Lenguaje Ensamblador y Arquitectura x86-64

Federico Bergero
Diego Feroldi

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

[*]Actualizado 21 de octubre de 2022 (D. Feroldi, feroldi@fceia.unr.edu.ar)

Índice

1. La arquitectura x86-64 **1**
- 1.1. Lenguaje de máquina . . . 1
- 1.2. Lenguaje Ensamblador de x86-64 . . . 2
- 1.3. Registros de propósito general . . . 4
- 1.4. Registros especiales . . . 5
  - 1.4.1. Puntero de pila . . . 5
  - 1.4.2. Puntero de instrucciones . . . 6
  - 1.4.3. Registros de segmentos . . . 6
  - 1.4.4. Registro de banderas (`rflags`) . . . 6
- 1.5. Registros SSE . . . 8
- 1.6. Operandos inmediatos . . . 8

2. Instrucciones **8**
- 2.1. Instrucciones de transferencia de datos . . . 9
  - 2.1.1. Instrucción MOV . . . 9
  - 2.1.2. Instrucción PUSH . . . 10
  - 2.1.3. Instrucción POP . . . 10
  - 2.1.4. Instrucción XCHG . . . 11
- 2.2. Instrucciones aritméticas . . . 11
  - 2.2.1. Instrucción ADD . . . 11
  - 2.2.2. Instrucción ADC . . . 12
  - 2.2.3. Instrucción SUB . . . 12
  - 2.2.4. Instrucción SBB . . . 12
  - 2.2.5. Instrucción INC . . . 13
  - 2.2.6. Instrucción DEC . . . 13
  - 2.2.7. Instrucción IMUL . . . 13
  - 2.2.8. Instrucción MUL . . . 14
  - 2.2.9. Instrucción IDIV . . . 15
  - 2.2.10. Instrucción DIV . . . 15
  - 2.2.11. Instrucción NEG . . . 16
- 2.3. Instrucciones de comparación . . . 16
  - 2.3.1. Instrucción CMP . . . 16
  - 2.3.2. Instrucción TEST . . . 16
- 2.4. Instrucciones lógicas . . . 17
  - 2.4.1. Instrucción AND . . . 17
  - 2.4.2. Instrucción OR . . . 17
  - 2.4.3. Instrucción XOR . . . 17
  - 2.4.4. Instrucción NOT . . . 17
- 2.5. Instrucciones rotación y desplazamiento . . . 18
  - 2.5.1. Instrucción SAL/SHL . . . 18
  - 2.5.2. Instrucción SAR . . . 18
  - 2.5.3. Instrucción SHR . . . 18
  - 2.5.4. Instrucción ROL . . . 19
  - 2.5.5. Instrucción ROR . . . 19
  - 2.5.6. Instrucción RCL . . . 19
  - 2.5.7. Instrucción RCR . . . 19

2.6. Instrucciones para saltos incondicionales . . . 20
2.6.1. Instrucción JMP . . . 20
2.7. Instrucciones para saltos condicionales . . . 20
2.8. Otras instrucciones de ruptura de secuencia . . . 22
2.8.1. Instrucción LOOP . . . 22
2.8.2. Instrucción CALL . . . 22
2.8.3. Instrucción RET . . . 23
2.9. Instrucciones para el registro de banderas . . . 23
2.10. Instrucciones de entrada/salida . . . 23
2.11. Instrucciones de conversión . . . 23
2.11.1. Instrucciones CXX/CXXE . . . 23
2.11.2. Instrucciones MOVSXX . . . 24
2.11.3. Instrucciones MOVZXX . . . 25

3. Comparaciones, Saltos y Estructuras de Control 25
3.1. Saltos incondicionales . . . 25
3.2. Saltos condicionales . . . 26
3.3. Estructuras de Control . . . 27
3.4. Iteraciones . . . 29

4. Manejo de Arreglos y Cadenas 30
4.1. Copia y manipulación de datos . . . 31
4.1.1. Instrucción LODS . . . 31
4.1.2. Instrucción STOS . . . 31
4.1.3. Instucción MOVS . . . 31
4.2. Búsquedas y Comparaciones . . . 33
4.2.1. Instrucción SCAS . . . 33
4.2.2. Instrucción CMPS . . . 33
4.3. Iteraciones con instrucciones de cadena . . . 34

5. Acceso a datos en memoria 35
5.1. Modelo de memoria de un proceso en Linux . . . 35
5.2. Directivas al Ensamblador . . . 36
5.3. Etiquetas . . . 39
5.4. Definir una etiqueta global . . . 39
5.5. Endianness . . . 40
5.6. Definición de variables . . . 40
5.7. Modos de direccionamiento . . . 43
5.7.1. Modo de direccionamiento inmediato . . . 43
5.7.2. Modo de direccionamiento directo a registro . . . 44
5.7.3. Modo de direccionamiento directo a memoria . . . 44
5.7.4. Modo de direccionamiento indirecto con registro . . . 45
5.7.5. Modo de direccionamiento indexado . . . 45
5.7.6. Modo de direccionamiento relativo . . . 46
5.8. Instrucción LEA . . . 46
5.9. Gestión de la pila . . . 48

6. Aritmética de Punto Flotante **51**
6.1. Copias y conversiones 51
6.2. Operaciones de punto flotante 52
6.3. Instrucciones SIMD 54

7. Funciones y Convención de Llamada **56**
7.1. Funciones 56
7.2. Convención de llamada 58

A. Compilando código ensamblador con GNU as **60**

B. Depurando el código con GDB **61**

Notas generales:

- Este apunte de clases reseña las principales características de la arquitectura x86-64 y de su lenguaje ensamblador. Durante casi todo el curso usaremos principalmente x86-64 sin importar si el fabricante es Intel o AMD.
- Este apunte 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. Para una información más detallada consultar las referencias. En particular, consultar [14] para una información más detallada sobre las instrucciones.
- Para poder compilar y depurar los ejemplos que se muestran en el apunte, ver los Apéndices A y B.

1. La arquitectura x86-64

La arquitectura x86-64 es una ampliación de la arquitectura x86, la cual 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. 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. 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.

Ejemplo

El fragmento de código de una función para sumar dos enteros que se encuentran guardados en registros se escribiría en ensamblador x86-64 como:

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

En la columna de la izquierda vemos las direcciones de memoria, en la columna central el código en lenguaje de máquina y en la columna derecha su equivalente en lenguaje ensamblador.

En el fragmento de código anterior se ve una sintaxis de operación seguida de argumentos donde las operaciones, llamadas instrucciones, tienen un nombre representativo (`movq` por "mover", `addq` por "sumar", etc.). Si bien todavía no sabemos cómo usar estas intrucciones, ya podemos ver que el el lenguaje ensamblador es mucho más comprensible que el el lenguaje de máquina. Veremos más adelante qué significa cada instrucción de ensamblador y sus formas de uso.

1.2. Lenguaje Ensamblador de x86-64

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

- En general, las instrucciones se escriben como:

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

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

Observación

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

`movq %rax, %rbx`

representa `rax` $\rightarrow$ `rbx`*. Es decir, copia el valor de* `rax` *a* `rbx`*. Por lo tanto,* `rbx=rax` *luego de ejecutar la instrucción. Es importante notar el valor del registro* `rax` *sigue siendo el mismo. En realidad, más que un movimiento es una copia de datos.*

- El nombre de los registros comienza con `%`. Por ejemplo, el registro `rax` se escribe como `%rax`.
- Los comentarios de línea comienzan con `#` (a partir de `#` comienza un comentario hasta el fin de línea).

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

  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.
- Para de-referenciar un valor se utilizan los paréntesis, por ejemplo `(%rax)` refiere a lo **apuntado** por `rax`.

Ejemplo

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

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

Esta notación permite también formas más complejas:

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

Ejemplos

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

Vemos que algunos de los registros pueden ser opcionales.

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

Ejemplo

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

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

Estos modos de direccionamiento los veremos con mayor detalle en la Sección 5.7.

Observación

En este apunte utilizaremos la sintaxis de AT&T de lenguaje ensamblador ya que es la utilizada por el GNU Assembler (GAS). Las principales diferencias entre ambas son las siguientes:

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

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

1.3. Registros de propósito general

La arquitectura x86-64 posee 16 registros de propósito general (cada uno de 64 bit): `rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp` y `r8-r15`[1]. Los 8 primeros registros se de-

[1]Si bien el registro `rsp` está dentro del grupo de registros de propósito general, veremos en la Sección 5.9 que su uso es bastante particular.

nominan de manera parecida a los 8 registros de propósito general de 32 bits disponibles en la arquitectura IA-32 (`eax, ebx, ecx, edx, esi, edi, ebp y esp`) dado que son extensiones de los mismos. En cambio, los registros `r8` al `r9` son registros nuevos. Además, dependiendo de la versión cuenta con registros adicionales para control, punto flotante, etc. Dentro del conjunto de registros disponibles hay algunos de uso especial como el `rsp` y el `rip` que son utilizados para manipular la pila (como veremos en la Sección 5.9) y apuntar a la próxima instrucción, respectivamente.

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` (la `e` es 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-r15`) 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 de propósito general del x86-64 con sus subregistros y su uso durante una llamada a función. Asimismo, vemos su rol en la convención de llamada (*caller saved* o *callee saved*) y si son preservado. Esto será visto en detalle en la Sección 7.

					Uso	Convención	Preservado?
rax	eax	ax	ah	al	Valor de retorno	Caller saved	No
rbx	ebx	bx	bh	bl		Callee saved	Sí
rcx	ecx	cx	ch	cl	4º argumento	Caller saved	No
rdx	edx	dx	dh	dl	3º argumento	Caller saved	No
rsi	esi	si		sil	2º argumento	Caller saved	No
rdi	edi	di		dil	1º argumento	Caller saved	No
rbp	ebp	bp		bpl	Puntero base de pila	Callee saved	Sí
rsp	esp	sp		spl	Puntero tope de pila	Callee saved	Sí
r8	r8d	r8w		r8b	5º argumento	Caller saved	No
r9	r9d	r9w		r9b	6º argumento	Caller saved	No
r10	r10d	r10w		r10b	Temporal	Caller saved	No
r11	r11d	r11w		r11b	Temporal	Caller saved	No
r12	r12d	r12w		r12b		Callee saved	Sí
r13	r13d	r13w		r13b		Callee saved	Sí
r14	r14d	r14w		r14b		Callee saved	Sí
r15	r15d	r15w		r15b		Callee saved	Sí
63	31		15	7 0			

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

1.4. Registros especiales

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

1.4.1. Puntero de pila

Uno de los registros de la CPU, el `rsp`, se usa para señalar el tope actual de la pila. Si bien este registro está en el listado de registros de propósito general, el registro `rsp` no debe utilizarse para datos u otros usos. Veremos en detalle el uso de este registro en la Sección 5.9.

1.4.2. Puntero de instrucciones

El puntero de instrucciones o contador de programa (en inglés *Instruction Pointer* o *Program Counter*) apunta o guarda la dirección de memoria de la próxima instrucción a ejecutar. Su denominación es `rip`.

1.4.3. Registros de segmentos

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

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

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

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

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

Observación

En modo de 64 bits se utiliza un modelo de segmentación plana de la memoria virtual. Es decir, el espacio de memoria virtual de 64 bits se trata como un único espacio de direcciones plano (no segmentado), por lo cual los registros de segmentos no tienen mayor utilidad. El tema ***Segmentación*** *lo veremos por fuera de este apunte cuando veamos* ***Memoria Virtual****.*

1.4.4. Registro de banderas (`rflags`)

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

En la Fig. 2 vemos el registro `eflags` (la versión de 32 bits del `rflags`, es decir, los 32 bits menos significativos). Los marcados con "S" son bits de estado mientras que los marcados con "C" son de control. Describimos brevemente las banderas más utilizadas:

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

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

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

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

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

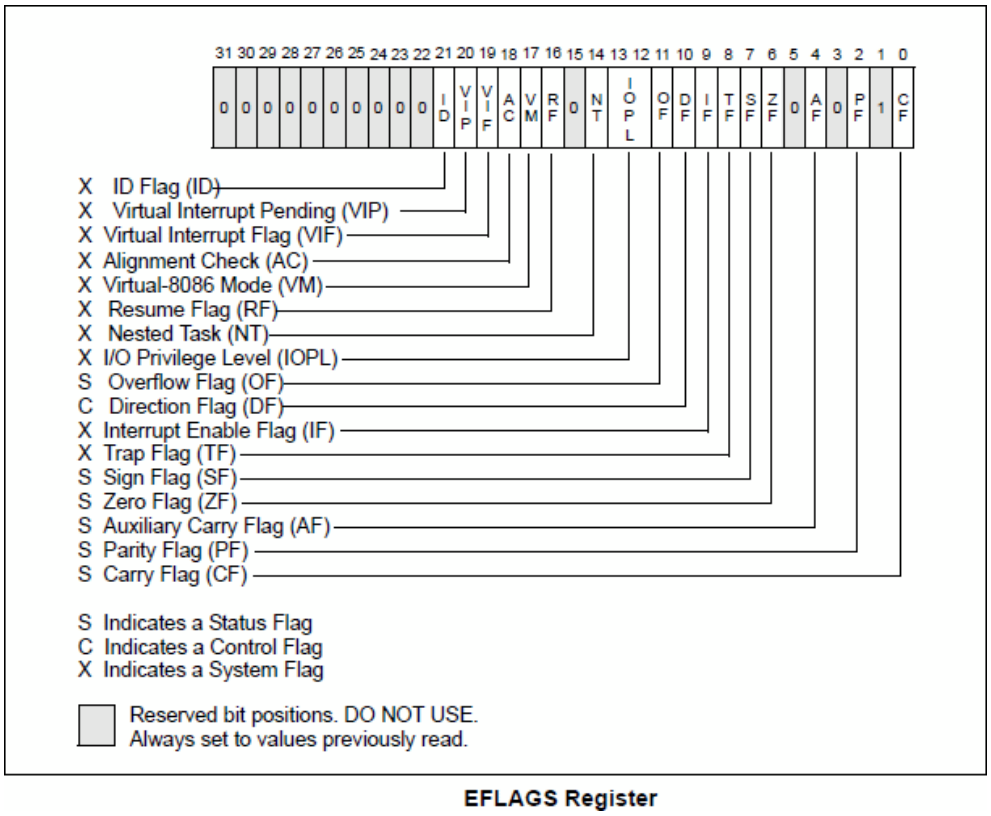


Figura 2: Registro EFLAGS

PF *Parity Flag*: en 1 si la cantidad de bits en 1 en los 8 bits menos significativos del resultado de la operación es par. En las operaciones con números en punto flotante además tiene otra interpretación que veremos en la Sección 6.

El registro `rflags` no es de propósito general por lo cual no puede ser accedido ni modificado por instrucciones regulares (`add`, `mov`, etc) de manera directa. En cambio, sí puede ser modificado de manera indirecta por instrucciones tales como las de comparación, aritméticas, etc. Es decir, las banderas del registro `rflags` se modifican como resultado de ciertas instrucciones, lo cual veremos más adelante que tiene importantes utilidades.

Ejemplo

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

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

1.5. Registros SSE

Adicionalmente, la arquitectura x86-64 proporciona 16 registros de 128 bits (`xmm0-xmm15`), denominados registros SSE ("streaming SIMD extensions"), donde SIMD significa *single instruction, multiple data*. 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 utilidad de estos registros será vista con mayor detalle en la Sección 6.3.

1.6. Operandos inmediatos

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

Ejemplos

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

Observación

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

```
48 c7 c0 44 33 22 11
```

Aquí se ve de manera explícita que el valor inmediato está contenido dentro de la propia instrucción. El motivo por el cual se ve invertido lo veremos en detalle en la Sección 5.5.

2. Instrucciones

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

no toman operandos o sus operandos son implícitos. Por ejemplo, la instrucción `ret` no toma operandos, mientras que `inc` solo toma un operando y lo incrementa en uno (el uno está implícito).

El juego de instrucciones de los procesadores x86-64 es muy amplio y en esta sección veremos las principales instrucciones para operar con valores enteros. Luego, en la Sección 6 se verán la instrucciones para operar con datos en punto flotante. Para una mayor información sobre las instrucciones en x86-64 ver [14].

2.1. Instrucciones de transferencia de datos

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

2.1.1. Instrucción MOV

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

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

donde "S" es el sufijo que indica el tamaño de los operandos (que deben ser del mismo tamaño) según lo visto en la Tabla de la página 3.

Es importante notar que la instrucción `mov` no mueve sino que copia valores. De esta manera, la instrucción `movq %rax, %rbx` hace que el quad-word alojado en `rax` quede alojado en `rbx`. Por lo tanto, luego de ejecutada `rbx=rax` y el valor de `rax` sigue siendo el mismo.

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

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

Ejemplos

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

```
movb $45, a             # Copia el valor 45 en la dirección de memoria
                        # con etiqueta a
```

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

Observación

La relación entre subregistros de diferentes tamaños es un poco confusa:

1. *La carga de un valor en un subregistro de 32 bits establece los 32 bits superiores del registro en cero. Por ejemplo, después de* `movl $-1, %eax`*, el registro* `%rax` *tiene el valor* `0x00000000ffffffff`*.*
2. *La carga de un valor en un subregistro de 16 u 8 bits deja todos los demás bits sin cambios. Por ejemplo, si el valor de* `%rax` *es* `0xffffffffffffffff`*, luego de* `movw $0, %ax` *el registro* `%rax` *tiene el valor* `0xffffffffffff0000`*.*

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

2.1.2. Instrucción PUSH

Esta instrucción mueve el operando de la instrucción al tope de la pila:

```
pushS <operando fuente>
```

Ejemplo

```
pushq %rax
```

Coloca el valor guardado en `rax` *en el "tope" de la pila. Es decir, en la locación de memoria apuntada por el registro* `rsp`*.*

2.1.3. Instrucción POP

Esta instrucción copia el dato que se encuentra en el tope de la pila[2] al operando destino:

```
popS <operando destino>
```

Ejemplo

```
popq %rax
```

Guarda en el registro `rax` *el valor alojado en el "tope" de la pila.*

[2]En la Sección 5.9 veremos en detalle qué es y cómo se utiliza la pila.

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

2.1.4. Instrucción XCHG

Esta instrucción intercambia el contenido de los operandos:

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

Ejemplo

```
movq $34, %rax
movq $0xf3fa, %rax
xchgq %rax, %rbx
```

Luego de ejecutarse, `rax=0xf3fa` *y* `rbx=34`.

2.2. Instrucciones aritméticas

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

2.2.1. Instrucción ADD

Esta instrucción realiza la suma aritmética de los dos operandos:

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

El resultado queda en el operando destino:

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

Ejemplo

```
movb $5, %al
movb $4, %bl
addb %al, %bl
```

Luego de ejecutarse, `al=5` *y* `bl=9`.

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

2.2.2. Instrucción ADC

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

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

Resulta:

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

Ejemplo

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

2.2.3. Instrucción SUB

Esta instrucción realiza la resta aritmética de los dos operandos:

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

Realiza la resta: `operando destino = operando destino - operando fuente`.

Ejemplo

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

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

2.2.4. Instrucción SBB

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

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

Resulta:

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

Ejemplo

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

2.2.5. Instrucción INC

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

```
incS <operando>
```

Ejemplo

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

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

2.2.6. Instrucción DEC

Decrementa el operando en una unidad:

```
decS <operando>
```

Ejemplo

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

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

2.2.7. Instrucción IMUL

Multiplicación entera con signo. La instrucción `imul` tiene tres formatos:

Con un operando: `imulS <operando>`

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

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

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

En este formato el operando destino es multiplicado por el operando fuente. El operando destino debe ser un registro de propósito general, mientras que el operando fuente puede ser un registro de propósito general, un valor inmediato o un valor en memoria. El resultado intermedio (el doble de tamaño que el operando fuente) es truncado y guardado en el operando destino.

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

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

Ejemplos

```
movq $9, %rax
movq $-3, %rbx
imulq %rbx                      # rax=0xfffffffffffffe5 (-27)
                                # rdx=0xffffffffffffffff (-1)
movq $9, %rax
imulq %rbx, %rax                # rax=0xfffffffffffffe5 (-27)
imulq $4, %rax                  # rax=0xffffffffffffff94 (-108)
imulq $2, %rax, %rbx            # rbx=0xffffffffffffff28 (-216)
movq $0x7fffffffffffffe, %rax   # rax=9223372036854775806
imulq $2, %rax                  # rax=0xfffffffffffffffc (-4)
```

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

2.2.8. Instrucción MUL

Multiplicación entera sin signo. Esta instrucción a diferencia de la anterior solo admite el formato con un operando:

`mulS <operando>`

Ejemplo

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

2.2.9. Instrucción IDIV

División entera con signo:

`idivS <operando divisor>`

La instrucción `idiv` en su versión de 64 bits divide el contenido del entero de 128 bits `rdx:rax` (construido interpretando a `rdx` como los ocho bytes más significativos y a `rax` como los ocho bytes menos significativos) por el valor del operando especificado. El resultado del cociente de la división se almacena en `rax`, mientras que el resto se coloca en `rdx`[3].

Ejemplo

```
movq $0xffff, %rax    # rax = 65535
movq $0, %rdx
movq $-1024, %rbx
idivq %rbx             # rax=0xffffffffffffffc1 (-63) y rdx=0x3ff (1023)
```

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

2.2.10. Instrucción DIV

División entera sin signo:

`divS <operando divisor>`

Ejemplo

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

El resultado entero es 63 y el resto es 1023.

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

2.2.11. Instrucción NEG

Negación aritmética en complemento a 2:

```
negS <operando>
```

Ejemplo

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

2.3. Instrucciones de comparación

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

2.3.1. Instrucción CMP

Esta instrucción realiza la comparación de los dos operandos:

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

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

Ejemplo

```
movq $45, %rbx
movq $66, %rax
cmpq %rax, %rbx
cmpq %rbx, %rax
```

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

2.3.2. Instrucción TEST

Comparación lógica de los dos operandos:

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

Hace una operación `and` lógica sin guardar el resultado.

Ejemplo

```
testb %cl, %cl          # ZF=1 si cl=0 y SF=1 si cl<0
```

2.4. Instrucciones lógicas

2.4.1. Instrucción AND

Operación `and` lógica bit a bit.

Ejemplo

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

2.4.2. Instrucción OR

Operación `or` lógica bit a bit.

Ejemplo

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

2.4.3. Instrucción XOR

Operación `xor` lógica bit a bit.

Ejemplo

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

2.4.4. Instrucción NOT

Negación lógica bit a bit.

Ejemplo

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

2.5. Instrucciones rotación y desplazamiento

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

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

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

2.5.1. Instrucción SAL/SHL

Desplazamiento aritmético/lógico a la izquierda.

Ejemplo

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

Las instrucciones `sal` y `shl` producen el mismo resultado.

2.5.2. Instrucción SAR

Desplazamiento aritmético a la derecha.

Ejemplo

```
movb $-4, %al   # al=0xfc (-4)
sarb $2, %al    # al=0xff (-1)
```

2.5.3. Instrucción SHR

Desplazamiento lógico a la derecha.

Ejemplo

```
movb $-4, %al   # al=0xfc (-4)
shrb $2, %al    # al=0x3f (63)
```

Notar la diferencia en el resultado con la instrucción `sar`.

2.5.4. Instrucción ROL

Rotación lógica a la izquierda.

Ejemplo

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

2.5.5. Instrucción ROR

Rotación lógica a la derecha.

Ejemplo

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

2.5.6. Instrucción RCL

Rotación lógica a la izquierda considerando el bit de acarreo.

Ejemplo

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

En realidad el resultado es `0x155` *pero no entra en* `al`.

2.5.7. Instrucción RCR

Rotación lógica a la derecha considerando el bit de acarreo.

Ejemplo

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

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

2.6. Instrucciones para saltos incondicionales

2.6.1. Instrucción JMP

```
jmp etiqueta
```

La instrucción `jmp` salta de manera incondicional a la dirección de memoria indicada en la etiqueta.

Observación

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

Ejemplo

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

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

2.7. Instrucciones para saltos condicionales

Las instrucciones para saltos condicionales tienen la forma:

```
jCC etiqueta
```

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

Ejemplo

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

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

```
        cmpq %rax, %rbx
        je etiqueta
        ....
        ....
etiqueta:
        ....
```

En la instrucción `cmpq %rax, %rbx` *se comparan los operandos y si son iguales* `ZF=1`*. Luego la instrucción* `je` *verifica la bandera* `ZF` *y si la encuentra seteada salta a* `etiqueta`*, salteando las instrucciones posteriores. Si la bandera no esta seteada, entonces sí las ejecuta.*

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

2.8. Otras instrucciones de ruptura de secuencia

2.8.1. Instrucción LOOP

```
loop etiqueta
```

La instrucción `loop` tiene dos efectos:

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

Ejemplo

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

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

Ver en detalle las aplicaciones de instrucción `loop` en la Sección 3.4.

2.8.2. Instrucción CALL

```
call etiqueta
```

Esta instrucción se utiliza para hace una llamada a subrutina. Esta instrucción y la siguiente se ven en detalle en la Sección 7.

2.8.3. Instrucción RET

`ret`

Esta instrucción se utiliza para hacer un retorno de subrutina.

2.9. Instrucciones para el registro de banderas

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

- **Apagar un bit:** `clc` (*clear carry flag*), `cld` (*clear direction flag*).
- **Prender un bit:** `stc` (*set carry flag*), `std` (*set direction flag*), `sti` (*set interruption flag*).
- **Sumar añadiendo el carry:** `adc` toma dos operandos, los suma junto con el bit de carry y lo guarda en el destino.
- **Acceder al registro:** `lahf` 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 en la Sección 3.

2.10. Instrucciones de entrada/salida

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

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

2.11. Instrucciones de conversión

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

2.11.1. Instrucciones CXX/CXXE

Existe un conjunto de instrucciones que doblan el tamaño del registro correspondiente, extendiendo con el signo el valor almacenado. Aquí vemos algunas de las instrucciones disponibles:

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

Observación

Notar que las instrucciones anteriores trabajan con operados implícitos. Notar también que hay instrucciones "sinónimos". Es decir, instrucciones con diferentes nombres pero que hacen la misma operación. Para un listado completo de las instrucciones de conversión consultar [14].

Ejemplo

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

Luego de ejecutarse el código anterior, `ax=0xffde` *y* `dx=0xffff`.

2.11.2. Instrucciones MOVSXX

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

Ejemplos

```
movsbl %bl, %ebx     # convierte un byte a un long
movswl %cx, %ecx     # convierte un word a un long
movswq %ax, %rax     # convierte un word a un quad
```

Por lo tanto si tenemos el siguiente código:

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

Luego de ejecutarse, `rax=0xfffffffffffffd3`.

2.11.3. Instrucciones MOVZXX

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

Ejemplos

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

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

3. Comparaciones, Saltos y Estructuras de Control

3.1. Saltos incondicionales

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 5.3).

Ejemplo

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

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

La instrucción `jmp` permite hacer saltos y es el equivalente a un `goto` de un lenguaje de alto nivel. Pero ¿cómo podemos implementar estructuras de control como bucles y

condicionales con ella? Respuesta: no se puede. Para ello debemos introducir los saltos condicionales.

3.2. Saltos condicionales

Los saltos condicionales 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 1.4.4, 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 dependiendo del valor de determinados bits del registro `rflags` dependiendo de la instrucción utilizada . 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.

Observación

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

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

Siguiendo la lógica de la instrucción `sub`,

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

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

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

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

En la Tabla 1 se mostró un listado completo de instrucciones `jCC` y los valores requeridos en las banderas.

Ejemplo

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

```
        ....
        ....
menor:
        ....
```

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

Observación

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

3.3. Estructuras de Control

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

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

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

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

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

Veamos en el fragmento anterior varias cosas:

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

Vemos ahora cómo traduciríamos el siguiente fragmento:

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

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

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

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

Vemos en el fragmento anterior varias cosas:

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

3.4. Iteraciones

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

Ejemplo

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

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

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

```
while_1:
        cmpq $0, i        # Evaluar la condición
        je fin_1          # Si resulta falsa, el lazo termina

cuerpo_del_while_1:       # Acá irá el cuerpo del while
        ...
        ...
        decq i
        jmp while_1
fin_1:
        ...
        ...
```

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

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.

Ejemplo

Dada la siguiente estructura tipo `for`*:*

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

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

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

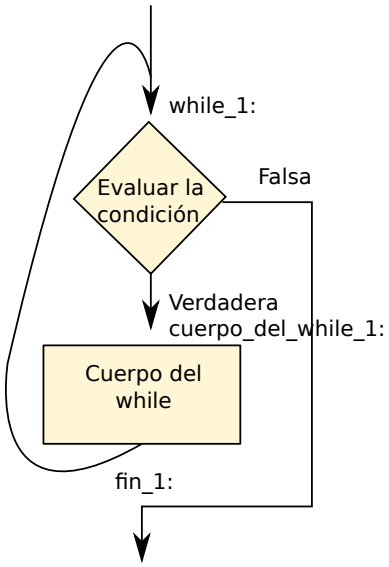


Figura 3: Estructura tipo `while`.

```
                                        # Se inicializa en 100

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

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

4. Manejo de Arreglos y Cadenas

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

Como los arreglos son estructuras de datos muy utilizadas, la arquitectura x86-64 incluye varias instrucciones (llamadas de cadena) para realizar copias, comparaciones, búsquedas, etc. Esta familia de instrucciones hace uso especial de dos registros: `rsi` (*source index*) y `rdi` (*destination index*) [4]. Cuando el procesador ejecuta una instrucción de cadena, éste incrementa/decrementa automáticamente esos registros [5] para apuntar al próximo elemento del arreglo. La cantidad incrementada/decrementada depende del

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

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

tamaño del dato en cuestión. Además, el bit DF (*direction flag*) del registro `rflags` le indica al procesador si debe incrementar o decrementar los registros de índice (se puede apagar con `cld` para que se incrementen o prender con `std` para que se decrementen). A continuación veremos las diferentes instrucciones de manejo de arreglos y cadenas con sus respectivos ejemplos.

4.1. Copia y manipulación de datos

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

4.1.1. Instrucción LODS

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

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

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

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

4.1.2. Instrucción STOS

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

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

4.1.3. Instucción MOVS

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

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

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

Observación

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

Ejemplo

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

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

que puede ser implementado en ensamblador como

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

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

Supongamos que ahora debemos modificar el arreglo como sigue:

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

Esto puede ser escrito utilizando instrucciones de cadena como sigue:

```
 .global f
f:
  # suponemos que rdi tiene el puntero al arreglo "a"
  movq %rdi, %rsi # el origen y el destino son el mismo arreglo
  movq $100, %rcx # iteramos 100 veces
  cld       # iremos incrementando rsi y rdi (DF=0)
```

```
l:
  lodsl     # cargamos en eax el elemento del arreglo (apuntado por rsi)
  incl %eax # lo incrementamos
  stosl     # lo guardamos en el arreglo (apuntado por rdi)
  loop l    # pasamos al siguiente elemento
  ret
```

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

4.2. Búsquedas y Comparaciones

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

4.2.1. Instrucción SCAS

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

4.2.2. Instrucción CMPS

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

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

Ejemplo

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

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

}
```

Esta función puede ser implementada en ensamblador como sigue:

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

4.3. Iteraciones con instrucciones de cadena

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

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

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

```
.global find
find:
  cld                   # iremos incrementando rdi (DF=0)
  movq $100, %rcx       # iteramos 100 veces
```

```
  movl %esi, %eax       # buscamos el 2do argumento
  repne scasl           # repetimos mientra sea distinto o a lo sumo 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.

5. Acceso a datos en memoria

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

5.1. Modelo de memoria de un proceso en Linux

La memoria para un proceso de Linux se divide en 4 regiones (segmentos):

- **Segmento de texto**: En este segmento van todas las instrucciones. El segmento de texto se denomina `.text` en GNU assembler (GAS). El segmento de texto no necesita crecer, por lo que el segmento de datos se puede colocar inmediatamente después.
- **Segmento de datos**: En este segmento están todos los datos estáticos inicializados cuando se inicia el programa. El segmento de datos está dividido en dos partes:
  - `.data` que contiene datos inicializados explícitamente.
  - `.bss` que contiene datos reservados (inicializados a 0 en caso de que no hayan sido inicializados explícitamente). "bss" significa "*Block Started by Symbol*".
- **Segmento heap**: En este segmento están los datos asignados por `malloc` o `new`.
- **Segmento pila**: Este segmento es la pila en tiempo de ejecución[6]. En este segmento se encuentran los siguientes elementos:

[6]El tema *Gestión de la Pila* será visto con mayor detalle en la Sección 5.9.

- direcciones de retorno
- algunos parámetros de la función
- variables locales de funciones
- espacio para variables temporales

En realidad el modelo de memoria de un proceso es más complejo pero el diagrama esquemático mostrado en la Fig. 4 es una buena aproximación. Por ejemplo, `main` no está en realidad en la dirección 0. En realidad el segmento de texto comienza en una dirección un poco superior a `0x400000`. Con respecto al tope superior, también es una aproximación. La pila se asigna a las direcciones más altas de un proceso y en Linux x86-64 es `0x7fffffffffff` o 131 TB[7]. Esta dirección es equivalente a 47 bits con todos los bits en 1.

Tanto el *heap* como la pila necesitan crecer mientras el proceso está en ejecución: el *heap* crece "hacia arriba" (direcciones de memoria mayores) mientras que la pila crece "hacia abajo" (direcciones de memoria menores). Ambos segmentos pueden llegar a encontrarse en el medio y por lo tanto pueden llegar a "explotar". El uso del espacio de *heap* y de pila en el lenguaje ensamblador no implica el uso de un segmento con sus nombres específicos.

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

5.2. Directivas al Ensamblador

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

Dentro de las directivas destacamos las siguientes:

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

Ejemplos

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

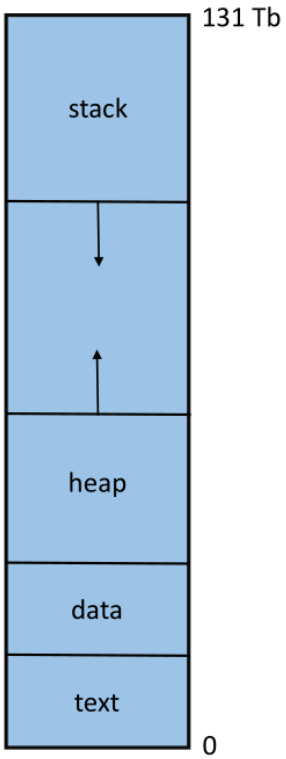


Figura 4: Modelo de memoria de un proceso en Linux [16].

```
.data
(A partir de aquí un segmento con datos inicializados)
.bss
(A partir de aquí un segmento con datos no inicializados)
.text
(A partir de aquí un segmento con código de programa)
```

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

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

Ejemplos

```
.asciz "Hola mundo"
.ascii "a" "b" "c"
```

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

byte Inicializa una lista de bytes.

Ejemplos

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

El primer ejemplo es un arreglo de bytes.

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

Ejemplos

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

El primer ejemplo es un arreglo de doubles.

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

Ejemplos

```
.short 20
.long 50
.quad 0
```

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

Ejemplos

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

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

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

Observación

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

```
.long
```

lo cual ***NO*** *reserva espacio. La versión correcta sería* `.long 0` *o alternativamente* `.space 8`.

5.3. 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**.

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

Ejemplo

```
a: .quad $126
```

Aquí se crea una variable de tipo `quad` *(8 bytes) inicializada en 126 en una dirección de memoria marcada con la etiqueta* `a`.

5.4. Definir una etiqueta global

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

Ejemplos

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

De no especificar esta directiva la etiqueta desaparece luego del proceso de compilación. Las etiquetas globales deben ser utilizadas, por ejemplo, con las etiquetas que

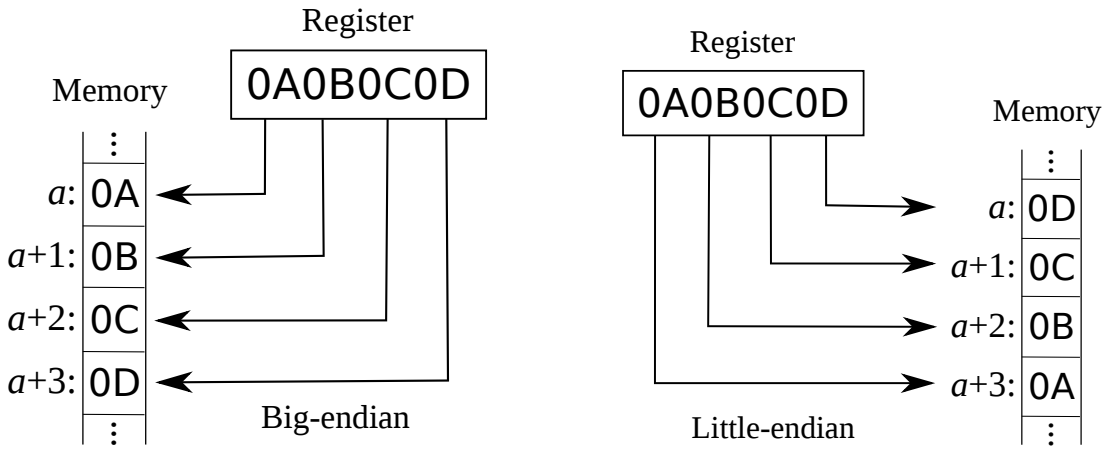
definan funciones que serán llamadas fuera del archivo compilado. Por ejemplo, cuando se enlaza un programa C con uno escrito en ensamblador, las funciones incluidas en ensamblador deben ser definidas como globales (siendo `main` el caso más común). Esto se verá en detalle en la Sección A.

5.5. Endianness

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

Supongamos que tenemos que almacenar el entero 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 byte 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 sucesivamente. Esto se conoce como convención *Big-Endian* como puede verse en la Fig. 5(a).



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

Figura 5: 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 sucesivamente. 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. 5(b) muestra la convención *Little-Endian*.

5.6. Definición de variables

La declaración de variables en un programa en ensamblador se realiza en la sección `.data`. Las variables de esta sección se definen utilizando las directivas vistas en la Sección 5.2. Por ejemplo, `var: .long 0x12345678` es una variable con el nombre `var` de tamaño 4 bytes inicializada con el valor `0x12345678` que comienza en la dirección de memoria cuya etiqueta es `var`. Es importante destacar que en ensamblador hay que estar muy alerta cuando accedemos a las variables que hemos definido previamente. Las variables se guardan en memoria consecutivamente a medida que las declaramos y no existe nada que delimite las unas de las otras. Veamos a continuación un ejemplo ilustrativo.

Ejemplo

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

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

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

La instrucción `movq var1, %rax` *copia 8 a partir de la dirección* `var1`*. Es decir, el procesador tomará como primer byte el valor de* `var1` *y los 7 bytes que están a continuación. Por lo tanto, como los datos se tratan en formato little-endian, en el registro* `rax` *quedará cargado el valor* `0x0001E26C02006100`*. Si este acceso no es el deseado, el compilador no reportará ningún error, ni tampoco se producirá un error durante la ejecución; solo podremos detectar que lo estamos haciendo mal probando el programa y depurando.*

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

Ejemplo

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

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

.bss
a: .quad

.text
.global main
main:
     movq $40, %rax        # rax=40
```

```
    movl i, %ebx          # ebx=0
    movq $-1, a           # a=0xffffffffffffffff (-1)
    movq f, %rdx          # rdx=0x40091eb851eb851f (3.14)
    movl str, %ecx        # ecx=0x616c6f48 ("aloH")
    retq
```

Aquí vemos que la etiqueta `i` *(dentro del segmento de datos* `.data`*) define la posición de memoria donde el ensamblador alojará un entero inicializado en 0 (4 bytes). Luego en* `f` *un valor de punto flotante inicializado en 3.14 (8 bytes). Luego en* `str` *arranca una cadena de caracteres de 11 bytes (el byte final corresponde al cero final). En el segmento* `.bss` *se crea una variable tipo* `quad` *sin inicializar. Finalmente, vemos que dentro del segmento de código se define una etiqueta global llamada* `main`*. Este será el punto de inicio del programa. Luego, a medida que se vayan ejecutando las instrucciones siguientes los registros y locaciones de memoria irán quedando con los valores indicados en los comentarios.*

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

Observación

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

```
incq i
```

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

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

```
addq i, %rax
```

Sin embargo, notar que `addq $i, %rax` *produce un efecto muy diferente. En este caso sumará una constante (la dirección de* `i`*) y no el valor alojado en* `i`*.*

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

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

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

Ejemplo

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

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

.text
.global main
main:
movq str , %rax # Instruccion 1
movq $str, %rax # Instruccion 2
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 y carga. Notemos que en este caso ningún carácter de esa cadena será copiado a* `rax`*. De hecho esa instrucción no accede a la memoria.*

5.7. Modos de direccionamiento

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

5.7.1. Modo de direccionamiento inmediato

En este caso, el operando origen hace referencia a un dato que se encuentra en la propia instrucción. El operando destino puede ser un registro o una locación en memoria. El valor inmediato especificado debe poder ser expresado con 32 bits como máximo. Este valor puede ser una constante o también puede ser el resultado de evaluar una expresión

aritmética formada por valores numéricos y operadores aritméticos. La única excepción es la instrucción `mov` cuando el segundo operando es un registro de 64 bits, en cuyo caso podremos especificar un valor inmediato de 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.*

5.7.2. Modo de direccionamiento directo a registro

En este caso, los operandos hacen referencia a datos que se encuentran almacenados en registros (no hay acceso a memoria). En este modo de direccionamiento podemos especificar cualquier registro de propósito general.

Ejemplo

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

Carga en el registro `rbx` *el valor contenido en* `rax`*.*

5.7.3. Modo de direccionamiento directo a memoria

En este caso, uno de los operandos hace referencia a un dato que se encuentra almacenado en una posición de memoria y el otro hace referencia a un registro. Ambos operandos no pueden ser referencias a memoria. La referencia a memoria puede ser a través de una etiqueta o directamente mediante una dirección de memoria.

Ejemplos

```
movq var, %rax
```

Carga en el registro `rax` *8 bytes a partir de la dirección de memoria* `var`*.*

```
movq 0x600880, %rax
```

Carga en el registro `rax` *8 bytes a partir de la dirección* `0x600880`.

```
movq %rax, var
```

Carga el contenido del registro `rax` *en la dirección de memoria* `var`.

5.7.4. Modo de direccionamiento indirecto con registro

En este caso, uno de los operandos hace referencia a un dato en una posición de memoria. El operando habrá de especificar un registro entre paréntesis. Dicho 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.

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

Guarda en la dirección de memoria especificada por `rbx` *el valor almacenado en* `rax`.

5.7.5. Modo de direccionamiento indexado

En este caso, uno de los operandos hace referencia a un dato que se encuentra almacenado en una posición de memoria. Dicho 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. O al revés, la constante se puede usar como índice y el registro como base.

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`.

5.7.6. Modo de direccionamiento relativo

En este caso, uno de los operandos hace referencia a un dato que se encuentra almacenado en una posición de memoria. Dicho 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

```
movq 3(%rbx, %rcx, 4), %rax
```

Carga en el registro `rax` *8 bytes a partir de la dirección* `rbx+rcx*4+3`.

```
movq (%rax, %rax, 2), %rax
```

Carga en el registro `rax` *8 bytes a partir de la dirección* `rax+rax*2`.

```
movq 4(%rbp, %rdx, 4), %rax
```

Carga en el registro `rax` *8 bytes a partir de la dirección* `rbp+rdx*4-4`.

```
movq 8(,%rax,4), %rax
```

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

5.8. Instrucción LEA

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

```
lea <operando fuente>, <operando destino>
```

El operando de origen es una dirección de memoria especificada con uno de los modos de direccionamiento de los procesadores mientras que el operando de destino es un registro de propósito general.

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. Así la instrucción `mov $str, %rax` es equivalente a

```
leaq str, %rax
```

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

Ejemplo

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

En las dos primeras instrucciones NO hay acceso a memoria. En la última instrucción SÍ hay acceso a memoria.

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

Ejemplos

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

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

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

5.9. 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 7, 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. 6 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).

Observación

En la Fig. 6 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*

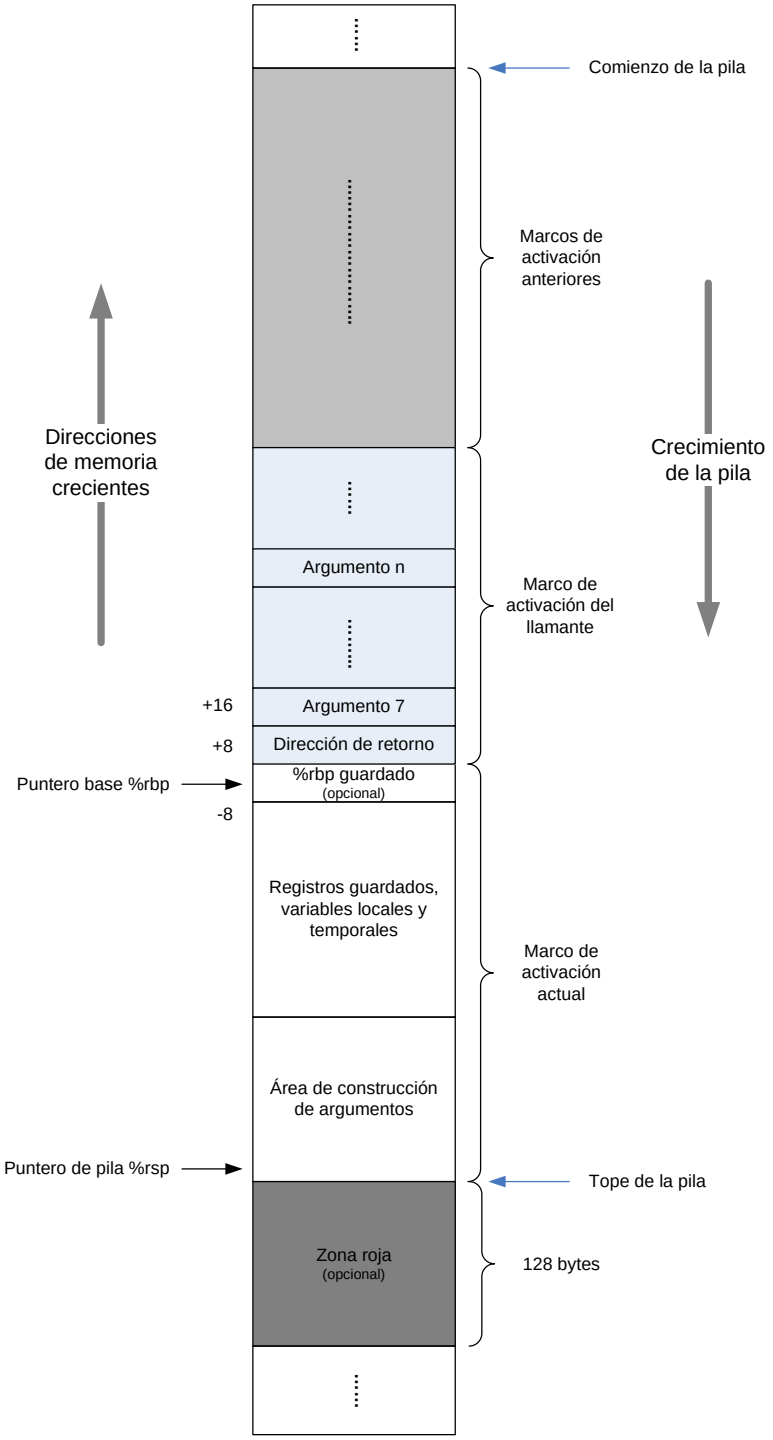


Figura 6: Diagrama de la estructura de pila x86-64.

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)

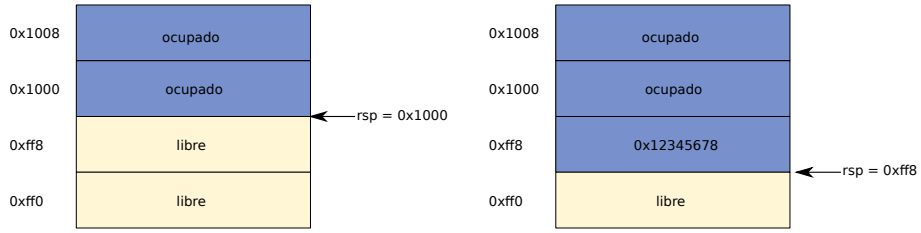


Figura 7: Diagrama de la memoria antes y después de ejecutar la instrucción `pushq`.

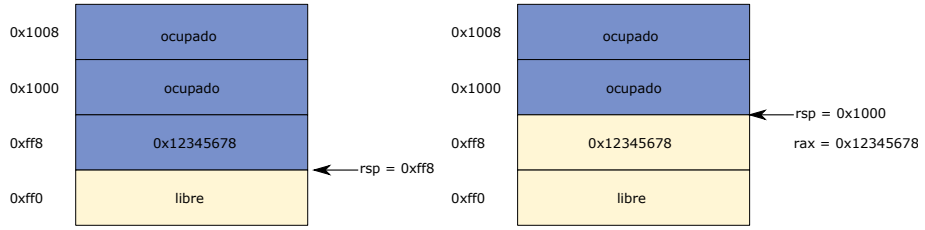


Figura 8: Diagrama de la memoria antes y después de ejecutar la instrucción `popq`.

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 7.

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. 8. Notar que el valor `0x12345678` continua almacenado en la dirección `0xff8`.

Observación

En la descripción de las instrucciones ***`push`*** *y* ***`pop`*** *solo podemos utilizar 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. 6 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, en lugar de ajustar el puntero de la pila en el prólogo y el epílogo.

6. Aritmética de Punto Flotante

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

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

Las instrucciones siguen algunas reglas:

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

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

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

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

6.1. Copias y conversiones

Al igual que con los registros de propósito general, existen instrucciones para copia de datos. Para los registros de punto flotante existen las instrucciones `movss` y `movsd` que

Tabla 2: Instrucciones de copia y conversiones para punto flotante [5].

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

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

copian un dato de precisión simple (*float*) y doble precisión (*double*), respectivamente, de un registro `xmm` a otro o desde/hacia la memoria.

A su vez existen múltiples instrucciones para convertir entre enteros y datos de punto flotante. En la Tabla 2 se recopilan las instrucciones de conversión.

Ejemplo

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

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

6.2. 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 Tabla 3 resume las operaciones más utilizadas para simple y doble precisión.

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

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

Ejemplo

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

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

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

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

  movq $0x3ffcccccccccccd, %rax
  # El valor inmediato es la representacion de 1.8 según IEEE 754
  movq %rax, -8(%rsp)
  movsd -8(%rsp), %xmm1    # Carga el valor 1.8 en xmm1
  movq $32, %rax
  cvtsi2sdq %rax, %xmm2
  # Carga el valor 32.0 convirtiendo el valor entero 32 de rax a xmm2
  mulsd %xmm1, %xmm0          # xmm0=xmm0*xmm1 => xmm0=t*1.8
  addsd %xmm2, %xmm0          # xmmo=xmm0+xmm2 => xmmo=t*1.8+32
  # como el valor de retorno se escribe en xmm0 hemos terminado
  ret
```

Al igual que con los valores enteros la arquitectura ofrece comparaciones de valores de punto flotante. Las instrucciones de comparación comparan dos valores (haciendo una resta virtual) y prenden las banderas correspondientes en el registro `rflags`. La comparación se comporta como una comparación de datos unsigned (i.e.: conviene utilizar `jae` para saltar por mayor o igual). Además, si los valores son incomparables (alguno es NaN) se prende la bandera `PF (Parity Flag)`. Las instrucciones de comparación en punto flotante se muestran en la Tabla 4

Tabla 4: Instrucciones de comparación en punto flotante[5].

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

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

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

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

6.3. Instrucciones SIMD

Los programas para procesamiento de señales multimedia (audio, imágenes, video, etc) muchas veces requieren repetir la misma operación en una gran cantidad de datos, por ejemplo para cada píxel realizar una determinada operación. Por ello, las arquitecturas actuales incluyen lo que se conoce como instrucciones *Streaming SIMD Extensiones* (SSE), donde SIMD significa *Single Instruction Multiple Data*. Es decir, son instrucciones que aplican la misma operación a muchos datos a la vez.

La mayoría de las instrucciones aritméticas SSE realizan operaciones paralelas con vectores de datos. Las operaciones vectoriales también se denominan operaciones empaquetadas (*packed* en inglés). Toman operandos vectoriales que consisten en múltiples elementos y todos los elementos se operan en paralelo. Sin embargo, algunas instrucciones SSE operan con escalares en lugar de vectores.

Recordemos que los registros `xmm0-xmm15` son de 128 bits por lo cual pueden alojar 4 valores de precisión simple o 2 de precisión doble o también 16 bytes, 8 words, 4 enteros de 32 bits o 2 de 64 bits. La Fig. 9 muestra los distintos tipos de datos que puede contener un registro `xmm`.

Hay varios tipos de instrucciones *packed*:

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

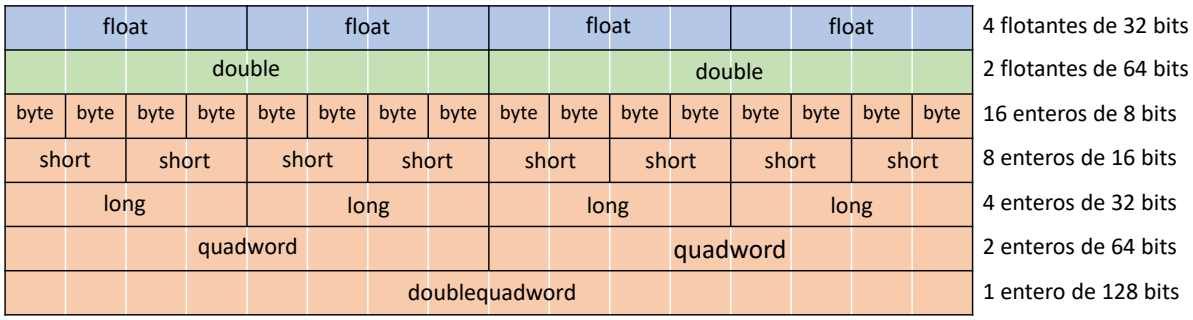


Figura 9: Estructura de los registros `xmm` (128 bits).

Tabla 5: Algunas intrucciones *packed*.

Mnemotécnico	**Descripción**
`movaps`	Mueve cuatro flotantes simple precisión alineados entre registros XMM o memoria.
`movapd`	Mueve dos flotantes dobles precisión alineados entre registros XMM o memoria.
`addps`	Suma flotantes simple precisión empaquetados.
`divps`	Divide flotantes simple precisión empaquetados.
`divss`	Divide flotantes simple precisión escalares.
`mulps`	Multiplica flotantes simple precisión empaquetados.
`subps`	Resta flotantes simple precisión empaquetados.
`cmpps`	Compara flotantes simple precisión empaquetados.
`andnps`	Realiza la operación AND NOT bit a bit de flotantes simple precisión empaquetados.
`andps`	Realiza la operación AND bit a bit de flotantes simple precisión empaquetados.
`orps`	Realiza la operación OR bit a bit de flotantes simple precisión empaquetados.
`xorps`	Realiza la operación XOR bit a bit de flotantes simple precisión empaquetados.

- Instrucciones lógicas.

La Tabla 5 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 de 256 bits, pero estas no serán vistas en este apunte.

Ejemplo

Veamos un ejemplo de instrucciones packed. La siguiente función suma cuatro flotantes almacenados a partir de la dirección etiquetada con `a` *con los cuatro flotantes almacenados a partir de la dirección etiquetada con* `b`*:*

```
.data
```

```
.align 16
a: .float 1.0, 2.0, 3.0, 4.0
b: .float 1.0, 2.0, 3.0, 4.0

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

Aquí la instrucción interesante es `addps` *que suma los 4 valores flotantes de precisión simple a la vez. La Fig. 10 ilustra esta instrucción. Notar que para poder usar la instrucción* `movaps` *los datos tienen que estar alineados a 16 bytes. Esto se puede lograr con la directiva* `.align`*.*

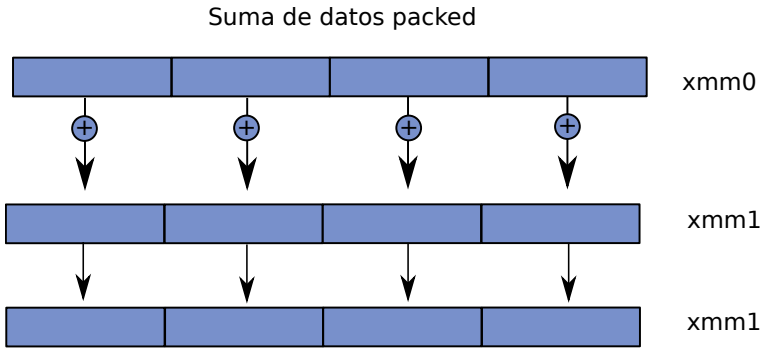


Figura 10: Instrucción `addps`

7. Funciones y Convención de Llamada

7.1. Funciones

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`) y el registro `rdi` solo se ha usado para ilustrar el funcionamiento con un código equivalente. En este código 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.

7.2. 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 específico, 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.

Convención de llamada para x86-64 en Linux

- Los seis primeros argumentos a la función son pasados por registro en el siguiente orden: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` (cuando son valores enteros o direcciones de memoria).
- Si son valores de punto flotantes pueden utilizarse hasta 8 de los registros `xmm` en el siguiente orden: `%xmm0`, `%xmm1`, `%xmm2`, `%xmm3`, `%xmm4`, `%xmm5`, `%xmm6` y `%xmm7`.
- Parámetros grandes mayores a 64 bits, por ejemplo estructuras pasadas por valor, se pasan utilizando la pila.
- Cuando la función toma como argumento una mezcla de valores enteros y flotantes `rdi` será el primer valor entero, `xmm0` el primer valor flotante, y así sucesivamente. Así, en la función `void f(int, double, int, double)` los argumentos irán en `rdi,xmm0,rsi,xmm1`.
- Si hubiera más argumentos de los que se pueden pasar por registros, éstos son pasados a la función utilizando la pila.
- 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 `%r12` a `%r15`. Esto no quiere decir que no los pueda usar sino que al retornar deben tener el mismo valor que al comenzar la función. La función podría guardarlos temporalmente en memoria o pila y restaurarlos antes de retornar. Estos registros se conocen como *callee saved* ya que es responsabilidad del llamado preservarlos.
- Los otros registros (incluso los utilizados para pasar los argumentos) pueden ser modificados libremente por la función sin necesidad de restaurar sus valores. Si el llamante desea preservar sus valores es responsabilidad de él, por lo cual estos registros se conocen como *caller saved*. En la Fig. 1 se puede observar el rol de los registros en la llamada a función.
- El bit `DF` de `rflags` está inicialmente apagado (esto incrementará los punteros en instrucciones de manejo de cadena) y debe ser apagado al finalizar la función (y antes de llamar a otra función).

- 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:

```
#prólogo
pushq %rbp        # Guardar el valor del rbp del llamante
movq %rsp, %rbp   # La pila para esta función comienza en el tope (vacía)
...
...
...
#epílogo
movq %rbp, %rsp   # El registro rsp vuelve a apuntar al tope de la pila anterior.
popq %rbp         # Restaurar el rbp del llamante
```

Ejemplo

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

```
...
movq $40, %rdi    # el primer argumento es 40 y va en el registro rdi
movq $45, %rsi    # el segundo argumento es 45 y va en el registro rsi
call sum          # guarda la dirección 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:
    # Prólogo
    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
                         # aquí el resultado YA está en rax

    # Epílogo
    movq %rbp, %rsp
    popq %rbp

    ret            # Retorna a la siguiente instrucción luego de call sum
```

Apéndices

A. 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 7.

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

Luego podemos ejecutar mediante:

```
./main
```

y obtendremos el resultado:

```
15.140000
```

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

B. Depurando el código con GDB

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

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

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

Luego podemos iniciar la sesión de depuración con GDB:

```
gdb ./main
```

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

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

Referencias

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

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

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

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

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

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

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

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

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

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

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

[12] *Debugging Assembly Code with GDB.*

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

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

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

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