Programación en ensamblador (x86-64)

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00218269

UOC
Universitat Oberta de Catalunya

Índice

Introducción..... 7

Objetivos..... 9

1. Arquitectura del computador..... 11
1.1. Modos de operación 11
1.1.1. Modo extendido de 64 bits 13
1.1.2. Modo heredado de 16 y 32 bits 14
1.1.3. El modo de gestión de sistema 15
1.2. El modo de 64 bits 15
1.2.1. Organización de la memoria 16
1.2.2. Registros 18

2. Lenguajes de programación..... 22
2.1. Entorno de trabajo 23

3. El lenguaje de ensamblador para la arquitectura x86-64..... 25
3.1. Estructura de un programa en ensamblador 25
3.2. Directivas 26
3.2.1. Definición de constantes 26
3.2.2. Definición de variables 27
3.2.3. Definición de otros elementos 31
3.3. Formato de las instrucciones 33
3.3.1. Etiquetas 34
3.4. Juego de instrucciones y modos de direccionamiento 35
3.4.1. Tipos de operandos de las instrucciones x86-64 36
3.4.2. Modos de direccionamiento 39
3.4.3. Tipos de instrucciones 43

4. Introducción al lenguaje C..... 46
4.1. Estructura de un programa en C 46
4.1.1. Generación de un programa ejecutable 47
4.2. Elementos de un programa en C 48
4.2.1. Directivas 48
4.2.2. Variables 49
4.2.3. Operadores 50
4.2.4. Control de flujo 52
4.2.5. Vectores 54
4.2.6. Apuntadores 56
4.2.7. Funciones 57
4.2.8. Funciones de E/S 58

5. Conceptos de programación en ensamblador y C 61
5.1. Acceso a datos 61
5.1.1. Estructuras de datos 63
5.1.2. Gestión de la pila 66
5.2. Operaciones aritméticas 68
5.3. Control de flujo 68
5.3.1. Estructura *if* 69
5.3.2. Estructura *if-else* 70
5.3.3. Estructura *while* 70
5.3.4. Estructura *do-while* 71
5.3.5. Estructura *for* 72
5.3.6. Estructura switch-case 72
5.4. Subrutinas y paso de parámetros 73
5.4.1. Definición de subrutinas en ensamblador 74
5.4.2. Llamada y retorno de subrutina en ensamblador 75
5.4.3. Paso de parámetros a la subrutina y retorno de resultados 77
5.4.4. Variables locales en ensamblador 83
5.4.5. Llamadas a subrutinas y paso de parámetros desde C ... 84
5.5. Entrada/salida 89
5.5.1. E/S programada 90
5.6. Controlar la consola 92
5.7. Funciones del sistema operativo (*system calls*) 93
5.7.1. Lectura de una cadena de caracteres desde el teclado ... 94
5.7.2. Escritura de una cadena de caracteres por pantalla 95
5.7.3. Retorno al sistema operativo (*exit*) 96

6. Anexo: manual básico del juego de instrucciones 98
6.1. ADC: suma aritmética con bit de transporte 99
6.2. ADD: suma aritmética 100
6.3. AND: Y lógica 101
6.4. CALL: llamada a subrutina 102
6.5. CMP: comparación aritmética 103
6.6. DEC: decrementa el operando 104
6.7. DIV: división entera sin signo 104
6.8. IDIV: división entera con signo 106
6.9. IMUL: multiplicación entera con signo 107
6.9.1. IMUL fuente: un operando explícito 107
6.9.2. IMUL destino, fuente: dos operandos explícitos 108
6.10. IN: lectura de un puerto de entrada/salida 109
6.11. INC: incrementa el operando 110
6.12. INT: llamada a una interrupción software 110
6.13. IRET: retorno de interrupción 111
6.14. Jxx: salto condicional 112
6.15. JMP: salto incondicional 113
6.16. LOOP: bucle hasta RCX=0 114

6.17. MOV: transferir un dato 114
6.18. MOVSX/MOVSXD: transferir un dato con extensión de signo .. 115
6.19. MOVZX: transferir un dato añadiendo ceros 117
6.20. MUL: multiplicación entera sin signo 118
6.21. NEG: negación aritmética en complemento a 2 119
6.22. NOT: negación lógica (negación en complemento a 1) 120
6.23. OUT: escritura en un puerto de entrada/salida 120
6.24. OR: o lógica 121
6.25. POP: extraer el valor de la cima de la pila 122
6.26. PUSH: introducir un valor en la pila 123
6.27. RET: retorno de subrutina 125
6.28. ROL: rotación a la izquierda 125
6.29. ROR: rotación a la derecha 126
6.30. SAL: desplazamiento aritmético (o lógico) a la izquierda 127
6.31. SAR: desplazamiento aritmético a la derecha 128
6.32. SBB: resta con transporte (*borrow*) 130
6.33. SHL: desplazamiento lógico a la izquierda 131
6.34. SHR: desplazamiento lógico a la derecha 131
6.35. SUB: resta sin transporte 132
6.36. TEST: comparación lógica 133
6.37. XCHG: intercambio de operandos 133
6.38. XOR: o exclusiva 134

Introducción

En este módulo nos centraremos en la programación de bajo nivel con el fin de conocer las especificaciones más relevantes de una arquitectura real concreta. En nuestro caso se trata de la arquitectura x86-64 (también denominada *AMD64* o *Intel 64*).

El lenguaje utilizado para programar a bajo nivel un computador es el lenguaje de ensamblador, pero para facilitar el desarrollo de aplicaciones y ciertas operaciones de E/S, utilizaremos un lenguaje de alto nivel, el lenguaje C; de esta manera, podremos organizar los programas según las especificaciones de un lenguaje de alto nivel, que son más flexibles y potentes, e implementar ciertas funciones con ensamblador para trabajar a bajo nivel los aspectos más relevantes de la arquitectura de la máquina.

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. También puede aclarar algunas de las características de los lenguajes de alto nivel que quedan escondidas en su compilación.

El contenido del módulo se divide en siete apartados que tratan los siguientes temas:

- Descripción de la arquitectura x86-64 desde el punto de vista del programador, y en concreto del modo de 64 bits, en el cual se desarrollarán las prácticas de programación.

- Descripción del entorno de programación con el que se trabajará: edición, compilación y depuración de los programas en ensamblador y en C.

- Descripción de los elementos que componen un programa en ensamblador.

- Introducción a la programación en lenguaje C.

- Conceptos de programación en lenguaje de ensamblador y en lenguaje C, y cómo utilizar funciones escritas en ensamblador dentro de programas en C.

- También se incluye una referencia de las instrucciones más habituales del lenguaje de ensamblador con ejemplos de uso de cada una.

Este módulo no pretende ser un manual que explique todas las características del lenguaje de ensamblador y del lenguaje C, sino una guía que permita iniciarse en la programación a bajo nivel (hacer programas) con la ayuda de un lenguaje de alto nivel como el C.

Objetivos

Con el estudio de este módulo se pretende que el estudiante alcance los objetivos siguientes:

1. Aprender a utilizar un entorno de programación con lenguaje C y lenguaje de ensamblador.

2. Conocer el repertorio de instrucciones básicas de los procesadores de la familia x86-64, las diferentes formas de direccionamiento para tratar los datos y el control de flujo dentro de un programa.

3. Saber estructurar en subrutinas un programa; pasar y recibir parámetros.

4. Tener una idea global de la arquitectura interna del procesador según las características del lenguaje de ensamblador.

1. Arquitectura del computador

En este apartado se describirán a grandes rasgos los modos de operación y los elementos más importantes de la organización de un computador basado en la arquitectura x86-64 desde el punto de vista del juego de instrucciones utilizado por el programador.

x86-64 es una ampliación de la arquitectura x86. La arquitectura x86 fue lanzada por Intel con el procesador Intel 8086 en el año 1978 como una arquitectura de 16 bits. Esta arquitectura de Intel evolucionó a una arquitectura de 32 bits cuando apareció el procesador Intel 80386 en el año 1985, denominada inicialmente *i386* o *x86-32* y finalmente *IA-32*. Desde 1999 hasta el 2003, AMD amplió esta arquitectura de 32 bits de Intel a una de 64 bits y la llamó *x86-64* en los primeros documentos y posteriormente *AMD64*. Intel pronto adoptó las extensiones de la arquitectura de AMD bajo el nombre de *IA-32e* o *EM64T*, y finalmente la denominó *Intel 64*.

IA-64

Hay que apuntar que, con el procesador Itanium, Intel ha lanzado una arquitectura denominada *IA-64*, derivada de la IA-32 pero con especificaciones diferentes de la arquitectura x86-64 y que no es compatible con el juego de instrucciones desde un punto de vista nativo de las arquitecturas x86, x86-32 o x86-64. La arquitectura IA-64 se originó en Hewlett-Packard (HP) y más tarde fue desarrollada conjuntamente con Intel para ser utilizada en los servidores empresariales y sistemas de computación de alto rendimiento.

La arquitectura x86-64 (AMD64 o Intel 64) de 64 bits da un soporte mucho mayor al espacio de direcciones virtuales y físicas, proporciona registros de propósito general de 64 bits y otras mejoras que conoceremos más adelante.

Cualquier procesador actual también dispone de una serie de unidades específicas para trabajar con números en punto flotante (juego de instrucciones de la FPU), y de extensiones para trabajar con datos multimedia (juego de instrucciones MMX y SSE en el caso de Intel, o 3DNow! en el caso de AMD). Estas partes no se describen en estos materiales, ya que quedan fuera del programa de la asignatura.

Nos centraremos, por lo tanto, en las partes relacionadas con el trabajo con enteros y datos alfanuméricos.

1.1. 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.

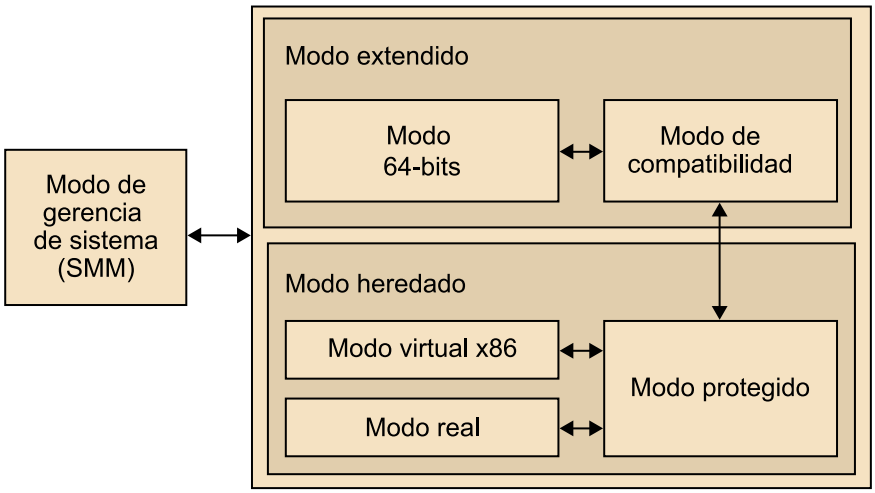
El modo de supervisor es utilizado por el núcleo del sistema para las tareas de bajo nivel que necesitan un acceso sin restricciones al hardware, como puede ser el control de la memoria o la comunicación con otros dispositivos. El modo protegido, en cambio, se utiliza para casi todo el resto de las tareas.

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. Puede haber otros modos similares al protegido, como el modo virtual, que se utiliza para emular otros procesadores de la misma familia, y de esta manera mantener la compatibilidad con los procesadores anteriores.

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.

Modos de operación de la arquitectura x86-64



Características de los dos modos principales de operación en la arquitectura x86-64

Modo de operación		Sistema operativo	Las aplicaciones necesitan recompilación	Por defecto		Tamaño de los registros de propósito general
				Tamaño (en bits) de las direcciones	Tamaño (en bits) de los operandos	
Modo extendido	Modo de 64 bits	Sistema operativo de 64 bits	sí	64	32	64
	Modo compatibilidad		no	32		32
				16	16	16
Modo heredado	Modo protegido	Sistema operativo de 32 bits	no	32	32	32
				16	16	
	Modo virtual-8086			16	16	16
	Modo real	Sistema operativo de 16 bits				

1.1.1. Modo extendido de 64 bits

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 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 2^{64} bytes.

El tamaño por defecto de los operandos se mantiene en 32 bits para la mayoría de las instrucciones.

El tamaño por defecto puede ser cambiado individualmente en cada instrucción mediante modificadores. Además, soporta direccionamiento relativo a PC (RIP en esta arquitectura) en el acceso a los datos de cualquier instrucción.

Modo de compatibilidad

El modo de compatibilidad permite a un sistema operativo de 64 bits ejecutar directamente aplicaciones de 16 y 32 bits sin necesidad de 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.

Desde el punto de vista de las aplicaciones, se ve como si se estuviera trabajando en el modo protegido dentro del modo heredado.

1.1.2. Modo heredado 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.

Dentro de este modo general hay tres modos:

1) Modo real. Implementa el modo de programación del Intel 8086, con algunas extensiones, como la capacidad de poder pasar al modo protegido o al modo de gestión del sistema. El procesador se coloca en modo real al iniciar el sistema y cuando este se reinicia.

Es el único modo de operación que permite utilizar un sistema operativo de 16 bits.

El modo real se caracteriza por disponer de un espacio de memoria segmentado de 1 MB con direcciones de memoria de 20 bits y acceso a las direcciones del hardware (sistema de E/S). No proporciona soporte para la protección de memoria en sistemas multitarea ni de código con diferentes niveles de privilegio.

2) Modo protegido. Este es el modo por defecto del procesador. Permite utilizar características como la memoria virtual, la paginación o la computación multitarea.

Entre las capacidades de este modo está la posibilidad de ejecutar código en modo real, modo virtual-8086, en cualquier tarea en ejecución.

3) Modo virtual 8086. Este modo permite ejecutar programas de 16 bits como tareas dentro del modo protegido.

1.1.3. El modo de gestión de sistema

El modo de gestión de sistema o *system management mode* (SMM) es un modo de operación transparente del software convencional (sistema operativo y aplicaciones). En este modo se suspende la ejecución normal (incluyendo el sistema operativo) y se ejecuta un software especial de alto privilegio diseñado para controlar el sistema. Tareas habituales de este modo son la gestión de energía, tareas de depuración asistidas por hardware, ejecución de microhardware o un software asistido por hardware. Este modo es utilizado básicamente por la BIOS y por los controladores de dispositivo de bajo nivel.

Accedemos al SMM mediante una interrupción de gestión del sistema (SMI, *system management interrupt*). Una SMI puede ser generada por un acontecimiento independiente o ser disparada por el software del sistema por el acceso a una dirección de E/S considerada especial por la lógica de control del sistema.

1.2. El modo de 64 bits

En este subapartado analizaremos con más detalle las características más importantes del modo de 64 bits de la arquitectura x86-64.

Nota

Es importante para el programador de bajo nivel conocer las características más relevantes de este modo, ya que será el modo en el que se desarrollarán las prácticas de programación.

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

1) Espacio de memoria: un programa en ejecución en este modo puede acceder a un espacio de direcciones lineal de 2^{64} bytes. El espacio físico que realmente puede dirigir el procesador es inferior y depende de la implementación concreta de la arquitectura.

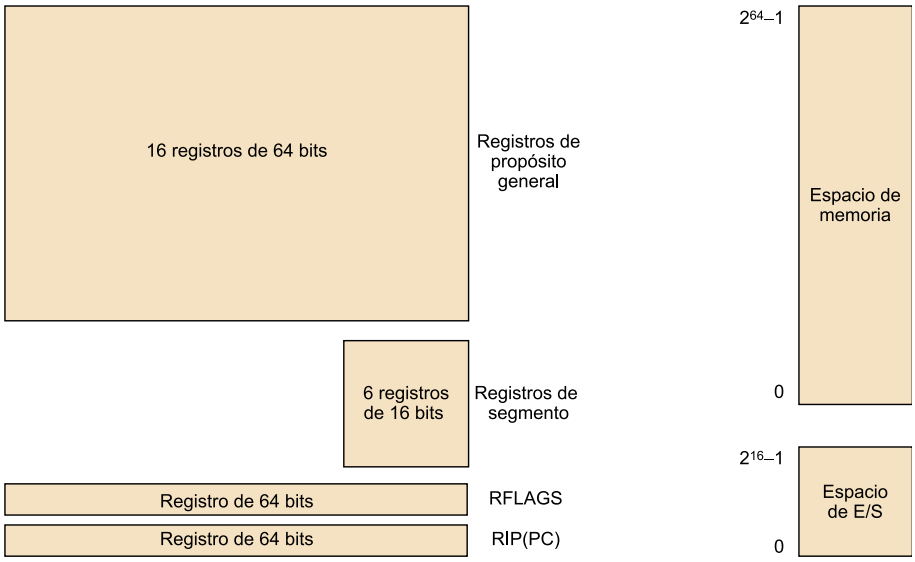
2) Registros: hay 16 registros de propósito general de 64 bits, que soportan operaciones de byte (8 bits), word (16 bits), double word (32 bits) y quad word (64 bits).

- 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.

Entorno de ejecución en el modo de 64 bits



1.2.1. Organización de la memoria

El procesador accede a la memoria utilizando direcciones físicas de memoria. El tamaño del espacio de direcciones físico accesible para los procesadores depende de la implementación: supera los 4 Gbytes, pero es inferior a los 2^{64} bytes posibles.

En el modo de 64 bits, la arquitectura proporciona soporte a un espacio de direcciones virtual o lineal de 64 bits (direcciones de 0 a $2^{64} - 1$), pero como el espacio de direcciones físico es inferior al espacio de direcciones lineal, es necesario un mecanismo de correspondencia entre las direcciones lineales y las direcciones físicas (mecanismo de paginación).

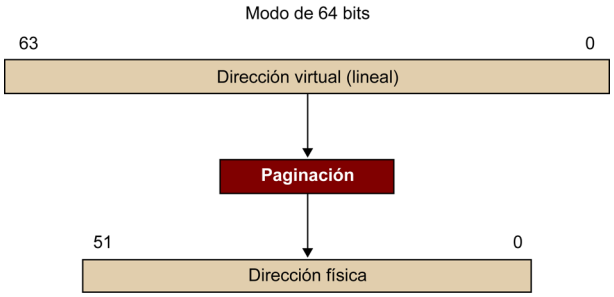
Al trabajar en un espacio lineal de direcciones, no se utilizan mecanismos de segmentación de la memoria, de manera que no son necesarios los registros de segmentos, excepto los registros de segmento FS y GS, que se pueden utilizar como registro base en el cálculo de direcciones de los modos de direccionamiento relativo.

Paginación

Este mecanismo es transparente para los programas de aplicación, y por lo tanto para el programador, y viene gestionado por el hardware del procesador y el sistema operativo.

Las direcciones virtuales son traducidas a direcciones físicas de memoria utilizando un sistema jerárquico de tablas de traducción gestionadas por el software del sistema (sistema operativo).

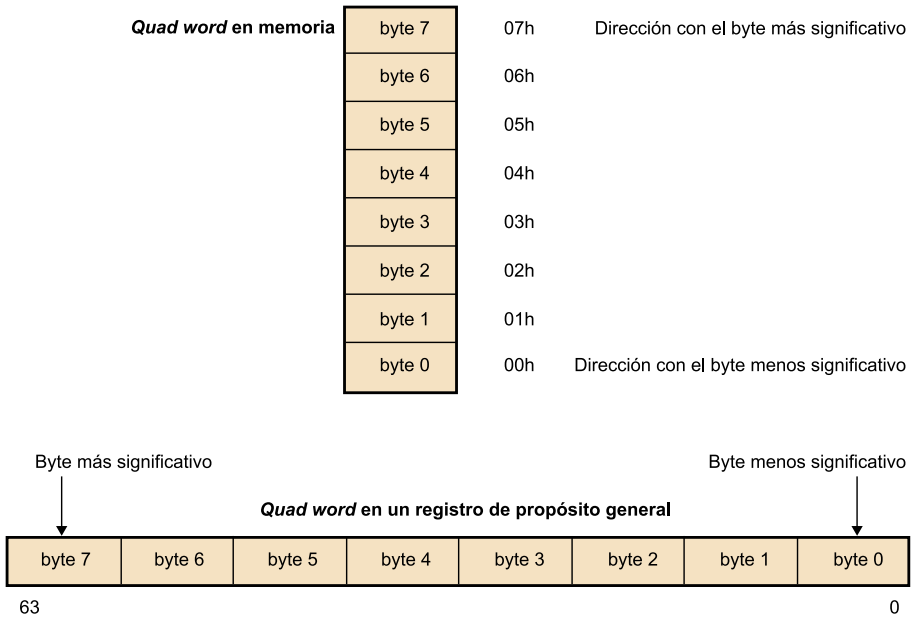
Básicamente, una dirección virtual se divide en campos, y cada campo actúa como índice dentro de una de las tablas de traducción. Cada valor en la posición indexada actúa como dirección base de la tabla de traducción siguiente.



Orden de los bytes

Los procesadores x86-64 utilizan un sistema de ordenación de los bytes cuando se accede a los datos que se encuentran almacenados en la memoria. En concreto, se utiliza un sistema *little-endian*, en el cual el byte de menos peso de un dato ocupa la dirección más baja de memoria.

En los registros también se utiliza el orden *little-endian* y por este motivo el byte menos significativo de un registro se denomina *byte 0*.



Tamaño de las direcciones

Los programas que se ejecutan en el modo de 64 bits generan directamente direcciones de 64 bits.

Modo compatibilidad

Los programas que se ejecutan en el modo compatibilidad generan direcciones de 32 bits. Estas direcciones son extendidas añadiendo ceros a los 32 bits más significativos de la

dirección. Este proceso es gestionado por el hardware del procesador y es transparente para el programador.

Tamaño de los desplazamientos y de los valores inmediatos

En el modo de 64 bits los desplazamientos utilizados en los direccionamientos relativos y los valores inmediatos son siempre de 32 bits, pero vienen extendidos a 64 bits manteniendo el signo.

Hay una excepción a este comportamiento: en la instrucción MOV se permite especificar un valor inmediato de 64 bits.

1.2.2. 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

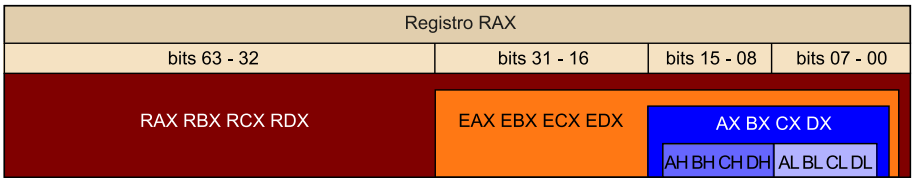
Son 16 registros de datos de 64 bits (8 bytes): RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP y R8-R15.

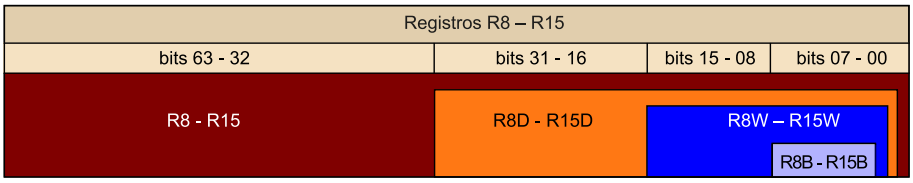
Los 8 primeros registros se denominan de manera parecida a los 8 registros de propósito general de 32 bits disponibles en la arquitectura IA-32 (EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP).

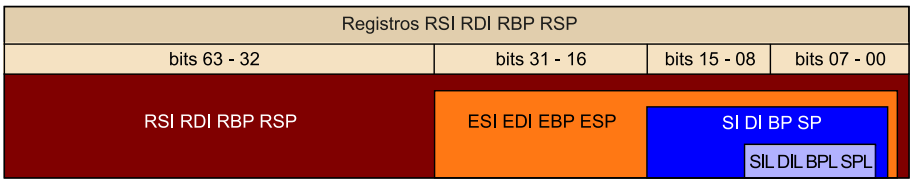
Los registros son accesibles de cuatro maneras diferentes:

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.







Ejemplo

RAX[63-0]				Registro de 8 bytes
EAX[31-0]	= RAX[31-0]			Registro de 4 bytes
AX[15-0]	= EAX[15-0]	= RAX[15-0]		Registro de 2 bytes
AH[7-0]	= AX[15-8]	= EAX[15-8]	= RAX[15-8]	Registro de 1 byte
Al/AL[7-0]	= AX[7-0]	= EAX[7-0]	= RAX[7-0]	Registre de 1 byte

Los registros EAX, AX, AH, AL no son registros independientes, sino que son partes del registro RAX. Esto sucede con todos los registros de propósito general: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14 y R15.

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 con otras finalidades, 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).

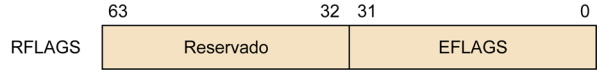
3) Registro de estado o *Flags register* (RFLAGS): es un registro de 64 bits que contiene información sobre el estado del procesador e información sobre el resultado de la ejecución de las instrucciones.

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.

Bits de resultado

Las instrucciones de salto condicional consultan algunos de los bits del registro, denominados *bits de resultado*. Hay más información sobre el tema en el manual básico del juego de instrucciones en el apartado 7 de este módulo.

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 tabla se describe el significado de los bits de este registro.

bit 0	CF	Carry Flag	0 = no se ha producido transporte; 1 = sí que se ha producido transporte
bit 1		No definido	
bit 2	PF	Parity Flag	0 = número de bits 1 es impar; 1 = número de bits 1 es par

bit 3		No definido	
bit 4	AF	Auxiliary Carry Flag	0 = no se ha producido transporte en operaciones BCD; 1 = sí que se ha producido transporte en operaciones BCD
bit 5		No definido	
bit 6	ZF	Zero Flag	0 = el resultado no ha sido cero; 1 = el resultado ha sido cero
bit 7	SF	Sign Flag	0 = el resultado no ha sido negativo; 1 = el resultado ha sido negativo
bit 8	TF	Trap Flag	Facilita la ejecución paso a paso.
bit 9	IF	Interrupt Enable Flag	Reservado para el sistema operativo en modo protegido.
bit 10	DF	Direction Flag	0 = autoincremento hacia direcciones altas; 1 = autoincremento hacia direcciones bajas
bit 11	OF	Overflow Flag	0 = no se ha producido desbordamiento; 1 = sí que se ha producido desbordamiento
bit 12	IOPL	I/O Privilege Level	Reservado para el sistema operativo en modo protegido.
bit 13	IOPL	I/O Privilege Level	Reservado para el sistema operativo en modo protegido.
bit 14	NT	Nested Task Flag	Reservado para el sistema operativo en modo protegido.
bit 15		No definido	
bit 16	RF	Resume Flag	Facilita la ejecución paso a paso.
bit 17	VM	Virtual-86 Mode Flag	Reservado para el sistema operativo en modo protegido.
bit 18	AC	Alignment Check Flag	Reservado para el sistema operativo en modo protegido.
bit 19	VIF	Virtual Interrupt Flag	Reservado para el sistema operativo en modo protegido.
bit 20	VIP	Virtual Interrupt Pending	Reservado para el sistema operativo en modo protegido.
bit 21	ID	CPU ID	Si el bit puede ser modificado por los programas en el espacio de usuario, CPUID está disponible.
bit 22-31		No definidos	

2. Lenguajes de programación

Un programa es un conjunto de instrucciones que siguen unas normas sintácticas estrictas especificadas por un lenguaje de programación concreto y diseñado de manera que, cuando se ejecuta en una máquina concreta, realiza una tarea determinada sobre un conjunto de datos. Los programas, por lo tanto, están formados por código (instrucciones) y datos. De manera genérica, el fichero que contiene el conjunto de instrucciones y la definición de los datos que utilizaremos se denomina **código fuente**.

Para poder ejecutar un programa, lo hemos de traducir a un lenguaje que pueda entender el procesador; este proceso se llama habitualmente *compilación*. Convertimos el código fuente en **código ejecutable**. Este proceso para generar el código ejecutable normalmente se descompone en dos fases: en la primera fase el código fuente se traduce a un **código objeto** y en la segunda fase se enlaza este código objeto y otros códigos objeto del mismo tipo que ya tengamos generados, si es necesario, para generar el código ejecutable final.

Código objeto

El código objeto es un código de bajo nivel formado por una colección organizada de secuencias de códigos siguiendo un formato estándar. Cada secuencia, en general, contiene instrucciones para la máquina en la que se tiene que ejecutar el código para llevar a cabo alguna tarea concreta; también puede tener otro tipo de información asociada (por ejemplo, información de reubicación, comentarios o los símbolos del programa para la depuración).

Para iniciar la ejecución, es necesario que tanto el código como los datos (al menos una parte) estén cargados en la memoria del computador.

Para escribir un programa, hay que utilizar un lenguaje de programación que nos permita especificar el funcionamiento que se quiere que tenga el programa. Hay muchos lenguajes de programación y, según la funcionalidad que queramos dar al programa, será mejor utilizar uno u otro. En muchos casos, elegir uno no es una tarea fácil y puede condicionar mucho el desarrollo y el funcionamiento. Hay muchas maneras de clasificarlos, pero por el interés de esta asignatura y como es una de las maneras más generales de hacerlo, los clasificaremos según el nivel de abstracción (proximidad a la máquina) en dos lenguajes:

1) Lenguajes de bajo nivel

Se denominan *lenguajes de bajo nivel* porque dependen de la arquitectura del procesador en el que queremos ejecutar el programa y porque no disponen de sentencias con una estructura lógica que faciliten la programación y la comprensión del código para el programador, sino que están formados por una lista de instrucciones específicas de una arquitectura.

Podemos distinguir entre dos lenguajes:

a) Lenguaje de máquina. Lenguaje que puede interpretar y ejecutar un procesador determinado. Este lenguaje está formado por instrucciones codificadas en binario (0 y 1). Es generado por un compilador a partir de las especificaciones de otro lenguaje simbólico o de alto nivel. Es muy difícil de entender para el programador y sería muy fácil cometer errores si se tuviera que codificar.

b) Lenguaje de ensamblador. Lenguaje simbólico que se ha definido para que se puedan escribir programas con una sintaxis próxima al lenguaje de máquina, pero sin tener que escribir el código en binario, sino utilizando una serie de mnemónicos más fáciles de entender para el programador. Para ejecutar estos programas también es necesario un proceso de traducción, generalmente denominado *ensamblaje*, pero más sencillo que en los lenguajes de alto nivel.

2) Lenguajes de alto nivel. Los lenguajes de alto nivel no tienen relación directa con un lenguaje de máquina concreto, no dependen de la arquitectura del procesador en el que se ejecutarán y disponen de sentencias con una estructura lógica que facilitan la programación y la comprensión del código para el programador; las instrucciones habitualmente son palabras extraídas de un lenguaje natural, generalmente el inglés, para que el programador las pueda entender mejor. Para poder ejecutar programas escritos en estos lenguajes, es necesario un proceso previo de compilación para pasar de lenguaje de alto nivel a lenguaje de máquina; el código generado en este proceso dependerá de la arquitectura del procesador en el que se ejecutará.

2.1. Entorno de trabajo

El entorno de trabajo que utilizaremos para desarrollar los problemas y las prácticas de programación será un PC basado en procesadores x86-64 (Intel64 o AMD64) sobre el cual se ejecutará un sistema operativo Linux de 64 bits.

El entorno de trabajo se podrá ejecutar de forma nativa sobre un PC con un procesador con arquitectura x86-64, con sistema operativo Linux de 64 bits, o utilizando algún software de virtualización que permita ejecutar un sistema operativo Linux de 64 bits.

Los lenguajes de programación que utilizaremos para escribir el código fuente de los problemas y de las prácticas de programación de la asignatura serán: un lenguaje de alto nivel, el lenguaje C estándar, para diseñar el programa principal y las operaciones de E/S y un lenguaje de bajo nivel, el lenguaje ensamblador x86-64, para implementar funciones concretas y ver cómo trabaja esta arquitectura a bajo nivel.

Hay diferentes sintaxis de lenguaje ensamblador x86-64; utilizaremos la sintaxis NASM (Netwide Assembler), basada en la sintaxis Intel.

3. El lenguaje de ensamblador para la arquitectura x86-64

3.1. Estructura de un programa en ensamblador

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:

```
section .data

section .bss

section .text
```

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 que se declaran 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, esta 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.

En el programa de ejemplo `hola.asm` utilizado anteriormente hemos destacado la declaración de las secciones *.data* y *.text*, la directiva global y la llamada al sistema operativo para finalizar la ejecución del programa.

```
                    ;1: fichero hola.asm
section .data       ;2: Inicio de la sección de datos
  msg db "Hola!",10 ;3:
                    ;4: El 10 corresponde al código ASCII del salto de línea.
                    ;5:
section .text       ;6: Inicio de la sección de código.
  global main       ;7: Esta directiva es para hacer visible
                    ;8: una etiqueta para el compilador de C.
                    ;9:
  main:             ;10: Por defecto el compilador de C reconoce como
                    ;11: punto de inicio del programa la etiqueta main.
                    ;12: Mostrar un mensaje
  mov rax,4         ;13: Pone el valor 4 en el registro rax
                    ;14: para hacer la llamada a la funció write (sys_write)
  mov rbx,1         ;15: Pone el valor 1 en el registro RBX
                    ;16: para indicar el descriptor que hace referencia
                    ;17: a la salida estándar.
  mov rcx,msg       ;18: Pone la dirección de la variable msg
                    ;19: en el registro RCX
  mov rdx,6         ;20: Pone la longitud del mensaje incluido el 10
                    ;21: del final en el registro RDX
  int 80h           ;22: llama al sistema operativo
                    ;23:
                    ;24: devuelve el control al terminal del sistema operativo.
  mov rax,1         ;25: Pone el valor 1 en el registro rax
                    ;26: para hacer la llamada a la función exit (sys_exit)
  mov rbx,0         ;27: Pone el valor 0 en el registro RBX
                    ;28: para indicar el código de retorno (0=sin errores)
  int 80h           ;29: llama al sistema operativo
                    ;30:
```

A continuación, se describen los diferentes elementos que se pueden introducir dentro de las secciones de un programa ensamblador.

3.2. Directivas

Las directivas son pseudooperaciones que solo son reconocidas por el ensamblador. No se deben confundir con las instrucciones, a pesar de que en algunos casos pueden añadir código a nuestro programa. 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.

A continuación, explicaremos algunas de las directivas que podemos encontrar en un programa con código ensamblador y que tendremos que utilizar: definición de constantes, definición de variables y definición de otros elementos.

3.2.1. 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
```

Ejemplos de definiciones de constantes

```
tamañoVec equ 5
ServicioSO equ 80h
Mensaje1 equ 'Hola'
```

3.2.2. Definición de variables

La declaración de variables en un programa en ensamblador se puede incluir en la sección *.data* o en la sección *.bss*, según el uso de cada una.

Sección *.data*, variables inicializadas

Las variables de esta sección 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
```

Ejemplos

```
var1 db 255 ;                    define una variable con el valor FFh
Var2 dw 65535 ;                    en hexadecimal FFFFh
var4 dd 4294967295 ;                en hexadecimal FFFFFFFFh
var8 dq 18446744073709551615 ; en            hexadecimal
FFFFFFFFFFFFFFFFh
```

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

```
tamañoVec equ 5
indexVec db tamañoVec
```

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.

Si se quieren separar los dígitos de los valores numéricos iniciales para facilitar la lectura, se puede utilizar el símbolo '_', sea cual sea la base en la que esté expresado el número.

```
var4 dd 4_294_967_295
var8 dq FF_FF_FF_FF_FF_FF_FF_FFh
```

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*.

```
var db 67   ;el valor 67 decimal
var db 67d ;el mismo valor
```

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.

```
var db 0xFF
var dw 0hA3
var db $43
var dw 33FFh
```

Las definiciones siguientes son incorrectas:

```
var db      $FF ;deberíamos escribir: var db $0FF
var db      FFh ;deberíamos escribir: var db 0FFh
```

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

```
var db 103o
var db 103q
var db 0o103
var db 0q103
```

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

```
var db 0b01000011
var dw 0b0110_1100_0100_0011
var db 01000011b
var dw 0110_0100_0011b
```

Los **caracteres** y las **cadenas** de caracteres han de escribirse entre comillas simples (''), dobles ("") o comillas abiertas *backquotes* (` `):

```
var db 'A'
var db "A"
var db `A`
```

Las cadenas de caracteres (*strings*) se definen de la misma manera:

```
cadena db 'Hola' ;define una cadena formada por 4 caracteres
cadena db "Hola"
cadena db `Hola`
```

Las cadenas de caracteres también se pueden definir como una serie de caracteres individuales separados por comas.

Las definiciones siguientes son equivalentes:

```
cadena db 'Hola'
cadena db 'H','o','l','a'
cadena db 'Hol','a'
```

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)
\0x[valor]: [valor] ha de ser 1 byte en hexadecimal, expresado con 2 dígitos hexadecimales.
\u[valor]: [valor] ha de ser la codificación hexadecimal de un carácter en formato UTF.

```
cadena db `Hola\n`   ;añade el carácter de salto de línea al final
cadena db `Hola\t`   ;añade el carácter de salto de línea al final
cadena db `\e[10,5H`   ;define una secuencia de escape que empieza con
                        ;el carácter ESC = \e = ASCII(27)
cadena db `Hola \u263a` ;muestra: Hola☺
```

Si queremos definir una cadena de caracteres que incluya uno de estos símbolos (', ", `), se debe delimitar la cadena utilizando unas comillas de otro tipo o poner una contrabarra (\) delante.

```
"El ensamblador","El ensamblador",'El ensamblador',
`El ensamblador´, `El ensamblador`,
'Vocales acentuadas "àèéìòóú", con diéresis "ïü" y símbolos
"ç€@#".'
`Vocales acentuadas "àèéìòóú", con diéresis "ïü" y símbolos
"ç€@#".`
```

Si queremos declarar una variable inicializada con un valor que se repite un conjunto de veces, podemos utilizar la directiva *times*.

```
cadena times 4 db 'PILA' ;define una variable inicializada
                         ;con el valor 'PILA' 4 veces
cadena2 db 'PILAPILAPILAPILA' ;es equivalente a la declaración
anterior
```

Vectores

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
```

También podemos utilizar la directiva *times* para inicializar vectores, pero entonces todas las posiciones tendrán el mismo valor.

```
vector5 times 5 dw 0 ;vector formado por 5 palabras inicializadas en 0
```

Valores y constantes numéricas

En ensamblador, todos los valores se almacenan como valores con signo expresados en complemento a 2.

Sección *.bss*, variables no inicializadas

Dentro de 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.

Ejemplos

```
section .bss

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
            ;equivalente a los dos casos anteriores
```

3.2.3. Definición de otros elementos

Otros elementos son:

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 declarar 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.

La directiva tiene el formato siguiente:

```
extern símbolo1, símbolo2, ..., símboloN
```

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.

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.

La directiva tiene el formato siguiente:

```
global símbolo1, símbolo2, ..., símboloN
```

En una misma directiva *global* se pueden declarar tantos símbolos como se quiera separados por comas.

Generación del ejecutable

Hay que hacer el ensamblaje de los dos códigos fuente ensamblador y enlazar los dos códigos objeto generados para obtener el ejecutable.

Ejemplo de utilización de las directivas *extern* y *global*

Prog1.asm

```
global printHola, msg ;hacemos visibles la subrutina
                       ;printHola y la variable msg.
section .data
   msg db "Hola!",10
   ...

section .text
printHola:
   mov rax,4
   mov rbx,1
   mov rcx,msg
   mov rdx,6
   int 80h
   ret
```

Prog2.asm

```
  extern printHola, msg ;indicamos que están declaradas
                      ;en otro fichero.
  global main ;hacemos visible la etiqueta main
                       ;para poder iniciar la ejecución.
  section. text
  main:
    call printHola ;ejecutamos el código del Prog1.asm

    mov rax,4     ;ejecutamos este código pero
    mov rbx,1     ;utilizamos la variable definida
    mov rcx,msg   ;en el fichero Prog1.asm
    mov rdx,6
    int 80h

    mov rax,1
    mov rbx,0
    int 80h
```

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.

El formato de la directiva es:

```
cpu tipo_procesador
```

Ejemplo

cpu 386

cpu x86-64

3.3. Formato de las 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]`

donde *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.

3.3.1. Etiquetas

Una etiqueta hace referencia a un elemento dentro del programa ensamblador. Su función es facilitar al programador 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.

Una cuestión importante que hay que tener en cuenta es que en sintaxis NASM se distingue entre minúsculas y mayúsculas. Por ejemplo, las etiquetas siguientes serían diferentes: Etiqueta1, etiqueta1, ETIQUETA1, eTiQueTa1, ETIqueta1.

Ejemplo

```
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 (podéis ver instrucciones de ruptura de secuencia en el subapartado 3.4.3).

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.

Ejemplo

```
          ...
Etiqueta1: mov rax,0
          ...
          jmp Etiqueta1 ;Esta instrucción salta a la
                        ;instrucción mov rax,0
          ...
```

Una etiqueta se puede escribir en la misma línea que ocupa una instrucción o en otra. El fragmento siguiente es equivalente al fragmento anterior:

```
    ...
Etiqueta1:

    mov rax,0
    ...
    jmp Etiqueta1 ;Esta instrucción salta a la instrucción mov
rax,0
    ...
```

3.4. Juego de instrucciones y modos de direccionamiento

Una instrucción en ensamblador está formada por un *código de operación* (el nombre de la instrucción) que determina qué debe hacer la instrucción, más un conjunto de operandos que expresan directamente un dato, un registro o una dirección de memoria; las diferentes maneras de expresar un operando en una instrucción y el procedimiento asociado que permite obtener el dato se denomina *modo de direccionamiento*.

Existen instrucciones que no tienen ningún operando y que trabajan implícitamente con registros o la memoria; hay instrucciones que tienen solo un operando y también las hay con dos o más operandos (en este caso los operandos se separan con comas):

1) Instrucciones sin ningún operando explícito: *código_operación*

Ejemplo

```
ret    ;retorno de subrutina
```

2) Instrucciones con un solo operando: *código_operación destino*

```
push rax    ;guarda el valor de rax en la pila; rax es un operando fuente
pop rax     ;carga el valor de la cima de la pila en rax; rax es un
            ; operando destino
call subr1  ;llama a la subrutina subr1, subr1 es un operando fuente
inc rax     ;rax=rax+1, rax hace de operando fuente y también
            de operando destino.
```

3) Instrucciones con dos operandos: *código_operación destino, fuente*

Ejemplos

```
mov rax [vec+rsi];mueve el valor de la posición del vector vec
                  ;indicada por rsi; es el operando fuente, en rax,
                  ;rax es el operando destino
add rax, 4        ;rax=rax+4
```

Operando fuente y operando destino

El operando fuente especifica un valor, un registro o una dirección de memoria donde hemos de ir para buscar un dato que necesitamos para ejecutar la instrucción.

El operando destino especifica un registro o una dirección de memoria donde hemos de guardar el dato que hemos obtenido al ejecutar la instrucción. El operando destino también puede actuar como operando fuente y el valor original se pierde cuando guardamos el dato obtenido al ejecutar la instrucción.

3.4.1. Tipos de operandos de las instrucciones x86-64

Operandos fuente y destino

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.

Ejemplos

```
push rax
```

El registro `rax` es un **operando fuente**; la instrucción almacena el valor del operando fuente en la pila del sistema, de manera que la pila es un operando destino implícito.

```
pop rax
```

El registro `rax` se comporta como **operando destino**; la instrucción almacena en `rax` el valor que se encuentra en la cima de la pila del sistema, de manera que la pila es un operando fuente implícito.

```
inc rax
```

El registro `rax` es a la vez **operando fuente y destino**; la instrucción `inc` realiza la operación `rax = rax + 1`, suma el valor original de `rax` con 1 y vuelve a guardar el resultado final en `rax`.

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.

Ejemplos

```
mov rax, rbx
```

El primer operando se comporta solo como operando destino; la instrucción almacena el valor indicado por el segundo operando en el primer operando (`rax = rbx`).

```
add rax, 4
```

El primer operando se comporta al mismo tiempo como operando fuente y destino; la instrucción `add` lleva a cabo la operación `rax = rax + 4`, suma el valor original de `rax` con el valor 4, vuelve a almacenar el resultado en `rax`, y se pierde el valor que teníamos originalmente.

Localización de los operandos

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

Ved también

Para saber qué instrucciones permiten utilizar un tipo de operando determinado, ya sea como operando fuente u operando destino, hay que consultar la referencia de las instrucciones en el apartado 7 de este módulo.

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.

Ejemplos

```
mov al, 10 b      ;un valor inmediato expresado en binario

cinco equ 5 h      ;se define una constante en hexadecimal
mov al, cinco      ;se utiliza el valor de la constante como
                        ;valor inmediato
mov eax, 0xABFE001C ;un valor inmediato expresado en hexadecimal.

mov ax, 'HI'       ;un valor inmediato expresado como una cadena
                       ;de caracteres
```

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.

Ejemplos

```
mov al, 100
mov ax, 1000
mov eax, 100000
mov rax, 10000000
mov rbx, rax
mul bl   ;ax = al * bl,
los registros al y ax
son implícitos,
         ;al como operando
fuente y ax como
operando de destino.
```

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).

Acceso a memoria

En sintaxis NASM se distingue el acceso al contenido de una variable de memoria del acceso a la dirección de una variable. Para acceder al contenido de una variable de memoria hay que especificar el nombre de la variable entre *[ ]*; si utilizamos el nombre de una variable sin *[ ]*, nos estaremos refiriendo a su dirección.

Ejemplo

```
section .data
     var1 dd 100      ;variable de 4 bytes (var1 = 100)

section .text
    mov rax, var1     ;se carga en rax la dirección de la variable var1
    mov rbx, [var1] ;se carga en rbx el contenido de var1, rbx=100
    mov rbx, [rax]   ;esta instrucción hará lo mismo que la anterior.
    mov [var1], rbx ;se carga en var1 el contenido del registro rbx
```

Debemos apuntar que, 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 podemos hacer lo siguiente:

```
;si consideramos que la dirección de memoria a la que hace referencia
var1
;es la dirección 12345678h
mov QWORD [var2], var1 ;se carga en var2 la dirección de var1
mov QWORD [var2],12345678h ;es equivalente a la instrucción anterior.
```

`[var2]` es el operando que hace el acceso a memoria y `var1` se codifica como un valor inmediato.

Tamaño de los operandos

En sintaxis NASM el tamaño de los datos especificados por un operando puede ser de byte, word, double word y quad word. Cabe recordar que lo hace en formato *little-endian*.

- BYTE: indica que el tamaño del operando es de un byte (8 bits).
- WORD: indica que el tamaño del operando es de una palabra (word) o dos bytes (16 bits).
- DWORD: indica que el tamaño del operando es de una doble palabra (double word) o cuatro bytes (32 bits).
- QWORD: indica que el tamaño del operando es de una cuádruple palabra (quad word) u ocho bytes (64 bits).

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.

Ejemplo

```
.data
var1 dd 100              ;variable definida de 4 bytes
. text
mov DWORD [var1], 0ABCh  ;se indica que la variable var1
                         ;es de 32 bits
```

La instrucción siguiente es incorrecta:

```
mov [var1], 0ABCh
```

En el caso anterior, el compilador no sería capaz de averiguar cómo ha de copiar este valor a la variable `[var1]`, aunque cuando lo hemos definido se haya especificado el tipo.

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

Ejemplo

```
inc QWORD [var1]  ;se indica que la posición de memoria afectada es
                  ;de tamaño de 8 bytes. [var1]=[var1]+1
push WORD [var1]  ;se indica que ponemos 2 bytes en la pila.
```

3.4.2. Modos de direccionamiento

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.

Ejemplos

```
mov rax, 0102030405060708h   ;el segundo operando utiliza direccionamiento inmediato
                             ;expresado con 64 bits.
mov QWORD [var], 100         ;el segundo operando utiliza direccionamiento inmediato
                             ;carga el valor 100 y se guarda en var
mov rbx, var                 ;el segundo operando utiliza direccionamiento inmediato
                             ;carga la dirección de var en el registro rbx
mov rbx, var+16 * 2          ;el segundo operando utiliza direccionamiento inmediato
                             ;carga la dirección de var+32 en el registro rbx
```

Para especificar un valor inmediato en una instrucción se utiliza la misma notación que la especificada en la definición de variables inicializadas.

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).

Ejemplo

```
mov rax, rbx     ;los dos operandos utilizan direccionamiento
                 ;directo a registro, rax = rbx
```

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.

Ejemplos

```
mov rax,[var]      ;el segundo operando utiliza direccionamiento
                   ;directo a memoria, rax = [var]
add [suma],rcx     ;el primer operando utiliza direccionamiento
                   ;directo a memoria [suma]=[suma]+rcx
```

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.

Ejemplos

```
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 *[ ]*.

Ejemplos

```
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.

Ejemplos

```
[rbx + rsi * 2 + 4 * (12+127)]
[r9 + r10 * 8]
[r9 - 124 * 8]
[rax * 4 + 12 / 4]
[vec+16 + rsi * 2]
```

Esta es la manera general de expresar un operando para acceder a memoria; los otros modos especificados anteriormente son casos concretos en los que solo se definen algunos de estos elementos.

7) Relativo a PC. 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.

Ejemplo

```
je 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.

Ejemplos

```
push word 23h
push qword [rax]
pop qword [var]
pop bx
```

3.4.3. Tipos de instrucciones

El juego de instrucciones de los procesadores x86-64 es muy amplio. Podemos organizar las instrucciones según los tipos siguientes:

Ved también

En el apartado 6 de este módulo se describe con detalle el formato y la utilización de las instrucciones de los procesadores x86-64 que consideramos más importantes.

1) Instrucciones de transferencia de datos:

- **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.
- **xchg** ***destino, fuente***: intercambia contenidos de los operandos.

2) Instrucciones aritméticas y de comparación:

- **add** ***destino, fuente***: suma aritmética de los dos operandos.
- **adc** ***destino, fuente***: suma aritmética de los dos operandos considerando el bit de transporte.
- **sub** ***destino, fuente***: resta aritmética de los dos operandos.
- **sbb** ***destino, fuente***: resta aritmética de los dos operandos considerando el bit de transporte.
- **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.
- **neg** ***destino***: negación aritmética en complemento a 2.
- **cmp** ***destino, fuente***: comparación de los dos operandos; hace una resta sin guardar el resultado.

3) Instrucciones lógicas y de desplazamiento:

a) Operaciones lógicas:

- **and** ***destino, fuente***: operación 'y' lógica.
- **or** ***destino, fuente***: operación 'o' lógica.
- **xor** ***destino, fuente***: operación `o exclusiva´ lógica.
- **not** ***destino***: negación lógica bit a bit.

- **test *destino, fuente***: comparación lógica de los dos operandos; hace una 'y' lógica sin guardar el resultado.

b) Operaciones de desplazamiento:

- **sal *destino, fuente*** / **shl *destino, fuente***: desplazamiento aritmético/lógico a la izquierda.
- **sar *destino, fuente***: desplazamiento aritmético a la derecha.
- **shr *destino, fuente***: desplazamiento lógico a la derecha.
- **rol *destino, fuente***: rotación lógica a la izquierda.
- **ror *destino, fuente***: rotación lógica a la derecha.
- **rcl *destino, fuente***: rotación lógica a la izquierda considerando el bit de transporte.
- **rcr *destino, fuente***: rotación lógica a la derecha considerando el bit de transporte.

4) Instrucciones de ruptura de secuencia:

a) Salto incondicional:

- **jmp etiqueta**: salta de manera incondicional a la etiqueta.

b) Saltos que consultan un bit de resultado concreto:

- **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.
- **jc *etiqueta***: salta a la etiqueta si el bit de transporte es activo.
- **jnc *etiqueta***: salta a la etiqueta si el bit de transporte no es activo.
- **jo *etiqueta***: salta a la etiqueta si el bit de desbordamiento es activo.
- **jno *etiqueta***: salta a la etiqueta si el bit de desbordamiento no es activo.
- **js *etiqueta***: salta a la etiqueta si el bit de signo es activo.
- **jns *etiqueta***: salta a la etiqueta si el bit de signo no es activo.

c) Saltos condicionales sin considerar el signo:

- **jb *etiqueta*** / **jnae *etiqueta***: salta a la etiqueta si es más pequeño.
- **jbe *etiqueta*** / **jna *etiqueta***: salta a la etiqueta si es más pequeño o igual.
- **ja *etiqueta*** / **jnbe *etiqueta***: salta a la etiqueta si es mayor.
- **jae *etiqueta*** / **jnb *etiqueta***: salta a la etiqueta si es mayor o igual.

d) Saltos condicionales considerando 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.

e) Otras instrucciones de ruptura de secuencia:

- **loop *etiqueta***: decrementa el registro rcx y salta si rcx es diferente de cero.
- **call *etiqueta***: llamada a subrutina.
- **ret**: retorno de subrutina.
- **iret**: retorno de rutina de servicio de interrupción (RSI).
- **int *servicio***: llamada al sistema operativo.

5) Instrucciones de entrada/salida:

- **in *destino, fuente***: lectura del puerto de E/S especificado en el operando fuente y se guarda en el operando destino.
- **out *destino, fuente***: escritura del valor especificado por el operando fuente en el puerto de E/S especificado en el operando destino.

4. Introducción al lenguaje C

En este apartado se hace una breve introducción al lenguaje C. No pretende ser un manual de programación ni una guía de referencia del lenguaje, solo explica los conceptos necesarios para poder desarrollar pequeños programas en C y poder hacer llamadas a funciones implementadas en lenguaje de ensamblador, con el objetivo de entender el funcionamiento del procesador y la comunicación entre un lenguaje de alto nivel y uno de bajo nivel.

4.1. Estructura de un programa en C

Un programa escrito en lenguaje C sigue en general la estructura siguiente:

- Directivas de compilación.
- Definición de variables globales.
- Declaración e implementación de funciones.
- Declaración e implementación de la función *main*.

La única parte imprescindible en todo programa escrito en C es la función *main*, aunque muchas veces hay que incluir algunas directivas de compilación.

Por ejemplo, es habitual utilizar la directiva *include* para incluir otros ficheros en los que se definen funciones de la biblioteca estándar *glibc* que se quieren utilizar en el programa.

Veamos a continuación un primer programa escrito en C:

```
1 /*Fichero hola.c*/
2 #include <stdio.h>
3 int main(){
4 printf ("Hola!\n");
5 return 0;
6 }
```

La primera línea define un comentario; en C los comentarios se escriben entre los símbolos `/*` y `*/`; también se puede utilizar `//` para escribir comentarios de una sola línea.

Ejemplos de comentarios en C

```
//Esto es un comentario de una línea
/* Esto es un
comentario
de varias líneas */
```

La segunda línea corresponde a una directiva; en lenguaje C, las directivas empiezan siempre por el símbolo *#*.

La directiva *include* indica al compilador que incluya el fichero indicado, *stdio.h*, al compilar el programa.

El fichero *stdio.h* incluye la definición de las funciones más habituales para trabajar con la pantalla y el teclado.

La tercera línea declara la función *main*, función principal de todo programa escrito en C en que se iniciará la ejecución del programa; se marca el inicio de la función con el símbolo {.

La cuarta línea es la primera instrucción del *main*, y corresponde a una llamada a la función *printf* de la biblioteca estándar; esta función está definida en el fichero *stdio.h*. La función *printf* permite escribir en la pantalla; en este caso se escribe la cadena de caracteres indicada. En lenguaje C debemos finalizar cada instrucción con un punto y coma (*;*).

La quinta línea define cuál es el valor que devolverá la función. Como el tipo de retorno es un número entero (*int*), se ha de especificar un valor entero.

Finalmente, en la sexta línea, se cierra el código de la función con el símbolo *}*. El código de una función en C siempre se ha de cerrar entre los símbolos *{* y *}*.

4.1.1. Generación de un programa ejecutable

Para generar un programa ejecutable a partir de un fichero de código fuente C, utilizamos el compilador GCC.

Para compilar el programa anterior, hay que ejecutar la orden siguiente:

```
$ gcc hola.c -o hola -g
```

Para ejecutar el programa solo debemos utilizar el nombre del fichero de salida generado al añadir *./* delante:

```
$ ./hola
Hola!
$ _
```

4.2. Elementos de un programa en C

4.2.1. Directivas

En C existe un conjunto de directivas de compilación muy amplio. A continuación, se describen solo las directivas que utilizaremos. Las directivas empiezan siempre por el símbolo #.

1) ***include***: permite incluir otro fichero, de manera que cuando se llame el compilador, aquel sea compilado junto con el código fuente.

Lo más habitual es incluir los denominados *ficheros de cabecera* (*header*), ficheros con extensión *.h* con las definiciones de las funciones incluidas en las bibliotecas estándar de C, de manera que puedan ser utilizadas en el programa fuente.

El formato de la directiva *include* es:

```
#include <nombre_fichero>
```

Ejemplo

```
#include <stdio.h>
```

2) ***define***: permite definir valores constantes para ser utilizados en el programa. Los valores de las constantes se pueden expresar de maneras diferentes: como cadenas de caracteres o como valores numéricos expresados en binario, octal, decimal o hexadecimal.

El formato de la directiva *define* es:

```
#define nombre_constante valor
```

Ejemplo

```
#define CADENA "Hola"
#define NUMBIN 01110101b
#define NUMOCT 014q
#define NUMDEC 12
#define NUMHEX 0x0C
```

3) ***extern***: declara un símbolo como externo. Lo utilizamos si queremos acceder a un símbolo que no se encuentra definido en el fichero que estamos compilando, sino en otro fichero de código fuente C o código objeto.

En el proceso de compilación, cualquier símbolo declarado como externo no generará ningún error; sin embargo, durante el proceso de enlazamiento, si no existe un fichero de código objeto en el que este símbolo esté definido, producirá error.

La directiva tiene el formato siguiente:

```
extern [tipo_del_retorno] nombre_de_función ([lista_de_tipos]);
extern [tipo] nombre_variable [= valor_inicial];
```

En una misma directiva *extern* se pueden declarar tantos símbolos como se quiera, separados por comas.

Por ejemplo:

```
extern int y;
extern printVariable(int);
```

4.2.2. Variables

En general, las variables se pueden definir de dos maneras:

Identificadores

Los identificadores son los nombres que damos a las variables, constantes, funciones, etiquetas y otros objetos.

En C los identificadores han de empezar por una letra o el símbolo _.

También cabe tener presente que en C se distingue entre mayúsculas y minúsculas, por lo tanto, las variables siguientes son diferentes: VARX, varx, varX, VarX.

1) Globales: accesibles desde cualquier punto del programa.

2) Locales: accesibles solo dentro de un fragmento del programa.

La manera más habitual de definir variables locales es al principio de una función, de modo que solo existen durante la ejecución de una función y solo son accesibles dentro de esta.

Ejemplo

```
int x=0;                                    //x: variable global
int main(){
    int y=1;                                //y: variable local de la función main
     for (y=0; y>3; y++) {int z = y*2; }    //z: variable local del for
}
```

Para definir una variable se ha de especificar primero el tipo de dato de la variable seguido del nombre de la variable; opcionalmente se puede especificar a continuación el valor inicial.

El formato general para definir una variable en C es el siguiente:

```
tipo nombre_variable [= valor_inicial];
```

Los tipos de datos más habituales para definir variables son:

- caracteres (*char*),
- enteros (*int*) y (*long*) y
- números reales (*float*) y (*double*).

Ejemplo

```
/*Una variable de tipo carácter inicializada con el valor 'A'*/
char c='A';
/*Dos variables de tipo entero, una inicializada y otra sin
inicializar*/
int x=0, y;
/*Un número real, expresado en punto fijo*/
float pi=3.1416;
```

Tamaño de las variables

En la tabla siguiente se pueden observar los tipos de datos básicos en C, su tamaño en bytes, el rango de valores que pueden tomar y los tipos equivalente en ensamblador.

Tipo de variable en C	Tamaño	Rango de valores	Tipo ensamblador	Modificador
char	1 byte	–127 a +127	db	BYTE
unsigned char	1 byte	0 a 255	db	BYTE
short int	2 bytes	–32.768 a +32.767	dw	WORD
unsigned short int	2 bytes	0 a 65.535	dw	WORD
int	4 bytes	–2.147.483.648 a +2.147.483.647	dd	DWORD
unsigned int	4 bytes	0 a 4.294.967.295	dd	DWORD
long int	8 bytes	–9.223.372.036.854.775.808 a +9.223.372.036.854.775.807	dq	QWORD
unsigned long int	8 bytes	0 a 18.446.744.073.709.551.615	dq	QWORD

4.2.3. Operadores

Los operadores de C permiten realizar operaciones aritméticas, lógicas, relacionales (de comparación) y operaciones lógicas bit a bit. Cualquier combinación válida de operadores, variables y constantes se denomina **expresión**.

Cuando se define una expresión, podemos utilizar paréntesis para clarificar la manera de evaluar y cambiar la prioridad de los operadores.

Prioridad de los operadores

De más prioridad a menos prioridad:

```
! , ~
++, --
*, /, %
+, -
<<, >>
<, <=, >, >=
==, !=
&
^
|
&&
||
=
```

Podemos clasificar los operadores de la siguiente manera:

1) Operador de asignación:

Igualar dos expresiones: =

Ejemplos

```
a = 3;
c = a;
```

2) Operadores aritméticos:

suma: +
resta y negación: –
incremento: ++
decremento: – –
producto: *
división: /
resto de la división: %

Ejemplos

```
a = b + c;
x = y * z
```

3) Operadores de comparación relacionales:

igualdad: = =
diferente: !=
menor: <
menor o igual: <=
mayor: >
mayor o igual: >=

4) Operadores de comparación lógicos:

Y lógica (AND): &&
O lógica (OR): ||
Negación lógica: !

Ejemplos

```
(a = = b) && (c != 3)
(a <= b) || !(c > 10)
```

5) Operadores lógicos:

Operación lógica que se hace bit a bit.

OR (O): |
AND (Y): &
XOR (O exclusiva): ^
Negación lógica: ~
Desplazamiento a la derecha: >>
Desplazamiento a la izquierda: <<

Ejemplos

```
z = x | y;
z = x & y;
z = x ^ y;
z = x >> 2; // desplazar los bits de la variable x 2 posiciones
            // a la derecha y guardar el resultado en z
z = ~x; //z es el complemento a 1 de x
z = -x; //z es el complemento a 2 de x
```

4.2.4. Control de flujo

A continuación, se explican las estructuras de control de flujo más habituales de C:

1) Sentencias y bloques de sentencias. Una sentencia es una línea de código que finalizamos con un punto y coma (;). Podemos agrupar sentencias formando un bloque de sentencias utilizando las llaves ({}).

2) Sentencias condicionales

a) if. Es la sentencia condicional más simple; permite especificar una condición y el conjunto de sentencias que se ejecutarán en caso de que se cumpla la condición:

```
if (condición) {
      bloque de sentencias
}
```

Si solo hemos de ejecutar una sentencia, no es necesario utilizar las llaves:

```
if (condición) sentencia;
```

La condición será una expresión cuyo resultado de la evaluación sea 0 (falsa) o diferente de cero (cierta).

Ejemplos

```
if (a > b) {
    printf("a es mayor que b\n");
    a--;
}

if (a >= b) b++;
```

b) if-else. Permite añadir un conjunto de instrucciones que se ejecutarán en caso de que no se cumpla la condición:

```
if (condición) {
    bloque de sentencias
}
else {
    bloque de sentencias alternativas
}
```

Ejemplo

```
if (a >b) {
    printf("a es mayor que b\n");
    a--;

}
else
    printf(("a no es mayor que b\n");
```

c) if-else-if. Se pueden enlazar estructuras de tipo *if* añadiendo nuevas sentencias *if* a continuación.

```
if (condición 1) {
      bloque de sentencias que se ejecutan
      si se cumple la condición 1
}
else if (condición 2) {
      bloque de sentencias que se ejecutan
      si se cumple la condición 2 y
      no se cumple la condición 1
      }
      else {
         bloque de sentencias que se ejecutan
       si no se cumple ninguna de las condiciones
}
```

Ejemplo

```
if (a > b) {
    printf("a es mayor que b\n");
}
else if (a < b) {
        printf("a es menor que b\n");
     }
     else {
        printf("a es igual que b\n");
}
```

3) Estructuras iterativas

a) for. Permite hacer un bucle controlado por una o más variables que van desde un valor inicial hasta un valor final y que son actualizadas a cada iteración. El formato es:

```
for (valores iniciales; condiciones; actualización) {
      bloque de sentencias que ejecutar
}
```

Si solo hemos de ejecutar una sentencia, no hay que utilizar las llaves.

```
for (valores iniciales; condiciones; actualización) sentencia;
```

Ejemplo

```
//bucle que se ejecuta 5 veces, mientras x<5, cada vez se
incrementa x en 1
int x;
for (x = 0; x < 5 ; x++) {
     printf("el valor de x es: %d\n", x);
}
```

b) while. Permite expresar un bucle que se va repitiendo mientras se cumpla la condición indicada. Primero se comprueba la condición y si esta se cumple, se ejecutan las sentencias indicadas:

```
while (condiciones) {
     bloque de sentencias que ejecutar
}
```

Si solo hemos de ejecutar una sentencia no hay que utilizar las llaves.

```
while (condiciones) sentencia;
```

Ejemplo

```
//bucle que se ejecuta 5 veces, cabe definir previamente la variable x
int x = 0;
while (x < 5) {
   printf("el valor de x es: %d\n", x);
   x++; //necesario para salir del bucle
}
```

c) do-while. Permite expresar un bucle que se va repitiendo mientras se cumpla la condición indicada. Primero se ejecutan las sentencias indicadas y después se comprueba la condición; por lo tanto, como mínimo se ejecutan una vez las sentencias del bucle:

```
do {
   bloque de sentencias a ejecutar
}while (condiciones);
```

Si solo hemos de ejecutar una sentencia, no hay que utilizar las llaves:

```
do sentencia;
while (condiciones);
```

Ejemplo

```
//bucle que se ejecuta 5 veces, cabe definir previamente la variable x
int x = 0;
do {
   printf("el valor de x es: %d\n", x);
   x++;
} while (x < 5);
```

4.2.5. Vectores

Los vectores en lenguaje C se definen utilizando un tipo de dato base junto con el número de elementos del vector indicado entre los símbolos *[ ]*.

El formato para un vector unidimensional es el siguiente:

```
tipo nombre_vector [tamaño];
```

El formato para un vector bidimensional o matriz es el siguiente:

```
tipo nombre_vector [filas][columnas];
```

Siguiendo este mismo formato, se podrían definir vectores de más de dos dimensiones.

Al declarar un vector, se reserva el espacio de memoria necesario para poder almacenar el número de elementos del vector según el tipo.

Ejemplo

```
int vector[5]; // vector de 5 enteros
char cadena[4]; // vector de 4 caracteres
int matriz[3][4]; // matriz de 3 filas y 4 columnas: 12 enteros
```

También se pueden definir vectores dando un conjunto de valores iniciales; en este caso no es necesario indicar el número de elementos, si se hace no se podrá dar un conjunto de valores iniciales superior al valor indicado:

```
int vector[]={1, 2, 3, 4, 5};          // vector de 5 enteros
char cadena[4]={'H', 'o', 'l', 'a'};   // vector de 4 caracteres
int vector2[3]={1, 2};                 // vector de 3 enteros con las dos
                                       // primeras posiciones inicializadas
int vector2[3]={1, 2, 3, 4};           // declaración incorrecta
    int matriz[][]={{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
                                       // matriz de enteros de 3 filas y 4 columnas
```

Los vectores de tipo *char* también se pueden inicializar con una cadena de caracteres entre comillas dobles. Por ejemplo:

```
char cadena[]="Hola";
```

Para acceder a una posición del vector, debemos indicar el índice entre corchetes ([ ]). Cabe tener presente que el primer elemento de un vector se encuentra en la posición 0; por lo tanto, los índices irán desde 0 hasta el número de elementos menos uno.

Ejemplo

```
int vector[3]={1, 2, 3}; //índices válidos: 0, 1 y 2
                         //lo que equivale a hacer lo siguiente:
   int vector[3];
   vector[0]=1;
   vector[1]=2;
   vector[2]=3;
                         // o también se puede hacer lo siguiente:
int i;
for (i=0; i<3; i++) vector[i]=i+1;
                         // Si hacemos lo siguiente, no se producirá error, pero realmente
                         // estaremos accediendo fuera del vector, lo que puede ocasionar
                         // múltiples problemas:
   vector[3]=4;
   vector[20]=300;
```

4.2.6. Apuntadores

Un aspecto muy importante del lenguaje C es que permite acceder directamente a direcciones de memoria. Esto se consigue utilizando un tipo de variable que se conoce como *apuntador a memoria*. Un apuntador es una variable que contiene una dirección de memoria.

Un apuntador se define indicando el tipo de dato al cual apunta y añadiendo el símbolo * ante el nombre del apuntador.

Cabe tener presente que al definir un apuntador solo se reserva espacio para almacenar una dirección de memoria; no se reserva espacio para poder almacenar un valor de un tipo determinado.

Símbolo * en C

El símbolo * tiene múltiples funciones: definir un apuntador, acceder al contenido de una posición de memoria y también se corresponde con el operador aritmético de multiplicación.

El formato general para definir un apuntador en C es el siguiente:

```
tipo *nombre_apuntador;
```

Utilizamos el operador & para obtener la dirección de memoria donde se encuentra almacenada una variable, no su contenido. Esta dirección la podemos asignar a un apuntador.

Ejemplo

```
int   *p1;    // Se define un apuntador de tipo int
char *p2;     // Se define un apuntador de tipo char

int x=1, y;   // se definen dos variables de tipo int
char c='a';   // se define una variables de tipo char

p1=&x;        // se asigna a p1 la dirección de la variable x
p2=&c;        // se asigna a p2 la dirección de la variable c
y=*p1;        // como p1 apunta a x, es equivalente a y = x.
```

Cuando trabajamos con vectores, el nombre del vector en realidad es un apuntador, una constante que contiene la dirección del primer elemento del vector.

```
   int vector[3]={1, 2, 3};
                   //lo que es equivalente a hacer lo siguiente:
   int vector[3];
   int *v;
   int i;
   v = vector;      //vector i *v son apuntadores a entero.
for (i=0; i<3; i++) *(v+i)=i+1;
                   // *(v+i): indica que estamos accediendo al contenido
                   // de la dirección 'v+i'
```

4.2.7. Funciones

Una función es un bloque de código que lleva a cabo una tarea concreta. Aparte de la función *main*, un programa puede definir e implementar otras funciones.

El formato general de una función es el siguiente:

```
tipo_del_retorno nombre_función(lista_de_parámetros){
    definición de variables;
    sentencias;
    return valor;
}
```

donde:

- `tipo_del_retorno`: tipo del dato que devolverá la función; si no se especifica el tipo de retorno, por defecto es de tipo entero (*int*).
- `lista_de_parámetros`: lista de tipos y nombres de variables separados por comas; no es obligatorio que la función tenga parámetros.
- `return`: finaliza la ejecución de la función y devuelve un valor del tipo especificado en el campo `tipo_del_retorno`. Si no se utiliza esta sentencia o no se especifica un valor, se devolverá un valor indeterminado.

Ejemplo

Definamos una función que sume dos enteros y devuelva el resultado de la suma:

```
int funcioSuma(int a, int b){
    int resultado;     //variable local

    resultado = a + b; //sentencia de la función
    return resultado;  //valor de retorno
}
```

Se muestra a continuación cómo quedaría un programa completo que utilizara esta función.

```
// fichero suma.c
#include <stdio.h>

int funcioSuma(int a, int b){
    int resultado;     //variable local

    resultado = a + b; //sentencia de la función
    return resultado;  //valor de retorno
}

int main(){
  int x, y, r;         //variables locales
  printf ("\nIntroduce el valor de x: ");
  scanf("%d",&x);
  printf ("Introduce el valor de y: ");
  scanf("%d",&y);
  r=funcioSuma(x,y);   //llamamos a la función que hemos definido
  printf("La suma de x e y es: %d\n", r);
}
```

Compilación y ejecución

```
$ gcc -o suma suma.c
$ ./suma
Introduce el valor de x: 3
Introduce el valor de y: 5
La suma de x e y es: 8
$ _
```

4.2.8. Funciones de E/S

Se describen a continuación las funciones básicas de E/S, para escribir en pantalla y para leer por teclado y que están definidas en el fichero *stdio.h*:

Nota

El repertorio de funciones de E/S es muy extenso, podréis encontrar mucha información de todas estas funciones en libros de programación en C y en Internet.

Función *printf*

printf permite escribir en pantalla información formateada, permite visualizar cadenas de caracteres constantes, junto con el valor de variables.

El formato general de la función es el siguiente:

```
printf("cadena de control"[, lista_de_parámetros])
```

La cadena de control incluye dos tipos de elementos:

- Los caracteres que queremos mostrar por pantalla.
- Órdenes de formato que indican cómo se mostrarán los parámetros.

La lista de parámetros está formada por un conjunto de elementos que pueden ser expresiones, constantes y variables, separadas por comas.

Debe existir el mismo número de órdenes de formato que de parámetros, se deben corresponder en orden y el tipo de la orden con el tipo del dato.

Las órdenes de formato empiezan con un símbolo % seguido de uno o más caracteres. Las más habituales son:

`%d`	Para mostrar un valor entero, el valor de una variable de tipo *char* o *int*.
`%ld`	Para mostrar un valor entero largo, el valor de una variable de tipo *long*.
`%c`	Para mostrar un carácter, el contenido de una variable de tipo *char*.

`%s`	Para mostrar una cadena de caracteres, el contenido de un vector de tipo *char*.
`%f`	Para mostrar un número real, el valor de una variable de tipo *float* o *double*. Se puede indicar el número de dígitos de la parte entera y de la parte decimal, utilizando la expresión siguiente: `%[dígitos enteros].[dígitos decimales]f`
`%p`	Para mostrar una dirección de memoria, por ejemplo el valor de un apuntador.

Dentro de la cadena de control se pueden incluir algunos caracteres especiales, empezando con un símbolo \ y seguido de un carácter o más. Los más habituales son:

`\n`	carácter de salto de línea
`\t`	carácter de tabulación

Si solo se quiere mostrar una cadena de caracteres constantes, no es necesario especificar ningún parámetro.

Ejemplos

```
int x=5;
float pi=3.1416;
char msg[]="Hola", c='A';
printf("Hola!" );
printf ("Valor de x: %d. Dirección de x: %p\n", x, &x);
printf ("El valor de PI con 1 entero y 2 decimales: %1.2f\n", pi);
printf ("Contenido de msg: %s. Dirección de msg: %p\n", msg, msg);
printf ("Valor de c: %c. El código ASCII guardado en c: %d\n", c, c);
printf ("Constante entera: %d, y una letra: %c", 100, 'A');
```

Función *scanf*

scanf permite leer valores introducidos por teclado y almacenarlos en variables de memoria.

La lectura de un dato acaba cuando se encuentra un carácter de espacio en blanco, un tabulador o un ENTER. Si la llamada a la función solicita la lectura de varias variables, estos caracteres tienen la función de separadores de campo, pero habrá que finalizar la lectura con un ENTER.

El formato general de la función es:

```
scanf("cadena de control", lista_de_parámetros)
```

La cadena de control está formada por un conjunto de órdenes de formato. Las órdenes de formato son las mismas definidas para la función *printf*.

La lista de parámetros debe estar formada por direcciones de variables en las que se guardarán los valores leídos; las direcciones se separan por comas.

Para indicar la dirección de una variable, se utiliza el operador & delante del nombre de la variable. Recordad que el nombre de un vector se refiere a la dirección del primer elemento del vector, por lo tanto no es necesario el operador &.

Ejemplos

```
int x;
float f;
char msg[5];

scanf("%d %f", &x, &f); // se lee un entero y un real
                        // si escribimos: 2 3.5 y se aprieta ENTER, se asignará x=2 y f=3.5
                        // hará lo mismo si hacemos: 2 y se aprieta ENTER, 3.5 y se
                        // aprieta ENTER
scanf("%s", msg);       // Al leer una cadena se añade un \0 al final
```

5. Conceptos de programación en ensamblador y C

5.1. 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 otras.

Ejemplo

```
.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.

Por una parte, el acceso a los datos es muy flexible, pero, por otra parte, si no controlamos muy bien el acceso a las variables, esta flexibilidad nos puede causar ciertos problemas.

En lenguaje C tenemos un comportamiento parecido. Supongamos que tenemos el código siguiente:

```
int vec[4]={1,2,3,4};
int x=258; //258=00000102h
char c=3;

int main() {
   int i=0;
   for (i=0;i<4;i++) printf("contenido de vec", vec[i]);
   c = x;
}
```

El índice `i` toma los valores 0, 1, 2, 3 y 4, el índice `i=4` no corresponde a una posición del vector; la primera posición es la 0 y la última es la 3; por lo tanto, estamos accediendo a una posición fuera del vector. Al compilar el código, no se generará ningún error y tampoco se producirá ningún error durante la ejecución. Si la variable `x` ocupara la posición de memoria siguiente a continuación de los 4 elementos del vector, se mostraría un 258.

En la asignación `c = x` como los tipos son diferentes, `x` (int de 4 bytes) y `c` (char de 1 byte), asigna el byte menos significativo de `x` a la variable `c`, después de la asignación `c = 2`. Si hiciéramos la asignación al revés, `x = c`, haría la extensión de signo de `c` a 4 bytes y, por lo tanto, después de la asignación `x = 3 (00000003h)`.

A diferencia del ensamblador, en el que las variables se encuentran en la memoria en el mismo orden en el que las declaramos, en C no podemos asegurarlo, ya que la reserva de espacio de memoria para las variables que hace el compilador GCC es diferente.

En ensamblador podemos acceder a variables globales definidas en un código escrito en C si compilamos conjuntamente el código C y el código ensamblador. Para acceder a una variable de C solo hay que declararla en ensamblador con la directiva *extern* (podéis ver el apartado 3.2.3).

Por ejemplo, supongamos que tenemos definidas las siguientes variables en C:

```
char vec_char[4];
int vec_int[4];
long int l_int;
short int s_int;
```

Para poder acceder a estas variables desde ensamblador añadiríamos lo siguiente:

```
;Variables definidas en C que se utilizan en ensamblador
extern vec_char, vec_int, l_int, s_int
```

A la hora de acceder a las variables definidas en C hay que tener presente su tamaño (podéis ver "tamaño de los operandos" al final del apartado 3.4.1 y "tamaño de las variables" al final del apartado 4.2.2).

5.1.1. Estructuras de datos

Veamos cómo acceder a vectores y matrices utilizando lenguaje C y lenguaje de ensamblador.

Los modos de direccionamiento que habitualmente se utilizan para acceder a vectores y matrices son el direccionamiento indirecto, relativo, indexado y la combinación de relativo e indexado.

Para acceder a cada elemento del vector o de la matriz hay que tener presente el tamaño del tipo de dato utilizado para definirlos, tanto para obtener el dato de un elemento como para acceder al elemento siguiente.

En vectores de tipo `char`, asignaremos el valor de cada dato a registros de tamaño 1 byte y para pasar al elemento siguiente habrá que hacer incrementos de una unidad. En tipo `int`, asignaremos el valor a registros de tamaño 4 bytes y habrá que hacer incrementos de 4 unidades.

Veamos algunos ejemplos de cómo acceder a vectores definidos en C desde ensamblador utilizando los modos de direccionamiento comentados anteriormente.

Utilizaremos el vector `vec_char` de tipo *char*, donde cada elemento ocupa 1 byte (1 posición de memoria), y el vector `vec_int` de tipo *int*, donde cada elemento ocupa 4 bytes (4 posiciones de memoria).

```
;direccionamiento indirecto
mov edx, vec_int             ;Tomamos la dirección del vector
mov DWORD [edx], 0           ;Ponemos a 0 el primer elemento del vector vec_int
add edx, 4                   ;Sumamos 4 para acceder al siguiente elemento del vector
add DWORD [edx], 0           ;Ponemos a 0 el segundo elemento del vector vec_int
add edx, 4                   ;Sumamos 4 para acceder al siguiente elemento del vector
...

;direccionamiento relativo
mov al, byte [vec_char+0]    ;primer elemento del vector, vec_char[0]
mov bl, byte [vec_char+1]    ;segundo elemento del vector, vec_char[1]

mov eax, dword [vec_int+0]   ;primer elemento del vector, vec_int[0]
mov ebx, dword [vec_int+12]  ;cuarto elemento del vector, vec_int[3]

;direccionamiento indexado
   mov esi, 0                ;Inicializamos el índice para acceder al vector.
  inicializa:
mov dword [vec_int+esi],0    ;Ponemos a 0 un elemento del vector
   add esi, 4                ;Sumamos 4 para acceder al siguiente elemento del vector
   cmp esi, 12               ;Accedemos a vec_int[0], vec_int[1]
   jle inicializa            ;vec_int[2] y vec_int[3]

;combinación indexado y relativo
mov edx, vec_int             ;Tomamos la dirección del vector
mov esi, 0                   ;Inicializamos el índice para acceder al vector.
  inicializa:
mov dword [edx+esi*4],0      ;Ponemos a 0 un elemento del vector
   add esi, 1                ;Sumamos 1 para acceder al siguiente elemento del
                             ;vector porque utilizamos el factor de escala 4
    cmp esi, 3               ;Accedemos a vec_int[0], vec_int[1]
   jle inicializa            ;vec_int[2] y vec_int[3]
```

Definimos en C un vector y una matriz y hacemos la suma de los elementos.

```
int main(){

   int vec[6]={1,2,3,4,5,6},mat[2][3]={{1,2,3},{4,5,6}};
   int i,j, sumaVec=0, sumaMat=0;

   for (i=0;i<6;i++) sumaVec=sumaVec+vec[i];

   for (i=0;i<2;i++){
   for(j=0;j<3;j++) sumaMat=sumaMat+mat[i][j];
  }
}
```

A continuación, veremos cómo hacer lo mismo con lenguaje de ensamblador, utilizando diferentes variantes del direccionamiento indexado (el formato general es `[dirección + Registro Base + Registro Índice * escala])`:

```
 vec dd 1,2,3,4,5,6
  mat dd 1,2,3
     dd 4,5,6

                                ;recorremos el vector para sumarlo
  mov esi,0                     ;esi será el índice para acceder a los datos
  mov eax,0                     ;eax será donde guardaremos la suma
loop_vec:
  add eax, dword[vec+esi*4]     ;multiplican el índice por 4
  inc esi                       ;porque cada elemento ocupa 4 bytes.
  cmp esi, 6                    ;comparamos con 6 porque es el índice del primer
                                ;elemento fuera del vector.
  jl loop_vec

                                ;recorremos la matriz con un solo índice para sumarla
  mov esi,0                     ;esi será el índice para acceder a los datos
  mov ebx,0                     ;ebx será donde guardaremos la suma
loop_mat:
  add ebx, dword[mat+esi*4]     ;multiplican el índice por 4
  inc esi                       ;porque cada elemento ocupa 4 bytes.
  cmp esi, 6                    ;comparamos con 6 porque es el índice del primer elemento
                                ;fuera de la matriz, la matriz tiene 2*3=6 elementos.
jl loop_mat

                                ;recorremos la matriz con dos índices para sumarla
  mov esi,0                     ;esi será el índice de la columna
  mov edi,0                     ;edi será el índice de la fila
  mov ebx,0                     ;ebx será donde guardaremos la suma
loop_mat2:
  add ebx, dword[mat+edi+esi*4] ;multiplican el índice de la columna
  inc esi                       ;por 4 porque cada elemento ocupa 4 bytes.
  cmp esi, 3                    ;comparamos con 3 porque son los elementos de una fila.
  jl loop_mat2
  mov esi, 0
  add edi, 12                   ;cada fila ocupa 12 bytes, 3 elementos de 4 bytes.
  cmp edi, 24                   ;comparamos con 24 porque es el primer índice
  jl loop_mat2                  ; fuera de la matriz.
```

Como podemos comprobar, el código para acceder al vector y a la matriz utilizando un índice son idénticos; en ensamblador las matrices se ven como vectores y todas las posiciones de memoria son consecutivas; por lo tanto, si queremos acceder a una posición concreta de una matriz, a partir de un nú-

mero de fila y de columna, deberemos calcular a qué posición de memoria hay que acceder; fila × elementos_de_la_fila × tamaño_del_dato + columna × tamaño_del_dato.

Por ejemplo, si queremos acceder al elemento mat[1][2], segunda fila, tercera columna (las filas y columnas se empiezan a numerar por 0), este elemento estará en la posición 1 × 3 × 4 + 2 × 4 = 20, [mat+20].

Recordemos que el ensamblador nos ofrece diferentes modos de direccionamiento que nos pueden facilitar el acceso a determinadas estructuras de datos, tales como los vectores y las matrices.

En el ejemplo anterior, en el que se utilizan dos índices para acceder a la matriz, un índice (edi) lo utilizamos para la fila y el otro índice (esi) para la columna, de manera que el recorrido por la matriz es el siguiente:

```
edi = 0
[mat+0+0*4]    [mat+0+1*4]    [mat+0+2*4]
edi = 12
[mat+12+0*4]  [mat+12+1*4]  [mat+12+2*4]
```

5.1.2. 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.

Ejemplo

```
  Supongamos que rax = 0102030405060708h (registro de 64 bits),
bx = 0A0Bh (registro de 16 bits) y rsp = 0000000010203050h.

push rax
push bx
pop bx
pop rax
```

Evolución de la pila al ejecutar estas cuatro instrucciones. En la tabla se muestra el estado de la pila después de ejecutar cada instrucción.

Dirección de memoria	Estado inicial	push rax	push bx	pop bx	pop rax
10203044h					
10203045h					
10203046h			**0B**		
10203047h			**0A**		
10203048h		**08**	08	08	
10203049h		**07**	07	07	
1020304Ah		**06**	06	06	
1020304Bh		**05**	05	05	
1020304Ch		**04**	04	04	
1020304Dh		**03**	03	03	
1020304Eh		**02**	02	02	
1020304Fh		**01**	01	01	
10203050h	xx	xx	xx	xx	xx
RSP	10203050h	10203048h	10203046h	10203048h	10203050h

- `push rax`. Decrementa RSP en 8 unidades y traslada el valor del registro RAX a partir de la posición de memoria indicada por RSP; funcionalmente la instrucción anterior sería equivalente a:

```
sub rsp, 8
mov qword[rsp], rax
```

- `push bx`. Decrementa RSP en 2 unidades y traslada el valor del registro BX a partir de la posición de memoria indicada por RSP; funcionalmente la instrucción anterior sería equivalente a:

```
sub rsp, 2
mov word[rsp], bx
```

- `pop bx`. Traslada hacia BX el valor de 2 bytes almacenado a partir de la posición de memoria indicada por RSP, a continuación se incrementa RSP en 2. Sería equivalente a efectuar:

```
mov bx, word [rsp]
add rsp, 2
```

- `pop rax`. Traslada hacia RAX el valor de 8 bytes almacenado a partir de la posición de memoria indicada por RSP, a continuación se incrementa RSP en 8. Sería equivalente a efectuar:

```
mov rax, qword [rsp]
add rsp, 8
```

5.2. Operaciones aritméticas

En este subapartado hemos de tener presente que en C podemos construir expresiones utilizando variables, constantes y operadores en una misma sentencia; en cambio, hacer lo mismo en ensamblador implica escribir una secuencia de instrucciones en la que habrá que hacer las operaciones una a una según la prioridad de los operadores dentro de la sentencia.

Ejemplo

Sentencia que podemos expresar en lenguaje C:

```
r=(a+b)*4 / (c>>2);
```

Traducción de la sentencia en lenguaje de ensamblador:

```
mov eax [a]
mov ebx, [b]
add eax, ebx  ;(a+b)
imul eax, 4   ;(a+b)*4
mov ecx,[c]
sar ecx, 2    ;(c>>2)
idiv ecx      ;(a+b)*4 / (c>>2)
mov [r], eax  ;r=(a+b)*4 / (c>>2)
```

5.3. Control de flujo

En este subapartado veremos cómo las diferentes estructuras de control del lenguaje C se pueden traducir a lenguaje de ensamblador.

5.3.1. Estructura *if*

Estructura condicional expresada en lenguaje C:

```
if (condición) {
      bloque de sentencias
}
```

Ejemplo

```
if (a > b) {
     maxA = 1;
     maxB = 0;
}
```

Se puede traducir a lenguaje de ensamblador de la manera siguiente:

```
mov rax, qword [a]     ;Se cargan las variables en registros
mov rbx, qword [b]

cmp rax, rbx           ;Se hace la comparación
jg cierto              ;Si se cumple la condición, salta a la etiqueta cierto
jmp fin                ;Si no se cumple la condición, salta a la etiqueta fin

cierto:
    mov byte [maxA], 1 ;Estas instrucciones solo se ejecutan
    mov byte [maxB], 0 ;cuando se cumple la condición

fin:
```

El código se puede optimizar si se utiliza la condición contraria a la que aparece en el código C (a <= b).

```
    mov rax, qword [a] ;Se carga solo la variable a en un registro
    cmp rax, qword [b] ;Se hace la comparación
    jle fin            ;Si se cumple la condición (a<=b) salta a fin

    mov byte [maxA], 1 ;Estas instrucciones solo se ejecutan
    mov byte [maxB], 0 ;cuando se cumple la condición (a > b)
fin:
```

Si tenemos una condición más compleja, primero habrá que evaluar la expresión de la condición para decidir si ejecutamos o no el bloque de sentencias de *if*.

Ejemplo

```
if ((a != b) || (a>=1 && a<=5)) {
    b = a;
}
```

Se puede traducir a lenguaje de ensamblador de esta manera:

```
mov rax, qword [a]     ;Se cargan las variables en registros
mov rbx, qword [b]

cmp rax, rbx           ;Se hace la comparación (a != b)
jne cierto             ;Si se cumple la condición salta a la etiqueta cierto
                       ;Si no se cumple, como es una OR, se puede cumplir la otra condición.
cmp rax, 1             ;Se hace la comparación (a >= 1), si no se cumple,
jl fin                 ;como es una AND, no hay que mirar la otra condición.
cmp rax, 5             ;Se hace la comparación (a <= 5)
jg fin                 ;Si no salta, se cumple que (a>=1 && a<=5)

cierto:
    mov qword [b], rax ;Hacemos la asignación cuando la condición
es cierta.

fin:
```

5.3.2. Estructura *if-else*

Estructura condicional expresada en lenguaje C considerando la condición alternativa:

```
if (condición) {
    bloque de sentencias
}
else {
    bloque de sentencias alternativas
}
```

Ejemplo

```
if (a > b) {
    max = 'a';
}
else { // (a <= b)
    max = 'b';
}
```

Se puede traducir a lenguaje de ensamblador de la siguiente manera:

```
   mov rax, qword [a]  ;Se cargan las variables en registros
   mov rbx, qword [b]
   cmp rax, rbx        ;Se hace la comparación
   jg cierto           ;Si se cumple la condición, se salta a cierto

   mov byte [max], 'b' ;else (a <= b)
   jmp fin

cierto:
   mov byte [max], 'a' ;if (a > b)

fin:
```

5.3.3. Estructura *while*

Estructura iterativa controlada por una condición expresada al principio:

```
while (condiciones) {
    bloque de sentencias que ejecutar
}
```

Ejemplo

```
resultado=1;
while (num > 1){ //mientras num sea mayor que 1 hacer ...
    resultado = resultado * num;
    num--;
}
```

Se puede traducir a lenguaje de ensamblador de esta manera:

```
  mov rax, 1           ;rax será [resultado]
  mov rbx, qword [num] ;Se carga la variable en un registro
while:
  cmp rbx, 1           ;Se hace la comparación
  jg cierto            ;Si se cumple la condición (num > 1) salta a cierto
  jmp fin              ;Si no, salta a fin
cierto:
  imul rax, rbx        ;rax = rax * rbx
  dec rbx
  jmp while
fin:
  mov qword [resultado], rax
  mov qword [num], rbx
```

Una versión alternativa que utiliza la condición contraria (num <= 0) y trabaja directamente con una variable de memoria:

```
  mov rax, 1            ;rax será [resultado]
while:
  cmp qword [num], 1    ;Se hace la comparación
  jle fi                ;Si se cumple la condición (num <= 1) salta a fin
  imul rax, qword [num] ;rax=rax*[num]
  dec qword [num]
  jmp while
fin:
  mov qword [resultado], rax
```

En este caso, reducimos tres instrucciones del código, pero aumentamos los accesos a memoria durante la ejecución del bucle.

5.3.4. Estructura *do-while*

Estructura iterativa controlada por una condición expresada al final:

```
do {
   bloque de sentencias que ejecutar
} while (condiciones);
```

Ejemplo

```
resultado = 1;
do {
   resultado = resultado * num;
   num--;
} while (num > 1)
```

Se puede traducir a lenguaje de ensamblador de la siguiente manera:

```
  mov rax, 1            ;rax será [resultado]
  mov rbx, qword [num]  ;Se carga la variable en un registro
while:
  imul rax, rbx
  dec rbx
  cmp rbx, 1            ;Se hace la comparación
  jg while              ;Si se cumple la condición salta a while

  mov qword [resultado], rax
  mov qword [num], rbx
```

5.3.5. Estructura *for*

Estructura iterativa, que utiliza la orden *for*:

```
for (valores iniciales; condiciones; actualización) {
    bloque de sentencias que ejecutar
}
```

Ejemplo

```
resultado=1;
for (i = num; i > 1; i--)
{
   resultado=resultado*i;
}
```

Se puede traducir a lenguaje de ensamblador de esta manera:

```
  mov rax, 1            ;rax será [resultado]
  mov rcx, qword [num]  ;rcx será [i] que inicializamos con [num]

for:
  cmp rcx, 1            ;Se hace la comparación
  jg cierto             ;Si se cumple la condición, salta a cierto
  jmp fin
cierto:
  imul rax,rcx
  dec rcx
  jmp for
fin:
  mov qword [resultado], rax
  mov qword [i], rcx
```

Al salir de la estructura iterativa, no actualizamos la variable *num* porque se utiliza para inicializar *i* pero no se cambia.

5.3.6. Estructura switch-case

Estructura de selección, que amplía el número de opciones de la estructura if-else y permite múltiples opciones. Se utiliza una variable que se compara por igualdad sucesivamente con diferentes valores constantes, se permite indicar una o varias sentencias por cada caso, finalizadas por la sentencia *break*, y especificar una o varias sentencias que se ejecutan en caso de no coincidir con ninguno de los valores indicados, *default*.

```
switch (variable) {
  case valor1:   sentencia1;
                 break;
```

```
  case valor2:   sentencia2;
                 ...
                 break;
                 ...
  default:       sentenciaN;
}
```

Ejemplo

```
switch (var) {
  case 1:   a=a+b;
            break;
  case 2:   a=a-b;
            break;
  case 3:   a=a*b;
            break;
  default:  a=-a;
}
```

Se puede traducir a lenguaje ensamblador de la manera siguiente:

```
switch:  mov rax, qword [var]
         cmp rax, 1
         jne case2
         mov rbx, qword [b]
         add qword [a], rbx
         jmp end_s...
case2:   cmp rax, 2
         jne case3
         mov rbx, qword [b]
         sub qword [a], rbx
         jmp end_s
case3:   cmp rax, 3
         jne default
         mov rbx, qword [b]
         mul qword [a], rbx
         jmp end_s
default: neg qword [a]
end_s:
```

5.4. Subrutinas y paso de parámetros

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. En este apartado veremos cómo definir subrutinas en ensamblador y cómo las podemos utilizar después desde un programa en C.

Primero describiremos cómo trabajar con subrutinas en ensamblador:

- Definición de subrutinas en ensamblador.
- Llamada y retorno de subrutina.
- Paso de parámetros a la subrutina y retorno de resultados.

A continuación veremos cómo hacer llamadas a subrutinas hechas en ensamblador desde código ensamblador y desde código C y qué implicaciones tiene en el paso de parámetros.

5.4.1. Definición de subrutinas en ensamblador

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*).

La estructura básica de una subrutina sería:

```
subrutina:
   ;
   ; Instrucciones de la subrutina
   ;
   ret
```

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*.

```
subrutina:
   ; Almacenar en la pila
   ; los registros modificados dentro de la subrutina.
   ;
   ; Instrucciones de la subrutina.
   ;
   ; Restaurar el estado de los registros modificados
   ; recuperando su valor inicial almacenado en la pila.
   ret
```

Ejemplo de subrutina que calcula el factorial de un número

```
factorial:
    push rax                ; Almacenamos en la pila los registros
    push rbx                ; modificados dentro de la subrutina

                            ; Instrucciones de la subrutina
    mov rax, 1              ; rax será el resultado
    mov rbx, 5              ; Calculamos el factorial del valor de rbx (=5).
  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 5 (=120)
    mov qword[result], rax  ; Almacenamos el resultado en la variable 'result'

    pop rbx                 ; Restauramos el valor inicial de los registros
    pop rax                 ; en orden inverso a como los hemos almacenado

    ret                     ; Finaliza la ejecución de la subrutina
```

5.4.2. Llamada y retorno de subrutina en ensamblador

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:

```
call factorial
```

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.

Funcionalmente, la instrucción *call* anterior sería equivalente a:

```
sub rsp, 8
mov qword[rsp], rip
mov rip, factorial
```

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*.

Funcionalmente, la instrucción *ret* sería equivalente a:

```
mov rip, qword[rsp]
add rsp, 8
```

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:

```
push rbp       ; Almacenar el registro rbp en la pila
mov rbp, rsp   ; Asignar a rbp el valor del registro apuntador rsp

...

mov rsp, rbp   ; Restauramos el valor inicial de rsp
pop rbp        ; Restauramos el valor inicial de rbp

ret
```

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*.

Ejemplo de llamada de una subrutina en ensamblador

```
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 rax                   ; Almacenamos en la pila los registros
    push rbx                   ; modificados dentro de la subrutina

    mov rax, 1                 ; rax será el resultado
    mov rbx, qword[x]          ; Calculamos el factorial de la variable 'x' (=5).
  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 'x' (=120)
    mov qword[result], rax     ; Almacenamos el resultado en la variable 'result'

    pop rbx                    ; Restauramos el valor inicial de los registros
    pop rax                    ; en orden inverso a como los hemos almacenado

    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
```

5.4.3. Paso de parámetros a la subrutina y retorno de resultados

Una subrutina puede necesitar que se le transfieran parámetros; los parámetros se pueden pasar mediante registros o la pila. Sucede lo mismo con el retorno de resultados, que puede efectuarse por medio de registro o de la pila. Consideraremos los casos en los que el número de parámetros de entrada y de retorno de una subrutina es fijo.

Paso de parámetros y retorno de resultado por medio de registros

Debemos definir sobre qué registros concretos queremos pasar parámetros a la subrutina y sobre qué registros haremos el retorno; podemos utilizar cualquier registro de propósito general del procesador.

Una vez definidos los registros que utilizaremos para hacer el paso de parámetros, deberemos asignar a cada uno el valor que queremos pasar a la subrutina antes de hacer *call*; para devolver los valores, dentro de la subrutina, tendremos que asignar a los registros correspondientes el valor que se debe devolver antes de hacer *ret*.

Recordemos que los registros que se utilicen para devolver un valor no se han de almacenar en la pila al inicio de la subrutina, ya que no hay que conservar el valor inicial.

Supongamos que en el ejemplo del factorial queremos pasar como parámetro un número cuyo factorial queremos calcular, y devolver como resultado el factorial del número transferido como parámetro, implementando el paso de parámetros y el retorno de resultados por medio de registros.

El número cuyo factorial queremos calcular lo pasaremos por medio del registro RBX y devolveremos el resultado al registro RAX.

La llamada de la subrutina será:

```
mov rbx, qword[x]       ; Ponemos en el registro rbx el valor de la variable 'x'
                        ; como parámetro de entrada
call factorial          ; Llamamos a la subrutina factorial
                        ; En rax tendremos el valor del factorial de 'x' como
mov qword[result],rax   ; parámetro de salida y lo asignamos a la variable 'result'
```

Subrutina:

```
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      ; rax será el resultado
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
```

Paso de parámetros y retorno de resultado por medio de la pila

Si queremos pasar parámetros y devolver resultados a una subrutina utilizando la pila, y una vez definidos los parámetros que queremos pasar y los que queremos retornar, hay que hacer lo siguiente:

1) Antes de hacer la llamada a la subrutina: es necesario reservar espacio en la pila para los datos que queremos devolver y a continuación introducir los parámetros necesarios en la pila.

2) Dentro de la subrutina: hay que acceder a los parámetros leyéndolos directamente de memoria, utilizando un registro que apunte a la cima de la pila.

El registro apuntador de pila, RSP, siempre apunta a la cima de la pila y, por lo tanto, podemos acceder al contenido de la pila haciendo un direccionamiento a memoria que utilice RSP, pero si utilizamos la pila dentro de la subrutina, no se recomienda utilizarlo.

El registro que se suele utilizar como apuntador para acceder a la pila es el registro RBP. Antes de utilizarlo, lo tendremos que almacenar en la pila para poder recuperar el valor inicial al final de la subrutina, a continuación se carga en RBP el valor de RSP.

RBP no se debe cambiar dentro de la subrutina; al final de esta se copia el valor sobre RSP para restaurar el valor inicial.

```
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

...

mov rsp, rbp  ; Restauramos el valor inicial de rsp con rbp
pop rbp       ; Restauramos el valor inicial de rbp
```

3) Después de ejecutar la subrutina: una vez fuera de la subrutina es necesario liberar el espacio utilizado por los parámetros de entrada y después recuperar los resultados del espacio que hemos reservado antes de hacer la llamada.

Supongamos que en el ejemplo del factorial queremos pasar como parámetro el número cuyo factorial queremos calcular y devolver como resultado el factorial del número pasado como parámetro, implementando el paso de parámetros y el retorno de resultados por medio de la pila.

La llamada de la subrutina:

```
sub rsp,8           ; Reservamos 8 bytes para el resultado que retornamos
push qword [x]      ; Introducimos como parámetro de entrada a la pila
                    ; el valor de la variable 'x'
call factorial      ; Llamamos a la subrutina factorial
                    ; En la pila tendremos el valor del factorial de 'x'
add rsp,8           ; Liberamos el espacio utilizado por el parámetro de entrada
pop qword[result]   ; Recuperamos el resultado retornado sobre la variable 'result'
```

Subrutina:

```
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 rax          ; Almacenar en la pila los registros que
  push rbx          ; modificamos dentro de la subrutina
                    ; Instrucciones de la subrutina
  mov rax, 1        ; rax será el resultado
  mov rbx, [rbp+16]; [rbp+16] referencia al parámetro de entrada
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
  mov [rbp+24], rax; [rbp+24] espacio que hemos reservado en la pila
                    ; para retornar el resultado
  pop rbx           ; Restauramos el valor inicial del registro
  pop rax
  mov rsp, rbp      ; Restauramos el valor inicial de rsp con rbp
  pop rbp           ; Restauramos el valor inicial de rbp
  ret
```

Cabe recordar que la memoria se direcciona byte a byte; cada dirección corresponde a una posición de memoria de un byte. En este ejemplo, ya que los elementos que ponemos en la pila son de 8 bytes (64 bits), para pasar de un dato al siguiente, tendremos que hacer incrementos de 8 bytes.

Evolución de la pila al ejecutar este código. La tabla muestra el estado de la pila después de ejecutar cada instrucción o conjunto de instrucciones. La dirección inicial de la pila es @ y consideramos que la variable *x* vale 5, y por lo tanto el factorial de 5 es 120.

Dirección	Estado inicial		sub rsp,8		push qword[x]		call factorial	
	apuntadores	pila	apuntadores	pila	apuntadores	pila	apuntadores	pila
@ - 48								
@ - 40								
@ - 32								
@ - 24							rsp→	@retorn
@ - 16					rsp→	5		5
@ - 8			rsp→	----		----		----
@	rsp→	----		----		----		----

	push rbp mov rbp, rsp		push rax push rbx		mov rbx, [rbp+16]		mov [rbp+24], rax	
	apuntadores	pila	apuntadores	pila	apuntadores	pila	apuntadores	pila
@ - 48			rsp→	rbx	rsp→	rbx	rsp→	rbx
@ - 40				rax		rax		rax
@ - 32	rsp, rbp→	rbp	rbp→	rbp	rbp→	rbp	rbp→	rbp
@ - 24		@retorn		@retorn		@retorn		@retorn
@ - 16		5		5	rbp+16→	**5**		**5**
@ - 8		----		----		----	rbp+24→	120
@		----		----		----		----

	pop rbx pop rax mov rsp,rbp pop rbp		ret		add rsp,8		pop qword[result]	
	apuntadores	pila	apuntadores	pila	apuntadores	pila	apuntadores	pila
@ - 48								
@ - 40								
@ - 32								
@ - 24	rsp→	@retorn						
@ - 16		**5**	rsp→	**5**		----		
@ - 8		120		120	rsp→	120		----
@		----		----		----	rsp→	----

Hay que tener presente que el procesador también utiliza la pila para almacenar los valores necesarios para poder efectuar un retorno de la subrutina de manera correcta. En concreto, siempre que se ejecuta una instrucción *call*, el procesador guarda en la cima de la pila la dirección de retorno (el valor actualizado del contador de programa RIP).

Es importante asegurarse de que en el momento de hacer *ret*, la dirección de retorno se encuentre en la cima de la pila; en caso contrario, se romperá la secuencia normal de ejecución.

Eso se consigue si no se modifica el valor de RBP dentro de la subrutina y al final se ejecutan las instrucciones:

```
mov rsp, rbp    ;Restauramos el valor inicial de RSP con RBP
pop rbp         ;Restauramos el valor inicial de RBP
```

Otra manera de acceder a los parámetros de entrada que tenemos en la pila es sacar los parámetros de la pila y almacenarlos en registros; en este caso, será necesario primero sacar la dirección de retorno, a continuación sacar los parámetros y volver a introducir la dirección de retorno. Esta manera de acceder a los parámetros no se suele utilizar porque los registros que guardan los parámetros no se pueden utilizar para otros propósitos y el número de parámetros viene condicionado por el número de registros disponibles. Además, la gestión de los valores de retorno complica mucho la gestión de la pila.

```
pop rax     ;rax contendrá la dirección de retorno
pop rdx     ;recuperamos los parámetros.
pop rcx
push rax    ;Volvemos a introducir la dirección de retorno
```

La eliminación del espacio utilizado por los parámetros de entrada de la subrutina se lleva a cabo fuera de la subrutina, incrementando el valor del registro RSP en tantas unidades como bytes ocupaban los parámetros de entrada.

```
add rsp,8 ;liberamos el espacio utilizado por el parámetro de entrada
```

Esta tarea también se puede hacer dentro de la subrutina con la instrucción *ret*, que permite especificar un valor que indica el número de bytes que ocupaban los parámetros de entrada, de manera que al retornar, se actualiza RSP incrementándolo tantos bytes como el valor del parámetro *ret*.

Poner la instrucción siguiente al final de la subrutina factorial sería equivalente a ejecutar la instrucción `add rsp,8` después de la llamada a subrutina:

```
ret 8
```

5.4.4. Variables locales en ensamblador

En los lenguajes de alto nivel es habitual definir variables locales dentro de las funciones definidas en un programa. Estas variables locales ocupan un espacio definido dentro de la pila.

En ensamblador no es habitual definir variables locales, en su lugar se utilizan registros, y en caso de necesitar más espacio se utiliza puntualmente la pila.

Ejemplo

```
long int factorial (long int x) {
  long int j, result;

  result=1;
  for (j = x; j > 1; j--){
    result=result*j;
  }

  return result;
}
```

```
factorial:
  push rbp
  mov rbp, rsp
  push rbx

  mov rax, 1
  mov rbx, rdi
for:
  cmp rbx, 1
  jle fi
  imul rax, rbx
  dec rbx
  jmp for
fi:
  pop rbx
  mov rsp, rbp
  pop rbp
  ret
```

El parámetro de entrada *x* del código C se recibe sobre el registro *rdi* en el código ensamblador (rdi es el registro que se utiliza como primer parámetro de entrada en las llamadas en C) y actúa como variable local dentro de la función de C.

La variable local *j* del código C se implementa utilizando el registro *rbx* en ensamblador.

El parámetro de salida del código C, la variable *result*, que también está declarada como variable local, se implementa con el registro *rax*, que es el registro que se utiliza para retornar el resultado en las llamadas en C.

Si se quieren implementar variables locales, tal como se gestionan en C, se debería hacer lo siguiente.

Para reservar el espacio necesario, hemos de saber cuántos bytes utilizaremos como variables locales. A continuación, es necesario decrementar el valor del apuntador a pila RSP tantas unidades como bytes se quieran reservar para las variables locales; de esta manera, si utilizamos las instrucciones que trabajan con la pila dentro de la subrutina (push y pop) no sobreescribiremos el espacio de las variables locales. La actualización de RSP se hace justo después de actualizar el registro que utilizamos para acceder a la pila, RBP.

```
subrutina:
   push rbp           ; Almacenar el registro que utilizaremos
                      ; de apuntador a la pila rbp
   mov rbp, rsp       ; Asignar a RBP el valor del registro apuntador RSP
   sub rsp, n         ; n indica el número de bytes reservados para las
                      ; variables locales
                      ;
                      ; Instrucciones de la subrutina
                      ;
      mov rsp, rbp    ; Restauramos el valor inicial de RSP con RBP
   pop rbp            ; Restauramos el valor inicial de RBP
ret
```

Como las variables locales están en la pila, se utiliza también el registro apuntador RBP para acceder a las variables. Se utiliza un direccionamiento relativo sobre el registro RBP, para acceder al espacio reservado, restando un valor al registro RBP.

Ejemplo

```
push rbp                ;Almacenar el registro que utilizaremos
                        ;de apuntador a la pila rbp
mov rbp, rsp            ;Asignar a RBP el valor del registro apuntador RSP
sub rsp, 8              ;reservamos 8 bytes para variables locales

mov al, byte[RBP-1]     ;accedemos a 1 byte de almacenamiento local
mov ax, word[RBP-2]     ;accedemos a 2 bytes de almacenamiento local
mov eax, dword[RBP-4]   ;accedemos a 4 bytes de almacenamiento local
mov rax, qword[RBP-8]   ;accedemos a 8 bytes de almacenamiento local
```

5.4.5. Llamadas a subrutinas y paso de parámetros desde C

En este subapartado se explica cómo llamar a subrutinas escritas en lenguaje de ensamblador desde un programa escrito en C y cómo transferirle parámetros.

Para poder utilizar una subrutina escrita en lenguaje de ensamblador dentro de un programa escrito en C, debemos tener presentes una serie de cuestiones que afectan tanto al código en lenguaje C como al código en lenguaje de ensamblador.

Definición e implementación de funciones en ensamblador

Cualquier subrutina en ensamblador a la que se quiera llamar desde otro programa se tendrá que declarar utilizando la directiva *global*.

Se puede acceder a cualquier variable global de C desde ensamblador declarándola en el código ensamblador con la directiva `extern`.

La estructura general del código ensamblador sería la siguiente:

```
section .text
                     ;Declararemos con global el nombre de las subrutinas a las que
                     ;queremos acceder desde otros ficheros de código fuente
   global subrutina1, ..., subrutina
```

```
                    ;Declaramos con extern las variables globales de C a las que
                    ;queremos acceder desde ensamblador
   extern variable1, ..., variableN

subrutina1:
   push rbp          ;Almacenar el registro rbp en la pila
   mov rbp,rsp       ;Asignar el valor del registro apuntador RSP
   sub rsp, n        ;Reservamos n bytes para variables locales
         ;
         ; código de la subrutina
         ;
         mov rsp, rbp ;Restauramos el valor inicial de RSP con RBP
         pop rbp      ;Restauramos el valor inicial de RBP
         ret
   ...
subrutinaN:
   push rbp          ;Almacenar el registro rbp en la pila
   mov rbp,rsp       ;Asignar el valor del registro apuntador RSP
   sub rsp, n        ;Reservamos n bytes para variables locales
   ;
   ; código de la subrutina
   ;
   mov rsp, rbp      ;Restauramos el valor inicial de RSP con RBP
   pop rbp           ;Restauramos el valor inicial de RBP
   ret
```

Ejemplo de llamada a una subrutina de ensamblador desde C y acceso a variables globales de C desde ensamblador

Código ensamblador:

```
.text

global factorial          ; Declaramos con global la subrutina de ensamblador
                          ; que queremos hacer visible en el código C

extern x, result          ; Declaramos con extern las variables globales de C
                          ; a las que queremos acceder desde ensamblador

factorial:
       push rbp
       mov rbp, rsp

       push rax           ; Almacenamos en la pila los registros
       push rbx           ; modificados dentro de la subrutina

       mov rax, 1         ; rax será el resultado
       mov rbx, [x]       ; Calculamos el factorial de la variable 'x' (=5)
while:
       cmp rbx, 1         ; Hacemos la comparación
   jle fi                 ; Si se cumple la condición saltamos a fin
   imul rax, rbx          ; En rax tendremos el valor del factorial de 'x' (=120)

   dec rbx
   jmp while              ; Salta a while
fin:
   mov [result], rax      ; Almacenamos el resultado en la variable result

   pop rbx                ; Restauramos el valor inicial de los registros
   pop rax                ; en orden inverso a como los hemos almacenado

   mov rsp, rbp
   pop rbp

ret
```

Código C:

```
long int x, result;   //Variables globales a las cuales se accede desde ensamblador

int main() {
   x=5;
   factorial();       //Llamada a la subrutina de ensamblador

   return 0;
}
```

Paso de parámetros y retorno de resultado

En el modo de 64 bits, cuando se llama desde C una función en lenguaje C o una subrutina en ensamblador, los seis primeros parámetros se pasan por registro utilizando los registros siguientes en el orden especificado:

RDI, RSI, RDX, RCX, R8 y R9.

El resto de los parámetros se pasan por medio de la pila.

El valor de retorno se pasa siempre por registro y se utiliza siempre el registro RAX.

Ejemplo

Queremos definir e implementar algunas subrutinas en ensamblador: *suma*, *producto*, *factorial*. La subrutina *suma* recibirá dos parámetros, hará la suma de los dos parámetros y retornará la suma; la subrutina *producto* también recibirá dos parámetros, hará el producto de los dos parámetros y retornará el resultado; finalmente, la subrutina *factorial* hará el factorial de un número recibido como parámetro (corresponde a la misma función factorial de los subapartados anteriores).

El código ensamblador sería el siguiente:

```
;Fichero funciones.asm

section .text

global suma, producto, factorial

suma:
   push rbp
   mov rbp, rsp
;2 parámetros de entrada: rdi, rsi
   mov rax, rdi
   add rax, rsi

   mov rsp, rbp
   pop rbp

   ret       ;retorno de resultado por medio de rax, rax=rdi+rsi



producto:
   push rbp
   mov rbp, rsp
;2 parámetros de entrada: rdi, rsi
   mov rax, rdi
   imul rax, rsi   ;rax=rax*rsi=rdi*rsi

   mov rsp, rbp
   pop rbp

   ret          ;retorno de resultado por medio de rax

factorial:
   push rbp
   mov rbp, rsp
;1 parámetro de entrada: rdi
;no hay variables locales
   push rdi       ;rdi es modificado por la subrutina
   mov rax, 1       ;rax será el resultado
while:
   cmp rdi, 1      ; Hacemos la comparación
   jle fi          ; Si se cumple la condición saltamos a fin
   imul rax, rdi
   dec rdi
   jmp while       ; Salta a while
fin:                ; En rax tendremos el valor del factorial de rbx
   pop rdi         ;restauramos el valor de rdi

   mov rsp, rbp
   pop rbp

   ret
```

Veamos cómo sería el código C que llama a las subrutinas *suma*, *producto* y *factorial*:

```
//Fichero principal.c

#include <stdio.h>
```

```
int main()
{
   int x, y, result;

   printf("\nIntroduce el valor de x: ");
   scanf("%d", &x);
   printf("Introduce el valor de y: ");
   scanf("%d", &y);

   result = suma(x, y); // Llamamos a la subrutina suma pasando las variables
                        // x e y como parámetros y retornando
                        // el resultado sobre la variable result
   printf ("La suma de x e y es %d\n", result);

   result = producto(x, y); // Llamamos a la subrutina producto pasando las
                            // variables x e y como parámetros y retornando
                            // el resultado sobre la variable result

   printf ("El producto de x e y es %d\n", result);

   result = factorial(x); // Llamamos a la subrutina factorial pasando la
                          // variable x como parámetro y retornando
                          // el resultado sobre la variable result

   printf ("El factorial de x es %d\n", result);
   return 0;
}
```

Para ejecutar este código, hay que hacer el ensamblaje del código fuente ensamblador, compilar el código fuente C con el código objeto ensamblador y generar el ejecutable.

Parámetros por referencia

Se pueden pasar parámetros por valor o por referencia.

En un paso por valor se está pasando a la subrutina un valor para ser utilizado dentro de la subrutina, se pasa el contenido de una variable. Al salir de la subrutina el valor de la variable no se habrá modificado.

La otra opción es pasar parámetros a una subrutina por referencia, pasando a la subrutina la dirección de una variable, de modo que, si dentro de la subrutina se modifica este parámetro, al salir de la subrutina este mantenga el valor obtenido dentro de la subrutina.

Ejemplo

Definimos dos subrutinas inivec y getvec.

inivec recibe la dirección de un vector (parámetro por referencia) y un índice del vector (parámetro por valor). Dentro de la subrutina se inicializa el elemento del vector a cero y, al salir de la subrutina, este elemento quedará asignado a cero.

getvec recibe la dirección de un vector (parámetro por referencia), un índice (parámetro por valor) y la dirección de una variable (parámetro por referencia). Dentro de la subrutina la variable se actualiza con el valor del elemento indicado por el índice y, al salir de la subrutina, la variable quedará asignada con el valor obtenido del vector.

Código ensamblador

```
section .text
global inivec, getvec

inivec:
   push rbp
   mov rbp, rsp
                          ; rdi: primer parámetro, dirección del vector
                          ; rsi: segundo parámetro, índice del vector
   mov byte [rdi+rsi], 0 ; se pone a cero el elemento del vector

   mov rsp, rbp
   pop rbp

ret

getvec:
   push rbp
   mov rbp, rsp
                           ; rdi: primer parámetro, dirección del vector
                           ; rsi: segundo parámetro, índice del vector
                           ; rdx: tercer parámetro, variable pasada por
                           ;      referencia
   mov byte al, [rdi+rsi] ; se asigna al registro 'al' el elemento del vector
   mov byte [rdx], al     ; se pasa el elemento del vector a la variable recibida
                           ; como parámetro
   mov rsp, rbp
   pop rbp

   ret
```

Código C

```
int main(){
   int j;
   char v[5],x;

   for(j=0;j<5;j++){
      inivec(v,j);    // v se pasa por referencia, se modifica dentro de la
                      // subrutina. El nombre de una variable de tipo vector
                      // representa la dirección del primer elemento del vector.
                      // j es el índice del vector al cual queremos acceder
                      // y se pasa por valor
   }
   j=0;
   getvec(v, j, &x)   // x se pasa por referencia, se modifica dentro de la
                      // subrutina con el valor del elemento que ocupa la
                      // posición j dentro del vector v. v se pasa por
                      // referencia y j se pasa por valor
}
```

5.5. Entrada/salida

El sistema de E/S en los procesadores de la familia x86-64 utiliza un mapa independiente de E/S.

Se dispone de un espacio separado de direccionamiento de 64 K, accesible como puertos de 8, 16 y 32 bits; según el dispositivo, se podrá acceder a unidades de 8, 16 o 32 bits.

El juego de instrucciones dispone de instrucciones específicas, IN y OUT, para leer de un puerto de E/S y escribir en un puerto de E/S.

5.5.1. E/S programada

Se pueden realizar tareas de E/S programada utilizando las instrucciones IN y OUT de ensamblador, o las instrucciones equivalentes de lenguaje C: *inb*, *outb*.

En los sistemas Linux debemos permitir el acceso a los puertos de E/S utilizando la llamada al sistema *io_perm*, antes de acceder a un puerto de E/S.

Ejemplo

A continuación se muestra un programa escrito en C que accede a los puertos de E/S 70h y 71h, correspondientes a los registros de direcciones y de datos del reloj de tiempo real del sistema (RTC).

La dirección 70h corresponde al registro de direcciones, en el que se escribe la dirección del dato del RTC que se quiere leer:

00: segundos
02: minutos
04: hora
08: mes del año

```
//Fichero: ioprog.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>

#define RTC_ADDR 0x70 // registro de direcciones del RTC
#define RTC_DATA 0x71 // registro de datos del RTC

int main()
{
   char month, hh, mm;

   // Se define cada mes del año de 10 caracteres
   // 'septiembre', 'noviembre' y 'diciembre' son los más largos
   // y ocupan 10 caracteres incluido un \0 al final
   char meses[12][10]={"enero", "febrero", "marzo", "abril", "mayo", "junio",
       "julio", "agosto", "septiembre", "octubre", "noviembre", "diciembre"};

   /* Se proporciona permiso para acceder a los puertos de E/S
    * RTC_ADDR dirección inicial a partir de la cual se quiere acceder
    * 2: se pide acceso a 2 bytes
    * 1: se activa el permiso de acceso */
   if (ioperm(RTC_ADDR, 2, 1)) {
      perror("ioperm");
      exit(1);
   }

   outb(8, RTC_ADDR);   // Se escribe en el registro de direcciones la dirección 8: mes
   month=inb(RTC_DATA); // Se lee del puerto de datos el mes del año

   // El valor obtenido es el número de mes entre 1 y 12, pero el índice
   // del vector meses está entre 0 y 11 y hay que restar 1
   printf("Mes año RTC: %d %s\n", month, meses[month-1]);

   outb(4, RTC_ADDR);  // Se escribe en el registro de direcciones la dirección 4: hora
   hh=inb(RTC_DATA);   // Se lee del puerto de datos la hora
```

```
    printf("Hora RTC: %0x:", hh);

    outb(2, RTC_ADDR);  // Se escribe en el registro de direcciones la dirección 2: minutos
    mm=inb(RTC_DATA);   // Se leen del puerto de datos los minutos

    printf("%0x\n", mm);

    // Se desactivan los permisos para acceder a los puertos poniendo un 0
    if (ioperm(RTC_ADDR, 2, 0)) {
       perror("ioperm");
       exit(1);
    }

    return 0;
}
```

Para poder ejecutar un programa que acceda a puertos de E/S en Linux, es necesario disponer de permisos de superusuario.

Ejemplo

A continuación se muestra otro ejemplo de acceso a puertos de E/S: se trata de leer el registro de datos del teclado, puerto 60h.

El código siguiente hace un bucle que muestra el código de la tecla pulsada (*scancode*) hasta que se pulsa la tecla ESC.

El registro 60h almacena en los 7 bits de menos peso el código de cada tecla que se pulsa.

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>

#define KBD_DATA 0x60

int main(){
   int salir=0;
   char data;

   if (ioperm(KBD_DATA, 1, 1)) {
      perror("ioperm");
      exit(1);
   }
   while (salir==0){
      data=inb(KBD_DATA);
      // se muestra el código de la tecla pulsada, scancode
      // se hace una AND lógica ('&') para tomar solo los
      // 7 bits menos significativos del dato leído
      printf("tecla: %0x\n", data & 0b01111111);
      // se pulsa ESC para salir del bucle, scancode=1
      if ((data & 0b01111111) == 1) salir=1;
   }
   if (ioperm(KBD_DATA, 1, 0)) {
      perror("ioperm");
      exit(1);
   }
   return 0;
}
```

5.6. Controlar la consola

En Linux no existen funciones estándar o servicios del sistema operativo para controlar la consola, por ejemplo, para mover el cursor. Disponemos, sin embargo, de la posibilidad de escribir secuencias de *escape* que nos permiten manipular el emulador de terminal de Linux, entre otras operaciones: mover el cursor, limpiar la consola, etc.

Una secuencia de *escape* es una cadena de caracteres que empieza con el carácter ESC (que corresponde al código ASCII 27 decimal o 1Bh).

Las principales secuencias de *escape* son las siguientes:

1) Mover el cursor. Para mover el cursor hay que considerar que la posición inicial en un terminal Linux corresponde a la fila 0 de la columna 0; para mover el cursor escribiremos la cadena de caracteres siguiente:

```
ESC[F;CH
```

donde *ESC* corresponde al código ASCII del carácter de *escape* 27 decimal o 1Bh, *F* corresponde a la fila donde queremos colocar el cursor, expresada como un valor decimal, y *C* corresponde a la columna donde queremos colocar el cursor, expresada como un valor decimal.

Ejemplo

Podemos mover el cursor a la fila 5 columna 10, escribiendo la secuencia `ESC[05;10H`.

```
section .data
   escSeq db 27,"[05;10H"
   escLen equ 8
section .text
   mov rax,4
   mov rbx,1
   mov rcx, escSeq
   mov rdx, escLen
   int 80h
```

2) Limpiar la consola. Para limpiar la consola debemos escribir la secuencia `ESC[2J`.

```
section .data
   escSeq db 27,"[2J" ;ESC[2J
   escLen equ 4       ; tamaño de la cadena escSeq

section .text
   mov rax,4
   mov rbx,1
   mov rcx, escSeq
   mov rdx, escLen
   int 80h
```

3) Secuencias de *escape* en C. En lenguaje C podemos conseguir el mismo resultado escribiendo la secuencia que emplea la función *printf*; el carácter ESC se puede escribir con su valor hexadecimal 1B.

Ejemplo

```
#include <stdio.h>

int main(){
   printf("\x1B[2J");       //Borra la pantalla
      printf("\x1B[5;10H"); //Sitúa el cursor en la posición (5,10)
}
```

5.7. Funciones del sistema operativo (*system calls*)

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; veremos solo las funciones siguientes:

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

5.7.1. Lectura de una cadena 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

```
section .bss
   buffer resb 10               ;se reservan 10 bytes para hacer la lectura del
teclado


section .text
                                ;lectura utilizando int 80h
   mov rax, 3
   mov rbx, 0
   mov rcx, buffer              ; se carga la dirección de buffer en rcx
```

```
    mov rdx,10                  ; número máximo de caracteres que se leerán
    int 80h                     ; se llama al kernel


                                ;lectura utilizando syscall
    mov rax, 0
    mov rdi, 0
    mov rsi, buffer             ; se carga la dirección de buffer en rsi
    mov rdx,10                  ; número máximo de caracteres que se leerán
    syscall                     ; se llama al kernel
```

5.7.2. Escritura de una cadena 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

Ejemplo

```
Section .data
   msg db "Hola!",0
   msgLen db 6

section .text
                    ; escritura utilizando int 80h
   mov rax,4
   mov rbx,1
   mov rcx, msg       ; se pone la dirección de msg1 en rcx
   mov rdx, msgLen    ; tamaño de msg
   int 80h            ; se llama al kernel

;escritura utilizando syscall
   mov rax, 1
   mov rdi, 1
   mov rsi, msg       ; se carga la dirección de msg en rsi
   mov rdx, msglen    ; tamaño de msg
   syscall            ; se llama al kernel
```

Se puede calcular el tamaño de una variable calculando la diferencia entre una posición dentro de la sección data y la posición donde se encuentra declarada la variable:

```
section .data
  msg db "Hola!",0
  msgLen db equ $ - msg ; $ indica la dirección de la posición actual
```

$ define la posición del principio de la línea actual; al restarle la etiqueta *msg*, se le resta la posición donde se encuentra declarada la etiqueta y, por lo tanto, se obtiene el número de bytes reservados a la posición de la etiqueta *msg*, 6 en este caso.

5.7.3. Retorno al sistema operativo (*exit*)

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

Ejemplo

```
;retorna al sistema operativo utilizando int 80h
mov rax,1
mov rbx,0  ;valor de retorno 0
int 80h   ; se llama al kernel

           ;retorna al sistema operativo utilizando syscall
mov rax,60
mov rbx,0  ;valor de retorno 0
syscall    ;se llama al kernel
```

6. Anexo: manual básico del juego de instrucciones

En este anexo se describen detalladamente las instrucciones más habituales del lenguaje de ensamblador de la arquitectura x86-64, pero debemos tener presente que el objetivo de este apartado no es ofrecer un manual de referencia completo de esta arquitectura y, por lo tanto, no se describen todas las instrucciones del juego de instrucciones.

Descripción de la notación utilizada en las instrucciones:

1) Bits de resultado (*flags*):

- OF: Overflow flag (bit de desbordamiento)
- TF: Trap flag (bit de excepción)
- AF: Aux carry (bit de transporte auxiliar)
- DF: Direction flag (bit de dirección)
- SF: Sign flag (bit de signo)
- PF: Parity flag (bit de paridad)
- IF: Interrupt flag (bit de interrupción)
- ZF: Cero flag (bit de cero)
- CF: Carry flag (bit de transporte)

2) Tipo de operandos:

a) `imm`: valor inmediato; puede ser un valor inmediato de 8, 16 o 32 bits. Según el tamaño del valor inmediato se podrán representar los intervalos de valores siguientes:

- Inmediato de 8 bits: sin signo [0, 255], con signo en Ca2 [–128, +127]

- Inmediato de 16 bits: sin signo [0, 65.535], con signo en Ca2 [–32.768, +32.767]

- Inmediato de 32 bits: sin signo [0, 4.294.967.295], con signo en Ca2 [–2.147.483.648, +2.147.483.647]

Los valores inmediatos de 64 bits solo se permiten para cargar un registro de propósito general de 64 bits, mediante la instrucción MOV. El intervalo de representación es el siguiente:

- sin signo [0, 18.446.744.073.709.551.615],
- con signo en Ca2 [–9.223.372.036.854.775.808, + 9.223.372.036.854.775.807]

b) `reg`: registro, puede ser un registro de 8, 16, 32 o 64 bits.
`reg8`: registro, tiene que ser un registro de 8 bits.
`reg16`: registro, tiene que ser un registro de 16 bits.
`reg32`: registro, tiene que ser un registro de 32 bits.
`reg64`: registro, tiene que ser un registro de 64 bits.

c) `tamaño mem`: posición de memoria; se indica primero si se accede a 8, 16, 32 o 64 bits y, a continuación, la dirección de memoria.

Aunque especificar el tamaño no es estrictamente necesario, siempre que se utiliza un operando de memoria lo haremos de esta manera, por claridad a la hora de saber a cuántos bytes de memoria se está accediendo.

Los especificadores de tamaño válidos son:

- BYTE: posición de memoria de 8 bits
- WORD: posición de memoria de 16 bits
- DWORD: posición de memoria de 32 bits
- QWORD: posición de memoria de 64 bits

6.1. ADC: suma aritmética con bit de transporte

```
ADC destino, fuente
```

Efectúa una suma aritmética; suma el operando fuente y el valor del bit de transporte (CF) al operando de destino, almacena el resultado sobre el operando destino y sustituye el valor inicial del operando destino.

Operación

destino = destino + fuente + CF

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Si el resultado no cabe dentro del operando destino, el bit de transporte se pone a 1. El resto de bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
ADC reg, reg
ADC reg, tamaño mem
ADC tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
ADC reg, imm
ADC tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, 32 bits como máximo.

Ejemplos

```
ADC R9,RAX
ADC RAX, QWORD [variable]
ADC DWORD [variable],EAX
ADC RAX,0x01020304
ADC BYTE [vector+RAX], 5
```

6.2. ADD: suma aritmética

ADD destino, fuente

Efectúa la suma aritmética de los dos operandos de la instrucción, almacena el resultado sobre el operando destino y sustituye el valor inicial del operando destino.

Operación

destino = destino + fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Si el resultado no cabe dentro del operando destino, el bit de transporte se pone a 1. El resto de bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
ADD reg, reg
ADD reg, tamaño mem
ADD tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
ADD reg, imm
ADD tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, 32 bits como máximo.

Ejemplos

```
ADD R9,RAX
ADD RAX, QWORD [variable]
ADD DWORD [variable],EAX
ADD RAX,0x01020304
ADD BYTE [vector+RAX], 5
```

6.3. AND: Y lógica

AND destino, fuente

Realiza una operación lógica AND ('y lógica') bit a bit entre el operando destino y el operando fuente, el resultado de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Se lleva a cabo una operación AND entre el bit *n* del operando destino y el bit *n* del operando fuente según la tabla de verdad de la función AND:

```
x       y           x AND y
---         ---           -------
0          0            0
0          1            0
1          0            0
1          1            1
```

Operación

destino = destino AND fuente

Bits de resultado modificados

OF=0, SF, ZF, PF, CF=0

Los bits de resultado OF y CF se ponen a 0, el resto se cambia según el resultado de la operación.

Formatos válidos

```
AND reg,reg
```

```
AND reg,tamaño mem
AND tamaño mem,reg
```

Los dos operandos han de ser del mismo tamaño.

```
AND reg,imm
AND tamaño mem,imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
AND R9,RAX
AND RAX, QWORD [variable]
AND DWORD [variable],EAX
AND RAX,0x01020304
AND BYTE [vector+RAX], 5
```

6.4. CALL: llamada a subrutina

CALL etiqueta

Llama a la subrutina que se encuentra en la dirección de memoria indicada por la etiqueta. Guarda en la pila la dirección de memoria de la instrucción que sigue en secuencia la instrucción CALL y permite el retorno desde la subrutina con la instrucción RET; a continuación carga en el RIP (*instruction pointer*) la dirección de memoria donde está la etiqueta especificada en la instrucción y transfiere el control a la subrutina.

En entornos de 64 bits, la dirección de retorno será de 64 bits, por lo tanto, se introduce en la pila un valor de 8 bytes. Para hacerlo, primero se actualiza el puntero de pila (registro RSP) decrementándose en 8 unidades, y a continuación se copia la dirección en la cima de la pila.

Operación

RSP=RSP-8
M[RSP]← RIP
RIP ← dirección_etiqueta

Bits de resultado modificados

Ninguno

Formatos válidos

```
CALL etiqueta
```

Existen otros formatos, pero quedan fuera de los objetivos de estos materiales. Podéis consultar las fuentes bibliográficas.

Ejemplos

```
CALL subrutina1
```

6.5. CMP: comparación aritmética

CMP destino, fuente

Compara los dos operandos de la instrucción sin afectar al valor de ninguno de los operandos, actualiza los bits de resultado según el resultado de la comparación. La comparación se realiza con una resta entre los dos operandos, sin considerar el transporte y sin guardar el resultado.

Operación

destino - fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado se modifican según el resultado de la operación de resta.

Formatos válidos

```
CMP reg,reg
CMP reg,tamaño mem
CMP tamaño mem,reg
```

Los dos operandos han de ser del mismo tamaño.

```
CMP reg,imm
CMP tamaño mem,imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
CMP R9,RAX
CMP RAX, QWORD [variable]
CMP DWORD [variable],EAX
CMP RAX,0x01020304
CMP BYTE [vector+RAX], 5
```

6.6. DEC: decrementa el operando

```
DEC destino
```

Resta 1 al operando de la instrucción y almacena el resultado en el mismo operando.

Operación

destino = destino – 1

Bits de resultado modificados

OF, SF, ZF, AF, PF

Los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
DEC reg
DEC tamaño mem
```

Ejemplos

```
DEC EAX
DEC R9
DEC DWORD [R8]
DEC QWORD [variable]
```

6.7. DIV: división entera sin signo

```
DIV fuente
```

Divide el dividendo implícito entre el divisor explícito sin considerar los signos de los operandos.

Si el divisor es de 8 bits, se considera como dividendo implícito AX. El cociente de la división queda en AL y el resto, en AH.

Si el divisor es de 16 bits, se considera como dividendo implícito el par de registros DX:AX; la parte menos significativa del dividendo se coloca en AX y la parte más significativa, en DX. El cociente de la división queda en AX y el resto, en DX.

Si el divisor es de 32 bits, el funcionamiento es similar al caso anterior, pero se utiliza el par de registros EDX:EAX; la parte menos significativa del dividendo se coloca en EAX y la parte más significativa, en EDX. El cociente de la división queda en EAX y el resto, en EDX.

Si el divisor es de 64 bits, el funcionamiento es parecido a los dos casos anteriores, pero se utiliza el par de registros RDX:RAX; la parte menos significativa del dividendo se coloca en RAX y la parte más significativa, en RDX. El cociente de la división queda en RAX y el resto, en RDX.

Operación

Si *fuente* es de 8 bits: AL = AX / *fuente*, AH = AX mod *fuente*

Si *fuente* es de 16 bits: AX = DX:AX / *fuente*, DX = DX:AX mod *fuente*

Si *fuente* es de 32 bits: EAX = EDX:EAX / *fuente*, EDX = EDX:EAX mod *fuente*

Si *fuente* es de 64 bits: RAX = RDX:RAX / *fuente*, RDX = RDX:RAX mod *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

La instrucción DIV no deja información a los bits de resultado, pero estos quedan indefinidos.

Formatos válidos

```
DIV reg
DIV tamaño mem
```

Ejemplos

```
DIV R8B    ; AX / R8B => Cociente en AL; resto en AH
DIV R8W    ; DX:AX / R8W => Cociente en AX; resto en DX
DIV ECX    ; EDX:EAX / ECX => Cociente en EAX; resto en EDX
DIV QWORD [R9] ; RDX:RAX / QWORD [R9] => Cociente en RAX,
resto en RDX
```

6.8. IDIV: división entera con signo

IDIV fuente

Divide el dividendo implícito entre el divisor explícito (fuente) considerando el signo de los operandos. El funcionamiento es idéntico al de la división sin signo.

Si el divisor es de 8 bits, se considera como dividendo implícito AX.

El cociente de la división queda en AL y el resto, en AH.

Si el divisor es de 16 bits, se considera como dividendo implícito el par de registros DX:AX; la parte menos significativa del dividendo se coloca en AX y la parte más significativa, en DX. El cociente de la división queda en AX y el resto, en DX.

Si el divisor es de 32 bits, el funcionamiento es similar al caso anterior, pero se utiliza el par de registros EDX:EAX; la parte menos significativa del dividendo se coloca en EAX, y la parte más significativa, en EDX. El cociente de la división queda en EAX y el resto, en EDX.

Si el divisor es de 64 bits, el funcionamiento es parecido a los dos casos anteriores, pero se utiliza el par de registros RDX:RAX; la parte menos significativa del dividendo se coloca en RAX y la parte más significativa, en RDX. El cociente de la división queda en RAX y el resto, en RDX.

Operación

Si *fuente* es de 8 bits: AL = AX / fuente, AH = AX mod *fuente*

Si *fuente* es de 16 bits: AX = DX:AX / *fuente*, DX = DX:AX mod *fuente*

Si *fuente* es de 32 bits: EAX = EDX:EAX / *fuente*, EDX = EDX:EAX mod *fuente*

Si *fuente* es de 64 bits: RAX = RDX:RAX / *fuente*, RDX = RDX:EAX mod *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

La instrucción IDIV no deja información a los bits de resultado, pero estos quedan indefinidos.

Formatos válidos

```
IDIV reg
IDIV tamaño mem
```

Ejemplos

```
IDIV CH   ; AX / CH => Cociente en AL; resto en AH
IDIV BX   ; DX:AX / BX => Cociente en AX; resto en DX
IDIV ECX   ; EDX:EAX / ECX => Cociente en EAX; resto en EDX
IDIV QWORD [R9] ; RDX:RAX / [R9] => Cociente en RAX, resto en RDX
```

6.9. IMUL: multiplicación entera con signo

IMUL fuente
IMUL destino, fuente

La operación de multiplicación con signo puede utilizar diferente número de operandos; se describirá el formato de la instrucción con un operando y con dos operandos.

6.9.1. IMUL fuente: un operando explícito

Multiplica el operando fuente por AL, AX, EAX, o RAX considerando el signo de los operandos y almacena el resultado en AX, DX:AX, EDX:EAX o RDX:RAX.

Operación

Si *fuente* es de 8 bits AX = AL * *fuente*

Si *fuente* es de 16 bits DX:AX = AX * *fuente*

Si *fuente* es de 32 bits EDX:EAX = EAX * *fuente*

Si *fuente* es de 64 bits RDX:RAX = RAX * *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado SF, ZF, AF, y PF quedan indefinidos después de que se ejecute la instrucción IMUL.

CF y OF se fijan en 0 si la parte alta del resultado es 0 (AH, DX, EDX o RDX), en caso contrario se fijan en 1.

Formatos válidos

```
IMUL reg
IMUL tamaño mem
```

Ejemplos

```
IMUL ECX   ; EAX * ECX => EDX:EAX
IMUL QWORD [RBX]   ; AX * [RBX] => RDX:RAX
```

6.9.2. IMUL destino, fuente: dos operandos explícitos

Multiplica el operando fuente por el operando destino considerando el signo de los dos operandos y almacena el resultado en el operando destino; sobreescribe el valor que tuviera.

Operación

destino = fuente * destino

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado SF, ZF, AF, y PF quedan indefinidos después de que se ejecute la instrucción IMUL.

CF y OF se fijan en 0 si el resultado se puede representar con el operando destino; si el resultado no es representable con el rango del operando destino (se produce desbordamiento), CF y OF se fijan en 1.

Formatos válidos

```
IMUL reg, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

```
IMUL reg, reg
IMUL reg, tamaño mem
```

Los dos operandos han de ser del mismo tamaño.

Ejemplos

```
IMUL EAX, 4
IMUL RAX, R9
IMUL RAX, QWORD [var]
```

6.10. IN: lectura de un puerto de entrada/salida

IN destino, fuente

Lee el valor de un puerto de E/S especificado por el operando fuente y lleva el valor al operando destino.

El operando fuente puede ser un valor inmediato de 8 bits, que permite acceder a los puertos 0-255, o el registro DX, que permite acceder a cualquier puerto de E/S de 0-65535.

El operando destino solo puede ser uno de los registros siguientes:

- AL se lee un byte del puerto
- AX se leen dos bytes
- EAX se leen cuatro bytes

Operación

destino=fuente(puerto E/S)

Bits de resultado modificados

Ninguno

Formatos válidos

```
IN AL, imm8
IN AX, imm8
IN EAX, imm8
IN AL, DX
IN AX, DX
IN EAX, DX
```

Ejemplos

```
IN AL, 60 h
IN AL, DX
```

6.11. INC: incrementa el operando

```
INC destino
```

Suma 1 al operando de la instrucción y almacena el resultado en el mismo operando.

Operación

destino = destino + 1

Bits de resultado modificados

OF, SF, ZF, AF, PF

Los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
INC reg
INC tamaño mem
```

Ejemplos

```
INC AL
INC R9
INC BYTE [RBP]
INC QWORD [var1]
```

6.12. INT: llamada a una interrupción software

```
INT servicio
```

Llamada a un servicio del sistema operativo, a una de las 256 interrupciones *software* definidas en la tabla de vectores de interrupción. El número de servicio ha de ser un valor entre 0 y 255. Es habitual expresar el número del servicio como un valor hexadecimal, de 00h a FFh.

Cuando se llama a una interrupción, el registro EFLAGS y la dirección de retorno son almacenados en la pila.

Operación

RSP=RSP-8
M(RSP) ← EFLAGS
RSP=RSP-8
M(RSP) ← RIP
RIP← dirección rutina de servicio

Bits de resultado modificados

IF, TF

Estos dos bits de resultado se ponen a 0; poner a 0 el bit de resultado IF impide que se trate otra interrupción mientras se está ejecutando la rutina de interrupción actual.

Formatos válidos

```
INT servicio
```

Ejemplos

```
INT 80h
```

6.13. IRET: retorno de interrupción

```
IRET
```

IRET se debe utilizar para salir de las rutinas de servicio a interrupciones (RSI). La instrucción extrae de la pila la dirección de retorno sobre el registro RIP, a continuación saca la palabra siguiente de la pila y la coloca en el registro EFLAGS.

Operación

RIP ← dirección retorno
RSP=RSP+8
EFLAGS ← M(RSP)
RSP=RSP+8

Bits de resultado modificados

Todos: OF, DF, IF, TF, SF, ZF, AF, PF, CF

Modifica todos los bits de resultado, ya que saca de la pila una palabra que es llevada al registro EFLAGS.

Formatos válidos

```
IRET
```

Ejemplo

```
IRET
```

6.14. Jxx: salto condicional

```
Jxx etiqueta
```

Realiza un salto según una condición determinada; la condición se comprueba consultando el valor de los bits de resultado.

La etiqueta codifica un desplazamiento de 32 bits con signo y permite dar un salto de -2^{31} bytes a $+2^{31} - 1$ bytes.

Si la condición se cumple, se salta a la posición del código indicada por la etiqueta; se carga en el registro RIP el valor RIP + desplazamiento,

Operación

RIP = RIP + desplazamiento

Bits de resultado modificados

Ninguno

Formatos válidos

Según la condición de salto, tenemos las instrucciones siguientes:

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
```

Ejemplos

```
JE etiqueta1     ;salta si Z=1
JG etiqueta2     ;salta si Z=0 y SF=OF
JL etiqueta3     ;salta si S≠FOF
```

6.15. JMP: salto incondicional

```
JMP etiqueta
```

Salta de manera incondicional a la dirección de memoria correspondiente a la posición de la etiqueta especificada; el registro RIP toma como valor la dirección de la etiqueta.

Operación

RIP=dirección_etiqueta

Bits de resultado modificados

Ninguno

Formatos válidos

```
JMP etiqueta
```

Ejemplo

```
JMP bucle
```

6.16. LOOP: bucle hasta RCX=0

LOOP etiqueta

La instrucción utiliza el registro RCX.

Decrementa el valor de RCX, comprueba si el valor es diferente de cero y en este caso realiza un salto a la etiqueta indicada.

La etiqueta codifica un desplazamiento de 32 bits con signo y permite efectuar un salto de -2^{31} bytes a $+2^{31} - 1$ bytes.

Operación

La instrucción es equivalente al conjunto de instrucciones siguientes:

```
DEC RCX
JNE etiqueta
```

Bits de resultado modificados

Ninguno

Formatos válidos

```
LOOP etiqueta
```

Ejemplo

```
MOV RCX, 10
bucle:
;
;Las instrucciones se repetirán 10 veces
;
LOOP bucle
```

6.17. MOV: transferir un dato

MOV destino, fuente

Copia el valor del operando fuente sobre el operando destino sobreescribiendo el valor original del operando destino.

Operación

destino = fuente

Bits de resultado modificados

Ninguno

Formatos válidos

```
MOV reg, reg
MOV reg, tamaño mem
MOV tamaño mem, reg
```

Los dos operandos deben ser del mismo tamaño.

```
MOV reg, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del registro; se permiten inmediatos de 64 bits si el registro es de 64 bits.

```
MOV tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
MOV RAX,R9
MOV RAX,QWORD [variable]
MOV QWORD [variable], RAX
MOV RAX,0102030405060708h
MOV WORD [RAX],0B80h
```

6.18. MOVSX/MOVSXD: transferir un dato con extensión de signo

MOVSX destino, fuente
MOVSXD destino, fuente

Copia el valor del operando fuente sobre el operando destino sobrescribiendo el valor original del operando destino y haciendo la extensión de signo. El tamaño del operando fuente tiene que ser inferior al tamaño del operando destino.

MOVSXD solo se utiliza para copiar un valor de 32 bits haciendo la extensión sobre un valor de 64 bits.

Operación

destino = Extensión signo (fuente)

Bits de resultado modificados

Ninguno

Formatos válidos

```
MOVSX reg16, reg8
MOVSX reg16, BYTE imm/mem

MOVSX reg32, reg8
MOVSX reg32, BYTE mem

MOVSX reg64, reg8
;reg8 puede ser cualquier registro de 8 bits excepto: AH,
BH, CH, DH
MOVSX reg64, BYTE mem

MOVSX reg32, reg16
MOVSX reg32, WORD mem

MOVSX reg64, reg16
MOVSX reg64, WORD mem

MOVSXD reg64, reg32
MOVSXD reg64, DWORD mem
```

Ejemplos

```
MOVSX AX,BL          ;extensión de signo de 8 bits a 16 bits

MOVSX EAX,BL         ;extensión de signo de 8 bits a 32 bits

MOVSX RAX,BL         ;extensión de signo de 8 bits a 64 bits
MOVSX RAX, BYTE [var]

MOVSX EAX,BX         ;extensión de signo de 16 bits a 32 bits

MOVSX RAX,BX         ;extensión de signo de 16 bits a 64 bits

MOVSXD RAX, EBX      ;extensión de signo de 32 bits a 64 bits
MOVSXD RAX, DWORD [var]
```

6.19. MOVZX: transferir un dato añadiendo ceros

MOVZX destino, fuente

Copia el valor del operando fuente sobre el operando destino sobrescribiendo el valor original del operando destino, añadiendo ceros delante. El tamaño del operando fuente tiene que ser inferior al tamaño del operando destino.

Operación

destino = Extensión con ceros (fuente)

Bits de resultado modificados

Ninguno

Formatos válidos

```
MOVZX reg16, reg8
MOVZX reg16, BYTE mem

MOVZX reg32, reg8
MOVZX reg32, BYTE mem

MOVZX reg64, reg8
;reg8 puede ser cualquier registro de 8 bits excepto: AH,
BH, CH, DH
MOVZX reg64, BYTE mem

MOVZX reg32, reg16
MOVZX reg32, WORD mem

MOVZX reg64, reg16
```

```
MOVZX reg64, WORD mem
```

Ejemplos

```
MOVZX AX,BL          ;extensión con ceros de 8 bits a 16 bits

MOVZX EAX,BL         ;extensión con ceros de 8 bits a 32 bits

MOVZX RAX,BL         ;extensión con ceros de 8 bits a 64 bits
MOVZX RAX, BYTE [var]

MOVZX EAX,BX         ;extensión con ceros de 16 bits a 32 bits

MOVZX RAX,BX         ;extensión con ceros de 16 bits a 64 bits
MOVZX RAX, WORD [var]
```

6.20. MUL: multiplicación entera sin signo

MUL fuente

MUL multiplica el operando explícito por AL, AX, EAX o RAX sin considerar el signo de los operandos y almacena el resultado en AX, DX:AX, EDX:EAX o RDX:RAX.

Operación

Si *fuente* es de 8 bits AX = AL * *fuente*

Si *fuente* es de 16 bits DX:AX = AX * *fuente*

Si *fuente* es de 32 bits EDX:EAX = EAX * *fuente*

Si *fuente* es de 64 bits RDX:RAX = RAX * *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado SF, ZF, AF, y PF quedan indeterminados después de que se ejecute la instrucción MUL.

CF y OF se fijan a 0 si la parte alta del resultado es 0 (AH, DX, EDX, o RDX); en caso contrario se fijan a 1.

Formatos válidos

```
MUL reg
MUL tamaño mem
```

Ejemplos

```
MUL CH           ; AL * CH --> AX
MUL BX           ; AX * BX --> DX:AX
MUL RCX          ; RAX * RCX --> RDX:RAX
MUL WORD [BX+DI] ; AX * [BX+DI]--> DX:AX
```

6.21. NEG: negación aritmética en complemento a 2

NEG destino

Lleva a cabo una negación aritmética del operando, es decir, hace el complemento a 2 del operando especificado; es equivalente a multiplicar el valor del operando por –1.

Esta operación no es equivalente a complementar todos los bits del operando (instrucción NOT).

Operación

destino = (–1) * destino

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Si el operando es 0, el resultado no varía y ZF=1 y CF=0; en caso contrario, ZF=0 y CF=1.

Si el operando contiene el máximo valor negativo (por ejemplo 80000000h si el operando es de 32 bits), el valor del operando no se modifica: OF=1 y CF=1.

SF=1 si el resultado es negativo; SF=0 en caso contrario.

PF=1 si el número de unos del byte de menos peso del resultado es par; PF=0 en caso contrario.

Formatos válidos

```
NEG reg
NEG tamaño mem
```

Ejemplos

```
NEG RCX
NEG DWORD [variable]
```

6.22. NOT: negación lógica (negación en complemento a 1)

NOT destino

Lleva a cabo una negación lógica del operando, es decir, hace el complemento a 1 del operando especificado, y complementa todos los bits del operando.

Operación

destino = –destino

Bits de resultado modificados

Ninguno

Formatos válidos

```
NOT reg
NOT tamaño mem
```

Ejemplos

```
NOT RAX
NOT QWORD [variable]
```

6.23. OUT: escritura en un puerto de entrada/salida

OUT destino, fuente

Escribe el valor del operando fuente en un puerto de E/S especificado por el operando destino.

El operando destino puede ser un valor inmediato de 8 bits, que permite acceder a los puertos 0-255, o el registro DX, que permite acceder a cualquier puerto de E/S de 0-65535.

El operando fuente solo puede ser uno de los registros siguientes:

- AL se escribe un byte
- AX se escriben dos bytes
- EAX se escriben cuatro bytes

Operación

destino(puerto E/S) = fuente

Bits de resultado modificados

Ninguno

Formatos válidos

```
OUT imm8, AL
OUT imm8, AX
OUT imm8, EAX
OUT DX, AL
OUT DX, AX
OUT DX, EAX
```

Ejemplos

```
OUT 60h, AL
OUT DX, AL
```

6.24. OR: o lógica

OR destino, fuente

Realiza una operación lógica OR (o lógica) bit a bit entre el operando destino y el operando fuente; el resultado de la operación se guarda sobre el operando destino, sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Se lleva a cabo una operación OR entre el bit *n* del operando destino y el bit *n* del operando fuente según la tabla de verdad de la función OR:

```
x     y      x OR y
---   ---      ------
0     0       0
0     1       1
1     0       1
1     1       1
```

Operación

destino = destino OR fuente

Bits de resultado modificados

OF=0, SF, ZF, PF, CF=0

Los bits de resultado OF y CF se ponen a 0, el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
OR reg, reg
OR reg, tamaño mem
OR tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
OR reg, imm
OR tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
OR R9, RAX
OR RAX, QWORD [variable]
OR DWORD [variable], EAX
OR RAX, 0x01020304
OR BYTE [vector+RAX], 5
```

6.25. POP: extraer el valor de la cima de la pila

```
POP destino
```

Extrae el valor que se encuentra en la cima de la pila (copia el valor de la memoria apuntado por el registro RSP) y lo almacena en el operando destino especificado; se extraen tantos bytes de la pila como el tamaño del operando indicado.

A continuación se actualiza el valor del registro apuntador de pila, RSP, incrementándolo en tantas unidades como el número de bytes extraídos de la pila.

El operando puede ser un registro de 16 o 64 bits o una posición de memoria de 16 o 64 bits.

Operación

destino = M[RSP]

La instrucción es equivalente a:

```
MOV destino, [RSP]
ADD RSP, <tamaño del operando>
```

Por ejemplo:

```
POP RAX
```

es equivalente a:

```
MOV RAX, [RSP]
ADD RSP, 8
```

Bits de resultado modificados

Ninguno

Formatos válidos

```
POP reg
POP tamaño mem
```

El tamaño del operando ha de ser de 16 o 64 bits.

Ejemplos

```
POP AX
POP RAX
POP WORD [variable]
POP QWORD [RBX]
```

6.26. PUSH: introducir un valor en la pila

PUSH fuente

Se actualiza el valor del registro apuntador de pila, RSP, decrementándolo en tantas unidades como el tamaño en bytes del operando fuente.

A continuación, se introduce el valor del operando fuente en la cima de la pila, se copia el valor del operando a la posición de la memoria apuntada por el registro RSP y se colocan tantos bytes en la pila como el tamaño del operando indicado.

El operando puede ser un registro de 16 o 64 bits, una posición de memoria de 16 o 64 bits o un valor inmediato de 8, 16 o 32 bits extendido a 64 bits.

Operación

M[RSP]=fuente

La instrucción es equivalente a:

```
SUB RSP, <tamaño del operando>
MOV [RSP], fuente
```

Por ejemplo:

```
PUSH RAX
```

es equivalente a:

```
SUB RSP, 8
 MOV [RSP], RAX
```

Bits de resultado modificados

Ninguno

Formatos válidos

```
PUSH reg
PUSH tamaño mem
```

El operando ha de ser de 16 o 64 bits.

```
PUSH imm
```

El valor inmediato puede ser de 8, 16 o 32 bits.

Ejemplos

```
PUSH AX
PUSH RAX
PUSH WORD [variable]
PUSH QWORD [RBX]
PUSH 0Ah
PUSH 0A0Bh
PUSH 0A0B0C0Dh
```

6.27. RET: retorno de subrutina

```
RET
```

Sale de la subrutina que se estaba ejecutando y retorna al punto donde se había hecho la llamada, a la instrucción siguiente de la instrucción CALL.

Extrae de la pila la dirección de memoria de retorno (la dirección de la instrucción que sigue en secuencia a la instrucción CALL) y la carga en el RIP (*instruction pointer*).

Actualiza el puntero de pila (registro RSP), para que apunte al siguiente elemento de la pila; como la dirección de retorno es de 8 bytes (en modo de 64 bits), incrementa RSP en 8 unidades.

Operación

RIP = M(RSP)
RSP = RSP + 8

Bits de resultado modificados

Ninguno

Formatos válidos

`RET`

Ejemplo

```
RET
```

6.28. ROL: rotación a la izquierda

```
ROL destino, fuente
```

Lleva a cabo una rotación de los bits del operando destino a la izquierda, es decir, hacia al bit más significativo; rota tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su izquierda; el bit de la posición más significativa pasa a la posición menos significativa del operando.

Por ejemplo, en un operando de 32 bits, el bit 0 pasa a ser el bit 1, el bit 1, a ser el bit 2 y así sucesivamente hasta el bit 30, que pasa a ser el bit 31; el bit más significativo (bit 31) se convierte en el bit menos significativo (bit 0).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascarán los dos bits de más peso del operando fuente, lo que permite rotaciones de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascarán los tres bits de más peso del operando fuente, lo que permite rotaciones de 0 a 31 bits.

Bits de resultado modificados

OF, CF

El bit más significativo se copia en el bit de transporte (CF) cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se activa si el signo del operando destino original es diferente del signo del resultado obtenido; en cualquier otro caso, OF queda indefinido.

Formatos válidos

```
ROL reg, CL
ROL reg, imm8
ROL tamaño mem, CL
ROL tamaño mem, imm8
```

Ejemplos

```
ROL RAX,CL
ROL RAX,1
ROL DWORD [RBX],CL
ROL QWORD [variable],4
```

6.29. ROR: rotación a la derecha

ROR destino, fuente

Realiza una rotación de los bits del operando destino a la derecha, es decir, hacia al bit menos significativo; rota tantos bits como indica el operando fuente.

Los bits pasan desde la posición que ocupan a la posición de su derecha; el bit de la posición menos significativa pasa a la posición más significativa del operando.

Por ejemplo, en un operando de 32 bits, el bit 31 pasa a ser el bit 30, el bit 30, a ser el bit 29, y así sucesivamente hasta el bit 1, que pasa a ser el bit 0; el bit menos significativo (bit 0) se convierte en el bit más significativo (bit 31).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascaran los dos bits de más peso del operando fuente, lo que permite rotaciones de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascaran los tres bits de más peso del operando fuente, lo que permite rotaciones de 0 a 31 bits.

Bits de resultado modificados

OF, CF

El bit menos significativo se copia en el bit de transporte (CF) cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se actualiza con el resultado de la XOR de los dos bits más significativos del resultado; en cualquier otro caso, OF queda indefinido.

Formatos válidos

```
ROR reg, CL
ROR reg, imm8
ROR tamaño mem, CL
ROR tamaño mem, imm8
```

Ejemplos

```
ROR RAX,CL
ROR RAX,1
ROR DWORD [RBX],CL
ROR QWORD [variable],4
```

6.30. SAL: desplazamiento aritmético (o lógico) a la izquierda

```
SAL destino, fuente
```

Lleva a cabo un desplazamiento a la izquierda de los bits del operando destino; desplaza tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su izquierda y se van añadiendo ceros por la derecha; el bit más significativo se traslada al bit de transporte (CF).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascaran los dos bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascaran los tres bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 31 bits.

La operación es equivalente a multiplicar por 2 el valor del operando destino tantas veces como indica el operando fuente.

Operación

destino = destino * 2^N, donde N es el valor del operando fuente

Bits de resultado modificados

OF, SF, ZF, PF, CF

CF recibe el valor del bit más significativo del operando destino cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se activa si el signo del operando destino original es diferente del signo del resultado obtenido; en cualquier otro caso, OF queda indefinido.

El resto de los bits se modifican según el resultado de la operación.

Formatos válidos

```
SAL reg, CL
SAL reg, imm8
SAL tamaño mem, CL
SAL tamaño mem, imm8
```

Ejemplos

```
SAL RAX,CL
SAL RAX,1
SAL DWORD [RBX],CL
SAL QWORD [variable],4
```

6.31. SAR: desplazamiento aritmético a la derecha

SAR destino, fuente

Lleva a cabo un desplazamiento a la derecha de los bits del operando destino; desplaza tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su derecha; el bit de signo (el bit más significativo) se va copiando a las posiciones de la derecha; el bit menos significativo se copia al bit de transporte (CF).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascaran los dos bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascaran los tres bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 31 bits.

La operación es equivalente a dividir por 2 el valor del operando destino tantas veces como indica el operando fuente.

Operación

destino = destino/2^N, donde N es el valor del operando fuente.

Bits de resultado modificados

OF, SF, ZF, PF, CF

Si el operando fuente vale 1, OF se actualiza con el resultado de la XOR de los dos bits más significativos del resultado; en cualquier otro caso, OF queda indefinido.

CF recibe el valor del bit menos significativo del operando destino, cada vez que se desplaza un bit.

El resto de los bits se modifican según el resultado de la operación.

Formatos válidos

```
SAR reg,CL
SAR reg, imm8
SAR tamaño mem, CL
SAR tamaño mem, imm8
```

Ejemplos

```
SAR RAX,CL
SAR RAX,1
SAR DWORD [RBX],CL
SAR QWORD [variable],4
```

6.32. SBB: resta con transporte (*borrow*)

SBB destino, fuente

Lleva a cabo una resta considerando el valor del bit de transporte (CF). Se resta el valor del operando fuente del operando destino, a continuación se resta del resultado el valor de CF y el resultado final de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Operación

destino = destino – fuente – CF

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

El bit de resultado CF toma el valor 1 si el resultado de la operación es negativo; el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
SBB reg, reg
SBB reg, tamaño mem
SBB tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
SBB reg, imm
SBB tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
SBB R9,RAX
SBB RAX, QWORD [variable]
SBB DWORD [variable],EAX
SBB RAX,0x01020304
SBB BYTE [vector+RAX], 5
```

6.33. SHL: desplazamiento lógico a la izquierda

Es equivalente al desplazamiento aritmético a la izquierda (podéis consultar la instrucción SAL).

6.34. SHR: desplazamiento lógico a la derecha

SHR destino, fuente

Realiza un desplazamiento a la derecha de los bits del operando destino; desplaza tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su derecha, el bit menos significativo se copia en el bit de transporte (CF) y se van añadiendo ceros a la izquierda.

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Bits de resultado modificados

OF, SF, ZF, PF, CF

CF recibe el valor del bit menos significativo del operando destino, cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se actualiza con el resultado de la XOR de los dos bits más significativos del resultado; en cualquier otro caso, OF queda indefinido.

El resto de los bits se modifican según el resultado de la operación.

Formatos válidos

```
SHR reg, CL
SHR reg, imm8
SHR tamaño mem, CL
SHR tamaño mem, imm8
```

Ejemplos

```
SHR RAX,CL
SHR RAX,1
SHR DWORD [RBX],CL
SHR QWORD [variable],4
```

6.35. SUB: resta sin transporte

SUB destino, fuente

Lleva a cabo una resta sin considerar el valor del bit de transporte (CF). Se resta el valor del operando fuente del operando destino, el resultado de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Operación

destino = destino – fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

El bit de resultado SF toma el valor 1 si el resultado de la operación es negativo; el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
SUB reg, reg
SUB reg, tamaño mem
SUB tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
SUB reg, imm
SUB tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
SUB R9,RAX
SUB RAX, QWORD [variable]
SUB DWORD [variable],EAX
SUB RAX,0x01020304
SUB BYTE [vector+RAX], 5
```

6.36. TEST: comparación lógica

TEST destino, fuente

Realiza una operación lógica 'y' bit a bit entre los dos operandos sin modificar el valor de ninguno de los operandos; actualiza los bits de resultado según el resultado de la 'y' lógica.

Operación

destino AND fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado CF y OF toman el valor 0, el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
TEST reg, reg
TEST reg, tamaño mem
TEST tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
TEST reg, imm
TEST tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
TEST R9,RAX
TEST RAX, QWORD [variable]
TEST DWORD [variable],EAX
TEST RAX,0x01020304
TEST BYTE [vector+RAX], 5
```

6.37. XCHG: intercambio de operandos

XCHG destino, fuente

Se lleva a cabo un intercambio entre los valores de los dos operandos. El operando destino toma el valor del operando fuente, y el operando fuente toma el valor del operando destino.

No se puede especificar el mismo operando como fuente y destino, ni ninguno de los dos operandos puede ser un valor inmediato.

Bits de resultado modificados

No se modifica ningún bit de resultado.

Formatos válidos

```
XCHG reg, reg
XCHG reg, tamaño mem
XCHG tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

Ejemplos

```
XCHG R9, RAX
XCHG RAX, QWORD [variable]
XCHG DWORD [variable], EAX
```

6.38. XOR: o exclusiva

```
XOR destino, fuente
```

Realiza una operación lógica XOR ('o exclusiva') bit a bit entre el operando destino y el operando fuente; el resultado de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Se lleva a cabo una operación XOR entre el bit *n* del operando destino y el bit *n* del operando fuente según la tabla de verdad de la función XOR:

```
x     y      x XOR y
---   ---    -------
0     0       0
0     1       1
1     0       1
1     1       0
```

Operación

destino = destino XOR fuente

Bits de resultado modificados

OF=0, SF, ZF, PF, CF=0

Los indicadores OF y CF se ponen a 0; el resto de los indicadores se modifican según el resultado de la operación.

Formatos válidos

```
XOR reg,reg
XOR reg,tamaño mem
XOR tamaño mem,reg
```

Los dos operandos han de ser del mismo tamaño.

```
XOR reg,imm
XOR tamaño mem,imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
XOR R9, RAX
XOR RAX, QWORD [variable]
XOR DWORD [variable], EAX
XOR RAX, 01020304 h
XOR BYTE [vector+RAX], 5
```