**Lenguaje ensamblador**

La importancia del lenguaje de ensamblador radica en el hecho de que es el lenguaje simbólico que trabaja más cerca del procesador. Prácticamente todas las instrucciones de ensamblador tienen una correspondencia directa con las instrucciones binarias del código máquina que utiliza directamente el procesador. Esto lleva a que el lenguaje sea relativamente sencillo, pero que tenga un gran número de excepciones y reglas definidas por la misma arquitectura del procesador, y a la hora de programar, además de conocer las especificaciones del lenguaje, hay que conocer también las especificaciones de la arquitectura. Así pues, este lenguaje permite escribir programas que pueden aprovechar todas las características de la máquina, lo que facilita la comprensión de la estructura interna del procesador.

**Modos de operación**

Los procesadores con arquitectura x86-64 mantienen la compatibilidad con los procesadores de la arquitectura IA-32 (x86-32). Por este motivo, disponen de los mismos modos de operación de la arquitectura IA-32, lo que permite mantener la compatibilidad y ejecutar aplicaciones de 16 y 32 bits, pero además añaden un modo nuevo denominado *modo extendido* (o modo *IA-32e*, en el caso de Intel), dentro del cual se puede trabajar en modo real de 64 bits.

|  |
| --- |
| *Los procesadores actuales soportan diferentes modos de operación, pero como mínimo disponen de un modo protegido y de un modo supervisor. El modo de supervisor es utilizado por el núcleo del sistema para las tareas de bajo nivel que necesitan un acceso sin restricciones al hardware, como puede ser el control de la memoria o la comunicación con otros dispositivos. El modo protegido, en cambio, se utiliza para casi todo el resto de las tareas.*  *Cuando ejecutamos programas en modo protegido, solo podremos utilizar el hardware haciendo llamadas al sistema operativo, que es el que lo puede controlar en modo supervisor.*  *Cuando un equipo se inicia por primera vez se ejecutan los programas de la BIOS, del gestor de arranque y del sistema operativo que tienen acceso ilimitado al hardware; cuando el equipo se ha iniciado, el sistema operativo puede pasar el control a otro programa y poner el procesador en modo protegido. En modo protegido, los programas tienen acceso a un conjunto más limitado de instrucciones y solo podrán dejar el modo protegido haciendo una petición de interrupción que devuelve el control al sistema operativo; de esta manera se garantiza el control para acceder al hardware.* |

El modo extendido de 64 bits es utilizado por los sistemas operativos de 64 bits. Dentro de este modo general, se dispone de un modo de operación de 64 bits y de un modo de compatibilidad con los modos de operación de las arquitecturas de 16 y 32 bits. En un sistema operativo de 64 bits, los programas de 64 bits se ejecutan en modo de 64 bits y las aplicaciones de 16 y 32 bits se ejecutan en modo de compatibilidad. Los programas de 16 y 32 bits que se tengan que ejecutar en modo real o virtual x86 no se podrán ejecutar en modo extendido si no son emulados.

###### **Modo de compatibilidad**

El modo de compatibilidad permite a un sistema operativo de 64 bits ejecutar directamente aplicaciones de 16 y 32 bits sin recompilarlas. En este modo, las aplicaciones pueden utilizar direcciones de 16 y 32 bits, y pueden acceder a un espacio de memoria de 4 Gbytes. El tamaño de los operandos puede ser de 16 y 32 bits.

El modo heredado de 16 y 32 bits es utilizado por los sistemas operativos de 16 y 32 bits. Cuando el sistema operativo utiliza los modos de 16 bits o de 32 bits, el procesador actúa como un procesador x86 y solo se puede ejecutar código de 16 o 32 bits. Este modo solo permite utilizar direcciones de 32 bits, de manera que limita el espacio de direcciones virtual a 4 GB.

###### **Modo de 64 bits**

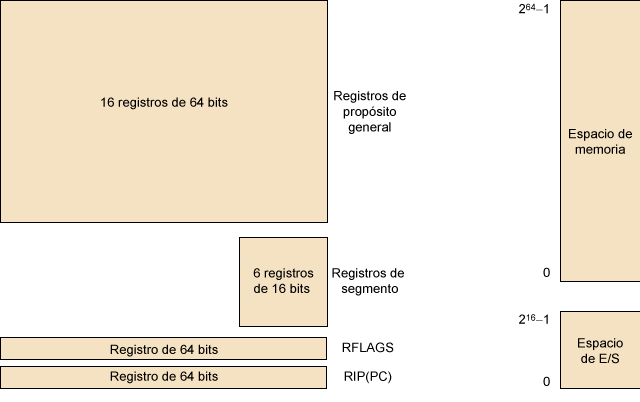
El modo de 64 bits proporciona acceso a 16 registros de propósito general de 64 bits. En este modo se utilizan direcciones virtuales (o lineales) que por defecto son de 64 bits y se puede acceder a un espacio de memoria lineal de 264 bytes. El tamaño por defecto de los operandos se mantiene en 32 bits para la mayoría de las instrucciones.

Los elementos que desde el punto de vista del programador son visibles en este modo de operación son:

**1) Espacio de memoria**: un programa en ejecución en este modo puede acceder a un espacio de direcciones lineal de 264 bytes. El espacio físico que realmente puede dirigir el procesador es inferior y depende de la implementación concreta de la arquitectura.

**2) Registros**: hay 16 registros de propósito general de 64 bits, que soportan operaciones de byte (8 bits), word (16 bits), double word (32 bits) y quad word (64 bits).

* El registro contador de programa (RIP, *instruction pointer register*) es de 64 bits.
* El registro de bits de estado también es de 64 bits (RFLAGS). Los 32 bits de la parte alta están reservados; los 32 bits de la parte baja son accesibles y corresponden a los mismos bits de la arquitectura IA-32 (registro EFLAGS).
* Los registros de segmento en general no se utilizan en el modo de 64 bits.



##### 

##### **Registros**

Los procesadores de la arquitectura x86-64 disponen de un banco de registros formado por registros de propósito general y registros de propósito específico. Registros de propósito general hay 16 de 64 bits y de propósito específico hay 6 registros de segmento de 16 bits, también hay un registro de estado de 64 bits (RFLAGS) y un registro contador de programa también de 64 bits (RIP).

###### **Registros de propósito general**

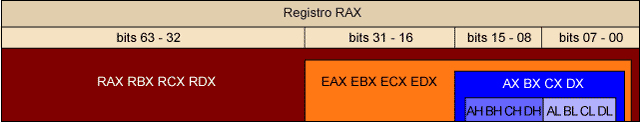
Son 16 registros de datos de 64 bits (8 bytes): RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP y R8-R15.

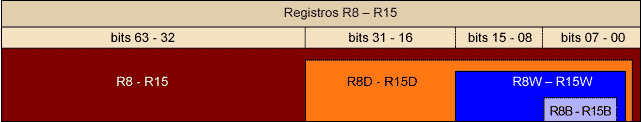
Los 8 primeros registros se denominan de manera parecida a los 8 registros de propósito general de 32 bits disponibles en la arquitectura IA-32 (EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP).

Los registros son accesibles de cuatro maneras diferentes:

**1)** Como registros completos de 64 bits (quad word).  
**2)** Como registros de 32 bits (double word), accediendo a los 32 bits de menos peso.  
**3)** Como registros de 16 bits (word), accediendo a los 16 bits de menos peso.  
**4)** Como registros de 8 bits (byte), permitiendo acceder individualmente a uno o dos de los bytes de menos peso según el registro.

Los registros de propósito general se dividen en partes (y se da un nombre a cada parte), lo que facilita trabajar con diferentes tipos de datos.







Hay algunas limitaciones en el uso de los registros de propósito general:

* En una misma instrucción no se puede usar un registro del conjunto AH, BH, CH, DH junto con uno del conjunto SIL, DIL, BPL, SPL, R8B – R15B.
* Registro RSP: tiene una función especial, funciona como apuntador de pila, contiene siempre la dirección del primer elemento de la pila. Si lo utilizamos para otra cosa perderemos el acceso a la pila.
* Cuando se utiliza un registro de 32 bits como operando destino de una instrucción, la parte alta del registro está fijada en 0.

###### **Registros de propósito específico**

Podemos distinguir varios registros de propósito específico:

**1) Registros de segmento**: hay 6 registros de segmento de 16 bits.

* CS: *code segment*
* DS: *data segment*
* SS: *stack segment*
* ES: *extra segment*
* FS: *extra segment*
* GS: *extra segment*

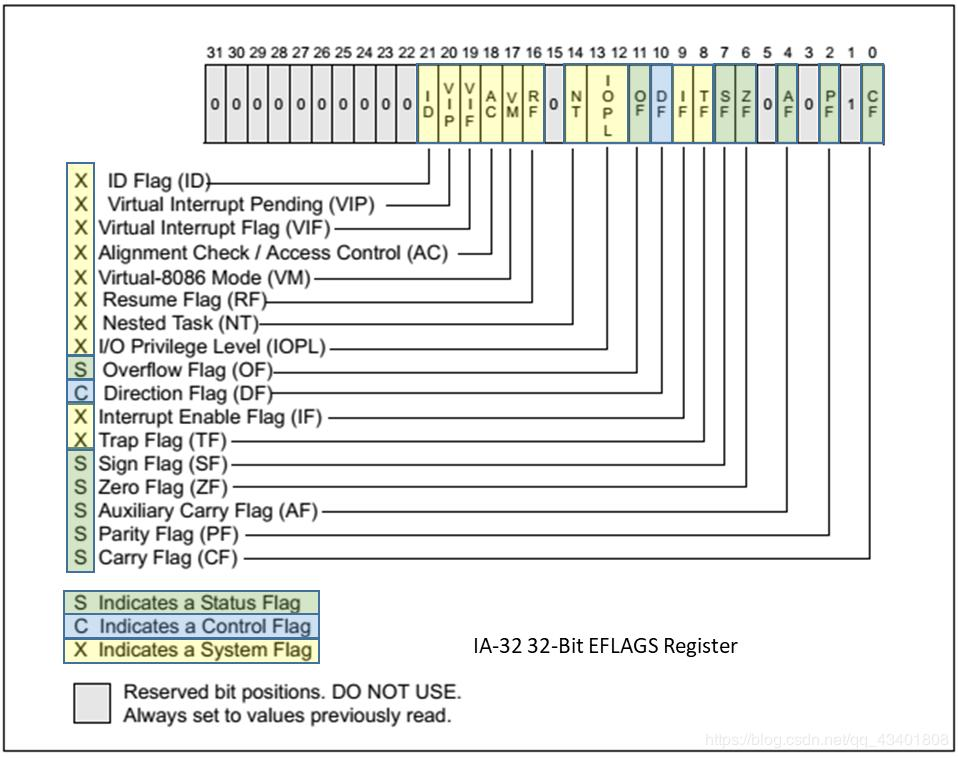
Estos registros se utilizan básicamente en los modelos de memoria segmentados (heredados de la arquitectura IA-32). En estos modelos, la memoria se divide en segmentos, de manera que en un momento dado el procesador solo es capaz de acceder a seis segmentos de la memoria utilizando cada uno de los seis registros de segmento. En el modo de 64 de bits, estos registros prácticamente no se utilizan, ya que se trabaja con el modelo de memoria lineal y el valor de estos registros se encuentra fijado en 0 (excepto en los registros FS y GS, que pueden ser utilizados como registros base en el cálculo de direcciones).

**2) Registro de instrucción o *instruction pointer* (RIP)**: es un registro de 64 bits que actúa como registro contador de programa (PC) y contiene la dirección efectiva (o dirección lineal) de la instrucción siguiente que se ha de ejecutar. Cada vez que se lee una instrucción nueva de la memoria, se actualiza con la dirección de la instrucción siguiente que se tiene que ejecutar; también se puede modificar el contenido del registro durante la ejecución de una instrucción de ruptura de secuencia (llamada a subrutina, salto condicional o incondicional).

**3) Registro de estado o *Flags register* (RFLAGS)**: es un registro de 64 bits que contiene información sobre el estado del procesador e información sobre el resultado de la ejecución de las instrucciones. Solo se utiliza la parte baja del registro (bits de 31 a 0), que corresponde al registro EFLAGS de la arquitectura IA-32; la parte alta del registro está reservada y no se utiliza.

El uso habitual del registro de estado consiste en consultar el valor individual de sus bits; eso se puede conseguir con instrucciones específicas, como, por ejemplo, las instrucciones de salto condicional que consultan uno o más bits para determinar si saltan o no según cómo ha dejado estos bits la última instrucción que los ha modificado (no ha de ser la última instrucción que se ha ejecutado).

En la siguiente imagen se describe el significado de los bits de este registro.



### **El lenguaje de ensamblador para la arquitectura x86-64**

Un programa ensamblador escrito con sintaxis NASM está formado por tres secciones o segmentos: *.data* para los datos inicializados, *.bss* para los datos no inicializados *y .text* para el código. Para definir una sección se pueden utilizar indistintamente las directivas *section* y *segment*.

La sección .*bss* no es necesaria si se inicializan todas las variables declaradas en la *sección .data*.

La sección *.text* permite también definir variables y, por lo tanto, permitiría prescindir también de la sección *.data*, aunque no es recomendable. Por lo tanto, *.text* es la única sección realmente obligatoria en todo programa.

La sección *.text* debe empezar siempre con la directiva *global*, que indica al GCC cuál es el punto de inicio del código. Cuando se utiliza GCC para generar un ejecutable, el nombre de la etiqueta donde se inicia la ejecución del código se ha de denominar obligatoriamente *main*; también ha de incluir una llamada al sistema operativo para finalizar la ejecución del programa y devolver el control al terminal desde el que hemos llamado al programa.

#### **Directivas**

Las directivas son pseudooperaciones que solo son reconocidas por el ensamblador. No se deben confundir con las instrucciones. Su función principal es declarar ciertos elementos de nuestro programa para que puedan ser identificados más fácilmente por el programador y también para facilitar la tarea de ensamblaje: definición de constantes, definición de variables y definición de otros elementos.

##### Definición de constantes

Una constante es un valor que no puede ser modificado por ninguna instrucción del código del programa. Realmente una constante es un nombre que se da para referirse a un valor determinado. La declaración de constantes se puede hacer en cualquier parte del programa: al principio del programa fuera de las secciones *.data, .bss*, .*text* o dentro de cualquiera de las secciones anteriores.

Las constantes son útiles para facilitar la lectura y posibles modificaciones del código. Si, por ejemplo, utilizamos un valor en muchos lugares de un programa, como podría ser el tamaño de un vector, y queremos probar el programa con un valor diferente, tendremos que cambiar este valor en todos los lugares donde lo hemos utilizado, con la posibilidad de que dejemos uno sin modificar y, por lo tanto, de que alguna cosa no funcione; en cambio, si definimos una constante con este valor y en el código utilizamos el nombre de la constante, modificando el valor asignado a la constante, se modificará todo.

|  |
| --- |
| Para definir constantes se utiliza la directiva equ, de la manera siguiente:  nombre\_constante equ valor |

##### Definición de variables

La declaración de variables en un programa en ensamblador se puede incluir en la sección *.data* para variables inicializadas, o en la sección *.bss* para variables sin inicializar.

Las variables de la sección .data se definen utilizando las siguientes directivas:

* db: define una variable de tipo byte, 8 bits.
* dw: define una variable de tipo palabra (word), 2 bytes = 16 bits.
* dd: define una variable de tipo doble palabra (double word), 2 palabras = 4 bytes = 32 bits.
* dq: define una variable de tipo cuádruple palabra (quad word), 4 palabras = 8 bytes = 64 bits.

El formato utilizado para definir una variable empleando cualquiera de las directivas anteriores es el mismo:

|  |
| --- |
| nombre\_variable directiva valor\_inicial |

Se puede utilizar una constante para inicializar variables. Solo es necesario que la constante se haya definido antes de su primera utilización.

Los valores iniciales de las variables y constantes se pueden expresar en bases diferentes como decimal, hexadecimal, octal y binario, o también como caracteres y cadenas de caracteres.

Los valores numéricos se consideran por defecto en decimal, pero también se puede indicar explícitamente que se trata de un **valor decimal** finalizando el número con el carácter d.

Los **valores hexadecimales** han de empezar por 0x, 0h o $, o deben finalizar con una h. Si se especifica el valor con $ al principio o con una h al final, el número no puede empezar por una letra; si el primer dígito hexadecimal ha de ser una letra (A, B, C, D, E, F), es necesario añadir un 0 delante.

Los **valores octales** han de empezar por 0o o 0q, o deben finalizar con el carácter o o q.

Los **valores binarios** han de empezar por 0b o finalizar con el carácter b.

Los **caracteres** y las **cadenas** de caracteres han de escribirse entre comillas simples (''), dobles ("") o comillas abiertas backquotes (` `). Las cadenas de caracteres pueden incluir caracteres no imprimibles y caracteres especiales; estos caracteres se pueden incluir con su codificación (ASCII), también pueden utilizar el código ASCII para cualquier otro carácter aunque sea imprimible.

|  |
| --- |
| *cadena db 'Hola',10 ;añade el carácter ASCII de salto de línea*  *cadena db 'Hola',9 ;añade el carácter ASCII de tabulación*  *cadena db 72,111,108,97 ;añade igualmente la cadena 'Hola'* |

Si la cadena se define entre comillas abiertas (` `), también se admiten algunas secuencias de escape iniciadas con \:

*\n: salto de línea  
\t: tabulador  
\e: el carácter ESC (ASCII 27)*

Los vectores en ensamblador se definen con un nombre de variable e indicando a continuación los valores que forman el vector.

|  |
| --- |
| *vector1 db 23, 42, -1, 65, 14 ;vector formado por 5 valores de tipo byte*  *vector2 db 'a'b', 'c', 'de ;vector formado por 4 bytes, inicializado usando caracteres*  *vector3 db 97, 98, 99, 100 ;es equivalente al vector anterior*  *vector4 dw 1000, 2000, -1000, 3000 ;vector formado por 4 palabras* |

*Dentro de la sección .bss esta sección se declaran y se reserva espacio para las variables de nuestro programa para las cuales no queremos dar un valor inicial.*

*Hay que utilizar las directivas siguientes para declarar variables no inicializadas:*

* *resb: reserva espacio en unidades de byte*
* *resw: reserva espacio en unidades de palabra, 2 bytes*
* *resd: reserva espacio en unidades de doble palabra, 4 bytes*
* *resq: reserva espacio en unidades de cuádruple palabra, 8 bytes*

*El formato utilizado para definir una variable empleando cualquiera de las directivas anteriores es el mismo:*

|  |
| --- |
| *nombre\_variable directiva multiplicidad* |

*La multiplicidad es el número de veces que reservamos el espacio definido por el tipo de dato que determina la directiva.*

|  |
| --- |
| *var1 resb 1 ;reserva 1 byte*  *var2 resb 4 ;reserva 4 bytes*  *var3 resw 2 ;reserva 2 palabras = 4 bytes, equivalente al caso anterior*  *var3 resd 1 ;reserva una cuádruple palabra = 4 bytes* |

##### Definición de otros elementos

**1) extern**. Declara un símbolo como externo. Lo utilizamos si queremos acceder a un símbolo que no se encuentra definido en el fichero que estamos ensamblando, sino en otro fichero de código fuente, en el que tendrá que estar definido y declarado con la directiva global. En el proceso de ensamblaje, cualquier símbolo declarado como externo no generará ningún error; es durante el proceso de enlazamiento cuando, si no hay un fichero de código objeto en el que este símbolo esté definido, producirá error. En una misma directiva *extern* se pueden declarar tantos símbolos como se quiera, separados por comas.

**2) global**. Es la directiva complementaria de *extern*. Permite hacer visible un símbolo definido en un fichero de código fuente en otros ficheros de código fuente; de esta manera, nos podremos referir a este símbolo en otros ficheros utilizando la directiva *extern*. El símbolo ha de estar definido en el mismo fichero de código fuente donde se encuentre la directiva global. En una misma directiva *global* se pueden declarar tantos símbolos como se quiera separados por comas.

Hay un uso especial de *global*: declarar una etiqueta que se debe denominar *main* para que el compilador de C (GCC) pueda determinar el punto de inicio de la ejecución del programa.

**3) section**. Define una sección dentro del fichero de código fuente. Cada sección hará referencia a un segmento de memoria diferente dentro del espacio de memoria asignado al programa. Hay tres tipos de secciones:

**a)** .data: sección en la que se definen datos inicializados, datos a los que damos un valor inicial.

**b)** .bss: sección en la que se definen datos sin inicializar.

**c)** .text: sección en la que se incluyen las instrucciones del programa.

La utilización de estas directivas no es imprescindible, pero sí recomendable con el fin de definir los diferentes tipos de información que utilizaremos.

**4) cpu**. Esta directiva indica que solo se podrán utilizar los elementos compatibles con una arquitectura concreta. Solo podremos utilizar las instrucciones definidas en esta arquitectura. Esta directiva se suele escribir al principio del fichero de código fuente, antes del inicio de cualquier sección del programa.

#### 

#### **Instrucciones**

El otro elemento que forma parte de cualquier programa escrito en lenguaje de ensamblador son las instrucciones. Las instrucciones en ensamblador tienen el formato general siguiente:

[etiqueta:] instrucción [destino[, fuente]] [;comentario]

Destino y fuente representan los operandos de la instrucción. Al final de la instrucción podemos añadir un comentario precedido del símbolo “;”. Los elementos entre [] son opcionales; por lo tanto, el único elemento imprescindible es el nombre de la instrucción.

##### Etiquetas

Una etiqueta hace referencia a un elemento dentro del programa ensamblador. Su función es facilitar la tarea de hacer referencia a diferentes elementos del programa. Las etiquetas sirven para definir constantes, variables o posiciones del código y las utilizamos como operandos en las instrucciones o directivas del programa.

Para definir una etiqueta podemos utilizar números, letras y símbolos *\_, $, #, @, ~, .* y *?*. Las etiquetas han de comenzar con un carácter alfabético o con los símbolos *\_, .*, o ?, pero las etiquetas que empiezan con estos tres símbolos tienen un significado especial dentro de la sintaxis NASM y por este motivo se recomienda no utilizarlos al inicio de la etiqueta. En sintaxis NASM se distingue entre minúsculas y mayúsculas.

|  |
| --- |
| Servicio equ 80h  section .data  msg db "Hola!",10  section .text  printHola:  mov rax,4  mov rbx,1  mov rcx,msg  mov rdx,6  int Servicio  jmp printHola |

En este fragmento de código hay definidas tres etiquetas: *Servicio* define una constante, *msg* define una variable y *printHola* define una posición de código.

Las etiquetas para marcar posiciones de código se utilizan en las instrucciones de salto para indicar el lugar donde se debe saltar y también para definir la posición de inicio de una subrutina dentro del código fuente.

Habitualmente, cuando se define una etiqueta para marcar una posición de código, se define con una secuencia de caracteres acabada con el carácter : (dos puntos). Pero cuando utilizamos esta etiqueta como operando, no escribiremos los dos puntos.

##### Operandos

En las instrucciones con un solo operando, este se puede comportar solo como operando fuente, solo como operando destino o como operando fuente y destino.

En las instrucciones con dos operandos, el primer operando se puede comportar como operando fuente y/o destino, mientras que el segundo operando se comporta siempre como operando fuente.

Los operandos de las instrucciones se pueden encontrar en tres lugares diferentes: en la instrucción (valores inmediatos), en registros o en memoria.

**1) Inmediatos**. En las instrucciones de dos operandos, se puede utilizar un valor inmediato como operando fuente; algunas instrucciones de un operando también admiten un valor inmediato como operando. Los valores inmediatos se pueden expresar como valores numéricos (decimal, hexadecimal, octal o binario) o como caracteres o cadenas de caracteres. También se pueden utilizar las constantes definidas en el programa como valores inmediatos. Para especificar un valor inmediato en una instrucción se utiliza la misma notación que la especificada en la definición de variables inicializadas.

**2) Registros**. Los registros se pueden utilizar como operando fuente y como operando destino. Podemos utilizar registros de 64 bits, 32 bits, 16 bits y 8 bits. Algunas instrucciones pueden utilizar registros de manera implícita.

**3) Memoria**. Las variables declaradas a memoria se pueden utilizar como operandos fuente y destino. En el caso **de instrucciones con dos operandos, solo uno de los operandos puede acceder a la memoria**, el otro ha de ser un registro o un valor inmediato (y este será el operando fuente). Aunque cuando especificamos un nombre de una variable sin los corchetes *[ ]*, no estamos haciendo un acceso a memoria, sino que la dirección de la variable está codificada en la propia instrucción y se considera un valor inmediato; por lo tanto puedes poner la dirección de memoria en número (como valor inmediato) y no contaría como acceso a memoria.

Debemos tener en cuenta el tamaño de los operandos que estamos utilizando en cada momento, sobre todo cuando hacemos referencias a memoria, y especialmente cuando esta es el operando destino, ya que utilizaremos la parte de la memoria necesaria para almacenar el operando fuente según el tamaño que tenga.

En algunos casos es obligatorio especificar el tamaño del operando; esto se lleva a cabo utilizando los modificadores BYTE, WORD, DWORD y QWORD ante la referencia a memoria. La función del modificador es utilizar tantos bytes como indica a partir de la dirección de memoria especificada, independientemente del tamaño del tipo de dato (db, dw, dd, dq) que hayamos utilizado a la hora de definir la variable.

En los casos en los que no es obligatorio especificar el modificador, se puede utilizar para facilitar la lectura del código o modificar un acceso a memoria.

Los casos en los que se debe especificar obligatoriamente el tamaño de los operandos son los siguientes:

**1)** Instrucciones de dos operandos en los que el primer operando es una posición de memoria y el segundo operando es uno inmediato; es necesario indicar el tamaño del operando.

**2)** Instrucciones de un operando en las que el operando sea una posición de memoria.

Cuando un operando se encuentra en la memoria, es necesario considerar qué modo de direccionamiento utilizamos para expresar un operando en una instrucción para definir la manera de acceder a un dato concreto. A continuación, veremos los modos de direccionamiento que podemos utilizar en un programa ensamblador:

**1) Inmediato**. En este caso, el operando hace referencia a un dato que se encuentra en la instrucción misma. No hay que hacer ningún acceso extra a memoria para obtenerlo. Solo podemos utilizar un direccionamiento inmediato como operando fuente. El número especificado ha de ser un valor que se pueda expresar con 32 bits como máximo, que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos y también sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable), con la excepción de la instrucción *mov* cuando el primer operando es un registro de 64 bits, para el que podremos especificar un valor que se podrá expresar con 64 bits.

**2) Directo a registro**. En este caso, el operando hace referencia a un dato que se encuentra almacenado en un registro. En este modo de direccionamiento podemos especificar cualquier registro de propósito general (registros de datos, registros índice y registros apuntadores).

**3) Directo a memoria**. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando habrá de especificar el nombre de una variable de memoria entre corchetes *[ ]*; cabe recordar que en sintaxis NASM se interpreta el nombre de una variable sin corchetes como la dirección de la variable y no como el contenido de la variable.

**4) Indirecto a registro**. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando habrá de especificar un registro entre corchetes *[ ]*; el registro contendrá la dirección de memoria a la cual queremos acceder.

|  |
| --- |
| mov rbx, var ;se carga en rbx la dirección de la variable var  mov rax, [rbx] ;el segundo operando utiliza la dirección que tenemos en rbx  ;para acceder a memoria, se mueven 8 bytes a partir de  ;la dirección especificada por rbx y se guardan en rax. |

**5) Indexado**. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. Digamos que un operando utiliza direccionamiento indexado si especifica una dirección de memoria como dirección base que puede ser expresada mediante un número o el nombre de una variable que tengamos definida, sumada a un registro que actúa como índice respecto a esta dirección de memoria entre corchetes *[ ]*.

|  |
| --- |
| mov rax, [vector+rsi] ;vector contiene la dirección base, rsi actúa como registro índice  add [1234h+r9],rax ;1234h es la dirección base, r9 actúa como registro índice. |

**6) Relativo**. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. Digamos que un operando utiliza direccionamiento relativo cuando especifica un registro sumado a un número entre corchetes *[ ]*. El registro contendrá una dirección de memoria que actuará como dirección base y el número como un desplazamiento respecto a esta dirección.

|  |
| --- |
| mov rbx, var ;se carga en rbx la dirección de la variable var  mov rax, [rbx+4] ;el segundo operando utiliza direccionamiento relativo  ;4 es el desplazamiento respecto a esta dirección  mov [rbx+16], rcx ;rbx contiene la dirección base, 16 es el desplazamiento  ;respecto a esta dirección. |

**Combinaciones del direccionamiento indexado y relativo**. La sintaxis NASM nos permite especificar de otras maneras un operando para acceder a memoria; el formato del operando que tendremos que expresar entre corchetes *[ ]* es el siguiente:

[Registro Base + Registro Index \* escala + desplazamiento]

El registro base y el registro índice pueden ser cualquier registro de propósito general, pero tienen que ser registros de 32 o 64 bits, la escala puede ser 1, 2, 4 u 8 y el desplazamiento ha de ser un número representable con 32 bits que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos; también podemos sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable). Podemos especificar solo los elementos que nos sean necesarios.

**7) Relativo a PC (Program Counter)**. En el modo de 64 bits se permite utilizar direccionamiento relativo a PC en cualquier instrucción; en otros modos de operación, el direccionamiento relativo a PC se reserva exclusivamente para las instrucciones de salto condicional.

Este modo de direccionamiento es equivalente a un direccionamiento relativo a registro base en el que el registro base es el registro contador de programa (PC). En la arquitectura x86-64 este registro se denomina *rip*, y el desplazamiento es el valor que sumaremos al contador de programa para determinar la dirección de memoria a la que queremos acceder.

Utilizaremos este modo de direccionamiento habitualmente en las instrucciones de salto condicional. En estas instrucciones especificaremos una etiqueta que representará el punto del código al que queremos saltar. La utilización del registro contador de programa es implícita, pero para emplear este modo de direccionamiento, se ha de codificar el desplazamiento respecto al contador de programa; el cálculo para obtener este desplazamiento a partir de la dirección representada por la etiqueta se resuelve durante el ensamblaje y es transparente para el programador.

|  |
| --- |
| je etiqueta1 ;si es igual salta a la etiqueta1 |

**8) Direccionamiento a pila**. Es un direccionamiento implícito; se trabaja implícitamente con la cima de la pila, mediante el registro apuntador a pila (*stack pointer*); en la arquitectura x86-64 este registro se llama *rsp*. En la pila solo podremos almacenar valores de 16 bits y de 64 bits.

Solo existen dos instrucciones específicas diseñadas para trabajar con la pila:

|  |
| --- |
| push fuente ;coloca un dato en la pila  pop destino ;extrae un dato de la pila |

Para expresar el operando fuente o destino podemos utilizar cualquier tipo de direccionamiento descrito anteriormente, pero lo más habitual es utilizar el direccionamiento a registro.

#### **Instrucciones más habituales**

* **mov *destino, fuente***: instrucción genérica para mover un dato desde un origen a un destino.
* **push *fuente***: instrucción que mueve el operando de la instrucción a la cima de la pila.
* **pop *destino***: mueve el dato que se encuentra en la cima de la pila al operando destino.
* **add *destino, fuente***: suma aritmética de los dos operandos.
* **sub *destino, fuente***: resta aritmética de los dos operandos.
* **inc *destino***: incrementa el operando en una unidad.
* **dec *destino***: decrementa el operando en una unidad.
* **mul *fuente***: multiplicación entera sin signo.
* **imul *fuente***: multiplicación entera con signo.
* **div *fuente***: división entera sin signo.
* **idiv *fuente***: división entera con signo.
* **cmp *destino, fuente***: comparación de los dos operandos; hace una resta sin guardar el resultado.
* **jmp etiqueta**: salta de manera incondicional a la etiqueta.
* **je *etiqueta*** / **jz *etiqueta***: salta a la etiqueta si igual, si el bit de cero es activo.
* **jne *etiqueta*** / **jnz *etiqueta***: salta a la etiqueta si diferente, si el bit de cero no es activo.
* **jb *etiqueta*** / **jnae *etiqueta***: salta a la etiqueta si es más pequeño sin considerar el signo.
* **jbe *etiqueta*** / **jna *etiqueta***: salta a la etiqueta si es más pequeño o igual sin considerar el signo.
* **ja *etiqueta*** / **jnbe *etiqueta***: salta a la etiqueta si es mayor sin considerar el signo.
* **jae *etiqueta*** / **jnb *etiqueta***: salta a la etiqueta si es mayor o igual sin considerar el signo.
* **jl *etiqueta*** / **jnge *etiqueta***: salta si es más pequeño.
* **jle *etiqueta*** / **jng *etiqueta***: salta si es más pequeño o igual.
* **jg *etiqueta*** / **jnle *etiqueta***: salta si es mayor.
* **jge *etiqueta*** / **jnl *etiqueta***: salta si es mayor o igual.
* **call *etiqueta***: llamada a subrutina.
* **ret**: retorno de subrutina.

#### **Gestión de la pila**

La pila es una zona de memoria que se utiliza para almacenar información de manera temporal. La pila también se utiliza habitualmente para pasar parámetros a las subrutinas y para guardar los registros que son modificados dentro de una subrutina, de manera que se pueda restaurar el valor antes de finalizar la ejecución. Se trata de una estructura de datos de tipo LIFO (*last in first out*): el último elemento introducido, que habitualmente se denomina *cima de la pila*, es el primer elemento que se saca y es el único directamente accesible.

En los procesadores x86-64, la pila se implementa en memoria principal a partir de una dirección base. Se utiliza el registro RSP como apuntador a la cima de la pila. La pila crece hacia direcciones más pequeñas; es decir, cada vez que se introduce un valor en la pila, este ocupa una dirección de memoria más pequeña, por lo tanto, el registro RSP se decrementa para apuntar al nuevo valor introducido y se incrementa cuando lo sacamos.

Los elementos se introducen y se sacan de la pila utilizando instrucciones específicas: PUSH para introducir elementos y POP para sacar elementos. En el modo de 64 bits los elementos que se pueden introducir en la pila y sacar de ella han de ser valores de 16 o 64 bits; por lo tanto, el registro RSP se decrementa o incrementa en 2 u 8 unidades respectivamente. Al ejecutar la instrucción PUSH, se actualiza el valor de RSP decrementándose en 2 u 8 unidades y se traslada el dato especificado por el operando hacia la cima de la pila. Al ejecutar la instrucción POP, se traslada el valor que está en la cima de la pila hacia el operando de la instrucción y se actualiza el valor de RSP incrementándose en 2 u 8 unidades.

#### **Acceso a datos**

Para acceder a datos de memoria en ensamblador, como sucede en los lenguajes de alto nivel, lo haremos por medio de variables que deberemos definir previamente para reservar el espacio necesario para almacenar la información.

En la definición podemos especificar el nombre de la variable, el tamaño y un valor inicial; para acceder al dato lo haremos con los operandos de las instrucciones especificando un modo de direccionamiento determinado.

Por ejemplo, var dd 12345678h es la variable con el nombre *var* de tamaño 4 bytes inicializada con el valor 12345678h. mov eax, dword[var] es la instrucción en la que accedemos a la variable *var* utilizando direccionamiento directo a memoria, en la que especificamos que queremos acceder a 4 bytes (dword) y transferimos el contenido, el valor 12345678h, al registro *eax*.

En ensamblador hay que estar muy alerta cuando accedemos a las variables que hemos definido. Las variables se guardan en memoria consecutivamente a medida que las declaramos y no existe nada que delimite las unas de las otra

|  |
| --- |
| .data  var1 db 0 ;variable definida de 1 byte  var2 db 61h ;variable definida de 1 byte  var3 dw 0200h ;variable definida de 2 bytes  var4 dd 0001E26Ch ;variable definida de 4 bytes |

Las variables se encontrarán en memoria tal como muestra la tabla.

|  |  |
| --- | --- |
| Dirección | Valor |
| var1 | 00h |
| var2 | 61h |
| var3 | 00h |
|  | 02h |
| var4 | 6Ch |
|  | E2h |
|  | 01h |
|  | 00h |

Si ejecutamos la instrucción siguiente: mov eax, dword[var1]

Cuando accedemos a var1 como una variable de tipo DWORD el procesador tomará como primer byte el valor de var1, pero también los 3 bytes que están a continuación, por lo tanto, como los datos se tratan en formato *little-endian*, consideraremos DWORD [var1] = 02006100h y este es el valor que llevaremos a EAX (eax=02006100h). Si este acceso no es el deseado, el compilador no reportará ningún error, ni tampoco se producirá un error durante la ejecución; solo podremos detectar que lo estamos haciendo mal probando el programa y depurando.

#### 

#### **Subrutinas en ensamblador**

Una subrutina es una unidad de código autocontenida, diseñada para llevar a cabo una tarea determinada y tiene un papel determinante en el desarrollo de programas de manera estructurada. Una subrutina en ensamblador sería equivalente a una función en C.

Básicamente, una subrutina es un conjunto de instrucciones que inician su ejecución en un punto de código identificado con una etiqueta que será el nombre de la subrutina, y finaliza con la ejecución de una instrucción *ret*, instrucción de retorno de subrutina, que provoca un salto a la instrucción siguiente desde donde se ha hecho la llamada (*call*).

Consideraciones importantes a la hora de definir una subrutina:

* Debemos almacenar los registros modificados dentro de la subrutina para dejarlos en el mismo estado en el que se encontraban en el momento de hacer la llamada a la subrutina, salvo los registros que se utilicen para devolver un valor. Para almacenar los registros modificados utilizaremos la pila.
* Para mantener la estructura de una subrutina y para que el programa funcione correctamente, no se pueden efectuar saltos a instrucciones de la subrutina; siempre finalizaremos la ejecución de la subrutina con la instrucción *ret*.

Para hacer la llamada a la subrutina se utiliza la instrucción call y se indica la etiqueta que define el punto de entrada a la subrutina.

La instrucción *call* almacena en la pila la dirección de retorno (la dirección de la instrucción que se encuentra a continuación de la instrucción *call*) y entonces transfiere el control del programa a la subrutina, cargando en el registro RIP la dirección de la primera instrucción de la subrutina.

Para finalizar la ejecución de la subrutina, ejecutaremos la instrucción *ret*, que recupera de la pila la dirección del registro RIP que hemos almacenado al hacer *call* y la carga otra vez en el registro RIP; continúa la ejecución del programa con la instrucción que se encuentra después de *call*.

Si al hacer el *ret* no tenemos en la cima de la pila la dirección de retorno que hemos almacenado cuando hemos hecho el *call*, por una mala gestión de la pila dentro de la subrutina, la instrucción *ret* utilizará el valor de la cima de la pila como dirección de retorno y se perderá el hilo de ejecución de nuestro programa.

Para asegurarnos de que una mala gestión de la pila dentro de la subrutina no afecte a la ejecución del código se recomienda guardar el estado de la pila al inicio y recuperarlo antes de salir. De esta manera nos aseguramos de que antes de hacer el *ret* tenemos en la cima de la pila la dirección de retorno almacenada cuando hemos hecho el *call*.

|  |
| --- |
| **section .data**  x dq 5 ; Declaramos las variables que utilizaremos  result dq 0    **section .text**    global main ; Hacemos visible la etiqueta main    factorial:  push rbp ; Almacenar el registro que usaremos de apuntador a la pila rbp  mov rbp, rsp ; Asignar a rbp el valor del registro apuntador rsp  push rbx ; Almacenar en la pila el registro que modificamos  ; y que no se utiliza para retornar el resultado    mov rax, 1  while:  cmp rbx, 1 ; Hacemos la comparación  jle fi ; Si se cumple la condición saltamos a fin  imul rax, rbx  dec rbx  jmp while ; Salta a while  fin: ; En rax tendremos el valor del factorial de rbx  pop rbx ; Restauramos el valor inicial del registro  mov rsp, rbp ; Restauramos el valor inicial de rsp con rbp  pop rbp ; Restauramos el valor inicial de rbp    ret ; Finaliza la ejecución de la subrutina    main:  call factorial ; Llamamos a la subrutina factorial    mov rax, 1  mov rbx, 0  int 80h ; Finaliza la ejecución del programa |

#### **Convenciones de llamada**

Una subrutina puede necesitar que se le transfieran parámetros. Para saber cómo funciona esto hay una serie de **convenciones de llamada** (calling conventions). Las más importante son:

* Los 6 primeros parámetros se pasan de izquierda a derecha a registros en este orden: rdi, rsi, rdx, rcx, r8, r9. En caso de tratarse de float los registros a utilizar serían: xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7.
* Si hay más parámetros que pasar se hará de derecha a izquierda utilizando la pila. Y la función que llama será responsable de borrarlos de la pila cuando la función llamada retorno.
* Cuando la función llamada toma el control, en la cima de la pila (apuntada por el registro rsp) está la dirección de retorno (la de la siguiente instrucción que se habrá de ejecutar cuando la función llamada termine). El primer parámetro estará en [rsp+8], el segundo en [rsp+16]...
* El puntero a la cima de la pila (rsp) debe estar alineado a 16-byte antes de hacer una llamada. Pero el proceso de hacer una llamada añade la dirección de retorno (8 bytes), por lo que cuando la función llamada toma el control la pila está desalineada. Para solucionarlo se puede hacer algún push o restar 8 de rsp.
* La función llamada debe encargarse de preservar los valores de los registros rbp, rbx, r12, r13, r14, r15. Es decir, si los va a modificar debe salvar primero su valor en la pila y restaurarlo antes de salir en orden inverso a como fueron guardados. Se puede aprovechar para alinear la pila a 16 bytes.
* La función que llama debe salvar los registros rax, rdx y rcx si quiere mantener su valor, porque pueden ser machacados por la función llamada. Para ello los copia en la pila antes de llamar a la subrutina y los recupera cuando retorna.
* Para devolver el valor de retorno utilizamos el registro rax si se trata de un entero o xmm0 en caso de float.

#### **Instrucciones condicionales**

Después de una operación lógica o aritmética o de la instrucción de comparación el procesador ajusta una serie de bits en rflags. Los más interesantes son:

* s (sign)
* z (zero)
* c (carry)
* o (overflow)

Una vez hecho esto podemos ejecutar tres procesos básicos:

* Jump para un salto condicional. Ej: (jz label) salta a la etiqueta label si el flag 0 es activo
* Conditional move para instrucción mov condicionada. Ej: (cmovno x,y) se ejecuta si no hay overflow
* Set. Ej: (setc x) x =1 si el flag carry está encendido (se cumple la condición evaluada) y 0 si no

#### **Llamadas al sistema o funciones del sistema operativo**

Desde un programa en ensamblador podemos hacer llamadas a diferentes funciones del núcleo (*kernel*) del sistema operativo; es lo que se conoce como *system calls* o *kernel system calls*. El lenguaje de ensamblador proporciona dos mecanismos para poder hacer llamadas al sistema operativo:

**1) *int 80h***: este es el mecanismo tradicional en procesadores x86 y, por lo tanto, también está disponible en los procesadores con arquitectura x86-64. El servicio que se solicita se especifica mediante el registro RAX. Los parámetros necesarios para la ejecución del servicio vienen especificados por medio de los registros RBX, RCX, RDX, RSI, RDI y RBP.

**2) *syscall***: los procesadores de la arquitectura x86-64 proporcionan un mecanismo más eficiente de hacer llamadas al sistema, la instrucción *syscall*. El servicio solicitado también se especifica por medio de RAX, pero los números que identifican cada servicio son diferentes de los utilizados con la instrucción *int* 80h. Los parámetros se especifican por medio de los registros RDI, RSI, RDX, RCX, R8 y R9.

El sistema operativo proporciona al programador muchas funciones de diferentes tipos; Algunas de las más usadas son:

* Lectura del teclado
* Escritura por pantalla.
* Retorno al sistema operativo

##### Lectura de caracteres desde el teclado

Lee caracteres del teclado hasta que se pulsa la tecla ENTER. La lectura de caracteres se hace llamando a la función de lectura *read*. Para utilizar esta función hay que especificar el descriptor de archivo que se utilizará; en el caso de una lectura de teclado se utiliza el descriptor correspondiente a la entrada estándar, un 0 en este caso. Según si se utiliza *int 80h* o *syscall*, los parámetros son los siguientes:

**1) *int 80h***

**a)** Parámetros de entrada

* RAX = 3
* RBX = 0, descriptor correspondiente a la entrada estándar (teclado)
* RCX = dirección de la variable de memoria donde se guardará la cadena leída
* RDX = número máximo de caracteres que se leerán

**b)** Parámetros de salida

* RAX = número de caracteres leídos
* La variable indicada se llena con los caracteres leídos.

**2) *syscall***

**a)** Parámetros de entrada

* RAX = 0
* RDI = 0, descriptor correspondiente a la entrada estándar (teclado)
* RSI = dirección de la variable de memoria donde se guardará la cadena leída
* RDX = número máximo de caracteres que se leerán

**b)** Parámetros de salida

* RAX = número de caracteres leídos

##### Escritura de caracteres por pantalla

La escritura de caracteres por pantalla se efectúa llamando a la función de escritura *write*. Para utilizar esta función hay que especificar el descriptor de archivo que se utilizará; en el caso de una escritura por pantalla se utiliza el descriptor correspondiente a la salida estándar, un 1 en este caso.

**1) *int 80h***

**a)** Parámetros de entrada

* RAX = 4
* RBX = 1, descriptor correspondiente a la salida estándar (pantalla)
* RCX = dirección de la variable de memoria que queremos escribir, la variable ha de estar definida con un byte 0 al final
* RDX = tamaño de la cadena que queremos escribir en bytes, incluido el 0 del final

**b)** Parámetros de salida

* RAX = número de caracteres escritos

**2) *syscall***

**a)** Parámetros de entrada

* RAX = 1
* RDI = 1, descriptor correspondiente a la salida estándar (pantalla)
* RSI = dirección de la variable de memoria que queremos escribir, la variable ha de estar definida con un byte 0 al final
* RDX = tamaño de la cadena que queremos escribir en bytes, incluido el 0 del final

**b)** Parámetros de salida

* RAX = número de caracteres escritos

##### Retorno al sistema operativo

Finaliza la ejecución del programa y retorna el control al sistema operativo.

**1) *int 80h***

a) Parámetros de entrada

* RAX = 1
* RBX = valor de retorno del programa

**2) *syscall***

**a) Parámetros de entrada**

* RAX = 60
* RDI = valor de retorno del programa

#### **Diferencias entre Linux y Mac**

* En Linux se compila con elf64 y en Mac con macho64
* En Linux las llamadas al sistema son 1, 2, 3… y en Mac 0x02000001, 0x02000002...
* En Mac las funciones usadas por un archivo diferente al que las implementa se usan precedidas de guión bajo
* Parece que el gcc linker en macOS no permite direccionamiento absoluto a menos que toques algunas cosas. Así que cuando vayas a usar etiquetas de memoria debes añadir “default rel” en la parte donde declaras las funciones externas, y usar “lea” como instrucción para obtenerla.
* En Linux el alineamiento a 16-bytes a veces no es obligatorio, pero en Mac sí.

#### **Instrucciones de salto**

Hay muchas instrucciones de salto. Puede haber:

* Salto incondicional: ‘jmp’ . Ej: jmp label → salta a la etiqueta ‘label’ sí o sí.
* Salto condicional: tras la instrucción ‘cmp’ u otra instrucción de operación lógica o aritmética que fije los flags en el registro ‘rflags’, dependiendo de la condición y de la situación de los flags se realiza el salto o no. Ej: cmp ecx edx → lo que hace es restar el operando origen (el último) al operando destino y fijar los flags correspondientes en función del resultado. Luego puedes utilizar las instrucciones de salto.

**1)** Instrucciones que no tienen en cuenta el signo

Instrucción Descripción Condición

---------- ------------------------------------------ --------------

JA/JNBE (Jump If Above/Jump If Not Below or Equal) CF=0 y ZF=0

JAE/JNB (Jump If Above or Equal/Jump If Not Below) CF=0

JB/JNAE (Jump If Below/Jump If Not Above or Equal) CF=1

JBE/JNA (Jump If Below or Equal/Jump If Not Above) CF=1 o ZF=1

**2)** Instrucciones que tienen en cuenta el signo

Instrucción Descripción Condición

---------- ------------------------------------------ --------------

JE/JZ (Jump If Equal/Jump If Zero) ZF=1

JNE/JNZ (Jump If Not Equal/Jump If Not Zero) ZF=0

JG/JNLE (Jump If Greater/Jump If Not Less or Equal) ZF=0 y SF=OF

JGE/JNL (Jump If Greater or Equal/Jump If Not Less) SF=OF

JL/JNGE (Jump If Less/Jump If Not Greater or Equal) S≠FOF

JLE/JNG (Jump If Less or Equal/Jump If Not Greater) ZF=1 o S≠FOF

**3)** Instrucciones que comprueban el valor de un bit de resultado

Instrucción Descripción Condición

---------- ------------------------------------------ --------------

JC (Jump If Carry flag set) CF=1

JNC (Jump If Carry flag Not set) CF=0

JO (Jump If Overflow flag set) OF=1

JNO (Jump If Overflow flag Not set) OF=0

JS (Jump If Sign flag set) SF=1

JNS (Jump If Sign flag Not set) SF=0

#### **Convenciones de los registros**

Cuando programas en lenguaje ensamblador puedes usar los registros que quieras (con ciertas limitaciones). Pero existen una serie de convenciones que suele ser preferible respetar.

* RAX → Almacena el valor de retorno de una función. También se usa como acumulador
* RBX → Almacena un puntero a la base de la sección de datos
* RCX → Se suele usar como contador para operaciones con strings o loops
* RDX →
* RSI → Origen para operaciones con strings
* RDI → Destino para operaciones con strings
* RSP → Puntero a la cima de la pila
* RBP → Puntero a la base de la pila
* RIP → Puntero a la próxima instrucción a ejecutar. Su valor no se puede cambiar directamente, solo con instrucciones como salto

#### **Otras consideraciones**

* Para funciones que no llaman otras funciones (leaf-node functions) hay un espacio de 128-bytes justo delante del punteo a la parte alta de la pila, llamado zona roja (red zone) que los compiladores usan para almacenasr variables locales. Así, en funciones que no llaman a otras funciones el compilador puede ahorrarse algunas instrucciones al principio de la función (ajuste de rsp y rbp).
* Si la función llamada acepta argumentos variables, el número de argumentos de tipo float o double pasados a la función debe guardarse en el registro ‘al’ por parte de la función que llama.
* No se puede ejecutar una instrucción ‘mov’ de memoria a memoria.
* El operador destino no puede ser un valor inmediato.
* Xor se usa a menudo para poner a 0 un registro (xor rax, rax) ya que es más rápido que ‘mov rax, 0’
* La instrucción ‘test’ es como la instrucción ‘and’ pero no almacena el resultado
* Las instrucciones ‘mul’ y ‘div’ utilizan implícitamente el valor que hay en eax y almacenan ahí el resultado.   
  Ej: mul ebx → eax = eax \* ebx

#### 

#### **Gestión de errores en los macOS de 42**

En FreeBSD (macOS de 42) los errores de las llamadas al sistema no guardan automáticamente el número de error en errno, sino que devuelven en rax el número de error correspondiente. Y a la vez enciende el ‘carry flag’. Y con la función \_\_\_error puedes hayar la dirección donde se aloja errno. Así que lo que hay que hacer para definir el error es:

* Si el carry flag está encendido guardo rax (con el valor del error) en otro registro, por ejemplo r15.
* Llamo a \_\_\_error para saber la dirección de errno.
* Guardo en errno el número de error que había recibido (y que guardé en r15)
* Guardo en rax -1 como valor de retorno porque ha habido un error

#### **Manejo del marco de la pila**

**Enlaces:**

<https://cs.lmu.edu/~ray/notes/nasmtutorial/>

<http://cv.uoc.edu/annotation/8255a8c320f60c2bfd6c9f2ce11b2e7f/619469/PID_00218273/PID_00218273.html>

<https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/>

<https://filippo.io/linux-syscall-table/>

<https://docs.microsoft.com/es-es/cpp/build/x64-calling-convention?view=msvc-160&viewFallbackFrom=vs-2019>

<https://www.nasm.us/doc/>

<https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html>

<https://stackoverflow.com/questions/672461/what-is-stack-alignment>

<https://github.com/cacharle/libasm_test>

<https://www.youtube.com/watch?v=_UP7WJ8iODY&list=PLmxT2pVYo5LB5EzTPZGfFN0c2GDiSXgQe&index=3>

<https://www.youtube.com/watch?v=Fz7Ts9RN0o4&list=PLetF-YjXm-sCH6FrTz4AQhfH6INDQvQSn&index=6>