- 总线
  - 数据总线
  - 控制总线
  - 地址总线
    在 PC 中，CPU 通过总线与其他器件连接/进行控制（通过拓展卡槽），并不能直接控制

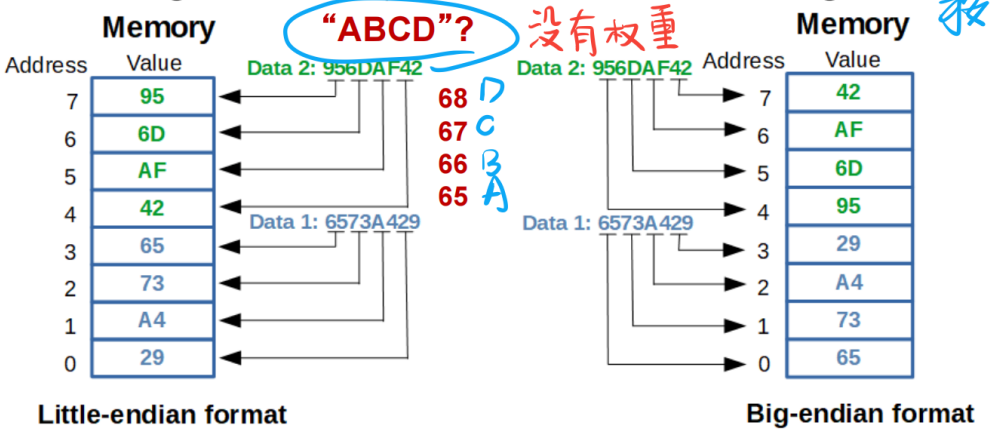
> “
>
> - 这里的一个word指的是两个字节！
> - 课件里的例题很重要
> - 101012模式
> - 指令前缀 - REX 理解什么时候启用
> - 中断向量
> - 特权级别（考的不多）

Ch 1

- Intel 是 little-endian format
  - string 没有权重，直接按照从左到右的顺序 无论大小端

Word-Sized Data

- A word (16-bits) is formed with two bytes of data. The data is stored in little-endian format:
  - The least significant byte always stored in the lowest-numbered memory location.
  - Most significant byte is stored in the highest.



数制转换

- **Example: Octal to Binary to Hexadecimal**

$$6\ 3\ 5\ .\ 1\ 7\ 7_{8}$$

Restate : $110|011|101\ .\ 001|111|111_{2}$

Regroup: $1|1001|1101\ .\ 0011|1111|1(000)_{2}$

Convert : $1\ 9\ D\ .\ 3\ F\ 8_{16}$

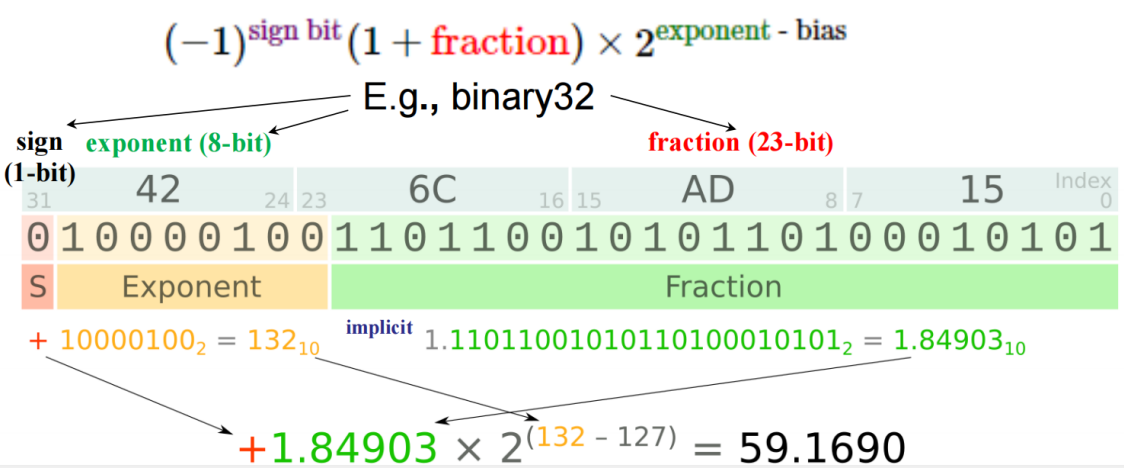
BCD

- Stored in packed BCD form:
  - packed BCD data stored as two digits per byte;
  - used for BCD addition and subtraction in the instruction set of the microprocessor or BCD counting
- Stored in unpacked BCD form:
  - unpacked BCD data stored as one digit per byte
  - returned from a keypad or keyboard

Decimal	packed BCD	unpacked BCD
12	0001 0010	0000 0001 0000 0010
96	1001 0110	0000 1001 0000 0110

IEEE 754 标准

- 32 bit
  - 1
  - 8
  - 23



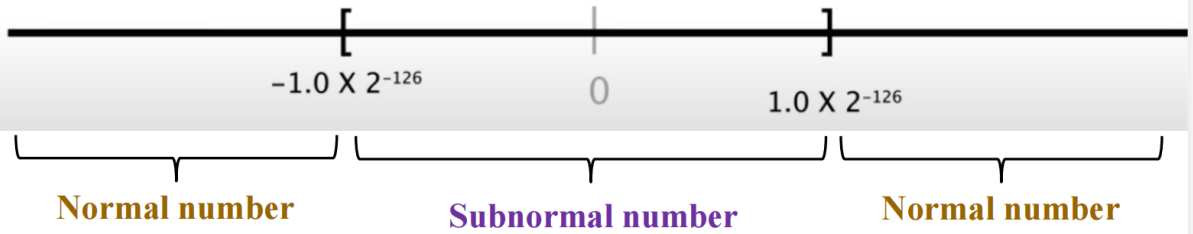
- special value

- Special values with all 0s or 1s in the exponent field:
  - Zero: sign bit = 0 and 1 for +0 and -0; biased exponent = all 0 bits; and the fraction = all 0 bits;
  - Infinity: sign bit = 0 and 1 for positive and negative infinity; biased exponent = all 1 bits; and the fraction = all 0 bits;
  - NaN (Not-A-Number): sign bit = 0 or 1; biased exponent = all 1 bits; and the fraction is anything but all 0 bits.

```
0 00000000 00000000000000000000000 = +0
1 00000000 00000000000000000000000 = - 0
0 11111111 00000000000000000000000 = +Infinity
1 11111111 00000000000000000000000 = - Infinity
0 11111111 00000000000001000001000 = NaN
```

- normalized number
- denormalized number

- Normal number 规格化
  - Exponent field is not all 0s or 1s.



使用软件(如中断)实现

- Subnormal number or denormalized number
  - subnormal is represented by having a zero exponent field with a non-zero significand field. 无前置1
  - subnormal = $(-1)^{sign} \times fraction \times 2^{1-bias}$, e.g., 0.11×2^{-126}
  - subnormal numbers may dramatically increase latency.

- 进位方式
  - 一般四舍六入五凑偶

- The IEEE 754 includes five rounding modes:
  - default: roundTiesToEven
  - optional: roundTiesToAway、roundTowardZero、roundTowardPositive、roundTowardNegative

Mode / Example Value	+11.5	+12.5	−11.5	−12.5
to nearest, ties to even	+12.0	+12.0	−12.0	−12.0
to nearest, ties away from zero	+12.0	+13.0	−12.0	−13.0
toward 0	+11.0	+12.0	−11.0	−12.0
toward +∞	+12.0	+13.0	−11.0	−12.0
toward −∞	+11.0	+12.0	−12.0	−13.0

- For example (using decimal radix with 7 digit precision):

$$123456.7 + 101.7654$$

$$= (1.234567 \times 10^5) + (1.017654 \times 10^2)$$

$$= (1.234567 \times 10^5) + (0.001017654 \times 10^5) \quad \textbf{shifting}$$

$$= (1.234567 + 0.001017654) \times 10^5 \quad \textbf{addition}$$

$$= 1.235584654 \times 10^5 \quad \textbf{(true sum)}$$

$$= 1.235585 \times 10^5 \quad \textbf{rounding}$$

Ch 2 微处理器及其结构

register

- 通用寄存器
  - 64 bit 模式下多了 R8 - R15（还有很多小的分块如R8D，但是这几个拓展寄存器不能单独访问16bit 下的高八位 AH，BH，CH，DH这几个也只能在没有REX前缀的情况下使用）

- # General-Purpose Registers

8

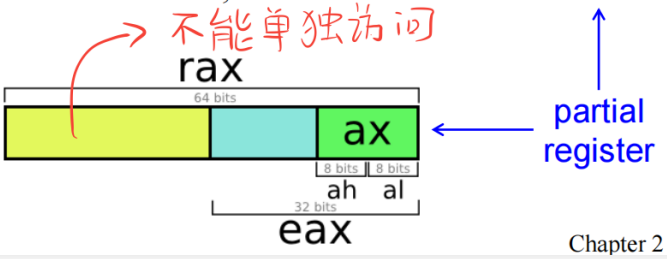
Register	Name	Commonly used as
A	**Accumulator**	Return value, especially the sum of arithmetic operations.
B	**Base index**	Starting point of an array or list structure.
C	**Counter**	Used by loops ie. the i in for(int i=0; i<9; i++)
D	**Data**	Extended space for accumulator. (ie. Multiplication IMUL 32-bit mode will combine EDX+EAX to work on 64-bit values)
BP	**Base Pointer**	Pointer to address of current stack frame. (where function parameters end, and local variables begin)
SP	**Stack Pointer**	Pointer to address of last bytes PUSHed to memory.
SI	**Source Index**	Starting point of unbounded stream data, especially a string.
DI	**Destination Index**	Ending point of unbounded data, especially in slicing operations.

- A 类
  - 做add指令会更短
    - 提高编码密度，单条指令的效果并不改变
- 部分寄存器
  - 不能重命名然后乱序执行

- Note that modifying a 32-bit partial register will set the rest of the register (bit 32-63) to zero, but modifying an 8-bit or 16-bit partial register does not affect the rest of the register. For example:

```
MOV RAX, 1111111111111111H    ; RAX = 1111111111111111H
MOV EAX, 22222222H            ; RAX = 0000000022222222H
MOV AX, 3333H                 ; RAX = 0000000022223333H
MOV AL, 44H                   ; RAX = 0000000022223344H
```

修改低32位会把高位置0



Chapter 2 1

- 状态寄存器 RFLAGS | EFLAGS |
  - OF ：针对有符号数，无符号数这个值没有什么意义

- **C (bit 0):** Carry flag holds the carry after addition or borrow after subtraction.
- **Z (bit 6):** Zero flag shows that the result of an arithmetic or logic operation is zero.
- **S (bit 7):** Sign flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes.
- **O (bit 11):** Overflow flag occurs when signed numbers are added or subtracted.
  - an overflow indicates the result has exceeded the capacity of the machine

- **P (bit 2):** Parity flag is set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise (parity even PF=1; parity odd PF=0).
- **A (bit 4):** Auxiliary carry holds the carry (half-carry after addition or the borrow after BCD operations between bit positions 3 and 4 of the result.

BCD arithmetic 支持 BCD 计算

example 1

8	1000	Eight
+5	+0101	Plus 5
13	1101	is 13 (> 9)
	+0110	so add 6
carry = 1	0011	

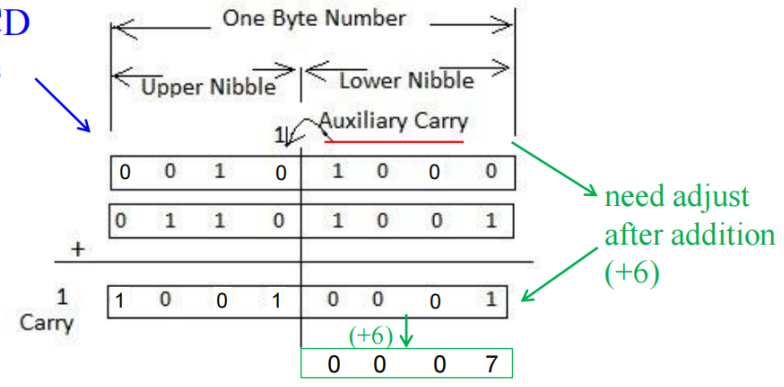
example 2

8	1000	Eight
+9	+1001	Plus 9
17	10001	is 17 (> 9)
	+0110	so add 6
carry = 1	0111	

- 加的结果和 9 进行比较，大于则加6

packed BCD arithmetic

```
  2 8
+ 6 9
-------
  9 1
+   6
-------
  9 7
```



    - D Flag
        - 决定 SI DI 的方向
- 段 Segment寄存器 64bit的分页管理只有部分寄存器 CS GS FS
    - CS
    - DS
    - ES
    - SS
        - BP addresses data within the stack segment
        - SP 共同决定stack起点
- System 寄存器

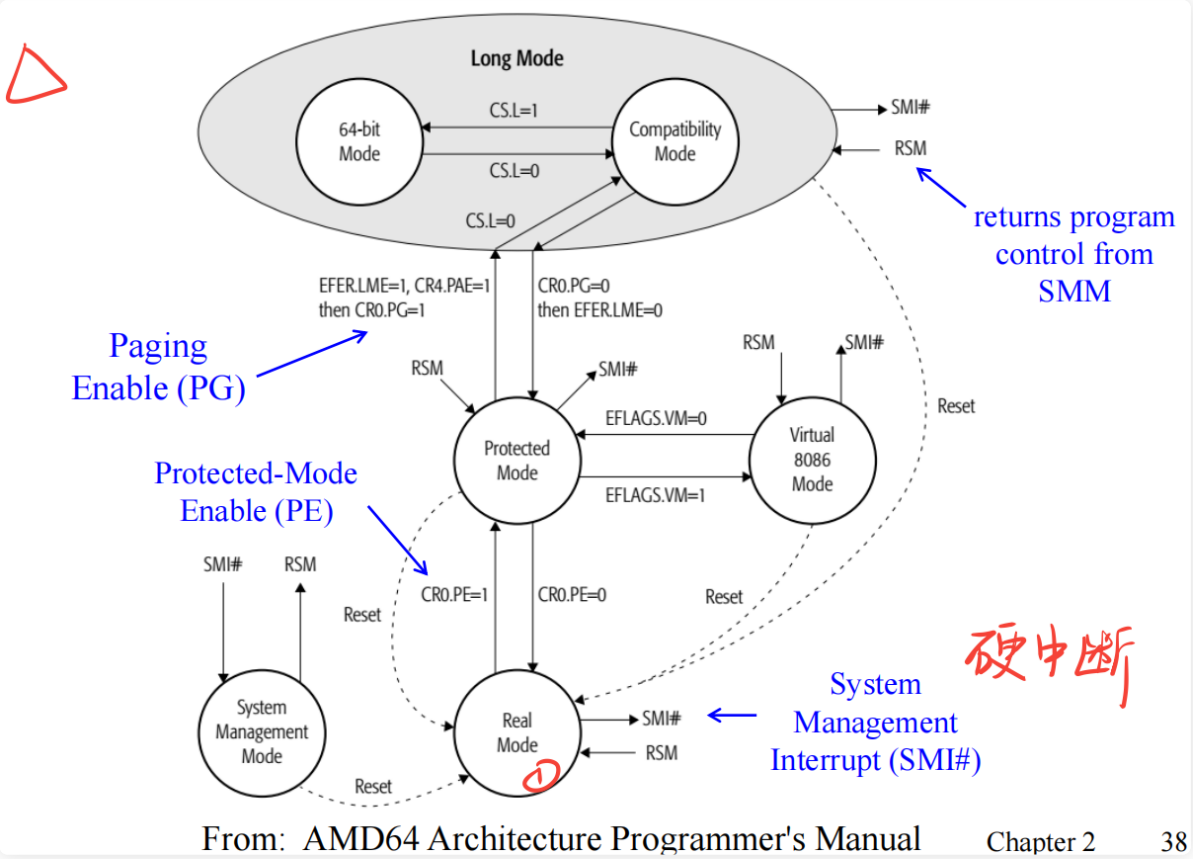
Modes of Operation

- 8086、8088到80186都是实模式、80286、80386、80486都是保护模式，兼容8086所 以有了 Virtual-8086 mode

Operating Mode		Operating System Required	Application Recompile Required	Defaults		Register Extensions	Typical
				Address Size (bits)	Operand Size (bits)		GPR Width (bits)
Long Mode	64-Bit Mode	64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16	16		16
Legacy Mode	Protected Mode	Legacy 32-bit OS	no	32	32	no	32
				16	16		
	Virtual-8086 Mode			16	16		16
	Real Mode	Legacy 16-bit OS					

Operating Modes

the process from real to 64

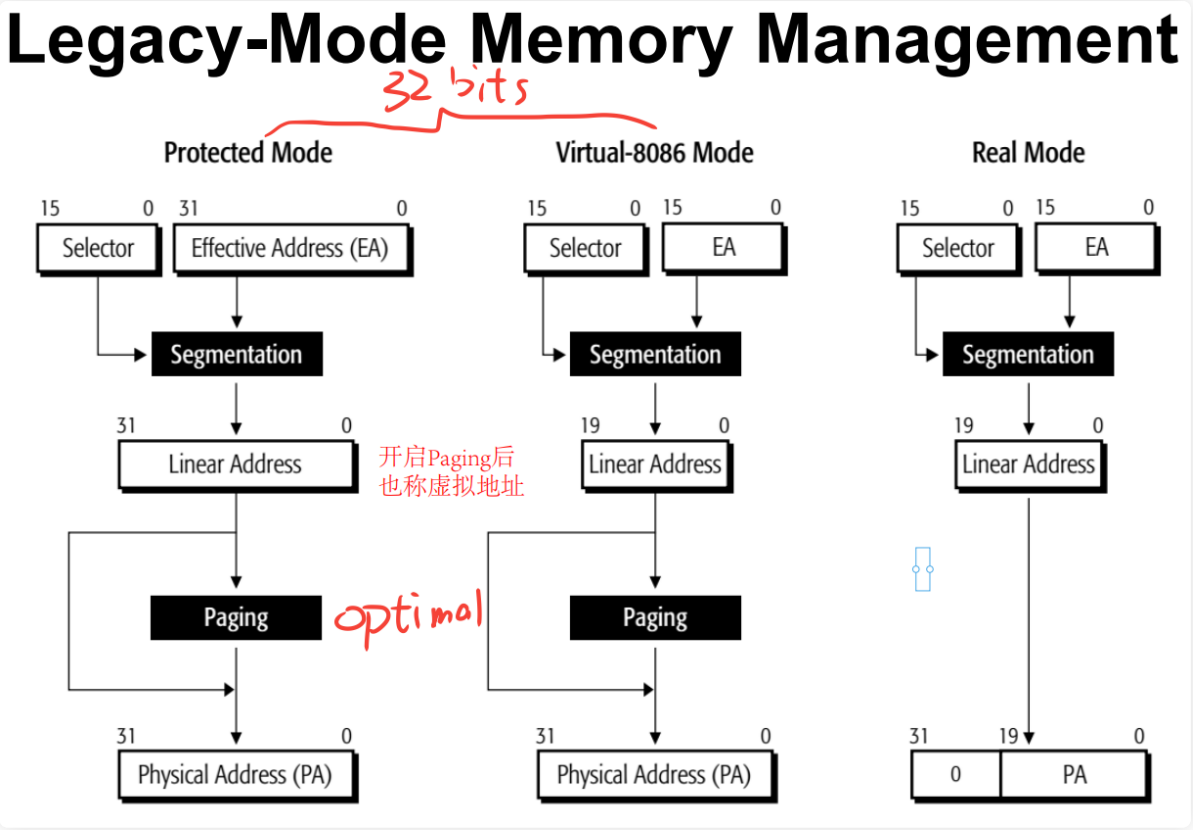


- 进入 real mode
- CR0.PE -> protected mode
- EFER.LME = 1, CR4.PAE = 1（物理地址拓展），CR0.PG = 1 -> 兼容模式
- CS.L = 1 - > 64-bit mode
- 任何模式下，只要发生了 SMI 系统管理中断，就会进入系统管理模式。 RSM 返回

Memory Management

- 满足
  - Relocation 重定位
  - Protection
  - Sharing
- Scheme
  - Segmentation
    - SMT

- Page



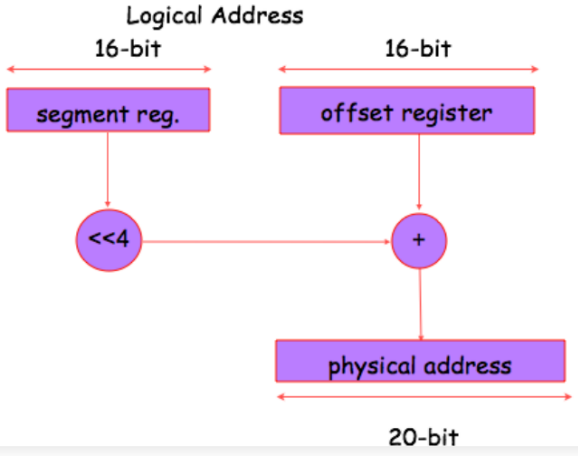
Memory Addressing

Real mode

- 只分段

Linear Address = Segment Address << 4+
Effective Address (offset)

Figure The real mode memory-addressing scheme, using a segment address plus an offset.



- 各个段其实可以重叠

- wrap-around segment 左移四位之后再加上offset可能超出限制，这时做地址回滚（把超出部分截断了）

- 1 MB address wrap-around was an inherent feature of the Intel 8086/8088/80186 CPU, but not the 80286 (24-bit address lines) and later models.
- Consider the following two examples:

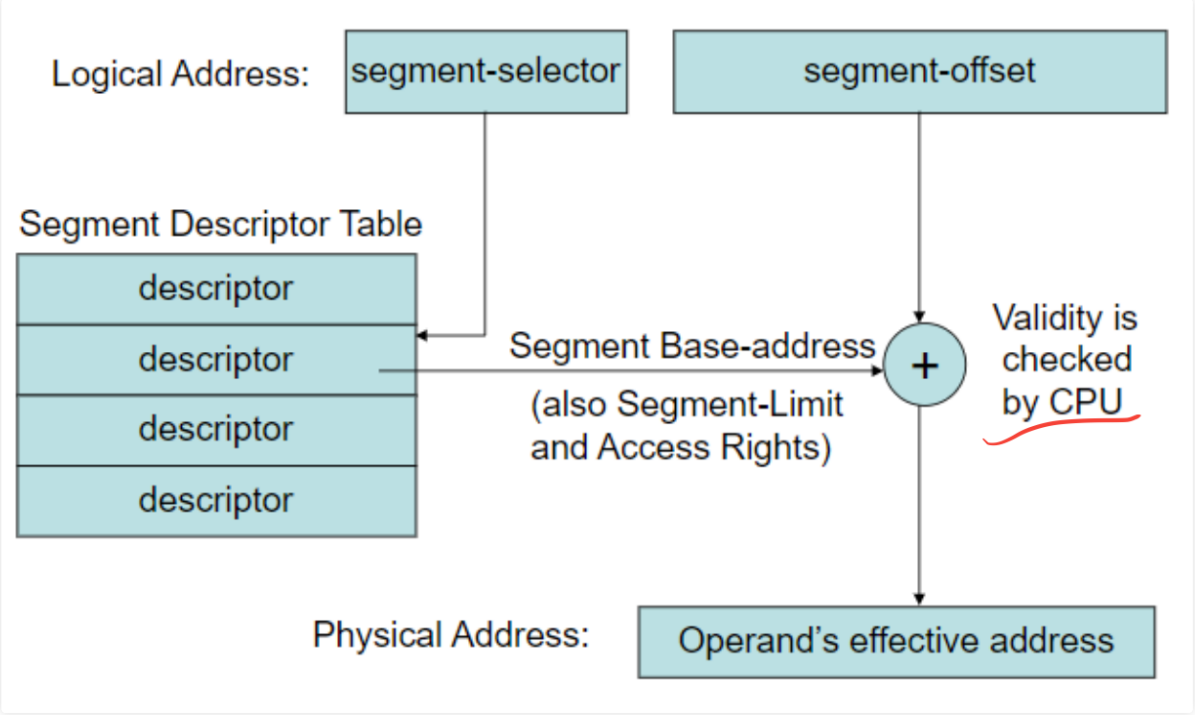
logical address:	0xFFFF:0xFFFF	0xF800:8000
linear address:	0x10FFEF	0x100000
physical address:	0x0FFEF	0x00000

逻辑地址

地址回滚

"wrap around"

Protected Mode



- 要满足隔离，刚刚的模式不行

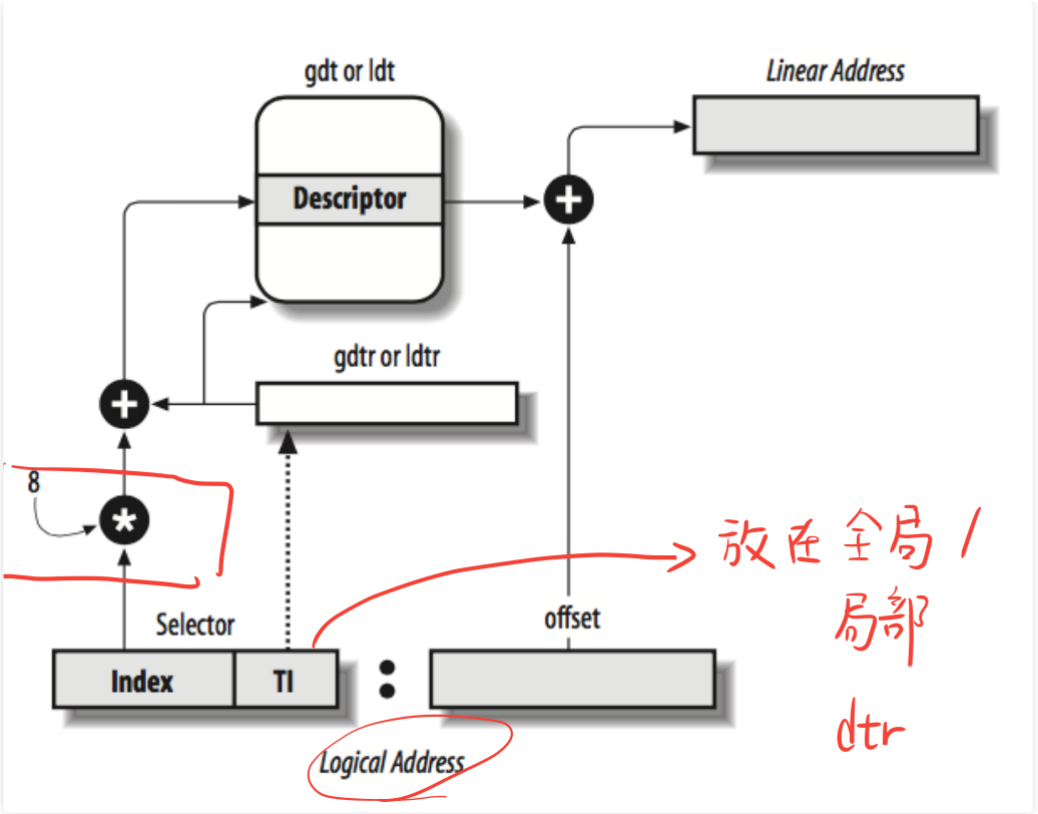
- Selector 用来定位描述符的位置

  - In place of a segment address, the segment register contains a **selector** that selects a descriptor from a descriptor table. 指向描述符
  - The **descriptor** describes the memory segment's location, length, and access rights.

- Descriptor
  - Table
    - Global | GDT 所有软件可见
      - 至少有两个：代码段的，数据段的
      - First null ： 第一个必须是 `null` ； 如果
    - Local | LDT 不一定存在局部描述符表
    - Interrupt | IDT 安全管理
  - GDTR LDTR的寻址方式

```
`TI = 0 GDTR`
```

```
<img src="https://zzh-pic-for-self.oss-cn-hangzhou.aliyuncs.com/img/202412271933429.png" alt="image-20241227193351312" style="zoom:50%;" />
```



- **必考：** 粒度位G相关的计算 P73
  - G = 1，一位代表 4k 的大小
  - - Problem 1: For a descriptor with a base address of 10000000H, a limit of 001FFH, and G=0, what is the starting and ending locations?
      - starting location: 10000000H
      - ending location : 10000000H+001FFH=100001FFH
    - Problem 2: For a descriptor with a base address of 10000000H, a limit of 001FFH, and G=1, what is the starting and ending locations?

G=1, what is the starting and ending locations?

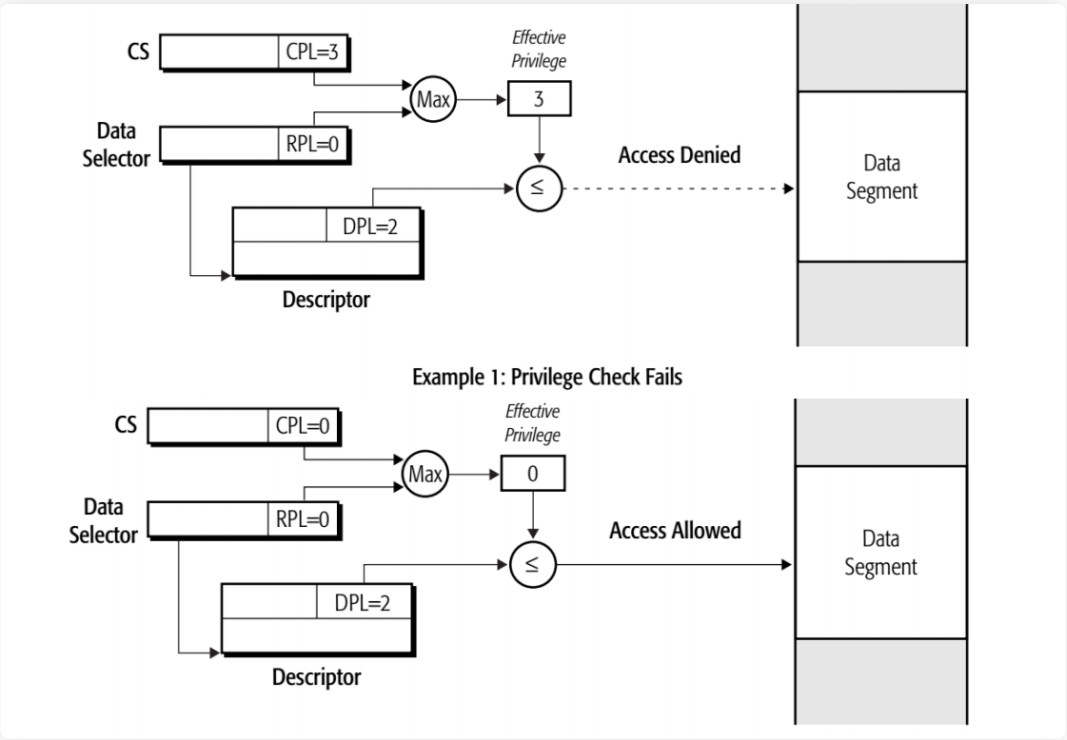
Segment Limit Example (2/2)

- If G = 1, ending = starting + (segment size-1), where the segment size = (limit+1) x 4K bytes
  - (Limit+1) x 4K – 1 = (Limit)000H + (4K – 1) = (Limit)FFF H
  - Limit is appended with FFFH to determine the ending address, namely ending = starting + (Limit)FFF H
- For Problem 2 (limit = 001FFH, G = 1):
  - starting location: 10000000H
  - ending location: 10000000H + 001FFFFFH= 101FFFFFH

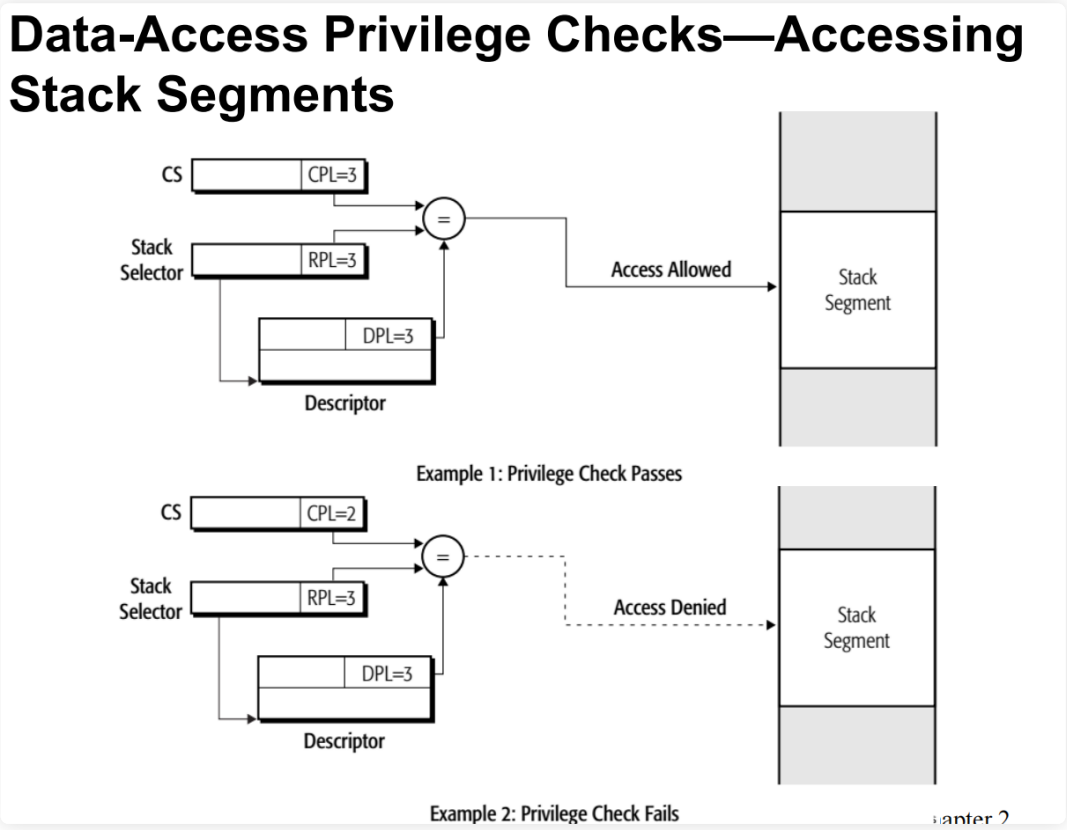
移位即可

- Privilege Level
  - details 不过它里面对于栈的访问的条件判断个人感觉不对
  - **权限级别值越小00，优先级越高**
  - DPL 描述符的权限级别，规定了哪个特权级别的代码可以访问该段
  - RPL 请求者的权限级别，是Selector（段选择子）中的；决定对段的访问权限
  - CPL 目前CPU的权限级别，**处理器当前运行代码的特权级别**
    - 防止发生越权访问
  - 访问情况

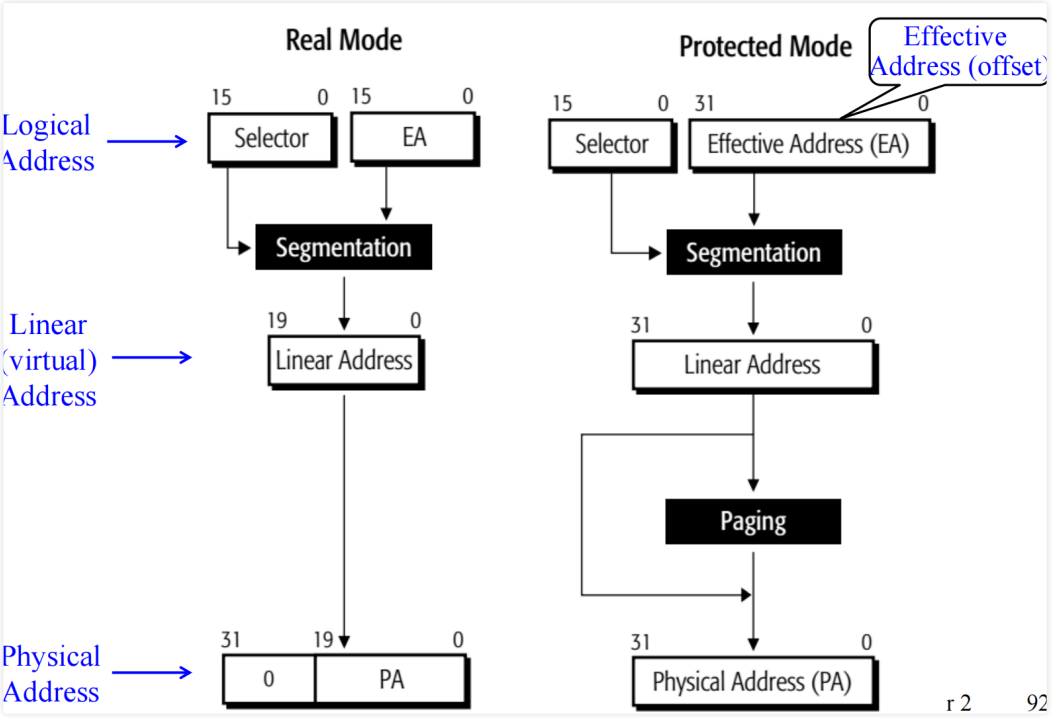
- 访问数据 CS和Selector中优先级较低的也要高于等于Descriptor，从值上来说：$DPL \geq \max(RPL, CPL)$



- 访问Stack 必须严格等于



四种内存地址类型及其转换关系



- **Effective Address** 偏移地址
  - 这里不太需要关注，之后会详细介绍这种寻址
    - Effective Addresses (near pointers): The offset into a memory segment is referred to as an effective address. The effective-address is represented as:

      Effective Address = Base + (Scale x Index) + Displacement

      必须 si/di 来自

      - Base: A value stored in register. BS BP
      - Scale: A value of 1, 2, 4, or 8.
      - Index: A value stored in register.
      - Displacement: A value encoded as part of the instruction.

      立即数

- **Logical Address**
  - 64bit flat mem 下，LA == EA
- **Linear Addresses (virtual addresses)**

- Linear Addresses (virtual addresses): A linear address is formed by adding the segment-base address to the effective address (segment offset). The linear address is represented as:

  Linear Address = Segment Base Address + Effective Address

- When the flat-memory model is used—as in 64-bit mode—a segment-base address is treated as 0. In this case, the linear address is identical to the effective address.

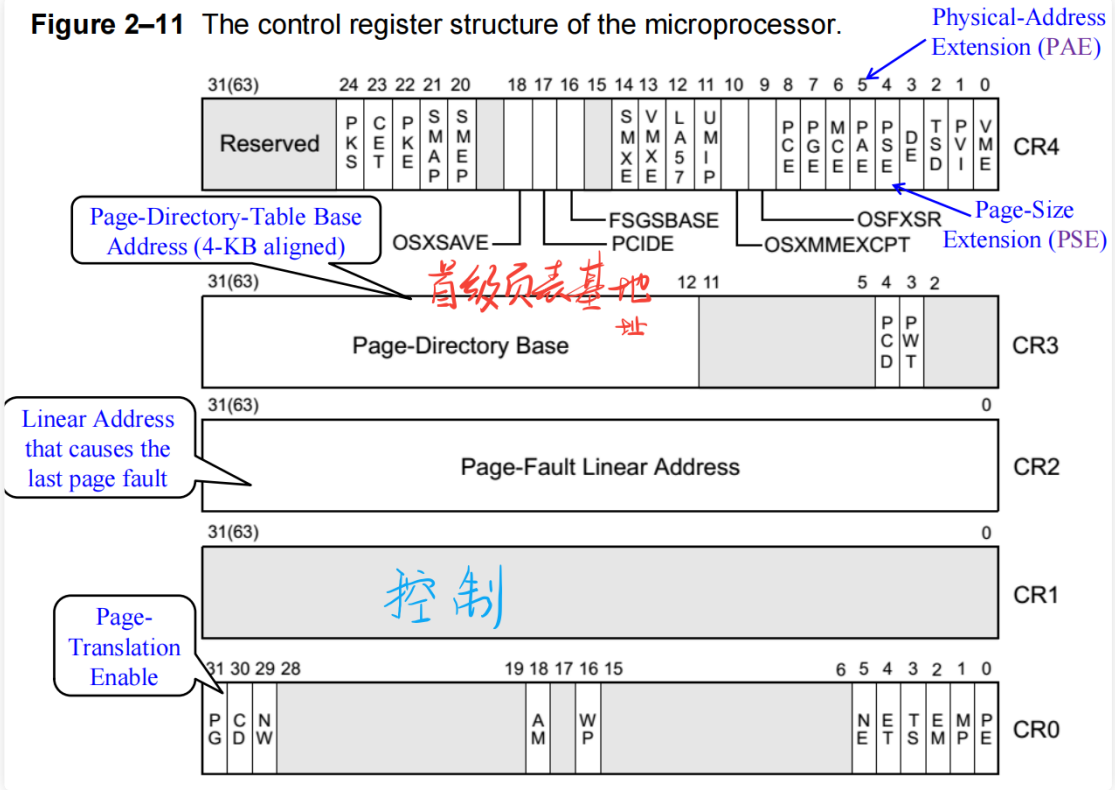
- **Physical Addresses**

页

> “
>
> OS学了

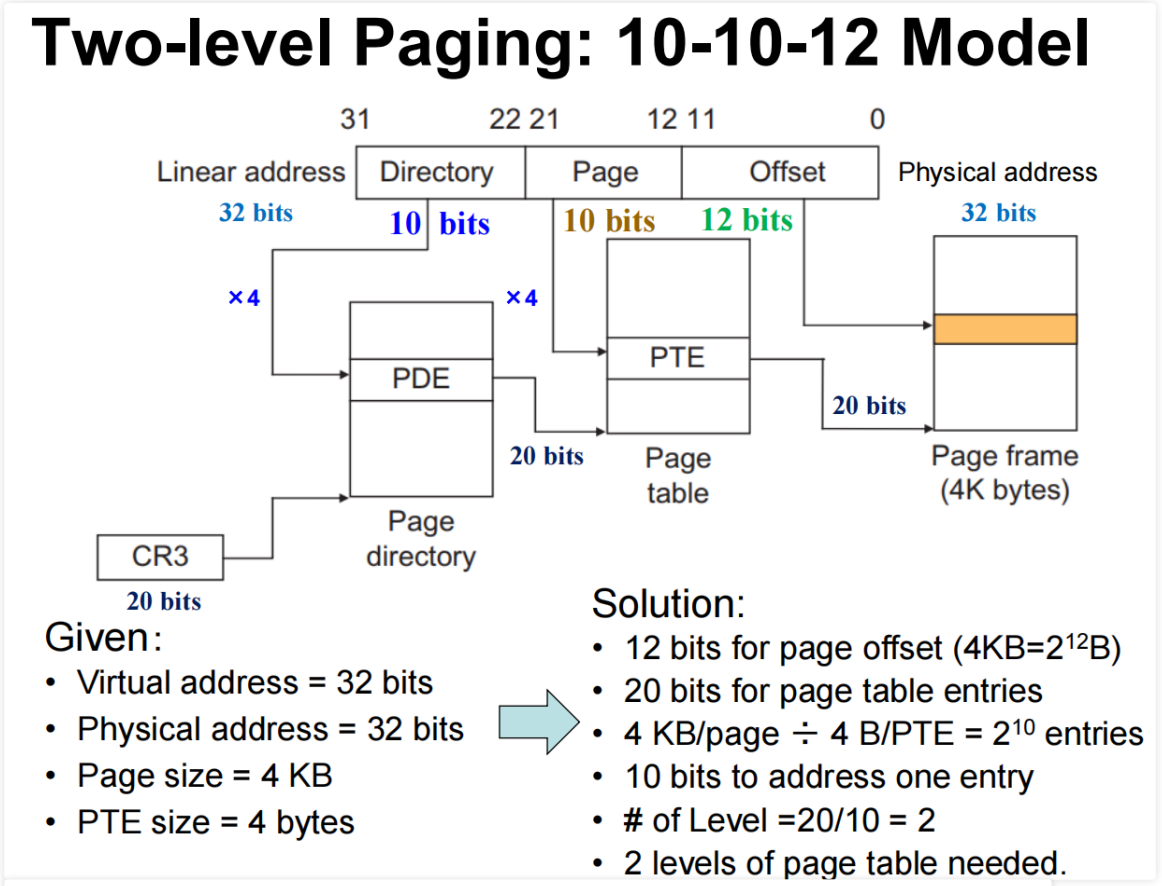
- 控制寄存器

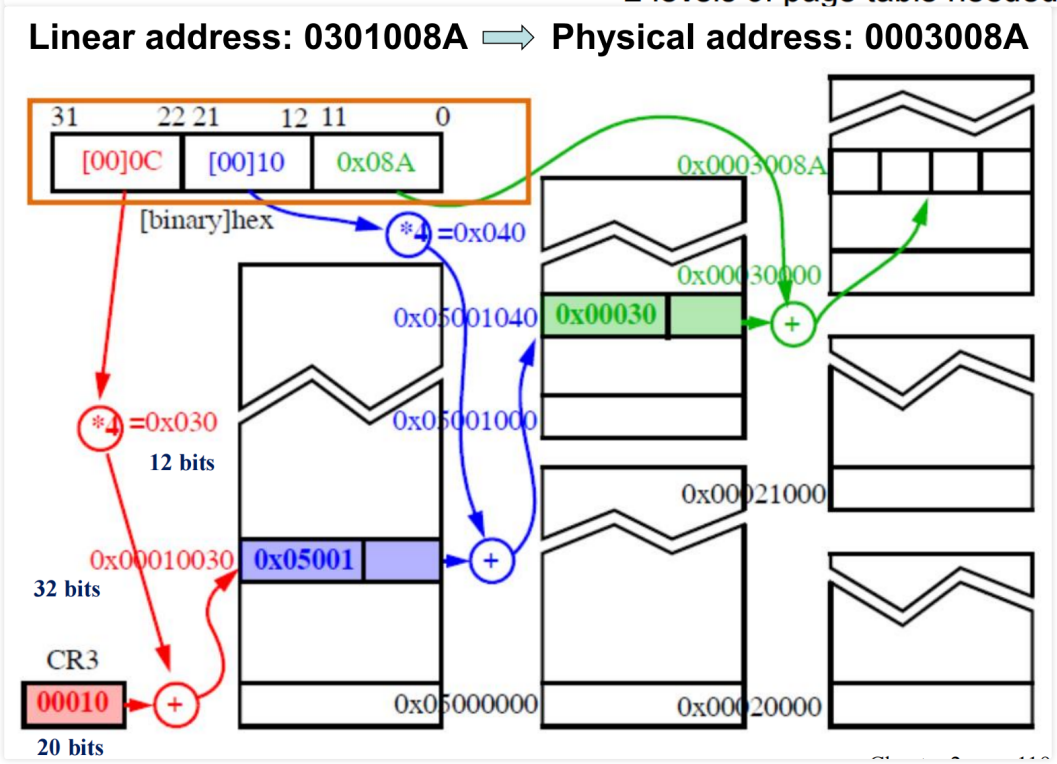
Figure 2–11 The control register structure of the microprocessor.



- CR3 20bit 指向首级的PDT

- 101012模式





- 这里的 *4 是因为每个目录项都是四字节，所以需要计算偏移字节量
- 注意 CR3 中包含的是高20bit（按4KB大小的页对齐）

Ch 3 Addressing Mode

- All labels must begin with a letter or one of the following special characters: @, $, -, or ?.
  - e.g., begin, data$, here@, etc.

Data-Addressing Modes

Addressing Form	Example
BaseReg	mov rax,[rbx]
BaseReg + Disp	mov rax,[rbx+16]
IndexReg * SF + Disp	mov rax,[r15*8+48]
BaseReg + IndexReg	mov rax,[rbx+r15]
BaseReg + IndexReg + Disp	mov rax,[rbx+r15+32]
BaseReg + IndexReg * SF	mov rax,[rbx+r15*8]
BaseReg + IndexReg * SF + Disp	mov rax,[rbx+r15*8+64]
RIP + Disp	mov rax,[Val]

Table Memory Operand Addressing Forms

- immediate operand 立即数寻址
  - 16位数的前面如果是字母开头，一定要加上一个0
    - In MASM, hexadecimal numbers must always start with a decimal digit (0–9). Otherwise they would be mistaken for label names.
    - If necessary, add a leading zero to distinguish between symbols and hexadecimal numbers that start with a letter. E.g.,
      - MOV AX, F2H ;load a label named F2H
      - MOV AX, 0F2H ;load a hexadecimal F2H
- 寄存器寻址
  - MOV 操作数的宽度要匹配

- 内存寻址
  - direct addressing
  - displacement addressing
    - Effective Address = Base + (Scale × Index) + Disp

      memory location (GPR)

      - For indirect addressing
        - In the 8086 through the 80286, indirect addressing can only use the BX, BP, SI and DI registers, e.g., MOV AX,[BX].
        - 80386 and above allow any extended register, e.g., MOV AX,[EDX]
        - In the 64-bit mode, segment registers are not used in address calculation.
      - 段的基地址：除了BP、ESP使用 SS ，其他默认使用 DS
  - 不能直接 MOV [内存] [内存]
    - 不允许两个同时都是内存，除了字符串
    - 一条指令中至少有一个要确定操作数长度 MOV [DI], 10H is ambiguous
      - Directive 伪指令 MOV BYTE PTR [DI],10H 指示内存操作数长度
- **Base-Plus-Index Addressing**
  - - For register relative addressing
      - In the 8086 through the 80286, the memory data are addressed by adding the displacement to the contents of a base register (BP or BX) or an index register (DI or SI), e.g., MOV AX,[DI+100H].
      - In the 80386 and above, the displacement can be a 32-bit number and the register can be any 32-bit register (except that ESP cannot be used as an index register), e.g., MOV DL,[EAX+10H].
    - Scale只能是 1 2 4 8
- **RIP-relative addressing** 64位模式引入

- RIP-relative addressing uses a signed 32-bit displacement to calculate the effective address of the next instruction by sign-extend the 32-bit value and add to the 64-bit value in RIP, e.g.,

```
int var;

void f(int x) {
        var = x;
}
```

```
f:
        mov     var[rip], edi    ; edi = x
        ret
```

- Using RIP-relative addressing makes position-independent code smaller and simpler, where all code and data need to be addressable within a 32-bit offset.

- 64 位地址的高16位必须要是符号拓展，要么 `FFFF` OR `0000`

- For a 48-bit linear address, a canonical address must have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one),
  - canonical address: FFFF8010BC001000, 00007C80B8102040
  - non-canonical address: 1122334455667788, 3375DA44B5667788

- Stack Memory-Addressing Modes
  - This mode involves stack registry operations, e.g., PUSH AX
- Program Memory-Addressing Modes
- 一定要注意ESP不能作为Index

Summary——Effective Address Computation

Offset = Base + (Index * Scale) + Displacement

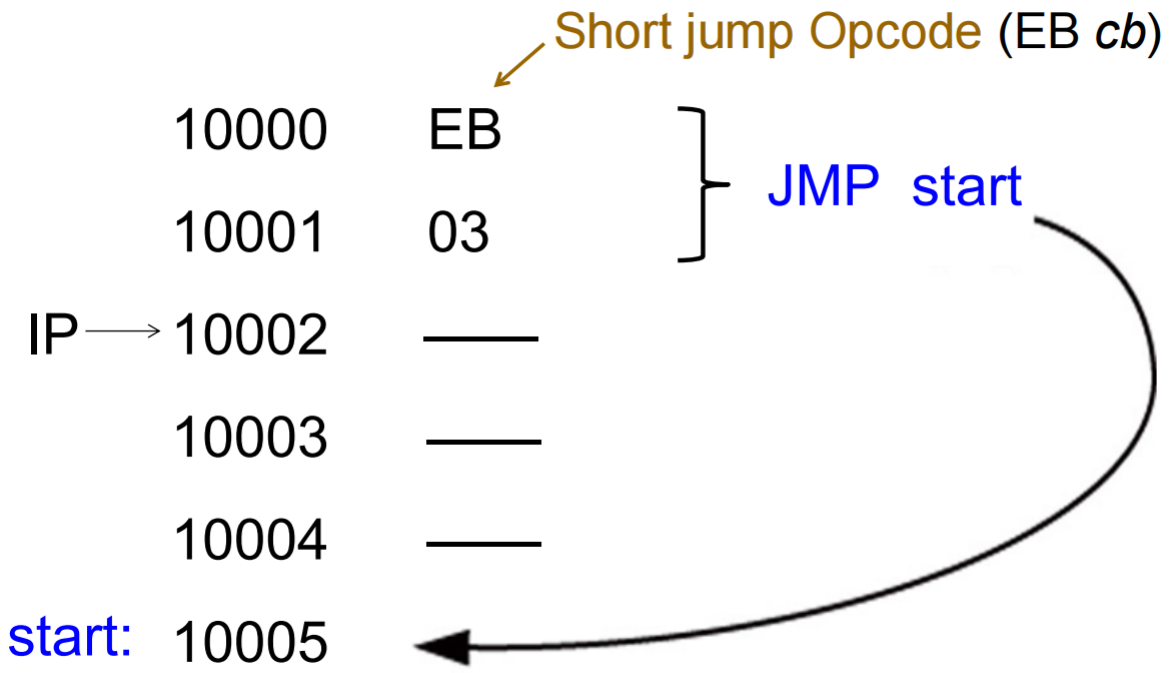
- 8086-80286:
  - Base: BX/BP
  - Index: SI/DI
  - Disp: 8-bit/16-bit
- 80386 and above:
  - Base: any 32-bit register
  - Index: any 32-bit register except ESP
  - Disp: 8-bit/16-bit/32-bit
- A scale factor can be 1, 2, 4, and 8, which may be used only when an index also is used.
- When the BP/EBP or ESP is used, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

代码相关的寻址

- 在不确定长度的情况下，需要加上 `JUMP NEAR PTR` 等标签表示 `displacement` 的长度

Assembly Language	*Operation*
JMP AX	Jumps to the current code segment location addressed by the contents of AX
JMP CX	Jumps to the current code segment location addressed by the contents of CX
JMP NEAR PTR[BX]	Jumps to the current code segment location addressed by the contents of the data segment location addressed by BX
JMP NEAR PTR[DI+2]	Jumps to the current code segment location addressed by the contents of the data segment memory location addressed by DI plus 2
JMP TABLE[BX]	Jumps to the current code segment location addressed by the contents of the data segment memory location address by TABLE plus BX
JMP ECX	Jumps to the current code segment location addressed by the contents of ECX
JMP RDI	Jumps to the linear address contained in the RDI register (64-bit mode)

- 看清楚加 3 是从哪里开始



- Figure 3–16 shows a jump table that is stored, beginning at memory location TABLE. The exact address chosen from the TABLE is determined by an index stored with the jump instruction.

This is a jump table that stores addresses of various programs.

```
TABLE  DW  LOC0
       DW  LOC1
       DW  LOC2
       DW  LOC3
```

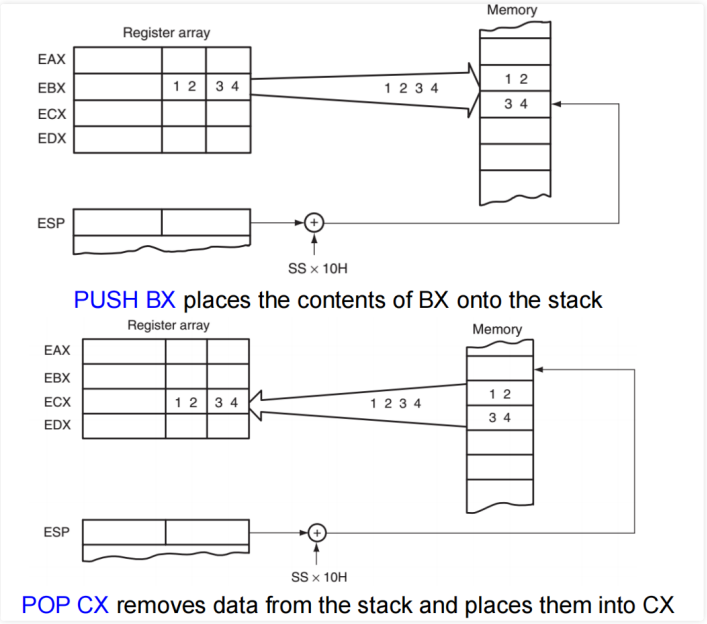
```
;Using indirect addressing for a jump
;
MOV BX,4            ;address LOC2
JMP TABLE[BX]       ;jump to LOC2
```

The jump table is referenced by the program

Stack 相关

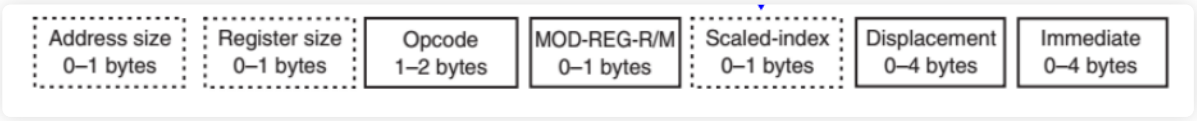
- Push、Pop 至少两个字节
  - 先入高的部分再低的部分（仍然是小端模式）
- 栈的位置

    - SS 基地址，16位 => 拼凑20位的地址要左移四位
    - SP 偏移量
- PUSH
- PUSHA
- PUSHF

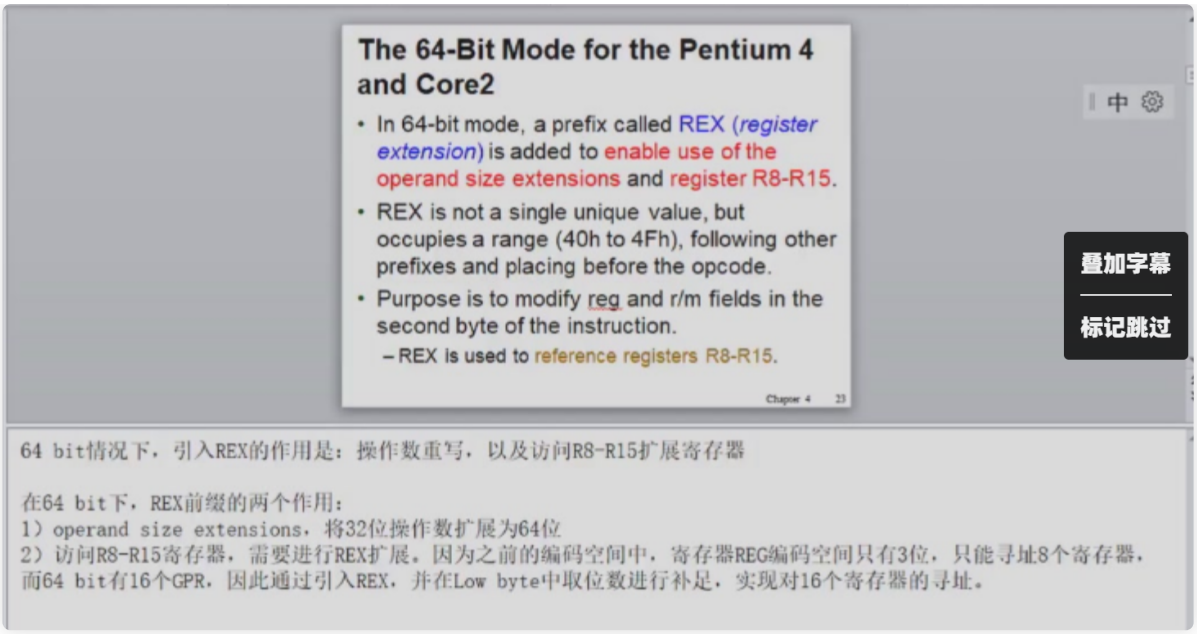


数据在内存中的存放规律：小端字节序

Ch 4 Data Movement Instruction



- 不同的寻址方式是如何实现的

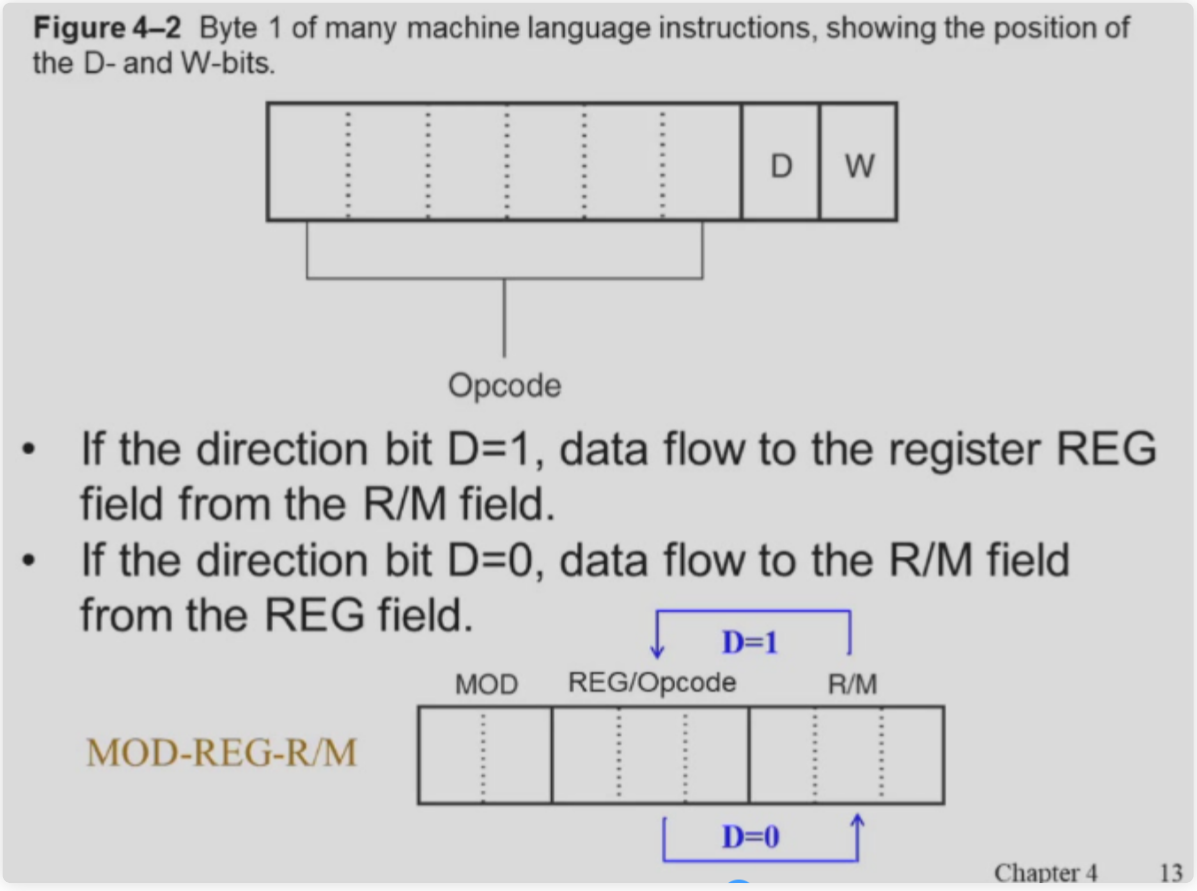


64 bit情况下，引入REX的作用是：操作数重写，以及访问R8-R15扩展寄存器

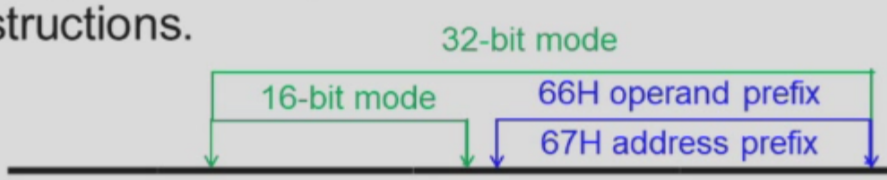
在64 bit下，REX前缀的两个作用：
1）operand size extensions，将32位操作数扩展为64位
2）访问R8-R15寄存器，需要进行REX扩展。因为之前的编码空间中，寄存器REG编码空间只有3位，只能寻址8个寄存器，而64 bit有16个GPR，因此通过引入REX，并在Low byte中取位数进行补足，实现对16个寄存器的寻址。

Operation Mode

- Prefix
- Opcode
- MOD
-



- If the W-bit=1, the data size is a *word* or *doubleword.*
- If the W-bit=0, the data size is always a *byte.*
- The W-bit appears in most instructions, while the D-bit appears mainly with the MOV and some instructions.

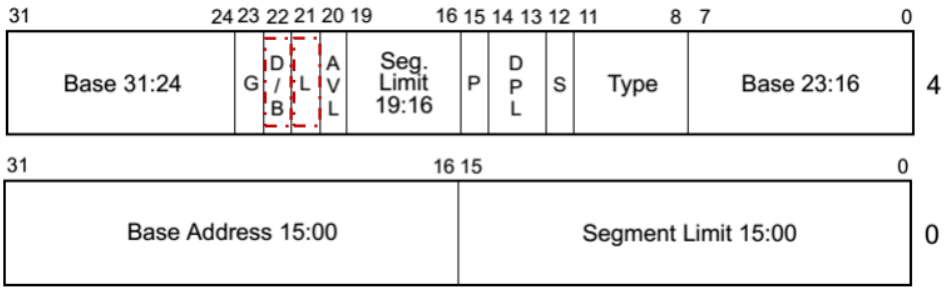


Code	W = 0 *(Byte)*	W = 1 *(Word)*	W = 1 *(Doubleword)*
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

- Address Mode 地址

  - There are three operation modes with following default address and operand size:
    - 16-bit modes (real, vm86, protected): default address and operand-size are 16-bit
    - 32-bit protected mode (protected): default address and operand-size are 32-bit
    - 64-bit mode: default address size is 64-bit, default operand-size is 32-bit

操作数

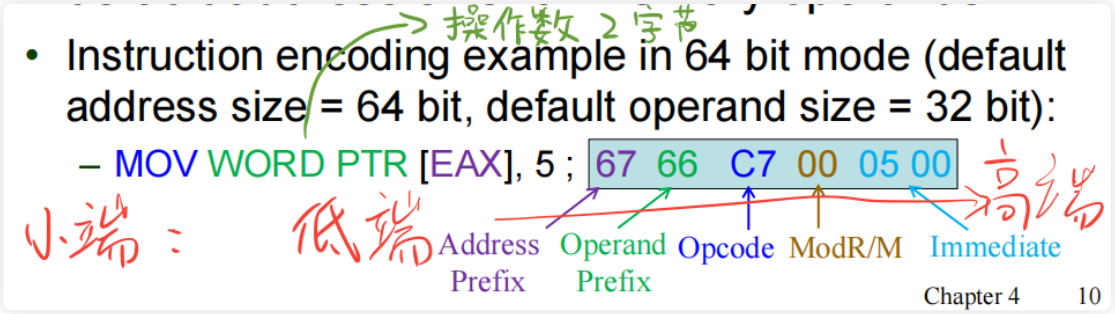


code segment descriptor

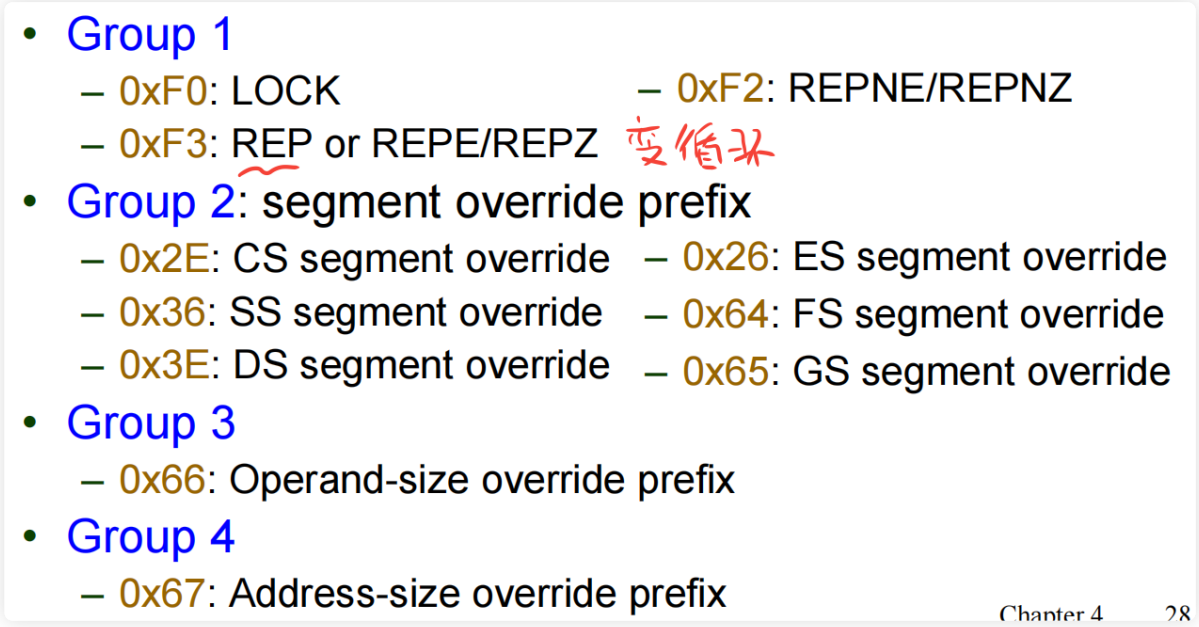
  - In the code segment descriptor, D/B-bit and L-bit indicate the operation mode:
    - L=0 and D/B =0 for 16-bit instruction mode
    - L=0 and D/B =1 for 32-bit instruction mode
    - L=1 for 64-bit instruction mode

- 操作数
- 前缀
  - 66H 改变默认的数据长度

- 67H 改变默认的地址长度



前缀



- 每一组不能同时出现
  - LOCK 需要写到内存 需要影响到内存
    - An undefined opcode exception (#UD) occurs if the LOCK prefix is used with any other instruction. 如果不影响内存的指令使用了这个前缀，那么将出现 exception
  - 更改类寄存器

- Instructions: CS
- Local Data: DS
- Stack: SS
- Destination Strings: ES

- Programmers can override the default segment with a segment-override prefix, which is a byte placed at the beginning of an instruction.
- For example, in 32-bit mode:

```
MOV EAX, [EBX]      ; 8B 03,  default segment = DS.
MOV EAX, CS: [EBX]  ; 2E 8B 03, the 2E is CS segment
                    ; override prefix.
```

◦ 操作数前缀

- In 64-bit mode, instructions default to a 32-bit operand size.
- The prefix allows mixing of 16, 32, and 64-bit data:
  - a REX (REX.W) prefix can specify a 64-bit operand size
  - a 66H prefix specifies a 16-bit operand size
  - the REX prefix takes precedence over the 66h prefix.

Operating Mode		Default Operand Size (Bits)	Effective Operand Size (Bits)	Instruction Prefix[1]	
				66h	REX.W[3]
Long Mode	64-Bit Mode	32[2]	64	don't care	yes
			32	no	no
			16	yes	no
	Compatibility Mode	32	32	no	Not Appli-cable
			16	yes	
		16	32	yes	
			16	no	
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no	
			16	yes	
		16	32	yes	
			16	no	

默认

Operand-Size Overrides

- The default operand size is defined by the current operation mode, but the operand-size override prefix (REX or 66H) can change the default operand size. E.g., in 64 bit mode (default operand size = 32 bit):
  - MOV WORD PTR [RAX], 5 ; 66 C7 00 05 00
    66H prefix opcode operand
  - MOV DWORD PTR [RAX], 5 ; C7 00 05 00 00 00
    opcode operand
  - MOV QWORD PTR [RAX], 5 ; 48 C7 00 05 00 00 00
    REX prefix opcode operand

0扩展到64位

- 地址前缀
  - The default address size is 64 bits in 64-bit mode. The size can be overridden to 32 bits, but 16-bit addresses are not supported in 64-bit mode.
  - The address-size override prefix (67H) selects the non-default address size.

Operating Mode		Default Address Size (Bits)	Effective Address Size (Bits)	Address-Size Prefix (67h)[1] Required?
Long Mode	64-Bit Mode	64	64	no
			32	yes
	Compatibility Mode	32	32	no
			16	yes
		16	32	yes
			16	no
Legacy Mode (Protected, Virtual-8086, or Real Mode)		32	32	no
			16	yes
		16	32	yes
			16	no

地址寻址

Address-Size Overrides

- The default address size for memory operands is determined by the current operation mode, but it can be overridden by an address-size override prefix (67H), for example
  - In 64 bit mode, addresses are 64 bits by default, but they can be overridden to 32 bits by an address size prefix.
- Some examples:
  - in 64 bit mode (default address size = 64 bit)

有效地址

地址32位

```
MOV  EAX, [RBX]  ; 8B 03          MOV  EAX, [EBX]  ; 67 8B 03
                   opcode                            67H prefix opcode
```

  - in 32 bit mode (default address size = 32 bit)

```
MOV  EAX, [EBX]  ; 8B 03          MOV  EAX, [BX]   ; 67 8B 07
                   opcode                            67H prefix opcode
```

3

REX

- REX前缀是一组值
- 两个地方用到
  - 访问拓展寄存器
  - 拓展操作数为64位

REX 寄存器拓展前缀 为了寻找64位下，拓展的8个通用寄存器

lower nibble is divided into four 1-bit fields (W, R, X, and B).

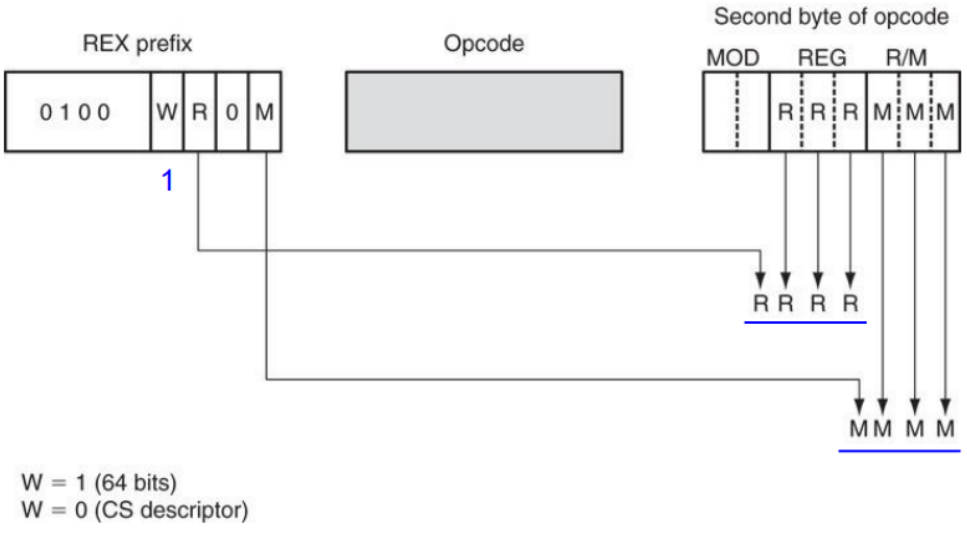
7	6	5	4	3	2	1	0
0	1	0	0	W	R	X	B

Mnemonic	Bit Position(s)	Definition
—	7:4	0100 (4h)
REX.W	3	0 = Default operand size 1 = 64-bit operand size
REX.R	2	1-bit (msb) extension of the ModRM *reg* field[1], permitting access to 16 registers.
REX.X	1	1-bit (msb) extension of the SIB *index* field[1], permitting access to 16 registers.
REX.B	0	1-bit (msb) extension of the ModRM *r/m* field[1], SIB *base* field[1], or opcode *reg* field, permitting access to 16 registers.

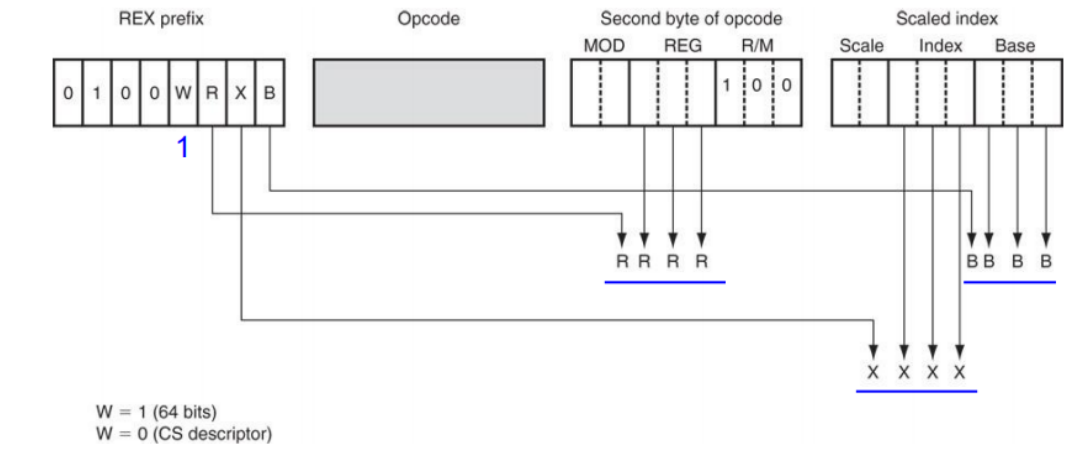
REX Prefix-Byte Fields

- 0100
- W 操作数长度 在64位模式下生效，默认长度 32 位，开启后64位
- R 这里的作为高位，
- X 是否做scale， R/M
- B
-

Figure 4–11 The application of REX without scaled index (R/M≠100).



- **Figure 4–12** The scaled-index byte and REX prefix for 64-bit operations (R/M=100).



- 转义编码 Escape Sequence
  - OF 开头

    – 0F AF is the opcode of IMUL r16, r/m16

    – 0F B6 is the opcode of MOVZX r16, r/m8

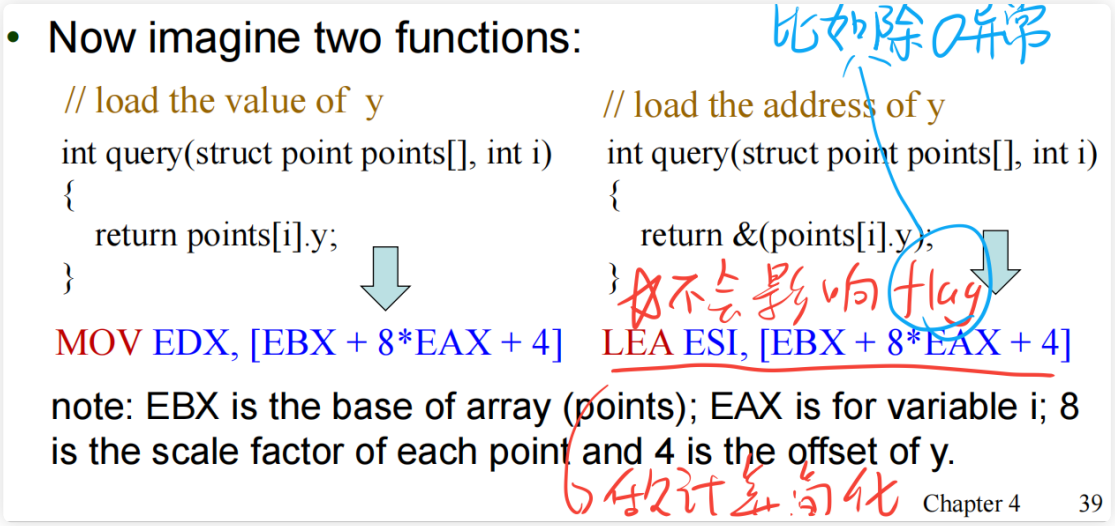
Load Effective Address 偏移地址

移地址

LEA | Effective address

- 用于获取地址 OFFSET（`Near`）
- flag 不变(MOV都是这样)

- 跳转的判断条件不要使用



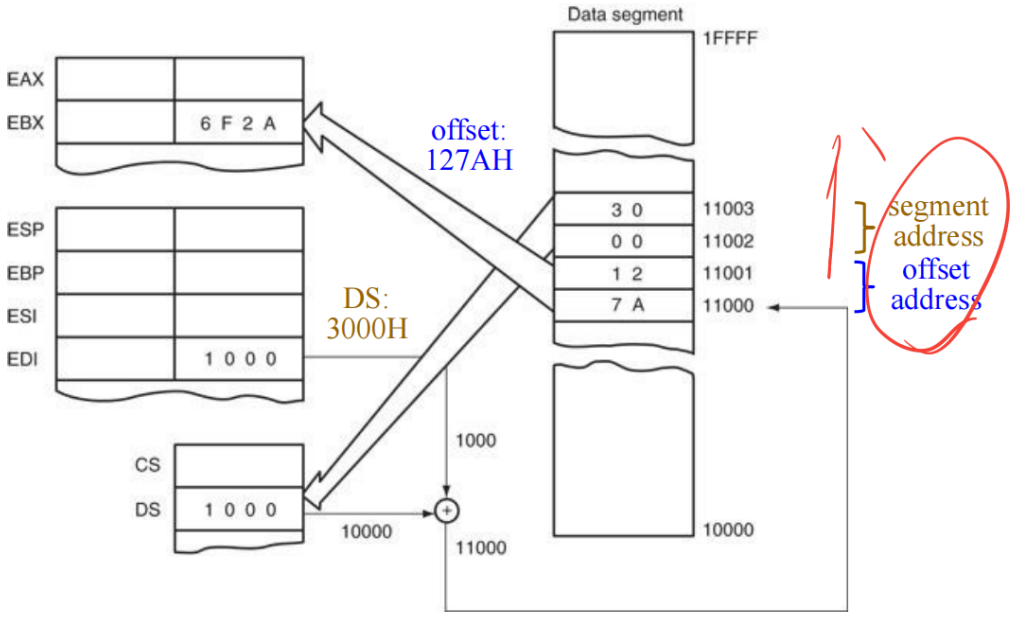
OFFSET Label

- 静态计算
- 后面只能跟着Label
- `LEA SI, DATA1` `MOV SI, OFFSET DATA1` 二者等效
  - `OFFSET` 更加高效

加载远地址

- LSS 堆栈的段地址
- `LDS` 数据段
  - 高地址为段地址
  - 低地址为偏移地址

Figure 4–17 When DS=1000H and DI = 1000H, the LDS BX,[DI] instruction loads register BX from addresses 11000H and 11001H and register DS from locations 11002H and 11003H. This instruction is shown at the point just before DS changes to 3000H and BX changes to 127AH.



String Data Transfers

至少一个操作数是内存

- LODS **内存 -> 寄存器**
- STOS
- MOVS **内存和内存**
- INS
  - 从**端口到内存**
  - DI 放内存地址
  - DX 放设备端口
- OUTS
- SCAS CMPS

相关寄存器

- DI SI / EDI ESI
- Flag - DF

- CLD
- STD

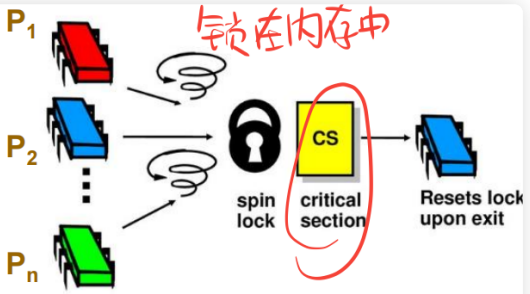
- Repeat Prefix
  - REP
  - REPZ REPE

- CX | ECX 作为 counter

MISCELLANEOUS DATA TRANSFER

- **XCHG**
  - **交换**一个寄存器和一个寄存器/内存的值
    - `XCHG AL,[DI] is identical to the XCHG [DI],AL`
    - 不能是段寄存器
  - 实现锁 有一个是内存，加 `LOCK` 前缀
    - acquire the spinlock

```
; lock variable. 1 = locked, 0 = unlocked.
locked    dd     0

; Set the EAX register to 1.
spin_lock:
      mov     eax, 1

;  Atomically swap the EAX with the lock.
lock xchg   eax, [locked]

; Test EAX with itself.
      test    eax, eax
;If EAX is 0, we just obtain and lock it.
; Otherwise, repeatedly request the lock.
      jnz     spin_lock
      ret
```

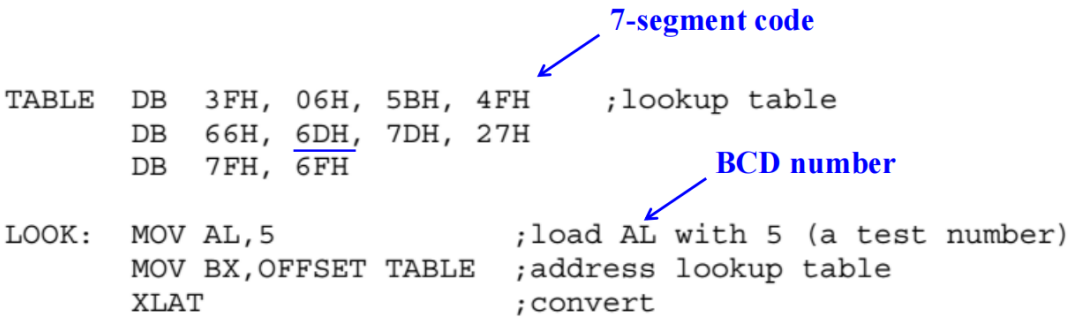


release the spinlock

```
; Set the EAX register to 0.
spin_unlock:
      mov     eax, 0

; Atomically swap the EAX register
; with  the lock variable.
lock xchg    eax, [locked]

      ret
```

- LAHF SAHF
  - XFLAG 的低八位和AH寄存器
  - `LAHF` 加载到AH中
  - `SAHF` AH存储到XFLAG中

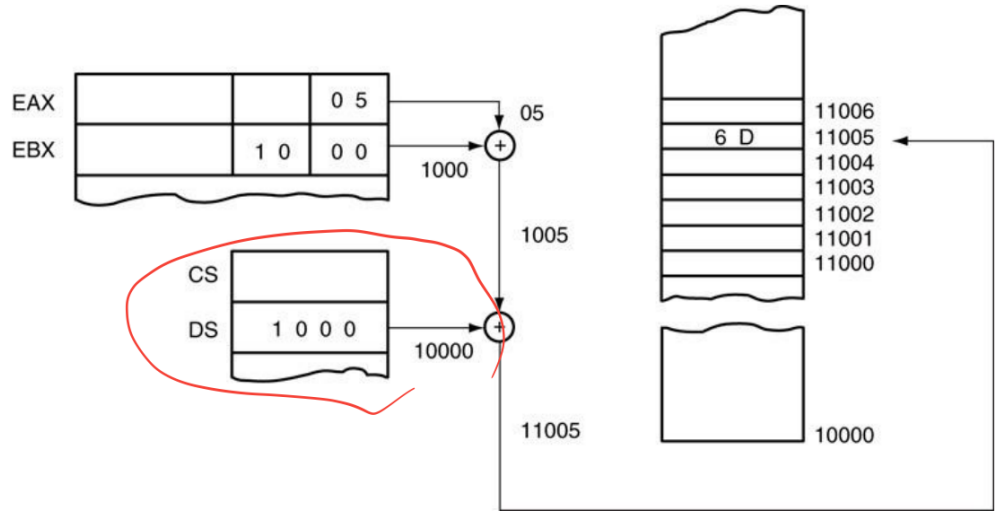
- **XLAT | TABLE Look-up Translation**
  - **查表使用**
  - 相当于 `MOV AL, [seg:BX + AL]`，这是一个非法的指令
  - **参数**
    - BX 基地址（一个表的基地址）
    - AL
      - 作为偏移
      - 作为目的寄存器 最终结果
  - - Suppose that a 7-segment LED display lookup table is stored in memory at address TABLE. The XLAT instruction then uses the lookup table to translate the BCD number in AL to a 7-segment code in AL.




```
TABLE   DB   3FH, 06H, 5BH, 4FH     ;lookup table
        DB   66H, 6DH, 7DH, 27H
        DB   7FH, 6FH

LOOK:   MOV AL,5               ;load AL with 5 (a test number)
        MOV BX,OFFSET TABLE    ;address lookup table
        XLAT                   ;convert
```

- 注意这里的 BX + AL

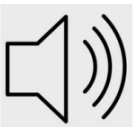
Figure 4–19 shows the operation of aforementioned example program if TABLE = 1000H, DS = 1000H, and the initial value of AL = 05H (BCD). After the translation, AL = 6DH.



- **IN OUT**
  - 对寄存器 A 系列和端口进行数据交换
  - - Two forms of I/O device (port) addressing:
    - *Fixed-port addressing* allows data transfer between AL, AX, or EAX using an 8-bit I/O port address, e.g., IN AL, 12H, OUT 25H, AX
      - port number is a byte-immediate value (00h to FFh) following the instruction's opcode
    - *Variable-port addressing* allows data transfers between AL, AX, or EAX and a 16-bit port address, e.g., IN AL, DX, OUT DX, AX
      - the I/O port number is stored in register DX (0000h to FFFFh), which can be changed (varied) during the execution of a program.

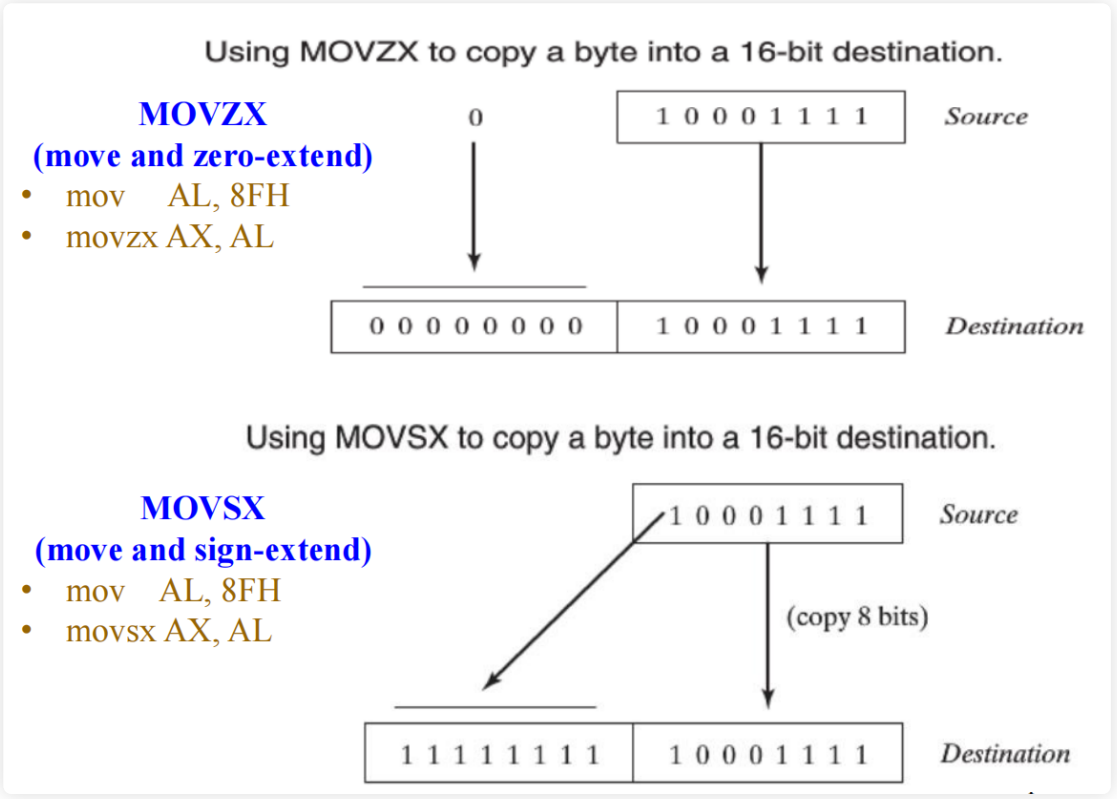
- A program that clicks the speaker in the computer appears in following example.
- The speaker (in DOS only) is controlled by accessing I/O port 61H. If the rightmost 2 bits of this port are set (11) and then cleared (00), a click is heard on the speaker.

```
.MODEL TINY             ;select tiny model
.CODE                   ;start code segment
.STARTUP                ;start program
        IN AL,61H       ;read I/O port 61H
        OR AL,3         ;set rightmost two bits
        OUT 61H,AL      ;speaker on
        MOV CX,8000H    ;load delay count
L1:
        LOOP L1         ;time delay
        IN AL,61H       ;speaker off
        AND AL,0FCH
        OUT 61H,AL
.EXIT
END
```



  - set

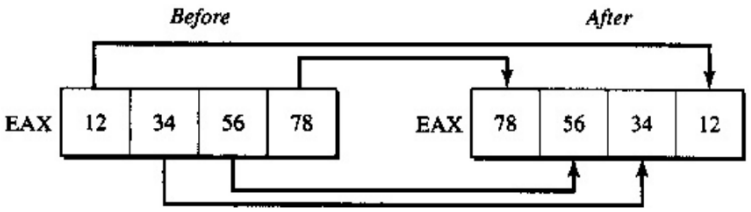
- MOVSX MOVZX
  - 符号拓展 零拓展



- BSWAP

- 大小端模式转换

  - For example, the BSWAP EAX instruction with EAX = 12345678H swaps bytes in EAX, resulting in EAX = 78563412H.



  - The proper byte swapping without using BSWAP is the following:

```
XCHG   AH, AL
ROR    EAX, 16          ; rotate right of EAX
XCHG   AH, AL
```

- **CMOVcc**

  - 条件跳转
  - `COMVZ` zero flag = 1
  - `COMVS` 负数跳转

```
1  `COMVP` 是parity
2
3  ![image-20241228182528349](https://zzh-pic-for-self.oss-cn-
   hangzhou.aliyuncs.com/img/202412281825522.png)
```

- The purpose of CMOV is to avoid a branch.
- When a CPU sees the branch (e.g., JNE) it will take a guess about whether the branch will be taken or not taken, and then start speculatively executing instructions.

```
if (condition)
    a = b;
else
    a = c;
```

original version

```
    cmp  condition
    jz   else
    mov  a, b        ; if part
    jmp  endif
else: mov a, c       ; else part
endif:
```

CMOV version

```
cmp  condition
; else part, mov unconditionally
mov a, c
; if condition is true, move b to a
cmovnz, a, b
```

- 减少由于投机失败的惩罚： cmovcc CPU不会guess
- 基本块更大

SEGMENT Override Prefix

instructions that include segments

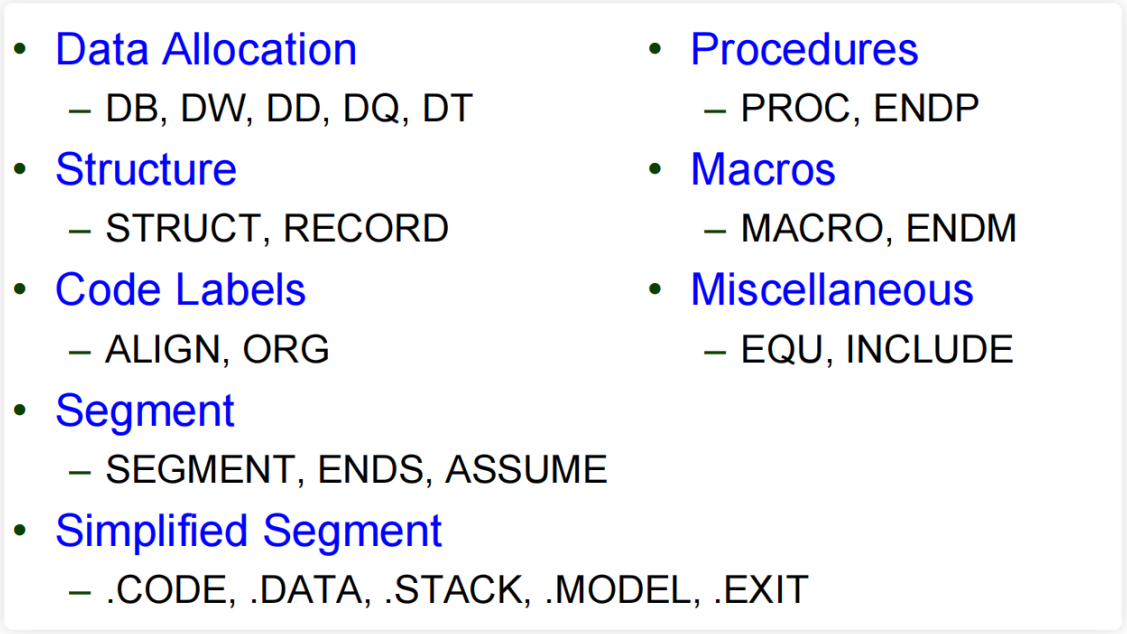
Assembly Language	*Segment Accessed*	*Default Segment*
MOV AX,DS:[BP]	Data	Stack
MOV AX,ES:[BP]	Extra	Stack
MOV AX,SS:[DI]	Stack	Data
MOV AX,CS:LIST	Code	Data
MOV ES:[SI],AX	Extra	Data
LODS ES:DATA1	Extra	Data
MOV EAX,FS:DATA2	FS	Data
MOV GS:[ECX],BL	GS	Data

Assembler Detail

- 知乎文章

伪指令

> “
>
> Directives: tell assembler how to do
>
> Instructions: tell CPU what to do



- `DUP`
  - The DUP directive allows multiple initializations to the same value. e.g., `DB 100 DUP(6)` reserves 100 bytes of 6 进行初始化
  - DUP(?) 未指定

- Storing Data
  - `DB` `DW` `DD` define ...
  - `ALIGN 2` 二字节对齐，不然上面的部分为奇地址

```
LIST_SEG      SEGMENT

DATA1 DB   1,2,3              ;define bytes
      DB   45H                ;hexadecimal
DATA3 DD   300H               ;define doubleword
      DD   2.123              ;real
      DD   3.34E+12           ;real
LISTA DB   ?                  ;reserve 1 byte
LISTB DB   10 DUP(?)          ;reserve 10 bytes

      ALIGN 2                 ;set word boundary

LISTC DW   100H DUP(0)        ;reserve 100H words
LISTD DD   22 DUP(?)          ;reserve 22 doublewords
```

> Problem： P113 例题 必看！

- Problem: Please determine the value in the register AX after the instruction is executed.

```
.data
DB    33H, 34H, 0AH, 06H
DW    1B7CH, 674CH, 07H, '12', '1'

.code
   mov ax, @data
   mov ds, ax

   xor si, si

   mov ax, [si]        ax = 3433H
   mov ax, [si+4]      ax = 1B7CH
   mov ax, [si+5]      ax = 4C1BH
   mov ax, [si+8]      ax = 0007H
   mov ax, [si+10]     ax = 3231H

   .exit
```

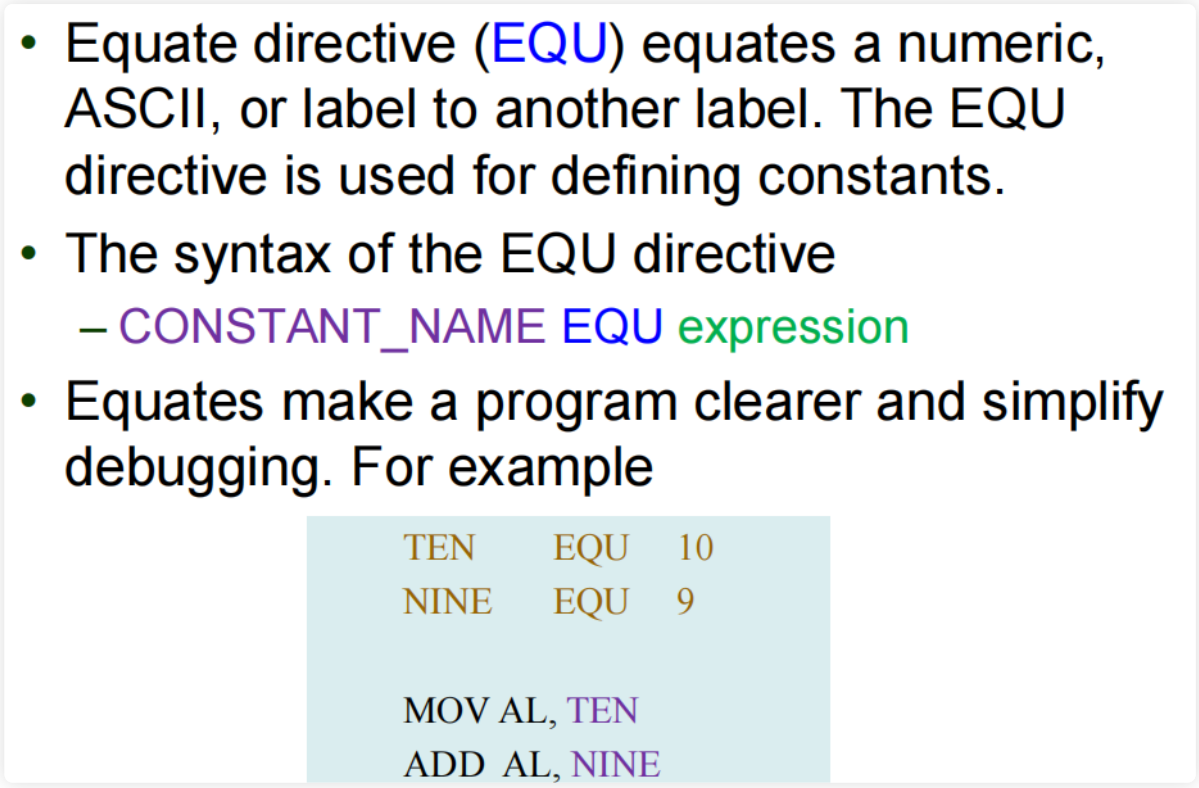
Memory: 33, 34, 0A, 06, 7C, 1B, 4C, 67, 07, 00, 31, 32, 31, 00

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

注意

1. 把内存画出来分析，注意小端模式
2. `DW` 标签有一个 `07H` 内存中需要补0
3. 注意 `string`，放入 ASCII 码

- EQU 可读性考虑，也可以把标签“重命名”
  - EQU伪指令定义的符号名不能与其他符号名重名，符号名必须唯一，且不能被重新定义，而使用等号伪指令定义的符号名可以重名，可以被重新定义、重新赋值。



- ORG
  - 改变入口点，确定它的位置（绝大部分情况不需要）
- ASSUME
  - 段的名字和类型的关系，`ASSUME [保留关键字]:[段的名字]`
- THIS 多功能标签，同一个地址空间但是不同的管理粒度
  - 指定与当前地址计数器相等的一个地址单元的类型

- The ORG (origin) statement can change the starting offset address of the data in the data segment or code in the code segment.
- At times, the origin of data or the code must be assigned to an absolute offset address with the ORG statement. For example, Boot Sector Entry must be assigned to 07c00h.
- ASSUME tells the assembler what names have been chosen for the code, data, extra, and stack segments.

```
;Using the THIS and ORG directives
;
DATA_SEG     SEGMENT

       ORG   300H          <-- use ORG directive to set the location

DATA1  EQU   THIS BYTE    <-- use THIS directive to assign a word
DATA2  DW    ?                address label to a byte label
DATA_SEG     ENDS

                                use SEGMENT directive to
CODE_SEG     SEGMENT 'CODE' <-- define a program segment
       ASSUME CS:CODE_SEG, DS:DATA_SEG
       MOV BL,DATA1
       MOV AX,DATA2
       MOV BH,DATA1+1

CODE_SEG     ENDS
```

- DATA1 DATA2 指向同一个位置
  - DATA1 一开始在低端，取值之后， AX = BX
- LABEL
  - 为当前存储单元定义一个指定类型的变量或标号
- PROC ENDP

```
name PROC   [near/far]
        statements
        ret
name ENDP
```

- 定义Procedure
  - A procedure named *SumOf* calculates the sum of three 32-bit integers by passing register arguments to the procedure.

```
SumOf PROC
     add   eax,ebx
     add   eax,ecx
     ret
SumOf ENDP
```

- Three integers are assigned to EAX, EBX, and ECX.
- The procedure returns the sum in EAX.

```
.data
theSum  DWORD  ?
.code
main PROC
     mov   eax,10000h          ; argument
     mov   ebx,20000h          ; argument
     mov   ecx,30000h          ; argument
     call  Sumof               ; EAX = (EAX + EBX + ECX)
     mov   theSum,eax          ; save the sum
```

Before calling SumOf, values are assigned to EAX, EBX, and ECX.

After the CALL, the sum in EAX are copied to “theSum” variable.

- MACRO ENDM

- 定义宏，可以有参数

  - A macro named mPutchar receives a input and displays it on the console by calling WriteChar.

```
mPutchar MACRO char
    push  eax
    mov   al,char   ; passing arguments to the procedure
    call  WriteChar
    pop   eax
ENDM
```

  - The statement "mPutchar 'A'" invokes mPutchar and passes it the letter A.
  - The assembler's preprocessor expands the statement into the following code:

```
.....
mPutchar 'A'
.....
```

inline expansion

```
push  eax
mov   al,'A'
call  WriteChar
pop   eax
```

  - A macro can also be used in data segment. For example, define a macro for GDT Descriptor.

```
Descriptor MACRO          Base, Limit, Attr
    dw   Limit & 0FFFFh         ; Limit 1 (2 bytes)
    dw   Base & 0FFFFh          ; Base 1        (2 bytes)
    db   (Base >> 16) & 0FFh    ; Base 2        (1 byte)
    dw   ((Limit >> 8) & 0F00h) | (Attr & 0F0FFh)  ; Attr 1 + Limit 2 + Attr 2
    db   (Base >> 24) & 0FFh    ; Base 3 (1 byte)
ENDM ; total  8 bytes
```

```
GDT SEGMENT
      Null_desc:  Descriptor   0, 0, 0
    Normal_desc:  Descriptor   0, 0ffffh, DA_DRW
    ......
GDT ENDS
```

Allocation of GDT Descriptor

- INCLUDE

  -

Memory Organization

- Full-segment

- Full-segment definitions use SEGMENT, ENDS and ASSUME directives to define segment and inform assembler and linker.

```
name SEGMENT [readonly] [align] [combine] [use]
                    ['combine-class']

        statements

name ENDS

cseg SEGMENT readonly word use32 'code'
        mov ax, 10
        inc ax
        ret
cseg ENDS
```

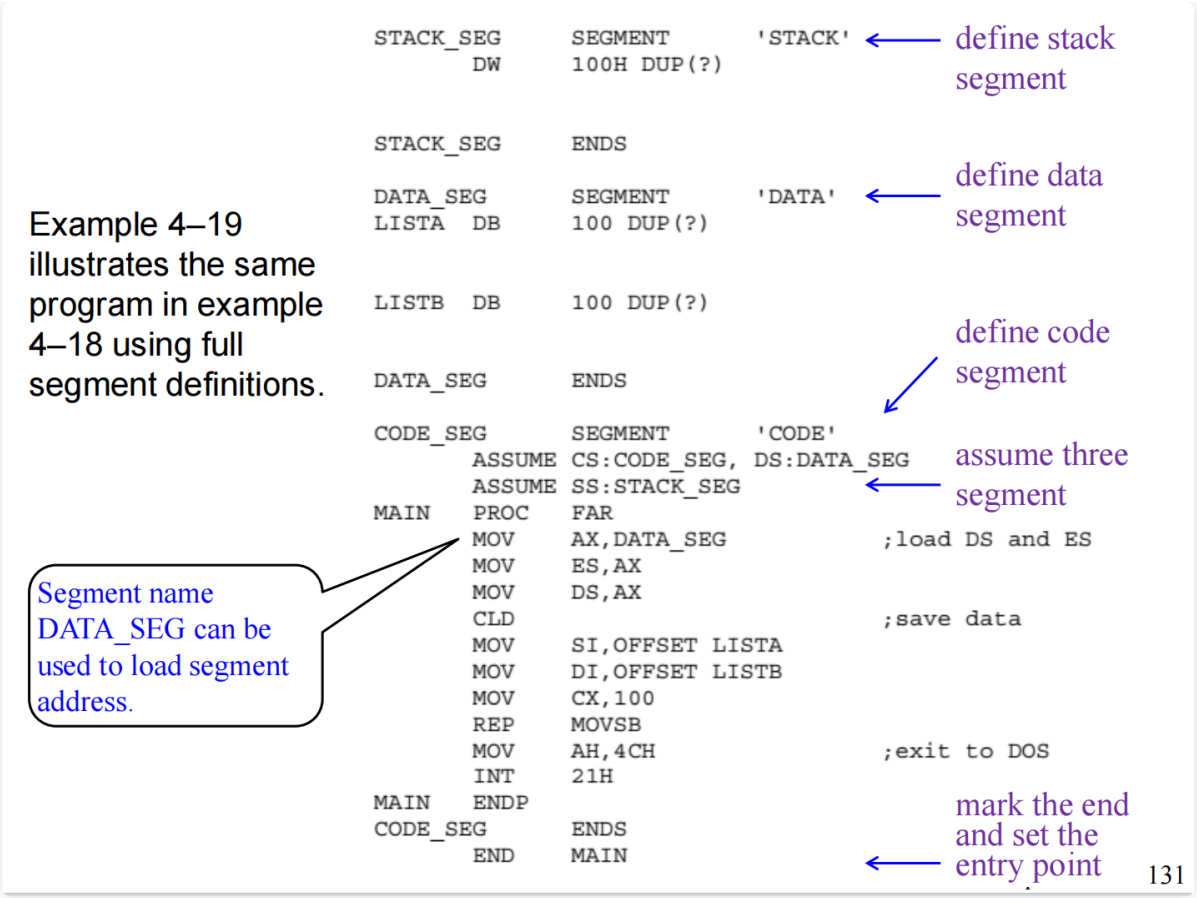

- The assume directive takes the following form:

```
assume [CS:seg,] [DS:seg,] [ES:seg,]
       [FS:seg,] [GS:seg,] [SS:seg]
```

- Examples of valid assume directives:
  - assume DS:DSEG
  - assume CS:CSEG, DS:DSEG, ES:DSEG, SS:SSEG
  - assume CS:CSEG, DS:NOTHING

- The ideal place to put assume directives is before all procedures in a program.

- END
  - The END directive is used to inform the assembler of the end of the module.
  - Generally, each source module will have an END statement as the last line of the module. One and only one module can have an END statement of the form.
  - The end directive can also set the entry point of the program.
  - Syntax: END label
    - E.g., END start
    - In this case, it specifies that label start is the entry point of the program, and DOS will begin execution of the program at that address.




```
STACK_SEG     SEGMENT       'STACK'
              DW            100H DUP(?)

STACK_SEG     ENDS

DATA_SEG      SEGMENT       'DATA'
LISTA  DB     100 DUP(?)

LISTB  DB     100 DUP(?)

DATA_SEG      ENDS

CODE_SEG      SEGMENT       'CODE'
              ASSUME CS:CODE_SEG, DS:DATA_SEG
              ASSUME SS:STACK_SEG
MAIN   PROC   FAR
              MOV    AX,DATA_SEG          ;load DS and ES
              MOV    ES,AX
              MOV    DS,AX
              CLD                         ;save data
              MOV    SI,OFFSET LISTA
              MOV    DI,OFFSET LISTB
              MOV    CX,100
              REP    MOVSB
              MOV    AH,4CH               ;exit to DOS
              INT    21H
MAIN   ENDP
CODE_SEG      ENDS
              END    MAIN
```

- .DATA：初始化数据段。
- .DATA?：未初始化数据段。
- .CODE：代码段。
- .STACK：栈段。

Memory

1. 全段定义
2. uses models
   A. .MODEL
   B. @DATA @STACK 获取指定的段

段寄存器不能直接move进数据，必须通过通用寄存器

```assembly
MOV AX,DATA_SEG
MOV ES,AX
MOV DS,AX
```

- 在代码段开始，要做一部分段的初始化。前期相当于一个占位符，并不知道真正的值

- Memory models are unique to MASM.
- .MODEL directive includes six memory models for the real mode in MASM, namely tiny, small, compact, medium, large, and huge.
- One model (flat) is available for the protected model.
- Special directives such as @DATA, @STACK, @CODE are used to identify various segments.
- .MODEL is not used in MASM for x64.

EXAMPLE 4–18

```
                        .MODEL SMALL          ;select small model
                        .STACK 100H           ;define stack
                        .DATA                 ;start data segment

0000 0064[         LISTA  DB    100 DUP(?)
          ??
            ]
0064 0064[         LISTB  DB    100 DUP(?)
          ??
            ]

                        .CODE                 ;start code segment

0000 B9 ---- R     HERE:  MOV   AX,@DATA      ;load ES and DS
0003 8E C0                MOV   ES,AX
0005 8E D8                MOV   DS,AX
0007 FC                   CLD                 ;move data
0008 BE 0000 R            MOV   SI,OFFSET LISTA
000B BF 0064 R            MOV   DI,OFFSET LISTB
000E B9 0064              MOV   CX,100
0011 F3/A4                REP   MOVSB

0013                      .EXIT 0             ;exit to DOS
                          END HERE
```

uses @DATA to load segment address.

Ch 5 Arithmetic and Logic Ins

除了取非（Logic NOT）都会设置flag

- 一旦出现填充问题，就要去区分究竟是正数还是负数

Addition

- ADD 半加
- ADC 带进位的，全加器 P15

```
.data
int1    DD      1,0,0,0,0,0,0,1
int2    DD      1,0,0,0,0,0,0,1

.code
        MOV   EDI,OFFSET int1
        MOV   ESI,OFFSET int2
        MOV   ECX,8             ; ECX = 8
        CLC                     ; start out with carry flag cleared
loop:   MOV   EAX, [ESI]
        ADC   [EDI], EAX        ; with carry from previous loop pass
        LEA   ESI, [ESI+4]      ; point to next source
        LEA   EDI, [EDI+4]      ; point to next destination
        DEC   ECX               ; adjust loop count
        JNZ   loop              ; if ECX > 0 then repeat
```

为了保持Hcy使用Mov.

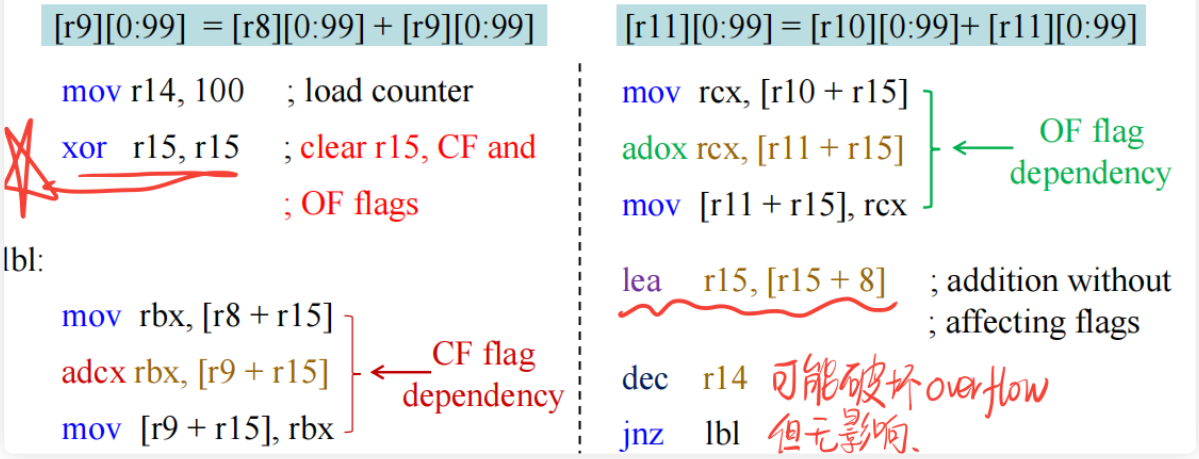
  - 分段累加
  - CLC 清除Carry
  - 过程中不要对Carry变化
    - 使用LEA，INC，DEC

```Assembly
LEA r15,[r15 + 8]
```

  - ADCX ADOX 为了并行性 P17 *
    - 使用不同的进位链，Carry Overflow

– ADCX uses the Carry flag as source and destination of overflow and leaves the other flags untouched.
– ADOX uses the Overflow flag as source and destination of overflow and leaves the other flags untouched.

An example of two parallel addition of numbers



- INC
  - 对寄存器、内存(必须使用长度的伪指令)，不允许对段寄存器
    `INC BL` `INC WORD PTR[SI]`
  - **保持Carry flag不变其他都可能变化**
    - 即使产生进位也不会变化
    - 为循环递增变量不产生干扰
      - 防止递增变量INC产生进位导致Flag变化

P10 16bit sum

- **XADD** *exchange and add* `XADD des, src` P22
  - 先交换 `des src`， 然后 `des = des + src`
  - 可以作为锁
    - 乐观锁 冲突可能性小
      - 先进行操作
      - 通过一个version标签
    - 如果前后版本号不一致需要rollback

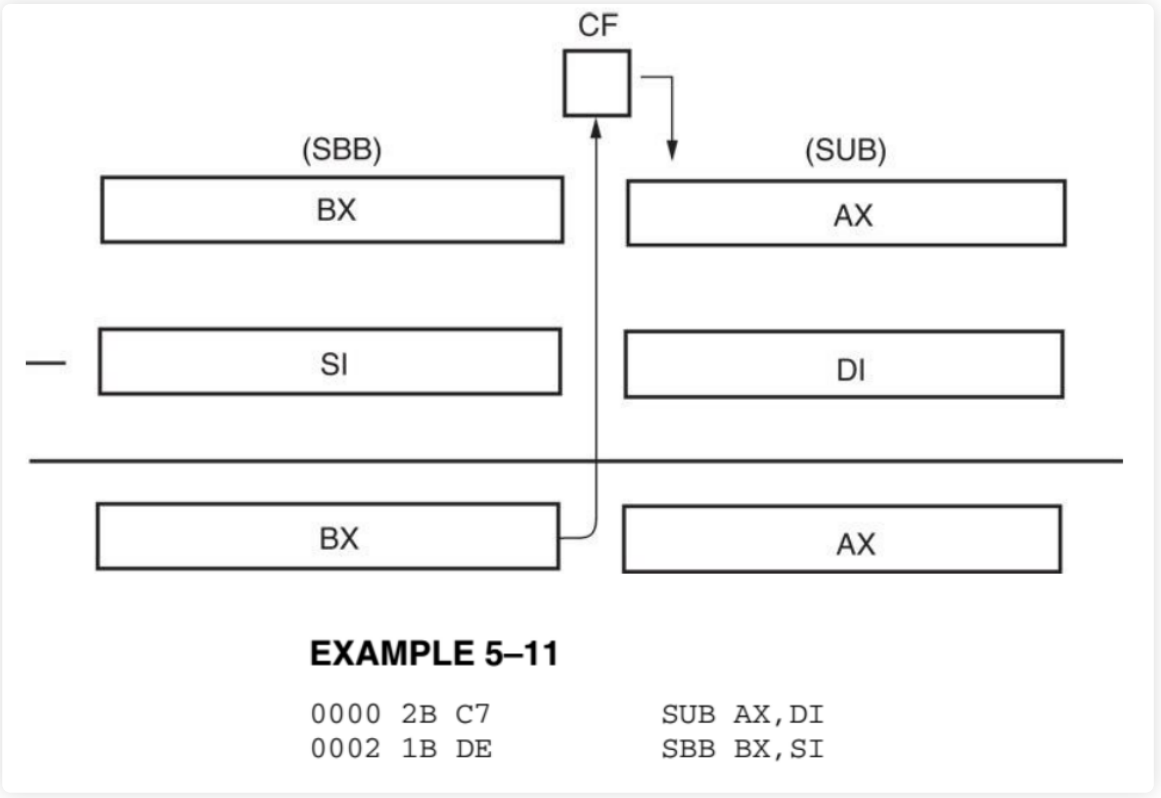
- XADD might be useful for optimistic locking, which is most applicable to high-volume systems.
- The following example uses optimistic locking to update a shared version by multiple threads safely.

```
.data
version   DD   0          ; shared version number initialized to 0

.code
          MOV  ECX, version  ; load the current value of version
          ......             ; working optimistically
          MOV  EAX, 1        ; EAX = 1
          XADD version, EAX  ; version ⇔ EAX, version = version+1
          CMP  EAX, ECX      ; check if the value was modified by
                             ; another thread
          JNE  retry         ; if version was updated then rollback
          ......
rollback: ......             ; handle the conflict
```


Subtraction

- SUB
- DEC
  - 只改变CF

- SBB 带 borrow 的



EXAMPLE 5–11

```
0000 2B C7          SUB AX,DI
0002 1B DE          SBB BX,SI
```

- CMP
  - 更新所有的flag
  - 不更改目的寄存器

- **CMPXCHG des, src (AL/AX/EAX)** P29

  - For example, CMPXCHG CX,DX (AX)
    - if CX == AX, CX = DX, ZF =1
    - if CX <> AX, AX = CX, ZF =0
  - The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared.

    Case 1: before execution:

    (CX)=00FFH, (DX)=00EFH, (AX)=00FFH;

    after execution:

    (CX)=00EFH, (DX)=00EFH, (AX)=00FFH, ZF=1;

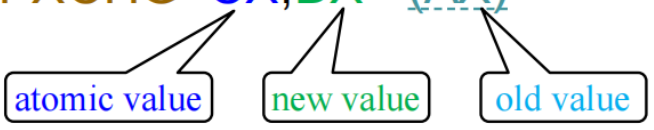
    Case 2: before execution:

    (CX)=00FFH, (DX)=00EFH, (AX)=00EEH ;

    after execution:

    (CX)=00FFH, (DX)=00EFH, (AX)=00FFH, ZF=0;

  For example, CMPXCHG CX,DX (AX)

  - if CX equals AX, DX is copied into CX;
  - if CX is not equal to AX, CX is copied into AX;
  - AX holds the value of CX before execution.

  - 可以作为锁 P33 *
    - 无锁数据结构
  - 先CMP后Exchange
    - des == accu, then des = src, ZF =1
    - CMP des 和 累加寄存器的值
- CMPXCHG8B CMPXCHG8B [mem64-operand]
- CMPXCHG16B

Multiplication

- 分为有符号和无符号
  - 有符号（`IMUL`）
    - 可能存在多个 2 / 3 操作数
    - 可能使用imm
    - 高位零拓展
  - 无符号 `MUL`
    - 单个操作数，乘数
    - 不能使用imm
    - 高位符号拓展
- multiplicand 隐含被乘数 放在A系列寄存器
- 乘数可以为内存 `MUL WORD PTR [BX]`
- 8 bit
  - AX = AL *
- 16 bit
  - DX:AX = AX *
- 32 bit
  - EDX:EAX = EAX *

IMUL P45

- 1 操作数
- 2 操作数
  - 可能会被截断 truncated
    - Product 不脱战
  - 可以引入立即数
    - signed extension
  - 会被汇编为三操作数
    - `IMUL BX,16H` -> `IMMUL BX,BX,16H`
- 3 操作数

– two-operand form: the destination operand is a register and the source operand is an immediate value, a register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.

e.g., IMUL ECX, [EAX+4] ; ECX = ECX * [EAX+4]

IMUL ECX, 16 ; ECX = ECX * 16

– three-operand form: the first source operand is multiplied by the second source operand. The intermediate product is truncated and stored in the destination operand.

e.g., IMUL ECX, [EAX+4], 5 ; ECX = [EAX+4] * 5

Flags

判断高位是否存在有效位

- When the product fits completely within the lower register of the product, the MUL instruction clears OF and CF flags, otherwise, OF and CF flags are set, e.g.,

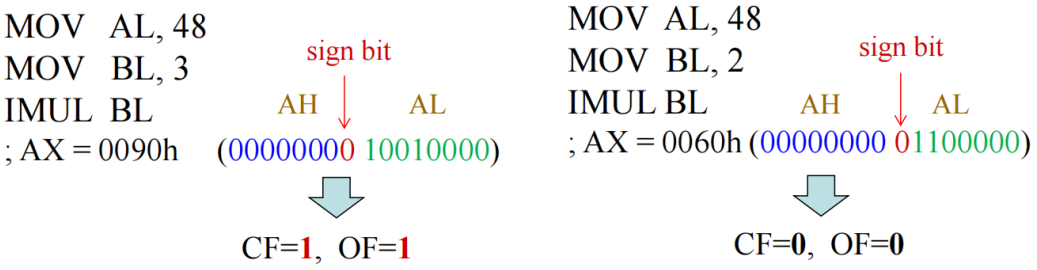
```
MOV  AL, 48
MOV  BL, 3
MUL  BL          AH       AL
; AX = 0090h   (00000000 10010000)
                  ⇩
              CF=0,  OF=0
```

```
MOV  AL, 48
MOV  BL, 8
MUL  BL          AH       AL
; AX = 0180h  (00000001 10000000)
                  ⇩
              CF=1,  OF=1
```

- CF
- OF
- IMUL必须判断

- 截断前后的结果是否一致，判断结果是否正确

- When the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, the CF and OF flags are set; otherwise the CF and OF flags are cleared.



- With the two and three-operand forms, because of the truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

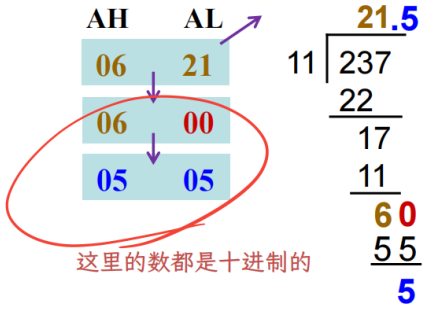
Division P50

- 除法不能有立即数
- DIV
- IDIV

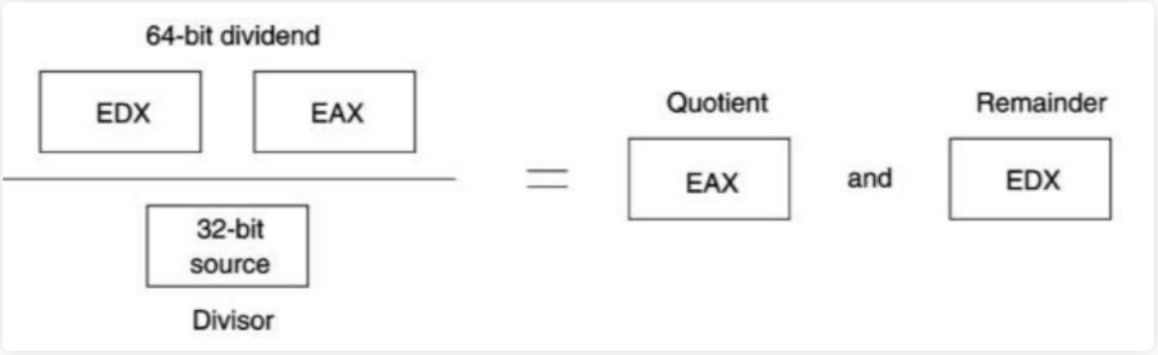
errors

- overflow
- divide by zero
- 8 bit P52
  - 使用 16 bit 的 AX 作为隐含寄存器
  - 除数可以是寄存器/memory
  - AL：商 Quotient
  - AH：余数 Remainder

- For example,
  - MOV AX, 237 ; AX = 237
  - MOV CL, 11 ; CL = 11
  - DIV CL ; AH: AL = AX÷CL
    ; AH = 6, AL = 21



- 16 bit
  - AX 商
  - DX 余数
  - 被除数 DX：AX
- 32 bit



- IDIV P55
  - round to zero 的舍入方式
    - -16 / 3 = -5 ••• -1
    - 余数的符号和被除数相同

填充指令 P59

- signed
  - CBW/CWDE/CDQE
    - 针对A类寄存器

        - Convert byte word
    - CWD/CDQ/CQO
        - 将A类寄存器拓展到D类
            - 比如在16bit除法，被除数需要是DX ： AX

```
MOV   AX,-100          ;load a -100
MOV   CX,9             ;load +9
CWD                    ; convert the signed 16-bit number in AX
IDIV   CX              ; to a 32-bit signed number in DX: AX
```

16-bit division

            - DX:AX
    - **MOVSX reg, reg/mem**
- MOVZX reg，reg/mem

余数处理

-

BCD | ASCII

BCD

- 先做正常加法之后做调整
- DAA
    - 对AL寄存器

- 跟在加法之后

- AL is the implied source and destination operand.
- Example 1: calculate BCD 35+48

```
MOV AL, 35H
ADD AL, 48H      ; AL = 7DH, AF = 0
DAA              ; AL = 83H, CF = 0
```

Auxiliary carry (half-carry)
进位产生在AF

- Example 2: calculate BCD 69+29

什么时候需要+6调整：
1. 大于10
2. AF = 1，在ADD存在进位

```
MOV AL, 69H
ADD AL, 29H      ; AL = 92H, AF = 1
DAA              ; AL = 98H, CF = 0
```

- Example 3: calculate BCD 35+65

```
MOV AL, 35H
ADD AL, 65H      ; AL = 9AH, AF = 0
DAA              ; AL = 00H, CF = 1
```

这里存在Carry

- DAS

ASCII

- AAA
  - 生成的是 unpacked 的BCD码结果，每一个数字占一个字节

- Use the AAA instruction after using the ADD instruction to add two unpacked BCD numbers.
- The AL register is the implied source and destination operand for AAA instruction.
- The AAA instruction adjusts the value in AL to an unpacked BCD result.
  - If the AL[3:0] > 9 or AF = 1, then AL = AL +6, AH = 1 and sets CF = 1 and AF = 1. 展开为两个BCD码
  - else, CF = 0 and AF = 0.
  - In either case, AL[7:4] = 0, leaving the correct decimal digit in AL[3:0]. 产生unpacked BCD，无论如何AL的高四位都要清空
- The AAA instruction adds ASCII numbers without having to mask off the upper nibble '3'.

- Example 1

```
MOV  AL, '3'     ;  AL = 0x33 (ASCII for '3')
ADD  AL, '4'     ;  AL = 0x67
                 ;  ASCII result for '3' + '4' = 0x33 + 0x34 = 0x67
AAA              ;  AH = 0, AL = 07, CF = 0 and AF = 0
```

高位就是“ overflow ”的结果，每个BCD码一个字节

- Example 2

```
MOV  AX, '1'     ;  AL = 0x31 (ASCII for '1')
ADD  AL, '9'     ;  AL = 0x6A
                 ;  ASCII result for '1' + '9' = 0x31 + 0x39 = 0x6A
AAA              ;  AH = 1, AL = 0, CF = 1 and AF = 1
```

Logic Instruction

- Logic operations always: 注意任何logic都会
  - clear the OF and CF flags
  - change the SF, ZF, and PF flags to reflect the result
  - the state of the AF flag is undefined

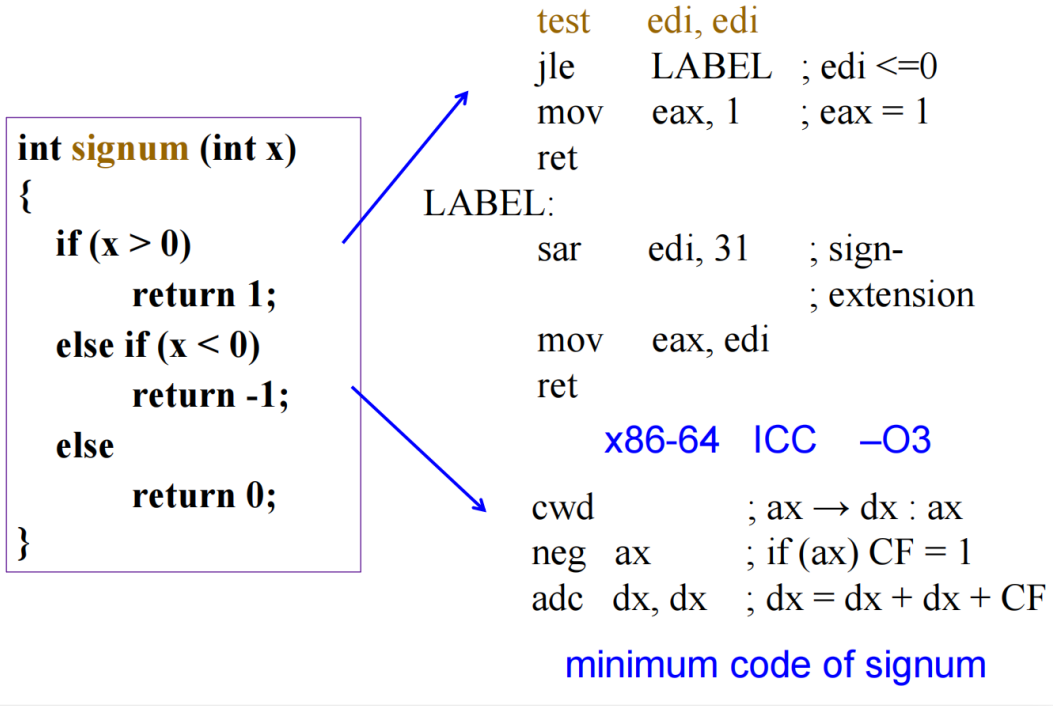
- NOT 取反
  - 不会改变Flag
- NEG 取补
  - while the NEG instruction affects flag bits as follows:

注意取反的CF位变化

  - if the operand = 0, then CF = 0, otherwise CF = 1.
  - the OF, SF, ZF, AF, and PF flags are set according to the result.

- `neg ax` if ax = 0, CF = 0

- An interesting example of signum function:



- AND
  - masking
- OR
- XOR
  - 可以用于清空所有的Flags `XOR AL,AL ; AL = 0`
- TEST

- 测试，执行AND 影响flag而不影响操作数

EXAMPLE 5–28 tests the rightmost and leftmost bit positions of the AL

```
TEST AL,1          ;test right bit
JNZ  RIGHT         ;if set
TEST AL,128        ;test left bit
JNZ  LEFT          ;if set
```

- CMP and TEST are the instructions that are commonly used for comparison, and these instructions are known as conditionals, e.g.,

```
        MOV  EAX, 1                          MOV  EAX, 1
减法    CMP  EAX, 1  ; C=0,Z=1,S=0,O=0       TEST EAX, 1  ; C=0,Z=0,S=0,O=0
        JE   LABEL ; jump to LABEL           JE   LABEL ; do not jump
```

- TEST same,same is used to determine if signed numbers are greater than zero, e.g.,

判断这个值的正负/0

```
TEST EAX, EAX   ; if EAX = 0 set Z = 1, if EAX < 0 set S = 1
JLE  ERROR      ; if EAX is equal or less than zero then jump
```

- TEST EAX,EAX is almost identical to CMP EAX, 0, except that it is shorter than CMP.

```
TEST EAX, EAX ; 85 C0          CMP EAX, 0 ; 83 F8 00
```

Test的指令编码更短

- bit test指令 设置的是CF
  - BT
  - BTS
  - BTR

- BTC

The following examples show bit operations on the CX using logic and bit test instructions, respectively.

```
1)  OR    CX,0600H        ;set bits 9 and 10
    AND   CX,0FFFCH       ;clear bits 0 and 1
    XOR   CX,1000H        ;invert bit 12
```

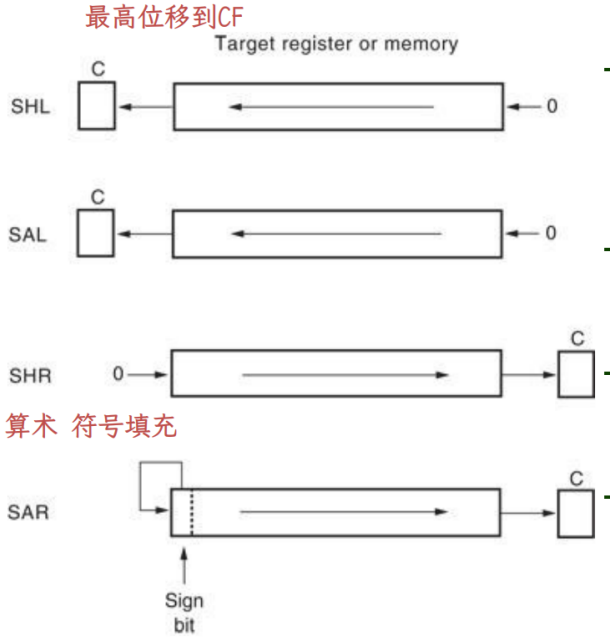
XOR反转

```
2)  BTS   CX,9            ;set bit 9
    BTS   CX,10           ;set bit 10
    BTR   CX,0            ;clear bit 0
    BTR   CX,1            ;clear bit 1
    BTC   CX,12           ;complement bit 12
```

Shift

- 实际上，移动的这个值是一个取模的值

最高位移到CF



– SHL and SAL shift the bits in the destination operand to the left

– SHL and SAL perform the same operation

– SHR copies a 0 through the number

– SAR copies the sign-bit through the number

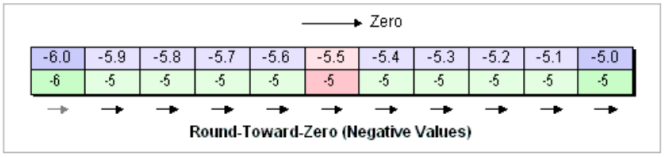
Assembly Language	*Operation*
SHL AX,1	AX is logically shifted left 1 place
SHR BX,12	BX is logically shifted right 12 places
SHR ECX,10	ECX is logically shifted right 10 places
SHL RAX,50	RAX is logically shifted left 50 places (64-bit mode)
SAL DATA1,CL	The contents of data segment memory location DATA1 are arithmetically shifted left the number of spaces specified by CL
SHR RAX,CL	RAX is logically shifted right the number of spaces specified by CL (64-bit mode)
SAR SI,2	SI is arithmetically shifted right 2 places
SAR EDX,14	EDX is arithmetically shifted right 14 places

要么是imm
要么指定CL寄存器

- **SAR**
  - 在移位下，需要对被除数做调整。
  - # SAR Rounding for Negative Numbers

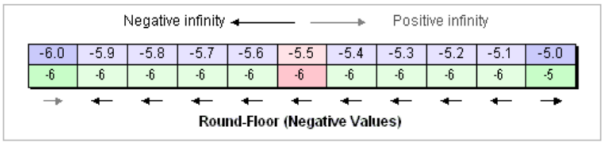
    - For negative numbers, the quotient from the IDIV is rounded toward zero, while the SAR is rounded toward negative infinity, producing inconsistent result. For example
      - IDIV: divide -9 by 4, the result is -2 with a remainder of -1 (-9÷4 = -2 R -1). ← -9÷4 = -2.25

    Round-Toward-Zero

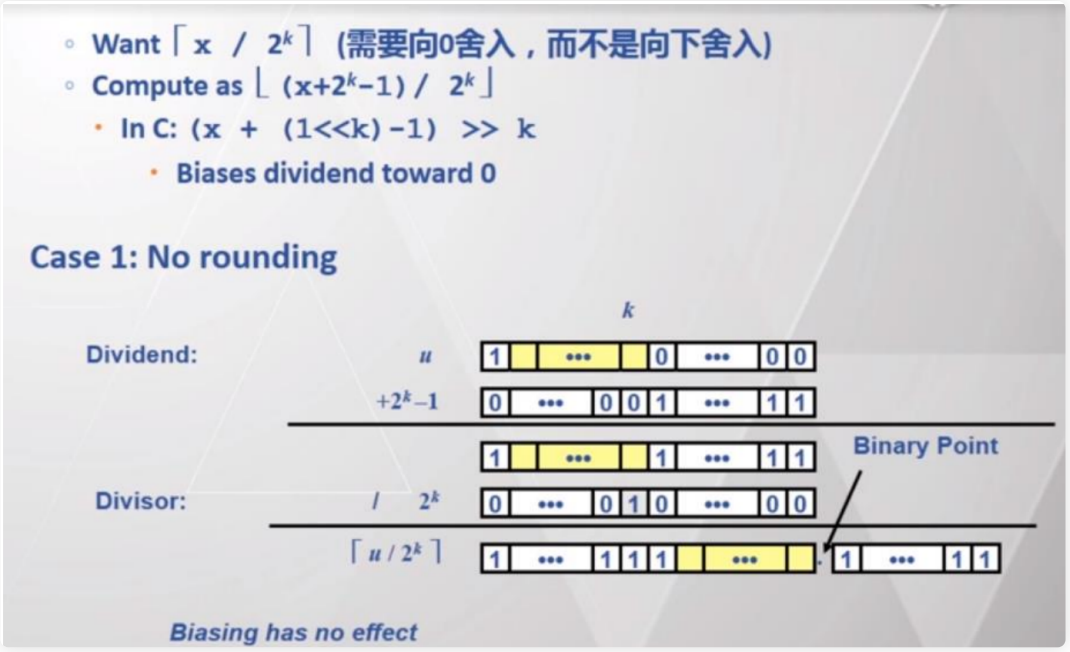
      - SAR: shift -9 (0x11110111) right by two, the result is -3 and the "remainder" is +3 (-9÷4 = -3 R +3).

    Round-Toward Negative Infinity

    Chapter 5 10
  - 如何修正？

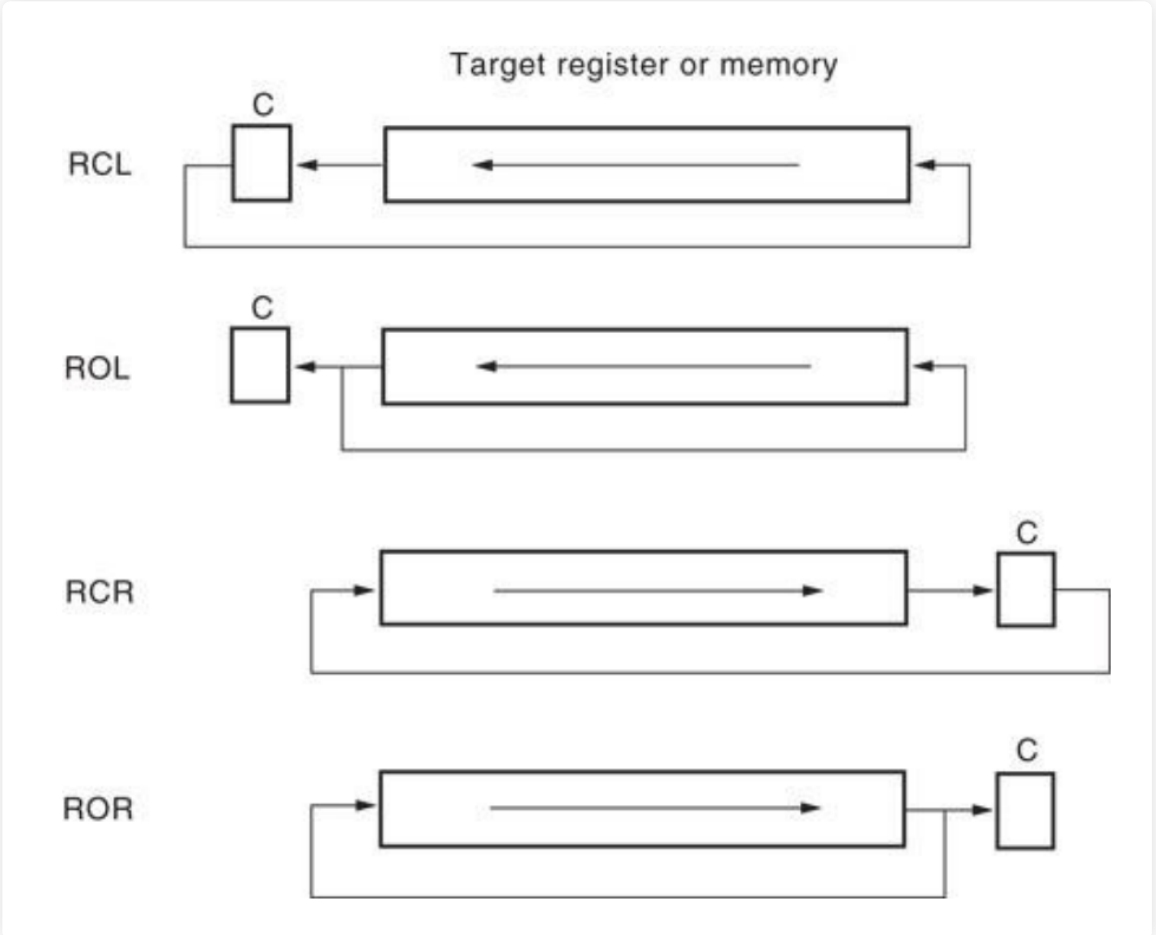
- 被除数修正



- 右移k bit

Rotate

- ROL/ROR/RCL/RCR REG/MEM, Count
-



Bit scan

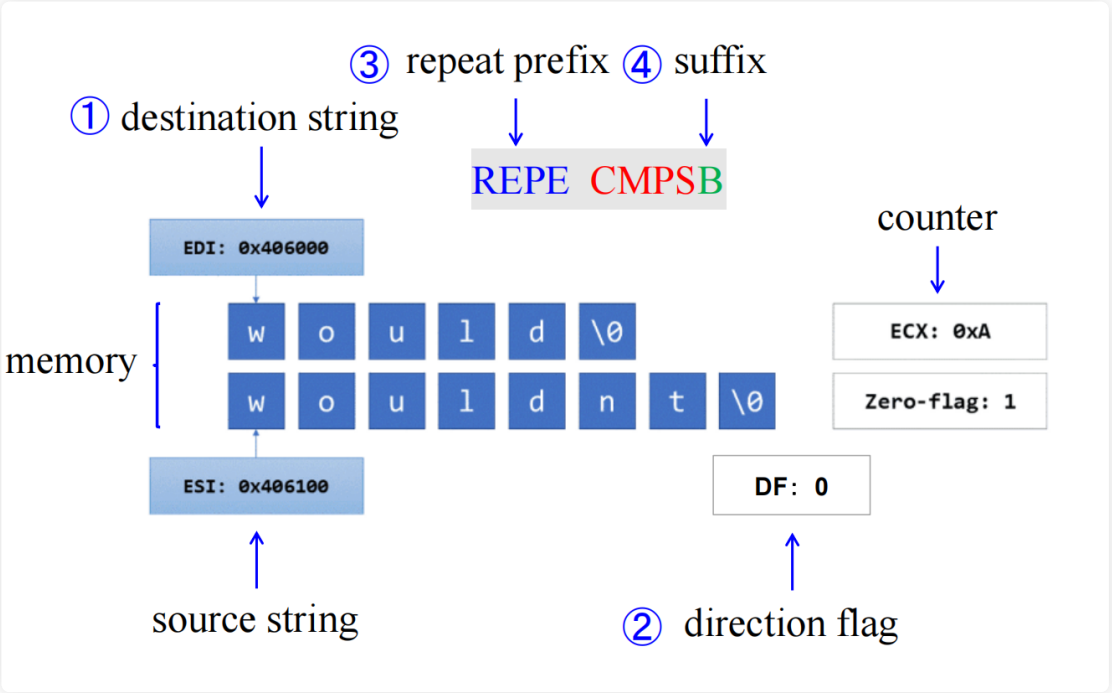
- :TODO

Bit Scan Instructions 位扫描

- Scan through a number searching for a 1-bit.
  - accomplished by shifting the number
  - available in 80386–Pentium 4
- BSF (bit scan forward) scans the source number from the least significant bit toward the left. 到最高有效位
- BSR (bit scan reverse) scans the source number from the most significant bit toward the right.
  - BSF/BSR REG, REG/MEM
  - if no 1-bit is encountered the zero flag is set (ZF = 1) 全零ZF
  - if a 1-bit is encountered, the zero flag is cleared (ZF = 0) and the bit position number of the 1-bit is placed into the destination operand

发现的第一个1的位置被放入des

- **CMPS**
  - if CX = 0 and ZF =1, then two strings match.
  - if CX ≠ 0 or ZF = 0, then the strings do not match
  -



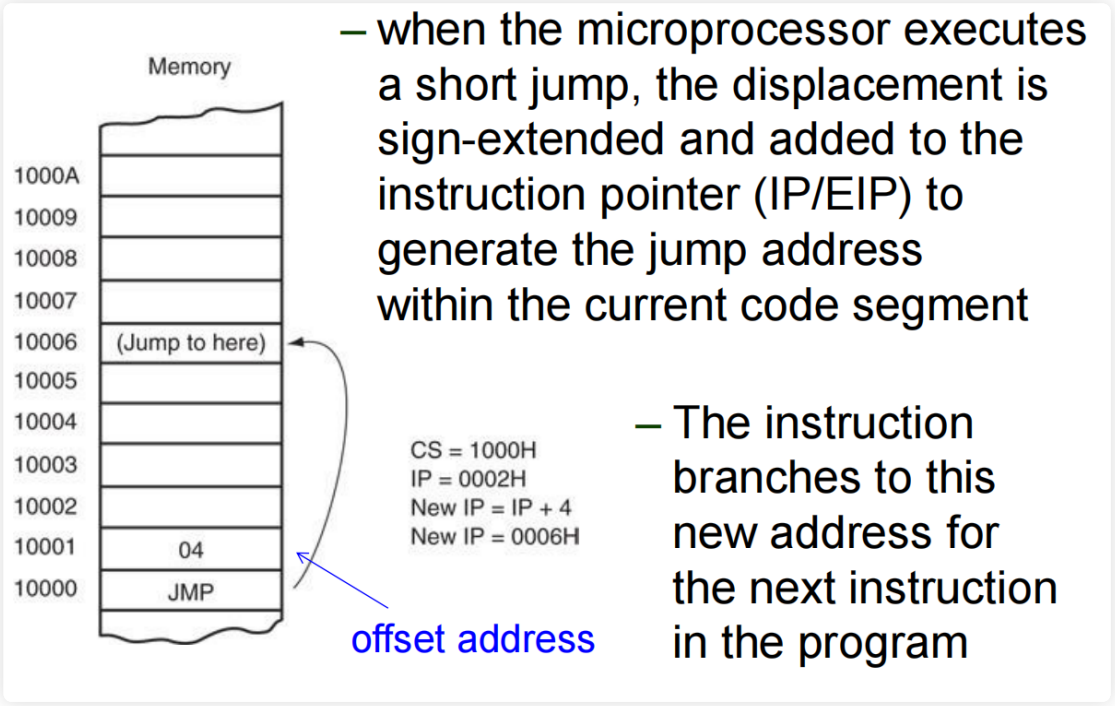
Ch 6 跳转

无条件跳转

- JMP
  - JMP imm
  - JMP reg 直接跳转到对应

跳转类型

- Short
  - 两字节指令，offset只有一个字节
  - 注意这张图中的IP位置



- Near

- 两个字节的offset

  - Notice that the instruction assembles as E9 0200 R. The letter R denotes a **relocatable jump address** of 0200H.
  - After linking, the jump instruction appears as E9 F6 01 (01F6H is the actual displacement).

```
0000 33DB                       XOR   BX,BX
0002 B8 0001              START: MOV   AX,1
0005 03 C3                       ADD   AX,BX
0007 E9 0200 R                   JMP   NEXT
000A                             <skipped memory locations>

0200 8B D8                NEXT:  MOV   BX,AX
0202 E9 0002 R                   JMP   START
```

0200H – 000AH = 01F6H

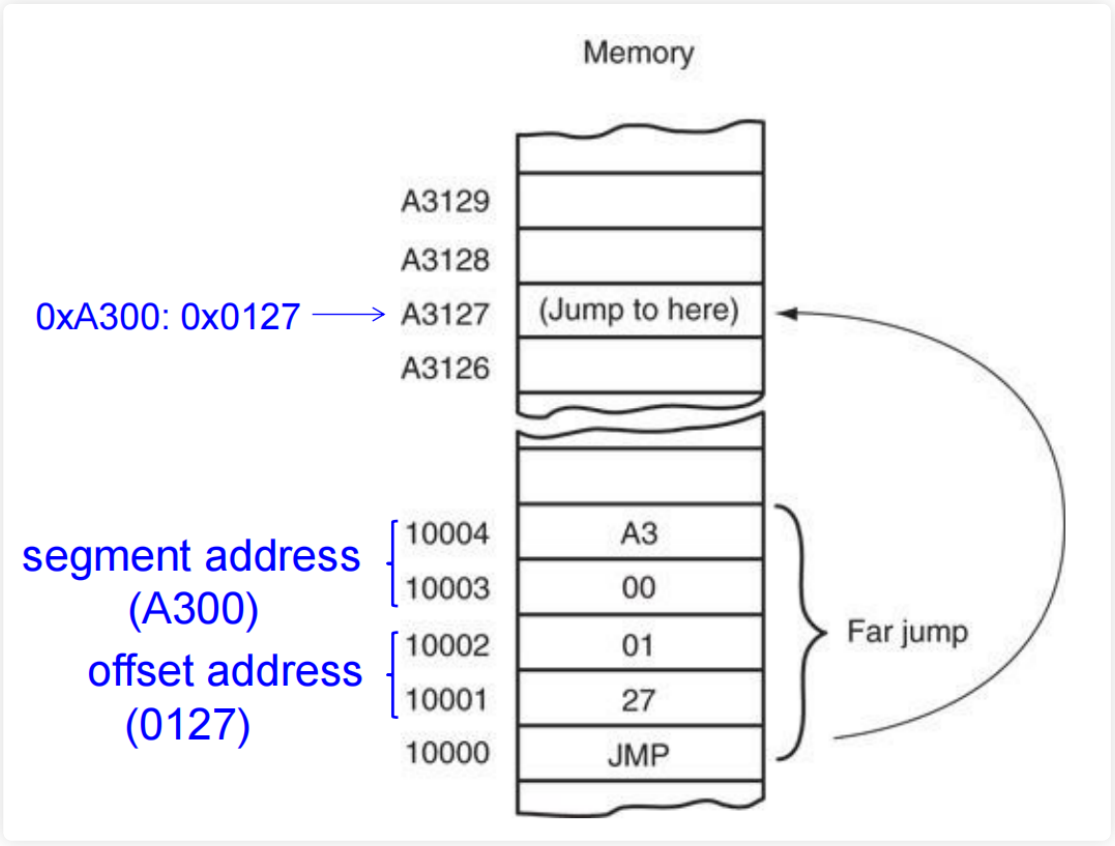
这里通过两次对代码的扫描，
第一次是把段内要跳过去的地址写到这里，同时写入重定位表；
第二次检测到重定位，需要计算对应的offset，绝对地址-下一条指令的地址

将offset写入的时候要注意顺序，低位先写，E9 F6 01

6

- Far
  - JMP TABLE [SI]
    -
  - JMP FAR PTR [SI] 或者 JMP TABLE [SI] 并且TABLE data被定义为DD
  - 怎样表示
    - JMP FAR PTR START
    - 定义标签 EXTRN START:FAR 或者是 START::

- 先offset后seg



有条件 JCC

- 注意对有符号数无符号数两种比较

无符号数

- When unsigned numbers are compared, use the JA, JB, JAE, JBE, JE, and JNE instructions.
  - terms *above* and *below* refer to unsigned numbers
- When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions.
  - terms *greater than* and *less than* refer to signed numbers

TABLE 6–1 Conditional jump instructions.

	Assembly Language	*Tested Condition*	*Operation*
for unsigned numbers	JA	Z = 0 and C = 0	Jump if above
	JAE	C = 0	Jump if above or equal
	JB	C = 1	Jump if below
	JBE	Z = 1 or C = 1	Jump if below or equal
	JC	C = 1	Jump if carry
	JE or JZ	Z = 1	Jump if equal or jump if zero
for signed numbers	JG	Z = 0 and S = 0	Jump if greater than
	JGE	S = 0	Jump if greater than or equal
	JL	S != O	Jump if less than
	JLE	Z = 1 or S != O	Jump if less than or equal
	JNC	C = 0	Jump if no carry
	JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
	JNO	O = 0	Jump if no overflow
	JNS	S = 0	Jump if no sign (positive)
	JNP or JPO	P = 0	Jump if no parity or jump if parity odd
	JO	O = 1	Jump if overflow
	JP or JPE	P = 1	Jump if parity or jump if parity even
	JS	S = 1	Jump if sign (negative)
test contents of CX/ECX/RCX	JCXZ	CX = 0	Jump if CX is zero
	JECXZ	ECX = 0	Jump if ECX equals zero
	JRCXZ	RCX = 0	Jump if RCX equals zero (64-bit mode)

Conditional Set

TABLE 6–2 Conditional set instructions.

Assembly Language	*Tested Condition*	*Operation*
SETA	Z = 0 and C = 0	Set if above
SETAE	C = 0	Set if above or equal
SETB	C = 1	Set if below
SETBE	Z = 1 or C = 1	Set if below or equal
SETC	C = 1	Set if carry
SETE or SETZ	Z = 1	Set if equal or set if zero
SETG	Z = 0 and S = 0	Set if greater than
SETGE	S = 0	Set if greater than or equal
SETL	S != O	Set if less than
SETLE	Z = 1 or S != O	Set if less than or equal
SETNC	C = 0	Set if no carry
SETNE or SETNZ	Z = 0	Set if not equal or set if not zero
SETNO	O = 0	Set if no overflow
SETNS	S = 0	Set if no sign (positive)
SETNP or SETPO	P = 0	Set if no parity or set if parity odd
SETO	O = 1	Set if overflow
SETP or SETPE	P = 1	Set if parity or set if parity even
SETS	S = 1	Set if sign (negative)

Loop

- `LOOP Label`
  - 使用C系列寄存器
  - 不改变Flag
- `LOOPE | LOOPZ`
  - `CX != 0 && Equal` 继续循环
  - 循环条件为Equal，找首个不匹配
- `LOOPNE | LOOPNZ`
  - 循环条件为not Equal，找首个匹配

- The following code finds the first positive value in an array:

```
.data
array DW -3,-6,-1,-10,10,30,40,4

.code
   MOV ESI,   OFFSET array
   MOV ECX,  LENGTHOF array      ; get the number of array
next:                                  negative      positive
   TEST WORD PTR [ESI],8000h     ; sign bit=1, Z=0, else Z=1
   PUSHF                         ; protect flags on stack
   ADD  ESI, 2                   ; inc ESI to next value
   POPF                          ; restore flags from stack
   LOOPNZ next                   ; if Z=0 and ECX≠0 continue
   JNZ   quit                    ; if Z=0, not found
   SUB   ESI, 2                  ; ESI points to value
quit:
```

可能影响flags

控制流的伪指令

- .WHILE
- .ENDW

- .REPEAT
- .UNTIL
- .UNTILCXZ

- .BREAK
- .CONTINUE

- .IF
- .ENDIF
- .ELSE
- .ELSEIF

Operators for Conditional Control Flow

- ! (logical not)
- != (not equal)
- || (logical or)
- && (logical and)
- < (less than)
- <= (less or equal)
- == (equal)
- > (greater than)
- >= (greater or equal)
- & (bitwise and)

- CARRY? (carry test)
- PARITY? (parity test)
- SIGN? (sign test)
- ZERO? (zero test)
- OVERFLOW? (overflow test)

- .REPEAT
  - .UNTIL AL == 0DH
  - .UNTILCXZ

```
        .CODE                           ;start code segment
        .STARTUP                        ;start program
              MOV   AX,DX               ;overlap DS with ES
              MOV   ES,AX
              CLD                       ;select auto-increment
              MOV   DI,OFFSET BUF       ;address buffer

              .REPEAT                   ;repeat until enter
              至少读取一次，do-while
*       @C0001:
              MOV   AH,1                ;read key
              INT   21H
              STOSB                     ;store key code

              .UNTIL AL == 0DH

*             cmp   al,0dh
*             jne   @C0001
              MOV   BYTE PTR[DI-1], '$'
              MOV   DX,OFFSET BUF
              MOV   AH,9
              INT   21H                 ;display BUF
        .EXIT
        END
```

- The following code uses the .UNTILCXZ to add the contents of byte-sized array ONE to byte-sized array TWO. The sums are stored in array THREE.

```
            MOV   CX,100                ;set count
            MOV   DI,OFFSET THREE       ;address arrays
            MOV   SI,OFFSET ONE
            MOV   BX,OFFSET TWO

            .REPEAT

* @C0001:
            LODSB  SI 自动加 1
            ADD AL,[BX]
            STOSB  DI 自动加 1
            INC BX

            .UNTILCXZ

*           LOOP @C0001
```

↑[DI] ↑[SI]

[THREE][0:99] = [ONE][0:99]
+ [TWO][0:99]

↓[BX]

Procedures

- NEAR
- FAR：全局的函数

```Assembly
1  Name PROC NEAR/FAR USE BX CX DX
2      USES ;
3      RET ;
4  Name ENDP
```

```
                0000                    SUMS    PROC    NEAR
                0000 03 C3                      ADD     AX,BX
                0002 03 C1                      ADD     AX,CX
                0004 03 C2                      ADD     AX,DX
                0006 C3                         RET
Near 只要offset  0007                    SUMS    ENDP

                0007                    SUMS1   PROC    FAR
                0007 03 C3                      ADD     AX,BX
                0009 03 C1                      ADD     AX,CX
                000B 03 C2                      ADD     AX,DX
                000D CB                         RET
                000E                    SUMS1   ENDP
far还需要其他信息

                000E                    SUMS3   PROC    NEAR USE BX CX DX
                0011 03 C3                      ADD     AX,BX
                0013 03 C1                      ADD     AX,CX
                0015 03 C2                      ADD     AX,DX
                                                RET
                001B                    SUMS    ENDP
```

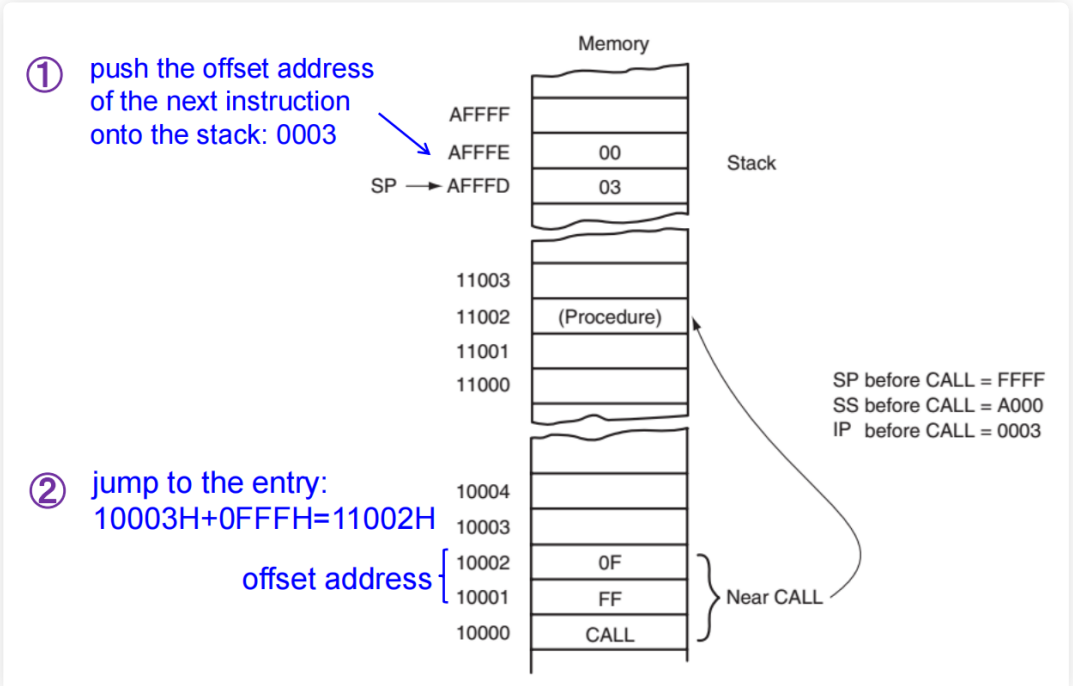
near return opcode

far return opcode

USES statement

USE ： 执行之前自动做保存之后做恢复

- CALL
  - Near
    - 3 byte long，两个字节的displacement
    - **先把下一条指令的地址的 offset 入栈**
    - CALL AX 绝对的CALL
    - CALL <LABEL> 相对的CALL



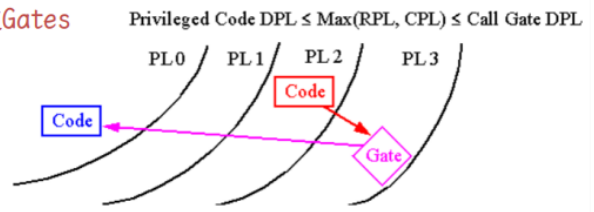
  - Far

- 入栈的是下一条指令的 `seg:offset`

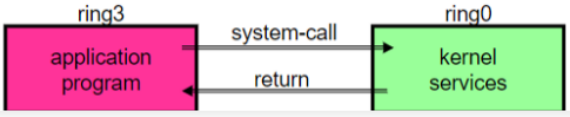
- Inter-privilege-level far call
  - 跨安全级别的跳，哪怕距离近，但只要跨安全级别

- Three ways of making an inter-privilege-level far call: 设置一致性代码段：让低安全级别调用，但仍按低级别的使用
  - defines conforming code segments to share libraries (e.g., math) for various privilege levels
  - through special segment descriptors called Gates

低优先级通过Gates调用PL0



  - utilizes fast system call instructions 系统调用 (SYSCALL/SYSRET or SYSENTER /SYSEXIT) to access ring 0 from ring 3.

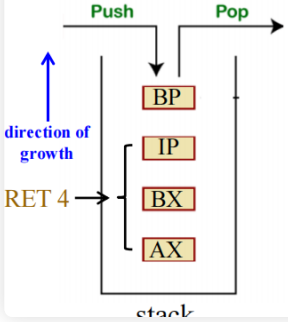


- Task switch

- RET
  - `RET n` 在跳回去之前清理多少个栈

- The example shows how this type of return erases the data placed on the stack by a few pushes. The RET 4 adds a 4 to SP after removing the return address from the stack.

```
        MOV   AX,30
        MOV   BX,40
        PUSH  AX                ;stack parameter 1
        PUSH  BX                ;stack parameter 2
        CALL  ADDM              ;add stack parameters
```




```
ADDM    PROC    NEAR
        PUSH    BP              ;save BP
        MOV     BP,SP           ;address stack with BP
        MOV     AX,[BP+4]       ;get parameter 2
        ADD     AX,[BP+6]       ;add parameter 1
        POP     BP              ;restore BP
        RET     4               ;return, dump parameters
ADDM    ENDP
```

先取出IP
之后再清理四个字节（被调用者清理栈）

Interrupts

- ISP interrupt service procedure
  - IVT 4 bytes
  - IDT 8 bytes

> - The processor uses the vector number (interrupt vector) assigned to an exception or interrupt as an index into the interrupt vector table (IVT) or interrupt descriptor table (IDT).
> - The IVT and IDT provide the entry point to an exception or interrupt handler
>   - IVT is used in real mode
>   - IDT is used in protected mode and long mode

- Interrupt

  中断:外部硬件/内部软件**主动**(voluntary requests by program)发起,大部分是异步,软件发出的是同步的 (using INT)
  - External
    - 分类
      - maskable

        > - External interrupts are classified as maskable or nonmaskable: 可屏蔽中断: INTR, FLAGS.IF=1才会打开 不可屏蔽中断: 比如系统本身的错误
        >   - Maskable interrupts are triggered through the INTR pin by the interrupt-handling mechanism only when FLAGS.IF=1. Otherwise they are held pending for as long as the FLAGS.IF bit is cleared to 0.
        >   - Nonmaskable interrupts (NMI) are unaffected by the value of the FLAGS.IF bit.

        - INTR pin
        - FLAGS.IF
      - non maskable I NMI
    - STI 操纵Interrupt Flags `INTR` , IF

- CLI
- Software
  - `INT n` e.g `INT 80h`
- ## Masking External Interrupts

  - Software can mask the occurrence of certain exceptions and interrupts. Masking can delay or even prevent triggering of the exception-handling or interrupt-handling mechanism.
  - External interrupts are classified as maskable or nonmaskable: 可屏蔽中断：INTR，FLAGS.IF=1才会打开 不可屏蔽中断：比如系统本身的错误
    - Maskable interrupts are triggered through the INTR pin by the interrupt-handling mechanism only when FLAGS.IF=1. Otherwise they are held pending for as long as the FLAGS.IF bit is cleared to 0.
    - Nonmaskable interrupts (NMI) are unaffected by the value of the FLAGS.IF bit.

- Exception
  - 分类
    - Exceptions come from three general sources:
      - Program-Error Exceptions: The processor generates one or more exceptions when it detects program errors during the execution. #DE
      - Software-Generated Exceptions: The INTO, INT1, INT3, and BOUND instructions permit exceptions to be generated in software. E.g., INT3 causes a breakpoint exception to be generated. 中间不带有空格overflow
      - Machine-Check Exceptions: Pentium processors provide both machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions.

- Faults are precise exceptions reported on the boundary before the faulting instruction.
  位置在之前
  - can be corrected and restarted with no loss of continuity.
  - the return address points to the faulting instruction.
- Traps are precise exceptions reported on the boundary following the trapping instruction.
  位置在之后
  - can be continued without loss of program continuity.
  - the return address points to the instruction following the trapping instruction.
- Aborts are imprecise exceptions and do not allow reliable program restart.

助记符 #GP (0) general protection exception with error code 0

- Precise exceptions are reported on an instruction boundary. 产生的位置是准确的，处理之后可以重启
  - Some report the boundary before the instruction causing the exception, while others report the boundary after the instruction causing the exception.
  - When the event handler returns to the interrupted program, it can be restarted at the interrupted-instruction boundary.
- Imprecise exceptions are not guaranteed to be reported on a predictable instruction boundary.
  - Imprecise events can be considered asynchronous.
  - The interrupted program is not restartable.

  异步的，不可重启

- In real mode
  - a 4-byte number stored in the first 1024 bytes of memory (00000H–003FFH). 1K 空间内
  - Each vector contains a value for IP and CS that forms the address of the ISP. 两个字节IP 两个字节CS 先存储IP，后CS
  - the first 2 bytes contain IP; the last 2 bytes CS.
- In protected mode, the vector table is replaced by an interrupt descriptor table (IDT) that uses 8-byte descriptors to describe each of the interrupts.

double fault

- 两个异常的嵌套并且符合下面的组合

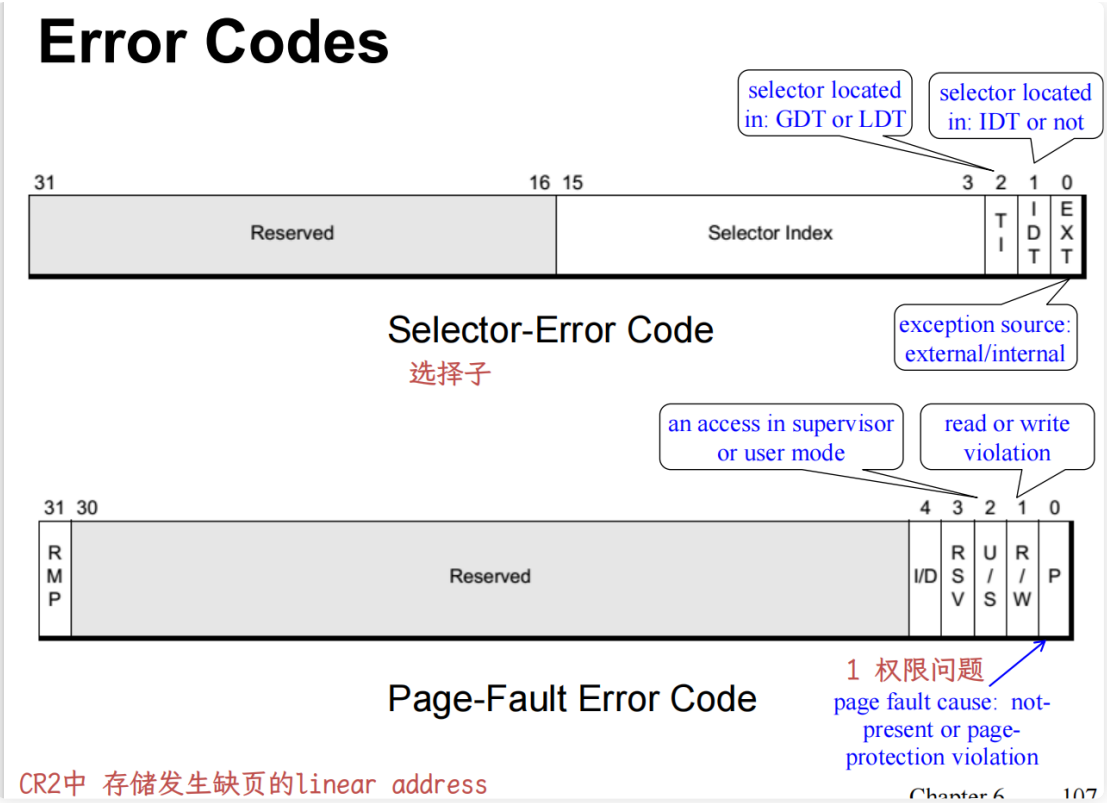
  - Only very specific combinations of exceptions lead to a double fault. These combinations are:

    Double-Fault Exception Conditions

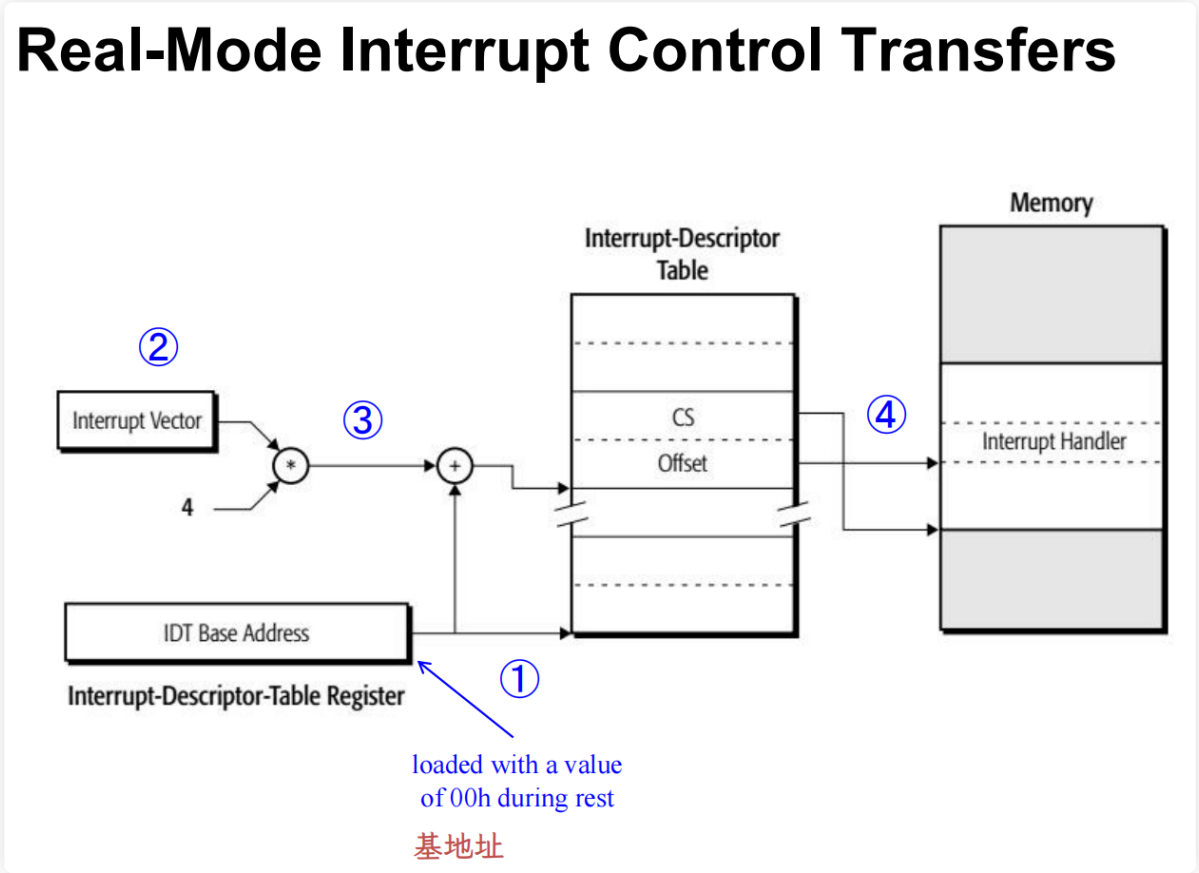
First Interrupting Event	Second Interrupting Event
Contributory Exceptions • Divide-by-Zero-Error Exception • Invalid-TSS Exception • Segment-Not-Present Exception • Stack Exception • General-Protection Exception	Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception
Page Fault Exception	Page Fault Exception Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception

  - For example:
    - divide-by-zero fault → a page fault: No double fault occurs
    - divide-by-zero fault → general-protection fault: A double fault occurs

- *error code*
  - 放在栈上



- **实模式下的处理**
  - IRET is used in real mode and IRETD in the protected mode.

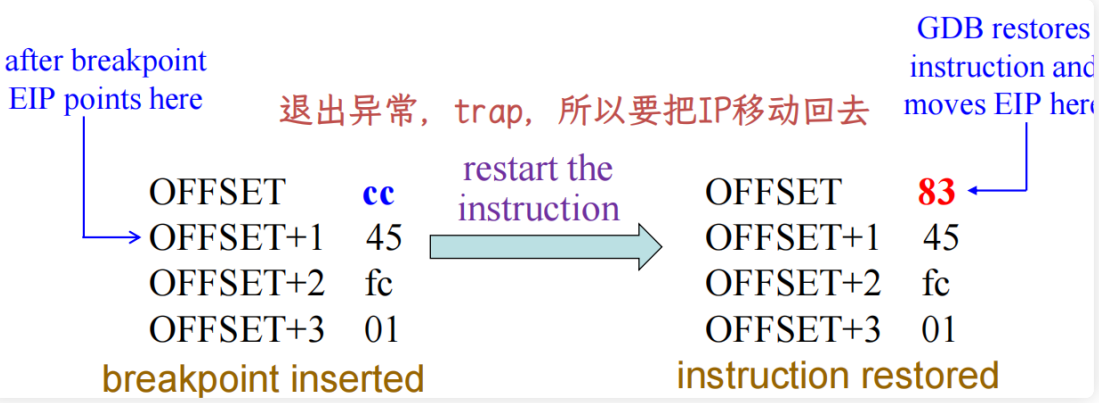


- 软中断的处理
  - INT N 和 far CALL 相近

    - INT N 还会把Flag入栈，使用 IRET
  - INT N 这里的N就是中断向量号，相当于一个index
    - INT 10H 内存位置 40H 的地方放着它的描述信息
  -

- When a software interrupt executes, it:
  - pushes the flags onto the stack
  - clears the IF flag bits
  - pushes CS onto the stack
  - fetches the new value for CS from the interrupt vector
  - pushes IP/EIP onto the stack
  - fetches the new value for IP/EIP from the vector
  - jumps to the new location addressed by CS and IP/EIP

软中断的执行

- INT3 做断点指令
  - INT 3 INT3 作用相同，编码长度不同
    - 方便指令的对齐，防止破坏其他的指令
  - 

- INTO 主动看overflow是否被set
  - 如果set直接去处理

MACHINE CONTROL AND MISCELLANEOUS INSTRUCTIONS

Controlling the Carry Flag Bit

- The carry flag (C) propagates the carry or borrow in multiple-word/doubleword addition and subtraction.
  - can indicate errors in assembly language procedures
- Three instructions control the contents of the carry flag:
  - STC (set carry), CLC (clear carry), and CMC (complement carry)

对Carry的操作

HLT 停机，热等待，之后可以继续执行

- HLT stops instruction execution and places the processor in a HALT state.
- Several ways to exit a halt
  - an enabled interrupt (NMI and SMI)
  - a debug exception
  - a hardware reset (BINIT#, INIT# or RESET# signal) 负电平有效硬件信号
- The saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

- LOCK 前缀
  - 对内存进行修改

- 算数/logic 才能加Lock,MOV就不可以
- 会报 #UD

- BOUND
  - BOUND REG, MEM
  -
    - For example,

```
.data
    start   DW  0, 10      ; define the boundary
.code
...
    MOV     AX, 5
    BOUND   AX, start      ⇒ start[0] < 5 < start[1], executes the next instruction
    MOV     AX, 11
    BOUND   AX, start      ⇒ 5 > start[1], raises a BOUND range exceeded exception
...
```

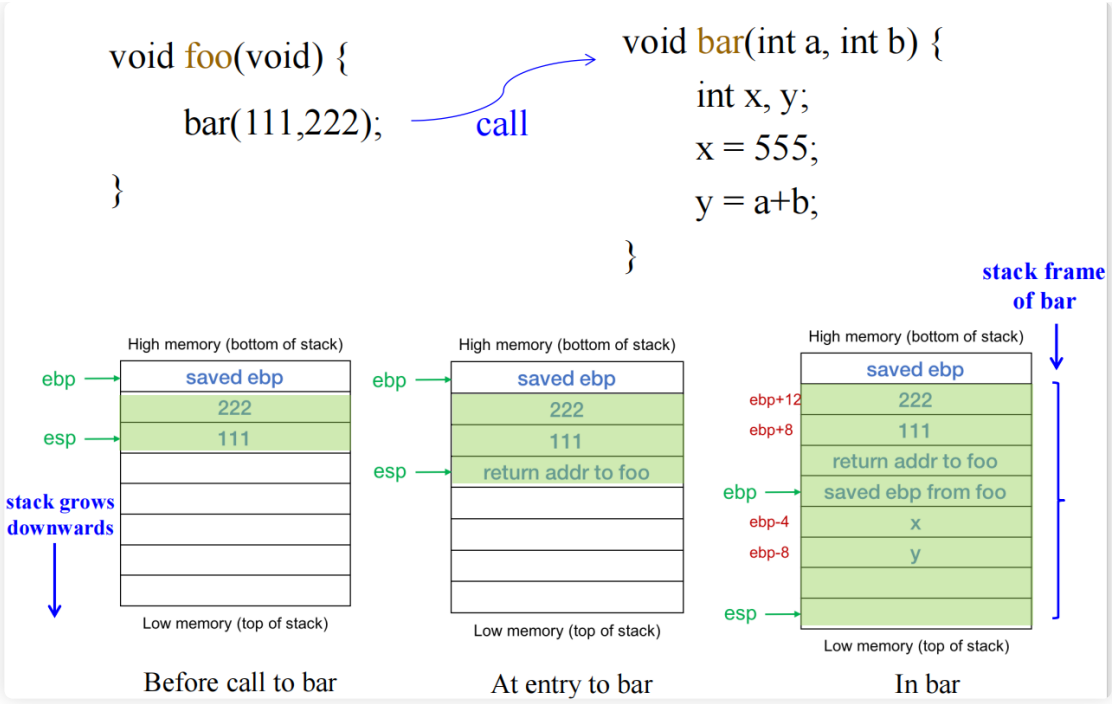
    - BOUND is not valid in 64-bit mode.

!!!重要!!!

- Stack Frame
  - A stack frame is comprised of: 栈帧
    - argument parameters
    - return address
    - previous stack frame pointers
    - local variables
    - saved copies of registers modified by the called procedures (the *callee*) that need restoration

    1.参数、返回地址、之前的EBP（父栈帧）
    2.局部变量
    3.相同的寄存器的备份

- EBP
  - 一个稳定的参考点，整个函数运行过程中不变
  - calling convention
    - 先保存老的EBP,把ESP的值赋给EBP形成新的基准点
    - 退出函数需要恢复EBP
  - EBP + n 访问父亲的para（其实是父亲给的参数）,EBP - n 访问 local var
    - 这里可以看出来EBP的值是在function开始运行的时候才赋予 的

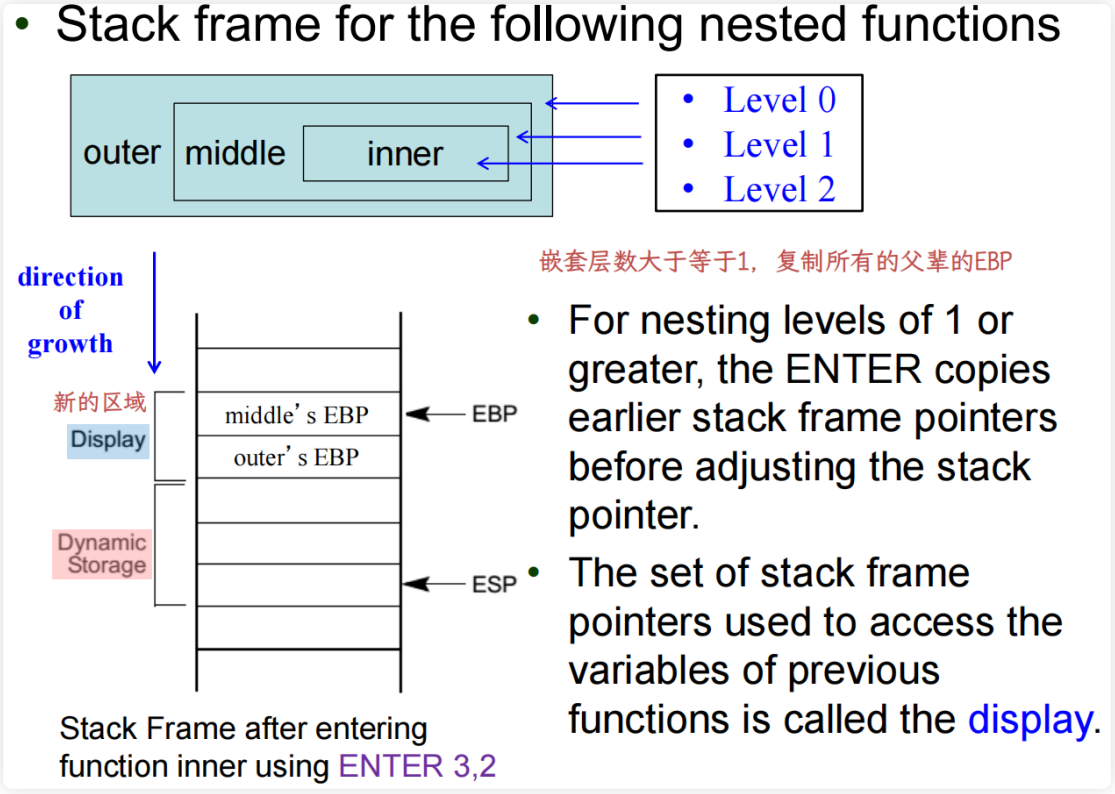


- Enter / LEAVE P148

- ENTER and LEAVE instructions create and release stack frames for called procedures.
- Syntax: ENTER stack space, nesting levels (预留局部变量的空间，嵌套（是第几层）)
  - The first operand specifies the size of the dynamic storage in the stack frame.
  - The second operand specifies the nesting level (0 to 31—the value is automatically masked to 5 bits).
- For example:

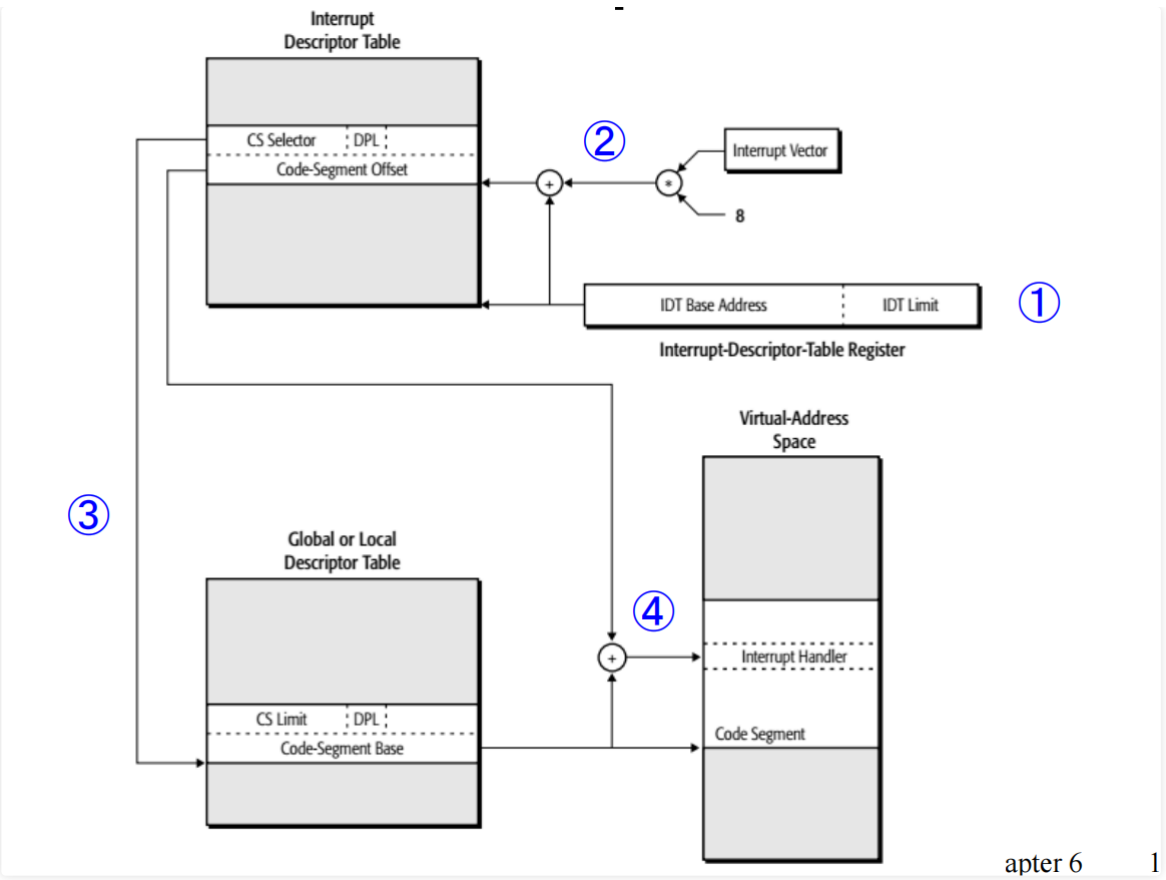
```
BAR  PROC                          SUM  BAR
push ebp       ; Save old frame pointer
mov ebp, esp   ; Point ebp to top-of-stack   ⟶  enter  8, 0
sub  esp,  8   ; Reserve 8 bytes of locals       ...
...                                              ...
mov esp, ebp   ; Restore ebp and remove          ...
               ; stack space for locals.     ⟶  leave
pop ebp                                          ret
ret
BAR  ENDP                          BAR  ENDP
```

- 注意这里EBP指向的位置

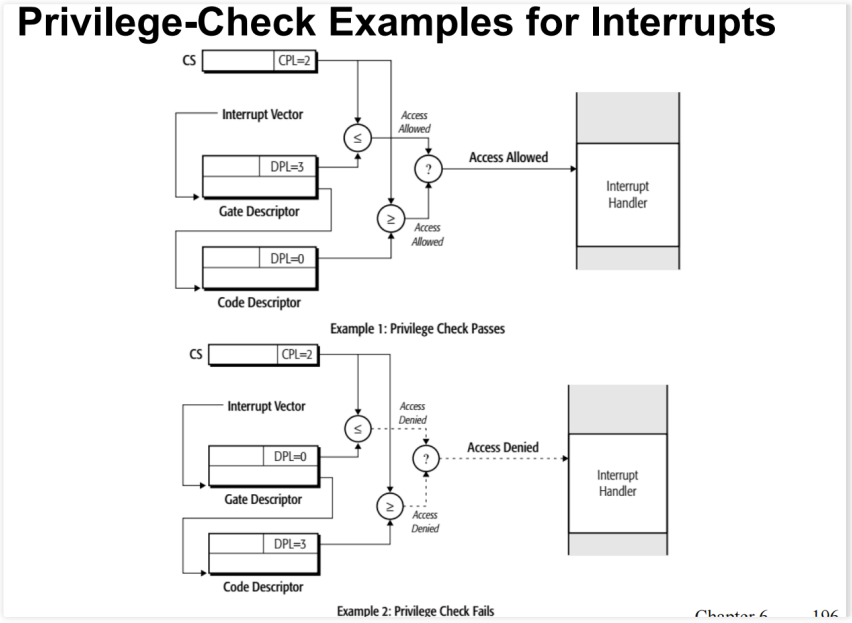


保护模式下的梳理

> “
>
> 这一块课上似乎没讲？（
>
> 在第六章最后的PPT有



apter 6 1



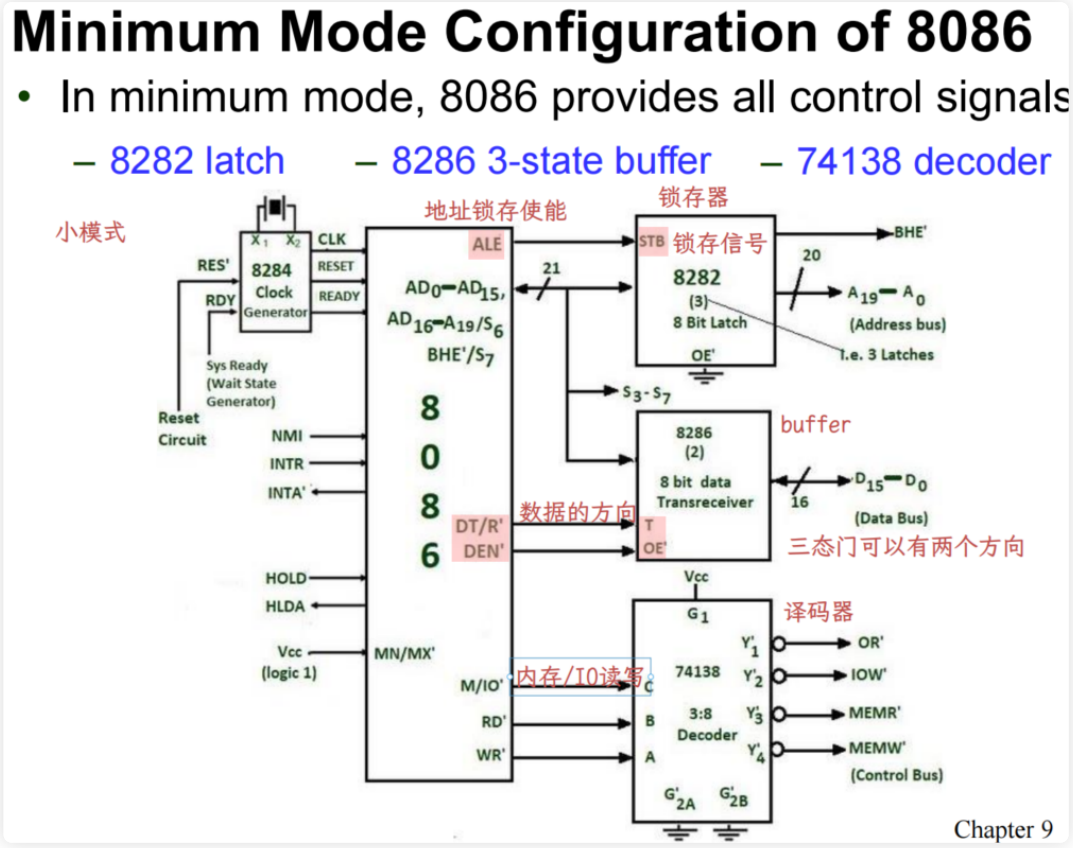
Ch 9 8086/8088 Hardware

- 8086 16 个 AD

The 8086 can operate in two modes:

- Minimum Mode 处理器只有一个8086，对外直接输出各种信号
- Maximum Mode 还有协处理器，不能直接输出控制信号，只能输出一些中间信号，需要总线控制器仲裁/...

- Minimum mode
  - **ALE：** 地址锁存使能
    - 0 containing data
  - **BLE** 访问总线上的高八位
  - 单独一个处理器，直接输出信号



- Maximum mode
  -

The Pin-Out

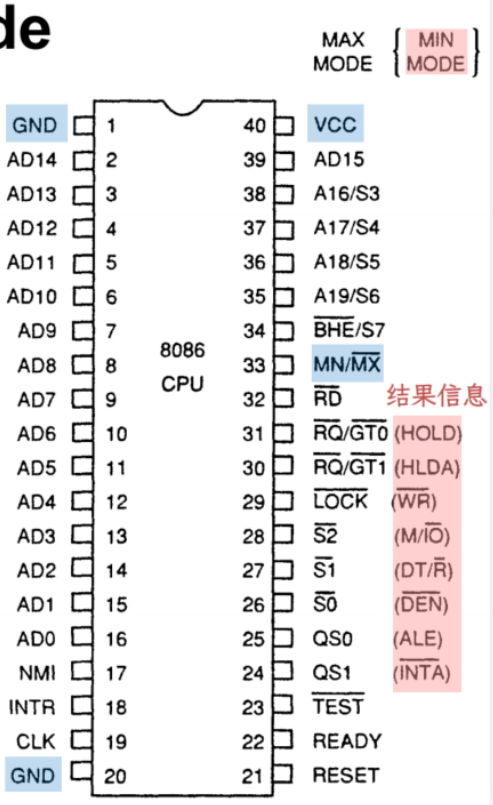
- Figure 9–1 illustrates pin-outs of 8086 & 8088.
  – both are packaged in 40-pin **dual in-line** packages (DIPs)

- 8086 is a 16-bit microprocessor with a 16-bit data bus; 8088 has an 8-bit data bus.
  – 8086 has pin connections AD_0–AD_{15}
  – 8088 has pin connections AD_0–AD_7

The Pin-out of the 8086 in Maximum Mode and Minimum Mode

- VCC (+5 V Power Supply)
- GND (Ground)
- MN / $\overline{MX}$ 33号引脚 (Minimum/Maximum):
  High: 小模式 Low: 大模式
  - indicates what mode the processor is to operate in.
  - Minimum mode: HIGH
  - Maximum mode: LOW

上下两个接地：数字和模拟信号分开
减少干扰



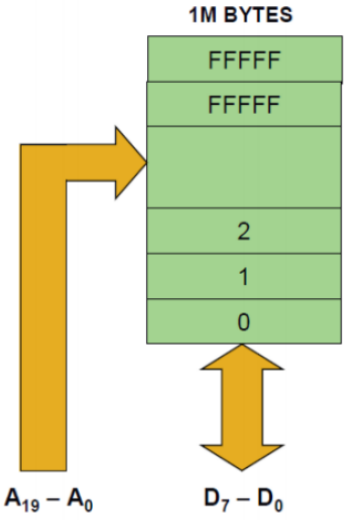
内存的访问

Memory Bank

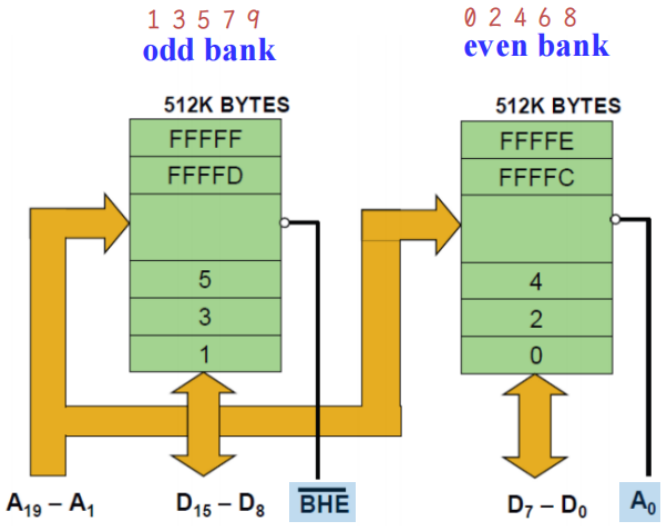
一个字节的访问，非对齐字节的访问

- X86 uses memory banks to support one byte transfer or unaligned memory accesses.
- A "bank" refers to 8-bit wide memory. For example: Bank的组织都是一个字节宽度
  - The 8088 has an 8-bit data bus and the memory address space is implemented as single 1 Mbyte memory bank.
  - While 8086 has a 16-bit data bus and the memory address space implemented as two independent 512 Kbyte banks. 两个512Kbyte的banks

Single Memory Bank vs. Dual Memory Bank



1Mx8 memory bank of the 8088

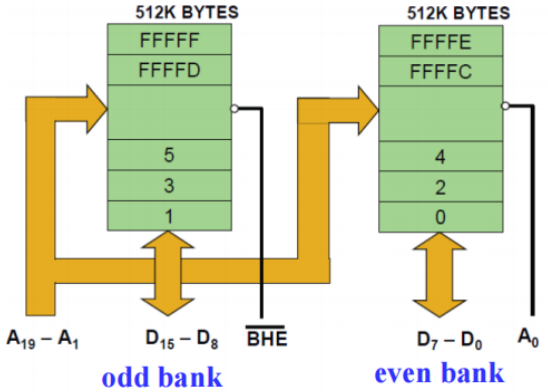


High and low memory banks of the 8086

访问总线低/高/两个
同时使能
Even A0=0

- Bank high enable ($\overline{BHE}$) and bank low enable ($\overline{BLE}$/A0) are used as bank-select signals:
  - $\overline{BHE}$ = 0 enables the high/odd bank.
  - $\overline{BLE}$/A0 = 0 enables the low/even bank.
- Address bits A1-A19 select the location.

都是低电平有效
效果为使能对应的8bit



High and low memory banks of the 8086

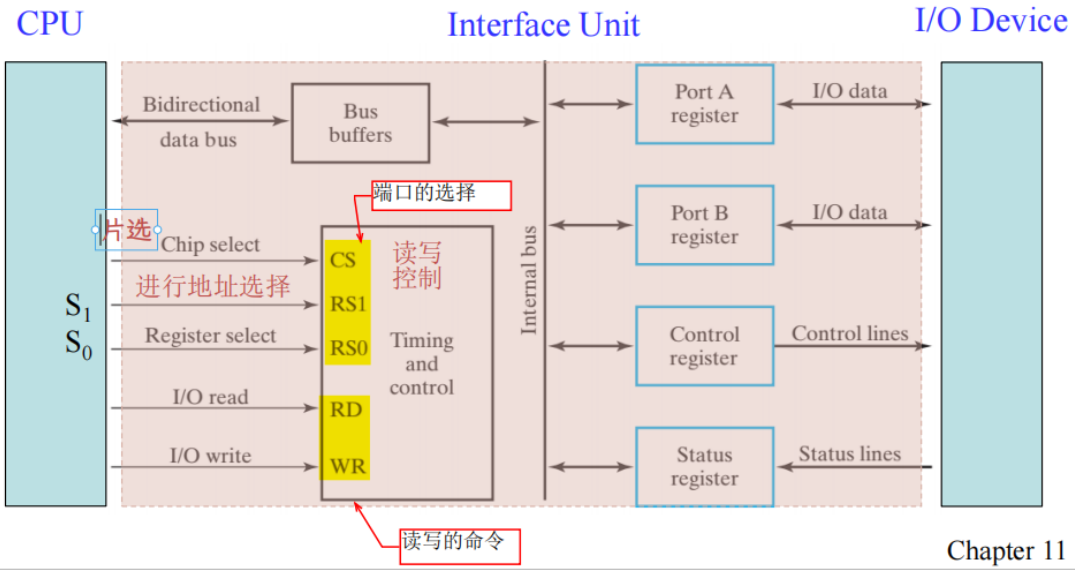
$\overline{BHE}$	$\overline{BLE}$	*Function*
0	0	Both banks enabled for a 16-bit transfer
0	1	High bank enabled for an 8-bit transfer
1	0	Low bank enabled for an 8-bit transfer
1	1	No bank enabled

Memory bank selection using BHE and BLE (A_0)

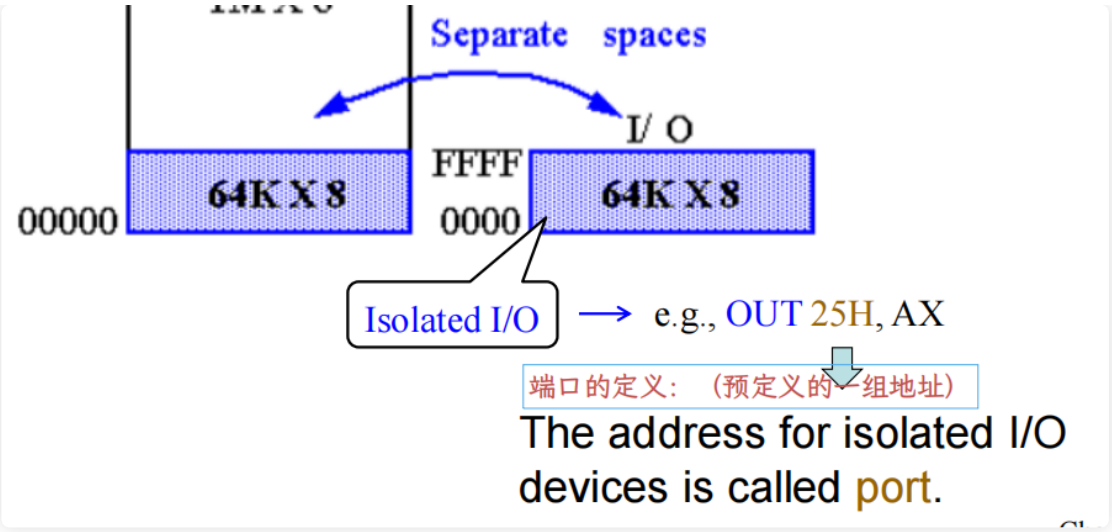
Basic Interface

Structure of Hardware Interface

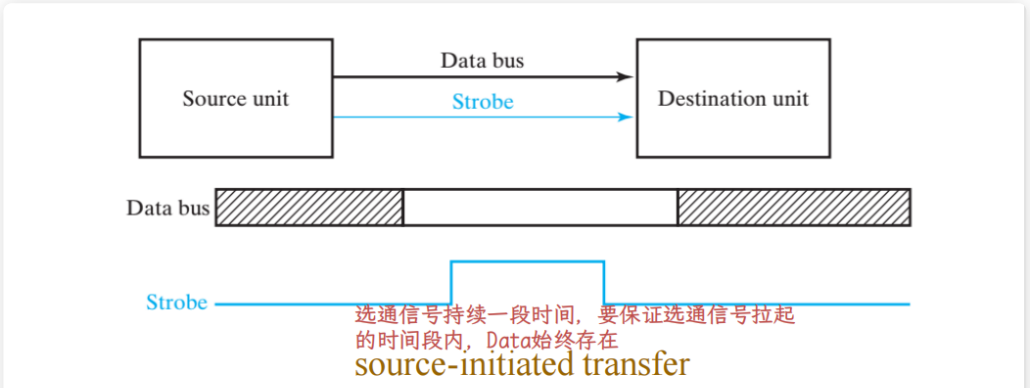
- An I/O interface unit contains the following blocks:
  - Read/Write Control Logic
  - Port register (e.g. port A, port B)
  - Data Bus Buffer
  - Control and Status register



- 连接
  - output interface - `latch`
  - input interface - 输入设备通过 `three state buffer` 接到总线上
- IO地址编码
  - Isolated IO
    - 和内存地址不同，独立编码：使用独立的指令 `IN OUT`
  - MM
    - 不需要额外的指令，速度更快
      - 更慢的反应速度
      - 限制了内存地址的大小
- 系统预定义端口



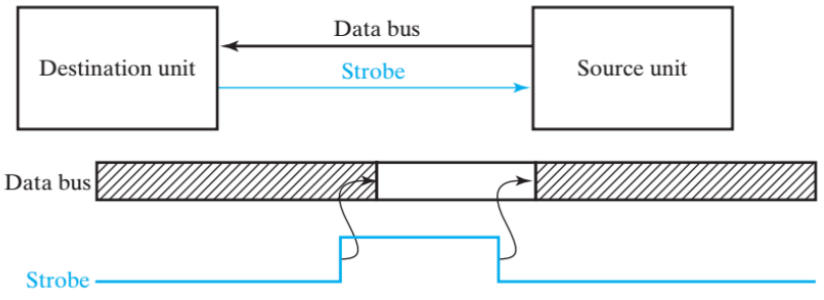
- 同步的解决（本身存在状态的设备，异步设备：如何做同步）
  - strobing 单项选通
    - 分类
      - 数据源



source-initiated transfer

- In source-initiated transfer:
  - the source first places the data on the data bus and then changes strobe from 0 to 1;
  - the destination sets up the transfer to a register;
  - the source then changes strobe from 1 to 0;
  - the source removes the data from the data bus.

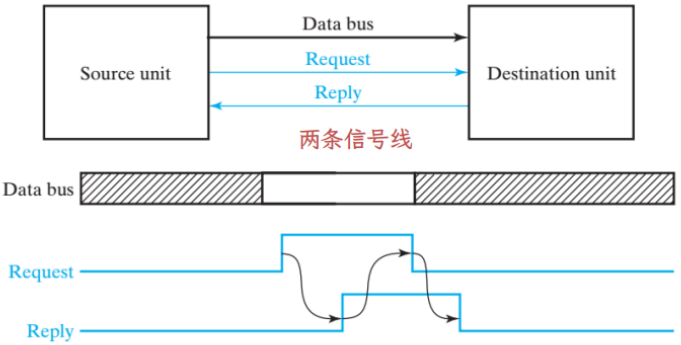
- 目的



destination-initiated transfer

- In destination-initiated transfer:
  - the destination changes strobe from 0 to 1;
  - the source places the data on the data bus;
  - destination captures the data in a register and changes strobe from 1 to 0;
  - the source removes the data from the data bus.

- handshaking polling 双向
  - 需要两条信号线
  -



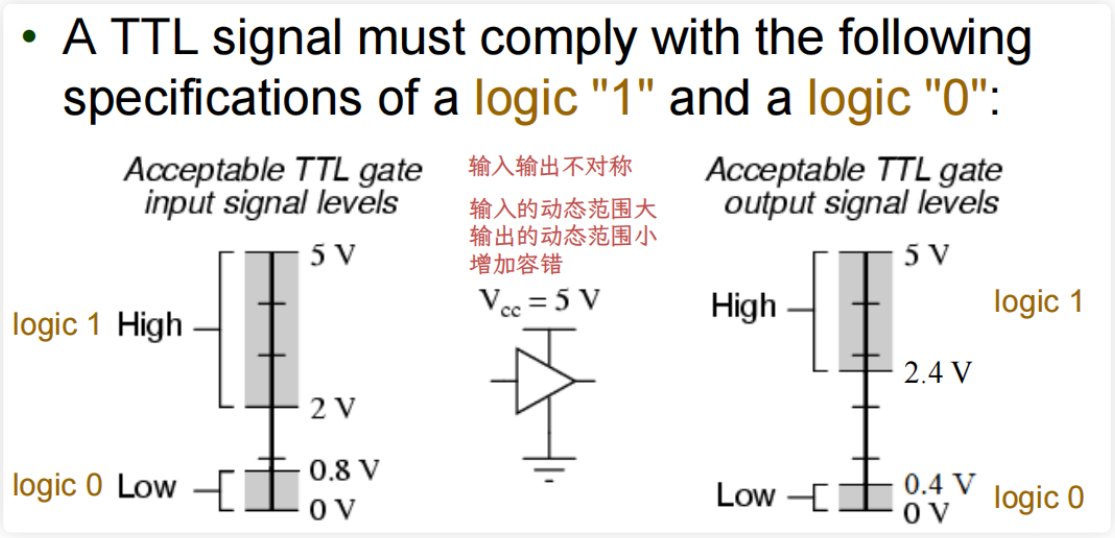
source-initiated transfer

- In source-initiated transfer:
  - source first places the data on the data bus and then enables request;
  - destination sets up the transfer and then activates reply;
  - source removes the data and resets request;
  - destination resets reply.

- polling
  -

- 物理器件

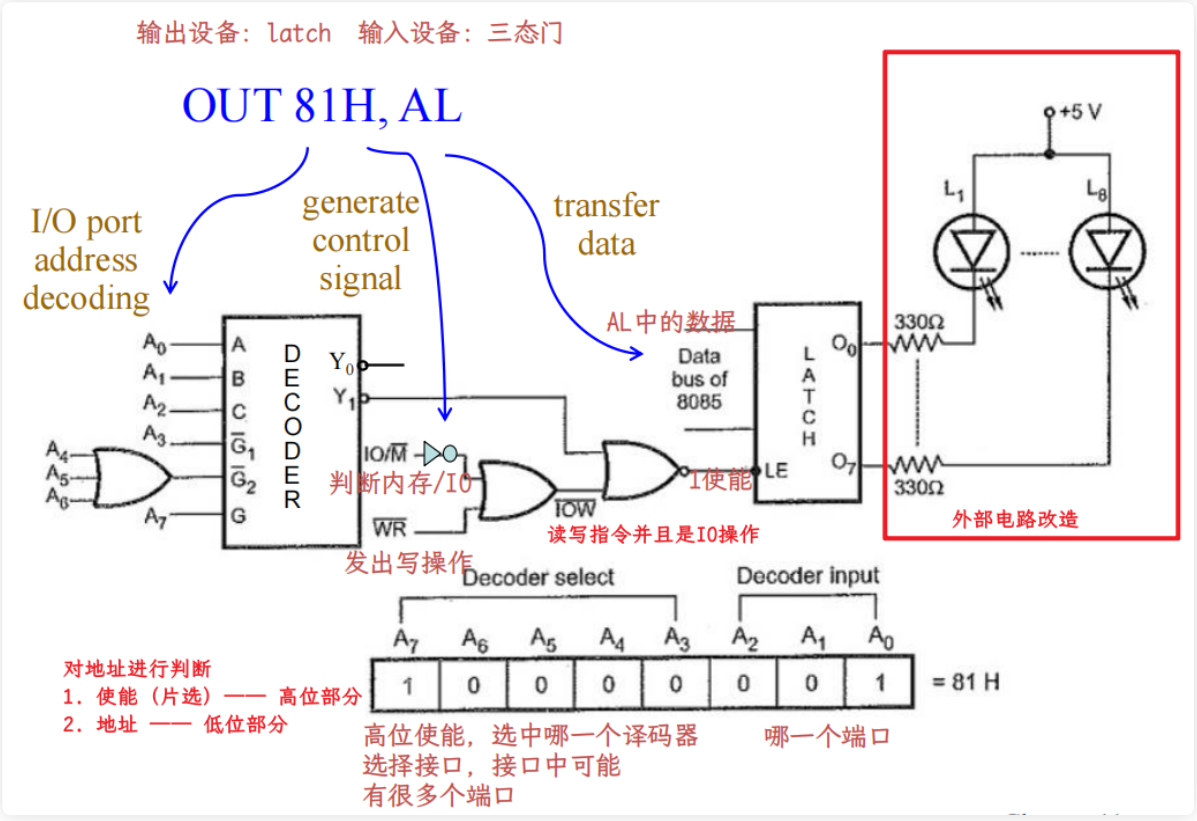
- TTL



- CMOS 比例定义动态范围

IO PORT ADDRESS DECODING

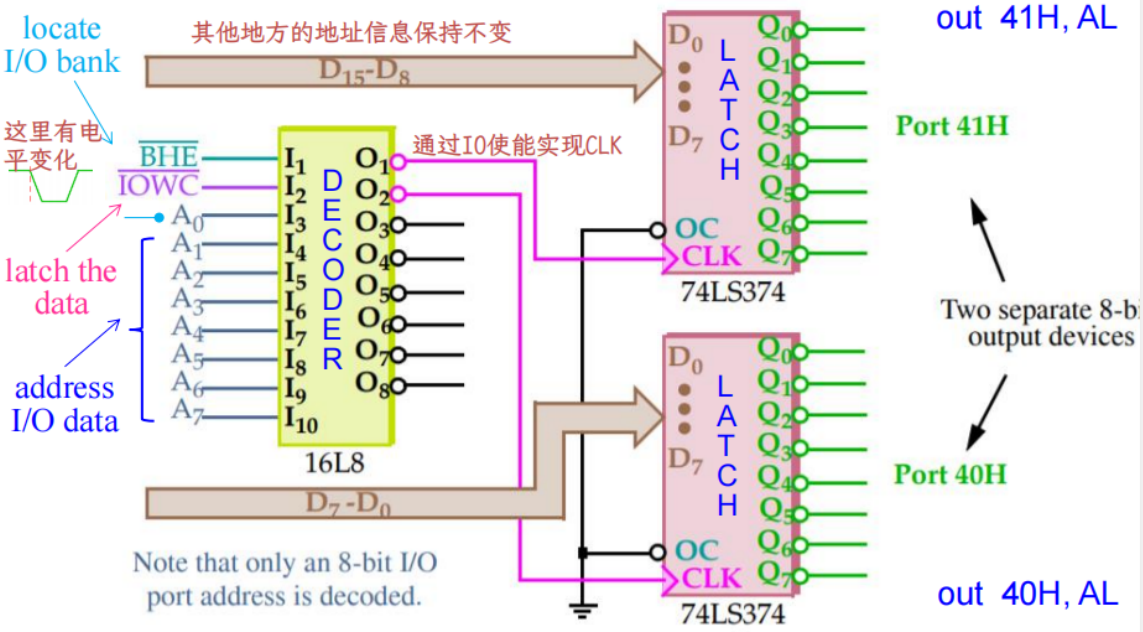
> 端口的数据和地址宽度是两件事



单字节输出

- 额外引入 BHE ， 对内存地址进行访问
-

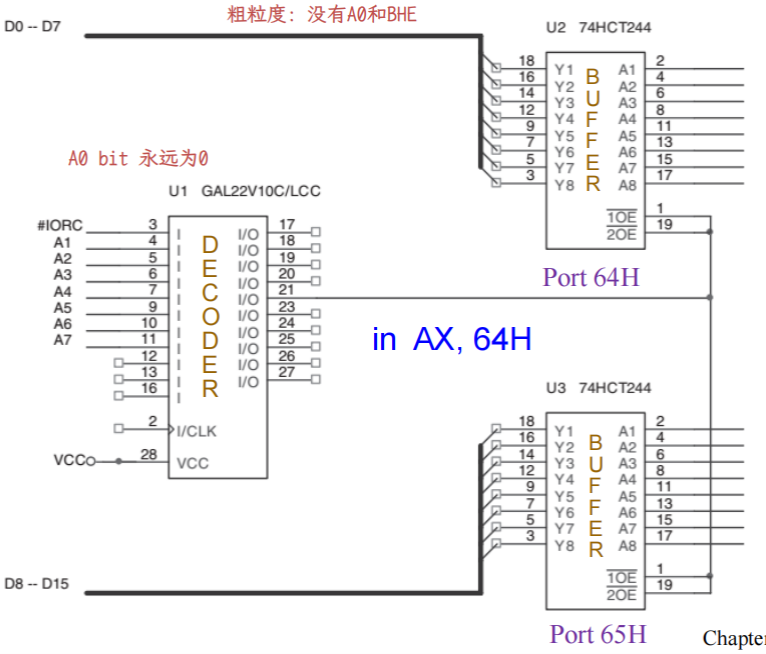
Figure 11–14 An I/O port decoder that selects ports 40H and 41H as separate 8-bit ports for output data.



十六位输出

-

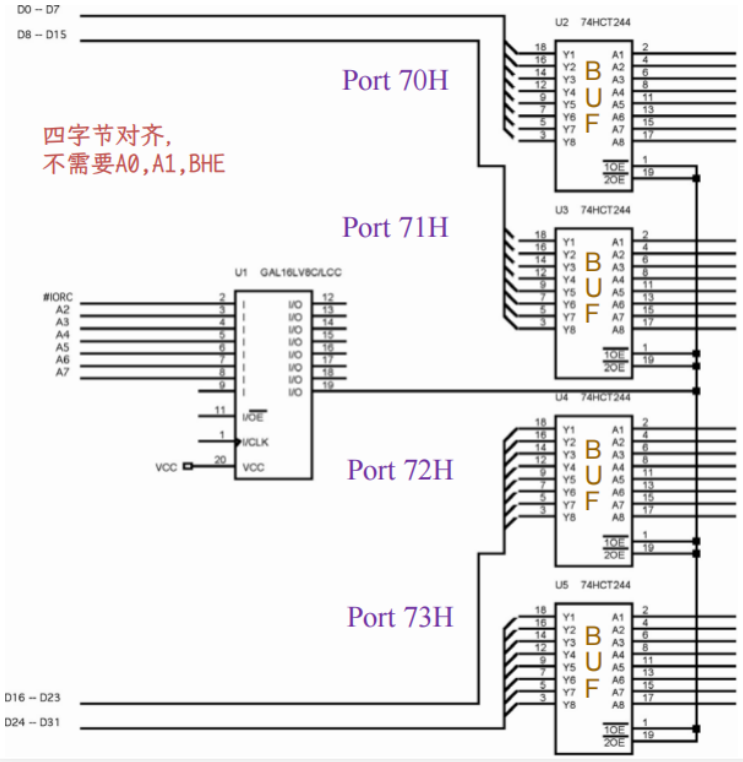
Figure 11–15 A 16-bit-wide port (2-byte aligned) decoded at I/O addresses 64H and 65H. 16-bit宽度的数据，按两字节对齐 粗粒度：没有A0和BHE



Chapter 11

三十二位

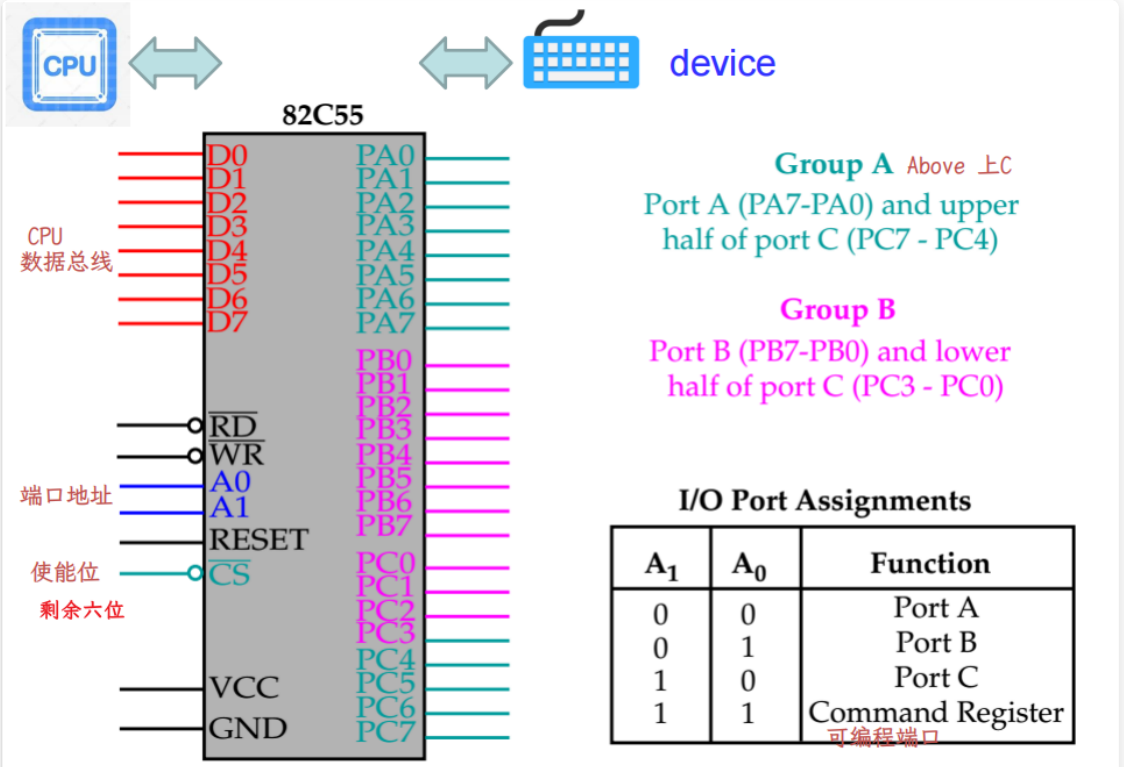
Figure 11–16 A 32-bit-wide port (4-byte aligned) decoded at 70H through 73H for the 80486DX microprocessor.



- I/O ports decoded by this interface are the 8-bit ports 70H–73H
- When writing to access this port, it is crucial to use the address 70H for 32-bit input
- as instruction in EAX, 70H

82C55 | 可编程并行接口

> “
>
> 端口和数据都是八位



- 使用A1,A2 选择端口，可变端口在中间部分

- 控制组
  - 端口A和B作数据，C口是用作握手的控制信号
    - C口的reading可以一次读取整个字节，但设置必须按照bit
      - 设置C口方向的前提是他没有被占用

- The three I/O ports (labeled A, B, and C) are programmed as groups.
  - A和上C group A connections consist of port A (PA_7–PA_0) and the upper half of port C (PC_7–PC_4)
  - B和下C group B consists of port B (PB_7–PB_0) and the lower half of port C (PC_3–PC_0)

C端口用于做握手时的控制信号

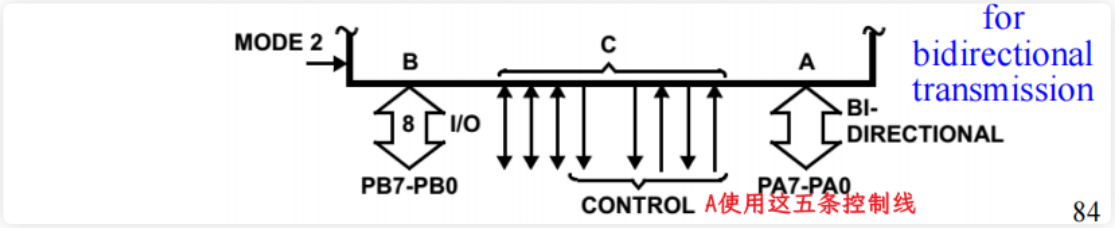
- 操作

82C55 Basic Operation

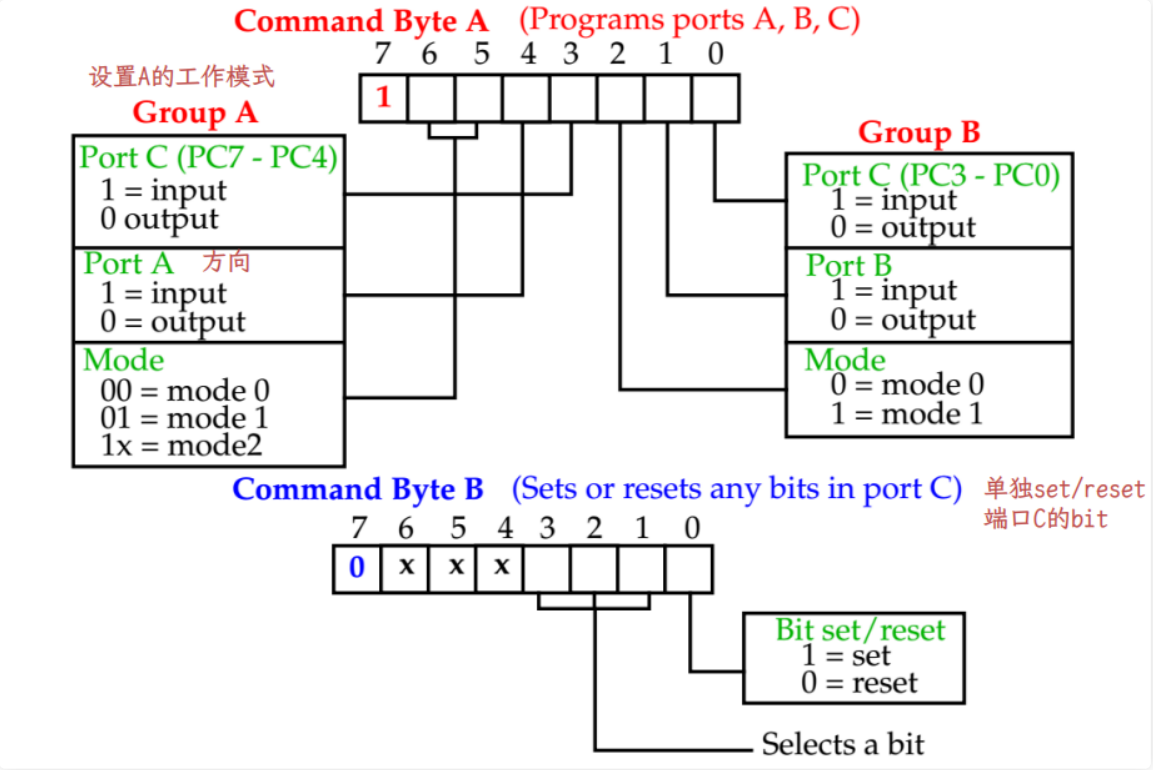
A1	A0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	INPUT OPERATION (READ)
0	0	0	1	0	Port A → Data Bus 从外部设备到CPU
0	1	0	1	0	Port B → Data Bus
1	0	0	1	0	Port C → Data Bus
1	1	0	1	0	Control Word → Data Bus
					OUTPUT OPERATION (WRITE)
0	0	1	0	0	Data Bus → Port A
0	1	1	0	0	Data Bus → Port B
1	0	1	0	0	Data Bus → Port C
1	1	1	0	0	Data Bus → Control 设置工作模式

for Mode 0, output only

- Port
  - A 输入输出都有一组单独的latch和buffer
  - B 共享
  - C 共享，按bit进行操作（其余两个按字节操作）
- **Mode**
  - 0：基本输入输出（无选通信号）只能选择单项，只有数据交互，和always-on设备
  - 1：选通握手(A & B) 外部设备可能有时序
  - 2：双向的bus(A only)



- **命令字**



  - A（同时设置 A B）
  - B（对端口C进行按位设置）

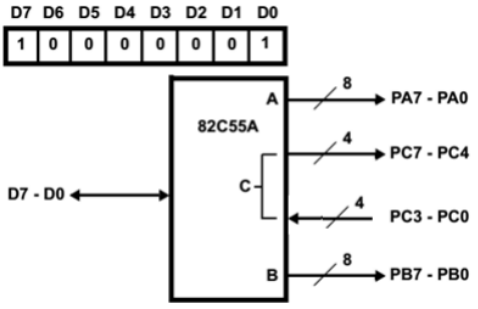
		Mode 0		Mode 1		Mode 2
Port A		IN	OUT	IN	OUT	I/O
Port B		IN	OUT	IN	OUT	Not used
Port C	0	IN	OUT	$INTR_B$	$INTR_B$	I/O
	1			IBF_B	$\overline{OBF}_B$	I/O
	2			$\overline{STB}_B$	$\overline{ACK}_B$	I/O
	3			$INTR_A$	$INTR_A$	INTR
	4			$\overline{STB}_A$	I/O	$\overline{STB}$
	5			IBF_A	I/O	IBF
	6			I/O	$\overline{ACK}_A$	$\overline{ACK}$
	7			I/O	$\overline{OBF}_A$	$\overline{OBF}$

mode 0

- Programming
  - A configuration example:
    - Group A and Group B in mode 0
    - port A and B as outputs
    - port C upper (PC7-PC4) as output
    - port C lower (PC3-PC0) as input

mode + 该port的方向 + 对应C口的方向

```
; programming the 82C55

COMMAND_ADDRESS EQU 703H

MOV AL, 10000001B
MOV DX, COMMAND_ADDRESS
OUT  DX, AL
```



- 写A类/B类指令字
- 比如一个设备占据了 700H - 703H 四个端口，那么我们根据端口选择信号就可以判断使用哪一个端口

```
DISP   PROC   NEAR USES AX BX DX SI

       PUSHF
       MOV    BX,8                ;load counter
       MOV    AH,7FH              ;load selection pattern
       MOV    SI,OFFSET MEM-1     ;address display data
       MOV    DX,701H             ;address Port B

;display all 8 digits

       .REPEAT
              MOV    AL,AH        ;send selection pattern to Port B
              OUT    DX,AL
              DEC    DX           ← address Port A
              MOV    AL,[BX+SI]   ;send data to Port A
              OUT    DX,AL
              CALL   DELAY        ;wait 1.0 ms
              ROR    AH,1         ;adjust selection pattern
              INC    DX           ← address Port B
              DEC    BX           ;decrement counter
       .UNTIL BX == 0

       POPF
       RET

DISP   ENDP
```

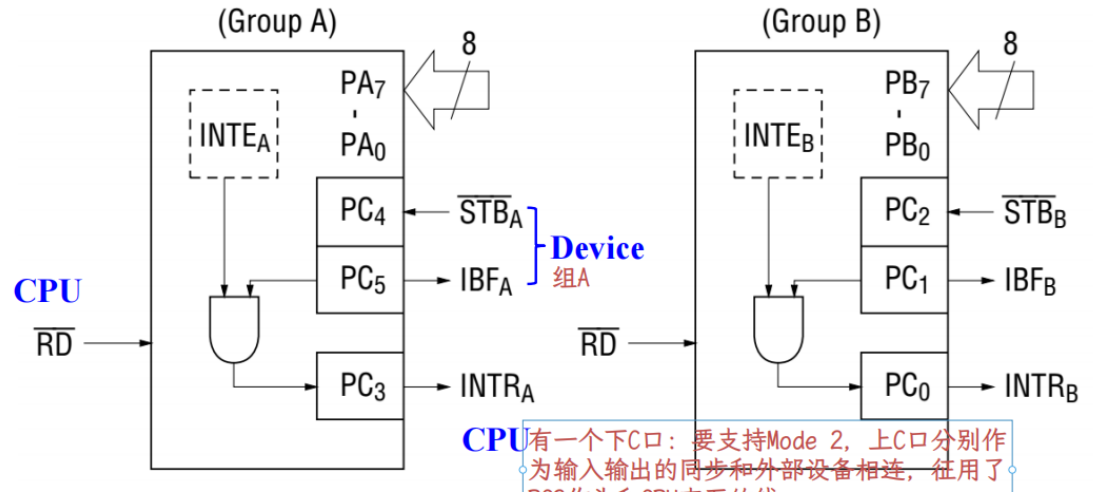
0111 1111 B
7th LED is selected

bx计数，选择哪一个管

mode 1

Figure 11–27 Strobed input operation (mode 1) of the 82C55. (Internal structure)

Input 情况：外部设备 -> CPU

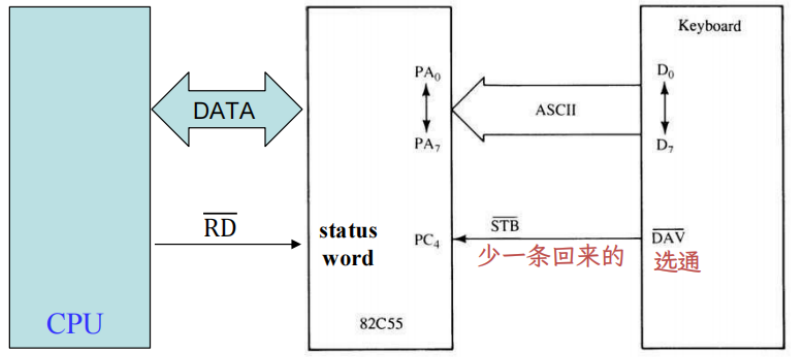


Note: Although belonging to group B, PC_3 operates as the control signal of group A functionally.

IN: strob

device-initiated transfer

- example




```
;A procedure that reads the keyboard encoder and
;returns the ASCII key code in AL
使用轮询的方式
BIT5   EQU   20H
PORTC  EQU   22H
PORTA  EQU   20H

READ   PROC  NEAR
       .REPEAT                    ;poll IBF bit
              IN    AL,PORTC
              TEST  AL,BIT5
       .UNTIL !ZERO?              ; until input buffer is full
       IN    AL.PORTA             ;get ASCII data
       RET

READ   ENDP
```

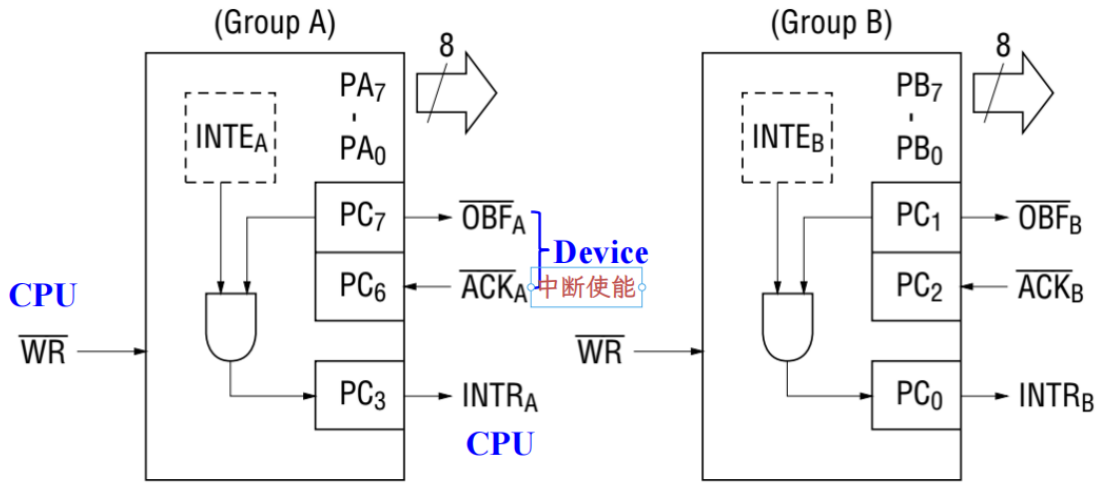
INPUT CONFIGURATION

D7	D6	D5	D4	D3	D2	D1	D0
I/O	I/O	IBFA	INTEA	INTRA	INTEB	IBFB	INTRB
			GROUP A		GROUP B		

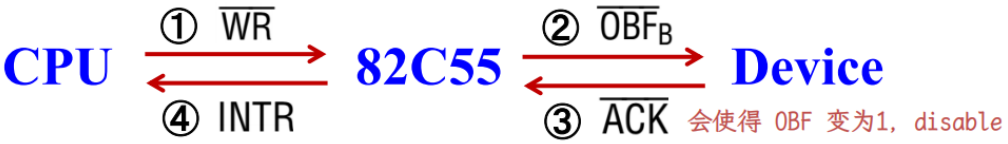
output

Figure 11–29 Strobed output operation (mode 1) of the 82C55 (Internal structure)

output情况



output buffer full



CPU-initiated transfer

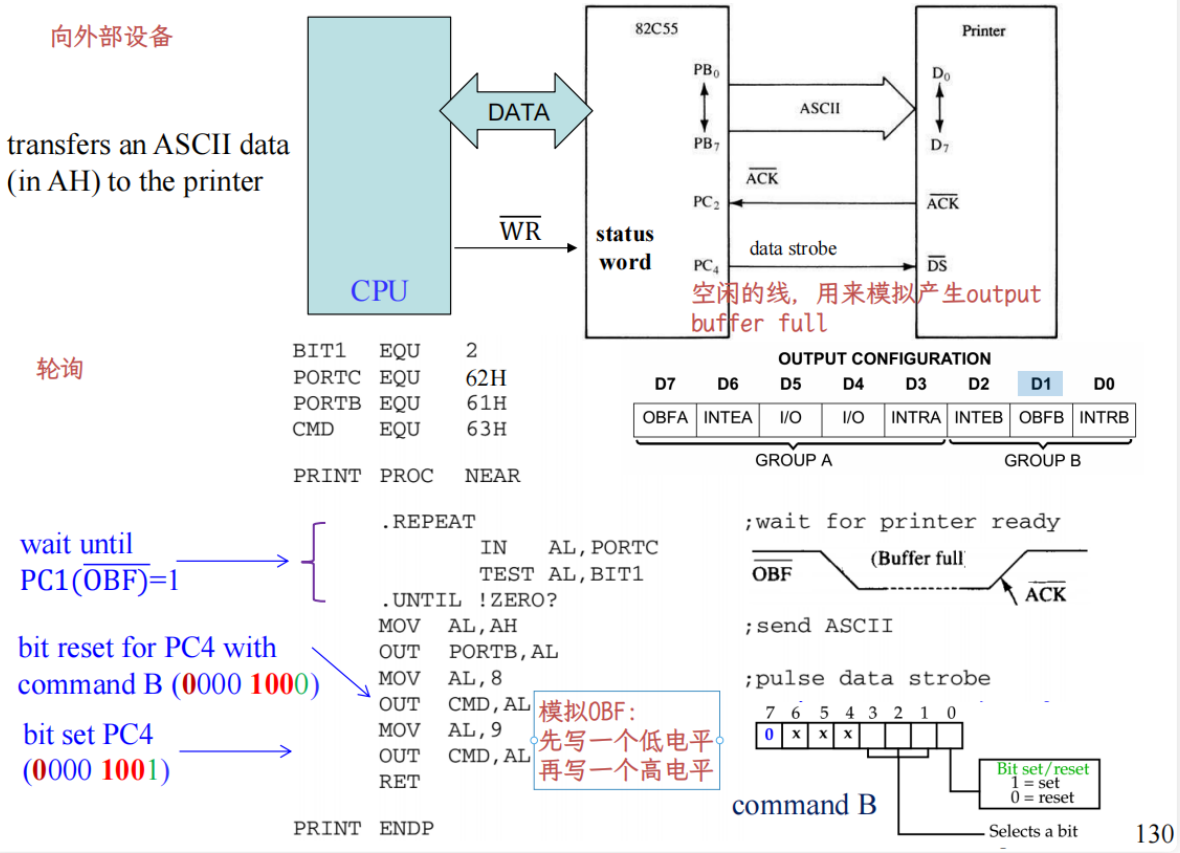
INTR

- Interrupt request interrupts the processor when the external device receives the data via the $\overline{ACK}$ signal.
  - INTR is set by the condition: INTE is a "1", $\overline{ACK}$ is a "1" and $\overline{OBF}$ is a "1".
  - Cleared when data are input from the port by the processor.

INTE

- Interrupt enable is neither input nor output; it is an internal bit programmed via PC_6 (port A) or PC_2 (port B).

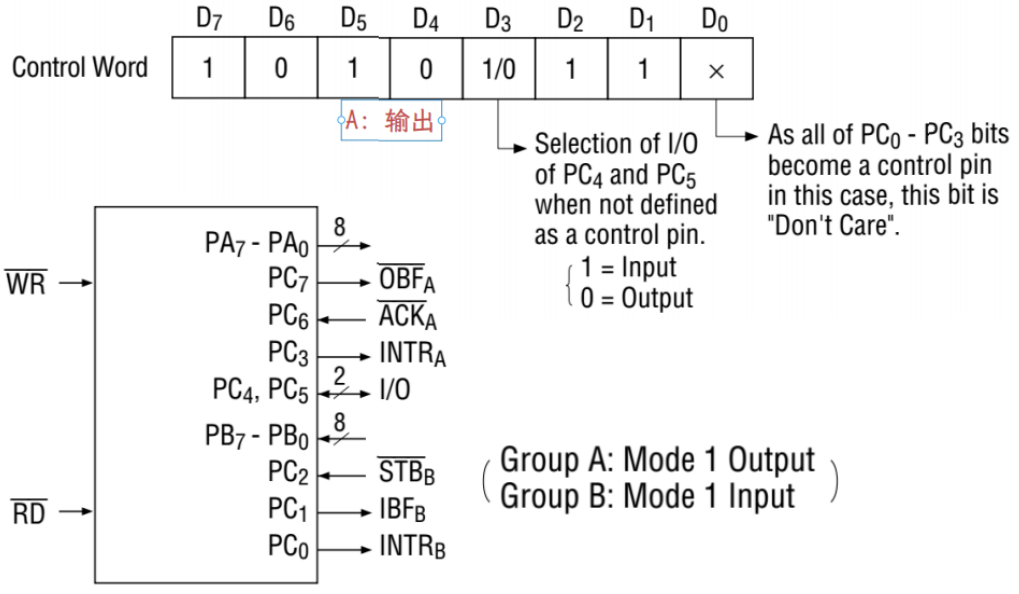
Figure 11–30 Using the 82C55 for strobed output operation to a parallel printer



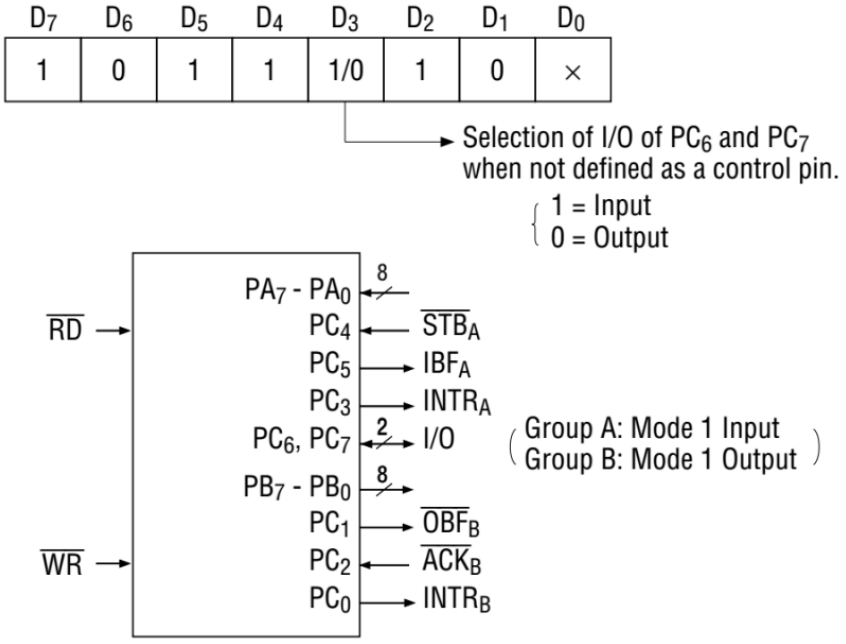
combination

Combinations of Mode 1

- Port A and Port B can be individually defined as input or output to support strobed I/O applications.



Combinations of Mode 1



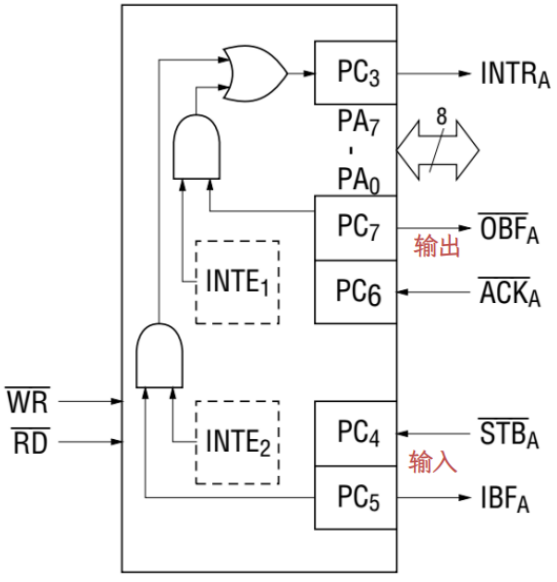
When group A is mode 1 input and group B is mode 1 output

mode 2

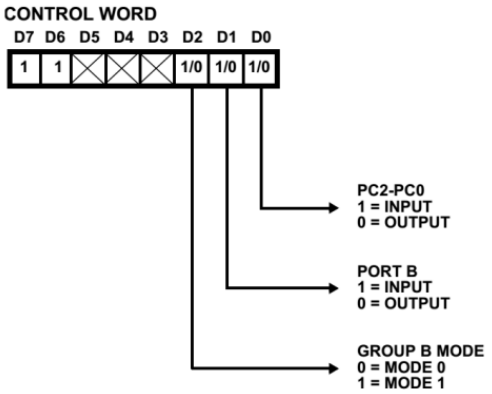
Figure 11–31 Mode 2 operation of the 82C55. (Internal structure)

最极限的情况就是A工作在mode2，B工作在mode1

宏观并行微观串行



Port C	Function
PC_0	Confirmed to the Group B Mode
PC_1	
PC_2	
PC_3	$INTR_A$
PC_4	$\overline{STBA}$
PC_5	IBF_A
PC_6	$\overline{ACK_A}$
PC_7	$\overline{OBF_A}$



Reading Port C Status

通过C口的状态，OBF 写数据，IBF 读数据；同时有效，同时做

- When port C is used for the control signal in mode 2, each control signal and bus status signal can be read out by reading the content of port C.
- Reading the contents of port C allows the program to test or verify the "status" of each peripheral device and change the program flow accordingly.

INTR 的竞争

D7	D6	D5	D4	D3	D2	D1	D0
OBFA	INTE1	IBFA	INTE2	INTRA	X	X	X
GROUP A					GROUP B		

Mode 2 status word format

- As soon as the output circuitry sees a logic 0 on $\overline{OBF}$, it sends back the $\overline{ACK}$ signal to remove it from the output buffer. The $\overline{ACK}$ signal sets the $\overline{OBF}$ bit and enables the three-state output buffers so that data may be read.
- Example 11–20 lists a procedure that transmits the contents of the AH register through port A.

```
BIT7   EQU  80H
PORTC  EQU  62H
PORTA  EQU  60H

TRANS  PROC NEAR

       .REPEAT                    ;test OBF
            IN   AL,PORTC
            TEST AL,BIT7
       .UNTIL !ZERO?
       MOV  AL,AH                 ;send data
       OUT  PORTA,AL
       RET

TRANS  ENDP
```

wait until PC7($\overline{OBF}$) =1 →

D7	D6	D5	D4	D3	D2	D1	D0
OBFA	INTE1	IBFA	INTE2	INTRA	X	X	X
GROUP A					GROUP B		

- When the IN executes, the IBF bit is cleared and data in the port are moved into AL.
- See Example 11–21 for a procedure that reads data from the port A.

EXAMPLE 11–21

```
;A procedure that reads data from the bidirectional bus into AL

BIT5   EQU  20H
PORTC  EQU  62H
PORTA  EQU  60H

READ   PROC NEAR

       .REPEAT                    ;test IBF
             IN    AL,PORTC
             TEST  AL,BIT5   <----- wait until PC5(IBF)=1
       .UNTIL !ZERO?
       IN  AL,PORTA
       RET

READ   ENDP
```

D7	D6	D5	D4	D3	D2	D1	D0
OBFA	INTE1	IBFA	INTE2	INTRA	X	X	X
GROUP A					GROUP B		

8254

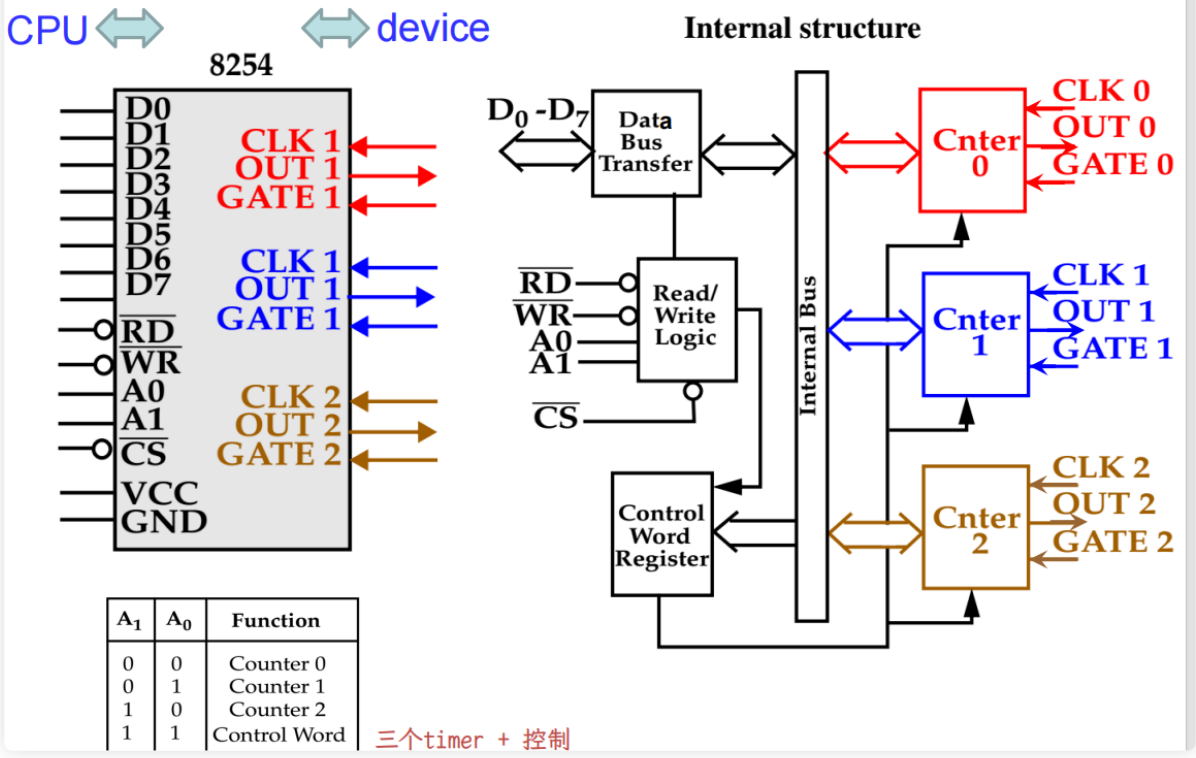
> 电平触发：在CLK上升沿被采样，进而出发计数
>
> 边沿触发：在任意时刻，GATE一旦产生上升沿，就会被内部的 flip-flop 采样保存，维持该信号，直到被下一个 CLK 上升沿采样到，然后下降沿触发计数

- 发控制信号
- 作为定时器，内部含有三个timer
  - 8259A 产生中断
  - 动态内存 DRAM 的flash
  - 和外部设备相关

- 每一个timer的组成
  - Each timer contains:
    - a CLK input which provides the basic operating frequency to the timer
    - a gate input pin which controls the timer in some modes 门控信号：让中间停止
    - an output (OUT) connection to obtain the output of the timer

- The Intel 8254 is a superset of the 8253. The basic function is as follows: 六种工作方式
  - Three independent 16-bit counters 十六位计数器
  - Binary or BCD counting
  - Six programmable counter modes
  - Counter latch command 计数状态
  - Multiple latch command for easy monitoring
  - Handles inputs from DC to 10 MHz

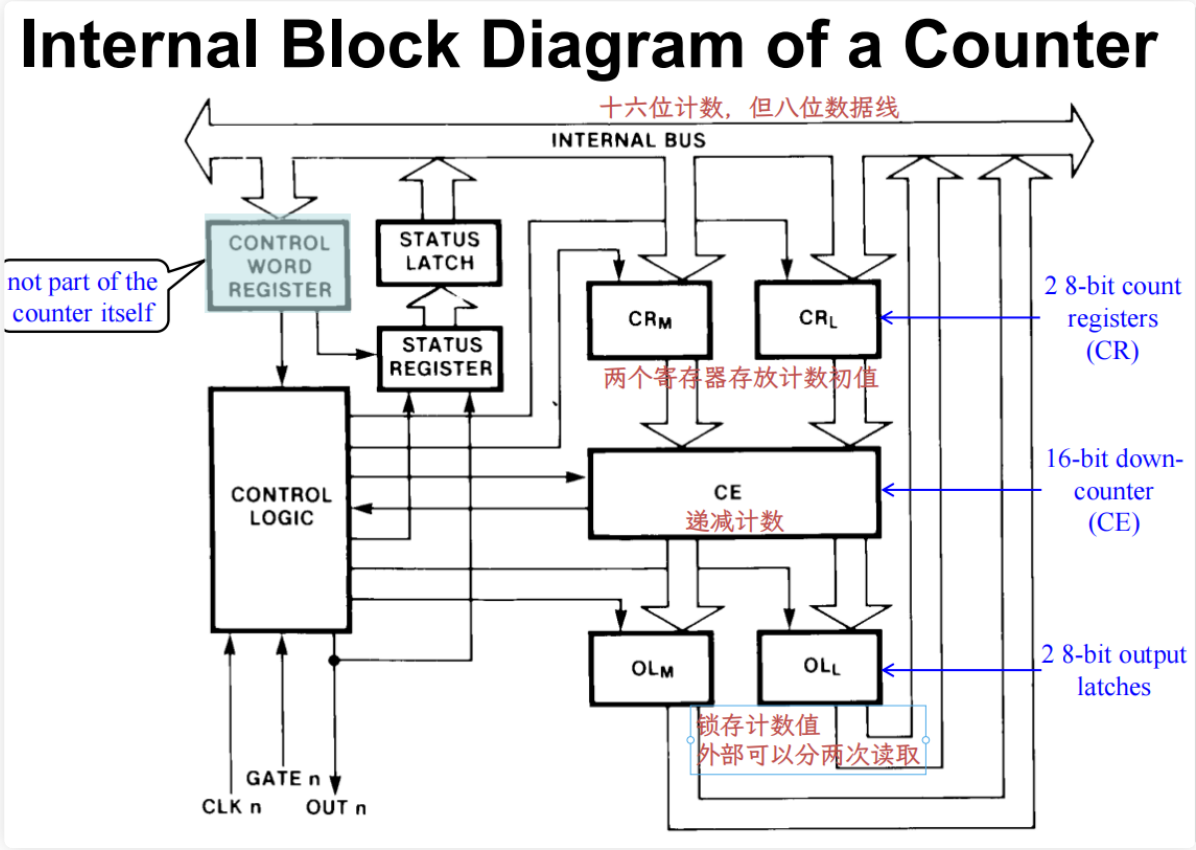
Figure 11–33 The 8254 programmable interval timer (Internal structure and pin-out)



A_1	A_0	Function
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word

三个timer + 控制

- 一个counter的内部结构



- 命令字
  - 每一个计数器都要设置control word

- 选择一个counter
- 如何进行读写
- 选择二进制 or BCD

  先减后判断，最大初始值为0；注意模式 2 和 3 的最小 Count为2（需要连续高低交替的电平）

- The largest initial count is 0:
  - 2^{16} for binary counting
  - 10^4 for BCD counting
- The smallest initial count:
  - 1 for mode 0, 1, 4, 5
  - 2 for mode 2, 3

Mode	Min Count	Max Count
0	1	0
1	1	0
2	2	0
3	2	0
4	1	0
5	1	0

- **规范**
  - 先写control后写计数初值
  - 计数初值：具体怎么写和上面命令相关（可以只写单独一个byte也可以写两个byte）
    - 可以随时写（control写完之后/运行中）
      - 先在CR锁存然后一次加载到 CE
    - 但是在同一个counter中，如果正在写一个16位寄存初值并且还没有写完，这时不能切换；（中间不能夹杂别的东西，必须要连续的写数据）
      - `control` -> `LSB` -> `MSB`

- In all four examples, all counters are programmed to read/write two-byte counts. These are only four of many possible programming sequences.

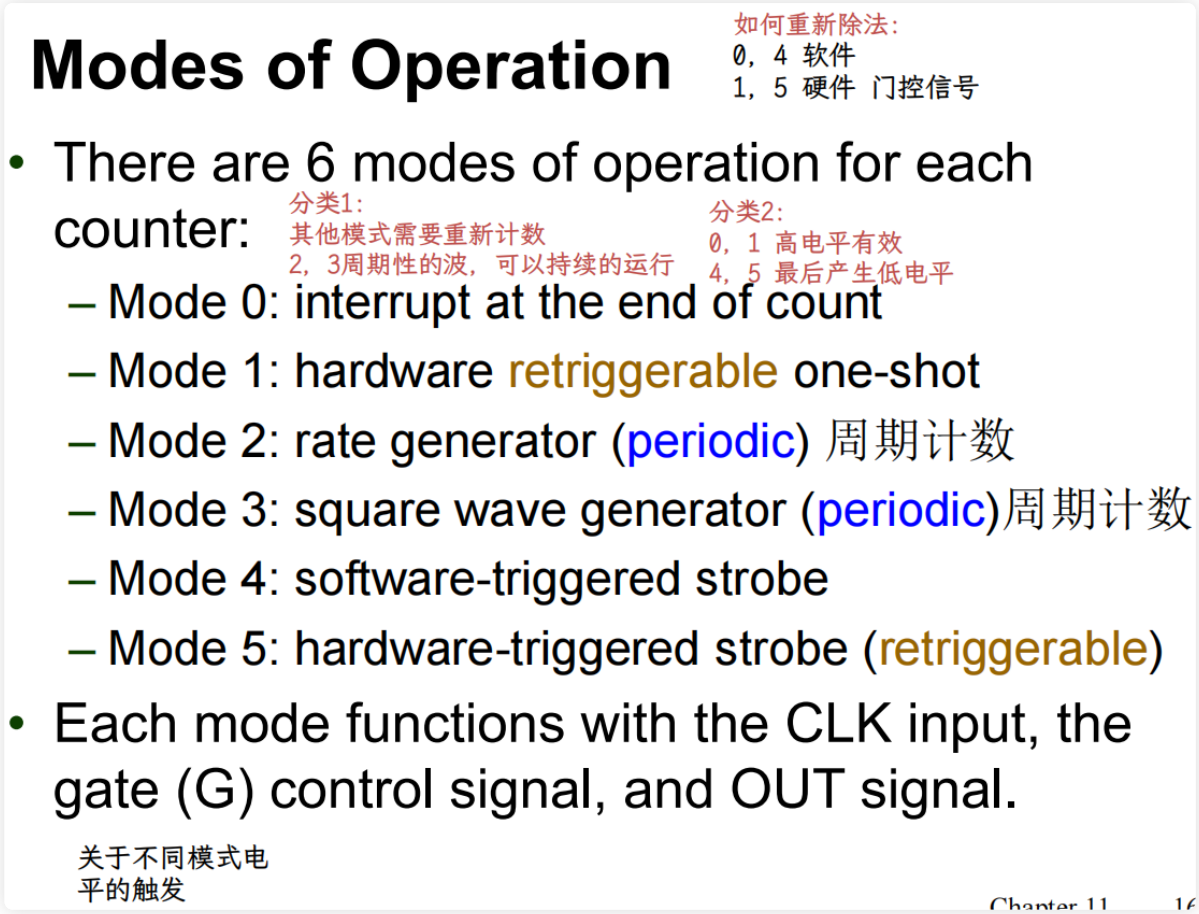
下面几种情况都可以，只要同一个counter的顺序是对的

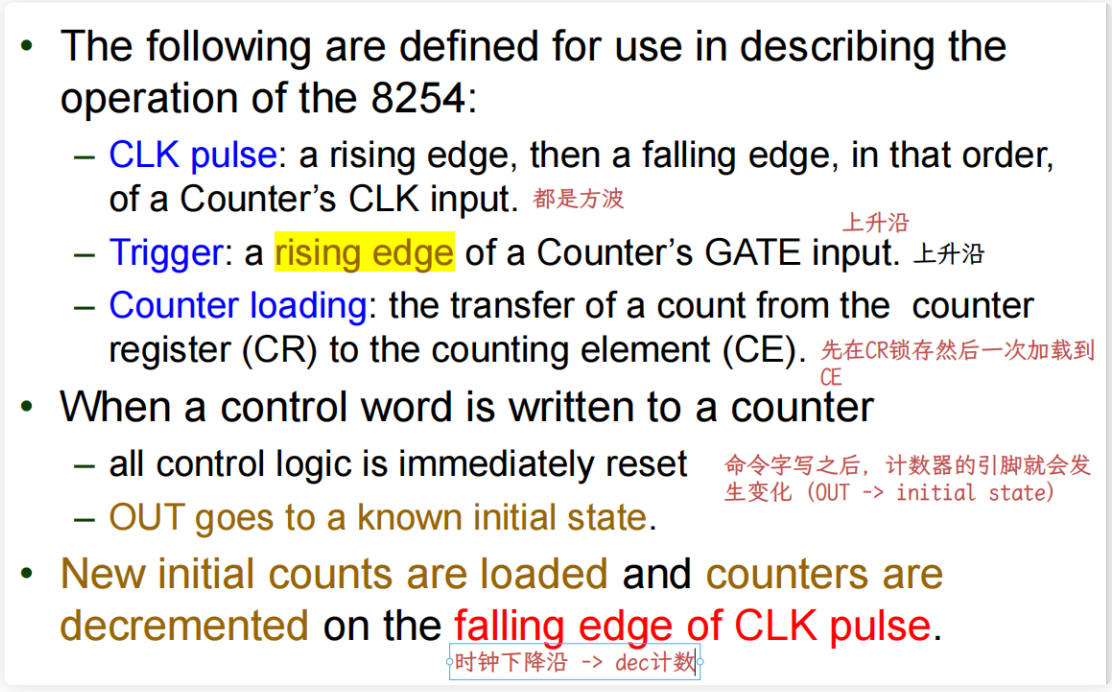
	A_1	A_0		A_1	A_0
Control Word—Counter 0	1	1	Control Word—Counter 2	1	1
LSB of count—Counter 0	0	0	Control Word—Counter 1	1	1
MSB of count—Counter 0	0	0	Control Word—Counter 0	1	1
Control Word—Counter 1	1	1	LSB of count—Counter 2	1	0
LSB of count—Counter 1	0	1	MSB of count—Counter 2	1	0
MSB of count—Counter 1	0	1	LSB of count—Counter 1	0	1
Control Word—Counter 2	1	1	MSB of count—Counter 1	0	1
LSB of count—Counter 2	1	0	LSB of count—Counter 0	0	0
MSB of count—Counter 2	1	0	MSB of count—Counter 0	0	0

	A_1	A_0		A_1	A_0
Control Word—Counter 0	1	1	Control Word—Counter 1	1	1
Control Word—Counter 1	1	1	Control Word—Counter 0	1	1
Control Word—Counter 2	1	1	LSB of count—Counter 1	0	1
LSB of count—Counter 2	1	0	Control Word—Counter 2	1	1
LSB of count—Counter 1	0	1	LSB of count—Counter 0	0	0
LSB of count—Counter 0	0	0	MSB of count—Counter 1	0	1
MSB of count—Counter 0	0	0	LSB of count—Counter 2	1	0
MSB of count—Counter 1	0	1	MSB of count—Counter 0	0	0
MSB of count—Counter 2	1	0	MSB of count—Counter 2	1	0

不同的Mode介绍

- mode 2 连续波形
- mode 3 连续的方波



- 共性
  - 

- 不同
  - 图中有些不正确的地方， Mode 2/3 是两种触发方式都有

Operation Common to All Modes

如何触发?

- For trigger, in mode 0, 2, 3, 4
  - GATE is level sensitive.
  - the GATE input is sampled on the rising edge of CLK pulse to enable the counting. 时钟上升沿采样到门控信号有效
- For trigger, in mode 1, 5 硬件控制的
  - GATE is edge sensitive. 边缘敏感：上升沿存在
  - the counting is triggered by a rising edge of GATE input.
- The counter does not stop when it reaches zero.
  - In modes 0, 1, 4, and 5 the counter "wraps around" to the highest count (FFFF or 9999), and continues counting. 回滚
  - Modes 2 and 3 are periodic; the counter reloads itself with the initial count and continues counting from there. 周期性

计数发生的时间都是在时钟的下降沿；重新加载intial count

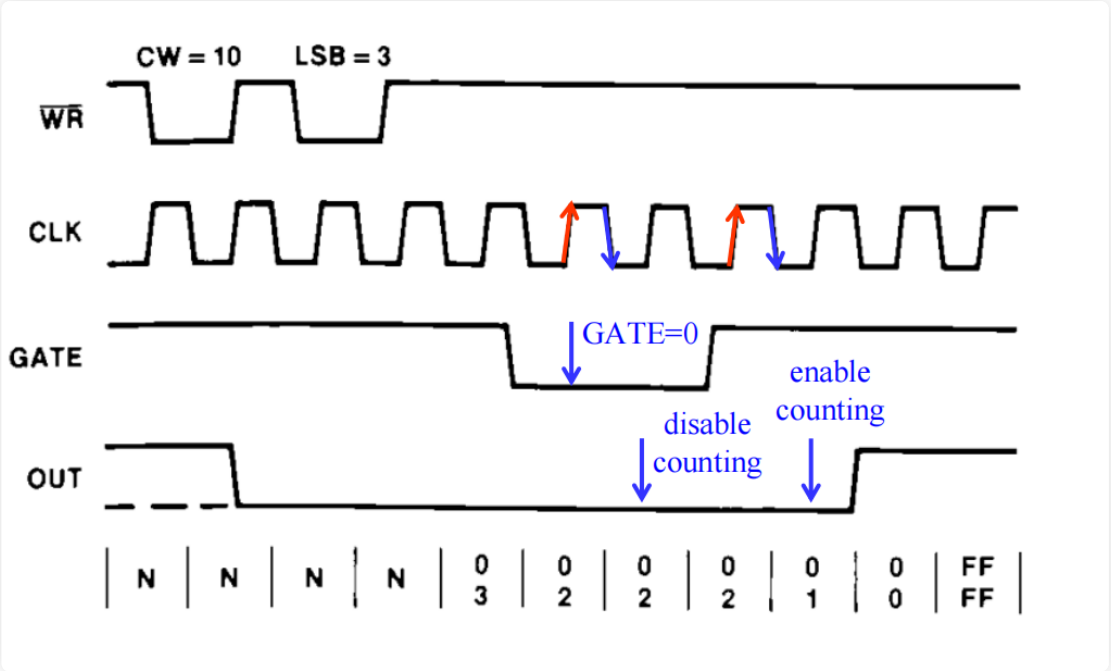
不停止是出于对简单设计的考虑

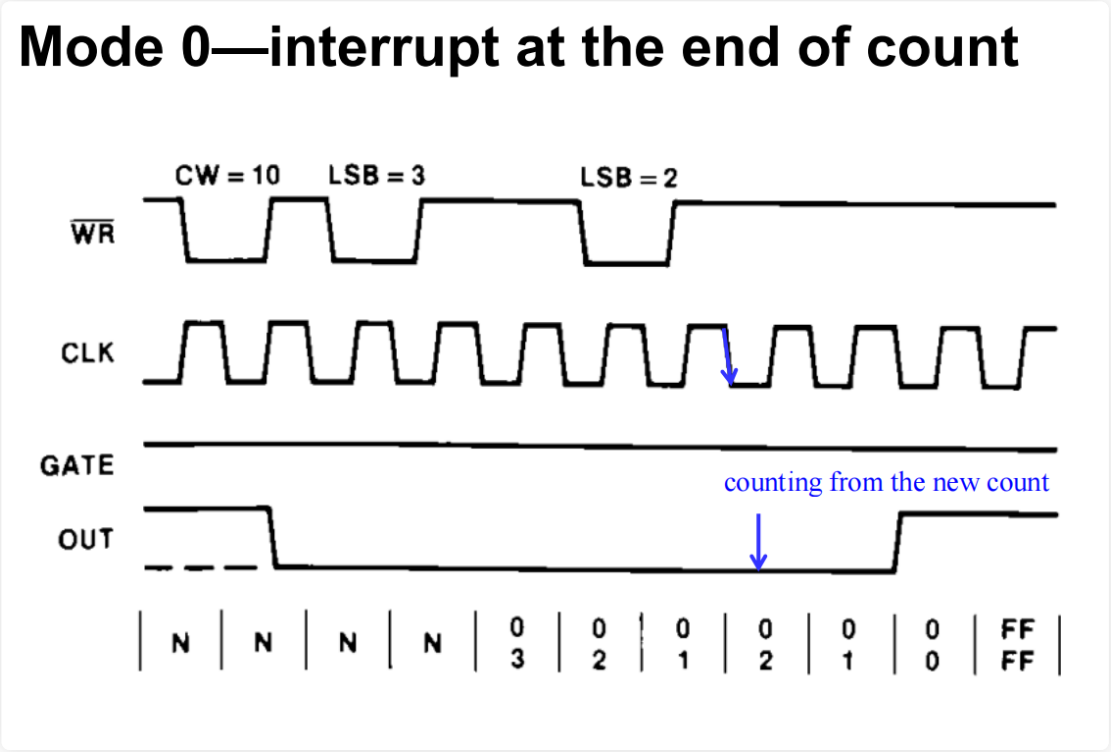
- 对于不同mode下对Gate的变化响应
  -

Gate Pin Operations Summary

Mode	Low	High	Rising-edge
Mode 0	disable counting	enable counting	
Mode 1			1) initiates counting 2) resets output
Mode 2	disable counting	enable counting	initiates counting
Mode 3	disable counting	enable counting	initiates counting
Mode 4	disable counting	enable counting	
Mode 5			initiates counting

Mode 0

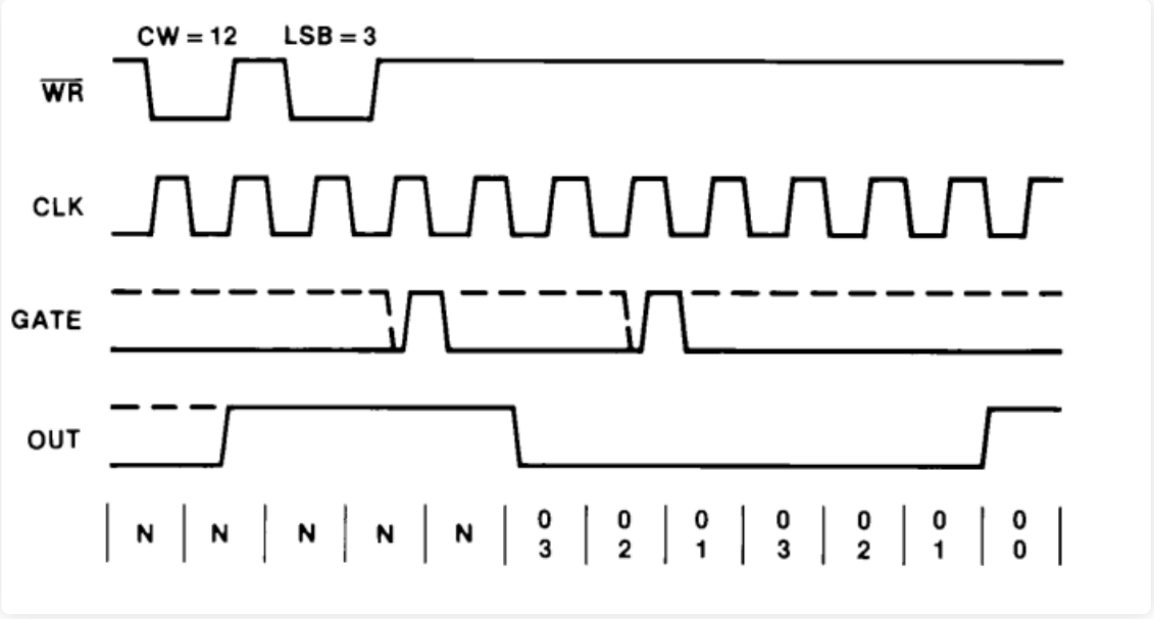
- 高电平有效
  - 在输入命令字之后变低电平，`counter`触发之后
- 上升沿采样GATE
  - 

- 下降沿dec同时也可能更新计数值
  - 

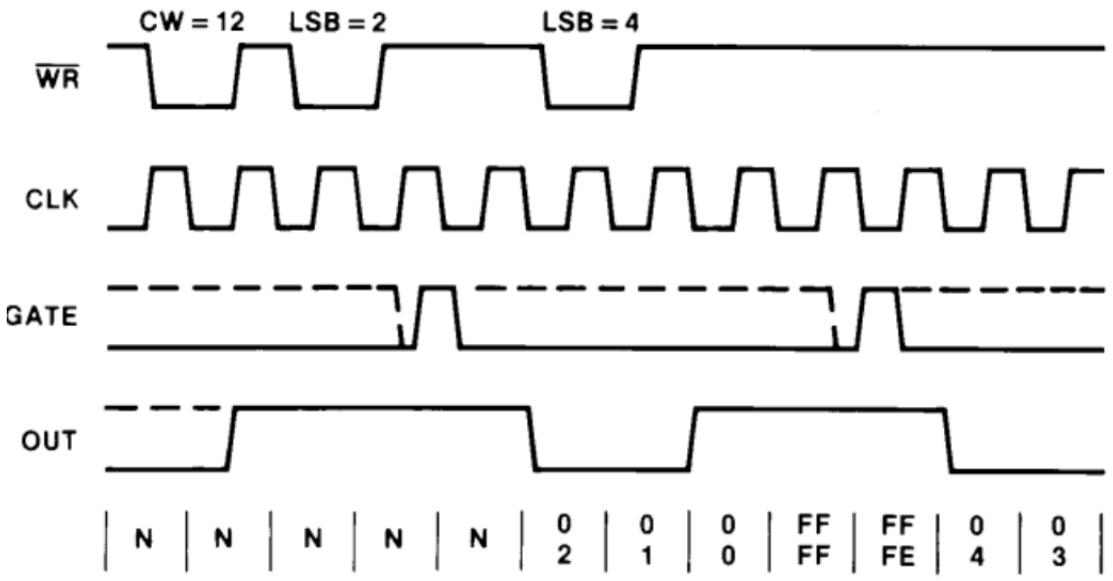
mode 1 硬件控制的 one-shot

> “
>
> 搞明白这里的边沿是怎么搞的
>
> - GATE 上升沿会被一个内部的 flip-flop 锁存
>   - 上升沿的作用不是 *enable* 而是*initialize counter*

- 高电平触发
- 但是
  - CW一写入，就变高
  - ~~门控信号有效之后~~ 装入初始值，才会变为低
  - 之后在下降沿会把初始值装入
- Gate
  - 门控信号不需要一直保持
  -
-

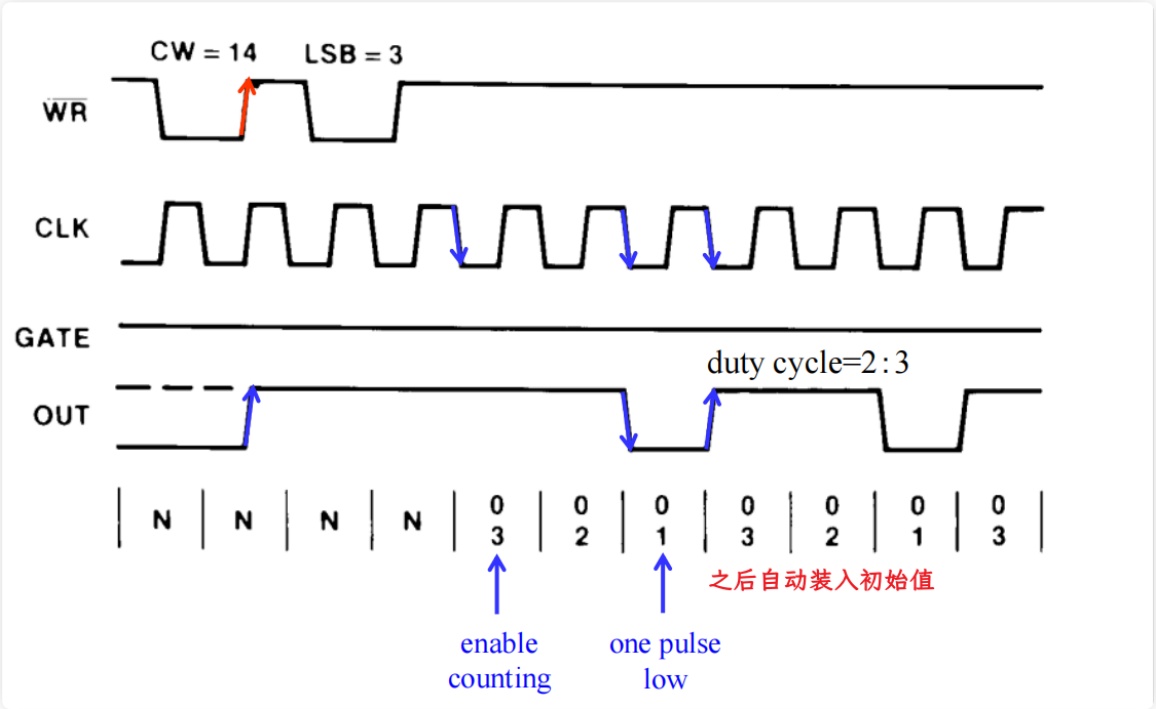


Mode 1—hardware retriggerable one-shot



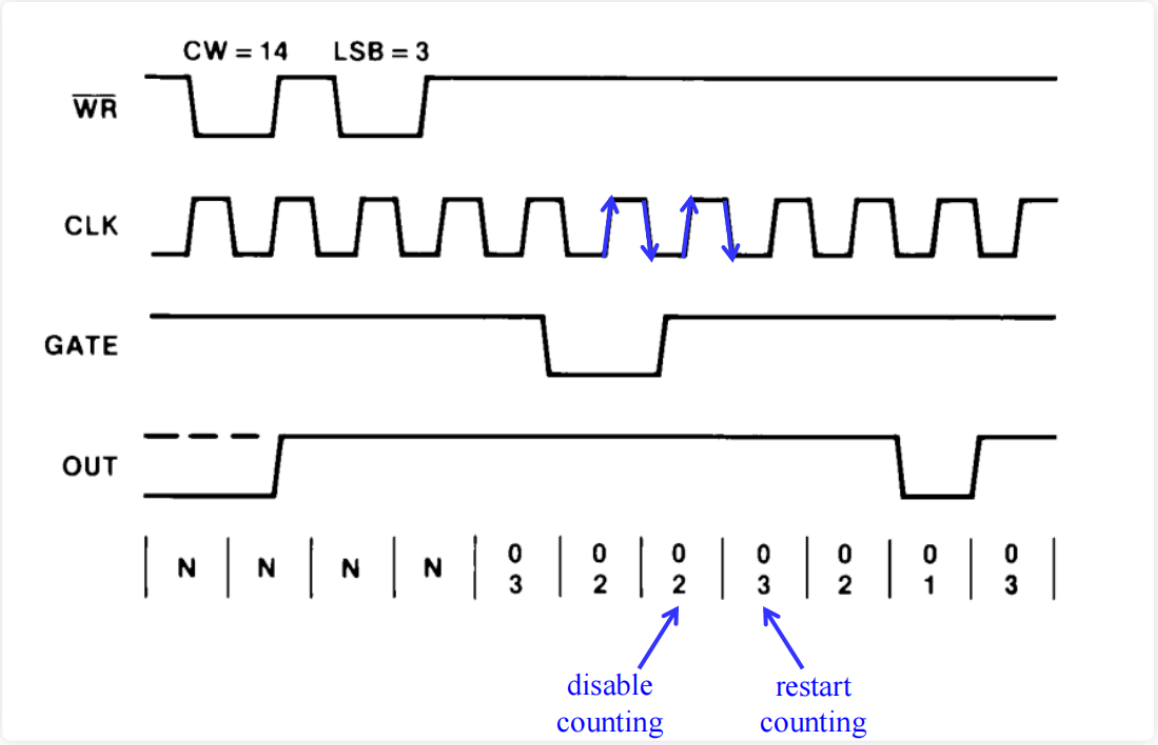
mode 2 N-分频器

- 计数到1产生低电平，之后自动装入原来的初始值重新计数



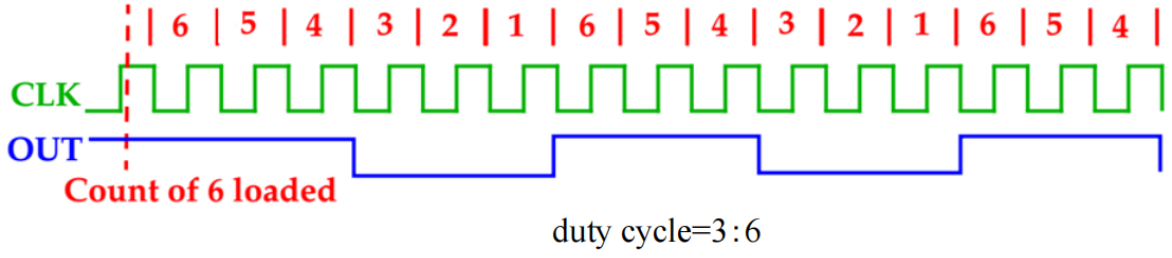
- duty cycle = $\frac{N-1}{N}$

- Gate信号会使得重新计数，并且会使得输出一直为高



mode 3 方波产生

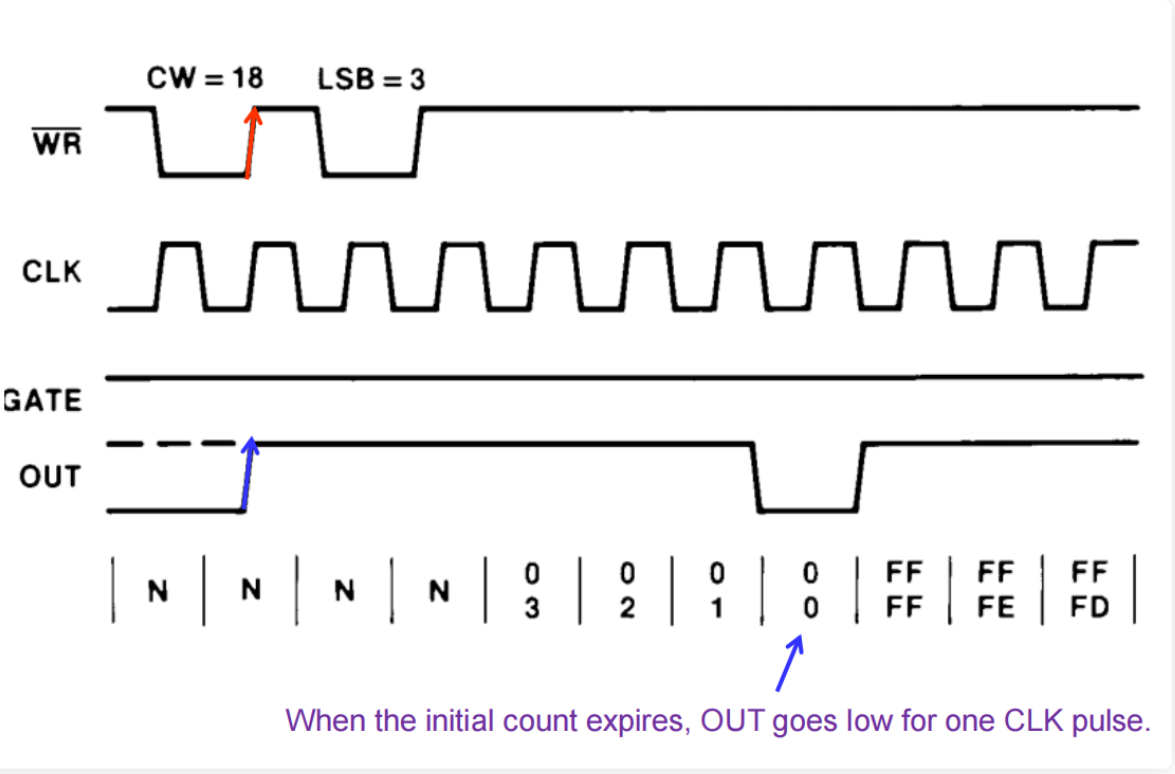
- Generates a continuous square wave at the OUT connection, provided the G pin is logic 1.
- OUT is initially high
  - even counts: 50% duty cycle. 尝试产生方波
  - odd counts: (N+1)/2 high and (N-1)/2 low counts.
  - duty cycle = $\frac{1}{2}$ or $\frac{N+1}{2N}$ 最小计数值如果为1那么将占空比 = 1



duty cycle=3:6

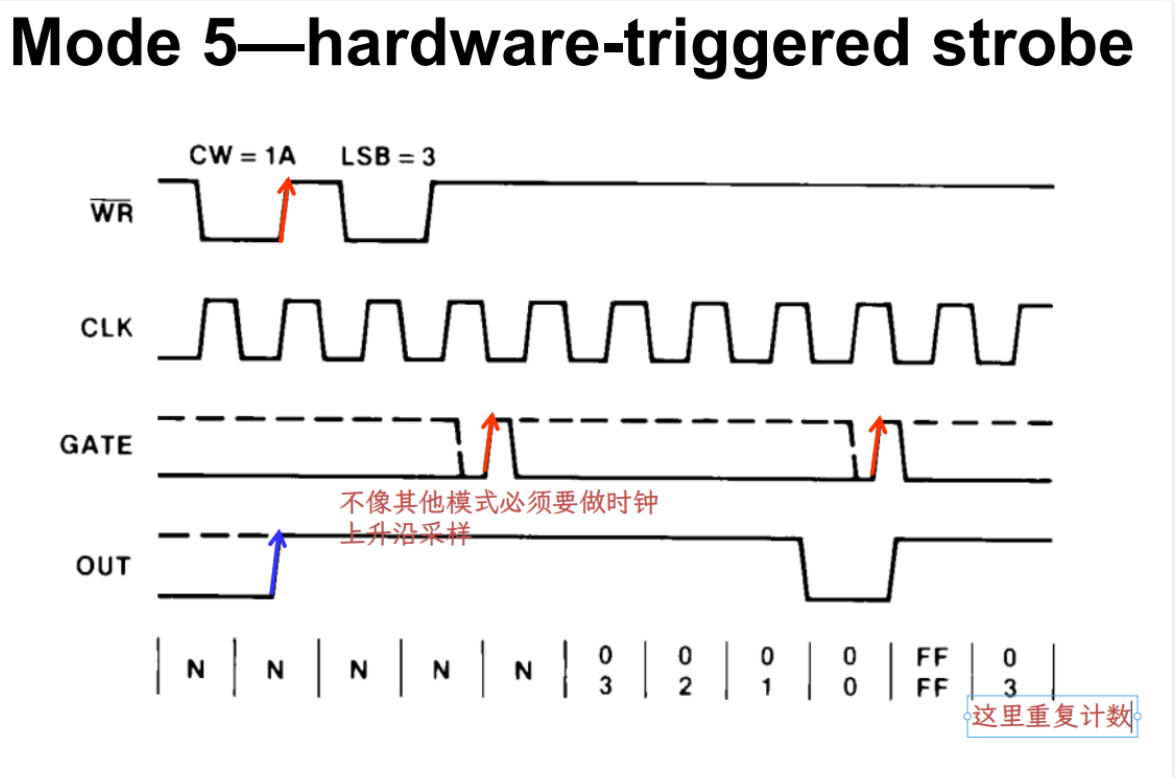
Mode 4 software 触发（选通）

- 触发 => 产生一个周期低电平



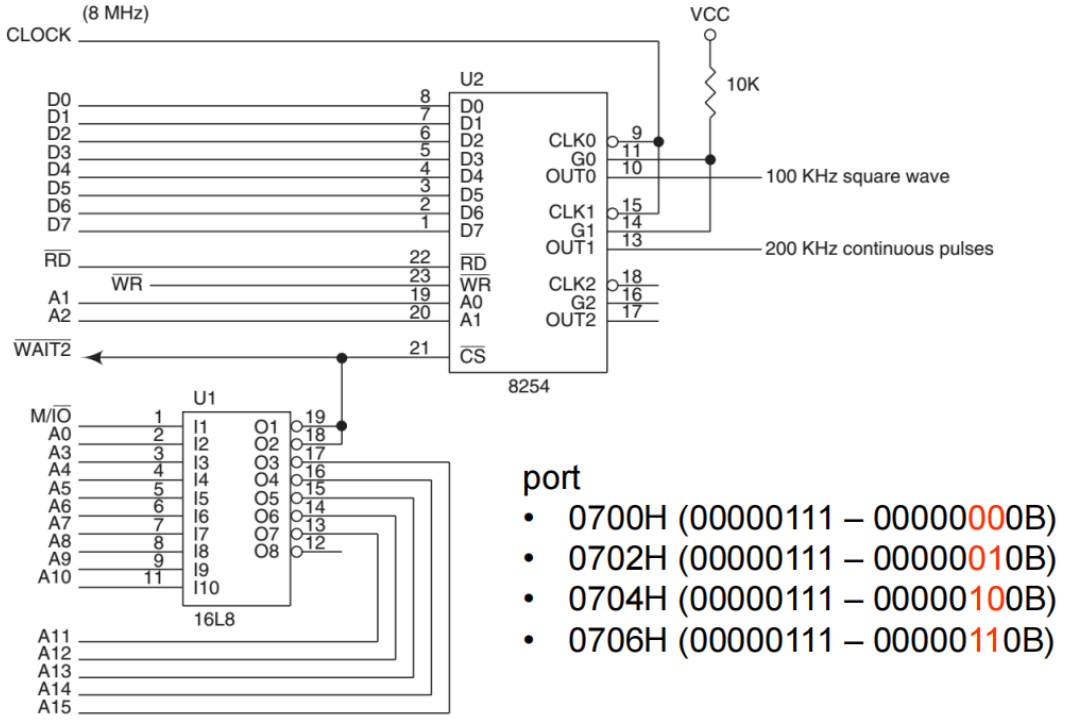
Mode 5

- 不像其他模式必须要做时钟上升沿采样



例子

Figure 11–36 The 8254 interfaced to an 8 MHz clock so that it generates a 100 KHz square wave at OUT0 and a 200 KHz continuous pulse at OUT1.



- 计算counter（都是N-分频）
  - 要注意是否能够写进去
  - 可能需要串联分频
- 写命令字
- 写counter初始值
  - 先写低
  - 再写高（必须写，否则状态机状态不对）

端口 3 counter + 1 control
I/O ports: 0700H, 0702H,
0704H, 0706H

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

control word format

EXAMPLE 11–24

```
;A procedure that programs the 8254 timer to function
;as illustrated in Figure 11-36

TIME   PROC   NEAR USES AX DX

       MOV   DX,706H                    ;program counter 0 for mode 3
       MOV   AL,00110110B               ; control word: 00 11 011 0
       OUT   DX,AL
       MOV   AL,01110100B               ;program counter 1 for mode 2
       OUT   DX,AL                      ; control word: 01 11 010 0

       MOV   DX,700H                    ;program counter 0 with 80
       MOV   AL,80
       OUT   DX,AL
       MOV   AL,0
       OUT   DX,AL

       MOV   DX,702H                    ;program counter 1 with 40
       MOV   AL,40
       OUT   DX,AL
       MOV   AL,0
       OUT   DX,AL

       RET

TIME   ENDP
```

write to control word register

write initial count for counter 0

write initial count for counter 1

读寄存器

simple read

Counter latch command

- 读取counter

Counter Latch Command 寄存器锁存当前的counter

- Count latch command is written to the control word register. The SC0, SC1 bits select one of the three counters, and D5, D4 bits with 00 designate a counter latch command.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	0	0	X	X	X	X

- The selected counter's output latch (OL) latches the count at the time the counter latch command is received. This count is held until it is read by the CPU (or until the counter is reprogrammed) and then return to "following" the CE. 只有当这个counter被读取之后/counter重新设置之后才能继续

- 读取例子
  - 根据之前设置的命令进行读
  - For example, read the counts from counter 2 in 8254 with port number 40H-43H, given that it is programmed for two-byte counts.

```
MOV   AL, 10000000B   <-- count latch command
OUT   43H, AL
IN    AL, 42H         <-- read least significant byte
MOV   AH, AL
IN    AL, 42H         <-- read most significant byte
XCHG  AH, AL          <-- reverse the byte order
```

Read-Back Command

- 读取的信息更多

Read-Back Command

- When necessary for contents of more than one counter to be read at the same time, the read-back control word is used.
- This command allows the user to check the count value, programmed mode, and current states of the OUT pin and Null Count flag of the selected counter(s).
- Null Count flag indicates whether the counter has been initialized correctly or if a count value has been written to it.

- 如果同时，那么先读取status

Figure 11–38 The 8254-2 read-back control word.

A0, A1 = 11 **$\overline{CS}$ = 0** **$\overline{RD}$ = 1** **$\overline{WR}$ = 0**

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	$\overline{COUNT}$	$\overline{STATUS}$	CNT 2	CNT 1	CNT 0	0

D_5: 0 = Latch count of selected counter(s)
D_4: 0 = Latch status of selected counters(s)
D_3: 1 = Select Counter 2
D_2: 1 = Select Counter 1
D_1: 1 = Select Counter 0
D_0: Reserved for future expansion; Must be 0

- If both count and status of a counter are latched:
  - the first read operation will return latched status.
  - the next one or two reads return latched count.

Read-Back Command Example

注意读取为低电平有效

Command								Description
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
1	1	0	0	0	0	1	0	Read back count and status of Counter 0
1	1	1	0	0	1	0	0	Read back status of Counter 1
1	1	1	0	1	1	0	0	Read back status of Counters 2, 1
1	1	0	1	1	0	0	0	Read back count of Counter 2
1	1	0	0	0	1	0	0	Read back count and status of Counter 1
1	1	1	0	0	0	1	0	Read back status of Counter 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	$\overline{\text{COUNT}}$	$\overline{\text{STATUS}}$	CNT 2	CNT 1	CNT 0	0

DC Motor Speed and Direction Control

考试不考，电路没太看懂

大体思路就是利用H桥，通过控制占空比，进而操作电机的方向（使用相位差）

16550 串行通信

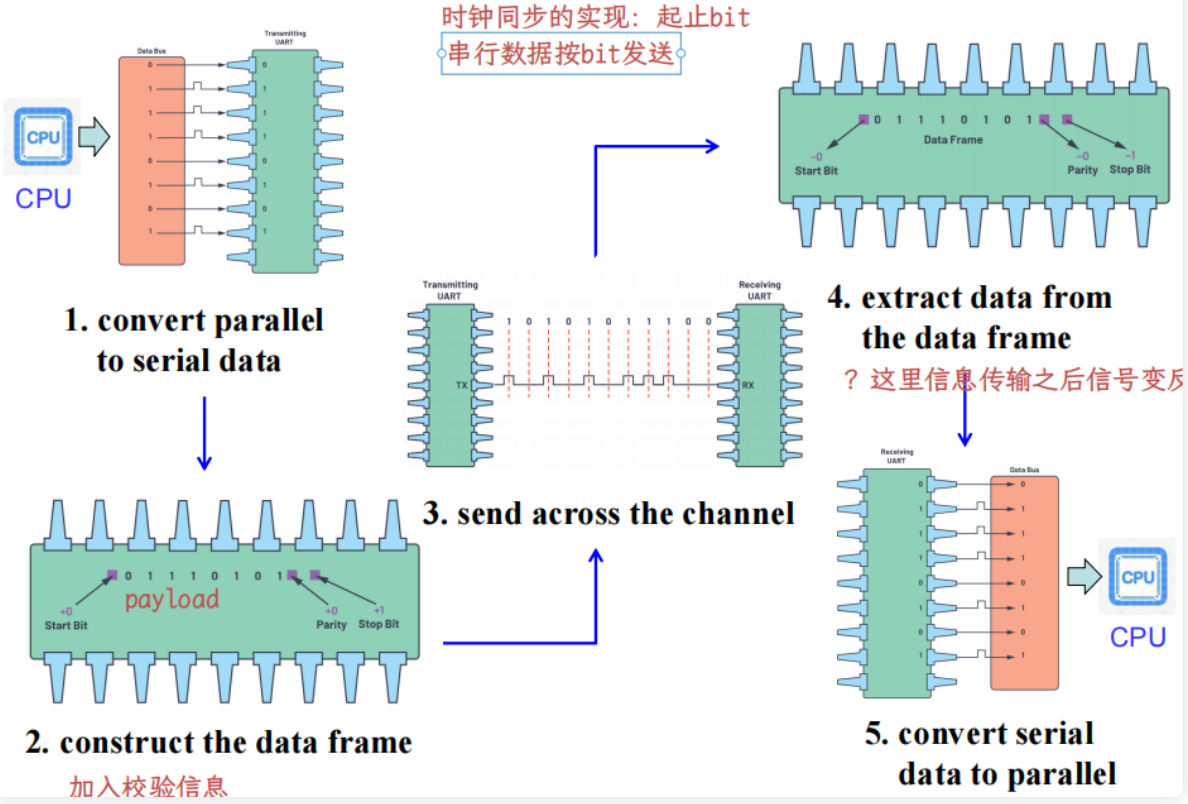
串行的基本概念

Three Modes of Transmission

和计网中的概念相同

- There are three modes of transmission:
  - **Simplex mode:** data can only be transferred from transmitter to receiver and not vice versa.
  - **Half duplex mode:** data transmission can occur in only one direction at a time.
  - **Full duplex mode:** data can be transmitted from the master to the slave, and from slave to the master simultaneously.

Steps of Serial Communication

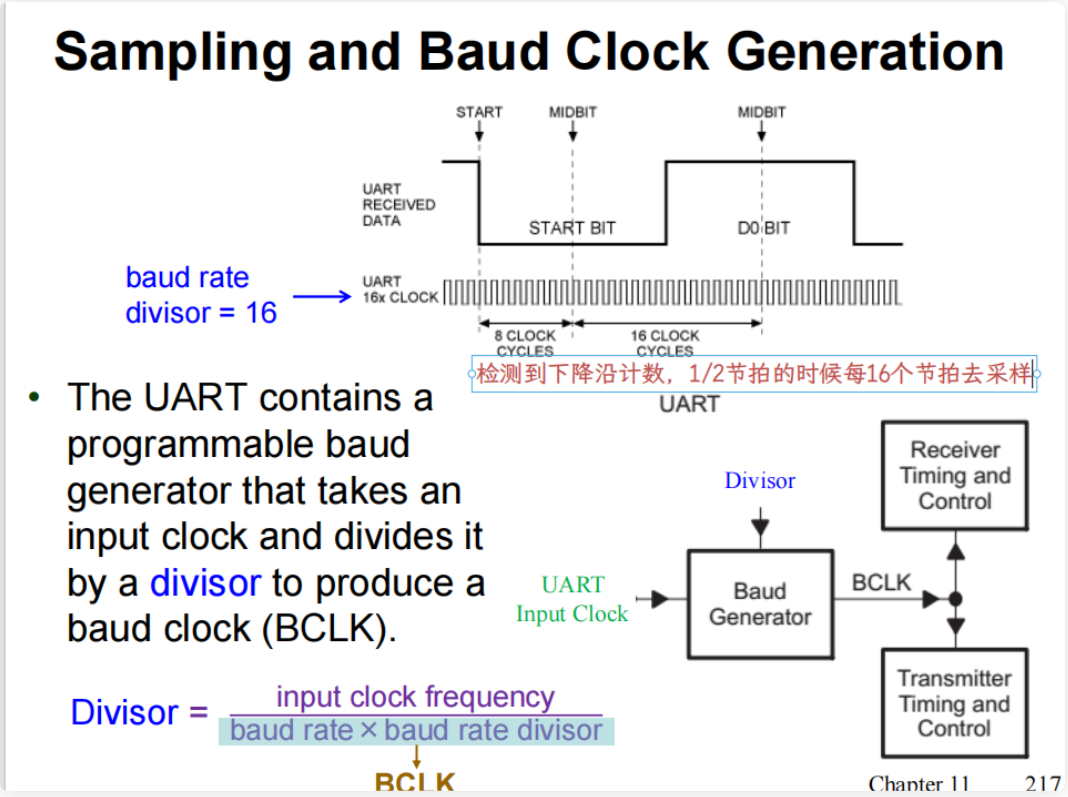


1. CPU通过数据总线将数据送给串行通信接口（Convert parrallel to serial）
2. 给数据头尾加一些信息（一些控制位，例如校验位、同步信号）
3. 发送给目的端串行通信芯片

4. 对控制位进行校验和分析，抽取出有效数据部分（Convert serial to parallel）

时钟同步

- 异步数据通信 asy
  - baud rate 每秒钟传输多少个bit，使用 bps 衡量，但是要注意的是，实际过程中传输的有效数据的值会远远小于这个理论值（其他信号 + 停顿 之类的会夹杂）
  - BCLK 波特率时钟 baud clock
    - 异步传输接收端实际使用的时钟
      - BCLK = baud rate * baud rate divisor 进行过采样
    - 分频的计算 查表



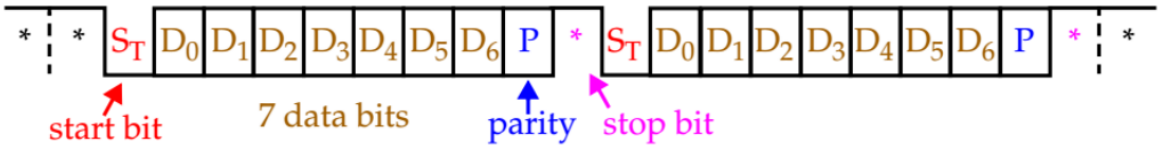
- 同步数据通信 sy
  - 使用统一的外部时钟
  - 或者
- UART 异步，更简单
- USART 可用同步/异步

正式介绍

- 16550 operates at 0–1.5 M baud.
  - baud rate is bps (bits transferred per second) including start, stop, data, and parity
  - bps are bits per second; Bps is bytes per second
- 16550 includes a programmable baud rate generator, which divides the input clock to produce a baud clock (BCLK). The frequency of BCLK is sixteen times (16×) the baud rate.

Asynchronous Serial Data

- Asynchronous serial data are transmitted and received without a clock or timing signal.



- 提供了 16 个 byte 大小的 FIFO 缓冲区
- 停止位 stop bit 比较特殊，可以是 1 / 2 / 1.5位(维持半个时钟)
- Modem control functions 内置的调制解调
- Status reporting
- 可以全双工工作
- 一共可以使用 12 个端口

16550 Pin Functions
A_0, A_1, A_2

- The **address inputs** are used to select an internal register for programming and also data transfer.

A_2	A_1	A_0	*Function*
			multiplexing r/w 两个端口，读写分离
0	0	0 m	Receiver buffer (read) and transmitter holding (write)
0	0	1 m	Interrupt enable
0	1	0 m	Interrupt identification (read) and FIFO control (write)
0	1	1	Line control 有效数据载荷
1	0	0	Modem control
1	0	1	Line status
1	1	0	Modem status
1	1	1	Scratch general-purpose 临时数据存储/debug

分频系数的设置

Register Addresses

DLAB	A_2	A_1	A_0	Register	
0	0	0	0	Receiver Buffer (read), Transmitter Holding Register (write)	address multiplexing
0	0	0	1	Interrupt Enable	
X	0	1	0	Interrupt Identification (read)	address multiplexing
X	0	1	0	FIFO Control (write)	
X	0	1	1	Line Control	
X	1	0	0	MODEM Control	
X	1	0	1	Line Status	
X	1	1	0	MODEM Status	
X	1	1	1	Scratch	
1	0	0	0	Divisor Latch (least significant byte)	
1	0	0	1	Divisor Latch (most significant byte)	

address multiplexing

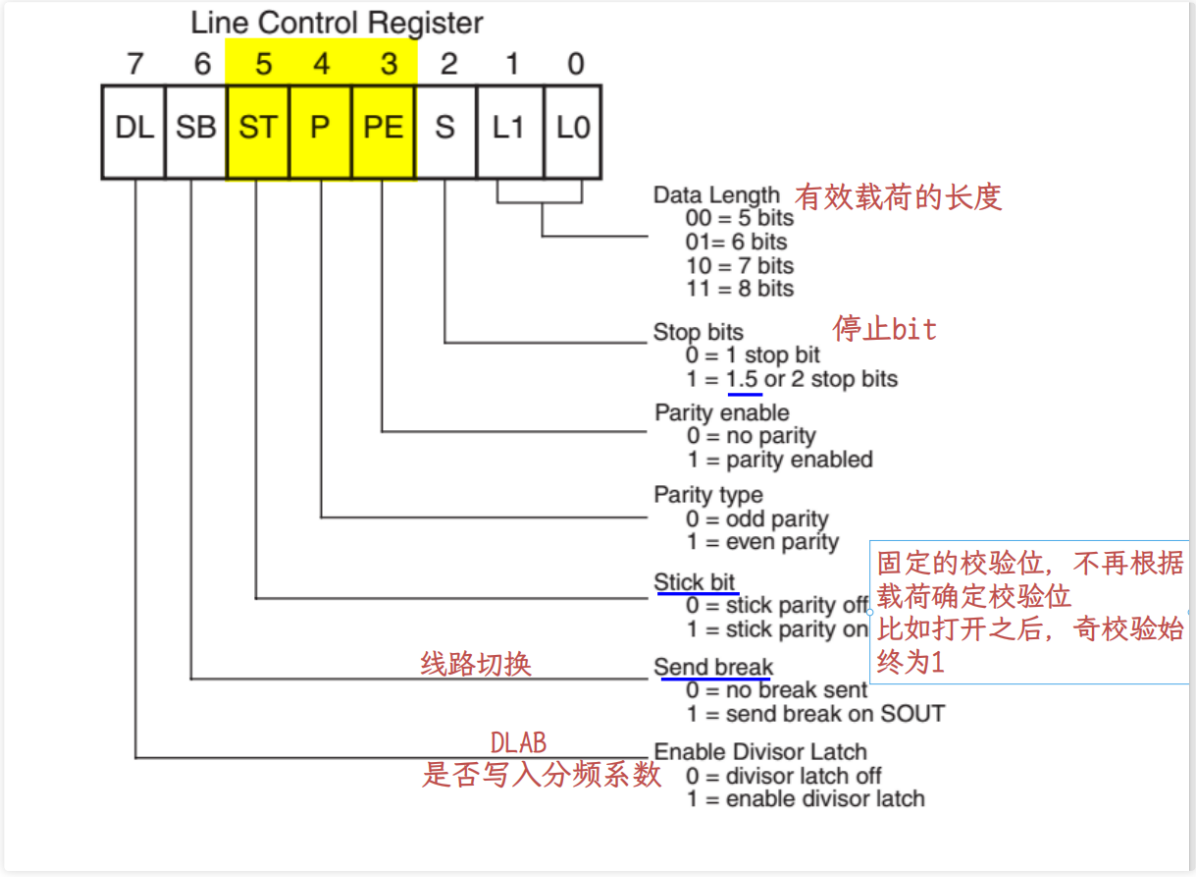
baud 的 分频系数 <= 分时，二者不可能同时发生

- Note that the state of the Divisor Latch Access Bit (DLAB), which is the most significant bit of the Line Control Register, must be set high to access the Baud Generator Divisor Latches.

编程

- Programming is a two-part process:
  - initialization (setup) 波特率 数据帧格式
    - Program the baud rate generator for the required baud rate 波特rate
    - Program the line control register to set transmission parameters (# of stop, data, and parity bits, etc.)
  - operation
    - Clear the transmitter & receiver FIFOs clear out
    - Actual communication

- Control 命令字



- ST, P and PE used to send even or odd parity, to send no parity or to send a 1 or a 0 in the parity bit position for all data.

ST	P	PE	Function
0	0	0	No parity
0	0	1	Odd parity
0	1	0	No parity
0	1	1	Even parity
1	0	0	Undefined
1	0	1	Send/receive 1
1	1	0	Undefined
1	1	1	Send/receive 0

send 1 in place of the parity bit

send 0 in place of the parity bit

具体流程

Register Addresses

DLAB	A_2	A_1	A_0	Register
0	0	0	0	Receiver Buffer (read), Transmitter Holding Register (write)
0	0	0	1	Interrupt Enable
X	0	1	0	Interrupt Identification (read)
X	0	1	0	FIFO Control (write)
X	0	1	1	Line Control
X	1	0	0	MODEM Control
X	1	0	1	Line Status
X	1	1	0	MODEM Status
X	1	1	1	Scratch
1	0	0	0	Divisor Latch (least significant byte)
1	0	0	1	Divisor Latch (most significant byte)

清除历史
使能FIFO
④ enable FIFO register

③ set line register
设置DLAB=0

② set divisor value
set Divisor

① enable divisor latch access bit (DLAB)
先设置DLAB=1

- **初始化**
  - 选择波特率时钟

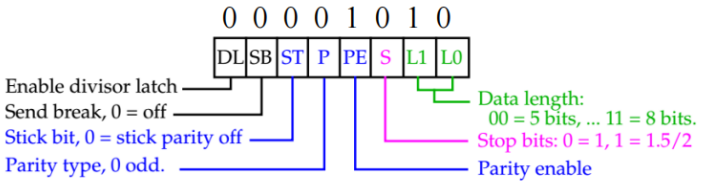
- For 18.432MHz crystal, 10,473 gives 110 band rate, 30 gives 38,400 baud.

$$10473 = \frac{18{,}432{,}000}{110 \times 16}$$

← input clock
← baud rate divisor

divisor　baud rate

EXAMPLE 11–26 Initialization for baud rate 9600, 7 data, odd parity, 1 stop




```
LINE   EQU    0F3H
LSB    EQU    0F0H
MSB    EQU    0F1H
FIFO   EQU    0F2H

INIT   PROC   NEAR

       MOV   AL,10001010B    ;enable baud rate divisor
       OUT   LINE,AL

       MOV   AL,120          ;program baud 9600
       OUT   LSB,AL

       MOV   AL,0
       OUT   MSB,AL

       MOV   AL,00001010B    ;program 7 data, odd
       OUT   LINE,AL         ;parity, 1 stop

       MOV   AL,00000111B    ;enable transmitter and
       OUT   FIFO,AL         ;receiver

       RET

INIT   ENDP
```

divisor=120 for baud rate 9600

高的部分清零

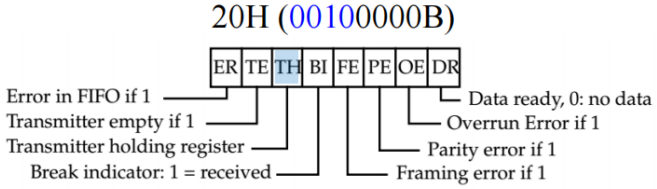
set transmission parameters

使能FIFO

0. 设置 DLAB bit
1. 设置 divisor
2. 设置 status
3. 使能 FIFO

- **发送数据**

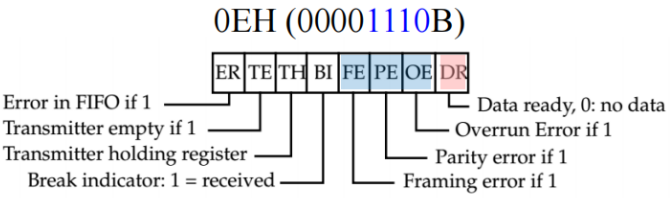
Example 11–27 A procedure that transmits the contents of AH to the 16550 is listed below. The TH bit is polled to determine whether the transmitter is ready to receive data.




```
    LSTAT  EQU    0F5H
    DATA   EQU    0F0H

    SEND   PROC   NEAR USES AX

polling能否发送 REPEAT                    ;test the TH bit
                IN    AL,LSTAT
                TEST  AL,20H
           .UNTIL !ZERO?

           MOV   AL,AH                    ;send data
           OUT   DATA,AL
           RET

    SEND   ENDP
```

- 接收数据

Example 11–28 A procedure that receives data from the 16550 UART and returns it in AL.




```
LSTAT  EQU    0F5H
DATA   EQU    0F0H

REVC   PROC   NEAR

       .REPEAT
             IN    AL,LSTAT        ;test DR bit
             TEST AL,1
       .UNTIL !ZERO?

       TEST AL,0EH                 ;test for any error
       .IF ZERO?                   ;no error
             IN    AL,DATA
       .ELSE                       ;any error
             MOV   AL,'?'
       .ENDIF
       RET

RECV   ENDP
```

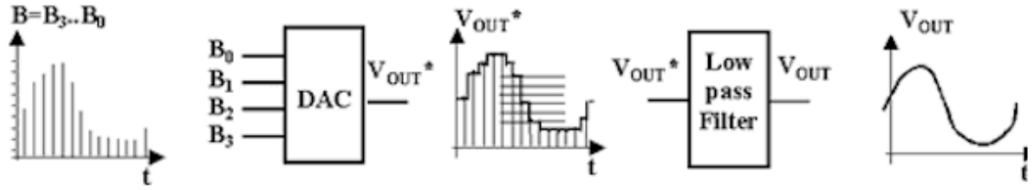
模拟信号 ADC DAC

> “
>
> 主要介绍DAC.

- 零阶保持的

- 低通滤波

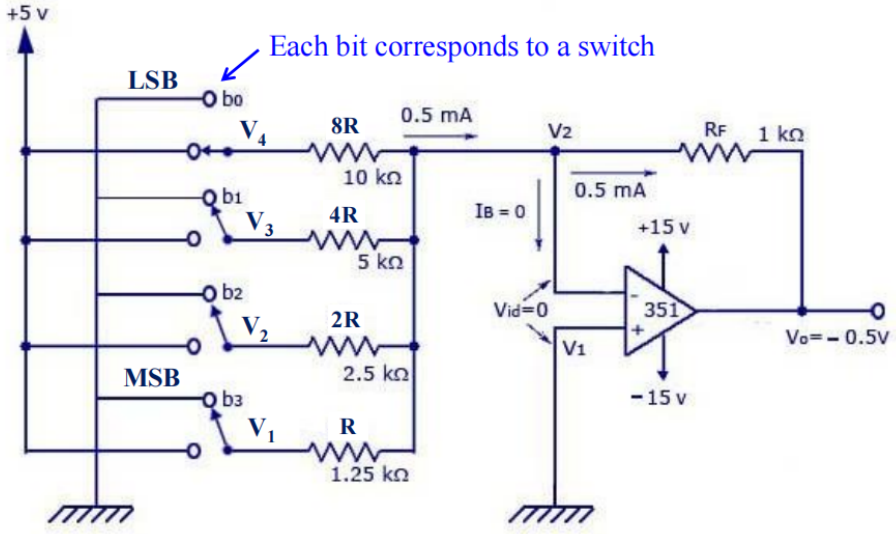
Zero-Order Hold



- In order to reconstruct a signal, a typical DAC will latch binary information for a full sample period using a clock signal; this is the reason that digital signals appear as stair-like lines when graphed.
- Therefore, a low-pass filter must be employed to eliminate extraneous harmonics and smooth the signal to render it suitable for output.

- Binary

Binary-Weighted Resistors Method



full-scale output voltage

MSB

LSB

$$V_{out} = -IR_f = -R_f\left(\frac{V_1}{R} + \frac{V_2}{2R} + \frac{V_3}{4R} + \cdots \frac{V_n}{2^{n-1}R}\right)$$

$$V_{out} = -IR_f = -R_f\left(\frac{V_1}{R} + \frac{V_2}{2R} + \frac{V_3}{4R} + \cdots \frac{V_n}{2^{n-1}R}\right)$$

If $R_f = R/2$, $V_1=V_2=\cdots\cdots V_n=V_{ref}$

$$V_{out} = -IR_f = -\left(\frac{V_1}{2} + \frac{V_2}{4} + \frac{V_3}{8} + \cdots \frac{V_n}{2^n}\right)$$

For example, a 4-bit DAC converter yields

$$V_{out} = -V_{ref}\left(b_3\frac{1}{2} + b_2\frac{1}{4} + b_1\frac{1}{8} + b_0\frac{1}{16}\right)$$

Where b_3 corresponds to bit-3, b_2 to bit-2, etc.

- R-2R

R-2R Ladder Method

For a 4-Bit R-2R Ladder:

$$V_{out} = -V_{ref}\left(b_3\frac{1}{2} + b_2\frac{1}{4} + b_1\frac{1}{8} + b_0\frac{1}{16}\right)$$

For n-bit R-2R ladder DAC:

$$V_{out} = -V_{ref}\sum_{i=1}^{n} b_{n-i}\frac{1}{2^i}$$

- 参考电压 REF
- 满量程电压 FS

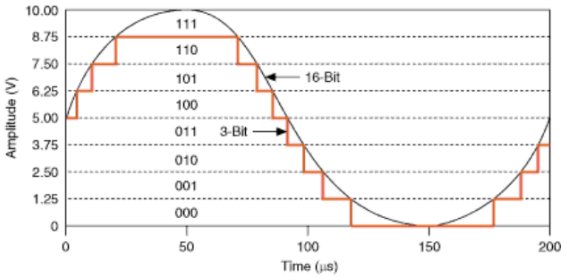
- **Reference Voltage (V_{REF})** 参考电压
  - V_{REF} is the external or internal voltage used as the basis for generating the output voltage levels. V_{REF} defines the voltage range.
- **Full-Scale Output Voltage (V_{FS})** 满量程输出
  - V_{FS} the maximum output voltage that the DAC can produce.
  - V_{FS} depends on the V_{REF} and the internal circuitry of the DAC. For many DACs, V_{FS} is slightly less than the V_{REF}.

	参考电压	满量程输出电压 比参考电压略小
	Reference Voltage (V_{REF})	Full-Scale Output Voltage (V_{FS})
Definition	Voltage that determines the output voltage range	Maximum output voltage the DAC can produce
Relation to Output	Determines the step size for D/A conversion	Defines the upper limit of the DAC's output range
Magnitude	Set externally or internally	Often slightly less than V_{REF}

- **参数**

- Resolution 这里可能出计算题

Resolution of DAC Converter



- Resolution is defined in two ways:
  - the number of different output values
    - For an n-bit DAC, resolution = 2^n
  - the step size that generates the smallest voltage or current change in LSB of input for the DAC
    - For an n-bit DAC,

$$\text{resolution} = \frac{V_{REF}}{2^n} = \frac{V_{FS}}{2^n - 1}$$

  - example

- Problem1: A 4-bit DAC has a full scale output voltage of 15V. Calculate the output voltage for the input code $(0110)_2$.
  - resolution = $15/(2^4-1)$ = 1V/LSB
  - the output voltage for the input of 0110_2 ($6_{(10)}$) is:
  - $V_o = 1 * 6 = 6V$

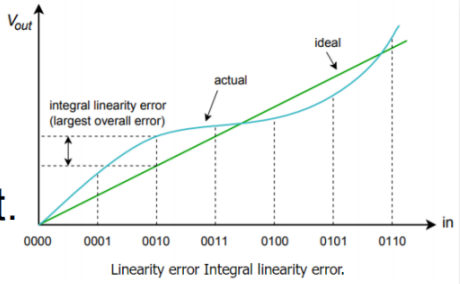
ref = 理论 max

- Problem2: A 12-bit DAC has a step size of 8 mV. Determine the reference voltage and the output voltage for the input code $(010101101101)_2$?
  - $V_{REF} = 8 * 10^{-3} * 2^{12} = 32.768V$
  - the output voltage for the input of 010101101101 is: 8 mv * $1389_{(10)}$ = 11.112V

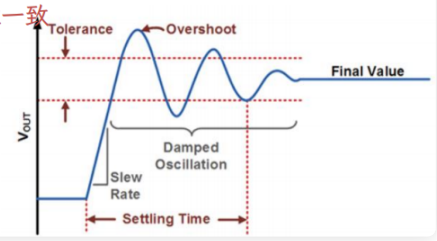
- Linearity 线性度
  - 输出电压的准确性：理想值和实际电压最大差值

- Settling Time 稳定时间

- Linearity 输出电压的准确性：最大差值
  - The maximum deviation between the desired analog output and the actual output.
- Settling Time 稳定时间：LSB/2 这里的LSB和之前的最小单位一致
  - The time required from a code transition until the output reaches within ±1/2 LSB of the final output.



Linearity error Integral linearity error.



DAC0830

App

64 bit 寻址

- **RIP-relative addressing** 64位模式引入
-
  - RIP-relative addressing uses a signed 32-bit displacement to calculate the effective address of the next instruction by sign-extend the 32-bit value and add to the 64-bit value in RIP, e.g.,

```
int var;

void f(int x) {
        var = x;
}
```

```
f:
        mov     var[rip], edi    ; edi = x
        ret
```

  - Using RIP-relative addressing makes position-independent code smaller and simpler, where all code and data need to be addressable within a 32-bit offset.

- 64 位地址的高16位必须要是符号拓展，要么 `FFFF` OR `0000`
  - For a 48-bit linear address, a canonical address must have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one),
    - canonical address: FFFF8010BC001000, 00007C80B8102040
    - non-canonical address: 1122334455667788, 3375DA44B5667788