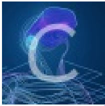
【翻译】intel指令格式与长度反汇编引擎ADE32分析——来自看雪软件安全网站

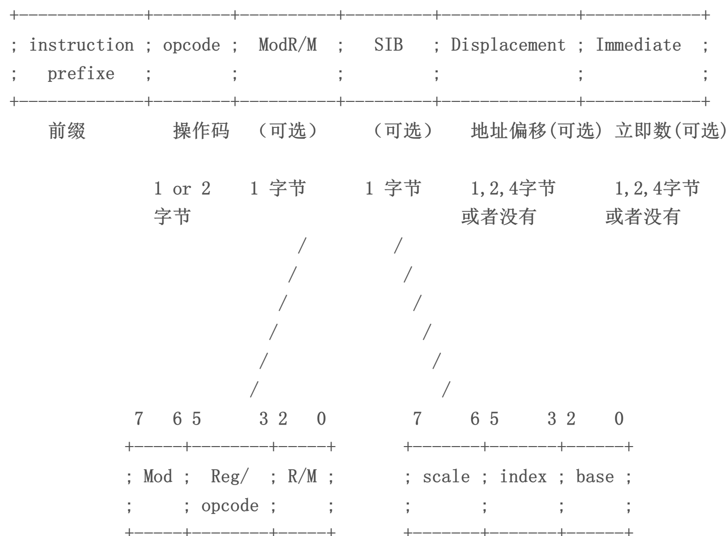
转载

stgsd 于 2008-12-04 13:14:00 发布 2344 收藏
分类专栏： 网摘 文章标签： 汇编 引擎 c struct table 扩展

 网摘 专栏收录该内容

1 篇文章 0 订阅
订阅专栏

```
标 题: 【翻译】intel指令格式与长度反汇编引擎ADE32分析
作 者: 火影
时 间: 2007-10-31,20:31
链 接: http://bbs.pediy.com/showthread.php?t=54180

我翻译的29A#7的一篇文章:
********************************
   ***                          ***
   **  INTEL INSTRUCTION FORMAT  **
   ***                          ***
   ********************************


    intel 指令具有特有的格式:


                        intel 指令格式
```



```
+-------------+--------+----------+---------+--------------+------------+
; instruction ; opcode ;  ModR/M  ;   SIB   ; Displacement ; Immediate  ;
;   prefixe   ;        ;          ;         ;              ;            ;
+-------------+--------+----------+---------+--------------+------------+
    前缀          操作码   （可选）     （可选）    地址偏移(可选) 立即数(可选)

               1 or 2    1 字节      1 字节     1,2,4字节      1,2,4字节
               字节                            或者没有        或者没有
                              /         /
                             /           /
                            /             /
                           /               /
                          /                 /
                         /                   /
            7   6 5      3 2   0          7     6 5     3 2     0
            +-----+--------+-----+        +-------+-------+------+
            ; Mod ;  Reg/  ; R/M ;        ; scale ; index ; base ;
            ;     ; opcode ;     ;        ;       ;       ;      ;
            +-----+--------+-----+        +-------+-------+------+
```

```
   Intel 指令在长度上可以不同, 但是它们都同样具有以上的6组
   ***************************
   ***                     ***
   **       指令前缀          **
   ***                     ***
   ***************************


    指令可以有（或者没有）0,1,2,3 or 4 个前缀，每个前缀占用一个字节
```

包含以下五种类型:

- 段寄存器(segment)前缀:指定如下段　2E 36 3E 26 64 65

- 操作数长度(Operand-Size)前缀 : 可以改变操作数长度 66

- 地址长度(Address-Size)前缀 : 可以改变地址长度 67

- 重复(REP/REPNE)前缀　　　: F3 F2

- 总线加锁(LOCK)前缀　　: 控制处理器总线　F0

```
+----------------+
段寄存器前缀
+----------------+
```

段寄存器前缀改变 **指令**的默认段，默认段为DS：

```
2EH : CS  segment override prefix.
36H : SS  segment override prefix.
3EH : DS  segment override prefix.
26H : ES  segment override prefix.
64H : FS  segment override prefix.
65H : GS  segment override prefix.


expl:     8B00    MOV EAX,DWORD PTR DS:[EAX]   (none prefix)
       2E:8B00    MOV EAX,DWORD PTR CS:[EAX]
          8B00    MOV EAX,DWORD PTR DS:[EAX]   (none prefix)
       36:8B00    MOV EAX,DWORD PTR SS:[EAX]

+---------------------+
操作数长度前缀
+---------------------+
```

该前缀允许一个程序在16 /32位 操作数长度之间转换(默认为32位）
如果指定前缀66H，则转换为16位

```
Quick example(in a win32 environement):

   89 C0    MOV EAX,EAX    (none prefix)
66 89 C0    MOV AX, AX     (prefix=66h)

+---------------------+
; 地址长度前缀 ;
+---------------------+
```

同上，WIN32编程，该前缀对我们没有用处

```
expl:     8B00   MOV EAX,DWORD PTR DS:[EAX]     32bits 寻址模式
       67:8B00   MOV EAX,DWORD PTR DS:[BX+SI]   16bits 寻址模式

+--------------------+
; 重复(REP/REPNE)前缀;
+--------------------+

      AD          LODS DWORD PTR DS:[ESI]
   F3 AD    REP   LODS DWORD PTR DS:[ESI]
   F2 AD    REPNE LODS DWORD PTR DS:[ESI]
3E F2 AD    REPNE LODS DWORD PTR DS:[ESI]

+-----------------+
; Bus LOCK Prefix ;
+-----------------+
```

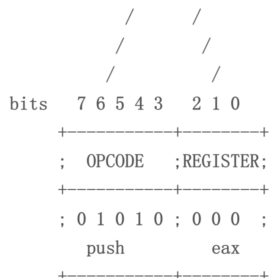
No use for us...Execpt if you want to know all the mystery of **intel** instruction
format in order to disasm the host, find a good place for the virus inside
the disassembled host code and reassembly all the infected file...

```
***************************
**                       **
**  操作码(OP CODE)        **
**                       **
***************************
```

```
+------------------+
; 1字节操作码 ;
+------------------+
```

在 **intel**文档你能找到这样的描述

PUSH REG **指令**:



```
PUSH REG <------> 01010reg    (where 'reg' are 3 bits)
PUSH EAX  --->  50h  --->  01010000   ---> 01010 000
                                              /      /
                                             /        /
                                            /          /
                                bits   7 6 5 4 3   2 1 0
                                     +-----------+--------+
                                     ;  OPCODE   ;REGISTER;
                                     +-----------+--------+
                                     ; 0 1 0 1 0 ; 0 0 0  ;
                                        push          eax
                                     +-----------+--------+

     Little Opcode Table:              Register Table:

                                     REG 8bit 16bit 32bit
     01000reg : INC  REG             000 : AL : AX : EAX
     01001reg : DEC  REG             001 : CL : CX : ECX
     01010reg : PUSH REG             010 : DL : DX : EDX
     01011reg : POP  REG             011 : BL : BX : EBX
     10010reg : XCHG EAX,REG         100 : AH : SP : ESP
     10111reg : MOV  REG,IMM32       101 : CH : BP : EBP
                                     110 : DH : SI : ESI
                                     111 : BH : DI : EDI
```

One another (fun) example:

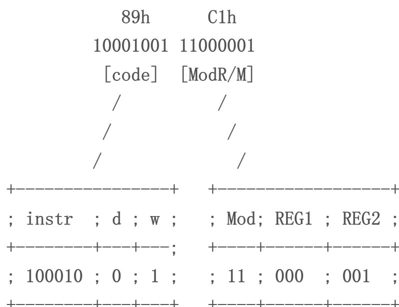
```
90h --->  10010 000 ---> XCHG EAX,EAX  (NOP)
```

只有 **指令** inc reg, dec reg, push reg, pop reg, xchg eax,reg, mov reg,imm32
可以这样编码!

其他方式编码:

```
89C1h ---> 1000100111000001b  ---> mov ecx,eax
```

仍旧为一字节操作码: C1h 为 [ModR/M] 字段!



```
            89h      C1h
         10001001 11000001
          [code]  [ModR/M]
           /          /
          /            /
         /              /
+----------------+   +------------------+
; instr  ; d ; w ;   ; Mod; REG1 ; REG2 ;
+--------+---+---;   +----+------+------+
; 100010 ; 0 ; 1 ;   ; 11 ; 000  ; 001  ;
+--------+---+---+   +----+------+------+
```

instr : 操作码 **指令**

(d)bit: 如果 (d)=0 顺序为 REG2-REG1
如果 (d)=1 顺序为 REG1-REG2

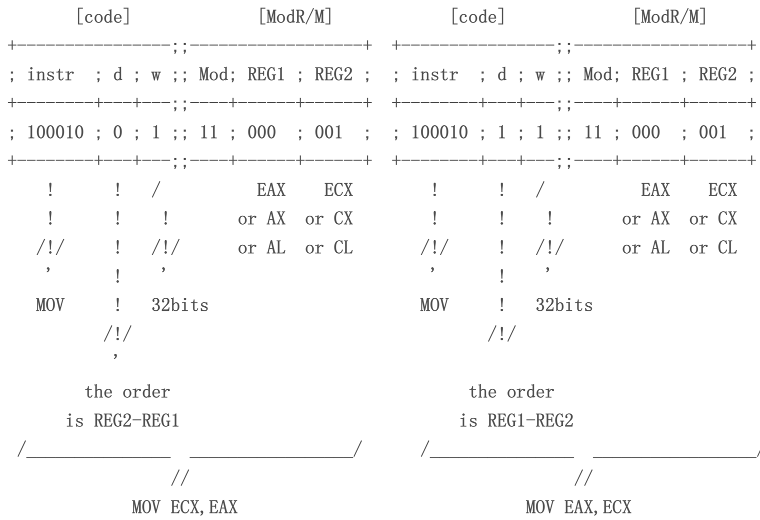
(w)bit: if (w)=0 we are in 8 bits mode in 32bits environement(Win32)
if (w)=1 we are in 32 bits mode in 32bits environement(Win32)

if (w)=0 we are in 8 bits mode in 16bits environement
if (w)=1 we are in 16 bits mode in 16bits environement

Mod : 以后再讲

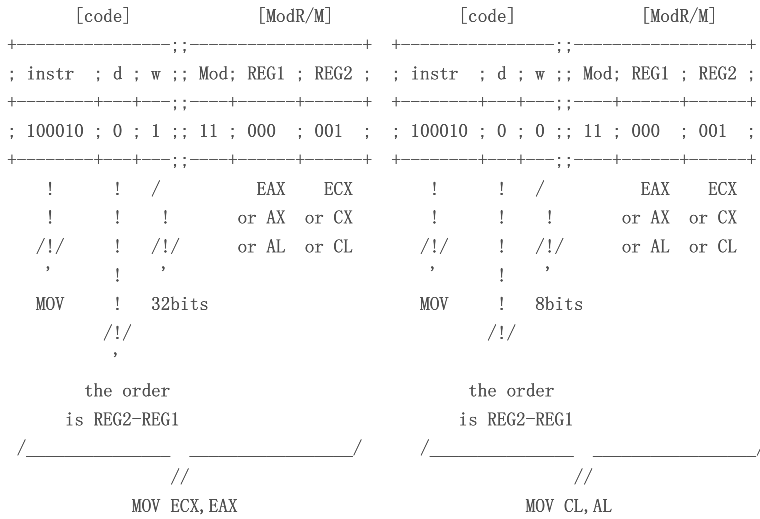
```
                         ############################
                         # example for the (d) bit: #
```

```
                    ############################
                      (in 32bits environement)
```



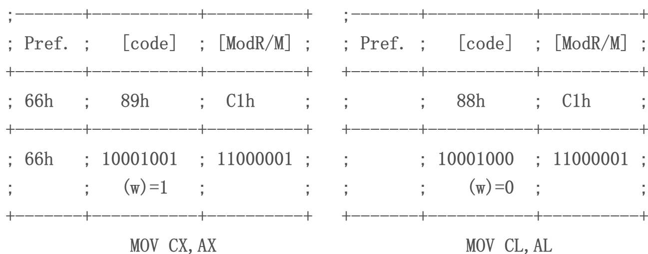
```
      [code]             [ModR/M]              [code]             [ModR/M]
+---------------;;-----------------+  +---------------;;-----------------+
; instr  ; d ; w ;; Mod; REG1 ; REG2 ;  ; instr  ; d ; w ;; Mod; REG1 ; REG2 ;
+--------+---+---;;----+------+------+  +--------+---+---;;----+------+------+
; 100010 ; 0 ; 1 ;; 11 ; 000  ; 001  ;  ; 100010 ; 1 ; 1 ;; 11 ; 000  ; 001  ;
+--------+---+---;;----+------+------+  +--------+---+---;;----+------+------+
    !      !   /          EAX    ECX        !      !   /          EAX    ECX
    !      !    !       or AX  or CX        !      !    !       or AX  or CX
   /!/     !   /!/      or AL  or CL       /!/     !   /!/      or AL  or CL
    '      !    '                           '      !    '
   MOV     !   32bits                      MOV     !   32bits
          /!/                                     /!/
           '

       the order                               the order
     is REG2-REG1                            is REG1-REG2
 /________________  __________________/    /________________  __________________/
                 //                                       //
            MOV ECX,EAX                              MOV EAX,ECX
```

```
                    ############################
                    # example for the (w) bit: #
                    ############################
```



```
      [code]             [ModR/M]              [code]             [ModR/M]
+---------------;;-----------------+  +---------------;;-----------------+
; instr  ; d ; w ;; Mod; REG1 ; REG2 ;  ; instr  ; d ; w ;; Mod; REG1 ; REG2 ;
+--------+---+---;;----+------+------+  +--------+---+---;;----+------+------+
; 100010 ; 0 ; 1 ;; 11 ; 000  ; 001  ;  ; 100010 ; 0 ; 0 ;; 11 ; 000  ; 001  ;
+--------+---+---;;----+------+------+  +--------+---+---;;----+------+------+
    !      !   /          EAX    ECX        !      !   /          EAX    ECX
    !      !    !       or AX  or CX        !      !    !       or AX  or CX
   /!/     !   /!/      or AL  or CL       /!/     !   /!/      or AL  or CL
    '      !    '                           '      !    '
   MOV     !   32bits                      MOV     !   8bits
          /!/                                     /!/
           '

       the order                               the order
     is REG2-REG1                            is REG2-REG1
 /________________  __________________/    /________________  __________________/
                 //                                       //
            MOV ECX,EAX                              MOV CL,AL
```

```
             examples with 66h prefix:
```

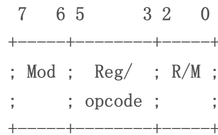


```
  ;-------+-----------+----------+    ;-------+-----------+----------+
  ; Pref. ;   [code]  ; [ModR/M] ;    ; Pref. ;   [code]  ; [ModR/M] ;
  +-------+-----------+----------+    +-------+-----------+----------+
  ; 66h   ;   89h     ;  C1h     ;    ;       ;   88h     ;  C1h     ;
  +-------+-----------+----------+    +-------+-----------+----------+
  ; 66h   ; 10001001  ; 11000001 ;    ;       ; 10001000  ; 11000001 ;
  ;       ;   (w)=1   ;          ;    ;       ;   (w)=0   ;          ;
  +-------+-----------+----------+    +-------+-----------+----------+
             MOV CX,AX                           MOV CL,AL
```

```
   Little Instruction Opcode table:

       BINARY       OPCODE
      -----------------------
      000010dw    OR  REG,REG
      001000dw    AND REG,REG
      001010dw    SUB REG,REG
      001100dw    XOR REG,REG
      001110dw    CMP REG,REG
      100000dw    ADD REG,REG
      100010dw    MOV REG,REG
   +--------------------+
```

```
; 双字节操作码 ;
+--------------------+
expl: 0f22 c0     mov cr0,eax
******************
**              **
**    ModR/M    **
**              **
******************
```

[ModR/M] 告诉处理器哪一个寄存器或者内存地址被使用
[ModRM] 8位被分为如下3组
groups (2 □3).



```
 7   6 5      3 2   0
+-----+--------+-----+
; Mod ;  Reg/  ; R/M ;
;     ; opcode ;     ;
+-----+--------+-----+
```

* [Mod]:

```
00  : 内存地址(memory address)                 expl: eax, [eax]
01  : memory address+1字节地址偏移量            expl: [eax+00]
10  : memory address+一个双字地址偏移量         expl: [eax+00000000]
11  : 两个操作数都为内存                        expl: eax,eax
```

* [Reg/opcode]:

这个字段被认为是代码扩展字段或者作为寄存器字段，处理器知道哪一个右侧编码相对这个字段

-如果为代码扩展字段(Code extension):

有的 **指令** 需要一个操作数，而有的需要两个操作数
ADD: instruction requires 2-Operands   (add eax, eax)
MUL: instruction requires only 1-Operand (mul eax)

如果我们使用单操作数
中间的3-Bits [Reg/opcode]field 就是代码扩展字段.

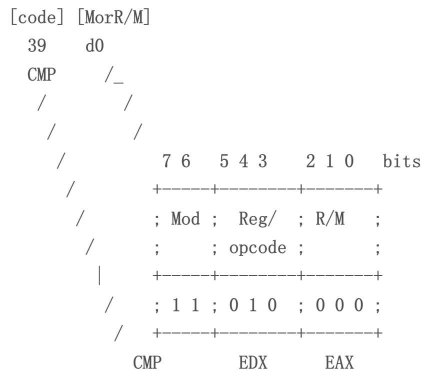
example:

```
F7D0 : 11110111 11 010 000   not eax
F7E0 : 11110111 11 100 000   mul eax
F7F0 : 11110111 11 110 000   div eax
```

they all have 0xF7 for the [code] byte, only [Reg/opcode] is different.
(000 is for the reg eax!)

-如果为寄存器字段:

example with the hex instruction:



```
[code] [MorR/M]
  39    d0
  CMP     /_
   /        /
    /        /
     /          7 6   5 4 3    2 1 0   bits
      /        +-----+--------+-------+
       /       ; Mod ;  Reg/  ; R/M   ;
        /      ;     ; opcode ;       ;
         |     +-----+--------+-------+
          /    ; 1 1 ; 0 1 0  ; 0 0 0 ;
           /   +-----+--------+-------+
             CMP        EDX       EAX
```

So 39d0h is CMP EAX,EDX (because of the (d) bits in [code] field)

* [R/M]

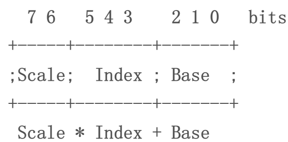
依赖于(Mode) Bits in the ModRM 字节:

```
- If [Mod]=00 and [R/M]=101 : 没有寄存器用于计算地址，取而代之为一个双字在[ModR/M]字段后面
                              expl: 3305 563412   xor eax,[12345678h]

- If [Mod]=00 and [R/M]=100 : [ModR/M]后面含有一个SIB字节

- If [Mod]=01 and [R/M]=100 : [ModR/M]后面含有一个SIB字节

- If [Mod]=10 and [R/M]=100 : [ModR/M]后面含有一个SIB字节
- If [Mod]=11               : exmpl: 89:C0 =  mov eax, eax
```

(P.S.: I've not verify this part, if [Mod]=xx then it means....)
(Let's make your own mind about that)

```
*****************
**             **
**     SIB     **
**             **
*****************
```

SIB 代表(Scale : Index :Base)全称.
一般格式为(Scale * Index + Base).



```
  7 6   5 4 3    2 1 0   bits
+-----+--------+-------+
;Scale;  Index ; Base  ;
+-----+--------+-------+
 Scale * Index + Base
```

* Scale : 被认为是乘数(of the index register)

```
00:xxx:xxx = 2^0 = 1  (*1)
01:xxx:xxx = 2^1 = 2  (*2)
10:xxx:xxx = 2^2 = 4  (*4)
11:xxx:xxx = 2^3 = 8  (*8)
```

* Index : 是一个寄存器(除了esp)
* Base : 基础寄存器 (Base Register) ?

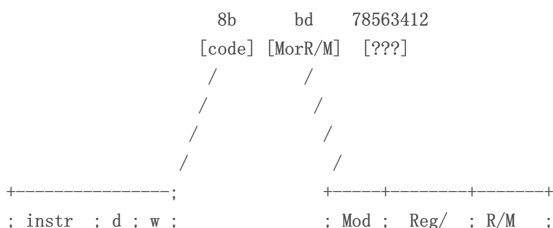
expl:

```
     [SIB]      |
----------------+----------------------------
00 : 000 : 001  |  mov reg, [1 * eax + ecx]
01 : 001 : 010  |  mov reg, [2 * ecx + edx]
01 : 110 : 111  |  mov reg, [2 * edi + esi]
10 : 010 : 011  |  mov reg, [4 * edx + ebx]
11 : 011 : 000  |  mov reg, [8 * eax + ebx]
```

- 如果索引寄存器为ESP，则index被忽略，在这个例子里scale也被忽略，只有base register被使用计算地址
- 如果我们需要如下编码(add reg, [esp]) 它可以使用SIB字节，但是它被编码成这样
  (add reg, [esp + DISPLACEMENT])

```
*****************
**             **
** DISPLACEMENT **
**             **
*****************
```

Just a quik example with the instruction 8b bd 78563412 :



```
                      8b      bd    78563412
                     [code] [MorR/M]  [???]
                      /         /
                     /           /
                    /             /
                   /               /
+----------------;                 +-----+--------+-------+
; instr  ; d ; w ;                 ; Mod ;  Reg/  ; R/M   ;
```

```
   +--------+---+---;                ;     ; opcode ;       ;
   ; 100010 ; 1 ; 1 ;                +-----+--------+-------+
   +--------+---+---;                ; 1 0 ; 1 1 1  ; 1 0 1 ;
   (d)=0 : the order is              +-----+--------+-------+
           REG1-REG2                         EDI      EBP
   (w)=1 : 32 bits mode                    or code
   instr : MOV r32,r/m32                  extension


[code]       = 8b        : opcode
                          ---> MOV r32,r/m32

[Mod]        = 10        : memory address with 1 dword displacement
                          ---> MOV REG,[REG+DWORD]

[Reg/opcode] = 111       : this instruction don't need an code extension
                           so it is a register
                          ---> MOV EDI,[EBP+DWORD]

[R/M]        = 101       : there is no [SIB] after the byte[ModR/M]

[78563412]   = 12345678h : it is the deplacement
                          ---> MOV EDI,[EBP+12345678h]


So the instruction 8b bd 78563412 is mov edi,[ebp+12345678h]

   *****************
   **             **
   **  IMMEDIATE  **
   **             **
   *****************

  expl:   05h          is the opcode for ADD EAX,imm32
          05h 00000010 is the instruction for add eax, 10000000



   ****************************
   **                        **
   **  ALGO OF DISASSEMBLY   **
   **                        **
   ****************************

       !
      /!/
 +-----'-------+
 ; Read 1 byte ; <---------
 +-------------+           /
       !                    /               +------------------+
      /!/                    --------<----- ; Convert to ASCII ;
 +-----'------------+                       +-------.----------+
 ; Check for prefix ;                              /!/
 +------------------+                               !
       !                                            !
      /!/                                   +------------------+
 ;-----'-----------------+                  ; [IMMEDIATE]      ;
 ; Decode [code] byte(s);                   +------------------+
 +-----------------------+                  ; [DISPLACEMENT]   ;
       !                                    +------------------+
      /!/                                   ; decode [SIB]     ;
 +-----'------------------+                 +------------------+
 ; JMP or CALL opcodes ? ; -------->------> ; decode [ModR/M] ;
 +------------------------+                 +------------------+
```

下面是我注释的ADE32的代码，很多语义表示不清楚，大家凑合看吧，偶水平不行啊

```
// ADE32 2.03c -- advanced 16/32-bit opcode assembler/disassembler engine

// this stuff is used to get instruction length and parse it into
// prefix, opcode, modregr/m, address, immediate, etc.
// 指令特征
#define C_ERROR   0xFFFFFFFF
#define C_ADDR1   0x00000001 //操作码中地址大小的位字段(字节)
#define C_ADDR2   0x00000002
```

```
#define C_ADDR4   0x00000004 //(双字)
#define C_LOCK    0x00000008 //加锁前缀
#define C_67      0x00000010 //地址大小修饰前缀(16/32位)
#define C_66      0x00000020 //操作数大小修饰前缀(16/32位)
#define C_REP     0x00000040 //重复前缀
#define C_SEG     0x00000080 //段寄存器前缀
#define C_ANYPREFIX  (C_66+C_67+C_LOCK+C_REP+C_SEG)
#define C_DATA1   0x00000100 //操作码中数据大小的位字段
#define C_DATA2   0x00000200
#define C_DATA4   0x00000400
#define C_SIB     0x00000800 //SIB字节
#define C_ADDR67  0x00001000 //地址字节数为disasm_defaddr
#define C_DATA66  0x00002000 //数据字节数为disasm_defdata
#define C_MODRM   0x00004000 //MODRM字节
#define C_BAD     0x00008000
#define C_OPCODE2 0x00010000 //操作码第二个字节
#define C_REL     0x00020000 // 这是跳转 指令jxx或者 call
#define C_STOP    0x00040000 // 这是回跳 指令， ret或者 jmp
// 指令信息
struct disasm_struct
{
   BYTE  disasm_defaddr ;         // 00
   BYTE  disasm_defdata ;         // 01
   DWORD disasm_len ;             // 02 03 04 05
   DWORD disasm_flag ;            // 06 07 08 09
   DWORD disasm_addrsize ;        // 0A 0B 0C 0D
   DWORD disasm_datasize ;        // 0E 0F 10 11
   BYTE  disasm_rep ;             // 12
   BYTE  disasm_seg ;             // 13
   BYTE  disasm_opcode ;          // 14
   BYTE  disasm_opcode2 ;         // 15
   BYTE  disasm_modrm ;           // 16
   BYTE  disasm_sib ;             // 17
  //地址
  union
  {
   BYTE  disasm_addr_b[8] ;       // 18 19 1A 1B  1C 1D 1E 1F
   WORD  disasm_addr_w[4] ;
   DWORD disasm_addr_d[2] ;
  char  disasm_addr_c[8] ;
   short disasm_addr_s[4] ;
  long  disasm_addr_l[2] ;
  } ;
  //数据
  union
  {
   BYTE  disasm_data_b[8] ;       // 20 21 22 23  24 25 26 27
   WORD  disasm_data_w[4] ;
   DWORD disasm_data_d[2] ;
  char  disasm_data_c[8] ;
   short disasm_data_s[4] ;
  long  disasm_data_l[2] ;
  } ;
} ; // disasm_struct
//按操作码从0x00开始排列，每一行代表一个操作
//码对应的 指令特征的集合
DWORD ade32_table[512] = {
/* 00 */  C_MODRM,
/* 01 */  C_MODRM,
/* 02 */  C_MODRM,
/* 03 */  C_MODRM,
/* 04 */  C_DATA1,
/* 05 */  C_DATA66,
/* 06 */  C_BAD,
/* 07 */  C_BAD,
/* 08 */  C_MODRM,
/* 09 */  C_MODRM,
/* 0A */  C_MODRM,
/* 0B */  C_MODRM,
/* 0C */  C_DATA1,
/* 0D */  C_DATA66,
/* 0E */  C_BAD,
```

```
/* 0F */  C_OPCODE2,
/* 10 */  C_MODRM+C_BAD,
/* 11 */  C_MODRM,
/* 12 */  C_MODRM+C_BAD,
/* 13 */  C_MODRM,
/* 14 */  C_DATA1+C_BAD,
/* 15 */  C_DATA66+C_BAD,
/* 16 */  C_BAD,
/* 17 */  C_BAD,
/* 18 */  C_MODRM+C_BAD,
/* 19 */  C_MODRM,
/* 1A */  C_MODRM,
/* 1B */  C_MODRM,
/* 1C */  C_DATA1+C_BAD,
/* 1D */  C_DATA66+C_BAD,
/* 1E */  C_BAD,
/* 1F */  C_BAD,
/* 20 */  C_MODRM,
/* 21 */  C_MODRM,
/* 22 */  C_MODRM,
/* 23 */  C_MODRM,
/* 24 */  C_DATA1,
/* 25 */  C_DATA66,
/* 26 */  C_SEG+C_BAD,
/* 27 */  C_BAD,
/* 28 */  C_MODRM,
/* 29 */  C_MODRM,
/* 2A */  C_MODRM,
/* 2B */  C_MODRM,
/* 2C */  C_DATA1,
/* 2D */  C_DATA66,
/* 2E */  C_SEG+C_BAD,
/* 2F */  C_BAD,
/* 30 */  C_MODRM,
/* 31 */  C_MODRM,
/* 32 */  C_MODRM,
/* 33 */  C_MODRM,
/* 34 */  C_DATA1,
/* 35 */  C_DATA66,
/* 36 */  C_SEG+C_BAD,
/* 37 */  C_BAD,
/* 38 */  C_MODRM,
/* 39 */  C_MODRM,
/* 3A */  C_MODRM,
/* 3B */  C_MODRM,
/* 3C */  C_DATA1,
/* 3D */  C_DATA66,
/* 3E */  C_SEG+C_BAD,
/* 3F */  C_BAD,
/* 40 */  0,
/* 41 */  0,
/* 42 */  0,
/* 43 */  0,
/* 44 */  C_BAD,
/* 45 */  0,
/* 46 */  0,
/* 47 */  0,
/* 48 */  0,
/* 49 */  0,
/* 4A */  0,
/* 4B */  0,
/* 4C */  C_BAD,
/* 4D */  0,
/* 4E */  0,
/* 4F */  0,
/* 50 */  0,
/* 51 */  0,
/* 52 */  0,
/* 53 */  0,
/* 54 */  0,
/* 55 */  0,
/* 56 */  0,
/* 57 */  0,
```

```
/* 58 */  0,
/* 59 */  0,
/* 5A */  0,
/* 5B */  0,
/* 5C */  C_BAD,
/* 5D */  0,
/* 5E */  0,
/* 5F */  0,
/* 60 */  C_BAD,
/* 61 */  C_BAD,
/* 62 */  C_MODRM+C_BAD,
/* 63 */  C_MODRM+C_BAD,
/* 64 */  C_SEG,
/* 65 */  C_SEG+C_BAD,
/* 66 */  C_66,
/* 67 */  C_67,
/* 68 */  C_DATA66,
/* 69 */  C_MODRM+C_DATA66,
/* 6A */  C_DATA1,
/* 6B */  C_MODRM+C_DATA1,
/* 6C */  C_BAD,
/* 6D */  C_BAD,
/* 6E */  C_BAD,
/* 6F */  C_BAD,
/* 70 */  C_DATA1+C_REL+C_BAD,
/* 71 */  C_DATA1+C_REL+C_BAD,
/* 72 */  C_DATA1+C_REL,
/* 73 */  C_DATA1+C_REL,
/* 74 */  C_DATA1+C_REL,
/* 75 */  C_DATA1+C_REL,
/* 76 */  C_DATA1+C_REL,
/* 77 */  C_DATA1+C_REL,
/* 78 */  C_DATA1+C_REL,
/* 79 */  C_DATA1+C_REL,
/* 7A */  C_DATA1+C_REL+C_BAD,
/* 7B */  C_DATA1+C_REL+C_BAD,
/* 7C */  C_DATA1+C_REL,
/* 7D */  C_DATA1+C_REL,
/* 7E */  C_DATA1+C_REL,
/* 7F */  C_DATA1+C_REL,
/* 80 */  C_MODRM+C_DATA1,
/* 81 */  C_MODRM+C_DATA66,
/* 82 */  C_MODRM+C_DATA1+C_BAD,
/* 83 */  C_MODRM+C_DATA1,
/* 84 */  C_MODRM,
/* 85 */  C_MODRM,
/* 86 */  C_MODRM,
/* 87 */  C_MODRM,
/* 88 */  C_MODRM,
/* 89 */  C_MODRM,
/* 8A */  C_MODRM,
/* 8B */  C_MODRM,
/* 8C */  C_MODRM+C_BAD,
/* 8D */  C_MODRM,
/* 8E */  C_MODRM+C_BAD,
/* 8F */  C_MODRM,
/* 90 */  0,
/* 91 */  0,
/* 92 */  0,
/* 93 */  C_BAD,
/* 94 */  C_BAD,
/* 95 */  C_BAD,
/* 96 */  C_BAD,
/* 97 */  C_BAD,
/* 98 */  C_BAD,
/* 99 */  0,
/* 9A */  C_DATA66+C_DATA2+C_BAD,
/* 9B */  0,
/* 9C */  C_BAD,
/* 9D */  C_BAD,
/* 9E */  C_BAD,
/* 9F */  C_BAD,
```

```
/* A0 */  C_ADDR67,
/* A1 */  C_ADDR67,
/* A2 */  C_ADDR67,
/* A3 */  C_ADDR67,
/* A4 */  0,
/* A5 */  0,
/* A6 */  0,
/* A7 */  0,
/* A8 */  C_DATA1,
/* A9 */  C_DATA66,
/* AA */  0,
/* AB */  0,
/* AC */  0,
/* AD */  C_BAD,
/* AE */  0,
/* AF */  C_BAD,
/* B0 */  C_DATA1,
/* B1 */  C_DATA1,
/* B2 */  C_DATA1,
/* B3 */  C_DATA1,
/* B4 */  C_DATA1,
/* B5 */  C_DATA1,
/* B6 */  C_DATA1+C_BAD,
/* B7 */  C_DATA1+C_BAD,
/* B8 */  C_DATA66,
/* B9 */  C_DATA66,
/* BA */  C_DATA66,
/* BB */  C_DATA66,
/* BC */  C_DATA66+C_BAD,
/* BD */  C_DATA66,
/* BE */  C_DATA66,
/* BF */  C_DATA66,
/* C0 */  C_MODRM+C_DATA1,
/* C1 */  C_MODRM+C_DATA1,
/* C2 */  C_DATA2+C_STOP,
/* C3 */  C_STOP,
/* C4 */  C_MODRM+C_BAD,
/* C5 */  C_MODRM+C_BAD,
/* C6 */  C_MODRM+C_DATA1,
/* C7 */  C_MODRM+C_DATA66,
/* C8 */  C_DATA2+C_DATA1,
/* C9 */  0,
/* CA */  C_DATA2+C_STOP+C_BAD,
/* CB */  C_STOP+C_BAD,
/* CC */  C_BAD,
/* CD */  C_DATA1,
/* CE */  C_BAD,
/* CF */  C_STOP+C_BAD,
/* D0 */  C_MODRM,
/* D1 */  C_MODRM,
/* D2 */  C_MODRM,
/* D3 */  C_MODRM,
/* D4 */  C_DATA1+C_BAD,
/* D5 */  C_DATA1+C_BAD,
/* D6 */  C_BAD,
/* D7 */  C_BAD,
/* D8 */  C_MODRM,
/* D9 */  C_MODRM,
/* DA */  C_MODRM,
/*  DB */  C_MODRM,
/* DC */  C_MODRM,
/*  DD */  C_MODRM,
/* DE */  C_MODRM,
/* DF */  C_MODRM,
/* E0 */  C_DATA1+C_REL+C_BAD,
/* E1 */  C_DATA1+C_REL+C_BAD,
/* E2 */  C_DATA1+C_REL,
/* E3 */  C_DATA1+C_REL,
/* E4 */  C_DATA1+C_BAD,
/* E5 */  C_DATA1+C_BAD,
/* E6 */  C_DATA1+C_BAD,
/* E7 */  C_DATA1+C_BAD,
/* E8 */  C_DATA66+C_REL,
```

```
/* E9 */  C_DATA66+C_REL+C_STOP,
/* EA */  C_DATA66+C_DATA2+C_BAD,
/* EB */  C_DATA1+C_REL+C_STOP,
/* EC */  C_BAD,
/* ED */  C_BAD,
/* EE */  C_BAD,
/* EF */  C_BAD,
/* F0 */  C_LOCK+C_BAD,
/* F1 */  C_BAD,
/* F2 */  C_REP,
/* F3 */  C_REP,
/* F4 */  C_BAD,
/* F5 */  C_BAD,
/* F6 */  C_MODRM,
/* F7 */  C_MODRM,
/* F8 */  0,
/* F9 */  0,
/* FA */  C_BAD,
/* FB */  C_BAD,
/* FC */  0,
/* FD */  0,
/* FE */  C_MODRM,
/* FF */  C_MODRM,
/* 00 */  C_MODRM,
/* 01 */  C_MODRM,
/* 02 */  C_MODRM,
/* 03 */  C_MODRM,
/* 04 */  C_ERROR,
/* 05 */  C_ERROR,
/* 06 */  0,
/* 07 */  C_ERROR,
/* 08 */  0,
/* 09 */  0,
/* 0A */  0,
/* 0B */  0,
/* 0C */  C_ERROR,
/* 0D */  C_ERROR,
/* 0E */  C_ERROR,
/* 0F */  C_ERROR,
/* 10 */  C_ERROR,
/* 11 */  C_ERROR,
/* 12 */  C_ERROR,
/* 13 */  C_ERROR,
/* 14 */  C_ERROR,
/* 15 */  C_ERROR,
/* 16 */  C_ERROR,
/* 17 */  C_ERROR,
/* 18 */  C_ERROR,
/* 19 */  C_ERROR,
/* 1A */  C_ERROR,
/* 1B */  C_ERROR,
/* 1C */  C_ERROR,
/* 1D */  C_ERROR,
/* 1E */  C_ERROR,
/* 1F */  C_ERROR,
/* 20 */  C_ERROR,
/* 21 */  C_ERROR,
/* 22 */  C_ERROR,
/* 23 */  C_ERROR,
/* 24 */  C_ERROR,
/* 25 */  C_ERROR,
/* 26 */  C_ERROR,
/* 27 */  C_ERROR,
/* 28 */  C_ERROR,
/* 29 */  C_ERROR,
/* 2A */  C_ERROR,
/* 2B */  C_ERROR,
/* 2C */  C_ERROR,
/* 2D */  C_ERROR,
/* 2E */  C_ERROR,
/* 2F */  C_ERROR,
/* 30 */  C_ERROR,
```

```
/* 31 */  C_ERROR,
/* 32 */  C_ERROR,
/* 33 */  C_ERROR,
/* 34 */  C_ERROR,
/* 35 */  C_ERROR,
/* 36 */  C_ERROR,
/* 37 */  C_ERROR,
/* 38 */  C_ERROR,
/* 39 */  C_ERROR,
/* 3A */  C_ERROR,
/* 3B */  C_ERROR,
/* 3C */  C_ERROR,
/* 3D */  C_ERROR,
/* 3E */  C_ERROR,
/* 3F */  C_ERROR,
/* 40 */  C_MODRM,
/* 41 */  C_MODRM,
/* 42 */  C_MODRM,
/* 43 */  C_MODRM,
/* 44 */  C_MODRM,
/* 45 */  C_MODRM,
/* 46 */  C_MODRM,
/* 47 */  C_MODRM,
/* 48 */  C_MODRM,
/* 49 */  C_MODRM,
/* 4A */  C_MODRM,
/* 4B */  C_MODRM,
/* 4C */  C_MODRM,
/* 4D */  C_MODRM,
/* 4E */  C_MODRM,
/* 4F */  C_MODRM,
/* 50 */  C_ERROR,
/* 51 */  C_ERROR,
/* 52 */  C_ERROR,
/* 53 */  C_ERROR,
/* 54 */  C_ERROR,
/* 55 */  C_ERROR,
/* 56 */  C_ERROR,
/* 57 */  C_ERROR,
/* 58 */  C_ERROR,
/* 59 */  C_ERROR,
/* 5A */  C_ERROR,
/* 5B */  C_ERROR,
/* 5C */  C_ERROR,
/* 5D */  C_ERROR,
/* 5E */  C_ERROR,
/* 5F */  C_ERROR,
/* 60 */  C_ERROR,
/* 61 */  C_ERROR,
/* 62 */  C_ERROR,
/* 63 */  C_ERROR,
/* 64 */  C_ERROR,
/* 65 */  C_ERROR,
/* 66 */  C_ERROR,
/* 67 */  C_ERROR,
/* 68 */  C_ERROR,
/* 69 */  C_ERROR,
/* 6A */  C_ERROR,
/* 6B */  C_ERROR,
/* 6C */  C_ERROR,
/* 6D */  C_ERROR,
/* 6E */  C_ERROR,
/* 6F */  C_ERROR,
/* 70 */  C_ERROR,
/* 71 */  C_ERROR,
/* 72 */  C_ERROR,
/* 73 */  C_ERROR,
/* 74 */  C_ERROR,
/* 75 */  C_ERROR,
/* 76 */  C_ERROR,
/* 77 */  C_ERROR,
/* 78 */  C_ERROR,
/* 79 */  C_ERROR,
```

```
/* 7A */  C_ERROR,
/* 7B */  C_ERROR,
/* 7C */  C_ERROR,
/* 7D */  C_ERROR,
/* 7E */  C_ERROR,
/* 7F */  C_ERROR,
/* 80 */  C_DATA66+C_REL,
/* 81 */  C_DATA66+C_REL,
/* 82 */  C_DATA66+C_REL,
/* 83 */  C_DATA66+C_REL,
/* 84 */  C_DATA66+C_REL,
/* 85 */  C_DATA66+C_REL,
/* 86 */  C_DATA66+C_REL,
/* 87 */  C_DATA66+C_REL,
/* 88 */  C_DATA66+C_REL,
/* 89 */  C_DATA66+C_REL,
/* 8A */  C_DATA66+C_REL,
/* 8B */  C_DATA66+C_REL,
/* 8C */  C_DATA66+C_REL,
/* 8D */  C_DATA66+C_REL,
/* 8E */  C_DATA66+C_REL,
/* 8F */  C_DATA66+C_REL,
/* 90 */  C_MODRM,
/* 91 */  C_MODRM,
/* 92 */  C_MODRM,
/* 93 */  C_MODRM,
/* 94 */  C_MODRM,
/* 95 */  C_MODRM,
/* 96 */  C_MODRM,
/* 97 */  C_MODRM,
/* 98 */  C_MODRM,
/* 99 */  C_MODRM,
/* 9A */  C_MODRM,
/* 9B */  C_MODRM,
/* 9C */  C_MODRM,
/* 9D */  C_MODRM,
/* 9E */  C_MODRM,
/* 9F */  C_MODRM,
/* A0 */  0,
/* A1 */  0,
/* A2 */  0,
/* A3 */  C_MODRM,
/* A4 */  C_MODRM+C_DATA1,
/* A5 */  C_MODRM,
/* A6 */  C_ERROR,
/* A7 */  C_ERROR,
/* A8 */  0,
/* A9 */  0,
/* AA */  0,
/* AB */  C_MODRM,
/* AC */  C_MODRM+C_DATA1,
/* AD */  C_MODRM,
/* AE */  C_ERROR,
/* AF */  C_MODRM,
/* B0 */  C_MODRM,
/* B1 */  C_MODRM,
/* B2 */  C_MODRM,
/* B3 */  C_MODRM,
/* B4 */  C_MODRM,
/* B5 */  C_MODRM,
/* B6 */  C_MODRM,
/* B7 */  C_MODRM,
/* B8 */  C_ERROR,
/* B9 */  C_ERROR,
/* BA */  C_MODRM+C_DATA1,
/* BB */  C_MODRM,
/* BC */  C_MODRM,
/* BD */  C_MODRM,
/* BE */  C_MODRM,
/* BF */  C_MODRM,
/* C0 */  C_MODRM,
/* C1 */  C_MODRM,
```

```
/* C2 */  C_ERROR,
/* C3 */  C_ERROR,
/* C4 */  C_ERROR,
/* C5 */  C_ERROR,
/* C6 */  C_ERROR,
/* C7 */  C_ERROR,
/* C8 */  0,
/* C9 */  0,
/* CA */  0,
/* CB */  0,
/* CC */  0,
/* CD */  C_DATA1,
/* CE */  0,
/* CF */  0,
/* D0 */  C_ERROR,
/* D1 */  C_ERROR,
/* D2 */  C_ERROR,
/* D3 */  C_ERROR,
/* D4 */  C_ERROR,
/* D5 */  C_ERROR,
/* D6 */  C_ERROR,
/* D7 */  C_ERROR,
/* D8 */  C_ERROR,
/* D9 */  C_ERROR,
/* DA */  C_ERROR,
/*  DB */  C_ERROR,
/* DC */  C_ERROR,
/*  DD */  C_ERROR,
/* DE */  C_ERROR,
/* DF */  C_ERROR,
/* E0 */  C_ERROR,
/* E1 */  C_ERROR,
/* E2 */  C_ERROR,
/* E3 */  C_ERROR,
/* E4 */  C_ERROR,
/* E5 */  C_ERROR,
/* E6 */  C_ERROR,
/* E7 */  C_ERROR,
/* E8 */  C_ERROR,
/* E9 */  C_ERROR,
/* EA */  C_ERROR,
/* EB */  C_ERROR,
/* EC */  C_ERROR,
/* ED */  C_ERROR,
/* EE */  C_ERROR,
/* EF */  C_ERROR,
/* F0 */  C_ERROR,
/* F1 */  C_ERROR,
/* F2 */  C_ERROR,
/* F3 */  C_ERROR,
/* F4 */  C_ERROR,
/* F5 */  C_ERROR,
/* F6 */  C_ERROR,
/* F7 */  C_ERROR,
/* F8 */  C_ERROR,
/* F9 */  C_ERROR,
/* FA */  C_ERROR,
/* FB */  C_ERROR,
/* FC */  C_ERROR,
/* FD */  C_ERROR,
/* FE */  C_ERROR,
/* FF */  C_ERROR
} ; // ade32_table[]

// ade32_disasm() -- returns opcode length  or 0

int ade32_disasm( IN  BYTE* opcode0,   IN  OUT disasm_struct* diza)
{
   BYTE* opcode = opcode0 ;

  disasm_struct temp_diza ;                    // comment if NULL is never passed
  if (diza == NULL) diza = &temp_diza ;       //
```

```
  memset(diza, 0x00, sizeof(disasm_struct)) ; // comment these lines,
  diza->disasm_defdata = 4 ;                  // and fill structure before call
  diza->disasm_defaddr = 4 ;                  // -- to 允许 16/32-bit disasm

  if (*( WORD*)opcode == 0x0000) return 0 ;
  if (*( WORD*)opcode == 0xFFFF) return 0 ;

   DWORD flag = 0 ;

repeat_prefix:
  //按字节取出操作码
   BYTE c = *opcode++ ;
  //查表,取出特征码
   DWORD t = ade32_table[ c ] ;
  //是否是已知前缀
  if (t & C_ANYPREFIX)
  {
    //是否为两次相同前缀
    if (flag & t) return 0 ;    // twice LOCK,SEG,REP,66,67
    //标志
    flag |= t ;
    //如果含有C_67，需要16位地址
    if (t & C_67)
    {
      //???
      diza->disasm_defaddr ^= 2^4 ;
    }
    //如果含有C_66，使用16位操作数
    else
    if (t & C_66)
    {
      //???
      diza->disasm_defdata ^= 2^4 ;
    }
    else
    //含有段标志
    if (t & C_SEG)
    {
      //保存段前缀
      diza->disasm_seg = c ;
    }
    else
    if (t & C_REP)
    {
      //原理同上
      diza->disasm_rep = c ;
    }
    //  LOCK

     goto repeat_prefix ;

  } // C_ANYPREFIX
  //保存标志
  flag |= t ;
  //保存操作码
  diza->disasm_opcode = c ;
  //操作码是否含有第二个字节
  if (c == 0x0F)
  {
    c = *opcode++ ;
    //取出第二个字节
    diza->disasm_opcode2 = c ;
    //根据第二字节取得 指令特征
    flag |= ade32_table[ 256 + c ] ; // 2nd flagtable half

    if (flag == C_ERROR) return 0 ;
  }
  //操作码大概为F7xx
  else
  if (c == 0xF7)
  {
    //opcode已经自加1
```

```
    if (((*opcode) & 0x38)==0)
      //xx为数据（立即数？？）
      flag |= C_DATA66 ;
  }
  //不详，同上
  else
  if (c == 0xF6)
  {
    if (((*opcode) & 0x38)==0)
      flag |= C_DATA1 ;
  }
  //如果设置C_MODRM标志
  if (flag & C_MODRM)
  {
    c = *opcode++ ;
    //保存值
    diza->disasm_modrm = c ;
    //在MorR/M右边含有一个SIB字节
    if ((c & 0x38) == 0x20)
        if (diza->disasm_opcode == 0xFF)
              flag |= C_STOP ;
    //MorR/M的高2位
     BYTE mod = c & 0xC0 ;
    //MorR/M的低3位
     BYTE rm  = c & 0x07 ;
    //如果高2位不为11
    if (mod != 0xC0)
    {
      if (diza->disasm_defaddr == 4)
      {
        //寄存器参与地址计算
        if (rm == 4)
        {
          在MorR/M右边含有一个SIB字节
          flag |= C_SIB ;
          //取出SIB
          c = *opcode++ ;
          //保存
          diza->disasm_sib = c ;
          //取出SIB的低3位，为base寄存器
          rm = c & 0x07 ;
        }
        //操作数为内存地址+字节偏移量，且地址在寄存器中
        if (mod == 0x40)
        {
          flag |= C_ADDR1 ;
        }
        //操作数为内存地址+双字偏移量，同上
        else
        if (mod == 0x80)
        {
          flag |= C_ADDR4 ;
        }
        //mod高2位为00
        else
        {
          //且mod低3位为101，则不使用寄存器计算地址
          if (rm == 5)
            flag |= C_ADDR4 ;
        }
      }
      else // MODRM 16-bit
      {

        if (mod == 0x40)
        {
          flag |= C_ADDR1 ;
        }
        else
        if (mod == 0x80)
        {
          flag |= C_ADDR2 ;
        }
```

```
        else
        {
          if (rm == 6)
            flag |= C_ADDR2 ;
        }
      }
    }
  } // C_MODRM
   //保存标志
  diza->disasm_flag = flag ;
  //取出标志
   DWORD a =  flag & (C_ADDR1 | C_ADDR2 | C_ADDR4) ;
  //#define C_DATA1   0x00000100
  //#define C_DATA2   0x00000200
  //#define C_DATA4   0x00000400
  //移动到低四位
   DWORD d = (flag & (C_DATA1 | C_DATA2 | C_DATA4)) >> 8 ;
  //全部地址长度
  if (flag & C_ADDR67) a += diza->disasm_defaddr ;
  //全都立即数长度
  if (flag & C_DATA66) d += diza->disasm_defdata ;
  //???
  diza->disasm_addrsize = a ;
  diza->disasm_datasize = d ;
  //取出偏移量
   DWORD i ;
  for(i=0 ; i<a; i++)
    diza->disasm_addr_b[i] = *opcode++ ;
  //取出立即数
  for(i=0 ; i<d; i++)
    diza->disasm_data_b[i] = *opcode++ ;
  //长度
  diza->disasm_len = opcode - opcode0 ;

  return diza->disasm_len ;

} // ade32_disasm()

// ade32_asm() -- returns assembled opcode length

int ade32_asm( OUT  BYTE* opcode,   IN  OUT disasm_struct* s)
{
   BYTE* opcode0 = opcode ;

  if (s->disasm_flag & C_SEG)      *opcode++ = s->disasm_seg ;
  if (s->disasm_flag & C_LOCK)     *opcode++ = 0xF0 ;
  if (s->disasm_flag & C_REP)      *opcode++ = s->disasm_rep ;
  if (s->disasm_flag & C_67)       *opcode++ = 0x67 ;
  if (s->disasm_flag & C_66)       *opcode++ = 0x66 ;
  *opcode++ = s->disasm_opcode ;
  if (s->disasm_flag & C_OPCODE2) *opcode++ = s->disasm_opcode2 ;
  if (s->disasm_flag & C_MODRM)    *opcode++ = s->disasm_modrm ;
  if (s->disasm_flag & C_SIB)      *opcode++ = s->disasm_sib ;
  for ( DWORD i=0 ; i<s->disasm_addrsize; i++)
    *opcode++ = s->disasm_addr_b[i] ;
  for ( DWORD i=0 ; i<s->disasm_datasize; i++)
    *opcode++ = s->disasm_data_b[i] ;

  return opcode - opcode0 ;
} // ade32_asm
```