

汇编语言程序开发

- 机器语言、汇编语言和高级语言。
- 汇编语言是介于机器语言和高级语言之间的编程语言，是用指令助记符、符号地址、标号等符号书写程序的语言，它的指令语句与机器指令一一对应，所以说它是面向机器的。用汇编语言书写的程序叫做汇编语言源程序。
- 计算机不能直接运行汇编语言源程序，要想执行它们，必须翻译成机器指令。使用汇编语言编程能够充分利用计算机的硬件特性和操作系统底层功能，它直接利用CPU的指令系统编写程序。
- 与高级语言相比，汇编语言占用存储空间少，执行速度快。

汇编语言编程环境

- 汇编语言.**ASM**源程序通过汇编器进行汇编，生成.**OBJ**文件后，通过链接器链接生成.**EXE**的可执行文件。

MASM汇编器

名 称	简 介	环 境
MASM	微软公司出品，持续升级，稳定性好	DOS，Windows
TASM	Borland 公司出品，工具包完整	DOS，Windows
MASM32	集成了微软的工具，完整的头文件，IDE 环境	Windows
GASM	GNU Assembler	各种平台
Pass32	支持 DOS 扩展器	DPML，Windows
VisualASM	附带 IDE 环境	Windows

常用的MASM选项

选 项	功 能
/c	仅进行编译，不自动进行链接
/coff	产生的 obj 文件格式为 COFF 格式
/Cp	源程序中区分大小写
/Fo filename	指定输出的 obj 文件名
/Fl [filename]	产生.lst 列表文件
/I pathname	指定 include 文件的路径
/link	指定链接时使用的选项

LINK链接器

选 项	功 能
/out:输出文件名	输出的文件名，扩展名默认为.exe
/map:文件名	生成 MAP 文件
/libpath:目录名	指定库文件的目录路径
/implib:文件名	指定导入库文件
/entry:标号	指定入口
/comment:字符串	在 PE 文件的文件头后面加上文本注释（版权信息）
/stack:数字	设定堆栈的大小
/subsystem:系统名	指定程序运行的环境，可以是以下几种之一： Native, Windows, Console, Windowsce, Posix

汇编链接步骤

- (1) 用MASM汇编hello.asm
- ml /c /coff hello.asm
- 这里使用/c选项表示只生成.obj文件而不直接产生.exe 文件； /coff选项要求MASM生成链接器所需要的COFF格式的.obj文件。
- (2) 用LINK链接hello.obj
- link /subsystem:console hello.obj
- /subsystem选项表示程序的运行环境，一般指定为“windows”； 当编写控制台（Console）程序的时候，要指定为“console”。
- 合二为一：
- ml /coff hello.asm /link /subsystem:console

汇编语言语句格式

- 汇编语言程序中的语句可以分为指令、伪指令和宏指令3种
- **每一条**指令语句都要生成机器代码，各对应一种CPU操作
- 伪指令语句（简称伪指令）提供给汇编程序信息，它由汇编程序在汇编过程中执行，除了数据定义语句分配存储空间外，其他伪指令**不生成**目标码
- 宏指令是由用户按照宏定义格式编写的一段程序，其中可以包含指令、伪指令甚至于**另一条**宏指令

- 汇编语言语句对大小写不敏感，它由名字、助记符、参数和注释4部分组成，其格式如下：
- [名字] 助记符 < 操作数 > [;注释]
- 名字域是语句的符号地址
- 数字不能出现在名字的第一个字符位置
- 名字作为符号地址具有3个属性，即段基址、偏移量和类型。标号的类型有NEAR型和FAR型，变量的类型有字节、字、双字、四字等。
- 注释以分号（;）开始

常用伪指令

类 型	助记符	简写	字节数	可表示的数字范围
字节	BYTE	DB	1	0~255
字	WORD	DW	2	0~65 535
双字	DWORD	DD	4	0~4 294 967 295
远字	FWORD	DF	6	通常作为 48 位全指针变量
四字	QWORD	DQ	8	
十字节	TBYTE	DT	10	
带符号字节	SBYTE		1	-128~+127
带符号字	SWORD		2	-32 768~+32 767
带符号双字	SDWORD		4	-2 147 483 648~+2 147 483 647

注：在 DOS 下多用 DB、DW、DD 等简写格式。在 Windows 汇编中，往往使用全称 byte、word、dword 等来定义数据，这种格式只有高版本的 MASM（例如 MASM 6.x）才支持。

- ① 数字常量及数值表达式。可以有十进制、二进制、十六进制、八进制数字常量，常用前3种格式。
- 十进制数：以D结尾，汇编语言中默认值是十进制数，所以D可以省略不写。有效数字是0~9。
- 二进制数：以B结尾，有效数字是0、1。例如：10100011B，10100011b。
- 十六进制数：以H结尾，有效数字是0~9和A~F。若第一位数字是字母形式，则必须在前边加上0。例如：0fe08h，16H。
- 八进制数：以Q或O结尾，有效数字是0~7。例如352Q。

- ② 字符串常量。在汇编语言中字符需要用单引号括起来，其值为字符的ASCII码。因为每个字符占用一个字节，所以最好用DB助记符定义字符串。例如：'A'的值为41H；'abc'的值为616263H。

- ③ 地址表达式。操作数可以是地址符号。若只定义符号的16位偏移量，则使用DW助记符；若要定义它的双字长地址指针（既含16位偏移量又含段基址），则使用DD助记符，其中低字中存放偏移量，高字中存放段基址；若要定义它的48位全地址指针（既含32位偏移量又含段选择符），则使用DF助记符，其中低32位存放偏移量，高16位存放段选择符。
- VAR DW LAB ;语句在汇编后VAR中含有LAB的16位偏移量。

- ④ ? 。在程序中使用“?”为变量预留空间而**不赋**初值。
- ⑤ <n> DUP (操作数, ……) 。若要对某些数据重复多次, 可以使用这种格式。其功能是把 () 中的内容复制n次。DUP可以**嵌套**。

数据定义的程序片段

- M1 DB 15,67H,11110000B,?
- M2 DB '15', 'AB\$'
- M3 DW 4*5
- M4 DD 1234H
- M5 DB 2 DUP(5, 'A')
- M6 DW M2 ;M2的偏移量
- M7 DD M2 ;M2的偏移量、段基址
- 汇编后其值为: 0F 67 F0 00 31 35 41 42 24 14 00 34 12 00 00 05
41 05 41 04 00 04 00 XX XX

- 可以直接通过变量名引用这些变量
- MOV AL, M1 ; (AL) = 15
- MOV BX, M3 ; (BX) = 20, 经汇编后已经计算出4*5的值
- ADD M3, 6 ; (M3) = 26
- MOV AL, M2 ; (AL) = '1' = 31H
- MOV BL, M2+2 ; (BL) = 'A' = 41H
- MOV M1+3, 5 ; (M1+3) = 5
- MOV CL, M2-3 ; 把M1中的67H送给CL
- M2+2不是把M2变量中的内容加2, 而是指M2的地址加2, 仍然表示一个地址

符号定义伪指令

- 等值伪指令
- 程序中有时会多次出现同一个表达式，为方便起见可以用符号定义伪指令给该表达式定义一个符号，以便于引用及减少程序修改量，并提高程序的可读性。汇编后该符号代表一个确定的值，该符号定义伪指令称为等值伪指令EQU。
- 格式：符号名 EQU 表达式
- 功能：用符号名代表表达式或表达式的值。
- 说明：表达式可以是任何有效的操作数格式。它可以是常数、数值表达式、另一符号名或助记符。
- 注意用EQU定义的符号在同一个程序中不能再定义。

- CR EQU 0DH ;用CR表示回车符的ASCII码
- LF EQU 0AH ;用LF表示换行符的ASCII码
- PORT_B EQU 61H ;用PORT_B表示B端口61H
- B EQU [BP+6] ;用B表示操作数[BP+6]
- L1: MOV AL, CR ;执行后 (AL) = 0DH
- L2: ADD BL, B ; (BL) = (BL) + (SS:[BP+6])
- L3: IN AL, PORT_B ;从61H端口输入一个字节数据
- L4: OR AL, 00000010B ;把D₁位置1
- L5: OUT PORT_B, AL ;再输出到61H端口

- 在程序中有多处使用PORT_B，如果端口号改变，则只需修改EQU语句，而程序中大量的引用则不必改动。
- 用于与\$操作符配合，得到变量分配的字节数，字符串改变而串长改变时，取串长的语句无需做任何修改
- MSG DB 'This is first string. '
- COUNT EQU \$-MSG
- MOV CL, COUNT ; (CL) =MSG的串长=21

- 等号伪指令
- 格式：符号名 = 数值表达式
- 功能：用符号名代替数值表达式的值。
- 说明：等号伪指令“=”与EQU伪指令功能相似，其区别是等号伪指令的表达式只能是常数或数值表达式，另外用定义的符号在同一个程序中可以再定义。通常在程序中用“=”定义常数。
- DPL1=20H
- K=1
- K=K+1

操作符伪指令

- \$操作符
- 功能：\$在程序中表示当前地址计数器的值。如果一行语句占用了存储空间，如定义变量或者是指令，则地址计数器就会增加，增加的字节数就是变量或指令所占的字节数。
- `wVar WORD 0102h, 1000, 100*100`
- `BYTESOFWVAR EQU $-wVar`
- `MOV EAX, $`
- 这条指令中的\$被替换成该指令所在的地址，如：
- `00401010 B8 10 10 40 00 MOV EAX, 00401010`

- ORG操作符
- 格式：ORG 数值表达式
- 功能：设置地址计数器内容为数值表达式的值。
- 说明：在汇编程序对源程序汇编的过程中，使用地址计数器保存当前正在汇编的语句地址（段内偏移量），汇编语言允许用户直接用“\$”引用地址计数器的当前值。
- ORG 100H ;设置地址计数器的值为100H
- ORG \$+6 ;跳过6个字节的存储区域

- OFFSET操作符
- 格式：OFFSET [变量|标号]
- 功能：OFFSET操作符用来取出变量或标号的地址（在段中的偏移量）。在32位编程环境中，地址是一个32位的数。
- MOV EBX, DVAR2
- MOV EBX, OFFSET DVAR2
- 在内存中，上面两条指令分别表示为：
- 00401010 8B 1D 2A 40 40 00 MOV EBX, [0040402A]
- 00401016 BB 2A 40 40 00 MOV EBX, 0040402A

算术操作符

- 算术操作符包括+（加），-（减），*（乘），/（除）和MOD（模）操作符。其中/表示整除，即只取商的整数部分；而MOD则表示只取余数。
- MOV EAX, 4*5
- MOV EAX, offset dVar2-10
- MOV EAX, 30/8
- MOV EAX, 30 MOD 8

逻辑操作符

- 逻辑操作符包括AND（逻辑与）、OR（逻辑或）、XOR（逻辑异或）和NOT（逻辑非）。逻辑操作符是按位操作的。
- MOV EAX, 4 AND 5
- MOV EAX, 01000001b XOR 01000011b
- MOV EAX, NOT 1

关系操作符

- 关系操作符包括EQ（等于）、NE（不等于）、LT（小于）、LE（小于等于）、GT（大于）、GE（大于等于）。这些操作符对其前后的两个操作数进行比较，其操作结果为一个逻辑值，若关系成立，结果为真（全部二进制位为1），否则结果为假（0）。其中的操作数必须是常数或常数表达式。
- MOV EAX, 0 LT -1 ; 结果EAX=00000000H
- MOV EAX, 0 GE -1 ; 结果EAX=0FFFFFFFFH
- MOV AX, 8 NE 8 ; 结果AX=0000H
- MOV AX, 8 EQ 8 ; 结果AX=0FFFFH

框架定义伪指令

- 以“.”开始，一般用在32位汇编语言程序框架中。
- 微处理器伪指令
- 使用微处理器伪指令说明本程序使用哪一种CPU的指令集。汇编程序在默认情况下只接受8086的指令系统，即使在Pentium上也是如此，因此在8086上编写的程序在较新的处理器上都可以顺利执行。为了能够使用其他微处理器或协处理器的指令系统编写软件，需要在程序中增加选择微处理器的伪指令。
- Pentium的指令集包括了80486的指令集，而80486的指令集又包括了80386的指令集。
- 如果用汇编语言编写的是驱动程序或者驱动程序的一个小模块，因为驱动程序是在特权级0上运行，因此就需要使用.386p。

伪指令格式	功 能
.286	选择 80286 微处理器指令系统
.386	选择 80386 微处理器指令系统
.486	选择 80486 微处理器指令系统
.586	选择 80586 微处理器指令系统
.386p	选择保护模式下的 80386 微处理器指令系统，表示程序中使用特权指令
.486p	选择保护模式下的 80486 微处理器指令系统，表示程序中使用特权指令
.586p	选择保护模式下的 80586 微处理器指令系统，表示程序中使用特权指令
.8087	选择 8087 数字协处理器指令系统
.287	选择 80287 数字协处理器指令系统
.387	选择 80387 数字协处理器指令系统
.mmx	选择 MMX 指令集

框架定义伪指令

伪指令格式	功 能
.DATA	定义数据段
.DATA?	定义存放未初始化变量的数据段
.CONST	定义存放常量的数据段
.CODE	定义代码段
.STARTUP	指定加载后的程序入口点
.EXIT	返回 DOS 或父进程
.STACK size	建立一个堆栈段并定义其大小（size 以字节为单位。若不指定 size 参数，则使用默认值 1 KB）。
.MODEL 内存模式[,调用规则][,其他模式]	定义程序工作的模式

框架定义伪指令

- 在Windows中，实际上只有代码区和数据区之分。`.DATA`、`.DATA?`、`.CONST`、`.STACK`都视为数据区。用`.DATA`、`.DATA?`、`.CONST`定义的都是数据段。
- 可以用`.CODE`定义代码区，用`.DATA`定义数据区。堆栈空间一般是系统自动分配的，用户程序不必考虑。
- **注意：**当`.386`等选择微处理器伪指令出现在`.MODEL` 伪指令之前时，**不能**使用`.STARTUP`和`.EXIT`，否则汇编时会出错。

- 程序内存模式的定义影响最后生成的可执行文件，在DOS的可执行程序中，有大小限制在64KB以下的.COM文件，也有可以超过64KB的.EXE文件。到了Windows环境下，PE格式的可执行文件最大可以用4GB内存。编写不同类型的可执行文件要用.MODEL语句定义不同的参数。

汇编源程序格式

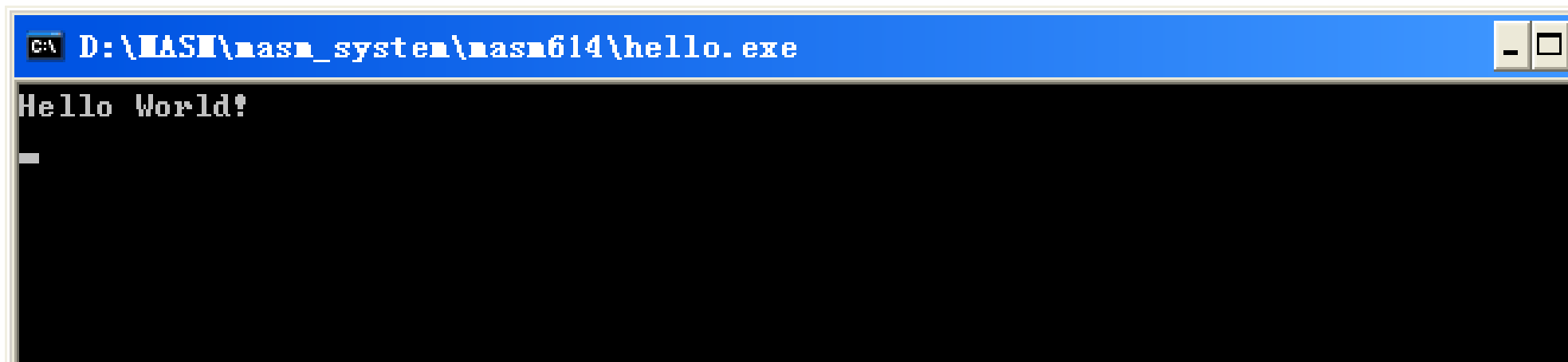
- CUI和GUI
- CUI
 - 生成CUI结构下的.EXE文件时，程序的运行环境要设置为console。
- GUI
 - 在生成.EXE文件时，程序的运行环境要设置为Windows。

控制台界面的汇编源程序

- 编写一个Windows控制台界面汇编程序，在控制台上显示一个字符串“Hello World!”
- .386
- .model flat, stdcall
- option casemap:none
- ;说明程序中用到的库、函数原型和常量
- includelib msvcrt.lib
- printf PROTO C :ptr sbyte, :VARARG
- ;数据区
- .data
- szMsg byte "Hello World! ", 0ah, 0
- ;代码区
- .code
- start:
- invoke printf, offset szMsg
- ret
- end start

汇编过程

- `ml /coff hello.asm /link /subsystem:console`
- 注意：在subsystem选项中必须指定`console`，而不是windows。
- 输入以下命令执行该程序：
- `D:\MASM\masm_system\masm614>hello`



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\ D:\MASM\masm_system\masm614\hello.exe" and standard window control buttons. The main area has a black background with white text. The first line displays "Hello World!". A white cursor is positioned on the line below.

```
C:\ D:\MASM\masm_system\masm614\hello.exe
Hello World!
_
```

等价的C程序

- `/* PROG0402.c*/`
- `#include <stdio.h>`
- `int main()`
- `{`
- `printf("Hello World!\n");`
- `return 0;`
- `}`
- C程序中字符串自动由编译器加上一个0字符作为字符串的结束，而汇编程序中必须在字符串后面**明确**的定义一个值为0的字节。

<pre>.386 .model flat,stdcall option casemap:none</pre>	模式定义
<pre>includelib msvcrt.lib printf PROTO C :ptr sbyte, :VARARG</pre>	库文件及函数声明
<pre>.data szMsg byte 'Hello World!', 0ah, 0</pre>	数据部分
<pre>.code start: invoke printf, OFFSET szMsg ret end start</pre>	代码部分

模式定义

- `.386`
- `.model flat, stdcall`
- `option casemap:none`
- (1) `.386`语句定义了程序使用80386指令集。
- (2) `.model flat, stdcall`语句。

- 因为Windows操作系统和应用程序运行在保护模式下，系统把每一个Windows可执行程序都放在一个虚拟地址空间中去运行，每一个程序都拥有其相互独立的4GB地址空间。因此，Windows可执行程序只有一种内存模式，即flat模式，表示内存是很“平坦”的，从00000000H延伸到0FFFFFFFFH，不再像DOS可执行程序那样，必须把超过64KB的一块内存划分为几个小的、不超过64 KB的段来使用。

- 在DOS下的汇编语言程序中：
- `MOV AX, DATA`
- `MOV DS, AX`
- 其作用是给数据段寄存器DS赋值，指向程序自己的名为DATA的数据段。每一个程序都有它自己的数据段。
- 在Windows汇编语言程序中，则不必考虑这些问题。这些步骤都已经由操作系统安排好了。

- 如果定义了.model flat, 则MASM自动为各种段寄存器做如下定义:
- ASSUME CS:FLAT, DS:FLAT, ES:FLAT, SS:FLAT, FS:ERROR, GS:ERROR
- CS、DS、ES、SS段全部使用平坦模式, FS和GS段不能使用。
- 需要使用FS和GS, 则需用下面的语句说明:
- ASSUME FS:NOTHING, GS:NOTHING
- 或: ASSUME FS:FLAT, GS:FLAT

- Windows API使用stdcall调用规则，在.model中指定。
- 使用stdcall规则调用子程序时，堆栈平衡的事情由被调用者（子程序）用RET n指令实现，因此在程序中调用Windows API函数或子程序后，不必调用者考虑堆栈平衡问题。

option语句

- 在Win32中需要定义option casemap:none，用以说明程序中的变量和子程序名是否对大小写敏感。
- 由于Windows API中的函数名称是区分大小写的，所以应该指定这个选项“casemap:none”，否则在调用函数的时候会出现问题。

include lib语句

- 用于在汇编程序中调用一些外部模块（子程序/函数）来完成部分功能。
- 例如，printf函数属于C语言的库函数。它的执行代码放在一个名字叫msvcrt.dll动态连接库DLL中，ms代表Microsoft，vc代表Visual C/C++，rt代表run time。msvcrt.dll是Windows自带的。
- msvcrt.lib是和msvcrt.dll相配套的一个库文件，用以指出库函数的位置。这种库文件也叫导入库（Import Library）。
- 一个DLL文件对应一个导入库，如msvcrt.dll的导入库是msvcrt.lib；kernel32.dll的导入库是kernel32.lib；user32.dll的导入库是user32.lib等。

函数声明语句

- 对于所有要用到的库函数，在程序的开始部分必须预先声明。包括函数的名称，参数的类型等
- 函数声明为：
- 函数名称 PROTO [调用规则] : [第一个参数类型] [,:后续参数类型]
- printf的函数声明：
- `_CRTIMP int __cdecl printf(const char *, ...)` ;
- printf函数的调用规则为C调用规则（__cdecl, 即c declare），第一个参数是字符串指针，后面的参数数量及类型不定。如果函数使用C调用规则，则PROTO后跟一个C。接下来是参数的说明。如果参数个数、类型不定，则用VARARG说明（variable argument）。

- 在汇编语言中，用ptr sbyte代表const char *
- 函数声明后，用INVOKE伪指令来调用。

include语句

- 在汇编语言编程中，也可以采用C语言办法，就是把所有的函数声明及常量定义等公用部分预先放在一个头文件中。一些汇编语言工具包提供了这样一些头文件。
- 例如：
- include kernel32.inc
- include user32.inc

数据和代码部分

- 程序中的数据部分从.data语句开始定义，代码部分从.code语句开始定义
- 所有的指令都必须写在代码区中。
- 对于运行在特权级3的应用程序来说，代码区是不可写的。在编程时，**不能**把那些需要修改的变量放到.code部分。

程序结束

- 程序在遇到end语句时结束。end语句后面跟的标号指出了程序执行的入口点，即装入执行的**第一条**指令的位置，表示源程序结束。
- END [过程名]
- 过程名指示程序执行的**起始**地址。[]中的过程名是可选的。只有主过程模块的END后可以带过程名，而且这个过程名必须是主程序的名字。若一个程序由多个模块组成，则除主模块外，其他模块的END语句**不能**带过程名。

跨行语句

- 当源程序的某一语句过长，可以用反斜杠“\”做换行符，将这条语句分为几行来写。每一行的最后加上一个反斜杠，说明下面一行是当前行的继续。
- 语句的最后一行不要加反斜杠。

程序中的数据归类

- (1) 可读可写的初始变量
- 这些数据在源程序中给出初始值，在程序的执行中可能被更改。这些数据必须定义在.data区中。在程序执行以前，内存中的变量就具有在源程序中指定的初始值。
- 保存在可执行文件中
- (2) 可读可写的未初始变量
- 这些变量一般是当做缓冲区或者在程序执行后才开始使用的，在程序中不需要给它们指定初值。
- 这些数据可以定义在.data节中，也可以定义在.data?节中。不需要在可执行文件中占用空间。

- (3) 常量数据
- 如一些要显示的字符串信息，它们在程序装入的时候已具有初值，在整个程序执行过程中不需要修改，这些数据可放在.const部分中。在这个部分的变量，只能读不能写。在程序中出现了对.const部分的变量做写操作的指令，会引起异常，操作系统会报告并结束程序。
- 常量数据也可以放在.data定义的段中，如例中的szMsg。

invoke伪指令

- 格式：invoke 函数名[,参数1] [,参数2]...
- 功能：调用函数或子程序。
- 与在DOS中使用中断调用方式调用系统功能一样，在Windows中用API方式调用存放在DLL中的函数。由于API函数的参数较多，为了简化工作，可以使用invoke伪指令。

- invoke不是指令而是伪指令。是MASM为了方便调用而提供的，是一种主程序调用子程序的简化方法。在汇编时会把它按照调用规则展开成相应的指令，通常展开成几条PUSH指令（有几个参数就展开几条）和一条CALL指令。对于像C那样要求调用程序清除堆栈中的参数者，最后还会展开一条add esp, n指令。在子程序调用不带参数的情况下，它等价于CALL指令。

invoke伪指令的展开

- ;代码段
- 00000000 .code
- 00000000 start:
- 00000000
- invoke printf, offset szMsg (注： 此处展开)
- 00000000 68 00000000 R * push dword ptr offset flat:szMsg
- 00000005 E8 00000000 E * call printf
- 0000000A 83 C4 04 * add esp, 000000004h
- 0000000D C3 ret
- 0000000E
- end start

```

.386
.model    flat, stdcall
option    casemap:none

MessageBoxA    PROTO    :dword, :dword, :dword, :dword
MessageBox     equ      <MessageBoxA>
Includelib     user32.lib
NULL           equ      0
MB_OK          equ      0

.stack    4096

.data
SzTitle       byte      'Hi!', 0
SzMsg         byte      'Hello World!' ,0

.code
start:

                invoke    MessageBox,
                        NULL,                ; HWND hWnd
                        offset SzMsg,        ; LPCSTR lpText
                        offset SzTitle,     ; LPCSTR lpCaption
                        MB_OK               ; UINT uType

                ret

end            start

```

说明

- 程序中使用的MessageBox属于user32.dll，是Windows的一个API函数。其功能是显示一个消息框。它的第1个参数是一个窗口句柄，即消息框的父窗口。这里使用NULL表示它没有父窗口。第2个参数是一个字符串指针，指向在消息框中显示的正文。第3个参数也是一个字符串指针，指向在消息框的窗口标题。第4个参数是一个整数，指定消息框的类型，这里使用MB_OK，消息框中显示一个OK（“确定”）按钮。

- 输入以下命令生成可执行文件hellow.exe:
- `ml /coff hellow.asm /link /subsystem:windows`
- 注意在subsystem选项中必须指定windows，而不是console。



输入输出有关的Windows API函数

- 对于所有要用到的库函数（或Windows API函数），在程序的开始部分必须预先声明。包括函数名称、参数类型等。
- 函数名称PROTO [调用规则] :[第一个参数类型] [:后续参数类型]
- 调用规则是可选项，可以是stdcall，也可以是C等。**缺省**时使用model语句中指定的调用规则。
- 如果函数使用C调用规则，则PROTO后跟一个**C**。
- 参数的说明中如果参数个数、类型不定，则用**VARARG**说明。
- printf和scanf适用于**控制台**程序（连接选项为/subsystem:console）；MessageBox适用于Windows风格的**窗口界面**程序（连接选项为/subsystem:windows）。

printf

- printf是一个实现输出的API函数。在程序调用时应指明printf的调用规则，以及它的参数类型。在C语言头文件stdio.h中printf的函数声明为：
- `_CRTIMP int __cdecl printf(const char *, ...);`
- 调用规则为C调用规则（__cdecl, 即c declare），第一个参数是字符串指针，后面的参数数量及类型不定。
- 在汇编语言中，用ptr sbyte代表const char *。

- `printf` `PROTO C :ptr sbyte,:vararg`
- 实际上调用时只注重它的类型，并不关心其名称，因此在程序中参数类型经常用 `DWORD` 来表示，它可以代表字符串指针、结构指针、整数等。例如 `printf` 也可以声明为：
- `printf` `PROTO C :dword,:vararg`
- `printf` 及其他 `msvcrt.dll` 输出的函数的连接信息都在这个库文件中。因此在程序开头应有以下语句：
- `include lib msvcrt.lib`

- invoke printf, offset szOut, x, n, p
- 其中, szOut要在数据区中定义, 例如:
- szOut byte 'x=%d n=%d x(n)=%d', 0ah, 0
- 其效果等价于:
- printf ("x=%d n=%d x(n)=%d\n" , x, n, p);

scanf

- scanf的连接信息也包括在msvcrt.lib库文件中。
- scanf的调用规则和参数类型说明为：
- scanf PROTO C :dword,:vararg
- 第1个参数是格式字符串的地址，后面的参数个数可变，可以一个没有

- szInFmtStr byte '%d %c %d', 0
- invoke scanf, offset szInFmtStr, offset a, offset b, offset d
- 其中， 第1个参数是格式字符串szInFmtStr的地址， 第2、3、4个参数分别是a、b、d的地址。其效果等价于：
- scanf("%d %c %d", &a, &b, &d);

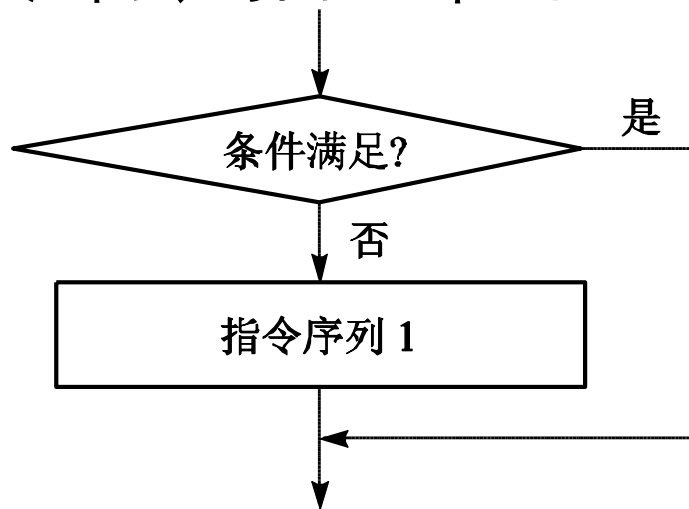
MessageBoxA

- MessageBoxA适用于Windows风格的窗口界面程序（连接选项为/subsystem: windows），实现在窗口中显示信息。
- 在汇编程序中需要使用MessageBoxA函数时的声明语句。
- `MessageBoxA PROTO stdcall :dword,:dword,:dword,:dword`
- MessageBoxA连接信息包含在user32.lib库文件中，因此在程序的开始应该增加语句：
- `includelib user32.lib`

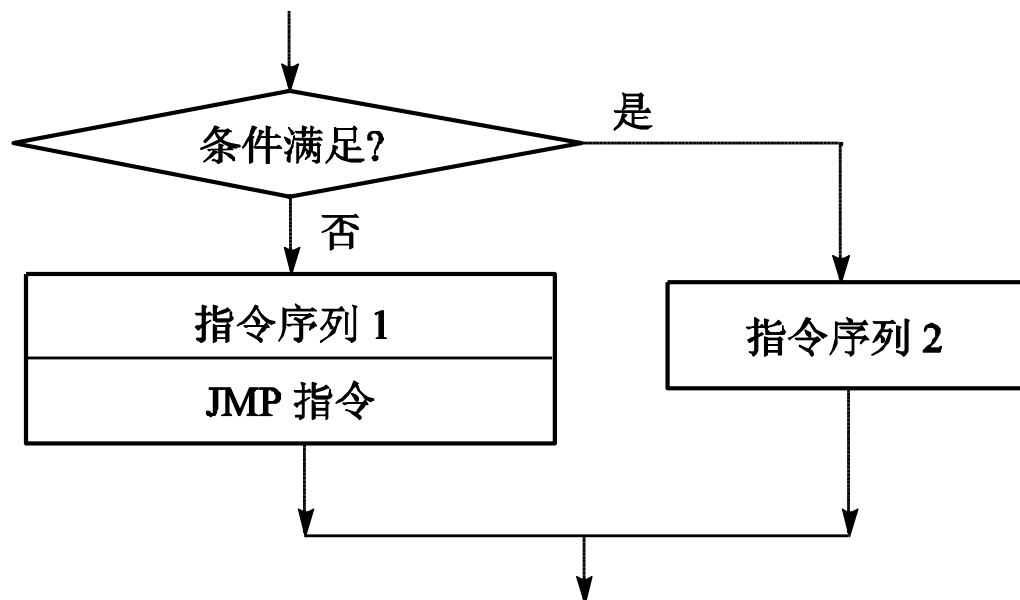
- 在编程中，应尽可能地利用已有的C库函数和Windows API函数，一般都可以转换为API函数来实现。通常，C库函数使用C调用规则，Windows API采用`stdcall`调用规则。
- 函数可能返回的是一个整数或者是一个指针。无论是什么，返回值都在`EAX`中。

分支与循环程序设计

- IF_THEN_ELSE结构分支程序设计
- 对于具有图（a）结构的程序，当条件满足时跳过指令序列实现转移，条件不满足时继续向下执行指令序列1。
- 对于具有图（b）结构的程序，当条件满足时转去执行指令序列2，条件不满足时继续向下执行指令序列1，无论执行哪个分支，最终都会到同一个出口。



(a)



(b)

单分支结构

- 求带符号数A和B的较大值 $MAXAB = MAX(A, B)$
- `MOV EAX, A`
- `CMP EAX, B`
- `JGE AlsLarger` ;如果 $A \geq B$, 跳转到AlsLarger标号处
- `MOV EAX, B`
- `AlsLarger:`
- `MOV MAXAB, EAX`

无符号类比

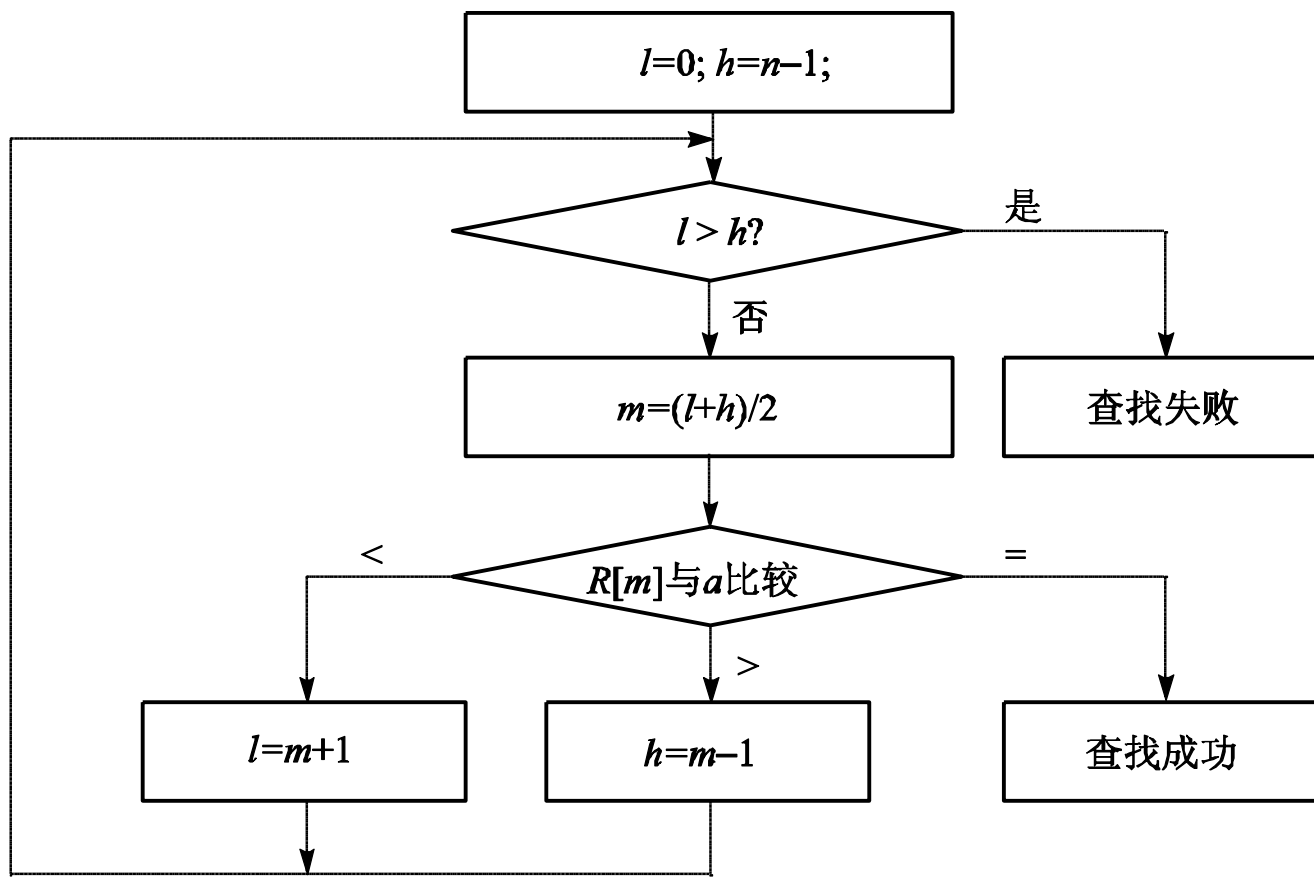
- 求无符号数A和B的较大值 $MAXAB = MAX(A, B)$
- `MOV EAX, A`
- `CMP EAX, B`
- `JAE AlsAbove` ; 如果 $A \geq B$, 跳转到AlsAbove标号处
- `MOV EAX, B`
- `AlsAbove:`
- `MOV MAXAB, EAX`

IF_THEN_ELSE结构程序举例

- 求带符号数X的符号，如果 $X \geq 0$ ，把SIGNX置为1；如果 $X < 0$ ，把SIGNX置为-1。
- X SDWORD 45
- SIGNX SDWORD ?
- MOV SIGNX, 0
- CMP X, 0
- JGE XisPostive ;X ≥ 0 ，跳转
- MOV SIGNX, -1
- JMP HERE ;跳过“MOV SIGNX, 1”语句
- XisPostive:
- MOV SIGNX, 1
- HERE:

在升序数组中查找一个数 (采用折半查找算法)

- 设数组名为dArray，数组字节型，下标为EBX，在程序中用dArray[EBX]来表示下标为EBX的元素。



之一

- .386
- .model flat,stdcall
- option casemap:none
- includelib msvcrt.lib
- printf PROTO C :dword,:vararg
- .data
- dArray byte 15, 27, 39, 40, 68, 71, 82, 100, 200, 230
- Items equ (\$-dArray) ;数组中元素的个数
- Element byte 82 ;在数组中查找的数字
- Index dword ? ;在数组中的序号
- Count dword ? ;查找的次数
- szFmt byte 'Index=%d Count=%d Element=%d', 0ah, 0 ; 格式字符串
- szErrMsg byte 'Not found, Count=%d Element=%d', 0ah, 0

之二

- .code
- start:
 - xor eax, eax
 - mov Index, -1 ;赋初值, 假设找不到
 - mov Count, 0 ;赋初值, 查找次数为0
 - mov esi, 0 ;ESI表示查找范围的下界
 - mov edi, Items-1 ;EDI表示查找范围的上界
 - mov al, Element ;EAX是要在数组中查找的数字
- Compare:
 - cmp esi, edi ;下界是否超过上界
 - jg notfound ;如果下界超过上界, 未找到

之三

- mov ebx, esi ;取下界和上界的中点
- add ebx, edi ;
- shr ebx, 1 ; $EBX = (ESI + EDI) / 2$
- inc Count ;查找次数加1
- cmp al, dArray[EBX] ;与中点上的元素比较
- jz Found ;相等, 查找结束
- ja MoveLow ;较大, 移动下界
- mov edi, ebx ;较小, 移动上界
- dec edi ;EDI元素已比较过, 不再比较
- jmp Compare ;范围缩小后, 继续查找

之四

- MoveLow:

- `mov esi, ebx`

;较大, 移动下界

- `inc esi`

;ESI元素已比较过, 不再比较

- `jmp Compare`

;范围缩小后, 继续查找

之五

- Found:
- mov Index, ebx ;找到, EBX是下标
- xor eax, eax
- mov al, dArray[ebx]
- invoke printf, offset szFmt, Index, Count, eax
- ret
- NotFound:
- invoke printf, offset szErrMsg, Count, eax
- ret
- end start

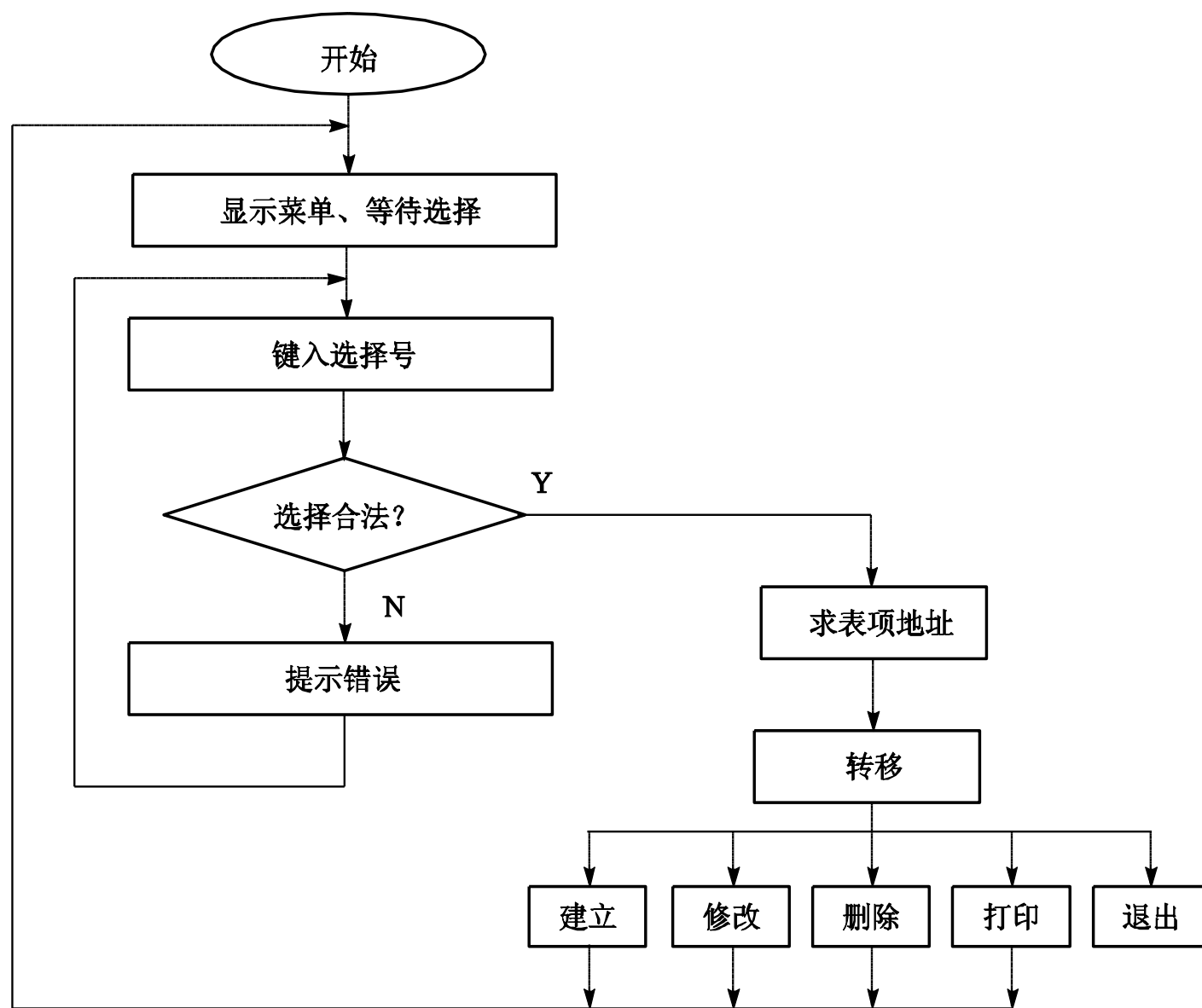
SWITCH_CASE结构分支程序设计

- 编制一个管理文件的菜单程序，要求能够实现建立文件、修改文件、删除文件、显示文件和退出应用程序5个主控功能。首先在屏幕上显示5种功能，然后从键盘上输入数字1~5即可转入相应的功能，而输入其他字符则提示输入非法。若选择退出功能，则能正确返回；若选择其他功能，应能返回到主菜单。

- 对于SWITCH_CASE结构，由于分支众多，可以把各分支入口地址集中在一起构成一个地址表，把这个地址表称为跳转表。设建立文件分支入口标号为CR，修改文件分支入口标号为UP，删除文件分支入口标号为DE，显示文件分支入口标号为PR，退出分支入口标号为QU

- JMPTAB DD OFFSET CR ;跳转表
- DD OFFSET UP
- DD OFFSET DE
- DD OFFSET PR
- DD OFFSET QU

- 索引号=K-起始功能号（例如功能号为1, 2, 3, ..., N, 则索引号=K-1）。
- 位移量=索引号×每项入口地址占用的字节数。对于用DD定义的则为4字节。
- 表项地址=表基址+位移量。



之一

- .386
- .model flat,stdcall
- option casemap:none
- includelib msvcrt.lib
- printf PROTO C:ptr sbyte,:vararg ; 用法: printf(str);
- scanf PROTO C:ptr sbyte,:vararg ; 用法: scanf("%d", &op);
- .data
- Msg1 db '1——create',0ah ; 菜单字符串
- db '2——update',0ah
- db '3——delete',0ah
- db '4——print',0ah
- db '5——quit',0ah,0

之二

- Msg2 db 'input select:',0ah,0 ;提示字符串
- Fmt2 db '%d',0 ;scanf的格式字符串
- op dd ? ;scanf结果(用户输入的整数)
- Msg3 db 'Error!',0ah,0 ;输入错误后的显示的字符串
- MsgC db 'Create a File',0ah,0ah,0 ;选择菜单1后的显示的字符串
- MsgU db 'Update a File',0ah,0ah,0 ;选择菜单2后的显示的字符串
- MsgD db 'Delete a File',0ah,0ah,0 ;选择菜单3后的显示的字符串
- MsgP db 'Print a File',0ah,0ah,0 ;选择菜单4后的显示的字符串
- MsgQ db 'Quit',0ah,0 ;选择菜单5后的显示的字符串
- JmpTab dd offset cr ;跳转表，保存5个标号
- dd offset up
- dd offset de
- dd offset pr
- dd offset qu

之三

- .code
- start:
- invoke printf,offset Msg1 ;显示菜单
- Rdkb:
- invoke printf,offset Msg2 ;显示提示
- invoke scanf,offset Fmt2,offset op ;接收用户的输入
- cmp op,1 ;与1比较
- jb Beep ;输入的数字比1小, 不合法
- cmp op,5 ;与5比较
- ja Beep ;输入的数字比5大, 不合法
- mov ebx,op
- dec ebx ;转换为跳转表的索引
- mov eax, JmpTab[ebx*4] ;得到表项地址
- jmp eax ;按表项地址转到对应的标号处
- *jmp JmpTab[ebx*4]* ;可以用这一指令替换上面两条

- Beep:

- ```

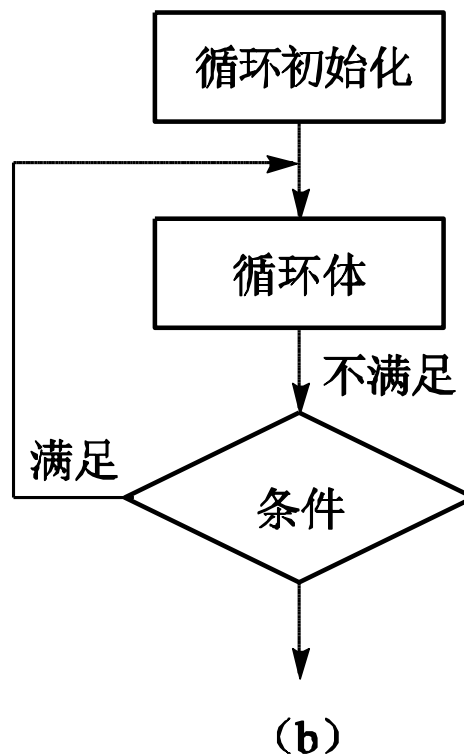
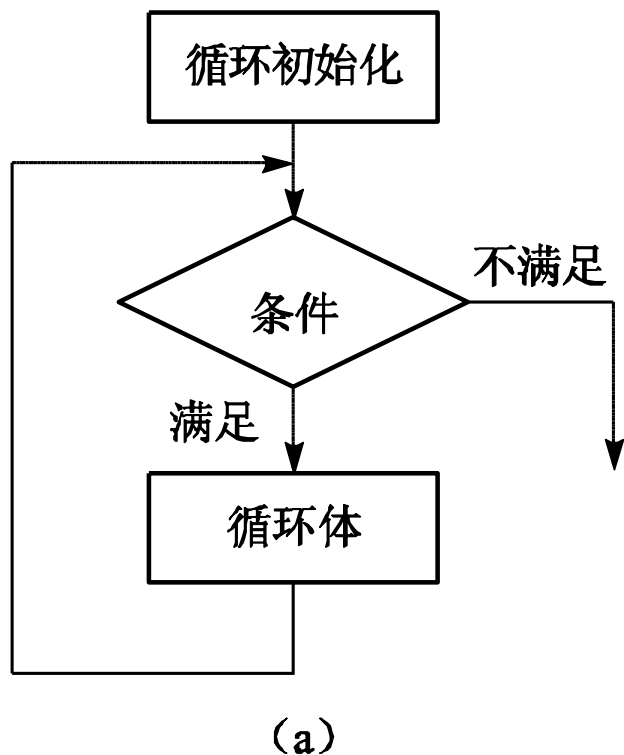
• invoke printf,offset Msg3 ;提示输入错误
• jmp Rdkb
•
• CR:
• invoke printf,offset MsgC ;显示 Create a File
• jmp start ;回到主菜单， 继续运行
•
• UP:
• invoke printf,offset MsgU ;显示 Update a File
• jmp start
•
• DE:
• invoke printf,offset MsgD ; 显示 Delete a File
• jmp start
•
• PR:
• invoke printf,offset MsgP ; 显示 Print a File
• jmp start
•
• QU:
• invoke printf,offset MsgQ ; 显示 Quit
• ret ;返回系统
• end start

```

- 采用跳转表的好处是：可以减少为决定程序分支而进行比较的次数；程序不必将输入的数字和1 ~ 5进行比较，直接根据数字计算出要跳转的地址。

# 循环程序设计

- 循环有两种基本结构，DO\_WHILE和DO\_UNTIL结构
- 在初始循环次数可能为0的情况下则**必须**使用DO\_WHILE结构



- (1) 循环初始化。它包括设置循环次数的初始值、地址指针的初始设置等。
- (2) 循环体。这是循环工作的主体，包括要重复执行的操作，以及循环的修改部分。修改部分包括地址指针的修改、循环控制条件的修改等。
- (3) 循环控制部分。它是控制循环的关键，判断循环条件满足与否。例如判断循环次数是否为0等。

计算 $1+2+3+\cdots+100$ ， 用一个循环来实现

- sum            dword        0
- mov           ecx, 100
- mov           eax, 1
- d10:
- add           sum, eax
- inc    eax
- loop          d10

# 计算n的阶乘

- .386
- .model flat,stdcall
- option casemap:none
- includelib msvcrt.lib
- printf PROTO C :dword, :vararg
- .data
- Fact dword ?
- N equ 6
- szFmt byte 'factorial(%d)=%d', 0ah, 0 ;输出结果格式字符串
- .code
- start:
- mov ecx, N ;循环初值
- mov eax, 1 ;Fact初值
- e10:
- imul eax, ecx ;Fact=Fact\*ECX
- loop e10 ;循环N次
- mov Fact, eax ;保存结果
- invoke printf, offset szFmt, N, Fact ;打印结果
- ret
- end start



- 设数组X、Y中分别存有10个双字型数据。试实现以下计算并把结果存入Z单元。
- $Z_1 = X_1 + Y_1$     $Z_2 = X_2 + Y_2$     $Z_3 = X_3 - Y_3$
- $Z_4 = X_4 - Y_4$     $Z_5 = X_5 - Y_5$     $Z_6 = X_6 + Y_6$
- $Z_7 = X_7 - Y_7$     $Z_8 = X_8 - Y_8$     $Z_9 = X_9 + Y_9$
- $Z_{10} = X_{10} + Y_{10}$

# 分析

- 10组数进行运算，运算操作符不同，且无规律可循。若直接用前边介绍的循环程序难以实现。这里设想把加用某个值表示（设用0），减用另一个值表示（设用1），10个式子的操作用10位二进制数表示。对于本例，若按 $Z_{10}$ 、 $Z_9$ 、 $\dots$ 、 $Z_1$ 的计算顺序把它们的操作符自右至左排列起来，则操作符数值化后得到一串二进制位0011011100，把它放入一个32位的内存变量中，高22位无意义（此处用0填充），这种存储单元一般被叫做逻辑尺。计算时按照 $Z_1 \dots Z_{10}$ 顺序，先求 $Z_1$ 的值。每次把逻辑尺右移一位，对移出位进行判断，若该位为0则加，为1则减。

之一

- .386
- .model flat,stdcall
- option casemap:none
- includelib msvcrt.lib
- printf PROTO C :dword,:vararg
- .data
- x dword 1,2,3,4,5,6,7,8,9,10
- y dword 5,4,3,2,1,10,9,8,7,6
- Rule dword 0000000011011100B
- z dword 10 dup (?)
- szFmt byte 'Z[%d]=%d', 0ah, 0 ;输出结果格式字符串

## 之二

- .code
- start:
  - mov    ecx,10                   ;循环次数
  - mov    edx,Rule                ;逻辑尺
  - mov    ebx,0
- next:
  - mov    eax,x[ebx]               ;取X中的一个数
  - shr    edx,1                    ;逻辑尺右移一位
  - jc     subs                    ;分支判断并实现转移
  - add    eax,y[ebx]               ;两数加
  - jmp    short result
- subs:
  - sub    eax,y[ebx]               ;两数减

# 之三

- result:
- mov    z[ebx],eax       ;存结果
- add    ebx,4            ;修改地址指针
- loop   next
- xor     ebx, ebx        ;显示出各元素的值
- PrintNext:
- invoke  printf, offset szFmt, ebx, Z[ebx\*4]; 显示
- inc     ebx     ;EBX下标加1
- cmp     ebx,10         ;是否已全部显示完
- jb      PrintNext     ;继续显示
- ret
- end     start

# 将一个字符串中的大写字符转换为小写字符，字符串以0结尾

- 分析：大写字符的ASCII码值为41H ~ 5AH，小写字符的ASCII码值为61H ~ 7AH。对大写字符，将它加上20H，即可以转换为小写字符。当遇到字符0时，循环结束。

- szStr BYTE 'Hello World!', 0
- MOV ESI, OFFSET szStr
- g10:
  - MOV AL, [ESI]
  - CMP AL, 0
  - JZ g30
  - CMP AL, 'A'
  - JB g20
  - CMP AL, 'Z'
  - JA g20
  - ADD AL, 'a' - 'A'
  - MOV [ESI], AL
- g20:
  - INC ESI
  - JMP g10
- g30:

# 多重循环程序设计-汇编中各层循环要控制好不能交叉

- 把数组中的7个元素用冒泡法按从小到大的顺序排列
- 在设计冒泡排序的程序时，需要两层循环。外层循环的循环次数是 $n-1$ ，以第0次、第1次、……、第 $n-2$ 次循环表示。第 $i$ 次外循环中，内层循环对数组下标为0至 $n-i-1$ 的元素依次“比较、交换”。内层循环的循环次数是 $n-i-1$ 。



之一

- .386
- .model flat,stdcall
- option casemap:none
- includelib msvcrt.lib
- printf PROTO C :dword,:vararg
- .data
- dArray dword 20, 15, 70, 30, 32, 89, 12
- ITEMS equ (\$-dArray)/4 ; 数组中元素的个数
- szFmt byte 'dArray[%d]=%d', 0ah, 0 ; 输出结果格式字符串

# 之 二

- .code
- start:
- mov     ecx, items-1
- i10:
- xor     esi, esi
- i20:
- mov     eax, dArray[esi\*4]
- mov     ebx, dArray[esi\*4+4]
- cmp     eax, ebx
- jl     i30
- mov     dArray[esi\*4], ebx
- mov     dArray[esi\*4+4], eax

# 之三

- i30:
- inc       esi
- cmp       esi, ecx
- jb        i20
- loop      i10
- xor       edi, edi
- i40:
- invoke    printf, offset szFmt, edi, dArray[edi\*4]
- inc       edi
- cmp       edi, ITEMS
- jb        i40
- ret
- end            start

# 浮点运算

- 专用于数值计算的浮点运算指令，包括浮点数的传送、浮点算术运算、浮点比较与控制等。
- 浮点处理单元x87 FPU
- IEEE浮点数格式

| 格式   | 说明                                |
|------|-----------------------------------|
| 单精度  | 32位：1位符号位，8位阶码，23位为有效数字的小数部分。     |
| 双精度  | 64位：1位符号位，11位阶码，52位为有效数字的小数部分。    |
| 扩展精度 | 80位：1位符号位，15位阶码，1位为整数部分，63位为小数部分。 |

# 浮点数的规格化

- 规格化浮点数的尾数域**最左位**（最高有效位）总是1，故这一位经常不予存储，而认为**隐藏**在小数点的左边。否则以修改阶码同时左右移小数点位置的办法，使其变为规格化数的形式。
- 在浮点数格式中，扩展双精度类型没有隐含位，因此它的有效位数与尾数位数一致，而单精度类型和双精度类型均有一个隐含整数位，因此它的有效位数比位数**多一个**。

# 浮点数存储

- float Var1 = 119.054f;  
为单精度浮点型

//定义float型变量Var1, f强制

- double Var2 = 119.054;

//定义double型变量Var2

- int main()

- {

- Var1 = Var1;

- Var2 = Var2;

- return 0;

- }

# 对应的汇编码

- 6:           Var1 = Var1;
- 0040E6B8           mov        eax,[\_Var1 (00426608)]
- 0040E6BD           mov        [\_Var1 (00426608)],eax
- 7:           Var2 = Var2;
- 0040E6C2           mov        ecx,dword ptr [\_Var2 (00426610)]
- 0040E6C8           mov        dword ptr [\_Var2 (00426610)],ecx
- 0040E6CE           mov        edx,dword ptr [\_Var2+4 (00426614)]
- 0040E6D4           mov        dword ptr [\_Var2+4 (00426614)],edx
- 8:           return 0;
- 0040E6DA           xor        eax,eax

# 对单精度数Var1

- 从内存00426608处取出变量Var1保存的值为： A6 1B EE 42，转化为二进制（逆序存放）：
- 01000010 11101110 00011011 10100110
- 根据单精度的划分方式把32位划分成三部分：
- ①.符号位为0，为正数；
- ②.指数为 10000101（133），减去127得6（移码）；
- ③.尾数加上1后为1.11011100001101110100110，十进制表示为：1.86021876
- 尾数乘以2的6次方后可得结果为：119.05400（单精度7~8位有效数字）



# 对双精度数Var2

- 从内存00426610和00426614处取出变量Var1保存的值为： FA 7E 6A BC 74 C3 5D 40， 转化为二进制（逆序存放）：
- 01000000 01011101 11000011 01110100 10111100 01101010 01111110 11111010
- ①.符号位为0， 为正数；
- ②.指数为10000000101（1029）， 减去1023得6；
- ③.尾数加上1后为  
1.1101110000110111010010111100011010100111111011111010
- 转化为10进制后乘以2的6次方后可得结果为119.054000000000  
（双精度15～16位有效数字）

# 浮点寄存器

- FPU不使用通用寄存器（EAX、EBX等），在执行浮点运算时有自己的一组寄存器，称为寄存器栈（register stack）
- 寄存器栈中有 8 个独立寻址的80位寄存器，名称分别为FPR0, FPR1, ……，FPR7，他们以堆栈形式组织在一起
- 栈顶也写做st(0)，最后一个寄存器写做st(7)
- FPU另有3个16位的寄存器，分别为控制寄存器、状态寄存器、标志寄存器



- (1) 当程序需要向寄存器栈中装入数据的时候，栈顶指针的值减1，然后将数据压入栈顶指针指向的浮点寄存器中。
- (2) 浮点寄存器栈是一个循环栈，当栈顶指针指向的地址值为0的时候，下一次入栈操作（FLD指令）则将数据压入地址值为7的浮点寄存器中，栈顶指针地址值为7。

- (3) 当需要将堆栈中的数据保存到内存中的时候 (FST指令), 则进行出栈操作。出栈操作与入栈顺序相反, 栈顶指针加1, 若出栈前栈顶地址值为7, 则出栈后栈顶值为0。
- (4) 在用户通过指令对浮点数据操作的时候, 这些浮点寄存器所使用的名称分别为ST(0)、ST(1)、ST(2)、ST(3)、ST(4)、ST(5)、ST(6)、ST(7), 这里的ST(i)中的i不是寄存器的地址, 而是距离栈顶的长度。也就是说, ST(i)表示距离栈顶的第i个单元的寄存器

# 计算表达式 $f = a + b * m$ 的值

- .586
- .model flat, stdcall
- option casemap:none
- includelib msvcrt.lib
- printf PROTO C :ptr sbyte, :VARARG
- .data
- szMsg byte   "%f", 0ah, 0
- a            real8         3.2
- b            real8         2.6
- m            real8         7.1
- f            real8         ?

- .code
- start:
  - finit ;finit为FPU栈寄存器的初始化
  - fld m ;fld为浮点值入栈
  - fld b
  - fmul st(0),st(1) ;fmul为浮点数相乘，结果保存在目标操作数中
  - fld a
  - fadd st(0),st(1) ;fmul为浮点数相加，结果保存在目标操作数中
  - fst f ;fst将栈顶数据保存到内存单元
  - invoke printf, offset szMsg, f
  - ret
- end start

# 寄存器栈变化

|       |       |       |       |                  |       |       |       |                  |       |                               |
|-------|-------|-------|-------|------------------|-------|-------|-------|------------------|-------|-------------------------------|
| 7.1   | ST(0) | 7.1   | ST(1) | 7.1              | ST(1) | 7.1   | ST(2) | 7.1              | ST(2) | <div>21.66</div> <div>f</div> |
|       | ST(7) | 2.6   | ST(0) | 18.46            | ST(0) | 18.46 | ST(1) | 18.46            | ST(1) |                               |
|       | ST(6) |       | ST(7) |                  | ST(7) | 3.2   | ST(0) | 21.66            | ST(0) |                               |
|       | ST(5) |       | ST(6) |                  | ST(6) |       | ST(7) |                  | ST(7) |                               |
|       | ST(4) |       | ST(5) |                  | ST(5) |       | ST(6) |                  | ST(6) |                               |
|       | ST(3) |       | ST(4) |                  | ST(4) |       | ST(5) |                  | ST(5) |                               |
|       | ST(2) |       | ST(3) |                  | ST(3) |       | ST(4) |                  | ST(4) |                               |
|       | ST(1) |       | ST(2) |                  | ST(2) |       | ST(3) |                  | ST(3) |                               |
| fld m |       | fld b |       | fmul st(0),st(1) |       | fld a |       | fadd st(0),st(1) |       | fst f                         |



# 标志寄存器

- 每个FPR寄存器，都有一个**两位**的标志（Tag）位，这8个标志tag0 ~ tag7组成一个16位的标志寄存器
- 标志寄存器记录了每个浮点寄存器的状态，当**装入**数据的时，硬件会将寄存器中相应的tag置为**有效**，反之置为**空**
- 当入栈时遇到已经标为有效的寄存器时，产生**上溢**；如果出栈时，遇到相应tag为空时，则产生**下溢**

FPU 寄存器栈

|      |
|------|
| FPR7 |
| FPR6 |
| FPR5 |
| FPR4 |
| FPR3 |
| FPR2 |
| FPR1 |
| FPR0 |

标志寄存器

|      |
|------|
| tag7 |
| tag6 |
| tag5 |
| tag4 |
| tag3 |
| tag2 |
| tag1 |
| tag0 |

标志 tag 值含义

- 00: 对应数据寄存器存有有效数据
- 01: 对应数据寄存器的数据为 0
- 10: 对应数据寄存器的数据为特殊数据：  
非数 NaN、无限大或非规格化数据
- 11: 对应数据寄存器内没有数据，为空状态

# 状态寄存器

- 状态寄存器16位，表明浮点处理单元当前的各种操作状态，每条浮点指令运算后都会更新状态寄存器以反映执行结果情况，与整数处理单元的EFLAGS作用功能类似

| 15 | 14 | 13  | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| B  | C3 | TOP |    |    | C2 | C1 | C0 | ES | SF | PE | UE | OE | ZE | DE | IE |

# (1) 堆栈标志

- TOP ( $\text{bit}_{11} \sim \text{bit}_{13}$ ) : 表明浮点数据寄存器中的栈顶位置, 即ST(0)所在的FPR地址, 三位编码 (000 ~ 111) 指向8个数据寄存器栈中的某一个寄存器。
- SF ( $\text{bit}_6$ ) : 表明堆栈是否发生溢出, 为1时表示发生溢出错误。
- C1 ( $\text{bit}_9$ ) : 表明溢出情况, C1=1为上溢, C1=0为下溢。
- C3 ( $\text{bit}_{14}$ )、C2 ( $\text{bit}_{10}$ )、C0 ( $\text{bit}_8$ ) : 保存浮点比较指令的比较结果。

# 异常标志

- 状态寄存器的低6位反映了浮点运算可能出现的6种异常。
- PE (bit<sub>5</sub>, Precision Exception, 精度异常) : 为1表示结果或者操作数超出指定的精度范围, 出现不准确的结果。
- UE (bit<sub>4</sub>, Underflow Exception, 下溢异常) : 为1表示非0的结果太小, 出现下溢。
- OE (bit<sub>3</sub>, Overflow Exception, 上溢异常) : 为1表示结果过大, 出现上溢。
- ZE (bit<sub>2</sub>, Zero divide Exception, 除数为0异常) : 为1表示除数为0错误。

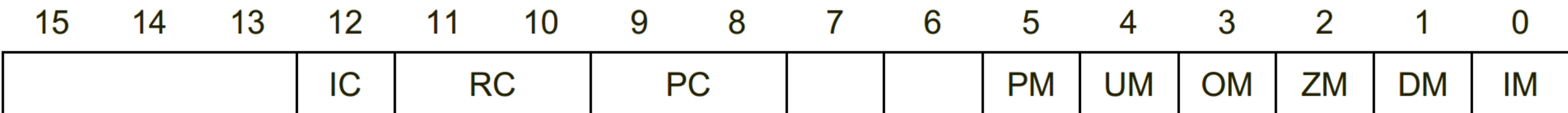
- DE ( $\text{bit}_1$ , Denormalized operand Exception, 非规格化操作数异常) : 为1表示至少有一个非规格化操作数。
- IE ( $\text{bit}_0$ , Invalid operation Exception, 非法操作异常) : 为1表示操作非法。

# 其他标志

- ES ( $\text{bit}_7$ , Error Summary, 错误总结)：当系统中任何一个未被屏蔽的异常发生时，该位置1，表明系统中是否出现异常。
- B ( $\text{bit}_{15}$ , FPU Busy, 浮点处理单元忙)：表示浮点处理单元的工作状态，为0表示空闲，为1表示正在执行浮点指令。

# 控制寄存器

- 16位浮点控制寄存器用于控制浮点处理单元的异常屏蔽、精度及舍入操作





# 异常屏蔽控制

- PM ( $\text{bit}_5$ , Precision Mask, 精度异常屏蔽)：为1表示屏蔽精度异常。
- UM ( $\text{bit}_4$ , Underflow Mask, 下溢异常屏蔽)：为1表示屏蔽下溢异常。
- OM ( $\text{bit}_3$ , Overflow Mask, 上溢异常屏蔽)：为1表示屏蔽上溢异常。
- ZM ( $\text{bit}_2$ , Zero divide Mask, 除数为0异常屏蔽)：为1表示屏蔽除数为0异常。
- DM ( $\text{bit}_1$ , Denormalized operand Mask, 非规格化操作数异常屏蔽)：为1表示屏蔽非规格化操作数异常。
- IM ( $\text{bit}_0$ , Invalid operation Mask, 非法操作异常屏蔽)：为1表示屏蔽非法操作异常。

# 精度控制

- PC ( $\text{bit}_8 \sim \text{bit}_9$ , Precision Control, 精度控制)：用于控制浮点计算结果的精度。PC=00时，32位单精度浮点数；PC=01时，保留；PC=10时，64位双精度浮点数；PC=11时，80位扩展精度浮点数。

# 舍入控制

- RC ( $\text{bit}_{10} \sim \text{bit}_{11}$ , Rounding Control, 舍入控制) 浮点处理单元无法产生要求的精确值时, 需要进行舍入操作, 以使得结果接近于精确值。

| RC | 舍入类型    | 舍入方式                        |
|----|---------|-----------------------------|
| 00 | 就近或偶数舍入 | 舍入结果接近准确值, 类似四舍五入一样接近则取偶数结果 |
| 01 | 向下舍入    | 正数截尾; 负数多余位不全为 0, 则最低位进 1   |
| 10 | 向上舍入    | 负数截尾; 正数多余位不全为 0, 则最低位进 1   |
| 11 | 向 0 舍入  | 正负数均截尾                      |

# 无限大控制

- IC ( $\text{bit}_{12}$ , Infinity Control, 无限控制) 用于兼容其他协处理器。

# 浮点指令及其编程

- 浮点处理单元具有自己的指令系统，指令助记符均已F开头
- (1) 浮点数据传送指令：完成内存与栈顶st(0)、数据寄存器st(i)与栈顶之间的浮点格式数据的**传送**。
- (2) 常数加载指令：实现将特定常数加载到**堆栈**。
- (3) 浮点算术运算指令：算术运算指令实现浮点数的加、减、乘、除运算，它们支持的**寻址**方式相同。这组指令还包括有关算术运算的指令，例如求绝对值、取整等。

- (4) 浮点超越函数指令：超越函数指令实现三角函数、指数和对数运算的操作。
- (5) 浮点比较指令：比较栈顶数据与指定的源操作数，比较结果通过浮点状态寄存器反映。
- (6) FPU控制指令：用于控制和检测浮点处理单元FPU的状态及操作方式。

# 数据定义

- 数据定义伪指令dd(dword) / dq(qword) / dt(tbyte)依次定义32/64/80位数据，它们可以用于定义单精度、双精度和扩展精度浮点数
- MASM 6.11建议采用REAL4、REAL8、REAL10定义单、双、扩展精度浮点数，但不能出现纯整数形式（整数后面补小数点即可）
- 实常数可以用E/e表示10的幂。

- a    real8    3.2            ;定义64位浮点数变量a, 初始化为3.2
- b    real10   100.25e9 ;定义80位浮点数变量b, 初始化为100.25e9
- c    qword    3.            ;定义64位浮点数变量c, 初始化为3.0
- d    qword    3            ;定义64位整型变量d, 初始化为3



# 寻址方式

- (1) 寄存器寻址：操作数保存在指定的数据寄存器栈中，用ST(i)表示。
- 指令：fadd st(0), st(1)
- 将寄存器栈中的ST(0)和ST(1)相加，结果存储在ST(0)中。
- (2) 存储器寻址：操作数在内存中，内存中的数据可以采用与数据有关的存储器寻址方式访问。
- 指令：fld m
- 将在内存定义的变量m加载到浮点寄存器栈中，m保存在内存中，以直接寻址方式访问。

# 浮点指令

- (1) 数据传送指令

| 指令        | 说明                                                          |
|-----------|-------------------------------------------------------------|
| FLD src   | 将源操作数 src (mem32/mem64/mem80/ST(i))加载到寄存器栈 ST(0)            |
| FST dest  | 将寄存器栈 ST(0)保存到目标操作数 dest(mem32/mem64/mem80/ST(i)), ST(0)不出栈 |
| FSTP dest | 将寄存器栈 ST(0)保存到目标操作数 dest(mem32/mem64/mem80/ST(i)), ST(0)出栈  |

## (2) 常数加载指令

| 指令     | 说明                              |
|--------|---------------------------------|
| FLD1   | 将常数 1.0 加载到寄存器栈 ST(0)           |
| FLDZ   | 将常数 0.0 加载到寄存器栈 ST(0)           |
| FLDPI  | 将常数 $\pi$ 加载到寄存器栈 ST(0)         |
| FLDL2T | 将常数 $\log_2 10$ 加载到寄存器栈 ST(0)   |
| FLDL2E | 将常数 $\log_2 e$ 加载到寄存器栈 ST(0)    |
| FLDLG2 | 将常数 $\log_{10} 2$ 加载到寄存器栈 ST(0) |
| FLDLN2 | 将常数 $\log_e 2$ 加载到寄存器栈 ST(0)    |

### (3) 算术运算指令-之一

| 指令   | 格式                | 说明                                                |
|------|-------------------|---------------------------------------------------|
| FADD | FADD              | ST(0)加 ST(1)，结果暂存在 ST(1)中，ST(0)出栈，新的 ST(0)保存运算结果  |
|      | FADD src          | 将 src 与 ST(0)相加，结果保存在 ST(0)中                      |
|      | FADD st(i),st(0)  | ST(0)加 ST(i)，结果保存在 ST(i)中                         |
|      | FADD st(0),st(i)  | ST(0)加 ST(i)，结果保存在 ST(0)中                         |
|      | FADDP st(i),st(0) | ST(0)加 ST(i)，结果保存在 ST(i)中，ST(0)出栈                 |
| FSUB | FSUB              | ST(1)减去 ST(0)，结果暂存在 ST(1)中，ST(0)出栈，新的 ST(0)保存运算结果 |
|      | FSUB src          | ST(0)减去 src，结果保存在 ST(0)中                          |
|      | FSUB st(i),st(0)  | ST(i)减去 ST(0)，结果保存在 ST(i)中                        |
|      | FSUB st(0),st(i)  | ST(0)减去 ST(i)，结果保存在 ST(0)中                        |
|      | FSUBP st(i),st(0) | ST(i)减去 ST(0)，结果保存在 ST(i)中，ST(0)出栈                |
| FMUL | FMUL              | ST(0)乘 ST(1)，结果暂存在 ST(1)中，ST(0)出栈，新的 ST(0)保存运算结果  |
|      | FMUL src          | ST(0)乘以 src，结果保存在 ST(0)中                          |

| 指令         | 格式                | 说明                                              |
|------------|-------------------|-------------------------------------------------|
|            | FMUL st(i),st(0)  | ST(i)乘以 ST(0)，结果保存在 ST(i)中                      |
|            | FMUL st(0),st(i)  | ST(0)乘以 ST(i)，结果保存在 ST(0)中                      |
|            | FMULP st(i),st(0) | ST(i)乘以 ST(0)，结果保存在 ST(i)中，ST(0)出栈              |
| FDIV       | FDIV              | ST(1)/ST(0)，结果暂存在 ST(1)中，ST(0)出栈，新的 ST(0)保存运算结果 |
|            | FDIV src          | ST(0)/src，结果保存在 ST(0)中                          |
|            | FDIV st(i),st(0)  | ST(i)/ST(0)，结果保存在 ST(i)中                        |
|            | FDIV st(0),st(i)  | ST(0)/ST(i)，结果保存在 ST(0)中                        |
|            | FDIVP st(i),st(0) | ST(i)/ST(0)，结果保存在 ST(i)中，ST(0)出栈                |
| 其他<br>算术运算 | FSQRT             | 计算 ST(0)的平方根，结果保存在 ST(0)中                       |
|            | FSCALE            | 计算 2 的 ST(0)次方，结果保存在 ST(0)中                     |
|            | FPREM             | 计算 ST(0)% ST(1)，结果保存在 ST(0)中                    |
|            | FABS              | 计算 ST(0)的绝对值，结果保存在 ST(0)中                       |
|            | FCHS              | 计算 ST(0)的相反数，结果保存在 ST(0)中                       |

## (4) 浮点比较指令

- CMP是对整数进行比较。对于浮点数，FPU提供了独立的比较机制和指令，采用浮点数独有的比较指令，比较结果通过状态寄存器中的C0、C2和C3标志位给出。
- 由于所有的浮点数都是带符号数，因此浮点比较指令是执行的带符号数比较。

| 指令          | 说明                                 |
|-------------|------------------------------------|
| FCOM        | 比较 ST(0)和 ST(1)的大小关系               |
| FCOM src    | 比较 ST(0)和 src 的大小关系                |
| FCOM st(i)  | 比较 ST(0)和 ST(i)的大小关系               |
| FCOMP       | 比较 ST(0)和 ST(1)的大小关系，完成比较后 ST(0)出栈 |
| FCOMP src   | 比较 ST(0)和 src 的大小关系，完成比较后 ST(0)出栈  |
| FCOMP st(i) | 比较 ST(0)和 ST(i)的大小关系，完成比较后 ST(0)出栈 |

| 条件        | C3 | C2 | C0 | 转移指令    |
|-----------|----|----|----|---------|
| ST(0)>操作数 | 0  | 0  | 0  | JA/JNBE |
| ST(0)<操作数 | 0  | 0  | 1  | JB/JNAE |
| ST(0)=操作数 | 1  | 0  | 0  | JE/JZ   |
| 无序        | 1  | 1  | 1  | (无)     |



# 超越函数指令

- 对实数求三角函数、指数和对数等运算

| 指令     | 说明                               |
|--------|----------------------------------|
| FSIN   | 计算 ST(0)的 sin 值，结果保存在 ST(0)中     |
| FCOS   | 计算 ST(0)的 cos 值，结果保存在 ST(0)中     |
| FPTAN  | 计算 ST(0)的 tan 值，结果保存在 ST(0)中     |
| FPATAN | 计算 ST(0)的 arctan 值，结果保存在 ST(0)中  |
| F2XM1  | 计算 2 的 ST(0)次方，减去 1，结果保存在 ST(0)中 |

# FPU控制指令

| 指令         | 说明                |
|------------|-------------------|
| FINIT      | 初始化 FPU           |
| FLDCW src  | 从 src 装入 FPU 的控制字 |
| FSTCW dest | 保存状态字的值到 dest     |
| FCLEX      | 清除异常              |
| FNOP       | 空操作               |

# 输入圆的半径， 计算圆面积

- .586
- .model flat, stdcall
- option casemap:none
- includelib msvcrt.lib
- scanf PROTO C :ptr sbyte, :VARARG
- printf PROTO C :ptr sbyte, :VARARG
- .data
- szMsg1 byte "%lf", 0
- szMsg2 byte "%lf", 0ah, 0
- r real8 ? ;圆半径
- S real8 ? ;圆面积



# 输入圆的半径，计算圆面积

- 程序整体采用C语言编写，计算圆面积采用汇编语言嵌入的方式实现

- #include "stdio.h"
- int main()
- {
- float r, S;
- printf("请输入圆半径: ");
- scanf("%f",&r);
- \_\_asm{
- fld     r
- fld     r
- fmulp st(1),st(0)
- fldpi
- fmulp st(1),st(0)
- fst S    }
- printf("\n圆面积为: %f",S);
- return 0;
- }

//此处为两个下划线