EL LLENGUATGE ASSEMBLADOR DEL 80x86

INTRODUCCIÓ

Fins ara hem vist els mnemònics de les instruccions que passades al corresponent codi binari que ja pot entendre el microprocessador. Si be es realitza un gran avenç al introduir els mnemònics respecte a programar directament en llenguatge màquina - es a dir, amb números en binari o hexadecimal - encara resultaria molt costós tenir que realitzar els càlculs dels desplaçaments en els salts a altres parts del programa en les transferències de control, reservar espai de memòria dins d'un programa per emmagatzemar dades, etc. Per facilitar aquestes operacions s'utilitzen les directives que indiquen a l'assemblador que te de fer amb les instruccions i les dades.

Els programes d'exemple que fem servir i la sintaxi de l'assemblador tractada son les del MASM de Microsoft i de l'assemblador de IBM. De totes formes, tots els programes han estat desenvolupats amb el Turbo Assembler 2.0 de Borland (TASM), compatible amb el clàssic MASM 5.0 de Microsoft però més potent i al mateix temps molt més ràpid i flexible. TASM genera a més a més un codi més reduït i òptim. Per altra part, MASM 5.0 no permet canviar (encara que si la 6.0) dintre d'un segment el mode del processador: això porta el risc d'executar instruccions de 32 bits no desitjades al no poder acotar exactament les línies on es vol emprar-les, una cosa vital per mantenir la compatibilitat amb processadors anteriors. També es propens a generar errors de fase i altres similars al tractar amb llistats una mica grans. Respecta a MASM 6.0, s'ha trobat que en determinades ocasions calcula incorrectament el valor d'alguns símbols i etiquetes, encara que es probable que la versió 6.1 (apareguda sospitosament molt poc temps després) ja hagi corregit aquestes errades, intolerables en un assemblador. Per un altra costat, las possibilitats addicionals de TASM no han estat emprades en general. Molts programes han estat assemblats amb una vegada amb MASM, para assegurar que aquest pot assemblarlos.

Convé dir aquí que tot el que ve pot ésser una mica costos per aquells que no coneixen l'assemblador de cap màquina. La raó es que la informació esta organitzada a mode de referència, per lo que amb freqüència s'utilitzen uns elements - per explicar altres - que encara no han estat definits. Això per altra part resulta inevitable també en algunes referències més bàsiques, degut a la complexitat de la sintaxi del llenguatge assemblador ideada per el fabricant (no la del microprocessador). Por tant, es un bon consell actuar a dues passades, al igual que el propi assemblador en ocasions: llegir tot una vegada primer –encara que no s'entengui del tot- i tornar a llegir després més poc a poc.

SINTAXI D'UNA LÍNIA EN ASSEMBLADOR.

Un programa font en assemblador conte dos tipus de sentencies: les instruccions i les directives. Les instruccions s'apliquen en temps d'execució, però les directives només son utilitzades durant el assemblat. El format d'una sentencia d'instrucció es el següent:

[etiqueta] nom_d'instrucció [operants] [comentari]

Els parèntesi quadrats (square brackets), com es normal al explicar les instruccions en el àmbit informàtic, indiquen que lo especificat entre ells es opcional, depenent de la situació que es tracti.

Camp de etiqueta: Es el nom simbòlic de la primera posició d'una instrucció, apuntador o dada. Consta de fins 31 caràcters que poden ser les lletres de la A a la Z, els números del 0 al 9 i alguns caràcters especials com «@», «_», «.» i «$».

Regles:

- Si s'utilitza el punt «.» aquest es col·loca com a primer caràcter de l'etiqueta.
- El primer caràcter no pot ser un dígit.
- No es poden utilitzar els noms de instruccions o registres com a noms d'etiquetes.

Les etiquetes son de tipus NEAR quan el camp d'etiqueta finalitza amb dos punts (:); així es, es considera pròxima: això vol dir que quan fem una crida sobre aquesta etiqueta l'assemblador considera que esta dins del mateix segment de codi (crides intrasegment) i el processador només carrega el apuntador d'instruccions IP. Cal tenir en compte que parlem de instruccions; les etiquetes emprades abans de les directives, com les directives de definició de dades per exemple, no porten els dos punts i en canvi son pròximes.

Les etiquetes son de tipus FAR si el camp d'etiqueta no acaba amb els dos punts: en aquestes etiquetes la instrucció a la que apunta no es troba en el mateix segment de codi sinó en un altra. Quan es fa referència a una transferència de control es carrega l'apuntador instruccions IP i el segment de codi CS (crides intersegment).

Camp de nom: Conte el mnemònic de les instruccions vistes abans, o be una directiva de les que veurem més endavant.

Camp d'operants: Indica quines son les dades implicades en la operació. N'hi pot haver 0, 1 ó 2; en el cas de que siguin dos al primer se l'anomena destí i al segon - separat per una coma - font.

mov ax, es:[di] --> ax es destí i es:[di] es origen

Camp de comentaris: Quan en una línia hi ha un punt i coma (;) tot el que segueix a la línia es un comentari que vol aclarir una mica el que se esta fent en aquest programa, resulta de gran utilitat de cara a realitzar futures modificacions al mateix.

CONSTANTS I OPERADORS.

Les sentencies font - tant instruccions com directives - poden contenir constants i operadors.

CONSTANTS.

Poden ser binàries (exemple: 10010b), decimals (exemple: 34d), hexadecimals (exemple: 0E0h) o octals (exemple: 21o ó 21q); també n'hi ha de cadena (exemple: 'pepet', "joan") i també amb cometes dintre de cometes de diferent tipus (com 'hola,"amigo"'). En les hexadecimals, si el primer dígit no es numèric s'hi ha de posar un 0. Només es pot posar el signe (-) en les decimals (en les demés, cal calcular el complement a dos). Per defecte, les numèriques estan en base 10 si no s'indica lo contrari amb una directiva (poc recomanable com ja es veurà).

OPERADORS ARITMÉTICS.

Poden emprar-se lliurament (+), (-), (*) i (/) - en aquest últim cas la divisió es sempre entera -. Es vàlida, per exemple, la següent línia en assemblador (que es recolza en la directiva DW, que es veurà més endavant, per reservar memòria per una paraula de 16 bits):

```
dada        DW      12*(numero+65)/7
```

També s'admeten els operadors MOD (resta de la divisió) i SHL/SHR (desplaçar a la esquerra/dreta un cert número de bits). Òbviament, l'assemblador no codifica les instruccions de desplaçament (al aplicar-se sobre dades constants el resultat es calcula en temps d'assemblat):

```
dada        DW      (12 SHR 2) + 5
```

OPERADORS LÓGICS.

Poden ser el AND, OR, XOR i NOT. Realitzen les operacions lògiques a les expressions. Exemple:

```
MOV     BL,(255 AND 128) XOR 128    ; BL = 0
```

OPERADORS RELACIONALS.

Ens retornen les condicions de cert (0FFFFh ó 0FFh) o fals (0) avaluant una expressió. Poden ser: EQ (igual), NE (no igual), LT (menor que), GT (major que), LE (menor o igual que), GE (major o igual que). Exemple:

```
Dada EQU  100            ; «dada» val 100
     MOV  AL,dada GE 10  ; AL = 0FFh (cert)
     MOV  AH,dada EQ 99  ; AH = 0 (fals)
```

OPERADORS DE RETORN DE VALORS.

Operador SEG: Ens retorna el valor del segment de la variable o etiqueta, només es pot emprar a programes de tipus EXE:

```
MOV  AX,SEG taula_dades
```

Operador OFFSET: Ens retorna el desplaçament de la variable o etiqueta en el seu segment:

```
MOV  AX,OFFSET variable
```

Si es desitja obtenir l'offset d'una variable respecte al grup (directiva GROUP) de segments en que esta definida i no respecte al segment concret en que esta definida:

```
MOV  AX,OFFSET nom_grup:variable
```

també es vàlid:

```
MOV  AX,OFFSET DS:variable
```

Operador .TYPE: Ens retorna el mode de l'expressió indicada en un byte. El bit 0 indica mode «relatiu al codi» i el 1 mode «relatiu a dades», si ambdós bits estan inactius significa mode absolut. El bit 5 indica si expressió es local (0 si esta definida externament o indefinida); el bit 7 indica si l'expressió conte una referència externa. El TASM utilitza també el bit 3 per indicar alguna cosa que es desconeix. Aquest operador es útil sobre tot a les macros per determinar el tipus dels paràmetres:

```
Info .TYPE  variable
```

Operador TYPE: Ens retorna la mida (bytes) de la variable indicada. No vàlid en variables DUP:

```
Quilos    DW   76
          MOV  AX,TYPE quilos      ; AX = 2
```

Tractant-se d'etiquetes - en lloc de variables - indica si es llunyana o FAR (0FFFEh) o propera o NEAR (0FFFFh).

Operadors SIZE i LENGTH: Ens retornen la mida (en bytes) o el nombre d'elements, respectivament, de la variable indicada (definida obligatòriament amb DUP):

```
Matriu    DW   100 DUP (12345)
          MOV    AX,SIZE matriu     ; AX = 200
          MOV    BX,LENGTH matriu   ; BX = 100
```

Operadors MASK y WIDTH: informen dels camps d'un registre de bits (veure RECORD).

OPERADORS D'ATRIBUTS.

Operador PTR: torna a definir l'atribut de tipus (BYTE, WORD, DWORD, QWORD, TBYTE) o el de distancia (NEAR o FAR) d'un operant de memòria. Per exemple, si es te una taula definida de la següent manera:

```
Taula     DW   10 DUP (0)          ; 10 paraules a 0
```

Per posar a AL el primer byte de la mateixa, la instrucció MOV AL,taula es incorrecta, dons taula (una cadena 10 paraules) no hi cabrà al registre AL. El que desitja el programador ho te de indicar en aquest cas explícitament a l'assemblador de la següent manera:

```
MOV  AL,BYTE PTR taula
```

Treballant amb varis segments, PTR pot tornar a definir una etiqueta NEAR de un d'ells per convertir-la en FAR des de l'altra, amb l'objecte de poder cridar-la.

Operadors CS:, DS:, ES: y SS: l'assemblador genera un prefix d'un byte que indica al microprocessador el segment que te de fer servir per accedir a les dades a memòria. Per defecte, se suposa DS per els registres BX, DI o SI (o sense registres de base o índex) i SS per SP y BP. Si al accedir a una dada aquest no es troba en el segment per defecte, l'assemblador afegirà el byte addicional de manera automàtica. De totes maneres, el programador pot forçar també aquesta circumstancia:

```
MOV  AL,ES:variable
```

En el exemple, variable se suposa ubicada en el segment extra. Quan es referència una adreça fixa s'ha d'indicar el segment, ja que l'assemblador no coneix a quin segment esta la variable, es un dels pocs casos en que es te d'indicar. Per exemple, la següent línia donarà un error al assemblar:

```
MOV  AL,[0]
```

Per solucionar-ho s'ha d'indicar en quin segment esta la dada (encara que sigui DS):

```
MOV  AL,DS:[0]
```

En aquest últim exemple l'assemblador no generarà el byte addicional ja que les instruccions MOV operen per defecte sobre DS (com casi totes), però ha estat necessari indicar DS per que l'assemblador ens entengui. De totes maneres, en el següent exemple

no es necessari, per que mdada esta declarada en el segment de dades i l'assemblador ho sap:

```
MOV  AL,mdada
```

En general no es molt freqüent la necessitat d'indicar explícitament el segment: al accedir a una variable l'assemblador mira en quin segment esta declarada (veure la directiva SEGMENT) i segons com estiguin assignats els ASSUME, posarà o no el prefix adient segons sigui convenient. Es responsabilitat exclusiva del programador iniciar els registres de segment al principi dels procediments per que el ASSUME no es quedi en tinta mullada... sí s'utilitzen amb força freqüència els prefixes CS a les rutines que gestionen interrupcions (ja que CS es l'únic registre de segment que apunta en principi a les mateixes, fins que es carregui DS o un altra).

Operador SHORT: indica que l'etiqueta a la que es fa referència, de tipus NEAR, pot arribar-s'hi amb un salt curt (-128 a +127 posicions) des de l'actual situació del comptador de programa. A l'assemblador TASM, si se sol·liciten dos passades, posa automàticament instruccions SHORT allà on es possible, per economitzar memòria (el MASM no).

Operador '$': indica la posició de comptador de posicions («Location Counter») utilitzat per l'assemblador dintre del segment per portat el compte de per on ha arribat assemblant. Molt útil:

```
Frase     DB        "simpàtic"
Longitud  EQU       $-OFFSET frase
```

En el exemple, longitud agafarà el valor 9.

Operadors HIGH i LOW: Ens retornen la part alta o baixa, respectivament (8 bits) de expressió:

```
Dada EQU       1025
MOV  AL,LOW    dada      ; AL = 1
MOV  AH,HIGH   dada      ; AH = 4
```

PRINCIPALS DIRECTIVES.

La sintaxi d'una sentencia directiva es molt similar a la d'una sentencia d'instrucció:

[nom] nom_directiva [operants] [comentari]

Només es obligatori el camp «nom_directiva»; els camps han d'estar separats per al menys un espai en blanc. La sintaxi de «nom» es anàloga a la de «etiqueta» de les línies d'instruccions, encara que mai es posa el sufix «:». El camp de comentari compleix també les mateixes normes. A continuació explicarem les directives emprades en els programes exemple i alguna mes, encara que en falta alguna que altra i les explicades no ho estan en tots els casos amb profunditat.

DIRECTIVES DE DEFINICIÓ DE DADES.

DB (definir byte), DW (definir paraula), DD (definir doble paraula), DQ (definir quatre paraules de longitud), DT (definir 10 bytes): serveixen per declarar les variables, assignant-les un valor inicial:

```
Any         DW      1991
Mes         DB      12
Numgros     DD      12345678h
Text        DB      "Hola",13,10
```

Es poden definir números reals de simple precisió (4 bytes) amb DD, de doble precisió (8 bytes) amb DQ i «reals temporals» (10 bytes) amb DT; tots ells amb l'intèrpret format usat per el coprocessador. Per que l'assemblador interpreti el número com real ha de portar el punt decimal:

```
Temperatura     DD      29.72
Esparrecs       DQ      38.9E6
```

Amb operant DUP es poden definir estructures repetitives. Per exemple, per assignar 100 bytes a zero i 25 paraules de contingut indefinit (no importa el que l'assemblador assigni):

```
Zeros       DB      100 DUP (0)
Brossa      DW      25 DUP (?)
```

S'admeten també els andinament. El següent exemple crea una taula de bytes on es repeteix 50 veces la seqüència 1,2,3,7,7:

```
Taula       DB      50 DUP (1, 2, 3, 2 DUP (7))
```

DIRECTIVES DE DEFINICIÓ DE SÍMBOLS.

EQU (EQUivalence): Assigna el valor d'una expressió a un nom simbòlic fix:

```
Olimpiades     EQU    1992
```

On olimpiades ja no podrà canviar de valor en tot el programa. Es tracta d'un operador molt flexible. Es vàlid fer:

```
Edat EQU    [BX+DI+8]
     MOV    AX,edat
```

= (signe '='): assigna el valor de la expressió a un nom simbòlic variable: Anàlogament a l'anterior però amb possibilitat de canviar en el futur. Molt usada en macros (sobre tot amb REPT).

```
Num  = 19
Num  = pepet + 1
Dada = [BX+3]
Dada = ES:[BP+1]
```

DIRECTIVES DE CONTROL DEL ASSEMBLADOR.

ORG (ORiGin): posa el comptador de posicions del assemblador, que indica l'offset on es diposita la instrucció o dada, on s'indiqui. En els programes COM (que es carreguen en memòria amb un OFFSET 100h) es necessari col·locar al principi un ORG 100h, i un ORG 0 en els controladors de dispositiu (si s'omet però s'assumeix de fet un ORG 0).

END [expressió]: indica el final de l'arxiu font. Si s'inclou, expressió indica el punt on arranca el programa. Es pot ometre en els EXE si aquests consten d'un sol mòdul. En els COM es necessari indicar-la i, a més a mes, expressió - realment una etiqueta - te d'estar immediatament després del ORG 100h.

.286, .386 Y .8087 obliguen a l'assemblador a reconèixer instruccions específiques del 286, el 386 i del 8087. També s'ha de posar el «.» inicial. Amb .8086 es força a que novament només es reconeguin instruccions del 8086 (mode per defecte). La directiva .386 pot ser col·locada dintre d'un segment (entre les directives SEGMENT/ENDS) amb l'assemblador TASM, el que permet emprar instruccions de 386 con segments de 16 bits; alternativament es poden ubicar fora dels segments (obligatori en MASM) i definir aquests explícitament com de 16 bits amb USE16.

EVEN: força el comptador de posicions a una posició parell, intercalant un byte amb la instrucció NOP si es precís. A bussos de 16 ó més bits (8086 i superiors, no en el 8088) es dos vegades més ràpid l'accés a paraules en posició parell:

```
                EVEN
dada_ràpida     DW      0
```

.RADIX n: canvia la base de numeració per defecte. No es molt aconsellable per que donat que la notació escollida per indicar les bases per part de IBM/Microsoft (si se canvia la base per defecte a 16, ¡Els números no poden acabar en 'd' ja que es confondrien amb el sufix de decimal!: lo ideal seria emprar un prefix i no un sufix, que molt sovint obliga a més a més a iniciar els números per 0 per distingir-los de les etiquetes).

DIRECTIVES DE DEFINICIÓ DE SEGMENTS I PROCEDIMENTS.

SEGMENT-ENDS: SEGMENT indica el començament d'un segment (codi, dades, pila, etc.) i ENDS el seu fi. El programa més simple, de tipus COM, necessita la declaració de un segment (comú per dades, codi i pila). Junt a SEGMENT pot aparèixer, opcionalment, el tipus de alineament, la combinació, l'ús i la classe:

```
nom  SEGMENT [alineament] [combinació] [us] ['classe']
                .     .     .     .
nom  ENDS
```

Es poden definir uns segments dins d'altres (l'assemblador els ubicarà uns rera els altres. Alineament pot ser BYTE (cap), WORD (el segment comença en posició parell), DWORD (comença en posició múltiple de 4), PARA (comença en una adreça múltiple de 16, opció per defecte) i PAGE (comença a una adreça múltiple de 256). La combinació pot ser:

(No indicada): els segments es posen uns rera els altres físicament, però son lògicament independents: cada un te la seva pròpia base i els seus propis offsets relatius.

PUBLIC: usada especialment quan es treballa amb segments definits en varis arxius que s'assemblen per separat o es compilen amb altres llenguatges, per això es te de declarar un nom entre cometes simples -'classe'- per ajudar al linker. Tots els segments PUBLIC d'igual nom i classe tenen una base comú i son col·locades adjacentment uns rera altres, sent l'offset relatiu al primer segment carregat.

COMMON: similar, encara que ara els segments d'igual nom i classe es trepitgin. Per això, les variables declarades han de ser-ho en el mateix ordre i mida.

AT: associa un segment a una posició de memòria fixa, no per assemblar sinó per declarar variables (iniciades sempre amb '?') de cara a accedir amb comoditat a zones de ROM, vectors d'interrupció, etc. Exemple:

```
vars_bios   SEGMENT AT 40h
p_serie0    DW ?
vars_bios   ENDS
```

D'aquesta manera, l'adreça del primer port sèrie es pot obtenir d'aquesta manera (per exemple):

```
MOV     AX,variables_bios     ; segment
MOV     ES,AX                 ; iniciar ES
MOV     AX,ES:p_serie0
```

STACK: segment de pila, en te d'existir un en els programes de tipus EXE; a més a més el Linkador de Borland (TLINK 4.0) exigeix obligatòriament que la classe d'aquest sigui també 'STACK', amb el LINK de Microsoft no sempre es necessari indicar la classe del segment de pila. Similar, per el demés, a PUBLIC.

MEMORY: segment que el linker ubicarà al final de tots els demés, lo que permetria saber on acaba el programa. Si es defineixen varis segments de aquest tipus l'assemblador accepta el primer i tracta als demés com COMMON. Cal tenir en compta que el linker no suporta aquesta característica, per el que emprar MEMORY es equivalent a tots els efectes a utilitzar COMMON. Cal oblidar-se de MEMORY.

L'ús indica si el segment es de 16 bits o de 32; al emprar la directiva .386 s'assumeixen per defecte segments de 32 bits per el que es necessari declarar USE16 per aconseguir que els segments siguin interpretats com de 16 bits per el linkador, el que permet emprar alguns instruccions del 386 en el mode real del microprocessador i sota el sistema operatiu DOS.

Per últim, 'classe' es un nom opcional que utilitzarà el linker per encadenar els mòduls, essent convenient nomenar la classe del segment de pila amb 'STACK'.

ASSUME (Suposar): Indica a l'assemblador el registre de segment que s'utilitzarà per adreçar cada segment dins del mòdul. Aquesta instrucció va normalment immediatament després del SEGMENT. El programa més senzill necessita que se «suposi» CS com a mínim per el segment de codi, de lo contrari l'assemblador començarà a protestar al no saber quin registre de segment associar al codi generat. També convé fer un ASSUME del registre de segment DS cap el segment de dades, inclòs en el cas de que aquest sigui el mateix que el de codi: si no, l'assemblador col·locarà un byte de prefix addicional a tots els accessos a memòria per forçar que aquests siguin sobre CS. Es pot indicar ASSUME NOTHING per cancel·lar un ASSUME anterior.
També es pot indicar el nom d'un grup o emprar «SEG variable» o «SEG etiqueta» en lloc de nom_segment:

```
ASSUME reg_segment:nom_segment[,...]
```

PROC-ENDP permet donar nom a una subrutina, marcant amb claredat el seu inici i el seu fi. Encara que es redundant, es molt recomanable per estructurar els programes.

```
cls     PROC
        ...
cls     ENDP
```

L'atribut FAR que apareix a vegades al costat de PROC indica que es un procediment llunya i les instruccions RET dins d'aquest s'assemblaran com RETF (els CALL cap a ell seran de 32 bits). Observar que la etiqueta mai acaba amb dos punts.

DIRECTIVES DE REFERENCIES EXTERNES.

PUBLIC: permet fer visibles a l'exterior (altres fitxers objecte resultants d'altres llistats en assemblador o altra llenguatge) els símbols - variables i procediments - indicats. Necessari per la programació modular i interfícies amb llenguatges d'alt nivell. Per exemple:

```
PUBLIC proc1, var_x
proc1       PROC   FAR
            ...
proc1       ENDP
var_x       DW     0
```

Declara la variable var_x i el procediment proc1 com accessibles des de l'exterior a traves de la directiva EXTRN.

EXTRN: Permet accedir a símbols definits en un altra fitxer objecte (resultant d'un altra assemblat o d'una compilació d'un llenguatge d'alt nivell); es necessari també indicar el tipus de dada o procediment (BYTE, WORD o DWORD; NEAR o FAR; s'utilitza també ABS per les constants numèriques):

```
EXTRN proc1:FAR, var_x:WORD
```

En el exemple s'accedeix als símbols externs proc1 i var_x (veure exemples de PUBLIC) i a continuació seria possible fer un CALL proc1 o un MOV CX,var_x. Si la directiva EXTRN es posa dins d'un segment, suposem el símbol dins del mateix. Si el símbol esta en un altra segment, es te de col·locar EXTRN fora de tots els segments indicant explícitament el prefix del registre de segment (o be fer el ASSUME apropiat) al fer-lo referència. Evidentment, al final, al linkar s'haurà que enllaçar aquest mòdul amb el que defineix els elements externs.

INCLUDE nom_fitxer: Afegeix al fitxer font en procés d'assemblat el fitxer indicat, en el punt en que apareix el INCLUDE. Es exactament el mateix que barrejar ambdós fitxers amb un editor de textos. Estalvia feina en fragments de codi que es repeteixen en varis programes (com pot ser una llibreria de macros). No se recomanen INCLUDE's niats.

DIRECTIVES DE DEFINICIÓ DE BLOCS.

NAME nom_modul_objete: indica el nom del mòdul objecte. Si no s'inclou NAME, s'agafarà de la directiva TITLE o, en el seu defecte, del nom del propi fitxer font.

GROUP segment1, segment2,... permet agrupar dos o més segments lògics en un sol de no més de 64 Kb totals (ull viu: l'assemblador no comprova aquest extrem, encara que sí l'enllaçador 'linker'). Exemple:

```
Superseg  GROUP dades, codi, pila

Codi      SEGMENT
             ...
codi      ENDS

dades     SEGMENT
dada      DW 1234
dades     ENDS

pila      SEGMENT STACK 'STACK'
          DB 128 DUP (?)
Pila      ENDS
```

Quan s'accedeix a una dada definida a algun segment d'un grup i s'utilitza l'operador OFFSET fera falta indicar el nom del grup com a prefix, si no l'assemblador no generarà el desplaçament correcte ¡ni emetrà errors!:

```
MOV    AX,dada                ; ¡incorrecte!
MOV    AX,supersegment:dada   ; correcte
```

L'avantatge d'agrupar segments es poder crear programes COM i SYS que continguin varis segments. En tot cas, cal tenir en compte encara en aquest cas que no poden emprar-se totes les característiques de la programació amb segments (per exemple, no es pot utilitzar la directiva SEG ni te d'existir segment de pila).

LABEL: Permet fer referència d'un símbol amb un altra nom, sent factible tornar a definir el tipus. La sintaxi es: nom LABEL tipus (tipus = BYTE, WORD, DWORD, NEAR o FAR). Exemple:

```
paraula     LABEL  WORD
byte_baix   DB     0
byte_alt    DB     0
```

En el exemple, amb MOV AX,paraula s'accedirà a ambdós bytes a la vegada (el us de MOV AX,byte_baix donaria error: no es pot carregar un sol byte en un registre de 16 bits i l'assemblador no suposa que realment pretenïem prendre dos bytes consecutius de la memòria).

STRUC - ENDS: permet definir registres a l'estil dels llenguatges d'alt nivell, per accedir d'una manera més elegant als camps d'una informació amb certa estructura. Aquests camps poden composar-se de qualsevol dels tipus de dades simples (DB, DW, DD, DQ, DT) i poden ser modificables o no en funció de si son simples o múltiples, respectivament:

```
alumne      STRUC
mot            DB '0123456789'     ; modificable
edataltura     DB 20,175           ; no modificable
pes            DB 0                ; modificable
altres         DB 10 DUP(0)        ; no modificable
telefon        DD ?                ; modificable
alumne      ENDS
```

La anterior definició d'estructura no porta implícita la reserva de memòria necessària, que s'ha de fer expressament utilitzant els angles '<' i '>':

```
felip      alumne <'Gordinflas',,101,,251244>
```

En el exemple se defineixen els camps modificables (els únics definibles) deixant sense definir (comes consecutives) els no modificables, creant-se l'estructura 'felip' que ocupa 27 bytes. Les cadenes de caràcters son emplenades amb espais en blanc al final si no arriben a la mida màxima de la declaració. El TASM es més flexible i permet definir també el primer element dels camps múltiples sense donar error. Després de crear l'estructura, es possible accedir als seus elements utilitzant un (.) per separar el nom del camp:

```
MOV    AX,OFFSET felip.telefon
LEA    BX,felip
MOV    CL,[BX].pes      ; equival a [BX+12]
```

RECORD: similar a STRUC però operant amb camps de bits. Permet definir una estructura determinada de byte o paraula per operar amb comoditat.
Sintaxi:

```
nom      RECORD nom_de_camp:mida[=valor],...
```

On nom permetrà fer referència a l'estructura en el futur, nom_de_camp identifica els diferents camps, als que assigna una mida (en bits) i opcionalment un valor per defecte.

```
registre    RECORD a:2=3, b:4=5, c:1
```

L'estructura registre totalitza 7 bits, per tant ocupa un byte. Esta dividida en tres camps que ocupen els 7 bits menys significatius del byte: el camp A ocupa els bits 6 y 5, el B els bits del byte: el campo A ocupa els bits del 1 al 4 i el C el bit 0:

```
6 5 4 3 2 1 0
1 1 0 1 0 1 ?
```

La reserva de memòria es realitza, per exemple, de la següent manera:

```
reg1        registre <2,,1>
```

Quedant reg1 amb el valor binari 1001011 (el camp B roman sense alteracions i A i C prenen els valors indicats). Exemples d'operacions suportades:

```
MOV  AL, A          ; AL = 5 (desplaçament del bit
                    ;      menys significatiu de A)
MOV  AL, MASK A     ; AL = 01100000b (màscara de A)
MOV  AL, WIDTH A    ; AL = 2 (amplada de A)
```

DIRECTIVES CONDICIONALS.

S'utilitzen per que l'assemblador avaluï unes condicions i, segons elles, assembli o no certes zones de codi. Es freqüent, per exemple, de cara a generar codi per varis ordinadors: poden existir certs símbols definits que indiquen en un moment donat si hi ha d'assemblar certes zones del llistat o no de manera condicional, segons la màquina. En els fragments en assemblador del codi que generen els compiladors també semblen actuar de manera diferent, per exemple, segons el model de memòria). Es interessant també la possibilitat de definir un símbol que indiqui que el programa esta en fase de proves i assemblar codi addicional en aquest cas amb objecte de depurar-lo. Sintaxi:

```
Ifxxx     [símbol/exp./arg.]  ; xxx es la condició
...
ELSE                          ; el ELSE es opcional
...
ENDIF
```

```
IF     expressió (expressió diferent de zero)
IFE    expressió      (expressió igual a zero)
IF1    (passada 1 de l'assemblador)
IF2    (passada 2 de l'assemblador)
IFDEF  símbol (símbol definit o declarat com extern)
IFNDEF símbol    (símbol ni definit ni declarat com extern)
IFB  <argument> (arg. en blanc en macros -incloure '<' i '>'-)
IFNB <argument> (lo contrari, també es obligat posar '<' i '>')
IFIDN <arg1>, <arg2> (arg1 idèntic a arg2, requereix '<' i '>')
IFDIF <arg1>, <arg2> (arg1 diferent de arg2, req. '<' i '>')
```

DIRECTIVES DE LLISTAT.

PAGE num_linies, num_columnes: Dona Format el llistat de sortida; per defecte son 66 línies per pàgina (modificable entre 10 i 255) i 80 columnes (es pot seleccionar de 60 a 132). PAGE salta de pàgina i incrementa el seu valor. «PAGE +» indica capítol nou (i s'incrementa el valor).

TITLE títol: indica el títol que apareix a la 1ª línia de cada pagina (màxim 60 caràcters).

SUBTTL subtítol: indica el subtítol que apareix a cada pàgina (màxim 60 caràcters).

.LALL: Llistar les macros i les seves expansions.

.SALL: No llistar les macros ni les seves expansions.

.XALL: Llistar solament les macros que generen codi objecte.

.XCREF: Suprimir llistat de referències creuades (llistat alfabètic de símbols juntament al nombre de línia en que son definides i fan referència, de cara a facilitar la depuració).

.CREF: Restaurar llistat de referències creuades.

.XLIST: Suprimir el llistat assemblador des d'aquest punt.

.LIST: Restaurar de nou la sortida de llistat assemblador.

COMMENT <caràcter que delimita> comentari <caràcter que delimita>: Defineix un comentari que pot ocupar varies línies, el caràcter que delimita (primer caràcter no blanc ni tabulador que segueix al COMMENT) indica l'inici i indicarà més tard el final del comentari. ¡No oblidar tancar el comentari!.

%OUT missatge: escriu a la consola el missatge indicat durant la fase d'assemblat i al arribar a aquest punt del llistat, excepte quan el llistat es per pantalla i no a un fitxer.

.LFCOND: Llistar els blocs de codi associats a una condició falsa (IF).

.SFCOND: Suprimir el llistat dels blocs de codi associats a una condició falsa.

.TFCOND: Invertir el mode vigent de llistat dels blocs associats a una condició falsa.

MACROS.

Quan un conjunt d'instruccions en assemblador apareixen freqüentment repetides en un llistat, es convenient agrupar-les sota un nom simbòlic que les substituirà en aquells punts on apareguin. Aquesta es la missió de les macros; per el fet de suportar-les l'assemblador eleva la seva categoria a la de macroassemblador, al ser les macros una eina molt cotitzada per els programadors.
No convé confondre les macros amb subrutines: son aquestes últimes, el conjunt instruccions apareix una sola vegada en tot el programa i que es criden amb CALL. En canvi, cada vegada que es referència a una macro, el codi que aquesta representa s'expandeix en el programa definitiu, es duplica tantes vegades com s'utilitzi la macro. Per això, aquelles tasques que puguin ser realitzades amb subrutines sempre serà més convenient realitzar-les amb les mateixes, per economitzar memòria. Es cert que les macros son quelcom més ràpides que les subrutines (s'estalvia un CALL i un RET) però la diferencia es tan mínima que a la practica es pot menysprear en el 99,99% dels casos. Por això, es absurd i irracional realitzar certes tasques amb macros que poden ser desenvolupades molt més eficientment amb subrutines: es una pena que en molts manuals d'assemblador encara es parli de macros per realitzar operacions sobre cadenes de caràcters, que generarien programes gegants amb menys d'un 1% de velocitat addicional.

DEFINICIÓ I ESBORRAT DE LES MACROS.

La macro es defineix a traves de la directiva MACRO. Es necessari definir la macro abans de ser utilitzada. Una macro en pot cridar una altra. Sovint, les macros es posen juntes en un fitxer independent i llavors s'inclouen en el programa principal amb la directiva INCLUDE:

```
IF1
     INCLUDE fitxer.ext
ENDIF
```

La sentencia IF1 assegura que l'assemblador llegeix el fitxer font de les macros només a la primera passada, per accelerar l'assemblat i evitar que apareguin en el llistat (generat a la segona fase). Convé remarcar que la definició de la macro no consumeix memòria, per el que a la pràctica es indiferent declarar-ne cents que cap macro:

```
nom_simbólic MACRO [paràmetres]
              ...
              ...          ; instruccions de la macro
             ENDM
```

El nom simbòlic es el que permetrà més endavant fer referència a la macro, i es construeix casi amb les mateixes regles que els noms de les variables i demés símbols. La macro pot contenir paràmetres de manera opcional. A continuació venen les instruccions que engloba y, finalment, la directiva ENDM assenyala el final de la macro. No se s'ha de repetir el nom simbòlic juntament a la directiva ENDM, això provocaria un error un tant curiós i estrany per part de l'assemblador (Quelcom així com «Fi del fitxer font inesperat, falta la directiva END»), al menys amb MASM 5.0 i TASM 2.0.

En la realitat, i a diferencia del que passa amb els demés símbols, el nom d'una macro pot coincidir amb el d'una instrucció màquina o una directiva d'assemblador: a partir d'aquest moment, la instrucció o directiva coincident perd el seu significat original. L'assemblador donarà a més a més un avis de advertència si s'utilitza una instrucció o directiva com nom de macro, però tolerarà la operació. Normalment s'assignarà noms normals, com a les variables. I, si alguna vegada tornem a definir una instrucció màquina o directiva, per restaurar el significat original del símbol, la macro pot ser esborrada o simplement per que ja no s'utilitzarà a partir d'un cert punt del llistat, i així ja no consumirà espai a las taules de macros que manté a memòria l'assemblador al assemblar. No es necessari esborrar les macros abans de tornar a definir-les. Per esborrar-les, la sintaxi es la següent:

```
PURGE  nom_simbólic[,nom_simbólic,...]
```

EXEMPLE D'UNA MACRO SENZILLA.

Des del 286 existeix una instrucció molt còmoda que introdueix a la pila 8 registres, i un altra que els treu (PUSHA i POPA). Qui estigui acostumat a emprar-les, pot crear unes macros que simulin aquestes instruccions en els 8086:

```
SUPERPUSH   MACRO
            PUSH   AX
            PUSH   CX
            PUSH   DX
            PUSH   BX
            PUSH   SP
            PUSH   BP
            PUSH   SI
            PUSH   DI
            ENDM
```

La creació de SUPERPOP es anàloga, es treuen els registres en ordre invers. L'ordre escollit no es per capritx i es correspon amb el de la instrucció PUSHA original, per fer-ho compatible. A partir de la definició d'aquesta macro, tenim a la nostra disposició una nova instrucció màquina (SUPERPUSH) que pot ser usada amb llibertat dins dels programes.

PARÁMETRES FORMALS I PARÁMETRES ACTUALS.

Per qui no hagi tingut relació prèvia amb algun llenguatge estructurat d'alt nivell, farem un breu comentari sobre del que son els paràmetres formals i actuals en una macro, similar aquí als procediments dels llenguatges d'alt nivell.

Quan es crida a una macro es poden passar opcionalment un cert nombre de paràmetres de cert tipus. Aquests paràmetres es denominen paràmetres actuals. A la definició de la macro, aquests paràmetres apareixen associats a certs noms arbitraris, la seva única missió es permetre distingir uns paràmetres dels altres i indicar en quin ordre son passats: son els paràmetres formals. Quan l'assemblador expandeixi la macro al assemblar, els paràmetres formals seran substituïts per els seus corresponents paràmetres actuals. Considerar el següent exemple:

```
SUMAR       MACRO  a,b,total
            PUSH   AX
            MOV    AX,a
            ADD    AX,b
            MOV    total,AX
            POP    AX
            ENDM
            ....
            SUMAR positius, negatius, total
```

En el exemple, «a», «b» i «total» son los paràmetres formals i «positius», «negatius» i «total» son els paràmetres actuals. Tant «a» com «b» poden ser variables, etiquetes, etc. en altra punt del programa; Però, dins de la macro, es comporten de manera independent. El paràmetre formal «total» ha coincidit en el exemple i per casualitat amb el seu corresponent actual. El codi que genera l'assemblador al expandir la macro serà el següent:

```
        PUSH    AX
        MOV     AX,positius
        ADD     AX,negatius
        MOV     total,AX
        POP     AX
```

Les instruccions PUSH i POP serveixen per no alterar el valor de AX i aconseguir que la macro es comporti com una caixa negra; no es necessari que això sigui així però es un bon costum de programació per evitar que els programes facin coses rares. En general, les macros d'aquest tipus no haurien d'alterar els registres i, si els canvien, s'ha de tenir molt clar quins.

Si s'indiquen més paràmetres dels que una macro necessita, se ignoraran els restants. En canvio, si en falten, el MASM assumirà que son nuls (0) i donarà un missatge advertència, el TASM es una mica més rígid i podria donar un error. En general, es tracta de situacions atípica que tenen de ser evitades.

També es pot donar el cas de que no sigui possible expandir la macro. En el exemple, no ves estat possible executar SUMAR AX,BX,DL per que DL es de 8 bits i la instrucció MOV DL,AX seria il·legal.

ETIQUETES DINTRE DE MACROS. VARIABLES LOCALS.

Son necessàries normalment per els salts condicionals que continguin les Marcos més complexes. Si es posa una etiqueta a on saltar, la marco solament podria ser emprada una sola vegada en tot el programa per evitar que aquesta etiqueta estigui duplicada. La solució esta en emprar la directiva LOCAL que ha de anar col·locada justament després de la directiva MACRO:

```
 MINIM      MACRO  dada1, dada2, resultat
            LOCAL  ja_esta
            MOV    AX,dada1
            CMP    AX,dada2       ; ¿es dada1 el menor?
            JB     ja_esta        ; sí
            MOV    AX,dada2       ; no, es dada2
ja_esta:    MOV    resultat,AX
            ENDM
```

En el exemple, al invocar la macro dos vegades l'assemblador no generarà l'etiqueta «ja_esta» sinó les etiquetes ??0000, ??0001, ... i així successivament. La directiva LOCAL no es solament útil per els salts condicionals a les macros, i també permet declarar variables internes. Es pot indicar un nombre casi indefinit d'etiquetes amb la directiva LOCAL, separánt-les per comes.

OPERADORS DE MACROS.

Operador ;;
Indica que el que ve a continuació es un comentari i que no te aparèixer al expansionar la macro. Quan a l'assemblar es genera un llistat del programa, les macros solen aparèixer expandides en els punts en els que se invoquen; Però només apareixeran els comentaris normales que comencin per (;). Els comentaris relacionats amb el funcionament intern de la macro haurien d'anar amb (;;), els relatius al us i sintaxi de la mateixa amb (;). Això es convenient perquè durant l'assemblat son mantinguts a memòria els comentaris de Marcos (no els de la resta del programa) que comencen per (;), i no convé malmetre memòria...

Operador &
Utilitzat per concatenar text o símbols. Es necessari per aconseguir que l'assemblador substitueixi un paràmetre dins d'una cadena de caràcters o com part d'un símbol:

```
SALUTACIO       MACRO  c
                MOV    AL,"&c"
etiqueta&c:     CALL   imprimir
                ENDM
```

Al executar SALUTACIO A es produirà la signeu expansió:

```
                MOV    AL,"A"
etiquetaA:      CALL   imprimir
```

Si no hi poséssim el & hauríem expandit com `MOV AL,"c"`

Quan s'utilitzen estructures repetitives REPT, IRP o IRPC (que es veuran una mica més endavant) existeix un problema addicional al intentar crear etiquetes, per que l'assemblador es menja un & al fer la primera substitució, generant la mateixa etiqueta a menys que es dupliqui l'operador &:

```
MEMORIA         MACRO  x
                IRP    i, <1, 2>
x&i             DB   i
                ENDM
                ENDM
```

Si s'invoca MEMORIA ET es produeix l'error "etiqueta ETi repetida", que es pot salvar afegint tants '&' com nivells de niament troba a les estructures repetitives emprades, com se exemplifica a continuació:

```
MEMORIA         MACRO  x
                IRP    i, <1, 2>
x&&i            DB   i
                ENDM
                ENDM
```

El que amb MEMORIA ET generarà correctament les línies:

```
ET1         DB 1
ET2         DB 2
```

Operador ! o <>
Emprat per indicar que el caràcter que ve a continuació te de ser interpretat literalment i no com un símbol. Por això, !; es equivalent a <;>.

Operador %
Converteix expressió que l'hi segueix - generalment un símbol - a un número; expressió te de ser una constant (no relocalitzable). Només se empra en els arguments de macros. Donada la macro següent:

```
PSUM        MACRO  missatge, suma
            %OUT   * missatge, suma *
            ENDM
```

(Evidentment, el % que precedeix a OUT forma part de la directiva i no es tracta del % operador que estem tractant)

Suposada l'existència dels símbols:

```
SIM1        EQU    120
SIM2        EQU    500
```

Invocant la macro amb les següents condicions:

```
PSUM        < SIM1 + SIM2 = >, (SIM1+SIM2)
```

Es produeix la següent expansió:

```
%OUT * SIM1 + SIM2 = (SIM1+SIM2) *
```

En canvi, invocant la macro de la següent manera (amb %):

```
PSUM < SIM1 + SIM2 = >, %(SIM1+SIM2)
```

Es produeix l'expansió desitjada:

```
%OUT * SIM1 + SIM2 = 620 *
```

DIRECTIVES ÚTILS PER MACROS.

Aquestes directives poden ser emprades també sense les macros, augmentant la comoditat de la programació, encara que abunden especialment dins de les macros.

```
REPT veces ... ENDM   (Repeat)
```

Permet repetir cert nombre de vegades una seqüència instruccions. El bloc d'instruccions es delimita amb ENDM (no confondre-ho amb el final d'una macro). Per exemple:

```
REPT   2
   OUT  DX,AL
ENDM
```

Aquesta seqüència es transformarà, a l'assemblar, en el següent:

```
OUT    DX,AL
OUT    DX,AL
```

Emprant símbols definits amb (=) i ajudant-se amb les macros es pot arribar a crear pseudo-instruccions molt potents:

```
SUCCESIO    MACRO n
            num = 0
            REPT n
              DB num
              num = num + 1
            ENDM              ; fi de REPT
            ENDM              ; fi de macro
```

La sentencia SUCCESIO 3 provocarà la següent expansió:

```
            DB     0
            DB     1
            DB     2
```

IRP simbol_control, <arg1, arg2, ..., arg_n> ... ENDM (Indefinite repeat)

Es relativament similar a la instrucció FOR dels llenguatges d'alt nivell. Els angles (<) i (>) son obligatoris. El símbol de control va prenent successivament els valors (no necessàriament numèrics) arg1, arg2, ... i passa per tot el bloc d'instruccions en cada passada fins arribar a ENDM (no confondre amb fi de macro) substituint simbol_control per aquests valors en tots els llocs en que apareix:

```
            IRP    i, <1,2,3>
              DB   0, i, i*i
            ENDM
```

Al expansionar-se, aquest conjunt d'instruccions es converteix en el següent:

```
        DB      0, 1, 1
        DB      0, 2, 4
        DB      0, 3, 9
```

Nota: Tot el tancat entre els angles es considera un únic paràmetre. Un (;) dins dels angles no s'interpreta com l'inici d'un comentari sinó com un element mes. Per altra part, al emprar macros niades, es tenen de indicar tants símbols angulars '<' i '>' consecutius com nivells de niament existeixen.

Lògicament, dins d'una macro també resulta força útil l'estructura IRP:

```
TETRAOUT    MACRO  p1, p2, p3, p4, valor
            PUSH   AX
            PUSH   DX
            MOV    AL,valor
            IRP    cn, <p1, p2, p3, p4>
              MOV  DX, cn
              OUT  DX, AL
            ENDM                  ; fi de IRP
            POP    DX
            POP    AX
            ENDM                  ; fi de macro
```

Al executar TETRAOUT 318h, 1C9h, 2D1h, 1A4h, 17 s'obtindrà:

```
            PUSH   AX
            PUSH   DX
            MOV    AL, 17
            MOV    DX, 318h
            OUT    DX, AL
            MOV    DX, 1C9h
            OUT    DX, AL
            MOV    DX, 2D1h
            OUT    DX, AL
            MOV    DX, 1A4h
            OUT    DX,AL
            POP    DX
            POP    AX
```

Quan es passen llistes com paràmetres s'han de tancar entre '<' i '>' al cridar-la, per no confondre-les amb elements independents. Per exemple, suposada la macro INCD:

```
INCD        MACRO  llista, p
            IRP    i, <llista>
              INC  i
            ENDM                  ; fi de IRP
            DEC    p
            ENDM                  ; fi de macro
```

Es compren la necessitat d'utilitzar els angles.

INCD AX, BX, CX, DX s'expandirà:

```
INC     AX
DEC     BX ; CX y DX s'ignoren (4 paràmetres)
```

INCD <AX, BX, CX>, DX s'expandirà:

```
INC     AX
INC     BX
INC     CX
DEC     DX              ; (2 paràmetres)
```

IRPC simbol_control, <c1c2 ... cn> ... ENDM (Indefinite repeat character)

Aquesta directiva es similar a l'anterior, amb una petita matisació: els elements situats entre els angles (<) i (>) - ara opcionals, per cert - son caràcters ASCII i no van separats per comes:

```
IRPC    i, <813>
  DB    i
ENDM
```

El bloc anterior generarà al expandir-se:

```
DB      8
DB      1
DB      3
```

Exemple d'utilització dins d'una macro (en combinació amb l'operador &):

```
INICIALITZA  MACRO  a, b, c, d
             IRPC   iter, <&a&b&c&d>
               DB   iter
             ENDM                   ; fi de IRPC
             ENDM                   ; fi de macro
```

Al executar INICIALITZA 7, 1, 4, 0 es produeix la següent expansió:

```
DB      7
DB      1
DB      4
DB      0
```

EXITM

Serveix per avortar l'execució d'un bloc MACRO, REPT, IRP ó IRPC. Normalment s'utilitza recolzant-se en una directiva condicional (IF...ELSE...ENDIF). Al sortir del bloc, es passa al nivell immediatament superior (que pot ser un altra bloc). Com exemple, la següent macro reserva n bytes de memòria a zero fins un màxim de 100, posant un byte 255 al final del bloc reservat:

```
MALLOC      MACRO n
            maxim=100
            REPT n
              IF maxim EQ 0        ; ja van 100?
                EXITM              ; abandonar REPT
              ENDIF
              maxim = maxim - 1
              DB 0                 ; reservar byte
            ENDM
            DB 255                 ; byte de fi de bloc
            ENDM
```

MACROS AVANÇADES AMB NOMBRE VARIABLE DE PARÁMETRES.

Com s'ha vist al estudiar la directiva IF, existeix la possibilitat de comprovar condicionalment la presencia d'un paràmetre a traves de IFNB, o la seva absència amb IFB. Unint això a la potencia de IRP es possible crear macros extraordinàriament versàtils. Com exemple, valgui la següent macro, destinada a introduir a la pila un nombre variable de paràmetres (fins 10): es especialment útil en els programes que gestionen interrupcions:

```
XPUSH       MACRO R1,R2,R3,R4,R5,R6,R7,R8,R9,R10
            IRP reg, <R1,R2,R3,R4,R5,R6,R7,R8,R9,R10>
              IFNB <reg>
                PUSH reg
              ENDIF
            ENDM                   ; fi de IRP
            ENDM                   ; fi de XPUSH
```

Per exemple, la instrucció:

```
            XPUSH  AX,BX,DS,ES,VAR1
```

S'expandirà a:

```
            PUSH   AX
            PUSH   AX
            PUSH   DS
            PUSH   ES
            PUSH   VAR1
```

L'exemple anterior es il·lustratiu del mecanisme de comprovació de presencia de paràmetres. Així, Aquest exemple es pot millorar notablement emprant una llista com únic paràmetre:

```
XPUSH       MACRO llista
            IRP i, <llista>
              PUSH i
            ENDM
            ENDM

XPOP        MACRO llista
            IRP i, <llista>
              POP i
            ENDM
            ENDM
```

L'avantatja es el nombre indefinit de paràmetres suportats (no nomes 10). Un exemple d'us pot ser el següent:

```
XPUSH  <AX, BX, CX>
XPOP   <CX, BX, AX>
```

Que al expandir-se queda:

```
PUSH   AX
PUSH   BX
PUSH   CX
POP    CX
POP    BX
POP    AX
```

PROGRAMACIÓ MODULAR I PAS DE PARÁMETRES.

Encara que el que ve a continuació no es indispensable per programar en assemblador, sí es convenient llegir-lo en 2 ó 3 minuts per observar certes regles molt senzilles que ajudaran a fer programes segurs i eficients.

La programació modular consisteix en dividir els problemes més complexes en mòduls separats amb unes certes interdependències, lo que redueix el temps de programació i augmenta la fiabilitat del codi. Es poden implementar en assemblador amb les directives PROC i ENDP que, encara que no generen codi son força útils per deixar ben clar on comença i acaba un mòdul. Regles per la bona programació:

Dividir els problemes en mòduls petits relacionats només per un conjunt de paràmetres d'entrada i sortida.

Una sola entrada i sortida en cada mòdul: un mòdul només te de cridar al inici 'un altra (amb CALL) i aquest te de retornar al final amb un únic RET, no hi tindria d'haver més punts de sortida i no es recomanable alterar l'adreça de retorn.

Excepte en els punts en que la velocitat o la memòria son crítics (l'experiència demostra que son menys del 1%) te de codificar-se el programa amb claredat, si es precís perdent eficiència. Aquest 1% cal documentar-lo convenientment com es faria per que ho llegeixi una altra persona.

Els mòduls han de ser «caixes negres» i no deuen modificar l'entorn exterior. Això significa que no tenen d'actuar sobre variables globals ni modificar els registres (excepte aquells registres i variables en que retornen els resultats, el que es te de documentar clarament al principi del mòdul). Tampoc han de dependre d'execucions anteriors, excepte excepcions a les que la pròpia claredat del programa obligui a lo contrari (per exemple, els generadores de números aleatoris poden dependre de la crida anterior).

Per el pas de paràmetres entre mòduls existeixen varis mètodes que s'exposen a continuació. Els paràmetres es poden passar també de dos maneres: directament per valor, o be indirectament per referència o adreça. En el primer cas s'envia el valor del paràmetre i en el segon l'adreça inicial de memòria a partir de la que esta emmagatzemat. El tipus dels paràmetres haurà d'estar degudament documentat al principi dels mòduls.

Pas de paràmetres en els registres: Els mòduls utilitzen certs registres molt concrets per comunicar-se. Tots els demés registres han de quedar sense cap canvi, per tant, si son emprats internament, han de ser preservats al principi del mòdul i restaurats al final. Aquest es el mètode emprat per el DOS i la BIOS en la majoria de les ocasions per comunicar-se amb qui els crida. Els registres seran preservats preferiblement a la pila (amb PUSH) i recuperats (amb POP en ordre invers); d'aquesta manera, els mòduls son reentrants i poden ser cridats de manera múltiple suportant, entre altres característiques, la recursivitat (Però, es requerirá també que les variables locals es generin sobre la pila).

Pas de paràmetres a través d'un àrea comú: s'utilitza una zona de memòria per la comunicació. Aquest tipus de mòduls no son reentrants i fins que no acabin de processar una crida no se les te de cridar novament al mig de la feina.

Pas de paràmetres per la pila. En aquest mètode, els paràmetres son apilats abans de cridar al modulo que els recollirà. Aquest ha de conèixer el nombre i mida dels mateixos, per equilibrar l'apuntador de la pila al final abans de retornar (mètode dels compiladors de llenguatge Pascal) o en cas contrari el programa que crida s'haurà d'encarregar d'aquesta operació (llenguatge C). La avantatja del pas de paràmetres per la pila es el pràcticament il·limitat el nombre de paràmetres admès, de còmode accés, y que els mòduls segueixen signeu reentrants. Un exemple pot ser el següent:

```
dadaL       DW      ?
dadaH       DW      ?
            ...
            PUSH    dadaL       ; apilar paràmetres
            PUSH    dadaH
            CALL    modulA      ; crida
            ADD     SP,4        ; equilibrar pila
            ...
```

```
modulA      PROC    NEAR
            PUSH    BP
            MOV     BP,SP
            MOV     DX,[BP+4]   ; part alta de la dada
            MOV     AX,[BP+6]   ; part baixa de la dada
            ...
            POP     BP
            RET
modulA      ENDP
```

En el exemple, tenim la variable dada de 32 bits dividida en dos parts de 16. Aquesta variable es col·locada a la pila començant per la part menys significativa. A continuació es crida a MODULA, el que comença per preservar BP (l'utilitzarà posteriorment) per respectar la norma de caixa negra. Es carrega BP amb SP degut a que el 8086 no permet l'adreçament indexat sobre SP. Com que la instrucció CALL es dirigeix a una adreça pròxima (NEAR), a la pila s'emmagatzema només el registre IP. Per tant, a [BP+0] hi ha el BP del programa que crida, a [BP+2] el registro IP del programa que crida i a [BP+4] i [BP+6] la variable enviada, que es el cas més complex (variables de 32 bits). Aquesta variable es carregada a DX:AX abans de procedir a utilitzar-la (també es tindrien d'apilar AX i DX per conservar l'estructura de caixa negra). Al final, es retorna amb RET i el programa principal equilibra la pila augmentant SP en 4 unitats per compensar l'apilament previ de dos paraules abans de cridar. Si MODULA fos un procediment llunya (FAR) la variable estaria a [BP+6] i [BP+8], degut a que al cridar al mòdul s'hauria guardat també a la pila el CS del programa que crida. El llenguatge Pascal ves retornat amb RET 4, fent innecessari que el programa que crida equilibri la pila. Però, el mètode del llenguatge C exposat es més eficient per que no requereix que el mòdul cridat conegui el nombre de paràmetres que se l'hi envien: Aquest pot ser variable (de fet, el C apila els paràmetres abans de cridar en ordre invers, Comencem per l'últim: d'aquesta manera s'accedeix correctament als primers N paràmetres que es necessiten).