Ensamblador y Arquitectura x86_64

Federico Bergero
fbergero@fceia.unr.edu.ar

Arquitectura del Computador 2014*
Departamento de Ciencias de la Computación
FCEIA-UNR

*Actualizado Setiembre 2015 (D. Feroldi)

Contents

1 La arquitectura x86_64

Nota Estas notas de clase reseñan algunos detalles 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 la clase teórica.

La arquitectura x86_64 es una extensión de la arquitectura Intel i386. Posee registros de 64 bits (aunque soporta también operaciones con valores de 256, 128, 32, 16, y 8 bits), buses de datos y direcciones de 64 bits por lo cual las direcciones de memoria (punteros) son también valores de 64 bits.

Esta arquitectura posee 16 registros de 64 bits de propósito general y además (dependiendo de la versión) registros adicionales para control, punto flotante, etc. Dentro de ellos hay algunos de uso especial como `rsp` y `rip` que son utilizados para manipular la pila (que veremos en la sección 1.3.5) y apuntar a la próxima instrucción respectivamente otros son de propósito general.

La mayoría de los registros de 64 bits son divididos en sub-registros de 32, 16 y 8 bits. Así el registro `rax` de 64 bits contiene en sus 32 bits más bajos a `eax` (e por extended) , en sus 16 bits más bajos a `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 figura 1 vemos (casi) todos los registros del x86_64 con sus sub-registros y su uso durante una llamada a función.

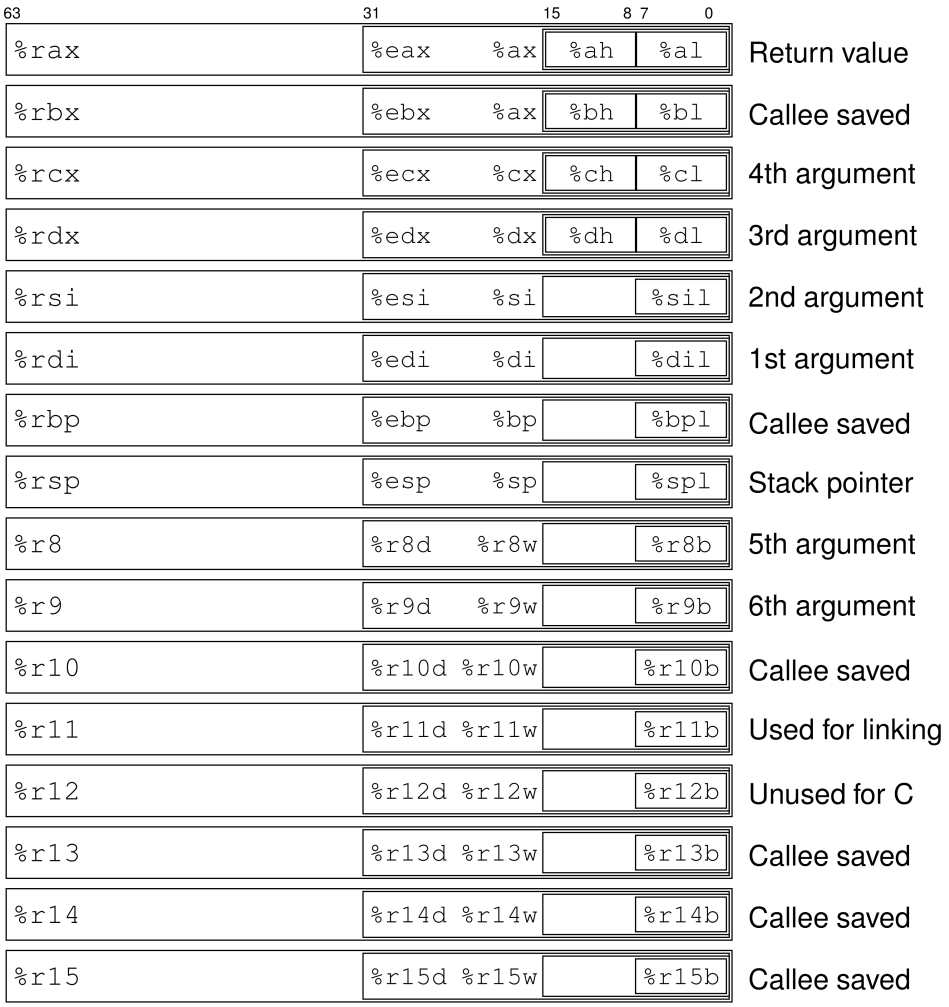

Figure 1: Registros del x86_64 y sus sub-divisiones

1.1 Registros Especiales

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

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.

1.1.1 Registro de `rflags`

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

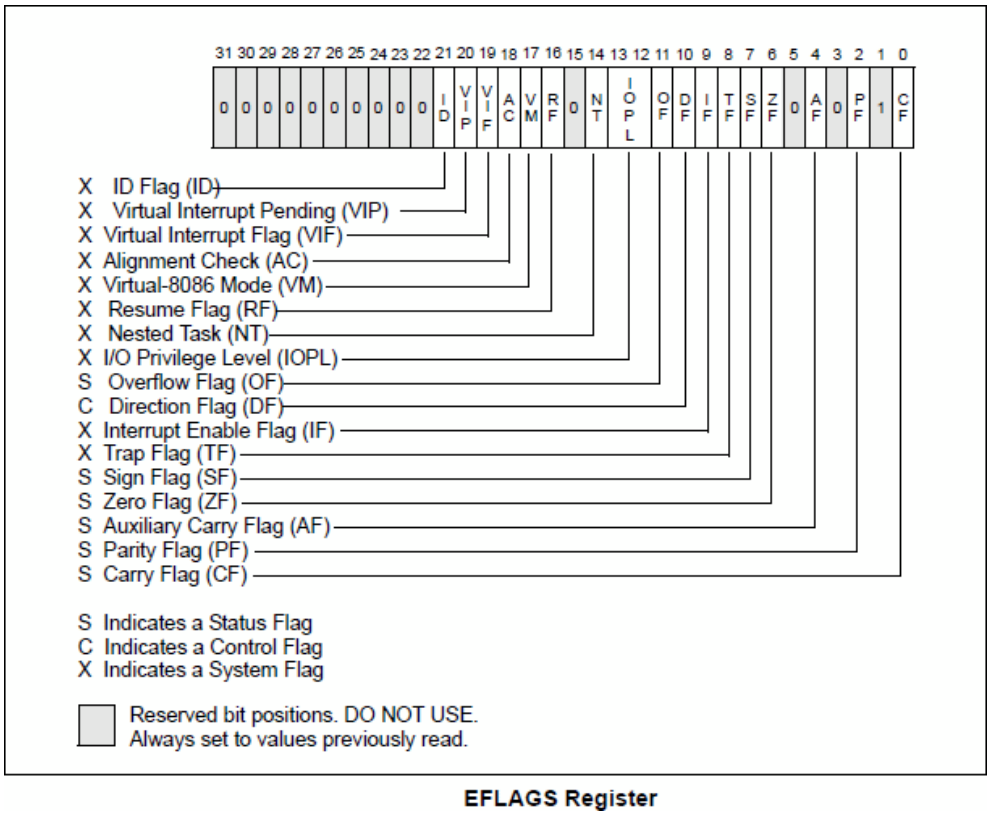

Figure 2: Registro EFLAGS

En la figura 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 alguno 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).

Existen instrucciones especiales. Entre ellas distinguimos varias clases:

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

Prender un bit `setc` (set carry), `std` (set direction), `setz` (set zero), `seto` (set overflow). Dentro de esta clase existen muchas más (todas las instrucciones que comienzan con `set`).

Sumar añadiendo el carry `adc` toma dos operandos, los suma junto con el bit de carry y lo guarda en el destino.

Acceder a todo el registro `lahf` y `sahf` copian ciertos bits del registro ah hacia el flags y viceversa, `popfq` y `pushfq` guardan o traen en/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.

1.2 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 50 se introdujo el lenguaje ensamblador. Éste 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
```

Del fragmento 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.

1.3 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 algunos 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 de CPU 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, así 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. Los sufijos posibles son `b` (por byte), `s` o `w` (por short o word , 2 bytes), `l` (por long, 4 bytes), `q` (por quad, 8 bytes), `t` (por ten, 10 bytes). 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:

  `oper 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`).
- Para de-referenciar un valor se utilizan los paréntesis, por ejemplo `(%rax)` refiere a lo **apuntado** por `rax`. Esta notación permite también la forma:
  - `K(%reg)` refiere al valor apuntado por reg más(menos) un corrimiento de K. 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). Ejemplo `(%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)
    ```

    .

1.3.1 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 en la dirección (0x0B) `a+1`, y así. Esto se conoce como convención Big-Endian y puede verse en la figura 3.

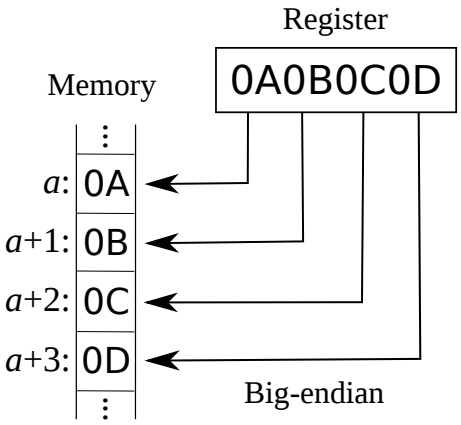

Figure 3: Convención Big-Endian (Fuente Wikipedia)

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 figura 4 muestra la convención Little-Endian.

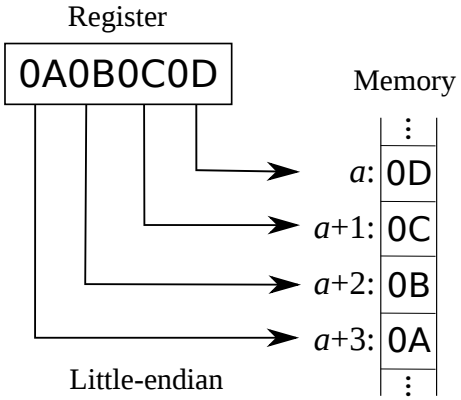

Figure 4: Convención Little-Endian (Fuente Wikipedia)

1.3.2 Directivas al Ensamblador as

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. Entre éstas 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 caracter 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. 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`.

1.3.3 Etiquetas

Las etiquetas son una parte fundamental del lenguaje ensamblador. Permiten al programador dar un nombre a una porción de código o datos. 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 luego sino que una etiqueta se convierte en una **dirección de memoria**.

En ensamblador una etiqueta es un nombre seguido de ":".

Veamos algunos ejemplos:

```
.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. Luego en `f` un valor de punto flotante inicializado en 3.14.

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

1.3.4 Accediendo a la memoria

Supongamos que en el código antes visto queremos incrementar a `i`, esto podemos hacerlo simplemente escribiendo:

```
incq i
```

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

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

```
addq i, %rax
```

Notar que `addq $i, %rax` sumará una constante 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 y 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.

Veamos 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
   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 ningún caracter de esa cadena será copiado a `rax`. De hecho esa instrucción no accede a la memoria.

La arquitectura x86_64 ofrece una instrucción similar al operador de referencia de C. Esta instrucción es `lea` (por "load effective address"). Así la segunda instrucción es equivalente a

```
leaq str, %rax
```

.

1.3.5 La pila

Una pila es una estructura de datos que permite almacenar información. Su funcionamiento puede verse 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 Last-In First-Out (el último en entrar es el primero en salir).

En su implementación informática la pila cuenta con dos operaciones:

push o apilar Inserta un elemento en la pila.

pop o desapilar Quita de la pila el último elemento insertado.

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 4, 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 figura 5 vemos un posible estado de la pila con varios marcos de activación (sólo **uno** está activo en un momento dado, el de la función que se está ejecutando).

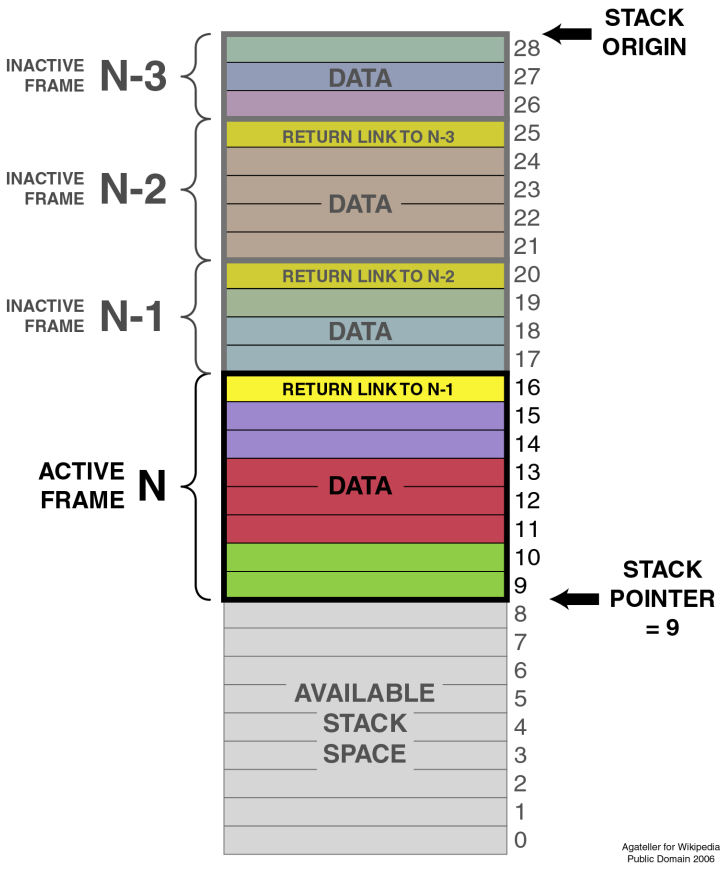

Figure 5: Estado de la pila. Fuente Wikipedia

De la figura 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.

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

Aunque ambos registros tienen este uso particular puede ser manipulados por las instrucciones habituales (`add`, `mov`, etc). La arquitectura 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)
```

Este comportamiento puede verse en la figura 6.

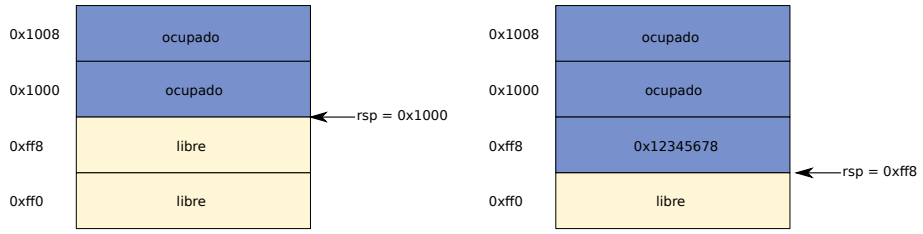

Figure 6: Diagrama de un puhsq

popq 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
```

Este comportamiento puede verse en la figura 7.

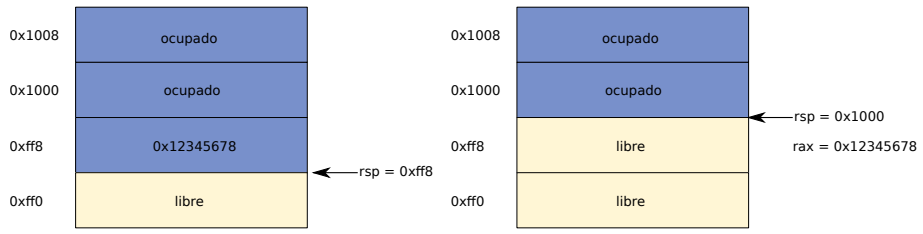

Figure 7: Diagrama de un popq

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.

2 Instrucciones

Como vimos previamente, las instrucciones de ensamblador en la arquitectura x86_64 son compuestas por una operación (ej. suma, resta, comparación) 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 y `inc` incrementa en UNO el valor de su operando, este uno está implícito.

2.1 Operaciones

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

Operaciones aritméticas `add`, `sub`, `inc`, `dec`, `idiv` y `div` (división con y sin signo), `imul` y `mul` (multiplicación con y sin signo).

Operaciones lógicas `and`, `or`, `not`, `xor`.

Operaciones de bits `shl` y `shr` (logical shift left y right), `rol` y `ror` (rotate left y right), `sal` y `sar` (shift aritmético a izquierda y derecha).

En general estas instrucciones realizan una operación binaria entre sus operandos, guardando el resultado en el operando destino. En la Fig. 8 se muestran las operaciones para aritmética y lógica con enteros de 64 bits. Damos algunos ejemplos de uso de estas instrucciones (con sus correspondiente en C en lo posible):

```
addq %rax, %rcx     # rcx+=rax
addq $10, %rcx      # rcx+=10
addq 10, %rcx       # rcx+=*(10) (notar que en la instrucción
# el 10 refiere a la dirección de memoria)
notl %eax           # eax = ~eax
xorb %al, %ch       # ch ^= al
shll $5, %eax       # eax <<= 5
addl (%rax), %ecx   # suma a ecx lo apuntado por rax
subl %ecx, 8(%rbp)  # resta ecx a (lo apuntado por rbp + 8)
xorq %rax, %rax     # rax^=rax (equivalente a rax=0)
```

Por otra parte, en la Fig. 9 se muestran operaciones especiales para generar el producto de 128 bits al operar con números de 64 bits, como así también operaciones para realizar la división de números de 64 bits. Las dos operaciones para dividir, `idivq` y `divq`, operan con `%rdx:%rax` como el dividendo de 128 bits y el operando fuente como el divisor de 64 bits. Luego, guardan el cociente en el registro `%rax` y el resto en el registro `%rdx`. La preparación del dividendo depende de si se realiza la operación sin signo (`divq`) o con signo (`idivq`). En el primer caso, el registro `%rdx` se deja en cero, mientras que en el segundo caso la instrucción `cqto` se usa para realizar la extensión del signo, copiando el bit de signo de `%rax` en cada bit de `%rdx`.

Notar que el sufijo de la instrucción debe estar acorde a sus operandos. Notar también que la arquitectura prohíbe operaciones con dos operandos en memoria debido al costo que implicaría dicha instrucción.

Ejemplos:

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
`incq`	D	$D \leftarrow D$ + 1	Increment
`decq`	D	$D \leftarrow D$ - 1	Decrement
`negq`	D	$D \leftarrow$ -D	Negate
`notq`	D	$D \leftarrow$ ~D	Complement
`addq`	S, D	$D \leftarrow D$ + S	Add
`subq`	S, D	$D \leftarrow D$ - S	Subtract
`imulq`	S, D	$D \leftarrow D$ * S	Multiply
`xorq`	S, D	$D \leftarrow D$ ^ S	Exclusive-or
`orq`	S, D	$D \leftarrow D$ | S	Or
`andq`	S, D	$D \leftarrow D$ & S	And
`salq`	k, D	$D \leftarrow D$ << k	Left shift
`shlq`	k, D	$D \leftarrow D$ << k	Left shift (same as `salq`)
`sarq`	k, D	$D \leftarrow D$ >> k	Arithmetic right shift
`shrq`	k, D	$D \leftarrow D$ >> k	Logical right shift

Figure 8: Operaciones para aritmética y lógica con enteros de 64 bits.

Instruction		Effect	Description
`imulq`	S	R[%rdx]:R[%rax] $\leftarrow S \times$ R[%rax]	Signed full multiply
`mulq`	S	R[%rdx]:R[%rax] $\leftarrow S \times$ R[%rax]	Unsigned full multiply
`cltq`		R[%rax] $\leftarrow$ SignExtend(R[%eax])	Convert %eax to quad word
`cqto`		R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
`idivq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Signed divide
`divq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Unsigned divide

Figure 9: Operaciones especiales para aritmética.

```
movq $10, %rax      # rax=10
movq $25, %rdx      # rdx=25
mulq %rdx           # rdx:rax=250
...
movb $0xff, %dl     # dl=-1
movb $0xbe, %al     # al=-66
imulb %dl           # dl:al=66
...
movq $0, %rdx       #rdx=0
movq $1234, %rax    #rax=1234
movq $-10, %rbx     #rbx=-10
idivq %rbx          #rax=-123 y rdx=4
```

2.2 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 ofrece varias

instrucciones para hacer copias de datos siendo la más importante `mov`. Ésta 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
```

Damos algunos ejemplos de uso de `mov` (con sus correspondiente 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
                       # comenzando en la dirección rbp+16
```

2.3 Comparaciones, Saltos y Estructuras de Control

2.3.1 Saltos

El 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 1.3.3) Veamos un ejemplo:

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

En el fragmento de código anterior la instrucción `movq $1, %rax` **nunca** es ejecutada ya que el `jmp` hace que el procesador salte a la instrucción en la dirección `cont`. Notar aquí que aunque `cont` es una constante (la dirección de memoria donde está la instrucción `movq 2, %rax`) ésta no va prefijada por `$`.

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 1.1.1 el procesador mantiene en el registro `rflags` el estado de la última operación realizada. Luego, los saltos condicionales de x86_64 hacen

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

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`. 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 éstos son 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 figura 10.

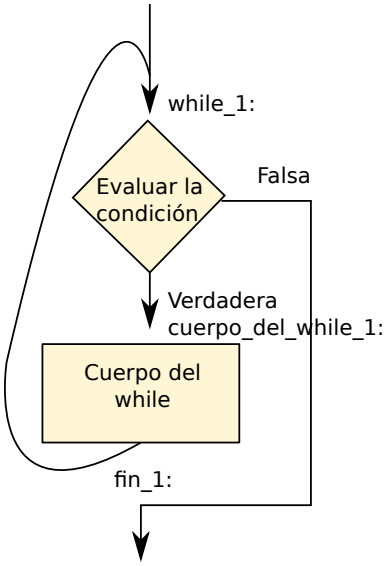

Figure 10: Estructura de While

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` 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).

[3] Aunque su nombre sugieren que son índices, estos registros son apuntadores.

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

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 `ax` por el sufijo de word y que el incremento es en 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) es equivale a

```
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).

Una caso típico de uso 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

Es lógico como vimos en los ejemplos anteriores que una instrucción de cadena se repita muchas veces, 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 re-escrito 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, `repe`, `repne` que 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 de/incrementados y `rcx` decrementado pero los bits del registro `rflags` quedan intactos dejando allí el valor de la última comparación. Así podemos realizar un salto condicional para ver si la última comparación dio igual o distinto.

2.5 Conversiones

La arquitectura x86_64 ofrece instrucciones para convertir entre enteros de distintos tamaño.

Conversiones entre enteros:

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`).

cdq Extiende (con signo) `eax` a `edx:eax` (los 32 bits altos a `edx` y los 32 bits mas bajo a `eax`).

movz Copia un valor de extiendo con *cero* si es necesario. 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 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 llamados `xmm0-xmm15` y 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).

3.1 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 figura 3.1 se recopilan las instrucciones de conversión.

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

Figure 11: Instrucciones de copia y conversiones. Fuente x86-64 Machine-Level Programming

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
```

3.2 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 figura 12 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

Figure 12: Instrucciones de copia y conversiones. Fuente x86-64 Machine-Level Programming

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 4 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
  # segun IEEE 754 de 1.8
  movabsq $4610785298501913805, %rax
  movq %rax, -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. Éstas comparan dos valores (haciendo una resta virtual) y prenden las banderas correspondientes en el registro de `rflags`. Las instrucciones son

ucomiss para comparación de precisión simple.

ucomisd para comparación de precisión doble.

3.3 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 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 y 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 figura 13 se ve los distintos valores que puede contener un registro `xmm`.

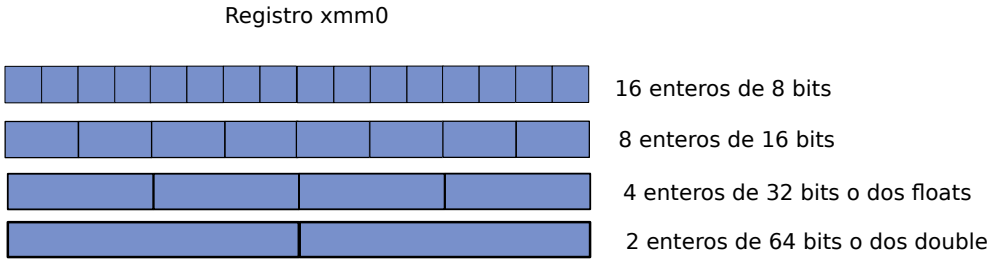

Figure 13: El contenido de un registro `xmm`

Veamos un ejemplo. 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];
}
```

```
 .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 figura 14 grafica esta instrucción.

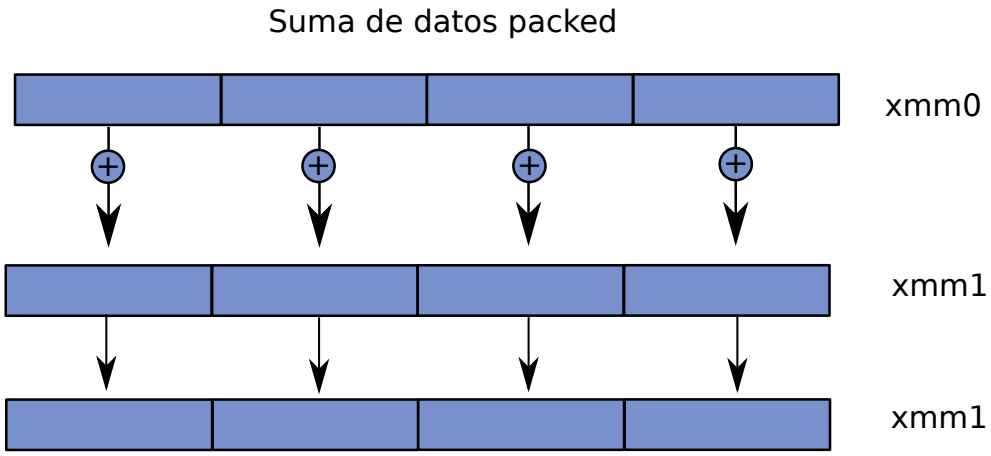

Figure 14: Instrucción `addps`

Hay varios tipos de instrucciones SSE:

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

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 [6] o [7]. Por otra parte, en el 2011 se introdujo una nueva tecnología de instrucciones SIMD llamadas AVX, pero estas no serán vistas en este apunte.

4 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);
```

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

Table 1: Algunas intrucciones *SSE*.

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. Éste 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, a partir de los argumentos recibidos por el llamante, computar el resultado (valor de retorno) de la función.

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 entonces (ver Figura 1):

- 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) y pueden utilizarse hasta 8 de los registros: `%xmm0`, `%xmm1`, `%xmm2`, `%xmm3`, `%xmm4`, `%xmm5`,`%xmm6` y `%xmm7` si son valores de punto flotantes.

  Cuando la función toma como argumento una mezcla de valores enteros y flotantes `rdi` será el primer valor entero, `xmm0` el primer valor flotante, etc. 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).
- 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:

```
#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 de 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
```

5 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`.

En el caso general escribir todo el programa en ensamblador no es la mejor opción. Es mejor escribir sólo 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 4.

Vemos un ejemplo:

```
// Este archivo es main.c
double sum(double a, double b);
int main() {
 double d=fun(12,3.14);
}
```

donde la implementación de `sum` en ensamblador sería

```
// este archivo es sum.s
 .global sum
 sum:
     # 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, ejecutando:

```
#gcc main.c test.s
```

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

References

[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] *x86-64 Machine-Level Programming*, Randal E. Bryant - David R. O'Hallaron, 2005.

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

[7] *x86 Assembly Language ReferenceManual*, Oracle, 2012.