Ensamblador y Arquitectura x86-64

Federico Bergero
Diego Feroldi

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

[*] Actualizado Septiembre 2019 (D. Feroldi, `feroldi@fceia.unr.edu.ar`)

Índice

Nota: Estas notas de clase reseñan las principales características de la arquitectura x86-64 y de su ensamblador. No es para nada una referencia completa del lenguaje ensamblador ni de la arquitectura sino que debe ser utilizado como material complementario con lo visto en las clases teóricas.

1. La arquitectura x86-64

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

La arquitectura x86-64 (AMD64 o Intel 64) de 64 bits da un soporte mucho mayor al espacio de direcciones virtuales y físicas. Proporciona registros de propósito general de 64 bits y buses de datos y direcciones también de 64 bits por lo cual las direcciones de memoria (punteros) son también valores de 64 bits. Aunque posee registros de 64 bits también permite operaciones con valores de 256, 128, 32, 16, y 8 bits.

1.1. Registros

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`. 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`). Además, dependiendo de la versión cuenta con registros adicionales para control, punto flotante, etc. Dentro de los registros hay algunos de uso especial como el `rsp` y el `rip` que son utilizados para manipular la pila (como veremos en la Sección 3.8) y apuntar a la próxima instrucción, respectivamente.

Adicionalmente, la arquitectura proporciona 16 registros SSE, cada uno de 128 bits de ancho (`xmm1-xmm15`). Intel AVX (*Advanced Vector Extensions*) proporciona además 16 registros AVX de 256 bits de ancho (`ymm0-ymm15`). Los 128 bits inferiores de `ymm0-ymm15` tienen un alias a los respectivos registros SSE de 128 bits (`xmm0-xmm15`).

La mayoría de los registros de 64 bits están divididos en subregistros de 32, 16 y 8 bits. Así, el registro `rax` de 64 bits contiene en sus 32 bits más bajos al subregistro `eax` (e por *extended*), en sus 16 bits más bajos al subregistro `ax` y a su vez `ax` se divide en dos registros de 8 bits, llamados `ah` (por *high*) y `al` (por *low*), respectivamente. Por razones históricas, esta última división en dos registros de 8 bits sólo se realiza para los registros `rax`, `rbx`, `rcx` y `rdx`. Para el resto de los registros sólo existe la parte baja de 8 bits.

Los registros introducidos en la versión de 64 bits (`r8-r16`) se dividen en `r8d` (por doble word, 32 bits), `r8w` (de word, 16 bits) y `r8b` (por byte, 8 bits).

En la Fig. 1 vemos (casi) todos los registros del x86-64 con sus subregistros y su uso durante una llamada a función.

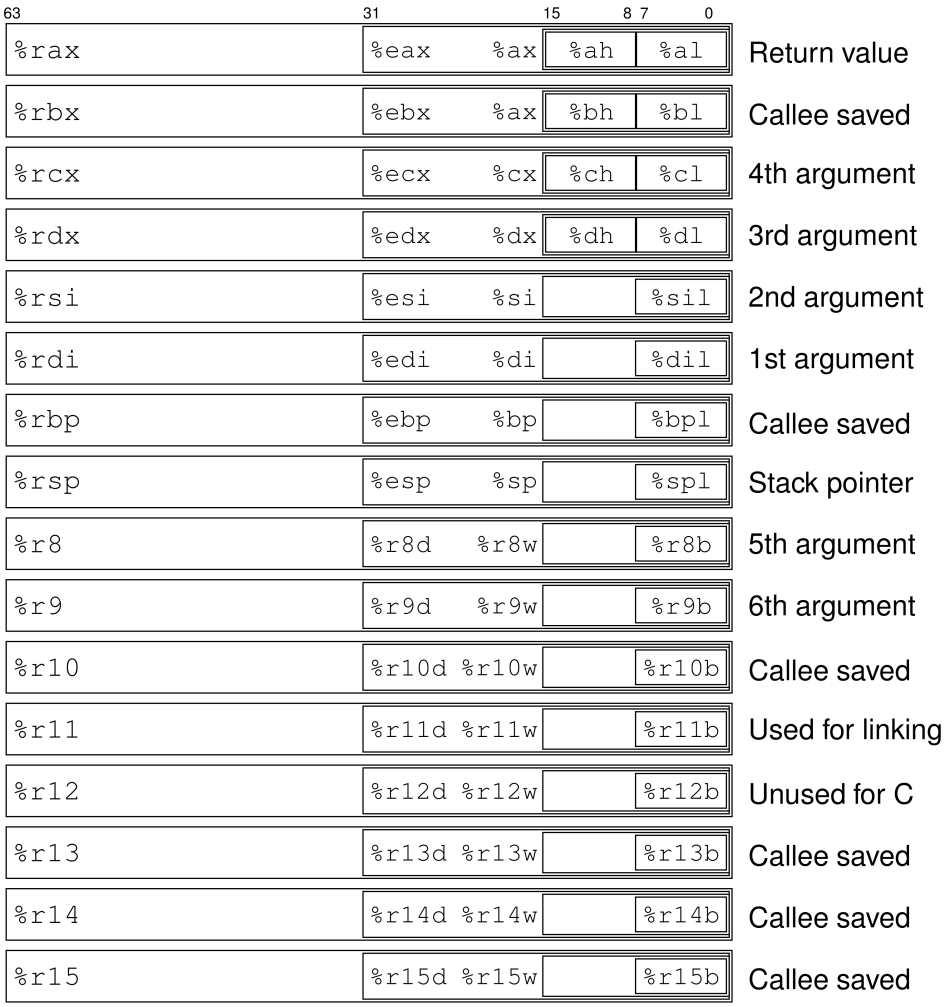


Figura 1: Registros de propósito general del x86-64 y sus subdivisiones [5].

1.2. Registros Especiales

Existen varios registros más que no son de uso general y no pueden ser modificados o leídos por las instrucciones habituales:

rip (Instruction Pointer o Contador de programa): Apunta o guarda la dirección de memoria de la próxima instrucción a ejecutar.

ss (Stack segment): Indica cuál es el segmento utilizado para la pila [1].

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).

[1] El comienzo y longitud de los segmentos son guardados en una tabla. Este registro es sólo un índice en esa tabla

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.

1.2.1. Registro `rflags`

El procesador incluye un registro especial llamado registro `rflags` o de status, en el cual se refleja el estado del procesador, información acerca de la última operación realizada y ciertos bits de control permiten cambiar el comportamiento del procesador.

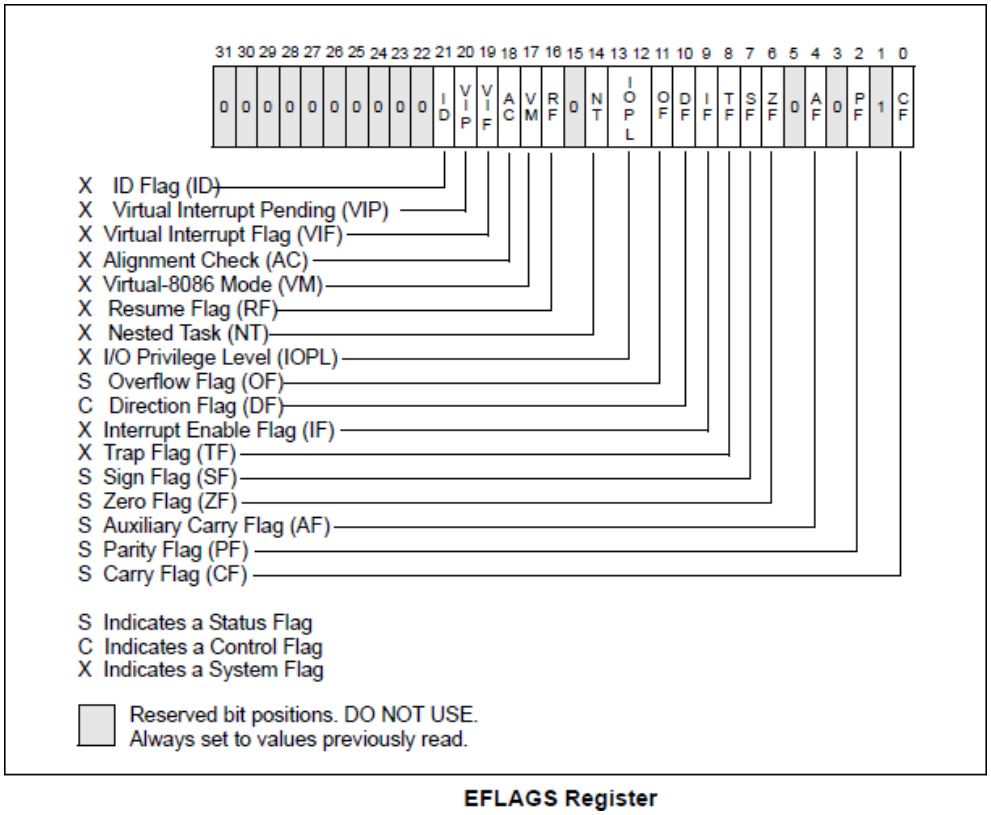


Figura 2: Registro EFLAGS

En la Fig. 2 vemos el registro EFLAGS (la versión de 32 bits del `rflags`). Vemos que hay varios bits de estado (todos los marcados con "S"). Describimos brevemente algunos de ellos:

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

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

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

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

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

El registro `rflags` no es de propósito general por lo cual no puede ser accedido ni modificado por instrucciones regulares (`add`, `mov`, etc).

1.3. Lenguaje de máquina

Los procesadores son dispositivos de hardware encargados de ejecutar el programa alojado en memoria. En la actualidad un programador escribe un programa en algún lenguaje de programación de alto nivel, por ejemplo, C, Java, Haskell, etc. La CPU no ejecuta el programa descripto en este lenguaje sino que este debe ser traducido (o compilado) a *lenguaje de máquina*.

El lenguaje de máquina es una representación muy críptica para los humanos, por ejemplo el código de máquina x86-64 (escrito en hexadecimal) de una función que suma dos enteros es como sigue:

```
48 89 f8
48 01 f0
c3
```

Para facilitar la tarea de los programadores de computadores en los años 50 se introdujo el lenguaje ensamblador, el cual tiene una representación más legible para las personas. Por ejemplo, el mismo código de la función para sumar dos enteros se escribiría en ensamblador x86-64 como:

```
movq %rdi, %rax
addq %rsi, %rax
ret
```

En este fragmento de código se ve una sintaxis de operación seguida de argumentos donde las operaciones, llamadas instrucciones, tienen un nombre representativo (`mov` por mover aunque en realidad copia un valor, `add` por adicionar, etc). Veremos de aquí en adelante qué significa cada instrucción de ensamblador y sus formas de uso.

1.4. Lenguaje Ensamblador de x86-64

En este apunte utilizaremos la sintaxis de AT&T de lenguaje ensamblador ya que es la utilizada por el Ensamblador de GNU: "as".

Detallamos aquí las principales características de esta sintaxis:

- Los comentarios de línea comienzan con `#` (a partir de `#` comienza un comentario hasta el fin de línea).
- El nombre de los registros comienza con `%`. Por ejemplo, el registro `rax` se escribe como `%rax`.
- Las constantes 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 se escribiría `$3000` por lo antes dicho).
- Las instrucciones que manipulan datos (tanto registros como memoria) se sufijan con el tamaño del dato. Por ejemplo, agregar el sufijo `q` a la instrucción `mov` resultando `movq`.

  Los sufijos posibles son los siguientes:

Sufijo	Denominación	Tamaño en bytes
`b`	*Byte*	1
`w`	*Word*	2
`l`	*Double word (o long)*	4
`q`	*Quad*	8
`t`	*Ten*	10
`s`	*Single precision float*	4
`d`	*Double precision float*	8

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.

- Las instrucciones se escriben como:

```
operando origen, destino
```

Notar que el destino es el argumento de la derecha por lo cual la instrucción `movq %rax, %rbx` representa rax → rbx (copiar el valor de `rax` a `rbx`, es decir que luego de ejecutar la instrucción `rbx=rax` y el valor del registro `rax` sigue siendo el mismo).

- Para de-referenciar un valor se utilizan los paréntesis, por ejemplo `(%rax)` refiere a lo **apuntado** por `rax`. Ejemplo: `movq (%rax), %rbx` copiará en el registro `rbx` lo apuntado por el registro `rax` y no el contenido del mismo.

  Esta notación permite también las formas:

  - `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** `$`. Ejemplo: `8(%rbp)`, `-16(%rbp)`.
  - `K(%reg1, %reg2, S)` donde `K` y `S` son constantes enteras y `S={1, 2, 4, 8}` refiere al valor `reg1 + (reg2 * S + K)`.

    Ejemplos: `(%rax,%rax,2)`, `-4(%rbp, %rdx, 4)`, `8(,%rax,4)`. En el último caso vemos que `reg1` es opcional. Este tipo de direccionamiento sirve para acceder a arreglos.

    Por ejemplo, si tenemos un arreglo de enteros de 32 bits (4 bytes) apuntado por `rax` y queremos acceder el elemento 6 podemos hacer:

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

Estos modos de direccionamiento los veremos con mayor detalle en la Sección 3.5.

2. Instrucciones

Como vimos previamente, las instrucciones de ensamblador en la arquitectura x86-64 están compuestas por una operación (por ejemplo, suma, resta, comparación, etc.) acompañada de operandos (por ejemplo, valores a sumar). En algunos casos las instrucciones no toman operandos o sus operandos son implícitos. Por ejemplo, la instrucción `ret` no toma operandos e `inc` incrementa en uno el valor de su operando, este uno está implícito.

2.1. Tipos de instrucciones

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

1. Instrucciones de transferencia de datos:
   - **mov destino, fuente**: instrucción genérica para mover un dato desde un origen a un destino. Ejemplo: `movq %rax, %rbx # rbx=rax` (Ver en detalle en la Sección 2.2).
   - **push fuente**: instrucción que mueve el operando de la instrucción al tope de la pila. Ejemplo: `pushq %rax` (Ver en detalle en la Sección 3.8).
   - **pop destino**: mueve el dato que se encuentra en el tope de la pila al operando destino. Ejemplo: `popq %rax` (Ver en detalle en la Sección 3.8).
   - **xchg destino, fuente**: intercambia contenidos de los operandos. Ejemplo: `xchg %rax, %rbx`
2. Instrucciones aritméticas y de comparación. La familia de procesadores x86 ofrece múltiples instrucciones para realizar operaciones numéricas, entre ellas:
   - **add destino, fuente**: suma aritmética de los dos operandos.
     Ejemplo: `addq %rax, %rbx # rbx+=rax`
   - **adc destino, fuente**: suma aritmética de los dos operandos considerando el bit de transporte. Ejemplo:

     ```
     movb $0, %dl
     movb $0xFF, %al
     addb $0xFF, %al  # al=0xFE, CF=1
     adcb $0, %dl     # dl=1
     ```

   - **sub destino, fuente**: resta aritmética de los dos operandos.
     Ejemplo: `subq %rax, %rbx # rbx-=rax`
   - **sbb destino, fuente**: resta aritmética de los dos operandos considerando el bit de transporte. Ejemplo:

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

   - **inc destino**: incrementa el operando en una unidad.
     Ejemplo: `incq %rax # rax++`

- **dec destino**: decrementa el operando en una unidad.
  Ejemplo: `decq %rax # rax--`
- **imul fuente**: multiplicación entera con signo. La instrucción `imul` tiene tres formatos: con un operando, con dos operandos y con tres operandos. La forma con un operando utliliza los registros `rax` y `rdx` de forma implícita: multiplica el valor del operando con `rax` y el resultado queda en `rdx:rax`. La forma con dos operandos multiplica sus dos operandos y almacena el resultado en el segundo operando. El operando resultado (es decir, el segundo) debe ser un registro. La forma con tres operandos multiplica sus operandos segundo y tercero y almacena el resultado en su último operando. Nuevamente, el operando resultante debe ser un registro. Además, el primer operando se limita a ser un valor constante. Ejemplos:

```
.data
a: .quad 4
b: .quad 5

.text
.global main
main:
movq $0xffffffffffffffff, %rax
movq $4, %rbx
mulq %rbx  # Multiplica rbx*rax y el resultado queda en rdx:rax.
           # Entonces: rax=0xfffffffffffffffc y rdx=3
movq $9, %rax
movq $3, %rbx
imulq %rbx, %rax                # rax=27
imulq a, %rax                   # rax=108
imulq $2, %rax, %rbx            # rbx=216
imulq $3, b, %rbx               # rbx=15
ret
```

- **mul fuente**: multiplicación entera sin signo. Esta instrucción a diferencia de la anterior solo admite el formato con un operando.
- **idiv fuente**: división entera con signo. La instrucción `idiv` 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`. Ejemplo:

```
movq $0xffffffffffffffff, %rax
movq $0xff, %rdx
movq $1024, %rbx
divq %rbx             # rax=0x3fffffffffffffff y rdx=0x3ff
```

- **div fuente**: división entera sin signo.
- **neg destino**: negación aritmética en complemento a 2.
  Ejemplo:

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

- **cmp destino, fuente**: comparación de los dos operandos; hace una resta sin guardar el resultado.
  Ejemplo: `cmp %rax, %rbx # Si rax==rbx entonces ZF=1, si no ZF=0`

3. Instrucciones lógicas y de desplazamiento

   a) Operaciones lógicas (Ver en detalle en el Apunte *Manejo de Bits en Lenguaje C*):

   - **and destino, fuente**: operación "*y*" lógica.
     Ejemplo: `andq %rax, %rbx # rbx=rax&rbx`
   - **or destino, fuente**: operación "*o*" lógica.
     Ejemplo: `orq %rax, %rbx # rbx=rax|rbx`
   - **xor destino, fuente**: operación "*o exclusiva*" lógica.
     Ejemplo: `xorq %rax, %rax # rax=rax^rax=0`
   - **not destino**: negación lógica bit a bit.
     Ejemplo:
     ```
     movb $0xff, %al
     notb %al        # al=0
     ```
   - **test destino, fuente**: comparación lógica de los dos operandos; hace una "*y*" lógica sin guardar el resultado.
     Ejemplo: `test %cl, %cl   # ZF=1 si cl=0 y SF=1 si cl<0`

   b) Operaciones de desplazamiento (Ver en detalle en el Apunte *Manejo de Bits en Lenguaje C*):

   - **sal cantidad, destino / shl cantidad, destino**: desplazamiento aritmético/lógico a la izquierda (estas dos instrucciones producen el mismo resultados). Ejemplo:
     ```
     movl $0xaa, %al
     salb $1, %al    # al=0x54
     ```
   - **sar cantidad, destino**: desplazamiento aritmético a la derecha. Ejemplo:
     ```
     movl $0xaa, %al
     sarb $1, %al    # al=0xd5
     ```
   - **shr cantidad, destino**: desplazamiento lógico a la derecha. Ejemplo:
     ```
     movl $0xaa, %al
     shrb $1, %al    # al=0x55
     ```
   - **rol cantidad, destino**: rotación lógica a la izquierda. Ejemplo:
     ```
     movl $0xaa, %al
     rolb $1, %al    # al=0xd5
     ```
   - **ror cantidad, destino**: rotación lógica a la derecha. Ejemplo:
     ```
     movl $0xaa, %al
     rorb $1, %al    # al=0x55
     ```
   - **rcl cantidad, destino**: rotación lógica a la izquierda considerando el bit de transporte. Ejemplo:

```
movl $0xaa, %al
stc # CF=1
rclb $1, %al    # al=0x55
```

- **rcr cantidad, destino**: rotación lógica a la derecha considerando el bit de transporte. Ejemplo:

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

4. Instrucciones de ruptura de secuencia (Ver en detalle en la Sección 2.3)

   a) Salto incondicional:

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

   b) Saltos que consultan un bit de resultado concreto:

   - **je etiqueta / jz etiqueta**: salta a la etiqueta si igual, si el bit de cero está activo.
   - **jne etiqueta / jnz etiqueta**: salta a la etiqueta si diferente, si el bit de cero no está activo.
   - **jc etiqueta / jnc etiqueta**: salta a la etiqueta si el bit de transporte está activo.
   - **jnc etiqueta**: salta a la etiqueta si el bit de acarreo no está activo.
   - **jo etiqueta**: salta a la etiqueta si el bit de desbordamiento está activo.
   - **jno etiqueta**: salta a la etiqueta si el bit de desbordamiento no está activo.
   - **js etiqueta**: salta a la etiqueta si el bit de signo está activo.
   - **jns etiqueta**: salta a la etiqueta si el bit de signo no está activo.

   c) Saltos condicionales sin considerar el signo:

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

   d) Saltos condicionales considerando el signo:

   - **jl etiqueta / jnge etiqueta**: salta si es más pequeño.
   - **jle etiqueta / jng etiqueta**: salta si es más pequeño o igual.
   - **jg etiqueta / jnle etiqueta**: salta si es mayor.
   - **jge etiqueta / jnl etiqueta**: salta si es mayor o igual.

   e) Otras instrucciones de ruptura de secuencia:

   - **loop etiqueta**: decrementa el registro `rcx` y salta si `rcx` es diferente de cero (Ver en detalle en la Sección 2.3.3).
   - **call etiqueta**: llamada a (Ver en detalle en la Sección 5).
   - **ret**: retorno de subrutina (Ver en detalle en la Sección 5).

5. Instrucciones para el registro `rflags`

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

   Apagar un bit: `clc` (*clear carry*), `cld` (*clear direction*).

   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` y `sahf` copian ciertos bits del registro `ah` hacia el flags y viceversa, `popfq` guarda en la pila el registro flags y `pushfq` trae de la pila el registro flags.

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

6. Instrucciones de entrada/salida

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

2.2. Instrucciones para copia 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`. Esta instrucción toma la forma *movS origen, destino* donde "S" es el sufijo [2]. Los operandos pueden ser registros, memoria o constantes inmediatas. Por ejemplo, para escribir un valor 3 en el registro `rax` podemos hacer:

```
movq $3, %rax
```

Algunos ejemplos de uso de `mov` (con el equivalente en C en lo posible):

```
movb $65, %al          # al = 'A'
movq %rax, %rcx        # rcx=rax
movw (%rax), dx        # copia en dx dos bytes comenzando
                       # en la dirección rax
movw dx, (%rax)        # copia dx en la dirección rax
movl 16(%rbp), %ecx    # copia en ecx cuatro bytes (debido al sufijo l)
                       # comenzando en la dirección rbp+16
```

[2] Recordar que la instrucción `mov` no mueve sino que copia valores

2.3. Comparaciones, Saltos y Estructuras de Control

2.3.1. Saltos

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. Veremos la más simple primero.

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 3.2). Veamos un 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.

Los saltos condicionales tienen la misma función que la instrucción `jmp` salvo que se realizan **sólo si** se da una condición, por ejemplo, el resultado de la última operación fue cero. Como vimos en la Sección 1.2.1, el procesador mantiene en el registro `rflags` el estado de la última operación realizada. Luego, los saltos condicionales de x86-64 hacen uso de este registro y realizan el salto si cierto bit de ese registro está en 1. De hecho, por cada bit de estado del registro `rflags` hay dos saltos condicionales, por ejemplo `jz` realiza el salto si el bit ZF está en uno y `jnz` lo realiza si el bit ZF **no** está en uno.

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 una instrucción para comparar dos valores, 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 de `rbx-rax`, se 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 los saltos condicionales por desigualdad: `jl` y `jg` (por lower y greater) para datos con signo y `ja` y `jb` (por above y below) para datos sin signo.

2.3.2. Estructuras de Control

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

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

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

```
.data
a: quad 0
.text

 cmpq $100, a # comparamos el valor de a con la constante 100
 jz igual_a_cien # si el resultado dio cero (rax-100) es porque son iguales
                 # en este caso debo incrementar a
 jmp seguir
igual_a_cien:
  incq a
  jmp seguir:
seguir:
```

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:

```
 .data
  a: quad 0

 cmpq $100, a # comparamos el valor de a con la constante 100
 jz igual_a_cien # si el resultado dio cero (rax-100) es porque son iguales
                 # en este caso debo incrementar a
 decq a
 jmp seguir

igual_a_cien:
  incq a
  #jmp seguir

seguir:
 ...
 ...
```

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.

2.3.3. 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. Supongamos que queremos traducir la siguiente estructura:

```
 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 condicion
  je fin_1   # Si resulto falsa, el lazo termino

cuerpo_del_while_1:  # Aca ira el cuerpo del while
```

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

El código corresponde a la estructura de control que puede verse en la Fig. 3.

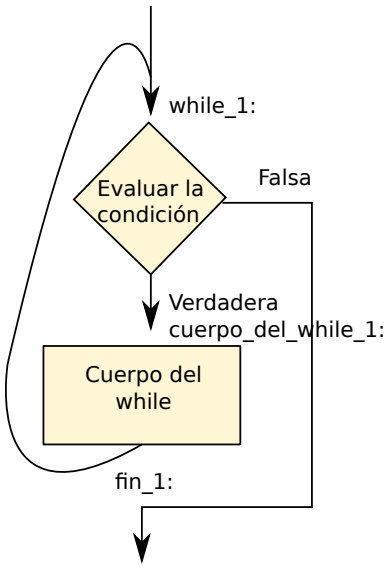


Figura 3: Estructura de *While*.

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. Por ejemplo es muy común algo como:

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

De hecho esto es tan común que la arquitectura incluye la instrucción `loop` para implementar estas estructuras. Esto puede traducirse como:

```
movq $100, %rcx # rcx se utiliza como iterador. Inicialmente 100

cuerpo_del_for_1:
  ...
  ...
  ...
  loop cuerpo_del_for_1
```

La instrucción `loop` tiene dos efectos:

- Decrementa en uno el registro `rcx`. Aquí vemos que `rcx` tiene un uso especial.
- Luego, salta a 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 al `loop`.

2.4. 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 esta estructura de datos es muy utilizada 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. 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).

2.4.1. Copia y manipulación de datos

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

lods (de *load string*) Copia en el registro `rax` (o en su sub-registro correspondiente) el valor apuntado por `rsi` e incrementa/decrementa `rsi` 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.

stos (de *store string*) Almacena el valor del registro `rax` (o su sub-registro correspondiente) en la dirección apuntada por `rdi` y luego incrementa/decrementa el valor de `rdi` en la cantidad de bytes indicada por el sufijo de tipo. Así la instrucción `lodsl` (asumiendo DF=1) equivale a:

[3] Aunque su nombre sugieren que son índices, estos registros son apuntadores.
[4] Algunas instrucciones solo incrementan/decrementan uno de estos registros.

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

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).

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 convencion de llamada tenemos en rdi el puntero a "a"
  # y en rsi el puntero a 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 "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.

2.4.2. 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:

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.

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`.

Veamos un caso de uso de estas instrucciones. 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
```

2.4.3. Iteraciones de 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 2.4.1 puede ser reescrito en ensamblador como:

```
.global f
f:
  # por convencion de llamada tenemos en rdi el puntero de a
  # y en rsi el puntero a b
  movq $100, %rcx # debemos iterar 100 veces
  cld  # iremos incrementando rsi y rdi (DF=0)
  rep movsb # repite movsb rcx 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 2.4.2 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 rcx 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
```

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.

2.5. Instrucciones de Conversión

Las instrucciones de conversión de datos realizan varias transformaciones de datos, como duplicación de tamaño de operando mediante extensión de signo, conversión de formato little-endian a big-endian, extracción de máscaras de signos, búsqueda en una tabla y soporte para operaciones con números decimales.

La arquitectura x86-64 ofrece instrucciones para convertir entre enteros de distintos tamaño:

cbw Extiende (con signo) `al` a `ax`.

cwde Extiende (con signo) `ax` a `eax`.

cwd Extiende (con signo) `ax` a `dx:ax` (los 16 bits altos a `dx` y los 16 bits mas bajo a `ax`).

cltq Extiende (con signo) `eax` a `edx:eax` (los 32 bits altos a `edx` y los 32 bits bajos a `eax`).

cqto Extiende (con signo) `rax` a `rdx:rax` (los 64 bits altos a `rdx` y los 64 bits bajos a `rax`).

movs Copia un valor del origen al destino extendiendo con el bit de signo. Esta instrucción se utiliza 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. Algunos ejemplos:

```
movsbl %al, %eax # convierte un byte a un long
movswl %ax, %eax # convierte un word a un long
movswq %ax, %rax # convierte un word a un quad
```

movz Copia un valor del origen al destino extendiendo con *cero*. Esta instrucción se utiliza 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. Algunos ejemplos:

```
movzbl %al, %eax # convierte un byte a un long
movzwl %ax, %eax # convierte un word a un long
movzwq %ax, %rax # convierte un word a un quad
```

3. Acceso a datos en memoria

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

3.1. Directivas al Ensamblador

Al igual que el compilador de C permite al programador indicar ciertas operaciones de preprocesamiento (`#include`, `#define`, `#pragma`, etc.), el compilador de ensamblador **as** permite al programador indicar ciertas operaciones y valores inicializados. Las directivas al compilador ensamblador comienzan siempre con ".". Dentro de ellas destacamos las siguientes:

Describir el segmento Con éstas el programador indica a **qué** segmento debe agregarse el siguiente bloque. Las más comunes son `.data` (el siguiente bloque debe ir al segmento de datos) y `.text` (indicando que lo que sigue es código ejecutable).

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

ascii, asciz Permiten inicializar una lista de cadenas con y sin carácter nulo al final de cada una. Ejemplos: `.asciz "Hola mundo"`, `.ascii "a" "b" "c"` (este ejemplo es una lista de strings).

byte Inicializa una lista de bytes. Ejemplo: `.byte 'a' 'b'`, `.byte 97`, `.byte 0x61`.

double, float Inicializa una lista de valores de punto flotante de doble y simple precisión, respectivamente. Ejemplo: `.double 3.1415 2.16`, `.float 5.3`.

short, long, quad Emite una lista de valores enteros de 2, 4 y 8 bytes. Ejemplo: `.short 20`, `.long 50`, `.quad 0`.

space Emite un bloque de tamaño fijo inicializado en cero o en un valor pasado como argumento. Ejemplo: `.space 128`, `.space 5000, 0`. Esta directiva es útil para obtener un bloque de memoria de tamaño dado (ya sea inicializado o no).

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:

```
.data
.long
```

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

Definir una etiqueta global `.global` indica que la etiqueta nombrada es de alcance global. De no especificar esta directiva la etiqueta desaparece luego del proceso de

compilación. Ésta debe ser utilizada, por ejemplo, con las etiquetas que definan funciones que serán llamadas fuera del archivo ensamblador. 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).

Ejemplos

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

3.2. Etiquetas

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

Por ejemplo, cuando uno define en C una variable (`long i;`) está indicándole al compilador que reserve espacio de memoria para un entero y que este espacio lo nombraremos 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 usada por la computadora sino que una etiqueta se convierte en una **dirección de memoria**.

Ejemplo

En ensamblador con sintaxis AT&T una etiqueta es un nombre seguido de "`:`".

```
.data
i: .long  0
f: .double 3.14

.text
.global main
main:
movq $0, %rax
retq
```

Aquí vemos que la etiqueta `i` *(dentro del segmento de datos) 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).*

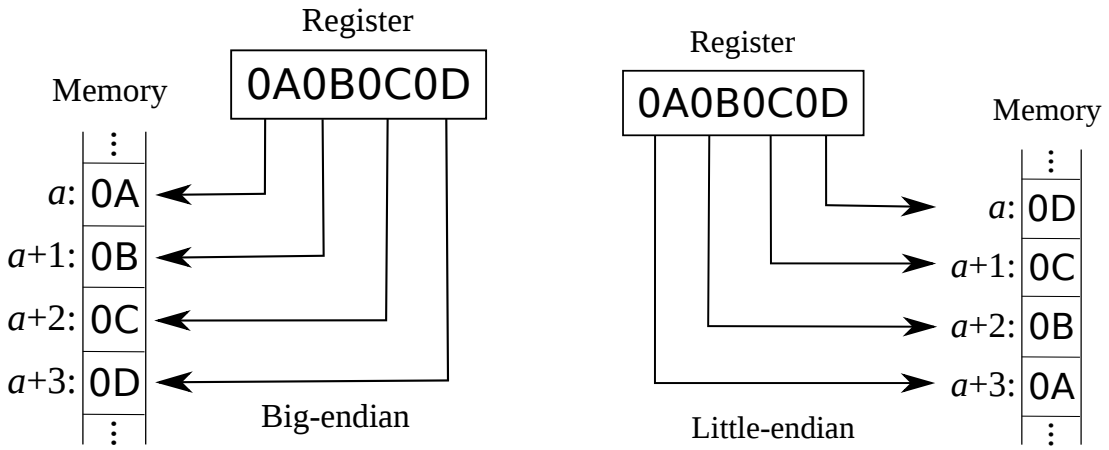
Finalmente, vemos que dentro del segmento de código se define una etiqueta ***global*** *llamada* ***`main`****. Este será el punto de inicio de todo programa.*

3.3. 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 de los bytes.

Supongamos que tenemos que almacenar el entero 168496141 en la dirección de memoria `a`. Este valor se representa mediante los cuatro bytes 0x0A 0x0B 0x0C 0x0D (escribiendo más a la izquierda el valor más representativo).

Una opción es guardar el byte **más** significativo (0x0A) en la dirección `a`, el segundo (0x0B) en la dirección `a+1`, y así. Esto se conoce como convención *Big-Endian* como puede verse en la Fig. 4(a).



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

Figura 4: Convenciones de *Endianness* (Fuente Wikipedia).

Otra opción es almacenar en la dirección `a` el byte **menos** significativo (0x0D), el siguiente (0x0C) en la dirección `a+1` y así. Esta última convención se denomina *Little-Endian* y es la utilizada por las arquitecturas x86 y por la tanto también por x86-64 . La Fig. 4(b) muestra la convención *Little-Endian*.

3.4. Definición de variables

La declaración de variables en un programa en ensamblador se puede incluir en la sección `.data`. Las variables de esta sección se definen utilizando las directivas vistas en la Sección 3.1. 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`*):*

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

Si ejecutamos la instrucción `mowq var1, %rax`*, el procesador tomará como primer byte el valor de* `var1`*, pero también 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. Por una parte, el acceso a los datos es muy flexible, pero, por otra parte, si no controlamos muy bien el acceso a las variables, esta flexibilidad puede causar ciertos problemas.*

3.5. Acceso a datos: modos de direccionamiento

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

1. **Inmediato.** En este caso, el operando hace referencia a un dato que se encuentra en la propia instrucción. El valor especificado debe poder ser expresado con 32 bits como máximo, que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos. También se puede sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable), con la excepción de la instrucción `mov` cuando el segundo operando es un registro de 64 bits, para el que podremos especificar un valor que se podrá expresar con 64 bits.

   Ejemplos

   ```
   movq 0x1122334455667788, %rax # Carga en el registro rax el valor
                                 # 0x1122334455667788
   movb $100, var # Carga el valor 100 en la dirección de memoria var
   movq $var, %rbx # Carga el valor de la dirección de memoria de la variable
                   # var en el registro rbx
   movq $var+16*2, %rbx # Carga en el registro rbx el valor de la dirección
                        # de memoria de la variable var más 32
   ```

2. **Directo a registro**. En este caso, el operando hace referencia a un dato que se encuentra almacenado en un registro (no hay acceso a memoria). En este modo de direccionamiento podemos especificar cualquier registro de propósito general.

   Ejemplo

   ```
   movq %rax, %rbx # rbx=rax
   ```

3. **Directo a memoria.** En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando habrá de especificar el nombre de una variable de memoria.

   Ejemplos

   ```
   movq var, %rax   # Carga en el registro rax 8 bytes a partir de la
                    # dirección de memoria var
   movq %rax, var   # Carga el contenido del registro rax en la dirección
                    # de memoria var
   ```

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

   Ejemplo

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

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

   Ejemplos

```
movq 2(%rax), %rbx   # Carga en el registro rbx 8 bytes a partir de la
                     # dirección de memoria rax+2
movq var(%rax), %rbx # Carga en el registro rbx 8 bytes a partir de la
                     # dirección de memoria rax+var
```

6. **Relativo.** En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando especifica una dirección de memoria de la siguiente manera:

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

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 que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos:

$$desplazamiento(registro\ base,\ registro\ índice,\ escala)$$

También podemos sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable). Podemos especificar solo los elementos que nos sean necesarios.

Ejemplos

```
movwq 3(%rbc, %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.
```

3.6. Comentario sobre acceso a memoria

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
```

Es importante notar que aunque la etiqueta `i` es una constante (la dirección de memoria donde se aloja ese entero) no lleva el signo `$`.

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

```
addq i, %rax
```

Sin embargo, notar que `addq $i, %rax` produce un efecto muy difrerente. 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.

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

Ejemplo

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

.text
.global main
main:
movq str , %rax # Instruccion 1
movq $str, %rax # Instruccion 2
retq
```

¿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. Notemos que en este caso ningún carácter de esa cadena será copiado a* `rax`*. De hecho esa instrucción no accede a la memoria.*

3.7. Instrucción "*load effective address*"

La arquitectura x86-64 ofrece una instrucción similar al operador de referencia de C. Esta instrucción del ejemplo es `lea` (por "*load effective address*"). Así la instrucción `mov $str, %rax` es equivalente a

```
leaq str, %rax
```

Las instrucciones `lea` y `mov` (desde memoria) están relacionadas: `mov` carga el contenido de una dirección de memoria mientras que `lea` carga la dirección en sí.

La instrucción `lea` a menudo se usa como un "truco" para hacer ciertos cálculos, aunque ese no sea su propósito principal. Usando sintaxis AT&T, los modos de direccionamiento útiles con `lea` son los siguientes:

```
lea desplazamiento(%base), %dest
lea (%offset, multiplicador), %dest
lea desplazamiento(, %índice, multiplicador), %dest
lea (%base, %índice, multiplicador), %dest
lea desplazamiento(%base, %índice, multiplicador), %dest
```

lo cual corresponde a

```
%dest = desplazamiento + %base
%dest = %índice * multiplicador
%dest = desplazamiento + %índice * multiplicador
%dest = %base + %índice * multiplicador
%dest = desplazamiento + %base + %índice * multiplicador
```

donde el desplazamiento es una constante entera, el multiplicador es 2, 4 u 8, y `%dest`, `%índice` y `%base` son registros.

Se puede usar esto para multiplicar un registro por 2, 3, 4, 5, 8, o 9, y sumar una constante en solo paso:

```
lea constante(, %src, 2), %dst
lea constante(%src, %src, 2), %dst
lea constante(, %src, 4), %dst
lea constante(%src, %src, 4), %dst
lea constante(, %src, 8), %dst
lea constante(%src, %src, 8), %dst
```

donde `%src` y `%dst` pueden ser el mismo registro.

3.8. Gestión de la pila

Una pila es una estructura de datos que permite almacenar información. Su funcionamiento puede analizarse pensando en una pila de platos sobre una mesa. Uno puede agregar platos y la pila irá creciendo. Luego, si uno quiere sacar un plato quitará el plato del "tope", achicando la pila de platos. Como se ve, cuando uno saca un elemento de la pila, sacará el último elemento insertado (si lo hubiera). Por ello la estructura de datos pila se conoce como *LIFO (Last-In First-Out)*, dado 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. Esto se conoce como el segmento de pila (que **no** es el mismo segmento que el segmento de datos ni de código).

La pila puede usarse para varias cosas:

- Espacio de almacenamiento temporario. Las variables automáticas de C por ejemplo se almacenan en la pila.
- Implementar el llamado a función (y en especial las recursivas). Notar que el orden de llamada y finalización de las funciones ocurre como una pila. Así, si tenemos que la función `f` llama a `g` y `g` llama a `h`, la primera función que finalizará es `h`, luego `g` y finalmente `f`.
- Preservar el valor de registros durante un llamado a función. Como veremos en la Sección 5, algunos registros son modificados cuando uno realiza un llamado a función. El programador puede guardar el valor de ese registro en la pila y restaurarlo luego de la llamada.

Aunque la arquitectura permite utilizar la pila con cualquier fin, es muy común que cada **función** utilice una porción de la pila para guardar sus variables locales, argumentos, dirección de retorno, etc. A esta sub-porción de pila se la conoce como **marco de activación** de la función. En la Fig. 5 vemos un posible estado de la pila con diferentes marcos de activación (sólo **uno** está activo en un momento dado, el de la función que se está ejecutando).

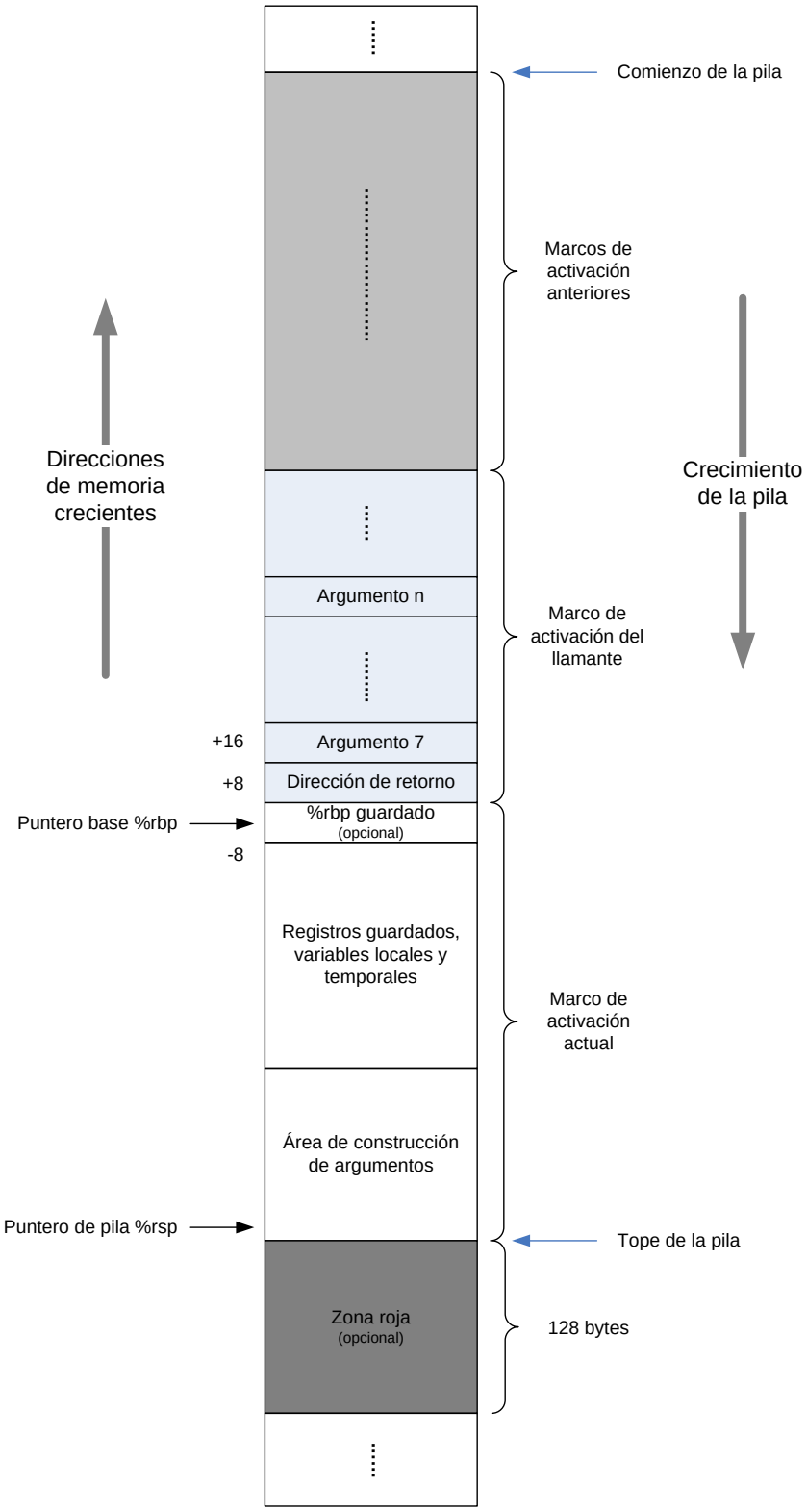


Figura 5: Diagrama de la estructura de pila x86-64.

En la Fig. 5 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).

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 este uso particular pueden ser manipulados por las instrucciones habituales (`add`, `mov`, etc). La arquitectura x86-64 ofrece también dos instrucciones especiales para apilar/desapilar elementos:

pushq 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 6.

popq 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
```

El comportamiento de la instrucción `popq` puede verse en la Fig. 7. Notar que el valor `0x12345678` continua almacenado en la dirección `0xff8`.

En la descripción de las instrucciones `push` y `pop` utilizamos el sufijo `q` (entero de 8 bytes) ya que por cuestiones de alineación los datos insertados en la pila deben ser de 8 bytes.

En la Fig. 5 se puede observar un área denominada ***zona roja***. La arquitectura x86-64 especifica que los programas pueden usar los 128 bytes más allá del puntero de la pila actual (es decir, en direcciones más bajas que el puntero). Así, el área de 128 bytes más allá de la ubicación señalada por `rsp` se considera reservada y no debe modificarse mediante señales o manejadores de interrupciones. Por lo tanto, las funciones pueden usar esta área para datos temporales que no son necesarios en las llamadas a funciones. En particular, las funciones de "hoja" (*leaf functions*) pueden usar esta área directamente,

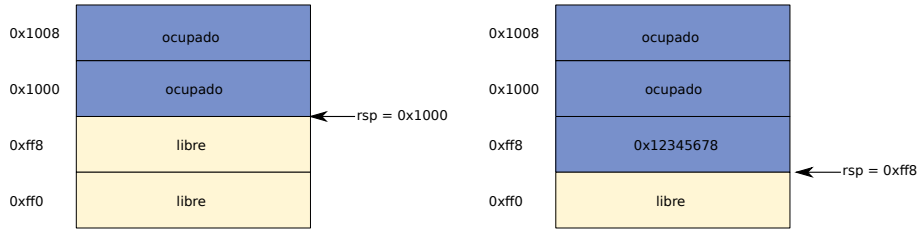


Figura 6: Diagrama de la memoria antes y después de ejecutar la instrucción `pushq`.

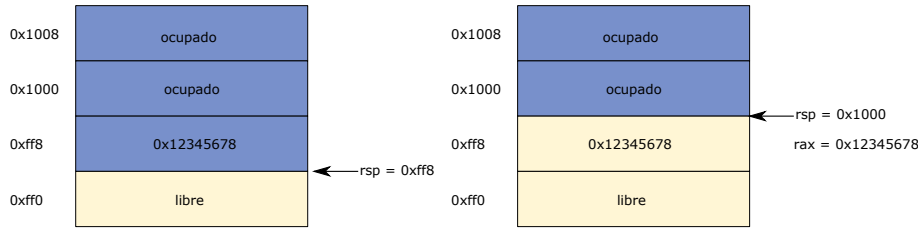


Figura 7: Diagrama de la memoria antes y después de ejecutar la instrucción `popq`.

en lugar de ajustar el puntero de la pila en el prólogo y el epílogo. La zona roja es una optimización. El código puede suponer que los 128 bytes por debajo de `rsp` no se modifican por señales o controladores de interrupción, y por lo tanto puede usarlo para datos temporales, sin mover explícitamente el puntero de pila. Sin embargo, hay que tener en cuenta que la zona roja será utlizada por llamadas a función, por lo que generalmente es más útil en funciones que no llaman a otras funciones.

4. 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 no utilizan los registros de propósito general sino que utilizan registros dedicados especiales. Éstos son 16 registros de 128 bits, del `xmm0` al `xmm15`. Además, pueden contener múltiples elementos (denominados "*packed format*") de distinto tamaño. En particular para cálculos de punto flotante pueden contener dos `double` o cuatro `float`.

Veremos primero las instrucciones de carga y movimiento, luego las operaciones aritméticas escalares (no "*packed*") y luego las operaciones *Single Instruction Multiple Data* (SIMD).

4.1. Copias y conversiones

Al igual que con los registros de propósito general, existen para los registros de punto flotante las instrucciones `movss` y `movsd` que copian un dato de precisión simple (*float*) y doble precisión (*double*) respectivamente de un registro `xmm` a uno de propósito general.

A su vez existen múltiples instrucciones para convertir entre enteros y datos de punto flotante. En la Fig. 8 se recopilan las instrucciones de conversión.

Veamos por ejemplo como inicializar una variable de tipo double (en el registro `xmm0`) con el valor de 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
```

Instruction	Source	Destination	Description
`movss`	M_{32}/X	X	Move single precision
`movss`	X	M_{32}	Move single precision
`movsd`	M_{64}/X	X	Move double precision
`movsd`	X	M_{64}	Move double precision
`cvtss2sd`	M_{32}/X	X	Convert single to double precision
`cvtsd2ss`	M_{64}/X	X	Convert double to single precision
`cvtsi2ss`	M_{32}/R_{32}	X	Convert integer to single precision
`cvtsi2sd`	M_{32}/R_{32}	X	Convert integer to double precision
`cvtsi2ssq`	M_{64}/R_{64}	X	Convert quadword integer to single precision
`cvtsi2sdq`	M_{64}/R_{64}	X	Convert quadword integer to double precision
`cvttss2si`	X/M_{32}	R_{32}	Convert with truncation single precision to integer
`cvttsd2si`	X/M_{64}	R_{32}	Convert with truncation double precision to integer
`cvttss2siq`	X/M_{32}	R_{64}	Convert with truncation single precision to quadword integer
`cvttsd2siq`	X/M_{64}	R_{64}	Convert with truncation double precision to quadword integer

X: XMM register (e.g., `%xmm3`)
R_{32}: 32-bit general-purpose register (e.g., `%eax`)
R_{64}: 64-bit general-purpose register (e.g., `%rax`)
M_{32}: 32-bit memory range
M_{64}: 64-bit memory range

Figura 8: Instrucciones de copia y conversiones [5].

4.2. Operaciones de 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 Fig. 9 resume las operaciones más utilizadas par simple y doble precisión.

Single	Double	Effect	Description
`addss`	`addsd`	$D \leftarrow D + S$	Floating-point add
`subss`	`subsd`	$D \leftarrow D - S$	Floating-point subtract
`mulss`	`mulsd`	$D \leftarrow D \times S$	Floating-point multiply
`divss`	`divsd`	$D \leftarrow D \,/\, S$	Floating-point divide
`maxss`	`maxsd`	$D \leftarrow \max(D,\ S)$	Floating-point maximum
`minss`	`minsd`	$D \leftarrow \min(D,\ S)$	Floating-point minimum
`sqrtss`	`sqrtsd`	$D \leftarrow \sqrt{S}$	Floating-point square root

Figura 9: Instrucciones de copia y conversiones [5].

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

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

Veremos en la siguiente Sección 5 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
```

```
# Carga el valor 1.8 en xmm1
# La constante es la representacion según IEEE 754 de 1.8
movq $4610785298501913805, -8(%rsp)
movsd -8(%rsp), %xmm1
# Carga el valor 32.0 convirtiendo el valor entero 32 de rax a xmm2
movq $32, %rax
cvtsi2sdq %rax, %xmm2
# xmm0=xmm0*xmm1 => xmm0=t*1.8
mulsd %xmm1, %xmm0
# xmmo=xmm0+xmm2 => xmmo=t*1.8+32
addsd %xmm2, %xmm0
# como el valor de retorno se escribe en xmm0 hemos terminado
ret
```

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 de `rflags`. Las instrucciones de comparación en punto flotante son:

ucomiss para comparación de precisión simple.

ucomisd para comparación de precisión doble.

4.3. Instrucciones *packed* - SSE

El software para procesamiento de señales multimedia (audio, imágenes, video, etc) muchas veces requiere repetir la misma operación en una gran cantidad de datos, por ejemplo para cada píxel realizar una determinada operación. Por ello, las arquitectura actuales incluyen lo que se conoce como instrucciones *Single Instruction Multiple Data* (SIMD). Es decir, que aplican la misma operación a muchos datos a la vez.

Hasta ahora vimos instrucciones que operan sobre valores de simple y doble precisión (32 y 64 bits). Los registros `xmm` son de 128 bits por lo cual éstos pueden alojar 4 valores de precisión simple o 2 de precisión doble. A su vez, existen instrucciones SSE (*Streaming SIMD Extensions*) que operan sobre enteros de 8, 16, 32 y 64 bytes. Por ello, cada registro `xmm` puede contener 16 bytes, 8 words, 4 enteros de 32 bits y 2 de 64 bits. En la Fig. 10 se ve los distintos valores que puede contener un registro `xmm`.

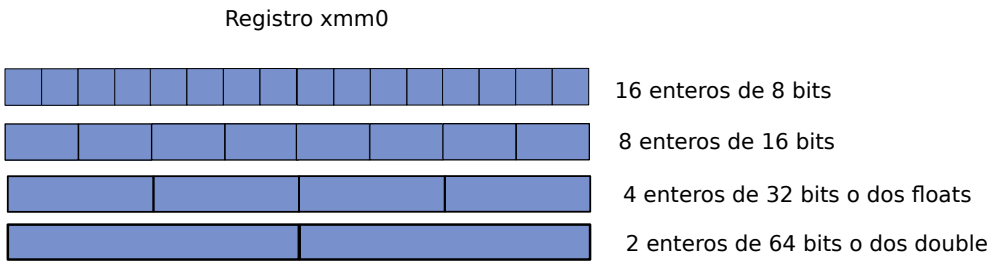


Figura 10: El contenido de un registro `xmm` (128 bits).

Veamos un ejemplo. Dada la siguiente función C:

```
void sum(float a[4], float b[4]) {
```

```
  int i;
  for (i=0;i<4;i++)
    a[i]=a[i]+b[i];
}
```

El código equivalente en lenguaje ensamblador es:

```
 .global sum
sum:
  # notar que en este caso los argumentos
  # son punteros y vienen en rdi y rsi respectivamente
  # copia los 4 floats de "a" a xmm0
  movaps (%rdi), %xmm0
  # copia los 4 floats de "b" a xmm1
  movaps (%rsi), %xmm1
  # suma los 4 floats a la vez
  addps %xmm0, %xmm1
  # guarda el resultado en "a"
  movaps %xmm1, (%rdi)
  ret
```

Aquí la instrucción interesante es la `addps` que suma los 4 valores de precisión simple a la vez (*packed*, que podríamos traducir como empaquetado). La Fig. 11 grafica esta instrucción.

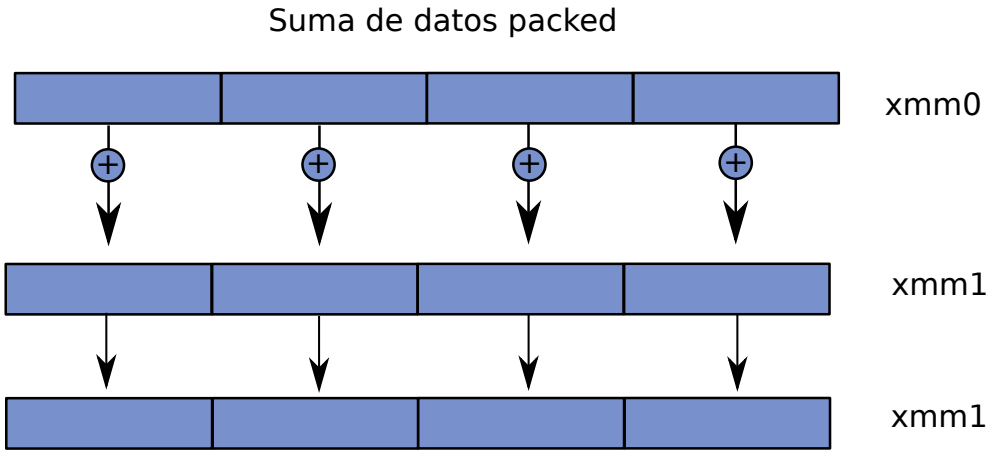


Figura 11: Instrucción `addps`

Hay varios tipos de instrucciones SSE:

- Instrucciones SSE de Transferencia de datos.
- Instrucciones SSE de Conversión.
- Instrucciones SSE Aritméticas.
- Instrucciones SSE lógicas.

Mnemotécnico	Descripción
movaps	Mueve cuatro flotantes simple precisión alineados entre registros XMM o memoria
movss	Mueve flotantes de simple precisión entre registros XMM o memoria
addps	Suma flotantes simple precisión empaquetados
addss	Suma flotantes simple precisión escalares
divps	Divide flotantes simple precisión empaquetados
divss	Divide flotantes simple precisión escalares
mulps	Multiplica flotantes simple precisión empaquetados
mulss	Multiplica flotantes simple precisión escalares
subps	Resta flotantes simple precisión empaquetados
subss	Resta flotantes simple precisión escalares
cmpps	Compara flotantes simple precisión empaquetados
cmpss	Compara flotantes simple precisión escalares
comiss	Compara flotantes simple precisión escalares y setea banderas en el registro EFLAGS
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

Tabla 1: Algunas intrucciones *SSE*.

La Tabla 1 muestra algunas instrucciones. Sin embargo, las extensiones SSE contienen muchas más instrucciones. En este apunte sólo se pretende dar una introducción. Un listado completo se puede consultar en [8] o [9]. Por otra parte, en el 2011 se introdujo una nueva tecnología de instrucciones SIMD llamadas AVX, pero estas no serán vistas en este apunte.

5. Funciones y Convención de Llamada

Otra parte fundamental del código estructurado son los procedimientos y funciones. Una función dentro de un programa puede pensarse como una función matemática que se aplica a ciertos valores del dominio y arroja un valor en el conjunto de llegada.

Así vemos por ejemplo que la función C:

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

tomará dos enteros largos y devolverá otro entero largo.

Desde el punto de vista del procesador una llamada a función es muy similar a un salto ya que el flujo del programa debe ser modificado (para ejecutar el código de la función llamada). La diferencia radica en que, como el código es secuencial, luego de una llamada a función el flujo del programa debe continuar la ejecución **con el código que sigue** a la llamada. Veamos esto en C:

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

Aquí vemos tres instrucciones. La segunda es una llamada a la función `printf` con dos argumentos, una cadena de caracteres `"%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` que debe continuar con esa instrucción (siendo que `printf` podría ser llamada de múltiples lugares distintos)? La respuesta es que no lo sabe, sino que **el código que invoca** a esta función debe indicarle adonde continuar la ejecución luego de finalizar la llamada. Esta **dirección** donde debe continuar se conoce como dirección de retorno.

Para realizar llamadas a función, la arquitectura x86-64 provee dos instrucciones:

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.

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
```

aunque `ret` **no modifica** ningún registro (más que el `%rip`). Aquí el asterisco es necesario por la sintaxis.

Cuando las funciones son "llamadas" dentro de un programa se reconocen **dos** actores en cuanto a responsabilidades:

El llamante (*caller*) es la parte de código que invoca a la función en cuestión. El *caller* quiere computar el valor de la función para ciertos valores de argumentos y luego seguir computando con el resultado obtenido.

El llamado (*callee*) es la parte de código que **implementa** la función. Éste debe computar el resultado (valor de retorno) de la función a partir de los argumentos recibidos por el llamante.

5.1. Convención de llamada

Se conoce como convención de llamada al acuerdo previo que tienen estos dos actores (llamante y llamado) sobre cómo invocar funciones, obtener sus resultados y sobre el estado de la máquina previa y posteriormente a la llamada. En lo especifico, una convención de llamada describe a nivel de ensamblador:

- Dónde deben ir los argumentos al invocar a una función.
- Dónde quedará el resultado obtenido.
- Qué registros mantendrán su valor luego de la llamada.

La convención de llamada para x86-64 es la siguiente[5]:

- Los seis primeros argumentos a la función son pasados por registro en el siguiente orden: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` (cuando son valores enteros o direcciones de memoria).
- Si son valores de punto flotantes pueden utilizarse hasta 8 de los registros `xmm`: `%xmm0`, `%xmm1`, `%xmm2`, `%xmm3`, `%xmm4`, `%xmm5`,`%xmm6` y `%xmm7`.
- 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 éstos son pasados por pila a la función.
- El resultado de la función (si lo hubiera) es devuelto en el registro `%rax` si es entero y sencillo o en el registro `xmm0` si es flotante.
- El llamado **se compromete** a preservar el valor de los registros `%rbx`, `%rbp`, `%rsp`, y `%r10` 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 de 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*.
- 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).
- Como `%rbp` y `%rsp` son preservados durante una llamada a función, el estado de la pila de llamante se mantiene.

Respecto a este último punto (la preservación de la pila), es muy común que cada función demarque el comienzo de **su porción** de pila utilizando el `%rbp`. Como este registro es *calle saved* debe ser preservado por el llamado. Por esta razón es muy común ver porciones llamadas prólogo y epilogo en una función como:

[5]En la Fig. 1 se puede observar el rol de los registros en la llamada a función.

```
#prologo
pushq %rbp      # Guardar el valor del rbp del llamante
movq %rsp, %rbp # La pila para esta función comienza en el tope (vacia)
...
...
...
#epilogo
movq %rbp, %rsp # El tope anterior de la pila es el inicio de la pila actual
popq %rbp       # Retaurar el rbp del llamante
```

Veamos cómo llamaríamos a la función `sum` antes vista con los valores 40 y 45:

```
...
movq $40, %rdi    # el valor del primer argumento es 40 y va en registro rdi
movq $45, %rsi    # el valor del segund argumento es 45 y va en registro rsi
call sum          # guarda la direcc de retorno en pila y salta a sum
movq %rax, i      # aquí %rax contiene el resultado (85)
...
```

Veamos ahora una posible implementación de `sum`:

```
.global sum       # la etiqueta sum debe ser global
sum:
     # Prologo
     pushq %rbp
     movq  %rsp, %rbp

     movq %rdi, %rax # copio el valor del primer arg en %rax
     addq %rsi, %rax # y le sumo el segundo argumento
                     # aqui el resultado YA esta en rax

     # Epilogo
     movq %rbp, %rsp
     popq %rbp

     ret            # Tomara el valor de retorno de la pila y saltara a el
```

6. Compilando código ensamblador con GNU as

Un programador puede escribir todo su programa en ensamblador. El único requerimiento es que el código defina una etiqueta global dentro del segmento de código llamada `main`. Una vez escrito el código, el programa puede ser compilado utilizando `gcc`:

```
#gcc  sum.s
```

Luego podemos ejecutar mediante:

```
./a.out
```

También podemos usar la opción `-o`:

```
#gcc -o sum sum.s
```

y luego ejecutar mediante:

```
./sum
```

Sin embargo, escribir todo el programa en ensamblador no es la mejor opción. Es mejor escribir solo la parte que necesariamente debe ser escrita en ensamblador (con fines de optimizar, acceder al hardware, etc). Por ello, podemos mezclar código C con ensamblador siempre y cuando el ensamblador respete la convención de llamada vista en la Sección 5. Vemos un ejemplo.

Ejemplo

```
// 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*

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

Luego podemos compilar todo junto:

```
#gcc main.c suma.s
```

Luego podemos ejecutar mediante:

```
./a.out
```

y obtendremos el resultado:

```
15.140000
```

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

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.