汇编语言

汇编指令

MOV

MOV [寄存器], [立即数]

MOV [寄存器 A], [寄存器 B] (A.size == B.size)

- A，B不可都为段寄存器

MOV [寄存器], [内存单元地址]

- MOV [段寄存器], [内存单元地址]
- MOV [通用寄存器], [内存单元地址]

MOV [内存单元地址], [寄存器]

- MOV [内存单元地址], [段寄存器]
- MOV [内存单元地址], [通用寄存器]

MOVSB

以字节为单位的串传送指令

将DS:SI中的字节送入ES:DI中, 然后根据标志寄存器DF位的值, 决定SI和DI递增或递减

相当于:

(1) ((es)×16 + (di)) = ((ds) ×16 + (si))

(2) 如果DF = 0则： (si) = (si) + 1

(di) = (di) + 1

如果DF = 0则：(si) = (si) - 1

(di) = (di) - 1

MOVSW

功能：（以**字**为单位传送）

将 ds:si指向的内存字单元中word送入es:di中，然后根据标志寄存器DF位的值，将si和di递增2或递减2。

REP

MOVSB 和 MOVSW 进行的是串传送操作中的一个步骤(即单个指令仅传送一个指令)

通常MOVSB和MOVSW都和REP配合使用, 格式:

REP MOVSB

REP的作用是根据CX的值, 重复执行后面的串传送指令, 汇编语言描述如下:

s: movsb

loop s

CLD && STD

设置DF标志位的命令:

CLD 设置DF为0, 正向传送

STD 设置DF为1, 逆向传送

ADD

ADD [寄存器], [立即数]

ADD [寄存器 A], [寄存器 B] (A.size == B.size)

ADD [寄存器], [内存单元]

ADD [内存单元], [寄存器]

ABC

ABC是带进位加法指令, 它利用了CF位上记录的进位值

指令格式: ABC 操作对象1, 操作对象2

功能: 操作对象1 = 操作对象1 + 操作对象2 + CF

可以利用ABC指令设计任意多位的数据相加

SUB

SUB [寄存器], [立即数]

SUB [寄存器 A], [寄存器 B] (A.size == B.size)

SUB [寄存器], [内存单元]

SUB [内存单元], [寄存器]

SBB

SBB是带借位减法指令, 它利用了CF位上的借位值

格式：sbb 操作对象1,操作对象2

功能：

操作对象1=操作对象1–操作对象2–CF

比如：sbb ax,bx

实现功能： (ax) = (ax) – (bx) – CF

JMP

JMP 段地址：偏移地址　　；同时修改CS、IP(仅debug中使用)

JMP 寄存器　　；用寄存器中的值仅修改IP（要先MOV到寄存器中）

JMP为无条件转移， 可以只修改IP， 也可以同时修改CS和IP

JMP指令要给出两种信息：

- 转移的目的地址
- 转移的距离（段间转移、段内短转移、段内近转移）

段内短转移：

JMP short 标号（转到标号处执行指令）

对应的机器码中包含转移的位移, 而不是目的地址

功能为：

(IP) = (IP) + 8位位移

- 8位位移 = "标号"处的地址 - JMP指令后的第一个字节的地址
- short指明此处的位移为8位位移
- 8位位移的范围为-128 ~ 127, 用补码表示
- 8位位移由编译程序在编译时算出

段内近转移

JMP near ptr 标号（转到标号处执行指令）

对应的机器码中包含转移的位移, 而不是目的地址

功能为：

(IP) = (IP) + 16位位移

- 16位位移 = "标号"处的地址 - JMP指令后的第一个字节的地址
- near ptr指明此处的位移为16位位移
- 8位位移的范围为-32769 ~ 32768, 用补码表示
- 16位位移由编译程序在编译时算出

段间转移

JMP far ptr 标号

实现的是断间转移, 又称为远转移:

功能: far ptr指明了指令用标号的段地址和偏移地址修改CS和IP

转移地址在内存中的JMP指令

- JMP word ptr 内存单元地址(段内转移)
- JMP dword ptr 内存单元地址(段间转移)

  功能: 从内存单元地址处开始存放着两个字, 高地址处的字是转移的目的段地址, 低地址处是转移的目的偏移地址: (CS) = (内存单元地址+2), (IP) = (内存单元地址)

有条件转移指令

所有的有条件转移指令都是短转移， 转移位移都是[-128， 127]

JCXZ

JCXZ指令为有条件转移指令

对应的机器码中包含转移的位移, 而不是目的地址

格式:

JCXZ 标号

如果(CX) = 0, 则转移到标号处执行

根据CMP指令的结果（标志寄存器状态）进行转移的有条件转移指令

因为CMP指令可以进行两种比较，无符号数比较和有符号数比较，所以根据CMP指令的比较结果进行转移的指令也分为两种：

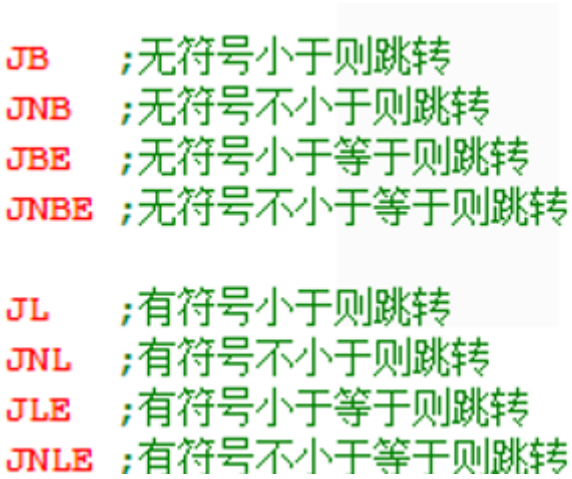
- 根据无符号数的比较结果进行转移的条件转移指令（检测ZF、CF的值）
- 根据有符号数的比较结果进行转移的条件转移指令（检测SF、OF和ZF的值）

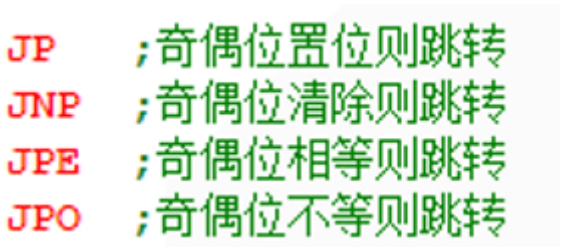
无符号数比较的条件转移指令如下：

指令	含义	检测的相关标志位
JE（equal）	等于即转移	ZF=1
JNE（not equal）	不等于即转移	ZF=0
JB（below）	低于即转移	CF=1
JNB（not below）	不低于即转移	CF=0
JA（above）	高于即转移	CF=0 且 ZF=0
JNA（not above）	不高于即转移	CF=1 或 ZF=1

注： 在条件转移指令前是否使用CMP指令，在于程序员的安排，并不是必须的

```
JE    ;等于则跳转
JNE   ;不等于则跳转

JZ    ;为 0 则跳转
JNZ   ;不为 0 则跳转

JS    ;为负则跳转
JNS   ;不为负则跳转

JC    ;进位则跳转
JNC   ;不进位则跳转

JO    ;溢出则跳转
JNO   ;不溢出则跳转

JA    ;无符号大于则跳转
JNA   ;无符号不大于则跳转
JAE   ;无符号大于等于则跳转
JNAE  ;无符号不大于等于则跳转

JG    ;有符号大于则跳转
JNG   ;有符号不大于则跳转
JGE   ;有符号大于等于则跳转
JNGE  ;有符号不大于等于则跳转
```





LOOP

循环有条件转移指令

PUSH

PUSH [寄存器]

PUSH [段寄存器]

PUSH [内存单元]

POP

POP [寄存器]

POP [段寄存器]

POP [内存单元]

PUSHF

将标志寄存器入栈

POPF

从栈中弹出数据, 送入标志寄存器中

XCHG

将一个字节或一个字的源操作数和目的操作数相交换

指令格式:

XCHG OPRD1, OPRD2

目次 源

交换指令可以在通用寄存器(不能为段寄存器)之间, 寄存器与存储器之间进行

- 不能同时都为内存操作数
- 任何一个操作数不能为段寄存器
- 任何一个操作数不能为立即数
- 两个操作数的长度必须相等

IN && OUT

由于I/O地址空间是独立编址的，因此系统需要提供独立的访问外设指令（I/O指令）

只能使用ax或al来存放从端口读入的数据或要发送到端口中的数据

访问8位端口用al, 访问16位端口用ax

- 对0～255端口进行读写：

  in al,20h ;从20h端口读入一个字节

  out 20h,al ;往20h端口写入一个字节

- 对256～65535端口读写时，端口号放在dx中：

  mov dx,3f8h ;将端口号3f8送入dx

  in al,dx ;从3f8h端口读入一个字节

  out dx,al ;向3f8h端口写入一个字节

LEA

INC

CMP

CMP是比较指令, 功能相当于减法指令, 只是**不保存结果**, 但**对标志寄存器产生影响**

CMP 操作对象1, 操作对象2(操作对象1只能为寄存器/地址访存不能为立即数, 操作对象2可以为立即数/寄存器/地址访存)

- 对无符号数进行比较时:
  - ZF = 1, 说明 1 = 2
  - ZF = 0, 说明 1 != 2
  - CF = 1, 说明1 < 2
  - CF = 0, 说明 1 >= 2
  - CF = 0 且 ZF = 0, 说明 1 > 2
  - CF = 1 或 ZF = 1, 说明 1 <= 2
- 对有符号数进行比较时:

  由于存在溢出的情况, 仅根据符号标志位SF无法判断1 和 2的相对大小, 因此还要根据OF加以判断

  - ZF = 1, 说明 1 = 2
  - ZF = 0, 说明 1 != 2
  - SF = 1 && OF = 0, 说明 1 < 2
  - SF = 0 && OF = 0, 说明 1 >= 2
  - SF = 1 && OF = 1, 说明 1 > 2
  - SF = 0 && OF = 1, 说明 1 < 2

MUL

MUL是无符号数乘法指令

格式:

- MUL register
- MUL 内存单元

相乘的两个数: 要么是8位, 要么是16位

8位: 一个数默认存放在AL中, 另一个放在8位register或内存字节单元中(用寻址方式给出时需要加上byte ptr), 结果默认放在AX中

16位: 一个数默认存放在AX中, 另一个放在16位register或内存字单元中(用寻址方式给出时需要加上word ptr), 结果高位默认在DX中存放, 低位在AX中存放

DIV

除数: 8位或16位, 放在寄存器或者内存单元中

被除数: (默认)放在AX(8位除数) 或 DX(高16位)和AX(低16位)中(16位除数)

结果

除数	**8位**	**16位**
商	AL	AX
余数	AH	DX

格式如下:

DIV [register]

DIV [内存单元] (用word ptr / word ptr 指出除数位数)

AND

AND [通用寄存器], [立即数位串]

AND [地址访存], [立即数位串]

OR

OR [通用寄存器], [立即数位串]

OR [地址访存], [立即数位串]

SHL && SHR

SHL和SHR是逻辑移位指令

SHL逻辑左移指令, 功能为:

1. 将一个寄存器或内存单元中的数据向左移位
2. 将最后移出的一位写入CF中
3. 最低位用0补充

如果移动位数大于1时, 必须将移动位数放在CL中

SHR的功能与SHL正好相反

RET

- RET

  RET指令用栈中的数据，修改IP的内容，从而实现近转移

  CPU执行RET指令时，进行下面两步操作：

  - (IP) = ((SS*16)+(SP))
  - (SP) = (SP) + 2

  相当于:POP IP

- RETF

  RETF指令用栈中的数据, 修改CS和IP中的内容, 从而实现远转移

  CPU执行RETF指令时, 进行下面4步操作:

  - (IP) = ((SS)*16+(SP))
  - (SP) = (SP) + 2
  - (CS) = ((SS)*16+(SP))
  - (SP) = (SP) + 2

  相当于: POP IP

  POP CS

IRET

IRET指令用于中断返回

IRET指令的功能用汇编语法描述如下:

POP IP

POP CS

POPF

与处理中断的入栈顺序正好相反

CALL

CPU执行CALL指令时, 进行两步操作:

1. 将当前的IP或CS和IP压入栈中
2. 转移

   CALL指令不能实现短转移, 除此之外, CALL指令实现转移的方法和JMP指令的原理相同

- 依据位移进行转移的CALL指令

  CALL 标号

  相当于执行:

  - PUSH IP
  - JMP NEAR PTR 标号
- 转移目的地址在指令中的CALL指令

  CALL FAR PTR 标号

  相当于执行:

  - PUSH CS
  - PUSH IP
  - JMP FAR PTR 标号
- 转移地址在寄存器的CALL指令

CALL 16位register

相当于执行:

- PUSH IP
- JMP 16位register

- 转移地址在内存中的CALL指令
  - CALL WORD PTR 内存单元地址

    相当于执行:

    - PUSH IP
    - JMP WORD PTR 内存单元地址
  - CALL DWORD PTR 内存单元地址

    相当于执行:

    - PUSH CS
    - PUSH IP
    - JMP DWORD PTR 内存单元地址

伪指令

XXX segment ... XXX ends

定义一个段

XXX 为标号，一个标号代表了一个地址，最终将被编译、连接程序处理为一个段的段地址

end

end是一个汇编程序的结束标记， 编译器在编译汇编程序的过程中，如果碰到了end，就结束对源程序的编译

end除了通知编译程序结束外, 还可以通知编译器程序的入口在什么地方.

end start 指令指明了程序的入口在标号start处

assume

这条伪指令的含义为“假设”。它假设某一段寄存器和程序中的某一个用segment定义的段相关联

db dw dd

db 定义字节型数据

dw定义字型数据

dd定义双字型数据

dup

和db, dw, dd等伪指令配合使用,用来进行数据的重复

使用格式:

db 重复的次数 dup (重复的字节型数据)

dw 重复的次数 dup (重复的字型数据)

dd 重复的次数 dup(重复的双字数据)

db 3 dup (0, 1, 2) : 0 1 2 0 1 2 0 1 2

db 3 dup ('abc', 'ABC'): abcABCabcABCabcABC

PROC

过程定义伪指令格式:

子程序名 PROC 属性

……

子程序名 ENDP

属性分为NEAR属性和FAR属性 默认为NEAR

子程序和主程序在同一代码段使用NEAR属性, 否则, 使用FAR属性

ORG

ORG伪指令用来指出其后的程序段或数据块存放的起始地址的偏移量。

其格式为：

ORG 表达式

汇编程序把语句中表达式之值作为起始地址,连续存放程序和数据，直到出现一个新的ORG指令。若省略ORG，则从本段起始地址开始连续存放。

课本知识

第一章 基础知识

1.2 汇编语言的产生

汇编语言是一种符号化的机器语言，即用指令助记符、符号地址、标号等符号书写程序的语言，。汇编语言的主体是汇编指令。

汇编器（编译器）（assembler）：一种工具程序，将汇编程序转换为机器语言。

链接器（linker）：一种工具程序，把汇编器生成的单个文件组合成一个可执行文件。

调试器（debugger）：是程序员在程序运行时，跟踪程序执行过程和各器件状态。

1.3 汇编语言的组成

汇编语言发展至今，有以下3类指令组成

- 汇编指令：机器码的助记符，有对应的机器码
- 伪指令：没有对应的机器码，由编译器执行，计算机并不执行
- 其他符号：由编译器识别，没有对应的机器码

8086CPU的系统总线按功能分为三类：

- 地址总线
- 控制总线
- 数据总线

第二章 寄存器

8086是16位CPU，有14个寄存器，每个寄存器都是16位的：

- 数据寄存器（通用寄存器）
  - AX
  - BX
  - CX
  - DX
- 地址寄存器
  - SI
  - DI
  - SP
  - BP
- 段寄存器
  - CS
  - DS
  - ES
  - SS
- 控制寄存器
  - IP
  - FLAGS（PSW）

2.3 通用寄存器

如果对al, bl, cl等8位寄存器进行指令操作时产生了进位，该进位并不会保存到ah, bh, ch这些高位寄存器中

2.4 段寄存器

段寄存器就是提供段地址的。

8086CPU有4个段寄存器：

CS、DS、SS、ES

当8086CPU要访问内存时，由这4个段寄存器提供内存单元的段地址。

8086CPU是16位结构的CPU，具有一下几方面的结构特性：

- 运算器一次最多可以处理16位的数据
- 寄存器的最大宽度为16位
- 寄存器和运算器之间的通路为16位

但8086有20位地址总线，可以传送20位地址，达到1MB的寻址能力。8086CPU在内部又只能一次性处理、传输、暂时存储16位地址，实际上寻址时CPU中的相关部件会提供两个16位的地址，通过地址加法器合成为20位的物理地址：

物理地址 = 段地址*16+偏移地址

段地址*16也被成为基地址，因此物理地址 = 基地址 + 偏移地址

段的类型

8086汇编语言中把逻辑段分为四种类型，分别是代码段、数据段、附加段和堆栈段。

各段的逻辑地址对应表：

段名	段寄存器	偏移地址
代码段	**CS**	**IP**
数据段	**DS**	**BX、SI、DI**等地址寄存器
附加段	**ES**	**BX、SI、DI**等地址寄存器
堆栈段	**SS**	**SP**或**BP**

2.5 CS和IP

8086PC工作过程：

（1）从CS:IP指向内存单元读取指令，读取的指令进入指令缓冲器；

（2）IP = IP + 所读取指令的长度，从而指向下一条指令；

（3）执行指令。 转到步骤（1），重复这个过程。

2.6 代码段

对段基址的限定：只要工作在实模式，段基址必须定位在地址为**16的整数倍**上，这种段起始边界通常称做节或小段。

故，1M字节空间的20位地址的**低4位**可以不表示出来，**高16位**放入段寄存器。

对段长的限定：在实模式下段长不能超过**64K**。

从80386开始，Intel的CPU具有3种运行模式：实模式、保护模式和虚拟8086模式。

第三章 寄存器（内存访问）

存储器中数据的组织方式

- 大端方式

  高位字节排放在内存的低地址端，低位字节存放在高地址端。
- 小端方式（8086）

  低位字节存放在内存的低地址端，高位字节存放在高地址端

3.7 CPU提供的栈机制

8086CPU的入栈和出栈操作是以**字**为单位进行的

在栈中，仍遵守用两个单元存放字型数据的规则，高地址单元存放高8位，低地址单元存放低8位

任意时刻，SS:SP指向栈顶元素, 栈空状态时, SS:SP指向栈的最底部单元下面的单元

PUSH AX的执行，由以下两步完成：

(1) SP = SP - 2, SS:SP指向当前栈顶前面的单元, 以当前栈顶前面的单元为新的栈顶.

(2) 将AX中的内容送入SS:SP指向的内存单元处, SS:SP此时指向新栈顶

POP AX的执行

(1) 将SS:SP指向的内存单元的数据送入ax; 但是该处数据不会被清除

(2) SP = SP + 2

第四章 第一个程序

汇编程序从写出到执行的过程

编程 -> 1.asm -> 编译 -> 1.obj -> 连接 -> 1.exe -> 加载 -> 内存中的程序 -> 运行

4.2 程序的基本结构

程序返回：

```
MOV AX， 4C00H

INT 21H
```

这是**由CPU执行**的中断指令，可以让程序结束返回到DOS系统

第五章 [BX]和loop指令

loop指令的格式是:

loop 标号

CPU执行loop指令的时候, 要进行两步操作:

- (CX) = (CX) - 1
- 判断CX中的值, 不为0则转至标号处执行程序, 如果为0则向下执行

(1) 在CX中存放循环次数;

(2) loop指令中的标号所标识地址要在前面;

(3) 要循环执行的程序段,要写在标号和loop指令的中间

Debug和汇编编译器对指令的不同处理

在Debug中写过类似的指令: mov ax,[0] 表示将ds:0处的数据送入al中.

汇编源程序中，指令“mov ax,[0]” 被编译器当作指令“mov ax,0”处理.

我们可以在源程序中这样处理得到类似的效果:

- 先将idata MOV 进一个寄存器, 然后通过[register]的方法访问内存
- 在[idata] 前面显式地给出段寄存器

段前缀

显式地指明内存单元的段地址的 “ds:” 、 “cs:” 、 “ss:”或“es:”，在 汇编语言中称为段前缀。

一段安全的空间

一般，DOS方式下，DOS和其他合法 的程序一般都不会使用0:200~0:2FF （ 0:200h~0:2FFh ）的256 个字节 的空间。所以，我们使用这段空间是安全的。

第六章 包含多个段的程序

将数据, 代码, 栈放入不同的段

MOV ds, data是错误的, 因为data将被编译器处理成立即数, 8086CPU不允许直接将立即数送入段寄存器

第七章 更灵活的定位内存地址的方法

[idata]

直接寻址

[bx]

寄存器间接寻址

[bx + idata]

寄存器相对寻址

mov ax, [bx+200]

也可以写成:

mov ax, [200+bx]

mov ax, 200[bx]

mov ax, [bx].200

SI和DI

SI和DI不能分成两个8位寄存器来使用

[bx+si]和[bx+di]

基址变址寻址

mov ax, [bx+si]

也可以写成:

mov ax, [bx] [si]

[bx+si+idata]和[bx+di+idata]

相对基址变址寻址

mov ax, [bx+si+idata]

也可以写成

mov ax, 200[bx] [si]

多层循环

应该利用栈将外层循环的cx值暂存, 在loop语句之前再POP给cx

地址寄存器

在8086CPU中, 只有4个寄存器(bx, bp, si, di)可以用在[...]中来进行内存单元的寻址,可以单个出现, 或只能以四种组合出现:

- bx和si
- bx和di
- bp和si
- bp和di

而只要在[...]中使用寄存器bp, 而指令中没有显性的给出段地址, 段地址就默认在ss中

第八章 数据处理的两个基本问题

指令要处理的数据有多长

在没有寄存器名存在的情况下, 用操作符 X ptr 指明内存单元的长度, X 在汇编指令中可以为word 或 byte

第九章 转移指令的原理

8086CPU的转移行为有以下几类：

- 只修改IP时，称为段内转移，比如jmp ax
  - 短转移IP的修改范围为-128 ~ 127
  - 近转移IP的修改范围为-32768 ~ 32768
- 同时修改CS和IP时， 称为段间转移， 比如jmp 1000:0

操作符offset

操作符offset在汇编语言中是由编译器处理的符号， 它的功能是取得标号的偏移地址。

编译器对转移位移超界的检测

根据唯一进行转移的指令, 它们的转移范围受到转移位移的限制, 如果在源程序中出现了转移范围超界的问题, 在编译的时候, 编译器将报错

第十一章 标志寄存器

8086CPU的标志寄存器有16位, 其中存储的信息通常被称为程序状态字(PSW)

标志寄存器(简称为flag)每一位都有专门的含义. 记录特定的信息

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

flag的1, 3, 5, 12, 13, 14, 15位在8086CPU中没有使用, 不具有任何含义

溢出标志OF(Overflow Flag)	**OV(1)**	**NV(0)**
方向标志DF(Direction Flag)	DN(1)	UP(0)
中断标志IF(Interrupt Flag)	EI(1)	DI(0)
符号标志SF(SIgn Flag)	NG(1)	PL(0)
零标志ZF(Zero Flag)	ZR(1)	NZ(0)
辅助标志AF(Auxillary carry Flag)	AC(1)	NA(0)
奇偶标志PF(Parity Flag)	PE(1)	PO(0)
进位标志CF(Carry Flag)	CY(1)	NC(0)

ZF标志

flag的第6位是ZF, 零标志位, 它记录相关指令执行后, 其结果是否为0

如果结果为0, ZF = 1

结果不为0, ZF = 0

注: 8086CPU的指令集中, 有的指令的执行是影响标志寄存器的, 比如ADD/SUB/MUL/DIV/INC/OR/ADD等

有的指令对标志寄存器没有影响, 比如MOV/PUSH/POP, 它们大都是传送指令

PF标志

flag的第2位是PF, 奇偶标志位, 它记录相关指令执行后, 其结果的所有bit位中1的个数是否为偶数

如果1的个数为偶数, PF = 1

如果为奇数, PF = 0

SF标志

flag的第7位是SF, 符号标志位, 它记录相关指令执行后, 其结果是否为负

如果为负, SF = 1

如果非负, SF = 0

我们将数据当作有符号数来运算的时候, 可以通过它来得到结果的正负, 如果当作无符号数来算, SF的值则没有意义

CF标志

flag的第0位是CF, 进位标志位, 一般情况下, 在进行**无符号数运算**的时候, 它记录了运算结果的最高位有效位向更高位的进位值, 或从更高位的借位值

INC和LOOP不影响CF位

OF标志

溢出:

在进行**有符号数运算**的时候, 如结果超出了机器所能表示的范围称为溢出

对于8位的有符号数据, 机器所能表示的范围就是-128 ~ 127

对于16位的有符号数据, 机器所能表示的范围是-32768 ~ 32767

flag的第11位是OF, 溢出标志位, 一般情况下, OF记录了有符号数运算的结果是否发生了溢出

如果发生溢出, OF = 1

如果没有, OF = 0

DF标志

flag的第10位是DF, 方向标志位, 在串处理指令中, 控制每次操作后SI, DI的增减

DF = 0, 每次操作后SI, DI递增

DF = 1, 每次操作后SI, DI递减

串传送指令

使用串传送指令进行数据传送时, 需要给它提供一些必要的信息:

- 传送的原始位置: DS:SI

- 传送的目的位置: ES:DI
- 传送的长度: CX
- 传送的方向: DF

第十二章 内中断

中断源分为软件中断和硬件中断，软件中断又称为内中断，硬件中断称为外部中断。

软件中断：由CPU内部的某些事件引起的，**不受中断允许标志IF的控制**。包括以下情况：

- 除法错误, 比如执行DIV指令产生的除法溢出
- 单步执行
- 执行into指令
- 执行INT指令

硬件中断：由输入输出外设产生的中断请求引起的中断。80X86系统的硬件中断分为可屏蔽中断和不可屏蔽中断两大类。所有的中断请求都有对应的中断处理子程序与之对应。

中断处理程序

中断信息中包含有标识中断源的类型码, **中断类型码**的作用就是用来定位中断处理程序

中断向量表

中断处理程序入口地址的列表

中断向量表在内存中保存, 其中存放着**256**个中断源所对应的中断处理程序的入口

对于8086CPU, 中断向量表指定放在内存地址0处. 从内存0000:0000 到 0000:03FF的1024个单元中存放着中断向量表, 8086CPU用8位的中断类型码通过中断向量表找到对应的中断处理程序的入口地址

1个中断源 = 4个内存单元 = IP(低地址) + CS(高地址)

256个中断源分布如下:

- 5个专用中断
  - 除法出错(00000H)
  - 单步终端(00004H)
  - NMI00008H)
  - 断点中断(0000CH)
  - 溢出中断(00010H)
- 系统保留中断(27个)

  类型5 ~ 类型31 (00014H ~ 0007FH)
- 用户自定义中断(224个)

  类型32 ~ 类型255 (00080H ~ 003FCH)

中断过程

8086CPU中断过程:

（1）取得中断类型码；

（2）标志寄存器的值入栈

（3）设置标志寄存器的第8位TF 和第9位IF的值为0

（4）CS的内容入栈；

（5）IP的内容入栈；

（6）从内存地址为中断类型码*4 和中断类型码 *4+2 的两个字单元中读取中断处理程序的入口地址设置IP和CS。

编程处理中断

由于中断处理程序在任意时刻都有可能被调用, 因此中断处理程序应该放在DOS系统和其他应用程序都不会随便使用的空间中, 例如可以利用中断向量表中的空闲单元(**0000:0200 ~ 0000:02FF这256个字节**)

单步中断

CPU在执行完一条指令之后, 如果检测到标志寄存器TF位为1, 则产生单步中断, 引发中断过程, 单步中断的类型码为1, CPU在执行中断过程时, 有TF=0这个步骤, 就是为了防止在处理中断程序的时候发生单步中断

响应中断的特殊情况

- 在执行完向 ss寄存器传送数据的指令后，即便是发生中断，CPU 也不会响应。

  这样做的主要原因是，ss:sp联合指向栈顶，而对它们的设置应该连续完成。

  所以CPU在执行完设置ss的指令后，不响应中断。

  这给连续设置 ss和sp，指向正确的栈顶提供了一个时机。 即，我们应该利用这个特性，将设置ss和sp的指令连续存放，使得设置sp的指令紧接着设置ss的指令执行，而在此之间，CPU不会引发中断过程。
- **IRET**指令是中断子程序返回指令，它也要求再执行一条后续指令后才能响应中断。这样做的目的是保护系统能够正常运行；**
- **当执行到****STI**指令时，**CPU**不会马上响应中断。**STI**指令是开中断指令，要求在开放中断后再执行后续的一条指令后才能响应中断；**

第十三章 INT指令

BIOS和DOS中断例程

安装过程:

1. 开机后，CPU 一加电，初始化(CS)=0FFFFH，(IP)=0，自动从FFFF:0单元开始执行程序。

   FFFF:0处为一条转跳指令，CPU执行该指令后，转去执行BIOS中的硬件系统检测和初始化程序。
2. 初始化程序将建立BIOS 所支持的中断向量，即将BIOS提供的中断例程的入口地址登记在中断向量表中。
3. 硬件系统检测和初始化完成后，调用int 19h进行操作系统的引导。从而将计算机交由操作系统控制。
4. DOS 启动后，除完成其它工作外，还将它所提供的中断例程装入内存，并建立相应的中断向量。

一般来说，一个供程序员调用的中断例程中往往包括多个子程序，中断例程内部用传递进来的参数来决定执行哪一个子程序，BIOS和DOS提供的中断例程，都用ah来传递内部子程序的编号

例如：

mov ah， 2

int 10H

表示调用10h号中断例程的2号子程序

DOS中断

（**1）显示一个字符**

格式：AH=02H

DL=字符

INT 21H

功能：屏幕上显示一个字符，光标跟随字符移动。检验DL是否为Ctrl_Break。

（2）显示一个字符

格式：AH=06H

DL=字符

INT 21H

功能：屏幕上显示一个字符，光标跟随字符移动。不检验 Ctrl_Break。

（3）显示一串字符

格式：AH=09H

DS:DX=字符串地址

INT 21H

功能：屏幕上显示一串字符，光标跟随字符移动。要求字符串必须以$结尾。

（4）打印一个字符

格式：AH=05H

DL=字符

INT 21H

功能：把一个字符送到打印机上打印出来。

键盘功能调用

（**1**）键入一个字符并回显

格式：AH=01H

INT 21H

返回值：AL=字符的ASCII码。

（2）键入一个字符不回显

格式：AH=07H

INT 21H

返回值：AL=字符的ASCil码。不检验键入的字符是否为Ctrl_Break。

（3）键入一个字符不回显

格式：AH=08H

INT 21H

返回值：AL=字符的ASCII码。对键入的字符检验是否为Ctrl_Break

（4）键入一串字符保存到缓冲区

格式：AH=0AH

DS:DX=字节缓冲区首址

INT 21H

要求：缓冲区的第1个字节单元为允许输入的最大字符数，第2个单元为实际键入个数，从第3个单元开始存放键入字符。

（5）读键盘状态

格式：AH=0BH

INT 21H

返回值：有键入，AL=0FFH；无键入，AL=00H。

（6）清除键盘缓冲区并调用

格式：AH=0CH

AL=功能号

INT 21H

功能：清除键盘缓冲区的同时，调用键盘输入功能（1、7、8、10功能号）。使用此功能可以在输入一个字符之前将以前输入的字符从缓冲区清除。

日期、时间功能调用

1）读取系统日期

格式：AH=2AH

INT 21H

返回值：CX=年，DH=月，DL=日，AL=星期。日期值为十六进制数。

（2）设置系统日期

格式：AH=2BH

CX=年

DH=月

DL=日

AL=星期

INT 21H

返回值：AL=00，设置成功；AL=0FFH，无效。

（3）读取系统时间

格式：AH=2CH

INT 21H

返回值：CH=小时（0～23），CL=分（0～59），DH=秒（0～59），DL百分秒（0～9）。

（4）设置系统时间

格式：AH=2DH

CH=小时（0～23）

CL=分（0～59）

DH=秒（0～59）

DL=百分秒（0～99）

INT 21H

返回值：AL=00，设置成功；AL=0FFH，无效。

DOS中断例程调用

- 置光标

  mov ah, 2 ;置光标

  mov bh, 0 ;第0页

  mov dh, 5 ;dh中放行号

  mov dl, 12 ;dl中放列号

  int 10H

- 重复显示字符

  mov ah, 9 ;在光标位置显示字符

  mov al，‘a' ;显示的字符

  mov bl, 7 ;颜色属性

  mov bh, 0 ;第0页

  mov cx, 3 ;字符重复个数

  int 10h

  bl中的颜色属性格式：

  7 6 5 4 3 2 1 0

  BL R G B I R G B

  闪烁 背景 高亮 前景

*宏汇编技术

宏

宏是源程序中一段有独立功能的程序代码。宏也可以称为宏指令、宏操作。宏的使用需要经过三个步骤：宏定义、宏调用和宏展开。

宏定义

宏定义语句MACRO和子程序定义语句PROC一样都是伪指令。

```
宏名字  MACRO [哑元 1，哑元 2，… ]

            …

          语句串

            …

        ENDM
```

说明：宏定义并不产生目标代码，只是用来说明"宏名字"与一段源代码之间的联系。 其中哑元1**，哑元**2**，**...是虚拟参数或形式参数，用逗号分隔。虚参或形参可不设置。

宏与子程序

宏与子程序都是编写结构化程序的重要手段，两者各有特色。

相同之处：宏和子程序都可以定义为一段功能程序，可以被其他程序调用。

·不同之处：

（1）宏指令利用哑元和实元进行参数传递。宏调用时用实元取代哑元，避免了子程序因参数传递带来的麻烦。使宏汇编的使用增加了灵活性。

（2）实元还可以是指令的操作码或操作码的一部分，在编译汇编的过程中指令可以改变。

（3）宏调用的工作方式和子程序调用的工作方式是完全不同的。

宏的参数

在宏定义的形参表中的参数可以有多种形式，灵活使用这些参数可以实现不同功能。

1. 变元是操作数
2. 变元是操作码
3. 变元是操作码的一部分 用&连接 S&A -> A=AL 或 A=HR
4. 变元是存储单元
5. 变元是字符串

第十四章 端口

CPU可以直接读写3个地方的数据

1. CPU内部的寄存器
2. 内存单元
3. 端口

对端口的读写不能用MOV, PUSH, POP等内存读写指令

端口的读写指令只有两条:

- IN
- OUT

PC机中有一个CMOS RAM芯片, 其有如下特征:

1. 包含一个实时钟和一个有128个存储单元的RAM存储器
2. 该芯片靠电池供电

3. 128 个字节的 RAM 中，内部实时钟占用 **0～0dh**(14个单元)单元来保存时间信息，其余大部分分单元用于保存系统配置信息，供系统启动时BIOS程序读取。BIOS也提供了相关的程序，使我们可以在开机的时候配置CMOS RAM 中的系统信息。
4. 该芯片内部有两个端口，端口地址为70h和71h。

   70h为地址端口，存放要访问的CMOS RAM单元的地址；

   71h为数据端口，存放从选定的CMOS RAM单元中读取的数据，或要写入到其中的数据

CMOS RAM中存储的时间信息

在CMOS RAM中, BCD码的方式存放当前时间:

- 秒：00H
- 分：02H
- 时：04H
- 日：07H
- 月：08H
- 年：09H

这6个信息的长度都为1个字节(8位 -> 4位/BCD码 -> 2BCD码)

第十五章 内中断

在PC系统中, 外中断源一共有两类:

1. 可屏蔽中断

   CPU可以不相应的外中断:

   当CPU检测到可屏蔽中断信息时:

   - 如果IF = 1, 则CPU在执行完当前指令后响应中断, 引发中断过程
   - 如果IF = 0, 则不响应可屏蔽中断

   将IF置0，在进入中断处理程序后，禁止其他的可屏蔽中断。

   如果在中断处理程序中需要处理可屏蔽中断，可以用指令将IF 置1 。

   8086CPU提供的设置IF的指令如下:

   - STI, 用于设置IF=1
   - CLI, 用于设置IF=0

   几乎所有由外设引发的外中断，都是可屏蔽中断。
2. 不可屏蔽中断

可屏蔽终端信息来自于CPU外部, 中断类型码是通过数据总线送入CPU的; 内中断的中断类型码是在CPU内部产生的

键盘输入的处理过程：

（1）键盘产生扫描码；

按下时产生通码, 松开时产生断码. 扫码描被送入主板中60H端口的寄存器中

扫描码长度为一个字节，通码的第7 位为 0 ，断码的第7位为1。

断码 = 通码 + 80H

（2）扫描码送入60h 端口；

（3）引发9 号中断；

（4）CPU执行int 9中断例程处理键盘输入。