CAPÍTULO 6

PROCEDIMIENTOS

Los programas que se han desarrollado hasta ahora contienen pocas líneas de código, una o dos páginas a lo sumo. Conforme los programas crecen en tamaño y complejidad se vuelve difícil y tedioso depurarlos. En este caso, resulta conveniente dividir el programa en varias partes llamadas procedimientos. Un **procedimiento** es una colección de instrucciones relacionadas que realiza una tarea específica. También un procedimiento puede contener un conjunto de instrucciones que deseamos que se ejecuten en varias partes del programa. Los procedimientos del lenguaje ensamblador tienen su contraparte en los lenguajes de alto nivel, por ejemplo, en el lenguaje C estos procedimientos se llaman funciones. Aunque en los lenguajes de alto nivel, el mecanismo empleado por los procedimientos para implementar su llamada, ejecución y regreso es transparente para el usuario, en ensamblador se requiere entender ese mecanismo. Un componente indispensable empleado por ese mecanismo es la pila del programa. La pila de un programa también se emplea para implementar el paso de parámetros a los procedimientos así como para crear las variables locales al procedimiento.

La Pila De Un Programa

Todo programa escrito en el lenguaje ensamblador del 8086 requiere de una pila. La pila se emplea para almacenar temporalmente direcciones y datos. En los programas hechos para el microprocesador 8086, una pila es un arreglo de palabras. El tamaño de la pila lo establece el programador mediante la directiva **stack** en el **código de inicio** de un programa. Tal como se estudió en el tema 3, la directiva **stack** cuya sintaxis es:

stack *tamPila*

crea una pila para el programa de tamaño *tampila*. El cargador del programa inicializa en forma automática el registro de segmento de pila **SS** a la dirección del segmento en que se creó la pila y el registro apuntador de pila **SP** al valor de *tamPila*. La figura 6.1 muestra el estado inicial de la pila de un programa al ser cargado a memoria para su ejecución (la pila se encuentra vacía).

Note que el registro apuntador de pila **SP** apunta a la palabra que está pasando el final de la pila. La pila difiere de los otros segmentos en la manera en que almacena los datos: empieza en la localidad más alta (la que tiene la dirección dada por **SS:*tamPila*-2**) y almacena los datos hacia las localidades más bajas de la pila (hacia **SS:0000**).

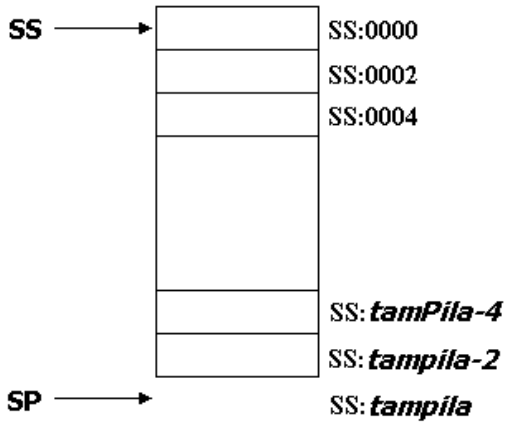

Figura 6-1

Instrucciones para el manejo de la pila

El ensamblador del 8086 posee un conjunto de instrucciones especiales para insertar y extraer datos en la pila:

push *fuente*

Transfiere el valor dado por *fuente* a la pila del programa.

Sintaxis:

push *regW|memW*

Donde *regW* puede ser cualquiera de los registros de propósito general o de segmento.

La instrucción **push** transfiere una palabra a la pila del programa. La instrucción **push** primero decrementa el valor del registro apuntador de pila **SP** en dos. A continuación copia el valor especificado por su operando a la localidad cuya dirección está dada por **SS:SP**. Note que esto hace que la pila crezca hacia las direcciones más pequeñas de memoria.

La instrucción **push** no afecta a las banderas.

pop *destino*

Extrae un valor del tope de la pila del programa y lo almacena en *destino*.

Sintaxis:

pop *regW|memW*

Donde *regW* puede ser cualquiera de los registros de propósito general o de segmento excepto el registro de segmento de código CS, que no está permitido. La instrucción **pop** extrae una palabra de la localidad cuya dirección está dada por **SS:SP** (el tope de la pila del programa) y lo almacena en la localidad especificada por el operando. A continuación incrementa el valor del registro apuntador de pila **SP** en dos.

La instrucción **pop** no afecta a las banderas.

En la figura 6.2 se muestran los estados de la pila después de insertar los valores de los registros **AX** y **BX** mediante las instrucción **push** y en la figura 6.3 se muestran los estados de la pila después de extraer esos valores de la pila mediante la instrucción **pop** al ejecutar el siguiente fragmento de código:

```
mov     ax, 0123h
mov     bx, 4567h
push    ax
push    bx
pop     bx
pop     ax
```

Note en la figura 6.3a que al extraer el valor 4567h de la pila, éste no se elimina físicamente de la pila. Sin embargo no hay una garantía de que ese valor permanezca en la pila. Si en ese momento insertamos un nuevo valor en la pila, éste se escribirá en la localidad **SS**:*tamPila-4* sobrescribiendo el valor de 4567h.

pushf

Transfiere el registro de banderas a la pila del programa.

Sintaxis:

pushf

La instrucción **pushf** transfiere el contenido del registro de banderas a la pila del programa de la misma manera en la que opera la instrucción **push**.

La instrucción **pushf** no afecta a las banderas.

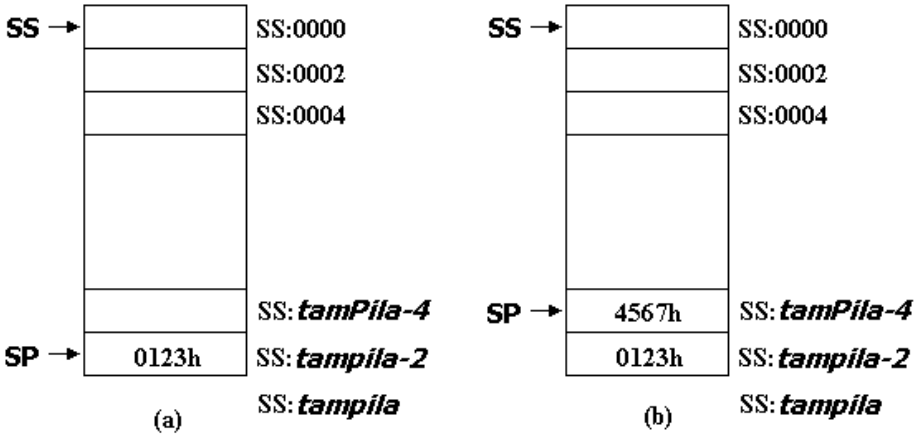

Figura 6-2

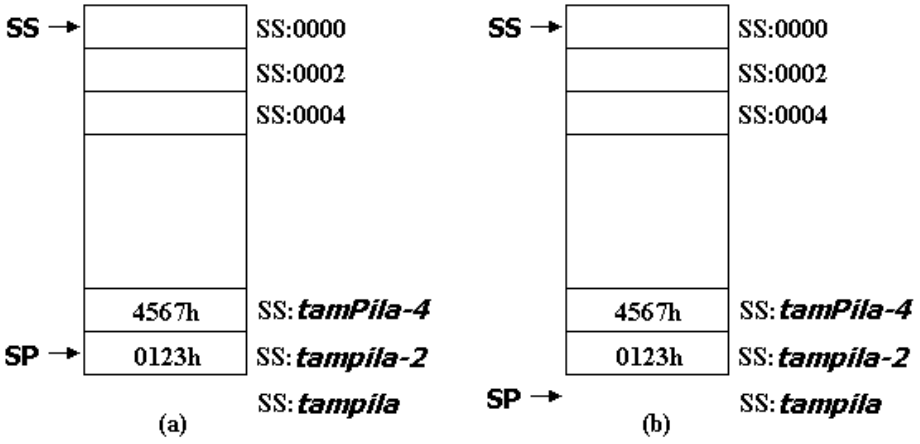

Figura 6-3

popf

Extrae un valor del tope de la pila del programa y lo almacena en registro de banderas.

Sintaxis:

popf

La instrucción **popf** extrae una palabra del tope de la pila y la inserta en el registro de banderas de la misma manera en que trabaja la instrucción **pop**. Normalmente se hace esto para extraer el estado de las banderas almacenado previamente por una instrucción **pushf**, o para fijar las banderas a nuevos valores.

La instrucción **popf** afecta todas las banderas.

Procedimientos

Como ya se mencionó anteriormente, un procedimiento es una colección de instrucciones que realizan una tarea específica: Sumar dos valores, desplegar un carácter en la pantalla, leer un carácter del teclado, inicializar un puerto, etc.

Dependiendo de su extensión y complejidad, un programa puede contener uno, algunos o inclusive cientos de procedimientos. Para emplear un procedimiento en un programa se requiere definir el procedimiento y llamarlo. Al definir a un procedimiento escribimos las instrucciones que contiene. Al llamar al procedimiento transferimos el control del flujo del programa al procedimiento para que sus instrucciones se ejecuten.

Definición de un procedimiento

La sintaxis de la definición de un procedimiento es la siguiente:

proc *nomProc*
proposición
[*proposición*]
...
endp [*nomProc*]

Las directivas **proc** y **endp** marcan el inicio y el final del procedimiento. No generan código.

nomProc es el nombre del procedimiento y etiqueta la primera instrucción del procedimiento.

Al menos una de las proposiciones de un procedimiento es la instrucción **ret**, la cual se describe más adelante.

Llamada a un procedimiento

La llamada a un procedimiento normalmente tiene la siguiente forma:

call *nomProc*

La instrucción **call** se describe más adelante. *nomProc* es el nombre que se le dio al procedimiento al definirlo.

Ejemplo sobre procedimientos

El siguiente programa suma dos variables de tipo palabra doble y guarda el resultado en una variable de tipo palabra doble. Este programa utiliza un procedimiento para efectuar la suma.

```
;*********************************************************
; SUM2DW_P.ASM
;
; Este programa suma dos variables de tipo palabra doble
; y guarda el resultado en una variable de tipo palabra
; doble. Este programa utiliza un procedimiento para
; efectuar la suma.
;
; El pseudocódigo de este programa es:
;
;    DX:AX = dato1
;    CX:BX = dato2
;
;    suma2dw()
;
;    resul = DX:AX
;*********************************************************

;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small
        stack   256

;****** VARIABLES DEL PROGRAMA ****************************

        dataseg
codsal  db      0
dato1   dd      ?
dato2   dd      ?
resul   dd      ?

;****** CÓDIGO DEL PROGRAMA *******************************

        codeseg
inicio:
        mov     ax, @data               ; Inicializa el
        mov     ds, ax                  ; segmento de datos

        mov     ax, [word dato1]        ; DX:AX = dato1
        mov     dx, [word dato1+2]
        mov     bx, [word dato2]        ; CX:BX = dato2
        mov     cx, [word dato2+2]
```

```
; Llama al procedimiento para efectuar la suma
        call    suma2dw                 ; suma2dw()

        mov     [word resul], ax        ; resul = DX:AX
        mov     [word resul+2], dx

salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h


;****** PROCEDIMIENTOS ************************************

;********************************************************
; SUMA2DW
;
; Este procedimiento suma dos variables de tipo
; palabra doble.
;
; Parámetros:
;
;       DX:AX = Primer sumando
;       CX:BX = Segundo sumando
;
; Regresa:
;
;       DX:AX = Suma
;
; El pseudocódigo de este procedimiento es:
;
;   DX:AX += CX:BX
;********************************************************
proc    suma2dw
        add     ax, bx                  ; DX:AX += CX:BX
        adc     dx, cx

        ret                             ; Regresa del
                                        ; procedimiento
endp    suma2dw

;****** CÓDIGO DE TERMINACIÓN *****************************

        end     inicio
```

Mecánica de la llamada a un procedimiento

Para implementar la llamada a un procedimiento y el regreso de éste, se tiene el siguiente mecanismo:

1. Cuando el programa lee de memoria y decodifica la instrucción que corresponde a la llamada a un procedimiento, el registro apuntador de instrucciones **IP** tiene el desplazamiento de la siguiente instrucción a ejecutar. Por ejemplo, en el listado parcial del archivo SUM2DW_P.LST que se muestra a continuación se puede ver que el desplazamiento de la siguiente instrucción a la llamada al procedimiento **suma2dw** es de 0017h.

```
31
32 0005  A1 0001r              mov     ax, [word dato1]     ; DX:AX = dato1
33 0008  8B 16 0003r           mov     dx, [word dato1+2]
34 000C  8B 1E 0005r           mov     bx, [word dato2]     ; CX:BX = dato2
35 0010  8B 0E 0007r           mov     cx, [word dato2+2]
36
37 0014  E8 000E               call    suma2dw              ; Llama al procedimiento
38                                                          ; para efectuar la suma
39
40 0017  A3 0009r              mov     [word resul], ax     ; resul = DX:AX
41 001A  89 16 000Br           mov     [word resul+2], dx


64 0025                 proc    suma2dw
65 0025  03 C3                  add     ax, bx               ; DX:AX += CX:BX
66 0027  13 D1                  adc     dx, cx
67
68 0029  C3                     ret                          ; Regresa del
69                                                          ; procedimiento
70 002A                 endp    suma2dw
```

2. La instrucción **call** inserta el valor de este desplazamiento en la pila del programa y carga en el registro apuntador de instrucciones el valor de la dirección etiquetada con el nombre del procedimiento. En este ejemplo cargaría a **IP** con el valor de 0025h. Esto hace que el control de flujo del programa se transfiera a la primera instrucción del procedimiento.

3. Al ejecutar una instrucción **ret**, ésta extrae del tope de la pila el valor previamente almacenado por la instrucción **call** y lo carga en el registro apuntador de instrucciones. Como este valor es el desplazamiento de la instrucción que va después de la llamada al procedimiento, ésta será la siguiente instrucción a ejecutarse. Esto es lo que se conoce como el regreso del procedimiento. En este ejemplo **IP** se carga con el valor de 0017h.

La descripción de las instrucciones **call** y **ret**, se muestran a continuación.

call *destino*

Llama a un procedimiento.

Sintaxis:

call *etiqueta*
call *regW|memW*
call *memDW*

Si el procedimiento a llamar se encuentra en el mismo segmento en el que se encuentra su llamada, la instrucción **call** inserta en la pila el desplazamiento de la siguiente instrucción después de la llamada al procedimiento y luego carga en el registro apuntador de instrucciones el desplazamiento dado por *destino.* Si el procedimiento está en un segmento diferente al en que está su llamada, la instrucción **call** inserta en la pila el segmento:desplazamiento de la siguiente instrucción después de la llamada al procedimiento y luego carga en el registro apuntador de instrucciones y en el registro de segmento de código el segmento:desplazamiento dado por *destino.*

En la mayoría de los programas, *destino* es una etiqueta que marca el inicio del procedimiento. Sin embargo también puede ser un registro o localidad de memoria que contenga la dirección del procedimiento.

ret

Regresa de un procedimiento

Sintaxis:

ret

Si el procedimiento del que se va a regresar se encuentra en el mismo segmento en el que se encuentra su llamada, la instrucción **ret** extrae de la pila el desplazamiento de la siguiente instrucción después de la llamada al procedimiento, el cual fue insertado en la pila por la llamada al procedimiento y lo carga en el registro apuntador de instrucciones **IP**. Si el procedimiento está en un segmento diferente al en que está su llamada, la instrucción **ret** extrae de la pila el segmento:desplazamiento de la siguiente instrucción después de la llamada al procedimiento y lo carga en los registros de segmento de código **CS** y apuntador de instrucciones **IP**, respectivamente.

Consideraciones sobre el diseño de un procedimiento

Un procedimiento bien diseñado debe llenar los siguientes requisitos:

- Debe hacer una sola tarea.
- Debe ser tan pequeño como sea posible y tan largo como sea necesario. Su listado no debiera de exceder una o dos páginas.

- Debe contener un comentario describiendo su propósito, sus datos de entrada y de salida.

- Debe entenderse por sí solo sin necesidad de saber que hace el programa completo.

- El comportamiento interno del procedimiento debe ser transparente para el usuario. Esto se llama **encapsulamiento de la información**. El usuario sólo debe saber cómo llamar al procedimiento, esto es, qué datos se le deben suministrar y cómo regresa el resultado. A esto se le llama la **interfase del procedimiento**.

  Para lograr esa transparencia se requiere que:

  - Un procedimiento, desde el punto de vista del usuario, funcione como lo haría una instrucción del microprocesador. Por lo tanto un procedimiento sólo puede alterar los registros en los que recibe los datos, los registros en los que regresa el resultado o el registro de banderas.

  - Un procedimiento tampoco debe utilizar variables globales ni para recibir datos o regresar un resultado, ni para almacenar temporalmente resultados intermedios.

Uso de los registros en un procedimiento

Un procedimiento puede utilizar los registros de propósito general de tres formas diferentes:

- Para recibir valores o direcciones del código que lo llama.

- Para regresar valores o direcciones al código que lo llama.

- Como variables locales del procedimiento o como registros de trabajo de las instrucciones del procedimiento.

En la sección anterior se mencionó que para que un procedimiento sea transparente al usuario, sólo debe modificar los registros que emplea para recibir datos o regresar resultados. ¿Cómo puede entonces emplear los registros como variables locales o registros de trabajo sin modificar su valor? La respuesta es que, si un procedimiento emplea un registro como variable local o registro de trabajo, debe guardar el valor original de ese registro en la pila del programa al entrar al procedimiento y recuperar su valor antes de salir del procedimiento.

Ejemplo sobre procedimientos que usan registros como variables locales o registros de trabajo

El siguiente programa intercambia los nibles más significativo y menos significativo de una variable de un byte. El programa utiliza un procedimiento para intercambiar los nibles.

Como el procedimiento **swapnibl** empleado para intercambiar los nibles requiere usar el registro **CX** como registro de trabajo de la instrucción **rol** el valor inicial de ese registro al entrar al procedimiento se almacena en la pila con una instrucción **push**. Antes de regresar del procedimiento, se restaura el valor original del registro **CX** mediante la instrucción **pop**.

```
;*********************************************************
; SWAPNIBL2.ASM
;
; Este programa intercambia los MSN y LSN de una variable
; de un byte. El resultado queda en la misma variable.
; Este programa utiliza un procedimiento para intercambiar
; los nibles.
;
; El pseudocódigo de este programa es:
;
;    AL = dato
;    swapnibl()
;    dato = AL
;*********************************************************

;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small
        stack   256

;****** VARIABLES DEL PROGRAMA ****************************

        dataseg
codsal  db      0
dato    db      ?

;****** CÓDIGO DEL PROGRAMA *******************************

        codeseg
inicio:
        mov     ax, @data       ; Inicializa el
        mov     ds, ax          ; segmento de datos

        mov     al, [dato]      ; AL = dato

; Llama al procedimiento para intercambiar los nibles
        call    swapnibl        ; swapnibl()

        mov     [dato], al      ; dato = AL

salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h
```

```
;****** PROCEDIMIENTOS ************************************

;*************************************************************
; SWAPNIBL
;
; Este procedimiento intercambia los MSN y LSN de una
; variable de un byte.
;
; Parámetro:
;
;       AL = dato
;
; Regresa:
;
;       AL = dato con los nibles intercambiados.
;
; El pseudocódigo de este procedimiento es:
;
;     CL = 4
;     rol al, 4
;*************************************************************
proc    swapnibl
        push    cx              ; Preserva CX

        mov     cl, 4
        rol     al, cl

        pop     cx              ; Recupera CX
        ret
endp    swapnibl

;****** CÓDIGO DE TERMINACIÓN *****************************

        end     inicio
```

Etiquetas locales a un procedimiento

Los identificadores que hemos empleado para etiquetar las direcciones de las instrucciones son etiquetas globales. Una **etiqueta global** es conocida en todo el programa, incluyendo en los procedimientos. Si un procedimiento definiera una etiqueta global, ésta también sería conocida por todo el programa. Como las etiquetas globales deben ser únicas, los nombres de las etiquetas de un procedimiento deben ser diferentes entre sí y diferentes a las de otros procedimientos y de las del resto del programa. Lo anterior viola el principio transparencia mencionado en la sección pasada ya que el programador debería conocer los nombres de las etiquetas empleadas en los diferentes procedimientos para no repetirlos.

Para evitar este problema Turbo Assembler nos permite definir etiquetas locales. El ámbito de una etiqueta local se extiende desde su definición hacia adelante y hacia atrás hasta la siguiente etiqueta

no local. Como las etiquetas definidas con la directiva **proc** son globales, los procedimientos están limitados por etiquetas globales, la del inicio del procedimiento y la de del inicio del siguiente procedimiento. Por lo tanto, las etiquetas locales son visibles sólo dentro del código del procedimiento y podemos reutilizarlas en otro lado sin problema.

La sintaxis de una etiqueta local es similar a la de las etiquetas globales sólo que empieza con dos arrobas, por ejemplo @@while, @@endwhi, etc.

Ejemplo sobre procedimientos que usan etiquetas locales

El siguiente programa suma los números enteros desde 1 hasta nfinal. El programa utiliza un procedimiento para obtener la suma.

```
;*********************************************************
; SERIE3.ASM
;
; Este programa suma los números enteros de 1 hasta nfinal
; Versión que utiliza un procedimiento para obtener la
; suma.
;
; El pseudocódigo de este programa es:
;
;   AX = nfinal
;
;   serie()
;
;   suma = AX
;*********************************************************

;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small
        stack   256

;****** VARIABLES DEL PROGRAMA ****************************

        dataseg
codsal  db      0
nfinal  dw      ?
suma    dw      ?

;****** CÓDIGO DEL PROGRAMA *******************************

        codeseg
inicio:
        mov     ax, @data       ; Inicializa el
        mov     ds, ax          ; segmento de datos
```

```
        mov     ax, [nfinal]    ; AX = nfinal

        call    serie

        mov     [suma], ax      ; suma = AX
salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h

;****** PROCEDIMIENTOS ************************************

;*******************************************************
; SERIE
;
; Este procedimiento suma los números de 1 hasta nfinal
;
; Parámetro:
;
;        AX = nfinal
;
; Regresa:
;
;        AX = suma
;
; El pseudocódigo de este procedimiento es:
;
;   CX = nFinal
;   if(CX == 0) goto enddo
;   AX = 0
;
;   do
;   {
;      AX+= CX
;   }
;   while(--CX > 0)
;*******************************************************
proc    serie
        push    cx              ; Preserva CX
        mov     cx, ax          ; CX = nfinal

        jcxz    @@enddo         ; if(CX == 0) goto enddo
        xor     ax, ax          ; AX = 0

@@do:                           ; do {
        add     ax, cx          ;    AX += CX
        loop    @@do            ; } while(--CX > 0)
@@enddo:

        pop     cx              ; Recupera CX
        ret
endp    serie
```

```
;****** CÓDIGO DE TERMINACIÓN ****************************

        end     inicio
```

Ejercicios sobre procedimientos

1. Cree un procedimiento que obtenga el máximo común divisor de dos números no signados de tipo palabra. El procedimiento recibe los dos números en los registros **AX** y **DX** y regresa el máximo común divisor de los números en **AX.**

2. Cree un procedimiento que obtenga el mínimo común múltiplo de dos números no signados de tipo palabra. El procedimiento recibe los dos números en los registros **AX** y **DX** y regresa el mínimo común múltiplo de los números en **DX:AX.**

Modo de direccionamiento base

En este modo de direccionamiento el cálculo de la dirección efectiva del dato emplea uno de los registros **BX** o **BP**. Las referencias a **BX** son desplazamientos con respecto al registro de segmento de datos **DS**, mientras que las referencias a **BP** son desplazamientos con respecto al registro de segmento de pila **SS**.

El modo de direccionamiento base tiene la siguiente sintaxis:

[bx+*n*]
[bx-*n*]
[bp+*n*]
[bp-*n*]

En los dos primeros casos el desplazamiento del dato con respecto a **DS** está dado por el valor de **BX** más o menos *n* bytes. En los dos últimos casos el desplazamiento del dato con respecto a **SS** está dado por el valor de **BP** más o menos *n* bytes.

El direccionamiento base utilizando el registro **BX** se emplea normalmente para localizar campos dentro de una estructura de datos. Por ejemplo:

```
        mov     bx, offset dato ; Apunta a la estructura
                                ; dato
        mov     ax, [bx+5]      ; Obtén el valor que se
                                ; encuentra a 5 bytes
                                ; del inicio de dato
        inc     [word bx+3]     ; Incrementa el valor de
                                ; tipo palabra que se
                                ; encuentra a 3 bytes del
                                ; inicio de dato
```

En la última instrucción del ejemplo anterior fue necesario indicar que la localidad de memoria apuntada por bx+3 es una localidad de tipo palabra. De otra manera la instrucción **inc** no podría determinar si la localidad de memoria era de un byte o de una palabra.

El direccionamiento base utilizando el registro **BP** se emplea normalmente para referenciar a variable locales en la pila del programa como se verá en la siguiente sección.

Variables locales en la pila

Los procedimientos en ensamblador utilizan por lo general los registros de propósito general para variables locales. En la mayoría de los casos los registros con que disponemos: **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, son suficientes para nuestras necesidades de almacenamiento temporal. Sin embargo si llegáramos a necesitar de más espacio de almacenamiento temporal podemos emplear el mecanismo usado por los lenguajes de alto nivel para ubicar las variables locales de un procedimiento. Los lenguajes de alto nivel crean a las variables locales de un procedimiento en la pila del programa.

El mecanismo empleado para crear las variables en la pila es el siguiente:

1. En la figura 6-4, se muestra el estado de la pila inmediatamente después de entrar al procedimiento. Note que en la localidad apuntada por el tope de la pila se encuentra la dirección de regreso del procedimiento *dirRegProc*.

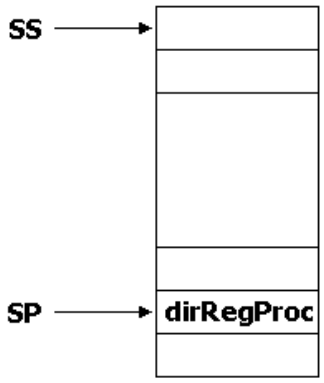

Figura 6-4

2. Una vez dentro del procedimiento, lo primero que se hace es preservar el valor del registro apuntador de base, **BP**, insertándolo en la pila. A continuación se hace que **BP** apunte al tope de la pila, ya que a partir de esa posición se ubicarán las variables locales. Las instrucciones para obtener lo anterior son:

```
push    bp
mov     bp, sp
```

El estado de la pila, después de estas operaciones se muestra en la figura 6-5.

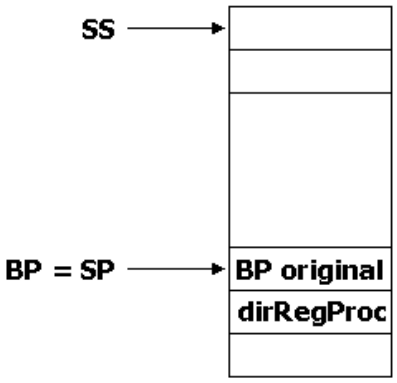

Figura 6-5

3. Se crean las variables locales decrementando **SP** en el número de bytes requeridos por las variables. Por ejemplo, para crear dos variables locales de tipo palabra, la instrucción sería:

```
sub sp, 4                ; 2 variables de 2 bytes
                         ; c/u = 4 bytes
```

El estado de la pila después de la instrucción se muestra en la figura 6-6. Las variables locales se identifican en la figura con los nombres: **varLoc1** y **varLoc2.** Sin embargo el acceso a las variables es mediante su dirección las cuales son: **BP**-2 y **BP**-4, respectivamente.

El mecanismo empleado para destruir las variables es el siguiente:

1. Se incrementa **SP** en el número de bytes requeridos por las variables locales. Esto se puede hacer mediante la instrucción:

```
mov     sp, bp           ; Equivale en este caso
                         ; a: add sp, 4
```

Esto nos deja la pila como en la figura 6-5.

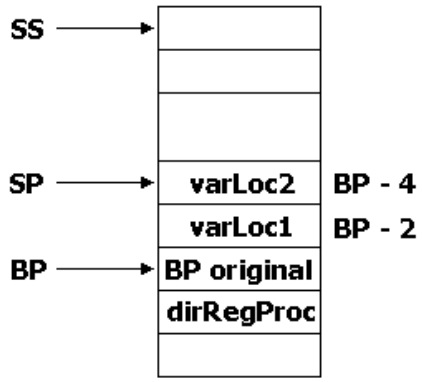

Figura 6-6

2. Se recupera el valor de BP de la pila, haciendo que **SP** apunte de nuevo a la dirección de regreso del procedimiento, mediante la instrucción:

```
pop     bp
```

   El estado final de la pila se muestra en la figura 6-4.

Ejemplo sobre variables locales en la pila

El siguiente procedimiento reduce una fracción a su expresión más simple. Por ejemplo el número 12/60 se reduce a 1/5, dividiendo el numerador y denominador entre el máximo común divisor. El procedimiento llama al procedimiento desarrollado en el ejercicio 1 de la sección anterior para encontrar el máximo común divisor.

```
;*********************************************************
; REDFRAC
;
; Este procedimiento reduce una fracción a su mínima
; expresión
;
; Parámetros:
;
;       AX = numerador
;       DX = denominador
;
; Regresa:
;
;       AX = numerador reducido
;       DX = denominador reducido
;
; El pseudocódigo para este procedimiento es:
;
```

```
;   BX = mcd(numerador, denominador)
;
;   numerador /= BX
;   denominador /= BX
;
;*******************************************************
proc    redfrac
        push    bp              ; Preserva BP
        mov     bp, sp          ; Crea variables
        sub     sp, 4           ; locales en la pila
        push    bx              ; Preserva BX

;num = [BP-2], den = [BP-4]
        mov     [bp-2], ax      ; num = numerador
        mov     [bp-4], dx      ; den = denominador

        call    mcd             ; BX = mcd(numerador,
                                ;          denominador)
        mov     bx, ax
        xor     dx, dx          ; num = numerador/BX
        mov     ax, [bp-2]
        div     bx
        mov     [bp-2], ax

        mov     ax, [bp-4]      ; DX = denominador/BX
        div     bx
        mov     dx, ax
        mov     ax, [bp-2]      ; AX = num

        pop     bx              ; Restaura BX
        mov     sp, bp          ; Elimina variables locales
        pop     bp              ; Restaura BP
        ret
endp    redfrac
```

Variables locales en la pila usando la directiva local

Turbo Assembler permite simplificar un poco el manejo de las variables locales permitiendo emplear identificadores locales para referenciar a las variables en lugar de emplear el direccionamiento base. Para ello se emplea la directiva **local**. La directiva **local** le asigna a cada variable local creada en la pila del programa un identificador local. La sintaxis de esta directiva es:

local *nomVar*: *tipo*[, *nomVar*: *tipo*]... [**=** *tamVarsLoc*]

donde *nomVar* es el nombre de la variable y *tipo* es su tipo: **byte**, **word**, **dword**, **qword**, etc. *tamVarsLoc* es un símbolo que toma el valor del número de bytes ocupado por las variables locales. Una vez que se ha empleado la directiva **local** se puede hacer referencia a las variables locales mediante sus nombres como si fueran variables globales.

Ejemplo sobre variables locales en la pila usando la directiva local

El siguiente procedimiento es una modificación del ejemplo anterior en el que se utiliza la directiva **local** para darle nombres locales a las variables locales creadas en la pila.

```
;*********************************************************
; REDFRAC2
;
; Este procedimiento reduce una fracción a su mínima
; expresión
;
; Parámetros:
;
;       AX = numerador
;       DX = denominador
;
; Regresa:
;
;       AX = numerador reducido
;       DX = denominador reducido
;
; El pseudocódigo para este procedimiento es:
;
;   numerador /= BX
;   denominador /= BX
;*********************************************************
proc    redfrac
; Declara las variables locales en la pila
        local   num: word, den: word = tamVarsLoc

        push    bp              ; Preserva BP
        mov     bp, sp          ; Crea variables
        sub     sp, tamVarsLoc  ; locales en la pila
        push    bx              ; Preserva BX

        mov     [num], ax       ; num = numerador
        mov     [den], dx       ; den = denominador

        call    mcd             ; BX = mcd(numerador,
        mov     bx, ax          ;          denominador)

        xor     dx, dx          ; num = numerador/BX
        mov     ax, [num]
        div     bx
        mov     [num], ax

        mov     ax, [den]       ; DX = denominador/BX
        div     bx

        mov     dx, ax
        mov     ax, [num]       ; AX = num
```

```
        pop     bx              ; Restaura BX
        mov     sp, bp          ; Elimina variables
                                ; locales

        pop     bp              ; Restaura BP
        ret
endp    redfrac
```

Ejercicios sobre variables locales en la pila

1. Modifique el procedimiento que encuentra el mínimo común múltiplo de la sección anterior para que el valor que se almacena en los registros **SI:DI** se guarde en una variable local en la pila. No utilice la directiva **local**

2. Modifique el procedimiento del ejercicio anterior para que emplee la directiva **local**.

Programación Modular

En los programas en ensamblador hechos hasta ahora, todo el código del programa reside en un sólo archivo. Sin embargo es posible y en muchos casos deseable dividir en código de un programa en varios archivos llamados módulos. Hay varias ventajas de emplear esta técnica llamada programación modular:

- Hay un límite en el tamaño de un archivo que un programa ensamblador puede manejar. Si el código del programa crece mucho, al dividirlo en módulos, tendremos archivos de menor tamaño. Como cada módulo puede ensamblarse por separado, esta técnica nos permite ensamblar programas más grandes.

- El proceso de ensamblado de un módulo es menor que el de un programa completo. Cuando se está en la etapa de construcción / depuración de un programa, por lo general uno escribe y depura un procedimiento a la vez, por lo que sólo se está haciendo cambios a un módulo, por lo que sólo ese módulo requiere de volver a ensamblarlo.

- Para obtener un programa ejecutable, los códigos objeto obtenidos al ensamblar cada módulo deben ligarse. Normalmente, cada módulo contiene un conjunto de procedimientos relacionados. Estos procedimientos pueden emplearse para formar diferentes programas, simplemente ligándolos con otros módulos.

- Podemos utilizar otros módulos escritos por terceros para resolver problemas específicos.

- Podemos escribir programas escritos en lenguaje de alto nivel y que utilicen módulos escritos en ensamblador.

Cada módulo en ensamblador puede contener declaraciones de constantes, declaraciones de variables y definiciones de procedimientos. La estructura de cada módulo es similar a la estructura de un programa sin módulos con las siguientes diferencias:

- Sólo uno de los módulos contiene las instrucciones que inicializan el segmento de datos y las instrucciones que terminan el programa y regresan el control al sistema operativo. A este módulo se le conoce como el módulo principal.

- Todos los módulos tienen código de inicio con las directivas: **ideal**, **dosseg**, **model**, pero sólo el módulo principal tiene la directiva **stack**.

- Todos los módulos tiene la directiva **end** que indica el final del código de ese módulo. Sin embargo sólo en el módulo principal la directiva **end** va seguida de la etiqueta que apunta al punto de entrada del programa. En los otros módulos la directiva **end** aparece por sí sola.

- Las constantes, variables y procedimientos declaradas y definidos en un módulo pueden ser conocidas sólo en el módulo donde fueron declaradas o pueden ser conocidas en otros módulos también. Los símbolos de las constantes, variables y procedimientos de un módulo que deseamos que se conozcan en otros módulos deben de exportarse utilizando la directiva **public**. La sintaxis de la directiva public es:

  ```
  public nomSimb[, nomSimb] ...
  ```

  donde *nomSimb* es el nombre de la constante, variable o procedimiento cuyo símbolo se va exportar. Sólo los símbolos de las constantes numéricas declaradas con la directiva = pueden exportarse. Los símbolos de las constantes declaradas con la directiva **equ** no pueden exportarse.

- Los símbolos de las constantes, variables y procedimientos declaradas y definidos en otro módulo y que fueron exportados con la directiva **public** tienen que importarse en el módulo en que se desea que sean conocidos también. Para importar un símbolo se utiliza la directiva **extrn**, cuya sintaxis es:

  ```
  extrn nomSimb: tipo[, nomSimb: tipo] ...
  ```

  donde *nomSimb* es el nombre de la constante, variable o procedimiento cuyo símbolo se va importar.

  tipo especifica el tipo de símbolo. Para los símbolos de las constantes numéricas declaradas con la directiva = el tipo es **abs**; para los símbolos de las variables a importar los tipos son: **byte**, **word**, **dword**, **qword**, etc. y para los procedimientos el tipo es **proc**.

En el siguiente ejemplo se muestra parte del código de un programa formado por dos módulos: MODULO.ASM y PRINCIPA.ASM. Este último contiene el módulo principal.

```
;*********************************************************
; MÓDULO.ASM
;
; Este módulo exporta algunos de sus símbolos e importa
; algunos símbolos del módulo con el programa principal.
;*********************************************************

;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small

;****** DECLARACIÓN DE CONSTANTES SIMBÓLICAS **************

        extrn   cte4: abs       ; importa cte4
        public  cte2            ; exporta cte2

cte1   equ    10               ; constante local
cte2   =      20               ; constante global

;****** VARIABLES DEL MÓDULO ******************************

        dataseg

        extrn   dato4: byte     ; importa dato4
        public  dato2           ; exporta dato2

dato1   db      ?               ; variable local
dato2   dw      ?               ; variable global

;****** CÓDIGO DEL MÓDULO *********************************

        codeseg

        extrn   proc4: proc     ; importa proc4
        public  proc2           ; exporta proc2

;****** PROCEDIMIENTOS ************************************

;*********************************************************
; PROC1
;
; Este procedimiento es local a este módulo.
;*********************************************************
proc    proc1
        ...
        ret
endp    proc1
```

```
;*********************************************************
; PROC2
;
; Este procedimiento es global.
;*********************************************************
proc    proc2
        ...
        ret
endp    proc2

;****** CÓDIGO DE TERMINACIÓN ***************************

        end
```

```
;*********************************************************
; PRINCIPA.ASM
;
; Este módulo que contiene el programa principal, también
; exporta algunos de sus símbolos e importa algunos
; símbolos del otro módulo.
;*********************************************************

;****** CÓDIGO DE INICIO *********************************

        ideal
        dosseg
        model   small
        stack   256

;****** DECLARACIÓN DE CONSTANTES SIMBÓLICAS **************

        extrn   cte2: abs       ; importa cte4
        public  cte4            ; exporta cte2

cte3   equ    10               ; constante local
cte4   =      20               ; constante global

;****** VARIABLES DEL MÓDULO *****************************

        dataseg

        extrn   dato2: word     ; importa dato4
        public  dato4           ; exporta dato2

dato3   db      ?               ; variable local
dato4   db      ?               ; variable global

;****** CÓDIGO DEL MÓDULO ********************************

        codeseg

        extrn   proc2: proc     ; importa proc2
```

```
        public  proc4           ; exporta proc4

inicio:
        mov     ax, @data       ; Inicializa el segmento
        mov     ds, ax          ; de datos
        ...

salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h

;****** PROCEDIMIENTOS ************************************

;*******************************************************
; PROC3
;
; Este procedimiento es local a este módulo.
;*******************************************************
proc    proc3
        ...
        ret
endp    proc3

;*******************************************************
; PROC4
;
; Este procedimiento es global.
;*******************************************************
proc    proc4
        ...
        ret
endp    proc4

;****** CÓDIGO DE TERMINACIÓN *****************************

        end     inicio
```

Macros

Una **macro** es un conjunto de instrucciones asociadas a un identificador: el nombre de la macro. Este conjunto de instrucciones es invocado como una sola instrucción o **macroinstrucción**. Normalmente las instrucciones de una macro se repiten varias veces en un programa o aparecen frecuentemente en los programas. Para emplear una macro en un programa debemos de hacer dos cosas: Definir la macro e invocar la macro.

La **definición de una macro** establece el nombre al que se asocia la macro, el número y nombre de sus parámetros formales y qué instrucciones contiene la macroinstrucción. La sintaxis de la definición de una macro es la siguiente:

```
macro nomMacro [parForm[, parForm]...]
        proposición
       [proposición]
       ...
endm [nomMacro]
```

donde las directivas **macro** y **endm** marcan el inicio y el final de la definición de la macro. No generan código.

nomMacro es el nombre de la macro y se emplea al invocar la macro.

parForm es cada uno de los parámetros formales de la macro. Los parámetros permiten que una misma macro opere sobre datos distintos.

proposición son cada una de las instrucciones que forman la macroinstrucción.

Aunque la definición de una macro puede ir en cualquier parte de un programa, el lugar más recomendable para su localización es al principio de un archivo, antes de los segmentos de datos y de código.

Al encontrar una **invocación de una macro** el macroensamblador, Turbo Assembler por ejemplo, substituye la línea con la invocación por las proposiciones que contiene la definición de la macro. Este proceso de substitución se conoce **expansión de la macro**. La sintaxis de la invocación de la macro es similar a cualquier instrucción:

```
nomMacro [parReal[, parReal]...]
```

donde cada *parReal* es conocido como un **parámetro real** de la macro. Al expandirse la macro cada una de las ocurrencias de un parámetro formal en la definición de la macro se substituye por su correspondiente parámetro real.

Los parámetros reales pueden ser símbolos, mnemónicos, directivas, palabras reservadas, expresiones y números.

Ejemplo sobre macros

El siguiente programa ejecuta todas las rotaciones de un bit con una variable de tipo palabra doble. Las operaciones de rotación se implementan mediante macros. El parámetro formal **dest** de cada macro representa la localidad de memoria que contiene el dato a rotar. Al invocar a cada una de las macros el parámetro real de cada macro es el nombre de la variable cuyo dato se va a rotar.

```
;*********************************************************
; ROTACIO2.ASM
;
```

```
; Este programa ejecuta todas las rotaciones de un bit con
; una variable de tipo palabra doble. Las operaciones de
; rotación se implementan mediante macros que rotan el
; contenido de la localidad de memoria dada por el
; parámetro de la macro un bit.
;*********************************************************
;
;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small
        stack   256

;****** MACROS ********************************************

;*********************************************************
;
; RCLDW1
;
; Esta macro rota el contenido de dest un bit a la
; izquierda a través de la bandera de acarreo.
;*********************************************************
;
macro   rcldw1  dest
        rcl     [word dest], 1          ;; rcl dest, 1
        rcl     [word dest+2], 1
endm    rcldw1

;*********************************************************
;
; RCRDW1
;
; Esta macro rota el contenido de dest un bit a la
; derecha a través de la bandera de acarreo.
;*********************************************************
;
macro   rcrdw1  dest
        rcr     [word dest+2], 1        ;; rcr dest, 1
        rcr     [word dest], 1
endm    rcrdw1

;*********************************************************
;
; ROLDW1
;
; Esta macro rota el contenido de dest un bit a la izquierda
;*********************************************************
;
macro   roldw1  dest
        push    bx                      ;; Preserva BX
        mov     bx, [word dest+2]       ;; rol dest, 1
        shl     bx, 1
        rcl     [word dest], 1
        rcl     [word dest+2], 1
        pop     bx                      ;; Restaura BX
endm    roldw1
```

```
;*********************************************************
; RORDW1
;
; Esta macro rota el contenido de dest un bit a la derecha.
;*********************************************************
macro   rordw1  dest
        push    bx                      ;; Preserva BX
        mov     bx, [word dest]         ;; ror dest, 1
        shr     bx, 1
        rcr     [word dest+2], 1
        rcr     [word dest], 1
        pop     bx                      ;; Restaura BX
endm    rordw1

;*** VARIABLES DEL PROGRAMA ******************************

        dataseg
codsal  db      0
r_rcl   dd      ?
r_rcr   dd      ?
r_rol   dd      ?
r_ror   dd      ?

;****** CÓDIGO DEL PROGRAMA ******************************

        codeseg
inicio:
        mov     ax, @data               ; Inicializa el
        mov     ds, ax                  ; segmento de datos

; Rotación a la izquierda a través de la bandera de acarreo.

        rcldw1  r_rcl                   ; rcl r_rcl, 1

; Rotación a la derecha a través de la bandera de acarreo
        rcrdw1  r_rcr                   ; rcr r_rcr, 1

; Rotación a la izquierda
        roldw1  r_rol                   ; rol r_rol, 1

; Rotación a la derecha
        rordw1  r_ror                   ; ror r_ror, 1

salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h

;****** CÓDIGO DE TERMINACIÓN *****************************

        end     inicio
```

En el ejemplo anterior podemos notar que los comentarios dentro de la definición de una macro empiezan con doble punto y coma. Esto se hace para que en el archivo con el listado del programa generado por el ensamblador no aparezcan los comentarios cada vez que se expande la macro.

También podemos notar que las macros **roldw1** y **rordw1** que implementan las rotaciones a la izquierda y a la derecha, respectivamente, utilizan el registro **BX** y por lo tanto se preserva su valor al inicio de la macro y se restaura al salir. Esto se hace con el fin de que el uso de la macro sea transparente para el usuario.

Ejercicio sobre macros

Escriba un programa que ejecute todos los corrimientos de un bit con una variable de tipo palabra doble. Las operaciones de corrimiento se implementan mediante macros. Cada macro tiene un parámetro formal **dest** que representa la localidad de memoria que contiene el dato a rotar.

Macros con etiquetas locales

Al igual que con los procedimientos, si una macro emplea identificadores para etiquetar las direcciones de instrucciones que componen la macro, estas etiquetas deben ser etiquetas locales, para evitar la duplicidad de etiquetas si la macro se invoca más de una vez en un programa. Para crear etiquetas locales en Turbo Assembler se emplea la directiva **local**. La directiva **local** en una macro hace que las etiquetas declaradas en la directiva tengan un alcance restringido a la macro. La sintaxis de esta directiva es:

local e*tiqueta* [, *etiqueta*]...

En una macro, la directiva local debe ir inmediatamente de la directiva macro. Pueden tenerse varias directivas local.

Ejemplo sobre macros con etiquetas locales

El programa siguiente encuentra el mayor de tres datos signados de tipo palabra. El programa utiliza la macro **mayor3** para encontrar el mayor de los tres datos. La macro tiene dos etiquetas locales: **sigcmp** y **fincmp**.

```
;*********************************************************
; MAYOR3_2.ASM
;
; Este programa encuentra el mayor de tres datos signados
; almacenados en variables de una palabra. El programa
; utiliza una macro para encontrar al mayor de los tres
; datos
;*********************************************************
```

```
;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small
        stack   256


;****** MACROS ********************************************

;*****************************************************
; MAYOR3
;
; Esta macro encuentra el mayor de tres datos signados
; dados por los parámetros dato1, dato2 y dato3. El mayor
; de los tres datos queda en el registro AX.
;*****************************************************
macro   mayor3  dato1, dato2, dato3
        local   sigcmp, fincmp

        mov     ax, [dato1]     ; AX = dato1
        cmp     ax, [dato2]     ; if(AX > dato2)
        jg      sigcmp          ;    goto sigcmp
        mov     ax, [dato2]     ; else AX = dato2
sigcmp:
        cmp     ax, [dato3]     ; if(AX > dato3)
        jg      fincmp          ;    goto fincmp
        mov     ax, [dato3]     ; else AX = dato3
fincmp:
endm    mayor3

;****** VARIABLES DEL PROGRAMA ****************************

        dataseg
codsal  db      0
dato1   dw      ?
dato2   dw      ?
dato3   dw      ?
mayor   dw      ?

;****** CÓDIGO DEL PROGRAMA *******************************

        codeseg
inicio:
        mov      ax, @data      ; Inicializa el
        mov      ds, ax         ; segmento de datos

        mayor3  dato1, dato2, dato3

        mov     [mayor], ax     ; mayor = AX

salir:
```

```
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h

;****** CÓDIGO DE TERMINACIÓN ****************************

        end     inicio
```

Ejercicio sobre macros con etiquetas locales

Escribe una macro que determine el mayor de dos datos signados de tipo palabra doble. La macro recibe los datos en parámetros y regresa el mayor de los dos datos en los registros **DX:AX**.

Directiva de repetición rept

Los macroensambladores del 8086 poseen varias directivas que permiten repetir instrucciones. Una de esas es la directiva **rept**. La directiva **rept** permite repetir un grupo de proposiciones varias veces. Su sintaxis es:

```
rept    expresión
        proposición
        [proposición]
        ...
endm
```

donde las directivas **rept** y **endm** delimitan la directiva de repetición **rept**.

expresión es una expresión constante entera y es el número de veces que se repiten las instrucciones dadas por *proposición*.

La directiva de repetición **rept** puede utilizarse por sí sola o dentro de macros para crear instrucciones que repitan instrucciones.

Ejemplo sobre macros que usan la directiva de repetición rept

El archivo siguiente muestra una modificación de las macros que rotan variables de tipo palabra doble para que en lugar de rotar las variables un solo bit, las roten varios bits. Las macros contienen la directiva de rotación **rept**. Las macros se encuentran en un archivo con el nombre ROTA_N.INC el cual puede incluirse en otros programas usando la directiva **include** la cual se verá en la siguiente sección. La extensión .INC no es obligatoria para los archivos de inclusión pero si es una práctica común.

```
;*********************************************************
; ROTA_N.INC
;
; Este archivo contiene una serie de macros que pueden
; incluirse en otros programas con la directiva include e
; implementan las operaciones de rotación del contenido de
; la localidad de memoria dada por el parámetro dest un
; número de bits dado por el parámetro cnta.
;*********************************************************


;*********************************************************
; RCLDWN
;
; Esta macro rota el contenido de dest cnta bits a la
; izquierda a través de la bandera de acarreo.
;*********************************************************
macro   rcldwn  dest, cnta
        rept    cnta
         rcl    [word dest], 1          ;; rcl dest, 1
         rcl    [word dest+2], 1
        endm
endm    rcldwn

;*********************************************************
; RCRDWN
;
; Esta macro rota el contenido de dest cnta bits a la
; derecha a través de la bandera de acarreo.
;*********************************************************
macro   rcrdwn  dest, cnta
        rept    cnta
         rcr    [word dest+2], 1        ;; rcr dest, 1
         rcr    [word dest], 1
        endm
endm    rcrdwn

;*********************************************************
; ROLDWN
;
; Esta macro rota el contenido de dest cnta bits a la
; izquierda.
;*********************************************************
macro   roldwn  dest, cnta
        push    bx                      ;; Preserva BX
        rept    cnta
         mov    bx, [word dest+2]       ;; rol dest, 1
         shl    bx, 1
         rcl    [word dest], 1
         rcl    [word dest+2], 1
        endm
        pop     bx                      ;; Restaura BX
endm    roldwn
```

```
;*********************************************************
; RORDWN
;
; Esta macro rota el contenido de dest cnta bits a la
; derecha.
;*********************************************************
macro   rordwn  dest, cnta
        push    bx                      ;; Preserva BX
        rept    cnta
         mov    bx, [word dest]         ;; ror dest, 1
         shr    bx, 1
         rcr    [word dest+2], 1
         rcr    [word dest], 1
        endm
        pop     bx                      ;; Restaura BX
endm    rordwn
```

Ejercicio sobre macros que usan la directiva de repetición rept

Modifica el programa de las macros que corren variables de tipo palabra doble para que en lugar de correr las variables un solo bit, las corran varios bits. Las macros deben usar la directiva de rotación **rept**.

Inclusión de archivos con la directiva include

La directiva **include** permite la inclusión de un archivo texto dentro de otro. La sintaxis de esta directiva es:

include "*nomArch*"

donde *nomArh* es el nombre del archivo cuyo contenido será substituido en lugar de la directiva **include**.

A continuación se muestra un ejemplo que ilustra el uso de la directiva **include** para incluir dentro del código el contenido del archivo ROTA_N.INC del ejemplo anterior. Este programa ejecuta todas las rotaciones de un número de bits con una variable de tipo palabra doble.

```
;*********************************************************
; ROTACIO3.ASM
;
; Este programa ejecuta todas las rotaciones de un número
; de bits con una variable de tipo palabra doble. Las
; operaciones de rotación están implementadas mediante
; macros y se encuentran en el archivo ROTA_N.INC. Este
```

```
; programa ilustra el uso de la directiva include.
;*********************************************************
;

;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small
        stack   256

;****** INCLUSIÓN DE ARCHIVOS *****************************

include "rota_n.inc"

;****** VARIABLES DEL PROGRAMA ****************************

        dataseg
codsal  db      0
r_rcl   dd      ?
r_rcr   dd      ?
r_rol   dd      ?
r_ror   dd      ?

;****** CÓDIGO DEL PROGRAMA *******************************

   codeseg
inicio:
        mov     ax, @data               ; Inicializa el
        mov     ds, ax                  ; segmento de datos

; Rotación a la izquierda a través de la bandera de acarreo
        rcldwn  r_rcl, 2                ; rcl r_rcl, 2

; Rotación a la derecha a través de la bandera de acarreo
        rcrdwn  r_rcr, 2                ; rcr r_rcr, 2

; Rotación a la izquierda
        roldwn  r_rol, 2                ; rol r_rol, 2

; Rotación a la derecha
        rordwn  r_ror, 2                ; ror r_ror, 2

salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h

;****** CÓDIGO DE TERMINACIÓN *****************************

        end     inicio
```

Bibliografía

1. Abel, Peter. Lenguaje Ensamblador y Programación para PC IBM y Compatibles. Tercera Edición. Prentice-Hall Hispanoamericana, S. A. México. 1996.

2. Borland Int. Turbo Assembler Reference Guide. Version 1. Borland International. Scotts Valley, CA. 1988.

3. Brey, Barry B. Los microprocesadores Intel: 8086/8088, 80186, 80286, 80386 y 80486. Arquitectura, programación e interfaces. Tercera Edición. Prentice-Hall Hispanoamericana, S. A. México. 1995.

4. Godfrey, J. Terry. Lenguaje Ensamblador para Microcomputadoras IBM para Principiantes y Avanzados. Prentice-Hall Hispanoamericana, S. A. México. 1991.

5. Hyde, Randall. The Art of Assembly Language Programming. Este libro se encuentra como una serie de documento PDF en el siguiente servidor FTP:

   ftp.cs.ucr.edu/pub/pc/ibmpcdir

6. Swan, Tom. Mastering Turbo Assembler. Hayden Books. Indiana, U.S.A. 1989.

Problemas

Escribe un programa que implemente las operaciones de suma, resta, multiplicación y división de números fraccionarios. Cada número fraccionario se representará mediante un par de números enteros (de tipo palabra). El programa constará de dos módulos.

El primer módulo llamado **ARITFRAC.ASM** contiene los siguientes procedimientos:

mcd.- que calcula el máximo común divisor de dos números enteros de tipo palabra. El procedimiento recibe los dos números en los registros AX y DX y regresa el máximo común divisor en el registro AX.

mcm.- que calcula el mínimo común múltiplo de dos números enteros de tipo palabra. El procedimiento recibe los dos números en los registros AX y DX y regresa el mínimo común múltiplo en los registros DX:AX.

redfrac.- que reduce una fracción a su mínima expresión. El procedimiento recibe los dos números que representan el numerador y el denominador en los registros AX y DX, respectivamente y regresa la fracción reducida en los mismos registros.

addfrac.- que suma dos números fraccionarios y regresa la suma como una fracción reducida a su mínima expresión. El procedimiento recibe los dos números fraccionarios en los registros AX y DX para el numerador y denominador del primer número y BX y CX para el numerador, denominador del segundo número. El procedimiento regresa el resultado en los registros AX y DX para el numerador y denominador, respectivamente.

subfrac.- que resta dos números fraccionarios y regresa la resta como una fracción reducida a su mínima expresión. El procedimiento recibe los dos números fraccionarios en los registros AX y DX para el numerador y denominador del primer número y BX y CX para el numerador, denominador del segundo número. El procedimiento regresa el resultado en los registros AX y DX para el numerador y denominador, respectivamente.

mulfrac.- que multiplica dos números fraccionarios y regresa el producto como una fracción reducida a su mínima expresión. El procedimiento recibe los dos números fraccionarios en los registros AX y DX para el numerador y denominador del primer número y BX y CX para el numerador, denominador del segundo número. El procedimiento regresa el resultado en los registros AX y DX para el numerador y denominador, respectivamente.

divfrac.- que divide dos números fraccionarios y regresa el cociente como una fracción reducida a su mínima expresión. El procedimiento recibe los dos números fraccionarios en los registros AX y DX para el numerador y denominador del primer número y BX y CX para el numerador, denominador del segundo número. El procedimiento regresa el resultado en los registros AX y DX para el numerador y denominador, respectivamente.

El segundo módulo llamado **DEMOFRAC.ASM** contiene un programa para probar los procedimientos del módulo anterior. En este módulo se declaran 8 variables: **numA**, **denA**, **numB**, **denB**, **numC**, **denC**, **numR**, **denR**, para almacenar cuatro números fraccionarios:

A = numA/denA

B = numB/denB

C = numC/denC

R = numR/denR

El programa deberá efectuar la siguiente operación:

R = ((A + B) * (C - A))/(B + C).

El programa ejecutable deberá llamarse **DEMOFRAC.EXE**