Introducción al desarrollo de software

Román Ginés Martínez Ferrández

31 de agosto de 2022

Índice general

1 El ordenador: una máquina programable 1

2 Relación software-hardware 5

3 Lenguajes de programación: evolución histórica 7

4 Práctica 1: Introducción al desarrollo de software y lenguajes de programación 11

5 Lenguajes de alto nivel: clasificación 13

6 Fases en la obtención del código 17

7 Laboratorio 1: generación de código máquina 27

8 Laboratorio 2: generación de bytecode 31

1

El ordenador: una máquina programable

Es bien conocido que el ordenador se compone de dos partes bien diferenciadas: **software** y **hardware**.

En este módulo nos vamos a centrar en la parte del software, al final estás cursando un ciclo de programación, pero no podemos obviar el hardware en tanto que lo que vas a aprender a programar es una máquina, es el hardware. Así, voy a empezar por explicarte qué es un ordenador, a nivel de hardware, y lo voy a hacer entrando en la arquitectura de los ordenadores actuales.

1.1 Arquitectura von Neumann

Una arquitectura de ordenador es un modelo y una descripción del funcionamiento del mismo. Especial es el papel que juega en esta descripción la forma en la que la Unidad Central de Proceso (CPU, del inglés Central Processing Unit) trabaja internamente y accede a las direcciones de memoria.

La arquiectura von Neumman es una arquitectura de ordenador basada en la descrita en 1945 por el matemático y físico John von Neumann. Esta arquitectura describe un ordenador con tres partes: unidad de procesamiento, memoria principal y unidad de entrada/salida:

1.1.1 Unidad de procesamiento

Conocida como **CPU** (*Centra Procesing Unit*), **procesador** o **microprocesador**, se encarga de todas las operaciones y el control del ordenador. Es la CPU lo que, en última instancia, estamos programando.

En la CPU encontramos una **Unidad Aritmético Lógica**, una serie de **registros** y una **Unidad de Control** que te describo brevemente a continuación:

- Una **Unidad Aritmético Lógica**, o **ALU** del inglés *Arithmetic Logic Unit* que es un circuito digital que realiza operaciones aritméticas (sumas, restas) y operaciones lógicas (IF, AND, OR, NOT, XOR) entre los valores de los argumentos (que pueden ser uno o dos, según el caso).

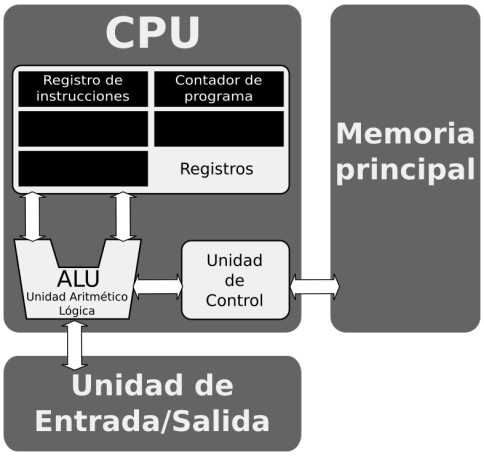

Figura 1: Esquema visual con el resumen de la Arquitectura von Neumann

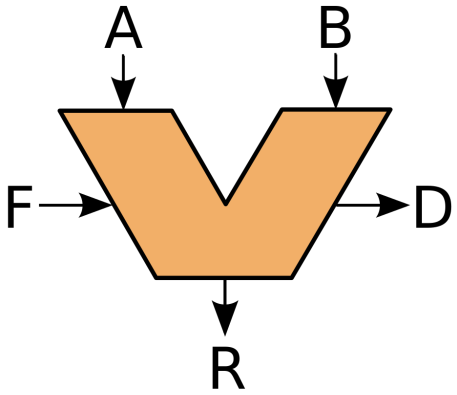

Figura 2: Típico símbolo esquemático para una ALU: A y B son operandos; R es la salida; F es la entrada de la Unidad de Control; D es un estado de la salida

Ver también:

Si sientes curiosidad puedes visitar este sitio web donde encontrarás detalles de cómo funciona una ALU[1].

- Una serie de **registros** que forman la memoria más rápida de la jerarquía de memoria de un ordenador. Se encuentran dentro de la CPU, y se usan para almacenar operandos y resultados de las operaciones que hace la CPU. Los registros se miden por el número de bits que almacenan, por ejemplo: un registro de 8 bits, o un registros de 32 bits, o un registro de 64 bits. Dependiendo de la arquitectura se tienen más o menos registros de más o menos bits.
- Una **Unidad de Control** que manda señales al resto de componentes para que todas las partes estén sincronizadas. Se comporta como los semáforos que regulan el tráfico en los cruces dejando actuar a unos y deteniendo a otros. Además, esta Unidad de Control usa una serie de registros especiales como son: un **registro de instrucciones** donde se almacena la instrucción que se está ejecutando y un **contador de programa** donde se almacena la dirección de la memoria donde está la instrucción que se está ejecutando.

1.1.2 Memoria principal

Esta es la memoria del ordenador que nosotros solemos conocer coloquialmente como «memoria RAM». En esta memoria se almacenan los programas, es decir, todas las instrucciones de todos los programas que se están ejecutando en un ordenador más los datos.

1.1.3 Unidad de Entrada/Salida

O unidad de E/S es el mecanismo que permite conectar al ordenador todo tipo de periféricos para ampliar las opciones de un ordenador. Por ejemplo, una impresora, un ratón o un teclado es conectado a este mecanismo.

1.2 Software y clasificación del software

El software es el conjunto de programas informáticos que actúan sobre el hardware para ejecutar lo que el usuario desee.

Si bien esta distinción es, en cierto modo, arbitraria, y a veces confusa, para fines prácticos se puede clasificar el software en tres tipos: software de sistema, software de programación y software de aplicación.

En el siguiente gráfico te resumo esta clasificación, con sus objetivos y algunos ejemplos de cada categoría:

[1] https://hardzone.es/reportajes/que-es/alu/

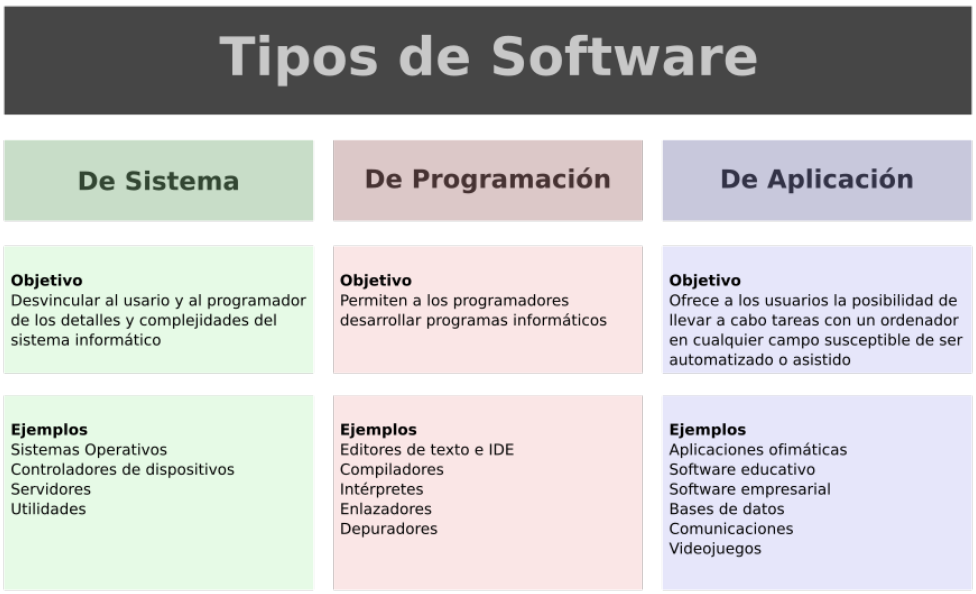

Figura 3: Tipos de software con objetivos y ejemplos

2 Relación software-hardware

Un ordenador consta de un buen número de elementos hardware, y su programación es compleja. En este ciclo vas a aprender a crear software de aplicación, en concreto aplicaciones web. Así pues, no vas a programar el ordenador directamente.

Como ves en el siguiente gráfico, solo el sistema operativo (que es un software de sistema) accede directamente al hardware. El resto de aplicaciones se programan sobre el sistema operativo.

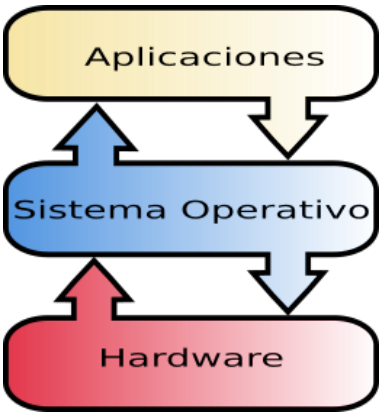

Figura 1: Relación jerárquica entre el software y el hardware de un ordenador

2.1 Sistemas operativos

En el mundo de los ordenadores personales hay tres sistemas operativos más o menos populares, que seguramente ya conocerás: Windows, MacOS y Linux.

El sistema operativo es, en realidad, un **conjunto de programas de sistema que gestiona los recursos hardware** y provee servicios (una interfaz) a los programas de aplicación. Resumiendo, sus funciones principales son:

- **Gestionar la memoria** de acceso aleatorio y ejecutar las aplicaciones, designando los recursos necesarios.
- **Administrar la CPU** gracias a un algoritmo de programación.

- Direccionar las **entradas y salidas** de datos (a través de drivers) por medio de los periféricos de entrada o salida.
- **Administrar la información** para el buen funcionamiento del PC.
- Dirigir las autorizaciones de uso para los **usuarios**.

2.2 Aplicaciones

Las aplicaciones son **programas de usuario** y son escritas por medio de **lenguajes de programación**.

Hay multitud de lenguajes de programación pero todos, hoy en día, tienen una característica en común: usan un léxico, sintaxis y semántica fácil de aprender y usar por el ser humano. Las sentencias de un lenguaje de programación tienen que ser traducidas a un lenguaje o códigos que la máquina pueda interpretar y ejecutar.

Como sabes, una máquina como el ordenador solo es capaz de interpretar señales eléctricas (ausencia o presencia de tensión) que, nosotros interpretamos como 0 (no hay tensión o corriente) y 1 (hay tensión o corriente). A esta codificación la conocemos como **binario**.

En los siguientes apartados vamos a entrar de lleno en los lenguajes de programación, podrás ver que hay varios tipos de lenguajes de programación y de qué forma se pueden traducir las sentencias de alto nivel a código máquina.

3 Lenguajes de programación: evolución histórica

Los lenguajes de programación han sufrido una evolución importante. Los primeros ordenadores eran programados por medio de **lenguaje máquina**, en binario directamente; con el **lenguaje ensamblador** se facilitó la programación añadiendo mnemotécnicos y alejándose del código binario; por último, hoy en día, tenemos **lenguajes de alto nivel** que son fáciles de aprender y, además, hacen la programación más fácil.

En los siguientes apartados te explico brevemente cómo funciona cada uno de los tres tipos de lenguajes.

3.1 Lenguaje máquina

Se trata de un lenguaje de programación de bajo nivel, consistente en **instrucciones en binario** que se usan para el control de un procesador o CPU. Cada instrucción hace que la CPU realice una tarea específica como: cargar un valor en un registro, almacenar un valor en memoria, saltar a una posición de memoria dada o realizar una operación aritmético-lógica, por ejemplo.

Nota: Hoy en día escribir programas en lenguaje máquina es muy raro.

Cada CPU tiene un juego de instrucciones. Si quieres programar en lenguaje máquina tienes que aprender a usar el juego de instrucciones de la CPU a programar, y no te servirá para otra CPU.

Por ejemplo, en las CPU de arquitectura MIPS todas las instrucciones son de 32 bits. Te muestro, a continuación, la instrucción que se usaría para cargar el valor que hay en la celda de memoria 68 a partir de la 8ª celda al registro número 8:

```
100011 00011 01000 00000 00001 000100
```

Estos son los significados de los bits:

```
 100011    00011    01000    00000 00001 000100
|------|  |-----|  |-----|  |------------------|
    |        |        |              |
```

(continué en la próxima página)

(proviene de la página anterior)

```
    |         |         |               +---- Dirección de memoria: 68 en decimal
    |         |         |
    |         |         +-------------------- Número a partir del cual se busca en␣
↪memoria: 8 en decimal
    |         |
    |         +---------------------------- Número de registro donde se carga el␣
↪valor: 3 en decimal
    |
    +------------------------------------- Código de operación: 35 en decimal
```

3.2 Lenguaje ensamblador

Se trata de un lenguaje de bajo nivel, más sencillo de aprender y programar que el lenguaje máquina. Este lenguaje también es dependiente de la arquitectura de la CPU a programar.

En este lenguaje se usan **mnemotécnicos** que representan las instrucciones del microprocesador.

Los programas escritos en lenguaje ensamblador tienen que ser traducidos a lenguaje máquina para que se puedan ejecutar y, como en el caso del lenguaje máquina, no son portables de una CPU a otra.

Nota: Hoy en día tampoco se escriben programas en ensamblador salvo en casos muy concretos como: el *bootloader* de los sistemas operativos y otras partes del mismo que tienen que ver con la parte que arranca y carga el sistema operativo en la memoria.

Por ejemplo, en las CPU de arquitectura x86, para cargar el número 61 en hexadecimal (97 en decimal) al registro `AL` se haría de la siguiente manera en ensamblador:

```
MOV AL, 61h
```

Una vez traducida esta instrucción a binario, lenguaje máquina, quedaría tal cual:

```
10110000 01100001
```

Estos bits tienen el siguiente significado:

```
1011   0000   01100001
   |     |         |
   |     |         |
   |     |         +------ Número 61 en hexadecimal o 97 en decimal
   |     |
   |     +---------------- Código para el registro AL (número 0 en decimal)
   |
   +---------------------- Código para la instrucción MOV (mover un valor a un␣
↪registro)
```

Por último, te muestro un programa escrito en ensamblador para CPU de arquitectura x86 que suma los números 14 y 10:

```
.globl main
    .type main, @function

main:
    movl $14, %eax
    movl $10, %ebx
    add %eax, %ebx
```

Nota: En los lenguajes ensamblador, como ves, se usan identificadores para las instrucciones y registros de la CPU. Así, `eax`, `ebx` o `ebx` son identificadores de tres de los registros de las CPU x86. Y `movl` o `add` son instrucciones de la arquitectura de CPU x86.

3.3 Lenguaje de alto nivel

Estos lenguajes son fáciles de usar y aprender porque **usan un léxico, sintaxis y semántica cercano al ser humano**. Estos lenguajes ocultan los detalles de la arquitectura de la CPU a programar, no hace falta aprenderse el juego de instrucciones del microprocesador, lo que hace mucho más fácil la programación. Así pues, estos lenguajes no dependen de la arquitectura de la CPU.

Cuando aprendes a usar un lenguaje de alto nivel es fácil la transición a otro lenguaje. Por ejemplo, si aprendes a programar en `Java` te resultará muy fácil pasar a programar con `PHP` o `Python` por ejemplo.

Un ejemplo de programa en `C` que carga dos valores en memoria y los suma dejando el resultado en otra posición de memoria sería el siguiente:

```
int main(int argc, char **argv)
{
    int num1, num2, resultado;

    num1 = 14;
    num2 = 10;

    resultado = num1 + num2;

    return 0;
}
```

3.4 En resumen

Tabla 1: Resumen de lenguajes de programación

Lenguaje Máquina	Lenguaje Ensamblador	Lenguaje de Alto Nivel
La programación es compleja	Facilita la programación aunque sigue siendo difícil escribir programas	La programación es fácil
Las instrucciones son en binario	Se programa usando mnemotécnicos (instrucciones complejas)	Se utilizan sentencias y órdenes con un léxico, sintaxis y semántica cercano al lenguaje humano
Estos programas se pueden ejecutar directamente	Necesita traducción al lenguaje máquina para poder ejecutarse	Necesita traducción al lenguaje máquina para poder ejecutarse
Es único para cada procesador (no es portable de un equipo a otro)	Hay diferentes lenguajes de ensamblador por cada arquitectura de CPU	Son independientes de la CPU
Hoy día nadie programa en este lenguaje	Se usa en caso muy concretos, sobre todo en el desarrollo de programas de sistema	Son los que se usan hoy en día

4

Práctica 1: Introducción al desarrollo de software y lenguajes de programación

En esta práctica tienes que responder a una serie de preguntas de tipo test. Aquí te dejo las preguntas. Llegado el momento te indicaré cómo y dónde tienes que responder a estas cuestiones:

- En la Arquitectura von Neumann se describen los siguientes componentes.
- Dentro de una CPU encontramos las siguientes unidades funcionales.
- ¿Dónde se cargan los programas a ser ejecutados?
- ¿Quién ejecuta los programas?
- Si se tiene que hacer una suma, a nivel hardware, ¿qué unidad funcional la realizará?
- ¿Quién gestiona la carga de los programas de usuario (aplicaciones) en la memoria para ser ejecutados?
- ¿Qué software tiene acceso al hardware del ordenador?
- ¿Cuál fue la evolución histórica de los lenguajes de programación (de más antiguo a más actual)?
- ¿Qué lenguaje o lenguajes puedes usar para escribir programas que puedan ser ejecutados directamente por la CPU, sin traducción?
- ¿En qué lenguaje ha sido escrita la siguiente instrucción: 00011 00011 01000 00000 00001 000100?
- ¿En qué lenguaje ha sido escrita la siguiente instrucción: MOVL $10, %ebx?
- ¿En qué lenguaje ha sido escrita la siguiente instrucción: int num1 = 10;?

4.1 Criterios de evaluación

En esta práctica se aplica el Resultado de Aprendizaje 1: reconoce los elementos y herramientas que intervienen en el desarrollo de un programa informático, analizando sus características y las fases en las que actúan hasta llegar a su puesta en funcionamiento.. Y en concreto los siguientes criterios de evaluación:

- a) Se ha reconocido la relación de los programas con los componentes del sistema informático: memoria, procesador y periféricos, entre otros. (10%)
- e) Se han clasificado los lenguajes de programación. (20%)

5 Lenguajes de alto nivel: clasificación

En el apartado anterior ya estudiamos la evolución histórica de los lenguajes de programación: máquina, ensamblador y alto nivel.

En este apartado nos vamos a centrar en los lenguajes de alto nivel, que al final son estos con los que vas a aprender a programar en este ciclo.

Dentro del mundo de los lenguajes de alto nivel existen varios paradigmas que determinan, en cierto modo, cómo se escriben y estructuran los programas. Algunos de los paradigmas más populares son: programación estructurada, programación modular y programación orientada a objetos (POO).

Ver también:

Speedcoding, **Speedcode** o **SpeedCo** es el primer lenguaje de alto nivel, creado para un ordenador IBM 701 por John W. Backus en 1953.

No obstante, el primer lenguaje con cierta popularidad fue **Fortran**, creado por la propia IBM en 1957. Hoy en día se sigue usando.

Fue en 1972, cuando Dennis Ritchie diseñó el lenguaje **C** que ha influenciado a casi la totalidad de lenguajes populares de hoy en día, como **C++**, **Java**, **Python**, **PHP** o **JavaScript**, por ejemplo. Hoy en día, **C** es ampliamente utilizado para la creació de programas de sistema como son los sistemas operativos y aplicaciones para sistemas embebidos.

5.1 Código espagueti: paradigma a evitar

El código espagueti es un **término peyorativo** para los programas de computación que tienen una estructura de control de flujo compleja e incomprensible. Su nombre deriva del hecho que este tipo de código parece asemejarse a un plato de espaguetis, es decir, un montón de hilos intrincados y anudados.

Tradicionalmente suele asociarse este estilo de programación con lenguajes básicos y antiguos, donde el flujo se controlaba mediante sentencias de control muy primitivas como goto y utilizando números de línea.

Para evitar este *paradigma* hay que emplear alguno de los paradigmas que te explico en los siguientes apartados.

5.2 Programación estructurada

Este es un paradigma de programación cuyo objetivo es mejorar la claridad, calidad y el tiempo de desarrollo de los programas haciendo un uso extensivo de:

- Estructuras de control de selección del flujo `if/then/else`.
- Estructuras de control de repetición (bucles) `while` y `for`.
- Estructuras de bloque: bloques de código relacionados y agrupados de alguna manera.
- Subrutinas: secuencias de instrucciones del programa que realizan una tarea específica, agrupadas como una unidad.

Nota: En diferentes lenguajes de programación, a las **subrutinas** se les puede llamar: **rutina**, **subprograma**, **función**, **método** o **procedimiento**.

Las ventajas de este paradigma son:

- Los programas son más fáciles de entender y leer.
- Los programas son más rápidos.
- La estructura de los programas es clara.
- Se construyen programas con más rapidez.
- Se facilita la depuración y detección de errores.
- Se reducen los costes de mantenimiento.

Los inconvenientes de este paradigma son:

- Todo el programa se concentra en un único bloque (si se hace demasiado grande es difícil de manejar y mantener).
- No permite reutilización eficaz de código, ya que todo va «en uno». Es por esto que a la programación estructurada le sustituyó la programación modular, donde los programas se codifican por módulos y bloques, permitiendo mayor funcionalidad.

Nota: Ejemplos de lenguajes estructurados: **Fortran**, **Pascal** y **C**.

5.3 Programación modular

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

Se presenta históricamente como una evolución de la programación estructurada para solucionar problemas de programación más grandes y complejos de lo que esta puede resolver.

Esto se suele traducir en que un programa está formado por varios ficheros, y en la práctica cada fichero sería un modulo, una parte del programa, que se ha dividido en diferentes piezas.

Está a medio camino entre la programación estructurada y la programación orientada a objetos.

5.4 Programación orientada a objetos

Después de comprender que la programación estructurada no es útil cuando los programas se hacen muy largos, es necesaria otra técnica de programación que solucione este inconveniente. Nace así la programación orientada a objetos.

Los lenguajes de programación orientados a objetos tratan a los programas no como un conjunto ordenado de instrucciones (tal como sucedía en la programación estructurada) sino como un conjunto de objetos que colaboran entre ellos para realizar acciones.

Su primera desventaja es clara: no es una programación tan intuitiva como la estructurada.

A pesar de eso, mucho del software que produce se hace usando esta técnica por las siguientes razones:

- El código es reutilizable.
- Si hay algún error, es más fácil de localizar y depurar en un objeto que en un programa entero.

Nota: Ejemplos de lenguajes orientados a objetos: **C++** y **Java**.

---

6 Fases en la obtención del código

Como ya sabes los programas escritos usando un lenguaje de alto nivel necesitan ser traducidos a lenguaje máquina. ¿Cómo se obtiene el código máquina o ejecutable? Es lo que vas a aprender en este apartado.

Nota: Para ilustrar los ejemplos voy a usar el lenguaje de programación C.

6.1 Código fuente, objeto y ejecutable

Antes tienes que saber que existen tres tipos de código: código **fuente**, código **objeto** y código **ejecutable o máquina**.

6.1.1 Código fuente

Se trata de los ficheros con el **código escrito en alto nivel** del lenguaje de programación escogido para la programación. También se denomina código fuente a los programas escritos en **ensamblador**. Un programa puede estar escrito en uno o varios ficheros.

A continuación te muestro el código fuente de un programa escrito en C y su equivalente en ensamblador. Se trata de un programa que suma dos números y muestra el resultado por pantalla.

Código fuente en C

Código fuente en ensamblador

Lista 1: Programa escrito en C que suma dos números y muestra el resultado por pantalla

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int num1, num2, resultado;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    num1 = 15;
    num2 = 20;

    resultado = num1 + num2;

    printf("Resultado de sumar %d y %d = %d\n", num1, num2, resultado);

    return 0;
}
```

Lista 2: Programa escrito en ensamblador que suma dos números y muestra el resultado por pantalla

```
    .file   "main.c"
    .text
    .section    .rodata
    .align 8
.LC0:
    .string "Resultado de sumar %d y %d = %d\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movl    %edi, -20(%rbp)
    movq    %rsi, -32(%rbp)
    movl    $15, -4(%rbp)
    movl    $20, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    movl    %eax, -12(%rbp)
    movl    -12(%rbp), %ecx
    movl    -8(%rbp), %edx
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Debian 10.2.1-6) 10.2.1 20210110"
    .section        .note.GNU-stack,"",@progbits
```

6.1.2 Código objeto

Es el **código binario resultante de traducir cada fichero fuente**. Este código no es inteligible por el ser humano pero tampoco está listo para ser ejecutado por la CPU.

Aquí te muestro el código *desensamblado* en hexadecimal, donde se indican las direcciones de la memoria virtual donde se cargará el programa y las instrucciones, también en hexadecimal.

Por cuestiones de claridad, a la derecha tienes el código ensamblador equivalente.

Lista 3: Código objeto del programa anterior desensamblado (a la derecha tienes el código en ensamblador)

```
0000000000000000 <main>:
0:  55                      push   %rbp
1:  48 89 e5                mov    %rsp,%rbp
4:  48 83 ec 20             sub    $0x20,%rsp
8:  89 7d ec                mov    %edi,-0x14(%rbp)
b:  48 89 75 e0             mov    %rsi,-0x20(%rbp)
f:  c7 45 fc 0f 00 00 00    movl   $0xf,-0x4(%rbp)
16: c7 45 f8 14 00 00 00    movl   $0x14,-0x8(%rbp)
1d: 8b 55 fc                mov    -0x4(%rbp),%edx
20: 8b 45 f8                mov    -0x8(%rbp),%eax
23: 01 d0                   add    %edx,%eax
25: 89 45 f4                mov    %eax,-0xc(%rbp)
28: 8b 4d f4                mov    -0xc(%rbp),%ecx
2b: 8b 55 f8                mov    -0x8(%rbp),%edx
2e: 8b 45 fc                mov    -0x4(%rbp),%eax
31: 89 c6                   mov    %eax,%esi
33: 48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 3a <main+0x3a>
3a: b8 00 00 00 00          mov    $0x0,%eax
3f: e8 00 00 00 00          callq  44 <main+0x44>
44: b8 00 00 00 00          mov    $0x0,%eax
49: c9                      leaveq
4a: c3                      retq
```

6.1.3 Código ejecutable o máquina

Es el código binario resultante de enlazar los archivos con el código objeto y ciertas rutinas y bibliotecas necesarias.

Siguiendo con el ejemplo anterior, aquí te muestro el código ejecutable *desensamblado*. Como ves es bastante más largo que el código objeto anterior porque se han añadido ciertas rutinas y bibliotecas necesarias, todo en un mismo fichero.

Por cuestiones de claridad, aquí también te muestro a la derecha el código ensamblador equivalente.

Lista 4: Código ejecutable o máquina del programa anterior (a la derecha se muestra el código ensamblador equivalente)

```
0000000000001000 <_init>:
    1000:   48 83 ec 08             sub    $0x8,%rsp
    1004:   48 8b 05 dd 2f 00 00    mov    0x2fdd(%rip),%rax        # 3fe8 <__gmon_
↪start__>
```

(continué en la próxima página)

(proviene de la página anterior)

```
    100b:   48 85 c0                test   %rax,%rax
    100e:   74 02                   je     1012 <_init+0x12>
    1010:   ff d0                   callq  *%rax
    1012:   48 83 c4 08             add    $0x8,%rsp
    1016:   c3                      retq

Desensamblado de la sección .plt:

0000000000001020 <.plt>:
    1020:   ff 35 e2 2f 00 00       pushq  0x2fe2(%rip)        # 4008 <_GLOBAL_OFFSET_
↪TABLE_+0x8>
    1026:   ff 25 e4 2f 00 00       jmpq   *0x2fe4(%rip)        # 4010 <_GLOBAL_
↪OFFSET_TABLE_+0x10>
    102c:   0f 1f 40 00             nopl   0x0(%rax)

0000000000001030 <printf@plt>:
    1030:   ff 25 e2 2f 00 00       jmpq   *0x2fe2(%rip)        # 4018 <printf@GLIBC_
↪2.2.5>
    1036:   68 00 00 00 00          pushq  $0x0
    103b:   e9 e0 ff ff ff          jmpq   1020 <.plt>

Desensamblado de la sección .plt.got:

0000000000001040 <__cxa_finalize@plt>:
    1040:   ff 25 b2 2f 00 00       jmpq   *0x2fb2(%rip)        # 3ff8 <__cxa_
↪finalize@GLIBC_2.2.5>
    1046:   66 90                   xchg   %ax,%ax

Desensamblado de la sección .text:

0000000000001050 <_start>:
    1050:   31 ed                   xor    %ebp,%ebp
    1052:   49 89 d1                mov    %rdx,%r9
    1055:   5e                      pop    %rsi
    1056:   48 89 e2                mov    %rsp,%rdx
    1059:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    105d:   50                      push   %rax
    105e:   54                      push   %rsp
    105f:   4c 8d 05 7a 01 00 00    lea    0x17a(%rip),%r8        # 11e0 <__libc_csu_
↪fini>
    1066:   48 8d 0d 13 01 00 00    lea    0x113(%rip),%rcx        # 1180 <__libc_csu_
↪init>
    106d:   48 8d 3d c1 00 00 00    lea    0xc1(%rip),%rdi        # 1135 <main>
    1074:   ff 15 66 2f 00 00       callq  *0x2f66(%rip)        # 3fe0 <__libc_start_
↪main@GLIBC_2.2.5>
    107a:   f4                      hlt
    107b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

0000000000001080 <deregister_tm_clones>:
    1080:   48 8d 3d a9 2f 00 00    lea    0x2fa9(%rip),%rdi        # 4030 <__TMC_END_
↪_>
    1087:   48 8d 05 a2 2f 00 00    lea    0x2fa2(%rip),%rax        # 4030 <__TMC_END_
↪_>
    108e:   48 39 f8                cmp    %rdi,%rax
    1091:   74 15                   je     10a8 <deregister_tm_clones+0x28>
    1093:   48 8b 05 3e 2f 00 00    mov    0x2f3e(%rip),%rax        # 3fd8 <_ITM_
↪deregisterTMCloneTable>
```

(continué en la próxima página)

(proviene de la página anterior)

```
    109a:   48 85 c0                test   %rax,%rax
    109d:   74 09                   je     10a8 <deregister_tm_clones+0x28>
    109f:   ff e0                   jmpq   *%rax
    10a1:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
    10a8:   c3                      retq
    10a9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000000010b0 <register_tm_clones>:
    10b0:   48 8d 3d 79 2f 00 00    lea    0x2f79(%rip),%rdi        # 4030 <__TMC_END_
↪_>
    10b7:   48 8d 35 72 2f 00 00    lea    0x2f72(%rip),%rsi        # 4030 <__TMC_END_
↪_>
    10be:   48 29 fe                sub    %rdi,%rsi
    10c1:   48 89 f0                mov    %rsi,%rax
    10c4:   48 c1 ee 3f             shr    $0x3f,%rsi
    10c8:   48 c1 f8 03             sar    $0x3,%rax
    10cc:   48 01 c6                add    %rax,%rsi
    10cf:   48 d1 fe                sar    %rsi
    10d2:   74 14                   je     10e8 <register_tm_clones+0x38>
    10d4:   48 8b 05 15 2f 00 00    mov    0x2f15(%rip),%rax        # 3ff0 <_ITM_
↪registerTMCloneTable>
    10db:   48 85 c0                test   %rax,%rax
    10de:   74 08                   je     10e8 <register_tm_clones+0x38>
    10e0:   ff e0                   jmpq   *%rax
    10e2:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    10e8:   c3                      retq
    10e9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

00000000000010f0 <__do_global_dtors_aux>:
    10f0:   80 3d 39 2f 00 00 00    cmpb   $0x0,0x2f39(%rip)        # 4030 <__TMC_END_
↪_>
    10f7:   75 2f                   jne    1128 <__do_global_dtors_aux+0x38>
    10f9:   55                      push   %rbp
    10fa:   48 83 3d f6 2e 00 00    cmpq   $0x0,0x2ef6(%rip)        # 3ff8 <__cxa_
↪finalize@GLIBC_2.2.5>
    1101:   00
    1102:   48 89 e5                mov    %rsp,%rbp
    1105:   74 0c                   je     1113 <__do_global_dtors_aux+0x23>
    1107:   48 8b 3d 1a 2f 00 00    mov    0x2f1a(%rip),%rdi        # 4028 <__dso_
↪handle>
    110e:   e8 2d ff ff ff          callq  1040 <__cxa_finalize@plt>
    1113:   e8 68 ff ff ff          callq  1080 <deregister_tm_clones>
    1118:   c6 05 11 2f 00 00 01    movb   $0x1,0x2f11(%rip)        # 4030 <__TMC_END_
↪_>
    111f:   5d                      pop    %rbp
    1120:   c3                      retq
    1121:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
    1128:   c3                      retq
    1129:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001130 <frame_dummy>:
    1130:   e9 7b ff ff ff          jmpq   10b0 <register_tm_clones>

0000000000001135 <main>:
    1135:   55                      push   %rbp
    1136:   48 89 e5                mov    %rsp,%rbp
```

(continué en la próxima página)

(proviene de la página anterior)

```
    1139:   48 83 ec 20             sub    $0x20,%rsp
    113d:   89 7d ec                mov    %edi,-0x14(%rbp)
    1140:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
    1144:   c7 45 fc 0f 00 00 00    movl   $0xf,-0x4(%rbp)
    114b:   c7 45 f8 14 00 00 00    movl   $0x14,-0x8(%rbp)
    1152:   8b 55 fc                mov    -0x4(%rbp),%edx
    1155:   8b 45 f8                mov    -0x8(%rbp),%eax
    1158:   01 d0                   add    %edx,%eax
    115a:   89 45 f4                mov    %eax,-0xc(%rbp)
    115d:   8b 4d f4                mov    -0xc(%rbp),%ecx
    1160:   8b 55 f8                mov    -0x8(%rbp),%edx
    1163:   8b 45 fc                mov    -0x4(%rbp),%eax
    1166:   89 c6                   mov    %eax,%esi
    1168:   48 8d 3d 99 0e 00 00    lea    0xe99(%rip),%rdi        # 2008 <_IO_stdin_
↪used+0x8>
    116f:   b8 00 00 00 00          mov    $0x0,%eax
    1174:   e8 b7 fe ff ff          callq  1030 <printf@plt>
    1179:   b8 00 00 00 00          mov    $0x0,%eax
    117e:   c9                      leaveq
    117f:   c3                      retq

0000000000001180 <__libc_csu_init>:
    1180:   41 57                   push   %r15
    1182:   4c 8d 3d 5f 2c 00 00    lea    0x2c5f(%rip),%r15        # 3de8 <__frame_
↪dummy_init_array_entry>
    1189:   41 56                   push   %r14
    118b:   49 89 d6                mov    %rdx,%r14
    118e:   41 55                   push   %r13
    1190:   49 89 f5                mov    %rsi,%r13
    1193:   41 54                   push   %r12
    1195:   41 89 fc                mov    %edi,%r12d
    1198:   55                      push   %rbp
    1199:   48 8d 2d 50 2c 00 00    lea    0x2c50(%rip),%rbp        # 3df0 <__do_
↪global_dtors_aux_fini_array_entry>
    11a0:   53                      push   %rbx
    11a1:   4c 29 fd                sub    %r15,%rbp
    11a4:   48 83 ec 08             sub    $0x8,%rsp
    11a8:   e8 53 fe ff ff          callq  1000 <_init>
    11ad:   48 c1 fd 03             sar    $0x3,%rbp
    11b1:   74 1b                   je     11ce <__libc_csu_init+0x4e>
    11b3:   31 db                   xor    %ebx,%ebx
    11b5:   0f 1f 00                nopl   (%rax)
    11b8:   4c 89 f2                mov    %r14,%rdx
    11bb:   4c 89 ee                mov    %r13,%rsi
    11be:   44 89 e7                mov    %r12d,%edi
    11c1:   41 ff 14 df             callq  *(%r15,%rbx,8)
    11c5:   48 83 c3 01             add    $0x1,%rbx
    11c9:   48 39 dd                cmp    %rbx,%rbp
    11cc:   75 ea                   jne    11b8 <__libc_csu_init+0x38>
    11ce:   48 83 c4 08             add    $0x8,%rsp
    11d2:   5b                      pop    %rbx
    11d3:   5d                      pop    %rbp
    11d4:   41 5c                   pop    %r12
    11d6:   41 5d                   pop    %r13
    11d8:   41 5e                   pop    %r14
    11da:   41 5f                   pop    %r15
```

(continué en la próxima página)

(proviene de la página anterior)

```
    11dc:   c3                      retq
    11dd:   0f 1f 00                nopl   (%rax)

00000000000011e0 <__libc_csu_fini>:
    11e0:   c3                      retq

Desensamblado de la sección .fini:

00000000000011e4 <_fini>:
    11e4:   48 83 ec 08             sub    $0x8,%rsp
    11e8:   48 83 c4 08             add    $0x8,%rsp
    11ec:   c3                      retq
```

6.2 Traductores de código

Existen dos tipos de lenguajes de programación en cuanto a la **forma de obtener el código máquina** o ejecutable: lenguajes **interpretados** y lenguajes **compilados**.

En los lenguajes interpretados, un programa llamado **intérprete** es el encargado de ir traduciendo el programa a medida que sea necesario, típicamente instrucción a instrucción, y normalmente no guardan el resultado de esa traducción.

En los lenguajes compilados, un programa llamado **compilador** traduce todo el programa a lenguaje máquina generando un segundo fichero con dicha traducción.

Sea interpretado o compilado **el programa es ejecutado por medio del sistema operativo** y, dicho código, está preparado para ser **ejecutado en una arquitectura de CPU** concreta (x86-32, x86-64, MIPS, ARM, etc). Este es el encargado de cargar el programa ejecutable en memoria y gestionar su ejecución por medio de una serie de algoritmos de gestión de procesos.

En el siguiente apartado te voy a explicar cómo funciona un **compilador**.

Nota: Ejemplos de lenguajes interpretados son: **Python** o **PHP**.

Ejemplos de lenguajes compilados son: **C** o **Java**. Aunque el compilador de **Java**, como veremos más adelante, genera un código máquina intermedio que será ejecutado por una máquina virtual.

6.3 Compiladores: proceso de traducción

Todo comienza cuando el programador **escribe el código fuente**, normalmente en varios ficheros.

A continuación, se ejecuta el compilador sobre dichos ficheros generando **el código objeto** y dando como resultado los mismos ficheros pero con dicho código objeto. Esta traducción solo se realiza si no hay errores léxicos, sintáctico y/o semánticos en el código fuente escrito por el programador. Si hay errores en el código, el compilador avisa al programador de dichos errores.

Una vez se tienen los ficheros objeto, un programa llamado **linker** enlaza todos los ficheros con el código objeto y bibliotecas generando un solo fichero con el **código máquina o ejecutable**. El programa **linker** forma parte del compilador y, si no se indica lo contrario, el compilador generará el código objeto y ejecuta, a continuación, el linker automáticamente, sin que tengas que hacer nada extra.

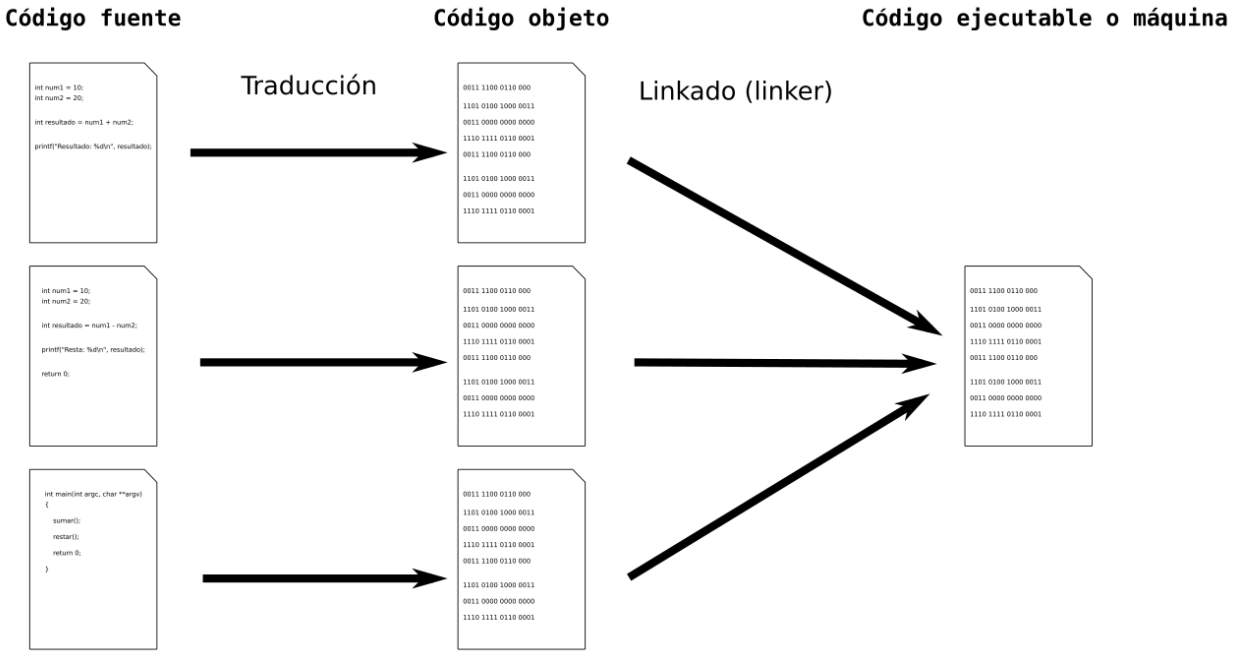

Figura 1: Fases de un compilador: traducción y enlace (*linkado*)

Importante: El compilador genera el código ejecutable para una CPU y sistema operativo concreto. Si quieres tener el programa listo para ser ejecutado en CPU MIPS, x86-64 y ARM, entonces tienes que compilar el programa 3 veces, para obtener el código ejecutable en cada una de estas tres arquitecturas.

6.4 Código intermedio y máquinas virtuales

Como ya se ha comentado, el código ejecutable o máquina está listo para ser ejecutado por un tipo de CPU concreta.

Hay lenguajes, como **Java**, para los que el compilador genera un código intermedio genérico que es ejecutado por una máquina virtual. Este código intermedio se denomina **bytecode**.

De esta manera no hace falta compilar el código fuente varias veces para ser ejecutado en varias CPU, se construye un código genérico ejecutable por cualquier máquina. No obstante, para poder ejecutar estos programas antes necesitas instalar la máquina virtual en tu ordenador.

Siguiendo con el ejemplo de **Java**, este tiene un compilador, **javac**, que genera código intermedio (**bytecode**) para ser ejecutado por la **Java Virtual Machine** o *JVM*.

Si quieres ejecutar en tu ordenador programas en Java, tienes que instalar el **JRE** (*Java Runtime Environment*). Si quieres desarrollar programas en Java, entonces necesitas instalar el **JDK** (*Java Development Kit*), que incluye el JRE.

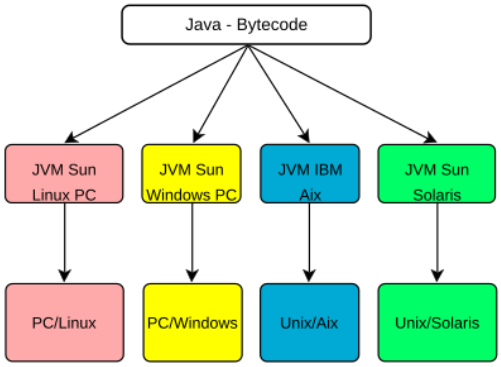

Figura 2: Esquema de la arquitectura general de un programa en ejecución en una JVM

Por último, aquí te muestro el código fuente de un programa en Java que suma dos números y muestra el resultado por pantalla. Y, también, el *bytecode*.

Código fuente en Java

Bytecode del código en Java

```
public class Main {
    public static void main(String[] args) {
        int num1, num2, resultado;

        num1 = 10;
        num2 = 15;

        resultado = num1 + num2;

        System.out.println("Resultado de sumar " + num1 + " y " + num2 + " = " +␣
↪resultado);
    }
}
```

```
public class Main {
    public Main();
        Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
        4: return

    public static void main(java.lang.String[]);
        Code:
        0: bipush        10
        2: istore_1
        3: bipush        15
        5: istore_2
        6: iload_1
        7: iload_2
        8: iadd
        9: istore_3
```

(continué en la próxima página)

(proviene de la página anterior)

```
       10: getstatic     #7                  // Field java/lang/System.out:Ljava/io/
↪PrintStream;
       13: iload_1
       14: iload_2
       15: iload_3
       16: invokedynamic #13,  0             // InvokeDynamic
↪#0:makeConcatWithConstants:(III)Ljava/lang/String;
       21: invokevirtual #17                 // Method java/io/PrintStream.
↪println:(Ljava/lang/String;)V
       24: return
   }
```

7 Laboratorio 1: generación de código máquina

En este laboratorio vas a generar el código máquina de una programa escrito en lenguaje de programación de alto nivel `C`.

Antes, tienes que instalar el compilador de `C` llamado `gcc`. En los equipos del aula ya lo tienes instalado. Si usas tu portátil u ordenador y usas Windows puedes seguir los pasos que se indican en la siguiente web: instalar gcc en Windows 10[2].

Una vez tengas `gcc` instalado, sigue los pasos que se indican a continuación.

7.1 Paso 1: escribir el programa

Copia y pega el código fuente que tienes aquí en tu editor de textos favorito y guárdalo con el nombre `main.c`.

```c
#include <stdio.h>

int main (int argc, char **argv)
{
    printf ("Números del 1 al 10:\n");
    for (int i = 1; i <= 10; i++)
    {
        printf("%d ", i);
    }
    printf("\n");

    return 0;
}
```

[2] https://platzi.com/tutoriales/1469-algoritmos/1901-como-instalar-gcc-para-compilar-programas-en-c-desde-la-consola-en-windov

7.2 Paso 2: genera el código ensamblador

Aunque no es necesario, en este paso vas a generar el código fuente en ensamblador equivalente.

Para ello, abre una terminal, sitúate en el directorio donde tienes el fichero con el código fuente `main.c` y ejecuta el siguiente comando:

```
$ gcc -S main.c -o main.asm
```

Tras ejecutar dicho comando verás que hay un nuevo fichero `main.asm` que contiene el código fuente en ensamblador.

7.3 Paso 3: genera el código objeto

Para obtener el código objeto, usa el compilador de `C` de la siguiente manera. Abre la terminal y situate donde está el fichero `main.c` y ejecuta el compilador así:

```
$ gcc -c main.c -o main.o
```

7.4 Paso 4: inspeccionar el código objeto

El código objeto no se puede visualizar. Intenta abrir el fichero `main.o` para comprobar que no se puede ver nada, no es inteligible para el ser humano.

No obstante se puede desensamblar para inspeccionar el código objeto en hexadecimal. Abre la terminal y sitúate en el directorio donde está el fichero `main.o` y ejecuta la siguiente orden:

```
$ objdump -d main.o > main.o.hex
```

Este comando genera una representación en hexadecimal, con sus respectivos mnemotécnicos en ensamblador, y lo guarda en el fichero `main.o.hex`. Ábrelo para verlo.

7.5 Paso 5: generar el código máquina ejecutable

El código objeto `main.o` no se puede ejecutar, porque falta enlazarlo con bibliotecas del lenguaje `C`, entre otras cosas.

Para poder ejecutar el programa que escribiste en el Paso 1 usando el lenguaje de programación `C`, tienes que compilar de la siguiente forma:

```
$ gcc main.c -o main
```

Como siempre, antes de ejecutar ese comando tienes que situarte en el directorio donde está el fichero `main.c`. Verás que se generará un fichero, con el código máquina ejecutable, llamado `main`.

7.6 Paso 6: ejecuta el programa

Tienes que ejecutar el código máquina ejecutable que tienes en el fichero `main`. Abre la termina, sitúate en el directorio donde está el programa (fichero `main`) y ejecútalo tal que así:

```
$ ./main
```

Verás que muestra los primeros 10 números.

7.7 Entrega

Hazme entrega de los siguientes ficheros:

- `main.c`
- `main.asm`
- `main.o`
- `main.o.hex`
- `main`

7.8 Criterios de evaluación

En esta práctica se aplica el `Resultado de Aprendizaje 1: reconoce los elementos y herramientas que intervienen en el desarrollo de un programa informático, analizando sus características y las fases en las que actúan hasta llegar a su puesta en funcionamiento.`. Y en concreto los siguientes criterios de evaluación:

- c) Se han diferenciado los conceptos de código fuente, código objeto y código ejecutable. (20%)
- d) Se han reconocido las características de la generación de código intermedio para su ejecución en máquinas virtuales. (10%)

8 Laboratorio 2: generación de bytecode

En este laboratorio vas a generar el código intermedio bytecode de Java desensamblado, para poder visualizar, qué aspecto tiene dicho código intermedio.

Antes, tienes que instalar el `JDK`. En los equipos del aula ya está instalado, así que solo lo tienes que hacer en tu ordenador o portátil siempre que este *lab* lo vayas a realizar desde tu equipo personal.

8.1 Paso 1: escribir el programa

Copia y pega el código fuente en Java que tienes aquí en tu editor de textos favorito y guárdalo con el nombre `Main.java`.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Números del 1 al 10:");
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

8.2 Paso 2: compilar programa en Java

Cuando tenemos un programa en Java en un simple fichero, compilar dicho programa es muy sencillo (sin ayuda de IDEs ni herramientas gráficas).

Tan solo tienes que abrir una terminal, situarte donde está el fichero `Main.java` y ejecutar el compilador de Java **`javac`** tal que así:

```
$ javac Main.java
```

Tras ejecutar este comando verás que se ha generado un fichero con el mismo nombre pero con extensión `.class`, en este ejemplo `Main.class`, que contiene el *bytecode* de Java ejecutable por la `JVM` o Máquina Virtual de Java.

Si intentas abrir este `.class` con un editor verás una serie de símbolos sin sentido. Recuerda que el código objeto es ininteligible para el ser humano.

8.3 Paso 3: análisis del bytecode

Cuando no dispones de los ficheros con el código fuente en Java, puedes usar la herramienta `javap` para desensamblar un fichero `.class` tal que así:

```
$ javap -c Main.class > Main.dis
```

Tras ejecutar dicho comando verás el código desensamblado en el fichero llamado `Main.dis`. Ábrelo con tu editor de textos favorito y observa el resultado.

8.4 Paso 4: visualizar el código máquina

Para poder echar un vistazo al código máquina, en formato hexadecimal, de un programa compilado en Java, debes usar el siguiente comando:

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly Main > Main.asm
```

Donde:

- **java**: se trata del comando que llama a la máquina virtual de Java para ejecutar un programa.
- **-XX:+UnlockDiagnosticVMOptions**: es una opción necesaria para ver el código máquina.
- **-XX:+PrintAssembly**: es para que te muestre el código ensamblador en hexadecimal.
- **Main**: es el nombre del programa (coincide con el nombre de la clase a ejecutar).
- **Main.asm**: nombre del fichero donde escribe el resultado.

Este comando escribe el contenido en un fichero llamado `Main.asm`. Ábrelo con tu editor de textos favorito y observa el resultado.

8.5 Entrega

Hazme entrega de los siguientes ficheros:

- `Main.java`
- `Main.class`
- `Main.dis`
- `Main.asm`

8.6 Criterios de evaluación

En esta práctica se aplica el `Resultado de Aprendizaje 1: reconoce los elementos y herramientas que intervienen en el desarrollo de un programa informático, analizando sus características y las fases en las que actúan hasta llegar a su puesta en funcionamiento.`. Y en concreto los siguientes criterios de evaluación:

- c) Se han diferenciado los conceptos de código fuente, código objeto y código ejecutable. (20%)
- d) Se han reconocido las características de la generación de código intermedio para su ejecución en máquinas virtuales. (10%)