Capítulo II

2. Conceptos básicos de Lenguaje Ensamblador 8086.

2.1 El ciclo de desarrollo de programas

Para crear un programa ejecutable utilizando lenguaje ensamblador es necesario realizar la serie de pasos, conocida como ciclo de desarrollo de programas, que se muestra en la figura 2.1.

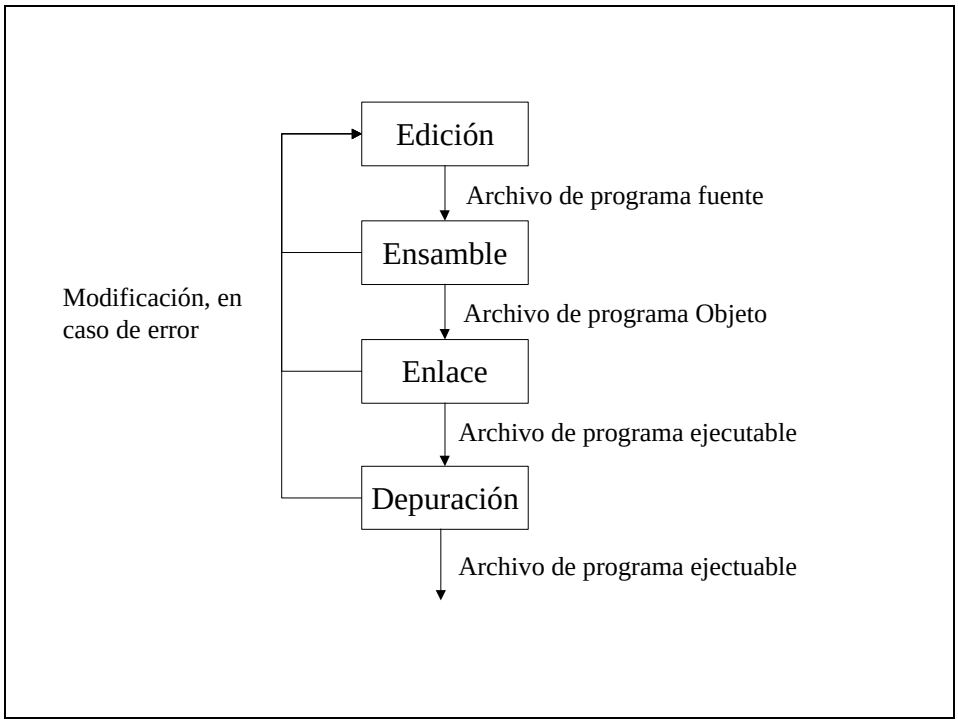

Figura 2.1Ciclo de desarrollo de un programa

Para cada programa que se realice, habrá eventualmente tres archivos en el disco. El paso inicial es crear un *archivo de programa fuente*. Para crear este programa es necesario usar un editor ASCII, como el programa EDIT de la versión 5.0 o posterior de MS-DOS. Este archivo contendrá todas las instrucciones de lenguaje ensamblador que conforman nuestro programa. Por convención, los archivos escritos en lenguaje ensamblador tienen asignados la extensión .ASM.

Los módulos de programa fuente que constituyen un programa ejecutable pueden estar organizados en una gran variedad de formas. Por ejemplo, pueden escribirse todos los procedimientos de un programa en un solo módulo o pueden dividirse estos procedimiento en diferentes módulos, de acuerdo a la especifidad de la tarea que realizan.

El programa en ensamblador, con extensión .ASM, es después usado para convertir el archivo de programa fuente en un *archivo objeto*, utilizando un traductor de lenguaje ensamblador, tal como el Macro Assembler, o MASM, de Microsoft o el Turbo Assembler, o TASM, de Borland.

Si durante el proceso de ensamblado del programa se encuentran errores en algún módulo de programa fuente, deberá regresarse al paso anterior para corregirlos antes de continuar. Para cada archivo fuente (con extensión .ASM) que se traduzca sin encontrar errores, el ensamblador creará un archivo objeto cuya extensión por omisión será .OBJ. También, durante el ensamblado de programas opcionalmente pueden generarse archivos de listado (con extensión .LST) y de referencias cruzadas (con extensión .CRF).

Para combinar todos los archivos objeto que integran un programa y formar un *archivo de programa ejecutable* (cuya extensión por omisión es .EXE), deberá utilizarse un *programa enlazador o ligador*. Durante este curso se hará uso del enlazador Turbo Link, o TLINK, de Borland, aunque también puede utilizarse el programa LINK de Microsoft. En este paso opcionalmente pueden crearse archivos con mapas del ejecutable producido, con extensión .MAP.

El archivo ejecutable debe ser depurado para descubrir errores lógicos. La depuración puede involucrar las siguientes técnicas:

- Ejecutar el programa para estudiar su entrada y su salida.
- Estudiar archivos fuente (.ASM) y de listado (.LST).
- Utilizar el programa CREF para crear un archivo de listado de referencias cruzadas (.REF)
- Utilizar un depurador, como Turbo Debugger de Borland Int'l, para seguir la ejecución del programa.

Si algún error lógico es encontrado durante la depuración, deberá retornarse al primer paso (edición) para corregir el código fuente.

Todo o parte de este ciclo de desarrollo de programas puede ser automatizado utilizando el programa MAKE para realizar archivos de descripción. MAKE generalmente es útil para desarrollar programas complejos que involucran numerosos módulos fuente. En caso de desarrollar programas de un solo archivo fuente, es más eficiente utilizar archivos de procesamiento de lotes del DOS.

2.1.1 Ensamblando y ejecutando un programa

Para facilitar el manejo del ciclo de desarrollo de programas, en esta sección se desarrolla un archivo de procesamiento por lotes que realiza las etapas de ensamblado del programa fuente, ligado de los módulos objetos y ejecución o depuración opcional del programa.

El programa batch del DOS ALE.BAT, mostrado en la figura 2.2, permite que el programador realice automáticamente las tareas del ciclo de desarrollo de programas, deteniéndose en cualquier etapa, en caso de que algún error ocurra.

```
@ECHO OFF
REM ALE.BAT
REM Ensambla, enlaza y ejecuta o depura el programa
REM pasado como argumento.
REM Sintaxis:
REM         ALE filename [x]
REM Donde:
REM   filename es el nombre del archivo fuente
REM   y x es cualquier carácter opcional, que evita
REM   la depuración del programa, para simplemente
REM   ejecutarlo
REM Autor: Ing. Jorge Luis Chuc López
IF NOT EXIST %1.ASM GOTO FIN
DEL %1.OBJ
TASM %1,%1;
IF NOT EXIST %1.OBJ GOTO FIN
TLINK %1,%1,NUL;
IF NOT EXIST %1.EXE GOTO FIN
IF NOT "A%2" == "A" GOTO EJECUTAR
TD %1
GOTO FIN:
:EJECUTAR
%1
:FIN
```

Figura 2.2. ALE.BAT, un archivo batch para ensamblar, ligar y ejecutar o depurar un archivo .ASM

Para la operación del programa se asume que se está utilizando el ensamblador y el ligador, TASM y TLINK respectivamente, de Borland. Además, no hace uso de las características avanzadas de estos productos y opera utilizando únicamente las opciones establecidas por

omisión. También, en el PATH del DOS deberán agregarse los directorios donde se encuentran instalados ambos productos de Borland.

La sintaxis del programa es:

```
ALE filename [x]
```

donde *filename* es el nombre del archivo a ensamblar, sin proporcionar su extensión.

x es un parámetro opcional que permite que después de ensamblar y ligar el programa, el ejecutable resultante se ejecute. Por omisión el ejecutable es sometido a depuración utilizando el Turbo Debugger de Borland Int´l,

La figura 2.3 muestra el programa HOLA.ASM que despliega el mensaje "Hola a todos." en la pantalla. Este programa será utilizado para mostrar el funcionamiento del archivo batch ALE.BAT.

```
;*********************************************
; Archivo: HOLA.ASM
; Autor: Ing. Jorge Luis Chuc López
; Fecha de creación: 25/08/93
; Ultima revisión: 30/08/93
; Propósito: Despliega un mensaje en la pantalla.
;*********************************************
      .MODEL  small
      .STACK      200h  ; 512 Kb para la pila
;
; --- Definición del segmento de datos ---
      .DATA
Msg   DB    'Hola a todos',0Ah, 0Dh, '$'
;
; --- Definición del segmento de código ---
      .CODE
Inicio:
      mov   ax,@data    ;Asigna a DS el registro de datos
      mov   ds,ax
      mov   dx,OFFSET Msg   ;Carga en DX el desplazamiento
                              ;mensaje número uno
      mov   ah,9        ;# Fn DOS. Despliega una cadena
      int   21h         ;Solicita servicio al DOS
      mov   ah,4Ch      ;# Fn DOS. Termina el programa
      int   21h         ;Solicita servicio al DOS
      END   Inicio      ;El programa inicia en la etiqueta
                        ;Inicio
```

Figura 2.3. Programa HOLA.ASM

Cuando se proporciona el comando

```
ALE HOLA
```

el archivo de procesamiento por lotes realiza el siguiente trabajo:

Verifica que el archivo que se desea ensamblar exista. De otra forma salta al final del archivo, sin hacer nada.

Si el archivo fuente existe, la tercera línea borra el archivo .OBJ del proceso de ensamblado que se haya realizado previamente. Si anteriormente no se hubiese realizado el ensamblado del programa fuente, esta línea desplegará un mensaje de error que no afectará el proceso en lote del archivo ALE.BAT. La línea

```
TASM %1,%1
```

ejecuta al programa TASM para realizar la traducción del programa fuente y producir el programa objeto. La línea siguiente verifica que el archivo .OBJ esté presente. Su ausencia indica un error durante el proceso de ensamblado, lo que ocasionará que se dé por terminado el procesamiento del ciclo de desarrollo de programas.

La línea

```
TLINK %1,%1,NUL
```

ejecuta el programa Turbo Link que enlazará los módulos para producir el programa ejecutable. Si hubiese algún error durante el ligado de los módulos existentes, la siguiente línea se encargará de verificarlo al probar la existencia del archivo con extensión .EXE.

La línea

```
IF NOT "A%1" == "A" GOTO EJECUTAR
```

verificará si el programador ha proporcionado un segundo parámetro en la línea de comando que ejecuta el archivo batch. Si lo anterior es cierto, se ejecutará el programa executable producido por TLINK; si no se proporciona segundo parámetro, se estará indicando que se desea depurar el archivo ejecutable y se invocará al programa Turbo Debugger para seguir la ejecución lógica del programa ejecutable.

2.2 El formato de una línea en lenguaje ensamblador 8086

Las líneas de código fuente de lenguaje ensamblador tienen el siguiente formato:

<etiqueta> <instrucción/directiva> <operandos> <;comentario>

donde <*etiqueta*> es un nombre simbólico opcional; <*instrucción/directiva*> es un nemónico para una instrucción o una directiva; <*operandos*> contiene una combinación de cero, uno o dos (y algunas pocas veces más) constantes, referencias de memoria, referencias

de registro y cadenas de texto, según lo requiera la instrucción particular o la directiva; <*;comentarios*> es un comentario opcional.

2.2.1 El campo de etiquetas

Las etiquetas son nombres usados para referenciar a números y cadenas de caracteres o localizaciones de memoria dentro de un programa. Las etiquetas permiten al programador asignarle un nombre a una variable de memoria, a valores y a las localizaciones de bloques de instrucciones particulares.

Las etiquetas pueden consistir de los siguientes caracteres:

Letras del alfabeto (A - Z, a - z)
Dígitos (0 - 9)
Carácter de subrayado (_)
Signo de pesos ($)
Signo de interrogación (?)

Los dígitos (0 - 9) no pueden ser utilizados como el primer carácter de una etiqueta. El signo de pesos ($) y el signo de interrogación (?) solos no pueden ser utilizados como un símbolo, puesto que así tienen un significado especial.

Cada etiqueta debe ser definida sólo una vez[1]; esto es, deben ser únicas. Las etiquetas pueden ser usadas como operandos cualquier número de veces.

Una etiqueta puede aparecer sola en una línea; es decir, sobre una línea sin una instrucción o directiva. En este caso, el valor de la etiqueta es la dirección de la instrucción o directiva en la siguiente línea del programa.

Las etiquetas que aparecen solas en una línea o acompañadas de instrucciones o directivas deben finalizar con un símbolo de dos puntos (:). Otras etiquetas generalmente no finalizan con este símbolo.

En el programa ejemplo HOLA.ASM se utilizan dos etiquetas: Msg, que es el nombre asignado a la cadena de caracteres ´Hola a todos´,0Ah,0Dh,´$´, almacenada en el segmento de datos del programa, e Inicio que, aunque aparezca sola en una línea del programa, representa la dirección donde se almacena en memoria la instrucción

```
mov   ax,@data
```

En la línea

```
mov   dx,OFFSET Msg
```

[1]Esta regla tiene su excepción en el procesamiento de macros del ensamblador, donde es posible definir más de una vez un identificador o etiqueta.

se está instruyendo al microprocesador para que almacene en el registro DX el desplazamiento que hay que recorrer, a partir del inicio del segmento de datos, para alcanzar la dirección donde inicia la cadena de caracteres representanda por la etiqueta Msg.

2.2.2 El campo de instrucciones y directivas

El campo clave en una línea de código ensamblador es el campo <*instrucción/directiva*>. Este campo puede contener un nemónico de instrucción o una directiva.

Los nemónicos de instrucciones son los nombres legibles para las instrucciones de lenguaje máquina que el 8086 ejecuta directamente. ADD, MOV y JMP son nemónicos de instrucciones, correspondientes directamente a las instrucciones de adición, movimiento y salto incondicional del 8086. El ensamblador se encargará de traducir estos nemónicos de instrucción a sus correspondientes instrucciones de código máquina.

En el programa HOLA.ASM, el bloque

```
mov   ax,@data
mov   ds,ax
mov   dx,OFFSET Msg
mov   ah,9
int   21h
mov   ah,4Ch
int   21h
```

representa el conjunto de instrucciones que despliega el mensaje en la pantalla.

Las directivas, por su parte, generalmente no producen código ejecutable, sino que controlan varios aspectos de como opera el ensamblador, del tipo de código ensamblado, de los segmentos usados, etc. En otras palabras, las directivas instruyen al ensamblador de ciertos aspectos relativos a la traducción de lenguaje ensamblador a lenguaje de máquina.

Las directivas utilizadas en el programa HOLA.ASM están escritas en mayúsculas, para distinguirlas de los mnemónicos de instrucción. Sin embargo, las directivas pueden ser escritas en minúsculas o en mayúsculas y minúsculas.

2.2.3 El campo de operandos

Los nemónicos de instrucciones y las directivas le dicen al ensamblador qué hacer. Los operandos, por otra parte, le dicen al ensamblador qué registros, parámetros, localizaciones de memoria, etcétera, están asociados a cada instancia de una instrucción o directiva. Una instrucción MOV no significa nada por sí misma; son necesarios los operandos para decirle al ensamblador de dónde tomará el valor a mover y dónde lo almacenará.

Se requieren uno, dos o más operandos para diferentes instrucciones, y virtualmente cualquier número de operandos que se ajusten en una línea pueden ser aceptados por varias directivas; el número correcto de operandos depende de la instrucción o directiva específicas. Los operandos pueden ser registros, valores constantes, etiquetas, variables de memoria y cadenas de texto.

Operandos registros

Los registros son quizá los operandos más frecuentemente usados por las instrucciones. Hay poco que pueda ser hecho con las constantes etiquetas o variables de memoria que no pueda ser realizado con los registros; por otra parte, algunas instrucciones únicamente pueden usar a los registros como operandos.

Para usar un registro como un operando, se especifica el nombre del registro como el operando de la instrucción.

- **Operandos constantes**

A menudo es necesario utilizar un valor constante como un operando.

Los caracteres pueden ser usados como operandos constantes puesto que un carácter tiene un valor bien definido. Los valores constantes pueden ser especificados en notación binaria, octal, hexadecimal o decimal. Algunas instrucciones no permiten utilizar a las constantes como sus operandos.

- **Expresiones**

Pueden utilizarse expresiones constantes en cualquier lugar donde se acepte un operador constante. Muchos ensambladores soportan la evaluación de expresiones completas, incluyendo paréntesis anidados, operadores aritméticos, relacionales y lógicos y una extensa variedad de operadores para propósitos tales como extraer los componentes de segmento y desplazamiento de etiquetas y la determinación del tamaño de las variables de memoria.

Es importante resaltar que las expresiones deberán evaluar a un valor constante.

- **Operandos de etiqueta**

Las etiquetas pueden servir como operandos de muchas instrucciones. Dados los operadores apropiados, las etiquetas pueden usarse para generar valores constantes. Las etiquetas también pueden ser usadas como destino de las instrucciones CALL (llamadas a procedimientos) y JMP (bifurcaciones incondicionales).

2.2.4 El campo de comentarios

Por último, pero no soslayable, tenemos al campo de <*;comentarios*>. Los comentarios no hacen nada, en el sentido de que no afectan el código del archivo ejecutable generado por el ensamblador, pero eso no significa que no sean importante.

El lenguaje ensamblador, a diferencia de los lenguajes de alto nivel estructurados, no cuenta con estructuras de control interconstruidas que hagan los programas virtualmente autodocumentables. Al contrario de esto, el código del lenguaje ensamblador está lejos de ser estructurado y fácil de mantener. Para facilitar esta característica, es necesario agregar comentarios a las líneas de código del lenguaje ensamblador.

Un enfoque útil de comentar un programa de lenguaje ensamblador es poner un comentario en el margen derecho de cada instrucción que agregue un poco de explicación a la acción que realiza. Este tipo de comentarios debe estar precedido de un símbolo de punto y coma (;).

2.3 Directivas de definición de segmento

Los segmentos son una parte fundamental de la programación en lenguaje ensamblador para la familia de procesadores 80x86. Están estrechamente relacionados con la arquitectura segmentada usada por los microprocesadores de 16 y 32 bits de Intel.

Un segmento es una colección de instrucciones o datos cuyas direcciones son relativas al mismo registro de segmento. El control de segmentos es uno de los aspectos más complejos del lenguaje ensamblador del 8086. Los segmentos pueden ser definidos usando uno de los dos conjuntos de directivas de control de segmentos proporcionados por Turbo Assembler. El primer conjunto, consistente de las *directivas de segmento simplificadas*, hace relativamente fácil el control de los segmentos y es ideal para enlazar módulos de ensamblador a lenguajes de alto nivel, pero soporta sólo algunas de las características relacionadas con los segmentos de que es capaz Turbo Assembler. El segundo conjunto, consistente de las *directivas de segmento estándar*, son más complicadas en su uso, pero proporciona el control de segmento completo requerido por las aplicaciones implementadas completamente en lenguaje ensamblador.

2.3.1 Directivas de segmento simplificadas

Las principales directivas de segmento simplificadas son .STACK, .CODE, .DATA, .MODEL y DOSSEG.

- **Las directivas .STACK, .CODE y .DATA**

.STACK, .CODE y .DATA definen los segmentos de pila, código y datos, respectivamente.

La directiva .STACK define el tamaño de la pila. Por ejemplo

```
.STACK 200h
```

define una pila de 200h (512) bytes de longitud. Eso es todo lo que tiene que hacerse en lo que se refiere a la pila; simplemente el programador debe asegurarse de que tiene una directiva .STACK en su programa y Turbo Assembler administrará la pila por él. Para los

programas normales, que no hacen un uso intensivo de la pila, 200h bytes es un buen tamaño de pila.

La directiva .CODE marca el inicio del segmento de código del programa. Esta directiva le dice a Turbo Assembler exactamente en qué segmento de código debe ejecutar las instrucciones. La definición del segmento de código es aun más simple de definir que el segmento de pila, utilizando las directivas de segmento simplificadas, puesto que no requiere de operandos. Por ejemplo,

```
...
.CODE
sub   ax,ax       ; Inicializa el acumulador a cero
mov   cx,100      ; # de ciclos a ejecutar
```

.DATA, por su parte, marca el inicio del segmento de datos. En este segmento deberán colocarse las variables de memoria. Por ejemplo:

```
            ...
            .DATA
ErrorMessage      DB 0Dh,0Ah,'*** Error ***',0Dh,0Ah,'$'
Counter           DW ?
            ...
```

A diferencia de las directivas de segmento anteriores, .DATA presenta cierta complejidad: en el código del programa, debe cargarse explícitamente el registro DS con el símbolo @data, antes de que se realicen accesos a localizaciones de memoria existentes en el segmento definido por .DATA. Puesto que un registro de segmento puede ser cargado con el contenido de un registro de propósito general o una localización de memoria, pero no con una constante, el registro de segmento es generalmente cargado con una secuencia de dos instrucciones:

```
...
mov   ax,@data
mov   ds,ax
...
```

Esta secuencia de instrucciones inicializa DS para que apunte al segmento de datos que inicia con la directiva .DATA.

Sin las dos instrucciones que inicializan el registro DS con el segmento definido por .DATA, las variables de memoria que residen en el segmento .DATA no podrán ser accesadas a menos que DS apunte a este segmento.

Los registros de segmento CS y SS son inicializados por el DOS cuando un programa inicia, por lo tanto no deberán ser cargados explícitamente como se hace con el registro DS.

Mientras que CS apunta a las instrucciones y SS apunta a la pila, DS apunta a los datos. Los programas no pueden manipular directamente el código y la pila; pero trabajan

constantemente con los datos de manera directa. Además, los programas pueden tener datos en diferentes segmentos a la vez; recuérdese que el 8086 permite accesar cualquier localidad de memoria en un rango de 1 Mb, pero únicamente en bloques de 64 Kb (relativo a un registro de segmento) a la vez. Por lo tanto, al principio el programador querrá inicializar el registro DS con un segmento, accesar datos en ese segmento y después cargar DS con otro segmento para accesar un bloque diferente de datos. Este esquema es utilizado generalmente en los programas más grandes de lenguaje ensamblador.

El registro ES, por su parte, es cargado tal y como se hace con el registro ES. A menudo, puede necesitarse trabajar con los dos segmentos de datos: el segmento de datos normal, apuntado por DS, y el segmento de datos extra, apuntado por ES.

- **La directiva DOSSEG**

La directiva DOSSEG causa que los segmentos en un programa de ensamblador sean agrupados acorde a las convenciones de ordenamiento de segmentos de Microsoft. Para la mayoría de las programas escritos completamente en lenguaje ensamblador el uso de esta directiva asegura que se desempeñen correctamente.

No es necesario especificar DOSSEG cuando se enlazan modulos de lenguaje ensamblador con lenguajes de alto nivel, debido a que los lenguajes de alto nivel automáticamente seleccionan el ordenamiento de segmentos de Microsoft; sin embargo, es útil su utilización para recordar que clase de ordenamiento de segmentos está en efecto.

Por lo tanto, el enfoque más simple es usar DOSSEG como la primera línea en todos los programas (a menos que se tengan razones para no hacerlo).

- **La directiva .MODEL**

.MODEL especifica el modelo de memoria para un módulo de lenguaje ensamblador que usa las directivas de segmento simplicadas.

Los modelos de memoria[2] disponibles se especifican en la tabla 2.1.

Se conoce como código NEAR a aquel código en el que las bifurcaciones (o saltos) se realizan cargando únicamente el registro IP, mientras se denomina código FAR a aquel código que es bifurcado cargando los registros CS e IP. Similarmente, los datos NEAR son accesados con sólo un desplazamiento, mientras que los datos FAR deben ser accesados con una dirección completa de *segmento:desplazamiento*. Brevemente, far significa que se utilizan direcciones completas de 32 bits, segmento:desplazamiento, mientras que near significa que se hace uso de desplazamientos de 16 bits.

[2]Los modelos de memoria descritos corresponden a los modelos de memoria utilizados por Turbo C y muchos otros compiladores para la PC y sistema operativo MS-DOS.

Pocos programas de ensamblador requieren más de 64 Kb de código o datos, así que el modelo SMALL es útil en casi todas las aplicaciones.

Modelo de Memoria	Descripción
tiny	El código y los datos del programa deben ajustarse dentro del mismo segmento de 64 Kb. Código y datos son near.
small	El código del programa debe ajustarse dentro de un segmento simple de 64 Kb, y los datos del programa deben estar en otro segmento separado de 64 Kb. Código y datos son near.
medium	El código del programa puede ser mayor que 64 Kb, pero los datos del programa deben ajustarse en un sólo segmento de 64 Kb. El código es far, mientras que los datos son near.
compact	El código del programa debe estar dentro de un segmento de 64 Kb, pero los datos del programa pueden ocupar más de 64 Kb. El código es near, mientras que los datos son far. Ningún arreglo de datos puede ser mayor de 64 Kb.
large	El código y los datos del programa pueden ocupar más de 64 Kb, pero ningún arreglo de datos puede ser mayor de 64 Kb. El código y los datos son far.
huge	El código y los datos del programa pueden ocupar más de 64 Kb y los arreglos de datos puede exceder los 64 Kb. El código y los datos son far. Los apuntadores a elementos dentro de un arreglo son far.

Tabla 2.1. Modelos de memoria

La directiva .MODEL es requerida si se utilizan las directivas de segmento simplificadas, puesto que de otra manera TASM no podría saber cómo inicializar los segmentos definidos con .CODE y .DATA. La directiva .MODEL debe preceder a .CODE, .DATA y .STACK.

- **La directiva END**

Todos y cada uno de los programas, independientemente que utilicen directivas de segmento simplificadas o estándar, deben contener una directiva END para marcar el final del código fuente del programa. Las líneas de código que sigan a la directiva END son ignoradas por Turbo Assembler. Si se omite la directiva END el ensamblador generará un error.

Además de terminar el programa fuente, END opcionalmente realiza una doble función al indicar dónde deberá iniciar la ejecución del programa. Muchas veces, por alguna razón, puede no desearse iniciar la ejecución del programa en la primera instrucción del archivo,

por lo que deberá hacer uso de la directiva END y proporcionar la etiqueta donde se desee que empiece a ejecutarse el programa.

En un programa consistente de únicamente un módulo (esto es, un archivo de código fuente), la directiva END deberá siempre especificar la dirección de inicio de ejecución del programa. En un programa consistente de más de un módulo, únicamente la directiva

END en el módulo conteniendo la instrucción en la que el programa empezará a ejecutarse deberá especificar la dirección de inicio; las directivas END en todos los otros módulos deberán aparecer como END, sin especificar dirección de inicio.

- **Modelo de programa utilizando las directivas de segmento simplificadas**

La figura 2.4 muestra el modelo de programa en lenguaje ensamblador, que hace uso de directivas de segmento simplificadas. Este modelo puede grabarse y ser utilizado cada vez que se inicie un nuevo programa

```
      .MODEL      small
;
; --- Definición del segmento de pila ---
      .STACK      200h
;
; --- Definición del segmento de datos ---
      .DATA
;
; --- Declaración de variables del programa ---
;
      ...
;
; --- Definición del segmento de código ---
      .CODE
Inicio:
      mov   ax,@data
      mov   ds,ax
      ...
; Código del programa haciendo uso de las variables
; declaradas en el segmento de datos.
      ...
      mov   ah,4Ch
      int   21h
      END   Inicio
```

Figura 2.4. Modelo de programa utilizando directivas de segmento simplificadas.

2.3.2 Directivas de segmento estándar

El modelo de programa que se lista en la figura 2.5 presenta un marco de trabajo que sustituye las directivas de segmento simplificadas por las directivas de segmento estándar.

Las directivas de segmento simplificadas hacen más fácil el trabajo de ligar módulos de lenguaje ensamblador con programas escritos en lenguajes de alto nivel. Las directivas de segmento estándar, por su parte, proporcionan un control de segmento más completo y son utilizadas para programas escritos totalmente en lenguaje ensamblador.

```
; --- Definición del segmento de pila ---
Pila        SEGMENT PARA STACK 'STACK'
            DB    200h DUP(?)
Pila        ENDS
;
; --- Definición del segmento de datos ---
Datos       SEGMENT WORD PUBLIC 'DATA'
;
; --- Declaración de variables del programa ---
Datos       ENDS
;
; --- Definición del segmento de código ---
Codigo      SEGMENT WORD PUBLIC 'CODE'
            ASSUME cs:_TEXT, ds:_DATA, ss:STACk
Inicio:
            mov   ax,Datos
            mov   ds,ax
            ...
; Código del programa haciendo uso de las variables
; declaradas en el segmento de datos.
            ...
            mov   ah,4Ch
            int   21h
Codigo      ENDS
            END   Inicio
```

Figura 2.5. Modelo de programa utilizando directivas de segmento estándar.

- **La directiva SEGMENT**

SEGMENT define el inicio de un segmento. La etiqueta precediendo a la directiva **SEGMENT** es el nombre del segmento; Por ejemplo, en el programa modelo anterior, la línea

```
Datos       SEGMENT WORD PUBLIC 'DATA'
```

define el inicio de un segmento llamado DATOS. La directiva SEGMENT puede especificar opcionalmente a un conjunto de atributos de segmento que le dan instrucciones al ligador y al ensamblador de cómo establecer y combinar los segmentos. Los atributos pueden ser especificados en cualquier orden y no es necesario proporcionarlos todos para un segmento.

- **El atributo de alineación**

El atributo opcional *align* define el rango de direcciones de memoria de la que puede seleccionarse una dirección de inicio para el segmento. Los valores que puede tomar un atributo se muestra en la tabla 2.2.

Tipo de alineación	Significado
BYTE	Usa la dirección del siguiente byte disponible
WORD	Usa la dirección de la siguiente palabra disponible (2 bytes por palabra)
DWORD	Usa la dirección de la siguiente doble palabra disponible (4 bytes por doble palabra); el tipo de alineación DWORD normalmente es utilizado en segmentos de 32 bites con procesadores 80386 o superiores
PARA	Usa la siguiente dirección de párrafo disponible (16 bytes por párrafo)
PAGE	Usa la siguiente dirección de página disponible (256 bytes por página)

Tabla 2.2. Atributos de alineación

Si no se proporciona el atributo de alineación, por omisión se utiliza el tipo PARA.

El ligador utiliza la información de alineación para determinar la dirección de inicio relativa para cada segmento. El DOS utiliza esta información para calcular la dirección de inicio actual cuando el programa es cargado en memoria.

- **El atributo de combinación**

El atributo opcional *combine* define cómo combinar segmentos que tienen el mismo nombre. El atributo de combinación puede tener cualquiera de los siguientes valores mostrados en la tabla 2.3.

Tipo de combinación	Significado
PUBLIC	Concatena todos los segmentos que tienen el mismo nombre para formar un segmento simple contiguo. Todas las direcciones de instrucciones y datos en el nuevo segmento son relativos a un registro de segmento simple y todos los desplazamientos son ajustados para que representen la distancia desde el inicio del segmento.

Tipo de combinación	Significado
STACK	Concatena todos los segmentos que tienen el mismo nombre para formar un segmento simple contiguo. Este tipo de combinación es similar al tipo de combinación PUBLIC, con la excepción de que todas las direcciones en el nuevo segmento son relativas al registro de segmento SS. El registro apuntador de la pila (SP) es inicializado con la longitud del segmento. El segmento de pila de tu programa deberá usar normalmente el tipo STACK, puesto que automáticamente inicializa el registro SS. Si se crea un segmento de pila y no se le aplica el atributo STACK, deberá proporcionarse las instrucciones para inicializar los registros SS y SP.
COMMON	Crea segmentos sobrepuestos colocando el inicio de todos los segmentos que tienen el mismo nombre en la misma dirección. La longitud del área resultante es la longitud del segmento más largo. Todas las direcciones en los segmentos son relativas a la misma dirección base. Si las variables son inicializadas en más de un segmento que tiene el mismo nombre y el atributo COMMON, el datos más recientemente inicializado reemplaza cualquier dato previamente inicializado.
MEMORY	Concatena todos los segmentos que tienen el mismo nombre a una forma de segmento contiguo simple. Algunos ligadores tratan a los segmentos MEMORY exactamente igual que los segmentos PUBLIC.
AT *address*	Causa que todas las direcciones de etiquetas y variables definidas en el segmento sean relativas a la dirección *address*

Tabla 2.3. Atributos de combinación de segmentos

Si ningún atributo de combinación es proporcionado, el segmento tendrá un atributo PRIVATE (Privado). Los segmentos que tienen el mismo nombre no son combinados. Cada segmento recibe su propio segmento físico cuando es cargado en memoria.

Aunque un nombre de segmento puede ser usado más de una vez en un archivo fuente, cada definición de segmento que usa el mismo nombre debe tener exactamente los mismos atributos o atributos quue no caen en conflictos. Si los atributos son dados para una

definición de segmento inicial, las definiciones subsecuentes para ese segmento no necesitan especificar los atributos.

Normalmente en un programa deberá proporcionarse al menos un segmento de pila (teniendo un atributo de combinación STACK). Si no se declara segmento de pila, el ligador generalmente desplegará un mensaje de precaución. Deberá ignorarse este mensaje si se desea una razón específica para no declarar un segmento de pila, como por ejemplo, cuando se está creando un programa con formato .COM.

- **El atributo de clase**

El atributo de clase es un medio de asociar segmentos que tienen diferentes nombres, pero propósitos similares. Puede ser usado para controlar el orden de los segmentos e identificar al segmento de código. El nombre de la clase debe estar encerrado en comillas simples (´).

Todos los segmentos pertenecen a una clase. Los segmentos a los que no se les especifica una clase explícitamente tienen el nombre de clase nulo. Los nombres asignados a los atributos de clase de los segmentos no deberán ser utilizados en otras definiciones de símbolos en el archivo fuente.

El ligador espera que los segmentos que tienen el nombre de clase CODE o un nombre de clase con el sufijo CODE contengan el código del programa. Deberá siempre asignarse este nombre de clase a los segmentos que contienen instrucciones del programa.

Los segmentos de datos por lo general tienen el nombre de clase DATA, mientras que el segmento de pila de un programa tendrá un atributo de clase STACK.

- **La directiva ENDS**

La directiva ENDS define el final de un segmento. Por ejemplo,

```
Datos       ENDS
```

finaliza el segmento cuyo nombre es DATOS, y que anteriormente debió ser iniciado con una directiva SEGMENT. Cuando se utilizan directivas de segmento estándar, debe finalizarse explícitamente todos los segmentos que se definan.

- **La directiva ASSUME**

ASSUME le dice a Turbo Assembler qué segmento inicializa un registro de segmento dado. Una directiva ASSUME CS: se requiere en todos los programas que usan las directivas de segmento estándar, debido a que TASM necesita conocer el segmento de código para inicializar un programa ejecutable. Además, usualmente también se necesitan directivas ASSUME DS: y ASSUME ES: para que Turbo Assembler sepa qué localizaciones de memoria pueden direccionarse en un momento dado.

ASSUME le dice a TASM que verifique que cada acceso a una variable de memoria con nombre sea válido, de acuerdo a los valores actuales de los registros de segmento. Por ejemplo, considere la siguiente sección de código:

```
Data1 SEGMENT WORD 'DATA'
Var1  DW    0
Data1 ENDS
      ...
Data2 SEGMENT WORD 'DATA'
Var2  DW    0
Data2 ENDS

Code  SEGMENT WORD 'CODE'
      ASSUME      cs:Code
ProgramStart:
      mov   ax,Data1
      mov   ds,ax ;Inicializa DS con Data1
      ASSUME ds:Data1
      mov   ax,[Var2]   ;Trata de cargar Var2 en AX --Esto
                        ;causará un error, puesto que Var2
                        ;no puede ser encontrado en el
                        ;segmento Data1
      ...
      mov   ah,4Ch
      int   21h
Code  ENDS
      END   ProgramStart
```

Durante el ensamblado de este programa, ocurrirá un error debido a que el código trata de accesar la variable de memoria VAR2 cuando el registro DS está puesto al segmento DATA1, y VAR2 no puede ser direccionado a menos que DS sea puesto al segmento DATA2.

Es importante entender que TASM no sabe en ese momento que DS ha sido puesto al segmento DATA1; en cambio, usando el enunciado ASSUME, se le dice a Turbo Assembler que haga esa suposición. ASSUME es la manera en que se le dice al ensamblador qué registros de segmento están inicializados en un momento dado, de tal forma que TASM puede hacerte saber cuándo se ha intentado algo imposible.

Sin embargo, todos los errores de este tipo no pueden ser atrapados por Turbo Assembler. Siempre que una referencia a memoria involucra una variable de memoria con nombre, tal como las variables VAR1 y VAR2 anteriores, TASM puede checar la validez de la referencia, puesto que cada variable de memoria con nombre está explícitamente asociada con un segmento. No hay forma de que el ensamblador pueda saber qué segmento pretende accesar una instrucción como esta:

```
      mov   al,[bx]
```

En tal caso, TASM debe suponer que el segmento al que DS está inicializado es el que se pretende accesar.

En algunos casos puede usarse un registro de segmento diferente que DS (ES, por lo general) para accesar memoria utilizando la directiva ASSUME. Por ejemplo, considérese esta versión corregida del programa anterior:

```
Data1 SEGMENT WORD 'DATA'
Var1  DW    0
Data1 ENDS
            ...
Data2 SEGMENT WORD 'DATA'
Var2  DW    0
Data2 ENDS

Code  SEGMENT WORD 'CODE'
      ASSUME      cs:Code

ProgramStart:
      mov   ax,Data1
      mov   ds,ax;      Inicializa DS con Data1
      ASSUME ds:Data1
      mov   ax,Data2
      mov   es,ax
      ASSUME es:Data2
      mov   ax,[Var2]   ;Trata de cargar Var2 en AX --Esto
                        ;causará un error, puesto que Var2
                        ;no puede ser encontrado en el
                        ;segmento Data1
      ...
      mov   ah,4Ch
      int   21h
Code  ENDS
      END   ProgramStart
```

Al ensamblar este programa TASM no reporta ningún error, pero no significa que esté permitiendo al programador hacer un error. Lo que hace el ensamblador es modificar la línea

```
      mov   ax,[Var2]
```

para accesar VAR2 relativo al registro de segmento ES en vez del registro de segmento **DS**.

Las dos directivas ASSUME han informado a TASM que DS está puesto al segmento DATA1 y que ES está usando el segmento DATA2. Entonces, cuando la instrucción MOV intenta accesar a VAR2, que está en el segmento DATA2, Turbo Assembler concluye correctamente que no hay manera de que VAR2 pueda ser accesada relativa a DS; sin embargo, VAR2 puede ser accesada relativa a ES. Consecuentemente, Turbo Assembler

inserta un código especial conocido como *Prefijo de Anulación de Segmento (Segment Override Prefix)* en la instrucción MOV para decirle al 8086 que use el registro ES en lugar del registro DS.

El programa en la figura 2.6 despliega el mensaje ´Hola a todos´, utilizando directivas de segmento estándar:

```
;************************************************
; Archivo: HOLA.ASM
; Autor: Ing. Jorge Luis Chuc López
; Fecha de creación: 25/08/93
; Ultima revisión: 30/08/93
; Propósito: Despliega el mensaje "Hola a Todos" en pantalla.
;************************************************
; --- Definición del segmento de pila ---
StckSeg     SEGMENT PARA STACK 'STACK'
      DB  512 DUP(' ')
StckSeg     ENDS
; --- Definición del segmento de datos ---
Data  SEGMENT PARA PUBLIC 'DATA'
Msg   DB    'Hola a todos',0Ah, 0Dh, '$'
Data  ENDS
;
; --- Definición del segmento de código ---
Code  SEGMENT PARA PUBLIC 'CODE'
      ASSUME CS:Code, DS:Data
Inicio:
      mov   ax,Data     ;Asigna a DS el registro de datos
      mov   ds,ax
      mov   dx,OFFSET Msg     ;Carga el mensaje en DX
      mov   ah,9        ;# Fn DOS. Despliega una cadena
      int   21h         ;Solicita servicio al DOS
      mov   ah,4Ch      ;# Fn DOS. Finaliza el programa
      int   21h         ;Solicita servicio al DOS
Code  ENDS
      END   Inicio
```

Figura 2.6. Programa HOLA.ASM que utiliza directivas de segmento estándar.

2.4 Declarando variables de memoria

2.4.1 Datos inicializados

Las directivas de definición de datos DB, DW, DD, DF, DP, DQ y DT permiten al programador definir variables de memoria de diferentes tamaños de datos.

Por ejemplo, la siguiente sección de código define cinco variables de memoria inicializadas e ilustra cómo algunas de estas variables pueden ser usadas.

Directiva	Define:
DB	1 byte
DW	2 bytes = una palabra
DD	4 bytes = una doble palabra
DF, DP	6 bytes = un apuntador de palabra far
DQ	8 bytes = un quadpalabra
DT	10 bytes

Tabla 2.4. Directivas de definición de datos

```
            ...
            .DATA
ByteVar     DB    'Z'   ;1 byte
WordVar     DW    101b  ;2 bytes (1 palabra)
DWordVar    DD    2BFh  ;4 bytes (1 doble palabra)
QWordVar    DQ    307o  ;8 bytes (1 quadword)
TWordVar    DT    100   ;10 bytes
      ...
      mov   ah,2        ;# Fn DOS de salida en pantalla
      mov   dl,[ByteVar]      ; Carácter a desplegar
      int   21h         ;Invoca al DOS para desplegar car.
      ...
      add   ax,[WordVar]
      ...
      add   WORD PTR [DWordVar],ax
      adc   WORD PTR [DWordVar+2],dx
```

- **Inicialización de arreglos**

Pueden aparecer múltiples valores en una sola directiva de definición de datos. Por ejemplo:

```
Arreglo     DW    0, 1, 2, 3, 4
```

crea un arreglo de cinco elementos (de una palabra de tamaño) cuyo nombre es ARREGLO. Cualquier número de valores que se ajusten en una línea puede ser usado con las directivas de definición de datos.

En caso de que un arreglo sea demasiado grande como para ajustarse en una sola línea, puede declararse en varias líneas, sin requerir definir etiquetas para cada una de las líneas

de definición de datos; solamente se define la etiqueta en la primera línea del arreglo. Por ejemplo, el siguiente código crea un arreglo de elementos de doble palabra llamado ARRCUADRADOS, consistente de los cuadrados de los primeros 15 números enteros:

```
            ...
ArrCuadrados      DD    0, 1, 4, 9, 16
                  DD    25, 36, 49, 64, 81, 100
                  DD    121, 144, 169, 196
            ...
```

Turbo Assembler permite definir bloques de memoria inicializados a un valor dado con el operador DUP. Por ejemplo,

```
BlankArray  DW    100h DUP(0)
```

crea un arreglo denominado BLANKARRAY, consistiendo de 256 palabras (decimales) inicializados a cero. De la misma manera, la siguiente línea crea un arreglo de 92 bytes, cada uno inicializado con el carácter A:

```
ArrayOfA    DB    92 DUP('A')
```

- **Inicializando cadenas de caracteres**

Los caracteres son operandos válidos para las directivas de definición de datos, así que puede definirse una cadena de caracteres utilizando los criterios establecidos anteriormente. Ejemplo

```
Cadena      DB    'A', 'B', 'C', 'D'
```

Las cadenas de caracteres se definen como arreglos con elementos del tamaño de un byte, puesto que cada carácter debe almacenarse en este tipo de datos.

Turbo Assembler permite definir una cadena de caracteres de una manera similar a como se realiza en los lenguajes de alto nivel, agrupando los caracteres dentro de comillas simples:

```
Cadena      DB    'ABCD'
```

Si se desea utilizar los caracteres de Retorno de Carro/Alimentación de Línea, deben listarse separadamente sus correspondientes valores. La siguiente línea define una cadena de texto seguida de un carácter de Retorno de Carro, un carácter de Alimentación de Línea y un byte cero, de terminación de cadena, tal y como se hace en el Lenguaje C.

```
HelloMsg    DB    'Hola, mundo', 0Dh, 0Ah, 0
```

- **Inicializando con expresiones y etiquetas**

El valor inicial de una variable inicializada debe ser una constante, pero no necesariamente tiene que ser un número. Expresiones como:

```
Var1  DB    65
Var2  DW    ((924/2)+1)
```

son correctas. También lo son las etiquetas:

```
      ...
      .DATA
Buffer      DW    16 DUP(0)
BufferPtr   DW    Buffer
      ...
```

Siempre que una etiqueta es usada como un operando en una directiva de definición de datos, es el valor de la etiqueta el que se usa, no el valor almacenado en esa etiqueta. En el ejemplo anterior, el valor inicial de BUFFERPTR es el desplazamiento de BUFFER en el segmento .DATA, no el valor cero que está almacenado en BUFFER.

2.4.2 Datos no inicializados

Algunas veces no tiene sentido asignar un valor inicial a una variable de memoria. Por ejemplo, supóngase que un programa lee los siguientes diez caracteres escritos en el teclado en un arreglo llamado KEYBUFFER:

```
      ...
      mov   cx,10                   ;# de cars. a leer
      mov   bx,OFFSET KeyBuffer     ;Los cars. a ser
                                    ;leídos
                        ; almacenados en KeyBuffer
GetKeyLoop:
      mov   ah,1        ;# Fn DOS para entrada del teclado
      int   21h         ;Obtiene el siguiente carácter
      mov   [bx],al     ;Guarda el carácter
      inc   bx          ;Apunta a la sig. localidad de
                        ;almacenamiento para sig. car.
      loop  GetKeyLoop
      ...
```

Podría definirse KEYBUFFER para ser inicializado con:

```
KeyBuffer   DB    10 DUP(0)
```

pero realmente no tiene sentido, puesto que los valores iniciales en KEYBUFFER son inmediatamente sobreescritos en el ciclo GETKEYLOOP. Lo que realmente se necesita es una manera de definir variables de memoria sin inicializar. Turbo Assembler proporciona esta capacidad con el signo de interrogación (**?**).

El signo de interrogación le dice al ensamblador que se está reservando una localidad de almacenamiento (memoria), pero no se está inicializando. Por ejemplo, la forma apropiada de definir KEYBUFFER sería:

```
KeyBuffer   DB    10 DUP(?)
```

Esta línea reserva 10 bytes iniciando en la etiqueta KEYBUFFER, pero no inicializa estos bytes con valor alguno.

Obviamente, el programador debe asegurarse de inicializar una variable de memoria no inicializada en la definición, antes de usarla en el programa.

2.4.3 Localizaciones de memoria con nombre

Como vimos anteriormente, con la directiva de definición de datos DB se asigna nombre a una variable de memoria. La directiva LABEL es otra manera de asignar un nombre de una localidad de memoria, pero sin localizar almacenamiento alguno.

LABEL permite especificar el nombre de una etiqueta y su tipo sin tener que definir dato alguno. Por ejemplo, el arreglo KEYBUFFER puede definirse también de la siguiente forma:

```
            ...
KeyBuffer   LABEL BYTE
            DB    10 DUP(?)
            ...
```

Una etiqueta definida con LABEL puede ser de los siguientes tipos:

```
BYTE        PWORD       FAR
WORD        QWORD       PROC
DWORD       TBYTE       UNKNOWN
FWORD       NEAR
```

BYTE, WORD, DWORD, FWORD, PWORD, QWORD Y TBYTE son autoexplícitos, etiquetando elementos de 1, 2, 4, 6, 8 y 10 bytes, respectivamente. A continuación se presenta un ejemplo, donde una variable de memoria es inicializada como un par de bytes, pero es accesada como una palabra:

```
            ...
            .DATA
WordPtr     LABEL WORD
            DB    1,2
            ...
            .CODE
            ...
            mov   ax,[WordPtr]
            ...
```

En este caso, AL es cargado con 1 (el primer byte de WORDPTR) y AH es cargado con 2.

Las etiquetas definidas mediante la directiva LABEL pueden utilizarse en expresiones de directivas de definición de datos. Por ejemplo, las siguientes líneas inicializan la variable WORDARRAYLENGTH con la longitud en bytes de WORDARRAY:

```
      ...
WordArray          DW    50 DUP(0)
WordArrayEnd       LABEL WORD
WordArrayLength    DW    (WordArrayEnd - WordArray)
      ...
```

Si se desea calcular la longitud de WORDARRAY en palabras de lugar de calcularlo en bytes, únicamente hay que dividir la longitud en bytes por dos;

```
WordArrayLengthInWords  DW    (WordArrayEnd - WordArray)/2
```

NEAR y FAR son usados en el código para seleccionar el tipo de llamada a procedimiento o bifurcación (salto) necesitado para alcanzar determinada etiqueta. Por ejemplo, en el siguiente ejemplo el primer JMP es un salto far (cargando CS e IP) debido a que es una etiqueta FAR, mientras que el segundo salto es un salto near (cargando únicamente IP) debido a que es una etiqueta NEAR:

```
            ...
            .CODE
            ...
FarLabel    LABEL FAR
NearLAbel   LABEL NEAR
            mov   ax,1
            ...
            jmp   FarLabel
            ...
            jmp   NearLabel
            ...
```

Cuando se usan las directivas de segmento simplificadas, PROC es una manera conveniente de definir una etiqueta en el tamaño apropiado, NEAR o FAR, para el modelo de código actual. Cuando el modelo de memoria es TINY, SMALL o COMPACT, LABEL PROC es similar a LABEL NEAR; cuando el modelo de memoria es MEDIUM, LARGE o HUGE, LABEL PROC es similar a LABEL FAR. Esto significa que si se cambia el modelo de memoria, se pueden cambiar ciertas etiquetas automáticamente también.

Por ejemplo, en

```
            .MODEL small
            ...
            .CODE
            ...
EntryPoint  LABEL PROC
            ...
```

ENTRYPOINT es NEAR, pero si se cambia el modelo de memoria a LARGE, ENTRYPOINT se convertirá a FAR. Normalmente el programador usa la directiva PROC en lugar de LABEL,

para definir la clase de puntos de entrada que se desee cambiar conforme cambie el modelo de memoria; sin embargo, algunas veces se necesitará más de un punto de entrada a una subrutina y entonces puede hacerse uso de LABEL, además de PROC.

Finalmente, LABEL UNKNOWN es simplemente una forma de decir que no se sabe qué tipo de datos tendrá la etiqueta que se define.

2.5 Modos de direccionamiento

Los operandos de las instrucciones pueden especificarse en diferentes formas conocidas como modos de direccionamiento. Los modos de direccionamiento le dicen al microprocesador cómo calcular el valor actual del operando de una instrucción en tiempo de ejecución.

Existen 3 modos de direccionamiento: inmediato, de registro y de memoria. Los operandos de memoria pueden ser divididos en dos grupos, operandos de memoria directos y operandos de memoria indirectos.

El microprocesador decodificará el modo de direccionamiento que está siendo referenciado por la sintaxis de la operación. Aunque dos enunciados pueden ser similares y sus mnemónicos de instrucción sean los mismos, el ensamblador producirá código máquina diferente para una instrucción cuando es usada con diferentes modos de direccionamiento.

Los mnemónicos de instrucción en lenguaje ensamblador pueden tener dos o más operandos que siempre son trabajados de derecha a izquierda. El operando de la derecha es el operando fuente. Especifica el dato que será utilizado, pero no cambiado, en la ejecución de la instrucción. El operando de la izquierda es el operando destino. Especifica el dato que será utilizado y posiblemente cambiado por la instrucción.

2.5.1 Direccionamiento inmediato

Consisten de datos numéricos constantes que son conocidos o calculados en tiempo de ensamble del programa. El dato a ser utilizado se especifica como un valor constante dentro de la instrucción misma y es procesado de la misma manera cada vez que el programa se ejecuta.

Algunas instrucciones tienen límites para el tamaño de valores inmediatos (usualmente 8, 16 y 32 bits). Las constantes de cadena de longitud mayor o igual a dos caracteres no pueden ser datos inmediatos. Deben ser almacenadas en memoria antes de que puedan ser procesadas por las instrucciones.

Muchas instrucciones permiten datos inmediatos en el operando fuente y datos en memoria o registros en el operando destino. La instrucción combina o reemplaza el dato en la dirección de memoria o el registro con el dato inmediato en la forma que define la instrucción. Ejemplos de este tipo de instrucción son MOV, ADD, CMP y XOR.

Unas pocas instrucciones, tal como RET e INT utilizan sólo un operando inmediato.

No está permitido utilizar el modo de direccionamiento inmediato en el operando destino.

2.5.2 Direccionamiento de registros

El valor del operando está almacenado en uno de los registros internos del 8086. Este puede ser un valor de 8 o 16 bits. El microprocesador interpretará la longitud del operando por el nombre del registro.

La mayoría de las instrucciones permiten el modo de direccionamiento de registro en uno o más operandos. Algunas instrucciones únicamente pueden usar ciertos registros como sus operandos. Generalmente, las instrucciones producen código de menor tamaño y de operación más rápida si especifican como su operando al registro acumulador. Los registros de segmentos sólo pueden ser utilizados en algunas pocas instrucciones y en circunstancias especiales.

Muchas instrucciones pueden utilizar como sus operandos a datos almacenados en la memoria principal de la computadora. Cuando se proporciona un operando de memoria, el microprocesador debe calcular la dirección del dato a ser procesado. Esta dirección es conocida como la *dirección efectiva*. El cálculo de la dirección efectiva depende del modo de direccionamiento de memoria que se especifique en el operando.

2.5.3 Direccionamiento directo de memoria

Un operando directo de memoria es un símbolo que representa la dirección (segmento y desplazamiento) de una instrucción o un dato. La dirección de desplazamiento representada por un operando directo es calculada en tiempo de ensamble. La dirección de cada operando relativo al inicio del programa es calculado en tiempo de enlace. La dirección efectiva se calcula en tiempo de carga del programa.

Los operandos directos de memoria pueden ser cualquier constante o símbolo que represente una dirección de memoria. Esto incluye a etiquetas, nombres de procedimientos, variables, o el valor del contador de localizaciones de memoria durante el ensamblado del programa.

La dirección efectiva es siempre relativa al registro de segmento. El registro de segmento por omisión es DS para los operandos directos de memoria, pero puede ser anulado proporcionando otro registro de segmento.

El desplazamiento dentro del segmendo de datos del operando está contenido en la instrucción como una cantidad de 16 bits. Este desplazamiento se suma al contenido desplazado del registro del segmento de datos (DS) para convertirlo a la dirección segmentada (o efectiva) de 20 bits. Habitualmente, el operando de direccionamiento directo es un rótulo.

2.5.4 Direccionamiento indirecto de registros

El valor del operando es señalado por una dirección de desplazamiento almacenada en uno de los siguientes registros: SI, DI, BX o, bajo algunas circunstancias, BP. El microprocesador reconoce el direccionamiento indirecto de registros por la sintaxis de la instrucción: el registro es encerrado dentro de corchetes (por ejemplo, [BX]). Este modo de direccionamiento puede ser utilizado para referenciar datos almacenados en forma de tabla. Así, es posible acceder a valores individuales con una iteración de incrementar el registro base (o cualquiera de los otros registros permitidos) y acceder a la posición de memoria.

2.5.5 Direccionamiento relativo de base

La dirección efectiva de un operando se obtiene de la suma del contenido de un registro base (BX o BP) y un desplazamiento, relativo al segmento seleccionado. Es usado frecuentemente para acceder a estructuras de datos complejas, como registros tipo Pascal: el registro base apunta a la base de la estructura, y se selecciona un campo particular con el desplazamiento. Al cambiar el desplazamiento se accede a campos diferentes en el registro. Acceder al mismo campo en registros diferentes es tan simple como cambiar el contenido del registro base.

2.5.6 Direccionamiento indizado directo

La dirección del desplazamiento del operando se calcula sumando el desplazamiento a un registro índice (SI o DI) en el segmento seleccionado. Frecuentemente, el direccionamiento indizado directo se utiliza para acceder a los elementos de un array (arreglo) estático. El valor del desplazamiento localiza el inicio del array y el valor almacenado en el registro índice seleeciona un elemento simple en la estructura. De manera distinta a los registros, cuyas longitudes de campo individual pueden variar en tamaño y tipo de datos, los elementos del array son homogéneos. Como los elementos del array son del mismo tipo de datos y tamaño, para moverse a través del arreglo basta con incrementar o decrementar sistemáticamente el desplazamiento.

2.5.7 Direccionamiento base indizado

El operando se localiza en el segmento seleccionado en un desplazamiento determinado por la suma de los contenidos del registro base, registro índice y, opcionalmente, un desplazamiento. Si no se incluye desplazamiento, entonces el direccionamiento base indizado se utiliza con más frecuencia para acceder a los elementos de un array dinámico (ésto es, un arreglo cuya dirección base puede cambiar durante la ejecución del programa). El incluir desplazamiento permite acceder a un elemento individual de un array, siendo el array un campo en una estructura, como por ejemplo un registro.

2.6 El conjunto de instrucciones del 8086

El conjunto de instrucciones primitivas que puede realizar un microprocesador es conocido como su *conjunto de instrucciones*. El conjunto de instrucciones del 8086 consiste de seis tipos de instrucciones, que están resumidas en la tabla 2.5. Programar satisfactoriamente en lenguaje ensamblador del 8086 requiere un entendimiento de todos estos tipos de instrucciones.

1. Instrucciones de transferencia de datos	
MOV	Mover
PUSH, POP	Operaciones de la pila (stack)
XCHG	Intercambio
IN, OUT	Puertos de E/S
2. Instrucciones aritméticas	
ADD	Adición
INC	Incremento
SUB	Substracción
DEC	Decremento
NEG	Negatividad
CMP	Comparar
MUL	Multiplicar
DIV	Dividir
3. Instrucciones lógicas	
NOT	Complemento
AND	AND
OR	OR inclusivo
XOR	OR exclusivo
TEST	Prueba de bits
SHL, SHR	Desplazamiento izquierda/derecha
ROL, ROR	Rotación izquierda/derecha
4. Instrucciones de manipulación de cadenas	
MOVS	Mover cadena
CMPS	Comparar cadenas
SCAS	Inspeccionar cadena
LODS	Cargar de cadena
STOS	Almacenar en cadena
5. Instrucciones de transferencia de control	

CALL	Llamada a una subrutina
RET	Retorno de una subrutina
JMP	Salto
JN, JNZ, Etc.	Saltos condicionales
LOOP	Iteración
LOOPNE..	Iteración condicional
INT	Interrupción
IRET	Retorno de interrupción
6. Instrucciones de control del procesador	
CLC,STC, Etc.	Aclarar/establecer banderas

Tabla 2.5. El conjunto de instrucciones del microprocesador 8086

2.6.1 Instrucciones de transferencia de datos

Las instrucciones de transferencia de datos nos permiten mover datos de un punto a otro. En general, los datos pueden ser movidos en tamaños de un byte o una palabra a la vez. Mover los datos puede parecer un concepto demasiado simple, pero la situación se complica debido a los diferentes modos disponibles para direccionar los datos a ser movidos.

La instrucción MOV mueve un dato entre los registros internos del 8086 y la memoria. Verdaderamente, más que mover la instrucción MOV almacena una copia del operando fuente en el operando destino, sin afectar al primero. La sintaxis de esta instrucción es la siguiente:

```
mov   destino fuente
```

Puede interpretarse esta instrucción de la siguiente forma: mueve el contenido de *fuente* hacia *destino*. Es importante subrayar que, por lo general, las instrucciones de lenguaje ensamblador manejan los operandos *fuente* y *destino* de manera consistente: el operando de la izquierda es el operando destino y el de la derecha es el operando fuente.

MOV acepta casi cualquier par de operandos que tienen sentido excepto cuando un registro de segmento es un operando. Cualquiera de los siguientes elementos puede utilizarse como el operando fuente de la instrucción MOV.

- Una constante
- Una expresión que evalúe a un valor constante
- Un registro de propósito general
- Una localidad de memoria accesada con cualquiera de los modos de direccionamiento de memoria discutidos anteriormente.

Como operando destino puede usarse cualquier registro de propósito general o una localidad de memoria.

Con la instrucción MOV es posible copiar valores de tamaño BYTE o palabra (WORD). En muchos casos, los operandos de MOV le dicen a Turbo Assembler exactamente qué tamaño de datos será utilizado. Si un registro de segmento está involucrado en la operación de copia, entonces el tamaño de los datos debe ser del tamaño del registro. Por ejemplo, los tamaños de los datos de las siguientes instrucciones son claros:

```
...
mov   al,1        ;Operandos de tamaño byte
mov   dx,1        ;Operandos de tamaño palabra
...
```

De manera similar, las localizaciones de memoria tienen un tamaño inherente, de tal forma que los tamaños de los datos son conocidos por TASM:

```
            ...
            .DATA
TestChar    DB    ?
TempPtr     DW    TestChar
            ...
            .CODE
            ...
            mov   [TestChar],'A'
            mov   [TempPtr],0
            ...
```

Algunas veces, los operandos de la instrucción MOV no tendrán un tamaño definido inherente. Por ejemplo, en la siguiente instrucción, el ensamblador no tiene forma de conocer el tamaño de los operandos involucrados en MOV.

```
            mov   [bx],1
```

y, de hecho, TASM determinará que no tiene forma de ensamblar la instrucción. Esta situación se repite cuando el programador considera conveniente accesar temporalmente una variable del tamaño de una palabra como un byte o viceversa.

TASM proporciona los medios para definir flexiblemente los tamaños de los datos en la forma de los operadores WORD PTR y BYTE PTR. WORD PTR le dice a Turbo Assembler que trate un operando de memoria determinado dimensionado como una palabra, y BYTE PTR le dice a TASM que accese un operando de memoria como un operando de tamaño byte, sin considerar su tamaño predefinido. Por ejemplo, la última línea de código puede reescribirse para almacenar un valor 1 de tamaño palabra en la dirección de memoria apuntada por BX.

```
            mov   WORD PTR [BX],1
```

WORD PTR y BYTE PTR no tienen sentido cuando son aplicados a registros, puestos que los registros son de tamaño fijo.

- **Accediendo a registros de segmento**

Aunque la instrucción MOV puede utilizarse para mover valores a y desde registros de segmento, este es un caso especial, más limitado que los otros usos de MOV. Si un registro de segmento es un operando de MOV, el otro operando debe ser un registro de propósito general o una localidad de memoria. No es posible cargar directamente una constante en un registro de segmento y un registro de segmento no puede ser copiado directamente a otro registro de segmento.

Por ejemplo, a continuación se muestran dos formas de inicializar el registro de segmento ES con el valor del segmento .DATA.

```
            ...
            .DATA
DataSeg     DW    @data
            ...
            .CODE
            ...
            mov   ax,@data
            mov   es,ax
            ...
            mov   es,[DataSeg]
            ...
```

Es importante notar que la instrucción MOV no es la única instrucción que limita el uso de los registros de segmento; la mayoría de las instrucciones no pueden hacer uso de los registros de segmento como operandos.

- **Moviendo datos hacia y desde la pila**

Las instrucciones PUSH y POP son instrucciones especiales de transferencia de datos que implementan una estructura de memoria denominada *pila*. Esta estructura sigue la política "el último en entrar es el primero en salir" (Last-In, First-Out, por sus siglas en inglés). La pila siempre reside en el Segmento de Pila, lo que ocasiona que el registro SS sea siempre utilizado, durante las referencias a la pila. El registro apuntador de la pila, SP, es implícitamente usado como una dirección de memoria en todas las operaciones de la pila. La instrucción PUSH decrementa el contenido de SP en dos y después almacena su operando en la dirección de memoria especificada por SP. La instrucción POP trae los datos de la dirección de memoria especificada por SP y lo guarda en su operando, para después incrementar el contenido de SP en dos. Es importante notar que las instrucciones PUSH y POP siempre transfieren una palabra de datos; en contraste con la instrucción MOV, que puede mover un byte o una palabra.

La instrucción MOV puede ser usada para accesar datos de la pila utilizando los modos de direccionamiento de memoria que utilizan al registro BP como un apuntador base; por ejemplo,

```
mov   ax,[bp+4]
```

carga el registro AX con el contenido de la palabra en el desplazamiento BP+4 en el segmento de pila.

- **Acceso a puertos de Entrada/Salida**

Las instrucciones IN y OUT son usados para accesar los puertos de entrada/salida del 8086. Los dispositivos externos como terminales, impresoras y manejadores de disco, se comunican con el microprocesador vía los puertos de E/S. Un byte de 8 bits o una palabra de 16 bits puede ser transferido a través de un puerto de E/S. Dentro del 8086, los datos deben ser enviados siempre (o recibidos) del registro acumulador, AX. En los casos en que se transfieren únicamente 8 bits, se usa la mitad baja del acumulador, AL. La dirección de E/S se especifica como el contenido del registro DX, o como un valor inmediato contenido en la instrucción. En el caso posterior, solamente los primeros 256 puertos de E/S (direcciones de E/S 00h a FFh) pueden ser accesadas. Se muestran a continuación ejemplos de uso de las instrucciones IN y OUT.

```
in    al,2Fh      ;Entrada de un byte del puerto 2Fh
out   5,al        ;Salida de un byte hacia el
                  ;puerto 5
mov   dx,3FCh
in    ax,dx       ;Entrada de una palabra del
                  ;puerto 3FCh
```

- **Intercambiando datos**

La instrucción XCHG intercambia el contenido de sus dos operandos. El segundo operando de la instrucción XCHG debe ser siempre un registro. El primer operando puede ser accesado usando cualquiera de los modos de direccionamiento descritos en la unidad anterior, excepto el modo inmediato. La operación de XCHG se ilustra a continuación:

```
mov   ax,5        ; AX = 5
mov   bx,10       ; BX = 10
xchg  ax,bx       ; AX = 10, BX = 5
```

La instrucción XLATB, o trasladar, realiza una operación de búsqueda en tabla. El contenido del registro AL se agrega al contenido del registro BX, y el valor resultante es usado como una dirección de memoria. El byte en esta dirección de memoria se coloca en el registro AL. Normalmente, esta instrucción se utiliza inicializando el registro BX para que apunte al inicio de la tabla de traducción. La instrucción XLATB después convierte el valor del byte en AL a su correspondiente valor de la tabla.

2.6.2 Instrucciones aritméticas

Las instrucciones aritméticas son usadas para realizar cálculos aritméticos. Las instrucciones ADD, SUB y CMP tienen dos operandos. Como siempre, el primer operando especificado servirá como destino para el resultado. Así, la instrucción

```
add   ax,bx
```

agregará el contenido del registro AX al contenido del registro BX y colocará el resultado en el registro AX. Similarmente, la instrucción

```
sub   ax,bx
```

substraerá el contenido del registro BX del contenido del registro AX y colocará el resultado en el registro AX. La instrucción de comparación, CMP, realiza la misma operación que la instrucción SUB, pero sin afectar a ninguno de los operandos. Es usada principalmente para activar las banderas, tal como se explicó anteriormente.

Las instrucciones INC, DEC y NEG utilizan sólo un operando. Puede ser direccionado usando cualquiera de los métodos de direccionamiento de memoria, excepto el inmediato. La instrucción INC (incremento) incrementa en uno a su operando. La instrucción DEC (decremento) disminuye en uno su operando. La instrucción NEG (negar) obtiene el complemento a dos de su operando.

La flexibilidad de estas instrucciones se ve incrementada por nuestra habilidad para usarlas con operandos de byte o palabra. En adición, todas las instrucciones aritméticas, afectan varios bits del registro bandera para indicar el estado de su resultado. Las instrucciones de transferencia condicional pueden utilizarse para probar estos bits y cambiar el flujo de un programa.

La multiplicación y la división son funciones avanzadas del microprocesador y son un poco menos flexibles de usar que las funciones aritméticas más comunes discutidas anteriormente. El operando destino es siempre el registro acumulador (AX), por lo que sólo un operando fuente se codifica con la instrucción. Es importante aclarar que cuando se multiplica un byte por un byte, el resultado puede ser tan grande como una palabra. Por lo tanto, en una operación de multiplicación de bytes, el operando fuente (un byte) se multiplica por el contenido del registro AL, y el resultado (una palabra) es colocado en el registro AX. En el caso de una operación de multiplicación entre palabras, el operando fuente (una palabra) se multiplica por el contenido del registro AX. El resultado puede ser tan grande como 32 bits: la palabra de orden bajo se coloca en el registro AX y la palabra de orden superior se coloca en el registro DX.

Inversamente, en una operación de división utilizando bytes, el numerador, tomado como el registro entero AX, se divide por el operando fuente de un byte. El cociente es colocado en AL, y el residuo se coloca en AH. Una operación de división utilizando palabras, asume un numerador de 32 bits, con la palabra de alto orden tomada del registro DX y la palabra de

bajo orden tomada del registro AX. Este numerador es dividido por el operando fuente de una palabra. El cociente se coloca en AX y el residuo en DX.

Las instrucciones MUL y DIV anteriormente explicadas realizan multiplicación y división *sin signo*, respectivamente. La multiplicación y la división *signada* puede realizarse con las versiones de operandos de tipo entero de estas instrucciones, llamadas IMUL y IDIV.

2.6.3 Instrucciones lógicas

Las instrucciones aritméticas siempre asumen que sus operandos representan información numérica. Por el contraste, las instrucciones lógicas tratan a sus operandos como simples cadenas de bits.

La instrucción NOT simplemente invierte cada bit en su operando, cambiando todos los ceros a unos y todos los unos a ceros. Las instrucciones AND, OR y XOR usan dos operandos: las combinaciones de operandos disponibles son las mismas que las disponibles en las operaciones aritméticas. El resultado de una instrucción AND tiene puestos los bits en las posiciones donde ambos bits de sus operandos son uno. El resultado de una instrucción OR tiene los bits puestos en aquellas posiciones donde algunos de sus operandos tiene puesto un bit. Es muy útil para forzar los bits seleccionados a uno dentro de su operando destino. El resultado de una instrucción XOR tiene los bits puestos únicamente en aquellas posiciones donde alguno pero no ambos de sus operandos tiene los bits puestos. Es muy útil para invertir bits seleccionados dentro de su operando destino.

Bit fuente A	Bit fuente B	A and B	A or B	A xor B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Tabla 2.6 La operación de las instrucciones lógicas AND, OR y XOR

La instrucción TEST realiza la misma función que la instrucción AND, con la diferencia que el operando destino no es alterado. Es usado principalmente para probar si uno o más bits específicos en un byte o palabra están puestos (son uno).

Las instrucciones lógicas siempre aclaran (ponen en cero) las banderas de acarreo (CF) y desbordamiento (OF). Además, ellas ponen (hacen uno) la bandera de cero (ZF) para indicar si el resultado fue cero o no.

Las instrucciones de *desplazamiento* (shift) mueven los bits hacia un lado (esto es, de izquierda a derecha o de derecha a izquierda) dentro de sus operandos.

SHL (desplazamiento a la izquierda) mueve cada bit del operando destino un lugar hacia la izquierda, o hacia el bit más significativo. El bit más significativo es desplazado fuera del operando hacia la bandera de acarreo; el bit menos significativo es llenado con el bit cero.

La figura 2.7 muestra cómo el valor 10010110b (96h) almacenado en el registro AL es desplazado hacia la izquierda con la instrucción

```
shl al, 1
```

El resultado es el valor 00101100b (2Ch), que se deposita en el mismo registro AL. La bandera de acarreo almacenará el valor 1, inicialmente en el bit más significativo de AL.

El uso más común de la instrucción SHL es realizar veloces multiplicaciones por potencias de dos.

El operando fuente de las operaciones de desplazamiento y rotación puede ser el valor inmediato 1, que indica que el desplazamiento se realizará sólo un bit, o el registro cl, para indicar la cuenta de un desplazamiento mayor que uno. El siguiente ejemplo multipliza el registro dx por cuatro.

```
mov cl,2
shl dx,cl
```

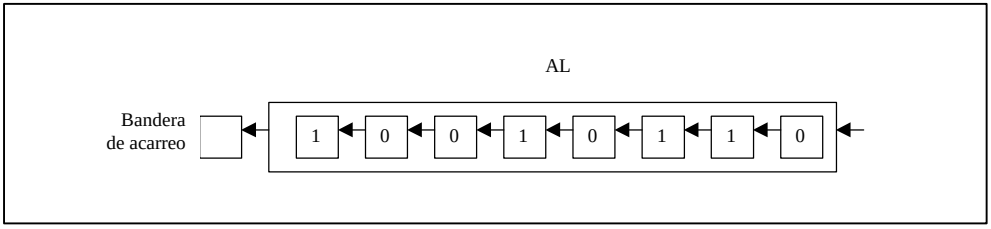

Fig. 2.7 Ejemplo de un desplazamiento a la izquierda

La instrucción SHR (desplazamiento a la derecha) es similar a SHL, sólo que realiza el desplazamiento de 1 o cl bits hacia la derecha. El bit menos significativo es colocado en la bandera de acarreo y el bit más significativo se rellena con cero. La instrucción SHR es utilizada para realizar divisiones sin signo por potencias de dos.

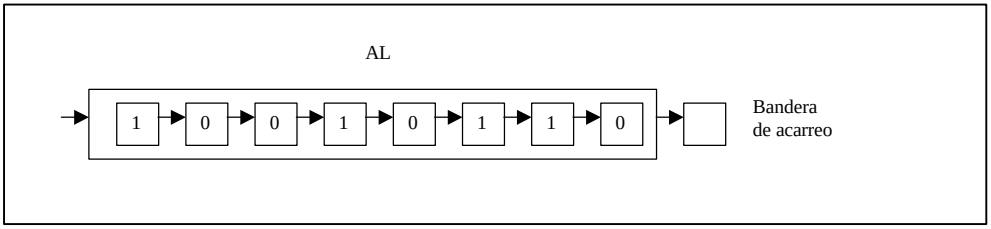

Fig. 2.8 Ejemplo de un desplazamiento a la derecha

2.6.4 Instrucciones de transferencia de control

- **Saltos incondicionales**

La instrucción fundamental de bifurcación en los microprocesadores de la familia 80x86 es la instrucción JMP. La instrucción JMP instruye al 8086 para que ejecute la instrucción en la etiqueta destino como la instrucción que sigue a JMP. Por ejemplo, cuando finaliza la ejecución del siguiente segmento de código

```
      ...
      mov   ax,1
      jmp   AddTwoToAX
AddOneToAX:
      inc   ax
      jmp   AXIsSet
AddTwoToAX:
      add   ax,2
AXIsSet:
      ...
```

AX contiene 3, y las instrucciones ADD y JMP que siguen a la etiqueta `AddOneToAX` nunca son ejecutadas. Aquí la instrucción

```
      jmp   AddTwoToAX
```

instruye al 8086 para que se le asigne al apuntador de instrucciones, el registro IP, el desplazamiento de la etiqueta `AddTwoToAX`, para que la siguiente instrucción a ejecutar sea

```
      add   ax,2
```

Algunas veces junto con la instrucción JMP se utiliza el operador SHORT. JMP usualmente utiliza un desplazamiento de 16 bits para apuntar a la etiqueta destino; SHORT instruye a TASM para que utilize un desplazamiento de 8 bits, ahorrando un byte por instrucción JMP. Por ejemplo, el último ejemplo es dos bytes más pequeño con las siguientes modificaciones:

```
      ...
```

```
      mov   ax,1
      jmp   SHORT AddTwoToAX
AddOneToAX:
      inc   ax
      jmp   SHORT AXIsSet
AddTwoToAX:
      add   ax,2
AXIsSet:
      ...
```

La desventaja en usar el operador SHORT es que algunos saltos cortos pueden alcanzar a las etiquetas que se encuentran únicamente dentro del alcance de 128 bytes de la instrucción JMP, de tal forma que TASM puede informar que no puede alcanzar una etiqueta determinada debido a que está fuera del alcance de la instrucción JMP que utiliza un operador SHORT.

Unicamente tiene sentido utilizar saltos SHORT cuando se hacen saltos hacia adelante en el código, debido a que TASM inteligentemente hace cortos los saltos hacia atrás cuando están dentro del alcance del destino; los saltos hacia atrás que están más allá de los 128 bytes de la instrucción JMP automáticamente son tomados como saltos con desplazamiento de 16 bits.

JMP puede ser utilizado para saltar a otro segmento de código, cargando los registros CS e IP con una sola instrucción. Por ejemplo,

```
            ...
CSeg1 SEGMENT
            ASSUME      cs:CSeg1
            ...
FarTarget   LABEL FAR
            ...
Cseg1 ENDS
            ...
CSeg2 SEGMENT
            ASSUME      cs:CSeg2
            ...
            jmp   FarTarget   ; este es un salto far
            ...
Cseg2 ENDS
            ...
```

realiza un salto a otro segmento de código.

Finalmente, puede saltarse a una dirección almacenada en un registro o en una variable de memoria. Por ejemplo,

```
      ...
      mov   ax, OFFSET TestLabel
```

```
      jmp   ax
      ...
TestLabel:
      ...
```

salta a la etiqueta `TestLabel`, tal y como también lo hace

```
      ...
      .DATA
JumpTarget  DW TestLabel
      ...
      .CODE
      ...
      jmp   [JumpTarget]
      ...
TestLabel:
      ...
```

- **Saltos condicionales**

Los saltos como los descritos en la sección anterior son únicamente parte de lo necesario para escribir programas útiles. La mayor parte de los programas se benefician de los saltos basados en la toma de decisiones. Una instrucción de salto condicional puede saltar o no a una etiqueta destino, dependiendo del estado del registro de banderas. Por ejemplo, considere el siguiente segmento de código:

```
      ...
      mov   ah,1        ; #Fn DOS para entrada desde teclado
      int   21h         ; Obtiene la siguiente tecla
      cmp   al,'A'      ; Se presionó la letra 'A'?
      je    AWasTyped   ; Sí, procesa la tecla presionada
      mov   [TempByte],al     ; No, almacena el carácter
      ...
AWasTyped:
      push  ax          ; Guarda el carácter en la pila
      ...
```

Primero, el código obtiene una tecla del teclado a través de la función 1 del DOS. Después, utiliza la instrucción CMP para comparar el carácter leído con la letra 'A'. La instrucción CMP hace que la bandera de cero (ZF) sea 1, si el resultado de la comparación es igual, o sea aclarada (0), si la comparación evalúa a diferente.

La instrucción JE es una instrucción de salto condicional que salta a la etiqueta destino sólo si la bandera de cero es 1. De otra forma, se ejecutará la instrucción inmediata a la instrucción JE, en este caso una instrucción MOV. La bandera de cero será 1 sólo cuando se presione la tecla A, y únicamente entonces saltará el 8086 a la instrucción PUSH en la etiqueta `AWasTyped`.

El 8086 proporciona una gran variedad de saltos condicionales, proporcionando la habilidad de saltar conforme el estado de cualquier bandera o combinación de banderas. Puede saltarse condicionalmente respecto al estado de las banderas de cero, acarreo, signo, paridad y desbordamiento y de acuerdo a la combinación de banderas que indique el resultado de operaciones con números con signo.

Nemónico	Significado	Sinónimo	Significado
JA	Salta si es superior	JNBE	Salta si no es inferior o igual
JAE	Salta si es superior o igual	JNB	Salta si no es inferior
JB	Salta si es inferior	JNAE	Salta si no es superior o igual
JBE	Salta si es inferior o igual	JNA	Salta si no es superior
JE	Salta si es igual	JZ	Salta si el resultado es cero
JNE	Salta si no es igual	JNZ	Salta si el resultado no es cero
JG	Salta si es mayor que	JNLE	Salta si no es menor o igual que
JGE	Salta si es mayor o igual que	JNL	Salta si no es menor que
JL	Salta si es menor que	JNGE	Salta si no es mayor o igual que
JLE	Salta si es menor o igual que	JNG	Salta si no es mayor que

Tabla 2.7. Nemónicos de instrucción de saltos condicionales y sus sinónimos

Condición	Operador de Pascal	Valores sin signo	Salta cuando	Valores con signo	Salta cuando
Igual	=	JE	ZF=1	JE	ZF=1
Diferente	<>	JNE	ZF=0	JNE	ZF=0
Mayor que	>	JA	CF=0 and ZF=0	JG	ZF=0 or
No menor o igual que		JNBE			SF=OF
Menor que	<	JB	CF=1	JL	SF<>OF
No mayor o igual que		JNAE		JNGE	
Mayor o igual que	>=	JAE	CF=0	JGE	SF=OF
No menor que		JNB		JNL	
Menor o igual que	<=	JBE	CF=1 or ZF=1	JLE	ZF=1 and SF<>OF

No mayor que		JNA			

Tabla 2.8 Pruebas aritméticas útiles después de una intrucción CMP.

- **Instrucciones de iteración**

Una iteración es un bloque de código que finaliza con un salto condicional, de tal forma que el código puede ser ejecutado repetidamente hasta que la condición de terminación sea alcanzada.

El microprocesador 8086 proporciona varias instrucciones especiales para realizar iteraciones.

La instrucción loop

Supóngase que se desea imprimir una cadena de una longitud de 10 caracteres. Esto podría ser realizado con el siguiente código:

Sin embargo, existe una forma más fácil de hacerlo mediante la instrucción LOOP.

La instrucción LOOP decrementa el contenido del registro CX y finaliza la iteración si CX ha alcanzado el valor de cero. Si CX es diferente de cero, entonces se ejecutará la instrucción indicada por la etiqueta destino, operando de la instrucción LOOP.

La instrucción LOOPE hace lo mismo que LOOP, excepto que LOOPE finalizará el ciclo si CX llega a cero o si la bandera de cero es 1. Es importante recordar que la bandera de cero es 1, cuando el resultado de una operación aritmética es cero o si los dos operandos en la última comparación son iguales.

La instrucción LOOPNE, por su parte, finalizará el ciclo si CX es cero o la bandera de cero es 0.

2.6.5 Instrucciones de manipulación de cadenas

Las instrucciones de manejo de cadenas difieren de las demás instrucciones del 8086 en que pueden accesar memoria e incrementar y decrementar un registro apuntador en una sola instrucción.

Como su nombre implica, las instrucciones de cadena son particularmente útiles en la manipulación de cadenas de caracteres. Son también adecuadas para el manejo de arreglos, búfers de datos y todo tipo de cadenas de bytes y palabras. Además, generan menos códigos y son más rápidas que las combinaciones equivalentes de instrucciones normales del 8086, tal como LOOP, INC y MOV.

Las instrucciones de cadena pueden agruparse en los siguientes grupos funcionales: instrucciones usadas para el movimiento de datos (LODS, STOS y MOVS) e instrucciones de cadena para inspección y comparación de datos (SCAS y CMPS).

- **Instrucciones de cadena de movimiento de datos**

Las instrucciones de cadena de movimiento de datos son similares a la instrucción MOV, pero hacen más que MOV y operan con mayor velocidad.

LODS

La instrucción LODS, que carga un byte o palabra de memoria hacia el acumulador, puede usarse para trasladar bytes, con su variante LODSB, o palabras, utilizando LODSW. LODSB carga el byte direccionado por DS:SI hacia AL, e incrementa o decrementa SI dependiendo del estado de la bandera de dirección. Si la bandera de dirección es 0 (puesta con CLD), entonces SI es incrementado; si la bandera de dirección es 1 (puesta con STD), entonces SI es decrementado. Es importante notar que la bandera de dirección controla la dirección en que los registros apuntadores son modificados para todas las instrucciones de cadena.

Por ejemplo, la instrucción LODSB en la siguiente porción de código

```
...
cld
mov   si,0
lodsb
...
```

carga AL con el contenido del byte en el desplazamiento 0 del segmento de datos e incrementa SI por 1. Es equivalente a

```
...
mov   si,0
mov   al,[si]
inc   si
...
```

Sin embargo,

```
lodsb
```

es considerablemente más rápido (y 2 bytes menor) que

```
mov   al,[si]
inc   si
```

LODSW es similar a LODSB, salvo que la palabra direccionada por DS:SI es cargada en AX y SI es incrementado o decrementado por 2, en lugar de por 1. Por ejemplo,

```
...
mov   si,10
lodsw
...
```

carga la palabra en el desplazamiento 10 del segmento de datos hacia AX y después decrementa SI por 2.

STOS

STOS es el complemento de LODS. Escribe un valor de byte o palabra del acumulador hacia una localidad de memoria apuntada por ES:DI, e incrementa o decrementa DI. STOSB escribe el byte en AL hacia la localidad de memoria apuntada por ES:DI y después incrementa o decrementa DI, dependiendo de la bandera de dirección. Por ejemplo,

```
      ...
      std
      mov   di,0FFFFh
      mov   al,55h
      stosb
      ...
```

escribe el valor 55h al byte en el desplazamiento 0FFFFh en el segmento apuntado por ES y después decrementa DI a 0FFFEh.

STOSW hace lo mismo, al escribir un valor 16 bits de AX hacia la dirección ES:DI y después incrementa o decrementa DI por 2. Por ejemplo:

```
      ...
      cld
      mov   di,0FFEh
      mov   ax,102h
      stosw
      ...
```

escribe el valor de palabra 102h de AX hacia el desplazamiento 0FFEh en el segmento apuntado por ES y después incrementa DI a 1000h.

LODS y STOS trabajan juntos para el copiado de buffers. Por ejemplo, la siguiente subrutina copia una cadena terminada en cero, en DS:SI hacia la cadena ES:DI.

```
; Subrutina para copiar una cadena terminada con ASCII cero
hacia otra cadena.
; Entrada:
;     DS:SI - Cadena fuente
;     ES:SI - Cadena destino
; Salida:
;     Ninguna
; Registros destruídos:
;     AL,SI,DI
CopyString  PROC
      cld         ; Hace qu SI y DI se incremente con
                  ; la inst. de cadena
CopyStringLoop:
```

```
      lodsb       ; Obtiene carácter de la cadena fuente
      stosb       ; Almacena carácter en la cadena destino
      cmp   al,0  ; Es el carácter 0 para terminar
                  ; la cadena?
      jnz   CopyStringLoop    ; no, ve por el siguiente
                  ; carácter
      ret
CopyString  ENDP
```

MOVS

MOVS es similar a LODS y STOS en una sola instrucción. MOVS lee el byte o palabra almacenados en DS:SI y después escribe el valor en la dirección ES:DI. El byte o palabra no pasa por registro alguno y, de esta manera, AX no es modificado. La siguiente porción de código muestra como funciona MOVS para copiar bloques de bytes que no están terminados con el ASCII cero.

```
      ...
      mov   cx,ARRAY_LENGTH_IN_WORDS
      mov   si,OFFSET SourceArray
      mov   ax,SEG SourceArray
      mov   ds,ax
      mov   di,OFFSET DestArray
      mov   ax,SEG DestArray
      mov   es,ax
      cld
CopyLoop:
      movsw
      loop  CopyLoop
      ...
```

- **Prefijos de repetición de cadenas**

Mientras el código en el último ejemplo parece eficiente, existe una manera de realizar la misma tarea de mover bloques de byte en una sola instrucción. El 8086 cuenta con la opción de prefijo de repetición, REP, en las instrucciones de cadena.

REP no es una instrucción; es un *prefijo de instrucción*. Los prefijos de instrucción modifican la operación de la siguiente instrucción en el código. REP indica que la siguiente instrucción de cadena se ejecute repetidamente hasta que el registro CX sea cero. (Si CX es cero cuando la instrucción a repetir inicia, la instrucción se ejecuta cero veces - en otras palabras no hace nada).

Usando REP puede reemplazarse la porción

```
CopyLoop:
      movsw
```

```
      loop  CopyLoop
```

en el último ejemplo, con

```
      rep   movsw
```

Esta instrucción simple moverá un bloque de hasta 65,535 palabras (0FFFFh) de la dirección de memoria iniciando en DS:SI a la localidad en ES:DI.

REP puede ser usada con LODS y STOS, como con MOVS (y también con SCAS y CMPS).

Es importante aclarar que REP únicamente funciona con las instrucciones de manipulación de cadena.

- **Instrucciones de cadena de inspección de datos**

Las instrucciones SCAS y CMPS son usadas para inspeccionar y comparar bloques de memoria.

SCAS

SCAS es usada para inspeccionar la memoria buscando una equivalencia o no equivalencia de un valor particular de tamaño byte o palabra. Como con todas las instrucciones de cadena SCAS viene en dos formas: SCASB y SCASW.

SCASB compara AL con el valor de tamaño byte almacenado en la dirección ES:DI, modificando las banderas para reflejar la comparación, tal como si se hubiera ejecutado una instrucción CMP. Como con STOSB, DI es incrementado o decrementado por SCASB.

El siguiente ejemplo, encuentra la primera ocurrencia de la letra 't' minúscula en la cadena de texto TEXTSTRING.

```
      ...
      .DATA
TextString  DB    'Este es un test',0
TEXT_STRING_LENGTH      EQU   ($-TextString)
      ...
      .CODE
      ...
      mov   ax,@data
      mov   es,ax
      mov   di,OFFSET TextString    ;ES:DI apunta al
                                    ; inicio de TextString
      mov   al,'t'                  ; Carácter a buscar
      mov   cx,TEXT_STRING-LENGTH   ; Longitud de la cadena
                                    ; a inspeccionar
      cld
ScanLoop:
      scasb                  ; El contenido de ES:DI es
igual a 't'
```

```
      je    Found_t           ; Sí; se encontró 't'
      loop  ScanLoop
;     No se encontró 't'
      ...
;     Se encontró una 't'
Found_t:
      dec   di
```

Nótese que DI es decrementado después de encontrar 't', lo que refleja que, después de ejecutar la instrucción de cadena, los registros apuntador (DI, SI, o ambos) no apuntan a la localidad de memoria recientemente accesada, sino que apuntan a la siguiente (o anterior) localidad de memoria de la que se accesó. En el ejemplo anterior, DI apunta al byte que sigue a la localidad de memoria donde se encuentra 't' que fue encontrada y debe ser ajustada para compensar este desfasamiento.

Una porción de código similar a la anterior, pero sin utilizar instrucciones de cadena, se muestra a continuación:

```
      ...
ScanLoop:
      cmp   es:[di],al        ; Compara es:[di] con al
      je    Found_t                 ; Se encontró 't'
      inc   di
      loop  ScanLoop          ; No se encontró; ve por el
siguiente carácter
      ...
```

En este ejemplo, el registro DI es incrementado después de que la instrucción JE se ejecuta para evitar alterar las banderas con la instrucción INC.

Es importante hacer la observación que las instrucciones de cadena nunca modifican las banderas para reflejar los cambios que realizaron a SI, DI y/o CX. LODS, STOS y MOVS no modifican ninguna bandera, y SCAS y CMPS únicamente cambian las banderas de acuerdo a los resultados de las comparaciones realizadas.

- **Otros prefijos de repetición**

El prefijo REP pudo utilizarse en el ejemplo de encontrar la 't' en la cadena. Sin embargo, puede desearse agregar flexibilidad a la comparación y desear detener la iteración cuando se encuentre una equivalencia o desigualdad entre los valores a comparar. Dos variantes de REP pueden ser utilizadas con SCAS (y también con CMPS): REPE y REPNE.

REPE (también conocida como REPZ) le dice al 8086 que repita SCAS (o CMPS) hasta que CX sea cero u ocurra una desigualdad entre los valores a comparar. REPE puede ser traducido como el prefijo "repite mientras sean iguales". Por otra parte, REPNE (conocido también como REPNZ) le dice al 8086 que repita SCAS (o CMPS) hasta que CX sea cero u ocurra una equivalencia. Puede tomarse a REPNE como el prefijo "repite mientras no sean iguales".

La siguiente porción de código utiliza una instrucción SCASB para inspeccionar TEXTSTRING en busca del carácter 't'.

```
      ...
      mov   ax,@data
      mov   es,ax
      mov   di,OFFSET TextString    ; ES:DI apunta al inicio
de TextString
      mov   al,'t'                  ; Carácter a buscar
      mov   cx,TEXT_STRING_LENGTH   ; Longitud de la cadena a
inspeccionar
      cld                       ; Inspecciona incrementando DI
      repne scasb               ; Inspecciona la cadena
completa mientras
                                ; ES:DI y AL no sean iguales
      je    Found_t                   ; Sí, se encontró
; No se encontró 't'
      ...
; Se encontró 't'
Found_t:
      dec   di          ; Hace que apunte exactamente a 't'
      ...
```

CMPS

La instrucción de cadena CMPS está diseñada para permitir comparar dos cadenas de bytes o palabras. Una reptición simple de CMPS compara dos localidades de memoria, y después incrementa SI y DI.

CMPSB compara el byte en DS:SI con el byte en ES:DI, modifica las banderas de acuerdo al resultado de la comparación e incrementa SI y DI, dependiendo de la bandera de dirección. AX no es modificado durante la ejecución de CMPS.

CMPSW hace lo mismo que CMPSB, pero con valores de 16 bits (una palabra) de tamaño.

La siguiente porción de código prueba si los primeros 50 elementos de dos arreglos con elementos de tamaño palabra son idénticos, usando REP CMPSW:

```
      ...
      mov   si,OFFSET Array1
      mov   ax,SEG Array1
      mov   ds,ax
      mov   di,OFFSET Array2
      mov   ax,SEG Array2
      mov   es,ax
      mov   cx,50
```

```
      cld
      repe  cmpsw
      jne   ArraysAreDifferent
; Los primeros 50 elementos son idénticos
      ...
; Al menos un elemento difiere entre los dos arreglos
ArraysAreDifferent:
      dec   si
      dec   si
      dec   di
      dec   di
      ...
```

- **Instrucciones de control del procesador**

La familia de procesadores 80x86 proporcionan instrucciones para el control del procesador. La tabla x.xx muestra las instrucciones más importantes de este grupo que permiten establecer o aclarar las banderas de acarreo, dirección e interrupción de manera directa. Además, existe una instrucción de complemento de la bandera de acarreo, CMC, que permite invertir el estado actual de la bandera de acarreo.

Por último, la instrucción HLT causa que el procesador detenga la ejecución de instrucciones. Puede utilizarse si no hay nada que ejecutar mientras se espera la recepción de una interrupción.

Mnemónicos de nstrucción	Significado	Acción realizada
CLC	Aclara la bandera de acarreo	CF = 0
CLD	Aclarar la bandera de dirección	DF = 0
CLI	Aclarar la bandera de interrupción	IF = 0
CMC	Complementa la bandera de acarreo	CF = NOT(CF)
STC	Establece la bandera de acarreo	CF = 1
STD	Establece la bandera de dirección	DF = 1
STI	Establece la bandera de interrupción	IF = 1
HLT	Detener la ejecución del programa	El procesador detiene la ejecución de instrucciones