Un programa es una serie de instrucciones escritas en un determinado lenguaje. Seguro que conocerás muchos lenguajes de programación: Basic, Visual Basic, Visual C++, Delphi, Borland C++, Real Basic, D y un largo etc... Sin embargo, un ordenador no puede entender directamente estos lenguajes de programación. Vamos a ver un sencillísimo ejemplo que he hecho en Visual Basic 6:

Dim i As Integer  
i = 0  
i = i + 666

Simplemente indico que la variable "i" es un entero, pongo su valor a cero y finalmente le sumo 666. Bien, esto para un programador puede no tener más misterio, me explico: el programador escribe su código y cuando quiere ejecutar su aplicación simplemente tiene que pulsar una opción dentro de Visual Basic 6 para generar su .exe y asunto terminado, no se tiene que preocupar por nada más. Sí, si no lo habías pensado nunca ya era hora: ¿Qué ha pasado con el código que he hecho en Visual Basic 6? ¿en qué se ha convertido? ¿Por qué?  
  
Bueno, como se ha repetido cientos de veces, un "ordenador" entiende directamente 0 (ceros) y 1 (unos), por lo tanto, el código que hemos hecho en Visual Basic 6 no lo puede entender así tal cual. El código o **lenguaje máquina** consta de cadenas de estos 0 y 1 que el microprocesador entiende directamente. Una y sólo una secuencia de estos 0 (ceros) y 1 (unos) ejecutará una determinada operación. El **lenguaje máquina** es complejo y difícil y varía de una arquitectura a otra. Cada procesador tiene un valor para sus instrucciones, no es lo mismo un procesador x86 que por ejemplo un Sparc. Yo tengo un procesador Intel de la familia x86. Observa la siguiente secuencia de 0(ceros) y 1(unos):

1000000

¡Nuestro primer **código máquina**!.  
Mi procesador entiende esa secuencia y lo que hace es lo siguiente: *aumentar en una unidad el registro eax*. Si no sabes lo que son los registros, imagina a eax como una variable. Si eax=1, al ejecutar la instrucción "1000000" de nuestro código máquina, conseguiremos que eax=2.

Como has podido observar, el procesador trabaja con código binario (ceros y unos) ya que se opera con transistores que sólo pueden estar de dos formas: abierto o cerrado, 1 o 0... Un bit es una unidad de información que sólo puede representar dos valores 1 ó 0. Dado que es muy incómodo representar números grandes en binario, el sistema más usado para la representación de valores es el [Sistema Hexadecimal](http://es.wikipedia.org/wiki/Sistema_hexadecimal). Para manejarnos con este sistema de 16 símbolos diferentes, podemos ayudarnos en Windows de la calculadora que trae por defecto. Por todo esto que acabo de comentar, el **código máquina** que vimos antes lo agrupamos en 8 bits (1 byte). Así, le añadiríamos a nuestro primer código máquina un cero delante (se presupone). De este modo, el ejemplo anterior 1000000 queda así: 01000000 o lo que es lo mismo, 40 en hexadecimal o 64 en decimal:

1000000 = 01000000 = 40hex = 64dec

Ahora ya sabemos que 40h corresponde a una instrucción que en la familia x86, aumenta en una unidad el registro eax. Hemos mejorado un poco, sin embargo, acordarse de esto es difícil, así que podemos utilizar nemotécnicos que nos hagan este trabajo mucho más sencillo. Podríamos poner a 40h un nemotécnico así: incrementa eax. Ahora si nosotros vemos "incrementa eax" ya sabemos que lo que va a hacer el procesador es aumentar una unidad el registro eax. Fíjate el camino que hemos seguido:

1000000 = 01000000 = 40hex = incrementa eax

Este lenguaje que hemos creado con nemotécnicos y que es la representación más directa del código máquina es el **Lenguaje Ensamblador**. Es mucho más fácil de entender para el ser humano. Sin embargo, aunque me haya inventado el nemotécnico "incrementar eax" o te inventes el tuyo propio, existen ya unos nemotécnicos para procesadores x86. La mayoría de los usuarios usa la sintaxis AT&T (usada en la mayoría de herramientas de desensamblado de Linux) o la de Intel, así pues, "incrementar eax" es en sintaxis Intel *inc eax*.  
Este proceso que hemos realizado:

1000000 => inc eax

se denomina **desensamblar**. Existen muchísimos programas que permiten el desensamblado. Puedes utilizar OllyDBG, IDA, WinDBG, DEBUG(DOS) o cualquier otro de los miles que hay (aunque todos los que acabo de nombrar permiten también la depuración de aplicaciones...).  
Del mismo modo se podría realizar el proceso al revés:

inc eax => 1000000

Este proceso por el cual se convierte el lenguaje ensamblador en código que pueda leer una máquina, se denomina **ensamblar**. Existen también muchos ensambladores. En DOS, por ejemplo, los más usados fueron MASM y TASM. El **lenguaje ensamblador** es un lenguaje de bajo nivel, y uno de los principales problemas que tiene es un programa escrito en ensamblador para una determinada plataforma, (por ej. x86) no nos valdrá en otra. Además, para programar en ensamblador hay que conocer detalles de bajo nivel de la arquitectura del procesador con el que estamos trabajando. Este problema comentado puede solucionarse mediante el **compilador**: cuando programamos en un lenguaje de alto nivel (Delphi, C, C++, Fortran...) y queremos traducir el código fuente del lenguaje de alto nivel a lenguaje máquina, el proceso se denomina compilar.  
  
En **ingeniería inversa** (**Reverse engineering**) se trabaja con lenguaje ensamblador. Si por ejemplo, abres cualquier ejecutable con un desensamblador podrás ver el código del mismo, esto se suele denominar "código muerto". Sin embargo, y es donde aumenta considerablemente la importancia de la ingeniería inversa, es en poder depurar ese código, esto es, poder ejecutar cada instrucción e ir observando los resultados, esto se consigue con los programas denominados debuggers. Estos programas permiten "debuggear" un determinado código, pararlo donde uno quiera, modificar parámetros, examinar y modificar la memoria y un largo etc... En Windows, el debugger que posiblemente más gente utiliza actualmente es OllyDBG junto con IDA. Hace unos años era utilizado uno llamado Softice. Con toda esta explicación puedes intuir a qué denominamos Ingeniería Inversa: es el proceso mediante el cuál se depura a un programa, Sistema Operativo, página Web etc... sin tener el código fuente disponible. Con esto conseguimos examinar errores de programación, bugs, debilidades, ver el código ensamblador, ver cómo un programador ha resuelto un programa, en fin, con Ingeniería Inversa y suficientes conocimientos se puede modificar un programa totalmente. Cuando la Ingeniería Inversa es utilizada para obtener licencias de un programa se suele emplear el verbo **Crackear**, esto es, utilizar a ésta para obtener números de serie válidos sin consentimiento de su autor o para modificar programas normalmente de pago para que sean gratuitos.  
  
Ahora ya has visto que es muy importante saber ensamblador(asm), en inglés assembly. No es necesario saber programar, pero sí muy útil. Si nunca has trabajado con asm te pongo un pequeño código para que empieces a reconocerlo, hecho en 32 bits de un programa mío:

**xor edx, edx** => pone edx a cero; edx = 0  
**mov eax, 2AF0** => eax = 2AF0  
**mov ecx, 4** => ecx = 4  
**div ecx**

La instrucción "div ecx" hace lo siguiente:  
-Une edx con eax quedando edx:eax  
-Hace la división: (edx:eax)/ecx  
-Finalmente, tras la operación, el resultado de la operación estará en eax y el resto de la división en edx.  
Veamoslo: **edx:eax = 02AF0 = 2AF0**  
(edx:eax)/ecx => 2AF0/4  
2AF0/4 = ABC y resto=0, por lo tanto, tras realizar div ecx, **eax = ABC** y **edx = 0**

**Los registros**

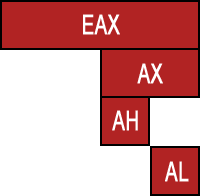
En el ejemplo anterior he utilizado los siguientes registros eax, edx, ecx... pero, ¿qué son los registros y para qué se utilizan?  
El código de los programas, la pila, los datos que utiliza y muchos otros elementos esenciales, se almacenan temporalmente en la memoria del ordenador. De la memoria, en algún momento, tendrán que pasar al microprocesador para que éste los pueda usar. Con este fin, el procesador cuenta con pequeñas zonas de memoria interna conocidas como **registros**. (Imagina los registros como un tipo de variables especiales del procesador)

Un byte(8bits) es la unidad básica en la que se mide la información. Cada una de las celdillas lógicas en que se divide la memoria del ordenador, tanto RAM como ROM, tienen capacidad para un byte. Existen otras medidas muy empleadas como el gigabyte, kilobyte y megabyte. Aparte del bit y byte se suelen utilizar a menudo **word**(palabra) y **dword**(doble palabra). 1 word = 2 bytes; 1 dword = 4 bytes.  
Existen distintos registros, y algunos para explicarlos requieren cierta complejidad que escapa a lo que pretendo en este tutorial y serán explicados en próximos. Aquí veremos lo más básico.  
Un pentium por ejemplo, dispone de 32 registros en su arquitectura interna, de los cuales, 16 son para uso del programador de aplicaciones. Estos 16 se pueden clasificar así:  
-Registros de propósito general  
-Registro Puntero de Instrucciones (EIP)  
-Registro de Estado o de Señalizadores (EFLAGS)  
-Registros de Segmento

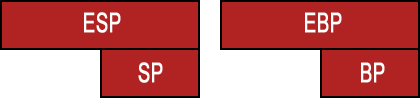
**Registros de propósito general:** Son 8 registros capaces de trabajar con información de 32 bits en su mayor tamaño. Pueden usarse tanto para almacenar datos como direcciones. Su nombre empieza por "E" que significa extendido. Son los siguientes:

* EAX: Acumulador.
* EBX: Base.
* ECX: Contador.
* EDX: Datos.
* ESP: Puntero de pila.
* EBP: Puntero de base.
* ESI: Índice fuente.
* EDI: Índice destino.

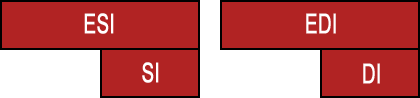
Estos registros, capaces de trabajar con información de 32 bits, pueden manejar datos de 16 bits y cuatro de ellos pueden manejar información de 8 bits. Cuando se accede únicamente a los 16 bits de menos peso, se designan por AX, BX, CX, DX, SP, BP, SI, DI, respectivamente. A los registros AX, BX, CX y DX se puede acceder a sus registros AL, BL, CL y DL cuando se accede al byte de menos peso y AH, BH, CH y DH cuando se accede al byte de más peso.  
Entonces, por ejemplo en el registro EAX podemos hacer el siguiente ejemplo:  
  
**EAX**: registro de 32 bits = 4 bytes => ej. 11223344  
**AX**: registro de 16 bits = 2 bytes de menor peso de EAX => ej. 3344  
**AH**: registro de 8 bits = 1 byte de mayor peso de AX => ej. 33  
**AL**: registro de 8 bits = 1 byte de menor peso de AX => ej. 44  
Gráficamente lo podríamos poner así:



**EAX**: Acumulador. (EAX, AX, AH, AL). Es un registro que se emplea en todas las operaciones lógico-aritméticas.  
Imagina que EAX = 11223344, ahora realizamos la instrucción: **mov ah, al** (mueve el valor de al a ah, dejando intacto el valor de al). ¿Cuál será el resultado final de EAX? Piensa un poco, mira el gráfico anterior y el párrafo que hay justo encima... (Resultado = 11224444)  
**EBX**: Base. (EBX, BX, BH, BL). Contiene una dirección que apunta a la base de un conjunto de datos.  
**ECX**: Contador. (ECX, CX, CH, CL). Verás muchas veces loops y bucles. ECX normalmente será cargado con el número de veces que se tiene que repetir dicho bucle.  
**EDX**: Datos. (EDX, DX, DH, DL).  
**ESP (ESP, SP) y EBP (EBP, BP)**: Sirven para controlar el direccionamiento de la pila. Los veremos más detenidamente en algún tutorial posterior. Observa que no es posible acceder a cada uno de los dos bytes del registro SP ó BP, a diferencia de AX, BX, CX y DX.



**ESI (ESI, SI) y EDI (EDI, DI)**: Son dos punteros de direcciones necesarios para trabajar con cadenas de caracteres.



Una instrucción donde los puedes ver en funcionamiento:   
**rep movs byte ptr es:[edi], byte ptr ds:[esi]**  
Voy a explicar esta instrucción con un ejemplo. Imaginemos que en la dirección 401000 tenemos la palabra karmany, de este modo:

Dirección Contenido memoria ASCII

00401000 6B 61 72 6D|61 6E 79 00| karmany.

00401008 00 00 00 00|00 00 00 00| ........

Como verás más tarde, toda cadena termina en cero para indicar el final de la misma, así que karmany ocupa desde la dirección 00401000 hasta 401007 (a.i.), por lo tanto ocupa 8 bytes. Lo que vamos a hacer es copiar toda la cadena karmany que está en 401000 y la vamos a pegar en 401008. Necesitamos 3 datos:  
1º - Dirección inicial = 401000  
2º - Dirección final = 401008  
3º - Longitud = 8 bytes  
Voy a mostrar el código, que es muy sencillo:

mov esi, 401000  
mov edi, 401008  
mov ecx, 8  
rep movs byte ptr es:[edi], byte ptr ds:[esi]

Se irán "moviendo" byte a byte desde esi hasta edi, hasta un total de 8 bytes (ecx). El resultado final tras el código anterior es este:

Dirección Contenido memoria ASCII

00401000 6B 61 72 6D|61 6E 79 00| karmany.

00401008 6B 61 72 6D|61 6E 79 00| karmany.

**Registro Puntero de Instrucciones (EIP)** : (EIP IP) Es el puntero de las instrucciones. Cuando debuggeamos, nos indica la dirección en la cuál se está ejecutando el código.

**Registro de Estado o de Señalizadores (EFLAGS)**: Consta de 32 bits, de los cuales la mayoría son señalizadores de estado. Explicarlos todos me parece muy alejado de este primer tutorial y poco acertado así que mostraré un simple ejemplo utilizando el bit ZF: Señalizador de cero. Se activa cuando ejecutamos una instrucción y el resultado es cero. Veamos el siguiente código:  
**mov eax, 1**  
**dec eax** => El Flag ZF pasa a ser 1.

**Registros de Segmento**: Un segmento es un trozo de la memoria de tamaño variable que contiene el mismo tipo de información (pila, código, datos). Para controlar los segmentos, se dispone de varios registros de 16 bits. Esto lo veremos más detenidamente en tutes posteriores.

Aunque al final hay bastante información, pienso que todo lo comentado hasta aquí es muy básico y todo el mundo más o menos, ya tendría que tener los conceptos bastante claros. De todos modos, he intentado hacerlo asequible a cualquier persona no iniciada. Con esta base, ya se pueden empezar a tratar temas más complejos.

karmany