Aprenda ensamblador 80x86 en dos patadas*

http://www.rinconsolidario.org/eps/asm8086/asm.html

INTRODUCCIÓN

Este documento de tan optimista título pretende ser una breve introducción al ensamblador; la idea es simplemente coger un poco de práctica con el juego de instrucciones de la familia 80x86, direccionamiento, etc.. sin entrar demasiado en la estructura de los ejecutables, modelos de memoria...

¿Por qué otro manual de ensamblador? En un principio estaba dirigido a quien ya supiera ensamblador de motorola 68000 (extendido en algunas universidades), siendo una especie de mini-guía para que los interesados (pocos, dicho sea), sabiendo 68000, pudieran saltarse a los Intel sin mucho esfuerzo. Lo que empezó con un par de apuntes sobre registros, segmentación y equivalencias entre instrucciones de uno y otro procesador acabó por desmadrarse hasta formar un modesto manual de cerca de una docena de apartados.

Perdida la motivación inicial, ha pasado a ser un manual más, ciertamente, pero sigo dedicándole tiempo por varios motivos. Para empezar, son pocos los textos que tocan todos los temas, y la inmensa mayoría tratan exclusivamente de MSDOS. Ya hay algunos, afortunadamente, que incluyen programación para Linux (y bastante buenos para mi gusto), pero los mejores suelen estar en inglés y ser demasiado técnicos, además de no ser habitual que hablen de instrucciones MMX, SSE o incluso las del coprocesador matemático (cosa que no comprendo, porque es donde se le saca juguillo). El objetivo principal que me he impuesto es la legibilidad, para que el que se encuentre con esto entienda rápidamente los aspectos que se tratan, y luego vaya a otro sitio sabiendo qué información buscar, y qué hacer con ella. Tampoco intento ser exhaustivo con esto; rondan por la red manuales tipo "Art of Assembly" que tratan intensamente temas como las estructuras de datos o la optimización, que no osaría emular. Procuro que sea esto lo más ameno posible, y nadar en detalles desde el primer momento no ayuda demasiado. Para eso ya está la sección de enlaces que incluyo.

Tal y como he dejado el manual, creo que se puede comprender en su totalidad sin ningún conocimiento previo, salvo algunas nociones básicas de programación. Si nunca antes has programado, te recomiendo que recurras a otras fuentes.

¿Qué vamos a hacer aquí? Aprenderemos los rudimentos de programación en MS-DOS, a fin de entendernos con el modo real, (16 bits, o sea, segmentos de 64k y gracias), y bajo Linux (como ejemplo de programación en 32 bits); también se aprenderá a usar el coprocesador matemático, porque hacer operaciones en coma flotante a pelo es menos saludable que fumar papel de amianto. Se añadirán instrucciones MMX, SSE (espero), SSE2, y si alguien dona algún AMD modernito a la Fundación Ernesto Pérez, 3DNow y familia. A lo largo de las explicaciones se irán comentando las diferencias entre los dos modos de funcionamiento para, ya hacia el final, introducir la interacción entre programas en C y ensamblador en Linux y, más concretamente, cómo hacer código en ensamblador lo más portable posible para las implementaciones x86 de sistemas Unix. El resto de capítulos explicarán algunos aspectos que quedan oscuros como la segmentación en modo protegido, el proceso de arranque del ordenador, la entrada/salida...

Intentaré contar más medias verdades que mentiras cuando se complique demasiado el asunto, a fin de no hacer esto demasiado pesado (y así de paso me cubro un poco las espaldas cuando no tenga ni idea del tema, que también pasa) Doy por supuesto que sabes sumar numeros binarios, en hexadecimal, qué es el complemento a 2, etc etc. Si no tienes ni idea de lo que te estoy hablando, échale un ojo al capítulo 0.

La notación que usaré será una "b" al final de un número para binario, una "h" para hexadecimal, y nada para decimal, que es lo que usa este ensamblador. Existen como notación alternativa igualmente admitida por NASM cosas del tipo 0x21 en vez de 21h, que es lo que se hace en C. Yo usaré la que llevo usando toda la vida, que para qué cambiar pudiendo no hacerlo. Por cierto, los números hexadecimales que comienzan por una letra requieren un 0 delante, a fin de no confundirlos el ensamblador con etiquetas o instrucciones. Si se me escapa por ahí algún 0 sin venir a cuento, es la fuerza de la costumbre. Y aunque estrictamente por una "palabra" se entiende el ancho típico del dato del micro (por lo que en un 386 serían 32 bits) me referiré a palabra por 2 bytes, o 16 bits; doble palabra o dword 32; palabra cuádruple o qword 64. Esta terminología está bastante extendida, y la razón de ser de esto (según creo) es que la "palabra-palabra" era inicialmente de 16 bits (8086). Cuando llegó el 386 con sus 32 bits, realmente todo el software seguía siendo de 16 bits, salvando los "extras" que proporcionaba el micro; así pues, lo del 386 eran más bien palabras dobles. Y así ha sido hasta ahora.

¿Qué te hace falta?

1. Un ordenador (si estás leyendo esto no creo que sea un problema) compatible con la familia 80x86 (¿alguien usa mac?)
2. En principio MS-DOS o Windows 95,98,Me.. para ejecutar los programas en modo real (o algún tipo de emulador de MSDOS si usas otro sistema). Mientras te ejecute programas DOS, perfecto. Lógicamente, para los ejemplos de Linux, necesitarás Linux (aunque si alguien un poco avispado se salta hasta -cuando lo suba- el capítulo referido a C, verá que se puede hacer más o menos lo mismo para sistemas *BSD).
3. Un ensamblador. Aquí usaré NASM para los ejemplos (que en el momento de escribir estas líneas va por la versión 0.98.36), aunque realmente apenas habrá de eso, pues esto es más bien una guía descriptiva. Las instrucciones serán las mismas en todo caso, pero si empleas otro ensamblador habrá, lógicamente, diferencias en la sintaxis. NASM, MASM (Microsoft) y TASM (Borland) usan la llamada sintaxis Intel (pues es la que usa Intel en su documentación), mientras que otros como GAS (GNU-ASM) emplean la AT&T. Además, dentro de cada tipo, existen numerosas diferencias en las directivas de un ensamblador a otro. Toca en ese caso leerse el manual del programa en cuestión. Algunos temas son más conceptuales que otra cosa (de hecho cuando hable de cosas que vengan en hojitas de referencia, lo más probable es que me salte el rollo y te remita a ellas), por lo que si te interesan, te serán de utilidad sea cual sea tu ensamblador. Sin embargo, en términos generales, este mini-manual está hecho para usar NASM.

   Por si alguien está pensando en para qué leches introducir la programación en MSDOS en estos tiempos tan modernos, debería saber que el modo real aún tiene relevancia pues el ordenador cuando arranca lo hace en este modo de funcionamiento. Ayuda además a entender las peculiaridades de los PCs, siendo la compatibilidad con esta característica un fuerte condicionante en la arquitectura de la familia. El planteamiento paralelo 16-32 bits quizá sea un poco confuso (me estoy pensando limitarme a los 32 bits, y dejar el modo real para el final, por si alguno quiere enredar con él), pero creo que es útil pues aún hay gente aprendiendo ensamblador en MSDOS (entre otras cosas porque puedes manipular todo el hardware sin restricciones, lo que resulta ideal para "jugar con el ordenador"). Agradecería mucho que la gente que pasara por aquí opinara sobre el tema.

A propósito de lo cual tengo que añadir algunas puntualizaciones. Me llegan de vez en cuando algunos emails (muy de vez en cuando) comentando cosas sobre la página, lo que me parece perfecto. Lo que ya no me lo parece tanto es que gente con no sé si ingenuidad o jeta tipo hormigón escriba pidiendo que les resuelva la vida. Comprendo que alguien pueda enviar una duda, pero me desconcierta que haya quien solicita que le hagan un programa para nosequé práctica de clase. Estos últimos que, por favor, ni se molesten.

Sobre los de las dudas existenciales esporádicas, sugiero otra estrategia. Puse una dirección de correo ahí en la portada para que quien tuviera alguna duda o comentario sobre la web, la expusiera. Es decir, me sirve alguien que me diga "oye, he leido tal y cual y no se entiende bien". O "mira, es que este tema me parece muy interesante y no dices nada al respecto". Y entonces voy yo, me lo pienso, y me busco un ratillo para añadir o corregir lo que haga falta. Lo que difícilmente haré será contestar directamente a las dudas (salvo que sean muy breves y educadas, y no estén claramente explicadas en la página o en los enlaces). Y no es que me lo tome a malas, sino que cuando uno responde persona por persona siente que pierde el tiempo si se puede subir la respuesta a la web, y más aún si ya está subida.

Por cierto, tampoco soy ningún gurú del ensamblador, sólo un aficionado que no dedica mucho tiempo a ver la tele. Dicho queda.

¿Listo todo? Abróchense los cinturones, damas y caballeros, que despegamos.

PROLEGÓMENOS: PARA LOS QUE NO HAYAN VISTO ENSAMBLADOR EN SU VIDA

En este capítulo introductorio veremos algunos conceptos básicos sobre el funcionamiento de un microprocesador. Muchos de ellos serán ya conocidos, pero es imprescindible tenerlos todos bien claros antes de empezar a programar en ensamblador. Por otra parte, este capítulo puede ser de utilidad a cualquier principiante, sea cual sea el microprocesador que vaya a manejar, pues son ideas generales que luego, llegado el caso, particularizaremos para los 80x86. Por mi (escasa) experiencia con gente que empieza con el ensamblador, puedo decir que muchos errores vienen derivados de no comprender completamente algunas de las ideas que intento presentar en este capítulo.

Es bastante recomendable tener algún conocimiento previo de progración en cualquier otro lenguaje, llámese C, pascal, ADA, basic o lo que sea. Hay gente que opina lo contrario, pero a mí me parece mucho más didáctico y fácil aprender primero un lenguaje de alto nivel antes que ensamblador..

- Sistemas de numeración

Como ya sabrá la mayoría, un bit es un dígito binario (abreviatura de BInary digiT), esto es, un número que puede ser 0 o 1. Los bits se pueden agrupar para formar números mayores; con N bits se pueden formar hasta 2^N números distintos (^ por "elevado a"). Por ejemplo, con cuatro bits 2^4=16 combinaciones:

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

Los bits constituyen la base numérica de la electrónica digital, que básicamente está fundamentada en interruptores electrónicos que abren o cierran el paso de corriente (podríamos decir que abierto es un 0 y cerrado es un 1, o viceversa) de acuerdo con ciertas reglas. En un sistema complejo como un ordenador se combinan elementos sencillos para crear estructuras con gran potencia de cálculo, pero todas las operaciones se realizan con ceros y unos, codificadas como tensiones. Así, podríamos tener un circuito digital en el que un punto a 5 voltios significaría un '1', y uno a 0 voltios, un '0'.

Un grupo de 8 bits se denomina byte. 1024 bytes forman un kilobyte (kbyte, kb, o simplemente k), 1024 kilobytes un megabyte, 1024 megabytes un gigabyte. Existen más submúltiplos como terabyte o petabyte, pero son de menor uso.

El hermano menor del byte es el nibble, que no es más que una agrupación de 4 bits. El nombre viene de un bobo juego de palabras en el que byte, octeto, se pronuncia igual que bite, mordisco, que es lo que significa nibble.

Sabemos que los bits existen pero.. ¿cómo representar información con ellos? ¿cómo contar o sumar dos números?

Veamos primero cómo manejamos nuestra cotidiana numeración decimal, pues con ella repararemos en detalles que normalmente obviamos, y que son generalizables a cualquier sistema de numeración.

Consideremos los números positivos. Para contar usamos los símbolos del 0 al 9 de la siguiente manera: 0,1,2,3,4,5,6,7,8,9.. y 10, 11, 12 etc. Acabados los diez símbolos o guarismos, volvemos a empezar desde el principio, incrementando en uno la cifra superior. Al 9 le sigue el 10, y así sucesivamente. Cada cifra dentro de un número tiene un peso, pues su valor depende de la posición que ocupe. En '467' el '7' vale 7, mientras que en '7891' vale 7000.

Así dividimos los números en unidades, decenas, centenas.. según el guarismo se encuentre en la primera, la segunda, la tercera posición (contando desde la derecha). Las unidades tienen peso '1', pues el '7' en '417' vale 7*1=7. Las unidades de millar tienen peso '1000', pues '7' en '7891' vale 7*1000=7000. Si numeramos las cifras de derecha a izquierda, comenzando a contar con el cero, veremos que la posición N tiene peso 10^N. Esto es así porque la base de numeración es el 10. El número 7891 lo podemos escribir como

7891 = 7*1000 + 8*100 + 9*10 + 1*1 = 7*10^3 + 8*10^2 + 9*10^1 + 1*10^0

Puede parecer una tontería tanto viaje, pero cuando trabajemos con otros sistemas de numeración vendrá bien tener esto en mente. Vamos con los números binarios. Ahora tenemos sólo dos símbolos distintos, 0 y 1, por lo que la base es el 2. El peso de una cifra binaria es 2^N

11101 = 1*10000 + 1*1000 + 1*100 + 0*10 + 1*1 = 1*10^4 + 1*10^3 + 1*10^2 + 0*10^1 + 1*10^0

¿Por qué seguimos usando potencias de 10 y no de 2? Porque ahora estamos usando notación binaria, y del mismo modo que en decimal al 9 le sigue al 10, en binario al 1 le sigue el 10. De aquí en adelante allí donde haya pie a confusión colocaré una 'd' para indicar decimal, 'b' para binario, 'o' para octal y 'h' para hexadecimal. O sea, que 2d = 10b

Uno ve un número tan feo como 11101b y así, a botepronto, puede no tener ni idea de a qué número nos referimos. Si expresamos lo anterior en notación decimal y echamos las cuentecillas vemos que

11101b = 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 16 + 8 + 4 + 1 = 29

Con la práctica se cogerá un poco soltura con estas cosas (y tampoco hace falta, en la mayoría de los casos, tener que manejarse mucho con esto), aunque no está de más tener una calculadora para estos menesteres.

Para convertir nuestro 29 decimal a binario, se puede hacer o bien a ojo (cosa que puede hacer bizquear a más de uno), calculadora en ristre o, si no queda más remedio, a mano. Un modo sencillo de hacerlo a mano consiste en dividir sucesivamente por la base, y extraer los restos. Por ejemplo, si quisiéramos hacer algo tan (ejem) útil como obtener los dígitos decimales de 29d, vamos dividiendo entre 10:

29 / 10 = 2, y de resto 9
2 / 10 = 0, y de resto 2

Cuando llegamos a una división que da 0 es que hemos terminado, así que recogemos los restos en orden inverso: 2, 9. Si hacemos lo mismo con base 2, para convertir a binario:

29 / 2 = 14, resto 1
14 / 2 = 7, resto 0
7 / 2 = 3, resto 1
3 / 2 = 1, resto 1
1 / 2 = 0, resto 1, y terminamos

1,1,1,0,1 => 11101

Los números binarios son imprescindibles, pero un poco engorrosos de manipular por humanos, porque cuando tenemos un número de más de 6 o 7 cifras ya nos empieza a bailar el asunto. Para ello se pusieron en práctica los sistemas octal y hexadecimal (que más o menos han existido desde que el mundo es mundo, pero se pusieron de moda con eso de la informática).

Octal	Base 8	0,1,2,3,4,5,6,7
Hexadecimal	Base 16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

El octal es muy simple porque todo funciona como hemos dicho, sólo que en vez de 0 y 1, contamos hasta el 7. Con el hexadecimal estamos en las mismas, con la salvedad de que como no existen 16 guarismos, nos tenemos que "inventar" los 6 últimos con las letras A,B,C,D,E y F (que valen 10,11,12,13,14 y 15 respectivamente)

La gracia del octal y el hexadecimal reside en que hay una equivalencia directa entre estos sistemas y el binario.

Octal:		Hexadecimal:			
0	000	0	0000	8	1000
1	001	1	0001	9	1001
2	010	2	0010	A	1010
3	011	3	0011	B	1011
4	100	4	0100	C	1100
5	101	5	0101	D	1101
6	110	6	0110	E	1110
7	111	7	0111	F	1111

Para formar un número en octal no tenemos más que agrupar las cifras binarias de 3 en 3 y sistituir; lo mismo en hexadecimal, pero de 4 en 4, y sustituir equivalencias. Por ejemplo, 145o se puede separar en 1 4 5, 001 100 101, formando el número binario 001100101 (los ceros a la izquierda podríamos quitarlos porque no tienen ningún valor, pero luego veremos que a veces es útil dejarlos puestos). Podemos ver entonces los sistemas octal y hexadecimal como abreviaturas de la notación binaria. Siempre es más fácil recordar 1F5E que 0001111101011110, ¿o no? Para obtener el valor numérico decimal aplicamos lo de siempre, con la base apropiada:

1*16^3 + 15*16^2 + 5*16^1 + 14*16^0 = 8030 (porque Fh=15d, Eh=14d)

Es frecuente, para evitar confusiones cuando programemos, colocar un cero ("0") delante de los números expresados en hexadecimal que empiecen por una letra, pues el ensamblador podría interpretarlo como una palabra clave o una posible etiqueta (que a lo mejor está definida más adelante en el código) en lugar de un número. En el caso de los 80x86, la palabra ADD es una instrucción para sumar enteros, mientras que 0ADDh sería el número decimal 2781. Veremos que la sintaxis de NASM permite también escribir 0xADD en su lugar.

- Operaciones lógicas

Existen algunas operaciones básicas que se pueden realizar con bits denominadas operaciones lógicas. Éstas se llaman normalmente por sus nombres ingleses: NOT, AND, OR, XOR. Existen algunas más, pero en realidad son combinaciones de las anteriores (de hecho XOR también lo es de las otras, pero es suficientemente útil para justificar su presencia).

La operación NOT ("no") se realiza sobre un bit, y consiste en invertir su valor. NOT 0 = 1, NOT 1 = 0. Normalmente se asimilan estos valores con "verdadero" (uno) y "falso" (cero). Así, podemos leer "no falso = verdadero", "no verdadero = falso".

AND significa "y", y relaciona dos bits. El resultado es verdadero si y sólo si los dos operandos son verdaderos:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

OR viene de "o"; para saber el resultado de "A OR B" nos preguntamos, ¿es verdadero A o B?

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

Ese resultado será verdadero con que uno de los operandos sea verdadero.

XOR tiene su origen en "eXclusive OR" ("o" exclusivo). Es verdadero si uno y sólo uno de los operandos es verdadero:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

Podemos definir (y de hecho se hace) estos mismos operadores para agrupaciones de bits, y se realizan entonces entre bits de igual peso. Por ejemplo 1101 XOR 0011 = 1110. Aunque según su propia definición sólo se pueden realizar con ceros y unos, podemos abreviar lo anterior como 0Dh XOR 3h = 0Eh

¿Sirven para algo las operaciones lógicas? La respuesta es, sin lugar a dudas, sí. Un circuito digital está compuesto casi exclusivamente por pequeños elementos llamados puertas lógicas que realizan estas operaciones, que se combinan masivamente para formar sistemas completos.

Claro que nosotros no vamos a diseñar circuitos, sino a programarlos, pero en este caso seguirán siendo útiles. Supongamos que queremos saber si los bits 0,2 y 4 de un grupo de 8 bits llamado 'A' valen 1. Un procesador, como veremos, incluye entre sus operaciones básicas las lógicas; podríamos hacer algo como "00010101 XOR (A AND (00010101))" Con dos operaciones (AND, y luego con lo que dé, XOR) no tenemos más que mirar el resultado: si es 0, esos 3 bits estaban a uno, y si no lo es, al menos uno de ellos estaba a cero. ¿Por qué? La operación AND pone a 0 todos los bits excepto los número 0,2 y 4, que los deja como están. La operación XOR lo que hace es poner a 0 los bits que estuvieran a uno, porque 1 XOR 1 = 0. Si el resultado no es cero, alguno de esos tres bits era un cero.

- Representación de información en un ordenador

Un procesador (llamados indistintamente así, "microprocesadores" o incluso simplemente "micros") contiene en su interior pequeños dispositivos que pueden almacenar números de cierta cantidad de bits, denominados registros. Ahí guardará la información que vaya a manejar temporalmente, para luego, si procede, guardarla en memoria, en el disco duro, o donde sea. En el caso de los 80x86 los registros básicos son de 16 bits, por lo que en uno de estos registros podemos escribir cualquier número desde el 0 hasta el 2^N - 1, es decir, desde 0000000000000000b hasta 1111111111111111b. Sin embargo, siguiendo este razonamiento, sólo hemos determinado cómo representar números enteros positivos. ¿Qué pasa con números como -184 o incluso 3.141592?

-Aritmética de enteros

Vamos a ver cómo operar con números en binario para entender la importancia del sistema de representación que se emplee. La suma/resta de enteros positivos en binario es idéntica a cuando empleamos representación decimal, pero con unas regla mucho más sencillas:

0+0=0
0+1=1
1+0=1
1+1=0 y me llevo 1

0-0=0
0-1=1 y me llevo 1
1-0=1
1-1=0

Por ejemplo, sumando/restando 10101 + 110:

```
llevadas:      1                  1 1 1
             1 0 1 0 1            1 0 1 0 1
SUMA    +      1 1 0     RESTA -      1 1 0
             - - - - -            - - - - -
             1 1 0 1 1            0 1 1 1 1
```

Las "llevadas" se conocen como bits de acarreo o, en inglés, de carry.

Si representamos números con un máximo número de bits, puede darse el caso de que al sumarlos, el resultado no quepa. Con cuatro bits 1111+1=10000, dando un resultado de 5 bits. Como sólo nos podemos quedar con los cuatro últimos, el carry se pierde. En un ordenador el carry se almacena en un sitio aparte para indicar que nos llevábamos una pero no hubo dónde meterla, y el resultado que se almacena es, en este caso, 0000. Salvo que se diga lo contrario, si el resultado de una operación no cabe en el registro que lo contiene, se trunca al número de bits que toque.

Un modo de representar tanto números negativos como positivos, sería reservar un bit (el de mayor peso, por ejemplo) para indicar positivo o negativo. Una posible representación de +13 y -13 con registros de ocho bits:

00001101 +13
10001101 -13

Para sumar dos números miraríamos los bits de signo; si son los dos negativos o positivos los sumamos y dejamos el bit de signo como está, y si hay uno negativo y otro positivo, los restamos, dejando el bit de signo del mayor. Si quisiéramos restarlos haríamos algo parecido, pero considerando de manera distinta los signos. Cómo primer método no está mal, pero es mejorable. Saber si un número es negativo o positivo es tan sencillo como mirar el primer bit,

pero hay que tener en cuenta que esas comparaciones y operaciones las tiene que hacer un sistema electrónico, por lo que cuanto más directo sea todo mejor. Además, tenemos dos posibles representaciones para el cero, +0 y -0, lo cual desperdicia una combinación de bits y, lo que es peor, obliga a mirar dos veces antes de saber si el resultado es 0. Una manera muy usada de comprobar si un número es igual a otro es restarlos; si la resta da cero, es que eran iguales. Si hemos programado un poco por ahí sabremos que una comparación de este tipo es terriblemente frecuente, así que el método no es del todo convincente en este aspecto.

La representación en complemento a dos cambia el signo a un número de la siguiente manera: invierte todos los bits, y suma uno. Con nuestro querido +13 como 00001101b, hacemos lo siguiente: al invertir los bits tenemos 11110010b, y si sumamos 1, 11110011b. Ya está, -13 en complemento a dos es 11110011.

Molaba más cambiar el signo modificando un bit, es cierto, pero veamos la ventaja. ¿Qué pasa cuando sumamos +13 y -13, sin tener en cuenta signos ni más milongas?

```
00001101
11110011
--------
00000000
```

Efectivamente, cero. Para sumar números en complemento a dos no hace falta comprobar el signo. Y si uno quiere restar, no tiene más que cambiar el signo del número que va restando (inviertiendo los bits y sumando uno), y sumarlos. ¿Fácil, no? Lo bueno es que con este método podemos hacer que el primer bit siga significando el signo, por lo que no hace falta mayores comprobaciones para saber si un número es positivo o negativo. Si seguimos con ocho bits tenemos que los números más altos/bajos que se pueden representar son 0111111b (127) y 10000000 (-128) En general, para N bits, podremos almacenar cualquier número entre 2^(N-1) - 1 y -2^(N-1).

Es un buen momento para introducir el concepto de overflow (desbordamiento, aunque no tan común en español) Cuando al operar con dos números el resultado excede el rango de representación, se dice que ha habido overflow. Esto es sutilmente diferente a lo que sucedía con el carry. No es que "nos llevemos una", es que el resultado ha producido un cambio de signo. Por ejemplo, la suma de 10010010 + 10010010, si consideramos ambos números con su signo (-110d + -110d), nos da como resultado 00100100, 36d. Ahora el resultado ya no es negativo, porque hemos "dado la vuelta al contador", al exceder -128 en la operación. Los procesadores también incluyen un indicador que se activa con estas situaciones, pero es tarea del programador comprobar los indicadores, y saber en todo momento si trabajamos dentro del rango correcto.

Una ventaja adicional del complemento a dos es que operamos indistintamente con números con signo o sin él, es decir, si queremos podemos utilizar todos los bits para representar números positivos exclusivamente (de 0 a 2^N - 1), haciendo exactamente las mismas operaciones. El que un número sea entero positivo o entero con signo depende únicamente de la interpretación que haga el programador del registro; tendrá que comprobar, si fuera necesario, los bits de overflow y carry para operar correctamente según sus intereses.

Las multiplicaciones/divisiones se pueden descomponer en una retahíla de sumas y restas, por lo que no merece la pena ahondar demasiado; responden a consideraciones similares, salvo que el producto de N bits por N bits da, en general, 2N bits, y que ahora sí que habrá que especificar si la operación va a ser entre números con o sin signo.

-Números reales

La forma más simple de representación de números reales es sin duda la de punto fijo (o coma fija, según digas usar una cosa u otra para separar las cifras). La idea consiste en plantar un punto entre los bits, y ya está. Ejemplo:

1110.0111b

El peso de las cifras a la izquierda del punto son las mismas que para enteros, mientras que a la derecha el peso sigue disminuyendo en potencias de dos.. pero negativas.

1110.0111b = 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0 + 0*2^(-1) + 1*2^(-2) + 1*^2(-3) +1*2^(-4)

Como 2^(-N) = 1/(2^N) tenemos que los pesos son 0.5, 0.25, 0.125 etc

1110.0111b = 8 + 4 + 2 + 0 + 0 + 0.25 + 0.125 + 0.0625 = 14.4375

que no es más que coger el número 11100111b, pasarlo a decimal, y dividirlo entre 2^P, siendo P el número de cifras a la derecha del punto. Se opera exactamente igual que con enteros (podemos dejar un bit de signo si nos apetece, haciendo complemento a dos), siempre que los dos operandos tengan no sólo el mismo número de bits, sino el punto en el mismo sitio. Podemos seguir usando nuestra calculadora de enteros para manejar reales si decidimos usar este método.

Algunos dispositivos de cálculo, especialmente de procesado de señal, usan este sistema por su sencillez y velocidad. La "pega" es la precisión. Ahora no sólo tenemos que intentar "no pasarnos" del número de bits por exceso, sino que las potencias negativas limitan nuestra precisión "por abajo". Un número como 1/3 (0.3333 etc) podemos intentar expresarlo en los 8 bits de arriba, pero nos encontraremos con que el número de potencias dos negativas necesarias para hacerlo nos sobrepasan ampliamente (infinitas).

0000.0101 es lo mejor que podemos aproximar, y sin embargo tenemos un número tan feo como 0.3125 El problema no es tanto el error como el error relativo. Cometer un error de aproximadamente 0.02 puede no ser grave si el número es 14 (0.02 arriba o abajo no se nota tanto), pero si se trata de 0.3333.., sí.

Los números en punto flotante (o coma flotante) amortiguan este problema complicando un poco (un mucho) las operaciones. Parecerá muy muy engorroso, pero es lo que se usa mayormente en ordenadores, así que algo bueno tendrá.

Vamos con un primer intento con anestesia, para no marear demasiado. Digamos que de los 8 bits de antes (que, de todos modos, son muy poquitos; lo normal en números reales es 32,64,80..) dedicamos 2 a decir cuánto se desplaza el punto, y los 6 restantes a almacenar el número propiamente.

Un número tal que 11011110 que representase a un número real estaría en realidad compuesto por 1.10111 y 10, o sea, 3.4375 y 2; el dos indica posiciones desplazadas (potencias de 2), así que el número en cuestión sería 3.4375 * 2^2 = 13.75 Podríamos aún así representar un número tan pequeño como 0.03125, 0.03125*2^0, con "0.00001 00", o sea, 00000100 Hemos ampliado el margen de representación por arriba y por abajo, a costa de perder precisión relativa, es cierto, pero haciéndola más o menos constante en todo el rango, porque cada número se puede escribir con 6 cifras binarias ya esté el punto un poco más a la izquierda o a la derecha. La parte que representa las cifras propiamente se denomina mantisa, y en este caso los dos bits separados forman el exponente.

El estándar IEEE 754 describe cómo almacenar números en punto flotante de manera un poco más profesional que todo esto. Para 32 bits se dedican 8 para el exponente, 23 para la mantisa y 1 para el signo. Se trabaja sólo con números positivos y bit de signo aparte porque dada la complejidad de las operaciones con números reales, no hay ninguna ventaja importante en usar cualquier otra cosa. En este formato el exponente se calcula de modo que la mantisa quede alineada con un uno a la izquierda de todo, y este primer uno no se almacena. El exponente además se guarda en "exceso 127", que consiste en restarle 127 al número almacenado: si la parte del exponente fuese 11011111b = 223d, al restarle 127 tendríamos el exponente que representa, 96. Vamos con un ejemplo:

signo	exponente	mantisa
1	0111 0101	101 0101 0110 1110 1110 0011

El bit de signo es 1, así que tenemos un número negativo. La mantisa es 1.10101010110111011100011, porque el primer '1' se sobreentiende siempre, ya que al almacenar el número se alteró el exponente para alinearlo así; el valor decimal de este número es "1.66744649410248". El exponente es 01110101b = 117d, es decir, -10 porque en exceso 127 se le resta esta cantidad. El número representado es -1.66744649410248*2^-10 = -0.0016283657169 más o menos (suponiendo que no me haya confundido).

Las operaciones con números en punto flotante son mucho más complejas. Exigen comprobar signos, exponentes, alinear las mantisas, operar, normalizar (esto es, desplazar la mantisa / modificar el exponente para que quede alineada con ese primer "1" asomando).. Sin embargo existen circuitos específicos dentro de los procesadores para realizar estos tejemanejes, e incluso para calcular raíces cuadradas, funciones trigonométricas... Lo que sucede es que son bastante más lentos, como era de esperar, que los cálculos con enteros, por lo que se separan unas y otras operaciones según el tipo de datos que vayamos a manejar. El estándar mencionado admite números especiales como +infinito, -infinito y NaN (Not a Number), con los que es posible operar. Por ejemplo, infinito+infinito=infinito, mientras que infinito-infinito es una indeterminación, y por tanto, NaN.

-Codificación BCD

BCD viene de Binary Code Digit, o dígito en código binario, y es una manera alternativa de codificar números. Consiste en algo tan simple como dividir el número en sus dígitos decimales, y guardar cada uno en 4 bits. Por ejemplo, el número 1492 se almacenaría en como mínimo 16 bits, resultando el número 1492h, o 0001 0100 1001 0010b (separo las cifras binarias de 4 en 4 para que se vea mejor). Así, sin más, se desperdicia un poco de espacio ya que no tienen cabida los seis nibbles 1010, 1011, 1100, 1101, 1110, y 1111 (el número más grande representable así en 16 bits es 9999), por lo que a veces se emplean estas combinaciones extra para el signo, un punto, una coma, etc.

La ventaja de este modo de almacenar números es que la conversión a decimal es inmediata (cada grupo de 4 bits tiene una correspondencia), mientras que si nos limitásemos a guardar el número que representa almacenado en binario el procedimiento sería más costoso. Con la potencia de cómputo de que se dispone ahora, y dado que las conversiones sólo son necesarias para representar la información final para que la lea un humano, puede parecer de escasa utilidad, pero en dispositivos en los que estas operaciones son tan frecuentes como cualquier otra, compensa especializar determinados circuitos a su manipulación. Algunas posibles aplicaciones podrían ser terminales "tontos" de banca, o calculadoras. Un caso claro de esta última podrían ser algunas calculadoras HP basadas en el microprocesador Saturn que tiene un modo de funcionamiento completamente en BCD. De hecho los números reales los almacena en BCD en 64 bits, formando 16 dígitos: uno para el signo, 12 para la mantisa, y 3 para el exponente. El BCD no será, sin embargo, la representación más habitual.

Existen dos tipos de formatos BCD: el conocido como empaquetado, con 2 dígitos en cada byte (que, recordemos, es con frecuencia la mínima unidad "práctica" en un procesador), y desempaquetado, con un dígito por byte (dejando siempre los 4 bits superiores a cero).

-Texto, sonido, imágenes

Queda ver cómo se manejan en un sistema digital otros datos de uso frecuente. Generalmente la unidad mínima de almacenamiento es el byte, y por ello la codificación más habitual para texto asigna un byte a cada carácter considerado útil, hasta formar una tabla de 256 correspondencias byte-carácter. Los que determinaron semejante lista de utilidad fueron, como casi siempre, americanos, en lo que se conocen como códigos ASCII. Como dejaron fuera del tintero algunos caracteres extranjeros, eñes, tildes variopintas, etc, surgieron numerosas

variantes. Realmente los primeros 128 caracteres (los formados con los siete bits menos significativos, y el octavo bit a cero) son comunes a todas ellas, mientras que los 128 siguientes forman los caracteres ASCII extendidos, que son los que incluyen las peculiaridades.

Si alguien tiene una ligera noción de lo que hacen los chinos, árabes o japoneses, se dará cuenta de que con 256 caracteres no hacen nada ni por el forro. A fin de hacer un único código universal surgió el UNICODE, que es con el que deberían almacenarse todos los textos, aunque por desgracia no sea así. Sin embargo tiene una implantación cada vez mayor, aunque a un paso un poco lento. Si bien tiene unas pequeñas reglas que no hacen la correspondencia directa código -> carácter, se podría decir grosso modo que cada carácter se representa con 2 bytes, y no uno, dando lugar a, en principio, 65536 caracteres distintos, dentro de los cuales, por supuesto, se incluye la tablita ASCII.

El sonido se produce cuando una tensión variable con el tiempo llega a un altavoz, convirtiendo esas variaciones de tensión en variaciones de presión en el aire que rodea a la membrana. Para almacenar audio se toman muestras periódicas del valor de dicha tensión (cuando se graba sucede al revés; son las variaciones de presión sobre el micrófono las que producen el voltaje mencionado), guardando ese valor. La precisión con que se grabe el sonido depende del número de muestras que se tomen por segundo (típicamente 8000, 22050 ó 44100) y de los bits que se empleen para cada muestra (con frecuencia 8 ó 16). De este modo, para grabar 1 minuto de audio a 44100 muestras por segundo, en estéreo (dos canales), y 16 bits, necesitaremos 60*44100*2*16 bits, es decir, 10 megabytes aproximadamente. Existen formatos de audio que toman esas muestras y las comprimen aplicando un determinado algoritmo que, a costa de perjudicar más o menos la calidad, reducen el espacio ocupado. Un archivo ".wav" corresponde, generalmente, a un simple muestreo del audio, mientras que uno ".mp3" es un formato comprimido.

Para las imágenes lo que hacen es muestrear en una cuadrícula (cada cuadrado, un píxel) las componentes de rojo, verde y azul del color, y asignarles un valor. Un archivo ".bmp" por ejemplo puede asignar 3 bytes a cada píxel, 256 niveles para cada color. Se dice entonces que la calidad de la imagen es de 24 bits. Una imagen de 800 píxels de ancho por 600 de alto ocuparía 800*600*24 bits, casi 1.4 megas. De nuevo surgen los formatos comprimidos como ".gif",".png",".jpg".. que hace que las imágenes ocupen mucho menos espacio.

- Memorias

Una memoria es un dispositivo electrónico en el que se pueden escribir y leer datos. Básicamente se distinguen en ROM, o memorias de sólo lectura (Read Only Memory), y RAM, mal llamadas memorias de acceso aleatorio (Random Access Memory). Esto último viene de que hay cierto tipo de sistemas de almacenamiento que son secuenciales, lo que significa que para leer un dato tienes primero que buscarlo; es lo que sucede con las cintas, que tienes que hacerlas avanzar o retroceder hasta llegar al punto de interés. En ese sentido, tanto las ROM como las RAM son memorias de acceso aleatorio (puedes acceder a cualquier posición directamente); la distinción entre unas y otras es actualmente bastante difusa, y podríamos decir que se reduce ya únicamente a la volatilidad, que ahora comentaré.

Una ROM pura y dura viene grabada de fábrica, de tal modo que sólo podemos leer los datos que se escribieron en ella en el momento de su fabricación. Existen luego las PROM (P de Programmable, que evidentemente significa programable), que pueden ser grabadas una única vez por el comprador. En este contexto se pueden encontrar las siglas OTP-ROM, de One Time Programmable, o programables una vez. La EPROM es una memoria que se puede borrar completamente por medios no electrónicos (Erasable Programmable ROM), generalmente descubriendo una pequeña abertura que tienen en la parte superior del encapsulado y exponiéndola durante un tiempo determinado a luz con contenido ultravioleta, dejándolas así listas para volver a programarlas. Las EEPROM(Electrically Erasable PROM) son memorias que pueden ser borradas sometiéndolas a tensiones más altas que las habituales de alimentación del circuito. Son de borrado instantáneo, al contrario que las EPROM normales, pero generalmente exigen también ser extraídas del circuito para su borrado. Todas éstas han ido dejando paso a las memorias FLASH, que pueden ser borradas

como parte del funcionamiento "normal" de la memoria, completamente o incluso por bloques; sin embargo la velocidad de escritura es muy inferior a la de las memorias RAM. La cualidad que caracteriza fundamentalmente todas estas memorias es que no son volátiles; uno puede apagar el circuito y el contenido de la memoria seguirá ahí la próxima vez que lo encendamos de nuevo. La BIOS de los ordenadores constituye un pequeño programa almacenado en ROM (antaño EPROMs, ahora FLASH) que ejecutará el procesador nada más comenzar a funcionar, y con el que comprobará que todos los cacharrillos conectados a él están en orden, para a continuación cederle el paso al sistema operativo que habrá ido buscar al disco duro o adonde sea.

Las memorias RAM, por el contrario, pierden toda su información una vez desconectadas de su alimentación; es, sin embargo, un precio muy pequeño a pagar a cambio de alta velocidad, bajo coste y elevada densidad de integración (podemos hacer memorias de gran capacidad en muy poco espacio), sin olvidar que podemos escribir en ella cuantas veces queramos, de manera inmediata. Aunque hay muchos subtipos de memoria RAM, sólo voy a comentar muy por encima los dos grandes grupos: estáticas (SRAM) y dinámicas (DRAM). Las denominadas SDRAM no son, como algún animal de bellota ha puesto por ahí en internet, Estáticas-Dinámicas (una cosa o la otra, pero no las dos). Las memorias SDRAM son RAMs dinámicas síncronas, lo cual quiere decir que funcionan con un relojito que marca el ritmo de trabajo. Las DRAM que se montan en los ordenadores actuales son de este tipo.

La distinción entre estática y dinámica es sencilla. Una memoria estática guarda sus datos mientras esté conectada a la alimentación. Una dinámica, además de esto, sólo retiene la información durante un tiempo muy bajo, en algunos casos del orden de milisegundos. ¿Cómo? ¿Milisegundos? ¿Dónde está el truco? Lo que sucede es que cada bit está guardado en la memoria en forma de carga de un pequeño condensador, que retiene una determinada tensión entre sus extremos. Ese condensador por desgracia no es perfecto (tiene pérdidas), así que con el paso del tiempo ese voltaje va decayendo lentamente hasta que, si nadie lo evita, su tensión cae a un nivel tan bajo que ya no se puede distinguir si lo que había ahí era un "1" o un "0" lógico. Por ello tiene que haber un circuito anexo a la memoria propiamente dicha que se encargue cada cierto tiempo de leer la tensión de ese condensador, y cargarlo para elevarla al nivel correcto. Todo esto se hace automáticamente a nivel de circuito, por lo que al programador le da igual el tipo de memoria del que se trate.

Las memorias dinámicas son más lentas que las estáticas, y no sólo por este proceso denominado refresco, pero a cambio son mucho más baratas e integrables; la memoria principal de un ordenador es DRAM por este motivo.

Las SRAM son, por tanto, más caras, grandes y rápidas. Se reservan para situaciones en las que la velocidad es vital, y ese sobrecoste bien merece la pena; es el caso de las memorias cache, que son las que se encuentran más cerca del procesador (generalmente dentro del propio encapsulado). Un caso particular de memoria estática, aunque por su naturaleza no considerada como tal, es la de los registros del procesador; se trata de, con diferencia, la memoria más rápida y cara de todo el sistema, y un procesador sólo cuenta con unos pocos registros (algunas decenas en el mejor de los casos) de tamaño variable (en el caso de la familia x86, 16 y 32 bits, fundamentalmente).

Una memoria está estructurada en palabras, y cada palabra es un grupo de bits, accesible a través de su dirección. Cada dirección es un número que identifica a una palabra. Un circuito de memoria está conectado entonces a, al menos, dos tipos de líneas, o hilos por donde viajan los bits (voltaje): datos, por donde viaja la información, y direcciones, por donde se identifica la posición que ocupa esa información. Existe un tercer tipo de líneas denominado de control; con estás líneas se indica el tipo de operación que se desea realizar sobre la memoria (leer o escribir, básicamente)

Un ejemplo de memoria podría tener 8 líneas de datos y 16 de direcciones. Si uno quisiera acceder a la palabra en la posición 165, colocaría el número 0000000010100101 (165 en binario) sobre las líneas de direcciones, y tras indicarle la operación con las líneas de control, aparecería en un pequeño lapso de tiempo el byte almacenado en esa dirección (un byte, en

este caso, porque hay 8 líneas de datos y, por tanto, 8 bits). Para escribir sería el procesador quien colocaría el dato sobre las líneas de datos, así como la dirección, y al indicar la operación de escritura, la memoria leería ese dato y lo almacenaría en la posición indicada. Una memoria con un bus (o conjunto de líneas) de 16 bits de ancho, y un bus de datos de 8 bits, tiene una capacidad de 2^16 * 8 = 524288 bits, es decir, 65536 bytes (64k). La anchura del bus de datos influye no sólo en el tamaño total de la memoria para un bus de direcciones dado, sino también la velocidad a la que son leídos/escritos estos datos; con un bus de 64 bits podríamos traer y llevar 8 bytes en cada viaje, y no uno sólo.

- Procesadores, ensambladores

Un procesador está compuesto fundamentalmente por una unidad de acceso a memoria (que se encarga de intercambiar datos con el exterior), un conjunto de registros donde almacenar datos y direcciones, una unidad aritmético-lógica (ALU o UAL, según se diga en inglés o en español) con la que realizar operaciones entre los registros, y una unidad de control, que manda a todas las demás. Por supuesto este modelo rudimentario se puede complicar todo lo que queramos hasta llegar a los procesadores actuales, pero en esencia es lo que hay.

El procesador obtiene la información de lo que debe hacer de la memoria principal. Ahí se encuentran tanto el código (el programa con las instrucciones a ejecutar) como los datos (la información sobre la que debe operar). Algunos procesadores tienen código y datos en dos sistemas de memoria diferentes, pero en el caso de los ordenadores domésticos van mezclados.

Uno de los registros, conocido como contador de programa, contiene la dirección de memoria donde se encuentra la siguiente instrucción a ejecutar. En un procesador simplificado, en cada ciclo de reloj se accede a la memoria, y se lee el dato apuntado por el contador de programa (llamado normalmente PC por sus siglas en inglés, Program Counter). Este dato contiene el código de la instrucción a ejecutar, que puede ser leer algo de memoria para guardarlo en un registro, guardar el contenido de un registro en memoria, realizar alguna operación con uno o más registros.. El PC mientras tanto se incrementa para apuntar a la nueva instrucción. Los distintos subcircuitos que forman el procesador están interconectados obedeciendo ciertas señales a modo de llaves de paso. Cuando el procesador obtiene de la memoria el código de una instrucción, es recogido por la unidad de control que decide qué datos deben moverse de un registro a otro, qué operación ha de realizarse, etc, enviando las señales necesarias a estos subcircuitos. Por ejemplo, es frecuente que el programa tenga que realizar un salto en función de los resultados de una operación (para ejecutar o no un cierto fragmento de código) a la manera de las estructuras de alto nivel if (condición) then {...} else {...}. Un salto no supone nada más que cargar en PC otro valor. La unidad de control comprueba por lo tanto si la condición es verdadera, y si es así le dice al PC que se cargue con el valor nuevo; en caso contrario, se limita a indicarle qué únicamente ha de incrementarse para apuntar a la siguiente instrucción.

El conjunto de posibles instrucciones que puede interpretar y ejecutar un procesador recibe el nombre de código máquina. El código asociado a cada instrucción se llama código de operación. Es posible hacer un programa con un editor hexadecimal (que en lugar de trabajar en ASCII nos permite introducir los valores numéricos de cada byte en cada posición de un archivo), introduciendo byte a byte el código de cada instrucción. El problema de esto es que, aparte de ser terriblemente aburrido, es muy lento y propenso a error. Para esto es para lo que están los ensambladores.

El ensamblador es un lenguaje de programación que se caracteriza porque cada instrucción del lenguaje se corresponde con una instrucción de código máquina; también se conoce como ensamblador el programa que realiza esta tarea de traducción (en inglés no existe confusión al distinguir assembly -el lenguaje- de assembler -el programa-). Un compilador, por el contrario, genera el código máquina a partir de unas órdenes mucho más generales dadas por el programador; muchas veces podemos compilar el mismo código fuente en distintos sistemas, porque cada compilador genera el código máquina del procesador que estemos usando. El ensamblador no es un único lenguaje; hay al menos un lenguaje ensamblador por cada tipo de

procesador existente en el mundo. En el caso de este tutorial, vamos a estudiar un conjunto de procesadores distintos que tienen en común una buena parte de su código máquina, de tal modo que un programa ensamblado (un código fuente en ensamblador pasado a código máquina) se podrá ejecutar en distintos equipos.

Por ejemplo, si tenemos una maquinita imaginaria con dos registros llamados A y B, y existe una instrucción máquina codificada como 1FE4h que copia el dato que haya en A, en B, podríamos encontrarnos con un ensamblador para este ordenador en el que escribiésemos:

MOVER A,B

de modo que al encontrarse el ensamblador con MOVER A,B lo sustituyese por 1FE4h en nuestro programa en código máquina (el ejecutable). A la palabra o palabras que representan a una determinada instrucción en código máquina se la denomina mnemónico.

Los programas que ensamblan se conocen a veces como macroensambladores, porque permiten utilizar etiquetas,macros, definiciones.. Todos estos elementos no corresponden exactamente con código máquina, y son conocidos como directivas, pues son en realidad indicaciones al ensamblador de cómo debe realizar su trabajo, facilitando enormemente la programación. Supongamos que queremos copiar el contenido del registro A a las posiciones de memoria 125,126 y 127. La manera de hacerlo más simple sería:

MOVER A,dirección(125)
MOVER A,dirección(126)
MOVER A,dirección(127)

pero a lo mejor existe una estructura en nuestro macroensamblador que nos permite sustituir todo eso por

REPITE i=125:127 {MOVER A,dirección(i)}

de modo que cuando el ensamblador detecte la directiva "REPITE" con el parámetro "i" genere, antes de proceder a ensamblar, el código equivalente de las tres líneas "MOVER".

En algunos casos se pueden llamar pseudoinstrucciones, cuando se manejan exactamente como instrucciones que en realidad no existen. Quizá el caso más claro de los 80x86 sea la instrucción NOP, que le indica al procesador que durante el tiempo que dura la ejecución de esa instrucción no haga nada (aunque parezca que no tiene ninguna utilidad, no es así, pues puede servir, por ejemplo, para crear pequeños retrasos y así esperar a que un dispositivo esté listo). Esta instrucción, estrictamente hablando, no existe. Sin embargo existe una instrucción que intercambia los contenidos de dos registros; XCHG registro1, registro2. Si los dos operandos son el mismo, la instrucción no hace nada. En el caso de nuestro NOP, lo que se codifica es la instrucción XCHG AX,AX. Cuando el procesador recoge esa instrucción, intercambia el valor de AX con él mismo, que es una manera tan buena como cualquier otra de perder un poco de tiempo.

¿Por qué aprender ensamblador? Bien, todo lo que se puede hacer en un procesador, se puede escribir en ensamblador. Además, el programa hará exactamente lo que le pidas, y nada más. Puedes conseguir programas más eficientes en velocidad y espacio que en cualquier otro lenguaje. Cuando lo que se programa es un sistema de limitados recursos, como un microcontrolador (que tienen hasta sus compiladores de C por ahí), esto es vital. Además, al existir una correspondencia directa ensamblador<->código máquina, puedes, en teoría, tomar fragmentos de código y desensamblarlo, y con suficiente experiencia modificar un programa ya compilado, sin el fuente (que es lo que se hace para desarrollar cracks, por ejemplo, que es ilícito pero notablemente didáctico).

La desventaja es que exige un trabajo mucho mayor que otros lenguajes. Para hacer algo sencillo tienes que dedicar mucho más tiempo, porque le tienes que especificar al procesador

paso por paso lo que debe hacer. En un lenguaje de más alto nivel como C, y no digamos ya Java o C++, los programas se hacen en mucho menos tiempo, y a menudo con un rendimiento similar (bueno, en Java no, desde luego). Además, cada procesador tiene su propio juego de instrucciones, y en consecuencia propio código máquina y propio lenguaje ensamblador (lo que en el caso que trataremos nosotros no supone demasiado problema porque un altísimo porcentaje del código será común a todos los procesadores de la familia, con pequeñas diferencias en los "extras"). Sin embargo, pese a todo ello, será interesante programar fragmentos de código en ensamblador para hacer nuestra aplicación más eficaz; y aun cuando esto no sea necesario, conocer ensamblador significa conocer el procesador que estamos usando, y con este conocimiento seremos capaces de sacar mucho más partido a los lenguajes de alto nivel, programar más eficientemente e identificar determinados errores con mayor facilidad. Sabremos incluso modificar el ensamblador generado por el compilador, antes de ser convertido definitivamente a código máquina, para optimizar nuestros programas (aunque he de reconocer que hay compiladores que hacen por sí solos un trabajo difícilmente superable). Y, por supuesto, y como siempre digo, está el placer de hacer las cosas uno mismo. Como habrás podido imaginar, el que escribe estas líneas es un entusiasta de la programación en ensamblador; soy, sin embargo, el primero en admitir sus limitaciones, pero no me tomaría en serio a ningún programador, aficionado o profesional, que dijera abiertamente que saber ensamblador no sirve de nada. Creo que para programar bien hay que, al menos, conocer el lenguaje ensamblador de algún microprocesador (aunque no sea el que se vaya a usar).

Más allá del romanticismo, o del conocimiento del funcionamiento del microprocesador como base para comprender otros lenguajes, la discusión más cabal sobre el uso práctico del ensamblador con la que me he topado es sin duda la que se encuentra en elAssembly-Howto por Konstantin Boldyshev y Francois-Rene Rideau, en su apartado "Do you need assembly?". En este documento encontrará el lector una argumentación bastante equilibrada a este respecto, así como temas tan útiles como la sintaxis de otros ensambladores distintos de NASM, interacción con otros lenguajes, macros..

CAPÍTULO I: BIENVENIDO A LA FAMILIA (INTEL)

Aunque ahora (¿casi?) todo el mundo gasta Pentium XXL o algo de calibre semejante, en el lanzamiento de cada procesador de esta familia se ha guardado siempre compatibilidad con los miembros anteriores. Así, es teóricamente posible ejecutar software de los años 80 en un ordenador actual sin ningún problema (suponiendo que tengamos instalado un sistema operativo que lo apoye).

Todo el asunto arranca con el 8086, microprocesador de 16 bits que apareció allá por el año 78; al año siguiente Intel lanzó el 8088, una versión más ecónomica pues contaba con un ancho de bus de datos de 8 bits (lo que le hacía el doble de lento en accesos a memoria de más de 8 bits). Estos micros direccionaban hasta 1Mb de memoria, y en vez de mapear los puertos E/S sobre ella empleaban un direccionamiento específico que abarcaba hasta teóricamente 65536 puertos. Este modo de acceder a los controladores de periféricos se ha mantenido en toda la familia, ya que es un punto de vital importancia en el asunto de la compatibilidad.

En 1982 se lanzaron los 80186/80188, que en su encapsulado además del procesador propiamente dicho incluían timers, controlador de interrupciones... Incluían algunas instrucciones adicionales para facilitar el trabajo a los compiladores, para acceso a puertos... Hasta donde sé se emplearon como procesadores de E/S en tarjetas de red y similares, más que en ordenadores.

Ese mismo año aparece en escena el 80286, mucho más complejo que el 8086. Aunque el juego de instrucciones es prácticamente el mismo, se diseñó pensando en la ejecución de sistemas multitarea. Contaba con dos modos de ejecución, el real y el protegido. En el modo real el 286 opera como un 8086; sin embargo en modo protegido cuando un programa intenta acceder a una región de la memoria o ejecutar una determinada instrucción se comprueba antes si tiene ese privilegio. Si no es así se activa un mecanismo de protección que generalmente gestionará el sistema operativo, que es el programa que controla los derechos del resto de programas. Además este modo protegido permitía manejar hasta 16Mb de memoria RAM (por contar con un bus de direcciones de 24 bits), aunque como los registros seguían siendo de 16 bits, no posibilitaba el manejo de bloques de memoria -segmentos- de más de 64kb. En cualquier caso, más allá del mero aumento de la velocidad de reloj (hasta 10MHz en el 8086, 16, 20 e incluso 25MHz en el 286), o el espacio de memoria direccionable, se produjo una fuerte reducción del número de ciclos de reloj por instrucción, lo que llevó a un incremento aún mayor del rendimiento.

En 1985 sale el 80386, lo que vendría a ser un peso pesado comparado a sus antecesores. Prácticamente todos los registros quedaban extendidos a 32 bits, e incluía un mecanismo de gestión de memoria más avanzado que el 286, facilitando el uso de memoria virtual (disco duro como si fuera ram). Contaba con un bus de direcciones de 32 bits, llegando a direccionar 4Gb de memoria, y memoria caché. Aparte del modo real incluía un nuevo modo protegido mejorado. En este modo protegido se permitía la ejecución de programas en un modo virtual o V86, posibilitando la ejecución de máquinas virtuales 8086; el sistema puede pasar con cierta facilidad de un modo a otro, permitiendo que funcionen varios "8086" en una especie de modo "real" al tiempo, cada uno con su memoria, estado... funcionando completamente ajeno al resto del software. Cuando se ejecuta un programa de MS-DOS (modo real) bajo Windows (modo protegido) en realidad entra en acción el modo virtual, ejecutando la aplicación con relativa seguridad aislada del sistema y resto de aplicaciones (y digo relativa porque he colgado muchas *muchas* veces el windows enredando con un programilla DOS; los mecanismos de protección del Windows dejan bastante que desear).

Un 286 en modo protegido no podía volver al modo real salvo con un reset, lo que supuso una seria traba al desarrollo de software que explotase este modo de funcionamiento (no olvidemos que por entonces el estándar de facto en sistemas operativos era MSDOS). Además, seguíamos estando limitados a segmentos de 64k, y con el 386 al caer (siempre que se lanza un procesador nuevo, están los de las siguientes generaciones en diferentes estados de

desarrollo) no compensaba el esfuerzo. Debido a la escasa relevancia del modo protegido del 286, nos referiremos genéricamente a modo protegido cuando hablemos de 386+.

Surgieron después el 386SX (un 386 económico, con bus de direcciones de 24 bits y de datos de 16, lo que permitió una cierta transición 286-386), y varios años después el 386SL, una versión del 386 que aprovechaba las nuevas tecnologías de fabricación de circuitos integrados (lo de siempre; menos consumo, más velocidad, más barato). Éste último se usó sobre todo en equipos portátiles.

El 486 era básicamente un 386 más rápido, en sus versiones SX (sin coprocesador) y DX (con él). Se hicieron famosas las versiones con multiplicador de reloj del micro, DX2 y DX4.

Y en 1993 llega el primero de los celebérrimos Pentium. Se mejoró, como en cada miembro de la familia, el número de ciclos por instrucción; bus de datos de 64 bits, más caché, más de todo. Luego llegarían los MMX (algunos dicen que MultiMedia eXtensions); un paquete de registros adicionales -bueno, no exactamente- e instrucciones especiales (SIMD, Single Instruction Multiple Data) para manipular vídeo, audio.. Toda la gama Pentium iría añadiendo sus cosillas (AMD por su parte haría lo propio en los suyos), pero sin cambios fundamentales en la arquitectura (al menos, en lo que a la visión del programador se refiere).

Aunque desde el 8086 la cosa se ha complicado mucho, se podría decir que el microprocesador que supuso el cambio más importante en esta familia fue el 386. En algunos contextos se habla de esta familia de microprocesadores como IA32 (Intel Architecture 32 bits) o i386, abarcando desde el 386 en adelante (cualquier micro compatible, Intel o no), pues básicamente funcionan todos ellos igual. El 8086 será el punto de partida de estas lecciones, ya que todos estos procesadores, cuando arrancan, se comportan como un 8086. Será el sistema operativo el que prepare el equipo para entrar en modo protegido y acceder así a todos los recursos, mecanismos de protección, etc, abandonando ya la compatibilidad con los procesadores anteriores al 386.

CAPÍTULO II: REPITE CONMIGO. TENGO UN 8086, TENGO 8086..

Nada más empezar toca hacer un acto de fuerza de voluntad, por renunciar a la mayor parte de nuestra memoria RAM, funciones MMX, direccionamiento de 32 bits.. Lo primero que vamos a ver es una descripción de los registros del primer bicho de la familia, el 8086 (aunque un 80286 es básicamente idéntico en este aspecto), pues Intel y los demás fabricantes han puesto cuidado en que todos ellos se puedan comportar como él. Es un micro de 16 bits, y como habrás adivinado sus registros son de 16 bits. Helos aquí:

Registros de datos	AX	BX	CX	DX
Punteros de pila	SP	BP		
Registros índice	DI	SI		
Registros de segmento	CS	DS	ES	SS
Registro de flags				

Los registros de datos, como su nombre indica, contienen generalmente datos. (Sí, lo sé, no parecen gran cosa, pero es lo que hay) A veces se les llama "de propósito general", y la verdad es que es un nombre más apropiado, si bien un poco más largo. Aunque tiene distinto nombre cada uno de ellos, cuentan básicamente con la misma funcionalidad, con algunas excepciones. Determinadas operaciones -por ejemplo la multiplicación- exigen que los operandos estén en registros específicos. En ese caso no quedarán más narices que usar esos concretamente.

> AX es a menudo llamado acumulador, más por motivos históricos que por otra cosa.
> BX se puede usar como registro base en algunos modos de direccionamiento, es decir, para apuntar a posiciones de memoria con él.
> CX es usado por algunas instrucciones como contador (en ciclos, rotaciones..)
> DX o registro de datos; a veces se usa junto con AX en esas instrucciones especiales mencionadas.

Cada registro de estos está dividido a su vez en dos registros de 8 bits, que pueden ser leídos o escrito de manera independiente:

> AX = AH | AL BX = BH | BL
> CX = CH | CL DX = DH | DL

Si AX contiene 00FFh, AH=00h (su parte alta) y AL=FFh (su parte baja); si lo incrementamos en 1, AX pasa a valer 0100h (lógicamente), y con ello AH=01h, AL=00h. Si en lugar de incrementar en 1 AX lo hacemos con AL (byte inferior), AH se quedará como estaba y AL será FFh+01h=00h, es decir, AX=0000h (vamos, que cuando se manipula una parte del registro la otra no se ve afectada en absoluto) Ah, por si alguno lo dudaba, H viene de high y L de low.

Uno puede mover los datos de unos registros a otros con prácticamente total libertad. También podremos realizar operaciones sobre ellos, como sumar el contenido de BX y DX para guardar el resultado en DX, y cosas así. La primera restricción al respecto (y bastante razonable) es que los operandos tendrán que ser del mismo tamaño (no podremos sumar BX con DH, por ejemplo).

Para explicar los registros que hacen referencia a memoria hay que contar brevemente qué es la segmentación.

Uno puede pensar que lo lógico y maravilloso para apuntar a una dirección de memoria es colocar dicha dirección en un registro, y ale, que apunte. Eso está muy bien para registros grandes pero da la casualidad de que con 16 bits tenemos 2^16=64k posiciones. Hombre, en aquellos tiempos estaba muy bien para según qué cosas, y aquí tampoco vamos a manejar

mucha más memoria, pero tampoco es plan. La solución por la que optó Intel fue usar dos registros de 16 bits (cosa que seguramente ya imaginabas), pero no dispuestos consecutivamente, como podría pensarse:

```
Segmento:      Desplazamiento
                      :
xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx <- Dirección de 32 bits
```

De esta manera -que, repito, no es la que se usa realmente- se podrían recorrer 2^32 posiciones de memoria. Con el registro llamado de segmento apuntamos al "bloque", esto es, las líneas de mayor peso, y con el de desplazamiento nos movemos dentro de 64k. El problema reside en que si nuestra estructura de datos se encuentra al final de uno de estos bloques (un vector, por ejemplo), y queremos recorrerla linealmente, lo podremos hacer así solamente hasta llegar a la posición FFFFh del segmento apuntado, pues para seguir avanzando es necesario incrementar el registro de segmento en 1 llegados a ese punto (y continuar ya normalmente incrementando el desplazamiento desde 0000h). Resulta que, para más inri, una característica más que deseable -por no decir imprescindible- de un programa es que sea reubicable, es decir, que lo podamos cargar en cualquier zona de la memoria, principalmente porque el sistema operativo nos lo coloca donde puede, y cada vez en un sitio distinto. No hace falta pensar mucho para ver que, por este camino, mal vamos.

Intel dispuso los registros de la siguiente manera:

```
        Segmento:          xxxxxxxxxxxxxxxx0000
                         +
Desplazamiento (u offset) 0000xxxxxxxxxxxxxxx
                          ---------------------------------
   Dirección de 20 bits   xxxxxxxxxxxxxxxxxxxx
```

Así pues para obtener la dirección real se multiplica el registro de segmento por 16, y se suma al de desplazamiento (esto lógicamente no lo tenemos que hacer nosotros, hay un circuitillo por ahí para eso) Existen además parejas segmento-offset que apuntan a la misma posición de memoria (no podría ser de otro modo, si estamos recorriendo 2^20 posiciones con 32 bits). De la manera propuesta ahora, cuando queremos manejar una estructura de no más de 64k (esto nos lo limitan los registros que son de 16 bits, con lo que no hay más vuelta de hoja), podremos apuntar con el registro de desplazamiento al bloque donde lo ubicaremos (alineado en un múltiplo de 16 bytes, claro) y nos moveremos por dentro de este segmento alterando el offset.

Recordemos además que todo procesador de esta familia que funcione en modo real (o virtual) se comporta como un 8086 más o menos rápido en cuanto a direccionamiento de memoria, por lo que bajo MSDOS siempre estaremos limitados de esta manera.

De la suma de segmento y offset puede suceder que, para segmentos altos, exista un bit de carry a 1. Los 286 y superiores pueden aprovechar esto para acceder a la llamada "memoria alta", 64k situados por encima de los 2^20=1024k iniciales. No será nuestro caso, pues nos limitaremos a manejar la memoria a la vieja usanza. De hecho únicamente poblaremos los primeros 640k (la famosa memoria convencional) pues en las posiciones de los segmentos A000h y superiores se mapea la memoria de vídeo, la expandida... Pero bueno, de momento con 640k vamos sobrados.

Ahora casi todos los registros cobran ya sentido; los registros de segmento apuntan al segmento, y los de índice indican el desplazamiento dentro del segmento. Los registros de índice se pueden usar como registros de datos sin problemas para sumas, movimiento de datos... no así los de segmento, que tienen fuertes limitaciones. Cuando se diga "un registro" como operando de una instrucción, eso incluye en principio cualquier registro menos los de segmento.

El par CS:IP indica la dirección de memoria donde está la instrucción que se ejecuta. Los nombres vienen de Code Segment e Instruction Pointer. A esta parejita sólo la modifican las instrucciones de salto, así que podemos olvidarnos -un poco- de ella. Cada vez que se ejecuta una instrucción, el contador IP se incrementa para apuntar al código de la operación siguiente; las instrucciones de salto cargan CS:IP con la dirección adonde queremos saltar, para que se ejecute el código a partir de ahí.

SS:SP apuntan a la pila. Como en todo micro que se precie, tenemos una pila adonde echar los valores de los registros que queremos preservar para luego, las direcciones de retorno de las subrutinas.. La pila crece hacia abajo, es decir, cuando metemos algo en la pila el puntero de pila se decrementa, y cuando lo sacamos se incrementa. Siempre se meten valores de 16 bits. Significan Stack Segment y Stack Pointer, claro. Por lo general SS no se toca, y SP quizá pero poquito, ya que hay instrucciones específicas para manejar la pila que alteran SP indirectamente. Digamos que uno dice "tiro esto a la pila" o "saco lo primero que haya en la pila y lo meto aquí" y el micro modifica SP en consecuencia. BP es un puntero Base, para indicar también desplazamiento, que se usa en algunos modos de direccionamiento y especialmente cuando se manejan subrutinas. La pila se estudiará en mayor profundidad en el tema VI.

DS y ES son registros de segmento adicionales, el primero llamado de datos (Data) y el segundo Extra. Con ellos apuntaremos a los segmentos donde tengamos nuestros datos (ya que del código se encarga CS), esto es, nuestras variables. Estos los manipularemos mucho más que los anteriores cuando programemos en modo real.

DI y SI son registros índice, es decir, sirven para indicar el offset dentro de un segmento. Acabarás harto de tanto usarlos. En las instrucciones de cadena DI se asocia por defecto a DS, y SI a ES.

Aunque el sistema de segmentación pueda parecer muy engorroso (bueno, es que a este nivel realmente lo es, y mucho) en realidad es vital. El modo protegido del 386+ (386+ por "386 y superiores") emplea segmentación, pero aprovechando 32 bits. En entornos multitarea hay que estructurar cada programa con su(s) bloque(s) de código y de memoria, cada uno con su nivel de acceso (por ejemplo si estás guarreando con la memoria te interesa que sólo enredes con la de tu programa, para no sobreescribir otros procesos en curso por accidente, o parte del sistema operativo incluso); lo que sucede es que ahora es una mierda porque esta segmentación es muy limitada. Sirva esto para evitar la cantinela segmentación=mala que le puede haber pasado a uno por la cabeza a estas alturas de película. Tras este pequeño paréntesis, sigamos.

El registro de flags está formado por varios bits cada uno con significado propio, que son modificados por las operaciones que realizamos:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Flag	--	--	--	--	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF

Aunque es un registro de 16 bits sólo algunos de ellos tienen significado. Los otros adquieren valores indeterminados, y se dice que son bits reservados; un programa no debe tocarlos, pues aunque un equipo dado puede que no los use, otros micros podrían manipularlos internamente, con resultados impredecibles.

- **CF** Carry Flag o indicador de acarreo. Normalmente indica si "nos llevamos algo" despues de haber sumado o restado.
- **OF** Overflow Flag o indicador de desbordamiento. Si después de una operación el resultado no cabe en el tamaño del registro, se pone a 1.
- **ZF** Zero Flag o indicador de cero. Si el resultado da 0 se pone a 1.
- **SF** Sign Flag o indicador de signo. Si el resultado es negativo se pone a 1.
- **PF** Parity Flag o indicador de paridad. Con algunas operaciones lógicas o aritméticas se pone a 1 si el resultado es par.

- **AF** Auxiliary Flag o indicador auxiliar. Se usa para operaciones BCD (si es que éstas valen para algo)
- **DF** Direction Flag o indicador de dirección. Se usa en operaciones llamadas "de cadena", indicando el sentido (ascendente o descendente) en que se recorre la memoria. (Este flag es más divertido de lo que parece.)
- **IF** Interrupt Flag o indicador de interrupciones. Cuando se pone a 1 se permiten las interrupciones, a 0 se ignoran; se dice que se enmascaran. Hay no obstante algunas muy especiales que pasan del flag; por ejemplo, si arrancas (físicamente, me refiero; coger con la mano y zás!) una tarjeta de RAM con el trasto encendido, la interrupción que salta -alerta roja, inmersión, inmersión, auuuuu- pasa de este flag olímpicamente (algo así no se puede pasar por alto bajo ninguna circunstancia).
- **TF** Trap Flag o indicador de trampa (para ejecución paso a paso, como en los depuradores; podemos olvidarnos de él)

No conviene aturullarse demasiado a cuenta de los flags, pues a medida que se vayan viendo las instrucciones que los usen se irán comentando con más calma. Resumen: es un registro que contiene 1) indicadores de los resultados de algunas operaciones y 2) modificadores del comportamiento del procesador.

- Bueno, vale, tengo un 386

Y con 386 incluyo el 486, Pentium, K5, K6.. Salvando las pijaditas añadidas, funcionan todos más o menos igual (a nivel de código máquina, claro).

Ahora los registros de datos son EAX,EBX,ECX,EDX funcionando de manera idéntica a los anteriores, sólo que de 32 bits. Si queremos acceder a la parte baja de 16 bits, nos referiremos normalmente a AX,BX.. y si queremos hacer lo propio con las dos partes que componen la parte baja de 16 bits, hablaremos de AH,AL,BH.. todo exactamente igual. Se añaden además registros de segmento adicionales (de 16 bits, como los otros), FS y GS. De igual manera se extienden a 32 bits BP (a EBP), DI (a EDI) y SI (a ESI). Llamando a unos u otros modificaremos los 16 bits inferiores, o los 32 bits del registro completo. Cuando trabajemos en modo real, los segmentos de 64k nos limitarán el valor de offset a 65535; esto significa que si nos empeñamos en usar los registros extendidos en MSDOS, tendremos que asegurarnos de que la parte alta (los 16 bits superiores) esté a 0.

Registros de datos	EAX	EBX	ECX	EDX		
Punteros de pila	ESP	EBP				
Registros índice	EDI	ESI				
Registros de segmento	CS	DS	ES	FS	GS	SS
Registro de flags						

En modo protegido (windows/unix) los registros que indiquen offset podrán tomar cualquier valor, siempre que el par segmento:offset apunte a la zona de nuestro programa. De este modo con un mismo registro de segmento podremos, teóricamente, modificando solamente el offset, recorrer 4Gb de memoria respecto a la dirección base del segmento.

Además, y esto es especialmente importante, la dirección efectiva de CS:EIP, por ejemplo, ya no se calcula como 16*CS+EIP. Ahora el registro de segmento no indica una posición de memoria múltiplo de 16, sino el índice dentro de una tabla (conocida como tabla de descriptores de segmento) donde está contenida toda la información de los segmentos, como cuánto ocupan, quién tiene permiso de acceso, qué tipo de permiso... La dirección base de ese segmento, será una dirección de, posiblemente, 32 bits, que a nosotros de momento no nos incumbe. El sistema te ubicará el segmento de código (y con segmento me refiero a ese bloque de memoria con sus permisos, etc) en una posición cualquiera de memoria y le asignará un

número correspondiente a un elemento de esa tabla, que guardará en CS. Cuando nos queramos referir a él usaremos el valor de CS con que nuestro programa ha arrancado. Lo mismo sucederá para los datos y la pila. Como los segmentos pueden ser de hasta 4Gb, código, datos y pila irán cada uno en un segmento, de modo que nuestro programa no tendrá que modificar los registros de segmento para nada. No es así en MSDOS, que siendo de 64k, es frecuente tener que repartir el código y los datos entre varios segmentos.

CAPITULO III. EL MOVIMIENTO SE DEMUESTRA MOVIENDO.

La instrucción MOV y los modos de direccionamiento.

He aquí nuestra primera instrucción:

MOV destino,origen

Efectivamente, sirve para mover. Lo que hace es copiar lo que haya en "origen" en "destino". Lo de que primero vaya el destino y luego el origen es común a todas las instrucciones del 8086 que tengan dos operandos, lo cual creará más de un quebradero de cabeza al principio. Existen no obstante algunos ensambladores que en su sintaxis intercambian los operandos, pues realmente no es necesario emplear los mnemónicos en ese orden mientras el código de operación sea el correcto. No hace falta decir (bueno, por si acaso lo digo) que destino y origen tienen que tener el mismo tamaño; así

MOV AX,BL

haría pitar al ensamblador como loco, y con toda la razón. Si intentamos copiar BL (parte baja de BX, de tamaño 8 bits) en AX (registro de 16 bits), el ensamblador nos dirá que eso no se puede, y que no existe ningún código de operación para eso.

MOV AX,BX sin embargo hace que el procesador coja el contenido de BX y lo copiara en AX; lo que había anteriormente en AX se pierde (puesto que un registro al fin y al cabo es un número, en este caso de 16 bits, y ahora le hemos asignado un nuevo valor), mientras que BX no se ve afectado. Cuando decimos "mover" en realidad sería más apropiado "copiar", porque no alteramos en absoluto el operando origen. Esta instrucción no sólo nos permite mover datos entre registros, sino que podemos mover valores concretos, especificados en la propia instrucción. Por ejemplo, si quisiéramos poner a 0 el registro DX podríamos poner

MOV DX,0

muy distinto de

MOV DX,[0]

Los corchetes significan "lo que haya en la dirección..". En este caso el micro cogería la palabra (16 bits, pues es el tamaño del operando destino, así que queda implícito) que estuviera en la dirección 0 y la copiaría en DX. Más aún. No se trata de la dirección 0 sino del offset 0; ¿de qué segmento? DS, que es el registro de segmento por defecto. Cuando en una operación de este tipo no especificamos segmento, el procesador coge el valor que haya en DS y lo usa como segmento para formar la dirección completa. Si quisiéramos copiar a DX la primera pareja de bytes del segmento apuntado por ES, porque es allí donde tenemos el dato, tendríamos que poner un prefijo de segmento (o segment override, para los guiris):

MOV DX,[ES:0]

Esto va al segmento ES, se desplaza 0 bytes, coge los primeros 2 bytes a partir de esa dirección y los guarda en DX. ¿Complicado? Pues no hemos hecho más que empezar. Si en este momento andas un poco perdido, échale otro vistazo al capítulo anterior y lee esto otra vez con más calma, porque vamos a seguir añadiendo cosas.

Se llama "modo de direccionamiento" a una forma de especificarle una dirección al micro para que acceda a algún dato, como cuando hemos dicho MOV DX,[ES:0]. Decirle al procesador directamente la dirección, en este caso offset 0, es efectivamente un modo de direccionamiento, aunque no demasiado flexible, porque debemos saber la posición exacta del dato en el momento de hacer el programa. Veamos todos los modos de direccionamiento que permite este micro, sin más que poner el offset dentro de []:

Nombre	Offset	Segmento por defecto
Absoluto	Valor inmediato	DS
Indirecto con base	BX+x	DS
	BP+x	SS
Indirecto con índice	DI+x	DS
	SI+x	DS
Ind. con base e índice	BX+DI+x	DS
	BX+SI+x	DS
	BP+DI+x	SS
	BP+SI+x	SS

Por x queremos decir un número en complemento a dos de 8 o 16 bits, es decir, los comprendidos entre -128 y 127 o -32768 y 32767.

Supongamos que tenemos una cadena de caracteres en el segmento indicado por ES, offset 100h, y queremos mover un carácter cuya posición viene indicada por el registro BP, a AL. El offset del carácter sería BP+100h; como el segmento por defecto para ese modo de direccionamiento es SS, es necesario un prefijo de segmento. La instrucción sería:

MOV AL,[ES:BP+100h]

Observando un poco la tabla podemos darnos cuenta de que todos emplean por defecto el registro de segmento DS excepto los que usan BP, que se refieren a SS. En general no es buena idea usar prefijos de segmento, pues las instrucciones que los usan se codifican en más bytes y por tanto son más lentas. Así si hemos de referirnos a DS usaremos otros registros distintos de BP siempre que sea posible.

El llamado prefijo de segmento es estrictamente hablando un prefijo, pues se codifica como tal, precediendo a la instrucción a la que afecta (un byte extra). En NASM es posible seguir literalmente esta construcción, pudiendo escribir la expresión anterior como

ES
MOV AL,[BP+100h]

Comprendidos bien los modos de direccionamiento del 8086, voy a añadir algunos más: los que permiten los 386+. Cuando se emplean en direccionamientos indirectos registros de 32 bits es posible usar cualquiera de ellos. Así "MOV EAX,[ECX]" sería perfectamente válido. Y más aún (y esto es muy importante para manipular arrays de elementos mayores que un byte), el registro "índice" puede ir multiplicado por 2,4 u 8 si de desea: es posible realizar operaciones del tipo "MOV CL,[EBX+EDX*8]". Al que todo esto le parezca pequeños detallitos en vez de potentes modos de direccionamiento, que se dedique a la calceta porque esto no es lo suyo; para manejar vectores de reales es una ayuda importantísima.

No hay que olvidar que aunque estemos usando registros de 32 bits, seguimos limitados a segmentos de 64k si programamos en MSDOS. Bajo esta circunstancia, cuando se emplee un direccionamiento de este tipo hay que asegurarse de que la dirección efectiva (es decir, el resultado de la operación EBX+EDX*8, o lo que sea) no supera el valor 0FFFFh.

La instrucción MOV tiene ciertas limitaciones. No admite cualquier pareja de operandos. Sin embargo esto obedece a reglas muy sencillas:

1. No se puede mover de memoria a memoria
2. No se pueden cargar registros de segmento en direccionamiento inmediato

3. No se puede mover a CS

Las dos primeras reglas obligan a pasar por un registro intermedio de datos en caso de tener que hacer esas operaciones. La tercera es bastante obvia, pues si se pudiera hacer eso estaríamos haciendo que el contador de instrucción apuntase a sabe Dios dónde.

Otra regla fundamental de otro tipo -y que también da sus dolores de cabeza- es que cuando se mueven datos de o hacia memoria se sigue la "ordenación Intel", que no es más que una manera de tocar las narices: los datos se guardan al revés. Me explico. Si yo hiciera

```
MOV AX,1234h
MOV [DS:0],AX
```

uno podría pensar que ahora la posición DS:0 contiene 12h y DS:1 34h. Pues no. Es exactamente al revés. Cuando se almacena algo en memoria, se copia a la posición señalada la parte baja y luego, en la posición siguiente, la parte alta. Lo gracioso del asunto (y más que nada porque si no fuera así Intel tocaría bastante más abajo de las narices) es que cuando se trae un dato de memoria a registros se repite la jugada, de tal manera que al repetir el movimiento en sentido contrario, tenga en el registro el mismo dato que tenía al principio. Pero la cosa no se detiene ahí. Tras

```
MOV EAX,12345678h
MOV [ES:0124h],EAX
```

la memoria contendría algo así como:

Segmento	Offset:	Contenido:
ES	0124h	78h
	0125h	56h
	0126h	34h
	0127h	12h

Vamos, que lo que digo para palabras lo digo para palabras dobles. ¿Divertido, eh?

Uno se puede preguntar además: ¿por qué no hacer "MOV [ES:0124h],12345678h"? Se puede, claro, pero no así (bueno, en este caso concreto tal vez, pero como norma general, no). El ensamblador no puede saber el tamaño que tiene el operando inmediato, así que no sabe cuantos bytes tiene que escribir. Si tú haces "MOV AX,8", está bien claro que tiene que meter un "8" en 16 bits, porque es el tamaño que tiene AX, pero cuando el destino es una posición de memoria, no sabe si poner 08h, 0008h, 00000008h.. Hay que especificar si es un byte, un word, o double word con lo que se conocen como typecast:

```
MOV BYTE [DI],0h ;DI es un puntero a byte
MOV WORD [DI],0 ; puntero a word
MOV DWORD [DI],0h ; puntero a double word
```

Cada una de ellas pone a cero 1,2 ó 4 bytes a partir de la dirección apuntada por DS:DI. Una última nota a cuento de estas cosas. Si se especifica por ejemplo "MOV AX,[DS:0]" la instrucción se codificará sin el prefijo de segmento, pues no hace falta al ser el registro por defecto. Esto es común a todas las instrucciones que empleen este modo de direccionamiento, y aunque es raro indicar al ensamblador direcciones de memoria mediante valores inmediatos, conviene tenerlo en cuenta: especificar un segmento que ya es el segmento por defecto para ese direccionamiento, no altera la codificación final de la instrucción.

Y eso es todo en cuanto a direccionamientos. En mi humilde opinión esto forma parte de lo más difícil del ensamblador, porque aunque, efectivamente, uno puede comprender cada ejemplo individual, tener una visión general de todo lo que se puede hacer, qué usar en cada caso, y qué significa, ya no queda tan claro. Si dominas esto, el concepto de segmentación, y cómo funciona la pila (que veremos dentro de nada), no tendrás ningún problema para asimilar todo lo demás.

CAPÍTULO IV: Programas. Ejecutables en MSDOS y Linux

En este tema se dan algunas pistas de cómo se estructuran los fuentes para generar ejecutables en entorno MSDOS y Linux; aunque es un poco temprano para programar nada, y hay un apartado especial para las directivas (es fundamental conocer las facilidades que da el ensamblador, y no sólo las instrucciones del procesador), creo que es bueno saber ensamblar algo desde el principio para así poder ir probando todas las cosas que se vayan viendo de aquí en adelante.

EJECUTABLES EN MSDOS

- PROGRAMAS DE TIPO COM

Es el tipo básico de ejecutable en MSDOS. Toda la información del programa está contenida en un mismo segmento, lo que supone que un programa de este tipo está limitado a 64k de tamaño máximo.

Vamos con un sencillo ejemplo de fuente que genera un ejecutable de tipo COM:

```
        org 100h

section .text
        ;Aquí va el código
        ;Cuando se hace int 21h se llama al sistema operativo, que ejecuta la función que
indique AH
        ;AH=9 imprime la cadena apuntada por DS:DX, terminada en "$"
        ;AH=4Ch termina el programa, con código de salida AL

        mov dx,cadena
        mov ah,9
        int 21h

        mov ax,0x4C00
        int 21h

section .data
        ;Aqui los datos inicializados
        ;los caracteres 10 y 13 significan retorno de línea en MS-DOS.
        cadena DB "Qué pasa, mundo",13,10,"$"

section .bss
        ;Y aqui los datos no inicializados, si los hubiera
```

Aunque se definen diferentes secciones, las tres se encontrarán en el mismo segmento. Cuando el programa sea ejecutado, el sistema operativo lo copiará en memoria en la posición que determine conveniente, y en el momento en que se le transfiera el control CS,DS,ES y SS apuntarán a dicho segmento (el resto de registros de segmento no se tocan porque puede ser que el procesador no los tenga, no olvidemos que es un sistema de 16 bits). El programa se encontrará a partir del offset 100h, pues los primeros 256 bytes los usa el sistema operativo para almacenar información del programa relativa a su ejecución, en lo que se conoce como PSP (Program Segment Prefix). Por ello es necesario incluir la directiva "org 100h" en la cabecera para indicar al ensamblador que calcule las direcciones relativas a ese desplazamiento.

Aunque el PSP en general puede resultar de cierta utilidad para el programador, no se pretende entrar a discutir la programación bajo MSDOS (se muestra para aprendizaje de programación en modo real); sin embargo puede ser interesante saber que en el offset 80h se encuentra la cadena que se introdujo en el intérprete de comandos para ejecutar el programa. El primer byte indica la longitud de la cadena, y a continuación los caracteres que la componen.

Tras la cadena aparece un byte con el código 13d, que no cuenta a la hora de contabilizar la longitud en el primer byte.

Las diferentes secciones del fuente no son realmente importantes, pues en un ejecutable de tipo COM no se distingue código de datos en tanto que simplemente comienza a ejecutar desde la posición CS:100h lo que encuentre a su paso. Lo que sucede es que el ensamblador colocará siempre el contenido de la sección .text en primer lugar, .data a continuación y por último .bss. De este modo podemos hacer más legible el código declarando la zona de datos en primer lugar. No hace falta definir la pila pues el sistema coloca SP al final del segmento, en la dirección 0FFFEh. Significa esto que el ejecutable tendrá que ocupar realmente menos de 64k, pues a medida que vayamos metiendo cosas en la pila, iremos comiendo terreno al propio programa.

La diferencia entre datos inicializados y no inicializados es muy simple. Los no inicializados no forman parte del programa; si reservamos espacio para un array de 1000 bytes en .bss el programa no ocupará 1000 bytes más. El ensamblador se limitará a utilizar los nombres de los datos que se definan como si hubiera algo ahí, pero cuando se cargue el programa esa zona de memoria quedará como estaba. Es por eso que primero se colocan en memoria los datos inicializados y luego los no inicializados. Estrictamente hablando, estos últimos no existen.

Un programa como éste se dice que es binario puro, pues absolutamente todo el código representa instrucciones o datos del programa. No es necesario enlazador, pues todo el programa va "de una pieza"; el ensamblador se limita a traducir los códigos de operación/directivas/datos/etiquetas que va encontrando a su paso, generando directamente el ejecutable.

Para ensamblar este programa, suponiendo que hola.asm sea el código fuente y hola.com la salida deseada, tendremos que escribir lo siguiente:

```
nasm -f bin hola.asm -o hola.com
```

La opción -f indica el tipo de salida, en este caso binaria.

- PROGRAMAS DE TIPO EXE

Era evidente que con ejecutables de 64k no íbamos a ninguna parte. El formato más común en MSDOS es EXE, pues permite generar código desparramado por diferentes segmentos, aprovechando así (de una manera un tanto picassiana) el espacio de memoria disponible. Hay referencias entre segmentos no resueltas en el momento de enlazado, por lo que el sistema operativo en el momento de cargarlo en memoria ha de realizar algunos ajustes en el código en función de los datos que encuentre en la cabecera. Como la estructura es más compleja (mejor dicho, como existe una estructura) el ensamblador ha de generar unos archivos objeto que le serán pasados al enlazador para finalmente formar el ejecutable. Por otra parte NASM proporciona un conjunto de macros para crear un fuente que, cuando es ensamblado tal cual, resulta un archivo EXE. Yo prefiero la primera opción, porque cuando quieres compilar distintos módulos no hay más remedio que enlazarlo. En la documentación de NASM figura el método alternativo.

Veamos un ejemplo de fuente para generar un EXE:

```
segment codigo
..start:
        ;Aqui va el codigo

        mov ax,datos
        mov ds,ax

        mov dx,cadena
```

```
        mov ah,9
        int 21h

        mov ax,0x4C00
        int 21h

segment datos
        ;Aqui los datos inicializados

        cadena DB "Hey, mundo",13,10,"$"

section pila stack
        ;Una pila de 256 bytes

        resb 256
```

Cuando se ejecuta un programa EXE, CS apunta al segmento de código de inicio, SS al segmento de pila (con SP al final del mismo), y DS/ES al PSP, que ahora no estará necesariamente ubicado antes del programa.

En este caso para obtener el archivo objeto el comando será

```
nasm -f obj hola.asm -o hola.obj
```

Ahora podremos generar el ejecutable con cualquier linker; por ejemplo, con tlink sería

```
tlink hola.obj
```

consiguiendo así un hola.exe

Como DS apunta inicialmente al PSP, nada más empezar lo cargamos con el valor del segmento de datos. El ensamblador interpreta la palabra "datos" como el segmento datos, referencia que en el caso de los archivos EXE (y casi todos los demás) no se resuelve hasta el momento de cargar el programa en memoria, en el que el sistema operativo coloca los segmentos que lo componen. No es así en el caso de los offsets (mov dx, cadena; el ensamblador entiende que se hace referencia al offset de la etiqueta "cadena" dentro del segmento donde se encuentre), que son perfectamente conocidos en tiempo de ensamblado.

La directiva ..start: indica al enlazador el punto de inicio del programa. stack señala al enlazador, de manera similar, cuál es el segmento de pila; al arrancar el programa, SS apuntará al segmento designado, y SP al final de éste.

EJECUTABLES EN LINUX

En realidad casi todos los sistemas que funcionan en modo protegido tienen básicamente el mismo tipo de ejecutables; todo programa se estructura en tres segmentos, para pila, código y datos. Los segmentos están limitados a un máximo de 4Gb, lo cual no es una condición en absoluto restrictiva. Con un segmento para cada cosa vamos servidos.

Del mismo modo que en MSDOS, y como es lógico, en el momento de ejecutar un programa éste es ubicado en una posición de memoria cualquiera, tomando los registros de segmento los valores que el sistema operativo determine convenientes. La peculiaridad estriba en que si quisiéramos modificar nosotros mismos los registros de segmento, sólo podrían apuntar a los segmentos de nuestro propio programa (impidiéndonos el sistema operativo hacerlo de otro modo), con lo que a efectos prácticos podremos olvidarnos de los segmentos a la hora de programar y limitarnos a manejar los registros de offset, de 32 bits (dónde están los segmentos o qué significan los registros de segmento es algo que, de momento, no nos atañe). Además, como sólo hay un segmento de datos, y el de código no se puede modificar (por cuestiones de

seguridad), en general serán innecesarios los prefijos de segmento en los direccionamientos. Esto simplifica mucho las cosas al programador, pues desde su programa ve su memoria, que recorre con registros de 32 bits.

Un fuente de este tipo se construye de manera similar a los COM (sólo que las secciones de datos y código corresponden a segmentos distintos, y la pila la determina el sistema operativo), aunque, obviamente, habremos de emplear un enlazador distinto. Como en Linux las llamadas al sistema son completamente distintas (la diferencia más evidente es que se acude a la INT 80h en lugar de la 21h), incluyo a continuación la versión del ejemplo debidamente modificado:

```
section .text
global _start
_start:
	;Aqui va el codigo

	;escribe (EAX=4) en la salida estándar (EBX=1) la cadena apuntada por ECX de longitud
EDX.
	mov edx,longitud
	mov ecx,cadena
	mov ebx,1
	mov eax,4
	int 80h

	;terminar la ejecución (EAX=1)
	mov eax,1
	int 80h

section .data
	;Aqui los datos inicializados

	cadena DB "Qué pasa, mundo",10,"$"
	longitud equ $-cadena
```

Para ensamblarlo como ejecutable para linux (generando el archivo objeto.o) y enlazar el resultado para crear el archivo ejecutable, nada más que

```
nasm -f elf archivo.asm -o objeto.o

ld -s -o ejecutable objeto.o
```

Con ./ejecutable veremos el mensaje "Qué pasa, mundo" en pantalla. En este caso la manera de definir el punto de entrada del programa para el enlazador es emplear la directiva global _start, pues es la etiqueta que ld espera. Con esto lo que hacemos es permitir que la etiqueta _start sea visible desde el exterior.

CAPITULO V. DIRECTIVAS Y MÁS DIRECTIVAS.

Conocer las directivas que acepta nuestro ensamblador simplifica mucho el trabajo, pero a fin de no hacer esto soporífero (aunque creo que ya es demasiado tarde) sólo incluiré las típicas. También hay que tener en cuenta que esta parte puede cambiar bastante de un ensamblador a otro. NASM presenta una sintaxis semejante (aunque con importantes diferencias) a TASM o MASM, ensambladores de Borland y Microsoft respectivamente, para MSDOS/Windows. GASM (GNU Assembler, también con un nutrido grupo de adeptos), por otra parte, emplea una notación bien distinta (y en muchos casos odiada por los que no la usamos), conocida como AT&T. Con esto digo que aunque las instrucciones sean las mismas, puede cambiar mucho la manera de escribirlas.

Y quizá sea esta la parte más ingrata del ensamblador, pero bueno, la ventaja es que muy fácil, y no hace falta conocer todas las directivas para realizar un código eficiente.. Simplemente nos harán la vida más fácil. Empecemos.

Cada línea de ensamblador tiene (salvando las excepciones de las macros y cosas similares) la misma pinta:

etiqueta: instruccion operandos ;comentario

Todos los campos son optativos (menos, evidentemente, cuando la instrucción requiera operandos); así pues es posible definir etiquetas en cualquier sitio, o simplemente colocar un comentario. Si la línea fuese demasiado larga y quisiéramos partirla en varias (cosa un tanto rara, pero que viene bien saber), se puede usar el carácter '\'. Si la línea termina en \, se colocará la siguiente a continuación.

El ensamblador admite operadores de multiplicación, suma, resta, paréntesis.. Cuando toca ensamblar, él solito opera y sustituye. Por ejemplo:

MOV AX,2*(2+4)

es exactamente lo mismo que

MOV AX,12

- Definición de datos

Las directivas básicas para definir datos son las siguientes:

- DB Define Byte, 1 byte
- DW Define Word, 2 bytes
- DD Define Double Word (4 bytes)
- DQ Define Quad Word (8 bytes)
- DT Define Ten Bytes (10 bytes; aunque parezca muy raro, cuando veas el coprocesador no podrás vivir sin ella)

Ejemplos:

```
db 'Soy una cadena' ;es cierto, es una cadena
dw 12,1             ;0C00h 0100h (ordenación Intel, no se te olvide)
dt 3.14159265359    ;representación en coma flotante de 10 bytes PI
```

Lo de definir números reales se puede hacer con DD (precisión simple, 32 bits), DQ (precisión doble, 64 bits) y DT (a veces llamado real temporal, una precisión de la leche, 80 bits; uno para

el signo, 64 para la mantisa y 15 para el exponente o_O'). Los caracteres ascii van entrecomillados con "; cuando el ensamblador encuentre un caracter entre comillas simples lo sustituye por el valor numérico del código ascii correspondiente.

Para reservar zonas amplias de memoria se emplean RESB y familia, que lo que hacen es dejar hueco para N bytes, palabras (RESW), palabras dobles (RESD), palabras cuádruples (RESQ) o bloques de 10 bytes (REST).

```
resb 10             ;deja un hueco de 10 bytes sin inicializar
resq 4*4 DUP (1) ;espacio para una matriz 4x4 (16 reales de precisión doble)
```

Una directiva muy peculiar, de uso muy esporádico pero muy interesante, es INCBIN; lo que hace es incluir dentro del código el contenido binario de un archivo dado. Puede ser útil, por ejemplo, para incrustar pequeñas imágenes o logotipos dentro del ejecutable:

```
incbin "datos.dat"            ;ensambla introduciendo datos.dat en este punto
incbin "datos.dat",100        ;incluye datos.dat, excepto los primeros 100 bytes
incbin "datos.dat",100,200 ;incluye los 200 bytes siguientes a los 100 primeros de datos.dat
```

- ORGORG es una directiva con la que hay que tener especial cuidado, pues en otros ensambladores puede tener un efecto distinto. En nasm solamente influye en los offsets de las etiquetas e instrucciones que se refieran a direcciones. Es decir, colocar un org 100h no genera un hueco de 256 bytes donde empieza a colocarnos nuestro programa; simplemente hace que las referencias a offsets se realicen según este desplazamiento inicial. Si ensamblamos el programa "COM" del tema anterior, vemos que no ocupa 256 bytes + el programa propiamente dicho; simplemente el programa está preparado para que cuando se copie en memoria a partir de un offset de 256 bytes las direcciones sean correctas.

El uso es tan simple como org numero_de_bytes

- *EQUEquivalencia. El ensamblador sustituye en todo el fuente la palabra indicada por la expresión que sigue a equ. Muy apropiada cuando tenemos que usar una constante en varios puntos del código, pues nos ahorra buscar las referencias por todo el fuente cada vez que cambiemos de idea y la asignemos otro valor.*

```
Siglo equ 21
MOV AX,Siglo ;AX ahora vale 21
```

- *Procesadores/Coprocesadores: 'CPU'La directiva CPU determina el tipo de procesador que vamos a emplear. En general no será importante pues todos compaten el juego de instrucciones básico, pero si quisiéramos programar en un procesador más limitado (pongamos un 286), y por un casual, intentásemos usar una instrucción mínimamente moderna (digamos alguna MMX), el ensamblador nos daría un error. Algunas CPU's (se puede especificar más):*

```
cpu 8086
cpu 186
cpu 286
cpu 386
cpu 486
cpu 586
cpu 686
cpu IA64
```

Si no se selecciona ninguna, por defecto se admiten todas las instrucciones.

- *%include archivoSustituye la directiva por el contenido del archivo (a la manera del #include de c)*
- *%defineDefinición de una macro de una sola línea. Cuando el ensamblador encuentra una referencia a una macro, la sustituye por su defínción, incluyendo los argumentos si los hubiera. Por ejemplo*

```
%define h2(a,b) ((a*a)+(b*b))
mov ax,h2(2,3)
```

es equivalente a

```
mov ax,(2*2)+(3*3)
```

es decir

```
mov ax,13
```

- *%macroDefinición de una macro. El funcionamiento es similar, sólo que para fragmentos de código:*

```
%macro nom_macro num_arg
.....
%endmacro
```

En los ejemplos del tema anterior vimos las llamadas al sistema para mostrar una cadena en pantalla. Si tuviéramos que repetir esa llamada varias veces, sería más cómodo y legible usar una macro como la siguiente:

```
%macro imprime 2
      mov edx,%1
      mov ecx,%2
      mov ebx,1
      mov eax,4
      int 80h
%endmacro
```

con lo que cada vez que cada vez que quisiéramos imprimir una cadena sólo tendríamos que escribir

```
imprime longitud,cadena
```

Aunque es tentador hacerlo, no hay que abusar de las macros porque el código engorda que da gusto. Las macros son más rápidas que las subrutinas (pues ahorran dos saltos, uno de ida y uno de vuelta, más los tejemanejes de la pila), pero a menudo no compensa por muy bonitas que parezcan: el que la pantalla de bienvenida de un programa cargue un par de milisegundos antes o después no se nota. Claro que ahora la memoria es cada vez menos problema... En cualquier caso, su uso es deseable siempre que se favorezca la legibilidad del código.

Una aplicación que da idea de la potencia de las macros es la de emplear macros con el nombre de instrucciones. Es así posible definir una macro llamada push que reciba dos argumentos, por ejemplo, y que ejecute las instrucciones push %1 y push %2. Cuando se escriba push eax, al recibir un número de argumentos distintos a los de la definición de la macro, sabrá que se refiere a una instrucción (aunque dará un warning por ello), mientras que al escribir push eax,ebx desarrollará la macro como push eax seguido de push ebx.

Para especificar que una etiqueta dentro de la macro es local (a fin de que no se duplique la etiqueta al usar dos veces la misma macro), hay que poner %% justo delante de la etiqueta.

```
%macro macroboba 0
        %%EtiquetaInutil:
%endmacro
```

Si llamamos a esta macro varias veces, cada vez que se la llame usará un nombre distinto para la etiqueta (pero igual en toda la macro, claro).

...y eso es todo en cuanto a directivas. Hay muchas más. Existen deficiones de condicionales para que ensamble unas partes de código u otras en función de variables que definamos con EQU, comparadores, operadores lógicos.. Son de uso muy sencillo todas ellas, por lo que basta con echar un ojo de vez en cuando a la documentación del ensamblador para tenerlas presentes.

CAPÍTULO VI: JUEGO DE INSTRUCCIONES DEL 8086

¡Mamá, ya sé sumar!

A continuación voy a describir brevemente todas las instrucciones que admite un 8086; entre paréntesis señalaré, si toca, los flags que altera cada una. Recordemos que el registro de flags contiene alguos indicadores que se ponen a "0" o a "1" según el resultado de la última operación realizada (se obtiene un número negativo, ha habido carry, overflow..) Con las instrucciones se mostrará qué flags son modificados por este motivo, pero no se garantiza que el resto de flags "informativos" no se verán alterados por la ejecución de la instrucción (o sea, el valor de ciertos flags puede quedar indeterminado tras la operación). Para detalles escabrosos sobre el funcionamiento de ciertas instrucciones, es mejor consultar una tabla de referencia en condiciones, como las que tiene Intel por ahí en pdf en su página web, o la que viene con el manual de NASM.

Aunque el título dice "del 8086", entremezclo algunas peculiaridades de los procesadores superiores; en la mayoría de los casos quedará claro si sirven o no en un 8086 (si usan EAX, por ejemplo, es evidente que no). Una diferencia fundamental del 8086 con sus descendientes es que algunas de estas mismas instrucciones pueden recibir, además de los operandos habituales, otros adicionales propios de los nuevos procesadores. Me explico. Uno en un 386 puede hacer ADD AX,BX (AX=AX+BX) como un 8086 cualquiera, pero además, por contar con registros de 32 bits, podrá hacer cosas del estilo ADD EAX,EBX. En ambos casos se tratará de la misma instrucción (ADD, sumar), sólo que si el procesador cuenta con más registros, o registros extendidos, o simplemente capacidad de admitir nuevos operandos, podremos usarlos. Para esta instrucción en particular la sintaxis se describirá como ADD destino, origen, pero por destino y origen no será necesariamente válido cualquier modo de direccionamiento.. Dependerá de la instrucción, y en cierta medida del procesador que se use. En la mayoría de los casos imperarán las reglas generales "no es válido operar de memoria a memoria" y "no es válido operar con registros de segmento". Hay además una cierta lógica en estas restricciones (el operando destino generalmente es un registro de propósito general), e instrucciones similares entre sí aceptarán casi con toda seguridad los mismos operandos. Se harán algunas observaciones sobre los operandos válidos para cada instrucción, pero de nuevo para cualquier duda lo mejor será echar mano de una tabla "oficial" (que es lo que hacemos todos, no te creas que alguien se lo aprende de memoria).

Los procesadores posteriores al 8086 incluyen no sólo mejoras en las instrucciones, sino instrucciones adicionales. De éstas sólo se indicarán, un tanto desperdigadas por ahí, las que puedan ser utilidad. Las demás no digo que sean inútiles, sino que en la mayoría de los casos no nos estarán permitidas. Hay un conjunto de instrucciones exclusivas del modo protegido que sólo podrán ser usadas por el sistema operativo, y se dice que son instrucciones privilegiadas; un programa de usuario que intente ejecutar una operación de este tipo será detenido, generalmente con un mensaje que indique un error de protección general. Casi todas las instrucciones privilegiadas corresponden a operaciones que pueden afectar al funcionamiento del resto de procesos del sistema (en modo protegido, por lo general, tendremos varios programas funcionando al tiempo, aunque en un momento dado el usuario intente cargar "el suyo"). Un ejemplo es INVD, que invalida la información de la memoria cache interna. Si no fuera ésta una instrucción privilegiada, cualquiera podría, en cualquier momento, desbaratar la cache con un programa malintencionado (recordemos que puede suceder que haya varios usuarios trabajando al tiempo en un cierto ordenador). Las instrucciones IN/OUT son usadas ya por el 8086 (de hecho son fundamentales), pero no se incluyen en este capítulo porque sin ser propias del modo protegido, cuando el micro funciona en este modo,se consideran instrucciones privilegiadas. A ellas me referiré en otro capítulo. Finalmente, los conjuntos especiales de instrucciones (coprocesador matemático, MMX, SSE..) se comentarán en capítulos aparte.

La discusión más delicada tiene lugar cuando diferenciamos un procesador funcionando "en 16 bits" (modo real, que se comporta como un 8086 aunque no lo sea), y "en 32 bits" (modo protegido, cualquier 386+ en Linux, Windows, etc), porque a menudo crea confusión. Cuando estamos ejecutando un programa para MSDOS (aunque sea desde Windows, ya que estará emulando un entorno de MSDOS), nuestro procesador dispondrá de, tal vez, más

registros, pero como ya hemos visto seguirá obteniendo las direcciones como segmento*16+offset (aunque pueda realizar operaciones con los registros extra, o incluso direccionar la memoria con ellos en esta circunstancia). Existen instrucciones que ya no según el procesador sino según el modo de funcionamiento se comportan de un modo u otro; ése es el caso de LOOP, que en modo real altera CX y en modo protegido ECX (normalmente prefiero decir "32 bits" a "modo protegido", porque existen condiciones excepcionales bajo las cuales se puede ejecutar código de 16 bits en modo protegido; no entraré de momento en esta discusión). En modo 32 bits, además, los offsets son de este tamaño, lo que significa que los punteros de pila, instrucción, etc, serán de tipo dword. Importantísimo tenerlo presente al pasar argumentos a subrutinas a través de la pila (daré mucho la brasa con esto a lo largo del tutorial).

- INSTRUCCIONES DE MOVIMIENTO DE DATOS

- MOV destino, origen (MOVe, mover)

Bueno, qué voy a contar de ésta que no haya dicho ya.. Vale, sí, un pequeño detalle. Cuando se carga SS con MOV, el micro inhibe las interrupciones hasta después de ejecutar la siguiente instrucción. Aún no he dicho lo que son las interrupciones ni qué tiene de especial la pila así que tardaremos en comprobar qué puede tener de útil este detalle, y la verdad es que normalmente nos dará lo mismo, pero bueno, por si las moscas.

- XCHG destino, origen (eXCHanGe, intercambiar)

Intercambia destino con origen; no se puede usar con registros de segmento. Esta instrucción encuentra su utilidad en aquellas instrucciones que tienen asociados registros específicos.

- OPERACIONES ARITMETICAS

- ADD destino, origen (ADDition, sumar) {O,S,Z,A,P,C}

Suma origen y destino, guardando el resultado en destino. Si al sumar los dos últimos bits "me llevo una" el bit de carry se pone a 1 (y si no, a 0).

- ADC destino,origen (ADdition with Carry, sumar con acarreo) {O,S,Z,A,P,C}

Suma origen, destino y el bit de carry, guardando el resultado en destino. Sirve entre otras cosas para sumar números de más de 16 bits arrastrando el bit de carry de una suma a otra. Si quisiéramos sumar dos números enteros de 64 bits almacenados en EAX-EBX y ECX-EDX, podríamos sumarlos con ADD EBX,EDX primero y ADC EAX,ECX después (para sumar las partes altas de 32 bits con "la que nos llevábamos" de las partes bajas). Sumar con ADC puede generar a su vez carry, con lo que teóricamente podríamos sumar números enteros de cualquier tamaño, propagando el carry de una suma a otra.

Podemos poner a 1 el flag de carry directamente mediante STC (SeT Carry) o a 0 mediante CLC (CLear Carry), ambas instrucciones sin argumentos.

- INC destino (INCrement, incrementar) {O,S,Z,A,P}

Incrementa el operando destino en 1. Puede ser un operando de tamaño "byte" o superior, tanto un registro como una posición de memoria; en este último caso hay que especificar tamaño con WORD, DWORD etc, tal y como explicamos en el capítulo III con los modos de direccionamiento. Esta instrucción no modifica el bit de carry; si quieres detectar cuándo "te pasas" al incrementar un contador, usa el flag "Zero" o el de signo.

- SUB destino, origen (SUBstract, resta) {O,S,Z,A,P,C}

Resta a destino lo que haya en origen.

- SBB destino, origen (SuBstract with Borrow, restar con llevada) {O,S,Z,A,P,C}

Resta a destino lo que haya en origen. Si el flag C=1 resta uno más. Análoga a ADC.

- DEC destino (DECrement, decrementar) {O,S,Z,A,P}

Igual que INC, pero que resta 1 en vez de sumarlo.

- IMUL origen (Integer MULtiplication, multiplicación entera con signo) {O,C}

Multiplica origen, entero con signo (en complemento a dos), de longitud byte o word, por AL o AX respectivamente. Si origen es un byte el resultado se guarda en AX; si es tamaño word se almacena en el par DX-AX (parte alta en DX, parte baja en AX). Si las mitades de mayor peso son distintas de 0, sea cual sea el signo, CF y OF se ponen a uno. En el caso del 386+, además, se puede usar un operando origen de 32 bits. Si es así se multiplica entonces por EAX, dejando el resultado de 64 bits en el par EDX-EAX. El operando debe ser un registro o un dato de memoria (nada de valores inmediatos). Esto se aplica para IMUL, MUL, IDIV y DIV.

IMUL tiene otras dos formas más para procesadores posteriores al 8086. La primera es IMUL destino, origen donde el destino es un registro de propósito general y el origen un valor inmediato, otro registro o una posición de memoria. La segunda forma tiene el aspecto IMUL destino, origen1, origen2. Destino es nuevamente un registro de propósito general, origen1 un registro o posición de memoria, y origen2 un valor inmediato. Lo que hace es almacenar el producto de los dos operandos origen en destino. En cualquiera de las dos formas, si el resultado del producto no cabe en el destino, queda truncado. Nos toca a nosotros comprobar, como siempre que sea necesario, los bits de overflow y carry (si se usa un único operando no hace falta, porque el destino siempre cabe).

- MUL origen (MULtiplication, multiplicación entera sin signo) {O,C}

Como IMUL, salvo que multiplica enteros sin signo. Sólo admite la forma con un único operando.

- IDIV origen (Integer DIVide, división entera con signo)

Divide números con signo. Calcula el cociente y el resto de dividir AX entre el operando (tamaño byte). Si el operando es de 16 bits lo que divide es el par DX-AX. Si el operando es de 32 bits (80386+), lo que divide es el par EDX-EAX. El cociente lo guarda en AL, AX o EAX según el caso. El resto en AH, DX o EDX. Si el cociente es mayor que el tamaño máximo (8, 16 o 32 bits) tanto cociente como resto quedan indefinidos, y salta una interrupción 0 (luego veremos cómo van estas cosas, pero te puedes hacer una idea de que no es muy sano). Si divides por cero pasa lo mismo.

- DIV origen (DIVide, división entera sin signo)

Igual que IDIV, sólo que para números sin signo.

- INSTRUCCIONES DE SOPORTE ARITMÉTICO

La mayor parte de estas instrucciones sirven para extender el signo. Ya hemos visto que puede suceder que tengamos que operar con un número repartido en dos registros como DX-AX. Imagina que quieres cargar un entero con signo de 16 bits en estos dos registros: ¿qué haces con los otros 16 más altos? Si el entero es positivo basta con poner a 0 DX, pero si es negativo tendrás que darle el valor 0FFFFh:

1d = 0000 0001h
-1d = FFFF FFFFh

Como todo lo que sea abreviar comparaciones y saltos es siempre bienvenido, a alguien se le ocurrió hacer instrucciones especialmente para esta tarea. Puedes extender el signo dentro de un registro o hacia otro, trabajando fundamentalmente sobre el acumulador (por lo que no son demasiado flexibles). Las MOVSX y MOVZX -del 386- darán más juego.

- CWD (Convert Word to Doubleword, convertir palabra a palabra doble)

Extiende el signo de AX a DX, resultando el número DX-AX

- CQD (Convert Doubleword to Quad-word, convertir palabra doble a palabra cuádruple)

Extiende el signo de EAX a EDX, resultando el número EDX-EAX

- CBW (Convert Byte to Word, convertir byte a palabra)

Extiende el signo de AL a AH, resultando el número AX

- CWDE (Convert Word to Doubleword Extended, convertir palabra a palabra doble extendida)

Extiende el signo de AX a EAX, resultando el número EAX

- MOVSX destino,origen (Move with Sign Extension, mover con extensión de signo)

Mueve origen a destino, extendiendo el signo. Si el destino es de 16 bits, origen ha de ser de 8. Si destino es de 32 bits, origen puede ser de 8 o de 16. Sólo acepta mover a un registro (de datos).

- MOVZX destino,origen (Move with Zero Extension, mover con extensión de ceros)

Exactamente como MOVZX, sólo que en vez de extender el signo rellena de 0s.

- NEG destino (NEGate, negar){O,S,Z,A,P,C}

Cambia de signo el número en complemento a dos del destino. Equivale a hacer NOT y luego INC.

-INSTRUCCIONES LÓGICAS

- AND destino,origen
- OR destino,origen
- XOR destino,origen
- NOT destino

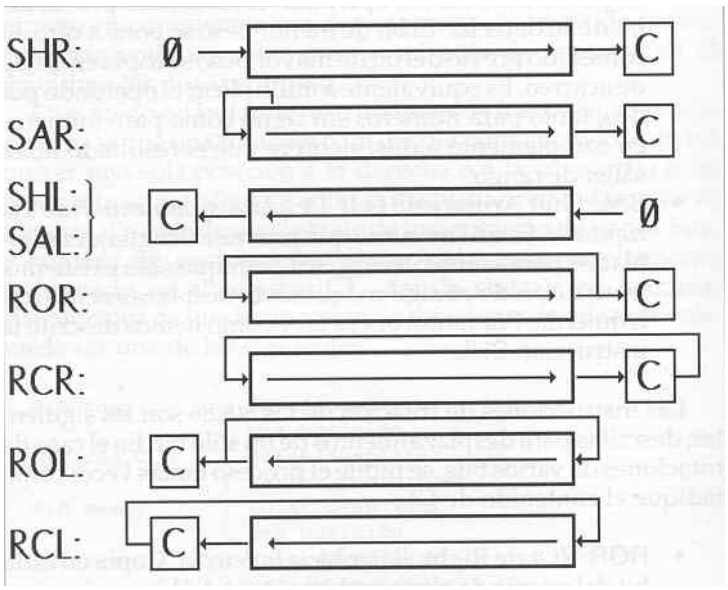
Creo que si sabes lo que significan estas operaciones lógicas (si no revisa el capítulo 0), estarán bien claritas :)

-DESPLAZAMIENTOS Y ROTACIONES

- SAL destino,origen (Shift Arithmetic Left, desplazamiento aritmético a la izquierda) {O,S,Z,P,C}
- SAR destino,origen (Shift Arithmetic Right, desplazamiento aritmético a la derecha) {O,S,Z,P,C}

- SHL destino,origen (SHift logic Left, desplazamiento lógico a la izquierda) {O,S,Z,P,C}
- SHR destino,origen (SHift logic Right, desplazamiento lógico a la derecha) {O,S,Z,P,C}
- ROL destino,origen (ROtate Left, rotación a la izquierda) {O,C}
- ROR destino,origen (ROtate Right, rotación a la derecha) {O,C}
- RCL destino,origen (Rotate with Carry Left, rotación a la izquierda con carry) {O,C}
- RCR destino,origen (Rotate with Carry Right, rotación a la derecha con carry) {O,C}

Ambas desplazan o rotan el operando destino hacia el sentido indicado tantas veces como diga el operando origen. Este último operando puede ser un valor inmediato o CL.



Es bueno recordar que desplazar un número a la izquierda un bit equivale a multiplicar por dos, y desplazarlo a la derecha dividir entre dos. Cuando queramos realizar alguna de estas operaciones en un factor potencia de dos siempre será mucho más rápido desplazar un registro (o varios, propagando el carry) que realizar una multiplicación/división. Hilando un poco más fino, si tenemos que multiplicar un número por algo que no sea una potencia de dos pero "se le parezca mucho", el método puede seguir compensando. Para multiplicar por 20 podemos hacer una copia del registro, desplazarlo por un lado 4 bits (para multiplicar por 16) y por otro 2 bits (para multiplicar por 4), y sumar ambos resultados parciales. Si nuestro procesador está un poco pasado de moda (386-486) puede que la mejora sea aún significativa.

-INSTRUCCIONES DE COMPARACIÓN

- CMP operando1,operando2 (CoMPare, comparar) {O,S,Z,A,P,C}

Funciona exactamente igual que SUB solamente que sin almacenar el resultado (o sea, efectua la operación operando1-operando2, alterando los flags en consecuencia). Al igual que la instrucción CMP del motorola se utiliza antes de efectuar un salto condicional.

- TEST operando1,operando2 (TEST, comprobar) {O,S,Z,A,P,C}

Como la anterior, pero con la operación AND en lugar de SUB.

-INSTRUCCIONES DE SALTO

- JMP dirección (JuMP, saltar)

Salta a la dirección indicada. Este salto puede ser tanto lejano como cercano, y la dirección puede venir dada en valor inmediato (generalmente mediante una etiqueta) o en memoria. Un salto cercano tiene lugar dentro del mismo segmento (llamado también salto intrasegmento por este motivo), cargando IP (EIP en modo protegido) con el nuevo offset. Los saltos lejanos cambiar el valor tanto de IP(EIP) como de CS.

El tipo de salto se puede indicar con la dirección mediante los prefijos near o far en cada caso. Existe un tercer tipo de salto denominado corto indicado por el prefijo short, cuya dirección viene codificada en un byte. Este último salto es en realidad una forma abreviada de expresar un salto cercano, donde en lugar de indicar la dirección a la que se salta, se especifica la distancia desde la posición actual como un número de 8 bits en complemento a dos.

- Jcc dirección

"cc" representa la condición. Si la condición es verdadera, salta. En caso contrario, continúa la ejecución. Las distintas condiciones se presentan en la tabla inferior. El ensamblador de Intel tiene algunos mnemónicos distintos para las mismas instrucciones, lo cual viene muy bien para recordar qué comparación usar en cada caso. Por ejemplo, JZ (saltar si cero) y JE (saltar si iguales) son exactamente la misma instrucción. Una curiosidad: en el 8086 los saltos condicionales sólo podían ser cortos (lo que significa que la distancia se codificaba en un byte), obligando a utilizar la comparación contraria y un JMP para poder saltar a más de 127/128 bytes de distancia. En el 80386 estos saltos son de hasta 32 bits, con lo que prácticamente no tenemos limitación alguna. Si queremos saltar cuando el bit de carry sea 1, podemos hacerlo con algo como JC etiqueta y olvidarnos de todo lo demás.

Instrucción			Condición
JZ	Jump if Zero	salta si cero	ZF=1
JNZ	Jump if Not Zero	salta si no cero	ZF=0
JC	Jump if Carry	salta si acarreo	CF=1
JNC	Jump if Not Carry	salta si no acarreo	CF=0
JO	Jump if Overflow	salta si overflow	OF=1
JNO	Jump if Not Overflow	salta si no overflow	OF=0
JS	Jump if Sign	salta si signo	SF=1
JNO	Jump if Not Sign	salta si no signo	SF=0
JP/JPE	Jump if Parity (Parity Even)	salta si paridad (Paridad Par)	PF=1
JNP/JPO	Jump if Not Parity (Parity Odd)	salta si no paridad (Paridad Par)	PF=0

Cuando queremos hacer un salto condicionado por una comparación, y no directamente por el estado de los flags, lo que hacemos es una comparación CMP A,B. A continuación usamos una instrucción de salto de entre las siguientes:

Instrucción			Condición
JA	Jump if Above	salta si por encima	A&gtB (sin signo)
JAE	Jump if Above or Equal	salta si por encima o igual	A>=B (sin signo)
JB	Jump if Below	salta si por debajo	A&ltB (sin signo)
JBE	Jump if Below or Equal	salta si por debajo o igual	A<=B (sin signo)
JE	Jump if Equal	salta si igual	A=B
JG	Jump if Greater	salta si mayor	A&gtB (con signo)
JGE	Jump if Greater or Equal	salta si mayor o igual	A>=B (con signo)
JL	Jump if Less	salta si menor	A&ltB (con signo)

JLE	Jump if Less or Equal	salta si menor o igual	A<=B (con signo)

CMP equivalía a una resta que no modificaba los operandos; si queremos saltar si A<B, podemos hacer CMP A,B. Si A<B al hacer la resta me llevaré una, por lo que el bit de carry se pondrá a 1. Vemos que JC es equivalente a JB. Con similares deducciones se pueden obtener algunos saltos más; otros, sin embargo, dependen del resultado de dos flags al tiempo ("menor o igual" saltará con CF=1 y ZF=1), por lo que sus mnemónicos corresponderán a instrucciones nuevas.

Para poner las cosas aún más fáciles, existen JNA,JNAE,JNB,JNBE,JNE,JNG, JNGE,JNL, JNLE. No hacía falta que estuvieran, porque son equivalentes a JBE,JB,JAE,JA,JNZ, JLE,JL y JG, pero así nos evitan pensar un poco. Una N delante indica la condición contraria. Si uno no quiere líos, basta que recuerde que Below/Above son para números sin signo y Less/Greater para con signo.

Una instrucción de salto muy útil en combinación con las siguientes (LOOP y variaciones) es JCXZ (Jump if CX is Zero, salta si CX es cero), o JECXZ si estamos operando en 32 bits.

Para hacer ciclos (algo así como for i=1 to 10) la familia del 8086 cuenta con LOOP y la pareja LOOPE/LOOPZ (mnemónicos de lo mismo)

- LOOP dirección (LOOP, ciclo)

Decrementa CX y si el resultado es distinto de 0 salta a la dirección dada. Estrictamente hablando no es una dirección sino un desplazamiento en el rango +127/-128 bytes respecto a IP para el 8086, o +(2^31-1)/(-2^31) para el 80386, igual que las instrucciones Jcc. Al programar no usaremos desplazamientos ni direcciones absolutas, sino etiquetas que se encargará de sustituir el ensamblador, por lo que no debe (en principio) pasar de ser una anécdota.

LOOPZ/LOOPE tienen la misma sintaxis que LOOP. Lo que hacen es decrementar CX y saltar a la dirección dada si CX es distinto de 0 y ZF=1. La idea es efectuar un ciclo con una condición dada, que se repita un máximo de CX veces. Cuando el modo de funcionamiento es de 32 bits LOOP, LOOPE y LOOPZ operan sobre ECX. LOOPNZ/LOOPNE son iguales a los anteriores, sólo que la condición adicional al salto es ZF=0 en vez de ZF=1.

Como cuando CX=0 (ECX=0) estos ciclos se ejecutan en principio 2^16 (2^32) veces, podemos evitarlo mediante el uso de JCXZ (JECXZ) justo antes del ciclo.

-MANEJO DE LA PILA

Mencionamos ya brevemente en el capítulo II en qué consiste la pila. Ahora veremos exactamente qué es, así como para qué y cómo se usa.

Los programas se dividen básicamente en una zona de datos y otra de código, habitualmente cada cada una en su región de memoria asociada a un valor de registro de segmento (lo que se llama, simplemente, un segmento). Siempre hay al menos un tercer segmento llamado pila, de suma utilidad en los microprocesadores. En él se almacenan valores temporales como las variables locales de las funciones, o las direcciones de retorno de éstas. Una función no es más que una subrutina, o un fragmento de código al que se le llama generalmente varias veces desde el programa principal, o desde una función jerárquicamente superior. Cuando se llama a una función se hace un mero salto al punto donde empieza ese código. Sin embargo esa subrutina puede ser llamada desde distintos puntos del programa principal, por lo que hay que almacenar en algún sitio la dirección desde donde se hace la llamada, cada vez que esa llamada tiene lugar, para que al finalizar la ejecución de la función se retome el programa donde se dejó. Esta dirección puede almacenarse en un sitio fijo (como hacen algunos microcontroladores), pero eso tiene el inconveniente de que si esa función a su vez llama a otra

función (¡o a sí misma!) podemos sobreescribir la dirección de retorno anterior, y al regresar de la segunda llamada, no podríamos volver desde la primera. Además, es deseable que la función guarde los valores de todos los registros que vaya a usar en algún sitio, para que el que la llame no tenga que preocuparse de ello (pues si sabe que los registros van a ser modificados, pero no sabe cuáles, los guardará todos por si acaso). Todas estas cosas, y algunas más, se hacen con la pila.

El segmento de pila está indicado por SS, y el desplazamiento dentro del segmento, por SP (o ESP si estamos en 32 bits; todo lo que se diga de SP de aquí en adelante será aplicable a ESP, salvo, como se suele decir, error u omisión).

Cuando arranca el programa, SP apunta al final del segmento de pila. En el caso de los programas de tipo EXE en MSDOS, por ejemplo, el tamaño que se reserva a la pila viene indicado en el ejecutable (que solicita al sistema operativo ese hueco; si no hay memoria suficiente para el código, los datos y la pila, devuelve mensaje de memoria insuficiente y no lo ejecuta). En otros, puede que el tamaño lo asigne el sistema operativo según su conveniencia. El caso es que la manera de meter algo en la pila es decrementar SP para que apunte un poco más arriba y copiarlo a esa posición de memoria, SS:SP. Para sacarlo, copiamos lo que haya en SS:SP a nuestro destino, e incrementamos el puntero.

Como con todo lo que se hace con frecuencia, hay dispuestas instrucciones propias para el manejo de la pila. Las dos básicas son PUSH origen (empujar) y POP destino (sacar). La primera decrementa el puntero de pila y copia a la dirección apuntada por él (SS:SP) el operando origen (de tamaño múltiplo de 16 bits), mientras que la segunda almacena el contenido de la pila (elemento apuntado por SS:SP) en destino y altera el puntero en consecuencia. Si el operando es de 16 bits se modifica en 2 unidades, de 32 en 4, etc. Lo que se incrementa/decrementa es siempre SP, claro, porque SS nos indica dónde está ubicado el segmento de pila.

PUSHA y POPA (de PUSH All y POP All) almacenan en la pila o extraen de la pila respectivamente los registros básicos. Para PUSHA se sigue este orden (el inverso para POPA): AX,CX,DX,BX,SP,BP,SI,DI. En el caso de los registros extendidos lo que tenemos son PUSHAD y POPAD, empujando a la pila entonces EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI. El puntero de pila que se introduce en la misma es el que hay antes de empujar AX/EAX. Cuando se ejecuta POPA el contenido de SP/ESP de la pila no se escribe, sino que "se tira". Se emplean generalmente al principio y final de una subrutina, para preservar el estado del micro.

No creo que nadie vaya a programar exactamente en un 386, pero si alguien quisiera tener en cuenta a estos procesadores tendría que considerar que la mayoría tienen un defecto de diseño por el cual POPAD no restaura correctamente el valor de EAX. Para evitarlo basta colocar la instrucción NOP detrás de aquélla.

PUSHF y POPF (PUSH Flags y POP Flags) empujan a la pila o recuperan de ella el registro de flags.

-MANEJO DE SUBRUTINAS

- CALL dirección (CALL, llamar)

Empuja a la pila la dirección de retorno (la de la siguiente instrucción) y salta a la dirección dada. Como en los saltos: una llamada cercana cambia el offset (IP/EIP), mientras que una lejana cambia offset y segmento (CS:IP/CS:EIP). Sólo empuja a la pila los registros que modifica con el salto (se habla de punteros cercanos o lejanos en cada caso). El tipo de salto se indica con un prefijo a la dirección, como con JMP: NEAR para cercano, FAR para lejano.

En vez de la dirección admite también un operando en memoria con cualquier modo de direccionamiento indirecto, cargando entonces IP/EIP o CS:IP/CS:EIP con el puntero

almacenado en esa posición de memoria. También se puede hacer un salto cercano al valor almacenado en un registro (de 16 bits o 32 bits según el caso). Para las direcciones de retorno que se guardan en la pila, o los punteros en memoria que se cargan, hay que recordar que en caso de ser un puntero lejano primero se almacena el offset y luego el segmento. A ver, para no liarla, vamos con algunos ejemplos de llamadas:

call near etiqueta; salta a la dirección donde se encuentre la etiqueta
call far [di]; salta a la dirección almacenada en la posición [ds:di] como segmento:offset
call ebx; salta al offset ebx (se sobreentiende que es near)

Si se omite far, se da por sentado que es una llamada cercana, pero por claridad recomiendo ponerlo siempre explícitamente.

- RET,IRET (RETurn, regresar)

Extrae una dirección de la pila y salta a ella. Puede ser un retorno cercano mediante RETN (Near) o lejano mediante RETF (Far); en el primer caso extrae el offset y en el segundo segmento y offset. Con el uso genérico de la instrucción RET el ensamblador elige la apropiada; sin embargo es buena costumbre indicarlo igual que con call. Es muy importante que la subrutina deje la pila como estaba justo al ser llamada para que la dirección de retorno quede en la parte superior, pues de lo contrario al llegar a RET saltaríamos a cualquier sitio menos la dirección correcta. IRET es el retorno de una interrupción; además de volver, restaura el registro de flags. Al acudir a una interrupción este registro se guarda en la pila. De momento no trataremos interrupciones.

La pila sirve no sólo para guardar los registros temporalmente o las direcciones de retorno, sino también las variables locales, que son imprescindibles para una programación cabal.

La idea es que cuando vamos a llamar a una subrutina, echamos primero a la pila los argumentos que va a necesitar. Luego la subrutina, nada más arrancar salva en la pila los valores de los registros que considera necesarios, y "abre un hueco" encima de éstos, decrementando el puntero de pila en tantas unidades como necesite. La ventaja fundamental de esto es que cada llamada a la función "crea" su propio espacio para variables locales, de modo que una función puede llamarse a sí misma sin crear conflictos entre las variables de la función "llamadora" y "llamada" (caller y callee en algunos textos ingleses, aunque no estoy muy seguro de que el segundo término sea muy correcto). El único problema que tenemos ahora es que una vez abierto el hueco y determinadas las posiciones que ocuparán dentro de la pila nuestras variables, puede suceder que queramos echar más cosas a la pila, con lo que SP/ESP se desplazará arriba y abajo durante la ejecución de la subrutina; llevar la cuenta de en dónde se encuentra cada variable a lo largo de la función es algo poco práctico, y con lo que es fácil equivocarse. Sucede aquí que tenemos un registro que usa por defecto el segmento de pila: BP (nuevamente se ha de entender que lo que se aplica a SP,BP es extensible a ESP,EBP). Éste es el registro que emplearemos como puntero inamovible para señalar a argumentos y variables locales, para que no perdamos la pista de estos datos hagamos lo que hagamos con SP. Primero lo haremos todo paso a paso porque es importante entender el procedimiento:

Vamos a crear una subrutina en modo real que sume dos números de 16 bits (un poco estúpido, pero es para hacerlo lo más simple posible). El programa principal haría algo así:

```
...
push word  [Sumando1]
push word  [Sumando2]
sub sp,2   ;hueco para el resultado (queremos que nos lo devuelva en la pila)
call SumaEstupida
pop word  [Resultado]            ;saca el resultado
add sp,4   ;elimina los sumandos de la pila
...
```

y la subrutina sería:

```
SumaEstupida:
        ;SP apunta a la dirección de retorno, SP+2 al resultado, SP+4 a Sumando2, SP+6 a
Sumando1
        push bp         ;ahora SP+4 apunta al resultado, SP+6 a Sumando2 y SP+8 a
Sumando1
        mov bp,sp               ;BP+4 apunta al resultado, BP+6 a Sumando2 y BP+8
a Sumando1
        push ax         ;altero SP pero me da igual, porque tengo BP como puntero a
los argumentos
        mov ax,[bp+6]   ;cargo el Sumando2
        add ax,[bp+8]   ;le añado el Sumando1
        mov [bp+4],ax   ;almaceno el resultado
        pop ax          ;restauro AX
        pop bp          ;restauro BP
        retn
```

Mediante la directiva EQU podemos hacer por ejemplo SUM1=BP+8 tal que para referirnos al sumando 1 hagamos [SUM1], y así con el resto. RET admite un operando inmediato que indica el número de bytes que queremos desplazar el puntero de pila al regresar. De esta manera, si colocamos primero el hueco para el resultado y luego los sumandos, mediante un "RETN 4" podemos ahorrarnos "ADD SP,4" tras la llamada. Algunos lenguajes de alto nivel hacen las llamadas a funciones de esta manera; no así el C. Usar un ADD en lugar de un "RET x" tiene la ventaja de que podemos usar un número variable de argumentos por pila; en general al que mezcle ensamblador con C le parecerá la mejor alternativa.

Los procesadores 286+ proporcionan un método alternativo de realizar estas operaciones tan comunes en los lenguajes de alto nivel con un cierto nivel de sofisticación. La primera instrucción al respecto es ENTER, que recibe dos operandos. El primero indica la cantidad de memoria que se reservará en la pila, en bytes (asignación de espacio a variables locales). El segundo es el llamado nivel de anidamiento del procedimiento, y hace referencia a la jerarquía que presentan las variables en el programa; determinará qué variables serán visibles para la función que ha sido llamada. Esto encierra bastantes sutilezas que corresponden a cómo se construye el código máquina de este tipo de estructuras y a este nivel no interesa meterse en berenjenales; si hubiera que usar ENTER, dejaríamos el segundo operando a 0. Quedémonos úicamente con que si quisiéramos reservar 4 bytes para variables locales, usaríamos la instrucción "ENTER 4,0"; empuja BP, copia SP en BP, decrementa SP en 4.

La instrucción que deshace el entuerto es LEAVE, sin argumentos. Lo único que hace es copiar BP en SP para liberar el espacio reservado, y a continuación restaurar BP desde la pila.

Para ver en más detalle el manejo de la pila en las subrutinas, así como las posibilidades de ENTER/LEAVE y el concepto de nivel de anidamiento de un procedimiento recomiendo acudir a la documentación Intel, al texto "Intel Architecture Software Developer's Manual Volume 1: Basic Architecture", que pese a lo que aparenta es un texto muy legible. Este tema lo aborda el capítulo 4, más concretamente en "Procedure calls for block-structured languages". Si alguien está pensando en mezclar C con ensamblador encontrará este texto de gran interés.

El programador de la subrutina ha de tener muy en cuenta dos cosas. La primera, si la subrutina va a ser llamada desde el mismo segmento u otro (llamada cercana o lejana), pues dependiendo de ello CALL introduce distinto número de elementos en la pila. La segunda es si estamos programando en 16 o en 32 bits, que nos determinará las dimensiones de lo que echemos a la pila. Cualquier consideración errónea de este tipo nos hará intentar acceder a los argumentos en posiciones equivocadas de la pila.

-INSTRUCCIONES PARA OBTENER DIRECCIONES

- LEA destino,origen (Load Effective Address, cargar dirección efectiva)

Carga la dirección efectiva del operando origen en destino. "LEA AX,[BX+DI+2]" calcularía la suma BX+DI+2 e introduciría el resultado en AX (y no el contenido de la dirección apuntada por BX+DI+2, pues eso sería un MOV). Como destino no se puede usar un registro de segmento. Si el destino es de 32 bits, el offset que se carga es de este tipo. En modo protegido sólo usaremos este último, pues offsets de 16 bits carecerán de sentido.

- LDS destino,origen (Load pointer using DS, cargar puntero usando DS)

Esta instrucción y sus variantes ahorran mucho tiempo e instrucciones en la carga de punteros. origen será siempre memoria conteniendo un puntero, es decir, un segmento y un desplazamiento. La primera palabra corresponde al offset y la segunda al segmento. El offset se carga en destino y el segmento en DS. Si estamos trabajando en un modo de 32 bits el desplazamiento será de 32 bits y el segmento de 16. Existen más instrucciones, una por cada registro de segmento: LES,LFS,LGS y LSS (mucho cuidadito con esta última, porque nos desmadra la pila).

-INSTRUCCIONES DE MANEJO DE BITS

- BT/BTS/BTR/BTC registro,operando (Bit Test/Test&Set/Test&Reset/Test&Complement

Estas instrucciones lo primero que hacen es copiar el bit del registro indicado por el operando, sobre el bit de carry. El operando puede ser un registro o un valor inmediato de 8 bits. Una vez hecho esto no hace nada (BT), lo pone a 1 (BTS), lo pone a 0 (BTR) o lo complementa (BTC) según corresponda.

-INSTRUCCIONES DE CADENA

Hay un conjunto de instrucciones conocido a veces como "de cadena", que sirven para mover y comparar elementos dispuestos en array, incrementándose cada vez el puntero a la cadena. Éstas se ven afectadas por el bit de dirección (que indica el sentido en que se recorre la cadena). Mediante la instrucción STD (SeT Direction flag) hacemos DF=1, y con CLD (CLear Direction flag) DF=0

LODSB y LODSW (LOaD String, Byte y LOad String, Word), sin operandos, leen el byte o palabra en la dirección dada por DS:SI(ESI) y la almacenan en AL o AX respectivamente. Podemos usar en su lugar LODS operando, con un operando en memoria, especificando con ello si se trata de un LODSB o LODSW según el tipo. Si el operando lleva segment override será de este segmento de donde se tomen los datos con SI. Por claridad es preferible usar LODSB o LODSW.

¿Qué tiene todo esto que ver con lo mencionado sobre el flag de dirección? Si el DF=0 SI se incrementa en uno para LODSB y dos para LODSW (apunta al siguiente byte/palabra). Si DF=1 se lee el array en dirección inversa, es decir, SI se decrementa. Así, por ejemplo, podemos ir cargando en el acumulador el contenido de un array a lo largo de un ciclo para operar con él.

STOSB y STOSW (STOre String, Byte/Word) funcionan con el mismo principio en cuanto al flag de dirección, y lo que hacen es almacenar en ES:DI(EDI) el contenido del acumulador (AL o AX según cada caso). De manera análoga podemos usar STOS operando.

MOVSB y MOVSW (MOV String, Byte/Word) van más allá; mueven el byte o palabra en DS:SI a ES:DI. Vemos ahora que SI es el Source Index o índice fuente, y DI el Destination Index o índice destino. Tras el movimiento de datos SI y DI son incrementados o decrementados siguiendo la lógica descrita para LODS. Es admisible además la instrucción MOVS destino,origen (ambos operandos en memoria). El ensamblador emplea los operandos para determinar si se trata de un MOVSB o MOVSW al codificar la instrucción correspondiente; además el operando origen puede llevar un prefijo de segmento (no así el destino) en cuyo caso se emplea ése registro de segmento como origen.

¡Ajajá, así que ahora podemos mover arrays de bytes o palabras metiendo estas instrucciones en un ciclo, olvidándonos de andar manipulando los registros índice! Pues no. Es decir, sí, claro que se puede, pero no se hace así. Existe un prefijo llamado REP que se coloca antes de la instrucción de cadena en cuestión, que lo que hace es repetir el proceso tantas veces como indique CX(ECX). Supongamos que queremos llenar de ceros 100 bytes a partir de la posición A000h:0000h. Podemos currarnos un ciclo que maneje un índice, empezar a hacer mov's de 0s a [di].. Una solución (en 16 bits) más inteligente y rápida sería:

```
mov cx,50d
mov ax,0A000h
mov es,ax
xor di,di
xor ax,ax
rep stosw
```

Como cada palabra son 2 bytes, lo que hacemos es cargar CX con 50 (pondremos a 0 50 palabras), apuntar con ES:DI al primer elemento, poner AX a 0, y empezar a almacenar AX en cada posición, hasta 50 veces. Al finalizar ese fragmento de código DI contendrá el valor 100, pues se ha incrementado en 2 en cada repetición del ciclo.

Pero como se suele decir, aún hay más. SCASB y SCASW (SCAn String) realizan la comparación "CMP AL,ES:[DI]" o "CMP AX,ES:[DI]", alterando lógicamente los flags, y a continuación modificando DI como es debido. (Existe SCAS operando con idénticas consideraciones a las anteriores instrucciones de cadena con operando en memoria).CMPSB y CMPSW (CoMPare String) equivalen a "CMP DS:[SI],ES:[DI]" tamaño byte o word según corresponda, alterando los flags en función de la comparación e incrementando SI y DI según el tamaño del dato y el flag de dirección (habiendo asimismo un CMPS que funciona análogamente a MOVS en cuanto a llevar operandos en memoria). Esto puede no parecer impresionante (ya que REP aquí no pinta nada), pero es que existen dos prefijos (en realidad 4, pero son parejas de mnemónicos de instrucciones idénticas)
más, REPE/REPZ y REPNE/REPNZ. Estos prefijos funcionan como REP en cuanto a que repiten la instrucción que preceden tantas veces como indique CX excepto en que ademásha de verificarse la condición que representan. REPZ se repite mientras el flag de cero esté a uno (REPeat while Zero), mientras que REPNZ se repite, lógicamente, mientras esté a cero (REPeat while Not Zero). De esta manera es muy fácil realizar un código que localice una determinada palabra dentro de un array, o que encuentre la diferencia entre dos zonas de memoria. Si antes de llegar CX a 0 se produce la condición contraria a la del prefijo, se sale del ciclo; DI/SI apuntarán al elemento siguiente al que provocó este efecto. Si se termina con CX=0 habrá que mirar el flag correspondiente para comprobar si en la última comprobación la condición se cumplió o no... pues en ambos casos el ciclo habrá terminado.

Los 386+ incluyen además STOSD,LODSD,MOVSD,SCASD,CMPSD. La "D" es de Double word, lo que significa que trabajan con palabras dobles, 32 bits. En lugar de AL/AX las operaciones tienen lugar con EAX; como todas las anteriores, en modo real usarán CX y SI/DI, y en modo protegido ECX y ESI/EDI. Todas ellas permiten prefijos REP.

-INSTRUCCIONES BCD

- AAA (ASCII Adjust AX After Addition) {A,C}

Convierte el número almacenado en AL a BCD desempaquetado. La idea es aplicarlo después de sumar BCDs no empaquetados. Esta instrucción mira los 4 bits más bajos de AL: si es mayor que 9 o AF (Flag Auxiliar) es igual a 1, suma 6 a AL, 1 a AH, hace AF=CF=1, y los cuatro bits más significativos de AL los deja a 0. ¿Lo qué?

Vamos con el ejemplillo de marras. Tenemos en AX el BCD no empaquetado 47 (0407h) y en BX 14 (0104h). Queremos sumarlos, y que el resultado siga siendo un BCD no empaquetado, para obtener un resultado coherente. Partimos de que AF=0. Primero sumamos con ADD AX,BX porque no sabemos hacer otra cosa, y el resultado que nos deja es AX=050Bh. Uf, ni

por el forro. ¿Qué hacemos? Aaa... Eso, la instrucción AAA. Como la parte baja de AL es mayor que 9, se da cuenta rápidamente de que ese número hay que ajustarlo (cosa que ya sabíamos, pero en fin). Aplica la receta: suma 6 a AL, y 1 a AH. AX entonces queda 0611h. Carga con ceros la parte alta de AH, AX=0601h. Vaya, esto ya está mejor. Así que con esto podemos hacer minisumas en BCD. Con los flags, movs y un poco de imaginación se pueden hacer sumas en BCD más grandes. Mi consejo es que, una vez entendido esto, te olvides de las instrucciones para BCD; el coprocesador matemático incluye instrucciones de conversión mucho menos enrevesadas (coges dos números enormes, los sumas, y guardas el resultado gordo directamente como BCD)

Lo de "ASCII" para este tipo de instrucciones con BCD no empaquetados viene de que se puede obtener fácilmente el código ASCII de un número de éstos: sólo hay que sumarle el código del cero (48) a cada dígito. Si estuviera empaquetado no podríamos, porque lo que tenemos es un nibble para cada dígito, y no un byte (y el ascii es un byte, claro, ya me dirás cómo le sumas 48 si no).

- DAA (Decimal Adjust AL after Addition) {S,Z,A,P,C}

Algo parecido a AAA, sólo que se usa tras la suma de dos bytes con dígitos BCD empaquetados (dos números por tanto de dos dígitos). En vez de sumar 6 a AH suma 6 al nibble alto de AL, etc. Para sumar dos números de dos dígitos BCD empaquetados, ADD AL,loquesea y luego DAA.

- AAS (Adjust AX After Subtraction) {A,C}

Como AAA pero para ajustar después de una resta, en vez de una suma.

- DAS (Decimal Adjust AL after Subtraction) {S,Z,A,P,C}

Análoga a DAA, pero para la resta.

- AAM (ASCII Adjust AX After Multiply) {S,Z,P}

De la misma calaña que AAA,AAS. Ajusta el resultado guardado en AX de multiplicar dos dígitos BCD no empaquetados. Por ejemplo, si AL=07h y BL=08h, tras MUL
BL y AAM tendremos AX=0506h (porque 7·8=56). Apasionante.

- AAD (ASCII Adjust AX Before Division) {S,Z,P}

Más de lo mismo. Pero ojo, que ahora AAD se aplica antes de dividir, y no después. Volviendo al ejemplo anterior, si con nuestros AX=0506h, BL=08h hacemos AAD y luego DIV
BL obtenemos.. ajá, AL=07h y BL=08h, lo que confirma nuestra teoría de que 7·8=56.

-<u>MISCELÁNEA</u>

- HLT (HaLT, parada)

Detiene el microprocesador hasta la llegada de una interrupción o de una señal de reset. Se encuentra entre las instrucciones denominadas privilegiadas.

- NOP (No OPeration, no hacer nada)

No hace nada más que consumir los ciclos de reloj necesarios para cargar la instrucción y procesarla. Corresponde a la codificación de XCHG AX,AX

- INT inmediato

Salta al código de la interrupción indicada por el operando inmediato (0-255). Aunque hay un capítulo que explica más en detalle las interrupciones, se puede decir que una instrucción de este tipo realiza una llamada lejana a una subrutina determinada por un cierto número. En el caso de Linux, una interrupción 80h cede la ejecución al sistema operativo, para que realice alguna operación indicada a través de los valores de los registros (abrir un archivo, reservar memoria, etc). La diferencia fundamental con las subrutinas es que al estar numeradas no es necesario conocer la posición de memoria a la que se va a saltar. La característica "estrella" de estas llamadas cuando tienen lugar en modo protegido es que se produce un cambio de privilegio; en el caso citado, el sistema operativo contará con permisos para realizar determinadas operaciones que el usuario no puede hacer directamente. Por ejemplo, si queremos acceder al disco para borrar un archivo, no podemos. Lo que hacemos es cederle el control al sistema diciéndole que borre ese archivo, pues sí que tiene permiso para ese tipo de acceso (al fin y al cabo es quien controla el cotarro), y comprobará si ese archivo nos pertenece, para borrarlo.

INTO, sin operandos, es mucho menos usada. Genera una interrupción 4 si el bit de overflow está a 1. En caso contrario continúa con la instrucción siguiente, como cualquier salto condicional.

Me dejo algunas instrucciones que no usaremos ya no digamos normalmente, si no en el 99.999..% de los casos. LOCK bloquea el bus para reservarlo, allí donde pueda haber otros aparatos (como otros procesadores, si el ordenador los tiene) que puedan meterle mano. De este modo se garantiza el acceso; pone el cartel de "ocupado" por una determinada línea del bus, y hace lo que tiene que hacer sin ser molestado. Cada vez que queramos hacer algo de esto, colocaremos el prefijo "LOCK" delante de la instrucción que queramos "blindar". Tan pronto se ejecuta la instrucción, el bloqueo del bus se levanta (a menos que estemos todo el día dando la vara con el prefijo).

Relacionado con los coprocesadores tenemos ESC, que pasa a esos dispositivos operandos que puedan necesitar (para acceder a registros, por ejemplo). Aquí no pienso hacer nada explícitamente con esta instrucción (ya veremos lo que quiero decir con esto cuando lleguemos al coprocesador matemático), así que puedes olvidar este párrafo.

Para finalizar, la instrucción WAIT, que detiene el procesador hasta que le llegue una cierta señal. Se utiliza para sincronizar las operaciones entre el procesador y el coprocesador matemático, y también comprobaremos que no hace falta preocuparse de ella.

CAPÍTULO VIII: MI ORDENADOR TIENE MMX. Y QUÉ.

Todos los Pentium MMX y posteriores procesadores Intel, así como prácticamente todos los AMD desde su fecha cuentan con un juego de instrucciones con este nombre. Estas instrucciones trabajan con la filosofía SIMD (que como ya dijimos viene a ser aplicar la misma operación a varios datos), que suele ser de aplicación en imágenes, vídeo y cosas así. Para hacer uso de lo aprendido en este capítulo hará falta un ensamblador mínimamente actualizado (cosa no demasiado problemática si venimos dándole al NASM), y aunque es un tema de aplicación bastante específica, no es algo que se les de muy bien a los compiladores: muchos no lo usan, y otros no lo usan demasiado bien.

Este tipo de micros (Pentiums MMX y posteriores) cuentan con 8 registros MMX de 64 bits, que en realidad son un alias (bueno, esto pronto dejó de ser así, pero en principio es cierto) de los registros de la FPU. Quiere decir esto que cuando escribo un dato en un registro MMX en realidad lo estoy haciendo sobre un registro del coprocesador. Más exactamente, sobre la mantisa de un registro del coprocesador. Esto requiere que cada vez que se alternen operaciones de coma flotante con MMX, como cuando se cambia decontexto en un entorno multitarea, se salve el estado de los registros de un tipo y se restaure del otro. Por tanto, aunque es posible emplear instrucciones MMX y de la FPU en un mismo código, es poco recomendable en tanto que cambiar constantemente de unas a otras dará un programa muy ineficiente. Intel recomienda en estos casos algo que es tan de sentido común como efectuar la ejecución de estas instrucciones de manera separada en ciclos, subrutinas, etc.. vaciando completamente los registros de un tipo u otro en cada cambio (y no dejar por ejemplo un real en la pila de la FPU antes de entrar en una función MMX para luego esperar que ese número siga ahí). Si alternamos código FP (floating point, para abreviar) con MMX, al acabar la parte FP se habrá vaciado completamente la pila (con FFREE, mismamente), se ejecutará el código MMX y se terminará con la instrucción EMMS (dejándolo libre de nuevo para instrucciones FP) y así sucesivamente.

Los registros MMX se nombran por las posiciones reales que ocupan (MM0,MM1...MM7), y no por la posición respecto de un puntero de pila como en la FPU. A estos ocho registros puede accederse con los modos de direccionamiento habituales, con lo que a ese respecto no tendremos que aprender nada nuevo.

TIPOS DE DATOS

Los registros son de 64 bits, que pueden distribuirse de distinta manera para representar la información. Tenemos por tanto paquetes de bytes (8 bytes por registro), palabras (4 words), palabras dobles (2 dwords) o cuádruples (todo el registro para un único dato de 64 bits, una qword)

¿Dónde está la gracia del asunto? Pues en que ahora podemos, por ejemplo, sumar 8 bytes con signo con otros 8 bytes con signo si están debidamente empaquetados en dos registros, ya que el micro efectuará la suma de cada byte de un registro con el byte correspondiente del otro, todo ello en paralelo. Esto se aplica para operaciones aritméticas y lógicas. Por ejemplo, si tenemos 2 registros MMX con datos de tipo byte sin signo "FF 18 AB C1 01 14 00 F0" y "81 9F 03 01 A1 BB 12 11" y los sumamos con PADDB obtendremos como resultado la suma de cada byte individual con cada byte: "80 B7 AE C2 A2 CF 12 01"

SATURACIÓN Y RECICLADO. Traducciones raras para conceptos simples.

Intel distingue dos modos de operación aritmética: saturating arithmetic y wraparound. Normalmente cuando se efectúa una suma, si el resultado excede el rango de representación aparece un bit de carry que va a algún sitio, o se pierde, y tenemos como resultado la parte baja del resultado "bueno". Esto es lo que se llama wraparound. Por el contrario, se realizan operaciones que saturan cuando un número que se sale del rango válido por arriba por abajo es limitado (como la palabra misma sugiere) al máximo o mínimo representable. Por ejemplo, para números sin signo, F0h + 10h no será 00h (y me llevo una), sino FFh. Dependiendo de si

son números con o sin signo los rangos serán diferentes, claro. Si tenemos una señal de audio muestreada con 16 bits (enteros con signo) y por lo que sea la amplitud ha de incrementarse, si la señal valiese ya 7FFFh en una determinada muestra, el overflow nos la destrozaría pues invertiría ahí la señal (8000h = -32768d). Y lo mismo sucede con el color (si FFh es el máximo de intensidad posible de un color y 00h el mínimo, oscurecer un poco más una imagen a todos los pixels por igual convertiría el negro, 00h, en blanco puro, FFh). En situaciones de este tipo las operaciones ordinarias son muy engorrosas pues exigen numerosas comprobaciones, cuando el concepto de operaciones con saturación nos resuelve inmediatamente la papeleta.

Si te cuento todo este rollo es porque, efectivamente, hay instrucciones MMX con ambos modos de operación.

JUEGO DE INTRUCCIONES MMX

Todas las instrucciones MMX (excepto EMMS) tienen dos operandos, destino y origen, y como siempre en ese mismo orden. El operando origen podrá ser memoria o un registro MMX, mientras que el destino siempre será un registro MMX (excepto en las operaciones de movimiento, claro, porque si no, apañados íbamos). Bueno, que empiece la fiesta.

- INSTRUCCIONES DE MOVIMIENTO DE DATOS

- MOVD (MOVe Dword)

Copia una palabra doble (32 bits) de origen en destino. Cualquiera de los operandos puede ser un registro MMX, memoria, o incluso un registro de propósito general de 32 bits. Sin embargo no es válido mover entre registros MMX, entre posiciones de memoria, ni entre registros de propósito general. Si el destino es un registro MMX el dato se copia a la parte baja y se extiende con ceros. Si el origen es un registro MMX se copia es la parte baja de dicho registro sobre el destino.

- MOVQ (MOVe Qword)

Copia una palabra cuádruple (64 bits) de origen en destino. Cualquier operando puede ser memoria o un registro MMX, o incluso ambos MMX. No se admiten los dos operandos en memoria.

A partir de ahora, y salvo que se diga lo contrario, todas las instrucciones admiten como operandos origen registros MMX o memoria, y como destino obligatoriamente un registro MMX.

- INSTRUCCIONES ARITMÉTICAS

- PADDB, PADDW, PADDD (Packed ADD Byte/Word/Dword)

Suman los paquetes con signo (bytes, words, o dwords) de destino y origen, almacenando el resultado en destino. Si el resultado de un paquete queda fuera del rango se trunca a la parte más baja (es decir, se ignoran los bits de carry de cada suma individual)

- PADDSB, PADDSW (Packed ADD with Saturation Byte/Word)

Suman los paquetes con signo (bytes o words) de origen y destino, almacenando el resultado en destino. Si el resultado de un paquete queda fuera del rango de representación, se limita al máximo o mínimo según corresponda: aplica saturación

- PADDUSB, PADDUSW (Packed ADD Unsigned with Saturation Byte/Word)

Suma los paquetes sin signo de origen y destino, almacenando el resultado en destino. Aplica saturación.

- PSUBB, PSUBW, PSUBD (Packed SUBstract Byte/Word/Dword)

Resta a los paquetes de destino (bytes, words o dwords) los de origen.

- PSUBSB, PSUBSW (Packed SUBstract with Saturation Byte/Word)

Resta a los paquets de destino con signo (bytes o words) los de origen. Aplica saturación.

- PSUBUSB, PSUBUSW (Packed SUBstract Unsigned with Saturation Byte/Word)

Resta a los paquets de destino con signo (bytes o words) los de origen. Aplica saturación.

- PMULHW, PMULLW (Packed MULtiply High/Low Word)

Multiplica las cuatro palabras del operando origen con las cuatro del operando destino,

resultando en cuatro dwords. La parte alta o baja respectivamente de cada dword es almacenada en los paquetes correspondientes del destino. Las operaciones se realizan sobre números con signo.

- PMADDWD (Packed Multiply and ADD)

Multiplica las cuatro palabras del operando origen con las cuatro del operando destino, resultando en cuatro dwords. Las dos dwords superiores son sumadas entre sí y almacenadas en la parte alta del destino. Lo mismo con las dwords inferiores en la parte inferior del destino.

- INSTRUCCIONES DE COMPARACIÓN

- PCMPEQB, PCMPEQW, PCMPEQD (Packed CoMPare for EQual Byte/Word/Dword)

PCMPGTB, PCMPGTW, PCMPGTD (Packed CoMPare for Greater Than Byte/Word/Dword)
Realiza la comparación de igualdad o de "mayor que" entre las palabras de origen y de destino, almacenando en destino el resultado de dicha comparación (verdadero todo a 1, falso todo a 0).

- INSTRUCCIONES LÓGICAS

- PAND, PANDN, POR, PXOR (Packed AND/AND Not/OR/XOR)

Realizan las operaciones lógicas correspondientes bit a bit entre operando y destino, almacenando el resultado en destino. PANDN realiza la operación NOT sobre los bits de destino, y a continuación efectúa AND entre operando y destino, almacenando el resultado en destino. PANDN de un registro consigo mismo equivale a NOT, pues A AND A = A.

- INSTRUCCIONES DE DESPLAZAMIENTO

- PSLLW,PSLLD (Packed Shift Left Logical Word/Dword)

PSRLW,PSRLD (Packed Shift Right Logical Word/Dword)
Efectuan desplazamientos lógicos hacia la izquierda o derecha (Left/Right) sobre las palabras o palabras dobles empaquetadas (Word/Dword); desplazan los bits y rellenan con ceros.

- PSRAW,PSRAD (Packed Shift Right Arithmetic Word/Dword)

Efectuan desplazamientos aritméticos hacia la derecha sobre las palabras o palabras dobles empaquetadas (Word/Dword); equivalen a dividir entre 2 números con signo, pues en el desplazamiento rellenan con el bit más significativo.

EMMS (Empty MMX State) restaura el estado de los registros para que puedan ser usados por la FPU, pues cada vez que se ejecuta una instrucción MMX se marcan todos los registros como con un resultado numérico válido, cuando para su uso por la FPU han de ser marcados como vacíos (que es exactamente lo que hace FFREE)

APLICACIONES

¿Realmente son tan útiles estas instrucciones? Pues sí. Aparte de la aplicación obvia de mover datos en memoria en tochos de 64 bits (cosa no demasiado recomendable mediante instrucciones de la FPU, pues pierde tiempo haciendo conversiones), como ya se dijo, en tratamiento de imágenes tenemos que manipular regiones de memoria bastante generosas, a menudo con operaciones repetitivas. Como es probable que al que no haya hecho nada parecido en su vida no se le ocurra ninguna aplicación, vamos con un ejemplillo con dibujos y todo.

He aquí dos imágenes de un juego que hipotéticamente estamos programando. La de la izquierda corresponde a un decorado estilo años 30, en blanco y negro y todo, y la de la derecha a nuestro protagonista (que a más de un amante de las aventuras gráficas le resultará sobradamente conocido). Las imágenes son almacenadas en 256 tonos de gris, o sea, un byte por pixel; cada imagen es una matriz de enteros, cada byte indicando el brillo (255 blanco puro, 0 negro puro). Nuestra idea es que el personaje se mueva por la pantalla, alternando imágenes en sucesión sobre el decorado. Así pues el primer objetivo es colocar UNA imagen del personaje sobre el fondo.

Fácil, cogemos la imagen del personaje y la pegamos sobre.. Ummm. Es posible que el fondo negro sea un problema. Bueno, entonces miramos byte a byte; si es negro (0) dejamos el fondo como está, y si no lo es sobreescrimos el fondo con el byte correspondiente de nuestro personaje. Podemos asignar entonces al 0 el color "transparente", y distribuir nuestra paleta de grises del 1 al 255 (pues los ojos son negros, y habría que asignarles un color distinto de 0).

No es una mala opción, pero al menos hay otra que es crear lo que se conoce como una máscara. Con cada imagen del personaje almacenamos otra de "negativo", que contiene FFh donde NO hay pixels del personaje, y 00h donde sea así. Si llamamos a F el fondo, P el personaje y M la máscara, tenemos que F_nuevo = [F AND M] + [P AND NOT(M)] (asumiendo por supuesto que asignamos los elementos correctos de cada matriz). Las operaciones lógicas las podremos hacer a toda leche cogiendo los bytes de 8 en 8 con nuestras instrucciones MMX. Puede parecer un poco retorcido en comparación, pero ya veremos que tiene sus ventajas.

Máscara del personaje

Seguimos con el ejemplo. Supongamos ahora que tenemos almacenados los "fotogramas" del movimiento de nuestro intrépido protagonista, pero que al emplearlos en un entorno como el anterior (de noche) vemos que queda demasiado claro: no "pega" la luminosidad que tiene esa imagen con la del fondo. Y más aún, queremos que según nos movamos por la pantalla a puntos más claros (bajo una farola) el personaje se ilumine un poco, y se oscurezca al meterse en zonas sin iluminación. Aquí entra en acción la suma con saturación. Cogemos la imagen del personaje, y sumamos o restamos un valor dado (en función de la luminosidad) a los atributos de color de la matriz. Supongamos que un punto es blanco (FFh) y por efecto del "aclarado" le sumamos 1 más. Gracias a la suma con saturación permanecería FF, pudiendo operar indistintamente con todos los elementos de la matriz. Ahora nos interesa el asunto de la máscara, pues el 00h sería el tope inferior de "oscuridad". Cambiando un simple parámetro podríamos generar con suma facilidad, y en tiempo de ejecución, degradados comos los siguientes:

Si imaginamos ahora lo que tendríamos que hacer para realizar todas esas sumas, comprobaciones, operaciones lógicas prácticamente byte a byte mediante instrucciones convencionales, es obvio que el factor de ganancia es incluso mayor que ocho. ¿Convencido?

CAPÍTULO X: INTERRUPCIONES (a vista de pájaro)

La descripción que sigue corresponde al funcionamiento del procesador en modo real. En modo protegido es algo más complejo, y además como lo normal es que no tengamos privilegios para gestionar interrupciones en un sistema de este tipo, no es tan importante.

El procesador está conectado con multitud de dispositivos externos como el controlador del teclado, discos, el ratón, etc. Sin embargo lo que no puede (o no debe) hacer es mirar periódicamente si cada dispositivo necesita que le envíen datos o tiene un dato por recoger, ya que perdería tiempo inútilmente. En su lugar la mayor parte de estos aparatos disponen de un mecanismo para avisar al micro de que se requiere su atención, conocido como interrupciones hardware (para no confundirse con las llamadas con "INT" que son conocidas como interrupciones software), o simplemente interrupciones. Así el procesador sólo atiende al dispositivo cuando necesita algo.

Cuando ocurre una interrupción hardware el curso del programa puede desviarse a atender dicha interrupción. Entre instrucción e instrucción el micro comprueba si hay alguna interrupción pendiente; si es así empuja el registro de flags en la pila, y a continuación CS e IP. Carga entonces CS:IP con la dirección de la rutina de atención a la interrupción (a menudo ISR de Interruption Service Routine). Ésta rutina contiene el código necesario para atender al dispositivo que provocó la interrupción. Cuando la rutina termina vuelve al punto desde donde saltó mediante IRET, que recupera CS:IP y el registro de flags de la pila. De este modo cuando la interrupción salta en medio de un programa, el funcionamiento de éste no se ve alterado (salvo en que se detiene momentáneamente).

Cada interrupción tiene asignada un número del 0 al 255 que la identifica. En el 8086 la dirección de cada ISR se encuentra en la tabla de vectores de interrupción, ubicada en el primer kbyte de la memoria RAM; esta tabla contiene 256 punteros de 4 bytes (offset + segmento), colocados en orden según el código de interrupción asociado a cada uno. Cuando salta la interrupción X, se lee el puntero en la posición de memoria 4*X.

En modo protegido tenemos en lugar de esta tabla otra más sofisticada llamada IDT (Interrupt Descriptor Table), donde además de la dirección de salto se controlan otros parámetros propios de este modo de funcionamiento. La estructura exacta sólo es de interés para el programador de sistemas, pues corresponde a un área de memoria protegida (sólo el sistema operativo tiene acceso a ella).

El flag de interrupciones determina cuándo y cuándo no se aceptan interrupciones. El bit a 1 las permite, a 0 las inhibe. Manipular este bit es fundamental en operaciones especialmente delicadas, como modificar la rutina de atención a la interrupción. Supongamos que estamos en MSDOS y queremos hacer un programa que se active por las interrupciones del timer del sistema, que las genera automáticamente cada 55 milisegundos. Tendríamos que leer la tabla de vectores de interrupción, guardar en algún otro sitio el puntero que se encuentra en la posición de la interrupción buscada, y escribir en su lugar la posición de memoria donde se ubica nuestro programa. Luego tendríamos que procurar que al finalizar la ejecución de nuestra rutina, el código saltara a la dirección que hubiera previamente en la tabla en lugar de regresar con IRET, para no alterar el funcionamiento del sistema (así tanto nuestra rutina como la anterior saltarían cada 55 milisegundos). El problema es que podría suceder, oh casualidad de las casualidades, que justo cuandos hubiéramos escrito el valor del segmento en la tabla, y fuéramos a escribir el del offset, saltara la interrupción. Es altamente improbable, pero posible. Hay que inhibir las interrupciones. Para ello tenemos las instrucciones STI (SeT Interruption flag) y CLI (CLear Interruption flag), que ponen a uno y a cero respectivamente el flag de interrupciones.

Una situación donde se detienen las interrupciones automáticamente se da cuando escribimos algo en el registro de segmento de pila (SS). Como ya se vio y probablemente no se recuerde, cuando se mueve un valor a SS se inhiben las interrupciones hasta que se ejecuta la instrucción siguiente. De este modo si justo detrás del MOV a SS hacemos un MOV a SP no hay posibilidad de que salte una interrupción entre ambas. Hay tener siempre una zona válida

para la pila porque las interrupciones pueden aparecer en cualquier momento, y por tanto bloquearlas mientras no sea así.

Toda interrupción que salte justo después de modificar SS, o mientras IF valga 0, queda en suspenso hasta que pase esta circunstancia, de modo que no se pierde; tan pronto vuelven a estar disponibles las interrupciones, son atendidas. Si hubiera varias esperando, se atenderían en orden de prioridad de acuerdo con el número de interrupción.

De entre todas las interrupciones hardware hay un tipo especial llamado NMI o interrupción no enmascarable (non maskable interrupt), que se atiende siempre que se produce, sea cual sea el estado del procesador. Normalmente son fallos críticos del sistema como errores físicos del hardware, donde lo mejor que se puede hacer es intentar salvar datos y colocar bien visible para el usuario un letrero de cerrado por defunción (es posible -aunque no siempre ocurra- que sea irrecuperable y requiera, como poco, reiniciar el equipo).

Existe otro tipo de interrupciones que son ocasionadas por el código, y no hardware externo, pero que funcionan de manera idéntica; las interrupciones software. Para llamar a una interrupción determinada usamos la instrucción INT. En la tabla de vectores de interrupción se encuentran las direcciones de gran cantidad de funciones proporcionadas por el sistema operativo y la BIOS, a las cuales podemos llamar mediante esta instrucción. Cuando se habla exclusivamente del sistema operativo suelen conocerse porllamadas al sistema, siendo el medio para solicitar todo tipo de servicios; manejar archivos, reservar memoria, finalizar la ejecución, interaccionar con el teclado/pantalla.. Algo de esto ya se explicó en el capítulo VI, en el apartado de esta misma instrucción.

Cuando el micro lee una instrucción INT se comporta igual que con cualquier interrupción, en tanto que empuja el registro de flags y la dirección de retorno en la pila y salta según el vector de interrupción designado, volviendo luego mediante un IRET. Es posible además mediante INT forzar una interrupción que en principio estaba asociada a un evento hardware, pero no es demasiado recomendable.

Se puede en MSDOS escribir en la tabla de vectores de interrupción y colocar ahí la dirección de un programa residente, que es lo que hace un driver. El ratón, por ejemplo, siempre se ha asociado a la INT 33h. Si uno quiere leer el estado del ratón, pasa los argumentos necesarios (generalmente en los registros) a la ISR, llama a la INT 33h y lo realiza. Cada fabricante de ratones habrá programado su driver para que interaccione con su ratón tal que responda igual que el resto de ratones a esa función. Así el programador tiene garantizado que el programa que haga que maneje el ratón con esa interrupción funcione en cualquier equipo. (Se dice que se accede al dispositivo desde un mayor nivel de abstracción, pues es en este caso el control es independiente del ratón en particular que se use. Cuanto mayor es el nivel de abstracción de un componente, habitualmente menor es la eficiencia, a cambio de facilidad de uso, seguridad y compatibilidad. Es una filosofía masivamente extendida en los sistemas operativos actuales)

Esto de echarle el guante a los vectores de interrupción con tanta facilidad con el MSDOS hacía que fuera muy fácil programar un virus (y uno increíblemente pequeño además). Supongamos que tenemos un ejecutable infectado, que lo que hace es nada más arrancar cargar el virus, y luego ejecutar el código del programa normalmente. El hipotético virus comprueba cuando se ejecuta si ya ha infectado el sistema, y si no es así carga su código en memoria de manera residente (permanece en una parte de la memoria incluso tras haber terminado la ejecución del programa infectado). Sobreescribe a continuación el vector de interrupción que más le interese, que será la que active determinadas funciones del virus. Podemos hacer, por ejemplo, que cada vez que se ejecute un programa éste quede infectado (cosa bastante típica) pues para ello se llaman a servicios del sistema operativo (uno dice "quiero ejecutar este programa", y el virus dice "vale, pero espera que primero le añado mi nota de pie de pagina"). También podemos activar el virus con el timer del ordenador, y que compruebe la fecha y hora continuamente tal que en un determinado momento actúe (no va a ser todo multiplicarse..), haciendo la perrería que su diseñador haya pensado. Simplemente habría que hacer un residente en memoria, sobreescribir el vector de interrupción deseado con la dirección de nuestro código, y que cuando terminase su ejecución en vez de regresar con

IRET saltara a la dirección antigua del vector de interrupción (para que esa interrupción haga, además, lo que debe hacer). Como el modo real carece de medios para impedir que un programa acceda a cualquier posición de memoria, siempre será posible infectar un sistema...

En sistemas de 32 bits la cosa se vuelve mucho más fastidiada, y más en sistemas multiusuario (Linux, Windows 2000/XP) donde los permisos al usuario se dan con cuentagotas. Cuando se ejecuta un programa se le asigna una región de memoria de la que no puede salir, pues en cuanto lo intenta hay un circuito que provoca una interrupción que cede el control al sistema operativo. Todo ello cortesía del Señor Modo Protegido.

Como tercer tipo de interrupciones tenemos las denominadas excepciones (realmente hay quien distingue interrupciones y excepciones como cosas distintas, pero vienen a ser cosas muy similares). Éstas son interrupciones producidas por el código, al igual que las anteriores, pero en respuesta a la ejecución de una instrucción. Por ejemplo, si el procesador se encontrase con que el código de operación que ha leído no corresponde con ninguna instrucción máquina (porque a lo mejor esa instrucción es propia de otro procesador y él no es capaz de interpretarla), se produciría una excepción de código de operación no válido, y saltaría la rutina que gestionaría este hecho. Otras excepciones frecuentes son las de fallo de página (cuando se usa memoria virtual, el procesador va a buscar un dato a memoria y no lo encuentra; tiene que ir antes al disco duro a recogerlo), de fallo de protección general (cuando se realiza una operación privilegiada sin estar autorizado)...

En la sección de enlaces del final se incluye un ZIP con un archivo de ayuda para Windows que comprende una referencia bastante completa de las interrupciones del sistema, tanto hardware como software (cubriendo los servicios del MSDOS, la BIOS y numerosos controladores típicos bajo este sistema operativo). Una fuente mucho más completa es la documentación de Ralph Brown que incluye descripciones de puertos, pinout de componentes, especificaciones hardware... Abrumadoramente extensa, pero muy muy recomendable.

CAPÍTULO XIII: EL MODELO DE MEMORIA EN MODO REAL

Brevísimas anotaciones sobre un sistema obsoleto.

Como bien dice el título, este sistema está obsoleto. Bueno, en parte. Si alguien se digna en leer este capítulo, al final del todo verá porqué lo digo. Es cierto sin embargo que lo incluyo fundamentalmente por culturilla, porque yo he usado MSDOS una buena temporada (pegándome con la configuración de inicio para rascar esos kbytes de memoria convencional extra para conseguir ejecutar un juego), y porque ayuda a entender ciertas situaciones de los equipos actuales que no encajan en nuestra concepción de memoria "plana". Lo que aquí se describa será cierto para cualquier ordenador sin importar la cantidad de memoria instalada; el que me refiera al MSDOS sólo condiciona el cómo era gestionada esa memoria. Veamos ya cómo está distribuida la memoria en un 80x86.

Los primeros 640kb de memoria RAM comprenden la memoria convencional. Esta región es donde se colocaba buena parte del MSDOS y sistemas hermanos del modo real, y donde se colocan normalmente las rutinas de inicio de los sistemas protegidos.

Se llama memoria superior a los 384kb de memoria direccionable que quedan por encima de la convencional (640+384=1024k=1Mb). Sobre ella se mapea la memoria de vídeo, la memoria expandida y la BIOS.

Dentro de la memoria superior las posiciones 0A0000h-0BFFFFh se reservan para la memoria de vídeo. Ahí mapea la tarjeta gráfica su memoria; escribiendo en esas direcciones se escribe realmente sobre la tarjeta de vídeo. El significado del contenido de esta zona depende del modo gráfico en el que estemos (resolución y profundidad de color), así como de la tarjeta. Ésta lo que hace es recorrer continuamente su memoria y proporcionar una salida analógica tras DAC's con las componentes de color junto con unas señales de sincronismo. Con el acceso a los puertos es posible elegir qué parte de la memoria se mapea en la RAM, pues las tarjetas pronto empezaron a contar con más de 64kb de memoria. Se dice entonces que la memoria está paginada, siendo ese segmento su marco de página; estas operaciones eran habituales en las SVGA.

A partir de 0C0000h se encuentra parte de la ROM (de algunas tarjetas de la placa y demás aparejos) y los residentes que el MSDOS ubicaba en el arranque para liberar memoria convencional. Las posiciones 0D0000h-0EFFFFh de memoria superior podían ser ocupados también por la BIOS o más residentes; sin embargo eran importantes porque se mapeaba en un segmento (generalmente 0D000h) la memoria expandida. En 0F0000h se coloca la ROM BIOS (el SETUP, por ejemplo)

Aquí uno solía decir; vale, pero yo tengo instalada más memoria. ¿Qué pasa con ella? Toda la memoria por encima del mega direccionable (directamente, se entiende) se llama memoria extendida. Los primeros 64k eran accesibles sin demasiado problema habilitando una línea adicional del bus, pues como se comentó al calcular la dirección a partir del segmento y el offset podía suceder que tuviéramos bit de carry. Aprovechando esta característica se accedía a la memoria alta, que aparece en los 286 y posteriores. Mediante funciones especiales era posible copiar bloques de memoria extendida sobre la memoria convencional, y viceversa (recordemos que la limitación del modo real en cuando a direccionamiento es que no podemos apuntar a posiciones de memoria por encima del primer mega). De esta manera se accedía a los datos más rápidamente que cargándolos desde el disco; nacieron con ello los discos virtuales o cachés de disco (la idea opuesta a la memoria virtual) para determinadas aplicaciones. El estándar que surgió de todo esto fueron las especificaciones XMS, que además hacían más rápidas las operaciones de memoria en los 386+ (haciendo uso del ancho de 32 bits).

Ya en los tiempos el XT la gente había empezado a darle al tarro para manejar más memoria. La idea era paginar la memoria RAM (como la tarjeta de vídeo) usando un segmento como marco de página, que se dividió en 4 bloques de 16 kb. Se incluyó memoria RAM con circuitería especial y mediante accesos E/S se elegía qué página se colocaba sobre esa

"ventana". Para facilitar la gestión de esta memoria se desarrolló la especificación LIM-EMS (Lotus-Intel-Microsoft Expanded Memory System), un driver que se cargaba en el inicio y accesible a través de la interrupción 67h, haciendo que el programador se pudiera olvidar del control directo de la tarjeta de memoria.

Los 386+ podían "convertir" memoria extendida en expandida a través de un gestor de memoria como el EMM386 (incluído con el MSDOS) o QEMM386. En el arranque el sistema entraba en modo protegido para tener completo control de la memoria y a continuación entraba en modo virtual, emulando en entorno DOS la memoria expandida. Así seguía siendo compatible con todo lo anterior (un casposillo 8086 con esteroides) pero pudiendo manejar la memoria extra como si fuera expandida.

En el Universo Digital de la sección de enlaces se explica el manejo tanto de la memoria extendida como de la expandida, por si alguno se aburre. No es que sea de gran utilidad, pero bueno, si alguien está pensando en hacer un emulador o algo por el estilo, ya sabe..

Esta distribución de la memoria tiene gran interés para el programador a bajo nivel, incluso en modo protegido. Aunque en este modo la memoria se direcciona con total libertad (por lo menos para el sistema operativo), y su gestión es completamente ajena al usuario, hay que saber que ese mapa de memoria está ahí realmente; hay zonas con ROM mapeada que no se deben tocar, la tarjeta de vídeo sigue en un determinado rango de direcciones, etc. Desde ese punto de vista el espacio de memoria no es algo conexo; un 286 que tuviera un megabyte de memoria, disponía de 640kb de memoria convencional y 384kb de memoria extendida (fuera del primer megabyte direccionable!) por este mismo motivo.

ENLACES Y MATERIAL VARIOPINTO

- Universo Digital. Aunque está un pelín desfasado en cuanto al sistema operativo (MSDOS) tiene abundante información sobre programación a muy bajo nivel; da detalles sobre el funcionamiento y acceso a unidades de disco, puertos serie, timers... Para el que quiera libros de esos de papel, que vaya a la bibliografía de esa misma página. Es genial.
- The Art of Assembly . Un conjunto de manuales extremadamente completos de programación en ensamblador, explicando desde estructuración de datos hasta acceso al hardware del sistema. Textos en PDF y online.
- Ralf Brown. Uno de tantos sitios desde donde bajarse la documentación de este caballero. Para muchos, la biblia.
- Web de Darío Alpern. Página personal de un tío al que no conoce ni su madre, que aunque no trae demasiado material de programación, contiene una minienciclopedia de los microprocesadores Intel que detalla gran parte de su arquitectura.
- DDJ Microprocessor Center. Este enlace contiene muchísima información sobre microprocesadores, entre ella la documentación oficial de Intel, así como numerosos bugs y características no documentadas. La leche.
- How to optimize for the Pentium family of microprocessors. Consejos para programación en microprocesadores Pentium.
- linuxassembly.org. Ensamblador para linuxeros.
- Web del canal #asm del irc hispano. Cuenta con algunos fuentes y descargas de los ensambladores más usados.
- Compilador C++ gratuito de Borland.
- Write your own operative system. Descripción básica del arranque del sistema (carga del código del sector de arranque), formatos de ejecutables, sistemas de archivos.. Quizá de título tan optimista como el de mi web.
- Beyond Logic. Recursos sobre acceso a puertos serie, usb, paralelo.. para diversas plataformas, con bastantes tutoriales.
- www.hardwarebook.net. Información sobre distintos cables, pinouts de conectores y circuitillos varios.
- Tutorial del Modo Protegido. Como entrar y salir del modo protegido en MSDOS. No tan sencillo como suena.
- Ejemplos de acceso al modo protegido desde MSDOS.