

# Tema 5. Lenguajes del computador: alto nivel, ensamblador y máquina

Fundamentos de Computadores  
Curso 2018-19

# Índice

## **5.1 Introducción**

- 5.1.1 Programas e instrucciones
- 5.1.2 Codificación de las instrucciones
- 5.1.3 Tratamiento de las instrucciones
- 5.1.4 Tipos de instrucciones

## **5.2 Jerarquía de traducción**

- 5.2.1 Lenguajes de alto nivel
- 5.2.2 Compiladores y ensambladores
- 5.2.3 Código objeto
- 5.2.4 Librerías
- 5.2.5 Enlazadores y cargadores
- 5.2.6 Visión global de la jerarquía de traducciones

## **5.3 Introducción al ISA Intel x86-64**

- 5.3.1 Ensamblador del x86-64
- 5.3.2 Operandos de las instrucciones x86-64
- 5.3.3 Repertorio de instrucciones x86-64

## **5.4. Codificación de las instrucciones**

- 5.4.1 Formato y codificación de instrucciones
- 5.4.2 Ejemplos de codificación en x86-64
- 5.4.3 Reubicación de código
- 5.4.4 Espacio virtual de direccionamiento

# Índice

## **5.1 Introducción**

- 5.1.1 Programas e instrucciones
- 5.1.2 Codificación de las instrucciones
- 5.1.3 Tratamiento de las instrucciones
- 5.1.4 Tipos de instrucciones

## **5.2 Jerarquía de traducción**

- 5.2.1 Lenguajes de alto nivel
- 5.2.2 Compiladores y ensambladores
- 5.2.3 Código objeto
- 5.2.4 Librerías
- 5.2.5 Enlazadores y cargadores
- 5.2.6 Visión global de la jerarquía de traducciones

## **5.3 Introducción al ISA Intel x86-64**

- 5.3.1 Ensamblador del x86-64
- 5.3.2 Operandos de las instrucciones x86-64
- 5.3.3 Repertorio de instrucciones x86-64

## **5.4. Codificación de las instrucciones**

- 5.4.1 Formato y codificación de instrucciones
- 5.4.2 Ejemplos de codificación en x86-64
- 5.4.3 Reubicación de código
- 5.4.4 Espacio virtual de direccionamiento

# Programas e instrucciones

- Instrucción  $\equiv$  Conjunto de símbolos que representa una orden de operación o tratamiento para el computador.
- Programa  $\equiv$  Conjunto ordenado de instrucciones que debe ejecutar el computador sobre los datos para procesarlos y obtener un resultado.
- Las instrucciones se almacenan en memoria principal en un orden determinado, y se van ejecutando en secuencia
- La secuencia sólo se rompe por posibles instrucciones de salto (bucles, condiciones if, saltos a funciones, etc.)

# Codificación de las instrucciones

- Cada instrucción indica una acción determinada a realizar por la CPU. P.e.:
  - Traer un dato de memoria a un registro de la CPU (o viceversa),
  - sumar dos registros y colocar el resultado en otro,
  - comparar dos registros y, dependiendo del resultado, saltar a otro lugar del programa o continuar secuencialmente,
  - Etc.
- Como todo en un computador (datos numéricos, caracteres, imágenes, etc.), las instrucciones en última instancia se codifican como ristras de bits
  - (de longitud fija o variable, dependiendo de la arquitectura).

# Codificación de las instrucciones

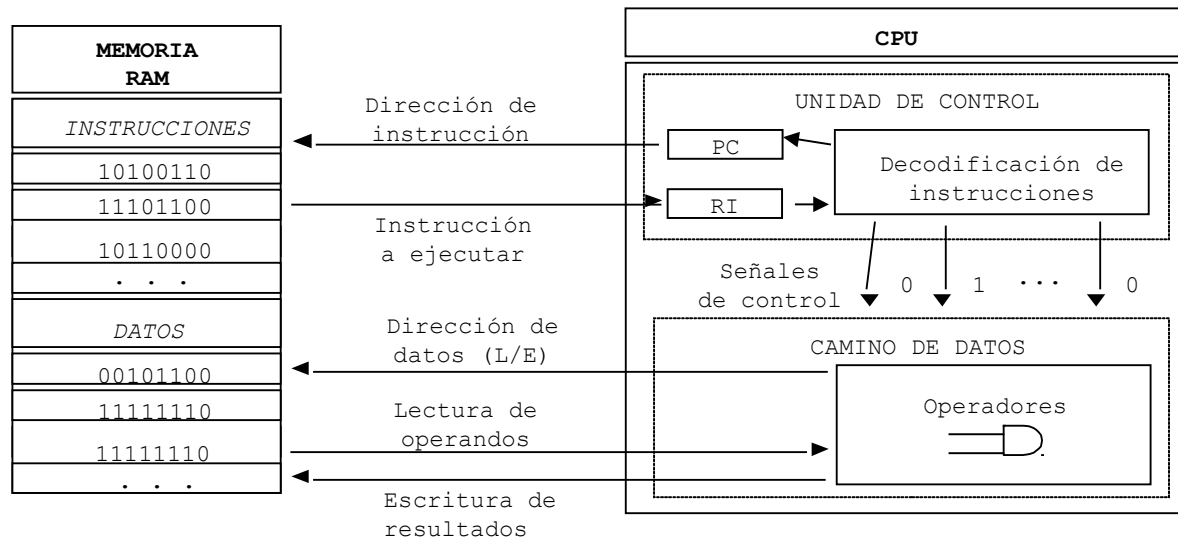
- Para codificar toda la información necesaria, las instrucciones se organizan en campos de bits.
- P.e., para una instrucción de suma acumulativa de un registro sobre otro, el formato podría ser:

Código operación (6 bits)	Registro 1 (5 bits)	Registro 2 (5 bits)
------------------------------	------------------------	------------------------

- La unidad de control (UC) de la CPU analizará e interpretará los distintos campos para saber:
  - La operación que debe llevar a cabo.
  - Los operandos de entrada.
  - El lugar en el que dejar el resultado.
- En este ejemplo, se permitirían hasta 64 códigos de operación distintos, y 32 posibles registros fuente/destino.
- Obviamente, distintos tipos de instrucciones (aritmético-lógicas, movimiento de datos, salto, etc.) utilizarán distintos formatos (puesto que necesitan codificar información distinta).

# Tratamiento de las instrucciones

- En todo momento, la UC mantiene:
  - Contador de programa (PC): contiene la dirección de la instrucción a ejecutar.
    - Tanto para ejecución secuencial (ver dirección de la instrucción siguiente) como para los saltos (condicionales o no), su constante actualización corresponde a la UC.
  - Registro de instrucción (RI): contiene la instrucción a ejecutar.



# Tipos de instrucciones

- A mayor número de instrucciones:
  - Más complejidad de la UC.
  - Mayor número de bits requeridos por el campo código de operación.
- Dos tendencias a este respecto:
  - RISC (*Reduced Instruction Set Computers*): pocas instrucciones, sencillas y se ejecutan en pocos ciclos.
  - CISC (*Complex Instruction Set Computers*): muchas instrucciones, complejas y muchos ciclos de reloj.
- Tipos de instrucciones:
  - Instrucciones de movimiento de datos:
    - A/desde/entre registros CPU/direcciones de memoria.
  - Instrucciones aritmético-lógicas.
    - Suma, resta, multiplicación, división, and, or, desplazamientos, ...
    - Operaciones punto flotante.
  - Instrucciones de salto
    - Condicionales
    - Incondicionales
    - Manejo de subrutinas



# Índice

## **5.1 Introducción**

- 5.1.1 Programas e instrucciones
- 5.1.2 Codificación de las instrucciones
- 5.1.3 Tratamiento de las instrucciones
- 5.1.4 Tipos de instrucciones

## **5.2 Jerarquía de traducción**

- 5.2.1 Lenguajes de alto nivel
- 5.2.2 Compiladores y ensambladores
- 5.2.3 Código objeto
- 5.2.4 Librerías
- 5.2.5 Enlazadores y cargadores
- 5.2.6 Visión global de la jerarquía de traducciones

## **5.3 Introducción al ISA Intel x86-64**

- 5.3.1 Ensamblador del x86-64
- 5.3.2 Operandos de las instrucciones x86-64
- 5.3.3 Repertorio de instrucciones x86-64

## **5.4. Codificación de las instrucciones**

- 5.4.1 Formato y codificación de instrucciones
- 5.4.2 Ejemplos de codificación en x86-64
- 5.4.3 Reubicación de código
- 5.4.4 Espacio virtual de direccionamiento

# Lenguajes de alto nivel

- Las instrucciones que procesa la CPU están almacenadas en memoria principal en binario (0 y 1):
  - Se dice que son instrucciones máquina.
  - Programar directamente de esa forma sería posible, pero muy difícil, propenso a errores y lejos del modo de pensar humano.
- Lenguajes de programación: instrucciones representadas simbólicamente (mediante palabras, abreviaturas, etc.).  
Ejemplo ensamblador Intel x86-64:

00000001 11010000 == add %edx, %eax

"Sumar"      "Cada registro un nombre"      "fuente"      "fuente y destino"

- Problema: el procesador no entiende "add".
  - Solución: usar la máquina para traducir a lenguaje binario (código máquina, tb llamado lenguaje máquina): programa traductor.
- Tipos de lenguajes de programación:
  - Lenguaje ensamblador (de bajo nivel).
  - Lenguaje de alto nivel (LAN): Java, C, C++, etc.

# Lenguajes de alto nivel

- Permiten al programador expresar sus programas en un lenguaje formal, relativamente cercano a su forma de pensar:
  - Variables, tipos de datos, funciones/procedimientos, asignación de variables, condiciones, bucles, etc.
- Multitud de *paradigmas* (imperativo, orientado a objetos, funcional, ...) y de lenguajes concretos (C, C++, Java, Haskell, etc.)
- Ilustraremos nuestros ejemplos con C:
  - Alto nivel, pero más cercano a la máquina.
  - Lenguaje de programación nativo de Unix/Linux
- Ejemplo:

```
#include<stdio.h>
int array[10] = {10,9,8,7,6,5,4,3,2,1};
int main() {
    int i;
    for(i=0;i<10;i++)
        array[i] = array[i]*i;
    for(i=0;i<10;i++)
        printf("%d ",array[i]);
    printf("\n");
}
```

# Lenguajes de alto nivel

- El programa anterior declara un vector global (`array`) de 10 datos de tipo entero, y una variable entera local (`i`).
- Después, tiene una función principal (`main`), que va recorriendo el array (bucle `for`).
- En cada paso del bucle se lee una posición del array, se hace una operación sobre él, y se almacena el resultado en la misma posición (`array[i] = array[i]*i;`).
- Finalmente, se vuelve a recorrer el vector para imprimir sus contenidos (función `printf`, de la librería estándar de C, con fichero de cabecera `stdio.h`)

```
#include<stdio.h>
int array[10] = {10,9,8,7,6,5,4,3,2,1};
int main() {
    int i;
    for(i=0;i<10;i++)
        array[i] = array[i]*i;
    for(i=0;i<10;i++)
        printf("%d ",array[i]);
    printf("\n");
}
```

# Compiladores y ensambladores

- **Compilador:**
  - Transforma el código en lenguaje de alto nivel (en texto ASCII) a ensamblador (lenguaje ya “pegado” a la máquina, pero aún expresado en texto ASCII).
- **Ensamblador:**
  - Convierte el programa ensamblador en un *fichero objeto*, que ya contiene datos binarios.
- **Contenido de un fichero objeto:**
  - Instrucciones en lenguaje máquina.
  - Datos (ya en formatos de almacenamiento interno: enteros en C2, reales en punto flotante, texto en ASCII, etc.)
  - Información de *reubicación* (para accesos a memoria, saltos, etc.)
    - Necesaria porque los programas se dividen en módulos objeto compilados por separado...
    - ...que luego en enlazador (*linker*) juntará en uno sólo.
    - En ese momento, se “pegarán todos los módulos”, se fijarán las referencias cruzadas entre ellos (acceso a datos o llamadas a funciones de otro módulo), y se usará la información anterior para fijar las direcciones definitivas en el programa completo resultante.

# Librerías

- Librerías:
  - Cuando un programador genera un conjunto de módulos relacionados, que pueden ser reutilizados por otros...
  - ... une todos los ficheros objetos generados en un fichero llamado **librería**
    - P.e., librerías de cálculo matricial, estadística, programación de gráficos 3D, de acceso a redes, ...
  - Centenares de librerías de funcionalidad muy heterogénea en una distribución de Linux convencional:
    - Para C en concreto, existe una librería estándar (para entrada/salida en ficheros/pantalla, funciones matemáticas, etc.), presente en todas las implementaciones (Windows/Linux/etc.) (p.e ejemplo el `printf` anterior está en esa librería, y su fichero de cabecera es el `stdio.h`).
  - Veremos que el código generado al compilar nuestro programa acaba enlazándose con código ya precompilado contenido en ficheros de librería:
    - Directorios `/lib`, `/usr/lib`, `/usr/local/lib` y similares, en Linux.

# Librerías

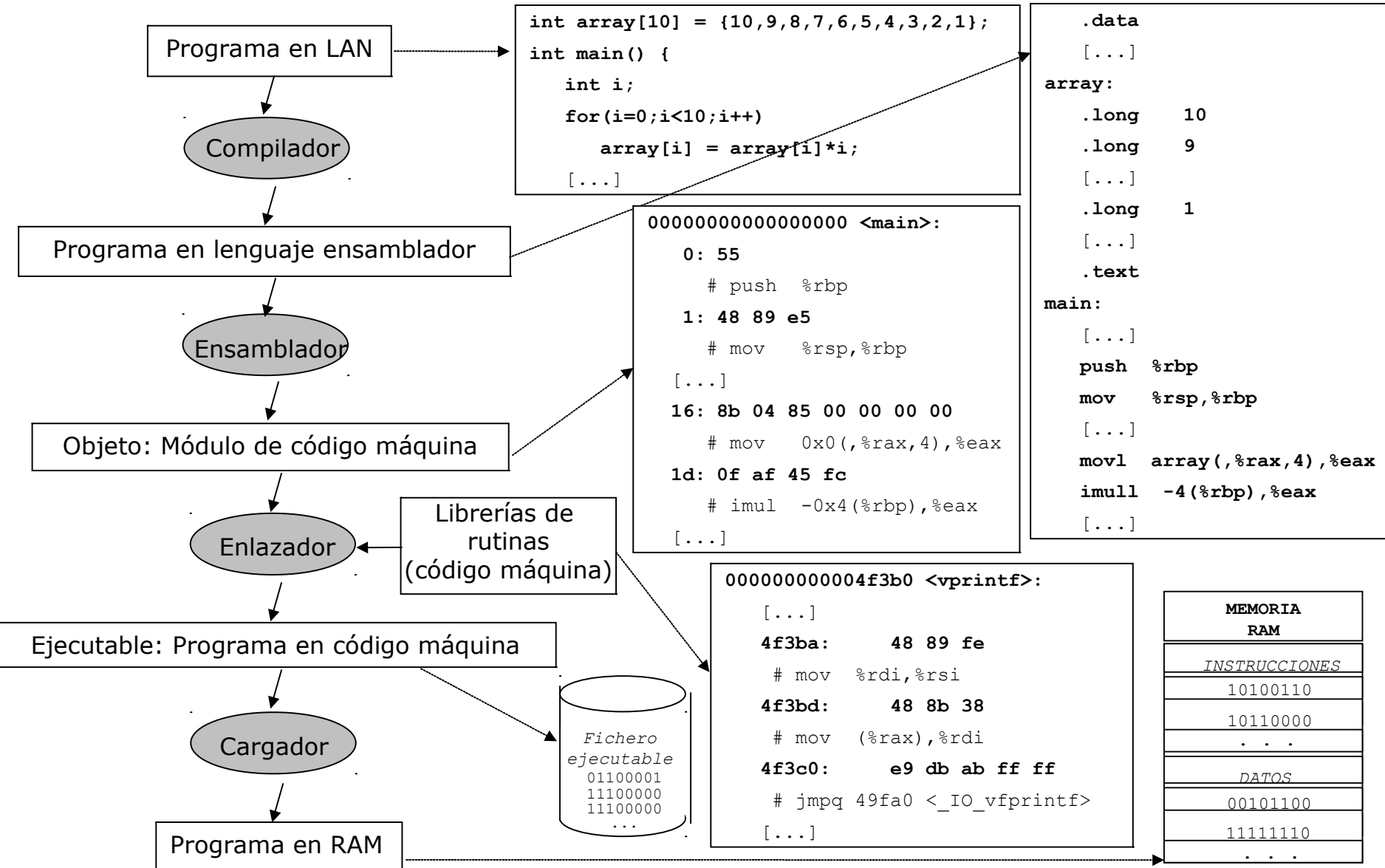
- Librerías estáticas vs. librerías dinámicas:
  - Las **librerías estáticas** almacenan código que, de alguna forma, es “cortado y pegado” en nuestro ejecutable final.
    - Así, si llamamos a más funciones, mayor es el tamaño de nuestro ejecutable final
    - En Linux, están en ficheros con extensión `.a`
      - P.e. `/usr/lib/lib*.a`, ...
  - Las **librerías dinámicas**, por el contrario, el código de las funciones de biblioteca no se incluye en el ejecutable final, sino que éste simplemente almacena la información necesaria para cargar dicho código en memoria desde el fichero de la librería en el momento de la ejecución.
    - Además, dichas funciones pueden ser compartidas por varios ejecutables simultáneamente, sin duplicar espacio necesario en memoria.
    - Ventaja principal: no se desperdicia espacio ni en disco ni en memoria
    - Inconveniente principal: el fichero ejecutable, al cambiarlo de máquina, puede no funcionar (necesita que la(s) librería(s) utilizada(s) esté(n) en el computador destino)
    - En Linux, están en ficheros con extensión `.so`
      - P.e. `/usr/lib/lib*.so`, ...

# Enlazadores y cargadores

- Enlazador (*linker*):
  - Une los distintos objetos generados por el programador entre sí, ...
  - ... y también con las funciones de librería utilizadas, programadas por otros y preexistentes en el sistema (ya compiladas y listas para enlazar contra ellas)
    - Directorios `/lib`, `/usr/lib`, `/usr/local/lib/`, etc. en Linux
  - Genera un **fichero ejecutable** final
- Cargador (*loader*):
  - Módulo del SO que lee el fichero ejecutable del disco, lo ubica en memoria y le pasa el control para comenzar la ejecución.



# Visión global de la jerarquía de traducciones



# Índice

## **5.1 Introducción**

- 5.1.1 Programas e instrucciones
- 5.1.2 Codificación de las instrucciones
- 5.1.3 Tratamiento de las instrucciones
- 5.1.4 Tipos de instrucciones

## **5.2 Jerarquía de traducción**

- 5.2.1 Lenguajes de alto nivel
- 5.2.2 Compiladores y ensambladores
- 5.2.3 Código objeto
- 5.2.4 Librerías
- 5.2.5 Enlazadores y cargadores
- 5.2.6 Visión global de la jerarquía de traducciones

## **5.3 Introducción al ISA Intel x86-64**

- 5.3.1 Ensamblador del x86-64
- 5.3.2 Operandos de las instrucciones x86-64
- 5.3.3 Repertorio de instrucciones x86-64

## **5.4. Codificación de las instrucciones**

- 5.4.1 Formato y codificación de instrucciones
- 5.4.2 Ejemplos de codificación en x86-64
- 5.4.3 Reubicación de código
- 5.4.4 Espacio virtual de direccionamiento

# Ensamblador del x86-64

- Recordemos el programa de ejemplo en C usado en secciones anteriores (escrito en un fichero de texto ASCII `main.c`):

```
#include<stdio.h>
int array[10] = {10,9,8,7,6,5,4,3,2,1};
int main() {
    int i;
    for(i=0;i<10;i++)
        array[i] = array[i]*i;
    for(i=0;i<10;i++)
        printf("%d ",array[i]);
    printf("\n");
}
```

- En principio, dicho programa puede compilarse directamente para generar ya un programa ejecutable (`main`):

```
$ gcc-4.8 main.c -o main
$ ./main
0 9 16 21 24 25 24 21 16 9
```

# Ensamblador del x86-64

- En lugar de eso, aquí vamos a ir siguiendo todos los pasos que en realidad ocurren en esta compilación, para ver todos los niveles de la jerarquía de traducción:
  1. Lenguaje de alto nivel (C)
  2. Lenguaje ensamblador de la arquitectura Intel x86-64
  3. Código máquina x86-64
  4. Código máquina x86-64 enlazado y cargado en memoria
- En primer lugar, el compilador traduce las instrucciones del lenguaje de alto nivel (en este caso C) a **lenguaje ensamblador**.
- El lenguaje ensamblador no es más que una forma intermedia entre “lo que piensa el programador” y como “la máquina realmente lo implementa”...
- ...pero aún así, esta forma se expresa de forma aún legible de modo relativamente fácil para un humano.

# Ensamblador del x86-64

- Por ejemplo, echémosle un vistazo al código ensamblador generado para el `main.c` anterior. Con el siguiente comando se genera dicho código en otro fichero ASCII, `main.s`:

```
$ gcc-4.8 -fno-asynchronous-unwind-tables -S main.c -o main.s
```

- Dicho fichero tiene más o menos este aspecto (recortado a lo que más nos interesa):

	Variable i (en pila)	array+rax*4
<pre>.data [... Segmento de datos ...] array:     .long 10     .long 9     [...]     .long 1 .LC0:     .string "%d "  .text [... Segmento de texto ...] main: [...codigo inicio...]     jmp .L2 [... Sigue ...]</pre>	<pre>.L3: [...Bucle sobre array...]     movl -4(%rbp), %eax     cltq     movl array(,%rax,4), %eax     imull -4(%rbp), %eax     movl %eax, %edx     movl -4(%rbp), %eax     cltq     movl %edx, array(,%rax,4)     addl \$1, -4(%rbp)  .L2: cmpl \$9, -4(%rbp)     jle .L3     movl \$0, -4(%rbp)     jmp .L4 [... Sigue ...]</pre>	<pre>.L5: [...Bucle impresión...]     movl -4(%rbp), %eax     cltq     movl array(,%rax,4), %eax     movl %eax, %esi     movl \$.LC0, %edi     movl \$0, %eax     call printf     addl \$1, -4(%rbp)  .L4: cmpl \$9, -4(%rbp)     jle .L5 [...código finalizar...]     ret</pre>

# Ensamblador del x86-64

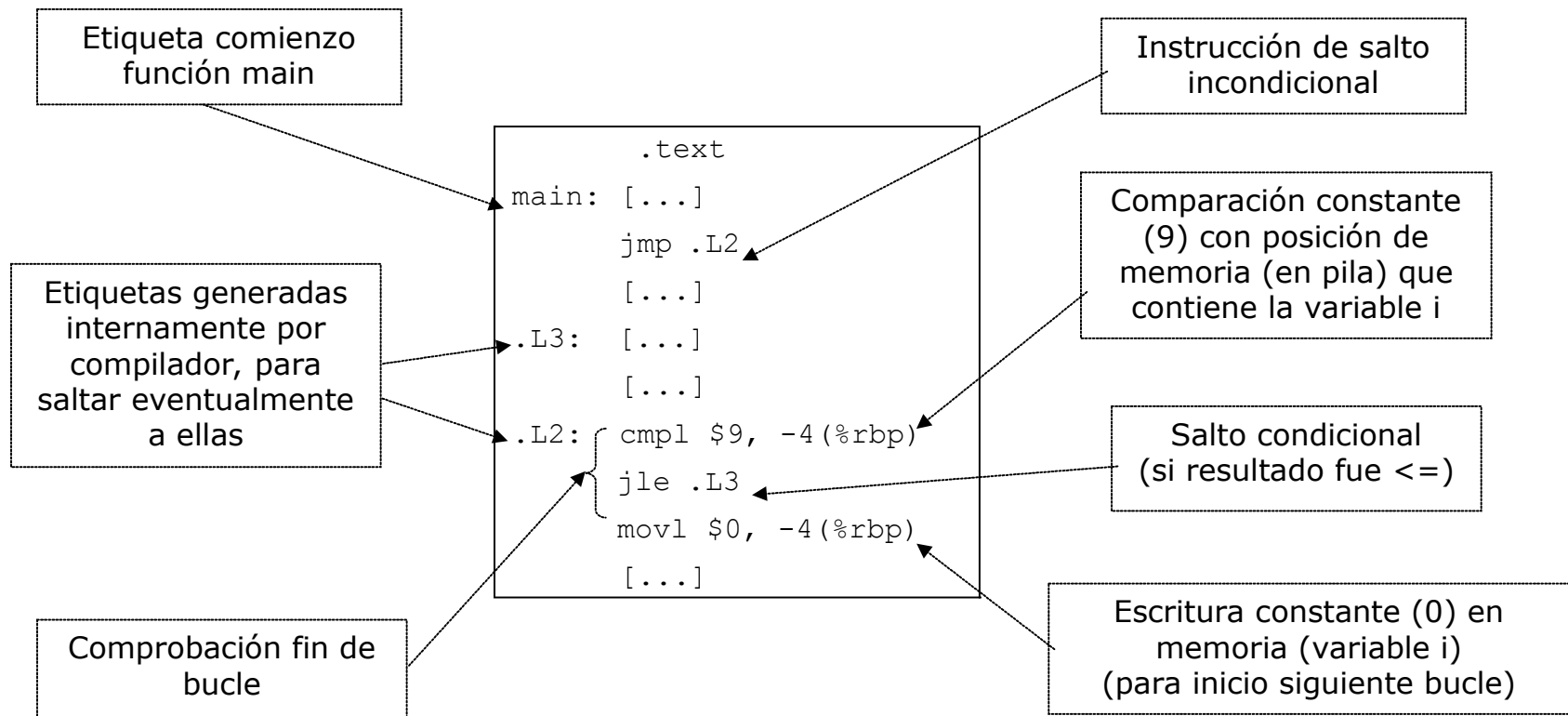
- Los **datos** (`array`, cadena `"%d "`) están almacenados de modo explícito en el segmento de datos (`.data`).
  - Cambiando los valores concretos en el fichero fuente `main.c` y recompilando podríamos comprobar cómo cambian las directivas de ensamblador correspondientes.

```
.data
[... Segmento datos ...]
array: .long    10
       .long    9
       [...]
       .long    1
.LC0:  .string  "%d "
```

- La variable `i` es local a la función `main()`, y se almacena en un lugar especial de la memoria, que veremos más adelante (la *pila*).
- Observar el uso de las **etiquetas** (identificadores seguidos de `:`), que representan direcciones de memoria (pero aún no están instanciadas a valores numéricos concretos):
  - Algunas vienen del propio código fuente en C (p.e. `array:`)
  - Otras son creadas automáticamente por el compilador (p.e. `.LC0:`).
- Ahora veremos que en el segmento de código (no sólo en los datos) también se generan ambos tipos de etiquetas.

# Ensamblador del x86-64

- El **código** (segmento `.text`) está en forma de instrucciones en ensamblador, que indican una operación elemental realizable directamente por la CPU.
- Por ejemplo, he aquí el código correspondiente a la comprobación de fin de bucle `for` (se ejecuta mientras `i <= 9`):

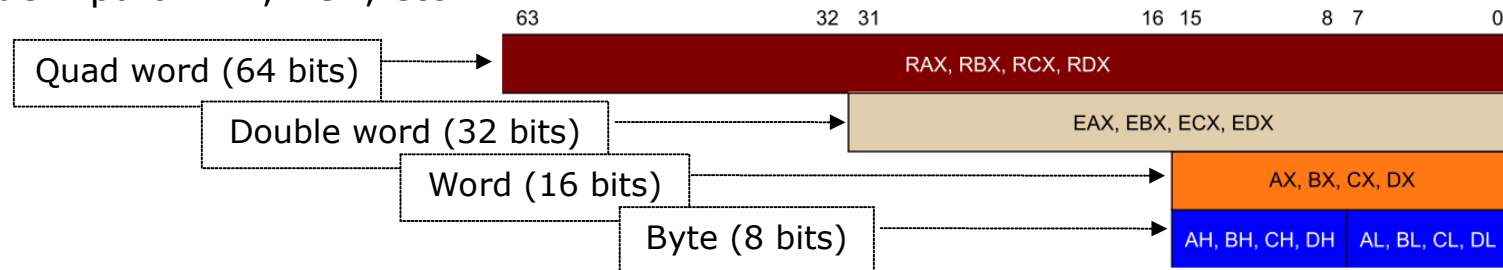


# Operandos de las instrucciones x86-64

## • Operandos de las instrucciones (1/2)

### – Registros:

- Contienen valores intermedios de nuestros cálculos
- Son de acceso muy rápido, puesto que están en la propia CPU
- Son de 64 bits en Intel x86-64. Estos son los principales:
  - Uso general: **RAX, RBX, RCX, RDX, R8-R15**
  - Índices (útiles para acceder a posiciones de un array, p.e): **RSI, RDI**
  - Para acceder a la pila: **RSP** (puntero de pila), **RBP** (puntero base de pila)
  - Puntero de instrucción: **RIP** (indica por dónde va ejecutándose el programa)
  - Registro de estado (flags): **RFLAGS** (contiene información sobre el estado del procesador y el resultado de la ejecución de las instrucciones; afecta a los saltos condicionales)
- Se puede también trabajar con menos de 64 bits (32, 16 u 8): nombres alternativos:
  - P.e. RAX=64 bits; EAX = 32 bits inferiores de RAX; AX = 16 bits inferiores de EAX; AL = 8 bits inferiores de AX; AH = 8 bits superiores de AX.
  - Ídem para EBX, ECX, etc. :





# Operandos de las instrucciones x86-64

- Operandos de las instrucciones (1/2)
  - **Memoria:**
    - Tabla de  $2^{64}$  celdas direccionables, con un byte por dirección (máximo 4 Exabytes). En realidad los programas se mueven en un *espacio virtual de direcciones* (el hardware luego transforma esas direcciones a otras físicas diferentes).
    - En x86-64 se puede trabajar muchas veces directamente con la memoria, sin pasar por registros:
      - P.e., en Intel se puede sumar una constante al dato almacenado en una posición de memoria con una sola instrucción, mientras que en otras muchas arquitecturas RISC eso siempre requiere traer primero el dato a un registro, para luego hacer la suma y volver a almacenarlo en memoria; es decir, harían falta un mínimo de tres instrucciones
    - Acceso a memoria más lento que a registros (variables fuera de la CPU)
    - Posiciones de memoria representadas por **etiquetas** (ya sea en los datos, para vectores, variables, etc.), o en el código (para representar direcciones de comienzo de rutinas, etiquetas para saltar, etc.)
  - **Constantes:**
    - Ciertas instrucciones admiten valores constantes (p.e., mover un valor 7 a una posición de memoria, comparar el valor 1983 con el contenido de un registro, etc.). En ensamblador se ponen siempre precedidas por un \$ (p.e. \$1983).
    - Se les suele llamar operandos **inmediatos**, porque están directamente disponibles en la propia codificación de la instrucción (no hay que buscar el operando en el segmento de datos ni en ningún registro)

# Repertorio de instrucciones x86-64

- Cada instrucción ensamblador representa una operación elemental realizable directamente por la CPU.
- Cada CPU concreta posee su propio **repertorio de instrucciones** (más conocido como ISA: Instruction Set Architecture).
- Objetivos de un ISA:
  - Permitir que el diseño del procesador sea simple.
  - Permitir que el diseño del compilador sea sencillo.
  - Maximizar el rendimiento.
  - Minimizar el coste.
- En todos estos ejemplos estamos usando el ISA x86-64, presente en los procesadores de los PCs en modo 64 bits
  - En el caso de Intel, un objetivo añadido fue la compatibilidad de código con procesadores anteriores, lo que llevó a soluciones de compromiso quizá menos elegantes y eficientes...
  - ...pero ayudaron a mantener la cuota de mercado.

# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64:

- **1. Instrucciones aritmético lógicas:**

- Sirven para hacer operaciones aritméticas (**suma, resta, multiplicación**, etc.) y/o lógicas (**and, or, not**, etc.) con los operandos.
    - Ejemplo: suma de un registro sobre otro:

```
add %rbx, %rax
```

- Suma RBX a RAX, y deja el resultado en RAX
    - También posible para 8, 16 y 32 bits:

```
add %bh, %al
```

```
add %bx, %ax
```

```
add %ebx, %eax
```

- De modo similar funcionan otras operaciones aritméticas (resta, multiplicación, división, ...) y lógicas (and, or, xor, ...):

```
and %rbx, %rdx
```

```
sub %ch, %ah
```

```
or %edx, %ecx
```

```
xor %bh, %al
```

```
imul %ebx, %eax
```

```
...
```

# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64:

- 1. Instrucciones aritmético lógicas (cont.):**

- También existe la posibilidad de operar **directamente una constante** (valor inmediato), en decimal o hexadecimal:

`add $189, %rax`

`add $-70, %eax`

`add $0x4c, %ax`

Constantes precedidas  
siempre por símbolo \$

- Incrementos y decrementos:**

`inc %eax`

`dec %bl`

- Hay también operaciones de **desplazamiento de bits:**

`shr $0x03, %rax`

`shl $0x03, %rax`

`sar $0x03, %rax`

...

Desplazamiento lógico a  
derecha (3 bits)

Desplazamiento lógico a  
izquierda (3 bits)

Desplazamiento aritmético  
a derecha (3 bits)

# Repertorio de instrucciones x86-64

## • Tipos de instrucciones x86-64

### – 2. Instrucciones de movimiento de datos

- Sirven para mover datos y/o constantes entre memoria y/o registros:

```
mov %rbx, %rax
```

RAX := RBX

```
mov $3100, %ebx
```

EBX := 3100

```
mov $31, %ah
```

AH := 31

```
mov var, %ecx
```

ECX := Variable var (memoria)

### – Operandos de memoria:

- Los operandos en memoria pueden expresarse:

- Usando un registro como forma indirecta de expresar la dirección a la que se quiere acceder (puntero):

```
mov (%rax), %rcx
```

RCX := Mem[RAX] (celda "apuntada" por RAX)

- Usando aritmética más o menos simple para obtener la dirección final: p.e., suma de un registro base, más un desplazamiento fijo, más un desplazamiento indizado (realizando quizá una multiplicación por 4, 8, etc.):

```
mov %rax, 0x601060(%rbx, %rsi, 4)
```

Mem[0x601060 + RBX + RSI\*4] := RAX

# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64

- 3. Saltos incondicionales**

- Son instrucciones que, cuando el PC llega a ellas y se ejecutan, provocan que éste cambie a una dirección de código fija, indicada por una etiqueta.
    - El programa, por tanto, sigue ejecutándose a partir de la instrucción destino del salto (en la dirección indicada por la etiqueta correspondiente).
    - Rompen la secuencia normal de ejecución del programa (una instrucción tras otra)

- Por ejemplo:
 

```

          jmp .L1
          [...]
          .L1:  mov %ebx,%eax
          
```

- La etiqueta puede estar tanto hacia atrás como hacia delante en el programa:

```

.L2:  mov %ebx,%eax
      [...]
      jmp .L2
  
```

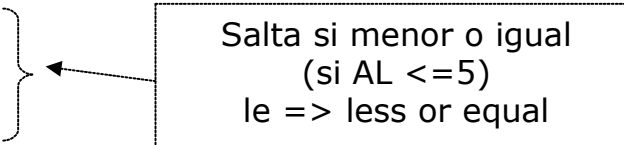
# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64

## 4. Saltos condicionales

- Funcionan como los saltos incondicionales, pero sólo ejecutan el salto si se cumple una determinada **condición**.
- Se emplean para implementar bucles (`for`, `while`, ...) y condiciones (`if`, `switch`, ...) de los lenguajes de alto nivel como C.
- La condición se comprueba en una instrucción `cmp` anterior, que deja las marcas (bits) adecuadas en el **registro de flags** (RFLAGS) para que después los saltos funcionen de acuerdo al resultado de la comparación.
- Por ejemplo:

```
    cmp $5,%al
    jle .L1
    [...]
.L1:  mov %rbx,%rax
```



Salta si menor o igual  
(si AL  $\leq$  5)  
le  $\Rightarrow$  less or equal

- Existen bastantes tipos de saltos condicionales:
  - `jle` (si igual), `jne` (si no igual), `jg` (si mayor), `jge` (si mayor o igual), `jl` (si menor), `jle` (si menor o igual), `ja` (si mayor sin signo), `jae` (si mayor o igual sin signo), `jb` (si menor sin signo), `jbe` (si menor o igual sin signo), ...

# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64

- 5. Soporte para procedimientos (1/9)**

- En C, todo código forma parte de una función (la principal es siempre `main`, que puede llamar a otras con nombres cualesquiera).
    - La división en funciones mejora la modularidad y legibilidad de los programas
    - Por ejemplo, una modificación sobre nuestro programa de ejemplo podría ser (equivalente al anterior):

```
#include<stdio.h>
int array[10] = {10,9,8,7,6,5,4,3,2,1};
int main() {
    int i;
    for(i=0;i<10;i++)
        array[i] = funcion(i);
    for(i=0;i<10;i++)
        printf("%d ",array[i]);
    printf("\n");
}
int funcion(int parametro) {
    return array[parametro]*parametro;
}
```

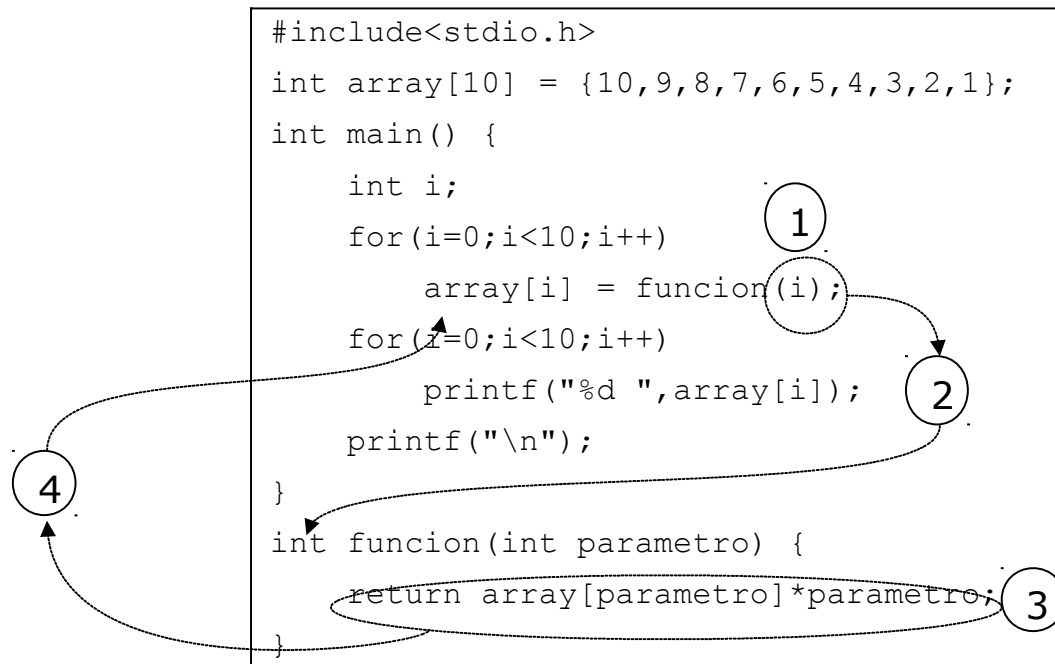


# Repertorio de instrucciones x86-64

## • Tipos de instrucciones x86-64

### – 5. Soporte para procedimientos (2/9)

- Una llamada a función supone un cambio en el flujo de ejecución de un programa, con los siguientes pasos:
  1. Se le pasan parámetros a la función (*i* en el ejemplo).
  2. Se pasan a ejecutar las instrucciones correspondientes a la función.
  3. Se devuelve un resultado.
  4. Se sigue ejecutando la siguiente instrucción posterior a la llamada.



# Repertorio de instrucciones x86-64

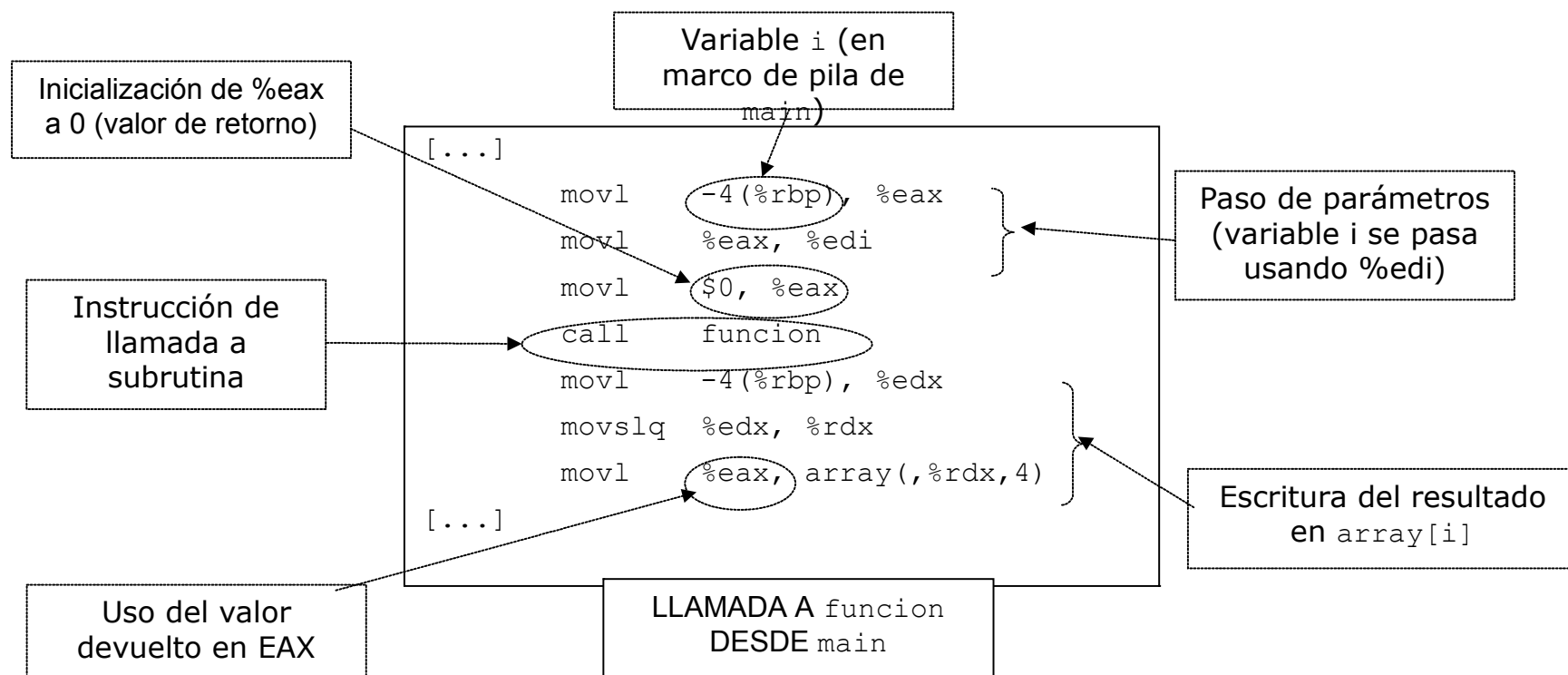
- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (3/9)**
  - Para dar soporte a los procedimientos, el lenguaje ensamblador debe proporcionar:
    1. Instrucciones que permitan **cambiar el flujo del programa** (saltar al principio del procedimiento)
    2. Proporcionar una estructura donde *a)* realizar el **paso de parámetros**, *b)* colocar **variables locales** y *c)* guardar la **dirección de retorno** (para poder volver luego a la instrucción siguiente a la llamada al procedimiento).
    3. Algún mecanismo para la **devolución de los resultados**.
  - Los pasos 2 y 3 podrían en principio hacerse con registros, (y de hecho éstos también se usan), pero sobre todo se usa para ello una estructura de datos localizada en una cierta parte de la memoria para ese uso específico: **la pila**.

# Repertorio de instrucciones x86-64

## • Tipos de instrucciones x86-64

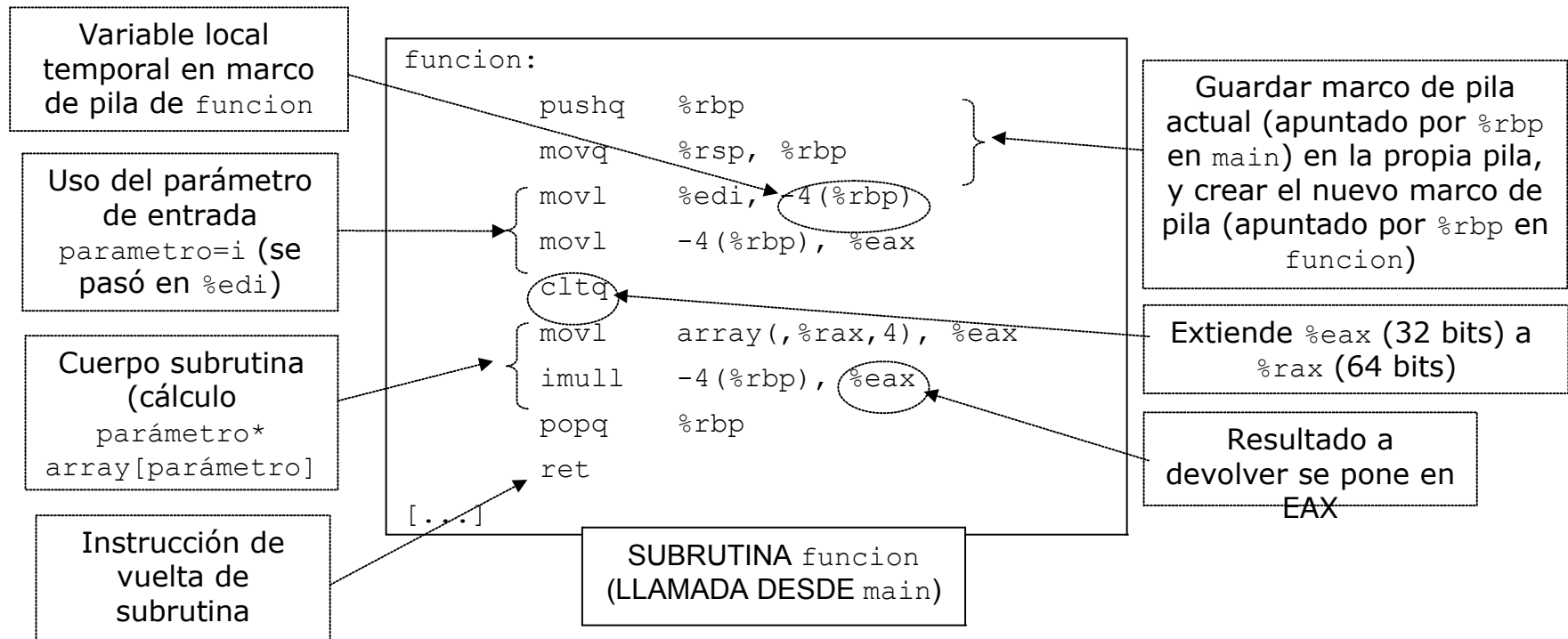
### – 5. Soporte para procedimientos (4/9)

- En ensamblador, las funciones C se mapean a “subrutinas”
- **Subrutina** = secuencia de instrucciones que pueden (o no) recibir unos parámetros de entrada, realizan alguna acción y pueden (o no) devolver un resultado (valor de retorno).



# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (5/9)**
    - Subrutina `funcion` llamada desde `main` (continuación)



Ilustraremos gráficamente el uso de los marcos de pila a través de los registros `%rsp` y `%rbp` en las transparencias siguientes...

# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (6/9)**
    - **Pila** = zona de memoria normal gestionada de una manera especial.
    - Empleada para:
      - Pasar posibles parámetros adicionales a subrutina (si usando registros no fuese suficiente).
      - Almacenar el valor de los registros que pueden ser machacados por la subrutina, de forma que su valor pueda ser restaurado al acabar la misma.
      - Almacenar variables que se usarán localmente a una subrutina, de forma que este espacio temporal se pueda recuperar al acabar la subrutina.
      - Almacenar dirección de vuelta a instrucción posterior a la llamada.
    - La pila es una estructura **LIFO** (*Last In First Out*): sólo dos operaciones posibles
      - **Apilar (push)**: guardar el contenido de un registro en la cima de la pila.
      - **Desapilar (pop)**: cargar en un registro el contenido de la cima de la pila.
    - Registro RSP (*Stack Pointer*): registro que apunta (almacena) la dirección de la cima de la pila.
    - Registro RBP (*Base Pointer*): registro que apunta a un lugar fijo de la pila durante la ejecución de cada rutina, para poder direccionar a partir de él los posibles parámetros y/o variables locales.

# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (7/9)**
    - **Funcionamiento de una pila:**

Estado final de la pila, suponiendo que al principio contenía A,B,C,D,E, el puntero de pila (`%rsp`) apuntaba a E, y se ejecutaron las siguientes operaciones:

1. Apilar F.
2. Apilar G.
3. Apilar H.
4. Desapilar H.
5. Desapilar G.
6. Apilar I.

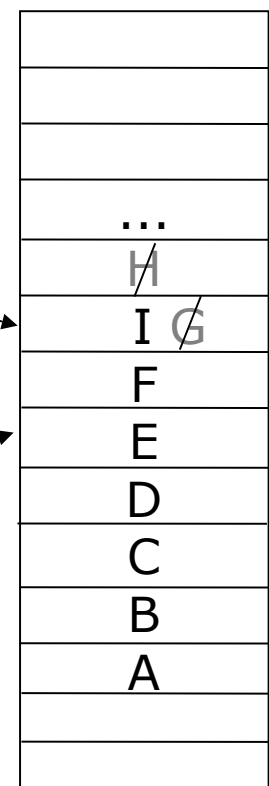
Direcciones más bajas  
(pila crece hacia ellas)

`%rsp`  
final

`%rsp`  
inicio

Direcciones  
más altas

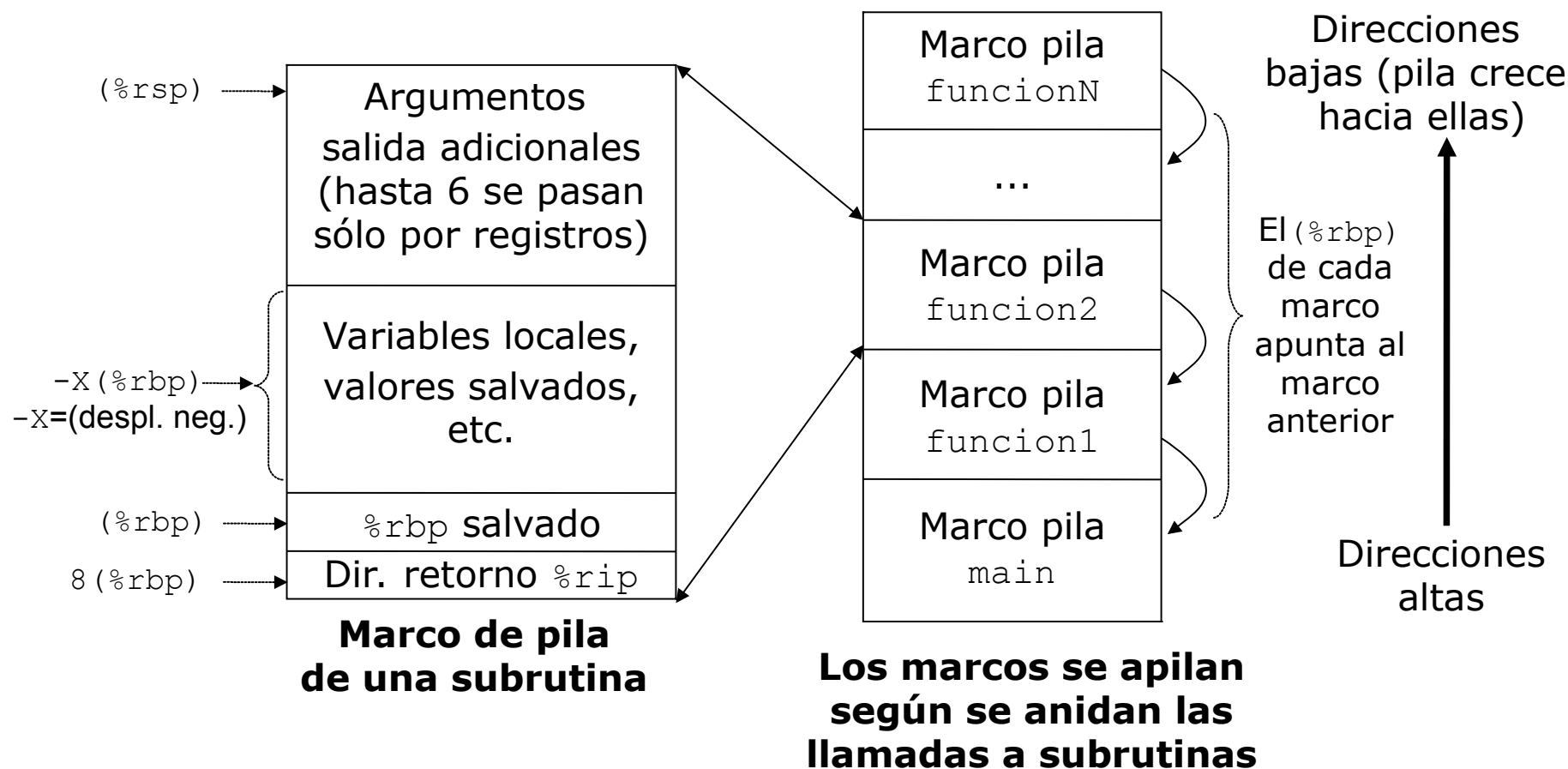
Pila



# Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (8/9)**
    - **Marcos de pila (stack frames):**

En este ejemplo `main` llama a `funcion1`, que llama a `funcion2`, ..., que llama a `funcionN`.



# Repertorio de instrucciones x86-64

## • Tipos de instrucciones x86-64

### – 5. Soporte para procedimientos (9/9)

#### • Manejo **explícito** de la pila en x86-64:

- Meter el contenido de un registro en la cima de la pila:

```
push %rbx
```

- Sería equivalente a (`%rsp`= *stack pointer*, puntero pila; pila crece hacia abajo):

```
sub $8,%rsp + mov %rbx, (%rsp)
```

- Sacar el contenido de un registro de la cima de la pila:

```
pop %rbx
```

- Sería equivalente a:

```
mov (%rsp), %rbx + add $8,%rsp
```

#### • Manejo **implícito** de la pila en llamadas/retorno a/desde subrutinas:

`call subrut` Automáticamente apila IP (`%rip`) (dirección de retorno)

`ret` Automáticamente desapila IP (`%rip`) (vuelve de la llamada)

`leave` Automáticamente desapila todo el marco de pila, excepto la dirección de retorno, y recupera anterior `%rbp` (equivalente a `%rsp=%rbp + pop %rbp`), útil justo antes de un `ret`.



# Repertorio de instrucciones x86-64

## • Tipos de instrucciones x86-64

### Variantes de instrucciones para 8, 16, 32 y 64 bits

- En realidad, hay *mnemónicos* (abreviatura con la que se hace referencia a cada instrucción) diferentes para distintos tamaños de datos (q=quad=64 bits; l=long=32 bits; w=word=16 bits; b=byte=8 bits), pero el lenguaje ensamblador permite el uso de un mnemónico más general.

- Por ejemplo:

`mov %rbx,%rax (general) = movq %rbx,%rax (64 bits)`

`mov %ebx,%eax (general) = movl %ebx,%eax (32 bits)`

`pop %ax (general) = popw %ax (16 bits)`

`add %bh,%ah (general) = addb %bh,%ah (8 bits)`

### Otras instrucciones

- Existen otros muchos tipos de instrucciones menos habituales, en los que no entraremos en detalle aquí:
  - Punto flotante
  - Llamada a interrupciones
  - Entrada/salida
  - Modo protegido
  - ...

# Índice

## **5.1 Introducción**

- 5.1.1 Programas e instrucciones
- 5.1.2 Codificación de las instrucciones
- 5.1.3 Tratamiento de las instrucciones
- 5.1.4 Tipos de instrucciones

## **5.2 Jerarquía de traducción**

- 5.2.1 Lenguajes de alto nivel
- 5.2.2 Compiladores y ensambladores
- 5.2.3 Código objeto
- 5.2.4 Librerías
- 5.2.5 Enlazadores y cargadores
- 5.2.6 Visión global de la jerarquía de traducciones

## **5.3 Introducción al ISA Intel x86-64**

- 5.3.1 Ensamblador del x86-64
- 5.3.2 Operandos de las instrucciones x86-64
- 5.3.3 Repertorio de instrucciones x86-64

## **5.4. Codificación de las instrucciones**

- 5.4.1 Formato y codificación de instrucciones
- 5.4.2 Ejemplos de codificación en x86-64
- 5.4.3 Reubicación de código
- 5.4.4 Espacio virtual de direccionamiento

# Formato y codificación de instrucciones

- Codificación de las instrucciones (1/2):
  - Cada instrucción ensamblador tiene una traducción directa en **lenguaje máquina** (como secuencia de 0's y 1's).
  - Las instrucciones en lenguaje máquina son las que realmente la CPU lee de memoria, decodifica y ejecuta.
  - **Formato de instrucción:** cómo se distribuyen los bits de la instrucción para almacenar
    - Código operación.
    - Operandos fuente y destino.
    - Posibles constantes (inmediatos que puedan haber).
  - Querríamos un único formato de instrucción, pero es imposible, hay muy diversos tipos de instrucciones con distintas necesidades:
    - Operandos de naturaleza variable (constantes, registros, memoria, ...)
    - Saltos necesitan codificar dirección destino
    - Etc.
  - En x86-64, instrucciones de longitud variable (CISC).
  - En 2º cuatrimestre se estudiará ISA MIPS32 (RISC), con instrucciones de longitud fija (32 bits): mucho más regular.

# Ejemplos de codificación en x86-64

## • Codificación de las instrucciones (2/2)

– Trozo de ejemplo de `main.o`:

**3. Códigos de operación (Cop)**

**6. Valores inmediatos**

**7. Codificación de saltos (relativa a PC instrucción siguiente):**

PC sig. =  $0x11 = 17$

Dir. Salto =  $0x33 = 51$

Despl. relativo =  $51 - 17 = 34 = 0x22$

**4. A veces, en el propio Cop se codifica la instrucción y el registro involucrado...:**

`push %rbp = 0x55`

**5. ... otras veces, el código de operación es más largo, y más registros involucrados hacen que haya que usar más bytes:**

`mov %rsp,%rbp = 0x4889e5`

**1. Desplazamientos relativos al comienzo**

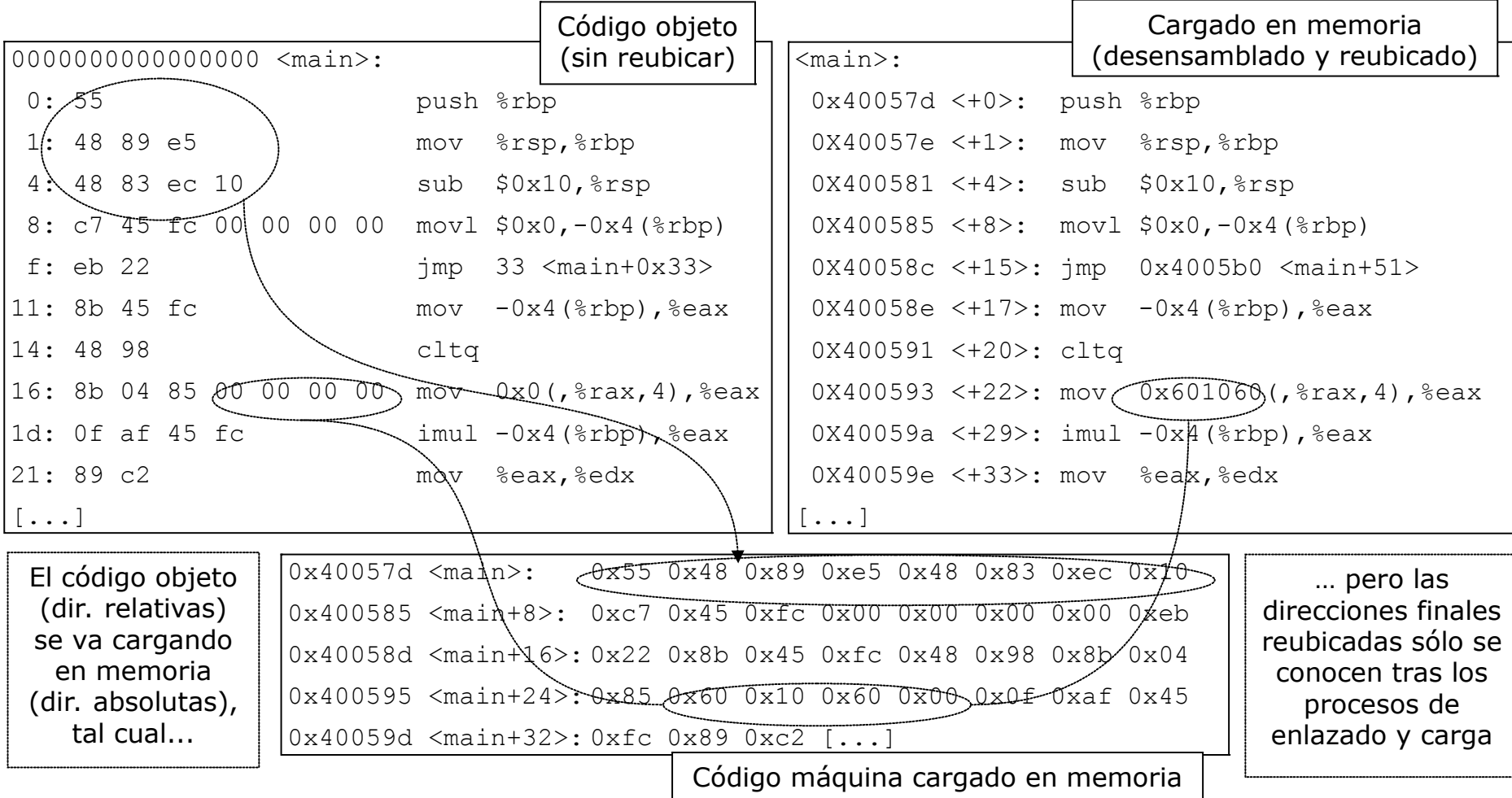
**2. Longitud variable de instrucciones (desde 1 a 7 bytes en el ejemplo)**

```

0000000000000000 <main>:
0: 55                                push    %rbp
1: 48 89 e5                         mov     %rsp,%rbp
4: 48 83 ec 10                      sub     $0x10,%rsp
8: c7 45 fc 00 00 00 00            movl    $0x0,-0x4(%rbp)
f: eb 22                          jmp     33 <main+0x33>
11: 8b 45 fc                        mov     -0x4(%rbp),%eax
14: 48 98                          cltq
16: 8b 04 85 00 00 00 00            mov     0x0(,%rax,4),%eax
1d: 0f af 45 fc                    imul    -0x4(%rbp),%eax
21: 89 c2                          mov     %eax,%edx
[...]
```

# Reubicación de código

- Programa cargado en memoria
  - Ya enlazados los objetos y las librerías, en el programa cargado ya no hay referencias sin resolver:



# Espacio virtual de direccionamiento

- **Memoria virtual:**

- Permite la compartición eficiente y sin peligros de memoria entre múltiples programas:
  - Varios programas se están ejecutando al mismo tiempo, en espacios de direcciones independientes
- Hay que asegurar que un programa sólo pueda leer y escribir las partes de la memoria que tiene asignadas
- Los programas manejan un espacio de direcciones de una memoria virtual como si se tratase de una gran memoria principal para ellos solos

- Solución: **traducción de direcciones**

- CPU genera direcciones virtuales
- Unidad de manejo de memoria (MMU) dentro de la CPU las traduce a direcciones físicas (de forma transparente al programa)
- A la memoria se accede finalmente con direcciones físicas
  - El programador ya desconoce completamente las direcciones físicas, y nada le importan...
  - ... porque el SO es el encargado de mantener las estructuras necesarias para la traducción, y permitir la multiprogramación