

El software

Lenguaje de máquina y lenguaje ensamblador

En la unidad anterior hemos presentado al MCBE, un ejemplo de computador ideal muy elemental. Como hemos visto, el MCBE cuenta con una CPU, una unidad de E/S, una memoria de unas pocas posiciones, y un reducido conjunto de instrucciones.

La CPU del MCBE es capaz de ejecutar un programa, es decir, una lista de instrucciones, que se disponen en los bytes de la memoria, conteniendo códigos de operaciones y datos en un cierto formato de bits. Esos códigos conforman el **lenguaje de máquina**¹, o **código de máquina**, del MCBE.

Por supuesto, las CPUs de las máquinas reales modernas tienen un conjunto de instrucciones más rico, así como un hardware más completo y una conducta más compleja; pero, en lo fundamental, se parecen mucho al MCBE, y también tienen, cada una, un lenguaje de máquina que le es propio. Como las instrucciones son muchas, de mayor longitud², y se presentan en muchos modos de funcionamiento, programar en código máquina estas computadoras de la realidad, tan complejas, rápidamente se vuelve una tarea inabordable para los humanos. Para programar estas máquinas resulta mucho más cómodo utilizar un lenguaje simbólico, el **lenguaje de ensamblado**³, **ensamblador**, o **Assembly** (a veces también llamado **lenguaje Assembler**, pero que no debe confundirse con el **programa Assembler** que enseguida describiremos).

En el lenguaje ensamblador, cada instrucción del lenguaje máquina está representada por una abreviatura o **mnemónico** que recuerda su función, y así resulta (un poco) más fácil programar la computadora. Por ejemplo, es *un poco* más fácil escribir, comprender y corregir el programa en Assembly del MCBE que se muestra en la figura, que su equivalente en lenguaje de máquina.

LD 30	→	01011110
SUB 5	→	10100101
ST 6	→	01100110
JZ -3	→	11111101
HLT	→	00100000
1	→	00000001
0	→	00000000

Sin embargo, como dijimos, éste no es más que un lenguaje simbólico. Sigue siendo un hecho que lo único que comprende la computadora son números binarios, y que lo único que podemos introducir en la memoria y en los registros de la CPU son bits y bytes. La computadora, de cualquier arquitectura que se trate, no puede ejecutar más que las instrucciones del código de máquina para el cual fue diseñada. Por lo cual, en algún momento, alguien o algo debe efectuar la **traducción** de los mnemónicos, escritos por el programador, a los bits y bytes del programa en código de máquina.

¿Quién se encargará de esta tarea de traducción? Inicialmente, la traducción era hecha a mano por el mismo programador; sin embargo, no tardó en surgir la idea de crear un programa que hiciera esta traducción. Al fin y al cabo, se trata de manipular información, cosa que las computadoras hacen

1 http://es.wikipedia.org/wiki/Lenguaje_de_máquina

2 Las instrucciones de máquina eran de 36 bits ya en 1954: http://es.wikipedia.org/wiki/IBM_704

3 http://es.wikipedia.org/wiki/Lenguaje_ensamblador

bien y rápidamente. Así se creó el primer **programa ensamblador**, o **Assembler**, para traducir de código ensamblador a código máquina. El programa ensamblador (el **programa Assembler**) recibe el código en lenguaje ensamblador (o **programa en Assembly**), y genera, línea por línea, bytes que son instrucciones de código máquina, que pueden ser cargadas en la memoria de la computadora y ser ejecutadas como un programa.

El programa en Assembly es esencialmente un texto, compuesto por una secuencia de caracteres legibles, que conforman la secuencia de mnemónicos que haya utilizado el programador. Por eso, el programa en Assembly es completamente diferente de las instrucciones de código máquina a las cuales equivale.

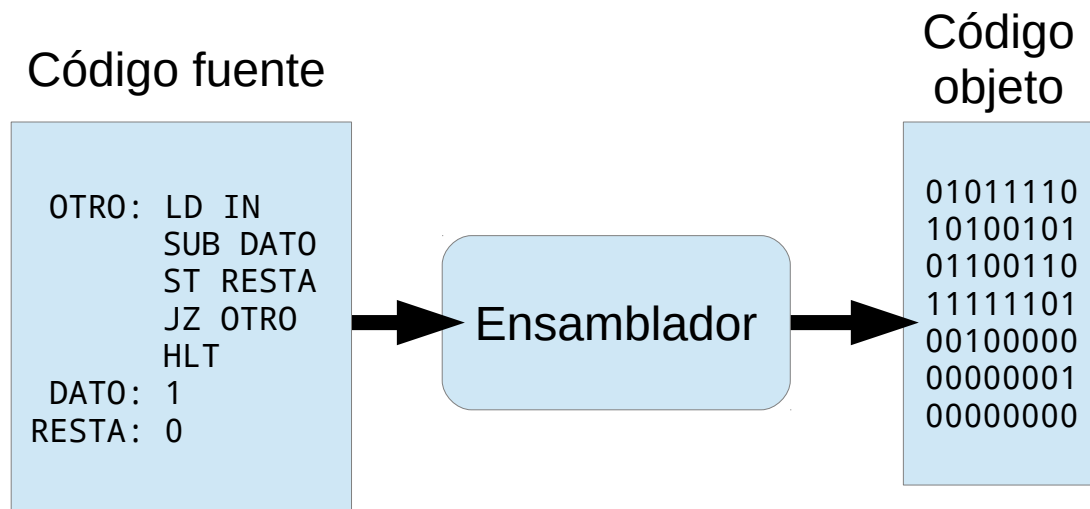
```
L 4C 01001100
D 44 01000100
  20 00100000
4 34 00110100
```

→ 01000100

La instrucción **LD 4** en **ensamblador** para el MCBE equivale a la instrucción **01000100** en **código máquina**. La instrucción en ensamblador está conformada por los caracteres L, D, espacio, y 4. Por lo tanto, mide 4 bytes. Si viéramos este texto escrito en ASCII, se trataría de la secuencia de códigos hexadecimales **4C, 44, 20, 34**. Si los vemos en binario, se trata de los bytes **01001100, 01000100, 00100000, y 00110100**. Obviamente, estos cuatro bytes dispuestos en memoria no son lo mismo que la instrucción en código máquina que necesitamos hacer aparecer en la memoria del MCBE, que no es otra que **01000100**.

Código fuente y código objeto

La tarea del ensamblador es precisamente convertir esos caracteres de las instrucciones en **código fuente** (el que escribe el programador) en los correspondientes bytes de **código objeto** (el que es ejecutable por la computadora).



Contar con un programa ensamblador nos permite disfrutar de ciertas comodidades a la hora de programar en Assembly. Por ejemplo, podemos agregar al programa ensamblador la capacidad de manejar **constantes simbólicas**, **rótulos** o etiquetas en el programa fuente, así como **macros** (secuencias de instrucciones que se repiten frecuentemente). El programa para el MCBE de más arriba se puede hacer aún más fácil de entender y de modificar, si reemplazamos las direcciones y

desplazamientos usados por nombres simbólicos:

OTRO: LD IN	→	01011110
SUB DATO	→	10100101
ST RESTA	→	01100110
JZ OTRO	→	11111101
HLT	→	00100000
DATO: 1	→	00000001
RESTA: 0	→	00000000

El ensamblador del MCBE, que está hecho a medida para esa máquina, conoce por construcción a qué dirección corresponde el rótulo **IN**, y puede calcular por sí solo las direcciones representadas por los rótulos **DATO** y **RESTA**, posiciones de memoria ubicadas a continuación de la última instrucción ejecutable; o computar el desplazamiento para el PC necesario a fin de desviar el control a la instrucción rotulada **OTRO**. Así, el programador no necesita efectuar estos cálculos incómodos y propensos a errores: escribe el texto en Assembly creando o modificando un archivo de texto o **archivo fuente**; luego, llama al programa Assembler que transforma ese archivo de texto y crea otro archivo conteniendo el código objeto, el **archivo objeto** o **archivo ejecutable**.

Para hacer su trabajo, el ensamblador procesa todo el texto del programa fuente, identificando elementos léxicos (como instrucciones, datos, rótulos u otras directivas), detectando posibles errores de programación, y creando tablas de símbolos y estructuras de datos auxiliares. Todo este trabajo del ensamblador es invisible; lo que ve el programador es simplemente el código objeto generado, listo para ejecutar en la máquina.

Para investigar

¿Qué aspecto tienen los programas en lenguaje ensamblador para otras arquitecturas? ¿Cómo es un programa en ensamblador para la PC (arquitectura Intel x86)? ¿Y para otras computadoras que tengan arquitectura MIPS? ¿Qué procesadores llevan las consolas de juegos XBox, PlayStation, Wii, y qué aspecto tiene un programa en Assembler para estos microprocesadores? La misma pregunta para los productos iPhone, iPad, iPod de Apple. ¿Y para los smartphones de las marcas Samsung, Blackberry u otras?

Lenguajes de alto nivel

Los lenguajes de máquina y el Assembler son **lenguajes de bajo nivel**⁴, en el sentido de que sus instrucciones y su forma de programación están condicionados por la arquitectura de la máquina. Pese a que las instrucciones de estos lenguajes son sencillas, la resolución de problemas usando programación en lenguajes de bajo nivel no resulta fácil, ya que el programador necesita especificar con mucho detalle las acciones a realizar. En otras palabras, programar con lenguajes extremadamente simples como los lenguajes de bajo nivel no es difícil, pero resolver problemas con ellos sí, porque requieren bastante esfuerzo de programación. Inversamente, un lenguaje más complejo puede ser más difícil de aprender, pero más expresivo, y reducir el esfuerzo de programación.

El paso siguiente en la evolución de los lenguajes de computación fue liberarse de la

4 http://es.wikipedia.org/wiki/Lenguaje_de_bajo_nivel

correspondencia “1 a 1” en términos de una instrucción de Assembly por una instrucción de código máquina. Teniendo la posibilidad de escribir programas traductores, era posible definir nuevos lenguajes de programación que se orientaran mejor a la forma como pensamos la resolución de los problemas, y no nos mantuvieran atados a la forma como las computadoras procesan los datos.

Así se crearon los **lenguajes de alto nivel**⁵, de los cuales hay varias familias o **paradigmas**⁶. Los lenguajes de alto nivel se parecen lo más posible a un idioma humano, pero sus características varían muchísimo; en especial su **sintaxis** (la forma como se escriben las instrucciones) puede tomar muchas formas. En Wikipedia puede encontrarse una colección muy interesante de programas “Hola, Mundo”⁷, mostrando la gran variedad de formas sintácticas de los lenguajes más conocidos⁸.

Hello, World

El siguiente ejemplo muestra un “Hola, Mundo” en C.

```
#include <stdio.h>
main() {
    printf("Hola, Mundo!\n");
}
```

Y el siguiente es su equivalente en Python.

```
print "Hola, Mundo!"
```

Hay lenguajes de alto nivel multipropósito y los hay especializados en la resolución de determinados problemas. Algunos tienen el objetivo de ser más veloces en su ejecución, y otros, el de ser más fáciles de programar o de mantener. Algunos lenguajes se prestan más para expresar los problemas de Inteligencia Artificial, otros los de la programación científica, otros los de las aplicaciones de negocios, etc. El **lenguaje C**⁹, en particular, es muy empleado por quienes escriben software que debe acceder a características especiales de la máquina, y debe ser veloz y eficiente (como los **sistemas operativos**). Puede decirse que el lenguaje C es posiblemente “el lenguaje de más bajo nivel entre los de alto nivel”.

Los lenguajes de bajo nivel tienen las ventajas de ser muy precisos, ofrecer control total del hardware, y tener traductores fáciles de implementar, pero las dificultades ya mencionadas para colaborar en la resolución de problemas. Además, están diseñados para un hardware determinado, y cada procesador tiene el suyo, es decir, resultan dependientes de la arquitectura de la máquina. Por estos motivos generalmente quedan relegados a la escritura de partes muy específicas del software, que estén fuertemente relacionadas con el hardware (como programas controladores de dispositivos o *drivers*).

Los lenguajes de alto nivel se sitúan en el otro extremo: se programan siempre de la misma manera, sin importar cuál sea la máquina que ejecute los programas. Un lenguaje de alto nivel intenta ser

5 http://es.wikipedia.org/wiki/Lenguaje_de_alto_nivel

6 https://es.wikipedia.org/wiki/Paradigma_de_programación

7 Es tradición en la enseñanza de lenguajes de programación hacer los primeros pasos en cada lenguaje demostrando la forma de crear un programa muy sencillo, que simplemente imprima por pantalla un mensaje. Por motivos históricos, este mensaje suele ser casi siempre el saludo “¡Hola, mundo!”.

8 https://es.wikipedia.org/wiki/Anexo:Ejemplos_deimplementación_del_«Hola_mundo»

9 http://es.wikipedia.org/wiki/Lenguaje_C

independiente de la plataforma y ocultar las particularidades del hardware al programador, permitiéndole concentrarse en el problema. Mientras que los lenguajes de máquina son “orientados al hardware”, los lenguajes de alto nivel son “orientados al problema”, porque su programación se realiza mediante instrucciones cuyo significado es más cercano al problema que el de las instrucciones de máquina.

En un lenguaje hipotético de alto nivel podríamos escribir, por ejemplo, una expresión como **SALDO = DEUDA – PAGO** (*una instrucción*), y la traducción a las instrucciones de máquina correspondientes sería la secuencia **LD 10, SUB 11, ST 12** (*tres instrucciones*).

Traductores de alto nivel

Para ejecutar programas escritos en un lenguaje de alto nivel, sigue siendo necesaria la traducción al lenguaje de máquina. Los programas traductores para estos lenguajes se dividen en dos grandes categorías: **intérpretes**¹⁰ y **compiladores**¹¹. La diferencia entre ambos radica en el modo como realizan la traducción. Si el traductor realiza el ciclo **traducir – ejecutar**, línea por línea del programa (o en base a unidades de traducción más o menos pequeñas), se llama un **intérprete**. Si, en cambio, el traductor realiza la traducción completa del programa fuente a código máquina, y sólo entonces se tiene disponible el código máquina para su ejecución, el traductor se denomina un **compilador**. La situación es parecida a la diferencia que existe entre traductores e intérpretes de idiomas humanos: se llama intérprete (humano) a la persona que traduce oralmente, frase a frase, durante una conversación. En cambio, se llama traductor (humano) al que elabora la traducción completa de un documento y entrega esa traducción sólo una vez que está completa.

En principio cualquier lenguaje puede tener un traductor de una u otra clase, pero habitualmente cada lenguaje se encuentra en alguna de las dos clases. Lenguajes muy conocidos como **Pascal, C, C++, Java, Fortran**, son habitualmente implementados como lenguajes compilados. **Python, Perl, Ruby, PHP**, son lenguajes habitualmente interpretados.

Ciclo de compilación

Al trabajar con un lenguaje de programación, el programador utiliza determinadas **herramientas** de software (programas que utiliza para desarrollar otros programas). Estas herramientas pueden ser programas separados o pueden estar contenidos dentro de un **ambiente de desarrollo integrado (IDE)**.

En primer lugar, el programador utiliza un **editor de textos** para crear o modificar un **archivo fuente** que contiene el **programa en código fuente**.

- Si el lenguaje que utiliza el programador es compilable, entonces **compila** el archivo fuente con un **compilador** para ese lenguaje, que genera un archivo conteniendo **código objeto**. Ese código objeto puede estar incompleto (porque el programador no ha especificado toda la conducta del programa) y requerir otros trozos de código almacenados en **bibliotecas**. Las bibliotecas son archivos que contienen trozos de código objeto usado frecuentemente. El código objeto del programa será **enlazado** con las bibliotecas utilizando un **vinculador** (o **linkeditor**, o **linker**) para generar, finalmente, un **archivo ejecutable**.

¹⁰ [http://es.wikipedia.org/wiki/Intérprete_\(informática\)](http://es.wikipedia.org/wiki/Intérprete_(informática))

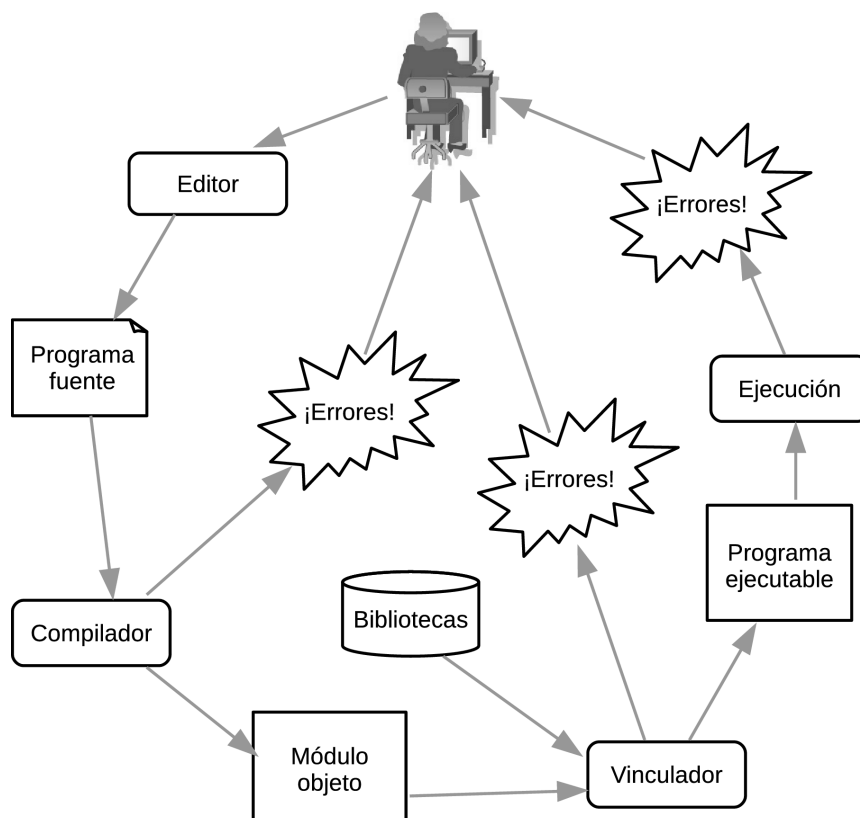
¹¹ <http://es.wikipedia.org/wiki/Compilador>

- Si, por otro lado, el programador trabaja con un lenguaje interpretable, todo el proceso de la traducción de fuente a objeto y la vinculación se harán dinámicamente, al momento de ejecución.

Bibliotecas

Las aplicaciones, aunque pueden tener muchos objetivos diferentes, utilizan el sistema de computación de maneras muy parecidas. Si todos los programadores de programas de música tuvieran que reescribir, por ejemplo, las rutinas que permiten reproducir un archivo de audio en formato **MP3**, o para generar el gráfico de una escena en tres dimensiones, se desperdiciarían muchas horas y se cometerían muchos errores en algo que es difícil y que de todas maneras se hace siempre de la misma forma. Tiene sentido ahorrar esas horas de frustración y esos errores, creando una sola vez, y compartiendo muchas veces, las **bibliotecas de funciones**.

La gran mayoría de las veces habrá que corregir **errores** (“**bugs**”) o defectos de programación o de funcionamiento del programa. Para esto puede ser necesario recurrir a un **debugger** o depurador de programas, que ayuda en la detección y corrección de errores, mostrando el funcionamiento del programa por dentro, o ejecutando paso a paso las instrucciones para encontrar el punto donde el programa falla. Una vez que el programa está listo, el programador **prueba** el resultado ejecutándolo. En este momento y con la información obtenida de la depuración, se vuelve a la fase de edición y el ciclo recomienza, repitiéndose hasta que la calidad del software elaborado sea aceptable.



Cuando el ambiente de desarrollo es integrado (IDE) los diferentes pasos son menos visibles al desarrollador, pero de todas formas ocurren por debajo de la “corteza” o interfaz que presenta el ambiente integrado.

Este **ciclo de compilación** determina entonces varios momentos:

Compilador	Intérprete
<ol style="list-style-type: none"> 1. Edición del programa 2. Compilación del fuente 3. Vinculación del módulo objeto con bibliotecas 4. Ejecución de prueba 5. Depuración 6. Vuelta a 1, tantas veces como sea necesario 7. Puesta en producción 	<ol style="list-style-type: none"> 1. Edición del programa 2. Traducción del fuente, vinculación con bibliotecas y ejecución de prueba 3. Depuración 4. Vuelta a 1, tantas veces como sea necesario 5. Puesta en producción

A este ciclo de producción hay que agregar las etapas posteriores de **mantenimiento** y salida de servicio o **baja** del software.

Errores

- **Errores de compilación**

Cuando el programador comete un error de sintaxis (es decir, no respeta la forma correcta de escribir las instrucciones del lenguaje de alto nivel), el compilador no puede proseguir la traducción hasta que se solucione ese error. La compilación se detendrá con un mensaje de error que el programador debe interpretar. Si el programador olvida el punto y coma final de la sentencia **printf()** en el “Hola, Mundo” que vimos anteriormente, ocurrirá una situación de error de compilación.

- **Errores de vinculación**

Si el programador tipea equivocadamente un nombre de función **printfd()** en lugar de **printf()**, el error no será detectado al tiempo de compilación sino al tiempo de vinculación. Para el compilador, la función **printfd()** podría ser efectivamente un nombre válido, correspondiente a una función cuyo código reside en alguna biblioteca. La detección del error la hará el linker al encontrar que no tiene con qué satisfacer la referencia a una función que ha quedado pendiente.

- **Errores de ejecución**

Los errores al tiempo de ejecución que se destacan en el diagrama anterior abarcan todas las conductas del programa que no se ajustan al funcionamiento esperado, porque el programador ha implementado mal la solución al problema.