The Netwide Assembler: NASM

Capítulo 1: Introducción

1.1 ¿Qué es NASM?

The Netwide Assembler, NASM, es un ensamblador 80x86 diseñado para la portabilidad y la modularidad. Soporta diversos formatos de ficheros objeto, incluyendo los 'a.out' de Linux y ELF, NetBSD/FreeBSD, COFF y OBJ de Microsoft 16-bit y Win32. También sacará ficheros en binario puro. Su sintaxis está diseñada para ser simple y fácil de entender, similar a la de Intel pero menos compleja. Soporta códigos de operación de Pentium, P6 y MMX, y tiene capacidades de macro.

1.1.1 ¿Por qué otro ensamblador?

The Netwide Assembler creció a raíz de una idea en 'comp.lang.asm.x86' (o posiblemente en 'alt.lang.asm' - he olvidado cuál), que esencialmente se resume en que parecía no haber ningún buen ensamblador gratis de las series x86, y que quizá debería alguien escribir alguno.

(*) 'a86' es bueno, pero no gratis, y en particular no consigues ninguna capacidad 32-bit hasta que pagas. Y además es sólo para DOS.

(*) 'gas' es gratis, y se porta sobre DOS y Unix, pero no es muy bueno, ya que está diseñado para trabajar de fondo del 'gcc' que siempre suministra el código correcto. Por eso, la comprobación de errores es mínima. Además, su sintaxis es horrible, desde el punto de vista de cualquiera que quiera _escribir_ actualmente algo en él. Y no puedes escribir código 16-bit en él (adecuadamente).

(*) 'as86' es específico de Linux, y (por lo menos mi versión) no parece tener mucha (o ninguna) documentación.

(*) MASM no es muy bueno, y es caro, y sólo corre bajo DOS.

(*) TASM es mejor, pero todavía se esfuerza por tener compatibilidad con MASM, lo que significa millones de directivas y toneladas de papeleo. Y su sintaxis es esencialmente del MASM, con las contradicciones y peculiaridades que conlleva (aunque clasifica algunas de estas por medio del modo Ideal). Es caro también. Y es sólo para DOS.

Por eso aquí, para el placer de tu código, está NASM. Hoy en día todavía está en fase de prototipo - no te prometemos que pueda mejorar cualquiera de estos ensambladores. Pero por favor, _por favor_ envíanos informes sobre bugs, arreglos, información útil, y cualquier cosa que puedas tener en tus manos (¡y gracias a todas las personas que ya han hecho esto!. Ya sabéis quiénes sois), y nosotros lo mejoraremos con mucho gusto. De nuevo.

1.1.2 Condiciones de licencia

Por favor, mira el fichero 'License', suministrado como parte de cualquier fichero de distribución del NASM, para ver las condiciones de licencia bajo las cuales puedes usar el NASM.

1.2 Información de contacto

NASM tiene una página WWW en 'http://www.cryogen.com/Nasm'. Los autores están localizables por E-Mail en 'jules@earthcorp.com' y 'anakin@pobox.com'. Si quieres informarnos de un bug, por favor, lee la sección 10.2 primero.

Los nuevos lanzamientos de NASM se subirán a 'sunsite.unc.edu', 'ftp.simtel.net' y 'ftp.coast.net'. Los anuncios se ponen en 'comp.lang.asm.x86', 'alt.lang.asm', 'comp.os.linux.announce' y 'comp.archives.msdos.announce' (en la última se hace automáticamente al subir algo a 'ftp.simtel.net').

Si no tienes acceso a Usenet, o quisieras ser informado por E-Mail cuando salgan nuevas versiones, E-Mail a 'anakin@pobox.com' y pregunta.

1.3 Instalación

1.3.1 Instalando NASM bajo MS-DOS o Windows

Una vez que has obtenido el fichero comprimido del NASM, 'nasmXXX.ZIP' (donde XXX denota el número de versión del NASM contenido en el archivo), descomprímelo dentro de su mismo directorio (por ejemplo 'c:\nasm').

El archivo contendrá cuatro ficheros ejecutables: los ejecutables NASM 'nasm.exe' y 'nasmw.exe', y los ficheros ejecutables NDISASM 'ndisasm.exe' y 'ndisasmw.exe'. En cada caso, el fichero cuyo nombre termina en 'w' es un ejecutable Win32, diseñado para correr bajo Windows 95 o Windows NT sobre Intel, y el otro es un ejecutable para DOS de 16-bit.

El único fichero que el NASM necesita para ejecutar es el propio ejecutable, por eso copia (al menos) el 'nasm.exe' o el 'nasmw.exe' a un directorio en tu PATH, o alternativamente edita el 'autoexec.bat' para añadir el directorio del NASM a tu 'PATH'. (si estás instalando sólo la versión Win32, puedes renombrarlo a 'nasm.exe').

Esto es todo - NASM está instalado. No necesitas que el directorio 'nasm' esté presente para ejecutar NASM (a menos que lo hayas añadido a tu 'PATH'), por lo que puedes borrarlo si necesitas liberar espacio; sin embargo, podrías querer mantener la documentación o los programas de prueba.

Si has bajado el archivo de código fuente para DOS, 'nasmXXXs.zip', el directorio 'nasm' contendrá también el código fuente completo, y una selección de los ficheros Make que puedes (espero) usar para reconstruir tu copia de NASM desde el principio. El fichero 'Readme' lista los distintos ficheros Make y con qué compiladores funcionarán. Nótese que los fichero fuente 'insnsa.c' y 'insnsd.c' se generan automáticamente desde la tabla maestra de instrucción 'insns.dat' mediante un script Perl; se proporciona una versión QBasic del programa, pero es preferible que uses la versión Perl. Un puerto DOS del Perl está disponible desde www.perl.org.

1.3.2 Instalando NASM bajo Unix

Una vez has obtenido el archivo con el código fuente Unix del NASM, 'nasm-X.XX.tar.gz' (donde 'X.XX' denota el número de versión del NASM contenido en el archivo), descomprímelo en un directorio como '/usr/local/src'. El archivo, cuando es descomprimido, creará su propio subdirectorio 'nasm-X.XX'.

NASM es un paquete auto-configurado: una vez lo has descomprimido, 'cd' hasta el directorio en que se ha descomprimido y escribe './configure'. Este script del shell encontrará el mejor compilador de C para usarlo para construir NASM y configurar los ficheros Make de manera adecuada.

Una vez que NASM se ha autoconfigurado, puedes escribir 'make' para construir los binarios 'nasm' y 'ndisasm', y entonces 'make install' para instalarlo en '/usr/local/bin' e instalar las páginas de manual 'nasm.1' y 'ndisasm.1' en '/usr/local/man/man1'. Alternativamente, puedes dar opciones como '--prefix' al script 'configure' (ver el fichero 'INSTALL' para más detalles), o instalar el programa tú mismo.

NASM también viene con un conjunto de utilidades para el manejo de formato personalizable de ficheros objeto RDOFF, que está en el subdirectorio 'rdoff' del archivo NASM. Puedes construir estos con 'make rdf' e instalarlos con 'make rdf_install', si los quieres.

Si NASM falla al auto-configurarse, puedes todavía hacerlo compilar usando el fichero make de Unix 'Makefile.unx'. Copia o renombra el fichero a 'Makefile' e intenta escribiendo 'make'. Hay también un fichero 'Makefile.unx' en el subdirectorio 'rdoff'.

Capítulo 2: Ejecutando NASM

2.1 Sintaxis de la línea de comando del NASM

Para ensamblar un fichero, darás un comando de la forma

nasm -f <formato> <nombre_del_fichero> [-o <salida>]

Por ejemplo,

nasm -f elf myfile.asm

ensamblará 'myfile.asm' en un fichero objeto ELF 'myfile.o'. Y

nasm -f bin myfile.asm -o myfile.com

ensamblará 'myfile.asm' en un fichero binario puro 'myfile.com'.

Para producir un fichero de lista, con la salida de los códigos hexadecimales desde NASM mostrados a la izquierda de los fuentes originales, usa la opción '-l' para dar el nombre de un fichero de lista, por ejemplo:

nasm -f coff myfile.asm -l myfile.lst

Para conseguir instrucciones de uso adicionales desde NASM, intenta escribiendo:

nasm -h

Esto también listará los formatos de salida disponibles, y cuáles son.

Si usas Linux pero no estás seguro si tu sistema es 'a.out' o ELF, escribe:

file nasm

(en el directorio en el cual pusiste el binario NASM cuando lo instalaste). Si te dice algo así como

nasm: ELF 32-bit LSB executable i386 (386 and up) Version 1

entonces tu sistema es ELF, y deberías usar la opción '-f elf' cuando quieras usar NASM para producir ficheros objeto de Linux. Si te dice

nasm: Linux/i386 demand-paged executable (QMAGIC)

o algo similar, tu sistema es 'a.out', y deberías usar '-f aout' en su lugar.

Al igual que los compiladores y ensambladores Unix, NASM se calla a no ser que haya algo mal: no verás ninguna salida en absoluto, a menos que dé un mensaje de error.

2.1.1 La opción '-o': Especificando el nombre del fichero de salida

NASM normalmente elegirá por ti el nombre de tu fichero de salida; precisamente cómo lo haga depende del formato del fichero objeto. Para los formatos de fichero objeto de Microsoft ('obj' y 'win32'), eliminará la extensión '.asm' (o cualquier extensión que te guste usar - a NASM no le importa) del nombre de tu fichero fuente y lo sustituirá por '.obj'. Para formatos de fichero objeto Unix ('aout', 'coff', 'elf' y 'as86') será sustituido por '.o'. Para 'rdf', usará '.rdf', y para el formato binario simplemente eliminará la extensión, por eso 'myfile.asm' produce el fichero de salida 'myfile'.

Si el fichero de salida ya existe, NASM lo sobreescribirá, a menos que tenga el mismo nombre que el fichero de entrada en cuyo caso dará un aviso y usará en su lugar 'nasm.out' como nombre del fichero de salida.

Para las situaciones en las que el comportamiento es inaceptable, NASM proporciona la opción '-o' del a línea de comandos, que te permite especificar el nombre que desees para el fichero de salida. Indicas la opción '-o' seguida del nombre que desees para el fichero de salida, con o sin espacio. Por ejemplo:

```
nasm -f bin program.asm -o program.com
nasm -f bin driver.asm -odriver.sys
```

2.1.2 La opción '-f': Especificando el formato del fichero de salida.

Si no suministras la opción '-f' a NASM, éste elegirá por sí mismo un formato para el fichero de salida. En las versiones de distribución de NASM, por defecto siempre es 'bin'; si has compilado tu propia copia de NASM, puedes redefinir 'OF_DEFAULT' en tiempo de compilación y elegir qué quieres que sea por defecto.

Al igual que '-o', el espacio entre '-f' y el formato del fichero de salida es opcional; por eso '-f elf' y '-felf' son ambos válidos.

Se puede obtener una lista completa de los formatos de salida disponibles enviando el comando 'nasm -h'.

2.1.3 La opción '-l': Generando un fichero de lista

Si proporcionas la opción '-l' al NASM, seguida (con el usual espacio opcional) de un nombre de fichero, NASM generará automáticamente un fichero fuente de lista, en el que son listadas a la izquierda las direcciones y el código generado, y el código fuente actual, con expansiones de macros multi-línea (excepto aquellos que específicamente soliciten no-expansión en las listas fuente: ver sección 4.2.9) a la derecha. Por ejemplo:

nasm -f elf myfile.asm -l myfile.lst

2.1.4 La opción '-s': Enviar errores a 'stdout'

Bajo DOS puede ser difícil (creo que hay maneras) redireccionar la salida estándar de errores desde un programa a un fichero. Puesto que NASM produce normalmente sus mensajes de error y de aviso en 'stderr', puede hacerse difícil capturar los errores si (por ejemplo) deseas cargarlos en un editor.

Por lo tanto, NASM proporciona la opción '-s', sin ningún argumento, que hace que los errores sean enviados a la salida estándar en lugar de a la estándar de errores. Por eso puedes redireccionar los errores a un fichero escribiendo

nasm -s -f obj myfile.asm > myfile.err

2.1.5 La opción '-i': Directorios de búsqueda de un fichero include

Cuando NASM ve la directiva '%include' en un fichero fuente (ver la sección 4.5), buscará dicho fichero no sólo en el directorio actual, sino también en cualquier directorio especificado en la línea de comandos usando la opción '-i'. Por lo tanto puedes incluir ficheros desde una librería de macros, por ejemplo, escribiendo

nasm -ic:\macrolib\ -f obj myfile.asm

(Como siempre, se permite el espacio entre '-i' y el path, y es opcional)
.

NASM, en base al interés de la completa portabilidad del código fuente, no entiende los convenios de nombres de ficheros del OS en el que se está ejecutando; la cadena que entregues como argumento a la opción '-i' precederá al fichero incluido exactamente como la hayas escrito. Por eso, es necesaria la barra invertida '\' final en el ejemplo de arriba. Bajo Unix, la barra '/' final es necesaria igualmente.

(Puedes usar esto en tu beneficio, si eres realmente perverso, si te das cuenta que la opción '-ifoo' causará que '%include "bar.i"' busque el fichero 'foobar.i'...)

Si quieres definir un path _estándar_ de búsqueda de include, parecido al '/usr/include' en los sistemas Unix, deberías colocar una o más directivas '-i' en la variable de entorno 'NASM' (ver sección 2.1.11).

2.1.6 La opción '-p': Pre-incluir un fichero

NASM te permite especificar ficheros para que sean _pre-incluidos_ en tu fichero fuente, usando la opción '-p'. Por eso, ejecutar

nasm myfile.asm -p myinc.inc

es equivalente a ejecutar 'nasm myfile.asm' y colocar la directiva '%include "myinc.inc"' al principio del fichero.

2.1.7 La opción '-d': Pre-definir un macro

Al igual que la opción '-p', da una manera alternativa de colocar una directiva '%include' al principio del fichero fuente, la opción '-d' da una manera alternativa de colocar una directiva '%define'. Puedes codificar

```
nasm myfile.asm -dFOO=100
```

como alternativa a colocar la directiva

```
%define FOO 100
```

al comienzo del fichero. Puedes omitir el valor del macro, esto es: la opción '-dFOO' es equivalente a codificar '%define FOO'. Esta forma de directiva puede ser útil para seleccionar opciones en tiempo de ensamblaje que son comprobadas usando '%ifdef', por ejemplo '-dDEBUG'.

2.1.8 La opción '-e': Sólo preprocesar

NASM permite al preprocesador que se ejecute solo, hasta un punto. Usando la opción '-e' (que no requiere argumentos) hará que NASM preprocese su fichero de entrada, expanda todas las referencias a macros, quite todos los comentarios y directivas del preprocesador, y escriba el fichero resultante en la salida estándar (o lo salve a un fichero, si la opción '-o' es usada a su vez).

Esta opción no se puede aplicar a programas que requieran del preprocesador para evaluar expresiones que dependan de los valores de símbolos: por ejemplo el siguiente código

```
%assign tablesize ($-tablestart)
```

causará un error en el modo de sólo preprocesar.

2.1.9 La opción '-a': No preprocesar en absoluto

Si NASM está siendo usado como 'back end' de un compilador, podría desearse suprimir completamente el preprocesado y asumir que el compilador ya lo ha hecho, para ganar tiempo e incrementar la velocidad de compilación. La opción '-a', que no requiere ningún argumento, le dice al NASM que sustituya su potente preprocesador por un preprocesador que no hace nada.

2.1.10 La opción '-w': Activar o desactivar los avisos de ensamblaje

NASM puede observar muchas condiciones durante el curso del ensamblaje que son de mención especial para el usuario, pero no errores lo suficientemente importante como para justificar que NASM rechace el generar un fichero de salida. Estas condiciones son informadas como errores, pero vienen con la palabra 'warning' antes del mensaje. Los avisos no evitan que NASM genere un fichero de salida y devuelva un estado satisfactorio al sistema operativo.

Algunas condiciones son todavía menos importantes que esto: son sólo algunas veces importantes para el usuario. Por eso, NASM implementa la opción '-w' en la línea de comandos, que activa o desactiva cierta clase

de avisos del ensamblador. Tales clases de avisos son descritas por un nombre, por ejemplo 'orphan-labels'; puedes activar los avisos de esta clase con la opción de línea de comandos '-w+orphan-labels' y desactivarlos con '-w-orphan-labels'.

Las clases de avisos suprimibles son:

(*) 'macro-params' cubre los avisos sobre los macros multi-línea que son invocados con un número erróneo de parámetros. Esta clase de avisos está activa por defecto; ver la sección 4.2.1 para un ejemplo de por qué puedes querer desactivarlos.

(*) 'orphan-labels' cubre los avisos sobre las líneas del fuente que no contienen pero definen una etiqueta que no está seguida de ':'. NASM no avisa por defecto de esto en algunas situaciones oscuras; ver la sección 3.1 para un ejemplo de por qué lo puedes querer.

(*) 'number-overflow' cubre los avisos sobre las constantes numéricas que no caben en 32 bits (por ejemplo, es fácil escribir una F de más y generar '0x7fffffff' por error). Esta clase de avisos está activada por defecto.

2.1.11 La variable de entorno 'NASM'

Si defines una variable de entorno llamada 'NASM', el programa la interpretará como una lista extra de opciones de la línea de comandos, que es procesada antes de la línea real de comandos. Puedes usar esto para definir directorios estándar de búsqueda de los ficheros include, poniendo la opción '-i' en la variable 'NASM'.

El valor de la variable se divide en el espacio en blanco, por eso el valor '-s -ic:\nasmlib' será tratado como dos opciones separadas. Sin embargo, esto quiere decir que el valor '-dNAME="my name"' no hará lo que podrías querer, porque será partida en el espacio en blanco y el procesamiento de la línea de comandos del NASM se verá confundido por las dos palabras sin sentido '-dNAME="my' y 'name"'.

Para evitar esto, NASM proporciona una característica, si comienzas la variable de entorno 'NASM' con algún carácter que no sea el signo menos, NASM tratará este carácter como un carácter separador de opciones. Por eso, estableciendo la variable 'NASM' con el valor '!-s!-ic:\nasmlib' equivale a establecerla con '-s -ic:\nasmlib', y '!-dNAME="my name"' funcionará.

2.2 Comienzo rápido para los usuarios de MASM

Si estás acostumbrado a escribir programas con MASM, o con TASM en el modo compatible con MASM (no Ideal), o con 'a86', esta sección intenta remarcar las principales diferencias entre la sintaxis de MASM y de NASM. Si no estás acostumbrado a MASM, es probablemente mejor que te saltes esta sección.

2.2.1 NASM es sensible a mayúsculas

Una sencilla diferencia es que NASM es sensible a mayúsculas. Diferencia si tu llamas a una etiqueta 'foo', 'Foo' o 'FOO'. Si estás ensamblando ficheros '.OBJ' para DOS u OS/2, puedes invocar la directiva 'UPPERCASE' (documentada en la sección 6.2) para asegurarte que todos los símbolos exportados a otros módulos de código se fuerzan a mayúsculas; pero incluso entonces, _dentro_ de un módulo sencillo, NASM distinguirá entre

etiquetas que difieran en ser mayúsculas o minúsculas.

2.2.2 NASM requiere de corchetes para las referencias a memoria

NASM fue diseñado teniendo en mente la simplificación de la sintaxis. Una de las metas de diseño de NASM es que debería ser posible, tanto como sea práctico, para el usuario mirar una simple línea de código NASM y decir que código de operación genera. Esto no se puede hacer en MASM: si declaras por ejemplo,

```
foo       equ 1
bar       dw 2
```

entonces las dos líneas de código

```
          mov ax,foo
          mov ax,bar
```

generaran códigos de operación completamente diferentes, a pesar de tener sintaxis de apariencia idéntica.

NASM evita estas situaciones indeseables teniendo una sintaxis mucho más simple para las referencias a memoria. La regla es sencilla ya que cualquier acceso al _contenido_ de una posición de memoria requiere de unos corchetes rodeando la dirección, y cualquier acceso a la _dirección_ de memoria de una variable no. Por eso, una instrucción de la forma 'mov ax,foo' _siempre_ se referirá a una constante en tiempo de compilación, ya sea un 'EQU' o la dirección de una variable; y para acceder al _contenido_ de la variable 'bar', debes codificar 'mov ax,[bar]'.

Esto también significa que NASM no necesita la palabra clave del MASM 'OFFSET', puesto que el código MASM 'mov ax,offset bar' significa exactamente lo mismo que el 'mov ax,bar' del NASM. Si estás intentando coger grandes cantidades de código MASM para ensamblar bajo NASM, siempre puedes codificar '%idefine offset' para hacer que el preprocesador trate la palabra clave 'OFFSET' como algo sin función alguna.

Este asunto es todavía más confuso en 'a86', donde declarar una etiqueta seguida de dos puntos la define como una 'etiqueta' en oposición a una 'variable', y hace que 'a86' adopte semántica del estilo de NASM; por eso en 'a86', 'mov ax,var' tiene un comportamiento distinto dependiendo de si 'var' ha sido declarada como 'var: dw 0' (una etiqueta) o como 'var dw 0' (una variable del tamaño de una palabra). NASM es más sencillo en comparación: _todo_ es una etiqueta.

NASM, en base a la simplicidad, tampoco implementa la sintaxis híbrida soportada por MASM y sus clones, como 'mov ax,table[bx]', donde una referencia a memoria está indicada por una porción fuera de los corchetes y otra dentro. La sintaxis correcta para lo de arriba es 'mov ax,[table+bx]'. Igualmente, 'mov ax,es:[di]' es incorrecto y 'mov ax,[es:di]' es correcto.

2.2.3 NASM no almacena los tipos de las variables

NASM, por diseño, decide no recordar los tipos de las variables que declares. Mientras que MASM recordará, al ver 'var dw 0', que has declarado 'var' como una variable del tamaño de la palabra, y será capaz de decidir en la ambigüedad del tamaño de la instrucción 'mov var,2', NASM no recordará deliberadamente nada sobre el símbolo

'var' excepto dónde comienza, y por eso debes especificar el código 'mov word [var],2'.

Por esta razón, NASM no implementa las instrucciones 'LODS', 'MOVS', 'STOS', 'SCAS', 'CMPS', 'INS' o 'OUTS', sino que sólo proporciona las formas tales como 'LODSB', 'MOVSW' y 'SCASD' que especifican explícitamente el tamaño de los componentes de las cadenas que están siendo manipuladas.

2.2.4 NASM no tiene 'ASSUME'

Como parte del camino de NASM hacia la simplicidad, tampoco implementa la directiva 'ASSUME'. NASM no seguirá la pista de los valores que decidas poner en tus registros de segmento, y nunca generará _automáticamente_ un prefijo de superposición de segmento.

2.2.5 NASM no soporta modelos de memoria

NASM tampoco tiene directivas para soportar los diferentes modelos de memoria de 16-bit. El programador deberá seguir el rastro de aquellas funciones que se supone que son llamadas con una llamada lejana y cuáles con una cercana, y es responsable de poner de forma correcta la instrucción 'RET' ('RETN' o 'RETF'; NASM acepta 'RET' en sí mismo como una forma alternativa de 'RETN'); además, el programador es responsable de codificar las instrucciones CALL FAR donde sea necesario cuando se llame a funciones _externas_, y también debe seguir la pista de qué definiciones de variables externas son lejanas y cuáles cercanas.

2.2.6 Diferencias en la coma flotante

NASM usa diferentes nombres para referirse a los registros de coma flotante desde MASM: donde MASM debería llamarlos 'ST(0)', 'ST(1)' y demás, y 'a86' los llamaría simplemente '0', '1', etc, NASM decide llamarlos 'st0', 'st1', etc.

Como en la versión 0.96, NASM trata las instrucciones con formas 'nowait' del mismo modo que los ensambladores compatibles con MASM. El tratamiento idiosincrásico empleado por la 0.95 y anteriores estaba basado en un malentendido por lo autores.

2.2.7 Otras diferencias

Por razones históricas, NASM usa la palabra clave 'TWORD' donde MASM y sus compatibles usan 'TBYTE'.

NASM no declara los almacenamiento no iniciados de la misma forma que MASM: donde un programador de MASM debería usar 'stack db 64 dup (?)', NASM requiere 'stack resb 64', entendiendo que se debe leer como 'reserva 64 bits'. Por un poco de compatibilidad, puesto que NASM trata '?' como un carácter válido en los nombres de los símbolos, puedes codificar '? equ 0', y entonces escribiendo 'dw ?' hará al menos algo vagamente útil. 'DUP', sin embargo, todavía no está implementado en la sintaxis.

Adicionalmente a todo esto, las macros y directivas funcionan completamente diferentes a MASM. Ver el capítulo 4 y 5 para más detalles.

Capítulo 3: El lenguaje NASM

3.1 Composición de una línea fuente de NASM

Como muchos ensambladores, cada línea de código fuente NASM contiene (a menos que sea una macro, una directiva del preprocesador o una directiva del ensamblador: ver el capítulo 4 y 5) algunas combinaciones de los cuatro campos

```
etiqueta:   instrucción operandos     ; comentario
```

Como es usual, la mayoría de estos campos son opcionales; se permite la presencia o ausencia de cualquier combinación de una etiqueta, una instrucción y un comentario. por supuesto, el campo del operando se requiere o se prohíbe dependiendo de la presencia y naturaleza del campo de instrucción.

NASM no pone ninguna restricción de espacios en blancos en una línea; las etiquetas pueden tener espacios en blanco antes de ellas, o las instrucciones pueden no tener espacios antes de ellas, o cualquier cosa. Los dos puntos después de la etiqueta también son opcionales. (Nótese que esto quiere decir que si intentas codificar 'lodsb' sólo en una línea, y escribes 'lodab' por error, entonces esto será una línea del fuente válida que no hace nada sino definir una etiqueta. Ejecutando NASM con la opción en la línea de comandos '-w+orphan-labels' hará que se te avise cuando definas una etiqueta solitaria en una línea que no esté seguida de los dos puntos.)

Los caracteres válidos en las etiquetas son letras, números, '_', '$', '#', '@', '~', '.', y '?'. Los únicos caracteres que deberían poderse usar como _primer_ carácter de un identificador son letras, '.' (con un significado especial: ver sección 3.8), '_' y '?'. Un identificador podría también estar precedido por un '$' para indicar que se pretende que se lea como un identificador y no como una palabra reservada; por eso, si algún otro módulo que estés enlazando define un símbolo llamado 'eax', puedes referirte a '$eax' en el código NASM para distinguir el símbolo del registro.

El campo de instrucción puede contener cualquier instrucción máquina: instrucciones Pentium o P6, instrucciones FPU, instrucciones MMX e incluso instrucciones no documentadas. Las instrucciones deberían ir precedidas de la manera habitual por 'LOCK', 'REP, 'REPE'/'REPZ' o 'REPNE'/'REPNZ'. Se permiten los prefijos explícitos de tamaño de dirección y de tamaño de operando, 'A16', 'A32', O16' y 'O32' - un ejemplo de su uso está en el capítulo 9. También puedes usar el nombre de un registro de segmento como prefijo de una instrucción: codificar 'es mov [b],ax' equivale a codificar 'mov [es:b],ax'. Recomendamos la última sintaxis, ya que es consistente con otras características sintácticas del lenguaje, aunque con instrucciones como 'LODSB', que no tiene operandos y todavía puede requerir una superposición de segmentos, no hay un modo sintáctico limpio de proceder aparte de 'es lodsb'.

No se requiere una instrucción para usar un prefijo: prefijos como 'CS', 'A32', 'LOCK' o 'REPE' pueden aparecer en una línea ellos solos, y NASM simplemente generará los bytes prefijo.

Adicionalmente a las actuales instrucciones máquina, NASM también soporta un número de pseudo-instrucciones, descritas en la sección 3.2.

Los operandos de instrucciones pueden tomar diversas formas: pueden ser registros, descritos simplemente por el nombre del registro (ej: 'ax', 'bp', 'ebx', 'cr0': NASM no usa la sintaxis del estilo 'gas' en la cual

los nombres de los registros han de estar precedidos por un símbolo '%'), o pueden ser direcciones efectivas (ver sección 3.3), constantes (sección 3.4) o expresiones (sección 3.5).

Para las instrucciones de coma flotante, NASM acepta un amplio rango de sintaxis: puedes usar las formas de dos operandos como la que soporta MASM, o puedes usar la forma nativa del MASM de un solo operando en la mayoría de los casos. Los detalles de todas las formas de cada una de las instrucciones soportadas está en el apéndice A. Por ejemplo, puedes codificar:

```
fadd st1              ; establece st0 := st0 + st1
fadd st0,st1          ; también lo hace

fadd st1,st0          ; establece st1 := st1 + st0
fadd to st1           ; ídem
```

Casi cualquier instrucción de coma flotante que referencie a memoria debe usar uno de los siguientes prefijos 'DWORD', 'QWORD' o 'TWORD' para indicar el tamaño del operando de memoria al que se refiere.

3.2 Pseudo-instrucciones

Las pseudo-instrucciones son cosas que, aunque no instrucciones máquina reales del x86, se usan en el campo de instrucción de cualquier modo porque es el lugar más conveniente para ponerlas. Las actuales pseudo-instrucciones son 'DB', 'DW', 'DD', 'DQ' y 'DT', sus contrapartidas no inicializadas 'RESB', 'RESW', 'RESD', 'RESQ' y 'REST', el comando 'INCBIN', el comando 'EQU' y el prefijo 'TIMES'.

3.2.1 'DB' y similares: Declarando datos inicializados

'DB', 'DW', 'DD', 'DQ' y 'DT' son usados, al igual que in MASM, para declarar datos inicializados en el fichero de salida. Pueden invocarse de un amplio rango de maneras:

```
db 0x55               ; simplemente el byte 0x55
db 0x55,0x56,0x57     ; tres bytes sucesivos
db 'a',0x55           ; constantes carácter son correctas
db 'hello',13,10,'$'  ; también hay constantes cadena
dw 0x1234             ; 0x34 0x12
dw 'a'                ; 0x41 0x00 (simplemente un número)
dw 'ab'               ; 0x41 0x42 (constante carácter)
dw 'abc'              ; 0x41 0x42 0x43 0x00 (cadena)
dd 0x12345678         ; 0x78 0x56 0x34 0x12
dd 1.234567e20        ; constante de coma flotante
dq 1.234567e20        ; coma flotante de doble precisión
dt 1.234567e20        ; coma flotante de precisión extendida
```

'DQ' y 'DT' no aceptan como operandos constantes numéricas o constantes cadena.

3.2.2 'RESB' y similares: Declarando datos no inicializados

'RESB', 'RESW', 'RESD', 'RESQ' y 'REST' están diseñados para ser usados en la sección BSS de un módulo: declaran espacio de almacenamiento _no inicializado_. Cada uno lleva un solo operando, que es el número de bytes, palabras, palabras dobles o lo que sea para reservar. Tal y como ha sido expuesto en la sección 2.2.7, NASM no soporta la sintaxis de reservar espacio no inicializado del MASM/TASM usando 'DW ?'o cosas

similares: esto es lo que hace en su lugar. El operando de un 'RESB'-tipo pseudo-instrucción es una _expresión crítica_: ver sección 3.7.

Por ejemplo:

```
buffer:   resb 64               ; reserva 64 bytes
wordvar:  resw 1                ; reserva una palabra
realarray resq 10               ; array de 10 reales
```

3.2.3 'INCBIN': Incluyendo ficheros binarios externos

'INCBIN' ha sido tomada prestada del antiguo ensamblador de Amiga DevPac: incluye un ficheros binario palabra por palabra dentro del fichero de salida. Esto puede ser práctico para (por ejemplo) incluir gráficos y sonido directamente en un fichero ejecutable de un juego. Puede llamarse de una de estas tres maneras:

```
          incbin "file.dat"      ; incluye todo el fichero
          incbin "file.dat",1024 ; se salta los primeros 1024 bytes
          incbin "file.dat",1024,512 ; salta los primeros 1024, e
                                 ; incluye como máximo 512
```

3.2.4 'EQU': Definiendo constantes

'EQU' define un símbolo para un valor determinado de constante: cuando se usa 'EQU', la línea de código fuente debe contener una etiqueta. La acción de 'EQU' es para definir el nombre de la etiqueta dada para el valor de su (único) operando. Esta definición es absoluta, y no puede cambiar después. Por eso, por ejemplo,

```
message   db 'hello, world'
msglen    equ $-message
```

define 'msglen' para ser la constante 12. 'msglen' no podría ser definido después. Esto no es una definición del preprocesador: el valor de 'msglen' se evalúa _una sola_ vez, usando el valor de '$' (ver sección 3.5 para una explicación de '$') en el momento de la definición, en lugar de ser evaluado si es referenciado y usando el valor de '$' en el punto de la referencia. Nótese que el operando de un 'EQU' es también una expresión crítica (sección 3.7).

3.2.5 'TIMES': Repitiendo instrucciones o datos

El prefijo 'TIMES' hace que la instrucción se ensamble múltiples veces. Esto está presente parcialmente como el equivalente 'DUP' de la sintaxis MASM soportada por los ensambladores compatibles con el MASM, en los que puedes codificar

```
zerobuf:  times 64 db 0
```

o cosas similares: pero 'TIMES' es más versátil que esto. El argumento de 'TIMES' no es simplemente una constante numérica, sino una _expresión_ numérica, por eso puedes hacer cosas como ésta

```
buffer:   db 'hello ,world'
          times 64-$+buffer db ' '
```

que almacenará exactamente los espacios necesarios para hacer el total de la longitud de 'buffer' hasta 64. Finalmente, 'TIMES' puede aplicarse a un instrucción normal, por eso puede codificar loops desenrollados

triviales en ella:

```
          times 100 movsb
```

Dése cuenta que no hay diferencias efectivas entre 'times 100 resb 1' y 'resb 100', excepto que éste último será ensamblado unas 100 veces más rápido debido a la estructura interna del ensamblador.

El operando de 'TIMES', al igual que el de 'EQU' y aquellos de 'RESB' y similares, es una expresión crítica (sección 3.7).

Nótese también que 'TIMES' no puede aplicarse a macros: la razón para esto es que 'TIMES' se procesa después de la fase de macros, que permite el argumento de 'TIMES' el contener una expresión como '64-$+buffer' como arriba. Para repetir de una línea de código, o una macro compleja, usa la directiva del preprocesador '%rep'.

3.3 Direcciones efectivas

Una dirección efectiva es cualquier operando de una instrucción que referencie a memoria. Las direcciones efectivas, en NASM, tienen una sintaxis simple: consisten en una evaluación de expresión en la dirección deseada, encerrada entre corchetes. Por ejemplo:

```
wordvar   dw 123
          mov ax,[wordvar]
          mov ax,[wordvar+1]
          mov ax,[es:wordvar+bx]
```

Cualquier cosa que no vaya conforme este ejemplo no es una referencia a memoria válida en NASM, por ejemplo 'es:wordvar[bx]'.

Direcciones efectivas más complicadas, como aquellas que involucran más de un registro, funcionan exactamente de la misma manera:

```
          mov eax,[ebx*2+ecx+offset]
          mov ax,[bp+di+8]
```

NASM es capaz de operar algebraicamente con estas direcciones efectivas, por eso, estas cosas que no necesariamente _parecen_ legales son perfectamente correctas:

```
          mov eax,[ebx*5]        ; ensambla como [ebx*4+ebx]
          mov eax,[label1*2-label2] ; esto es [label1+(label1-label2)]
```

Algunas formas de direcciones efectivas tienen más de una forma ensamblada; en la mayoría de los casos NASM generará la forma más pequeña que pueda. Por ejemplo, hay distintas formas ensambladas para la dirección efectiva de 32-bit '[eax*2+0]' y '[eax+eax]', y NASM generará generalmente la última, por la razón por que el offset 0 requiere cuatro bytes de almacenamiento.

NASM tiene un mecanismo de ayuda que hará que '[eax+ebx]' y '[ebx+eax]' generen diferentes códigos de operación; ocasionalmente esto es útil porque '[esi+epb]' y '[ebp+esi]' tienen por defecto diferentes registros de segmento.

Sin embargo, puedes forzar a NASM a que genere una dirección efectiva de una manera particular usando las palabras clave 'BYTE', 'WORD', 'DWORD' y 'NOSPLIT'. Si necesitas que '[eax+3]' se ensamble usando un campo de

offset de palabra doble en lugar de la de un byte que normalmente NASM generará, puedes codificar '[dword eax+3]'. Similarmente, puedes forzar a NASM a usar un offset de un byte para un valor pequeño que no ha sido visto en la primera pasada (ver sección 3.7 para un ejemplo de un fragmento de código) usando '[byte eax+offset]'. Como casos especiales, '[byte eax]' codificará '[eax+0]' con un offset cero de un byte , y '[dword eax]' lo codificará con un offset cero de palabra doble. La forma normal, '[eax]', será codificada sin campo de offset.

De forma similar, NASM partirá '[eax+2]' en '[eax+eax]' porque esto permite omitir el campo de offset y salvar espacio; de hecho, también partirá '[eax*2+offset]' en '[eax+eax+offset]'. Puedes combatir este comportamiento usando la palabra clave 'NOSPLIT':
'[nosplit eax*2]' forzará que se genere literalmente '[eax*2+0]'

3.4 Constantes

NASM entiende cuatro tipos diferentes de constante: numéricas, carácter, cadena y coma flotante.

3.4.1 Constantes numéricas

Una constante numérica es simplemente un número. NASM te permite especificar números en diferentes bases, de diversas maneras: puedes usar un sufijo 'H', 'Q' y 'B' para hexadecimal, octal y binario, o puedes usar un prefijo '0x' para hexadecimal al estilo del C, o poner el prefijo '$' para hexadecimal al estilo del Borland Pascal. Nótese, sin embargo, que el prefijo '$' hace una doble tarea como prefijo de identificadores (ver sección 3.1), por eso un número hexadecimal precedido por un signo '$' debe tener un dígito después del '$' en lugar de una letra.

Algunos ejemplos:

```
mov ax,100            ; decimal
mov ax,0a2h           ; hex
mov ax,$0a2           ; hex de nuevo: el 0 es necesario
mov ax,0xa2           ; hex nuevamente
mov ax,777q           ; octal
mov ax,10010011b      ; binario
```

3.4.2 Constantes carácter

Una constante carácter consiste en hasta cuatro caracteres encerrados entre comillas simples o dobles. NASM no hace distinción entre el tipo de comilla, excepto, por supuesto, que encerrando la constante con comillas simples permite que aparezcan comillas dobles dentro y viceversa.

Una constante carácter con más de un carácter será ordenada con el orden little-endian en mente: si codificas

```
mov eax,'abcd'
```

entonces la constante generada no es '0x61626364', sino '0x64636261', por eso si vas a almacenar el valor en memoria, leerá 'abcd' en lugar de 'dcba'. Este es también el sentido de la comprensión de las constantes carácter de la instrucción 'CPUID' de los Pentium (ver sección A.22).

3.4.3 Constantes cadena

Las constantes cadena sólo son aceptables en algunas

pseudo-instrucciones, a saber, la familia 'DB' e 'INCBIN'.

Una constante cadena se parece a una constante carácter, sólo que más larga. Es tratada en las mismas condiciones que una concatenación de constantes carácter del máximo tamaño. Por eso, lo siguiente es equivalente:

```
db 'hello'              ; constante cadena
db 'h','e','l','l','o' ; el equivalente en constantes carácter
```

Y lo siguiente también es equivalente:

```
dd 'ninechars'          ; constante cadena de palabra doble
dd 'nine','char','s'    ; se convierte en tres palabras dobles
db 'ninechars',0,0,0    ; y realmente parece esto
```

Dése cuenta que cuando se usa como un operando de 'db', una constante como ''ab'' es tratada como una constante cadena a pesar de ser lo suficientemente pequeña como para ser una constante carácter, porque de otro modo, 'db 'ab'' tendría el mismo efecto que 'db 'a'', lo que sería estúpido. De igual forma, constantes de tres o cuatro caracteres se tratan como cadenas cuando son operandos de 'dw'.

3.4.4 Constantes de coma flotante

Las constantes de coma flotante sólo son aceptables como argumentos de 'DD', 'DQ' y 'DT'. Estas son expresadas de la forma tradicional: dígitos, luego el período, y opcionalmente más dígitos, luego opcionalmente una 'E' seguida de un exponente. El período es obligatorio, por lo que NASM puede distinguir entre 'dd 1', que declara una constante entera, y 'dd 1.0' que declara una constante de coma flotante.

Algunos ejemplos:

```
dd 1.2                  ; una fácil
dq 1.e10                ; 10,000,000,000
dq 1.e+10               ; igual a 1.e10
dq 1.e-10               ; 0.000 000 000 1
dt 3.14159265358979323846 ; pi
```

NASM, en tiempo de compilación, no puede hacer aritmética sobre constantes de coma flotante. Esto es debido a que NASM está diseñado para ser portable - aunque siempre genera código que funciona bajo procesadores x86, el ensamblador en sí mismo puede ejecutarse en cualquier sistema con un compilador ANSI C. Por lo tanto, el ensamblador no te puede garantizar la presencia de una unidad de coma flotante que sea capaz de manejar el formato de Intel de los números, y por eso NASM para poder hacer aritmética con coma flotante tendría que incluir su propio conjunto de rutinas de coma flotante, lo que incrementaría significativamente el tamaño del ensamblador para tan poco beneficio.

3.5 Expresiones

Las expresiones en NASM son similares en sintaxis a las de C.

NASM no garantiza el tamaño de los enteros usados para evaluar expresiones en tiempo de compilación: ya que NASM puede compilar y ejecutarse en sistemas de 64-bit bastante bien, no asume que las expresiones son evaluadas en registros de 32-bit, por eso hace un uso deliberado del desbordamiento de entero. Esto podría no funcionar

siempre. Lo único que NASM garantiza es lo que garantiza el ANSI C: siempre tienes para trabajar _al menos_ 32-bit.

NASM implementa en las expresiones dos marcas especiales, permitiendo cálculos que involucran a la actual posición del ensamblador: el '$' y el '$$'. '$' evalúa la actual posición del ensamblador al comienzo de la línea que contiene la expresión; por eso puedes codificar un bucle infinito usando 'JMP $'. '$$' evalúa al comienzo de la actual sección; por eso puedes indicar hasta dónde dentro de la sección en la que estás, usando '($-$$)'.

Los operadores aritméticos proporcionados por NASM son listados aquí, en orden creciente de precedencia.

3.5.1 '|': Operador OR a nivel de bit

El operador '|' proporciona un OR a nivel de bit, exactamente tal y como lo hace la instrucción 'OR' de la máquina. El OR de bit es el de menor prioridad de entre los operadores aritméticos soportados por NASM.

3.5.2 '^': Operador XOR a nivel de bit

'^' proporciona la operación XOR entre bits.

3.5.3 '&': Operador AND a nivel de bit

'&' proporciona la operación AND entre bits.

3.5.4 '<<' y '>>': Operadores de desplazamiento a nivel de bit

'<<' hace un desplazamiento de bits hacia la izquierda, al igual que en C. Por eso, '5<<3' evalúa 5 por 8,esto es 40. '>>' hace un desplazamiento de bits hacia la derecha; en NASM, como un desplazamiento es _siempre_ sin signo, entonces los bits desplazados desde la parte izquierda serán rellenados con ceros en lugar de con una extensión de signo del bit superior.

3.5.5 '+' y '-': Operadores de adición y sustracción

Los operadores '+' y '-' hacen perfectamente la suma y resta ordinarias.

3.5.6 '*', '/', '//', '%' y '%%': Multiplicación y división

'*' es el operador de multiplicación. '/' y '//' son ambos operadores de división: '/' es división sin signo y '//' es división con signo. Igualmente, '%' y '%%' son los operadores de módulo sin signo y con signo respectivamente.

NASM, al igual que en ANSI C, no garantiza nada en la operación lógica del operador módulo con signo.

Puesto que el caracter '%' se usa mucho por el preprocesador de macros, deberías asegurarte que tanto el operador de módulo con signo y sin signo van seguidos de un espacio en blanco siempre que aparezcan.

3.5.7 Operadores unitarios: '+', '-', '~' y 'SEG'

Los operadores de mayor prioridad en la gramática de las expresiones de NASM son aquellos que sólo se aplican a un argumento. '-' niega su operando, '+' no hace nada (se provee por simetría con '-'), '~' calcula

el complemento a uno de su operando, y 'SEG' proporciona la dirección del segmento de su operando (explicado con más detalle en la sección 3.6).

3.6 'SEG' y 'WRT'

Cuando se escriben programas largos de 16-bit, casi siempre han de partirse en múltiples segmentos, a menudo es necesario poder referirse a la parte del segmento de la dirección de un símbolo. NASM proporciona el operador 'SEG' para hacer esta función.

El operador 'SEG' devuelve el segmento base _preferente_ de un símbolo, definido como la base del segmento relativa a la cual tiene sentido el offset del símbolo. Por eso el código

```
mov ax,seg symbol
mov es.ax
mov bx,symbol
```

cargará 'ES:BX' con un puntero válido al símbolo 'symbol'.

Las cosas pueden ser más complejas que esto: ya que los segmentos y los grupos se pueden superponer, ocasionalmente podrías querer referirte a un símbolo usando un segmento base desde el preferente. NASM te permite hacerlo, usando la palabra clave 'WRT' (With Reference To). Por eso puedes hacer cosas como ésta

```
mov ax,weird_seg       ;weird_seg es el segmento base
mov es,ax
mov bx,symbol wrt weird_seg
```

para cargar 'ES:BX' con un puntero diferente al símbolo 'symbol', pero equivalente funcionalmente.

NASM soporta llamadas y saltos lejanos (entre segmentos) mediante la sintaxis 'call segment:offset', donde 'segment' y 'offset' representan valores inmediatos. Por eso, para llamar a un procedimiento lejano, deberías codificar alguna de éstas

```
call (seg procedure):procedure
call weird_seg:(procedure wrt weird_seg)
```

(Los paréntesis se incluyen simplemente para clarificar, para mostrar la interpretación intencionada de las instrucciones arriba mencionadas. En la práctica no son necesarios.)

NASM soporta la sintaxis 'call far procedure' como sinónimo para la primera de las arriba usadas 'JMP' trabaja de forma idéntica a 'CALL' en estos ejemplos.

Para declarar un puntero lejano a un dato en un segmento de datos, debes codificar

```
dw symbol, seg symbol
```

NASM no soporta ningún sinónimos conveniente para esto, sin embargo siempre puedes inventarte uno usando el procesador de macros.

3.7 Expresiones críticas

Una limitación de NASM es que es un ensamblador de dos pasadas; al

contrario que TASM y demás, siempre hará exactamente dos pasadas de ensamblaje. Por lo tanto, es incapaz de hacer frente con ficheros fuente con la suficiente complejidad que requieran tres o más pasadas.

La primera pasada se usa para determinar el tamaño de todo el código y datos, por eso la segunda pasada, cuando se generará todo el código, conoce todas las direcciones de los símbolos a los que se refiere el código. Por eso, una cosa que NASM no puede manejar es el código cuyo tamaño dependa del valor de un símbolo declarado después del código en cuestión. Por ejemplo,

```
          time (label-$) db 0
label:    db 'Where am I?'
```

El argumento de 'TIMES' en este caso podría no evaluar nada de forma igualmente legal; NASM rechazará este ejemplo porque no te puede decir el tamaño de la línea 'TIMES' cuando la ve por primera vez. Rechazará firmemente el siguiente código ligeramente paradójico

```
          times (label-$+1) db 0
label:    db 'NOW where am I?'
```

en la cual _cualquier_ valor como argumento de 'TIMES' será erróneo por definición.

NASM rechaza estos ejemplos mediante un concepto llamado _expresión crítica_, que se define como una expresión cuyos valores se requieren en una primera pasada para poder computarse, y que por lo tanto deben depender sólo de símbolos definidos antes. El argumento del prefijo 'TIMES' es una expresión crítica; por la misma razón, los argumentos de la familia de pseudo-instrucciones 'RESB' son también expresiones críticas.

Las expresiones críticas pueden presentarse en otros contextos como: considere el siguiente código.

```
          mov a,symbol1
symbol1   equ symbol2
symbol2:
```

En la primera pasada, NASM no puede determinar el valor de 'symbol1', porque 'symbol1' se le define para que sea igual a 'symbol2' que NASM todavía no ha visto. En la segunda pasada, por lo tanto, cuando se encuentre la línea 'mov ax,symbol1', es incapaz de generar el código para ella porque todavía no sabe el valor de 'symbol1'. En la siguiente línea, podría ver el 'EQU' otra vez y ser capaz de determinar el valor de 'symbol1', pero para entonces será demasiado tarde.

NASM evita este problema definiendo la parte derecha de una sentencia 'EQU' como expresión crítica, por eso la definición de 'symbol1' será rechazada en la primera pasada.

Hay un asunto relacionado que involucra las siguiente referencias: considere el siguiente fragmento de código.

```
          mov eax,[eb+offset]
offset    equ 10
```

NASM, en la primera pasada, debe calcular el tamaño de la instrucción 'mov eax,[ebx+offset]' sin saber el valor del 'offset'. No tiene forma

de saber qué 'offset' es lo suficientemente pequeño para caber en un campo offset de un byte y que podría por lo tanto continuar generando una forma más corta de la codificación de la dirección efectiva; por lo que sabe, en la primera pasada, 'offset' podría ser un símbolo en el segmento de código, y podría necesitar la forma completa de los cuatro bytes. Por eso está forzado a calcular el tamaño de la instrucción para acomodar una parte de dirección de cuatro bytes. En la segunda pasada, habiendo hecho esta decisión, está forzada a respetarla y mantiene la instrucción grande, por eso el código generado en este caso no es tan pequeño como podría haber sido. Este problema puede resolverse definiendo 'offset' antes de usarlo, o forzando un tamaño de byte en la dirección efectiva codificando '[byte ebx+offset]'.

3.8 Etiquetas locales

NASM da un trato especial a los símbolos que empiezan por un punto. Una etiqueta que comience con un simple punto se trata como una etiqueta _local_, lo que quiere decir que está asociada a la etiqueta no local previa. Por eso, por ejemplo:

```
label1    ; algo de código
.loop     ; algo de código más
          jne .loop
          ret
label2    ; algo de código
.loop     ; algo de código más
          jne .loop
          ret
```

En este fragmento de código, cada instrucción 'JNE' salta a la línea inmediatamente superior, porque las dos definiciones de '.loop' son mantenidas de forma separada en virtud de que cada una está asociada a la etiqueta no local previa.

Esta forma de manejar las etiquetas locales ha sido tomada prestada del viejo ensamblador de Amiga, DevPac; sin embargo, NASM va un paso más allá, permitiendo acceder a las etiquetas locales desde otras partes de código. Esto se ha logrado mediante la _definición_ de una etiqueta local en términos de la etiqueta no local previa: la primera definición de '.loop' vista anteriormente está realmente definiendo un símbolo llamado 'label1.loop', y la segunda define un símbolo llamado 'label2.loop'. Por eso, si realmente lo necesitas, podrías escribir

```
label3    ; algo de código más
          ; y un poco más
          jmp label1.loop
```

Algunas veces es útil - en una macro, por ejemplo - el poder definir una etiqueta que pueda ser referenciada desde cualquier lugar pero que no interfiera con el mecanismo normal de etiquetas locales. Tal y como, una etiqueta no puede ser no local porque interferiría con definiciones subsecuentes de, y referencias a, etiquetas locales; y no puede ser local porque la macro que la define no sabría el nombre completo de la etiqueta. NASM, por lo tanto, introduce un tercer tipo de etiqueta, que es útil posiblemente sólo en las definiciones de macros: si una etiqueta comienza con el prefijo especial '..@', entonces no hace nada al mecanismo de la etiqueta local. Por eso podrías codificar

```
label1:   ; una etiqueta no local
.local:   ; esto es realmente label1.local
```

```
..@foo:   ; esto es un símbolo especial
label2:   ; otra etiqueta no local
.local:   ; esto es realmente label2.local
          jmp ..@foo            ; esto saltará tres líneas hacia arriba
```

NASM tiene la capacidad de definir otros símbolos especiales empezando con un punto doble: por ejemplo, '..start' es usado para especificar el punto de entrada en el formato 'obj' de salida (ver sección 6.2.6).

Capítulo 4: El preprocesador NASM

NASM contiene un poderoso procesador de macros, que soporta ensamblaje condicional, inclusión multi-nivel de ficheros, dos formatos de macros (simple y multi-nivel), y un mecanismo de pila de contexto para una potencia extra de macros. Las directivas del preprocesador comienzan todas por el signo '%'.

4.1 Macros de una línea

4.1.1 La forma normal: '%define'

Las macros de una simple línea son definidas usando la directiva del preprocesador '%define'. Las definiciones trabajan de un modo similar a como lo hace C; por eso puedes hacer cosas como

```
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))
          mov byte [param(2,exb)], ctrl 'D'
```

que se expandirá a

```
          mov byte [(2)+(2)*(eb)], 01F & 'D'
```

Cuando la expansión de una macro de una sólo línea contiene señales que invoquen otras macros, la expansión se realiza en tiempo de invocación, no en el momento de la definición. Así, el código

```
%define a(x) 1+b(x)
%define b(x) 2*x
          mov ax,a(8)
```

evaluará de la forma esperada 'mov ax,1+2*8', incluso aunque la macro 'b' no estuviese definida en el momento de la definición de 'a'.

Las macros definidas con '%define' son sensibles a mayúsculas: después de '%define foo bar', sólo 'foo' se expandirá a 'bar': 'Foo' o 'FOO' no lo harán. Usando '%idefine' en lugar de '%define' (la 'i' se pone por 'insensitive') puedes definir de una vez todas las variaciones de mayúsculas y minúsculas de una macro, por eso '%idefine foo bar' haría que 'foo', Foo', 'FOO', 'fOO' se expandan a 'bar'.

Hay un mecanismo que detecta cuando ha ocurrido una llamada de macro como resultado de una expansión previa de la misma macro, para salvarse de referencias circulares y bucles infinitos. Si esto pasa, el preprocesador expandirá sólo la primera ocurrencia de una macro. Por lo tanto, si codificas

```
%define a(x) 1+a(x)
          mov ax,a(3)
```

la macro 'a(3)' se expandirá una vez, siendo '1+a(3)', y no se expandirá más. Este comportamiento puede ser útil: ver sección 8.1 para ver un ejemplo de su uso.

Puedes sobrecargar macros de una sólo línea: si escribes

```
%define foo(x) 1+x
%define foo(x,y) 1+x*y
```

el preprocesador será capaz de manejar ambos tipos de llamadas de macro, contando los parámetros que pases; por eso 'foo(3)' será '1+3' mientras que 'foo(ebx,2)' será '1+ebx*2'. Sin embargo, si defines

```
%define foo bar
```

no se aceptará ninguna otra definición de 'foo': una macro sin parámetros prohíbe la definición de una macro con el mismo nombre _con_ parámetros, y viceversa.

Puedes predefinir macros de una línea usando la opción '-d' en la línea de comandos de NASM: ver sección 2.1.7.

4.1.2 Variables del preprocesador: '%assign'

Una forma alternativa de definir macros simples es mediante el comando '%assign' (y su contrapartida insensible a mayúsculas '%iassign', que difiere de '%assign' de la misma manera que difiere '%idefine' de '%define').

'%assign' se usa para definir macros de una sola línea que no tomen parámetros y tengan un valor numérico. Este valor puede especificarse en la forma de una expresión, y será evaluado una vez, cuando sea procesada la directiva '%assign'.

'%assign' es útil para controlar el final de lo bucles '%rep' del preprocesador: ver sección 4.4 para ver un ejemplo de esto. Otro uso de '%assign' se da en las secciones 7.4 y 8.1.

La expresión pasada a '%assign' es una expresión crítica (ver sección 3.7), y debe evaluar un número puro (en lugar de una referencia relocalizable como podría ser una dirección de código o segmento, o cualquier cosa que involucre a un registro).

4.2 Macros multi-nivel: '%macro'

Las macros multi-nivel son mucho más parecidas a las macros vistas en MASM y TASM: una definición de una macro multi-nivel en NASM se parece a algo similar a esto.

```
%macro prologue 1
          push ebp
          mov ebp,esp
          sub esp,%1
%endmacro
```

esto define, de forma parecida al C, una función 'prologue' como una macro: por eso podrías invocar la macro con una llamada como ésta

```
myfunc:   prologue 12
```

que se expandirá a las tres líneas de código

```
myfunc:   push ebp
          mov ebp,esp
          sub esp,12
```

El número '1' después del nombre de la macro en la línea '%macro' define el número de parámetros que la macro 'prologue' espera recibir. El uso de '%1' dentro de la definición de la macro se refiere al primer parámetro de la llamada a la macro. Con una macro que tome más de un parámetro, los subsiguientes parámetros serían referidos como '%2', '%3', y así.

Las macros multi-línea, al igual que las macros de una sólo línea, son sensibles a mayúsculas, a menos que las definas usando la directiva alternativa '%imacro'.

Si necesitas pasar una coma como _parte_ de un parámetro en una macro multi-nivel, puedes hacerlo encerrando el parámetro entero entre llaves. Por eso, podrías codificar cosas como

```
%macro silly 2
%2:       db %1
%endmacro
          silly 'a', letter_a    ; letter_a:  db 'a'
          silly 'ab', string_ab  ; string_ab: db 'ab'
          silly {13,10}, crlf    ; crlf:      db 13,10
```

4.2.1 Sobrecargando macros multi-línea

Al igual que con las macros de una sólo línea, las macros multi-línea puedes sobrecargarse definiendo el mismo nombre de macro varias veces con diferente número de parámetros. Esta vez, no hay excepciones en las macros que no tienen parámetros. Por eso puedes definir

```
%macro prologue 0
          push ebp
          mov ebp,esp
%endmacro
```

para definir una forma alternativa de la función 'prologue' que no coloca espacio de pila.

Algunas veces, sin embargo, podrías querer sobrecargar una instrucción máquina; por ejemplo, podrías querer definir

```
%macro push 2
          push %1
          push %2
%endmacro
```

por eso podrías codificar

```
          push ebx               ; esta línea no es una llamada de macro
          push eax,ecx           ; pero ésta sí
```

Generalmente, NASM dará un aviso para la primera de las dos líneas, ya que 'push' está ahora definida como una macro, y está siendo invocada con un número de parámetros para los cuales no se ha dado definición.

El código correcto todavía será generado, pero el ensamblador dará un aviso. Este aviso puede desactivarse usando la opción de la línea de comando '-w-macro-params' (ver sección 2.1.10).

4.2.2 Etiquetas locales de macros

NASM te permite definir etiquetas dentro de la definición de una macro multi-línea de modo que la hacemos local a la llamada de la macro: por eso llamar a la misma macro múltiples veces usará una etiqueta diferente cada vez. Puedes hacer esto usando el prefijo '%%' en el nombre de la macro. Por eso, puedes inventar una instrucción que ejecute una 'RET' si el flag 'Z' se activa, haciendo esto:

```
%macro retz 0
          jnz %%skip
          ret
%skip:
%endmacro
```

Puedes llamar a esta macro tantas veces como quieras, y cada vez que la llames NASM creará un nombre 'real' diferente para sustituir la etiqueta '%%skip'. Los nombre que NASM inventa son de los forma '..@2354.skip', donde el número 2345 cambia con cada llamada de macro. El prefijo '..@' previene a las etiquetas locales de macros de interferir con el mecanismo de etiquetas locales, tal y como ha sido descrito en la sección 3.8. Deberías evitar definir tus propias etiquetas de esta forma (el prefijo '..@'. luego un número, luego otro punto) en el caso que interfieran con las etiquetas locales de la macro.

4.2.3 Parámetros de macros avariciosas

A veces, es útil definir una macro que amontone toda su línea de comandos en un definición de parámetro, posiblemente después de extraer uno o dos parámetros pequeños de delante. Un ejemplo podría ser una macro que escriba una cadena de texto a un fichero en MS-DOS, donde podrías querer ser capaz de escribir

```
          writefile [filehandle],"hello, world",13,10
```

NASM te permite definir el último parámetro de una macro para que sea _avariciosa_, lo que quiere decir que invocas la macro con más parámetros de los que espera, todos los parámetros sobrantes son amontonados en el último definido junto con las comas de separación. Por eso, si codificas:

```
%macro writefile 2+
          jmp %%endstr
%%str:    db %2
%%endstr: mov dx,%%str
          mov cx,%%endstr-%%str
          mov bx,%1
          mov ah,0x40
          int 0x21
%endmacro
```

entonces la llamada de ejemplo al 'writefile' de arriba trabajará como se espera: el texto antes de la primera coma, '[filehandle], se usa como primer parámetro de la macro y se expando cuando haya una referencia a '%1', y todo el texto subsiguiente se amontona en '%2' y es colocado después de 'db'.

La naturaleza avariciosa de la macro se indica a NASM usando el signo '+' después de la cuenta de parámetros en la línea '%macro'.

Si defines una macro avariciosa, efectivamente le estás diciendo a NASM cómo deberías expandir la macro dándola _cualquier_ número de parámetros desde el número actual especificado hasta el infinito; en este caso, por ejemplo, NASM ahora sabe qué hacer cuando vea una llamada a 'writefile' con 2, 3, 4 ó más parámetros. NASM tendrá esto en cuenta cuando haya sobrecarga de macros, y no te permitirá definir otra forma de 'writefile' tomando 4 parámetros (por ejemplo).

Por supuesto, la macro arriba mencionada podría haber sido implementada como una macro no avariciosa, en cuyo caso la llamada tendría que ser algo así como

```
          writefile [filehandle], {"hello, world",13,10}
```

NASM proporciona ambos mecanismos para poner comas en los parámetros de las macros, y tu eliges cuál prefieres para cada definición de macros.

Ver las sección 5.2.1 para ver un mejor modo de escribir la macro de arriba.

4.2.4 Parámetros por defecto de las macros

NASM también te permite definir una macro multi-línea con un _rango_ de parámetros permitidos. Si haces esto, puedes especificar los valores por defecto de los parámetros omitidos. Por eso, por ejemplo,

```
%macro die 0-1 "Painful program death has occurred."
          writefile 2,%1
          mov ax,0x4c01
          int 0x21
%endmacro
```

Esta macro (que hace uso de la macro 'writefile' definida en la sección 4.2.3) puede ser llamada con un mensaje de error explícito, que mostrará en la salida de errores antes de salir, o puede llamarse sin parámetros, en cuyo caso usará el mensaje de error proporcionada en la definición de la macro.

En general, tú facilitas un mínimo y un máximo número de parámetros para una macro de este tipo; en la llamada de la macro se requiere el mínimo número de parámetros, y entonces proporcionas los valores por defecto de los opcionales. Por eso, si una definición de macro comienza con la línea

```
%macro foobar 1-3 eax,[ebx+2]
```

entonces podría ser llamada con entre uno y tres parámetros, y '%1' siempre será tomada de la llamada de la macro. '%2', si no está especificado en la llamada a la macro, se usará por defecto 'eax', y si '%3' tampoco está especificado usaría por defecto '[ebx+2]'.

Podría omitir los parámetros por defecto desde la definición de la macro, en cuyo caso, los parámetros por defecto se considerarán en blanco. Esto puede ser útil para las macros que puedan recibir un número variable de parámetros, ya que '%0' (ver sección 4.2.5) te permite determinar cuántos parámetros han sido realmente pasados a la llamada de la macro.

Este mecanismo de valores por defecto puede combinarse con el mecanismo

de avaricia de parámetros; por eso, la macro 'die' de arriba podría hacerse más potente, y más útil, cambiando la primera línea de la definición a

```
%macro die 0-1+ "Painful program death has occurred.",13,10
```

El máximo número de parámetros puede ser infinito, denotado con '*'. En este caso, por supuesto, es imposible proporcionar un conjunto _completo_ de parámetros por defecto. Ejemplos de este uso se muestran en la sección 4.2.6

4.2.5 '%0': Contador de parámetros de macro

Para una macro que pueda recibir un número variable de parámetros, la referencia de parámetro '%0' devolverá una constante numérica dando el número de parámetros pasados a la macro. Esto puede usarse como un argumento para '%rep' (ver sección 4.4) en vistas a iterar a través de todos los parámetros de una macro. Se dan ejemplos en la sección 4.2.6.

4.2.6 '%rotate': Rotando los parámetros de macro

Los programadores de shells de Unix estarán familiarizados con el comando 'shift' del shell, que permite mover un lugar a la izquierda los argumentos pasados a un shell script (referenciados como '$1', '$2' etc.)
,
por eso el argumento previamente referenciado como '$2' pasa a estar disponible como '$1', y el argumento que previamente estaba referenciado como '$1' ya no está disponible en absoluto.

NASM proporciona un mecanismo similar, en forma de '%rotate'. Tal y como su nombre sugiere, difiere del 'shift' de Unix en que no se pierden los parámetros: los parámetros rotados fuera del lado izquierdo de la lista de argumentos reaparecen en la derecha, y viceversa.

'%rotate' se invoca con un argumento numérico simple (que podría ser una expresión). Los parámetros de macro son rotados a la izquierda tantos lugares como indique este valor. Si el argumento de '%rotate' es negativo, los parámetros de la macro son rotados a la derecha.

Por eso un par de macro para salvar y restaurar un conjunto de registros podría trabajar como sigue:

```
%macro multipush 1-*
%rep %0
          push %1
%rotate 1
%endrep
%endmacro
```

Esta macro invoca la instrucción 'PUSH' en cada de uno de sus argumentos por turno, desde la derecha a la izquierda. Comienza empujando su primer argumento, '%1', luego invoca '%rotate' para mover todos los argumentos un lugar a la izquierda, por eso, el segundo argumento original ahora está disponible como '%1'. Repitiendo este proceso tantas veces como argumentos haya (que se consigue suministrando '%0' como argumento de '%rep') se consigue que cada argumento sea empujado por turno.

Dése cuenta también del uso de '*' como máximo número de parámetros, indicando que no hay límite superior en el número de parámetros que puedes proporcionar en la macro 'multipush'.

Sería conveniente, cuando se usa esta macro, el tener un 'POP' equivalente, que _no_ requiera que los argumentos se den en orden inverso. Idealmente, podría escribir una llamada a un 'multipush', luego cortar y pegar la línea allí donde se necesite que se haga un pop, y cambiar el nombre de la macro a 'multipop', y luego la macro se encargaría de sacar los registros en el orden opuesto al que han sido empujados.

esto se puede hacer con la siguiente definición:

```
%macro multipop 1-*
%rep %0
%rotate -1
          pop %1
%endrep
%endmacro
```

Esta macro comienza rotando sus argumentos un lugar a la _derecha_, por eso el _último_ argumento original aparece como '%1'. Entonces, éste es sacado, y los argumentos son rotados a la derecha otra vez, por eso el segundo por el final pasa a ser '%1'. Así se van iterando los argumentos en orden inverso.

4.2.7 Concatenando los parámetros de macro

NASM puede concatenar los parámetros de macro en otro texto que lo envuelva. Esto te permite declarar una familia de símbolos, por ejemplo, en una definición de macro. Si, por ejemplo, quieres generar una tabla de códigos de tecla además de los offsets en la tabla, podrías codificar algo como

```
%macro keytab_entry 2
keypos%1 equ $-keytab
          db %2
%endmacro
keytab:
          keytab_entry F1,128+1
          keytab_entry F2,128+2
          keytab_entry Return,13
```

se expandiría a

```
keytab:
keyposF1 equ $-keytab
          db 128+1
keyposF2 equ $-keytab
          db 128+2
keyposReturn equ $-keytab
          db 13
```

Puedes concatenar texto fácilmente al final de otro parámetro de macro, escribiendo '%1foo'.

Si necesitas unir un _dígito_ a un parámetro de macro, por ejemplo definiendo etiquetas 'foo1' y 'foo2' cuando se pasa el parámetro 'foo', no puedes codificar '%$11' porque sería tomado como el undécimo parámetro de macro. En su lugar, debes codificar '%{1}1', que separará el primer '1' (dando el número de parámetro de macro' del segundo (texto literal para ser concatenado al parámetro).

Esta concatenación también puede aplicarse a otros objetos en línea de preprocesador, como etiquetas locales de macro (sección 4.2.2) y etiquetas locales de contexto (sección 4.6.2). En todos los casos, las ambigüedades en la sintaxis se pueden resolver encerrando entre llaves todo lo que va después del signo '%' y antes del texto literal: por eso '%{%foo}bar' concatena el texto 'bar' al final del nombre real de la etiqueta local de macro '%%foo'. (Esto es innecesario, ya que la forma que NASM usa para los nombres reales de las etiquetas locales de macros implica que los dos usos '%{%foo}bar' y '%%foobar' serían expandidos de cualquier forma a la misma cosa; no obstante, la capacidad está ahí.)

4.2.8 Códigos de condición como parámetros de macro

NASM puede dar un trato especial a un parámetro de macro que contenga un código de condición. En principio, te puedes referir a un parámetro de macro '%1' con la sintaxis alternativa '%+1', que informa a NASM que este parámetro de macro se supone que contiene código de condición, y hará que el preprocesador informe un mensaje de error si la macro es llamada con un parámetro que _no_ sea un código de condición válido.

Incluso más útil todavía, creo, te puedes referir al parámetro de macro con '%-1', el cual NASM expandirá como el código de condición _inverso_. Por eso la macro 'retz' definida en la sección 4.2.2 puede sustituirse por una macro general de retorno condicional como ésta:

```
%macro retc 1
          j%-1 %%skip
          ret
%%skip:
%endmacro
```

Esta macro puede invocarse ahora usando llamadas como 'retc ne', que hará que la instrucción de salto condicional en la expansión de la macro aparezca como 'JE', o 'retc po' que hará del salto un 'JPE'.

La referencia de parámetro de macro '%+1' interpreta felizmente los argumentos 'CXZ' y 'ECXZ' como códigos de condición válidos; sin embargo, '%-1' informará de un error si pasa cualquiera de estos, porque no existen códigos de condición inversa.

4.2.9 Deshabilitando la expansión de listado

Cuando NASM está generando un fichero de listado de tu programa, generalmente expande las macro multi-línea por medio de escribir la llamada a la macro y luego listando cada una de las líneas de la expansión. Esto te permite ver qué instrucciones en la expansión de la macro está generando qué código; sin embargo, para algunas macro esto atesta innecesariamente el listado.

Por lo tanto NASM proporciona el calificador '.nolist', que puedes incluir en un definición de macro para inhibir la expansión de la macro en el fichero de lista. El calificador '.nolist' va inmediatamente después del número de parámetros, así:

```
%macro foo 1.nolist
```

O así:

```
%macro bar 1-5+.nolist a,b,c,d,e,f,g,h
```

4.3 Ensamblaje condicional

Al igual que el preprocesador de C, NASM permite que haya secciones de un fichero fuente que se ensamblen sólo si se cumplen ciertas condiciones. La sintaxis general de esta característica es algo así:

```
%if<condition>
; algo de código que sólo aparece si se cumple <condition>
%elif<condition2>
; sólo aparece si no se cumple <condition> pero sí <condition2>
%else
; esto aparece si no se cumple ninguna de las condiciones <condition> y
; <condition2>
%endif
```

La cláusula '%else' es opcional, al igual que la cláusula '%elif'. Puedes tener más de una cláusula '%elif'.

4.3.1 '%ifdef': Comprobando la existencia de una macro de una línea

Comenzando el ensamblaje de un bloque condicional con la línea '%ifdef MACRO' ensamblará el subsiguiente código si, y sólo si, la macro de una línea llamada 'MACRO' está definida. Si no, entonces los bloques '%elif' y '%else' (si los hay) serán procesados en su lugar.

Por ejemplo, cuando depuras un programa, podrías querer escribir código como

```
          ; realiza algunas funciones
%ifdef DEBUG
          writefile 2,"Function performed successfully",13,10
%endif
          ; ir y hacer algo más
```

Entonces podrías usar la opción '-dDEBUG' de la línea de comandos para crear una versión del programa que produzca mensajes de depuración, y quite la opción para generar la versión final del programa.

Puedes comprobar una macro que _no_ haya sido definida usando '%ifndef' en lugar de '%ifdef'. También puedes comprobar definiciones de macros en los bloques '%elif' usando '%elifdef' y '%elifndef'.

4.3.2 '%ifctx': Comprobando la pila de contexto

La construcción de ensamblaje condional '%ifctx ctxname' hará que el código subsiguiente sea ensamblado si y sólo si el contexto superior en la pila de contexto del preprocesador tiene el nombre 'crxname'. Al igual que con '%ifdef', el inverso y las formas '%elif', '%ifnctx', '%elifctx' y '%elifnctx' también están implementadas.

Para más detalles sobre la pila de contexto, ver la sección 4.6. Para ver un ejemplo del uso de 'ifctx', ver la sección 4.6.5.

4.3.3 '%if': Comprobando expresiones numéricas arbitrarias

La construcción de ensamblaje condicional '%if expr' hará que el código subsiguiente se ensamble si y sólo si el valor de la expresión numérica 'expr' no es cero. Un ejemplo del uso de esta característica está en decidir cuándo abortar un bucle '%rep' del preprocesador: ver la sección

4.4 para un ejemplo detallado.

La expresión dada a '%if', y su contrapartida '%elif', es una expresión crítica (ver sección 3.7).

'%if' extiende la sintaxis normal de las expresiones NASM, proporcionando un conjunto de operadores relacionados que no están normalmente disponibles en las expresiones. Los operadores '=', '<', '>', '<=', '>=' y '<>' comprueban la igualdad, menor que, mayor que, menor o igual que, mayor o igual que y distinto, respectivamente. La formas del estilo de C tales como '==' y '!=' están implementadas como formas alternativas a '=' y '<>'. Adicionalmente, se proveen los operadores lógicos de menor prioridad '&&', '^^' y '||', suministrando el AND lógico, XOR lógico y el OR lógico. Estos funcionan como los operadores lógicos de C (aunque C no tiene XOR lógico), y siempre devuelven 0 ó 1, y tratan cualquier entrada que no sea cero como un 1 (por eso '^^', por ejemplo, devuelve 1 si exactamente una de los entradas es cero, y 0 de otra forma). Los operadores relacionales también devuelven 1 para el cierto y 0 para el falso.

4.3.4 '%ifidn' y '%ifidni': Comprobando la identidad exacta de textos

La construcción '%ifidn text1,text2' hará que le subsiguiente código se ensamble si y sólo si 'text1' y 'text2', tras expandir las macros de una sola línea, son piezas idénticas de texto. Las diferencias en los espacios en blanco no se tiene en cuenta.

'%ifidni' es parecido a '%ifidn', pero es sensible a mayúsculas.

Por ejemplo, la siguiente macro empuja un registro o número en la pila, y te permite tratar 'IP' como un registro real:

```
%macro pushparam 1
%ifidni %1,ip
          call %%label
%%label:
%else
          push %1
%endif
%endmacro
```

Al igual que otras construcciones '%if', '%ifidn' tiene una contrapartida '%elifidn', y formas negativas '%ifnidn' y '%elifnidn'. De forma similar, '%ifidni' tiene contrapartidas '%elifidni', '%ifnidni' y '%elifnidni'.

4.3.5 '%ifid', '%ifnum', '%ifstr': Comprobando los tipos de los objetos

Algunas macros querrán realizar diferentes tareas dependiendo de si se las pasa un número, una cadena, o un identificador. Por ejemplo, una macro de salida de cadena podría querer ser capaz de funcionar si se la pasa tanto una constante de cadena como un puntero a una cadena ya existente.

La construcción condicional '%ifid', recibe un parámetro (que podría ser un blanco), ensambla el código subsiguiente si y sólo si el primer objeto en el parámetro existe y es un identificador. '%ifnum' trabaja de forma similar, pero comprueba si el objeto es una constante numérica; '%ifstr' comprueba si es una cadena.

Por ejemplo, la macro 'writefile' definida en la sección 4.2.3 puede

extenderse para tomar ventaja de '%ifstr' de la siguiente manera:

```
%macro writefile 2-3+
%ifstr %2
          jmp %%endstr
%if %0 = 3
%%str:     db %2,%3
%else
%%str:     db %2
%endif
%%endstr: mov dx,%%str
          mov cx,%%endstr-%%str
%else
   mov dx,%2
   mov cx,%3
%endif
          mov bx,%1
          mov ah,0x40
          int 0x21
%endmacro
```

Entonces la macro 'writefile' funcionará si es llamado de cualquiera de las siguiente formas:

```
          writefile [file], strpointer, length
          writefile [file], "hello, 13, 10
```

Primero, 'strpointer' se usa como la dirección de una cadena ya declarada, y 'length' es usado como su longitud; segundo, a la macro se la entrega una cadena, la cual por lo tanto se declara a sí misma y elabora la dirección y longitud por sí misma.

Nótese el uso de '%if' dentro de '%ifstr': esto es para detectar si a la macro se la pasan dos argumentos (por eso, la cadena podría ser una simple constante de cadena, y 'db %2' sería adecuado) o más (en cuyo caso, todo excepto los dos primeros podrían agruparse en '%3', y se requeriría 'db %2,%3').

Las versiones usuales '%elifXXX', '%ifnXXX' y '%elifnXXX' existen para cada '%ifid', '%ifnum' y '%ifstr'.

4.3.6 '%error': Informando errores definidos por el usuario

La directiva del preprocesador '%error' hará que NASM informe de un error si ocurre en el código ensamblado. Por eso, si otros usuarios intentar ensamblar tus ficheros fuente, puedes asegurarte de que definen las macros correctas mediante código como éste:

```
%ifdef SOME_MACRO
; hacer algo de configuración
%elifdef SOME_OTHER_MACRO
; hacer otra configuración diferente
%else
%error Neither SOME_MACRO nor SOME_OTHER_MACRO was defined.
%endif
```

Entonces, cualquier otro usuario que no entienda el modo en que se supone que tu código fuente debe ser ensamblado rápidamente percibirá su error, en lugar de tener que esperar a que el programa se venga abajo cuando sea ejecutado y no sepa qué está mal.

4.4 Bucles del preprocesador: '%rep'

El prefijo 'TIMES' de NASM, aunque útil, no puede usarse para invocar varias veces una macro multi-línea, porque NASM lo procesa después de que las macros hayan sido expandidas. Por lo tanto, NASM proporciona otra forma de bucle, esta vez a nivel de preprocesador: '%rep'.

Las directivas '%rep' y '%endrep' ('%rep' recive un argumento numérico, que puede ser una expresión; '%endrep' no recibe argumentos) pueden usarse para encerrar un trozo de código, que es replicado por el preprocesador tantas veces como sea especificado:

```
%assign i 0
%rep 64
          inc word [table+2*i]
%assign i i+1
%endrep
```

Esto generará una secuencia de 64 instrucciones 'INC', incrementando cada palabra de memoria desde '[table]' hasta '[table+126]'.

Para condiciones de terminación más complejas, o para abortar un bucle, puedes usar la directiva '%exitrep' para terminar el bucle, como esto:

```
fibonacci:
%assign i 0
%assign j 1
%rep 100
%if j > 65535
%exitrep
%endif
          dw j
%assign k j+i
%assign i j
%assign j k
%endrep
fib_number equ ($-fibonacci)/2
```

Esto produce una lista de todos los número Fibonacci que quepan en 16 bits. Nótese que todavía es necesario dar un número máximo de repeticiones a '%rep'. Esto es para prevenir la posibilidad de que NASM se meta en un bucle infinito en el preprocesador, que (en sistemas multitarea y multiusuario) típicamente hará que la memoria se vaya usando y otras aplicaciones empiecen a fallar.

4.5 Incluyendo otros ficheros

Usando, otra vez, una sintaxis muy parecida a la del preprocesador de C, el preprocesador de NASM te permite incluir otros ficheros fuente dentro de tu código. Este se hace usando la directiva '%include'.

```
%include "macros.mac"
```

incluirá el contenido del fichero 'macros.mac' dentro del fichero fuente que contenga la directiva '%include'.

Los ficheros include se buscan en el directorio actual (el directorio en el que estabas cuando ejecutaste NASM, en lugar de en la localización del ejecutable NASM o la localización del fichero fuente), más cualquier otro

directorio especificado en la línea de comandos del NASM usando la opción '-i'.

El idioma estándar de C para prevenir que un fichero se incluya más de una vez es aplicable en NASM: si un fichero 'macros.mac' tiene la forma

```
%ifndef MACROS_MAC
%define MACROS_MAC
; ahora define algunas macros
%endif
```

entonces incluir el fichero más de una vez no provocará error, porque la segunda vez que el fichero es incluido no pasará nada porque la macro 'MACROS_MAC' ya está definida.

Puedes forzar a que un fichero sea incluido incluso si no hay directiva '%include' que lo incluya explícitamente, usando la opción '-p' en la línea de comandos de NASM (ver sección 2.1.6).

4.6 La pila de contexto

El tener etiquetas que son locales a una definición de macro no es a veces lo suficientemente potente: algunas veces quieres ser capaz de compartir etiquetas entre varias llamadas de macros. Un ejemplo podría ser un bucle 'REPEAT' ... 'UNTIL', en el cual, la expansión de la macro 'REPEAT' podría necesitar ser capaz de referirse a un etiqueta que la macro 'UNTIL' haya definido. Sin embargo, para tales macros querrías poder anidar estos bucles.

NASM proporciona este nivel de potencia mediante una _pila de contexto_. El preprocesador mantiene una pila de _contextos_, cada uno de los cuales está caracterizado por un nombre. Usando la directiva '%push' añades un nuevo contexto a la pila, y quitas uno usando '%pop'. Puedes definir etiquetas de la pila que sean locales a un contexto en particular.

4.6.1 '%push' y '%pop': Creando y destruyendo contextos

La directiva '%push' se usa para crear un nuevo contexto y colocarlo en la parte superior de la pila de contexto. '%push' requiere un argumento, que es el nombre del contexto. Por ejemplo:

```
%push foobar
```

Esto empuja un nuevo contexto llamado 'foobar' en la pila. Puedes tener varios contextos en la pila con el mismo nombre: aun así se pueden distinguir.

La directiva '%pop', no necesita argumentos, quita el contexto de la parte superior de la pila de contexto y lo destruye, junto con las etiquetas asociadas con él.

4.6.2 Etiquetas de contexto local

Al igual que en el uso de '%%foo' define una etiqueta que es local a una llamada de macro en particular en la cual se usa, la costumbre es usar '%%foo' para definir una etiqueta que sea local al contexto de la parte de arriba de la pila. Por eso, el ejemplo de 'REPEAT' y 'UNTIL' dado arriba, podría implementarse mediante:

```
%macro repeat 0
```

```
%push repeat
%$begin:
%endmacro

%macro until 1
          j%-1 %$begin
%pop
%endmacro
```

e invocarlo mediante, por ejemplo,

```
          mov cx,string
          repeat
          add cx,3
          scasb
          until e
```

que buscaría todos los cuartos bytes de una cadena en la búsqueda del byte en 'AL'.

Si necesitas definir, o acceder, etiquetas locales al contexto que está _debajo_ del de la parte superior de la pila, puedes usar '%$$foo' o '%$$$foo' para acceder al contexto que está debajo, y así.

4.6.3 Macros de una línea y de contexto local

NASM también te permite definir macros de una línea que sean locales a un contexto en particular, de la misma manera:

```
%define %$localmac 3
```

definirá la macro de una línea '%$localmac' como local al contexto de la parte superior de la pila. Por supuesto, después de un '%push' subsiguiente, todavía podrá ser accedida por el nombre '%$$localmac'.

4.6.4 '%repl': Renombrando un contexto

Si necesitas cambiar el nombre del contexto de la parte superior de la pila (para responder, por ejemplo, de manera diferente a '%ifctx'), puede
s
ejecutar un '%pop' seguido de un '%push'; pero esto tendrá el efecto lateral de destruir todas las etiquetas de contexto local y las macros asociadas con el contexto que acaba de ser sacado.

NASM proporciona la directiva '%repl', que _sustituye_ un contexto con un nombre diferente, sin tocar las macros y etiquetas asociadas. Por eso podrías sustituir el código destructivo

```
%pop
%push newname
```

con la versión no destructiva '%repl newname'.

4.6.5 Ejemplo de uso de la pila de contexto: Bloque IFs

Este ejemplo hace uso de casi todas las características la pila de contexto, incluyendo la construcción de ensamblaje condicional '%ifctx', para implementar un bloque de sentencia IF como un conjunto de macros.

```
%macro if 1
```

```
    %push if
    j%-1 %$ifnot
%endmacro

%macro else 0
    %ifctx if
        %repl else
        jmp %$ifend
        %$ifnot:
    %else
        %error "expected `if' before `else'"
    %endif
%endmacro

%macro endif 0
    %ifctx if
        %$ifnot:
        %pop
    %elifctx else
        %$ifend:
        %pop
    %else
        %error "expected `if' or `else' before `endif'"
    %endif
%endmacro
```

Este código es más robusto que las macros 'REPEAT' y 'UNTIL' dadas en la sección 4.6.2, porque usa ensamblaje condicional para comprobar que las macros se llaman en el orden correcto (por ejemplo, no llamando a 'endif' antes que 'if') y emitiendo un '%error' si no.

Adicionalmente, la macro 'endif' tiene que ser capaz de hacer frente a los casos distintos de seguir directamente a un 'if', o seguir a un 'else'. Se consigue esto, de nuevo, usando ensamblaje condicional para hacer cosas diferentes dependiendo de si el contexto de la parte superior de la pila es un 'if' o un 'else'.

La macro 'else' tiene que preservar el contexto de la pila, para tener el '%$ifnot' referido para, por la macro 'if', ser la misma que la definida por la macro 'endif', pero tiene que cambiar el nombre del contexto, por eso, ese 'endif' sabrá que hay interviniendo un 'else'. Esto se hace con el uso de '%repl'.

Un ejemplo de uso de estas macros podría ser como:

```
          cmp ax,bx
          if ae
            cmp bx,cx
            if ae
              mov ax,cx
            else
              mov ax,bx
            endif
          else
            cmp ax,cx
            if ae
              mov ax,cx
            endif
          endif
```

Las macros de bloque 'IF' se anidan bastante fácilmente, empujando otro contexto, describiendo el 'if' más interno, encima del que describe el 'if' de la parte más exterior; por eso, 'else' y 'endif' siempre se refieren al último 'if' o 'else' no emparejado.

4.7 Macros estándar

NASM define un conjunto de macros estándar, que están ya definidas cuando empieza a procesar cualquier fichero fuente. Si realmente necesitas que un programa se ensamble sin las macros predefinidas, puedes hacer uso de la directiva '%clear' para vaciar el preprocesador de cualquier cosa.

La mayoría de las directivas a nivel de usuario (ver capítulo 5) están implementadas como macros que invocan directivas primitivas; estas están descritas en el capítulo 5. El resto del conjunto de macros estándar se describe aquí.

4.7.1 '__NASM_MAJOR__' y '__NASM_MINOR__': Versión del NASM

Las macros de una línea '__NASM_MAJOR__' y '__NASM_MINOR__' se expanden a las partes principal y secundaria del número de versión del NASM que está siendo usado. Por eso, bajo NASM 0.96, por ejemplo, '__NASM_MAJOR__' sería definida como 0 y '__NASM_MINOR__' sería definida como 96.

4.7.2 '__FILE__' y '__LINE__': Nombre de fichero y número de líneas

Al igual que el preprocesador de C, NASM permite al usuario encontrar el nombre de fichero y el número de línea que contiene la instrucción actual. La macro '__FILE__' se expande a una constante cadena dando el nombre del fichero de entrada actual (que podría cambiar a lo largo de un ensamblaje si se usan directivas '%include'), y '__LINE__' se expande a una constante numérica dando la línea actual en el fichero de entrada.

Estas macros podrían usarse, por ejemplo, para comunicar información de depuración a una macro, ya que invocando '__LINE__' dentro de una definición de macro (ya sea de una línea o de múltiples líneas) devolverá el número de línea de la _llamada_ a la macro, en lugar de la _definición_. Por eso, para determinar dónde está fallando un trozo de código, por ejemplo, uno podría escribir una rutina 'stillhere', que recibe un número de línea en 'EAX' y devuelve algo como 'line 155: still here'. Podrías escribir una macro

```
%macro notdeadyet 0
          push eax
          mov eax,__LINE__
          call stillhere
          pop eax
%endmacro
```

y luego salpicar tu código con llamadas a 'notdeadyet' hasta que encuentres el punto del fallo.

4.7.3 'STRUC' y 'ENDSTRUC': Declarando tipos de estructura de datos

El núcleo de NASM no contiene medios intrínsecos para definir estructuras de datos; en su lugar, el preprocesador es lo suficientemente potente que las estructuras de datos pueden implementarse como un conjunto de macros. Las macros 'STRUC' y 'ENDSTRUC' se usan para definir un tipo estructura de datos.

'STRUC' recibe un parámetro, que es el nombre del tipo de dato. Este nombre está definido como un símbolo con el valor cero, y también tiene añadido el sufijo '_size' y entonces es definido como un 'EQU' dando el tamaño de la estructura. Una vez que 'STRUC' ha sido emitido, estás definiendo la estructura, y deberías definir los campos usando la familia 'RESB' de pseudo-instrucciones, y luego invocar 'ENDSTRUC' para terminar la definición.

Por ejemplo, para definir una estructura llamada 'mytype' conteniendo una longword, una word, un byte y una cadena de bytes, podrías codificar

```
          struc mytype
mt_long:  resb 1
mt_word:  resw 1
mt_byte:  resb 1
mt_str:   resb 32
          endstruc
```

Este código define seis símbolos: 'mt_long' como 0 (el offset del principio de la estructura 'mytype' al campo longword), 'mt_word' como 4, 'mt_byte' como 6, 'mt_str' como 7, 'mytype_size' como 39, y 'mytype' como cero.

La razón de por qué el nombre del tipo estructura se define como cero es un efecto lateral de permitir a las estructuras trabajar con el mecanismo de las etiquetas locales: si los miembros de tu estructura tienden a tener los mismos nombres en más de una estructura, puedes definir la estructura de arriba como esto:

```
          struc mytype
.long:    resd 1
.word:    resw 1
.byte:    resb 1
.str:     resb 32
          endstruc
```

Esto define los offsets a los campos de la estructura como 'mytype.long', 'mytype.word', 'mytype.byte' y 'mytype.str'.

NASM, ya que no tiene _intrínseco_ el soporte de estructuras, no soporta ninguna forma de notación periódica para referirse a un elemento de una estructura una vez tengas una (excepto la notación de etiquetas locales arriba mencionado), por eso, código como 'mov ax,[mystruc.mt_word] no es válido. 'mt_word' es una constante como cualquier otra constante, por eso la sintaxis correcta es 'mov ax,[mystruc+mt_word]' o 'mov ax,[mystruc+mytype.word]'.

4.7.4 'ISTRUC', 'AT' and 'IEND': Declarando instancias de estructuras

Teniendo definida un tipo estructura, la siguiente cosa que típicamente querrás hacer es declarar instancias de esa estructura en tu segmento de datos. NASM proporciona un modo fácil de hacer esto en el mecanismo 'ISTRUC'. Para declarar una estructura del tipo 'mytype' en un programa, codificas algo similar a esto:

```
mystruc:  istruc mytype
          at mt_long, dd 123456
          at mt_word, dw 1024
          at mt_byte, db 'x'
          at mt_str, db 'hello, world', 13, 10, 0
```

```
          iend
```

La función de la macro 'AT' es hacer uso del prefijo 'TIMES' para avanzar la posición de ensamblaje al punto correcto del campo especificado de la estructura, y luego declarar el dato especificado. Por lo tanto los campos de estructura deben ser declarados en el mismo orden en que son especificados en la definición de la estructura.

Si el dato que va en el campo de la estructura requiere más de una línea del fuente para ser especificado, las restantes línea del fuente pueden fácilmente venir después de la línea 'AT'. Por ejemplo:

```
          at mt_str, db 123,134,145,156,167,178,189
          db 190,100,0
```

Dependiendo del gusto personal, puedes también omitir completamente la parte de código de la línea 'AT', y empezar el campo de la estructura en la siguiente línea:

```
          at mt_str
          db 'hello, world'
          db 13,10,0
```

4.7.5 'ALIGN' y 'ALIGNB': Alineamiento de datos

Las macros 'ALIGN' y 'ALIGNB' proporcionan un modo conveniente de alinear el código o los datos a una palabra, palabra larga, párrafo u otro límite. (Algunos ensambladores llaman a esta directiva 'EVEN'.) La sintaxis de las macros 'ALIGN' y 'ALIGNB' es

```
          align 4               ; alinéa en el límite de 4-bytes
          align 16              ; alinéa en el límite de 16-bytes
          align 8,db 0          ; rellena con 0 en lugar de con NOPs
          align 4.resb 1        ; alinéa a 4 en el BSS
          alignb 4              ; equivalente a la línea anterior
```

Ambas macros requieren que su primer argumento sea una potencia de dos; ambas calculan el número de bytes adicionales requeridos para traer la longitud de la sección actual a un múltiplo de esa potencia de dos, y luego aplican el prefijo 'TIMES' a su segundo argumento para realizar el alineamiento.

Si no se especifica el segundo argumento, por defecto para 'ALIGN' es 'NOP', y por defecto para 'ALIGNB' es 'RESB 1'. Por eso si está especificado el segundo argumento, las dos macros son equivalentes. Normalmente, simplemente puedes usar 'ALIGN' en las secciones de código y datos, y 'ALIGNB' en las secciones BSS, y nunca necesitar el segundo argumento excepto para propósitos especiales.

'ALIGN' y 'ALIGNB', al ser macros sencillas, no realizan ninguna comprobación de errores: no te pueden avisar si su primer argumento falla el ser una potencia de dos, o si el segundo argumento genera más de un byte de código. En cada uno de estos casos hará en silencio la cosa equivocada.

'ALIGNB' ('ALIGN' con un segundo argumento 'RESB 1') pueden usarse dentro de definiciones de estructuras:

```
          struc mytype2
mt_byte:  resb 1
```

```
            alignb 2
mt_word:    resw 1
            alignb 4
mt_long:    resd 1
mt_str:     resb 32
            endstruc
```

Esto asegurará que los miembros de la estructura están alineados sensatamente en relación a la base de la estructura.

Un comentario final: 'ALIGN' y 'ALIGNB' trabajan en relación al principio de la _sección_, no al principio del espacio de dirección del ejecutable final. El alineamiento a un límite de 16-byte cuando en la sección en la que estás sólo garantiza el alineamiento a un límite de 4-byte, por ejemplo, es un esfuerzo perdido. De nuevo, NASM no comprueba que las características del alineamiento de la sección sean correctas para el uso de 'ALIGN' y 'ALIGNB'.

Capítulo 5: Directivas del ensamblador

NASM, aunque intenta evitar la burocracia de ensambladores como MASM o TASM, está sin embargo forzado a soportar unas _pocas_ directivas. Estas están descritas en este capítulo.

Las directivas de NASM viene en dos tipos: directivas a nivel de usuario y directivas primitivas. Típicamente, cada directiva tiene una forma a nivel de usuario y una forma primitiva. En casi todos los casos, recomendamos que los usuarios usen las formas a nivel de usuario de las directivas, que están implementadas como macros que llaman a las formas primitivas.

Las directivas primitivas están encerradas entre corchetes; las directivas a nivel de usuarios, no.

Adicionalmente a las directivas universales descritas en este capítulo, cada formato de fichero objeto puede suministrar opcionalmente directivas extra para controlar características particulares de ese formato de fichero. Estas directivas específicas del formato de fichero están documentadas además de los formatos que las implementas, en el capítulo 6.

5.1 'BITS': Especificando el modo de proceso del destino

La directiva 'BITS' especifica si NASM debería generar código diseñado para correr en un procesador operando en modo de 16-bit, o código diseñado para correr en un procesador operando en modo de 32-bit. La sintaxis es 'BITS 16' or 'BITS 32'.

En la mayoría de los casos, no deberías necesitar usar 'BITS' explícitamente. Los formatos objeto 'aout', 'coff', 'elf' y 'win32', que están diseñados para ser usados en sistemas operativos de 32-bit, hacen que NASM seleccione por defecto el modo de 32-bit. El formato objeto 'obj' te permite especificar cada segmento que definas como 'USE16' o 'USE32', y NASM establecerá de acuerdo a esto su modo de operación, por eso, el uso de la directiva 'BITS' es una vez más innecesario.

La mejor razón para usar la directiva 'BITS' es para escribir código 32-bit en un fichero binario plano; esto es debido a que el formato de salida 'bin' está por defecto en el modo de 16-bit en anticipación a que

va a ser usado con mayor frecuencia para escribir programas '.COM' del DOS, controladores de dispositivo '.SYS' y software cargador de arranque.

_No_ necesitas especificar 'BITS 32' simplemente para usar instrucciones de 32-bit en un programa DOS de 16-bit; si lo haces, el ensamblador generará código incorrecto porque estará escribiendo código destino en una plataforma de 32-bit, para ser ejecutado en una de 16-bit.

Cuando NASM está en el estado 'BITS 16', las instrucciones que usen datos de 32-bit son prefijadas con un byte 0x66, y aquellas que se refieran a direcciones de 32-bit tendrán el prefijo 0x67. En el estado 'BITS 32', lo contrario es cierto: las instrucciones de 32-bit no requieren prefijo, sin embargo las instrucciones que usen datos de 16-bit necesitan un 0x66 y aquellas que trabajen en direcciones de 16-bit necesitan un 0x67.

La directiva 'BITS' tienen una forma primitiva exactamente equivalente, '[BITS 16]' y '[BITS 32]'. La forma a nivel de usuario es una macro que no tiene otra función que llamar a la forma primitiva.

5.2 'SECTION' o 'SEGMENT': Cambiando y definiendo secciones

La directiva 'SECTION' ('SEGMENT' es un sinónimo exactamente equivalente) cambia la sección del fichero de salida en la que va a ser ensamblado el código que escribas. En algunos formatos de fichero objeto, el número y nombre de las secciones es fijo; en otros, el usuario puede inventarse tantas como quiera. Por lo tanto, 'SECTION' podría a veces dar un mensaje de error, o podría definir una nueva sección, si intentas cambiar a una sección que (todavía) no existe.

Tanto los formatos objeto de Unix como el formato objeto 'bin' soportan los nombres de sección estandarizados '.text', '.data' y '.bss' para las secciones de código, datos y datos no inicializados. El formato 'obj', al contrario, no reconoce como especiales estos nombres de sección, y de hecho desmontará el período previo de cualquier nombre de sección que tenga uno.

5.2.1 La macro '__SECT__'

La directiva 'SECTION' es inusual ya que las funciones de su forma a nivel de usuario difieren de su forma primitiva. La forma primitiva, '[SECTION xyz]', simplemente cambia la sección destino actual por la dada. La forma del nivel usuario, 'SECTION xyz', sin embargo, define primero la macro de una línea '__SECT__' como la directiva primitiva de '[SECTION]' que es para emitir, y luego la emite. Por eso la directiva a nivel usuario

```
          SECTION .text
```

se expande a las dos líneas

```
%define __SECT__ [SECTION .text]
          [SECTION .text]
```

Los usuarios pueden encontrar útil hacer uso de esto en sus propias macros. Por ejemplo, la macro 'writefile' definida en la sección 4.2.3 puede reescribirse de forma más útil de la siguiente forma mucho más sofisticada:

```
%macro writefile 2+
          [section .data]
```

```
%%str:    db %2
%%endstr:
          __SECT__
          mov dx,%%str
          mov cx,%%endstr-%%str
          mov bx,%1
          mov ah,0x40
          int 0x21
%endmacro
```

Esta forma de la macro, una vez pasada una cadena a la salida, primero cambia temporalmente a la sección de datos del fichero, usando la forma primitiva de la directiva 'SECTION' y no modifica '__SECT__'. Entonces declara su cadena en la sección de datos, y luego invoca '__SECT__' para cambiar de vuelta a _cualquier_ sección en la que el usuario estuviese trabajando. De este modo evita la necesidad, en la versión anterior de la macro, de incluir una instrucción 'JMP' para saltar al dato, y no falla si, en un complicado módulo de formato 'OBJ', el usuario podría estar potencialmente ensamblando el código en varias secciones de código separadas.

5.3 'ABSOLUTE': Definiendo etiquetas absolutas

La directiva 'ABSOLUTE' puede pensarse como una forma alternativa de 'SECTION': hace que el código subsiguiente no sea dirigido a ninguna sección física, sino a la hipotética sección que comienza en la dirección absoluta dada. Las únicas instrucciones que puedes usar en este modo son la familia 'RESB'.

'ABSOLUTE' se usa como sigue:

```
          absolute 0x1A
kbuf_chr  resw 1
kbuf_free resw 1
kbuf      resw 16
```

Este ejemplo describe una sección del área de datos de la BIOS PC, en la dirección de segmento 0x40: el código de arriba define 'kbuf_chr' como 0x1A, 'kbuf_free' como 0x1C, y 'kbuf' como 0x1E.

La forma a nivel de usuario de 'ABSOLUTE', al igual que 'SECTION', redefine la macro '__SECT__' cuando es invocado.

'STRUC' y 'ENDSTRUC' están definidas como macros que usan 'ABSOLUTE' (y también '__SECT__').

'ABSOLUTE' no tiene que coger una constante absoluta como un argumento: puede coger una expresión (actualmente, una expresión crítica: ver la la sección 3.7) y puede ser un valor en un segmento. Por ejemplo, un TSR puede reutilizar su código de inicialización como un BSS en tiempo de ejecución como éste:

```
          org 100h              ; es un programa .COM
          jmp setup             ; el código de inicio viene al final
          ; aquí viene la parte residente del TSR
setup:    ;ahora escribe aquí el código que instala el TSR
          absolute setup
runtimevar1 resw 1
runtimevar2 resd 20
tsr_end:
```

Esto define algunas variables 'encima de' el código de inicio, por eso después que la inicialización ha terminado de ejecutarse, el espacio que tomó puede reutilizarse como almacenamiento de datos para el TSR que se está ejecutando. El símbolo 'tsr_end' puede usarse para calcular el tamaño total de la parte del TSR que se necesita hacer residente.

5.4 'EXTERN': Importando símbolos desde otros módulos

'EXTERN' es similar a la directiva de NASM 'EXTRN' y la palabra clave de C 'extern': se usa para declarar un símbolo que no está definido en ningún sitio del módulo que está siendo ensamblado, pero que se asume que que está definido en algún otro módulo y necesita ser referenciado por éste. No todos los formatos de fichero objeto soportan variables externas: el formato 'bin' no puede.

La directiva 'EXTERN' recibe tantos argumentos como quieras. Cada argumento es el nombre de un símbolo:

```
          extern _printf
          extern _sscanf,_fscanf
```

Algunos formatos de fichero objeto proporcionan características extras a la directiva 'EXTERN'. En todos los casos, las características extras son usadas poniendo como sufijo dos puntos al nombre del símbolo seguido por el texto específico del formato objeto. Por ejemplo, el formato 'obj' te permite declarar el segmento base por defecto de un externo debería ser el grupo 'dgroup' mediante la directiva

```
          extern _variable:wrt dgroup
```

La forma primitiva de 'EXTERN' difiere de la forma a nivel de usuario en que aquella sólo puede recibir un argumento cada vez: el soporte para múltiples argumentos está implementado a nivel de procesador.

Puedes declarar la misma variable como 'EXTERN' más de una vez: NASM tranquilamente ignorará la segunda y posteriores declaraciones. Sin embargo, no puedes declarar una variable como 'EXTERN' y además como algo más.

5.5 'GLOBAL': Exportando símbolos desde otros módulos

'GLOBAL' es el otro lado de 'EXTERN': si un módulo declara un símbolo como 'EXTERN' y se refiere a él, entonces, para prevenir errores del enlazador, algunos otros módulos deben _definir_ actualmente el símbolo y declararlo como 'GLOBAL'. Algunos ensambladores usan el nombre 'PUBLIC' para este mismo propósito.

La directiva 'GLOBAL' aplicada a un símbolo debe aparecer _antes_ que la definición del símbolo.

'GLOBAL' usa la misma sintaxis que 'EXTERN', excepto que debe referirse a símbolos que _están_ definidos en el mismo módulo con la directiva 'GLOBAL'. Por ejemplo:

```
          global _main
_main     ; algo de código
```

'GLOBAL', al igual que 'EXTERN', permite a los formatos objeto definir extensiones privadas mediante dos puntos. El formato objeto 'elf', por

ejemplo, te permite especificar si los datos globales son funciones o datos:

```
global hashlookup:function, hashtable:data
```

Al igual que 'EXTERN', la forma primitiva de 'GLOBAL' difiere de la forma a nivel de usuario en que sólo puede recibir un argumento cada vez.

5.6 'COMMON': Definiendo áreas de datos comunes

La directiva 'COMMON' se usa para declarar _variables comunes_. Una variable común es como una variable global declarada en la sección de datos no inicializados, por eso

```
common intvar 4
```

es similar en funcionamiento a

```
global intvar
section .bss
intvar    resd 1
```

La diferencia es que si más de un módulo definen la misma variable común, entonces en tiempo de enlace estas variables serán _unidas_, y las referencias a 'intvar' en todos los módulos apuntarán al mismo lugar de memoria.

Al igual que 'GLOBAL' y 'EXTERN', 'COMMON' soporta extensiones específicas de los formatos objeto. Por ejemplo, el formato 'obj' permite a las variables comunes ser NEAR o FAR, y el formato 'elf' te permite especificar los requerimientos del alineamiento de una variable común:

```
common commvar 4:near  ; funciona en OBJ
common intarray 100:4  ; funciona en ELF: 4 bytes alineados
```

Una vez más, al igual que 'EXTERN' y 'GLOBAL', la forma primitiva de 'COMMON' difiere de la forma a nivel de usuario en que sólo puede recibir un argumento cada vez.

Capítulo 6: Formatos de salida

NASM es un ensamblador portable, diseñado para ser capaz de compilar en cualquier plataforma que soporte ANSI C y produce salidas que corren en una variedad de sistemas operativos de Intel x86. Por esta razón, tiene disponibles un gran número de formatos de salida, seleccionados usando la opción '-f' en la línea de comandos de NASM. Cada uno de estos formatos, además de sus extensions en la sintaxis base de NASM, son detallados en este capítulo.

Según se mencionó en la sección 3.1.1, NASM elige un nombre por defecto para tu fichero de salida basado en el fichero de entrada y el formato de salida elegido. Este será generado quitando la extensión ('.asm', '.s' o cualquiera que sea la que uses) del nombre del fichero de entrada, y sustituyéndola por la extensión definida por el formato de salida. Las extensiones están dadas aquí abajo con cada formato.

6.1 'bin': Salida binaria de forma plana

El formato 'bin' no produce ficheros objeto: no genera nada en el fichero

de salida excepto el código que escribas. Por eso, los ficheros de 'binario puro' son usados por MS-DOS: los ejecutables '.COM' y los controladores de dispositivo '.SYS' son ficheros de binario puro. La salida de binario puro también es útil para el desarrollo de sistemas operativos y de cargadores de arranque.

'bin' soporta solamente los tres nombres de sección estandarizados '.text', '.data' y '.bss'. El fichero que NASM saca contendrá primero el contenido de la sección '.text', seguido del contenido de la sección '.data', alineado con un límite de 4-bytes. La sección '.bss' no se almacena para nada en el fichero de salida, pero se asume que aparece directamente después del final de la sección '.data', nuevamente alineado en un límite de 4-bytes.

Si no especificas explícitamente la directiva 'SECTION', el código que escribas será dirigido por defecto a la sección '.text'.

Al usar el formato 'bin' se pone a NASM por defecto en el modo 16-bit (ver la sección 5.1). Para usar 'bin' para escribir código 32-bit como un núcleo de sistema operativo, necesitas mandar explícitamente la directiva 'BITS 32'.

'bin' no tiene por defecto extensión del nombre del fichero de salida: en su lugar, deja el nombre de tu fichero tal cual está una vez que haya sido quitada la extensión original. Por eso, por defecto está que NASM ensamble 'binprog.asm' en un fichero binario llamado 'binprog'.

6.1.1 'ORG': Origen del programa del fichero binario

El formato 'bin' proporciona una directiva adicional a la lista dada en el capítulo 5: 'ORG'. La función de la directiva 'ORG' es especificar la dirección de origen en la cual NASM asume que comienza el programa cuando se carga en memoria.

Por ejemplo, el siguiente código generará la palabra larga '0x00000104':

```
          org 0x100
          dd label
label:
```

Al contrario que la directiva 'ORG' proporcionada por los ensambladores compatibles con MASM, que te permiten saltar por el fichero objeto y sobreescribir código que ya has generado, el 'ORG' de NASM hace exactamente lo que dice la directiva: _origen_. Su única función es especificar un offset que es añadido a todas las referencias de direcciones internas dentro del fichero; no permite ninguno de los trucos que las versiones MASM permiten. Ver la sección 10.1.3 para más comentarios.

6.1.2 Extensiones 'bin' de la directiva 'SECTION'

El formato de salida 'out' extiende la directiva 'SECTION' (o 'SEGMENT') para permitirte especificar los requerimientos de alineamiento de los segmentos. Esto se hace añadiendo el calificador 'ALIGN' al final de la línea de definición de sección. Por ejemplo,

```
          section .data align=16
```

cambia a la sección '.data' y también especifica que debe alinearse con un límite de 16-bytes.

El parámetro de 'ALIGN' especifica cuántos bits bajos de la dirección de comienzo de la sección deben forzarse a cero. El valor dado del alineamiento debería ser una potencia de dos.

6.2 'obj': Ficheros objeto Microsoft OMF

El formato de fichero 'obj' (NASM lo llama 'obj' en lugar de 'omf' por razones históricas) es el producido por MASM y TASM, que se suministra a los enlazadores de 16-bits del DOS para producir ficheros '.EXE'. Este es también el formato usado por OS/2.

'obj' proporciona por defecto la extensión '.obj' para el nombre del fichero de salida.

'obj' no es un formato exclusivo de 16-bits, esto es: NASM soporta totalmente las extensiones de 32-bits del formato. En particular, los ficheros de formato 'obj' de 32-bits se usan por los compilador Win32 de Borland, en lugar de usar el nuevo formato de fichero objeto de Microsoft, 'win32'.

El formato 'obj' no define ningún nombre especial de segmento: puedes llamar a tus segmentos como quieras. Nombres típicos de segmentos en los ficheros de formato 'obj' son 'CODE', 'DATA' y 'BSS'.

Si tu fichero fuente contiene código antes de especificar una directiva 'SEGMENT' explícita, entonces NASM se inventará por ti su propio segmento llamado '__NASMDEFSEG'.

Cuando defines un segmento en un fichero 'obj', NASM define el nombre del segmento como un símbolo, por eso puedes acceder a la dirección de segmento del segmento. Así, por ejemplo:

```
          segment data
dvar:     dw 1234
          segmento code
function: mov ax,data           ; coge la dirección de segmento de data
          mov ds,ax             ; y la mueve a DS
          inc word [dvar]       ; ahora esta referencia funcionará
          ret
```

El formato 'obj' también permite el uso de los operadores 'SEG' y 'WRT', por eso puedes escribir código que haga cosas como

```
          extern foo
          mov ax,seg foo        ; como el segmento preferido de foo
          mov ds,ax
          mov ax,data           ; un segmento diferente
          mov es,ax
          mov ax,[ds:foo]       ; esto accede a 'foo'
          mov [es:foo wrt data],bx  ; también hace esto
```

6.2.1 Extensiones 'obj' de la directiva 'SEGMENT'

El formato de salida 'obj' extiende la directiva 'SEGMENT' (o 'SECTION') para permitirte especificar varias propiedades del segmento que estás definiendo. Esto se hace añadiendo calificadores al final de la línea de definición del segmento. Por ejemplo,

```
          segment code private align=16
```

define el segmento 'code', pero también lo declara como segmento privado, y requiere que la porción descrita en este módulo de código esté alineado en un límite de 16-byte.

Los calificadores disponibles son:

(*) 'PRIVATE', 'PUBLIC', 'COMMON' y 'STACK' especifican las características combinadas del segmento. Los segmento 'PRIVATE' no son combinados por el enlazador con ningún otro; los segmentos 'PUBLIC' y 'STACK' se concatenan juntos en tiempo de enlace; y los segmentos 'COMMON' son todos superpuestos en la parte superior de cada uno de los otros en lugar de pegarlos final con final.

(*) 'ALIGN' se usa, como se ha visto arriba, para especificar cuántos bits bajos de la dirección de comienzo de segmento deben forzarse a cero. El valor del alineamiento debería ser cualquier potencia de dos desde 1 a 4096; en realidad, los únicos valores soportados son 1, 2, 4, 16, 256 y 4096, por eso si especificas 8 será redondeado hasta 16, y 32, 64 y 128 serán redondeados hasta 256, y así. Nótese que el alineamiento en límite de 4096-bytes es una extensión de PharLap del formato y podría no ser soportada por todos los enlazadores.

(*) 'CLASS' puede usarse para especificar la clase del segmento; esta característica indica al enlazador que los segmentos de la misma clase deberían colocarse cerca unos de otros en fichero de salida. El nombre de la clase puede ser cualquier palabra, por ejemplo 'CLASS=CODE'.

(*) 'OVERLAY', al igual que 'CLASS', se especifica con una palabra arbitraria como argumento, y proporciona información de solapamiento para un enlazador capaz de solapar.

(*) Los segmentos se pueden declarar como 'USE16' o 'USE32', que tiene el efecto de grabar la elección en el fichero objeto y también de asegurar que el modo de ensamblaje por defecto de NASM, cuando se está ensamblando en este segmento, es 16-bit o 32-bit respectivamente.

(*) Cuando escribas ficheros objeto de OS/2, deberías declarar los segmentos de 32-bits como 'FLAT', lo que hará que el segmento base por defecto para cualquier cosa en el segmento sea del grupo especial 'FLAT', y también define el grupo si no está ya definido.

(*) El formato de fichero 'obj' también permite a los segmentos ser declarados teniendo predefinido una dirección de segmento absoluta, aunque no se conocen enlazadores que actualmente hagan un uso sensato de esta característica, no obstante, NASM te permite declarar si lo necesitas un segmento como 'SEGMENT SCREEN ABSOLUTE=0xB800'. Las palabras claves 'ABSOLUTE' y 'ALIGN' son excluyentes entre sí.

Los atributos de segmento por defecto de NASM son 'PUBLIC', 'ALIGN=1', sin clase, sin solapamiento, y 'USE16'.

6.2.2 'GROUP': Definiendo grupos de segmentos

El formato 'obj' también permite agrupar a los segmentos, por esto es que puede usarse un simple registro de segmento para referirse a todos los segmentos en un grupo. Por lo tanto, NASM proporciona la directiva 'GROUP', y puedes codificar

```
segment data
; algunos datos
segment bss
; algunos datos no inicializados
group dgroup data bss
```

lo que definirá un grupo llamado 'dgroup' para contener los segmentos 'data' y 'bss'. Al igual que 'SEGMENT', 'GROUP' hace que el nombre del grupo sea definido como un símbolo, por eso te puedes referir a la variable 'var' en el segmento 'data' como 'var wrt data' o como 'var wrt dgroup', dependiendo de qué valor de segmento está actualmente en tu registro de segmento.

Si simplemente te refieres a 'var', sin embargo, y 'var' está declarada en un segmento que forma parte de un grupo, entonces NASM por defecto te dará el offset de 'var' desde el principio del _grupo_, no del segmento. Por lo tanto, 'SEG var', también devolverá la base del grupo, en lugar de la base del segmento.

NASM permitirá a un segmento ser parte de más de un grupo, pero generará un warning si lo haces. Las variables declaradas en un segmento que forma parte de más de un grupo por defecto será relativas al primer grupo que fue definido para contener el segmento.

Un grupo no tiene por qué contener algún segmento; todavía puedes hacer referencias 'WRT' a un grupo que no contenga la variable a la que te estás refiriendo. OS/2, por ejemplo, define el grupo especial 'FLAT' que no contiene segmentos.

6.2.3 'UPPERCASE': Desactivando en la salida la sensibilidad a mayúsculas

Aunque NASM en sí mismo es sensible a mayúsculas, algunos enlazadores OMF no lo son; por eso puede ser útil para NASM dar ficheros objeto de salida que no sean sensibles a mayúsculas. La directiva específica de formato 'UPPERCASE' hace que todos los nombres de segmentos, grupos y símbolos escritos en el fichero objeto se pasen a mayúsculas antes de escribirse. Dentro de un fichero fuente, NASM todavía es sensible a mayúsculas; pero el fichero objeto puede escribirse por completo si se quiere en mayúsculas.

'UPPERCASE' se usa solo en una línea; no requiere parámetros.

6.2.4 'IMPORT': Importando símbolos DLL

La directiva específica de formato 'IMPORT' define un símbolo que va a ser importado desde una DLL, para usarlo si estás escribiendo en NASM una biblioteca importada de una DLL. Todavía necesitas declarar el símbolo como 'EXTERN' así como usar la directiva 'IMPORT'.

La directiva 'IMPORT' necesita dos parámetros, separados por espacios en blanco, que son (respectivamente) el nombre del símbolo que quieres importar y el nombre de la biblioteca desde la que lo deseas importar. Por ejemplo:

```
import WSAStartup wsock32.dll
```

Un tercer parámetro opcional da el nombre con el cual es conocido el símbolo en la biblioteca desde la que lo estás importando, en este caso no es el mismo que el nombre con el que quieres que sea conocido el

símbolo en tu código una vez que haya sido importado. Por ejemplo:

```
import asyncsel wsock32.dll WSAAsyncSelect
```

6.2.5 'EXPORT': Exportando símbolos DLL

La directiva específica de formato 'EXPORT' define un símbolo global para que sea exportado como un símbolo DLL, para usar si estás escribiendo DLL en NASM. Todavía necesitas declarar el símbolo como 'GLOBAL' así como usar la directiva 'EXPORT'.

'EXPORT' requiere un argumento, que es el nombre del símbolo que deseas exportar, tal y como ha sido definido en tu fichero fuente. Un segundo parámetro opcional (separado del primero por espacios en blanco) da el nombre _externo_ del símbolo: el nombre con el que deseas que sea conocido por los programas que usen la DLL. Si este nombre es el mismo que el nombre interno, deberías dejarlo sin el segundo parámetro.

Se pueden dar otros parámetros para definir los atributos del símbolo exportado. Estos parámetros, como el segundo, están separados por espacios en blanco. Si se dan estos parámetros adicionales, el nombre externo se debe especificar también, incluso si el mismo que el interno. Los atributos disponibles son:

(*) 'resident' indica que el nombre exportado es para que el cargador del sistema lo mantenga como residente. Esto es una optimización para los símbolos importados por nombre que se usan frecuentemente.

(*) 'nodata' indica que el símbolo exportado es una función que no hace uso de ningún dato inicializado.

(*) 'parm=NNN', donde 'NNN' es un entero, establece el número de palabras del parámetro para el caso que el símbolo sea una puerta de llamada entre segmentos de 32 y 16 bits.

(*) Un atributo que simplemente sea un número indica que el símbolo debería exportarse con un número identificativo (ordinal), y de el número deseado.

Por ejemplo:

```
export myfunc
export myfunc TheRealMoreFormalLookingFunctionName
export myfunc myfunc 1234  ; exporta por ordinal
export myfunc myfunc resident parm=23 nodata
```

6.2.6 '..start': Definiendo el punto de entrada del programa

Los enlazadores OMF requieren exactamente que uno de los fichero objeto que están siendo enlazados defina el punto de entrada del programa, donde comenzará la ejecución cuando sea ejecutado el programa. Si el fichero objeto que define el punto de entrada está ensamblado con NASM, especificas el punto de entrada usando el símbolo especial '..start' en el punto en que quieras que comience la ejecución.

6.2.7 Extensiones 'obj' de la directiva 'EXTERN'

Si declaras un símbolo externo con la directiva

```
extern foo
```

entonces, referencias como 'mov ax,foo' te darán el offset de 'foo' desde el segmento base prioritario (tal y como está especificado en cualquier módulo en el que 'foo' haya sido definido). Por eso para acceder al contenido de 'foo' usualmente necesitarás hacer algo como

```
mov ax,seg foo          ; coge el segmento base prioritario
mov es,ax               ; los mueve dentro de ES
mov ax,[es:foo]         ; y usa el offset 'foo' desde él
```

Esto es poco manejable, particularmente si sabes que uno externo va a ser accesible desde un grupo o segmento dado, digamos 'dgroup'. Por eso, si 'DS' ya contiene a 'dgroup', podrías codificar simplemente

```
mov ax,[foo wrt dgroup]
```

Sin embargo, tener que escribir esto cada vez que quieras acceder a 'foo' puede ser doloroso; por eso NASM te permite declarar 'foo' de la forma alternativa

```
extern foo:wrt dgroup
```

Esta forma hace que NAMS pretenda que el segmento base prioritario de 'foo' sea de hecho 'dgroup'; por eso la expresión 'seg foo' ahora devolverá 'dgroup', y la expresión 'foo' es equivalente a 'foo wrt dgroup'.

Este mecanismo 'WRT' por defecto puede usarse para hacer que externos parezcan relativos a cualquier grupo o segmento en tu programa. También pueden aplicarse a variables comunes: ver sección 6.2.8.

6.2.8 Extensiones 'obj' para la directiva 'COMMON'

El formato 'obj' permite que las variables comunes sean cercanas o lejanas; NASM te permite especificar cuál deberían ser tus variables usando la sintaxis

```
common bearvar 2:near  ; 'nearvar' es una común cercana
common farvar 10:far   ; y 'farvar' es lejana
```

Las variables lejanas comunes pueden ser mayores en tamaño que 64Kb, y por eso la especificación OMF dice que son declaradas como un número de _elementos_ de un tamaño dado. Por eso una variables lejana común de 10-bytes podría declararse como diez elementos de un byte, cinco elementos de dos bytes, dos elementos de diez bytes o uno de diez bytes.

Algunos enlazadores OMF requieren el tamaño del elemento, así como el tamaño de la variable, para que coincida cuando se resuelvan las variables lejanas comunes declaradas en más de un módulo. Por lo tanto, NASM debe permitirte especificar el tamaño del elemento en tus variables lejanas comunes. Esto se hace con la siguiente sintaxis:

```
common c_5by2 10:far 5 ; dos elementos de cinco bytes
common c_2by5 10:far 2 ; cinco elementos de dos bytes
```

Si no se especifica el tamaño del elemento, por defecto es 1. Tampoco se requiere la palabra clave 'FAR' cuando se especifica el tamaño de un elemento, ya que sólo las lejanas comunes deberían tener tamaño de los elementos. Por eso, las declaraciones de arriba podrían equivaler a

```
common c_5by2 10:5     ; dos elementos de cinco bytes
common c_2by5 10:2     ; cinco elementos de dos bytes
```

Adicionalemte a estas extensiones, la directiva 'COMMON' en 'obj' también soporta la especificación 'WRT' por defecto al igual que lo hace 'EXTERN' (explicado en la sección 6.2.7). Por eso también puedes declarar las cosas como

```
common foo 10:wrt dgroup
common bar 16:far 2:wrt data
common baz 24:wrt data:6
```

6.3 'win32': Ficheros objeto Win32 de Microsoft

El formato de salida 'win32' genera ficheros objeto del Win32 de Microsoft, indicados para ser pasados a enlazadores de Microsoft como Visual C++. Nótese que los compiladores Win32 de Borland no usan este formato, sino que usan 'obj' en su lugar (ver sección 6.2).

'win32' proporciona una extensión por defecto para el fichero de salida que es '.obj'.

Dése cuenta que aunque Microsoft dice que los ficheros objeto de Win32 siguen el estándar COFF (Common Object File Format), los ficheros objeto producidos por los compiladores Win32 de Microsoft no son compatibles con los enlazadores COFF como DJGPP, y viceversa. Esto es debido a una diferencia de opinión sobre la semántica precisa de las reubicaciones relativas al PC. Para producir ficheros COFF apropiados para DJGPP, usa el formato de salida 'coff' del NASM; a la inversa, el formato 'coff' no produce ficheros objeto que los enlazadores Win32 puedan usar correctamente para generar una salida.

6.3.1 Extensiones 'win32' para la directiva 'SECTION'

Similar al formato 'obj', 'win32' te permite especificar información adicional en la línea de la directiva 'SECTION', para controlar el tipo y las propiedades de las secciones que declares. Los tipos de sección y las propiedades son generados automáticamente por NASM para los nombres de secciones estándares '.text', '.data' y '.bss', pero aún así, pueden ser superpuestos por estos calificadores.

Los calificadores disponibles son:

(*) 'code', o de manera equivalente 'text', define la sección como una sección de código. Esto marca la sección como legible y ejecutable, pero no como escribible, y también indica al enlazador que el tipo de la sección es code.

(*) 'data' y 'bss' definen la sección como una sección de datos, análogamente a 'code'. Las secciones de datos están marcadas como legibles y escribibles, pero no como ejecutables. 'data' declara una sección de datos inicializados, mientras que 'bss' declara una sección de datos no inicializados.

(*) 'info' define la sección como una sección informativa, que no es incluida por el enlazador en el fichero ejecutable, pero podría (por ejemplo) pasar información _al_ enlazador. Por ejemplo, declarando una sección del tipo 'info' llamada '.drectve' haces que el enlazador interprete el contenido de la sección como opciones de la línea de comandos.

(*) 'align=', usado seguido de un número como en 'obj', da los requerimientos de alineamiento de la sección. El máximo que puedes especificar es 64: el fichero objeto Win32 no contiene otra manera de pedir un mayor alineamiento de sección que ésta. Si el alineamiento no está especificado explícitamente, por defecto es una alineamiento de 16-bytes para la sección de código, y un alineamiento de 4-bytes para las secciones de datos (y BSS). Las secciones informativas tienen por defecto un alineamiento de 1 byte (sin alineamiento), aunque el valor no importa.

Los valores por defecto asumidos por NASM si no especificas los calificadores de arriba son:

```
section .text code align=16
section .data data slign=4
section .bss bss align=4
```

Cualquier otro nombre de sección es tratado por defecto como '.text'.

6.4 'coff': Common Object File Format

El tipo de salida 'coff' produce ficheros objeto COFF apropiados para enlazar con el enlazador DJGPP.

'coff' proporciona una extensión '.o' por defecto para el nombre del fichero de salida.

El formato 'coff' soporta las mismas extensiones que la directiva 'SECTION' al igual que lo hace 'win32', excepto que el calificador 'align' y el tipo de sección 'info' no están implementadas.

6.5 'elf': Ficheros objeto ELF de Linux

El formato de salida 'elf' genera fichero objeto ELF32 (Executable and Linkable Format), como los usados por Linux. 'elf' proporciona una extensión '.o' por defecto para el nombre del fichero de salida.

6.5.1 Extensiones 'elf' para la directiva 'SECTION'

Al igual que el formato 'obj', 'elf' te permite especificar información adicional en la línea de la directiva 'SECTION', para controlar el tipo y las propiedades de las secciones que declares. Los tipos de la sección y las propiedades son generadas automáticamente por NASM para las nombres de secciones estándar '.text', '.data' y '.bss', pero aun así pueden ser superpuestas por estos calificadores.

Los calificadores disponibles son:

(*) 'alloc' define la sección para que sea una de las que son cargadas en memoria cuando se ejecuta el programa. 'noalloc' la define como las que no son cargadas al inicio, como puedan ser las secciones informativas o de comentarios.

(*) 'ecex' define la sección como una de las que tiene que tener permiso de ejecución cuando se ejecuta el programa. 'noecex' la define como las que no lo necesitan.

(*) 'write' define la sección como una de las que deberían ser escribibles cuando se ejecuta el programa. 'nowrite' como las que no.

(*) 'progbits' define la sección como una de las que tiene contenido explícito almacenado en el fichero objeto: una sección corriente de datos o de código, por ejemplo, 'nobits' define la sección como una que no tiene contenido dado explícitamente, como la sección BSS.

(*) 'align=' usado seguido de un número al igual que en 'obj', da los requerimientos del alineamiento de la sección.

Los valores por defecto asumidos por NASM si no especificas los calificadores de arriba son:

```
section .text progbits alloc   exec nowrite align=16
section .data progbits alloc noexec   write align=4
section .bss    nobits alloc noexec   write align=4
section other progbits alloc noexec nowrite align=1
```

(Cualquier otro nombre de sección que no sea '.text', '.data' o '.bss' es tratado por defecto como 'other' en el código de arriba.)

6.5.2 Código independiente de la posición: Símbolos 'elf' especiales y 'WRT'

La especificación ELF contiene suficientes características para permitir escribir código independiente de la posición (PIC), lo que hace muy flexibles las bibliotecas compartidas de ELF. Sin embargo, esto también quiere decir que NASM tiene que ser capaz de generar una variedad de tipos de recolocación extraños en los ficheros objeto ELF, si quiere ser un ensamblador que pueda escribir PIC.

Ya que ELF no soporta referencias al segmento base, el operador 'WRT' no se usa para se propósito normal; por lo tanto el formato de salida 'elf' de NASM hace un uso de 'WRT' para un propósito diferente, a saber, los tipos específicos de la relocalización PIC.

'elf' define cinco símbolos especiales que puedes usar en el lado derecho del operador 'WRT' para obtener tipos de relocalización PCI. Estos son '..gotpc', '..gotoff', '..got', '..plt' y '..sym'. Sus funciones están resumidas aquí:

(*) Refiriéndose al símbolo que marca la base de la tabla global de offset, usando 'wrt ..gotpc' terminará dándonos la distancia desde el principio de la sección actual hasta la tabla global de offset. ('_GLOBAL_OFFSET_TABLE_' es el nombre estándar de símbolo referido comúnmente como GOT.) Por eso, deberías entonces necesitar añadir '$$'' al resultado para recibir la dirección real del GOT.

(*) Refiriéndose a una posición en una de tus propias secciones, usando 'wrt ..gotoff' te dará la distancia desde el principio del GOT hasta la posición especificada, por eso, esto añadido a la dirección del GOT nos daría la dirección real del la posición que quieres.

(*) Refiriéndose a un símbolo externo o global, usando 'wrt ..got' hace que el enlazador construya una entrada _en_ el GOT que contenga la dirección del símbolo, y la referencia da la distancia desde el principio del GOT hasta la entrada; por eso puedes añadir la dirección del GOT, cargar desde la dirección resultante, y terminar en la dirección del símbolo.

(*) Refiriéndose a un nombre de procedimiento, usando 'wrt ..plt' hace que el enlazador construya para el símbolo una entrada en la tabla de

enlazamiento de procedimientos, y las referencias den la dirección de la entrada a la PLT. Sólo puedes usar esto en contextos que pudiesen generar una relocalización relativa al PC de manera normal (esto es, como el destino de un 'CALL' o un 'JMP'), ya que ELF no contiene en absoluto ningún tipo de relocalización para referirse a la PLT.

(*) Refiriéndose a un nombre de símbolo, usando 'wrt ..sym' hace que NASM escriba una relocalización normal, pero en lugar de hacer la relocalización relativa al principio de la sección y luego añadir el offset del símbolo, escribirá un 'record' de relocalización apuntando directamente al símbolo en cuestión. Esta distinción es necesario debido a una peculiaridad del enlazador dinámico.

Una explicación más completa de cómo usar estos tipos de relocalizaciones para escribir completamente en NASM bibliotecas compartidas está en la sección 8.2.

6.5.3 Extensiones 'elf' para la directiva 'GLOBAL'

Los ficheros objeto ELF pueden contener más información de un símbolo global además de su dirección: pueden contener el tamaño del símbolo así como su tipo. No son sólo convenientes para la depuración, sino que son actualmente necesarios cuando el programa que se está escribiendo es una biblioteca compartida. NASM por lo tanto soporta algunas extensiones para la directiva 'GLOBAL', permitiéndote especificar estas características.

Puedes especificar si una variable global es una función o datos poniendo como sufijo el nombre con dos puntos y la palabra 'function' o 'data'. ('object' es un sinónimo de 'data'.) Por ejemplo:

```
          global hashlookup:function, hashtable:data
```

exporta el símbolo global 'hashlookup' como una función y 'hashtable' como un dato.

También puedes especificar el tamaño del dato asociado con el símbolo, como una expresión numérica (que podría involucrar etiquetas, e incluso referencias más adelante) después del especificador de tipo. Como éste:

```
          global hashtable:data (hashtable.end - hashtable)
hashtable:
          db this,that,theother  ; aquí algunos datos
.end:
```

Esto hace que NASM calcule automáticamente la longitud de la tabla y coloque la información dentro de la tabla de símbolos ELF.

Es necesario declarar el tipo y el tamaño de los símbolos globales cuando escribes código de bibliotecas compartidas. Para ver más información, ver la sección 8.2.4.

6.5.4 Extensiones 'elf' para la directiva 'COMMON'

ELF también te permite especificar los requerimientos de alineamiento en las variables comunes. Esto se hace poniendo un número (que debe ser una potencia de dos) después del nombre y el tamaño de la variable común, separado (como siempre) por dos puntos. Por ejemplo, un array de palabras dobles (doubleword) se beneficiaría de un alineamiento de 4-bytes:

```
common dwordarray 128:4
```

Esto declara el tamaño total del array en 128 bytes, y requiere que esté alineado con un límite de 4-bytes.

6.6 'aout': Ficheros objeto 'a.out' de Linux

El formato 'aout' genera ficheros objeto 'a.out', de la forma usado por los primeros sistemas Linux. (Estos difieren de los otros ficheros objeto 'a.out' en que el número mágico en los primeros cuatro bytes es diferente. También, algunas implementaciones de 'a.out', por ejemplo las de NetBSD, proporcionan código independiente de la posición, el cual la implementación de Linux no soporta.)

'a.out' proporciona una extensión '.o' por defecto para los nombres de sus ficheros de salida.

'a.out' es un formato objeto muy sencillo. No soporta directivas especiales, ni símbolos especiales, no usa 'SEG' ni 'WRT', y no tiene extensiones de ninguna directiva estándar. Sólo soporta los tres nombres de sección estándar '.text', '.data' y '.bss'.

6.7 'aoutb': Ficheros objeto 'a.out' de NetBSD/FreeBSD/OpenBSD

El formato 'aoutb' genera ficheros objeto 'a.out', de la manera a la usada por varios clónicos gratuitos de BSD Unix, NetBSD, FreeBSD y OpenBSD. Para los ficheros objeto simples, este formato objeto es exactamente el mismo que 'aout' excepto para el número mágico en los primero cuatro bytes del fichero. Sin embargo, el formato 'aoutb' soporta código independiente de la posición del mismo modo que el formato 'elf', por eso puedes usarlo para escribir bibliotecas BSD compartidas.

'aoutb' proporciona una extensión '.o' por defecto para los nombres de sus ficheros de salida.

'aoutb' no soporta directivas especiales, ni símbolos especiales, y sólo los tres nombres estándar de sección '.text', '.data' y '.bss'. Sin embargo, también soporta el mismo uso de 'WRT' que hace 'elf', para proporcionar tipos de relocalización de código independiente de la posición. Ver la sección 6.5.2 para una documentación completa de esta característica.

'aoutb' también soporta las mismas extensiones para la directiva 'GLOBAL' que 'elf': ver la sección 6.5.3 para la documentación de esto.

6.8 'as86': Ficheros objeto 'as86' de Linux

El ensamblador de Linux de 16-bit 'as86' tiene su propio formato no estándar de ficheros objeto. Aunque su enlazador compañero 'ld86' produce como salida algo cercano a un binario 'a.out', el formato del fichero objeto usado para comunicarse entre 'as86' y 'ld86' no es en sí mismo un 'a.out'.

NASM soporta este formato, sólo en caso de que sea necesario, como 'as86'. 'as86' proporciona la extensión '.o' por defecto para el nombre del fichero de salida.

'as86' es un formato objeto muy sencillo (desde el punto de vista del usuario de NASM). No soporta directivas especiales, ni símbolos especiales, no usa 'SEG' ni 'WRT', y no tiene extensiones para ninguna

directiva estándar. Sólo soporta los tres nombre de sección estándar '.text', '.data' y '.bss'.

6.9 'rdf': Formato de ficheros objeto dinámicos recolocables

El formato de salida 'rdf' produce ficheros objeto RDOFF. RDOFF (Relocatable Dynamic Object File Format) es el formato de ficheros objeto propio, diseñado al lado de NASM y reflejando en el propio formato de fichero la estructura interna del ensamblador.

Ningún sistema operativo bien conocido usa RDOFF. Estos escriben sus propios sistemas, sin embargo, bien podrían desear usar RDOFF como su formato objeto, en la base que está diseñado principalmente en busca de simplicidad y contiene muy poca burocracia en las cabeceras de fichero.

Tanto el paquete NASM de Unix, y el de DOS que incluye los fuentes, contienen un subdirectorio 'rdoff' que tiene un conjunto de utilidad de RDOFF: un enlazador RDF, un gestor de bibliotecas RDF estáticas, una utilidad de volcado de ficheros RDF, y un programa que cargará y ejecutará un ejecutable RDF bajo Linux.

'rdf' sólo soporta los nombres de sección estándar '.text', '.data' y '.bss'.

6.9.1 Requiriendo una biblioteca: La directiva 'LIBRARY'

RDOFF contiene un mecanismo por el que un fichero objeto puede pedir que se enlace una biblioteca a un módulo, ya sea en tiempo de carga o de ejecución. Esto se hace con la directiva 'LIBRARY', que recibe un argumento que es el nombre del módulo:

```
library mylib.rdl
```

6.10 'dbg': Formato de depuración

El formato de salida 'dbg' no está construido en NASM en la configuración por defecto. Si estás construyendo tus propios ejecutables NASM desde los fuentes, puedes definir 'OF_DBG' en 'outform.h' en la línea de comandos del compilador, y obtener el formato de salida 'dbg'.

El formato de salida no saca un fichero objeto tal cual; en su lugar, saca un fichero de texto que contiene una lista completa de todas las transacciones entre el cuerpo principal de NASM y el módulo del formato de salida. Está principalmente para ayudar a la gente que quiere escribir sus propios controladores de salida, así pueden recibir una idea más clara de las distintas peticiones que hace el programa principal al controlador de salida, y en qué orden suceden.

Para ficheros sencillos, uno puede fácilmente usar el formato 'dbg' tal como éste:

```
nasm -f dbg filename.asm
```

que generará un fichero de diagnóstico llamado 'filename.dbg'. Sin embargo, esto no funcionará bien en ficheros que hayan sido diseñados para una formato objeto diferente, porque cada formato objeto define sus propias macros (normalmente formas de directivas a nivel de usuario), y estas macros no serán definidas en el formato 'dbg'. Por lo tanto, puede ser útil ejecutar NASM dos veces, para hacer el preprocesado con el formato objeto nativo seleccionado:

```
nasm -e -f rdf -o rdfprog.i rdfprog.asm
nasm -a -f dbg rdfprog.i
```

Esto preprocesa 'rdfprog.asm' en 'rdfprog.i', manteniendo el formato objeto 'rdf' seleccionado para asegurarse de que las directivas especiales son convertidas en su forma primitiva de forma correcta. Entonces el fuente preprocesado es alimentado a través del formato 'dbg' para generar la definitiva salida del diagnóstico.

Esta forma de actuar típicamente no funcionará para los programas que se deseen en el formato 'obj', porque las directivas 'SEGMENT' y 'GROUP' del 'obj' tienen el efecto lateral de definir los nombres del segmento y el grupo como símbolos; 'dbg' no hará esto, por eso el programa no se ensamblará. Tendrás que trabajar en esto definiendo los símbolos por ti mismo (usando 'EXTERN', por ejemplo) si realmente necesitas un 'dbg' para seguir la pista de un fichero fuente 'obj' específico.

'dbg' acepta cualquier nombre de sección y cualquier directiva, y toma nota en el fichero de salida.

Capítulo 7: Escribiendo código 16-bit (DOS, Windows 3/3.1)

Este capítulo intenta cubrir algunas de las cuestiones comunes que se encuentran cuando se escribe código de 16-bit para ejecutar bajo MS-DOS o Windows 3.x. Cubre cómo enlazar programas para producir ficheros '.EXE' o '.COM', cómo escribir controladores de dispositivos '.SYS', y cómo juntar código en lenguaje ensamblador con compilador de C de 16-bit y con Borland Pascal.

7.1 Produciendo ficheros '.EXE'

Cualquier programa grande escrito bajo DOS necesita construirse como un fichero '.EXE': sólo los ficheros '.EXE' tiene la estructura interna necesaria requerida para extenderse más allá del segmento de 64K. Los programas Windows, también, han de construirse como ficheros '.EXE', ya que Windows no soporta el formato '.COM'.

En general, generas ficheros '.EXE' usando el formato objeto 'obj' para producir uno o más ficheros '.OBJ', y luego los enlazas usando un enlazador. Sin embargo, NASM también soporta la generación directa de ficheros '.EXE' simples del DOS usando el formato de salida 'bin' (usando 'DB' y 'DW' para construir la cabecera del fichero '.EXE'), y para ello se proporciona un paquete de macros. Gracias a Yann Guidon por contribuir al código con esto.

NASM también podría soportar en futuras versiones '.EXE' de forma nativa.

7.1.1 Usando el formato 'obj' para generar ficheros '.EXE'

Esta sección describe el método usual de generar ficheros '.EXE' enlazando juntos ficheros '.OBJ'.

La mayoría de los paquetes de lenguajes de programación de 16-bit vienen con un enlazador adecuado; si no tienes ninguno de estos, hay un enlazador gratuito llamado VAL, disponible comprimido en formato 'LZH' desde 'x2ftp.oulu.fi'. Un compresor LZH se puede encontrar en 'ftp.simtel.net'. Hay otro enlazador 'gratuito' (aunque este viene sin los fuentes) llamado FREELINK, disponible en 'www.pcorner.com'. Un

tercero, 'djlink', escrito por DJ Delorie, está disponible en 'www.delorie.com'.

Cuando enlaces varios ficheros '.OBJ' en un fichero '.EXE', deberías asegurarte de que exactamente uno de ellos tiene definido un punto de comienzo (usando el símbolo especial '..start' definido en el formato 'obj': ver la sección 6.2.6). Si ningún módulo define el punto de comienzo, el enlazador no sabrá qué valor dar en el campo de punto de entrada en la cabecera del fichero de salida; si más de uno define un punto de comienzo, el enlazador no sabrá _qué_ valor usar.

Un ejemplo de fichero fuente NASM que puede ensamblarse a un fichero '.OBJ' y enlazarlo solo a un '.EXE' viene a continuación. Demuestra los principios básicos para definir una pila, inicializar los registros de segmento, y declarar un punto de comienzo. El fichero se proporciona también en el subdirectorio 'test' del archivo NASM, bajo el nombre 'objexe.asm'.

```
          segment code

..start:  mov ax.data
          mov ds.ax
          mov ax,stack
          mov ss.ax
          mov sp,stacktop
```

Este trozo de código inicial prepara 'DS' para apuntar al segmento de datos, e inicializa 'SS' y 'SP' para apuntar a la parte de arriba de la pila proporcionada. Nótese que las interrupciones están deshabilitadas de forma implícita por una instrucción después de un 'mov' a 'SS', precisamente por esta situación, es por eso que no hay oportunidad de que ocurra una interrupción entre la carga de 'SS' y 'SP' y no tenga una pila sobra la que ejecutarse.

Nótese también que el símbolo especial '..start' está definido al principio de este código, lo que significa que será el punto de entrada en el fichero ejecutable resultante.

```
          mov dx,hello
          mov ah,9
          int 0x21
```

Lo de arriba es el programa principal: carga 'DS:DX' con un puntero al mensaje de saludo ('hello' es de forma implícita relativo al segmento 'data', que fue cargado en 'DS' en el código de inicialización, por eso todo el puntero es válido), y llama a la función imprimir cadena del DOS.

```
          mov ax,0x4c00
          int 0x21
```

Esto termina el programa usando otra llamada de sistema del DOS.

```
          segment data
hello:    db 'hello, world', 13, 10, '$'
```

El segmento de datos contiene la cadena que queremos mostrar.

```
          segment stack stack
          resb 64
stacktop:
```

El código de arriba declara un segmento de pila que contiene 64 bytes de espacio de pila no inicializado, y apunta 'stacktop' a la parte superior de ella. La directiva 'segment stack stack' define un segmento _llamado_ 'stack', y también de _tipo_ 'STACK'. Lo siguiente no es necesario para la ejecución correcta del programa, pero los enlazadores suelen emitir avisos y errores si tu programa no tiene segmento de tipo 'STACK'.

El fichero de arriba, cuando se ensambla en un fichero '.OBJ', enlazará por si solo un fichero '.EXE' válido, que cuando se ejecute imprimirá 'hello, world' y saldrá.

7.1.2 Usando el formato 'bin' para generar ficheros '.EXE'

El formato de fichero '.EXE' es lo suficientemente simple como para construir un fichero '.EXE' escribiendo un programa binario puro y pegándole delante una cabecera de 32-bytes. Esta cabecera es lo suficientemente sencilla que NASM puede generarla por sí mismo usando los comandos 'DB' y 'DW', por eso puedes usar el formato de salida 'bin' para generar directamente los ficheros '.EXE'.

Incluido en el archivo NASM, en el subdirectorio 'misc', está un fichero 'exebin.mac' de macros. Define tres macros: 'EXE_begin', 'EXE_stack' y 'EXE_end'.

Para producir un fichero '.EXE' usando este método, deberías empezar por usar '%include' para cargar el paquete de macros 'exebin.mac' en tu fichero fuente. Deberías luego hace una llamada a la macro 'EXE_begin' (que no recibe argumentos) para generar los datos de la cabecera del fichero. Luego escribir el código como siempre para el formato 'bin' - puedes usar las tres secciones estándar '.text', '.data' y '.bss'. Al final del fichero deberías llamar a la macro 'EXE_end' (nuevamente, sin argumentos), que define algunos símbolos para marcar los tamaños de las secciones, y estos símbolos están referidos en el código de cabecera generado por 'EXE_begin'.

En este modelo, el código que acabas de escribir comienza en '0x100', al igual que un fichero '.COM' - de hecho, si eliminas los 32-bytes de cabecera del fichero '.EXE' resultante, tendrías un programa '.COM' válido. Todas las bases de segmentos son las mismas, por eso estás limitado a un programa de 64K, nuevamente al igual que un fichero '.COM'. Nótese que la macro 'EXE_begin' emite una directiva 'ORG', por eso tú no deberías hacer explícitamente una.

No puedes referirte directamente al valor de tu segmento base, desafortunadamente, ya que esto requeriría una recolocación en la cabecera, y las cosas se podrían hacer mucho más complicadas. Por eso deberías en su lugar coger tu segmento base copiando 'CS'.

En la entrada a tu fichero '.EXE', 'SS:SP' están ya preparadas para apuntar al principio de la pila de 2K. Puedes ajustar el tamaño de 2K por defecto de la pila llamando a la macro 'EXE_stack'. Por ejemplo, para cambiar el tamaño de la pila de tu programa a 64 bytes, llamarías a 'EXE_stack 64'.

Un programa de ejemplo que genera un fichero '.EXE' de esta forma está en el subdirectorio 'test' del archivo NASM, como 'binexe.asm'.

7.2 Produciendo ficheros '.COM'

Mientras que los programas grandes de DOS han de escribirse como ficheros '.EXE', los pequeños, a menudo, mejor se escriben como ficheros '.COM'. Los ficheros '.COM' son binario puro, y por lo tanto se producen mucho más fácilmente usando el formato de salida 'bin'.

7.2.1 Usando el formato 'bin' para generar ficheros '.COM'

Los ficheros '.COM' se espera que se carguen en el offset '100h' en su segmento (aunque el segmento pueda cambiar). Le ejecución entonces comienza en '100h', esto es, a la derecha del comienzo del programa. Por eso para escribir un programa '.COM', deberías crear un fichero fuente parecido a éste

```
          org 100h
          section .text
start:    ; pon aquí tu código
          section .data
          ; pon aquí los datos
          section .bss
          ; pon aquí los datos no inicializados
```

El formato 'bin' pone en el fichero primero la sección '.text', por eso si quieres puedes declarar los data y BSS antes de empezar a escribir código y el código todavía irá a parar delante del fichero al que pertenezca.

La sección BBS (datos no inicializados) no ocupa espacio en el fichero '.COM' por sí misma: en su lugar, los direccionamientos a objetos del BSS se resuelven apuntando al espacio más allá del final del fichero, al principio de lo que será memoria libre cuando se ejecute el programa. Por lo tanto, no deberías confiar en que tu BSS sea inicializado todo a ceros cuando los ejecutes.

Para ensamblar el programa de arriba, deberías usar una línea de comandos como

```
nasm myprog.asm -fbin -o myprog.com
```

El formato 'bin' produciría un fichero llamado 'myprog' si no se especifica explícitamente un nombre para el fichero de salida, por eso deberás eliminarlo dando el nombre de fichero deseado.

7.2.2 Usando el formato 'obj' para generar ficheros '.COM'

Si estás escribiendo un programa '.COM' como más de un módulo, podrías querer ensamblar varios ficheros '.OBJ' y enlazarlos juntos en un programa '.COM'. Puedes hacerlo, siempre que tengas un enlazador capaz de crear ficheros '.COM' directamente (TLINK lo hace), o alternativamente un programa convertidor como 'EXE2BIN' para transformar un fichero de salida '.EXE' del enlazador a un fichero '.COM'.

Si haces esto, debes tener en cuenta varias cosas:

(*) El primer fichero objeto que contiene el código debería empezar su segmento de código con una línea como 'RESB 100h'. Esto es para asegurar que el código empieza en el offset '100h' relativo al principio del segmento de código, así el enlazador o el programa convertidor no tiene que ajustar las referencias de dirección dentro del fichero '.COM' cuando se genere. Otros ensambladores usan para este propósito una directiva 'ORG', pero 'ORG' en NASM es una

directiva específica del formato del salida 'bin', y no quiere decir lo mismo que en los ensambladores compatibles con MASM.

(*) No necesitas definir el segmento de pila.

(*) Todos tus segmentos deberían estar en el mismo grupo, por eso cada vez que tu código o tus datos referencien un offset de un símbolo, todos los offsets son relativos al mismo segmento base. Esto es debido a que cuando es cargado un fichero '.COM', todos los registros de segmento contienen el mismo valor.

7.3 Produciendo ficheros '.SYS'

Los controladores de dispositivos de MS-DOS - ficheros '.SYS' - son binario puro, parecido a los ficheros '.COM', excepto que empiezan en el origen cero en lugar del '100h'. Así pues, si estás escribiendo un controlador de dispositivo usando el formato 'bin', no necesitas la directiva 'ORG', ya que el origen por defecto para 'bin' es cero. De forma parecida, si estás usando 'obj' no necesitas el 'RESB 100h' al principio de tu segmento de código.

Los fichero '.SYS' comienzan con una estructura de cabecera, que contiene punteros a diversas rutinas dentro del controlador que son las que hacen el trabajo. Esta estructura debería definirse al comienzo del segmento de código, aunque realmente no sea código.

Para ver más información del formato de los ficheros '.SYS', y de los datos que deben ir en la cabecera, se da una lista de libros en la lista de preguntas frecuentemente realizadas (FAQ) para el grupo de noticias 'comp.os.msdos.programmer'.

7.4 Interactuando con programas en C de 16-bits

Esta sección cubre los principios básicos para escribir rutinas en ensamblador que llamen, o sean llamadas desde, programas en C. Para esto, típicamente deberías escribir un módulo en ensamblador como un fichero '.OBJ', y enlazarlo con tus módulos de C para producir un programa en lenguajes mezclados.

7.4.1 Nombres de símbolos externos

Los compiladores de C tienen el convenio de que los nombres de todos sus símbolos globales (funciones o datos) que definan están formados precediendo un subrayado al nombre tal cual aparece en el programa C. Por eso, por ejemplo, la función que el programador de C conoce como 'printf' aparece para el programador en ensamblador como '_printf'. Esto quiere decir que en tus programas en ensamblador, puedes definir símbolos sin el subrayado precedente y así no preocuparte de si el nombre entra en conflicto con algún símbolo de C.

Si encuentras inconvenientes los subrayados, puedes definir macros para reemplazar las directivas 'GLOBAL' y 'EXTERN' como sigue:

```
%macro cglobal 1
         global _%1
%define %1 _%1
%endmacro

%macro cextern 1
         extern _%1
```

```
%define %1 _%1
%endmacro
```

(Estas formas de las macros sólo reciben un argumento cada vez; una construcción '%rep' podría resolver esto.)

Si entonces declaras una externa como ésta:

```
          cextern printf
```

entonces la macro se expandirá como

```
          extern _printf
%define printf _printf
```

Después, te puedes referir a 'printf' como si fuese un símbolo, y el preprocesador pondrá el subrayado precedente allí donde sea necesario.

La macro 'cglobal' funciona de forma similar. Debes usar 'cglobal' antes de definir el símbolo en cuestión, aunque tendrías que hacer esto de todas formas si usases 'GLOBAL'.

7.4.2 Modelos de memoria

NASM no contiene ningún mecanismo para soportar directamente los diferentes modelos de memoria de C; deberás recordar por ti mismo para cuál estás escribiendo. Esto quiere decir que deberás recordar las siguientes cosas:

(*) En modelos que usen un simple segmento de código (tiny, small y compact), las funciones son cercanas. Lo que significa que los punteros a funciones, cuando se almacenan en segmentos de datos o son empujados a la pila como argumentos de una función, son de 16-bit y sólo contienen un campo de offset (el registro 'CS' nunca cambia su valor, y siempre da la parte el segmento en la dirección completa de la función), y estas funciones se llaman usando la instrucción cercana 'CALL' normal y vuelve usando 'RETN' (la cual, en NASM, es sinónimo de 'RET' de todas formas). Esto quiere decir que deberás escribir tus propias rutinas para volver con un 'RETN', y que deberías llamar a las rutinas C externas con instrucciones 'CALL' cercanas.

(*) En los modelos que usen más de un segmento de código (medium, large y huge), las funciones son lejanas. Esto significa que los punteros de función son de 32-bits (que consisten en un offset de 16-bit seguido de un segmento de 16-bit), y estas funciones se llaman usando 'CALL FAR' (o 'CALL seg:offset) y vuelven usando 'RETF'. Nuevamente, deberías por lo tanto escribir tus propias rutinas para volver con 'RETF' y usar 'CALL FAR' para llamar a rutinas externas.

(*) En los modelos que usen un solo segmento de datos (tiny, small y medium), los punteros de datos son de 16-bits, y contiene sólo un campo de offset (el registro 'DS' no cambia su valor, y siempre da la parte del segmento de la dirección completa del dato).

(*) En los modelos que usen más de un segmento de datos (compact, large y huge), los punteros de datos son de 32-bit, y consisten en un offset de 16-bit seguido de un segmento de 16-bit. Deberías tener cuidado de no modificar 'DS' en tus rutinas sin restaurarlo después a la vuelta, aunque 'ES' está libre para ti para que lo uses para

acceder al contenido de los punteros de 32-bit a datos que pases.

(*) El modelo de memoria huge permite a los datos simples exceder el tamaño de 64K. En todos los demás modelos, puedes acceder a la totalidad de un dato haciendo simplemente aritmética sobre el campo de offset del puntero que estás dando, si está presente un campo de segmento o no; en el modelo huge, debes ser más cuidadoso en tu aritmética de punteros.

(*) En la mayoría de los modelos de memoria, hay un segmento de datos _por defecto_, aquellos cuya dirección de segmento se guarde en 'DS' durante todo el programa. Este segmento de datos es normalmente el mismo segmento que la pila, guardado en 'SS', por eso estas variables locales de funciones (que están guardadas en la pila) y los datos globales pueden accederse fácilmente sin cambiar 'DS'. En particular los datos grandes se suelen guardar en otros segmentos. Sin embargo, algunos modelos de memoria (aunque normalmente no los estándar) permiten que se asuma que 'SS' y 'DS' guardan el mismo valor para ser borrado. Ten cuidado en este último caso con las variables locales de las funciones.

En los modelos con un solo segmento de código, el segmento se llama '_TEXT', por eso tu segmento de código debe ir con este mismo nombre para que sea enlazado en el mismo lugar que el segmento de código principal. En los modelos con solo segmento de datos, o con un segmento de datos por defecto, se llama '_DATA'.

7.4.3 Definiciones de funciones y llamadas a las funciones

El convenio de llamadas de C en los programas de 16-bits es el que sigue. En la siguiente descripción, las palabras _llamante_ y _llamado_ se usan para denotar la función que hace la llamada y la que la recibe.

(*) El llamante empuja los parámetros de la función a la pila, uno tras otro, en orden inverso (de derecha a izquierda, por eso el primer argumento especificado a la función es empujado el último).

(*) El llamante entonces ejecuta una instrucción 'CALL' para pasarle el control al llamado. Este 'CALL' puede ser cercano o lejano dependiendo del modelo de memoria.

(*) El llamado recibe el control, y normalmente (aunque realmente no es necesario, en funciones que no necesitan acceder a sus parámetros) comienza salvando el valor de 'SP' en 'BP' para ser capaz de usar 'BP' como puntero base para encontrar sus parámetros en la pila. Sin embargo, el llamante estuvo haciendo esto también, por eso, parte del convenio de llamadas dictamina que 'BP' debe conservarse por cualquier función de C. Por lo tanto el llamado, si va a establecer 'BP' como _puntero de marco_, debe empujar a la pila previamente el valor que contiene.

(*) El llamado podría entonces acceder a sus parámetros relativo a 'BP'. La palabra en '[BP]' contiene el valor previo de 'BP' según fue empujado a la pila; la siguiente palabra, en '[BP+2]', contiene la parte del offset de la dirección de retorno, empujada implícitamente por 'CALL'. En una función (cercana) del modelo small, los parámetros comienzan después de esto, en '[BP+4]'; en una función (lejana) del modelo large, la parte del segmento de la dirección de retorno está en '[BP+4]', y los parámetros comienzan en '[BP+6]'. El parámetro más a la izquierda de la función, ya que fue empujado el

último, está accesible en este offset desde 'BP'; los demás siguen en los sucesivos offsets cada vez mayores. Así, en una función como 'printf' que recibe un número variable de parámetros, el empujar parámetros en el orden inverso quiere decir que la función conoce dónde encontrar el primer parámetro, que le dice el número y tipo de los restantes.

(*) El llamado podría también desear el decrementar 'SP' más adelante, así como para asignar espacio en la pila para las variables locales, que serán accesibles en offsets negativos desde 'BP'.

(*) El llamado, si quiere devolver un valor al llamante, debería dejar el valor en 'AL', 'AX' o 'DX:AX' dependiendo del tamaño del valor. Los resultados en coma flotante son a veces (dependiendo del compilador) devueltos en 'ST0'.

(*) Una vez el llamado ha terminado de procesar, restaura 'SP' desde 'BP' si ha asignado espacio de la pila local, entonces saca el valor previo de 'BP', y regresa vía 'RETN' o 'RETF' dependiendo del modelo de memoria.

(*) Cuando el llamante retoma el control desde el llamado, los parámetros de la función están todavía en la pila, por eso típicamente añade una constante inmediata a 'SP' para eliminarlos (en lugar de ejecutar un número de las lentas instrucciones 'POP'). Así, si se llama accidentalmente a una función con un número erróneo de parámetros debido a un error del prototipo, la pila todavía será devuelta a un estado correcto ya que el llamante, que _conoce_ cuántos parámetros empujó, hace le eliminación.

Es instructivo comparar este convenio de llamadas con el de los programas en Pascal (descrito en la sección 7.5.1). Pascal tiene un convenio simple, ya que las funcionen no tienen un número variable de parámetros. Por lo tanto el llamado conoce cuántos parámetros deberían pasársele, y es capaz de desasignarlos de la pila por sí mismo pasando un argumento inmediato a la instrucción 'RET' o 'RETF', por eso el llamante no tiene que hacerlo. También, los parámetros son empujados en orden de izquierda a derecha, no de derecha a izquierda, lo que significa que un compilador puede dar mayores garantías en la secuencia de puntos sin que sufra el rendimiento.

Por eso, deberías definir una función en el estilo de C de la siguiente forma. El siguiente ejemplo es para el modelo small:

```
          global _myfunc
_myfunc:  push bp
          mov bp,sp
          sub sp,0x40           ; 64 bytes de espacio de la pila local
          mov bx,[bp+4]         ; primer parámetro de la función
          ; algo más de código
          mov sp,bp             ; deshace el "sub sp,0x40" de arriba
          pop bp
          ret
```

Para una función en un modelo large, deberías sustituir 'RET' por 'RETF', y buscar el primer parámetro en '[BP+6]' en lugar de '[BP+4]'. Por supuesto, si uno de los parámetros es un puntero, entonces los offsets de los parámetros subsiguientes cambiarán dependiendo también del modelo de memoria: los punteros lejanos ocupan cuatro bytes en la pila cuando son pasados como parámetros, por el contrario los punteros cercanos

ocupan dos bytes.

En el otro extremo del proceso, para llamar a una función C desde tu código ensamblador, deberías hacer algo como esto:

```
          extern _printf
          ; y luego, más abajo...
          push word [myint]      ; una de mis variables enteras
          push word mystring     ; puntero en mi segmento de datos
          call _printf
          add sp,byte 4          ; 'byte' salva espacio
          ; luego estos datos
          segment _DATA
myint     dw 1234
mystring  db 'This number -> %d <- should be 1234',10,0
```

Este trozo de código es el equivalente en el modelo small del ensamblador al código C

```
    int myint = 1234;
    printf("This number -> %d <- should be 1234\n", myint);
```

En el modelo large, el código de la llamada a la función podría parecerse más a éste. En este ejemplo, se asume que 'DS' ya contiene el segmento base del segmento '_DATA'. Si no, deberías inicializarlo antes.

```
          push word [myint]
          push word seg mystring ; Ahora empuja el segmento, y...
          push word mystring     ; ... el offset de "mystring"
          call far _printf
          add sp,byte 6
```

El valor entero todavía ocupa una palabra en la pila, ya que el modelo large no afecta al tamaño del tipo de dato 'int'. El primer argumento (empujado el último) a 'printf', sin embargo, es un puntero de dato, y por lo tanto tiene que contener una parte segmento y otra offset. El segmento debería almacenarse en memoria en segundo lugar, y por lo tanto debe empujarse primero. (Por supuesto, 'PUSH DS' debería haber tenido una instrucción más corta que 'PUSH WORD SEG mystring', si 'DS' ha sido establecido como en el ejemplo de arriba.) Entonces la llamada actual se hace una llamada lejana, ya que las funciones esperan llamadas lejanas en el modelo de memoria large; y 'SP' tiene que incrementarse después en 6 en lugar de 4 para inventarse la palabra extra de los parámetros.

7.4.4 Accediendo a los datos

Para leer el contenido de las variables, o para declarar variables que se puedan acceder desde C, sólo necesitas declarar los nombres como 'GLOBAL' o 'EXTERN'. (Nuevamente, los nombre requieren los subrayados precedentes, como se dijo en la sección 7.4.1.). Así, una variable C declarada como 'int i' puede accederse desde ensamblador como

```
          extern _i
          mov ax,[_i]
```

Y para declarar tu propia variable entera a la que puedan acceder los programas en C como 'extern int j', harías esto (asegurándote de que estás ensamblando en el segmento '_DATA', si es necesario):

```
          global _j
```

```
_j        dw 0
```

Para acceder a un array C, necesitas conocer el tamaño de los componentes del array. Por ejemplo, las variables 'int' son de dos bytes, por eso si un programa C declara un array como 'int a[10]', puedes acceder al 'a[3]' codificando 'mov ax,[_a+6]'. (El offset del byte 6 se obtiene multiplicando el índice del elemento deseado del array, 3, por el tamaño de elemento del array, 2.) Los tamaños de los tipos básicos de C en compiladores de 16-bit son:
1 para el 'char', 2 para el 'short' e 'int', 4 para el 'long' y 'float', y 8 para el 'double'.

Para acceder a un estructura de datos en C, necesitas conocer el offset desde la base de la estructura al campo que esté interesado. También puedes hacer esto convirtiendo la definición de la estructura de C en una definición de estructura de NASM (usando 'STRUC'), o calculando el offset y simplemente usándolo.

Para hacer cualquiera de estas dos cosas, deberías leer el manual de tu compilador de C para encontrar cómo organiza las estructuras. NASM no da un alineamiento especial a los miembros de la estructura en su macro 'STRUC', por eso tienes que especificar el alineamiento por ti mismo si el compilador lo genera. Típicamente, podrías encontrar que una estructura como ésta

```
struct {
    char c;
    int i;
} foo;
```

podría ocupar cuatro bytes en lugar de tres, ya que el campo 'int' sería alineado a un límite de dos bytes. Sin embargo, este tipo de característica tiende a ser una opción configurable en el compilador de C, ya sea usando opciones de la línea de comandos o líneas '#pragma', por eso tienes que encontrar cómo lo hace tu compilador.

7.4.5 'c16.mac': Macros de ayuda para la interface de C de 16-bit

Incluido en el archivo NASM, en el directorio 'misc', hay un fichero de macros, 'c16.mac'. Define tres macros: 'proc', 'arg' y 'endproc'. Son deseadas para usar en las definiciones de procedimientos al estilo de C, y automatizan un montón de trabajo involucrado en llevar la cuenta de los convenios de llamadas.

Un ejemplo de una función en ensamblador usando el conjunto de macros dadas aquí:

```
          proc _nearproc
%$i       arg
%$j       arg
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          add ax,[bx]
          endproc
```

Esto define '_nearproc' como un procedimiento que recibe dos argumentos, el primero ('i') es un entero y el segundo ('j') es un puntero a un entero. Devuelve 'i + *j'.

Nótese que la macro 'arg' tiene un 'EQU' como primera línea en su

expansión, y ya que la etiqueta antes de la llamada a la macro queda unida a la primera línea de la macro expandida, el 'EQU' funciona, definiendo '%$i' como un offset desde 'BP'. Se usa una variable de contexto local, local al contexto empujado por la macro 'proc' y sacada por la macro 'endproc', por eso se puede usar el mismo nombre de argumento en posteriores procedimientos. Por supuesto, _no tienes_ por qué hacerlo.

Por defecto el conjunto de macro genera código para las funciones cercanas (modelos tiny, small y compact). Puedes generar funciones lejanas (modelos medium, large y huge) mediante la codificación de '%define FARCODE'. Esto cambia el tipo de la instrucción de retorno generada por 'endproc', y también cambia el punto de entrada de los offsets de los argumentos. El conjunto de macros no contiene dependencias intrínsecas de si los punteros de los datos son lejanos o no.

'arg' puede recibir un parámetro opcional, dando el tamaño del argumento. Si no se da el tamaño, se asume que es 2, ya que la mayoría de los argumentos de las funciones serán de tipo 'int'.

El equivalente al modelo large de la función de arriba sería algo así:

```
%define FARCODE
          proc _farproc
%$i       arg
%$j       arg 4
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          mov es,[bp + %$j + 2]
          add ax,[bx]
          endproc
```

Esto hace uso del argumento de la macro 'arg' para definir un tamaño de parámetro de 4, porque 'j' ahora un puntero lejano. Cuando nosotros cargamos desde 'j', debemos cargar un segmento y un offset.

7.5 Interpretando programas en Borland Pascal

La interpretación de los programas de Borland Pascal es parecida en el concepto a la interpretación de los programas de C de 16-bit. Las diferencias son:

(*) El subrayado precedente requerido para la interpretación de los programas C no se necesita para los de Pascal.

(*) El modelo de memoria siempre es large: las funciones son lejanas, los punteros a los datos son lejanos, y no puede haber datos de mayor tamaño que 64K. (Realmente, algunas funciones son cercanas, pero sólo aquellas funciones que son locales a una unidad Pascal y nunca son llamadas desde funciones externas a ella. Todas las funciones en ensamblador que llame Pascal, y todas las funciones Pascal que son capaces de llamar las rutinas ensamblador, son lejanas.) Sin embargo, todos los datos estáticos declarados en un programa Pascal va en el segmento de datos por defecto, que es aquel cuya dirección de segmento estará en 'DS' cuando el control se pase a tu código ensamblador. La única cosa que no está en el segmento de datos por defecto son las variables locales (están en el segmento de pila) y las variables asignadas dinámicamente. Todos los _punteros_ de datos, sin embargo, son lejanos.

(*) Los convenios de las llamadas a funciones son diferentes - descritos más abajo.

(*) Algunos tipos de datos, como las cadenas, se almacenan de manera diferente.

(*) Hay restricciones en los nombres de segmentos que estás autorizado a usar - Borland Pascal ignorará el código o los datos declarado en un segmento cuyo nombre no sea válido. Las restricciones están descritas más abajo.

7.5.1 El convenio de las llamadas en Pascal

El convenio de las llamadas Pascal de 16-bit es como sigue. En la siguiente descripción, las palabras _llamante_ y _llamado_ se usan para denotar la función que hace la llamada y la función que la recibe.

(*) El llamante empuja a la pila los parámetros de la función, una tras otra, en otro orden (de izquierda a derecha, por eso el primer argumento especificado a la función se empuja primero).

(*) El llamante luego ejecuta una instrucción 'CALL' lejana para pasar el control al llamado.

(*) El llamado recibe el control, y típicamente (aunque esto no es realmente necesario, en las funciones que no necesitan acceder a sus parámetros) comienza salvando el valor de 'SP' en 'BP' para poder usar 'BP' como puntero base para encontrar sus parámetros en la pila. Sin embargo, el llamante probablemente hizo esto, por eso parte del convenio de llamadas determina que 'BP' debe ser conservado por cualquier función. Por lo tanto el llamado, si va a establecer 'BP' como un puntero de marco, debe empujar previamente el valor a la pila.

(*) El llamado podría entonces acceder a sus parámetros relativo al 'BP'. La palabra en '[BP]' contiene el valor previo de 'BP' según fue empujado. La siguiente palabra, en '[BP+2]', contiene la parte de offset de la dirección de retorno, y la siguiente '[BP+4]' contiene la parte del segmento. Los parámetros comienzan en '[BP+6]'. El parámetro más a la derecha de la función, ya que fue empujado el último, está accesible en este offset desde 'BP'; los siguientes le siguen en offsets cada vez mayores.

(*) El llamado podría también desear decrementar 'SP' más adelante, para asignar espacio en la pila para las variables locales, que entonces estarán accesibles en offsets negativos desde 'BP'.

(*) El llamado, si desea devolver un valor al llamante, debería dejar el valor en 'AL', 'AX' o 'DX:AX' dependiendo del tamaño del valor. Los resultados en coma flotante se devuelven en 'ST0'. Resultados del tipo 'Real' (el tipo de dato personal en coma flotante de Borland, no manejado directamente por la FPU) se devuelven en 'DX:BX:AX'. Para devolver un resultado del tipo 'String', el llamante empuja un puntero a una cadena temporal antes de empujar los parámetros, y el llamado coloca el valor de la cadena de retorno en esta posición. El puntero no es un parámetro, y no debería ser eliminado de la pila por la instrucción 'RETF'.

(*) Una vez el llamado ha terminado de procesar, restaura 'SP' desde 'BP'

si ha asignado espacio de pila local, luego saca el valor previo de 'BP', y retorna vía 'RETF'. Usa la forma de 'RETF' con un parámetro inmediato, dando el número de bytes ocupados en la pila por los parámetros. Esto hace que se eliminen de la pila los parámetros como un efecto lateral de la instrucción de retorno.

(*) Cuando el llamante retoma el control desde el llamado, los parámetros de la función ya han sido eliminados de la pila, por eso no tiene que hacer nada más.

Así, definirías una función en el estilo de Pascal, que recibe dos parámetros del tipo entero, del siguiente modo:

```
          global myfunc
myfunc:   push bp
          mov bp,sp
          sub sp,0x40           ; 64 bytes de espacio de pila local
          mov bx,[bp+8]         ; el primer parámetro de la función
          mov bx,[bp+6]         ; el segundo parámetro de la función
          ; algo más de código
          mov sp,bp             ; deshace el 'sub sp,0x40' de arriba
          pop bp
          retf 4                ; el tamaño total de los parámetros es 4
```

En el otro extremo del proceso, para llamar a un función Pascal desde tu código ensamblador, deberías hacer algo como:

```
          extern SomeFunc
          ; y ahora, más abajo...
          push word seg mystring ; ahora empuja el segmento, y...
          push word mystring     ; ... el offset de "mystring"
          push word [myint]      ; una de mis variables
          call far SomeFunc
```

Esto es equivalente al código Pascal

```
procedure SomeFunc(String: PChar; Int: Integer);
    SomeFunc(@mystring, myint);
```

7.5.2 Restricciones de Borland Pascal a los nombres de segmento

Ya que la unidad interna del formato de fichero del Pascal de Borland es completamente diferente del 'OBJ', sólo hace un trabajo incompleto de la lectura y comprensión real de la diversa información que está contenida en un fichero 'OBJ' real cuando lo enlaza en él. Por lo tanto, un fichero objeto que se desee enlazar a un programa Pascal debe obedecer una serie de restricciones:

(*) Los procedimientos y las funciones deben estar en un segmento cuyo nombre sea 'CODE', 'CSEG', o algo terminado en '_TEXT'.

(*) Los datos inicializados deben estar en un segmento cuyo nombre sea 'CONST' o algo terminado en '_DATA'.

(*) Los datos no inicializados deben estar en un segmento cuyo nombre sea 'DATA', 'DSEG', o algo terminado en '_BSS'.

(*) Cualquier otro segmento en el fichero objeto será completamente ignorado. También son ignoradas las directivas 'GROUP' y los atributos de segmentos.

7.5.3 Usando 'c16.mac' con programas en Pascal

El paquete de macros 'c16.mac', descrito en la sección 7.4.5 también puede usarse si codificas '%define PASCAL' para simplificar la escritura de funciones que sean llamadas desde programas en Pascal. Esta definición asegura que las funciones son lejanas (implica 'FARCODE'), y también hace que la instrucción de retornos del procedimiento se genere con un operando.

Definir 'PASCAL' no cambia el código que calcula los offsets del argumento; debes declarar en orden inverso los argumentos de tu función. Por ejemplo:

```
%define PASCAL
          proc _pascalproc
%$j       arg 4
%$i       arg
          mov ax,[bp + %$i]
          mov bx,[bp + %$j]
          mov es,[bp + %$j + 2]
          add ax,[bx]
          endproc
```

Esto define la misma rutina, conceptualmente, que en el ejemplo de la sección 7.4.5: define una función que recibe dos argumentos, un entero y un puntero a entero, que devuelve la suma del entero y el contenido del puntero. La única diferencia entre este código y la versión de C con modelo large es que 'PASCAL' está definida en lugar de 'FARCODE', y que los argumentos se declaran en orden contrario.

Capítulo 8: Escribiendo código 32-bit (Unix, Win32, DJGPP)

Este capítulo intenta cubrir algunas de las cuestiones comunes que surgen cuando se escribe código de 32-bit, para correr bajo Win32 o Unix, o para ser enlazado con código C generado por un compilador de C del estilo de Unix como pueda ser DJGPP. Cubre cómo escribir código ensamblador para interactuar con rutinas C, y cómo escribir código independiente de la posición para bibliotecas compartidas.

Casi todo el código de 32-bit, y en particular todo el código que se ejecuta bajo Win32, DJGPP y cualquiera de las variantes de Unix para PC, se ejecuta en modelos de memoria _plana_. Esto quiere decir que los registros de segmentos y paginación ya han sido establecidos para darte el mismo espacio de 4GB de direcciones de 32-bit, independientemente de en relación a qué segmento trabajes, y deberías ignorar completamente todos los registros de segmento. Cuando escribes código de aplicaciones en un modelo plano, nunca necesitas usar solapamiento de segmentos ni modificar ningún registro de segmento, y las direcciones de las secciones de código que pasas a una 'CALL' y 'JMP' están en el mismo espacio de direcciones que las direcciones de datos con las que accedes a tus variables y las direcciones de la sección de pila con las que accedes a tus variables locales y parámetros de procedimiento. Cada dirección es de 32-bit y sólo contiene la parte del offset.

8.1 Interactuando con programas en C de 32-bits

Mucho de lo discutido en la sección 7.4, sobre la interacción con programas en C de 16-bits, todavía se aplica cuando se trabaja con

32-bits. La ausencia de modelos de memoria o los problemas de segmentación simplifican mucho las cosas.

8.1.1 Nombres de los símbolos externos

La mayoría de los compiladores de C de 32-bit comparten el convenio usado en los compiladores de 16-bits, que los nombres de todos los símbolos globales (funciones o datos) que definan estén formados por un subrayado precediendo al nombre que aparece en el programa C. Sin embargo, no todos lo hacen: la especificación ELF determina que los símbolos C _no_ tienen un subrayado precediendo los nombres en lenguaje ensamblador.

El antiguo compilador 'a.out' de C de Linux, todos los compiladores Win32, DJGPP, y NetBSD y FreeBSD, todos usan el subrayado precedente; para estos compiladores, las macros 'cextern' y 'cglobal', como las dadas en la sección 7.4.1, todavía funcionarán. Para ELF, sin embargo, el subrayado prefijado no debería usarse.

8.1.2 Definiciones de funciones y llamadas a funciones

El convenio de llamadas en C en programas de 32-bits es como sigue. En la siguiente descripción, las palabras _llamante_ y _llamado_ se usan para denotar a la función que hace la llamada y la función que la recibe.

(*) El llamante empuja a la pila los parámetros de la función, uno tras otro, en orden contrario (de derecha a izquierda, por eso el primer argumentos especificado en la función es empujado el último).

(*) El llamante luego ejecuta una instrucción 'CALL' cercana para pasarle el control al llamado.

(*) El llamado recibe el control, y normalmente (aunque esto no es realmente necesario en las funciones que no necesitan acceder a sus parámetros) comienza salvando el valor de 'ESP' en 'EBP' para poder ser capaz de usar 'EBP' como puntero base para encontrar sus parámetros en la pila. Sin embargo, el llamante habrá hecho probablemente también esto, por eso, parte del convenio de llamadas determina que 'EBP' debe conservarse en cualquier función C. Por lo tanto, el llamado, si va a establecer 'EBP' como un puntero de marco, debe empujar previamente el valor a la pila.

(*) El llamado podría luego acceder a sus parámetros en relación a 'EBP'. La palabra doble en '[EBP]' contiene el anterior valor de 'EBP' según fue empujado; la siguiente palabra doble, en '[EBP+4]' contiene la dirección de retorno, empujada implícitamente por 'CALL'. Los parámetros comienzan después de ésta, en '[EBP+8]'. El parámetro más a la izquierda de la función, ya que fue empujado el último, está accesible en este offset desde 'EBP'; los demás le siguen, en offsets sucesivos. Así, en una función como 'printf' que recibe un número variable de argumentos, empujar los parámetros en orden contrario quiere decir que la función conoce dónde encontrar el primer parámetro, que le dice el número y tipo de los restantes.

(*) El llamado también podría desear decrementar 'ESP' en un futuro, para asignar espacio en la pila para las variables locales, que serán accesibles en offsets negativos desde 'EBP'.

(*) El llamado, si desea devolver un valor al llamante, debería dejarlo en 'AL', 'AX' o 'EAX' dependiendo del tamaño del valor. Los resultado en coma flotante normalmente se devuelven en 'ST0'.

(*) Una vez que el llamado ha terminado de procesar, restaura 'ESP' desde 'EBP' si ha asignado espacio de pila local, luego saca el anterior valor de 'EBP', y retorna vía 'RET' (equivalente a 'RETN').

(*) Cuando el llamante recupera el control desde el llamado, los parámetros de la función todavía están en la pila, por eso, normalmente añade un valor inmediato a 'ESP' para eliminarlos (en lugar de ejecutar un número de lentas instrucciones 'POP'). Así, si una función es llamada accidentalmente con un número erróneo de parámetros debido un fallo en el prototipo, la pila será devuelta a un estado correcto ya que el llamante, que _conoce_ cuántos parámetros empujó, hace la eliminación.

Hay un convenio alternativo para las llamadas, usado por los programas de Win32 para las llamadas de la API, y también para las funciones llamadas _por_ el API de Windows como los procedimientos de la ventanas: ellas siguen lo que Microsoft llama el convenio '__stdcall'. Este es un poco más parecido al convenio de Pascal, en el que el llamado limpia la pila pasando un parámetro en la instrucción 'RET'. Sin embargo, el parámetro sigue siendo empujado en el orden de derecha a izquierda.

Así, definirías una función en el estilo de C del siguiente modo:

```
          global _myfunc
_myfunc:  push ebp
          mov ebp,esp
          sub esp,0x40           ; 64 bytes de espacio de pila local
          mov ebx,[ebp+8]        ; primer parámetro de la función
          ; algo más de código
          leave                  ; mov esp,ebp / pop ebp
          ret
```

En el otro extremo del proceso, para llamar a una función C desde tu código ensamblador, harías algo parecido a esto:

```
          extern _printf
          ; y luego, más abajo...
          push dword [myint]     ; una de mis variables enteras
          push dword mystring    ; un puntero en mi segmento de datos
          call _printf
          add esp,byte 8         ; 'byte' libera espacio
          ; luego estos datos...
          segment _DATA
myint     dd 1234
mystring  db 'This number -> %d <- should be 1234',10,0
```

Este trozo de código es el equivalente en ensamblador al código de C

```
int myint = 1234;
printf("This number -> %d <- should be 1234\n", myint);
```

8.1.3 Accediendo a los datos

Para obtener los contenidos de las variables de C, o para declarar variables que se puedan acceder desde C, sólo necesitas declarar los nombres como 'GLOBAL' o 'EXTERN'. (Nuevamente, los nombres requieren el subrayado delantero, como se dijo en la sección 8.1.1). Así, una variable en C declarada como 'int i' se puede acceder desde ensamblador como

```
          extern _i
          mov eax,[_i]
```

Y para declarar tu propia variable entera que pueda ser accedida desde programas en C como 'extern int j', haz esto (asegurándote de que estás ensamblando el segmento '_DATA', si es necesario):

```
          global _j
_j        dd 0
```

Para acceder a un array C, necesitas conocer el tamaño de los componentes del array. Por ejemplo, las variables 'int' son de cuatro bytes, por eso si un programa C declara un array como 'int [10]', puedes acceder a 'a[3]' codificando 'mov ax,[_a+12]'. (El offset del byte 12 se obtiene multiplicando el índice deseado del array, 3, por el tamaño de elemento del array, 4). Los tamaños de los tipos base de C en un compilador de 32-bits son: 1 para el 'char', 2 para el 'short', 4 para el 'int', 'long' y 'float', y 8 para el 'double'. Los punteros, siendo direcciones de 32-bits, son también de 4 bytes.

Para acceder a una estructura de C, necesitas conocer el offset desde la base de la estructura hasta el campo en que estás interesado. Puedes hacerlo convirtiendo la definición de la estructura de C en una definición de estructura de NASM (usando 'STRUC'), o calculando el offset y simplemente usándolo.

Para hacer cualquiera de estas dos, deberías leer el manual de tu compilador de C para encontrar cómo organiza las estructuras de datos. NASM no da en su propia macro 'STRUC' un alineamiento especial a los miembros de la estructura, por eso tienes que especificar el alineamiento si el compilador de C lo genera. Típicamente, podrías encontrar que una estructura como

```
struct {
    chart c;
    int;
} foo;
```

podría ser de ocho bytes en lugar de cinco, ya que el campo 'int' podría alinearse a un límite de cuatro bytes. Sin embargo, este tipo de característica es a veces una opción configurable en el compilador de C, ya sea usando las opciones de la línea de comandos o líneas '#pragma', por eso tienes que encontrar cómo lo hace tu compilador.

8.1.4 'c32.mac': Macros de ayuda para la interface de 32-bit

Incluido en el archivo NASM, en el directorio 'misc' está un fichero 'c32.mac' de macros. Este define tres macros: 'proc', 'arg' y 'endproc'. Viene bien usarlos en las definiciones de procedimientos en el estilo de C, y automatizan mucho trabajo del que implica tener en cuenta el convenio de llamadas.

Un ejemplo de una función en ensamblador usando el conjunto de macros es éste:

```
          proc _proc32
%$i       arg
%$j       arg
          mov eax,[ebp + %$i]
          mov ebx,[ebp + %$j]
```

```
        add eax,[ebx]
        endproc
```

Este define '_proc32' como un procedimiento que recibe dos argumentos, el primero ('i') un entero y el segundo ('j') un puntero a un entero. Devuelve 'i + *j'.

Nótese que la macro 'arg' tiene un 'EQU' como primera línea en su expansión, y ya que la etiqueta de la llamada a la macro se conserva en la primera línea de la macro expandida, el 'EQU' funciona, definiendo '%$i' como un offset desde 'BP'. Se usa una variable de contexto local, local al contexto empujado por la macro 'proc' y sacado por la macro 'endproc', por eso se puede usar el mismo nombre de argumento en posteriores procedimientos. Por supuesto, _no tienes_ por qué hacerlo.

'arg' puede recibir un argumento opcional, dando el tamaño del argumento. Si no se da el tamaño, se asume 4, ya que la mayor parte de los parámetros de las funciones serán del tipo 'int' o punteros.

8.2 Escribiendo bibliotecas compartidas para NetBSD/FreeBSD/OpenBSD y Linux/ELF

ELF sustituye bajo Linux el antiguo formato de fichero objeto 'a.out' porque contiene soporte para código independiente de la posición (PIC), que hace que escribir bibliotecas compartidas sea mucho más fácil. NASM soporta las características del código ELF independiente de la posición, por eso puedes escribir bibliotecas ELF compartidas de Linux en NASM.

NetBSD, y es un primo cercano de FreeBSD y OpenBSD, toma un acercamiento diferente introduciendo el soporte de PIC en el formato 'a.out'. NASM soporta esto como el formato de salida 'aoutb', por eso también puedes escribir en NASM bibliotecas BSD compartidas.

El sistema operativo carga una biblioteca compartida PIC mediante el mapeado de memoria del fichero biblioteca en un punto arbitrario del espacio de direcciones en el proceso que se está ejecutando. El contenido de la sección de código de la biblioteca no debe depender por lo tanto de dónde sea cargado en memoria.

Así, no puedes leer tus variables escribiendo código como éste:

```
        mov eax,[myvar]         ; ERRONEO
```

En su lugar, el enlazador proporciona un área de memoria llamada _tabla global de offset_, o GOT; el GOT está situado a una distancia constante desde el código de tu biblioteca, por eso puedes encontrar dónde está cargada tu biblioteca (que normalmente se hace usando una combinación de 'CALL' y 'POP'), puedes obtener la dirección del GOT, y luego puedes cargar las direcciones de tus variables fuera de las entradas generadas en el GOT por el enlazador.

La sección _de datos_ de una biblioteca compartida PIC no tiene estas restricciones: ya que la sección de datos es escribible, tiene que ser copiada en memoria de cualquier forma en lugar de simplemente paginarla desde el fichero biblioteca, por eso con tal que sea copiada puede ser también recolocada. Así puedes poner los tipos normales de recolocación en la sección de datos sin mucha preocupación (pero mira la sección 8.2.4 para más aclaración).

8.2.1 Obteniendo la dirección del GOT

Cada módulo en tu biblioteca compartida debería definir la GOT como un símbolo externo:

```
          extern _GLOBAL_OFFSET_TABLE_   ; en ELF
          extern __GLOBAL_OFFSET_TABLE_  ; en el a.out de BSD
```

Al comienzo de cualquier función en tu biblioteca compartida que planee acceder a tus datos o a las secciones BSS, debes primero calcular la dirección del GOT. Esto se hace normalmente escribiendo la función de esta forma:

```
func:     push ebp
          mov ebp,esp
          push ebx
          call .get_GOT
.get_GOT: pop ebx
          add ebx,_GLOBAL_OFFSET_TABLE_+$$-.get_GOT wrt ..gotpc
          ; el cuerpo de la función va aquí
          mov ebx,[ebp-4]
          mov esp,ebp
          pop ebp
          ret
```

(Para BSD, nuevamente, el símbolo '_GLOBAL_OFFSET_TABLE_' requiere un segundo subrayado delantero.)

Las dos primeras líneas de esta función son simplemente el prólogo estándar de C para preparar el marco de pila, y las últimas tres líneas son el epílogo estándar de las funciones de C. La tercera línea, y desde la cuarta hasta la última, salvan y restauran el registro 'EBX', porque las bibliotecas compartidas PIC usan este registro para almacenar la dirección del GOT.

El punto interesante es la instrucción 'CALL' y las siguientes dos líneas. La combinación 'CALL' y 'POP' obtiene la dirección de la etiqueta '.get_GOT', sin tener que conocer previamente dónde ha sido cargado el programa (ya que la instrucción 'CALL' está codificada en relación a la actual posición). La instrucción 'ADD' hace uso de uno de los tipos especiales de recolocación: recolocación GOTPC. Con el calificador específico 'WRT ..gotpc', el símbolo referenciado (aquí '_GLOBAL_OFFSET_TABLE_', el símbolo especial asignado al GOT) está dado como un offset desde el principio de la sección. (Realmente, ELF lo codifica como el offset desde el campo del operando de la instrucción 'ADD', pero NASM lo simplifica deliberadamente, por eso hace las cosas igual tanto para ELF como para BSD.) Así la instrucción luego _suma_ el principio de la sección, para conseguir la dirección real de la GOT, y resta el valor de '.get_GOT' que sabe que está en 'EBX'. Por lo tanto, una vez que la instrucción ha terminado, 'EBX' contiene la dirección de la GOT.

Si no has seguido esto, no te preocupes: nunca es necesario obtener la dirección del GOT mediante otras formas, por eso puedes poner estas tres instrucciones en una macro y olvidarte:

```
%macro get_GOT 0
          call %%getgot
%%getgot: pop ebx
          add ebx,_GLOBAL_OFFSET_TABLE_+$$-%%getgot wrt ..gotpc
%endmacro
```

8.2.2 Encontrando tus datos locales

Habiendo cogido la GOT, entonces puedes usarla para obtener las direcciones de tus datos. La mayoría de las variables residirán en las secciones que tú habías declarado; pueden accederse usando el tipo especial de 'WRT', '..gotoff'. La forma en que funciona es como ésta:

```
lea eax,[ebx+myvar wrt ..gotoff]
```

La expresión 'myvar wrt ..gotoff' es calculada, cuando se enlaza la biblioteca compartida, para ser el offset de la variable local 'myvar' desde el comienzo del GOT. Por lo tanto, sumándolo a 'EBX' como arriba colocará la dirección real de 'myvar' en 'EAX'.

Si declaras variables como 'GLOBAL' sin especificar el tamaño para ellas, son compartidas entre los módulos de código de la biblioteca compartida, pero no exportadas desde la biblioteca al programa que la cargue. Estarán en tus secciones normales de datos y BSS, por eso puedes accederlas de la misma forma que a una variable local, usando el mecanismo '..gotoff' de arriba.

Nótese que debido a una peculiaridad del manejo que hace el formato 'a.out' de BSD de este tipo de recolocación, debe haber al menos un símbolo no local en la misma sección que la dirección a la que intentas acceder.

8.2.3 Encontrando datos comunes y externos

Si tu biblioteca necesita coger una variable externa (externa a la _biblioteca_, no simplemente una de los módulos dentro de ella), debes usar el tipo '..got' para cogerla. El tipo '..got', en lugar de darte el offset desde la base GOT hasta la variable, te da el offset desde la base GOT hasta la _entrada_ GOT que contiene la dirección de la variable. El enlazador establecerá esta entrada cuando construya la biblioteca, y el enlazador dinámico colocará la dirección correcta en tiempo de carga. Por eso para obtener la dirección de una variable externa 'extvar' en 'EAX', codificarías

```
mov eax,[ebx+extvar wrt ..got]
```

Esto carga la dirección de 'extvar' fuera de la entrada al GOT. El enlazador, cuando construya la biblioteca compartida, recoge juntas todas las recolocaciones del tipo '..got', y construye el GOT de tal forma que asegure que tiene presente todas las entradas necesarias.

Las variables comunes también deben acceder usando este método.

8.2.4 Exportando símbolos al usuario de la biblioteca

Si quieres exportar símbolos al usuario de la biblioteca, tienes que declarar si son funciones o datos, y si son datos, tienes que dar el tamaño del dato. Esto es porque en enlazador dinámico tiene que construir las entradas a la tabla de enlace del procedimiento para cualquier función exportada, y también mueve los datos exportados fuera de la sección data de la biblioteca en la que están declarados.

Por eso, para exportar una función a un usuario de la biblioteca, debes usar

```
          global func:function   ; la declara como una función
func:     push ebp
          ; etc.
```

Y para exportar un dato como por ejemplo un array, tendrías que codificar

```
          global array:data array.end-array ; también da el tamaño
array:    resd 128
.end:
```

Cuidado: si exportas una variable al usuario de la biblioteca, declarándola como 'GLOBAL' y dando el tamaño, la variable terminará viviendo en la sección de datos del programa principal, en lugar de hacerlo en la sección de datos de tu biblioteca, en donde la has declarado. Por eso tendrás que acceder a tus propias variables globales con el mecanismo '..got' en lugar de con '..gotoff', si eran externas (las cuales, efectivamente, lo son).

Igualmente, si necesitas almacenar la dirección de una global exportada en una de tus secciones de datos, no puedes hacerlo mediante este tipo estándar de código:

```
dataptr:  dd global_data_item    ; ERRONEO
```

NASM interpretará este código como una recolocación normal, en la cual 'global_data_item' es simplemente un offset desde el principio de la sección (o lo que sea); por eso esta referencia terminará apuntando a tu sección de datos en lugar de a la global exportada que reside en cualquier otro lugar.

En lugar del código de arriba, entonces, debes escribir

```
dataptr:  dd global_data_item wrt ..sym
```

que hace uso del tipo especial de 'WRT' '..sym' para indicarle a NASM que busque la tabla de símbolos para un símbolo en particular en esa dirección, en lugar de simplemente recolocar desde la sección base.

Cualquier método funcionará para las funciones: refiriéndote a una de tus funciones mediante

```
funcptr:  dd my_function
```

dará al usuario la dirección del código que escribiste, mientras que

```
funcptr:  dd my_function wrt ..sym
```

dará la dirección de la tabla de enlace del procedimiento para la función, que está donde el programa llamante _creerá_ que vive la función. Cualquier dirección es válida para llamar a la función.

8.2.5 Llamando a procedimientos desde fuera de la biblioteca

Llamar a procedimientos desde fuera de tus bibliotecas compartidas tiene que hacerse mediante una _tabla de enlace de procedimiento_, o PLT. La PLT está en un offset conocido desde donde se carga la biblioteca, por eso el código de la biblioteca puede hacer llamadas a la PLT de forma independiente de la posición. Dentro de la PLT hay código para saltar a offsets contenidos en el GOT, por eso las llamadas a las funciones de otras bibliotecas compartidas o a rutinas en el programa principal pueden

pasarse transparentemente desde sus destinos reales.

Para llamar a una rutina externa, debes usar otro tipo especial de recolocación, 'WRT ..plt'. Esto es mucho más fácil que el basado en el GOT: simplemente sustituyes las llamadas como 'CALL printf' con la versión relativa a la PLT 'CALL printf WRT ..plt'.

8.2.6 Generando el fichero biblioteca

Teniendo escrito algunos módulos de código y ensamblados a ficheros '.o', tienes que generar tu biblioteca compartida con un comando como

```
ld -shared -o library.so module1.o module2.o       # para ELF
ld -Bshareable -o library.so module1.o module2.o   # para BSD
```

Para ELF, si tu biblioteca compartida va a residir en directorios de sistema como 'usr/lib' o '/lib', es aconsejable normalmente usar la opción '-soname' en el enlazador, para almacenar el nombre del fichero final de la biblioteca, con el número de versión, en la biblioteca:

```
ld -shared -soname library.so.1 -o library.so.1.2 *.o
```

Entonces copiarías 'library.so.1.2' en el directorio de la biblioteca, y crearías 'library.so.1' como un enlace simbólico a ella.

Capítulo 9: Mezclando código de 16 y 32 bits

Este capítulo intenta cubrir algunas de las cuestiones, enormemente relacionadas con formas inusuales de direccionamiento e instrucciones de salto, encontradas cuando se escribe código de sistema operativo como las rutinas de inicialización del modo protegido, que requieren código que opere en diversos tamaños de segmentos, como código en un segmento de 16-bit que trata de modificar datos en uno de 32-bit, o saltos entre segmentos de diferentes tamaños.

9.1 Saltos de distinto tamaño

La forma más común de las instrucciones de diverso tamaño es la usada cuando se escribe un S.O. de 32-bit: habiendo hecho tu instalación en el modo de 16-bit, al igual que la carga del kernel, tienes que reiniciarlo cambiando al modo protegido y saltando a la dirección de comienzo del kernel de 32-bit. En un S.O. completamente de 32-bit, ésta tiende a ser la _única_ instrucción de tamaño mezclado que necesitas, ya que todo lo anterior puede hacerse con código puramente de 16-bit, y todo lo posterior puede hacerse en puro 32-bit.

Este salto debe especificar una dirección lejana de 48-bit, ya que el segmento de destino es uno de 32-bit. Sin embargo, debe ensamblarse en un segmento de 16-bit, por eso simplemente codificando, por ejemplo,

```
jmp 0x1234:0x56789ABC  ; ¡erróneo!
```

no funcionará, ya que la parte del offset de la dirección será truncada a '0x9ABC' y el salto será uno lejano normal de 16-bit.

El código de instalación del kernel de Linux rodea la incapacidad del 'as86' para generar la instrucción requerida codificándola manualmente, usando instrucciones 'DB'. NASM puede hacerlo mejor que esto, generando actualmente por sí mismo la instrucción correcta. Aquí va cómo hacerlo

correctamente:

```
jmp dword 0x1234:0x56789ABC  ; correcto
```

El prefijo 'DWORD' (estrictamente hablando, debería ir _después_ de los dos puntos, ya que está declarando el campo del _offset_ como una palabra doble; pero NASM aceptará cualquiera de las dos formas, ya que ambas son ambiguas) fuerza a la parte del offset a ser tratado como lejano, asumiendo que estás deliberadamente escribiendo un salto desde un segmento de 16-bits a unos de 32-bit.

Puedes hacer la operación contraria, saltar desde un segmento de 32-bit a uno de 16-bit, mediante el prefijo 'WORD':

```
jmp word 0x8765:0x4321 ; 32 a 16 bit
```

Si el prefijo 'WORD' se especifica en un modo de 16-bit, o el prefijo 'DWORD' en uno de 32-bit, serán ignorados, ya que cada uno está forzando explícitamente a NASM a un modo en el que ya está de cualquier forma.

9.2 Direccionando entre segmentos de diferente tamaño

Si tu S.O. tiene mezclado 16 y 32 bit, o si estás escribiendo un extensor de DOS, estás obligado a tener que tratar con algunos segmentos de 16-bit y otros de 32-bit. En este punto, probablemente terminarás escribiendo código en un segmento de 16-bit que tiene que acceder a datos en un segmento de 32-bit, o viceversa.

Si el dato que estás intentando acceder en un segmento de 32-bit está entre los primeros 64K del segmento, podrías ser capaz de hacer uso impunemente de una operación de direccionamiento de 16-bit para este propósito; pero tarde o temprano, querrás hacer direccionamiento de 32-bit desde un modo de 16-bit.

El modo más fácil de hacerlo es asegurarte de que usar un registro para la dirección, ya que cualquier dirección efectiva que contenga a un registro de 32-bit está forzada a ser una dirección de 32-bit. Por eso puedes hacer

```
mov eax,offset_into_32_bit_segment_specified_by_fs
mov dword [fs:eax],0x11223344
```

Esto está bien, pero ligeramente molesto (ya que desperdicia una instrucción y un registro) si ya sabes el offset preciso al que estás apuntando. La arquitectura x86 no permite direcciones efectivas de 32-bit más que para offset de 4 bytes, por eso, ¿por qué podría NASM ser capaz de generar la mejor instrucción para esto?.

Puede. Como en la sección 9.1, sólo necesitas preceder la dirección con la palabra clave 'DWORD', y será forzado a ser una dirección de 32-bit:

```
mov dword [fs:dword my_offset],0x11223344
```

También en la sección 9.1, NASM no es exigente sobre si el prefijo 'DWORD' viene antes o después que el solapamiento del segmento, por eso una forma más bonita de codificar la instrucción de arriba es

```
mov dword [dword fs:my_offset],0x11223344
```

No confunda el prefijo 'DWORD' de _fuera_ de los corchetes, que controla

el tamaño del dato almacenado en la dirección, con el de _dentro_ de los corchetes que controla la longitud de la dirección en sí misma. Los dos pueden ser diferentes fácilmente:

```
mov word [dword 0x12345678],0x9ABC
```

Esto mueve 16 bits de datos a un offset especificado por un offset de 32-bit.

También puedes especificar los prefijos 'WORD' y 'DWORD' con el prefijo 'FAR' para indireccionar los saltos o llamadas lejanas. Por ejemplo:

```
call dword far [fs:word 0x4321]
```

Esta instrucción contiene una dirección especificada por un offset de 16-bit; carga un puntero lejano de 48-bit desde ése (segmento de 16-bit y offset de 32-bit), y llama a esta dirección.

9.3 Otras instrucciones de tamaño mezclado

El otro modo en que podías querer acceder a datos podría ser usar las instrucciones de cadenas ('LODSx', 'STOSx' y demás) o la instrucción 'XLATB'. Estas instrucciones, ya que no reciben parámetros, podría parecer que no tienen una forma fácil de hacerlas realizar direccionamientos de 32-bit cuando se ensamblan en segmentos de 16-bit.

Este es el propósito de los prefijos de NASM 'a16' y 'a32'. Si estás codificando 'LODSB' en un segmento de 16-bit pero se supone que está accediendo a una cadena en un segmento de 32-bit, deberías cargar la dirección deseada en 'ESI' y luego codificar

```
a32 lodsb
```

El prefijo fuerza al tamaño del direccionamiento a 32-bits, mediante este 'LODSB' carga desde '[DS:ESI]' en lugar de '[DS:SI]'. Para acceder a una cadena en un segmento de 16-bit cuando se está codificando en uno de 32-bit, se puede usar el prefijo correspondiente 'a16'.

Los prefijos 'a16' y 'a32' pueden aplicarse a cualquier instrucción en la tabla de instrucciones de NASM, pero la mayoría de ellas pueden generar las formas correctas sin ellos. Los prefijo son necesarios sólo para las instrucciones con direccionamiento implícito: 'CMPSx' (sección A.19), 'SCASx' (sección A.149), 'LODSx' (sección A.98), 'STOSx' (sección A.157), 'MOVSx' (sección A.105), 'INSx' (sección A.80), 'OUTSx' (sección A.112), y 'LATB' (sección A.169). También, las diversas instrucciones push y pop ('PUSHA' y 'POPF' así como las más usuales 'PUSH' y 'POP') pueden aceptar los prefijos 'a16' y 'a32' para forzar en particular a 'SP' o 'ESP' a ser puntero de pila, en el caso que el segmento de pila en uso tenga un tamaño de pila diferente del segmento de código.

'PUSH' y 'POP', cuando se aplican a registros en el modo de 32-bit, también tienen el comportamiento un poco extraño de empujar y sacar 4 bytes a la vez, del cual los dos superiores son ignorados y los dos inferiores dan el valor del registro de segmento que está siendo manipulado. Para forzar el comportamiento de 16-bit del registro de segmento de las instrucciones push y pop, puedes usar el prefijo de tamaño de operando 'o16':

```
o16 push ss
o16 push ds
```

Este código salva una palabra doble de espacio de pila encajando dos registros de segmento en el espacio que normalmente sería consumido empujando uno.

(También puedes usar el prefijo 'o32' para forzar el comportamiento de 32-bit cuando estás el modo de 16-bit, pero esto parece menos útil.)

Capítulo 10: Resolución de problemas

este capítulo describe algunos de los problemas más comunes que los usuarios se han ido encontrando con NASM, y sus respuestas. También da instrucciones para informar de bugs en NASM si tú encuentras alguna dificultad que no está listada aquí.

10.1 Problemas comunes

10.1.1 NASM genera código ineficiente

Recibo un montón de informes de 'bugs' de NASM generando código ineficiente, e incluso erróneo, en instrucciones como 'ADD ESP,8'. Esto es una característica deliberada de diseño, ligada a la predecibilidad de la salida: NASM, al ver 'ADD ESP,8', generará la forma de la instrucción que deje espacio para un offset de 32-bits. Necesitas codificar 'ADD ESP,BYTE 8' si quieres la forma de la instrucción eficiente en espacio. Esto no es un bug: en el peor caso un defecto, y esto es sólo cuestión de opiniones.

10.1.2 Mis saltos están fuera de rango

De forma similar, la gente se queja de que cuando comprueban saltos condicionales (que son 'SHORT' por defecto) intentan saltar demasiado lejos, NASM informa de 'short jump out of range' en lugar de hacer los saltos más largos.

Esto, nuevamente, es parcialmente una cuestión de predecibilidad, pero de hecho tiene una razón más práctica. NASM no tiene medios de que se le diga en qué procesador se ejecutará el código que está generando; por eso no puede decidir por sí mismo si debería generar instrucciones del tipo 'Jcc NEAR', porque no sabe que está trabajando en un 386 o superior. Alternativamente, podría sustituir el 'fuera de rango' de las instrucciones cortas 'JNE' con una instrucción 'JE' muy corta que salte sobre un 'JMP NEAR'; ésta es una solución sensata para procesadores inferiores al 386, pero poco eficiente en procesadores que tiene una buena predicción de saltos _y_ podrían haber usado en su lugar 'JMP NEAR'. Por eso, una vez más, es cosa del usuario, no del ensamblador, el decidir qué instrucción debería generar.

10.1.3 'ORG' no funciona

Las personas que están escribiendo programas del sector de arranque en el formato 'bin' a menudo se quejan de que 'ORG' no funciona de la manera que ellos quieren: para colocar la palabra de firma '0xAA55' al final de los 512 bytes de sector de arranque, las personas que usan MASN tienden a codificar

```
ORG 0
; algo de código del sector de arranque
ORG 510
```

```
DW 0xAA55
```

Este no es el uso que se pretende de la directiva 'ORG' en NASM, y no funcionará. El modo correcto de solucionar este problema en NASM es usar la directiva 'TIMES', como ésta:

```
ORG 0
; algo de código del sector de arranque
TIME 510-($-$$) DB 0
DW 0xAA55
```

La directiva 'TIMES' insertará exactamente suficientes bytes cero en la salida para el punto de ensamblaje al 510. Este método también tiene la ventaja de que si accidentalmente llenas demasiado tu sector de arranque, NASM detectará el problema en tiempo de ensamblaje e informará de ello, por eso no terminarás con un sector de arranque que tendrás que desensamblar para encontrar qué es lo que está mal.

10.1.4 'TIMES' no funciona

El otro problema común con el código de arriba es que la gente que escribe la línea 'TIMES' así

```
TIMES 510-$ DB 0
```

razonando que '$' debería ser un número puro, al igual que 510, por lo que la diferencia entre ellos es también un número puro y pude pasarse felizmente a 'TIMES'.

NASM es un ensamblador _modular_: las distintas partes que lo componen están diseñadas para ser fácilmente separadas para reutilizarlas, por eso no intercambian información innecesariamente. En consecuencia, el formato de salida 'bin', incluso aunque se haya dicho mediante la directiva 'ORG' que el la sección '.text' debería empezar en 0, no se pasa esta información de nuevo al evaluador de expresiones. Por eso desde el punto de vista del evaluador, '$' no es un número puro: es un offset desde la sección base. Por lo tanto, la diferencia entre '$' y 510 tampoco es un número puro, sino que implica a la sección base. Los valores que implican a la sección base no se pueden pasar como argumentos a 'TIMES'.

La solución, como en la sección previa, está en codificar la línea de 'TIMES' de la forma

```
TIMES 510-($-$$) DB 0
```

en la cual '$' y '$$' son offsets desde la misma sección base, y por eso su diferencia es un número puro. Esto resolverá el problema y generará código correcto.

10.2 Bugs

Nosotros nunca hemos sacado una versión de NASM con bugs _conocidos_. Aunque no solemos pararnos allí donde no sabemos nada. Cualquiera que encuentres debería informarse a 'anakin@pobox.com'.

Por favor, lee primero la sección 2.2, y no informes del bug si está listado allí como una característica intencionada. (Si crees que la característica está mal pensada, siéntete libre de enviarnos las razones de por qué piensas que debería ser cambiado, pero no nos mandes sólo un mensaje diciendo que 'Esto es un bug' si la documentación dice que está

así a propósito). Luego lee la sección 10.1, y no te molestes de informarnos del bug si está listado allí.

Si informas de un bug, _por favor_ danos la siguiente información:

(*) ¿Qué sistema operativo estás usando para ejecutar NASM?. DOS, Linux, NetBSD, Win16, Win32, VMS (estaría impresionado), o cualquiera que sea.

(*) Si estás ejecutando NASM bajo DOS o Win32, dinos si has compilado tu propio ejecutable desde el archivo fuente de DOS, o si estás usando los ficheros binarios de la distribución estándar. Si estás usando un ejecutable construido de manera local, intenta reproducir el problema usando los ficheros binarios estándar, ya que nos facilitará las cosas para poder reproducir tu problema para poder arreglarlo.

(*) ¿Qué versión de NASM estás usando, y exactamente cómo lo invocas?. Danos la línea de comandos precisa, y el contenido de la variable de entorno 'NASM', si la hay.

(*) ¿Qué versiones de cualquier programa suplementario que estés usando, y cómo los invocas?. Si el problema sólo sa hace visible en el tiempo de compilación, dinos el compilador que estás usando, qué versión tienes, y la línea de comandos exacta. Si el problema implica enlazar ficheros objeto generados por un compilador, dinos qué compilador, qué versión, y qué línea de comando u opciones has usado. (Si estás compilando en un IDE, por favor, intenta reproducir el problema con la versión del compilador de la línea de comandos.)

(*) Si es posible, mándanos un fichero de fuente NASM que presente el problema. Si esto causa problemas de Copyright (por ejemplo, sólo puedes reproducir el bug en código de distribución restringida) entonces ten en mente los dos puntos siguiente: primeramente, garantizamos que cualquier código fuente que se nos mande con fines para la depuración de NASM serán usados _sólo_ para el propósito de depurar NASM, y serán borradas todas nuestras copias tan pronto como hayamos encontrado y reparado el bug o los bugs en cuestión; y segundo, preferimos de cualquier forma que no se nos manden trozos grandes de código. Cuanto menor sea el fichero, mejor. Un fichero de ejemplo de tres líneas que no haga nada útil _excepto_ demostrarnos el problema es mucho mejor para trabajar que un programa funcional de diez mil líneas. (Por supuesto, algunos errores sólo surgen en ficheros grandes, por eso esto podría no ser posible.)

(*) Una descripción de cuál _es_ realmente el problema. 'No funciona' ¡_no_ es una descripción útil!. Por favor, describe exactamente qué está pasando que no debería pasar, o qué es lo que no está pasando y debería. Algunos ejemplos podrían ser: 'NASM me genera un mensaje de error diciendo Línea 3 para un error que realmente está en la Línea 5'; 'NASM genera un mensaje de error que creo que no debería generar en absoluto'; 'NASM no genera un mensaje de error que creo que _debería_ generar'; 'el fichero objeto producido desde este fuente hace fallar a mi enlazador'; 'el noveno byte del fichero de salida es 66 y creo que debería ser en su lugar 77'.

(*) Si crees que el fichero de salida desde NASM es defectuoso, envíanoslo. Esto nos permite determinar si nuestra propia copia de NASM genera el mismo fichero, o si el problema está relacionado con cuestiones de portabilidad entre nuestras plataformas de desarrollo y las tuyas. Nosotros podemos manejar ficheros binarios enviados con

MIME, uuencode, e incluso BinHex. Alternativamente, podríamos quizá proporcionarte un sitio FTP para que puedas subir los ficheros sospechosos; pero enviándonoslos por correo es mucho más fácil para nosotros.

(*) Podría ser útil cualquier otra información o fichero de datos. Si, por ejemplo, el problema implica un fallo de NASM al generar un fichero objeto mientras TASM puede generar sin problemas un fichero equivalente, entonces envíanos _ambos_ ficheros objeto, así podremos ver qué está haciendo TASM distinto a nosotros.

Apéndice A: Referencia a las instrucciones de Intel x86

Este apéndice proporciona una lista completa de las instrucciones máquina qué NASM ensamblará, y una corta descripción de la función de cada una.

No es su intención ser una documentación exhaustiva de cada uno de los detalles de las funciones de las instrucciones, así como de las excepciones que pueden activar: para esta información, deberías ir a la Web de Intel, 'http://www.intel.com'.

En su lugar, este apéndice pretende principalmente proporcionar una documentación del modo en que deberían usarse las instrucciones con NASM. Por ejemplo, buscando 'LOOP' te dirá que NASM permite que se especifique 'CX' o 'ECX' como un segundo argumento opcional para la instrucción 'LOOP', para forzar cuál de los dos posibles contadores debería usarse si el que está por defecto no es el deseado.

las instrucciones no están listadas en orden alfabético, ya que los grupos de instrucciones con una función similar están amontonadas juntas en la misma entrada. Muchas no están muy lejos de su posición alfabética.

A.1 Simbología para la especificación de operandos.

Las descripciones de las instrucciones en este apéndice especifican sus operandos usando la siguiente notación:

(*) Registros: 'reg8' denota un registro de uso general de 8-bit, 'reg16' denota un registro de uso general de 16-bit, y 'reg32' uno de 32-bit. 'fpureg' denota uno de los ocho registros de pila de la FPU, 'mmxreg' denota uno de los ocho registros MMX de 64-bit, y 'segreg' denota un registro de segmento. Adicionalmente, algunos registros (como 'AL', 'DX' o 'ECX') podrían especificarse explícitamente.

(*) Operandos inmediatos: 'imm' denota un operando inmediato genérico. 'imm8', 'imm16' e 'imm32' se usan cuando se pretende que el operando sea de un tamaño específico. Para algunas de estas instrucciones, NASM necesita un especificador explícito: por ejemplo, 'ADD ESP,16' podría interpretarse tanto como 'ADD r/m32,imm32' o como 'ADD r/m32,imm8'. NASM elige la forma por defecto, y por eso debes especificar 'ADD ESP,BYTE 16' para esta última.

(*) Referencias a memoria: 'mem' denota una referencia genérica a memoria; 'mem8', 'mem16', 'mem32', 'mem64' y 'mem80' se usan cuando el operando necesita tener un tamaño específico. Nuevamente, en algunos casos se necesita un especificador: 'DEC [address]' es ambiguo y NASM lo rechazará. Debes especificar en su lugar 'DEC BYTES [address]', 'DEC WORD [address]' o 'DEC DWORD [address]'.

(*) Referencias a memoria restringida: una forma de la instrucción 'MOV' permite especificar una dirección de memoria _sin_ permitir el rango normal de la combinación de registros y procesado de direcciones efectivas. Esto se denota como 'memoffs8', 'memoffs16' y 'memoffs32'.

(*) Elecciones entre registros o memoria: muchas instrucciones pueden aceptar como un operando tanto un registro como una referencia a memoria. 'r/m8' es una abreviatura para 'reg8/mem8'; de forma similar 'r/m16' y 'r/m32'. 'r/m64' está relacionado con MMX, y es una abreviatura de 'mmxreg/mem64'.

A.2 Simbología para las descripciones del código de operación

Este apéndice también proporciona los código de operación que NASM generará para cada forma de cada instrucción. Los códigos de operación están listados de la siguiente forma:

(*) Un número hexadecimal, como '3F', indica un byte fijo que contiene ese número.

(*) Un número hexadecimal seguido de un '+r', como 'C8+r', indica que uno de los operandos de la instrucción es un registro, el 'valor del registro' de ese registro debería añadirse al número hexadecimal para producir el byte generado. Por ejemplo, EDX tiene un valor de registro de 2, por eso el código 'C8+r', cuando el operando registro es EDX, genera el byte hexadecimal 'CA'. En la sección A.2.1 se dan los valores de los registros para registros específicos.

(*) Un número hexadecimal seguido de '+cc', como '40+cc', indica que el nombre de la instrucción tiene un código de condición como sufijo, y la representación numérica del código de condición debería añadirse al número hexadecimal para producir el byte generado. Por ejemplo, el código '40+cc', cuando la instrucción contiene la condición 'NE', genera el byte hexadecimal '45'. Los códigos de condición y sus representaciones numéricas se dan en la sección A.2.2.

(*) Una barra seguida de un dígito, como '/2', indica que uno de los operandos de la instrucción es una dirección de memoria o un registro (denotado 'mem' o 'r/m', con un tamaño opcional). Esto es para codificarlo con una dirección efectiva, con un byte ModR/M, un byte SIB opcional, y un desplazamiento opcional, y el campo (registro) sobrante del byte ModR/M debería ser el dígito dado (que será entre 0 y 7, por eso cabe en tre bits). La codificación de la dirección efectiva se da en A.2.3.

(*) El código '/r' combina los dos de arriba: indica que uno de los operandos es una dirección de memoria o 'r/m', y el otro es un registro, y que una dirección efectiva se debería generar con el campo (registro) sobrante en el byte ModR/M siendo igual al 'valor del registro' del registro operando. La codificación de la dirección efectiva se da en la sección A.2.3; los valores de los registros se dan en la sección A.2.1.

(*) Los códigos 'rb', 'rw' y 'rd' indican que uno de los operandos de la instrucción es un valor inmediato, y que la _diferencia_ entre este valor y la dirección del final de la instrucción se codifica como u byte, palabra o doble palabra respectivamente. Donde aparezca la forma 'rw/rd', indica que tanto 'rw' como 'rd' deberían usarse de manera acorde a si el ensamblaje se está haciendo en 'BITS 16' o 'BITS 32'.

(*) Los códigos 'ow' y 'od' indican que uno de los operandos de la instrucción es una referencia al contenido de una dirección de memoria especificado como un valor inmediato: esta codificación se usa en algunas formas de la instrucción 'MOV' en lugar del mecanismo estándar de direcciones efectivas. El desplazamiento se codifica como una palabra o doble palabra. Nuevamente, 'ow/od' denota que debería elegirse 'ow' o 'od' de acuerdo al establecimiento de 'BITS'.

(*) Los códigos 'o16' y 'o32' indican que la forma dada de la instrucción debería ensamblarse con un tamaño de operando de 16 o de 32 bits. En otras palabras, 'o16' indica un prefijo '66' en el estado 'BITS 32', pero no genera ningún código en el estado 'BITS 16'; y 'o32' indica un prefijo '66' en el estado 'BITS 16' pero no genera nada en 'BITS 32'.

(*) Los códigos 'a16' y 'a32', de forma similar a 'o16' y 'o32', indican el tamaño de la dirección para la forma dada de la instrucción. Cuando no coincida con 'BITS', requiere un prefijo '67'.

A.2.1 Valores de los registros

Donde una instrucción requiera un valor de registro, está ya implícito en la codificación del resto de la instrucción que tipo de registro se desea: un registro de propósito general de 8-bit, un registro de segmento, un registro de depuración, un registro MMX, o lo que sea. Por lo tanto, no hay ningún problema con los registros de diferentes tipos que comparten un valor de codificación.

Las codificaciones de la varias clases de registros son:

(*) Registros generales de 8-bit: 'AL' es 0, 'CL' es 1, 'DL' es 2, 'BL' es 3, 'AH' es 4, 'CH' es 5, 'DH' es 6, y 'BH' es 7.

(*) Registros generales de 16-bit: 'AX' es 0, 'CX' es 1, 'DX' es 2, 'BX' es 3, 'SP' es 4, 'BP' es 5, 'SI' es 6, y 'DI' es 7.

(*) Registros generales de 32-bit: 'EAX' es 0, 'ECX' es 1, 'EDX' es 2, 'EBX' es 3, 'ESP' es 4, 'EBP' es 5, 'ESI' es 6, y 'EDI' es 7.

(*) Registros de segmento: 'ES' es 0, 'CS' es 1, 'SS' es 2, 'DS' es 3, 'FS' es 4, y 'GS' es 5.

(*) {Registros de punto flotante}: 'ST0' es 0, 'ST1' es 1, 'ST2' es 2, 'ST3' es 3, 'ST4' es 4, 'ST5' es 5, 'ST6' es 6, y 'ST7' es 7.

(*) Registros MMX de 64-bit: 'MM0' es 0, 'MM1' es 1, 'MM2' es 2, 'MM3' es 3, 'MM4' es 4, 'MM5' es 5, 'MM6' es 6, y 'MM7' es 7.

(*) Registros de control: 'CR0' es 0, 'CR2' es 2, 'CR3' es 3, y 'CR4' es 4.

(*) Registros de depuración: 'DR0' es 0, 'DR1' es 1, 'DR2' es 2, 'DR3' es 3, 'DR6' es 6, ' DR7' es 7.

(*) Registros de comprobación: 'TR3' es 3, 'TR4' es 4, 'TR5' es 5, 'TR6' es 6, y 'TR7' es 7.

(Nótese que donde quiera que un registro contenga un número, ese número es también el valor del registro para ese registro.)

A.2.2 Códigos de condición

Los códigos de condición disponibles se dan aquí, con sus representaciones numéricas como parte del su código de operación. Muchas de estos códigos de condición tienen sinónimos, por eso se listan varios a la vez.

En las siguientes descripciones, la palabra 'o', cuando se aplica a dos posibles condiciones, se usa para decir 'una o ambas'. Si se quiere decir 'una pero no ambas', se usa la frase 'exactamente una de'.

(*) 'O' es 0 (se lanza si el flag de acarreo está activo); 'NO' es 1.

(*) 'B', 'C' y 'NAE' son 2 (se lanzan si el flag de acarreo está activo); 'AE', 'NB' y 'NC' son 3.

(*) 'E' y 'Z' son 4 (se lanzan si el flag de cero está activo); 'NE' y 'NZ' son 5.

(*) 'BE' y 'NA' son 6 (se lanzan si el flag de acarreo o el de cero están activos); 'A' y 'NBE' son 7.

(*) 'S' es 8 (se lanza si el flag de signo está activo); 'NS' es 9.

(*) 'P' y 'PE' son 10 (se lanzan si el flag de paridad está activo); 'NP' y 'PO' son 11.

(*) 'L' y 'NGE' son 12 (se lanzan si exactamente el flag de signo o el de desbordamiento está activo); 'GE' y 'NL' son 13;

(*) 'LE' y 'NG' son 14 (se lanzan si el flag de cero o exactamente el flag de signo o el de desbordamiento está activo); 'G' y 'NLE' son 15.

Nótese que en todos los casos, el sentido de la condición podría invertirse cambiando el bit bajo de la representación numérica.

A.2.3 Codificación de la dirección efectiva: ModR/M y SIB

Una dirección efectiva se codifica hasta en tres partes: un byte ModR/M, un byte SIB opcional, y un campo de desplazamiento opcional de un byte, palabra o doble palabra.

El ModR/M consiste en tres campos: el campo 'mod', que va desde 0 a 3, en los dos bits más altos del byte, el campo 'r/m', que va desde 0 a 7, en los tres bits más bajos, y el campo (registro) sobrante en el medio (bits 3 al 5). El campo sobrante no es relevante para la dirección efectiva que está siendo codificada, y contiene una extensión al código de operación o el valor de registro de otro operando.

El sistema ModR/M puede usarse para codificar una referencia directa a un registro en lugar de un acceso a memoria. Esto siempre se hace poniendo el campo 'mod' a 3 y el campo 'r/m' al valor de registro del registro en cuestión (debe ser un registro de propósito general, y el tamaño del registro debe estar ya implícito en la codificación del resto de la instrucción). En este caso, el byte SIB y el campo desplazamiento están ausentes.

En el modo de direccionamiento de 16-bit (ya sea 'BITS 16' sin el prefijo

'67', o el 'BITS 32' con un prefijo '67'), el byte SIB nunca se usa. Las reglas generales para 'mod' y 'r/m' (hay una excepción. dada abajo) son:

(*) El campo 'mod' da la longitud del campo desplazamiento: 0 significa que no hay desplazamiento, 1 significa un byte, y 2 significa dos bytes.

(*) El campo 'r/m' codifica la combinación de registros para añadir al desplazamiento para dar la dirección accedida: 0 significa 'BX+SI', 1 significa 'BX+DI', 2 significa 'BP+SI', 3 'BP+DI', 4 'SI' sólo, 5 'DI' sólo, 6 'BP' sólo, y 7 'BX' sólo.

Sin embargo, hay un caso especial:

(*) Si 'mod' es 0 y 'r/m' es 6, la dirección efectiva codificada no es '[BP]' como sugieren las reglas de arriba, sino que es '[disp16]': el campo desplazamiento está presente y es de dos bytes de largo, y no se añaden registros al desplazamientos.

Por lo tanto la dirección efectiva '[BP]' no puede codificarse tan eficientemente como '[BX]'; por eso, si codificas '[BP]' en un programa, NASM añadirá un desplazamiento cero de 8-bit, y establecerá 'mod' a 1, 'r/m' a 6, y el campo desplazamiento de un byte a 0.

En el modo de direccionamiento de 32-bit (ya sea 'BITS 16' con un prefijo '67', o 'BITS 32' sin el prefijo '67') las reglas generales (nuevamente, hay excepciones) para 'mod' y 'r/m' son:

(*) El campo 'mod' da la longitud del campo desplazamiento: 0 significa que no hay desplazamiento, 1 significa un byte, y 2 significa cuatro bytes.

(*) Si sólo se va a añadir un registro al desplazamiento, y no es 'ESP', el campo 'r/m' da su valor de registro, y el byte SIB está ausente. Si el campo 'r/m' es 4 (que codificaría 'ESP'), está presente el byte SIB y da la combinación y escalado de los registros que va a ser añadidos al desplazamiento.

Si está presente el byte SIB, describe la combinación de registros (un registro base opcional, y un registro índice opcional escaldo mediante una multiplicación por 1, 2, 4 u 8) a ser añadidos al desplazamiento. El byte SIB está dividido en el campo 'scale', en los dos bit superiores, el campo 'index' en los tres siguientes, y el campo 'base' en los últimos tres. Las reglas generales son:

(*) El campo 'base' codifica el valor de registro del registro base.

(*) El campo 'index' codifica el valor de registro del registro índice, a menos que sea 4, en cuyo caso no se usa registro índice (por eso, no se puede usar 'ESP' como un registro índice).

(*) El campo 'scale' codifica el multiplicador por el cual se escala el registro índice antes de añadirlo a la base y al desplazamiento: 0 codifica un multiplicador de 1, 1 codifica 2, 2 codifica 4 y 3 codifica 8.

Las excepciones a las reglas de codificación de 32-bits son:

(*) Si 'mod' es 0 y 'r/m' es 5, la dirección efectiva codificada no es '[EBP]' como sugieren las reglas de arriba, sino que en su lugar es

'[disp32]': el campo desplazamiento está presente y es de cuatro bytes, y no se añaden registros al desplazamiento.

(*) Si 'mod' es 0, 'r/m' es 4 (lo que quiere decir que el byte SIB está presente) y 'base' es 4, la dirección efectiva codificada no es '[EBP+index]' como sugieren las reglas de arriba, sino que es '[disp32+index]': el campo desplazamiento está presente y es de cuatro bytes de largo, y no hay registro base (pero el registro índice todavía se procesa de la forma normal).

A.3 Simbología de los flags de instrucción

Con cada instrucción en este apéndice se da un conjunto de flags, denotando el tipo de instrucción. Los tipos son los siguientes:

(*) '8086', '186', '286', '386', '486', 'PENT' y 'P6' denotan el procesador más bajo que soporta la instrucción. La mayoría de las instrucciones correrán en todos los procesadores por encima del tipo dado; aquellos que no están documentadas. El Pentium II no contiene ninguna instrucción adicional más allá del P6 (Pentium Pro); desde el punto de vista del conjunto de instrucciones, podría pensarse como un P6 con capacidad MMX.

(*) 'CYRIX' indica que la instrucción es específica de los procesadores Cyrix, por ejemplo las instrucciones MMX extra en el conjunto extendido de instrucciones MMX.

(*) 'FPU' indica que la instrucción es una de coma flotante, y sólo ejecutará en ordenadores con coprocesador (incluido automáticamente en los 486DX, Pentium y superiores).

(*) 'MMX' indica que esa instrucción es una de MMX, y correrá en procesadores Pentium con capacidad MMX o en el Pentium II.

(*) 'PRIV' indica que esa instrucción es una instrucción de manejo del modo protegido. Muchas de estas podrían usarse sólo en modo protegido, o en un nivel de privilegio cero.

(*) 'UNDOC' indica que esa instrucción está indocumentada, y no es parte la Arquitectura Intel oficial; podría o no ser soportada por una determinada máquina.

A.4 'AAA', 'AAS', 'AAM', 'AAD': Ajustes ASCII

```
AAA                         ; 37                [8086]

AAS                         ; 3F                [8086]

AAD                         ; D5 0A             [8086]
AAD imm                     ; D5 ib             [8086]

AAM                         ; D4 0A             [8086]
AAM imm                     ; D4 ib             [8086]
```

Estas instrucciones se usan en conjunción con las instrucciones de suma, resta, multiplicación y división para realizar aritmética binaria codificada decimal en forma _desempaquetada_ (un dígito BCD por byte - por lo tanto los nombres de la instrucción, fácil de traducir a y desde ASCII). Hay también instrucciones BCD empaquetadas 'DAA' y 'DAS': ver la sección A.23.

'AAA' debería usarse después de una instrucción 'ADD' de un byte cuyo destino haya sido el registro 'AL': mediante el examen del valor en el nibble bajo de 'AL' y también el flag auxiliar de acarreo 'AF', determina si la suma tiene desbordamiento, y lo ajusta (y activa el flag de acarreo) si es necesario. Puedes sumar juntas cadenas BCD largas haciendo 'ADD'/'AAA' en los bits bajos, luego haciendo 'ADC'/'AAA' en cada uno de los bits subsiguientes.

'AAS' funciona de forma similar a 'AAA', pero es para usar después la instrucción 'SUB' en lugar de 'ADD'.

'AAM' es para usar después de que hayas multiplicado juntos dos dígitos decimales y dejes el resultado en 'AL': esto divide 'AL por diez y almacena el cociente en 'AH', dejando el resto en 'AL'. El divisor 10 puede cambiarse especificando un operando en la instrucción: un uso de esto particularmente práctico es 'AAM 16', haciendo que los dos nibbles en 'AL' se separen en 'AH' y 'AL'.

'AAD' realiza la operación inversa a 'AAM': multiplica 'AH' por diez, lo suma a 'AL', y pone 'AH' a cero. Nuevamente, el multiplicador 10 se puede cambiar.

A.5 'ADC': Suma con acarreo

```
ADC r/m8,reg8               ; 10 /r              [8086]
ADC r/m16,reg16             ; o16 11 /r          [8086]
ADC r/m32,reg32             ; o32 11 /r          [386]

ADC reg8,r/m8               ; 12 /r              [8086]
ADC reg16,r/m16             ; o16 13 /r          [8086]
ADC reg32,r/m32             ; o32 13 /r          [386]

ADC r/m8,imm8               ; 80 /2 ib           [8086]
ADC r/m16,imm16             ; o16 81 /2 iw       [8086]
ADC r/m32,imm32             ; o32 81 /2 id       [386]

ADC r/m16,imm8              ; o16 83 /2 ib       [8086]
ADC r/m32,imm8              ; o32 83 /2 ib       [386]

ADC AL,imm8                 ; 14 ib              [8086]
ADC AX,imm16                ; o16 15 iw          [8086]
ADC EAX,imm32               ; o32 15 id          [386]
```

'ADC' realiza la suma entera: suma los dos operandos, más el valor del flag de acarreo, y deja el resultado en su operando (el primero) destino. Los flag se activan de acuerdo al resultado de la operación: en particular, se afecta al flag de acarreo y una instrucción 'ADC' subsiguiente puede hacer uso de él.

En las formas con un inmediato de 8-bit como segundo operando y un primer operando más largo, el segundo operando se considera con signo, y se hace extensión de signo hasta la longitud del primer operando. En estos casos, el calificador 'BYTE' se necesita para forzar a NASM a generar esta forma de la instrucción.

Para sumar dos números sin sumar el contenido del flag de acarreo, usa 'ADD' (sección A.6).

A.6 'ADD': Sumar enteros

```
ADD r/m8,reg8                ; 00 /r              [8086]
ADD r/m16,reg16              ; o16 01 /r          [8086]
ADD r/m32,reg32              ; o32 01 /r          [386]

ADD reg8,r/m8                ; 02 /r              [8086]
ADD reg16,r/m16              ; o16 03 /r          [8086]
ADD reg32,r/m32              ; o32 03 /r          [386]

ADD r/m8,imm8                ; 80 /0 ib           [8086]
ADD r/m16,imm16              ; o16 81 /0 iw       [8086]
ADD r/m32,imm32              ; o32 81 /0 id       [386]

ADD r/m16,imm8               ; o16 83 /0 ib       [8086]
ADD r/m32,imm8               ; o32 83 /0 ib       [386]

ADD AL,imm8                  ; 04 ib              [8086]
ADD AX,imm16                 ; o16 05 iw          [8086]
ADD EAX,imm32                ; o32 05 id          [386]
```

'ADD' realiza la suma entera: suma sus dos operandos, y deja el resultado en el operando (el primero) destino. Los flags se activan de acuerdo al resultado de la operación: en particular, se afecta al flag de acarreo y puede usarse en subsiguientes instrucciones 'ADC' (sección A.5).

En las formas con un inmediato de 8-bit como segundo operando y un primer operando más largo, el segundo operando se considera con signo, y se le hace extensión de signo hasta la longitud del primer operando. En estos casos, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

A.7 'AND': AND a nivel de bit

```
AND r/m8,reg8                ; 20 /r              [8086]
AND r/m16,reg16              ; o16 21 /r          [8086]
AND r/m32,reg32              ; o32 21 /r          [386]

AND reg8,r/m8                ; 22 /r              [8086]
AND reg16,r/m16              ; o16 23 /r          [8086]
AND reg32,r/m32              ; o32 23 /r          [386]

AND r/m8,imm8                ; 80 /4 ib           [8086]
AND r/m16,imm16              ; o16 81 /4 iw       [8086]
AND r/m32,imm32              ; o32 81 /4 id       [386]

AND r/m16,imm8               ; o16 83 /4 ib       [8086]
AND r/m32,imm8               ; o32 83 /4 ib       [386]

AND AL,imm8                  ; 24 ib              [8086]
AND AX,imm16                 ; o16 25 iw          [8086]
AND EAX,imm32                ; o32 25 id          [386]
```

'AND' realiza la operación AND entre bits entre sus dos operandos (esto es, cada bit del resultado es 1 si y sólo si los bits correspondientes de las dos entradas son ambos 1), y almacena el resultado en el operando (el primero) destino.

En las formas con un inmediato de 8-bit como segundo operando y un primer operando más largo, el segundo operando se considera con signo, y se le hace extensión de signo hasta la longitud del primer operando. En estos

casos, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

La instrucción MMX 'PAND' (ver ka sección A.116) realiza la misma operación en los registros MMX de 64-bit.

A.8 'ARPL': Ajusta el campo RPL del selector

```
ARPL r/m16,reg16            ; 63 /r               [286,PRIV]
```

'ARPL' espera que sus dos palabras operando sean selectores de segmento. Esto ajusta el campo RPL (se requiere nivel de privilegio - almacenado en los dos bits más inferiores del selector) del operando (el primero) destino para asegurar que no es menos (esto es, no es más privilegiado que) el campo RPL del operando fuente. Se activa el flag de cero si y sólo si se ha hecho algún cambio.

A.9 'BOUND': Comprueba los índices de array contra redondeos

```
BOUND reg16,mem             ; o16 62 /r           [186]
BOUND reg32,mem             ; o32 62 /r           [386]
```

'BOUND' espera que su segundo operando apunte a un área de memoria que contenga dos valores con signo del mismo tamaño que su primer operando (esto es, dos palabras para la forma de 16-bit; dos palabras dobles para la forma de 32-bit). Esto realiza dos comparaciones con signo: si el valor en el registro pasado como primer operando es menor que el primero de los valores en memoria, o es mayor o igual que el segundo, lanza una excepción BR. De otro modo, no hace nada.

A.10 'BSF', 'BSR': Examinar bits

```
BSF reg16,r/m16             ; o16 0F BC /r        [386]
BSF reg32,r/m32             ; o32 0F BC /r        [386]

BSR reg16,r/m16             ; o16 0F BD /r        [386]
BSR reg32,r/m32             ; o32 0F BD /r        [386]
```

'BSF' busca un conjunto de bits en su operando (el segundo) fuente, empezando desde la parte inferior, y si encuentra uno, almacena el índice en su operando (el primero) destino. Si no se encuentra el conjunto de bits, el contenido del operando destino está indefinido.

'BSR' realiza la misma función, pero en su lugar busca desde la parte superior, por eso encuentra el conjunto de bits más significativos.

Los índices de los bits son desde 0 (menos significante) hasta 15 o 31 (más significante).

A.11 'BSWAP': Intercambio de bytes

```
BSWAP reg32                 ; o32 0F C8+r         [486]
```

'BSWAP' intercambia el orden de los cuatro bytes de un registro de 32-bit: los bits del 0-7 intercambian su lugar con los bits del 24-31, y los bits 8-15 se intercambian con los bits 16-23. No hay un equivalente explícito de 16-bit: para intercambiar los bytes en 'AX', 'BX', 'CX' o 'DX', se puede usar 'XCHG'.

A.12 'BT', 'BTC', 'BTR', 'BTS': Comprobación de bits

```
BT r/m16,reg16              ; o16 0F A3 /r          [386]
BT r/m32,reg32              ; o32 0F A3 /r          [386]
BT r/m16,imm8               ; o16 0F BA /4 ib       [386]
BT r/m32,imm8               ; o32 0F BA /4 ib       [386]

BTC r/m16,reg16             ; o16 0F BB /r          [386]
BTC r/m32,reg32             ; o32 0F BB /r          [386]
BTC r/m16,imm8              ; o16 0F BA /7 ib       [386]
BTC r/m32,imm8              ; o32 0F BA /7 ib       [386]

BTR r/m16,reg16             ; o16 0F B3 /r          [386]
BTR r/m32,reg32             ; o32 0F B3 /r          [386]
BTR r/m16,imm8              ; o16 0F BA /6 ib       [386]
BTR r/m32,imm8              ; o32 0F BA /6 ib       [386]

BTS r/m16,reg16             ; o16 0F AB /r          [386]
BTS r/m32,reg32             ; o32 0F AB /r          [386]
BTS r/m16,imm               ; o16 0F BA /5 ib       [386]
BTS r/m32,imm               ; o32 0F BA /5 ib       [386]
```

Todas estas instrucciones comprueban un bit de su primer operando, cuyo índice lo da el segundo operando, y almacena el valor de ese bit en el flag de acarreo. Los índices de los bits van desde 0 (menos significativo) hasta 15 o 31 (más significativo).

Adicionalmente al almacenar el valor original del bit en el flag de acarreo, 'BTR' también limpia el bit en el operando. 'BTS' activa el bit, y 'BTC' complementa el bit. 'BT' no modifica el operando.

El offset del bit no debería ser más grande que el tamaño del operando.

A.13 'CALL': Llamar a una subrutina

```
CALL imm                    ; E8 rw/rd              [8086]
CALL imm:imm16              ; o16 9A iw iw          [8086]
CALL imm:imm32              ; o32 9A id iw          [386]
CALL FAR mem16              ; o16 FF /3             [8086]
CALL FAR mem32              ; o32 FF /3             [386]
CALL r/m16                  ; o16 FF /2             [8086]
CALL r/m32                  ; o32 FF /2             [386]
```

'CALL' llama a una subrutina, mediante empujar el puntero de la instrucción actual ('IP') y opcionalmente 'CS', a la pila, y luego saltando a la dirección dada.

Se empuja 'CS' al igual que 'IP' si y sólo si la llamada es una llamada lejana, esto es, se especifica en la instrucción una dirección de segmento de destino. Las formas que implican dos argumentos separados por dos puntos son llamadas lejanas; por eso son las formas 'CALL FAR mem'.

Puedes elegir entre las dos formas inmediatas de llamadas lejanas ('CALL imm:imm') usando las palabras clave 'WORD' y 'DWORD': 'CALL WORD 0x1234:0x5678') o 'CALL DWORD 0x1234:0x56789abc'.

Las formas 'CALL FAR mem' ejecutan una llamada lejana mediante la carga de la dirección de destino por la memoria. La dirección cargada consisten en un offset de 16 o 32 bits (dependiendo del tamaño del operando), y 16 bits de segmento. El tamaño del operando puede superponerse usando 'CALL WORD FAR mem' o 'CALL DWORD FAR mem'.

Las formas 'CALL r/m' ejecutan una llamada cercana (dentro del mismo segmento), cargando la dirección de destino por memoria o por un registro. Podría especificarse la palabra clave 'NEAR', para mayor claridad, es estas formas, pero no es necesario. Nuevamente, se puede superponer el tamaño del operando usando 'CALL WORD mem' o 'CALL DWORD mem'.

Como conveniencia, NASM no te pide que llames a un procedimiento lejano codificando la molesta 'CALL SEG rutina:rutina', sino que en su lugar permite el sinónimo más fácil 'CALL FAR rutina'.

Las formas 'CALL r/m' dadas arriba son llamadas cercanas; NASM aceptará la palabra clave 'NEAR' (por ejemplo, 'CALL NEAR [address]'), incluso aunque no sea estrictamente necesario.

A.14 'CBW', 'CWD', 'CDQ', 'CWDE': Extensiones del signo

```
CBW                         ; o16 98            [8086]
CWD                         ; o16 99            [8086]
CDQ                         ; o32 99            [386]
CWDE                        ; o32 98            [386]
```

Todas estas instrucciones hacen extensión de signo de un valor corto a otro más largo, replicando el bit superior del valor original para rellenar el valor extendido.

'CBW' extiende 'AL' en 'AX' repitiendo el bit superior de 'AL' en cada uno de los bits de 'AH'. 'CWD' extiende 'AX' en 'DX:AX' repitiendo el bit superior de 'AX' por todo 'DX'. 'CWDE' extiende 'AX' en 'EAX', y 'CDQ' extiende 'EAX' en 'EDX:EAX'.

A.15 'CLC', CLD', 'CLI', 'CLTS': Limpiar flags

```
CLC                         ; F8                [8086]
CLD                         ; FC                [8086]
CLI                         ; FA                [8086]
CLTS                        ; 0F 06             [286,PRIV]
```

Estas instrucciones limpian diversos flags. 'CLC' limpia el flag de acarreo; 'CLD' limpia el flag de dirección; 'CLI' limpia el flag de interrupción (deshabilitando así las interrupciones); y 'CLTS' limpia el flag de cambio de tareas ('TS') en 'CR0'.

Para activar los flags de acarreo, dirección o interrupción, usa las instrucciones 'STC', 'STD' y 'STI' (sección A.156). Para invertir el flag de acarreo, usa 'CMC' (sección A.16).

A.16 'CMC': Complementa el flag de acarreo

```
CMC                         ; F5                [8086]
```

'CMC' cambia el valor del flag de acarreo: si era 0, se pone a 1, y viceversa.

A.17 'CMOVcc': 'Move' condicional

```
CMOVcc reg16,r/m16          ; o16 0F 40+cc /r   [P6]
CMOVcc reg32,r/m32          ; o32 0F 40+cc /r   [P6]
```

'CMOV' mueve su operando (el segundo) fuente a su operando (el primero) destino si se satisface el código condicional que se da; de otra forma no hace nada.

Para ver una lista de los códigos de condiciones, ver la sección A.2.2.

Aunque la instrucción 'CMOV' está marcada como 'P6' o superior, podría no estar soportada por todos los procesadores Pentium Pro; la instrucción 'CPUID' (sección A.22) devolverá un bit que indica si están soportados los movimientos condicionales.

A.18 'CMP': Compara enteros

```
CMP r/m8,reg8               ; 38 /r              [8086]
CMP r/m16,reg16             ; o16 39 /r          [8086]
CMP r/m32,reg32             ; o32 39 /r          [386]

CMP reg8,r/m8               ; 3A /r              [8086]
CMP reg16,r/m16             ; o16 3B /r          [8086]
CMP reg32,r/m32             ; o32 3B /r          [386]

CMP r/m8,imm8               ; 80 /0 ib           [8086]
CMP r/m16,imm16             ; o16 81 /0 iw       [8086]
CMP r/m32,imm32             ; o32 81 /0 id       [386]

CMP r/m16,imm8              ; o16 83 /0 ib       [8086]
CMP r/m32,imm8              ; o32 83 /0 ib       [386]

CMP AL,imm8                 ; 3C ib              [8086]
CMP AX,imm16                ; o16 3D iw          [8086]
CMP EAX,imm32               ; o32 3D id          [386]
```

'CMP' realiza una resta 'mental' de su segundo operando del primero, y afecta a los flags como si hubiese tenido lugar una resta, pero no almacena en ningún lugar el resultado de la resta.

En las formas con inmediatos de 8-bit como segundo operando y un primer operando más largo, el segundo operando se considera con signo, y se le hace extensión de signo hasta la longitud del primer operando. En estos casos, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

A.19 'CMPSB', 'CMPSW', 'CMPSD': Compara cadenas

```
CMPSB                       ; A6                 [8086]
CMPSW                       ; o16 A7             [8086]
CMPSD                       ; o32 A7             [386]
```

'CMPSB' compara el byte en '[DS:SI]' o '[DS:ESI]' con el byte en '[ES:DI]' o '[ES:EDI]', y activa los flag de manera acorde. Luego incrementa o decrementa (dependiendo del flag de dirección: incrementa si el flag está limpio, decrementa si está activo) 'SI' y 'DI' (o 'ESI' y 'EDI').

Los registros usados son 'SI' y 'DI' si el tamaño de la dirección es de 16 bits, y 'ESI' y 'EDI' si son de 32 bits. Si necesitas usar un tamaño de dirección diferente al actual 'BITS', puedes usar explícitamente un prefijo 'a16' o 'a32'.

El registro de segmento usado para cargar desde '[SI]' o '[ESI]' se puede

superponer usando el nombre de un registro de segmento como un prefijo (por ejemplo, 'es cmpsb'). El uso de 'ES' para cargar desde '[DI]' o '[EDI]' no se puede superponer.

'CMPSW' y 'CMPSD' funcionan del mismo modo, pero comparan una palabra o una palabra doble en lugar de un byte, e incrementan o decrementan los registros de direccionamiento en 2 ó 4 en lugar de en 1.

Los prefijos 'REPE' y 'REPNE' (de forma equivalente, 'REPZ' y 'REPNZ') podría usarse para repetir la instrucción hasta 'CX' veces (o 'ECX' - nuevamente el tamaño de la dirección determina cuál) hasta que se encuentre el primer byte igual o diferente.

A.20 'CMPXCHG', 'CMPCHG486': Compara e intercambia

```
CMPXCHG r/m8,reg8            ; 0F B0 /r            [PENT]
CMPXCHG r/m16,reg16          ; o16 0F B1 /r        [PENT]
CMPXCHG r/m32,reg32          ; o32 0F B1 /r        [PENT]

CMPXCHG486 r/m8,reg8         ; 0F A6 /r            [486,UNDOC]
CMPXCHG486 r/m16,reg16       ; o16 0F A7 /r        [486,UNDOC]
CMPXCHG486 r/m32,reg32       ; o32 0F A7 /r        [486,UNDOC]
```

Estas dos instrucciones realizan exactamente la misma operación; sin embargo, aparentemente, algunos (no todos) procesadores 486 las soportan bajo un código de operación no estándar, por eso NASM proporciona la forma no documentada de 'CMPXCHG486' para generar el código de operación no estándar.

'CMPXCHG' compara su operando (el primero) destino con el valor en 'AL', 'AX' o 'EAX' (dependiendo del tamaño de la instrucción). Si son iguales, copia su operando (el segundo) origen en el destino y activa el flag de cero. De otro modo, limpia el flag de cero y deja el destino solo.

'CMPXCHG' viene bien usarlo para operaciones atómicas en entornos multitarea o multiprocesador. Para actualizar de forma segura un valor en memoria compartida, por ejemplo, podrías cargar el valor en 'EAX', cargar el valor actualizado en 'EBX', y luego ejecutar la instrucción 'lock cmpxchg [value],ebx'. Si 'value' no ha cambiado desde que fue cargado, es actualizado con tu nuevo valor deseado, y se activa el flag de cero para hacerte saber que ha funcionado. (El prefijo 'LOCK' previene el que otro procesador haga otra cosa en mitad de esta operación: garantiza la atomicidad.) Sin embargo, si otro procesador ha modificado el valor entre tu carga y tu intento de almacenarlo, el almacenaje no tiene lugar, y se te avisa del fallo mediante un flag de cero limpio, por eso puedes volver atrás y volverlo a intentar.

A.21 'CMPXCHG8B': Compara e interacambia ocho bytes

```
CMPXCHG8B mem                ; 0F C7 /1            [PENT]
```

Esta es una versión mayor y más difícil de usar de 'CMPXCHG': compara el valor de 64-bit (ocho bytes) almacenado en '[mem]' con el valor en 'EDX:EAX'. Si son iguales, se activa el flag de cero y almacena 'ECX:EBX' en el área de memoria. Si son diferentes, limpia el flag de cero y deja sin tocar el área de memoria.

A.22 'CPUID': Coge el código de identificación de la CPU

```
CPUID                        ; 0F A2               [PENT]
```

'CPUID' devuelve diversa información sobre el procesador en que está siendo ejecutada. Llena los cuatro registros 'EAX', 'EBX', 'ECX' y 'EDX' con información, que varía dependiendo del contenido de entrada de 'EAX'.

'CPUID' también actúa como una barrera para serializar la ejecución de las instrucciones: ejecutar la instrucción 'CPUID' te garantiza que todos los efectos (modificaciones de memoria, modificaciones de flags, modificaciones de los registros) de las instrucciones previas se han completado antes de que se prepare la siguiente instrucción.

La información devuelta es como sigue:

(*) Si 'EAX' es cero en la entrada, 'EAX' en la salida contiene el máximo valor de entrada de 'EAX', y 'EAB:EDX:ECX' contiene la cadena '"GenuineIntel"' (o no, si tienes un procesador clónico). Esto es decir, 'EBX' contiene '"Genu"' (en el sentido propio de las constantes carácter en NASM, descrito en la sección 3.4.2), 'EDX' contiene '"ineI"' y 'ECX' contiene '"ntel"'.

(*) Si 'EAX' vale uno en la entrada, 'EAX' en la salida contiene la versión de la información sobre el procesador, y 'EDX' contiene un conjunto de flag característicos, mostrando la presencia y ausencia de distintas características. Por ejemplo, el bit 8 se activa si se soporta la instrucción 'CMPXCHG8B' (sección A.21), el bit 15 se activa si se soportan las instrucciones de movimientos condicionales (sección A.17 y sección A.34), y el bit 23 se activa si se soportan instrucciones MMX.

(*) Si 'EAX' tiene un dos en la entrada, 'EAX', 'EBX', 'ECX' y 'EDX' contienen información sobre cachés y TLBs (Translation Lookahead Buffers).

Para ver más información sobre los datos devueltos desde 'CPUID', ver la documentación en el Web de Intel.

A.23 'DAA', 'DAS': Ajustes decimales

```
DAA                         ; 27                [8086]
DAS                         ; 2F                [8086]
```

Estas instrucciones se usan en conjunción con las instrucciones de suma y resta para realizar aritmética binaria codificada decimal en forma _empaquetada_ (un dígito BCD por cada nibble). Para ver las equivalentes desempaquetadas, ir a la sección A.4.

'DAA' debería usarse después de una instrucción 'ADD' de un byte cuyo destino fuese el registro 'AL': examinando el valor en el 'AL' y también al flag auxiliar 'AF', determina si un dígito ha sufrido desbordamiento, y lo ajusta si es necesario (y activa los flag de acarreo y auxiliar). Puedes sumar largas cadenas BCD haciendo 'ADD'/'DAA' en los dos dígitos inferiores, luego haciendo 'ADC'/'DAA' en cada par de dígitos subsiguientes.

'DAS' funciona de modo similar a 'DAA', pero es para usar después de instrucciones 'SUB' en lugar de 'ADD'.

A.24 'DEC': Decremento entero

```
DEC reg16                   ; o16 48+r          [8086]
```

```
DEC reg32                   ; o32 48+r          [386]
DEC r/m8                    ; FE /1             [8086]
DEC r/m16                   ; o16 FF /1         [8086]
DEC r/m32                   ; o32 FF /1         [386]
```

'DEC' resta 1 de su operando. _No_ afecta al flag de acarreo: para afectar al flag de acarreo, usa 'SUB something,1' (ver la sección A.159). Ver también 'INC' (sección A.79).

A.25 'DIV': División entera sin signo

```
DIV r/m8                    ; F6 /6             [8086]
DIV r/m16                   ; o16 F7 /6         [8086]
DIV r/m32                   ; o32 F7 /6         [386]
```

'DIV' realiza la división entera sin signo. El operando explícito proporcionado es el divisor; los operandos dividendo y destino están implícitos, del siguiente modo:

(*) Para 'DIV r/m8', 'AX' se divide por el operando dado; el cociente se almacena en 'AL' y el resto en 'AH'.

(*) Para 'DIV r/m16', 'DX:AX' se divide por el operando dado; el cociente se almacena en 'AX' y el resto en 'DX'.

(*) Para 'DIV r/m32', 'EDX:EAX' se divide por el operando dado; el cociente se almacena en 'EAX' y el resto en 'EDX'.

La división entera con signo se realiza con la instrucción 'IDIV': ver la sección A.76.

A.26 'EMMS': Estado MMX vacío

```
EMMS                        ; 0F 77             [PENT,MMX]
```

'EMMS' activa la palabra etiqueta FPU (indicando que registros de coma flotante están disponibles) a todos, lo que quiere decir que todos los registros están disponibles para la FPU. Debería usarse después de ejecutar instrucciones MMX y antes de ejecutar cualquier operación de coma flotante subsiguiente.

A.27 'ENTER': Crea un marco de pila

```
ENTER imm,imm               ; C8 iw ib          [186]
```

'ENTER' construye un marco de pila para una llamada a un procedimiento de alto nivel. El primer operando (el 'iw' en la definición del código de operación de arriba se refiere al primer operando) da la cantidad de espacio de pila para asignar las variables locales; el segundo (el 'ib' de arriba) da el nivel de anidamiento del procedimiento (para lenguajes como Pascal, con procedimientos anidados).

La función 'ENTER', con un nivel de anidamiento de cero, es equivalente a

```
        PUSH EBP            ; o PUSH BP         en 16 bits
        MOV EBP,ESP         ; o MOV BP,SP       en 16 bits
        SUB ESP,operand1    ; o SUB SP,operand1 en 16 bits
```

Esto crea un marco de pila con los parámetros del procedimiento

accesibles hacia arriba de 'EBP', y las variables locales accesibles desde 'EBP' hacia abajo..

Con un nivel de anidamiento de uno, el marco de pila creado es 4 (ó 2) bytes más grande, y el valor del puntero de marco final 'EBP' está accesible en memoria en '[EBP-4]'.

Esto permite a 'ENTER', cuando es llamada con un nivel de anidamiento de dos, considerando el marco de pila descrito por el valor _previo_ de 'EBP', encontrar el puntero de marco en un offset -4 desde éste, y empujarlo con el nuevo puntero de marco, por eso cuando se llama a un procedimiento de nivel dos desde un procedimiento de nivel uno, '[EBP-4]' contiene el puntero de marco de la llamada de procedimiento de nivel uno más reciente y '[EBP-8]' contiene la llamada de procedimiento de nivel dos más reciente. Y así, para niveles de anidamiento de hasta nivel 31.

Los marcos de pila creados con 'ENTER' se pueden destruir con la instrucción 'LEAVE': ver la sección A.94.

A.28 'F2XM1': Calcula 2**X-1

```
F2XM1                       ; D9 F0             [8086,FPU]
```

'F2XM1' eleva 2 a la potencia de 'ST0', resta uno, y almacena el resultado de nuevo en 'ST0'. El contenido inicial de 'ST0' debe ser un número del rango -1 a +1.

A.29 'FABS': Valor absoluto de coma flotante

```
FABS                        ; D9 E1             [8086,FPU]
```

'FABS' calcula el valor absoluto de 'ST0', almacenando el resultado de nuevo en 'ST0'.

A.30 'FADD', 'FADDP': Suma en coma flotante

```
FADD mem32                  ; D8 /0             [8086,FPU]
FADD mem64                  ; DC /0             [8086,FPU]

FADD fpureg                 ; D8 C0+r           [8086,FPU]
FADD ST0,fpureg             ; D8 C0+r           [8086,FPU]

FADD TO fpureg              ; DC C0+r           [8086,FPU]
FADD fpureg,ST0             ; DC C0+r           [8086,FPU]

FADDP fpureg                ; DE C0+r           [8086,FPU]
FADDP fpureg,ST0            ; DE C0+r           [8086,FPU]
```

'FADD', dado un operando, suma el operando a 'ST0' y almacena el resultado de nuevo en 'ST0'. Si el operando tiene el modificador 'TO', el resultado se almacena en el registro dado en lugar de en 'ST0'.

'FADDP' realiza la misma función que 'FADD TO', pero saca la pila de registros después de almacenar el resultado.

La forma de dos operandos es sinónimo de la forma de un operando.

A.31 'FBLD', 'FBSTP': Carga y almacenamiento de BCD en coma flotante

```
FBLD mem80                  ; DF /4             [8086,FPU]
FBSTP mem80                 ; DF /6             [8086,FPU]
```

'FBLD' carga un número empaquetado codificado decimal de 80-bits (diez bytes) desde la dirección de memoria dada, lo convierte a real, y lo empuja a la pila de registros. 'FBSTP' almacena el valor de 'ST0', en BCD empaquetado, en la dirección dada y luego saca la pila de registros.

A.32 'FCHS': Cambio de signo en coma flotante

```
FCHS                        ; D9 E0             [8086,FPU]
```

'FCHS' niega el número en 'ST0': los números negativos se vuelven positivos, y viceversa.

A.33 'FCLEX', {FNCLEX}: Limpia las excepciones de coma flotante

```
FCLEX                       ; 9B DB E2          [8086,FPU]
FNCLEX                      ; DB E2             [8086,FPU]
```

'FCLEX' limpia cualquier excepción de coma flotante que pudiese estar pendiente. 'FNCLEX' hace lo mismo pero no espera a que se acaben primero las operaciones de coma flotante previas (incluyendo el _manejo_ de excepciones pendientes).

A.34 'FCMOVcc': Movimientos condicionales en coma flotante

```
FCMOVB fpureg               ; DA C0+r           [P6,FPU]
FCMOVB ST0,fpureg           ; DA C0+r           [P6,FPU]

FCMOVBE fpureg              ; DA D0+r           [P6,FPU]
FCMOVBE ST0,fpureg          ; DA D0+r           [P6,FPU]

FCMOVE fpureg               ; DA C8+r           [P6,FPU]
FCMOVE ST0,fpureg           ; DA C8+r           [P6,FPU]

FCMOVNB fpureg              ; DB C0+r           [P6,FPU]
FCMOVNB ST0,fpureg          ; DB C0+r           [P6,FPU]

FCMOVNBE fpureg             ; DB D0+r           [P6,FPU]
FCMOVNBE ST0,fpureg         ; DB D0+r           [P6,FPU]

FCMOVNE fpureg              ; DB C8+r           [P6,FPU]
FCMOVNE ST0,fpureg          ; DB C8+r           [P6,FPU]

FCMOVNU fpureg              ; DB D8+r           [P6,FPU]
FCMOVNU ST0,fpureg          ; DB D8+r           [P6,FPU]

FCMOVU fpureg               ; DA D8+r           [P6,FPU]
FCMOVU ST0,fpureg           ; DA D8+r           [P6,FPU]
```

La instrucción 'FCMOV' realiza operaciones de movimientos condicionales: cada una de ellas mueve el contenido del registro dado en 'ST0' si se satisface la condición, y si no, no hace nada.

Las condiciones no son las mismas que los códigos de condición estándar usados en las instrucciones de salto condicional. Las condiciones 'B', 'BE', 'NB', 'NBE', 'E' y 'NE' son exactamente como las normales, pero no se soporta ninguno de los otros códigos estándar. En su lugar, se

proporcionaba la condición 'U' y su contrapartida 'NU'; la condición 'U' se satisface si los dos últimos números en coma flotante comparados estaban _desordenados_, esto es, no son iguales pero no se podría decir si uno es mayor o no que el otro, por ejemplo, si fuesen NaNs. (El flag de estado que señala esto es el flag de paridad: por eso la noción de la condición 'U' es equivalente a 'PE', y 'NU' es equivalente a 'PO').

Las condiciones 'FCMOV' comprueban el flag de estado del procesador, no los flags de estado de la FPU, por eso usar 'FCMOV' directamente después de 'FCOM' no funcionará. En su lugar, deberías usar o 'FCOMI' que escribe directamente en la palabra de flags de la CPU principal, o usar 'FSTSW' para extraer los flags de estado de la FPU.

Aunque las instrucciones 'FCMOV' están marcadas como 'P6' o superior, podrían no estar soportadas por todos los procesadores Pentium Pro; la instrucción 'CPUID' (sección A.22) devolverá un bit que indica si se soportan los movimientos condicionales.

A.35 'FCOM', 'FCOMP', 'FCOMPP', 'FCOMI', 'FCOMIP': Compara en coma flotante

```
FCOM mem32                  ; D8 /2              [8086,FPU]
FCOM mem64                  ; DC /2              [8086,FPU]
FCOM fpureg                 ; D8 D0+r            [8086,FPU]
FCOM ST0,fpureg             ; D8 D0+r            [8086,FPU]

FCOMP mem32                 ; D8 /3              [8086,FPU]
FCOMP mem64                 ; DC /3              [8086,FPU]
FCOMP fpureg                ; D8 D8+r            [8086,FPU]
FCOMP ST0,fpureg            ; D8 D8+r            [8086,FPU]

FCOMPP                      ; DE D9              [8086,FPU]

FCOMI fpureg                ; DB F0+r            [P6,FPU]
FCOMI ST0,fpureg            ; DB F0+r            [P6,FPU]

FCOMIP fpureg               ; DF F0+r            [P6,FPU]
FCOMIP ST0,fpureg           ; DF F0+r            [P6,FPU]
```

'FCOM' compara con el operando dado, y activa los flags de la FPU de manera acorde. 'ST0' es tratado como el lado izquierdo de la comparación, por eso se activa el flag de acarreo (para un resultado 'menor que') si 'ST0' es menor que el operando dado.

'FCOMP' hace lo mismo que 'FCOM' pero saca después la pila de registros. 'FCOMPP' compara 'ST0' con 'ST1' y luego saca el pila de registros dos veces.

'FCOMI' y 'FCOMIP' funcionan como sus formas correspondientes de 'FCOM' y 'FCOMP', pero escriben directamente sus resultados en el registro de flags de la CPU en lugar de la palabra de estado de la FPU, por eso pueden estar seguidos inmediatamente de instrucciones de salto condicional o de movimientos condicionales.

Las instrucciones 'FCOM' difieren de las instrucciones 'FUCOM' (sección A.69) sólo en el modo en que manejan los tranquilos NaNs: 'FUCOM' los manejará silenciosamente y activará los flags de los códigos de condición para un resultado 'desordenado', mientras que 'FCOM' generará una excepción.

A.36 'FCOS': Coseno

```
FCOS                        ; D9 FF             [386,FPU]
```

'FCOS' calcula el coseno de 'ST0' (en radianes), y almacena el resultado en 'ST0'. Ver también 'FSINCOS' (sección A.61).

A.37 'FDECSTP': Decrementa el puntero de pila en coma flotante

```
FDECSTP                     ; D9 F6             [8086,FPU]
```

'FDECSTP' decremente el campo 'superior' en la palabra de estado en coma flotante. Esto tiene el efecto de rotar en uno la pila de registros de la FPU, como si el contenido de 'ST7' hubiese sido empujado a la pila. Ver también 'FINCSTP' (sección A.46).

A.38 'FxDISI', 'FxENI': Desactiva y activa las interrupciones en coma flotante

```
FDISI                       ; 9B DB E1          [8086,FPU]
FNDISI                      ; DB E1             [8086,FPU]

FENI                        ; 9B DB E0          [8086,FPU]
FNENI                       ; DB E0             [8086,FPU]
```

'FDISI' y 'FENI' desactiva y activa las interrupciones de coma flotante. Estas instrucciones sólo son significativas en los procesadores 8087 originales: el 287 y superiores las trata como instrucciones que no hacen ninguna operación.

'FNDISI' y 'FNENI' hacen la misma cosa que 'FDISI' y 'FENI' respectivamente, pero sin esperar a que el procesador de coma flotante haya acabado primero con lo que estuviese haciendo.

A.39 'FDIV', 'FDIVP', FDIVR', 'FDIVRP': División en coma flotante

```
FDIV mem32                  ; D8 /6             [8086,FPU]
FDIV mem64                  ; DC /6             [8086,FPU]

FDIV fpureg                 ; D8 F0+r           [8086,FPU]
FDIV ST0,fpureg             ; D8 F0+r           [8086,FPU]

FDIV TO fpureg              ; DC F8+r           [8086,FPU]
FDIV fpureg,ST0             ; DC F8+r           [8086,FPU]

FDIVR mem32                 ; D8 /0             [8086,FPU]
FDIVR mem64                 ; DC /0             [8086,FPU]

FDIVR fpureg                ; D8 F8+r           [8086,FPU]
FDIVR ST0,fpureg            ; D8 F8+r           [8086,FPU]

FDIVR TO fpureg             ; DC F0+r           [8086,FPU]
FDIVR fpureg,ST0            ; DC F0+r           [8086,FPU]

FDIVP fpureg                ; DE F8+r           [8086,FPU]
FDIVP fpureg,ST0            ; DE F8+r           [8086,FPU]

FDIVRP fpureg               ; DE F0+r           [8086,FPU]
FDIVRP fpureg,ST0           ; DE F0+r           [8086,FPU]
```

'FDIV' divide 'ST0' por el operando dado y almacena el resultado de nuevo en 'ST0', a menos que se de el calificador 'TO', en cuyo caso divide el

operando dado por 'ST0' y almacena el resultado en el operando.

'FDIVR' hace lo mismo, pero hace la división de la otra forma: por eso si no se da 'TO', divide el operando dado por 'ST0' y almacena el resultado en 'ST0', mientras que si se da 'TO', divide 'ST0' por el operando y almacena el resultado en el operando.

'FDIVP' opera como 'FDIV TO', pero saca la pila de registros una vez haya terminado. 'FDIVRP' opera de igual forma que 'FDIVR TO', pero saca la pila de registros una vez que haya terminado.

A.40 'FFREE': Pone el registro de flags de coma flotante como no usado

```
FFREE fpureg                ; DD C0+r             [8086,FPU]
```

'FFREE' marca el registro dado como si estuviese vacío.

A.41 'FIADD': Suma coma flotante/entero

```
FIADD mem16                 ; DE /0               [8086,FPU]
FIADD mem32                 ; DA /0               [8086,FPU]
```

'FIADD' suma el entero de 16-bit o 32-bit almacenado en la posición de memoria dado, a 'ST0', almacenando el resultado en 'ST0'.

A.42 'FICOM', 'FICOMP': Compara coma flotante/entero

```
FICOM mem16                 ; DE /2               [8086,FPU]
FICOM mem32                 ; DA /2               [8086,FPU]

FICOMP mem16                ; DE /3               [8086,FPU]
FICOMP mem32                ; DA /3               [8086,FPU]
```

'FICOM' compara 'ST0' con el entero de 16-bit o 32-bit almacenado en la posición de memoria dada, y activa de forma acorde los flags FPU. 'FICOMP' hace lo mismo, pero saca después la pila de registros.

A.43 'FIDIV', 'FIDIVR': División coma flotante/entero

```
FIDIV mem16                 ; DE /6               [8086,FPU]
FIDIV mem32                 ; DA /6               [8086,FPU]

FIDIVR mem16                ; DE /0               [8086,FPU]
FIDIVR mem32                ; DA /0               [8086,FPU]
```

'FIDIV' divide 'ST0' por el entero de 16-bit o 32-bit almacenado en la posición de memoria dada, y almacena el resultado en 'ST0'. 'FIDIVR' hace la división del otro modo: divide en entero por 'ST0', pero sigue almacenando el resultado en 'ST0'.

A.44 'FILD', 'FIST', 'FISTP': Conversión coma flotante/entero

```
FILD mem16                  ; DF /0               [8086,FPU]
FILD mem32                  ; DB /0               [8086,FPU]
FILD mem64                  ; DF /5               [8086,FPU]

FIST mem16                  ; DF /2               [8086,FPU]
FIST mem32                  ; DB /2               [8086,FPU]

FISTP mem16                 ; DF /3               [8086,FPU]
```

```
FISTP mem32                 ; DB /3             [8086,FPU]
FISTP mem64                 ; DF /0             [8086,FPU]
```

'FILD' carga un entero desde una dirección de memoria, lo convierte a un real, y lo empuja en la pila de registros de la FPU. 'FIST' convierte 'ST0' a un entero y lo almacena en memoria; 'FIST' hace lo mismo que 'FIST', pero saca después de la pila de registros.

A.45 'FIMUL': Multiplicación coma flotante/entero

```
FIMUL mem16                 ; DE /1             [8086,FPU]
FIMUL mem32                 ; DA /1             [8086,FPU]
```

'FIMUL' multiplica 'ST0' por el entero de 16-bit o 32-bit almacenado en la posición de memoria dada, y almacena el resultado en 'ST0'.

A.46 'FINCSTP': Incrementa el puntero de pila de coma flotante

```
FINCSTP                     ; D9 F7             [8086,FPU]
```

'FINCSTP' incrementa el campo 'superior' en la palabra de estado de coma flotante. Esto tiene el efecto de rotar en uno la pila de registros, como si la pila de registros hubiese sido sacada; sin embargo, al contrario que el 'pop' hecho por muchas instrucciones FPU, no marca el nuevo 'ST7' (anteriormente 'ST0') como vacío. Ver también 'FDECSTP' (sección A.37).

A.47 'FINIT', 'FNINIT': Inicializa la unidad de coma flotante

```
FINIT                       ; 9B DB E3          [8086,FPU]
FNINIT                      ; DB E3             [8086,FPU]
```

'FINIT' inicializa la FPU a su estado por defecto. Marca todos los registros como vacíos, aunque realmente no cambie sus valores. 'FNINIT' hace lo mismo, pero sin esperar a que las excepciones pendientes de hayan limpiado.

A.48 'FISUB': Resta coma flotante/entero

```
FISUB mem16                 ; DE /4             [8086,FPU]
FISUB mem32                 ; DA /4             [8086,FPU]

FISUBR mem16                ; DE /5             [8086,FPU]
FISUBR mem32                ; DA /5             [8086,FPU]
```

'FISUB' resta de 'ST0' el entero de 16-bit o 32-bit almacenado en la posición de memoria dada, y almacena el resultado en 'ST0'. 'FISIBR' hace la resta del otro modo, esto es, resta 'ST0' del entero dado, pero sigue almacenando el resultado en 'ST0'.

A.49 'FLD': Carga en coma flotante

```
FLD mem32                   ; D9 /0             [8086,FPU]
FLD mem64                   ; DD /0             [8086,FPU]
FLD mem80                   ; DB /5             [8086,FPU]
FLD fpureg                  ; D9 C0+r           [8086,FPU]
```

'FLD' carga un valor en coma flotante desde el registro o posición de memoria dada, y lo empuja a la pila de registros de la FPU.

A.50 'FLDxx': Carga de constantes en coma flotante

```
FLD1                        ; D9 E8             [8086,FPU]
FLDL2E                      ; D9 EA             [8086,FPU]
FLDL2T                      ; D9 E9             [8086,FPU]
FLDLG2                      ; D9 EC             [8086,FPU]
FLDLN2                      ; D9 ED             [8086,FPU]
FLDPI                       ; D9 EB             [8086,FPU]
FLDZ                        ; D9 EE             [8086,FPU]
```

Estas instrucciones empujan una constante estándar específica a la pila de registros de la FPU. 'FLD1' empuja el valor 1; 'FLDL2E' empuja el logaritmo de e en base 2; 'FLDL2T' empuja el logaritmo de 10 en base 2; 'FLDLG2' empuja el logaritmo de 2 en base 10; 'FLDLN2' empuja el logaritmo de 2 en base e; 'FLDPI' empuja pi; y 'FLDZ' empuja cero.

A.51 'FLDCW': Carga la palabra de control de coma flotante

```
FLDCW mem16                 ; D9 /5             [8086,FPU]
```

'FLDCW' carga un valor de 16-bit desde memoria y lo almacena en la palabra de control de la FPU (que gobierna cosas como el modo de redondeo, la precisión, y las máscaras de excepciones). Ver también 'FSTCW' (sección A.64).

A.52 'FLDENV': Carga el entorno de coma flotante

```
FLDENV mem                  ; D9 /4             [8086,FPU]
```

'FLDENV' carga el entorno de operación de la FPU (palabra de control, palabra de estado, palabra etiqueta, puntero de instrucción, puntero de datos y último código de operación) desde memoria. El área de memoria es de 14 o 28 bytes de largo, dependiendo del modo de la CPU en ese momento. Ver también 'FSTENV' (sección A.65).

A.53 'FMUL', 'FMULP': Multiplicación en coma flotante

```
FMUL mem32                  ; D8 /1             [8086,FPU]
FMUL mem64                  ; DC /1             [8086,FPU]

FMUL fpureg                 ; D8 C8+r           [8086,FPU]
FMUL ST0,fpureg             ; D8 C8+r           [8086,FPU]

FMUL TO fpureg              ; DC C8+r           [8086,FPU]
FMUL fpureg,ST0             ; DC C8+r           [8086,FPU]

FMULP fpureg                ; DE C8+r           [8086,FPU]
FMULP fpureg,ST0            ; DE C8+r           [8086,FPU]
```

'FMUL' multiplica 'ST0' por el operando dado, y almacena el resultado en 'ST0', a menos que se use el calificador 'TO' en cuyo caso se almacena el resultado en el operando. 'FMULP' realiza la misma operación que 'FMUL TO', y luego saca la pila de registros.

A.54 'FNOP': No_operación en coma flotante

```
FNOP                        ; D9 D0             [8086,FPU]
```

'FNOP' no hace nada.

A.55 'FPATAN', 'FPTAN': Arcotangente y tangente

```
FPATAN                          ; D9 F3             [8086,FPU]
FPTAN                           ; D9 F2             [8086,FPU]
```

'FPATAN' calcula el arcotangente, en radianes, del resultado de dividir 'ST1' por 'ST0', almacena el resultado en 'ST1', y saca la pila de registros. Funciona como la función 'atan2' de C, en que cambiando el signo tanto de 'ST0' como de 'ST1' cambia el valor de salida en pi (por eso realiza una verdadera conversión de coordenadas rectangulares a polares, siendo 'ST1' la coordenada Y y 'ST0' la X, no simplemente un arcotangente).

'FPTAN' calcula la tangente del valor en 'ST0' (en radianes), y almacena el resultado de nuevo en 'ST0'.

A.56 'FPREM', 'FPREM1': Restos parciales en coma flotante

```
FPREM                           ; D9 F8             [8086,FPU]
FPREM1                          ; D9 F5             [386,FPU]
```

Estas instrucciones producen el resto obtenido de dividir 'ST0' por 'ST1'. Esto se calcula, conceptualmente, dividiendo 'ST0' por 'ST1', redondeando el resultado a un entero, multiplicando 'ST1' nuevamente, y calculando el valor que necesitamos sumar al resultado para conseguir de nuevo el valor original en 'ST0'.

Las dos instrucciones difieren en el modo en que se realiza la operación de redondeo a entero. 'FPREM' lo hace redondeando hacia cero, por eso el resto que devuelve siempre tiene el mismo signo que el valor original en 'ST0'; 'FPREM1' hace el redondeo hacia el entero más próximo, por eso el resto siempre tiene como mucho la mitad de la magnitud de 'ST1'.

Ambas instrucciones calculan restos _parciales_, lo que quiere decir que podrían no encargarse de proporcionar el resultado final, pero podrían dejar en su lugar resultado intermedios en 'ST0'. Si esto pasa, activarán el flag C2 en la palabra de estado de la FPU; por lo tanto, para calcular un resto, deberías ejecutar repetidamente 'FPREM' o 'FPREM1' hasta que se limpie C2.

A.57 'FRNDINT': Redondeo a entero en coma flotante

```
FRNDINT                         ; D9 FC             [8086,FPU]
```

'FRNDINT' redondea el contenido de 'ST0' a un entero, de acuerdo al modo actual de redondeo establecido en la palabra de control de la FPU, y almacena el resultado de nuevo en 'ST0'.

A.58 'FSAVE', 'FRSTOR': Salva/restaura el estado de la coma flotante

```
FSAVE mem                       ; 9B DD /6          [8086,FPU]
FNSAVE mem                      ; DD /6             [8086,FPU]

FRSTOR mem                      ; DD /4             [8086,FPU]
```

'FSAVE' salva entera la unidad de estado de la coma flotante, incluyendo toda la información salvada por 'FSTENV' (sección A.65) más el contenido de todos los registros, a un área de memoria de 94 o 108 bytes (dependiendo del modo de la CPU). 'FRSTOR' restaura el estado de la coma flotante desde ese mismo área de memoria.

'FNSAVE' hace lo mismo que 'FSAVE', pero sin esperar primero a que se limpien las excepciones pendientes de coma flotante.

A.59 'FSCALE': Escala valores en coma flotante por una potencia de dos

```
FSCALE                      ; D9 FD             [8086,FPU]
```

'FSCALE' escala un número por una potencia de dos: redondea 'ST1' hacia cero para obtener un entero, luego multiplica 'ST0' por dos a la potencia de ese entero, y almacena el resultado en 'ST0'.

A.60 'FSETPM': Activa el modo protegido

```
FSETPM                      ; DB E4             [286,FPU]
```

Esta instrucción inicializa el modo protegido en el coprocesador de coma flotante 287. Sólo tiene sentido en este procesador: el 387 y superiores tratan esta instrucción como una no_operación.

A.61 'FSIN', 'FSINCOS': Seno y coseno

```
FSIN                        ; D9 FE             [386,FPU]
FSINCOS                     ; D9 FB             [386,FPU]
```

'FSIN' calcula el seno de 'ST0' (en radianes) y almacena el resultado en 'ST0'. 'FSINCOS' hace lo mismo, pero empuja el coseno del mismo valor a la pila de registros, por eso el seno va a parar a 'ST1' y el coseno en 'ST0'. 'FSINCOS' es más rápido que ejecutar sucesivamente 'FSIN' y 'FCOS' (ver sección A.36).

A.62 'FSQRT': Raíz cuadrada en coma flotante

```
FSQRT                       ; D9 FA             [8086,FPU]
```

'FSQRT' calcula la raíz cuadrada de 'ST0' y almacena el resultado en 'ST0'.

A.63 'FST', 'FSTP': Almacena en coma flotante

```
FST mem32                   ; D9 /2             [8086,FPU]
FST mem64                   ; DD /2             [8086,FPU]
FST fpureg                  ; DD D0+r           [8086,FPU]

FSTP mem32                  ; D9 /3             [8086,FPU]
FSTP mem64                  ; DD /3             [8086,FPU]
FSTP mem80                  ; DB /0             [8086,FPU]
FSTP fpureg                 ; DD D8+r           [8086,FPU]
```

'FST' almacena el valor en 'ST0' a una posición de memoria dada o en otro registro de la FPU. 'FSTP' hace lo mismo, pero luego saca la pila de registros.

A.64 'FSTCW': Almacena la palabra de control de coma flotante

```
FSTCW mem16                 ; 9B D9 /0          [8086,FPU]
FNSTCW mem16                ; D9 /0             [8086,FPU]
```

'FSTCW' almacena la palabra de control de la FPU (que gobierna cosas como el modo de redondeo, la precisión, y la máscara de excepciones) en un área de memoria de 2 bytes. Ver también 'FLDCW' (sección A.51).

'FNSTCW' hace lo mismo que 'FSTCW', pero sin esperar a que se limpien las excepciones pendientes de coma flotante.

A.65 'FSTENV': Almacena el entorno de la coma flotante

```
FSTENV mem                  ; 9B D9 /6          [8086,FPU]
FNSTENV mem                 ; D9 /6             [8086,FPU]
```

'FSTENV' almacena el entorno de operación de la FPU (palabra de control, palabra de estado, palabra etiqueta, puntero de instrucción, puntero de datos y último código de operación) en memoria. El área de memoria es de 14 o 28 bytes de largo, dependiendo del modo de la CPU en ese momento. Ver también 'FLDENV' (sección A.52).

'FNSTENV' hace lo mismo que 'FSTENV', pero sin esperar primero a que se limpien las excepciones pendiente de la coma flotante.

A.66 'FSTSW': Almacena la palabra de estado de la coma flotante

```
FSTSW mem16                 ; 9B DD /0          [8086,FPU]
FSTSW AX                    ; 9B DF E0          [286,FPU]

FNSTSW mem16                ; DD /0             [8086,FPU]
FNSTSW AX                   ; DF E0             [286,FPU]
```

'FSTSW' almacena la palabra de estado de la FPU en 'AX' o en un área de memoria de 2 bytes.

'FNSTSW' hace lo mismo que 'FSTSW', pero sin esperar primero a que se limpien las excepciones pendientes de la coma flotante.

A.67 'FSUB', 'FSUBP', 'FSUBR', 'FSUBRP': Resta en coma flotante

```
FSUB mem32                  ; D8 /4             [8086,FPU]
FSUB mem64                  ; DC /4             [8086,FPU]

FSUB fpureg                 ; D8 E0+r           [8086,FPU]
FSUB ST0,fpureg             ; D8 E0+r           [8086,FPU]

FSUB TO fpureg              ; DC E8+r           [8086,FPU]
FSUB fpureg,ST0             ; DC E8+r           [8086,FPU]

FSUBR mem32                 ; D8 /5             [8086,FPU]
FSUBR mem64                 ; DC /5             [8086,FPU]

FSUBR fpureg                ; D8 E8+r           [8086,FPU]
FSUBR ST0,fpureg            ; D8 E8+r           [8086,FPU]

FSUBR TO fpureg             ; DC E0+r           [8086,FPU]
FSUBR fpureg,ST0            ; DC E0+r           [8086,FPU]

FSUBP fpureg                ; DE E8+r           [8086,FPU]
FSUBP fpureg,ST0            ; DE E8+r           [8086,FPU]

FSUBRP fpureg               ; DE E0+r           [8086,FPU]
FSUBRP fpureg,ST0           ; DE E0+r           [8086,FPU]
```

'FSUB' resta el operando dado de 'ST0' y almacena el resultado de nuevo en 'ST0', a menos que se de el calificador 'TO', en cuyo caso resta 'ST0'

del operando dado y almacena el resultado en el operando.

'FSUBR' hace lo mismo, pero hace la resta del otro modo: por eso si no se da 'TO', resta 'ST0' del operando dado y almacena el resultado en 'ST0', mientras que si se da 'TO' resta el operando de 'ST0' y almacena el resultado en el operando.

'FSUBP' opera como 'FSUB TO', pero saca la pila de registros una vez ha terminado. 'FSUBRP' opera como 'FSUBR TO', pero saca la pila de registros una vez ha terminado.

A.68 'FTST': Comprueba 'ST0' con cero

```
FTST                        ; D9 E4             [8086,FPU]
```

'FTST' compara 'ST0' con cero y activa de acuerdo a esto los flags de la FPU. 'ST0' es tratado como el lado izquierdo de la comparación, por eso se genera un resultado 'menor que' si 'ST0' es negativo.

A.69 'FUCOMxx': Comparación desordenada en coma flotante

```
FUCOM fpureg                ; DD E0+r           [386,FPU]
FUCOM ST0,fpureg            ; DD E0+r           [386,FPU]

FUCOMP fpureg               ; DD E8+r           [386,FPU]
FUCOMP ST0,fpureg           ; DD E8+r           [386,FPU]

FUCOMPP                     ; DA E9             [386,FPU]

FUCOMI fpureg               ; DB E8+r           [P6,FPU]
FUCOMI ST0,fpureg           ; DB E8+r           [P6,FPU]

FUCOMIP fpureg              ; DF E8+r           [P6,FPU]
FUCOMIP ST0,fpureg          ; DF E8+r           [P6,FPU]
```

'FUCOM' compara 'ST0' con el operando dado, y activa de forma acorde los flags de la FPU. 'ST0' es tratado como el lado izquierdo de la comparación, por eso el flag de acarreo se activa (para un resultado 'menor que') si 'ST0' es menor que el operando dado.

'FUCOMP' hace lo mismo que 'FUCOM', pero después saca la pila de registros. 'FUCOMPP' compara 'ST0' con 'ST1' y luego saca dos veces la pila de registros.

'FUCOMI' y 'FUCOMIP' funcionan como las formas correspondientes de 'FUCOM' y 'FUCOMP', pero escriben el resultado directamente en el registro de flags de la CPU en lugar de en la palabra de estado de la FPU, por eso pueden estar seguidas inmediatamente de instrucciones de salto condicional o de movimientos condicionales.

Las instrucciones 'FUCOM' difieren de las instrucciones 'FCOM' (sección A.35) sólo en el modo en que manejan las NaNs: 'FUCOM' las manejará silenciosamente y activará los flags del código de condición para un resultado 'desordenado', mientras que 'FCOM' generará una excepción.

A.70 'FXAM': Examina la clase del valor en 'ST0'

```
FXAM                        ; D9 E5             [8086,FPU]
```

'FXAM' activa los flags C3, C2 y C0 de la FPU dependiendo del tipo del

valor almacenado en 'ST0': 000 (respectivamente) para un formato no soportado, 001 para una NaN, 010 para un número finito normal, 011 para uno infinito, 100 para un cero, 101 para un registro vacío, y 110 para uno anormal. También activa el flag C1 para el signo del número.

A.71 'FXCH': Intercambio en coma flotante

```
FXCH                        ; D9 C9             [8086,FPU]
FXCH fpureg                 ; D9 C8+r           [8086,FPU]
FXCH fpureg,ST0             ; D9 C8+r           [8086,FPU]
FXCH ST0,fpureg             ; D9 C8+r           [8086,FPU]
```

'FXCH' intercambia 'ST0' con un registro dado de la FPU. La forma sin operando intercambia 'ST0' con 'ST1'.

A.72 'FXTRACT': Extrae el exponente y la mantisa

```
FXTRACT                     ; D9 F4             [8086,FPU]
```

'FXTRACT' separa el número en 'ST0' en su exponente y su mantisa, almacena el exponente de nuevo en 'ST0', y luego empuja la mantisa a la pila de registros (por eso la mantisa termina en 'ST0', y el exponente en 'ST1').

A.73 'FYL2X', 'FYL2XP1': Calcula Y veces Log2(X) o Log2(X+1)

```
FYL2X                       ; D9 F1             [8086,FPU]
FYL2XP1                     ; D9 F9             [8086,FPU]
```

'FYL2X' multiplica 'ST1' por el logaritmo de 'ST0' en base 2, almacena el resultado en 'ST1', y saca la pila de registros (por eso el resultado termina en 'ST0'). 'ST0' debe ser distinto de cero y positivo.

'FYL2XP1' funciona del mismo modo, pero sustituye el logaritmo de 'ST0' en base 2 por el de 'ST0' más uno. Esta vez, 'ST0' debe tener una magnitud no mayor que 1 menos la mitad de la raíz cuadrada de dos.

A.74 'HLT': Para el procesador

```
HLT                         ; F4                [8086]
```

'HLT' pone el procesador en el estado parado, en donde no realizará más operaciones hasta que vuelva a empezar por una interrupción o por un reset.

A.75 'IBTS': Inserta una cadena de bits

```
IBTS r/m16,reg16            ; o16 0F A7 /r      [386,UNDOC]
IBTS r/m32,reg32            ; o32 0F A7 /r      [386,UNDOC]
```

Parece que no hay una documentación clara disponible de esta instrucción: lo mejor que he podido encontrar dice 'Coge una cadena de bits desde el segundo operando y lo pone en el primer operando'. Sólo está presente en los primeros procesadores 386, y entra en conflicto con los códigos de operación de 'CMPXCHG486'. NASM la soporta sólo por completar. Su contrapartida es 'XBTS' (ver la sección A.167).

A.76 'IDIV': División de entero con signo

```
IDIV r/m8                   ; F6 /7             [8086]
```

```
IDIV r/m16                  ; o16 F7 /7           [8086]
IDIV r/m32                  ; o32 F7 /7           [386]
```

'IDIV' realiza la división entera con signo. El operando explícito que se proporcione es el divisor; el dividendo y el destino están implícitos, de la siguiente forma:

(*) Para 'IDIV r/m8', 'AX' es dividido por el operando dado; el cociente se almacena en 'AL' y el resto en 'AH'.

(*) Para 'IDIV r/m16', 'DX:AX' es dividido por el operando dado; el cociente se almacena en 'AX' y el resto en 'DX'.

(*) Para 'IDIV r/m32', 'EDX:EAX' es dividido por el operando dado; el cociente se almacena en 'EAX' y el resto en 'EDX'.

La división entera sin signo se realiza con la instrucción 'DIV': ver la sección A.25.

A.77 'IMUL': Multiplicación entera con signo

```
IMUL r/m8                   ; F6 /5               [8086]
IMUL r/m16                  ; o16 F7 /5           [8086]
IMUL r/m32                  ; o32 F7 /5           [386]

IMUL reg16,r/m16            ; o16 0F AF /r        [386]
IMUL reg32,r/m32            ; o32 0F AF /r        [386]

IMUL reg16,imm8             ; o16 6B /r ib        [286]
IMUL reg16,imm16            ; o16 69 /r iw        [286]
IMUL reg32,imm8             ; o32 6B /r ib        [386]
IMUL reg32,imm32            ; o32 69 /r id        [386]

IMUL reg16,r/m16,imm8       ; o16 6B /r ib        [286]
IMUL reg16,r/m16,imm16      ; o16 69 /r iw        [286]
IMUL reg32,r/m32,imm8       ; o32 6B /r ib        [386]
IMUL reg32,r/m32,imm32      ; o32 69 /r id        [386]
```

'IMUL' realiza la multiplicación entera con signo. Para la forma de un solo operando, el otro operando y el destino están implícitos, de la siguiente forma:

(*) Para 'IMUL r/m8', 'AL' es multiplicado por el operando dado; el producto se almacena en 'AX'.

(*) Para 'IMUL r/m16', 'AX' es multiplicado por el operando dado; el producto se almacena en 'DX:AX'.

(*) Para 'IMUL r/m32', 'EAX' es multiplicado por el operando dado; el producto se almacena en 'EDX:EAX'.

La forma de dos operandos multiplica los dos operandos y almacena el resultado en el operando (el primero) destino. La forma de tres operandos multiplica los dos últimos y almacena el resultado en el primer operando.

La forma de dos operandos en de hecho un atajo para la de tres operandos, como se puede ver examinando las descripciones del código de operación: en la forma de dos operandos, el código '/r' toma ambos registros y 'r/m' parte desde el mismo operando (el primero).

En las formas con operando inmediatos de 8-bit y otro operando fuente más largo, el operando inmediato se considera con signo, y se le hace una extensión de signo hasta la longitud del otro operando fuente. En estos casos, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

La multiplicación entera sin signo se realiza con la instrucción 'MUL': ver la sección A.107.

A.78 'IN': Entrada desde el puerto I/O

```
IN AL,imm8                  ; E4 ib             [8086]
IN AX,imm8                  ; o16 E5 ib         [8086]
IN EAX,imm8                 ; o32 E5 ib         [386]
IN AL,DX                    ; EC                [8086]
IN AX,DX                    ; o16 ED            [8086]
IN EAX,DX                   ; o32 ED            [386]
```

'IN' lee un byte, palabra o palabra doble desde el puerto I/O especificado, y lo almacena en el registro destino dado. El número de puerto podría especificarse como un valor inmediato si está entre 0 y 255, de otra forma debe estar almacenado en 'DX'. Ver también 'OUT' (sección A.111).

A.79 'INC': Incrementa un entero

```
INC reg16                   ; o16 40+r          [8086]
INC reg32                   ; o32 40+r          [386]
INC r/m8                    ; FE /0             [8086]
INC r/m16                   ; o16 FF /0         [8086]
INC r/m32                   ; o32 FF /0         [386]
```

'INC' suma 1 a su operando. _No_ afecta al flag de acarreo: para afectar al flag de acarreo, usa 'ADD something,1' (ver sección A.6). Ver también 'DEC' (sección A.24).

A.80 'INSB', 'INSW', 'INSD': Entra una cadena desde el puerto I/O

```
INSB                        ; 6C                [186]
INSW                        ; o16 6D            [186]
INSD                        ; o32 6D            [386]
```

'INSB' entra un byte desde el puerto I/O especificado en 'DX' y lo almacena en '[ES:DI]' o '[ES:EDI]'. Luego incrementa o decrementa (dependiendo del flag de dirección: incrementa si el flag está limpio, decrementa si está activo) 'DI' o 'EDI'.

Si el tamaño de la dirección es de 16 bits se usa el registro 'DI', y el 'EDI' si es de 32 bits. Si necesitas usar un tamaño de dirección distinto del actual 'BITS', puedes usar en prefijo explícito 'a16' o 'a32'.

Los prefijos de solapamiento de segmentos no tienen efecto en esta instrucción: el uso de 'ES' para la carga de '[DI]' o '[EDI]' no se puede superponer.

'INSW' e 'INSD' funcionan de la misma forma, pero meten una palabra o una palabra doble en lugar de un byte, e incrementan o decrementan el registro de direccionamiento en 2 ó 4 en lugar de en 1.

'El prefijo 'REP' podría usarse para repetir la instrucción 'CX' veces (o 'ECX' - nuevamente el tamaño de la dirección elige cuál).

Ver también 'OUTSB', 'OUTSW' y 'OUTSD' (sección A.112).

A.81 'INT': Interrupción software

```
INT imm8                    ; CD ib             [8086]
```

'INT' provoca una interrupción software por medio de un número de vector específico desde 0 a 255.

El código generado por la instrucción 'INT' siempre es de dos bytes de largo: aunque hay formas cortas para algunas instrucciones 'INT', NASM no las genera cuando ve el mnemónico 'INT'. Para generar instrucciones de ruptura de un solo byte, use en su lugar las instrucciones 'INT3' o 'INT1' (ver la sección A.82).

A.82 'INT3', 'INT1', 'ICEBP', 'INT01': Puntos de ruptura

```
INT1                        ; F1                [P6]
ICEBP                       ; F1                [P6]
INT01                       ; F1                [P6]

INT3                        ; CC                [8086]
```

'INT1' e 'INT3' son formas cortas de las instrucciones 'INT 1' e 'INT 3' (ver la sección A.81). Hacen una función similar a sus contrapartidas largas, pero ocupan menos espacio de código. Son usadas como puntos de ruptura por los depuradores.

'INT1', y su sinónimo alternativo 'INT01' e 'ICEBP', es una instrucción que se usa en emuladores de circuitos (ICEs). Está presente aunque no documentada, en algunos procesadores inferiores al 286, pero sólo está documentada para el Pentium Pro. 'INT3' es la instrucción que se usa normalmente como punto de ruptura en los depuradores.

'INT3' no es precisamente el equivalente de 'INT 3': la forma corta, ya que está diseñada para usarse como punto de ruptura, evita la comprobación normal de IOPL en el modo virtual 8086, y tampoco va a través de redirección de interrupción.

A.83 'INTO': Interrupción si hay desbordamiento

```
INTO                        ; CE                [8086]
```

'INTO' realiza una interrupción software 'INT 4' (ver sección A.81) si y sólo si se activa el flag de desbordamiento.

A.84 'INVD': Invalida los cachés internos

```
INVD                        ; 0F 08             [486]
```

'INVD' invalida y vacía los cachés internos del procesador, y hace que el procesador indique lo mismo a los cachés externos. No escribe primero los contenidos de los cachés de vuelta a memoria: cualquier dato modificado contenido en los cachés se perderá. Para escribir los datos primero, usa 'WBINVD' (sección A.164).

A.85 'INVLPG': Invalida la entrada a TLB

```
INVLPG mem                      ; 0F 01 /0            [486]
```

'INVLPG' invalida la entrada a la TLB (Translation Lookahead Buffer) asociada con la dirección de memoria suministrada.

A.86 'IRET', 'IRETW', 'IRETD': Volver desde una interrupción

```
IRET                            ; CF                  [8086]
IRETW                           ; o16 CF              [8086]
IRETD                           ; o32 CF              [386]
```

'IRET' vuelve desde una interrupción (hardware o software) sacando desde la pila el 'IP' (o 'EIP'), 'CS' y los flags y luego continuando la ejecución desde el nuevo 'CS:IP'.

'IRETW' saca desde la pila 'IP', 'CS' y los flags como dos bytes cada uno, liberando en total 6 bytes de la pila. 'IRETD' saca 'EIP' como 4 bytes, saca 4 bytes adicionales de los cuales los dos superiores se descartan y los dos inferiores van a 'CS', y saca los flags como 4 bytes, liberando 12 bytes de la pila.

'IRET' es un atajo de 'IRETW' o 'IRETD', dependiendo de la configuración de 'BITS' en ese momento.

A.87 'JCXZ', 'JECXZ': Salta si CX/ECX es cero

```
JCXZ imm                        ; o16 E3 rb           [8086]
JECXZ imm                       ; o32 E3 rb           [386]
```

'JCXZ' realiza un salto corto (con un rango máximo de 128 bytes) si y sólo si el contenido del registro 'CX' es 0. 'JECXZ' hace lo mismo, pero con 'ECX'.

A.88 'JMP': Salto

```
JMP imm                         ; E9 rw/rd            [8086]
JMP SHORT imm                   ; EB rb               [8086]
JMP imm:imm16                   ; o16 EA iw iw        [8086]
JMP imm:imm32                   ; o32 EA id iw        [386]
JMP FAR mem                     ; o16 FF /5           [8086]
JMP FAR mem                     ; o32 FF /5           [386]
JMP r/m16                       ; o16 FF /4           [8086]
JMP r/m32                       ; o32 FF /4           [386]
```

'JMP' salta a la dirección de dada. La dirección podría especificarse como un offset y un segmento absoluto, o como un salto relativo dentro del segmento actual.

'JMP SHORT imm' tiene un rango máximo de 128 bytes, ya que el desplazamiento se especifica con sólo 8 bits, pero ocupa menos espacio de código. NASM no elige por ti cuando generar 'JMP SHORT': debes codificar explícitamente 'SHORT' cada vez que quieras un salto corto.

Puedes elegir entre las dos formas de salto lejano inmediato ('JMP imm:imm') usando las palabras clave 'WORD' y 'DWORD': 'JMP WORD 0x1234:0x5678' o 'JMP DWORD 0x1234:0x5678abc'.

La forma 'JMP FAR mem' ejecuta un salto lejano cargando la dirección destino desde la memoria. La dirección cargada consiste en un offset de

16 o 32 bits (dependiendo del tamaño del operando), y 16 bits del segmento. El tamaño del operando podría ser superpuesto usando 'JMP WORD FAR mem' o 'JMP DWORD FAR mem'.

La forma 'JMP r/m' ejecuta un salto cercano (dentro del mismo segmento), cargando la dirección de destino desde memoria o desde un registro. La palabra clave 'NEAR' en estas formas se puede especificar, para mayor claridad, pero no es necesaria. Nuevamente, el tamaño del operando se puede superponer usando 'JMP WORD mem' o 'JMP DWORD mem'.

Por comodidad, NASM no te pide que saltes a un símbolo lejano mediante la codificación de un molesto 'JMP SEG rutina:rutina', sino que en su lugar permite el sinónimo más fácil 'JMP FAR rutina'.

Las formas 'CALL r/m' dadas arriba son llamadas cercanas; NASM aceptará la palabra clave 'NEAR' (por ejemplo 'CALL NEAR [address]'), incluso aunque no sea estrictamente necesaria.

A.89 'Jcc': Ramificación condicional

```
Jcc imm                     ; 70+cc rb          [8086]
Jcc NEAR imm                ; 0F 80+cc rw/rd    [386]
```

Las condiciones de saltos condicionales ejecutan un salto cercano (mismo segmento) si y sólo si se satisfacen sus condiciones. Por ejemplo, 'JNZ' salta sólo si no está activo el flag de cero.

Las formas normales de las instrucciones tienen sólo un rango de 128 bytes; la forma 'NEAR' es una extensión del 386 al conjunto de instrucciones, y puede abarcar el tamaño total del segmento. NASM no ignorará tu elección de la instrucción de salto: si quieres 'Jcc NEAR', tendrás que usar la palabra clave 'NEAR'.

La palabra clave 'SHORT' está permitida en la primera forma de la instrucción para más claridad, pero no es necesaria.

A.90 'LAHF': Cargar AH desde los flags

```
LAHF                        ; 9F                [8086]
```

'LAHF' establece el registro 'AH' de acuerdo con el contenido del byte bajo de la palabra de flags. Ver también 'SAHF' (sección A.145).

A.91 'LAR': Cargar derechos de acceso

```
LAR reg16,r/m16             ; o16 0F 02 /r      [286,PRIV]
LAR reg32,r/m32             ; o32 0F 02 /r      [286,PRIV]
```

'LAR' toma el selector de segmento especificado por su operando (el segundo) fuente, encuentra en la GDT o LDT el correspondiente descriptor de segmento, y carga el byte de derechos de acceso del descriptor en su operando (el primero) destino.

A.92 'LDS', 'LES', 'LFS', 'LGS', 'LSS': Carga un puntero lejano

```
LDS reg16,mem               ; o16 C5 /r         [8086]
LDS reg32,mem               ; o32 C5 /r         [8086]

LES reg16,mem               ; o16 C4 /r         [8086]
LES reg32,mem               ; o32 C4 /r         [8086]
```

```
LFS reg16,mem                ; o16 0F B4 /r         [386]
LFS reg32,mem                ; o32 0F B4 /r         [386]

LGS reg16,mem                ; o16 0F B5 /r         [386]
LGS reg32,mem                ; o32 0F B5 /r         [386]

LSS reg16,mem                ; o16 0F B2 /r         [386]
LSS reg32,mem                ; o32 0F B2 /r         [386]
```

Estas instrucciones cargan entero de una vez un puntero lejano (16 ó 32 bits de offset, más 16 bits de segmento) desde memoria. 'LDS', por ejemplo, carga 16 ó 32 bits desde la dirección de memoria dada en el registro dado (dependiendo del tamaño del registro), luego carga los _siguientes_ 16 bits desde memoria en 'DS'. 'LES'. 'LFS', 'LGS' y 'LSS' funcionan del mismo modo pero usan los otros registros de segmento.

A.93 'LEA': Carga la dirección efectiva

```
LEA reg16,mem                ; o16 8D /r            [8086]
LEA reg32,mem                ; o32 8D /r            [8086]
```

'LEA', a pesar de su sintaxis, no accede a memoria. Calcula la dirección efectiva especificada por su segundo operando como si fuese a cargar o almacenar datos desde ella, pero en su lugar almacena la dirección en el registro especificado por su primer operando. Esto puede usarse para realizar cálculos bastante complejos en una instrucción (por ejemplo, 'LEA EAX,[EBX+ECX*4+100]').

'LEA', a pesar de ser una instrucción puramente aritmética que no accede a memoria, sigue requiriendo los corchetes alrededor de su segundo operando, como si fuese una referencia a memoria.

A.94 'LEAVE': Destruye el marco de pila

```
LEAVE                        ; C9                   [186]
```

'LEAVE' destruye un marco de pila de la forma creada por la instrucción 'ENTER' (ver la sección A.27). Funcionalmente equivale a 'MOV ESP,EBP' seguido de un 'POP EBP' (o 'MOV SP,BP' seguido de un 'POP BP' en el modo de 16-bit).

A.95 'LGDT', 'LIDT', 'LLDT': Carga las tablas de descriptores

```
LGDT mem                     ; 0F 01 /2             [286,PRIV]
LIDT mem                     ; 0F 01 /3             [286,PRIV]
LLDT r/m16                   ; 0F 00 /2             [286,PRIV]
```

'LGDT' y 'LIDT' cogen como operando un área de memoria de 6 bytes: cargan una dirección lineal de 32-bits y un tamaño límite de 16-bit desde ese área (en el orden contrario) en el GDTR (Global Descriptor Table Register) o IDTR (Interrupt Descriptor Table Register). Estas son las únicas instrucciones que usan directamente direcciones _lineales_, en lugar de pares segmento/offset.

'LLDT' toma como operando un selector de segmento. El procesador mira este selector en la GDT y almacena las direcciones límite y base dadas allí en la LDTR (Local Descriptor Table Register).

Ver también 'SGDT', 'SIDT' y 'SLDT' (sección A.151).

A.96 'LMSW': Carga/almacena la palabra de estado de la máquina

```
LMSW r/m16                  ; 0F 01 /6          [286,PRIV]
```

'LMSW' carga los cuatro bits inferiores del operando fuente en los cuatro bits inferiores del registro de control 'CR0' (o la palabra de estado de la máquina, en procesadores 286). Ver también 'SMSW' (sección A.155).

A.97 'LOADALL', 'LOADALL286': Carga el estado del procesador

```
LOADALL                     ; 0F 07             [386,UNDOC]
LOADALL286                  ; 0F 05             [286,UNDOC]
```

Esta instrucción, en sus dos formas diferentes de código de operación, aparentemente está soportada en la mayoría de los 286, algunos 386 y posiblemente algunos 486. Los códigos de operación difieren entre el 286 y el 386.

La función de la instrucción es cargar toda la información relativa al estado del procesador desde un bloque de memoria: en el 286, este bloque está localizado implícitamente en la dirección absoluta '0x800', y en los 386 y 486 está en '[ES:EDI]'.

A.98 'LODSB', 'LODSW', 'LODSD': Carga desde una cadena

```
LODSB                       ; AC                [8086]
LODSW                       ; o16 AD            [8086]
LODSD                       ; o32 AD            [386]
```

'LODSB' carga un byte desde '[DS:SI]' o '[DS:ESI]' en 'AL'. Luego incrementa o decrementa (dependiendo del flag de dirección: incrementa si el flag está limpio, decrementa si está activo) 'SI' o 'ESI'.

Si el tamaño de la dirección es de 16 bits se usa el registro 'SI', y 'ESI' si es de 32 bits. Si necesitas usar un tamaño de dirección distinto del actual 'BITS', puedes usar un prefijo 'a16' o 'a32' explícito.

El registro de segmento usado para cargar desde '[SI]' o '[ESI]' se puede superponer usando un nombre de registro de segmento como prefijo (por ejemplo, 'es lodsb').

'LODSW' y 'LODSD' funcionan del mismo modo, pero cargan una palabra o una doble palabra en lugar de un byte, e incrementan o decrementan el registro de direccionamiento en 2 ó 4 en lugar de en 1.

A.99 'LOOP, 'LOOPE', 'LOOPZ', 'LOOPNE', 'LOOPNZ': Bucle con contador

```
LOOP imm                    ; E2 rb             [8086]
LOOP imm,CX                 ; a16 E2 rb         [8086]
LOOP imm,ECX                ; a32 E2 rb         [386]

LOOPE imm                   ; E1 rb             [8086]
LOOPE imm,CX                ; a16 E1 rb         [8086]
LOOPE imm,ECX               ; a32 E1 rb         [386]
LOOPZ imm                   ; E1 rb             [8086]
LOOPZ imm,CX                ; a16 E1 rb         [8086]
LOOPZ imm,ECX               ; a32 E1 rb         [386]

LOOPNE imm                  ; E0 rb             [8086]
```

```
LOOPNE imm,CX                ; a16 E0 rb           [8086]
LOOPNE imm,ECX               ; a32 E0 rb           [386]
LOOPNZ imm                   ; E0 rb               [8086]
LOOPNZ imm,CX                ; a16 E0 rb           [8086]
LOOPNZ imm,ECX               ; a32 E0 rb           [386]
```

'LOOP' decrementa su registro contador (ya sea 'CX' o 'ECX' - si no se especifica explícitamente, la configuración de 'BITS' dictamina cuál se usa) en uno, y si el contador no se hace cero como resultado de esta operación, salta a la etiqueta dada. El salto tiene un rango de 128 bytes.

'LOOPE' (o su sinónimo 'LOOPZ') añade la condición adicional de que sólo salta si el contador no es cero _y_ está activo el flag de cero. De forma similar, 'LOOPNE' (y 'LOOPNZ') salta sólo si el contador no es cero y el flag de cero está limpio.

A.100 'LSL': Carga el límite de segmento

```
LSL reg16,r/m16              ; o16 0F 03 /r        [286,PRIV]
LSL reg32,r/m32              ; o32 0F 03 /r        [286,PRIV]
```

A 'LSL' se le da un selector de segmento en su operando (el segundo) fuente; calcula el valor del límite de segmento cargando el campo del límite de segmento desde el descriptor de segmento asociado en la GDT o LDT. (Esto implica desplazar 12 bits hacia la izquierda si el límite de segmento es 'page-granular' y no 'byte-granular'; por eso terminas con un límite de byte en cualquier caso). El límite de segmento obtenido es más tarde cargado en el operando (el primero) destino.

A.101 'LTR': Carga el registro de tarea

```
LTR r/m16                    ; 0F 00 /3            [286,PRIV]
```

'LTR' busca la base del segmento y el límite en el descriptor especificado de la GDT o LDT mediante el selector de segmento dado como operando, y los carga en el registro de tarea.

A.102 'MOV': Mueve datos

```
MOV r/m8,reg8                ; 88 /r               [8086]
MOV r/m16,reg16              ; o16 89 /r           [8086]
MOV r/m32,reg32              ; o32 89 /r           [386]
MOV reg8,r/m8                ; 8A /r               [8086]
MOV reg16,r/m16              ; o16 8B /r           [8086]
MOV reg32,r/m32              ; o32 8B /r           [386]

MOV reg8,imm8                ; B0+r ib             [8086]
MOV reg16,imm16              ; o16 B8+r iw         [8086]
MOV reg32,imm32              ; o32 B8+r id         [386]
MOV r/m8,imm8                ; C6 /0 ib            [8086]
MOV r/m16,imm16              ; o16 C7 /0 iw        [8086]
MOV r/m32,imm32              ; o32 C7 /0 id        [386]

MOV AL,memoffs8              ; A0 ow/od            [8086]
MOV AX,memoffs16             ; o16 A1 ow/od        [8086]
MOV EAX,memoffs32            ; o32 A1 ow/od        [386]
MOV memoffs8,AL              ; A2 ow/od            [8086]
MOV memoffs16,AX             ; o16 A3 ow/od        [8086]
MOV memoffs32,EAX            ; o32 A3 ow/od        [386]
```

```
MOV r/m16,segreg            ; o16 8C /r          [8086]
MOV r/m32,segreg            ; o32 8C /r          [386]
MOV segreg,r/m16            ; o16 8E /r          [8086]
MOV segreg,r/m32            ; o32 8E /r          [386]

MOV reg32,CR0/2/3/4         ; 0F 20 /r           [386]
MOV reg32,DR0/1/2/3/6/7     ; 0F 21 /r           [386]
MOV reg32,TR3/4/5/6/7       ; 0F 24 /r           [386]
MOV CR0/2/3/4,reg32         ; 0F 22 /r           [386]
MOV DR0/1/2/3/6/7,reg32     ; 0F 23 /r           [386]
MOV TR3/4/5/6/7,reg32       ; 0F 26 /r           [386]
```

'MOV' copia el contenido de su operando (el segundo) fuente en su operando (el primero) destino.

En todas las formas de la instrucción 'MOV', el segundo operando es del mismo tamaño, excepto para mover entre un registro de segmento y un operando 'r/m32'. Estas instrucciones se tratan exactamente igual que las correspondientes equivalentes de 16-bit (por eso, por ejemplo, 'MOV DS,EAX' funciona idénticamente que 'MOV DS,AX' pero ahorra un prefijo en el modo de 32-bit), excepto cuando un registro de segmento se mueve a un destino de 32-bits, los dos bytes superiores del resultado están indefinidos.

'MOV' no debería usar 'CS' como destino.

'CR4' es un registro sólo soportado en el Pentium y superiores.

A.103 'MOVD': Mueve una palabra doble a/desde un registro MMX

```
MOVD mmxreg,r/m32           ; 0F 6E /r           [PENT,MMX]
MOVD r/m32,mmxreg           ; 0F 7E /r           [PENT,MMX]
```

'MOVD' copia 32 bits desde su operando (el segundo) fuente a su operando (el primero) destino. Cuando el destino es un registro MMX de 64-bits, los 32 bits superiores se ponen a cero.

A.104 'MOVQ': Mueve una palabra cuádruple a/desde un registro MMX

```
MOVQ mmxreg,r/m64           ; 0F 6F /r           [PENT,MMX]
MOVQ r/m64,mmxreg           ; 0F 7F /r           [PENT,MMX]
```

'MOVQ' copia 64 bits desde su operando (el segundo) fuente a su operando (el primero) destino.

A.105 'MOVSB', 'MOVSW', 'MOVSD': Mueve una cadena

```
MOVSB                       ; A4                 [8086]
MOVSW                       ; o16 A5             [8086]
MOVSD                       ; o32 A5             [386]
```

'MOVSB' copia el byte en '[ES:DI]' o '[ES:EDI]' a '[DS:SI]' o '[DS:SI]'. Luego incrementa o decrementa (dependiendo del flag de dirección: incrementa si el flag está limpio, decrementa si está activo) 'SI' y 'DI' (o 'ESI' y 'EDI').

Si el tamaño de la dirección es de 16 bits los registros usados son 'SI' y 'DI', y 'ESI' y 'EDI' si son de 32 bits. Si necesitas usar un tamaño de dirección distinto de la configuración actual de 'BITS', puedes usar

un prefijo 'a16' o 'a32' explícito.

El registro de segmento usado para cargar desde '[SI]' o '[ESI]' se puede superponer usando un nombre de registro de segmento como prefijo (por ejemplo, 'es movsb'). El uso de 'ES' para almacenar a '[DI]' o '[EDI]' no se puede superponer.

'MOVSW' y 'MOVSD' funcionan del mismo modo, pero copian en su lugar una palabra o una palabra doble en lugar de un byte, e incrementan o decrementan el registro de direccionamiento en 2 ó 4 en lugar de en 1.

'El prefijo 'REP' se podría usar para repetir la instrucción 'CX' veces (o 'ECX' - nuevamente, el tamaño de la dirección determina cúal).

A.106 'MOVSX', 'MOVZX': Mueve datos con extensión de signo o de cero

```
MOVSX reg16,r/m8              ; o16 0F BE /r         [386]
MOVSX reg32,r/m8              ; o32 0F BE /r         [386]
MOVSX reg32,r/m16             ; o32 0F BF /r         [386]

MOVZX reg16,r/m8              ; o16 0F B6 /r         [386]
MOVZX reg32,r/m8              ; o32 0F B6 /r         [386]
MOVZX reg32,r/m16             ; o32 0F B7 /r         [386]
```

'MOVSX' hace extensión de signo a su operando (el segundo) fuente hasta la longitud de su operando (el primero) destino, y copia el resultado en el operando destino. 'MOVZX' hace lo mismo, pero hace extensión de cero en lugar de extensión de signo.

A.107 'MUL': Multiplicación entera sin signo

```
MUL r/m8                      ; F6 /4                [8086]
MUL r/m16                     ; o16 F7 /4            [8086]
MUL r/m32                     ; o32 F7 /4            [386]
```

'MUL' realiza la multiplicación entera sin signo. El otro operando de la multiplicación, y el operando destino, están implícitos, de la siguiente forma:

(*) Para 'MUL r/m8', 'AL' se multiplica por el operando dado; el producto se almacena en 'AX'.

(*) Para 'MUL r/m16', 'AX' se multiplica por el operando dado; el producto se almacena en 'DX:AX'.

(*) Para 'MUL r/m32', 'EAX' se multiplica por el operando dado; el producto se almacena en 'EDX:EAX'.

La multiplicación entera con signo se realiza usando la instrucción 'IMUL': ver la sección A.77.

A.108 'NEG', 'NOT': Complemento a dos y a uno

```
NEG r/m8                      ; F6 /3                [8086]
NEG r/m16                     ; o16 F7 /3            [8086]
NEG r/m32                     ; o32 F7 /3            [386]

NOT r/m8                      ; F6 /2                [8086]
NOT r/m16                     ; o16 F7 /2            [8086]
NOT r/m32                     ; o32 F7 /2            [386]
```

'NEG' sustituye el contenido de su operando por la negación en complemento a dos (invierte todos los bits y luego suma uno) del valor original. 'NOT', de forma similar, realiza el complemento a uno (invierte todos los bits).

A.109 'NOP' No operación

```
NOP                         ; 90                [8086]
```

'NOP' realiza la no operación. Su código de operación es el mismo que el generado por 'XCHG AX,AX' o 'XCHG EAX,EAX' (dependiendo del modo del procesador; ver la sección A.168).

A.110 'OR': OR a nivel de bit

```
OR r/m8,reg8                ; 08 /r             [8086]
OR r/m16,reg16              ; o16 09 /r         [8086]
OR r/m32,reg32              ; o32 09 /r         [386]

OR reg8,r/m8                ; 0A /r             [8086]
OR reg16,r/m16              ; o16 0B /r         [8086]
OR reg32,r/m32              ; o32 0B /r         [386]

OR r/m8,imm8                ; 80 /1 ib          [8086]
OR r/m16,imm16              ; o16 81 /1 iw      [8086]
OR r/m32,imm32              ; o32 81 /1 id      [386]

OR r/m16,imm8               ; o16 83 /1 ib      [8086]
OR r/m32,imm8               ; o32 83 /1 ib      [386]

OR AL,imm8                  ; 0C ib             [8086]
OR AX,imm16                 ; o16 0D iw         [8086]
OR EAX,imm32                ; o32 0D id         [386]
```

'OR' realiza una operación OR a nivel de bit entre sus dos operandos (esto es, cada bit del resultado es 1 si y sólo si al menos uno de los bits correspondientes de las dos entradas es 1), y almacena el resultado en el operando (el primero) destino.

En las formas con inmediatos de 8-bits como segundo operando y un primer operando más largo, se considera el segundo operando con signo, y se le hace extensión de signo hasta la longitud del primer operando. En estos caso, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

La instrucción MMX 'POR' (ver sección A.129) realiza la misma operación en los registros MMX de 64-bits.

A.111 'OUT': Manda datos al puerto I/O

```
OUT imm8,AL                 ; E6 ib             [8086]
OUT imm8,AX                 ; o16 E7 ib         [8086]
OUT imm8,EAX                ; o32 E7 ib         [386]
OUT DX,AL                   ; EE                [8086]
OUT DX,AX                   ; o16 EF            [8086]
OUT DX,EAX                  ; o32 EF            [386]
```

'OUT' escribe el contenido del registro fuente dado en el puerto I/O especificado. El número de puerto se podría especificar como un valor

inmediato si está entre 0 y 255, y de otro modo debe estar almacenado en 'DX'. Ver también 'IN' (sección A.78).

A.112 'OUTSB', 'OUTSW', 'OUTSD': Manda una cadena al puerto I/O

```
OUTSB                       ; 6E                [186]

OUTSW                       ; o16 6F            [186]

OUTSD                       ; o32 6F            [386]
```

'OUTSB' carga un byte desde '[DS:SI]' o '[DS:ESI]' y lo escribe al puerto I/O especificado en 'DX'. Luego incrementa o decrementa (dependiendo del flag de dirección: incrementa si el flag está limpio, decrementa si está activo) 'SI' o 'ESI'.

Si el tamaño de la dirección es de 16 bits se usa el registro 'SI', y 'ESI' si es de 32 bits. Si necesitas usar un tamaño de dirección diferente de configurado por 'BITS', puedes usar un prefijo 'a16' o 'a32' explícito.

El registro de segmento usado para cargar desde '[SI]' o '[ESI]' puede superponerse usando un nombre de registro de segmento como prefijo (por ejemplo, 'es outsb').

'OUTSW' y 'OUTSD' funcionan de forma similar, pero sacan una palabra o una palabra doble en lugar de un byte, e incrementan o decrementan el registro de direccionamiento en 2 ó 4 en lugar de en 1.

Se podría usar el prefijo 'REP' para repetir la instrucción 'CX' veces (o 'ECX' - nuevamente, el tamaño de la dirección elige cuál).

A.113 'PACKSSDW', 'PACKSSWB', 'PACKUSWB': Empaqueta datos

```
PACKSSDW mmxreg,r/m64       ; 0F 6B /r          [PENT,MMX]
PACKSSWB mmxreg,r/m64       ; 0F 63 /r          [PENT,MMX]
PACKUSWB mmxreg,r/m64       ; 0F 67 /r          [PENT,MMX]
```

Todas estas instrucciones comienzan formando una palabra de 128-bit colocando el operando (el segundo) fuente a la izquierda del operando (el primero) destino. 'PACKSSDW' luego separa esta palabra de 128-bits en cuatro palabras dobles, convierte cada una en una palabra, y las carga juntas en el registro destino; 'PACKSSWB' y 'PACKUSWB' separan la palabra de 128-bits en ocho palabras, convirtiendo cada una en un byte, y cargándolas juntas en el registro destino.

'PACKSSWD' y 'PACKSSWB' realizan una saturación con signo cuando se reduce la longitud de números: si el número es demasiado grande para caber en un espacio reducido, lo sustituyen con el mayor número con signo ('7FFFh' o '7Fh') que _cabrá_, y si el número es demasiado pequeño entonces sustituyen el número con signo más pequeño ('8000h' o '80h') que cabrá. 'PACKUSWB' realiza la saturación sin signo: trata su entrada como si fuese sin signo, y la sustituye por el mayor número sin signo que cabrá.

A.114 'PADDxx': Suma de MMX empaquetado

```
PADDB mmxreg,r/m64          ; 0F FC /r          [PENT,MMX]
PADDW mmxreg,r/m64          ; 0F FD /r          [PENT,MMX]
PADDD mmxreg,r/m64          ; 0F FE /r          [PENT,MMX]
```

```
PADDSB mmxreg,r/m64          ; 0F EC /r            [PENT,MMX]
PADDSW mmxreg,r/m64          ; 0F ED /r            [PENT,MMX]

PADDUSB mmxreg,r/m64         ; 0F DC /r            [PENT,MMX]
PADDUSW mmxreg,r/m64         ; 0F DD /r            [PENT,MMX]
```

Todas las 'PADDxx' realizan la suma empaquetada entre sus dos operandos de 64-bit, almacenando el resultado en el operando (el primero) destino. Las formas 'PADDxB' tratan los operandos de 64-bit como vectores de ocho bytes, y suman cada byte individualmente; 'PADDxW' tratan los operandos como vectores de cuatro palabras; y 'PADDD' trata sus operandos como un vector de dos palabras dobles.

'PADDSB' y 'PADDSW' realizan la saturación de signo en la suma de cada par de bytes o palabras: si el resultado de una suma es demasiado grande o demasiado pequeño para caber en un resultado de byte o palabra con signo, se recorta (saturado) al valor más grande o más pequeño en el cual _cabrá_. 'PADDUSB' y 'PADDUSW' de forma similar realizan la saturación sin signo, recortando a '0FFh' o '0FFFFh' si el resultado es mayor que ése.

A.115 'PADDSIW': Suma MMX empaquetada a un destino implícito

```
PADDSIW mmxreg,r/m64         ; 0F 51 /r            [CYRIX,MMX]
```

'PADDSIW', específico de las extensiones Cyrix al conjunto de instrucciones MMX, realiza la misma función que 'PADDSW', excepto que el resultado no se coloca en el registro especificado por el primero operando, sino en su lugar, en el registro cuyo número difiera del primer operando sólo en su último bit. Por eso 'PADDSIW MM0,MM2' pondría el resultado en 'MM1', pero 'PADDSIW MM1,MM2' lo pondría en 'MM0'.

A.116 'PAND', 'PANDN': AND y AND-NOT del MMX a nivel de bit

```
PAND mmxreg,r/m64            ; 0F DB /r            [PENT,MMX]
PANDN mmxreg,r/m64           ; 0F DF /r            [PENT,MMX]
```

'PAND' realiza una operación AND a nivel de bit entre sus dos operandos (esto es, cada bit del resultado es 1 si y sólo si los correspondientes bits de las dos entradas son ambos 1), y almacena el resultado en el operando (el primero) destino.

'PANDN' realiza la misma operación, pero realiza primero un complemento a uno del operando (el primero) destino.

A.117 'PAVEB': Media empaquetada del MMX

```
PAVEB mmxreg,r/m64           ; 0F 50 /r            [CYRIX,MMX]
```

'PAVEB', específico de las extensiones Cyrix del MMX, trata su dos operandos como vectores de ocho bytes sin signo, y calcula la media de los bytes correspondientes en los operandos. El vector resultante de ocho medias se almacena en el primer operando.

A.118 'PCMPxx': Comparación empaquetada del MMX

```
PCMPEQB mmxreg,r/m64         ; 0F 74 /r            [PENT,MMX]
PCMPEQW mmxreg,r/m64         ; 0F 75 /r            [PENT,MMX]
PCMPEQD mmxreg,r/m64         ; 0F 76 /r            [PENT,MMX]
```

```
PCMPGTB mmxreg,r/m64          ; 0F 64 /r            [PENT,MMX]
PCMPGTW mmxreg,r/m64          ; 0F 65 /r            [PENT,MMX]
PCMPGTD mmxreg,r/m64          ; 0F 66 /r            [PENT,MMX]
```

Las instrucciones 'PCMPxx' tratan a sus operandos como vectores de bytes, palabras, o doble palabras; se comparan los elementos correspondientes de la fuente y el destino, y los elementos correspondientes del operando (el primero) destino se ponen todos a cero o a unos dependiendo del resultado de la comparación.

'PCMPxxB' trata los operandos como vectores de ocho bytes, 'PCMPxxW' los trata como un vector de cuatro palabras, y 'PCMPxxD' como dos dobles palabras.

'PCMPEQx' pone a uno el elemento correspondiente del operando destino si los dos elementos comparados son iguales; 'PCMPGTx' pone a uno el destino si el elemento del primer (el destino) operando es mayor (tratado como un entero con signo) que el segundo operando (el fuente).

A.119 'PDISTIB': Distancia y acumulación MMX empaquetada con registro implícito

```
PDISTIB mmxreg,mem64          ; 0F 54 /r            [CYRIX,MMX]
```

''PDISTIB', específico de la extensión MMX de Cyrix, trata su dos operandos de entrada como vectores de ocho bytes sin signo. Por cada posición de byte, encuentra la diferencia absoluta entre los bytes en esa posición en los dos operandos de entrada, y suma el valor al bytes en la misma posición en el registro de salida implícito. La suma está saturada a un byte sin signo del mismo modo que 'PADDUSB'.

El registro de salida implícito se encuentra del mismo modo que en 'PADDSIW' (sección A.115).

Nótese que 'PDISTIB' no puede tomar un registro como su segundo operando fuente.

A.120 'PMACHRIW': Multiplicación y acumulación MMX empaquetada con redondeo

```
PMACHRIW mmxreg,mem64         ; 0F 5E /r            [CYRIX,MMX]
```

'PMACHRIW' actúa casi idénticamente a 'PMULHRIW' (sección A.123), pero en lugar de _almacenar_ su resultado en un registro de destino implícito, _suma_ el resultado, como cuatro palabras empaquetadas, al registro destino implícito. No se hace saturación: la suma puede excederse por uno de los límites y aparecer por el otro ('dar la vuelta al marcador').

Nótese que 'PMACHRIW' no puede tomar un registro como su segundo operando fuente.

A.121 'PMADDWD': Multiplicación y suma MMX empaquetada

```
PMADDWD mmxreg,r/m64          ; 0F F5 /r            [PENT,MMX]
```

'PMADDWD' trata sus dos operandos de entrada como vectores de cuatro palabras con signo. Multiplica el elemento correspondiente de los dos operandos, dando como resultado cuatro dobles palabras con signo. Las dos superiores se suman y colocan en los 32 bits superiores del operando (el primero) destino; las dos inferiores se suman y colocan en los 32 bits inferiores.

A.122 'PMAGW': Magnitud MMX empaquetada

```
PMAGW mmxreg,r/m64          ; 0F 52 /r          [CYRIX,MMX]
```

'PMAGW', específico de la extensión MMX de Cyrix, trata sus dos operandos como vectores de cuatro palabras con signo. Compara los valores absolutos de las palabras en posiciones correspondientes, y establece cada palabra del operando (el primero) destino a cualquiera de las dos palabras en esa posición que tenga mayor valor absoluto.

A.123 'PMULHRW','PMULHRIW': Multiplicación MMX empaquetada elevada con redondeo

```
PMULHRW mmxreg,r/m64        ; 0F 59 /r          [CYRIX,MMX]
PMULHRIW mmxreg,r/m64       ; 0F 5D /r          [CYRIX,MMX]
```

Estas instrucciones, específicas de las extensiones MMX de Cyrix, tratan a sus operandos como vectores de cuatro palabras con signo. Las palabras en posiciones correspondientes se multiplican, para dar un valor de 32-bit en el cual los bits 30 y 31 se garantiza que son iguales. Los bits 30 al 15 de este valor (máscara de bits '0x7FFF8000') se almacenan en la posición correspondiente del operando destino, después de redondear primero el bit bajo (equivalente a sumar '0x4000' antes de extraer los bits 30 al 15).

Para 'PMULHRW', el operando destino es el primer operando; para 'PMULHRIW' el operando destino está implícito en el primer operando en la manera de 'PADDSIW' (sección A.115).

A.124 'PMULHW', 'PMULLW': Multiplicación MMX empaquetada

```
PMULHW mmxreg,r/m64         ; 0F E5 /r          [PENT,MMX]
PMULLW mmxreg,r/m64         ; 0F D5 /r          [PENT,MMX]
```

'PMULxW' trata sus dos operandos de entrada como vectores de cuatro palabras con signo. Multiplica los elementos correspondientes de los dos operandos, dando como resultado cuatro dobles palabras con signo.

'PMULHW' luego almacena los 16 bits superiores de cada palabras doble en operando (el primero) destino; 'PMULLW' almacena los 16 bits inferiores de cada palabra doble en el operando destino.

A.125 'PMVccZB': Movimiento condicional MMX empaquetado

```
PMVZB mmxreg,mem64          ; 0F 58 /r          [CYRIX,MMX]
PMVNZB mmxreg,mem64         ; 0F 5A /r          [CYRIX,MMX]
PMVLZB mmxreg,mem64         ; 0F 5B /r          [CYRIX,MMX]
PMVGEZB mmxreg,mem64        ; 0F 5C /r          [CYRIX,MMX]
```

Estas instrucciones, específicas de las extensiones MMX de Cyrix, realizan movimiento condicionales paralelos. Los dos operandos de entrada son tratados como vectores de ocho bytes. Cada byte del operando (el primero) destino es o escrito desde el byte correspondiente del operando (el segundo) fuente, o dejado solo, dependiendo del valor del byte en el operando _implícito_ (especificado del mismo modo que en 'PADDSIW', en la sección A.115).

'PMVZB' realiza cada movimiento si es cero el byte correspondiente en el operando implícito. 'PMVNZB' mueve si el byte no es cero. 'PMVLZB' mueve si el byte es menor que cero. y 'PMVGEZB' mueve si el byte es mayor o

igual que cero.

Nótese que estas instrucciones no pueden tomar un registro como su segundo operando fuente.

A.126 'POP': Saca datos desde la pila

```
POP reg16                   ; o16 58+r          [8086]
POP reg32                   ; o32 58+r          [386]

POP r/m16                   ; o16 8F /0         [8086]
POP r/m32                   ; o32 8F /0         [386]

POP CS                      ; 0F                [8086,UNDOC]
POP DS                      ; 1F                [8086]
POP ES                      ; 07                [8086]
POP SS                      ; 17                [8086]
POP FS                      ; 0F A1             [386]
POP GS                      ; 0F A9             [386]
```

'POP' carga un valor desde la pila (desde '[SS:SP]' o '[SS:ESP]') y luego incrementa el puntero de pila.

El atributo de tamaño de dirección de la instrucción determina si se usa como puntero de pila 'SP' o 'ESP': para superponer deliberadamente el dado por defecto por 'BITS', puedes usar un prefijo 'a16' o 'a32'.

El atributo de tamaño del operando de la instrucción determina si el puntero de pila se incrementa en 2 ó en 4: esto quiere decir que sacar en el modo 'BITS 32' sacará 4 bytes de la pila y descartará los dos superiores. Si necesitas superponer esto, puedes usar el prefijo 'o16' o 'o32'.

La lista de códigos de operación de arriba da dos formas de las instrucciones de sacar registros de propósito general: por ejemplo, 'POP BX' tiene las dos formas '5B' y '8F C3'. NASM siempre generará la forma más corta cuando se le dé 'POP BX'. NDISASM desensamblará ambas.

'POP CS' es una instrucción no documentada, y no está soportada en ningún procesador por encima del 8086 (ya que usan '0Fh' como un prefijo de código de operación para las extensiones del conjunto de instrucciones). Sin embargo, al menos algunos procesadores 8086 la soportan, y por eso NASM la genera para completar.

A.127 'POPAx': Saca todos los registros de propósito general

```
POPA                        ; 61                [186]
POPAW                       ; o16 61            [186]
POPAD                       ; o32 61            [386]
```

'POPAW' saca una palabra desde la pila en cada uno de los, sucesivamente, 'DI', 'SI', 'BP', nada (descarta una palabra desde la pila que corresponde con 'SP'), 'BX', 'DX', 'CX' y 'AX'. Se desea invertir la operación de 'PUSHAW' (ver la sección A.135), pero ignora el valor de 'SP' que fue empujado a la pila por 'PUSHAW'.

'POPAD' saca el doble de datos, y coloca los resultado en 'EDI', 'ESI', 'EBP', nada (corresponde a 'ESP'), 'EBX', 'EDX', 'ECX' y 'EAX'. Invierte la operación de 'PUSHAD'.

'POPA' es un mnemónico para 'POPAW' o 'POAPD', dependiendo de la configuración actual de 'BITS'.

Nótese que los registros se sacan en orden inverso al de sus valores numéricos en los códigos de operación (ver la sección A.2.1).

A.128 'POPFx': Saca el registro de flags

```
POPF                        ; 9D                [186]
POPFW                       ; o16 9D            [186]
POPFD                       ; o32 9D            [386]
```

'POPFW' saca una palabra desde la pila y la almacena en los 16 bits inferiores del registro de flags (en el registro de flags completo, en los procesadores inferiores al 386). 'POPFD' saca una palabra doble y la almacena entera en el registro de flags.

'POPF' es un mnemónico para 'POPFW' o 'POPFD', dependiendo de la configuración actual de 'BITS'.

Ver también 'PUSHF' (sección A.136).

A.129 'POR': OR a nivel de bit del MMX

```
POR mmxreg,r/m64            ; 0F EB /r          [PENT,MMX]
```

'POR' realiza un OR a nivel de bit entre sus dos operandos (esto es, cada bit del resultado es 1 si y sólo si lo es al menos uno de los bits correspondientes de sus dos entradas), y almacena el resultado en el operando (el primero) destino.

A.130 'PSLLx', 'PSRLx', 'PSRAx': Desplazamiento de bits del MMX

```
PSLLW mmxreg,r/m64          ; 0F F1 /r          [PENT,MMX]
PSLLW mmxreg,imm8           ; 0F 71 /6 ib       [PENT,MMX]

PSLLD mmxreg,r/m64          ; 0F F2 /r          [PENT,MMX]
PSLLD mmxreg,imm8           ; 0F 72 /6 ib       [PENT,MMX]

PSLLQ mmxreg,r/m64          ; 0F F3 /r          [PENT,MMX]
PSLLQ mmxreg,imm8           ; 0F 73 /6 ib       [PENT,MMX]

PSRAW mmxreg,r/m64          ; 0F E1 /r          [PENT,MMX]
PSRAW mmxreg,imm8           ; 0F 71 /4 ib       [PENT,MMX]

PSRAD mmxreg,r/m64          ; 0F E2 /r          [PENT,MMX]
PSRAD mmxreg,imm8           ; 0F 72 /4 ib       [PENT,MMX]

PSRLW mmxreg,r/m64          ; 0F D1 /r          [PENT,MMX]
PSRLW mmxreg,imm8           ; 0F 71 /2 ib       [PENT,MMX]

PSRLD mmxreg,r/m64          ; 0F D2 /r          [PENT,MMX]
PSRLD mmxreg,imm8           ; 0F 72 /2 ib       [PENT,MMX]

PSRLQ mmxreg,r/m64          ; 0F D3 /r          [PENT,MMX]
PSRLQ mmxreg,imm8           ; 0F 73 /2 ib       [PENT,MMX]
```

'PSxxQ' realiza un sencillo desplazamiento de bit en los registros MMX de 64-bits: el operando (el primero) destino es desplazado a la izquierda o derecha un número de bits dado en el operando (el segundo) fuente, y los

bits vacíos se rellenan con ceros (para el desplazamiento lógico) o copia el bit de signo original (para el desplazamiento aritmético a la derecha).

'PSxxW' y 'PSxxD' realiza desplazamientos empaquetados de bits: el operando destino es tratado como un vector de cuatro palabras o de dos palabras dobles, y cada elemento se desplaza individualmente, por eso los bits desplazados fuera de un elemento no interfieren con los bits vacíos que quedan en el siguiente.

'PSLLx' y 'PSRLx' realiza desplazamientos lógicos: los bits vacíos en uno de los extremos del número desplazado se rellenan con ceros. 'PSRAx' realiza un desplazamiento aritmético hacia la derecha: los bits vacíos en la parte superior de número desplazado se rellenan con copias del bit de signo original (el bit superior).

A.131 'PSUBxx': Resta MMX empaquetada

```
PSUBB mmxreg,r/m64          ; 0F F8 /r          [PENT,MMX]
PSUBW mmxreg,r/m64          ; 0F F9 /r          [PENT,MMX]
PSUBD mmxreg,r/m64          ; 0F FA /r          [PENT,MMX]

PSUBSB mmxreg,r/m64         ; 0F E8 /r          [PENT,MMX]
PSUBSW mmxreg,r/m64         ; 0F E9 /r          [PENT,MMX]

PSUBUSB mmxreg,r/m64        ; 0F D8 /r          [PENT,MMX]
PSUBUSW mmxreg,r/m64        ; 0F D9 /r          [PENT,MMX]
```

'PSUBxx' realizan la resta empaquetada entre su dos operandos de 64-bits, almacenando el resultado en el operando (el primero) destino. Las formas 'PSUBxB' tratan los operandos de 64-bits como vectores de ocho bytes, y restan cada byte individualmente; 'PSUBxW' trata los operandos como vectores de cuatro palabras; y 'PSUBD' trata sus operandos como vectores de dos palabras dobles.

En todos los casos, los elementos del operando de la derecha se restan de los correspondientes elementos del operando de la izquierda, no de la otra forma.

'PSUBSB' y 'PSUBSW' realiza la saturación con signo en la suma de cada par de bytes o palabras: si el resultado de una resta es demasiado grande o demasiado pequeño como para caber en el byte o palabra resultado, es recortado (saturado) al más grande o más pequeño que _cabrá_. 'PSUBUSB' y 'PSUBUSW' realiza de forma similar la saturación sin signo, recortando a '0FFh' o '0FFFFh' si el resultado es mayor que ése.

A.132 'PSUBSIW': Resta MMX empaquetada con saturación a un destino implícito

```
PSUBSIW mmxreg,r/m64        ; 0F 55 /r          [CYRIX,MMX]
```

'PSUBSIW', específico de las extensiones MMX de Cyrix, realiza la misma función que 'PSUBSw', excepto que el resultado no se coloca en el registro especificado por el primer operando, sino que en su lugar se hace en el registro destino implícito, especificado como en 'PADDSIW' (sección A.115).

A.133 'PUNPCKxxx': Desempaquetar datos

```
PUNPCKHBW mmxreg,r/m64      ; 0F 68 /r          [PENT,MMX]
PUNPCKHWD mmxreg,r/m64      ; 0F 69 /r          [PENT,MMX]
```

```
PUNPCKHDQ mmxreg,r/m64        ; 0F 6A /r            [PENT,MMX]

PUNPCKLBW mmxreg,r/m64        ; 0F 60 /r            [PENT,MMX]
PUNPCKLWD mmxreg,r/m64        ; 0F 61 /r            [PENT,MMX]
PUNPCKLDQ mmxreg,r/m64        ; 0F 62 /r            [PENT,MMX]
```

'PUNPCKxx' tratan a sus operandos como vectores, y producen un nuevo vector generado entrelazando elementos desde sus dos entradas. Las instrucciones 'PUNPCKHxx' comienzan tirando la mitad inferior de cada operando de entrada, y las instrucciones 'PUNPCKLxx' tiran la mitad superior.

Los elementos restantes, totalizan 64 bits, son entrelazados en el destino, alternando elementos desde el segundo operando (el fuente) y el primer operando (el destino): por eso el elemento más a la izquierda de resultado siempre viene del segundo operando, y el más a la derecha del destino.

'PUNPCKxBW' funciona con un byte cada vez, 'PUNPCKxWD' una palabra cada vez, y 'PUNPCKxDQ' un palabra doble cada vez.

Por eso, por ejemplo, si el primer operando contiene '0x7A6A5A4A3A2A1A0A' y el segundo contiene '0x7B6B5B4B3B2B1B0B', entonces:

(*) 'PUNPCKHBW' devolvería '0x7B7A6B6A5B5A4B4A'.

(*) 'PUNPCKHWD' devolvería '0x7B6B7A6A5B4B5A4A'.

(*) 'PUNPCKHDQ' devolvería '0x7B6B5B4B7A6A5A4A'.

(*) 'PUNPCKLBW' devolvería '0x3B3A2B2A1B1A0B0A'.

(*) 'PUNPCKLWD' devolvería '0x3B2B3A2A1B0B1A0A'.

(*) 'PUNPCKLDQ' devolvería '0x3B2B1B0B3A2A1A0A'.

A.134 'PUSH': Empuja datos a la pila

```
PUSH reg16                    ; o16 50+r            [8086]
PUSH reg32                    ; o32 50+r            [386]

PUSH r/m16                    ; o16 FF /6           [8086]
PUSH r/m32                    ; o32 FF /6           [386]

PUSH CS                       ; 0E                  [8086]
PUSH DS                       ; 1E                  [8086]
PUSH ES                       ; 06                  [8086]
PUSH SS                       ; 16                  [8086]
PUSH FS                       ; 0F A0               [386]
PUSH GS                       ; 0F A8               [386]

PUSH imm8                     ; 6A ib               [286]
PUSH imm16                    ; o16 68 iw           [286]
PUSH imm32                    ; o32 68 id           [386]
```

'PUSH' decrementa el puntero de pila ('SP' o 'ESP') en 2 ó en 4, y luego almacena el valor dado en '[SS:SP]' o '[SS:ESP]'.

El atributo de tamaño de dirección de la instrucción determina si se usa como puntero de pila 'SP' o 'ESP': para superponer deliberadamente el

valor por defecto dado por la configuración de 'BITS', puedes usar un prefijo 'a16' o 'a32'.

El atributo de tamaño de operando de la instrucción determina si el puntero de pila se decrementa en 2 ó en 4: esto quiere decir que el empujar en el modo 'BITS 32' empujará 4 bytes a la pila, de los cuales los dos superiores estarán indefinidos. Si necesitas superponer esto, puedes usar un prefijo 'o16' o 'o32'.

La lista de códigos de operación de arriba da dos formas de las instrucciones de empujar registros de propósito general: por ejemplo, 'PUSH BX' tiene las dos formas '53' y 'FF F3'. NASM siempre generará la forma más corta cuando se le dé 'PUSH BX'. NDISASM desensamblará ambas.

A diferencia de la indocumentada y raramente soportada 'POP CS', 'PUSH CS' es una instrucción perfectamente válida y correcta, soportada por todos los procesadores.

La instrucción 'PUSH SP' se podría usar para distinguir un 8086 de posteriores procesadores: en un 8086, el valor de 'SP' almacenado es el valor que tiene _después_ de la instrucción push, mientras que en procesadores posteriores es el valor de _antes_ de la instrucción push.

A.135 'PUSHAx': Empujar todos los registros de propósito general

```
PUSHA                       ; 60                [186]
PUSHAD                      ; o32 60            [386]
PUSHAW                      ; o16 60            [186]
```

'PUSHAW' empuja, sucesivamente, 'AX', 'CX', 'DX', 'BX', 'SP', 'BP', 'SI' y 'DI' a la pila, decrementando el puntero de pila en un total de 16.

'PUSHAD' empuja, sucesivamente, 'EAX', 'ECX', 'EDX', 'EBX', 'ESP', 'EBP', 'ESI' y 'EDI' a la pila, decrementando el puntero de pila en un total de 32.

En ambos casos, el valor de 'SP' o 'ESP' empujado es el valor _original_, tal y como estaba antes de que se ejecutase la instrucción.

'PUSHA' es un mnemónico para 'PUSHAW' o 'PUSHAD', dependiendo de la configuración actual de 'BITS'.

Nótese que se empujan los registros en orden de sus valores numéricos en los códigos de operación (ver la sección A.2.1).

Ver también 'POPA' (sección A.127).

A.136 'PUSHFx': Empuja el registro de flags

```
PUSHF                       ; 9C                [186]
PUSHFD                      ; o32 9C            [386]
PUSHFW                      ; o16 9C            [186]
```

'PUSHFW' saca una palabra desde la pila y la almacena en los 16 bits inferiores del registro de flags (o en todo el registro, en procesadores inferiores al 386). 'PUSHFD' saca una palabra doble y la almacena en todo el registro de flags.

'PUSHF' es un mnemónico sinónimo para 'PUSHFW' o 'PUSHFD', dependiendo de la configuración actual de 'BITS'.

Ver también 'POPF' (sección A.128).

A.137 'POR': XOR a nivel de bit del MMX

```
PXOR mmxreg,r/m64            ; 0F EF /r            [PENT,MMX]
```

'PXOR' realiza una operación XOR a nivel de bit entre sus dos operandos (esto es, cada bit del resultado es 1 si y sólo si exactamente uno de los bits correspondientes de las dos entradas es 1), y almacena el resultado en el operando (el primero) destino.

A.138 'RCL', 'RCR': Rotar a nivel de bit usando el bit de acarreo

```
RCL r/m8,1                   ; D0 /2               [8086]
RCL r/m8,CL                  ; D2 /2               [8086]
RCL r/m8,imm8                ; C0 /2 ib            [286]
RCL r/m16,1                  ; o16 D1 /2           [8086]
RCL r/m16,CL                 ; o16 D3 /2           [8086]
RCL r/m16,imm8               ; o16 C1 /2 ib        [286]
RCL r/m32,1                  ; o32 D1 /2           [386]
RCL r/m32,CL                 ; o32 D3 /2           [386]
RCL r/m32,imm8               ; o32 C1 /2 ib        [386]

RCR r/m8,1                   ; D0 /3               [8086]
RCR r/m8,CL                  ; D2 /3               [8086]
RCR r/m8,imm8                ; C0 /3 ib            [286]
RCR r/m16,1                  ; o16 D1 /3           [8086]
RCR r/m16,CL                 ; o16 D3 /3           [8086]
RCR r/m16,imm8               ; o16 C1 /3 ib        [286]
RCR r/m32,1                  ; o32 D1 /3           [386]
RCR r/m32,CL                 ; o32 D3 /3           [386]
RCR r/m32,imm8               ; o32 C1 /3 ib        [386]
```

'RCL' y 'RCR' realizan una operación de rotación de 9-bit, 17-bit o 33-bit, implicando el operando fuente/destino dado y el bit de acarreo. Así, por ejemplo, en la operación 'RCR AL,1', se realiza una rotación de 9-bit en la cual 'AL' es desplazado a la izquierda en 1, el bit superior de 'AL' se mueve al flag de acarreo, y el valor original del flag de acarreo se coloca como el bit inferior de 'AL'.

El número de bits a rotar se da como segundo operando. En los procesadores superiores al 8086 sólo se consideran los cinco bits inferiores del contador de rotación.

Puedes forzar la forma larga (286 en adelante, empezando con un byte 'C1') de 'RCL foo,1' usando un prefijo 'BYTE': 'RCL foo,BYTE 1'. De forma similar con 'RCR'.

A.139 'RDMSR': Leer los registros específicos del modelo

```
RDMSR                        ; 0F 32               [PENT]
```

'RDMSR' lee el registro específico del modelo del procesador (MSR) cuyo índice se almacena en 'ECX', y almacena el resultado en 'EDX:EAX'. Ver también 'WRMSR' (sección A.165).

A.140 'RDPMC': Leer los contadores de control de rendimiento

```
RDPMC                        ; 0F 33               [P6]
```

'RDPMC' lee el contador de control de rendimiento cuyo índice se almacena en 'ECX', y almacena el resultado en 'EDX:EAX'.

A.141 'RDTSC': Leer el contador de la fecha

```
RDTSC                       ; 0F 31               [PENT]
```

'RDTSC' lee el contador de la fecha del procesador en 'EDX:EAX'.

A.142 'RET', 'RETF', 'RETN': Vuelve desde una llamada a un procedimiento.

```
RET                         ; C3                  [8086]
RET imm16                   ; C2 iw               [8086]

RETF                        ; CB                  [8086]
RETF imm16                  ; CA iw               [8086]

RETN                        ; C3                  [8086]
RETN imm16                  ; C2 iw               [8086]
```

'RET', y su sinónimo exacto 'RETN', saca 'IP' o 'EIP' desde la pila y transfiere el control a la nueva dirección. Opcionalmente, si se proporciona un segundo operando, incrementa posteriormente el puntero de pila en 'imm16' bytes después de sacar la dirección de retorno.

'RETF' ejecuta un retorno lejano: tras sacar 'IP'/'EIP', luego saca 'CS', y _luego_ si está presente el argumento opcional incrementa el puntero de pila en lo que corresponda.

A.143 'ROL', 'ROR': Rotar a nivel de bit

```
ROL r/m8,1                  ; D0 /0               [8086]
ROL r/m8,CL                 ; D2 /0               [8086]
ROL r/m8,imm8               ; C0 /0 ib            [286]
ROL r/m16,1                 ; o16 D1 /0           [8086]
ROL r/m16,CL                ; o16 D3 /0           [8086]
ROL r/m16,imm8              ; o16 C1 /0 ib        [286]
ROL r/m32,1                 ; o32 D1 /0           [386]
ROL r/m32,CL                ; o32 D3 /0           [386]
ROL r/m32,imm8              ; o32 C1 /0 ib        [386]

ROR r/m8,1                  ; D0 /1               [8086]
ROR r/m8,CL                 ; D2 /1               [8086]
ROR r/m8,imm8               ; C0 /1 ib            [286]
ROR r/m16,1                 ; o16 D1 /1           [8086]
ROR r/m16,CL                ; o16 D3 /1           [8086]
ROR r/m16,imm8              ; o16 C1 /1 ib        [286]
ROR r/m32,1                 ; o32 D1 /1           [386]
ROR r/m32,CL                ; o32 D3 /1           [386]
ROR r/m32,imm8              ; o32 C1 /1 ib        [386]
```

'ROL' y 'ROR' realizan una rotación a nivel de bit en el operando destino/fuente (el primero). Así, por ejemplo, en la operación 'ROR AL,1'. se realiza una operación de 8-bits en la cual se desplaza 'AL' a la izquierda en 1 y el bit superior original de 'AL' se mueve a la posición de bit inferior.

El número de bits a rotar se da como segundo operando. Sólo en los procesadores superiores al 8086 se consideran los 3, 4 ó 5 bits

inferiores de la cuenta de rotación (dependiendo del tamaño del operando fuente).

Puedes forzar la forma larga (286 y posteriores, empezando con un byte 'C1') de 'ROL foo,1' usando un prefijo BYTE: 'ROL foo,BYTE 1'. De forma similar con 'ROR'.

A.144 'RSM': Continúa desde el modo de gestión del sistema

```
RSM                         ; 0F AA               [PENT]
```

'RSM' devuelve al procesador a su modo de operación normal cuando estaba en el modo de gestión del sistema.

A.145 'SAHF': Almacena AH en los flags

```
SAHF                        ; 9E                  [8086]
```

'SAHF' activa el byte inferior de la palabra de flags de acuerdo con el contenido del registro 'AH'. Ver también 'LAHF' (sección A.90)

A.146 'SAL', 'SAR': Desplazamientos aritméticos a nivel de bit.

```
SAL r/m8,1                  ; D0 /4               [8086]
SAL r/m8,CL                 ; D2 /4               [8086]
SAL r/m8,imm8               ; C0 /4 ib            [286]
SAL r/m16,1                 ; o16 D1 /4           [8086]
SAL r/m16,CL                ; o16 D3 /4           [8086]
SAL r/m16,imm8              ; o16 C1 /4 ib        [286]
SAL r/m32,1                 ; o32 D1 /4           [386]
SAL r/m32,CL                ; o32 D3 /4           [386]
SAL r/m32,imm8              ; o32 C1 /4 ib        [386]

SAR r/m8,1                  ; D0 /0               [8086]
SAR r/m8,CL                 ; D2 /0               [8086]
SAR r/m8,imm8               ; C0 /0 ib            [286]
SAR r/m16,1                 ; o16 D1 /0           [8086]
SAR r/m16,CL                ; o16 D3 /0           [8086]
SAR r/m16,imm8              ; o16 C1 /0 ib        [286]
SAR r/m32,1                 ; o32 D1 /0           [386]
SAR r/m32,CL                ; o32 D3 /0           [386]
SAR r/m32,imm8              ; o32 C1 /0 ib        [386]
```

'SAL' y 'SAR' realizan una operación de desplazamiento aritmético sobre el operando destino/fuente dado (el primero). Los bits vacíos se rellenan en 'SAL' con ceros, y en 'SAR' con copias del bit superior original del operando fuente.

'SAL' es sinónimo de 'SHL' (ver la sección A.152). NASM ensamblará cualquier de los dos con el mismo código, pero NDISASM siempre desensamblará este código como 'SHL'.

El número de bits a desplazar se da como segundo operando. Sólo en los procesadores superiores al 8086 se consideran los 3, 4 ó 5 bits inferiores de la cuenta de rotación (dependiendo del tamaño del operando fuente).

Puedes forzar la forma larga (286 o posteriores, empezando con un byte 'C1') de 'SAL foo,1' usando un prefijo 'BYTE': 'SAL foo,BYTE 1'. De forma similar con 'SAR'.

A.147 'SALC': Establece AL desde el flag de acarreo

```
SALC                        ; D6                [8086,UNDOC]
```

'SALC' es una instrucción no documentada relativamente nueva, similar en concepto a 'SETcc' (sección A.150). Su función es poner 'AL' a cero si el flag de acarreo está limpio, o a '0xFF' si está activo.

A.148 'SBB': Resta con préstamo

```
SBB r/m8,reg8               ; 18 /r             [8086]
SBB r/m16,reg16             ; o16 19 /r         [8086]
SBB r/m32,reg32             ; o32 19 /r         [386]

SBB reg8,r/m8               ; 1A /r             [8086]
SBB reg16,r/m16             ; o16 1B /r         [8086]
SBB reg32,r/m32             ; o32 1B /r         [386]

SBB r/m8,imm8               ; 80 /3 ib          [8086]
SBB r/m16,imm16             ; o16 81 /3 iw      [8086]
SBB r/m32,imm32             ; o32 81 /3 id      [386]

SBB r/m16,imm8              ; o16 83 /3 ib      [8086]
SBB r/m32,imm8              ; o32 83 /3 ib      [8086]

SBB AL,imm8                 ; 1C ib             [8086]
SBB AX,imm16                ; o16 1D iw         [8086]
SBB EAX,imm32               ; o32 1D id         [386]
```

'SBB' realiza la resta entre enteros: resta su segundo operando, más el valor del flag de acarreo, de su primero, y deja el resultado en el operando (el primero) destino. Los flags se activan de forma acorde con el resultado de la operación: en particular, se afecta al flag de acarreo y se puede usar en subsiguientes instrucciones 'SBB'.

En las formas con un inmediato de 8-bit como segundo operando y un primer operando más largo, el segundo operando se considera con signo, y se le hace extensión de signo hasta la longitud del primer operando. En estos casos, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

Para restar un número de otro sin sumar el contenido del flag de acarreo, usa 'SUB' (sección A.159).

A.149 'SCASB', 'SCASW', 'SCASD': Buscar cadena

```
SCASB                       ; AE                [8086]
SCASW                       ; o16 AF            [8086]
SCASD                       ; o32 AF            [386]
```

'SCASB' compara el byte que está en 'AL' con el byte en '[ES:DI]' o '[ES:EDI]', y activa los flags de acuerdo a esto. Luego incrementa o decrementa (dependiendo del flag de dirección: incrementa si está limpio, decrementa si está activo) 'DI' (o 'EDI').

Si la dirección es de 16 bits se usa el registro 'DI', y 'EDI' si es de 32 bits. Si necesitas usar un tamaño de dirección distinto de la configuración actual de 'BITS', puedes usar un prefijo 'a16' o 'a32' explícito.

Los prefijos de superposición de segmentos no tienen efecto en esta instrucción: el uso de 'ES' para cargar desde '[DI]' o '[EDI]' no se puede solapar.

'SCASW' y 'SCASD' funcionan de la misma manera, pero comparan una palabra con 'AX' o doble palabra con 'EAX' en lugar de un byte con 'AL', e incrementan o decrementan el registro de direccionamiento en 2 ó en 4 en lugar de en 1.

Los prefijos 'REPE' y 'REPNE' (equivalentes a 'REPZ' y 'REPNZ') se pueden usar para repetir una instrucción hasta 'CX' veces (o 'ECX' - nuevamente el tamaño de la dirección elige cuál) hasta que se encuentre el primer byte distinto o igual, depende.

A.150 'SETcc': Establece un registro dependiendo de una condición

```
SETcc r/m8                    ; 0F 90+cc /2          [386]
```

'SETcc' pone a cero el operando de 8-bits dado si no se satisface la condición, y a 1 si sí lo hace.

A.151 'SGDT', 'SIDT', 'SLDT': Almacena los punteros de la tabla de descriptores

```
SGDT mem                      ; 0F 01 /0             [286,PRIV]
SIDT mem                      ; 0F 01 /1             [286,PRIV]
SLDT r/m16                    ; 0F 00 /0             [286,PRIV]
```

'SGDT' y 'SIDT' toman como operando un área de memoria de 6 bytes: almacenan el contenido de la GDTR (Global Descriptor Table Register) o la IDTR (Interrupt Descriptor Table Register) en ese área como una dirección lineal de 32-bits y un límite de tamaño de 16-bit (en ese orden). Son las únicas instrucciones que usan directamente direcciones _lineales_, en lugar del par segmento/offset.

'SLDT' almacena en el operando dado, el selector de segmento correspondiente a la LDT (Local Descriptor Table).

Ver también 'LGDT', 'LIDT' y 'LLDT' (sección A.95).

A.152 'SHL', 'SHR': Desplazamientos lógicos a nivel de bit.

```
SHL r/m8,1                    ; D0 /4                [8086]
SHL r/m8,CL                   ; D2 /4                [8086]
SHL r/m8,imm8                 ; C0 /4 ib             [286]
SHL r/m16,1                   ; o16 D1 /4            [8086]
SHL r/m16,CL                  ; o16 D3 /4            [8086]
SHL r/m16,imm8                ; o16 C1 /4 ib         [286]
SHL r/m32,1                   ; o32 D1 /4            [386]
SHL r/m32,CL                  ; o32 D3 /4            [386]
SHL r/m32,imm8                ; o32 C1 /4 ib         [386]

SHR r/m8,1                    ; D0 /5                [8086]
SHR r/m8,CL                   ; D2 /5                [8086]
SHR r/m8,imm8                 ; C0 /5 ib             [286]
SHR r/m16,1                   ; o16 D1 /5            [8086]
SHR r/m16,CL                  ; o16 D3 /5            [8086]
SHR r/m16,imm8                ; o16 C1 /5 ib         [286]
SHR r/m32,1                   ; o32 D1 /5            [386]
SHR r/m32,CL                  ; o32 D3 /5            [386]
```

```
SHR r/m32,imm8              ; o32 C1 /5 ib        [386]
```

'SHL' y 'SHR' realizan un desplazamiento lógico sobre el operando fuente/destino dado. Los bits desocupados se rellenan con cero.

'SAL' es un sinónimo de 'SHL' (ver la sección A.146). NASM ensamblará cualquiera de ellos con el mismo código, pero NDISASM siempre desensamblará este código como 'SHL'.

El segundo operando da el número de bits a desplazar. En los procesadores superiores al 8086 sólo se consideran los 3, 4 ó 5 bits inferiores de la cuenta de desplazamiento (dependiendo del tamaño del operando fuente).

Puedes forzar la forma más larga (286 y posteriores, empezando con un byte 'C1') de 'SHL foo,1' usando un prefijo 'BYTE': 'SHL foo,BYTE 1'. De forma similar con 'SHR'.

A.153 'SHLD', 'SHRD': Desplazamientos de doble precisión a nivel de bit

```
SHLD r/m16,reg16,imm8       ; o16 0F A4 /r ib     [386]
SHLD r/m16,reg32,imm8       ; o32 0F A4 /r ib     [386]
SHLD r/m16,reg16,CL         ; o16 0F A5 /r        [386]
SHLD r/m16,reg32,CL         ; o32 0F A5 /r        [386]

SHRD r/m16,reg16,imm8       ; o16 0F AC /r ib     [386]
SHRD r/m32,reg32,imm8       ; o32 0F AC /r ib     [386]
SHRD r/m16,reg16,CL         ; o16 0F AD /r        [386]
SHRD r/m32,reg32,CL         ; o32 0F AD /r        [386]
```

'SHLD' realiza un desplazamiento a la izquierda de doble precisión. Conceptualmente coloca su segundo operando a la derecha de su primero, luego desplaza a la izquierda la cadena entera de bits así generada el número de bits especificado por su tercer operando. Luego sólo actualiza el _primer_ operando de acuerdo con el resultado de esto. El segundo operando no se modifica.

'SHRD' realiza el correspondiente desplazamiento a la derecha: conceptualmente coloca el segundo operando a la _izquierda_ del primero, desplaza a la derecha la cadena entera de bits, y actualiza sólo el primer operando.

Por ejemplo, si 'EAX' contiene '0x01234567' y 'EBX' contiene '0x89ABCDEF', luego la instrucción 'SHLD EAX,EBX,4' actualizaría 'EAX' para contener '0x12345678'. Bajo las mismas condiciones, 'SHRD EAX,EBX,4' actualizaría 'EAX' para contener '0xF0123456'.

El número de bits a desplazar viene dado por su tercer operando. Sólo se consideran los 5 bits inferiores de la cuenta de desplazamiento.

A.154 'SMI': Interrupción de gestión del sistema

```
SMI                         ; F1                  [386,UNDOC]
```

Este es un código de operación que está soportado aparentemente por algunos procesadores AMD (la cual es por la que pueden generar el mismo código de operación como 'INT1'), y coloca la máquina en el modo de gestión del sistema, un modo especial de depuración.

A.155 'SMSW': Almacena la palabra de estado de la máquina

```
SMSW r/m16                    ; 0F 01 /4          [286,PRIV]
```

'SMSW' almacena la mitad inferior del registro de control 'CR0' (o la palabra de estado de la máquina, en los procesadores 286) en el operando destino. Ver también 'LMSW' (sección A.96).

A.156 'STC', 'STD', 'STI': Activa flags

```
STC                           ; F9                [8086]
STD                           ; FD                [8086]
STI                           ; FB                [8086]
```

Estas instrucciones activan diversos flags. 'STC' activa el flag de acarreo; 'STD' activa el flag de dirección; y 'STI' activa el flag de interrupción (así permite interrupciones).

Para limpiar el flag de acarreo, dirección o interrupción, usa las instrucciones 'CLC', 'CLD' y 'CLI' (sección A.15). Para invertir el flag de acarreo, usa 'CMC' (sección A.16).

A.157 'STOSB', 'STOSW', 'STOSD': Almacena un byte a una cadena

```
STOSB                         ; AA                [8086]
STOSW                         ; o16 AB            [8086]
STOSD                         ; o32 AB            [386]
```

'STOSB' almacena el byte que está en 'AL' en '[ES:DI]' o '[ES:EDI]', y activa los flags de acuerdo con esto. Luego incrementa o decrementa (dependiendo del flag de dirección: incrementa si el flag está limpio, decrementa si está activo) 'DI' (o 'EDI').

Si el tamaño de la dirección es 16 bits se usa el registro 'DI', y 'EDI' si es de 32 bits. Si necesitas usar un tamaño de dirección distinto de la configuración actual de 'BITS', puedes usar un prefijo 'a16' o 'a32' explícito.

Los prefijos de superposición de segmentos no tienen efecto en esta instrucción: no se puede superponer el uso de 'ES' para almacenar a '[DI]' o '[EDI]'.

'STOSW' y 'STOSD' funcionan del mismo modo, pero almacenan una palabra en 'AX' o una doble palabra en 'EAX' en lugar de un byte en 'AL', e incrementan o decrementan el registro de direcciones en 2 ó en 4 en lugar de en 1.

Se puede usar el prefijo 'REP' para repetir la instrucción 'CX' veces (o 'ECX' - nuevamente, el tamaño de la dirección elige cuál).

A.158 'STR': Almacena el registro de tarea

```
STR r/m16                     ; 0F 00 /1          [286,PRIV]
```

'STR' almacena en su operando el selector de segmento correspondiente al contenido del registro de tarea.

A.159 'SUB': Resta enteros

```
SUB r/m8,reg8                 ; 28 /r             [8086]
SUB r/m16,reg16               ; o16 29 /r         [8086]
SUB r/m32,reg32               ; o32 29 /r         [386]
```

```
SUB reg8,r/m8                ; 2A /r              [8086]
SUB reg16,r/m16              ; o16 2B /r          [8086]
SUB reg32,r/m32              ; o32 2B /r          [386]

SUB r/m8,imm8                ; 80 /5 ib           [8086]
SUB r/m16,imm16              ; o16 81 /5 iw       [8086]
SUB r/m32,imm32              ; o32 81 /5 id       [386]

SUB r/m16,imm8               ; o16 83 /5 ib       [8086]
SUB r/m32,imm8               ; o32 83 /5 ib       [386]

SUB AL,imm8                  ; 2C ib              [8086]
SUB AX,imm16                 ; o16 2D iw          [8086]
SUB EAX,imm32                ; o32 2D id          [386]
```

'SUB' realiza la resta entre enteros: resta su segundo operando del primero, y deja el resultado en su operando (el primero) destino. Los flags se activan de acuerdo con el resultado de la operación: en particular, se afecta al flag de acarreo y se puede usar en subsiguientes instrucciones 'SBB' (sección A.148).

En las formas con un valor inmediato de 8-bits como segundo operando y un primer operando más largo, el segundo operando se considera con signo, y se le hace extensión de signo hasta la longitud del primero. En estos casos, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

A.160 'TEST': Comprueba bits (conceptualmente un AND a nivel de bit)

```
TEST r/m8,reg8               ; 84 /r              [8086]
TEST r/m16,reg16             ; o16 85 /r          [8086]
TEST r/m32,reg32             ; o32 85 /r          [386]

TEST r/m8,imm8               ; F6 /7 ib           [8086]
TEST r/m16,imm16             ; o16 F7 /7 iw       [8086]
TEST r/m32,imm32             ; o32 F7 /7 id       [386]

TEST AL,imm8                 ; A8 ib              [8086]
TEST AX,imm16                ; o16 A9 iw          [8086]
TEST EAX,imm32               ; o32 A9 id          [386]
```

'TEST' realiza 'mentalmente' un AND a nivel de bit de sus dos operandos, y afecta a los flags como si la operación hubiese tenido lugar, pero no almacena en ningún lugar el resultado de la operación.

A.161 'UMOV': Mueve datos del usuario

```
UMOV r/m8,reg8               ; 0F 10 /r           [386,UNDOC]
UMOV r/m16,reg16             ; o16 0F 11 /r       [386,UNDOC]
UMOV r/m32,reg32             ; o32 0F 11 /r       [386,UNDOC]

UMOV reg8,r/m8               ; 0F 12 /r           [386,UNDOC]
UMOV reg16,r/m16             ; o16 0F 13 /r       [386,UNDOC]
UMOV reg32,r/m32             ; o32 0F 13 /r       [386,UNDOC]
```

Esta instrucción no documentada se usa en emuladores en circuito para acceder a memoria del usuario (opuesta a la memoria del host). Se usa como una instrucción 'MOV' normal de memoria/registro o registro/registro, pero accediendo a espacio de usuario.

A.162 'VERR', 'VERW': Verifica la disponibilidad de escritura/lectura de un segmento

```
VERR r/m16                    ; 0F 00 /4            [286,PRIV]

VERW r/m16                    ; 0F 00 /5            [286,PRIV]
```

'VERR' activa el flag de cero si el segmento especificado por el selector en su operando se puede leer desde el nivel de privilegio actual. 'VERW' activa el flag de cero si el segmento se puede escribir.

A.163 'WAIT': Espera al procesador de coma flotante

```
WAIT                          ; 9B                  [8086]
```

'WAIT', en sistemas 8086 con una FPU 8087 separada, espera a que la FPU termine cualquier operación en la que estuviese ocupada antes de continuar con las operaciones del procesador principal, por eso (por ejemplo) un almacenamiento de la FPU en la memoria principal se puede garantizar que está completo antes de que la CPU intente leer el resultado.

En procesadores superiores, 'WAIT' es innecesario para este propósito, y tiene el propósito alternativo de asegurar que cualquier excepción no enmascarada de la FPU pendiente ha ocurrido antes de que la CPU continúe la ejecución.

A.164 'WBINVD': Escribe de vuelta la caché y la invalida

```
WBINVD                        ; 0F 09               [486]
```

'WBINVD' invalida y vacía la caché interna del procesador, y hace que el procesador indique a las cachés externas que hagan lo mismo. Escribe el contenido de las cachés de vuelta a la memoria, por eso no se pierden datos. Para limpiar las cachés rápidamente sin preocuparse de escribir primero los datos, usa 'INVD' (sección A.84).

A.165 'WRMSR': Escribe los registros específicos del modelo

```
WRMSR                         ; 0F 30               [PENT]
```

'WRMSR' escribe el valor en 'EDX:EAX' al registro específico del modelo (MSR) del procesador cuyo índice se almacena en 'ECX'. Ver también 'RDMSR' (sección A.139).

A.166 'XADD': Intercambia y suma

```
XADD r/m8,reg8                ; 0F C0 /r            [486]
XADD r/m16,reg16              ; o16 0F C1 /r        [486]
XADD r/m32,reg32              ; o32 0F C1 /r        [486]
```

'XADD' intercambia los valores de sus dos operandos, y luego los suma y escribe el resultado en su operando (el primero) destino. Esta instrucción se puede usar con un prefijo 'LOCK' para cuestiones de sincronismo en multiprocesadores.

A.167 'XBTS': Extrae una cadena de bits

```
XBTS reg16,r/m16              ; o16 0F A6 /r        [386,UNDOC]
```

```
XBTS reg32,r/m32            ; o32 0F A6 /r        [386,UNDOC]
```

No parece haber una documentación clara para esta instrucción: lo mejor que he encontrado dice 'Toma una cadena de bits de su primer operando y la pone en su segundo operando'. Sólo está presente en los primeros procesadores 386, y entra en conflicto con el código de operación de 'CMPXCHG486'. NASM lo soporta sólo para completar. Su contrapartida es 'IBTS' (ver la sección A.75).

A.168 'XCHG': Intercambia

```
XCHG reg8,r/m8              ; 86 /r               [8086]
XCHG reg16,r/m8             ; o16 87 /r           [8086]
XCHG reg32,r/m32            ; o32 87 /r           [386]

XCHG r/m8,reg8              ; 86 /r               [8086]
XCHG r/m16,reg16            ; o16 87 /r           [8086]
XCHG r/m32,reg32            ; o32 87 /r           [386]

XCHG AX,reg16               ; o16 90+r            [8086]
XCHG EAX,reg32              ; o32 90+r            [386]
XCHG reg16,AX               ; o16 90+r            [8086]
XCHG reg32,EAX              ; o32 90+r            [386]
```

'XCHG' intercambia los valores de sus dos operandos. Se puede usar con un prefijo 'LOCK' para cuestiones de sincronismo en multiprocesadores.

'XCHG AX,AX' o 'XCHG EAX,EAX' (dependiendo de la configuración de 'BITS') genera el código de operación '90h', y por eso es sinónimo de 'NOP' (sección A.109).

A.169 'XLATB': Traduce un byte en una tabla de búsqueda

```
XLATB                       ; D7                  [8086]
```

'XLATB' suma el valor en 'AL', tratado como un byte sin signo, a 'BX' o 'EBX', y carga el byte desde la dirección resultante (en el segmento especificado por 'DS') de nuevo en 'AL'.

Si el tamaño de la dirección es de 16 bits se usa el registro base 'BX', y 'EBX' si es de 32 bits. Si necesitas usar un tamaño de dirección distinto del marcado por 'BITS', puedes usar un prefijo 'a16' o 'a32' explícito.

El registro de segmento usado para cargar desde '[BX+AL]' o '[EBX+AL]' se puede superponer usando como prefijo un nombre de registro de segmento (por ejemplo, 'es xlatb').

A.170 'XOR': OR exclusivo a nivel de bit

```
XOR r/m8,reg8               ; 30 /r               [8086]
XOR r/m16,reg16             ; o16 31 /r           [8086]
XOR r/m32,reg32             ; o32 31 /r           [386]

XOR reg8,r/m8               ; 32 /r               [8086]
XOR reg16,r/m16             ; o16 33 /r           [8086]
XOR reg32,r/m32             ; o32 33 /r           [386]

XOR r/m8,imm8               ; 80 /6 ib            [8086]
XOR r/m16,imm16             ; o16 81 /6 iw        [8086]
```

```
XOR r/m32,imm32             ; o32 81 /6 id        [386]

XOR r/m16,imm8              ; o16 83 /6 ib        [8086]
XOR r/m32,imm8              ; o32 83 /6 ib        [386]

XOR AL,imm8                 ; 34 ib               [8086]
XOR AX,imm16                ; o16 35 iw           [8086]
XOR EAX,imm32               ; o32 35 id           [386]
```

'XOR' realiza un OR exclusivo a nivel de bit entre sus dos operandos (esto es, cada bit del resultado es un 1 si y sólo si exactamente uno de los bits correspondientes de las dos entradas es 1), y almacena el resultado en su operando (el primero) destino.

En las formas con un inmediato de 8-bits como segundo operando y un primer operando más largo, el segundo operando se considera con signo, y se le hace extensión de signo hasta la longitud del primer operando. En estos casos, se necesita el calificador 'BYTE' para forzar a NASM a generar esta forma de la instrucción.

La instrucción MMX 'PXOR' (ver la sección A.137) realiza la misma operación en los registros MMX de 64-bits.