2011

ASSEMBLER

UNNOBA – Arquitectura de Computadoras

Mariano Ruggiero
UNNOBA

INTRODUCCIÓN

SET DE INSTRUCCIONES

Cada CPU tiene un conjunto de instrucciones que puede ejecutar. Estas instrucciones son parte del microprocesador y es el fabricante (Intel, AMD, IBM…) quien determina a qué instrucciones responde el mismo.

> ***Set de Instrucciones:*** *Es el conjunto de todas las instrucciones que una CPU o microprocesador puede ejecutar.*

El set de instrucciones es propio de una arquitectura de CPU en particular. Cada arquitectura tiene su propio set y los programas que corren sobre una CPU deben estar diseñados específicamente para ese set de instrucciones[1].

El set de instrucciones de una CPU también se conoce como **lenguaje de máquina**. Una instrucción en lenguaje de máquina es una secuencia de bits (ceros y unos) que la CPU sabe interpretar y le indica qué operación realizar.

UNA INSTRUCCIÓN

Una instrucción en lenguaje de máquina tiene un aspecto similar al siguiente:

```
10111000000000010100000000
```

Y un pequeño fragmento de un programa escrito en lenguaje de máquina se varía así:

```
10111000000000010100000000
10111000000000001000000000
10100011000001111000000000
```

Claramente, es imposible escribir código directamente en lenguaje de máquina. Este lenguaje es totalmente ilegible para un humano y tener que memorizar todas las combinaciones de ceros y unos que forman cada instrucción o, peor aún, tener que encontrar un error entre una lista de cientos de instrucciones escritas así, es impensable.

LENGUAJES DE ALTO NIVEL

Para evitar tener que escribir en lenguaje de máquina, habitualmente utilizamos los llamados **lenguajes de alto nivel**. Estos lenguajes (como C++, Pascal, Ruby, C#, Visual Basic, Java y otros) poseen instrucciones entendibles por los humanos (programadores) que hacen mucho más fácil escribir y leer el código fuente de un programa. De hecho, estos lenguajes nos simplifican la tarea aún más, agrupando dentro de una **instrucción de alto nivel** varias instrucciones del lenguaje de máquina (o **lenguaje de bajo nivel**).

[1] A partir de 2006, Apple, la compañía fabricante de las computadoras Macintosh, decidió cambiar la arquitectura **PowerPC**, utilizada durante mucho tiempo en las Macs, por la arquitectura **Intel x86**. Como resultado, las aplicaciones creadas para la arquitectura anterior debían correr bajo un **emulador** en las nuevas Intel Macs que traducía las instrucciones originales a instrucciones que la arquitectura Intel x86 pudiera comprender, de esta forma se mantuvo la compatibilidad con las aplicaciones más antiguas.
Otro efecto del cambio de arquitecturas fue que ahora los usuarios de Mac podían elegir entre usar el sistema operativo de Apple, *MacOS X*, o bien instalar *Windows*, que antes sólo estaba disponible para las PCs (que se basan en la arquitectura x86).

Las tres instrucciones de bajo nivel mostradas anteriormente sirven para sumar 5 + 2 y almacenar el resultado. Sí, todos esos ceros y unos lo único que logran es sumar dos números… ¿Cómo haríamos lo mismo en un lenguaje de alto nivel? Sencillamente:

```
resultado = 5 + 2
```

¿Por qué se necesitan *tantas* instrucciones de bajo nivel, entonces? Porque las instrucciones de bajo nivel solo realizan tareas sumamente sencillas. Esto hace que los microprocesadores sean mucho más fáciles y baratos de construir. La gran potencia de los microprocesadores es consecuencia de su altísima velocidad que le permite ejecutar millones de esas instrucciones sencillas por segundo (y combinando millones de instrucciones sencillas se logran resultados muy complejos, como puede ser cualquiera de los programas que usamos habitualmente).

> *Cada **instrucción de bajo nivel** realiza una sola tarea básica.*

COMPILADORES

Para convertir el código fuente escrito en un lenguaje de alto nivel a código de máquina se utiliza una herramienta de software especial llamada compilador.

> *Un **compilador** convierte código fuente de alto nivel en instrucciones de máquina de bajo nivel.*

Cada lenguaje de programación posee un compilador propio. El compilador actúa como un traductor. Toma como entrada el listado de código de alto nivel y, por cada instrucción, la transforma en una o más instrucciones de bajo nivel.

El "problema" de los lenguajes de alto nivel es que, a cambio de facilidad de uso, estamos resignando control. Al permitir que el compilador escriba por nosotros el código de máquina, dejamos de tener el control absoluto sobre cada operación que realiza la CPU al ejecutar nuestro programa. Esto es generalmente algo deseable, pero existen casos donde es interesante poder tener el control directo de lo que sucede en nuestro programa.

Dado que no podemos controlar las instrucciones de máquina que genera el lenguaje de alto nivel y que escribir directamente en código de bajo nivel es demasiado difícil, ¿qué alternativa tenemos?

ASSEMBLER

Para cada lenguaje de máquina (de cada arquitectura de CPU) existe lo que se conoce como lenguaje ensamblador o **Assembler**. Assembler no es realmente un *lenguaje*, sino que es simplemente una *representación legible* del lenguaje de máquina. Existe una correspondencia uno a uno entre cada instrucción de bajo nivel y cada instrucción de Assembler.

> ***Assembler** es un lenguaje que representa cada una de las instrucciones de bajo nivel de forma legible para los programadores.*

Si traducimos las tres instrucciones que habíamos visto anteriormente a su representación en Assembler, obtendremos lo siguiente:

```
mov ax, 5
add ax, 2
mov resultado, ax
```

Más allá de que no nos quede claro en este momento qué es exactamente lo que hace cada instrucción, es evidente que esta representación del lenguaje de máquina es mucho más "amigable" para los humanos que la expresión original en binario. Podemos imaginarnos que escribir en este lenguaje sí es factible y que encontrar errores es una tarea más sencilla.

Para convertir código Assembler en código de máquina, en lugar de un compilador, se utiliza un **ensamblador**. A pesar de que tiene un objetivo similar al compilador, la gran diferencia es que un ensamblador no *genera* nuevo código sino que simplemente convierte cada instrucción de Assembler en una única instrucción de máquina.

> *El **ensamblador** es un programa que traduce el código Assembler a la representación binaria (código de máquina) que el microprocesador puede comprender.*

NUESTRO OBJETIVO

Hoy en día el uso de Assembler como lenguaje de programación para cualquier proyecto de desarrollo es muy poco común. Los lenguajes, en realidad, tratan de alejarse cada vez más del código de máquina, generando código de más alto nivel, que nos evita tener que lidiar con los detalles. Nosotros podemos escribir código en un lenguaje que nos sea cómodo, que nos brinde mucha funcionalidad lista para usar y dejar en manos del compilador la tarea de generar un código de bajo nivel eficiente y veloz que haga lo que queremos.[2]

Entonces, ¿para qué aprender Assembler? La gran ventaja de aprender Assembler es que nos permite entender cómo funciona un programa compilado a nivel del microprocesador. Al existir una correspondencia directa entre las instrucciones de Assembler y las de código de máquina, sabemos que cada instrucción que escribimos es una que el procesador va a ejecutar y esto nos permite ver qué es lo que realmente debe hacer nuestro código de alto nivel para lograr las funcionalidades complejas a las que estamos acostumbrados.

Piensen en el ejemplo anterior. ¿Hubieran pensado que para sumar dos números el procesador necesita tres instrucciones? Eso es algo que en el lenguaje de alto nivel no se nota, pero en Assembler sí.

LA ARQUITECTURA INTEL 8086

En nuestro estudio de Assembler, nos basaremos en la arquitectura Intel 8086. Esta es la arquitectura del microprocesador utilizado en la IBM PC original. A pesar de que la arquitectura ha evolucionado mucho con el tiempo, los principios generales se han mantenido estables[3]. Estudiar la arquitectura original hace que nuestro aprendizaje de los conceptos básicos sea más fácil.

[2] Ocasionalmente tal vez necesitemos escribir una porción de código en Assembler porque necesitamos el control absoluto y no podemos dejar la generación de código al compilador. Para ese caso, muchos lenguajes de alto nivel incluyen lo que se conoce como ensamblador en línea (*inline assembly*), que nos permite escribir un bloque de código Assembler dentro de nuestro código de alto nivel.
También existen ramas de la programación donde la codificación en Assembler aún tiene su lugar, como por ejemplo, la programación de sistemas de tiempo real, el desarrollo de sistemas operativos y la implementación de software embebido.

[3] La arquitectura se conoce en general como Intel x86 y abarca todos los procesadores desde el 8086 original hasta los más recientes procesadores de múltiples núcleos. El hecho de que, a pesar de evolucionar, la arquitectura se haya mantenido estable es lo que nos permite tener compatibilidad hacia atrás y poder ejecutar hoy programas escritos hace más de diez años.

HERRAMIENTAS

Para programar en Assembler necesitamos tres herramientas:

- Un editor donde escribir el código fuente.
- Un ensamblador que traduce el programa fuente a lenguaje de máquina.
- Un vinculador o *linker* transforma el código en lenguaje de máquina en un ejecutable (*.exe*).

Como editor puede utilizarse cualquier editor de texto plano (sin formato), como el *Bloc de Notas* que viene con Windows o el *Edit* de DOS. Para ensamblar el programa, usaremos el programa *MASM* de Microsoft. Y para generar el ejecutable usaremos el programa *LINK*.

CREACIÓN DE UN EJECUTABLE A PARTIR DE CÓDIGO ASSEMBLER

Una vez escrito el código fuente del programa, lo guardamos con extensión *.asm*. Para generar el código de máquina, desde la línea de comandos[4], usamos MASM así[5]:

```
MASM Nombre;
```

Donde *Nombre.asm* es el nombre que le dimos a nuestro código fuente[6]. Esto dará como resultado un archivo *Nombre.obj* (es decir, un archivo con el mismo nombre que el original, pero con extensión *.obj*), conteniendo el código de objeto. El código de objeto es código de máquina, pero que todavía no tiene el formato de ejecutable requerido por el sistema operativo.

Para generar el ejecutable, usamos LINK:

```
LINK Nombre;
```

Nuevamente, *Nombre.obj* es el nombre del archivo de objeto. Como resultado de este paso, obtendremos *Nombre.exe*.

[4] Para acceder a la línea de comandos de Windows: ir a *Inicio > Ejecutar*, escribir **cmd** y presionar *Enter*. Luego navegar hasta la carpeta que contiene el archivo Assembler. Con el comando **cd "NombreCarpeta"**, se accede a una subcarpeta de la actual y con **cd..** se vuelve a la carpeta anterior (luego de cada comando hay que presionar *Enter*).

[5] Para poder escribir estas instrucciones, los programas MASM.exe y LINK.exe deben encontrarse en la *misma* carpeta que el archivo de código Assembler.

[6] Dado que los programas MASM y LINK no soportan nombres largos, para que nuestros programas ensamblen y enlacen correctamente, deberemos asegurarnos que el nombre que le pongamos al archivo de código fuente tenga **8 caracteres o menos** (*además* del **.ASM**).

PRIMEROS PASOS

PRIMER PROGRAMA EN ASSEMBLER

Programar en Assembler no es difícil, sólo lleva un poco más de trabajo que hacerlo en un lenguaje de alto nivel porque las instrucciones son mucho más simples, hacen menos cosas. Pero esa es también una ventaja: podemos saber exactamente qué sucederá cuando se ejecute una instrucción, así que para hacer un programa sólo tenemos que asegurarnos de darle las instrucciones adecuadas al procesador.

Como primer programa, analicemos el mismo ejemplo que habíamos presentado en la sección anterior:

```
mov ax, 5
add ax, 2
mov resultado, ax
```

Ya dijimos que este fragmento de código Assembler lo que hace es sumar los números 5 y 2, y almacenar el resultado. Ahora veamos **cómo** funciona:

```
mov ax, 5
```

Esta línea asigna el número 5 al registro AX.

Los **registros** son pequeñas unidades de memoria que se encuentran dentro del procesador y podemos pensar en ellos como si fueran *variables*. Lo que hacen, básicamente, es almacenar un dato. Existen varios registros que se utilizan para distintas funciones y los veremos más adelante. **AX** es uno de los registros disponibles en la CPU.

La instrucción **MOV** es una instrucción de asignación[7]. Es equivalente al signo = de los lenguajes de alto nivel. Lo que hace es copiar lo que está del lado derecho de la coma sobre el registro que está del lado izquierdo.

> ***MOVE Destino, Valor**. Se utiliza para realizar la asignación **Destino = Valor**.*

```
add ax, 2
```

Esta instrucción es un poco más complicada de entender. La instrucción **ADD** es una instrucción de suma[8]. Lo que hace es sumar los dos elementos que se escriben a continuación (en este caso AX y 2) y *almacena el resultado sobre el primer elemento*.

Al principio el comportamiento puede resultar un poco extraño, ¿no? Podemos entenderlo mejor si leemos "`add ax, 2`" como "sumarle a AX el número 2".

> ***ADD Destino, Incremento**. Se utiliza para realizar la suma **Destino = Destino + Incremento**.*

En *todas* las instrucciones de Assembler donde haya que almacenar un resultado, este *siempre* se almacenará en el **primer operando**. Ya vimos que esto es así para las instrucciones MOV y ADD.

[7] **MOV** viene de la palabra en inglés *move* que significa "mover".
[8] **ADD** en inglés significa "sumar".

Recapitulando, ¿qué tenemos hasta ahora?

```
mov ax, 5
add ax, 2
```

Con la primera instrucción guardamos un 5 en AX. Con la segunda, sumamos un 2 a AX, con lo que AX pasa a tener un valor de 7.

La última instrucción es otro MOV:

```
mov resultado, ax
```

La palabra RESULTADO representa en este código una ubicación de memoria, una **variable**. A diferencia de los registros, las variables se almacenan en la memoria principal del sistema, la RAM[9].

Con esta línea, asignamos a la variable RESULTADO el valor de AX, con lo cual RESULTADO pasará a tener el valor 7.

Ahora que podemos entender cada paso, repasemos el código original:

```
mov ax, 5           ; Cargar AX con 5
add ax, 2           ; Sumarle 2
mov resultado, ax   ; Almacenar la suma en la variable RESULTADO
```

En el listado anterior lo que hicimos fue agregar **comentarios**. Los comentarios en Assembler se escriben comenzando por un punto y coma. Todo lo que sigue del punto y coma se considera un comentario y es ignorado por el ensamblador.

Al programar en Assembler, el uso de comentarios es fundamental. Dado que el Assembler es un lenguaje muy escueto y se necesitan muchas instrucciones para lograr comportamientos sencillos, los comentarios ayudan muchísimo a entender qué es lo que el código hace.

EL PRIMER PROGRAMA COMPLETO

Lo que acabamos de ver es sólo un *fragmento* de código Assembler. Para poder realizar un *programa* en Assembler debemos agregar algunas instrucciones más.

A continuación veremos nuevamente el código anterior, pero esta vez en la forma de un programa completo. Este programa puede escribirse en un editor de texto y luego ensamblarlo y enlazarlo para generar un ejecutable como se explicó en la sección anterior.

[9] **RAM** viene de *Random Access Memory*, que en inglés significa "Memoria de Acceso Aleatorio".

```
.MODEL SMALL
.STACK
.DATA

    resultado DW ?       ; Declarar la variable RESULTADO de 16 bits

.CODE
inicio:

    mov ax, @data        ; Inicializar el segmento de datos
    mov ds, ax           ;

    mov ax, 5            ; Cargar AX con 5
    add ax, 2            ; Sumarle 2
    mov resultado, ax    ; Almacenar la suma en la variable RESULTADO

    mov ax, 4C00h        ; Terminar
    int 21h              ;

END inicio
```

Como podemos ver, es un poco más complejo que lo que teníamos hasta ahora. Pero hay mucho del código anterior que, para simplificar, podemos tomar como una *fórmula estándar*. Es decir, si bien cada línea tiene su significado y puede ser alterada para lograr distintos resultados al ensamblar el programa, por ahora no nos preocuparemos por ellas y simplemente las usaremos así.

Entonces, el esqueleto de *cualquier* programa en Assembler que hagamos será el siguiente:

```
.MODEL SMALL
.STACK
.DATA

    ; --> Acá va la declaración de variables <--

.CODE
inicio:

    mov ax, @data        ; Inicializar el segmento de datos
    mov ds, ax           ;


    ; --> Acá va nuestro código <--

    mov ax, 4C00h        ; Terminar
    int 21h              ;

END inicio
```

Es decir, que más allá de todas las líneas que son estándar, tenemos sólo dos secciones importantes en un programa Assembler:

- El **segmento de datos** (marcado por la directiva **.DATA**) que es donde se declaran todas las variables que utiliza el programa.
- El **segmento de código** (marcado por la directiva **.CODE**) que siempre comienza y termina con las instrucciones que vemos en el modelo, entre las cuales ubicaremos nuestro código.

CONCEPTOS BÁSICOS

DATOS

El código de máquina trabaja exclusivamente con datos binarios. Si bien estos datos se pueden interpretar de distintas formas (como un número entero, como un real, como un carácter), a nivel de código de máquina no hay ninguna distinción entre, por ejemplo, un entero y un carácter.

La secuencia de bits 01100001 puede ser *interpretada* como el número 97 y como la letra "a", dependiendo de la operación que se realice sobre él. Lo importante es entender que en el código de máquina (y por lo tanto en Assembler) *no existen los tipos de datos*.

> *En Assembler no existen los tipos de datos. El significado de una secuencia de bits depende de la operación que se ejecute sobre ella.*

Lo que sí debemos tener en cuenta es el **tamaño** de los datos. En la arquitectura Intel 8086 existen dos tamaños posibles para los datos:

- **8 bits** (también llamado 1 **byte**)
- **16 bits** (también llamado 1 **word** o **palabra**)

Siempre debemos asegurarnos que el tamaño de los operandos en una instrucción sea el correcto. Por ejemplo, si hacemos un MOV (como en el ejemplo anterior), ambos operandos deben ser de 8 bits o bien deben ser ambos de 16 bits, pero *no se pueden mezclar* bytes con words.

Todos los datos tienen un **tamaño fijo**. Por eso, si tenemos el valor binario **10010**, este se escribirá como **00010010** en binario de 8 bits y como **0000000000010010** en binario de 16 bits. Siempre se escriben los *ceros a la izquierda* para indicar el tamaño exacto del dato.

Para simplificar la expresión de números de 8 y 16 bits es común utilizar el sistema de numeración hexadecimal (base 16). Cada dígito hexa representa 4 bits. De esa forma, por ejemplo, la palabra 1001111001011100 puede escribirse como 9E5C. Utilizaremos esta forma abreviada en este apunte.

SINTAXIS

Dentro del código Assembler, los datos se pueden expresar en binario, hexadecimal y decimal. Para identificar en qué sistema de numeración se está expresando un valor se utilizan las siguientes reglas:

- A los datos binarios se les agrega una **b** al final. Por ejemplo: `1001111001011100b` es un dato en binario.
- A los datos hexadecimales se les agregar una **h** al final. Por ejemplo: `9E5Ch` es un dato en hexa.[10]
- Los datos decimales se escriben normalmente. Por ejemplo: `40540` es un dato en decimal.

[10] En Assembler, cuando un número hexa comienza por un dígito entre A y F (una "letra"), debe agregarse un cero delante del número. Por ejemplo, A001 debe escribirse como `0A001h`. Ese cero que se agrega no incrementa la cantidad de bits del número, es simplemente una forma de escribirlo para que el ensamblador no lo confunda con un nombre de variable. En este apunte, a menos que sea dentro de un ejemplo de código, *omitiremos el cero inicial* para evitar confusiones.

BITS Y BYTES

Para un dato cualquiera (una secuencia de 8 o 16 bits), los bits se numeran de derecha a izquierda comenzando por 0. Por ejemplo, para el byte 00111001:

0	0	1	1	1	0	0	1
bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**

El **bit 0** se conoce como **bit menos significativo** o *least significat bit* (**LSB**).

El **último bit** (bit 7 para los bytes y bit 15 para las palabras) se conoce como **bit más significativo** o *most significant bit* (**MSB**).

También se habla de los bits "más altos" o "más bajos". Un bit es más bajo si está más cerca del bit 0 y más alto si está más cerca del último bit.

En el caso de las palabras, pueden dividirse en dos bytes:

bits 15 al 8	bits 7 al 0
byte alto	**byte bajo**

Estos bytes se conocen como **byte alto** (o **parte alta**) y **byte bajo** (o **parte baja**). También se los suele llama **byte más significativo** y **byte menos significativo**, respectivamente.

REGISTROS

Casi todas las instrucciones de Assembler involucran la utilización (lectura o escritura) de un registro. Los registros, como dijimos, son áreas de memoria dentro de la CPU que pueden ser accedidas muy rápidamente, sin pasar por el bus del sistema. A los efectos de la programación, los registros funcionan en forma similar a las variables, sirviendo para almacenar datos; pero se diferencian de ellas puesto que las variables se ubican en la RAM.

> *Un **registro** es un área de memoria dentro de la CPU que se puede acceder rápidamente, sin pasar por el bus del sistema.*

En el procesador Intel 8086 existen registros de 8 y de 16 bits para almacenar los dos tamaños de datos que nombramos antes (bytes y words).

A continuación presentamos una tabla con todos los registros del procesador Intel 8086:

	Word (16 bits)																
	Byte alto								Byte bajo								
Bits:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
AX	AH								AL								Acumulador
BX	BH								BL								Base
CX	CH								CL								Contador
DX	DH								DL								Datos

CS		Seg. de Código
DS		Seg. de Datos
ES		Segmento Extra
SS		Segmento de Pila

IP		P. de instrucción

SI		Índice de Fuente
DI		Índice de Destino

SP		Puntero de Pila
BP		Puntero de Base

Flags					OF	DF	IF	TF	SF	ZF		AF		PF		CF	(Flags)

Por ahora, nos concentraremos sólo en los primeros cuatro registros, conocidos como **registros de propósito general**. Dejaremos los demás registros para un más adelante.

> ***Registros de propósito general****: son los registros **AX, BX, CX** y **DX** (y sus fracciones AH, AL, BH, BL, etc.) que, a diferencia de los demás registros de la CPU, pueden utilizarse para almacenar cualquier dato, tanto de 8 como de 16 bits.*

Los registros de propósito general (AX, BX, CX y DX) pueden almacenar un valor de 16 bits cada uno. Por ejemplo, para asignar el valor 0057h a CX lo hacemos así:

```
mov cx, 0057h
```

Además, estos registros se subdividen en dos, su parte alta y su parte baja, que pueden ser accedidos independientemente. Por ejemplo, BX se divide en BH y BL[11]. Para asignar un valor de 8 bits, por ejemplo 2Fh, a la parte alta de BX lo hacemos así:

```
mov bh, 2Fh
```

Los registros de propósito general son los **únicos** que pueden subdividirse en partes de 8 bits.

[11] Las letras **L** y **H** vienen de *low* y *high*, en inglés, que significan “bajo” y “alto”.

ALGUNAS INSTRUCCIONES BÁSICAS

INSTRUCCIONES DE ASIGNACIÓN

MOV

Ya la vimos anteriormente, viene de *move* y sirve para realizar un asignación. La forma general es:

```
MOV Destino, Fuente
```

Su función es asignar el valor de `Fuente` a `Destino`. Sólo `Fuente` puede ser un número, `Destino` *no* porque no sería un lugar válido donde asignar un dato.

Ejemplos:

```
mov ax, 000Fh       ; Asignar el valor F en hexa a AX
mov cx, ax          ; Copiar el valor de AX en CX (CX pasa a valer
000Fh)
mov dl, 01111101b   ; Asignar el valor 1111101 en binario a DL
mov bh, 7           ; Asignar el valor 7 en decimal a BH
```

XCHG

XCHG nombre viene de *exchange* y su función es intercambiar los valores de los dos operandos.

```
XCHG Operando1, Operando2
```

Luego de ejecutar esta instrucción, `Operando1` tendrá el valor original de `Operando2` y viceversa. Claramente, *no puede* utilizarse un número como ninguno de los operandos, ya que no habría lugar donde almacenar uno de los datos.

Ejemplo:

```
mov ax, 1     ; AX = 1
mov bx, 2     ; BX = 2

xchg ax, bx   ; Intercambiar AX y BX
              ; Ahora AX = 2 y BX = 1
```

INSTRUCCIONES ARITMÉTICAS BÁSICAS

ADD

Lo vimos anteriormente. Suma los dos operandos y guarda el resultado sobre el primero.

```
ADD Destino, Operando2
```

Es decir, luego de ejecutar la operación, `Destino` tendrá su valor original más el `Operando2`. Igual que como con MOV, sólo el `Operando2` puede ser un valor numérico.

Ejemplo:

```
mov ah, 1    ; AH = 1
mov al, 2    ; AL = 2

add ah, al   ; AH = AH + AL
             ; Ahora AH = 3
```

INC

INC viene de *incrementar*. Es una instrucción especial que sirve para sumar 1.

```
INC Destino
```

Funcionalmente, es equivalente a `ADD Destino, 1` pero es *más rápida*. Es muy común que en los programas queramos incrementar algún registro en 1 (por ejemplo si estamos contando algo), por eso es que existe esta instrucción especial.

Ejemplos:

```
mov dx, 6    ; DX = 6
inc dx       ; DX = 7
inc dx       ; DX = 8
inc dx       ; DX = 9
```

SUB

SUB viene de *subtract* y sirve para realizar una resta.

```
SUB Destino, Operando2
```

El resultado de la operación sería equivalente a la siguiente expresión en un lenguaje de alto nivel:

```
Destino = Destino - Operando2
```

La resta es siempre en ese orden (`Destino - Operando2`) y no al revés.

Ejemplo:

```
mov dh, 10   ; DH = 10
mov bl, 2    ; BL = 2

sub dh, bl   ; DH = DH – BL
             ; Ahora DH = 8
```

DEC

DEC, de *decrementar*, sirve para restar 1.

```
DEC Destino
```

De forma similar a lo que sucede con INC, la instrucción anterior es equivalente a `SUB Destino, 1` pero es más rápida. También es muy común decrementar un registro en 1 en los programas Assembler, por eso es que existe esta instrucción especial.

UN EJEMPLO SIMPLE

A continuación presentamos un programa de ejemplo utilizando las instrucciones básicas que acabamos de ver.

Realice un programa en Assembler que calcule el resultado del siguiente cálculo aritmético y lo almacene en el registro DX.

(1988 + 9992) – (530 – 170)

```
.MODEL SMALL
.STACK
.DATA
.CODE
inicio:

    mov ax, @data        ; Inicializar el segmento de datos
    mov ds, ax           ;

    ; Resolver el primer paréntesis (el resultado queda en DX)
    mov dx, 1988
    add dx, 9992

    ; Resolver el segundo paréntesis (el resultado queda en AX)
    mov ax, 530
    sub ax, 170

    ; Resolver la resta entre los dos paréntesis
    ; (el resultado queda en DX)
    sub dx, ax

    mov ax, 4C00h        ; Terminar
    int 21h              ;

END inicio
```

Los programas que haremos serán programas no interactivos, sin salida por pantalla. Si bien estos no representan programas reales, son mucho más simples al momento de aprender Assembler.

Lo que nos interesa no es crear programas útiles con Assembler, sino entender cómo funcionan los programas que creamos. Para ver cómo trabaja el programa, paso a paso, podemos utilizar un **debugger** como el *Trubo Debugger* de Borland.

MEMORIA Y VARIABLES

LAORGANIZACIÓN DE LA MEMORIA

La mínima unidad de memoria que se puede leer o escribir en un procesador Intel 8086 son **8 bits** (1 byte). No se pueden acceder a fracciones inferiores.

> ***Mínima unidad de memoria direccionable**: es la mínima porción de memoria a la que se puede acceder (leer o escribir).*

En cambio, como máximo se puede acceder a **16 bits** (2 bytes o 1 palabra) simultáneamente.

> ***Máxima unidad de memoria direccionable**: es la máxima cantidad de bits que pueden ser accedidos (leídos o escritos) simultáneamente.*

La memoria puede ser pensada como una secuencia de bytes, uno detrás del otro. Generalmente se representa como una tabla vertical:

0Ah
F6h
14h
44h
…

A cada celda de memoria de 1 byte se le asigna una **dirección**. La dirección de memoria es un número que identifica a la celda en cuestión y nos permite hacer referencia a ella para escribir o leer un valor. A cada celda de memoria consecutiva se le asigna un valor incremental.

Dirección	Contenido
00001h	0Ah
00002h	F6h
00003h	14h
00004h	44h
…	…

Podemos pensar en las direcciones de memoria como si fueran las direcciones de viviendas. Si queremos encontrar un lugar en la ciudad, debemos conocer su dirección. Si queremos encontrar un byte en la memoria, también.

Las **celdas de memoria** del Intel 8086 tienen **8 bits** cada una. Las **direcciones de memoria** son números de **20 bits** que identifican a cada celda de memoria.

> *En el Intel 8086 las direcciones de memoria son números de **20 bits**.*

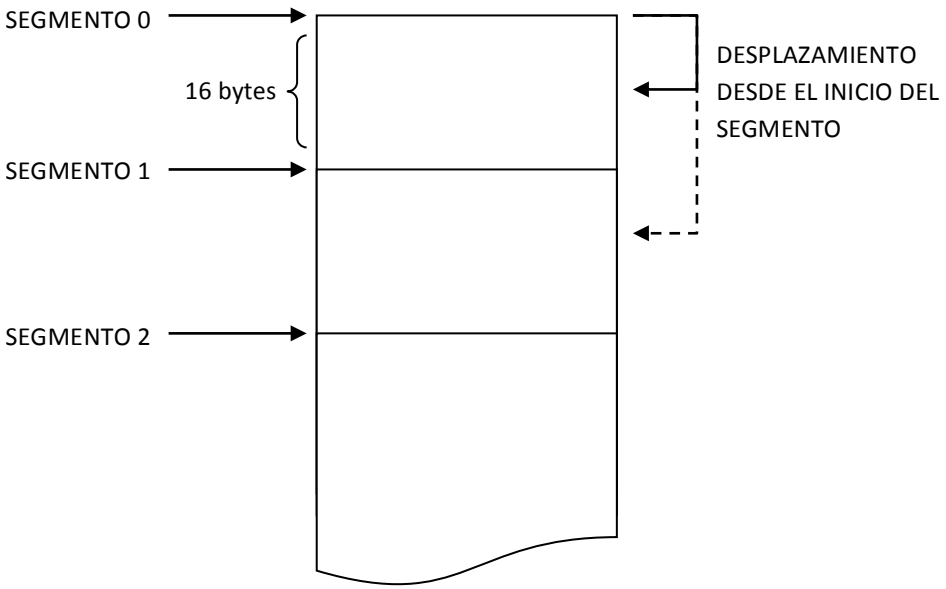
MEMORIA SEGMENTADA

Al diseñar la arquitectura 8086, Intel decidió permitir direccionar hasta 1 MB de memoria. Por eso el bus de direcciones se hizo de 20 bits (ya que 2^{20} = 1048576 = 1M). El problema es que, como dijimos, el procesador 8086 sólo puede manejar datos de 16 bits como máximo.

Para resolver este problema y permitir un tamaño mayor de memoria (con 16 bit sólo puede accederse a 2^{16} = 65536 = 64K), Intel implementó un sistema de **memoria segmentada**.

En este sistema, una memoria se expresa como dos números de 16 bits. El primero (llamado número de **segmento**), permite seleccionar una porción de memoria de hasta 64 KB; y el segundo (llamado **desplazamiento**) permite direccionar 1 byte dentro de ese segmento de memoria. La notación es la siguiente:

```
SEGMENTO : DESPLAZAMIENTO
```



Este esquema es similar al uso de meses y días en el calendario: cada fecha está formada por un número entre 1 y 12, el mes, y otro entre 1 y 31, el día dentro del mes. Esto es equivalente al segmento (mes) y al desplazamiento (día) de una dirección segmentada. La fecha se escribe día/mes y la dirección segmento:desplazamiento.

La única diferencia, es que en el esquema de memoria segmentada los segmentos se superponen. Cada segmento comienza 16 bytes después del segmento anterior, pero el desplazamiento permite direccionar 64 KB a partir del comienzo del segmento. Es claro que a partir del byte 17 estaremos "pisando" el segmento siguiente. Es como si en las fechas fuera válido poner 41/1 (que sería, en realidad, 10/2). Entonces, en el esquema de memoria segmentada un mismo byte puede nombrarse de varias formas, cambiando el segmento y el desplazamiento.

A menudo los datos de un programa se ubican dentro de un solo segmento, o bien el segmento se da por sobreentendido. En esos casos, es común referirse a la *dirección de memoria* 1000h cuando, en realidad, deberíamos decir formalmente el *desplazamiento* 1000h dentro del segmento actual.

SEGMENTOS EN LOS PROGRAMAS

Un programa utiliza distintos segmentos de memoria para almacenar sus datos e instrucciones. Dos de los segmentos más utilizados son el **segmento de datos**, donde se almacenan todos los datos (por ejemplo, las variables) que utiliza el programa, y el **segmento de código**, donde se encuentran las instrucciones del programa. En nuestro código, estos segmentos están marcados por las directivas **.DATA** y **.CODE**.

La CPU utiliza registros especiales, conocidos como **registros de segmentos**, para almacenar los segmentos correspondientes a cada área del programa. En particular:

- **DS**: almacena el segmento de **datos**.
- **CS**: almacena el segmento de **código**.

> *Los programas utilizan dos segmentos de memoria principales: el **segmento de código** (que se almacena en el registro **CS**) y el **segmento de datos** (que se almacena en el registro **DS**).*

Miremos algunas de las primeras líneas de nuestro esqueleto básico de código Assembler:

```
.DATA

    ; --> Acá va la declaración de variables <--

.CODE
inicio:

    mov ax, @data        ; Inicializar el segmento de datos
    mov ds, ax           ;

    ; --> Acá va nuestro código <--
```

Vemos que la declaración de variables se hace debajo de *.DATA*. Todo lo que pongamos ahí irá a parar al segmento de datos del programa.

En *.CODE* arranca nuestro programa. Las dos primeras líneas lo que hacen es inicializar el registro DS con la dirección del segmento de datos. **@DATA** representa la dirección correspondiente al segmento de datos. Esto se copia en AX (MOV AX, @DATA) y luego de AX se pasa a DS. [12]

En Assembler una dirección se escribe así:

```
DS:[DESPLAZAMIENTO]
```

En lugar de DS puede ponerse otro registro de segmento, pero lo más común es direccionar sólo sobre el segmento de datos. Por ejemplo, para copiar el contenido del byte 22 dentro del segmento de datos a AL haríamos lo siguiente:

```
mov al, ds:[22]
```

Lo más común es escribir las direcciones de memoria utilizando números hexadecimales. De esa manera, el ejemplo anterior quedaría: `mov al, ds:[16h]`.

[12] Esto se hace así porque existen restricciones en cuanto a cómo puede asignarse valores a un registro de segmento. La única operación MOV permitida que tiene a DS como destino de la asignación es MOV DS, AX. Por lo tanto, hay que cargar el valor primero en AX y luego pasárselo a DS.

DIRECCIONAMIENTO DE DATOS DE TAMAÑO WORD

Dijimos que la memoria está dividida en bytes. ¿Cómo podemos, entonces, almacenar datos de 16 bits? Simplemente, utilizando dos celdas de memoria.

La arquitectura Intel 8086 almacena los datos de tamaño word en un formato llamado **little endian**. En este formato se guarda primero la parte baja (los bits 0 a 7) y luego la parte alta (los bits 8 a 15). Es decir, para almacenar el dato 1011111000100000h (BE20h) a partir de la dirección 1000h:

Dirección	Contenido
1000h	20h
1001h	BEh

El número queda almacenado "al revés". Si lo leyéramos en orden quedaría 20BEh, en lugar de BE20h.

> *Al almacenar números de 16 bits, se almacena **primero la parte baja y luego la parte alta** (formato **little endian**).*

Para escribir o leer un dato de 16 en memoria, debemos especificar sólo la dirección del primero de los bytes (la parte baja). Por ejemplo, si queremos cargar en AX el dato del ejemplo anterior:

```
mov ax, ds:[1000h]
```

El ensamblador determina qué porción de la memoria leer en base al otro operando. En este caso, como pusimos AX y AX es un registro de 16 bits, traeremos de memoria 16 bits (los ubicados en 1000h y 1001h). Si hubiéramos puesto, por ejemplo, AL que es de 8 bits, hubiéramos leído sólo el byte ubicado en 1000h.

De lo anterior se desprende que siempre que utilicemos una dirección de memoria en una instrucción, **el otro operando debe ser un registro**, para que le indique al ensamblador si estamos trabajando con 8 o 16 bits.

Otra opción es utilizar los descriptores de tamaño **BYTE PTR** y **WORD PTR** para indicar el tamaño de un dato de memoria. De esa forma, no hay problema en utilizar una dirección y un número:

```
mov word ptr ds:[1000h], 250
```

Sabiendo que 250 es 00FAh en binario de 16 bits, la instrucción anterior resulta en:

Dirección	Contenido
1000h	FAh
1001h	00h

> ***BYTE PTR** indica que la dirección de memoria hace referencia a un dato de 8 bits (1 byte).*
> ***WORD PTR** indica que la dirección hace referencia a un dato de 16 bits (1 word).*

VARIABLES EN ASSEMBLER

Una variable en Assembler es simplemente un nombre que se da a una dirección de memoria. Trabajar con los desplazamientos de memoria directamente es poco práctico ya que si agregamos datos o movemos datos de lugar, la dirección de memoria cambiará y deberemos corregir todos los puntos en el código que utilizan esa dirección. Al utilizar

nombres simbólicos (variables) para las direcciones de memoria, nos olvidamos del verdadero valor de la dirección y dejamos que el ensamblador se ocupe de reemplazar el nombre de la variable por la dirección apropiada.

> *Una **variable** en Assembler es un nombre que se asigna a una dirección de memoria. El ensamblador se encarga de reemplazar las variables de nuestro código por la dirección de memoria apropiada.*

Las variables se declarar en la sección **.DATA** del código utilizando el siguiente formato:

```
NOMBRE TAMAÑO VALOR_INICIAL
```

NOMBRE es, claramente, el nombre que queremos darle a la variable, igual que en los lenguajes de alto nivel.

TAMAÑO indica si el dato a almacenar en esa variable es un byte (8 bits) o una word (16 bits). El tamaño se especifica utilizando DB o DW, para bytes o words respectivamente.[13]

> ***DB**: Indica que la variable almacena valores de tamaño byte. **DW**: Indica que la variable almacena valores de tamaño word.*

Ejemplos:

```
.DATA
    NumeroPrimo DW 29
    Mes         DB 6
    Resultado   DW ?
```

`NumeroPrimo`, por ejemplo, reservará un espacio de *2 bytes* y en él escribirá el número 29 (en binario de 16 bits). En memoria quedará:

00011101
00000000

`Mes` reservará un espacio de *1 byte*, inicializado con el valor 6:

00000110

Por último, `Resultado` reservará otro espacio de *2 bytes*, pero *no realizará una inicialización*. Esto significa que el contenido de la memoria quedará con cualquier valor que hubiera allí en el momento de realizar la reserva de memoria (tal vez un valor utilizado por algún otro programa que se ejecutó anteriormente). Por ejemplo (aunque podría ser *cualquier valor*):

01011001
11001100

> *Cuando utilizamos el signo **?** como valor inicial de la variable, sólo estamos reservando el espacio, pero sin establecer un valor inicial. La memoria quedará con el valor que tuviera hasta ese momento.*[14]

[13] Existen otros descriptores de tamaño (DD, DQ, etc.) pero no los nombramos porque no serán utilizados en este apunte.

Las variables quedan en memoria en forma **consecutiva**, tal cual como se fueron declarando en el código:

00011101
00000000
00000110
01011001
11001100

Vimos que para utilizar una dirección de memoria en nuestro código debemos utilizar la sintaxis `DS:[Dirección]`. Por ejemplo, si la dirección de la variable NumeroPrimo fuera 12AAh, para copiar su valor en DX utilizaríamos el siguiente código:

```
mov ax, ds:[12AAh]
```

Cuando utilizamos variables, la sintaxis se simplifica muchísimo:

```
mov ax, NumeroPrimo
```

Funciona exactamente igual que en los lenguajes de alto nivel.

OBTENER LA DIRECCIÓN DE UNA VARIABLE

Como dijimos, las variables nos abstraen de la dirección real del dato. Simplemente utilizando el nombre de la variable accedemos al *contenido* de la dirección de memoria. Sin embargo, a veces puede interesarnos conocer la *dirección* de una variable. ¿Cómo hacemos? Utilizamos el comando OFFSET:

```
mov bx, offset Resultado
```

Utilizando el ejemplo anterior, estamos almacenando en BX la dirección de la variable Resultado. Más adelante veremos cómo esto puede sernos de utilidad.

DECLARACIÓN DE ARREGLOS

Si queremos declarar un **arreglo** de datos (una secuencia ordenada de datos del mismo tamaño), lo hacemos con la misma sintaxis anterior, sólo que esta vez ponemos varios valores de inicialización separados por comas:

```
NOMBRE TAMAÑO_ELEMENTO ELEMENTO_0, ELEMENTO_1, ...
```

Por ejemplo:

```
.DATA
    Pares DB 2, 4, 6, 8
```

En memoria, obtendremos esto:

00000010
00000100
00000110
00001000

En el caso de los arreglos, la variable identifica la dirección del **primer elemento**. Por ejemplo:

[14] En general, dejar variables sin inicializar no es una buena idea y puede llevar a errores en el código si utilizamos inadvertidamente la variable sin asignarle un valor previamente.

```
mov ah, Pares
```

El código anterior carga el primer elemento del arreglo Pares en AH. Con lo que AH pasará a tener el valor 2 (00000010b).

Si queremos acceder a otros elementos del arreglo, podemos utilizar una notación de subíndice:

```
NOMBRE_ARREGLO[DESPLAZAMIENTO]
```

El valor DESPLAZAMIENTO, entre corchetes, indica cuántos **bytes** debemos avanzar desde la dirección del primer elemento. Por ejemplo, `Pares[1]` equivale al *segundo* elemento del arreglo Pares (ya que nos corremos 1 byte desde el inicio).

Otros ejemplos:

```
mov al, Pares[0]   ; Obtiene el primer  elemento (00000010)
mov ah, Pares[1]   ; Obtiene el segundo elemento (00000100)
mov bl, Pares[2]   ; Obtiene el tercer  elemento (00000110)
mov bh, Pares[3]   ; Obtiene el cuarto  elemento (00001000)
```

Cabe notar que `Pares` y `Pares[0]` son equivalentes, ya que `Pares[0]` indica un desplazamiento de 0 bytes desde el comienzo del arreglo.

> *Para obtener un elemento de un arreglo, usamos el nombre del arreglo seguido de un número entre corchetes. El número indica cuántos **bytes** debemos avanzar desde el comienzo del arreglo para obtener el elemento*

ARREGLOS DE WORDS

Los arreglos de words, se declaran igual que los de bytes, pero usando DW:

```
.DATA
    NumerosGrandes DW 1002h, 55A1h, 3037h, 000Fh
```

Este ejemplo, produce el siguiente resultado en memoria (recordar que se almacenan con el ordenamiento *little endian*).

02h
10h
A1h
55h
37h
30h
0Fh
00h

La regla para acceder a cada elemento del arreglo sigue siendo la misma: poner entre corchetes el desplazamiento **en bytes** desde el comienzo del arreglo. De modo que:

- Primer elemento: `NumerosGrandes[0]`
- Segundo elemento: `NumerosGrandes[2]`

- Tercer elemento: `NumerosGrandes[4]`
- Cuarto elemento: `NumerosGrandes[6]`

Simplemente, hay que recordar que el desplazamiento es **siempre en bytes**, más allá de que sea un arreglo de words.

ARREGLOS SIN INICIALIZAR

Si queremos declarar un arreglo pero no inicializarlo (no darle un valor distinto a *cada* elemento), podemos utilizar la directiva DUP. El siguiente ejemplo, declara un arreglo de 50 elementos de tipo byte, todos inicializados en 0.

```
Arreglo DB 50 DUP (0)
```

Es decir, se pone la cantidad de elementos (50), DUP y, entre paréntesis, el valor con que se inicializan *todos* los elementos. El ejemplo anterior reserva 50 lugares de 1 byte, todos con el valor 0.

Si no queremos inicializar el arreglo con *ningún* valor, podemos usar:

```
Arreglo DB 50 DUP (?)
```

CARACTERES

En Assembler, los caracteres se representan utilizando el código ASCII[15]. Cada carácter ASCII ocupa 8 bits, de modo que podemos declarar una variable para almacenar un carácter utilizando DB:

```
Caracter DB 41h
```

Para especificar el dato, podemos utilizar tanto el valor ASCII en decimal, hexadecimal (como en el ejemplo) o binario, o bien escribir el carácter entre comillas. Todas las representaciones son equivalentes y, en el último caso, es el ensamblador quien se encarga de hacer la traducción al momento de generar el código de máquina. El ejemplo anterior es equivalente a:

```
Caracter DB "A"
```

De hecho, `"A"` puede utilizarse como reemplazo de 41h en cualquier lugar del código (no sólo en la declaración de variables). Escribir `MOV AL, "A"` es totalmente válido. (AX ser cargará con 41h = 01000001b).

STRINGS

En Assembler, los strings no son más que un **arreglo de caracteres**. Por lo tanto, la declaración de un string es la siguiente:

```
Nombre DB "M", "a", "r", "i", "a", "n", "o"
```

Declaramos un arreglo de **bytes** e inicializamos cada elemento con el carácter ASCII correspondiente. Esto nos dará como resultado, en memoria, lo siguiente:

[15] ASCII: *American Standard Code for Information Interchange*, "Código Estadounidense Estándar para el Intercambio de Información". Se pronuncia generalmente como "aski". El código ASCII representa cada carácter como un número de 8 bits. Por ejemplo, la "A" es 01000001, la "B" es 01000010, etc.

4Dh
61h
72h
69h
61h
6Eh
6Fh

Para simplificar la notación, Assembler provee una forma más conveniente e intuitiva de inicializar un string:

```
Nombre DB "Mariano"
```

El efecto este código es *exactamente* el mismo que el del ejemplo anterior.

MODOS DE DIRECCIONAMIENTO

LOS OPERANDOS

Hasta ahora hemos visto varias formas de expresar los operandos de una instrucción en Assembler:

```
mov ax, bx            ; Utilizando registros solamente
mov dh, 7             ; Utilizando valores numéricos
mov dl, "x"           ; Utilizando caracteres ASCII
mov ds:[150Ah], cx    ; Utilizando una dirección de memoria explícita
mov bx, suma          ; Utilizando variables
mov ah, lista[5]      ; Utilizando variables de tipo arreglo
```

Los operandos de una instrucción se pueden clasificar básicamente en tres tipos:

- **Registro**: El operando está contenido en uno de los registros de la CPU. (En todos los ejemplos de arriba utilizamos registros en alguno de los operandos).
- **Dirección de memoria**: Ubicación dentro de la memoria del programa donde se encuentra el operando. (Podemos usar una dirección de memoria escribiéndola directamente o utilizando variables).
- **Constante**: Un valor especificado directamente en el código. (Por ejemplo un número o un carácter ASCII).

En general, para una operación con dos operandos (como MOV) existen las siguientes posibilidades[16]:

- MOV Registro, Registro
- MOV Registro, Constante
- MOV Registro, Dirección de memoria
- MOV Dirección de memoria, Registro
- MOV Dirección de memoria, Constante

Notar que **no existen operaciones entre direcciones de memoria**. Si, por ejemplo, se quiere copiar el contenido de una dirección de memoria en otra dirección, se debe llevar el dato primero a un registro y luego del registro a la nueva posición. Para llevar el contenido de la dirección 1BC0h a la dirección 9FC5h:

```
mov ax, ds:[1BC0h]   ; Copiar el contenido de la dirección 1BC0h en AX
mov ds:[9FC5h], ax   ; Pasar lo copiado en AX a la dirección 9FC5h
```

> ***No existen*** *operaciones que utilicen simultáneamente dos direcciones de memoria. Siempre que un operando es una dirección, el otro debe ser o bien un registro o bien una constante.*

Notar, además, que **las constantes sólo pueden utilizarse como operando derecho**, ya que el primer operando se utiliza para almacenar el resultado de la operación, con lo cual sólo puede ser un registro o una dirección de memoria.

[16] Hay que tener en cuenta que las posibilidades para cada operando dependen de cada instrucción en particular. Para conocer qué posibilidades existen para cada instrucción hay que leer la especificación de la arquitectura provista por el fabricante, en este caso Intel. Puede visitar http://www.intel.com/design/Pentium4/documentation.htm para encontrar los manuales de la arquitectura Intel x86 (formalmente conocida como IA-32).

MODOS DE DIRECCIONAMIENTO

Los **modos de direccionamiento** son las distintas formas que tenemos de indicar en una instrucción la ubicación de uno de los operandos. A continuación veremos los modos de direccionamiento posibles en la arquitectura Intel 8086:

INMEDIATO

Un operando utiliza **direccionamiento inmediato** si está escrito como una **constante** dentro de la instrucción. Por ejemplo, el operando derecho de esta instrucción utiliza direccionamiento inmediato:

```
add ax, 50
```

Para especificar un operando con direccionamiento inmediato podemos utilizar números binarios, decimales y hexadecimales, y también caracteres ASCII.

INHERENTE O DE REGISTRO

Un operando utiliza direccionamiento **inherente** (o de registro) si está ubicado dentro de un **registro** de la CPU. En el siguiente ejemplo, ambos operandos utilizan direccionamiento inherente:

```
mov ax, bx
```

DIRECTO

Un operando utiliza **direccionamiento directo** si está ubicado en una memoria y **la dirección de memoria se especifica explícitamente en la instrucción** o bien utilizando una variable. Ejemplos:

```
mov ax, ds:[107Ah]
mov bx, numeroPrimo
sub ch, arreglo[7]
```

INDIRECTO POR REGISTRO

Esta es una forma que no habíamos visto antes. Un operando utiliza **direccionamiento indirecto** si está ubicado en memoria y **la dirección de memoria se especifica a través de un registro**. Para indicarlo escribimos:

```
DS:[REGISTRO]
```

Los únicos registros admitidos en este modo son: **BX**, **BP**, **DI** y **SI**. Por ejemplo:

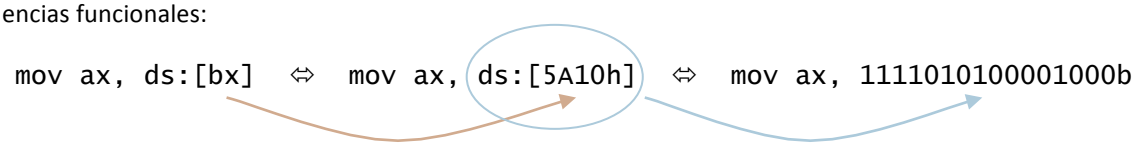
```
mov ax, ds:[bx]
```

Notar la diferencia con el ejemplo de direccionamiento inherente. Aquí BX, al estar entre corchetes, indica una dirección de memoria. Para realizar la operación hay que leer el contenido del registro y luego utilizar la dirección de memoria obtenida para encontrar el dato.

Supongamos que BX = 5A10h y que el contenido de la memoria es el siguiente:

Dirección	Contenido
5A10h	00001000
5A11h	11110101

Al ejecutar la instrucción del ejemplo, AX será cargado con 1111010100001000b. Podríamos hacer las siguientes equivalencias funcionales:



Las tres instrucciones tienen el mismo resultado: AX es cargado con 1111010100001000b. Pero la ubicación del dato es distinta en cada caso: la primera utiliza direccionamiento indirecto, la segunda direccionamiento directo y la tercera direccionamiento inmediato.

> *Siempre que escribimos **DS:[dirección]** estamos indicando que el dato se encuentra en **memoria**. Lo que va entre corchetes es la **dirección** del dato, ya sea que se escriba explícitamente (direccionamiento directo) o a través de un registro (direccionamiento indirecto).*
>
> *Si recordamos lo anterior, es fácil distinguir entre **MOV AX, BX** y **MOV AX, DS:[BX]**.*

MODOS DE DIRECCIONAMIENTO RELATIVO

Un operando utiliza **direccionamiento relativo** cuando **la dirección de memoria se especifica utilizando sumas de registros y constantes**. El formato es el siguiente:

```
DS:[REGISTRO_BASE + REGISTRO_ÍNDICE + CONSTANTE]
```

Los registros **base** son **BX** y **BP**. Los registros **índice** son **SI** y **DI**. Puede omitirse cualquiera de los dos registros (pero no ambos, ya que en ese caso tendríamos direccionamiento directo).

Ejemplos:

```
mov ax, ds:[bx + si + 5]
mov ds:[si + 01Ah], bh
add al, ds:[di + 1011b]
```

Es muy común utilizar direccionamiento relativo con arreglos. En ese caso, se acostumbra a utilizar la siguiente notación:

```
mov ax, arreglo[bx]
```

En este ejemplo, estamos utilizando un registro base (BX) y el lugar de la constante numérica lo toma el nombre de la variable. Otros ejemplo:

```
add bx, numeros[bx + di]
```

En este caso, tenemos tanto un registro base como uno índice. También puede utilizarse la notación `numeros[bx][di]`.

INSTRUCCIONES DE CONTROL

EJECUCIÓN DE PROGRAMAS

Para ejecutar un programa, la CPU realiza un ciclo de tres fases:

1. **Fetch - Búsqueda de la instrucción**: La CPU busca la instrucción a ejecutar en el segmento de código y la carga en la unidad de control.
2. **Decode - Decodificación de la instrucción**: La unidad de control interpreta el código de instrucción dando origen a los comandos necesarios para su ejecución.
3. **Execute - Ejecución de la instrucción**: Se ubican los operandos y se realiza la operación indicada por la instrucción sobre los mismos. Finalmente, se almacena el resultado de la operación.

Para conocer la ubicación de la instrucción a ejecutar, la CPU utiliza el registro **IP** (*Instruction Pointer*, "Puntero de Instrucción"). El IP contiene el desplazamiento dentro del segmento de código de la instrucción actual. (La dirección completa de la instrucción es **CS:IP**). Cada vez que se completa el *fetch* de una instrucción, IP es incrementado automáticamente para apuntar a la siguiente instrucción.

> *El registro **IP** almacena la dirección de la instrucción a ejecutar.*

Como los programas raramente siguen una secuencia lineal de ejecución sino que necesitan iterar y bifurcar, Assembler provee instrucciones especiales que nos permiten cambiar el registro IP.

BIFURCACIONES

JMP

La instrucción JMP (*Jump*) nos permite saltar a cualquier punto del código. La sintaxis es como sigue:

```
JMP nombre_etiqueta
```

Para poder indicar a qué lugar del código queremos saltar, debemos definir una **etiqueta**. Las etiquetas son nombres simbólicos para direcciones dentro del segmento de código (similares a las variables en el segmento de datos). Para definir una etiqueta, simplemente ponemos un nombre seguido de dos puntos (por ejemplo "inicio:") a la altura del código hacia donde queramos saltar.

```
inicio: . . . .
        . . . .
        . . . .
        jmp inicio
```

Lo que hace JMP es cambiar el puntero de instrucción (CS:IP) de manera que apunte a la dirección especificada por la etiqueta.

JMP también se conoce como **salto incondicional**, porque siempre transfiere el control del programa a la etiqueta especificada. Un **salto condicional**, en cambio, transferirá el control a la etiqueta **si se cumple cierta condición**, esto es equivalente a una sentencia **if** de un lenguaje de alto nivel.

CMP Y SALTOS CONDICIONALES

Una de las estructuras de control más básicas es la construcción **if…else**. Por ejemplo, en Ruby:

```
if a == b
    Instrucciones a ejecutar si a == b
else
    Instrucciones a ejecutar si a != b
end
Instrucciones posteriores al bloque if…else
```

Para emular el comportamiento de esta construcción de alto nivel en Assembler debemos hacer uso de instrucciones de comparación y de saltos condicionales. Un ejemplo análogo al anterior en Assembler:

```
cmp ax, bx
jne else
    Instrucciones a ejecutar si ax == bx
jmp end
else:
    Instrucciones a ejecutar si ax != bx
end:
Instrucciones posteriores al bloque if…else
```

CMP se utiliza para comparar dos valores. A continuación de CMP debemos utilizar una instrucción de salto condicional que saltará o no, dependiendo del resultado de la comparación.

Por ahora, utilizaremos los siguientes saltos condicionales[17]:

- **JE**: Saltar si son iguales
- **JNE**: Saltar si no son iguales
- **JA**: Saltar si es mayor (todas las comparaciones son del el operando de la izquierda respecto del de la derecha)
- **JAE**: Saltar si es mayor o igual
- **JB**: Saltar si es menor
- **JBE**: Saltar si es menor o igual

En el ejemplo se utiliza la instrucción CMP para comparar AX y BX. A continuación se utiliza la instrucción de salto condicional JNE. JNE (*Jump if Not Equal*) realiza la bifurcación a la etiqueta indicada si la comparación anterior dio que los operandos son distintos.

Entonces, si la comparación da que son distintos se saltará a la etiqueta ELSE. Si son iguales, se sigue ejecutando normalmente. En este caso, la última instrucción antes del bloque ELSE debe ser un salto hacia la primera instrucción luego del ELSE, ya que hay que saltear este bloque. De esta forma, logramos un código con el mismo comportamiento que la construcción if…else de un lenguaje de alto nivel.

[17] Las letras en estos saltos significan en inglés: J = *Jump*, E = *Equal*, N = *Not*, A = *Above*, B = *Below*.

Notar que la condición de salto es la **opuesta** a la condición expresada en el código de alto nivel. Esto es así porque el if trabaja al revés, hace que se *saltee* el código siguiente si *no* se cumple la condición. Es medio un trabalenguas, pero pensándolo un poco se entiende.

Veamos el ejemplo, paso a paso con números concretos:

Supongamos que AX = 5 y BX = 2.

```
cmp ax, bx
jne else
```

Comparamos AX con BX y debemos saltar si no son iguales a la etiqueta ELSE. Como justamente no son iguales (5 != 2), la condición de salto se cumple. Entonces, saltamos y llegamos al siguiente código:

```
else:
    Instrucciones a ejecutar si ax != bx
```

Ejecutamos todas las instrucciones y llegamos a:

```
end:
Instrucciones posteriores al bloque if…else
```

Cuando el código llega a una etiqueta, simplemente la ignora[18]. Así que seguimos ejecutando las instrucciones que siguen.

Volvamos a comenzar, pero ahora supongamos que AX = BX = 4.

```
cmp ax, bx
jne else
```

Comparamos AX con BX y debemos saltar si no son iguales a la etiqueta ELSE. Como sí son iguales (4 == 4), la condición de salto no se cumple. Por lo tanto, no saltamos y seguimos ejecutando las siguientes instrucciones normalmente:

```
    Instrucciones a ejecutar si ax == bx
jmp end
```

Al terminar de ejecutar las instrucciones nos encontramos con un salto incondicional (JMP). Esto quiere decir que, sin evaluar ninguna condición, saltamos directamente a la etiqueta indicada (END):

```
end:
Instrucciones posteriores al bloque if…else
```

[18] De hecho, las etiquetas en las instrucciones de salto se reemplazan por el **valor de la dirección** dentro del segmento de código al ensamblar el programa, así que las etiquetas dentro del cuerpo del código realmente **no existen** en la versión en lenguaje de máquina.

ITERACIONES

ITERACIÓN FIJA

Assembler brinda la instrucción **LOOP** para realizar iteraciones del tipo **for**. El formato es:

```
mov cx, N
etiqueta:

    Instrucciones a repetir N veces

loop etiqueta
```

LOOP utiliza **CX** como contador. CX debe inicializarse en el número de iteraciones. Cada vez que se ejecuta la instrucción LOOP, CX se decrementa en 1 y se realiza la bifurcación a la etiqueta indicada sólo si CX es distinto de 0.

El siguiente ejemplo llena de ceros un arreglo de 50 elementos:

```
    mov cx, 50              ; Vamos a iterar 50 veces
    mov si, 0               ; Comenzar por el elemento 0

siguiente:

    mov arreglo[si], 0      ; Copiar un 0 al elemento actual del arreglo

    inc si                  ; Ir al próximo elemento

    loop siguiente          ; Iterar
```

ITERACIÓN CONDICIONAL

También podemos realizar iteraciones de tipo **while** (mientras se cumple una condición). Para ello debemos utilizar un esquema similar al siguiente:

```
    mov si, 0               ; Comenzar por el elemento 0

siguiente:

    cmp arreglo[si], 0FFh   ; Si el elemento actual es FFh
    je salir                ; Salir del bucle

    mov arreglo[si], 0      ; Copiar un 0 al elemento actual del arreglo

    inc si                  ; Ir al próximo elemento

    jmp siguiente           ; Iterar

salir:
```

El fragmento de código anterior llena de ceros un arreglo mientras que el elemento actual sea distinto de FFh. Siguiendo el código paso a paso con un arreglo de ejemplo, es fácil ver cómo funciona.

UN EJEMPLO COMPLETO CON CONTROL DE FLUJO

Se tiene un arreglo de elementos de tamaño byte. El último elemento del arreglo tiene el valor 00h (este valor no se repite en ningún otro elemento, sólo en el último). Realice un programa en Assembler que determine la longitud del arreglo, sin contar el último elemento, y la almacene en el registro DX.

```
.MODEL SMALL
.STACK
.DATA

    arreglo DB 15h, 07h, 22h, 9Ah, 4Dh, 00h   ; Datos de prueba

.CODE
inicio:

    mov ax, @data           ; Inicializar el segmento de datos
    mov ds, ax              ;

    mov dx, 0               ; DX arranca en 0 (lo incrementaremos en la
                            ;   iteración)
    mov si, 0               ; SI va a apuntando a cada elemento

iterar:

    cmp arreglo[si], 00h    ; Si llegamos al final (valor 00h)
    je fin                  ; salir del bucle

    inc dx                  ; Incrementar la longitud (al terminar el
                            ;   bucle DX tendrá la longitud total del
                            ;   arreglo)
    inc si                  ; Ir al próximo elemento

    jmp iterar              ; Iterar

fin:

    mov ax, 4C00h           ; Terminar
    int 21h                 ;

END inicio
```

CONCEPTOS MÁS AVANZADOS

NÚMEROS NEGATIVOS

En el procesador Intel 8086 los números enteros negativos se almacenan utilizando la representación de **complemento a 2**.

Nuevamente, a los efectos del almacenamiento en memoria, tanto números negativos (Ca2) como positivos (BSS) son simples secuencias de bits. Depende de las instrucciones que utilicemos para operar sobre esos datos que los mismos sean considerados números positivos, números negativos o cualquier otra cosa.

INICIALIZACIÓN DE VARIABLES

Para declarar una variable e inicializarla con un número negativo, se puede hacer simplemente escribiendo un número expresado en cualquiera de los sistemas habituales (binario, hexadecimal o decimal) precedido de un **signo menos**. O bien, en el caso de los números binarios y hexa, puede ponerse directamente un valor que represente un número negativo (MSB = 1):

```
numeros_negativos DB -2, -00011110b, -2Fh
otro_negativo     DB 11111010   ; Equivale a -6 en Ca2 y a 250 en BSS
```

El signo menos lo único que hace es obtener el complemento a 2 de la representación binaria del número indicado. Por ejemplo:

$$-2 \rightarrow -00000010b \rightarrow 11111110b$$

Esto no quiere decir que el número deba ser necesariamente interpretado como negativo. En el ejemplo anterior, el byte resultante se interpreta como –2 en Ca2, y como 254 en BSS.

De hecho, utilizar un signo menos delante del número **no garantiza que el valor resultante sea un número negativo válido en Ca2**. Supongamos la siguiente declaración:

```
numero DB -195
```

Obtendríamos el siguiente resultado:

$$-195 \rightarrow -11000011b \rightarrow 00111101b$$

El valor resultante es el número 61, tanto en BSS como en Ca2. Esto sucede porque *no existe* una representación del número –195 en binario Ca2 de 8 bits. Por ello hay que tener mucho cuidado al utilizar el signo menos para especificar un valor negativo.

> *Al poner un **signo menos** delante de una constante, indicamos que queremos **aplicar la regla de complementación a 2** sobre ese valor. El número resultante puede ser o no negativo, dependiendo del valor original (puesto que se puede producir un desborde).*

IDENTIFICACIÓN DE VALORES NEGATIVOS

Al trabajar con Assembler, a menudo necesitaremos interpretar valores expresados en binario o en hexadecimal. Si bien lleva su tiempo hacer la conversión para obtener el número en formato decimal, a veces no necesitamos saber el número en sí, sino si se trata de un número positivo o negativo, y esto puede lograrse fácilmente.

Recordemos que en Ca2, todo número negativo comienza por 1 y todo positivo comienza por 0. Entonces, para determinar si un número expresado en binario es positivo o negativo, basta **mirar el valor del MSB** (bit más significativo)[19].

Para los números hexadecimales, partimos de la misma premisa: el MSB debe ser 1 para los negativos. Cada dígito hexa representa 4 bits. ¿En qué dígitos hexa el bit de la izquierda es 1?

- 8h = 1000b
- 9h = 1001b
- Ah = 1010b
- …
- Fh = 1111b

Entonces, para saber si un número expresado en hexa es negativo, basta fijarse que comience por un dígito entre 8 y F.

> *Los valores expresados en **hexadecimal** que comienzan por un dígito entre **8** y **F**, son números **negativos en Ca2**.*

EL REGISTRO DE FLAGS

Ya hemos visto el funcionamiento de casi todos los registros de la CPU Intel 8086. Nos queda por comentar uno muy importante: el registro de **flags** (o banderas).

Este registro es alterado por las operaciones lógicas y aritméticas que realiza la ALU para indicar al procesador algunas condiciones especiales del resultado obtenido.

El registro de flags es un registro de 16 bits, de los cuales sólo 9 tienen una función definida. La organización del registro es la siguiente:

				O	D	I	T	S	Z		A		P		C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A continuación, describimos el funcionamiento de los bits más relevantes para nuestro estudio:

- **Z**: Bit de **cero** (*zero*, en inglés). Se activa (se pone en 1) si el resultado de una operación es 0.
- **S**: Bit de **signo**. Se activa si el resultado de la operación es negativo (es decir, si el MSB es 1).
- **C**: Bit de **carry**. Se activa si hay un acarreo del MSB en una operación aritmética en BSS.
- **O**: Bit de **desborde** (***o**verflow*, en inglés). Se active para indicar desborde en una operación aritmética en Ca2.

Ejemplo:

[19] Hay que tener cuidado de estar observando la representación **completa** del número. Por ejemplo, el número 1110b *no* es un número negativo. Ya que, si completamos el número para hacerlo de 8 bits, tenemos 00001110b, con lo cual vemos que le MSB es 0. (Lo mismo sucede si el número fuera de 16 bits).

```
mov ax, 5
mov bx, 7
sub ax, bx    ; El resultado es 11111110b
              ; Z = 0 (no es cero)
              ; S = 1 (MSB = 1 - negativo en Ca2)
              ; C = 1 (hubo carry – en realidad, borrow)
              ; O = 0 (no hubo desborde en Ca2)
```

Si bien describimos los flags en base a operaciones aritméticas, estos se utilizan en general para indicar distintos estados resultantes de las operaciones realizadas, ya sean aritméticas o no. Para saber cómo afecta cada instrucción al registro de flags, hay que consultar las especificaciones del fabricante[20].

MÁS OPERACIONES

OPERACIONES LÓGICAS

AND

```
AND Destino, Operando2
```

Realiza la **conjunción lógica** bit a bit y guarda el resultado en Destino.

```
mov al, 01001110b
mov ah, 00111001b
and al, ah
;  AL = 00001000
```

OR

```
OR Destino, Operando2
```

Realiza la **disyunción inclusiva** bit a bit y guarda el resultado en Destino.

```
mov al, 01001110b
mov ah, 00111001b
or al, ah
;  AL = 01111111b
```

XOR

```
XOR Destino, Operando2
```

Realiza la **disyunción exclusiva** bit a bit y guarda el resultado en Destino.

```
mov al, 01001110b
mov ah, 00111001b
xor al, ah
;  AL = 01110111b
```

[20] Nuevamente, puede visitar http://www.intel.com/design/Pentium4/documentation.htm para descargar las especificaciones completas de todas las instrucciones de la arquitectura Intel x86.

NEG

```
NEG Destino
```

Genera el **complemento a 2** del operando y lo almacena sobre sí mismo.

```
mov al, 01001110b
neg al
;  AL = 10110010b
```

NOT

```
NOT Destino
```

Realiza la **negación** del operando bit a bit. Es decir que **invierte cada bit**. (Es equivalente a generar el **complemento a 1**).

```
mov al, 01001110b
not al
;  AL = 10110001b
```

TEST

```
TEST Operando1, Operando2
```

Realiza la conjunción lógica bit a bit (como AND), pero a diferencia de AND, **no guarda el resultado**. Simplemente **tiene efectos sobre el registro de flags** (por ejemplo, Z = 1 si todos los bits del resultado son ceros).

```
mov al, 01001110b
mov ah, 00111001b
test al, ah
;  AL no se modifica
;  El registro de flags es afectado (Z = 0, S = 0)
```

SALTOS CONDICIONALES CON SIGNO

Hasta ahora, utilizamos los siguientes saltos condicionales para comparar dos operandos según su relación de orden:

- **JA**: Saltar si es mayor
- **JAE**: Saltar si es mayor o igual
- **JB**: Saltar si es menor
- **JBE**: Saltar si es menor o igual

También son válidas las instrucciones que utilizan las condiciones negadas: **JNA**, **JNAE**, **JNB** y **JNBE**.

Estas instrucciones consideran a los operandos como valores expresados en BSS (recordemos que la misma cadena de bits puede considerarse en BSS o Ca2).

Si queremos considerar los números en Ca2, debemos utilizar las siguientes[21]:

[21] Las nuevas letras significan: G = *Greater*, L = *Less*.

- **JG**: Saltar si es mayor (Ca2)
- **JGE**: Saltar si es mayor o igual (Ca2)
- **JL**: Saltar si es menor (Ca2)
- **JLE**: Saltar si es menor o igual (Ca2)

También son válidas las instrucciones que utilizan las condiciones negadas: **JNG**, **JNGE**, **JNL** y **JNLE**.

Los siguientes saltos condicionales son válidos tanto para BSS como para Ca2:

- **JE**: Saltar si son iguales
- **JNE**: Saltar si no son iguales

MÁS OPERACIONES ARITMÉTICAS

MUL / IMUL

Existen dos instrucciones de multiplicación: **MUL**, para valores sin signo, e **IMUL**, para valores con signo. MUL e IMUL permiten la multiplicación de dos valores de 8 bits o de dos valores de 16 bits.

MUL e IMUL sólo utilizan *un operando*. Para multiplicar dos bytes, ponemos uno de los factores en el registro AL. Luego usamos MUL o IMUL usando el otro factor como operando. Para el operando de MUL se puede usar cualquier modo de direccionamiento, **excepto modo inmediato**. El resultado de la multiplicación se devuelve en AX. (Notar que el valor más grande que se obtiene al multiplicar dos bytes es 255 * 255 = 65025, lo que no entra en un byte). Por ejemplo:

```
mov al, 5
mov bl, 2
mul bl
; En este punto AX = 10
```

Para multiplicar dos words, se pone una en AX y se usa MUL o IMUL con el otro factor. El resultado de multiplicar dos números de 16 bits puede necesitar hasta 32 bits. Como el procesador sólo maneja números de 16 bits, divide el resultado en dos: la palabra alta y la baja. *La palabra baja se almacena en AX y la alta en DX*.

```
mov ax, -2000
mov bx, 30000
imul bx
; En este punto el resultado está DXAX
; AX = 0111100100000000
; DX = 1111110001101100
```

DIV / IDIV

La división trabaja en forma similar. Tenemos **DIV** e **IDIV**, para números sin y con signo respectivamente. Estas instrucciones permiten dividir un número de 16 bits por uno de 8, o un número de 32 bits por uno de 16.

Para dividir un número de 16 bit por un número de 8, se pone el dividendo en AX y se usa DIV o IDIV pasando como parámetro el divisor de 8 bits. El cociente se almacena en AL. El resto de la división se almacena en AH.

```
mov ax, 120
mov bl, 2
div bl
; AL = cociente, AH = resto.
```

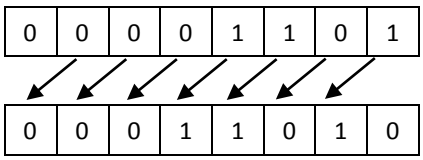
Para dividir un número de 32 bit por uno de 16, se pone el dividendo separado en dos palabras: la más baja en AX y la más alta en DX. Luego se usa DIV o IDIV con el divisor de 16 bits. El cociente se almacena en AX y el resto en DX.

```
mov dx, 0001h
mov ax, 0ABC0h
mov cx, 0015h
idiv cx
; AX = cociente, DX = resto.
```

MÁS OPERACIONES LÓGICAS

CORRIMIENTOS

Realizar un corrimiento (*shift*) sobre un número binario significa trasladar cada bit del número una cantidad fija de posiciones hacia la derecha o hacia la izquierda (todos hacia el mismo lado). Por ejemplo, si aplicamos un corrimiento de 1 hacia la izquierda al número 00001101, obtenemos el número 00011010.



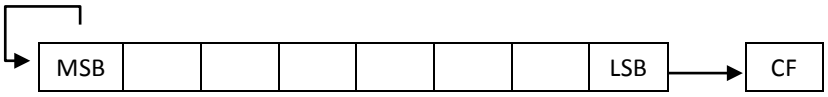
Este resultado en Assembler se consigue usando las instrucciones **SHL** (*SHift Left*) y **SHR** (*SHift Right*). Ambas instrucciones reciben como primer parámetro la ubicación del número y como segundo parámetro la cantidad de lugares a correr. Los números en el extremo izquierdo (en el caso de SHL) se descartan, excepto el último que queda en el carry. Por el extremo opuesto se introducen ceros:

SHR trabaja exactamente en el sentido opuesto. Se introducen ceros por la parte izquierda y los números que salen por la derecha van al flag de carry.

Esta operación se puede aplicar tanto sobre operandos de 8 como de 16 bits. El segundo operando debe ser o bien el número 1 o bien el registro CL. SHL y SHR no permite especificar la cantidad de corrimiento mediante direccionamiento inmediato para otro número distinto de 1.

```
shr ax, 1
mov cl, 2
shl bh, cl
```

Debemos recordar que este tipo de corrimientos sirven para multiplicar (SHL) o dividir (SHR) un número por una potencia de dos. Pero usar corrimientos a derecha para dividir números negativos no funciona, ya que el bit de signo es quitado de su posición. Para remediar este inconveniente, Assembler provee otra instrucción que es **SAR** (corrimiento aritmético a derecha). SAR opera así:



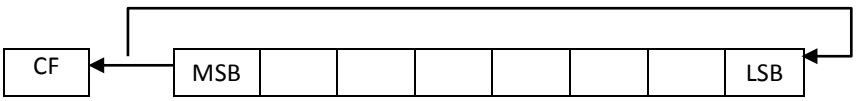
Es decir, en lugar de insertar ceros, inserta una copia del MSB, con lo que el signo se conserva.

SAL, la operación de corrimiento aritmético a izquierda, es equivalente a SHL. Sin embargo, se utiliza cuando se trabaja con números con signo, para hacer el código más legible.

ROTACIONES

Las rotaciones de bits son similares a los corrimientos, excepto que cuando un valor es rotado, los bits que salen por un extremo son copiados en el extremo opuesto.

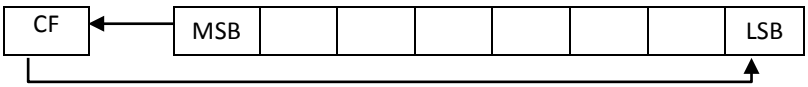
Veamos una rotación a izquierda. Estas se logran mediante la instrucción **ROL**. Si rotamos un byte hacia la izquierda un bit, movemos todos los bits hacia la izquierda un lugar. Luego, el último bit (el 7) se copia en el lugar 0:



Además, los bits que van saliendo también se copian en el flag de carry.

Las rotaciones a derecha se hacen mediante **ROR**. Funcionan exactamente al revés de las rotaciones a izquierda.

También existen otras rotaciones **RCL** y **RCR**. Estas toman al bit de carry como un bit más del número (a la izquierda del MSB si se usa RCL y a la derecha del LSB si se usa RCR). Así, la rotación hace que los números que se "caen" de un extremo pasen al carry y cuando un número se cae del carry vaya al LSB, para una rotación a izquierda (o al LSB, para una a derecha).



FUNCIONAMIENTO DE CMP Y DE LOS SALTOS CONDICIONALES

Considere este fragmento de código:

```
cmp ax, bx
jne final
```

Dado que CMP y JNE son dos instrucciones completamente aisladas, ejecutadas en su propio ciclo fetch–decode–execute ¿Cómo se conecta la instrucción CMP con el salto condicional para que éste pueda decidir si debe o no realizar el salto? La respuesta es: *a través del registro de flags*.

La comparación es en realidad una **resta**. Similar a SUB, CMP resta ambos operandos (op. Izquierdo – op. derecho) y, como cualquier operación aritmética, **afecta el registro de flags**. La diferencia es que **no guarda el resultado**.

Luego, analizando el registro de flags, el salto condicional determina si la condición se cumple o no. Por ejemplo, para determinar si los operandos eran iguales, JE puede mirar el flag de Z. Si Z = 1, significa que la resta entre los operandos dio 0, con lo cual los operandos debían ser iguales. (En caso contrario, si Z = 0, significa que la resta no dio 0, con lo cual los operandos eran distintos).

Existen otros dos saltos condicionales equivalentes a JE y JNE que dejan claro que utilizan el flag Z:

- **JZ**: Saltar si es 0 (Z = 1)
- **JNZ**: Saltar si no es 0 (Z = 0)

Es completamente válido utilizar cualquiera de las dos expresiones. De hecho, al ensamblar el código tanto JE como JZ resultan en la misma operación en lenguaje de máquina. (Lo mismo vale para JNE y JNZ).[22]

También existen saltos condicionales para preguntar por otros de los flags. Por ejemplo:

- Carry: **JC** / **JNC**
- Signo: **JS** / **JNS**
- Overflow: **JO** / **JNO**

SALTOS CONDICIONALES SIN CMP

Dado que los saltos condicionales simplemente utilizan el estado del registro de flags para determinar si deben o no realizar el salto, en realidad no es necesario utilizar CMP antes de un salto condicional. Cualquier operación que afecte los flags también es válida.

Por ejemplo:

```
test al, 00001111b   ; Realiza un "and" entre AL y 00001111
                     ; Resultado --> 0000xxxx (los 4 bits más bajos
                     ; dependen del contenido de AL)

jz abajo             ; Salta si es cero (Z = 1)
                     ; Considerando el resultado (0000xxxx), saltará si
                     ; los cuatro bits más bajos de AL son 0
```

Otro ejemplo (muy común cuando queremos hacer una iteración pero no podemos usar CX para el LOOP):

```
        mov dx, 10   ; DX arranca en 10 (para iterar 10 veces)
        mov si, 0

iterar: mov arreglo[si], 0
        inc si

        dec dx
        jnz iterar   ; Saltar si DX no llegó a 0
```

[22] De hecho, hay muchos saltos condicionales que son equivalentes además de estos dos. Por ejemplo, JBE y JNA son iguales.