询方式

- 左边总线控制部件为总线判优逻辑
- BS：总线忙
- BR：总线请求
- BG：总线同意



> 工作流程

优先选择靠前的接口，优先级为距离。

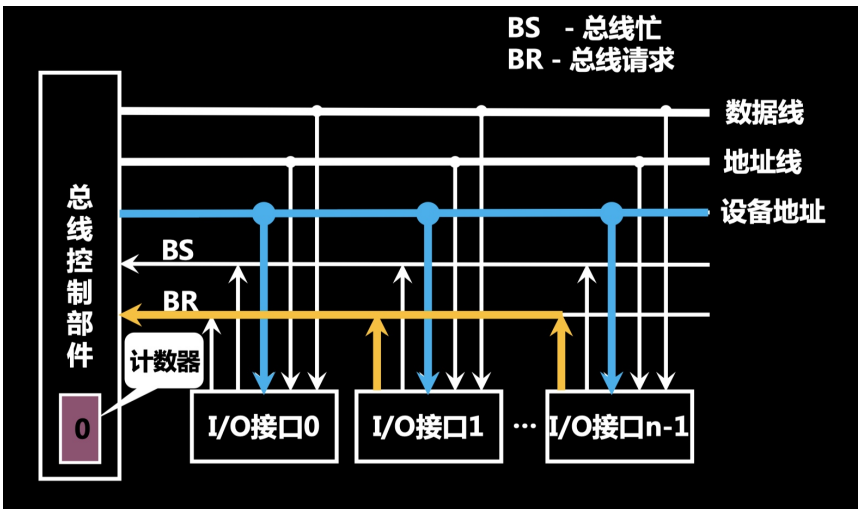
1. 申请总线，BR = 1
2. 逻辑部件依次向后传递 BG 信号，若发现其中一个接口申请了总线，则令其获得总线
3. 接口获取总线后不再向后传递BG信号，设置BS = 1

> 特点

1. 自动支持优先级
2. 连接简单，易于扩充设备
3. 对电路故障非常敏感，优先级低的设备获得总线难度很大
    - 若0号设备故障，则BG信号无法向后传递

2.6.4 计数器定时查询方式

用计数器来进行挨个查询，放弃了原有的BG信号，但优先级仍为距离。



> 工作流程

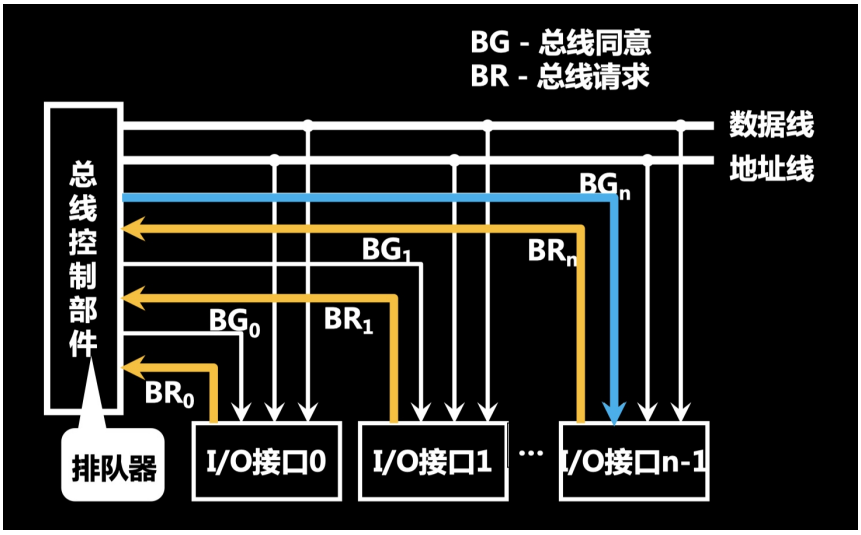
1. 申请总线，BR = 1
2. 逻辑部件开启计数器，从0开始递增。例如当计数器数值为2时，就在总线上查询2号设备，如果2号设备发出了请求，则得到总线。
3. 接口获取总线后将设置BS = 1，之后再次请求需要重新从0开始计数。

> 特点

1. 对故障不敏感
2. 增加了设备地址线，增加了控制的复杂性。设备地址线的数量为 $log_2 n$。

2.6.5 独立请求方式

放弃 I/O 总线，将设备统一连到总线控制部件上。



> 工作过程

1. 申请总线，BR = 1
2. 控制部件收到 BR 请求后，放入内部的排队器中进行排序，将优先级最高设备的 BG 设为

1。

> 特点

1. 响应速度快
2. 优先级次序控制灵活
3. 控制线数目太多，达到了 $2*n$ 条线。

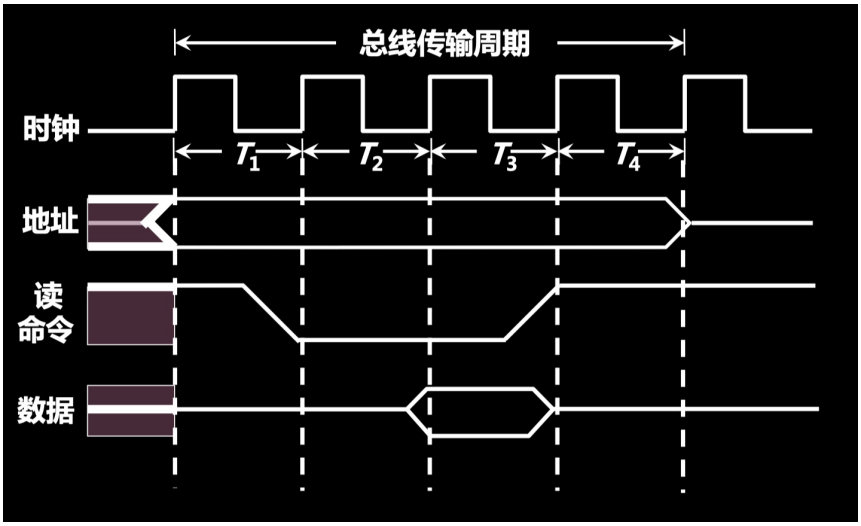
2.7 总线的通信控制

2.7.1 总线四种通信方式

1. 同步通信：发送时钟信号来同步数据传送
    - A把苹果给B，A拍一下B，B就拿走苹果，"拍一下" 就是一个时钟信号
2. 异步通信：没有时钟，采用握手信号同步
    - A把苹果给B，不断询问B是否可以拿苹果，是否吃完了苹果等问题
3. 半同步通信：同步、异步结合，插入等待周期
4. 分离式通信：传输周期划分为功能独立的子周期

2.7.2 同步通信

> 同步读

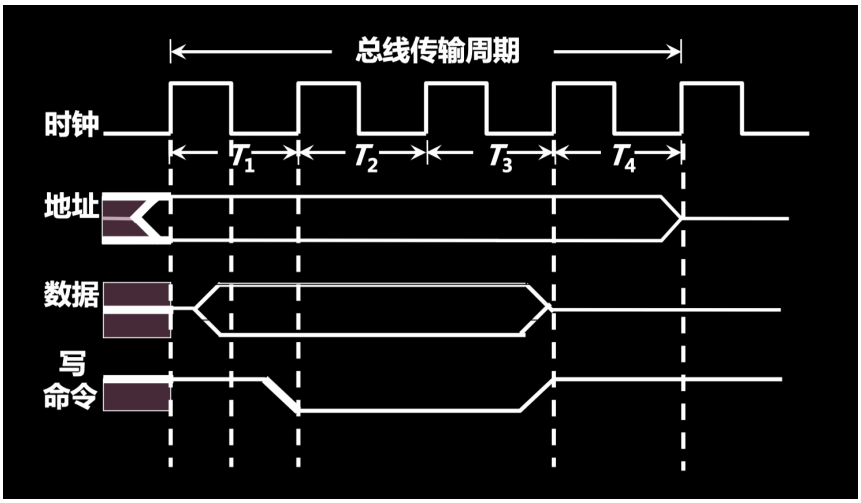


命令均在时钟周期上升沿发出。

1. T_1 周期：CPU发出读地址。
2. T_2 周期：CPU发出读命令。有了地址与读命令后，内存开始内部寻址找到数据。
3. T_3 周期：内存将数据准备好放到了数据总线上。CPU到数据总线上读走数据。

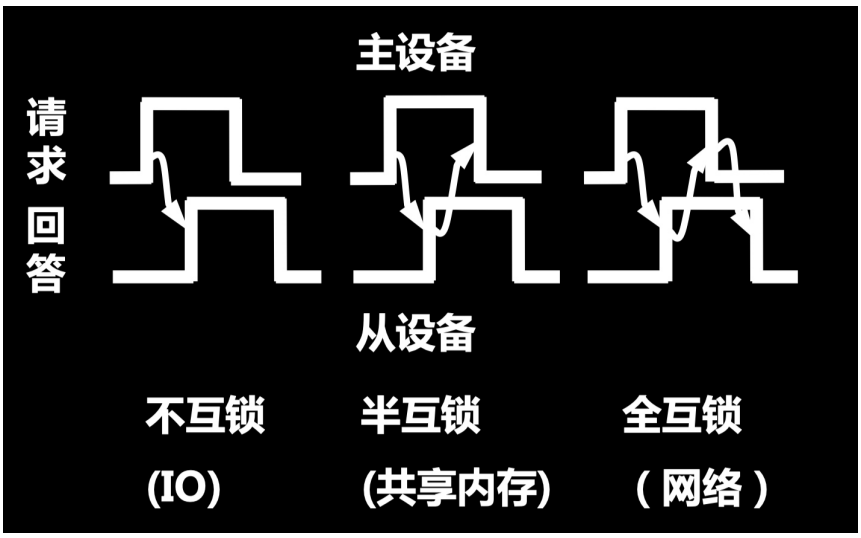
4. T_4 周期：内存撤销数据总线上信号，CPU撤销读信号，但为了保证数据传输过程准确无误，地址信息多保留一个周期。

> 同步写



1. T_1 周期：上升沿CPU发出读地址，下降沿CPU将数据送到数据总线上。
2. T_2 周期：上升沿CPU发出写命令。
3. T_3 周期：在 $T_2 \sim T_3$ 周期，内存将数据写入内存单元。
4. T_4 周期：上升沿撤销数据信号（数据总线）与写命令（控制总线）。为了保证数据能够完整写入内存中，地址总线上地址信号总要多保留一个时钟周期。

2.7.3 异步通信



> 不互锁

过程：

1. 主模块发出信号，经过一段时间即撤销，不等待接收从模块信号
2. 从模块收到请求信号，向主模块发出应答信号，经过一段时间即撤销

特点：

1. 双方均不存在互锁关系
2. 适用于非常可靠的通信场合中，如 I/O 通信。

> 半互锁

过程：

1. 主模块发出信号，必须等收到从模块应答后才撤销
2. 从模块收到请求信号，向主模块发出应答信号，经过一段时间即撤销

特点：

1. 主模块具有互锁关系，从模块不具有
2. 适用于共享内存。

> 全互锁

过程：

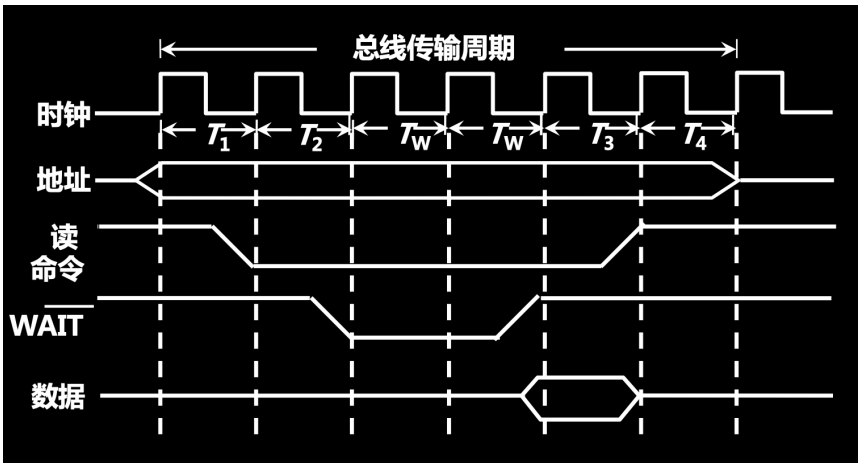
1. 主模块发出信号，必须等收到从模块应答后才撤销，并向从模块发出应答。
2. 从模块收到请求信号，向主模块发出应答信号，必须等收到主模块的应答才撤销。

特点：

1. 主、从模块均具有互锁关系
2. 适用于网络中的数据传送。

2.7.4 半同步通信

- 发送方仍然发送时钟。
- 接收方根据自身情况反馈信息给发送方，使发送方进行响应调整，增加一条等待响应信号线（Wait）
- “主体同步，细节异步”
- 半同步通信允许不同速率的设备协调工作。



> 工作过程

1. T_1：上升沿发出地址
2. T_2：上升沿发出读命令
3. T_W：内存没有准备好，WAIT=1
4. T_3：内存准备好了，CPU读取数据
5. T_4：上升沿释放数据与读命令，一个时钟周期后再释放地址

> 上述三种通信方式总结

以读命令为例，总线周期总是分成这三个过程，而第二个过程不占用总线却需要CPU等待，非常低效，因此我们可以进行改进。

主模块发出地址、命令	占用总线
从模块准备数据	不占用总线
从模块发出数据	占用总线

2.7.5 分离式通信

> 方式改进

每个总线传输周期分为两个子周期。

1. 子周期 1：主模块申请占用总线，使用完后即放弃总线的使用权。
2. 子周期 2：从模块申请占用总线，将各种信息送到总线上。

因此在从模块准备数据的过程中，主模块就可以去从事别的事情，而不是一味的忙式等待。

充分挖掘系统总线每个瞬间的潜力。

> 特点

1. 各模块有权申请占用总线。
2. 采用同步方式通信，省去握手开销。
    - 各模块清楚其它模块所需工作时间，因此采用同步方式即可，无需进行握手确认。
3. 各模块准备数据时，不占用总线资源。总线无空闲等待时间，提高了总线利用率。

第三章 存储器

3.1 半导体存储器概述

3.1.1 半导体存储芯片基本结构

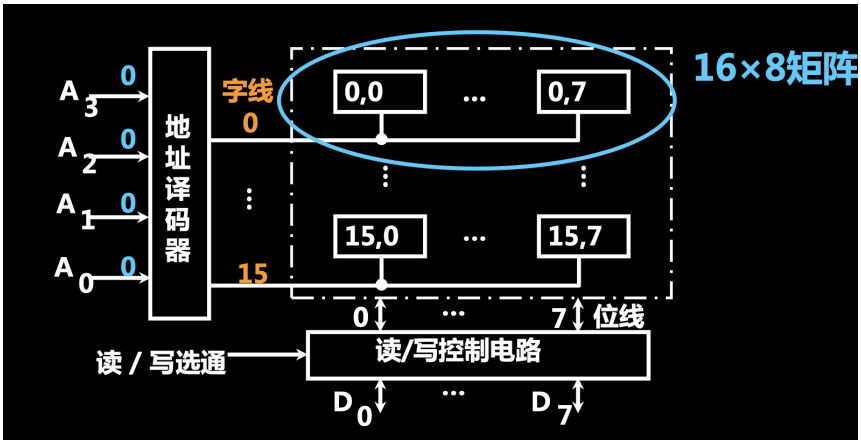
其中片选信号是 $\overline{CE}$ 或 $\overline{CS}$，表示 chip select 或者 chip enable。

读/写控制线中 $\overline{WE}$ 表示低电平写，高电平读；$\overline{OE}$ 表示允许读，$\overline{WE}$ 表示允许写。

3.1.2 半导体芯片的译码驱动

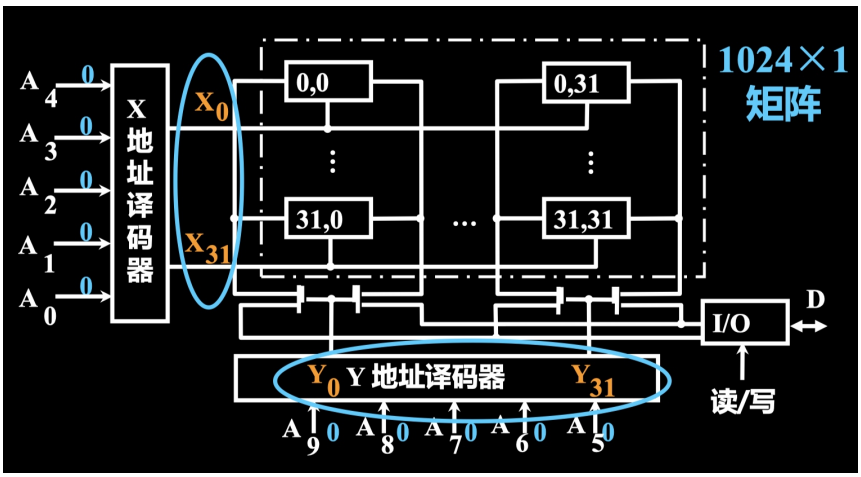
> 线选法

用一根字选择线直接选中存储单元，例如一个字节。



每个字都对应着一根地址线。

> 重合法



规模从 $O(n)$ 缩减为 $O(\sqrt{n})$，二维片选，减少了地址译码输出线的数目。

3.1.3 主存的技术指标

> 存储容量

- 存储容量：主存能存放的二进制数的总位数
- 存储器容量 = 存储单元个数 * 存储字长

> 访存周期

相邻两次存储器访问的时间间隔

> 存储器带宽

单位时间内存储器存取的信息量（字节/秒、位/秒、字/秒）

3.1.4 存储器分类

> 按存储介质分类

1. 【易失】半导体存储器：TTL、MOS
    - 体积小、功耗低、存取时间短、易失
2. 【非易失】磁表面存储器：磁盘、磁带、磁鼓
3. 【非易失】磁芯存储器：硬磁材料的环状元件
4. 【非易失】光盘存储器：激光、磁光

> 按数据保存方式分类

1. 随机存储器（RAM）
    - 易失性
    - 静态随机存储器、动态随机存储器
2. 只读存储器（ROM）
    - 非易失性

RAM 和 ROM 是一种分类，常见的还有一种分类是 Flash 和 Hard Disk。其中 Flash 指 Solid State drives（SSDs），Hard Disk 指硬盘。

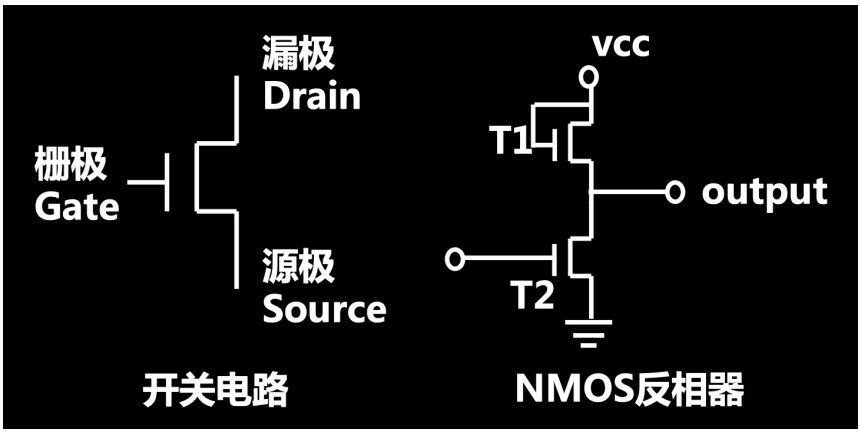
> 按数据存取方式分类

1. 直接访问
    - 访问时间不随访问位置而变化
    - 内存
2. 串行访问
    - 访问时间随访问位置而变化
    - 磁带
3. 部分串行访问
    - 介于上述二者之间
    - 磁盘（寻道直接，等待串行）

3.2 静态随机存储器（Cache）

3.2.1 MOS 管

MOS 管工作在开关状态：饱和导通 & 截止



> 开关电路

在栅极加一个高电平时，MOS管就导通，低电平则截止。

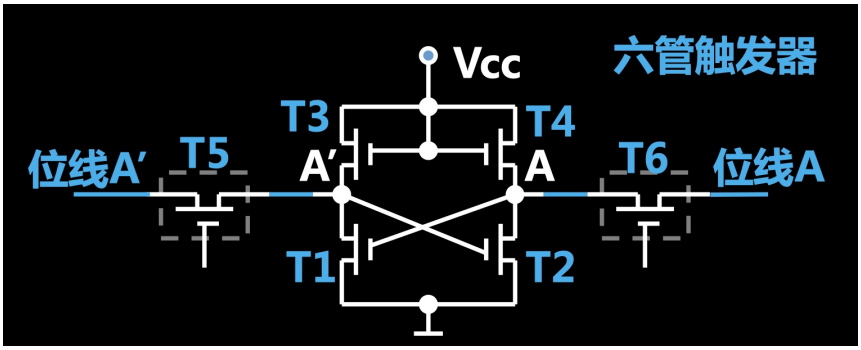
> NMOS 反相器（非门）

两个MOS管串联而成，T_1 总是导通状态，且 T_1 导通内阻是 T_2 的 100 倍以上。

1. T_2 为高电平，T_2 导通。由于分压，T_1 分得大量电压，因此输出引脚变为低电平。
2. T_2 为低电平，T_2 截止，其电阻变得非常大，因此 T_1 分压极少，输出引脚变为高电平。

3.2.2 基本单元电路

> 基本单元电路



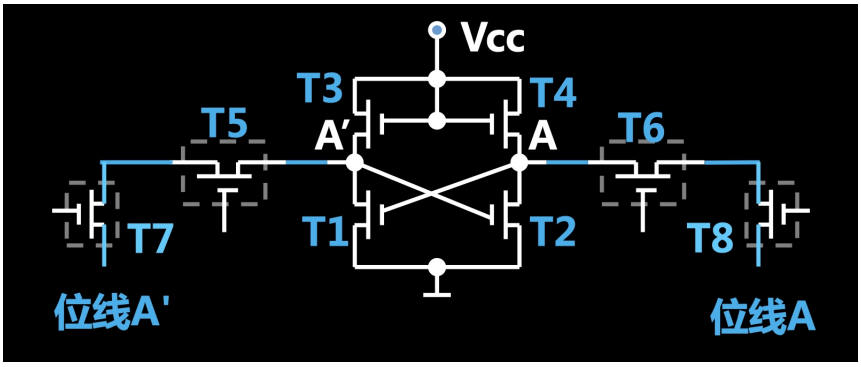
上述单元电路中，$\{T_1、T_2\}$，$\{T_3、T_4\}$，$\{T_1、T_3\}$，$\{T_4、T_2\}$ 均构成了一对 NMOS 反相器。

1. 若 A 为低电平，则 T_1 截止，A' 为高电平，则 T_2 导通，A 为低电平。形成了互相制约的一种关系。
2. 若 A 为高电平，则如上述分析一致。

> 基本单元电路特性

1. $T_1 \sim T_4$ 构成互锁的触发器，A 和 A' 点的电位总相反，A 点电位定义为二进制位的值。高电平则保存了逻辑 1，反之为逻辑 0。
2. T_5 与 T_6 也相当于一把锁，控制内部数值与外部的联系。
   - 当 T_5、T_6 导通时，可通过位线改变其值。
   - 当 T_5、T_6 截止时，只要不掉电，状态一直保持。
3. 静态随机存储器靠寄存器来存储信息，因此不需要刷新。

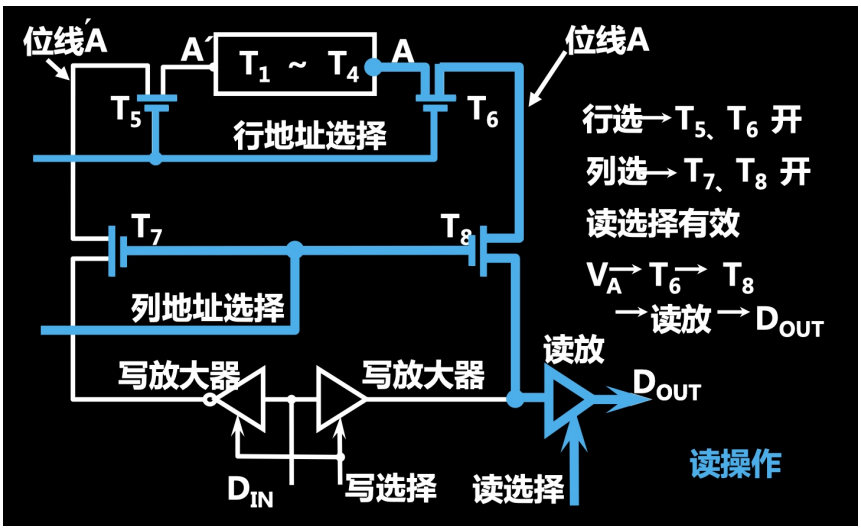
> 基本单元电路扩展



在 T_5 与 T_6 外面再加两把锁 T_7 与 T_8，则保证除非两把锁都打开，互锁的触发器才可以和外界通信。否则只要不掉电，信息永久保持。

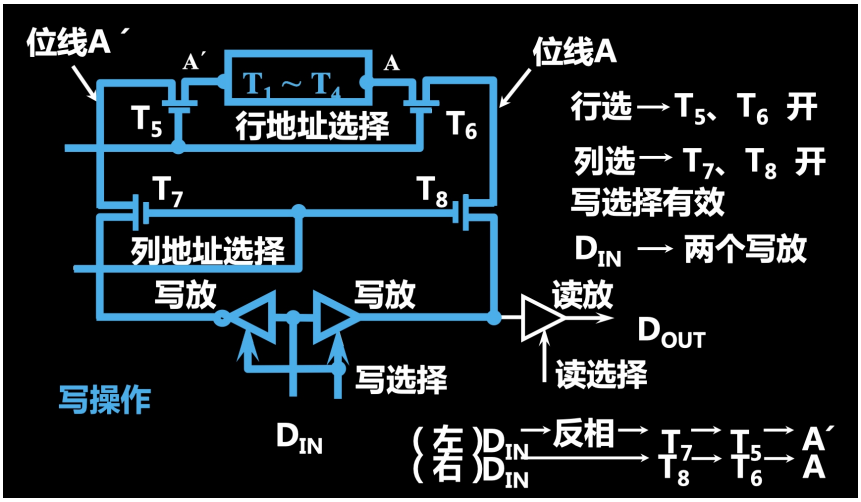
3.2.3 基本单元电路举例

> 读操作



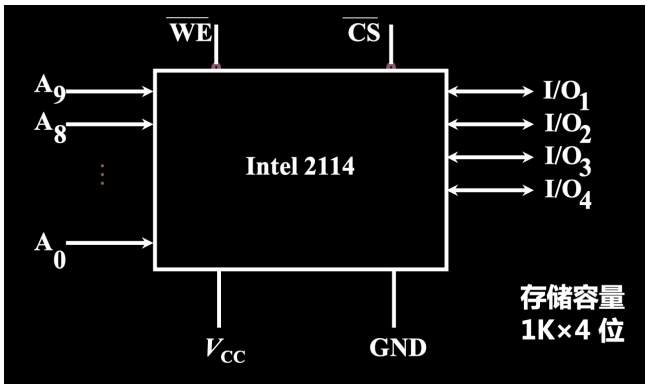
先打开行、列片选信号，再另读选择有效。最后从 T_6 将数据读出。

> 写操作

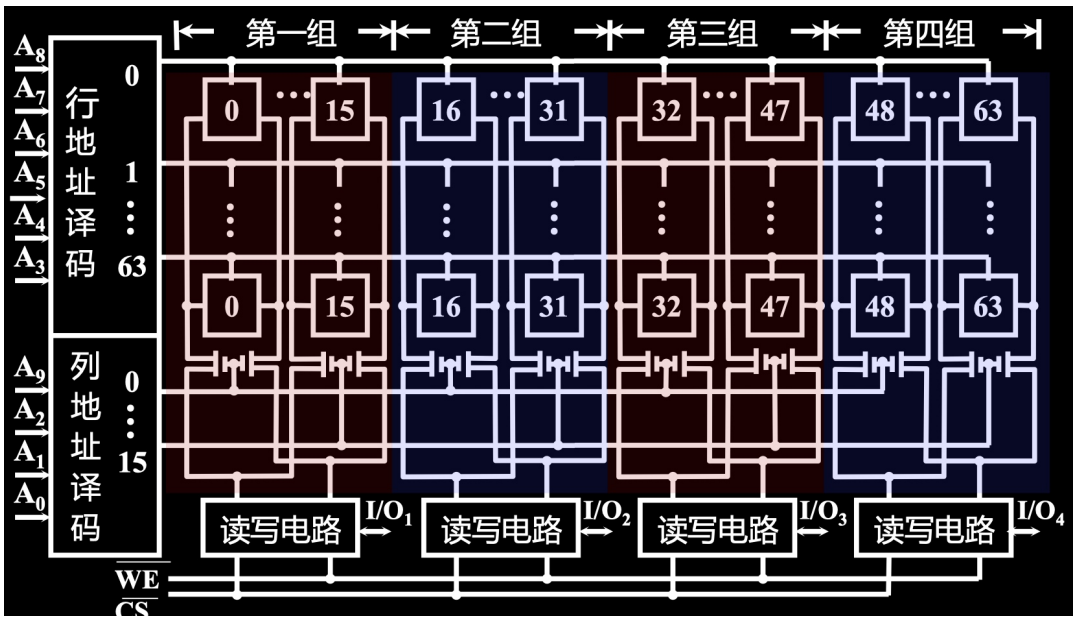


写操作要注意 A 与 A' 均要写，因此左边的写放输出需反向。

> Intel 2114 举例

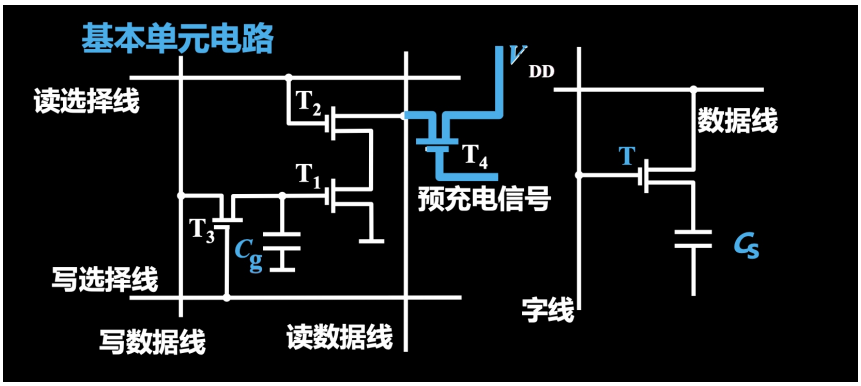


每个方框都是一个六管存储单元。因此行地址译码控制 T_5、T_6，列地址译码控制 T_7、T_8。其中 4096 个存储单元分为四组，每组 64 行，16列，采用重合法译码。



3.3 动态随机存储器

3.3.1 三管动态随机存储器



在上述基本电路中，T_4 充电，电路开始工作。

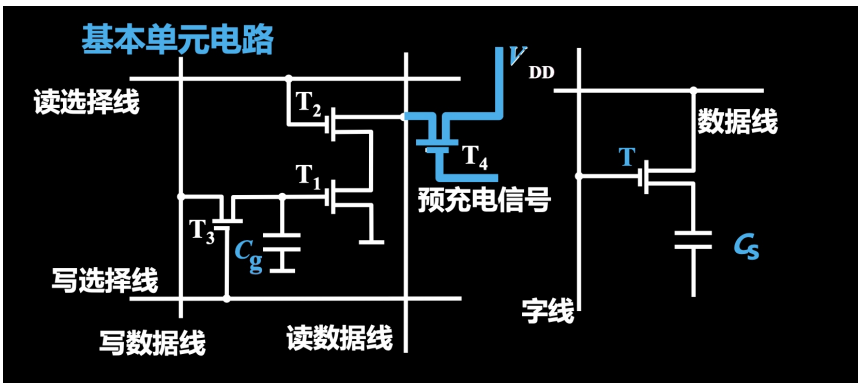
> 读过程

1. 读选择线置1，T_2 导通；读数据线也置1
2. 若 C_g 上为1，则 T_1 导通，读数据线返回 0
3. 若 C_g 上为0，则 T_1 截止，读数据返回 1

> 写过程

1. 写选择线置1，T_3 导通
2. 写数据线置1，则写入1；置0则写入0

3.3.2 单管动态随机存储器



字线置1，表示片选到了这个单元。

> 读操作

1. 电容上有电荷，则数据线上有电流，为1
2. 电容上没有电荷，则数据线上没有电流，为0

> 写操作

1. 数据线置为1，则给电容充电
2. 数据线置为0，则对电容进行放电

3.3.3 动态随机存储器缺陷

1. 上述两种电路，对这个基本单元电路的 “读” 都是一种破坏性的 “读”。
2. 上述两种电路依靠电容充放电来保存信息。而电容存在漏电流，电荷会自动泄漏。

3.3.4 刷新

> 解决方案

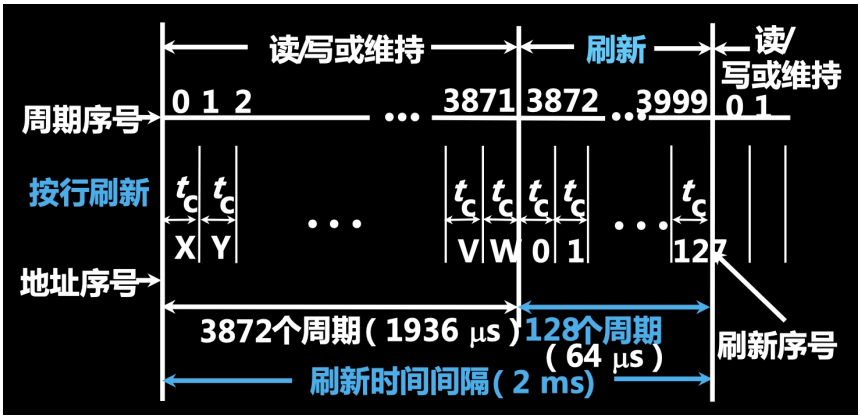
1. 读出放大器
   - 当读取动态随机存储器的一个二进制位时，电容充放电本身的电荷量非常小，因此需要一个读出放大器进行读取和放大。
   - 读出放大器会在读出数据后立刻写回到电容中，解决 “破坏性读出” 的问题。
2. 刷新
   - 电容漏电的时间间隔为 $2mm$，因此在 $2mm$ 内对每一个二进制进行一次读操作，即可保证电容数值不变。

> 集中刷新

集中刷新即是在一个刷新周期内，前面的时间均正常工作，而将最后的时间用于刷新。

下面给出一个例子，假设存取周期为 $0.5us$，128*128的矩阵，按行刷新，需要刷新 128 次。刷新周期为 $2mm$，即可以存取 4000 次，因此我们将 0～3871 存取周期用于正常工作，

3872～3999 这 128 个周期用于刷新。

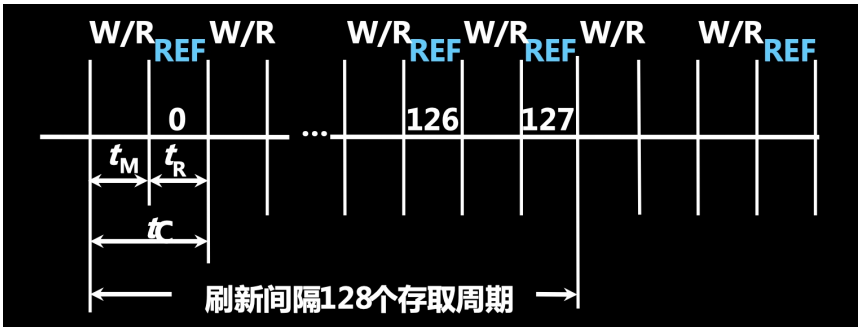


上述例子中，“死区” 为 $0.5us * 128 = 64us$，“死时间率” 为 $\frac{128}{4000} = 3.2\%$。

> 分散刷新

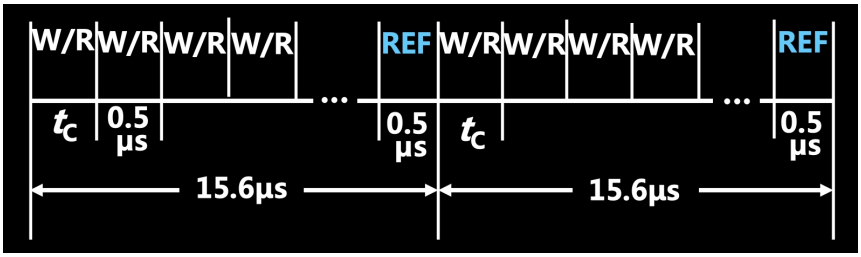
将存取周期 t_c 分为两段，前半段用于工作，后半段用于刷新，即 $t_c = t_M + t_R$，不存在停止读、写操作的死时间，但效率低下。

无时无刻不在刷新，$0.5us$ 用于工作，$0.5us$ 用于刷新，因此每 $128us$ 就全刷新一遍，浪费的效率达到了 50%。



> 异步刷新

$2mm$ 时间内需刷新 128 次，因此每 $15.6us$ 就刷新一行，死区时间仅为 $0.5us$。



指令的执行分别取指、译码、执行，因此如果把刷新安排在指令译码阶段，则不会出现 “死区”。

3.3.5 引脚复用技术

受封装技术的限制，无法保证在有限的芯片面积或周长的前提下引出这么多的引脚，因此动态随机存储器采用了一种引脚复用技术。

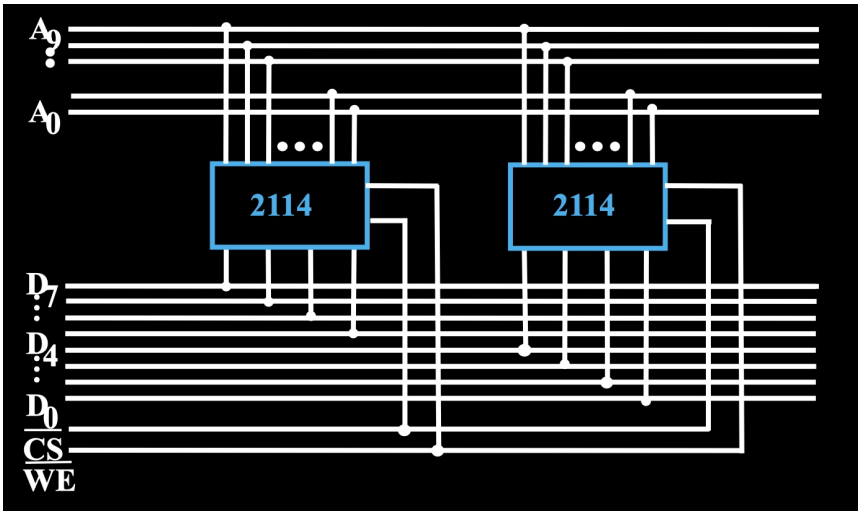
将芯片的地址引脚数目设置为需要数目的一半，分别通过两个时钟周期送入两批地址信号，然后在芯片内部将两批地址信号合并为完整的地址信息再去寻址。

3.4 存储器的扩展方式

3.4.1 字扩展

字扩展即是对于数据线进行扩展，具体扩展如下所示。

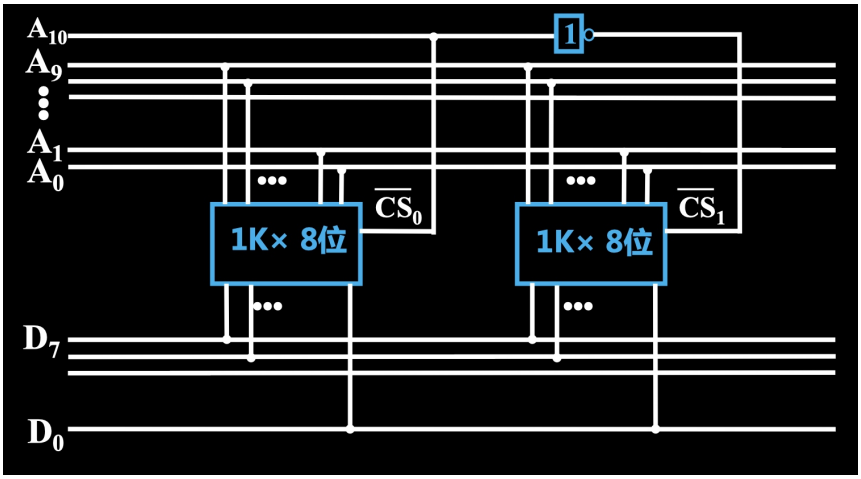
用 $1K*4$ 芯片组成 $1K*8$ 的存储器，对数据线进行扩展。



3.4.2 位扩展

位扩展即是对于地址线进行扩展，具体扩展如下所示。

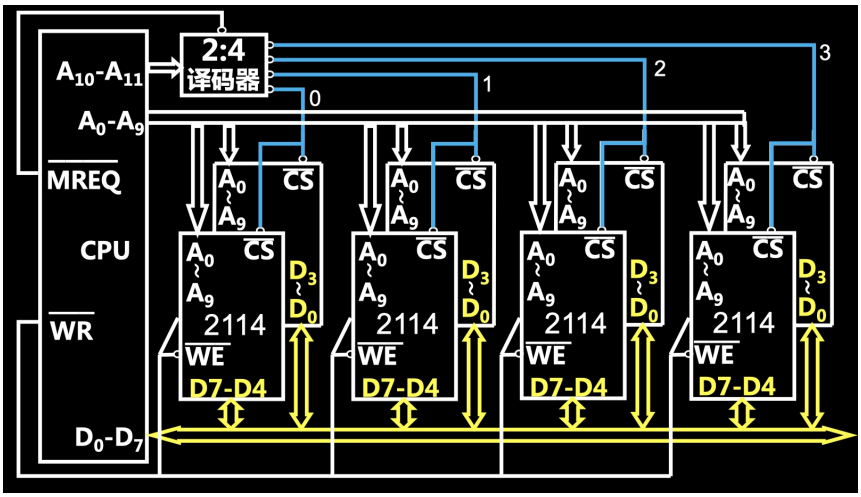
用 $1K*8$ 芯片组成 $2K*8$ 的存储器，对地址线进行扩展。



3.4.3 字、位扩展

字、位扩展即是对数据线和地址线同时进行扩展，具体扩展方式如下图所示。

用 $1K*4$ 芯片组成 $4K*8$ 的存储器，利用 2-4 译码器扩展地址线，同时扩展数据线实现字、位同时扩展。



3.5 Cache 的工作原理

3.5.1 Cache 的提出

> 问题提出原因

1. 解决 CPU 和存储器速度差异的矛盾
2. 解决 CPU 访存优先级低于 I/O 的问题
    - CPU 访存优先级低于 I/O 是因为，如果 I/O 中数据不及时取走，可能就会被冲刷掉。

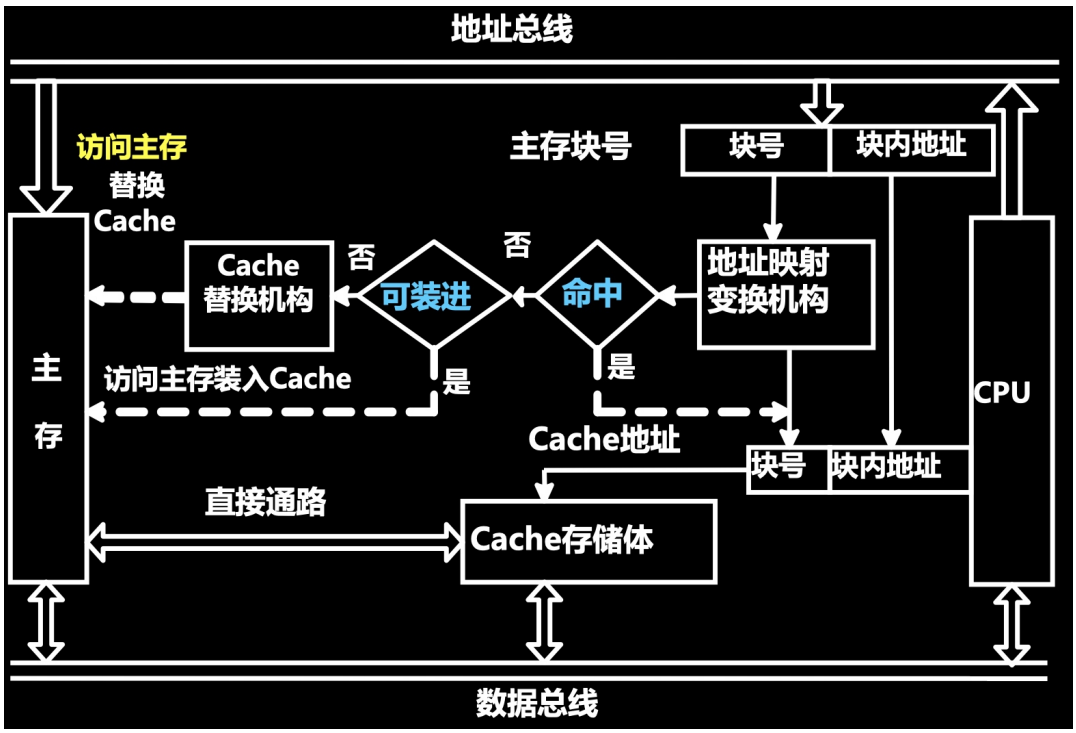
> 理论基础

1. 时间局部性原理
2. 空间局部性原理

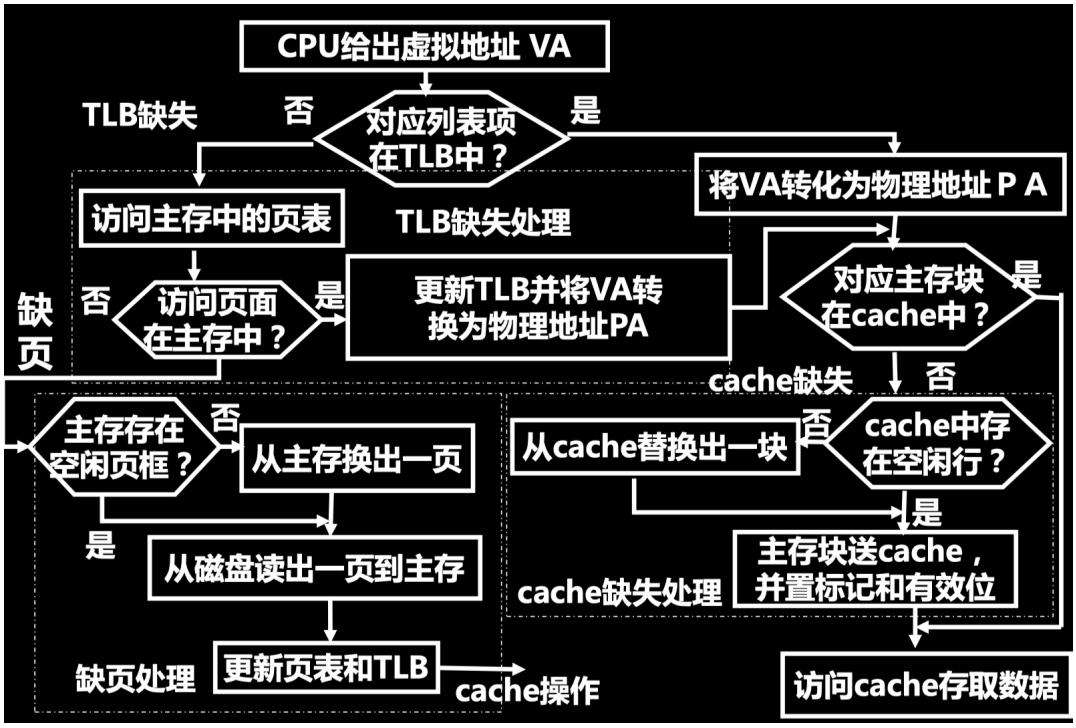
3.5.2 Cache 块替换算法

1. 先进先出算法（FIFO）
2. 最近最少使用算法
3. 随机算法

3.5.3 Cache 工作过程



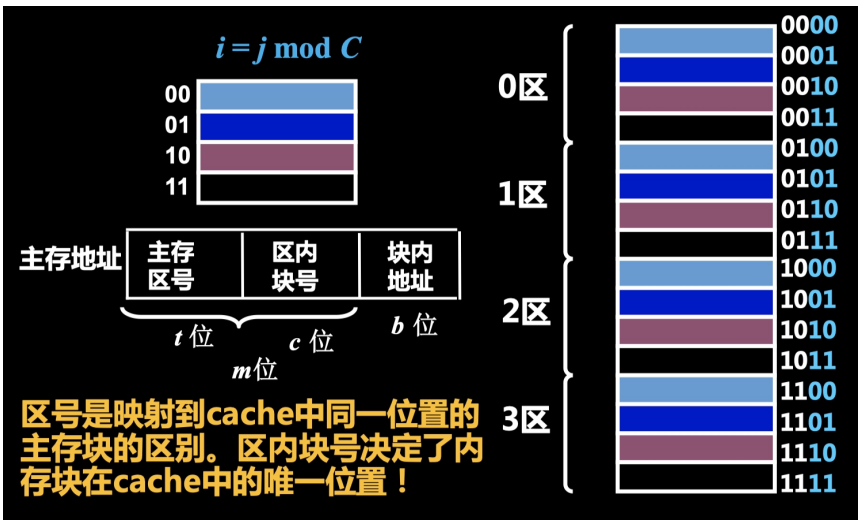
> CPU 工作过程



3.5.4 Cache 映射方式

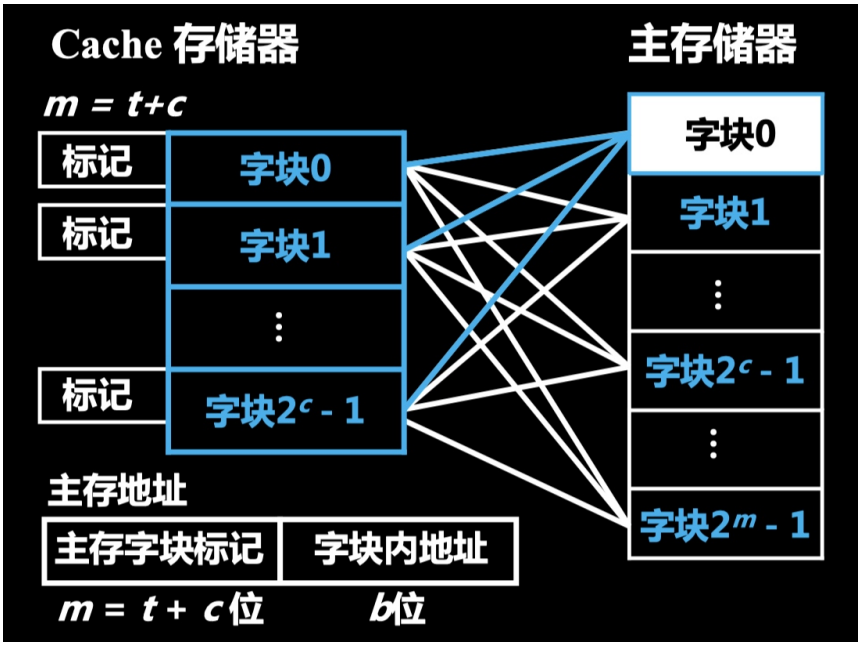
> 直接映射

最直接的映射方式，每4个对应一组进行分配。



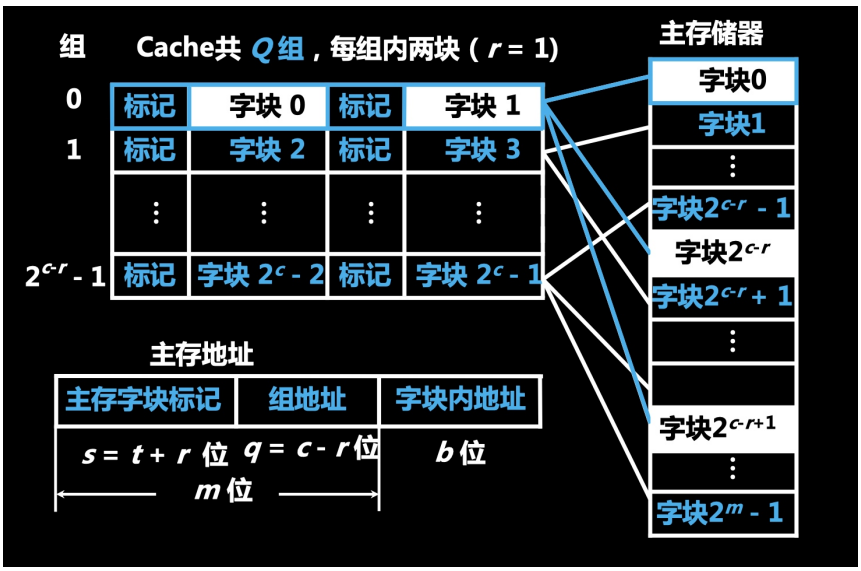
> 全相连映射

多对多映射，主存中的任一块可以映射到 Cache 中的任一块。



> 组相连映射

结合上述两种映射方式，划一个组专门用于存储 $\%4 = 0$ 的主存块。



> 三种方式比较

1. 直接映射：某一主存块只能映射到唯一缓存块。映射方式简单，但是不够灵活，容易导致块冲突。
2. 全相联映射：某一主存块可映射到任一缓存块，最灵活，不易冲突，但需要采用相联存储器。
3. 组相联映射：某一主存块可映射到某一组中的任一块，兼顾了灵活性和成本。

3.5.5 Cache 的一致性问题

> 写直达法（Write - through）

- 写操作时既写入Cache，也写入主存。
- 写操作时间变为访问主存所需时间。

> 写回法（Write - back）

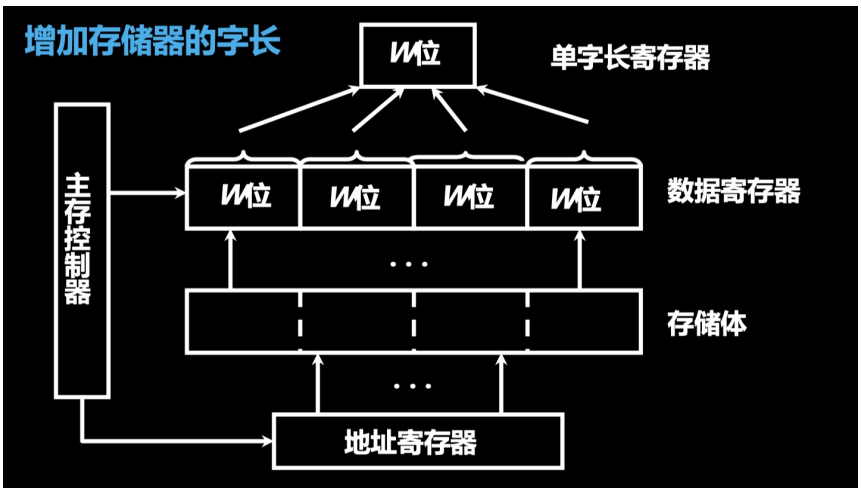
- 写操作时只写入 Cache，不写入主存，当 Cache 被替换出去时才写回主存
- 命中时读写操作时间就是访问 Cache 的时间

3.6 多模块存储器

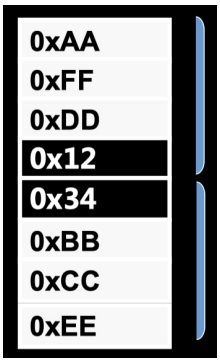
3.6.1 单体多字系统

> 结构概述

一个存储体，但对每次读出的字长进行扩充。一个存取周期取出 4 个字，然后逐次送至 CPU 执行，适用于指令或数据在内存中连续存放的情况。



> 优缺

1. 取指冲突
   - 例如出现了转移指令，则之后多取出的指令均浪费
2. 读操作数冲突
   - 

   - 需要的两个操作数分别为 0x12 和 0x34，因此读了8个字，但只有2个字有效，其余均为浪费。
3. 写数据冲突
   - 每次修改或写数据，都必须凑齐 n 个字才能写，或者 "读-改-写"

3.6.2 多体并行系统

> 存储体

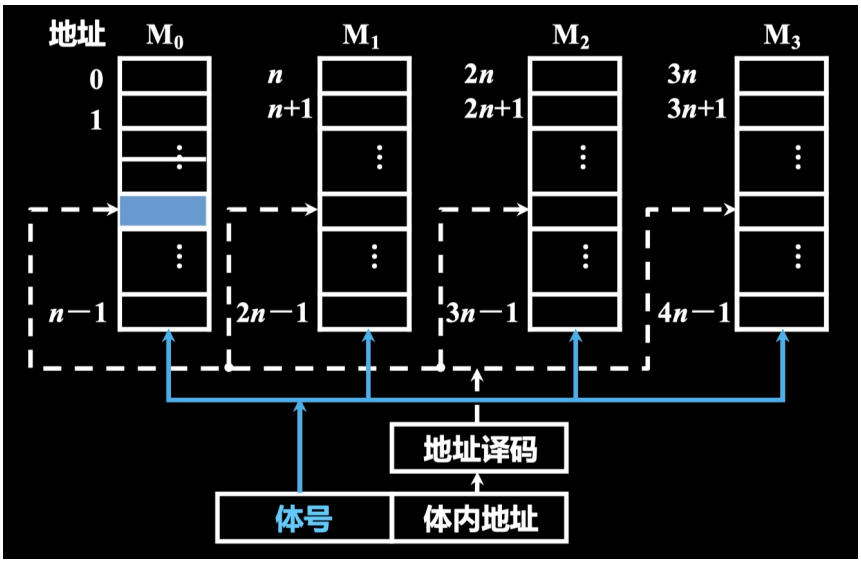
存储体为具有独立地址寄存器、数据寄存器、地址译码、驱动电路、读写电路等功能的独立存储电路，可独立工作。

> 多体并行

存储器由多个存储体构成，各存储体有相同的容量和存取速度。

> 高位交叉编址

即对于每个体按顺序向下编址。

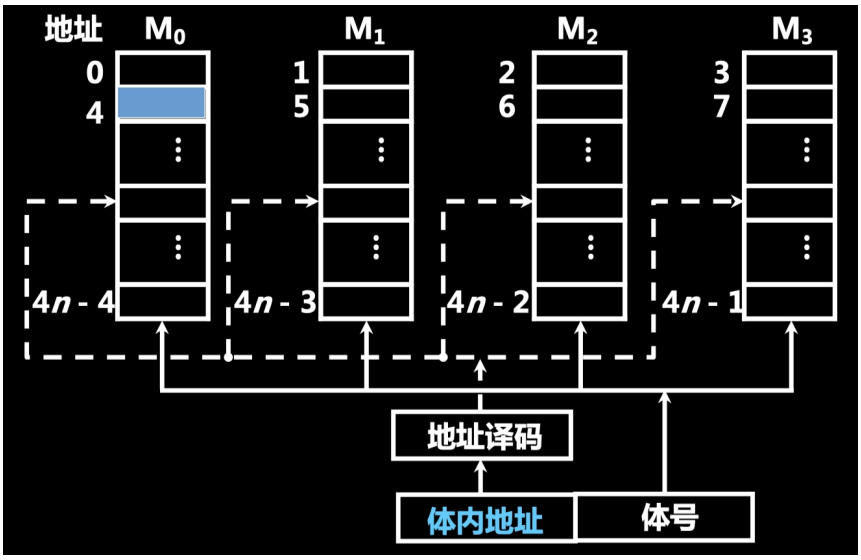


优缺点：

1. 频繁访问单个存储体时，吞吐量无法提升
2. 不同请求源可同时对各体发出请求，此时各体可同时工作
3. 便于存储器扩充

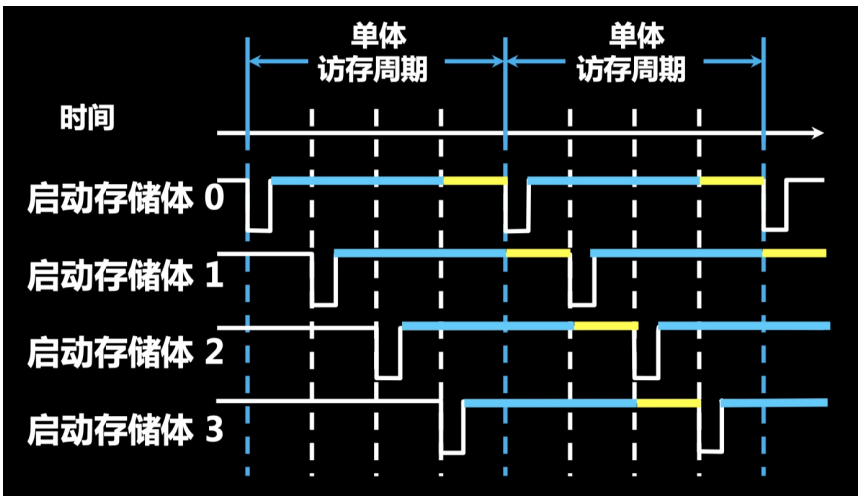
> 低位交叉编址

即轮流依次编址。



优点：

1. 在不改变存取周期和数据总线宽度的前提下，增加了存储器的带宽
2. 可利用其它存储体寻址的时间继续操控其它存储体
    - 连续运行时，每个时钟周期即可完成一个数据传输



3.7 数据校验码

3.7.1 码距

> 定义

一种编码体系中，任意两组合法编码之间不同的二进制位数的最小值，叫做这种编码的码距。

> 码距公式

码距决定了编码的纠错和检错能力。

L: 码距，D: 检错位数，C: 纠错位数

$$L-1=D+C$$

所有的校验码（如奇偶校验码、循环冗余校验码、海明码）都是通过增加码距来提高纠错和检错能力。

3.7.2 模 2 运算

> 模 2 加减运算规则

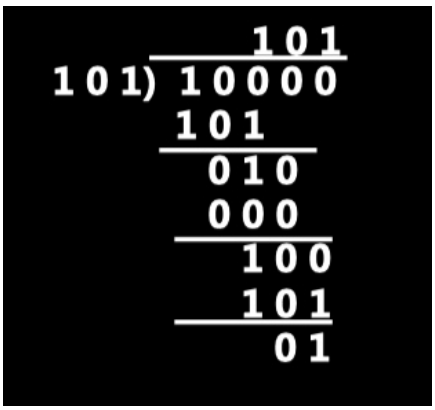
按位模2相加减，不考虑进位和借位。

> 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。



3.7.3 多项式和编码的对应关系

$$M(x) = C_{k-1}x^{k-1} + C_{k-2}x^{k-2} + \ldots + C_i x^i + \ldots + C_1 x + C_0$$

例如 $M(x) = x^3 + x + 1$，对应编码为 1011。

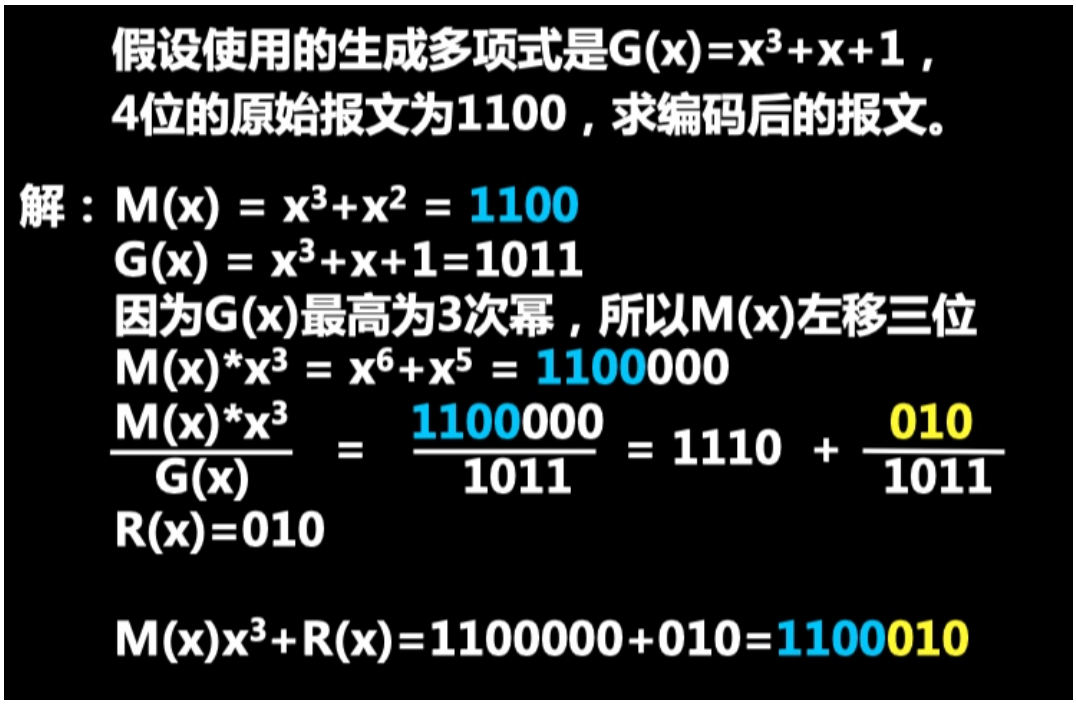
编码左移 r 位，对应的多项式为 $M(x)x^r$ 。

3.7.4 循环冗余校验码（CRC编码）

> 概念

1. 在 K 位信息码后再拼接 r 位的校验码，整个编码长度为 N 位，这种编码又叫 (N, K) 码。
2. 对于一个给定的码，可以证明存在一个最高次幂为 $N - K = r$ 的多项式 $G(x)$。根据 $G(x)$ 可以生成 r 位校验码，而 $G(x)$ 叫做这个 CRC 码的生成多项式。

> 举例



> 总结

1. 确定生成多项式 $G(x)$ 的最高次数为 r
2. 原始报文 $M(x)$ 左移 r 位
3. 用左移后的报文 $M(x)x^r$ 模 2 除以生成多项式，取余数 $R(x)$
4. 将余数 $R(x)$ 拼接在原始报文 $M(x)$ 右侧，即得 CRC 编码

第四章 输入输出系统

4.1 输入输出概述

4.1.1 输入输出系统的组成

输入输出系统由 I/O 软件和 I/O 硬件两部分组成。

> I/O 软件

（1）I/O指令

I/O指令属于CPU，由操作码、命令码、设备码三部分组成。

1. 操作码: I/O 指令标志
    - 操作码为真，则代表为I/O指令
2. 命令码: 指令功能，读、写、检测、控制命令
3. 设备码: 地址码，指明 I/O 指令要访问的设备地址

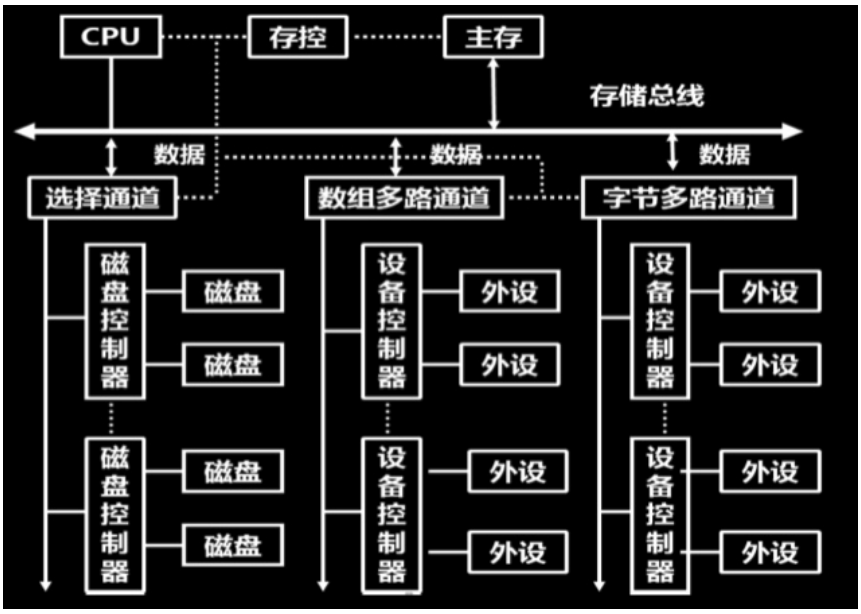
（2）通道指令

- CPU内部的专用处理器，只能进行输入输出操作。通道的级别是处理器的级别，因此有自己的指令集。
- 通道自身的指令，用于执行I/O操作，存放于主存，由通道执行，完成输入输出功能。

- 在具有通道的计算机中，I/O指令不负责数据输入输出，而主要完成启停设备、通道控制，实际数据传输由通道完成。

> I/O 硬件

通道 -> 设备控制器 -> 设备，三级硬件结构。

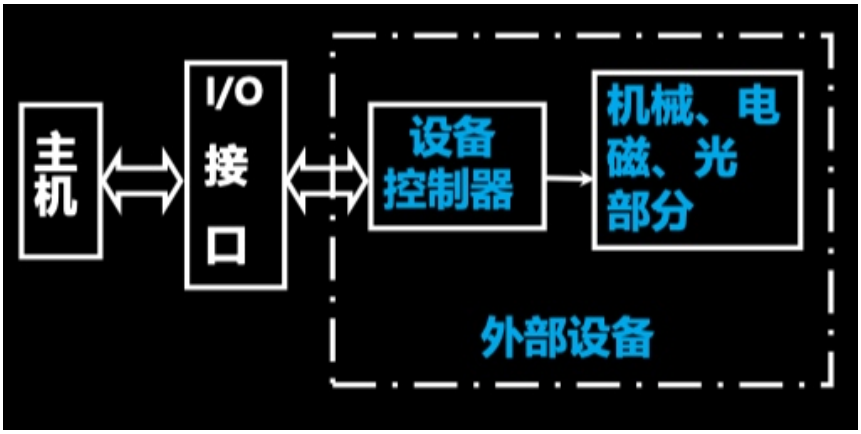


每一个设备控制器只能够连接同种类型的若干设备。

4.1.2 I/O 接口

> I/O 接口在系统中位置

接口可以看做两个系统或部件之间的交接部分，它既是两种硬件设备之间的连接电路，也可以看作两个软件之间的共同逻辑边界。



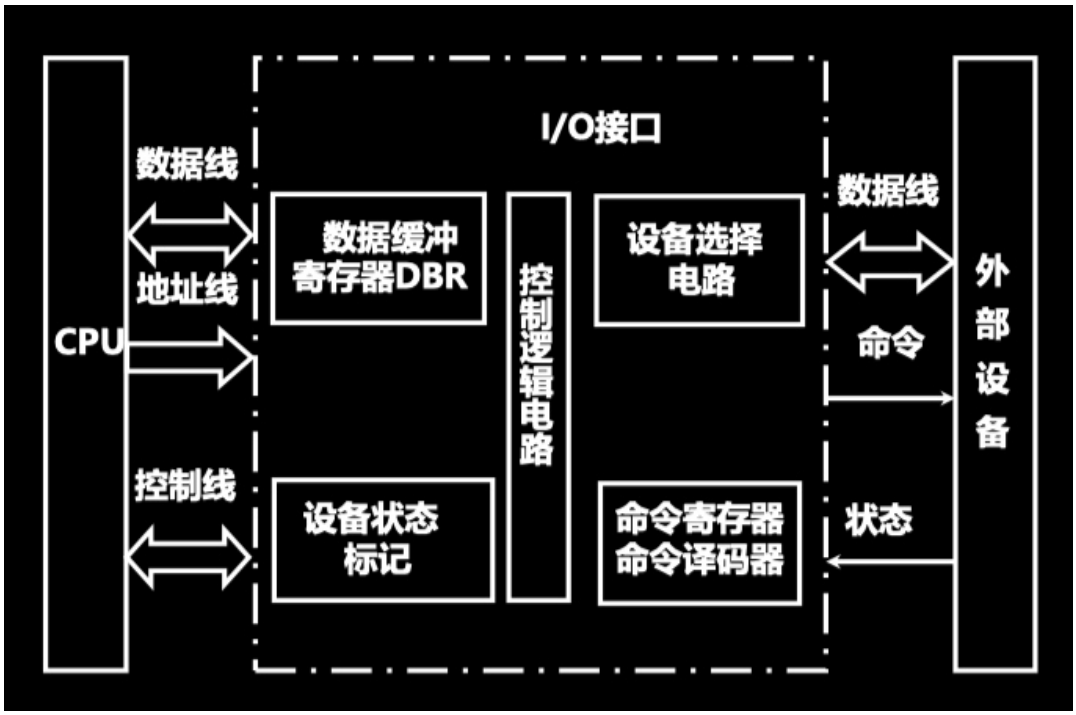
> I/O接口的功能

1. 设备选择
    - 所有接口都会收到主机的请求，接口会判断主机找的是不是自己

2. 速度匹配
    - 匹配主存与外设的传输速度
3. 串并转换
    - 串并数据形式转换
4. 电平转换
    - 设备电平定义与主机不同，需要转换
5. 传送控制命令
6. 反映设备状态

> I/O接口的基本组成

- 数据缓冲寄存器: CPU 会将数据送到该寄存器中
- 设备选择电路: 分析 CPU 发送来的地址，判断 CPU 找的是不是当前接口
- 命令寄存器: 接口通过命令译码器对命令译码，之后由命令线控制外设进行工作
- 设备状态标记: 保存外设状态，供 CPU 读取



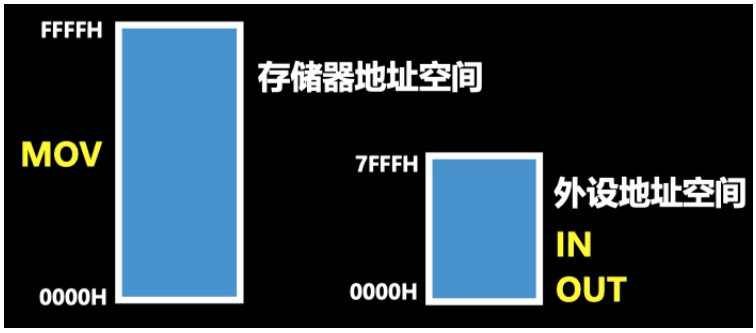
4.1.3 I/O 设备编址方式

> 统一编址

- I/O 占用存储器地址空间，无须专门的 I/O 指令。
- 采用不同地址来区分访问对象，减少了存储器最大容量。

> 独立编址

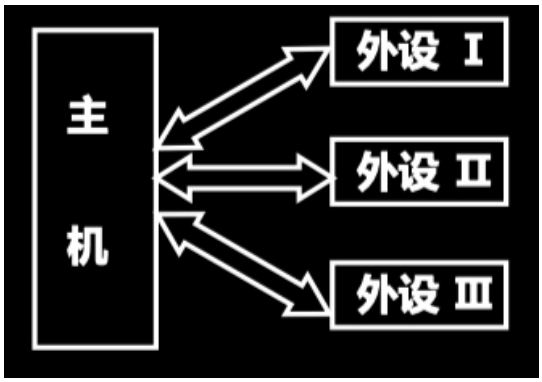
- I/O 地址与存储器地址分开，采用专门指令来访问 I/O。
- 采用不同指令形式来区分访问对象，不占用主存容量。



4.1.4 I/O 设备与主机的连接方式

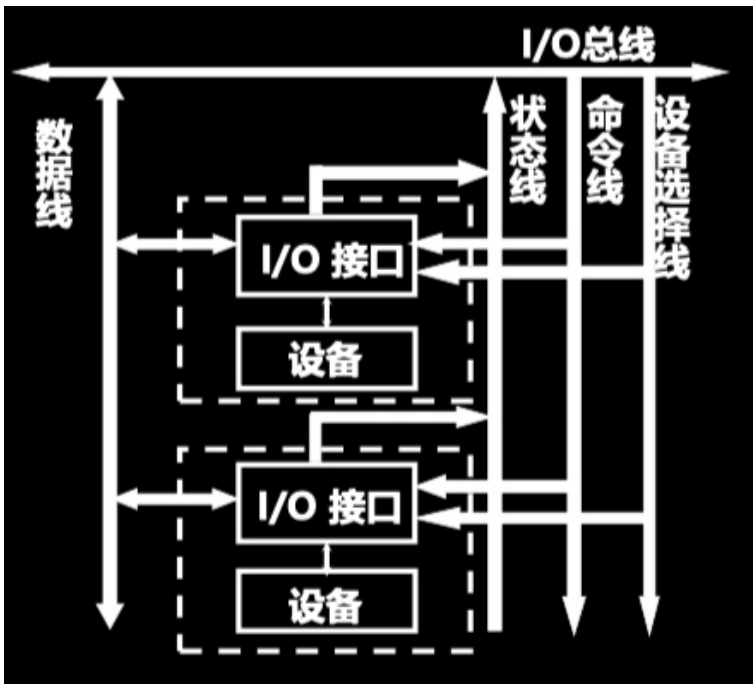
> 辐射式

每台设备都配有一套控制线路和一组信号线。



> 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



4.2 程序中断方式

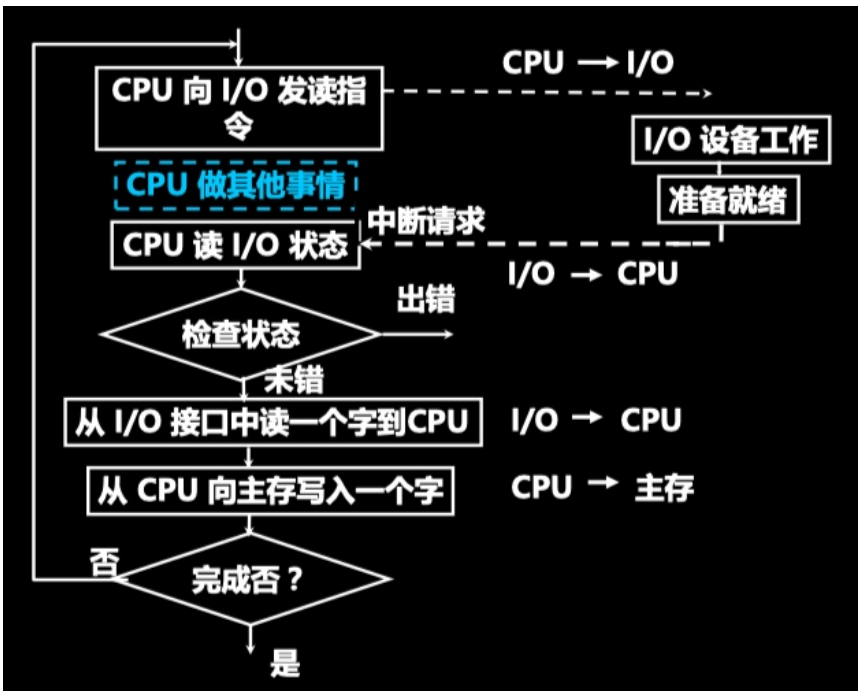
4.2.1 中断过程

> 中断概念

计算机在执行程序的过程中，当出现异常情况或特殊请求时，计算机停止现行程序的运行，转向对这些异常情况或特殊请求的处理，处理结束后再返回到现行程序的间断处，继续执行原程序，这就是中断。

> 中断方式

中断程序消除了CPU原地踏步的情况，但是，CPU在响应中断请求后，必须暂停现程序，转而去执行中断服务子程序，消耗了CPU资源，并且中断处理过程是有开销的。



4.2.2 中断源

> 引起中断的各种因素

1. 人为设置的中断（定时中断）
2. 程序性事故：溢出、操作码不能识别、除法非法
3. 硬件故障
4. I/O设备
5. 外部事件（按键中断）

> 可屏蔽中断

每一个中断源都对应一个中断屏蔽位，一旦处理器把中断屏蔽位设为“真”，即屏蔽了这个中断源。这个中断源再发出中断请求，也不会被响应。

> 不可屏蔽中断

只要外设通过相应的引脚发出了中断请求，CPU一定会响应这个中断。

> 各中断源如何向 CPU 提出请求?

每个中断源都对应一个**中断请求标记触发器**，中断源提出了中断请求，就会把这个触发器置为“真”，这就是中断源提出中断请求的唯一标志。

> 各中断源同时提出请求，怎么办?

采用**中断优先级比较电路**或软件方法，来比较中断优先级，选出其中优先级最高的那个中断去响应。

> CPU在什么条件下响应中断?

【条件1】**程序状态字**，一般写作PSW，其中有两类信息，一类信息叫做控制位，另一类信息叫做状态位。

中断允许位，是一个中断的总开关，如果这个位的值是 “1”，就代表 CPU 现在可以接受外部的中断；如果是 “0”，就代表 CPU 根本不接受中断，所以这个位也称作中断总开关。

【条件2】每一个可屏蔽中断，都有一个**中断屏蔽位**，中断屏蔽位如果是 ”1“，就代表 CPU 要把这个中断源屏蔽掉。只有当这个位是 ”0“，表示没有屏蔽这个中断源时，CPU才可以去响应中断。

> CPU在什么时间下响应中断?

CPU 只能在执行完当前指令后，才能去响应中断。CPU 不会在指令执行过程中响应中断。

> CPU以什么方式响应中断?

在中断响应中计算机会去做三件事，这三件事没有任何一条指令控制，都是自动去做，因此叫做**中断隐指令**。

1. 硬件关中断
    - 进入中断程序之后会有一个非常关键的动作，叫做保护现场，我们希望保护现场的时候中断的总开关是关着的，所以要有一个硬件自动关中断的操作。
2. 保护断点
    - 中断服务程序的返回地址，当发生中断跳转时，即将要执行到的那条指令的地址，断点必须保存。
3. 向中断服务程序的入口跳转
    - 硬件方式、软件方式

> 如何保护现场?

把中断服务程序运行过程中要用到的那些寄存器值送到堆栈里面保护起来，这就是保护现场。

> 如何恢复现场，如何返回?

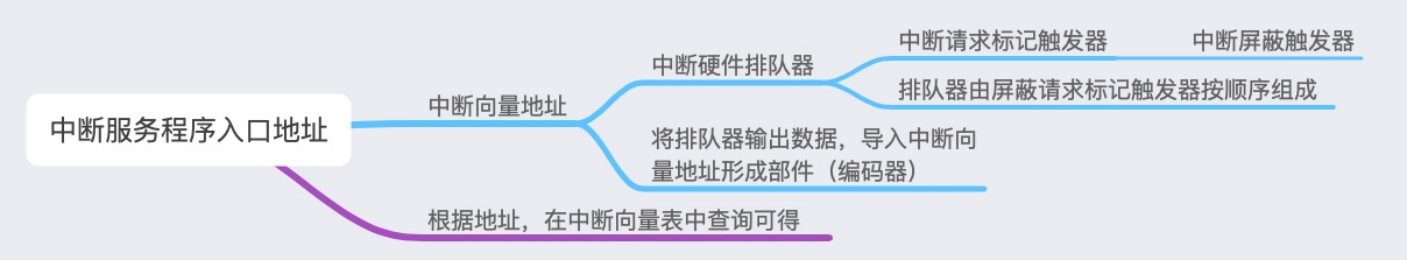
恢复现场和保护现场总是成对出现。恢复现场是指在中断服务程序返回之前，按照刚才入栈的逆序把堆栈顶端的若干个信息弹回到相应的寄存器中。

把断点的值送给程序计数器，程序计数器就会指向刚才在主程序中马上要执行而又没有执行到的那条指令的位置，接下来就会取这条指令并执行，完成中断的返回。

> 处理中断的过程中又出现新的中断怎么办?

中断的嵌套。

4.2.3 中断接口电路



> 中断请求标记触发器

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。

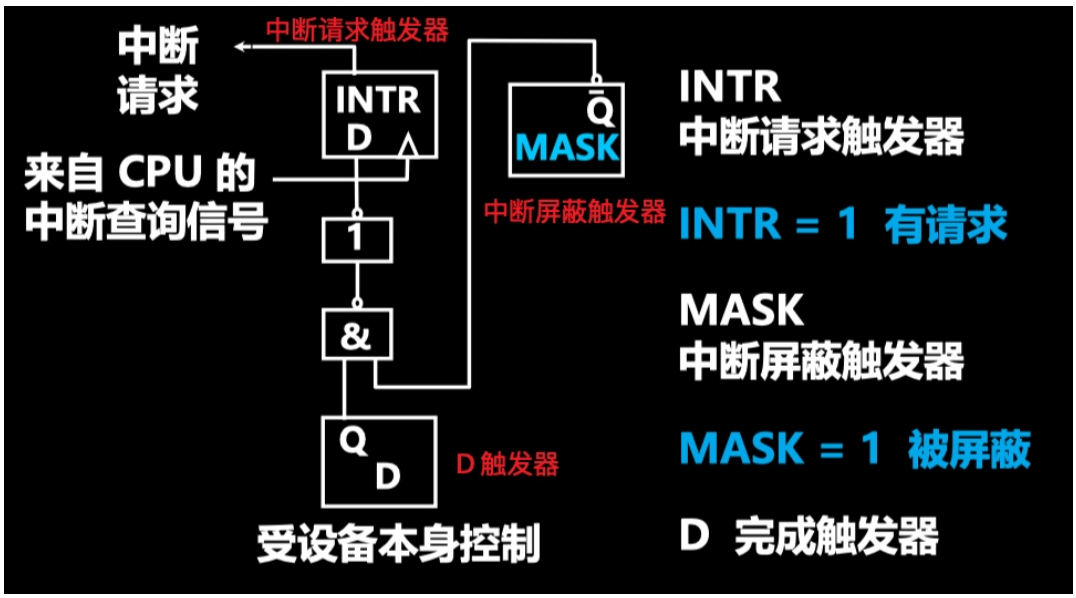
两种存在方式：

1. 分散在各个中断源的接口电路中
2. 集中在 CPU 的中断系统内

> 中断屏蔽触发器

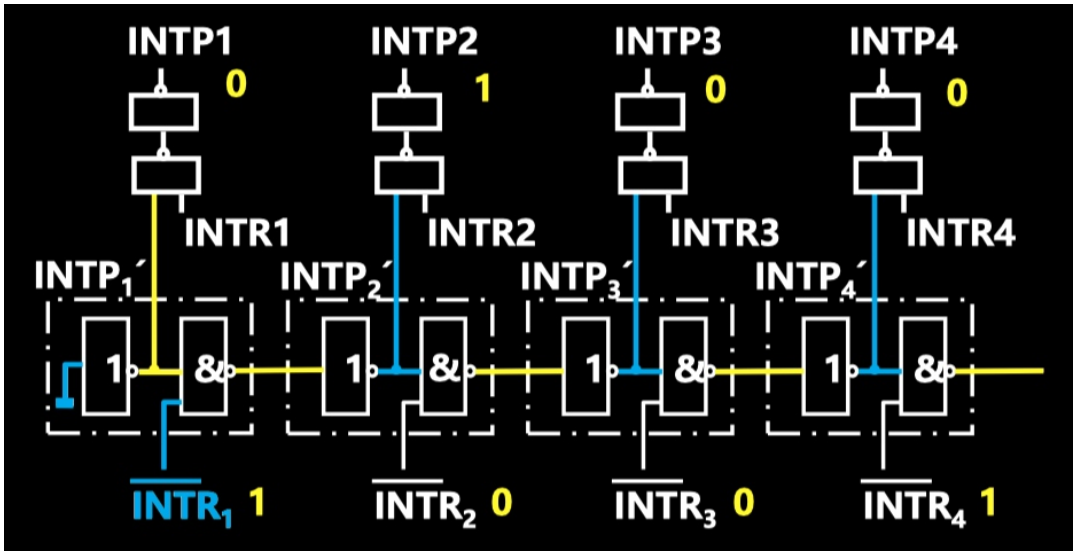
一个请求源对应一个中断屏蔽触发器，多个中断屏蔽触发器组成中断屏蔽寄存器。

下图为中断屏蔽触发器与D触发器一起构成了中断请求触发器中的数值，如果中断屏蔽触发器为 1，则请求触发器中一定为 0。



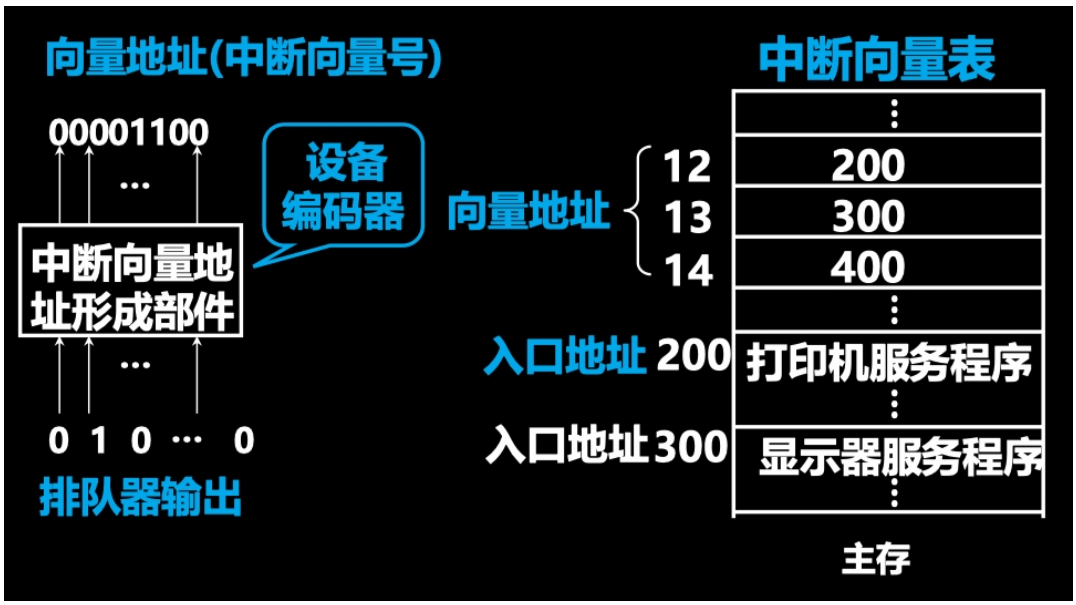
> 中断硬件排队器

硬件实现，从左到右依次执行，$INTR_i = 1$ 表示有请求，一旦左边一个设备提出请求，则之后的设备均被屏蔽，优先级依次递减。

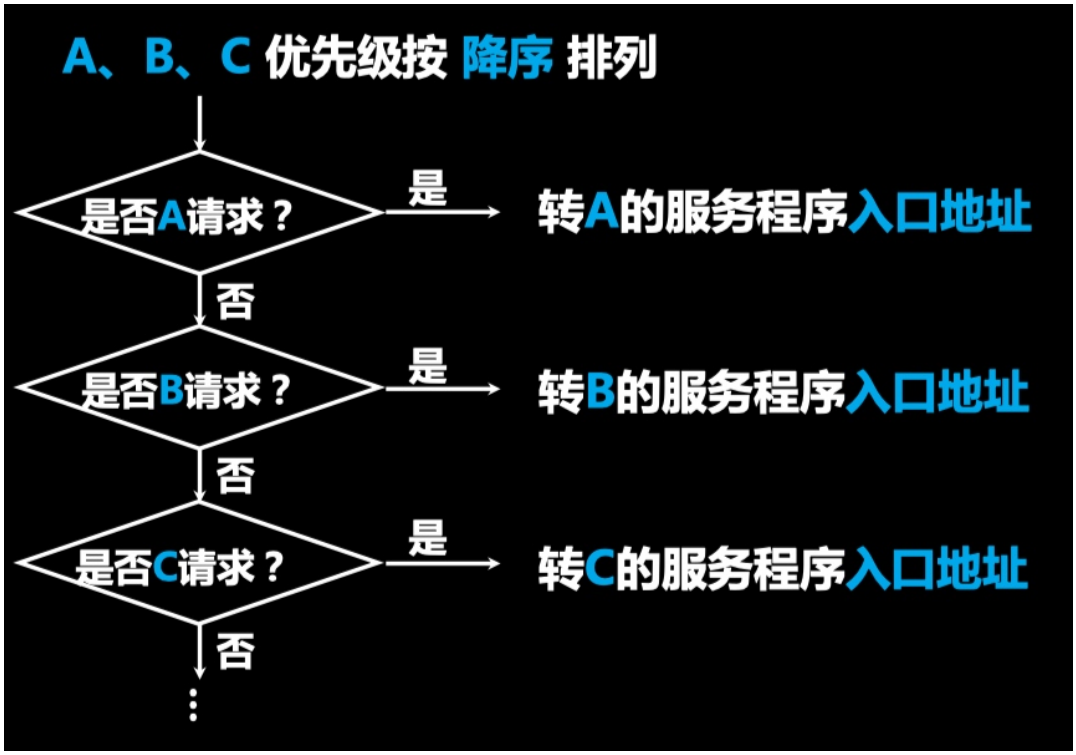


> 中断向量地址形成部件

将排队器输出的数据导入到编码器中，得到向量地址，再将向量地址加入中断向量表中即可找到入口地址。

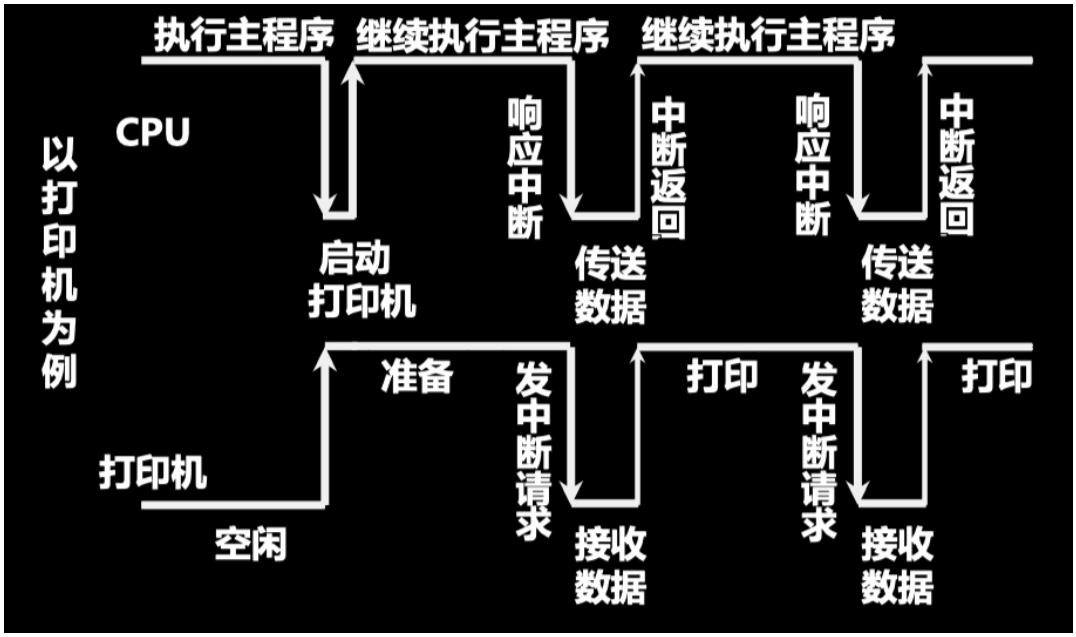


> 中断服务程序软件查询



4.2.4 中断工作方式

以打印机为例。



4.2.5 中断处理过程

> CPU 响应中断的条件和时间

条件：

- CPU允许中断，即中断允许触发器 EINT=1（开关中断指令即改变EINT）
- 相关中断的掩码（MASK）为0

时间：

- 每条指令执行结束（CPU 发出中断查询信号，将 INTR 置 1）

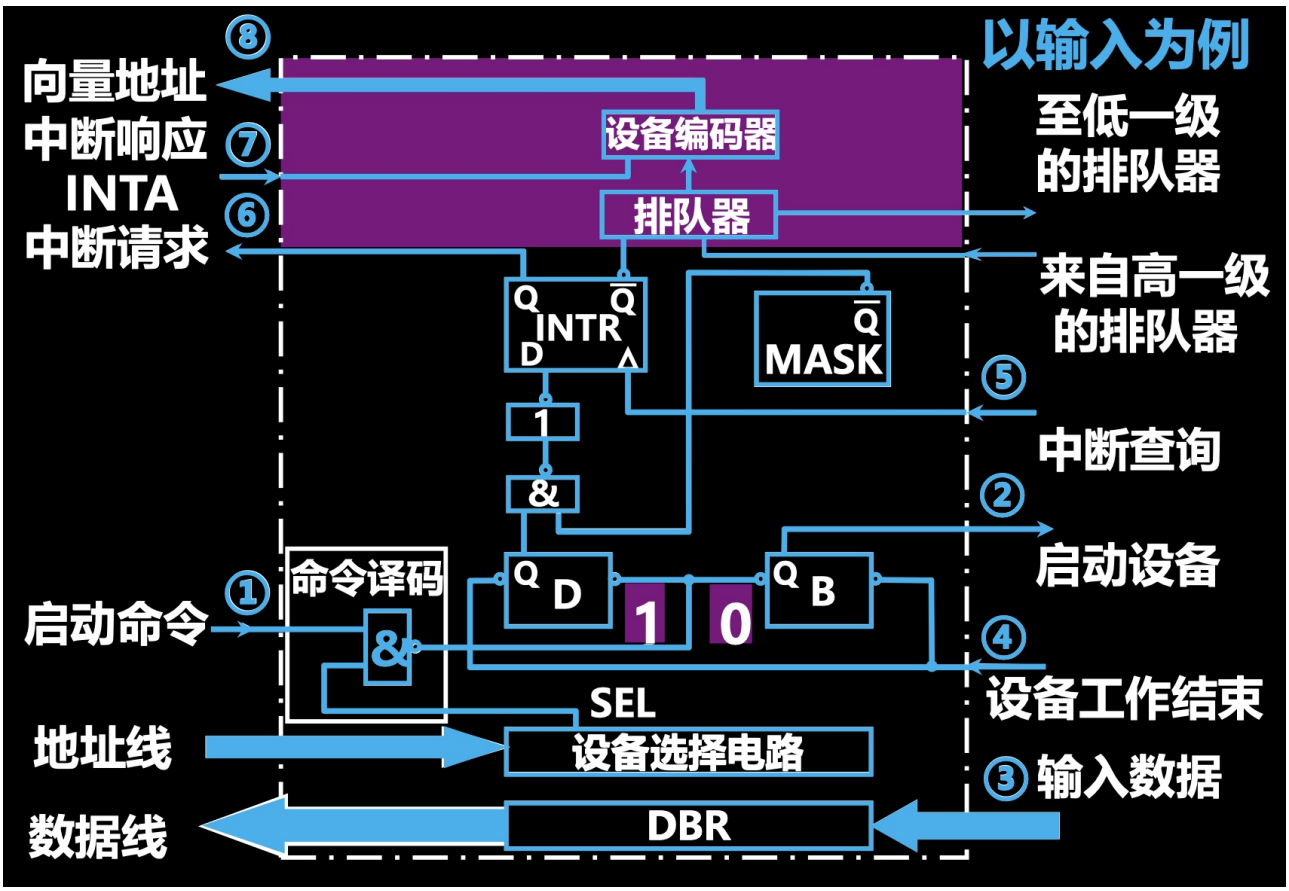
> 中断响应的实现

通过中断隐指令完成。

- 保存程序断点。
  - 断点存于特定地址（0号地址）内，断点进栈。
- 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）
- 硬件关中断。

> I/O 中断处理过程

INTR：中断请求标记触发器
MASK：中断屏蔽位
D触发器：完成触发器（Done）
B触发器：设备忙触发器（Busy）



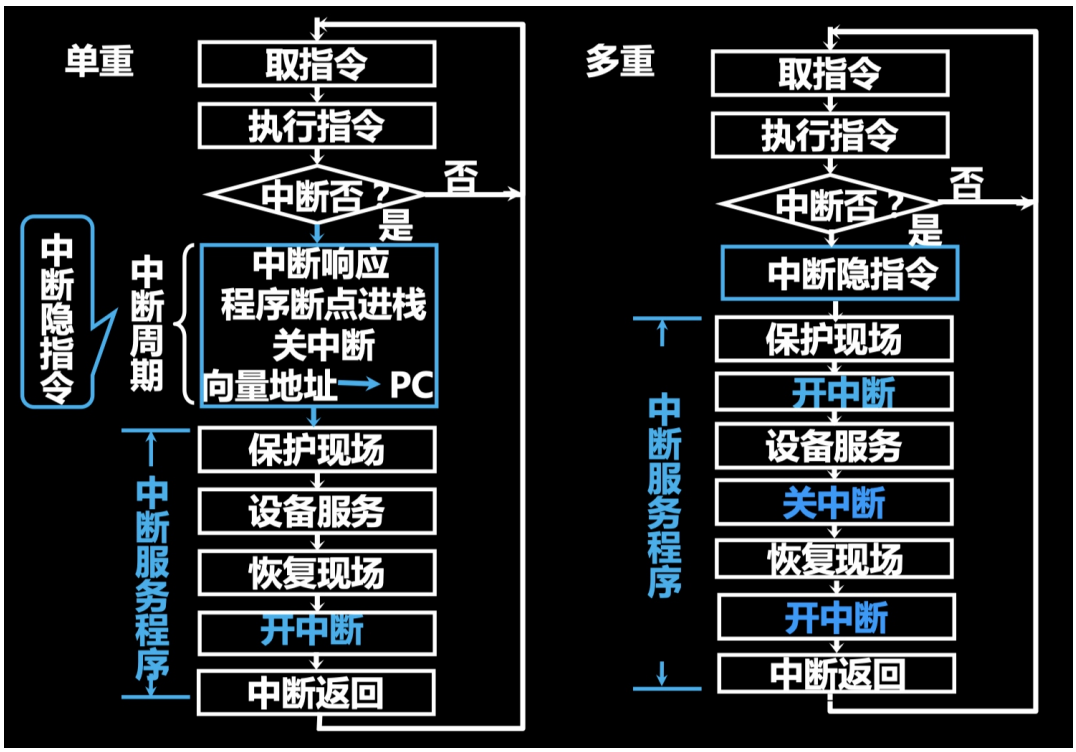
1. 地址线发出设备地址。所有设备将地址与自身地址进行比较，查看 CPU 找的是不是自己。如果 CPU 找到是自己，就会令 SEL = 1。
2. 设备发出启动命令 1，命令译码输出 0。D触发器被置0，B触发器被置1。B触发器被置1，表示设备正忙，且工作没有完成。同时B触发器会启动设备，此时设备准备数据与CPU继续执行别的程序同时在发生。
3. 设备准备好数据，就将数据传给 DBR。
4. 设备工作结束，则将 Done 触发器置 1，同时把 Busy 置为 0。Done 触发器向上连接 INTR，此时 MASK = 0，设备不被屏蔽，触发器才能继续工作。
5. CPU执行完一条指令，发出中断查询信号，作为时钟脉冲将 1 打入 INTR 中。
6. INTR 被置为 1，发出中断请求。同时多个设备同时进入排队器进行排队。
7. CPU发出中断响应信号，即编码器的输出允许。
8. 设备编码器发出当前中断设备的中断号。CPU根据中断号查询中断向量表得到中断服务程序的入口地址。

> 中断服务程序流程

CPU查找向量表，找到中断服务程序入口，然后执行中断服务程序。

1. 保护现场
   - 保存寄存器值：进栈指令。
   - 中断服务程序用到的寄存器，都应入栈。
2. 中断服务
   - 视情形开、关中断

3. 恢复现场
    - 恢复寄存器的值，出栈指令
4. 开中断
5. 中断返回
    - 中断返回指令。将返回地址赋值给PC。

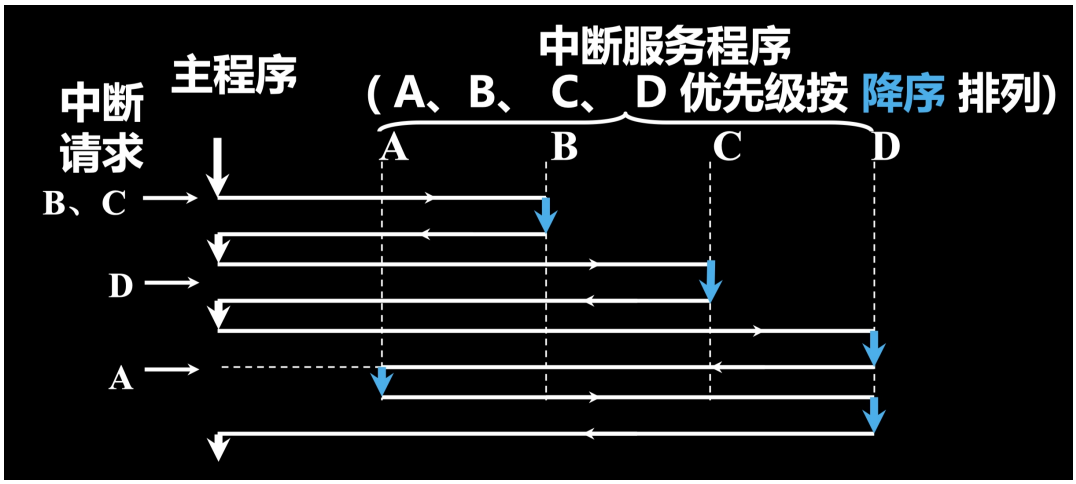


单重中断与多重中断的区别。

4.2.6 中断屏蔽技术

> 多重中断的条件

1. 执行中断服务程序时，在保护现场之后，设置开中断指令。
2. 优先级别高的中断源有权中断优先级低的中断源。



> 中断屏蔽字

每一个中断源对其它中断源都有对应的屏蔽字。为1表示屏蔽，为0表示不屏蔽。

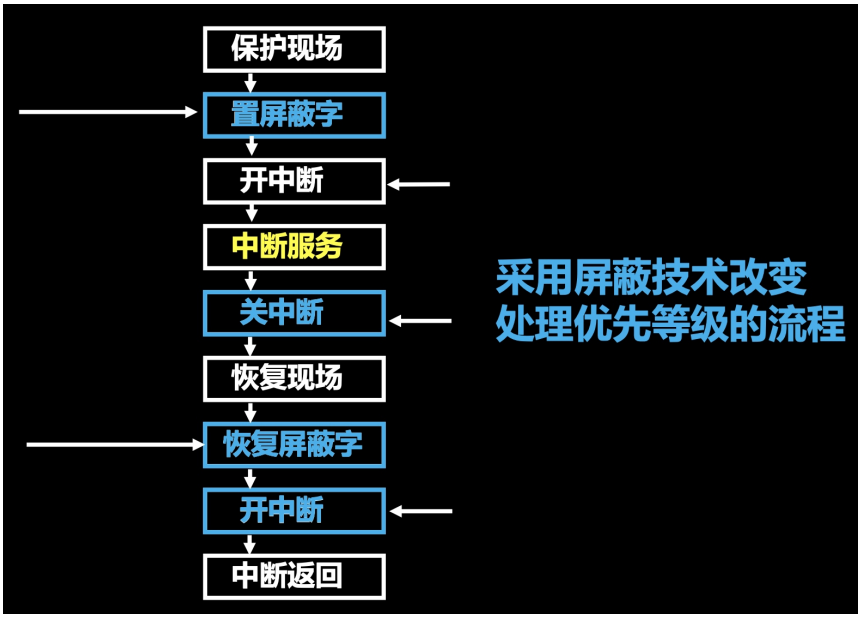
中断源	屏蔽字
1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
3	0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
4	0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
5	0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
6	0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
⋮	⋮
15	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
16	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

16个中断源的优先级从上到下依次为 1、2、3...16，按降序排列。

> 通过中断屏蔽技术改变处理优先级

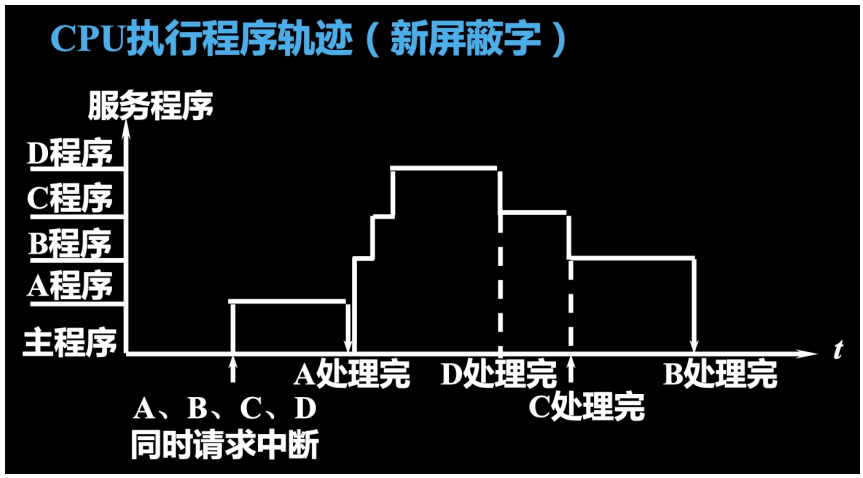
响应优先级: CPU响应各中断源请求的优先次序。
处理优先级: CPU实际对各中断源请求处理的优先次序。

新屏蔽字的设定（处理优先级的修改）：在中断服务程序执行过程的保护现场之后进行。



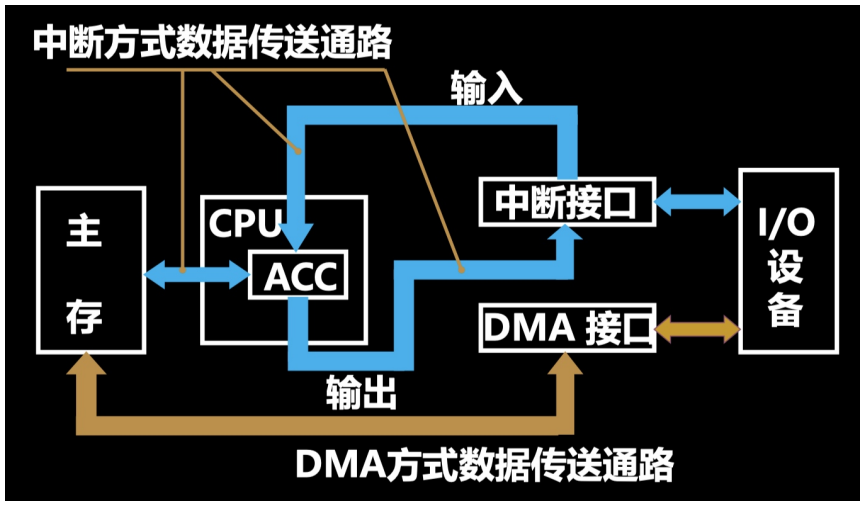
中断源	原屏蔽字	新屏蔽字
A	1 1 1 1	1 1 1 1
B	0 1 1 1	0 1 0 0
C	0 0 1 1	0 1 1 0
D	0 0 0 1	0 1 1 1

上图中，响应优先级为 A → B → C → D 降序排列。处理优先级为 A → D → C → B 降序排列。



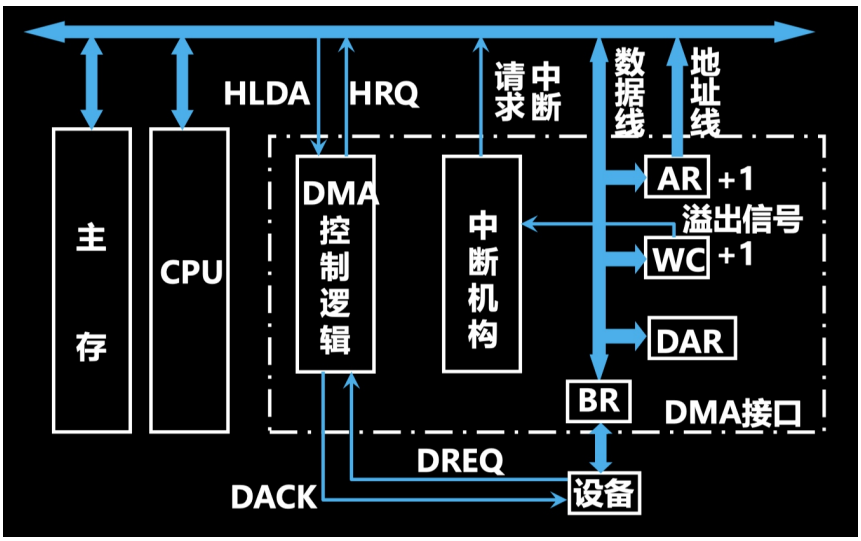
4.3 DMA 工作方式

4.3.1 DMA 与中断的比较



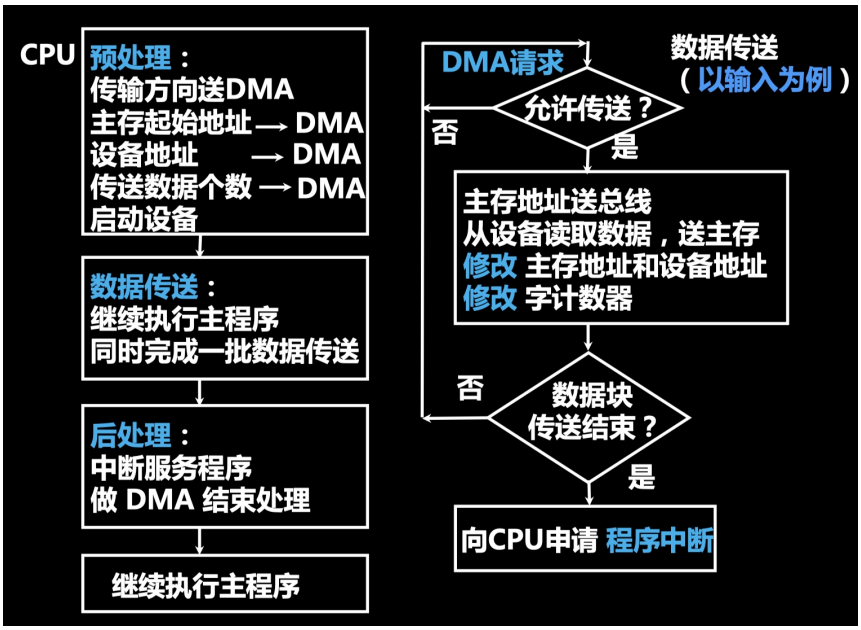
4.3.2 DMA 接口

HLDA: CPU 发给接口的总线应答信号
HRQ: DMA 接口发给 CPU 的总线使用请求信号
DACK: DMA 控制器发给设备的应答信号
DREQ: 设备发给 DMA 控制器的请求信号
AR: 主存地址寄存器
WC（Word Counter）：计数器
DAR: 设备地址寄存器
BR: 数据缓冲寄存器



4.3.3 DMA 的工作过程

1. 预处理：CPU 参与
2. 数据传送：CPU 不参与，由 DMA 控制器完成
3. 后处理：中断形式，CPU 参与



> 后处理

由中断服务程序完成，主要目的如下。

1. 检验输入主存的数是否正确。
2. 决定是否继续使用 DMA，重新初始化 DMA，或者停止设备。
3. 测试传送过程是否正确，错则转诊断程序。

4.3.4 DMA 与中断方式的比较

DMA方式在存取周期结束时即可响应，例如如下的存取周期：

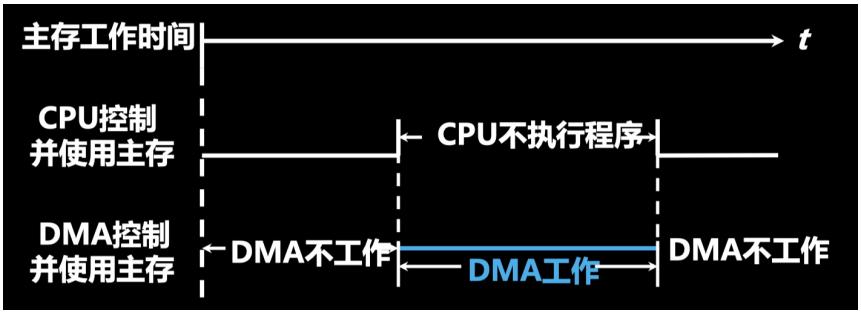
1. CPU 刚取完指令
2. CPU 刚执行完间址阶段
3. CPU 刚做完一次乘法操作

	中断方式	DMA方式
（1）数据传送	程序	硬件
（2）响应时间	指令执行结束	存取周期结束
（3）异常处理情况	能	不能
（4）中断请求的作用	传送数据	后处理
（5）优先级	低	高

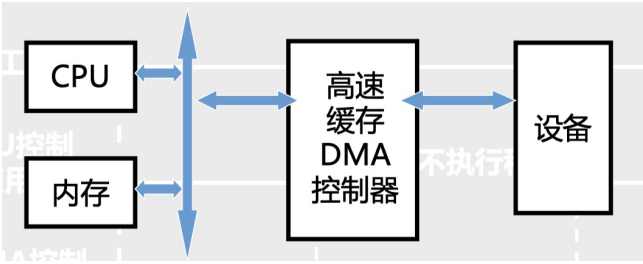
4.3.5 DMA 与主存交换数据的三种方式

> 停止 CPU 访问主存

控制简单，CPU处于不工作状态或保持状态。



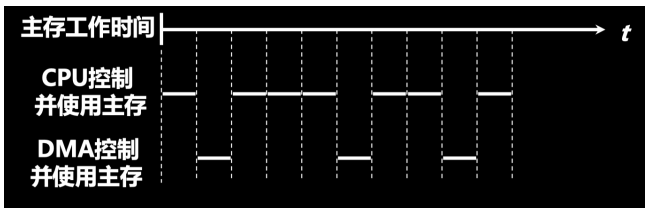
未充分发挥 CPU 对主存的利用率，可以在 DMA 接口中增加高速缓存，减少 DMA 传送数据时占用总线的时间。



> 周期挪用（周期窃取）

DMA 访问主存有三种可能：

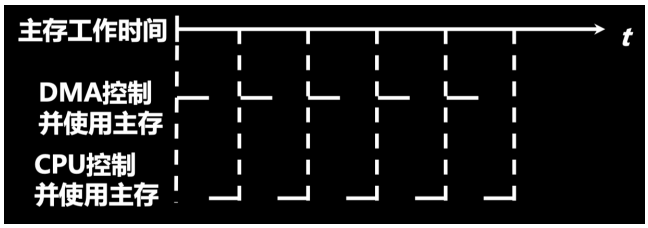
1. CPU 此时不访存：总线立刻交给 DMA
2. CPU 此时正访存：CPU 访存结束后总线立刻交给 DMA
3. CPU 与 DMA 同时请求访存：总线交 DMA



第一种方式：只要 DMA 在工作，CPU 就不能控制主存，即使 DMA 并没有控制主存。
第二种方式：只有 DMA 控制主存时，CPU才不能控制主存，即 DMA 在准备数据的过程中，CPU 仍可以控制主存。

> DMA 与 CPU交替访问

CPU工作周期分为两部分，C1: 专供 DMA 访存，C2: 专供 CPU 访存。此种方法不需要申请建立和归还总线的使用权。

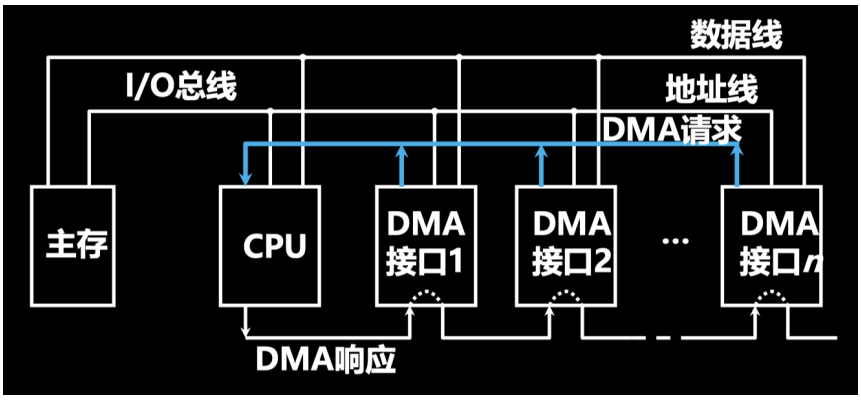


4.3.6 DMA 接口功能

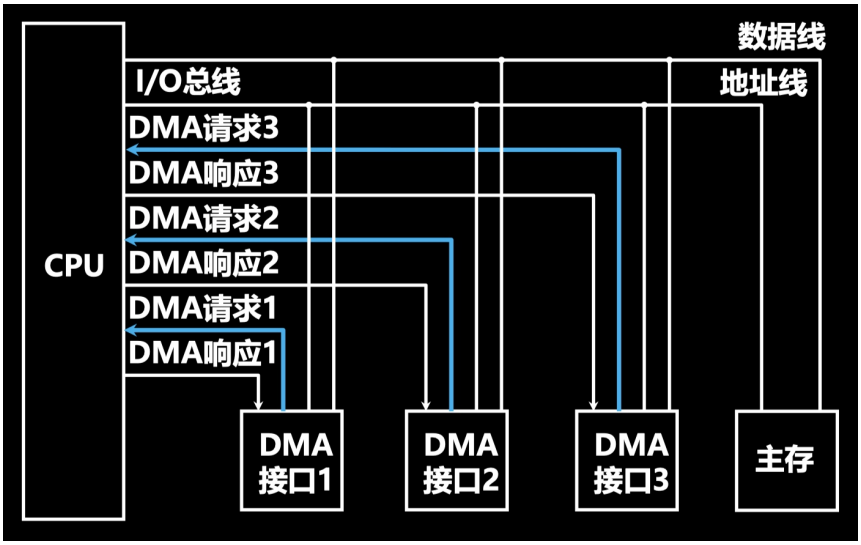
1. 向 CPU 申请 DMA 传送
2. 处理总线控制权的转交
3. 管理系统总线、控制数据传送
4. 确定数据传送的首地址和长度
5. 修正传送过程中的数据地址和长度
6. DMA 传送结束时，给出操作完成信号

4.3.7 DMA 接口与系统的连接方式

> 具有公共请求线的 DMA 请求



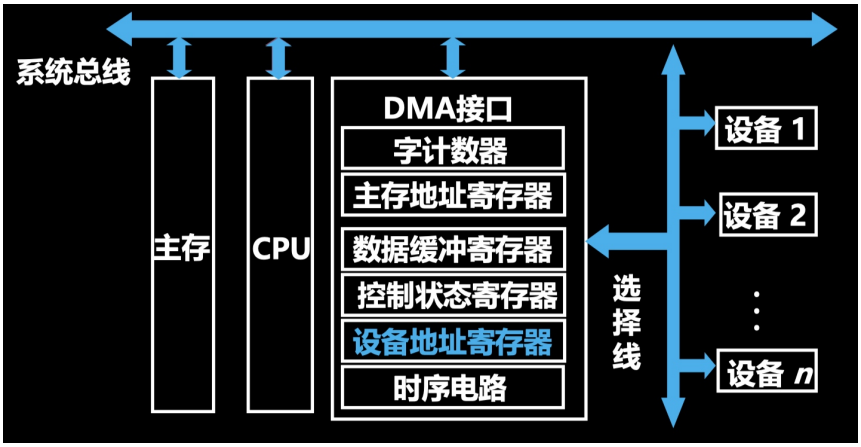
> 独立的 DMA 请求



4.3.8 DMA 接口类型

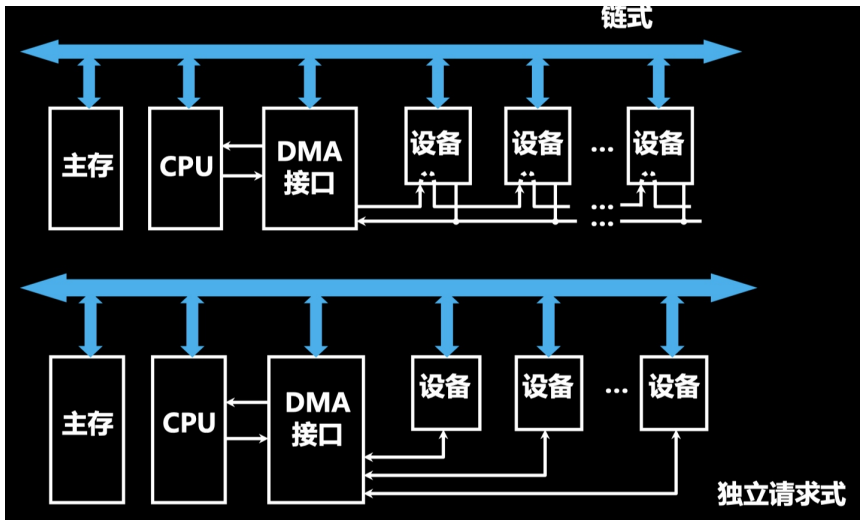
> 选择型

在物理上连接多个设备，在逻辑上只允许连接一个设备。



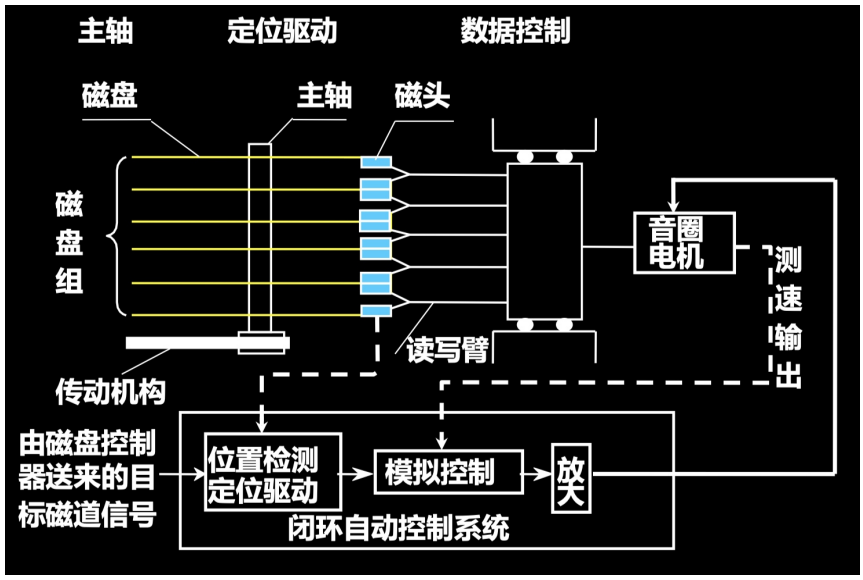
> 多路型

物理上连接多个设备，逻辑上允许连接多个设备同时工作。



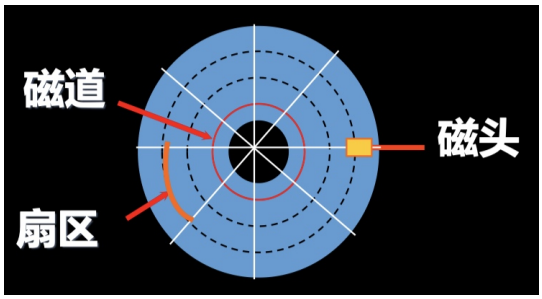
4.3.9 磁盘

> 磁盘驱动器



> 磁盘盘片基本结构

磁盘机、记录面（surface）、磁道（track）、柱面（cylinder）、扇区（sector）



> 磁盘地址

磁盘机号	记录面号	块号/扇区	块内地址

> 磁盘容量计算（格式化容量）

1. 扇区容量: S
2. 磁道容量: T = 每道扇区数 n * S
3. 单面容量: P = 每面磁道数 t * T
4. 磁盘容量: V = 总计盘面数 m * P

第五章 计算方法

5.1 机器数表示方法

5.1.1 数的原码表示

> 概念

符号位为 0 表示正数，为 1 表示负数，数值部分为真值的绝对值，又称为带符号的绝对值。

> 书写

约定书写时，正数的符号位与数值位用逗号分隔，小数的符号位与数值位用小数点分隔。

> 原码整数定义

$$[x]_{原} = \begin{cases} 0, x & 2^n > x \geq 0 \\ 2^n - x & 0 \geq x > -2^n \end{cases}$$

> 原码小数定义

$$[x]_{原} = \begin{cases} x & 1 > x \geq 0 \\ 1 - x & 0 \geq x > -1 \end{cases}$$

> 举例：求 x=0 的原码

若 x 为小数，则

$$[+0.0000]_{原} = 0.0000$$
$$[-0.0000]_{原} = 1.0000 - 0.0000 = 1.0000$$

若 x 为整数，则

$$[+0]_{原} = 0,0000$$
$$[-0]_{原} = 1,0000$$

> 特点

优点：表示方法简单，直观

缺点：加减运算复杂

5.1.2 数的补码表示

> 引入

减去一个数，可以用加上这个数的补数来代替，从而可以把减法转变为加法。

- 一个负数可用其正的补数来代替，这个正补数可以用此负数加模来求得。
- 一个正数和负数互为补数时，他们的绝对值之和即为模数。
- 正数的补数即为该正数本身。
- （负数的补码）符号位不变，数值位从左到右依次取反，最右边的 1 及其后的 0 不变。

> 补码整数定义

$$[x]_{补} = \begin{cases} 0,x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 > x \geq -2^n (mod\ 2^{n+1}) \end{cases}$$

$$[-1011000]_{补} = 2^{7+1} + (-1011000) = 1,0101000$$
$$[+1010]_{补} = 0,1010$$

> 补码小数定义

$$[x]_{补} = \begin{cases} x & 1 > x \geq 0 \\ 2 + x & 0 > x \geq -1 (mod\ 2) \end{cases}$$

$$[+0.1110]_{补} = 0.1110$$
$$[-0.1100000]_{补} = 2 + (-0.1100000) = 1.0100000$$

> 举例

$$
\begin{aligned}
6-3 &= [[6-3]]_{补} \\
&= [[6]+[-3]]_{补} \\
&= [0,110+1,101]_{补} \\
&= [10,011]_{补} \\
&= [0,011]_{补} \\
&= +3
\end{aligned}
$$

5.1.3 数的移码表示

> 引入

补码表示很难直接判断其真值大小，因此对所有补码都加上一个 2^n，得到了移码，可以直观地比较数值大小。

因此用移码表示浮点数的阶码可以方便地判断浮点数的阶码大小。

> 定义

$$[x]_{移} = 2^n + x(2^n > x \geq -2^n)$$

移码与补码的数值部分相同，符号位相反。

5.1.4 各机器数表示总结

二进制代码	无符号数对应的真值	原码对应的真值	补码对应的真值	反码对应的真值
00000000	0	+0	±0	+0
00000001	1	+1	+1	+1
00000010	2	+2	+2	+2
⋮	⋮	⋮	⋮	⋮
01111111	127	+127	+127	+127
10000000	128	-0	-128	-127
10000001	129	-1	-127	-126
⋮	⋮	⋮	⋮	⋮
11111101	253	-125	-3	-2
11111110	254	-126	-2	-1
11111111	255	-127	-1	-0

5.2 定点数与浮点数

5.2.1 定点数

> 定义

定点数即小数点位置固定的数。

定点机	小数	整数
原码	$-(1-2^{-n}) \sim +(1-2^{-n})$	$-(2^n-1) \sim +(2^n-1)$
补码	$-1 \sim +(1-2^{-n})$	$-2^n \sim +(2^n-1)$
反码	$-(1-2^{-n}) \sim +(1-2^{-n})$	$-(2^n-1) \sim +(2^n-1)$

5.2.2 浮点数

> 浮点数一般形式

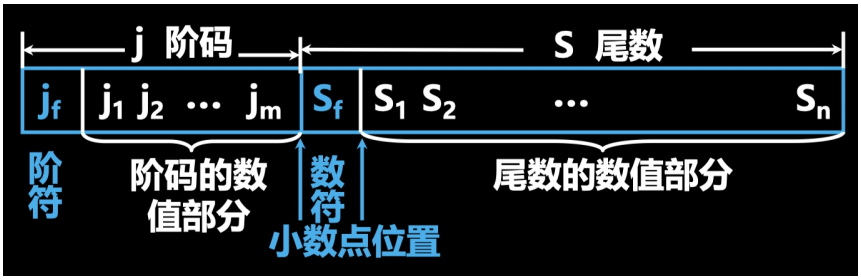
$$N = S * r^j$$

S为尾数，j为阶码，r为基数（可取2、4、8、16等）

$$\begin{aligned} N &= 11.0101 \\ &= 0.110101 * 2^{10} \end{aligned}$$

小数位第 1 位为 1，且整数部分为 0，为规格化数。

> 浮点数在机器中的表示形式

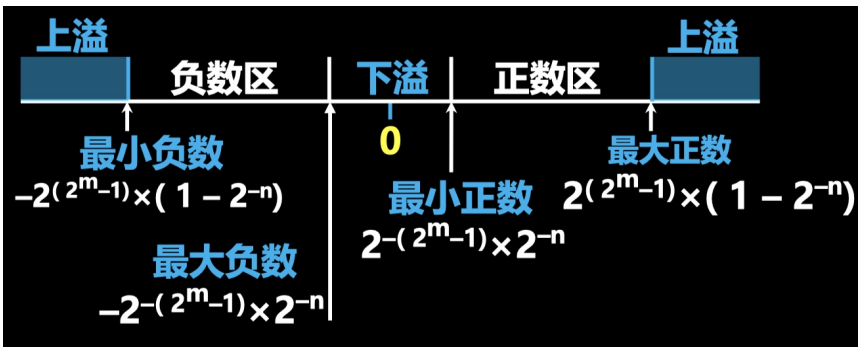


S_f : 代表浮点数的符号
n : 其位数反映浮点数的精度
m : 其位数反映浮点数的表示范围
j_f, m : 共同表示小数点的实际位置

> 浮点数的表示范围

上溢: 阶码 > 最大阶码 中断溢出处理

下溢: 阶码 < 最小阶码 按机器零处理



> 浮点数的规格化表示

1. r=2，尾数的最高位为 1
2. r=4，尾数的最高 2 位不全为 0
3. r=8，尾数的最高 3 位不全为 0

5.3 定点数移位和加减法运算

5.3.1 定点数移位运算

> 算术移位（有符号数移位）规则

符号位不变，数值位如下变化。

	码　制	添补代码
正数	原码、补码、反码	0
负数	原　码	0
	补　码	左移 添 0
		右移 添 1
	反　码	1

> 正数移位举例

$$A = +26 = +11010$$
$$[A]_{原} = [A]_{补} = [A]_{反} = 0,0011010$$

移位操作	机　器　数 $[A]_{原}=[A]_{补}=[A]_{反}$	对应的真值
移位前	0,0011010	+26
左移一位	0,0110100	+ 52
左移两位	0,1101000	+104
右移一位	0,0001101	+13
右移两位	0,0000110	+ 6

> 负数原码移位举例

$$A = -26 = -11010$$
$$[A]_{原} = 1,0011010$$

移位操作	机　器　数	对应的真值
移位前	1,0011010	– 26
左移一位	1,0110100	– 52
左移两位	1,1101000	– 104
右移一位	1,0001101	– 13
右移两位	1,0000110	– 6

> 负数反码移位举例

$$A = -26 = -11010$$
$$[A]_{反} = 1,1100101$$

移位操作	机　器　数	对应的真值
移位前	1,1100101	– 26
左移一位	1,1001011	– 52
左移两位	1,0010111	– 104
右移一位	1,1110010	– 13
右移两位	1,1111001	– 6

> 负数补码移位举例

$$A = -26 = -11010$$
$$[A]_{补} = 1,1100110$$

移位操作	机　器　数	对应的真值
移位前	1,1100110	– 26
左移一位	1,1001100	– 52
左移两位	1,0011000	– 104
右移一位	1,1110011	– 13
右移两位	1,1111001	– 7

> 移位总结

定点数移位，符号位不变。左移时，高位丢有效位，结果出错；右移时，低位丢有效位，影响精度。

原码: 高位有效位 - 1，低位有效位 - 1
反码: 高位有效位 - 0，低位有效位 - 0
补码: 高位有效位 - 0，低位有效位 - 1

> 算术移位和逻辑移位

1. 算术移位: 有符号数的移位
2. 逻辑移位: 无符号数的移位
    - 逻辑左移: 低位补 0，高位移丢
    - 逻辑右移: 高位补 0，低位移丢

5.3.2 定点数加减法及溢出判断

> 定点数加减法

连同符号位一起相加，符号位产生的进位自然丢掉。

1. 加法

整数 $[A]_{补} + [B]_{补} = [A+B]_{补} \ (\bmod 2^{n+1})$

小数 $[A]_{补} + [B]_{补} = [A+B]_{补} \ (\bmod 2)$

2. 减法

$A-B = A+(-B)$

整数 $[A-B]_{补} = [A+(-B)]_{补} = [A]_{补} + [-B]_{补}$

小数 $[A-B]_{补} = [A+(-B)]_{补} = [A]_{补} + [-B]_{补}$

> 溢出

运算结果超出计算机字长所能表示范围的情况，称为溢出。

下述两个例子中的补码运算均发生了溢出。

练习：设 $x = \frac{9}{16}$ $y = \frac{11}{16}$，用补码求 x+y

$x + y = -0.1100 = -\frac{12}{16}$ 错

练习：设机器数字长为 8 位（含 1 位符号位）且 A = – 97, B = +41，用补码求A – B。

$A - B = +1110110 = +118$ 错

> 溢出判断1 —— 符号相同两数相加减

参加运算的两个数（补码形式）符号相同，若其结果的符号与原操作数的符号不同，则发生了溢出。

$$A = -0.1011 \quad B = -0.0111$$

$$[A]_{补} = 1.0101$$
$$[B]_{补} = 1.1001$$

$$[A]_{补} + [B]_{补} = 10.1110 = 0.1110$$

两数符号位相同，结果的符号位发生变化，溢出。

> 溢出判断2 —— 最高位进位与符号位

参加运算的两个数，符号位可以不同，如果运算结果中最高数值位的进位与符号位的进位不同，则发生了溢出。

$$A = -0.1011 \quad B = -0.0111$$

$$[A]_{补} = 1.0101$$
$$[B]_{补} = 1.1001$$

$$[A]_{补} + [B]_{补} = 10.1110 = 0.1110$$

最高数值位无进位，符号位有进位，溢出。

> 溢出判断3 —— 两位符号位

参加运算的两个数，均采用双符号位（变形补码），如果结果的两位符号位不同，则发生了溢出。

$$A = -0.1011 \quad B = -0.0111$$

$$[A]_{补} = 11.0101$$
$$[B]_{补} = 11.1001$$

$$[A]_{补} + [B]_{补} = 110.1110 = 10.1110$$

输出结果两位符号位不同，溢出。

无论是否发生溢出，双符号位的高位总代表真正的符号，如果该位为 1，表示结果为负，此时称为下溢，如果该位为 0，表示结果为正，此时称为上溢。

5.4 定点数乘除法运算

5.4.1 定点数乘法运算

> 定点数乘法分类

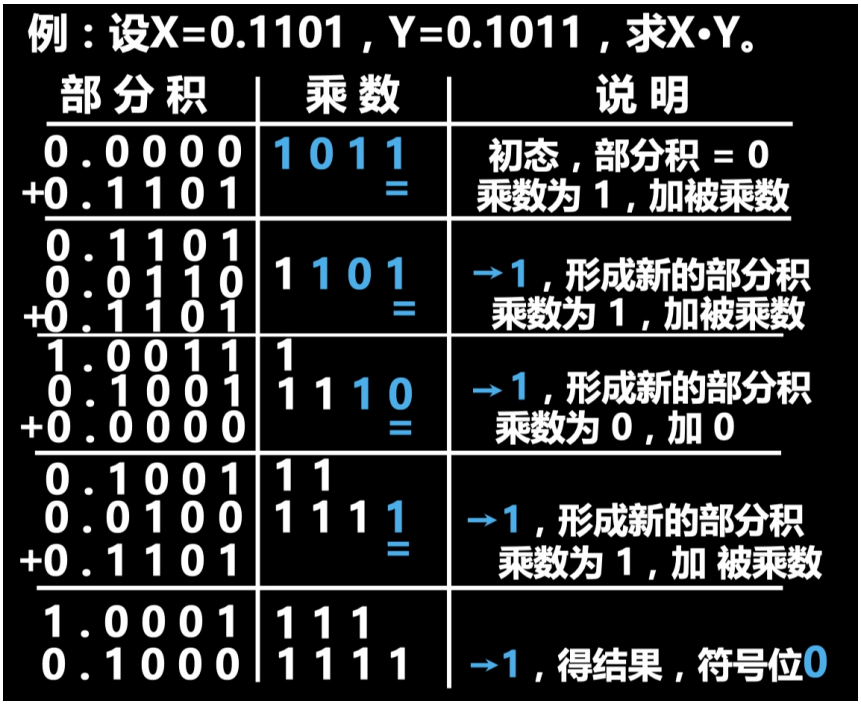
1. 定点数一位乘法
   - 定点原码一位乘法
   - 定点补码一位乘法
     - 校正法
     - 布斯法（booth）
2. 定点数二位乘法
   - 定点原码两位乘
   - 定点补码两位乘

> 定点原码一位乘性质

1. 右移代替左移。部分积相加时最低位只用一次，此位将来不再参与运算，因此可将部分积寄存器右移一位。
2. 乘数的各个位从右向左均只用一次，最低位不再使用，因此部分积右移时，乘数寄存器也可右移一位，即可用乘数寄存器的最高位接收部分积移出的位。

> 定点原码一位乘举例

1. 乘积的符号位 $x_0\ xor\ y_0 = 0\ xor\ 0 = 0$
2. 数值部分按绝对值相乘



> 定点原码一位乘特点

1. 用移位的次数判断乘法是否结束

2. 绝对值运算
3. 逻辑移位

5.4.2 定点数除法运算

> 定点数除法分类

1. 定点原码一位除
    - 恢复余数法
    - 加减交替法
2. 定点补码一位除

> 恢复余数法

特点:

1. 左移被除数（余数）来代替右移除数。
2. 速度慢，控制方式复杂。

上商规则:

1. 将部分余数 R_i 与除数 Y 相减，判断余数的符号，确定商的值。
2. 差为正: 商上 1，不恢复余数，部分余数左移 1 位。
3. 差为负: 商上 0，恢复余数，部分余数左移 1 位。

> 例题

$X = 0.1011$, $Y = 0.1101$, 求X/Y

采用双符号位进行计算，高位符号位代表了实际极性，所以可以防止因溢出而丢掉正确的符号。

最终结果: $X/Y = 0.1101$，符号为正，余数为 $0.0111 * 2^{-4}$。

2^{-4} 是因为计算过程中左移了 4 次。

> 总结

1. 运算过程中，被除数寄存器不断左移，右侧补零
2. 商的初值为0，不断左移
3. 可以认为是将商的高位移入零被除数寄存器的低位部分。
4. 所以初始时可以用存放商的寄存器存放被除数的低 n 位，实现 2n 位数除以 n 位数。

5.5 浮点数运算

5.5.1 浮点加减运算步骤

$$x = S_x * 2^{j_x}$$
$$y = S_y * 2^{j_y}$$

五个步骤: 对阶、尾数相加减、规格化、舍入、判溢出

5.5.2 对阶

操作目的: 使两数阶码一致，尾数加减可以进行

- 比较两阶码，$\Delta J = J_x - J_y$
- 将小阶码数的尾数右移 $|\Delta J|$ 位，其阶码值加 $|\Delta J|$
- 注意尾数右移特点: 原码补0，补码补符号位
- 右移过程中移掉的位可用附加位电路暂时保存

5.5.3 尾数加/减运算

$$\Delta S = S_x \pm S_y$$

采用双符号位加减运算。

5.5.4 尾数规格化

操作目的: 尽可能提高精度

$$2^{-1} \leq |[S]_{原}| \leq 1 - 2^{-n}$$
$$2^{-1} \leq |[S]_{补}| \leq 1$$

> 右规

结果的两个符号位不同表示尾数结果溢出，右移 1 位尾数，阶码 $J+1$。

> 左规

结果的两个符号位相同，如是原码且数值位最高位为 0，则左移；如是补码且数值位与符号位相同也需左移。同时阶码减去移动的位数。

5.5.5 舍入

> 截断法

直接丢掉附加位电路中的数据。

> 恒置 1 法

类似于上取整，把计算结果最后一位置 1。

> 0舍1入法

类似于四舍五入，如果附加位电路中最高位为1，则对数值进 1。

5.5.6 判断阶码溢出

- 阶码下溢，则置运算结果为机器零
- 阶码上溢，则置溢出标志，报警中断

5.5.7 例题

$$X = 2^{0010} * (0.11011011), Y = 2^{0100} * (-0.10101100)\text{，求}X+Y$$

- 对阶操作

$$\Delta J = [J_x]_{补} + [-J_y]_{补} = 11110$$
$$[S_x]_{补} = 00.00110110\ 11, J = 00100$$

- 尾数相加

$$[S_x]_{补} + [S_y]_{补} = 00.0011011011\ 11 + 11.01010100 = 11.10001010\ 11$$

- 规格化操作

符号位相同，左规，移 1 位，$ans = 11.00010101\ 10, J = 00011$。

- 舍入

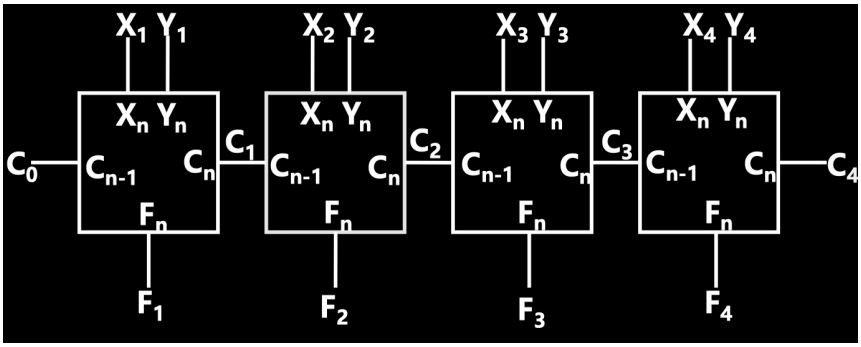
采用 0 舍 1 入法，得到 $[S]_{补} = 11.00010110, S = -0.11101010$。

- 判溢出

未溢出，最终结果为 $X + Y = 2^{0011} * (-0.11101010)$。

5.6 加法器与算术逻辑单元

5.6.1 n 位串行加法器



$$C_1 = X_1Y_1 + (X_1 + Y_1)C_0$$
$$C_2 = X_2Y_2 + (X_2 + Y_2)X_1Y_1 + (X_2 + Y_2)(X_1 + Y_1)C_0$$
$$\cdots$$

5.6.2 超前进位加法器

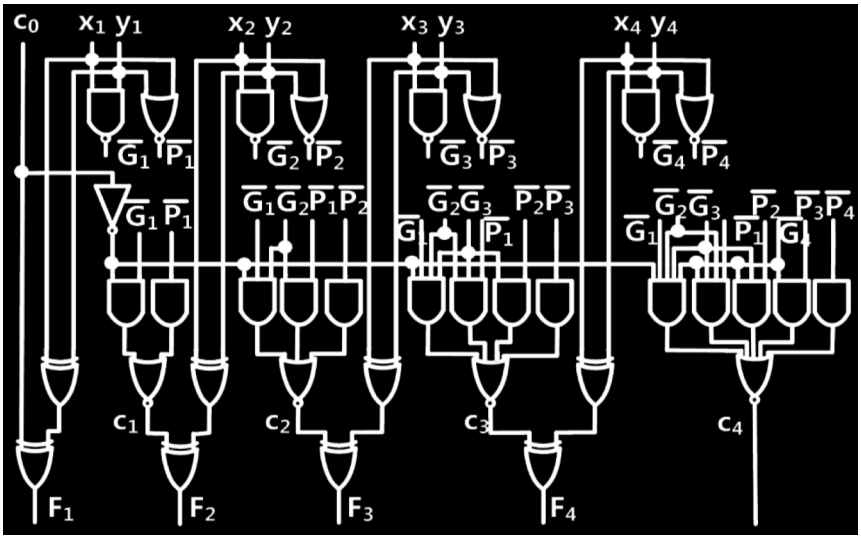
令 $P_i = X_i + Y_i, G_i = X_iY_i$，则

$$C_1 = G_1 + P_1C_0$$
$$C_2 = G_2 + P_2G_1 + P_2P_1C_0$$

$$C_1 = G_1 + P_1C_0 = \overline{\overline{P_1} + \overline{G_1}\,\overline{C_0}}$$

$$C_2 = G_2 + P_2G_1 + P_2P_1C_0 = \overline{\overline{P_2} + \overline{G_2}\,\overline{P_1} + \overline{G_1}\,\overline{G_2}\,\overline{C_0}}$$

因此可以改进串行加法器，使得加法运算速度与运算数字长度无关。



第六章 指令系统

6.1 指令系统概述

6.1.1 指令系统

物理的计算机只能够执行机器语言程序，组成程序的每一条语句称作一条机器指令，一种计算机能够执行的机器指令的集合就是这种计算机的指令系统。

6.1.2 机器指令格式

> 指令的一般格式

操作码字段	地址码字段

> 操作码

操作码位数反映机器指令数目，内容反映机器做什么操作。

操作码类别:

- 长度固定，如 RISC 机器，指令规整，译码简单
- 长度可变，操作码分散在指令字的不同字段中，控制器设计复杂

> 地址码

地址码用来指定该指令操作数的地址、结果的地址，以及（可能有的）下一条指令的地址。

例如指令: MOV AX, [40]，其中 “40” 指明了要操作的源操作数的地址，AX 指明了目的操作数的地址。

> 操作码扩展技术

通过操作码扩展（对应的地址码长度收缩）技术，能够有效地缩短指令的平均长度。

6.1.3 操作码设计举例

> 例题

假设一台计算机指令字长 16 位，操作码与地址码都为 4 位，请列出几种可能的操作码设计方法。

> 固定操作码

> 扩展操作码

扩展方案非常多，下面给出两种扩展方式。

6.1.3 指令字长

- 指令字长取决于操作码长度、地址码长度和地址码个数。指令按照字长是否可变分为两种:
  - 指令字长固定: 指令字长 = 存储字长
  - 指令字长可变: 按字节的倍数变化
- 指令字长可变，导致控制电路复杂，多字长指令需要多次访存，应尽量把常用指令设计为单字长或短字长指令。

6.2 操作类型

6.2.1 数据传送类指令

数据传送类指令主要分为 “源地址” 和 “目的地址” 两个部分。

1. 寄存器-寄存器 MOV AX, BX
2. 寄存器-存储器 MOV [20], AX（STORE指令）
3. 存储器-寄存器 MOV AX, [20]（LOAD指令）
4. 存储器-存储器 MOV [20], [30]
5. 堆栈操作: PUSH AX（入栈），POP AX（出栈）
6. 清零、置1: MOV AX,0（清零），MOV AX,1（置1）

6.2.2 运算类指令

> 算术运算

加、减、乘、除、求补、浮点、十进制运算

- ADD AX, 20

- DIV AX, 3

> 逻辑运算

与、或、非、异或

- AND AX, 30
- XOR AX, 30

> 其他运算

位测试、位清楚、位求反

6.2.3 移位指令

一般来说，有 8 种移位指令:
算术左移、算术右移、逻辑左移、逻辑右移、小循环左移、小循环右移、大循环左移、大循环右移

6.2.4 转移指令

> 无条件转移指令

直接跳转到某处，不取决于任何条件。类似 C 中的 goto 语句，例如 JMP LOOP。

> 条件转移指令

根据机器当前的程序状态字中的某位来决定是否执行转移，例如 JZ LOOP。

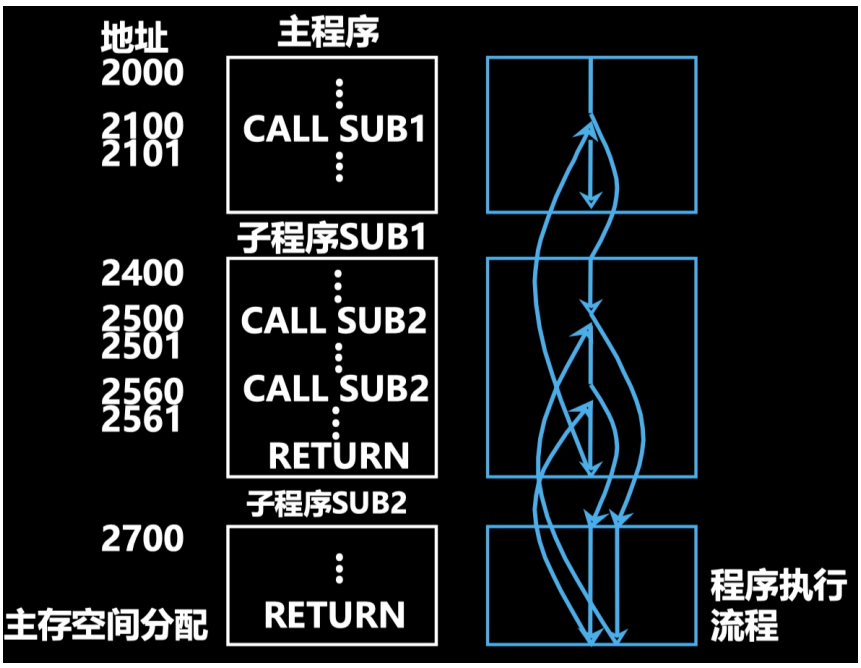
JZ LOOP: 如果当前指令运算结果为0，则会跳转到 LOOP 标号处。

> 调用与返回指令

类比 C 程序中的函数调用，以及函数返回。例如 CALL PRO1、RET

子程序调用特点:

1. 子程序可以在多处被调用
2. 子程序调用可以嵌套
3. CALL 与 RET 指令配对使用
4. 要妥善保存子程序的返回地址



> 陷阱（TRAP）指令

- 一旦机器运行出现意外故障（未定义指令、除0、设备故障、电压不稳），计算机发出陷阱信号（陷阱隐指令），暂停当前指令的执行，转入故障处理程序。陷阱指令不提供给用户使用，由机器自动执行。
- 也有某些机器提供陷阱指令，例如 IBM-PC 提供的 INT xx 软中断指令，用来完成系统调用。

6.2.5 输入输出指令

对 I/O 单独编址对计算机，设置有专门的输入输出指令，用来操纵外设，如下所示。

- IN AX, [20]（外设端口 → CPU 的寄存器）
- OUT DX, AX（CPU 的寄存器 → 外设端口）

6.2.6 其他指令

1. 停机指令、空操作指令、开中断指令、关中断指令、置条件码指令
2. 字符串传送、字符串比较、字符串查询
3. 特权指令（操作系统用）
4. 向量指令
5. 多处理机指令

6.3 数据寻址方式

数据寻址有多种，需要在指令中明确指出采用哪一种寻址方式，可以专门设置一个寻址方式特征字段，或纳入地址码中。

操作码	寻址特征	形式地址

形式地址 A: 指令字中的地址
有效地址 EA: 操作数的真实地址

有效地址由形式地址根据寻址方式来确定。

在大多数机器中，指令字长 = 存储字长 = 机器字长。

6.3.1 立即寻址

又称为立即数寻址，即指令中的形式地址部分不是一个操作数的地址，而是操作数本身，因此该指令的执行过程不需要寻址。

如指令 "MOV AX, 20H"，将 20H 放入 AX 寄存器中，其中立即数可正可负，负数需要用补码来表示。

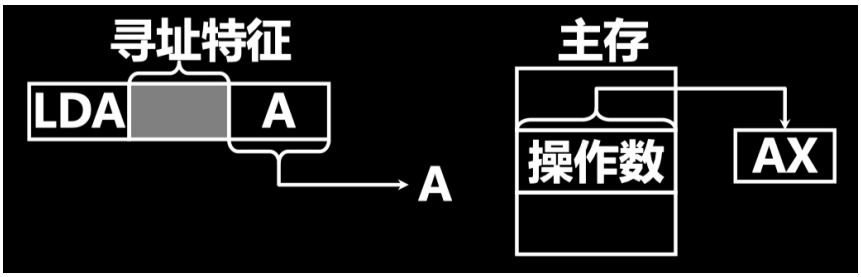
> 特点

- 立即寻址的指令执行阶段不访存
- A 的位数限制了立即数的范围

6.3.2 直接寻址

指令中的形式地址部分即为有效地址，即 EA = A。

例如指令 "MOV AX, [20H]"，即是将 [20H] 地址中的数据传输到 AX 寄存器中。



> 特点

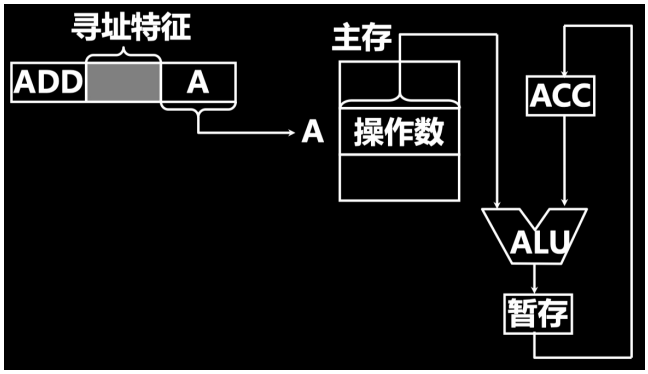
- 执行阶段访问一次存储器

- A 的位数限制了该指令操作数的寻址范围
- 操作数的地址不易修改

6.3.3 隐含寻址

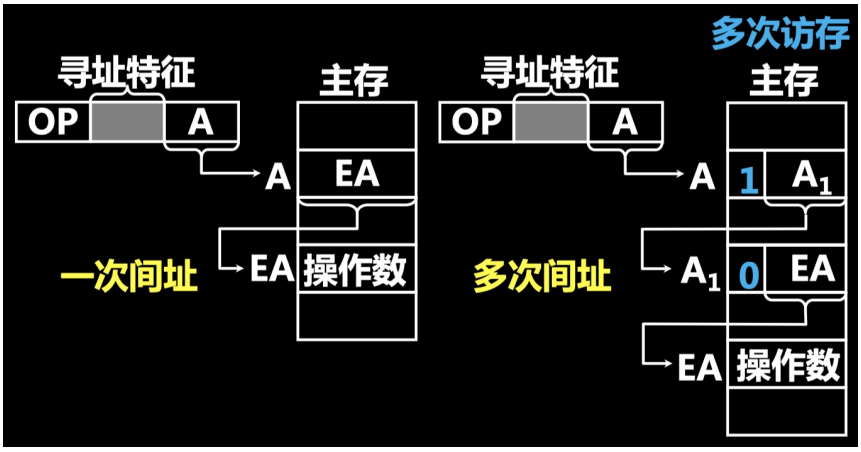
指令中不直接给出操作数地址，操作数地址通常隐含在操作码或某个（约定）寄存器中。

如 ADD 操作，即是默认另一个操作数在 ACC 中。将 A 地址中的数取出与 ACC 中的数进行加法运算。



6.3.4 间接寻址

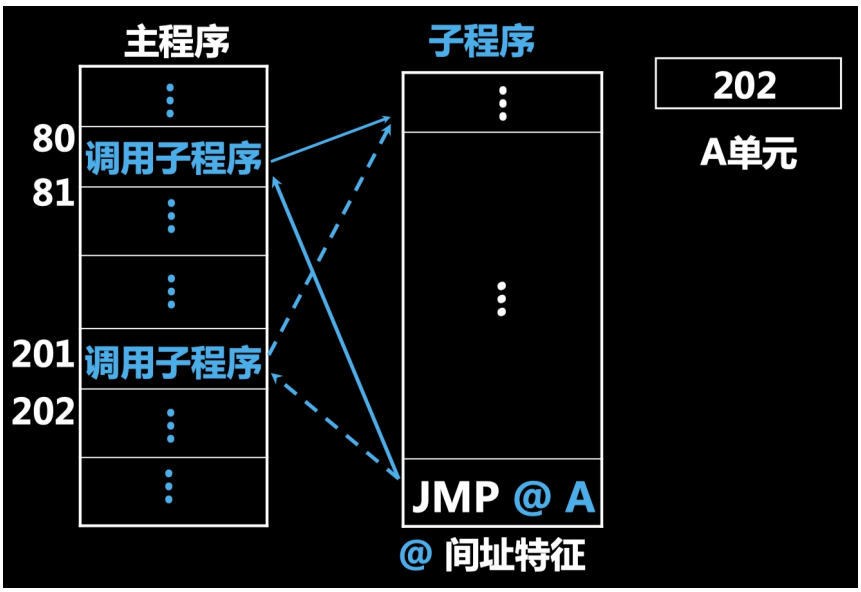
指令中的形式地址不是操作数的地址，而是 “操作数地址的地址”。



其中在多次间址时，需要查看数据单元格中的第一位数据，如果为 1 则表示继续间址，如果为 0 则为真正地址。

> 特点

1. 可以扩大寻址范围
2. 便于编制转移程序

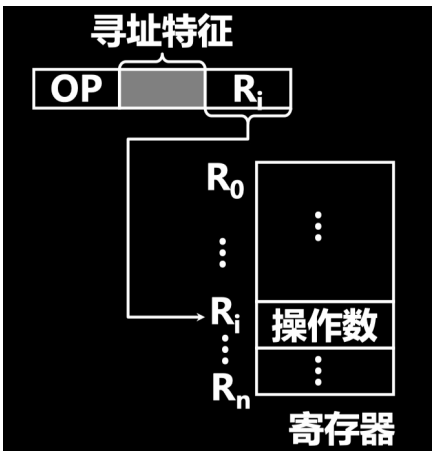


JMP @A 指令，表示跳转到 A 单元中所指示的地址，为间接寻址。正因为这条间址指令，我们转入子程序时只需修改 A 单元中数据，即可实现从子程序再跳转回主程序的功能。

6.3.5 寄存器寻址

指令中的形式地址直接指出寄存器的编号，操作数存储于寄存器中，即 $EA = R_i$，有效地址即为寄存器编号。

如 "MOV AX, BX" 指令，即是将 BX 寄存器中的数据送到 AX 中。



> 特点

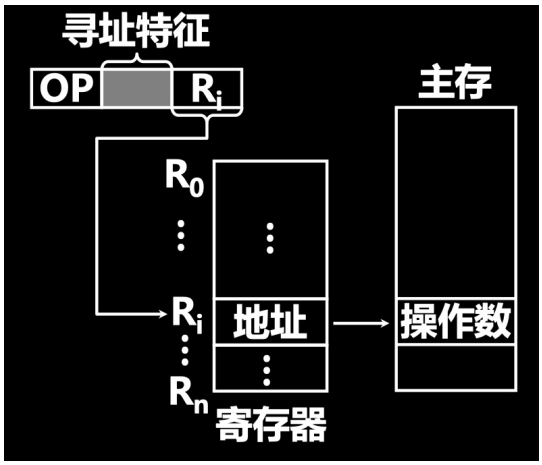
- 执行阶段不访存，只访问寄存器，执行速度快
- 寄存器个数有限，可缩短指令字长

6.3.6 寄存器间接寻址

指令中的形式地址为寄存器的编号，寄存器的内容是操作数的有效地址。

$EA = (R_i)$，有效地址在寄存器中。

如指令 "MOV AX, [BX]"，就是根据 BX 寄存器中的地址，去主存中访问对应数据，并将数据存入 AX 寄存器中。



> 特点

有效地址在寄存器中，操作数在存储器中，执行阶段访存便于编制循环程序。

- 可以访存一次，就将寄存器数值加1，实现类似于数组的访问。

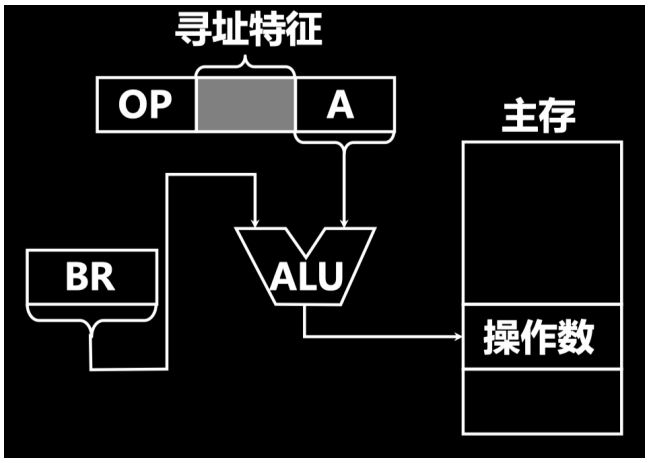
6.3.7 基址寻址

指令中的形式地址与基址寄存器内容之和为有效地址。

1. 采用专用寄存器作为基址寄存器（隐式）
2. 采用通用寄存器作为基址寄存器（显式）

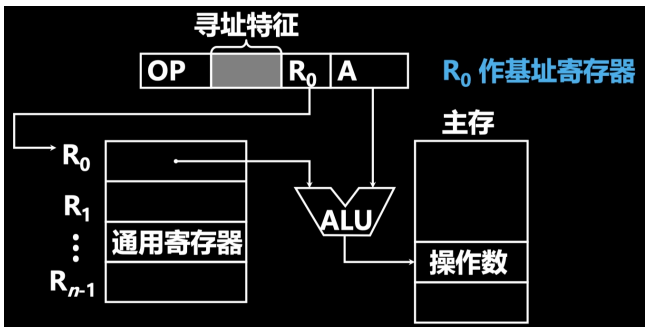
> 专用寄存器基地寻址

$EA = (BR) + A$，其中 BR 为基址寄存器



> 通用寄存器基址寻址

需要给出寄存器编号作为基址寄存器。



- 可指定由哪个通用寄存器作为基址寄存器。
- 在程序的执行过程中 R_0 内容不变，形式地址 A 可变。

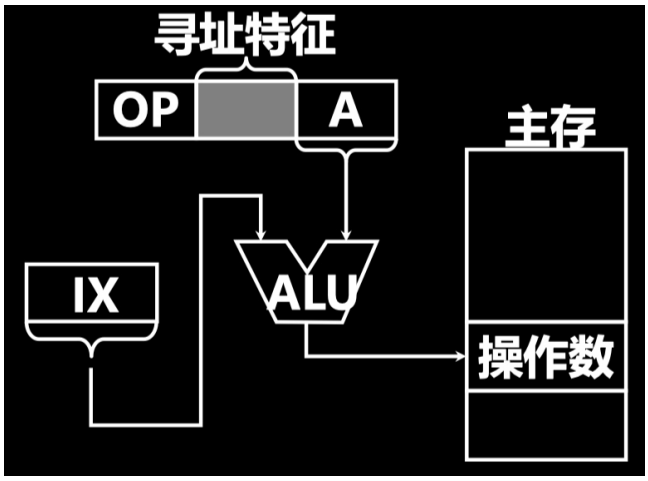
> 特点

- 可扩大寻址范围
- 有利于多道程序
  - 将程序开头地址定位于某个寄存器中，即可实现更换通用寄存器，即实现程序切换。
- 基址寄存器内容由操作系统或管理程序确定

6.3.8 变址寻址

指令中的形式地址与变址寄存器内容之和为有效地址。

EA = (IX) + A，与基地寻址非常相似。唯一的不同点在于基址寄存器中的值由操作系统或管理程序决定，而 IX 的内容由用户指定。



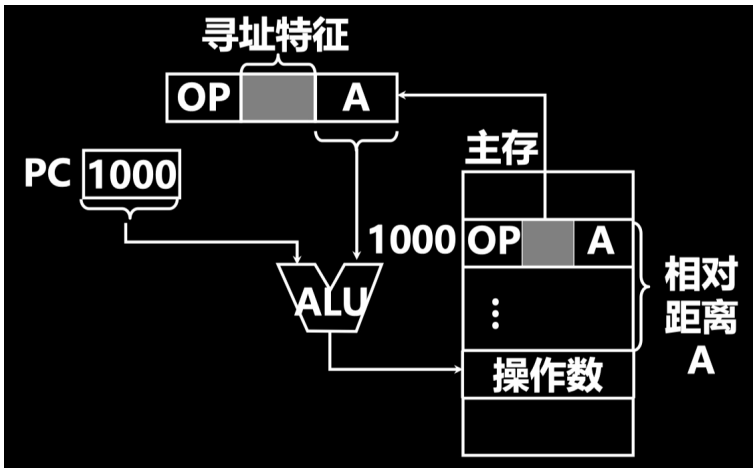
> 特点

- 可以扩大寻址范围
- 由于 IX 的内容由用户指定，因此在程序执行过程中，IX 内容可变，可以用于设计处理一些数组问题。

6.3.9 相对寻址

有效地址为程序计数器 PC 的值与形式地址之和。

EA = (PC) + A，A 的位数决定操作数的寻址范围，可编写浮动程序（程序放哪都能执行）。



6.3.10 堆栈寻址

堆栈有两个操作，分别是 PUSH（压栈） 与 POP（弹栈）。进栈与出栈的操作分别由 SP 指针的加减完成。

第七章 CPU 的结构与功能

7.1 控制器概述

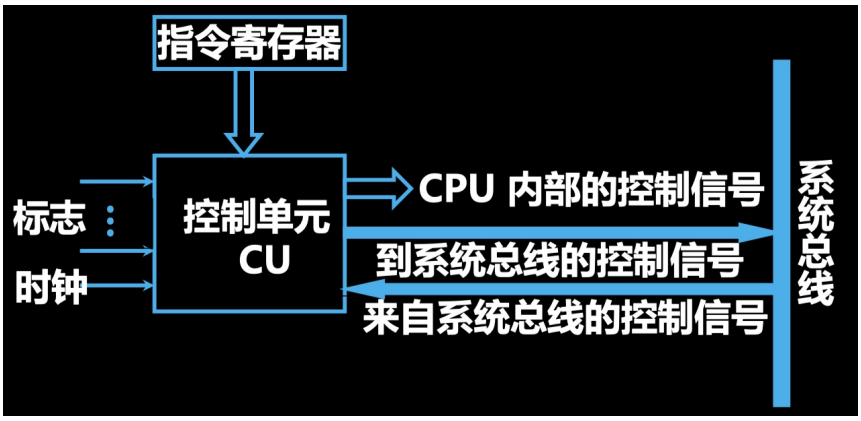
7.1.1 控制器的功能

1. 取指令
2. 分析指令

3. 执行指令，发出各种操作命令
4. 总线管理
5. 处理中断、异常等特殊情况

> 控制单元的外特性

控制信号分为 CPU 内部的控制信号，以及传到系统总线上的外部控制信号。



7.1.2 指令周期

> 概念

- 计算机读取并执行完一条指令的全部时间，称为指令周期
- 指令周期大体可以分成四个阶段
  - 取指周期: 从内存中取得指令，并分析指令。
  - 间址周期: 间接寻址的指令都包含间址周期。发出指令中的形式地址，从内存中读回有效地址。
  - 执行周期: 除空操作以及 NOP 指令外，所有指令都包括执行周期，用以完成指令的主要功能。
  - 中断周期: 如果在前面三个过程中有设备发出了中断请求，则在执行完上述三个周期后，执行中断隐指令。
- 并不是每一条指令都包括完整的四个阶段。

7.1.3 机器周期

> 概念

指令执行的每个阶段，称为一个机器周期，也称为工作周期。

> 如何确定机器周期的长度

- 每条指令的执行步骤
- 每一步骤所需的时间
- 以完成最复杂、最慢指令功能的时间为准

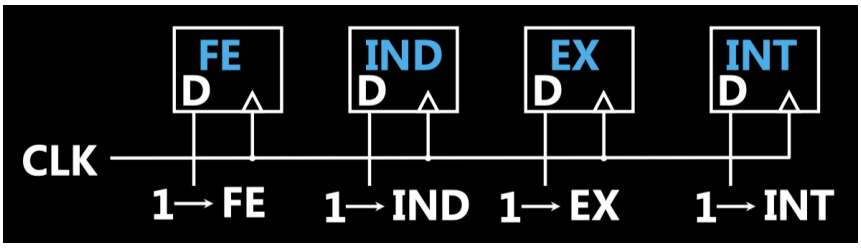
> 统一机器周期的确定

- 计算机中，最慢指令为访存指令，因此以一次访存所需时间作为机器周期。
- 若 “指令字长 = 存储字长”，则 “访存周期 = 机器周期”。

> CPU 工作周期的标志

我们需要根据下述 4 个触发器的输出值来确定 CPU 当前所处的工作周期。

其中，若 CPU 处于取指周期，则会将 FE 置 1，若为间址周期，则会置 IND 为 1，以此类推。



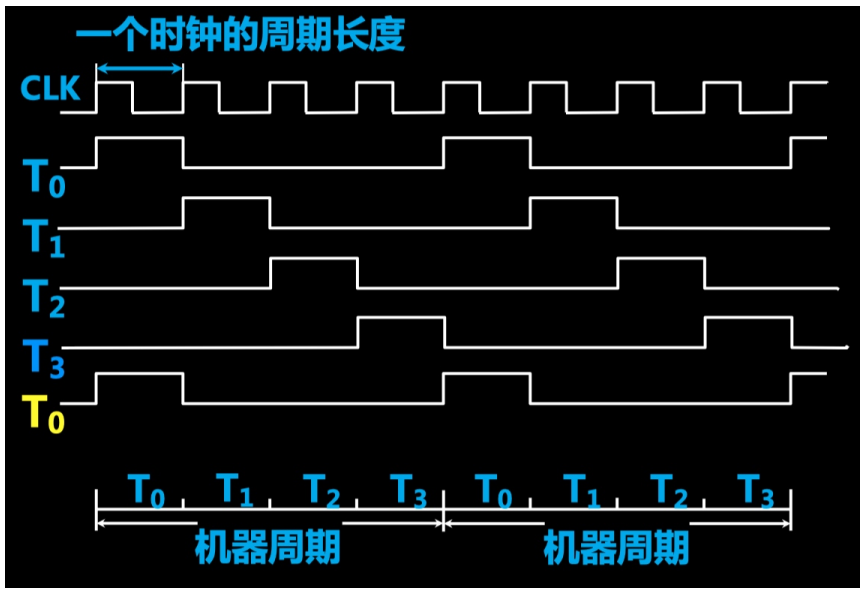
7.1.4 时钟周期

> 概念

- 将一个机器周期分成若干个时间相等的时间段，每段称为一个时钟周期。（节拍、状态）
- 时钟周期是控制计算机操作的最小单位时间。
- 用时钟周期控制微操作命令，每个时钟周期产生一个或几个微操作命令。
- 常用 T_0, T_1, T_2, T_3 来描述时钟周期。

> 时钟周期的产生方法

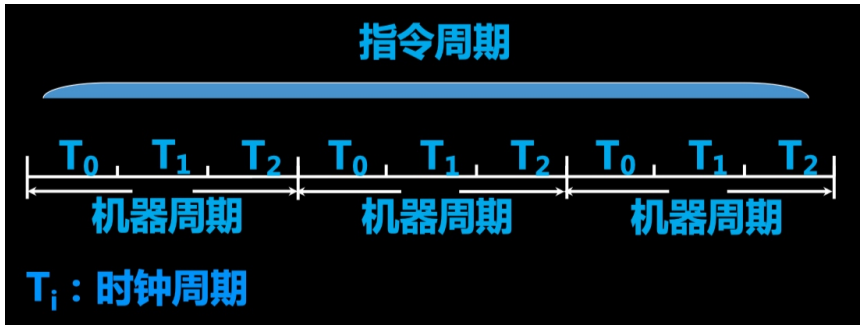
用计数器和译码器来控制每次执行的微操作。



7.1.5 多级时序系统

指令周期、机器周期、时钟周期（节拍、状态）组成多级（三级）时序系统。

其中如果每个机器周期都包含了相同数量的时钟周期，则称为定长机器周期，否则为不定长机器周期。一个指令周期包含了若干个机器周期，一个机器周期包含了若干个时钟周期。



7.1.6 超线程技术与双核处理器

> 超线程技术的概念

- 超线程技术（Hyper-Threading）是 Intel 公司提出的一种提高 CPU 性能的技术，可将一个物理 CPU 当作两个逻辑 CPU 使用，同时执行多个线程，从而提高效率。
- 超线程处理器内部的两个逻辑处理器共享一组处理器执行单元，运算能力提高 30%。
- 超线程技术可将处理器内部的闲置资源利用起来，当并行执行两个线程时，负责处理第二个线程的逻辑处理器，仅使用运行第一个线程的处理器不使用的资源。

> 超线程技术的实现

- 实现超线程技术的前提是需要五大支持：CPU、主板芯片组、主板 BIOS、操作系统、应用软件
- 执行单个线程时，该技术反而下降，因为打开超线程后，处理器内部缓存被划分为几个区

域，互相共享内部资源，单个子系统性能反而下降。
- 超线程技术最早出现在 2002 年的 pentium4 处理器，以及后续 2009 的 Core i 系统处理器。

> 双核处理器

- 双核处理器是在一个处理器芯片上集成了两个 “物理的” 运算核心。
- 1989年， Intel 工程师首先提出了双核的概念，1995 年首先发布了双核产品，2005 年首先提供主流价位的双核处理器。
- 双核处理器并不能达到同频率两个单核处理器的计算能力，IBM 公司曾对比过 AMD 和 Intel 双核处理器的性能，在多线程任务环境下，大约比单核提高 60%。

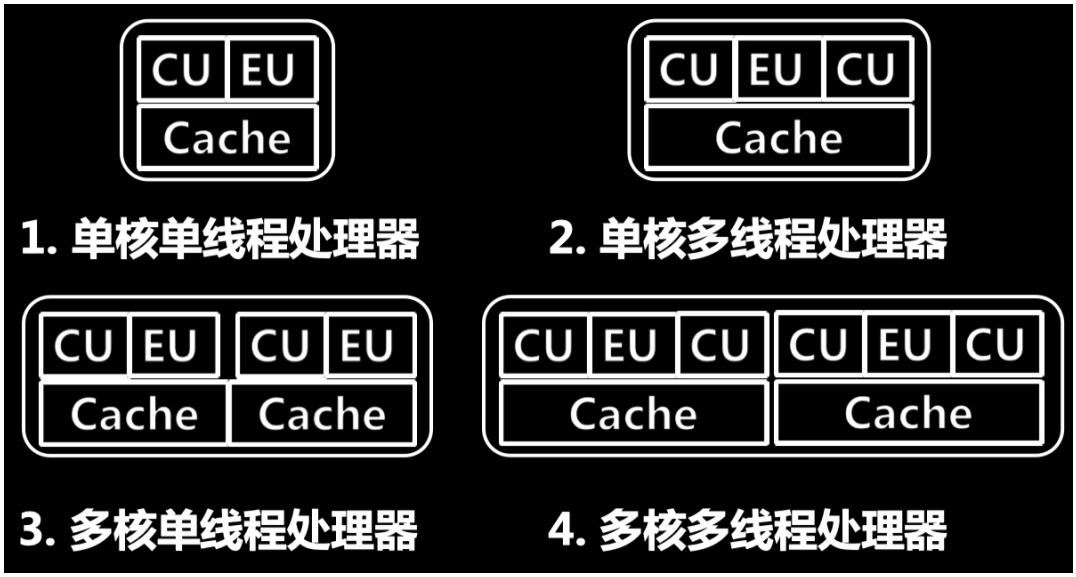
> 超线程与双核处理器的区别

- 开启了超线程技术的单核处理器与双核处理器在操作系统中都被识别为两个处理器，二者的区别在于，前者是两个 “逻辑” 处理器，后者是两个 “物理” 处理器。
- 超线程的逻辑处理器并没有独立的执行单元、整数单元、寄存器甚至缓存，它们在运行时仍需要共用执行单元、缓存和总线接口。
- 执行多线程时，超线程的两个处理器交替工作，当争用某资源时，一个线程必须暂停并让出资源。

> 多核多线程技术

处理器有多个物理内核，每个内核采用超线程技术。

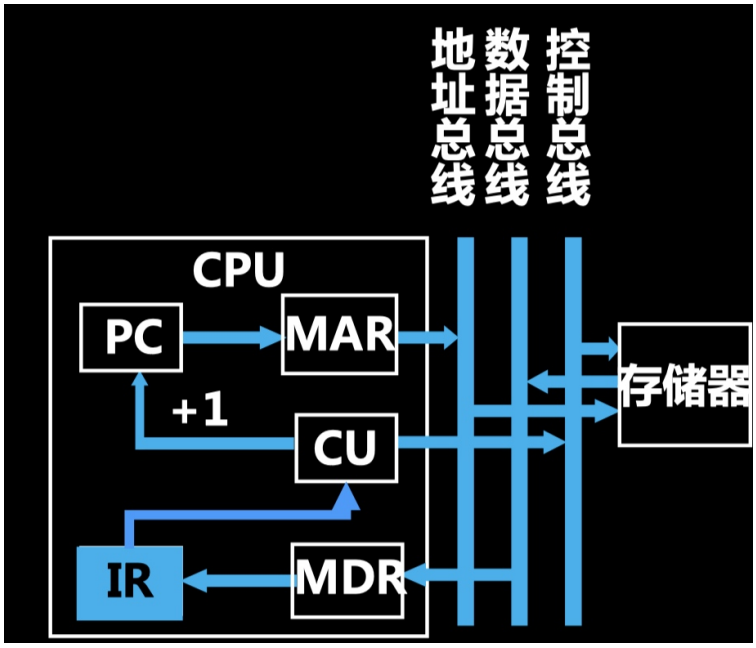
> 几种微处理器的内部结构



7.2 指令周期的操作命令分析

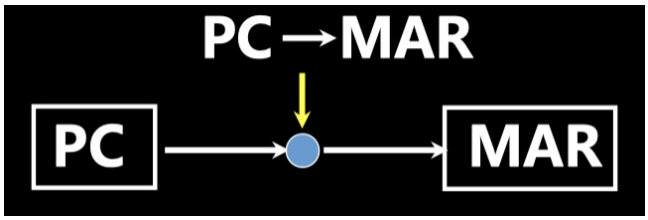
7.2.1 取指周期

1. PC → MAR → 地址线
2. 1 → R（CU 发信号到控制总线）
3. M(MAR) → MDR
4. MDR → IR
5. OP(IR) → CU（将操作码部分交给 CU 译码）
6. (PC)+1 → PC



> 微操作数据传递的原理

以下图 PC 将数据传给 MAR 为例，数据传递通道中间有一个受控门，只有当受控门中的受控端收到有效信号后，受控门就会打开，PC 的数据就会流向 MAR。



> 控制单元根本任务

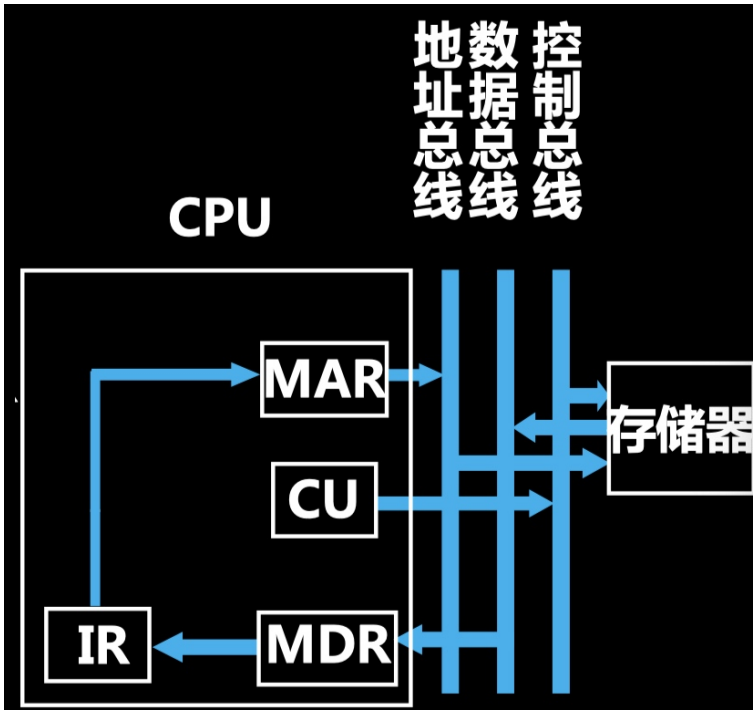
1. 根据指令要求依次发出一系列控制信号
2. 控制信号依次打开一系列受控门
3. 依次打开的受控门导致一系列数据流动

控制信号也称为微操作命令。

7.2.2 间址周期

1. Ad(IR) → MAR（将指令中的地址部分送到 MAR 中）

2. 1 → R（CU 发信号到控制总线）
3. M(MAR) → MDR（从主存中读回数据到 MDR 中）
4. MDR → Ad(IR)（将数据写回到指令的地址部分）



7.2.3 执行周期

> 非访存指令

指令的执行阶段，都不会访存到存储器。

- CLA

清 0 指令，微操作是 $0 \to ACC$，将累加器清零。

- COM

取反指令，微操作是 $\overline{ACC} \to ACC$，累加器内容按位取反。

- SHR

算术右移，有两个微操作。

1. $L(ACC) \to R(ACC)$，表示累加器每一位都移到右边一位。
2. $ACC_0 \to ACC_0$，表示最左边的符号位保持不变，因此是算术右移。

- CSL

循环左移，有两个微操作。

1. $R(ACC) \to L(ACC)$，表示累加器每一位都移到左边一位。

2. $ACC_0 \to ACC_n$，表示最左边的位移动到最右边，因此是循环左移。

- STP

停机指令，微操作是 $0 \to G$，一旦控制信号有效机器就会停掉。

> 访存指令

- 加法指令（ADD X）

1. $Ad(IR) \to MAR$（将指令的地址部分送到 MAR）
2. $1 \to R$（读信号）
3. $M(MAR) \to MDR$（将加法所需数据从内存读出）

4. $(ACC) + (MDR) \to ACC$（与 ACC 中的数相加）

- 存数指令（STA X）

1. $Ad(IR) \to MAR$（将指令的地址部分送到 MAR）
2. $1 \to W$（写信号）
3. $ACC \to MDR$（将存数写入MDR）

4. $MDR \to M(MAR)$（写入数据）

- 取数指令（LDA X）

1. $Ad(IR) \to MAR$（将指令的地址部分送到 MAR）
2. $1 \to R$（读信号）
3. $M(MAR) \to MDR$（读出数据到 MDR）
4. $MDR \to ACC$（将数据写入累加器）

> 转移指令

- 无条件转（JMP X）

$Ad(IR) \to PC$（将跳转地址赋给PC）

- 条件转移（BAN X，负则转）

$ACC_0 Ad(IR) + \overline{A_0}(PC) \to PC$（$ACC_0$ 为累加器中数据的符号位，若符号位为 1，即为负数，则跳转）

7.2.4 中断周期

> 一、保存断点

中断隐指令中需要保存断点，通常断点都保存在内存的 0 号单元中。

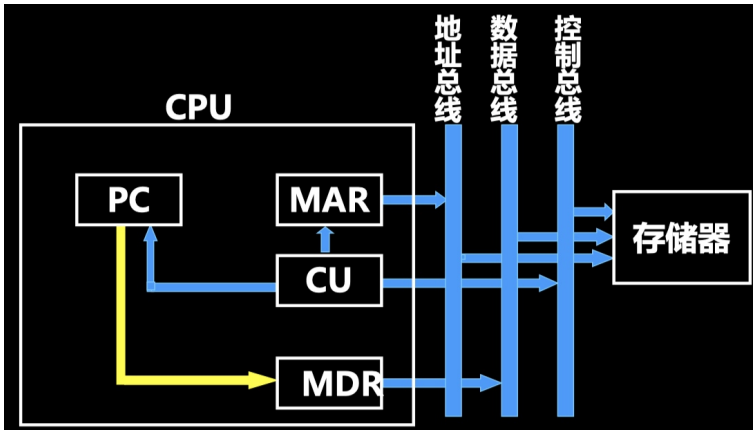
1. 0 → MAR（CU 传 0 到 MAR，作为断点保存地址）
2. 1 → W（CU 发出写命令）
3. PC → MDR（将即将要执行的指令地址送到 MDR，PC 中的内容就是断点）
4. MDR → M(MAR)（将断点写入内存中）

> 二、执行中断处理程序（PC跳转）

1. 向量地址 → PC（实现 PC 的跳转，CU 将数据流交给 PC）

> 三、硬件关中断

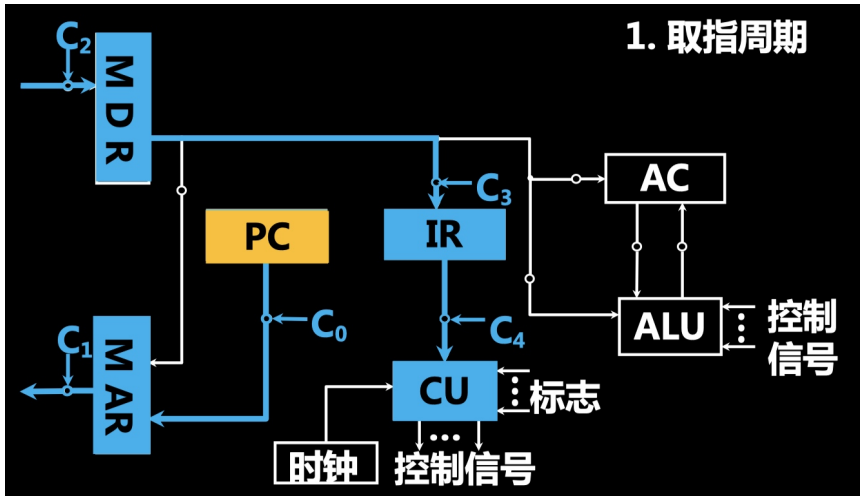
1. 0 → EINT（将中断总允许位置 0）



7.2.5 指令执行的完整分析

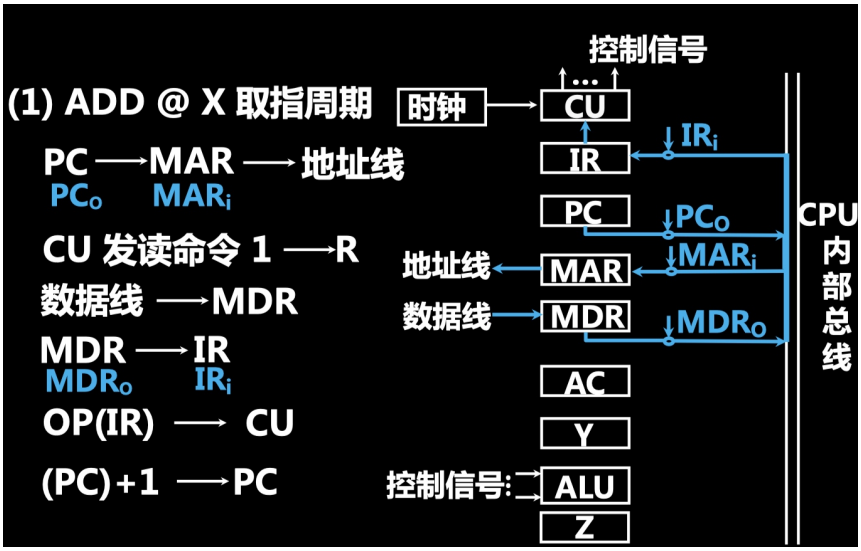
以 $ADD\ @X$ 指令为例来分析指令执行周期。

> 取指周期

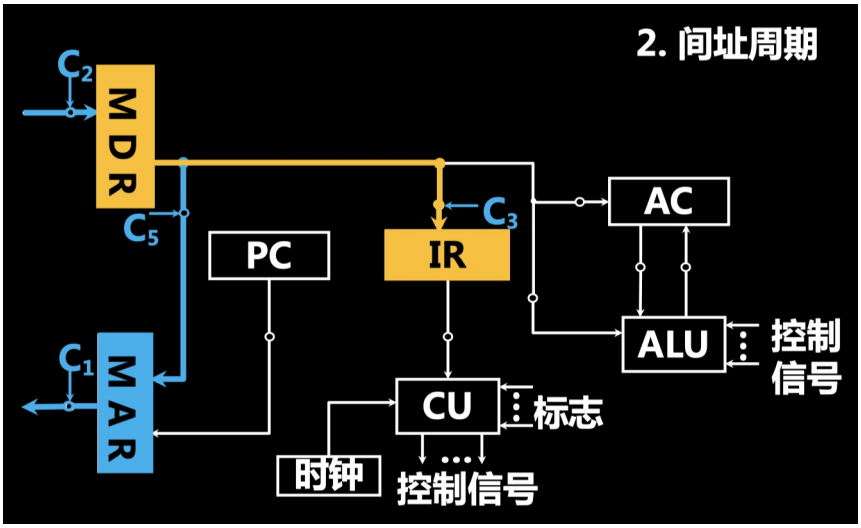


CU 下的点为各种控制信号，数据通道中的小圈为受控门，受控门收到控制信号的控制。

1. $C_0 : PC \rightarrow MAR$
2. $C_1 : MAR \rightarrow$ 地址线
3. $C_2 : M(MAR) \rightarrow MDR$
4. $C_3 : MDR \rightarrow IR$
5. $C_4 : OP(IR) \rightarrow CU$
6. $(PC) + 1 \rightarrow PC$

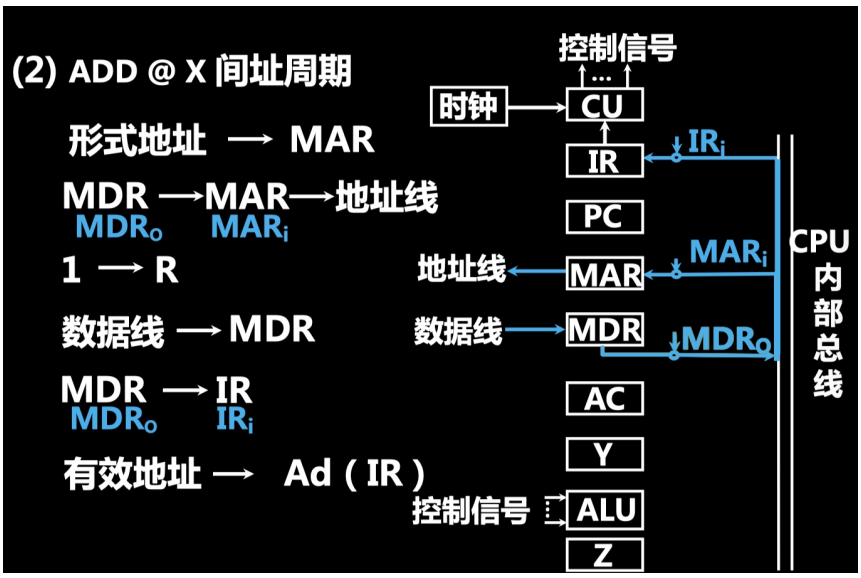


> 间址周期

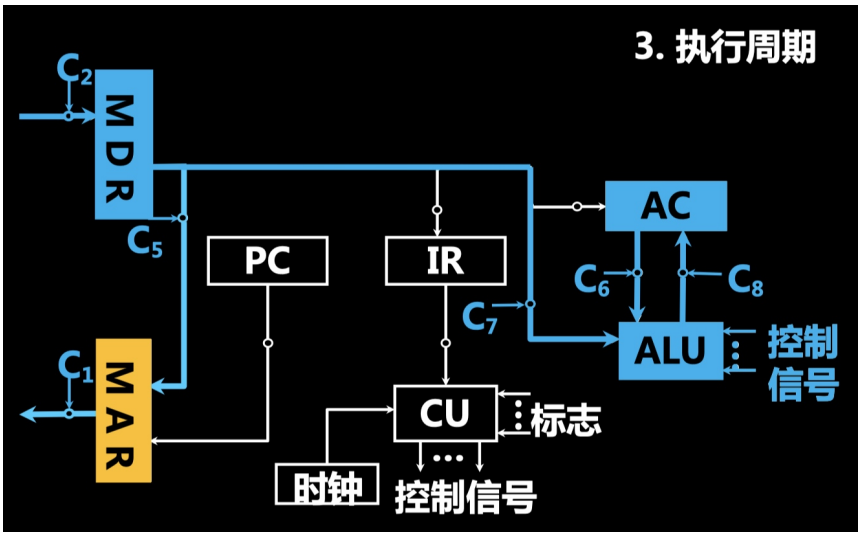


此处的指令需要间址，因此有间址周期。

1. $C_5 : MDR \rightarrow MAR$（将 MDR 指令中的形式地址传给 MAR）
2. $C_1 : MAR \rightarrow$ 地址线
3. $C_2 : M(MAR) \rightarrow MDR$（取回有效地址）
4. $C_3 : MDR \rightarrow Ad(IR)$（将有效地址送到 IR 的地址部分）

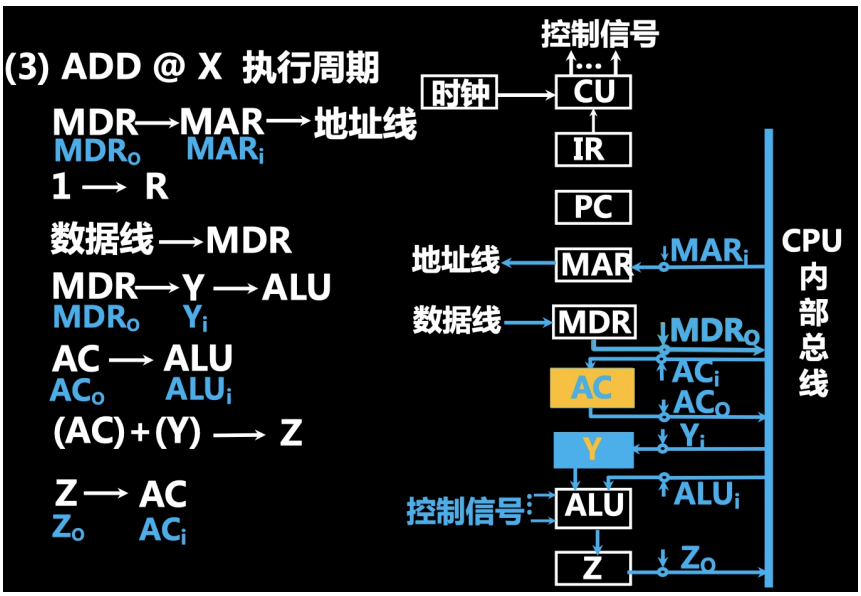


> 执行周期



接下来需要根据有效地址取回数据，再将数据与累加器中数据相加得到结果。

1. $C_5: MDR \to MAR$（将 MDR 指令中的有效地址传给 MAR）
2. $C_1: MAR \to$ 地址线
3. $C_2: M(MAR) \to MDR$（取回数据）
4. $C_7: MDR \to ALU$（将数据送到算术逻辑单元）
5. $C_6: AC \to ALU$（将累加器中数据送到算术逻辑单元）
6. $CU \to ALU$（加信号有效）
7. $C_8: ALU \to AC$（将加法结果送到累加器）



7.3 流水线技术

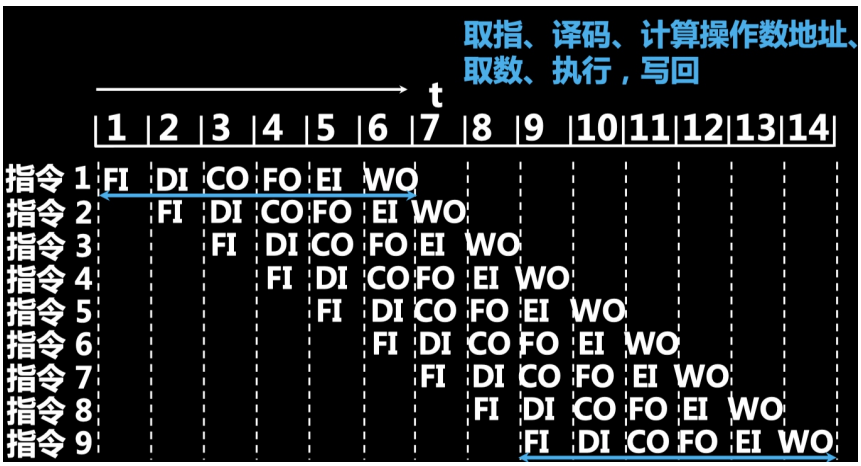
7.3.1 指令的二级流水

- 采用指令预取的方式，第一条指令还未执行结束，第二条指令就开始取指。
- 若取指和执行阶段时间完全重叠，则指令周期减半，速度提高 1 倍。

7.3.2 指令的六级流水

将指令的执行分为下述六个阶段。

1. 取指
2. 译码
3. 计算操作数地址
4. 取数
5. 执行
6. 写回



六级流水，则从 T_6 开始，每个时间单位完成一条指令，9 条指令只需 14 个时间单位。

7.3.3 流水线性能

> 吞吐量

单位时间内流水线所完成指令或输出结果的数量。

设 m 级流水线，各段时间为 Δt，则最大吞吐率为

$$T_{pmax} = \frac{1}{\Delta t},$$

而实际吞吐率为（连续处理 n 条指令的吞吐率）

$$T_p = \frac{n}{m * \Delta t + (n-1) * \Delta t},$$

可以发现，如果 n 趋向于无穷大

$$\lim_{n \to \infty} T_p = T_{pmax}$$

> 加速比 S_p

m 段流水线的速度与等功能的非流水线的速度之比。

设流水线各段时间为 Δt，则完成 n 条指令在 m 段流水线上共需

$$T = m * \Delta t + (n-1) * \Delta t,$$

而完成 n 条指令在等效的非流水线上共需

$$T' = n * (m * \Delta t),$$

则我们可以求得

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

当 $n \to \infty$ 时，我们可以得到

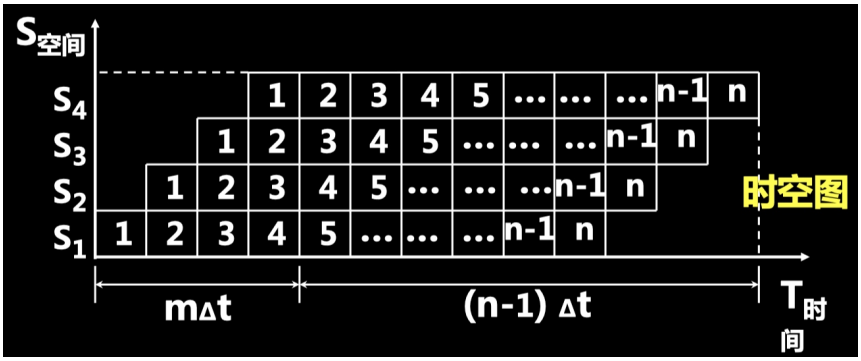
$$\lim_{n\to\infty} S_p = m$$

> 效率

- 效率，即流水线中各功能段的利用率。
- 由于流水线有建立时间和排空时间，因此各功能段的设备不可能一直处于工作状态。

$$\text{效率} = \frac{\text{流水线各段处于工作时间的时空区}}{\text{流水线中各段总的时空区}} = \frac{mn\Delta t}{m(m+n-1)\Delta t} = \frac{n}{m+n-1}$$

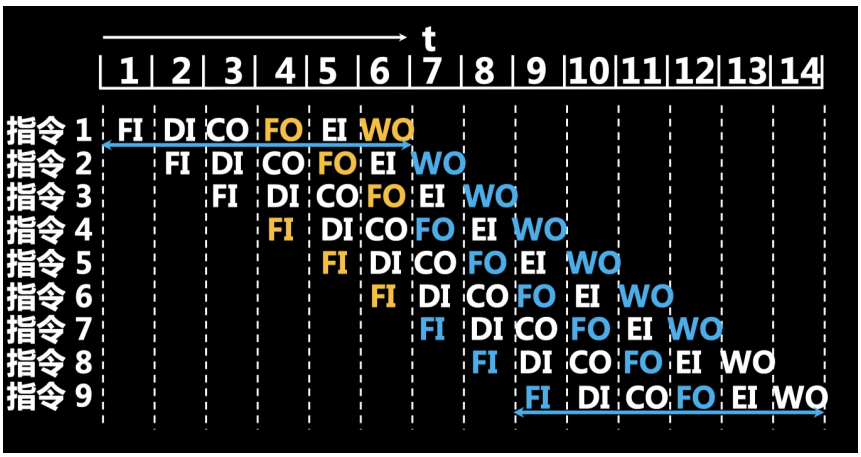
当 $n \to \infty$ 时，效率趋近于 1。



7.3.4 流水线的相关问题

> 结构相关

不同指令争用同一功能部件产生资源冲突。



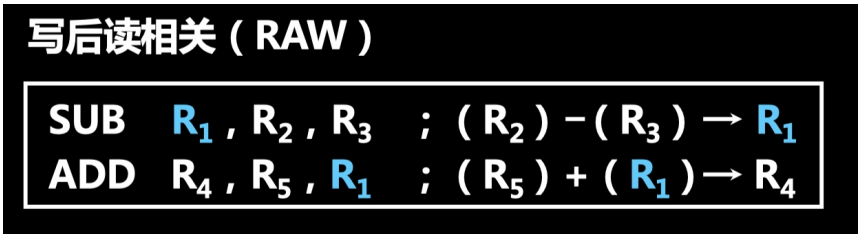
如在上述的六级流水线中，取指与取数发生了冲突。

解决办法:

1. 后续指令停顿
2. 指令存储器和数据存储器分开，多部件（哈佛结构）
3. 指令预取技术（仅适用于访存周期短的情况）
    - 指令执行过程中，只要取指部件空闲，就一直去内存中取指令，将指令放入CPU的指令预取队列中。
    - 将来一旦发现取指与取数冲突的情况，就以取数为有限，取指暂停。

> 数据相关

不同指令因重叠操作，可能改变操作数的读/写访问顺序，数据相关有三种，分别是 (1) 写后读相关，(2) 读后写相关，(3) 写后写相关。下面主要分析写后读相关冲突。



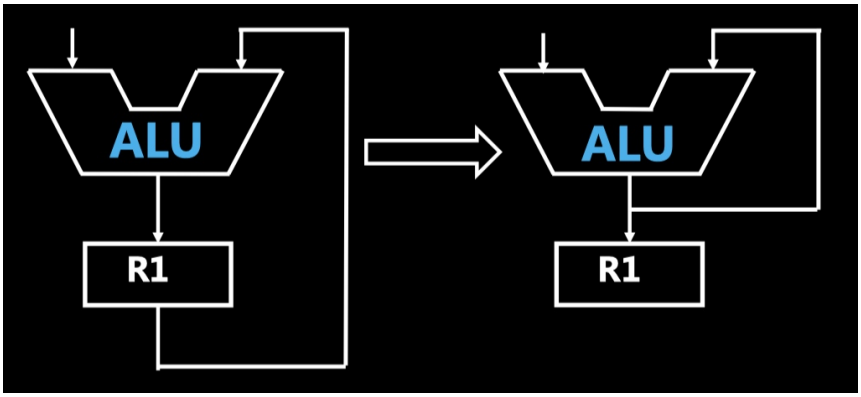
现有上述两条指令，其流水线执行过程如下。

可以看到在 R_1 还未写回时，就在取数过程中被取走了，因此发生了错误。

解决办法：旁路技术。

- 我们将 ALU 的输出端直接连向输入端，即可在第一条指令执行结束时，就将 R_1 写入

ALU 的输出端口，从而避免错误发生。



> 控制相关

控制相关问题由转移指令引起，如下述的执行指令。DIV 指令已经执行了几个周期了，但是 BNE 指令是跳转指令，因此 DIV 指令执行的多余周期全部作废，造成了流水线效率的浪费。

```
        LDA   # 0
        LDX   # 0
M : ADD   X, D
        INX
        CPX   # N
        BNE   M
        DIV   # N
        STA   ANS
```

解决办法：

- 为了解决这个问题，我们采用了分支预测法。
- 例如对于一个 100 次的循环，不管循环内容，只要知道对于这种 100 次的循环来说，前 99 次走到循环的末尾，都要跳回到循环的上部继续往下走。因此我们我们遇到跳转指令时，就直接从跳转位置开始取指令，降低出错概率。

7.3.5 提高机器速度的方法

> 一、提高访存速度（CPU 与主存）

高速芯片、Cache、多体并行

> 二、提高 I/O 和主机之间的传送速度

中断、DMA、通道、I/O处理机、多总线

> 三、提高运算器速度

高速芯片、改进算法、快速进位链

> 总结

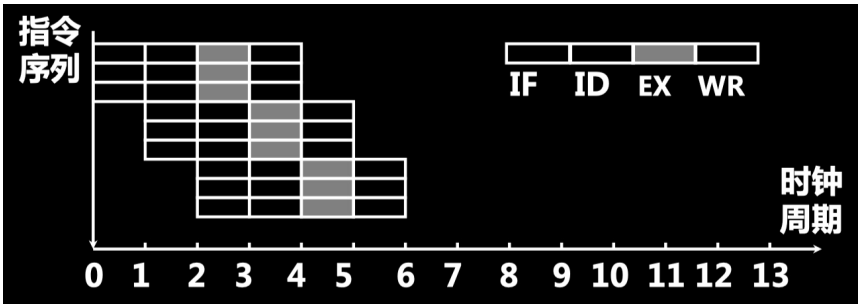
因此，提高整机处理能力，主要从两个角度入手。

1. 高速器件
2. 改进系统结构，挖掘系统的并行性

7.3.6 流水线的多发技术

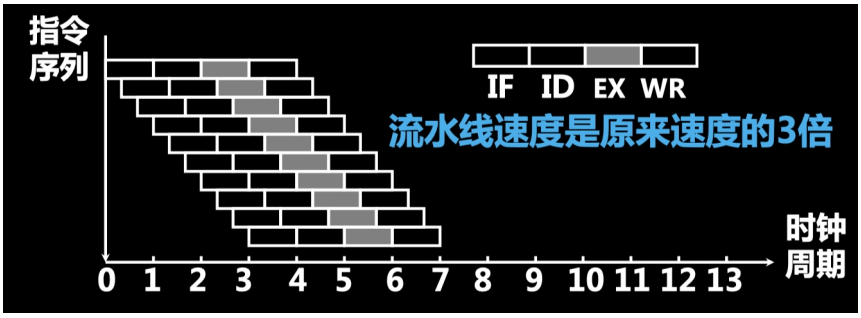
> 一、超标量技术

- 每个时钟周期内可并发发射多条独立指令，配置多个功能部件。
- 不能调整指令的执行顺序: 通过编译优化技术，把可并行执行的指令搭配起来。



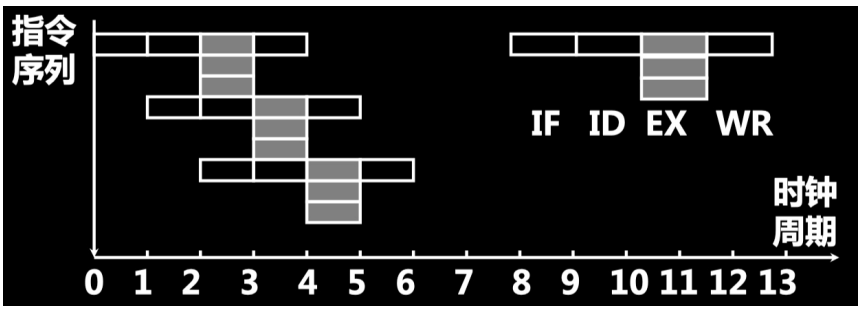
> 二、超流水线技术

- 在一个时钟周期内再分段（3段），在一个时钟周期内一个功能部件使用多次（3次）
- 不能调整指令的执行顺序: 靠编译程序解决优化问题



> 三、超长指令字技术

- 由编译程序挖掘出指令间潜在的并行性，将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字（可达几百位）
- 采用多个处理部件。



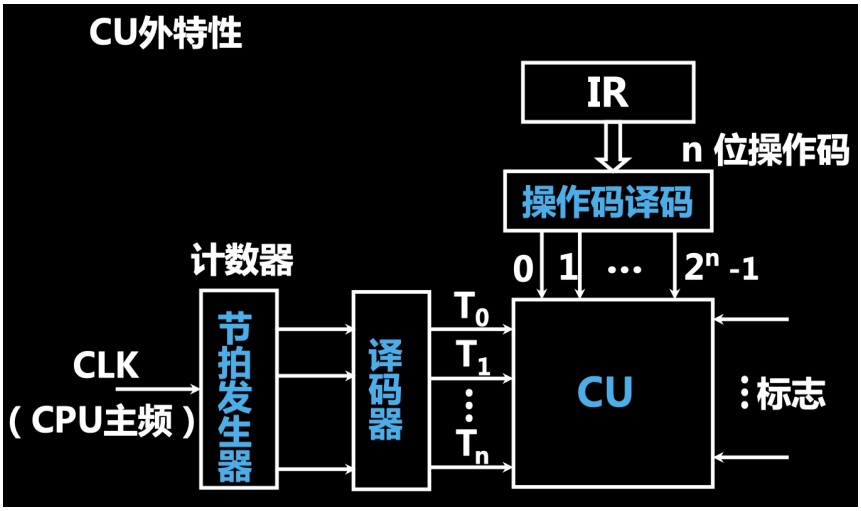
第八章 控制器设计

8.1 组合逻辑控制器

8.1.1 操作码译码

CU 执行的关键在于 IR 将指令操作码部分传入译码器中，译码器根据指令操作码部分输出对应的指令编号。

CU 再根据输入的指令编号，输出相应的微操作命令。



8.1.2 微操作设计概述

> 微操作设计的关键任务

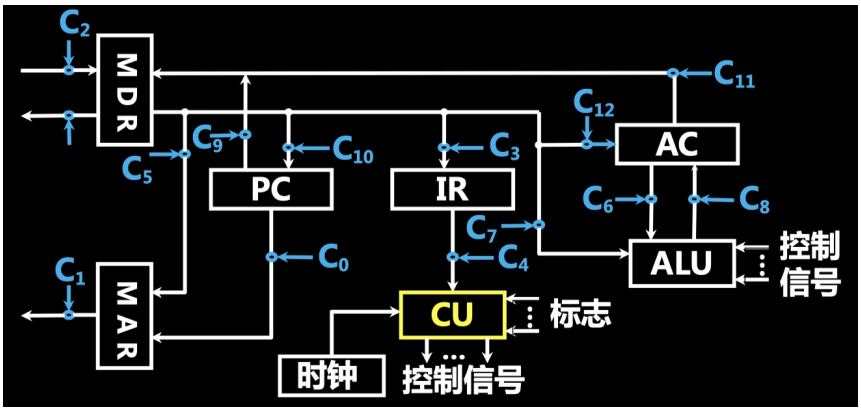
由于控制器实质是根据当前指令功能，依次发出一系列控制信号，因此我们需要了解如下信息。

1. 有哪些控制信号
2. 这些控制信号如何依次发出？即这些控制信号应该安排到哪个机器周期的哪个节拍中发

出。

> 微操作设计的前提假设

- 采用同步控制方式。
- 一个机器周期内有 3 个节拍。（时钟周期）
- CPU 内部结构采用非总线方式。



> 安排微操作时序的原则

1. 微操作的先后顺序一般不得随意更改。
2. 被控对象不同的微操作尽量安排在一个节拍内完成。
    - 由于被控对象不同，因此不同微操作可以并行执行，提高执行效率。
3. 占用时间较短的微操作，尽量安排在一个节拍内完成，并允许有先后顺序。
    - 两个微操作占用时间非常短，因此可以在同一个节拍内依次执行，加快硬件执行效率。

> 组合逻辑控制器设计流程

组合逻辑控制器就是一个大规模的组合逻辑电路，因此又称为硬布线控制器。

组合逻辑控制器设计一共分为以下 5 个流程。

1. 指令周期的微操作节拍安排。
2. 绘制操作时间表。
3. 设计微操作命令的最简逻辑表达式。
4. 设计微操作命令的逻辑电路图。
5. 综合优化。

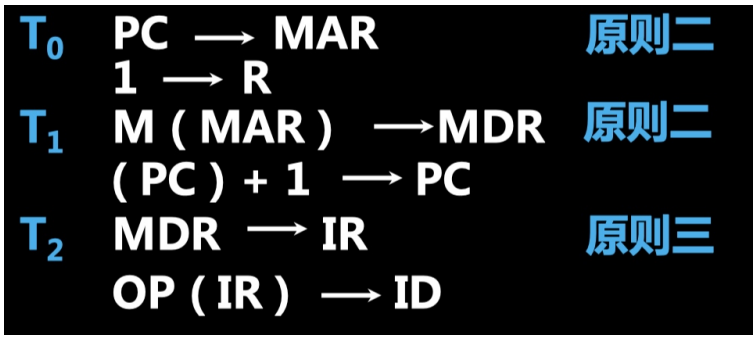
> 组合逻辑控制器设计的特点

1. 思路清晰，简单明了。
2. 电路庞杂，调试难，修改难，升级维护难。
3. 速度快。
4. 这种设计方式主要被 RISC 处理器采用，因此 RISC 处理器速度快，但一旦更新换代则很

难做到兼容。

8.1.3 (1) 指令周期微操作节拍安排

> 取指周期微操作的节拍安排



> 间址周期微操作的节拍安排

> 执行周期微操作的节拍安排

以下述十条指令的执行周期为例。

- CLA（清零）

T_0, T_1 为空，T_2 时安排 $0 \to AC$。

- COM（取反）

T_0, T_1 为空，T_2 时安排 $\overline{AC} \to AC$。

- SHR（ACC右移，符号位不变）

T_0, T_1 为空，T_2 时安排 $L(AC) \to R(AC), AC_0 \to AC_0$。

- CSL（循环左移）

T_0, T_1 为空，T_2 时安排 $R(AC) \to L(AC), AC_0 \to AC_n$。

- STP（停机）

T_0, T_1 为空，T_2 时安排 $0 \to G$。

- ADD X（ACC加上 X 对应数据）

T_0 Ad (IR) ⟶ MAR 1 ⟶ R
T_1 M (MAR) ⟶ MDR
T_2 (AC) + (MDR) ⟶ AC

- STA X（存数）

T_0 Ad (IR) ⟶ MAR 1 ⟶ W
T_1 AC ⟶ MDR
T_2 MDR ⟶ M (MAR)

- LDA X（读数）

T_0 Ad (IR) ⟶ MAR 1 ⟶ R
T_1 M (MAR) ⟶ MDR
T_2 MDR ⟶ AC

- JMP X（跳转）

T_0, T_1 为空，T_2 时安排 $Ad(IR) \to PC$。

- BAN X（条件跳转）

T_0, T_1 为空，T_2 时安排 $A_0 * Ad(IR) + \overline{A_0}(PC) \to PC$。

> 中断周期微操作的节拍安排

中断周期即三条中断隐指令的实现。

- T_0 : 执行微操作 $0 \to MAR, 1 \to W$ 以及硬件关中断
- T_1 : 执行微操作 $PC \to MDR$
- T_2 : 执行微操作 $MDR \to M(MAR)$, 向量地址 $\to PC$

8.1.4 (2) 绘制操作时间表

在这个步骤中需要根据第一步中设计的微操作来安排每个指令的具体执行过程。下图中空格子表示不执行。

> 取指周期操作时间表

工作周期标记	节拍	状态条件	微操作命令信号	CLA	COM	SHR	CSL	STP	ADD	STA	LDA	JMP	BAN
FE 取指	T_0		PC →MAR	1	1	1	1	1	1	1	1	1	1
			1 → R	1	1	1	1	1	1	1	1	1	1
	T_1		M(MAR)→ MDR	1	1	1	1	1	1	1	1	1	1
			(PC) +1 →PC	1	1	1	1	1	1	1	1	1	1
	T_2		MDR → IR	1	1	1	1	1	1	1	1	1	1
			OP(IR) → ID	1	1	1	1	1	1	1	1	1	1
		I	I →IND						1	1	1	1	1
		$\bar{I}$	I → EX	1	1	1	1	1	1	1	1	1	1

间址周期操作时间表

工作周期标记	节拍	状态条件	微操作命令信号	CLA	COM	SHR	CSL	STP	ADD	STA	LDA	JMP	BAN
IND 间址	T_0		Ad (IR) →MAR						1	1	1	1	1
			I → R						1	1	1	1	1
	T_1		M(MAR)→ MDR						1	1	1	1	1
	T_2		MDR →Ad (IR)						1	1	1	1	1
		$\overline{IND}$	I → EX						1	1	1	1	1

执行周期操作时间表

ADD、STA、LDA 三个指令的执行过程如下。

工作周期标记	节拍	状态条件	微操作命令信号	CLA	COM	SHR	CSL	STP	ADD	STA	LDA	JMP	BAN
EX 执行	T_0		Ad (IR) → MAR						1	1	1		
			1 → R						1		1		
			1 → W							1			
	T_1		M(MAR)→ MDR						1		1		
			AC → MDR							1			
	T_2		(AC)+(MDR)→AC						1				
			MDR → M(MAR)							1			
			MDR → AC								1		
			0 → AC	1									

其余七条指令的执行过程如下。

工作周期标记	节拍	状态条件	微操作命令信号	CLA	COM	SHR	CSL	STP	ADD	STA	LDA	JMP	BAN
EX 执行	T_2		$\overline{AC} \rightarrow AC$		1								
			L(AC) →R(AC),AC_0不变			1							
			$P^{-1}(AC)$				1						
			Ad(IR) → PC									1	
		A_0	Ad(IR) → PC										1
			0 → G					1					

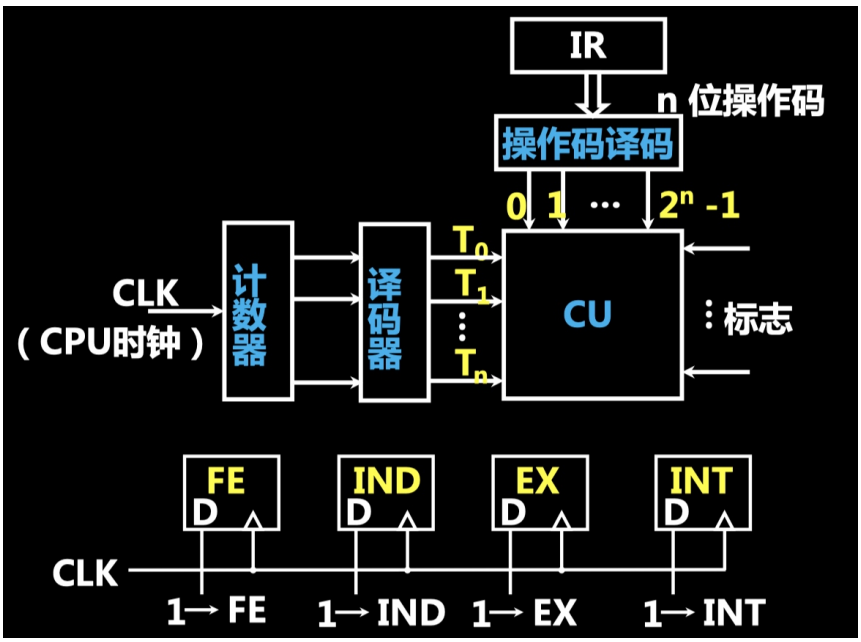
8.1.5 (3) 设计微操作命令的最简逻辑表达式

我们以 $M(MAR) \rightarrow MDR$ 这条命令为例。

$$
\begin{aligned}
&M(MAR) \rightarrow MDR \\
&= FE * T_1 + IND * T_1(ADD + STA + LDA + JMP + BAN) \\
&\qquad + EX * T_1(ADD + LDA) \\
&= T_1\{FE + IND(ADD + STA + LDA + JMP + BAN) + EX(ADD + LDA)\}
\end{aligned}
$$

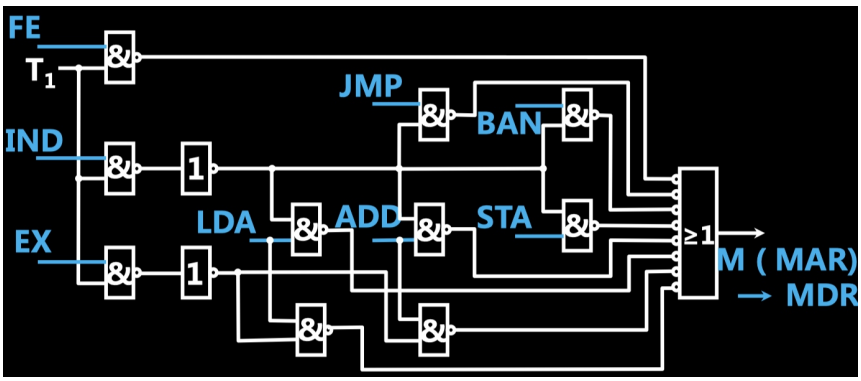
上述的逻辑表达式恰好是根据第二部分中各指令执行的操作时间表来决定的。

我们可以发现其中时钟信号 T_1，机器周期 FE, IND, EX 以及十种操作类型均在下述的硬件电路中已经给出。



8.1.6 (4) 设计微操作命令的逻辑图

我们根据上述的最简逻辑表达式设计出下述的逻辑电路图。



8.1.7 (5) 综合优化

上述的过程只涉及到了一条微操作的设计，在实际工作中，我们需要将全部微操作命令的逻辑图综合起来，从整体上进行优化工作。

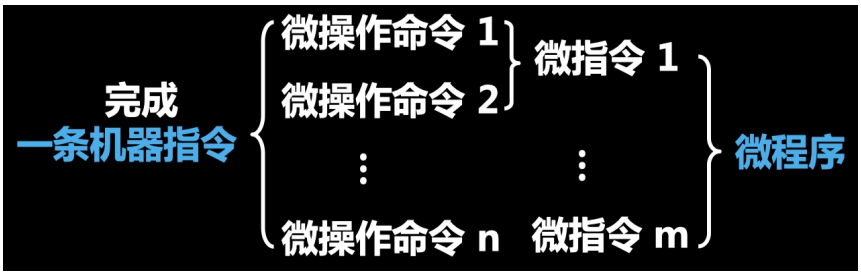
8.2 微程序控制器工作原理

8.2.1 微程序设计概述

> 微程序设计思想

以 "微程序-微指令-微操作命令" 三级对一条机器指令进行划分。

一条机器指令对应一个微程序，将微程序存入 ROM 中，称为控制存储器。



> 微程序与硬布线设计方式的联系

- 微程序以微指令为单位，硬布线以时钟周期为单位。其中微指令与时钟周期一样，都会发出一个或几个微操作命令。
- 硬布线方式的三级结构: "指令周期-机器周期-时钟周期-微操作 (控制信号)"
- 微程序方式的三级结构: "大段微程序-小段微程序-微指令-微操作 (控制信号)"

8.2.2 微程序工作原理

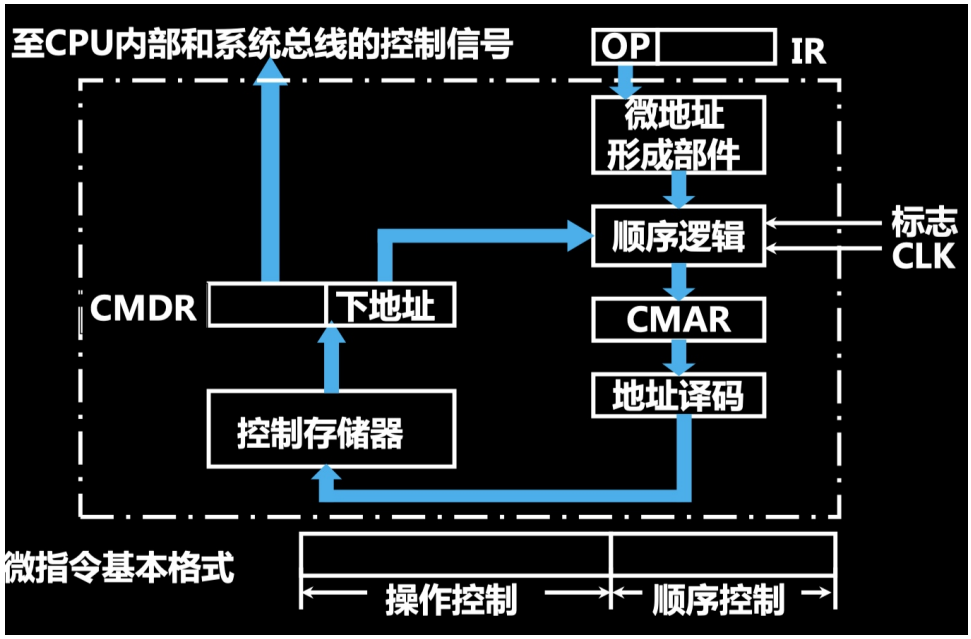
> 控制存储器的组成

由于指令的取指、间址、中断周期均一致，因此在控制存储器中的固定位置统一存放这三个过程的微程序。并在其余部分依次存放各类操作执行周期对应的微程序。

> 微程序控制单元的基本框图

CMDR: 控存数据寄存器，保存当前微指令。
CMAR: 控存地址寄存器，保存微指令的地址。

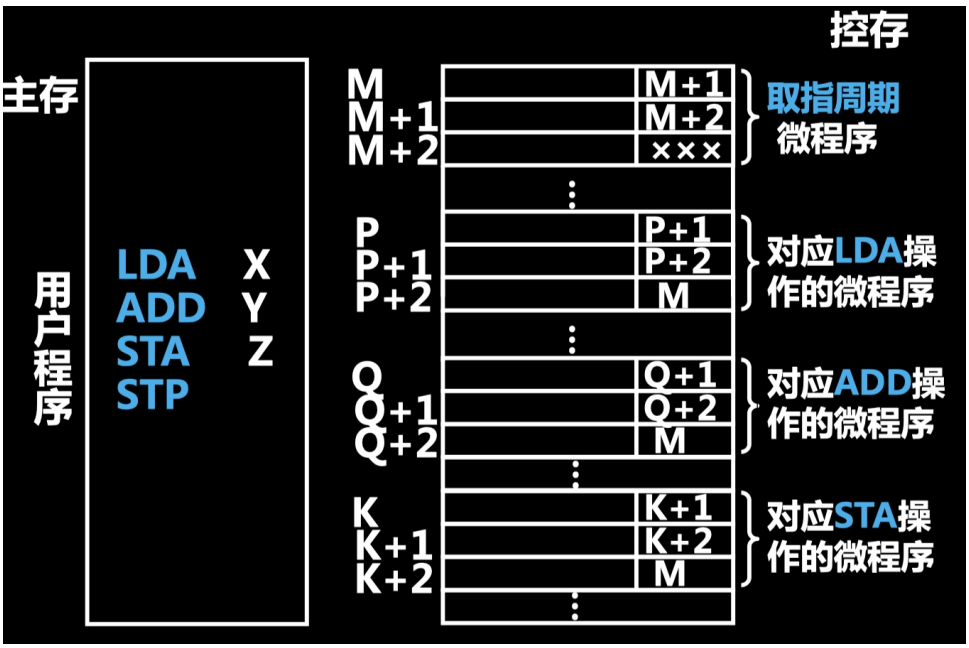
微指令分成两个部分，"操作控制字段" 与 "顺序控制字段" (保存下一条微指令地址，因此也称为 "下址字段")。



> 微程序的存储结构

根据控制存储器中的内容，我们可以看到取指周期的最后一条微指令的下址字段不确定，原因是不知道接下来执行哪一段微程序。

并且所有执行周期微程序最后一条微指令的下址字段均为 M，因为执行完后开始下一条指令的取址。

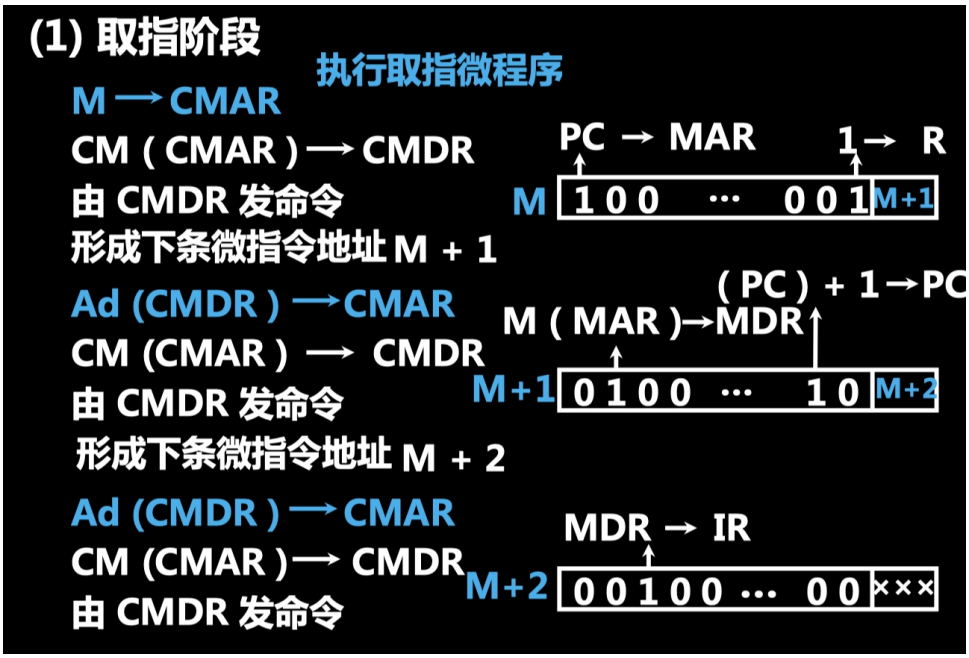


8.2.3 微程序工作流程

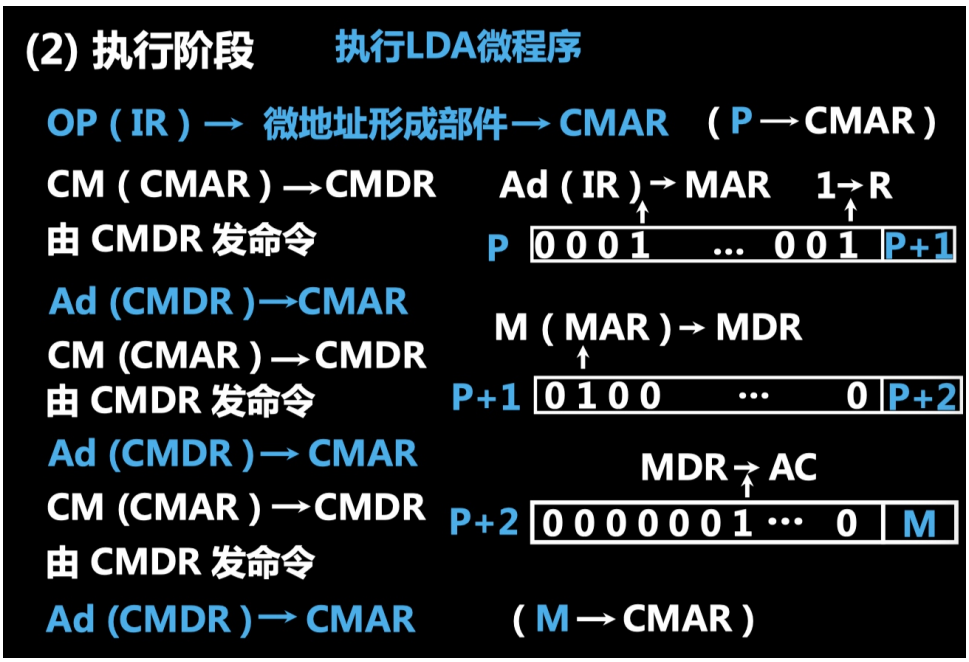
> 取指阶段

取指阶段入口地址 M 送给 CMAR

微指令的控制字段是一个0、1序列，其中 1 代表一个控制信号的开启。



> 执行阶段



8.2.3 微程序设计特点总结

> 特点

- 指令执行的每个阶段都是一段微程序，共同组成完成指令的微程序。
- 全部微指令存在控制存储器（CM）中，程序执行过程中只需读出。因此一旦指令集更新换代，也只需要更新 CM 即可，比硬布线方式方便很多。
  - 因此微程序设计方式为 CISC 处理器所采用。

> 实现关键点

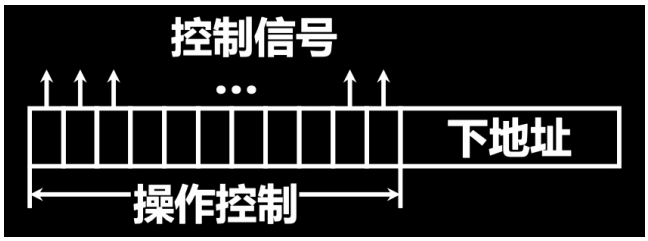
- 微指令的操作控制字段如何形成微操作命令。
- 微指令的后续地址如何形成。

8.3 微程序控制器设计方法

8.3.1 微指令的编码方式

> 直接控制方式

在微指令的操作控制字段中，每一位代表一个微操作命令，这种方式速度最快。

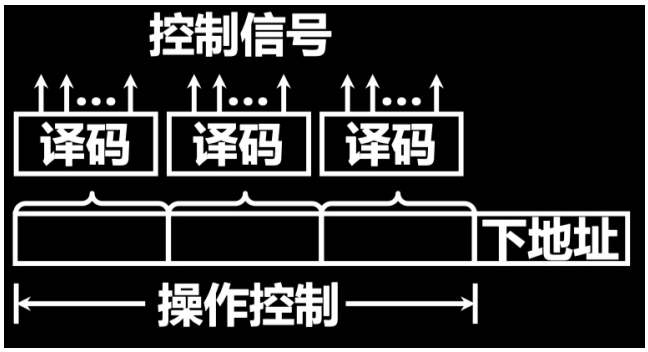


在操作控制码字段，某位为 “1” 即表示该控制信号有效。

> 字段直接编码方式

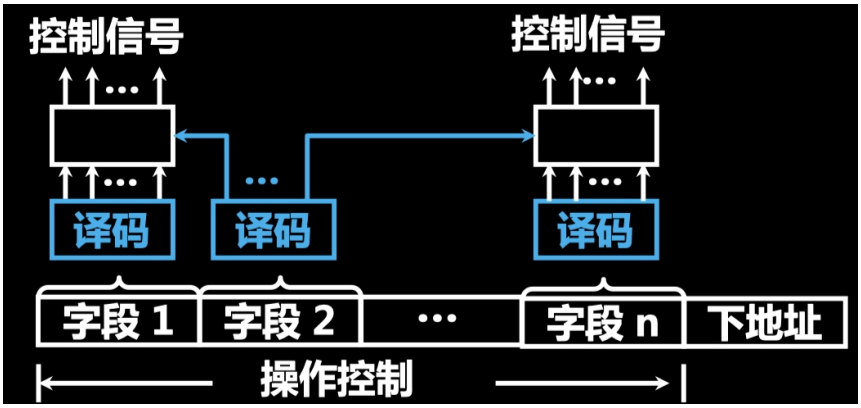
将互斥的微操作经编码合在一起作为一个 “段”，每段经译码后发出控制信号。

例如 8 个互斥微操作，我们可以用 4 位进行二进制编码，最后再译码得到对应控制信号。（4 位的原因是还有一个无操作信号，一共 9 个信号）



这种方式缩短了微指令字长，增加了译码时间，执行较慢。

> 字段间接编码方式



> 混合编码

直接编码和字段编码（直接和间接）混合使用。

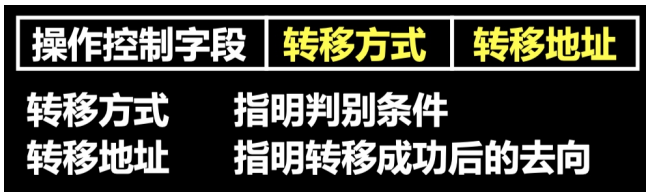
8.3.2 微指令序列地址的形成

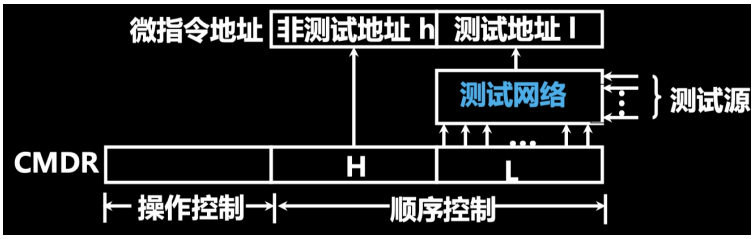
> 6 种形成方式

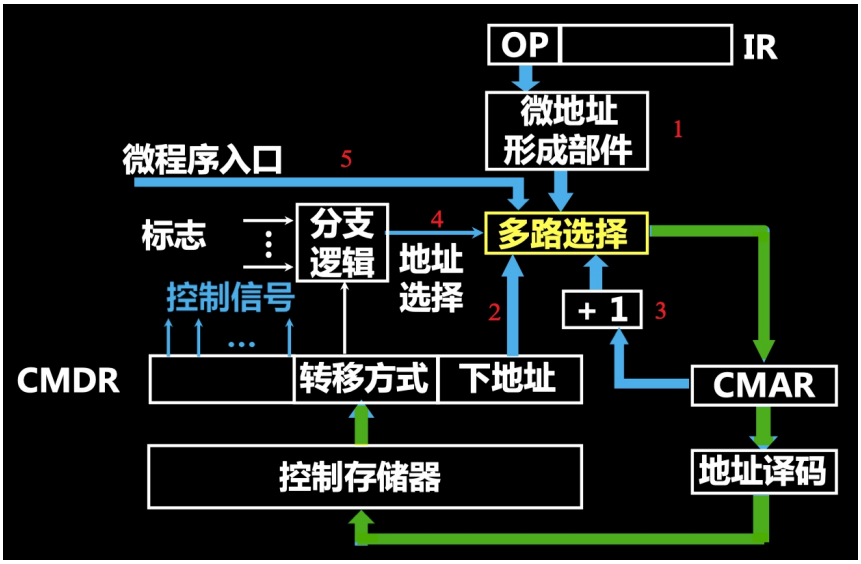
1. 微指令的下地址字段指出
2. 根据机器指令的操作码形成（IR）
3. 增量计数器

$$(CMAR) + 1 \rightarrow CMAR$$

4. 分支转移
    - 根据转移方式来判断转移去向。

- 

5. 由硬件产生微程序入口地址
   - 第一条微指令地址（由专门硬件产生）
   - 中断周期（由硬件产生中断周期微程序首地址）
6. 通过测试网络
   - 

> 具体形成过程

微程序序列地址由下述的 5 种方式依次产生，并通过一个多路选择器来确定最终的序列地址，然后将地址传入 CMAR 中，经过译码后将控制存储器中对应的微程序存入CMDR中，然后发出对应的控制信号。