本文改编补充于董佬的dhy2000/x86Course: BUAA X86 汇编程序设计 课程笔记和实验作业 (github.com),并添加了一些自己的理解

常用子程序

DOS 功能调用相关

调用指令：`INT 21H`，其中 `21H` 表示 DOS。

用寄存器 `AH` 指定功能调用号。

在实际编码发现虽然可能只使用了AH也可能AL被修改，所以实际编码中最好清空整个AX或DX

功能调用号 (AH)	输入	输出	功能
1	console	AL	输入一个ASCII字符
2	DL	console	输出一个ASCII字符
9	DX	console	输出字符串
A	console	DX(包含字符串长度及内容)	输入字符串
4C	AL(常设为00H表示返回值为0)	/	返回DOS

返回 DOS

```
MOV AH, 4CH     ; AH = 4C, DOS 4C 号功能调用: 返回 DOS
MOV AL, 00H     ; AL = 给 DOS 的返回值
INT 21H         ; 21H 表示 DOS 功能调用
```

输入输出

输入一个 ASCII 字符: 1 号功能调用

```
MOV AH, 1           ; 1 号功能调用
INT 21H
MOV BYTE PTR X, AL  ; 结果存储在 AL 中
```

输出一个 ASCII 字符: 2 号功能调用

```
MOV DL, 'A'         ; 输出的字符在 DL 中
MOV AH, 2           ; 2 号功能调用
INT 21H             ; 输出的字符保存在 AL 中（不确定）
```

注意： `AL` 寄存器会改变

输入字符串： `0AH` 号功能调用

AH	功能	调用参数	返回参数
0A	键盘输入至缓冲区	DS:DX=缓冲区首 DS:[DX]=缓冲区最大容量	DS:[DX+1]=输入的字符数 DS:DX+2=字符串首

使用时需要先构造一个特定格式的缓冲区，定义输入的最大允许长度。使用时键入一行字符串并按回车键。

```
; ---- DATA SEGMENT PARA ----
; 构造一个特定格式的缓冲区
LEN     EQU 101          ; 允许输入字符个数(含回车)
BUFFER  DB  LEN-1        ; 预设最大长度(不含回车)
        DB  ?            ; 实际输入长度(不含回车)
        DB  LEN DUP(?)   ; 输入的串
; ---- DATA ENDS ----

LEA DX, BUFFER           ; 设定 DS:DX 为缓冲区首地址
MOV AH, 0AH              ; 0AH 号功能调用
INT 21H
```

输出字符串： 9 号功能调用

使用9号功能输出的字符串需要以'$'结尾

```
LEA DX, STR          ; 输出的字符串首地址在 DX 中
MOV AH, 9            ; 9 号功能调用
INT 21H
```

IO相关

输入不需要保护最后获取数据的寄存器，而输出需要保护需要打印数据的寄存器

从console读入一个十进制数字

将读入数据存储在AX，子程序中需要将BX和CX压栈，分别代表radix和integer cache，所以实际上这个程序通过改BX可以实现任意数值读入(十六进制本身有简单读入)

```
    ; read a decimal integer from console
GETINT PROC                                 ; usage: AX = getint()
    ; protect registers
                PUSH    BX
                PUSH    CX
    ;   CX = 0
    ;   do
    ;       AL = getchar()
    ;   while AL < '0' || AL > '9'
                MOV     CX, 0               ; use CX cache the number
```

```
    GETINT_LOOP_1:
                MOV     AH, 1
                INT     21H                     ; read char -> AL
    ; AL ⩾ '0' && AL ⩽ '9' -- break
    ; AL < '0' || AL > '9' -- loop
                MOV     AH, 0
                CMP     AL, '0'
                JB      GETINT_LOOP_1
                CMP     AL, '9'
                JA      GETINT_LOOP_1
    ; end of LOOP_1
    ;   do
    ;       AL -= '0'
    ;       CX = CX * 10 + AL
    ;       AL = getchar()
    ;   until AL < '0' || AL > '9'
    GETINT_LOOP_2:
    ; CX = CX * 10 + (AL - '0')
                SUB     AL, 30H             ; c -= '0'
                XCHG    AX, CX
                MOV     BX, 10
                MUL     BX                  ; AX *= 10
                ADD     AX, CX              ; AX += (c - '0')
                XCHG    AX, CX
    ; another getchar
                MOV     AH, 1
                INT     21H
                MOV     AH, 0
                CMP     AL, '0'
    ; AL < '0' || AL > '9' -- break
                JB      GETINT_RET
                CMP     AL, '9'
                JA      GETINT_RET
                JMP     GETINT_LOOP_2       ; loop
    GETINT_RET:
                MOV     AX, CX              ; set return value
    ; restore registers
                POP     CX
                POP     BX
                RET
GETINT ENDP
```

用十进制输出一个字的数据

将要输出的字数据存储在AX，子程序需要将AX，BX，CX，DX压栈，AX防止本身要输出的数字被破坏，BX为radix，DX为除以基数得到的商并利用DL完成DOS输出，CX记录AX内数据转化为对应进制后的位数，便于对DX进行出栈输出，所以这个程序可以实现对AX的任意进制输出

```
; print a decimal integer to console
PUTINT PROC                                 ; usage: putint AX
    ; protect registers
                PUSH    AX
```

```
                PUSH    BX
                PUSH    CX
                PUSH    DX
    ; if (AX == 0) putchar '0'
                CMP     AX, 0
                JZ      PUTINT_ZERO
    ;   do
    ;       DX, AX = AX % 10, AX / 10
    ;       CX++
    ;       push DX
    ;   while (AX != 0)
                MOV     CX, 0
    PUTINT_LOOP1:
                MOV     DX, 0
                MOV     BX, 10
                DIV     BX
                PUSH    DX
                INC     CX
                CMP     AX, 0
                JNZ     PUTINT_LOOP1
    ;   do
    ;       pop DX
    ;       putchar DX + '0'
    ;       CX--
    ;   while (CX > 0)
    PUTINT_LOOP2:
                POP     DX
                ADD     DL, 30H
                MOV     AH, 2
                INT     21H
                LOOP    PUTINT_LOOP2
                JMP     PUTINT_RET
    ; putchar '0'
    PUTINT_ZERO:
                MOV     AH, 2
                MOV     DL, '0'
                INT     21H
    PUTINT_RET:
    ; restore registers
                POP     DX
                POP     CX
                POP     BX
                POP     AX
                RET
PUTINT ENDP
```

用十进制输出一个双字的数据

将要输出的字数据存储在mem64@[BX]，CX为对应radix，AX用于提取mem64@[BX]中的字，DX存储商并利用DL完成DOS输出，利用BP和SP指针位置比较决定是否完成出栈

```
    ; Display 64-bit integer in decimal.
```

```
PUTINT64 PROC                                   ; print mem64@[BX]
    ; protect registers
                PUSH    AX
                PUSH    BX
                PUSH    CX
                PUSH    DX
                PUSH    BP                      ; store temp-result
to stack
    ; if [BX] == 0 : print 0
                MOV     AX, [BX]
                OR      AX, [BX+02H]
                OR      AX, [BX+04H]
                OR      AX, [BX+06H]
                CMP     AX, 0
                JNZ     PUTINT64_MAIN
    ; print zero
    PUTINT64_ZERO:
                MOV     AH, 2
                MOV     DL, 30H
                INT     21H
                JMP     PUTINT64_RET
    PUTINT64_MAIN:
    ; copy mem64@[BX] to stack
                SUB     SP, 16                  ; two 64-bit
                MOV     BP, SP
                MOV     AX, [BX+00H]
                MOV     [BP+08H], AX
                MOV     AX, [BX+02H]
                MOV     [BP+0AH], AX
                MOV     AX, [BX+04H]
                MOV     [BP+0CH], AX
                MOV     AX, [BX+06H]
                MOV     [BP+0EH], AX
                JMP     PUTINT64_LOOP1
    ; [BP] / 10, [BP] % 10 : per word, 4 times
    PUTINT64_LOOP1:
                MOV     CX, 0AH                 ; divisor: 10
                MOV     DX, 0                   ; higher part
(partial remainder)
                MOV     AX, [BP+0EH]
                DIV     CX                      ; DX, AX = AX % 10,
AX / 10
                MOV     [BP+06H], AX            ; partial-quotient
                MOV     AX, [BP+0CH]
                DIV     CX
                MOV     [BP+04H], AX
                MOV     AX, [BP+0AH]
                DIV     CX
                MOV     [BP+02H], AX
                MOV     AX, [BP+08H]
                DIV     CX
                MOV     [BP+00H], AX            ; DX is the remainder
                PUSH    DX                      ; follow BP
    ; copy BP[0:3] to BP[4:7]
```

```
                    MOV     AX, [BP+00H]
                    MOV     [BP+08H], AX
                    MOV     AX, [BP+02H]
                    MOV     [BP+0AH], AX
                    MOV     AX, [BP+04H]
                    MOV     [BP+0CH], AX
                    MOV     AX, [BP+06H]
                    MOV     [BP+0EH], AX
    ; if [BX] ≠ 0 LOOP
                    MOV     AX, [BP+08H]
                    OR      AX, [BP+0AH]
                    OR      AX, [BP+0CH]
                    OR      AX, [BP+0EH]
                    CMP     AX, 0
                    JNZ     PUTINT64_LOOP1
    ; end of LOOP1
    ; print each digit
    PUTINT64_LOOP2:
                    POP     DX
                    ADD     DL, 30H                         ; + '0'
                    MOV     AH, 2
                    INT     21H
                    CMP     SP, BP                          ; pop until stack-
under-BP empty
                    JNZ     PUTINT64_LOOP2
    ; end of LOOP2
    ; finished output.
                    ADD     SP, 16                          ; two 64-bit
    ; restore registers and return
    PUTINT64_RET:
                    POP     BP
                    POP     DX
                    POP     CX
                    POP     BX
                    POP     AX
                    RET
PUTINT64 ENDP
```

用十六进制输出一个字节的数据

将要输出的字节数据存储在AL，子程序需要保护CX，DX，SI和AX，AX防止输出数据被破坏，CX利用CL完成位运算，DX利用DL调用DOS完成输出，SI进行寄存器相对寻址获取DL的值

```
DATA SEGMENT PARA
    ; hexadecimal characters
    HEXCHAR   DB '0123456789ABCDEF'
DATA ENDS
; print a byte value in hexadecimal
PUTHEX8 PROC                                                ; print AL in
hexadecimal,for example, 0x7A
    ; protect registers
                    PUSH    CX
```

```
                    PUSH    DX
                    PUSH    SI
    ; print high digit
                    PUSH    AX
                    MOV     DH, 0
                    MOV     DL, AL
                    MOV     CL, 4
                    SHR     DL, CL                      ; AL >> 4
                    MOV     SI, DX
                    MOV     DL, [SI+HEXCHAR]            ; relative addressing,
[HEXCHAR + SI] is the same as [SI + HEXCHAR]
                    MOV     AH, 2
                    INT     21H                         ; putchar
    ; print low digit
                    POP     AX
                    PUSH    AX
                    MOV     DL, AL
                    AND     DL, 0FH                     ; low digit
                    MOV     SI, DX
                    MOV     DL, [SI+HEXCHAR]
                    MOV     AH, 2
                    INT     21H
    ; restore registers and return
                    POP     AX
                    POP     SI
                    POP     DX
                    POP     CX
                    RET
PUTHEX8 ENDP
```

以十六进制输出一个字的数据

将要输出的字数据存储在DX,需要调用PUTHEX8，保护AX和DX用于DOS输出

```
DATA SEGMENT PARA
    ; hexadecimal characters
    HEXCHAR   DB '0123456789ABCDEF'
DATA ENDS
; print 16-bit integer in hexadecimal
PUTHEX16 PROC                                           ; print DX
    ; protect registers
                    PUSH    AX
                    PUSH    DX
    ; print 4 bytes from high to low
                    MOV     AL, DH
                    CALL    PUTHEX8
                    MOV     AL, DL
                    CALL    PUTHEX8
    ; restore registers and return
                    POP     DX
                    POP     AX
                    RET
```

```
PUTHEX16 ENDP
```

以十六进制输出一个双字的数据

将要输出的字数据存储在mem64@[BX],需要调用PUTHEX8，保护AX和DX用于DOS输出

```
    ; Display 64-bit integer in hexadecimal
PUTHEX64 PROC                                          ; print mem64@[BX]
    ; protect registers
                PUSH    AX
                PUSH    BX
                PUSH    DX
    ; display 8 bytes from high to low
                MOV     AL, [BX+7]
                CALL    PUTHEX8
                MOV     AL, [BX+6]
                CALL    PUTHEX8
                MOV     AL, [BX+5]
                CALL    PUTHEX8
                MOV     AL, [BX+4]
                CALL    PUTHEX8
                MOV     AL, [BX+3]
                CALL    PUTHEX8
                MOV     AL, [BX+2]
                CALL    PUTHEX8
                MOV     AL, [BX+1]
                CALL    PUTHEX8
                MOV     AL, [BX]
                CALL    PUTHEX8
    ; restore registers and return
                POP     DX
                POP     BX
                POP     AX
                RET
PUTHEX64 ENDP
```

打印数组

AX和DX用于DOS输出，CX存储当前打印元素数组下标index，SI用于寄存器相对寻址获取元素

```
DATA SEGMENT PARA
    TABLE_LEN DW 16
    TABLE     DW
200,300,400,10,20,2137H,3191H,1,8,41H,40,42H,3321h,60,0FFFFH,2,3
DATA ENDS
PRINT_TABLE PROC                                       ; print `TABLE` with
`TABLE_LEN` before
                PUSH    AX
                PUSH    DX
                PUSH    CX
```

```
                    PUSH    SI
                    MOV     CX,0
                    MOV     SI,0
    PRINT_LOOP:
                    MOV     DX, [SI+TABLE]
                    CALL    PUTHEX16                    ; use PUTHEX16 to
print DX
    ; cx += 1
                    INC     CX
                    ADD     SI, 2
                    CMP     CX, TABLE_LEN
                    JZ      PRINT_LOOP_END
    ; putchar ' '
                    MOV     DL, 20H
                    MOV     AH, 2
                    INT     21H
    ; loop
                    JMP     PRINT_LOOP
    PRINT_LOOP_END:
    ; putchar '\n'
                    MOV     DL, 0AH
                    MOV     AH, 2
                    INT     21H
                    POP     SI
                    POP     CX
                    POP     DX
                    POP     AX
                    RET
PRINT_TABLE ENDP
```

字符串相关

指令	功能描述	操作大小	影响寄存器
LODSB	将源串 DS:[SI] 的一个字节取到 AL，根据 DF 修改 SI	字节	SI
LODSW	将源串 DS:[SI] 的一个字取到 AX，根据 DF 修改 SI	字	SI
STOSB	将 AL 中的一个字节存入目的串 ES:[DI]，并根据 DF 修改 DI	字节	DI
STOSW	将 AX 中的一个字存入目的串 ES:[DI]，并根据 DF 修改 DI	字	DI
MOVSB	将源串 DS:[SI] 的字节或字传送到目的串 ES:[DI]，并根据 DF 修改 SI 及 DI	字节	SI，DI
MOVSW	同 MOVSB，但传送的是字	字	SI，DI
CMPSB	比较源串 DS:[SI] 与目的串 ES:[DI] 的一个字节或字，根据 DF 修改 SI 及 DI	字节	标志寄存器 (CF，SF，ZF)
CMPSW	同 CMPSB，但比较的是字	字	标志寄存器 (CF，SF，ZF)
SCASB	在目的串 ES:[DI] 中扫描是否有 AL 指定的字节，根据 DF 修改 DI	字节	标志寄存器
SCASW	在目的串 ES:[DI] 中扫描是否有 AX 指定的字，根据 DF 修改 DI	字	标志寄存器

指令	等价指令	含义	执行条件	停止条件
REP		重复	CX 不为零	CX 为零
REPE	REPZ	相等时重复	CX 不为零，且 ZF 为 1	CX 为零 或 ZF 不为 1
REPNE	REPNZ	不相等时重复	CX 不为零，且 ZF 为 0	CX 为零 或 ZF 为 1

MEMSET

内存初始化(memset(BUFF, 0, LEN);),其中DI为BUFF, AL为0, CX为LEN

- 用 REP STOSB 指令将长度为 LEN 的缓冲区 BUFF 清零。

```
MEMSET PROC
        PUSH  ES                ; 修改 ES 前先保存

        PUSH  DS
        POP   ES                ; ES = DS
```

```
        MOV   DI, OFFSET BUFF ; DI = BUFF 首地址
        ; 也可以用 LEA DI, BUFF

        MOV   CX, LEN         ; CX = BUFF 长度
        CLD                   ; 设置方向为自增，DF=0
        MOV   AL, 0           ; AL = 要存入的字节
        REP   STOSB           ; 重复 CX 次执行 STOSB

        POP   ES              ; 恢复 ES
        RET
MEMSET ENDP
```

MEMCPY

字符串复制(memcpy(BUFF2, BUFF1, LEN);)

- 用 `REP MOVSB` 指令将缓冲区 `BUFF1` 内容传送到 `BUFF2`，长度为 `LEN`。

```
MEMCPY PROC
            PUSH  ES                          ; 修改 ES 前先保存

            PUSH  DS
            POP   ES                          ; ES = DS
            MOV   SI, OFFSET BUFF1            ; LEA SI, BUFF1
            MOV   DI, OFFSET BUFF2            ; LEA DI, BUFF2
            ; prepare: DS:[SI] at String1, ES:[DI] at String2
            MOV    CX, LEN
            CLD
            REP    MOVSB                      ; move string by byte
            POP   ES
            RET
MEMCPY ENDP
```

STRCMP

字符串比较(由于是相同则继续比较，故使用 `REPZ`)

- 情况1:长度不定长比较(strcmp(STRING1, STRING2);)

比较 `STRING1` 与 `STRING2` 按字典序排序的大小，改变 `CF` 后返回

将 `CL` 赋值为 `STRING1` 和 `STRING2` 中较短的长度作为比较次数

```
STRCMP PROC
    ; requires: string1, string2 from input, both should terminate with '$'
    ; provides: CMP flags ZF, CF, SF
    ; use REPZ CMPSB
    ; CX: min(LEN1, LEN2)
    ; prepare CX, use stack
                MOV    CH, 0
```

```
                MOV     CL, STR1_LEN                            ; assume CX =
str1_len
    ; len1: immediate addr, len2: direct addr
                MOV     AL, STR1_LEN
                CMP     AL, STR2_LEN
                JBE     STRCMP_MINLEN                           ; len1 ⩽
len2 skip
                MOV     CL, STR2_LEN                            ; if (len1 >
len2) CX=len2
    STRCMP_MINLEN:
                CLD
                REPZ    CMPSB
    ; now we have CX and ZF
    ; ZF: prefix equal, return CMP len1, len2
    ; NZ: returns cmp of character
                JNZ     STRCMP_END
                MOV     AL, STR1_LEN
                CMP     AL, STR2_LEN
    STRCMP_END:
                RET
STRCMP ENDP
```

- 情况2:长度定长比较(strncmp(STRING1, STRING2, LEN);)

比较 `STRING1` 与 `STRING2` 按字典序排序的大小，假定都是大写字母且长度都为 `LEN` 。

```
STRNCMP PROC
    PUSH  ES

    PUSH  DS
    POP   ES      ; ES = DS

    MOV   SI, OFFSET STRING1
    MOV   DI, OFFSET STRING2

    MOV   CX, LEN
    CLD
    REPZ  CMPSB
    ; while (CX ≠ 0 && DS:[SI] == ES:[DI]) SI++, DI++, CX--;

    JZ    EQUAL   ; 两字符串相等
    JA    GREATER ; STRING1 > STRING2
    JB    LESS    ; STRING1 < STRING2

    ; 后续处理
    EQUAL:
    ; ......

    GREATER:
    ; ......

    LESS:
    ; ......
```

```
    POP   ES
    RET
STRNCMP ENDP
```

上述代码还可以通过 `CX` 的值判断比较操作进行了多少次，如果 `CX=0` 则比较至最后了。

FIND

字符串查找子串(由于是相同则代表找到，不相同继续比较，故使用REPNZ)

- `REPNZ SCASB` 停下来后，如果 `ZF=1` 则 `AL=ES:[DI]`，通过 `CX` 的值可以看出 AL 在 `STRING1` 中的位置。如果 `ZF=0` 则已经遍历到最后仍未找到。

```
FIND PROC
    ; requires: AL - char to find, STACK - STRING and LEN
    ; provides: AX - count of AL in STRING
    ; use BP fetch STRING and LEN from stack
                MOV    BP, SP
                MOV    DI, [BP+04H]                          ; STRING
                MOV    CX, [BP+02H]                          ; LEN
                MOV    SI, 0                                 ; use SI as
counter
    FIND_LP:                                                 ; LOOP until
reach the end of string
                REPNZ  SCASB                                 ; scan
    ; stops either find or end
                JNZ    FIND_END                              ; end and not
found
                ADD    SI, 1                                 ; count++
                JMP    FIND_LP
    FIND_END:
    ; end of FIND_LP
                MOV    AX, SI                                ; return count
via AX
                RET    04H                                   ; return with
poping STRING and LEN,04H is used to pop 2 words
FIND ENDP
```

STRCAT

字符串拼接(strcat(dst, src);)

```
    ; concat str2 after str1 (both asciiz)
STRCAT PROC
    ; arg: head address of str1 and str2 (terminated with 0H)
              PUSH   BP
              MOV    BP, SP
              PUSH   SI
              PUSH   DI
    ; SS:[SP]: DI, SI, BP, IP, str2, str1, ...
```

```
    ; 1. scan the end of str1
              MOV    DI, [BP+6H]
              MOV    CX, MAX_LEN
              MOV    AL, 0H
              CLD
              REPNZ  SCASB
    ; now DI is after '\0' of str1
              DEC    DI                                  ; [DI] is end of
str1
              MOV    SI, [BP+4H]                         ; SI at str2
    ; 2. copy str2 to after str1
    STRCAT_LP:
              CLD
              LODSB                                      ; AL = *(str2++)
              STOSB                                      ; *(str1++) = AL
              CMP    AL, 0H                              ; if (AL ≠ 0)
loop
              JNZ    STRCAT_LP
    ; after copy
    ; return
              POP    DI
              POP    SI
              POP    BP
              RET    4H                                  ; pop str2, str1
STRCAT ENDP
```

TOUPPER

字母转大写

```
int CX = LEN;
char *SI = BUFF1, *DI = *SI;
while (cx--) {
    char AL = *SI;
    if (AL ⩾ 'a' && AL ⩽ 'z')
    AL -= 0x20;
    *DI = AL;
    SI++, DI++;
}
```

- 将长度为 `LEN` 的缓冲区 `BUFF1` 中的小写字母变成大写。

注意小写字母和对应大写字母相差20H

```
TOUPPER PROC
    PUSH  ES

    PUSH  DS
    POP   ES                ; ES = DS

    MOV   SI, OFFSET BUFF1  ; LEA SI, BUFF1
```

```
    MOV   DI, SI

    MOV   CX, LEN
    CLD

    LP1:
    LODSB                   ; AL ← DS:[SI], SI++
    CMP   AL, 'a'
    JB    CONTINUE
    CMP   AL, 'z'
    JA    CONTINUE
    SUB   AL, 20H           ; 小写变大写

    CONTINUE:
    STOSB                   ; ES:[DI] ← AL, SI--
    LOOP  LP1

    POP   ES
TOUPPER ENDP
```

algorithm相关

冒泡排序

> 注意LOOP默认减少CX即通过CX控制循环次数即可

```
    ; bubble sort
BUBBLE_SORT PROC                                    ; sort `TABLE` in
memory with `TABLE_LEN` before.
    LP1:
                MOV    BX, 1                        ; flag
                MOV    CX, TABLE_LEN
                DEC    CX                           ; loop TABLE_LEN times
                LEA    SI, TABLE                    ; i = 0
    LP2:
                MOV    AX, [SI]                     ; a[i], a[i + 1]
                CMP    AX, [SI+2]
                JBE    CONTINUE                     ; if a[i] > a[i + 1]
swap
                XCHG   AX, [SI+2]                   ; swap
                MOV    [SI], AX
                MOV    BX, 0                        ; swap happen in a
pass
    CONTINUE:
                ADD    SI, 2                        ; i++
                LOOP   LP2
    ; end of LP2
                CMP    BX, 1                        ; if (not swapped)
break
                JZ     EXIT
                JMP    SHORT LP1                    ; loop LP1
```

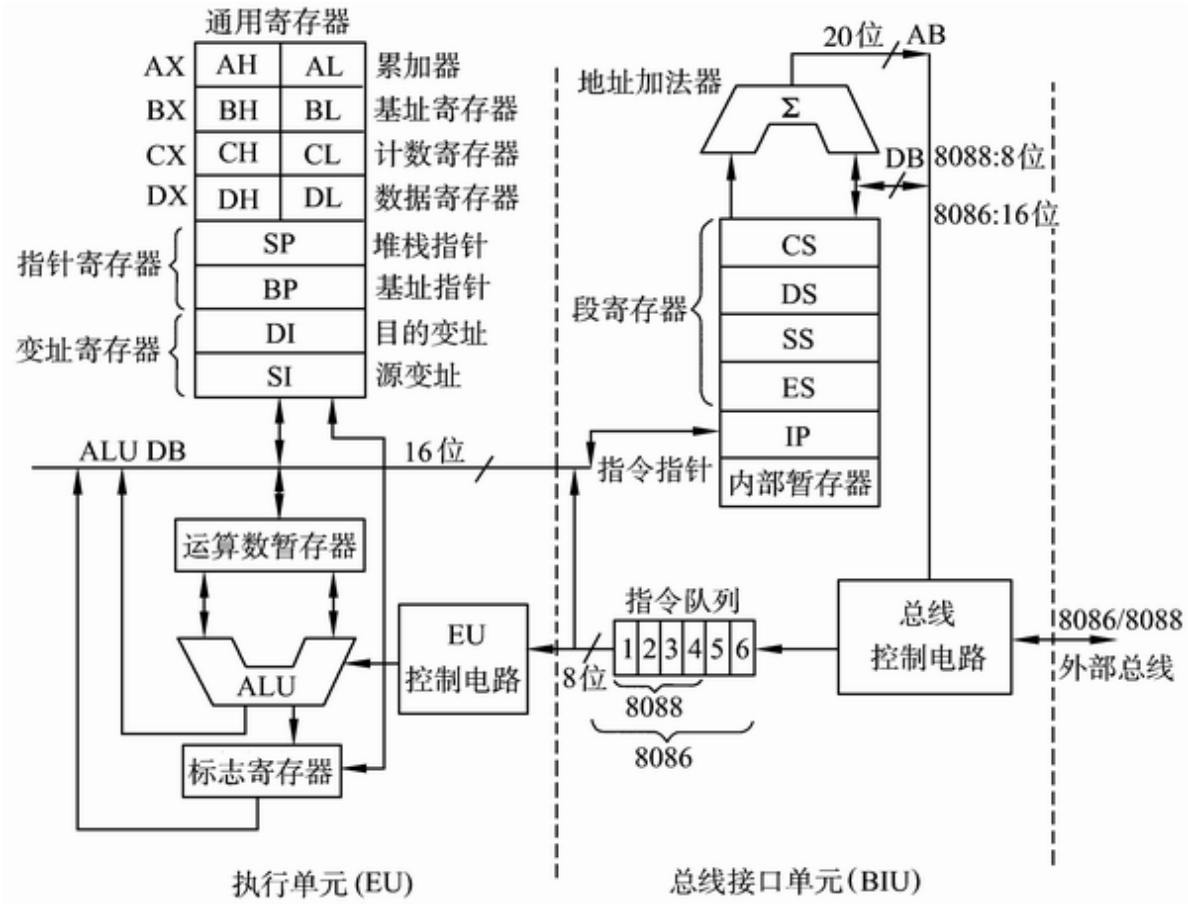
```
    ; end of LP1
    EXIT:
                    RET
BUBBLE_SORT ENDP
```

数制

- MASM 中十六进制数的表示方法：以 `h` 或 `H` 结尾，如果以字母 `A-F` / `a-f` 开头还需加前缀 `0` 以避免和标识符(变量/标签/寄存器)混淆。
- 例如：`0AH`，`0FFFFFh`，`10h`，分别对应 C 语言中的 `0xa`，`0xFFFFF`，`0x10`。其中 `0AH` 和 `0FFFFFh` 的前缀 `0` 是为了区分寄存器名称 `AH` 与标识符名称 `FFFFFh`。

8086 机器

基本设定



机器字长：16 位（ALU，寄存器等的位宽）

地址线宽：20 位（寻址空间为 1MB）

数据总线：16 位（8086），8 位（8088）

主要字符的ASCII码

表 1.1　常见字符的 ASCII 编码(十六进制值)

字符	ASCII 码	字符	ASCII 码	字符	ASCII 码	字符	ASCII 码
NUL	00	4	34	M	4D	f	66
BEL	07	5	35	N	4E	g	67
LF	0A	6	36	O	4F	h	68
FF	0C	7	37	P	50	i	69
CR	0D	8	38	Q	51	j	6A
SP	20	9	39	R	52	k	6B
!	21	:	3A	S	53	l	6C
"	22	;	3B	T	54	m	6D
#	23	<	3C	U	55	n	6E
$	24	=	3D	V	56	o	6F
%	25	>	3E	W	57	p	70
&	26	?	3F	X	58	q	71
'	27	@	40	Y	59	r	72
(	28	A	41	Z	5A	s	73
)	29	B	42	[	5B	t	74
*	2A	C	43	\	5C	u	75
+	2B	D	44	]	5D	v	76
,	2C	E	45	↑	5E	w	77
-	2D	F	46	←	5F	x	78
.	2E	G	47	'	60	y	79
/	2F	H	48	a	61	z	7A
0	30	I	49	b	62	{	7B
1	31	J	4A	c	63	|	7C
2	32	K	4B	d	64	}	7D
3	33	L	4C	e	65	~	7E

以下为十六进制表示，**注意十六进制输出时从9到A需要加7，从A到a需要加20H**

- 数字 0-9：30-39
- 大写字母 A-Z：41-5A
- 小写字母 a-z：61-7A
- 空格：20
- 换行：0A

寄存器

- 通用寄存器：共 4 个，每个寄存器 16 位（ `*X` ），分为两个 8 位（ `*H` ， `*L` ）
  - `AX (AH, AL)` ：累加器（Add）
  - `BX (BH, BL)` ：基址寄存器（Base）
  - `CX (CH, CL)` ：计数寄存器（Count）
  - `DX (DH, DL)` ：数据寄存器（Data）
- 指针寄存器：2 个

  BP经常用于存取非栈顶的数据或和SP进行位置比较决定是否继续出栈

  - `SP` ：16 位堆栈指针
  - `BP` ：16 位基址指针

- 变址寄存器（字符串指针）
  - `SI`：源串（Source）
  - `DI`：目的串（Destination）
- 段寄存器：4 个
  - `CS`：代码段（Code）
  - `DS`：数据段（Data）
  - `SS`：堆栈段（Stack）
  - `ES`：附加段（Extra）
- 指令指针
  - `IP`，相当于 RISC 中的 `PC`
- 标志寄存器 `PSW`

`SS:SP` 组成堆栈（`SS` 是堆栈基地址，`SP` 是栈顶相对基地址的偏移）

`CS:IP` 组成当前可执行点（`CS` 是代码段基地址，`IP` 是当前指令相对基地址的偏移）

表 2.1　8 个通用寄存器的一般用途

寄存器	主要用途	是否隐含或特定使用
AX	① 在乘法中，存放乘数和乘积 ② 在除法中，存放被除数、商、余数 ③ 在串操作指令中存放操作数(AL 或 AX) ④ 在 IN/OUT 指令中用作数据寄存器 ⑤ 在功能调用中存放功能号(AH) ⑥ 一般运算中存放操作数或结果	隐含使用 隐含使用 隐含使用 特定使用 特定使用
BX	① 间接寻址或基址加变址寻址时的基址寄存器 ② 一般运算中存放操作数或结果	特定使用
CX	① 在循环指令中，作循环次数计数器 ② 在移位指令中，作移位次数计数器	隐含使用 特定使用
DX	① 在 16 位×16 位乘法中用于存放乘积高位 ② 在 32 位÷16 位除法中用于存放被除数高位及余数 ③ 在 IN,OUT 指令中用于间址寄存器	隐含使用 隐含使用 隐含使用
SP	在堆栈操作中用作堆栈指针	隐含使用
BP	① 在相对于堆栈段的基址加变址寻址时，用作基址寄存器 ② 在利用堆栈来向子程序传递参数时，用于基址寄存器	特定使用
SI	① 在串操作指令中，用作源变址寄存器 ② 在基址加变址寻址时，用作变址寄存器	隐含使用 特定使用
DI	① 在串操作指令中，用作目的地址寄存器 ② 在基址加变址寻址时，用作变址寄存器	隐含使用 特定使用

标志寄存器

```
 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |OF|DF|IF|TF|SF|ZF|  |AF|  |PF|  |CF|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

标志含义：

- `OF` 溢出标志
- `DF` 方向标志（地址递增/递减)[CLD/STD]
- `IF` 中断标志
- `TF` 陷阱标志
- `SF` 符号标志
- `ZF` 结果为零标志
- `AF` 辅助进位标志
- `PF` 奇偶标志
- `CF` 进位标志[CLC/STC]

表 2.2　在 DEBUG 中显示的标志寄存器符号的含义

标志寄存器名	标为 1 时	标志为 0 时
进位标志 CF	CY(有进位)	NC(无进位)
奇偶标志 PF	PE(偶数个 1)	OP(奇数个 1)
辅助进位标志 AF	AC(有进位)	NA(无进位)
零标志 ZF	ZR(结果为 0)	NZ(结果不为 0)
符号标志 SF	NG(结果为负)	PL(结果为正)
中断标志 IF	EI(允许中断)	DI(禁止中断)
方向标志 DF	DN(减量)	UP(增量)
溢出标志 OF	OV(溢出)	NV(未溢出)

存储器

8086 地址线共有 20 位，可寻址空间为 1MB。

总体结构

```
+------------+ ---     ---     ---
|   00000h   |  | Byte |       |
+------------+ ---      | Word |
|   00001h   |          |       |
+------------+         ---      | DoubleWord
|   00002h   |                  |
+------------+                  |
|   00003h   |                  |
+------------+                 ---
|            |
|   ......   |
|            |
+------------+
|   FFFFFh   | 1MB
```

```
+------------+
```

内存的基本单元为一个**字节**（Byte，8 位），按线性顺序存放；两个字节组合为一个**字**（Word，16 位），两个字组合成为一个**双字**（Double Word，32 位）。

内存中可以任意组合存放字节、字、双字，**无需字对齐**（与 MIPS 不同）。

在内存中存放一个字时，低字节在前，高字节在后；存放双字时，低字在前，高字在后（即：按字节小端序存储）

例：在 `00000h` 地址存放一个双字 `12345678h`

```
  addr : data
00000h : 78h
00001h : 56h
00002h : 34h
00003h : 12h
```

字节、字、双字的存取

例题：`CL` 中有字节 `'A'`，`BX` 中有字 `2000h`，`DX:AX` 中有双字 `12345678h`。

- 依次将 `CL`，`BX`，`DX:AX` 中的内容按照字节、字、双字的结构存入地址 `30000h` 处。

```
addr : data
    30000h : 41h     ; 'A'
    30001h : 00h     ; 2000h
    30002h : 20h     ;
    30003h : 78h     ; 12345678h
    30004h : 56h     ;
    30005h : 34h     ;
    30006h : 12h     ;
```

- 将内存 `30000h` 处的一个字取出来送入 `BX` 中，将 `30002` 处的一个双字送入 `DX:AX` 中

```
BX = 0041h [BH = 00h, BL = 41h]
DX = 3456h [DH = 34h, DL = 56h]
AX = 7820h [AH = 78h, AL = 20h]
DX:AX = 34567820h
```

逻辑地址和物理地址

用 16 位的地址寄存器来表示地址，最多可寻址 64KB 空间。要表示 20 位地址，需要对内存进行**分段**，用一个寄存器（段寄存器）表示段地址，用另一个寄存器（指针寄存器）表示段内地址偏移。

将 1MB 内存分段，每段最大 64KB。

- 物理地址：20 位二进制，与内存单元（字节）一一对应
- 逻辑地址：用**段地址**和**偏移值**组合表示内存地址，常写作 `段地址:偏移值` 的形式
- 物理地址 = 段地址 x 16D(10h) + 偏移地址，一个物理地址可能有多个逻辑地址的组合。

> 即把段地址左移一个十六进制位再加上偏移地址

- 典型的程序在内存中执行时，一般都有**代码段**，**数据段**，**堆栈段**，其段地址分别用 `CS` ， `DS` ， `SS` 来存放。

堆栈的组成和操作

堆栈：由 `SS` 和 `SP` 确定的一块特殊区域，严格按照**先进后出**的方式工作。增长方向为从高地址向低地址（与 MIPS 相似）

例： `SS = 2000h, SP = 0100h`

- 堆栈区域为 `2000h:0000h` 至 `2000h:00FFh`
- 栈顶指针值为 `0100h` （注意栈顶指针值本身不属于堆栈）
- 压栈操作 `PUSH op`
  1. `SP = SP - 2`
  2. `op -> SS:[SP]`
- 出栈操作 `POP op`
  1. `op <- SS:[SP]`
  2. `SP = SP + 2`

指令格式与寻址方式

指令格式

Intel 8086/8088 基本指令格式：

```
op dst, src  ; 由 "源" 至 "目的", 结果在目的操作数中
```

（分号 `;` 以后的内容为行注释）

指令可以有 2 个，1 个或 0 个操作数：

- `op op2, op1`
- `op op1`
- `op`

指令编码由 1-7 个字节组成，为变长编码。编码要素：

- 操作码字节
- 寻址方式字节
- 段超越字节
- 操作数

寻址方式

与数据有关：

寻址方式	描述	示例
立即寻址	指令中直接给出操作数，操作数大小通常与指令中操作码所指定的一致。	`MOV AX, 1234H` 将立即数0x1234移动到AX寄存器
寄存器寻址	操作数直接存储在寄存器中，指令中指定寄存器名。	`ADD AX, BX` 将BX寄存器的内容加到AX寄存器
直接寻址	操作数的地址直接给出，指令中包含操作数的完整地址。	`MOV AX, [1000H]` 将地址0x1000处的值移动到AX寄存器
寄存器间接寻址	操作数的地址存储在寄存器中，指令中指定寄存器名，实际操作的是该寄存器中的地址所指向的内存单元。	`MOV AX, [BX]` 将BX寄存器指向的内存地址处的值移动到AX寄存器
寄存器相对寻址	基址寄存器中存储着一个地址，指令中给出的是一个偏移量，实际操作的地址是基址寄存器中的地址加上偏移量。	`MOV AX, [BX + 5]` 将BX寄存器的值加上5后得到的地址处的值移动到AX寄存器
基址变址寻址	结合基址寄存器和变址寄存器，形成最终的操作数地址，通常是基址寄存器的值加上变址寄存器的值再加上指令中的偏移量。	`MOV AX, [BX + SI + 10h]` 将BX寄存器的值加上SI寄存器的值再加上16进制数10h后得到的地址处的值移动到AX寄存器

与转移/调用指令有关：

寻址方式	描述	示例
段内直接寻址	直接使用段内偏移量进行跳转或调用，适用于同一数据段内的跳转。	`JMP p1` 跳转到 `label p1` 中存储的偏移地址继续执行
段内间接寻址	通过寄存器间接访问段内地址进行跳转或调用，适用于间接访问地址的情况。	`JMP [BX]` 跳转到BX寄存器指向的内存地址继续执行
段间直接寻址	使用段地址和偏移地址进行跳转或调用，适用于不同段间的跳转。	`JMP FAR PTR p2` 跳转到 `label p2` 地址继续执行
段间间接寻址	通过段寄存器和间接寻址方式，结合段选择器和偏移量进行跳转或调用。	`CALL DWORD PTR [BX]` 调用BX寄存器指向的远指针地址

与 IO 有关：

- 直接 I/O 端口寻址
- 寄存器 DX 间接寻址

与数据有关的寻址方式（6 种）

（目的，源）操作数可以来自：

- 寄存器：8 个通用寄存器，4 个段寄存器，1 个标志寄存器
- 立即数：指令本身给出的立即数（常量）
- 内存单元：直接寻址/间接寻址
- 定义操作数：
  - `EQU`：常量
  - `DB`：字节
  - `DW`：字
  - `DD`：双字

寻址方式：

- 立即寻址：指令操作数包含在指令中，为一个常量或常数
- 寄存器寻址：指令操作数为 CPU 的寄存器
- 直接寻址：操作数偏移地址 `EA` 在指令中给出，如**变量名**
- 寄存器间接寻址：操作数地址 `EA` 位于**间指**寄存器（`BX`，`BP`，`SI`，`DI`）中
- 寄存器相对寻址：操作数地址 `EA` 由间指寄存器 + 8 位或 16 位的常量组成
- 基址变址寻址：操作数地址 `EA` 为一个基址寄存器和一个变址寄存器之和

掌握和理解寻址要点：

1. 寄存器的使用规则
2. 类型匹配
3. 数据通路
4. 操作的是"内容"还是"地址"(指针)

立即寻址

指令所需操作数直接包含在指令代码中，可以是一个**常量**（由 `EQU` 定义）或者一个**常数**，成为**立即数。**

立即数可以是 8 位或 16 位，需要看与之对应的另一个操作数的类型（二者需要匹配）

```
VALUE EQU 512       ; 定义一个常量，名称为 VALUE，值为 512
MOV AL, 05H         ; AL = 05H
MOV AL, 00000101b   ; AL = 05H，b 表示二进制
MOV AX, 512         ; AX = 0200H（512 的十六进制）
MOV AX, 512         ; AX = 0200H（VALUE 是常量，值是 512）
```

错误示例：

```
MOV AL, 100H        ; 100H 超出了 1 字节的范围
MOV BL, VALUE       ; VALUE = 512，超出了 1 字节的范围
MOV AX, 10000H      ; 10000H 超出了 16 位（一个字）的范围
```

寄存器寻址

指令中所需的操作数是 CPU 的某个寄存器，取操作数完全在 CPU 内部进行，不需要访存。

- 对于 8 位操作数，寄存器可以是 `AH`，`AL`，`BH`，`BL`，`CH`，`CL`，`DH`，`DL` 中的一个。
- 对于 16 位操作数，寄存器可以是 `AX`，`BX`，`CX`，`DX`，`SP`，`BP`，`SI`，`DI` 及 `CS`，`DS`，`SS`，`ES` 的任何一个（没有 `IP`）

寄存器寻址方式示例：

```
MOV AX, BX      ; 源操作数和目的操作数都是寄存器寻址
MOV AX, 1234H   ; 目的操作数是寄存器寻址
ADD X, AX       ; 源操作数是寄存器寻址
PUSH DS         ; 源操作数是寄存器寻址
```

当使用 `CS`，`DS`，`SS`，`ES` 段寄存器时，必须遵循数据通路要求。

操作数中的寄存器可能是隐含的寄存器（没有明确出现在指令的源或目的操作数中），例如：

```
PUSHF           ; PSW（标志寄存器）作为源操作数，是寄存器寻址方式
STD             ; 设置 DF = 1，目的操作数是寄存器寻址方式
```

直接寻址

操作数的偏移地址直接在指令中给出。例如：

```
MOV AX, [2000H]     ; 源操作数 [2000H] 为直接寻址
; 相当于 AX = *(uint16_t*)(0x2000)
MOV AX, [ARRAY]     ; ARRAY定义在DATA段，为直接寻址
```

以上示例中源操作数 `[2000H]` 是直接寻址。（加 `[]` 表示其内部的 `2000H` 是地址，如不加则是立即寻址）

如没有段超越，通常以这种方式直接寻址去操作数都是相对数据段 `DS` 的。

如使用变量（符号）定义内存中的单元，在指令中直接使用符号也是直接寻址，虽然操作数中并未直接出现地址，但汇编语言程序经过汇编器后，汇编器计算出符号的偏移值并进行替换。例如：

```
; 以下两行为伪指令，定义了两个变量
x DW ?      ; 定义一个字变量(DW)，? 表示未指定初始值
c DB 'A'    ; 定义一个字节变量，初始值为 41H
; 以下指令为直接寻址
MOV AX, x   ; 将变量 x 的字存入 AX 寄存器
MOV AL, c   ; 将变量 c 的字节存入 AL 寄存器
; 以下指令也是直接寻址
MOV AX, x+1 ; 将内存 x+1 单元的字存入 AX 寄存器
```

寄存器间接寻址

操作数的有效地址 EA 不位于指令中，而是位于**基址寄存器** `BX` ， `BP` 或**变址寄存器** `SI` ， `DI` 中（不能是其他寄存器）。因为地址值未在指令中直接指出，而是通过一个寄存器来指明，因此称为间接寻址。效果上，这个寄存器相当于一个地址指针。

例如下面指令的源操作数的寻址方式都是间接寻址：

```
MOV AX, [BX]    ; 内存操作数的偏移地址位于 BX 中，在 DS 段内
MOV BH, [BP]    ; 内存操作数的偏移地址位于 BP 中，在 SS 段内
MOV CX, [SI]    ; 内存操作数的偏移地址位于 SI 中，在 DS 段内
MOV DL, [DI]    ; 内存操作数的偏移地址位于 DI 中，在 DS 段内。
```

错误示例：

```
MOV AX, [DX]    ; DX 不能作为间接寻址寄存器
MOV DL, [BL]    ; 只能用完整的 BX，不能用 BL 作为间接寻址寄存器，因为偏移值是 16 位
```

隐含段规则

使用以上 4 个寄存器进行间接寻址时，如果未显式指定段寄存器，则 `BX` ， `SI` ， `DI` 是相对 `DS` 段的偏移地址，而 `BP` 是相对 `SS` 段的偏移地址。

上述四条指令分别与下面四条指令等价：

> 由于BP经常与SP配合使用，所以不难理解BP是相对于SS的

```
MOV AX, DS:[BX]
MOV BH, SS:[BP]
MOV CX, DS:[SI]
MOV DL, DS:[DI]
```

注意： 堆栈指针 `SP` 不可以用来间接寻址！

段超越

如果寻址是不用寄存器默认隐含的段，而是显式地指定一个段寄存器，则称为**段超越**。例如：

```
MOV AX, SS:[BX] ; BX 默认相对段 DS，此处指定用段 SS 代替默认的 DS
MOV CS:[BP], DX ; 指定段 CS 代替 BP 寄存器默认的 SS，该指令会修改代码段，有一定危险性
```

寄存器相对寻址

操作数的有效地址 EA 是一个基址寄存器或变址寄存器的内容和指令中指定的 8 位和 16 位位移量之和。

即：EA = 间址寄存器的值 + 8 位或 16 位常量

也就是在间接寻址的基础上增加了一个常量（间接寻址 + 相对寻址）。

可用来寻址的寄存器与隐含段规则同间接寻址， `BX` ， `SI` ， `DI` 寄存器寻址的默认段是数据段 `DS` ， `BP` 寄存器寻址的默认段是堆栈段 `SS` 。

寄存器相对寻址示例(以下指令中的源操作数)：

```
MOV AX, [SI+10H]    ; SI 的值加 10H 形成偏移地址，在 DS 段内寻址
```

其中 `[SI+10H]` 也可以写成 `10H[SI]`，即上面的指令和下面的指令等价：

```
MOV AX, 10H[SI]
```

与直接寻址一样，相对寻址的 16 位偏移量也可以是个符号名或变量名。因为符号名和变量名在段内的位置(偏移值)是固定的，所以作用等同于常量。

采用符号名进行相对寻址的示例：

```
MOV AX, ARRAY[SI]   ; EA = SI 的值 + ARRAY 相对 DS 的偏移值
MOV TABLE[DI], AL   ; EA = DI 的值 + TABLE 相对 DS 的偏移值
MOV TABLE[DI+1], AL ; EA = DI 的值 + TABLE 的偏移值 + 1
```

基址变址寻址

操作数的有效地址 EA 等于一个基址寄存器和一个变址寄存器的内容之和。

显著特点：两个寄存器均出现在指令中。

> 基址寄存器为 `BX` 或 `BP`，变址寄存器为 `SI` 或 `DI`。
>
> 如果基址寄存器为 `BX`，则缺省段寄存器为 `DS`；如果基址寄存器为 `BP`，则缺省段寄存器为 `SS`。

示例：

```
MOV AX, [BX][SI]    ; 源操作数 EA = BX + SI，段为 DS
MOV AX, [BX+SI]     ; 等同上一条指令
MOV ES:[BX+SI], AL  ; 目的操作数 EA = BX + SI，段为 ES（采用了段超越）
MOV [BP+DI], AX     ; 目的操作数 EA = BP + DI，段为 SS
```

可以将基址变址寻址方式理解为寄存器相对寻址方式加上一个变址寄存器。例如：

```
MOV AX, [BX+SI+200] ; 源操作数的 EA = BX + SI + 200，段为 DS
MOV ARRAY[BP + SI], AX  ; 目的操作数 EA = BP + SI + ARRAY 的偏移量，段为 SS
```

注意：基址变址寻址方式的基址寄存器**只能为** `BX` **或** `BP`，变址寄存器**只能是** `SI` **或** `DI`。

错误示例：

```
MOV [BX+CX], AX     ; CX 不能作为变址寄存器
MOV [BX+BP], AX     ; BP 只能用作基址寄存器，不能用于变址寄存器
MOV [BX+DI], ARRAY  ; 如果 ARRAY 为变量，则源和目的操作数都在内存中，不合法
```

最后一条指令如果 ARRAY 是变量，则源操作数是直接寻址，源和目的操作数不能都在内存中（此部分详见**数据通路**），但 ARRAY 如果是立即数，则源操作数是立即寻址，没问题。

与转移地址有关的寻址方式

与转移地址有关的寻址方式主要运用于转移指令 `JMP` 和过程调用指令 `CALL`，寻址方式共有四种：

- 段内直接寻址
- 段内间接寻址
- 段间直接寻址
- 段间间接寻址

标号与过程名

标号示例：

```
...
l1: MOV AX, ARRAY[SI]
...
```

上例中 `l1` 是一个标号，后面跟有一个冒号，一般位于一条指令的前面。标号的作用与变量名类似，确定了标号后的指令在代码段中的偏移地址。

过程名示例：

```
    ...
p1  PROC near
    ...
    RET
p1  ENDP
```

或

```
    ...
p2  PROC far
    ...
    RET
p2  ENDP
```

过程位于程序代码中，上述两个例子的 `p1` 和 `p2` 是过程名，确定了该过程第一条指令在代码段中的偏移值。其中 `p2` 还同时指出了它所处的 `CS` 段值（用 `far` 表示）

段内直接寻址

要转向（由 `JMP`，条件转移，`CALL` 等）指令实际的有效地址是当前 `IP` 寄存器的内容和指令中指定的 8 位或 16 位位移量之和。

在定义了前面的标号 `l1` 或子程序名 `p1`，`p2` 后，段内直接寻址的示例：

```
JMP l1   ; 转移至标号 l1 处
CALL p1 ; 先保存 CALL 下一条指令的偏移地址至堆栈中，然后转移至 p1 处
```

与操作数的直接寻址方式不同的是，上述指令在汇编后，指令的机器码不会直接出现 `l1` 或 `p1` 的偏移地址，而是相对于当前 IP 的位移量，是一种相对寻址。

根据位移量是 8 位还是 16 位，可以加 `SHORT` 和 `NEAR PTR` 操作符，如下例所示：

```
JMP SHORT l1    ; l1 与当前 IP 的位移量是一个 8 位值
JMP NEAR PTR l1 ; l1 与当前 IP 的位移量是一个 16 位值
```

对于**条件转移指令，只能是 8 位位移量**，省略 `SHORT` 操作符。

如果 JMP 指令省略了 `NEAR PTR` 或 `SHORT` 操作符，则使用 16 位位移量。使用 8 位位移量的转移称为短跳转。

段内间接寻址

转向的有效地址是一个**寄存器**或**存储单元**的内容。

- 寄存器：可以是 `AX` , `BX` , `CX` , `DX` , `SI` , `DI` , `BP` , `SP` 中的任何一个。
- 存储单元：位于数据段中，可以使用四种内存寻址方式中的任何一种。

在转移指令中使用寄存器间接寻址时，寄存器保存的是相对**代码段** `CS` 而不是数据段的偏移值。（与数据相关的寻址不同！）

示例：用寄存器 `AX` 作为间接寻址寄存器

```
MOV AX, OFFSET p1   ; 获取 p1 过程在代码段内的偏移值
CALL AX
```

上例中 `AX` 也可以用 `BX` , `CX` , `DX` , `BP` , `SI` , `DI` 来代替，且都是相对于代码段的。使用 `SP` 在语法上也允许，但逻辑上通常不这样使用。

注意段内间接寻址和数据寄存器间接寻址的差别：

```
MOV AX, [BX]    ; 数据寄存器间接寻址
JMP BX          ; 转移指令的段内间接寻址
```

数据寻址的寄存器间接方式中， `[BX]` 有方括号，相对于数据段寻址。

当间接寻址使用内存单元存放时，要转移的偏移地址位于数据段的某个位置，而要转移至的位置则位于代码段中。

在下列指令中，转移地址同样使用的是段内间接转移：

```
MOV AX, OFFSET p1   ; 获取过程 p1 的偏移地址，存入 AX
MOV ADD1, AX        ; 将 AX 内容送入数据段内的 ADD1 处
CALL ADD1           ; 转移至 ADD1 中存放的偏移地址处
MOV BX, OFFSET ADD1 ; 获取数据 ADD1 的偏移地址，存入 BX
CALL [BX]           ; 转移地址的偏移地址位于数据段中，通过 BX 间接寻址获取
```

对于 `CALL ADD1` 指令，当 `ADD1` 是数据段中的一个地址而不是一个过程名/标号时，获取转移地址的方式是间接的，而不是直接的。它先从数据段 `ADD1` 确定的偏移地址中取出要转移去执行的地方的偏移值，然后再转移到代码段的此地址中。所以，当 `ADD1` 为变量名， `p1` 为过程名时，如下两条指令的差别是很大的：

```
CALL p1
CALL ADD1
```

前者是段（CS）内直接转移，后者是段内间接转移，其转移地址存放在数据段的 ADD1 处。

如果假设 `p1` 在代码段内的偏移值为 `000AH`，`ADD1` 在数据段内的偏移值为 `000AH`，则上述两条指令在 DEBUG 下变为：

```
CALL 000A   ; 直接转移至代码段 00A 处
CALL [000A] ; 转移至代码段某处，其偏移地址位于数据段的 00A 处
```

对于 `CALL BX` 指令，按照同样的道理，其转移至的代码段的偏移地址不是在 `BX` 中，而是在数据段的某个位置，这个位置由 `BX` 指出。`[BX]` 是普通的数据寄存器间接寻址，获取的数据单元的内容才是要转移至代码段内的偏移地址。所以如下两条指令虽然都是段内间接转移，但转移地址是不一样的：

```
CALL BX     ; 转移到的偏移地址位于 BX 中
CALL [BX]   ; 转移到的偏移地址在数据段某单元中，此单元通过 BX 间接寻址得到
```

段间直接寻址

与段内直接寻址不同的是，段间直接寻址在指令中给出了要转移至（由 `JMP` 和 `CALL` 完成）的地址的代码段和偏移值内容。例如，如果 `p2` 为由 `FAR` 属性定义的过程，则如下指令为段间直接转移：

```
CALL    FAR PTR p2
```

要转移至的标号或过程名必须具备 `FAR` 属性。

段间间接寻址

类似于段内间接寻址，但间接寻址时不能将要转移至的地址直接放入寄存器，而必须放入内存单元中，且是一个**双字**。格式如下：

```
JMP     DWORD PTR [BX+INTERS]
```

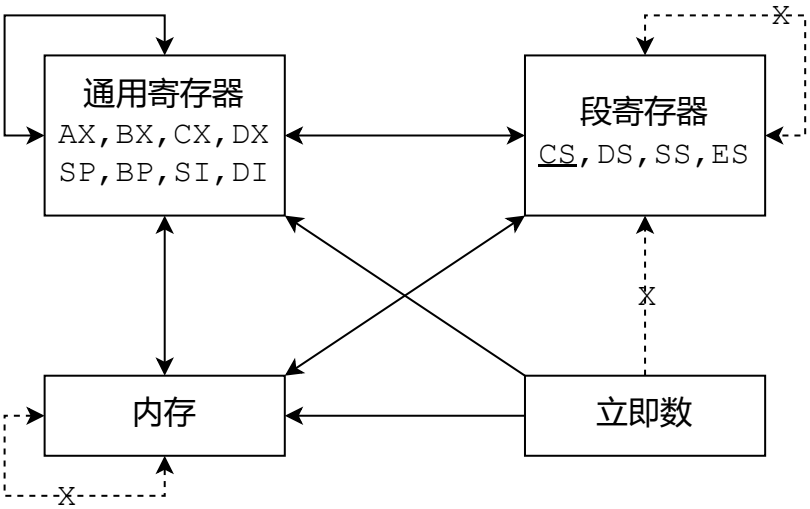
其中 `[BX+INTERS]` 为数据的寄存器相对寻址方式，`DWORD PTR` 后可以是除立即寻址和寄存器寻址以外（也就是内存寻址）的任何一种方式。

内存单元中的转移地址是一个双字，低位在前高位在后。转移后，低位字变成 `IP`，高位字变成 `CS`。

指令系统

数据通路

8086 CPU 数据通路图示：



图中有 4 种实体，实体之间有 12 条 "可能" 的数据通路。其中 9 条实心箭头，对应 **9 条数据通路。**另有 3 条带 `X` 号的虚线箭头不是有效的数据通路。

根据数据通路定义，以下的九种 `MOV` 指令是合法的（其中 `ac` 为立即数，`reg` 为通用寄存器，`segreg` 为段寄存器，`mem` 为内存）：

```
MOV reg, ac
MOV reg, reg
MOV reg, segreg
MOV reg, mem
MOV mem, ac
MOV mem, reg
MOV mem, segreg
MOV segreg, reg
MOV segreg, mem
```

注意：使用段寄存器作为目的操作数时，不允许用 `CS`（修改 `CS` 只能通过段间跳转指令）。

段超越

采用数据的寄存器间接寻址或跳转的寄存器寻址时，寄存器中存储的都是相对于某个段的偏移量。特定寻址场景、特定寄存器对应有默认的段（隐式，一般不需要写出）：

- 数据访问：
  - `BX`，`SI`，`DI` ⟶ `DS`
  - `BP` ⟶ `SS`
- 转移指令：`CS`
- 串指令的 `DI` 默认相对于 `ES`

如需改变默认相对的段，则使用段超越，即在 `[]` 前加 `段寄存器:` 以指定偏移量寄存器参考的段，例如 `SS:[BX]`，`CS:[DI]`。

类型转换

有时仅凭符号名/变量名很难准确知道内存操作数的类型。由于内存单元的基本单位是字节，所以无论变量如何定义，都可以进行类型转换。

通过 `BYTE PTR` ， `WORD PTR` ， `DWORD PTR` 三个操作符，明确地指定内存操作数的类型，或进行强制类型转换。

例如：

```
MOV  BYTE PTR x, AL     ; 将 8 位的 AL 存入 x 对应的内存
MOV  WORD PTR [DI], AX  ; 将 AX 中的一个 16 位的字存入 DI 寄存器所指向的内存地址
```

在判断操作数与数据变量是否定义匹配时只检查变量名是否匹配，见下例：

```
EXAMPLE_DATA SEGMENT PARA
                ORG 100H
    EXM_BYTE    DB  'A',42H
    EXM_WORD    DW  4344H
    EXM_DWORD   DD  12345678H
    DATA_LEN    EQU $-EXM_BYTE
    EXM1_BYTE   DB  01H
EXAMPLE_DATA ENDS
```

```
MOV    AL,EXM_BYTE                          ; AL=41H,match
MOV    AL,EXM_BYTE+1                        ; AL=42H,match
MOV    AL,EXM_BYTE+3                        ; AL=43H,match
MOV    AX,WORD PTR EXM_DWORD                ; AX=5678H,dismatch,need WORD
PTR
MOV    AX,WORD PTR EXM_WORD+2               ; AX=5678H,match
```

第1个语句两个操作数都是8位的,直接匹配。

第2个语句的两个操作数也都是8位的,直接匹配。

在第3个语句中,虽然EXM_BYTE+3的地址已到了EXMWORD的单元,但汇编器检查类型匹配时查看的是变量名,所以第3个语句的操作数类型仍直接匹配。

类似的道理,第5个语句的两个操作数的类型也直接匹配。

第4个语句通过变量名EXM_DWORD来访问一个字,所以必须加上WORD PTR的属性转换操作符

主要指令

- 传送指令

  ```
  MOV, XCHG, PUSH, POP, PUSHF, POPF
  LEA, LDS, LES
  ```

  操作符：

```
OFFSET, SEG ; 获取变量符号的偏移量和段地址
BYTE PTR, WORD PTR, DWORD PTR ; 类型转换
```

- 算术运算指令

```
ADD, ADC, SUB, SBB, INC, DEC, CMP, NEG
MUL, IMUL, CBW
DIV, IDIV, CWD
```

- 逻辑运算指令

```
AND, OR, XOR, NOT
TEST
SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR
```

- 控制转移指令

```
JMP (short, near, word, far, dword)
JA/JB/JE 系列, JG/JL/JE 系列
LOOP, LOOPZ, LOOPNZ
CALL (near, word, far, dword)
RET, RETF
INT, IRET
```

- 处理器控制指令

```
CLC, STC, CLI, STI, CLD, STD, NOT, HLT
```

- 其他指令

```
LODS, STOS, MOVS, CMPS, SCAS, REP ; 串处理
IN, OUT
```

传送指令

```
MOV, XCHG, PUSH, POP, PUSHF, POPF
```

类型转换：打破类型匹配约定，按照希望的类型来寻址

- `BYTE PTR`，`WORD PTR`，`DWORD PTR`

段超越：打破操作数的段缺省约定，转向指定的段来寻址

- `CS:`，`DS:`，`ES:`，`SS:`

`MOV` 指令

语法为 `MOV DST, SRC`，必须遵守数据通路和以下规则：

- 源和目的操作数必须类型匹配（8 位对 8 位，16 位对 16 位）
- 目的操作数不能是立即数
- 源和目的操作数不能同时为内存操作数（串指令除外）
- 源和目的操作数不能通识为段寄存器

以下指令是错误的：

```
MOV AX, BL  ; 将 BL 赋值给 AX，扩展成 16 位 -- 类型不匹配
MOV ES, DS  ; 将 ES 设成与 DS 相同 -- 源和目的不能同时为段寄存器
MOV y x     ; 赋值 y = x -- 不能内存到内存
MOV [DI], [SI]  ; 间接寻址，内存变量传送 -- 不能内存到内存
```

类型转换：

```
MOV BYTE PTR x, AL
MOV WORD PTR [DI] AX
```

`MOV` 指令的运用（也称 `MOV` 体操）：

1. 遵循寻址方式规则：在哪里，怎么存取，access
2. 遵循数据通路规则：可以 / 不可以
3. 注意默认段（ `DS`，`SS` ）和段超越
4. 类型匹配和类型转换： `DB`，`DW`，`DD` 的范围，相互如何 access

`MOV` 指令不改标志位，只有运算指令改标志位。

交换指令 `XCHG`

格式为 `XCHG OPR1, OPR2`，使两个操作数互换。不允许任何一个操作数是立即数。

示例：

```
XCHG BX, [BP+SI]    ; 交换寄存器 BX 与堆栈段 SS:[BP+SI] 的操作数
```

堆栈指令

包括 `PUSH`，`POP`，`PUSHF`，`POPF`。示例：

```
PUSH SRC    ; SP=SP-2, SS:[SP]←SRC
PUSHF       ; SP=SP-2, SS:[SP]←PSW
POP DST     ; DST←SS:[SP], SP=SP+2
POPF        ; PSW←SS:[SP], SP=SP+2
```

其中 SRC 和 DST 都可以是寄存器及内存操作数。

其他传送指令

```
LEA     reg, src    ; 将源操作数 src 的偏移地址送入 reg 中
LDS     reg, src    ; 将 src 中的双字内容依次送入 reg 和 DS
LES     reg, src    ; 将 src 中的双字内容依次送入 reg 和 ES
```

上述三条指令中的 reg 不能是段寄存器。

- `LEA` 指令：获取 src 的偏移地址。
- `LDS` 和 `LES` 获取的是该单元处的双字内容，不是地址。
  - `LDS`：将低字送入 src，高字送入 `DS` 寄存器
  - `LES` 与 `LDS` 类似，高字使用 `ES` 寄存器
  - 使用 `LDS` 及 `LES` 时，src 处保存的双字通常是某个形如 `seg:offset` 的逻辑地址。

取地址还是取内容

当变量名（使用 `DW`，`DB`，`DD` 等伪指令定义的变量）直接位于 `MOV` 等指令中时，都是**直接寻址**，得到的是变量的内容。如果需要得到该变量地址，需要使用 `LEA` 指令。

```
x   DW  ?       ; 假定 x 在 DATA 段内，偏移地址为 000AH，内容为 1234H

MOV AX, x       ; AX = x = 1234H
LEA BX, x       ; BX = &x = 000AH
MOV AX, [BX]    ; AX = *BX = 1234H
```

`OFFSET` 和 `SEG` 操作符也可用于获取地址。

```
MOV BX, OFFSET x    ; 与 LEA BX, x 相同，获取 x 的偏移地址
MOV AX, SEG x       ;
```

传送指令示例

数据段：

```
X1  EQU     100
X2  DW      1234h
X3  DD      20005678h
```

内存图（`地址:数据`）：

```
DS: 0000:   34  X2
    0001:   12
    0002:   78  X3
    0003:   56
    0004:   00
    0005:   20
```

指令示例：

```
MOV X2, X1  ; 源：立即寻址，目的：直接寻址
MOV X3, X2  ; 错误：数据通路不对，类型不匹配

; MOV X3, X2 的正确版
MOV AX, X2  ; 源：直接寻址，目的：寄存器寻址
MOV WORD PTR X3, AX ; 源：寄存器寻址，目的：直接寻址
; 执行后 DS:[0002] ← 34h, DS:[0003] ← 12h

; 以下两条指令等价
LEA BX, X3  ; 源：直接寻址，目的：寄存器寻址，BX = &X3 = 0002h
MOV BX, OFFSET X3;  OFFSET 得到的偏移地址是 16 位的立即数
```

另一组示例

数据段：

```
X1  DW  2000h
X2  EQU 100
X3  DB  '1' ; 31h
X4  DD  12345678h
X5  DD  ?
```

内存图（ `地址:数据` ）：

```
DS: 0000:   00  X1
    0001:   20
    0002:   31  X3
    0003:   78  X4
    0004:   56
    0005:   34
    0006:   12
    0007:   ??  X5
    0008:   ??
    0009:   ??
    000A:   ??
```

指令示例（将 `X4` 的值赋给 `X5` ）：

```
LEA DI, X5              ; DI = &X5
MOV AX, WORD PTR X4     ; 源：直接寻址，目的：寄存器寻址
MOV [DI], AX            ; 源：寄存器寻址，目的：寄存器间接寻址
MOV AX, WORD PTR X4+2   ; 源：直接寻址(不是相对寻址)
MOV [DI+2], AX          ; 目的：寄存器相对寻址
; 每次搬运一个字，分两次完成双字的赋值
```

一些错误的指令：

```
MOV X5, X4              ; 错误，不能内存操作数直接赋值
MOV [DI], WORD PTR X4   ; 错误，内存-内存不能赋值
MOV AX, X4              ; 错误，类型不匹配
MOV [CX], AL            ; 错误,不能直接使用寄存器 CX 作为内存地址。
```

指令示例（将 `X3` 的值赋给 `X5`，`X5` 高位置零）：

注意 `X3` 是 `BYTE`，`X5` 是双字 `DD`。

采用直接寻址：

```
; 搬运最低位
MOV AL, X3
MOV BYTE PTR X5, AL     ; 目的：直接寻址
; 置零高位
XOR AL, AL              ; 清零
MOV BYTE PTR X5+1, AL   ; *(X5+1) = 00h
MOV BYTE PTR X5+2, AL
MOV BYTE PTR X5+3, AL
```

其中置零 `X5` 的高字也可以写成

```
XOR AX, AX  ; 将 AX 清零
MOV WORD PTR X5+2, AX   ; *(WORD*)(X5+2) = 0000h
```

仍然是 `X3` 赋给 `X5`，采用间接寻址：

```
MOV BX, OFFSET X5   ; BX=0007h, 取地址

MOV AL, X3          ; AL = X3
MOV [BX], AL        ; *BX = AL
XOR AL, AL          ; 清零 AL
INC BX              ; BX = BX + 1 = &X3 + 1
MOV [BX], AL        ; *BX = 0
INC BX              ; BX = &X3 + 2
MOV [BX], AL
INC BX              ; BX = &X3 + 3
MOV [BX], AL
```

注意：上述代码中的 `BX` 不能换成 `BP`，因为 `BP` 默认相对 `SS` 寻址，破坏堆栈段且 `X5` 没有改变。

算术运算指令

加减法指令

```
ADD dst, src    ; dst += src
ADC dst, src    ; dst += src + CF（带进位加）
INC opr         ; opr++
SUB dst, src    ; dst -= src
SBB dst, src    ; dst -= src - CF（带借位减）
DEC opr         ; opr--
```

算术运算影响符号位：

- `ZF`：如果运算结果为零则 `ZF=1`
- `SF`：等于运算结果 dst 的最高位，即符号位
- `CF`：加法有进位或减法有借位，则 `CF=1`
- `OF`：若操作数符号相同且相加结果符号与操作数相反，则 `OF=1`

加法举例

```
; 数据定义
X   DW  ?
Y   DW  ?
Z   DD  ?

; 代码实现 Z = X + Y
MOV DX, 0   ; 用 DX:AX 当被加数，先清零 DX
MOV AX, 0
MOV AX, X   ; AX 做被加数的低 16 位
ADD AX, Y   ; AX += Y，可能产生进位 CF=1
ADC DX, 0   ; DX += CF
MOV WORD PTR Z, AX      ; 储存和的低字
MOV WORD PTR Z+2, DX    ; 储存和的高字
```

减法举例

```
; 数据定义
X   DD  ?
Y   DD  ?
Z   DD  ?

; 代码实现 Z = X - Y
MOV DX, WORD PTR X+2    ; 用 DX:AX 作被减数，DX 作高字
MOV AX, WORD PTR X      ; AX 作低字
SUB AX, WORD PTR Y      ; 先进行低 16 位减法
SBB DX, WORD PTR Y+2    ; 高 16 位借位减法
MOV WORD PTR Z, AX      ; 储存差的低字
MOV WORD PTR Z+2, DX    ; 储存差的高字
```

求补和比较

```
NEG opr         ; opr = -opr
CMP opr1, opr2  ; opr1 - opr2，结果不送回，只影响标志位
```

指令	影响的标志位	标志位含义及设置条件	相关的J类跳转指令
`CMP`	`ZF`	零标志位 - 如果操作结果为零，则设置	`JE`，`JZ`（如果 `ZF` 设置，则跳转，即操作结果为零）
	`SF`	符号标志位 - 如果操作结果为负数，则设置	无直接相关，但可辅助其他标志位确定有符号比较的结果
	`OF`	溢出标志位 - 如果有符号整数溢出，则设置	无直接相关，但可辅助确定有符号操作是否溢出
	`AF`	辅助进位标志位 - 如果在低四位之间有进位或借位，则设置	无直接相关，但可辅助确定是否在低四位有进位或借位
	`CF`	进位标志位 - 如果最高位产生了进位或借位，则设置	`JB`，`JNAE`，`JC`（如果 `CF` 设置，则跳转，即有进位或借位）
			`JA`，`JNBE`（如果 `CF` 未设置且 `ZF` 未设置，则跳转，即无进位且结果非零，无符号大于）
			`JAE`，`JNB`，`JNC`（如果 `CF` 未设置，则跳转，即无进位，无符号大于等于）
			`JBE`，`JNA`（如果 `CF` 设置或 `ZF` 设置，则跳转，即有进位或结果为零，无符号小于等于）
	`PF`	奇偶标志位 - 根据操作结果的低八位的奇偶性设置	无直接相关，但可辅助确定结果的奇偶性

乘除法

无符号乘 `MUL`

- 字节操作数：8 位 x 8 位，`AX = AL * src`
- 字操作数：16 位 x 16 位，`DX:AX = AX * src`

其中 `src` 为 8 位或 16 位的 reg 或 mem，**不能是立即数。**

无符号除法 `DIV`

- 字节操作数：`AX / src`，商在 `AL`，余数在 `AH`
- 字操作数：`DX:AX / src`，商在 `AX`，余数在 `DX`

同理，`src` 不能是立即数。

举例：

```
MUL AL              ; AX = AL * AL
MUL 10              ; 错误，src 不能为立即数
DIV 10              ; 错误，src 不能为立即数
MUL X1              ; X1 为 DB 或 DW 变量
MUL [SI]            ; 错误，虽然可用内存操作数但类型不明
MUL BYTE PTR [SI]   ; 正确，AX = AL * op8
MUL WORD PTR [SI]   ; 正确，DX:AX = AX * op16
```

举例 `Y=X*10`

```
X   DW  ?
Y   DW  ?

MOV AX, X
MOV BX, 10
MUL BX      ; DX:AX = AX * BX
MOV Y, AX   ; 只考虑低 16 位，不考虑溢出
```

举例 `X=Y/10`

```
X   DW  ?
Y   DW  ?
MOV AX, Y
MOV BX, 10
MOV DX, 0   ; 清零 DX，被除数是 DX:AX
DIV BX      ; DX:AX / BX，商在 AX 余数在 DX
MOV X, AX   ; 忽略余数
```

注意这里不能用 8 位除法，否则会溢出。

逻辑运算

```
AND dst, src    ; dst &= src
OR dst, src     ; dst |= src
XOR dst, src    ; dst ^= src
NOT dst         ; dst = ~dst
```

可以用这些指令实现组合/屏蔽/分离/置位，例如：

```
AND AL, 0FH     ; 清零高 4 位
AND AL, F0H     ; 清零低 4 位
AND AL, FEH     ; 清零最低位
OR  AL, 80H     ; 最高位置 1
XOR AL, AL      ; 清零 AL，等价于 MOV AL, 0 且效率更高

OR AL,  30H     ; 将 0~9 变为 '0'~'9'
AND AL, 0FH     ; 将 '0'~'9' 变为 0~9
```

移位指令

```
SHL dst, count  ; 逻辑左移
SAL dst, count  ; 算术左移
SHR dst, count  ; 逻辑右移
SAR dst, count  ; 算术右移
ROL dst, count  ; 循环左移
ROR dst, count  ; 循环右移
RCL dst, count  ; 进位循环左移
RCR dst, count  ; 进位循环右移
```

注意：`count` 只能为 `1` 或 `CL`

> 上述结论来自熊老师的课本P51，但网上说count可以为int8的立即数，存疑

示例 `X = X * 10`，`X = (X << 3) + (X << 1)`

```
X   DW  ?

MOV BX, X
SHL BX, 1
PUSH BX     ; X << 1
SHL BX 1
SHL BX 1    ; X << 3
POP AX      ; AX = X << 1
ADD AX, BX  ; AX = (X<<1) + (X<<3)
MOV X, AX
```

示例 双字 `X`，`X << 4`

```
X   DD  ?

MOV AX, WORD PTR X
MOV DX, WORD PTR X+2
SHL AX, 1   ; 移出 CF
RCL DX, 1   ; 移入 CF
SHL AX, 1
RCL DX, 1
SHL AX, 1
RCL DX, 1
SHL AX, 1
RCL DX, 1

; 也可以用循环实现但不如展开的效率高
MOV CX, 4
LP1:
SHL AX, 1
RCL DX, 1
LOOP LP1
```

> 需要注意CF所在的位置是转移方向的最高位，而不是数据的最高位，具体见下方图例

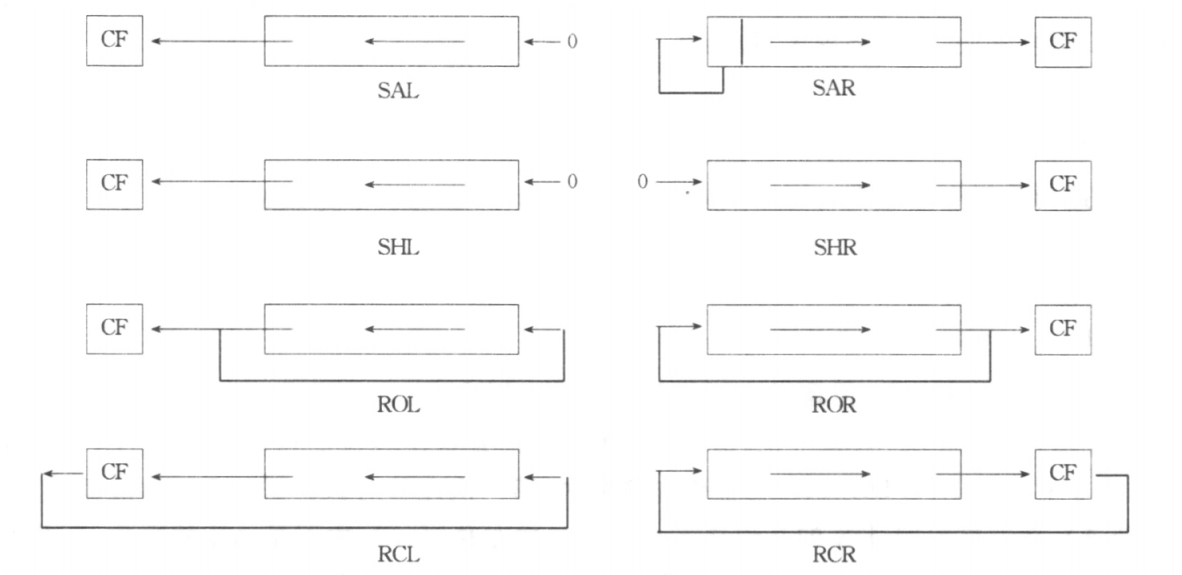


图3.8　八条移位指令操作图解

转移指令

条件转移

指令	别称	含义	无符号比较条件	有符号比较条件	标志依据
`JA`	`JNBE`	跳转如果高于	无符号大于（`CF=0` 且 `ZF=0`）		CF ZF
`JAE`	`JNB` 、 `JNC`	跳转如果高于等于	无符号大于等于（`CF=0`）		CF

指令	别称	含义	无符号比较条件	有符号比较条件	标志依据
`JE`	`JZ`	跳转如果等于	等于（`ZF=1`）	等于（`ZF=1`）	ZF
`JBE`	`JNA`	跳转如果低于等于	无符号小于等于（`CF=1` 或 `ZF=1`）		CF ZF
`JB`	`JNAE`、`JC`	跳转如果低于	无符号小于（`CF=1`）		CF
`JNE`	`JNZ`	跳转如果不等于	不等于（`ZF=0`）	不等于（`ZF=0`）	ZF
`JG`		跳转如果大于		有符号大于（`ZF=0` 且 `SF=OF`）	ZF SF OF
`JL`		跳转如果小于		有符号小于（`ZF=0` 且 `SF≠OF`）	ZF SF OF

- 无符号比较：`JA` / `JB` / `JE` 系列（**A**bove / **B**elow / **E**qual）
- 有符号比较：`JG` / `JL` / `JE` 系列（**G**reater / **L**ess / **E**qual）

指令格式：`JX 标号`，比较的依据是**标志位**（紧跟 `CMP` 指令）。标号位于指令前面，实质是一个段内偏移值。

无符号数的条件转移指令：

- `JA`（`JNBE`）：无符号高于时转移
- `JAE`（`JNB` / `JNC`）：无符号高于等于时转移（`CF=0` 时转移）
- `JE`（`JZ`）等于时转移（`ZF=1` 时转移）

- `JBE`（`JNA`）：无符号低于等于时转移
- `JB`（`JNAE` / `JC`）：无符号低于时转移（`CF=1` 时转移）
- `JNE`（`JNZ`）：不等于时转移（`ZF=0` 时转移）

示例 求 `Z=|X-Y|`，`X`，`Y`，`Z` 都是无符号数。

```
    MOV AX, X
    CMP AX, Y   ; if (AX < Y) swap AX, Y
    JAE L1      ; AX ≥ Y 则跳过交换
    XCHG AX, Y  ; 交换 AX 和 Y
L1: SUB AX, Y   ; AX -= Y
    MOV Z, AX
```

循环指令

格式：`LOOP 标号`。

`LOOP`：先 `CX -= 1`，**然后**当 `CX` 不为零时转移。（与循环体无关，只是个转移指令）

`JCXZ`：当 `CX=0` 时转移（不执行 `CX -= 1`）

示例

```
MOV CX, 4
LP1:
......
LOOP LP1    ; loop body is executed 4 times

MOV CX, 4
LP2:
DEC CX
JCXZ LP2
```

无条件转移指令

`JMP` 指令，格式：`JMP 标号|寄存器操作数|内存操作数`

1. 段内直接短转移：`JMP SHORT PTR 标号`，EA 是 8 位
2. 段内直接转移：`JMP NEAR PTR 标号`
3. 段内间接转移：`JMP WORD PTR 寄存器或内存`
4. 段间直接转移：`JMP FAR PTR 标号`
5. 段间间接转移：`JMP DWORD PTR 寄存器或内存`

子程序调用

- `CALL` 指令：子程序调用
- `RET` 指令：从子程序中返回

`CALL` 指令：

1. 段内直接调用：`CALL dst`
   - `SP=SP-2`，`SS:[SP]`：返回地址偏移值
   - `IP=IP+dst`
2. 段内间接调用：`CALL dst`

- `SP=SP-2`，`SS:[SP]`：返回地址偏移值
- `IP=*dst`

3. 段间直接调用：`CALL dst`

- `SP=SP-2`，`SS:[SP]`：返回地址段值
- `SP=SP-2`，`SS:[SP]`：返回地址偏移值
- `IP=OFFSET dst`
- `CS=SEG dst`

4. 段间间接调用：`CALL dst`

- `SP=SP-2`，`SS:[SP]`：返回地址段值
- `SP=SP-2`，`SS:[SP]`：返回地址偏移值
- `IP` 为 `EA` 的低 16 位
- `CS` 为 `EA` 的高 16 位

段内调用，`dst` 应为 `NEAR PTR`，段间调用则为 `FAR PTR`。

示例

```
CALL P1             ; 段内直接调用 P1, P1 为 NEAR
CALL NEAR PTR P1    ; 同上
CALL P2             ; 段间直接调用 P2, P2 为 FAR
CALL FAR PTR P2     ; 同上
CALL BX             ; 段内间接寻址，过程地址位于 BX 中
CALL [BX]           ; 段内间接地址，过程地址位于数据段中
CALL WORD PTR [BX]  ; 同上
```

`RET` 指令：

1. 段内返回：`RET`

- `IP=[SP]`，`SP=SP+2`

2. 段内带立即数返回：`RET exp`

- `IP=[SP]`，`SP=SP+2`
- `SP=SP+exp`

3. 段间返回：`RET`

- `IP=[SP]`，`SP=SP+2`
- `CS=[SP]`，`SP=SP+2`

4. 段间带立即数返回：`RET exp`

- `IP=[SP]`，`SP=SP+2`
- `CS=[SP]`，`SP=SP+2`
- `SP=SP+exp`

过程的定义

```
过程名      PROC [near | far]
             过程体
             RET
过程名      ENDP
```

应用举例

将内存中的值 x 显示为 10 进制

```
MOV AX, X        ; 取 AX
XOR DX, DX       ; DX:AX 作为被除数
MOV BX, 10000    ; 依次除以 10000, 1000, 100, 10 显示商
DIV BX           ; DX:AX / BX, 商在 AX, 余数在 DX
PUSH DX          ; 保存余数
MOV DL, AL       ; 显示的字符送 DL
OR DL, 30H       ; 0~9 → '0'~'9'
MOV AH, 2        ; DOS 2 号功能调用, 显示 DL 的 ASCII 字符
INT 21H          ; DOS 功能调用

POP AX           ; 上次的余数作为被除数
XOR DX, DX
MOV BX, 1000     ; 上次 10000, 这次 1000
DIV BX
PUSH DX
MOV DL, AL
OR DL, 30H
MOV AH, 2
INT 21H

POP AX
XOR DX, DX
MOV BX, 100      ; 1000 → 100
DIV BX
PUSH DX
MOV DL, AL
OR DL, 30H
MOV AH, 2
INT 21H

POP AX
XOR DX, DX
MOV BX, 10       ; 100 → 10
DIV BX
PUSH DX
MOV DL, AL
OR DL, 30H
MOV AH, 2
INT 21H

POP AX
MOV DL, AH       ; 最后一个余数送给输出
OR DL, 30H       ; 转成 ASCII 码
MOV AH, 2
INT 21H
```

以上代码也可以写成循环（次数为 4），不如展开了效率高

程序结构

三段式程序结构

堆栈段，数据段，程序段。

定义堆栈段：

```
STACK       SEGMENT PARA STACK
STACK_AREA  DW  100h DUP(?)
STACK_TOP   EQU $-STACK_AREA
STACK       ENDS
```

定义数据段：

```
DATA        SEGMENT PARA
TABLE_LEN   DW  16
TABLE       DW  200, 300, 400, 10, 20, 0, 1, 8
            DW  41H, 40, 42H, 50, 60, 0FFFFH, 2, 3
DATA        ENDS
```

定义段

伪指令：`SEGMENT` 和 `ENDS`

格式：

```
段名    SEGMEMT [对齐类型] [组合类型] [类别名]
        ; 本段中的程序和数据定义语句
段名    ENDS
```

对齐类型：定义了段在内存中分配时的起始边界设定

- `PAGE`：本段从页边界开始，一页为 256B
- `PARA`：本段从节边界开始，一节为 16B
- `WORD`：本段从字对齐地址(偶地址)开始，段间至多 1B 空隙
- `BYTE`：本段从字节地址开始，段间无空隙

组合类型：确定段与段之间的关系

- `STACK`：该段为堆栈段的一部分。链接器链接时，将所有同名的具有 `STACK` 组合类型的段连成一个堆栈段，并将 `SS` 初始化为其首地址，`SP` 为段内最大偏移。（正确定义段的 `STACK` 属性可以在主程序中省略对 `SS` 和 `SP` 的初始化）

定义过程

伪指令：`PROC`，`ENDP`，`END 标号`

格式：

```
过程名  PROC    [NEAR|FAR]
        ; 过程代码
        RET
过程名  ENDP
```

- 如果过程为 `FAR` 则 `RET` 被编译为 `RETF`（段间返回）
- `RET 2n`：在 `RET` 或 `RETF` 后 `SP+=2n`

`END 标号` 为程序总结束，标号为入口（被设置为初始的 `CS:IP` 值）

定义数据

格式：

```
变量名  伪指令    值
```

用 `DB`，`DW`，`DD` 伪指令定义内存中的变量，变量名对应内存地址。

用 `EQU` 伪指令定义常量，不占内存，变量名被翻译成立即数。

值的表示：

- 常数
- `DUP` 重复操作符（用于定义数组，或定义堆栈空间）
- `?` 不预置任何内容
- 字符串表达式
- 地址表达式
- `$` 当前位置计数器

`DUP` 表达式

```
ARRAY1  DB  2 DUP (0, 1, 0FFH, ?)
ARRAY2  DB  100 DUP (?)
```

其中 `ARRAY1` 定义了一个二维数组，等价于

```
ARRAY1  DB  0, 1, 0FFH, ?, 0, 1, 0FFH, ?
```

`ARRAY2` 定义了一段长度为 100 字节的连续空间，值未初始化。

`DUP` 表达式可以嵌套，相当于定义多维数组。

```
ARRAY3  DB  2 DUP(1, 2 DUP ('A', 'B'), 0) ; 'A', 'B' 为字符 ASCII 码 41H 和 42H
```

对应的内存图：

```
ARRAY3+ 0000:   01H
        0001:   41H
        0002:   42H
        0003:   41H
        0004:   42H
        0005:   00H
        0006:   01H
        0007:   41H
        0008:   42H
        0009:   41H
        000A:   42H
        000B:   00H
```

地址表达式

使用 `DW` 及 `DD` 伪指令后跟标号，表示变量的偏移地址或逻辑地址。使用 `DD` 存储逻辑地址时，低字为偏移地址，高字为段地址。

当前位置计数器：用 `$` 表示，为当前标号所在的偏移地址。

```
X1  DW  ?   ; X1 内容未定义
X2  DW  $   ; X2 单元存放当前（X2 自身的）偏移地址
X3  DW  X1  ; X3 单元存放 X1 偏移地址
X4  DW  L1  ; X4 单元存放 L1 标号的偏移地址
X5  DW  P1  ; X5 单元存放 P1 子程序的偏移地址
X6  DD  X1  ; X6 单元存放 X1 的逻辑地址
X7  DD  L1  ; X7 单元存放 L1 标号的逻辑地址
X8  DD  P1  ; X8 单元存放 P1 子程序的逻辑地址
```

字符串表达式

```
STR1    DB  'ABCD', 0DH, 0AH, '$'
STR2    DW  'AB', 'CD'
```

其中 `STR1` 分配了 7 个单元(字节)，按顺序存放。其中 `$` 为字符串的结束（使用 DOS 9 号功能调用输出字符串，结果为 `ABCD\r\n` ）

`STR2` 分配了两个字（4 个字节），按顺序其值分别为 `42H` ， `41H` ， `44H` ， `43H` 。对于 `DW` 和 `DD` 伪指令，不允许使用两个以上字符的字符串作为其参数。

初始化寄存器

```
MOV AX, STACK
MOV SS, AX
MOV SP, STACK_TOP
MOV AX, DATA
MOV DS, AX
```

`ASSUME` 伪指令：告诉汇编器，将 `CS` ， `DS` ， `SS` 分别设定成相应段的首地址（仅仅是"设定"，并没有真正将地址存入段寄存器，仍然需要使用上述几条指令来初始化段寄存器）

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

HELLOWORLD示例

```
STACK       SEGMENT PARA STACK
STACK_AREA  DW  100h DUP(?)
STACK_TOP   EQU $-STACK_AREA
STACK       ENDS

DATA        SEGMENT PARA
MESSAGE     DB  'Hello, World!', '$'
DATA        ENDS

CODE        SEGMENT
ASSUME      CS:CODE, DS:DATA, SS:STACK

MAIN        PROC
            ; initialize
            MOV AX, STACK
            MOV SS, AX
            MOV SP, STACK_TOP
            MOV AX, DATA
            MOV DS, AX
            ; display message
            MOV AH, 9
            LEA DX, MESSAGE
            INT 21H
            ; return to dos
            MOV AX, 4C00H
            INT 21H

MAIN        ENDP
CODE        ENDS
END         MAIN
```

字符串处理

串操作指令

使用方法

串操作隐含着间接寻址：`DS:[SI] --> ES:[DI]`

使用串指令的常见过程：

- 设置 `DS`，`SI`（源串），`ES`，`DI`（目的串）
  - `DS` 和 `ES` 是段寄存器，不能直接用 `MOV ES, DS` 赋值
  - 借助其他寄存器（`MOV AX, DS`，`MOV ES, AX`）或堆栈（`PUSH DS`，`POP ES`）
- 设置 `DF` 标志（Direction Flag）

    - 通过 `CLD`（`DF=0`）和 `STD`（`DF=1`）指令
    - `DF=0` 则每次 `SI/DI++`（或 `+=2`），`DF=1` 则每次 `SI/DI--`（或 `-=2`）
- 选用重复执行指令 `REP*` 以及设置重复次数 `CX`
    - 重复指令包括无条件重复 `REP`，有条件重复 `REPE/REPZ`，`REPNE/REPNZ`
    - `CX` 表示最大的重复次数
    - 重复的串指令每执行一次则 `CX--`

指令种类

串指令种类：

- 取串指令 `LODSB` / `LODSW`
    - 将源串 `DS:[SI]` 的一个字节或字取到 `AL` 或 `AX`，同时按照 `DF` 修改 `SI` 的值。
    - `LODSB` 取字节，`LODSW` 取字
- 存串指令 `STOSB` / `STOSW`
    - 将 `AL` 中的一个字节或 `AX` 的一个字存入目的串 `ES:[DI]`，并根据 `DF` 修改 `DI`。
    - `STOSB` 存字节，`STOSW` 存字
- 串传送指令 `MOVSB` / `MOVSW`
    - 将源串 `DS:[SI]` 的字节或字传送到目的串 `ES:[DI]`，并根据 `DF` 修改 `SI` 及 `DI`。
    - `MOVSB` 传送字节，`MOVSW` 传送字
- 串比较指令 `CMPSB` / `CMPSW`
    - 比较源串 `DS:[SI]` 与目的串 `ES:[DI]` 的一个字节或字。执行完后根据 `DF` 修改 `SI` 及 `DI`。
    - 用源串的字节或字减去目的串的字节或字，影响标志寄存器 `CF`，`SF`，`ZF`。
    - 相当于 `CMP DS:[SI], ES:[DI]`
    - 后面往往跟着条件转移指令
- 串扫描指令 `SCASB` / `SCASW`
    - 在目的串 `ES:[DI]` 中扫描是否有 `AL` 或 `AX` 指定的字节或字。执行完后根据 `DF` 修改 `SI` 及 `DI`。
    - 相当于 `CMP AL/AX, ES:[DI]`
    - 比较结果不保存，只影响标志寄存器。

重复前缀指令 `REP`：

- 格式：`REP 串操作指令`，例如 `REP MOVSB`
- 每执行一次串操作，`CX--`，直到 `CX` 为零时停止。

条件重复前缀指令 `REPE` / `REPZ`，`REPNE` / `REPNZ`

- 格式与 `REP` 指令相似
- 每执行一次，`CX--`，当 `CX` 等于零或 `ZF` 不满足条件时停止。
- `REPE` / `REPZ`：当 `CX ≠ 0` 且 `ZF=1` 时执行串操作并 `CX--`
- `REPNE` / `REPNZ`：当 `CX ≠ 0` 且 `ZF=0` 时执行串操作并 `CX--`

修改 `DF` 方向标志指令

- `CLD`：`DF=0`，执行串操作后 `SI` 和 `DI` 增加（根据字节或字操作决定 `+1` 还是 `+2`）

- `STD`：`DF=1`，执行串操作后 `SI` 和 `DI` 减少（根据字节或字操作决定 `-1` 还是 `-2`）

基本应用

- 扫描长度为 `LEN` 的串 `STRING1` 中是否含有字母 `A`

```
PUSH  ES

PUSH  DS
POP   ES          ; ES = DS

MOV   DI, OFFSET STRING1

MOV   CX, LEN
MOV   AL, 'A'
CLD
REPNZ SCASB
; while (CX ≠ 0 && AL ≠ ES:[DI]) CMP AL, ES:[DI] / SI++,DI++

JZ    FOUND   ; 找到

POP   ES
```

`REPNZ SCASB` 停下来后，如果 `ZF=1` 则 `AL=ES:[DI]`，通过 `CX` 的值可以看出 `A` 在 `STRING1` 中的位置。如果 `ZF=0` 则已经遍历到最后仍未找到。

综合应用

- 在缓冲区中查找 `\r\n`（`0DH`，`0AH`）并将其删掉（将若干行文本拼接成不换行的文本）
- 在缓冲区查找换行符 `0AH` 并将其补成回车换行 `0DH, 0AH`。

调试器

调试程序：`DEBUG.EXE HELLO.EXE`

常用调试命令：

命令	功能描述	使用示例
`d`	显示内存单元内容	`d` 显示从 `CS:IP` 开始的内存内容
		`d DS:0000` 显示DS段地址0000开始的内存内容
`e`	修改内存单元内容，每次修改一个字节，可以连续修改多个字节	`e 078A:0000` 修改地址078A:0000开始的内存内容
`r`	查看和修改寄存器	`r AX` 查看或修改AX寄存器的值
`u`	反汇编，查看机器码对应的汇编程序	`u` 反汇编从 `CS:IP` 开始的机器码
		`u DS:0000` 反汇编DS段地址0000开始的机器码
`a`	修改汇编指令，可以连续修改多条指令	`a` 修改 `CS:IP` 指令
		`a 078A:0000` 修改078A:0000地址的汇编指令
`t`	执行一条指令，同 Step In	`t` 单步执行当前指向的指令
`p`	执行完子程序/循环/功能调用，同 Step Over	`p` 执行到下一个断点或程序结束
`g`	执行到某个指令地址处，相当于断点，不加参数则执行到结束	`g 078A:0000` 执行到078A:0000地址处的指令
`l`	装入文件	`l myfile.asm` 装入名为myfile.asm的文件
`w`	写回文件	`w myoutput.asm` 将修改后的内容写回到myoutput.asm文件
`q`	退出调试器	`q` 退出当前的调试会话