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

Fundamentos de Computadores

Curso 2022/23

Índice

Índice

Programas e instrucciones

- **Instrucción**: Conjunto de símbolos que representa una orden de operación para el computador
- **Programa**: Conjunto ordenado de ***instrucciones*** que debe ejecutar el computador sobre los ***datos*** para procesarlos y obtener un resultado
- Instrucciones:
  - Se almacenan en memoria principal en un orden determinado
  - Se van ejecutando en secuencia, una tras otra
  - Dicha secuencia sólo se rompe por posibles instrucciones de salto (bucles, sentencias "if", llamadas a funciones...)

Codificación de las instrucciones

- Cada instrucción indica una acción determinada a realizar por la CPU. Ejemplos:
  - Leer un valor de memoria y copiarlo a un registro de la CPU
  - Escribir en memoria el valor de un registro de la CPU
  - 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

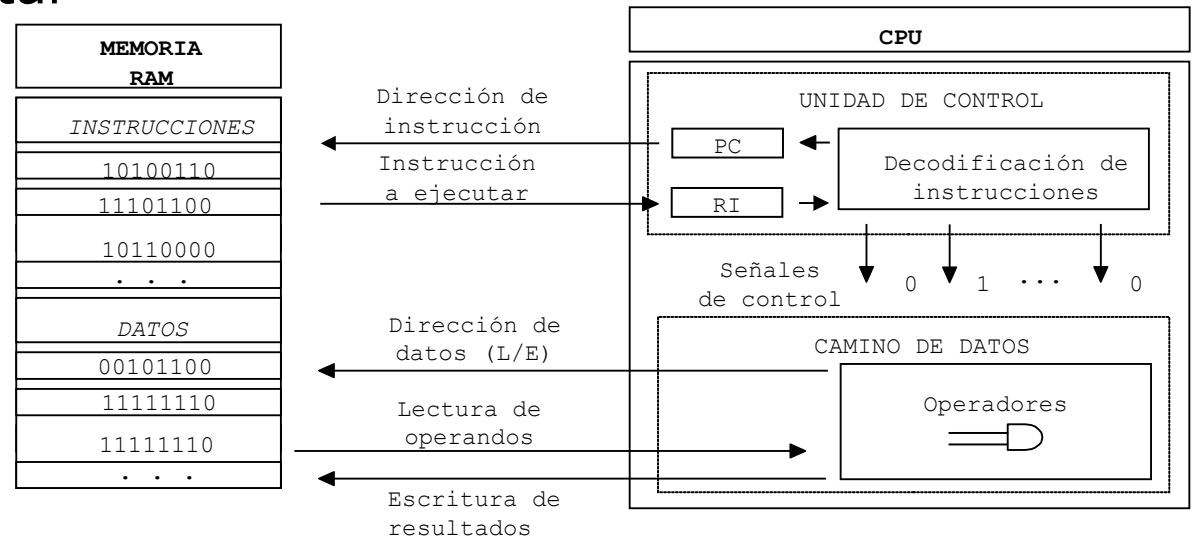
Codificación de las instrucciones

- Para codificar toda la información necesaria, las instrucciones se organizan en **campos de bits**
- Ejemplo: El formato de una instrucción de suma acumulativa de un registro sobre otro (R1 = R1 + R2) 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 (código de operación)
  - Los operandos de entrada (p.ej. registros R1 y R2)
  - El lugar en el que dejar el resultado (registro R1)
- 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

- La unidad de control (UC) de la CPU mantiene:
  - Contador de programa (PC): contiene la dirección de memoria 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



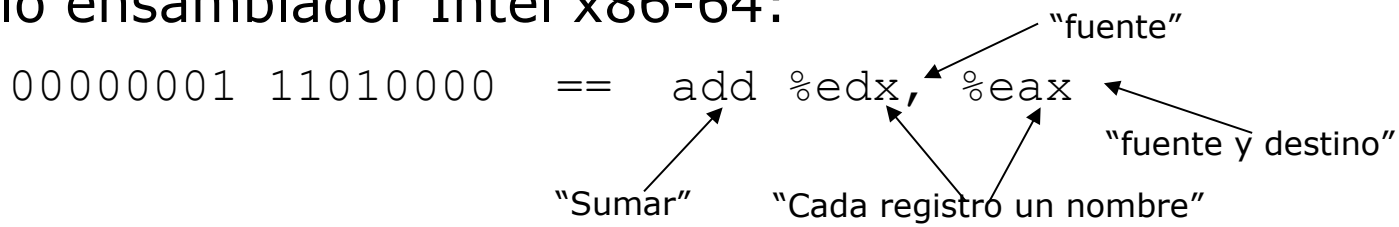
Tipos de instrucciones

- 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:



  - 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): Python, Java, C, C++

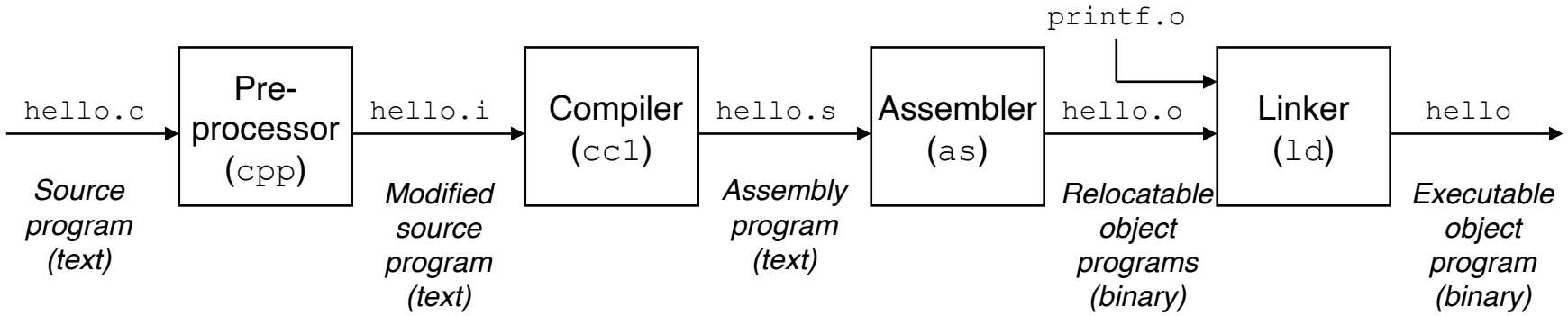
Lenguajes de alto nivel

- 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 (Python, C, C++, Java, Go, 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");
}
```

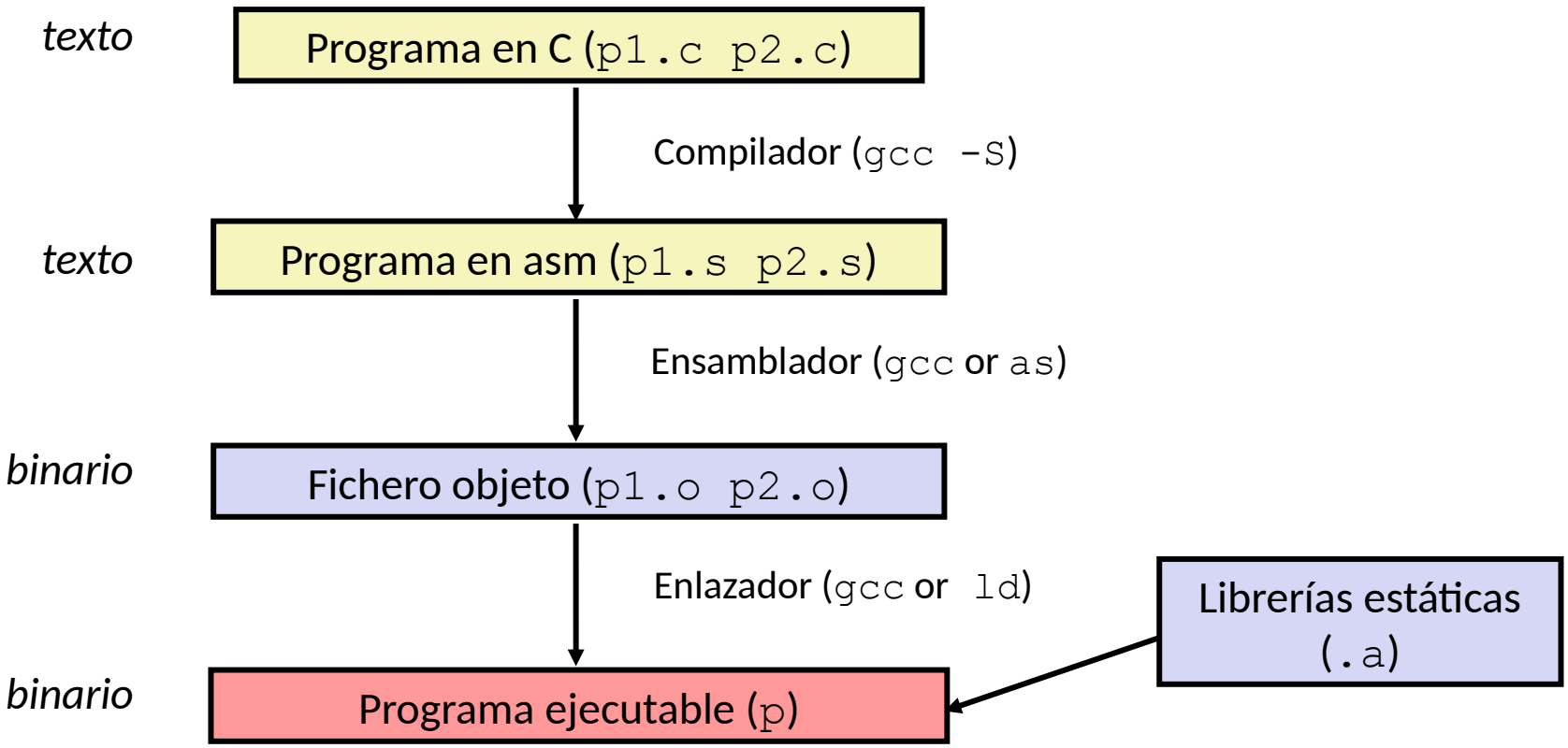
El sistema de compilación

- Los programas escritos en un lenguaje son traducidos por otros programas (**programas traductores**)
- Al **proceso general de traducción** se le suele llamar "compilación", e involucra las siguientes fases:
  1. Preprocesado: edición inicial con inserciones de código (#include, etc.)
  2. Compilación: traducción de lenguaje de alto nivel a lenguaje de bajo nivel (ensamblador de la máquina)
  3. Ensamblado: traducción de lenguaje ensamblador a código máquina
  4. Enlazado: unión de módulos de código objeto y resolución de llamadas a funciones de librerías



Compilación

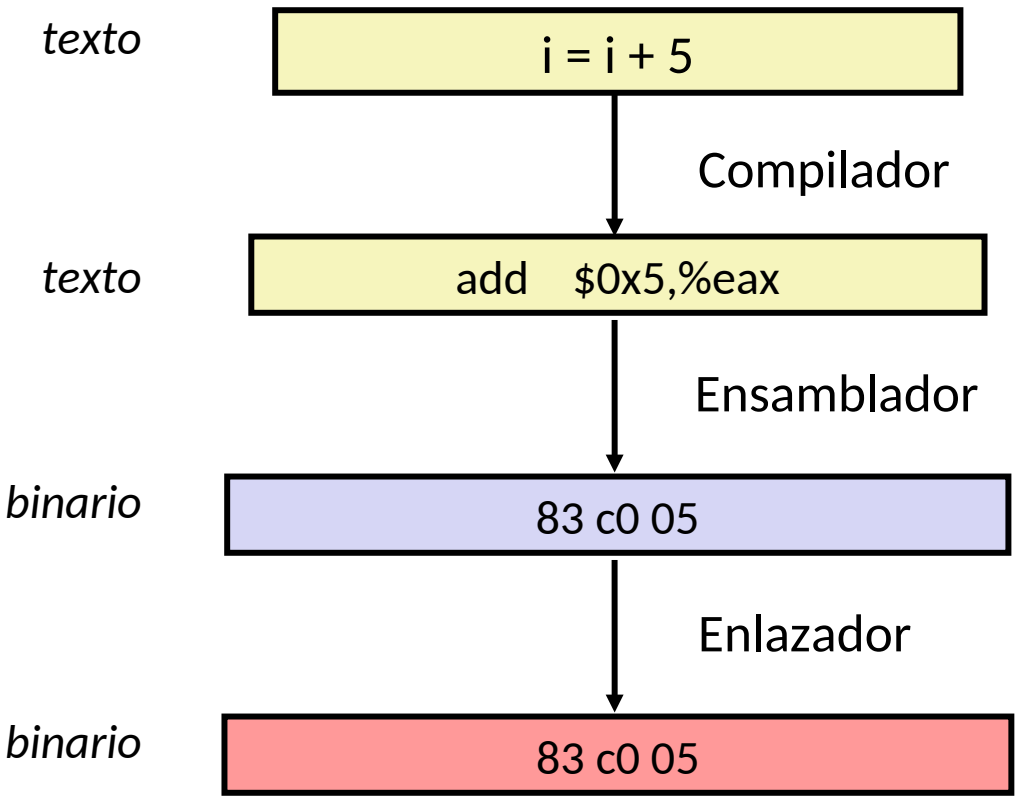
- Esquema general en el caso de C y sistemas UNIX:



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Compilación

- Ejemplo:



Programas traductores

- **Compilador**:
  - Transforma el código en lenguaje de alto nivel (en texto ASCII) a **lenguaje ensamblador**
    - Lenguaje ya "pegado" a la máquina, para un ISA concreto
    - Pero aún expresado mediante texto (legible)
    - Hace uso de nombres simbólicos para referirse a instrucciones, operandos, direcciones de memoria
    - Incluye directivas para indicar al ensamblador cómo producir el código
- **Ensamblador**:
  - Transforma el programa en lenguaje ensamblador en un *fichero de código objeto* en **lenguaje máquina**
    - Fichero de contenido binario (instrucciones y datos codificados)
    - Instrucciones ejecutables directamente por el procesador
    - Cada instrucción es una cadena binaria que contiene códigos de operación y operandos

Lenguaje ensamblador

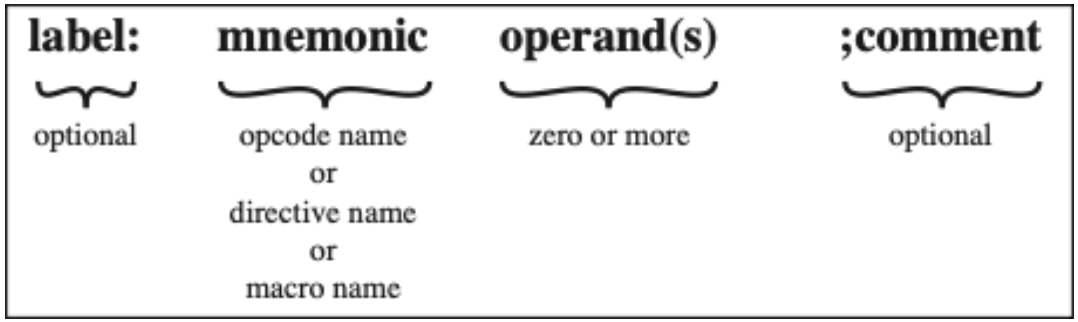
- Cada instrucción en ensamblador → una instrucción máquina

```
etiqueta:        addl    $5, %eax        # Comentario
```

- Dependiente de la arquitectura de la CPU (ISA)
- Las instrucciones en ensamblador hacen referencia a los **registros** de la CPU, **direcciones de memoria** y las distintas **operaciones** soportadas por la CPU -> el programador debe entender la arquitectura del computador
- Ventajas:
  - Depuración detallada
  - Control del hardware (sistemas embebidos)
  - Optimización del programa
- Desventajas:
  - Tiempo de desarrollo
  - Fiabilidad y seguridad
  - Depuración complicada
  - Mantenimiento del código difícil
  - Portabilidad limitada

Lenguaje ensamblador

- Las **etiquetas** se traducen en tiempo de ensamblado a la dirección de memoria donde se encuentra la instrucción
  - Pueden actuar como operandos en las instrucciones del programa
  - Se utilizan frecuentemente en las instrucciones de salto
  - Hacen más legible los programas y evitan que el programador tenga que calcular direcciones de memoria relativas o absolutas
- El **nemónico** es el nombre de la instrucción, que puede ser una instrucción de la máquina o una *pseudo-instrucción* (traducida por el ensamblador en instrucciones de la máquina)
- Las instrucciones incluyen cero o más **operandos**, que hacen referencia valores inmediatos, registros o direcciones



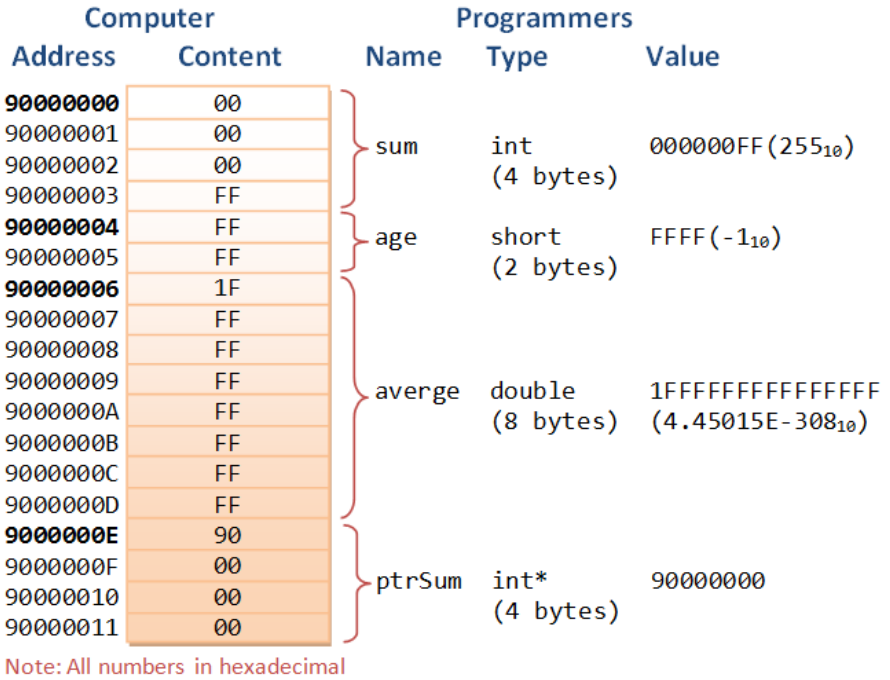
Operandos de las instrucciones en ensamblador

- **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 de 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`

Direcciones y contenido de la memoria

- ¡Ojo! No confundir dirección de memoria de una celda (posición que 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

Código objeto

- 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 el 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
- **Tipos** de ficheros objeto:
  - Fichero objeto reubicable (*relocatable object file*)
  - Fichero ejecutable
  - Fichero objeto compartido (ejemplo: bibliotecas dinámicas)
- Los compiladores y ensambladores generan ficheros objeto reubicables y/o compartidos
- Los enlazadores generan ficheros objeto ejecutables

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`)
  - 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

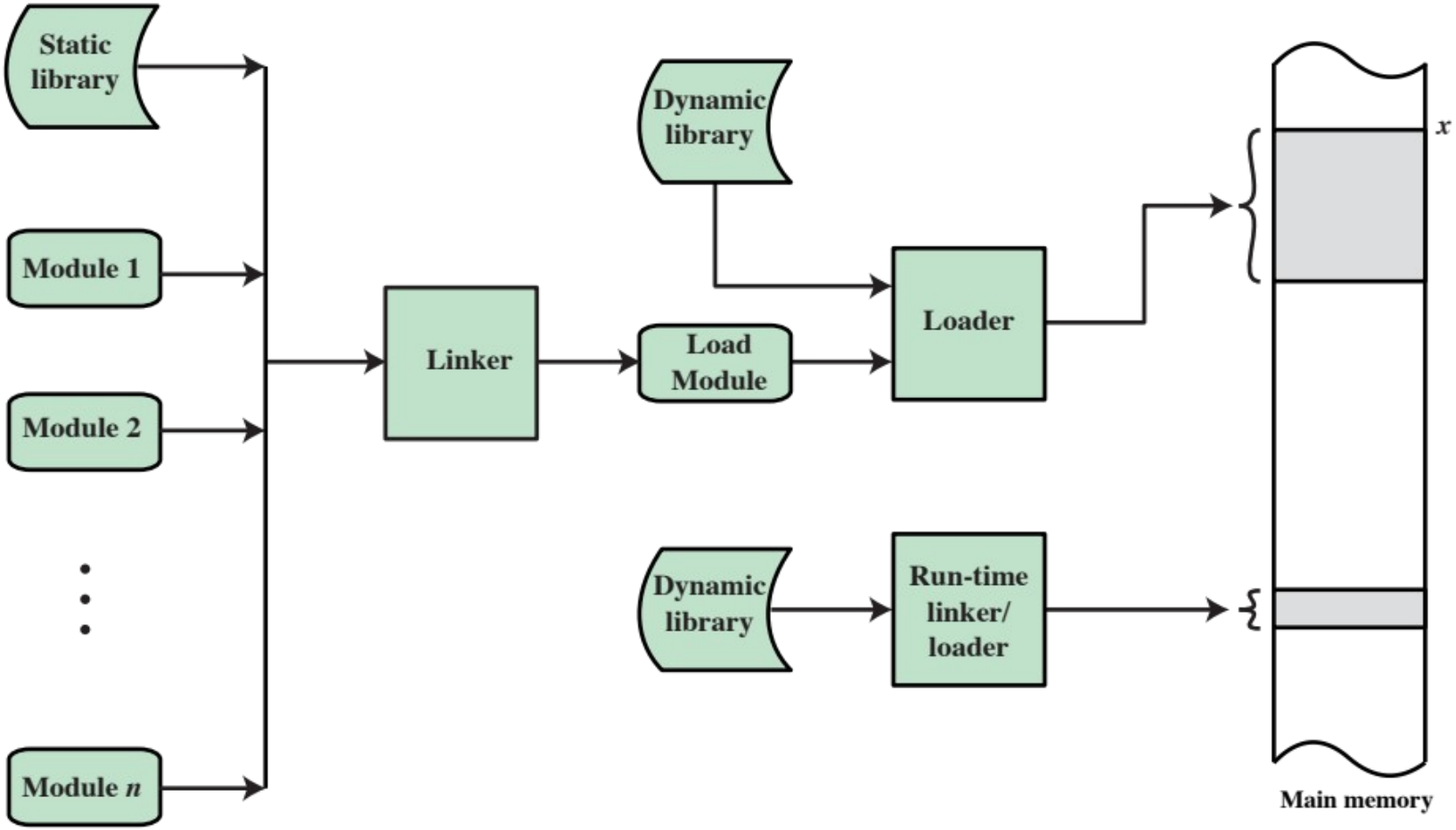
Librerías

- 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 el 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

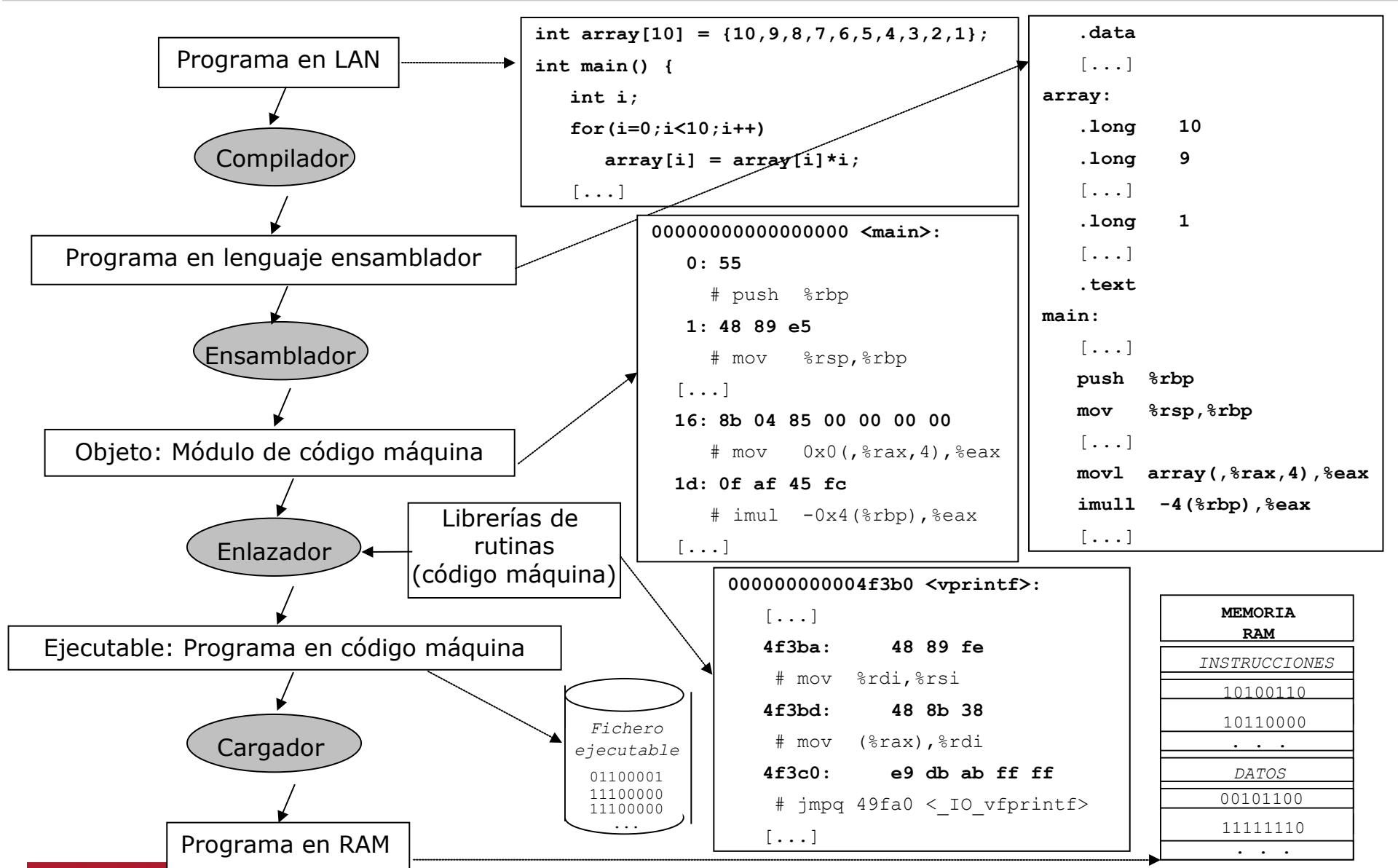
- Enlazador (*linker*):
  - Une los distintos objetos generados por el programador entre sí, ...
  - ... y también con las funciones de librería utilizadas, programadas por otros y preexistentes en el sistema (ya compiladas y listas para enlazar contra ellas)
    - Directorios `/lib`, `/usr/lib`, `/usr/local/lib/`, 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.

Enlazadores y cargadores



William Stallings, Organización y Diseño de Computadores, Séptima Edición

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)
  - 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

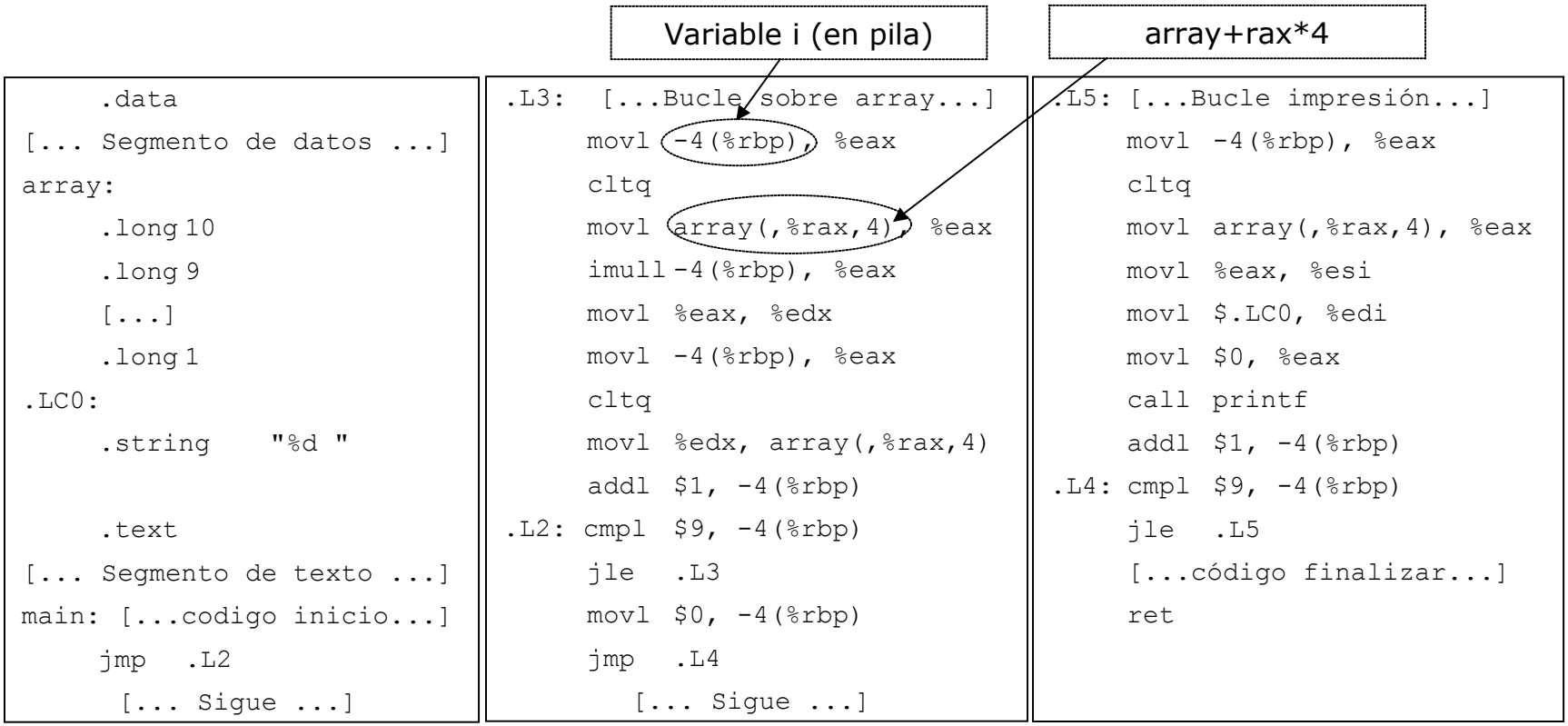
Ejemplo: Un programa en Lenguaje C

- El siguiente programa 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");
}
```

Traducción a ensamblador del x86-64

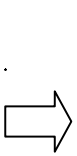
- Código ensamblador generado para el programa en C anterior
  - Recortado a lo que más nos interesa



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};
```



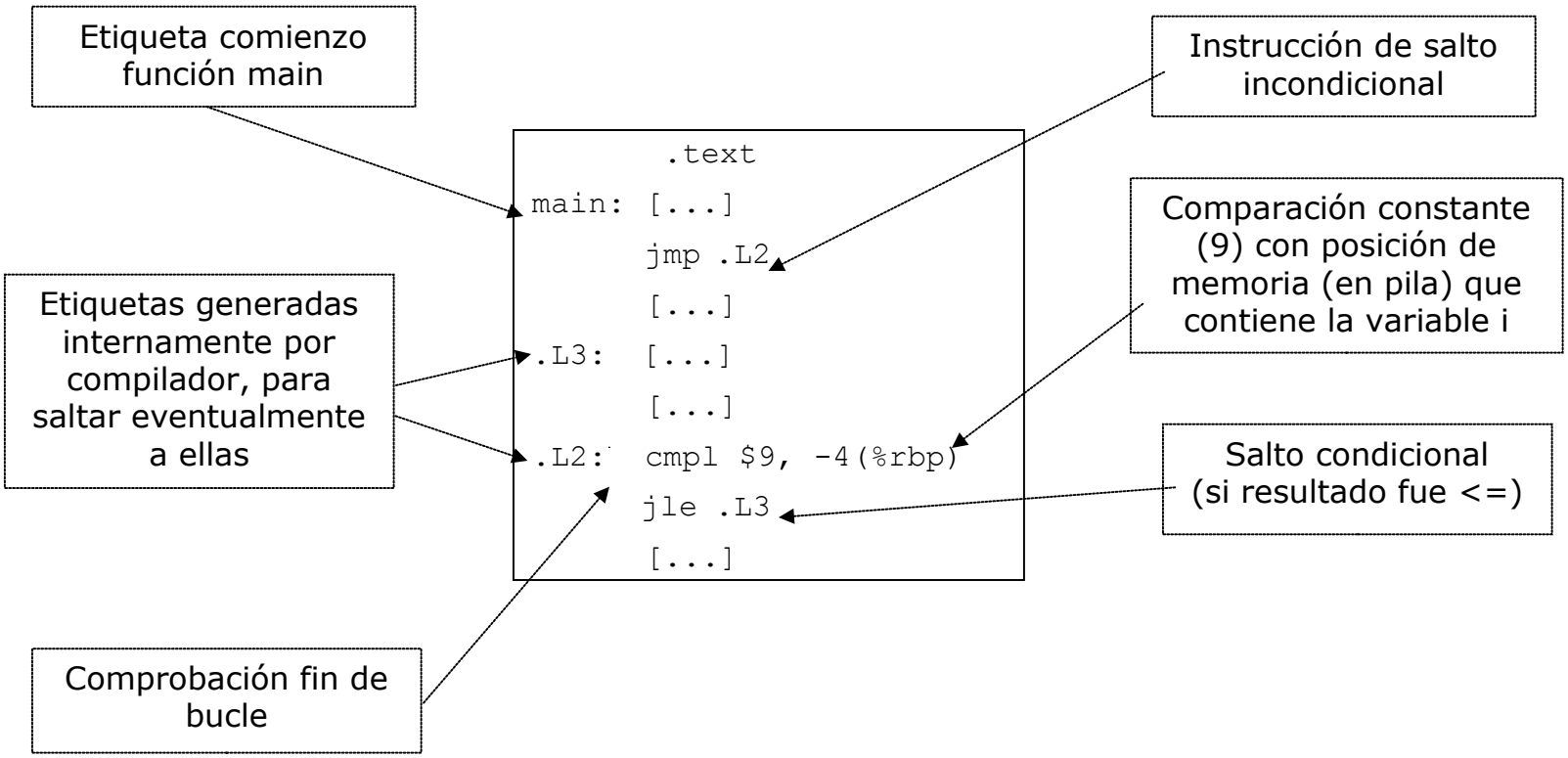
```
        .data
[...  Segmento datos    ...]
array: .long    10
       .long    9
       [...]
       .long    1
.LC0:  .string  "%d "
```

`.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

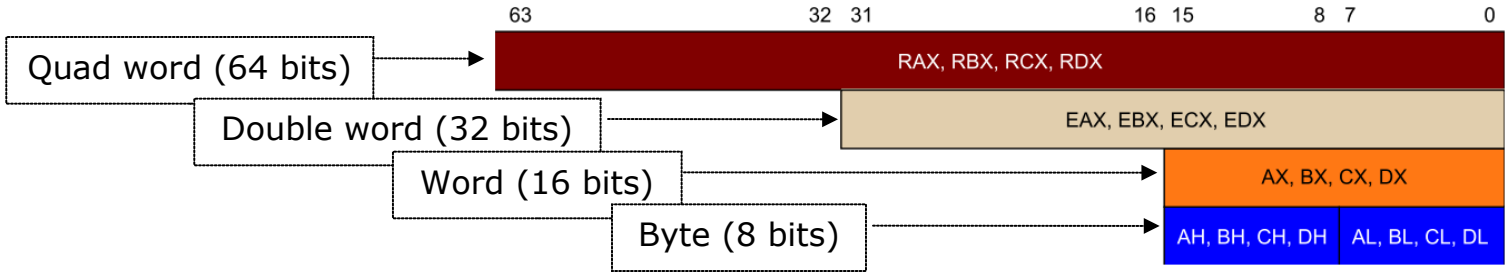
- El **segmento de código** (`.text`) contiene las instrucciones en ensamblador. Cada instrucción indica una operación elemental directamente realizable por la CPU.
- Por ejemplo, he aquí el código correspondiente a la comprobación de fin de bucle `for` (se ejecuta mientras `i<=9`):



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
```



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)
      ```

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. 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 pueden usar 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**.

Índice

Compilación vs Interpretación

- ¿Cuándo realizar la traducción del programa fuente a código máquina?
  - Con anterioridad a la ejecución del programa
  - Durante la propia ejecución del programa
- **Lenguajes compilados**
  - Ejemplos: C, C++, Erlang, Haskell, Rust, Go...
  - Usan compilación *Ahead-of-Time (AOT)*
    - El programa al completo se traduce a código máquina para una determinada CPU antes de poder ser ejecutado → Necesita un paso previo de "compilación".
  - Suelen ofrecer al desarrollador mayor control sobre aspectos del hardware: gestión de la memoria, uso de la CPU, etc.
  - Ventajas:
    - Ejecución de programas más rápida que el código interpretado
      - Al evitar el coste de traducir en tiempo de ejecución
  - Desventajas:
    - Tiempo necesario para la compilación en cada prueba (ciclo: modificar-compilar-probar)
    - Necesario recompilar el programa cada vez que hace un cambio
    - Dependencia de la plataforma del código binario generado

Compilación vs Interpretación

- **Lenguajes interpretados**
  - Ejemplos: Python, Java, JavaScript, PHP, Ruby...
  - Usan interpretación
    - El código fuente no es traducido directamente al código máquina de la CPU destino. En su lugar, un programa diferente, **el intérprete**, lee y ejecuta el código.
  - Los intérpretes recorren el programa línea por línea y ejecutan cada sentencia.
  - Ventajas
    - Código portable. Si el intérprete puede ejecutarse en muchos SOs distintos y tipos CPUs (como suele ser el caso), el software se ejecutará también en estos SO y CPUs.
    - La compilación a *bytecode* suele ser más rápida que la compilación a código máquina.
      - Puede hacer que todo el ciclo de desarrollo del software sea más rápido
  - Inconvenientes
    - Un programa interpretado es (a menudo) más lento de ejecutar que uno compilado
      - Pero esa diferencia se está reduciendo gracias a la compilación "justo a tiempo" (*just-in-time* o JIT)
    - Los usuarios tendrán que instalar una máquina virtual en su ordenador, y tendrán que actualizarla. Las máquinas virtuales también consumen recursos.
- También es posible un **enfoque híbrido**: Compilación + interpretación
  - Cualquier lenguaje puede implementarse con un compilador o con un intérprete.
  - Compilador traduce a código intermedio + intérprete lo ejecuta (Java, Python)

Una receta

- Imagina que tienes una receta que quieres preparar, pero está escrita en griego antiguo.
- Para una persona que no habla griego antiguo, hay dos formas de poder seguir sus instrucciones:
  - La primera es que alguien la haya traducido al español.
    - Cualquier persona que sepa hablar español (incluido tú) puede leer la versión española de la receta y preparar el plato
      - Esta receta traducida sería la versión “compilada”.
  - La segunda forma es si tienes un amigo que además de español sepa hablar griego antiguo.
    - Cuando estés listo para preparar el plato, tu amigo se sienta a tu lado y traduce la receta al español a medida que avanzas, línea por línea.
      - Tu amigo es el intérprete de la versión “interpretada” de la receta.

https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/

Compilación *ahead-of-time* (AOT)

- Traducción del código fuente de un programa con anterioridad a su ejecución
  - El código fuente se convierte al lenguaje máquina de la CPU destino (donde se desea ejecutarlo), obteniendo un fichero que contiene el programa ejecutable
  - Adapta y optimiza los programas **para una máquina destino concreta**
  - Compilación cruzada: compilación para una plataforma (un lenguaje máquina) distinta a la usada en la compilación
    - Ejemplo: ejecutar el compilador en una CPU x86-64 para generar código ARM
- Ventajas
  - El compilador puede optimizar el código máquina en función de la CPU haciendo que se ejecute más rápidamente.
  - No se necesita ninguna máquina virtual en el ordenador para ejecutar el programa
- Inconvenientes
  - Programas ejecutables dependientes de la plataforma (SO+CPU).
    - Para un mismo programa, diferentes ficheros ejecutables para cada combinación SO+CPU
  - Compilar a código máquina lleva más tiempo que compilar a *bytecode*
    - Cuanto más grande sea tu programa, más lenta será la compilación.

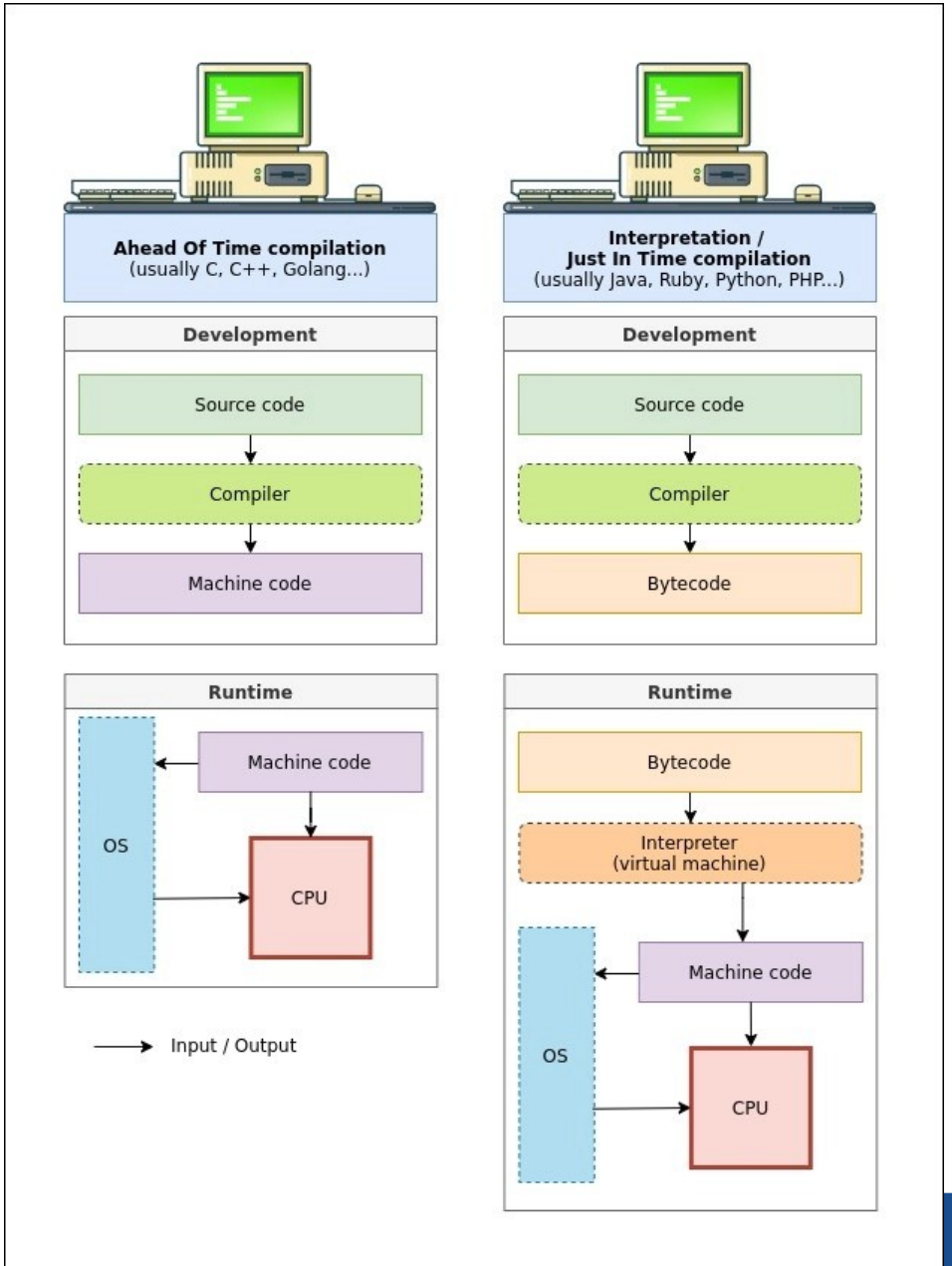
Interpretación

- El programa se ejecuta a partir de un código que no es el lenguaje de la máquina
  - Se puede interpretar el código a distintos niveles:
    - A partir del código fuente en un lenguaje de alto nivel
    - A partir de una traducción intermedia (*bytecode*) --> Enfoque híbrido
  - La interpretación necesita un **intérprete del lenguaje**
    - Parte del **entorno de ejecución** (*máquina virtual)* de dicho lenguaje
  - El propio intérprete puede generar representaciones intermedias del código
    - Código de bajo nivel, como **bytecode**
    - Estructuras de datos, como los árboles de sintaxis abstracta (AST)

Compilación *just-in-time* (JIT)

- También llamada traducción dinámica (*dynamic translation*).
- Combinación de dos enfoques: compilación AOT + interpretación
- No genera un ejecutable en código máquina con anterioridad a la ejecución
  - El programa se toma en un lenguaje determinado y se compila cuando se va a ejecutar
- La traducción al código máquina se realiza en el momento de la ejecución
- Puede partir de código fuente, aunque para mejorar el rendimiento se hace uso de una pre-compilación a un lenguaje intermedio (bytecode)
  - La compilación JIT traduce de bytecode a código máquina, que luego se ejecuta directamente sin necesidad de un intérprete
- El entorno de ejecución (VM) analiza el bytecode que se está ejecutando...
  - Identifica las partes del código que se ejecutan repetidamente
  - Indica al compilador JIT que traduzca dichas partes si resulta rentable
    - Siempre que la mejora obtenida por la compilación compense la sobrecarga de compilar ese código
- Puede aplicar optimizaciones, pero no son tan precisas como en AOT
- La interpretación de bytecode y la compilación JIT son muy similares
  - Interpretación de bytecode: traducción a código máquina y ejecución están entrelazadas
  - Compilación JIT de bytecode: primero se genera la traducción y después se ejecuta

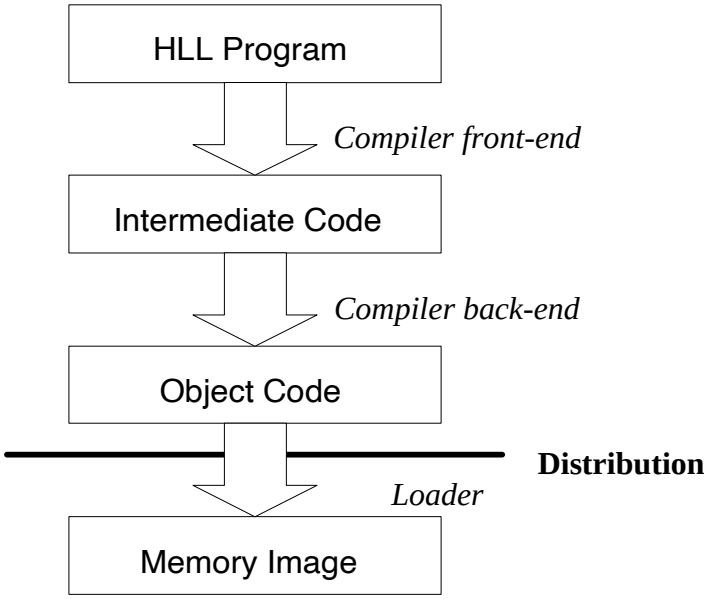
Compilación AOT vs Interpretación/JIT

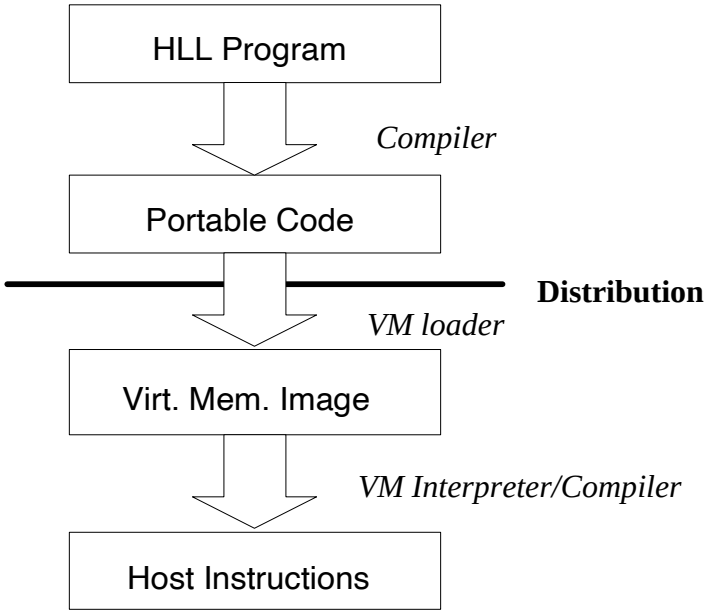


Bytecode

- **Representación intermedia** del código fuente, usada internamente por el compilador o máquina virtual
- Idea: Traducir el código fuente a una representación intermedia genérica + usar un intérprete o máquina virtual (VM) para cada plataforma (SO+CPU)
  - En vez de compilar el código fuente para distintas plataformas
    - P.ej. Linux/x86-64, Windows/x86-64, Android/ARM64
  - Necesaria una VM del lenguaje de alto nivel para cada plataforma
    - Java VM, Python VM, etc.
- Instrucciones: independientes del ISA de la CPU donde se ejecutará el programa
  - El bytecode es el ISA de la máquina virtual
- Objetivo principal: **Portabilidad** (independencia de la plataforma)
- Origen del nombre: usar 1 byte para codificar cada instrucción
  - En la actualidad esto puede variar según el lenguaje

Compilación AOT vs Interpretación/JIT





Fuente: J. E. Smith and R. Nair. Virtual Machines.

Bytecode

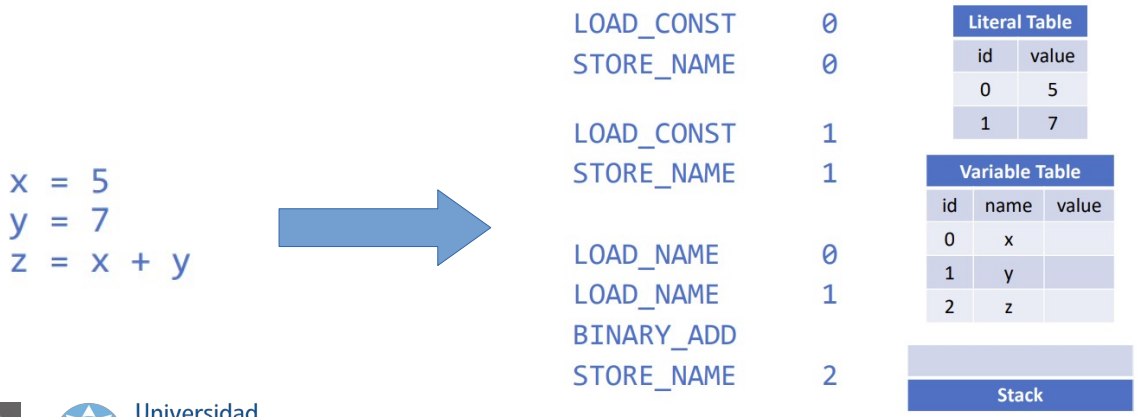
- ISA virtual que **abstrae los detalles de un procesador** concreto
  - La mayoría de VMs para interpretar bytecode son máquinas de pila
- Diseñado para su **ejecución eficiente mediante un intérprete** sw
  - Sintaxis de fácil interpretación. Reducción de la complejidad del lenguaje de alto nivel
  - El intérprete forma parte de la máquina virtual de un lenguaje de alto nivel (HLL VM)
- Instrucciones en bytecode: simples y estructuradas
  - Cada instrucción consta de:
    - Nombre del comando (*opcode)*
    - Un número como parámetro del comando (opcional, según la instrucción)
- Como el lenguaje intermedio es tan sencillo, se basa en componentes adicionales para funcionar:
  - **Tablas de valores**. Forman la memoria del programa.
    - Tabla de literales (constantes)
    - Tabla de variables
  - Una **máquina de pila**. Mantiene el estado del programa conforme se ejecuta. La pila almacena:
    - Los valores de entrada de los comandos
    - Los resultados de las operaciones

Ejemplo: Python Bytecode

- Ejemplo en Python:

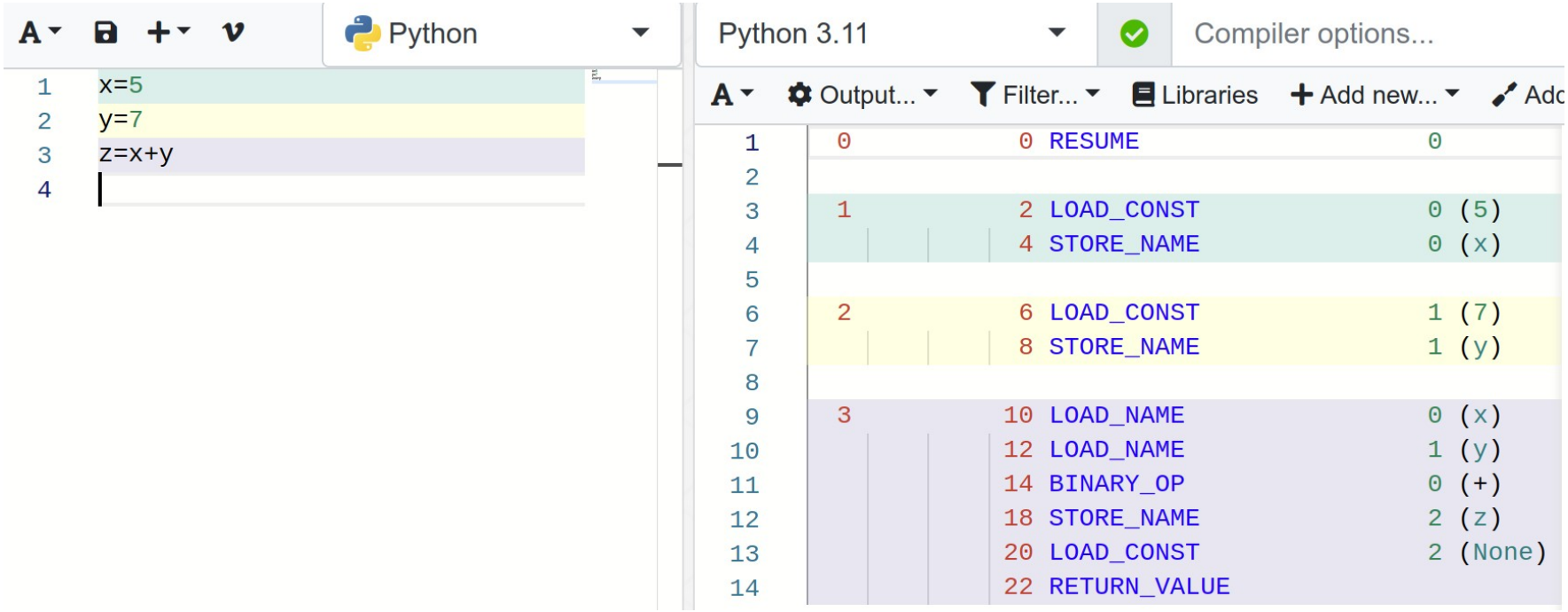
```
x = 5
y = 7
z = x + y
```

- Instrucciones en bytecode en las que se traduce:
  - LOAD_CONST y LOAD_NAME: se usan para mover información de la tabla de literales/variables a la pila
  - STORE_NAME se usa para mover información de la pila a la tabla de variables
  - BINARY_ADD suma los dos valores en la cima de la pila y los reemplaza con el resultado (desapila los dos operandos de entrada y apila el resultado)



Ejemplo: Python Bytecode

- Formato del bytecode de Python 3:
  - Cada instrucción es de 16 bits
  - 1 byte para el código de operación (157 actualmente)
  - 1 byte para el argumento

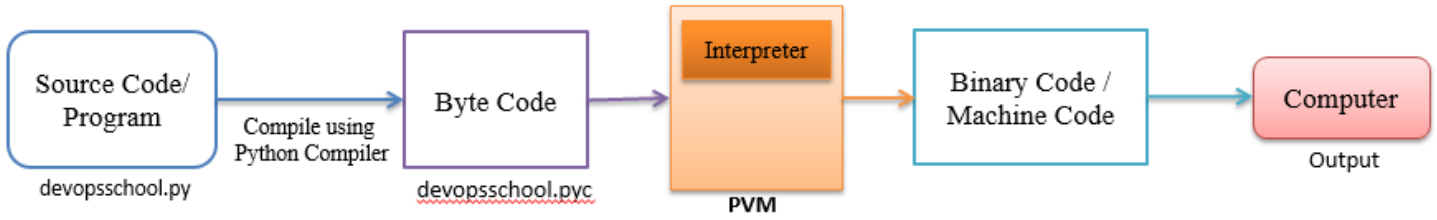


El lenguaje Python

- Lenguaje creado por Guido van Rossum a principios de los 90
- Nombre inspirado en los cómicos “Monty Python”
- Lenguaje interpretado, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos
- Traducción a código intermedio bytecode en la primera ejecución o cuando se indica explícitamente que se quiere realizar la compilación
  - .pyc
  - .pyo (bytecode optimizado)
- Lenguaje muy flexible, con el que se puede desarrollar con programación imperativa, funcional y orientada a objetos
- Ventajas que han ayudado a su expansión:
  - Sintaxis clara y sencilla (similar al pseudocódigo)
  - Tipado dinámico
  - Gestor de memoria eficiente y transparente al programador
  - Gran cantidad de librerías disponible
  - Potencia del lenguaje
- Ojo:
  - No es adecuado para programación de bajo nivel
  - Especial cuidado en aplicaciones con rendimiento crítico

Ejecución de programas en Python

- Dos **modos de trabajo**:
  - Sesión interactiva
    - Ejecución línea a línea a modo comandos de Shell
  - Intérprete de programa
    - Ejecución de programa en un archivo
    - En la primera ejecución se realiza la compilación a bytecode
    - Se puede forzar la traducción inicial para evitar el retardo en la primera ejecución
- Fases en la ejecución de un programa en Python:
  - 1. Inicialización
    - Reserva y copia de datos en las diferentes estructuras de datos necesarias para el proceso Python (solo aplicable al modo de trabajo no interactivo)
  - 2. Compilación
    - Análisis léxico y sintáctico, y generación de código en bytecode
  - 3. Interpretación
    - Ejecución de código bytecode



DevOpsSchool.com

Interpretación en Python

- La máquina virtual de Python (PVM) ejecuta las instrucciones en bytecode
  - **La PVM se ejecuta sobre el hardware** (ejecuta instrucciones máquina de la CPU destino)
  - Las instrucciones en bytecode son interpretadas por la PVM
- Fases de la interpretación en la PVM:
  1. Inicialización del intérprete y el estado de los hilos de ejecución
  2. Inicialización de zonas de memoria del programa a interpretar
  3. **Bucle de evaluación** de instrucciones en bytecode
     - Simplemente un bucle muy grande con una sentencia switch:

```
while (1) {
   bc = *ip++;
   switch (bc) {
     ...
     case 0x76:
       *++sp = ConstOne;
       break;
     ...
   }
}
```

- Ejemplo de instrucciones:
  - 0: Carga el valor de una variable “x” en la pila
  - 2: Carga el valor de una constante “1” en la pila
  - 4: Saca dos valores de la cima la pila, los suma, y mete el resultado en la pila
  - 6: Saca el valor donde apunta la pila y lo devuelve

```
def f(x):
    return x+3
```

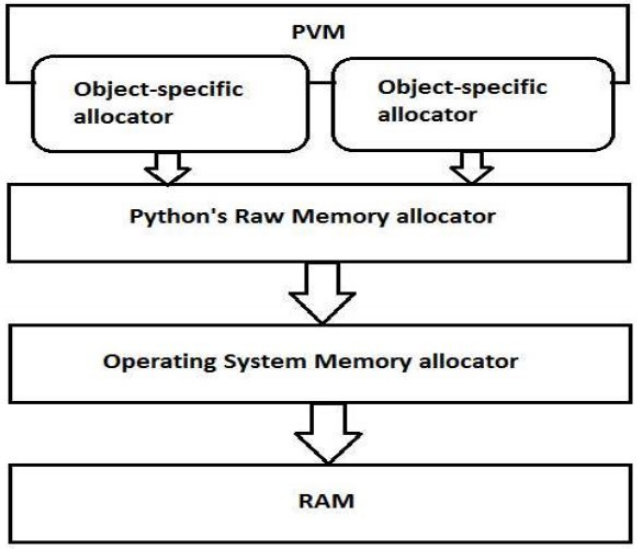
```
0    LOAD_FAST     0 (x)
2    LOAD_CONST    1 (3)
4    BINARY_OP     0 (+)
6    RETURN_VALUE
```

Implementaciones de Python

- Implementación del compilador e intérprete (PVM)
- Varias implementaciones disponibles:
  - CPython
    - La más extendida y la que se supone por defecto
    - Implementada en lenguaje C
    - Instalada por defecto en Linux y MacOS
  - Jython
    - Desarrollada en Java
    - Integración con librerías Java
  - IronPython
    - Desarrollada en C#
    - Integración con librerías .NET
  - PyPy
    - ¡Desarrollada en Python!

Gestión de la memoria en Python

- Reserva y liberación de memoria automática en tiempo de ejecución
- Almacén de datos en forma de objetos en la memoria montón de la PVM
- Recolector de basura (Garbage Collector)
  - Libera memoria cuando los objetos ya no son referenciados



Python Online Tutorial. GKIndex