Bienvenidos a:

El curso de Assembler de CaoS ReptantE

Versión 1.0
Octubre de 2.002

INTRODUCCIÓN

He estado un tiempo dudando sobre si era oportuno escribir este manual, ya que existen manuales escritos por gente que sin duda sabe más que yo, pero finalmente he pensado que podía aportar mi punto de vista y hacer un trabajo que complementara la información que puede encontrarse actualmente en la Red sobre este tema.

Este manual trata sobre el lenguaje Ensamblador de los procesadores 80X86, pero enfocado principalmente a programas de 32 bits.

Normalmente, a todos nosotros nos enseñaron de niños a leer y a escribir, porque una cosa no tenía sentido sin la otra. Aquí, por el contrario, mi intención no es la de enseñar a programar (para esto tendría que empezar por aprender yo), sino a "leer" programas, es decir, a comprender lo que vemos que está pasando durante su ejecución. Para programar, sin duda es mejor utilizar lenguajes de alto nivel, ya que, desde mi punto de vista, el Assembler está reservado para programar aplicaciones sencillas o para ser utilizado por seres de un intelecto fuera de lo común :-)

Como podéis ver, este curso empieza indicando la versión del documento. Lógicamente esto significa que está sujeto a la introducción de posibles modificaciones y mejoras, para lo cual solicito vuestra colaboración. Para ello, os agradeceré que me hagáis llegar vuestras correcciones o comentarios por correo electrónico a: caos_reptante@hotmail.com.

Lo primero sobre lo que debería tratar es la terminología a emplear, pero no voy a hacerlo. El que no sepa lo que es un bit o un byte, quizá debería empezar por otra cosa, ya que aunque no voy a hablar de tecnología punta, si que sería de desear un mínimo nivel. En cuanto a lo que significan otras expresiones como por ejemplo nibble o doubleword, ya lo iremos viendo sobre la marcha.

Por otro lado, cada uno tiene sus costumbres o manías y así podréis ver por ejemplo, que siempre me refiero a la pila por su nombre en castellano, dejando de lado la palabra anglosajona stack, al contrario de lo que sucede con los flags, a los que nunca llamo banderas. Espero que esta manera de expresarme no haga este texto más ininteligible de lo que ya es...

En las operaciones matemáticas, he utilizado los signos matemáticos tradicionales, en vez de los empleados en informática. Por ejemplo, he empleado la representación tradicional de potencia, aunque en programación el símbolo que la representa es "^". Así 2^5 equivale a 2^5

Diré finalmente que hace poco me preguntaron por qué me dedicaba al cracking y yo contesté que porque jugaba muy mal al ajedrez. Espero que para vosotros sea también un desafío intelectual y no un sistema para adquirir programas gratis. No os dejéis seducir por el lado oscuro de la Fuerza... ;-)

ÍNDICE

1. SISTEMAS DE NUMERACIÓN

1.1. Decimal

Este es el sistema que conocemos todos y no parece necesario explicar gran cosa sobre él :-) Se le llama decimal o en base diez porque cada posición de las cifras que forman un número, representa el múltiplo de esa cifra por una potencia de diez dependiente de su posición en dicho número:

10^4	10^3	10^2	10^1	10^0	
4	5	1	8	3	=

```
4 x 10000   = 40000
5 x  1000   =  5000
1 x   100   =   100
8 x    10   =    80
3 x     1   =     3
              45183
```

1.2. Binario

Desde el diseño de los primeros ordenadores se emplearon circuitos biestables con la posibilidad de reconocer sólo dos estados: con tensión y sin tensión, es decir: 1 y 0. Lógicamente, para representar cualquier número binario sólo necesitamos dos caracteres: el "1" y el "0". El valor de un número binario se obtiene igual que el de uno decimal, sólo que se emplea la base dos en vez de diez:

2^5	2^4	2^3	2^2	2^1	2^0	
1	1	1	0	0	1	=

```
1 x 32 (2^5) = 32
1 x 16 (2^4) = 16
1 x  8 (2^3) =  8
0 x  4 (2^2) =  0
0 x  2 (2^1) =  0
1 x  1 (2^0) =  1
               57
```

1.3. Hexadecimal

La representación de un número binario requiere una gran cantidad de cifras. Para escribir 65536 (2^{16}) se necesitan 17 cifras: un uno y dieciséis ceros detrás. Por esta razón se buscó un sistema que fuera más o menos manejable por seres humanos (Esta definición, aunque no lo creáis, incluye también a los programadores :-D) y que fuera también fácilmente convertible a binario. La solución está en el sistema hexadecimal. El primer problema con que nos encontramos es el de que nos faltan cifras. En el sistema decimal tenemos las cifras del 0 al 9, y en binario, el 0 y el 1, pero en un sistema hexadecimal debemos utilizar otros caracteres para las cifras del "10" al "15" ya que en hexadecimal el 10 equivale al 16 decimal. Estos caracteres son las letras mayúsculas de la A a la F. Veamos ahora el valor en decimal de un número hexadecimal:

16^3	16^2	16^1	16^0	
7	A	2	E	=

```
7    x 4096(16^3)= 28672
A(10)x  256(16^2)=  2560
2    x   16(16^1)=    32
E(14)x    1(16^0)=    14
                    31278
```

He dicho que era fácil la conversión entre números en binario y hexadecimal. En efecto, podemos convertir este número hexadecimal en binario sin necesidad de calculadora y ni siquiera de lápiz y papel:

```
        8421 8421 8421 8421
        |||| |||| |||| ||||
7A2E = 0111 1010 0010 1110
     (4+2+1)(8+2)(2) (8+4+2)
```

o a la inversa:

```
          8421 8421 8421 8421
          |||| |||| |||| ||||
          1111 0011 1001 1100 = F39C
(8+4+2+1)(2+1)(8+1)(8+4)
```

Fijáos que cada grupo de cuatro cifras en binario equivale a una cifra hexadecimal. Así: 0111=7 (4+2+1), 1010=A (8+2) etc. Esto es debido a que una cifra hexadecimal puede representar un valor decimal comprendido entre 0 (0) y 15 (F) exactamente igual que un grupo de cuatro cifras en binario: 0 (0000) y 15 (1111).

1.4. Octal

Este es un sistema que prácticamente no se utiliza pero que no está de más saber que existe. La idea es la misma que la del sistema hexadecimal, pero en grupos de 3 bits: valor entre 000 (0) y 111 (7). Lógicamente, el 8 decimal equivale al 10 octal.

1.5. Conversión

Las conversiones que más nos interesan son entre números en decimal y hexadecimal. La conversión de hexadecimal a decimal ya la hemos visto más arriba, pero está hecha trabajando con números decimales. Como el ordenador trabaja sólo con números hexadecimales (esto no es del todo cierto, ya que el ordenador trabaja con números binarios, pero los desensambladores nos lo pasan a hexadecimal), la conversión la hace de otra manera:

```
7A2E:A = C37  resto=        8
 C37:A = 138  resto=       7
 138:A = 1F   resto=      2
  1F:A = 3    resto=     1
   3:A = 0    resto=    3
                        31278
```

Está claro que si dividimos un número por diez, el resto de esta división será siempre menor que diez, o sea una cifra que podremos representar en formato decimal. Vamos a ver ahora la conversión de decimal a hexadecimal. Si la hiciéramos trabajando con números decimales sería así:

```
31278:16 = 1954  resto= 14      (E)
 1954:16 = 122   resto= 2         2
  122:16 = 7     resto= 10     (A)
    7:16 = 0     resto= 7      7
                               7A2E
```

Pero nos encontramos de nuevo con que la máquina lo hace de otra manera:

```
   31278
  0+3     =   3 x A = 1E
 1E+ 1    =  1F x A = 136
 136+  2  = 138 x A = C30
 C30+   7 = C37 x A = 7A26
7A26+    8 =           7A2E
```

Curioso ¿no?

También conviene saber como pasar de binario a hexadecimal y de hexadecimal a binario, pero esto ya lo hemos visto anteriormente. En cambio, creo que carece de interés estudiar el paso de decimal a binario y viceversa, por lo que sólo diré que las conversiones se hacen: en el primer caso, multiplicando por potencias de dos, como ya hemos visto; y en el segundo, dividiendo sucesivamente por dos.

1.6. Identificación

Aunque confío en que haya quedado claro a qué nos referíamos en cada ocasión, habréis visto que hasta ahora hemos empleado números binarios, decimales y hexadecimales sin especificar a que sistema pertenecían, lo cual, en algún caso, puede dar lugar a confusiones. Para ello se suelen emplear sufijos al final del número, que determinan a que sistema pertenece dicho número. Estos sufijos son: "d" para decimal, "h" para hexadecimal, "b" para binario (aunque no suele ser necesario indicarlo) y "o" ó "q" para el octal (dato puramente anecdótico). Así escribiríamos:

```
31278d      7A2Eh      00101110b
```

De todos modos, a veces es más claro no poner los sufijos que ponerlos, ya que en muchos casos no es necesario y puede contribuir a embarullar algo que ya puede ser bastante lioso por si mismo. De hecho, podría haber empezado por explicar este sistema de representación y salpicar todos los números anteriores de sufijos, pero creo que hubiera resultado peor el remedio que la enfermedad. De ahora en adelante los iré poniendo, pero sólo en los casos en que pueda haber confusión.

Otra cosa a tener en cuenta es que los números hexadecimales que empiezan con una letra, se suelen representar poniendo un cero delante. Esto es necesario cuando se trata de un texto que se va a compilar, porque es la manera de que el compilador sepa al empezar a leerlo que aquello es un número; luego, a partir de la letra del final, sabrá en que formato está representado. Tampoco creo necesario seguir esta norma en el presente documento.

2. SISTEMAS DE REPRESENTACIÓN

2.1. Números negativos

Aquí la cosa empieza a complicarse. En nuestro viejo y querido sistema decimal, cuando un número es negativo se le pone un guión delante para indicarlo (excepto los bancos que lo ponen en números rojos en clara referencia a la sangre que nos chupan). Pero en sistemas binario y hexadecimal no podemos hacer semejante cosa, por lo que hubo que echar mano del invento diabólico de alguien que debía sufrir pesadillas por las noches :-(Se trata del complemento a dos. Vamos a ver si sé explicarlo :-|

Con un número binario de ocho dígitos se pueden representar los números del 0 al 255 en decimal. Pues a partir de aquí, a alguien se le ocurrió que si el primer dígito era un cero, significaría que el número era positivo, y negativo en caso contrario. Esto significa que con ese número binario de ocho dígitos, en lugar de representar los números del 0 (00000000) al 255 (11111111), se representarán ahora los números del 0 (00000000) al 127 (01111111) y del -1 (11111111) al -128 (10000000). Los números positivos creo que están claros, pero ¿por qué son esos y no otros los números negativos? Pues porque para pasar de un número positivo a uno negativo o viceversa, se invierten todos los dígitos que lo forman y al resultado se le suma uno. Aquí tenéis unos ejemplos:

```
00000001   (1) -> 11111110 + 1 -> 11111111   (-1)
00111111  (63) -> 11000000 + 1 -> 11000001  (-63)
01111111 (127) -> 10000000 + 1 -> 10000001 (-127)
```

Este sistema tiene una ventaja y es la de que si efectuamos la suma entre un número positivo y uno negativo, el resultado es correcto, o sea que es igual a la resta de ambos números sin tener en cuenta el signo:

```
  01111111 (127)        01000000 (64)          10000001 (-127)
+ 11000001 (-63)      + 11000001 (-63)       + 11111111 (-1)
 101000000 (64)        100000001 (1)          110000000 (-128)
```

El uno de la izquierda (en cursiva) se desborda por la izquierda ya que en este ejemplo estamos trabajando con 8 bits y no hay sitio para el noveno, por lo que no se le tiene en cuenta.

Visto así, quizá la cosa no parece tan complicada, pero las cosas siempre tienden a torcerse y esto no es una excepción. En este ejemplo, nos hemos limitado a ocho bits, lo que representaría por ejemplo el registro AL, que más que un registro, es la parte inferior (8 bits=1 byte) del registro EAX (32 bits=4 bytes). Si tratáramos con 32 bits, o sea con un registro completo, los números positivos estarían comprendidos entre 00000001 y 7FFFFFFF y los negativos entre 80000000h y FFFFFFFF. Y es que todo esto depende

del entorno en el que nos movamos, por lo que un número puede ser positivo o negativo según le referencia en que lo situemos. Por ejemplo, si decimos que AX (16 bits=2 bytes) es igual a FFFF, estamos diciendo que es igual a -1, pero si decimos que EAX es igual a 0000FFFF, lo que estamos diciendo es que es igual a 65535. De la misma manera, si le sumamos FF a AL, le estamos sumando -1 (o dicho de otra manera, restándole uno), pero si le sumamos esta cantidad a EAX, le estamos sumando 255. Este es un tema que me causó bastantes problemas cuando me dedicaba a hacer keygens, porque si probaba nombres en los que entraban letras acentuadas o la letra ñ, resultaba que a veces, según como estuviera programada la generación del serial, los resultados se veían falseado por la presencia de dichos caracteres.

En fin, de momento no os preocupéis demasiado por este tema, que espero irá quedando más claro.

En cuanto a lo que he dicho sobre registros, enseguida lo vamos a ver con detalle.

2.2. Coma (o punto) Flotante

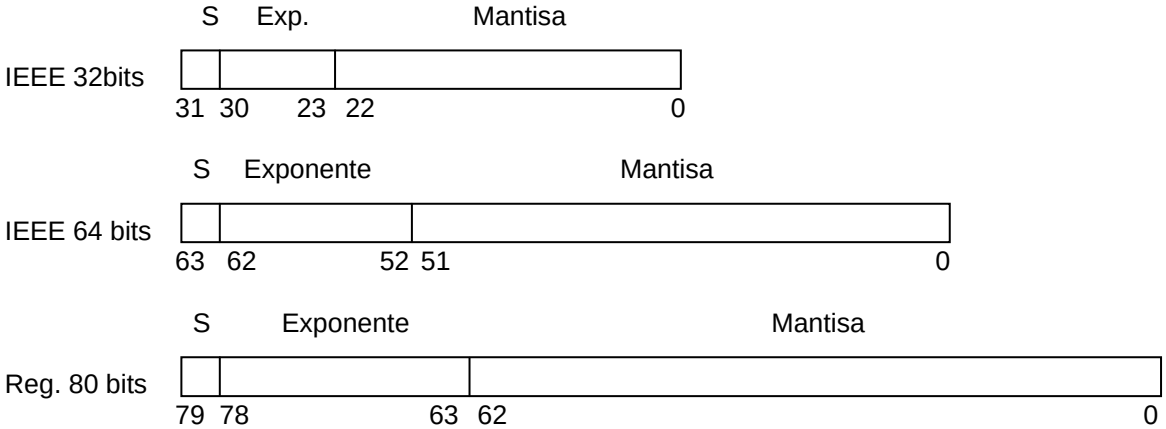
Hay un tema más complicado aún que el de los números negativos. Se trata de la representación de números no enteros. De la misma manera que no podíamos poner un guión para representar un número negativo en binario o hexadecimal, tampoco podemos poner una coma (o un punto en el caso de los yanquis) para separar la parte entera de la decimal. Para solucionar esto se emplea la representación en coma flotante, mucho más versátil que la representación en coma fija.

Hasta la llegada del procesador 80486, las operaciones de coma flotante las efectuaba el coprocesador matemático. Para ello contaba con unos registros y unas instrucciones propias. A partir de la aparición del 80486, estas funciones han quedado incorporadas a la CPU. Las instrucciones de coma flotante, las veremos brevemente en el Apéndice A de este manual, mientras que los registros los veremos ahora, separadamente del resto.

Únicamente nos interesa saber que se trata de 8 registros de 80 bits, numerados del ST o ST(0) al ST(7) y cuya composición veremos inmediatamente.

Existen tres formatos de numeros reales en coma flotante que nos interesan especialmente: de 32, 64 y 80 bits. Los dos primeros son formatos standard IEEE (Institute of Electrical and Electronics Engineers) para simple y doble precisión, y el tercero es el formato de los registros de coma flotante.

La representación de estos números esta formada por tres elementos: signo, exponente y mantisa. El signo está indicado por un bit. El exponente por 8, 11 y 16 bits según el formato y la mantisa por 23, 52 y 63 bits; segun el gráfico siguiente:



Vamos a ver unos ejemplos de como pasar números reales a formato en coma flotante de 32 bits (El procedimiento para los formatos más largos es similar).

Ejemplo nº 1: 105,28

En primer lugar, hay que pasar los números de la parte entera y decimal a binario. Para pasar a binario la parte entera ya vimos anteriormente como se hacía:

```
105:2 = 52    resto= 1          1
 52:2 = 26    resto= 0         0
 26:2 = 13    resto= 0        0
 13:2 = 6     resto= 1       1
  6:2 = 3     resto= 0      0
  3:2 = 1     resto= 1     1
  1:2 = 0     resto= 1    1
                          1101001
```

Ya tenemos 7 numeros, ahora sólo nos faltan 17 para completar los 24 números que necesitamos para la mantisa (Necesitamos 24 porque vamos a eliminar el primer número).

La parte decimal se pasa a binario multiplicando la parte decimal por 2 y con la parte entera del resultado se va formando el número binario:

```
0,28x2 = 0,56  entero=0    0
0,56x2 = 1,12  entero=1     1
0,12x2 = 0,24  entero=0      0
0,24x2 = 0,48  entero=0       0
0,48x2 = 0,96  entero=0        0
0,96x2 = 1,92  entero=1         1
0,92x2 = 1,84  entero=1          1
0,84x2 = 1,68  entero=1           1
0,68x2 = 1,36  entero=1            1
0,36x2 = 0,72  entero=0             0
0,72x2 = 1,44  entero=1              1 ...
```

y así hasta 17 ...

```
01000111101011100
```

Componemos nuestro binario con decimales:

```
1101001 , 01000111101011100
```

Ahora corremos la coma hacia la izquierda hasta que no quede parte entera, en este caso, siete lugares:

```
0,11010010100011110101110 0
```

Eliminamos la coma y el primer uno del número y ya tenemos los 23 bits de la mantisa:

```
10100101000111101011100
```

Ahora vamos a por el exponente. Hemos corrido siete lugares a la izquierda, esta cifra la sumaremos a 126 (número costante) y pasaremos a binario el resultado:

```
7+126 = 133d = 10000101b
```

Ya tenemos también el exponente. El signo es positivo por lo tanto el bit que correspondiente al signo estará puesto a cero. Ya tenemos el número completo, es el:

```
0 10000101 10100101000111101011100
```

Ahora vamos a pasar ese número a hexadecimal y comprobaremos con el SoftIce cual es el valor almacenado en el registro.

```
0100 0010 1101 0010 1000 1111 0101 1100
  4    2    D    2    8    F    5    C

C7035C8FD242            mov dword ptr [ebx], 42D28F5C
D903                    fld dword ptr [ebx]
```

Si ejecutamos este código (Simplemente se trata de un injerto a uno de mis keygens hecho con el Masm, como muchos de los ejemplos que aparecen en este manual), y ponemos un bpx a la dirección de la instrucción fld, veremos como después de ejecutarse dicha instrucción, aparece en la ventana de los registros de coma flotante (esta ventana se activa con `wf`) el valor 105,279998779296875 en el registro ST(0). El error viene dado por el nivel de precisión que hemos empleado. Si hubiéramos trabajado con 64

u 80 bits, el resultado hubiera sido más ajustado.

Pensaba poner más ejemplos pero para no eternizarme con este tema sólo pondré un par de valores para que podáis comprobarlos:

```
0,00425      0 01110111 00010110100001110010101

C70395438B3B            mov dword ptr [ebx], 3B8B4395
D903                    fld dword ptr [ebx]          (4,249999765306711197e-3)

-205,5       1 10000110 10011011000000000000000

C70300804DC3            mov dword ptr [ebx], C34D8000
D903                    fld dword ptr [ebx]          (-205,5)
```

Ahora vamos a ver el paso inverso, es decir, vamos a pasar un número en coma flotante a decimal. Podemos utilizar como ejemplo uno de los dos anteriores que no hemos visto con detalle:

Ejemplo nº 2: 3B8B4395=0 01110111 00010110100001110010101

Vemos por el bit de signo que se trata de un número positivo.

El exponente es 01110111=119. Le restamos 126 y nos da -7.

Añadimos un uno a la izquierda de la mantisa y le ponemos una coma delante. A continuación pasamos esta mantisa a decimal (Lo más sencillo es pasarla con lápiz y papel a hexadecimal y luego con la calculadora científica de Window$, a decimal) y nos da como resultado:

$$0 , 1000\ 1011\ 0100\ 0011\ 1001\ 0101 = 8B4395h = 9126805d$$

Ahora movemos la coma a la derecha, los lugares necesarios para que nos quede un número entero (en este caso, 24) y restamos el número de desplazamientos al exponente que teníamos (-7 -24 = -31). Así que el número resultante es:

$$9126805 \times 2^{-31} = 0,00424999976530671119689941406 25$$

Resultado obtenido mediante la calculadora científica de Window$.

2.3. Formato BCD

Se trata simplemente de una manera de almacenar números decimales. Cada dígito decimal se pasa a binario y se guarda en un nibble (cuatro bits). Estos números binarios estarán comprendidos entre el 0000 y el 1001 (9), no siendo válido un número superior. A partir de aquí, hay dos maneras de guardar estos números: BCD empaquetado y BCD desempaquetado. En el formato empaquetado se guardan dos dígitos en un byte, en el desempaquetado se guarda un dígito en un byte, del cual quedan a cero los cuatro bits de mayor valor. Veremos un par de ejemplos:

Decimal	BCD empaq.	BCD desemp.
76	0111 0110	00000111 00000110
91	1001 0001	00001001 00000001

Con números representados en este formato, se pueden realizar operaciones aritméticas, pero se efectúan en dos tiempos. En el primero se lleva a cabo la operación aritmética, y en el segundo, se ajusta el resultado.

2.4. Caracteres ASCII - ANSI

Aunque no es un tema directamente relacionado con el lenguaje Assembler, también debemos conocer algo sobre como se las arregla el ordenador para escribir caracteres en la pantalla, mandarlos a la impresora, etc. Lo hace asignando un código a cada carácter que se quiera representar. Así nació en su

dia el juego de caracteres ASCII (American Standard Code for Information Interchange), que algunas veces se identifica como OEM (Original Equipment Manufacturer), compuesto por 256 caracteres, que era el utilizado por el MS-DOS. Window$ en cambio, utilizó el ANSI (American National Standards Institute) que consta de 224 caracteres, y posteriormente, en las versiones NT el Unicode, que al tener dos bytes de longitud en vez de uno, como los sistemas anteriores, es capaz de representar 65536 caracteres.

Los 32 primeros caracteres (del 0 al 31) del formato ASCII son códigos de control, como por ejemplo el 13 que equivale a CR (Enter), o el 8 que equivale a BS (Retroceso). Los caracteres del 32 al 127 son idénticos en ASCII y ANSI (El juego de caracteres ANSI empieza en el 32). Finalmente los caracteres del 128 al 255 (llamados también extendidos) son distintos para ambos sistemas y varían según los sistemas operativos o configuraciones nacionales.

Los 96 caracteres comunes a ambos sistemas son los siguientes:

Dec.	Hex.	Caract.	Dec.	Hex.	Caract.	Dec.	Hex.	Caract.
32	20	esp	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	

Es bastante frecuente encontrarse, al seguirle la pista a un serial number, con instrucciones en las que se le resta 30h al código ASCII de un número para obtener su valor numérico real, o bien se le suma 30h a un número para obtener el carácter ASCII que lo representa, o con otras que suman o restan 20h al código de una letra para pasarla de mayúscula a minúscula o viceversa.

3. OPERACIONES LÓGICAS

Son cuatro y se ejecutan bit a bit según las tablas que veremos a continuación.

3.1. and

El resultado es 1 si los dos operandos son 1, y 0 en cualquier otro caso.

```
1 and 1 = 1
1 and 0 = 0
0 and 1 = 0
0 and 0 = 0
```

Ejemplo: `1011 and 0110 = 0010`

3.2. or

El resultado es 1 si uno o los dos operandos es 1, y 0 en cualquier otro caso.

```
1 or 1 = 1
1 or 0 = 1
0 or 1 = 1
0 or 0 = 0
```

Ejemplo: `1011 or 0110 = 1111`

3.3. xor

El resultado es 1 si uno y sólo uno de los dos operandos es 1, y 0 en cualquier otro caso

```
1 xor 1 = 0
1 xor 0 = 1
0 xor 1 = 1
0 xor 0 = 0
```

Ejemplo: `1011 xor 0110 = 1101`

3.4. not

Simplemente invierte el valor del único operando de esta función

```
not 1 = 0
not 0 = 1
```

Ejemplo: `not 0110 = 1001`

Estos ejemplos los hemos visto con números binarios. Como estas operaciones se hacen bit a bit, si quisiéramos hacer una operación entre números en formato hexadecimal, que es lo más corriente, deberíamos pasarlos antes a binario. O bien utilizar la calculadora de Window$ en formato científico.

4. ESTRUCTURA DE LA MEMORIA

Si alguien sobrevivió a la explicación sobre representación de números negativos y números reales en coma flotante, tendrá mayores probabilidades de supervivencia ante esta nueva prueba. A esto se le llama selección natural :-)

Empezaremos viendo lo que sucede cuando se trabaja con 16 bits. En estas condiciones, sólo se pueden direccionar 64 Kb de memoria (2^{16} = 65536). ¿Cómo se puede hacer para direccionar en la memoria programas de más de 64 Kb? Muy fácil, dividiendo la memoria en segmentos de 64 Kb. Así pues, una dirección de memoria esta compuesta por dos números: el primero, el del segmento y el segundo, el del offset o desplazamiento, es decir, el de su posición dentro del segmento. Veamos como desensambla el W32Dasm la parte correspondiente al Entry Point de un programa de 16 bits:

```
//******************** Program Entry Point ********

Exported fn(): Sol - Ord:0000h
Exported fn(): Microsoft Solitaire - Ord:0000h
:0008.0083 33ED                  xor bp, bp
```

```
:0008.0085 55                    push bp
:0008.0086 9AFFFF0000            call KERNEL.INITTASK
```

Fijáos que el segmento correspondiente al Entry Point es el 0008, sin embargo, al ejecutar el programa, veo que el segmento es el 188F, y como me quedo con la duda, lo ejecuto de nuevo y esta vez el segmento es el 1347. Esto es debido a que en cada ejecución, el programa es cargado en la memoria en el segmento que al ordenador le parece conveniente. En cambio, el desensamblador empieza a contar por el segmento número uno, porque ni sabe ni le importa donde va a ser cargado el programa:

```
//********************** Start of Code in Segment: 1 **************

Exported fn(): FUNC077 - Ord:0055h
:0001.0000 1E                    push ds
:0001.0001 58                    pop ax
```

Hay algo que todavía no he dicho. Y es que para facilitar el manejo de la información, alguien tuvo la idea de solapar estos segmentos, de tal forma que a los dieciseis bytes de empezado un segmento, empieza el siguiente, y así sucesivamente. La lógica de esto es que si se multiplica por 16 el número del segmento (que tiene una longitud de 16 bits), se obtiene un número de 20 bits, al cual se le suma el número correspondiente al desplazamiento, para obtener la dirección física en la memoria. En el ejemplo anterior, veíamos que en una de las ejecuciones, el Entry Point estaba en la dirección 188F:0083, lo que significa que la dirección real era 188F0+0083h=18973h. O sea que la dirección de un punto de la memoria está determinada por un número de 20 bits. La pregunta ahora es: ¿cuantas posiciones de memoria se pueden direccionar con 20 bits? La respuesta es: 2^{20}=1.048.576 o sea 1 Mb.

Y aquí la cosa se complica, porque resulta que una posición en la memoria se puede representar de maneras distintas. Así la dirección 1234:0040h sería la misma que 1235:0030h y 1236:0020h (12380h). En fin, procurad no preocuparos por esto :-)

Trabajando con 32 bits la cosa es algo distinta. Se pueden direccionar hasta 4 GB (4096x1024x1024 bytes) de memoria, lo cual sería suficiente (hasta el momento) para no tener que utilizar segmentos. Sin embargo se siguen utilizando, aunque ahora tenemos la ventaja de que un programa nos cabe en un solo segmento. Veamos ahora un ejemplo de desensamblado en un programa de 32 bits:

```
//******************** Program Entry Point ********

:004010CC 55                     push ebp
:004010CD 8BEC                   mov ebp, esp
:004010CF 83EC44                 sub esp, 00000044
```

Ved como ahora se omite el segmento y el W32Dasm sólo nos muestra los 8 bytes que forman la dirección. Esto es debido a que, como ahora todo el programa cabe en un solo segmento, no es necesario numerarlo como se hacía en el ejemplo de 16 bits para distinguir un segmento de otro. Si ahora ejecuto el programa, éste es colocado en el segmento 0167. Pues bueno.

Espero que con lo visto os hagáis un poco de idea de como esta estructurada la memoria, sólo me falta decir que cuando veamos los registros, veremos como quedan almacenados los distintos segmentos que intervienen en los procesos que se ejecutan en cada momento.

5. LA PILA

La pila es un lugar de la memoria donde se van guardando determinados valores para recuperarlos posteriormente. Por esta razón, cuando se introduce un nuevo valor no se hace en el lugar ocupado por el valor introducido anteriormente, sino que se pone justo en la posición o posiciones inmediatamente anteriores a la ocupada por ese valor previo.

La pila sigue la norma LIFO (last in, first out) y funciona como una pila de platos. Si colocamos uno a uno cinco platos sobre una pila, y luego los vamos cogiendo, tomaremos en primer lugar el último que hayamos puesto, luego el penúltimo etc., hasta llegar al que hayamos puesto en primer lugar, que será el último que cogeremos. La dirección para acceder al plato superior de la pila, es decir al valor en que podemos acceder en cada momento, está contenida en un registro (ESP) y, lógicamente, va variando según se van añadiendo o retirando valores.

Hay diferentes maneras de modificar el estado de la pila: directamente, mediante las instrucciones que se emplean para poner o quitar valores, o mediante instrucciones que tienen el efecto de modificarla, como la instrucción `call` (llamada a subrutina) que tiene como efecto secundario el guardar en la pila la dirección a la que ha de volver el programa cuando, una vez terminada la ejecución de la subrutina, se encuentra con la instrucción `ret`, la cual retira la dirección introducida de la pila, dejándola como estaba antes de la ejecución de la subrutina.

La finalidad principal de la pila es liberar registros de forma temporal para que puedan realizar otras funciones para luego, una vez terminadas las mismas, reintegrarles su valor primitivo. Otra utilidad puede ser la que a veces emplean programadores temerosos de que los chicos malos puedan robarles sus tesoros ;-) y es la de saltar de un punto a otro durante la ejecución de un programa sin que esto quede reflejado en el desensamblado de dicho programa:

```
push  41456A
...
ret
```

Esto quiere decir que el programa, al encontrar la instrucción `ret`, saltará a la dirección contenida en la pila, es decir 41456A, creyendo regresar de una subrutina. Pero esto, si la instrucción `push` está puesta de manera que se confunda entre otras, sólo se puede ver durante la ejecución del programa. Desde luego, hay gente mala :´-(

6. LOS REGISTROS

Los registros son elementos de almacenamiento de datos contenidos en el procesador y que tienen la ventaja de la rapidez de acceso y la finalidad de contener datos necesarios para la ejecución del programa. En principio, casi todos ellos pueden utilizarse libremente, pero cada uno de ellos tiene sus funciones específicas. Existen distintos tipos de registro:

6.1. Generales

Son cuatro: EAX, EBX, ECX y EDX.

El registro EAX (Acumulador) además de su empleo como registro para uso general, es utilizado en algunas instrucciones como las de multiplicar y dividir, que como veremos más adelante lo utilizan como factor y como resultado. También se utiliza para contener el valor de retorno después de la ejecución de una API, por ejemplo, al regreso de `lstrlen`, EAX contiene la longitud de la cadena de texto examinada, y al regreso de `RegQueryValueExA`, EAX estará puesto a cero si el registro se ha leído correctamente.

El registro EBX (Base) además de su uso general, suele utilizarse para direccionar el acceso a datos situados en la memoria.

El registro ECX (Contador) además de su uso general, se utiliza como contador en determinadas instrucciones, como por ejemplo:

```
mov ecx, 00000010h
repnz
movsb
```

Estas instrucciones copiarían 16 bytes de una dirección a otra de la memoria (Más adelante veremos como lo hace). Veamos otro ejemplo:

```
            mov ecx, 00000005
00043A2C    ....
            loop 00043A2C
```

Se ejecutaría 5 veces un bucle.

El registro EDX (Datos) además de su uso general, se utiliza junto con EAX para formar números mayores de 32 bits en algunas instrucciones, como las de multiplicar y dividir. También se utiliza en operaciones de Entrada/Salida.

Todos estos registros son de 32 bits, aunque se puede operar con subdivisiones de ellos. Así por ejemplo, los 16 bits de menor valor del registro EAX, se pueden manejar como AX, y este AX, a su vez, se divide en AH (bits 5º a 8º contados a partir de la derecha) y AL (bits 1º a 4º). Vamos a ver un ejemplo:

```
mov eax, 1A2B3C4D       EAX=1A2B3C4D
mov ax, FFFF            EAX=1A2BFFFF
mov al, 34              EAX=1A2BFF34
mov ah, 12              EAX=1A2B1234
```

Volvamos a ver aquello que decíamos sobre los números negativos con otro ejemplo:

```
mov eax, FFFFFFFF       EAX=FFFFFFFF
add eax, 00000001       EAX=00000000
mov eax, FFFFFFFF       EAX=FFFFFFFF
add ax, 0001            EAX=FFFF0000
mov eax, FFFFFFFF       EAX=FFFFFFFF
add al, 01              EAX=FFFFFF00
mov eax, FFFFFFFF       EAX=FFFFFFFF
add ah, 01              EAX=FFFF00FF
```

Espero no haber creado más confusión :-)

6.2. De base

Son dos: EBX y EBP.

El registro EBX lo acabamos de ver como registro de uso general.

El registro EBP se utiliza para direccionar el acceso a datos situados dentro del espacio ocupado por la pila.

También el registro EBP es de 32 bits y se puede utilizar para uso general. Puede operarse con los 16 bits inferiores: BP.

6.3. De índice

Son dos: ESI y EDI.

Los registros ESI y EDI se utilizan en instrucciones que acceden a grupos de posiciones contiguas de la memoria que funcionan como origen y destino. El registro ESI apunta a la dirección del primer (o último, como veremos al tratar el flag D) byte del origen y el EDI al primero (o último) del destino. Así en el ejemplo que veíamos antes:

```
mov ecx, 00000010h
repnz
movsb
```

Decía que se copiaban 16 bytes de un lugar a otro de la memoria, pero lo que no he dicho es que estos 16 bytes se tomaban de la dirección indicada por DS:ESI (S=Source) y se copiaban en la dirección indicada por ES:EDI (D=Destination). Lo de DS: y ES: lo veremos enseguida.

Como los anteriores, estos registros son de 32 bits y se pueden utilizar para uso general. Puede operarse con los 16 bits inferiores de cada uno: SI y DI.

6.4. De puntero

Son dos: ESP y EIP.

El registro ESP apunta a la dirección del último valor introducido en la pila (o sea, del primero que podríamos sacar) y aunque puede ser modificado mediante las instrucciones del programa (por ejemplo: sumándole o restándole un determinado valor), es más corriente que se modifique automáticamente según las entradas y salidas de valores en la pila. Por ejemplo, las siguientes instrucciones restarían y sumarían 4 al registro ESP:

```
push eax    (ESP=ESP-4)
pop eax     (ESP=ESP+4)
```

El registro EIP apunta a la dirección del código de la instrucción a ejecutarse y se va modificando según se va ejecutando el código del programa.

Estos registros también son de 32 bits, pero tanto ESP como EIP no se pueden utilizar para otros usos que los previstos.

6.5. De segmento

Son seis: CS, DS, SS, ES, FS y GS.

El registro CS contiene el segmento que corresponde a la dirección del codigo de las instrucciones que forman el programa. Así la dirección de la siguiente instrucción a ejecutarse de un programa vendría determinada por los registros CS:EIP

Los registros DS y ES se utilizan como ya hemos visto en instrucciones que tienen origen (DS:ESI) y destino (ES:EDI).

El registro SS es el que contiene el segmento que corresponde a la dirección de la pila. Así la dirección del valor que se puede extraer de la pila en un momento dado, estaría indicada por SS:ESP.

Los registros FS y GS son de uso general.

Estos registros son de 16 bits y no se pueden utilizar para otros usos que los previstos.

6.6. Flags

Estan situados en un registro (EFLAGS) de 32 bits de los cuales se utilizan 18, y de estos vamos a ver sólo 8, que son los que nos muestra el SoftIce.

O (Overflow o desbordamiento)

Este flag se pone a uno, cuando se efectúa una operación cuyo resultado, debido a operar con números en complemento a dos, cambia de signo, dando un resultado incorrecto. Veamos algún ejemplo:

```
mov eax, 7FFFFFFF       EAX=7FFFFFFF  (2147483647d)
add eax, 00000001       EAX=80000000h (-2147483648d)  OF=1
add eax, FFFFFFFF       EAX=7FFFFFFF  (2147483647d)   OF=1
```

Podemos ver fácilmente que estas dos sumas no son correctas, debido a que el resultado traspasa la línea del cambio de signo.

D (Dirección)

Si está a cero, las instrucciones que utilizan los registros ESI y EDI por operar con una serie de bytes consecutivos, los tomarán en el orden normal, del primero al último; en caso contrario, los tomaran del último al primero. Por lo tanto este flag no varía de acuerdo a los resultados de determinadas operaciones, sino que se le pone el valor adecuado a voluntad del programador, mediante las instrucciones `std` y `cld` que veremos más adelante.

I (Interrupción)

Cuando se pone a uno, el procesador ignora las peticiones de interrupción que puedan llegar de los periféricos. La finalidad de esto es la de evitar que durante la ejecución de algún proceso más o menos crítico, una interrupción pudiera provocar la inestabilidad del sistema. Esto supongo que en teoría debe servir para evitar las pantallas azules :-D

S (Signo)

Se pone a uno, cuando se efectúa una operación cuyo resultado es negativo, como en los ejemplos siguientes:

```
mov eax, 00007FFF       EAX=00007FFF
```

```
add ax, 0001            EAX=00008000h(AX=-1)          SF=1
mov eax, 00007FFF       EAX=00007FFF
add eax, 00000001       EAX=00008000h(EAX= 32768d)    SF=0
mov ax, 0012h           EAX=00000012h
sub ax, 002E            EAX=0000FFE4 (AX=-28)         SF=1
xor eax, EAX            EAX=00000000
dec eax                 EAX=FFFFFFFF (EAX=-1)         SF=1
```

Fijáos en la diferencia que hay en el resultado de los dos primeros ejemplos. En el primero, el resultado de la operación es negativo, porque se toma como referencia el valor de AX, que es el registro al que se refiere la instrucción ejecutada. En el segundo, en cambio, se toma como referencia el valor de EAX, por lo cual el resultado es positivo, así que hemos de tener en cuenta que el flag se activa o no, según el resultado de la operación efectuada, independientemente del valor del registro afectado por esa operación.

Z (Cero)

Se pone a uno, cuando se efectúa una operación cuyo resultado es cero. He aquí un ejemplo:

```
mov eax, FFFFFFFF
add eax, 00000001       EAX=-1+1=0        ZF=1
mov eax, FFFFFFFF
xor eax, eax            EAX=0             ZF=1
mov ax, 005A
sub ax, 005A            EAX=5A-5A=0       ZF=1
```

A (Auxiliar)

Este flag es similar al de acarreo que veremos enseguida, pero responde a las operaciones efectuadas con números en formato BCD.

P (Paridad)

Se pone a uno, cuando se efectúa una operación cuyo resultado contiene un número par de bits con el valor 1. Veamos un ejemplo:

```
mov eax, 00000007       EAX=00000007
add eax, 00000005       EAX=0000000C      PF=1  (C=1010)
inc eax                 EAX=0000000D      PF=0  (D=1101)
add eax, 00000006       EAX=00000013h     PF=0  (13h=10011)
inc eax                 EAX=00000014h     PF=1  (14h=10100)
xor eax, eax            EAX=00000000      PF=1
```

Como podemos ver, le atribuye paridad par al cero.

C (Carry o acarreo)

Se pone a uno, cuando se efectúa una operación que no cabe en el espacio correspondiente al resultado. Volvamos a ver lo que sucede con uno de los ejemplos anteriores:

```
mov eax, FFFFFFFF       EAX=FFFFFFFF
add eax, 00000001       EAX=00000000      CF=1
mov eax, FFFFFFFF       EAX=FFFFFFFF
add ax, 0001            EAX=FFFF0000      CF=1
```

7. INSTRUCCIONES

Ahora que ya estamos situados :-) es el momento de empezar a explicar con cierto detenimiento las instrucciones de los procesadores 80X86.

Hemos ido viendo varias de estas instrucciones al explicar el funcionamiento de los registros. Aunque no había explicado su funcionamiento creo que éste ha quedado bastante claro.

Antes de hablar de las instrucciones, veremos los distintos sistemas que tienen las instrucciones para acceder a los datos necesarios para su ejecución.

7.1. Modos de direccionamiento

La terminología que empleo aquí no es standard, por lo que puede ser que algunos términos no sean los mismos que los empleados en otros lugares.

Directo

El dato se refiere a una posición de la memoria, cuya dirección se escribe entre corchetes.

```
mov dword ptr [00513450], ecx
mov ax, word ptr [00510A25]
mov al, byte ptr [00402811]
```

O sea, que en el primer ejemplo, se copia el contenido del registro ECX, no en la dirección 513450, sino en la dirección que hay guardada en la posición 513450 de la memoria.

Aquí aparece una aclaración que, aunque la veremos en innumerables ocasiones, unas veces es necesaria y otras no tanto. Por esta razón, se representa o no según el programa que haga el desensamblado y, por supuesto, de la necesidad que haya de representarla. Se trata de los typecasts, es decir, de la especificación del tamaño del dato que se obtiene de la dirección de memoria indicada entre corchetes. Como vemos, aquí no es necesaria esta especificación, ya que la longitud del dato viene dada por el tipo de registro. En otros casos puede existir una ambigüedad que haga necesaria esta información. Por ejemplo, si tratamos de ensamblar la instrucción `mov [edx], 00000000` el compilador nos dará error porque no hemos indicado el tamaño del valor a introducir en EDX (aunque hayamos puesto ocho ceros), por lo que deberemos especificar: `mov dword ptr [edx], 0` lo que generaría una instrucción que nos pondría cuatro bytes a cero a partir de la dirección indicada por el registro EDX.

Indirecto

Aquí el dato también se refiere a una posición de la memoria, pero en vez de indicar la dirección como en el caso anterior, se indica mediante un registro que contiene dicha dirección.

```
mov eax, 00513450
mov dword ptr [eax], ecx
```

En este ejemplo, el resultado es el mismo que en el primer ejemplo del modo directo.

Registro

Los parámetros de la instrucción están contenidos en registros. Ejemplos:

```
inc eax
push ebx
xchg eax, ebx
```

Inmediato

Se le asigna un valor numérico a un operando de la instrucción:

```
cmp eax, 00000005
mov cl, FF
```

De base

Este direccionamiento emplea exclusivamente los registros EBX y EBP, que sumados a un valor numérico, nos dan la posición de memoria origen o destino del dato que maneja la instrucción.

```
cmp ebx, dword ptr [ebp+02]
mov edx, dword ptr [ebx+08]
```

De índice

Similar al anterior, pero empleando los registros de índice: ESI y EDI.

```
mov cl, byte ptr [esi+01]
cmp cl, byte ptr [edi+01]
```

Hay algún modo de direccionamiento más, pero creo que esto ya está resultando demasiado teórico :-(

7.2. La reina de las instrucciones :-)

Ahora por fin, pasamos a ver las instrucciones. Vamos a ver la mayoría, pero no todas, he dejado de lado las que me han parecido menos importantes, teniendo en cuenta la finalidad de este manual. Podéis obtener una información exhaustiva en el documento Intel Pentium Instruction Set Reference que se puede encontrar fácilmente en la Red.

Las instrucciones se representan de dos maneras: mediante el opcode o código de operación y el nombre o mnemónico. El código de operación es un grupo de cifras más o menos largo, según la instrucción, que en principio está pensado para uso del ordenador, aunque nosotros a veces metamos las zarpas :-) El mnemónico es el nombre de la instrucción, que trata de representar una descripción del significado de la misma. Para apreciar mejor esto, junto al mnemónico de la instrucción voy a poner el nombre en inglés.

A partir de ahora, en los ejemplos, pondré los códigos de operación de las instrucciones (simplemente a título orientativo) y sólo cuando sea conveniente (en el caso de saltos), la dirección en que están situadas.

nop (No Operation)

Esta instrucción es de gran utilidad: no hace nada. Su verdadera utilidad reside en rellenar los huecos provocados por la eliminación de instrucciones o su substitución por instrucciones de menor longitud. Su nombre da origen al verbo "nopear" que tan a menudo utilizan los chicos malos ;-) Hay corrientes filosóficas dentro del cracking que sostienen que poner `nops` es una cochinada y por ello substituyen por ejemplo, una instrucción de salto condicional de dos bytes como:

```
:0040283E 7501          jne 00402841
:00402840 C3            ret
:00402841 31C0          xor eax, eax
```

por:

```
:0040283E 40            inc eax
:0040283F 48            dec eax
:00402840 C3            ret
:00402841 31C0          xor eax, eax
```

en lugar de:

```
:0040283E 90            nop
:0040283F 90            nop
:00402840 C3            ret
:00402841 31C0          xor eax, eax
```

A mi no me gusta demasiado poner `nops`, pero tampoco me preocupa hacerlo. En este caso concreto, hay una solución que a mi me parece muy limpia:

```
:0040283E 7500          jne 00402840
:00402840 C3            ret
:00402841 31C0          xor eax, eax
```

Cambiando un solo byte, se consigue que tanto si se efectúa el salto como si no, se vaya a la instrucción siguiente. O sea, que es como si el salto no existiera. Pero todo esto es cuestión de gustos... De todos modos, me gustaría saber si los que se oponen al empleo de la instrucción `nop` saben que el opcode que corresponde a dicha instrucción (90) es en realidad el correspondiente a la instrucción `xchg eax,eax`, o sea que en realidad, la instrucción `nop` no existe, sino que se trata sólo de una una interpretación convencional. Otra cosa curiosa es que el opcode 6690 que corresponde a la instrucción `xchg ax,ax`, se desensambla también como `nop`.

7.3. Instrucciones de la pila

En la descripción de la pila, hemos visto que hay instrucciones que se utilizan para poner y quitar valores de ella, cambiando su dirección de inicio contenida en el registro ESP, de manera que siempre apunte al último valor introducido.

push (Push Word or Doubleword Onto the Stack)

Esta instrucción resta del registro ESP, la longitud de su único operando que puede ser de tipo word o doubleword (4 u 8 bytes), y a continuación lo coloca en la pila.

pop (Pop a Value from the Stack)

Es la inversa de la anterior, es decir que incrementa el registro ESP y retira el valor disponible de la pila y lo coloca donde indica el operando.

pushad /pusha (Push All General-Purpose Registers 32 / 16 bits)

Estas instrucciones guardan el contenido de los registros en la pila en un orden determinado. Así pues, `pushad` equivale a: `push` EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI y `pusha` equivale a: `push` AX, CX, DX, BX, SP, BP, SI, DI.

popad / popa (Pop All General-Purpose Registers 32 / 16 bits)

Estas instrucciones efectúan la función inversa de las anteriores, es decir, restituyen a los registros los valores recuperados de la pila en el orden inverso al que se guardaron. Así `popad` equivale a: `pop` EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX y `popa` equivale a: `pop` DI, SI, BP, SP, BX, DX, CX, AX (los valores recuperados correspondientes a ESP y SP, no se colocan en los registros sino que se descartan).

pushfd /pushf (Push EFLAGS Register onto the Stack 32 / 16 bits)
popfd / popf (Pop Stack into EFLAGS Register 32 / 16 bits)

Estas parejas de instrucciones colocan y retiran de la pila el registro de flags.

7.4. Instrucciones de transferencia de datos

mov (Move)

Esta instrucción tiene dos operandos. Copia el contenido del operando de origen (representado en segundo lugar) en el de destino (en primer lugar), y según el tipo de estos operandos adopta formatos distintos. He aquí unos ejemplos:

```
6689C8                  mov ax, cx
8BC3                    mov eax, ebx
8B5BDC                  mov ebx, dword ptr [ebx-24]
893438                  mov dword ptr [eax+edi], esi
```

movsx (Move with Sign-Extension)

Copia el contenido del segundo operando, que puede ser un registro o una posición de memoria, en el primero (de doble longitud que el segundo), rellenandose los bits sobrantes por la izquierda con el valor del bit más significativo del segundo operando. Aquí tenemos un par de ejemplos:

```
33C0                    xor eax, eax
BB78563412              mov ebx, 12345678
0FBFC3                  movsx eax, bx           EAX=00005678
33C0                    xor eax, eax
BBCDAB3412              mov ebx, 1234ABCD
0FBFC3                  movsx eax, bx           EAX=FFFFABCD
```

Vemos como en el primer ejemplo los espacios vacíos se rellenan con ceros y en el segundo con unos.

movzx (Move with Zero-Extend)

Igual a `movsx`, pero en este caso, los espacios sobrantes se rellenan siempre con ceros. Veamos como el segundo ejemplo de la instrucción anterior da un resultado distinto:

```
33C0                    xor eax, eax
BBCDAB3412              mov ebx, 1234ABCD
0FB7C3                  movzx eax, bx          EAX=0000ABCD
```

lea (Load Effective Address)

Similar a la instrucción `mov`, pero el primer operando es un registro de uso general y el segundo una dirección de memoria. Esta instrucción es útil sobre todo cuando esta dirección de memoria responde a un cálculo previo.

```
8D4638                  lea eax, dword ptr [esi+38]
```

xchg (Exchange Register/Memory with Register)

Esta instrucción intercambia los contenidos de los dos operandos.

```
87CA                    xchg edx, ecx
870538305100            xchg dword ptr [00513038], eax
8710                    xchg dword ptr [eax], edx
```

En el primer ejemplo, el contenido del registro ECX, se copia en el registro EDX, y el contenido anterior de EDX, se copia en ECX. Se obtendría el mismo resultado con:

```
51                      push ecx
52                      push edx
59                      pop ecx
5A                      pop edx
```

La instrucción `pop ecx` toma el valor que hay en la pila y lo coloca en el registro ECX, pero como podemos ver por la instrucción anterior `push edx`, este valor, procedía del registro EDX. Luego se coloca en EDX el valor procedente de ECX.

bswap (Byte Swap)

Esta instrucción es para empleo exclusivo con registros de 32 bits como único parámetro. Intercambia los bits 0 a 7 con los bits 24 a 31, y los bits 16 a 23 con los bit 8 a 15.

```
B8CD34AB12              mov eax, 12AB34CD      EAX=12AB34CD
0FC8                    bswap eax              EAX=CD34AB12
```

7.5. Instrucciones aritméticas

inc (Increment by 1) / dec (Decrement by 1)

Estas dos instrucciones incrementan y decrementan respectivamente el valor indicado en su único operando.

```
FF0524345100            inc dword ptr [00513424]
FF0D24345100            dec dword ptr [00513424]
40                      inc eax
4B                      dec ebx
```

En los dos primeros ejemplos, se incrementaría y decrementaría el valor contenido en los cuatro bytes situados a partir de la dirección 513424.

add (Add)

Esta instrucción suma los contenidos de sus dos operandos y coloca el resultado en el operando representado en primer lugar.

```
02C1        add al, cl                    AL + CL -> AL
01C2        add edx, eax                  EDX + EAX -> EDX
8345E408    add dword ptr [ebp-1C], 0000008  dword ptr [EBP-1C]+ 8 -> [EBP-1C]
```

En el tercer ejemplo, como el resultado se coloca siempre en el primer operando, no sería aceptada por el compilador una instrucción como `add 00000008, dword ptr [ebp-1C]`

adc (Add with Carry)

Esta instrucción es similar a la anterior, con la diferencia de que se suma también el valor del flag de acarreo. Se utiliza para sumar valores mayores de 32 bits. Supongamos que queremos sumar al contenido de los registros EDX:EAX (EDX=00000021h y EAX=87AE43F5), un valor de más de 32 bits (3ED671A23). Veamos como se hace:

```
add eax, ED671A23       EAX=75155E18      (87AE43F5+ED671A23)     CF=1
adc edx, 0000003        EDX=25h           (21h+3+1)
```

sub (Subtract)

Esta instrucción resta el contenido del segundo operando del primero, colocando el resultado en el primer operando.

```
83EA16      sub edx, 00000016             EDX - 16 -> EDX
29C8        sub eax, ecx                  EAX - ECX -> EAX
2B2B        sub ebp, dword ptr [ebx]      EBP - dword ptr [EBX] -> EBP
```

sbb (Integer Subtraction with Borrow)

Esta instrucción es una resta en la que se tiene en cuenta el valor del flag de acarreo. Supongamos que del contenido de los registros EDX:EAX después del ejecutado el ejemplo de la instrucción `adc` (EDX=00000025h y EAX=75155E18), queremos restar el valor 3ED671A23. El resultado es el valor que tenían inicialmente los dos registros en el ejemplo citado:

```
sub eax, ED671A23       EAX=87AE43F5      (75155E18-ED671A23)     CF=1
sbb edx, 0000003        EDX=21h           (25h-3-1)
```

mul (Unsigned Multiply) / imul (Signed Multiply)

Estas dos instrucciones se utilizan para multiplicar dos valores. La diferencia más importante entre las dos, es que en la primera no se tiene en cuenta el signo de los factores, mientras que en la segunda sí. Como veremos en algunos ejemplos, esta diferencia se refleja en los valores de los flags.
En la instrucción `mul`, hay un solo operando. Si es un valor de tamaño byte, se multiplica este valor por el contenido de AL y el resultado se guarda en EAX, si el valor es de tamaño word (2 bytes), se multiplica por AX, y el resultado se guarda en EAX y finalmente, si el valor es de tamaño dword (4bytes), se multiplica por EAX y el resultado se guarda en EDX:EAX. O sea, que el espacio destinado al resultado siempre es de tamaño doble al de los operandos.

```
F7E1        mul ecx                 EAX*ECX -> EDX:EAX
F72424      mul dword ptr [esp]     EAX*[ESP] -> EDX:EAX
```

En la instrucción `imul`, hay también una mayor variedad en el origen de sus factores. Además de la utilización de los registros EAX y EDX, así como de sus subdivisiones, pueden especificarse otros orígenes y destinos de datos y puede haber hasta tres operandos. El primero, es el lugar donde se va a

guardar el resultado, que debe ser siempre un registro, el segundo y el tercero son los dos valores a multiplicar. En estos ejemplos vemos como estas instrucciones con dos o tres operandos, tienen el mismo espacio para el resultado que para cada uno de los factores:

```
F7EB            imul ebx                              EAX x EBX -> EDX:EAX
696E74020080FF  imul ebp, dword ptr [esi+74], FF800002  [ESI+74] x FF800002 -> EBP
0FAF55E8        imul edx, dword ptr [ebp-18]          EDX x [EBP-18] -> EDX
```

Veamos finalmente la diferencia entre la multiplicación con signo y sin él:

```
66B8FFFF    mov ax, FFFF      AX=FFFF
66BBFFFF    mov bx, FFFF
66F7E3      mul bx            AX=0001     OF=1  CF=1
66B8FFFF    mov ax, FFFF
66F7EB      imul bx           AX=0001     OF=0  CF=0
```

En la primera operación, como se consideran los números sin signo, se ha multiplicado 65535d por 65535d y el resultado ha sido 1. Debido a este resultado "anómalo", se han activado los flags OF y CF. En cambio, en la segunda operación, en la que se toma en cuenta el signo, se ha multiplicado -1 por -1 y el resultado ha sido 1. En este caso no se ha activado ningún flag. Otro ejemplo:

```
B8FFFFFF7F  mov eax, 7FFFFFFF
BB02000000  mov ebx, 00000002
F7E3        mul ebx                 EAX=FFFFFFFE      OF=0  CF=0
B8FFFFFF7F  mov eax, 7FFFFFFF
F7EB        imul ebx                EAX=FFFFFFFE      OF=1  CF=1
```

Esta vez, en el primer caso, el resultado ha sido correcto, porque 2147483647d multiplicado por dos ha dado 4294967294d, por tanto, no se ha activado ningun flag. Pero en el segundo caso, teniendo en cuenta el signo, hemos multiplicado 2147483647d por dos y nos ha dado como resultado -2. Ahora si se han activado los flags.

div (Unsigned Divide) / idiv (Signed Divide)

El caso de la división es muy parecido al de la multiplicación. Hay dos instrucciones: `div` para números en los que no se considere el signo e `idiv` para números que se consideren con signo. El dividendo está formado por una pareja de registros y el divisor es el único operando. He aquí varios ejemplos de una y otra instrucción:

```
66F7F3      div bx                  DX:AX : BX -> AX          resto -> DX
F7F3        div ebx                 EDX:EAX : EBX -> EAX      resto -> EDX
F77308      div dword ptr [ebx+08]  EDX:EAX : [EBX+8] -> EAX  resto -> EDX
F7F9        idiv ecx                EDX:EAX : ECX -> EAX      resto -> EDX
```

Ahora, como hemos hecho con la multiplicación, vamos a ver el diferente resultado que se obtiene empleando una u otra instrucción:

```
33D2        xor edx, edx
66B80100    mov ax, 0001
66BBFFFF    mov bx, FFFF
66F7F3      div bx                  AX=0000     DX=0001
33D2        xor edx, edx
66B80100    mov ax, 0001
66F7FB      idiv bx                 AX=FFFF     DX=0000
```

En el primer caso, al no considerar el signo de los números, se ha dividido 1 por 65535, que ha dado un cociente de 0 y un resto de 1. En el segundo caso, se ha dividido -1 por 1, lo que ha dado un cociente de -1 y un resto de 0. No ha habido overflow ni acarreo en ninguno de los dos casos.

xadd (Exchange and Add)

Intercambia los valores de los dos operandos y los suma, colocando el resultado en el primer operando. El primer operando puede ser un registro o una posición de memoria, pero el segundo sólo puede ser un

registro.

```
C703CDAB3412            mov dword ptr [ebx], 1234ABCD
```

En la dirección indicada por EBX tendremos el valor CD AB 34 12.

Vemos el valor que hemos puesto en la memoria invertido, porque el paso del valor de un registro a la memoria y viceversa se hace empezando por el último byte y terminando por el primero.

```
B8CD34AB12              mov eax, 12AB34CD
B934120000              mov ecx, 00001234
0FC1C8                  xadd eax, ecx
```

EAX contiene el valor 12AB4701 (12AB34CD+1234) y ECX el valor 12AB34CD.

```
B934120000              mov ecx, 00001234
0FC10B                  xadd dword ptr [ebx], ecx
```

La dirección indicada por EBX contiene el valor 01 BE 34 12 (1234ABCD+1234) y el registro ECX el valor 1234ABCD.

aaa (Ascii Adjust for Addition) / daa (Decimal Adjust AL after Addition)

Antes hemos mencionado el formato BCD, ahora vamos a operar con él. Mediante la primera de estas dos instrucciones, se ajusta el resultado de una suma de números en formato BCD desempaquetado, contenido en AL. La segunda instrucción hace lo mismo pero con formato BCD empaquetado. Vamos a ver un par de ejemplos:

```
33C0        xor eax, eax
33DB        xor ebx, ebx
B007        mov al, 07
B309        mov bl, 09
02C3        add al, bl          AL=10       10h=7+9
37          aaa                 AX=0106     16d(7+9)en BCD desempaq.)
33C0        xor eax, eax
B065        mov al, 65
B328        mov bl, 28
02C3        add al, bl          AL=8D       (8Dh=65h+28h)
27          daa                 AL=93       93d(65d+28d)en BCD empaquetado.)
```

Podemos ampliar un poco las posibilidades de daa aprovechando el flag de acarreo:

```
B065        mov al, 65
B398        mov bl, 98
02C3        add al, bl          AL=FD       (FDh=65h+98h)
27          daa                 AL=63       CF=1
80D400      adc ah, 00          AX=0163     163d(65d+98d)en BCD empaquetado.)
```

aas (ASCII Adjust AL After Subtraction) / das (Decimal Adjust AL after Subtraction)

Estas instrucciones hacen lo mismo, pero después de una resta. Personalmente no les veo ninguna utilidad, pero si están ahí por algo será...

aam (Ascii Adjust AX After Multiply)

Otra instrucción de dudosa utilidad. Ajusta el resultado de una multiplicación entre dos números de una cifra en formato BCD desempaquetado. Sin embargo, tiene una interesante posibilidad: la de poder trabajar en la base que se defina modificando MANUALMENTE (resulta que no hay un mnemónico para eso) el segundo byte del código que normalmente tiene el valor 0Ah (10d). Veamos unos ejemplos:

```
33C0        xor eax, eax
33DB        xor ebx, ebx
B009        mov al, 09
```

```
B305        mov bl, 05
F6E3        mul bl            AL=2D
D40A        aam               AX=0405     45d(9x5)en BCD desempaq.)
33C0        xor eax, eax
B009        mov al, 09
B305        mov bl, 05
F6E3        mul bl            AL=2D
D408        aam (base8)       AX=0505     55q=45d(9x5)en BCD desempaq.)
```

Lo de (base8) no lo he puesto yo, lo ha puesto el W32dasm :-o

aad (Ascii Adjust AX Before Division)

Más de lo mismo, pero con la división. La diferencia está en que se ejecuta la instrucción antes de dividir.

```
33C0        xor eax, eax
33DB        xor ebx, ebx
66B80307    mov ax, 0703
D50A        aad               EAX=0049    49h=73d
B308        mov bl, 08
F6F3        div bl            EAX=0109    73d/8=9 (AL) resto=1 (AH)
33C0        xor eax, eax
33DB        xor ebx, ebx
66B80307    mov ax, 0703
D508        aad (base=8)      EAX=003B 3Bh=73q
B308        mov bl, 08
F6F3        div bl            AL=0307     73q/8=7 (AL) resto=3 (AH)
```

La mayor, por no decir la única utilidad de estas instrucciones, es la de pasar un número en base hexadecimal a otra base.

neg (Two's Complement Negation)

Esta instrucción tiene la finalidad de cambiar de signo el número representado por su único operando, mediante una operación de complemento a dos.

```
B81C325100              mov eax, 0051321C
F7D8                    neg eax                 EAX=FFAECDE4

      neg   0000 0000 0101 0001 0011 0010 0001 1100 = 0051321C
            1111 1111 1010 1110 1100 1101 1110 0011
        +                                       1
            1111 1111 1010 1110 1100 1101 1110 0100 = FFAECDE4
```

7.6. Instrucciones lógicas

and (Logical AND)

Efectúa una operación AND entre cada uno de los bits de los dos operandos.

```
B8DAC70704  mov eax, 0407C7DA
25C30E6F00  and eax, 006F0EC3       EAX=000706C2

      0407C7DA =  0000 0100 0000 0111 1100 0111 1101 1010
      006F0EC3 =  0000 0000 0110 1111 0000 1110 1100 0011
             and  0000 0000 0000 0111 0000 0110 1100 0010  =  000706C2

25FF000000  and eax, 000000FF       EAX=000000C2

      000706C2 =  0000 0000 0000 0111 0000 0110 1100 0010
      000000FF =  0000 0000 0000 0000 0000 0000 1111 1111
             and  0000 0000 0000 0000 0000 0000 1100 0010  =  000000C2
```

Hacer un AND de cualquier byte con FF lo deja como estaba. Por esta razón, se utilizan frecuentemente instrucciones de este tipo para eliminar bytes que no se van a utilizar de un registro.

or (Logical Inclusive OR)

Efectúa una operación OR entre cada uno de los bits de los dos operandos.

```
B8DAC70704  mov eax, 0407C7DA
0DC30E6F00  or eax, 006F0EC3        EAX=046FCFDB

      0407C7DA =  0000 0100 0000 0111 1100 0111 1101 1010
      006F0EC3 =  0000 0000 0110 1111 0000 1110 1100 0011
             or   0000 0100 0110 1111 1100 1111 1101 1010  =  046FCFDB
```

xor (Logical Exclusive OR)

Efectúa una operación XOR entre cada uno de los bits de los dos operandos.

```
B8DAC70704  mov eax, 0407C7DA
35C30E6F00  xor eax, 006F0EC3       EAX=0468C919

      0407C7DA =  0000 0100 0000 0111 1100 0111 1101 1010
      006F0EC3 =  0000 0000 0110 1111 0000 1110 1100 0011
             xor  0000 0100 0110 1000 1100 1001 0001 1001  =  0468C919
```

not (One's Complement Negation)

Efectúa una operación NOT con cada uno de los bits del único operando.

```
B8DAC70704  mov eax, 0407C7DA
F7D0        not eax           EAX= FBF83825

      0407C7DA =  0000 0100 0000 0111 1100 0111 1101 1010
             not  1111 1011 1111 1000 0011 1000 0010 0101  =  FBF83825
```

7.7. Instrucciones de comprobación y verificación

Ciertamente, aquí he hecho una agrupación un tanto arbitraria de instrucciones...

cmp (Compare Two Operands)

La comparación entre dos valores es en realidad una resta entre ambos. Según cual sea el resultado, podemos saber si los valores son iguales y en caso contrario, cual de ellos es el mayor. Así, se podría utilizar la instrucción `sub ecx, ebx` para comparar el resultado de estos dos registros, sin embargo el hacerlo así tiene el problema de que el resultado de la resta se colocaría en el registro ECX, cuyo valor anterior desaparecería. Para evitar este problema el programador dispone de la instrucción `cmp`.

Esta instrucción resta el segundo operando del primero. El resultado no se guarda en ningún sitio, pero según cual sea este resultado, pueden modificarse los valores de los flags CF, OF, SF, ZF, AF y PF. Es en base al estado de estos flags, que se efectúa o no el salto condicional que suele acompañar a esta instrucción. Veremos esta instrucción en los ejemplos de saltos condicionales.

cmpxchg (Compare and Exchange)

Esta instrucción compara el valor de AL, AX o EAX (según el tamaño de los operandos) con el primer operando. Si son iguales, se pone a uno el flag de cero y el segundo operando se copia en el primero; en caso contrario, se pone a cero el flag de cero y el segundo operando se copia en AL, AX o EAX. A ver si con un ejemplo se entiende:

```
B81CA23456              mov eax, 5634A21C
```

```
BB1CA23456              mov ebx, 5634A21C
B9CDAB3412              mov ecx, 1234ABCD
0FB1CB                  cmpxchg ebx, ecx       EBX=1234ABCD      ZF=1
0FB1CB                  cmpxchg ebx, ecx       EAX=1234ABCD      ZF=0
```

cmpxchg8b (Compare and Exchange 8 Bytes)

Esta instrucción es parecida a la anterior, pero funciona con parejas de registros (64bits). Se comparan EDX:EAX con el único operando, que debe ser una zona de memoria de 8 bytes de longitud. Si son iguales, se copia el valor de los registros ECX:EBX en la zona de memoria indicada por el operando; en caso contrario, el contenido de dicha zona se guarda en EDX:EAX.

He estado a punto de no incluir la instrucción anterior por no parecerme de interés para este manual. Con mayor razón he dudado en poner esta última, pero al ver que el W32Dasm no era capaz de desensamblarla he tenido la certeza de que me la podía haber ahorrado. En fin, ya que está puesta...

test (Logical Compare)

El principio de esta instrucción es, en cierto modo, el mismo de `cmp`, es decir, una operación entre dos valores que no se guarda, sino que puede modificar el estado de algunos flags (en este caso, SF, ZF y PF) que determinan si debe efectuarse el salto que también suele acompañar a esta instrucción. La diferencia está en que en este caso, en vez de tratarse de una resta, se trata de una operación AND.

Esta instrucción se utiliza para averiguar si determinados bits de uno de los operandos están a 1 o 0, empleando una máscara como segundo operando. También se utiliza para saber si un valor es cero, comprobándolo consigo mismo. Es el caso de este código que resulta tan familiar:

```
E89DFFFFFF              call 0040143C
85C0                    test eax, eax
7505                    jne 004014A8
```

Si EAX es igual a cero, el resultado sera cero; pero si es distinto de cero, al hacer un AND de un bit de valor uno consigo mismo, el valor sera uno y, en este caso concreto, se produciría el salto. Nos hemos adelantado al poner en el ejemplo la instrucción `jne`, pero enseguida veremos su significado.

bt (Bit Test)

Esta instrucción comprueba el valor del bit indicado por el segundo operando en el primer operando y lo copia en el flag de acarreo (CF).

```
0FA30424                bt dword ptr [esp], eax
```

Este ejemplo real formaba parte de un bucle. EAX iba tomando los valores del código ASCII de los caracteres del serial introducido, y comprobaba el valor de los bits a partir de la dirección de inicio de una tabla (en ESP), en la que los caracteres "permitidos" estaban representados por unos. Veremos esto con más detalle en el Apéndice B del manual.

btc (Bit Test and Compliment)

Esta instrucció es similar a la anterior, pero se diferencia en que después de copiar en el flag de acarreo el valor del bit comprobado, pone en este bit el complemento de su valor, es decir, el valor inverso. Veamos en un ejemplo la comparación con la instrucción anterior:

```
B8CD34AB12              mov eax, 12AB34CD
0FBAE010                bt eax, 10

   bit 31            bit 16 (10h)         bit 0
      |                  |                   |
EAX = 0001 0010 1010 1011 0011 0100 1100 1101
```

Después de efectuar esta instrucción, el registro EAX queda como estaba y el flag de acarreo queda con el valor del bit 16 (CF=1).

```
0FBAF810                btc eax, 10
```

Después de efectuar esta otra instrucción, el bit 16 del registro EAX invierte su valor, pasando a valer 0 (EAX=12AA34CD) y el flag de acarreo queda con el valor anterior del bit 16 (CF=1).

btr (Bit Test and Reset)

Una nueva variante. Esta vez guarda en el flag de acarreo el valor del bit comprobado, pero pone este bit a cero.

```
B8CD34AB12              mov eax, 12AB34CD
0FBAF010                btr eax, 10
```

El resultado de efectuar esta instrucción es el mismo que el del anterior ejemplo (EAX=12AA34CD y CF=1).

bts (Bit Test and Set)

Igual a la anterior, pero pone el bit comprobado a uno.

```
B8CD34AB12              mov eax, 12AB34CD
0FBAE810                bts eax, 10
```

En este caso, EAX no varía y el flag de acarreo queda puesto a uno.

bsf (Bit Scan Forward)

Esta instrucción busca en el segundo operando, que puede ser un registro o una posición de memoria, el bit menos significativo cuyo valor sea igual a uno y coloca en el primer operando, que debe ser un registro, la posición que ocupa este bit empezando por el cero, desde la derecha. Con un ejemplo se verá mejor:

```
BBC0B3A201              mov ebx, 01A2B3C0
0FBCC3                  bsf eax, ebx            EAX=6

  0    1    A    2    B    3    C    0
0000 0001 1010 0010 1011 0011 1100 0000
|                             |      |
Bit 20h                     Bit 6  Bit 0h
```

bsr (Bit Scan Reverse)

Esta instrucción es similar a la anterior, pero buscando el bit más significativo puesto a uno. Veamos un ejemplo:

```
BBC0B3A201              mov ebx, 01A2B3C0
0FBDC3                  bsr eax, ebx            EAX=18

  0    1    A    2    B    3    C    0
0000 0001 1010 0010 1011 0011 1100 0000
|       |                           |
Bit 20h   Bit 18h                   Bit 0
```

7.8. Instrucciones de salto

El listado de un programa consiste en una sucesión de instrucciones. Sin embargo a la hora de ejecutarlo, la ejecución del mismo no sigue el orden del listado, sino que, de acuerdo con distintas circunstancias y mediante las instrucciones de salto, se interrumpe la ejecución lineal del programa para continuar dicha ejecución en otro lugar del código.

Las instrucciones de salto son básicamente de dos tipos: de salto condicional y de salto incondicional. En

el primer tipo, la instrucción de salto suele ponerse después de una comparación y el programa decide si se efectúa o no el salto, según el estado de los flags (excepto en un caso, en el que el salto se efectúa si el valor del registro ECX o CX es cero). En el segundo, el salto se efectúa siempre.

Tipos de salto

Según la distancia a la que se efectúe el salto, estos se dividen en tres tipos: corto, cercano y lejano. También se dividen en absolutos o relativos, según como se exprese en la instrucción la dirección de destino del salto. Como veremos, los saltos cortos y cercanos son relativos, y los largos absolutos.

En el salto corto, la dirección de destino se expresa mediante un byte, que va a continuación del código de la instrucción. Este byte contiene un número con signo que, sumado a la dirección de la instrucción siguiente a la del salto, nos da la dirección de destino del mismo. Como este número es con signo, podemos deducir que un salto corto sólo puede efectuarse a una distancia hacia adelante de 127 bytes y hacia atras de 128. Veamos dos ejemplos de salto corto:

```
:004011E5 83FB05                cmp ebx, 00000005
:004011E8 7505                  jne 004011EF
:004011EA C645002D              mov [ebp+00], 2D
...
:004011EF 83FB09                cmp ebx, 00000009
:004011F2 72E2                  jb 004011D6
:004011F4 58                    pop eax
```

En el primer salto, la dirección de destino es la suma de la dirección de la instrucción siguiente y el desplazamiento: 4011EA+5=4011EF. En el segundo caso, E2 es un número negativo, por lo que la dirección de destino es la de la instrucción siguiente menos la equivalencia en positivo de E2 (1E): 4011F4-1E=4011D6. Estos cálculos, por supuesto, no los hace el programador, que simplemente dirige el salto a una etiqueta para que luego el compilador coloque los códigos correspondientes, pero me ha parecido que valía la pena explicarlo aquí.

El salto cercano es básicamente lo mismo que el salto corto. La diferencia está en que la distancia a que se efectúa es mayor, y no se puede expresar en un byte, por lo que se dispone de cuatro bytes (en programas de 16 bits) que permiten saltos de 32767 bytes hacia adelante y 32768 hacia atrás o de ocho bytes (en programas de 32 bits) que permiten vertiginosos saltos de 2147483647 bytes hacia adelante y 21474836478 bytes hacia atrás.

```
:0040194F 0F8E96000000          jle 004019EB
:00401955 8D4C2404              lea ecx, dword ptr [esp+04]
...
:004019CB 0F8566FFFFFF          jne 00401937
:004019D1 8D4C240C              lea ecx, dword ptr [esp+0C]
```

En la primera instrucción, la dirección de destino es: 401955+96= 4019EB. En la segunda la dirección es: 4019D1-9A= 401937. Fijáos en que los bytes que indican el desplazamiento estan escritos al revés y que 9A es la equivalencia en positivo de FFFFFF66.

Los saltos largos se utilizan cuando la instrucción de destino está en un segmento distinto al de la instrucción de salto.

```
:0003.0C28 2EFFA72D0C           jmp word ptr cs:[bx+0C2D]
```

Saltos Condicionales

Las instrucciones de salto condicional sólo admiten los formatos de salto corto y salto cercano, por lo que su código está formado por el código de la instrucción más un byte (cb), un word (cw) o un doubleword (cd) que determinan el desplazamiento del salto.

Aquí tenéis una relación de las instrucciones de salto condicional, desglosadas según el tipo de salto, en la que pueden apreciarse las equivalencias entre instrucciones de nombre distinto pero idéntica función.

Estas instrucciones equivalentes tienen el mismo código y se desensamblan con un nombre u otro, dependiendo del desensamblador. Por ejemplo, el SoftIce identifica una instrucción de código 75 cb como

`jnz`, y en cambio el W32dsam lo hace como `jne`.

Salto corto:

77 cb	JA rel8	Si es superior	(CF=0 y ZF=0)
	JNBE rel8	Si no es inferior o igual	(CF=0 y ZF=0)
73 cb	JAE rel8	Si es superior o igual	(CF=0)
	JNB rel8	Si no es inferior	(CF=0)
	JNC rel8	Si no hay acarreo	(CF=0)
76 cb	JNA rel8	Si no es superior	(CF=1 o ZF=1)
	JBE rel8	Si es inferior o igual	(CF=1 o ZF=1)
72 cb	JNAE rel8	Si no es superior o igual	(CF=1)
	JB rel8	Si es inferior	(CF=1)
	JC rel8	Si hay acarreo	(CF=1)
7F cb	JG rel8	Si es mayor	(ZF=0 y SF=OF)
	JNLE rel8	Si no es menor o igual	(ZF=0 y SF=OF)
7D cb	JGE rel8	Si es mayor o igual	(SF=OF)
	JNL rel8	Si no es menor	(SF=OF)
7E cb	JNG rel8	Si no es mayor	(ZF=1 o SF<>OF)
	JLE rel8	Si es menor o igual	(ZF=1 o SF<>OF)
7C cb	JNGE rel8	Si no es mayor o igual	(SF<>OF)
	JL rel8	Si es menor	(SF<>OF)
74 cb	JE rel8	Si es igual	(ZF=1)
	JZ rel8	Si es cero	(ZF=1)
75 cb	JNE rel8	Si no es igual	(ZF=0)
	JNZ rel8	Si no es cero	(ZF=0)
70 cb	JO rel8	Si hay desbordamiento	(OF=1)
71 cb	JNO rel8	Si no hay desbordamiento	(OF=0)
7A cb	JP rel8	Si hay paridad	(PF=1)
	JPE rel8	Si es paridad par	(PF=1)
7B cb	JNP rel8	Si no hay paridad	(PF=0)
	JPO rel8	Si es paridad impar	(PF=0)
78 cb	JS rel8	Si es signo negativo	(SF=1)
79 cb	JNS rel8	Si no es signo negativo	(SF=0)
E3 cb	JCXZ rel8	Si CX=0	
	JECXZ rel8	Si ECX=0	

Salto cercano:

0F 87 cw/cd	JA rel16/32	Si es superior	(CF=0 y ZF=0)
	JNBE rel16/32	Si no es inferior o igual	(CF=0 y ZF=0)
0F 83 cw/cd	JAE rel16/32	Si es superior o igual	(CF=0)
	JNB rel16/32	Si no es inferior	(CF=0)
	JNC rel16/32	Si no hay acarreo	(CF=0)
0F 86 cw/cd	JNA rel16/32	Si no es superior	(CF=1 o ZF=1)
	JBE rel16/32	Si es inferior o igual	(CF=1 o ZF=1)

```
0F 82 cw/cd  JNAE rel16/32    Si no es superior o igual     (CF=1)
             JB rel16/32      Si es inferior                (CF=1)
             JC rel16/32      Si hay acarreo                (CF=1)

0F 8F cw/cd  JG rel16/32      Si es mayor                   (ZF=0 y SF=OF)
             JNLE rel16/32    Si no es menor o igual        (ZF=0 y SF=OF)

0F 8D cw/cd  JGE rel16/32     Si es mayor o igual           (SF=OF)
             JNL rel16/32     Si no es menor                (SF=OF)

0F 8E cw/cd  JNG rel16/32     Si no es mayor                (ZF=1 o SF<>OF)
             JLE rel16/32     Si es menor o igual           (ZF=1 o SF<>OF)

0F 8C cw/cd  JNGE rel16/32    Si no es mayor o igual        (SF<>OF)
             JL rel16/32      Si es menor                   (SF<>OF)

0F 84 cw/cd  JE rel16/32      Si es igual                   (ZF=1)
             JZ rel16/32      Si es cero                    (ZF=1)

0F 85 cw/cd  JNE rel16/32     Si no es igual                (ZF=0)
             JNZ rel16/32     Si no es cero                 (ZF=0)

0F 80 cw/cd  JO rel16/32      Si hay desbordamiento         (OF=1)

0F 81 cw/cd  JNO rel16/32     Si no hay desbordamiento      (OF=0)

0F 8A cw/cd  JP rel16/32      Si hay paridad                (PF=1)
             JPE rel16/32     Si es paridad par             (PF=1)

0F 8B cw/cd  JNP rel16/32     Si no hay paridad             (PF=0)
             JPO rel16/32     Si es paridad impar           (PF=0)

0F 88 cw/cd  JS rel16/32      Si es signo negativo          (SF=1)

0F 89 cw/cd  JNS rel16/32     Si no es signo negativo       (SF=0)
```

NOTA: La diferencia entre mayor/menor y superior/inferior es la de que mayor/menor se refiere al resultado de la comparación entre números con signo, y superior/inferior al de la comparación entre números sin signo.

Bueno, una vez relacionados todos los saltos habidos y por haber, pensemos que en general los únicos saltos que nos interesan son `jz/je` o `jnz/jne`, `jg` o `jng` y `jle` o `jnle`. La primera pareja de saltos se efectúa para aceptar que el programa está registrado o para admitir como correcto el serial number que hemos introducido. Las otras dos las podemos encontrar en comprobaciones de los días que restan del período de prueba de un programa.

Veamos ahora el pedazo de código más inútil que he escrito jamás, pero que contiene tres ejemplos de comparación. Veremos el estado de los flags afectados por el resultado y los saltos que se ejecutan según este estado. Como se puede ver, me he limitado a los saltos más corrientes.

```
:00401144 B81F126700          mov eax, 0067121F
:00401149 3D998C1900          cmp eax, 00198C99     SF=0  CF=0  OF=0  ZF=0
```

El estado de los flags hace que puedan efectuarse estos cinco saltos:

```
:0040114E 7700                ja 00401150
:00401150 7300                jae 00401152
:00401152 7F00                jg 00401154
:00401154 7D00                jge 00401156
:00401156 7500                jne 00401158
```

Otra comparación:

```
:00401158 3DE1019300          cmp eax, 009301E1     SF=1  CF=1  OF=0  ZF=0
```

Y los cinco saltos posibles:

```
:0040115D 7200                jb 0040115F
:0040115F 7600                jbe 00401161
:00401161 7C00                jl 00401163
:00401163 7E00                jle 00401165
:00401165 7500                jne 00401167
```

La última comparación:

```
:00401167 3D1F126700          cmp eax, 0067121F     SF=0  CF=0  OF=0 ZF=1
```

Y los últimos cinco saltos:

```
:0040116C 7400                je 0040116E
:0040116E 7300                jae 00401170
:00401170 7600                jbe 00401172
:00401172 7D00                jge 00401174
:00401174 7E00                jle 00401176
```

También querría insistir en mis ataques contra la fea costumbre de invertir los saltos condicionales para registrar un programa. Debemos obligar al programa a que haga lo que queremos que haga, por lo que si ha de saltar para quedar registrado, debemos substituir el salto condicional por uno incondicional y si no debe saltar, substituir el salto condicional por nops u otra solución más elegante. Por ejemplo:

```
:00401135 3BCB                   cmp ecx, ebx
:00401137 7547                   jne 00401180        No queremos que salte
:00401139 84C0                   test al, al
:0040113B 7443                   je 00401180         Queremos que salte
...
:00401149 3BCB                   cmp ecx, ebx
:0040114B 0F85E3000000           jne 00401234        No queremos que salte
:00401151 84C0                   test al, al
:00401153 0F84DB000000           je 00401234         Queremos que salte
:00401159 40                     inc eax
```

Se puede substituir por:

```
:00401135 3BCB                   cmp ecx, ebx
:00401137 EB00                   jmp 00401139        Salta a la instrucción siguiente
:00401139 84C0                   test al, al
:0040113B EB43                   jmp 00401180        Salta siempre
...
:00401149 3BCB                   cmp ecx, ebx
:0040114B EB00                   jmp 0040114D        Salta a la instrucción siguiente
:0040114D EB00                   jmp 0040114F        Salta a la instrucción siguiente
:0040114F EB00                   jmp 00401151        Salta a la instrucción siguiente
:00401151 84C0                   test al, al
:00401153 90                     nop
:00401154 E9DB000000             jmp 00401234        Salta siempre
:00401159 40                     inc eax
```

Como veis el resultado es el mismo. Fijáos en los tres saltos seguidos para evitar poner seis nops (o tres nops "dobles" 6690). Esta solución está dedicada a los que no les gusta poner nops, aunque a mi me parece peor el remedio que la enfermedad.

Salto incondicional

Es un salto que no está sujeto a ninguna condición, es decir, que se efectúa siempre. Hay una sola instrucción: `jmp`. Esta instrucción admite los tres tipos de salto: corto, cercano y lejano. Creo que con unos ejemplos será suficiente:

```
:004011DA  EB30                  jmp 0040120C
:00402B35  E9F8000000            jmp 00402C32
:0001.02B4 E94D01                jmp 0404
:0003.0C28 2EFFA72D0C            jmp word ptr cs:[bx+0C2D]
```

7.9. Instrucciones de subrutinas

Las subrutinas son grupos de instrucciones con una finalidad concreta, que pueden ser llamadas desde uno o varios puntos del programa, al que regresan después de la ejecución de dicha subrutina. La utilización de subrutinas permite una mejor estructuración del programa al dividirlo en bloques independientes y evita la repetición del código de procesos que se ejecutan varias veces a lo largo de la ejecución del programa.

Para la ejecución de una subrutina, son necesarios algunos parámetros que pueden estar en registros, en posiciones de la memoria o en la pila. En este último caso (generalmente en programas escritos en lenguajes de alto nivel), la llamada a la subrutina está precedida de algunas instrucciones `push` que tienen la misión de colocar esos parámetros en la pila. Según el lenguaje empleado en la programación, a veces es necesario ajustar la pila, modificando directamente el valor del registro ESP o añadiendo un parámetro a la instrucción `ret`, consistente en el número que hay que sumar al registro ESP para que la dirección de retono sea la correcta

Los resultados obtenidos después de la ejecución de la subrutina suelen guardarse en los registros (especialmente en EAX) o bien en posiciones de memoria indicadas por éstos. Por esta razón, es conveniente al encontrarse con una subrutina "sospechosa" mirar los datos de entrada, viendo los valores colocados en la pila o las direcciones a las que apuntan estos valores y los datos de salida mirando el contenido de los registros o de las direcciones a que apuntan.

Un caso especial de subrutinas lo constituyen las APIs, de las que hablaremos en el Apéndice C.

call (Call Procedure)

Es la instrucción que efectúa el salto al punto de inicio de la subrutina. Además de esto, coloca en la pila la dirección de la instrucción siguiente, que será el punto de regreso después de ejecutarse la subrutina.

ret (Return from Procedure)

Complementa a la anterior y tiene la misión de regresar a la instrucción siguiente a la de llamada a la subrutina. Para ello, efectúa un salto a la dirección contenida en la pila, quedando ésta como estaba antes del call. Como hemos visto, puede ir acompañada de un parámetro para ajuste de la pila.

Veamos un ejemplo de como va variando el registro ESP según se ejecutan las instrucciones descritas:

```
...                                          ESP=63FB1C
:00401144 53                  push ebx       ESP=63FB18  (-4)
:00401145 6651                push cx        ESP=63FB16  (-2)
:00401147 6808304000          push 00403008  ESP=63FB12  (-4)
:0040114C E8B3000000          call 00401204  ESP=63FB0E  (-4)

      :00401204 40            inc eax
      ...                     ...            ESP=63FB0E
      :00401296 C20A00        ret 000A       ESP=63FB1C  (+A+4)

:00401151 Continúa la ejecución del programa...
```

o bien:

```
...                                          ESP=63FB1C
:00401144 53                  push ebx       ESP=63FB18  (-4)
:00401145 6651                push cx        ESP=63FB16  (-2)
:00401147 6808304000          push 00403008  ESP=63FB12  (-4)
:0040114C E8B3000000          call 00401204  ESP=63FB0E  (-4)
```

```
      :00401204 40               inc eax
      ...                        ...                ESP=63FB0E
      :00401296 C3               ret                ESP=63FB12  (+4)

:00401151 83C40A                 add esp, 0000000A  ESP=63FB1C  (+A)
:00401154 Continúa la ejecución del programa...
```

Como vemos, felizmente el resultado es el mismo; lo contrario causaría un crash del programa.

Hay algo que todos hemos visto muchas veces con emoción ;-)

```
      E89DFFFFFF             call 0040143C
      85C0                   test eax, eax
      7505                   jne 004014A8
```

Si, es el clásico call con la comprobación al regreso que decide si somos chicos buenos o chicos malos. Una forma de resolver la situación es la de substituir las primeras instrucciones del call por:

```
      33C0                   xor eax, eax
      C3                     ret
```

o bien:

```
      33C0                   xor eax, eax
      40                     inc eax
      C3                     ret
```

Con esto conseguimos que EAX, al regreso del call, tenga el valor cero o uno, según nos interese. Pero esto no siempre puede hacerse. Hay que asegurarse de que dentro del call no haya algún reajuste de la pila que nos lleve a la catástrofe. Para ello, debemos verificar el valor de ESP antes y después de la ejecución del call. Si son los mismos no hay problema, pero si son distintos podemos tratar de ajustar el valor de la pila y, según el resultado, buscar otro sistema. Hay que tener en cuenta que si hay alguna instrucción `push` antes del call, ésta también modifica el registro ESP y es posible que eliminándola se corrija el desajuste. Lo mismo vale para el clásico call que se quiere eliminar bien porque hace aparecer una nag, o bien por cualquier otra causa.

7.10. Instrucciones de bucle

loop (Loop According to ECX Counter)

Esta instrucción efectúa un bucle un número de veces determinado por el registro ECX. Equivale a las instrucciones `dec ecx / jnz`. Veamos un ejemplo doble:

```
:00401150 33C0                   xor eax, eax
:00401152 B90A000000             mov ecx, 0000000A
:00401157 40                     inc eax
:00401158 E2FD                   loop 00401157
```

Se ejecutaría diez veces el bucle, por lo que el valor final de EAX seria de 0000000A. También se ejecutaría diez veces el bucle equivalente:

```
:00401150 33C0                   xor eax, eax
:00401152 B90A000000             mov ecx, 0000000A
:00401157 40                     inc eax
:00401158 49                     dec ecx
:00401159 75FC                   jne 00401157
```

Existen variantes de esta instrucción que, además de depender del registro ECX, están condicionadas al estado del flag de cero (ZF):

loope / loopz Se efectúa el bucle si ECX es distinto de cero y ZF=1

loopne / loopnz Se efectúa el bucle si ECX es distinto de cero y ZF=0

7.11. Instrucciones de cadenas

En este apartado he agrupado instrucciones que podían haber estado igualmente en otros apartados, como de comparación, de entrada/salida, etc. pero me ha parecido más coherente poner todas las instrucciones de cadenas juntas.

rep (Repeat String Operation Prefix)

Esta instrucción se emplea en combinación con otras instrucciones de cadenas: MOVS, LODS, STOS, INS y OUTS. Tiene la finalidad de repetir la instrucción asociada una cantidad de veces determinada por los registros CX o ECX, según el programa trabaje con 16 ó 32 bits (En realidad el valor del registro se va decrementando en cada repetición, hasta llegar a cero).

Esta instrucción o, mejor dicho, las instrucciones asociadas, están relacionadas con los registros ESI y EDI, por lo que en cada repetición dichos registros se van incrementando o decrementando para apuntar siempre a la posición de memoria correspondiente.

repe (Repeat While Equal) / repz (Repeat While Zero)

Estas instrucciones se combinan con CMPS y SCAS. Son similares a `rep`, pero en este caso la repetición está condicionada a que el contenido del registro ECX sea distinto de cero y a que el flag de cero (ZF) sea igual a uno.

repne (Repeat While Not Equal) / repnz (Repeat While Not Zero)

Instrucciones iguales a las anteriores, con la diferencia de qué, para que se efectúe la repetición, el flag de cero debe ser igual a cero, en vez de igual a uno.

movs / movsb /movsw / movsd (Move Data from String to String)

Estas instrucciones tienen la misión de copiar el contenido de un espacio de memoria a partir de la dirección DS:ESI a otro espacio direccionado por ES:EDI. La longitud del espacio de memoria a copiar está indicada por el sufijo de la instrucción (byte / word / doubleword) y por el contenido del registro ECX.

```
* Possible StringData Ref from Data Obj ->"CaoS ReptantE"
                                |
      BE88304000              mov esi, 00403088
      BF49304000              mov edi, 00403049
      B908000000              mov ecx, 00000008
      F3                      repz
      A4                      movsb
```

En este ejemplo, se moverían los ocho primeros bytes correspondientes a las ocho primeras letras de la cadena de texto (CaoS Rep) al espacio de memoria situdo a partir de la posición 403049. Está claro que en este caso, tratándose de mover 8 bytes, hubiera sido lo mismo hacer EBX=2 y luego emplear la instrucción `movsd`. Esto aún se podría hacer de otra manera bastante rebuscada:

```
      BE8F304000              mov esi, 0040308F
      FD                      std
      BF49304000              mov edi, 00403049
      B908000000              mov ecx, 00000008
      F3                      repz
      A4                      movsb
```

En este caso no tiene lógica hacerlo así. Se trata sólo de un ejemplo de la utilización del flag de dirección.
Un último comentario relativo a esta instrucción: he tratado de ensamblar aquí la instrucción `repz` con el Masm y no me lo ha permitido, ha aceptado únicamente la instrucción `rep` (cosa lógica si miramos la definición de dicha instrucción), pero en cambio tanto el SoftIce como el W32Dasm la han desensamblado

como `repz`. También debo hacer notar que el Masm me ha obligado a poner las instrucciones `rep` y `movsb` en la misma línea. El SoftIce también las considera como una sola instrucción, a diferencia del W32Dasm que, como podemos ver, las separa.

lods / lodsb / lodsw lodsd (Load String)

Estas instrucciones se encargan de copiar un byte, word o doubleword, de la dirección indicada por DS:ESI al registro AL, AX o EAX, según el tamaño de la información a copiar indicado por el sufijo de la instrucción. Veremos un ejemplo al tratar de la instrucción `scas`.

stos / stosb / stosw / stosd (Store String)

Estas instrucciones son, en cierto modo, las inversas de las anteriores, ya que colocan el contenido de AL, AX o EAX, según el tamaño de la información a copiar indicado por el sufijo de la instrucción, en la dirección indicada por ES:EDI.

cmps / cmpsb / cmpsw / cmpsd (Compare String Operands)

Estas instrucciones comparan un byte, word o doubleword de la dirección indicada por DS:ESI con el valor del mismo tamaño situado en la dirección ES:EDI. Veamos un ejemplo real como la vida misma:

```
F3                      repz
A6                      cmpsb
```

En la dirección DS:ESI estaba el serial introducido, en la ES:EDI, el serial correcto y en ECX la longitud del serial introducido, que naturalmente era preciso que fuera igual que la del correcto ;-)

scas / scasb /scasw / scasd (Scan String)

Estas instrucciones comparan un byte, word o doubleword de la dirección indicada por ES:EDI con el valor del registro AL, AX o EAX, según el sufijo de la instrucción. Veamos un ejemplo:

```
AC                      lodsb
F2                      repnz
AE                      scasb
```

Primero colocaríamos en AL el contenido de la posición de memoria indicada por DS:ESI. A continuación compararíamos el contenido de ese registro con el byte situado en la dirección indicada por ES:EDI, repitiendose la operación con los bytes siguientes hasta encontrar un byte del mismo valor o hasta que el contenido del registro ECX fuera cero.

ins / insb / insw / insd (Input from Port to String)

Estas instrucciones se emplean para colocar en la dirección ES:EDI un byte, word o doubleword según el sufijo de la instrucción, obtenido de un port de entrada / salida cuyo número es el indicado por el valor del registro DX.

outs / outsb / outsw / outsd (Output String to Port)

Estas instrucciones funcionan al revés que las anteriores, es decir que colocan un byte, word o doubleword, según el sufijo de la instrucción, situado en la dirección DS:ESI, en un port de entrada / salida cuyo número es el indicado por el valor del registro DX.

7.12. Instrucciones de Entrada / Salida

in (Input from Port)

Copia el valor procedente del port de entrada / salida indicado por el segundo operando, en el primero. El segundo operando puede ser el registro DX o un valor inmediato del tamaño de un byte (sólo para valores

comprendidos entre 0 y 255). El primer operando serán los registros AL, AX o EAX, según el tamaño del valor a transferir.

out (Output to Port)

Copia el valor contenido en el segundo operando, en el port de entrada / salida indicado por el primer operando. El primer operando puede ser el registro DX o un valor inmediato del tamaño de un byte (sólo para valores comprendidos entre 0 y 255). El segundo operando serán los registros AL, AX o EAX, según el tamaño del valor a transferir.

Veamos una aplicación de estas dos instrucciones:

```
E461                    in al, 61
0C03                    or al, 03
E661                    out 61, al
```

Inicia el sonido del altavoz interno.

```
33C0                    xor eax, eax
E642                    out 42, al
40                      inc eax
3DFF000000              cmp eax, 000000FF
76F6                    jbe 0040114D
```

La ejecución de este código permite oir sólo un chasquido pero, si se ejecuta paso a paso con el SoftIce, se puede oir toda una escala de tonos :-)

```
E461                    in al, 61
24FC                    and al, FC
E661                    out 61, al
```

Finaliza el sonido del altavoz interno. Este código es importante, porque si no estuviera deberíamos apagar el ordenador para dejar de oir el ruido :-D

7.13. Instrucciones de rotación y desplazamiento

Estas instrucciones toman los bits de un valor indicado en el primer operando y los desplazan un número de posiciones determinado por el segundo operando, que puede ser un valor determinado, o un valor contenido en el registro CL. Estos desplazamientos pueden ser aritméticos o lógicos. Los desplazamientos aritméticos se efectúan con números sin signo y los espacios vacíos que se crean se rellenan con ceros. Los lógicos se efectúan con números con signo y el valor con que se rellenan los espacios que se crean es el adecuado para que el resultado sea el correcto.

sal (Shift Arithmetic Left) / shl (Shift Left)

Estas dos instrucciones son de desplazamiento aritmético a la izquierda y desplazamiento lógico a la izquierda respectivamente. Colocan el bit menos significativo de los que desaparecen, en el flag de acarreo y rellenan los espacios que se crean a la derecha con ceros. Son equivalentes e incluso tienen el mismo codigo de operación. En la práctica, lo que hacen es multiplicar el primer operando por dos, tantas veces como indique el segundo.

```
B84B28D17C              mov eax, 7CD1284B
D1E0                    shl eax, 1              EAX=F9A25096      CF=0

          7    C    D    1    2    8    4    B_
<- 1  0111 1100 1101 0001 0010 1000 0100 1011
       1111 1001 1010 0010 0101 0000 1001 0110
         F    9    A    2    5    0    9    6
```

$7CD1284Bx(2^1)=F9A25096$

```
B83B52E48A              mov eax, 8AE4523B
C1E008                  shl eax, 08             EAX=E4523B00      CF=0

          8    A    E    4    5    2    3    B_
      <- 8 1000 1010 1110 0100 0101 0010 0011 1011
                     1110 0100 0101 0010 0011 1011 0000 0000
                       E    4    5    2    3    B    0    0
```

8AE4523Bx(2^8)=E4523B00

sar (Shift Arithmetic Right)

Instrucción de desplazamiento aritmético a la derecha. El bit más significativo de los que desaparecen se coloca en el flag de acarreo y las posiciones que se crean a la izquierda se rellenan con el bit más significativo. En la práctica significa dividir con signo el primer operando por dos, tantas veces como indique el segundo, redondeando por defecto.

```
B83A52E48A              mov eax, 8AE4523A
C1F802                  sar eax, 02             EAX=E2B9148E   CF=1     SF=1

            8    A    E    4    5    2    3    A_
      2 ->  1000 1010 1110 0100 0101 0010 0011 1010
          1110 0010 1011 1001 0001 0100 1000 1110
            E    2    B    9    1    4    8    E
```

8AE4523A:(2^2)=E2B9148E -1964748230d:4=-491187058d

shr (Shift Right)

Instrucción de desplazamiento lógico a la derecha. El bit más significativo de los que desaparecen, se coloca en el flag de acarreo y las posiciones que se crean a la izquierda se rellenan con ceros. En la práctica significa dividir sin signo el primer operando por dos, tantas veces como indique el segundo, redondeando por defecto.

```
B83A52E48A              mov eax, 8AE4523A
C1E802                  shr eax, 02             EAX=22B9148E   CF=1

            8    A    E    4    5    2    3    A_
      2 ->  1000 1010 1110 0100 0101 0010 0011 1010
          0010 0010 1011 1001 0001 0100 1000 1110
            2    2    B    9    1    4    8    E
```

8AE4523A:(2^2)=22B9148E 2330219066:4=582554766

shld (Double Precision Shift Left)

Instrucción de desplazamiento lógico a la izquierda, parecida a `shl` pero con la diferencia de que los bits añadidos por la derecha no son ceros, sino que se toman del segundo operando, empezando por la izquierda, es decir por el bit más significativo. Vamos a ver un ejemplo:

```
B8CDAB3412              mov eax, 1234ABCD
BB78563512              mov ebx, 12355678
0FA4D808                shld eax, ebx, 08       EAX=34ABCD12

          1    2    3    4    A    B    C    D   1    2    3    5
      <- 8 0001 0010 0011 0100 1010 1011 1100 1101 0001 0010 0011 0101 ...
                     0011 0100 1010 1011 1100 1101 0001 0010
                       3    4    A    B    C    D    1    2
```

shrd (Double Precision Shift Right)

Instrucción de desplazamiento lógico a la derecha, parecida a `shr` pero con la diferencia de que los bits

añadidos por la izquierda no son ceros, sino que se toman del segundo operando, empezando por la derecha, es decir por el bit menos significativo. Veamos un ejemplo:

```
B8CDAB3412              mov eax, 1234ABCD
BB78563412              mov ebx, 12345678
0FACD808                shrd eax, ebx, 08       EAX=781234AB

        ...   7    8   1    2    3    4    A    B    C    D_
      8 ->  0111 1000 0001 0010 0011 0100 1010 1011 1100 1101
            0111 1000 0001 0010 0011 0100 1010 1011
              7    8    1    2    3    4    A    B
```

rol (Rotate Bits Left)

Instrucción de rotación a la izquierda. Los bits que desaparecen por la izquierda, se van colocando por la derecha y en el flag de acarreo se coloca el valor del último bit que ha rotado.

```
B83A52E48A              mov eax, 8AE4523A
C1C002                  rol eax, 02             EAX=2B9148EA   CF=0

                  8    A    E    4    5    2    3    A_
          <- 2   1000 1010 1110 0100 0101 0010 0011 1010
                 0010 1011 1001 0001 0100 1000 1110 1010
                   2    B    9    1    4    8    E    A
```

ror (Rotate Bits Right)

Instrucción de rotación a la derecha. Los bits que desaparecen por la derecha, se van colocando por la izquierda y en el flag de acarreo se coloca el valor del último bit que ha rotado.

```
B83A52E48A              mov eax, 8AE4523A
C1C802                  ror eax, 02             EAX=A2B9148E   CF=1

                  8    A    E    4    5    2    3    A_
          2 ->    1000 1010 1110 0100 0101 0010 0011 1010
                1010 0010 1011 1001 0001 0100 1000 1110
                  A    2    B    9    1    4    8    E
```

rcl (Rotate Bits Left with CF)

Instrucción de rotación a la izquierda pasando por el bit de acarreo. Los bits que desaparecen por la izquierda, se van colocando en el bit de acarreo y el valor anterior de éste, se coloca a la derecha.

Si CF=0:

```
B83A52E48A              mov eax, 8AE4523A                   CF=0
C1D002                  rcl eax, 02             EAX=2B9148E9   CF=0

                  8    A    E    4    5    2    3    A_
          <- 2   1000 1010 1110 0100 0101 0010 0011 1010      CF=0
                 0010 1011 1001 0001 0100 1000 1110 1001
                   2    B    9    1    4    8    E    9
```

Si CF=1:

```
B83A52E48A              mov eax, 8AE4523A                   CF=1
C1D002                  rcl eax, 02             EAX=2B9148EB   CF=0

                  8    A    E    4    5    2    3    A_
          <- 2   1000 1010 1110 0100 0101 0010 0011 1010      CF=0
                 0010 1011 1001 0001 0100 1000 1110 1011
                   2    B    9    1    4    8    E    B
```

rcr (Rotate Bits Right with CF)

Instrucción de rotación a la derecha pasando por el bit de acarreo. Los bits que desaparecen por la derecha, se van colocando en el bit de acarreo y el valor anterior de éste, se coloca a la izquierda.

Si CF=0:

```
B83A52E48A              mov eax, 8AE4523A                    CF=0
C1D802                  rcr eax, 02           EAX=22B9148E   CF=1

              8    A    E    4    5    2    3    A
      2 ->   1000 1010 1110 0100 0101 0010 0011 1010
           0010 0010 1011 1001 0001 0100 1000 1110
             2    2    B    9    1    4    8    E
```

Si CF=1:

```
B83A52E48A              mov eax, 8AE4523A                    CF=1
C1D802                  rcr eax, 02           EAX=62B9148E   CF=1

              8    A    E    4    5    2    3    A
      2 ->   1000 1010 1110 0100 0101 0010 0011 1010
           0110 0010 1011 1001 0001 0100 1000 1110
             6    2    B    9    1    4    8    E
```

7.14. Instrucciones de conversión

Se utilizan para duplicar la longitud de un número con signo. Para esto, rellenan el espacio añadido con el valor del bit más significativo del número original.

cbw (Convert Byte to Word)

Esta instrucción convierte el byte contenido en el registro AL en un word contenido en AX. En el primero de los ejemplos siguientes, el espacio ampliado se rellena con unos porque el bit más significativo de 80h es uno. En el segundo ejemplo, se rellena con ceros porque el bit más significativo de 7832h es cero.

```
B880E3DA12              mov eax, 12DAE380
6698                    cbw                  EAX=12DA**FF**80
B832782100              mov eax, 00217832
6698                    cbw                  EAX=0021**00**32
```

cwde (Convert Word to Doubleword)

Esta instrucción tiene una función similar a cbw, pero entre AX y EAX.

```
B880E3DA12              mov eax, 12DAE380
98                      cwde                 EAX=**FFFF**E380
B832782100              mov eax, 00217832
98                      cwde                 EAX=**0000**7832
```

cwd (Convert Word to Doubleword)

Esta instrucción añade al word contenido en el registro AX, el word contenido en DX, formando el doubleword AX:DX. El registro DX se rellena con unos o ceros, igual que en los casos anteriores, como se puede ver en los ejemplos siguientes:

```
B880E3DA82              mov eax, 82DAE380
BA11111111              mov edx, 11111111
6699                    cwd                  EAX=82DAE380  EDX=1111**FFFF**
B832782100              mov eax, 00217832
```

```
BA11111111              mov edx, 11111111
6699                    cwd                 EAX=00217832  EDX=1111 0000
```

cdq (Convert Double to Quad)

Esta instrucción tiene una función similar a cwd, pero entre EAX y EDX, formando el quadword EAX:EDX.

```
B880E3DA82              mov eax, 82DAE380
BA11111111              mov edx, 11111111
99                      cdq                 EAX=82DAE380  EDX=FFFFFFFF
B832782100              mov eax, 00217832
BA11111111              mov edx, 11111111
99                      cdq                 EAX=00217832  EDX=00000000
```

7.15. Instrucciones de flags

clc (Clear Carry Flag)

Pone el flag de acarreo (CF) a cero.

stc (Set Carry Flag)

Pone el flag de acarreo (CF) a uno.

cmc (Complement Carry Flag)

Complementa el flag de acarreo (CF). Es decir, que invierte su valor.

cld (Clear Direction Flag)

Pone el flag de dirección (DF) a cero.

std (Set Direction Flag)

Pone el flag de dirección (DF) a uno.

cli (Clear Interrupt Flag)

Pone el flag de interrupción (IF) a cero.

sti (Set Interrupt Flag)

Pone el flag de interrupción (IF) a uno.

lahf (Load Status Flags into AH Register)

Guarda en AX los flags de signo, cero, auxiliar, paridad y acarreo. Los bits 1, 3 y 5 tienen valores fijos, por lo que el registro AX queda formado por: `SF:ZF:0:AF:0:PF:1:CF`. En el siguiente ejemplo, si suponemos que: `SF=0`, `ZF=1`, `AF=0`, `PF=1` y `CF=0`:

```
9F              LAHF            EAX=xxxx46xx

                      SZ0A 0P1C
                      |||| ||||
AH valdrá 46h que es igual a:  0100 0110
```

sahf (Store AH into Flags)

Esta instrucción efectúa la operación inversa a la anterior. Es decir, que pone los flags a uno o a cero, de

acuerdo con el valor de AH.

7.16. Instrucciones de interrupción

int (Call to Interrupt Procedure)

Esta instrucción causa una interrupción de software determinada por su único operando y se desglosa en tres, según su código de operación:

```
CC              INT 3
```

La interrupción 3 hace que un debugger detenga la ejecución del programa. Es la instrucción que inserta el SoftIce en un programa cuando ponemos un breakpoint en un punto de la ejecución del mismo.

```
CD nn           INT nn
```

El número de la interrupción viene dado por un valor inmediato de un byte de longitud.

```
CE              INTO
```

Esta instrucción corresponde a la interrupción 4. Comprueba el valor del flag de desbordamiento (OF) y, si está activado, efectúa una llamada a dicha interrupción.

iretd / iret (Interrupt Return 32 / 16 bits)

Se produce el retorno de la rutina de interrupción y continúa la ejecución del programa.

7.17. Instrucciones del procesador

enter (Make Stack Frame for Procedure Parameters)

Esta instrucción crea un espacio en la pila requerido por algunos lenguajes de alto nivel. Está acompañada de dos parámetros, el primero indica la longitud del espacio reservado y el segundo el nivel de anidamiento léxico, o nivel de anidamiento de los procedimientos del programa. La longitud del espacio a reservar se resta de ESP, y el nuevo contenido de este registro se copia en EBP. Espero que un ejemplo aclare los efectos de esta instrucción:

```
...                                 ESP=63FB1C  EBP=63FB44
C8000404        enter 0400, 04      ESP=63F708  EBP=63FB18
```

La nueva dirección de la pila es el resultado de restarle a la anterior 404h, es decir, 4 bytes para guardar el contenido de EBP y 400h de espacio reservado. El nuevo contenido de EBP es la dirección en que está situado su anterior valor.

leave (High Level Procedure Exit)

Instrucción que complementa a la anterior. Restituye los valores de la pila anteriores a la última instrucción `enter`. Veamos la continuación del ejemplo anterior, suponiendo que posteriores instrucciones no hubiesen modificado los valores de EBP y ESP:

```
...                                 ESP=63F708  EBP=63FB18
C9              leave               ESP=63FB1C  EBP=63FB44
```

bound (Check Array Index Against Bounds)

Esta instrucción tiene dos parámetros y se encarga de determinar si el primer operando (un registro), que actúa como índice de una array, tiene un valor comprendido entre los limites establecidos de los valores de dicha array.

```
:0040114C 622B                    bound ebp, dword ptr [ebx]
```

En el ejemplo, EBX contiene el valor inferior establecido para la array (en este caso, 00000000) y EBX+4, el valor superior (en este caso, 000000FFh). Si el valor de EBP no estuviera comprendido entre ambos, se produciría una interrupción 5. En efecto, si le damos a EBP un valor superior a FFh el resultado es que nos aparece el siguiente aviso:

```
xxxx provocó una excepción 05H en el módulo xxxx.EXE de 0167:0040114c.
```

hlt (Halt)

Instrucción sin parámetros que detiene la ejecución del programa. Dicha ejecución se puede reanudar si ocurre una interrupción o mediante un reset del sistema.

wait (Wait)

Instrucción que suspende la ejecución del programa mientras espera que termine de efectuarse alguna operación de coma flotante.

8. ÍNDICE DE INSTRUCCIONES

jcc (salto condicional)
jmp
lahf
lea
leave
lods/b/w/d
loop
mov
movs/b/w/d
movsx
movzx
mul
neg
nop
not
or
out
outs/b/w/d
pop
popa
popad
popf
popfd
push
pusha
pushad
pushf
pushfd
rcl
rcr
rep
repe
repne
ret
rol
ror
sahf
sal
sar
sbb
scas/b/w/d
shl
shld
shr
shrd
stc
std
sti
stos/b/w/d
sub
test
wait
xaad
xchg
xor

APÉNDICE A - Instrucciones de coma flotante

A continuación veremos una descripción de las instrucciones de coma flotante que pueden resultar más interesantes de cara al objetivo de este manual. He decidido colocarlas separadas de las demás porque aunque al escribir un program pueden utilizarse indistintamente unas y otras, sus características y su entorno son distintos.

Como se ha explicado en el apartado 2.2 de este manual, estas instrucciones, así como los registros, flags etc. que forman su entorno, corresponden al coprocesador matemático 80X87 que hasta la aparición del procesador 80486 formaba una unidad aparte. Básicamente el entorno de coma flotante está formado por los 8 registros descritos en el apartado 2.2, los Tag Word, Control Word, Status Word y los punteros de instrucciones y operandos. Este entorno ocupa en total 94 bytes (16 bits) o 108 bytes (32 bits).

Las definiciones de las instrucciones están extraidas del documento *Floating Point Help* adjunto al MASM32. Se puede encontrar en la web de Steve Hutchesson o en la de Iczelion.

Empezaremos como antes por la reina de las instrucciones :-D

fnop (No Operation)

Aunque no requiere mayores explicaciones, veamos una comparación:

```
90              nop
D9D0            fnop
```

Aritméticas

fadd (Add)
faddp (Add and Pop)
fiadd (Integer Add)

Suman los dos operandos y colocan el resultado en el lugar del primero. Si no hay operandos, se suma el contenido de ST(0) y ST(1), se efectúa un pop del registro de coma flotante y el resultado de la suma se guarda en ST(0). Veamos un ejemplo:

```
C7030000C03F    mov dword ptr [ebx], 3FC00000
D903            fld dword ptr [ebx]             ST(0)=1,5
C7010000A03F    mov dword ptr [ecx], 3FA00000
D901            fld dword ptr [ecx]             ST(0)=1,25  ST(1)=1,5
DEC1            faddp st(1), st(0)              ST(0)=2,75
```

(El Masm ha compilado como `faddp st(1), st(0)` la instrucción `fadd` sin operandos)

fsub (Subtract)
fsubp (Subtract and Pop)
fisub (Integer Subtract)

Lo mismo que las anteriores, pero restando el segundo operando del primero. Pasemos directamente al ejemplo:

```
C7030000C03F    mov dword ptr [ebx], 3FC00000
D903            fld dword ptr [ebx]             ST(0)=1,5
C743040000A03F  mov [ebx+04], 3FA00000
D94304          fld dword ptr [ebx+04]          ST(0)=1,25  ST(1)=1,5
DEE9            fsubp st(1), st(0)              ST(0)=0,25
```

Ha efectuado la operación ST(1) - ST(0).

fsubr (Subtract with Reversed Operands)
fsubrp (Subtract with Reversed Operands and Pop)
fisubr (Integer Subtract with Reversed Operands)

Instrucciones de resta igual que las anteriores, pero invirtiendo los operandos. Veamos la diferencia:

```
C7030000C03F   mov dword ptr [ebx], 3FC00000
D903           fld dword ptr [ebx]             ST(0)=1,5
C743040000A03F mov [ebx+04], 3FA00000
D94304         fld dword ptr [ebx+04]          ST(0)=1,25  ST(1)=1,5
DEE1           fsubrp st(1), st(0)             ST(0)=-0,25
```

Ha efectuado la operación ST(0) - ST(1).

fmul (Multiply)
fmulp (Multiply and Pop)
fimul (Integer Multiply)

Instrucciones para multiplicar. Vamos de cabeza al ejemplo:

```
C7030000C03F   mov dword ptr [ebx], 3FC00000
D903           fld dword ptr [ebx]             ST(0)=1,5
C743040000A03F mov [ebx+04], 3FA00000
D94304         fld dword ptr [ebx+04]          ST(0)=1,25  ST(1)=1,5
DEC9           fmulp st(1), st(0)              ST(0)=1,875
```

Ha efectuado la operación ST(0) x ST(1).

fdiv (Divide)
fdivp (Divide and Pop)
fidiv (Integer Divide)

Instrucciones para dividir. Directamente al ejemplo:

```
C7030000C03F   mov dword ptr [ebx], 3FC00000
D903           fld dword ptr [ebx]             ST(0)=1,5
C743040000A03F mov [ebx+04], 3FA00000
D94304         fld dword ptr [ebx+04]          ST(0)=1,25  ST(1)=1,5
DEF9           fdivp st(1), st(0)              ST(0)=1,19999999...
```

Ha efectuado la operación ST(1) : ST(0).

fdivr (Divide with Reversed Operands)
fdivrp (Divide and Pop with Reversed Operands)
fidivr (Integer Divide with Reversed Operands)

Instrucciones para dividir, como las anteriores, pero invirtiendo los operandos.

```
C7030000C03F   mov dword ptr [ebx], 3FC00000
D903           fld dword ptr [ebx]             ST(0)=1,5
C743040000A03F mov [ebx+04], 3FA00000
D94304         fld dword ptr [ebx+04]          ST(0)=1,25  ST(1)=1,5
DEF1           fdivrp st(1), st(0)             ST(0)=0,83333333...
```

Ha efectuado la operación ST(0) : ST(1).

fabs (Calculate Absolute Value)

Convierte el número situado en ST(0) a su valor absoluto. Por ejemplo:

```
C7010000A0BF   mov dword ptr [ecx], BFA00000
D901           fld dword ptr [ecx]             ST(0)=-1,25
D9E1           fabs                            ST(0)=1,25
```

fchs (Change Sign)

Cambia el signo del número situado en ST(0). Por ejemplo:

```
C7010000A0BF   mov dword ptr [ecx], BFA00000
D901           fld dword ptr [ecx]             ST(0)=-1,25
D9E0           fchs                            ST(0)=1,25
```

frndint (Round to Integer)

Redondea la parte entera del número situado en ST(0). Por ejemplo:

```
C7030000C03F   mov dword ptr [ebx], 3FC00000
D903           fld dword ptr [ebx]             ST(0)=1,5
D9FC           frndint                         ST(0)=2
C743040000A03F mov [ebx+04], 3FA00000
D94304         fld dword ptr [ebx+04]          ST(0)=1,25
D9FC           frndint                         ST(0)=1
```

Con el método de redondeo por defecto (se puede cambiar) se puede ver como a partir de 0,5 redondea por exceso a la unidad superior.

fsqrt (Calculate Square Root)

Calcula la raiz cuadrada del número situado en ST(0). Un ejemplo muy conocido:

```
C70302000000   mov dword ptr [ebx], 00000002
DB03           fild dword ptr [ebx]            ST(0)=2
D9FA           fsqrt                           ST(0)=1,41421356...
```

fscale (Scale top of stack by power of 2)

Efectúa la operación $Y = Y \times 2^X$. El valor de Y lo toma de ST(0) y el de X de ST(1). El resultado lo coloca en ST(0). Vamos a ver un ejemplo:

```
C70302000000   mov dword ptr [ebx], 00000002
DB03           fild dword ptr [ebx]                ST(0)=2
C70303000000   mov dword ptr [ebx], 00000003
DB03           fild dword ptr [ebx]                ST(0)=3     ST(1)=2
D9FD           fscale                              ST(0)=12    ST(1)=2
```

En efecto: $Y = 3 \times 2^2 = 12$

fxtract (Extract Exponent and Mantissa)

Separa el exponente y la mantisa del número situado en ST(0), colocando el exponente en el lugar que ocupaba dicho número y colocando la mantisa en la pila, es decir en ST(0). Lógicamente al introducir un nuevo valor en la pila, el exponente que estaba en ST(0), pasa a ST(1).

```
C70332000000   mov dword ptr [ebx], 00000032
DB03           fild dword ptr [ebx]            ST(0)=50
D9F4           fxtract                         ST(0)=1,5625      ST(1)=5
```

Efectivamente, $50 = 1{,}5625 \times 2^5$

fprem (Calculate Partial Remainder)

fprem1 (Calculate Partial Remainder in IEEE Format)

Estas instrucciones calculan el resto de dividir el número situado en ST(0) por el situado en ST(1) y lo colocan el ST(0). Este resto tiene el mismo signo que el dividendo. La primera de las dos instrucciones hace una división normal, es decir, que obtiene un cociente que multiplicado por el divisor, da un número igual o inferior al dividendo. La segunda, en cambio, obtiene el cociente que, multiplicado por el divisor, dé el número inferior o superior al dividendo que resulte más cercano a éste. Por esta razón, el resultado de la primera instrucción será siempre positivo, mientras que el de la segunda puede ser negativo.

```
C70307000000   mov dword ptr [ebx], 00000007
DB03           fild dword ptr [ebx]            ST(0)=7
C7033C000000   mov dword ptr [ebx], 0000003C
DB03           fild dword ptr [ebx]            ST(0)=60    ST(1)=7
D9F8           fprem                           ST(0)=4     ST(1)=7
```

En efecto, 60 : 7 = 8 y el resto es 4.

```
C70307000000   mov dword ptr [ebx], 00000007
DB03           fild dword ptr [ebx]            ST(0)=7
C7033C000000   mov dword ptr [ebx], 0000003C
DB03           fild dword ptr [ebx]            ST(0)=60    ST(1)=7
D9F5           fprem1                          ST(0)=4     ST(1)=-3
```

Ahora se compara 7 x 8 = 56 con 7 x 9 = 63 y como este número está más cerca de 60, el resto será -3.

Trigonométricas y logarítmicas

fsin (Calculate Sine)

Calcula el seno del contenido de ST(0) expresado en radianes y coloca el resultado en la pila substituyendo al valor previo de ST(0). Veamos el seno de 180º:

```
D9EB           fldpi                       ST(0)=3,14159265...
D9FE           fsin                        ST(0)=0 (bueno, casi...)
```

fcos (Calculate Cosine)

Lo mismo pero para el coseno. Ejemplo:

```
D9EB           fldpi                       ST(0)=3,14159265...
D9FF           fcos                        ST(0)=-1
```

fsincos (Calculate Quick Sine and Cosine)

Calcula seno y coseno del valor existente en ST(0). El valor del seno se coloca en ST(0), en lugar del valor original. El valor del coseno se "empuja" a la pila quedando en ST(0), mientras el valor del seno pasa a ST(1).

```
D9EB           fldpi                       ST(0)=3,14159265...
D9FB           fsincos                     ST(0)=-1    ST(1)=0
```

fptan (Partial Tangent)
fpatan (Partial Arctangent)
f2xm1 (Calculate 2(^X)-1)
fyl2x (Y $\log_2(X)$)
fyl2xp1 (Y $\log_2(X+1)$)

Estas instrucciones no voy a comentarlas porque afortunadamente no son de uso frecuente en el tema que nos interesa ;-)

Transferencia de datos

fld (Push Real onto Stack)
fild (Convert Two's Complement Integer to Real and Push)
fbld (Convert BCD to Real and Push to Stack)

Estas instrucciones colocan un valor numérico en la pila de coma flotante. Las instrucciones `fld y fild` las hemos estado viendo en anteriores ejemplos.

fld... (Load Constant)

Esta serie de instrucciones coloca un valor determinado en la pila. Este valor es distinto según las variantes de la instrucción:

```
D9E8          fld1          ST(0)=1
D9EE          fldz          ST(0)=0
D9EB          fldpi         ST(0)=3,14159265...   el viejo y querido número pi
D9EA          fldl2e        ST(0)=1,44269504...   log en base 2 de e
D9E9          fldl2t        ST(0)=3,32192809...   log en base 2 de 10
D9EC          fldlg2        ST(0)=0,30102999...   log en base 10 de 2
D9ED          fldln2        ST(0)=0,69314718...   log neperiano de 2
```

fst (Store Floating-point Number from Stack)
fstp (Store Floating-point Number from Stack and Pop)
fist (Store Integer from Stack)
fistp (Store Integer from Stack and Pop)
fbstp (Store BCD to Integer and Pop Stack)

Estas instrucciones guardan el valor contenido en ST(0) en un lugar determinado de la memoria indicado por el operando. Las instrucciones terminadas en "p" efectúan un pop de la pila de coma flotante. Veamos un par de ejemplos:

```
C70300010000   mov dword ptr [ebx], 00000100
DB03           fild dword ptr [ebx]            ST(0)=256
D919           fstp dword ptr [ecx]            [ECX]=00 00 80 43
```

Como podríamos comprobar, 43800000h es la representación en coma flotante de 256d. Otro ejemplo:

```
C70300010000   mov dword ptr [ebx], 00000100
DB03           fild dword ptr [ebx]            ST(0)=256
DB19           fistp dword ptr [ecx]           [ECX]=00 01 00 00
```

Aquí es mas fácil ver que 000100h equivale a 256d.

fxch (Exchange Registers)

Intercambia los contenidos de ST(0) y ST(1).

```
D9E8          fld1                            ST(0)=1     ST(1)=?
D9EE          fldz                            ST(0)=0     ST(1)=1
D9C9          fxch st(0), st(1)               ST(0)=1     ST(1)=0
```

Control de ejecución

fcom (Compare)
fcomp (Compare and Pop)
fcompp (Compare and Pop and Pop)

ficom (Compare Integer)
ficomp (Compare Integer and Pop)

Instrucciones de comparación. Si se especifica un operando, se compara ST(0) con este operando. Si no se especifica ningún operando, se compara ST(0) con ST(1). En las instrucciones terminadas en "p" se efectúa un pop de la pila de coma flotante, excepto en el caso de fcompp en que se efectuan dos. El resultado de esta comparación se refleja en los valores del status word. Posteriormente mediante las instrucciones `fstsw` y `sahf`, se puede guardar dicho status word en un registro y efectuar o no un salto, según el valor de este registro. Espero que con unos ejemplos quede más claro:

```
C70364000000   mov dword ptr [ebx], 00000064
DB03           fild dword ptr [ebx]            ST(0)=100
C74304C8000000 mov [ebx+04], 000000C8
DA5304         ficom dword ptr [ebx+04]
9B             wait
DFE0           fstsw ax
9E             sahf
7200           jb 0040114C
```

Comparamos 100 (en ST(0)) con 200 (en [EBX+4]). Es menor y por lo tanto se efectúa el salto.

```
C70364000000   mov dword ptr [ebx], 00000064
DB03           fild dword ptr [ebx]            ST(0)=100
C7430464000000 mov [ebx+04], 00000064
DA5B04         ficomp dword ptr [ebx+04]
9B             wait
DFE0           fstsw ax
9E             sahf
7400           je 00401164
```

Comparamos 100 (en ST(0)) con 100 (en [EBX+4]). Son iguales y por lo tanto se efectúa el salto.

```
C70364000000   mov dword ptr [ebx], 00000064
DB03           fild dword ptr [ebx]            ST(0)=100
C74304C8000000 mov [ebx+04], 000000C8
DB4304         fild dword ptr [ebx+04]
DED9           fcompp
9B             wait
DFE0           fstsw ax
9E             sahf
7700           ja 0040117E
```

Comparamos 200 (en ST(0)) con 100 (en ST(1)). Es mayor y por lo tanto se efectúa el salto.

fucom (Unordered Compare)
fucomp (Unordered Compare and Pop)
fucompp (Unordered Compare and Pop and Pop)

Estas instrucciones son prácticamente iguales a las anteriores.

ftst (Test for Zero)

Compara ST(0) con 0. Como en las instrucciones anteriores, el resultado de esta comparación se refleja en los valores del status word. Veamos unos ejemplos:

```
C70301000000   mov dword ptr [ebx], 00000001
DB03           fild dword ptr [ebx]            ST(0)=1
D9E4           ftst
9B             wait
DFE0           fstsw ax
9E             sahf
7702           ja 00401146
```

Compara 1 con 0 y salta porque es mayor.

```
C70300000000   mov dword ptr [ebx], 00000000
DB03           fild dword ptr [ebx]            ST(0)=0
D9E4           ftst
9B             wait
DFE0           fstsw ax
9E             sahf
7400           je 00401156
```

Compara 0 con 0 y salta porque es igual.

```
C703FFFFFFFF   mov dword ptr [ebx], FFFFFFFF
DB03           fild dword ptr [ebx]            ST(0)=-1
D9E4           ftst
9B             wait
DFE0           fstsw ax
9E             sahf
7202           jb 00401168
```

Compara -1 con 0 y salta porque es menor.

fxam (Examine)

Coloca en los flags de condición del status word los valores correspondientes según el valor de ST(0).

Entorno

fstsw (Store Status Word -wait-)
fnstsw (Store Status Word -no wait-)

Estas instrucciones colocan el valor del status word en una posición de memoria (ocupa 16 bits) o en el registro AX. Hemos visto anteriormente un ejemplo del empleo de la instrucción `fstsw`.

fstcw (Store Control Word -wait-)
fnstcw (Store Control Word -no wait-)

Colocan el valor del control word en una posición de memoria (ocupa 16 bits) indicada por el operando.

fldcw (Load Control Word)

Coloca en el control word el valor de una posición de memoria (ocupa 16 bits) indicada por el operando.

fstenv (Store Environment State -wait-)
fnstenv (Store Environment State -no wait-)

Estas instrucciones colocan el valor del entorno de coma flotante (compuesto por: control word, status word, tag word, instruction pointer y operand pointer) en un rango de posiciones de memoria (de 14 ó 28 bytes, según se trabaje con 16 ó 32 bits) indicado por el operando.

fldenv (Load Environment State)

Coloca en el entorno de coma flotante el valor de un rango de posiciones de memoria (de 14 ó 28 bytes) indicado por el operando.

fsave (Save Coprocessor State -wait-)
fnsave (Save Coprocessor State -no wait-)

Estas instrucciones colocan el valor de los registros y el entorno de coma flotante en un rango de posiciones de memoria (de 94 ó 108 bytes, según se trabaje con 16 ó 32 bits) indicado por el operando.

frstor (Restore Coprocessor State)

Coloca en los registros y el entorno de coma flotante el valor de un rango de posiciones de memoria (de 94 ó 108 bytes) indicado por el operando.

Sistema

finit (Initialize Coprocessor State -wait-)
fninit (Initialize Coprocessor State -no wait-)

Inicializa el coprocesador y coloca en los registros y flags los valores por defecto.

fclex (Clear Exceptions State -wait-)
fnclex (Clear Exceptions State -no wait-)

Pone a cero los flags de excepción, el flag de ocupado y el bit 7 del status word.

fincstp (Increment Stack Pointer)
fdecstp (Decrement Stack Pointer)

Instrucciones que incrementan o decrementan el puntero en el status word

ffree (Free Register)

Curiosa instrucción que pone la etiqueta de "vacio" a un registro, pero sin cambiar su contenido.

fwait (Wait)

Se trata sólo de un nombre alternativo de la instrucción `wait`.

APÉNDICE B - Rutinas de interés

Longitud de un texto

Vamos a ver la rutina que emplea la función `lstrlenA` perteneciente a la librería `Kernel32.dll` para averiguar la longitud de una cadena de texto. En este caso, vamos a suponer que se trata de un serial que hemos introducido y que es de nueve caracteres.

```
:BFF7117E F2                repnz
:BFF7117F AE                scasb
```

Estas instrucciones (o instrucción porque el SoftIce las desensambla como una), buscan a partir de la dirección indicada por ES:EDI (la dirección de nuestro serial) el contenido de AL (00) y va repitiendo la operación hasta encontrarlo al final de nuestro serial. En cada una de estas operaciones de búsqueda se resta 1 del contenido del registro ECX cuyo valor inicial es FFFFFFFF. Como en este caso el serial introducido es de 9 caracteres, el contenido final de este registro es FFFFFFF5.

```
:BFF71180 83C8FE            or eax, FFFFFFFE
```

Como EAX valía 0, pasa a valer FFFFFFFE.

```
:BFF71183 2BC1              sub eax, ecx
```

FFFFFFFE-FFFFFFF5=9, es decir la longitud de nuestro serial.

Así pues, al regreso al programa, el registro EAX contiene la longitud de la cadena de texto cuya longitud se quería conocer.

Un poco complicado el sistema ¿no?

Movimiento de textos

Vamos a ver un fragmento de la rutina perteneciente al programa krnl386.exe que suele emplearse para mover cadenas de texto. Supondremos también que se trata de un serial de nueve caracteres:

```
:9F1A 6651                  push ecx
```

ECX contiene la longitud del serial que hemos introducido (9). Con esta instrucción, se guarda en la pila.

```
:9F1C 66C1E902              shr ecx, 02
```

Esta instrucción desplaza los bits de ECX dos lugares a la derecha. Como ECX contiene 9 (00001001), el resultado es 2 (00000010). Fijáos que el resultado equivale a dividir ECX por 4.

```
:9F20 F3                    repz
:9F21 6766A5                movsd
```

Se copian cuatro caracteres de nuestro serial desde su posición actual indicada por DS:ESI hasta una nueva ubicación señalada por ES:EDI. Esto se repite las veces que indica el registro ECX (En nuestro caso 2, lo que significa que se habrán copiado los ocho primeros caracteres).

```
:9F24 6659                  pop ecx
```

Se recupera de la pila el valor que contenía el registro ECX (9).

```
:9F26 6683E103              and ecx, 00000003
```

El resultado de esta operación es 1 (00001001 and 00000011 = 00000001)

```
:9F2A F3                    repz
:9F2B 67A4                  movsb
```

Esta instrucción es igual que la anterior, pero aquí se copia sólo un carácter (aquí la "b" final significa "byte" y la "d" de la instrucción anterior, "doubleword" o sea, cuatro bytes).

O sea, que se copian los caracteres en grupos de cuatro, y cuando quedan menos de cuatro caracteres para copiar, se hace de uno en uno.

Comprobación de caracteres mediante una máscara

Veamos a continuación una subrutina, extraida de un programa comercial, que permite comprobar la existencia de determinados caracteres en una cadena de texto.

El interés de esta rutina está en que nos permite "filtrar" los caracteres de esa cadena de texto para eliminar los que no nos interesen o no admitir un texto que contenga dichos caracteres.

```
:00445608 41                inc ecx
:00445609 8A06              mov al, byte ptr [esi]
:0044560B 0AC0              or al, al
:0044560D 7407              je 00445616
:0044560F 46                inc esi
:00445610 0FA30424          bt dword ptr [esp], eax
:00445614 72F2              jb 00445608
:00445616 ...
```

La instrucción `bt dword ptr [esp], eax` testea el bit señalado por EAX, a partir de la dirección indicada

por ESP.

Esto es lo que hay a partir de esa dirección:

```
00 00 00 00 00 00 FC 03 FE 7D FF 07 00 00 ...
```

Como se trata de bits, vamos a representarlo en binario a partir de los cuarenta y ocho ceros iniciales:

```
1111 1100 0000 0011 1111 1110 0111 1101 1111 1111 0000 0111 ...
  F    C    0    3    F    E    7    D    F    F    0    7
```

Cuando el primer operando de la instrucción `bt` es una dirección de memoria, la instrucción va tomando grupos de cuatro bytes invertidos, como cuando los carga en un registro, por lo que el orden real es este:

```
0011 1111 1100 0000 0111 1111 1011 1110 1111 1111 1110 0000 ...
```

Como hemos visto que hay 48 ceros por delante, que en este caso representan los caracteres del 0 al 47, el primer cero de esta serie corresponderá al número 48 (30h) es decir, el código ASCII de "`0`"; el segundo al código de "`1`" y así sucesivamente. De este modo podemos establecer la relación entre los valores de esta serie de bits y los códigos ASCII correspondientes:

```
0011 1111 1100 0000 0111 1111 1011 1110 1111 1111 1110 0000 ...
|||| |||| |||| |||| |||| |||| |||| |||| |||| |||| |||| ||||
0123 4567 89:; <=>? @ABC DEFG HIJK LMNO PQRS TUVW XYZ[ \]^_
```

Volviendo a la instrucción `bt`, ésta nos ira colocando un uno en el flag de acarreo si el carácter examinado es un número o una letra mayúscula exceptuando "`0`", "`1`", "`O`" e "`I`".

O sea, que el programador que incluyó esta rutina en uno de sus programas, no pone en sus seriales ninguno de estos caracteres, seguramente harto de que le digan que "el serial no me funciona".

Esto en sí no tiene nada de particular, ya que he encontrado algún caso parecido en el que el programador no ponía estos caracteres ni tampoco "5" y "S" que también se prestan a confusión. Sin embargo sí me ha parecido interesante la forma en que lo hace.

Comprobación de caracteres mediante una tabla

Hay un sistema muy utilizado por los programadores para determinar a que tipo pertenece un carácter determinado. Se basa en una tabla con la que nos vamos a encontrar muy a menudo.

La reconoceremos inmediatamente por la forma en que aparece en el lado derecho de la ventana de datos del SoftIce:

```
 . . . . . . . .
 .(.(.(.(.(. . .
 . . . . . . . .
 . . . . . . . .
H...............
etc.
```

¿Para qué sirve esta tabla? Es una tabla mágica :) Representa los codigos ASCII de los 128 primeros caracteres, separados por pares de ceros. Vamos a ver los códigos que la componen:

```
20 00 20 00 20 00 20 00 20 00 20 00 20 00 20 00  | Códigos del 0 al 8
20 00                                            | (caracteres no gráficos)

      28 00 28 00 28 00 28 00 28 00              | Códigos del 9 al 13
                                                 |  (caracteres especiales)

                                  20 00 20 00    |
20 00 20 00 20 00 20 00 20 00 20 00 20 00 20 00  | Códigos del 14 al 31
20 00 20 00 20 00 20 00 20 00 20 00 20 00 20 00  | (caracteres no gráficos)
```

```
48 00                                                       | Código del 32 (espacio)

      10 00 10 00 10 00 10 00 10 00 10 00 10 00             | Códigos del 33 al 47
10 00 10 00 10 00 10 00 10 00 10 00 10 00                   | (caracteres varios)

84 00 84 00 84 00 84 00 84 00 84 00 84 00                   | Códigos del 48 al 57
84 00 84 00                                                 | (números del 0 al 9)

            10 00 10 00 10 00 10 00 10 00 10 00             | Códigos del 58 al 64
10 00                                                       | (caracteres varios)

      81 00 81 00 81 00 81 00 81 00 81 00                   | Códigos del 65 al 70
                                                            | (letras de la "A" a la "F")
                                                  01 00     |
01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00             | Códigos del 71 al 90
01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00             | (letras de la "G" a la "Z")
01 00 01 00 01 00                                           |

                  10 00 10 00 10 00 10 00 10 00             | Códigos del 91 al 96
10 00                                                       | (caracteres varios)

      82 00 82 00 82 00 82 00 82 00 82 00                   | Códigos del 97 al 102
                                                            | (letras de la "a" a la "f")
                                                  02 00     |
02 00 02 00 02 00 02 00 02 00 02 00 02 00 02 00             | Códigos del 103 al 122
02 00 02 00 02 00 02 00 02 00 02 00 02 00 02 00             | (letras de la "g" a la "z")
02 00 02 00 02 00                                           |

                  10 00 10 00 10 00 10 00 10 00             | Códigos del 123 al 126
                                                            | (caracteres varios)

                                                  20 00     | Código 127
```

Como podemos ver, los valores de la tabla están relacionados con los caracteres que representan. Vamos a hacer una tabla de significados:

	8 4 2 1	8 4 2 1	
	| | | |	| | | |	
Caracteres no gráficos	0 0 1 0	0 0 0 0	(20)
Caracteres especiales	0 0 1 0	1 0 0 0	(28)
Caracteres varios	0 0 0 1	0 0 0 0	(10)
Números	1 0 0 0	0 1 0 0	(84)
Letras (A-F)	1 0 0 0	0 0 0 1	(81)
Letras (G-Z)	0 0 0 0	0 0 0 1	(01)
Letras (a-f)	1 0 0 0	0 0 1 0	(82)
Letras (g-z)	0 0 0 0	0 0 1 0	(02)
Espacio	0 1 0 0	1 0 0 0	(48)

Así pues, el programa puede determinar a cual de estos grupos pertenece un carácter, mediante una instrucción del tipo:

```
mov al, byte ptr [ecx+2*eax]
```

En este caso ECX apunta al inicio de la tabla y EAX contiene el código ASCII del carácter a examinar. Según el valor que nos devuelva la tabla en AL y utilizando la instrucción `and eax, nn` adecuada, se puede comprobar si el carácter es por ejemplo:

letra mayúscula	`and eax, 01`
letra minúscula	`and eax, 02`
letra en general	`and eax, 03`
letra o número	`and eax, 07`

número decimal	`and eax, 04`
número hexadecimal	`and eax, 80 o o and eax, 84` (más utilizado)
espacio	`and eax, 40`
otro caracter	`and eax, 10`

O cualquier otra combinación basada en los números de la tabla.

APÉNDICE C - MASM32

Como he dicho al principio, este manual no pretende enseñar a programar. De todas maneras, sí es conveniente ser capaz de programar pequeñas aplicaciones o insertar código en un programa.

Para insertar código en un programa, utilizo el SoftIce que permite ver la ejecución de ese código sobre la marcha. Luego con un editor hexadecimal se puede introducir el código de manera permanente.

Para programar pequeñas aplicaciones en assembler utilizo el MASM32.

De esto tiene la culpa ;-) Mr. Crimson (del que me considero deudor) ya que la lectura de sus tutoriales me convenció para ello (se pueden encontrar en la compilación 4.0 del Pr@fesor X).

He aquí un fragmento de uno de estos tutoriales en que hace una descripción del programa y de una de sus principales características:

El MASM32 de Hutchesson ha sido mi opción. Consiste en un sencillo editor desde el que se puede compilar y ejecutar el código. Incorpora una librería bastante completa y los 'includes' correspondientes. Introduce también algunas novedades en la sintaxis del código. La más curiosa, quizás, es la sentencia INVOKE.

En ASM convencional, para llamar a una rutina de 3 argumentos deberíamos hacer:

```
PUSH Arg3
PUSH Arg2
PUSH Arg1
CALL [RUTINA]
```

con INVOKE quedaría:

```
invoke [RUTINA],Arg1,Arg2,Arg3
```

A los puristas no acaba de convencerles esta 'contaminación' del ASM con sintaxis de alto nivel pero el resultado es que no hay diferencias apreciables en relación a tamaño de ejecutables y velocidad. El código fuente queda bastante más despejado y fácil de entender. En los ejemplos incluidos en este tutorial emplearé esta distribución por lo que si quieres aplicar los contenidos con TASM32 deberás adaptar los programas que se incluyen. Personalmente, te recomiendo el uso de Masm32.

Así pues, para colocar una MessageBox en la pantalla utilizaríamos la sentencia:

```
invoke MessageBox,hDlg,ADDR texto,ADDR titulo,MB_ICONSTOP
```

Que quedaría compilada así:

```
6A10            push 00000010           (Icono de stop y botón de aceptar)
68A2304000      push 004030A2           (Dirección del título en la barra superior)
68AA304000      push 004030AA           (Dirección del texto del interior)
FF7508          push [ebp+08]           (Handle de la ventana)
E8AA000000      Call USER32.MessageBoxA
```

Otra característica es la existencia de macros que, por ejemplo, nos permiten emplear instrucciones del tipo `if...then...else`:

```
.if ebx < 80h
      mov al, byte ptr [edx+ebx]
.else
```

```
        xor eax, eax
    .endif
```

Que al compilar quedaría así:

```
:004011A3 81FB80000000      cmp ebx, 00000080
:004011A9 7305              jnb 004011B0
:004011AB 8A0413            mov al, byte ptr [ebx+edx]
:004011AE EB02              jmp 004011B2
:004011B0 33C0              xor eax, eax
:004011B2 ...
```

También se pueden usar macros que si bien facilitan el trabajo al programar, en ocasiones pueden generar un código no muy elegante. Veamos un ejemplo de la utilización de `for` :

```
for codigo, <49,52,48,51>
      mov eax, 50
      mov edx, codigo
      mul edx
      add ebx, eax
      inc ebp
endm
```

Que una vez ensamblado quedaría así:

```
B832000000      mov eax, 00000032
BA31000000      mov edx, 00000031
F7E2            mul edx
03D8            add ebx, eax
45              inc ebp
B832000000      mov eax, 00000032
BA34000000      mov edx, 00000034
F7E2            mul edx
03D8            add ebx, eax
45              inc ebp
B832000000      mov eax, 00000032
BA30000000      mov edx, 00000030
F7E2            mul edx
03D8            add ebx, eax
45              inc ebp
B832000000      mov eax, 00000032
BA33000000      mov edx, 00000033
F7E2            mul edx
03D8            add ebx, eax
45              inc ebp
```

Como podemos ver, se ha repetido el código cuatro veces. Para una aplicación de cierta complejidad se podría buscar una solución más limpia utilizando una subrutina, pero si se trata de un pequeño programa, esto no tiene ninguna importancia.

Para terminar esta breve descripción de las características del MASM32, diré que, como quizá habréis observado en los ejemplos anteriores, el programa trabaja por defecto con números decimales, debiéndose añadir el sufijo "h" si se trata de números hexadecimales. En este último caso, si el número empieza por una letra, se debe poner un cero delante para que el programa sepa que se trata de un número.

Como he dicho antes, se puede obtener este programa en la web de Steve Hutchesson o en la de Iczelion. En el paquete del programa está incluida una colección de tutoriales de Iczelion que tratan exhaustivamente de como programar con él.

APÉNDICE D - APIs

Las APIs (Application Programming Interface) son rutinas "prefabricadas" destinadas a facilitar la labor del

programador y al mismo tiempo a reducir la extensión de los programas, porque no hace falta incluir estas rutinas en el ejecutable, ya que están incluidas en librerías de Window$ como `user32.dll`, `kernel32.dll`, `gdi32.dll`, etc.

El procedimiento para acceder a estas rutinas consiste en introducir los parámetros necesarios utilizando instrucciones `push` y llamar a la ejecución de la rutina mediante una instrucción `call`. Después de la ejecución, la rutina suele devolver un valor en el registro EAX que indica si ésta se ha ejecutado con éxito y en algún caso, un valor con información complementaria.

Así pues, para crear una ventana en la que aparezca un texto y unos botones de Aceptar o Cancelar, no hace falta diseñarla a partir de cero, sino que es suficiente con llamar a MessageBoxA y proporcionarle los parámetros correspondientes como hemos visto en uno de los ejemplos de utilización del MASM32. En este caso concreto, el valor devuelto por la función, si ésta se ha ejecutado con éxito, nos permite saber que botón se ha pulsado.

Veamos ahora un ejemplo de la utilización de GetLocalTime:

```
680D314000       push 0040310D
E826010000       Call KERNEL32.GetLocalTime
```

La función GetLocalTime nos devuelve en la dirección 40310D, 16 bytes con la siguiente información:

```
A  A  M  M  ds  ds  D  D  h  h  m  m  s  s  ms  ms
```

Donde el año está en formato de cuatro cifras, `ds` es el día de la semana empezando por el domingo (0) y `ms` son milisegundos. O sea que el valor:

```
D2 07 0A 00 05 00 0B 00 10 00 28 00 33 00 5E 01
```

Significa:

```
año  = 2002    (07D2)
mes  = octubre (000A)
ds   = viernes (0005)
día  = 11      (000B)
hora = 16      (0010)
min. = 40      (0028)
seg. = 51      (0033)
mseg.= 350     (015E)
```

¡Se acabó mirar el reloj o el calendario! :-) Esta función es una de las que no devuelve ningún valor significativo en EAX.

Existe en la Red abundante información sobre este tema:

En inglés se pueden encontrar por ejemplo: +Sync's Win32 Reference 1.0 o Win32 Programmer's Reference (No pongo links porque suelen variar y por otra parte, son fáciles de encontrar).

En español hay que destacar las cRACKER's n0TES de TORN@DO traducidas al español por TXELI (incluidas en la compilación 4.0 del Profesor X), así como el trabajo de traducción de API's desarrollado por MNEMOX que podéis visitar en APIS-CORNER.