# Ingeniería inversa

## Curso práctico

# Ingeniería inversa

## Curso práctico

*Cayetano de Juan*

Ra-Ma®

La ley prohíbe
fotocopiar este libro

Ingeniería inversa. Curso práctico


Editado por:
RA-MA Editorial
Calle Jarama, 3A, Polígono Industrial Igarsa
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: *editorial@ra-ma.com*
Internet: *www.ra-ma.es* y *www.ra-ma.com*
ISBN impreso: 978-84-19444-36-3
ISBN ePub: 978-84-19444-37-0
Depósito legal: M-27757-2022
Maquetación: Antonio García Tomé
Diseño de portada: Antonio García Tomé
Filmación e impresión: Safekat
Impreso en España en noviembre de 2022

## AGRADECIMIENTOS

Cayetano Úbeda Granero, Diego De Juan Fernández, Juan Úbeda Granero, vosotros me marcasteis el camino a seguir y me disteis recursos para caminar sobre él, yo no me he desviado nunca de ese camino, ahora lo ando con mi mujer, Natalia Monfort, Gracias infinitas.

# ÍNDICE

# ACERCA DEL AUTOR

**CAYETANO DE JUAN UBEDA**

Técnico Superior En Informática De Gestión. Auditor sénior en ciberseguridad, colegiado como perito Judicial en informática forense. Diferentes certificaciones, como Cybersecurity Essentials por CISCO, CPHE (Certificado profesional de hacking ético). Cursos finalizados pendientes de certificación, PCAP (Programming Essentials In Python) por CISCO, CyberOps Associate por CISCO. EC Council Hacking Y Ciberoperaciones por CISCO, formador ocupacional por la Junta De Andalucía. Formador de formadores especialidad Tele Formación. Programación VB.NET, Python, lenguaje Ensamblador. Actualmente cursando el CEH.

Ha trabajado como consultor en The Security Sentinel, auditando empresas de nivel internacional como Farmacéuticas y partidos políticos de gran relevancia, organismos oficiales como la Junta De Andalucía, Entidad Nacional De Acreditación. Ha trabajado como formador, impartiendo módulos al servicio Andaluz De Salud, Fundación O.N.C.E…. Su última formación fue por medio de la Fundació IL3- De La Universidad De Barcelona al PDI de Chile, edición 2022 (Estándares corporativos en protección de datos y detección y análisis de datos vulnerables).

En estos momentos dirige el departamento técnico auditor de The Security Sentinel, compaginándolo con proyectos formativos a destacar ARELANCE, impartiendo curso de ciberseguridad.

# PRÓLOGO

En este libro encontraréis, explicado de manera muy dinámica, la programación de aplicaciones, librerías, drivers y todo lo que os imaginéis usando el lenguaje Ensamblador, el lenguaje más próximo al "hierro" e ingeniería inversa.

Podréis practicar, sin necesidad de conocimientos previos, cada capítulo mediante ejercicios prácticos que os ayudará a asentar los conocimientos adquiridos, de manera sencilla y explicados paso a paso.

Conocer el código máquina os abrirá un mundo nuevo en muchos campos de la informática, veréis los programas de otra manera y estaréis deseando explorarlos.

En primer lugar, seréis capaces de crear aplicaciones más eficientes debido a que los lenguajes interpretados necesitan convertir el código creado en lenguaje máquina, haciendo que el código del programa crezca de manera innecesaria.

Además, os permitirá poder hacer ingeniería inversa en ejecutables ya creados y os permitirá entender el funcionamiento de un programa y detectar fallos, puertas traseras, desbordamientos de buffer, etc.

Sobre el autor, Cayetano De Juan, solo puedo deciros que es un apasionado de la tecnología y un gran profesional, experto en seguridad informática y programador con más años de experiencia que páginas tiene el libro.

Es un honor poder escribir este prólogo, como alumno del curso de ensamblador, en el que está basado este libro, os puedo decir que al comenzar no sabía si sería capaz de disfrutar programando en ensamblador ya que en la carrera de ingeniería informática había dado clases de ensamblador y el recuerdo no era muy agradable.

La manera de explicar, y que desde el primer momento ya te pones a practicar las lecciones aprendidas, hace que la curva de aprendizaje sea muy rápida y nada más acabar un tema ya tienes en la mente programas que podrías mejorar con lo aprendido.

Recomiendo encarecidamente este libro, no es uno de esos libros de informática que tenemos en la estantería y que le tenemos que quitar el polvo los sábados, es un libro que tendréis encima del escritorio para poder consultarlo a diario.

Raül Rivero.
Ingeniero Informático.

# INTRODUCCIÓN

La Ingeniería Inversa, se refiere al estudio detallado de las funciones del malware, paso a paso, con el fin de descubrir cuál es el código responsable por su funcionamiento. Es una de las disciplinas más gratificantes dentro de la seguridad informática.

Esta obra te explica de forma secuencial como poner en práctica esta materia a través de explicaciones claras y didácticas, acompañados de ejemplos y ejercicios de autoevaluación.

Para facilitar la asimilación de los contenidos se ha dividido en dos partes:

En la primera parte aprenderás el lenguaje de más bajo nivel legible que existe, el lenguaje **Ensamblador**, y lo harás comenzando desde cero con este orden:

- A moverte por el mundo de las API de Windows.
- A enlazar Ensamblador con lenguajes de alto nivel como Python y VB.Net.
- A crear su propia Shell Inversa en Ensamblador y conectarla con Python.
- A crear sus propias DLL.

En la segunda parte asimilarás a interpretar los programas compilados y aprenderás:

- A crear ficheros Binarios PE.
- A poner puntos de ruptura (memoria, anular objetivos por API, por cadenas, etc.).
- A crear sus propios parches o cambios en un binario.
- A cifrar texto por XOR.
- A reconstrucción de código intermedio.
- A analizar un binario contaminado por Malware real.

Con este libro obtendrás los conocimientos necesarios para poder usar desensambladores y depuradores como IDA Pro, OllyDBG, Inmmunity Debugger, WinDBG, etc.

Además, con esta obra tendrás acceso a 40 videos y supuestos prácticos descargables desde la web del libro que complementan al contenido y que están indicados en el libro con los iconos  y  .

# Parte 1

## XXXXXX

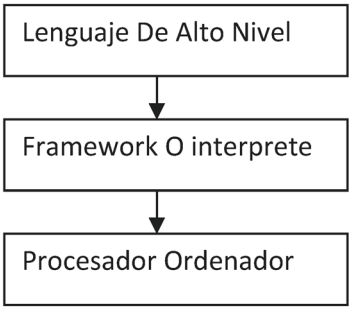
# 1

# CONCEPTOS BÁSICOS Y EXPECTATIVAS DEL CURSO

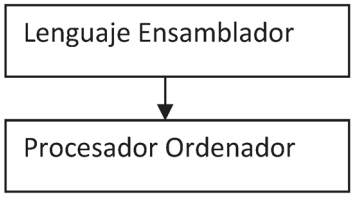
**Video 1**

El lenguaje ensamblador es un lenguaje de bajo nivel, se compone de una serie de instrucciones básicas para los microprocesadores. Este lenguaje no necesita un Framework para poder ejecutar los programas realizados con él, como por ejemplo Vb.net. Esto quiere decir que nuestros ejecutables podrán cargarse o ejecutados sin necesidad de una plataforma detrás que haga la interpretación.

Lenguajes de Alto Nivel:

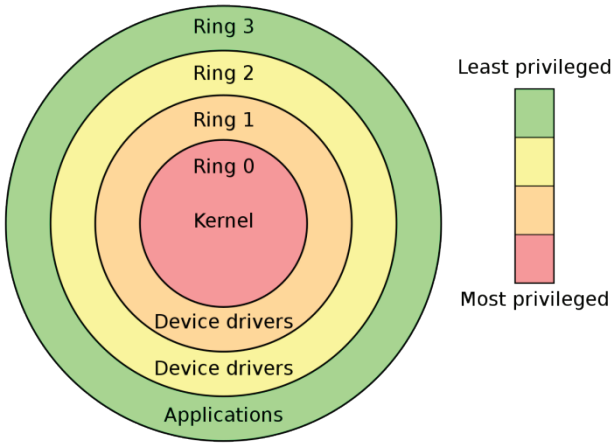


## Lenguaje Ensamblador (Lenguaje de bajo nivel)




Desde el comienzo de los sistemas operativos, empezando por el Ms-Dos. El lenguaje ensamblador ha estado presente, con este lenguaje se podría controlar absolutamente todo el ordenador, memoria, disco duro, periféricos. De hecho otros lenguajes como C, realmente provienen del lenguaje ensamblador. Antiguamente en un sistema Ms-Dos con el lenguaje ensamblador se podía programar usando las interrupciones del sistemas, en los sistemas operativos actuales como Windows, que es en lo que se apoya este curso, el leguaje ensamblador usaría las denominadas **API** de Windows que vienen a ser esas interrupciones de Ms-Dos.

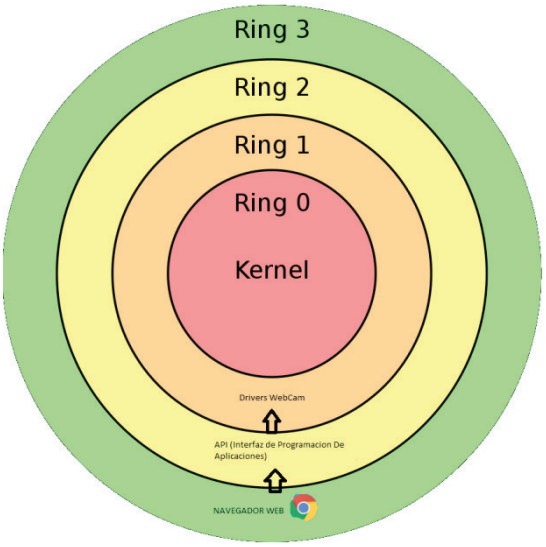
El esquema de desarrollo de Windows quedaría algo parecido a esto:




Windows dispone de un sistema de protección del **Kernel**, basado en anillos, de esta forma se asegura de que ningún programa mal intencionado pueda dañar su núcleo. Los anillos se pueden comunicar entre sí mediante "puertas" que controla el propio sistema operativo.

Supongamos el siguiente ejemplo:

Tenemos un ordenador con una webcam, la webcam tiene instalado en el sistema sus drivers necesarios para poder funcionar. Desde el navegado queremos realizar una video conferencia, usado esa webcam. Bien, pues el navegador estaría en el anillo tres, los drivers de control de la cámara estarían en el anillo/sdos o uno. Algo así:




| |
|---|
| Lenguaje De Alto Nivel<br>Nivel 4 |
| Lenguaje Ensamblador<br>Nivel 3 |
| Sistema Operativo<br>Nivel 2 |
| CPU<br>Nivel 1 |

- **Nivel 1:** En este nivel esta la unidad central de proceso, que resumiendo se componen de pequeñas funciones en lenguaje máquina que introduce el propio fabricante, sumar, restar.
- **Nivel 2:** El sistema operativo se compone de diferentes funciones ya más adecuadas al programador o usuario del mismo, con las que se pueden crear carpetas, archivos, ejecutar programas. Estas funciones se traducen a código maquina las cuales pasarían al **Nivel 1**.
- **Nivel 3:** Los lenguajes de programación, se encuentran siempre por encima del sistema operativo, estos lenguajes como ya hemos comentado, anteriormente, pueden ser lenguajes de Alto o Bajo nivel. En el caso del Lenguaje Ensamblador es de bajo nivel, esto quiere decir que este lenguaje usa instrucciones básicas que pueden ser interpretadas directamente por el **Nivel 1**, y al mismo tiempo puede

usar funciones propias del sistema operativo del **Nivel 2**, en nuestro caso estas funciones propias del sistema operativo serán las **APIS**. Los programas realizados en ensamblador, una vez compilados se traducen en su totalidad a lenguaje de código máquina.

- **Nivel 4:** Los lenguajes de Alto nivel, como C++, Vb.Net, Java, etc. Se compone de framework o poderosas librerías que agilizan y nos simplifican el tratamiento por ejemplo de acceder a un disco duro, cámara web. Los compiladores de estos lenguajes traducen los programas realizados con ellos, a programas de **Nivel 3**, y a su vez traducen en código de **Nivel 3**.

En este curso empezaremos desde cero, para conocer la base del lenguaje ensamblador bajo Windows 32Bits. Se presentará el **EASY CODE**, un Software con entorno visual, para poder desarrollar en lenguaje ensamblador, este Software creado por Ramón Salas, nos ayudará a crear nuestros programas en ensamblador de una forma más sencilla y amena.

Una vez terminado el curso, deberíamos haber obtenido los siguientes conocimientos:

- Base y desarrollo de proyectos bajo lenguaje ensamblador.
- Conocimientos sobre el funcionamiento interno de Windows.
- Conocimientos sobre aplicaciones de Windows.
- Conocimientos sobre las API'S de Windows.

## 1.1 FUNCIONAMIENTO DE WINDOWS, MENSAJES Y EVENTOS

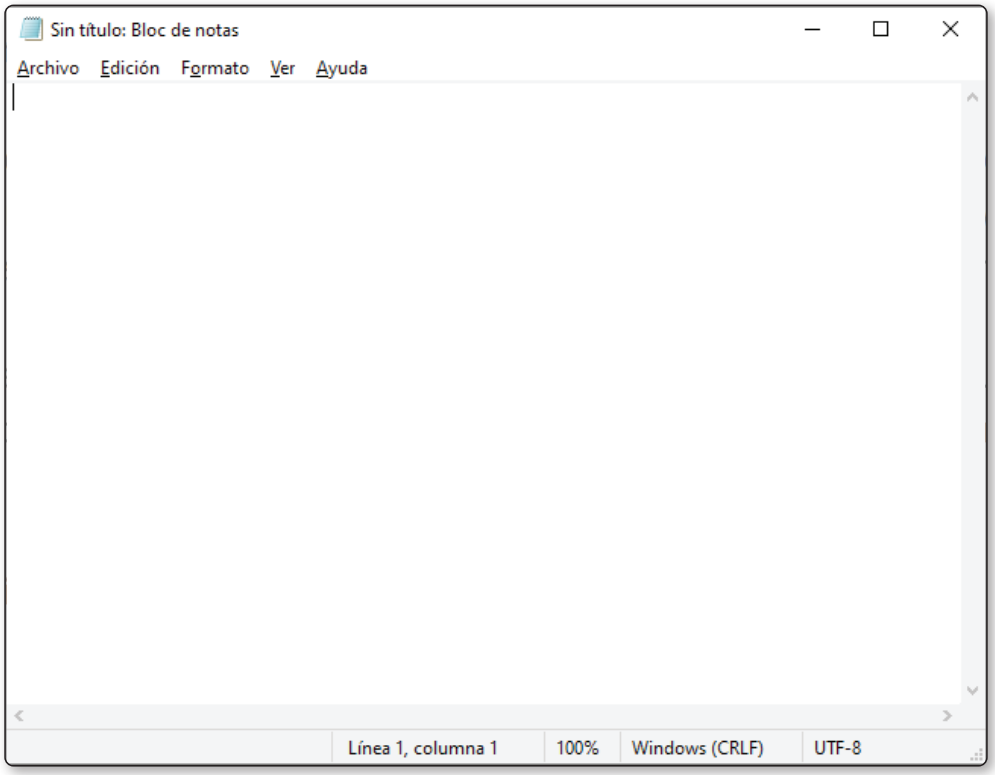
Si alguna vez has desarrollado un programa, por muy pequeño que sea, con algún lenguaje de alto nivel, como Vb.net, Java, etc. Y si este leguaje de alto nivel, es un lenguaje visual, donde se parte de un formulario, al que le vamos agregando controles personalizados como botones, cajas de texto. Pues toda esta integración al final de traduce en las **API de Microsoft Windows**.

Básicamente el funcionamiento interno de Windows se traduce en dos principios:

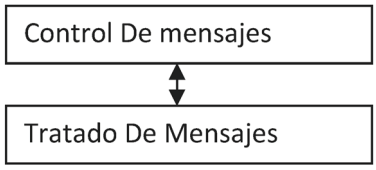
- Para el sistema, todo parte de una ventana (un simple botón, para Windows es una ventana, en general cualquier objeto, es tratado como una ventana).
- Las ventanas y objetos que la componen se comunican por medio de mensajes.

Todos los objetos, o ventanas de Windows se identifican mediante un **handle**. Y a su vez, de un procedimiento que es capaz de interceptar mediante mensajes como se tiene que comportar.

Por ejemplo si abrimos el Bloc De Notas, veremos seguramente una pantalla parecida a esta:




Si pudiéramos ver el código del núcleo de esta aplicación, sería algo parecido a esto.




El control de mensajes, se comportaría como un bucle, donde está siempre recibiendo mensajes por parte de la aplicación, si movemos el ratón dentro de nuestro programa Bloc De Notas, ese mensaje será recibido en el control de mensajes, acciones de teclado, como la pulsación de una tecla, apertura del menú. El control de mensajes a su vez los procesa en el segundo cuadro Tratado De Mensajes.

## 1.2 ¿PARA QUÉ PODEMOS USAR EL LENGUAJE ENSAMBLADOR?

El lenguaje ensamblador, al ser un lenguaje de bajo nivel, nos va a permitir en resumen dos cosas:

- Rapidez.
- Poder crear pequeños procesos o utilidades que los lenguajes de alto nivel, no nos dejan o es demasiado engorroso hacerlo.

Otra de las ventajas del lenguaje ensamblador es que no tiene que pasar por ningún framework para poder ejecutar su código, una vez compilado, el lenguaje ensamblador esta convertido en lenguaje máquina.

Entender este lenguaje, nos abre también una puerta a la ingeniería inversa o a desensamblar programas compilados con lenguajes de alto nivel.

## 1.3 NUMERACIÓN Y CÁLCULO ARITMÉTICO

Antes de meternos de lleno con el lenguaje ensamblador, explicare unas pequeñas nociones de cómo cuentan los ordenadores. Esto debe parecer bastante simple: 1, 2 ,3, 4, 5, 6, 7, 8, 9, 10, 11. Esta numeración la entenderíamos bien 1 más 1 es 2, mas 1, 3. Bien pues los ordenadores no cuentan así, lo hacen en formato binario (base 2), para contar hasta 3 lo haría de esta forma: 1, 10, 11. El sistema binario es un sistema numérico que solo tiene dos dígitos, el 0 y el 1. Al contrario de los 10 dígitos que tenemos en nuestro sistema decimal (base 10).

El microprocesador de nuestro ordenador, cuenta en este modo binario, pero como para nosotros sería un poco complejo el tratar con ciertos números y traducirlos a números binarios, vamos a usar algo intermedio, que serían los números hexadecimales. Los números hexadecimales, al menos nos van a ayudar a tratar con los números binarios recortando su longitud.

### 1.3.1 Números hexadecimales

Para intentar hacer más ameno, el aprendizaje de los números Hexadecimales, vamos a usar un viejo programa, que viene de Ms-Dos **"Debug**". La evolución de este programa, es "**WinDBG**".

El nombre de "**Debug**" seguro que ya nos dice algo, "Bugs" informáticamente, son bichos… que serían los fallos de un programa.

**Debug**, nos ayudara a ejecutar programa instrucción por instrucción, esto seguro que para los que os gusta la ingeniería inversa, os llamara un poco la atención, pues pudiendo ver cada una de las líneas de ejecución de un programa podríamos estar analizando o depurando dicho programa.

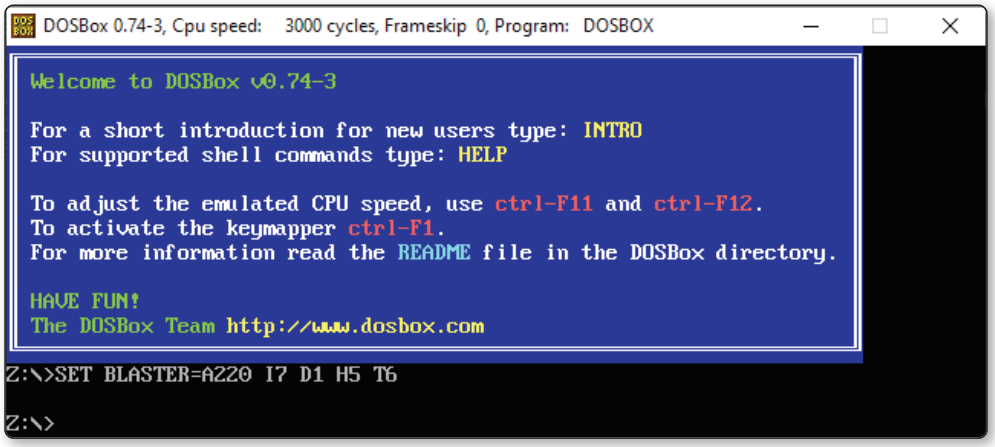
Para poder usar **Debug**, en nuestro sistema operativo actual, Windows 8.1, Windows 10 o Windows 11. Tendremos que descargar dos programas.

- El primero una especie de simulador de Ms-Dos.
- El segundo el propio Debug.exe.

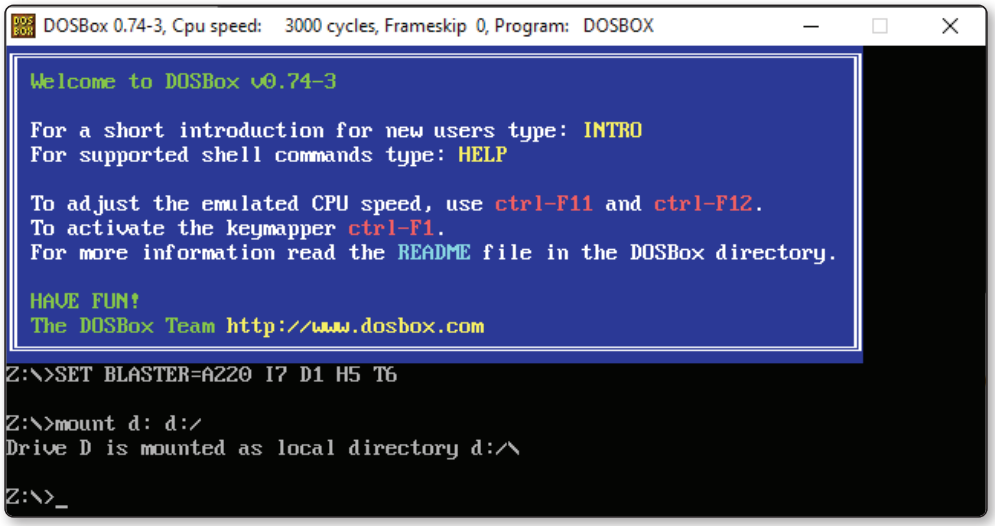
*https://www.dosbox.com/download.php?main=1*
*http://www.mediafire.com/file/bopdmu16tav8thg/debug.zip/file*

Una vez instalado el primer programa, vamos a descomprimir nuestro "Debug.exe" en la unidad D: (en su caso la unidad que corresponda)

Una vez descomprimido, ejecutamos el primer programa "**DosBox**".

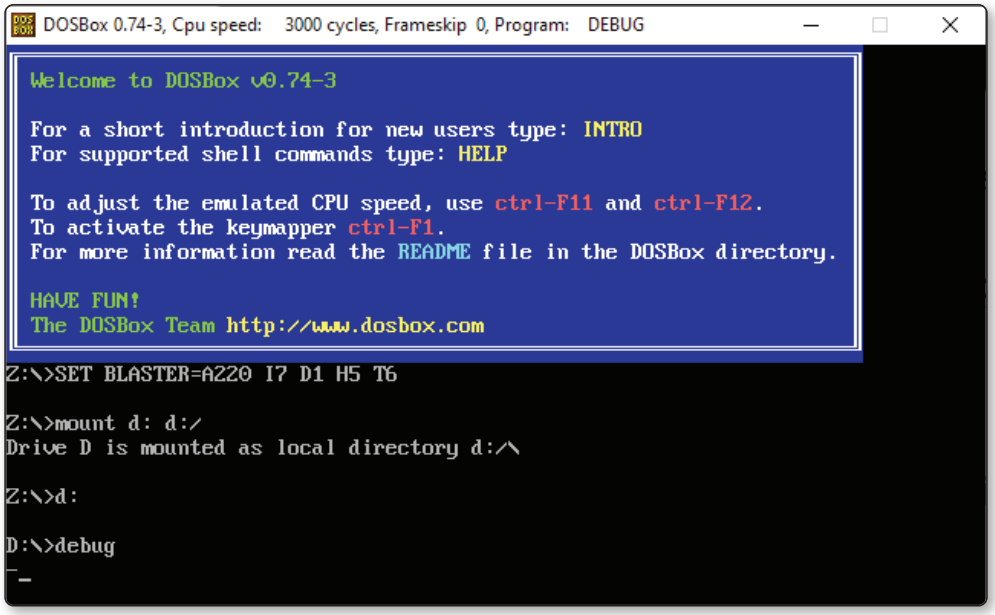



Y vamos a montar la unidad D:, que es donde está nuestro debug (en su caso la unidad que corresponda).



De esta forma, ya tendríamos acceso a nuestro programa "Debug.exe"

Pasándonos a la unidad D: y tecleando Debug. Ya estaríamos dentro del mismo.




Si se fija en la foto superior, al ejecutar el **Debug**, solo aparece un guion. Bien esto sería correcto, el programa estaría esperando a que empecemos a meter comandos. Para salir de **Debug**, y regresar al Ms-Dos, vamos a usar la letra "**Q**".

Veamos otro comando de **Debug**, pero ya aplicado a lo que nos interesa aprender, la suma de dos números hexadecimales.

Vamos a decirle a **Debug**, que sume 2 + 3 = 5.

Pondríamos lo siguiente en nuestro **Debug**.

```
-H 3 2
```

**Debug**, nos devolvería:

```
0005  0001
```

En general se vería algo así:

```
D:\>debug
-H 3 2
0005  0001
-
```

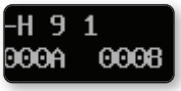
El comando **H** en **Debug**, nos va a permitir, realizar operaciones de aritmética básica, como sumas y restas.

El comando nos devuelve por un lado la suma, 0005 y la resta 0001 de los números 3 y 2. En principio estas dos operaciones, son las mismas que en nuestro sistema Decimal, 3 + 2 = 5 y 3 – 2 = 1.

¿Qué sucedería si volvemos a escribir el comando H, pero esta vez ponemos primero el 2 y luego el 3?


Nos fijamos, que la suma la sigue haciendo según nuestro sistema decimal, y nos devuelve 5, pero la resta nos devuelve **FFFF**. Pues en realizad es correcto pues **FFFF**, es la referencia hexadecimal de -1.

Vamos a ver cómo funciona el comando **H** con números más grandes. Vamos a sumar 9 + 1.




Fíjese que en la suma a devuelto **A** y en la resta 8, esto nos estaría diciendo que el número 10 en hexadecimal es **A**.

La palabra Hexadecimal viene de “hexa” (6) más “deca” (10) las cuales sumadas serían 16.

| Decimal | Hexadecimal |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | A |
| 11 | B |
| 12 | C |
| 13 | D |
| 14 | E |
| 15 | F |

**Sistema Base Dígitos posibles**

| Sistema | Base | Dígitos |
|---|---|---|
| Binario | 2 | 0 1 |
| Octal | 8 | 0 1 2 3 4 5 6 7 |
| Decimal | 10 | 0 1 2 3 4 5 6 7 8 9 |
| Hexadecimal | 16 | 0 1 2 3 4 5 6 7 8 9 A B C D E F |

¿Qué pasaría si sumásemos números más grandes? Por ejemplo que pasaría si sumamos un 1 al número 9, en base 10, el resultado sería 10.

¿Qué pasaría si sumásemos en hexadecimal 1 a F?, Hacer la prueba con el **Debug**, pero veréis que la suma de 1 + F = 10. ¿Cómo que 10?, pero si coincide con nuestra suma del número decimal!!!. Bien pues para saber y poder diferenciar los números decimales de los hexadecimales, siempre a los números hexadecimales, les pondremos una **h** al final del mismo. Así que nuestro número 10 en hexadecimal quedaría 10h.

Ahora vamos a ver cómo podemos convertir números de un sistema a otro. Sabemos que 10h, en realidad es 16 en sistema decimal.

Intentemos pasar A5h a decimal.

Vamos a dar antes, una pequeña clase de matemáticas, algo simple para no aburrirle mucho. Por ejemplo el número 856, ¿sabes qué significa?

8 centenas.
5 Decenas.
6 Unidades.

8 * 100 = 800
5 * 10 = 50
6 * 1 = 6

856 = 856

Ahora vamos a hacerlo con nuestro número hexadecimal A5h.

A - 10 * 16 = 160
5 - 5 * 1 = 5

A5H = 165

Si se fija, en la primera conversión del número 856, se usó 1, 10, 100. En la segunda conversión para el numero hexadecimal, usamos el 1, 16. En el primero base 10, en el segundo siempre base 16.

## 1.3.2 Conversiones decimal – hexadecimal

Vamos a ver ahora como convertimos números decimales en hexadecimales.

Pues seguimos usando los mismos valores que aprendimos en el colegio… pero como vamos a trabajar con numero hexadecimales, tendremos que hacer divisiones en base 16. Ejemplo quiero pasar el número 493 de decimal a hexadecimal.

493 / 16 = 30 Y el Resto 13 En Hexadecimal Dh

El resultado (cociente) de la división 493 / 16 = **30,** se vuelve a dividir entre 16.

30 / 16 = 1 Y el Resto 14 En Hexadecimal Eh

….

1 / 16 = 0 Y el Resto 1 En Hexadecimal 1h

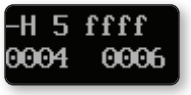
493 = 1EDh

### 1.3.3 Números negativos

**Video 2**

Si recuerda, al principio de todo esto, vimos que -1 decimal en hexadecimal es FFFFh. Si pasamos FFFFh a decimal, nos dice que es 65.535.

Si volvemos a probar con nuestro **Debug**. Veremos que el sí lo trata como -1.




Vemos que 5 – 1 (ffffh) = 4.

### 1.3.4 Bits, bytes, palabras y sistema binario

Bueno hasta aquí, hemos visto un poco, las diferentes bases, decimal/hexadecimal, hemos pasado de una base a otra. Ahora toca meternos de lleno dentro del procesador, como trata el procesador a estos números decimales o hexadecimales. Bien todos estos números se pasan a base 2 o sistema binario. Y realmente es lo que entiende nuestro procesador.

El sistema binario como ya hemos visto solo conoce 0 y 1, los números en formato binario, suelen ser cadenas muy largas de 0 y 1.

Veamos un ejemplo cojamos el numero binario 1011, este número es el equivalente al 11 en decimal y al Bh en hexadecimal.

Para comprobarlo, tenemos que multiplicar 1011 por la base del número, que sería 2, potencias de 2:

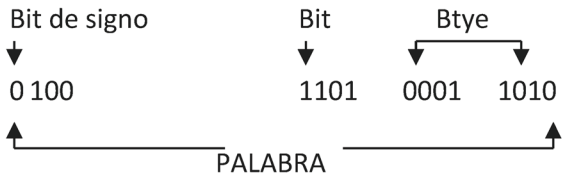
$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$

El resultado una vez obtenida la potencia.

```
1 * 8 = 8
0 * 4 = 0
1 * 2 = 2
1 * 1 = 1
-----------
1011  = 11 o en hexadecimal Bh
```

Cada digito binario, 0 o 1 se conoce como "**Bit**", cada ocho dígitos binarios juntos, 1011 1111, se conoce como "**Byte**", cada dieciséis dígitos binarios juntos, se conoce como "**Palabra**". Realizadas las presentaciones, no olvide esto, pues lo usara en la definición de variables y a lo largo de toda su vida con el lenguaje ensamblador.

| Decimal | Hexadecimal | Binario |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |




Observe que dos dígitos hexadecimales son un **Byte**, y cuatro dígitos hexadecimales serían una **Palabra**.

### 1.3.5 Registros como variables

Vamos a ahondar un poco más dentro del procesador o CPU de nuestro ordenador. Y vamos a seguir usando nuestra primera herramienta ya mencionada **Debug**. Seguramente estará pensando, ¿Por qué usar una herramienta de 16 bits, que ya ni se usa? Bien, la respuesta es simple, quiero hacerle ver, básicamente el inicio del lenguaje ensamblador, de una forma resumida, y pienso que esta metodología le ayudara a comprender la forma de trabajar en ensamblador bajo un Windows 10 por ejemplo.

Volvamos a nuestro procesador. Nuestro procesador dispone de una serie de "**Registros**" fijos, que los utiliza de diferentes formas, los "**Registros**" del procesador para quien venga de algún lenguaje de programación de alto nivel, serían como variables, pero no exactamente igual.

Entremos de nuevo en nuestro **Debug**. Le vamos a preguntar a Debug, que nos muestre los registros de nuestro procesador. Para eso, vamos a usar el comando "**R**".

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
```

Nos vamos a centrar en los cuatro primeros bloques de registros, que son: AX, BX, CX, DX. Comentarle, que los 4 bloques de registros que está viendo son de 16Bits, puesto que **Debug**, funcionaba para **16Bits**. El comportamiento es el mismo que verá más adelante cuando volvamos a mencionar estos registros en **32Bits**.

El resto de registros, SP, BP, SI, DI, DS, ES, SS, CS, IP, los veremos más adelante, serían registros de tratamiento especial, para controlar la pila, segmento de datos, segmento de código. Pero no nos centremos en estos registros ahora, como decía, vamos a centrarnos en los cuatro registros mencionados anteriormente.

En el punto anterior, vimos que una palabra se componía de **16Bits**, recordar que un bit es un 0 o un 1 en binario, si nos fijamos los Registros, AX, BX, CX, DX, se componen de 4 dígitos en hexadecimal, como vimos anteriormente, un digito hexadecimal, en binario serían 1111 (cuatro bits), con lo cual, si AX = 0000, quiere decir que podría contener dentro cuatro bloques de dígitos binarios, como por ejemplo 1111 0000 1111 0000, que esto a su vez dijimos que se llamaba **Palabra**.

Así que AX, BX, CX, DX, son registros de **16Bits**.

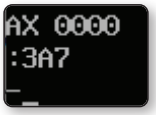
El comando **R** de **Debug**, no solo nos permite ver los registros si no que también nos permite cambiar su contenido.

Si en nuestro **Debug**, ponemos **R AX,** Veremos que nos muestra el contenido actual y nos permite cambiarlo. Vamos a hacerlo.

```
-R AX
AX 0000
:_
```

Fíjese, que al poner **R** AX, y dar al enter, nos aparece el contenido actual de AX, y justo debajo aparecen dos puntos ":", **Debug**, está esperando que introduzca el nuevo valor.

Vamos a introducir por ejemplo: 3A7h




Si ponemos en Nuestro **Debug**, otra vez el comando **R**, veremos que el nuevo contenido para el registro AX es 3A7h.

```
-R
AX=03A7  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:0000=CD
```

Bien, vamos a avanzar un poco más, anteriormente vimos como **Debug**, mediante el comando H, era capaz de realizar una suma y resta al mismo tiempo de dos números hexadecimales. Ahora en vez del comando H propio de **Debug**, vamos a usar un comando propio de nuestro microprocesador, **ADD**.

Básicamente lo que va a ver ahora, será su primer programa en ensamblador desde el **Debug**.

Lo primero que vamos a hacer, es cargar o armar otro registro, con el valor a sumar al registro AX, bien, pues vamos a armar al registro BX, con el valor 92Ah. Lo hacemos igual que lo hicimos con el Registro AX, mediante el comando **R** BX.


Nos saldrán los dos puntos, esperando a meter el nuevo valor, y metemos 92Ah.

Si volvemos a poner solo el comando **R** de **Debug**, veremos cómo los registros AX y BX, están armados con AX=03A7h y BX = 92Ah.

```
-r
AX=03A7  BX=092A  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL                      DS:092A=00
```

Bien, los registros están armados, ahora tenemos que fijar dentro de **Debug**, en que parte de la memoria voy a poner mi instrucción **ADD**, para que me sume los dos registros, AX + BX.

**NOTA**

Preste mucha atención a esta parte porque esto lo utilizara durante todos los programas que realice en Ensamblador.

Con el programa **Debug**, básicamente estamos viendo la memoria actual de nuestro ordenador, todos sabemos que los programas se ejecutan en la memoria del mismo, ¿Pero cómo decidimos donde empieza mi primera línea de código?, ¿Cómo le digo a **Debug**, donde está mi primera línea de código a ejecutar?

Pues bien, nuestro ordenador divide la memoria en segmentos, tendremos un segmento de datos, un segmento de pila, un segmento de código. Pero esto lo veremos un poco más adelante. Ahora mismo nos quedamos con la teoría básica, de que nuestra suma tiene que estar en alguna parte de la memoria y para ello vamos a usar nuestro **Debug**, y la instrucción **ADD**. Quiero comentarle que la instrucción **ADD**, es una instrucción propia de nuestro procesador, y se compone de dos bytes. Que son **01h** y **D8H**, así es como nuestro procesador identifica la instrucción **ADD**.

Para poder ver e introducir valores en memoria, vamos a usar el comando **E** de nuestro **Debug**.

Lo primero que vamos a hacer es meter nuestra instrucción **ADD**, en un segmento de memoria, y dentro de ese segmento, en una posición, y lo haremos de la siguiente forma:

```
-E 100
073F:0100  00.01

-E 101
073F:0101  00.D8
```

Yo he tecleado el E 100, le he dicho que dentro del segmento que proporciona el propio **Debug**, en la posición 100, quiero meter el primer byte de mi instrucción **ADD** que es 01h.

Seguidamente he vuelto a poner el comando E, de **Debug**, y como posición he puesto 101, puesto que nuestra instrucción **ADD** ocupa dos bytes, dentro de la posición 101, meto D8h.

No se preocupe si su **Debug**, muestra un segmento diferente al mío, pues será lo más seguro, usted solo mantenga la dirección dentro de ese segmento que la hemos fijado para el primer byte en 100h y para el segundo byte en 101h.

Tampoco se preocupe si en su posición de memoria dentro de su segmento, en este caso la 100h o 101h, le aparece en vez de 00. Le parece algún valor, recuerde que está viendo posiciones de memoria de su ordenador, si antes cargo algún programa seguramente este viendo el código de ese programa.

Si ponemos el comando que ya conocemos **R** de nuestro **Debug**, veremos cómo queda nuestro pequeño programa de suma de registros.

```
-R
AX=03A7  BX=092A  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 01D8          ADD     AX,BX
```

**Fíjese porque aquí hay una cosa muy interesante**, podemos ver nuestros registros AX y BX, que siguen armados, con los valores que les pusimos para sumar y en la **tercera línea**, vemos perfectamente colocada nuestra línea de código, con la instrucción **ADD**, diciendo que sume AX, BX.

```
073F:0100 01D8          ADD     AX,BX
```

Vamos a analizar esa tercera línea:

Lo primero es el segmento asignado por el propio **Debug**, este segmento seguramente será diferente en su ordenador, pero no nos preocupamos. Nos fijamos también, en la dirección de memoria del segmento, esta si debiese de coincidir con la mía, vemos que está en la posición 100h. No fijamos bien, también, que después de la posición 0100 está el código de nuestra instrucción **ADD**, que es **01D8**, eso es en binario **código máquina**, es a lo que luego se traduce cualquier programa. **01D8**h estos códigos que para nosotros son valores hexadecimales que no comprendemos, para el procesador es una instrucción de suma.

Pues hasta aquí, nuestro procesador ha entendido, que estamos haciendo un programa que va a sumar el registro AX y BX, y adelanto, que la instrucción **ADD**, cuando sume los dos registros dejara el **resultado en AX**. No se preocupe, porque estoy seguro que tienen muchas preguntas por hacer y alguna que otra laguna, pero lo iremos viendo y solucionando conforme vayamos avanzando.

Una pregunta que seguro que le ronda la cabeza, es. Bueno bien, el **Debug**, ha propuesto un segmento  y yo he fijado su dirección, pero…¿Cuándo le diga que ejecute el programa, como sabe **Debug**, la dirección dentro del segmento que yo he fijado?

Bien pues ha llegado el momento de conocer el primer segmento dentro de nuestro lenguaje ensamblador, si se fija en la segunda línea:

```
-R
AX=03A7  BX=092A  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 01D8          ADD     AX,BX
```

Tenemos un Registro que se llama **CS**, este registro siempre apuntará al **segmento de código**, en otra palabras le estará diciendo al ordenador en que parte de la memoria se ha cargado nuestro cuerpo o código del programa.

¿Bien, ya está claro que **CS**, guarda la dirección del **segmento de código**, pero dentro de ese segmento de código, nosotros fijamos una dirección para nuestra primera instrucción ADD, que era 100h? Correcto, a esa dirección se le llama **Desplazamiento**, y el desplazamiento siempre estará indicado en el Registro **IP**.

```
-R
AX=03A7  BX=092A  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 01D8          ADD     AX,BX
```

Registro IP, apuntado a nuestro desplazamiento de código.

¿Qué listo ha sido **Debug**!!, como ha sabido que mi desplazamiento iba a ser 0100h?

La verdad es que cuando se arranca **Debug**, el fija siempre ese desplazamiento en 0100h, así que yo me he aprovechado de este conocimiento para que el registro **IP**, ya apareciera fijado.

Bueno pues como parece que todo está en orden, podíamos decirle a **Debug,** que ejecute nuestro mini programa de suma. Para ejecutar vamos a usar un comando de **Debug**, el comando "**T**", este comando empezando en la primera línea de código que es la que apunta el registro **IP**, junto con el segmento de código que está en el registro **CS**, ejecutara la primera línea y se pasara a la siguiente línea y así sucesivamente. .

```
-T

AX=0CD1  BX=092A  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV UP EI PL NZ AC PE NC
073F:0102 0000          ADD     [BX+SI],AL                      DS:092A=00
```

Al ejecutarlo, verá en que el registro AX, se ha almacenado el resultado de la suma, tendrá que estar viendo AX = 0CD1h.

**Ojo**!, porque si vuelve a poner en el **Debug**, "**T**" y le da al enter, ejecutara la siguiente línea que se encuentre, y nuestro programa solo disponía de una línea!!, así que mejor no hacerlo.

Fíjese en los registros, fíjese, que **CS** que apunta al segmento de código, sigue correcto, pero **IP** que apunta al desplazamiento, o en definitiva a la línea de código a ejecutar esta en 0102h.

Esto es totalmente normal, el comando "**T**" de **Debug**, si lo vamos llamando iría ejecutando línea por línea.

¿Qué puedo hacer si quisiera, volver a ejecutar la suma?

1. Volver a configurar el registro CS.
2. Volver a configurar el registro BX.
3. Volver a configurar el registro IP, con su desplazamiento de inicio de línea de código a 0100h.

La correcta es la 3, pruébelo.

Hemos visto que una **palabra** está formada por dos bytes, los registros de uso general, como AX, se pueden dividir en dos bytes.

Por ejemplo AX = 4D1Ah.

Se pude dividir en:

AH = 4D
AL = 1A

En binario quedaría algo así:

AH = 0100 1101
AL = 0001 1100

AH, se guarda el byte más alto y en AL se guarda el byte más bajo.

## 1.4 RESUMEN AUTOEVALUACIÓN

Recuerde que: El lenguajes ensamblador nos dará posibilidad de hacer programas:

- Rápidos.
- No pesados en tamaño, en bytes.
- Podremos acceder y gestionar memoria, dispositivos.
- Es un lenguaje de bajo nivel.

Nuestro procesador solo entiende el código maquia o binario, solo se mueve con 0 y 1. Para que la programación en ensamblador pueda manejar valores binarios usamos el sistema hexadecimal.

1 bit = 0 o 1.
8 bits = Byte.
16 bits = Palabra.

Los registros básicos de nuestro procesador, son:

AX, BX, CX, DX

Son registros de 16 Bits. El microprocesador los suele usar para:

AX = Acumulador.
BX = Base.
CX = Contador.
DX = Dato.

Los registros se pueden dividir en por ejemplo AH, AL, donde cada uno de ellos, puede contener 8 bits (1 Byte)

Esto lo veremos en profundidad en los siguientes capítulos.

## 1.5 EJERCICIOS

1. Hacer una lista de los equivalentes binarios de los números decimales 20 al 32.
2. Pasar los siguientes números binarios a sus equivalentes decimales:
    a. 00110101
    b. 00010000
3. Pasar los siguientes números Hexadecimales a Decimales
    a. E3h
    b. 52h
    c. AAAAh

### 1.5.1 Resultados

1. Resultado.

| Decimal | Binario |
|---|---|
| 20 | 10100 |
| 21 | 10101 |
| 22 | 10110 |
| 23 | 10111 |
| 24 | 11000 |
| 25 | 11001 |
| 26 | 11010 |
| 27 | 11011 |
| 28 | 11100 |
| 29 | 11101 |
| 30 | 11110 |
| 31 | 11111 |
| 32 | 100000 |

2. Resultado.

    El resultado es **53** y **16**. Desarrollo el primer número binario: **00110101**

    1x32=32
    1x16=16
    0x8=0
    1x4=4
    0x2=0
    1x1=1

La suma sería de la siguiente forma:

32
16
00
04
00
01
---
53

3. Resultado.

**E3h** = (3x1) + (14x16) = 227
**52h** = (2x1) + (5x16) = 82
**AAAAh** = (10x1) + (10x16) + (10x256) + (10x4096) = 43690