LENGUAJE ENSAMBLADOR DEL MICROPROCESADOR

REGISTROS INTERNOS DEL MICROPROCESADOR

La ***U***nidad ***C***entral de ***P***roceso (***CPU***, *por sus siglas en inglés*) tiene **14** ***registros internos*** cada uno de **16** bits. Los primeros cuatro, ***AX***, ***BX***, ***CX*** y ***DX***, son de uso general y se pueden usar también como registros de **8** bits. Es decir, ***AX*** se puede dividir en ***AH*** y ***AL*** (***AH*** es el byte alto, ***high***, y ***AL*** es el byte bajo, ***low***) Lo mismo es aplicable a los otros tres (***BX*** en ***BH*** y ***BL***, ***CX*** en ***CH*** y ***CL*** y ***DX*** en ***DH*** y ***DL***)

Estos son los únicos registros que pueden usarse de modo dual (en **8** o **16** bits) Los ***registros*** de la ***CPU*** son conocidos por sus nombres propios, que son:

- ***AX*** (acumulador)
- ***BX*** (registro base)
- ***CX*** (registro contador)
- ***DX*** (registro de datos)
- ***DS*** (registro del segmento de datos)
- ***ES*** (registro del segmento extra)
- ***SS*** (registro del segmento de pila)
- ***CS*** (registro del segmento de código)
- ***BP*** (registro de apuntadores base)
- ***SI*** (registro índice fuente)
- ***DI*** (registro índice destino)
- ***SP*** (registro del apuntador de pila)
- ***IP*** (registro del apuntador de siguiente instrucción)
- ***F*** (registro de banderas)

El registro ***AX*** se usa para almacenar resultados, lectura o escritura desde o hacia los puertos. El ***BX*** sirve como apuntador base o índice. El ***CX*** se utiliza en operaciones de iteración, como un contador que automáticamente se incrementa o decrementa de acuerdo con el tipo de instrucción usada. El ***DX*** se usa como puente para el acceso de datos.

El ***DS*** es un registro de segmento cuya función es actuar como policía donde se encuentran los datos. Cualquier dato, ya sea una variable inicializada o no, debe estar dentro de este segmento. La única excepción es cuando tenemos programas del tipo ****.com***, ya que en éstos sólo puede existir un segmento. El registro ***ES*** tiene el propósito general de permitir operaciones sobre cadenas, pero también puede ser una extensión del ***DS***.

El ***SS*** tiene la tarea exclusiva de manejar la posición de memoria donde se encuentra la pila (***stack***) Esta es una estructura usada para almacenar datos en forma temporal, tanto de un programa como de las operaciones internas de la computadora personal (***PC***, *por sus siglas en inglés*) En términos de operación interna, la ***CPU*** usa este segmento para almacenar las direcciones de retorno de las llamadas a rutinas. El registro de segmentos más importante es el ***CS*** o segmento de código. Es aquí donde se encuentra el código ejecutable de cada programa, el cual está directamente ligado a los diferentes modelos de memoria.

El registro ***BP*** (***base pointer***) se usa para manipular la pila sin afectar al registro de segmentos ***SS***. Es útil cuando se usa interfaz entre lenguajes de alto nivel y el

ensamblador. Puesto que dicha interfaz se basa en el concepto de la pila ***BP***, nos permite acceder parámetros pasados sin alterar el registro de segmento ***SS***. Los registros ***SI*** y ***DI*** son útiles para manejar bloques de cadenas en memoria, siendo el primero el índice fuente y el segundo el índice destino. En otras palabras, ***SI*** representa la dirección donde se encuentra la cadena y ***DI*** la dirección donde será copiada.

El registro ***SP*** apunta a un área específica de memoria que sirve para almacenar datos bajo la estructura ***LIFO*** (último en entrar, primero en salir), conocida como pila (***stack***) El registro ***IP*** (***instruction pointer***) apunta a la siguiente instrucción que será ejecutada en memoria.

A continuación se describe el significado de cada bit del registro ***F*** (**b**anderas)

Todas las ***banderas apagadas***:

NV ***UP*** ***DI*** ***PL*** ***NZ*** ***NA*** ***PO*** ***NC***

Todas las ***banderas prendidas***:

OV ***DN*** ***EI*** ***NG*** ***ZR*** ***AC*** ***PE*** ***CY***

Significado de los bits:

- ***Overflow*** — ***NV*** = no hay desbordamiento; ***OV*** = Sí lo hay
- ***Direction*** — ***UP*** = hacia adelante; ***DN*** = hacia atrás
- ***Interrupts*** — ***DI*** = desactivadas; ***EI*** = activadas
- ***Sign*** — ***PL*** = positivo; ***NG*** = negativo
- ***Zero*** — ***NZ*** = no es cero; ***ZR*** = sí lo es
- ***Auxiliary Carry*** — ***NA*** = no hay acarreo auxiliar; ***AC*** = hay acarreo auxiliar
- ***Parity*** — ***PO*** = paridad non; ***PE*** = paridad par
- ***Carry*** — ***NC*** = no hay acarreo; ***CY*** = sí lo hay

El registro de **b**anderas es un registro de **16** bits, pero no todos los bits se usan.

PSW Contiene **9** banderas. **Tres** banderas de control ***TF***, ***DF***, ***IF*** y **seis** banderas de status ***CF***, ***PF***, ***AF***, ***ZF***, ***SF***, ***OF***.

Estas **6** últimas banderas representan el resultado de una operación ***aritmética*** o ***lógica***. Permiten al programa alterar el curso de ejecución basado en los valores lógicos que almacenan.

- ***AF*** ***Llevar auxiliar* = 1**, indica que hubo "***llevar***" del *nibble* (**4** bits) **0** al *nibble* **1**. O un "***pedir préstamo***" del *nibble* **alto** al *nibble* **bajo**.
- ***CF*** ***Llevar* = 1**, cuando ha ocurrido un "***llevar***" o "***pedir préstamo***" del resultado (**8** o **16** bits)
- ***OF*** ***Sobreflujo* = 1**, indica que ha ocurrido un sobreflujo aritmético. Esto significa que el tamaño del resultado excede la capacidad de **ALMACENAMIENTO** del destino y el dígito significativo se perdió.
- ***SF*** ***Signo***. Esta bandera se activa cuando el bit más significativo del resultado es **1.** Ya que los números binarios negativos son representados usando notación C_2, ***SF*** refleja el signo del resultado:
  0 indica **+**
  1 indica **-**
- ***PF*** ***Paridad***. Cuando esta bandera está activa, el resultado de la operación tiene un número par de unos. Esta bandera se usa para verificar errores en la transmisión.
- ***ZF*** ***Cero***. Esta bandera se activa cuando el resultado de la operación es cero.

Las **tres** banderas de control serán discutidas después durante el curso

- ***DF*** = bandera de dirección
- ***IF*** = bandera de interrupción
- ***TF*** = bandera de trampa

CONJUNTO DE INSTRUCCIONES (Microprocesadores *8086*/*8088*)

Se pueden clasificar en los siguientes grupos:

Instrucciones de *Transferencia de Datos*.

Estas instrucciones mueven datos de una parte a otra del sistema; desde y hacia la memoria principal, de y a los registros de datos, puertos de ***E/S*** y registros de segmentación.

Las instrucciones de transferencia de datos son las siguientes:

- ***MOV*** transfiere
- ***XCHG*** intercambia
- ***IN*** entrada
- ***OUT*** salida
- ***XLAT*** traduce usando una tabla
- ***LEA*** carga la dirección efectiva
- ***LDS*** carga el segmento de datos
- ***LES*** carga el segmento extra
- ***LAHF*** carga los indicadores en ***AH***
- ***SAHF*** guarda ***AH*** en los indicadores
- ***PUSH*** **FUENTE** (***sp***) ← ***fuente***
- ***POP*** **DESTINO** ***destino*** ← (***sp***)

Control de *Bucles* (instrucciones simples)

Éstas posibilitan el grupo de control más elemental de nuestros programas. Un bucle es un bloque de código que se ejecuta varias veces. Hay **4** tipos de bucles básicos:

- Bucles ***sin fin***
- Bucles por ***conteo***
- Bucles ***hasta***
- Bucles ***mientras***

Las instrucciones de control de bucles son las siguientes:

INC	**incrementar**
DEC	**decrementar**
LOOP	**realizar** un bucle
LOOPZ,***LOOPE***	**realizar** un bucle ***si es cero***
LOOPNZ,***LOOPNE***	**realizar** un bucle ***si no es cero***
JCXZ	**salta** si ***CX*** es ***cero***

Instrucciones de *Prueba, Comparación y Saltos*.

Este grupo es una continuación del anterior, incluye las siguientes instrucciones:

TEST	**verifica**
CMP	**compara**
JMP	**salta**
JE, ***JZ***	**salta** si ***es igual a cero***
JNE, ***JNZ***	**salta** si ***no igual a cero***
JS	**salta** si ***signo negativo***
JNS	**salta** si ***signo no negativo***
JP, ***JPE***	**salta** si ***paridad par***
JNP, ***JOP***	**salta** si ***paridad impar***
JO	**salta** si ***hay capacidad excedida***
JNO	**salta** si ***no hay capacidad excedida***
JB, ***JNAE***	**salta** si ***por abajo*** (no encima o igual)
JNB, ***JAE***	**salta** si ***no está por abajo*** (encima o igual)
JBE, ***JNA***	**salta** si ***por abajo o igual*** (no encima)
JNBE, ***JA***	**salta** si ***no por abajo o igual*** (encima)
JL, ***JNGE***	**salta** si ***menor que*** (no mayor o igual)
JNL, ***JGE***	**salta** si ***no menor que*** (mayor o igual)
JLE, ***JNG***	**salta** si ***menor que o igual*** (no mayor)
JNLE, ***JG***	**salta** si ***no menor que o igual*** (mayor)

Instrucciones de *Llamado y Retorno de Subrutinas*.

Para que los programas resulten eficientes y legibles tanto en lenguaje ensamblador como en lenguaje de alto nivel, resultan indispensables las subrutinas:

CALL	***llamada*** a subrutina
RET	***retorno*** al programa o subrutina que llamó

Instrucciones *Aritméticas*.

Estas instrucciones son las que realiza directamente el ***8086/8088***

a. *Grupo de adición:*

- ***ADD*** **suma**
- ***ADC*** **suma** con acarreo
- ***AAA*** ajuste **ASCII** para la **suma**
- ***DAA*** ajuste decimal para la **suma**

b. *Grupo de sustracción:*

- ***SUB*** **resta**
- ***SBB*** **resta** con acarreo negativo
- ***AAS*** ajuste **ASCII** para la **resta**
- ***DAS*** ajuste decimal para la **resta**

c. *Grupo de multiplicación:*

- ***MUL*** **multiplicación**
- ***IMUL*** **multiplicación** entera
- ***AAM*** ajuste **ASCII** para la **multiplicación**

d. *Grupo de división:*

- ***DIV*** **división**
- ***IDIV*** **división** entera
- ***AAD*** ajuste **ASCII** para la **división**

e. *Conversiones:*

- ***CBW*** **pasar** octeto a palabra
- ***CWD*** **pasar** palabra a doble palabra
- ***NEG*** **negación**

f. *Tratamiento de cadenas:*

Permiten el movimiento, comparación o búsqueda rápida en bloques de datos:

- ***MOVC*** **transferir** ***carácter*** de una cadena
- ***MOVW*** **transferir** ***palabra*** de una cadena
- ***CMPC*** **comparar** ***carácter*** de una cadena
- ***CMPW*** **comparar** ***palabra*** de una cadena
- ***SCAC*** **buscar** ***carácter*** de una cadena
- ***SCAW*** **buscar** ***palabra*** de una cadena
- ***LODC*** **cargar** ***carácter*** de una cadena
- ***LODW*** **cargar** ***palabra*** de una cadena
- ***STOC*** **guardar** ***carácter*** de una cadena
- ***STOW*** **guardar** ***palabra*** de una cadena

- ***REP*** **repetir**
- ***CLD*** **poner** a **0** el indicador de dirección
- ***STD*** **poner** a **1** el indicador de dirección

Instrucciones *Lógicas*.

Son operaciones **bit** a **bit** que trabajan sobre **octetos** o **palabras** completas:

- ***NOT*** **negación**
- ***AND*** **producto** lógico
- ***OR*** **suma** lógica
- ***XOR*** **suma** lógica exclusiva

Instrucciones de *Desplazamiento, Rotación y Adeudos*.

Básicamente permiten ***multiplicar*** y ***dividir*** por potencias de **2**

- ***SHL***, ***SAL*** **desplazar** a la ***izquierda*** (desplazamiento aritmético)
- ***SHR*** **desplazar** a la ***derecha***
- ***SAR*** **desplazamiento** aritmético a la ***derecha***
- ***ROL*** **rotación** a la ***izquierda***
- ***ROR*** **rotación** a la ***derecha***
- ***RCL*** **rotación** con acarreo a la ***izquierda***
- ***RCR*** **rotación** con acarreo a la ***derecha***
- ***CLC*** **borrar** acarreo
- ***STC*** **poner** acarreo a **1**

Instrucciones de *Pila*.

Una de las funciones de la pila del sistema es la de salvaguardar (conservar) datos (la otra es la de salvaguardar las direcciones de retorno de las llamadas a subrutinas):

- ***PUSH*** **introducir**
- ***POP*** **extraer**
- ***PUSHF*** **introducir** indicadores
- ***POPF*** **extraer** indicadores

Instrucciones de *Control del microprocesador*.

Hay varias instrucciones para el control de la ***CPU***, ya sea a ella sola, o en conjunción con otros procesadores:

- ***NOP*** **no operación**
- ***HLT*** **parada**
- ***WAIT*** **espera**
- ***LOCK*** **bloquea**
- ***ESC*** **escape**

Instrucciones de *Interrupción*.

- ***STI*** **poner** a **1** el indicador de interrupción
- ***CLI*** **borrar** el indicador de interrupción
- ***INT*** **interrupción**
- ***INTO*** **interrupción** por capacidad excedida (desbordamiento)
- ***IRET*** **retorno** de interrupción

Las instrucciones de ***transferencia condicional del control*** del programa se pueden clasificar en **3** grupos:

1. Instrucciones usadas para comparar dos enteros sin signo:

a. ***JA*** o ***JNBE***. **Salta** ***si está arriba*** o **salta** ***si no está abajo o si no es igual*** (*jump if above* o *jump if not below or equal*) El salto se efectúa si la bandera ce ***CF*** **= 0** o si la bandera de *ZF* **= 0**
b. ***JAE*** o ***JNB***. **Salta** ***si está arriba o es igual*** o **salta** ***si no está abajo*** (*jump if above or equal* o *jump if not below*) El salto se efectúa si ***CF*** **= 0**.
c. ***JB*** o ***JNAE***. **Salta** ***si está abajo*** o **salta** ***si no está arriba o si no es igual*** (*jump if below or equal* o *jump if not above or equal*) El salto se efectúa si ***CF*** **= 1**.
d. ***JBE*** o ***JNA***. **Salta** ***si está abajo o si es igual*** o **salta** ***si no está arriba*** (*jump if below or equal* o *jump if not above*) El salto se efectúa si ***CF*** **= 1**.
e. ***JE*** o ***JZ***. **Salta** ***si es igual*** o **salta** ***si es cero*** (*jump equal* o *jump if zero*) El salto se efectúa si *ZF* **= 1** (también se aplica a comparaciones de enteros con signo)
f. ***JNE*** o ***JNZ***. **Salta** ***si no es igual*** o **salta** ***si no es cero*** (*jump if not equal* o *jump if not zero*) El salto se efectúa si *ZF* **= 0** (también se aplica a comparaciones de enteros con signo)

2. Instrucciones usadas para comparar dos enteros con signo:

a. ***JG*** o ***JNLE***. **Salta** ***si es más grande*** o **salta** ***si no es menor o igual*** (*jump if greater* o *jump if not less or equal*) El salto se efectúa si *ZF* **= 0** o ***OF*** **=** ***SF***.
b. ***JGE*** o ***JNL***. **Salta** ***si es más grande o igual*** o **salta** ***si no es menor que*** (*jump if greater or equal* o *jump if not less*) El salto se efectúa si ***SF*** **=** ***OF***.
c. ***JL*** o ***JNGE***. **Salta** ***si es menor que*** o **salta** ***si no es mayor o igual*** (*jump if less* o *jump if not greater or equal*) El salto se efectúa si ***SF*** **=** ***OF***.
d. ***JLE*** o ***JNG***. **Salta** ***si es menor o igual*** o **salta** ***si no es más grande*** (*jump if less or equal* o *jump if not greater*) El salto se efectúa si *ZF* **= 1** o ***SF*** **=** ***OF***.

3. Instrucciones usadas según el estado de banderas:

a. ***JC*** **Salta** ***si hay acarreo*** (*jump if carry*) El salto se efectúa si ***CF*** **= 1**.
b. ***JNC*** **Salta** ***si no hay acarreo*** (*jump if not carry*) El salto se efectúa si ***CF*** **= 0**.
c. ***JNO*** **Salta** ***si no hay desbordamiento*** (*jump if not overflow*) El salto se efectúa si ***OF*** **= 0**.
d. ***JNP*** o ***JPO*** **Salta** ***si no hay paridad*** o **salta** ***si la paridad en non***. El salto se efectúa si ***PF*** **= 0**.
e. ***JNS*** **Salta** ***si el signo está apagado*** (*jump if not sign*) El salto se efectúa si *SF* **= 0**.
f. ***JO*** **Salta** ***si hay desbordamiento*** (*jump if overflow*) El salto se efectúa si ***OF*** **= 1**.
g. ***JP*** o ***JPE*** **Salta** ***si hay paridad*** o **salta** ***si la paridad es par*** (*jump if parity* o *jump if parity even*) El salto se efectúa si ***PF*** **= 1**.
h. ***JS*** **Salta** ***si el signo está prendido*** (*jump if sign set*) El salto se efectúa si *SF* **= 1**.

Las comparaciones con signo van de acuerdo con la interpretación que usted le quiera dar a los bytes o palabras de su programa. Por ejemplo, suponga que tiene un byte cuyo valor es **11111111** en binario y que desea compararlo con otro cuyo valor es **00000000**. ¿Es **11111111** mayor que **00000000**? ***SÍ*** y ***NO***, eso depende de la interpretación que usted le quiera dar. Si trabaja con números enteros ***sin signo*** ***SÍ LO SERÁ***, pues **255** es mayor que **0**. Por el contrario, si tiene ***signo*** entonces ***SERÁ MENOR*** puesto que **–1** es siempre menor que **0**.

Lo anterior lleva a seleccionar las instrucciones de comparación y de salto de acuerdo con la interpretación que se les dé a los bytes o palabras; reflexione sobre este punto.

Los saltos condicionales se encuentran limitados al rango de **–128** a **+127** bytes como máxima distancia, ya sea adelante o hacia atrás. Si desea efectuar un salto a mayores distancias es necesario crear una condición mixta entre saltos condicionales y no condicionales.

Iteraciones.

Con los saltos condicionales y no condicionales se pueden crear estructuras de iteración bastante complejas, aunque existen instrucciones específicas para ello tal como ***loop***.

Esta instrucción es muy útil cuando se va a efectuar cierto bloque de instrucciones un número finito de veces. He aquí un ejemplo:

```
CUENTA:     DW, 100
.
.
.
MOV         CX, CUENTA
ITERA:
.
.
LOOP        ITERA
```

El bloque de instrucciones que se encuentra entre la etiqueta ***ITERA*** y la instrucción ***loop*** será ejecutado hasta que el registro ***CX*** sea igual a **0**. Cada vez que se ejecuta la instrucción ***loop***, el registro ***CX*** es decrementado en **1** hasta llegar a **0**. Esta instrucción tiene la limitante de que debe encontrarse en el rango de **+128** a **–127** (máximo número de bytes entre ***ITERA*** y ***loop***)

Iteraciones condicionales

Existen otras dos variantes de la instrucción ***loop***. Las instrucciones ***loope*** y ***loopz*** decrementan ***CX*** e iteran si ***CX*** **=** **0** y ***ZF*** **=** **1**, mientras que ***loopne*** y ***looppnz*** iteran si ***CX*** **≠** **0** y ***ZF*** **≠** **0**. Un punto importante es que al decrementarse ***CX*** las banderas **NO RESULTAN AFECTADAS**. Por lo tanto, le corresponde a usted afectarlas dentro del bloque de iteración.

FORMATO DE LAS INSTRUCCIONES

Cada instrucción en lenguaje ensamblador del ***8088*** está compuesta de **4** campos:

etiqueta **operación** **operando** **comentario**

El campo comentario se utiliza para propósitos de documentación y es opcional.

Campo ***etiqueta***: Una ***etiqueta*** debe comenzar con un carácter alfabético y puede contener hasta **31** caracteres, incluyendo:

- Letras de la ***A*** a la ***Z***
- Números del ***0*** al ***9***
- Los símbolos especiales: **- $.** @ **%**

No se puede utilizar un nombre que coincida con una palabra reservada o directiva del ensamblador. Si el nombre incluye un ***punto***, entonces el ***punto*** debe ser el primer carácter.

Campo ***operación***: Contiene el ***nemotécnico*** de la instrucción, que es de **2** a **6** caracteres.

Campo ***operando***: Contiene la posición o posiciones donde están los ***datos*** que van a ser manipulados por la instrucción.

Campo ***comentario***: Se utiliza para ***documentar*** el código fuente del ensamblador. Debe separarse del último campo por al menos un espacio e iniciar con **;**.

Cuando inicia un ***comentario*** en una línea ésta deberá tener en la primera columna el carácter **;**.

MODOS DE DIRECCIONAMIENTO Y GENERACIÓN DEL CÓDIGO OBJETO

Generación de la dirección de la instrucción.

Todos los registros internos del ***8086/8088*** son de **16** bits. El bus de ***dirección*** es de **20** bits, por lo que se usa más de un registro interno para generar la ***dirección*** de **20** bits.

Los **2** registros usados para la ***dirección*** de la instrucción son el ***IP*** y el ***CS***. Se combinan en una forma especial para generar la ***dirección*** de **20** bits.

dirección **de 20 bits =** 16_{10} * ***CS*** + ***IP***

Por **ejemplo**: Si los registros ***CS*** e ***IP*** contienen los valores:

CS **= 1000H**
IP **= 0414 H**

La ***dirección*** de **20** bits es:

$$16_{10} * 1000H + 0414H = 10000H + 0414H = 10414H$$

Esta es la ***dirección*** en memoria desde la cual la nueva instrucción debe buscarse.

Al registro ***IP*** se le refiere como ***offset***, el registro ***CS*** * 16_{10} apunta a la ***dirección*** de inicio o segmento en memoria desde el cual se calcula el ***offset***. La ***Figura A*** muestra gráficamente cómo se calcula la ***dirección*** de **20** bits.

***FIGURA A*. Cálculo de la *dirección* de 20 bits**

Cada ***dirección*** generada por el ***8086/8088*** usa uno de los **4** registros de segmento. Este registro de segmento es recorrido **4** bits hacia la izquierda antes de ser sumado al ***offset***.

La instrucción del ***CPU*** especifica cuáles registros internos se usan para generar el ***offset***.

Vamos a ver los diferentes modos de direccionamiento tomando como ejemplo la instrucción ***MOV***.

Instrucción *MOV*

Transfiere un byte desde el ***operando fuente*** al ***operando destino***. Tiene el siguiente formato:

```
MOV     destino, fuente
```

Direccionamiento *Inmediato*

El ***operando fuente*** aparece en la instrucción. Un **ejemplo**, es el que mueve un valor constante a un registro interno.

```
MOV     AX, 568
```

Direccionamiento a *Registro*

Indica que el ***operando*** a ser usado está contenido en uno de los registros internos de propósito general del ***CPU***. En el caso de los registros ***AX***, ***BX***, ***CX*** o ***DX*** los registros pueden ser de **8** a **16** bits

Ejemplos:

```
MOV     AX, BX          ; AX ← BX
MOV     AL, BL          ; AL ← BL
```

Cuando usamos direccionamiento a registro, el ***CPU*** realiza las operaciones internamente, es decir, no se genera ***dirección*** de **20** bits para especificar el ***operando fuente***.

Direccionamiento *Directo*

Especifica en la instrucción la localidad de memoria que contiene al ***operando***. En este tipo de direccionamiento, se forma una ***dirección*** de **20** bits.

Ejemplo:

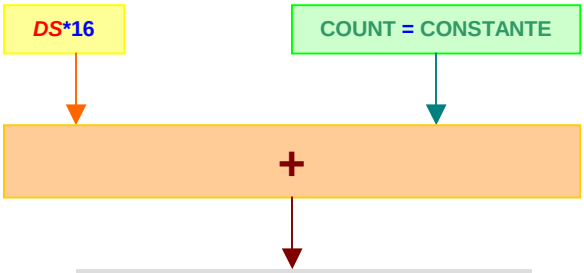
```
MOV     CX, COUNT
```

El valor de ***COUNT*** es una constante. Es usada como el valor ***offset*** en el cálculo de la ***dirección*** de **20** bits

El ***8086/8088*** siempre usa un registro de segmento cuando calcula una ***dirección*** física.

¿**Cuál registro se debe usar para esta instrucción? Respuesta**: ***DS***

En la ***Figura B***, se muestra el cálculo de la ***dirección*** desde la cual se tomará el dato que se carga en ***CX***.



Cuando accedamos datos, el registro *DS* se usa con un *offset* para calcular la dirección de 20 bits, éste es el segmento por omisión. Puede ser invalidado por usar un prefijo de segmento en la instrucción.
Ejemplo:

```
MOV     CX, ES: COUNT
```

***FIGURA B*. Uso del segmento de datos y una constante para desplazamiento**

Este es el segmento por omisión que se usa. Sin embargo, cualquiera de los **4** segmentos puede usarse. Esto se efectúa especificando el registro apropiado en la instrucción.

Por **ejemplo**, suponga que deseamos usar el registro ***ES*** en lugar del ***DS***:

```
MOV     CX, ES: COUNT
```

Direccionamiento de *Registro Indirecto*

Con el modo de direccionamiento de registro índice, la ***dirección*** *offset* de **16** bits está contenida en un registro base o registro índice. Esto es, la ***dirección*** reside en el registro ***BX***, ***BP***, ***SI*** o ***DI***.

Ejemplo:

```
MOV     AX, [SI]
```

El valor de **16** bits contenido en el registro ***SI*** debe ser el *offset* usado para calcular la ***dirección*** de **20** bits.

Otra vez, debe usarse un registro de segmento para generar la ***dirección*** final. El valor de **16** bits en ***SI*** se combina con el segmento apropiado para generar la ***dirección***.

Direccionamiento de *Registro Indirecto con Desplazamiento*

Este tipo de direccionamiento incluye a los dos modos de direccionamiento anteriores. La ***dirección*** *offset* de **16** bits se calcula sumando el valor de **16** bits especificado en un registro interno y una constante.

Por **ejemplo**, si usamos el registro interno ***DI*** y el valor constante (desplazamiento), donde ***COUNT*** ha sido previamente definido, el ***nemotécnico*** para esta construcción es:

```
MOV     AX, COUNT [DI]
```

```
Si:     COUNT =     0378H
          DI =      04FAH
                    0872H
```

Entonces, la ***dirección*** *offset* de **16** bits es **0872H**

Direccionamiento de *Registro Indirecto con un Registro Base y un Registro Índice*

Este modo de direccionamiento usa la suma de dos registros internos para obtener la ***dirección*** *offset* de **16** bits a usarse en el cálculo de la ***dirección*** de **20** bits.

Ejemplos:

```
MOV     [BP] [DI], AX       ; el offset es BP + DI
MOV     AX, [BX] [SI]       ; el offset es BX + SI
```

Direccionamiento de Registro Índice Indirecto con un Registro Base, un Registro Índice y un Registro Constante

Este es el modo de direccionamiento más complejo. Es idéntico al modo de direccionamiento anterior, excepto que se suma una constante.

Ejemplo: Suponga que tenemos los siguientes valores en los registros:

```
DI =        0367H
BX =        7890H
COUNT =     0012H
            7C09H
```

Este modo de direccionamiento indica que el *offset* especificado por la suma de *DI* + *BX* + *COUNT* sea usado para mover el dato en memoria en el registro *AX*.

MOV AX, COUNT [BX] [DI]

La *dirección offset* de **16** bits es 7C09H. La *dirección* completa en **20** bits se calcula de la expresión:

16_{10}*DS + 7C09H

Si el *DS* contiene **3000H**, la dirección completa de **20** bits es:

3000H + 7C09H = 37C09H

Código Objeto del *8086*/*8088*

Como programador, debes escribir los *nemotécnicos*. El *código objeto* es generado por la computadora (son los bytes que ejecuta el *CPU*) Con el conjunto de instrucciones del *8086*/*8088*, cada tipo de modo de direccionamiento puede requerir un número diferente de bytes. En los ejemplos siguientes proporcionaremos el número de bytes requeridos por cada modo de direccionamiento.

Bit *W* y campo *REG*

La instrucción *MOV* AX, 568H

Indica mover inmediatamente al registro interno *AX* el valor 568H. El registro interno puede ser de **1** byte o de una palabra. Esta instrucción requiere **2** o **3** bytes, como se indica en la *Figura C*.

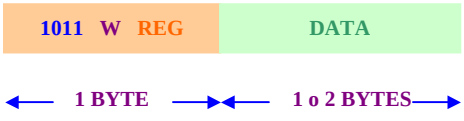


FIGURA C. Uso del bit *N* y del campo *REG*.

El primer byte contiene los *bits más significativos* (*MSB*) como **1011**. El próximo bit es *W*.

W indica: **1 para *word***
0 para *byte*

Esto es, si el registro destino es de **16** bits o de **8** bits.

Los siguientes **3** bits del primer byte, campo *REG*, determinan cuál registro está involucrado. La *Figura D*, muestra el código de selección del registro.

REG	REGISTRO DE 16 BITS	REGISTRO DE 8 BITS
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH

REG	REGISTRO DE 16 BITS	REGISTRO DE 8 BITS
111	DI	BH

***FIGURA D*. Registro involucrado en la operación**

Campo *DATA*. Si el registro de destino es de **1** byte, el dato debe estar en el segundo byte de la instrucción. Si el destino es de una palabra, el segundo byte de la instrucción son los **8** *bits menos significativos* (***lsb***) del ***dato***, el tercer byte de la instrucción son los **8** *bits más significativos* (***MSB***) del ***dato***. La siguiente tabla, muestra los ***nemotécnicos*** **2** o **3** bytes

NEMOTÉCNICO	CÓDIGO OBJETO
MOV AX, 568	Instrucción de 3 bytes B8 68 05
MOV *AL*, 56	instrucción de 2 bytes B0 56

Bit *D*, *MOD* y *R/M*

En este ejemplo, moveremos ***datos*** desde memoria o moveremos un ***registro*** hacia o desde otro ***registro***. Usaremos una instrucción como:

MOV AX, BX

Esta instrucción es de **2** bytes porque no nos referimos a memoria. Los bytes aparecerán como lo muestra la ***Figura E***:

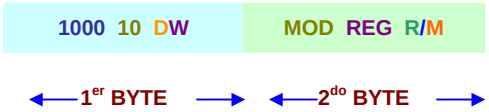


***FIGURA E*.**

El primer byte contiene los 2 *bits menos significativos* como ***DW***. El bit ***W*** es para **word=1** o para **byte=0**. La ***D*** es para indicar si el ***dato*** será almacenado en el operando especificado por los campos ***MOD*** y ***R/M*** (***D* = 0**) o si va a ser almacenado en el registro especificado por el campo ***REG*** (***D* = 1**)

La ***Figura F*** muestra las asignaciones para ***MOD*** y ***R/M***. Note en la descripción de ***MOD*=11**, el campo ***R/M*** es codificado con un formato de registro. Este formato se mostró en la ***Figura D***.

Registros base e índice especificados por *R/M* para operandos en memoria (*MOD* <> 11)

R/M	REGISTRO BASE	REGISTRO ÍNDICE
000	BX	SI
001	BX	DI
010	BP	SI
011	BP	DI
100	NINGUNO	SI
101	NINGUNO	DI
110	BP	NINGUNO
111	BX	NINGUNO

MOD	DESPLAZAMIENTO	COMENTARIO
00	CERO	
01	8 BITS contenido del próximo byte de la instrucción, signo extendido a 16 bits	La instrucción contiene un byte adicional
10	16 bits contenidos en los próximos 2 bytes de la instrucción	La instrucción contiene 2 bytes adicionales
11	Registro R/M	

Si MOD = 00 y R/M = 110, entonces
1. Lo expuesto arriba no se aplica
2. La instrucción contiene 2 bytes adicionales
3. La dirección offset es contenida en esos bytes

***FIGURA F*. Definiciones para el código objeto del *8086/8088* de los campos MOD y R/M**

Para esta instrucción deseamos almacenar el ***dato*** en el registro ***AX***. Por lo tanto el bit ***D* = 0**. Esto significa que el dato debe ser almacenado en la localidad especificada por los campos ***MOD*** y ***R/M***. Por lo tanto, ***MOD* = 11**. El campo ***R/M* = 000**, indicando que el registro ***AX*** es el destino para los datos. El campo ***REG*** para el segundo byte de datos es **011**. indicando que el registro ***BX*** es el registro fuente a ser utilizado. El segundo byte de la instrucción es **11 011 000 = D8**. Por lo que el código objeto para la instrucción es:

MOV AX, BX es 89
D8

Código Objeto para el uso de *Registro Base y Registro Índice*

Examinemos un último ejemplo para generar código objeto para el ***8086/8088***. En éste vamos a calcular el ***código objeto*** para la instrucción:

MOV CX, COUNT [BX] [SI]

Esta instrucción es de **4** bytes, como se muestra en la ***Figura G***:

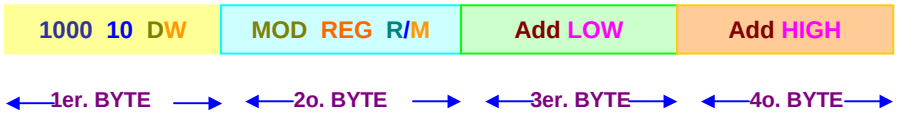


***FIGURA G*. Formato del *código objeto* para una instrucción como:**
MOV CX, COUNT [BX] [SI]

El primer byte de la ***Figura G***, debe tener el bit ***D*=1**. Esto es debido a que el destino para el dato debe ser especificado por el campo ***REG*** en el segundo byte. El bit ***W*=1**. porque es una transferencia de palabra. El primer byte es:

10001011 = 8B

En el segundo byte, ya que estamos usando una constante que requiere **16** bits, el campo ***MOD* = 10**. Refiriendo a la ***Figura F***, ésta indica que el desplazamiento debe ser formateado en **2** bytes y deben seguir a este segundo byte. El próximo campo para el segundo byte es el campo de registro (***REG***) Ya que debemos usar el registro ***CX***, este valor debe ser **001** (esto se obtiene de la ***Figura D***)

Finalmente, el campo ***R/M*.** Ya que el campo ***MOD*<> 11**, este campo debe especificar cuál registro base y cuál registro de índice están siendo usados para generar la dirección *offset* de **16** bits. En nuestro caso, usamos el campo **[*BX* + *SI* + DESPLAZAMIENTO]**

- Esto corresponde a ***R/M* = 000**, ver ***Figura F***
- El segundo byte es **1000 1000 = 88**
- El tercer y cuarto byte corresponden al ***desplazamiento***
- En este caso, el valor de ***COUNT* = 0345H**. Los últimos **2** bytes son **4503H**

Esto da el siguiente ***código objeto*** total para la instrucción:

```
MOV     CX, COUNT [BX] [SI]      8BH
                                 88H
                                 45H
                                 03H
```

Sumario del *Código Objeto*

Una pregunta que surge al programador ¿**Debo conformar los campos *D*, *W*, *REG*, *MOD* y *R/M*, en cada instrucción**? **NO**, la computadora lo hace (el lenguaje ensamblador lo genera) Esta sección se presentó para permitirle al programador un mejor entendimiento del trabajo interno del microprocesador ***8086/8088***

Interrupciones de los *Servicios Básicos de Entrada y Salida* (*BIOS*, *por sus siglas en inglés*)

FUNCIÓN INT 21

- **(AH)=1 ENTRADA DESDE EL TECLADO**

  Esta función espera a que se digite un carácter en el teclado. Muestra el carácter en la pantalla (*eco*) y retorna el código **ASCII** en el registro *AL*.

  (*AL*) = carácter leído desde el teclado

  Ejemplo:

```
MOV     AH, 1
INT     21h          ;AL = dato ASCII leído desde el teclado
```

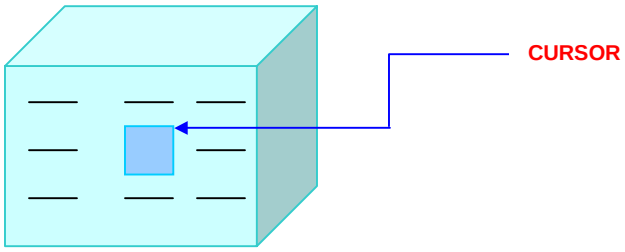
- **(AH)=2 SALIDA EN EL EXHIBIDOR (*display*)**

Despliega un carácter en la pantalla. Algunos caracteres tienen un significado especial:

- **7** ***CAMPANA***: ***Suena*** durante un segundo
- **8** ***BACKSPACE***: ***Mueve*** el cursor hacia la izquierda un carácter
- **9 TABULADOR**: ***Mueve*** el tabulador a su próxima posición (cada 8 caracteres)
- **0Ah LF**: ***Mueve*** el cursor a la siguiente línea
- **0Dh CR**: ***Mueve*** el cursor al inicio de la línea corriente
- **(DL)**: Carácter a ***desplegar*** en la pantalla

Ejemplo: **Desplegar un carácter**

```
MOV     DL, 40          ; carácter a desplegar
MOV     AH, 2
INT     21h             ; aparece en la posición corriente del cursor
                        ; el carácter contenido en DL
```



Ejemplo: **Hacer que suene la campana 2 segundos**

```
MOV     DL, 7           ; DL = campana
MOV     AH, 02
INT     21h             ; 1 segundo
INT     21h             ; 1 segundo
```

- **(AH)=8 ENTRADA DESDE EL TECLADO SIN ECO**

Lee un carácter desde el teclado, pero no se despliega en la pantalla

(***AL***) = carácter leído desde el teclado

```
MOV     AH, 08
INT     21h             ;AL = carácter
```

- **(AH)=9 DESPLIEGA UNA CADENA DE CARACTERES**

Despliega en la pantalla la cadena apuntada por el par de registros ***DS:DX***. Debemos marcar el fin de la cadena con el carácter "**$**"

DS:DX apuntan a la cadena que se va a desplegar

- **(AH)=0A h LEE UNA CADENA**

Lee una cadena de caracteres desde el teclado ¿**Dónde queda la información**?

- **(AH)=25h ACTIVA EL VECTOR DE INTERRUPCIÓN**

  Activa un vector de interrupción, para que apunte a una nueva rutina

  (*AL*) = número de interrupción
  ES:**BX** dirección del manipulador de interrupciones

- **(AH)=35h CONSIGUE VECTOR DE INTERRUPCIÓN**

  Consigue la dirección de la rutina de servicio para el número de interrupción dado en *AL*

  (*AL*) = número de interrupción
  ES:**BX** dirección del manipulador de interrupción

- **(AH)=4Ch SALIDA AL DOS**

  Retorna al **DOS**. Trabaja para ambos archivos ***.com** y ***.Exe**. Recuerde que ***INT* 20h** trabaja solamente para archivos ***.com**

  (*AL*) = código de retorno, normalmente activo a **0**, pero se puede activar a cualquier otro número y usar los comandos del **DOS**, ***IF*** y ***ERRORLEVEL***, para detectar errores

PROGRAMACIÓN EN LENGUAJE ENSAMBLADOR

Los archivos deben terminar con la extensión "**ASM**". Las letras minúsculas trabajan igual que las mayúsculas, pero durante el presente trabajo se utilizarán mayúsculas para evitar confusión entre el número **1** y la minúscula **l**, el **0** (cero) y la letra **O**. Considérense las siguientes líneas de un programa:

```
.MODEL SMALL
.CODE

        MOV   AH, 2H
        MOV   DL, 2AH
        INT   21H
        INT   20H

        END
```

Una **H** después de cada número indica al ensamblador que los números son hexadecimales. Recuerde que ***DEBUG*** asume que todos los números son hexadecimales pero el ensamblador asume que todos los números son decimales.

El ensamblador puede confundir números con etiquetas, para evitar esto coloque un **0** (**cero**) antes de un número hexadecimal que inicie con una letra.

Ejemplo:

```
        MOV   DL, ACH          ; AC es una etiqueta
        MOV   DL, 0ACH         ; AC es un número hexadecimal
```

Con el fin de hacer más legibles los programas, usaremos el tabulador para el espaciado.

A las ***directivas*** del ensamblador se les llama también ***pseudo-operaciones***. Se les conoce como ***directivas*** porque en lugar de generar instrucciones, proporcionan información y direcciones al ensamblador.

La ***pseudo-operación END*** marca el fin del archivo fuente.

Creación de Archivos Fuente

El ensamblador puede usar archivos fuente que contengan caracteres **ASCII** estándar. Considere que no todos los procesadores de texto escriben archivos en disco usando solamente los caracteres **ASCII** estándar. Antes de ensamblar un programa verifique que esté en código **ASCII**.

Puede ver caracteres extraños en el programa. Muchos procesadores de texto agregan información de formateo adicional en el archivo. El ensamblador los trata como errores. Utilice la versión no documento de su procesador de texto. También se requiere ***una línea en blanco*** después de la instrucción ***END***

Para ensamblar el programa:

```
A>MASM PROGRAMA;

MICROSOFT ® MACRO ASSEMBLER VERSION 5.10
COPYRIGHT © MICROSOFT CORP 1981, 1988. ALL RIGHTS RESERVED

49822 + 219323 BYTES SYMBOL SPACE FREE
        0 WARNING ERRORS
        0 SEVERE ERRORS

A>
```

El ensamblador crea un archivo intermedio ***.OBJ** el cual contiene nuestro programa e información adicional usada por otro programa llamado ***LINKER*** <encadenador>.

Encadenar al archivo *.OBJ

```
A>LINK ARCHIVO;

Microsoft ® Overlay Linker Version 3.64
copyright © microsoft corp 1983-1988. All rights reserved

LINK : warning L4021: No Stack Segment
```

Hemos creado nuestro archivo ***.EXE**. Ahora sí necesitamos crear nuestra versión ***.COM**. El archivo ***EXE2BIN.EXE*** del **DOS** convierte un archivo **EXE** a un archivo **BIN**.

```
A>EXE2BIN ARCHIVO ARCHIVO.COM
A>
```

Si listamos los archivos que hemos creado, obtendríamos:

```
A>DIR ARCHIVO.*
Volume in drive A has no label
directory of A:\
archivo.ASM    100
archivo.OBJ    200
archivo.EXE    600
archivo.COM     50
```

Recuerde los detalles del ***DEBUG***.

```
A>DEBUG ARCHIVO.COM
-    U

397F:0100      B402          MOV    AH, 02
397F:0102      B22A          MOV    DL, 2A
397F:0104      CD21          INT    21
397F:0106      CD20          INT    20
```

Note que las dos primeras y la última línea no aparecen en el listado. Se eliminan en la versión final del lenguaje de máquina porque son directivas y éstas son para documentación. El ensamblador toma en cuenta esta documentación a costa de más líneas de código.

Comentarios. Para comentar una línea ponga el "**;**". Todo lo que está después del "**;**" el ensamblador lo considera como ***comentario***.

Etiquetas. Pueden tener hasta **31** caracteres y pueden contener ***letras***, ***números*** y cualesquiera de los siguientes ***símbolos***:

¿	interrogación
.	punto
@	arroba
_	subrayado
$	dólar

Las ***etiquetas*** no deben iniciar con un ***número decimal*** y el ***punto*** se utiliza solamente como el primer carácter.

Una de las principales diferencias entre el ***DEBUG*** y el ensamblador reside en las etiquetas. Recuerde que con ***DEBUG*** debemos hacer el cálculo nosotros. El ensamblador refiere a etiquetas y él calcula el desplazamiento.

Cuando ponemos **:** después de una ***etiqueta***, decimos que la ***etiqueta*** es cercana (***NEAR***). El término ***NEAR*** tiene que ver con los segmentos.

Procedimientos en ensamblador

El ensamblador asigna direcciones a las instrucciones. Cada vez que hacemos un cambio al programa, debemos ensamblar nuevamente dicho programa. Considérese el siguiente ***programa***:

```
.MODEL SMALL
.CODE

PRINT _A_J    PROC
        MOV   DL, "A"          ; inicia con el carácter A
        MOV   CX, 10           ; imprime 10 caracteres

PRINT _LOOP:
        CALL  WRITE _CHAR      ; imprime carácter
        INC   DL               ; siguiente carácter del alfabeto
        LOOP  PRINT _LOOP      ; continua
        MOV   AH, ACh          ; retorna al DOS
        INT   21h

PRINT _A_J    ENDP

WRITE _CHAR PROC
        MOV   AH, 02           ; activa el código de la función para sacar CHAR
        INT   21h              ; imprime el carácter que está en DL
        RET                    : retorna de este procedimiento
WRITE _CHAR ENDP
        END   PRINT _A_J
```

PROC y *ENDP* son directivas para definir procedimientos. *PROC* define el inicio y *ENDP* define el final.

En este ejemplo, tenemos **2** procedimientos; por lo tanto, necesitamos indicarle al ensamblador cuál debe usar como el procedimiento principal (donde debe el microprocesador iniciar la ejecución de nuestro programa) La directiva *END* indica al ensamblador cual es el procedimiento principal. El procedimiento principal puede estar en cualquier lugar del programa. Sin embargo como estamos tratando con archivos ***.COM**, debemos colocar primero el procedimiento principal.

NOTA: Si encuentras algún mensaje de error que no reconozcas, verifica que hayas digitado el programa adecuadamente. Si aún falla, consulta el manual del ensamblador

Después, usa el *DEBUG* para desensamblar el programa y ver cómo el ensamblador pone los procedimientos juntos.

```
C> DEBUG   PRINT:A_J.COM
```

Procedimiento para exhibir números decimales en el monitor

Digita el siguiente programa y nómbralo *VIDEO_IO.ASM*.

```
.MODEL SMALL
.CODE

TEST_WRITE_HEX    PROC

        MOV   DL, 3Fh          ; prueba con 3Fh
        CALL  WRITE_HEX
        INT   20h                                  MOV   AH, 4Ch
TEST_WRITE_HEX    ENDP         ; retorna al DOS    INT   21h
```

```
PUBLIC          WRITE_HEX

;este procedimiento convierte el byte en el registro DL a hex
;y escribe los dos dígitos hexadecimales en la posición corriente
;del cursor
;               DL byte a ser convertido a hexadecimal
;usa a:         WRITE_HEX_DIGIT

WRITE_HEX   PROC
        PUSH    CX                  ; almacena registros usados en este procedimiento
        PUSH    DX
        MOV     DH, DL              ; hacemos una copia del byte
        MOV     CX, 4
        SHR     DL, CX
        CALL    WRITE_HEX_DIGIT     ;despliega el primer dígito hexadecimal
        MOV     DL, DH              ; vamos con el nibble bajo
        AND     DL, 0Fh             ; elimina el nibble alto
        CALL    WRITE_HEX_DIGIT     ; despliega al segundo dígito hexadecimal
        POP     DX
        POP     CX
        RET
WRITE_HEX   ENDP

PUBLIC          WRITE_HEX

;este procedimiento convierte los 4 bit menos significativos de DL a un dígito hexadecimal
;y lo escribe en la pantalla
;;                  DL los 4 bits menos significativos contienen el número a ser impreso
;usa a:             WRITE_CHAR

WRITE_HEX_DIGIT     PROC
        PUSH    DX
        CPM     DL, 10              ; ¿es el nibble <10?
        JAE     HEX,_LETTER         ; no convierte a letra
        ADD     DL, "0"             ; suma 30
        JMP     SHORT WRITE_DIGIT   ;escribe carácter

HEX_LETTER:
        ADD     DL, "A"-10          ;suma 37, convierte a letra hexadecimal

WRITE_DIGIT:
        CALL    WRITE_CHAR          ;despliega letra en la pantalla
        POP     DX
        RET
WRITE_HEX_DIGIT     ENDP

PUBLIC          WRITE_CHAR

;este procedimiento imprime un carácter en la pantalla usando
;función del DOS
;               DL byte a imprimir en la pantalla

WRITE_CHAR  PROC
        PUSH    AX
        MOV     AH, 2               ; función para sacar carácter
        INT     21h                 ; saca carácter
        POP     AX
        RET
WRITE_CHAR  ENDP
```

```
END     TEST_WRITE_HEX
```

En este programa, hay una nueva directiva ***PUBLIC*** la cual indica al ensamblador que genere información adicional al ***LINKER***. El ***LINKER*** nos permite traer partes separadas de nuestro programa, ensamblarlas desde diferentes archivos fuente en un solo programa. La directiva ***PUBLIC*** informa al ensamblador que el procedimiento nombrado después de ***PUBLIC*** debe ser hecho público o disponible a los procedimientos de otros archivos.

Esqueleto de un programa en ensamblador

Para referencia futura, lo mínimo que requiere un programa en ensamblador es:

```
.MODEL SMALL
.CODE

NOMBRE_PROCEDIMIENTO  PROC

        INSTRUCCIONES
        ┌─────────────┐
        │             │
        │             │
        │             │
        │             │
        │             │
        │             │
        └─────────────┘

                      { MOV   AH, 4Ch
INT     20h           { INT   21h

        NOMBRE_PROCEDIMIENTO  ENDP
END
```

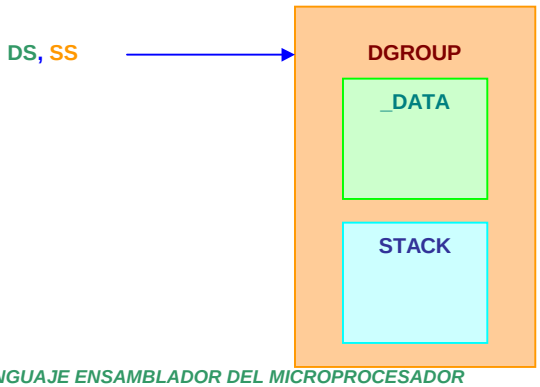
Directiva: .DATA

Declara un segmento de datos que se usa para variables de memoria. El segmento creado por ***.DATA*** es **_DATA**.

Directiva: .MODEL SMALL

Se considera un programa como ***SMALL*** cuando contiene hasta **64K** de código y **64K** de datos.

Ya que ***.DATA*** y ***.STACK*** son datos, se ponen en un solo segmento cuando se usa el modelo de memoria ***MODEL SMALL***.



El agrupamiento de los segmentos *STACK* y *DATA* en un solo segmento es controlado por un mecanismo del ensamblador llamado *GROUPS*.

El ensamblador crea un grupo llamado *DGROUP* que a su vez crea un solo segmento con *.DATA* y *STACK*.

Directiva: .DOSSEG

Causa que el ***STACK SEGMENT*** se cargue en memoria después del ***DATA SEGMENT***; hay una razón para esto. El segmento de datos que creamos tiene datos que necesitan estar en el archivo ***.EXE**, así que puedan ser copiados en memoria cuando nuestro programa está "corriendo". De igual forma, el ***STACK*** necesita tomar espacio en memoria, pero la memoria del ***STACK*** no necesita estar inicializada (solamente el ***SS:SP*** tiene que activarse)

De esta forma, poniendo el ***STACK SEGMENT*** después del ***DATA SEGMENT*** no necesitamos espacio en el disco para el ***STACK***.

La directiva ***DOSSEG*** indica al ensamblador que queremos los segmentos de nuestro programa cargado en un orden específico (el segmento de código primero y el ***STACK*** al último)

Veamos lo expuesto en un ejemplo:

Obtener la suma de 10 datos y desplegar el resultado en la pantalla.

```
DOSSEG
.MODEL SMALL
.STACK                                  ; asigna un STACK de 1K
. DATA

        PUBLIC DATOSSUM
DATOSSUM    DB      01h, 02h, 03h, 04h, 05h     ; datos que vamos
            DB      06h, 07h, 08h, 09h, OAh     ; a procesar

.CODE
.SUMA       PROC

            MOV     AX, DGROUP
            MOV     DS, AX

            XOR     BX, BX                  ; índice de acceso al área de datos
            XOR     DL, DL                  ; acumulador de datos
            MOV     CX, OAh                 ; número de datos a procesar
ACUMULA:
            ADD     DL, DATOSSUM [BX]
            INC     BX
            LOOP    ACUMULA
                                            ; DL contiene el resultado
            MOV     AH, 02h                 ; despliega el resultado
            INT     21h                     ; AL = dato a desplegar
            MOV     AH, 4Ch                 ;
            INT     21h                     ; salida al DOS

      SUMA      ENDP

                END
```

Los siguientes ***ejercicios*** propuestos, son con el fin de mostrar el uso del conjunto de instrucciones del microprocesador y sus modos de direccionamiento.

Empezamos a usar ***instrucciones básicas*** y ***modos de direccionamiento básicos*** y vamos aumentando la complejidad de las instrucciones y de los modos de direccionamiento en forma tal que al concluir los ejercicios hayamos cubierto un **80**% del conjunto de instrucciones de la máquina.

Para cada ejercicio está propuesta la solución en un ***diagrama de flujo***, después se muestra la ***codificación en lenguaje ensamblador*** y para los usuarios de la utilería ***DEBUG*** del ***DOS***, se muestra la ***codificación en lenguaje de máquina***.

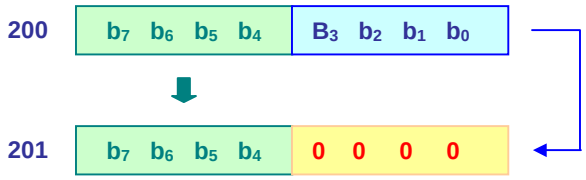
Todos los programas con ***DEBUG*** inician en la localidad **100** y los comandos a utilizar son: ***R*** (***registro***), ***T*** (***ejecución*** paso a paso) y ***E*** (***examina*** y cambia memoria) Con estos comandos, podemos ejecutar cualquiera de los programas.

Con el fin de aprovechar el lenguaje ensamblador, también se muestra la ***codificación*** y la ***ejecución*** en el *Turbo Ensamblador de Borlan*.

EJERCICIO 1. El contenido de la localidad **0200** tiene un dato de **8** bits. Realizar un programa que coloque en la localidad de memoria **0201** el *nibble* (***4 bits***) *más significativo*, tomando en cuenta que el *nibble menos significativo* debe ser **0**.

SOLUCIÓN

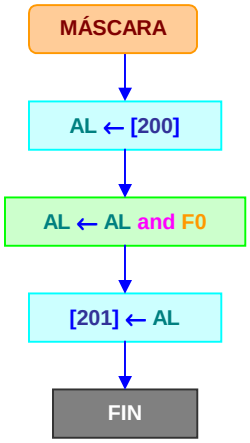
Se puede ver gráficamente el contenido de las localidades **0200** y **0201** para comprender mejor el enunciado:



Podemos hacerlo con la operación ***AND***, donde:

$b_n \bullet 1 = b_n$
$b_n \bullet 0 = 0$

Diagrama de flujo:



Codificación en lenguaje ensamblador:

```
MASCARA:
    MOV   AL, [200]      ;mueve el contenido de la localidad 200 a AL
    AND   AL, F0         ;coloca 1111 0000 en AL
    MOV   [201], AL      ;mueve el contenido de AL a la localidad 201
    MOV   AH, 4C         ;mueve el contenido de 4C a AH
    INT   21
```

Codificación en lenguaje máquina:

```
0100   A0   00   20
0103   24   F0
9105   A2   01   02
0108   4B   4C
010A   CD   21
```

Ejecución:

- R
- T
- T
- T
- T
- E

EJERCICIO 2.

EJERCICIO 3. ***Realizar un programa que coloque en la localidad de memoria 202 el número menor de los contenidos en las localidades 200 y 201***. Considere números sin signo.

NOTA: El ***DEBUG*** asume que los datos son ***hexadecimales***, mientras que el ***ensamblador*** considera que son ***decimales***. En caso de usar el ensamblador agregar **H** al final del dato.

SOLUCIÓN

Gráficamente, se tiene:

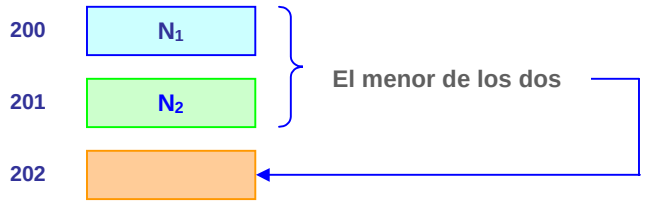
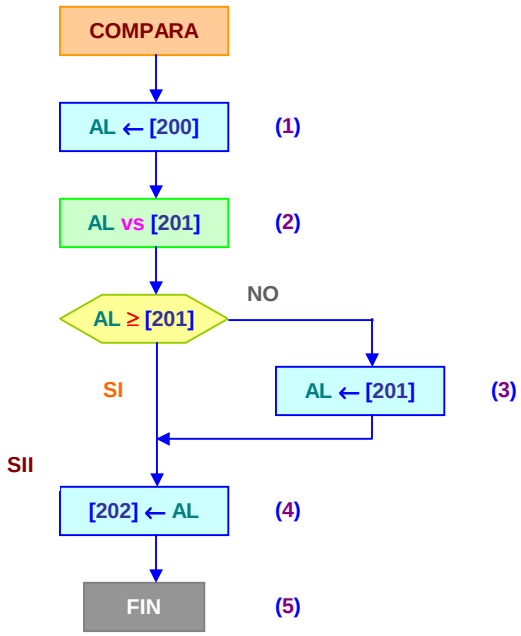


Diagrama de flujo:



Codificación en lenguaje ensamblador:

```
COMPARA: MOV   AL, [200]
         CMP   AL, [201]
         JNB   SII
         MOV   AL, [201]
SII:     MOV   [202], AL
         MOV   AH, 4C
         INT   21
```

Si empleamos el ***DEBUG***, podemos utilizar el comando ***ASSEMBLY*** e iniciar en la localidad de memoria **100**; o bien, podemos utilizar la tabla de instrucciones y codificar directamente en lenguaje de máquina empleando el comando ***E***, que permite capturar en hexadecimal el programa.

Codificación en lenguaje máquina:

```
0100  A0  00  02
0103  3A  06  01  02
0107  73  03
0109  A0  01  02
010C  A2  02  02
010F  B4  4C
0111  CD  21
```

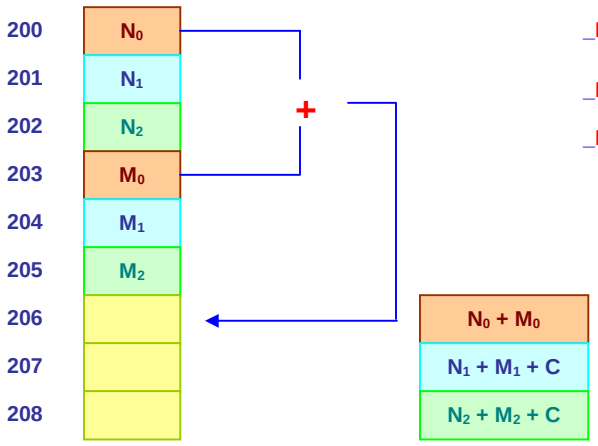
Para ejecutar el programa, cargamos el registro ***IP*** con el valor de inicio del programa, escribiendo:

_**R** **IP** **100** ↵ después con el comando _**T** (*Trace*) ejecutamos paso a paso el programa.

EJERCICIO 4. ***Sumar 2 números binarios de 24 bits y almacenar el resultado a partir de la localidad de memoria 0206h***. Los *bits menos significativos* de los números empiezan en las localidades **0200h** y **0203h**.

SOLUCIÓN

Se puede ver gráficamente el contenido de las localidades **0200h** a **0208h** para comprender mejor el enunciado:




```
_E  200  AA  AA  AA
_E  203  BB  BB  BB
_E  206  00  00  00
         65  66  66
```

Para introducir los datos, usamos la instrucción E (examina y/o cambia el contenido de la localidad de memoria

Codificación en lenguaje ensamblador:

```
SUMA:
    MOV  AL, [200]   ;mover el contenido de la localidad 200 a AL      } 8 LSB
    ADD  AL, [203]   ;sumar el contenido de la localidad 201 a AL      }
    MOV  [206], AL   ;mover el contenido de AL a la localidad 206      }
    MOV  AL, [201]   ;mover el contenido de la localidad 201 a AL      }
    ADC  AL, [204]   ;sumar el contenido de la localidad 204 con AL    } 2º BYTE
    MOV  [207], AL   ;mover el contenido de AL a la localidad 207      }
    MOV  AL, [202]   ;mover el contenido de la localidad 202 a AL      }
    ADC  AL, [205]   ;sumar el contenido de la localidad 205 con AL    } 8 MSB
    MOV  [208], AL   ;mover el contenido de AL a la localidad 208
    RET
```

NOTA: Observe que en las instrucciones de **MOVer** no se afecta ninguna bandera del ***PSW***. [200] se refiere al contenido de la dirección **200**.

AA AA AA =	1010	1010	1010	1010	1010	1010
BB BB BB =	1011	1011	1011	1011	1011	1011
Acarreos	111	111	111	111	111	1
	0110	0110	0110	0110	0110	0101

EJERCICIO 5. ***Uso del registro CX como contador***. Vamos a mover los **8** *bits menos significativos* del registro ***BL*** al registro ***BH***.

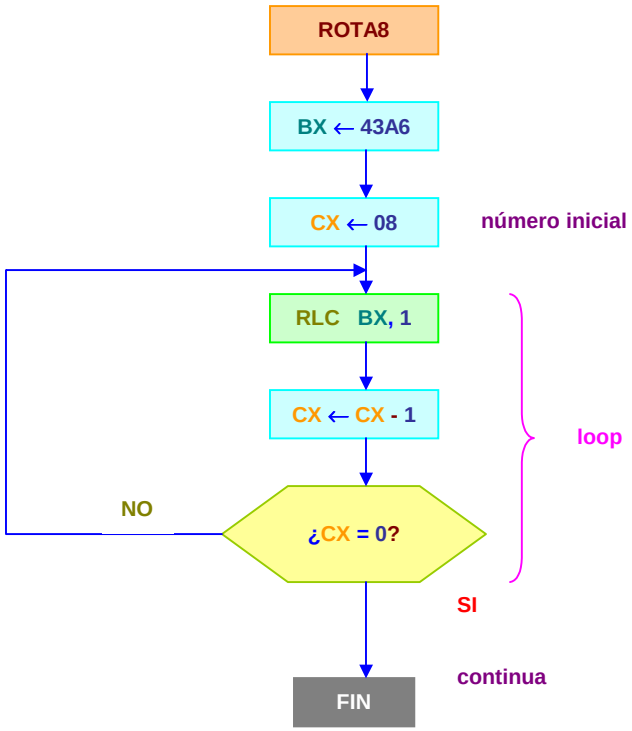
Sean los datos iniciales:

BX = BH y BL = 43A6

La instrucción que usaremos será **RLC BX, 1**, **8** veces. Para este tipo de repeticiones usaremos al registro ***CX*** como contador.

SOLUCIÓN

Diagrama de flujo:



Codificación en lenguaje ensamblador:

```
ROTA8:    MOV   BX, 43A6      ;(1)
          MOV   CX, 08        ;(2)
OTRAVEZ:  RLC   BX, 1         ;(3)
          LOOP  OTRAVEZ       ;(4)
          MOV   AH, 4C        ;(5)
          INT   21            ;(6)
```

NOTA: Observar que en **(4)** ponemos **LOOP** **OTRAVEZ** y como es un salto hacia atrás el ***DEBUG*** pone un ……..

Codificación en lenguaje máquina:

```
0100   B3   A6   43      ;(1)
0103   B9   08   00      ;(2)
0106   D1   D3            ;(3)
0108   E2   FC            ;(4)
010A   B4   4C            ;(5)
010C   CD   21            ;(6)
```

NOTA: En **0108** saltamos hacia atrás **4** localidades de memoria, es decir:

0000 0100

Como el salto es hacia atrás, lo ponemos como negativo en notación ***complemento a 2***, es decir:

C_1 = 1111 1011
+1
C_2 = 1111 1100
F C

Cuando saltamos hacia delante el número es positivo. Para salto corto usamos **8** bits y para salto largo **16** bits.

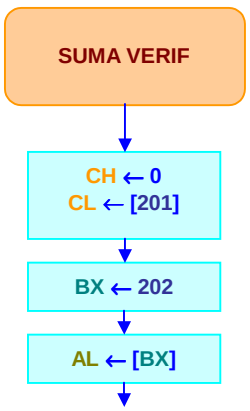
EJERCICIO 7. ***Calcular la suma verificación de una serie de datos***. La ***longitud*** de la serie de datos está en la localidad **201h** y la ***serie*** comienza en la localidad **202h**. Almacenar la ***suma verificación*** en la localidad **200h**. La ***suma verificación*** se forma haciendo la suma ***O exclusiva*** entre todos los números de la serie.

SUGERENCIA: Utilice ***CX*** como contador.

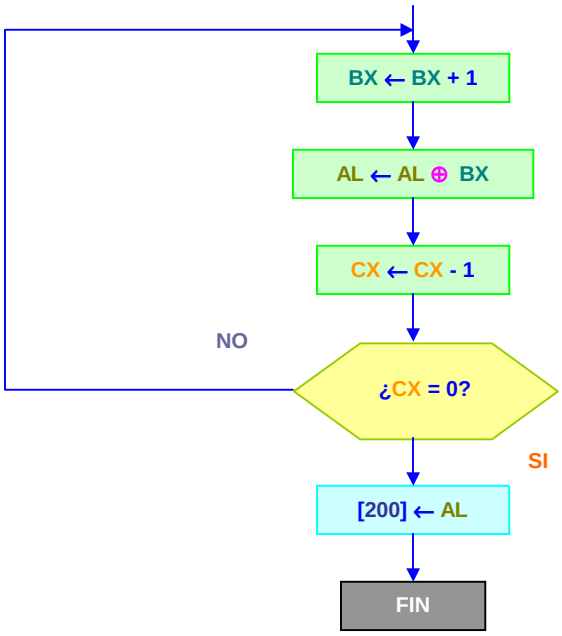
SOLUCIÓN

200	Suma verificación
201	longitud
202	inicio

Diagrama de flujo:



Dirección de inicio



Codificación en lenguaje ensamblador:

```
SUMAVERIF:
          MOV   CH, 0          ;poner 0 en CH
          MOV   CL, [201]      ;mover el contenido de la localidad 201 a CL
          MOV   BX, 200        ;mover el contenido de 200 a BX
          MOV   AL, [BX]       ;mover el contenido de BX a AL
SUMAO:    INC   BX             ;incrementar BX
          XOR   AL, [BX]       ;O exclusiva contenidos AL y BX
          DEC   CX             ;decrementar CX
          LOOP  SUMAO
          MOV   [200], AL      ;mover el contenido de AL a la localidad 200
          RET
```

Codificación en lenguaje máquina:

```
0100   B5   00
0102   8A   0E   01   02
0106   3B   02   02
0108   4B
0109   8A   07
010C   43
010D   32   07
010F   E2   F8
0111   A2   00   02
0114   B4   4C
0116   CD   21        } Retorno al DOS
```

Si utilizamos la tabla de instrucciones y sus respectivos códigos de ***DEBUG***, podemos capturar en lenguaje de máquina el programa. Estando en **DOS** teclear:

```
C>DEBUG↵
_E 100↵
_R IP,100↵
_T↵
```

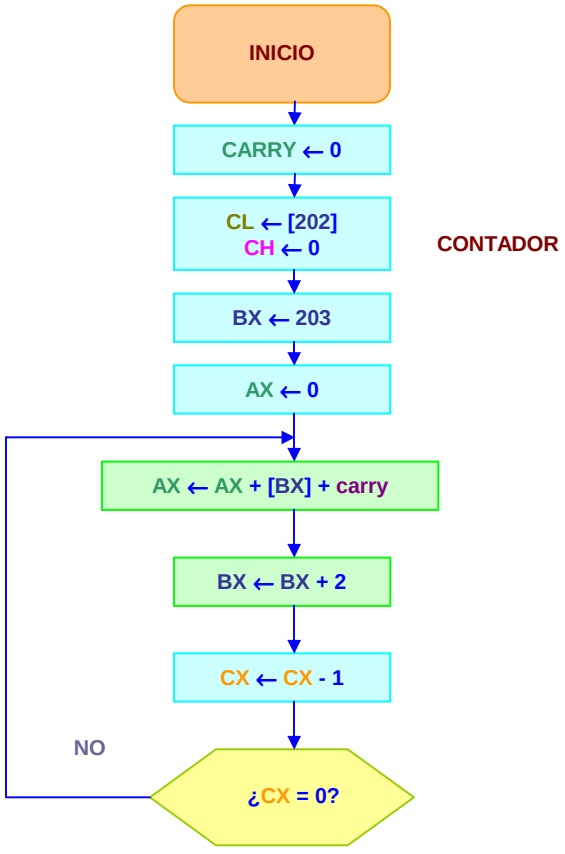
Introducimos el programa. Para ejecutarlo, colocamos **100** en el registro **IP**: después, para ejecutar el programa paso a paso usamos el comando **T**: etc.

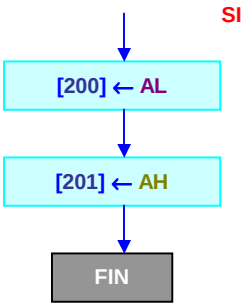
EJERCICIO 8. ***Calcular la suma de una serie de números de 16 bits***. La ***longitud*** de la serie está en la localidad de memoria **0202** y la serie ***empieza*** en la localidad de memoria **0203**. Almacenar la ***suma*** en las localidades de memoria **0200** y **0201**. Considere que la adición puede estar contenida en **16** bits

SOLUCIÓN

200	8 bits menos significativos	
201	8 bits más significativos	
202	Longitud = 3	
203	01 (bits menos significativos)	N_0
204	02 (bits más significativos)	
205	01 (bits menos significativos)	N_1
206	02 (bits más significativos)	
207	01 (bits menos significativos)	N_2
208	02 (bits más significativos)	

Diagrama de flujo:





Codificación en lenguaje ensamblador:

```
INICIO:  CLC
         MOV   CH, 0
         MOV   CL, [202]
         MOV   BX, 203
         MOV   AX, 0
RETORNO: ADC   AX, [BX]
         INC   BX
         LOOP  RETORNO
         MOV   [200], AX
         END
```

Codificación en lenguaje máquina: Usamos el ***DEBUG*** para ejecutar el programa en lenguaje de máquina

0100	F8				CLC	
0101	B5	00			MOV	CH, 0
0103	8A	0E	02	02	MOV	CL, [202]
0107	B3	03	02		MOV	BX, 203
010A	B8	00	00		MOV	AX, 0
010D	13	07			ADC	AX, [BX]
010F	83	C3	02		ADD	BX, 2
0112	E2	F9			LOOP	RETORNO
0114	A3	00	20		MOV	[200], AX

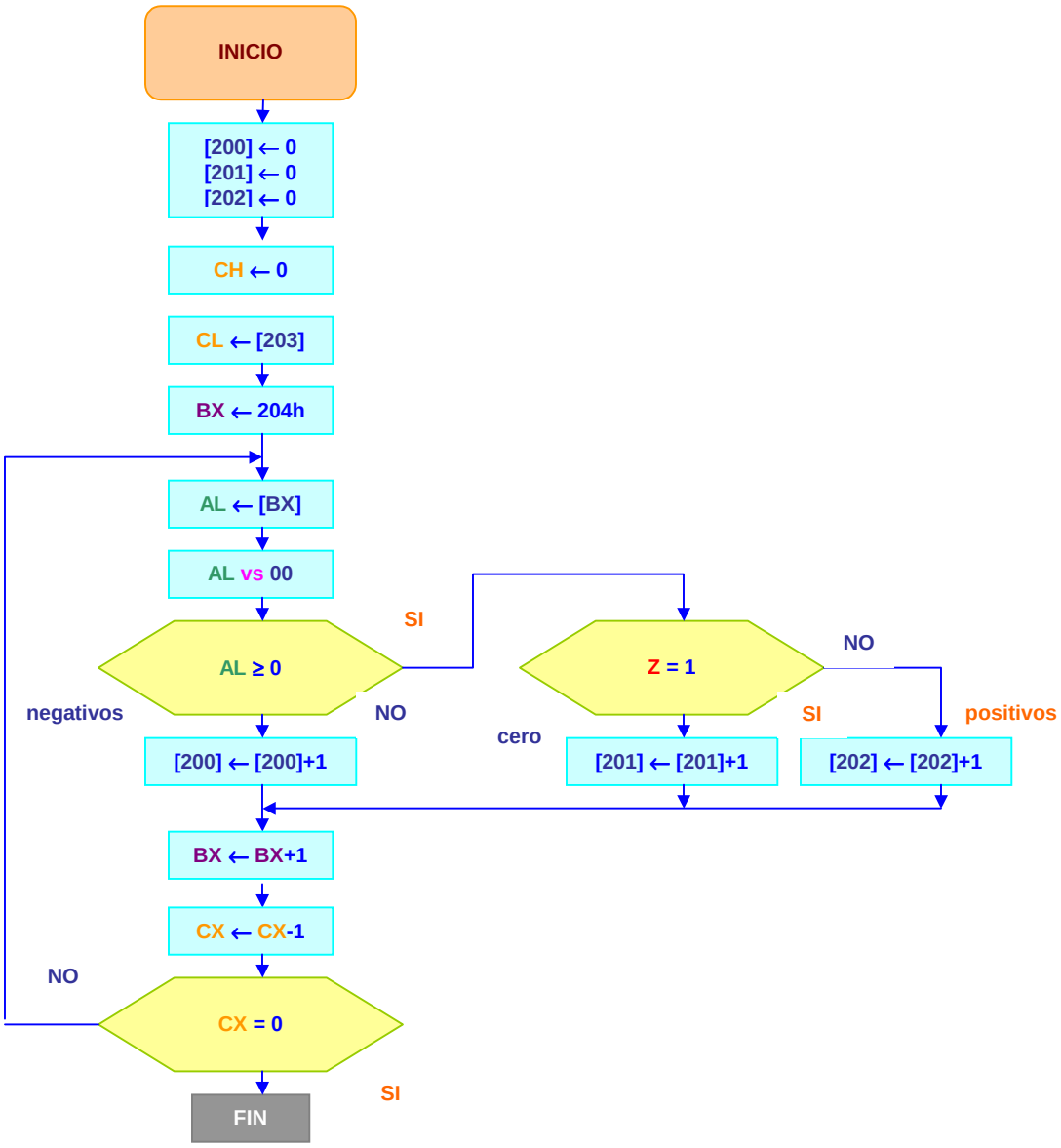
EJERCICIO 9. ***Determinar el número de elementos que sean positivos, negativos y cero***. La ***longitud*** del bloque está en la localidad **0203**, el bloque ***inicia*** en la localidad **0204**. En la localidad **0200** deposite el número de ***elementos negativos***, en la **0201** el número de ***elementos ceros*** y en la **0202** el número de ***elementos positivos***.

SOLUCIÓN

En forma gráfica, se tiene:

200	Número de elementos NEGATIVOS
201	Número de elementos CERO
202	Número de elementos POSITIVOS
203	LONGITUD
204	INICIO

Diagrama de flujo:



Codificación en lenguaje ensamblador:

```
INICIO:   MOV     AL, 00h
          MOV     [200], AL
          MOV     [201], AL
          MOV     [202], AL
          MOV     CH, AL
          MOV     CL, [203]
          MOV     BX, 0204
          MOV     AL, [BX]
RETORNO:  CMP     AL, 00h
          JGE     SII          ;SI es palabra reservada
          INC     [200]
SIGUE:    INC     BX           ;SI índice fuente
          LOOPNZ  RETORNO
FIN:      RET
SII:      JNZ     NO
          INC     [201]
          JMP     SIGUE
```

```
NO:     INC     [202]
        JMP     SIGUE
```

EJERCICIO 10. ***Encontrar al elemento más pequeño de un bloque de datos***. La ***longitud*** del bloque está en la localidad **4001** y el bloque ***inicia*** en la localidad **4002**. Almacene el ***resultado*** en la localidad **4000**, considerando números sin signo.

SOLUCIÓN

Gráficamente, se tiene:

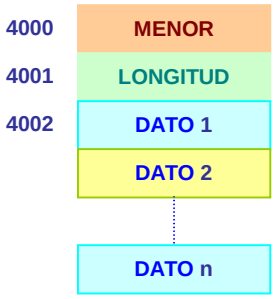
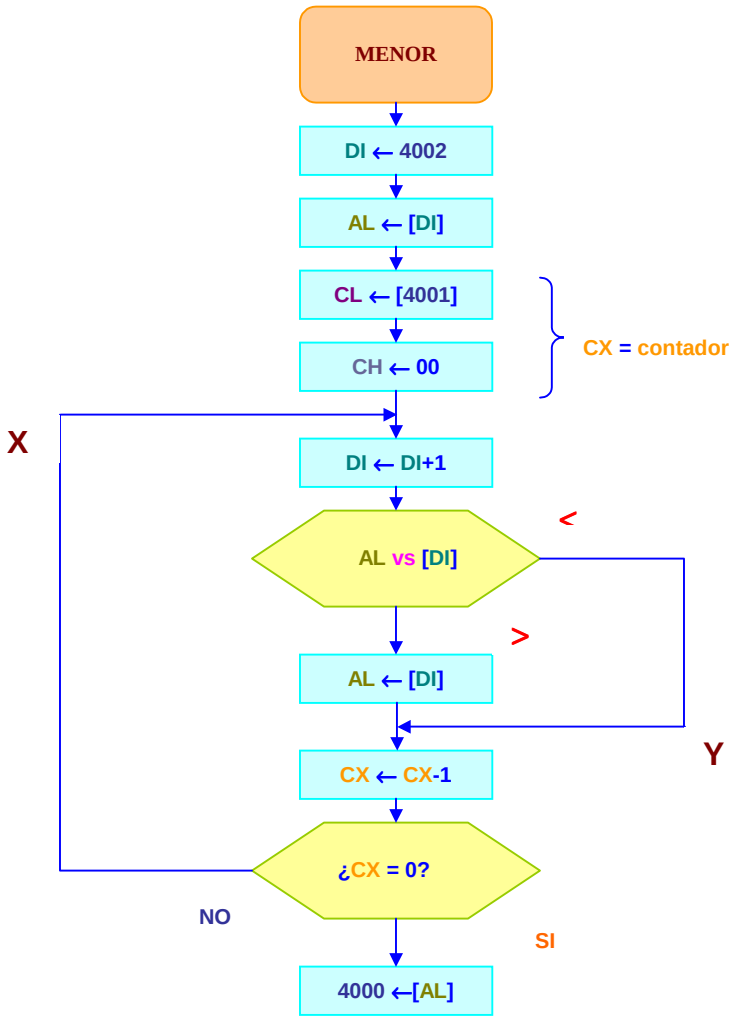
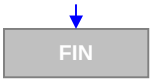


Diagrama de flujo





Codificación en lenguaje ensamblador:

```
MENOR:  MOV   DI, 4002
        MOV   AL, [DI]
        MOV   CL, [4001]
        MOV   CH, 0
X:      INC   DI
        CMP   AL, [DI]
        JB    Y                ;JL para números con signo
        MOV   AL, [DI]
Y:      LOOP  X
        MOV   [4000], AL
        END
```

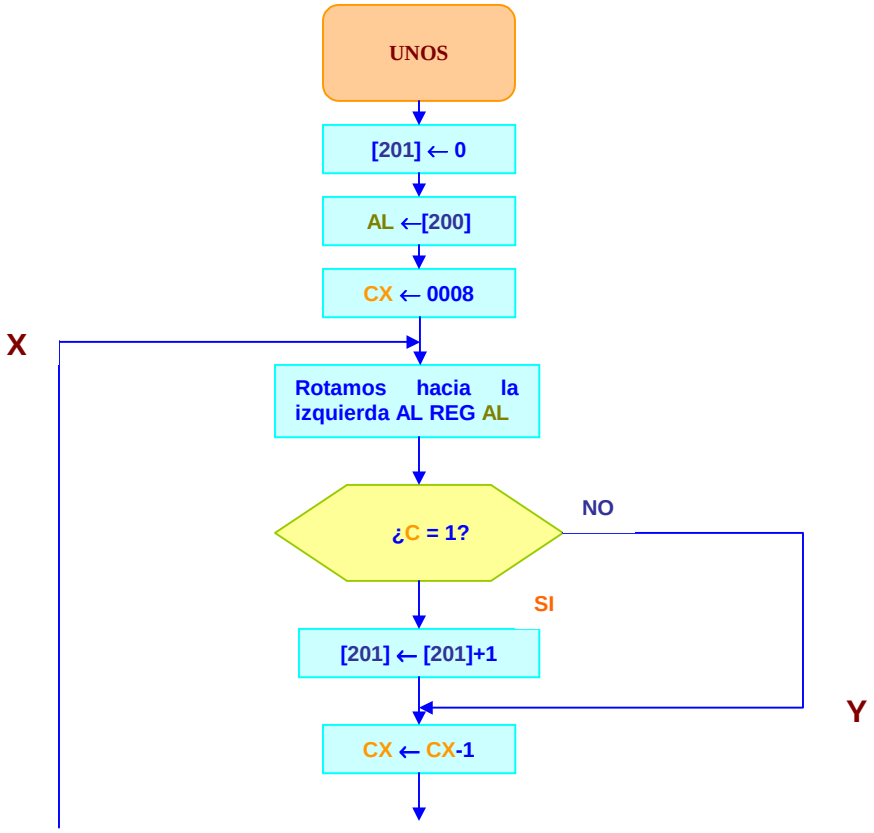
EJERCICIO 11. ***Determinar cuántos bits valen 1 en el contenido de la localidad de memoria 200***. Almacene el ***resultado*** en la localidad **201**

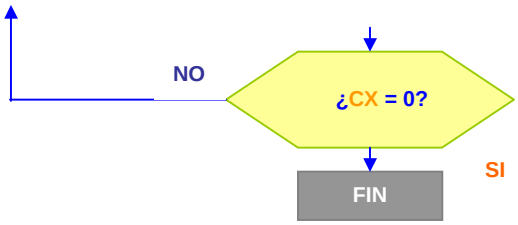
SOLUCIÓN

Gráficamente, se tiene:

200	DATO
201	# DE UNOS DE DATO

Diagrama de flujo





Codificación en lenguaje ensamblador:

```
MENOR:  MOV   AL, 00h
        MOV   [201], AL
        MOV   AL, [200]
        MOV   CX, 0008h
X:      ROL   AL
        JNC   Y
        INC    [201]
Y:      LOOP  X
        END
```

EJERCICIO 12. ***Determinar la longitud de un mensaje ASCII***. Los ***caracteres*** son de **7** bits, el octavo es **0**. La ***cadena*** de caracteres en la cual viene incluido el mensaje ***inicia*** en la localidad **201**. El mensaje inicia con ***STX*** (**02h**) y finaliza con *ETX* (**03h**) Colocar la longitud del mensaje en la localidad **200** (no contar ***STX*** ni *ETX*)

SOLUCIÓN

Gráficamente, se tiene:

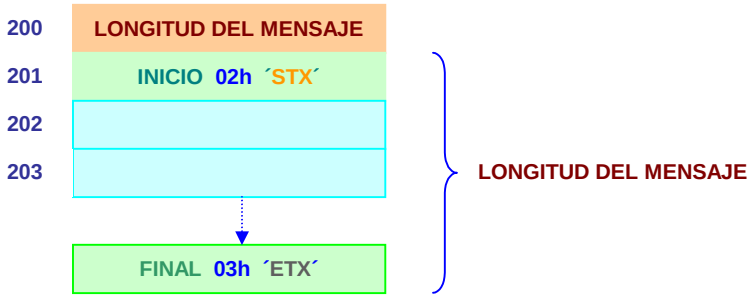
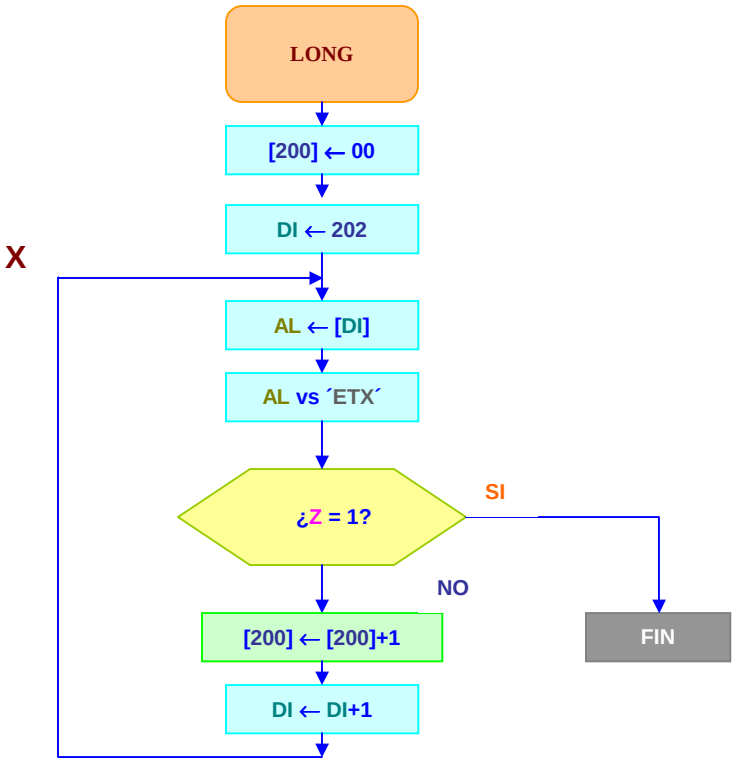


Diagrama de flujo



Codificación en lenguaje ensamblador:

```
LONG:   MOV     [200], 00h
        MOV     DI, 0202h
X:      MOV     AL, [DI]
        CMP     AL, ´ETX´
        JE      FIN
        INC     [200]
        INC     DI
        JMP     X
FIN:    END
```

EJERCICIO 13. ***Investigar una cadena de caracteres ASCII para determinar el último carácter distinto del blanco***. La cadena ***empieza*** en la localidad **202** y ***finaliza*** con ***CR*** (**0Dh**) Colocar la ***dirección*** del último carácter diferente del espacio en la localidad **200** y **201**.

SOLUCIÓN

Gráficamente, se tiene:

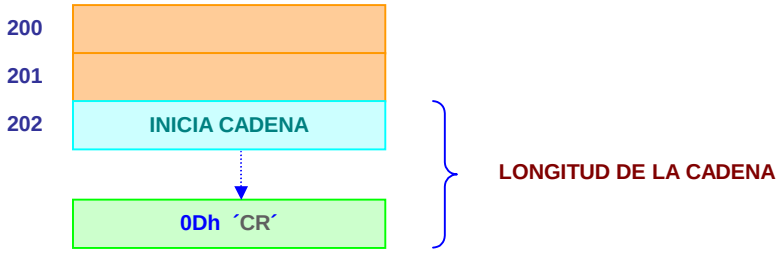
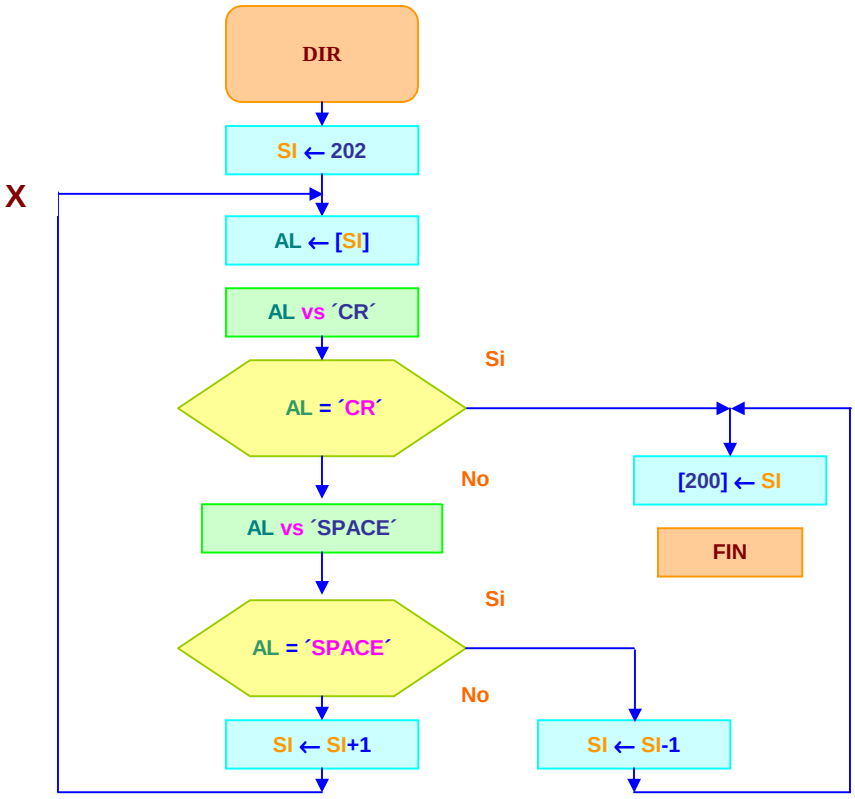


Diagrama de flujo



Codificación en lenguaje ensamblador:

```
DIR:        MOV     SI, 202
X:          MOV     AL, [SI]
            CMP     AL, 0dh          ;AL vs ´CR´
            JE      Almacena
            CMP     AL, 20h          ;AL vs ´SPACE´
            JE      ANTESALM
            INC     SI
            JMP     X
ANTESALM:   DEC     SI
ALMACENA:   MOV     200, SI
            END
```

EJERCICIO 14. ***Reemplazar todos los dígitos que están a la derecha del punto decimal por caracteres blancos***. La cadena ***inicia*** en la localidad **201** y consiste de ***números decimales*** codificados en ***ASCII*** y un posible punto decimal (**2Eh**) La ***longitud*** está en la localidad **200**. Si no aparece punto decimal, asuma que está implícito en el lado derecho.

NOTA: En ***ASCII*** los decimales **0**,…,**9** se representan como **30**,…,**39**.

SOLUCIÓN

Gráficamente, se tiene:

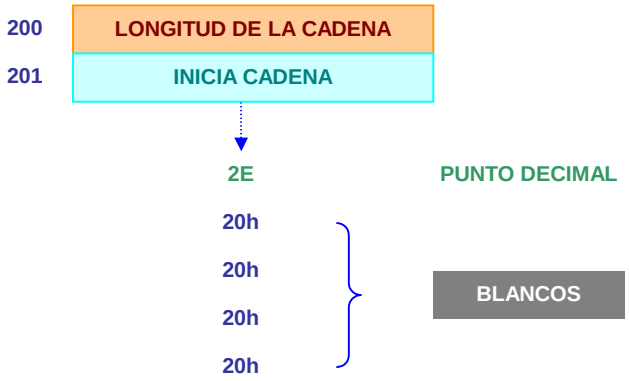
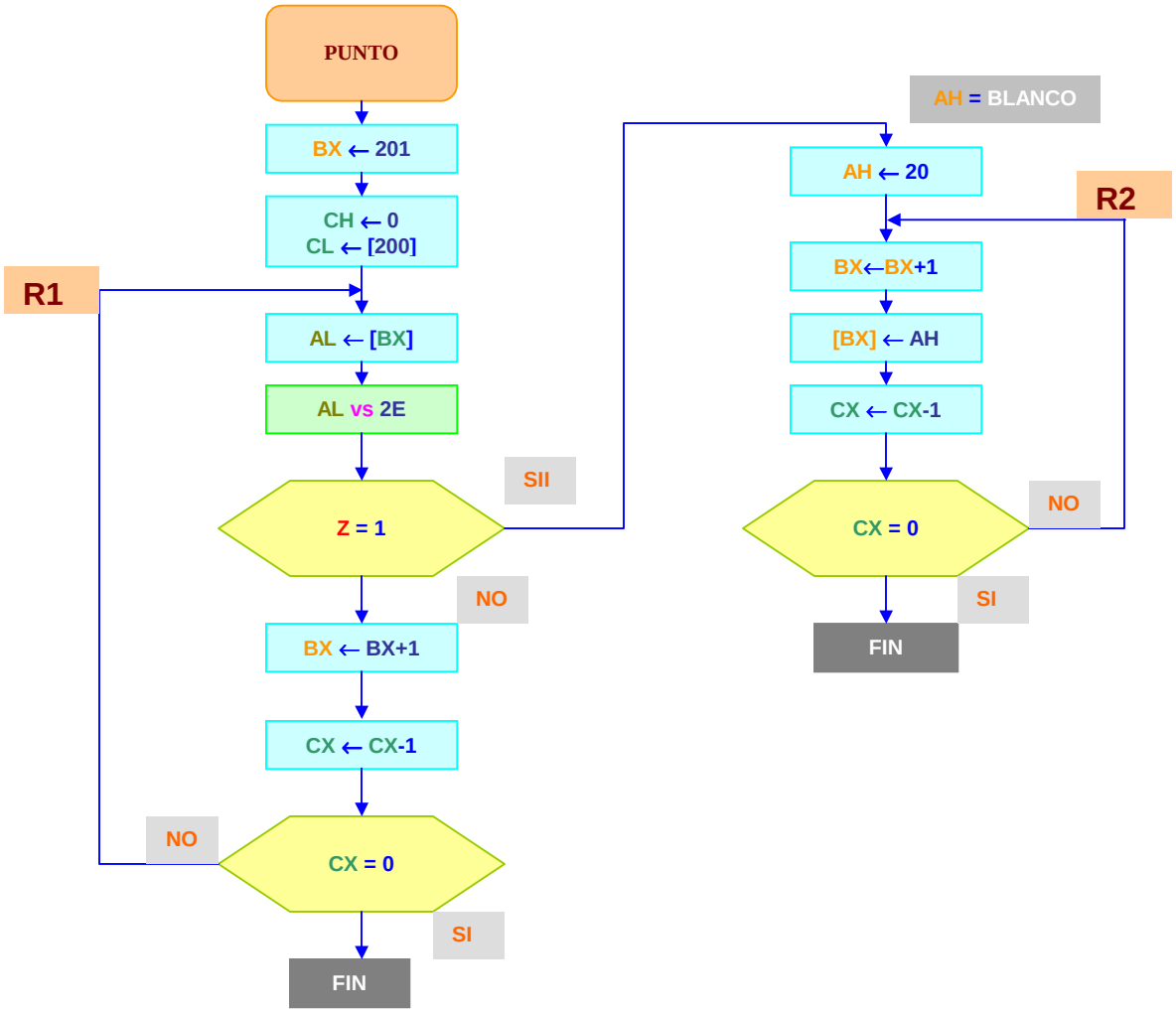


Diagrama de flujo



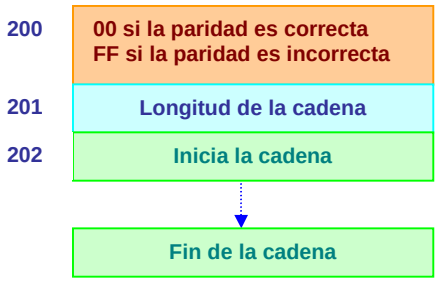
Codificación en lenguaje ensamblador:

```
PUNTO:  MOV   BX, 201
        MOV   CL, [0200]
        MOV   CH, 00h
RI:     MOV   AL, [BX]
        CMP   AL, 2Eh
        JZ    SII
        INC   BX
        LOOP  R1
        END
SII:    MOV   AH, 20h
R2:     INC   BX
        MOV   [BX], AH
        DEC   CX
        JNZ   R2
        END
```

EJERCICIO 15. ***Verificar la paridad par de caracteres ASCII***. La ***longitud*** de la cadena está en la localidad **201h** e ***inicia*** en la **202h**. Si la paridad de todos los caracteres de la cadena es correcta, ***borrar*** el contenido de la localidad **200h**; en caso contrario ***colocar FF***

SOLUCIÓN

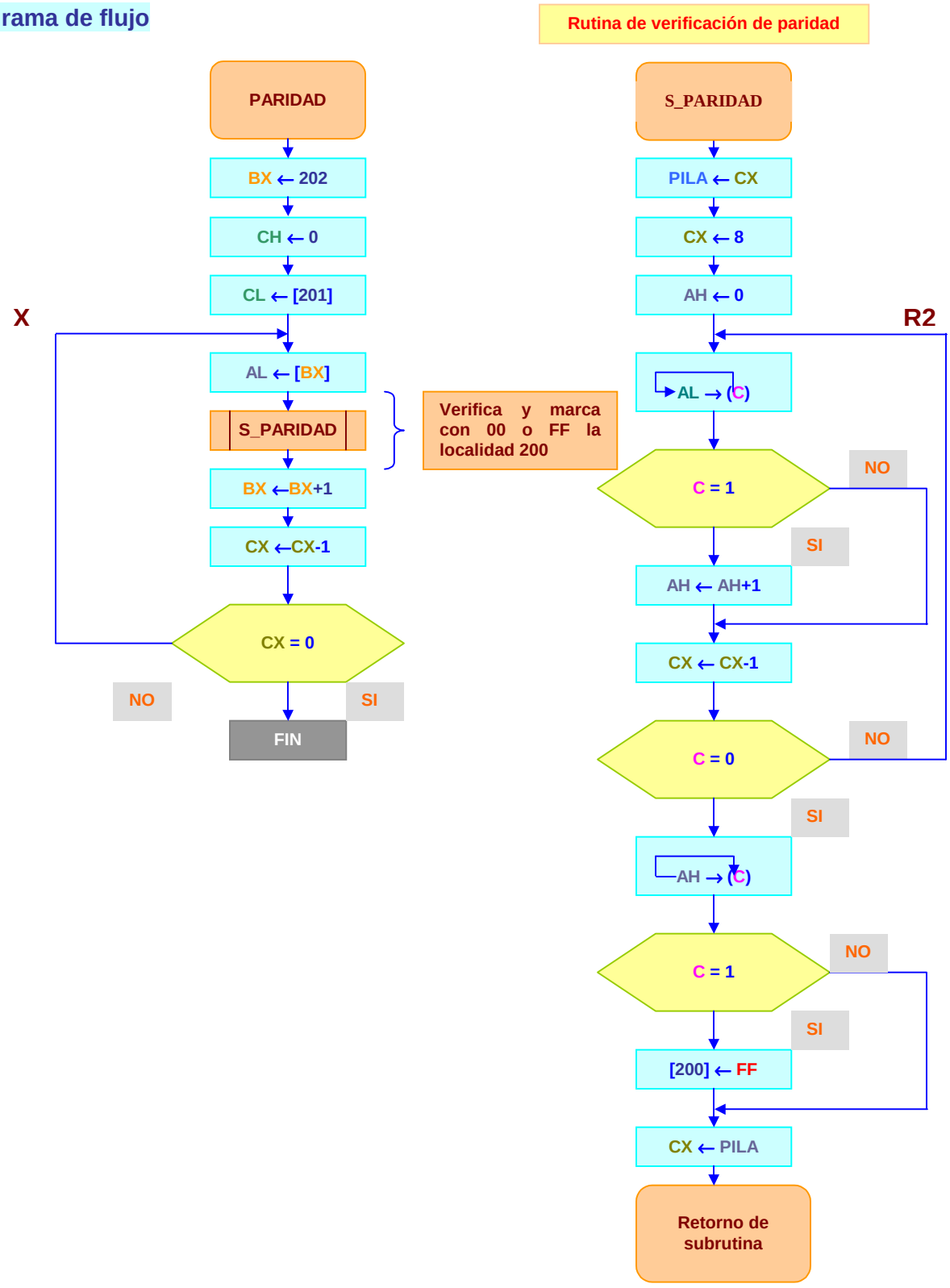
Gráficamente, se tiene:



AL = DATO

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	Contamos los UNOS
Si contador = par								Correcto
Si contador = non								Incorrecto

Diagrama de flujo



Codificación en lenguaje ensamblador:

Paridad:

```
PARIDAD:  MOV     BX, 202
          MOV     CH, 00
          MOV     CL, [201]
          MOV     AL, [BX]
X:        CALL    S_PARIDAD
          INC     BX
          LOOPNZ  X
          END
```

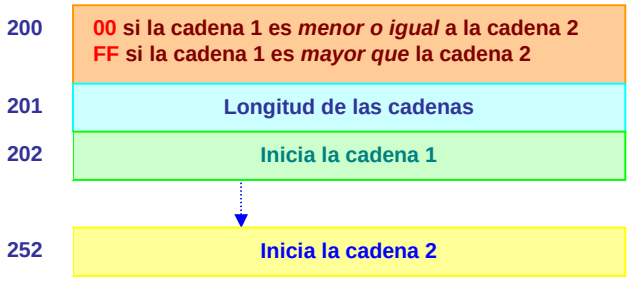
Rutina de verificación de paridad:

```
S_PARIDAD: PUSH   CX
           MOV    CX, 0008h
           MOV    AX,00h
R2:        RCR    AL
           JNC    SII
           ADD    AH, 01h
SII:       LOOPNZ R2
           RCR    AH
           JNC    NO
           MOV    [0200], FFh
NO:        POP    CX
           RET
```

EJERCICIO 16. ***Comparar dos cadenas de caracteres ASCII para determinar cuál sigue a la otra en orden alfabético***. La ***longitud*** de las cadenas está en la localidad **201**. Una cadena ***inicia*** en la localidad **202** y la otra en la localidad **252**. Si la cadena que inicia en la localidad **202** ***es menor o igual*** a la otra cadena, ***borrar*** la localidad **200**, en caso contrario ***almacenar*** **FF** en la localidad **200**.

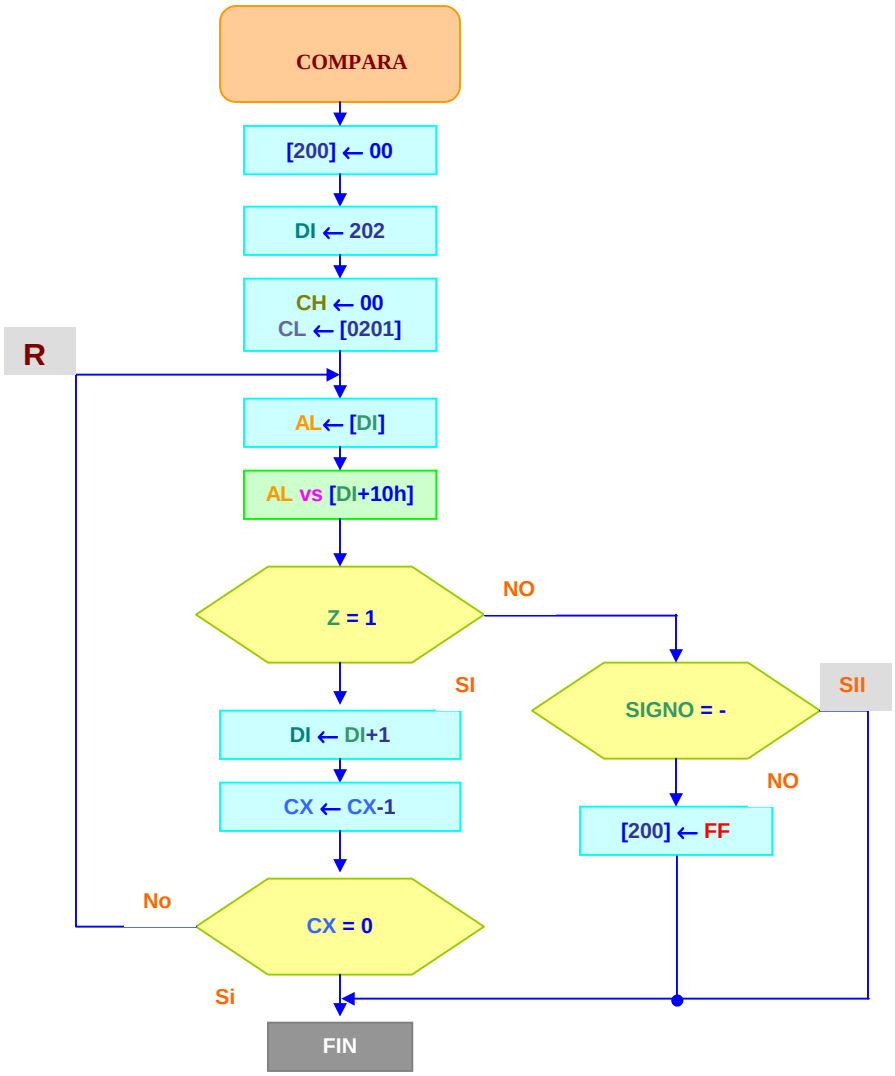
SOLUCIÓN

Gráficamente, se tiene:



NOTA: En lugar de utilizar los dos registros de índice, vamos a utilizar un apuntador más un desplazamiento.

Diagrama de flujo



Codificación en lenguaje ensamblador:

```
COMPARA: SUB   [200], [200]     ;asumimos que cadena 1 es menor o igual a cadena 2
         MOV   DI, 0202h        ;DI apunta al inicio de la cadena 1
         MOV   CH, 00h          ;CX igual a la longitud de las cadenas
         MOV   CL, [0201]
R:       MOV   AL, [DI]
         CMP   AL, [DI+10h]     ;apunta a la otra cadena
         JNC   NO
         INC   DI
         LOOPNZ R
         END
NO:      JB    SII
         MOV   [0200], FFh
SII:     END
```

EJERCICIO 17.

Tablas.

La instrucción ***XLAT*** realiza la traducción a través de una ***tabla*** de búsqueda. La ***tabla*** se debe cargar en memoria y la ***dirección de inicio*** (base) se guarda en el registro ***BX*** antes de utilizar esta instrucción.

Si los datos de esta ***tabla*** son **A(0)**, **A(1)**, … , **A(255)**, un valor **i** en **AL** se reemplaza por el valor **A(i)** al ejecutarse ***XLAT***.

Funcionamiento de la instrucción *XLAT*.

Los contenidos de los registros **BX** y **AL** se suman para obtener la dirección de una posición de memoria y el contenido de dicha posición se copia en **AL**.

La instrucción ***XLAT*** traduce datos de **8** bits; por lo tanto, están restringidos a un rango de **0** a **255**.

BX →

A(0)
A(1)
A(2)
A(3)
A(i)

BX + AL = ***Dirección*** **de una posición de memoria**
El contenido de dicha posición se copia en **AL**

EJERCICIO 18. *Conversión del código GRAY a código GRAY-EXCESO 3, sin utilizar la instrucción XLAT*.

SOLUCIÓN

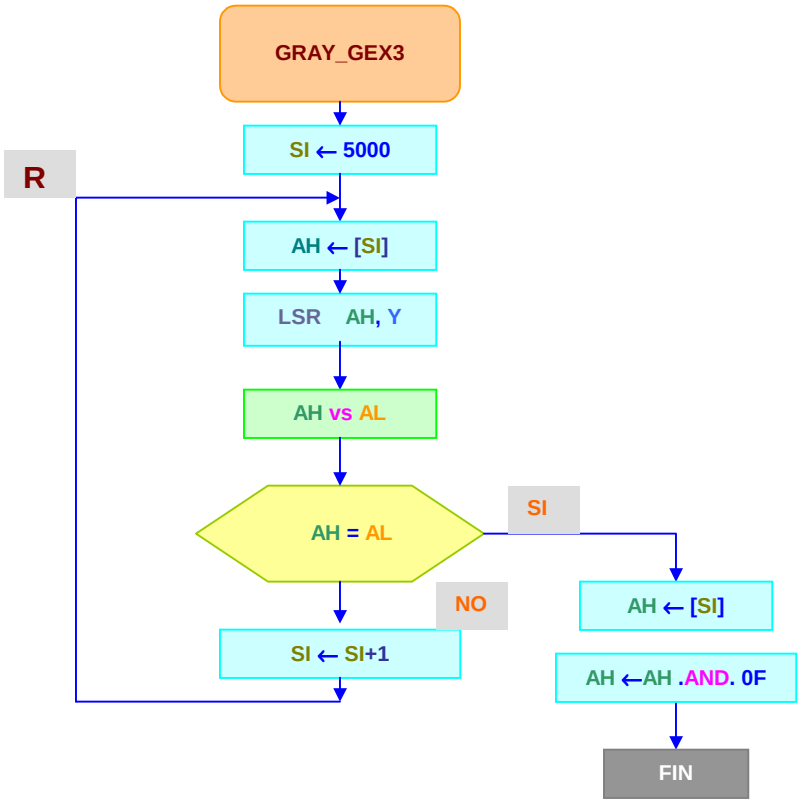
La ***tabla funcional*** para la conversión de códigos, se muestra a continuación:

	DEC	Código GRAY				Código GRAY EXCESO 3			
		G_3	G_2	G_1	G_0	G_{E3}	G_{E2}	G_{E1}	G_{E0}
5000	0	0	0	0	0	0	0	1	0
	1	0	0	0	1	0	1	1	0
	2	0	0	1	1	0	1	1	1
	3	0	0	1	0	0	1	0	1
	4	0	1	1	0	0	1	0	0
	5	0	1	1	1	1	1	0	0
	6	0	1	0	1	1	1	0	1
	7	0	1	0	0	1	1	1	1
	8	1	1	0	0	1	1	1	0
	9	1	1	0	1	1	0	1	0
	10	1	1	1	1	1	0	1	1
	11	1	1	1	0	1	0	0	1
	12	1	0	1	0	1	0	0	0
	13	1	0	1	1	0	0	0	0
	14	1	0	0	1	0	0	0	1
	15	1	0	0	0	0	0	1	1

Si consideramos que esta tabla de valores se encuentra almacenada a partir de la localidad **5000h**, podemos comparar el contenido del acumulador contra cada uno de los contenidos de las localidades, hasta encontrar la que acople.

Cuando se encuentra el acoplamiento, los **4** *bits menos significativos* de esa localidad son el resultado de la conversión.

Diagrama de flujo



Codificación en lenguaje ensamblador:

```
GRAY_GEX3: MOV     SI, 5000h      ;dirección de inicio de la tabla
R:         MOV     AH, [SI]       ;obtenemos GRAY en los 4 bits
           LSR     AH             ;menos significativos
           LSR     AH
           LSR     AH
           LSR     AH
           CMP     AH, AL
           JE      CONVIERTE
           INC     SI
           JMP     R
CONVIERTE: MOV     AH, [SI]       ;obtenemos la conversión en los
           AND     AH,0Hh         ;4 bits menos significativos de AH
           END
```

Errores de programación frecuentes.

1. **Invertir** el *orden* de sus operandos
2. **Utilizar** las *banderas* en forma incorrecta
3. **Considerar** la *lógica* en forma errónea
4. **Confundir** *direcciones* y *datos*
5. **Manejar** erróneamente *matrices* y *cadenas*. El problema suele encontrarse en que se exceden los límites
6. **Organizar** el *programa* en forma inadecuada. *Inicio de contadores*, *apuntadores*, *no almacenar resultados*, etc.

La *mejor manera* de acelerar los programas del *microprocesador* consiste en *reducir el número de saltos*.

Ejemplos de solución de problemas utilizando *software* y *hardware*.

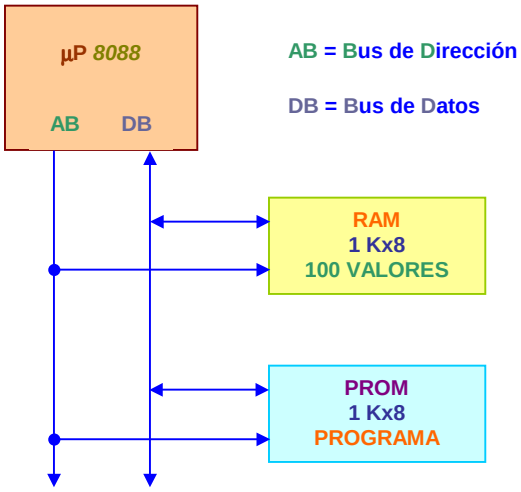
Ejemplo 1. Se tiene un convertidor ***analógico/digital*** (***A/D***) y se desea ***leer*** la información ***cada segundo*** hasta tener **100** valores, los cuales deben almacenarse en una ***tabla de datos***. Además,

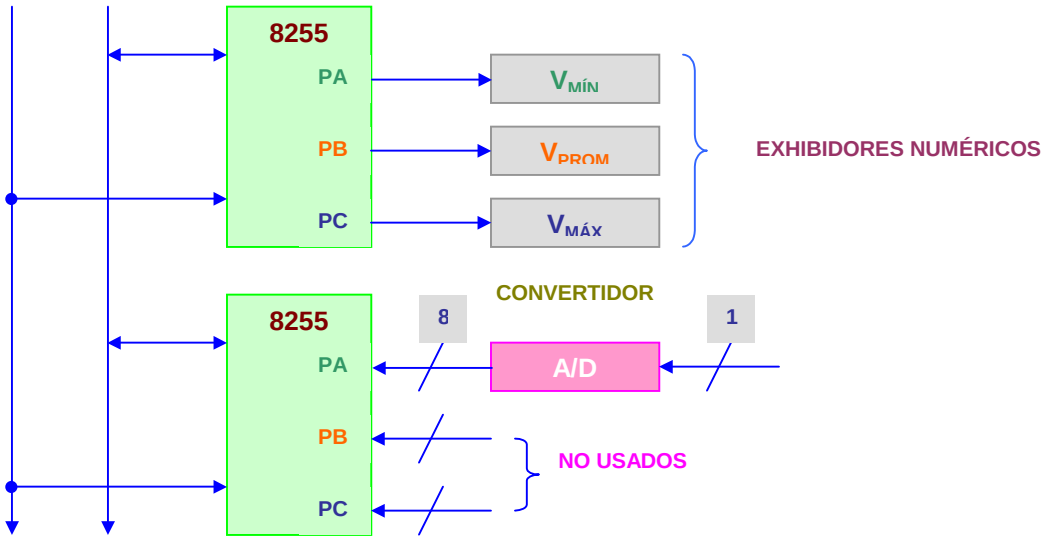
- **Calcular** el *valor promedio* de los valores leídos e indicarlo en un exhibidor numérico.
- **Obtener** el *valor mínimo* y desplegarlo en un exhibidor numérico.
- **Adquirir** el *valor máximo* y presentarlo en un exhibidor numérico.
- **Iniciar** nuevamente el ciclo después de una hora.

SOLUCIÓN

- Se necesita una ***m****emoria de* ***a****cceso* ***a****leatorio* (***RAM***) para los valores leídos del convertidor ***A/D***.
- Se requieren ***tres*** ***p****uertos* ***p****aralelos* de ***e****ntrada/****s****alida* (***E/S***) para los exhibidores numéricos correspondientes a $V_{MÁX}$, $V_{MÍN}$ y V_{PROM}.
- Se necesita un ***p****uerto* de ***E/S*** para el convertidor ***A/D***.
- Se requiere una ***m****emoria de* ***p****rogramable de* ***s****ólo* ***l****ectura* (***PROM***) para el programa.

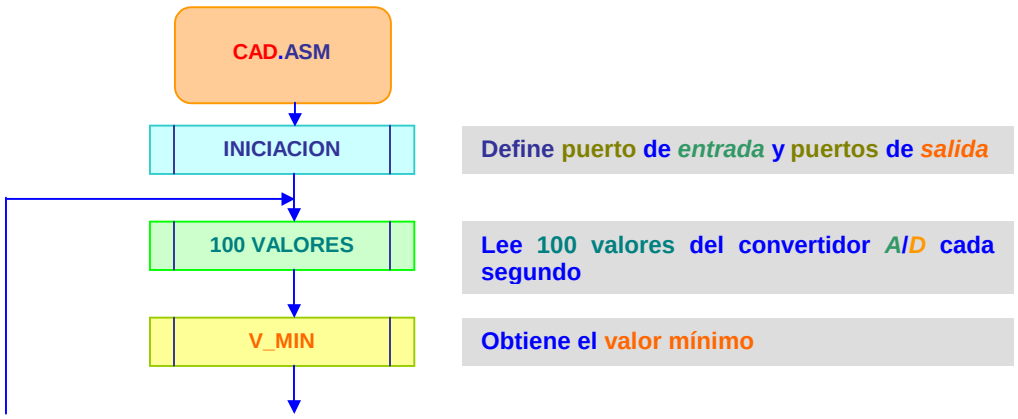
El diagrama a bloques de la siguiente figura, muestra la solución:

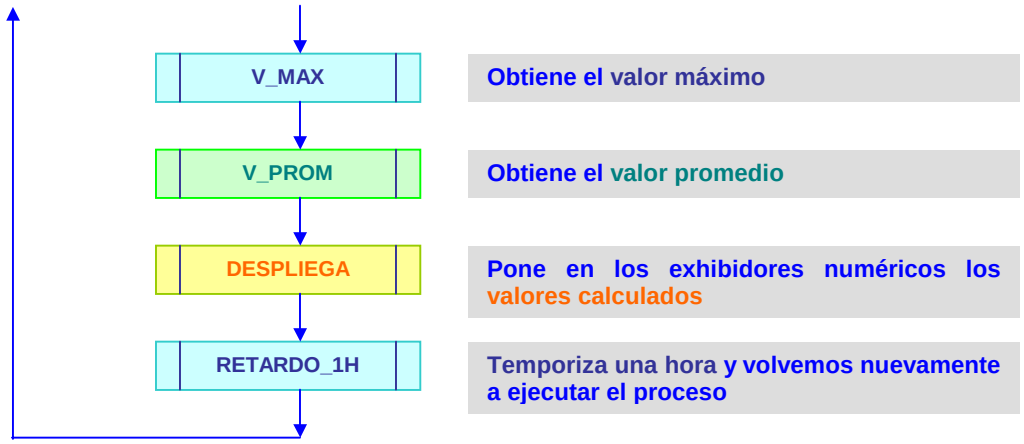




Configuración de los Circuitos Integrados		Localidades	
RAM ROM		Las primeras de 1K Las últimas de 1K	
8255	E/S	Dirección 40 41 42 43	Contenido V_{MIN} = PA V_{PROM} = PB $V_{MÁX}$ = PC C A/D = CONTROL
8255		Dirección 50 51 52 53	Contenido C A/D = PA No usado = PB No usado = PC CONTROL

Diagrama de flujo



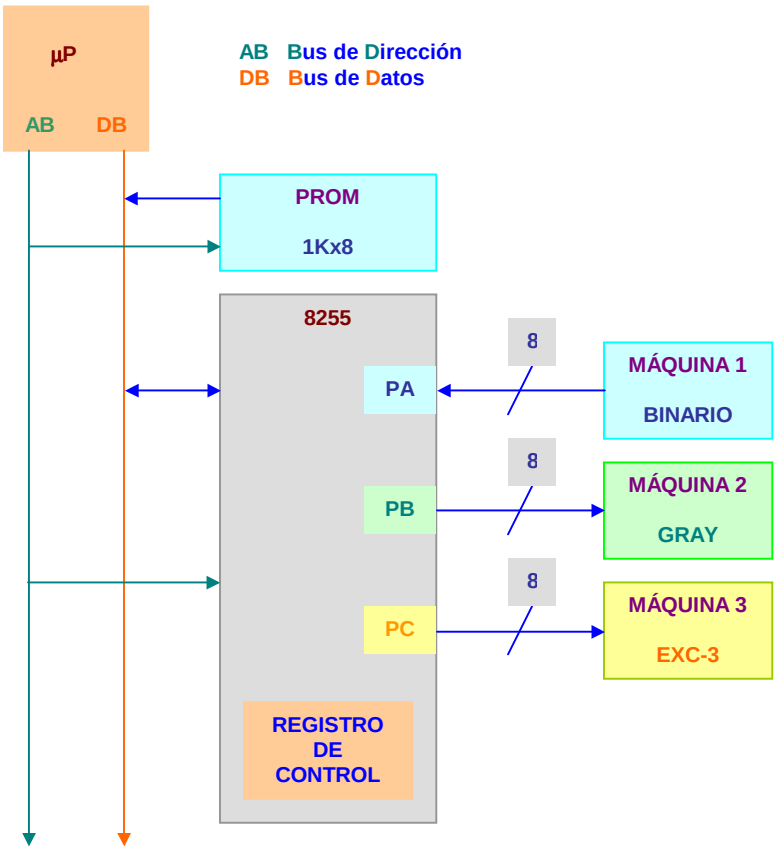


Ejemplo 2. Se tienen **2** máquinas herramientas que aceptan información en código ***Gray*** y en código ***Exceso-3***. La máquina que les proporciona la información trabaja en código ***binario*** y alimenta cada segundo un ***dato***, consistente en **2** *nibbles*.

Resuelva el problema de ***conversión de código*** con un ***microprocesador***.

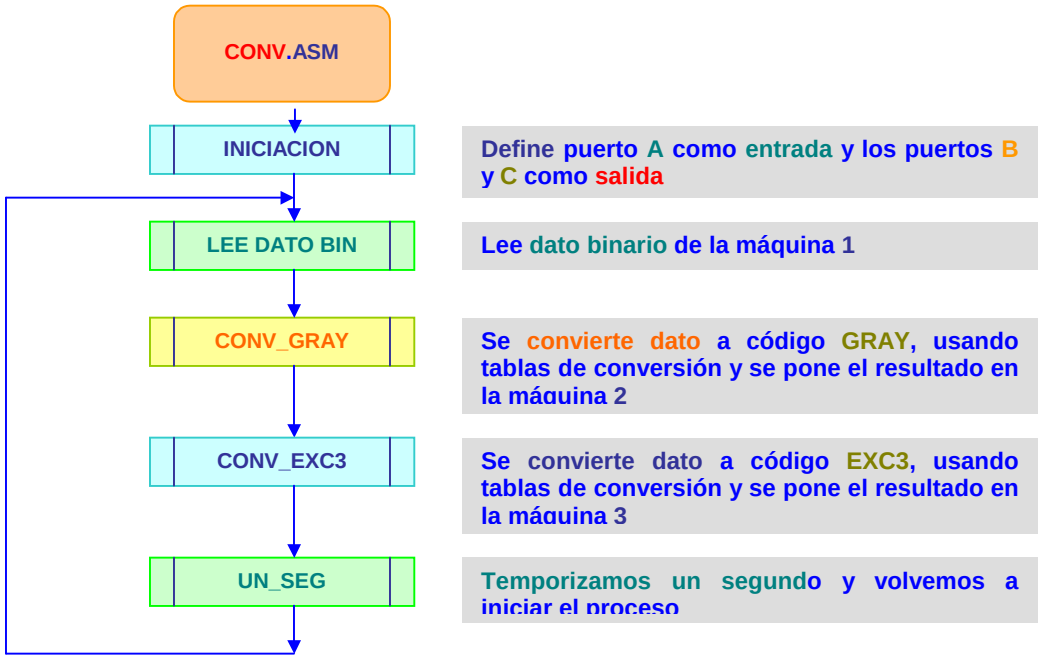
SOLUCIÓN

La ***solución*** del *hardware* (**mecamática**) se presenta en el siguiente diagrama a bloques:



Configuración de los Circuitos Integrados		Localidades	
PROM		Las últimas de 1K	
8255	E/S	Dirección 40 41 42 43	Contenido MAQ1 = PA Entrada MAQ2 = PB Salida MAQ3 = PC Salida CONTROL

Diagrama de flujo

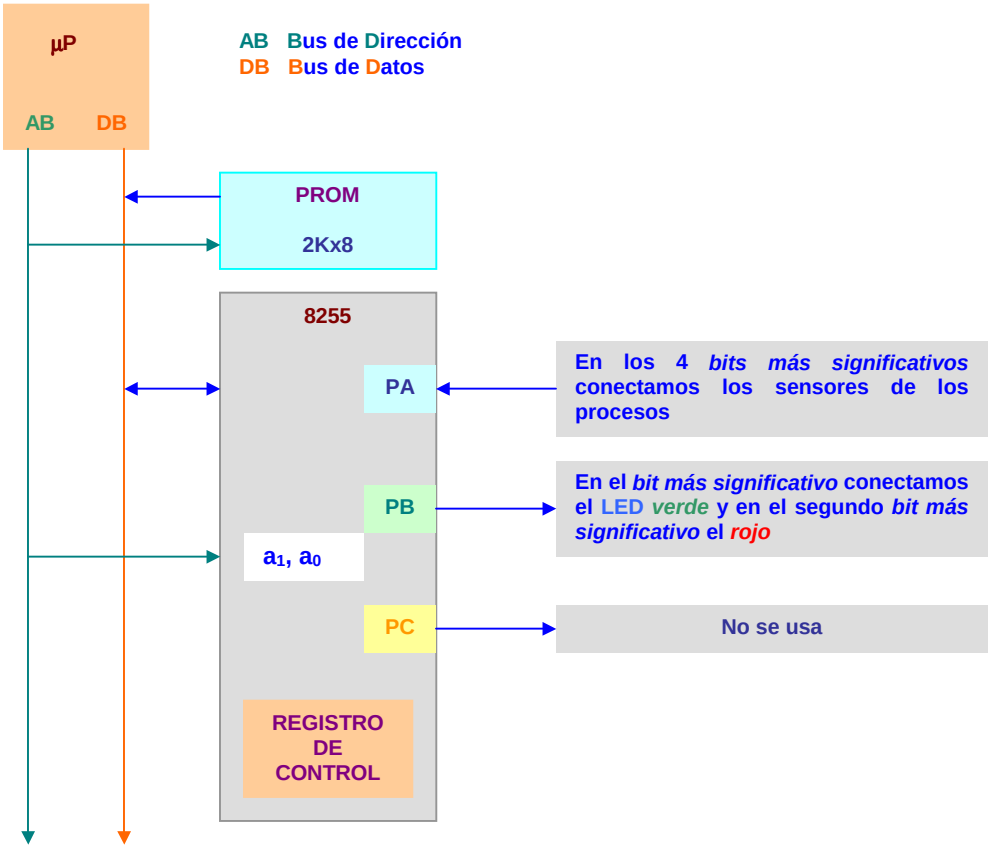


Ejemplo 3. Cierto proceso debe llevarse a cabo en forma ***secuencial*** cada segundo por cada paso. El **proceso** se realiza empezando por ***A***, después ***B***, a continuación ***C*** y finalmente ***D***. Si se efectúa en el orden indicado deberá activarse desde el inicio un *diodo emisor de luz* (***LED***, *por sus siglas en inglés*) ***verde*** y al finalizar el proceso, retornar al programa principal.

Si el proceso no se lleva a cabo en el orden mencionado, lo llevaremos al estado ***ERROR*** y se debe activar un ***LED rojo*** para indicar esta condición. El circuito deberá permanecer en ese estado hasta que se aplique un pulso de ***RESET*** retornando al estado inicial.

SOLUCIÓN

Para la parte del *hardware*, proponemos un sistema mínimo conteniendo una ***PROM*** de **2Kx8** y un *adaptador de periféricos en paralelo* **8255**, como se muestra en el siguiente diagrama a bloques:

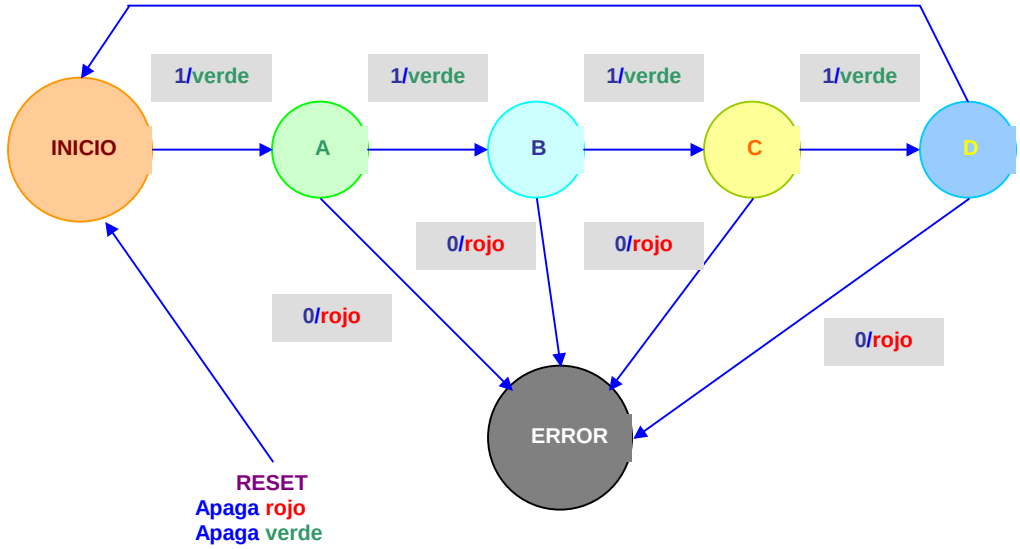


Vamos a ubicar la ***PROM*** en las ***últimas localidades*** del espacio de direccionamiento y al **8255** en las ***primeras localidades*** del espacio de ***ENTRADA/SALIDA***, como se indica en la siguiente tabla:

Dirección de E/S	Puerto	
0	A	Entrada
1	B	Salida
2	C	No usado
3	Registro de control	

Se deben leer los **4** sensores conectados al puerto ***A*** cada segundo.

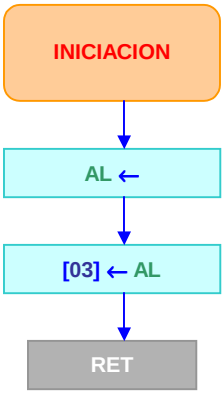
Diagrama de estados:



Subrutinas requeridas:

INICIACIÓN
ACTIVA_ROJO
ACTIVA_VERDE
UN_SEGUNDO

Diagrama de flujo



Puerto A = ENTRADA
Puerto B = SALIDA
Puerto C = NO USADO