2. El código máquina y el lenguaje ensamblador

El código máquina: el lenguaje que realmente entiende el procesador

El **código máquina** es el lenguaje más básico y fundamental que entiende el procesador de una computadora. Este lenguaje consiste en instrucciones codificadas en sistema binario (secuencias de 0s y 1s) que el procesador puede interpretar y ejecutar directamente. Cada tipo de procesador está diseñado para trabajar con un conjunto específico de instrucciones binarias conocido como su **conjunto de instrucciones** (Instruction Set Architecture o ISA).

Por ejemplo, una instrucción en código máquina puede indicar al procesador que realice operaciones como sumar dos números, mover datos de una ubicación de memoria a otra o realizar una comparación lógica.

2.1. La necesidad de conversión desde otros lenguajes

Dado que el código máquina es difícil de leer y escribir para los humanos, los programadores suelen trabajar con **lenguajes de programación de alto nivel**, como C++, Python o Java, que son más comprensibles y abstractos. Sin embargo, las computadoras no pueden entender directamente estos lenguajes.

Para que un programa escrito en un lenguaje de alto nivel pueda ser ejecutado por la computadora, debe ser **convertido a código máquina**. Este proceso se lleva a cabo mediante herramientas específicas como:

- **Compiladores**: Traducen el código fuente completo de un programa a un archivo ejecutable que contiene código máquina.
- **Intérpretes**: Traducen y ejecutan el código línea por línea en tiempo real.
- **Lenguajes intermedios**: Algunos lenguajes (como Java o C#) generan primero un código intermedio que luego es interpretado o compilado en código máquina por una máquina virtual.

2.2. El lenguaje ensamblador: una representación textual del código máquina

El **lenguaje ensamblador** es un lenguaje de programación especial que proporciona una representación más legible del código máquina. En lugar de trabajar directamente con números binarios, el ensamblador utiliza **mnemonics** o abreviaturas textuales para representar las instrucciones del procesador.

Por ejemplo:

- Una instrucción en código máquina como `10110000 01100001` podría representarse en ensamblador como `MOV AL, 61h`.
- Aquí, `MOV` es un mnemonic que significa "mover", `AL` es un registro del procesador, y `61h` es un valor hexadecimal.

Cada instrucción en lenguaje ensamblador **corresponde exactamente a una instrucción en código máquina**, por lo que el ensamblador puede considerarse una traducción directa y textual del lenguaje que entiende el procesador.

2.3. Ventajas del lenguaje ensamblador

1. **Control total**: Permite al programador aprovechar al máximo las capacidades del hardware.
2. **Eficiencia**: Es útil para tareas donde el rendimiento es crítico, como controladores de dispositivos o sistemas embebidos.
3. **Compatibilidad**: Dado que cada instrucción ensamblador mapea directamente a código máquina, es fácil optimizar el código para un procesador específico.

Limitaciones

Complejidad: Programar en ensamblador requiere un conocimiento profundo del hardware y es más propenso a errores.

Especificidad del hardware: Un programa escrito en ensamblador es difícil de portar a otros tipos de procesadores, ya que cada arquitectura tiene su propio conjunto de instrucciones.

En resumen, mientras que el código máquina es el lenguaje nativo del procesador, el lenguaje ensamblador actúa como un puente comprensible para los humanos, facilitando la creación de programas optimizados que interactúan directamente con el hardware. Ambos niveles son esenciales para entender cómo una computadora ejecuta programas y cómo se optimizan para el rendimiento.

2.4 Ejemplo: Sumar dos números

El siguiente programa en ensamblador muestra un programa en ensamblador que carga dos números en los registros eax y ebx y los suma, quedando el resultado en eax y en ecx. La intrucción `int 0x80` le avisa al sistema operativo que el proceso ha finalizado, analizaremos esta comunicación en el siguiente módulo.

```
section .text
    global _start

_start:
    mov eax, 5          ; Cargar 5 en eax
    mov ebx, 3          ; Cargar 3 en ebx
    add eax, ebx        ; Sumar eax + ebx (resultado en eax)
    mov ecx, eax        ; Almacenar el resultado en ecx (opcional)
    int 0x80            ; Terminar
```