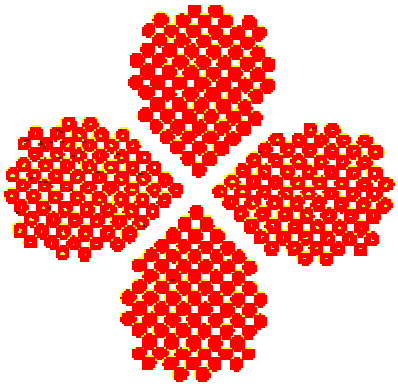
Digital II

Conceptos básicos sobre la programación en Assembler



Departamento de Sistemas e Informática

Escuela de Electrónica

Facultad de Ciencias Exactas Ingeniería y Agrimensura

Universidad Nacional de Rosario

Silvina Ferradal

2000

1. ORGANIZACIÓN DE LOS SEGMENTOS

1.1- **Segmentos Físicos**

Un segmento físico sólo puede comenzar en direcciones de memoria pares divisibles por 16, incluyendo la dirección 0. A estas direcciones se las denomina “párrafos” (paragraph). Se puede reconocer un párrafo ya que su dirección hexadecimal siempre termina con 0, como lo es en 10000h o 2EA70h. Los procesadores 8086/286 utilizan segmentos de un tamaño de 64K .

1.2- **Segmentos Lógicos**

Los segmentos lógicos contienen los tres componentes de un programa: código, datos y pila. Los registros de segmento CS, DS y SS contienen las direcciones de los segmetos de memoria físicos en donde residen los segmentos lógicos.

Se pueden definir segmetos de dos formas distintas: con **directivas simplificadas** o con **directivas completas** (también se pueden usar los dos tipos en el mismo programa).

Las directivas simplificadas guardan muchos de los detalles de la definición del segmento, mientras que las directivas completas requieren una sintaxis más compleja pero brindan un control más completo respecto de la forma en que el assembler genera los segmentos. Si se usan directivas completas en la definición de los segmentos, se debe escribir el código necesario para manejar todas las tareas que se hacían en forma automática con las directivas simplificadas.

1.3- **Uso de Directivas Simplificadas**

Las directivas simplificadas son: **.MODEL**, **.CODE**, **.CONST**, **.DATA**, **.DATA?**, **.FARDATA**, **.FARDATA?**, **.STACK**, **.STARTUP** y **.EXIT**. Los programas consisten en módulos formados por segmentos. Cada programa escrito en assembler tiene un módulo principal, en donde comienza la ejecución del programa. Este módulo principal puede contener segmentos de código, de datos o de pila que estén definidos con directivas simplificadas. Cualquier otro módulo adicional puede contener sólo segmentos de datos o de código. Cada módulo que usa directivas simplificadas debe comenzar con la directiva **.MODEL**.

El siguiente ejemplo muestra la estructura de un módulo principal usando directivas simplificadas:

```
.MODEL      small, c

.STACK                  ; Definición del segmento de pila

.DATA                   ; Comienzo del segmento de datos
   .
   .                    ; Lugar para las declaraciones de datos
   .

.CODE                   ; Comienzo del segmento de código

.STARTUP                ; Generación del código startup
   .
   .                    ; Lugar para las instrucciones
   .

.EXIT                   ; Generación del código exit

 END
```

Las directivas **.DATA** y **.CODE** no requieren de otra instrucción para indicar el fin del segmento ya que cierran el segmento precedente y abren uno nuevo en forma automática.La directiva **.STACK** abre y cierra el segmento de pila pero no cierra el segmento actual.
La directiva **END** cierra el último segmento e indica el fin del código fuente. Esta directiva debe ir siempre al final de cada módulo.

1.3.1- **Directiva .MODEL**

La directiva **.MODEL** define los atributos que afectan al módulo completo: modelo de memoria, ubicación de la pila, etc. Se debe colocar **.MODEL** en el programa antes que cualquier otra directiva simplificada. Su sintaxis es:

.MODEL modelodememoria , opcionesdelmodelo

El campo *modelodememoria* siempre es necesario y debe ir a continuación de la directiva **.MODEL**, mientras que el uso de *opcionesdelmodelo*, el cual define otros atributos, es opcional. Este último debe ir separado por comas. La siguiente lista resume los campos *modelodememoria* y *opcionesdelmodelo* (este último especifica la ubicación de la pila y el lenguaje):

Campo	Descripción
Modelo de memoria	**TINY**, **SMALL**, **COMPACT**, **MEDIUM**, **LARGE** o **HUGE** .
Lenguaje	**C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL** o **STDCALL**. Esta opción permite combinar (a través del *linker*) el programa en assembler con un módulo de otro lenguaje de alto nivel.
Ubicación de la pila	**NEARSTACK** o **FARSTACK**. Si se especifica **NEARSTACK** se grupa el segmento de la pila en un solo segmento físico (DGROUP) junto con los datos.Entonces SS coincide con DS. **FARSTACK** no agrupa la pila con DGROUP por lo que SS no coincidirá con DS.

Los siguientes ejemplos muestran cómo se pueden combinar los distintos campos:

```
.MODEL    small                ; Modelo de memoria small

.MODEL    large, c, farstack   ; Modelo de memoria large
                               ; Lenguaje C, Segmento de pila separado
                               ; del seg. de datos

.MODEL    medium,pascal        ; Modelo de memoria medium
                               ; Pascal, nearstack (por defecto)
```

NOTAS:

a) Direcciones cercanas y lejanas:

Una dirección cercana (near) es una dirección que se encuentra dentro del mismo segmento (para acceder a ella es suficiente con especificar el desplazamiento u offset), mientras que una dirección lejana (far) puede estar en otro segmento y es alcanzada por medio de una dirección de segmento y un desplazamiento. Esto se explicará con mayor detalle más adelante.

b) Ubicación de la pila:

La directiva **.NEARSTACK** ubica el segmento de la pila junto con el segmento de datos. La directiva **.STARTUP**, entonces, genera el código necesario para ajustar el SS:SP de forma que el SS (Stack Segment register) tenga la misma dirección que el DS (Data Segment register). La directiva **.FARSTACK** le da a la pila un segmento propio, es decir que SS no coincidirá con DS. El tipo de pila por defecto es **.NEARSTACK** ya que es conveniente para la mayoría de los programas**,** mientras que **.FARSTACK** se utiliza para casos especiales.

1.3.2- **Creación del Segmento de Pila**

La pila es el lugar de memoria donde se guardan las variables locales y temporales. Se usa la directiva **.STACK** sólo cuando se escribe un módulo principal en assembler. Esta directiva crea el segmento de pila que por defecto tiene un tamaño de 1K (este tamaño es suficiente para la mayoría de los programas pequeños). Para crear un segmento de un tamaño distinto del tamaño por defecto, se le da a la directiva **.STACK** un único argumento numérico indicando el tamaño en bytes:

```
.STACK      2048        ; Usa una pila de 2K
```

1.3.3- **Creación del Segmento de Datos**

Los programas pueden contener datos cercanos (near) o lejanos (far). En general, se guardan en el área de datos cercana aquellos que son más importantes y frecuentes ya que su acceso es más rápido. Pero si los 64 K del segmento no son suficientes, se pueden guardar otros datos menos frecuentes en un segmento de datos lejano. Las directivas **.DATA**, **.DATA?**, **.CONST**, **.FARDATA** y **.FARDATA?** se utilizan para tal fin.

- *Segmentos de Datos Cercanos*

La directiva **.DATA** crea segmentos de datos cercanos. Este segmento está ubicado en un grupo especial identificado como DGROUP, el cual es de 64 K. Cuando se usa **.MODEL**, el assembler automáticamente define DGROUP para el segmento de datos cercano. Los segmentos que se encuentran en DGROUP contienen datos cercanos que pueden ser accedidos a través de DS o SS. También se pueden usar las directivas **.DATA?** y **.CONST** para definir segmentos dentro de DGROUP (**.DATA?** se utiliza para almacenar variables no definidas mientras que **.CONST** se usa para definir constantes).

- *Segmentos de Datos Lejanos*

Los modelos de memoria compact, large y huge usan tipos de datos lejanos por defecto (a pesar de que aún se pueden contruir segmentos de datos usando **.DATA**, **.DATA?** y **.CONST** ya que el efecto de estas directivas no cambia con el tipo de memoria usado). Con las directivas **.FARDATA** y **.FARDATA?**, el assembler crea los segmentos de datos lejanos FAR_DATA y FAR_BSS, respectivamente.

1.3.4- **Creación del Segmento de Código**

Si se está escribiendo un módulo principal o un módulo que será llamado por otro módulo se pueden usar segmentos de código cercanos (near) o lejanos (far).

- *Segmentos de Código Cercanos*

El modelo de memoria small es a menudo la mejor elección para programas en assembler que no serán linkeados con módulos de otro lenguaje, especialmente si no se necesitan más de 64K de código. Este modelo utiliza por defecto direcciones cercanas. Cuando se usa **.MODEL** y directivas simplificadas, la directiva **.CODE** generará un segmento de código. La próxima directiva de segmento cerrará el segmento previo y la directiva **END** al final del programa cierra los segmentos restantes, como se explicó anteriormente.

- *Segmentos de Código Lejanos*

Cuando se requieren más de 64K de código se deben usar los modelos de memoria medium, large o huge para crear segmentos lejanos (que usan direcciones lejanas por defecto). Si se usan múltiples segmentos de código en los modelos small, compact, o tiny, el linker (enlazador) combinará todos los segmentos **.CODE** para todos los módulos dentro de un único segmento.

1.3.5- **Comienzo y fin del código con .STARTUP y .EXIT**

La forma más fácil de comenzar y terminar un programa es utilizando las directivas **.STARTUP** y **.EXIT** en el módulo principal (estas directivas no son necesarias en un módulo que es llamado por otro). El módulo principal contiene el punto de inicio y generalmente, también el de fin. Estas directivas generan automáticamente el código apropiado para la ubicación de la pila especificado con **.MODEL**.

Para comenzar el programa, se debe colocar la directiva **.STARTUP** donde se desee que comience la ejecución. Generalmente, se encuentra inmediatamente después de la directiva **.CODE** :

```
.CODE
.STARTUP
      .
      .                    ; Código ejecutable
      .

.EXIT
 END
```

La directiva **.EXIT** genera código ejecutable, mientras que la directiva **END** informa al assembler que se ha alcanzado el fin del código. Todos los módulos deben terminar con **END**, tanto en directivas simplificadas como en directivas completas. Si la pila fue definida con el atributo **NEARSTACK** (atributo por defecto) **.STARTUP** hace que DS y SS apunten al mismo grupo (DGROUP), generando el siguiente código:

```
@Startup:
          mov dx, DGROUP
          mov    ds, dx
          mov    bx, ss
          sub     bx, dx
          shl     bx, 1
          shl     bx, 1
          shl     bx, 1
          shl     bx, 1
          cli
          mov    ss, dx
          add    sp, bx
          sti
END    @Startup
```

Un programa con el atributo **FARSTACK** no necesita ajustar SS:SP, así que **.STARTUP** sólo se encarga de inicializar DS, como se muestra a continuación:

```
@Startup:

          mov    dx, DGROUP
          mov    ds, dx
            .
            .
 END    @Startup
```

Cuando el programa finaliza, se puede retornar al sistema operativo un código de salida. La directiva **.EXIT** genera un código que le devuelve el control al MS-DOS, terminando el programa.

2. DIRECCIONAMIENTO

2.1- **Direcciones cercanas y lejanas**

Como se comentó anteriormente, las **direcciones cercanas** son aquellas que tienen implícito el segmento al que pertenecen ; es suficiente con especificar el desplazamiento u offset. En las **direcciones lejanas**, en cambio, se debe explicitar tanto el offset como el segmento al cual pertenecen.Todos los datos cercanos y la pila se colocan en un grupo llamado DGROUP, mientras que el código cercano se coloca en un grupo llamado _TEXT. Cada módulo de datos y código lejanos se coloca en segmentos separados.

2.2- **Operandos**

Las instrucciones del lenguaje Assembler trabajan con fuentes de datos llamados operandos. La tabla siguiente resume los cuatro tipos de operandos:

Tipo de Operando	Descripción
Registro	Registro de 16-bit Se puede acceder a su parte alta o baja.
Inmediato	Es un valor constante.
Directo	Representa una ubicación fija en memoria.
Indirecto	La dirección se encuentra almacenada en un registro.

En las instrucciones en que se utilizan dos operandos el operando de la derecha es el operando fuente (contiene el dato que será leído pero no modificado por la operación). El operando de la izquierda es el operando destino (especifica el dato que será cambiado durante la operación).

➢ REGISTRO: este operando se refiere a datos que están almacenados en registros. Los siguientes son algunos ejemplos:

```
mov    bx, 10          ; guarda una constante en BX
add    ax, bx          ; suma BX a AX
jmp    di              ; salta a la dirección en DI
```

Un offset guardado en un registro índice siempre sirve como puntero de memoria.

➢ INMEDIATO: un operando inmediato es una constante o el resultado de una operación que resulta constante. Algunos ejemplos típicos son los siguientes:

```
mov    cx, 20          ; guarda la constante en el registro
add    var, 1Fh        ; suma una constante hexadecimal a una variable
sub    bx, 25          ; resta una expresión constante
```

Un dato inmediato nunca puede ir en el operando destino. Si el operando fuente es inmediato, el operando destino debe ser un registro o un operando directo para proveer un lugar donde almacenar el resultado de la operación. Una expresión involucrada con este tipo de operandos es OFFSET.

OFFSET: Este operador devuelve el desplazamiento de una dirección de memoria, como se muestra a continuación:

```
mov bx,OFFSET var  ; guarda en bx el desplazamiento de var
```

➢ DIRECTO: En este formato, uno de los operandos hace referencia a una localidad de memoria y el otro a un registro. Es importante notar que no existen instrucciones que permitan que ambos operandos sean direcciones de memoria. Por ejemplo:

```
DRC byte  20d   ; se define la variable DRC
mov AX,DRC      ; se mueve DRc a AX
```

Por defecto, las instrucciones que usan direccionamiento directo utilizan el registro DS.

➢ INDIRECTO: Igual que en el direccionamiento directo, el operando se refiere al contenido de una dirección dada. En este caso, los registros utilizados son BX, DI, SI y BP exclusivamente. En la instrucción los registros deben aparecer entre corchetes. Una dirección indirecta tal como [BX] indica que la dirección de memoria a usar está contenida en el registro BX. Por ejemplo: **mov AX,[BX]** coloca el contenido de la localidad de memoria apuntada por BX en el registro AX. Otro ejemplo de este tipo de direccionamiento es el sgte:

```
tabla byte  ?
     .....
mov  BX,OFFSET tabla
mov  AX,[BX]
```

Carga en BX el desplazamiento y luego guarda el contenido de esa dirección en AX. Se pueden usar distintas sintaxis en el direccionamiento indirecto:

- Especificando el operando: En este caso, se usan los datos apuntados por los registros BX, DI, SI o BP. Por ejemplo, la siguiente instrucción mueve al reg. AX el contenido de la dirección en DS:BX:

```
mov     ax, WORD PTR [bx]
```

Cuando se especifica más de un registro, el procesador suma el contenido de las dos direcciones para determinar la dirección efectiva(es decir, la dirección del dato sobre el que se realizará la acción):

```
mov     ax, [bx+si]
```

- Especificando el desplazamiento: Se puede especificar el desplazamiento, el cual se suma a la dirección efectiva. Por ejemplo:

```
mov     ax, tabla[si]
```

En esta expresión, el desplazamiento tabla es la dirección base de un arreglo y el registro SI contiene un índice que apunta a algún elemento del arreglo. El valor de SI se modifica a medida que se va corriendo el programa, generalmente, a través de un bucle. El valor en AX depende del contenido de SI en el momento en que se ejecutó la instrucción. Cada desplazamiento puede ser una dirección o un valor numérico constante. Si hay más de un desplazamiento, el programa los suma para obtener el desplazamiento total. Por ejemplo:

```
tabla   WORD    100 DUP (0)
              .
              .
              .

mov     ax, tabla[bx][di]+6
```

Tanto tabla como 6, son desplazamientos. El programa suma el valor 6 a tabla para obtener el desplazamiento final.

- Especificando el tamaño del operando: Se puede dar el tamaño de un operando de tres formas distintas:

1) A través del tamaño de la variable ya declarada. Por ejemplo:

```
mov     tabla[bx], 0      ; 2 bytes – del tamaño de tabla definido anteriormente
```

2) Con el operador **PTR** .Por ejemplo:

```
mov     BYTE PTR table, 0 ; 1 byte  - especifcado por  BYTE
```

3) Implícito en el tamaño de otro operando.Por ejemplo:

```
mov     ax, [bx]          ; 2 bytes – implícito en AX
```

NOTA:

Aunque el assembler permite una gran variedad de opciones en la sintaxis del direccionamiento indirecto, hay que respetar algunas reglas. Todos los registros deben ir entre corchetes. Se pueden encerrar entre corchetes en forma individual o bien se pueden encerrar todos juntos dentro de un único corchete y separados por el operador “+ ”.La siguiente tabla presenta todas las formas de direccionamiento indirecto posibles:

Tipo	Sintaxis	Dirección Efectiva
De registro indirecto	[BX] [BP] [DI] [SI]	Contenido de un registro.
Relativo a la base	*Desplazamiento*[BX] *desplazamiento* [BP] *desplazamiento* [DI] *desplazamiento* [SI]	Contenido de un registro más desplazamiento.
Base más índice	[BX][DI] [BP][DI] [BX][SI] [BP][SI]	Contenido de un registro base más el contenido de un registro índice.
Base más índice con desplazamiento	*Desplazamiento* [BX][DI] *desplazamiento* [BP][DI] *desplazamiento* [BX][SI] *desplazamiento* [BP][SI]	Suma de un registro base, registro índice y desplazamiento.

2.3- **Manejo de la Pila**

La pila (stack) es un área de memoria para almacenar información temporal.A diferencia de otros segmentos que comienzan el almacenamiento de información en direcciones de memoria bajas, la pila almacena en direcciones altas.La pila es una parte esencial de un programa, ya que en ella se almacenan en forma temporal, direcciones de retorno, argumentos de procedimientos, datos de memoria, banderas o registros. El registro SP es un puntero de la pila. Al principio, el segmento de pila apunta a la dirección más alta de la pila, pero a medida que se le agrega información se mueve desde direcciones altas a direcciones más bajas. Cuando se le extrae información ocurre el proceso inverso, es decir que el puntero se moverá de una dirección baja a otra más alta.

Instrucciones de la pila:

Las instrucciones **PUSH** y **POP** son importantes porque almacenan y recuperan datos de la memoria de la pila. La siguiente tabla resume las distintas formas que pueden tener estas instrucciones:

Instrucción	Comentarios
PUSH *fuente*	Primero se decrementa en 2 el registro SP y después se transfiere una palabra desde el operando fuente a la pila.
POP *destino*	Esta instrucción transfiere una palabra desde una localidad de la pila, cuya dirección está dada por el SP, al operando destino. Una vez hecho esto, se incrementa en dos el registro SP.
PUSHF *fuente*	Primero se decrementa en dos el SP y después se transfiere el registro de banderas a la localidad de la pila indicada por el SP.
POPF *destino*	Esta instrucción transfiere bits de la palabra que se encuentra en la parte superior de la pila hacia el registro de banderas.

3. MANEJO DE DATOS

3.1.1- **Declaración de variables enteras**

Cuando se declara una variable entera, el assembler reserva un lugar en la memoria para dicho entero. El nombre que se le asigna a la variable se transforma en una etiqueta que identifica ese lugar de memoria. Las siguientes directivas indican el tamaño del entero:

Tipo de dato	Bytes
BYTE , **DB** (byte)	1 (de 0 a 255)
SBYTE (byte con signo)	1 (de –128 a +127)
WORD, **DW** (word)	2 (de 0 a 65.535)
SWORD (word con signo)	2 (de –32.768 a +32.767)
DWORD, **DD** (doubleword)	4 (de 0 a 4.294.967.295)
SDWORD (doubleword con signo)	4 (de –2.147.483.648 a +2.147.483.647)

3.1.2- **Declaración de datos**

Se pueden inicializar variables cuando se las declara con constantes o con expresiones que evalúan dichas constantes. El assembler generará un error si se especificó un valor inicial que es demasiado grande para el tipo de variable. Si se utiliza un **?** en lugar de una constante , entonces se está indicando que no se requiere una inicialización y el assembler reservará automáticamente un espacio de memoria pero no escribirá ningún valor en él. Se pueden declararar e inicializar variables en un solo paso como se muestra a continuación:

```
entero      BYTE      16          ; inicializa un byte en 16
negativo    SBYTE     -16         ; inicializa un byte con signo en -16
expression  WORD      4*3         ; inicializa un word en 12 a través de la expresión 4*3
positivo    SWORD     4*3         ; inicializa un worq con signo en 12
vacio       DWORD     ?           ; declara una variable dword pero no la inicializa
lista       BYTE      1,2,3,4,5,6 ; inicializa 6 bytes
```

3.2- **Trabajando con variables simples**

Una vez que se declararon las variables en el programa, se pueden realizar diferentes operaciones con ellas como copiar, mover, sumar, restar, multiplicar y dividir. Puede ocurrir que sea necesario trabajar con un tamaño de dato distinto del que fue declarado originalmente. Como las instrucciones de MASM requieren que los operandos sean del mismo tamaño, se puede realizar cualquiera de las operaciones mencionadas, utilizando el operador **PTR** . Por ejemplo, se puede usar **PTR** para acceder a la parte más significativa de una variable de tamaño DWORD. La sintaxis para este operador es la siguiente:

tipodedato **PTR** *expresión*

donde el operador **PTR** fuerza a *expresión* a ser tratada como si fuera del tipo de dato especificado. Un ejemplo de su uso es el sgte:

```
.DATA
num    DWORD   0
.CODE
mov    AX, WORD PTR num[0]   ; carga en AX los 2 bytes menos significativos de num
mov    DX, WORD PTR num[2]   ; carga en DX los 2 bytes más significativos de num
```

3.2.1- **Transferencia de datos**

Entre las instrucciones más importantes para mover datos de un operando a otro o cargarlos en registros se encuentran: **MOV** (Move), **XCHG** (Exchange), **CBW** (convert byte to word) y **CWD** (convert word to doubleword).

- **MOV**: es una instrucción que copia el operando fuente en el operando destino sin afectar a la fuente. Después de ejecutar esta instrucción ambos operandos (fuente y destino) contienen el mismo dato. El siguiente ejemplo muestra los diferentes usos de esta instrucción:

```
mov    ax, 7          ;  valor inmediato a registro
mov    mem, 7         ;  valor inmediato a memoria (direccionamiento directo)
mov    mem[bx], 7     ;  valor inmediato a memoria (direccionamiento indirecto )

mov    mem, ax        ; registro a memoria (direcc. directo)
mov    mem[bx], ax    ; registro a memoria (direcc. indirecto)
mov    ax, bx         ; registro a registro
mov    ds, ax         ; registro de propósitos generales a registro de segmento

mov    ax, mem        ; memoria a registro (direcc. directo)
mov    ds, mem        ; memoria a registro de segmento

mov    ax, mem[bx]    ; memoria a registro (direcc. indirecto)
mov    ds, mem[bx]    ; memoria a registro de segmento (direcc.indirecto)

mov    mem, ds        ; registro de segmento a memoria (direcc. directo)
mov    mem[bx], ds    ; registro de segmento a memoria (direcc.indirecto)
mov    ax, ds         ; registro de segmento a registro de propósitos generales
```

El siguiente ejemplo muestra varios tipos comunes de MOV que requieren de dos instrucciones:

```
;Mover un valor inmediato a un registro de segmento:

mov    ax, DGROUP  ; carga en AX el valor inmediato DGROUP
mov    ds, ax      ; copia de  AX  al registro DS

;Mover de memoria a memoria:

 mov    ax, mem1   ; carga en AX el valor de la dirección de memoria
 mov    mem2, ax   ; copia de  AX  a la otra dirección de memoria

;Mover de un registro de segmento a otro registro de segmento:

mov    ax, ds         ; carga en AX el dato en DS
mov    es, ax         ; copia de AX a ES
```

- **LEA** (carga la dirección efectiva): La instrucción **LEA** carga en un registro el desplazamiento de los datos especificado por el operando. Como se indica en el ejemplo a continuación, la dirección NUMB del operando se carga en el registro AX, pero no su contenido:

```
lea ax,NUMB
```

Al comparar LEA con MOV, se observa el siguiente efecto: **lea BX,[DI]** carga la dirección de desplazamiento especificada por [DI] (contenido de DI) en el registro BX. Con **mov BX,[DI]** se cargan en el registro BX los datos almacenados en la localidad de memoria direccionada por [DI].

La directiva OFFSET efectúa la misma función que el operador LEA si el operando es un desplazamiento. Por ejemplo, **mov BX, OFFSET LISTA** desempeña la misma función que **lea BX,LISTA**.

Ambas instrucciones cargan el desplazamiento de la localidad LISTA de la memoria en el registro BX. Sin embargo, la directiva OFFSET sólo funciona con operandos sencillos como LISTA; no se puede emplear para un operando como [DI], LISTA[SI], etc. La directiva OFFSET es más eficiente que la instrucción LEA para operandos sencillos, ya que el procesador requiere más tiempo para ejecutar la instrucción **lea BX,LISTA** que **mov BX,OFFSET LISTA**.

3.2.2- **Otras instrucciones para la transferencia de datos**

- **XCHG**: La instrucción **XCHG** (Exchange) intercambia el dato del operando fuente con el dato del operando destino. Se pueden intercambiar datos entre registros o entre registros y memoria, pero no de memoria a memoria. Por ejemplo:

```
xchg   ax, bx          ; coloca en AX el contenido de BX y viceversa
xchg   memory, ax      ; coloca el contenido de "memoria"en AX y viceversa
xchg   mem1, mem2      ; incorrecto: no se pueden intercambiar datos entre memoria
                       ; y memoria
```

- **XLAT**: Esta instrucción carga BX con la dirección de inicio de una localidad de memoria, generalmente una tabla. AL contiene un número, que representa el número de bytes, a partir de la dirección de inicio. El contenido de AL es entonces reemplazado por el byte de la memoria (tabla).
- **IN / OUT:** Una instrucción **IN** transfiere datos de un dispositivo de E/S a AL o AX ; una instrucción **OUT** transfiere datos desde AL o AX a un dispositivo de E/S.

Hay dos formas para la dirección del dispositivo (puerto) de E/S para IN y OUT: *puerto fijo* y *puerto variable*. El direccionamiento de *puerto fijo* permite la transferencia de datos entre AL o AX con el empleo de una dirección del puerto de E/S de 8 bits. Se llama direccionamiento de *puerto fijo* porque el número de puerto se obtiene del código de la instrucción. Por ejemplo, si se ejecuta la instrucción **in AL ,6Ah** , se ingresan en AL los datos de la dirección 6Ah del puerto de E/S. El direccionamiento de *puerto variable* permite transferencias de datos entre AL o AX, y una dirección de 16 bits del puerto. Se llama así porque el número del puerto de E/S está cargado en el registro DX, y se puede variar durante la ejecución del programa.

3.2.3- **Extensión de enteros con signo y sin signo**

Como mover datos entre registros de distinto tamaño no es posible, se debe "extender su signo" para convertir a dicho dato en otro de tamaño más grande. Extender el signo significa copiar el bit de signo del operando original en todos los bits del nuevo operando "extendido". De esta forma, el dato conservará su signo y su valor. Se disponen de dos instrucciones para realizar la extensión del signo que actúan con el registro acumulador (AL o AX):

Instrucción	**Extensión del signo**
CBW (convierte byte a word)	De AL a AX
CWD (convierte word a doubleword)	De AX a DX:AX

En el ejemplo siguiente, se convierten enteros utilizando **CBW** y **CWD**:

```
.DATA

mem8   SBYTE  -5
mem16  SWORD  +5

.CODE
   .
   .
mov    AL, mem8    ;  se carga en AL el dato de 8 bits -5 (FBh)
cbw                ;  se convierte en un dato de 16 bits (FFFBh) en AX
mov    ax, mem16   ;  se carga en AX el dato de 16 bits + 5
cwd                ;  se convierte en un dato de 32 bits (0000:0005h) en DX:AX
```

Estas instrucciones también convierten valores sin signo, siempre que el bit de signo sea cero. De hecho, en el ejemplo anterior, se extiende la variable mem16 sin importar si se la trata como una variable con o sin signo.

3.2.4- **Suma y resta de enteros**

Se pueden usar las instrucciones **ADD**, **INC**, **SUB** y **DEC** para sumar, incrementar, restar y decrementar valores de registros, respectivamente. Las instrucciones ADD, INC, SUB y DEC operan con valores de 8 y16 bits. Estas instrucciones tienen dos requerimientos cuando utilizan dos operandos:

1- Sólo uno puede referirse a una dirección de memoria.

2- Ambos deben tener el mismo tamaño.

El siguiente ejemplo muestra las operaciones de suma y resta entre operandos de 8 bits con y sin signo:

```
.DATA
mem8   BYTE   39
.CODE
 ;Suma
                                          ; C/signo    S/signo
mov     al, 26      ; carga el 26 en AL       26          26
inc     al          ; incrementa               1           1
add     al, 76      ; suma 76                + 76        + 76
                    ;                         ----        ----
                    ;                         103         103
add     al, mem8    ; suma el valor en mem8  + 39        + 39
                    ;                         ----        ----
mov     ah, al      ; copia en AH            -114         142
                                            +overflow
add     al, ah      ; suma a AL el                       142
                    ; contenido de AH                    ----
                    ;                                     28+carry

; Resta
                                          ; C/signo    S/signo
mov     al, 95      ; carga el 95 en AL        95          95
dec     al          ; decrementa               -1          -1
sub     al, 23      ; resta 23                -23         -23
                    ;                         ----        ----
                    ;                          71          71
sub     al, mem8    ; resta el valor en mem8  -122        -122
                    ;                         ----        ----
                    ;                         -51         205+signo

mov     ah, 119     ; carga el 119             119
sub     al, ah      ; resta  -51               -51
7                   ;                         ----
                    ;                          86+overflow
```

Suma con acarreo: Una instrucción de suma con acarreo (**ADC**), suma el bit de la bandera de acarreo a los datos del operando. Esta instrucción aparece casi siempre en los programas que suman elementos de un ancho mayor de 16 bits.Al igual que la instrucción ADD, la ADC afecta a las banderas después de la suma.

Resta con préstamo: Una instrucción de resta con préstamo (**SBB**) funciona igual que una resta normal, excepto que la bandera de acarreo también se resta de la diferencia. Las restas “anchas” (datos de más de 16 bits) requieren que los préstamos se propaguen en la resta, igual que en las sumas “anchas” se propagó el acarreo.

Las instrucciones **INC** y **DEC** tratan enteros como valores sin signo. Cuando una suma entre dos operandos de 8 bits supera el valor 127, el procesador activa la bandera de overflow (también se activa si ambos operandos son negativos y su suma es menor o igual que –128). Cuando la suma supera el 255, el procesador activa la bandera de carry. En el ejemplo anterior de resta, el procesador activa la bandera de signo si el resultado se hace negativo.

3.2.5- **Multiplicación y división de enteros**

- *Multiplicación*: La instrucción **MUL** multiplica números sin signo, mientras que la instrucción **IMUL** multiplica números con signo (en forma de complemento a 2, si es negativo). En ambas instrucciones, un factor debe encontrase en el acumulador (AL para datos de 8 bits y AX para datos de 16 bits). El otro factor puede encontrarse en cualquier registro o dirección de memoria y el resultado de la operación se almacenará en el acumulador. Multiplicar dos datos de 8 bits dará por resultado otro de 16 bits que se almacenará en AX. Multiplicando dos datos de 16 bits se obtendrá otro de 32 bits que se almacenará en los registros DX:AX. El siguiente ejemplo muestra la multiplicación entre datos de 8 y 16 bits:

```
.DATA
 mem16   SWORD   -30000
.CODE

; Multiplicación sin signo con datos de 8 bits
mov     al, 23          ; carga el 23 en AL                      23
mov     bl, 24          ; carga el 24 en BL                    * 24
mul     bl              ; multiplica al acumulador por BL        -----
                        ; Producto en  AX                       552
                        ; se activan las banderas de overflow y carry

; Multiplicación con signo con datos de 16 bits
mov     ax, 50          ; carga el 50 en AX                      50
mul     mem16           ; multiplica el dato en el acumulador   -30000
                        ; por el dato de memoria                ----------
                        ; Producto en DX:AX                   -1500000
                        ; se activan las banderas de overflow y carry
```

Un valor distinto de cero en la mitad más alta del resultado (AH para byte y DX para word) activa las banderas de overflow y carry.

- *División:* la instrucción **DIV** divide números sin signo mientras que **IDIV** realiza la división de números con signo representados en complemento a 2. Ambas instrucciones devuelven un cociente y un resto. El dividendo es el número que será dividido, mientras que el divisor es el número por el que se divide. El resultado es el cociente. El divisor puede encontrarse en cualquier registro o dirección de memoria. La tabla siguiente resume la operación de división para operandos de distintos tamaños:

Tamaño del operando	Registro dividendo	Tamaño del divisor	Cociente	Resto
16 bits	AX	8 bits	AL	AH
32 bits	DX:AX	16 bits	AX	DX

El siguiente ejemplo muestra una división con signo:

```
.DATA

mem16   SWORD   -2000
mem32   SDWORD  500000

.CODE
      .

; División de un dato de 16 bits (sin signo) por otro de 8bits

mov     ax, 700          ; carga el dividendo        700
mov     bl, 36           ; carga el divisor           36
div      bl              ; divide por BL             ------
                         ; cociente en  AL            19
                         ; resto en AH                16

; División de un dato de 32 bits (con signo) por otro de 16 bits

mov     ax, WORD PTR mem32[0] ;   carga en DX:AX
mov     dx, WORD PTR mem32[2] ;                      500000
idiv     mem16                ;                       -2000
                              ;                     ----------
                              ; cociente en AX        -250
                              ; resto en DX              0

; División de un dato de 16 bits (con signo) por otro de 16 bits

mov     ax, WORD PTR mem16    ; carga en AX                    -2000
cwd                           ; extiende el signo a DX:AX
mov     bx,-421               ;                                 -421
idiv    bx                    ; divide por BX                   -----
                              ; cociente en AX                    4
                              ; resto en DX                    -316
```

Si el dividendo tiene el mismo tamaño que el divisor, se debe extender el signo del dividendo de la forma que se explicó anteriormente de manera que este último tenga el tamaño requerido por la instrucción de división.

3.2.6- **Instrucciones Lógicas**

Las instrucciones lógicas **AND**, **OR** ,**TEST** y **XOR** comparan bits en dos operandos. Basados en el resultado de estas comparaciones, estas instrucciones modifican los bits del primer operando (destino). Las instrucciones lógicas **NOT** y **NEG** trabajan con un solo operando. El siguiente ejemplo muestra cómo operan las instrucciones lógicas sobre un valor almacenado en el registro AX:

```
mov     ax, 035h      ; carga el dato en AX          00110101
and     ax, 0FBh      ;                        AND 11111011
                      ;                            ----------
                      ; el nuevo valor es 31h        00110001
or      ax, 016h      ;                         OR 00010110
                      ;                            ----------
                      ; el nuevo valor es 37h        00110111
xor     ax, 0ADh      ;                        XOR 10101101
                      ;                            ----------
                      ; el nuevo valor es 9Ah        10011010
not     ax            ; el nuevo valor es 65h        01100101
```

El grupo de instrucciones lógicas se utiliza para generar “máscaras de prueba” que sirven como base para verificar datos.

La instrucción **AND** borra los bits que no están "enmascarados" (es decir, los bits que no están protegidos por 1). Para enmascarar ciertos bits y borrar otros se debe usar una máscara apropiada en el operando fuente. Los bits de la máscara deben ser 0 para cualquier posición en la que quiera borrar un bit y 1 para aquellos que no quiera alterar.

La instrucción **OR** fuerza bits específicos al valor 1 sin importar su valor actual. Los bits de la máscara, en este caso, deben ser iguales a uno en aquellas posiciones en las que quiera setear el bit en 1 y deben ser 0 en aquellas posiciones en las que no quiera modificar su valor.

La siguiente tabla contiene el grupo de instrucciones lógicas mencionadas:

Instrucción	**Comentario**
AND destino,fuente	Efectúa un "and" entre ambos operandos y retorna el resultado al operando destino.
OR destino,fuente	Efectúa un "or" entre ambos operandos y retorna el resultado al operando destino.
TEST destino,fuente	Es similar a la instrucción and, excepto que no retorna el resultado sino que sólo afecta al registro de banderas.
NEG destino	Esta instrucción genera el complemento a 2 del operando destino.
NOT destino	Esta instrucción genera la negación (complemento a 1) del operando destino.
XOR destino,fuente	Realiza un "or exclusivo" entre ambos operandos devolviendo el resultado al operando destino.

3.2.7- **Corrimiento y Rotación**

Las instrucciones de corrimiento desplazan al operando destino un número determinado de bits, previamente especificados, hacia la izquierda (**SHL** o **SAL**) o hacia la derecha (**SHR** o **SAR**). Las instrucciones de rotación también desplazan el operando destino ya sea a la izquierda (**RCL** o **ROL**) o a la derecha (**RCR** o **ROR**) pero en este caso, colocan los bits que salen del registro al principio o al final del mismo. Todas las instrucciones de corrimiento tienen el mismo formato. Antes de ejecutarse la operación, el operando destino contiene el valor que será desplazado y una vez que ésta se ejecutó contiene el dato desplazado. El operando fuente contiene el número de bits que se correrá o rotará, que puede ser un valor inmediato o encontrarse almacenado en el registro CL. El siguiente ejemplo muestra el funcionamiento de las instrucciones de corrimiento y rotación:

```
.DATA
num  BYTE    00000010
.CODE
mov     cl, 2
mov     bl, 57h     ; carga el valor que será cambiado   01010111
rol     num, cl     ; rota  dos bits a la izquierda      00001000
or      bl,num      ;                                    ----------
                    ; el nuevo valor es 05Fh             01011111
rol     num, cl     ; rota dos posiciones más            00100000
or      bl,num      ;                                    ----------
                    ; el nuevo valor es 07Fh             01111111
```

3.2.8- **Multiplicación y División con Corrimiento**

Se pueden usar las instrucciones de corrimiento y rotación para realizar multiplicaciones y divisiones. Desplazar un bit hacia la derecha tiene el efecto de dividir por dos, mientras que desplazarlo hacia la izquierda tiene el efecto de multiplicarlo por dos. Es decir que se pueden aprovechar estas instrucciones para realizar multiplicaciones o divisiones por potencias de dos. Para dividir números sin signo se debe usar **SHR** (Shift Right), mientras que para dividir números con signo se debe usar **SAR** (Shift Arithmetic Right). De la misma forma, se pueden usar **SAL** or **SHL** para realizar multiplicaciones entre números con signo y sin signo, respectivamente.

La ventaja de usar estas instrucciones en lugar de usar las instrucciones específicas para tal fin, es que el procesador requiere de menos tiempo para ejecutar estas últimas.

4. INSTRUCCIONES PARA EL CONTROL DE PROGRAMAS

Muy pocos programas ejecutan todas sus líneas en forma secuencial desde **.STARTUP** hasta **.EXIT**. En general, se salta de un punto del programa a otro, se repite una acción hasta que se alcanzó una determinada condición y se pasa el control a y desde un procedimiento. En esta parte, se explicarán las instrucciones para el control del programa, que incluyen saltos y llamadas a procedimientos.

4.1- **Jumps**

La instrucción para salto (**jump**) permite al programador saltar secciones de un programa y transferir el control a cualquier parte de la memoria para la siguiente instrucción. Una instrucción de salto condicional permite tomar decisiones basadas en pruebas numéricas. Los resultados de estas pruebas numéricas se conservan en los bits de bandera, los cuales se testean después con instrucciones condicionales para salto.

4.1.1- **Jumps incondicionales**

Hay tres tipos de instrucciones para salto incondicional: salto corto, cercano y lejano. El salto corto es una instrucción que permite transferir el programa a localidades de memoria que están dentro del intervalo +127 bytes y –128 bytes de la localidad de memoria después del salto. El salto cercano permite transferir el programa dentro de un intervalo de ± 32 Kbytes desde la instrucción en el segmento de código actual. Por último, el salto lejano permite un salto a cualquier localidad en la memoria. La instrucción **JMP** es la que se utiliza para el salto incondicional.

- *Saltos Cortos:* En el ejemplo siguiente se muestra la forma en que las instrucciones para salto corto pasan el control de una parte del programa a otra:

```
INICIO:   mov AX, 1
          add AX, 1
jmp short SIGUE          ; salto corto
SIGUE:  mov BX, AX
          jmp INICIO
```

Se debe tener en cuenta la forma en que JMP SHORT SIGUE, utiliza la directiva SHORT para obligar a que se produzca un salto corto, mientras que la otra JMP INICIO, no lo hace. En casi todos los programas de ensamblador, se selecciona la mejor forma de la instrucción para salto, de modo que la segunda instrucción (JMP INICIO) también se ensambla como un salto corto.

Siempre que en una instrucción para salto se menciona una dirección, ésta se determina con una *etiqueta*. Un ejemplo es JMP SIGUE que salta a la etiqueta SIGUE para al próxima instrucción. Nunca se utiliza una dirección hexadecimal real con una instrucción para salto. La etiqueta SIGUE debe ir seguida de dos puntos (SIGUE:) para permitir que una instrucción de salto la tenga como referencia. Si la etiqueta no va seguida de dos puntos, no se puede saltar a ella. Se debe tener en cuenta que la única vez que se usarán los dos puntos después de una etiqueta, es cuando ésta se emplea con una instrucción para salto o llamada.

➢ *Salto Cercano*: Este salto es similar al anterior, excepto que su distancia es más larga. Un salto cercano puede saltar a cualquier localidad de la memoria dentro del segmento de código actual. En el ejemplo se muestra la misma porción de programa que antes, excepto que ahora se trata de una distancia mayor:

```
INICIO:   mov AX, 1
          add AX, 1
jmp near SIGUE ; salto cercano
SIGUE:  mov BX, AX
          jmp INICIO
```

➢ *Salto Lejano:* Los saltos lejanos adquieren nuevas direcciones de segmentos y desplazamientos para lograr el salto (es decir que cuando se realiza un salto lejano se modifica tanto el registro IP(Instruction Pointer) como el CS).

NOTA: Cuando no se especifica la distancia del salto a través de las directivas **SHORT**, **NEAR** y **FAR** , el assembler optimiza el tipo de salto según lo que considere más adecuado.

4.1.2- **Jumps Condicionales**

La forma más común de transferir el control en assembler, es a través de los saltos condicionales. Esta instrucción se compone de dos pasos para su ejecución:

1. Primero se testea la condición para que se realice el salto (probando uno o algunos bits de bandera).
2. Si la condición que se prueba es verdadera, ocurre una transferencia a la etiqueta relacionada con el salto condicional. Si la condición es falsa, se ejecuta la siguiente instrucción en la secuencia del programa.

Los saltos condicionales siempre son *cortos* en los microprocesadores 80186. Esto limita el alcance del salto a una distancia de +127 bytes y –128 bytes desde la localidad que sigue del salto condicional.

Registro de Banderas: Las *banderas* indican la condición del microprocesador, a la vez que controlan su funcionamiento. Los bits de bandera cambian después de ejecutar muchas de las instrucciones aritméticas y lógicas. A continuación, aparece una lista con cada bit de bandera, con una breve descripción de su función:

CF*(acarreo)*: Indica un acarreo después de una suma o un "préstamo" después de una resta.
PF*(paridad)*: Es un 0 para una paridad impar y un 1 para paridad par.
AF*(acarreo auxiliar)*: Indica la presencia de un acarreo generado del cuarto bit de un byte. Su mayor uso es durante operaciones aritméticas con números BCD:
ZF*(cero)*: Indica que el resultado de una operación aritmética o lógica es cero. Si Z=1 el resultado es cero, en caso contrario, será Z=0.
SF*(signo)*: Indica el signo aritmético del resultado después de una suma o resta. Si S=1 el resultado es negativo, en caso contrario, S=0. Se debe tener en cuenta que el valor de la posición del bit más significativo se coloca en el bit de signo para cualquier instrucción que afecte las banderas.
TF*(trampa)*: Cuando este bit es activa, el procesador ejecuta una instrucción a la vez.
IF*(interrupción)*: Esta bandera es la de habilitación de una interrupción. El procesador atenderá una interrupción sólo cuando este bit sea activado.
DF*(dirección)*: Cuando está activada, causa que el contenido de los registros índice se decremente después de cada operación de una cadena de caracteres.
OF*(overflow)*: Esta bandera es activada cuando el resultado de una operación es mayor que el máximo valor que es posible representar con el número de bits del operando destino.

La tabla siguiente muestra las instrucciones de salto condicional que dependen de los bits de bandera. Como se muestra en la lista a continuación, varios saltos condicionales tienen 2 o 3 nombres—**JE** (Jump if Equal) y **JZ** (Jump if Zero), por ejemplo. Los nombres compartidos se ensamblan como la misma instrucción de máquina, de forma que el programador pueda elegir el mnemónico que le parezca más apropiado.

Instrucción	Salta si
JC/JB/JNAE	CF=1
JNC/JNB/JAE	CF=0
JBE/JNA	CF=1 o ZF=1
JA/JNBE	CF=0 y ZF=0
JE/JZ	ZF=1
JNE/JNZ	ZF=0
JL/JNGE	SF ≠ OF
JGE/JNL	SF = OF
JLE/JNG	ZF=1 o SF ≠ OF
JG/JNLE	ZF=0 y SF = OF
JS	SF=1
JNS	SF=0
JO	OF=1
JNO	OF=0
JP/JPE	PF=1 (paridad par)
JNP/JPO	PF=0 (paridad impar)

Existen tres categorías de saltos condicionales basados en:

1) La comparación de dos valores.
2) En el estado de activación individual de los bits de un operando.
3) Si un valor es nulo.

1) **Jumps basados en la comparación de dos valores**

La instrucción **CMP** (comparar) es la forma más común de testeo para saltos condicionales. Esta instrucción compara dos valores sin modificarlos, y luego, activa las banderas según el resultado de la comparación. La instrucción **CMP** realiza la misma operación que la instrucción **SUB**, excepto que **CMP** no cambia el operando destino. Sin embargo, ambas instrucciones afectan al registro de banderas. Se pueden comparar valores con signo o sin signo, pero para eso se debe elegir la instrucción de salto adecuada:

Saltos Codicionales basados en la Comparación de dos valores

Valores Con Signo		Valores Sin Signo	
Instrucción	**Saltar si**	**Instrucción**	**Saltar si**
JE	ZF = 1	**JE**	ZF = 1
JNE	ZF = 0	**JNE**	ZF = 0
JG/JNLE	ZF = 0 y SF = OF	**JA/JNBE**	CF = 0 y ZF = 0
JLE/JNG	ZF = 1 o SF ≠ OF	**JBE/JNA**	CF = 1 o ZF = 1
JL/JNGE	SF ≠ OF	**JB/JNAE**	CF = 1
JGE/JNL	SF = OF	**JAE/JNB**	CF = 0

Los mnemónicos de los jumps siempre se refieren a la comparación del primer operando (destino) de la instrucción **CMP** con el segundo operando (fuente). En el siguiente ejemplo, **JG** testea si el primer operando es mayor que el segundo:

```
cmp    ax, bx            ; Compara AX  con  BX
jg     ETIQUETA1         ; Es equivalente a: Si ( AX > BX ) ir a ETIQUETA1
jl     ETIQUETA2         ; Es equivalente a: Si ( AX< BX ) ir a ETIQUETA2
```

2)**Jumps basados en el estado de activación de los bits de un operando**

El estado de activación individual de los bits de un único valor, también puede servir como un criterio de testeo para saltos condicionales. La instrucción **TEST** testea si los bits específicos de un operando están activados o no. Esta instrucción realiza la misma operación que **AND**, excepto que **TEST** no cambia ningún operando. El siguiente ejemplo muestra una aplicación de **TEST**:

```
.DATA
 BITS   BYTE   ?
.CODE
      .
      .
      .
; Si el bit 2 o el bit 4 están en uno,  entonces llamar al procedimiento TAREA1

test   BITS, 10100
jz     SEGUIR1          ; Salta si el resultado de la operación anterior es cero
call   TAREA100         ; Llama al procedimiento TAREA1
SEGUIR1:
      .
      .

; Si los bit 2 y 4 valen cero, entonces llamar al procedimiento TAREA2


test    BITS, 10100
jnz     SEGUIR2         ; Salta si el resultado de la operación anterior no es nulo
call    TAREA2          ; Llama al procedimiento TAREA2
SEGUIR2:
      .
      .
```

El operando fuente de **TEST** es en general una “máscara”, en la cual los únicos bits que valen uno son los que se quieren testear. El operando destino contiene el valor que será testeado.

3) **Jumps basados en un valor nulo**

En un programa, a menudo se realizan saltos basados en el valor que contiene un registro en particular. Por ejemplo, la instrucción **JCXZ** salta dependiendo del valor contenido en el registro CX. La condición paraa el salto se puede testear por cero o cualquier otro valor de un registro con la instrucción **OR**. En el siguiente ejemplo, se testea si el registro BX contiene un valor nulo:

```
or     bx, bx
jz     CERO       ; Saltar si es cero
```

El código anterior es equivalente a:

```
cmp    bx, 0      ; Compara al contenido de BX con cero
je     CERO       ; Salta si son iguales
```

pero el primero produce un código más rápido y más corto, ya que no usa un operando inmediato.

4.2- **LOOPS**

La instrucción **LOOP** repite una acción hasta que se alcanzó una condición de fin. Es una combinación de de decremento de CX y un salto condicional. La siguiente lista compara los distintos tipos de estructuras de **LOOPS**:

Instrucción	Acción
LOOP *etiqueta*	Decrementa a CX en forma automática y si éste no es igual a cero salta a la dirección indicada por la etiqueta. Cuando CX = 0, se ejecuta la siguiente instrucción.
LOOPE *etiqueta* **LOOPZ** *etiqueta*	Repite un ciclo mientras sea CX≠ 0 y además, mientras se cumpla una condición de igualdad (es decir, mientras se cumpla que ZF=1).
LOOPNE *etiqueta* **LOOPNZ** *etiqueta*	Estas instrucciones son opuestas a las anteriores. Repite un ciclo mientras CX≠ 0 y además, mientras exista una condición de desigualdad (es decir, mientras se cumpla que ZF=0).
JCXZ *etiqueta*	Transfiere el control a la dirección indicada por la etiqueta si CX=0.

El siguiente ejemplo muestra el uso de las distintas instrucciones **LOOP**:

```
; Instrucción LOOP
mov    cx, 200              ; Seteo el contador
PROXIMO1:   .               ; Repite el ciclo 200 veces
            .
            .
loop   PROXIMO

; Instrucción LOOPNE : Se repite mientras AX≠Dato
mov     cx, 256             ; Seteo el contador
PROXIMO2:  .                ; El ciclo se repetirá como máximo 256 veces
           .                ; Instrucciones que modifican el contenido de AX
           .
cmp     al, Dato            ; Compara al con Dato
loopne  PROXIMO2            ; Si no son iguales se repite el ciclo
                            ; Si son iguales se debe continuar con la próxima instrucción
```

4.3- **PROCEDIMIENTOS**

El procedimiento o subrutina es un grupo de instrucciones que, por lo general, desempeña una tarea. Un procedimiento es una sección de un programa que se puede volver a utilizar y que se almacena una vez en la memoria, pero que se emplea tantas veces como se lo necesite. Esto ahorra espacio en la memoria y facilita el desarrollo de la programación. La instrucción **CALL** llama al procedimiento y la instrucción **RET** retorna del procedimiento.

4.3.1- **Definición de un Procedimiento**

Todos los procedimientos requieren una etiqueta al comienzo y una instrucción **RET** al final del mismo. En general, se definen usando la directiva **PROC** al inicio del procedimiento y con la directiva **ENDP** al final. La instrucción **RET** normalmente se ubica inmediatamente antes de la directiva **ENDP**. El assembler se asegura que la distancia de la instrucción **RET** coincida con la distancia definida por la directiva **PROC** directive. La sintaxis básica para **PROC** es:

etiqueta **PROC** [[**NEAR** | **FAR**]]
.
.
.
RET [*constante*]
etiqueta **ENDP**

La instrucción **CALL** guarda en la pila la dirección de la próxima instrucción y transfiere el control a la dirección especificada. La sintaxis es:

CALL *etiqueta*

Las llamadas a procedimientos pueden ser cercanas (near) o lejanas (far). Las llamadas cercanas guardan en la pila el offset de la dirección de la siguiente instrucción, es decir que el procedimiento se debe localizar en el mismo segmento que el programa principal. La instrucción **CALL** lejana guarda en la pila el offset y segmento correspondientes a la dirección siguiente, por lo que permite llamar a procedimientos ubicados en cualquier lugar en la memoria.

La instrucción **RET** extrae un número (offset de la dirección de retorno) de 2 bytes de la pila y lo almacena en el registro IP, o bien un número de 4 bytes (offset y segmento de la dirección de retorno) y lo coloca en los registros IP y CS. Las directivas para retorno cercano o lejano están definidas con la directiva **PROC** para el procedimiento. Esto selecciona en forma automática la instrucción adecuada para el retorno.

Existe otra forma para la instrucción de retorno, la cual suma un número al contenido del apuntador de pila (SP) antes del retorno. Si hay que borrar lo salvado en la pila antes de un retorno, se suma un número al SP antes de que se recupere la dirección de retorno de la pila. El efecto es omitir o borrar datos de la pila.

En el ejemplo siguiente, se muestra la forma en que este tipo de retorno borra los datos salvados en la pila por algunas instrucciones **PUSH**:

```
PRUEBA proc near
                push AX
                push BX
                  .
                  .
                ret 4
PRUEBA endp
```

RET 4 suma 4 a SP antes de recuperar la dirección de retorno de la pila. Debido a que **PUSH AX** y **PUSH BX**, juntas, ocuparon 4 bytes de datos en la pila, este retorno borra efectivamente a AX y BX de la pila. Este tipo de retorno rara vez aparece en los programas.

5. BIBLIOGRAFÍA

1. Los Microprocesadores INTEL – Barry Brey – Ed. Prentice Hall (3º edición)

2. Lenguaje ensamblador para microcomputadoras IBM – J. Terry Godfrey – Ed. Prentice Hall

3. Lenguaje ensamblador y programación para PC IBM y compatibles – Peter Abel – Ed. Pearson Educación

4. Programmer's Guide – Microsoft MASM