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

Fundamentos de Computadores

Curso 2019-20

Índice

Índice

Programas e instrucciones

- Instrucción ≡ Conjunto de símbolos que representa una orden de operación o tratamiento para el computador.
- Programa ≡ Conjunto ordenado de instrucciones que debe ejecutar el computador sobre los datos para procesarlos y obtener un resultado.

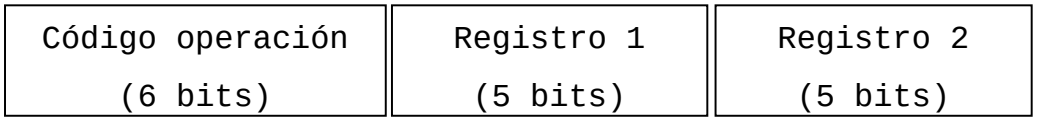
- Las instrucciones se almacenan en memoria principal en un orden determinado, y se van ejecutando en secuencia
- La secuencia sólo se rompe por posibles instrucciones de salto (bucles, condiciones if, saltos a funciones, etc.)

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

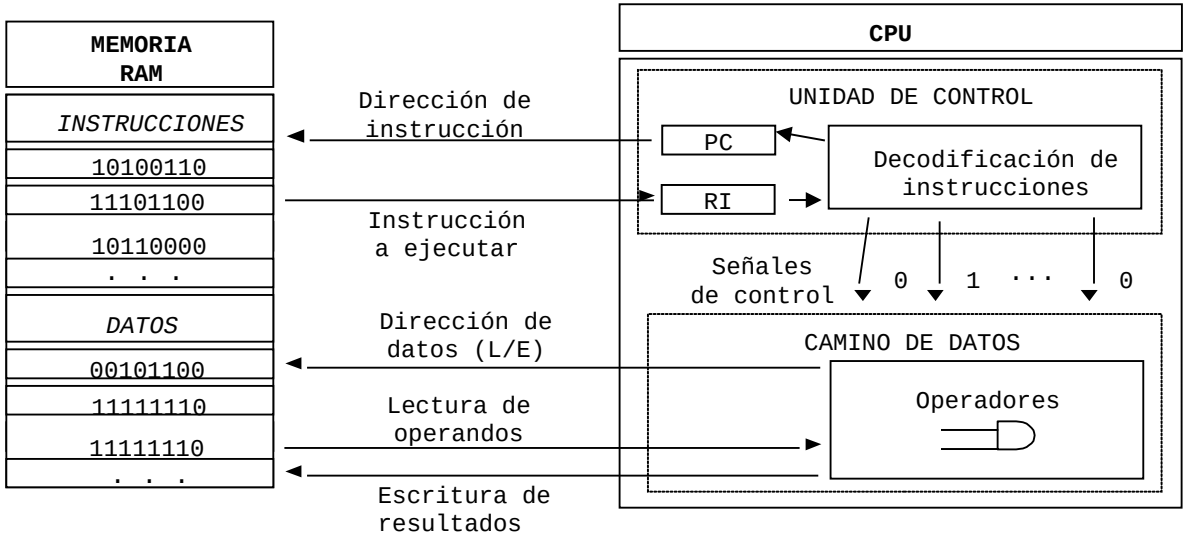
- Para codificar toda la información necesaria, las instrucciones se organizan en campos de bits.
- P.e., para una instrucción de suma acumulativa de un registro sobre otro, el formato podría ser:



- 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

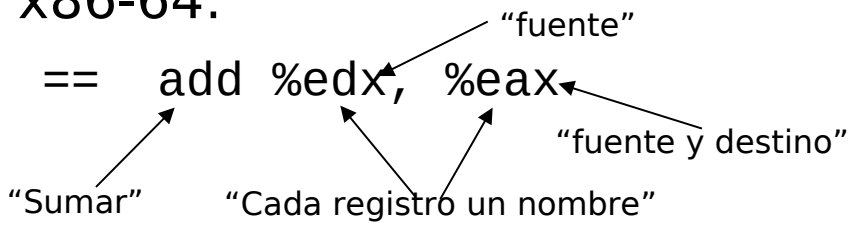
- A mayor número de instrucciones:
  - Más complejidad de la UC.
  - Mayor número de bits requeridos por el campo código de operación.
- Dos tendencias a este respecto:
  - RISC (*Reduced Instruction Set Computers*): pocas instrucciones, sencillas y se ejecutan en pocos ciclos.
  - CISC (*Complex Instruction 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

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
```



  - 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.

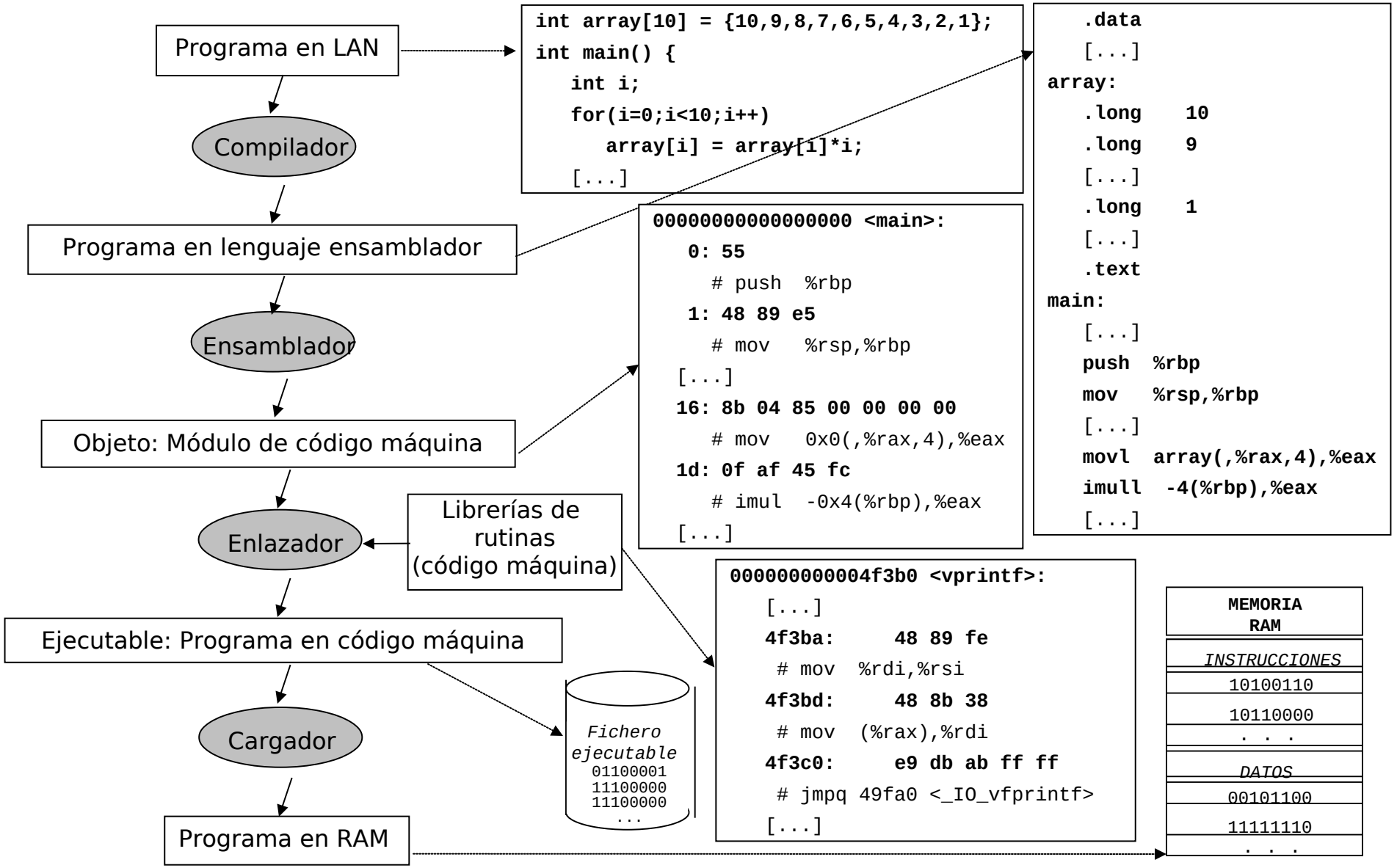
Librerías

- 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

Repertorios de instrucciones (ISA)

- Cada CPU posee su propio **repertorio de instrucciones,** más conocido como **ISA** (*Instruction Set Architecture*):
  - El más extendido es el Intel x86 (procesadores de Intel y AMD, tipo CISC)
  - En la asignatura de *Estructura y Tecnología de Computadores* (2º cuatrimestre de 1º) estudiaremos el ISA de MIPS (tipo RISC) y aprenderemos a programar con él
  - Otros ISA: ARM, Power (IBM), SPARC (Oracle), RISC-V (UC Berkeley)
- Cada instrucción ensamblador representa una operación elemental realizable directamente por la CPU
  - Aunque las CPUs x86 actuales internamente traducen las instrucciones CISC complejas a otras micro-instrucciones más sencillas (tipo RISC)
- Objetivos de un ISA
  - Facilitar el diseño del procesador y del compilador
  - Maximizar el rendimiento y minimizar el coste
  - 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 para mantener la cuota de mercado.
- En esta asignatura: aspectos básicos del ISA x86-64
  - Presente en la gran mayoría de PCs, desde portátiles hasta servidores

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

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

     .text
[... Segmento de texto ...]
main: [...codigo inicio...]
     jmp   .L2
      [... Sigue ...]
```

```
.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 ...]
```

```
.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
```

Ensamblador del x86-64

- Las **variables globales** del programa (p.e. `array`) están almacenadas 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.

```
// VARIABLES GLOBALES DEL PROGRAMA EN C
int array[10] = {10,9,8,7,6,5,4,3,2,1};
```

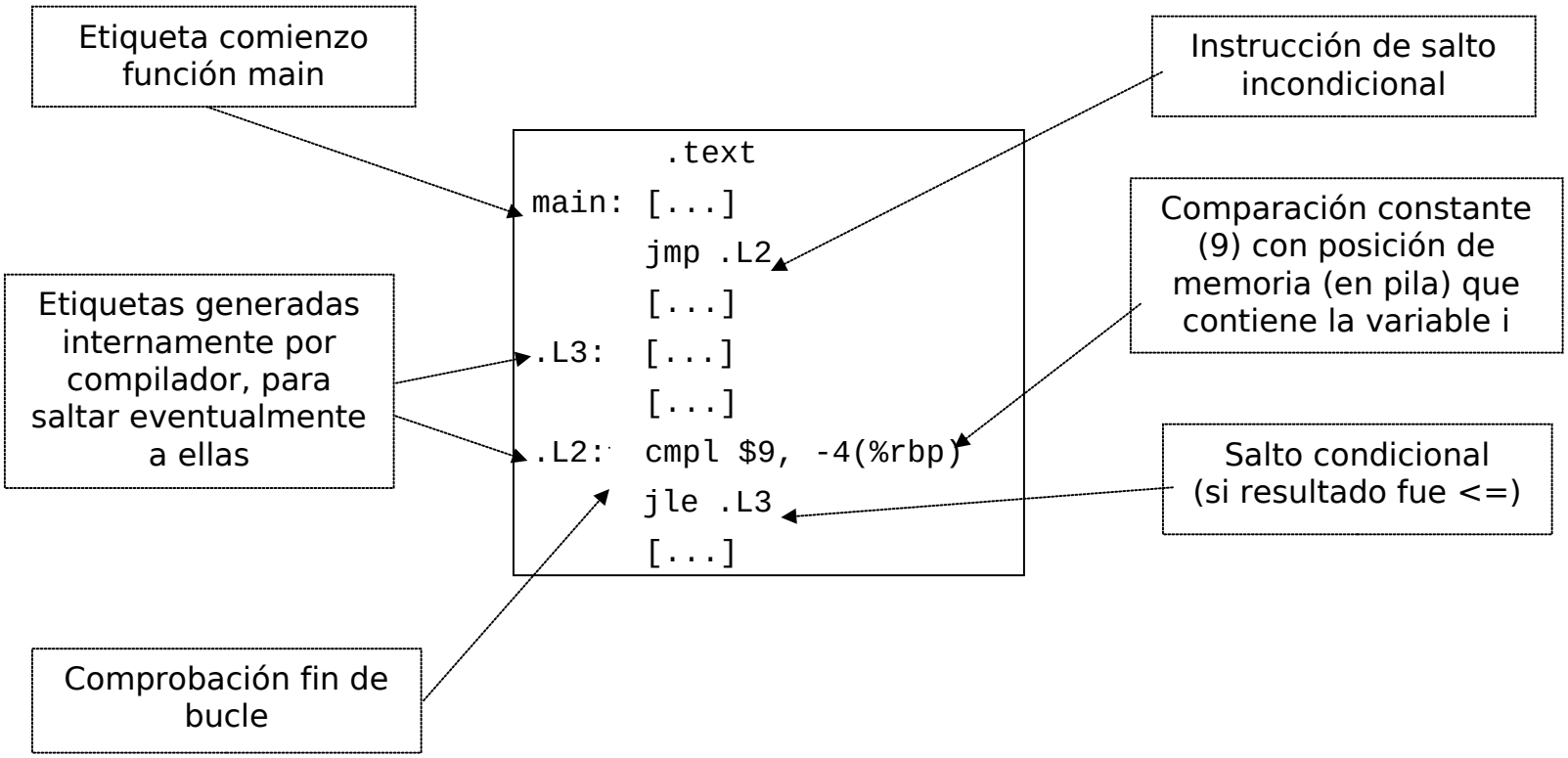
```
        .data
[...  Segmento datos   ...]
array: .long    10
       .long    9
       [...]
       .long    1
.LC0:  .string  "%d "
```

`.long`: 4 bytes en memoria

- La **variable** `i` es **local** a la función `main()`, y no se almacena en el segmento de datos sino en una zona de la memoria llamada ***pila***.
- Las **etiquetas** (identificadores seguidos de :) *representan* direcciones de memoria (aún no están instanciadas a direcciones concretas):
  - Algunas vienen del propio código fuente en C (p.e. `array:`)
  - Otras son creadas automáticamente por el compilador (p.e. `.LC0:`).
  - También en el segmento de código (no sólo en los datos)

Ensamblador del x86-64

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



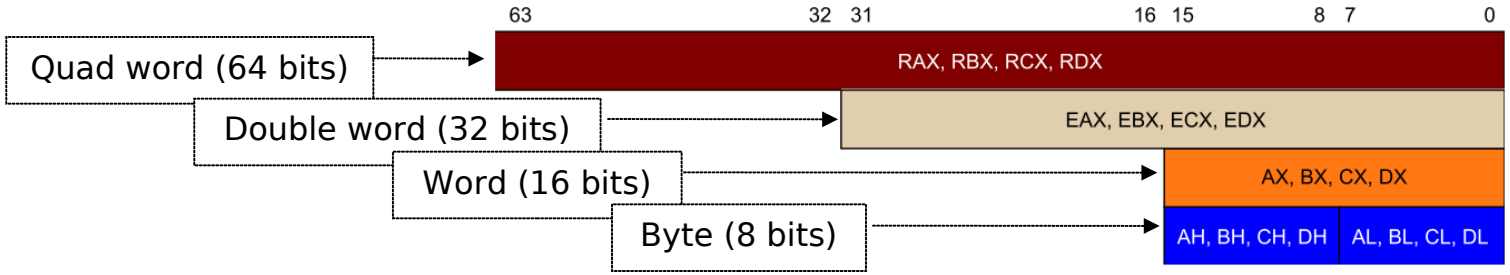
Operandos de las instrucciones x86-64

- **Registros**:
  - Contienen valores intermedios de nuestros cálculos
  - Son de acceso muy rápido, puesto que están en la propia CPU
    - Ejemplo: Copiar el valor de un registro a otro: `mov %rax, %rsi`
- **Memoria**:
  - Tabla formada por celdas, cada una de tamaño 1 byte
  - Cada celda tiene asociada una **dirección de memoria**
    - Un número que se usa para referirse a dicha celda para leer/escribir su **contenido**
  - Acceso más lento que a los registros (fuera de la CPU)
  - **Etiquetas**: representaciones *simbólicas* de direcciones de memoria en lenguaje ensamblador
    - Segmento de datos: nombres variables globales (p.e. “array”), etc.
    - Segmento de código: nombres de procedimientos (p.e. “main”), destinos de saltos, etc.
- **Constantes (operandos "*inmediatos*")**:
  - Ciertas instrucciones utilizan valores constantes
    - Ejemplo: Establecer el contenido del registro RAX con el valor 1234: `mov $1234, %rax`
  - Disponibles en la propia codificación de la instrucción, no en el segmento de datos ni en ningún registro

    La instrucción en ensamblador `sub $0x10,%rsp` en lenguaje máquina: `48 83 ec 10`

Registros en x86-64

- Los registros enteros son de 64 bits
  - Puntero de instrucción: **RIP** (indica por dónde va ejecutándose el programa)
  - Uso general: **RAX**, **RBX**, **RCX**, **RDX**, **R8-R15**
  - Índices (útiles para acceder a posiciones de un array, p.e): **RSI**, **RDI**
  - Manejo de la pila: **RSP** (puntero de pila), **RBP** (puntero base de pila)
  - 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)
- Nombres alternativos para operar con menos de 64 bits (32, 16 u 8)
  - RAX=64 bits, EAX = 32 bits inferiores de RAX, AX = 16 bits inferiores de EAX..
  - Ídem para el resto de registros
- Existen otros registros para cálculos en punto flotante y vectoriales
  - Hasta 512 bits por registro
- Al igual que con las etiquetas, los nombres de registros son una forma de referirse simbólicamente a un número de registro en el procesador
  - » `add %rbx,%rax` → `48 01 c3`
  - » `add %rcx,%rax` → `48 01 c1`



Ensamblador del x86-64: enteros vs punto flotante

```
float array[10] = {10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0};
int main() {
    int i;
    for(i=0;i<10;i++)
      array[i] = array[i]*(float)i;
[... Sigue ...]
```

```
        .data
[...  Segmento datos   ...]
array:  .long    1092616192
        .long    1091567616
        [...]
        .long    1065353216
.LC0:   .string  "%d "
```

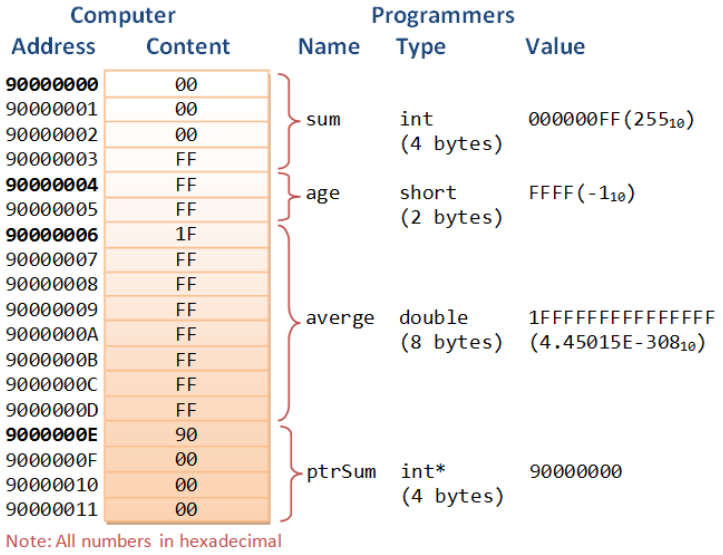
4 bytes en memoria: valor inicial 1092616192 = 10 en IEEE 754 simple precisión

```
      .text
[... Segmento código ...]
.L3:  [...Bucle sobre array...]
      movl  -4(%rbp), %eax
      cltq
      movss array(,%rax,4), %xmm1
      pxor  %xmm0, %xmm0
      cvtsi2ss   -4(%rbp), %xmm0
      mulss %xmm1, %xmm0
      movl  -4(%rbp), %eax
      cltq
      movss %xmm0, array(,%rax,4)
      addl  $1, -4(%rbp)
.L2:
        [... Sigue ...]
```

Multiplicación escalar en punto flotante de simple precisión (xmmN: registros de punto flotante de 128 bits)

Direcciones de memoria en x86-64

- El ISA x86-64 define direcciones *virtuales* de 64 bits
  - Al final de este tema veremos la diferencia entre direcciones *virtuales* y direcciones *físicas*
- Tamaño máximo teórico de la memoria: 2^{64} = 4 Exabytes
  - Los procesadores actuales que implementan x86-64 usan direcciones de 48 bits (256 TB): El hardware ignora los 16 bits de más peso de cada dirección (por ahora). (¿Por qué?)
- ¡Ojo! No confundir **dirección de memoria** de una celda (posición ocupa en la "tabla") y su **contenido** (valor que tiene en cada momento)
- ¿Cómo se sabe si una secuencia de 0's y 1's que hay en memoria representa un carácter, un entero, un real en punto flotante, etc.?
  - El compilador sabe el tipo de dato almacenado en cada dirección de memoria
  - Dependiendo del tipo de dato, genera unas instrucciones u otras para manejar dichos datos



Fuente: https://www.ntu.edu.sg/home/ehchua/programming/cpp/images/MemoryAddressContent.png

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**, **xor**, **manipulación de bits**, 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 pueden operar con constantes (siempre precedidas por $):
      ```
      sub $1234, %rax    # Resta 1234 a RAX, y deja el resultado en RAX
      ```
    - Incrementos y decrementos: inc, dec
      ```
      inc %rax          # Incrementa en uno RAX
      ```
    - Desplazamiento de bits:
      ```
      shl $2, %rax       # Desplaza RAX dos bits a la izquierda
      ```
  - **2. Instrucciones de movimiento de datos**
    - Sirven para mover/copiar datos y/o constantes entre memoria y/o registros:
      ```
      mov %rbx, %rax    # RAX := RBX
      mov $1234, %rax   # RAX := 1234
      ```
    - Operandos de memoria:
      ```
      mov myvar, %rax    # RAX := variable "myvar" (variable global, en seg. datos)
      mov (%rbx), %rax   # RAX := Mem[RBX] (contenido de la dirección de memoria
                         # "apuntada" por RBX, nótense los paréntesis)
      mov -4(%rbp), %rax    # RAX := variable local en la pila (contenido de la
                            # dirección apuntada por RBP menos 4 posiciones)
      ```

Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **3. Saltos incondicionales**
    - Rompen el flujo secuencial de ejecución del programa (una instrucción tras otra)
    - Establecen el registro contador de programa (RIP en x86-64) a una dirección de código fija, indicada por una etiqueta (hacia atrás o adelante en el programa).
    - El programa sigue ejecutándose a partir de la instrucción destino del salto

```
        jmp .L1
        [...]
.L1:    mov %rax, %rbx
```

  - **4. Saltos condicionales**
    - Sólo saltan a la etiqueta si se cumple una determinada **condición:**
      `je` (si igual), `jne` (si no igual), `jg` (si mayor), `jge` (si mayor o igual)
    - Traducción de 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 modifica el **registro de flags** (RFLAGS), y que los saltos condicionales leen

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

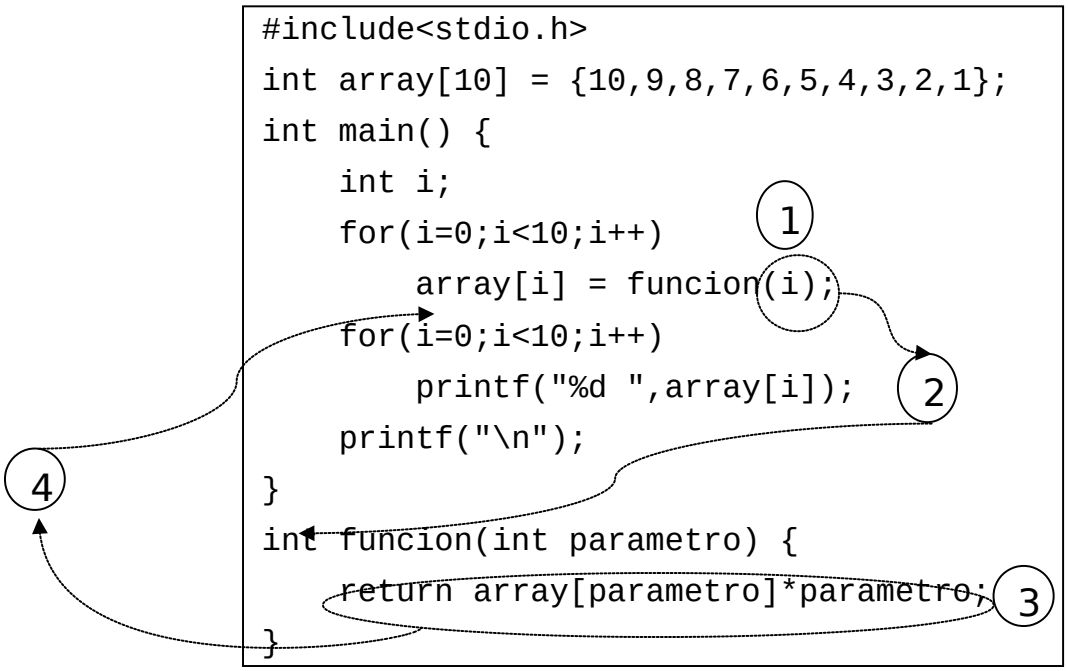
Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (1/6)**
    - 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 (2/6)**
    - 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

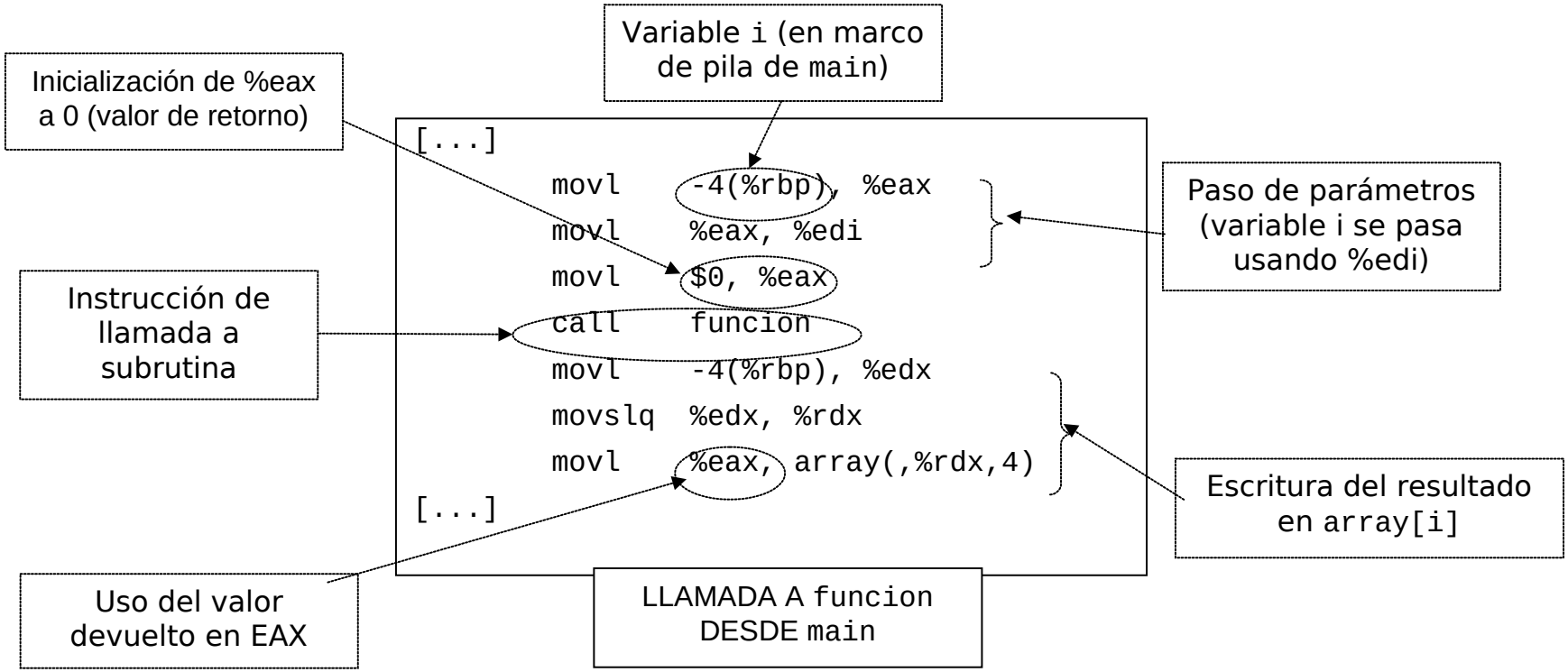
- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (3/6)**
  - El lenguaje ensamblador debe proporcionar:
    1. Instrucciones que permitan **cambiar el flujo del programa** (saltar al principio del procedimiento y regresar)

```
call misubrutina   # Guarda RIP (dirección de retorno) y salta a misubrutina
ret                # Regresa de la llamada al poner en RIP la dirección de
                   # retorno guardada por la instrucción call correspondiente
```

    2. Proporcionar una estructura de datos en memoria donde:
       a) Realizar el **paso de parámetros**
       b) Almacenar las **variables locales**
       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 registros se usan para 2 y 3, pero no son suficientes
    El número de parámetros y variables locales de un procedimiento (o de llamadas anidadas entre procedimientos) en un programa puede ser arbitrariamente grande, mientras que el número de registros del procesador es muy limitado.
  - Se necesita una estructura de datos en memoria para soportar procedimientos: la **pila**.

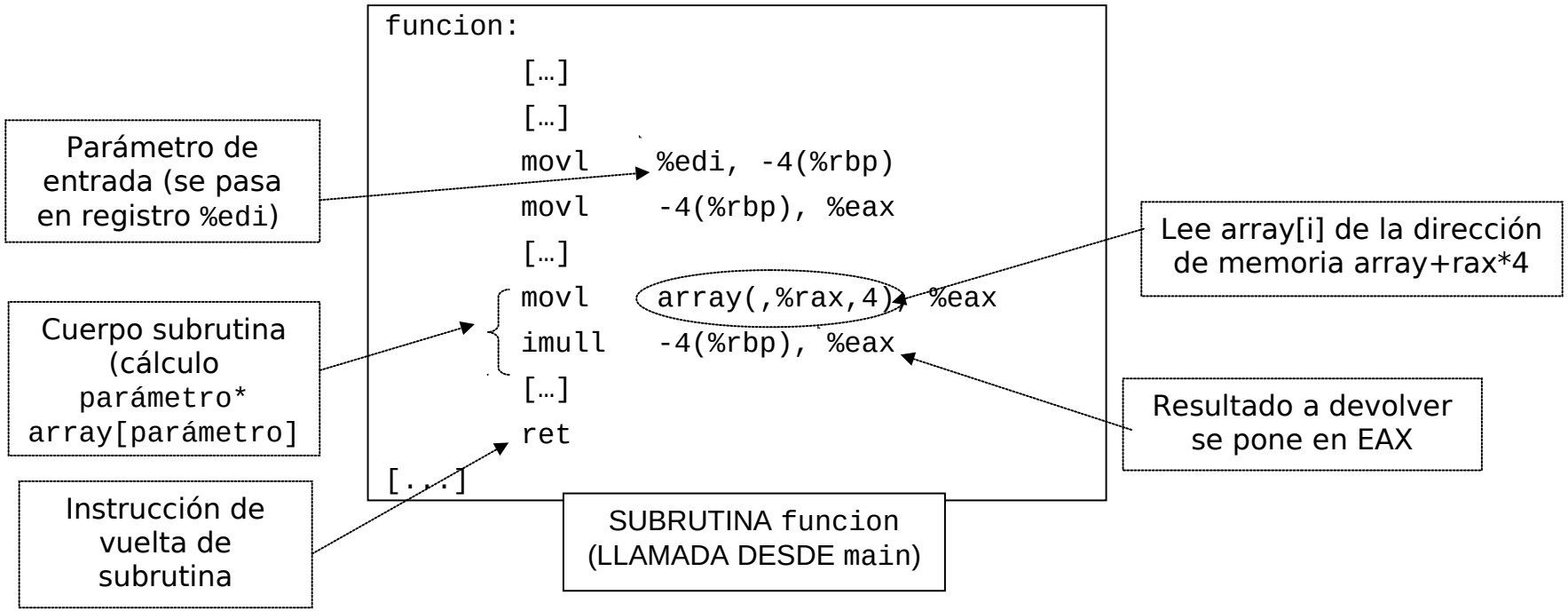
Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (4/6)**
    - 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 (5/6)**
    - Subrutina `funcion` llamada desde `main` (continuación)



Los registros `%rsp` y `%rbp` apuntan a la pila y se usan para acceder a las variables locales a la subrutina actualmente en ejecución, sus parámetros, la dirección de retorno, etc.

Repertorio de instrucciones x86-64

- Tipos de instrucciones x86-64
  - **5. Soporte para procedimientos (6/6)**
    - **Pila** = zona de memoria *normal* gestionada de una manera *especial*.
    - Empleada para:
      - Pasar posibles parámetros adicionales a subrutina (si registros insuficientes).
      - 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 retorno a la instrucción posterior a la llamada.
    - La pila es una estrutura **LIFO** (*Last In First Out*):
      - **Apilar** (**push**): guarda un registro o constante en la cima de la pila
      - **Desapilar** (**pop**): carga en un registro el contenido de la cima de la pila
    - Registro RSP (Stack Pointer): apunta (almacena) la dirección de la *cima* de la pila (último elemento válido). Modificado por las instrucciones `call, ret, push, pop`.
      La pila crece y decrece continuamente durante la ejecución del proceso, pero no puede crecer indefinidamente (*stack overflow*)
    - Registro RBP (*Base Pointer*): apunta a un lugar fijo de la pila durante la ejecución de cada rutina (al comienzo de su *marco de pila*), para poder direccionar a partir de él los posibles parámetros y/o variables locales.

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
  - ...

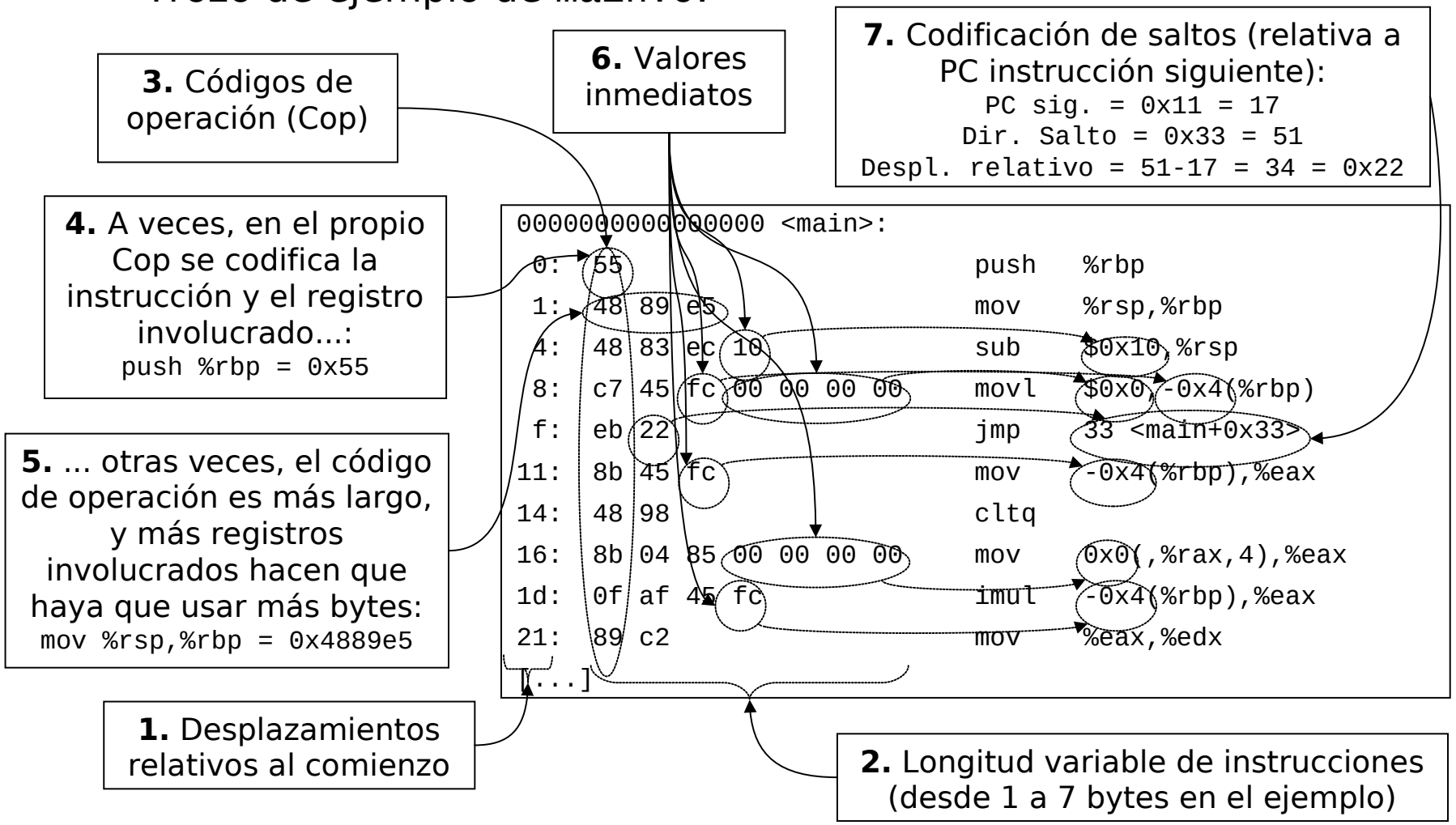
Índice

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`:



Reubicación de código

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

Código objeto (sin reubicar)

```
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
[...]
```

Cargado en memoria (desensamblado y reubicado)

```
<main>:
 0x40057d <+0>:  push %rbp
 0X40057e <+1>:  mov  %rsp,%rbp
 0X400581 <+4>:  sub  $0x10,%rsp
 0X400585 <+8>:  movl $0x0,-0x4(%rbp)
 0X40058c <+15>: jmp  0x4005b0 <main+51>
 0X40058e <+17>: mov  -0x4(%rbp),%eax
 0X400591 <+20>: cltq
 0X400593 <+22>: mov  0x601060(,%rax,4),%eax
 0X40059a <+29>: imul -0x4(%rbp),%eax
 0X40059e <+33>: mov  %eax,%edx
[...]
```

El código objeto (dir. relativas) se va cargando en memoria (dir. absolutas), tal cual...

```
0x40057d <main>:    0x55 0x48 0x89 0xe5 0x48 0x83 0xec 0x10
0x400585 <main+8>:  0xc7 0x45 0xfc 0x00 0x00 0x00 0x00 0xeb
0x40058d <main+16>: 0x22 0x8b 0x45 0xfc 0x48 0x98 0x8b 0x04
0x400595 <main+24>: 0x85 0x60 0x10 0x60 0x00 0x0f 0xaf 0x45
0x40059d <main+32>: 0xfc 0x89 0xc2 [...]
```

Código máquina cargado en memoria

... pero las direcciones finales reubicadas sólo se conocen tras los procesos de enlazado y carga

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