INSTRUCCIONES DE TRANSFERENCIA BÁSICAS, ARITMÉTICAS Y LÓGICAS

Sistemas Digitales con Microprocesadores.

M.C. Juan Carlos Olguín R.

Clasificación de registros 8086

- Los registros del 8086 pueden clasificarse en tres grupos:

- **1)Registros de uso general, 2)Registros apuntadores y de índice y 3)Registros de segmentos.**

- Adicionalmente tienen un registro de banderas que señala las condiciones respecto al funcionamiento de la unidad aritmética y lógica (ALU).

Modelo de programación del microprocesador 8086

Programa en Ensamblador

- Normalmente podemos considerar que un programa en ensamblador está formado de las siguientes secciones:

✓Código de inicio
✓Declaración de constantes simbólicas
✓Variables del programa
✓Código del programa
✓Código de terminación

```
;*****************************************************************
; PRIMER.ASM.
;
; Este programa ilustra la estructura de un programa en
; ensamblador. También muestra las principales directivas
; para El ensamblador.
;*****************************************************************

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

        ideal
        dosseg
        model small
        stack 256

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

cte     equ     10

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

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

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

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

        mov     al, [dato1]  ; resul = dato1 + dato2 + cte
        add     al, [dato2]
        add     al, cte
        mov     [resul], al

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

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

        end    inicio
```

3.-Variables del programa

- Las variables de un programa se declaran en el segmento de datos, por lo que la sección de declaración de variables empieza con la directiva **dataseg**. Esta directiva le indica al ensamblador que almacene las variables en el segmento de datos del programa. Las variables pueden estar inicializadas o no. Los valores de las variables inicializadas se almacenan en el código del programa y se cargan en las variables al ejecutar el programa.

- Para declarar variables podemos utilizar las directivas **db, dw, dd o dq. La directiva db** define una variable de un byte, la directiva **dw** define una variable de tipo palabra (dos bytes), la directiva **dd** define una variable de tipo palabra doble (cuatro bytes) y la directiva **dq** define una variable de tipo palabra cuádruple (ocho bytes).

La sintaxis de estas directivas es:

```
nomVar    db    exp
nomVar    dw    exp
nomVar    dd    exp
nomVar    dq    exp
```

donde *nomVar es el nombre de la variable y exp es una expresión constante cuyo valor se utiliza para* inicializar la variable.

Si la variable no se inicializa *exp se substituye por un signo de interrogación,* ***?.***

Ejemplo:

```
bdato   db      10h                 ; Variable de un byte
                                    ; inicializada a 10h.

cresp   db      'A'                 ; Variable de un byte
                                    ; inicializada a 65d.

bdatx   db      ?                   ; Variable de un byte
                                    ; no inicializada.

wdato   dw      0123h               ; Variable de tipo
                                    ; palabra inicializada
                                    ; a 0123h.

wdatx   dw      ?                   ; Variable de tipo
                                    ; palabra no
                                    ; inicializada.

ddato   dd      01234567h           ; Variable de tipo
                                    ; palabra doble
                                    ; inicializada a
                                    ; 01234567h.

ddatx   dd      ?                   ; Variable de tipo
                                    ; palabra doble
                                    ; no inicializada

qdato   dq      0123456789ABCDEFh   ; Variable de tipo
                                    ; palabra cuadruple
                                    ; inicializada a
```

4.- Código del programa

- En esta sección van las instrucciones que el microprocesador ejecutará al correr el programa. Todas las instrucciones van en el segmento de código, por lo que la sección de código del programa empieza con la directiva **codeseg.**
- Esta directiva le dice al ensamblador que almacene las instrucciones en el segmento de código del programa.

- El código del programa en la mayoría de los casos empieza con las siguientes líneas:

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

La primera línea contiene la directiva **codeseg** mencionada anteriormente**.**
En otras versiones es: .code

La segunda línea establece que ***inicio*** *es una etiqueta asociada a la dirección de la primera instrucción* del programa, que en este ejemplo aparece en la siguiente línea.

Esta etiqueta es usada en la directiva **end explicada en la siguiente sección.**

En la tercera y cuarta línea se carga en el registro DS la dirección de inicio del segmento de datos del programa. El símbolo predefinido **@data** tiene el valor de esa dirección.
La instrucción **mov** mueve datos entre registros o entre registros y localidades de memoria. En este ejemplo la primera instrucción **mov** almacena la dirección del inicio del segmento de datos en el registro **AX** y la segunda instrucción mov mueve esa dirección del registro **AX** al registro de segmento **DS.**

Nota:

No
se puede cargar una dirección directamente a uno de los registros de segmento, por lo que la instrucción:

```
mov     ds, @data          ; No válido
```

no es válida.

La dirección debe cargarse primero a un registro de propósito general y luego de ahí moverse al registro de segmento

También la mayoría de los programas que se van a ejecutar bajo el sistema operativo MSDOS deben terminar con las siguientes líneas de código:

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

El propósito de este código es llamar a una de las rutinas del sistema operativo que termina la ejecución del programa y nos regresa al sistema operativo. Esta rutina que se conoce como una rutina de servicio a interrupción se verá más adelante en el tema de: TIPOS INTERRUPCIONES.

La llamada a esa rutina se hace mediante la instrucción: **int 21**

Es decir, Previamente se cargó en el registro **AH** el valor **04Ch** que especifica que se desea terminar el programa y regresar al sistema operativo y en el registro **AL** el valor de la variable ***codsal,*** *que es el código de* salida del programa.

El programa le regresa al sistema operativo el valor del código de salida en la variable de ambiente **errorlevel.** Por convención, un código de salida de cero denota que el programa terminó con éxito y un código diferente de cero indica que hubo un error durante la ejecución del programa.

El valor del código identifica al error ocurrido.

En los programas en C también se acostumbra regresarle un código de salida al sistema operativo. Esto se hace mediante la proposición return en la función *main():*

```
int main(void)
{
    ...

    return 0;
}
```

5.- Código de terminación

```
end   [etiqueta]
```

Todo programa en ensamblador debe contener una directiva **end,** la cual marca el final del código fuente. En otras veciones se ocupa: end main

El ensamblador ignora cualquier línea después de la directiva **end.**

etiqueta *es el nombre simbólico de la dirección de la instrucción por la que empezará a ejecutarse el* programa. Si se omite, el programa empezará su ejecución en la primera instrucción del código fuente.

En un programa de un sólo módulo, es decir con un archivo con código fuente, la directiva **end debe** siempre especificar el inicio del programa. En un programa que consiste de más de un módulo, sólo la directiva **end** en el módulo que contiene la primera instrucción a ejecutar debe llevar una etiqueta. En los otros módulos la directiva **end** debe aparecer por sí sola.

INSTRUCCIONES DE TRANSFERENCIA BÁSICAS Y ARITMÉTICAS

Sistemas Digitales con Microprocesadores.

M.C. Juan Carlos Olguín R.

Operandos Y Modos De Direccionamiento

- Los operandos de una instrucción especifican los datos con los que va a trabajar la instrucción. Esos datos se encuentran en registros o en localidades de memoria y los operandos le dicen al ensamblador en qué registro o localidad de memoria está el dato.
- En otras palabras los operandos o la ausencia de ellos establece el mecanismo para calcular las direcciones donde se encuentran los datos.
- A ese mecanismo se le conoce como **modo de direccionamiento.**

Instrucciones Sin Operandos O Modo De Direccionamiento Inherente

Algunas instrucciones no tienen operandos. Esto se debe a que la instrucción no necesita datos o a que los datos con los que opera la instrucción están en registros predefinidos.

Ejemplo:

```
clc     ; Pone la bandera de acarreo en 0. No hay
        ; dato.

cbw     ; Extiende el valor signado del registro
        ; de 8 bits AL al registro de 16 bits AX.
        ; En este caso el dato se encuentra en el
        ; registro AL antes de la operación y en
        ; el registro AX después de la operación.
```

Operandos Constantes O Modo De Direccionamiento Inmediato

Muchas de las instrucciones permiten que uno de los operandos sea una constante. La constante puede ser una constante numérica, una expresión constante o una etiqueta.

Una **constante numérica** es un número de uno o dos bytes expresado en decimal, hexadecimal o binario. Por ejemplo:

```
mov     al, 65          ; Constante decimal.
mov     al, 41h         ; Constante hexadecimal.
mov     al, 01000001b   ; Constante binaria.
mov     al, 'A'         ; Constante de carácter.
                        ; Se almacena su código
                        ; ASCII.
add     ax, 1234        ; Constante de dos bytes.
```

Una **expresión constante** es una expresión donde todos sus operandos son constantes. Las expresiones que el Assembler puede evaluar incluyen paréntesis anidados, operadores aritméticos, lógicos y relacionales y otros operadores que nos dan la dirección de segmento y desplazamiento de etiquetas y que determinan el tamaño de las variables.

La evaluación de estas expresiones ocurre al tiempo de ensamblado. Algunos de los operadores que pueden usarse en expresiones se muestran en la tabla siguiente.

La precedencia de los operadores va de mayor a menor con los operadores en la misma celda con la misma precedencia:

Tabla Algunos operadores usados en expresiones constantes

Operador	Descripción	Ejemplos
() [] **offset** **seg** size	Cambia precedencia Indirección Desplazamiento Dirección de segmento Tamaño de una variable	mov ax, 2*(3 + 2)/3 add bx, [12h] mov ax, offset dato1 mov ax, seg dato1 mov ah, size dato1
-	Negativo	mov ah, -2
* / **mod** **shl** **shr**	Multiplicación División Módulo Corrimiento a la izquierda Corrimiento a la derecha	mov ax, 2*(3 + 2)/3 mov cx, 15 mod 4 mov ax, 3 shr 4 and 0Ah
+ -	Suma Resta	mov ax, 2*(3 + 2)/3
not	Complemento de bits	mov ax, not 3
and	Intersección de bits	mov ax, 3 shr 4 and 0Ah
or **xor**	Unión inclusiva de bits Unión exclusiva de bits	mov ax, 3 shl 4 or 3Ah

Una **etiqueta puede generar una constante de dos formas:**

Si se utiliza como operando de ciertos operadores en expresiones constantes. Por ejemplo, en las siguientes instrucciones la etiqueta dato1, que es el nombre de una variable, genera una constante:

```
mov     ax, offset dato1
mov     ax, seg dato1
mov     ah, size dato1
```

Si se utilizan como el destino de instrucciones de salto y llamada a subrutina. Por ejemplo, la etiqueta ciclo que es el operando de la operación de salto jb, genera una constante: La dirección a la que salta el programa.

```
ciclo:
        ...
        cmp     ax, 10
        jb      ciclo
```

Direccionamiento directo

Este modo se conoce como directo, debido a que en el segundo operando se indica la dirección de desplazamiento donde se encuentran los datos de origen.

Ejemplo:

```
Mov AX,[1000h]        ;Copia en AX lo que se encuentre almacenado en
                      ; DS:1000h
```

En este modo de direccionamiento el dato está en una localidad de memoria. El operando es una expresión constante sobre la que está operando el operador de indirección, []. Su sintaxis tiene la forma:

[*expCte*]

donde *expCte representa el desplazamiento del dato con respecto al segmento en que está contenido,* normalmente el segmento de datos.

Ejemplo:

```
dato1   dw      0123h
dato2   dd      456789ABh
        ...

        mov     ax, [0AF3h]     ; Guarda en el registro
                                ; AX la palabra
                                ; almacenada en la

                                ; dirección 0AF3h.

        mov     bx, [dato1]     ; Guarda en el registro
                                ; BX el valor de la
                                ; variable dato1

        mov     cl, [dato1]     ; Guarda en el registro
                                ; CL el byte menos
                                 ; significativo del valor
                                ; de la variable dato1,
                                ; esto es, 23h.
```

```
mov     dh, [dato1+1]   ; Guarda en el registro
                        ; DH el byte más
                        ; significativo del valor
                        ; de la variable dato1,
                        ; esto es, 01h.

mov     ax, [dato2]     ; Guarda en el registro
                        ; AX la palabra menos
                        ; significativa del valor
                        ; de la variable dato2,
                        ; esto es, 89ABh.

mov     bx, [dato2+2]   ; Guarda en el registro
                        ; BX la palabra más
                        ; significativa del valor
                        ; de la variable dato2,
                        ; esto es, 4567h.

mov     cx, [dato2+1]   ; Guarda en el registro
                        ; CX la palabra que se
                        ; encuentra un byte más
                        ; adelante del inicio de
                        ; la variable dato2, esto
                        ; es, 6789h.
```

Instrucciones en lenguaje ensamblador

- Las separaremos por su función en 6 grupos:

- 1. Instrucciones de transferencia de datos
- 2. Instrucciones aritméticas
- 3. Instrucciones lógicas
- 4. Instrucciones de control de flujo
- 5. Instrucciones de control del procesador
- 6. Instrucciones de cadenas.

Para describir la sintaxis de las instrucciones se usará la sig. notación

Notación empleada en la sintaxis de las instrucciones

Símbolo	Significado
|	Una de dos o más opciones mutuamente excluyentes
[]	Los términos entre corchetes son opcionales
	Si la instrucción aparece sola es que no tiene operandos.
immB	Operando de un byte con direccionamiento inmediato
immW	Operando de una palabra con direccionamiento inmediato
memB	Operando de un byte que se encuentra en una localidad de memoria.
memW	Operando de una palabra que se encuentra en una localidad de memoria.
regB	Registro de propósito general de un byte.
regW	Registro de propósito general de una palabra.

Instrucciones De Transferencia

I. **mov** *destino, fuente* ;Copia un valor de *fuente a destino*.

Sintaxis:

```
mov   regB, inmB|regB|memB
mov   memB, inmB|regB
mov   regW, inmW|regW|memW
mov   memW, inmW|regW
```

El valor de *fuente no se modifica*. *Sólo uno de los operandos puede ser una localidad de memoria*. *Si* uno de los operandos es un registro de segmento, el otro registro debe ser un registro de propósito general.

La instrucción **mov** no afecta el estado de las banderas.

Cuando uno de los registros es el registro **AX o AL,** el Assembler genera código más rápido.

II. **xchg** *destino, fuente* ; Intercambia los valores en *destino y fuente*.

Sintaxis:

```
xchg regB, regB|memB
xchg memB, regB
xchg regW, regW|memW
xchg memW, regW
```

Sólo uno de los operandos puede ser una localidad de memoria.

La instrucción **xchg** no afecta el estado de las banderas.

Cuando los valores a intercambiar están en registros, el Assembler genera código más rápido.

Instrucciones Aritméticas

I. **add** *destino, fuente* **;**Suma los contenidos de *fuente y destino y guarda el resultado en destino.*

Sintaxis:

```
add   regB, inmB|regB|memB
add   memB, inmB|regB
add   regW, inmB|inmW|regW|memW
add   memW, inmB|inmW|regW
```

La instrucción **add** modifica las siguientes banderas:

sobreflujo **O**, signo **S,** cero **Z,** acarreo auxiliar **A,** paridad **P** y acarreo **C.**

II. adc *destino, fuente* **;**Suma los contenidos de *fuente, destino y el valor de la bandera de acarreo y guarda el resultado en destino.*

Sintaxis:

```
adc   regB, inmB|regB|memB
adc   memB, inmB|regB
adc   regW, inmB|inmW|regW|memW
adc   memW, inmB|inmW|regW
```

La instrucción **adc** modifica las siguientes banderas:

sobreflujo **O,** signo **S**, cero **Z,** acarreo auxiliar **A,** paridad **P** y acarreo **C.**

NOTA.- Al sumar valores multibytes o multipalabras, use **add** para sumar los bytes o palabras menos significativos y después use la instrucción **adc** para sumar los bytes o palabras más significativos y los posibles acarreos generados por las sumas de los bytes o palabras menos significativos.

III. **inc** *destino* ;Incrementa en uno el contenido de *destino*.

Sintaxis:

```
inc   regB|memB
inc   regW|memW
```

La instrucción **inc** modifica las siguientes banderas:
sobreflujo **O,** signo **S**, cero **Z,** acarreo auxiliar **A,** paridad **P** y acarreo **C.**

IV. **sub** *destino, fuente* **;**Resta al valor de *destino el valor de fuente y guarda el resultado en destino*.

Sintaxis:

```
sub   regB, inmB|regB|memB
sub   memB, inmB|regB
sub   regW, inmB|inmW|regW|memW
sub   memW, inmB|inmW|regW
```

La instrucción **sub** modifica las siguientes banderas:
sobreflujo **O,** signo **S**, cero **Z,** acarreo auxiliar **A,** paridad **P** y acarreo **C.**

V. **sbb** *destino, fuente* ;Resta al valor de *destino el valor de fuente tomando en consideración un posible préstamo de una* instrucción **sbb** o **sub** previa y guarda el resultado en *destino*.

Sintaxis:

```
sbb   regB, inmB|regB|memB
sbb   memB, inmB|regB
sbb   regW, inmB|inmW|regW|memW
sbb   memW, inmB|inmW|regW
```

La instrucción **sbb** modifica las siguientes banderas:
sobreflujo **O,** signo **S**, cero **Z,** acarreo auxiliar **A,** paridad **P** y acarreo **C.**

Nota.- Al restar valores multibytes o multipalabras, use **sub** para restar los bytes o palabras menos significativos y después use la instrucción **sbb** para restar los bytes o palabras más significativos tomando en cuenta los posibles préstamos de las restas de los bytes o palabras menos significativos.

VI. cmp *destino, fuente* ;Compara los contenidos de *fuente, y destino*.

Sintaxis:

```
cmp   regB, inmB|regB|memB
cmp   memB, inmB|regB
cmp   regW, inmB|inmW|regW|memW
cmp   memW, inmB|inmW|regW
```

La instrucción **cmp** modifica las siguientes banderas:
sobreflujo **O,** signo **S**, cero **Z,** acarreo auxiliar **A,** paridad **P** y acarreo **C.**

Nota.- Sólo uno de los operandos puede ser una localidad de memoria. La instrucción **cmp** trabaja restando el valor de *fuente de destino y tirando el resultado pero afectando las banderas.*

VII. dec *destino* ;Decrementa en uno el contenido de *destino*.

Sintaxis:

```
dec   regB|memB
dec   regW|memW
```

VIII. mul *fuente* **;**Multiplica dos operandos sin signo.

Sintaxis:

```
mul   regB|memB
mul   regW|memW
```

Si la multiplicación es de un byte por un byte, los operandos y el resultado están en:

Multiplicando: **AL**
Multiplicador: *fuente* (Un byte)
Producto: **AX**

Si la multiplicación es de una palabra por una palabra, los operandos y el resultado están en:

Multiplicando: **AX**
Multiplicador: *fuente* (Una palabra)
Producto: **DX:AX** (La palabra menos significativa en **AX**)

Después de la instrucción **mul** las siguientes banderas quedan en un estado indefinido: signo **S**, cero **Z**, acarreo auxiliar **A** y paridad **P.**
NOTA.- Si las banderas de acarreo **C** y de sobreflujo **O** son ambas **0** después de la operación entonces la parte más significativa del resultado es 0. Si las banderas de acarreo **C** y de sobreflujo **O** son ambas **1** después de la operación entonces el resultado ocupa todo el (los) registro(s) del producto.

IX. imul *fuente* ;Multiplica dos operandos con signo.

Sintaxis:

```
imul regB|memB
imul regW|memW
```

Si la multiplicación es de un byte por un byte, los operandos y el resultado están en:

Multiplicando: **AL**
Multiplicador: *fuente* (Un byte)
Producto: **AX**

Si la multiplicación es de una palabra por una palabra, los operandos y el resultado están en:

Multiplicando: **AX**
Multiplicador: *fuente* (Una palabra)
Producto: **DX:AX** (La palabra menos significativa en **AX**)

Después de la instrucción **imul** las siguientes banderas quedan en un estado indefinido: signo **S**, cero **Z**, acarreo auxiliar **A** y paridad **P.**
NOTA.- Si las banderas de acarreo **C** y de sobreflujo **O** son ambas **0** después de la operación entonces la parte más significativa del resultado es simplemente la extensión del signo de la parte menos significativa. Si las banderas de acarreo **C** y de sobreflujo **O** son ambas **1** después de la operación entonces el resultado ocupa todo el (los) registro(s) del producto.

X. **div** *fuente* ;Divide dos operandos sin signo.

Sintaxis:

```
div   regB|memB
div   regW|memW
```

Si la división es de una palabra entre un byte, los operandos y los resultados están en:

Dividendo: **AX**
Divisor:*fuente* (Un byte)
Cociente: **AL**
Residuo: **AH**

Si la división es de una palabra doble entre una palabra, los operandos y los resultados están en:

Dividendo: **DX:AX** (La palabra menos significativa en **AX**)
Divisor:*fuente* (Una palabra)
Cociente: **AX**
Residuo: **DX**

Después de la instrucción **div** las siguientes banderas quedan en un estado indefinido: sobreflujo **O,** signo **S**, cero **Z**, acarreo auxiliar **A,** paridad **P** y acarreo **C.**

NOTA.- Si el resultado de la división es mayor que el valor máximo permitido en el registro donde se almacena el cociente o si el divisor vale cero, se genera una interrupción tipo 0: división entre 0. El programa se detiene y despliega el mensaje de división entre 0.

XI. idiv *fuente* ;Divide dos operandos con signo.

Sintaxis:

```
idiv regB|memB
idiv regW|memW
```

Si la división es de una palabra entre un byte, los operandos y los resultados están en:

Dividendo: **AX**
Divisor:*fuente* (Un byte)
Cociente: **AL**
Residuo: **AH**

Si la división es de una palabra doble entre una palabra, los operandos y los resultados están en:

Dividendo: **DX:AX** (La palabra menos significativa en **AX**)
Divisor:*fuente* (Una palabra)
Cociente: **AX**
Residuo: **DX**

Después de la instrucción **idiv** las siguientes banderas quedan en un estado indefinido: sobreflujo **O,** signo **S**, cero **Z**, acarreo auxiliar **A,** paridad **P** y acarreo **C.**
NOTA.- Si el resultado de la división es mayor que el valor máximo permitido en el registro donde se almacena el cociente o si el divisor vale cero, se genera una interrupción tipo 0: división entre 0. El programa se detiene y despliega el mensaje de división entre 0.

XII. cbw ;Convierte un **byte** con signo a una **palabra** por extensión de signo.

Sintaxis:

```
cbw
```

La instrucción **cbw** no afecta el estado de las banderas.
NOTA.- Utilice **cbw** para extender un valor de 8 bits con signo en el registro **AL** a un valor de 16 bits con signo de la misma magnitud en **AX.**
La instrucción trabaja copiando el bit más significativo de **AL** a todos los bits de **AH,** esto es, colocando 0FFh en **AH** si AL es negativo o colocando 0h en **AH** si **AL** es positivo.

XIII. cwd ; Convierte una **palabra** con signo a una **palabra doble** por extensión de signo.

Sintaxis:

```
cwd
```

La instrucción **cwd** no afecta el estado de las banderas.
NOTA.- Utilice **cwd** para extender un valor de 16 bits con signo en el registro **AX** a un valor de 32 bits con signo de la misma magnitud en **DX:AX.** La instrucción trabaja copiando el bit más significativo de **AX** a todos los bits de **DX,** esto es, colocando 0FFFFh en **DX** si **AX** es negativo o colocando 0h en **DX** si **AX** es positivo.

XIII. **neg** *destino* ;Obtiene el negativo de *destino*
(forma el complemento de 2)

Sintaxis:

```
neg   regB|memB
neg   regW|memW
```

La instrucción **neg** modifica las siguientes banderas:

sobreflujo **O,** signo **S**, cero **Z**, acarreo auxiliar **A,** paridad **P** y acarreo **C.**

Ejemplos Sobre Instrucciones Aritméticas

Ejemplo 1)

```
;*************************************************************
; SUMABYWS.ASM
;
; Este programa suma una variable de 1 byte con una
; variable de una palabra, ambas signadas. El pseudocódigo
; de este programa es el siguiente:
```

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

        ideal
        dosseg
        model   small
        stack   256

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

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

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

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

        mov     al, [dato1]     ; AL = dato1
        cbw                     ; AX = AL
        add     ax, [dato2]     ; AX += dato2
        mov     [resul], ax     ; resul = AX
salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h

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

        end     inicio
```

Ejemplo 2)

```
;*****************************************************************
; SUMA2DW.ASM
;
; Este programa suma dos variables de tipo palabra doble.
; El resultado queda en una variable tipo palabra doble.
; El pseudocódigo de este programa es el siguiente:
;
;    DX:AX = dato1
;    DX:AX += dato2
;    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]
        add     ax, [word dato2]        ; DX:AX += dato2
        adc     dx, [word dato2+2]
        mov     [word resull], ax       ; resul = DX:AX
        mov     [word result+2], dx
salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h

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

        end     inicio
```

Instrucciones Lógicas

Sistemas Digitales con Microprocesadores.

M.C. Juan Carlos Olguín R.

I. **not** *destino* ;Obtiene el complemento de 1 de *destino*.

Sintaxis:

```
not   regB|memB
not   regW|memW
```

La instrucción **not** deja inalteradas a las banderas.

II. **and** *destino, fuente* ;Calcula la intersección lógica entre los bits de *fuente y destino y guarda el resultado en destino*.

Sintaxis:

```
and   regB,  inmB|regB|memB
and   memB,  inmB|regB
and   regW,  inmB|inmW|regW|memW
and   memW,  inmB|inmW|regW
```

La instrucción **and** modifica las siguientes banderas:

sobreflujo **O** = 0, signo **S**, cero **Z**, paridad **P** y acarreo **C** = **0**.
La bandera de acarreo auxiliar **A** queda en un estado indefinido.

III. test *destino, fuente* **;**Compara valores mediante la intersección lógica entre los bits de *fuente y destino*.

Sintaxis:

```
test regB, inmB|regB|memB
test memB, inmB|regB
test regW, inmW|regW|memW
test memW, inmW|regW
```

La instrucción **test** trabaja en forma idéntica a la instrucción **and** excepto que el resultado de la operación se descarta y sólo se afectan las banderas.

La instrucción **test** modifica las siguientes banderas:

sobreflujo **O** = 0, signo **S**, cero **Z**, paridad **P** y acarreo **C** = 0. La bandera de acarreo auxiliar **A** queda en un estado indefinido.

IV. **or** *destino, fuente* ;Calcula la unión inclusiva lógica entre los bits de *fuente y destino y guarda el resultado en destino*. Sintaxis:

```
or    regB, inmB|regB|memB
or    memB, inmB|regB
or    regW, inmB|inmW|regW|memW
or    memW, inmB|inmW|regW
```

La instrucción **or** modifica las siguientes banderas:
sobreflujo **O** = 0, signo **S**, cero **Z**, paridad **P** y acarreo **C** = 0.
La bandera de acarreo auxiliar **A** queda en un estado indefinido.

V. **xor** *destino, fuente* ;Calcula la unión exclusiva lógica entre los bits de *fuente y destino y guarda el resultado en destino*.

Sintaxis:

```
xor   regB, inmB|regB|memB
xor   memB, inmB|regB
xor   regW, inmB|inmW|regW|memW
xor   memW, inmB|inmW|regW
```

La instrucción **xor** modifica las siguientes banderas:
sobreflujo **O** = 0, signo **S**, cero **Z**, paridad **P** y acarreo **C** = 0.
La bandera de acarreo auxiliar **A** queda en un estado indefinido.

sal | shl *destino, cuenta*

;Corre hacia la izquierda los bits de *destino un número de veces dado por cuenta*.

Sintaxis:

```
sal|shl   regB|memB, 1|cl
sal|shl   regW|memW, 1|cl
```

Los mnemónicos **sal y shl** son sinónimos. **sal|shl** corre hacia la izquierda los bits de *destino*. *Esto es,* en cada corrimiento el bit más significativo de *destino se mueve a la bandera de acarreo y se inserta un* cero como el bit menos significativo de *destino, mientras todos los demás bits se mueven una posición* a la izquierda.

El mnemónico **sal** se emplea para multiplicar *destino por potencias de 2, mientras que el* mnemónico **shl** se emplea para correr los bits de *destino a la izquierda. Aunque ambos son* intercambiables.

sar *destino, cuenta*

Corre hacia la derecha los bits de *destino, preservando el signo, un número de veces dado por cuenta*.

Sintaxis:

```
sar   regB|memB, 1|cl
sar   regW|memW, 1|cl
```

A diferencia de los mnemónicos **sal y shl, sar y shr** no son sinónimos. **sar corre hacia la derecha los** bits de *destino preservando el signo*. *Esto es, en cada corrimiento una copia del bit más significativo de destino se inserta en el bit más significativo*. *El bit menos significativo se mueve a la bandera de acarreo*, mientras todos los demás bits se mueven una posición a la derecha. La instrucción **sar** se emplea para dividir un valor signado en *destino entre potencias de 2*.

Si el número de corrimientos deseado es de uno, *cuenta puede ser el valor inmediato 1*. *Para otros* valores, el número de corrimientos debe asignarse previamente al registro **CL** y especificar éste como *cuenta*.

shr *destino, cuenta*
;Corre hacia la derecha los bits de *destino un número de veces dado por cuenta*.

Sintaxis:

```
shr   regB|memB, 1|cl
shr   regW|memW, 1|cl
```

shr corre hacia la derecha los bits de *destino*. *Esto es, en cada corrimiento se inserta un cero en el bit* más significativo de *destino*. *El bit menos significativo se mueve a la bandera de acarreo, mientras todos* los demás bits se mueven una posición a la derecha.

La instrucción **shr** se emplea para dividir un valor sin signo en *destino entre potencias de 2*.

Si el número de corrimientos deseado es de uno, *cuenta puede ser el valor inmediato 1*.
Para otros valores, el número de corrimientos debe asignarse previamente al registro **CL** y especificar éste como *cuenta*.

rcl *destino, cuenta*
Rota hacia la izquierda los bits de *destino a través de la bandera de acarreo un número de veces dado* por *cuenta*.

Sintaxis:

```
rcl   regB|memB,  1|cl
rcl   regW|memW,  1|cl
```

rcl rota hacia la izquierda los bits de ***destino*** *incluyendo a la bandera de acarreo* ***C*** *como parte del* valor original. Esto es, en cada rotación el bit más significativo de *destino se mueve a la bandera de* acarreo y el valor en la bandera de acarreo se inserta como el bit menos significativo de *destino,* mientras todos los demás bits se mueven una posición a la izquierda.
Si el número de rotaciones deseado es de uno, *cuenta puede ser el valor inmediato 1*.
Para otros valores, el número de rotaciones debe asignarse previamente al registro **CL** y especificar éste como *cuenta*.
La instrucción **rcl** modifica la bandera de acarreo **C.** Si *cuenta es un uno inmediato, la instrucción* también modifica a la bandera de sobreflujo **O,** si *cuenta es un valor en el registro* ***CL,*** *la bandera de* sobreflujo **O** queda en un estado indefinido.

rcr *destino, cuenta*
;Rota hacia la derecha los bits de *destino a través de la bandera de acarreo un número de veces dado* por *cuenta*.

Sintaxis:

```
rcr   regB|memB, 1|cl
rcr   regW|memW, 1|cl
```

rcr rota hacia la derecha los bits de *destino incluyendo a la bandera de acarreo* ***C*** *como parte del valor* original. Esto es, en cada rotación el bit menos significativo de *destino se mueve a la bandera de* acarreo y el valor en la bandera de acarreo se inserta como el bit más significativo de *destino, mientras* todos los demás bits se mueven una posición a la derecha.
Si el número de rotaciones deseado es de uno, *cuenta puede ser el valor inmediato 1. Para otros* valores, el número de rotaciones debe asignarse previamente al registro **CL** y especificar éste como *cuenta*.
La instrucción **rcr** modifica la bandera de acarreo **C.** Si *cuenta es un uno inmediato, la instrucción* también modifica a la bandera de sobreflujo **O,** si *cuenta es un valor en el registro* ***CL,*** *la bandera de* sobreflujo **O** queda en un estado indefinido.

rol *destino, cuenta*

Rota hacia la izquierda los bits de *destino un número de veces dado por cuenta*.

Sintaxis:

```
rol   regB|memB,  1|cl
rol   regW|memW,  1|cl
```

rol rota hacia la izquierda los bits de *destino*. *Esto es, en cada rotación el bit más significativo de destino se inserta como el bit menos significativo de destino, mientras todos los demás bits se mueven* una posición a la izquierda.
Adicionalmente el bit más significativo del valor original de *destino se copia* a la bandera de acarreo **C.**
Si el número de rotaciones deseado es de uno, *cuenta puede ser el valor inmediato 1. Para otros* valores, el número de rotaciones debe asignarse previamente al registro **CL** y especificar éste como *cuenta*.

ror *destino, cuenta*

Rota hacia la derecha los bits de *destino un número de veces dado por cuenta*.

Sintaxis:

```
ror   regB|memB,  1|cl
ror   regW|memW,  1|cl
```

ror rota hacia la derecha los bits de *destino*. *Esto es, en cada rotación el bit menos significativo de destino se inserta como el bit más significativo de destino, mientras todos los demás bits se mueven una* posición a la derecha.
Adicionalmente el bit menos significativo del valor original de *destino se copia a* la bandera de acarreo **C.**
Si el número de rotaciones deseado es de uno, *cuenta puede ser el valor inmediato 1. Para otros* valores, el número de rotaciones debe asignarse previamente al registro **CL** y especificar éste como *cuenta*.

La instrucción **ror** modifica la bandera de acarreo **C.** Si *cuenta es un uno inmediato, la instrucción* también modifica a la bandera de sobreflujo **O,** si *cuenta es un valor en el registro **CL,** la bandera de* sobreflujo **O** queda en un estado indefinido.

Ejemplos

```
;*************************************************************
; BIT.ASM
;
; Este programa determina el valor del i-ésimo bit de una
; variable de un byte. Si el bit vale 0 deja un cero en la
; variable resul. Si el bit vale 1 deja un valor diferente
; de cero. El pseudocódigo de este programa es el
; siguiente:
;
;    AH = dato
;    AL = 1 << CL
;    AH = AH & AL
;    resul = AH
;*************************************************************

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

         ideal
         dosseg
         model small
         stack   256

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

         dataseg
codsal   db      0
dato     db      ?
bit      db      ?
resul    db      ?

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

         codeseg
inicio:
         mov     ax, @data                ; Inicializa el
```

```
        mov     ds, ax                          ; segmento de datos

        mov     ah, [dato]                      ; AH = dato
        mov     al, 1                           ; Al = 1
        mov     cl, [bit]                       ; CL = bit
        shl     al, cl                          ; AL <= CL
        and     ah, al                          ; AH &= Al
        mov     [resul], ah                     ; resul = AH

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

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

        end     inicio
```

2. El siguiente programa implementa todas las rotaciones con una variable de tipo palabra doble.

```
;*************************************************************
; ROTACION.ASM
;
; Este programa implementa todas las rotaciones con una
; variable de tipo palabra doble. El pseudocódigo de este
; programa es el siguiente:
;
;    DX:AX = dato
;    DX:AX = rcl DX:AX
;    r_rcl = DX:AX
;
;    DX:AX = dato
;    DX:AX = rcr DX:AX
;    r_rcr = DX:AX
;
;    DX:AX = dato
;    DX:AX = rol DX:AX
;    r_rol = DX:AX
;
;    DX:AX = dato
;    DX:AX = ror DX:AX
;    r_ror = DX:AX
;*************************************************************

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

        ideal
        dosseg
        model   small
        stack   256
```

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

        dataseg
codsal  db      0
dato    dd      ?
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
        mov     ax, [word dato]         ; DX:AX = dato
        mov     dx, [word dato+2]

        rcl     ax, 1                   ; rcl DX:AX, 1
        rcl     dx, 1

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

; Rotación a la derecha a través de la bandera de acarreo
        mov     ax, [word dato]         ; DX:AX = dato
        mov     dx, [word dato+2]

        rcr     dx, 1                   ; rcr DX:AX, 1
        rcr     ax, 1

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

```
; Rotación a la izquierda
        mov     ax, [word dato]         ; DX:AX = dato
        mov     dx, [word dato+2]

        mov     bx, dx                  ; rol DX:AX, 1
        shl     bx, 1
        rcl     ax, 1
        rcl     dx, 1

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

; Rotación a la derecha
        mov     ax, [word dato]         ; DX:AX = dato
        mov     dx, [word dato+2]
```

```
        mov     bx, ax                  ; ror DX:AX, 1
        shr     bx, 1
        rcr     dx, 1
        rcr     ax, 1

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

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

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

        end     inicio
```