Programación en ensamblador x86-16bits
Índice

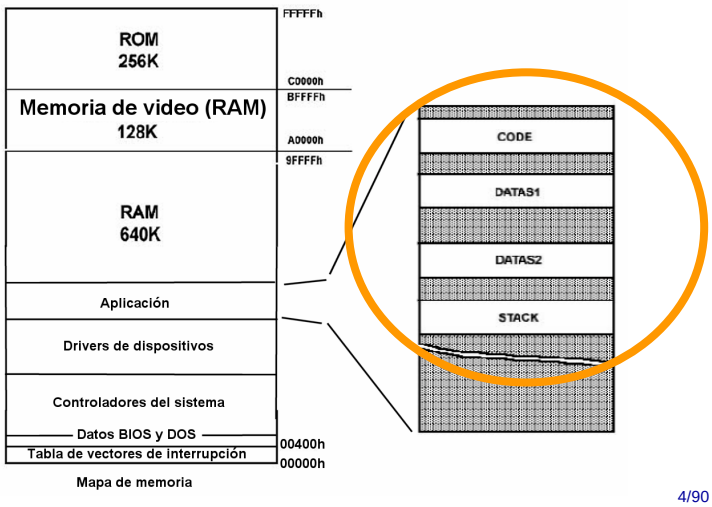
Programación en ensamblador x86-16bits
1. Organización de memoria de un programa

1. Organización de memoria de un programa
Índice

Programación en ensamblador x86-16bits
1. Organización de memoria de un programa

- Un programa NO es sólo un conjunto de instrucciones
- Un programa organiza su mapa de memoria en secciones en aras de la eficiencia
  - Código
  - Datos
  - Pila
  - …
- El programa compilado se dispone en una imagen guardada como un fichero ejecutable

Programación en ensamblador x86-16bits
1.1. Secciones de un programa



- Las secciones de un programa son:
  - Código o texto
  - Datos
  - Datos no inicializados (*bss*)
  - *Heap*
  - Pila

- Sección de código o texto
  - Contiene la secuencia de instrucciones a ejecutar
  - Suele ser de sólo lectura aunque algunos programas se automodifican o reservan pequeños espacios para datos en esta sección
  - Requiere de un puntero especial (puntero de instrucción o contador de programa) que señala la posición de la siguiente instrucción a ejecutar
    - Característica típica del modelo de Von Neumann

- **Sección de datos**
  - Contiene las variables globales (accesibles desde todo el programa) y las estáticas (su tiempo de vida abarca todo el tiempo de ejecución[1]) que contienen datos inicializados por el programador
  - Es una sección de lectura y escritura

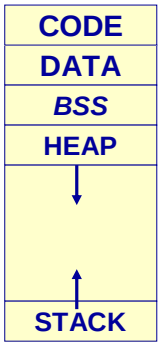
- **Sección *bss* (*Block Started by Symbol*)**
  - Contiene variables estáticas no inicializadas, es decir, cuyo valor inicial es 0
  - Normalmente esta sección no se almacena en el fichero imagen del ejecutable sino que es el cargador del sistema operativo (*dispacher*) quien realiza la reserva de espacio en memoria principal y el relleno con 0

[1] Una variable estática de un procedimiento, no se destruye al terminar el procedimiento pero sólo es accesible desde el procedimiento

- **Sección de *heap***
  - Se usa para hacer reservas dinámicas de memoria en tiempo de ejecución
  - Las reservas de memoria pueden liberarse o incrementarse en ejecución
  - No aparece en el fichero imagen del ejecutable

- **Sección de pila**
  - **Implementa un área de memoria con un tipo de acceso LIFO (*Last Input First Output*)**
    - Se utiliza en el manejo de procedimientos (subrutinas) para almacenar temporalmente los argumentos y variables locales
    - También se usa para salvaguardar datos de los registros
  - **Requiere de un puntero especial (puntero de pila) que indica la posición de memoria de la cima de la pila**
  - **Se suele utilizar otro puntero especial (puntero de marco de pila) que sirve para referenciar los argumentos y variables locales propios del procedimiento (subrutina) en curso**

- **CODE: código del programa**
- **DATA: datos globales y estáticos**
- **BSS: datos no inicializados**
- **HEAP: variables ubicadas ('alocadas') dinámicamente**
- **STACK: datos locales y argumentos de funciones**



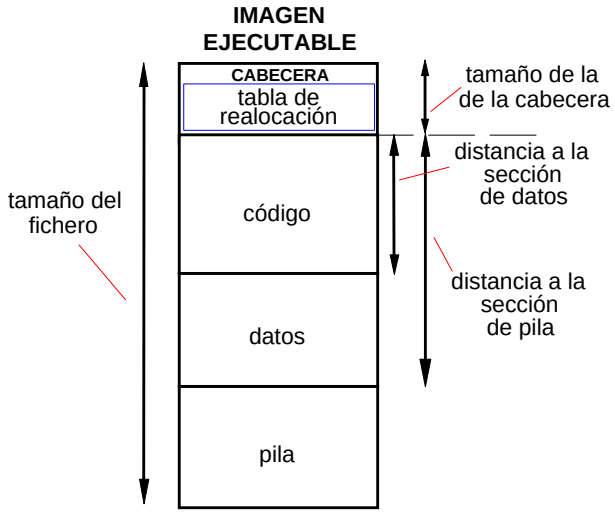

```
int g = 1;              /* g se sitúa en DATA */
int i;                  /* i se sitúa en BSS */

int main() {
int a;                  /* a se sitúa en STACK */
...
int *b=malloc(...); /* b se sitúa en HEAP */
...
a = ...;                /* código en CODE */
}
```

- **Cuando se compila el programa se crea un fichero con 2 partes:**
  - **Binario**
    - Imagen de las secciones del programa
    - No lleva ni la sección *bss* ni la sección *heap*
      - De ambas se encarga el sistema operativo
  - **Cabecera (información para el sistema operativo)**
    - Tamaño de la cabecera
    - Tamaño del fichero binario
    - Tabla de direcciones de memoria absolutas
    - Máxima y mínima cantidades de memoria requeridas
    - Valores iniciales de los punteros de instrucción y de pila

- **Tabla de direcciones de memoria absolutas (I)**
  - **"*Tabla de realocación*" en x86-16bits**
  - **La imagen de un ejecutable es una mapa de memoria reubicable**
    - Ha de funcionar igual sea cual sea el rango de memoria asignado
  - **Las referencias absolutas son desconocidas**
    - Por ejemplo, el comienzo de la sección de datos es desconocido hasta que el sistema operativo no asigne un mapa de memoria al proceso

- **Tabla de direcciones de memoria absolutas (II)**
  - **Para que el mapa de memoria de la imagen ejecutable sea reubicable, se confecciona una tabla con todas las posiciones de memoria absolutas que han de corregirse**
    - La tabla se guarda en la cabecera del ejecutable
  - **En las posiciones absolutas de la imagen se escribe su distancia hasta el comienzo de la imagen**
  - **Una vez que el sistema operativo conoce el rango del mapa de memoria asignado al proceso, modifica todos los valores de las direcciones absolutas sumando la posición de memoria inicial del proceso con la distancia que aparece en la imagen**

- **Tabla de direcciones de memoria absolutas (III)**



Programación en ensamblador x86-16bits
1.1.1. Cabecera y binario

- **Formatos de cabeceras**
  - **a.out (*assembler output*): originario de UNIX evolucionó a COFF**
  - **COFF (*Common Object File Format*): reemplazado en UNIX por ELF sirvió de base para formatos de Windows**
  - **ELF (*Executable & Linkable Format*): formato para ejecutables, código objeto, librerías compartidas y volcado de memoria en entorno UNIX; admite una gran variedad de secciones en los ejecutables**
  - **MZ (Mark Zbikowski): utilizado en el entorno DOS evolucionó dando lugar a varias extensiones**
  - **PE (*Portable Executable*): formato para ejecutables, código objeto, librerías compartidas (DLL), archivos de fuentes y otros usos en Windows; evolución de COFF**

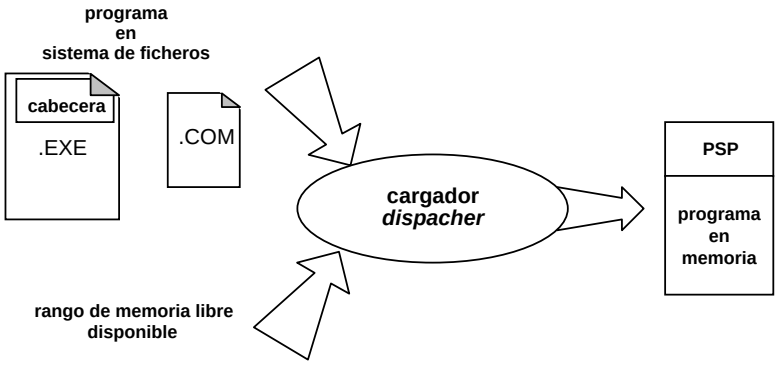
Programación en ensamblador x86-16bits
1.1.2. Ejecutables .COM

- **¡Un caso especial!**
  - **Los ficheros ejecutables .COM no tienen cabecera**
    - No requieren de tabla de realocación
  - **Contienen solamente el binario de las secciones de código y datos**
    - Ocupan un máximo de 64KB – 256B, es decir, un segmento de 64KB menos un bloque de 100hB reservado para el sistema (estado del proceso)
    - El código comienza en el desplazamiento 100h (256B)
    - La pila siempre comienza al final del segmento de 64KB
    - Los datos suelen estar dentro de la sección de código
    - Son interesantes por su simplicidad y tamaño reducido

- **Un proceso es un programa en ejecución**
  - **Una secuencia de código con una serie de recursos asociados y un estado**
    - **Recursos**: contador de programa, datos de memoria, pila y su puntero, registros y operadores de la ruta de datos y E/S (puertos, descriptores de ficheros, etc.)
    - **Estado**: información del punto de ejecución, situación de procesamiento, propietario, privilegios, comunicaciones, mecanismo de devolución del control al sistema operativo, etc.
      - *Process descriptor*
      - *Process control block* (PCB)
      - *Program segment prefix* (PSP)

- **Creación de un proceso**
  - **El sistema operativo…**
    - toma la imagen ejecutable y la copia en memoria,
    - actualiza sus direcciones absolutas e inicializa los datos que lo requieran,
    - asigna recursos, y
    - transfiere el control a la primera instrucción
  - **La transferencia de control a la primera instrucción del nuevo proceso se hace como si fuera un salto al código de una interrupción o procedimiento**
    - Por eso lo primero que se suele hacer es guardar la dirección de retorno (al proceso llamador) en la pila

- **En el entorno DOS:**



- **Estados de un proceso**
  - **Un proceso puede estar en…**
    - **Ejecución**
    - **Bloqueado** (a la espera de algún evento externo); o
    - **Listo** (esperando a disponer de los recursos de ejecución del procesador)

- **Finalización de un proceso**
  - **Un proceso puede terminar…**
    - de manera normal, devolviendo el control al sistema,
    - puede ser abortado por otro proceso, o
    - puede finalizar a consecuencia de una excepción
  - **Puesto que el proceso se "ve" como un procedimiento invocado por el proceso llamador, la salida se efectúa como un retorno (RET), leyendo de la pila la dirección de retorno**

- **Usamos el entorno de desarrollo *Microsoft*® Macro Assembler (MASM) 5.10**
  - **Trabaja en modo real de 16bits (DOS)**
  - **En Windows se ejecuta sobre DPMI (*DOS Protected Mode Interface* o Interfaz de Modo Protegido para DOS) que permite a un programa de DOS ejecutarse en modo protegido**
  - **Proporciona herramientas para ensamblar (`masm`), enlazar (`link`), depurar (`cv`), etc.**

- **Algunas limitaciones del entorno:**
  - **No admite nombres largos, sólo el formato 8.3 (8 caracteres para el nombre, punto (‘.’) y 3 caracteres para la extensión**
  - **No admite rutas con nombres largos ni rutas con espacios en blanco**

- **Dos modos de declarar las secciones de un programa:**
  - **Definición completa de secciones**
    - El programador tiene el control completo de la organización de las secciones de un programa
  - **Definición simplificada de secciones**
    - El programador deja que MASM organice las secciones
      - Este modo se invoca con la directiva **DOSSEG**

Programación en ensamblador x86-16bits

1.3.1. Definición completa de secciones

Segmentos separados
Salida mediante INT 20h

```
ASSUME CS:codigo, DS:datos, SS:pila
;---------- segmento de datos ----------
datos    SEGMENT
         // DECLARACIÓN DE DATOS //
datos    ENDS
;---------- segmento de pila ------------
pila     SEGMENT STACK
   DB tamaño DUP(?)      ;tamaño de la pila en bytes
pila     ENDS
;---------- segmento de código ----------
codigo   SEGMENT
   main  PROC FAR
         PUSH DS         ;dirección de retorno a la pila
         XOR AX, AX      ;apunta al PSP (DS:0)
         PUSH AX
         MOV AX, datos   ;inicializamos el segmento de datos
         MOV DS, AX
         // CÓDIGO DEL PROGRAMA //
         RET             ;devolvemos el control al DOS
   main  ENDP
codigo   ENDS
END main     ;indica dónde arranca el programa (CS:IP)
```


Programación en ensamblador x86-16bits

1.3.1. Definición completa de secciones

Segmentos separados
Salida mediante INT 21h

```
ASSUME CS:codigo, DS:datos, SS:pila
;---------- segmento de datos ----------
datos    SEGMENT
         // DECLARACIÓN DE DATOS //
datos    ENDS
;---------- segmento de pila ------------
pila     SEGMENT STACK
   DB tamaño DUP(?)      ;tamaño de la pila en bytes
pila     ENDS
;---------- segmento de código ----------
codigo   SEGMENT
inicio:  MOV AX, datos   ;inicializamos el segmento de datos
         MOV DS, AX

         // CÓDIGO DEL PROGRAMA //

         MOV AX, 4C00h   ;el servicio 4Ch devuelve el control
         INT 21h         ;al MS-DOS
codigo   ENDS
END inicio      ;indica dónde arranca el programa (CS:IP)
```

Programación en ensamblador x86-16bits

1.3.1. Definición completa de secciones

Segmentos solapados
Salida mediante INT 21h

```
comun  GROUP codigo, datos, pila
ASSUME CS:comun, DS:comun, SS:comun
;---------- segmento de datos -----------
datos    SEGMENT
         // DECLARACIÓN DE DATOS //
datos    ENDS
;---------- segmento de pila ------------
pila     SEGMENT STACK
   DB tamaño DUP(?)      ;tamaño de la pila en bytes
pila     ENDS
;---------- segmento de código ----------
codigo   SEGMENT
inicio:  MOV AX, comun  ;inicializamos el segmento de datos
         MOV DS, AX

         // CÓDIGO DEL PROGRAMA //

         MOV AX, 4C00h  ;el servicio 4Ch devuelve el control
         INT 21h        ;al MS-DOS
codigo   ENDS
END inicio      ;indica dónde arranca el programa (CS:IP)
```


Programación en ensamblador x86-16bits

1.3.1. Definición completa de secciones

Segmentos solapados
Sin segmento de datos

```
comun  GROUP codigo, pila
ASSUME CS:comun, DS:comun, SS:comun
;---------- segmento de pila ------------
pila     SEGMENT STACK
   DB tamaño DUP(?)      ;tamaño de la pila en bytes
pila     ENDS
;---------- segmento de código ----------
codigo   SEGMENT
inicio:  JMP ppio

         // DECLARACIÓN DE DATOS //

ppio:    MOV AX, comun  ;inicializamos el segmento de datos
         MOV DS, AX

         // CÓDIGO DEL PROGRAMA //

         MOV AX, 4C00h  ;el servicio 4Ch devuelve el control
         INT 21h        ;al MS-DOS
codigo   ENDS
END inicio      ;indica dónde arranca el programa (CS:IP)
```

- **Definición de segmentos DOSSEG**
  - **Sólo es posible a partir de la versión 5.0 de MASM**
  - **Es necesario declarar un modelo de memoria**
    - Especifica el tamaño de datos y código
    - Directiva .MODEL {modelo}
  - **El uso de DOSSEG permite ordenar los segmentos de forma consistente (según los criterios de *Microsoft* ®) independientemente del orden utilizado en el código**
    - El orden es CODE, DATA, BSS y STACK
  - **Los segmentos se definen utilizando directivas propias**
    - La directiva de segmento indica el comienzo del mismo y el final del anterior


```
DOSSEG
.MODEL SMALL
.STACK 100h
.DATA
   // DECLARACIÓN DE DATOS //

.CODE
inicio:
   MOV AX, @DATA
   MOV DS, AX

   // CÓDIGO DEL PROGRAMA //

   MOV AH, 4Ch
   INT 21h
END inicio
```

- **Modelos de memoria DOSSEG:**
  - **TINY [1]: código y datos usan el mismo segmento**
  - **SMALL: código y datos usan un segmento cada uno**
  - **MEDIUM: datos usa un segmento; código usa más de uno**
  - **COMPACT: código usa un segmento; datos más de uno (pero los *arrays* no pueden exceder 64K)**
  - **LARGE: código y datos más de un segmento (*arrays* < 64K)**
  - **HUGE: código y datos usan más de un segmento (los *arrays* pueden exceder los 64K)**

[1] Sólo a partir de MASM6.0; está pensado para generar ejecutables .COM

- **Segmentos básicos DOSSEG:**
  - **.STACK [tamaño]: segmento de pila (por defecto 1K)**
  - **.CODE [nombre]: segmento de código**
  - **.DATA: segmento de datos tipo *near* inicializados**
  - **.DATA?: segmento de datos tipo *near* no inicializados**
  - **.FARDATA [nombre]: segmento de datos tipo *far* inicializados**
  - **.FARDATA? [nombre]: segmento de datos tipo *far* no inicializados**
  - **.CONST: segmento para constantes**
    - .DATA? Y .FARDATA? → sección BSS

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- Cuando arranca un proceso, los valores de los registros de segmento (CS, DS, SS, ES) y de los registros IP y SP, deben tener valores correctos
- Los programas en código ensamblador deben suministrar la información necesaria para cargar los valores adecuados antes de que se ejecuten instrucciones que acceden a dichos registros
- La inicialización de cada segmento se realiza de forma diferente
  - En unos casos, lo hace el sistema operativo
  - En otros, se hace por código

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- En el entorno MASM, existe una constante por cada segmento definido en el programa
- La constante tiene el mismo nombre que el segmento pero con el caracter '@' delante
  - @STACK, @DATA, @FARDATA ...
- La constante indica la posición de memoria del comienzo del segmento al que representa en la imagen del ejecutable

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- **Inicialización de CS e IP (I)**
  - **Los registros CS e IP los inicializa el sistema operativo con la información que obtiene del programa**
  - **El programa indica cual es la instrucción de comienzo (la instrucción a la que se le pasará el control al arrancar el proceso) con la directiva END**
    - END [dirección de inicio]
  - **Sólo una directiva END debe tener dirección de inicio**
    - No hay ninguna transferencia explícita a CS en código (MOV CS, … ← ¡prohibido!)

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- **Inicialización de CS e IP (II)**
  - **Los valores de CS e IP se anotan en la cabecera del ejecutable [1]**
  - **El valor de CS es relativo al comienzo de la imagen binaria**
  - **Al cargar el ejecutable en memoria, se actualiza el valor de CS de acuerdo al mapa de memoria del proceso**
    - Es decir, se suma al valor de CS en la cabecera, el valor de la primera posición de memoria del proceso

[1] **La cabecera para programas DOS es del tipo MZ**

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- **Inicialización de DS (I)**
  - **El registro de segmento DS se carga mediante código**
  - **Debe contener la dirección del segmento que se usará para datos**
  - **Se debe realizar en 2 pasos, porque un registro de segmento no se puede cargar directamente con un dato inmediato**
    - La inicialización suele aparecer al comienzo de la sección de código:

```
MOV AX, @DATA
MOV DS, AX
```


Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- **Inicialización de DS (II)**
  - **@DATA es una constante predefinida que contiene un puntero al segmento de datos en la imagen del ejecutable**
  - **Cuando el programa se carga en memoria, el sistema operativo suma al valor @DATA el de la primera posición de memoria del proceso**
    - @DATA es un elemento típico de la tabla de realocación

- **Inicialización de DS (III)**
  - **También podemos usar un nombre asignado con GROUP**

```
comun  GROUP datos, extradatos
ASSUME DS:comun, ES:comun
/ / / / / / /
MOV AX, comun
MOV DS, AX
MOV ES, AX
```

  - **GROUP declara un grupo de segmentos que comparten la misma dirección base, es decir, están solapados**
    - Los mismos datos tienen "vistas" diferentes

- **Inicialización de DS (IV)**
  - **Antes de cargar DS por programa, el sistema hace que apunte al PSP [1]**
    - Es decir, la dirección del PSP es `DS:0` antes de que una instrucción MOV asigne valor a DS

[1] **El PSP (*Program Segment Prefix*) es la estructura de memoria que almacena el estado de los procesos bajo DOS**

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- **Inicialización de SS y SP (I)**
  - **Los registros SS y SP los inicializa el sistema operativo con la información que obtiene del programa**
  - **Los valores de SS e SP se anotan en la cabecera del ejecutable**
    - El registro de segmento SS tomará el valor del último segmento de tipo STACK (puede haber varios)
    - El registro SP se inicializa con el valor definido para el tamaño del segmento de pila

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- **Inicialización de SS y SP (II)**
  - **Al cargar el ejecutable en memoria, se actualiza el valor de SS de acuerdo al mapa de memoria del proceso**
    - Es decir, se suma al valor de SS en la cabecera, el valor de la primera posición de memoria del proceso
  - **Los registros SS y SP también se pueden inicializar por programa:**

```
MOV AX, @DATA
MOV SS, AX
MOV SP, @DATA+tamaño_pila
```

Programación en ensamblador x86-16bits
1.3.3. Inicialización de registros de segmento

- **Inicialización de ES**
  - **El registro de segmento ES se inicializa por código**

```
MOV AX, @FARDATA
MOV ES, AX
```

  - **…o solapando segmentos**

```
MOV AX, @DATA
MOV ES, AX
```

  - **Antes de cargar ES por programa, el sistema hace que apunte al PSP**

Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **Tipos de datos**
  - **Simples**
    - Enteros
    - BCD
    - Reales
    - Punteros
  - **Compuestos o agregados**
    - Cadenas de caracteres
    - Arrays
    - Estructuras
    - Registros (*records*)

- **Se declaran en el segmento de datos**

```
[nombre] directiva valor [,valor .....]
```

➡ **`nombre`** → **nombre simbolico asignado a la variable**

➡ **`nombre`** toma como valor la dirección de comienzo de la variable

➡ Si no se proporciona **`nombre`**, se crea la variable y se reserva memoria, pero a la dirección de comienzo no se le asocia un nombre simbólico

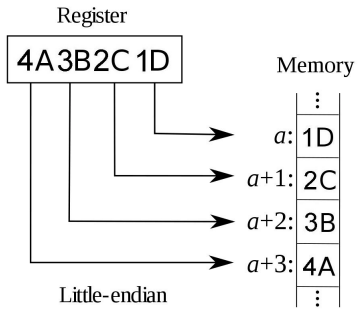
➡ **`directiva`** → **indica el tamaño de cada elemento**

nombre	directiva de declaración	argumento de directiva [1]	tamaño en bytes
byte	DB	BYTE	1
word	DW	WORD	2
doubleword	DD	DWORD	4
farword	DF	FWORD	6
quadword	DQ	QWORD	8
tenbyte	DT	TBYTE	10

[1] Algunas directivas admiten argumentos de tamaño

Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **El almacenamiento de los valores multibyte es del tipo *little endian***
  - **Es decir, el peso más bajo en la posición de memoria más baja y el peso más alto en la posición de memoria más alta**



Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **`valor` → dato con el que inicializar la memoria**
  - puede ser una constante
  - puede ser una expresión que da una constante al ser evaluada (por ejemplo, 3 * 4 + 15)
  - puede ser una interrogación ('?') indicando "no definido", es decir, no inicializado

Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **Pueden declararse varios valores separados por comas**
  - Se referencian con el nombre indexado por su orden comenzando en 0

```
nombre      DB 41,42,43,44,45
nombre[3] = 44
```

  - Los datos numéricos se pueden declarar con o sin signo
    - Si se declaran datos negativos, el programa ensamblador calcula su representación en C-2
  - Los reales se pueden declarar en notación científica

```
real_corto DD 91.67
real_largo DQ 6.452E-4
```


Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **Para declarar punteros, usaremos DW si son de tipo *near* y DD si son de tipo *far***
  - Ejemplo:

```
cadena          DB “hola mundo”,10,13,’$’
puntero_near    DW cadena
puntero_far     DD cadena
```

Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- Para declarar cadenas de caracteres se utiliza la directiva DB
  - Se inicializan declarando la cadena entre comillas dobles (“”)

```
cadena   DB “hola mundo”
```

  - También se pueden inicializar indicando cada carácter

```
cadena   DB ‘a’,’b’,’c’
cadena   DB 97,98,99
cadena   DB “hola”,10,13,’$’
cadena   DB “hola”,0
```


Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **Array**: agregado con múltiples datos del mismo tipo

```
[nombre] directiva contador DUP(valor[,valor…])
```

  - `contador` → indica el nº de veces que se repiten los valores
  - Se pueden anidar hasta 17 DUP en la misma definición

```
array      DD 8 DUP(1)
mascaras   DB 3 DUP(040h,020h,04h,02h)
anidados   DB 3 DUP(3 DUP(4 DUP(7)))
buffer     DB 5 DUP(?)
cadena     DB 4 DUP("hola")
```

DUP → *duplicate*

- **Estructura: colección de datos de igual o diferente tipo que pueden ser accedidos simbólicamente como uno solo**

  - **Dos pasos:**
    - Definir la estructura → permite dar valores iniciales

```
nombre STRUC
        campos
nombre ENDS
```

    - Definir las variables → permite modificar los valores iniciales

```
[nombre] nombre_struct <[valores]>
```


- **Ejemplo:**

```
socio STRUC
     id         DW ?
     nombre     DB "Nombre          "
     apellidos  DB "Apellidos       "
     edad       DB 3 DUP(0)
socio ENDS

S1 socio <>
S2 socio <168,"Ana","López Gil"," 22">
S3 socio 5 DUP (<>)
```

- **Registro (*record*): variable de tipo byte o palabra (*word*) que permite acceder de forma simbólica a conjuntos de bits**

  - **Dos pasos**
    - Definir el *record* → permite dar valores iniciales

```
nombre RECORD campo[,campo…]
campo: nombre_campo:ancho[=valor]
```

    - Definir las variables → permite modificar los valores iniciales

```
[nombre] nombre_record <[valores]>
```


- **Ejemplos:**

```
atributo RECORD
    parpadeo:1,fondo:3,brillo:1,caracter:3


mi_estilo atributo 50 DUP (<1,2,0,4>)
```

Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **Etiqueta: permite definir una variable de un tipo (tamaño) determinado en la dirección actual**

```
nombre LABEL tipo
tipo: BYTE,WORD,DWORD,FWORD,QWORD o TBYTE

warray  LABEL WORD
darray  LABEL DWORD
barray  DB    100 DUP(?)
```

  - Permite acceder al array 'barray' por bytes o por words ('warray') o por doublewords ('darray')

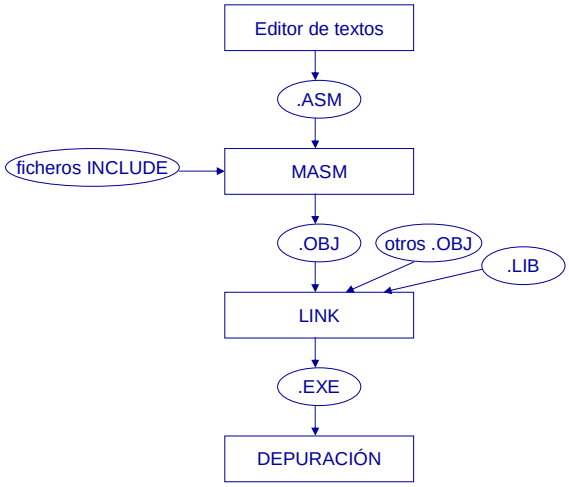
Programación en ensamblador x86-16bits
1.3.4. Declaración de datos

- **Para especificar un dato de memoria se usa el operador de índice '[]'**
- **Ejemplos:**
  - **TABLA[4]** → byte en offset 4 a partir de **TABLA**; es equivalente a **[TABLA + 4]**
  - **DS:[100h]** → se debe indicar el segmento si la etiqueta se omite con un índice constante; sin especificar registro no se puede conocer el segmento por defecto; es equivalenete a **DS:100H**
  - **[BX]** → el dato está en la posición de memoria **DS:BX**
  - **[BP+6]** → el dato está en **SS:BP+6**
  - **FOO[SI]+3** → el dato está en **DS:FOO+SI+3**
  - **BYTE PTR[BX][SI]** → se accede al byte en **DS:BX+SI**

2. Ciclo de desarrollo de un programa

Índice

- **Fases en el ciclo de desarrollo de un programa**



Programación en ensamblador x86-16bits
2. Ciclo de desarrollo de un programa

- **Entorno de desarrollo *Microsoft*® Macro Assembler (MASM) 5.10**
  - **Ensamblado: programa `masm`**
    - Creación de módulo objeto
  - **Enlazado: programa `link`**
    - Creación de ejecutable
  - **Depuración: programa `codeview`**

Programación en ensamblador x86-16bits
2.1. Creación de un módulo objeto

- **Programa `masm`**

```
masm  [opciones] fuente [,[objeto]
      [,[fichero_lista]
      [,[fichero_referencias]]]] [;]
```

  - **`fuente` → fichero con el código a ensamblar**
    - Extensión por defecto: `.asm`
  - **`objeto` → fichero de salida con el código objeto**
  - **`fichero_lista` → fichero con el listado de ensamblaje**
  - **`fichero_referencias` → fichero con las referencias cruzadas de los símbolos**
  - **';' → evita que nos proponga los nombres de los ficheros**

- Variables de entorno que afectan al programa `masm`
  - PATH → debe incluir el directorio donde están los ejecutables
    - `c:\masm51\bin`
  - INCLUDE → indica el directorio donde `masm` busca los ficheros incluidos
  - MASM → variable que especifica las opciones que el programa activa por defecto (`set MASM=/Zi`)
    - Las opciones usadas en la llamada al programa sobreescriben las opciones especificadas en la variable de entorno MASM

- Opciones del programa `masm`
  - `/D<sym>[=<val>]` → define símbolo (y valor)
    - Ensamblado condicional
  - `/I<path>` → ruta al directorio con los ficheros *include*
  - `/p` → comprueba la pureza del código
  - `/v` → muestra estadísticas
  - `/z` → muestra la línea de código fuente en los mensajes de error
  - `/Zi` → genera información simbólica para `codeview`
  - `/Zd` → genera información de número de línea
  - `/h` → muestra un listado de ayuda

- Programa `link`

```
link  [opciones] objeto[+objeto[+objeto]…]
      [,[ejecutable]
      [,[fichero_simbolos]
      [,[librerías]]]] [;]
```

  - `objeto` → fichero(s) de entrada con el código objeto
    - Extensión por defecto: `.obj`
  - `ejecutable` → fichero de salida
  - `fichero_símbolos` → mapa de símbolos
  - `librerías` → referencias externas
  - ';' → evita que nos proponga los nombres de los ficheros

- Variables de entorno que afectan al programa `link`
  - PATH → debe incluir el directorio de los ejecutables
    - `c:\masm51\bin`
  - LIB → indica el directorio donde `link` busca las librerías
  - TMP → indica el directorio donde `link` escribe los ficheros temporales
  - LINK → variable que especifica las opciones que el programa activa por defecto (`set LINK=/Co`)
    - Las opciones usadas en la llamada al programa sobreescriben las opciones especificadas en LINK

- Opciones del programa `link`
  - `/MAP[:numero]` → crea un fichero con el listado de símbolos globales y el mapa de memoria de secciones
  - `/STACK:numero` → especifica un tamaño máximo de pila; si la aplicación lo supera se emite un mensaje de error; en caso contrario, se podría disminuir la sección de pila
  - `/DOSSEG` → fuerza a organizar las secciones de acuerdo a las convenciones de *Microsoft*; es equivalente a la directiva DOSSEG en código
  - `/Co` → prepara el código para depuración con `codeview` incluyendo información simbólica

- Opciones del programa `link`
  - `/F` → optimiza los `CALL far` en tiempo y tamaño mediante la sustitución por:

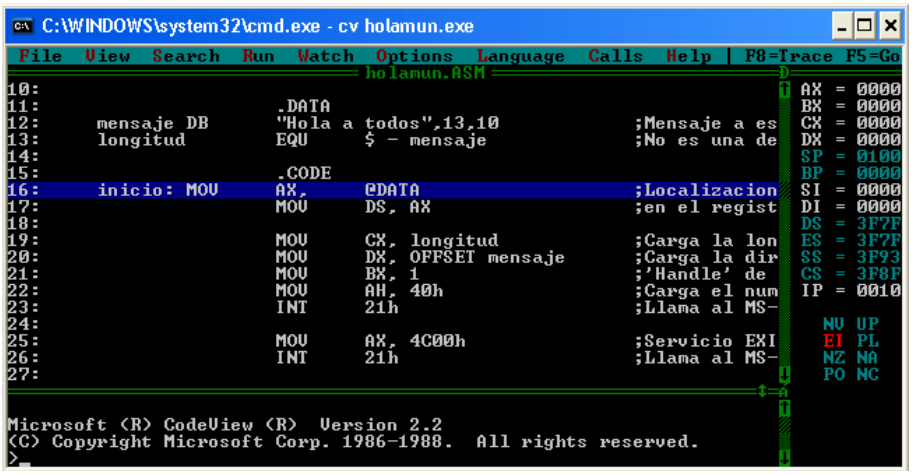
```
PUSH CS
CALL near
NOP
```

  - Sólo es efectivo en 286 y 386

- El proceso de depuración permite encontrar errores y hacer más eficiente el código
  - Los depuradores nos permiten
    - Ver el estado de la memoria y de los registros
    - Modificar el estado de la memoria y de los registros
    - Ejecutar el código paso a paso
- En el entorno DOS existen dos depuradores:
  - **debug**
  - **codeview**

Ver manuales de **debug** y **codeview** colgados en la página web de la asignatura

- Uso de **codeview**
  - Es necesario ensamblar y enlazar el código con las opciones de depuración (**/Zi** y **/Co** respectivamente)
- Comandos de ejecución de **codeview**
  - **T[n]** → (trace) ejecuta *n* líneas entrando en las funciones (tecla ràpida F8)
  - **P[n]** → (program step) ejecuta *n* líneas sin entrar en funciones (tecla ràpida F10)
  - **G[addr]** →(go) ejecuta hasta dirección *addr* o fin de programa (tecla ràpida F5)



- Comandos de inspección de `codeview`
  - `W[tipo]rango` →(watch) muestra las posiciones de memoria en el *rango* indicado
    - Podemos pasarle el nombre simbólico de una variable
  - `E[tipo]addr[list]` →(enter) modifica las posiciones de memoria
  - `D[tipo][rango]` → (*dump*) muestra las posiciones de memoria
  - `R[reg[expresión]]` → cambia el valor de un registro
  - `RF[flag]` → cambia el valor de un *flag*

3. Recursos de programación
Índice

- El programa ensamblador cuenta con operadores que se pueden usar en expresiones para inicializar datos o calcular inmediatos
  - Aritméticos: +, –, x, /, MOD, SHL, SHR
  - Lógicos: AND, OR, XOR, NOT
  - Relacionales: EQ, NE, LT, GT, LE, GE
  - Información: SEG, OFFSET, TYPE, SIZE, LENGTH, MASK
  - Atributos: PTR, $, HIGH, LOW

- Ejemplos:

```
mensaje   DB  "hola", 10, 13, '$'
pmensaje  DW  mensaje
pcrlf     DW  mensaje+4
longitud  EQU pmensaje-mensaje
seg_hora  EQU 60*60
mascara_h EQU 30h
mascara_l EQU mascara_h SHR 4
notmask_l EQU mascara_l XOR 0ffh
dato      EQU 50h

mov ax, dato EQ 50   ;mov ax, 0ffffh
mov ax, dato NE 50   ;mov ax, 0000h
```


- Operadores de información
  - SEG → devuelve el valor del segmento
    `mov ax, SEG tabla`
  - OFFSET →devuelve el valor del desplazamiento
    `mov ax, OFFSET tabla`
  - TYPE → devuelve el tipo de dato (1: byte, 2: word, 4: doble, 8: qword, 10: tenbyte, –1: near, –2: far)
  - SIZE → devuelve el tamaño de un DUP en bytes
  - LENGTH →devuelve el tamaño de un DUP en unidades

```
array DW 100 DUP(0)
mov ax, SIZE array    ;mov ax,200
mov ax, LENGTH array  ;mov ax,100
```

Programación en ensamblador x86-16bits
3.1. Operadores

- **Operadores de atributos**
  - **PTR → redefine el tipo**

```
array DW 100 DUP(0)
byte5 EQU byte ptr array+4
```

  - **$ → valor del contador de posiciones dentro del segmento**

```
mensaje   DB  "hola", 10, 13, '$'
longitud  EQU  $ - mensaje
```


Programación en ensamblador x86-16bits
3.2. Directivas

- **Las directivas son instrucciones para el programa ensamblador**
  - **ASSUME → indica el registro de segmento que se va a utilizar en cada sección**
  - **COMMENT → sirve para hacer comentarios largos**
    - Uso: COMMENT delimitador texto delimitador
    - Ejemplo:

```
COMMENT # Esto es un comentario de
varias líneas… hasta que vuelva a
aparecer el carácter delimitador #
```

  - **GROUP → agrupa varios segmentos en uno**

Programación en ensamblador x86-16bits
3.2. Directivas

- **IF*xx expresión*, ELSE, ENDIF → permiten el ensamblado condicional; *xx* puede ser E: igual a cero, DEF: definido, NDEF: no definido, nada: distinto de cero, etc.**
  - Ejemplo:

```
VALOR = 0
IF VALOR EQ 0
      MOV AX, 0F5h
ELSE
      MOV AX, 060h
ENDIF
```


Programación en ensamblador x86-16bits
3.2. Directivas

- **INCLUDE *fichero* → incluye un fichero con sentencias fuente**
  - Las sentencias fuente son aquellas que se repiten con mucha frecuencia o también macros
- **.RADIX *expresión* → cambia la base de numeración por defecto (los números sin sufijo se consideran por defecto en base 10); *expresión* será 2, 8, 10, 16 y los sufijos son: b, o/q, d y h**
- **SEGMENT → indica el comienzo de un segmento**
- **TITLE → nombre del módulo (se usará en los listados, podrá dar nombre el fichero .OBJ, etc.)**

Programación en ensamblador x86-16bits
3.3. Constantes predefinidas

- Existen un conjunto de constantes predefinidas, que se pueden utilizar en cualquier parte del código
  - @curseg → indica el nombre del segmento activo
  - @filename → contiene el nombre (sin extensión pero con el punto) del fichero fuente
  - @codesize → indica un valor según el modelo de código empleado (0: *small* y *compact*; 1: *medium*, *large* y *huge*)
  - @datasize → indica un valor según el modelo de código empleado (0: *small* y *medium*; 1: *compact* y *large*; 2: *huge*)

Programación en ensamblador x86-16bits
3.4. Macros

- En general, una macro (de macroinstrucción) es una instrucción compleja, formada por otras instrucciones más sencillas
  - Se ejecuta de manera secuencial, sin alterar el flujo de control
- En programación ensamblador, una macro es una secuencia de instrucciones a la que se identifica mediante un nombre
- Cada vez que aparece el nombre de la macro en el código, se sustituye por la secuencia de instrucciones (se dice que se expande)

- **Las macros son útiles ya que…**
  - **reducen la cantidad de codificación repetitiva**
  - **minimizan las oportunidades de provocar un error**
  - **simplifican la vista de los programas haciéndolos más legibles**

- **Declaración de macro**

```
nombre_macro MACRO [parámetro[,parámetro…]]
             //cuerpo de la macro//
             ENDM
```

  - **Consta de tres partes:**
    - Cabecera → contiene el nombre de la macro y, opcionalmente, variables ficticias o parámetros
    - Cuerpo → contiene el código real que será insertado expandiendo el nombre de la macro
    - Fin → directiva ENDM

- **Ejemplos:**

```
iniciarDS   MACRO
            mov ax,@DATA
            mov ds,ax
            ENDM

fincodigo   MACRO
            mov ax,4C00h
            int 21h
            ENDM

mostrarcadena     MACRO cadena
                  mov ah,09h
                  lea dx,cadena
                  int 21h
                  ENDM
```


- **Uso:**

```
            DOSSEG
            .MODEL SMALL
            //DECLARACIÓN DE MACROS//

            .STACK 100h
            .DATA
  mensaje   DB "Hola a todos",13,10,'$'

            .CODE
  inicio:   iniciarDS
            mostrarcadena mensaje
            fincodigo
            END
```

Las macros se declaran antes que cualquier segmento

- **A tener en cuenta!!! (I)**
  - **Si se pasan más parámetros de los necesarios, se ignoran**
  - **Si se pasan menos parámetros de los necesarios, se ignoran las instrucciones que los usan**
  - **La expansión de las macros hace crecer el tamaño del código**
    - Puede que los saltos condicionales no alcancen su destino

- **A tener en cuenta!!! (II)**
  - **El programador es responsable de salvar los registros que utilice la macro**
  - **Las etiquetas definidas dentro de una macro se expanden también duplicando los nombres de las mismas (p. ej. los destinos de los saltos)**
    - Si queremos que las etiquetas no se repitan debemos usar la directiva LOCAL
  - **Las macros suelen implementar mecanismos de ensamblado condicional (directiva IF)**

- Directiva LOCAL

```
LOCAL etiqueta[,etiqueta[,…]]
```

  - Indica el conjunto de etiquetas que debe cambiar de nombre cada vez que se expande la macro evitando las definiciones múltiples
  - Ejemplo:

```
retardo    MACRO contador
           local seguir
           mov cx,contador
seguir:    loop seguir
           ENDM
```


- Bibliotecas de macros
  - Las macros se pueden agrupar en bibliotecas y se pueden incluir en un programa
  - La biblioteca se crea reuniendo todas las macros que se desee en un archivo de texto
  - Para incluir la biblioteca en un programa basta con usar la directiva `INCLUDE nombre_biblioteca` antes de cualquier segmento
  - Ejemplo:

```
DOSSEG
.MODEL SMALL
INCLUDE macros.inc
```