

Copyright@潘禧辰

contact: [panxichen.pxc@gmail.com](mailto:panxichen.pxc@gmail.com)

全部资料来源朱弘恣老师[hongzi@cs.sjtu.edu.cn](mailto:hongzi@cs.sjtu.edu.cn)

---

## Ch 01: introduction to microcomputer & embedded systems

冯诺依曼结构

组成:

三个重要概念:

流程:

微处理器

微型计算机 microcomputer

硬件

CPU

ALU

CU

Instruction Set 指令集

Memory 存储器

Bus

数据总线

地址总线

控制总线

寻址方式 (判断地址总线从CPU发出的地址是传给memory还是IO)

微控制器 microcontrollers MCS

嵌入式系统 embedded systems

## Ch 02: More on Memory and IO Modules

存储的特征

位置

容量

转移单位

访问方法

性能

RAM

ROM

存储器层级结构

层级结构列表

Chip Organisation 如何物理上把bits安排成words(in chip)

Chip Packaging

位拓展

字扩展

更多例子

I/O 问题

I/O 模块

外设

I/O 步骤

I/O module 结构

I/O 技术

programmed I/O 程序控制I/O

Interrupt driven I/O, 中断驱动I/O

CPU如何得知是哪个模块请求的中断

如何定位到对应的中断处理程序

如何处理多中断请求

Direct Memory Access (DMA), 直接存储器访问

## Ch 03 8086 微处理器

BIU

EU

寄存器

8086中的流水

8086/8088的接脚

最大最小工作模式

data transceivers 数据收发器

Latch 锁存器

接脚详细解释

Memory/IO Control Signals

Address/Data 去复用 & Address latching

Address latching设计

transceiver的设计 两个与门控制

8086/88 Bus Cycle (for data transfers)

8086 Programming

物理地址和逻辑地址

物理地址

逻辑地址

逻辑地址映射到物理

段值范围

Wrap-around

物理地址映射到逻辑地址

重叠

几个segment

cs code segment

ds data segment

ss stack segment

push&pop

es extra segment

IBM PC memory map

BIOS 功能

状态寄存器 PSW

有符号数 CF&OF

8086寻址方式

寄存器寻址方式

immediate直接寻址方式

直接寻址方式

寄存器间接寻址方式

base relative addressing mode

Indexed Relative Addressing Mode

Based Indexed Relative Addressing Mode

segment overrides覆写

## Ch 04 汇编语言

编程语言

汇编语言程序

声明形式

真实程序的shell

model 定义

Simplified Segment定义

segment

Full Segment定义

程序执行

控制转移说明

有条件跳转

无条件跳转

子程序和CALL语句

数据类型和定义

有关变量的更多信息

有关标签的更多信息  
有关PTR指令的更多信息  
.COM可执行文件

## Ch 05 汇编语言

运算指令  
无符号加法  
    单字节加法示例  
    多字节加法示例  
无符号减法  
    单字节减法示例  
    多字节数字减法示例  
无符号乘法  
**无符号乘法示例**  
无符号除法  
    无符号除法示例  
逻辑指令  
AND  
OR  
XOR  
NOT  
Logical SHIFT  
    示例：BCD和ASCII数字转换  
    ASCII->未压缩的BCD转换  
    ASCII->压缩BCD转换  
ROTATE  
    ROTATE with carry  
比较无符号数字  
比较有符号数  
QUIZ

## Ch06 Lecture 06: Memory Address Decoding

回顾存储芯片  
要学什么？  
**内存地址解码**  
QUIZ  
The 74LS138 Decoder Chip  
Using 74LS138 to Decode  
QUIZ  
有关地址解码的更多信息  
**示例1.**  
**示例2.**  
数据的完整性  
Checksum byte  
奇偶校验位  
8086的memory organization  
字节存储操作  
对齐的word-memory操作  
错位的word-memory操作  
例子  
X86系列中的I / O -内存中的其他空间  
I / O指令- 8 Bit实例  
I / O示例  
I / O指令- 16位实例  
将8位I / O模块连接到16位数据总线  
输出端口设计  
输入端口设计

## Ch 07 8255 PPI Chip

内部结构和接脚  
操作模式

## 控制寄存器和操作模式

输入/输出模式

    输入/输出模式示例

BSR模式

    BSR模式示例

mode 0 (简单I / O)

    例子

模式1 (频闪I / O)

模式1：作为输入端口

    模式1输入中的时间线

模式1：作为输出端口

    模式1输出中的时间线

模式2 (双向总线)

    模式2：作为输入和输出端口

    轮询与中断

用8255编程

    地址解码

    CODE

## Ch 08 8253/4 Timer

包装及内部结构

系统接口

内部结构

    Counter0

特征

内部结构和引脚

写/读操作

例子

8253的特点

模式0：终端计数中断 N减到0后中断

模式1：硬件可触发单稳态 (one shot)

模式2：速率生成器

模式3：方波速率发生器

模式4：软件触发的选通 Stobe

模式5：硬件触发的选通 (可触发)

编程范例

## Ch 09 中断和8259

8086/8088中的中断

主程序和ISR

中断类别

    硬件中断

        处理可屏蔽中断的过程

    软件中断

INT和CALL之间的区别

软件中断

    处理不可屏蔽和软件中断的过程

8086/8088的中断向量表

中断优先级

INTR中断的优先级

8259

中断嵌套

## Ch 10 串行数据通讯和8251

数据通讯

两种方式：并行和串行

串行通讯的全貌

串行通讯

    资料传输率

    同步方法

        异步传输

异步传输

接收方提高频率

例子

同步传输

通讯方式

错误检测

调制与解调

8251 USART芯片

8251硬件

8251通讯接口

与8088接口

8251信号

8251模式字

8251命令字

8251状态字

8251A的操作

8251A上电内部复位

8251编程实例

8251编程实例

## Ch 11 Cortex-M3 / M4嵌入式系统: Cortex-M3 / M4处理器基础知识

嵌入式系统是什么?

嵌入式系统中的处理器

有关ARM处理器的历史

Cortex-M3 / M4处理器概述

Cortex-M3 / M4处理器与MCU

ARM处理器: 指令状态开关

Cortex-M3 / M4指令集

Cortex-M3 / M4基础知识: 寄存器

# Ch 01: introduction to microcomputer & embedded systems

---

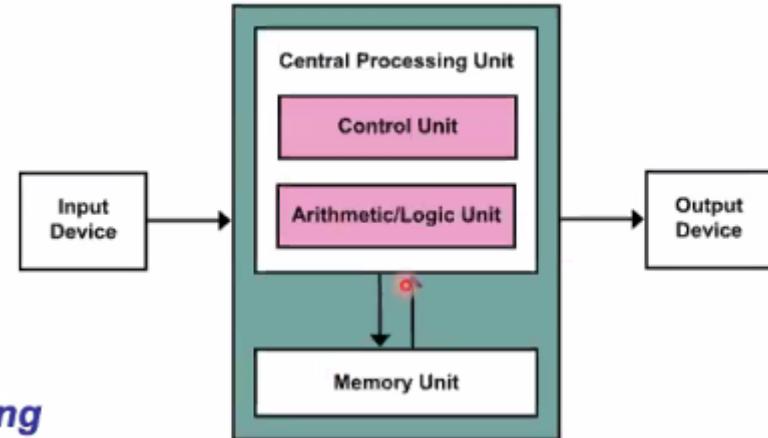
微型计算机和嵌入式系统

## 冯诺依曼结构

---

组成:

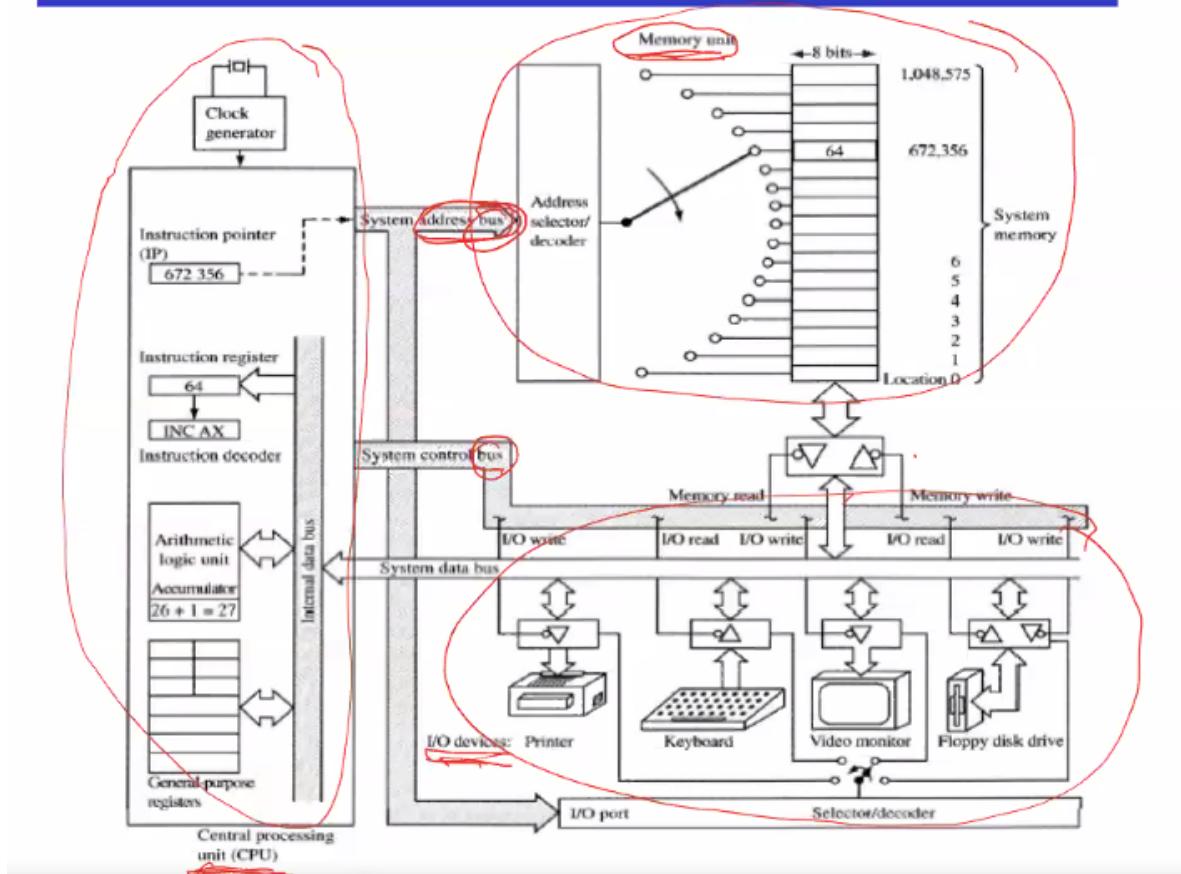
- I/O
  - 输入
  - 输出
- 存储
- CPU
  - ALU 算术逻辑单元
  - Control unit 控制单元



### 三个重要概念：

- 指令和数据都存储在一个单一可读可写的存储器中
- 存储器的内容是可寻址的(addressable by location), 无论数据的类型
- 程序的执行是串行(sequential)的方式

## Stored Program Concept



- control unit
  - instruction pointer指向下一个执行指令
  - instruction register
  - instruction decoder
- ALU
- register 输入输出暂时存储

### 流程：

- instruction pointer-地址总线->memory unit
- CPU同时在控制总线读信号读取memory unit, 进入数据总线进入CPU, instruction decoder进行解码, ALU计算后存储或输出。

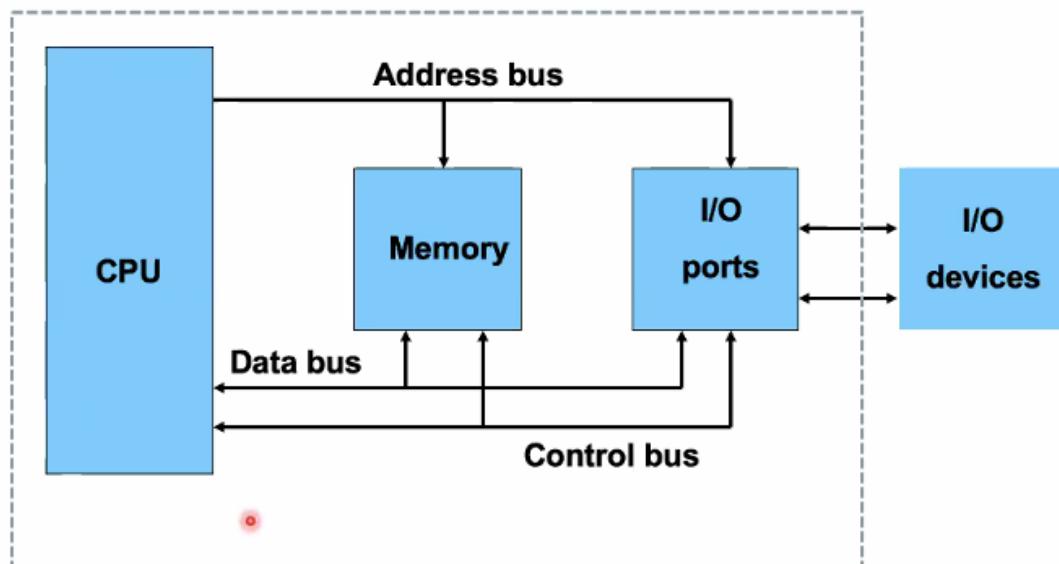
## 微处理器

- microprocessor 集成电路包括ALU/CU/register
- 无 RAM/ROM/IO
- e.g., Intel's x86 family (8088, 8086, 80386, 80386, 80486, Pentium); Motorola's 680x0 family (68000, 68010, 68020, etc)

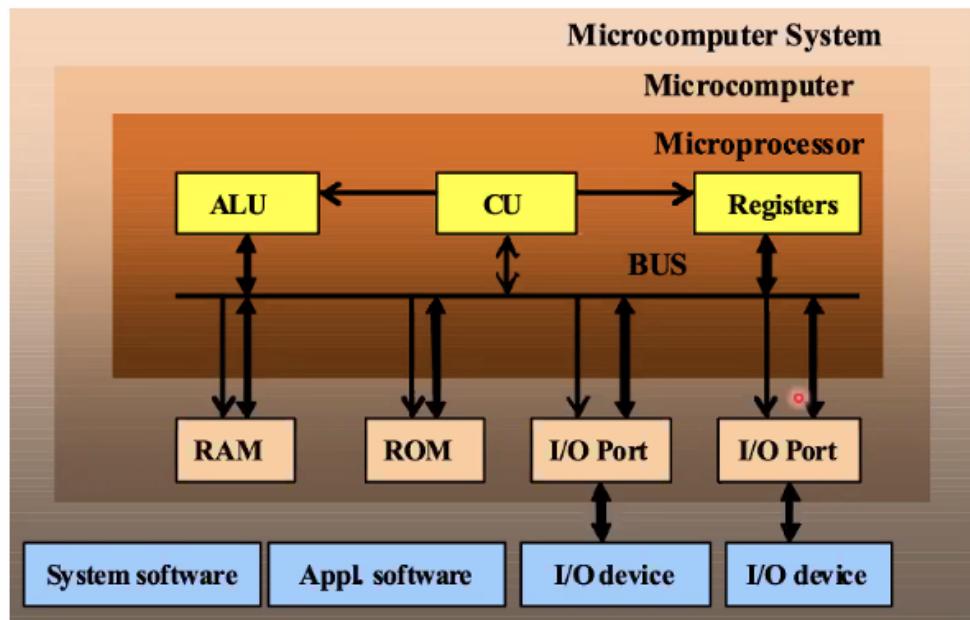
## 微型计算机 microcomputer

- CPU-microprocessor
- memory-ROM/RAM
- I/O接口 (如树莓派, 无设备有接口)
- BUS 总线
  - 地址总线address bus
  - 数据总线data bus
  - 控制总线control bus

# Microcomputer Structure



# Microcomputer System Structure



## Microcomputer System

- Microcomputer
- Peripheral I/O devices
- Software
  - System software
    - e.g., OS, compilers, drivers
  - Application software
    - e.g. Word, MatLab, Media player, Latex...



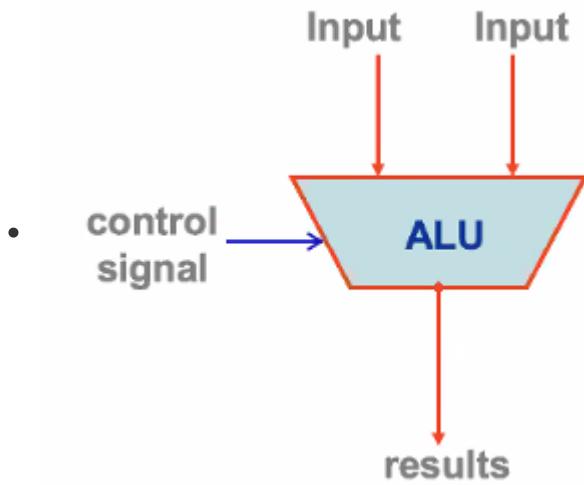
IO和软件属于微机系统而不属于微型计算机

## 硬件

### CPU

#### ALU

- 算术运算
- 逻辑运算



- ALU 是多功能运算器 受到控制信号控制
- 有两个输入
- 临时结果存储到寄存器中

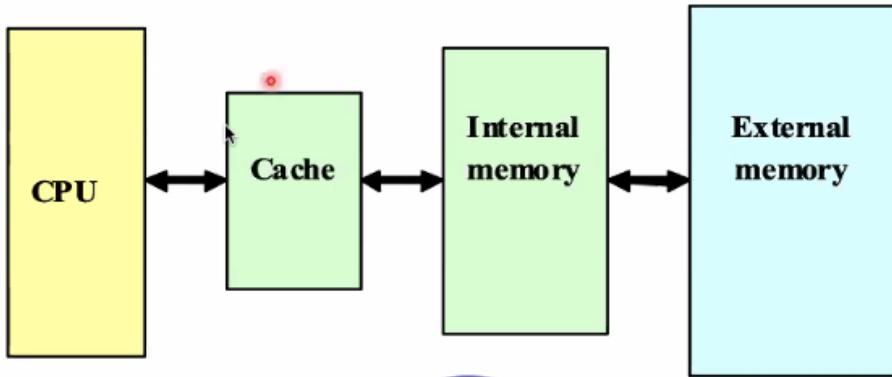
## CU

- 在指令控制下工作
- instruction decoder 指令解码器
  - 解码指令产生控制信号调节计算机各活动
- program counter 程序计数器
  - 指向下一个要执行的指令地址

## Instruction Set 指令集

- 能够被指令解码器解码的指令
- CISC 复杂指令集
  - 指令长度可变 (1-n) words
  - 执行的时间不同
  - 有很多指令格式
  - 向下兼容
  - 80\*86有3000个指令
- RISC 缩减指令集计算机
  - 指令长度为1word
  - 指令执行时长相同
  - 很容易用流水线(pipeline)技术
  - 更少的格式-简单的硬件设计，简单的芯片设计
  - e.g., PowerPC, MIPS, ARM, PIC's MCU

## Memory 存储器



- **Memory hierarchy**

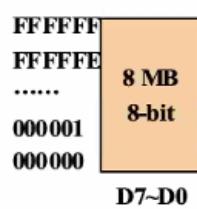


- Registers
- Cache
- Primary memory: ROM, RAM
- Secondary memory: magnetic disk, optical memory, tape, ...
- Bit (b) 一个二进制数字 0或1
- Byte (B) 字节 八个连续bits构成单元 最小addressable unit
- Nibble 4个bits
- Word 字, CPU一次能够处理的最大bits数
  - 和CPU中寄存器宽度、数据总线宽度有关
  - 数据总线宽度为32位, Word字长为32位
  - eg. - 整个系统来看, 受限于数据总线32位; CPU内部位数是寄存器位数64位
- Double word 双字 两个字长
- Kilo $2^{10}$ , mega $2^{20}$ , giga $2^{30}$ , tera $2^{40}$ , peta $2^{50}$

## Memory Module Organization

To organize a memory module with given memory chips

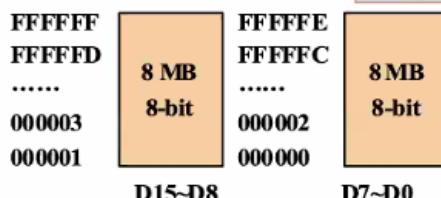
- **8-bit**



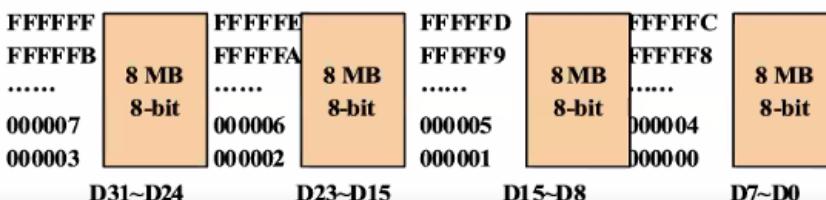
To organize a memory module:

- If the module needs larger unit of transfer than that of given memory chips, bit extension
- If the module needs large number of words than that of given memory chips, word extension

- **16-bit**



- **32-bit**



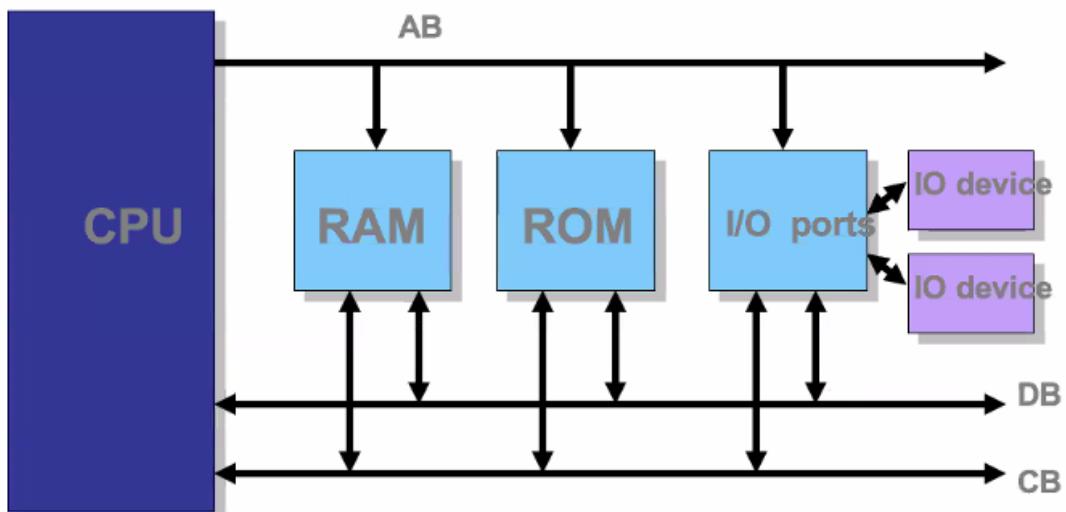
- bit extension

- 000000/000000
- 000001/000001
- 000002/000002
- .....
- word extension
  - 000000/000001/000002/000003
  - 000004/000005/000006/000007
  - .....

## Bus

- 信道是一个通信通路，连接多个设备
  - 共享：每次一个设备
  - 系统总线：连接计算机各个组分
  - arbitration 仲裁
    - 分布式协议
    - centralized scheme: master/slave
- 类型
  - dedicated 专线
  - multiplexed 可复用
- arbitration
  - centralized: 总线控制器
  - distributed: 协作
- timing
  - 同步
  - 异步 频率相似但不同

## Single-bus Structure



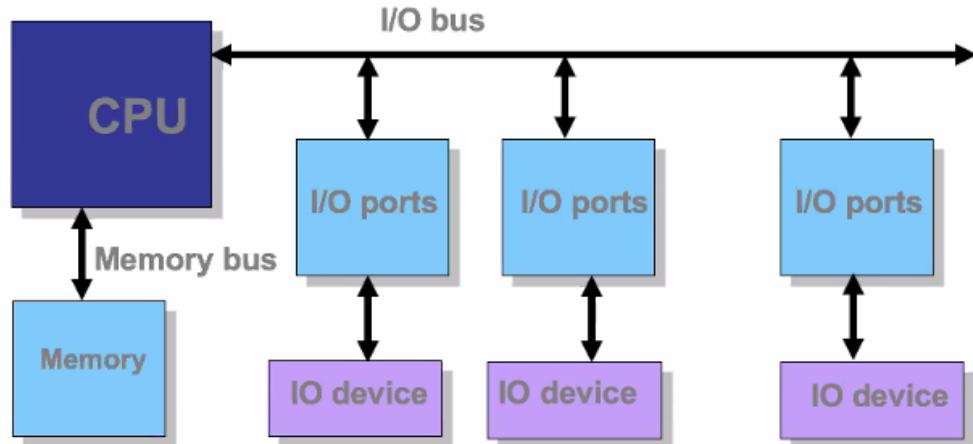
**A bus connects all modules**

- **pro:** simple
- **con:** poor performance in terms of throughput

**Why**

eg. CPU 无法同时读取RAM和IO

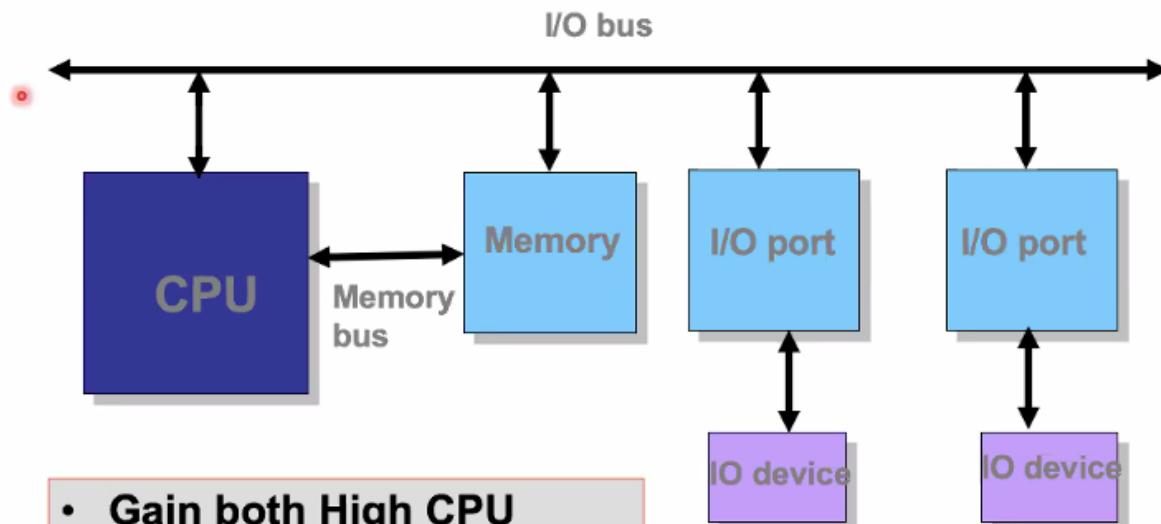
# CPU-Central Dual-Bus Structure



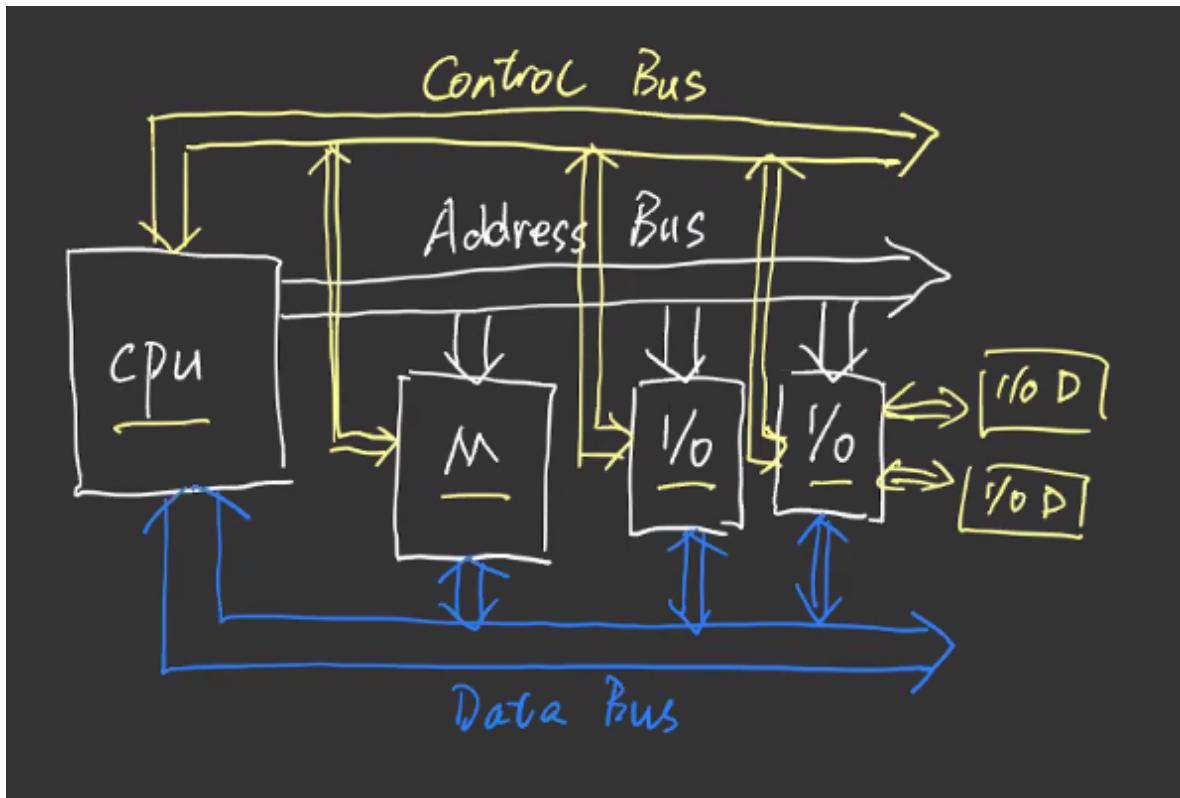
- A dedicated bus between CPU and memory, and a dedicated bus between CPU and I/O devices
- **pro:** efficient in terms of data transfer
- **con:** information between memory and I/O devices has to go through CPU. Therefore, poor CPU performance

双总线，memory和I/O通信经过CPU，影响CPU性能

# Memory-Central Dual-Bus Structure



- **Gain both High CPU performance and data transfer throughput**



### 数据总线

- 用于搬运数据
- read memory io->CPU
- write CPU->memory io
- 宽度和CPU寄存器相同，等于字长

### 地址总线

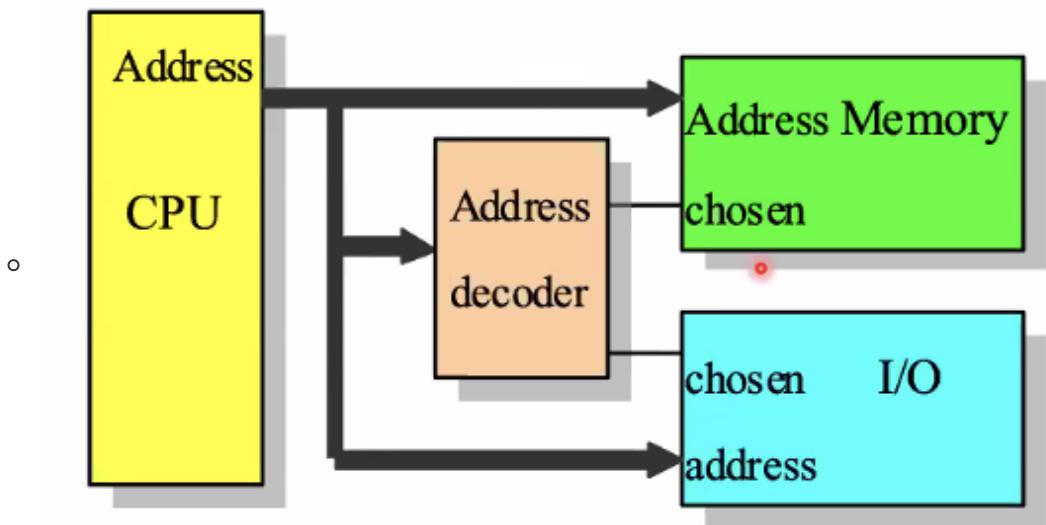
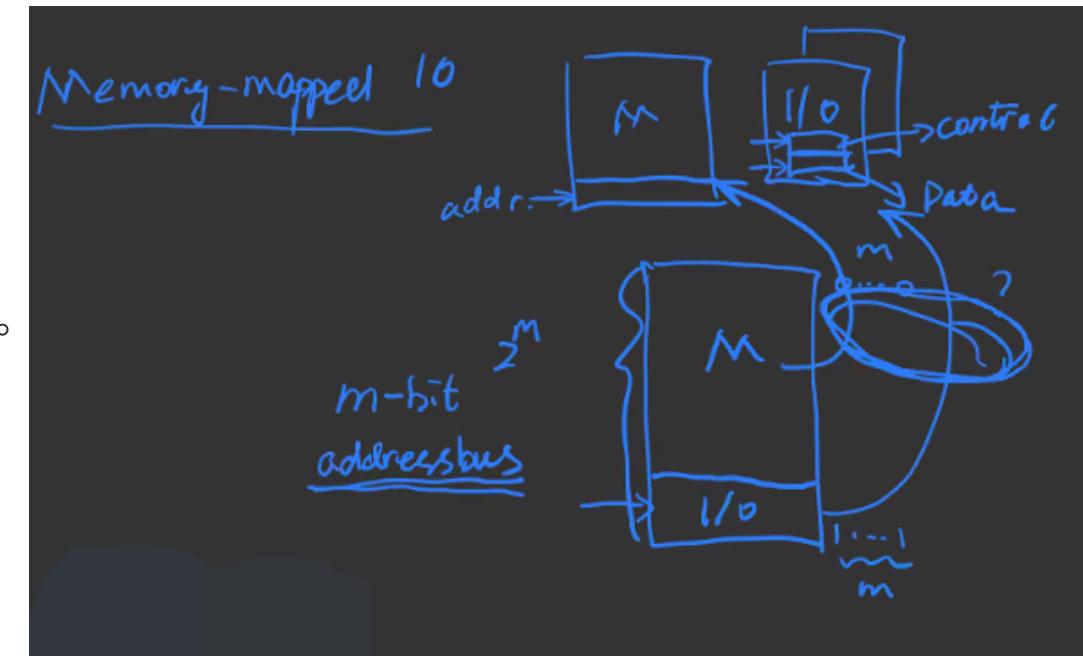
- 用来指定数据的源和目的地
- 单向 CPU->memory io
- 宽度 n
  - 决定寻址范围  $2^n$
  - 8086有20位地址总线,  $2^{20}$
  - pentium, 32位,  $2^{32}$

### 控制总线

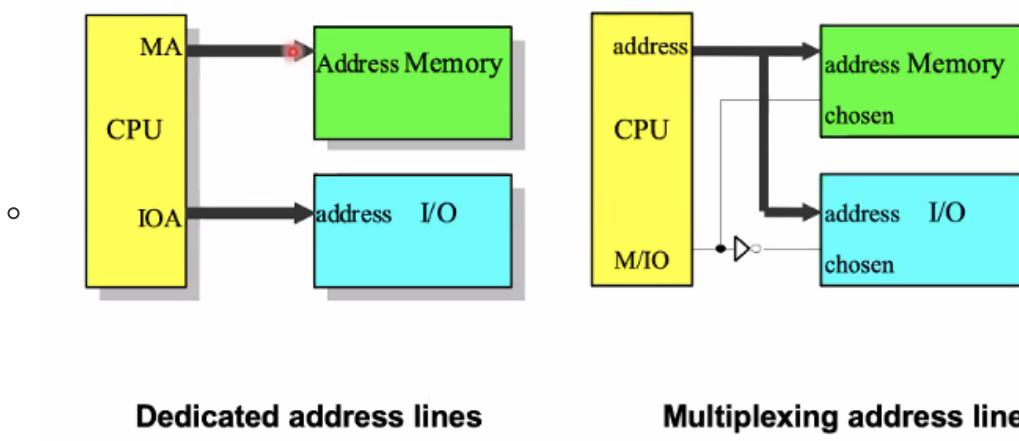
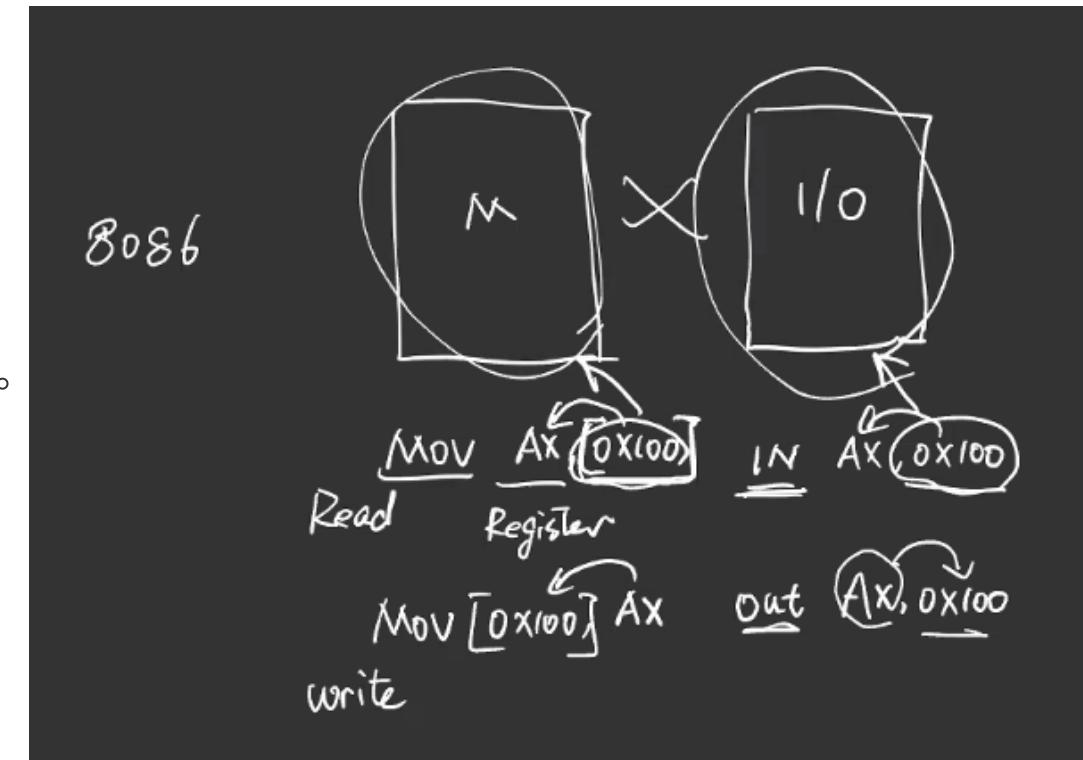
- 用于传输控制信号
  - command和timing信号, 一般CPU发出
  - 内存读写, I/O读写, 总线请求
- 两种控制信号构成:
  - command signal: CPU->Memory/IO
  - State signal: Memory/IO->CPU
- 输入输出是从处理器角度来说的

### 寻址方式 (判断地址总线从CPU发出的地址是传给memory还是IO)

- 存储器映像寻址方式
  - 将IO映射为memory, 认为IO是某一种memory
  - 相同的status/data register/instruction
  - 例如ARM



- IO单独编址方式
    - 两个不同地址空间
    - 使用不同的instruction
    - 例如8086



- multiplexing address lines 中 CPU 发出控制信号 M/IO，而存储器映像寻址方式不会主动发出

### 微控制器 microcontrollers MCS

- 单片上有固定的CPU RAM ROW IOports

### 嵌入式系统 embedded systems

- 用一个微处理器或微控制器来执行一项任务，且仅执行一项任务

## Ch 02: More on Memory and IO Modules

### 存储的特征

#### 位置

- CPU 内部 寄存器
- 内存
  - 缓存

- 内存
  - 外部存储
    - 硬盘
    - 磁带
    - DVD

# 容量

- Word size 存储器的word
  - words数量

- | Word size 存儲器的 Word
  - | The natural unit of organization = Capacity
  - | Number of words
  - | E.g., 2M 8-bit, 16M 1-bit
  - | Same capacity but different organization

$$\frac{2M \text{ address bits}}{8\text{-bit}} \times \frac{2M \text{ words}}{1\text{-bit}} = \frac{16M \text{ bytes}}{16M \text{ bits}}$$

- 8bit/1bit指字长
  - 2M/16M指字数

## 转移单位

- 内部存储
    - 通常是字长，受到数据总线宽度的限制
    - 存储字长<系统字长 (register/数据总线)
  - 外部存储
    - 通常是block (eg512bits)，大于word
  - 最小可寻址单位
    - 通常是一个Byte
    - 外部存储器通常是一个簇cluster

# 访问方法

- 串行访问
    - 例如磁带
    - 时长取决于数据的位置以及当前磁头的位置
  - 直接访问
    - 例如硬盘
    - 跳转至附近后进行串行访问
  - 随机访问

- 通过给定地址，唯一标定一个location
- 无机械装置，与上次读取无关
- 关联访问
  - 访问cache中的内容，cache通过算法预先读取有关内容
  - 无机械装置，与上次读取无关。与数据内容有关

## 性能

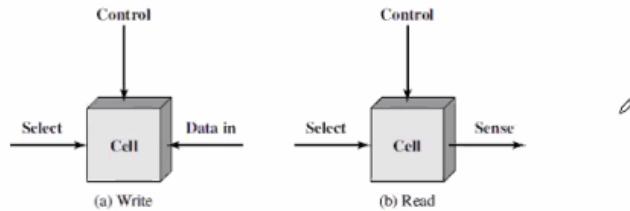
- 存取时间access time
- 循环时长cycle time=access+recovery
- 传输速率Transfer Rate
- 物理类型
  - RAM-半导体材料
  - 磁
  - 光
  - 电

## RAM

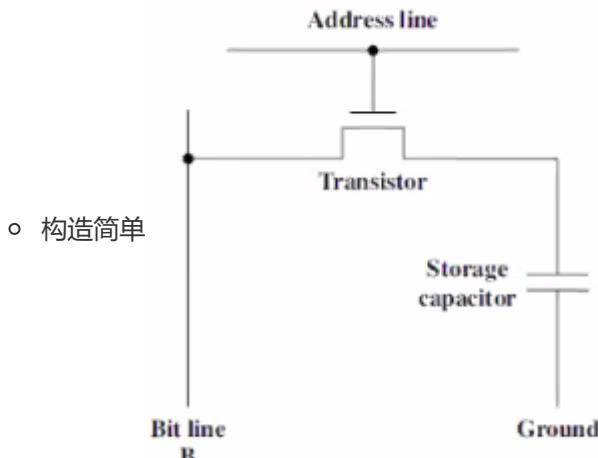
- memory cell

■ The basic element of a semiconductor memory is the memory cell

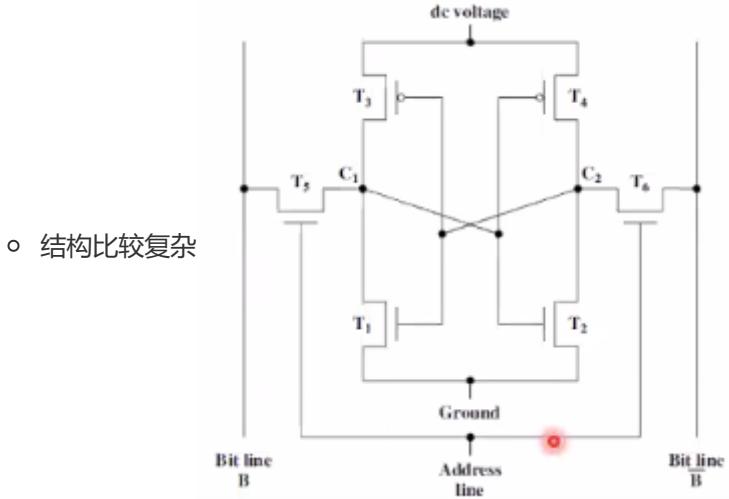
- exhibit two stable states, representing binary 1 and 0
- capable of being written into to set the state
- ■ capable of being read from to sense the state



- 可读可写
- 靠电维持，需要靠电维持状态
- DRAM
  - 用电压存储数据与电容中
  - 需要不断充电维持电压



- 需要设计refresh电路
- 比SRAM便宜且慢
- 用作main memory
- SRAM
  - 存储于开关状态上
  - 不会电压泄露



- 不需要设计refresh
- 比DRAM快且贵
- 用作cache

## ROM

- 永久存储
- 不需要上电维持，非易失的
- 用途
  - Lib库函数
  - BIOS系统程序
  - 函数表
- 种类
  - 出厂时预先写入
  - 允许用户一次编程的 PROM 需要特殊设备来编程
  - 大多数情况下只读存储
    - EPROM 可以用紫外线抹除
    - EEPROM 写入用时比读出要长
    - Flash 用电子方式抹除

## 存储器层级结构

- 寄存器register 在CPU
- 内部主存
  - 可能包括一层或多层高速缓存cache
  - RAM
- 外部存储

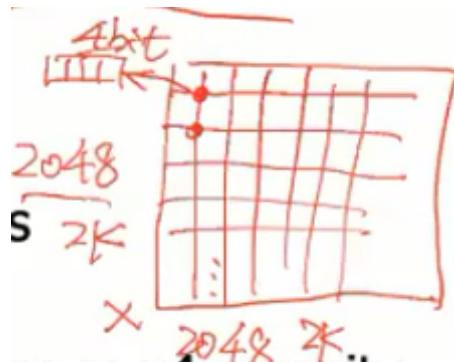
## 层级结构列表

- registers

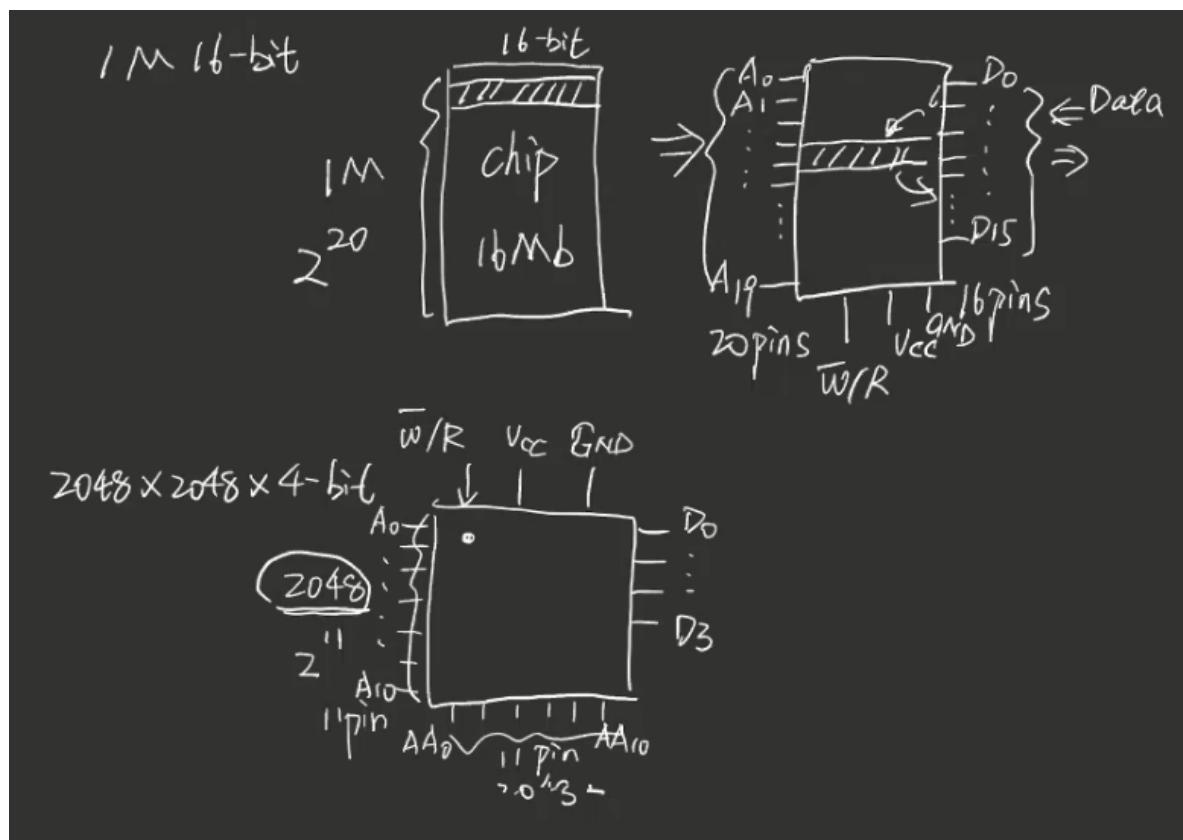
- L1 cache in CPU
- L2 cache in CPU
- 主存
- disk cache
- disk
- optical eg.DVD
- tape

## Chip Organisation 如何物理上把bits安排成words(in chip)

- 16Mbit 可被安排成  $1M \times 16-bit$ ,  $2048 \times 2048 \times 4bit$



两种方案对比



第二种方案

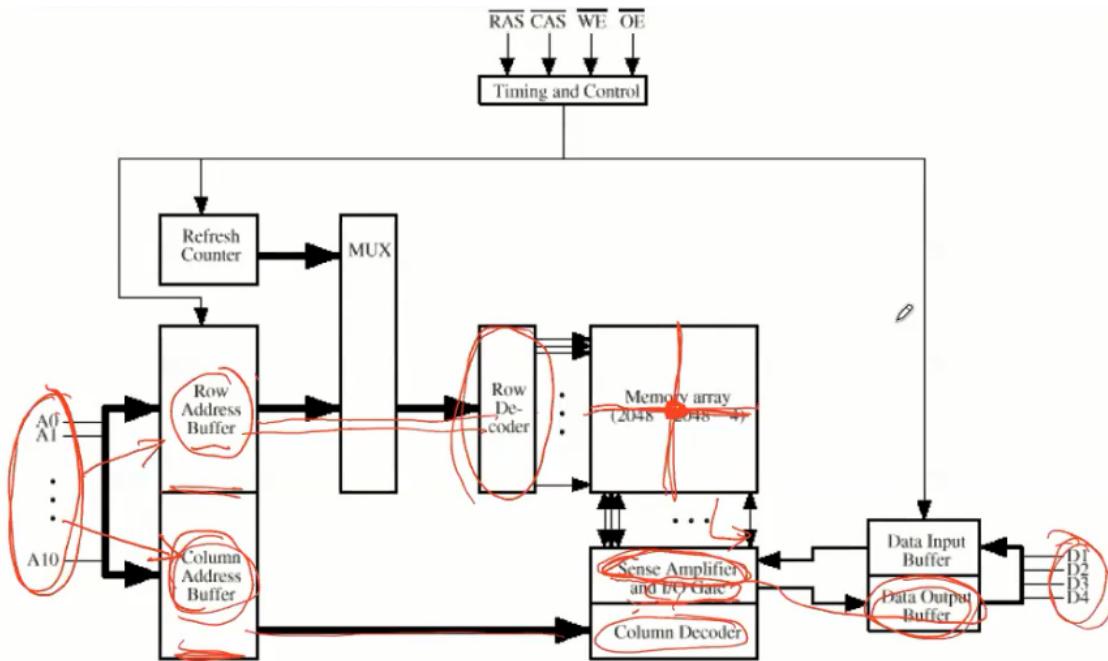
A0到A10时分复用 RAS/CAS控制

refresh counter刷新电路

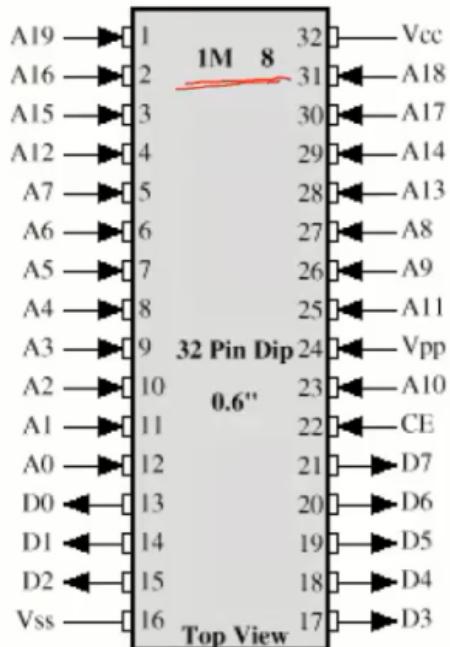
MUX 二选一控制器 选择counter刷新或buffer

WE 写入开关

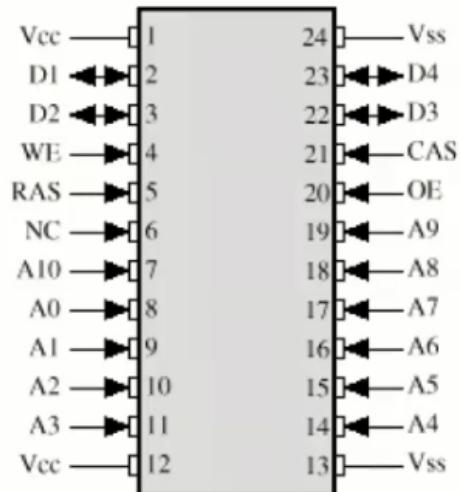
OE 芯片开关



## Chip Packaging



(a) 8 Mbit EPROM



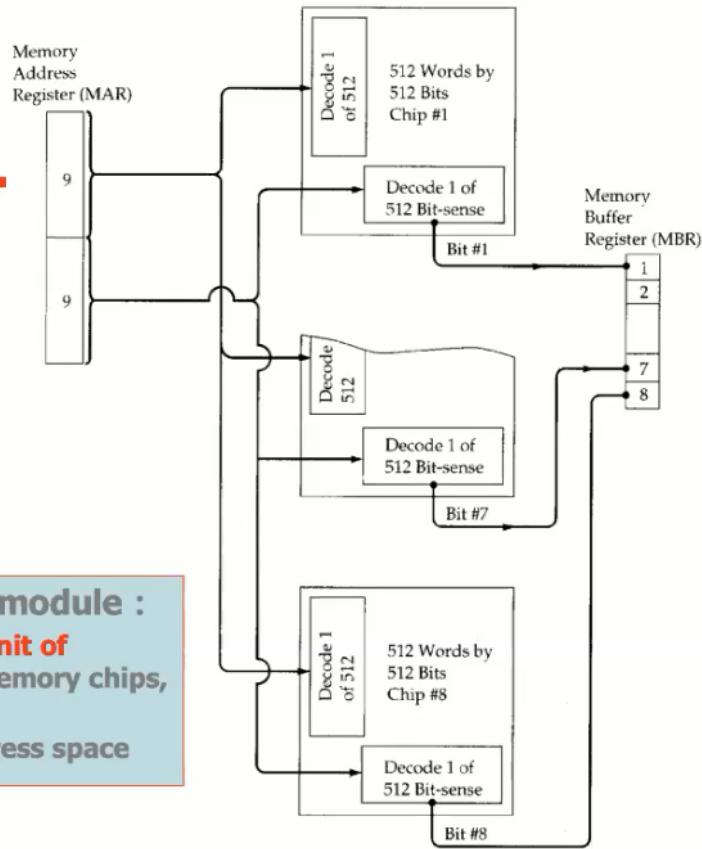
(b) 16 Mbit DRAM

- 1M 8 表示1M个8bit的word，需要20pins
- CE 片选信号 chip enable

## 位拓展

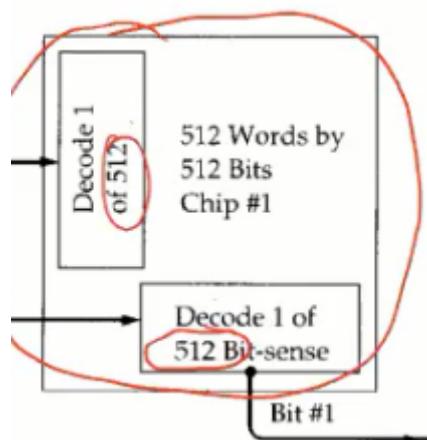
# Module Organisation

A memory module consisting of 256K 8-bit words, using 8 256K×1-bit chips

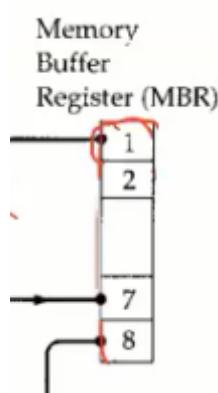


- To organize a memory module :
- ◆ If the module needs bigger **unit of transfer** than that of given memory chips, **bit extension, 位扩展**
- ◆ Every chip has the same address space

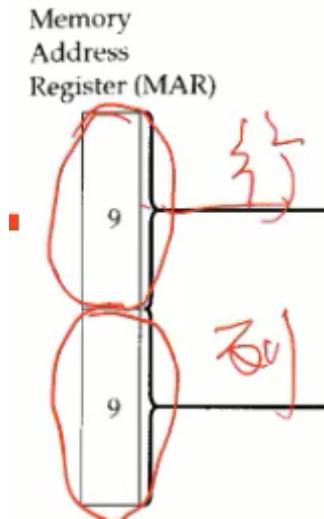
单个芯片



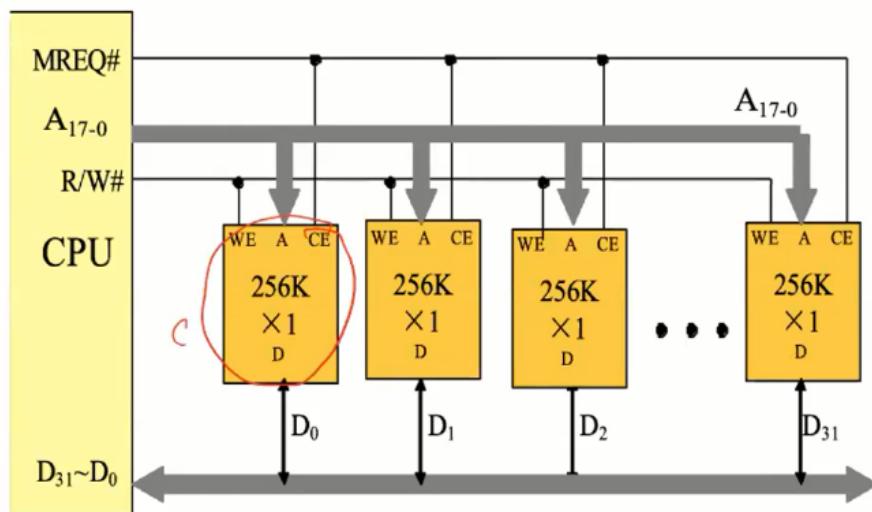
8个芯片分别接入memory buffer register中8个单元



memory address register有9位，对应256K，地址传给每个芯片。每个芯片有同样的地址空间



You have ~~256Kx1-bit RAM chips~~. How can you build a memory module of ~~256Kx32-bit~~ and how to connect this module with a computer system?



D0-D31都提供1bit数据，都接入data bus，接入CPU的D0-D31

$2^{18}$ 个地址，对应A0-A17, address bus接入每个chip进行寻址

R/W 读写信号 送给芯片中的WE

MREQ memory request 给出低电平选中每个芯片

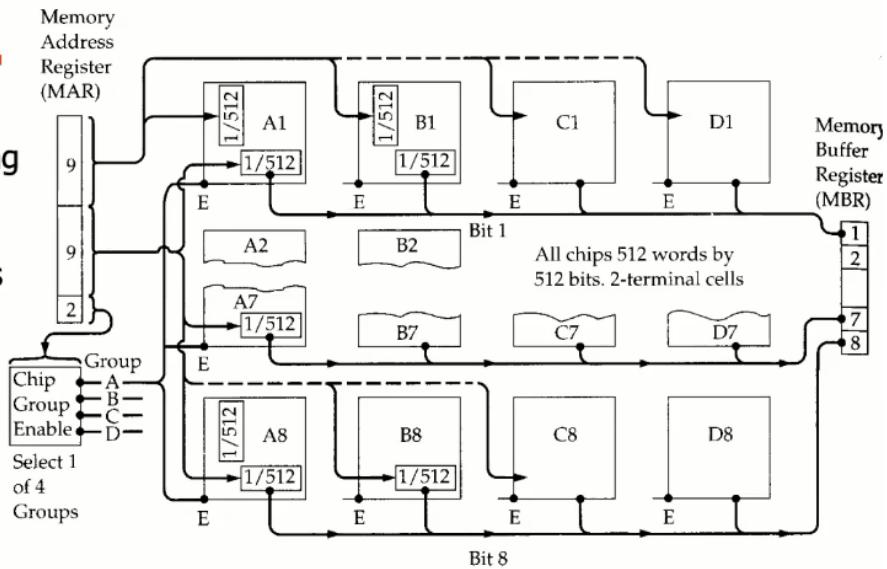
**可寻址单元 (address unit)** 32bit, **寻址方式 (addressing scheme)** isolated IO因为有MREQ, CPU知道访问的是IO还是Memory

## 字扩展

字和位均不同，同时字扩展和位扩展

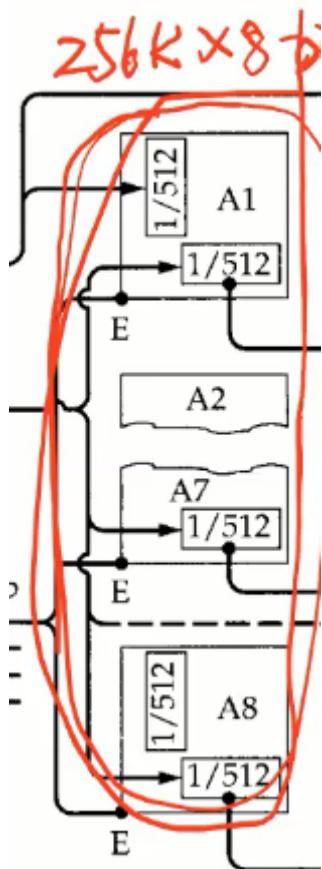
## Module Organisation (2)

A memory consisting of ~~1M × 8-bit words~~, having four groups of ~~256K × 1-bit chips~~

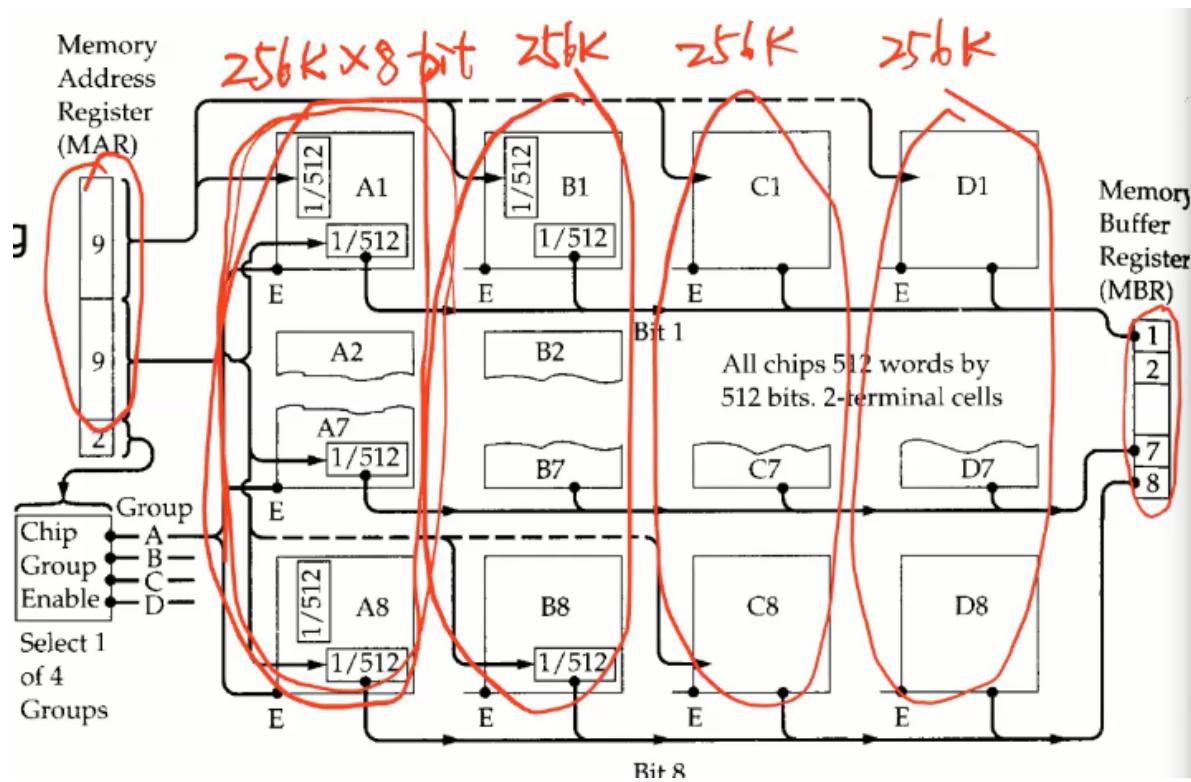


- To organize a memory module :
- ◆ If the module needs larger number of words than that of given memory chips, **word extension**, 字扩展
- ◆ Chips in different group has different address range

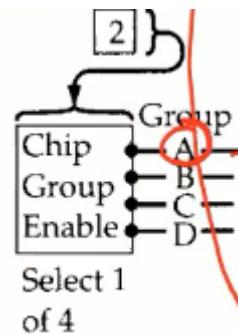
进行位扩展



4个 $256k \times 8bit$ group完成字扩展

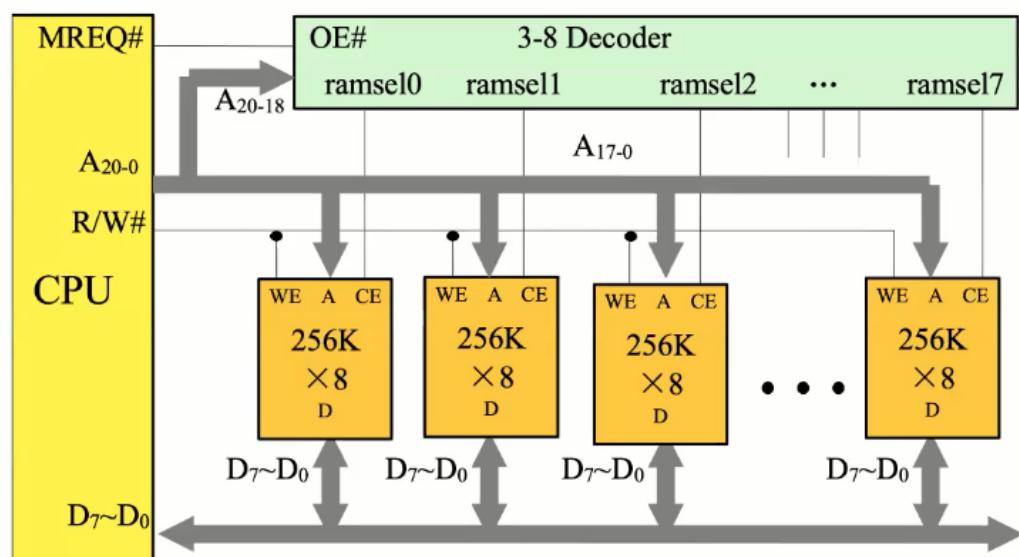


9行9列的memory address register同样传给所有chip，新增加两位地址用于选择group，送给chip group enable译码器。译码器接到4个group，用于enable chip。



字数不同位数相同，做字扩展

You have 256Kx8-bit RAM chips. How can you build a memory module of 2Mx8-bit and how to connect this module with a computer system?



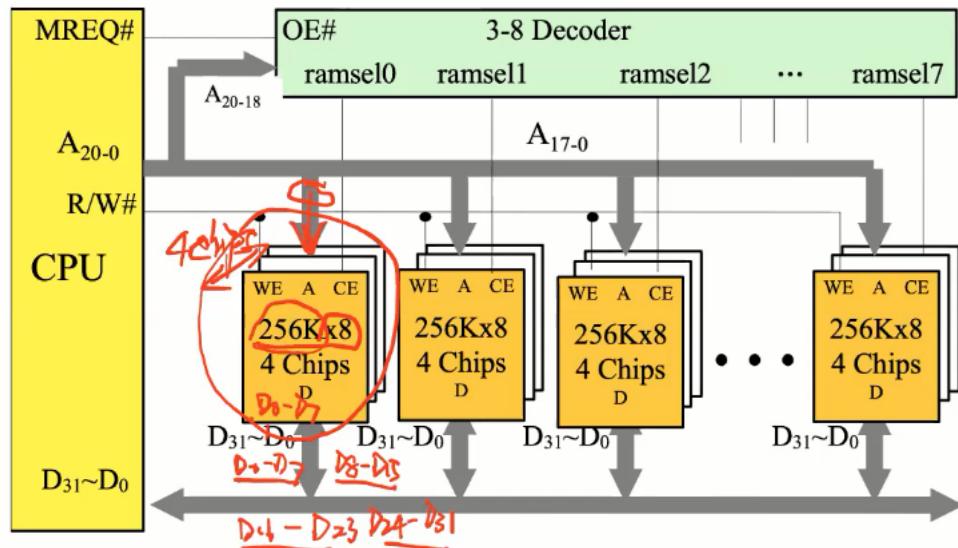
D0-D7从每个芯片通过data bus接到CPU，对应8bit

R/W送给每个芯片的WE

2M对应 $2^{21}$ 个pin，A0-A20，其中A18-A20接到3-8译码器，译码器输出ramsel0-7中选择一个输出高电平，选中该芯片，地址总线送出的数据只有该芯片进行ACK，找到该芯片中的数据。

## 更多例子

Now you have 256Kx8-bit RAM chips. How can you build a memory module of 2Mx32-bit and how to connect this module with a computer system?



相较上个例子需要再做32bit的位扩展，将data bus改为32位，分别接入每个group的4个chip  
addressable unit是32bit。

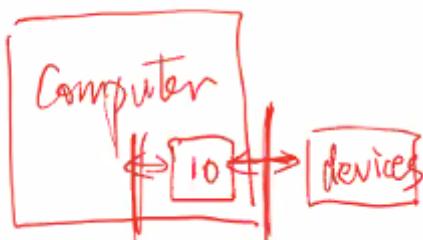
若改变为字节寻址address为一个byte，需要增加2个bit，A0-A22，A20-A22送给3-8译码器选择group，A2-A19送给每个chip，A0-A1设计一个电路，同3-8译码器的信号一同确认一个group中的chip。（字长和可寻址单元大小不相同）

## I/O 问题

- 不同的操作逻辑
- 不同的指令语言
- 不同的速度

## I/O 模块

- 对CPU和memory的接口
- 对外设的接口
- 



- bridge/interpreter/buffer的作用

- 功能

- 控制和计时
- CPU通信
- 设备通信
- 数据缓存
- 错误检测

## 外设

- 人机交互设备

- 屏幕
- 键盘

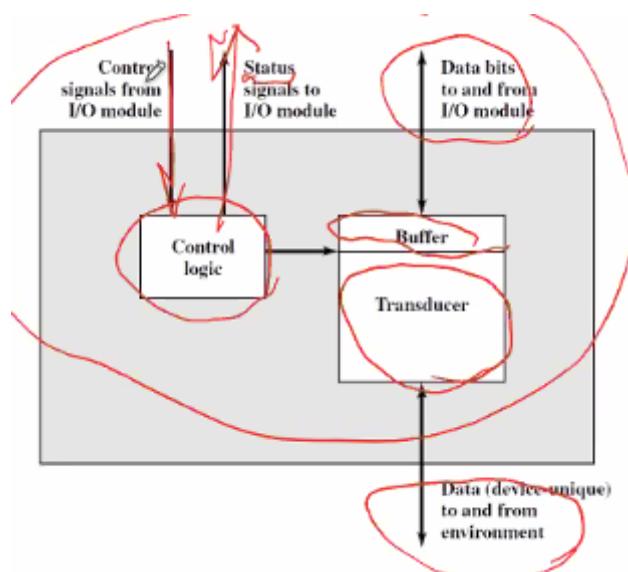
- 机器交互设备

- 监视器控制器等

- 交流设备

- 调制解调器

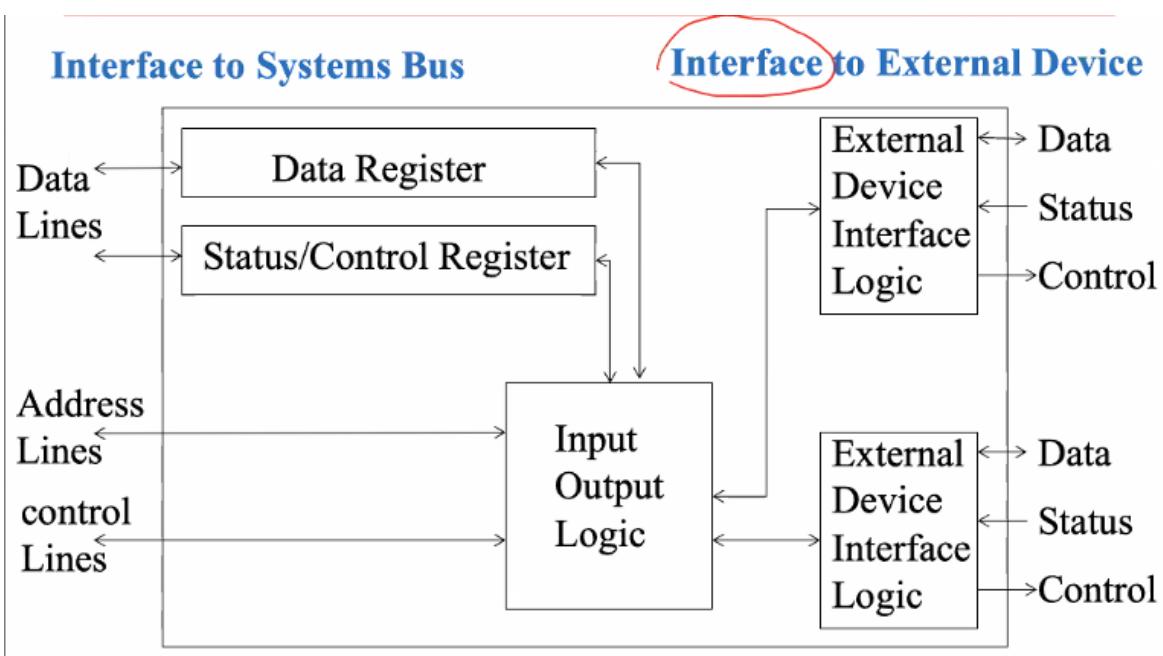
- 外设结构图



## I/O 步骤

- CPU发指令给I/O module, I/O module检测I/O设备状态
- CPU发送数据读写请求给I/O module
- I/O module从设备获取数据
- I/O module将数据发回CPU

## I/O module 结构

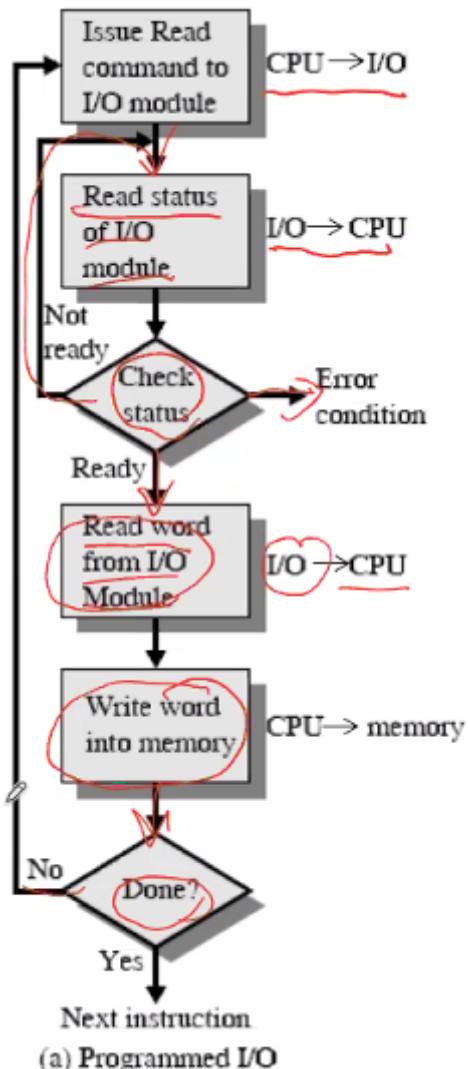


- Status/Control register从data line获取控制字或者I/O状态字

## I/O 技术

### programmed I/O 程序控制I/O

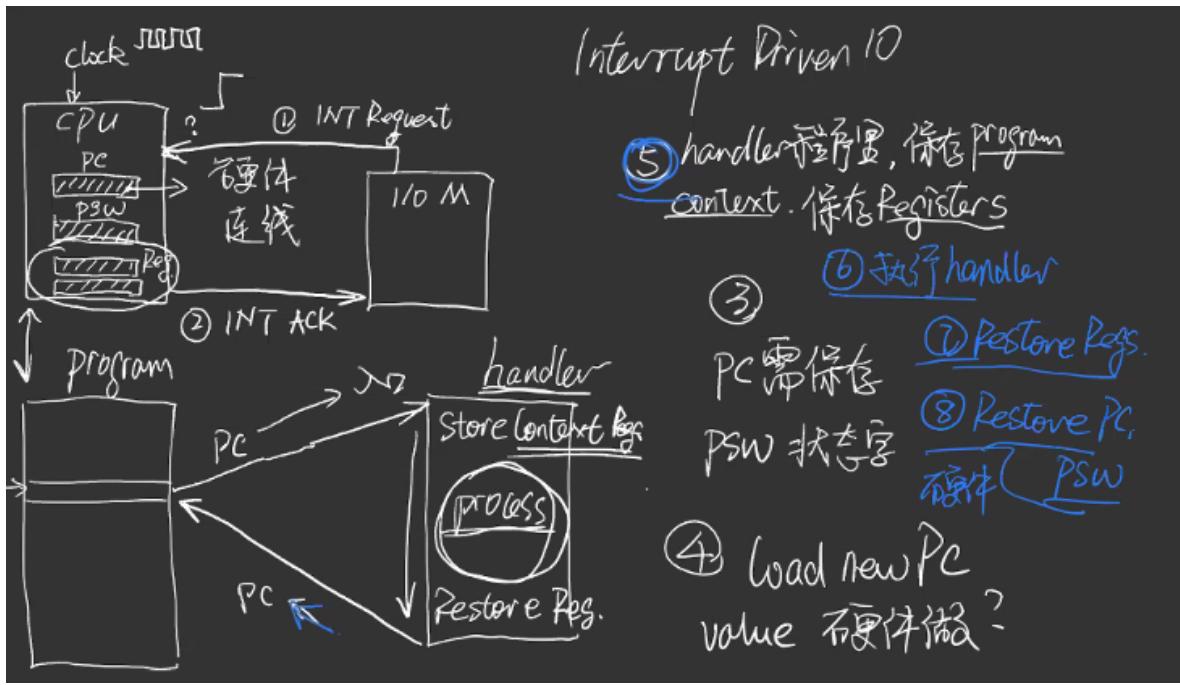
- CPU直接控制I/O操作
  - 检测设备状态
  - 对I/O module进行读写操作
  - 转移数据
- 问题是CPU要等待I/O进行操作
- 步骤
  - CPU requests I/O operation
  - I/O module performs operation
  - I/O module sets status bits
  - CPU checks status bits periodically
  - I/O module does not inform CPU directly
  - I/O module does not interrupt CPU
  - CPU may wait or come back later (for example, with the help of time-sharing OS)
-



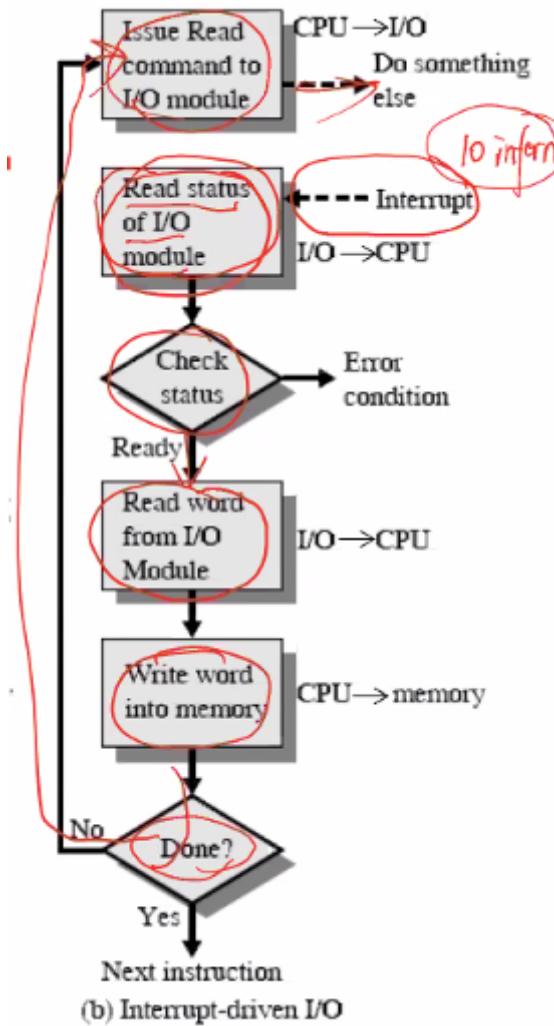
(a) Programmed I/O

- 命令
  - 发送地址
  - 发送指令
    - 控制
    - 测试
    - 读写
- I/O映射
  - 存储器映射IO
    - 设备和内存共享地址空间
    - IO看起来像内存读写
    - 不需要为IO单独设计指令
  - 分离式IO
    - 单独的地址空间
    - 需要IO和内存选择控制线
    - 需要特别的IO指令
- 问题
  - 好处：简单
  - 坏处：浪费CPU资源

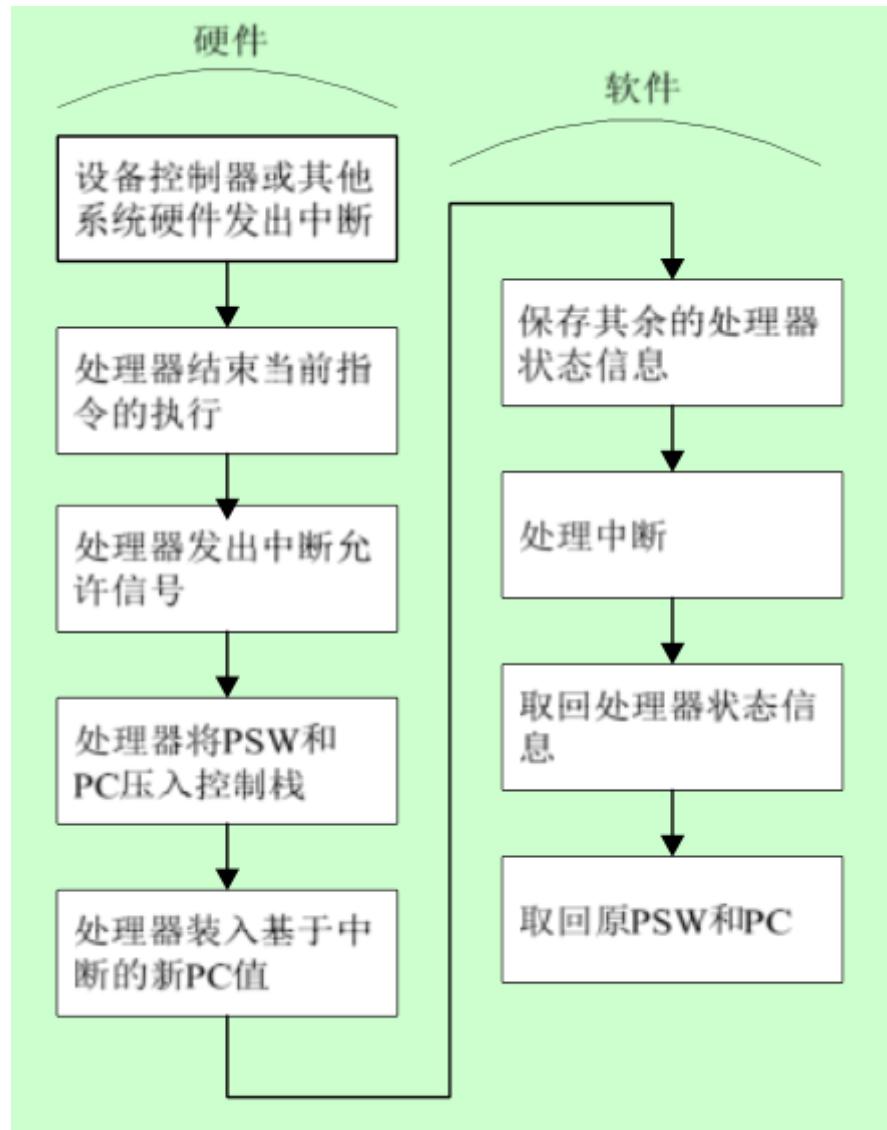
## Interrupt driven I/O, 中断驱动I/O



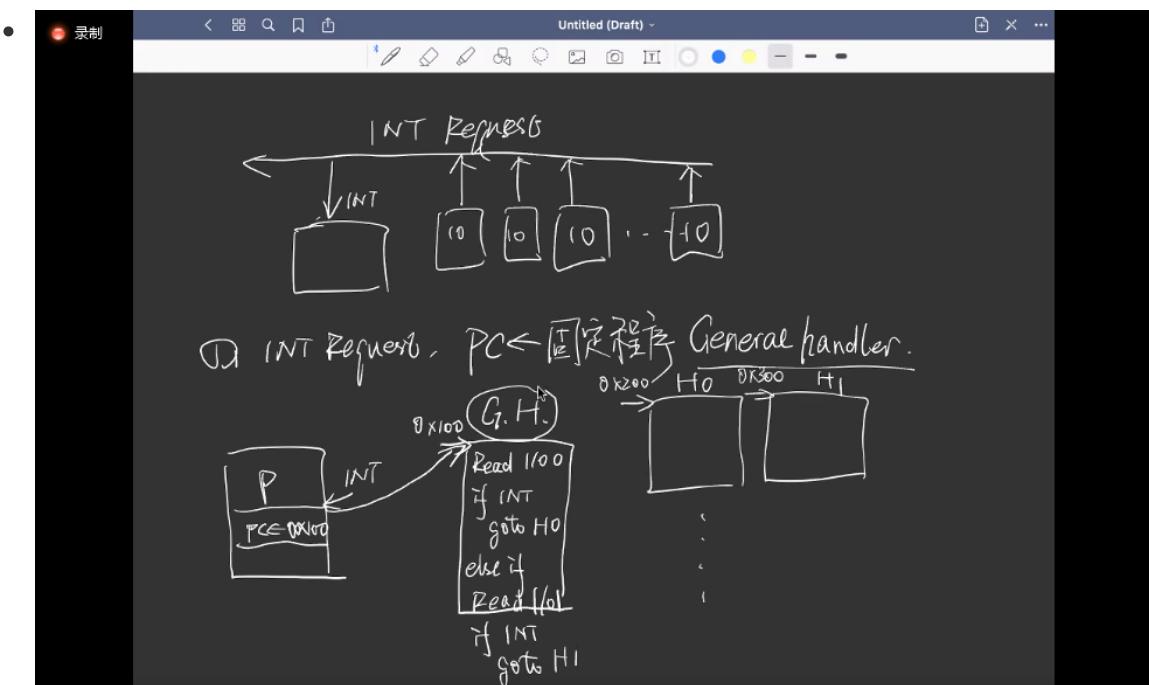
- CPU对外设请求IO操作
- IO module执行操作，CPU返回执行其他任务
- 当状态好了后IO module用中断通知CPU
- 



- 协议角度来看

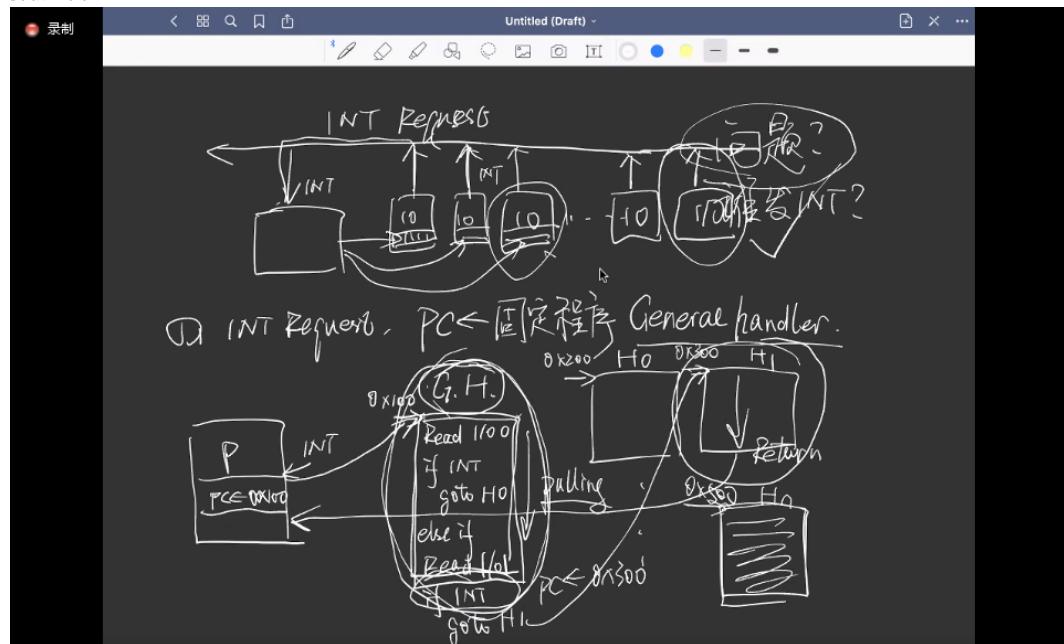


CPU如何得知是哪个模块请求的中断



◦ 轮询

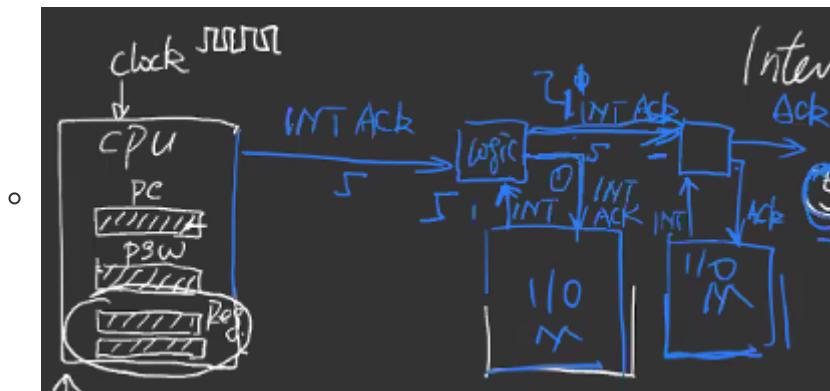
- 拓展后



- 需要修改GH和增添handler
- 菊花链轮询

## Daisy Chain or Hardware poll

- **I All devices share one common Interrupt Request line to interrupt CPU**
- **I Interrupt Acknowledge signal is sent down a chain**



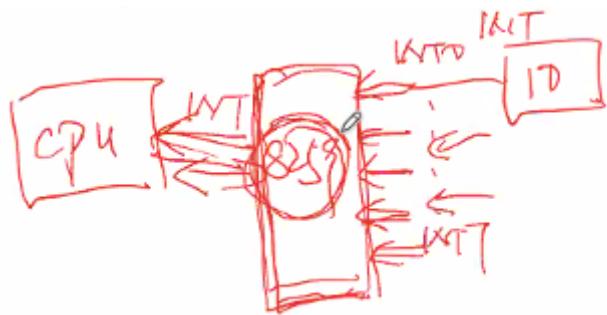
- 硬件逻辑实现，一个逻辑单元检测ACK和INT两个信号，若都为真，则传递INT ACK给I/O，向后传递假的ACK；否则，传递假的ACK给I/O，继续向后传递ACK。
- 先后顺序固定，不够灵活

- 总线控制器

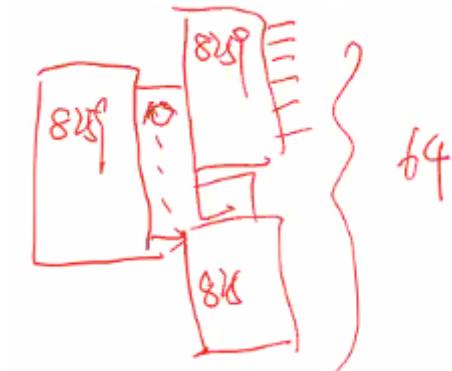
- **Module must claim the bus before it can raise interrupt**

- 中断控制器 8259

-

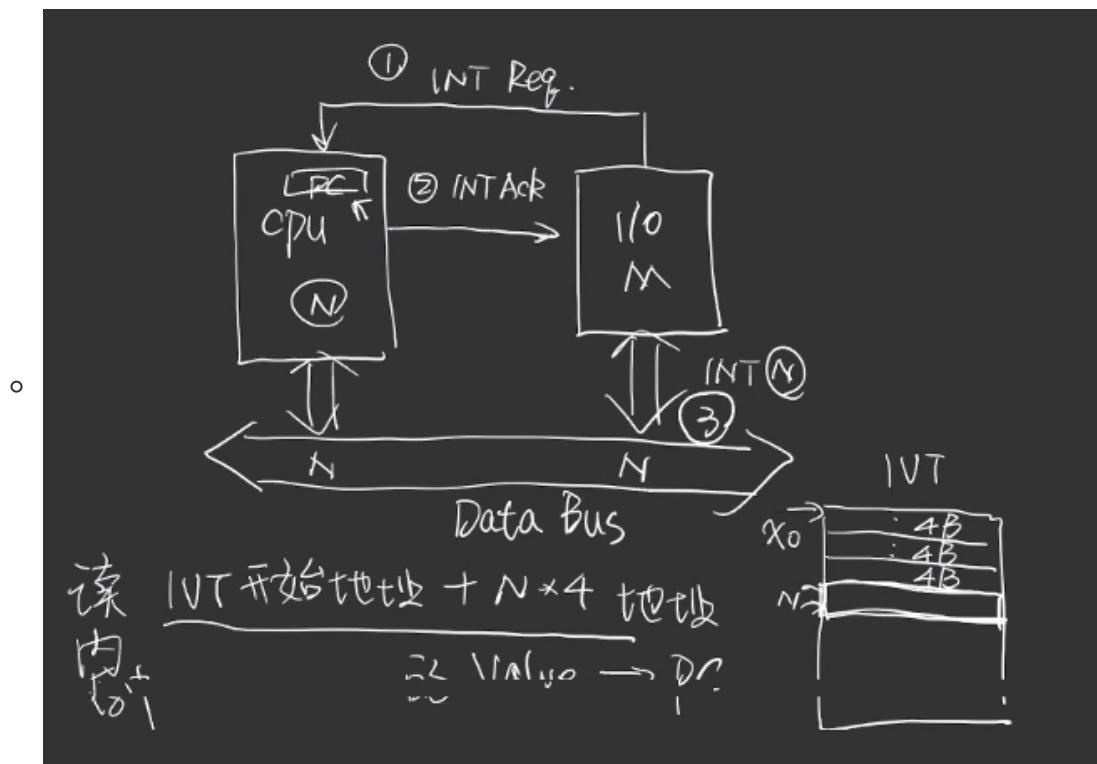


- 可以级联



### 如何定位到对应的中断处理程序

- 通用处理程序
  - 该程序在内存中位置应该固定
  - CPU固定跳转至该通用处理程序，I/O将handler程序地址硬编码至通用处理程序中
  - 性能和灵活性不佳
- 中断向量



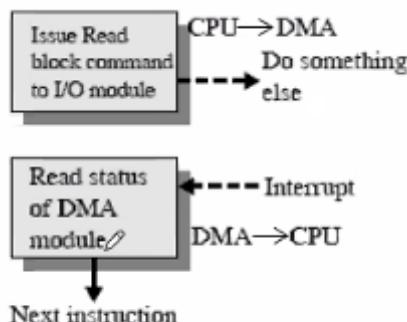
- 中断处理程序可以任意存储
- 通过指针指向中断处理程序
- 指针位置固定，CPU知道该位置
- 多个指针组成中断向量

### 如何处理多中断请求

- 给中断分配优先级
  - 允许嵌套 (nesting) - 高优先级的中断不允许被低优先级中断所中断，高优先级中断可以中断低优先级中断

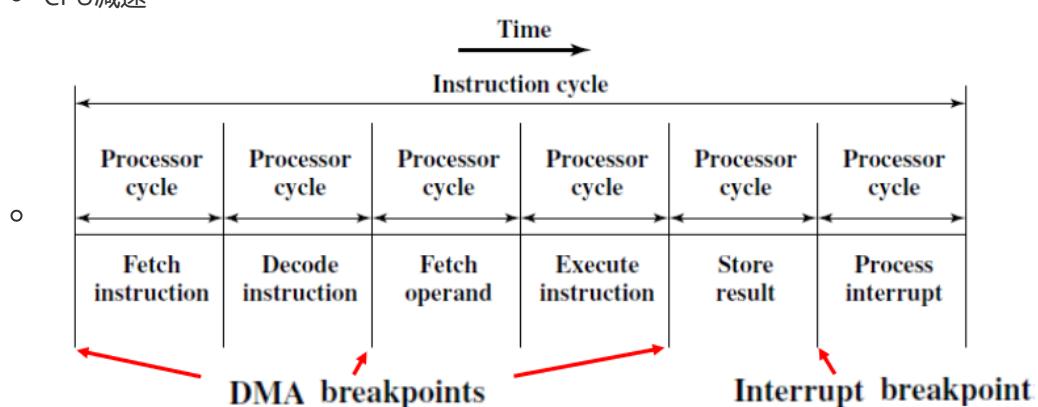
## Direct Memory Access (DMA), 直接存储器访问

- I/O直接与memory进行数据传输
- 通过DMA controller来进行控制 (执行CPU功能)
- CPU告诉DMA四个信息
  - 读写
  - 设备地址
  - 数据起始地址
  - 传输数据大小
- CPU继续其他工作
- DMA处理传输
- DMA完成发送数据后中断
- 



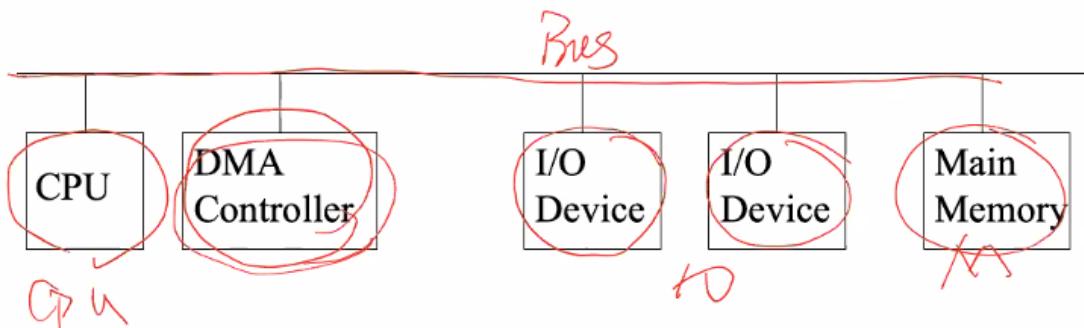
(c) Direct memory access

- DMA Transfer Cycle Stealing
  - CPU访问总线前被挂起
  - DMA接管总线，完成一个bus cycle
  - 每次传输一个字
  - CPU不切换内容
  - CPU减速



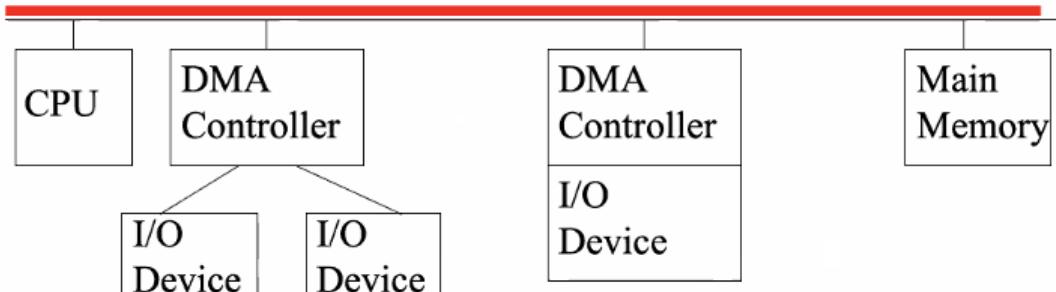
-

# DMA Configurations (1)



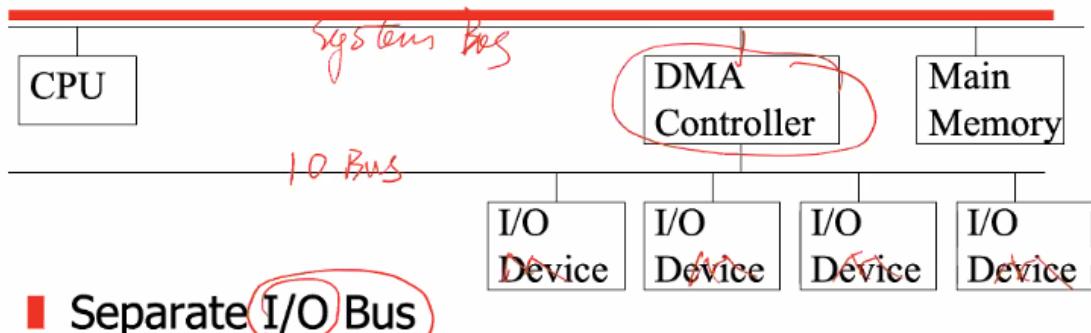
- Single Bus, Detached DMA controller *simple*
- Each transfer uses bus twice *performance?*
  - e.g., I/O to DMA then DMA to memory *Read: Input → M*
- For one transfer, CPU is suspended twice
  - DMA controller先读IO, 再传给Memory

## DMA Configurations (2)



- Single Bus, Integrated DMA controller
- Controller may support >1 device
- Each transfer uses bus once
  - DMA to memory
- For one transfer, CPU is suspended once

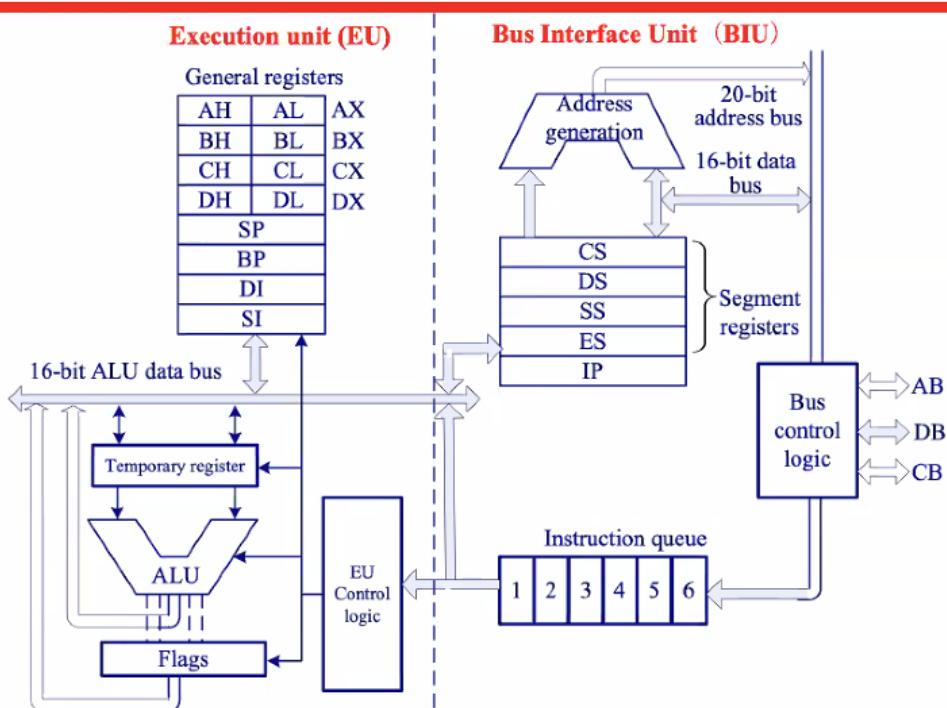
# DMA Configurations (3)



- Separate I/O Bus
- Bus supports all DMA enabled devices
- Each transfer uses bus once
  - DMA to memory
- For one transfer, CPU is suspended once

## Ch 03 8086 微处理器

### Internal Structure of 8086



### BIU

- 负责CPU和memory以及IO间的数据传输
  - 指令获取，指令队列，操作数获取和存储，地址重定位和总线控
- 组成
  - 四个16位段寄存器：CS, DS, ES, SS
  - 一个16位指令指针：IP

- 一个20位地址加法器：例如，CS左移4位+ IP ( $CS * 16 + IP$ )

$CS = 0100\ 0000\ 1010\ 0001\ 0000$

- 6字节指令队列
- 在EU执行指令时，BIU将从内存中获取下一条或多条指令并放入队列中

## EU

- 负责指令执行
- 组成
- 四个16位通用寄存器：累加器 (AX) , Base (BX) , 计数 (CX) 和数据 (DX)
- 两个16位指针寄存器：堆栈指针 (SP) , Base指针 (BP)
- 两个16位索引寄存器：源索引 (SI) 和目标索引 (DI)
- 一个16位标志寄存器：FR 使用16位中的9位
- ALU

## 寄存器

# Registers

- On-chip storage: super fast & expensive
- Store information temporarily

AX 16-bit register		8-bit register:	D7 D6 D5 D4 D3 D2 D1 D0
AH 8-bit reg.	AL 8-bit reg.	16-bit register:	D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

- Six groups

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment), SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

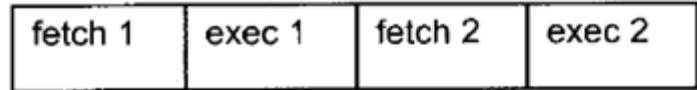
Note:

The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

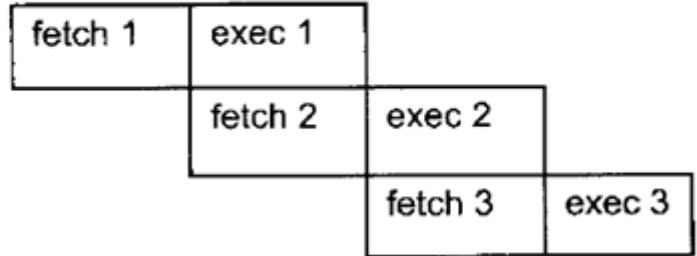
## 8086中的流水

- 一旦队列有2个以上的空字节，BIU就会获取并存储指令
- EU使用同时预取并存储在队列中的指令
- 提高CPU效率

nonpipelined  
(e.g., 8085)



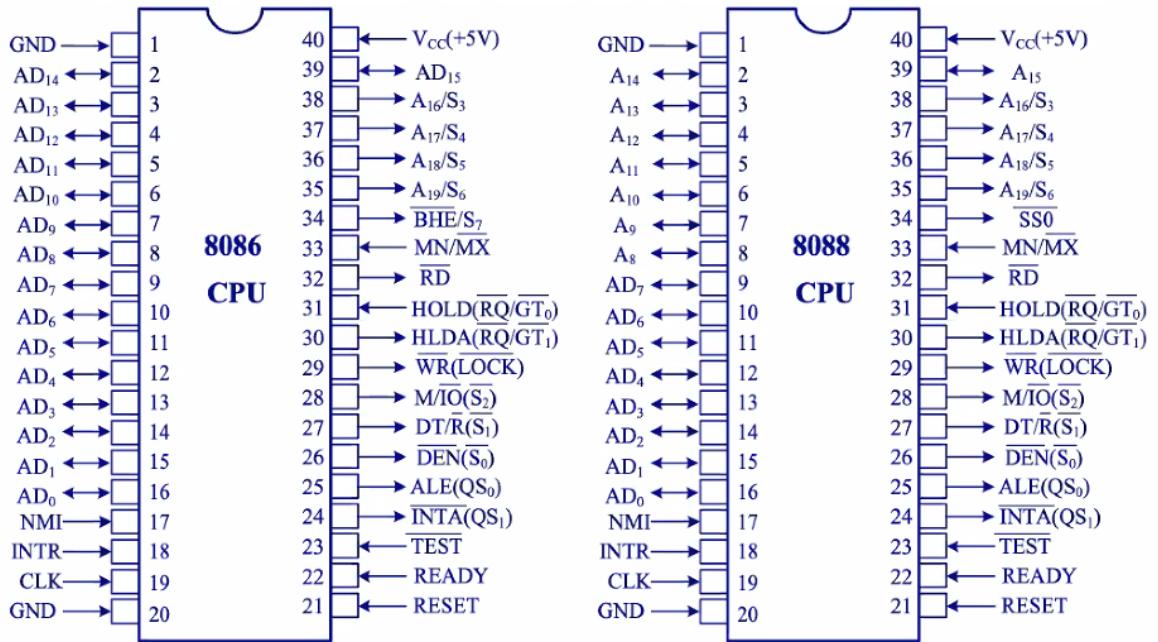
pipelined  
(e.g., 8086)



什么时候起作用?

- 顺序指令执行
- 分支惩罚: 执行跳转指令时, 所有预取指令都将被丢弃

## 8086/8088的接脚



AD: address/data复用

地址线都是20bit

NMI不可屏蔽中断请求

INTR中断请求

CLK时钟

GND接地

$V_{CC}$  电源

MN/MX 最小/最大工作模式

RD 读信号

WR 写信号 WR和RD只有一个低电平

M/IO memory or IO (isolated IO)

DT/R 读写

HOLD 挂起信号 (DMA在循环中控制总线)

HLDA HOLD ACK

INTA INT ACK

TEST 验证信号

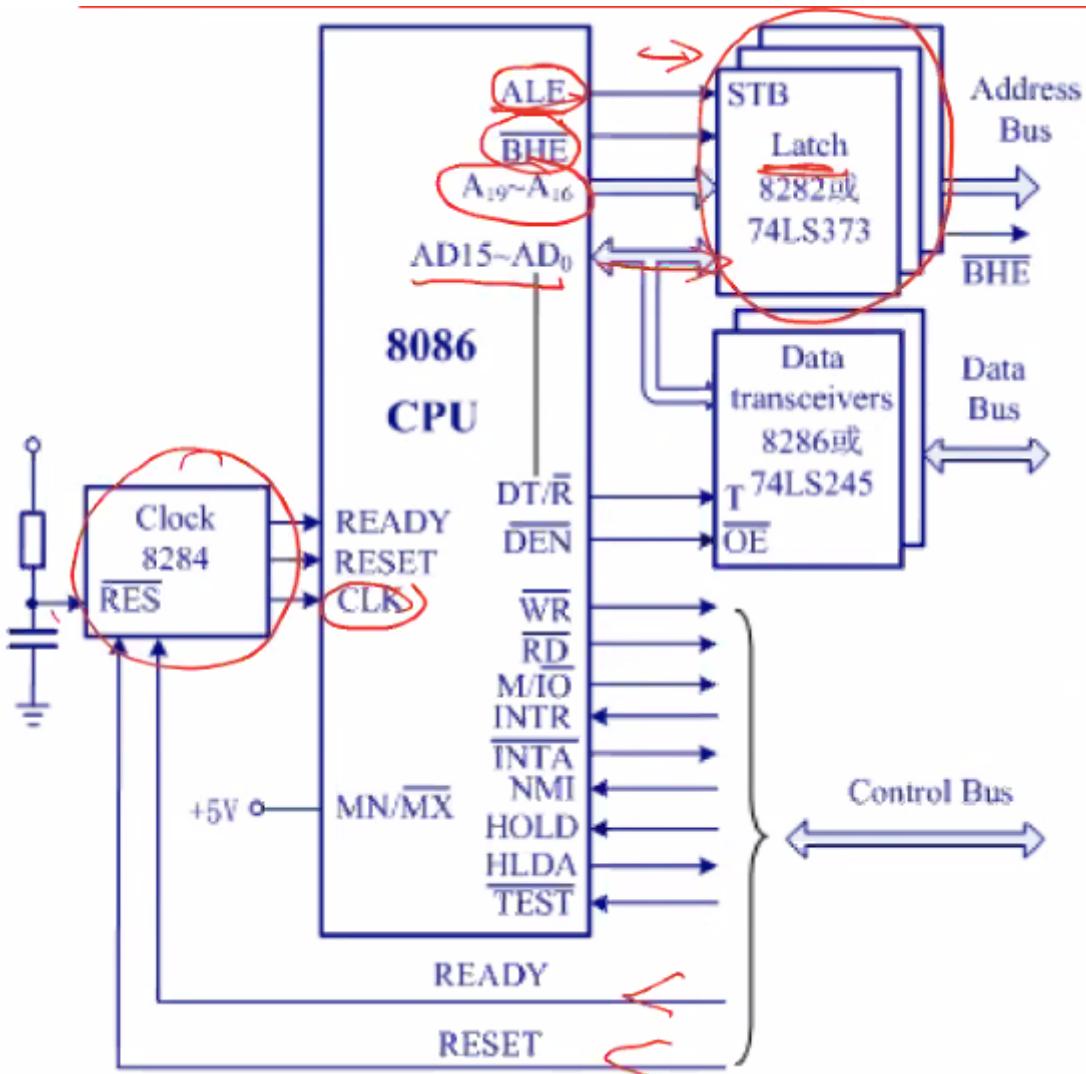
READY I/O传递ready给CPU标志就绪

RESET 重启

ALE 锁存控制信号

## 最大最小工作模式

- **Minimum mode** :  $MN/\overline{MX}=1$ 
  - Single CPU;
  - Control signals from the CPU
- **Maximum mode** :  $MN/\overline{MX}=0$ 
  - Multiple CPUs(8086+8087)
  - 8288 control chip supports



### data transceivers 数据收发器

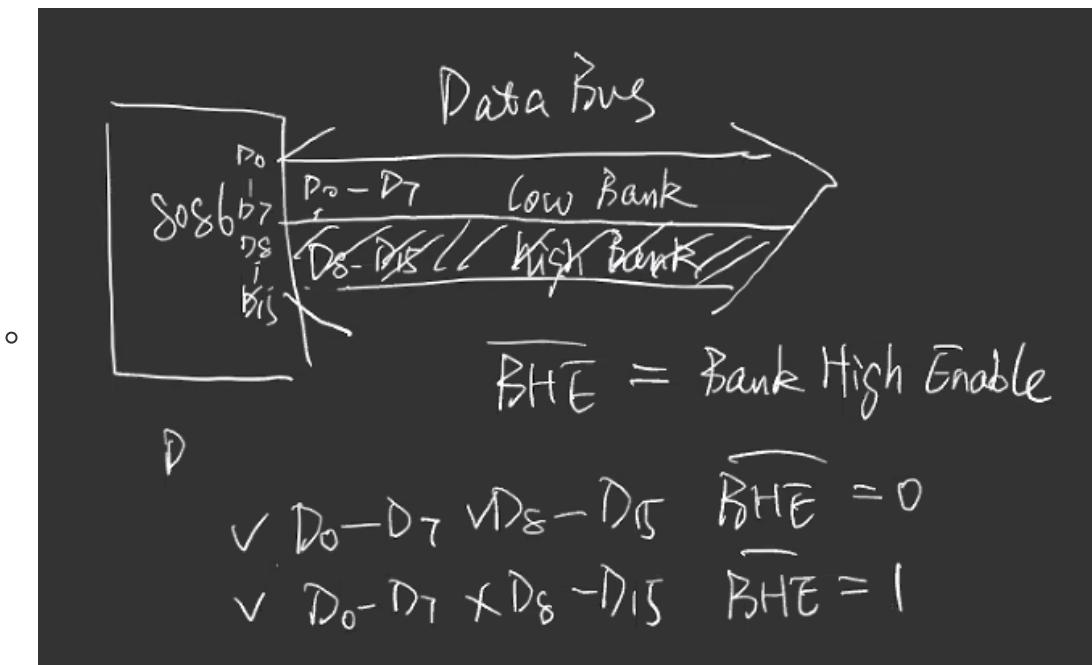
- 增强信号，防止负载过多影响电流
- 左流\右流\分隔

### Latch 锁存器

- 锁定输出

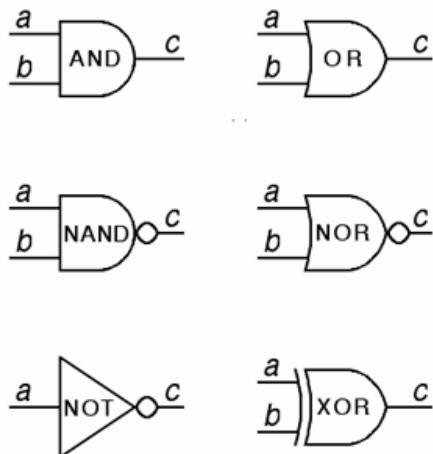
### 接脚详细解释

- $MN / \neg MX$ : 最小模式 (高电平), 最大模式 (低电平)
- $\neg RD$ : 输出, CPU正在从内存或IO读取
- $\neg WR$ : 输出, CPU正在写入内存或IO
- $M / \neg IO$ : 输出, CPU正在访问内存 (高电平) 或IO (低电平)
- READY: 输入, 存储器/ IO准备进行数据传输
- $\neg DEN$ : 输出, 用于启用数据收发器 (data transceivers)
- $\neg DT / \neg R$ : 输出, 用于通知数据收发器数据传输的方向, 即发送数据 (高电平) 或接收数据 (低电平)
- $\neg BHE$ : 输出,  $\neg BHE = 0$ , 使用 $AD_8 \sim AD_{15}$ ,  $\neg BHE = 1$ , 未使用 $AD_8 \sim AD_{15}$



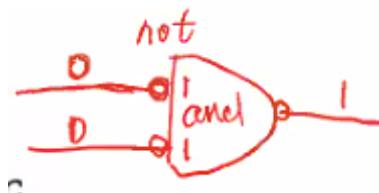
- ALE: 输出, 用作地址锁存器的锁存使能信号
- HOLD: 输入信号, 保持总线请求
- HLDA: 输出信号, 保持请求确认
- INTR: 输入, 来自8259中断控制器的中断请求, 可通过清除标志寄存器中的IF屏蔽
- INTA: 输出, 中断确认
- NMI: 输入, 不可屏蔽中断, 完成当前指令后CPU中断; 不能被软件掩盖
- RESET: 输入信号, 复位CPU
  - IP, DS, SS, ES和指令队列已清除
  - CS = FFFFH
  - 复位后CPU将执行的第一条指令的地址是什么?

FFFF, 左移四位FFFF0, 加上IP为0, 为FFFF0



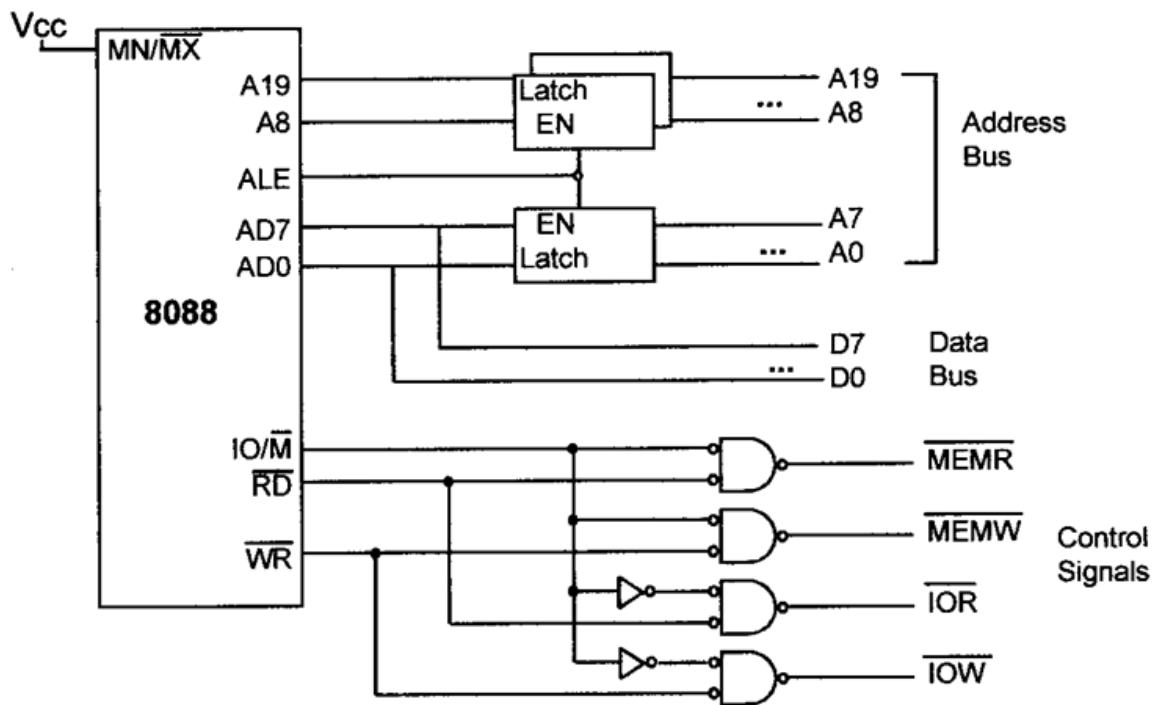
inputs		the output $c$					
$a$	$b$	AND	OR	NAND	NOR	NOT	XOR
0	0	0	0	1	1	1	0
0	1	0	1	1	0	1	1
1	0	0	1	1	0	0	1
1	1	1	1	0	0	0	0

Boolean logic operations

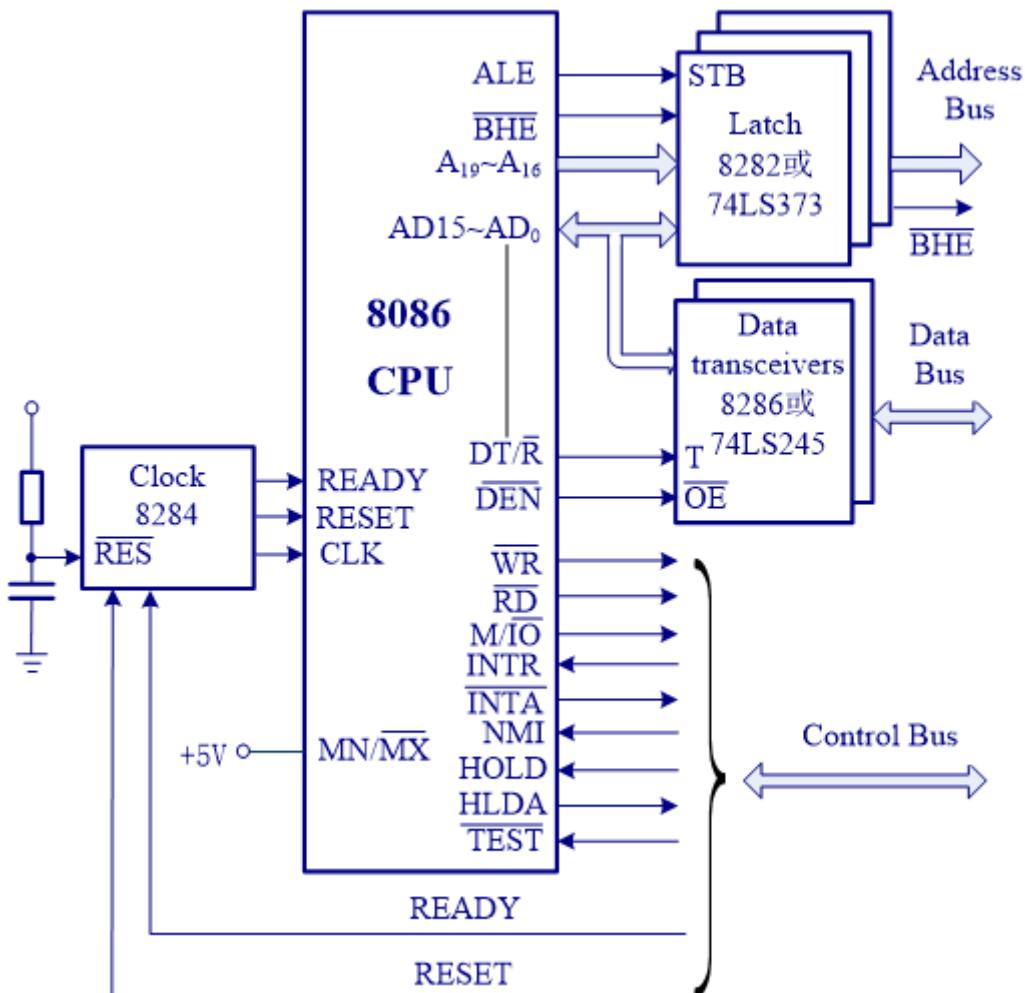


◦ 指取非

## Memory/IO Control Signals



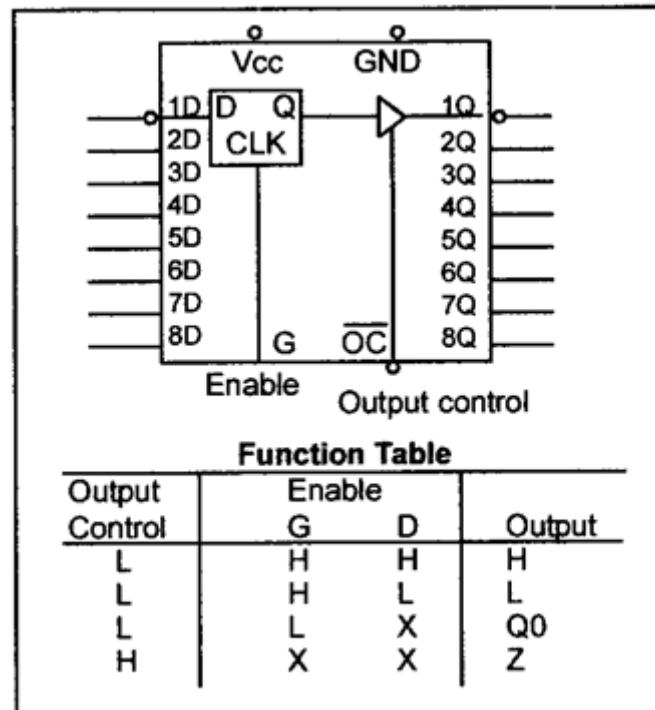
## Address/Data 复用 & Address latching



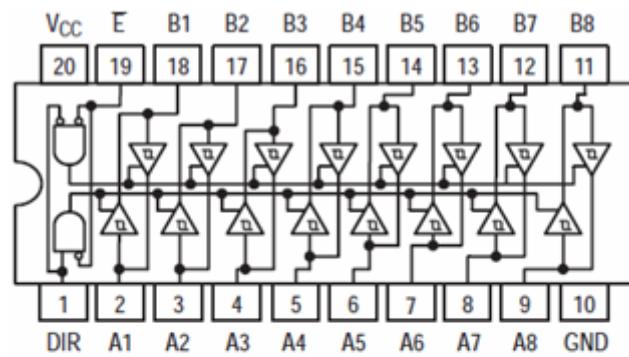
分时复用

latch可锁住 $AD_{15} \sim AD_0$ 地址，之后 $AD_{15} \sim AD_0$ 可传data

Address latching设计

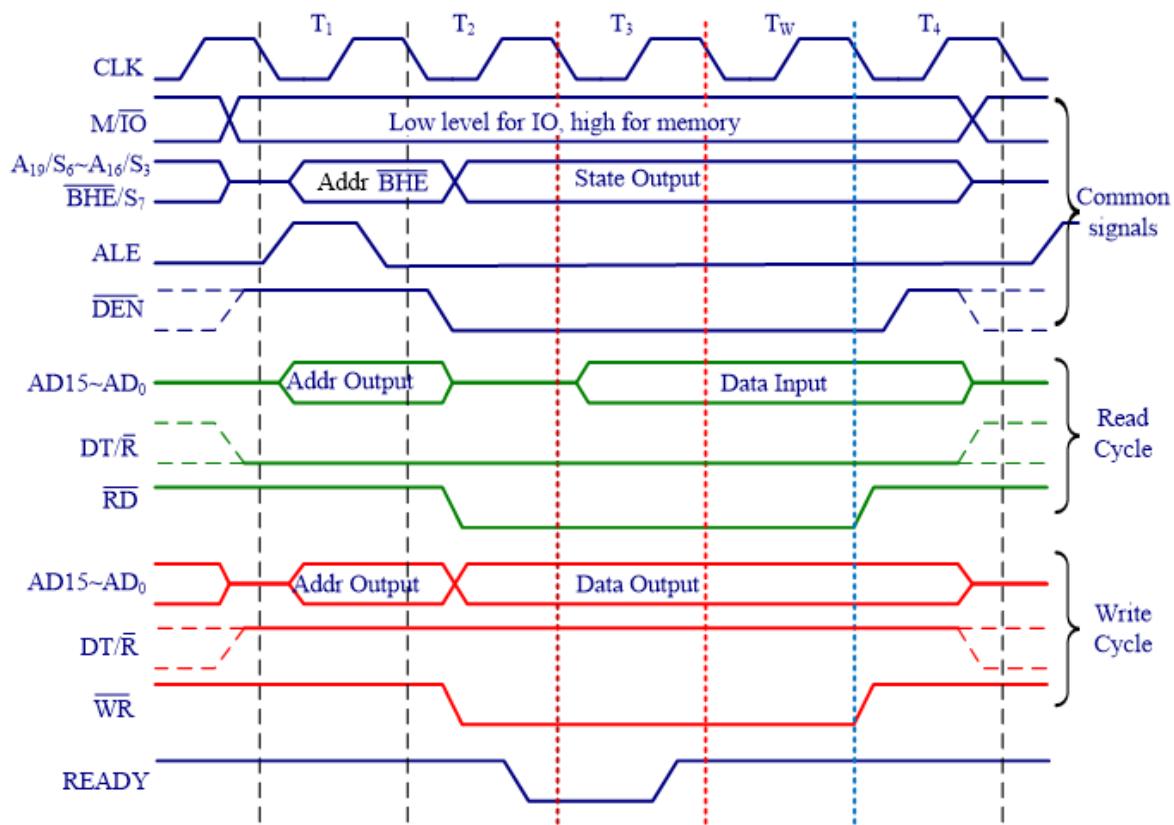


transceiver的设计 两个与门控制



INPUTS		OUTPUT	
E	DIR		
L	L	Bus B Data to Bus A	
L	H	Bus A Data to Bus B	
H	X	Isolation	

8086/88 Bus Cycle (for data transfers)



## 8086 Programming

8086上的典型程序至少包含三个部分

- CS代码段：包含完成某些任务的指令
- DS数据段：存储要处理的信息
- SS堆栈段：临时存储信息
- ES额外

### 什么是段segment?

内存中一个连续的空间

一个存储块最多包括64KB。为什么？

字长为16位，寄存器是16位，内部储存地址范围是16bit

从一个可被16整除的地址开始，即一个地址看起来像XXXX0H。为什么？

左移4位形成的

### 物理地址和逻辑地址

#### 物理地址

实际上放在地址总线上的20位地址

00000H至FFFFFH的1MB范围

内存中的实际物理位置

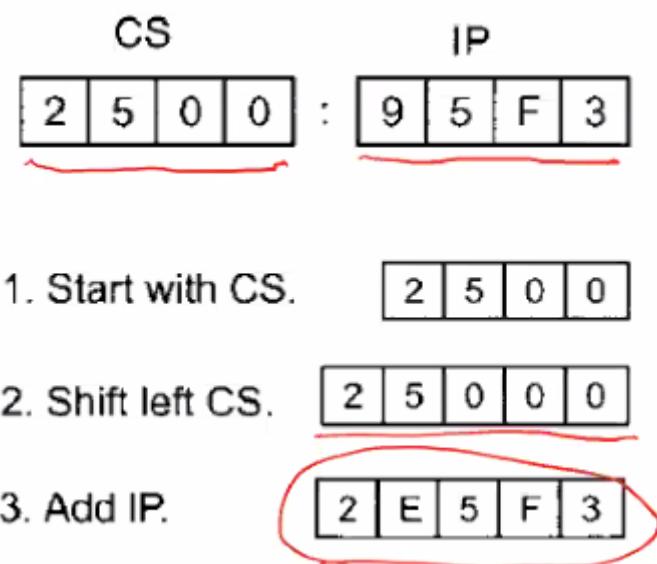
#### 逻辑地址

由段值（确定段的开始）和偏移地址（64KB段内的相对位置）组成

例如，代码段中的一条指令的逻辑地址为CS（代码段寄存器）的形式：IP（指令指针） seg:offset

#### 逻辑地址映射到物理

段值左移1位+IP，逻辑地址和物理地址一一对应



### 段值范围

Maximum 64KB

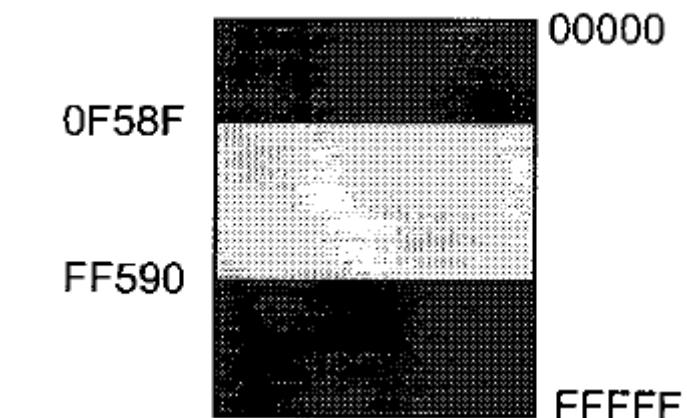
logical 2500:0000 – 2500:FFFF

Physical 25000H – 34FFFFH (25000 + FFFF)

### Wrap-around

adding the offset to the shifted segment value results in an address beyond the maximum value FFFFFH

The low range is FF590H, and the range goes to FFFFFH and wraps around from 00000H to 0F58FH (FF590+FFFF).



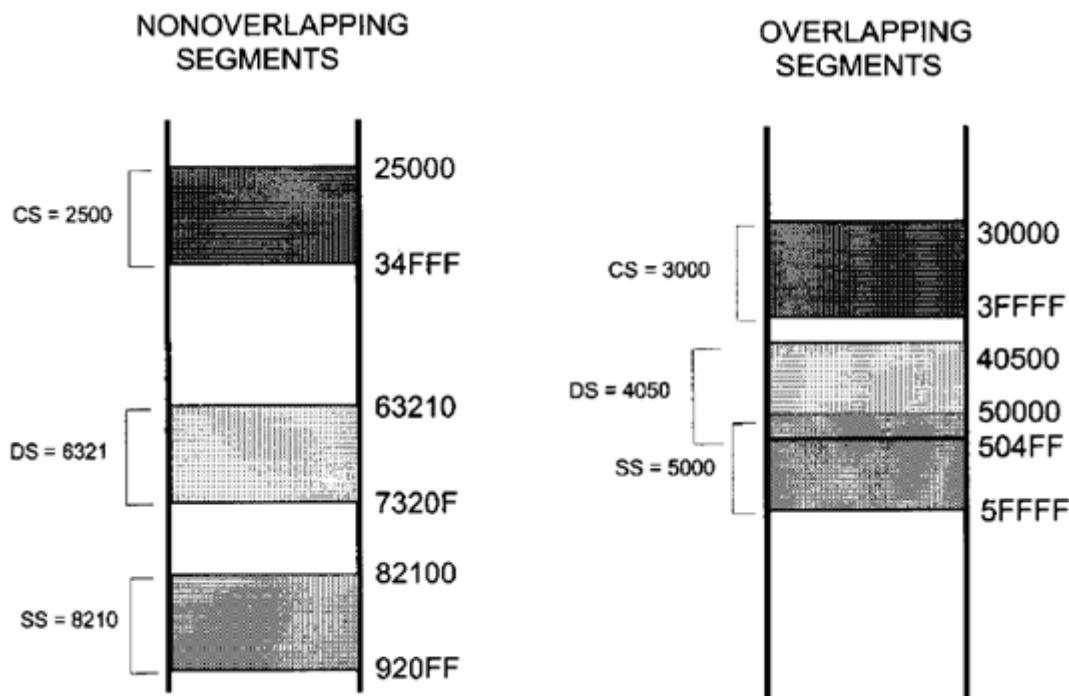
### 物理地址映射到逻辑地址

不是一一对应

<u>Logical address (hex)</u>	<u>Physical address (hex)</u>
1000:5020	15020
1500:0020	15020
1502:0000	15020
1400:1020	15020
1302:2000	15020

## 重叠

可以通过重叠实现shared memory



## 几个segment

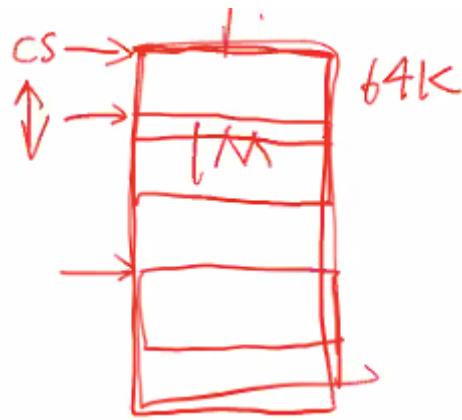
### cs code segment

8086从代码段获取指令

- 指令的逻辑地址: CS: IP 无法手动修改
- 生成物理地址以从存储器中检索该指令
- 如果所需的指令实际位于当前代码段之外怎么办?

通过移动CS来访问

通过语句指令jump, call procedure



- 解决方案：更改CS值，以便可以使用新的逻辑地址查找这些指令

### **ds data segment**

要处理的信息存储在数据段中

- 一条数据的逻辑地址：DS： offset
  - 偏移值：例如0000H, 23FFH
  - 数据段的偏移寄存器：BX, SI和DI
- 生成物理地址以从内存中检索数据（8位或16位）
- 如果所需数据实际位于当前数据段之外怎么办？
- 解决方案：更改DS值，以便可以使用新的逻辑地址定位这些数据，物理上可以向DS写入值（先将值写入通用寄存器，再由通用计算器写入DS）
- 逻辑上可以将内存想象为连续的字节块
- 如何存储大小大于字节的数据？
  - 小尾数法：数据的低字节进入低内存位置 34/12（12高字节 34低字节）
  - 大尾数法：数据的高字节进入低内存位置 12/34

### **ss stack segment**

是CPU用于临时存储信息的一部分RAM存储器

- 一条数据的逻辑地址：SS： SP（带有BP的特殊应用程序）
- 可以将CPU内部的大多数寄存器（段寄存器和SP除外）存储在堆栈中，并分别使用push和pop从堆栈中带回CPU中。
- 从分配给程序的内存中的高位地址向下扩展到低位地址
  - 为什么？保护其他程序不受破坏
  - 注意：确保代码部分和堆栈部分不会互相覆盖

### **push&pop**

# 16-bit operation

## Example 1-6

Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed:

PUSH AX  
PUSH DI  
PUSH DX

**Solution:**

SS:1230

SS:1231

SS:1232

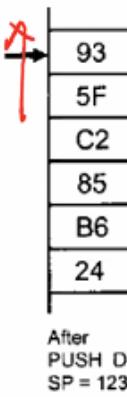
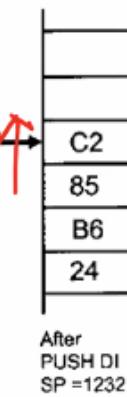
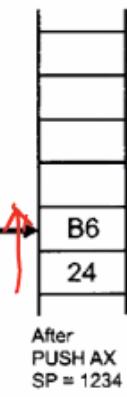
SS:1233

SS:1234

SS:1235

SS:1236

START  
SP = 1236



小尾数法

## Example 1-7

Assuming that the stack is as shown below, and SP = 18FA, show the contents of the stack and registers as each of the following instructions is executed:

POP CX  
POP DX  
POP BX

**Solution:**

SS:18FA

SS:18FB

SS:18FC

SS:18FD

SS:18FE

SS:18FF

SS:1900

START  
SP = 18FA

After  
POP CX  
SP = 18FC  
CX = 1423

After  
POP DX  
SP = 18FE  
DX = 2C6B

After  
POP BX  
SP = 1900  
BX = F691

*push AX  
push BX  
push AX, BX  
POP AX  
POP BX  
POP AX*

先进后出 后进先出

es extra segment

额外的ds data segment, 对于字符串操作至关重要

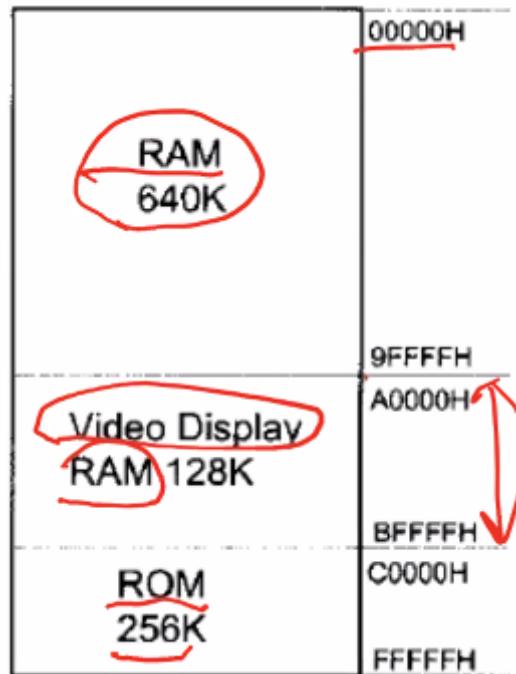
- 一条数据的逻辑地址: ES: offset
  - 直接给出偏移值: 例如0000H, 23FFH
  - 使用偏移寄存器: BX, SI和DI

Table 1-3: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

IBM PC memory map

- 1MB逻辑地址空间
- 640K-最大RAM
  - 在1980年代, 64kB-256KB
  - MS-DOS, 应用软件
  - DOS执行内存管理; 没有设置CS, DS和SS, 但仍然需要装载DS
- 视频显示RAM
- 只读存储器
  - 64KB BIOS
  - 各种适配器卡



## BIOS 功能

- 基本输入输出系统 (BIOS)
  - 开机时测试连接到PC的所有设备, 并报告错误 (如果有)
  - 将DOS从磁盘加载到RAM
  - 将PC的控制权移交给DOS
- CPU复位后, CPU将执行的第一条指令是什么?
  - 除了CS都清零, CS恢复至0FFFFH, IP为0。FFFFOH地址取指令, 属于BIOS范围

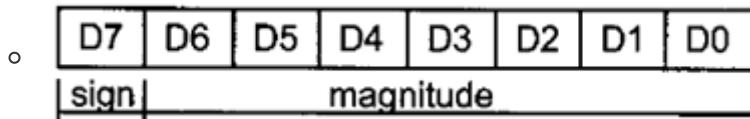
## 状态寄存器 PSW

- 16位状态寄存器, 处理器状态字 (PSW)
- 6个条件标志
  - CF (carry进位), PF (奇偶校验位), AF (第三到第四位有无进位), ZF (是否为0), SF (符号位) 和OF (overflow)
  - CF (进位标志) : 每当有进位时设置, 从8位运算后的d7开始, 从16位运算后的d15开始
  - PF (奇偶校验标志) : 运算结果的低位字节的奇偶校验, 当字节中1的个数为偶数时设置为1
  - AF (辅助进位标志) : 设置是否存在从d3到d4的进位, 由BCD相关算法使用
  - ZF (零标志) : 结果为零时设置
  - SF (符号标志) : 从op之后的符号位 (最高有效位) 复制
  - OF (溢出标志) : 当有符号数字运算的结果太大而导致符号位错误时置位
- 3个控制标志

- DF (direction flag) , IF (interrupt flag, 决定是否响应INTR) , TF (try flag 单步调试)
- IF (中断标志) : 设置或清除以仅启用或禁用外部可屏蔽中断请求  
复位后, 所有标志均被清除, 这意味着您(作为程序员)如果允许INTR, 则必须在程序中设置IF。  
DF (方向标志) : 指示字符串操作的方向  
TF (陷阱标志) : 设置后, 它允许程序执行单步操作, 这意味着一次执行一条指令以进行调试

## 有符号数 CF&OF

- 最高有效位 (MSB) 为符号位, 其余位为幅度



- 对于负数, D7为1, 但幅度以2的补码表示
- CF用于检测无符号算术运算中的错误
- OF用于检测有符号算术运算中的错误
  - 例如, 对于8位运算, 当从d6到d7或从d7进位但不同时有两个进位时设置OF

$$\begin{array}{r}
 & 38 & 0011 & 1000 \\
 + & \underline{2F} & \underline{0010} & \underline{1111} \\
 & 67 & 0110 & 0111
 \end{array}$$

CF = 0 since there is no carry beyond d7  
 PF = 0 since there is an odd number of 1s in the result  
 AF = 1 since there is a carry from d3 to d4  
 ZF = 0 since the result is not zero  
 SF = 0 since d7 of the result is zero  
 OF = 0 since there is no carry from d6 to d7 and no carry beyond d7

$$\begin{array}{r}
 + 96 & 0110 0000 \\
 + \underline{70} & \underline{0100 0110} \\
 + 166 & 1010 0110
 \end{array}$$

According to the CPU, this is -90,  
which is wrong. (OF = 1, SF = 1, CF = 0)

$$\begin{array}{r}
 -128 & 1000 0000 \\
 + -\underline{2} & \underline{1111 1110} \\
 -130 & 0111 1110
 \end{array}$$

According to the CPU, the result is +126.  
OF=1, SF=0 (positive), CF=1

## 8086寻址方式

- CPU如何访问操作数 (数据)
- 80X86具有七种不同的寻址模式
  - Register
  - Immediate
  - Direct
  - Register indirect
  - Based relative
  - Indexed relative
  - Based indexed relative
- 以MOV指令为例  
MOV destination, source  
目标和源应具有相同的大小

## 寄存器寻址方式

数据本身就在寄存器中

从寄存器的值赋给另外一个寄存器

E.g.

<b>MOV BX,DX</b>	;copy the contents of DX into BX
<b>MOV ES,AX</b>	;copy the contents of AX into ES

第二条指令给ES/DS赋值

数据可以在CS, IP以外的所有寄存器之间通过move指令赋值

## immediate直接寻址方式

- 源操作数是一个常量
  - 嵌入到指令中
  - 无需访问内存

例如

<b>MOV AX,2550H</b>	;move 2550H into AX
<b>MOV CX,625</b>	;load the decimal value 625 into CX
<b>MOV BL,40H</b>	;load 40H into BL

第一条指令从16位赋值到16位

第三条指令从8位赋值到8位

数不能直接移到段寄存器

必须先放到通用寄存器中再赋值

MOV AX, 2550H  
MOV DS, AX ✓

## 直接寻址方式

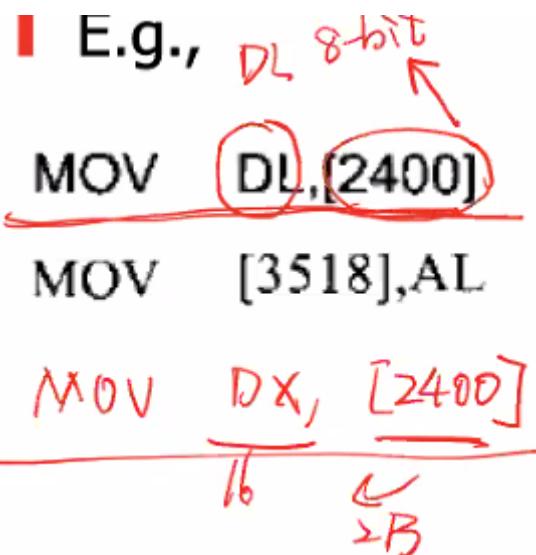
- 数据存储在内存中，地址在指令中给出
  - 默认情况下，数据段 (DS) 中的偏移地址
  - 需要访问内存以获取数据

例如

<b>MOV DL,[2400]</b>	;move contents of DS:2400H into DL
<b>MOV [3518],AL</b>	

[2400]=DS:[2400]

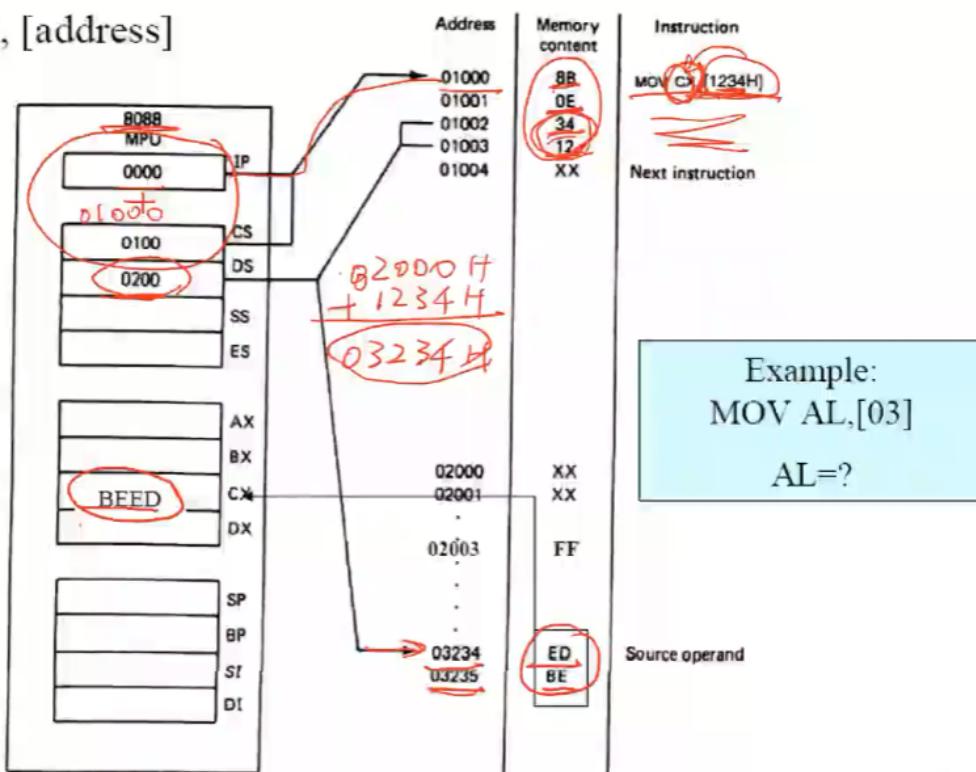
8位和16位不同



又例如

## Direct Addressing Mode

MOV CX, [address]



### 寄存器间接寻址方式

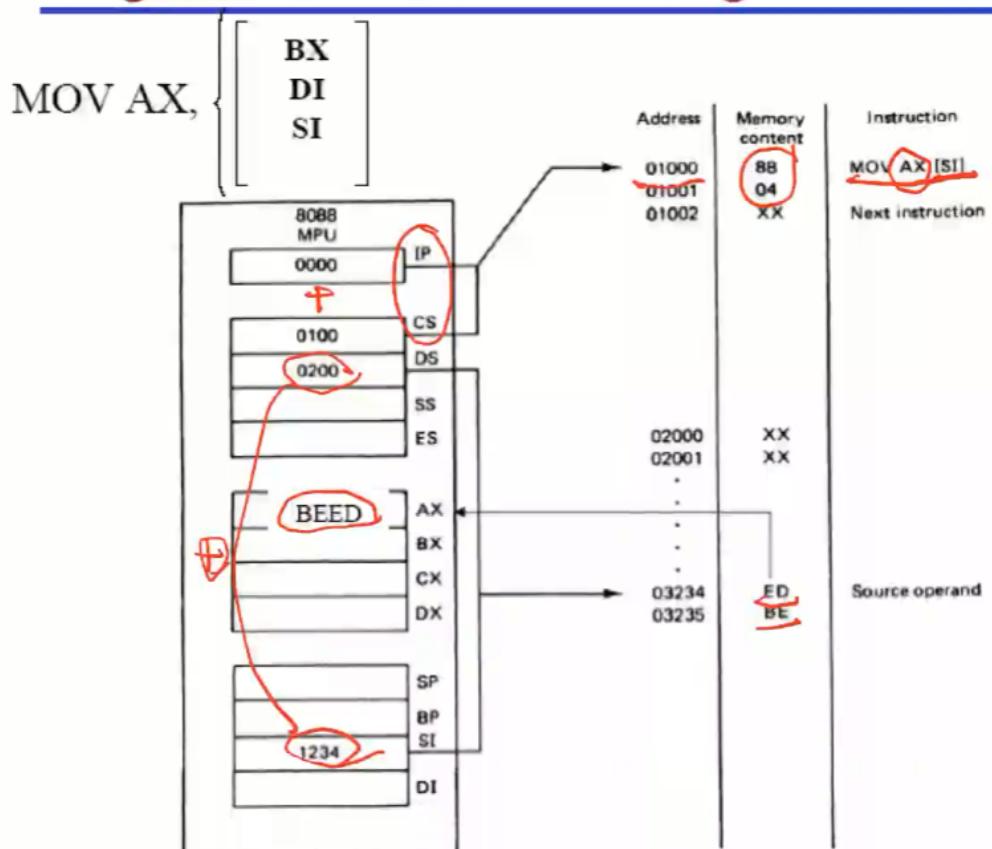
数据存储在存储器中，地址由寄存器保存，而直接寻址方式中地址直接给出

- 默认情况下，数据段（DS）中的偏移地址
- 用于此目的的寄存器是SI, DI和BX
- 需要访问内存以获取数据

例如

MOV AL,[BX] ;moves into AL the contents of the memory location  
;pointed to by DS:BX.

# Register Indirect Addressing Mode



## base relative addressing mode

数据存储在存储器中，并且可以通过基址寄存器BX和BP以及位移值计算地址

- 对于BX，默认段是数据段（DS），对于BP，默认段是堆栈段（SS）
- 需要访问内存以获取数据

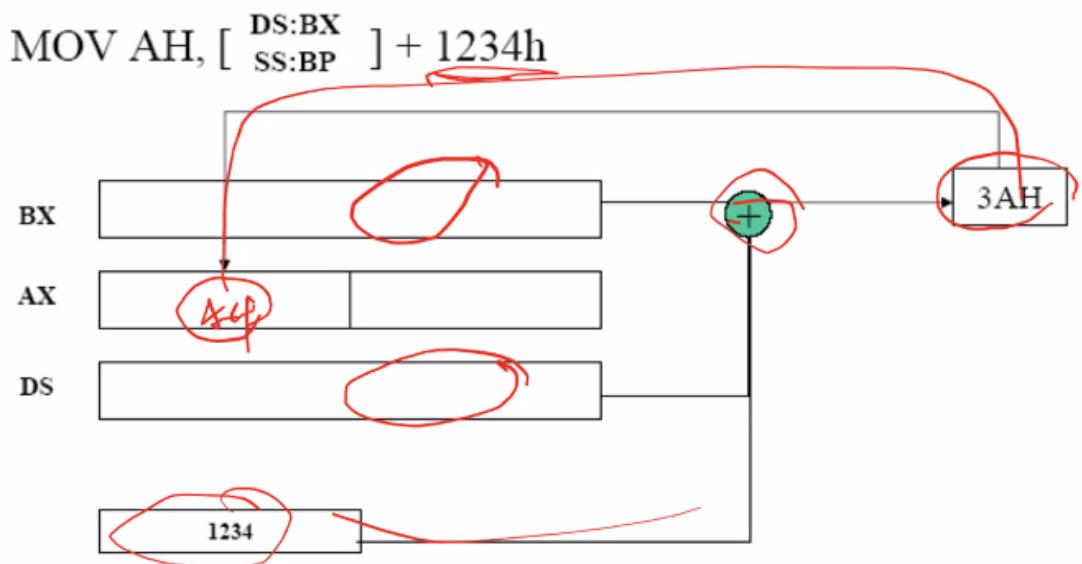
例如

`MOV CX,[BX]+10 ;move DS:BX+10 and DS:BX+10+1 into CX  
;PA = DS (shifted left) + BX + 10`

`MOV AL,[BP]+5 ;PA = SS (shifted left) + BP + 5`

第一条是DS:[BX]+10

第二条是SS:[BP]+5



### Indexed Relative Addressing Mode

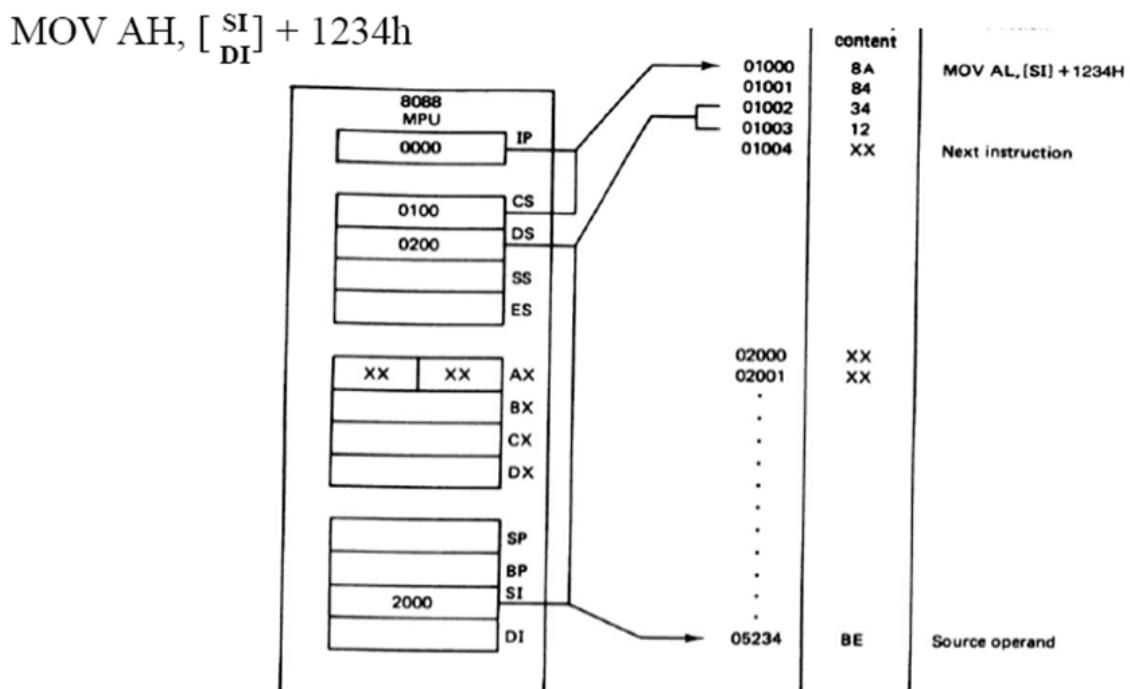
数据存储在存储器中，并且可以使用索引寄存器DI和SI以及位移值来计算地址

- 默认段是数据段 (DS)
- 需要访问内存以获取数据

例如

`MOV DX,[SI]+5` ;PA = DS (shifted left) + SI + 5  
`MOV CL,[DI]+20` ;PA = DS (shifted left) + DI + 20

### Indexed Relative Addressing Mode



Example: What is the physical address MOV [DI-8],BL if DS=200 & DI=30h ?  
 $DS:200 \text{ shift left once } 2000 + DI + -8 = 2028$

7

### Based Indexed Relative Addressing Mode

结合了基址和索引寻址模式，使用了一个基址寄存器和一个索引寄存器

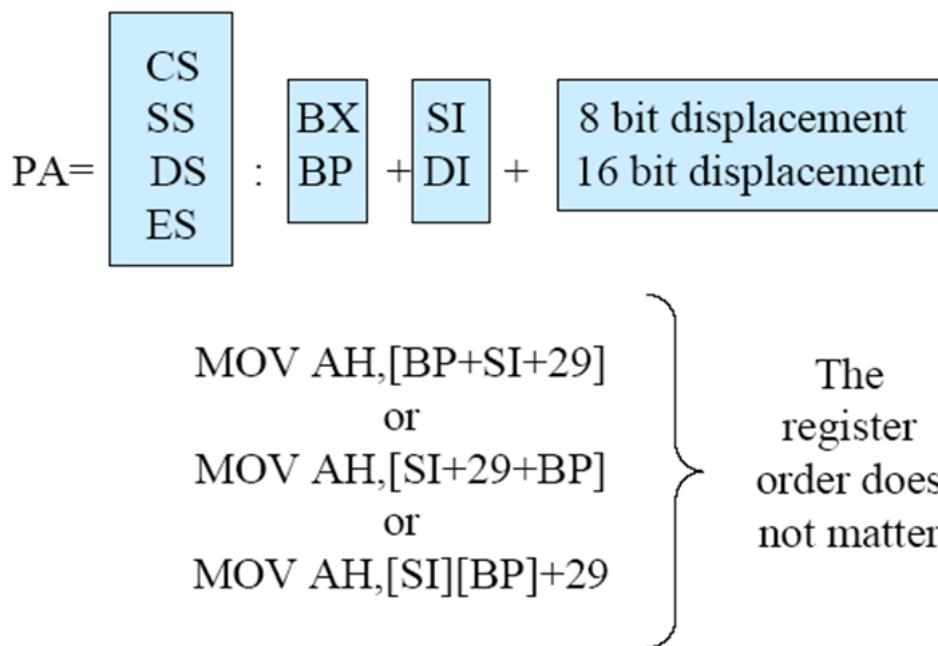
- 对于BX，默认段是数据段 (DS) ，对于BP，默认段是堆栈段 (SS)
- 需要访问内存以获取数据

例如

MOV CL,[BX][DI]+8	;PA = DS (shifted left) + BX + DI + 8
MOV CH,[BX][SI]+20	;PA = DS (shifted left) + BX + SI + 20
MOV AH,[BP][DI]+12	;PA = SS (shifted left) + BP + DI + 12
MOV AH,[BP][SI]+29	;PA = SS (shifted left) + BP + SI + 29

## Based-Indexed Relative Addressing Mode

- Based Relative + Indexed Relative
- We must calculate the PA (physical address)



### segment overrides 覆写

- 偏移寄存器与默认段寄存器一起使用

Table 1-3: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

- 80X86允许程序覆盖默认段寄存器
  - 在代码中指定段寄存器

Instruction	Segment Used	Default Segment
MOV AX,CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

# Ch 04 汇编语言

---

## 编程语言

---

- 机器语言
  - 二进制代码，适用于CPU，不适用于人类
- 汇编语言
  - 机器代码指令的助记符
  - 低级语言：处理CPU的内部结构
  - 难以编程，可移植性差但效率很高
- BASIC, Pascal, C, Fortran, Perl, TCL, Python等
  - 高级语言：不必关心CPU的内部细节
  - 易于编程，良好的可移植性但效率较低

## 汇编语言程序

---

一系列声明

- 汇编语言指令
  - 执行程序的实际工作
- Directives (伪指令)
  - 提供有关如何将程序转换为机器代码的汇编程序说明。

由多个段组成

- 但是CPU只能访问一个数据段，一个代码段，一个堆栈段和一个额外的段（为什么？）

## 声明形式

[label:] mnemonic [operands] [:comment]

- 标签是对此声明的引用  
名称规则：每个标签必须唯一；字母，0-9，（?），（.），（@），（\_）和（\$）；第一个字符不能为数字。且少于31个字符
- 如果是一条指令，则需要“：“，否则省略
- “；”引起注释，汇编程序在分号后省略此行的任何内容

## 真实程序的shell

完整的段定义（旧风格）

简化的段定义

```

;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
    .MODEL SMALL
    .STACK 64
    .DATA
DATA1    DB      52H
DATA2    DB      29H
SUM      DB      ?
.CODE
MAIN     PROC FAR      ;this is the program entry point
        MOV AX,@DATA   ;load the data segment address
        MOV DS,AX       ;assign value to DS
        MOV AL,DATA1    ;get the first operand
        MOV BL,DATA2    ;get the second operand
        ADD AL,BL       ;add the operands
        MOV SUM,AL      ;store the result in location SUM
        MOV AH,4CH      ;set up to return to DOS
        INT 21H         ;
MAIN     ENDP
END     MAIN      ;this is the program exit point

```

Figure 2-1. Simple Assembly Language Program

## model 定义

### ■ The MODEL directive

- Selects the size of the memory model
- SMALL: code <=64KB, data <=64KB
- MEDIUM: data <=64KB, code >64KB
- COMPACT: code<=64KB, data >64KB
- LARGE: data>64KB but single set of data<64KB, code>64KB
- HUGE: data>64KB, code>64KB
- TINY: code + data<64KB

```

;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
    .MODEL SMALL
    .STACK 64
    .DATA
DATA1    DB      52H
DATA2    DB      29H
SUM      DB      ?
.CODE
MAIN     PROC FAR      ;this is the program entry point
        MOV AX,@DATA   ;load the data segment address
        MOV DS,AX       ;assign value to DS
        MOV AL,DATA1    ;get the first operand
        MOV BL,DATA2    ;get the second operand
        ADD AL,BL       ;add the operands
        MOV SUM,AL      ;store the result in location SUM
        MOV AH,4CH      ;set up to return to DOS
        INT 21H         ;
MAIN     ENDP
END     MAIN      ;this is the program exit point

```

Figure 2-1. Simple Assembly Language Program

## Simplified Segment 定义

.CODE, .DATA, .STACK

只有3个段可被定义

自动对应到 CPU's CS, DS, SS

DOS determines the CS and SS segment registers automatically. DS (and ES) has to be manually specified.

## segment

- Stack segment
- Data segment
  - Data definition
- Code segment

- o Write your statements
- o Procedures definition  
label PROC [FAR|NEAR]  
label ENDP
- o Entrance proc should be FAR

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
.MODEL SMALL
.STACK 64
.DATA
DATA1    DB  52H
DATA2    DB  29H
SUM      DB  ?
.CODE
MAIN     PROC FAR      ;this is the program entry point
        MOV AX,@DATA   ;load the data segment address
        MOV DS,AX      ;assign value to DS
        MOV AL,DATA1    ;get the first operand
        MOV BL,DATA2    ;get the second operand
        ADD AL,BL      ;add the operands
        MOV SUM,AL      ;store the result in location SUM
        MOV AH,4CH      ;set up to return to DOS
        INT 21H         ;
MAIN     ENDP
END MAIN           ;this is the program exit point
```

Figure 2-1. Simple Assembly Language Program

## Full Segment 定义

Full segment definition

```
label SEGMENT
label ENDS
```

- 可以自由命名label
- as many as needed
- DOS assigns CS, SS
- Program assigns DS (manually load data segments) and ES

```

DaSeg1 segment
    str1 db 'Hello World! $'
DaSeg1 ends

StSeg segment
    dw 128 dup(0)
StSeg ends

CoSeg segment
    start proc far
        assume cs:CoSeg, ss:StSeg

        mov ax, DaSeg1      ; set segment registers:
        mov ds, ax
        mov es, ax

        call subr          ;call subroutine

        mov ah, 1           ; wait for any key....
        int 21h

        mov ah, 4ch         ; exit to operating system.
        int 21h
    start endp

    subr proc

        mov dx, offset str1
        mov ah, 9
        int 21h            ; output string at ds:dx

        ret
    subr endp

CoSeg ends

    end start    ; set entry point and stop the assembler.

```

## 程序执行

Program starts from the entrance

- Ends whenever calls 21H interruption with AH = 4CH(mov ah,4ch)

Procedure caller and callee

- CALL procedure 函数调用 例子call subr
- RET

int 21h软中断

## 控制转移说明

- 范围
  - **SHORT, 段内**
    - IP变更: 一节范围 (-128~127)
  - **NEAR, 段内**
    - IP已更改: 两节范围 (-32768~32767)
    - 如果控制权在同一代码段内转移
    - 可以跳至段内任何位置
  - **FAR, 段间**
    - CS和IP均已更改
    - 如果控制权转移到当前代码段之外

- jumps
- call statement

## 有条件跳转

- 根据标志寄存器的值跳转
- short jumps

Mnemonic	Condition Tested	“Jump IF ...”
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

## 无条件跳转

- JMP [SHORT | NEAR | FAR PTR] *label*
- 默认为NEAR

## 子程序和CALL语句

- 范围
  - **NEAR**: 过程与调用方在同一代码段中定义
  - **FAR**: 过程在调用方的当前代码段之外定义
- **PROC**和**ENDP**用于定义子程序
- **CALL**用于调用子例程
- - **RET**放在子程序的末尾
  - *far call*和*near call*之间的区别?

## Calling a NEAR proc

- ✓ The CALL instruction and the subroutine it calls are in the same segment.
- ✓ Save the current value of the IP on the stack.
- ✓ load the subroutine's offset into IP (nextinst + offset)

Calling Program	Subroutine	Stack						
Main proc 001A: call sub1 001D: inc ax . . . Main endp	sub1 proc 0080: mov ax,1 ... ret sub1 endp	<table border="1"><tr><td>1ffd</td><td>1D</td></tr><tr><td>1ffe</td><td>00</td></tr><tr><td>1fff</td><td>(not used)</td></tr></table>	1ffd	1D	1ffe	00	1fff	(not used)
1ffd	1D							
1ffe	00							
1fff	(not used)							

## Calling a FAR proc

- ✓ The CALL instruction and the subroutine it calls are in the “Different” segments.
- ✓ Save the current value of the CS and IP on the stack.
- ✓ Then load the subroutine’s CS and offset into IP.

Calling Program	Subroutine	Stack															
Main proc 1FCB:001A: call far ptr sub1 1FCB:001F: inc ax . . . Main endp	sub1 proc far 4EFA:0080: mov ax,1 .... .... ret (retf opcode generated) sub1 endp	<table border="1"><tr><td>1ffb</td><td>1F</td><td>I</td></tr><tr><td>1ffc</td><td>00</td><td>P</td></tr><tr><td>1ffd</td><td>CB</td><td>S</td></tr><tr><td>1ffe</td><td>1F</td><td>E</td></tr><tr><td>1fff</td><td>N/A</td><td>G</td></tr></table>	1ffb	1F	I	1ffc	00	P	1ffd	CB	S	1ffe	1F	E	1fff	N/A	G
1ffb	1F	I															
1ffc	00	P															
1ffd	CB	S															
1ffe	1F	E															
1fff	N/A	G															

34

## 数据类型和定义

- CPU可以处理8位或16位运算
- ◦ 如果数据更大，该怎么办？
- 伪指令
- ◦ ORG: origin 表示偏移地址的开头
  - 例如，ORG 10H

- 定义变量:
- ■ **DB**: 分配字节大小的块
  - ■ 例如, **x DB 12 | y DB 23H, 48H | Z DB "good morning" | str DB "im good"**
  - **DW, DD, DQ** (访问尺寸word\double\quarter)
- **EQU**: 定义一个常量
- ■ 例如, **NUM EQU 234**相当于#define NUM 234
- **DUP**: 复制给定数量的字符
- ■ 例如, **x DB 6 DUP (23H) | y DW 3 DUP (0FF10H)** 不能写成FF10H, 会被当成label

## 有关变量的更多信息

- 对于变量, 它们可能具有名称
- ◦ 例如, **luckyNum DB 27H, time DW 0FFFFH**
- 变量名称具有三个属性:
- ◦ **段值**
- **offset地址**
- **类型**: 如何访问变量 (例如, DB是按字节的方式, DW是按字的方式)
- 获取变量的段值
- ◦ 使用**SEG**指令 (例如, **MOV AX, SEG luckyNum**)
- 获取变量的偏移地址
- ◦ 使用**OFFSET**指令或**LEA**指令
  - 例如, **MOV AX, offset time**或**LEA AX, time**

## 有关标签的更多信息

- 标签定义:
- ◦ 隐式地:
  - ■ 例如, **AGAIN: ADD AX, 03423H**
  - 使用**LABEL**伪指令:
    - ■ 例如, **AGAIN LABEL FAR**
    - **ADD AX, 03423H**
- 标签具有三个属性:
- ◦ **段值**: 逻辑地址
- **偏移地址**: 逻辑地址
- **类型**: range for jumps, near, far

## 有关PTR指令的更多信息

- 临时更改变量(标签)的类型(范围)属性
- ◦ 为了保证指令中的两个操作数都匹配
  - 为了确保跳转可以到达标签
- 例如, **DATA1 DB 10H, 20H, 30H;**
- **DATA2 DW 4023H, 0A845H**
- ....
- **MOV BX, WORD PTR DATA1; 2010H-> BX**
- **MOV AL, BYTE PTR DATA2; 23H-> AL**

- MOV WORD PTR [BX], 10H; [BX], [BX + 1]←0010H
- 例如, JMP FAR PTR *aLabel*

## .COM可执行文件

- 总共一个segment
- - 将数据和代码放在一起
  - 小于64KB

```
TITLE PROG2-4 COM PROGRAM TO ADD TWO WORDS
PAGE 60,132
CODSG SEGMENT
ORG 100H
ASSUME CS:CODSG,DS:CODSG,ES:CODSG
;——THIS IS THE CODE AREA
PROGCODE PROC NEAR
MOV AX,DATA1 ;move the first word into AX
MOV SUM,AX ;move the sum
MOV AH,4CH ;return to DOS
INT 21H
PROGCODE ENDP
;——THIS IS THE DATA AREA
DATA1 DW 2390
DATA2 DW 3456
SUM DW ?
;——
CODSG ENDS
END PROGCODE
```

```
TITLE PROG2-5 COM PROGRAM TO ADD TWO WORDS
PAGE 60,132
CODSG SEGMENT
ASSUME CS:CODSG,DS:CODSG,ES:CODSG
ORG 100H
START: JMP PROGCODE ;go around the data area
;——THIS IS THE DATA AREA
DATA1 DW 2390
DATA2 DW 3456
SUM DW ?
;——THIS IS THE CODE AREA
PROGCODE: MOV AX,DATA1 ;move the first word into AX
ADD AX,DATA1 ;add the second word
MOV SUM,AX ;move the sum
MOV AH,4CH
INT 21H
;——
CODSB ENDS
END START
```

第一种先return to DOS 后进行data area

第二种直接跳转至procedure, 不执行data area

# Ch 05 汇编语言

## 运算指令

- 加法
- 减法
- 乘法
- 除法

## 无符号加法

- ADD dest, src;  $dest = dest + src$
- - Dest可以是寄存器或内存中
  - Src可以是内存中的寄存器, 也可以是立即数
  - 80X86中没有内存到内存操作
  - 更改ZF, SF, AF, CF, OF, PF
- ADC dest, src;  $dest = dest + src + CF$
- - 对于多字节数字
  - 如果最后加法有进位, 则将结果加1

## 单字节加法示例

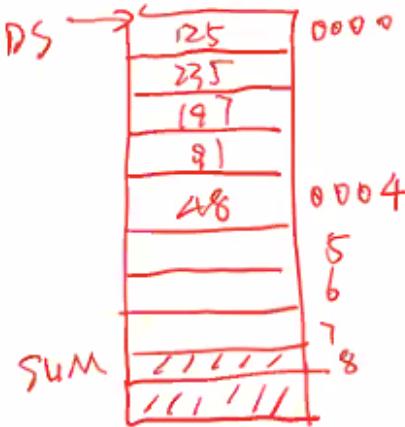
TITLE PROG3-1A (EXE) ADDING 5 BYTES

PAGE 60,132

MODEL SMALL

STACK 64

```
COUNT    .DATA  
EQU      05  
DATA   DB    125,235,197,91,48  
ORG     0008H  
SUM     DW    ?  
  
.CODE  
MAIN    PROC  FAR  
        MOV AX,@DATA  
        MOV DS,AX  
        MOV CX,COUNT ;CX is the loop counter  
        MOV SI,OFFSET DATA;SI is the data pointer  
        MOV AX,00 ;AX will hold the sum  
BACK:   ADD AL,[SI] ;add the next byte to AL  
        JNC OVER ;If no carry, continue  
        INC AH ;else accumulate carry in AH  
OVER:   INC SI ;increment data pointer  
        DEC CX ;decrement loop counter  
        JNZ BACK ;if not finished, go add next byte  
        MOV SUM,AX ;store sum  
        MOV AH,4CH  
        INT 21H ;go back to DOS  
MAIN    ENDP  
END     MAIN
```



@DATA 取ds段首

SI读取各个地址的数据

CX作为计数常量 DEC CX循环对CX减小

JNZ BACK 除非CX为0，否则回到BACK

多字节加法示例

```

TITLE      PROG3-2 (EXE) MULTIWORD ADDITION
PAGE       60,132
.MODEL SMALL
.STACK 64
;
; .DATA
DATA1    DQ      548FB9963CE7H
          ORG    0010H
DATA2    DQ      3FCD4FA23B8DH
          ORG    0020H
DATA3    DQ      ?
;
; .CODE
MAIN     PROC   FAR
          MOV    AX,@DATA
          MOV    DS,AX
          CLC
          MOV    SI,OFFSET DATA1
          MOV    DI,OFFSET DATA2
          MOV    BX,OFFSET DATA3
          MOV    CX,04
BACK:    MOV    AX,[SI]
          ADC    AX,[DI]
          MOV    [BX],AX
          INC    SI
          INC    DI
          INC    DI
          INC    BX
          INC    BX
          LOOP   BACK
          MOV    AH,4CH
          INT    21H
          ENDP
MAIN     END    MAIN
;
```

CLC清除进位 clear carry

SI/DI分别指向data1和data2首位

CX循环计数器

LOOP BACK=DEC CX JNZ BACK

## 无符号减法

- **SUB dest, src; dest = dest-src**
- - 目标可以是寄存器或内存中
  - Src可以是内存中的寄存器，也可以是立即数
  - 80X86中没有内存到内存操作
  - 更改ZF, SF, AF, CF, OF, PF
- **SBB dest, src; dest = dest-src - CF**
- - 对于多字节数字
  - 如果最后一次减法有借位，则从结果中减去1

## 单字节减法示例

- CPU执行
- - 取src的2的补码
  - 将其添加到目标
  - 反转进位

经过这三个步骤后，如果

- CF = 0: 正值;

MOV	AL,3FH	AL	3F	0011 1111	0011 1111	
MOV	BH,23H	- BH	- 23	- 0010 0011	+1101 1101	(2's complement)
SUB	AL,BH		1C		1 0001 1100	CF=0 (step 3)

- CF = 1: 负值, 保留2的补码
- ◦ 幅度: NOT + INC (如果需要幅度)

4C 0100 1100	0100 1100	
- 6E 0110 1110	2's comp	+1001 0010
- 22		0 1101 1110

CF=1 (step 3) the result is negative

## 多字节数字减法示例

```

DATA_A DD 62562FAH
DATA_B DD 412963BH
RESULT DD ?
...
    MOV AX,WORD PTR DATA_A
    SUB AX,WORD PTR DATA_B
    MOV WORD PTR RESULT+1,AX
    MOV AX,WORD PTR DATA_A+2
    SBB AX,WORD PTR DATA_B+2
    MOV WORD PTR RESULT+2,AX

```

$$AX = 62FA - 963B = CCBF \quad CF = 1$$

$$AX = 625 - 412 - 1 = 212. \quad CF = 0$$

RESULT is 0212CCBF.

## 无符号乘法

- **MUL** operand
- 字节 X 字节:
  - 一个隐式操作数是**AL**, 另一个是操作数, 结果存储在**AX**中
- 字 X 字:
  - 一个隐式操作数是**AX**, 另一个是操作数, 结果存储在**DX & AX**中
- 字 X 字节:
  - **AL**保留字节, **AH = 0**, 字为操作数, 结果存储在**DX & AX**中;

## 无符号乘法示例

---

```
MOV AL,DATA1  
MOV BL,DATA2  
MUL BL  
MOV RESULT,AX
```

```
MOV AL,DATA1  
MOV SI,OFFSET DATA2  
MUL BYTE PTR [SI]  
MOV RESULT,AX
```

DATA3	DW	2378H
DATA4	DW	2F79H
RESULT1	DW	2 DUP(?)

```
MOV AX,DATA3  
MUL DATA4  
MOV RESULT1,AX  
MOV RESULT1+2,DX
```

DATA5	DB	6BH
DATA6	DW	12C3H
RESULT3	DW	2 DUP(?)

```
MOV AL,DATA5  
SUB AH,AH  
MUL DATA6  
MOV BX,OFFSET RESULT3  
MOV [BX],AX  
MOV [BX]+2,DX
```

## 无符号除法

- DIV *denominator*
- ◦ 分母不能为零
- ◦ 对于分配的寄存器，商不能太大
- 字节/字节：
  - ◦ **AL中的分子**, AH清除; 商在AL, 余数在AH
- 字/字：
  - ◦ **AX中的分子**, 清除DX; 商在AX, 余数在DX
- 双字/词：
  - ◦ **AX中的分子**; 商在AL (最大0FFH), 余数在AH
- 分子在DX, AX; 商在AX (最大0FFFFH), 余数在DX
  - 分母可以在寄存器或内存中

## 无符号除法示例

```
MOV AL,DATA7  
SUB AH,AH  
DIV 10
```

```
MOV AX,10050  
SUB DX,DX  
MOV BX,100  
DIV BX  
MOV QOUT2,AX  
MOV REMAIND2,DX
```

```
MOV AX,2055  
MOV CL,100  
DIV CL  
MOV QUO,AL  
MOV REMI,AH
```

DATA1	DD	105432
DATA2	DW	10000
QUOT	DW	?
REMAIN	DW	?
MOV	AX,WORD PTR DATA1	
MOV	DX,WORD PTR DATA1+2	
DIV	DATA2	
MOV	QUOT,AX	
MOV	REMAIN,DX	

## 逻辑指令

- AND
- OR
- XOR
- NOT
- Logical SHIFT
- ROTATE
- COMPARE

### AND

- AND dest, src
  - 逐位逻辑
  - dest可以是寄存器，也可以是内存；src可以是寄存器，可以是内存中的，也可以是立即数

### OR

- OR dest, src
  - 逐位逻辑
  - dest可以是寄存器，也可以是内存；src可以是寄存器，可以是内存中的，也可以是立即数

### XOR

- XOR dest, src
  - 逐位逻辑
  - dest可以是寄存器，也可以是内存；src可以是寄存器，可以是内存中的，也可以是立即数

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

## NOT

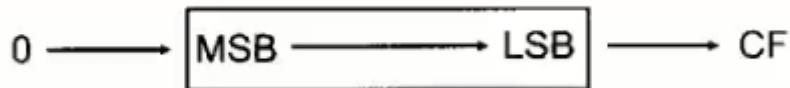
- 非操作数
- - 逐位逻辑
  - 操作数可以是寄存器或内存中

## Logical SHIFT

- SHR dest, times

  - shift right

    - 





  - dest可以是寄存器或内存中

  - 0-> MSB->...-> LSB-> CF

  - Times= 1:

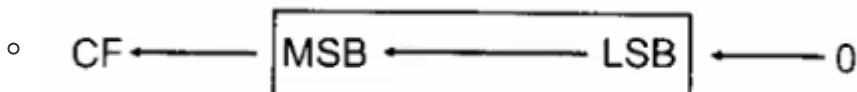
    - SHR xx, 1

  - Times> 1:

    - MOV CL, times

    - SHR xx, CL

- SHL dest, times

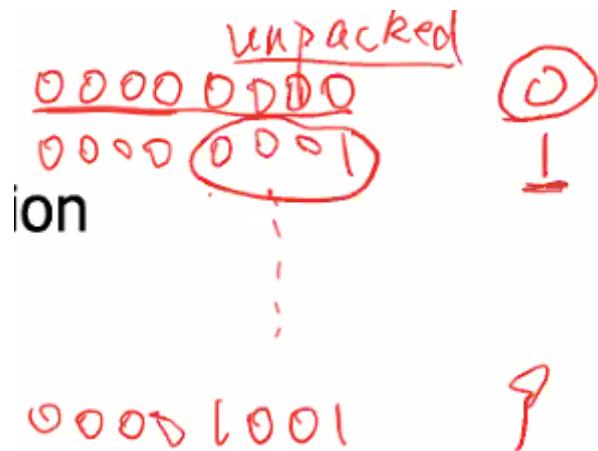




  - 除反方向外，其余均相同

## 示例：BCD和ASCII数字转换

- BCD: 二进制编码的十进制
- - 0~9二进制表示
  - 未压缩, 已压缩



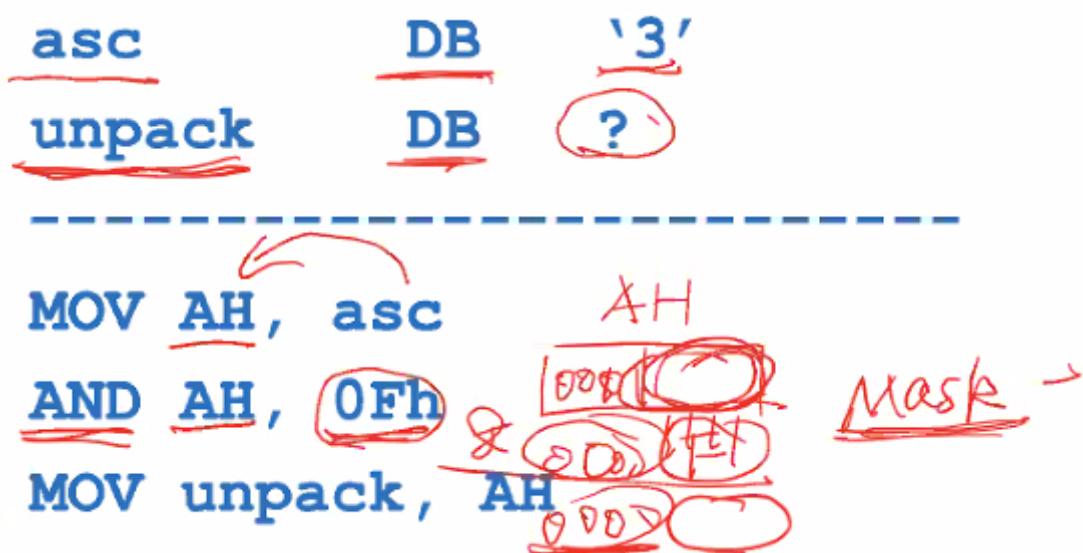
packed使用四位即可

- ASCII码
- - 可以读写的所有字符的代码
  - 数字字符'0'~'9'

<u>Key ASCII (hex)</u>	<u>Binary</u>	<u>BCD (unpacked)</u>
0 30	011 0000	0000 0000
1 31	011 0001	0000 0001
2 32	011 0010	0000 0010
3 33	011 0011	0000 0011
4 34	011 0100	0000 0100
5 35	011 0101	0000 0101
6 36	011 0110	0000 0110
7 37	011 0111	0000 0111
8 38	011 1000	0000 1000
9 39	011 1001	0000 1001

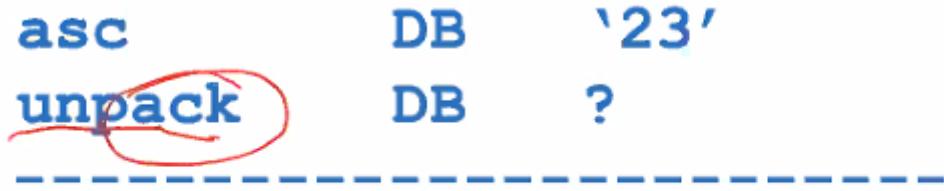
### ASCII->未压缩的BCD转换

- 只需删除较高的4位“0011”
- 例如，



### ASCII->压缩BCD转换

- 首先将ASCII转换为未压缩的BCD
- 然后，将两个未包装的包装合并为一个包装
- 例如



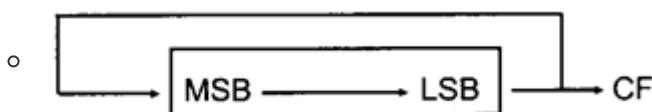
```

MOV AH, asc
MOV AL, asc+1
AND AX, 0F0Fh
MOV CL, 4
SHL AH, CL
OR AH, AL
MOV unpack, AH

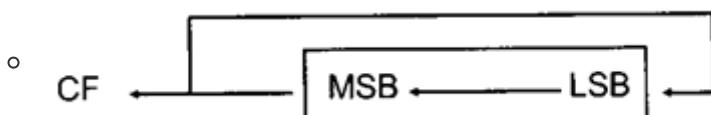
```

## ROTATE

- ROR dest, times**
- dest*可以是内存中的寄存器
  - times* = 1:
  - ROR xx, 1**



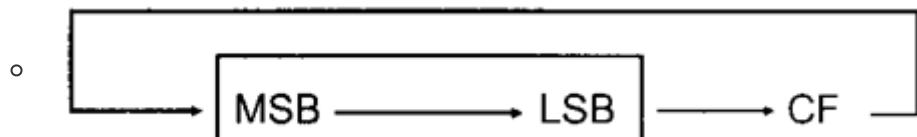
- times* > 1:
- MOV CL, times**
- ROR xx, CL**



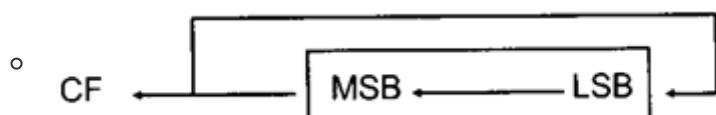
- ROL dest, times**
- 除反方向外，其余均相同

## ROTATE with carry

- RCR dest, times**
- dest*可以是内存中的寄存器
  - times* = 1:
  - RCR xx, 1**



- $times > 1$ :
- MCV CL, times
- RCR xx, CL



- RCL dest, times
- ○ 除反方向外，其余均相同

## 比较无符号数字

- CMP dest, src
- ○ 标志受影响为  $(dest - src)$ ，但操作数保持不变

**Table 3-3: Flag Settings for Compare Instruction**

Compare operands	CF	ZF
destination > source	0	0
destination = source	0	1
destination < source	1	0

- 例如，CMP AL, 23
- JA label1 ; jump if above, CF = ZF = 0

# Jump Based on Unsigned Comparison

These flags are based on unsigned comparison

Mnemonic	Description	Flags/Registers
JA	Jump if above op1>op2	CF = 0 and ZF = 0
JNBE	Jump if not below or equal op1 not <= op2	CF = 0 and ZF = 0
JAE	Jump if above or equal op1>=op2	CF = 0
JNB	Jump if not below op1 not < op2	CF = 0
JB	Jump if below op1<op2	CF = 1
JNAE	Jump if not above nor equal op1< op2	CF = 1
JBE	Jump if below or equal op1 <= op2	CF = 1 or ZF = 1
JNA	Jump if not above op1 <= op2	CF = 1 or ZF = 1

## 比较有符号数

- CMP dest, src
- ◦ 与无符号案例相同的指令
- 但是对数字的理解不同，因此检查了不同的标志

**destination > source**  
**destination = source**  
**destination < source**

OF=SF or ZF=0  
ZF=1  
OF=negation of SF

These flags are based on signed comparison

Mnemonic	Description	Flags/Registers
JG JA	Jump if GREATER op1>op2	SF = OF AND ZF = 0
JNLE	Jump if not LESS THAN or equal op1>=op2	SF = OF AND ZF = 0
JGE	Jump if GREATER THAN or equal op1>=op2	SF = OF
JNL	Jump if not LESS THAN op1>op2	SF = OF
JL	Jump if LESS THAN op1<op2	SF <> OF
JNGE	Jump if not GREATER THAN nor equal op1<op2	SF <> OF
JLE	Jump if LESS THAN or equal op1 <= op2	ZF = 1 OR SF <> OF
JNG	Jump if NOT GREATER THAN op1 <= op2	ZF = 1 OR SF <> OF

## QUIZ

Given the ASCII table, write an algorithm to convert lowercase letters in a string into uppercase letters and implement your algorithm using 86 assembly language.

ASCII value	Character	Control character	ASCII value	Character	ASCII value	Character	ASCII value	Character
000	(null)	NUL	032	(space)	064	@	096	
001	⌚	SOH	033	!	065	A	097	🔤
002	⌚	STX	034	"	066	B	098	🔤
003	♥	ETX	035	#	067	C	099	🔤
004	♦	EOT	036	\$	068	D	100	🔤
005	♣	ENQ	037	%	069	E	101	🔤
006	♠	ACK	038	&	070	F	102	🔤
007	(beep)	BEL	039	,	071	G	103	🔤
008	█	BS	040	(	072	H	104	🔤
009	(tab)	HT	041	)	073	I	105	🔤
010	(line feed)	LF	042	*	074	J	106	🔤
011	(home)	VT	043	+	075	K	107	🔤
012	(form feed)	FF	044	.	076	L	108	🔤
013	(carriage return)	CR	045	-	077	M	109	🔤
014	☒	SO	046	.	078	N	110	🔤
015	⌚	SI	047	/	079	O	111	🔤
016	◀	DLE	048	0	080	P	112	🔤
017	▶	DC1	049	1	081	Q	113	🔤
018	↑	DC2	050	2	082	R	114	🔤
019	!!	DC3	051	3	083	S	115	🔤
020	π	DC4	052	4	084	T	116	🔤
021	§	NAK	053	5	085	U	117	🔤
022	---	SYN	054	6	086	V	118	🔤
023	↑	ETB	055	7	087	W	119	🔤
024	↑↓	CAN	056	8	088	X	120	🔤
025	↓	EM	057	9	089	Y	121	🔤
026	→	SUB	058	:	090	Z	122	🔤
027	←	ESC	059	:	091	[	123	{
028	(cursor right)	FS	060	<	092	\	124	:
029	(cursor left)	GS	061	=	093	]	125	}
030	(cursor up)	RS	062	>	094	^	126	~
031	(cursor down)	US	063	?	095	-	127	□

Copyright 1998, JimPrice.Com Copyright 1992, Leading Edge Computer Products, Inc.

.MODEL SMALL  
.STACK 64

```

        .DATA
DATA1    DB      'mY NAME is jOe'
        ORG    0020H
DATA2    DB      14 DUP(?)
;

        .CODE
MAIN     PROC   FAR
        MOV    AX,@DATA
        MOV    DS,AX
        MOV    SI,OFFSET DATA1      ;SI points to original data
        MOV    BX,OFFSET DATA2      ;BX points to uppercase data
        MOV    CX,14                ;CX is loop counter
BACK:    MOV    AL,[SI]           ;get next character
        CMP    AL,61H              ;if less than 'a'
        JB     OVER               ;then no need to convert
        CMP    AL,7AH              ;if greater than 'z'
        JA     OVER               ;then no need to convert
OVER:    AND    AL,11011111B    ;mask d5 to convert to uppercase
        MOV    [BX],AL             ;store uppercase character
        INC    SI                  ;increment pointer to original
        INC    BX                  ;increment pointer to uppercase data
        LOOP   BACK               ;continue looping if CX > 0
        MOV    AH,4CH
        INT    21H                 ;go back to DOS
MAIN    ENDP
END    MAIN

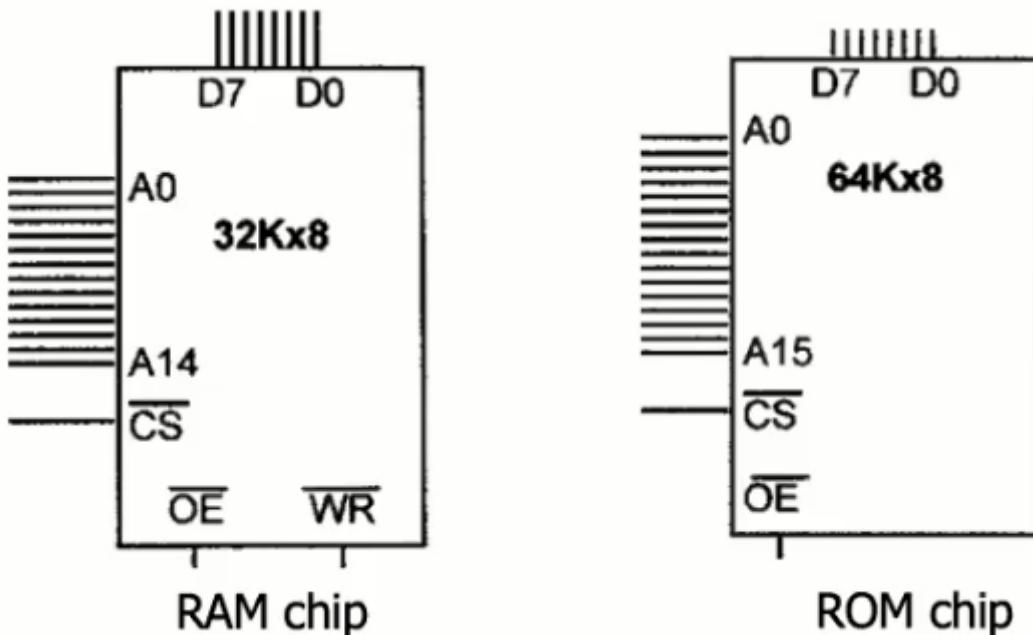
```

# Ch06 Lecture 06: Memory Address Decoding

## 回顾存储芯片

- 关键概念：容量，组织（位置数X可寻址单元的大小），访问时间
- 打包

RAM芯片&ROM芯片



CS/OE片选

RAM比ROM多WR信号，因为ROM只读

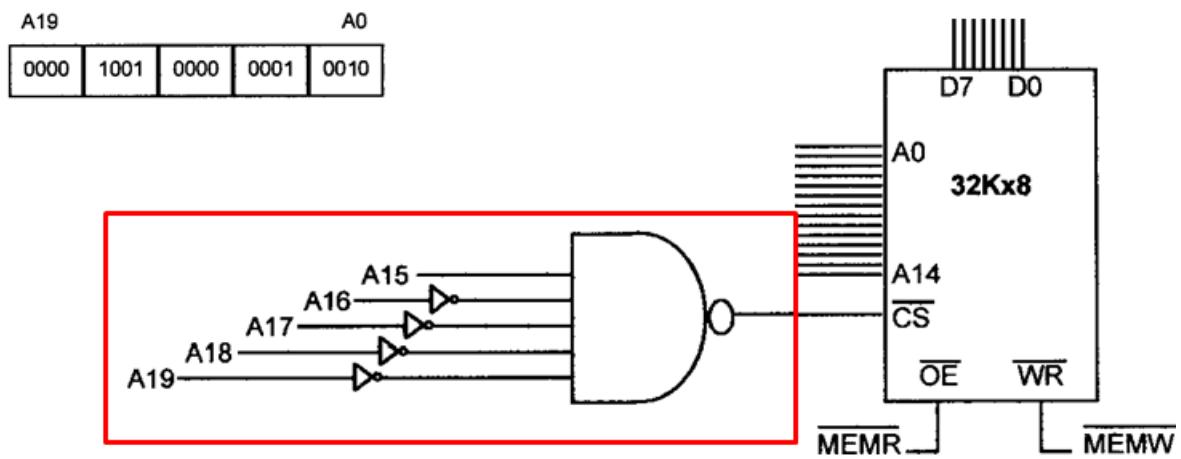
## 要学什么？

- 8086 CPU如何执行MOV BX [1000h]之类的指令？
- 执行MOV BX [1000h]与执行MOV [1001h] BX有什么区别？
- 访问内存和访问I / O设备有什么区别？

## 内存地址解码

- 根据你的指令访问内存
  - 例如，MOV AX, [0012H]
- CPU计算操作数的物理地址，并将相应的信号放在地址总线上
  - 例如，如果DS = 0900H, PA是多少? 09000+0012=09012H
- 内存地址解码电路可找到存储所需数据的特定内存芯片
  - 使用逻辑门和38译码器芯片检查地址解码

PA 0900: 0012 = 09012H



A19	0000	1000	0000	0000	0000	A0
-----	------	------	------	------	------	----

=08000H

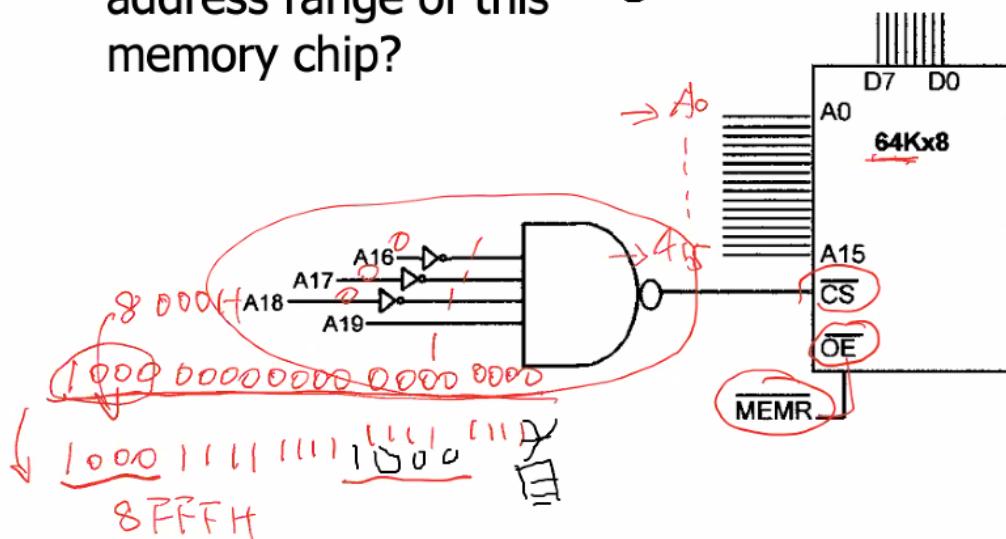
0000	1111	1111	1111	1111	1111	A0
------	------	------	------	------	------	----

=FFFFFH

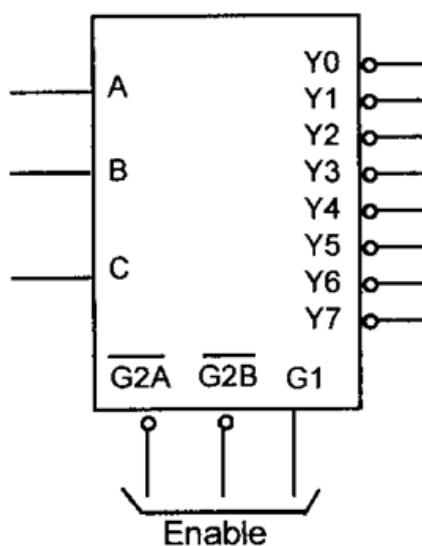
红圈为了使得满足00001时才可开启片选

## QUIZ

What's the type and the address range of this memory chip?



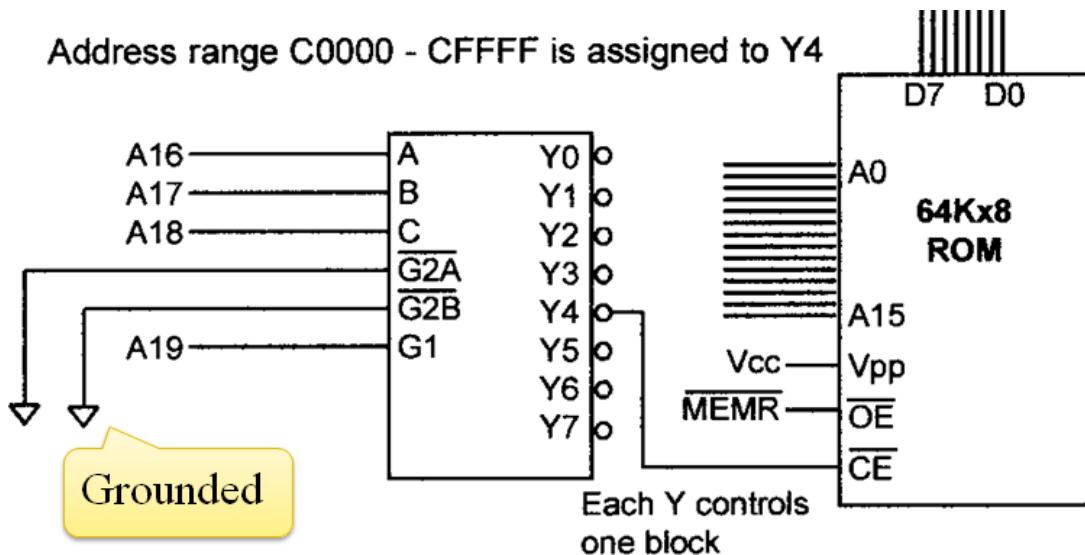
The 74LS138 Decoder Chip



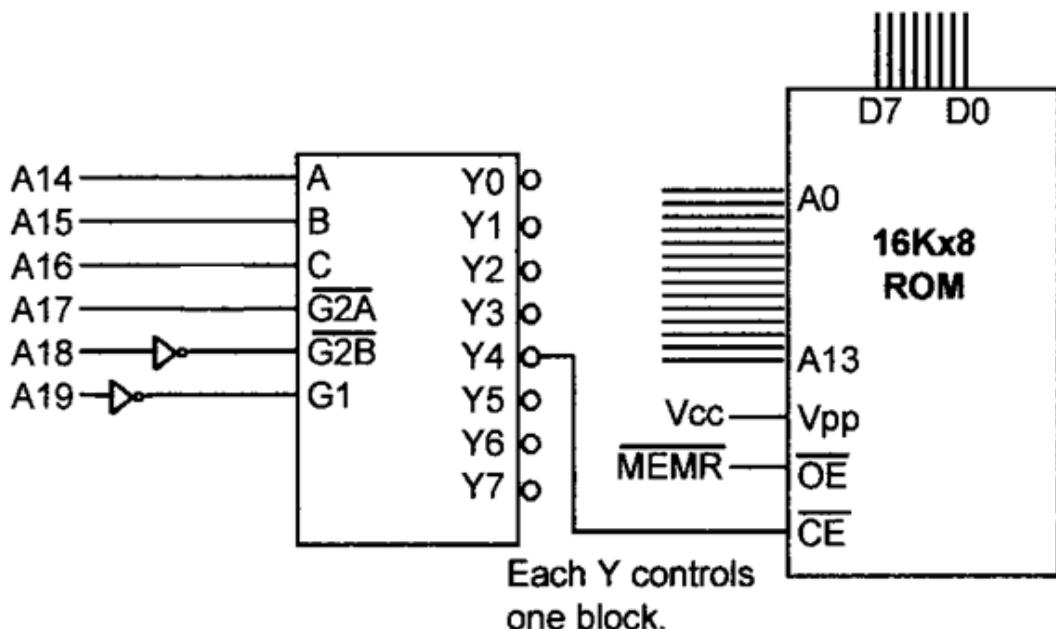
Inputs			Outputs							
Enable	Select		Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
G1G2	C	B	A							
X	H		XXX				H	H	H	H
L	X		XXX				H	H	H	H
H	L		LLL				L	H	H	H
H	L		LLH				H	L	H	H
H	L		LHL				H	H	L	H
H	L		LHH				H	H	H	L
H	L		HLL				H	H	H	L
H	L		HLH				H	H	H	H
H	L		LLL				H	H	H	H
H	L		HHH				H	H	H	H

## Using 74LS138 to Decode

Address range C0000 - CFFFF is assigned to Y4



## QUIZ



## 有关地址解码的更多信息

- 绝对地址解码
  - 所有地址线都参与译码
- 线性选择解码
  - 部分线参与译码
  - 廉价
  - 但是具有别名：具有多个地址的同一存储单元（I/O端口）
  - ■ 为什么会这样？
    - 例如用A0-A17进行译码，空闲A18和A19，则相同地址有4个别名，因为A18-A19可以任意选取

### 示例1.

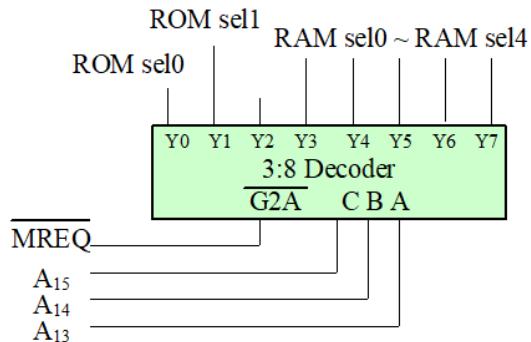
在特定的计算机中，ROM使用的地址范围是0000h至3FFFh(16K)，保留的地址范围是4000h至5FFFh(8K)供将来使用，而RAM的范围是6000h至0FFFFh(40K)。假设RAM的控制信号是CS~和WE~，并且CPU有16个地址引脚（即A15~A0），8个数据引脚（即D7~D0）以及R/W~和MREQ~控制信号。实现以下要求。

#### (1) 使用38译码器绘制地址解码解决方案

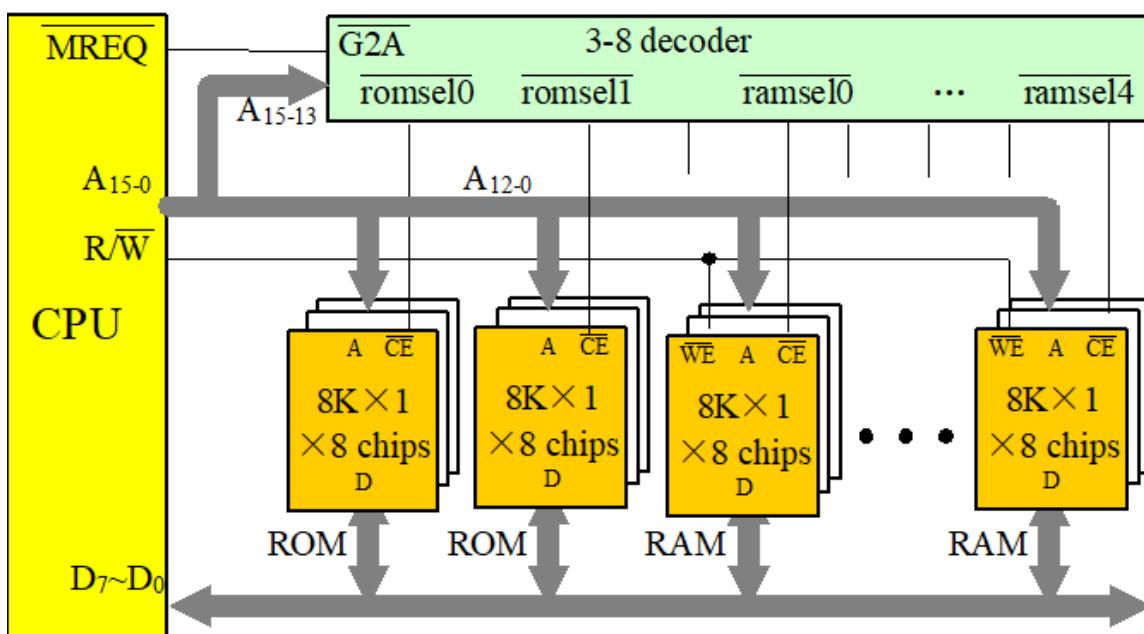
利用A15-A13当做输入，选择总共64K中的8K片区

The logic expression of each output of the 74LS138 chip:

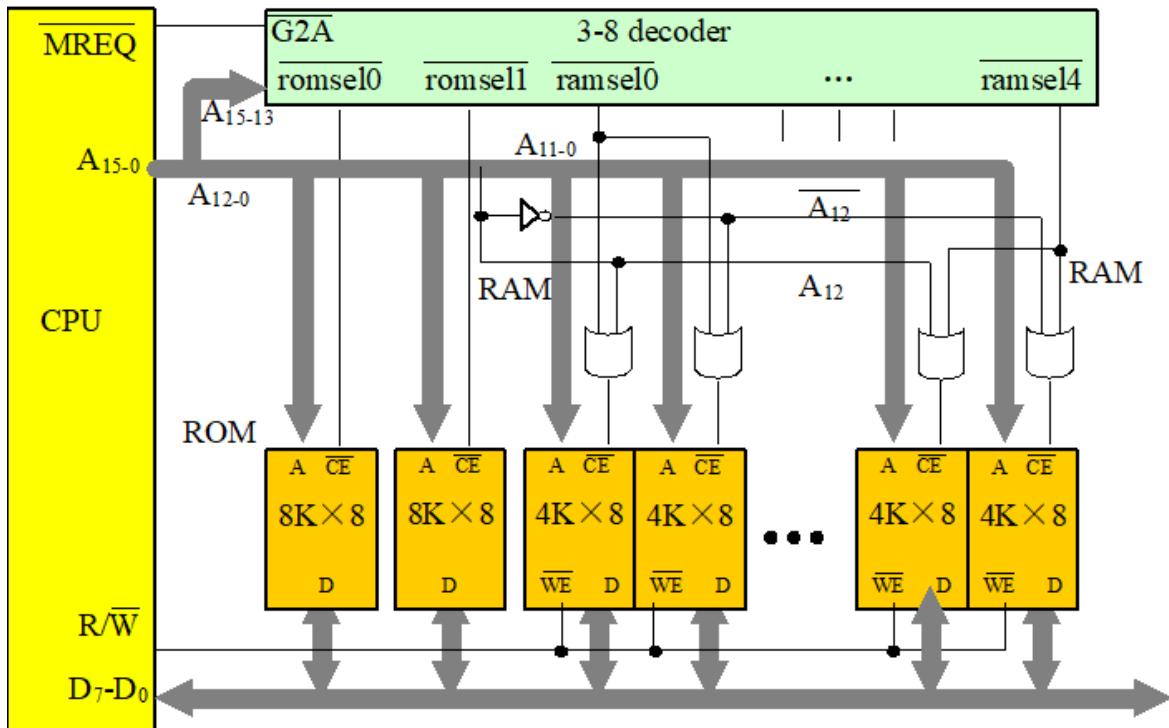
$$\begin{aligned}
 \text{romsel0} &= \overline{\text{A15}} * \overline{\text{A14}} * \overline{\text{A13}} * \overline{\text{MREQ}\#} \\
 \text{romsel1} &= \overline{\text{A15}} * \overline{\text{A14}} * \overline{\text{A13}} * \overline{\text{MREQ}\#} \\
 \text{ramsel0} &= \overline{\text{A15}} * \overline{\text{A14}} * \overline{\text{A13}} * \overline{\text{MREQ}\#} \\
 \text{ramsel1} &= \text{A15} * \overline{\text{A14}} * \overline{\text{A13}} * \overline{\text{MREQ}\#} \\
 \text{ramsel2} &= \text{A15} * \overline{\text{A14}} * \overline{\text{A13}} * \overline{\text{MREQ}\#} \\
 \text{ramsel3} &= \text{A15} * \overline{\text{A14}} * \overline{\text{A13}} * \overline{\text{MREQ}\#} \\
 \text{ramsel4} &= \text{A15} * \overline{\text{A14}} * \overline{\text{A13}} * \overline{\text{MREQ}\#}
 \end{aligned}$$



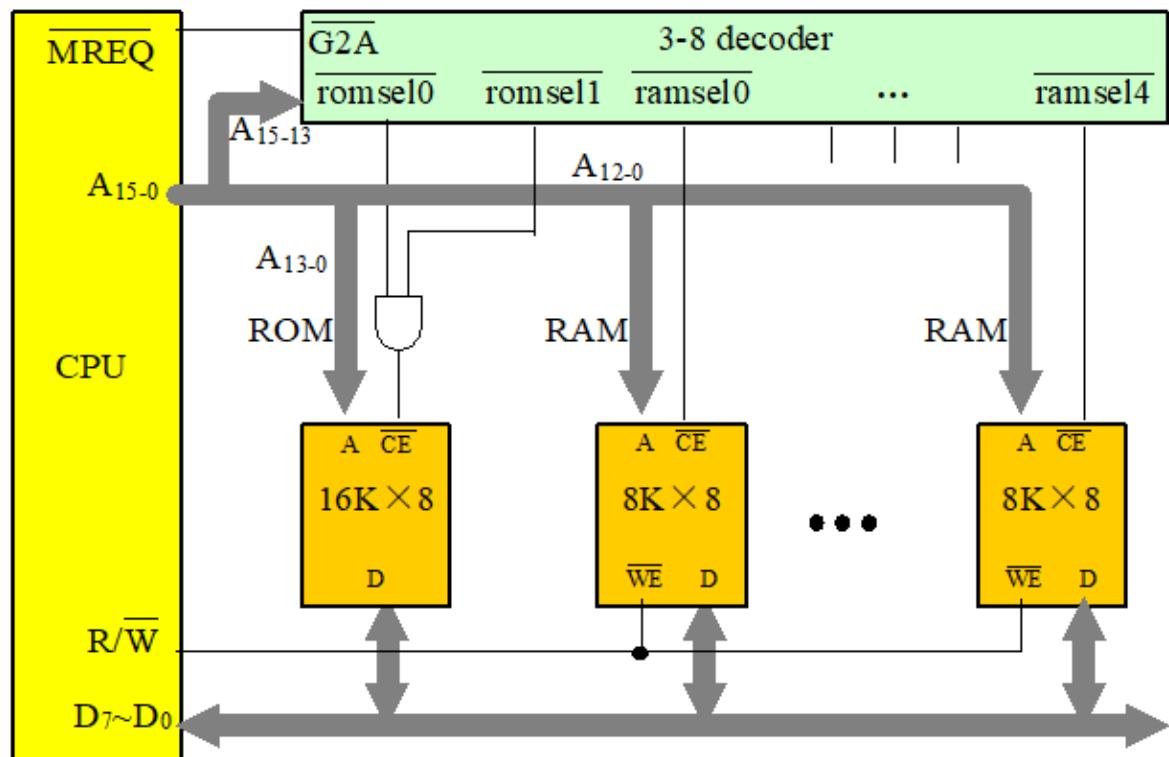
#### (2) 如果ROM和RAM均内置8K×1内存芯片，请尝试绘制CPU与内存之间的连接。



(3) 如果ROM是由 $8K \times 8$ 的内存芯片构建的，而RAM是由 $4K \times 8$ 的内存芯片构建的，则尝试绘制CPU与内存之间的连接。



(4) 如果ROM是由 $16K \times 8$ 存储芯片构建的，而RAM是由 $8K \times 8$ 存储芯片构建的，那该怎么办？



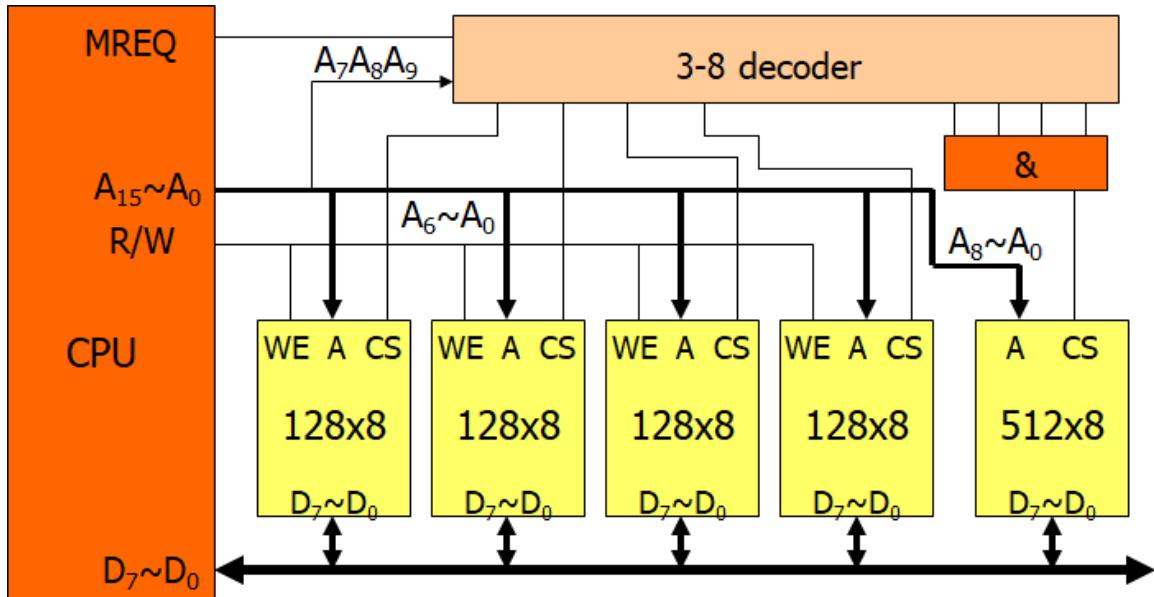
## 示例2.

假设一个计算机系统需要512字节RAM和512字节ROM。如果RAM由 $128 \times 8$ 存储芯片构成，而ROM由 $512 \times 8$ 存储芯片构成，请指定每个存储芯片的地址范围。考虑到RAM芯片需要 $CS \sim$ 和 $WE \sim$ 控制信号，ROM芯片只需要 $CS \sim$ 控制信号，并且CPU有16个地址引脚 ( $A_{15} \sim A_0$ )，8个数据引脚 ( $D_7 \sim D_0$ ) 以及 $R/W \sim$ 和 $MREQ \sim$ 控制信号，绘制CPU与内存之间的连接。

每个存储芯片的地址范围：

Memory Chip	Address range (hex)	binary
RAM1	0000~007F	0 0 0 x x x x x x x x
RAM2	0080~00FF	0 0 1 x x x x x x x x
RAM3	0100~017F	0 1 0 x x x x x x x x
RAM4	0180~01FF	0 1 1 x x x x x x x x
ROM	0200~03FF	1 x x x x x x x x x x

由于存储器的总容量为1K，因此我们需要10条地址线。RAM芯片需要7条地址线，ROM芯片需要9条地址线。



## 数据的完整性

- ROM的Checksum byte
- DRAM的奇偶校验位
- 磁盘和Internet的CRC校验

## Checksum byte

- 检查一系列字节的完整性
- - 计算方式
    - 将所有字节加在一起并丢弃所有进位
    - 取总和的2的补码
  - 将校验和字节与数据一起存储
  - 通过将数据和校验和相加来检查完整性
  - 例如38H, 23H, 33H, 07H, 校验和字节是6BH

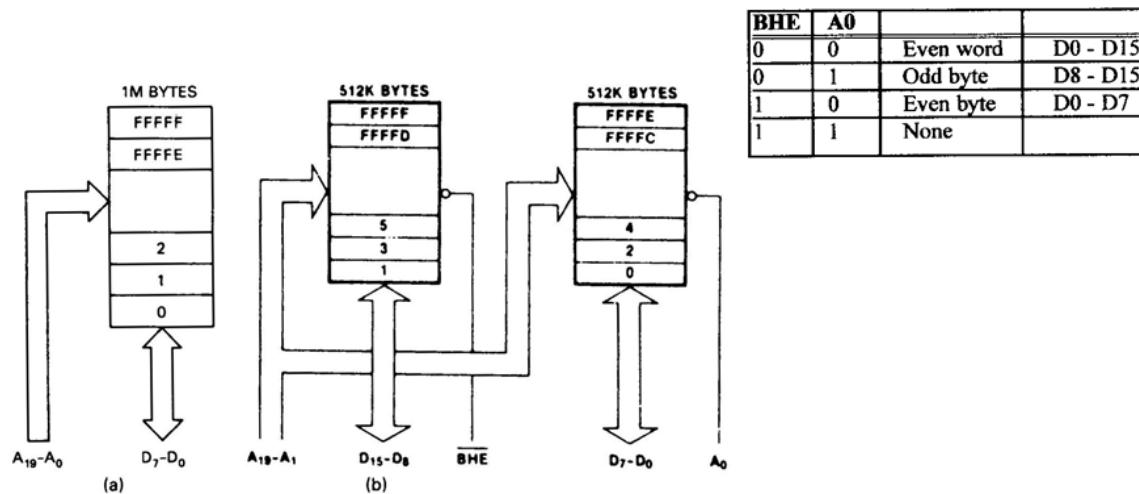
## 奇偶校验位

- 检查一系列位（一个字节）的完整性
- - 偶校验：如果位序列中的1的数目为奇数，则将奇偶校验位设置为1；否则，设置为0，使偶数的总数为1（数据+奇偶校验位）
  - 奇校验：如果位序列中的1的数目为奇数，则将奇偶校验位设置为0；否则，设置为1

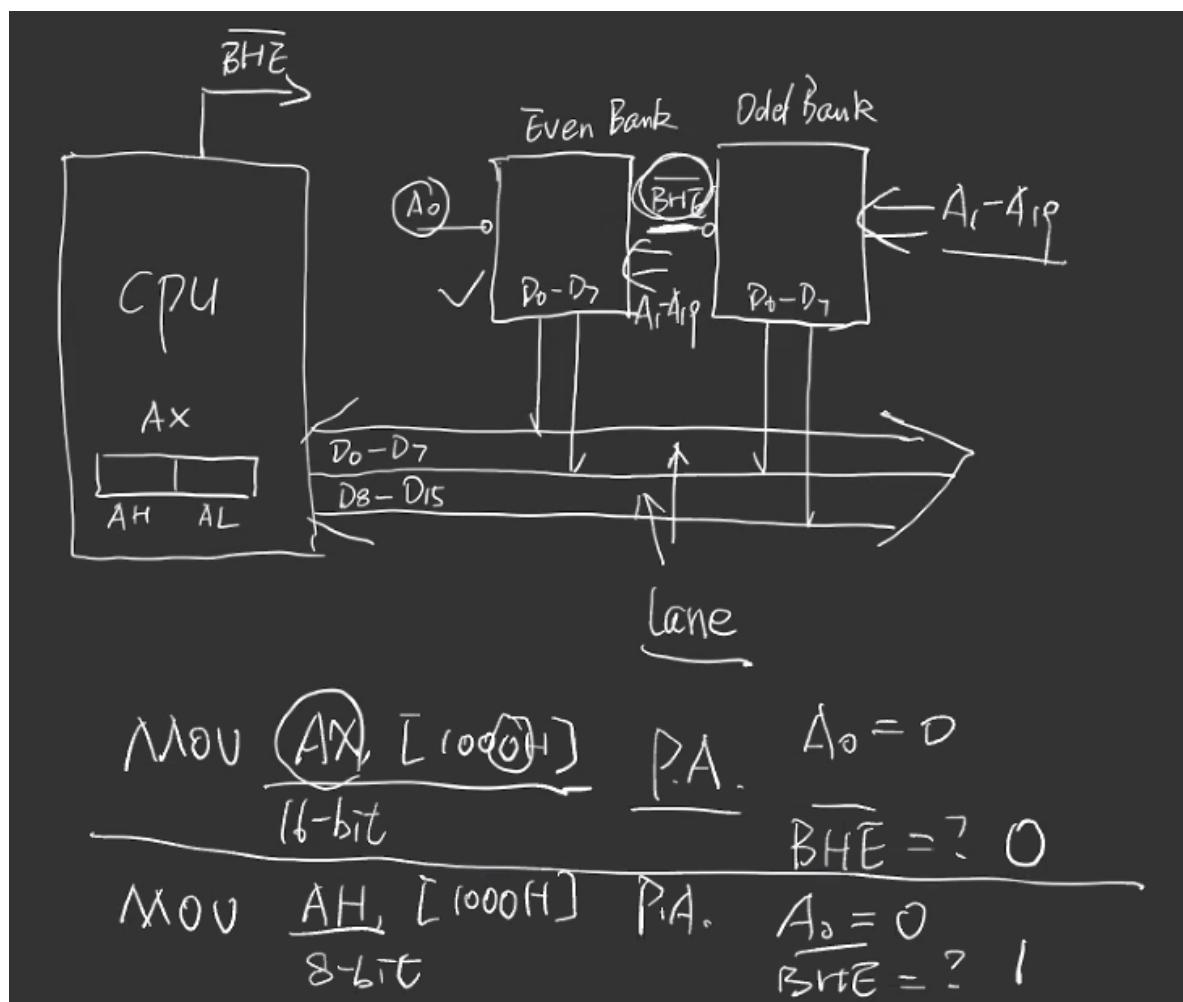
- 对于8086中的PF，使用奇校验

## 8086的memory organization

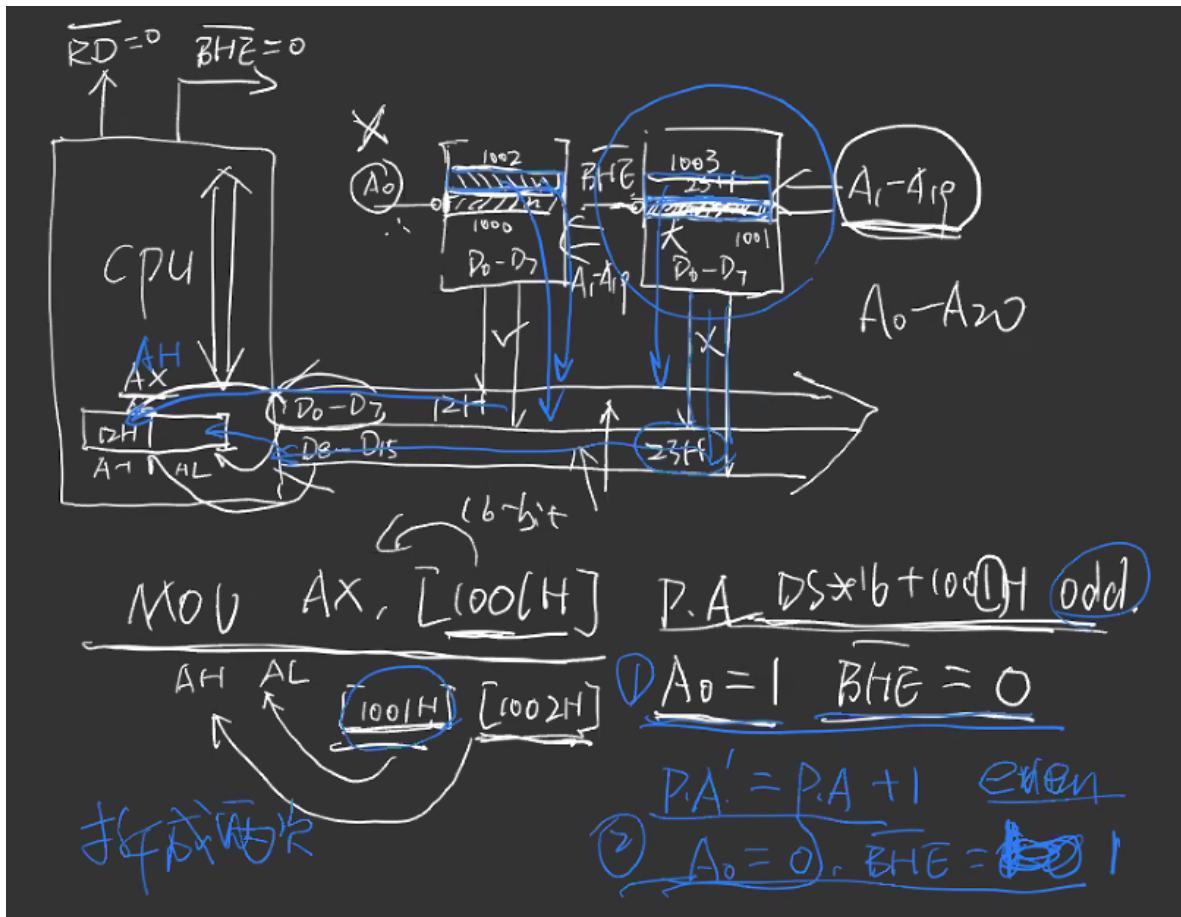
- 偶数和奇数bank



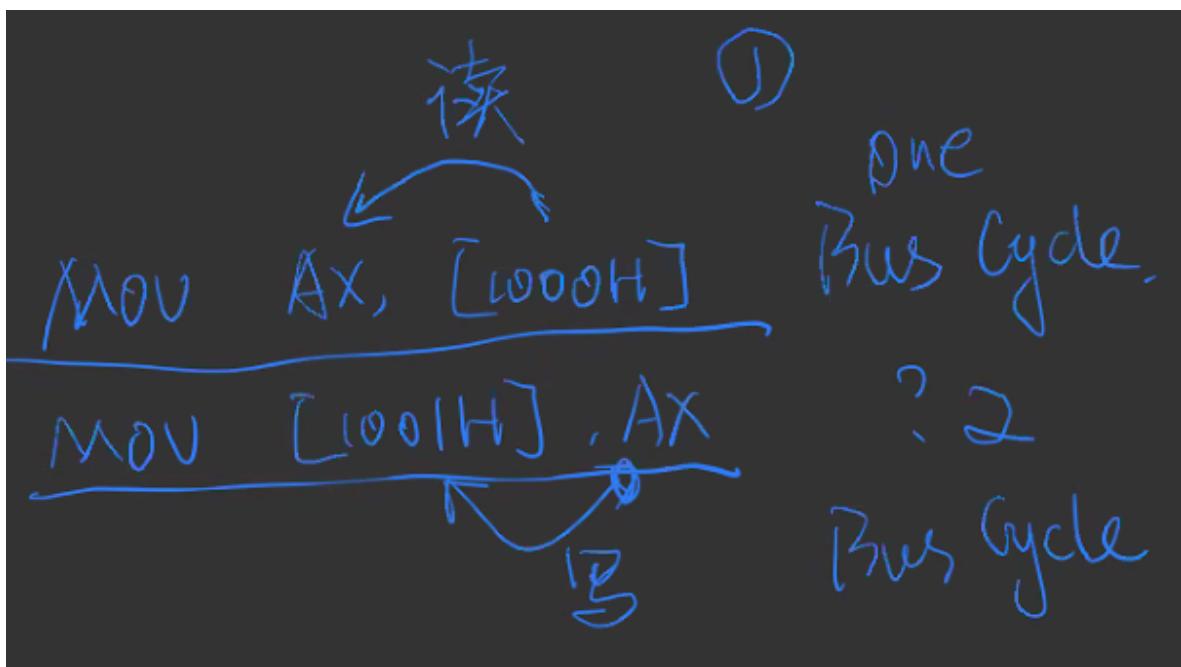
Address bits  $A_1$  through  $A_{19}$  select the storage location that is to be accessed. They are applied to both banks in parallel.  $A_0$  and bank high enable ( $\overline{BHE}$ ) are used as **bank-select** signals.



由于AX16位，CPU将BHE设置为0

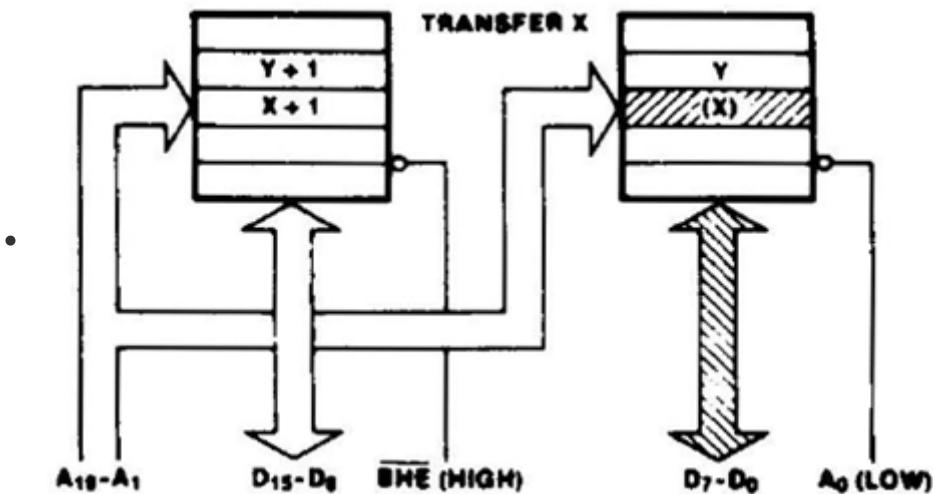


MOC AX,[1001H]无法一次实现，CPU将其拆成两段MOV AL,[1001H];MOV AH,[1002H]

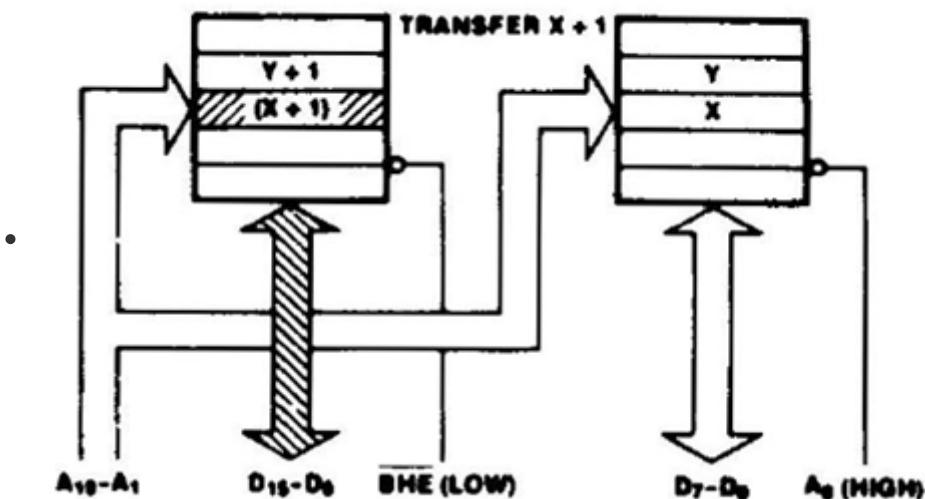


## 字节存储操作

- 偶数地址X的字节存储操作
- MOV AL, [100h]

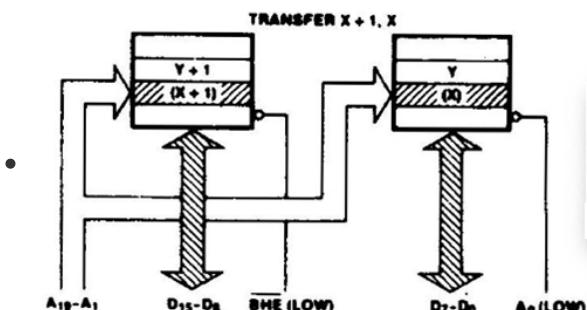


- 奇数地址X + 1处的字节存储器操作
- MOV AL, [101h]



## 对齐的word-memory操作

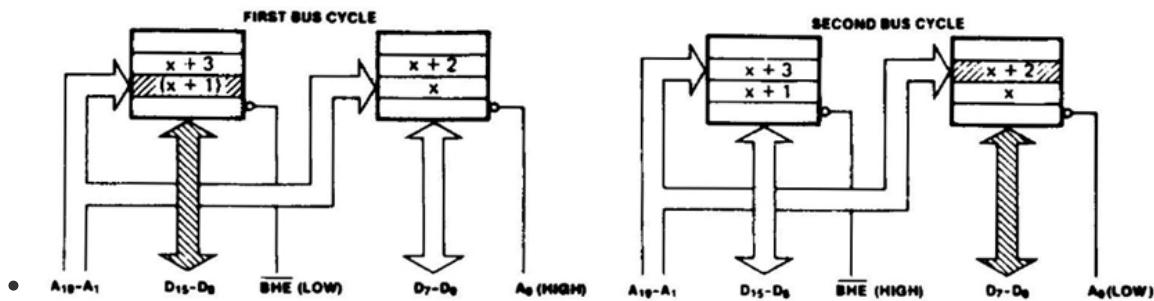
- 在偶数地址X处访问对齐的单词
- MOV AX, [100h]



Both the high and low **banks** are accessed at the same time. Both A<sub>0</sub> and **BHE** are set to 0. This 16-bit word is transferred over the complete data bus D<sub>0</sub> through D<sub>15</sub> in just one bus cycle.

## 错位的word-memory操作

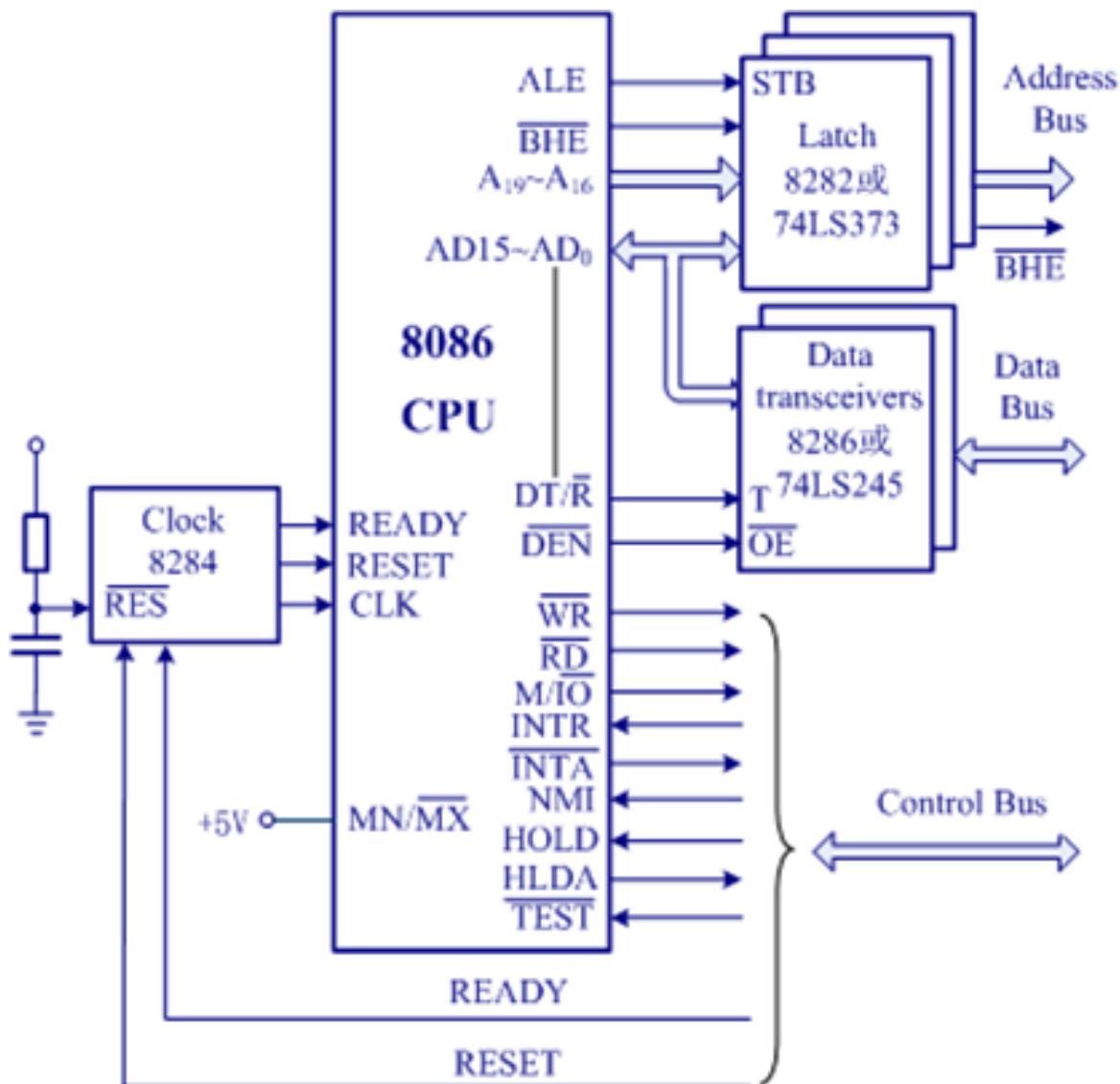
- 在奇数地址X + 1处访问未对齐的单词
- MOV AX, [101h]

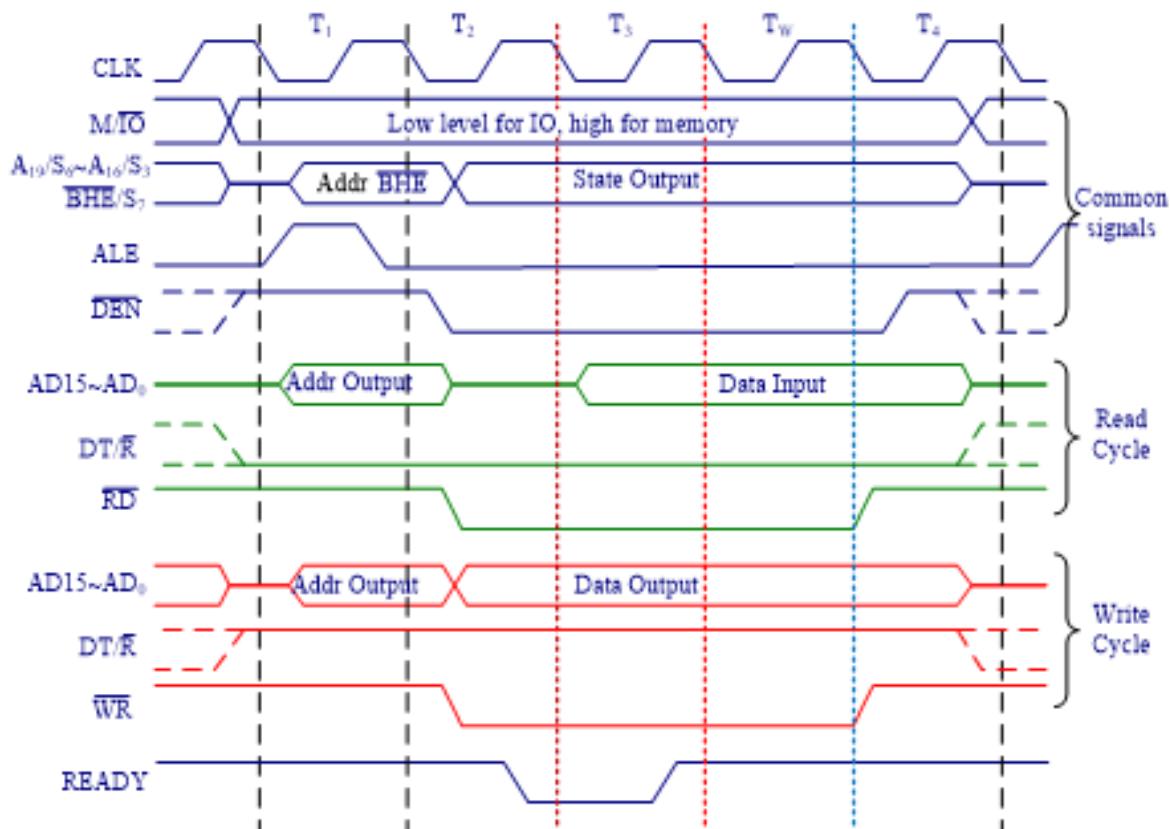


**Two bus cycles** are needed. During the first bus cycle, the byte of the word located at address X + 1 in the high bank is accessed over  $D_8$  through  $D_{15}$ . Even though the data transfer uses data lines  $D_8$  through  $D_{15}$ , to the processor it is the **low byte** of the addressed data word. In the **second memory bus cycle**, the even byte located at X + 2 in the low bank is accessed over bus lines  $D_0$  through  $D_7$ .

## 例子

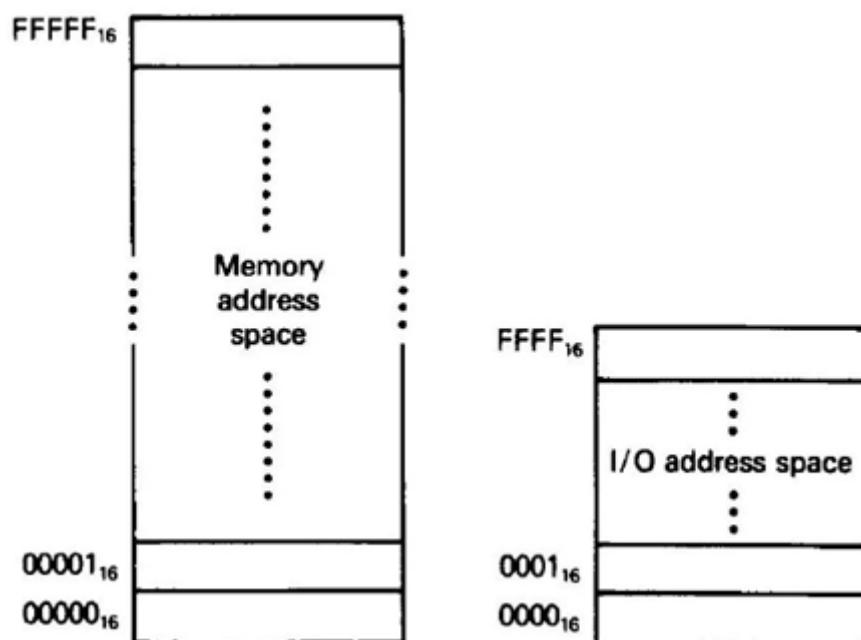
您能否说明8086 CPU执行诸如MOV BX [1000h]之类的指令的完整过程？MOV [1001H], BX呢？





## X86系列中的I / O -内存中的其他空间

- X86微处理器除存储空间外还具有I / O空间
- 使用特殊的I / O指令访问I / O端口上的I / O设备（即，I / O地址）
- 内存可以包含机器代码和数据，I / O端口仅包含数据
- 也称为外围I / O或隔离I / O



## I / O指令- 8 Bit实例

Format:	<u>Inputting Data</u>	<u>Outputting Data</u>
	IN dest,source	OUT dest,source

- 直接I/O指令：
- 端口号范围从00h到0ffh, 总共256个端口, 只能使用AL
  - (1) IN AL, port# OUT port#, AL

- 间接I/O指令：
- 端口号范围从0000h到0ffffh, 总共65536个端口
  - (2) MOV DX, port#  
IN AL, DX  
MOV DX, port#  
OUT DX, AL

- 注意：端口地址没有段概念

## I/O示例

In a given 8088-based system, port address 22H is an input port for monitoring the temperature. Write Assembly language instructions to monitor that port continuously for the temperature of 100 degrees. If it reaches 100, then BH should contain 'Y'.

### Solution:

```
BACK:    IN    AL, 22H
          CMP   AL, 100
          JNZ   BACK
          MOV   BH, 'Y'
```

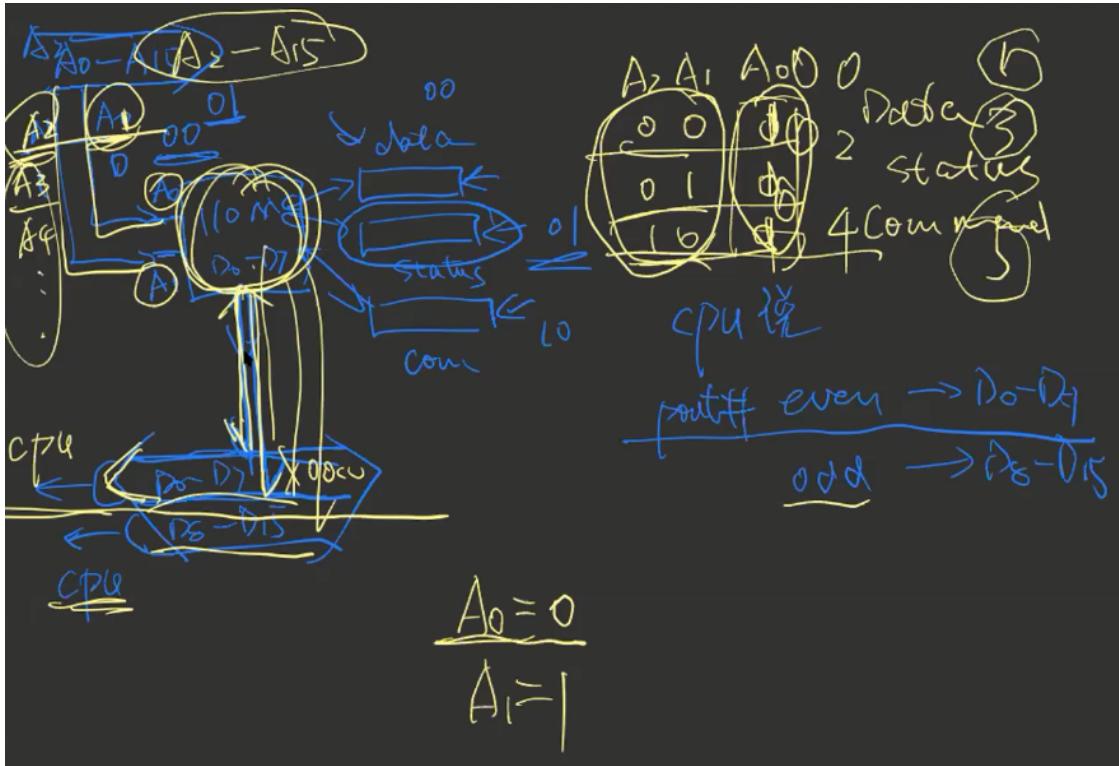
## I/O指令- 16位实例

- 对于16位I/O模块
- 直接I/O指令：
  - 端口号范围从00h到0ffh, 总共256个端口
    - IN AX, 端口号 OUT端口号, AX
- 间接I/O指令：
  - 端口号范围从0000h到0ffffh, 总共65536个端口
    - MOV DX, 端口号 MOV DX, 端口号
      - IN AX, DX OUT DX, AX

## 将8位I/O模块连接到16位数据总线

- 对于8086, 偶数地址端口的数据承载在数据总线D0-D7上, 奇数地址端口的数据承载在数据总线D8-D15上

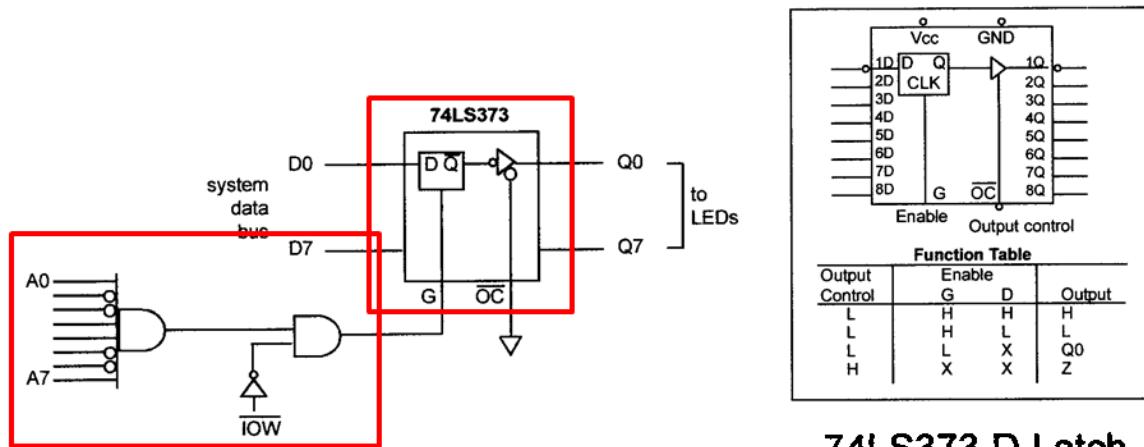
<u>BHE</u>	<u>A0</u>	
0	0	Even-addressed words (uses D0 - D15)
0	1	Odd-addressed byte (uses D8 - D15)
1	0	Even-addressed byte (uses D0 - D7)



如果I/O选择D0-7作为输出，那么输入口只能从A1-15选，因为A0必须保证为偶(A0=0)

## 输出端口设计

- 锁存来自CPU的数据
- 地址解码

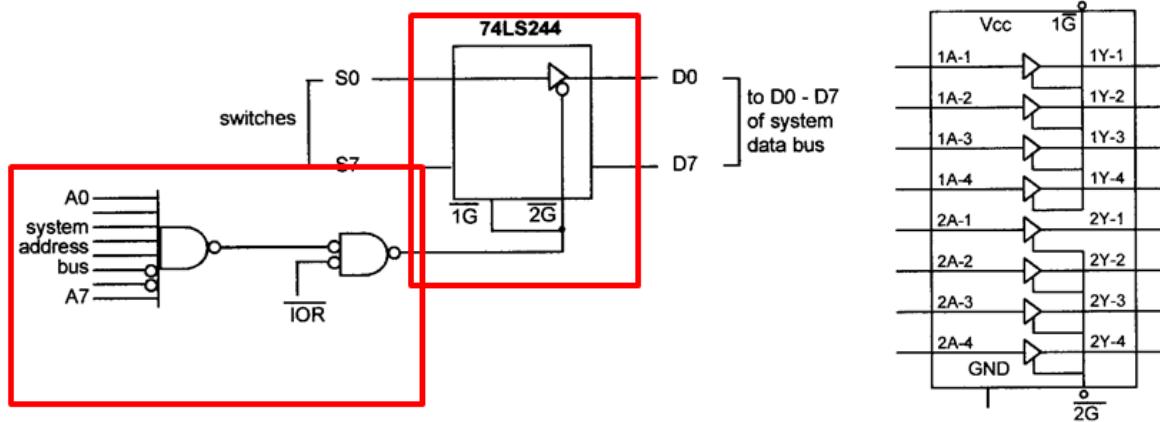


74LS373 D Latch

该端口的地址是什么？

## 输入端口设计

- 使用三态缓冲器连接到系统数据总线
- 地址解码



该端口的地址是什么？

## Ch 07 8255 PPI Chip

### 内部结构和接脚

- 三个数据端口：A, B和C
- 端口A (PA0~PA7)：全部可以编程为input/output
  - 端口B (PB0~PB7)：全部可以编程为input/output
  - 端口C (PC0~PC7)：可以分为两个独立的部分PCU和PCL；任何一位都可以单独进行编程
- 控制寄存器 (CR)
- 内部寄存器：用于setup芯片
  - A组, B组和控制逻辑
- A组 (PA和PCU)
  - B组 (PB和PCL)
- 数据总线缓冲器
- CPU和8255之间的接口
    - 双向，三态（独立，CPU->8255, 8255->CPU），8位
- 读/写控制逻辑
- 内部和外部控制信号
    - RESET**: 高电平有效，清除控制寄存器，所有端口均设为输入端口
    - $\sim CS$ ,  $\sim RD$ ,  $\sim WR$
    - A1, A0**: 端口选择信号

$\sim CS$	$A_1$	$A_0$	$\sim RD$	$\sim WR$	Function
0	0	0	0	1	PA->Data bus
0	0	1	0	1	PB->Data bus
0	1	0	0	1	PC->Data bus
0	0	0	1	0	Data bus->PA
0	0	1	1	0	Data bus->PB
0	1	0	1	0	Data bus->PC
0	1	1	1	0	Data bus->CR
1	x	x	1	1	D <sub>0</sub> -D <sub>7</sub> in float

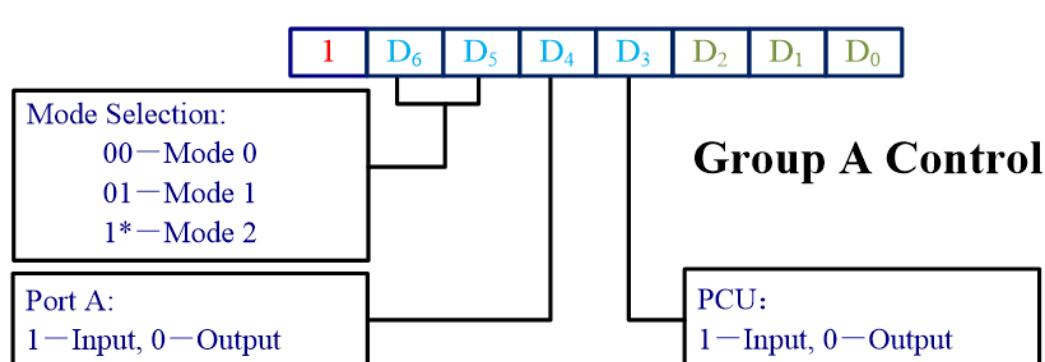
### 操作模式

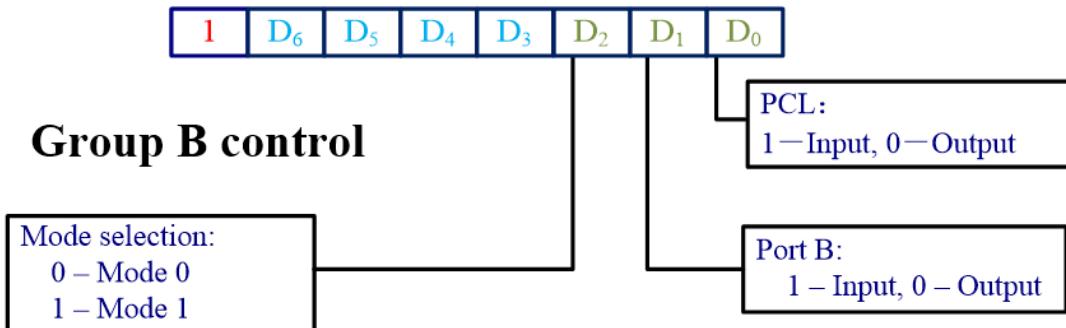
- 输入/输出模式
- ○ 模式0, 简单I / O模式:
  - ■ PA, PB, PC: PCU {PC4~PC7}, PCL {PC0~PC3}
  - 无需握手: 在通信之前两个实体之间进行协商
  - 每个端口均可编程为输入/输出端口
- 模式1:
  - ■ PA, PB可用作具有握手功能的输入/输出端口
  - PCU {PC3~PC7}, PCL {PC0~PC2}分别用作PA和PB的握手线
- 模式2:
  - ■ 仅PA可以用于双向握手数据传输
  - PCU {PC3~PC7}用作PA的握手线
- bit set/reset (BSR) 模式
- ○ 仅PC可用作输出端口
  - PC的每一行都可以单独设置/重置

## 控制寄存器和操作模式

- 控制寄存器
- ○ 8255中的8位内部寄存器
  - 当A1 = 1, A0 = 1时选择
  - 模式选择字
    - ■ 输入/输出模式
      -
    - ■ BSR模式
      -

## 输入/输出模式





## 输入/输出模式示例

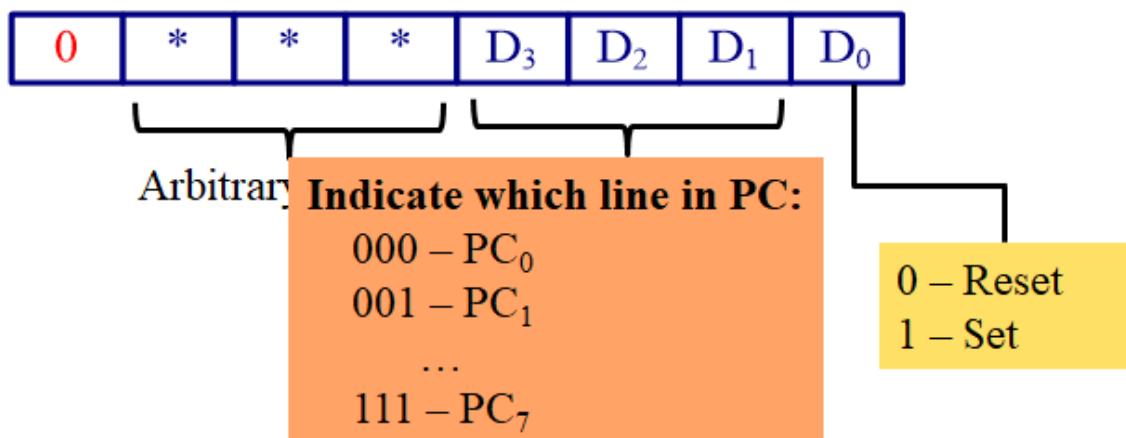
- 编写ASM指令，将8255设置为简单I/O模式，其中PA和PB为输出端口，PC为输入端口。

```
MOV AL, 10001001B
MOV DX, ContralPort
OUT DX, AL
```

- 假设8255的控制寄存器的地址为63H，给出在模式0中设置8255的指令，其中PA，PB和PCU用作输入端口，PCL用作输出端口。

```
MOV AL, 10011010B
OUT 63H, AL
```

## BSR模式



\*代表可为任意值

## BSR模式示例

- 假设PC用作连接8个LED段的输出端口，现在关闭第二个LED段，其余部分保持不变（例如，对于这种情况下的LED段，为1-on, 0-off）

```

MOV AL, 10000000
OUT ContralPort,AL
...
IN AL, CPORt
AND AL, 11111101B
OUT CPORt,AL

```

- 改为使用BSR模式：

```

MOV AL, 00000010B
OUT ControlPORT,AL

```

0BSR模式/000任意值/001代表第二位/0设置低电平

- 假设8255的地址范围是60H~63H，PC5输出低电平，写代码产生一个正脉冲

```

MOV AL, 00001011B ; set PC5high level
OUT 63H,AL
MOV AL, 00001010B ; set PC5low level
OUT 63H,AL

```

101代表PC5

## mode 0 (简单I / O)

- 对于简单的输入/输出方案
- 无需握手
  - PA, PB和PC的任何端口 (PCU, PCL) 均可独立设置为输入或输出端口
  - PCU = PC4~PC7, PCL = PC0~PC3
  - CPU使用IN和OUT指令直接从端口读取或写入端口
  - 输入数据未锁存，输出数据已锁存
  - 例如，为模式0设置控制寄存器

D <sub>7</sub> =1	0	0	*	*	0	*	*
-------------------	---	---	---	---	---	---	---

## 例子

The 8255 shown in Figure 11-13 is configured as follows: port A as input, B as output, and all the bits of port C as output.

(a) Find the port addresses assigned to A, B, C, and the control register.

(b) Find the control byte (word) for this configuration.

(c) Program the ports to input data from port A and send it to both ports B and C.

**Solution:**

(a) The port addresses are as follows:

<u>CS</u>	<u>A1</u>	<u>A0</u>	<u>Address</u>	<u>Port</u>
11 0001 00	0	0	310H	Port A
11 0001 00	0	1	311H	Port B
11 0001 00	1	0	312H	Port C
11 0001 00	1	1	313H	Control register

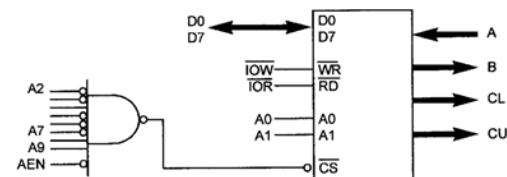


Figure 11-13

(b) The control word is 90H, or 1001 0000.

(c) One version of the program is as follows:

```

MOV AL,90H      ;control byte PA=in, PB=out, PC=out
MOV DX,313H     ;load control reg address
OUT DX,AL       ;send it to control register
MOV DX,310H     ;load PA address
IN  AL,DX       ;get the data from PA
MOV DX,311H     ;load PB address
OUT DX,AL       ;send it to PB
MOV DX,312H     ;load PC address
OUT DX,AL       ;and to PC

```

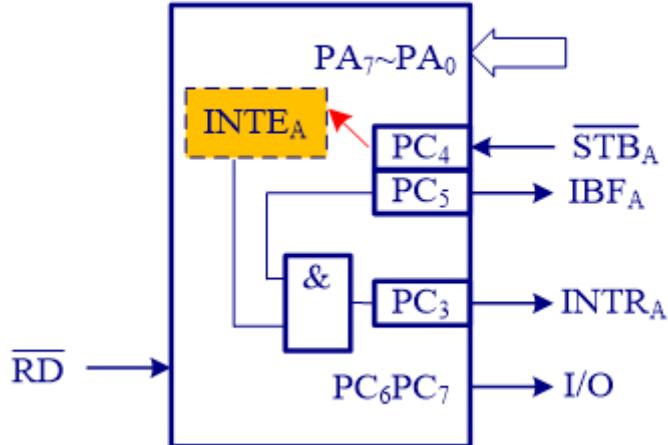
是8088处理器，因为8086会有16-8bit的对应，不连续

## 模式1 (频闪 I / O)

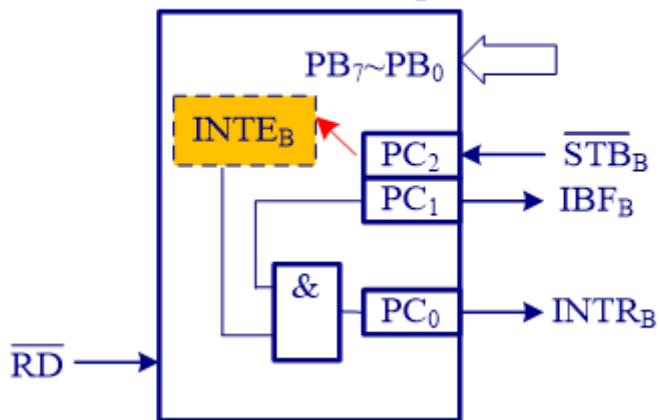
- 握手输入/输出方案
- - PA和PB可用作输入或输出端口
  - PCU = PC3~PC7, 用作PA的握手线
  - PCL = PC0~PC2, 用作PB的握手线
  - 输入和输出数据均被锁存, 类似PWM调光

## 模式1：作为输入端口

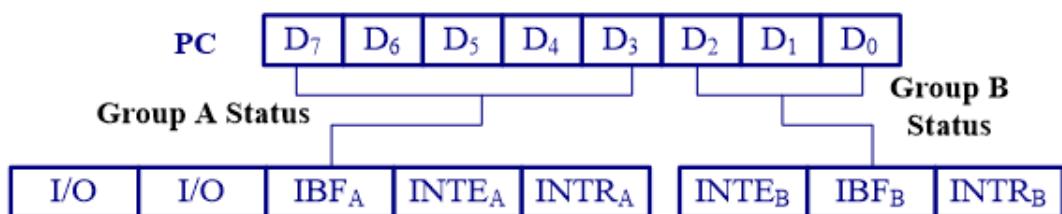
### PA, Mode 1, Input



### PB, Mode 1, Input



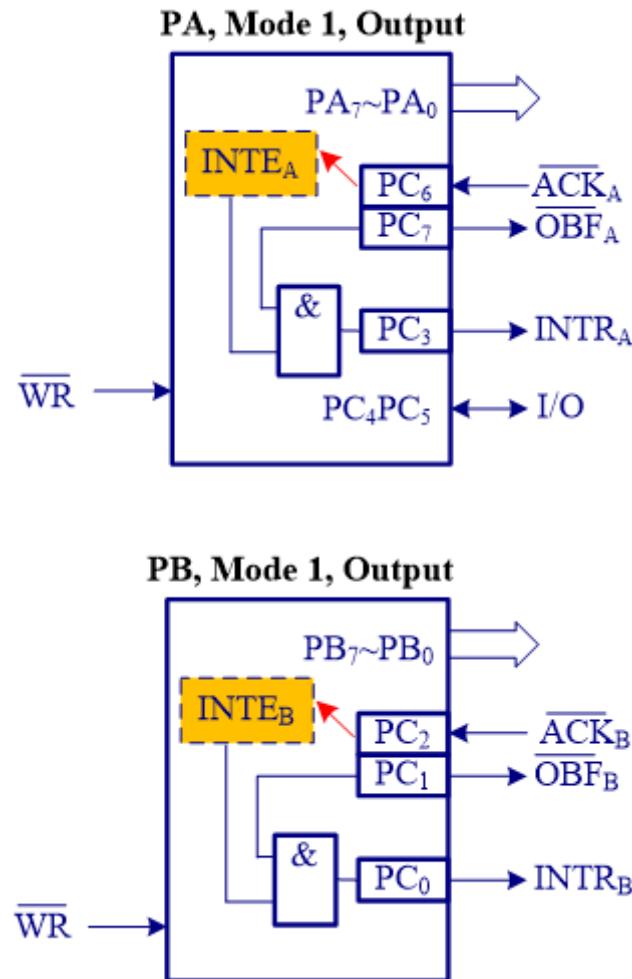
- PC3~PC5和PC0~PC2分别用作PA和PB的握手线
- - $\neg STB$ : 来自输入设备的选通输入信号将数据加载到端口锁存器
  - IBF: IBF到设备的输出信号表示输入锁存器包含信息 (也可用于已编程的I / O)
  - INTR: 中断请求是对CPU的请求中断的输出 (用于中断的I / O)
- PC6和PC7可用作任何目的的独立I / O线
- INTE: 中断允许信号既不是输入也不是输出; 它是通过PC4 (端口A) 或PC2 (端口B) 编程的内部位; 1允许, 0禁止



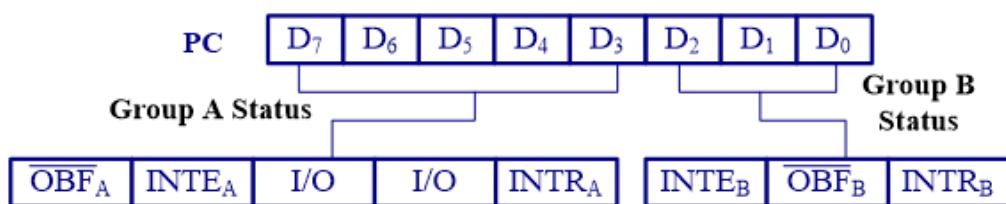
## 模式1输入中的时间线

- 输入设备首先将数据放入PA0~PA7, 然后激活 $\neg STBA$ , 数据被锁存在端口A中。
- 8255激活IBFA, 它指示设备输入锁存器包含信息, 但CPU尚未获取它。因此, 在清除IBFA之前, 设备无法发送新数据。
- 当IBFA,  $\neg STBA$ 和INTEA都为高电平时, 8255激活INTR<sub>A</sub>以通知CPU通过中断来获取PA中的数据;
- CPU响应中断并从PA读取数据;  $\neg RD$ 信号将清除INTR<sub>A</sub>信号;
- CPU完成从PA的数据读取后 (即 $\neg RD$ 信号变为高电平), 将清除IBFA信号。

## 模式1：作为输出端口



- PC3, PC6, PC7 and PC0~PC2 respectively serve as handshake lines for PA and PB.
- - $\sim\text{OBF}$ : OBF is an output signal indicating data has been stored in the port.
  - $\sim\text{ACK}$ : ACK input signal indicating external equipment has received data.
  - INTR: interrupt request signal sent from the port to the CPU.
- PC4 and PC5 can be used as independent I/O lines.
- INTE: interrupt enable signal, neither input nor output; it is programmed through PC6 (Port A) or PC2 (Port B).



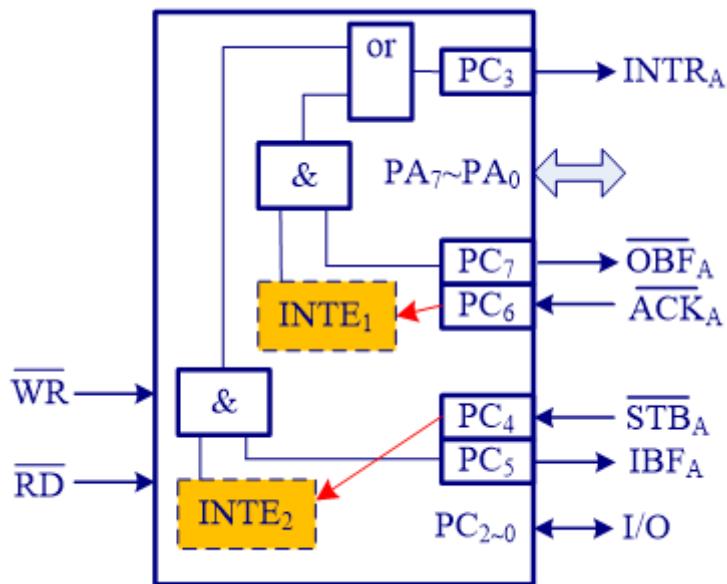
## 模式1输出中的时间线

- If INTRA is activated, the CPU responds by writing data to PA and clearing the INTRA signal.
- When data is stored in PA, 8255 activates  $\sim\text{OBFA}$ ,通知 output device to fetch data.
- After the output device receives data, it sends  $\sim\text{ACKA}$  back to 8255, indicating it has received data and setting  $\sim\text{OBFA}$  high to allow new data write.
- When  $\sim\text{OBFA}$ ,  $\sim\text{ACKA}$  and INTEA are high, 8255 sends INTRA to the CPU via interrupt to write new data to PA.

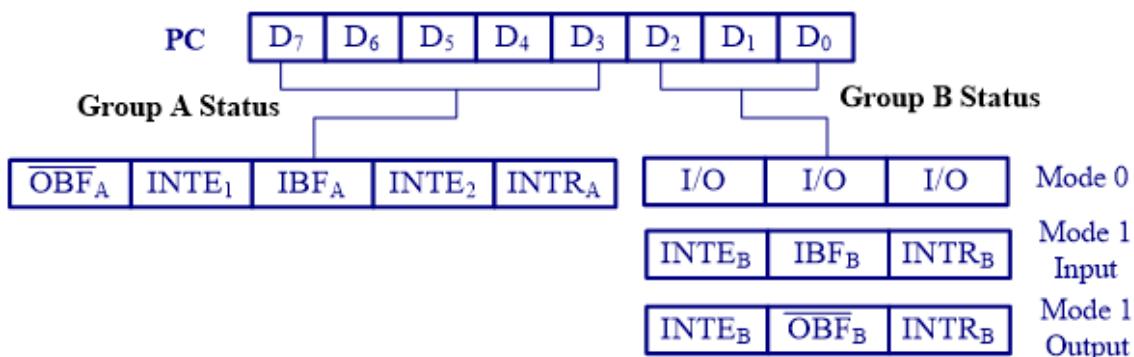
## 模式2 (双向总线)

- 对于双向握手输入/输出方案
- 仅PA可用作输入和输出端口
  - PCU = PC3~PC7, 用作PA的握手线
  - 输入和输出数据均被锁存

### 模式2：作为输入和输出端口



- PC3~PC7用作PA的握手线
- $\neg\text{OBFA}$ ,  $\neg\text{ACK}$ ,  $\neg\text{IBFA}$ ,  $\neg\text{STBA}$ ,  $\text{INTR}$
- PC0~PC2可以用作独立的I/O线, 用于任何用途, 或者作为握手线PB
- 当CPU响应在模式2下工作的8255中断时, 中断处理程序必须检查 $\neg\text{OBFA}$ 和 $\text{IBFA}$ , 以判断输入进程还是输出进程正在生成中断。

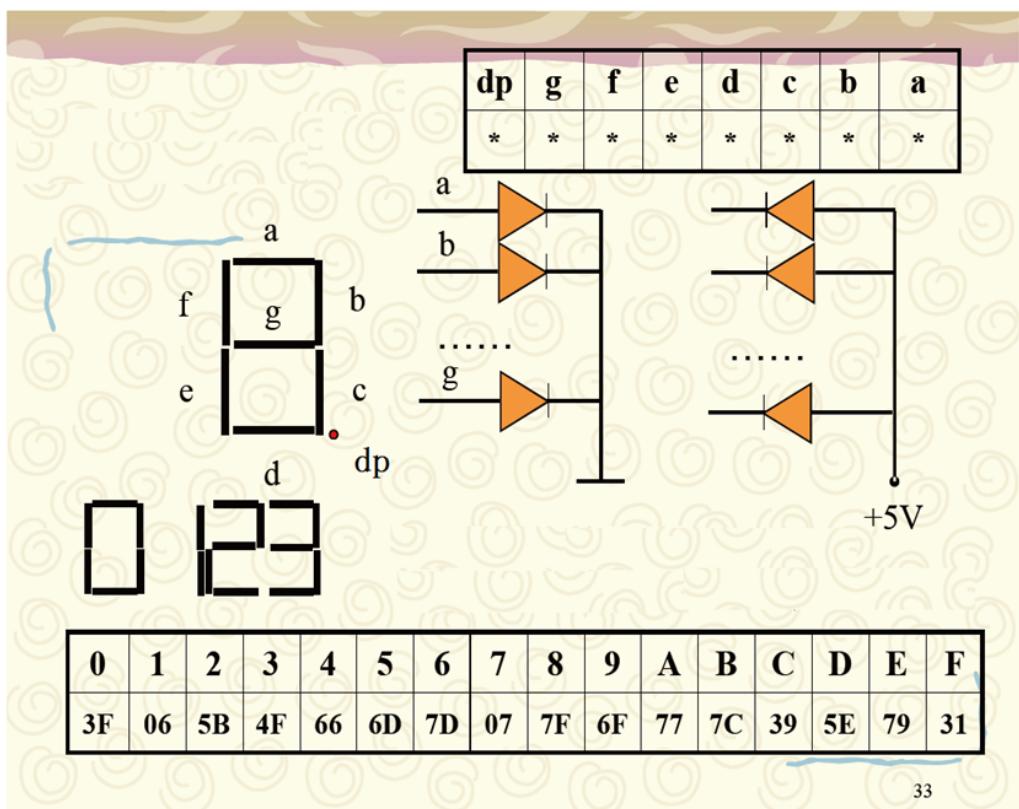
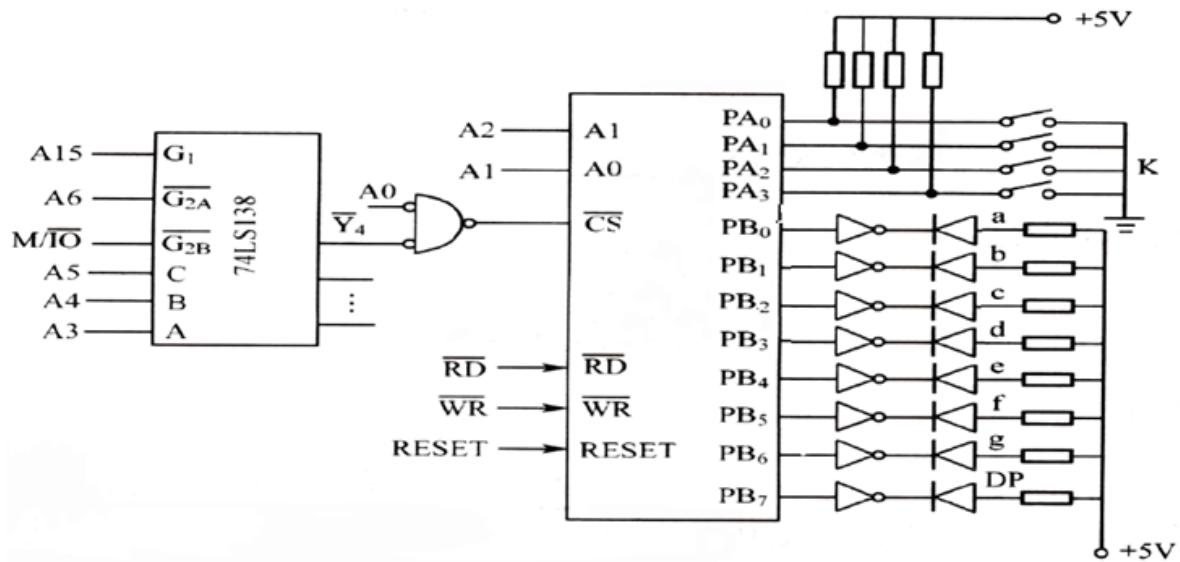


### 轮询与中断

- 在模式1和2中, PC存储组A和/或组B的状态
- 通过使用IN指令从PC读取, 可以使用轮询方法检查I/O设备的状态

## 用8255编程

如图所示, 8255的PA和PB在模式0下工作。用作输入端口的PA连接到4个开关, 用作输出端口的PB连接到7段LED。编写程序以显示开关可以代表的十六进制数字。



## 地址解码

- 端口和控制寄存器的地址是什么？

A <sub>19</sub>	A <sub>18</sub>	A <sub>17</sub>	A <sub>16</sub>	A <sub>15</sub>	A <sub>14</sub>	A <sub>13</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>10</sub>	A <sub>9</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	
0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

- PA: 8020H
- PB: 8022H
- PC: 8024H
- CR: 8026H

## CODE

```
A_PORT EQU 8020H  
B_PORT EQU 8022H  
C_PORT EQU 8024H  
CTRL_PORT EQU 8026H
```

## DATA SEGMENT

```
TAB1 DB C0H, F9H, C4H, ..., 0DH
```

```
DATA ENDS
```

## CODE SEGMENT

```
ASSUME CS:CODE, DS:DATA
```

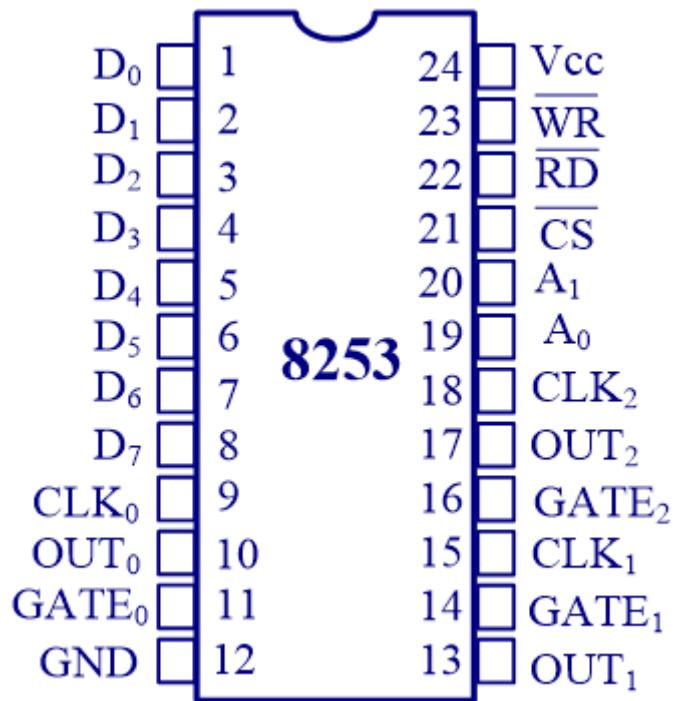
```
START: MOV AX, DATA
```

```
    MOV DS, AX
```

```
    MOV AL, 90H      ; Set up the mode of 8255  
    MOV DX, CTRL_PORT  
    OUT DX, AL  
ADD1: MOV DX, A_PORT  
    IN AL, DX       ; read the status of switches  
    AND AL, 0FH  
    MOV BX, OFFSET TAB1  
    XLAT  
    MOV DX, B_PORT ; output to LED  
    OUT DX, AL  
  
    MOV CX, 0600H   ; delay for lighting the LED  
ADD2: LOOP ADD2  
    JMP ADD1  
CODE ENDS  
END START
```

## Ch 08 8253/4 Timer

### 包装及内部结构



8253/54可编程间隔计时器用于生成较低频率，将输入的时钟频率变慢

CLK输入

OUT输出

GATE控制信号，有效时OUT才输出

有三个Timer，使用A0和A1用于选择timer或CR

### 用途

1. 当做计数器
2. 产生准确的Time delay

### 软件

设置定时循环

```
MOV CX, N
```

```
AGAIN: LOOP AGAIN
```

### 硬件

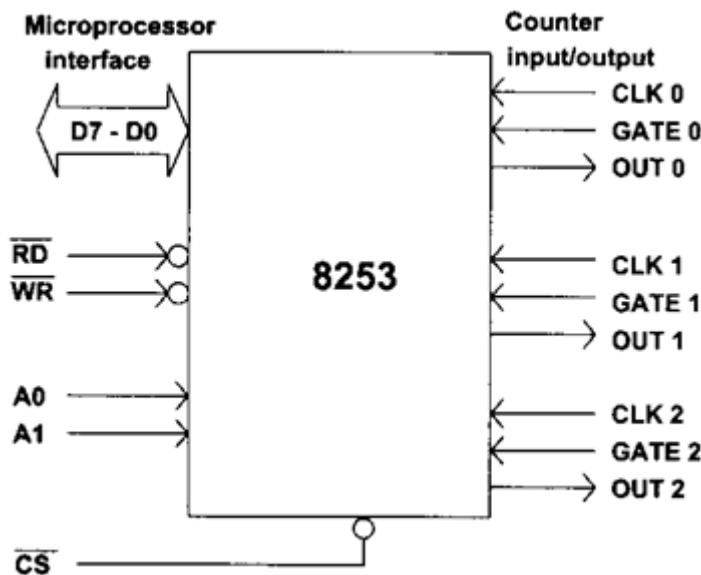
使用8253来计算延迟并中断CPU

根据CLK信号减n，减到0发出中断

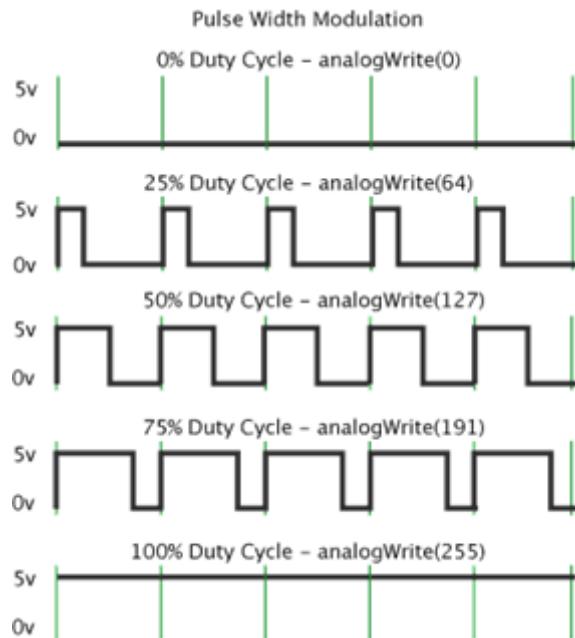
**软件方便，硬件精准**

## 系统接口

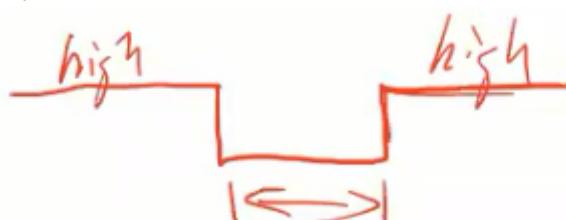
---



- 门用于启用（高）或禁用（低）计数器。
- 如果双向总线D0-D7连接到系统总线的D0-D7，则偶数地址在8086系统中。

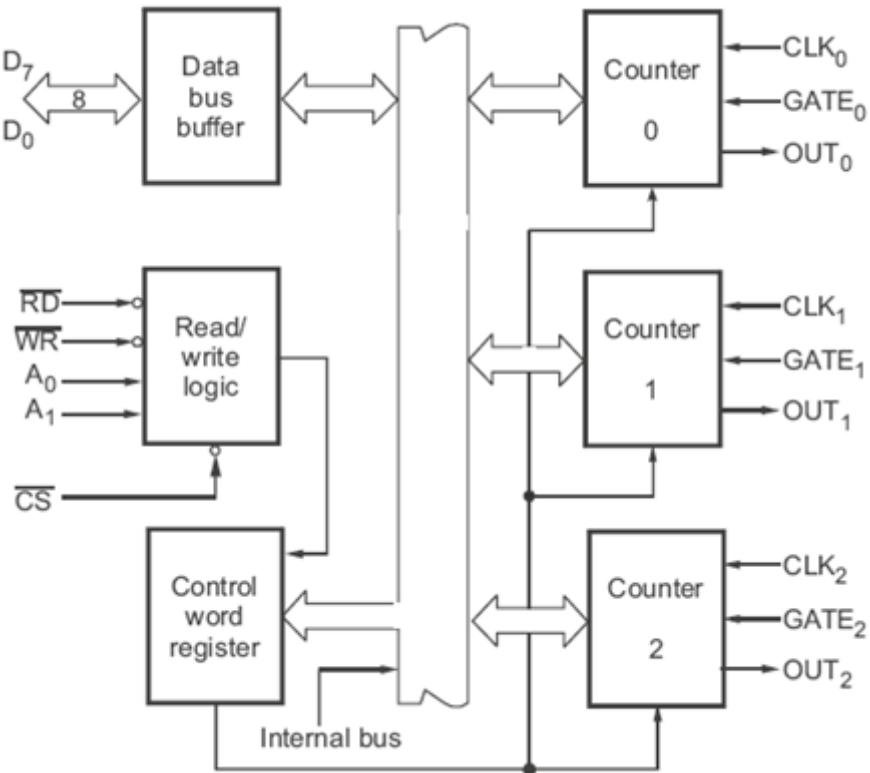


- 有三个独立的计时器。
- 输入频率可以从1到65536（二进制编码 16bit最大值）或从1到10000（BCD编码 每4bit表示一个十进制的数）划分
- 输出频率形状：
  - 方波
  - 产生固定时长低电平
- 

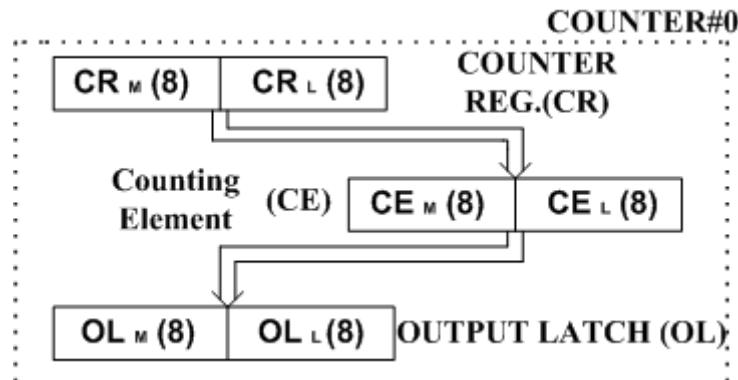


- 具有各种占空比的方波。

## 内部结构



## Counter0



counter reg放计数初值n

counting element n减后结果

CE可以锁住，让OL读出CE值

### counter-16位递减计数器

- 计数器中装入了16位计数
- 开始递减计数直到达到0
- 产生可用于中断CPU的脉冲

## 特征

- 三个独立的16位递减计数器
- 8254可以处理从DC到10 MHz的输入 (5MHz 8254-5 8MHz 8254 10MHz 8254-2)，而8253可以在2.6 MHz的频率下运行
- 三个计数器是相同且可预设的，并且可以针对二进制或BCD计数进行编程
- 计数器可以在六种不同模式下编程
- 与所有英特尔和大多数其他微处理器兼容

- 8254具有功能强大的命令READ BACK，该命令使用户可以检查计数值，编程模式和当前模式以及计数器的当前状态

## 内部结构和引脚

- 数据总线缓冲器
  - 将8253/4连接到系统数据总线
  - 双向三态8位
- 读/写控制逻辑
  - $\sim CS$ 
    - 绑定到解码的地址
    - $\sim RD, \sim WR$
    - 在隔离的I / O中:  $\sim IOR, \sim IOW$
    - 内存映射的I / O:  $\sim MEMR, \sim MEMW$
  - A1, A0
    - 选择控制字寄存器和counter
    - 通常连接到地址线A1, A0

A <sub>1</sub>	A <sub>0</sub>	Selection
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control word Register

/CS	/RD	/WR	A1A0	FUNCTION
0	1	0	00	Write counter0 (to CR0)
0	1	0	01	Write counter1 (to CR1)
0	1	0	10	Write counter2 (to CR2)
0	1	0	11	Write control port
0	0	1	00	Read counter0 (from OL0)
0	0	1	01	Read counter1 (from OL1)
0	0	1	10	Read counter2 (from OL2)
0	0	1	11	Read control port (for 8254)
1	X	X	XX	Not available

- 控制字寄存器:
  - 当A1 = 1, A0 = 1时选择
  - 用于指定要使用的计数器，其模式以及读取或写入操作

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
SC <sub>1</sub>	SC <sub>0</sub>	RW <sub>1</sub>	RW <sub>0</sub>	M <sub>2</sub>	M <sub>1</sub>	M <sub>0</sub>	BCD

SC<sub>1</sub> SC<sub>0</sub> **SC - Select counter**

0	0	Select counter 0
0	1	Select counter 1
1	0	Select counter 2
1	1	Illegal for 8253 Read -Back command for 8254 (See Read operations)

RW<sub>1</sub> RW<sub>0</sub> **RW - Read /Write**

0	0	Counter latch command (See Read operations)
0	1	Read / Write least significant byte only
1	0	Read / Write most significant byte only
1	1	Read / write least significant byte first, then most significant byte

00锁存

01读写低字节

10读写高字节

11同时读写高低字节，先送低字节后送高字节

M <sub>2</sub>	M <sub>1</sub>	M <sub>0</sub>	M - Mode
0	0	0	Mode 0
0	0	1	Mode 1
x	1	0	Mode 2
x	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

BCD :

0	Binary counter 16 - bits
1	Binary coded decimal (BCD) Counter (4 Decades)

- **counter:**

- - 每个计数器都包含一个16位可预置的递减计数器
  - 可以二进制或BCD操作
  - 输入，门和输出通过模式选择进行配置
  - 从计数器读取不会干扰实际的计数

## 写/读操作

- 写：

- - 将控制字写入控制寄存器
  - 将计数的低位字节加载到计数器寄存器中
  - 将计数的高位字节加载到计数器寄存器中

- 读：

- - **简单读取：**两次I / O读取操作，第一个用于低位字节，最后一个用于高位字节
  - **计数器锁存命令：**一个I / O写入操作用于将控制字写入控制寄存器以锁存输出锁存器中的计数，然后如简单读取中所述，使用两个I / O读操作读取锁存的计数。
  - **读回命令：**仅适用于8254

## 例子

CS	A1A0	Port	Port address (hex)
1001 01	00	Counter 0	94
1001 01	01	Counter 1	95
1001 01	10	Counter 2	96
1001 01	11	Control register	97

- (a) counter 0 for binary count of mode 3 (square wave) to divide CLK0 by number 4282 (BCD)  
(b) counter 2 for binary count of mode 3 (square wave) to divide CLK2 by number C26A hex  
(c) Find the frequency of OUT0 and OUT2 in (a) and (b) if CLK0 = 1.2 MHz, CLK2 = 1.8 MHz.

a

(a) To program counter 0 for mode 3, we have 00110111 for the control word. Therefore,

```
MOV AL,37H      ;counter 0, mode 3, BCD
OUT 97H,AL      ;send it to control register
MOV AX,4282H    ;load the divisor (BCD needs H for hex)
OUT 94H,AL      ;send the low byte
MOV AL,AH       ;to counter 0
OUT 94H,AL      ;and then the high byte to counter 0
```

00 对counter0编程

11 读2次

011 mode3

1 BCD

BCD之所以用4282H，是因为这样可以分成4个4bit的数字，分别转为二进制，因为10位小于16位，所以结果一样

b

(b) By the same token:

```
MOV AL,B6H      ;counter2, mode 3, binary(hex)
OUT 97H,AL      ;send it to control register
MOV AX,C26AH    ;load the divisor
OUT 96H,AL      ;send the low byte
MOV AL,AH       ;to count 2
OUT 96H,AL      ;send the high byte to counter 2
```

10110110

10 counter2

11 读两次

011 mode3

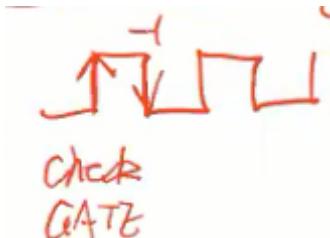
0 16bit

c

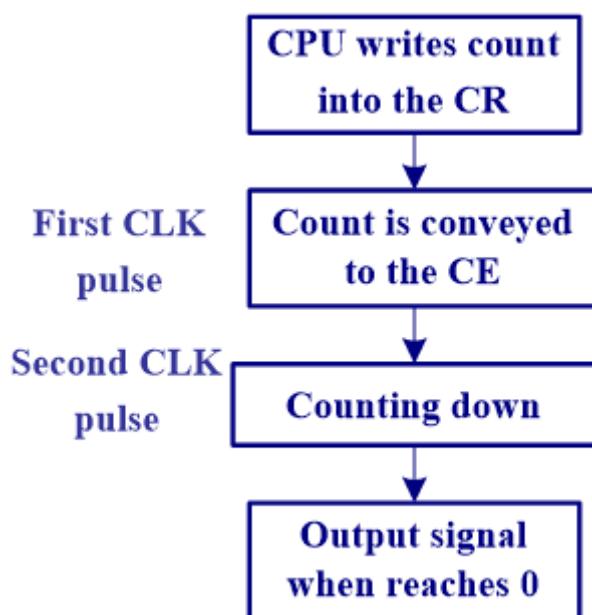
- (c) The output frequency for OUT0 is 1.2MHz divided by 4282, which is 280 Hz. Notice that the program in part (a) used instruction "MOV AX,4282H" since BCD and hex numbers are represented in the same way, up to 9999. For OUT2, CLK2 of 1.8 MHz is divided by 49770 since C26AH = 49770 in decimal. Therefore, OUT2 frequency is a square wave of 36 Hz.

# 8253的特点

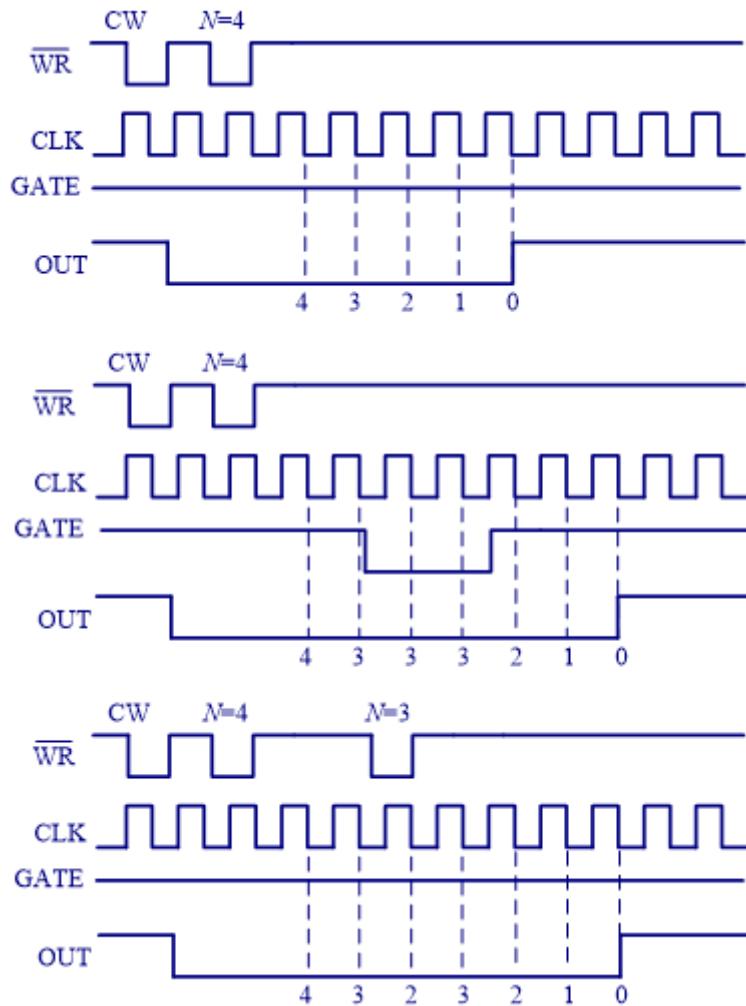
- 8253需要一个CLK脉冲将计数从CR传送到CE
- CE仅在GATE = 1时才开始计数
- - 什么时候检查GATE？
  - 在每个CLK脉冲的上升沿 (0-1)
  - 什么时候倒计时？
  - 在每个CLK脉冲的下降沿 (1-0)



## 模式0：终端计数中断 N减到0后中断



- 普通操作：
  - 模式设置操作后，输出最初将为**低电平**
  - 将计数加载到选定的CR后，输出将保持**低电平**
  - 当达到terminal计数时，输出将变为**高电平**并保持高电平，直到重新加载所选计数器
  - **输出：写入计数后，N个时钟脉冲为低，之后是高**
- GATE：
  - GATE= 1使能计数
  - GATE= 0禁用计数
- new count：
  - 如果将新的计数写入计数器，它将在下一个CLK脉冲加载，并且将从新的计数继续计数
  - 如果是两个字节计数：
    - 写第一个字节将禁用当前计数
    - 写入第二个字节将在下一个CLK脉冲加载新计数，并且将从新计数继续计数



WR写CW (control word) 和计数初值N

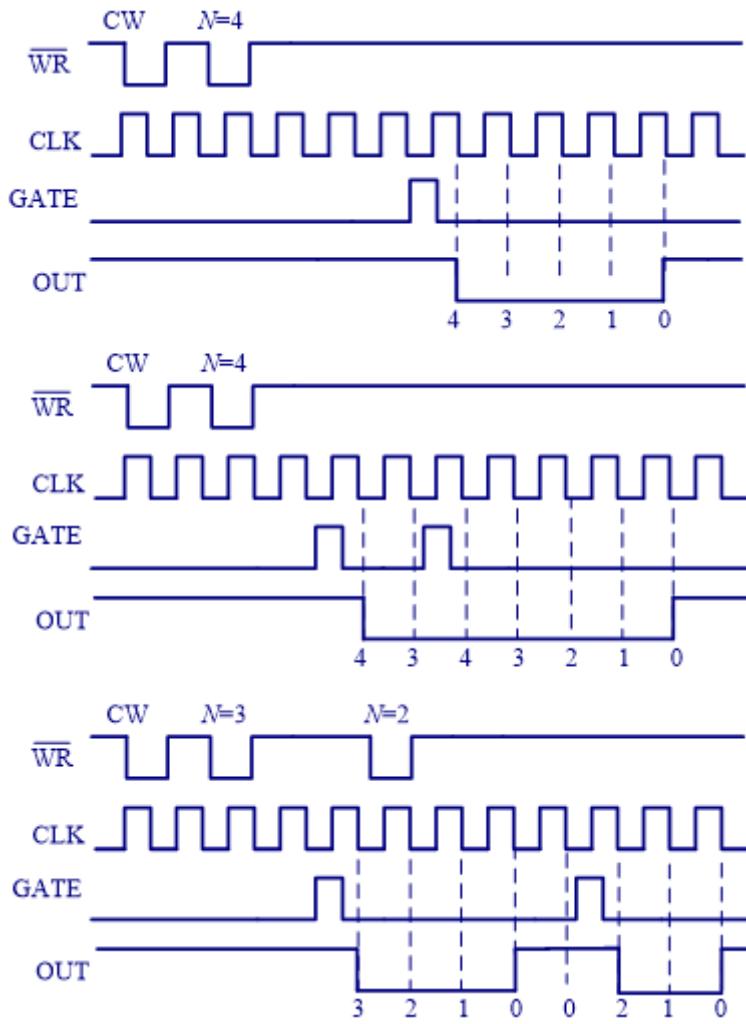
当加载新的计数N时, OUT中的CLK脉冲的实际数量为 $N + 1$ , 因为需要一个周期送到CE

该过程在恢复高电平后不会自动重复

## 模式1：硬件可触发单稳态 (one shot)



- 普通操作:
  - 模式设置操作后, 输出最初为高电平;
  - 在GATE输入的上升沿 (从0到1) 之后, 输出将在该CLK周期后变低电平。
  - 输出将在终端数量上变高电平, 并保持高电平, 直到GATE的下一个上升沿为止。
  - **输出: 每次触发时, N个时钟脉冲的单稳态**
- 触发:
  - 可重新触发, 因此在门输入的任何上升沿之后, 输出将在整个计数期间保持低电平
- 新计数:
  - 如果在一次触发脉冲期间加载计数器, 则除非重新触发计数器, 否则当前一次触发不会受到影响
  - 如果重新触发, 计数器将加载新的计数, 并且单脉冲持续直到新计数到期

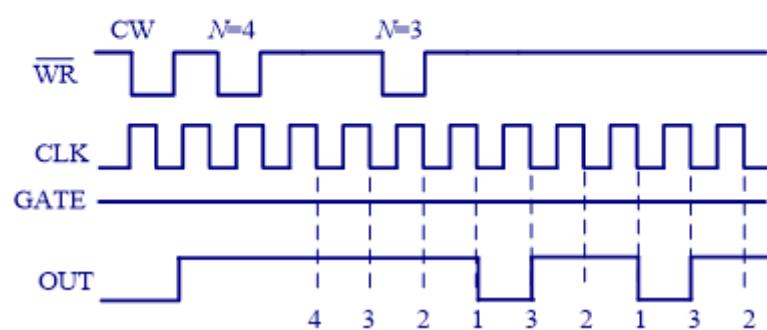
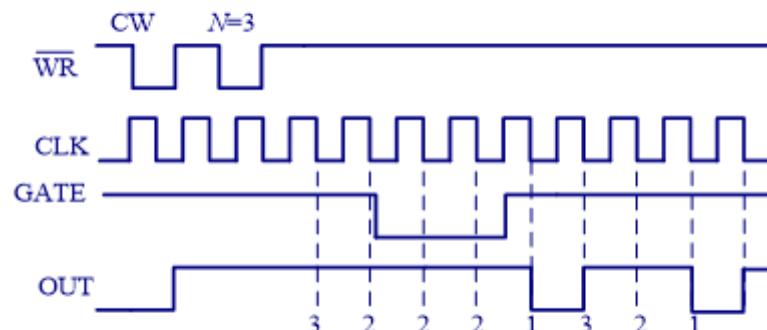
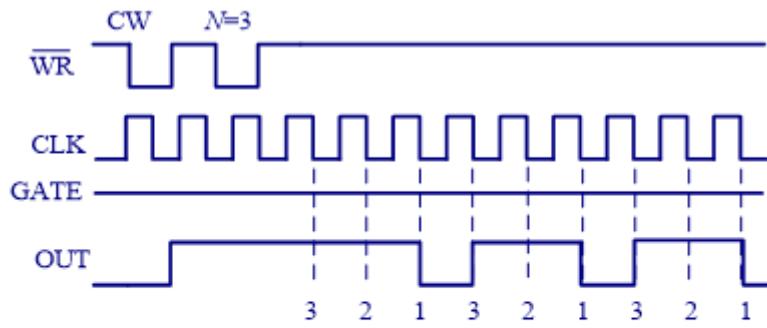


加载新计数  $N$  时，当前计数将不受影响

不会自动重复

## 模式2：速率生成器

- 普通操作：
  - 最初的输出为**高电平**；
  - 在到达 $N$ 之前，输出将产生一个时钟周期的**低电平**；
  - 然后输出变**高电平**，计数器重新加载初始计数，然后重复该过程
  - **输出：周期信号，周期为 $N-1$ 个时钟脉冲为高，一个为1个时钟脉冲为低**
- GATE：
  - 如果 $\text{Gate} = 1$ ，则启用计数，否则禁用计数 ( $\text{Gate} = 0$ )
  - 如果在低输出脉冲期间 $\text{Gate}$ 变低，则输出立即设置为高
- 新计数：
  - 写入新计数时，当前计数顺序不受影响
  - 如果触发信号（**GATE**的上升沿）发生在写入新计数之后，当前周期结束之前，则新计数将在下一个**CLK**脉冲上加载新计数，并且计数将从新计数继续
  - 否则，新计数将在当前计数周期结束时加载
  - 注意：在模式2中，计数1是非法的



$F_{OUT} = F_{IN}/N$   $N-1$ 个高电平  $1$ 个低电平

DUTY CYCLE= $N-1/N$

不工作，HOLD当前CE值

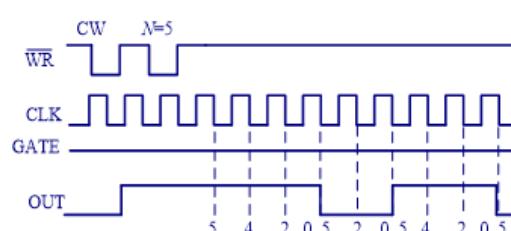
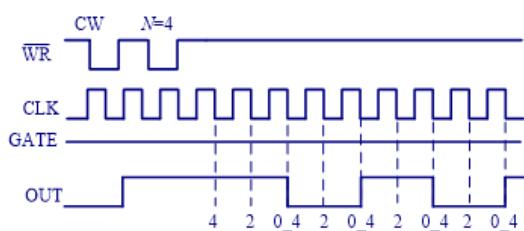
中途更改，下周期生效

加载新计数 $N$ 时，当前计数将不受影响

自动重复终端计数

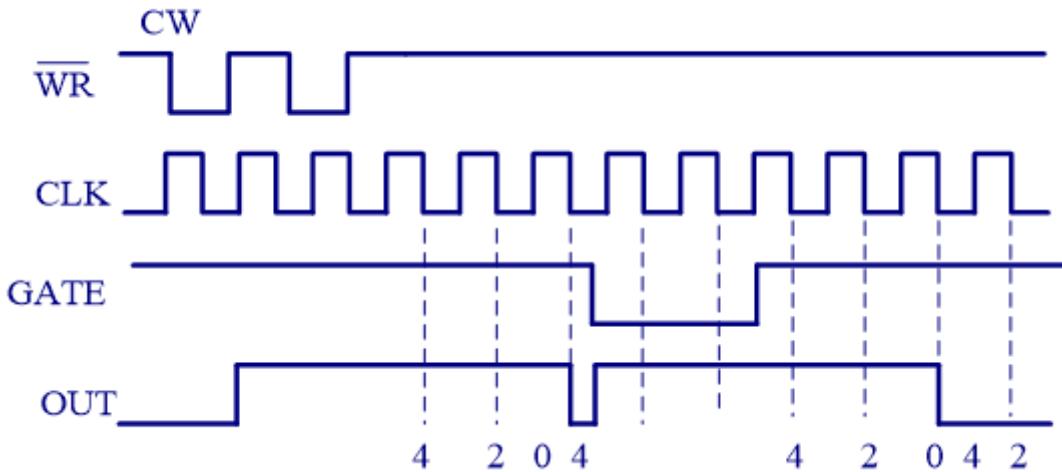
### 模式3：方波速率发生器

方波：占空比为50%的矩形波



- 普通操作：
- ◦ 最初的输出为高电平；
- 对于偶数计数，在每个时钟脉冲的下降沿将计数器减2。当达到终端计数时，输出的状态将更改，并且计数器将全部计数重新加载，并重复整个过程

- 对于奇数计数，第一个时钟脉冲将计数减1。随后的时钟脉冲将时钟减2。超时后，输出变为低电平，并重新加载完整计数。第一个时钟脉冲（跟随重载）使计数递减3，随后的时钟脉冲使计数递减2。然后重复整个过程。
- 输出：如果计数为奇数，则输出将在  $(n + 1) / 2$  个时钟周期为高电平，在  $(n-1) / 2$  个时钟周期为低电平。DUTY CYCLE=N+1/2N**
- GATE：
- 如果Gate为1，则启用计数，否则将被禁用。
- 如果Gate在输出为低电平时变为低电平，则输出将立即设置为高电平。此后，当Gate变为高电平时，计数器将在下一个时钟脉冲上加载初始计数，并重复该序列。



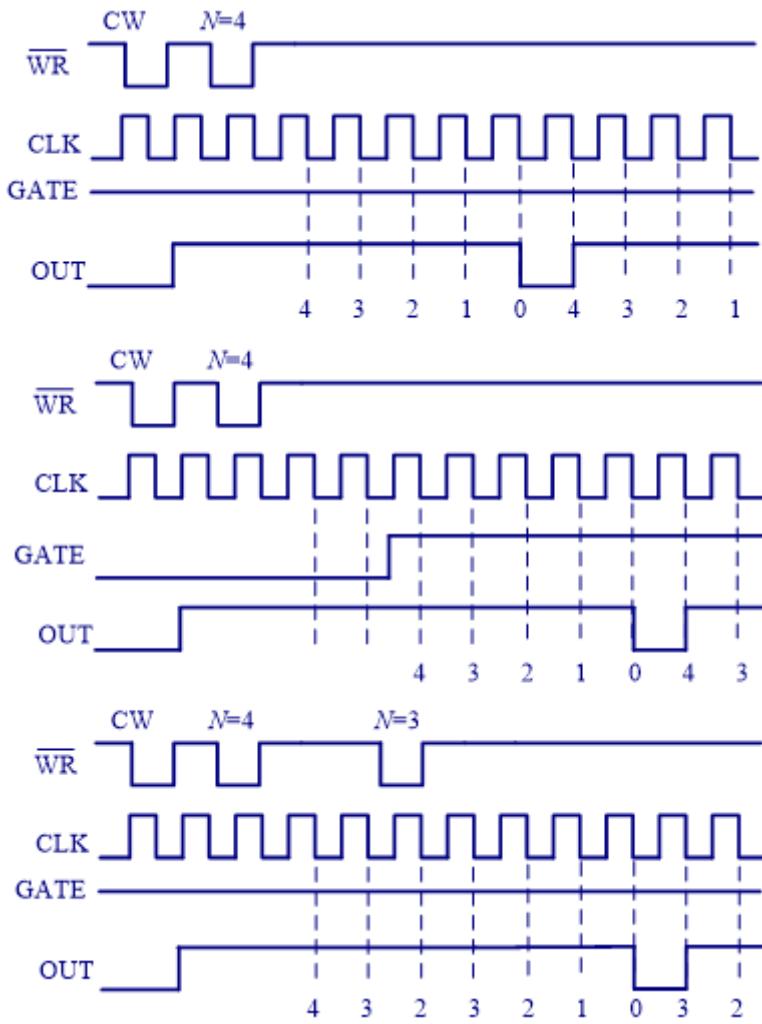
- 新计数：
  - 写入新计数时当前计数顺序不受影响
  - 如果触发发生在写入新的计数之后，方波的当前半周期结束之前，则计数器将在下一个CLK脉冲上加载新的计数，并且将从新的计数继续计数。
  - 否则，新计数将在当前半周期末加载。

加载新的计数N时，当前一半将不受影响

自动重复终端计数

## 模式4：软件触发的选通 Stobe

- 普通操作：
- 最初的输出为**高电平**；  
到达目标值后，输出将在一个CLK脉冲下变**低电平**
- GATE：
- 如果Gate为1，则启用计数；否则，它被禁用
- 新计数：
- 如果在计数过程中写入了新计数，它将在下一个CLK脉冲时加载，并且将从新计数继续计数。如果计数是两个字节，则：
  - 写第一个字节对计数没有影响
  - 写入第二个字节允许将新的计数在下一个CLK脉冲时被加载

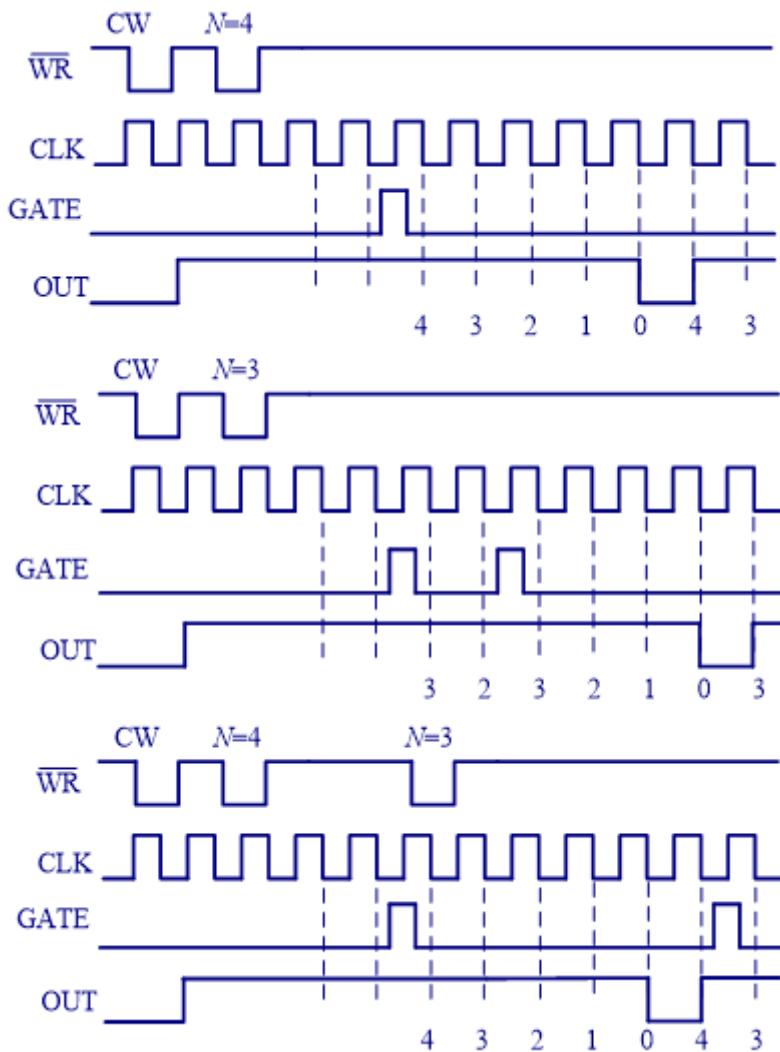


当加载新的计数 $N$ 时, OUT中的CLK脉冲的实际数量为 $N + 1$

自动重复

## 模式5：硬件触发的选通（可触发）

- 普通操作:
  - 最初的输出为**高电平**;
  - 计数由GATE的上升沿触发
  - 终端计数后, 输出将在一个CLK脉冲下变**低电平**
- 触发:
  - 如果在计数过程中触发, 下一个CLK脉冲时初始计数将加载, 并且计数将继续进行, 直到达终端计数为止
- 新计数:
  - 当前的计数顺序将不受影响。如果触发发生在新计数写入之后但在终端计数之前, 则计数器将在下一个CLK脉冲加载新计数, 并从此处继续计数



加载新计数N时，当前计数将不受影响

自动重复终端计数

## 编程范例

**Example 1:** Write a program to initialize counter 2 in mode 0 with a count of C030H.  
 Assume address for control register = 0BH, counter 0 = 08H, counter 1 = 09H  
 and counter 2 = 0AH.

Sol. : Control word

D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	
SC <sub>1</sub>	SC <sub>2</sub>	RW <sub>1</sub>	RW <sub>0</sub>	M <sub>2</sub>	M <sub>1</sub>	M <sub>0</sub>	BCD	= B0H

### Source Program

```

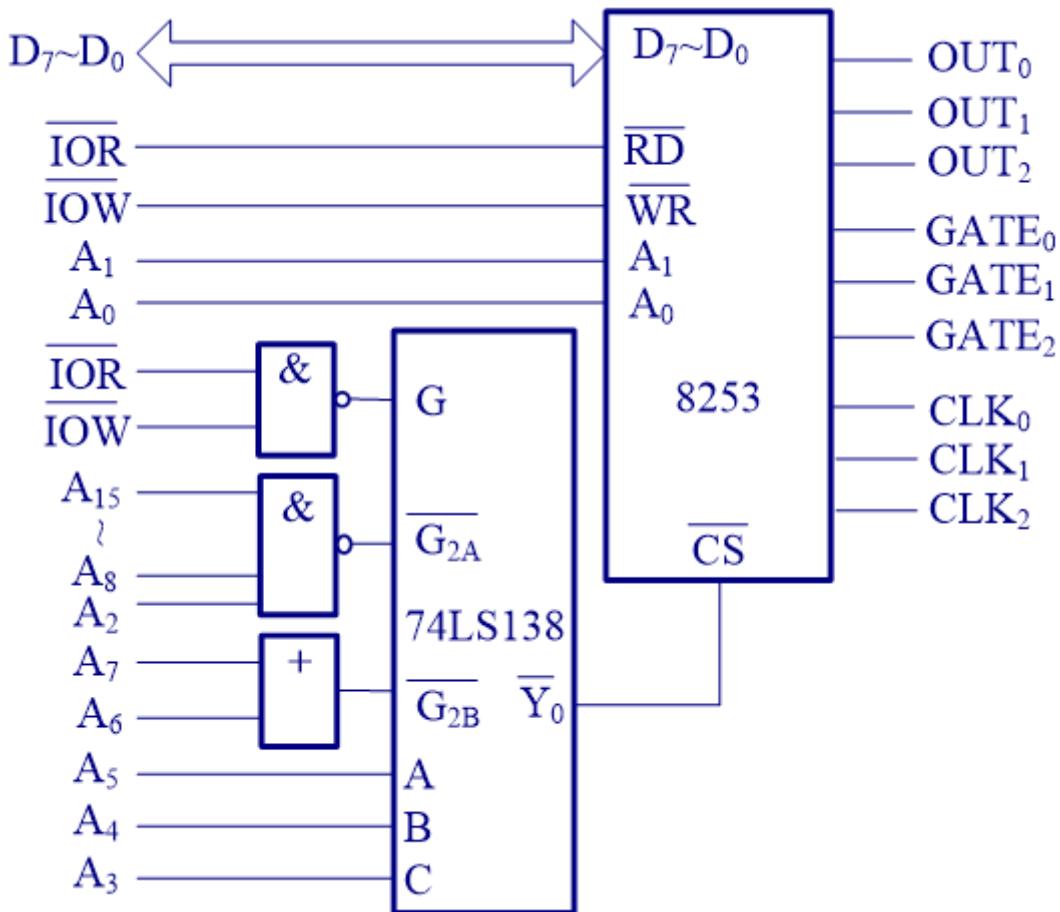
MOV AL,B0H
OUT 0BH,AL      ; Loads control word (B0H) in the control
                  ; register.

MOV AL,30H
OUT 0AH,AL      ; Loads lower byte of (30H) the count.

MOV AL,0C0H
OUT 0AH,AL      ; Loads higher byte (C0H) of the count.

```

CLK的频率为2MHz, 编写启动程序让计数器0在100μs之后产生中断请求(M0), 让计数器1产生周期为10μs的50%占空比方波(M3), 并让计数器2每1ms产生一个负脉冲。



```

MOV DX, OFF07H
MOV AL, 00010000B      ;counter 0, write LSB only, mode 0, binary
OUT DX, AL
MOV AL, 01010110B      ;counter 1, write LSB only, mode 3, binary
OUT DX, AL

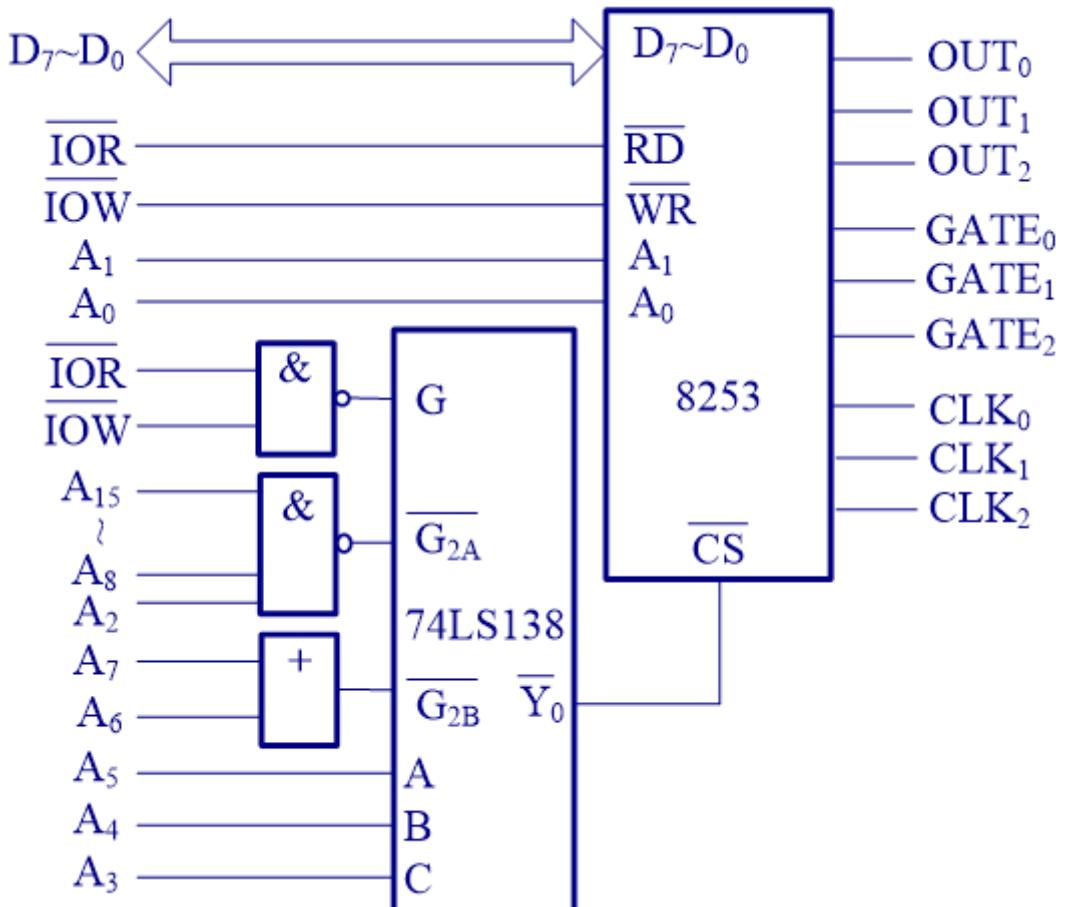
MOV DX, OFF04H
MOV AL, 200            ; initial count for counter 0
OUT DX, AL
MOV DX, OFF05H
MOV AL, 20              ;initial count for counter 1
OUT DX, AL

MOV DX, OFF07H
MOV AL, 10110100B      ;counter 2, write LSB and MSB, mode 2
OUT DX, AL

MOV DX, OFF06H
MOV AX, 2000           ; initial count for counter 2
OUT DX, AL
MOV AL, AH
OUT DX, AL

```

CLK的频率为2MHz, 写启动程序让计数器1产生周期为1s的占空比为50%的方波。



```

MOV DX, OFF07H
MOV AL, 00110110B      ;counter 0, write LSB and MSB, mode 3, binary
OUT DX, AL

MOV DX, OFF04H
MOV AL, 1000            ; initial count for counter 0 2M=2K
OUT DX, AL
MOV AL, AH
OUT DX, AL

;OUT0=CLK1

MOV DX, OFF07H
MOV AL, 01110110B      ;counter 1, write LSB and MSB, mode 3, binary
OUT DX, AL

MOV DX, OFF05H
MOV AL, 2000            ; initial count for counter 1 2K=1
OUT DX, AL
MOV AL, AH
OUT DX, AL

```

## Ch 09 中斷和8259

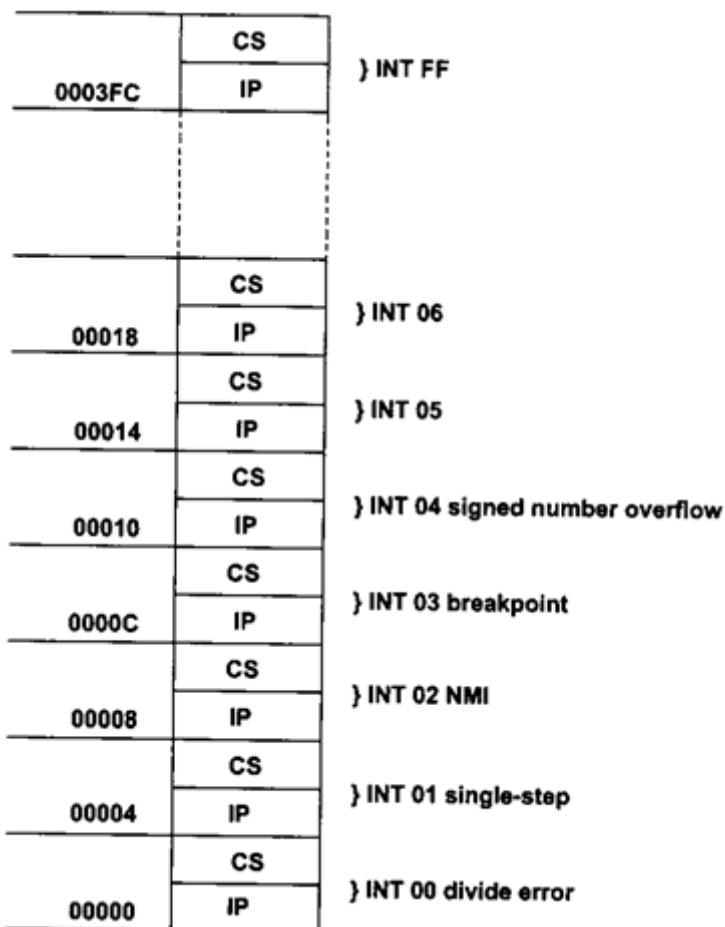
### 8086/8088中的中斷

- 总共256种**中断类型**
- ○ INT 00~INT OFFh

- Type \*4 = 中断向量的 PA (CS:IP)
- ○ 前1KB用于存储中断向量，称为中断向量表 (IVT)

Table 14-1: Interrupt Vector

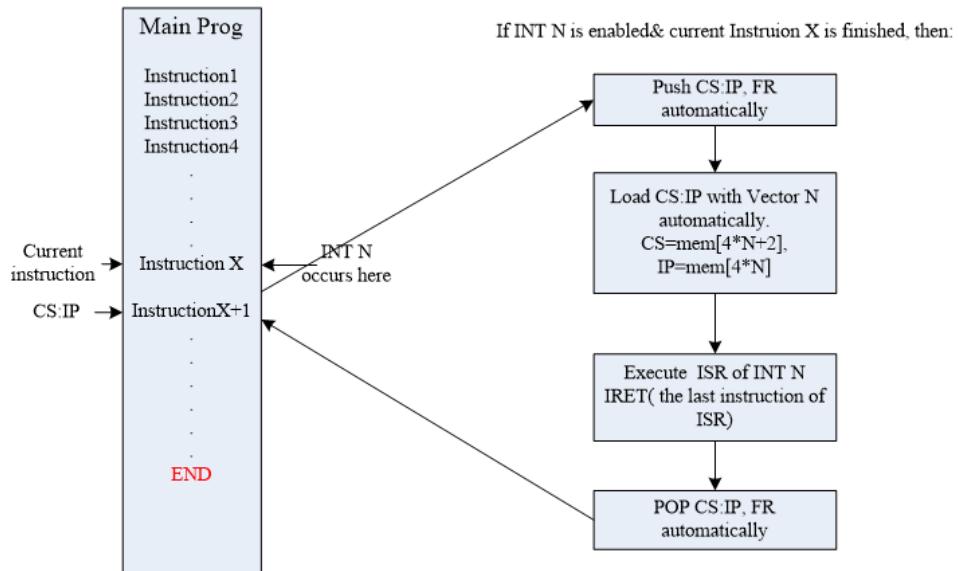
INT Number	Physical Address	Logical Address
INT 00	00000	0000:0000
INT 01	00004	0000:0004
INT 02	00008	0000:0008
INT 03	0000C	0000:000C
INT 04	00010	0000:0010
INT 05	00014	0000:0014
...	...	...
INT FF	003FC	0000:03FC



- 中断向量指向相应**中断服务程序 (ISR)** 的入口地址

## 主程序和ISR

---



ISR由中断事件（内部“INT xx”或外部NMI和INTR）启动。因此，ISR与主体“分离”了。

遇到IRET恢复

## 中断类别

- 硬件（外部）中断
  - 可屏蔽（来自INTR）
  - 不可屏蔽（来自NMI）
- 软件（内部）中断
  - 使用INT指令
  - 预定义的条件（异常）中断

## 硬件中断

- 不可屏蔽中断 Non-maskable interrupt
  - 触发：NMI引脚，输入信号，上升沿和两周期高电平激活
  - 类型号：INT 02
  - 不受IF影响 (interrupt flag)
  - 原因：
    - 例如，RAM奇偶校验错误，来自辅助CPU 8087的中断请求
- 可屏蔽中断 maskable interrupt
  - 触发：INTR引脚，输入信号，高电平有效
  - TYPE：无预定义类型
  - IF = 1, 使能；如果= 0, 禁用
  - STI设置IF, CLI清除IF
  - 原因：
    - 是外部I / O设备的中断请求

## 处理可屏蔽中断的过程

- CPU响应INTR中断请求
- - 外部I / O设备向CPU发送中断请求
  - CPU将在指令的最后一个周期检查INTR引脚：如果INTR为高电平且IF = 1，则CPU响应中断请求。中断服务程序写入STI后，该中断服务程序可以被其他中断再次中断
  - CPU将~INTA引脚上的两个负脉冲发送到I / O设备，清除INTR
  - I / O设备收到第二个~INTA后，在数据总线上发送中断类型N
    - 先将N写入外设中
    - 将N对应的位置设为INTR入口
- CPU执行INT N的ISR
- - CPU从数据总线读取N
  - 将FR推入堆栈
  - 清除IF和TF，CPU进入INTR后不再接受其他中断请求
  - 将下一条指令的CS和IP推入堆栈
  - 加载ISR入口地址并移至ISR
  - 在ISR结束时，IRET将依次弹出IP，CS和FR，CPU返回上一个程序并继续

## 软件中断

- INT xx指令
- - 根据“INT xx”等指令调用ISR
    - 例如int 21h; Dos service
  - CPU始终响应并执行相应的ISR
  - 不受IF影响
  - 您可以使用INT指令“调用”任何ISR

## INT和CALL之间的区别

INT调用ISR，CALL调用PROC

- CALL FAR可以跳转到1MB以内的任何位置，而INT可以跳转到固定位置（查找相应的ISR）
- 调用FAR按指令顺序进行，INT与CALL指令差不多，而硬件中断可以随时进入
- 不能屏蔽（禁用）CALL FAR，而可以屏蔽外部中断
- CALL FAR保存下一条指令的CS: IP，而INT保存下一条指令的FR + CS: IP
- 最后一个指令：RETF与IRET

## 软件中断

- 预定义的条件中断
- - “INT 00”（除法错误）
    - 原因：将数字除以零，或商太大
  - “INT 01”（单步）
    - 如果TF = 1，则CPU将在执行每条调试指令后生成INT 1中断。
    - 用于debug
    - 如何清除TF？

```

■ PUSHF
    POP AX
    AND AX 0FEFFF
    PUSH AX
    POPF

```

- “INT 03” (断点)
  - 当CPU达到程序中设置的断点时，CPU会生成INT 3中断以进行调试
- “INT 04” (带符号的数字溢出)
  - INT0指令
  - 算术指令后检查OF

```

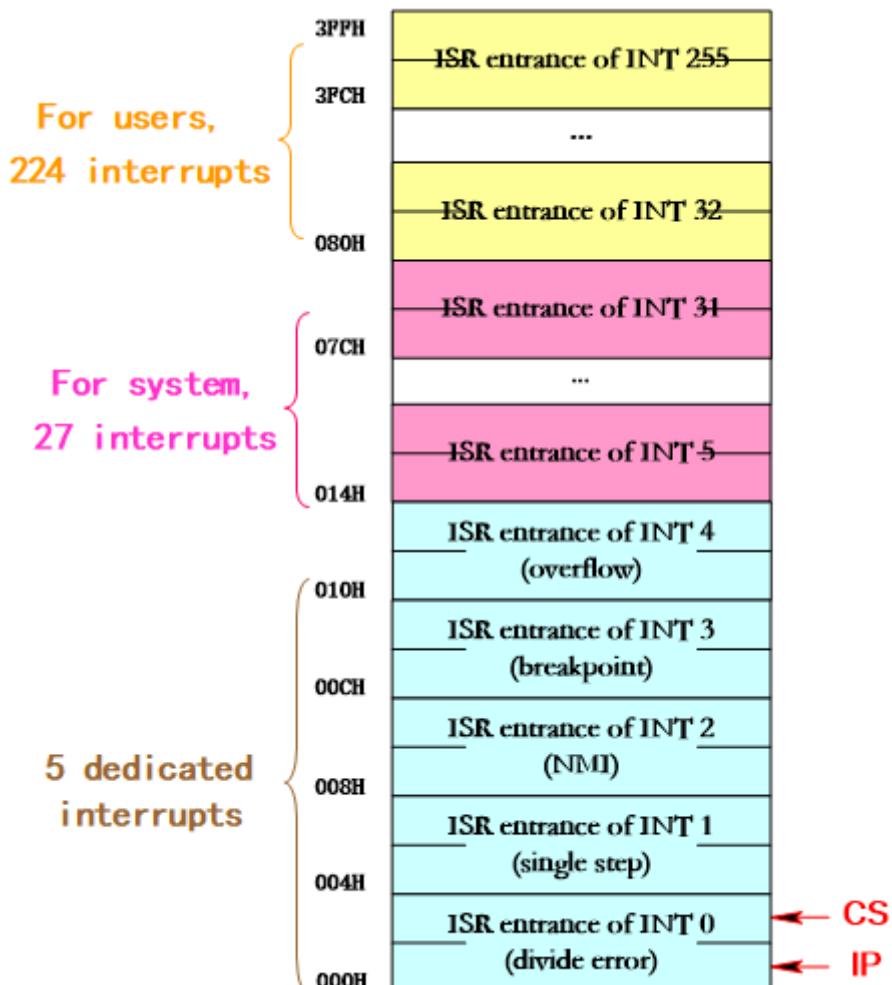
■ MOV AX,0009H
    ADD AX,0080H
    INTO

```

## 处理不可屏蔽和软件中断的过程

- 对于NMI
  - CPU检查NMI，无论是否出现IF都自动生成INT 02中断并执行ISR
- 对于软件（内部）中断
  - CPU 自动生成INT N中断并执行相应的ISR

## 8086/8088的中断向量表



## 256次中断

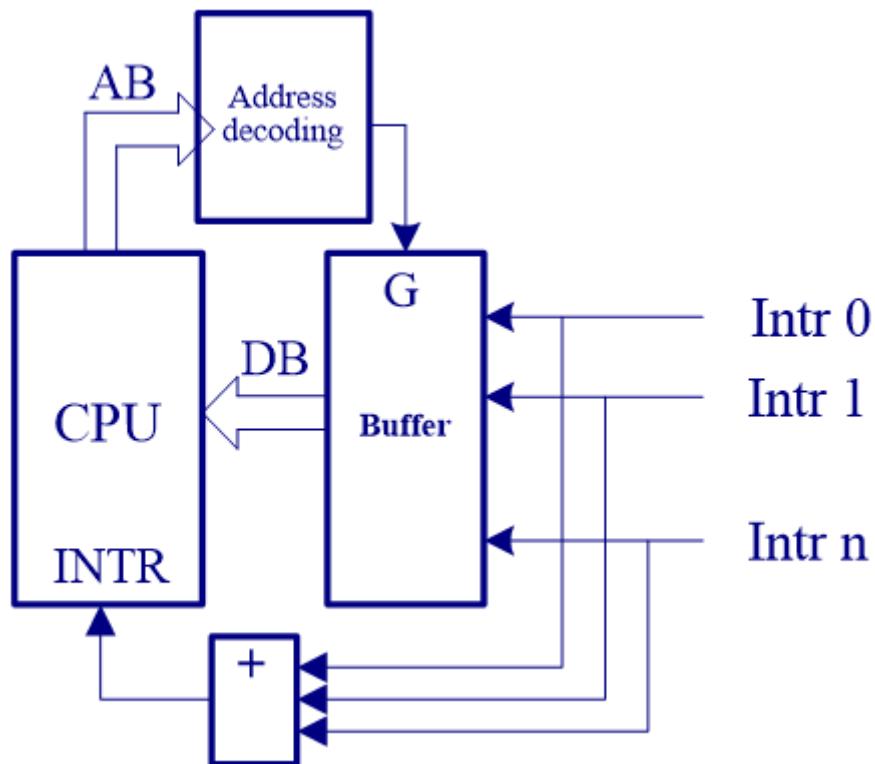
- dedicated 0~4专用
- 5~31供系统使用
  - 08H~0FH: 8259A
  - 10H~1FH: BIOS
- 32~255供用户使用
  - 20H~3FH: DOS
  - 40H~FFH: open

## 中断优先级

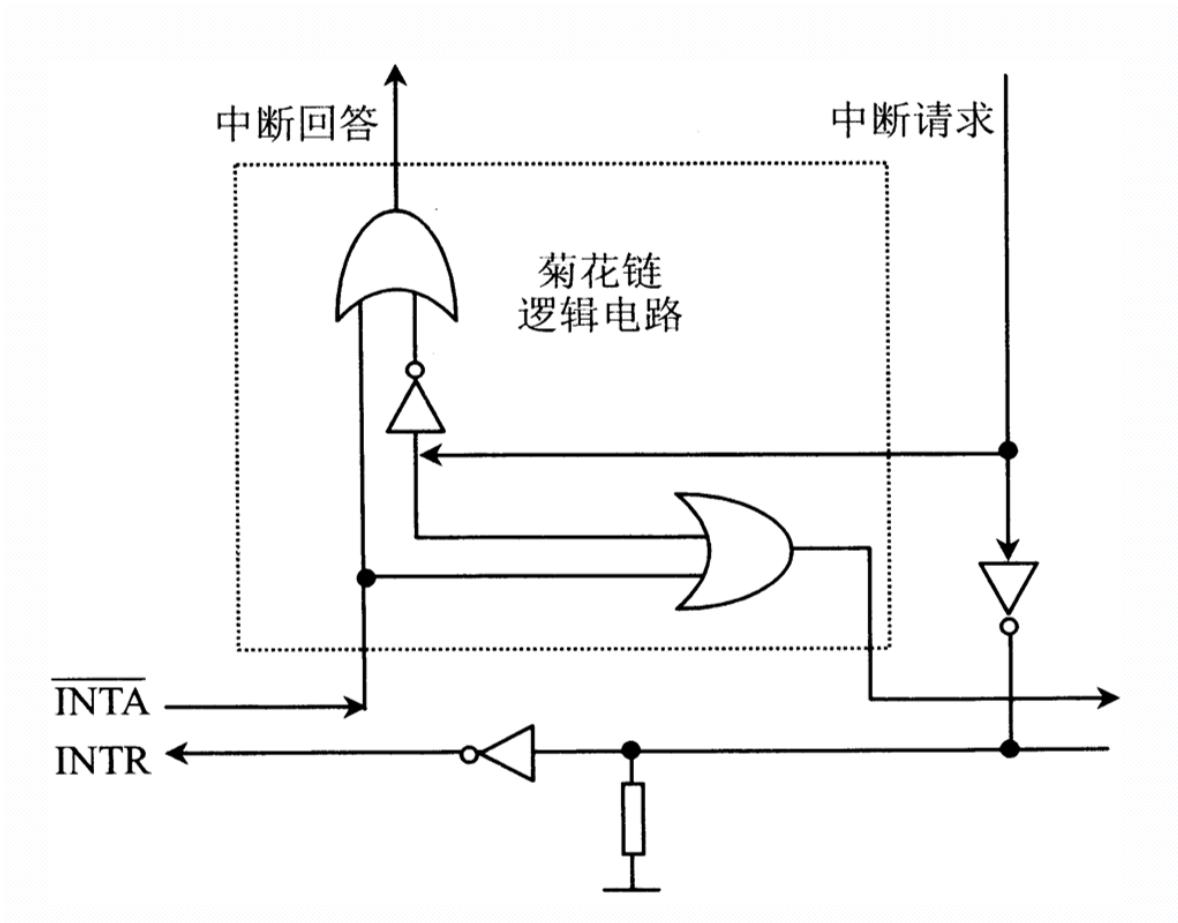
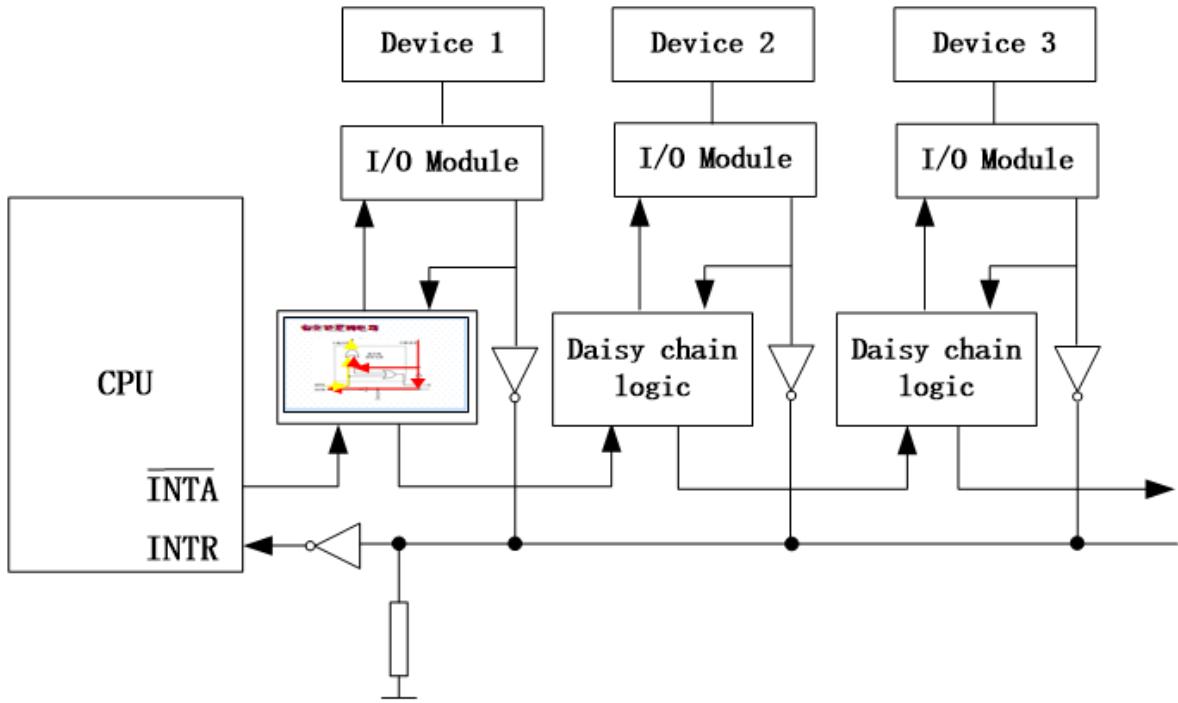
- INT指令的优先级高于INTR和NMI
- NMI的优先级高于INTR
- 对于不同的外部中断请求，可以使用不同的策略来确定其优先级。

## INTR中断的优先级

- 软件轮询
  - 跳转到通用处理程序
  - 检查顺序确定优先级



- 硬件检查
  - 菊花链中的位置很重要



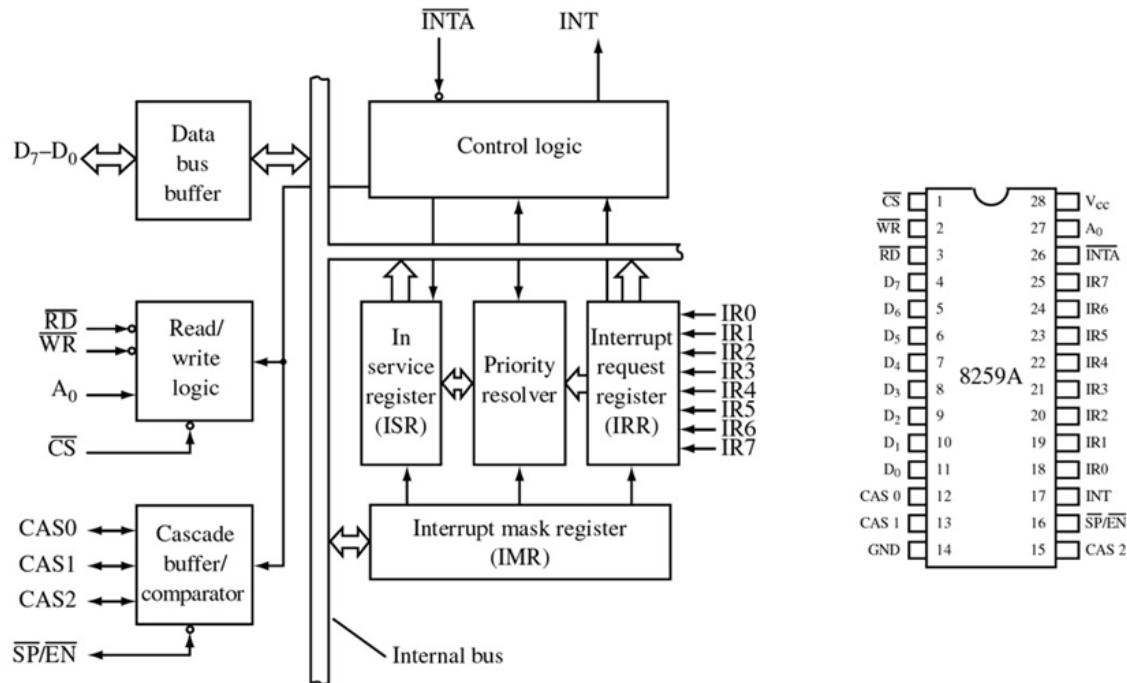
- 向量中断控制器
- 例如8259

## 8259

- 8259是可编程中断控制器 (PIC)
- 它是用于管理中断请求的工具。
- 8259是一种非常灵活的外围控制器芯片：
  - PIC最多可处理64个中断输入 级联

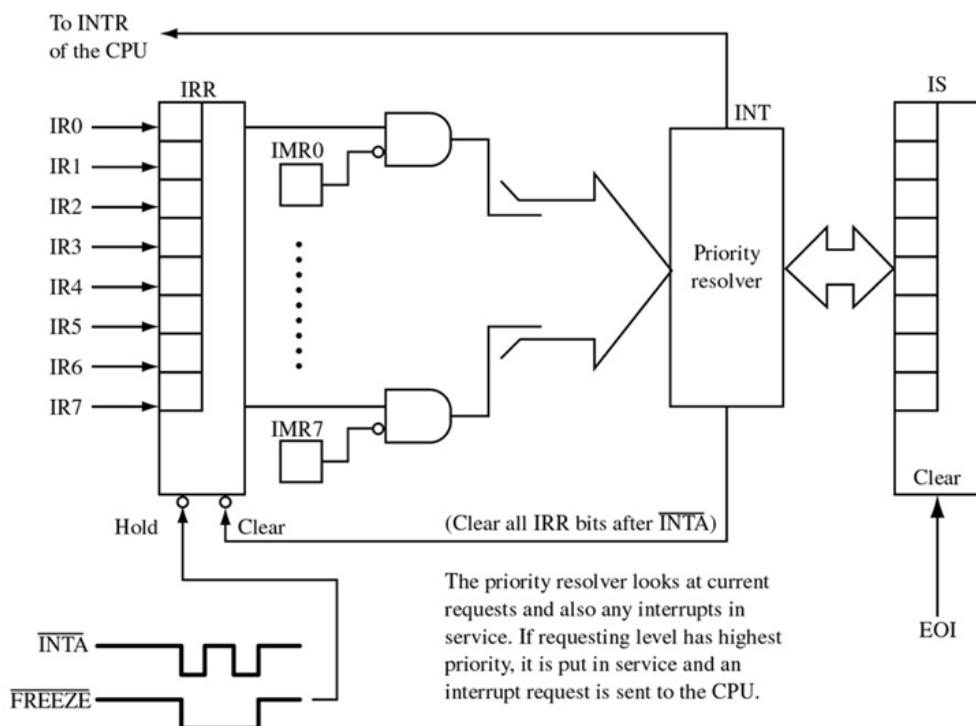
- 中断可以被屏蔽，内部对每个请求都有屏蔽位
- 各种优先级方案也可以编程。
- 最初（在PC XT中）可以作为单独的IC使用
- 后来（两个PIC）的功能在主板芯片组中。
- 在某些现代处理器中，内置了PIC的功能。

图1 8259A可编程中断控制器（PIC）的框图和引脚定义。（由英特尔公司提供）



根据IMR和ISR还有IRR综合判断是否送CPU

图2 所有中断请求都必须通过PIC的中断请求寄存器（IRR）和中断屏蔽寄存器（IMR）。如果投入使用，则会设置服务中（IS）寄存器的相应位。



IRQ0-7连接device

IMR控制是否屏蔽

priority resolver判断优先级

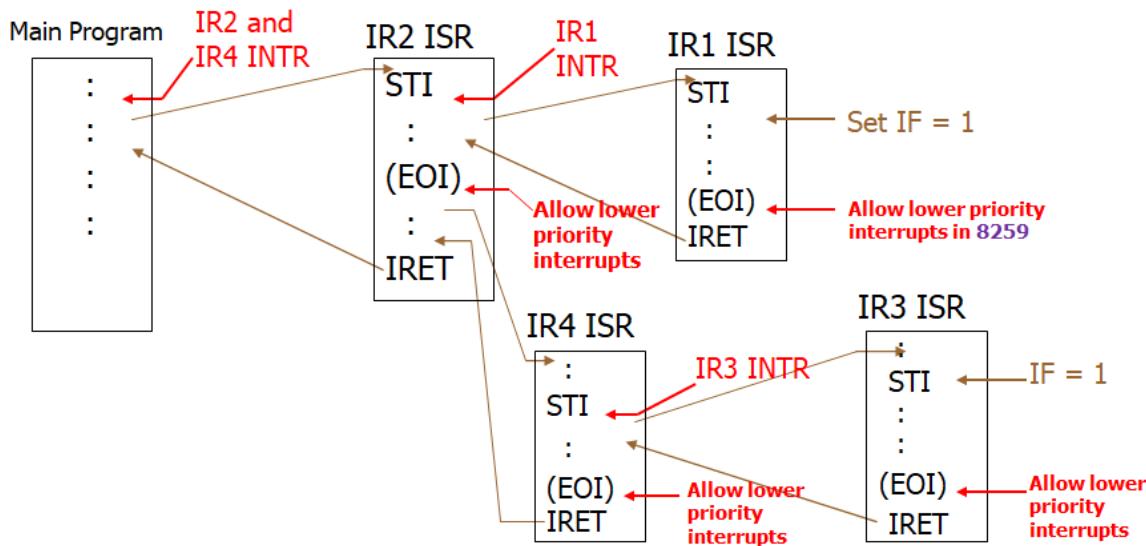
IS判断CPU正在处理的请求，判断可否抢占

INTA后clear所有IRR，IR4继续请求还是值1

IS必须在ISR里手动清除

## 中断嵌套

较高优先级的中断可以中断较低的中断



EOI end of interrupt 清除IS状态

## Ch 10 串行数据通讯和8251

### 数据通讯

- **数据传输**是点对点之间的数据传输，通常表示为物理通信通道上的电磁信号
- 一个**通信信道**是指用于从一个发送器（或发送器）向接收器传送信息的媒介。
- ○ 例如：铜线，光纤或无线通信通道。

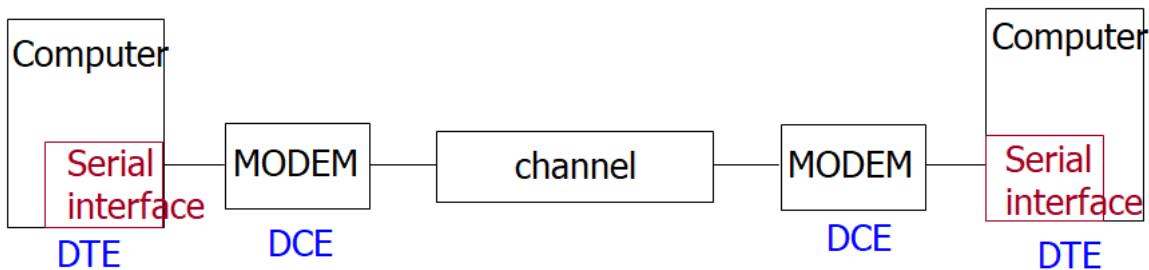
### 两种方式：并行和串行

- **并行数据传输：**
  - 每一位使用一条单独的线（线）
  - 通常使用8行或更多行
  - 除了控制信号
  - 快速，昂贵且用于短距离通信
- **串行数据传输：**
  - 一条数据线
  - 比特通过线路一一发送
  - 没有专用的控制信号线
  - 廉价&慢速&适用于远距离通讯

# 串行通讯的全貌

发送者和接收者需要一个协议来理解数据：

- 例如，如何打包数据，组成一个字符的位数，何时开始和结束数据



DTE-数据终端设备，通常是计算机。

DCE-数据通信设备，通常是调制解调器。

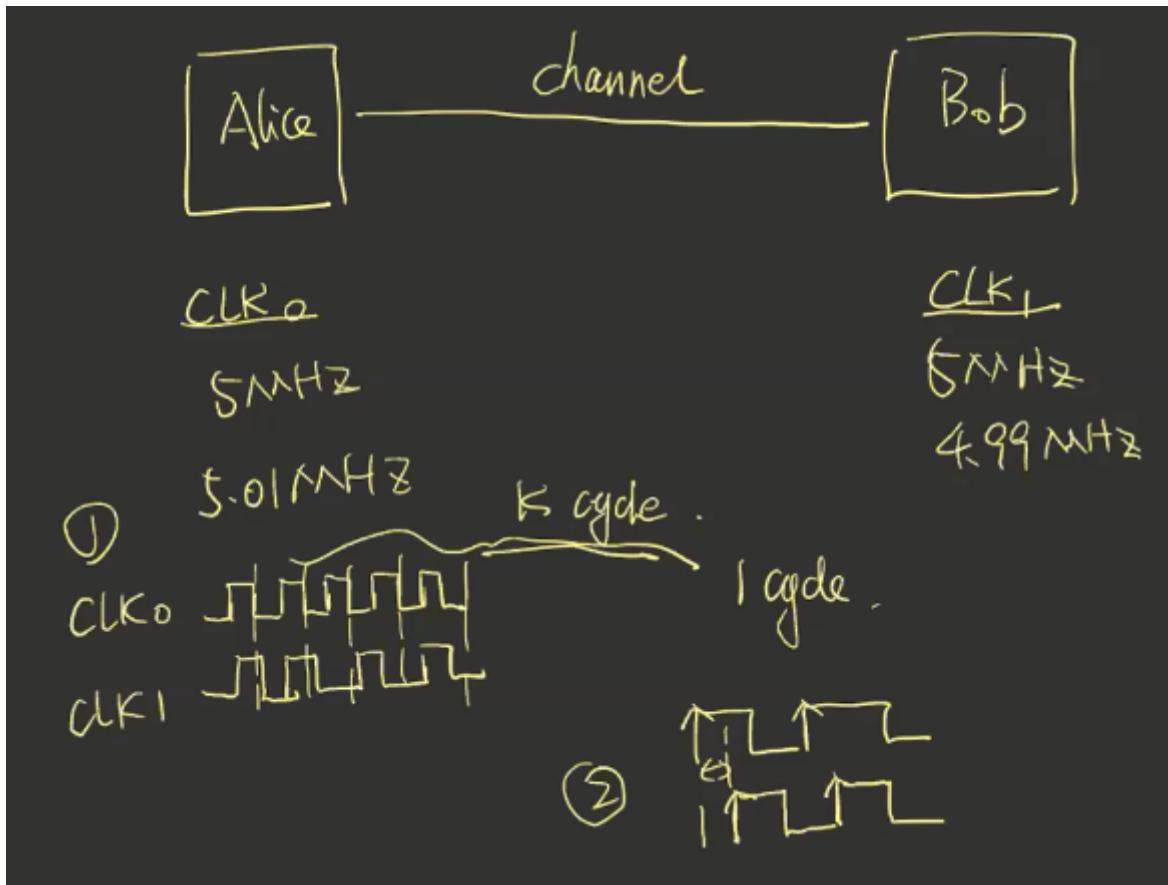
串行接口-连接DTE和DCE的IC，例如8251A, 16550和8250。

## 串行通讯

### 资料传输率

- symbol rate, 误称是波特率
- 单位时间中symbol的数量
  - 每个symbol可以表示一个（二进制编码信号）或几位数据
- 比特率
- 单位时间传输的bit数量，量化为比特每秒 (**bit/s**或**BPS**)
  - $B = S \times \log_2 N$

### 同步方法

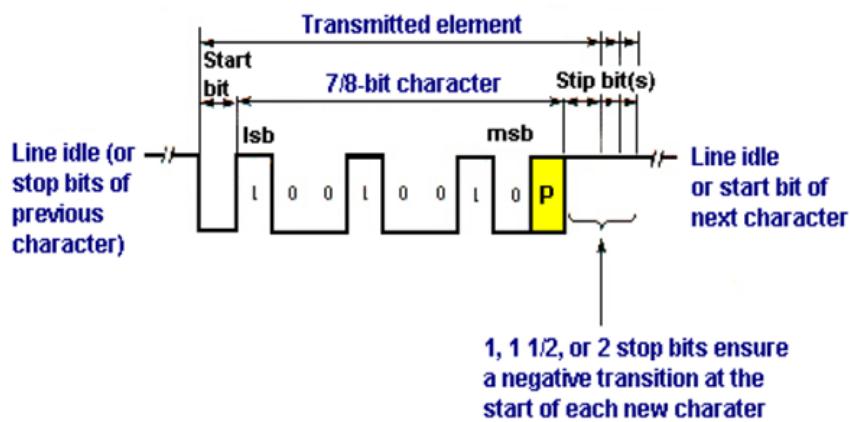


如何解决频率不同和起始时间不同两个问题？

- 异步串行通讯：
  - 通过每次少传解决问题
- ◦ 一次传输一个字节
- 每个字节的开始都是异步的，因此每个字节都需要使用start bit在发送者和接收者之间进行同步。
- 同步串行通讯：
  - 通过使用相同时钟信息来解决问题，将时钟信息嵌入发送的数据中
- ◦ 一次传输一个数据块
- 在发送数据时，使用同步字符同步发送方和接收方。

## 异步传输

- In *asynchronous transmission*, the receiver clock ( $R \times C$ ) runs *unsynchronized* with respect to the incoming signal ( $R \times D$ ).
- Each character (byte) is encapsulated between an additional **start bit** and one or more **stop bits**.
- The state of the signal on the transmission line between characters is **idle** state.

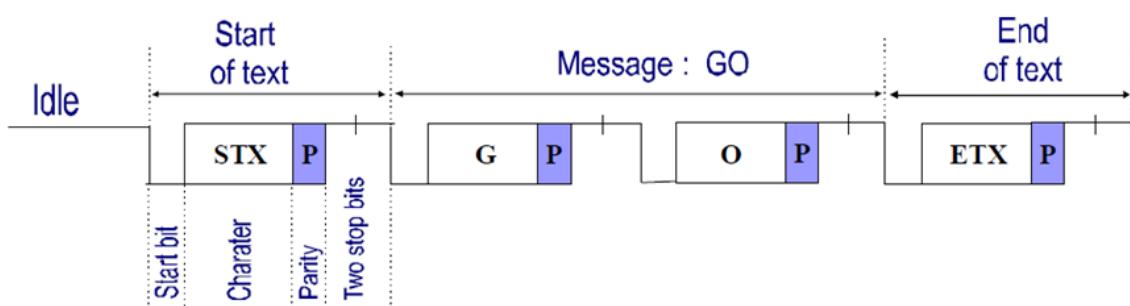


idle为高电平，start bit用低电平通知开始

每次传输7bit是一个协议

### Example:

Construct the transmitted frame using *asynchronous transmission mode* which contains the following data: GO. Assume that the number of stop bits is 2 and parity bit is used.



# Principle of Operation and Timing:

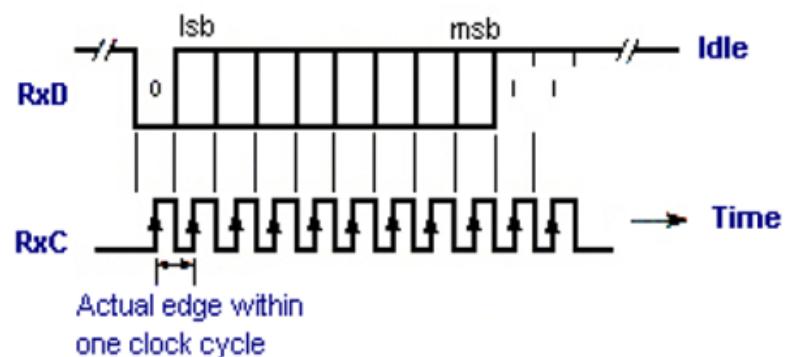
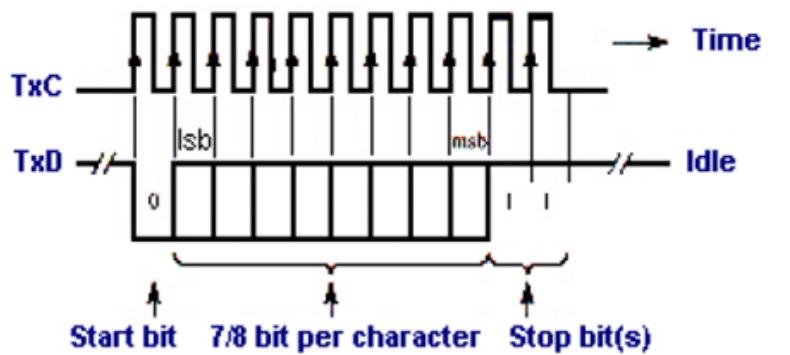


Figure 1 below (from Notes 99-2, page 7) shows the reconstruction of the pulse train using the  $a_0 +$  first 9 coefficients  $a_n$  of the Fourier series, that is, using up to the term  $a_9$ . (These coefficients are:  $a_0 = 0.25$  and  $a_1$  through  $a_9$  respectively  $0.4502 \quad 0.3183 \quad 0.1501 \quad 0.0000 \quad -0.0900 \quad -0.1061 \quad -0.0643 \quad 0.0000 \quad 0.0500$ )

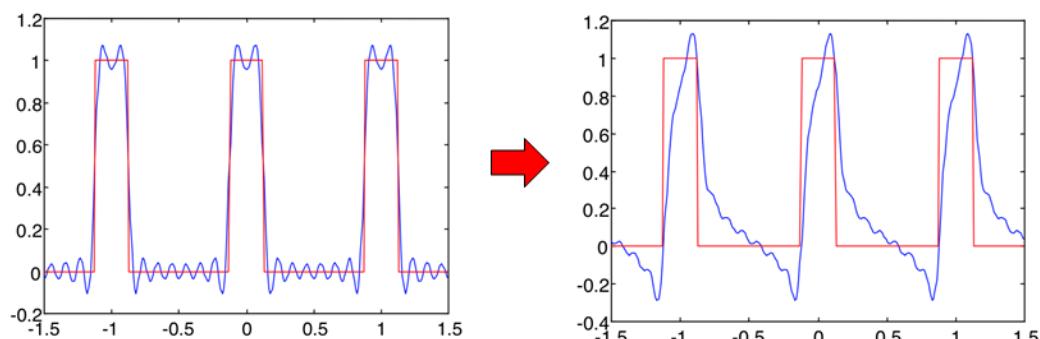


Figure 1

高频衰减导致出现偏差，所以希望在中间进行采样（但是时钟没有对齐）

## 异步传输

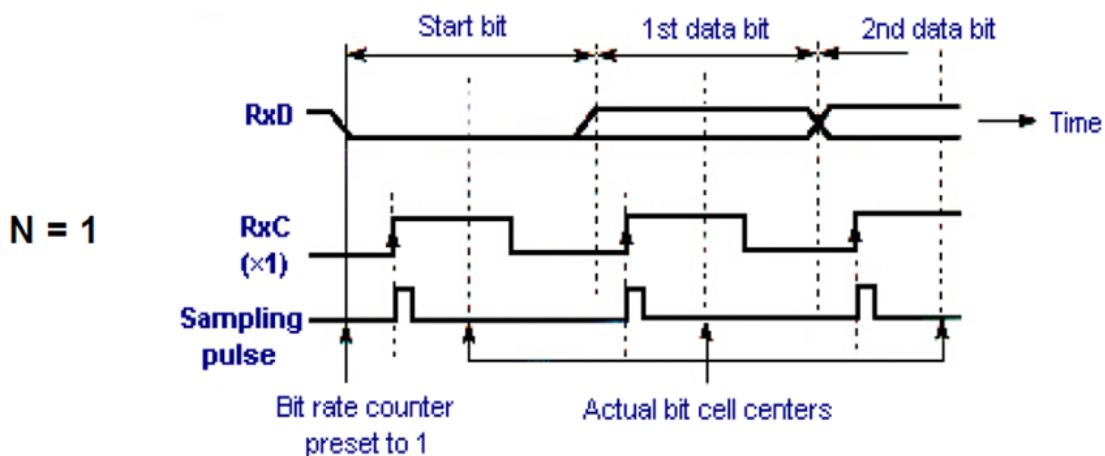
接收方提高频率

## Bit Synchronization in Asynchronous Transmission:

- The local receiver clock is  $N$  times the transmitted bit rate ( $N=16$  is common).
- The first  $1 \rightarrow 0$  transition is associated with the start bit.
- Each bit is sampled at the center to avoid delay distortion problem.
- After the first transition is detected, the signal is sampled after  $N/2$  clock cycles and then subsequently after  $N$  clock cycles for each bit in the character.

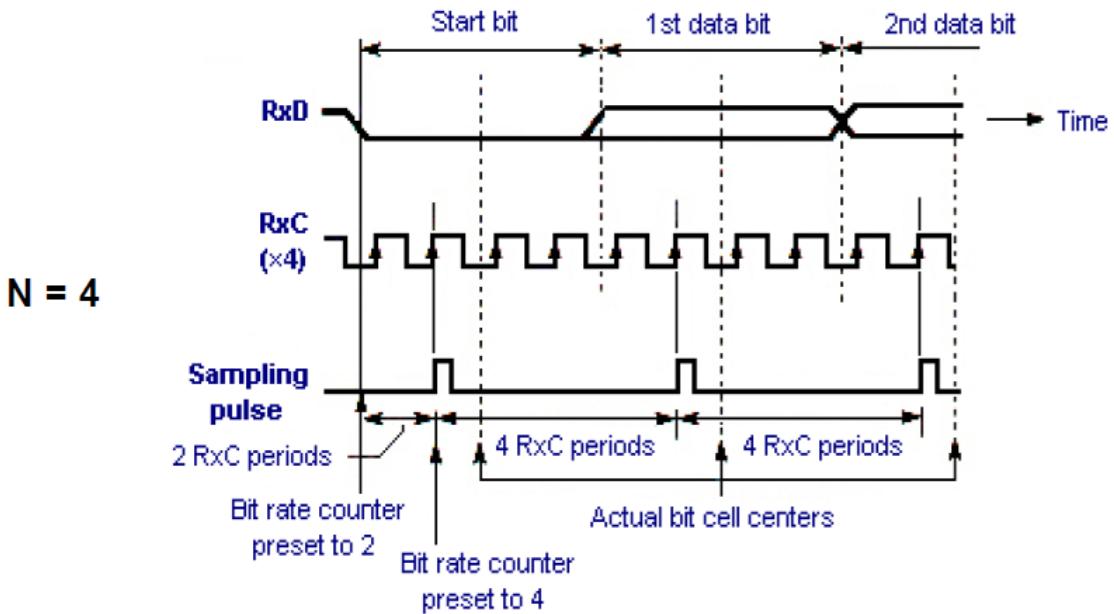
例子

## Bit Synchronization in Asynchronous Transmission:

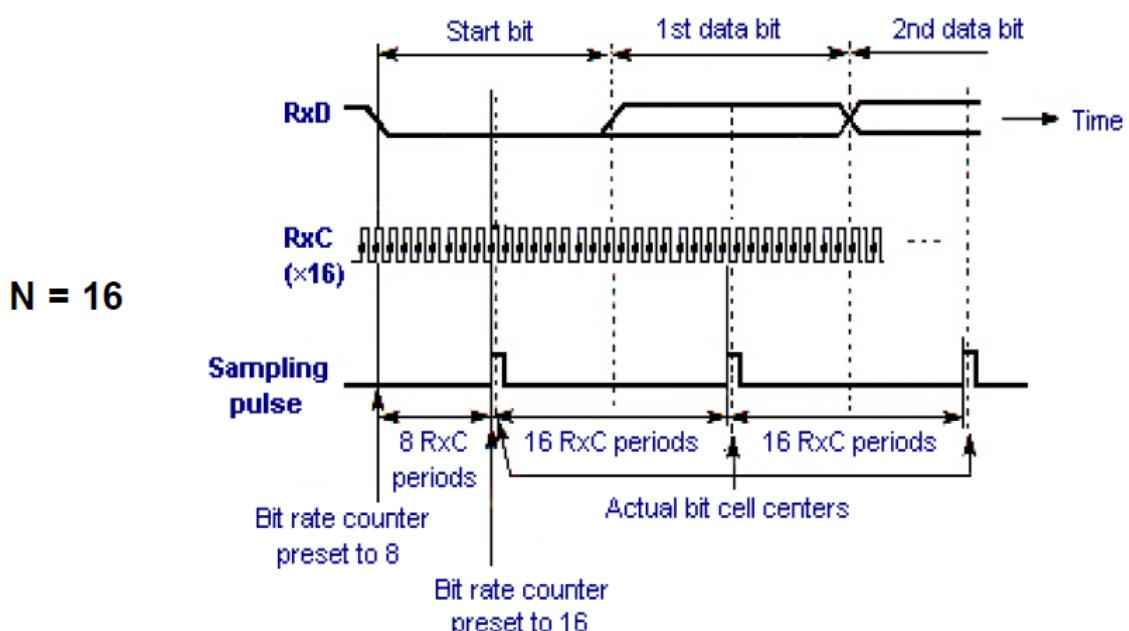


无法保证RxC起始位置在RxD中的位置

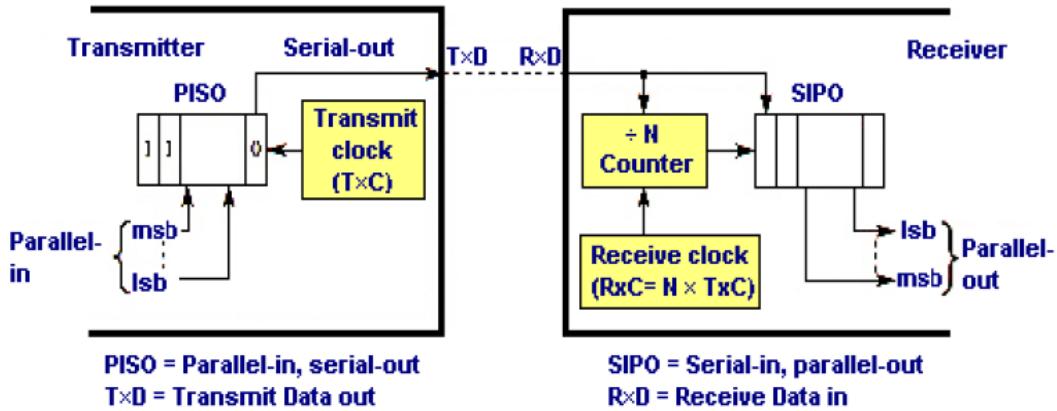
## Bit Synchronization in Asynchronous Transmission:



## Bit Synchronization in Asynchronous Transmission:



# Principle of operation and Timing:

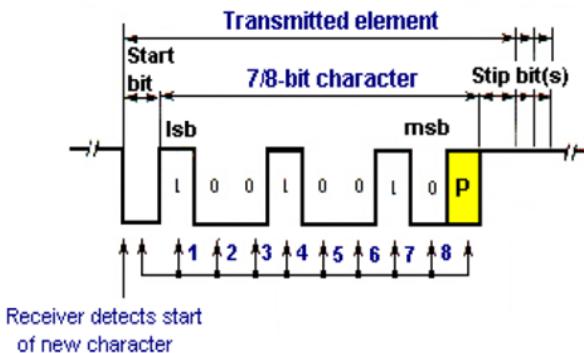


接收方如何知道数据传输结束?

用start bit和stop bit来检测开始和结束，每次传固定的bit数，作为协议。

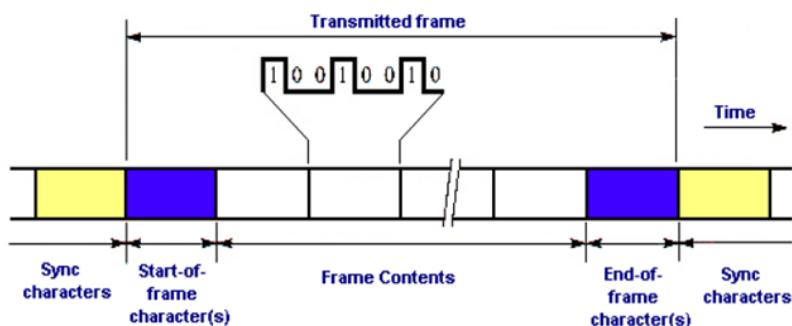
## Character Synchronization in Asynchronous Transmission:

- After the start bit is detected, the receiver achieves character synchronization simply by counting the programmed number of bits.



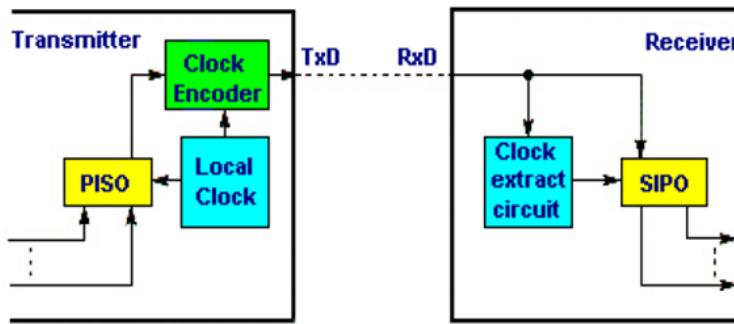
## 同步传输

- The complete block or frame of data is transmitted as a **contiguous stream** with no delay between each 8-bit element.



## Bit Synchronization using Synchronous Transmission:

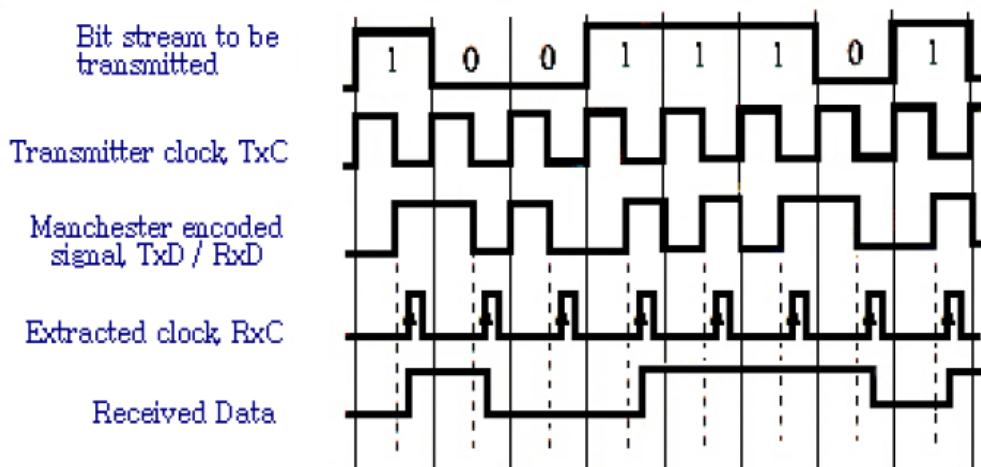
- With synchronous transmission, the receiver clock ( $RxC$ ) operates in synchronism with the received data signal ( $RxD$ ).
- Clock Encoding and Extraction:** The clock information is embedded into the transmitted signal and subsequently extracted by the receiver.



接收的时钟和接收的数据同步。将时钟嵌入至数据中

## Bit Synchronization using Synchronous Transmission:

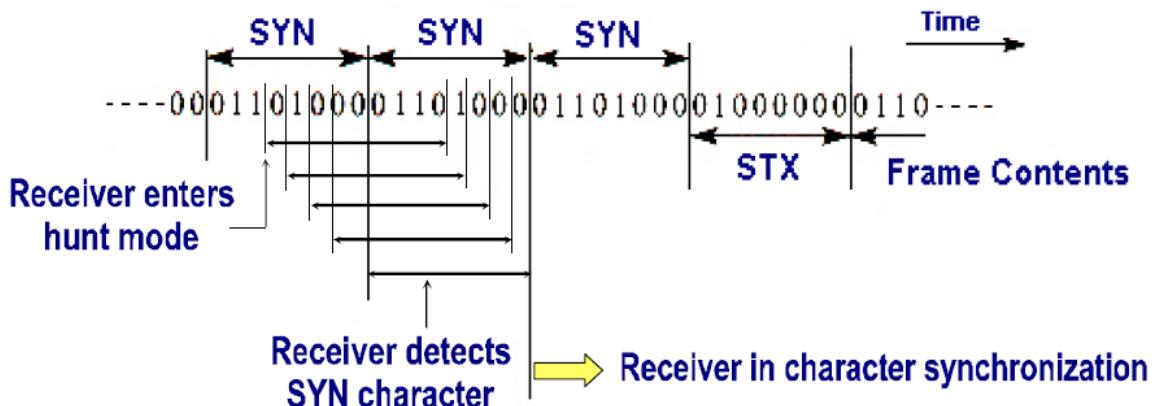
### Clock Encoding and Extraction:



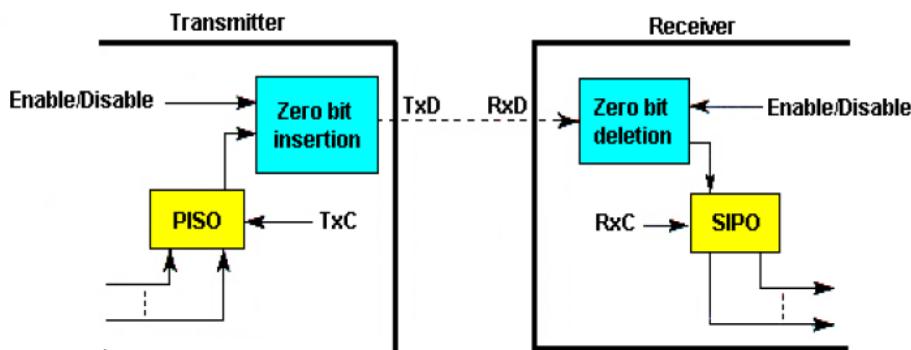
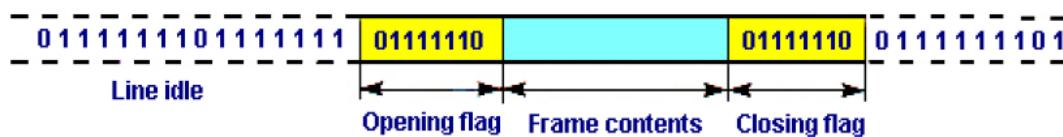
高电平上升，低电平下降进行加码。

接收方解码时钟

## 1. Character-Oriented Synchronous Transmission:

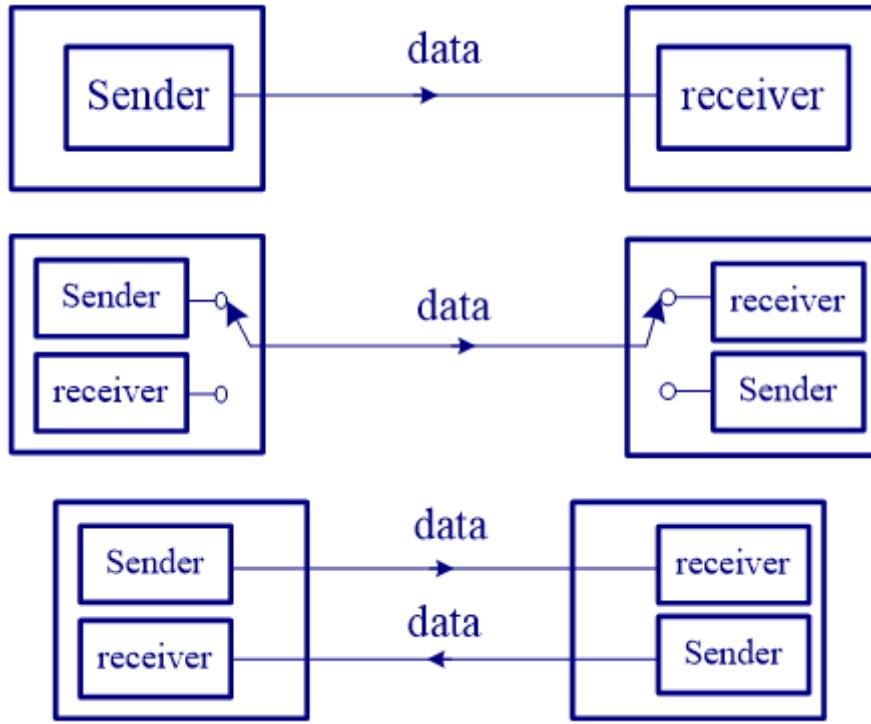


## 2. Bit-Oriented Synchronous Transmission:



位填充（或零位插入）：在五个连续的1秒后插入“0”，防止在数据段中产生01111110。接收方则在出现五个1之后删掉0

## 通讯方式



- 单工:
  - 只能单向传播
  - 例如，打印机
- 半双工:
  - 可以双向传播，但不可同时进行
  - 例如，对讲机
- 全双工
  - 数据可以同时双向传播
  - 例如，电话

## 错误检测

---

- 奇偶校验位
  - 用于异步串行通信
  - 在每个字符的末尾添加一个额外的奇偶校验位
    - 偶校验：数据和校验位的1之和为偶数；奇校验：数据和校验位的1之和为奇数；
- CRC计算

- $k$ 位数据,  $n$ 位CRC:
 
$$\frac{M(X) \times X^n}{G(X)}$$

- 例：
- 给定  $G(x) = x^3 + x^2 + 1 \rightarrow 1101$ , (取多项式的系数,  $n = 3$ )
- 如果数据为1010110, 则  $M(x) * X^n \rightarrow 1010110000$
- $CRC = 1010110000 \% 1101$  (二进制除法的其余部分, 使用XOR操作)

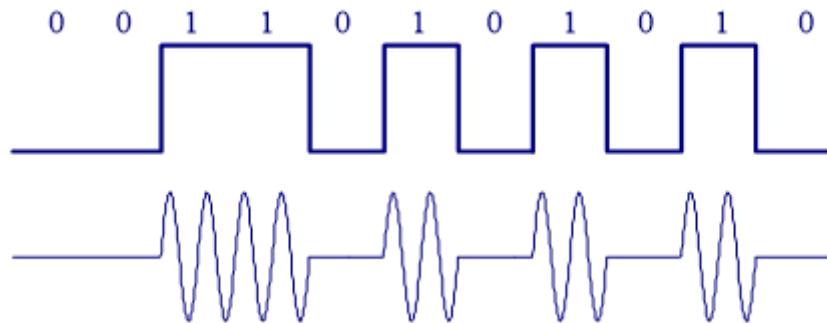
## 调制与解调

---

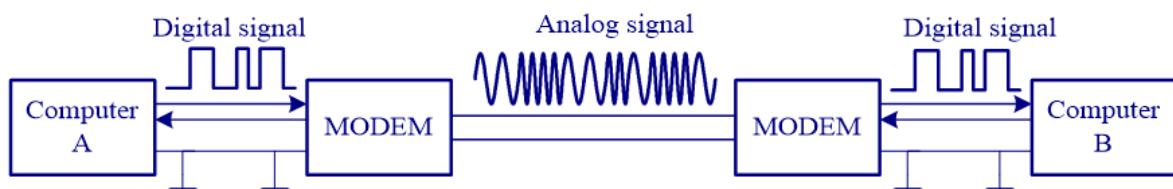
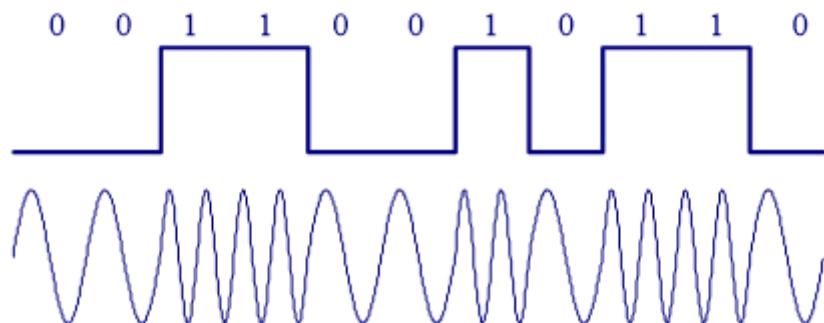
- 直接在信道上长距离传输数字信号是不合适的
  - 信号失真

- 需要调制数字信号并在发送器处获取模拟信号，并解调模拟信号并获取原始数字信号
- 载波的三个参数（幅度，频率，相位）可用于调制和解调目的
  - 调幅（AM）
  - 调频（FSK）
  - 调相（PSK）

1和0用不同的幅度表示



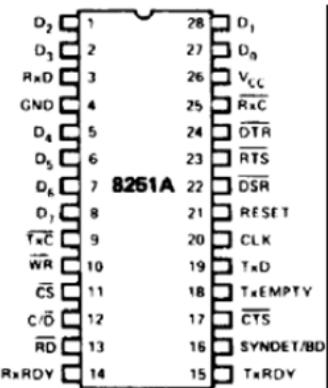
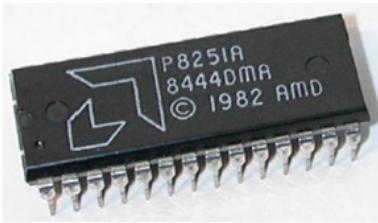
1和0用不同的频率表示



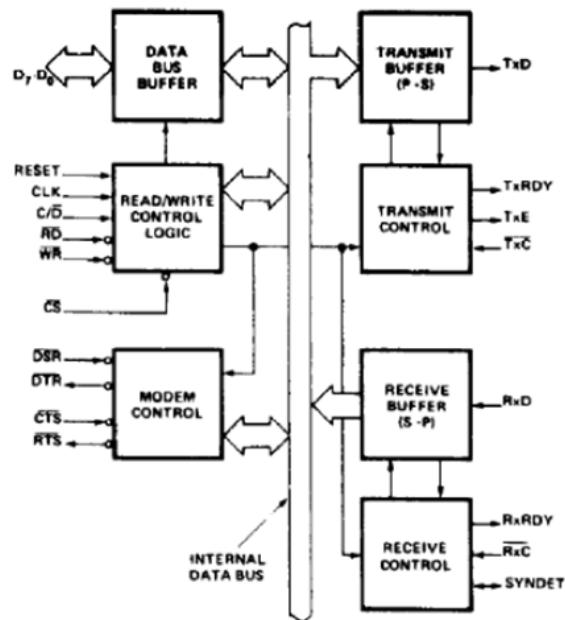
## 8251 USART芯片

- 能够进行异步和同步数据通信
- 同步：波特率0-64K，字符可以为5、6、7或8位，自动检测或插入同步字符
- 异步：波特率0-19.2K，字符可以是5、6、7或8位，自动插入起始位，停止位和奇偶校验位，TxC和RxC时钟可以是波特率的1、16或64倍
- 全双工，双缓冲
- 错误检查电路

## 8251硬件

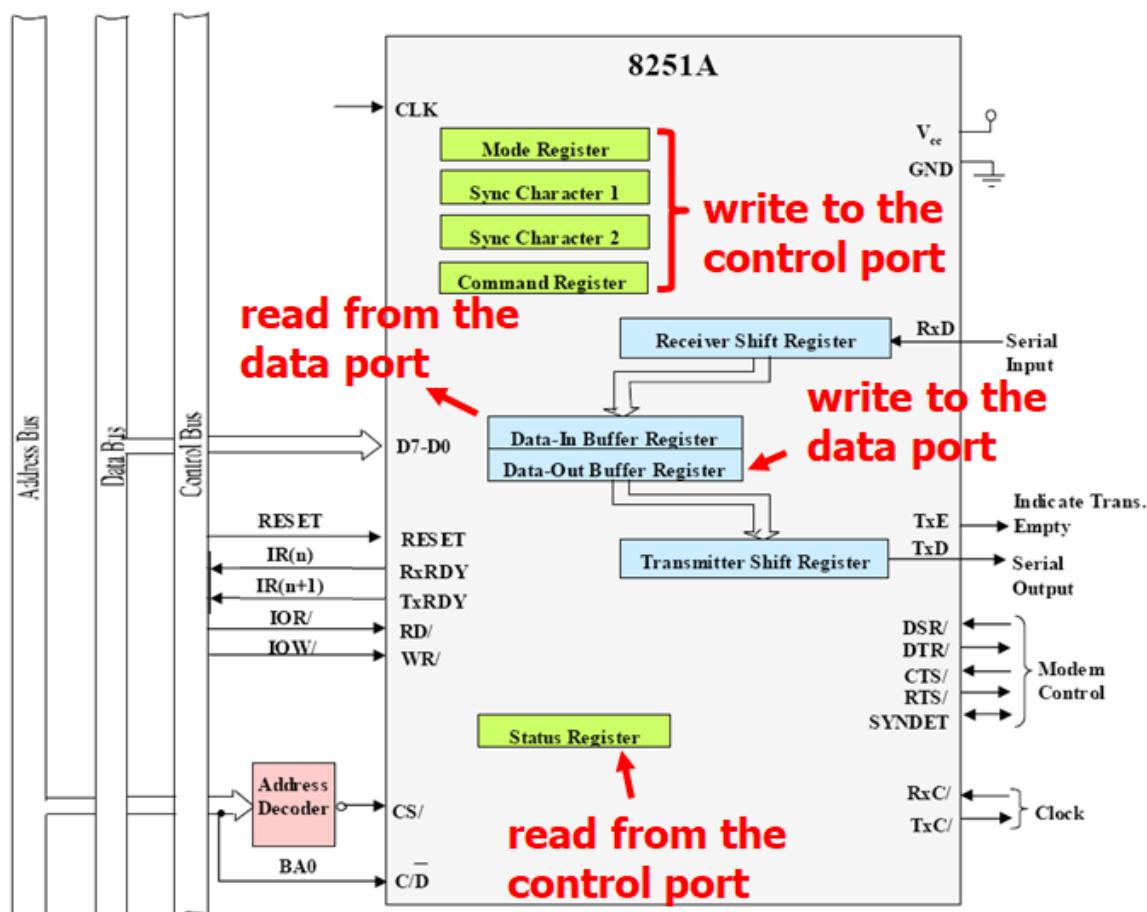


Pin Configuration

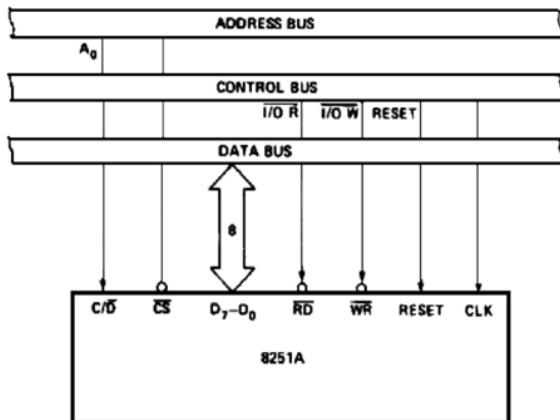


Block Diagram

## 8251通讯接口



## 与8088接口



C/D	RD/	WR/	CS/	Function
0	0	1	0	8251A DATA → DATA BUS
0	1	0	0	DATA BUS → 8251A DATA
1	0	1	0	STATUS → DATA BUS
1	1	0	0	DATA BUS → Control
X	1	1	0	DATA BUS → 3-STATE
X	X	X	1	DATA BUS → 3-STATE

## 8251信号

The 8251A is **doubled-buffered**. This means that one character can be loaded into a **data-out buffer register** while another character is being shifted out of the actual **transmit shift register**.

The **TxRDY** output of the 8251A will go high when:

- The **data-out buffer register** is Empty for another character from the CPU.
- The **CTS/** input has been asserted low.
- The **transmit-enable (TxEN)** bit of the 8251A's command word is set.

The **TxEMPTY** output of the 8251A will go high when both the **data-out buffer register** and the **transmit shift register** are empty.

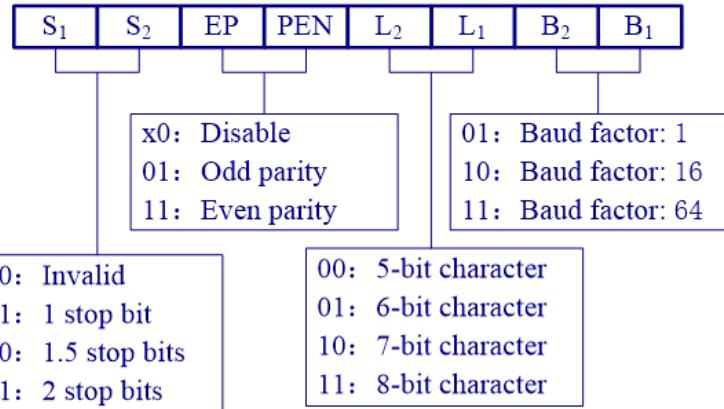
The **RxRDY** output of the 8251A will go high when:

- The **data-in buffer register** is full and is ready to be read by the CPU.
- The **receive-enable (RxEN)** bit of the 8251A's command word is set.

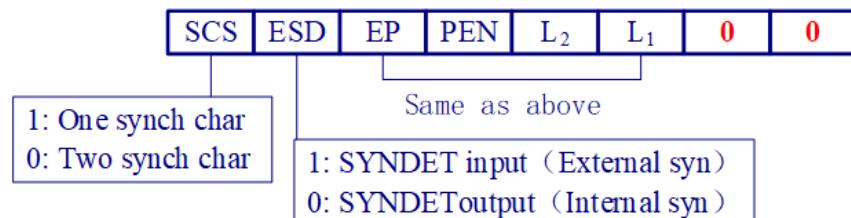
If the CPU does not read a character from the **data-in buffer register** before another character is shifted in, the first character will be overwritten and lost.

## 8251模式字

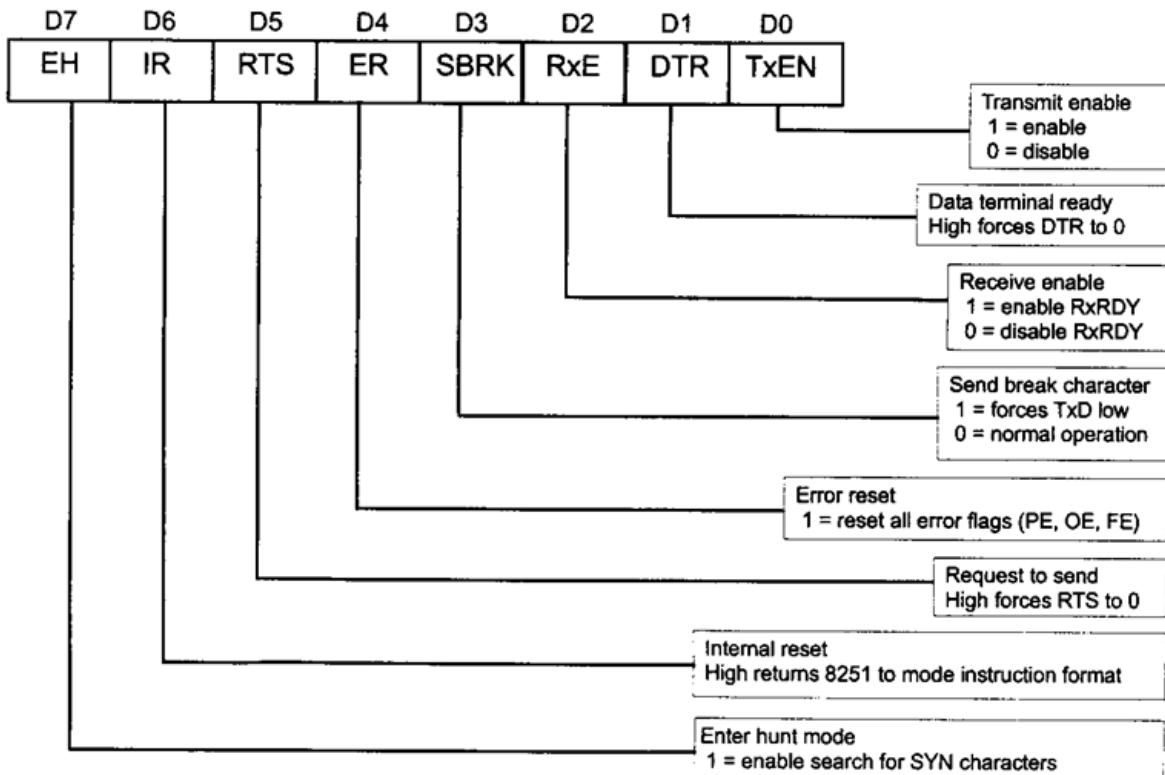
## Asynchronous



## Synchronous



## 8251命令字



Initializing the **TxEN** bit to 1 will enable the transmitter section of the 8251A and the **TxRDY** output.

Initializing **DTR** bit to 1 will cause the DTR/ output of the 8251A to be asserted low. This signal is used to tell a modem that a PC or terminal is operational.

Initializing **RxE** bit to 1 will enable the RxRDY output of the 8251A.

Initializing **SBRK** bit to 1 will cause the 8251A to output characters of 0's including start bits, data bits, and parity bits (**break character**). A break character is used to indicate the end of block of transmitted data.

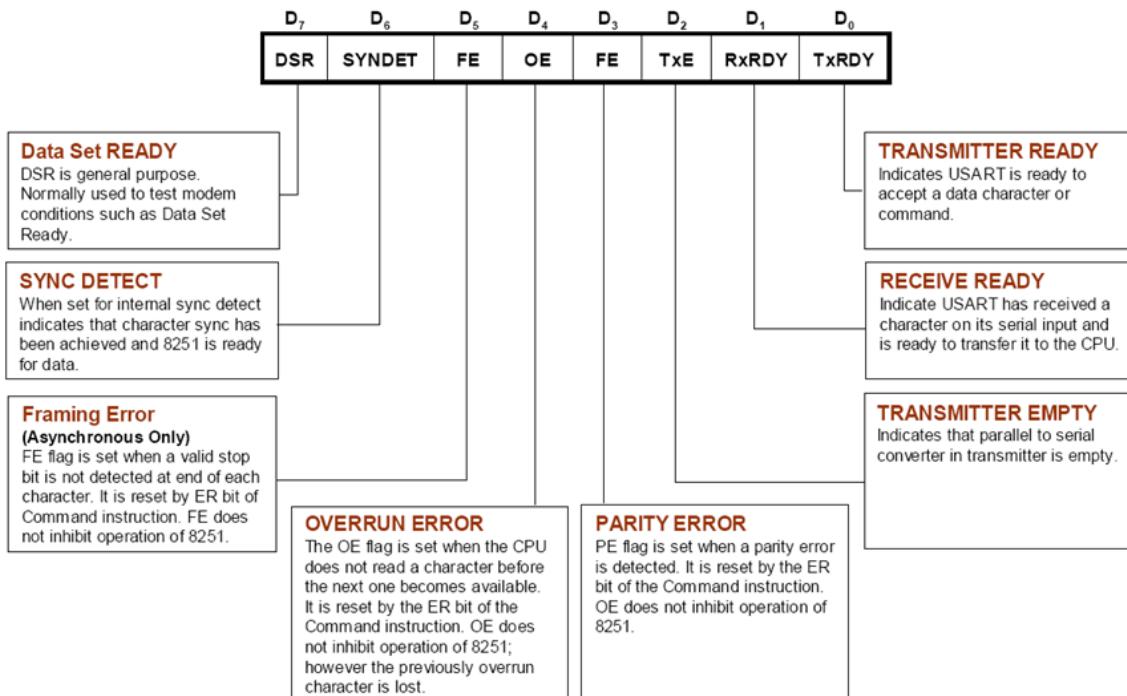
Initializing **ER** bit to 1 will cause the 8251A to reset the **parity**, **overrun**, and **framing** error flags in the 8251A status register.

Initializing **RTS** bit to 1 will cause the 8251A to assert its **request-to-send** (**RTS/**) output low. This signal is sent to a modem to ask whether a modem and the receiving system are ready for a data character to be sent.

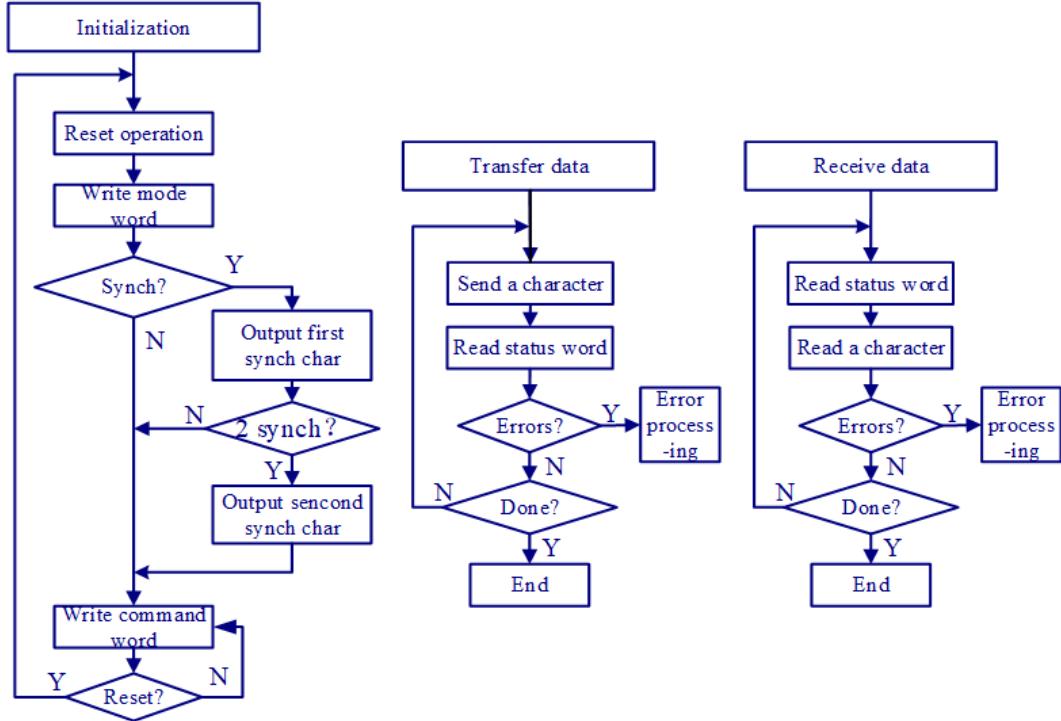
Initializing **IR** bit to 1 will cause 8251A to be internally reset. After the software- reset command, a new mode word must be sent.

Initializing **EH** bit to 1 will cause 8251A to enter hunt mode (search for **SYN** characters, and is used only in synchronous mode).

## 8251状态字



## 8251A的操作



## 8251A上电内部复位

When power is first applied, the 8251A may come up in the mode, SYN character or command format.

It is safest to execute the worst-case initialization sequence (**SYNC** mode with two **SYN** characters). Loading three 00H consecutively into the device with C#/D = 1.

An **internal reset command** (40H) may then be issued to return the device to **mode word**.

The **mode word** must then be issued, and followed by the **command word**.

## 8251编程实例

Use 8251 to transfer 256 characters in asynchronous mode, assuming that the port addresses are 208H and 209H, the baud factor is 16, and 1 stop bit, 1 start bit, no parity bit, and 8-bit character are used.

**Solution:** Sender side: data is stored in Buf1

<pre> LEA DI, Buf1 MOV DX,209H MOV AL,00H      ;worse-case init. OUT DX, AL CALL DELAY MOV AL,00H      ; OUT DX, AL CALL DELAY MOV AL,00H      ; OUT DX, AL CALL DELAY MOV AL,40H      ;reset command OUT DX, AL </pre>	<pre> MOV AL, 01001110B ; mode word OUT DX, AL MOV AL, 00110111B ; command word OUT DX, AL MOV CX, 256       ; to send 256 char. NEXT: MOV DX, 209H IN AL, DX         ; status word AND AL, 01H       ; TxRDY? JZ NEXT MOV AL, [DI] MOV DX, 208H      ; data register 208H OUT DX, AL        ; send the char. INC DI LOOP NEXT </pre>
---	---

## 8251编程实例

Receiver side: data will be stored in Buf2

<pre> Data segment buf2 DB 256 dup(?) Data ends   MOV DX,209H MOV AL,00H OUT DX, AL CALL DELAY MOV AL,40H      ; reset OUT DX, AL </pre>	<pre> MOV AL, 01001110B ; mode word OUT DX, AL MOV AL, 00110111B ; command word OUT DX, AL MOV CX, 256       ; to receive 256 char. MOV SI, 0 NEXT: MOV DX, 209H IN AL, DX         ; status word AND AL, 02H       ; RxRDY? JZ NEXT MOV DX, 208H IN AL, DX         ; receive a char MOV buf2[SI], AL INC SI LOOP NEXT </pre>
---	--

# Ch 11 Cortex-M3 / M4嵌入式系统： Cortex-M3 / M4处理器基础知识

## 嵌入式系统是什么？

一个**嵌入式系统**是计算机系统设计为与限制，例如经常执行的一个或几个专用功能，实时地以低功率或有限的存储器，...

嵌入式系统：

- 一些专用功能
- 在受限条件下

通用计算机：

- 灵活满足各种最终用户需求
- 没有如此严格的限制

## 嵌入式系统中的处理器

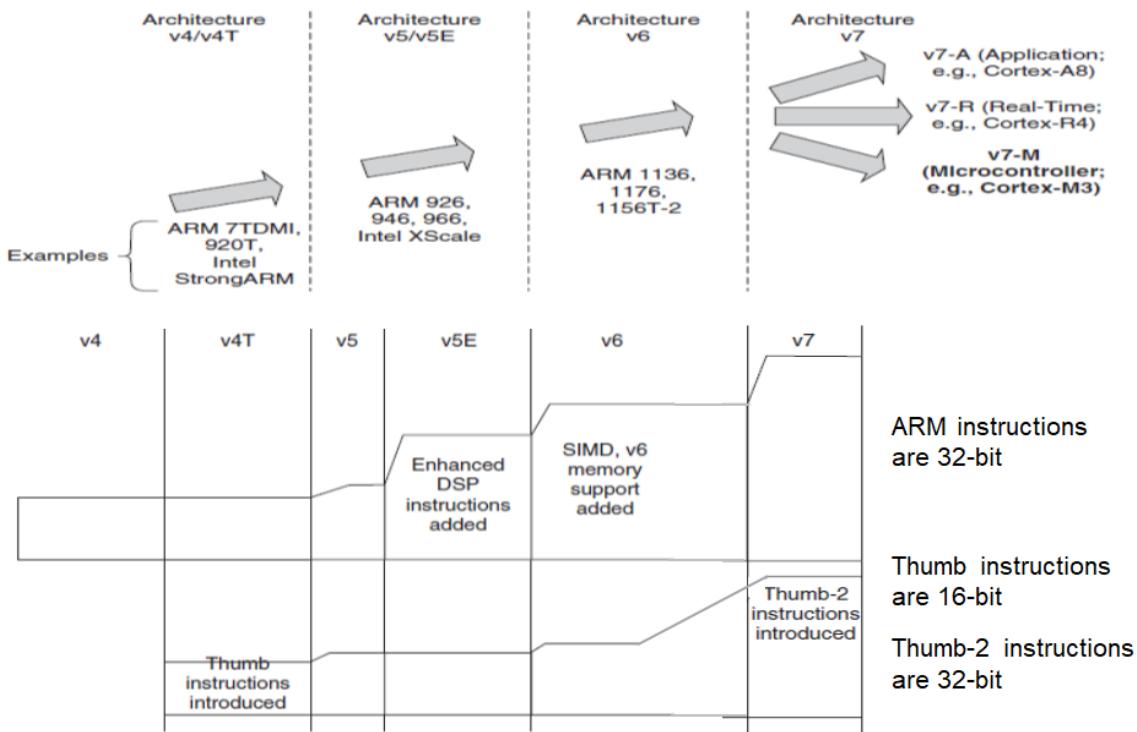
嵌入式处理器可以分为两大类：普通微处理器（μP）和微控制器（μC）。微控制器中有微处理器

- 使用了大量的基本CPU体系结构。
- - 冯·诺依曼（Von Neumann）以及哈佛架构
  - RISC和非RISC
- 字长从4位到64位不等（主要在DSP处理器中）。
- 大多数体系结构都有大量不同的变体和形状。

## 有关ARM处理器的历史

ARM公司被称为*Advanced RISC Machine*, 诞生于1990年

- ARM不生产处理器或直接销售芯片。他们将其处理器设计许可给半导体公司，这被称为**知识产权（IP）许可**。
- - 每年可能出货超过20亿个ARM处理器
  - 截至2009年，ARM处理器约占所有嵌入式32位RISC处理器的90%

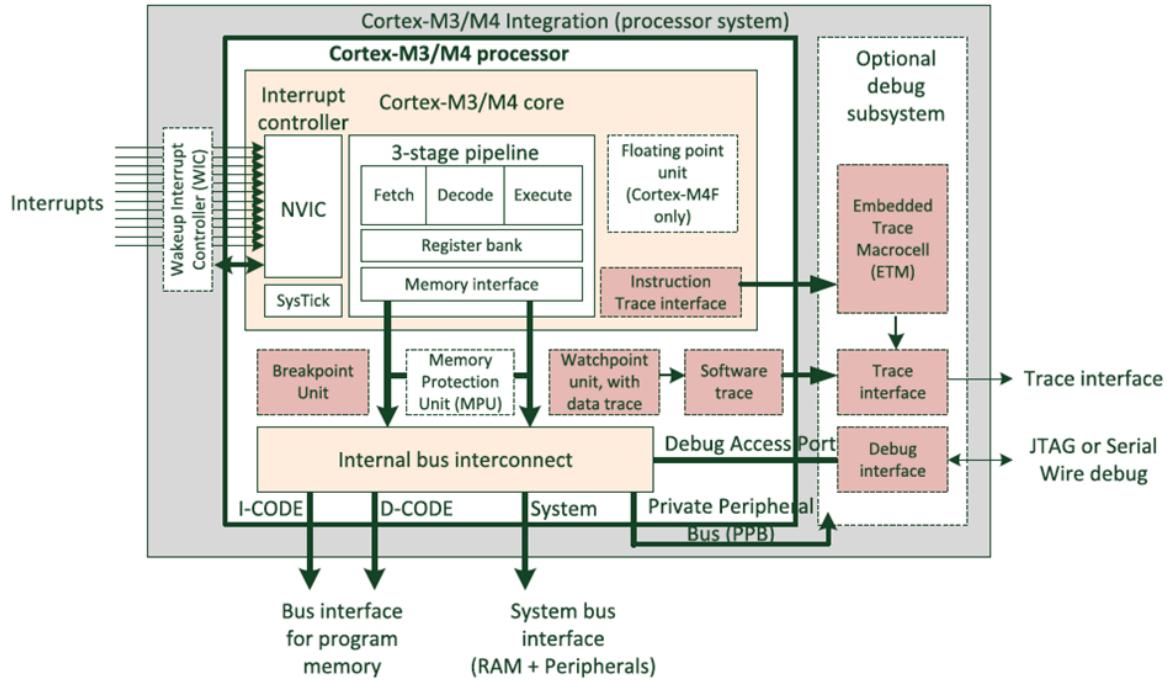


ARM指令为32位

Thumb指令为16位

Thumb-2指令为32位

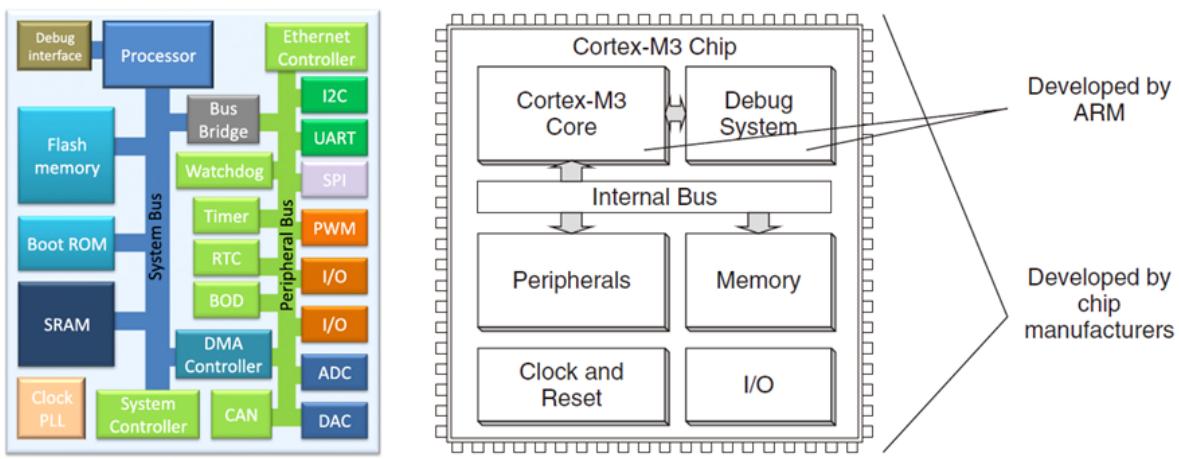
## Cortex-M3 / M4处理器概述



- 32位微处理器：**32位数据路径，32位寄存器，32位存储器接口。一个word 32位
- 哈佛体系结构：**分开的指令总线和数据总线，允许同时进行指令和数据访问。
- 4GB内存空间：** $2^{32}$
- 寄存器：**寄存器 (R0至R15) 和特殊寄存器。
- 两种操作模式：**线程模式和处理程序模式
- 两种访问级别：**特权级别和用户级别。
- 中断和异常：**内置的嵌套矢量中断控制器，支持11个系统异常以及240个外部IRQ。
- MPU：**可选的内存保护单元允许为特权访问和用户程序访问设置访问规则。
- 指令集：**Thumb-2指令集允许将32位指令和16位指令一起使用。不允许ARM指令
- 固定的内部调试组件：**提供调试操作支持和功能，例如断点，单步执行。
- M4具有其他功能：**DSP扩展和可选的单精度浮点单元
- 更高的性能效率，**无需增加频率或功率要求即可完成更多工作
- 低功耗，**延长电池寿命
- 增强的确定性，**确保关键任务和中断在已知的周期数内得到服务，实时性很强
- 改进的代码密度，**确保代码适合最小的内存
- 易于使用，**提供更容易的可编程性和调试
- 低成本解决方案，**首次将基于32位的系统成本降低到不足1美元
- 开发工具的选择范围广泛，**从低成本或免费的编译器到功能齐全的开发套件

## Cortex-M3 / M4处理器与MCU

MCU(左)本质为一片单片机，指将计算机的CPU、RAM、ROM、定时计数器和多种I/O接口集成在一片芯片上，形成的芯片级的计算机。

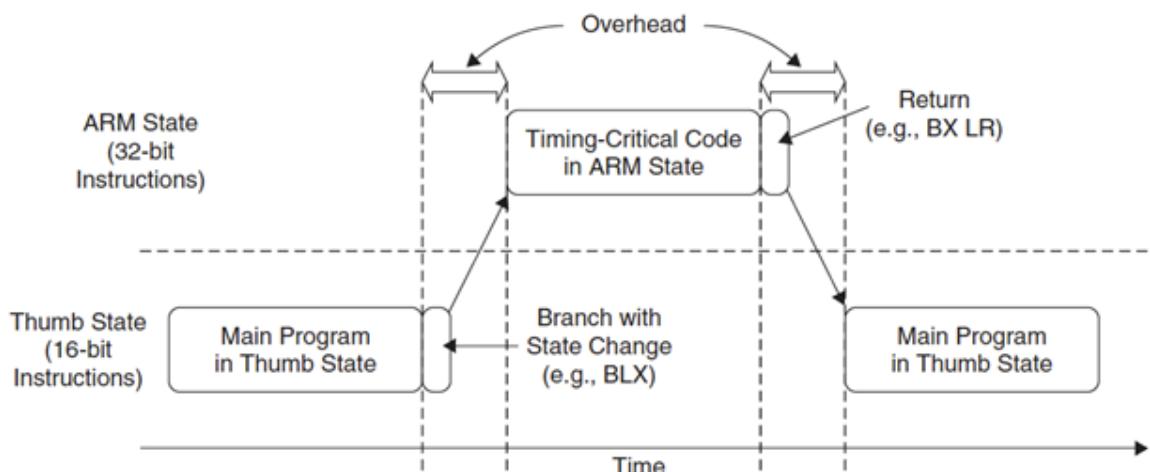


- Cortex-M3 / M4处理器是微控制器芯片的中央处理单元 (CPU)
- 芯片制造商获得Cortex-M3 / M4处理器许可后，他们可以将Cortex-M3 / M4处理器放入其芯片设计中，从而增加内存，外设，输入/输出 (I / O) 和其他功能。
- 来自不同制造商的基于Cortex-M3 / M4处理器的芯片将具有不同的内存大小，类型，外设和功能

## ARM处理器：指令状态开关

- 在ARM状态下，这些指令是32位的，并且可以以很高的性能执行所有受支持的指令
- 在Thumb状态下，指令为16位，因此指令代码密度更高

我们可以将它们结合起来以实现两全其美吗？

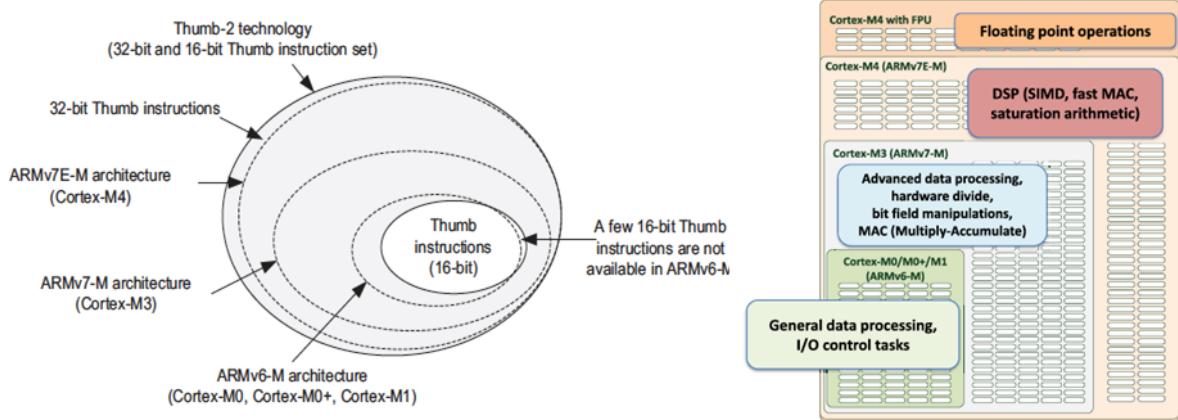


需要任何开销吗？

- 状态切换的开销
- 复杂的软件开发

## Cortex-M3 / M4指令集

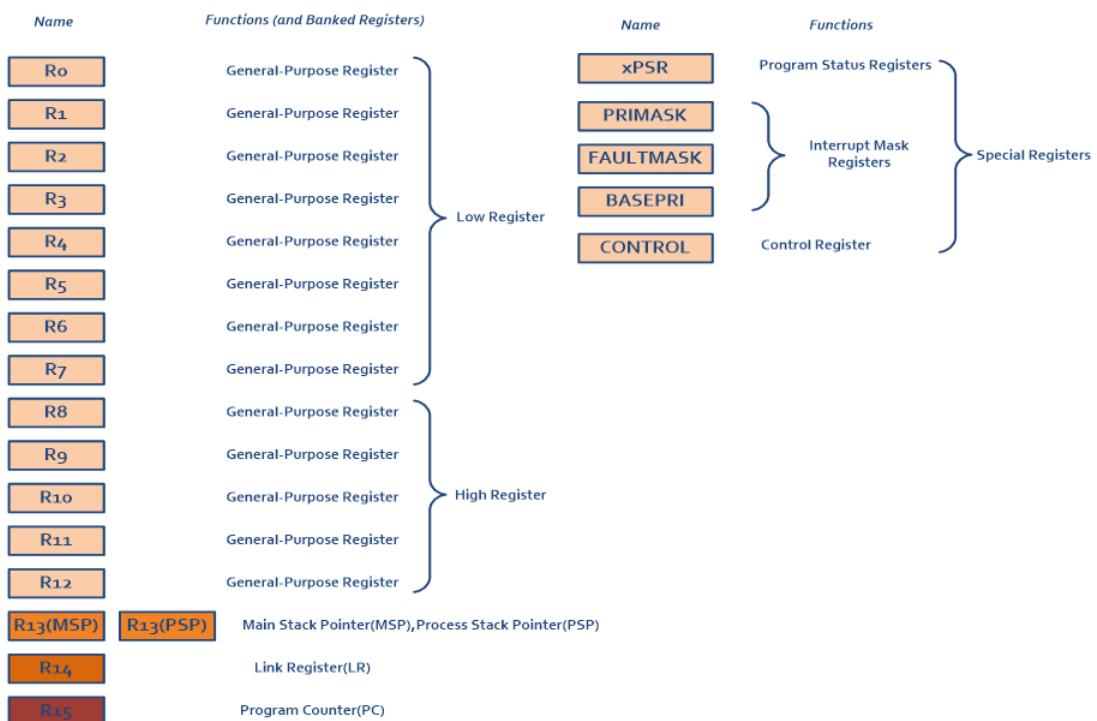
## Cortex-M3 / M4仅支持Thumb-2 (包括传统的Thumb) 指令集



好处：

- 没有状态切换开销，节省了执行时间和指令空间
- 无需分离ARM代码和Thumb代码源文件，从而使软件开发和维护更加容易

## Cortex-M3 / M4基础知识：寄存器

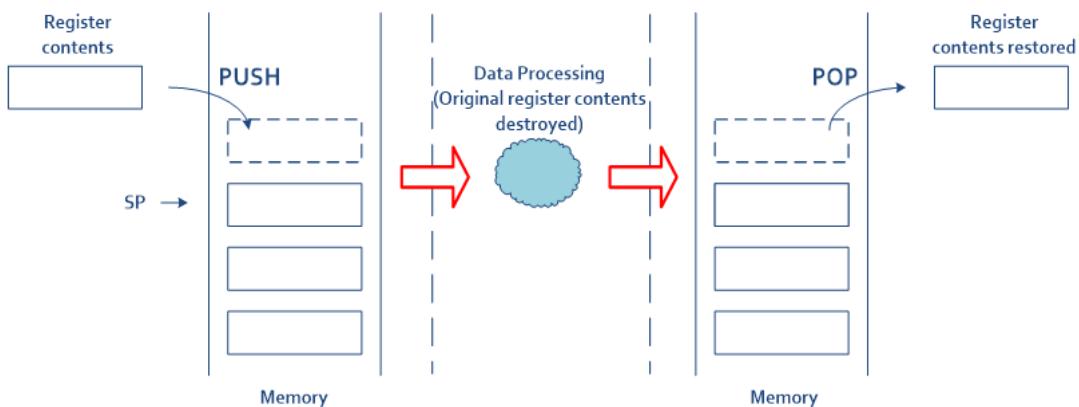


### 通用寄存器

- **R0~R7 (低位寄存器)**
  - 可由所有16位Thumb指令和所有32位Thumb-2指令访问
  - 重置值不可预测
- **R8~R12 (高位寄存器)**
  - 所有Thumb-2指令均可访问，但并非所有16位Thumb指令均可访问
  - 重置值不可预测
- **堆栈指针R13**
  - 两个堆栈指针被存入bank，以便每次只有一个可见。
  - 主堆栈指针 (**MSP**)：这是默认堆栈指针，由OS内核，异常处理程序和特权模式程序使用
  - 进程堆栈指针 (**PSP**)：由用户应用程序代码使用

## 堆栈：先进先出缓冲区

- 堆栈指针用于通过**PUSH**和**POP**访问堆栈存储器
- 例如，在子例程开始时将寄存器内容存储到堆栈存储器中，然后在子例程结束时从堆栈中恢复寄存器。
- Stack PUSH operation to back up register contents**
- Stack POP operation to restore register contents**



- Cortex-M3使用**全降**堆栈结构
  - 当将新数据推入堆栈时，堆栈指针递减
  - 汇编语句语法如下：

```
PUSH {R0} ; R13=R13-4, then Memory[R13]=R0  
POP {R0} ; R0=Memory[R13], then R13=R13+4
```

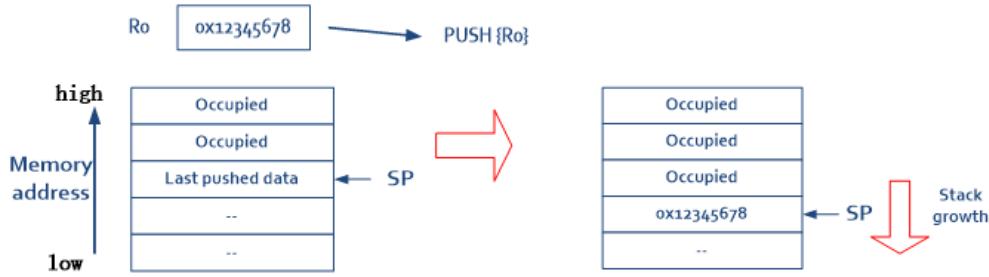
- 要么**R13**或**SP**可以在程序代码中使用（参照正在使用的当前堆栈指针，无论是MSP或PSP）
- 可以使用特殊的寄存器访问指令（**MRS / MSR**）访问特定的堆栈指针（**MSP / PSP**）
- 由于PUSH和POP操作始终按**字对齐**，因此堆栈指针R13的位0和位1硬连线为零，并且始终读为零（RAZ）
- 可以在一条指令中使用逗号将多个寄存器压入并弹出，以逗号分隔

```
PUSH {reglist} ; push the largest numbered register first  
POP {reglist} ; pop the lowest numbered registers first
```

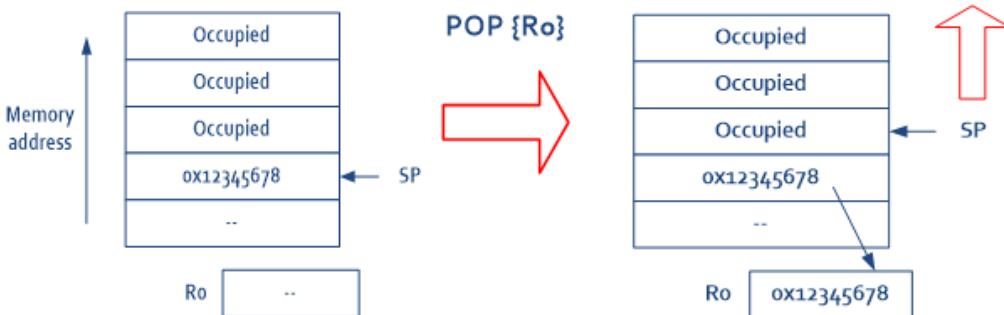
- 如果POP指令的注册表中包含PC，则在POP指令完成后执行到该位置的跳转。
- 请注意，用于设置PC的值的位[0]必须为1（用于更新APSR T位）

```
PUSH {R0-R7, LR} ; Save registers  
...  
POP {R0-R7, PC} ; Restore registers
```

- 堆栈指针（SP）指向最后被压入堆栈存储器的数据，并且在新的**PUSH**操作中，SP会首先递减。



- 对于**POP**操作，将从SP指向的内存位置读取数据，然后递增堆栈指针。存储器位置中的内容不变。



- 链接寄存器R14

- R14**是链接寄存器 (LR)，用于在调用子例程或函数时存储返回程序计数器。
- 例如，当使用**BL** (带链接的分支) 指令时：

```

main      ; Main program
...
    BL  function1 ; Call function1 using Branch with Link
                  ; instruction.
                  ; PC = function1 and
                  ; LR = the next instruction in main
...
function1
...
    BX  LR       ; Program code for function 1
                  ; Return
  
```

- 程序计数器R15

- 当您读取该寄存器时，由于处理器的流水线处理，您会发现该值与执行指令的位置相差4。

0x1000 : MOV R0, PC ; R0 = 0x1004

- 读取PC时，LSB (位0) 始终为0。为什么？

- 写入PC时，将导致分支。必须将LSB设置为1才能指示Thumb状态操作（设置为0表示切换到ARM状态，这将导致Cortex-M3中出现故障异常）
- 可以在8086中写入PC吗？由于所有指令都是半字或字对齐的，为什么我们需要设置R15的LSB (R14为true) ?

- 特殊寄存器

- Cortex-M3处理器中的特殊寄存器包括：

- 程序状态寄存器 (PSR)
- 中断屏蔽寄存器 (PRIMASK, FAULTMASK和BASEPRI)
- 控制寄存器 (CONTROL)
- 只能通过MSR和MRS指令访问

- MRS <reg>, <special\_reg> ; Read special register  
MSR <special\_reg>, <reg> ; write to special register

- 注意：** MSR和MRS不能具有内存地址，仅允许寄存器

- 程序状态寄存器 (PSR)

- 程序状态寄存器分为三个状态寄存器：  
1.applicationPSR (**APSR**) , 2.中断PSR (**IPSR**) , 3.执行PSR (**EPSR**)

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR																Exception Number
EPSR						ICI/IT	T			ICI/IT						

- 当它们作为一个集合项被访问时，将使用名称xPSR (程序代码中使用的**PSR**) 。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT						Exception Number

- EPSR**和**IPSR**是只读的：

- MRS R0, APSR ; Read Flag state into R0  
MRS R0, IPSR ; Read Exception/Interrupt state  
MRS R0, EPSR ; Read Execution state  
MSR APSR, R0 ; Write Flag state

- 访问xPSR：

- MRS R0, PSR ; Read the combined program status word  
MSR PSR, R0 ; Write combined program state word