Arquitectura x86-64 y Lenguaje Ensamblador

Diego Feroldi

Arquitectura del Computador [*]
Departamento de Ciencias de la Computación
FCEIA-UNR

[*]Actualizado 3 de octubre de 2025 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

Notas generales:

- Este apunte de clase resume las principales características de la arquitectura x86-64 y su lenguaje ensamblador. Durante la asignatura, utilizaremos principalmente x86-64, sin importar si el fabricante es Intel o AMD.
- Este apunte no pretende ser una referencia completa del lenguaje ensamblador ni de la arquitectura, sino que debe usarse como material complementario a lo visto en las clases teóricas. Para obtener información más detallada, consultar las referencias. En particular, se recomienda consultar [14] para una información más detallada sobre las instrucciones.
- Para compilar y depurar los ejemplos mostrados en el apunte, consulte los Apéndices 9 y 9.

1. La arquitectura x86-64

La arquitectura x86-64 es una extensión de la arquitectura x86, la cual fue introducida por Intel con el procesador Intel 8086 en 1978 como una arquitectura de 16 bits. Esta arquitectura evolucionó a 32 bits con el lanzamiento del procesador Intel 80386 en 1985, inicialmente conocida como i386 o x86-32, y más tarde como IA-32. Entre 1999 y 2003, AMD amplió esta arquitectura de 32 bits a una de 64 bits, denominándola x86-64 en los primeros documentos, y posteriormente AMD64. Intel adoptó rápidamente las extensiones de AMD bajo los nombres IA-32e o EM64T, y finalmente la llamó Intel 64.

La arquitectura x86-64 (también conocida como AMD64 o Intel 64) de 64 bits ofrece un soporte significativamente mayor para el espacio de direcciones virtuales y físicas. Proporciona registros de propósito general de 64 bits, así como buses de datos y direcciones también de 64 bits, lo que permite que las direcciones de memoria (punteros) sean valores de 64 bits. Aunque cuenta con registros de 64 bits, también permite operaciones con valores de 256, 128, 32, 16 y 8 bits.

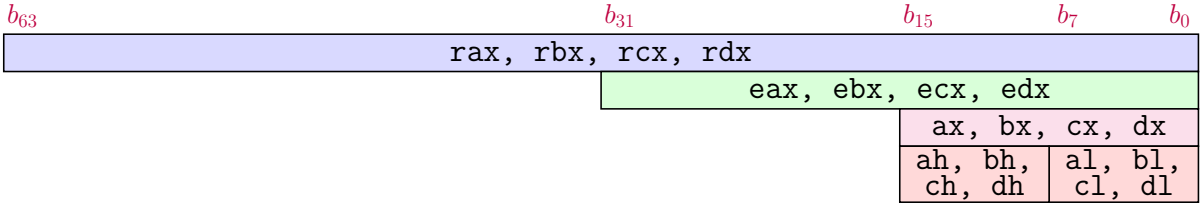
1.1. Registros de propósito general

La arquitectura x86-64 posee 16 registros de propósito general (cada uno de 64 bit): `rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp` y `r8-r15`[1]. 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`) dado que son extensiones de los mismos. En cambio, los registros `r8` al `r15` son registros nuevos. Además, dependiendo de la versión cuenta con registros adicionales para control, punto flotante, etc. Dentro del conjunto de registros disponibles hay algunos que en realidad tienen un uso especial como el `rsp` y el `rbp` que son utilizados para manipular la pila (como veremos en la Sección 6.1).

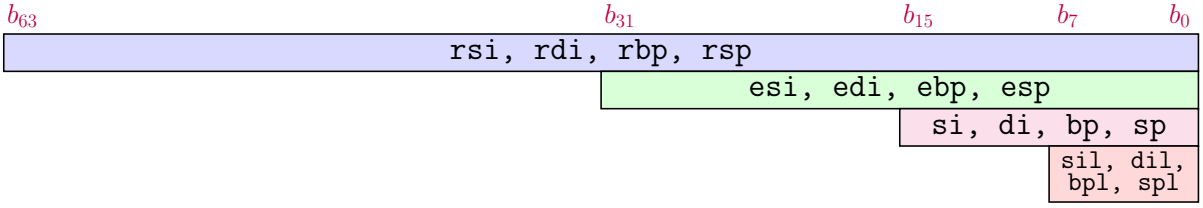
La mayoría de los registros de 64 bits se dividen en subregistros de 32, 16 y 8 bits. Por ejemplo, el registro `rax` de 64 bits contiene en sus 32 bits inferiores al subregistro `eax`

[1]Si bien el registro `rsp` está dentro del grupo de registros de propósito general, veremos en la Sección 6.1 que su uso es bastante particular.

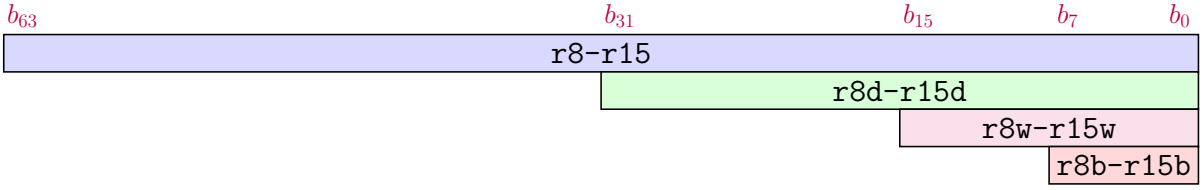
(donde `e` proviene de *extended*), en sus 16 bits inferiores al subregistro `ax`, y este, a su vez, se divide en dos registros de 8 bits: `ah` (por *high*) y `al` (por *low*), como se muestra en la Figura 1a. Por razones históricas, esta última división en registros de 8 bits solo ocurre en `rax`, `rbx`, `rcx` y `rdx`; en el resto, solo existe la parte baja de 8 bits, como se observa en la Figura 1b.



(a) Registros `rax, rbx, rcx, rdx` y sus subregistros.



(b) Registros `rsi, rdi, rbp, rsp` y sus subregistros.



(c) Registros `r8-r15` y sus subregistros.

Figura 1: Registros y subregistros de propósito general en x86-64.

Los registros introducidos en la versión de 64 bits (`r8-r15`) se dividen en un subregistro de 32 bits (por ejemplo, `r8d`, de *doble word*), un subregistro de 16 bits (por ejemplo, `r8w`, de *word*) y un subregistro de 8 bits (por ejemplo, `r8b`, de *byte*), como se muestra en la Figura 1c.

Observaciones

- El registro `rsp` se utiliza para señalar el tope actual de la pila. Aunque forma parte del conjunto de registros de propósito general, no debe emplearse para almacenar datos ni para otros fines. Su uso se analizará en detalle en la Sección 6.1.
- En GDB, los registros de propósito general de x86-64 se pueden ver usando:

  ```
  info registers
  ```

  Si se desea inspeccionar únicamente algunos registros específicos, se puede usar:

  ```
  info register rax rbx rcx
  ```

1.2. Registros especiales

Existen varios registros más que no son de uso general y, por lo tanto, no pueden ser utilizados por las instrucciones habituales.

1.2.1. Puntero de instrucciones

El puntero de instrucciones o contador de programa (en inglés *Instruction Pointer* o *Program Counter*) apunta o guarda la dirección de memoria de la próxima instrucción a ejecutar. En la arquitectura x86-64, este registro se denomina `rip`.

Cada vez que se lee la siguiente instrucción desde la memoria, este registro se actualiza con la dirección de la próxima instrucción a ejecutar. Su contenido también puede modificarse durante la ejecución de una instrucción de ruptura de secuencia, como una llamada a una subrutina o un salto, ya sea condicional o incondicional.

1.2.2. Registros de segmentos

Los registros de segmento contienen los selectores que se utilizan para acceder a los segmentos de memoria. Son seis registros de 16 bits cada uno:

ss (Stack segment): Indica cuál es el segmento utilizado para la pila.

cs (Code Segment): Indica cuál es el segmento de código. En este segmento debe alojarse el código ejecutable del programa. En general este segmento es marcado como sólo lectura.

ds (Data Segment): Indica cuál es el segmento de datos. Allí se alojan los datos del programa (como variables globales).

es, fs, gs: Estos registros tienen un uso especial en algunas instrucciones (las de cadena) y también pueden ser utilizados para referir a uno o más segmentos extras.

Observación

En modo de 64 bits se utiliza un modelo de segmentación plana de la memoria virtual. Esto significa que el espacio de memoria virtual de 64 bits se trata como un único espacio de direcciones plano (no segmentado), lo que reduce la utilidad de los registros de segmentos. El tema de **Segmentación** se abordará por separado cuando estudiemos **Memoria Virtual**.

1.2.3. Registro de banderas

El procesador incluye un registro especial denominado `rflags` o registro de banderas, que refleja el estado del procesador, proporciona información sobre la última operación realizada y contiene ciertos bits de control que permiten modificar su comportamiento. Aunque tiene 64 bits, solo se utiliza la parte baja (bits 0 a 31), equivalente al registro EFLAGS de la arquitectura de 32 bits; la parte alta permanece sin uso.

En la Figura 2 vemos el registro `eflags` (la versión de 32 bits del `rflags`). Los marcados con “S” son bits de estado mientras que los marcados con “C” son de control. Describimos brevemente las banderas más utilizadas:

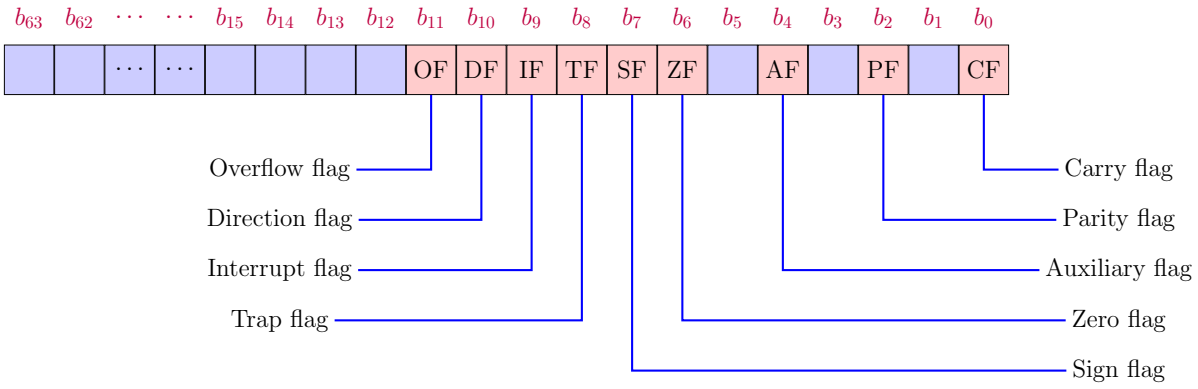


Figura 2: Esquema simplificado del registro `rflags` mostrando las banderas más relevantes.

CF *Carry Flag*: en 1 si la última operación realizó acarreo.

ZF *Zero Flag*: en 1 si la última operación produjo un resultado igual a cero.

OF *Overflow Flag*: en 1 si la última operación desbordó (el resultado no es representable en el operando destino).

SF *Sign Flag*: en 1 si la última operación arrojó un resultado negativo.

DF *Direction Flag*: indica la dirección para instrucciones de manejo de cadenas (que veremos más adelante).

PF *Parity Flag*: en 1 si la cantidad de bits en 1 en los 8 bits menos significativos del resultado de la operación es par. En las operaciones con números en punto flotante además tiene otra interpretación que veremos en la Sección 8.

El resto de los bits no se utilizan o corresponden a banderas más específicas que no son relevantes para esta asignatura.

Observaciones

- El registro `rflags` no es de propósito general por lo cual no puede ser accedido ni modificado por instrucciones regulares (`add`, `mov`, etc) de manera directa. En cambio, sí puede ser modificado de manera indirecta por instrucciones tales como las de comparación, aritméticas, etc. Es decir, las banderas del registro `rflags` se modifican como resultado de ciertas instrucciones, lo cual veremos más adelante que tiene importantes utilidades.
- En GDB, podemos ver el contenido del registro de flags usando el comando `info registers eflags`, que muestra su valor en hexadecimal y permite interpretar cada bandera de estado individualmente; aunque el registro sea de 64 bits, GDB lo lista con el nombre `eflags`.

1.3. Registros SSE

La arquitectura x86-64 proporciona, además, 16 registros de 128 bits (`xmm0`–`xmm15`), conocidos como registros SSE (*Streaming SIMD Extensions*), donde SIMD significa *Single Instruction, Multiple Data*. Permiten procesar múltiples datos en paralelo, lo que resulta especialmente útil en aplicaciones de gráficos, multimedia, cómputo científico y optimización de rendimiento. Estos registros forman parte de las extensiones SSE introducidas para mejorar el procesamiento vectorial.

La Figura 3 muestra la estructura de los registros SSE y los distintos tipos de datos que puede contener un registro `xmm`, tanto de tipo punto flotante como de tipo entero. Vemos, por ejemplo, que un registro `xmm` puede almacenar un vector de cuatro `float` de 32 bits o un vector de dos `quadword` de 64 bits.

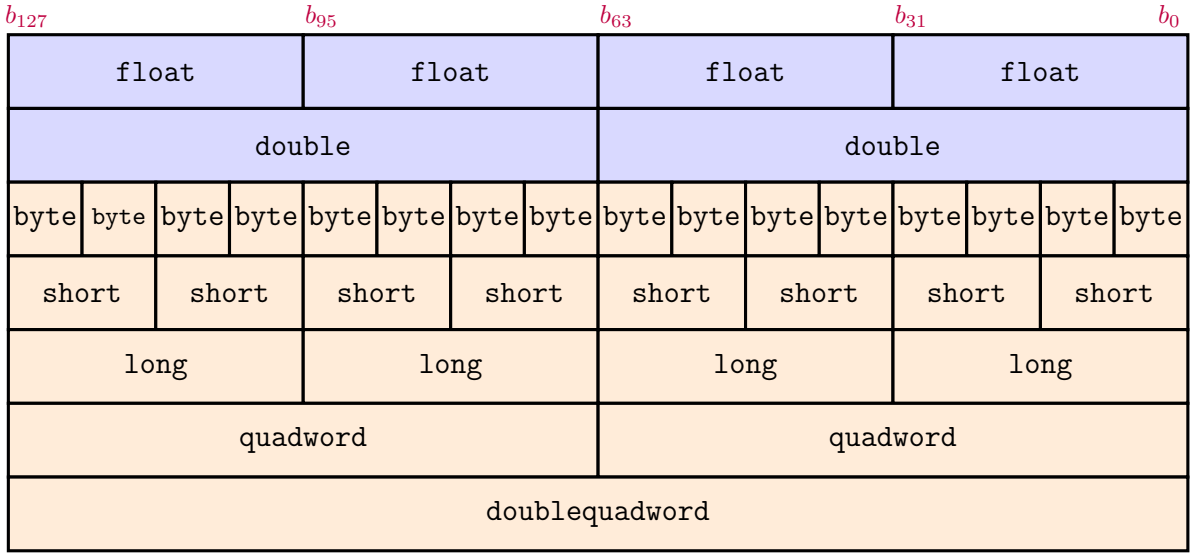


Figura 3: Registros vectoriales SSE (`xmm0`–`xmm15`): estructura y características de sus 128 bits.

Intel AVX (*Advanced Vector Extensions*) proporciona, además, 16 registros AVX de 256 bits de ancho (`ymm0-ymm15`). Los 128 bits inferiores de `ymm0-ymm15` actúan como alias de los respectivos registros SSE de 128 bits (`xmm0-xmm15`). La utilidad de los registros SSE se analizará con más detalle en la Sección 8.

1.4. Lenguaje de máquina

Los procesadores son dispositivos de hardware encargados de ejecutar el programa alojado en memoria. Actualmente, un programador escribe el código en un lenguaje de programación de alto nivel, como C, Java, Haskell, entre otros. Sin embargo, la CPU no ejecuta directamente el programa escrito en ese lenguaje, sino que este debe ser traducido (o compilado) a **lenguaje de máquina**.

El lenguaje de máquina (o código de máquina) está compuesto por secuencias de bytes que codifican operaciones de bajo nivel destinadas a manipular datos, administrar la memoria, interactuar con dispositivos de almacenamiento y comunicarse a través de redes. Un compilador genera este código a través de una serie de etapas, guiadas por las reglas del lenguaje de programación, el conjunto de instrucciones de la máquina de destino y las convenciones impuestas por el sistema operativo.

Sin embargo, el lenguaje de máquina resulta muy críptico para las personas. Por ejemplo, la secuencia `10001001 11111000` copia el valor del registro `edi` al registro `eax`. Para facilitar la tarea de programación en los años 50, se introdujo el lenguaje ensamblador, que ofrece una representación más legible para los humanos.

El **lenguaje ensamblador** es una forma simbólica del lenguaje de máquina que permite escribir instrucciones mediante mnemónicos legibles por humanos, en lugar de secuencias de bits. Por ejemplo, la instrucción

```
movl %edi, %eax
```

es equivalente a la instrucción binaria `10001001 11111000`, que copia el valor del registro `edi` al registro `eax`. Claramente, esta representación es mucho más legible para una persona.

Ejemplo

El siguiente fragmento de código realiza la suma de dos enteros:

```
0x0000000000001125 <+0>:  89 f8   movl %edi, %eax
0x0000000000001127 <+2>:  01 f0   addl %esi, %eax
0x0000000000001129 <+4>:  c3      ret
```

En la columna izquierda se muestran las direcciones de memoria; en la columna central, el código en lenguaje de máquina (en formato hexadecimal, para mayor compacidad); y en la columna derecha, su equivalente en lenguaje ensamblador x86-64 (sintaxis AT&T).

Si bien todavía no sabemos cómo utilizar estas instrucciones, ya podemos observar que el lenguaje ensamblador resulta mucho más comprensible que el lenguaje de máquina. De hecho, presenta una sintaxis basada en una operación seguida de sus argumentos, donde las operaciones —llamadas instrucciones— tienen nombres representativos (por ejemplo, `movl` para "mover" y `addl` para "sumar"). Más adelante analizaremos el significado de cada instrucción de ensamblador y cómo se utiliza.

La Figura 4 ilustra el proceso de traducción desde un código fuente —en este caso, escrito en C— hasta su forma binaria en lenguaje de máquina. Aunque este proceso se presenta en dos pasos, algunos compiladores eliminan el paso intermedio y generan directamente el código binario ejecutable.

2. Lenguaje Ensamblador de x86-64

El lenguaje ensamblador es específico de cada máquina. Por ejemplo, el código escrito para un procesador x86-64 no se ejecutará en un procesador diferente, como un procesador ARM, que es popular en tabletas y teléfonos inteligentes. El lenguaje ensamblador es un lenguaje de "bajo nivel" y proporciona la interfaz de instrucción básica para el procesador de la computadora. Para un programador, el ensamblador es lo más cercano al procesador. Los programas escritos en un lenguaje de alto nivel se traducen al lenguaje ensamblador para que el procesador pueda ejecutarlos. En este sentido, el lenguaje de alto nivel actúa como una abstracción entre el código y las instrucciones reales del procesador.

El ensamblador ofrece al programador control directo sobre los recursos del sistema, lo que implica configurar registros del procesador, acceder a ubicaciones de memoria e

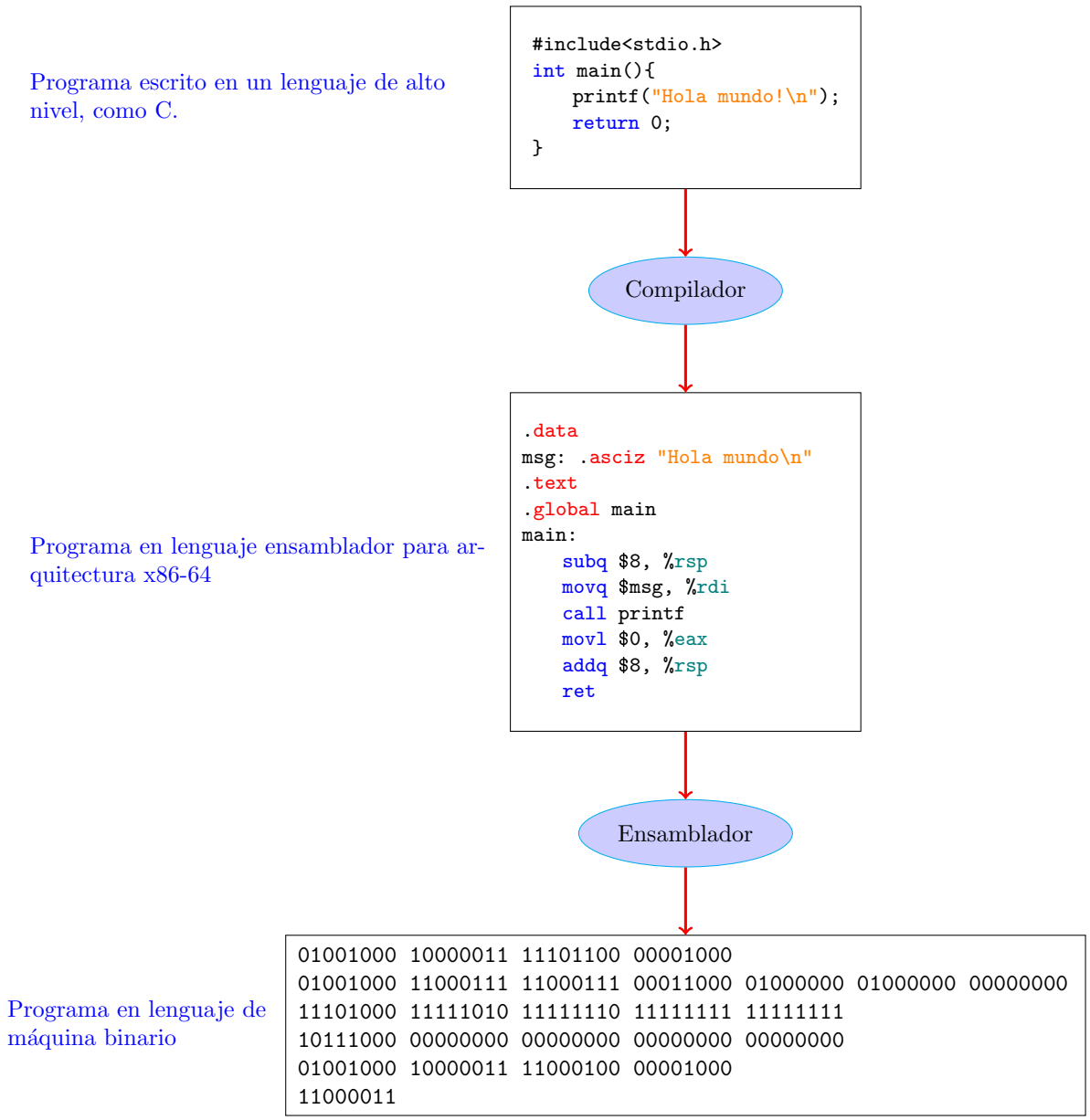


Figura 4: Programa en C compilado a lenguaje ensamblador y luego ensamblado a lenguaje de máquina binario.

interactuar con otros componentes de hardware. Esto requiere una comprensión profunda del funcionamiento del procesador y la memoria.

Al programar en un lenguaje de alto nivel como C, no es necesario preocuparse por la implementación detallada del programa a nivel de máquina. En cambio, al escribir programas en código ensamblador, como se hacía en los primeros días de la informática, el programador debe especificar las instrucciones de bajo nivel que el programa utilizará para realizar un cálculo. La mayoría de las veces, resulta mucho más productivo y confiable trabajar en un nivel más alto de abstracción proporcionado por un lenguaje de alto nivel. La verificación de tipos realizada por un compilador ayuda a detectar muchos errores de programación y asegura que los datos se manejen de manera consistente.

Con los compiladores y optimizadores modernos, el código generado suele ser tan eficiente como el que escribiría manualmente un programador experto en ensamblador.

Además, un programa escrito en un lenguaje de alto nivel puede compilarse y ejecutarse en múltiples plataformas, mientras que el código ensamblador es altamente específico de la máquina para la que fue desarrollado.

Entonces, ¿por qué deberíamos aprender lenguaje ensamblador? Este lenguaje ofrece varios beneficios:

- **Velocidad**: Los programas en lenguaje ensamblador son generalmente los programas más rápidos.
- **Espacio**: Los programas en lenguaje ensamblador suelen ser los más pequeños.
- **Capacidad**: Se pueden hacer cosas en lenguaje ensamblador que son difíciles o imposibles en lenguajes de alto nivel.
- **Conocimiento**: El conocimiento del lenguaje ensamblador ayuda a escribir mejores programas, incluso cuando se use lenguajes de alto nivel.

Aunque los compiladores hacen la mayor parte del trabajo en la generación de código ensamblador, poder leerlo y comprenderlo es una habilidad importante para los programadores avanzados. Al invocar al compilador con los parámetros de línea de comandos apropiados, el compilador generará un archivo que muestra su salida en forma de código de ensamblaje. Al leer este código, podemos entender las capacidades de optimización del compilador y analizar las ineficiencias subyacentes en el código.

Los programadores que buscan maximizar el rendimiento de una sección crítica de código a menudo prueban diferentes variaciones del código fuente, cada vez que compilan y examinan el código de ensamblaje generado para tener una idea de qué tan eficientemente se ejecutará el programa. Además, hay momentos en que la capa de abstracción proporcionada por un lenguaje de alto nivel oculta información sobre el comportamiento en tiempo de ejecución de un programa que debemos comprender. Por ejemplo, cuando se escriben programas concurrentes utilizando un paquete de subprocesos, es importante saber qué región de la memoria se usa para contener las diferentes variables del programa. Esta información es visible a nivel de código ensamblador.

Otro ejemplo es que muchas de las formas en que se pueden atacar los programas, permitiendo que gusanos y virus infesten un sistema, implican matices de la forma en que los programas almacenan la información en tiempo de ejecución. Muchos ataques implican la explotación de las debilidades en los programas del sistema para sobrescribir la información y, por lo tanto, tomar el control del sistema. Comprender cómo surgen estas vulnerabilidades y cómo protegerse de ellas requiere un conocimiento de la representación de los programas a nivel de máquina.

La necesidad de que los programadores aprendan lenguaje ensamblador ha cambiado con el paso del tiempo: de ser fundamental para escribir programas directamente en ensamblador, ha pasado a ser importante principalmente para leer y comprender el código generado por los compiladores. En definitiva, la razón principal para aprender lenguaje ensamblador radica en comprender cómo funciona una computadora.

2.1. Características principales

En esta sección detallamos las principales características de la sintaxis de lenguaje ensamblador x86-64 con sintaxis AT&T a modo de presentación. Luego, a lo largo del apunte iremos profundizando sobre estas características.

- En general, las instrucciones tienen dos operandos y se escriben de la siguiente manera:

  `operadorS <operando origen>, <operando destino>`

  donde `S` es el sufijo de tamaño mencionado anteriormente. No obstante, también existen instrucciones con un solo operando, sin operandos o incluso con tres operandos.

- Los operandos en una instrucción pueden ser:
  - Registros
  - Direcciones de memoria
  - Valores inmediatos

- El nombre de los registros comienza con `%`. Por ejemplo, el registro `rax` se escribe como `%rax`.

- Los comentarios de línea comienzan con `#` (a partir de `#` comienza un comentario hasta el fin de línea).

- Las constantes (valores inmediatos) se prefijan con `$`. Así, la constante 5 se escribe como `$5`. Un caso particular que veremos luego son las etiquetas.

- Las direcciones de memoria se escriben sin ninguna decoración. Por lo tanto, la expresión `3000` refiere a la dirección de memoria 3000, y no a la constante 3000, que —como se explicó antes— se escribiría como `$3000`.

- Las instrucciones que manipulan datos (ya sea en registros o en memoria) se escriben con un sufijo que indica el tamaño del dato. Por ejemplo, al agregar el sufijo `q` a la instrucción `mov`, se obtiene `movq`.

  Los sufijos posibles son los siguientes:

Sufijo	**Denominación**	**Declaración** (en `.data`)	**Tamaño** (bytes)	**Equivalente en C**	**GDB**
`b`	*Byte*	`.byte`	1	`char`	`b`
`w`	*Word*	`.word` o `.short`	2	`short`	`h`
`l`	*Double word*	`.long`	4	`int`	`w`
`q`	*Quad word*	`.quad`	8	`long int`	`g`
`s`	*Single precision float*	`.float`	4	`float`	`w`
`d`	*Double precision float*	`.double`	8	`double`	`g`

  En el ensamblador de GNU (as) este sufijo es opcional cuando el tamaño de los operandos puede ser deducido, aunque es conveniente escribirlo siempre para detectar posibles errores.

Observación

En este apunte utilizaremos la sintaxis de AT&T de lenguaje ensamblador ya que es la utilizada por defecto en GNU Assembler (GAS). Si bien originalmente GCC solo trabajaba con la sistaxis AT&T, actualmente se puede trabajar también con la sintaxis Intel utilizando la bandera `-masm=intel` al momento de compilar.

Las principales diferencias entre ambas sintaxis son las siguientes:

	Intel	AT&T
Orden de los operandos	destino ← origen	origen → destino
Comentarios	;	#
Operadores	Sin sufijo: `add`	Con sufijo: `addq`
Registros	`eax, ebx`, etc.	`%eax, %ebx`, etc.
Valores inmediatos	`0x100`	`$0x100`
Direccionamiento indirecto	`[eax]`	`(%eax)`
Direc. (forma general)	`[base+(índice*scale)+K]`	`K(base, índice, scale)`

Como ejemplo comparativo la instrucción `addq %ebx, %eax` en sintaxis AT&T es equivalente a la instrucción `add eax, ebx` en sintaxis Intel.

2.2. Directivas al Ensamblador

Las instrucciones y los datos no son los únicos elementos que componen un programa en lenguaje ensamblador. Los ensambladores reservan palabras clave especiales para instruir al ensamblador sobre cómo realizar funciones especiales a medida que los mnemotécnicos se convierten en códigos de instrucción. Las directivas al compilador ensamblador comienzan siempre con ".".

Dentro de las directivas destacamos las siguientes:

Describir el segmento: Con las directivas de segmento el programador indica a **qué** segmento debe agregarse el siguiente bloque. Las más comunes son `.data` para datos inicializados (indicando que el siguiente bloque debe ir al segmento de datos) y `.text` (indicando que lo que sigue es código ejecutable). También existe el segmento `.bss` para los datos no inicializados.

Ejemplos

```
.data
(Desde este punto, un tramo con datos sin inicializar.)
.bss
(Desde este punto, un tramo con datos sin inicializar.)
.text
(Desde este punto, un tramo con datos sin inicializar.)
```

Inicializar valores: Esta clase de directivas emite valores constantes indicados por el programador directamente en el bloque, es decir no se hace traducción. Dentro de esta clase tenemos:

asciz, ascii Permiten inicializar una lista de cadenas con y sin carácter nulo al final de cada una.

Ejemplos

```
.asciz "Hola mundo"
.ascii "abcde"
.string "abcde"
```

En el primer ejemplo se almacena la cadena `Hola mundo\000` (11 caracteres, con el carácter nulo al final). El segundo ejemplo es una cadena de caracteres sin el cero final. El tercer ejemplo es equivalente al segundo.

byte Inicializa una lista de bytes.

Ejemplos

```
.byte 'a', 'b'
.byte 97
.byte 0x61
```

El primer ejemplo es un arreglo de bytes.

double, float Inicializa una lista de valores de punto flotante de doble y simple precisión, respectivamente.

Ejemplos

```
.double 3.1415, 2.16
.float 5.3
```

El primer ejemplo es un arreglo de *doubles*.

short, long, quad Emite una lista de valores enteros de 2, 4 y 8 bytes, respectivamente.

Ejemplos

```
.short 20, 30, 40, 50
.long 50, 200
.quad 0, 0xff, 0xbeef
```

space Emite un bloque de tamaño fijo inicializado en cero o en un valor pasado como argumento.

Ejemplos

```
.space 128
.space 5000, 0
.zero 5000
```

En el primer ejemplo se reservó un bloque de memoria de 128 bytes pero no está inicializado (puede tener cualquier valor). En el segundo ejemplo se reservó un bloque de 5000 bytes y está inicializado en 0. El tercer ejemplo es equivalente al segundo.

Esta directiva es útil para obtener un bloque de memoria de tamaño dado (ya sea inicializado o no).

Observación

Es importante notar que todas estas directivas toman como argumento una **lista** de valores a inicializar. Un error muy común es no indicar ningún elemento en esa lista, por ejemplo:

```
.long
```

lo cual **NO** reserva espacio. La versión correcta sería `.long 0` o, alternativamente,

```
.space 8
```

2.3. Etiquetas

Las etiquetas son una parte fundamental del lenguaje ensamblador, ya que hacen referencia a elementos dentro del programa. Su función principal es facilitar al programador la tarea de referenciar diferentes partes del programa, como constantes, variables o posiciones del código, que se utilizan como operandos en las instrucciones o directivas.

Por ejemplo, cuando se define una variable en C (`long i;`), se le indica al compilador que reserve un espacio de memoria para un entero y que este espacio se referenciará mediante el identificador `i`. Tanto en C como en ensamblador, nombrar un espacio de memoria es útil para el programador, pero esta información no es utilizada directamente por la computadora; en su lugar, una etiqueta se convierte en una **dirección de memoria**.

En ensamblador con sintaxis AT&T una etiqueta es un nombre seguido del símbolo “:”.

Ejemplos

```
a: .quad 126
b: .long 455
c: .byte 0x22
```

```
f: .float 3.14
g: .double 0.1
msg: .ascii "Hola mundo\n"
```

En este ejemplo se crean variables de distintos tipos. Por ejemplo, una variable de tipo `quad` (8 bytes) se inicializa con el valor 126 y se almacena en una dirección de memoria etiquetada como `a`.

2.4. Etiquetas globales

La directiva `.global` indica que la etiqueta nombrada a continuación es de alcance global.

Ejemplos

```
.global main
.global sum
```

De no especificar esta directiva la etiqueta desaparece luego del proceso de compilación. Las etiquetas globales deben ser utilizadas, por ejemplo, con las etiquetas que definan funciones que serán llamadas fuera del archivo compilado. Por ejemplo, cuando se enlaza un programa C con uno escrito en ensamblador, las funciones incluidas en ensamblador deben ser definidas como globales (siendo `main` el caso más común). Esto se verá en detalle en el Apéndice 9.

2.5. Operandos inmediatos

En ciertas instrucciones, un operando de origen, llamado operando inmediato, se incluye como parte de la instrucción en lugar de acceder a él desde un registro o una ubicación de memoria. En la sintaxis que veremos en este apunte (AT&T) cada operando inmediato debe ir precedido por un signo peso para indicar que es un valor inmediato. Los valores también se pueden expresar en varios formatos diferentes, siendo el formato decimal y el formato hexadecimal los más usuales. Estos valores no se pueden cambiar después de que el programa es ensamblado y *linkeado* en el archivo de programa ejecutable. En el modo de 64 bits, el tamaño máximo de un operando inmediato es de 32 bits, excepto en la instrucción `mov`, que puede copiar un inmediato de 64 bits en un registro de propósito general.

Ejemplos

```
movl $0, %eax          # copia el valor 0 al registro eax
movb $0x80, %bl        # copia el valor hexadecimal 80 al registro bl
addb $0xff, %ah        # suma 0xff con el valor en ah y lo guarda en ah
movl $0x11223344, %eax      # copia el valor 0x11223344 al registro eax
```

```
movq $0x1122334455667788, %rax      # copia el valor 0x1122334455667788 a
    el registro rax
```

Observación

Es interesante ver el equivalente en lenguaje de máquina de la última instrucción del ejemplo anterior. Esto se puede lograr utilizando GDB, con el comando `disassemble/r`. Obtenemos el siguiente equivalente en lenguaje de máquina (en formato hexadecimal):

```
48 b8 88 77 66 55 44 33 22 11
```

Aquí se ve de manera explícita que el valor inmediato está contenido dentro de la propia instrucción. El motivo por el cual se ve invertido lo veremos en detalle en la Sección 5.2.

2.6. Formato general de las instrucciones

Como vimos previamente, las instrucciones de ensamblador en la arquitectura x86-64 están compuestas por un operador (por ejemplo, suma, resta, comparación, etc.) acompañada de operandos (por ejemplo, valores a sumar):

```
operadorS <operando origen>, <operando destino>
```

donde `S` es el sufijo de tamaño mencionado anteriormente.

En algunos casos, las instrucciones no requieren operandos o bien utilizan operandos implícitos. Por ejemplo, la instrucción `ret` no toma operandos, mientras que `inc` solo toma un operando y lo incrementa en uno (el uno está implícito).

El juego de instrucciones de los procesadores x86-64 es muy amplio y en esta sección veremos las principales instrucciones para operar con valores enteros. Luego, en la Sección 8 se verán la instrucciones para operar con datos en punto flotante. Para más información sobre las instrucciones en x86-64, véase [14].

2.7. Instrucciones de transferencia de datos

Una operación muy común es la de copiar valores de un lugar a otro. Un programa debe intercambiar valores con la memoria, registros, etc. La arquitectura x86-64 ofrece varias instrucciones para hacer copias de datos siendo la más importante la instrucción `mov`.

2.7.1. Instrucción `MOV`

La instrucción `mov` es la instrucción genérica para copiar un dato desde un origen a un destino. Esta instrucción toma la forma

```
movS <operando origen>, <operando destino>
```

donde "S" es el sufijo que indica el tamaño de los operandos (que deben ser del mismo tamaño) según lo visto en la Tabla de la página 9.

Observación

El operando destino es el argumento de la derecha, por lo que la instrucción

```
movq %rax, %rbx
```

representa `rax` → `rbx`. Es decir, copia el valor de `rax` a `rbx`. Después de ejecutar la instrucción, el valor de `rbx` será igual al de `rax`. Es importante destacar que el valor del registro `rax` permanece sin cambios. En realidad, más que un movimiento, es una copia de datos.

El operando origen puede ser un valor inmediato, un registro de propósito general o un valor en memoria. El operando destino puede ser un registro de propósito general o un valor en memoria. Los dos operandos no pueden ser valores de memoria. Por lo tanto, a continuación podemos observar las diferentes formas que puede tomar la instrucción:

```
movS <registro>, <registro>
movS <memoria>, <registro>
movS <registro>, <memoria>
movS <valor immediato>, <memoria>
movS <valor immediato>, <registro>
```

Ejemplos

```
movb $65, %al              # al='A'
movq %rax, %rcx            # rcx=rax
movw (%rax), %dx           # Copia en dx dos bytes comenzando en la
                           # dirección guardada en rax.
movw %dx, (%rax)           # Copia dx en la dirección guardada en rax.
movl 16(%rbp), %ecx        # Copia en ecx cuatro bytes (debido al
                           # sufijo l) comenzando en la dirección rbp+16.
movb $45, a                # Copia el valor 45 en la dirección de memoria
                           # con etiqueta a
```

Nota: Algunos de estos ejemplos se comprenderán mejor luego de ver la Sección 5.

Observación

La relación entre subregistros de diferentes tamaños es la siguiente:

1. La carga de un valor en un subregistro de 32 bits establece los 32 bits superiores del registro en cero. Por ejemplo, después de `movl $-1, %eax`, el registro `%rax` tiene el valor `0x00000000ffffffff`, independientemente del valor anterior.

2. La carga de un valor en un subregistro de 16 u 8 bits deja todos los demás bits sin cambios. Por ejemplo, si el valor de `%rax` es `0xffffffffffffffff`, luego de `movw $0, %ax` el registro `%rax` tiene el valor `0xffffffffffff0000`.

Esto puede parecer un poco arbitrario pero es así por una cuestión de compatibilidades a medida que fueron apareciendo procesadores con registros con mayor cantidad de bits.

2.7.2. Instrucción PUSH

La instrucción `pushS` tiene la forma:

```
pushS <operando fuente>
```

y produce dos efectos:

1. decrementa el registro `rsp` en una cantidad de bytes de acuerdo al sufijo `S`,
2. mueve el operando de la instrucción a la dirección apuntada por el registro `rsp` luego de que dicho registro es decrementado.

Ejemplo

```
movq $45, %rax    # rsp=0x7fffffffebc8
pushq %rax        # rsp=0x7fffffffebc0
```

La instrucción `pushq` decrementa primero el registro `rsp` en 8 y luego almacena el valor contenido en `rax` en la dirección de memoria apuntada por `rsp`.

2.7.3. Instrucción POP

Esta instrucción tiene la forma:

```
popS <operando destino>
```

y tiene dos efectos:

1. copia el dato apuntado por el registro `rsp` al operando destino,
2. incrementa el valor de `rsp` en una cantidad de bytes de acuerdo al sufijo `S`.

Esta instrucción es la instrucción opuesta a la `push`.

Ejemplo

```
popq %rax     # Guarda en el registro rax el valor contenido en la
   dirección de memoria apuntada por el registro rsp.
```

`rsp = 0x7fffffffebc8` luego de ser ejecutada, si antes era `rsp = 0x7fffffffebc0` y `rax = 45`, continuando con el ejemplo anterior.

Observaciones

- Las instrucciones `push` y `pop` pueden interpretarse como instrucciones complementarias. Es decir, en general se las utiliza de manera conjunta en concordancia.
- En la arquitectura X86-64 las instrucciones `push` y `pop` solo admiten los sufijos `w` y `q`. Sin embargo, lo usual es solo usar dichas instrucciones con el sufijo `q` para mantener la alineación de la pila.
- En la Sección 6.1 veremos en detalle cómo se utilizan las instrucciones `push` y `pop` para manejar la "pila".

2.7.4. Instrucción XCHG

Esta instrucción intercambia el contenido de los operandos:

```
xchgS <operando fuente>, <operando destino>
```

Ejemplo

```
movq $34, %rax
movq $0xf3fa, %rbx
xchgq %rax, %rbx     # rax=0xf3fa y rbx=34
```

2.8. Instrucciones aritméticas

La familia de procesadores x86 ofrece múltiples instrucciones para realizar operaciones numéricas, entre ellas:

2.8.1. Instrucción ADD

Esta instrucción realiza la suma aritmética de los dos operandos:

```
addS <operando fuente>, <operando destino>
```

El resultado queda en el operando destino:

```
<operando destino=operando fuente + operando destino>
```

Ejemplo

```
movb $5, %al
movb $4, %bl
addb %al, %bl     # al=5 y bl=9
```

La instrucción `add` realiza la suma entera. Notar que evalúa el resultado tanto para la operación sin signo como con signo y establece las banderas `CF` y `OF` para indicar si el resultado es correcto. La bandera `SF` indica el signo del resultado signado y la bandera `ZF` si el resultado es nulo.

Ejemplo

```
movb $45, %al
addb $100, %al
```

Luego de realizarse la suma resulta `al=0x91=-111`, `SF=1`, `CF=0` y `OF=1`. El estado de las banderas indica que el resultado de la suma con los operando interpretados como números con signo es negativo y es incorrecto. En cambio, si los operandos se toman como números sin signo, el resultado es correcto (`0x91=145`).

2.8.2. Instrucción ADC

Esta instrucción realiza la suma aritmética de los dos operandos más el bit de acarreo (`CF` del `rflags`):

```
adcS <operando fuente>, <operando destino>
```

Resulta:

```
<operando destino=operando fuente + operando destino + acarreo>
```

Ejemplo

```
movb $0, %dl
movb $0xff, %al
addb $0xff, %al   # al=0xfe, CF=1
adcb $0, %dl      # dl=1
```

2.8.3. Instrucción SUB

Esta instrucción realiza la resta aritmética de los dos operandos:

```
subS <operando fuente>, <operando destino>
```

Realiza la resta:

```
operando destino = operando destino - operando fuente.
```

Ejemplo

```
movq $45, %rbx
movq $23, %rax
subq %rax, %rbx    # rbx=22
```

La instrucción `sub` realiza la resta entera. Notar que evalúa el resultado tanto para la operación sin signo como con signo y establece las banderas `CF` y `OF` para indicar si el resultado es correcto. La bandera `SF` indica el signo del resultado signado y la bandera `ZF` si el resultado es nulo.

2.8.4. Instrucción SBB

Resta aritmética de los dos operandos considerando el bit de acarreo:

```
sbbS <operando fuente>, <operando destino>
```

Resulta:

```
operando destino = operando destino - operando fuente - acarreo
```

Ejemplo

```
movl $1, %edx
movl $0, %eax
subl $1, %eax   # CF=1.
sbbl $0, %edx   # edx=0
```

2.8.5. Instrucción INC

Incrementa el operando en una unidad (`operando=operando+1`):

```
incS <operando>
```

Ejemplo

```
movq $56, %rax
incq %rax         # rax=57
```

Esta instrucción es equivalente a `addq $1, %rax`. Sin embargo, la instrucción `inc` no modifica el valor de la bandera `CF`. El resto de las banderas son modificadas de acuerdo al resultado.

2.8.6. Instrucción DEC

Decrementa el operando en una unidad:

```
decS <operando>
```

Ejemplo

```
movq $45, %rax
decq %rax         # rax=44
```

Esta instrucción es equivalente a `subq $1, %rax`. Sin embargo, la instrucción `dec` no modifica el valor de la bandera `CF`. El resto de las banderas son modificadas de acuerdo al resultado.

2.8.7. Instrucción IMUL

La instrucción `IMUL` realiza la multiplicación entera con signo. La instrucción `imul` tiene tres formatos:

Con un operando: `imulS <operando>`

El formato con un operando utiliza los registros `rax` y `rdx` (o una parte) de forma implícita. Es decir, si el operando es de 64 bits multiplica el valor del operando con `rax` y el resultado queda en `rdx:rax`. Notar que el resultado es de 128 bits y los 64 bits menos significativos quedan en `rax` mientras que los 64 más significativos en `rdx`.

De manera análoga, se puede trabajar con operandos de 32 y 16 bits. Es decir, si se multiplica el valor de un operando de 32 con `eax`, el resultado queda en `edx:eax`. Si se multiplica el valor de un operando de 16 con `ax`, el resultado queda en `dx:ax`. Sin embargo, si se multiplica el valor de un operando de 8 bits con `al`, el resultado queda en `ah:al`.

El operando puede ser un registro o una dirección de memoria, pero no un valor inmediato.

Con dos operandos: `imulS <operando fuente>, <operando destino>`

En este formato el operando destino es multiplicado por el operando fuente. El operando destino debe ser un registro de propósito general, mientras que el operando fuente puede ser un registro de propósito general, un valor inmediato o un valor en memoria. En este formato no existen registros implícitos, y el resultado intermedio (de un tamaño doble al del operando fuente) se trunca y se guarda en el operando destino.

Con tres operandos: `imulS <op. fuente 1>, <op. fuente 2>, <op. destino>`

Este formato requiere dos operandos fuentes y un operando destino. El segundo operando fuente (que puede ser un registro de propósito general o un valor en memoria) es multiplicado por el primer operando fuente (un valor inmediato). El resultado intermedio (el doble de tamaño que el operando fuente) es truncado y guardado en el operando destino (un registro).

Ejemplos

```
movq $9, %rax
movq $-3, %rbx
imulq %rbx                      # rax=0xfffffffffffffe5 (-27)
                                # rdx=0xffffffffffffffff (-1)

movq $9, %rax
```

```
imulq %rbx, %rax                # rax=0xfffffffffffffe5 (-27)
imulq $4, %rax                  # rax=0xfffffffffffff94 (-108)
imulq $2, %rax, %rbx            # rbx=0xfffffffffffff28 (-216)
movq $0x7ffffffffffffe, %rax    # rax=9223372036854775806
imulq $2, %rax                  # rax=0xfffffffffffffc (-4)
```

Notar que en la última multiplicación el resultado es erróneo dado que el verdadero resultado (1.8447×10^{19}) no entra en 64 bits.

Las tres formas de la instrucción `IMUL` son similares en que la longitud del producto se calcula al doble de la longitud de los operandos. En la forma de un operando, el producto se almacena exactamente en el destino. Sin embargo, en las formas de dos y tres operandos, el resultado se trunca a la longitud del destino antes de ser almacenado en el registro de destino. Debido a este truncamiento, las banderas CF u OF deben ser verificadas para asegurar que no se pierdan bits significativos.

Observación

En todas las variantes de imul, las banderas CF y OF indican desbordamiento aritmético con signo.

- En la forma de un operando, se activan si la parte alta del resultado no es una extensión por signo de la parte baja.
- En las formas de dos y tres operandos, se activan si el resultado no puede representarse con signo en el tamaño del operando destino.
- En todos los casos, CF y OF siempre tienen el mismo valor, mientras que las demás banderas quedan indefinidas.

2.8.8. Instrucción `MUL`

La instrucción `MUL` realiza la multiplicación entera sin signo. Esta instrucción a diferencia de la `IMUL` solo admite el formato con un operando:

```
mulS <operando>
```

Ejemplo

```
movq $0xffffffffffffffff, %rax
movq $4, %rbx
mulq %rbx                       # rax=0xfffffffffffffffc y rdx=3
```

De manera similar a la instrucción `IMUL` con un solo operando, el resultado se distribuye entre dos registros (`rdx` y `rax`) o sus respectivos subregistros, según el tamaño de los operandos. Si los bits de orden superior del producto son 0, las banderas CF y OF se borran; de lo contrario, las banderas se establecen.

2.8.9. Instrucción IDIV

La instrucción `idiv` realiza la división entera con signo:

```
idivS <operando divisor>
```

La instrucción `idiv` en su versión de 64 bits divide el contenido del entero de 128 bits `rdx:rax` (construido interpretando a `rdx` como los ocho bytes más significativos y a `rax` como los ocho bytes menos significativos) por el valor del operando especificado. El resultado del cociente de la división se almacena en `rax`, mientras que el resto se coloca en `rdx`[2].

Ejemplo

```
movq $0xffff, %rax   # rax = 65535
movq $0, %rdx
movq $-1024, %rbx
idivq %rbx           # rax=0xfffffffffffffc1 (-63) y rdx=0x3ff (1023)
```

El resultado entero es -63 y el resto es 1023.

2.8.10. Instrucción DIV

La instrucción `div` realiza la división entera sin signo:

```
divS <operando divisor>
```

Ejemplo

```
movq $0xffff, %rax    # rax = 65535
movq $0, %rdx
movq $1024, %rbx
divq %rbx             # rax=0x3f (63) y rdx=0x3ff (1023)
```

El resultado entero es 63 y el resto es 1023.

2.8.11. Instrucción NEG

La instrucción `neg` realiza la negación aritmética en complemento a 2:

```
negS <operando>
```

Ejemplo

```
movb $0xff, %al
negb %al           # al=1
```

[2]Esta instrucción también admite operandos de otros tamaños. Para mayor información ver [14]

2.9. Instrucciones lógicas

2.9.1. Instrucción AND

Operación `and` lógica bit a bit.

Ejemplo

```
movw $0xdeaa, %ax
movw $0xf0f0, %bx
andw %bx, %ax            # ax=0xd0a0
```

2.9.2. Instrucción OR

Operación `or` lógica bit a bit.

Ejemplo

```
movw $0xdeaa, %ax
movw $0xf0f0, %bx
orw %bx, %ax           # ax=0xfefa
```

2.9.3. Instrucción XOR

Operación `xor` lógica bit a bit.

Ejemplo

```
xorl %eax, %eax          # eax=0
movl $0xffffffff, %ebx   # ebx=0xffffffff
xorl %eax, %ebx          # ebx=0xffffffff
```

2.9.4. Instrucción NOT

Negación lógica bit a bit.

Ejemplo

```
movb $0xff, %al
notb %al            # al=0
```

2.10. Instrucciones rotación y desplazamiento

Las instrucciones de rotación y desplazamiento realizan una rotación cíclica o un desplazamiento no cíclico, por un número dado de bits, sobre un operando dado:

```
operaciónS <primer operando>, <segundo operando>
```

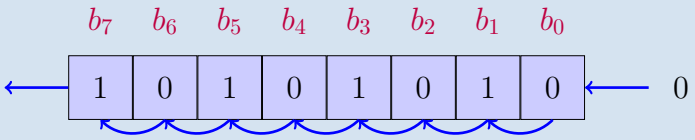
donde el primer operando es la cantidad de veces que se rota o desplaza el segundo operando.

2.10.1. Instrucción SAL/SHL

Las instrucciones `sal` (*Shift Arithmetic Left*) y `shl` (*Shift Logical Left*) son idénticas en cuanto a funcionamiento: ambas realizan un desplazamiento lógico hacia la izquierda del contenido de un operando, rellenando con ceros por la derecha y sin rotar.

Ejemplo

```
movb $0xaa, %al
salb $1, %al        # al=0x54
```

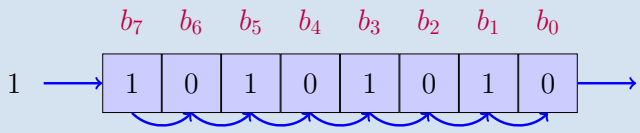


2.10.2. Instrucción SAR

La instrucción `sar` (*Shift Arithmetic Right*) realiza un desplazamiento aritmético hacia la derecha, conservando el signo del operando. Es decir, si el bit más significativo es uno, se desplaza ingresando unos; en caso contrario, se ingresan ceros.

Ejemplo

```
movb $-0xaa, %al    # al=0xaa (-86)
sarb $1, %al        # al=0xd5 (-43)
```

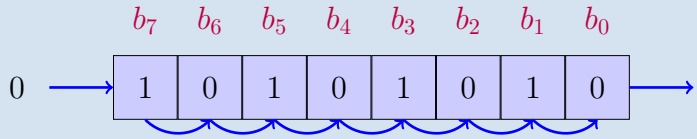


2.10.3. Instrucción SHR

La instrucción `shr` (*Shift Logical Right*) realiza un desplazamiento lógico hacia la derecha del contenido de un operando. A diferencia de la instrucción `sar`, no conserva el bit de signo: siempre inserta ceros por la izquierda.

Ejemplo

```
movb $0xaa, %al       # al=0xfc (-86)
shrb $1, %al          # al=0x55 (85)
```

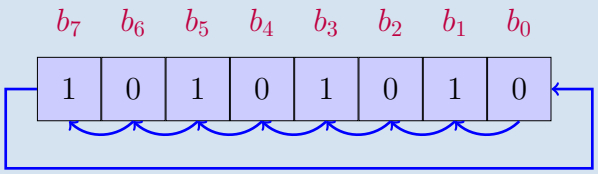


2.10.4. Instrucción ROL

La instrucción `rol` (*Rotate Left*) realiza una rotación lógica hacia la izquierda del contenido de un operando. Es decir, los bits del operando se desplazan hacia la izquierda, y los bits que "salen" por el extremo izquierdo vuelven a entrar por el extremo derecho.

Ejemplo

```
movb $0xaa, %al
rolb $1, %al          # al=0x55
```



2.10.5. Instrucción ROR

La instrucción `ror` (*Rotate Right*) es similar a la instrucción anterior pero con rotación hacia la derecha.

Ejemplo

```
movb $0xaa, %al
rorb $1, %al          # al=0x55
```

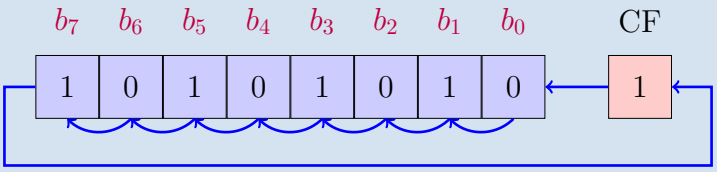
2.10.6. Instrucción RCL

La instrucción `rcl` rota los bits del operando hacia la izquierda, incorporando el bit de acarreo (CF) como un bit adicional en el extremo derecho. De este modo, el número total de bits rotados es igual al tamaño del operando más uno. El bit que sale por el

extremo izquierdo se almacena en CF, mientras que el valor previo de CF ingresa por el extremo derecho.

Ejemplo

```
movb $0xaa, %al
stc                 # CF=1 (se enciende la bandera de acarreo)
rclb $2, %al        # al=2*al+1=0x55
```



2.10.7. Instrucción RCR

La instrucción `rcr` realiza una rotación lógica a la derecha considerando el bit de acarreo. Similar a la instrucción anterior pero con rotación hacia la izquierda.

Ejemplo

```
movb $0xaa, %al
stc                 # CF=1
rcrb $1, %al        # al=0xd5
```

Los operadores lógicos y de desplazamiento se abordan con detalle en el Apunte *Manejo de Bits en Lenguaje C*.

2.11. Instrucciones de comparación

Una instrucción de comparación es la forma más común de evaluar dos valores para luego hacer un salto condicional. Una instrucción de comparación hace exactamente lo que dice su nombre, compara dos valores y establece las banderas del registro EFLAGS en consecuencia.

2.11.1. Instrucción CMP

La instrucción `cmp` realiza la comparación de los dos operandos:

```
cmpS <operando fuente>, <operando destino>
```

Hace la resta `destino=destino-fuente` sin guardar el resultado, solo modifica las banderas correspondientes. Aunque no se escriba el resultado, el destino tiene que ser un registro y no puede ser una constante.

Ejemplo

```
movq $45, %rbx
movq $66, %rax
cmpq %rax, %rbx   # CF=1, SF=1, ZF=0
cmpq %rbx, %rax   # CF=0, SF=0, ZF=0
```

En la primera instrucción `cmp` el operando destino es menor que el operando fuente. Por lo tanto `CF=1`, `ZF=0` y `SF=1`. En la segunda instrucción `cmp` el operando destino es mayor que el operando fuente. Por lo tanto `CF=0`, `ZF=0` y `SF=0`. En ninguna de las instrucciones el operador destino fue modificado.

2.11.2. Instrucción TEST

La instrucción `test` realiza la comparación lógica de los dos operandos:

```
testS <operando fuente>, <operando destino>
```

Realiza la operación lógica `and` bit a bit sin guardar el resultado. Solo modifica las banderas:

- ZF se activa si el resultado es 0.
- SF: refleja el bit más significativo del resultado (indica si sería negativo).
- PF: indica si el número de bits en 1 del byte menos significativo es par.
- CF y OF: se limpian (quedan en 0).

Ejemplo

```
movq $5, %rax
test %rax, %rax    # Realiza un AND bit a bit entre %rax y
    %rax
jz es_cero         # Salta a 'es_cero' si el resultado fue
   cero (es decir, si %rax == 0)
movq $1, %rbx      # Si %rax no era cero, carga 1 en %rbx
jmp fin
es_cero:
movq $0, %rbx      # Si %rax era cero, carga 0 en %rbx
fin:
```

2.12. Instrucciones para saltos incondicionales

2.12.1. Instrucción JMP

La instrucción `jmp` realiza un salto incondicional a la dirección de memoria indicada en su operando, el cual puede ser una etiqueta o un registro:

```
jmp etiqueta
jmp *%registro
```

Observación

Dado que el operando de las instrucciones de salto es siempre una dirección de memoria, estas instrucciones no llevan sufijo de tamaño.

Ejemplo

```
    .............
    jmp etiqueta
    movq %rax, %rbx
etiqueta:
    movq $45, %rcx
    .............
```

Luego de ejecutarse la instrucción `jmp` la siguiente instrucción ejecutada es `movq $45, %rcx` y la instrucción `movq %rax, %rbx` es salteada y nunca se ejecuta.

Ejemplo

```
    .............
    movq $cont, %rax
    jmp *%rax
    movq $1, %rax
cont:
    movq $2, %rax
    .............
```

En este ejemplo `jmp *%rax` realiza un salto a la dirección contenida en `rax`, que es la dirección de `cont`. Observar el uso del `*` antes del nombre del registro, lo cual es requerido por la sintaxis.

2.13. Instrucciones para saltos condicionales

Las instrucciones para saltos condicionales tienen la forma:

```
jCC etiqueta
```

donde `CC` es un sufijo que depende de la condición que se debe cumplir para realizar el salto. Es decir, salta a la etiqueta si se cumple la condición indicada con `CC`. De lo contrario, ejecuta la siguiente instrucción. Por lo tanto, antes de la instrucción `jCC` debe haber alguna instrucción que modifique las banderas correspondientes (por ejemplo, una instrucción de comparación o una instrucción aritmética). En la Tabla 1 mostramos un listado completo de instrucciones `jCC` y los valores requeridos en las banderas, donde `CC` es el sufijo que depende de la condición que se debe verificar.

Tabla 1: Instrucciones `jCC` y sus correspondientes `rFLAGS`.

Mnemónico	**Estado de banderas requerido**	**Descripción**
`JO`	OF = 1	*Jump near if overflow*
`JNO`	OF = 0	*Jump near if not overflow*
`JB`	CF = 1	*Jump near if below*
`JC`		*Jump near if carry*
`JNAE`		*Jump near if not above or equal*
`JNB`	CF = 0	*Jump near if not below*
`JNC`		*Jump near if not carry*
`JAE`		*Jump near if above or equal*
`JZ`	ZF = 1	*Jump near if zero*
`JE`		*Jump near if equal*
`JNZ`	ZF = 0	*Jump near if not zero*
`JNE`		*Jump near if not equal*
`JNA`	CF = 1 or ZF = 1	*Jump near if not above*
`JBE`		*Jump near if below or equal*
`JNBE`	CF = 0 and ZF = 0	*Jump near if not below or equal*
`JA`		*Jump near if above*
`JS`	SF = 1	*Jump near if sign*
`JNS`	SF = 0	*Jump near if not sign*
`JP`	PF = 1	*Jump near if parity*
`JPE`		*Jump near if parity even*
`JNP`	PF = 0	*Jump near if not parity*
`JPO`		*Jump near if parity odd*
`JL`	SF ≠ OF	*Jump near if less*
`JNGE`		*Jump near if not greater or equal*
`JGE`	SF = OF	*Jump near if greater or equal*
`JNL`		*Jump near if not less*
`JNG`	ZF = 1 or SF ≠ OF	*Jump near if not greater*
`JLE`		*Jump near if less or equal*
`JNLE`	ZF = 0 and SF = OF	*Jump near if not less or equal*
`JG`		*Jump near if greater*

Ejemplo

```
    movq $15, %rax
    cmp $10, %rax
    ja  mayor
    decq  %rax
    jmp fin
mayor:
    incq %rax
fin:
```

La instrucción `cmp $10, %rax` compara el valor 10 con el contenido de `rax` (es decir, evalúa `10 - rax` sin modificar los operandos, pero actualiza las banderas del procesador). Luego, `ja mayor` salta a la etiqueta mayor si el resultado de la comparación indica que `rax` es mayor que 10, es decir, si el valor de `rax` es estrictamente mayor que 10 sin signo (`ja` = *jump if above*).

En la Sección 3.2 se aborda en detalle la aplicación de las instrucciones para saltos condicionales.

2.14. Otras instrucciones de ruptura de secuencia

2.14.1. Instrucción LOOP

La instrucción

```
loop etiqueta
```

tiene dos efectos:

- Decrementa en uno el registro `rcx`. Aquí vemos que `rcx` tiene un uso especial.
- Luego, salta a la dirección de memoria indicada en la etiqueta **sólo si** el resultado de decrementar `rcx` dio distinto de cero. Si el resultado dio cero, el flujo del programa sigue en la siguiente instrucción a la instrucción `loop`.

Ejemplo

```
    movq $10, %rcx
    xorq %rax, %rax
etiqueta:
    incq %rax
    loop etiqueta
    .....
```

La instrucción `incq %rax` se ejecuta 10 veces. Por lo tanto, luego de ejecutarse el código anterior `rax=10`.

Las aplicaciones de la instrucción `loop` las veremos en la Sección 3.4.

2.14.2. Instrucción `CALL`

La instrucción `call` se utiliza para hace una llamada a subrutina:

```
call etiqueta
```

Su funcionamiento se basa en dos acciones principales:

- Guarda la dirección de retorno (la instrucción siguiente a `call`) en la pila.
- Transfiere el control a la dirección de la función llamada (es decir, realiza un salto la dirección de memoria indicada con la etiqueta).

También se puede hacer una llamada indirecta:

```
call *operando
```

En lugar de saltar a una dirección codificada explícitamente, transfiere el control a la dirección contenida en un registro.

2.14.3. Instrucción `RET`

La instrucción `ret` se utiliza para hacer un retorno de subrutina. No tiene operandos. Esta instrucción y la anterior se ven en detalle en la Sección 7.

2.15. Instrucciones para el registro de banderas

Existen instrucciones especiales para trabajar con el registro `rflags`. Entre ellas distinguimos varias clases:

- **Apagar un bit:** `clc` (*clear carry flag*), `cld` (*clear direction flag*).
- **Prender un bit:** `stc` (*set carry flag*), `std` (*set direction flag*), `sti` (*set interruption flag*).
- **Sumar añadiendo el carry:** `adc` toma dos operandos, los suma junto con el bit de carry y lo guarda en el destino.
- **Acceder al registro:** `lahf` (Load AH from Flags) y `sahf` (Store AH into Flags) copian ciertos bits del registro `ah` hacia el `rflags` y viceversa, `popfq` guarda en la pila el registro `rflags` y `pushfq` trae de la pila el registro `rflags`.

El uso del registro `rflags` se verá más claro en breve cuando expliquemos cómo se usa el registro para hacer saltos condicionales en la Sección 3.

2.16. Instrucciones de entrada/salida

Las instrucciones de entrada/salida realizan lecturas y escrituras desde y hacia el espacio de direcciones de entrada/salida. Este espacio de direcciones se puede utilizar para acceder y administrar dispositivos externos. Estas instrucciones requieren privilegios especiales.

- **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.

2.17. Instrucciones de conversión

Las instrucciones de conversión de datos realizan diferentes transformaciones de datos. En particular, la arquitectura x86-64 ofrece numerosas instrucciones para convertir entre enteros de distintos tamaño.

2.17.1. Instrucciones `CXX/CXXE`

Existe un conjunto de instrucciones que doblan el tamaño del registro correspondiente, extendiendo con el signo el valor almacenado, que tienen la siguiente forma:

```
cXX
cXXe
```

donde `XX` son dos sufijos de tamaño de acuerdo al tamaño del origen y del destino. Estas instrucciones no tienen operandos explícitos y operan de manera implícita con el registro `rax` o sus subregistros.

Aquí vemos algunas de las instrucciones disponibles:

Instrucción	Descripción
`cbw`	Extiende (con signo) `al` a `ax`.
`cwde`	Extiende (con signo) `ax` a `eax`.
`cwd`	Extiende (con signo) `ax` a `dx:ax`.
`cdq`	Extiende (con signo) `eax` a `edx:eax`.
`cdqe`	Extiende (con signo) `eax` a `rax`.
`cqo`	Extiende (con signo) `rax` a `rdx:rax`.

Observación

Notar que las instrucciones anteriores trabajan con operados implícitos. Notar también que hay instrucciones muy parecidas que difieren en que terminan con el sufijo `e`. Es decir, son instrucciones con diferentes nombres que hacen la misma conversión pero en el caso de las instrucciones que termina con `e` el resultado queda todo en un subregistro y no repartido en dos subregistros. Para un listado completo de las instrucciones de conversión consultar [14].

Ejemplo

```
movw $-34, %ax
cwd
```

Luego de ejecutarse el código anterior, `ax=0xffde` y `dx=0xffff`. En cambio, si hacemos `cwde` el resultado de la conversión queda en el subregistro `eax=0xffffffde`.

2.17.2. Instrucciones `CXTX`

Existe otro grupo de conversión de datos con la forma

```
cXtX
```

que también se usan para hacer conversiones donde se dobla el tamaño del dato. De manera análoga al conjunto de instrucciones vistas en la sección anterior se usan dos sufijos de acuerdo al tamaño del origen y del destino. Sin embargo, en este grupo de instrucciones los sufijos están separados por una `t` correspondiente a la palabra en inglés "*to*".

Aquí vemos algunas de las instrucciones disponibles:

Instrucción	Descripción
cwtl	Extiende (con signo) `ax` en `eax`.
cltq	Extiende (con signo) `eax` en `rax`.
cqto	Extiende (con signo) `rax` en `rdx:rax`.

Observación

Los sufijos de tamaño de las instrucciones en esta sección y la anterior corresponden a lo visto en la tabla de de la Página 9. Sin embargo, notar que aquí el sufijo `d` hace referencia a "*doble word*", es decir 32 bits, y no a doble precisión. De hecho, todas estas instrucciones utilizan datos de tipo entero. Veremos en la Sección 8.1 las instrucciones de conversión para datos de tipo flotante.

2.17.3. Instrucciones `MOVSXX`

Las instrucciones `movsXX` copian un valor del origen al destino extendiendo de acuerdo al signo. Estas instrucciones se utilizan para extender datos con signo y tiene dos sufijos, el primero es el tamaño del dato origen y el segundo es el tamaño del dato destino. Estas instrucciones son similares a las `cXX/cXXe` pero tienen mucha más versatilidad dado que permiten mayor cantidad de conversiones (no solamente doblando el tamaño) y no trabajan con registros implícitos.

Ejemplos

```
movsbl %bl, %ebx   # convierte un byte a 4 bytes
movswl %cx, %ecx   # convierte un word a 4 bytes
movswq %ax, %rax   # convierte un word a 8 bytes
```

Por lo tanto si tenemos el siguiente código:

```
movb $-45, %al        # al = 0xd3
movsbq %al, %rax      # rax=0xffffffffffffffd3
```

Luego de ejecutarse, `rax=0xffffffffffffffd3`, lo cual corresponde al valor -45 interpretado como un entero con signo en complemento a dos.

2.17.4. Instrucciones MOVZXX

Las instrucciones `movzXX` copian un valor del origen al destino extendiendo con *cero*. Estas instrucciones se utilizan para extender datos sin signo y tiene dos sufijos, el primero es el tamaño del dato origen y el segundo es el tamaño del dato destino.

Ejemplos

```
movzbl %al, %eax   # convierte un byte a 4 bytes
movzwl %ax, %eax   # convierte un word a 4 bytes
movzwq %ax, %rax   # convierte un word a 8 bytes
```

Dado el comportamiento predeterminado al trabajar con registros de 32 bits, no hay necesidad de una instrucción `movzlq` explícita. En efecto, si queremos extender sin signo el registro `eax` a `rax`, basta con hacer `movl %eax, %eax`.

3. Comparaciones, Saltos y Estructuras de Control

Cualquier código estructurado requiere que la ejecución no siempre siga con la siguiente instrucción escrita, sino que ciertas veces el procesador debe continuar la ejecución en otra porción de código (por ejemplo, al llamar a una función o en distintas ramas de una estructura *if*). Para ello, todas las arquitecturas incluyen funciones de salto, tanto incondicionales como condicionales.

3.1. Saltos incondicionales

Los saltos incondicionales en ensamblador x86-64 permiten modificar el flujo de ejecución de un programa sin necesidad de evaluar ninguna condición previa. Utilizando la instrucción `jmp`, el control se transfiere directamente a otra parte del código, especificada mediante una dirección o una etiqueta. Este tipo de salto es esencial para implementar bucles, ramificaciones y llamadas indirectas, y es una herramienta clave en la programación de bajo nivel para estructurar el comportamiento del programa.

Como hemos visto, la instrucción `jmp` toma como único operando una dirección a la cual "saltar". El efecto que tiene este salto es que la próxima instrucción a ejecutar no será la siguiente al `jmp` sino la indicada en su operando. La dirección del salto en general se da usando etiquetas (ver Sección 2.3).

Ejemplo

```
    movq $0, %rax
    jmp cont
    movq $1, %rax
cont:
    movq $2, %rax
```

En el fragmento de código anterior la instrucción `movq $1, %rax` **nunca** es ejecutada ya que la instrucción `jmp` hace que el procesador salte a la instrucción en la dirección

`cont`. Notar aquí que aunque `cont` es una constante (la dirección de memoria donde está la instrucción `movq 2, %rax`) ésta no va prefijada por `$`.

La instrucción `jmp` permite hacer saltos y es el equivalente a un `goto` de un lenguaje de alto nivel. Pero ¿cómo podemos implementar estructuras de control como bucles y condicionales con ella? Respuesta: no se puede. Para ello debemos introducir los saltos condicionales.

3.2. Saltos condicionales

Los saltos condicionales en ensamblador x86-64 permiten desviar el flujo de ejecución según el resultado de una operación previa, típicamente una comparación o una instrucción aritmética. Estas instrucciones evalúan los indicadores (flags) del registro de estado y realizan el salto solo si se cumple cierta condición, como igualdad, desigualdad o signo. Los saltos condicionales son fundamentales para implementar decisiones, bucles y estructuras de control en programas escritos en bajo nivel.

Los saltos condicionales en Assembler x86-64 utilizan una variedad de instrucciones que dependen del estado de los flags (indicadores) establecidos por instrucciones anteriores, como `cmp` o `test`. Estas instrucciones de salto comienzan con la letra `j` (de *jump*) seguida de un sufijo o abreviatura que indica la condición. En la Tabla 1 se mostró un listado completo de instrucciones `jCC` y los valores requeridos en las banderas.

Observación

Tanto los saltos condicionales como los incondicionales no llevan sufijo ya que su operando es siempre una dirección de memoria (dentro del segmento de código).

Junto con los saltos condicionales la arquitectura x86-64 incluye instrucciones para comparar dos valores. Una de estas instrucciones es la instrucción `cmp`. Como ya se mencionó, esta instrucción realiza una diferencia (resta) entre sus dos operandos, descartando el resultado pero **prendiendo los bits del registro `rflags`** acorde al resultado obtenido.

Siguiendo la lógica de la instrucción `sub`,

```
cmpq %rax, %rbx
```

realiza la resta `rbx-rax`, prende el bit `SF` (que indica negatividad) si `rax` es mayor que `rbx` pero a diferencia de `sub`, **no modifica el valor del registro destino `rbx`**. Notar que si ambos valores son iguales la resta tendrá un resultado nulo, prendiendo el bit `ZF`.

Como la relación que guardan dos valores (cuál es menor y cuál es mayor) depende de si dichos números se asumen con signo o sin signo, existen dos versiones de saltos condicionales por comparación de desigualdad. Por ejemplo:

- `jl` y `jg` (por *lower* y *greater*, respectivamente) para datos con signo.
- `jb` y `ja` (por *below* y *above*, respectivamente) para datos sin signo.

Ejemplo

```
        movq $45, %rbx
        movq $-66, %rcx
        cmpq %rbx, %rcx     # SF=1    OF=0
        jl menor
        ....
        ....
menor:
        ....
```

Luego de ejecutarse `cmpq %rbx, %rcx` las banderas quedan seteadas de la siguiente manera: `SF=1` y `OF=0`. Por lo tanto, luego de ejecutarse `jl menor` salta directamente a la etiqueta `menor`.

Observación

Es necesario que la instrucción de comparación esté ubicada inmediatamente antes que la instrucción de salto condicional. Si se colocan otras instrucciones entre la comparación y el salto condicional, el registro `rFlag` puede ser alterado y por lo tanto es posible que el salto condicional no refleje la condición correcta.

3.3. Estructuras de Control

Tratemos ahora de traducir el siguiente fragmento de función C en ensamblador:

```
long a=0;
if (a==100) {
    a++;
}
// seguir
......
```

cuya estructura se ilustra en la Figura 5.

Teniendo en cuenta lo que vimos sobre saltos y comparaciones, una posible traducción sería:

```
.global main
main:
    movq $0, %rax
    cmpq $100, %rax
    jz igual_a_cien
    jmp seguir
igual_a_cien:
    incq %rax
    jmp seguir
```

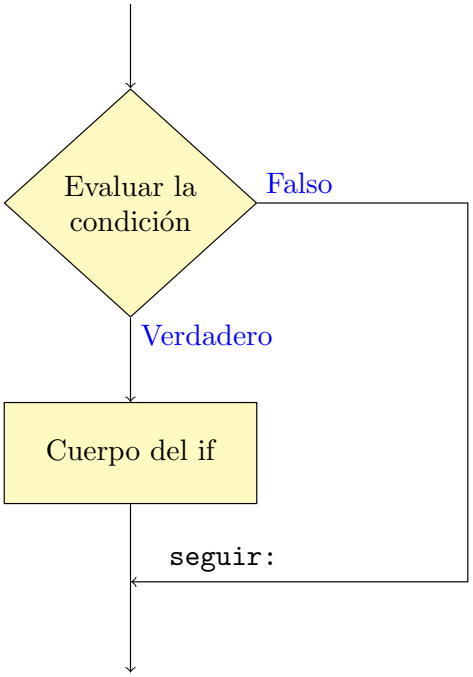


Figura 5: Estructura tipo `if`.

```
seguir:
    ....
```

En este código comparamos el valor de `rax` con la constante 100. Si el resultado dio cero (`rax-100`) es porque son iguales. En este caso debemos incrementar `rax`.

Veamos en el fragmento anterior varias cosas:

- El orden de los argumentos en la instrucción `cmp` es importante ya que la resta no es conmutativa. Notar también que esta instrucción necesita un sufijo de tamaño.
- Inmediatamente después de hacer la comparación realizamos el salto condicional. De tener más instrucciones en el medio, éstas podrían modificar el estado del registro `rflags`.
- Por la naturaleza del `if`, debemos definir dos etiquetas, una para saltar cuando la condición es verdadera (`igual_a_cien`) y otra para continuar la ejecución tanto si la condición fue verdadera o no (`seguir`). Notar que si la condición resulta falsa el programa salteará el bloque `igual_a_cien`.

Vemos ahora cómo traduciríamos el siguiente fragmento:

```
long a;
if (a==100) {
  a++;
} else {
  a--;
}
// seguir
```

En este caso el `if` tiene un `else`. Una posible traducción sería:

```
    movq $0, %rax
    cmpq $100, %rax
    jz igual_a_cien
    decq %rax
    jmp seguir
igual_a_cien:
    incq %rax
    jmp seguir
seguir:
   ...
```

En este código comparamos el valor de `rax` con la constante 100. Si el resultado dio cero (`rax-100`) es porque son iguales. En este caso debemos incrementar `rax`.

Vemos en el fragmento anterior varias cosas:

- En este caso si el salto condicional no se realiza (porque la condición resultó falsa) se ejecutará el decremento.
- Como ambas ramas del if deben unificarse, luego de hacer el decremento saltamos a `seguir` "salteando" la rama verdadera del if.
- Notar que como la etiqueta `seguir` está a continuación del bloque `igual_a_cien` el salto puede ser obviado.

3.4. Iteraciones

Otra estructura común en los lenguajes de alto nivel son las iteraciones, bucles o lazos. Con lo visto hasta ahora podemos ya traducir la mayoría de las estructuras iterativas.

Ejemplo

Supongamos que queremos traducir la siguiente estructura tipo `while`:

```
long int i;
while (i!=0) {
   cuerpo_del_while();
   i--;
}
```

Como antes, asumiremos que en ensamblador `i` es una etiqueta que aloja lugar para un entero de ocho bytes. Esto puede traducirse como:

```
while_1:
    cmpq $0, i          # Evaluar la condición
    je fin_1            # Si resulta falsa, el lazo termina
cuerpo_del_while_1:     # Acá irá el cuerpo del while
    ...
```

```
    ...
    decq i
    jmp while_1
fin_1:
    ...
    ...
```

El código anterior corresponde a la estructura de control que puede verse en la Figura 6.

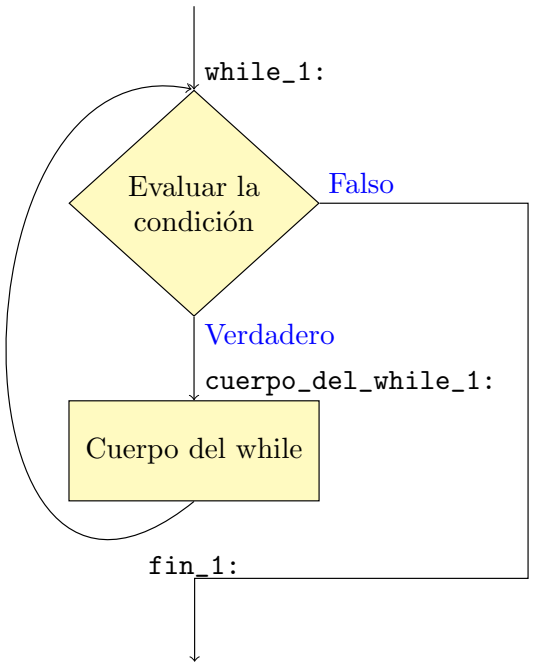


Figura 6: Estructura tipo `while`.

Las estructuras del tipo `for` son también muy comunes en lenguajes de alto nivel. Una forma particular de `for` es repetir un bloque de código una cantidad de veces dadas.

Dada la siguiente estructura tipo `for`:

```
int i;
for (i=100;i>0;i--) {
    cuerpo_del_for();
}
```

Se puede traducir de la siguiente manera utilizando la instrucción loop:

```
    movq $100, %rcx   # rcx se utiliza como iterador, inicializado en 100
cuerpo_del_for_1:
    ...
    ...
    loop cuerpo_del_for_1
```

Las instrucciones comprendidas entre la etiqueta `cuerpo_del_for_1` y la instrucción `loop` se ejecutan 100 veces.

4. Manejo de Arreglos y Cadenas

Un arreglo es una estructura de datos que almacena una colección de elementos del mismo tipo (por lo tanto del mismo tamaño) y le asigna un índice entero a cada uno. Existen distintas variantes de arreglos (largo fijo/variable, uni/multi-dimensional) pero en este apunte nos centraremos en arreglos a la "C", esto es, un arreglo `a` será la dirección del primer elemento (el de índice 0). Como cada elemento del arreglo tiene tamaño fijo al que llamaremos `s`, podemos calcular la dirección del elemento `i` del arreglo `a` como `a+i*s`.

Como los arreglos son estructuras de datos muy utilizadas, la arquitectura x86-64 incluye varias instrucciones (llamadas de cadena) para realizar copias, comparaciones, búsquedas, etc. Esta familia de instrucciones hace uso especial de dos registros: `rsi` (*source index*) y `rdi` (*destination index*) [3]. Cuando el procesador ejecuta una instrucción de cadena, éste incrementa/decrementa automáticamente esos registros [4] para apuntar al próximo elemento del arreglo. La cantidad incrementada/decrementada depende del tamaño del dato en cuestión. Además, el bit DF (*direction flag*) del registro `rflags` le indica al procesador si debe incrementar o decrementar los registros de índice (se puede apagar con `cld` para que se incrementen o prender con `std` para que se decrementen). A continuación veremos las diferentes instrucciones de manejo de arreglos y cadenas con sus respectivos ejemplos.

4.1. Copia y manipulación de datos

El procesador ofrece tres instrucciones para la copia y manipulación de datos almacenados en arreglos.

4.1.1. Instrucción LODS

La instrucción `lods` (de *load string*) copia en el registro `rax` (o en su sub-registro correspondiente) el valor apuntado por `rsi` e incrementa o decrementa `rsi` (dependiendo del valor de la bandera `DF`) en la cantidad de bytes indicada por el sufijo de tipo.

Así la instrucción `lodsw` (asumiendo `DF=0`) es equivalente a:

```
movw (%rsi),%ax
addq $2,%rsi
```

Notar que aquí se utiliza el subregistro `ax` para compatibilizar con el sufijo `w` de word y que por lo tanto el incremento es dos bytes.

4.1.2. Instrucción STOS

La instrucción `stos` (de *store string*) almacena el valor del registro `rax` (o su sub-registro correspondiente) en la dirección apuntada por `rdi` y luego incrementa/decre-

[3] Aunque su nombre sugieren que son índices, estos registros se utilizan como apuntadores en estas instrucciones.

[4] Algunas instrucciones solo incrementan/decrementan uno de estos registros.

menta el valor de `rdi` en la cantidad de bytes indicada por el sufijo de tipo. Así, la instrucción `stosl` (asumiendo `DF=1`) equivale a:

```
movl %eax, (%rdi)
subq $4, %rdi
```

4.1.3. Instucción MOVS

La instrucción `movs` (de *move string*) realiza las acciones de `lods` y `stos` aunque sin utilizar el registro `rax`, esto es, copia el valor apuntado por `rsi` en la posición de memoria apuntada por `rdi` e incrementa/decrementa **ambos** en la cantidad de bytes indicada por el sufijo de tipo. Así la instrucción `movsb` (asumiendo `DF=0`) es equivalente a

```
movb (%rsi),%regtemp
movb %regtemp, (%rdi)
addq $1, %rsi
addq $1, %rdi
```

siendo `regtemp` un registro temporario del procesador (en realidad no existe ese registro).

Observación

Notar que las instrucciones para manejo de arreglos y cadenas trabajan con operandos implícitos, es decir, los operandos no se declaran explícitamente sino que ya viene prefijado con que operandos se trabaja.

Un caso típico de uso de estas instrucciones de cadena es para traducir el siguiente fragmento C:

```
int f(char *a, char *b) {
    int i;
    for (i=0;i<100;i++)
    a[i]=b[i];
}
```

que puede ser implementado en ensamblador como

```
.global f
f:
    # por convención de llamada tenemos en rdi el puntero al arreglo "a"
       y en rsi el puntero al arreglo "b"
    movq $100, %rcx      # debemos iterar 100 veces
    cld                  # iremos incrementando rsi y rdi (DF=0)
sigue:
    movsb
    loop sigue
    ret
```

Al repetir 100 veces la instrucción `movsb` copiamos los 100 bytes de `b` hacia `a`. El mismo efecto se podría haber obtenido copiando 50 veces un word (con `movsw`), 25 veces un long (con `movsl`) o 12 veces un quad (con `movsq`) y un long **extra**.

Supongamos que ahora debemos modificar el arreglo como sigue:

```
int f(int *a){
int i;
for (i=0;i<100;i++)
    a[i]++;
}
```

Esto puede ser escrito utilizando instrucciones de cadena como sigue:

```
.global f
f:
    # suponemos que rdi tiene el puntero al arreglo "a"
    movq %rdi, %rsi      # el origen y el destino son el mismo arreglo
    movq $100, %rcx      # iteramos 100 veces
    cld                  # iremos incrementando rsi y rdi (DF=0)
l:
    lodsl          # cargamos en eax el elemento del arreglo (apuntado por
        rsi)
    incl %eax      # lo incrementamos
    stosl          # lo guardamos en el arreglo (apuntado por rdi)
    loop l         # pasamos al siguiente elemento
    ret
```

Vemos que en este caso el uso del registro `eax` es útil para obtener el valor original del elemento (con `lodsl`), modificar el registro (con `incl`) y luego guardarlo de nuevo (con `stosl`). Notar también que en este caso el arreglo destino y origen son el mismo, por ello copiamos `rdi` en `rsi` al iniciar la función.

4.2. Búsquedas y Comparaciones

Una operación común es buscar un elemento dentro de un arreglo o comparar dos arreglos. La arquitectura ofrece para esto dos instrucciones.

4.2.1. Instrucción SCAS

La instrucción `scas` (de *scan string*) compara lo apuntado por `rdi` con el valor del registro `rax` (o del sub-registro según corresponda) e incrementa/decrementa `rdi` en la cantidad de bytes dada por el sufijo de tipo.

4.2.2. Instrucción CMPS

La instrucción `cmps` (de *compare string*) compara el valor apuntado por `rsi` con el valor apuntado por `rdi` e incrementa/decrementa ambos registros en la cantidad de bytes dada por el sufijo de tipo.

Al igual que la instrucción `cmp` estas comparaciones prenden los bits correspondiente en el registro `rflags`.

Ejemplo

Veamos un caso de uso de las instrucciones de búsquedas y comparaciones de cadenas. Supongamos que queremos implementar en ensamblador la siguiente función C que busca un elemento en un arreglo.

```
int find(int *a, int k){
int i;
for (i=0;i<100;i++)
    if (a[i]==k) return 1;
return 0;
}
```

Esta función puede ser implementada en ensamblador como sigue:

```
.global find
find:
    cld                 # iremos incrementando rdi (DF=0)
    movq $100, %rcx     # iteramos 100 veces
    movl %esi, %eax     # buscamos el 2do argumento
sigue:
    scasl               # comparamos el elemento actual con eax
    je found            # si lo encontramos terminamos
    loop sigue          # si no seguimos
    movq $0, %rax       # no lo encontramos, retornar 0
    jmp fin
found:
    movq $1, %rax       # lo encontramos, retornar 1
fin:
    ret
```

4.3. Iteraciones con instrucciones de cadena

Como vimos en los ejemplos anteriores, es lógico que una instrucción de cadena se repita muchas veces. Por ejemplo, una por cada elemento del arreglo o cadena. Para facilitar la escritura de estas estructuras iterativas la arquitectura ofrece la familia de **prefijos** `rep` que pueden ser antepuestos a cualquier instrucción de cadena. Al igual que la instrucción `loop` el prefijo repite la instrucción la cantidad de veces indicada por `rcx`. Así, el ejemplo de copia de un arreglo a otro de la Sección 4.1 puede ser reescrito en ensamblador como:

```
.global find
.global f
f:
    # por convención de llamada tenemos en rdi el puntero al arreglo a y
       en rsi el puntero al arreglo b
    movq $100, %rcx          # debemos iterar 100 veces
    cld                      # iremos incrementando rsi y rdi (DF=0)
    rep movsb                # repite movsb 100 veces
    ret
```

Al igual que existen los saltos condicionales, existen los prefijos de repetición condicionales. Así, los prefijos `repe` y `repne` repiten la instrucción mientras el bit `Z` esté prendido/apagado a lo sumo `rcx` veces. El ejemplo de la búsqueda de un entero de la Sección 4.2 puede ser reescrito utilizando prefijos de repetición condicional como:

```
.global find
find:
    cld                 # iremos incrementando rdi (DF=0)
    movq $100, %rcx     # iteramos 100 veces
    movl %esi, %eax     # buscamos el 2do argumento
    repne scasl         # repetimos mientra sea distinto o a lo sumo 100
       veces
    je found            # si lo encontramos terminamos
    movq $0, %rax       # no lo encontramos, retornar 0
    jmp fin
found:
    movq $1, %rax       # lo encontramos, retornar 1
fin:
    ret
```

Observación

Notemos que el prefijo `repne` repite la instrucción mientras la comparación resulte distinta y a lo sumo `rcx` veces, pero ¿cómo saber por cuál de las dos causas finalizó la repetición?

Cuando la condición del prefijo resulta falsa los registros `rsi,rdi` son incrementados o decrementados según corresponda y el registro `rcx` es decrementado pero los bits del registro `rflags` quedan intactos dejando allí el valor de la última comparación. Por lo tanto, podemos realizar un salto condicional para ver si la última comparación dio igual o distinto.

5. Acceso a datos en memoria

Para acceder a datos de memoria en lenguaje 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. Veamos primero algunos conceptos importantes.

5.1. Modelo de memoria de un proceso en Linux

El modelo de memoria virtual[5] de un proceso[6] en Linux se divide en 4 regiones (segmentos):

- **Segmento de texto**: En este segmento van todas las instrucciones. El segmento de texto se denomina `.text` en GNU assembler (GAS). El segmento de texto no crece de manera dinámica, por lo que el segmento de datos se puede colocar inmediatamente después.
- **Segmento de datos**: Este segmento contiene todos los datos estáticos inicializados al iniciar el programa y no crece de forma dinámica. Además, se divide en dos partes:
  - `.data` que contiene datos inicializados explícitamente.
  - `.bss` se utiliza para reservar espacio de memoria para variables no inicializadas. El término "bss" proviene de "*Block Started by Symbol*".
- **Segmento heap**: En este segmento están los datos asignados por `malloc` o `new`. A diferencia de los segmentos anteriores, el heap crece de manera dinámica.
- **Segmento pila**: Este segmento es la pila en tiempo de ejecución[7]. En este segmento se encuentran los siguientes elementos:
  - direcciones de retorno
  - algunos parámetros de la función
  - variables locales de funciones
  - espacio para variables temporales

  El segmento de pila también crece de manera dinámica.

La Figura 7 muestra el modelo de memoria de un proceso. Aunque este modelo es en realidad más complejo, el diagrama esquemático proporciona una buena aproximación. Por ejemplo, `main` no está en realidad en la dirección 0. En realidad el segmento de texto comienza en una dirección un poco superior a `0x400000`. Con respecto al tope superior, también es una aproximación. La pila se asigna a las direcciones más altas de un proceso y en Linux x86-64 es `0x7fffffffffff` o 131 TB[8]. Esta dirección es equivalente a 47 bits con todos los bits en 1.

Tanto el *heap* como la pila necesitan crecer mientras el proceso está en ejecución: el *heap* crece "hacia arriba" (direcciones de memoria mayores) mientras que la pila crece "hacia abajo" (direcciones de memoria menores). Ambos segmentos pueden llegar a encontrarse en el medio y por lo tanto pueden llegar a "explotar". El uso del espacio de *heap*

[5] El tema *Memoria Virtual* lo veremos en detalle en el Apunte **Organización y Gestión de la Memoria**.

[6] En sistemas operativos, un proceso es una instancia en ejecución de un programa. Incluye el código del programa, su estado actual, datos, variables, y recursos asignados, como archivos abiertos y memoria. Es la unidad básica de trabajo que el sistema operativo administra para ejecutar tareas. Este se tema se verá en detalle en la asignatura **Sistemas Operativos II**.

[7] El tema *Gestión de la Pila* será visto con mayor detalle en la Sección 6.1.

[8] Esto es realidad no es exactamente así. La región superior del espacio de direcciones está reservada para el núcleo (Memoria virtual del núcleo) pero para los fines prácticos podemos asumir que la región superior es la pila.

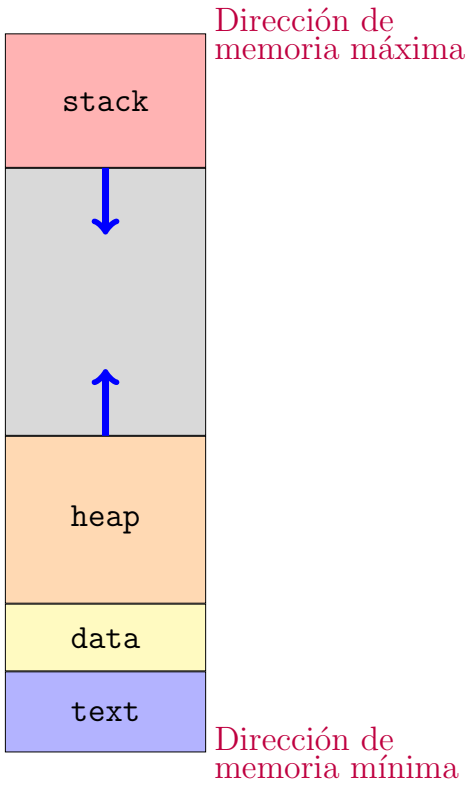


Figura 7: Modelo de memoria de un proceso en Linux [16].

y de pila en el lenguaje ensamblador no implica el uso de un segmento con sus nombres específicos.

El segmento de pila está limitado por el kernel de Linux. El tamaño típico es de 16 MB para Linux de 64 bits. Esto se puede inspeccionar usando "`ulimit -a`". 16 MB parece bastante pequeño, pero está bien a menos que se usen matrices grandes como variables locales en las funciones. El rango de direcciones de la pila es `0x7fffff000000` a `0x7fffffffffff`. El kernel reconoce si ocurre una falla en las direcciones fuera de este rango (*segmentation fault*).

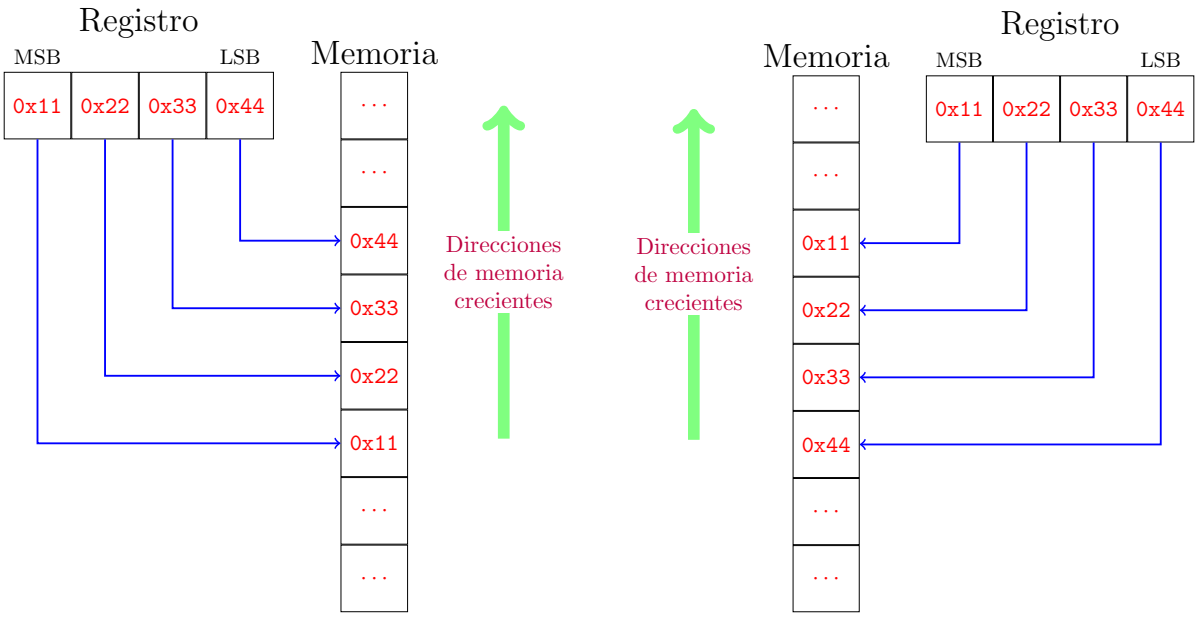
5.2. Endianness

El término inglés *endianness* designa el formato en el que se almacenan en memoria los datos de más de un byte. El problema es similar a los idiomas en los que se escribe de derecha a izquierda, como el árabe, o el hebreo, frente a los que se escriben de izquierda a derecha, pero trasladado de la escritura al almacenamiento en memoria.

Supongamos que tenemos que almacenar el entero `0x11223344` en la dirección de memoria `a`. Este valor se representa mediante los cuatro bytes 0x11 0x22 0x33 0x44, escribiendo más a la izquierda el byte más representativo (MSB) y más a la derecha el byte menos representativo (LSB).

Una opción es guardar el byte **más** significativo (`0x11`) en la dirección `a`, el segundo (`0x22`) en la dirección `a+1`, y así sucesivamente. Esto se conoce como convención *Big-Endian*.

Otra opción es almacenar en la dirección `a` el byte **menos** significativo (`0x44`), el siguiente (`0x33`) en la dirección `a+1` y así sucesivamente. Esta convención se denomina



(a) Big-Endian (b) Little-Endian

Figura 8: Convenciones de *Endianness*.

Little-Endian y es la utilizada por las arquitecturas x86, y por lo tanto, también por x86-64. La Figura 8 muestra ambas convenciones.

5.3. Definición de variables

La declaración de variables en un programa en ensamblador se realiza en la sección `.data`. Las variables de esta sección se definen utilizando las directivas vistas en la Sección 2.2. Por ejemplo, `var: .long 0x12345678` es una variable con el nombre `var` de tamaño 4 bytes inicializada con el valor `0x12345678` que comienza en la dirección de memoria cuya etiqueta es `var`. Es importante destacar que en ensamblador hay que estar muy alerta cuando accedemos a las variables que hemos definido previamente. Las variables se guardan en memoria consecutivamente a medida que las declaramos y no existe nada que delimite las unas de las otras. Veamos a continuación un ejemplo ilustrativo.

Ejemplo

```
.data
var1: .byte 0
var2: .byte 0x61
var3: .word 0x0200
var4: .long 0x0001E26C
```

Las variables se encontrarán en memoria tal como muestra la siguiente tabla (suponiendo que la variable `var1` está en la dirección `0x600880`):

Etiqueta	Dirección de memoria (en bytes)	Valor
var1	0x600880	0x00
var2	0x600881	0x61
var3	0x600882	0x00
	0x600883	0x02
var4	0x600884	0x6C
	0x600885	0xE2
	0x600886	0x01
	0x600887	0x00

La instrucción `movq var1, %rax` copia 8 a partir de la dirección `var1`. Es decir, el procesador tomará como primer byte el valor de `var1` y los 7 bytes que están a continuación. Por lo tanto, como los datos se tratan en formato *little-endian*, en el registro `rax` quedará cargado el valor `0x0001E26C02006100`. 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.

Conclusión: El acceso a los datos es muy flexible, pero, por otra parte, si no controlamos muy bien el acceso a las variables esta flexibilidad puede causar serios problemas.

Ejemplo

Veamos ahora un ejemplo completo que engloba todos los conceptos vistos en las secciones anteriores:

```
.data
i: .long 0
f: .double 3.14
str: .asciz "Hola mundo"

.bss
a: .quad

.text
.global main
main:
    movq $40, %rax      # rax=40
    movl i, %ebx        # ebx=0
    movq $-1, a         # a=0xffffffffffffffff (-1)
    movq f, %rdx        # rdx=0x40091eb851eb851f (3.14)
    movl str, %ecx      # ecx=0x616c6f48 ("aloH")
    ret
```

Aquí vemos que la etiqueta `i` (dentro del segmento de datos `.data`) define la posición de memoria donde el ensamblador alojará un entero inicializado en 0 (4 bytes). Luego en `f` un valor de punto flotante inicializado en 3.14 (8 bytes). Luego en `str` arranca una cadena de caracteres de 11 bytes (el byte final corresponde al cero final). En el segmento `.bss` se crea una variable tipo `quad` sin inicializar. Finalmente, vemos que dentro del

segmento de código se define una etiqueta global llamada `main`. Este será el punto de inicio del programa. Luego, a medida que se vayan ejecutando las instrucciones siguientes los registros y locaciones de memoria irán quedando con los valores indicados en los comentarios.

Notar que luego de ejecutarse la instrucción `movl str, %ecx` el valor del subregistro `ecx` es `"aloH"`, que corresponde a los primeros 4 bytes de la cadena `str` pero con los caracteres en orden invertido debido al formato *little-endian*. Sin embargo, hay que notar que las cadenas de caracteres se almacenan en la memoria "concatenando" los caracteres consecutivamente desde el primer carácter hasta el último comenzando en las posiciones más bajas de memoria. Por lo tanto, el carácter `'H'` estará almacenado en la posición `str`, el carácter `'o'` en la posición `str+1`, y así sucesivamente hasta llegar al último carácter que es el `null`.

Como hemos visto, podemos acceder a un dato en memoria utilizando la etiqueta que define la dirección de memoria donde dicho dato comienza. Ahora supongamos que queremos incrementar el valor de una variable definida por la etiqueta `i`, esto podemos hacerlo simplemente escribiendo:

```
incq i
```

Si antes era `i=23`, ahora es `i=24`. Es importante notar que aunque la etiqueta `i` es una constante, es decir, la dirección de memoria donde se aloja ese valor, la etiqueta `i` no lleva el signo `$`.

Si ahora quisiéramos sumar `i` con el registro `rax` podemos escribir:

```
addq i, %rax
```

Sin embargo, notar que esta última instrucción produce un efecto muy diferente. En este caso sumará una constante (la dirección de `i`) y no el valor alojado en `i`.

Muchas veces es útil conocer la dirección de memoria donde está alojado un valor. Esto en C se conoce como obtener un puntero al dato. Así, si tenemos una variable `long int i;` podemos obtener un puntero a dicha variable utilizando el operador de referencia, escribiendo `&i`. Como antes mencionamos, en ensamblador una etiqueta es una dirección de memoria constante. Por ello si quisiéramos obtener el valor de esa dirección podríamos escribir:

```
movq $i, %rax
```

Luego `rax` guardará la dirección de memoria del entero antes definido.

Ejemplo

Este ejemplo es interesante para ver la diferencia entre usar una etiqueta y el valor allí guardado.

```
.data
str: .asciz "hola mundo"

.text
.global main
main:
```

```
movq str , %rax   # Instruccion 1
movq $str, %rax   # Instruccion 2
ret
```

¿Qué diferencia hay entre la instrucción 1 y la 2? Aunque casi similares, las dos instrucciones son muy distintas entre sí. Ambas son un movimiento con destino a `rax`, pero veamos qué mueven.

Al ejecutar la primera, `rax` toma el valor de 7959387902288097128. ¿Qué ha ocurrido aquí? La instrucción le indica al procesador que debe copiar 8 bytes (ya que es un quad) desde la región de memoria indicada por la etiqueta `str` a `rax`. Como en esa región de memoria se aloja la cadena de caracteres `"hola mundo"` los primeros 8 bytes son `hola mun` y de allí el valor tan extraño. El valor 7959387902288097128 se puede descomponer en hexadecimal en los siguientes bytes `0x6e 0x75 0x6d 0x20 0x61 0x6c 0x6f 0x68`, donde cada uno corresponde en decimal a `110 117 109 32 97 108 111 104` y al convertirlo en caracteres ASCII son "num aloh" (notar que la frase aparece al revés por ser x86-64 little endian).

Al ejecutar la segunda lo que ocurrirá es que en `rax` se guardará la **dirección de memoria** donde está guardada la cadena de caracteres. Este valor dependerá del proceso de compilación y carga. Notemos que en este caso ningún carácter de esa cadena será copiado a `rax`. De hecho esa instrucción no accede a la memoria.

5.4. Modos de direccionamiento

A continuación, veremos los diferentes modos de direccionamiento que podemos utilizar en un programa ensamblador para acceder a datos en memoria.

5.4.1. Modo de direccionamiento inmediato

En el modo de direccionamiento inmediato uno de los operandos hace referencia a un dato en memoria cuya dirección se encuentra en la propia instrucción. El valor inmediato especificado debe poder ser expresado con 32 bits como máximo. Este valor puede ser una constante o también puede ser el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos.

Ejemplos

- Carga en el registro `rax` 8 bytes a partir de la dirección `0x404028`:

  ```
  movq 0x404028, %rax
  ```

- Carga en el registro `rax` 8 bytes a partir de la dirección cuya etiqueta es `a`:

  ```
  movq a, %rax
  ```

- Carga en el registro `rax` 8 bytes a partir de la dirección calculada como `a+8`:

  ```
  movq a+8, %rax
  ```

- Carga en el registro `rax` 8 bytes a partir de la dirección calculada como `a+8*2`:

```
movq a+8*2, %rax
```

5.4.2. Modo de direccionamiento indirecto con registro

En este caso, uno de los operandos hace referencia a un dato en memoria utilizando un registro que contendrá la dirección de memoria del dato al cual queremos acceder. Este registro, que actúa como un puntero al dato en memoria, se pone entre paréntesis.

Ejemplos

- El primer operando utiliza la dirección que tenemos en `rax` para acceder a memoria. Se mueven 8 bytes a partir de la dirección especificada por `rax` y se guardan en `rbx`:

```
movq (%rax), %rbx
```

- Guarda en la dirección de memoria especificada por `rbx` el valor almacenado en `rax`:

```
movq %rax, (%rbx)
```

5.4.3. Modo de direccionamiento indexado

En este caso, el operandos que hace referencia a un dato en memoria especifica una dirección de memoria como una dirección base (cargada en un registro que está entre paréntesis) sumada a un desplazamiento o índice (que puede ser expresado mediante un número o el nombre de una variable que tengamos definida). También se lo puede interpretar al revés. Es decir, la constante se puede usar como base y el registro como índice.

Ejemplos

- Carga en el registro `rbx` 8 bytes a partir de la dirección de memoria `rax+8`:

```
movq 8(%rax), %rbx
```

- Carga el contenido del registro `rbx` (8 bytes) a partir de la dirección de memoria `rax-16`:

```
movq %rbx, -16(%rax)
```

- Carga en el registro `rbx` 8 bytes a partir de la dirección de memoria `rax+var`:

```
movq var(%rax), %rbx
```

5.4.4. Modo de direccionamiento relativo

En este caso, el operando que hace referencia a memoria tiene la siguiente forma general:

$$desplazamiento(\%base, \%índice, escala) \tag{1}$$

donde la base y el índice pueden ser cualquier registro de propósito general, la escala puede ser 1, 2, 4 u 8 y el desplazamiento ha de ser un número representable con 32 bits. De esta manera la dirección especificada resulta:

$$[base + índice \times escala + desplazamiento]$$

Ejemplos

```
movq 3(%rbx, %rcx, 4), %rax     # Carga en el registro rax 8 bytes a
   partir de la dirección rbx+rcx*4+3.

movq (%rax, %rax, 2), %rax      # Carga en el registro rax 8 bytes a
   partir de la dirección rax+rax*2.

movq 4(%rbp, %rdx, 4), %rax     # Carga en el registro rax 8 bytes a
   partir de la dirección rbp+rdx*4-4.

movq 8(,%rax,4), %rax    # Carga en el registro rax 8 bytes a partir de
   la dirección rax*4+8. En este caso vemos que el registro base es
   opcional.
```

5.5. Desreferenciar memoria

Para desreferenciar un valor en memoria se utilizan los modos de direccionamiento vistos en la sección anterior. Por ejemplo, `(%rax)` se refiere a lo **apuntado** por `rax`.

Ejemplo

```
movq (%rax), %rbx
```

Copia en el registro `rbx` lo apuntado por el registro `rax` y no el contenido del mismo. Es decir, copia los 8 bytes (debido al sufijo `q`) a partir de la dirección de memoria guardada en el registro `rax` en el registro `rbx`.

Esta notación también permite formas más complejas utilizando los modos de direccionamiento vistos en las secciones anteriores:

- `K(%reg)` refiere al valor apuntado por `reg` más un corrimiento de `K` bytes, donde `K` es entero. El valor de `K` puede ser negativo, por lo cual se puede conseguir un corrimiento ascendente o descendente. Notar que aquí la constante `K` **no lleva** el símbolo `$`.
- `K(%reg1, %reg2, S)` refiere al valor `reg1+(reg2*S+K)`, donde `K` y `S` son constantes enteras y además $S \in \{1, 2, 4, 8\}$.

Ejemplos

```
movb 8(%rbp), %al              # al <--- *(rbp+8)
movw -16(%rbp), %ax            # ax <--- *(rbp-16)
movl %eax, 0x20(%rsp)          # *(rsp+32) <--- eax
movq (%rax,%rax,2), %rbx       # rbx <--- *(rax+rax*2)
movq -4(%rbp, %rdx, 4), %rbx   # rbx <--- *(rbp+rdx*4-4)
movq 8(,%rax,4), %rbx          # rbx <--- *(rax*4+8)
```

Vemos que algunos de los registros en la Ecuación 1 pueden ser opcionales.

Este tipo de direccionamiento sirve para acceder a arreglos y estructuras.

Ejemplo

Si tenemos un arreglo de enteros de 32 bits (4 bytes) apuntado por `rax` y queremos acceder el sexto elemento podemos hacer:

```
movq $6, %rcx
movl (%rax,%rcx, 4), %edx   # edx <--- *(rax+4*6)
```

5.6. Instrucción `LEA`

La arquitectura x86-64 ofrece una instrucción similar al operador de referencia de C. Esta instrucción se denomina `lea` (por "*load effective address*") y calcula la dirección efectiva del operando de origen y la almacena en el operando de destino.

```
leaS <operando fuente>, <operando destino>
```

donde `S` es el sufijo según lo visto en la Tabla 2.1.

El operando de origen es una dirección de memoria especificada con uno de los modos de direccionamiento de los procesadores mientras que el operando de destino es un registro de propósito general. Usando sintaxis AT&T, los modos de direccionamiento útiles con `lea` son los siguientes:

Modo	Dirección
K(%base)	K + base
(,%índice, M)	índice * M
(%base, %índice, M)	base + índice*M
K(, %índice, M)	K + índice*M
K(%base, %índice, M)	K + base + índice*M
etiqueta	Dirección de etiqueta

donde K es una constante entera que actúa como desplazamiento, M es una constante con un valor de 2, 4 u 8, que funciona como multiplicador, y %dest, %índice y %base son registros.

Ejemplo

```
.data
num: .long 10, 20, 30, 40, 50    # arreglo de enteros de 4 bytes

.text
.global main

main:
    movl $2, %eax
    leaq num, %rbx               # Carga la dirección de 'num' en RBX
    leaq (%rbx,%rax,4), %rcx     # Calcula la dirección del tercer
       elemento
    movl (%rcx), %eax            # Retorna el valor del tercer elemento
    ret
```

Observación

Las siguientes instrucciones son equivalentes:

```
leaq str, %rax        # En rax queda la dirección de la etiqueta str
movq $str, %rax       # Esta instrucción es equivalente a la anterior
```

En estas instrucciones NO hay acceso a memoria. Notar que a pesar de que el primer operando en las dos primeras instrucciones parece ser una referencia de memoria, en lugar de leer desde la ubicación designada, en realidad la instrucción solo copia la dirección efectiva al destino y NO accede a memoria. Esta instrucción es equivalente al operador & utilizado en el lenguaje C.

Por otra parte, en la siguiente instrucción SÍ hay acceso a memoria con la dirección contenida en rax:

```
movq (%rax), %rbx       # Se dereferencia la dirección str
```

La instrucción lea a menudo se usa como un "truco" para hacer ciertos cálculos, aunque ese no sea su propósito principal.

Ejemplos

La instrucción `lea` se puede usar para multiplicar un registro por 2, 3, 4, 5, 8, o 9:

```
lea constante(, %src, 2), %dst          # dst = src*2 + constante
lea constante(%src, %src, 2), %dst      # dst = src*3 + constante
lea constante(, %src, 4), %dst          # dst = src*4 + constante
lea constante(%src, %src, 4), %dst      # dst = src*5 + constante
lea constante(, %src, 8), %dst          # dst = src*8 + constante
lea constante(%src, %src, 8), %dst      # dst = src*9 + constante
```

donde `%src` y `%dst` pueden ser el mismo registro. Además, se le puede sumar una constante, todo en un solo paso.

6. Gestión de la pila

En la arquitectura x86-64, la gestión de la pila (stack) desempeña un papel fundamental en el control del flujo de ejecución de los programas y en la organización del espacio de memoria durante las llamadas a funciones. La pila es una estructura de datos de tipo LIFO (*Last In, First Out*) que se utiliza principalmente para almacenar direcciones de retorno, parámetros de funciones, variables locales y ciertos registros temporales. Su correcto manejo es esencial para garantizar la integridad del programa y la seguridad del sistema.

A diferencia de arquitecturas más simples, la arquitectura x86-64 amplía el conjunto de registros y establece convenciones específicas para la gestión de la pila, incluyendo el uso de los registros `rsp` (*stack pointer*) y `rbp` (*base pointer*), así como reglas estrictas para la alineación de la pila y el paso de argumentos. Comprender estas convenciones permite interpretar correctamente el comportamiento de programas compilados, depurar errores, y escribir código ensamblador eficiente y compatible con funciones escritas en lenguajes de alto nivel como C o C++.

6.1. Definición de pila

Una pila es una estructura de datos que permite almacenar información de manera ordenada. Su funcionamiento puede entenderse mediante la analogía con una pila de platos sobre una mesa: uno puede ir apilando platos, y la pila crecerá hacia arriba. Para retirar un plato, se debe quitar el que está en la parte superior, lo que reduce el tamaño de la pila. Como se observa, al extraer un elemento de la pila, se remueve el último que fue insertado (si lo hubiera).

Por esta razón, la estructura de datos pila se conoce como **LIFO** (*Last-In, First-Out*), ya que el último en entrar es el primero en salir.

La arquitectura x86-64 permite al programador utilizar una porción de la memoria como pila, conocida como **segmento de pila**.

La pila puede emplearse para distintas finalidades, tales como:

- **Almacenamiento temporal:** por ejemplo, las variables automáticas en C se almacenan en la pila.

- **Implementación de llamadas a funciones:** el orden de llamada y finalización de las funciones sigue el comportamiento de una pila. Si la función `f` llama a `g` y `g` llama a `h`, la primera en finalizar será `h`, seguida de `g` y finalmente `f`. Esto es especialmente útil en funciones recursivas.
- **Preservación de valores de registros:** como se explicará en la Sección 7, algunos registros se modifican durante una llamada a función. El programador puede almacenar los valores de estos registros en la pila y restaurarlos después de la llamada.

Aunque la arquitectura permite utilizar la pila con diversos fines, es muy común que cada función reserve una porción de la pila para almacenar sus variables locales, argumentos, dirección de retorno, entre otros datos. A esta subporción de la pila se la denomina **marco de activación** de la función. En la Figura 9 se muestra un diagrama de pila con un posible estado que contiene varios marcos de activación. En un momento dado, sólo uno de ellos está activo: el correspondiente a la función que se está ejecutando.

Observación

En la Figura 9 se ve que el último elemento insertado en la pila está ubicado en direcciones **más bajas de memoria**, es decir, en la implementación de x86-64 la pila crece hacia direcciones más bajas. Esto es así por cuestiones históricas y para permitir que tanto el segmento de datos como el de pila crezcan de forma de optimizar el espacio libre (el de datos crece desde abajo hacia arriba y el de pila desde arriba hacia abajo).

6.2. Uso de registros en la gestión de la pila

La arquitectura posee dos registros especiales para manipular la pila:

rsp (*stack pointer*): Es un registro de 64 bits que apunta (guarda la dirección de memoria) al último elemento apilado dentro del segmento de pila (**tope**).

rbp (*base pointer*): Es un registro de 64 bits que apunta al **inicio** de la sub-pila o marco de activación.

Aunque ambos registros tienen funciones específicas en la gestión de la pila (como se verá a continuación), pueden manipularse con instrucciones comunes como `add`, `mov`, etc.

Como se mencionó en las Secciones 2.7.2 y 2.7.3, la arquitectura x86-64 también ofrece dos instrucciones específicas para la gestión de la pila: una para "apilar" elementos y otra complementaria para "desapilar".

6.3. Instrucción `PUSH`

La instrucción `push` primero decrementa el registro `rsp` en 8 (recordemos que la pila "crece" hacia direcciones más bajas) y luego almacena en esa dirección el valor que toma como argumento. Así, la instrucción `pushq $0x12345678` es equivalente a

```
subq $8, %rsp
movq $0x12345678, (%rsp)
```

El comportamiento de la instrucción `pushq` puede verse en la Figura 10.

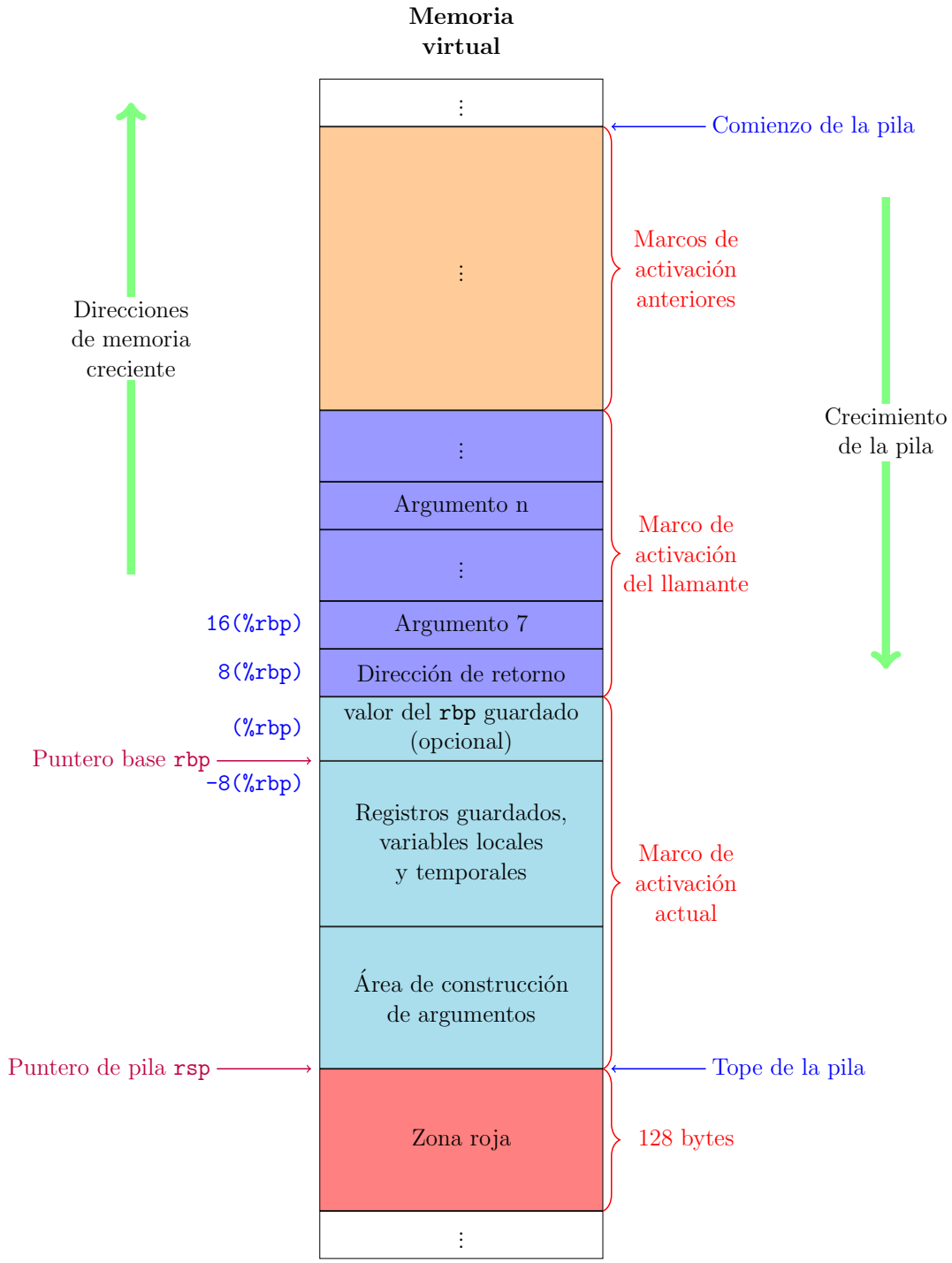


Figura 9: Diagrama de la estructura de pila en arquitectura x86-64.

6.4. Instrucción `POP`

La instrucción `pop` primero copia el valor apuntado por el registro `rsp` en el operando que toma como argumento, luego incrementa el registro `rsp` en 8 (la pila decrece hacia direcciones más altas). Así, la instrucción `popq %rax` es equivalente a

```
movq (%rsp), %rax
addq $8, %rsp
```

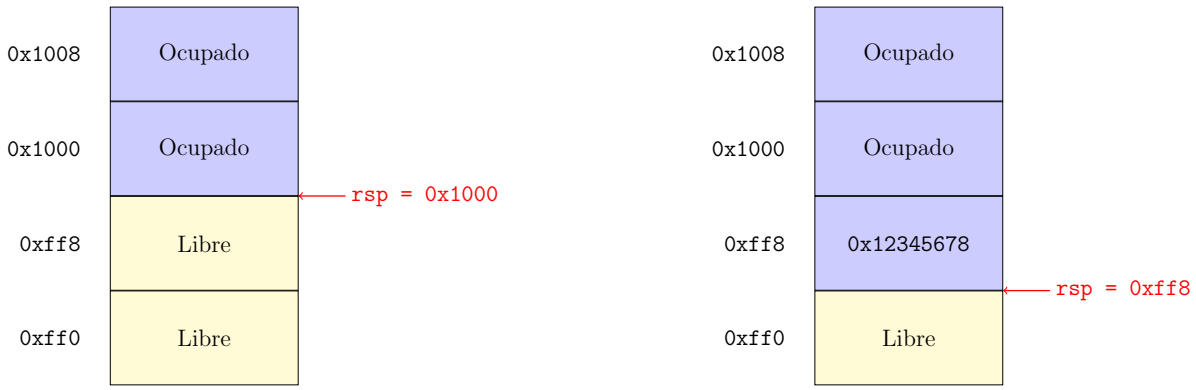


Figura 10: Diagrama de la pila antes y después de ejecutar la instrucción `pushq $0x12345678`.

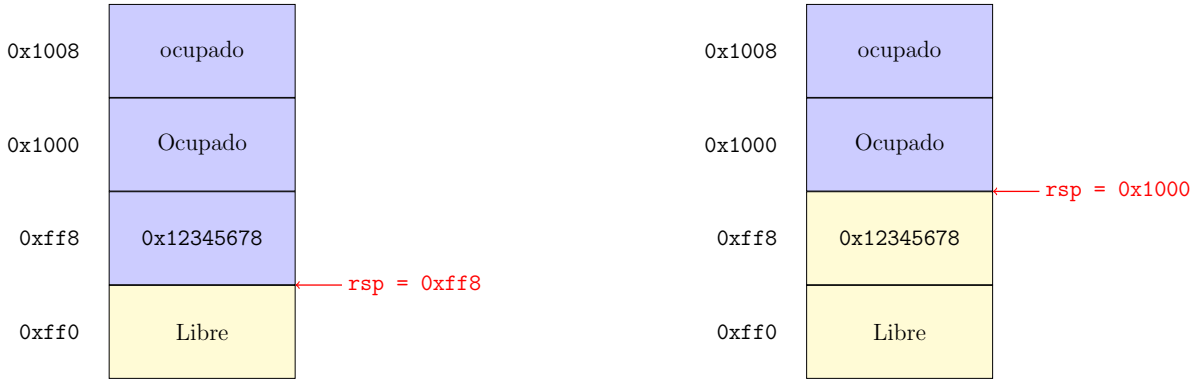


Figura 11: Diagrama de la pila antes y después de ejecutar la instrucción `popq %rax`.

El comportamiento de la instrucción `popq` puede verse en la Figura 11. Notar que el valor `0x12345678` continua almacenado en la dirección `0xff8`.

Observaciones

- Si bien en las instrucciones `push` y `pop` podemos utilizar tanto el sufijo `w` como el sufijo `q`, por cuestiones de alineación de la pila en general los datos insertados en la pila deben ser de 8 bytes utilizando el sufijo `q`.
- El marco de la pila para determinadas funciones debe estar alineado a 16 bytes. Sin embargo, este requisito tiene varias excepciones y en general basta con que esté alineado a 8 bytes.
- ¿Qué significa que un dato o un marco de pila está alineado a una cantidad determinada de bytes? Significa que su dirección de memoria es divisible por la cantidad de bytes en cuestión. Por ejemplo: la dirección de memoria `0x404030` está alineada a 16 bytes mientras que la dirección `0x7fffffffebb8` está alienada a 8 bytes.

7. Llamada a funciones

Una parte fundamental del código estructurado son los procedimientos y funciones. Estas construcciones permiten dividir un programa en partes más pequeñas, comprensibles y reutilizables. En particular, una función en un programa puede compararse con una función matemática: recibe ciertos argumentos y devuelve un valor de retorno.

Así vemos por ejemplo que la función en C:

```
long int sum(long int a, long int b);
```

tomará dos enteros largos como argumentos y devolverá otro entero largo.

Desde el punto de vista del procesador, una llamada a función es similar a un salto, ya que implica modificar el flujo del programa para ejecutar el código de la función llamada. La diferencia principal es que, dado que el código se ejecuta secuencialmente, tras la llamada a función el control debe regresar al punto siguiente a dicha llamada para continuar con la ejecución normal del programa. Veamos esto en un ejemplo en C:

```
...
i++;
printf("%d\n",i);
i--;
...
```

Aquí vemos tres instrucciones. La segunda es una llamada a la función `printf` con dos argumentos, la cadena de formato `"%d\n"` y el valor de `i`. Luego de finalizada la impresión por parte de `printf` el código debe seguir con el decremento de `i`. Pero, ¿cómo sabe `printf` a qué instrucción debe retornar, considerando que puede ser llamada desde múltiples lugares distintos? La respuesta es que no lo sabe: es el código que realiza la llamada quien debe indicarle adónde continuar la ejecución una vez finalizada. Esta dirección se conoce como **dirección de retorno**.

Para realizar llamadas a función, la arquitectura x86-64 provee dos instrucciones: la instrucción `call`, que transfiere el control a la función llamada, y `ret`, que devuelve el control a la función llamante.

7.1. Instrucción `call`

La instrucción `call` realiza la invocación a la función indicada como operando (la etiqueta que la define) guardando en la pila la dirección de retorno (la dirección de la próxima instrucción al `call`). Así la instrucción `call f` sería equivalente a

```
    pushq $direccion_de_retorno
    jmp f
direccion_de_retorno:
    .....
```

donde la constante `dirección_de_retorno` indica la dirección de la próxima instrucción a la llamada a función.

7.2. Instrucción `ret`

La instrucción `ret` retorna de una función sacando el valor de retorno que se encuentra en el tope de la pila (puesto allí por el `call`) y salta a ese lugar. Así la instrucción `ret` equivale a

```
popq %rdi
jmp *%rdi
```

Sin embargo, `ret` **no modifica** ningún registro (excepto `%rip`), y el uso de `%rdi` en este código es solo ilustrativo para mostrar el comportamiento de la instrucción con un código equivalente. En este ejemplo, el asterisco es necesario por la sintaxis.

7.3. Convención de llamada en x86-64

Al invocar una función en un programa, es necesario acordar cómo se pasan los argumentos, cómo se devuelve el resultado y quién se encarga de preservar el contenido de los registros. Estas reglas forman lo que se conoce como **convención de llamada**.

En la arquitectura x86-64, se sigue una convención específica que define, entre otras cosas, qué registros se utilizan para pasar los primeros argumentos, cómo se debe alinear la pila, y qué registros deben ser preservados por la función llamada. Esta convención garantiza que funciones escritas en distintos lenguajes o por distintos compiladores puedan interoperar correctamente.

En una llamada a función, se distinguen dos partes: el **caller** (llamador) y el **callee** (llamada).

- El **caller** es el código que invoca la función.
- El **callee** es el código de la función que se ejecuta como resultado de esa llamada.

Ambos tienen responsabilidades específicas dentro de la convención de llamada: por ejemplo, quién debe preservar ciertos registros, cómo se pasan los argumentos y cómo se retorna el control al final de la ejecución.

Convención de llamada para x86-64 en Linux

- Los seis primeros argumentos a la función son pasados por registro en el siguiente orden: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` (cuando los argumentos son valores enteros o direcciones de memoria).
- Si los argumentos son valores de punto flotantes pueden utilizarse hasta 8 de los registros `xmm` en el siguiente orden: `%xmm0`, `%xmm1`, `%xmm2`, `%xmm3`, `%xmm4`, `%xmm5`, `%xmm6` y `%xmm7`.
- Parámetros grandes mayores a 64 bits, por ejemplo estructuras pasadas por valor, se pasan utilizando la pila.
- Cuando la función toma como argumento una mezcla de valores enteros y flotantes `rdi` será el primer valor entero, `xmm0` el primer valor flotante, y así sucesivamente. Así, en la función `void f(int, double, int, double)` los argumentos irán en `rdi,xmm0,rsi,xmm1`.
- Si hubiera más argumentos de los que se pueden pasar por registros, éstos son pasados a la función utilizando la pila.
- Los valores escalares (como enteros y punteros) se devuelven en el registro `rax`.

- Los valores en punto flotante (`float` y `double`) se devuelven en `xmm0`.
- El llamado **se compromete** a preservar el valor de los registros `%rbx`, `%rbp`, `%rsp`, y `%r12` a `%r15`. Esto no quiere decir que no los pueda usar sino que al retornar deben tener el mismo valor que al comenzar la función. La función podría guardarlos temporalmente en memoria o pila y restaurarlos antes de retornar. Estos registros se conocen como *callee saved* ya que es responsabilidad del llamado preservarlos.
- Los otros registros (incluso los utilizados para pasar los argumentos) pueden ser modificados libremente por la función sin necesidad de restaurar sus valores. Si el llamante desea preservar sus valores es responsabilidad de él, por lo cual estos registros se conocen como *caller saved*. En la Tabla 2 se puede observar el rol de los registros en la llamada a función.
- El bit `DF` de `rflags` está inicialmente apagado (esto incrementará los punteros en instrucciones de manejo de cadena) y debe ser apagado al finalizar la función (y antes de llamar a otra función).
- El puntero de la pila (`rsp`) debe estar alineado a 16 bytes antes de realizar una llamada a función. Esto asegura que cualquier instrucción de operaciones con registros SIMD (usados para procesar datos en paralelo) funcione correctamente.
- Como `%rbp` y `%rsp` son preservados durante una llamada a función, el estado de la pila del llamante se mantiene.
- Si la función llamada es variádica, entonces el número de argumentos de punto flotante que se pasan a la función en los registros vectoriales debe ser proporcionado por la función llamante en el subregistro `al`.

En la Tabla. 2 vemos todos los registros de propósito general del x86-64 y su uso durante una llamada a función. Asimismo, vemos su rol en la convención de llamada (*caller saved* o *callee saved*) y si son preservado.

7.4. Prólogo y epílogo de una función

Respecto a la preservación de la pila, es muy común que cada función utilice el registro `rbp` como puntero al inicio de su sección en la pila. Dado que este registro es *callee-saved*, debe ser preservado por la función llamada.

Por esta razón, es habitual encontrar secciones denominadas **prólogo** y **epílogo** en una función, como se muestra a continuación:

```
#prólogo
pushq %rbp          # Preserva en la pila el valor del rbp del llamante
movq %rsp, %rbp     # La pila para esta función comienza en el tope

CÓDIGO DE LA FUNCIÓN

#epílogo
movq %rbp, %rsp     # rsp vuelve a apuntar al tope de la pila anterior
```

Tabla 2: Registros de la arquitectura x86-64 y su rol en la convención de llamadas.

Registro	Uso	Convención	¿Preservado?
`rax`	Valor de retorno	*Caller saved*	No
`rbx`		*Callee saved*	Sí
`rcx`	4^{o} argumento	*Caller saved*	No
`rdx`	3^{o} argumento	*Caller saved*	No
`rsi`	2^{o} argumento	*Caller saved*	No
`rdi`	1^{o} argumento	*Caller saved*	No
`rbp`	Puntero base	*Callee saved*	Sí
`rsp`	Puntero de pila	*Callee saved*	Sí
`r8`	5^{o} argumento	*Caller saved*	No
`r9`	6^{o} argumento	*Caller saved*	No
`r10`	Temporal	*Caller saved*	No
`r11`	Temporal	*Caller saved*	No
`r12`		*Callee saved*	Sí
`r13`		*Callee saved*	Sí
`r14`		*Callee saved*	Sí
`r15`		*Callee saved*	Sí

```
popq %rbp              # Restaura el rbp del llamante
```

7.5. Llamada a función

Como se mencionó en las secciones anteriores, una llamada a función en la arquitectura x86-64, implica una serie de convenciones y mecanismos que permiten el paso de argumentos, la transferencia del control de ejecución y el posterior retorno al punto de invocación. Estas llamadas siguen el estándar de llamada conocido como System V AMD64 ABI en sistemas tipo Unix.

En este esquema, los primeros seis argumentos de una función se pasan mediante registros (en orden: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`), y los restantes se colocan en la pila (stack). Para transferir el control, se utiliza la instrucción `call`, que guarda la dirección de retorno en la pila y salta a la dirección de la función. Al finalizar, la función ejecuta `ret`, que recupera dicha dirección y continúa la ejecución del programa. Además, deben respetarse convenciones sobre qué registros deben preservarse entre llamadas (por ejemplo, `rbx`, `rbp`, `r12-r15`) y cuáles pueden ser modificados libremente (como `rax`, `rcx`, `rdx`, etc.).

La Figura 12 ilustra la función `suma`, invocada por `main`, que recibe tres números enteros como argumentos. Esta función los suma y devuelve el resultado a `main`. En la figura se muestra, a la izquierda, el código en lenguaje C, y a la derecha, un código equivalente en Assembler x86-64.

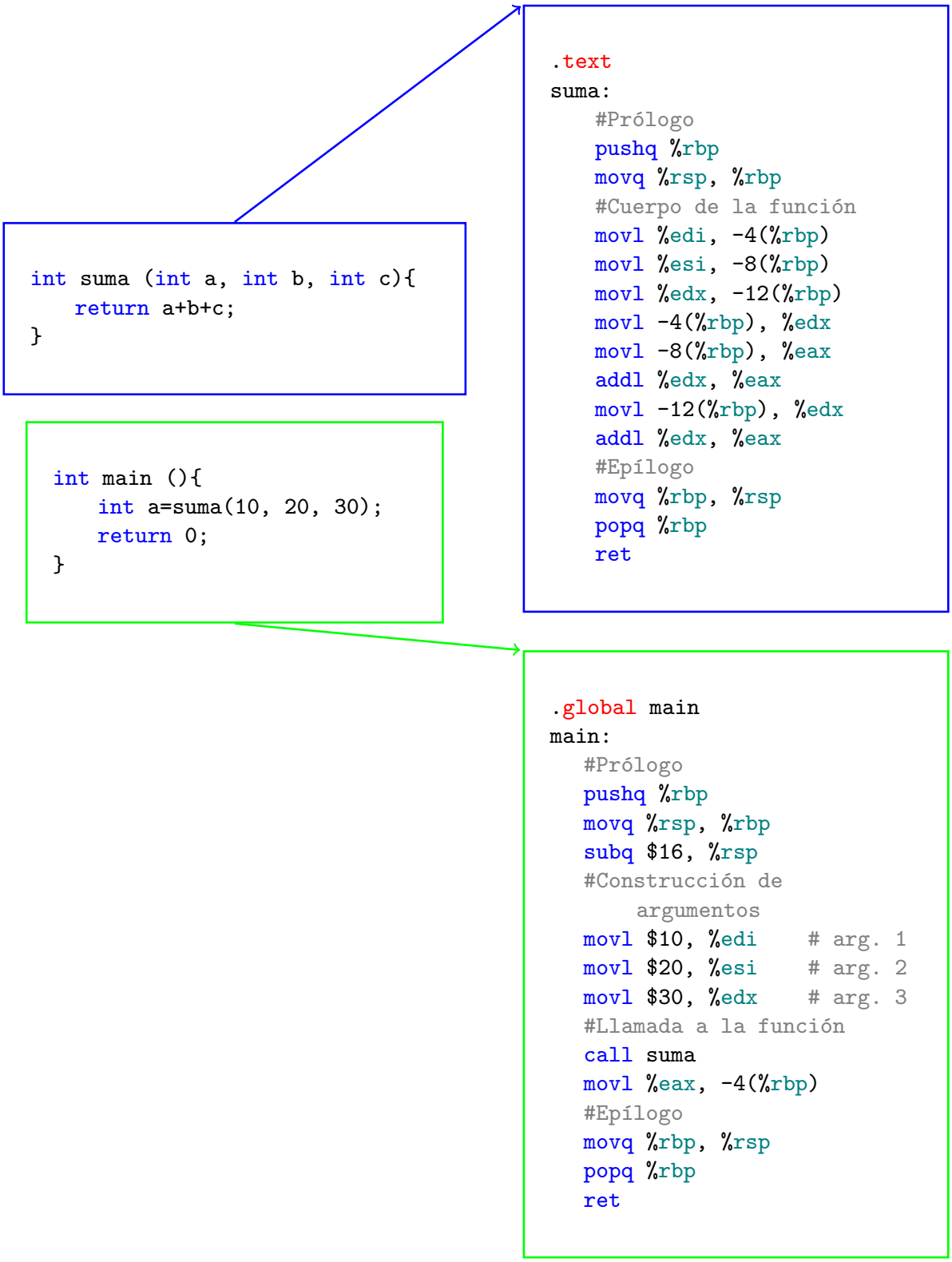


Figura 12: Ejemplo de llamada a función.

7.6. Zona roja en x86-64

En la Figura 9 se muestra un área de la memoria denominada ***"zona roja"***. La arquitectura x86-64 especifica que los programas pueden utilizar los 128 bytes situados

por debajo del puntero de la pila actual (es decir, en direcciones más bajas que el valor de `rsp`). Esta área se considera reservada y no es modificada por señales o manejadores de interrupciones.

Debido a esto, las funciones pueden aprovechar la zona roja para almacenar datos temporales que no sean necesarios en llamadas a otras funciones. En particular, las funciones de "hoja" (*leaf functions*) pueden usar esta área directamente en lugar de ajustar el puntero de la pila en el prólogo y el epílogo.

La zona roja es una optimización. El código puede asumir que los 128 bytes por debajo de `rsp` no serán alterados por señales o manejadores de interrupciones, lo que permite usarlos para datos temporales sin necesidad de mover explícitamente el puntero de la pila. Sin embargo, dado que esta área es utilizada por llamadas a función, su principal utilidad se encuentra en funciones que no invocan a otras funciones.

Ejemplo

En este ejemplo, presentaremos una función llamada `suma`, que recibe dos números enteros largos como argumentos. Estos números son sumados, y la función retorna el resultado de la suma. A continuación, veremos cómo llamar a la función `suma` con los argumentos 40 y 45:

```
.data
i: .space 8

.text
.global main
main:
    # Prólogo
    pushq %rbp
    movq %rsp, %rbp

    subq $16, %rsp          # Se reserva espacio en la pila

    # Cuerpo de la función main
    movq $40, -8(%rbp)      # Variable local
    movq $45, -16(%rbp)     # Variable local
    movq -8(%rbp), %rdi     # Primer arg. en rdi
    movq -16(%rbp), %rsi    # Segundo arg. en rsi
    call suma               # Guarda la dirección de retorno en pila y
       salta a "suma"
    movq %rax, i            # Aquí rax contiene el resultado (85)

    # Epílogo
    movq %rbp, %rsp
    popq %rbp

    ret

suma:
    # Prólogo
```

```
pushq %rbp                # Preserva el registro rbp
movq %rsp, %rbp           # rbp "apunta" al inicio de la pila

# Cuerpo de la función suma
movq %rdi, -8(%rbp)       # Variable local
movq %rsi, -16(%rbp)      # Variable local
movq -8(%rbp), %rax
addq -16(%rbp), %rax      # Aquí el resultado ya está en rax
movq %rax, -24(%rbp)      # Variable local

# Epílogo
movq %rbp, %rsp       # rsp "apunta" nuevamente al inicio de la pila
popq %rbp             # Se restaura el registro rbp

ret                   # Retorna de la función "suma" a "main"
```

En la Figura 13, se presenta un esquema de la pila que muestra cómo se almacenan las direcciones de retorno, cómo se preserva el registro `rbp` y cómo se almacenan las variables locales. También se puede observar la evolución del valor del registro puntero de pila `rsp`.

En este diagrama, vemos que las variables locales 40 y 45 se almacenan en el marco de activación de la función `main`, espacio que se reserva mediante la instrucción `subq $16, %rsp`. Sin embargo, en la función `suma`, las variables locales no requieren espacio reservado. ¿Por qué? Porque la "zona roja"garantiza que los 128 bytes por debajo del valor apuntado por el registro `rsp` no se modificarán por señales o interrupciones.

Es importante destacar que esto es posible porque `suma` es una función "hoja". En cambio, si dentro de `suma` se llamara a otra función, lo anterior ya no sería válido, y sería necesario reservar espacio adicional mediante un nuevo decremento del valor del registro `rsp`.

8. Aritmética de Punto Flotante

La arquitectura x86-64 soporta aritmética de datos de punto flotante utilizando el estándar IEEE 754 tanto para simple como doble precisión. Las operaciones de punto flotante se realizan a través de una extensión de la arquitectura que podemos considerar separada conceptualmente de la ALU (llamada SSE -Streamming SIMD Extension)

Por lo tanto se utilizan otros registros e instrucciones. Para esto hay 16 registros de 128 bits (16 bytes): `xmm0` a `xmm15`. Cada registro puede contener un elemento (i.e.: un flotante de simple o doble precisión) en cuyo caso el valor se considera "escalar" (scalar) y se usa sólo una parte del registro, o puede contener múltiples elementos del mismo tamaño (formato "empaquetado" -packed-). Por ejemplo, en `xmm0` entran 4 flotantes de simple precisión o también 16 enteros de 1 byte (chars). El formato empaquetado permite que algunas instrucciones realicen la misma operacion sobre varios datos a la vez (SIMD: Single Instruction Multiple Data).

Las instrucciones siguen algunas reglas:

- Las letras `s` (por "scalar") y `p` ("packed") indican qué formato se utiliza.

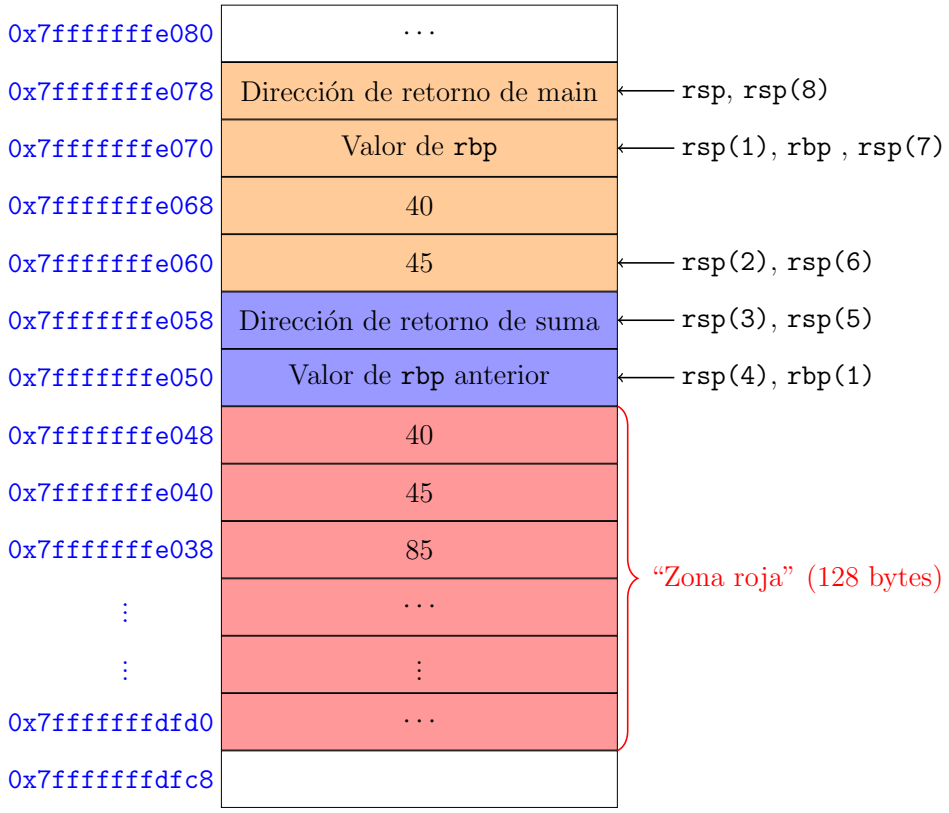


Figura 13: Ejemplo de esquema de pila.

- Las letras `s` (por "single"), `d` ("double") e `i` (ïnteger") indican el tipo de datos involucrado. Además `q` indica que un entero es tamaño quadword (i.e.: 8 bytes).

Por ejemplo, `cvtsi2sdq` permite convertir un entero almacenado en un *quadword* a un *double* en formato escalar. Se interpreta así:

- `cvt`: convert (convertir)
- `si`: scalar integer (un entero con signo)
- `2`: two ("two" suena como "to" - a -)
- `sd`: scalar double (un flotante escalar de doble precisión)
- `q`: quadword (el entero mencionado es un quadword)

Veremos primero las instrucciones de copias y conversiones, luego las operaciones aritméticas escalares y luego las operaciones sobre datos empaquetados (SIMD).

8.1. Copias y conversiones

Al igual que con los registros de propósito general, existen instrucciones para copia de datos. Para los registros de punto flotante existen las instrucciones `movss` y `movsd` que copian un dato de precisión simple (*float*) y doble precisión (*double*), respectivamente, de un registro `xmm` a otro o desde/hacia la memoria.

A su vez existen múltiples instrucciones para convertir entre enteros y datos de punto flotante. En la Tabla 3 se recopilan las instrucciones de conversión.

Tabla 3: Instrucciones de copia y conversiones para punto flotante [5].

Instrucción	**S**	**D**	**Descripción**
`movss S, D`	M32/X	X	Copiar precisión simple
`movss S, D`	X	M32	Copiar precisión simple
`movsd S, D`	M64/X	X	Copiar precisión doble
`movsd S, D`	X	M64	Copiar precisión doble
`cvtss2sd S, D`	M32/X	X	Convertir de simple a doble precisión
`cvtsd2ss S, D`	M64/X	X	Convertir de doble a simple precisión
`cvtsi2ss S, D`	M32/R32	X	Convertir entero a simple precisión
`cvtsi2sd S, D`	M32/R32	X	Convertir entero a doble precisión
`cvtsi2ssq S, D`	M64/R64	X	Convertir quadword entero a simple precisión
`cvtsi2sdq S, D`	M64/R64	X	Convertir quadword entero a doble precisión
`cvttss2si S, D`	X/M32	R32	Convertir (truncado) simple precisión a entero
`cvttsd2si S, D`	X/M64	R32	Convertir (truncado) doble precisión a entero
`cvttss2siq S, D`	X/M32	R64	Convertir (truncado) simple precisión a quadword entero
`cvttsd2siq S, D`	X/M64	R64	Convertir (truncado) doble precisión a quadword entero

X: Registro XMM (e.g., `%xmm3`)
R32: Registro de propósito general de 32 bits (e.g., `%eax`)
R64: Registro de propósito general de 64 bits (e.g., `%rax`)
M32: 32 bits de memoria
M64: 64 bits de memoria

Tabla 4: Instrucciones aritméticas en punto flotante[5].

Simple precisión	Doble precisión	Efecto	Descripción
`addss S, D`	`addsd S, D`	$D \leftarrow D + S$	Suma en punto flotante
`subss S, D`	`subsd S, D`	$D \leftarrow D - S$	Resta en punto flotante
`mulss S, D`	`mulsd D, D`	$D \leftarrow D \times S$	Multiplicación en punto flotante
`divss S, D`	`divsd S, D`	$D \leftarrow D \div S$	División en punto flotante
`maxss S, D`	`maxsd S, D`	$D \leftarrow \text{máx}(D,\ S)$	Máximo en punto flotante
`minss S, D`	`minsd S, D`	$D \leftarrow \text{mín}(D,\ S)$	Mínimo en punto flotante
`sqrtss S, D`	`sqrtsd S, D`	$D \leftarrow \sqrt{S}$	Raíz cuadrada en punto flotante

Ejemplo

Veamos el procedimiento para inicializar una variable de tipo double (en el registro `xmm0`) con el valor 1.0:

```
movq $1, %rax          # Copiar un 1 entero a rax
cvtsi2sdq %rax, %xmm0 # Convierte el 1 de rax al double 1.0 en xmm0
```

8.2. Operaciones en punto flotante

Las operaciones entre valores de punto flotante siempre involucran dos operandos, el operando fuente puede ser tanto un registro `xmm` como un valor almacenado en memoria. El destino debe ser un registro `xmm`. La Tabla 4 resume las operaciones más utilizadas para simple y doble precisión.

Ejemplo

Veamos, con lo que tenemos cómo traducir la siguiente función C:

```
double convert(double t) {
    return t*1.8 + 32;
}
```

Veremos en la Sección 7 que la convención de llamada indica que los argumentos de punto flotante se pasan por los registros `xmm` y el valor de retorno se deja en el registro `xmm0`. Sabiendo esto podemos escribir:

```
.global convert
convert:
    # en xmm0 viene t por convención de llamada

    movq $0x3ffcccccccccccd, %rax  # 1.8 en IEEE754 doble precisión. El
       valor inmediato es la representacion de 1.8 según IEEE 754
    movq %rax, -8(%rsp)
```

Tabla 5: Instrucciones de comparación en punto flotante[5].

Instrucción	Basada en	Descripción
`ucomiss S2, S1`	$S_1 - S_2$	Comparación de precisión simple
`ucomisd S2, S1`	$S_1 - S_2$	Comparación de precisión doble

```
movsd -8(%rsp), %xmm1     # Carga el valor 1.8 en xmm1
movq $32, %rax
cvtsi2sdq %rax, %xmm2
# Carga el valor 32.0 convirtiendo el valor entero 32 de rax a xmm2
mulsd %xmm1, %xmm0        # xmm0=xmm0*xmm1 => xmm0=t*1.8
addsd %xmm2, %xmm0        # xmmo=xmm0+xmm2 => xmmo=t*1.8+32
# como el valor de retorno se escribe en xmm0 hemos terminado
ret
```

8.3. Comparaciones en punto flotante

Al igual que con los valores enteros la arquitectura ofrece comparaciones de valores de punto flotante. Las instrucciones de comparación comparan dos valores (haciendo una resta virtual) y prenden las banderas correspondientes en el registro `rflags`. La comparación se comporta como una comparación de datos unsigned (i.e.: conviene utilizar `jae` para saltar por mayor o igual). Además, si los valores son incomparables (alguno es NaN) se prende la bandera `PF (Parity Flag)`.

Las instrucciones de comparación en punto flotante se muestran en la Tabla 5. Las instrucciones de comparación de punto flotante establecen tres banderas de condición: la bandera cero `ZF`, la bandera de acarreo `CF` y la bandera de paridad `PF`. Las banderas de condición se establecen de la siguiente manera:

Orden	CF	ZF	PF
"desordenado"	1	1	1
$S_1 < S_2$	1	0	0
$S_1 = S_2$	0	1	0
$S_1 > S_2$	0	0	0

El caso "desordenado" ocurre cuando cualquiera de los operandos es NaN. Esto se puede detectar con la bandera de paridad. Comúnmente, la instrucción `jp` (para "saltar en paridad") se usa para saltar condicionalmente cuando la comparación en punto flotante arroja un resultado desordenado. Por otra parte, `ZF` se establece cuando los dos operandos son iguales y `CF` cuando $S_1 < S_2$. Las instrucciones `ja` y `jb` se usan para saltar condicionalmente en estos casos.

8.4. Convención de llamada en punto flotante

Como hemos visto, en la arquitectura x86-64, la convención de llamada a funciones define cómo se pasan los parámetros y cómo se gestionan los registros durante las invocaciones. Aquí vemos algunas particularidades para el caso de funciones con argumentos de punto flotante:

- Para las funciones que utilizan números en punto flotante, los primeros ocho argumentos de tipo flotante se pasan a través de los registros `xmm0` a `xmm7`.
- Si hay más argumentos flotantes, se pasan en la pila.
- Una función que devuelve un valor en punto flotante lo hace en el registro `xmm0`.
- Todos los registros `xmm` son preservados por el llamador. Por lo tanto, la función llamada puede sobrescribir cualquiera de estos registros sin guardarlos previamente.

Ejemplo

En este ejemplo vemos cómo llamar a la función `printf` para imprimir un entero y un flotante doble precisión:

```
.data
str: .asciz "%d %f\n"
a: .long 45
f: .double 3.14

.text
.global main
main:
    pushq %rbp          # Alineamos la pila a 16 bytes
    leaq str, %rdi      # Le pasamos la direc. de la cadena de formato
    movl a, %esi        # Le pasamos el segundo argumento
    movsd f, %xmm0      # le pasamos el tercer argumento
    movb $1, %al        # Cantidad de argumentos de punto flotante
    call printf         # Llamamos a la función printf
    popq %rbp           # Desapilamos para preservar el valor de rsp
    xorl %eax, %eax     # Retornamos cero
    ret
```

Notar que para poder utilizar la función `printf` la convención de llamada AMD64 System V ABI[11] requiere varias cuestiones:

- Justo antes de una instrucción `call` la pila debe estar alineada al menos con 16 bytes.
- La convención de llamada también requiere que que el subregistro `al` contenga el número de registros vectoriales utilizados para una función de argumento variable. `printf` es una función de argumento variable, por lo que es necesario configurar `al`. En este ejemplo la cantidad de argumentos de punto flotante es uno.
- `rdi` debe ser un puntero a la cadena de formato.

- También debemos terminar la cadena de formato con NULL. Por lo tanto, en lugar de utilizar `.ascii`, utilizar `.asciz`.

9. Instrucciones SIMD

Los programas de procesamiento de señales multimedia (audio, imágenes, video, etc.) suelen requerir la repetición de una misma operación sobre una gran cantidad de datos, como aplicar un cálculo específico a cada píxel. Para este tipo de tareas, las arquitecturas modernas incluyen las instrucciones conocidas como Streaming SIMD Extensions (SSE), donde SIMD significa Single Instruction, Multiple Data. Estas instrucciones permiten aplicar una única operación de manera simultánea sobre múltiples datos. La mayoría de las instrucciones aritméticas SIMD ejecutan operaciones paralelas con vectores, también denominadas operaciones "empaquetadas" (*packed*, en inglés), ya que utilizan operandos que contienen varios elementos procesados en paralelo.

Recordemos que los registros `xmm0-xmm15` son de 128 bits (ver Figura 3) por lo cual pueden alojar 4 valores de precisión simple o 2 de precisión doble o también 16 bytes, 8 words, 4 enteros de 32 bits o 2 de 64 bits.

Hay varios tipos de instrucciones *packed*:

- Instrucciones de transferencia de datos.
- Instrucciones de conversión.
- Instrucciones aritméticas.
- Instrucciones lógicas.

La Tabla 6 muestra algunas instrucciones para operaciones con flotantes empaquetados. Sin embargo, las extensiones SSE incluyen muchas más instrucciones. En este apunte, solo se presenta una introducción. Un listado completo puede consultarse en [8] o [9].

Ejemplo

La siguiente figura y código ilustra cómo sumar dos vectores de cuatro elementos de tipo `float` en una sola operación:

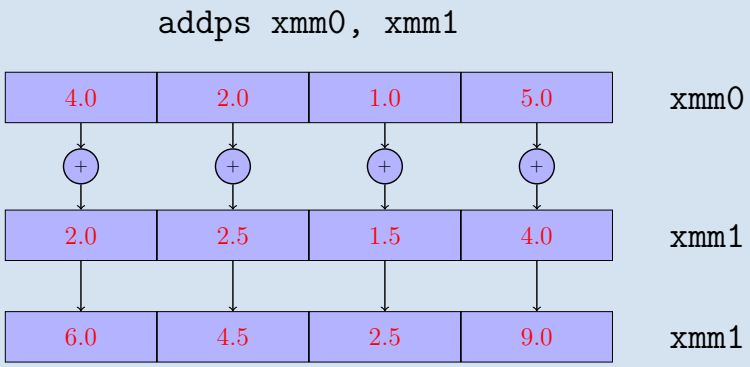


Tabla 6: Instrucciones SIMD para operaciones con flotantes empaquetados.

Instrucción	**Descripción**
`movaps`	Mueve cuatro flotantes simple precisión alineados entre registros XMM o memoria, asumiendo que la dirección de memoria está alineada a 16 bytes ("aps" significa *Aligned, Packed, Single-precision*).
`movapd`	Similar a la anterior pero mueve dos flotantes dobles precisión alineados entre registros XMM o memoria.
`movups`	Mueve cuatro flotantes simple precisión alineados entre registros XMM o memoria, sin requerir alineación de 16 bytes en la dirección de memoria ("ups" significa *Unaligned, Packed, Single-precision*).
`movupd`	Similar a la anterior pero mueve dos flotantes dobles precisión alineados entre registros XMM o memoria.
`addps`	Suma flotantes simple precisión empaquetados.
`divps`	Divide flotantes simple precisión empaquetados.
`divss`	Divide flotantes simple precisión escalares.
`mulps`	Multiplica flotantes simple precisión empaquetados.
`subps`	Resta flotantes simple precisión empaquetados.
`cmpps`	Compara flotantes simple precisión empaquetados.
`andnps`	Realiza la operación AND NOT bit a bit de flotantes simple precisión empaquetados.
`andps`	Realiza la operación AND bit a bit de flotantes simple precisión empaquetados.
`orps`	Realiza la operación OR bit a bit de flotantes simple precisión empaquetados.
`xorps`	Realiza la operación XOR bit a bit de flotantes simple precisión empaquetados.

```
.data
vec1: .float 4.0, 2.0, 1.0, 5.0
vec2: .float 2.0, 2.5, 1.5, 4.0
result: .space 16    # Reserva 16 bytes (4 bytes por cada float)

.text
.global main
main:
    movups vec1, %xmm0    # cargar vec1 en xmm0
    movups vec2, %xmm1    # cargar vec2 en xmm1
    addps %xmm0, %xmm1    # sumar: xmm1 = xmm0 + xmm1
    movups %xmm1, result # guardar el resultado
    xorl %eax, %eax
    ret
```

Por otra parte, existen instrucciones que forman parte del conjunto de extensiones SIMD y están diseñadas para realizar operaciones eficientes sobre enteros empaquetados. Estas instrucciones permiten efectuar comparaciones, movimientos y multiplicaciones en paralelo sobre múltiples datos en registros vectoriales, lo que optimiza el rendimiento en aplicaciones que requieren procesamiento intensivo de datos.

En particular, las instrucciones `pmov` facilitan movimientos con extensión de signo o ceros, mientras que la familia de instrucciones `padd` y `pmul` ofrece distintas variantes de suma y multiplicación, algunas con almacenamiento del resultado alto o bajo. Las instrucciones `pmax` y `pmin` permiten determinar el valor máximo o mínimo entre elementos de enteros empaquetados, con o sin signo. Estas operaciones son fundamentales en aplicaciones de procesamiento de imágenes, criptografía y cálculos científicos de alto rendimiento.

Ejemplo

Veamos un ejemplo de instrucciones *packed*:

```
.data
.align 16
a: .float 1.0, 2.0, 3.0, 4.0
b: .float 1.0, 2.0, 3.0, 4.0
c: .byte 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
d: .byte 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

.text
.global main
main:
    movq $a, %rdi               # rdi apunta a "a"
    movq $b, %rsi               # rsi apunta a "b"
    movaps (%rdi), %xmm0        # copia 4 floats en xmm0
    movaps (%rsi), %xmm1        # copia 4 floats en xmm1
    addps %xmm0, %xmm1          # suma los dos vectores
    movaps %xmm1, (%rdi)        # guarda el resultado en "a"

    movq $c, %rdi               # rdi apunta a "c"
    movq $d, %rsi               # rsi apunta a "d"
    movaps (%rdi), %xmm0        # copia 16 bytes en xmm0
    movaps (%rsi), %xmm1        # copia 16 bytes en xmm1
    paddb %xmm0, %xmm1          # suma los dos vectores
    movaps %xmm1, (%rsi)        # guarda el resultado en "d"

    xorl %eax, %eax
    ret
```

La instrucción relevante en este caso es `addps`, que suma simultáneamente cuatro valores en punto flotante de precisión simple. Es importante notar que, para utilizar `movaps`, los datos deben estar alineados a 16 bytes, lo cual se puede garantizar mediante la directiva `.align`. Por otro lado, la instrucción `paddb` permite sumar en paralelo 16 bytes empaquetados en registros `xmm`.

Por otra parte, en el 2011 se introdujo una nueva tecnología de instrucciones SIMD llamadas AVX de 256 bits, pero estas no serán vistas en este apunte.

Apéndice A: Compilando código ensamblador con GCC

Como vimos, es necesario traducir el código desde un lenguaje de alto nivel al lenguaje de máquina que el procesador puede ejecutar. Para ello, se utiliza un software específico denominado **compilador**.

Por ejemplo, el compilador GCC realiza esta traducción en varias etapas. Primero, convierte el código fuente a un código intermedio en lenguaje ensamblador (Assembler), una representación textual del lenguaje de máquina que describe instrucciones individuales de manera más legible.

Luego, el **ensamblador** toma este código ensamblador y lo convierte en código de máquina puro, es decir, en instrucciones binarias que pueden almacenarse en memoria.

Finalmente, el **enlazador** (o linker) se encarga de combinar este código con otros módulos o bibliotecas necesarias, resolviendo las referencias entre ellos y generando un archivo ejecutable que puede ser cargado y ejecutado por el sistema operativo.

La Figura 14 ilustra este proceso completo. Dado un archivo con el código fuente `hola.c`, el comando `gcc -S hola.c -o hola.s` traduce el código C a código ensamblador (`hola.s`). Este comando no genera aún código máquina ni un ejecutable. El resultado puede visualizarse con el comando `cat hola.s`. Luego, el comando `gcc -c hola.s -o hola.o` traduce el archivo ensamblador (`hola.s`) a un archivo objeto binario (`hola.o`), que contiene código máquina incompleto (sin enlazar). Finalmente, el comando `gcc hola.o -o hola` enlaza el archivo objeto con las librerías necesarias (por ejemplo, la biblioteca estándar `libc` para la función `printf`) y genera el ejecutable llamado `hola`, que puede ejecutarse con `./hola`.

Es importante destacar que, en la práctica, este proceso suele realizarse en un solo paso. En el ejemplo mostrado, basta con ejecutar el comando `gcc -o hola hola.c`.

Por otra parte, un programador puede escribir un programa completamente en lenguaje ensamblador. Uno de los requisitos para ello es definir una etiqueta global llamada `main` dentro del segmento de código. Sin embargo, escribir todo el programa en ensamblador no suele ser la mejor opción. Es preferible limitar su uso a aquellas partes que realmente lo requieran, por ejemplo, para optimizar secciones críticas o acceder directamente al hardware. Por esta razón, es común mezclar código en C con ensamblador, siempre que este último respete la convención de llamada presentada en la Sección 7.

Ejemplo

```c
// Este archivo es main.c
#include<stdio.h>
double suma(double a, double b);
int main(){
    printf("La suma es: %f\n", suma(12, 3.14));
    return 0;
}
```

donde la implementación de `suma` en ensamblador sería

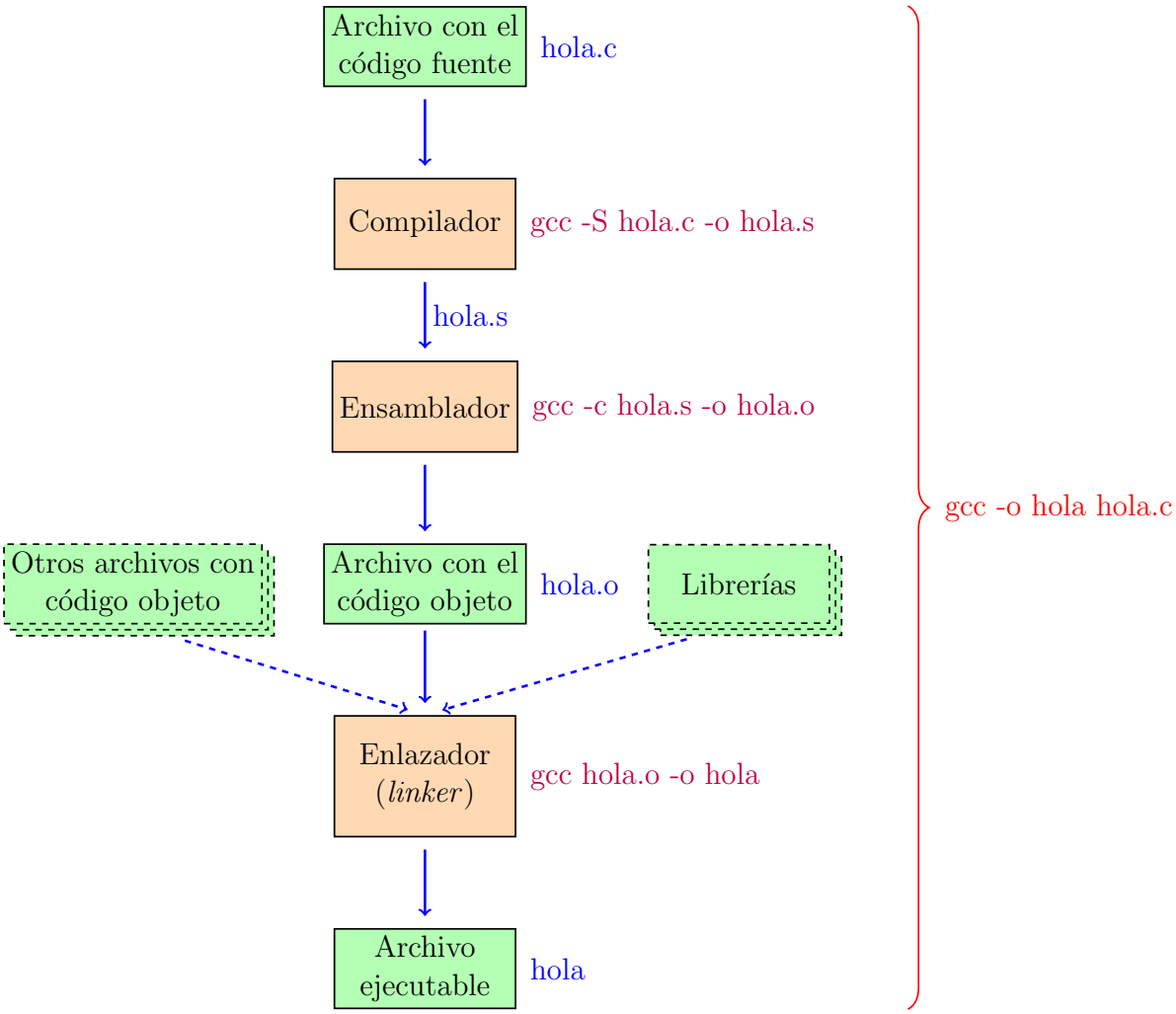


Figura 14: Proceso de compilación.

```
# Este archivo es suma.s
.global suma
suma:
   addsd %xmm1, %xmm0   # por convención de llamada el primer argumento
      viene en xmm0 y el segundo en xmm1

   ret                  # el valor de retorno en xmm0
```

Luego podemos compilar todo junto:

```
gcc -o main main.c suma.s
```

Luego podemos ejecutar mediante:

```
./main
```

y obtendremos el resultado:

```
15.140000
```

El enlazador se encargará de que la llamada a `suma` se corresponda con su implementación en ensamblador.

Apéndice B: Depurando el código con GDB

GDB o GNU Debugger es el depurador estándar para el compilador GNU. Se puede utilizar tanto para programas escritos en lenguajes de alto nivel como C y C ++ como para programas de código ensamblador.

Continuando con el ejemplo anterior, compilamos de la siguiente manera agregando la opción `-g` para incluir información en el archivo objeto para relacionarlo con el archivo fuente:

```
gcc -g -o main main.c suma.s
```

Luego podemos iniciar la sesión de depuración con GDB:

```
gdb ./main
```

Una vez iniciada la sesión, tenemos comandos para ejecutar el código línea por línea, de a tramos, visualizar contenido de memoria, registros, etc.

Para ver una guía detallada de los comandos consultar los documentos [12] y [13]. Ambos se encuentran en la Sección Apuntes varios del Campus Virtual de la asignatura. También hay disponible un vídeo tutorial en la Sección Ejemplos.

Apéndice C: Otras opciones útiles para compilar

Al momento de compilar se pueden usar diferentes banderas. A continuación, presentaremos algunas de las más útiles:

- `-S`

  Esta bandera permite obtener el código ensamblador

Ejemplo

`gcc -S -o hola hola.c`

Se obtiene un archivo `hola.s` con el código equivalente en ensamblador.

- `-g`

  Incluye información de depuración para herramientas como gdb.

- `-O0, -Og, -O1, -O2, -O3, -Ofast`

  Controlan el nivel de optimización del compilador.

Opción	Optimización	Apta para depuración	Uso típico
`-O0`	Ninguna	Sí	Desarrollo inicial, pruebas simples. Opción por defecto.
`-Og`	Básica	Sí	Depuración con algo de rendimiento.
`-O1`	Ligera	No siempre	Código final con bajo impacto.
`-O2`	Moderada	No	Código de producción estándar.
`-O3`	Alta	No	Código muy intensivo en CPU.
`-Ofast`	Máxima (sin estándares)	No	Cálculo numérico rápido, sin portabilidad garantizada.

- `-fverbose-asm`

  Se utiliza para generar un archivo de ensamblador que contiene comentarios adicionales explicativos.

- `-fomit-frame-pointer`

  Elimina el uso del puntero base (*frame pointer*) para todas las funciones. Puede ser útil en funciones pequeñas, pero puede dificultar la depuración de funciones complejas.

- `-pie`

  Compila el binario como Position Independent Executable.

- `-no-pie`

  Se usa para desactivar la generación de ejecutables como PIE (Position Independent Executable). Es decir, le indica al compilador que genere un binario con direcciones fijas, en lugar de uno que se cargue en una dirección aleatoria cada vez que se ejecuta. En general, lo utilizaremos siempre que exista un segmento `.data` para simplificar el debugging.

- `-z noexecstack`

  Se utiliza para marcar la pila como no ejecutable, lo que mejora la seguridad del programa al prevenir la ejecución de código en la pila.

- `-Wall`

  Sirve para activar un conjunto amplio de advertencias (warnings) que ayudan a detectar posibles errores o malas prácticas en el código fuente.

Referencias

[1] Andrew S. Tanembaum, *Organización de computadoras: Un enfoque estructurado*, cuarta edición, Pearson Education, 2000

[2] Paul A. Carter, *PC Assembly Language*, Disponible en formato electrónico: http://www.drpaulcarter.com/pcasm/, 2006.

[3] M. Morris Mano, *Computer system architecture*, tercera edición, Prentice-Hall, 1993.

[4] Randall Hyde, *The art of assembly language*, segunda edición, No Starch Pr, 2003.

[5] Randal E. Bryant - David R. O'Hallaron, *X86-64 Machine-Level Programming*, 2005.

[6] Bryant, Randal E, David Richard, O'Hallaron y David Richard, O'Hallaron, *Computer systems: A programmer's perspective*, Prentice Hall, 2003.

[7] *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, AMD64 Technology, 2015.

[8] *AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions*, AMD64 Technology, 2015.

[9] *X86 Assembly Language Reference Manual*, Oracle, 2012.

[10] Miquel Albert Orenga y Gerard Enrique Manonellas, *Programación en ensamblador (x86-64)*, Universitat Oberta de Catalunya (UOC), 2011.

[11] M. Matz, J. Hubicka, A. Jaeger, M. Mitchell, *System V Application Binary Interface: AMD64 Architecture Processor Supplement*, Draft Version 0.99.7, 2014.

[12] *Debugging Assembly Code with GDB.*

[13] *GDB Tutorial, A Walkthrough with Examples*, 2009.

[14] *Intel 64 and IA-32 Arquitectures Software Developer's Manual, Volume 2*, Intel, diciembre 2021.

[15] Richard Blum, *Professional Assembly Language*, Wiley Publishing, Inc., 2005.

[16] Ray Seyfarth, *Introduction to 64 Bit Intel Assembly Language Programming*, 2011.