5. El lenguaje ensamblador: operaciones

En este capítulo, vamos a ver cómo los programas hacen uso de la arquitectura del procesador descrita en el capítulo anterior. Para esto necesitamos conocer el lenguaje de la máquina. Este capítulo presenta los rudimentos del lenguaje ensamblador centrado en el caso particular de la IA32.

La presentación de las operaciones en ensamblador se hace estableciendo una relación con sus equivalentes en los lenguajes de alto nivel. Esto presenta dos ventajas: la primera, que se parte de algo conocido y se tiene un punto de referencia para entender cómo usar el ensamblador; la segunda, que se entiende cómo efectúan realmente sus operaciones los lenguajes de alto nivel.

5.1 PRESENTACIÓN DEL LENGUAJE ENSAMBLADOR

Antes de entrar en el lenguaje ensamblador propiamente dicho, haremos algunas precisiones sobre el lenguaje de máquina.

Lenguaje de máquina

Una instrucción es una operación que se efectúa sobre unos operandos (operandos fuente) y produce un resultado (operando destino). Es necesario especificar el código de operación y los operandos; para estos últimos se debe especificar dónde se encuentran: en la instrucción misma, en los registros o en memoria. En este último caso, la dirección puede ser estática (direccionamiento directo) o dinámica (direccionamiento indexado, indirecto, etc.).

Como hemos mencionado anteriormente, las instrucciones están en la memoria codificadas en binario. La codificación se parte en varios campos; lo que hemos llamado el formato de instrucción.

Vemos un ejemplo: tenemos una máquina de 16 bits —direcciones y datos—, con 16 registros, 64 operaciones e instrucciones con dos operandos; el primer operando siempre es un registro y el segundo puede tener uno de 4 modos de direccionamiento: inmediato, directo, de registro e indirecto por registro.

En una máquina como esta tenemos que codificar cuatro elementos:

- Código de operación. Puesto que hay 64 operaciones, necesitamos 6 bits para codificarlas. La codificación es arbitraria en principio; puede que haya algún condicionamiento debido a la estructura, pero, en general, no hay mayores restricciones para establecerla. Se tratará entonces de una codificación del estilo: 000000 es `ADD`, 000001 es `SUB`, 000010 es `AND`, etc.
- Modo de direccionamiento del segundo operando. Como el primer operando es siempre un registro, no es necesario codificar su forma de direccionamiento. En cuanto al segundo, dado que puede tener uno cualquiera de cuatro modos de direccionamiento, necesitamos 2 bits para codificarlo; algo del estilo: 00 es inmediato, 10 es directo, 01 es de registro y 11 es indirecto por registro.
- Primer operando. El primer operando es un registro, puesto que hay 16, necesitamos 4 bits para codificarlo.
- Segundo operando. Si el segundo operando es de modo inmediato o directo, necesitamos 16 bits para codificarlo —puesto que la máquina es de 16 bits—; si es de registro o indirecto por registro, necesitamos 4 bits para codificarlo.

El segundo operando introduce un problema: según su modo de direccionamiento requiere un número diferente de bits. Tenemos dos posibles soluciones:

- Reservar un número fijo de bits. Tendríamos que reservar el máximo posible; 16, en este caso. Eso implica que, cuando el operando sea un registro, se desperdiciaría espacio, pero tendríamos un solo tamaño de instrucción.
- Tener un número variable de bits. En este caso, el operando tendría 4 ó 16 bits dependiendo del modo de direccionamiento. Esto implica que tendríamos dos formatos de instrucción, uno más largo que el otro. En este ejemplo vamos a tomar esta opción.

Para los dos primeros modos de direccionamiento, el formato de instrucción es:

Operación	Modo	Registro	Constante o Dirección
6 bits	2 bits	4 bits	16 bits

En total, 28 bits; no es múltiplo de 8, así que no es un número entero de bytes. Esto nos conduce a introducir un relleno, “basura”, que no codifica nada, para tener un número entero de bytes:

Operación	Modo	Relleno	Registro	Constante o Dirección
6 bits	2 bits	4 bits	4 bits	16 bits

Ahora una instrucción de este tipo tendría 4 bytes.

Para los otros dos modos de direccionamiento tenemos:

Operación	Modo	Registro	Registro
6 bits	2 bits	4 bits	4 bits

Es decir dos bytes.

Con estas convenciones la instrucción: `SUB reg3, 5`, se codificaría:

000001 00 0011 0000 00000000 00000101

En tanto que la instrucción: `SUB reg3, [reg5]`, se codificaría:

000001 11 0011 0101

Por supuesto, la unidad de control sabe "partir" la instrucción para extraer la información que necesita. Esto es, justamente, el paso del ciclo de ejecución que llamamos *decodificación de instrucción*.

Lenguaje ensamblador

Es fácil darse cuenta de que programar escribiendo largas series de números binarios no es nada cómodo. Por esto, se pensó en desarrollar un lenguaje simbólico: en lugar de números binarios vamos a escribir palabras que designan la operación, los operandos, el modo de direccionamiento, etc.

Este es el *lenguaje ensamblador*. Puesto que es simbólico, la máquina no lo puede interpretar directamente —esta solo puede interpretar el lenguaje binario—, por lo que necesitamos un programa traductor que se encargue de leer un programa en ensamblador y lo traduzca a binario; este es el *programa ensamblador*.

El lenguaje ensamblador es una notación del estilo de la que usamos para expresar los programas de la máquina en el capítulo anterior. Note que esta notación es muy cercana al formato de la instrucción: código de operación, primer operando y segundo operando —solo el modo de direccionamiento se encuentra en otro lugar— ; por lo cual la tarea del ensamblador es relativamente sencilla: traducir cada una de las palabras a su código binario y con ellas ensamblar la instrucción; de aquí viene su nombre de “ensamblador”.

El hecho de tener este intermediario entre el programador y la máquina introduce una serie de ventajas. En primer lugar, claro está, los nombres simbólicos; el diseñador del ensamblador elige estos nombres según lo que le parezca más cómodo. Por ejemplo, para los registros, podemos elegir los nombres “reg0”, “reg1”, etc. O podemos elegir algo como “$0”, “$1”, etc. O, como Intel, “`eax`”, “`ebx`”, etc.

Pero hay otras ventajas; por ejemplo, podemos permitir que el programador escriba los números en la base que desee —2, 10, 16— y dejar que el ensamblador los traduzca a binario. En el caso del ensamblador de Microsoft —MASM, *Macro ASseMbler*—, este acepta que los números se escriban en diferentes bases: 26 ó 26D (base 10), 1AH (base 16), 11010B (binario) y 32Q (octal), y los traduce a binario.[1]

También, se puede introducir un mayor nivel de abstracción. Por ejemplo, en nuestra notación, el modo de direccionamiento está asociado al operando; no es un campo independiente como lo es en la instrucción de máquina.

Hay otras ventajas que trae consigo el ensamblador que veremos posteriormente.

---

[1] En el texto, según lo que parezca más claro, usaremos una u otra base.

Como se mencionó al comienzo, en las secciones siguientes vamos a partir de las instrucciones de alto nivel, y a mostrar cómo pueden ser implementadas usando las instrucciones de la máquina. Empezamos por movimiento de información, es decir, el equivalente de las asignaciones; después veremos la forma de evaluar expresiones y, por último, cómo se hacen las declaraciones.

5.2 LA ASIGNACIÓN

La instrucción de máquina que permite implementar la asignación es "`mov`" (del inglés *move*). En realidad, se puede decir que hay varias instrucciones de movimiento: entre registros, de memoria a registro y de registro a memoria. Pero podemos hacer la abstracción y suponer que es una sola; el tipo de los operandos y su posición nos indicará, en cada caso, cuál es la operación deseada.

Como se mencionó anteriormente, es una instrucción de dos operandos, y asigna el de la derecha al de la izquierda. Lo mismo que en los lenguajes de alto nivel, el elemento de la derecha no pierde su valor original; pero el elemento que de la izquierda adquiere el valor del otro.

Esta instrucción permite realizar diversas acciones:

- Traer una variable (cargar) de memoria a un registro. Las operaciones no se pueden efectuar directamente sobre las variables en memoria; en todo caso, aunque se pudiera, es más rápido hacer las operaciones en los registros. Por esto es necesario poder trasladar las variables de memoria a registros:

```
mov  ax, [122]
```

- Actualizar una variable en memoria con el valor de un registro. Después de realizados los cálculos es necesario actualizar el valor de la variable en memoria (el registro es una suerte de variable temporal, pero es necesario actualizar la variable original):

```
mov  [122], ax
```

- Mover valores entre registros. Mientras se hacen los cálculos, puede ser necesario replicar el valor en otros registros:

```
mov  bx, ax
```

- Inicializar registros o variables en memoria. Como en los lenguajes de alto nivel, en ocasiones es necesario asignar un valor constante:

```
mov  [122], 4
mov  bx, 4
```

A continuación, estudiaremos varios casos de asignación. Para no complicarnos con direcciones de memoria, trabajaremos con nombres simbólicos de variables; es decir, en lugar de escribir "`mov [122],4`", escribiremos algo del estilo "`mov x,4`".

Donde suponemos que la variable `x` está en la posición 122 de memoria. Posteriormente veremos que el ensamblador efectivamente presta este servicio de nombres simbólicos para las variables.

Asignación simple

Tomemos el siguiente segmento de código en C:

```
int x, y;
y = 3;
x = y;
```

La primera asignación no tiene problema:

```
mov  y, 3
```

En cuanto a la segunda, puesto que en la IA32 no se pueden tener dos operandos en memoria, nos tocaría pasar por un registro intermediario:

```
mov  ecx, y
mov  x, ecx
```

Lo anterior es la traducción más formal y literal; sin embargo, un programador inteligente, o un compilador con optimización, podrían efectuar otras acciones. Por ejemplo, podríamos mantener la variable `y`, temporalmente, en el registro `ebx`, con lo cual el código se convertiría en:

```
mov  ebx, 3
mov  x, ebx
```

Este código es más eficiente que el anterior porque solo hace un acceso a memoria —contra tres del otro—. Por supuesto, es posible que más adelante en el programa nos toque actualizar la variable `y` en memoria.

Los compiladores asignan variables en los registros en diversas circunstancias, por ejemplo, si se trata de una variable temporal que solo se utiliza brevemente para hacer unos cálculos, es más fácil hacerlos en los registros y no crearla en memoria; por otro lado, si una variable se usa mucho en algún momento del programa, es más eficiente mantenerla en un registro (luego, cuando ya no esté siendo tan utilizada, se actualizará en memoria).

Al hacer asignaciones, se debe tener en cuenta el tamaño de las variables: los caracteres se manejan en registros de 8 bits (`ah`, `al`, `bh`, etc.), los enteros cortos en registros de 16 bits (`ax`, `bx`, etc.) y los enteros en registros de 32 bits (`eax`, `ebx`, etc.). Los números de punto flotante se manejan en unos registros especiales que se encuentran en la unidad de punto flotante. Los apuntadores, como todas las direcciones en general, se manejan sobre 32 bits, es decir usando `eax`, `ebx`, etc.

Por ejemplo, el mismo código anterior, pero usando caracteres:

Código C	Código Ensamblador

`char x, y;`
`y = 'a';`
`x = y;`
	`mov  bl, 'a'`
	`mov  x, bl`

Note que, para facilitar el manejo de caracteres, el ensamblador permite escribir los caracteres en notación simbólica. No olvide, sin embargo, que es sólo una manera

de representar el código ASCII correspondiente. Dicho de otra forma, 'a' y 61H son completamente equivalentes para el ensamblador.

Asignación con apuntadores

En lenguaje de máquina, un apuntador es una variable (de 32 bits) que contiene la dirección de otra variable. Por ejemplo, si en la posición 1000 de memoria tenemos un entero, y en la posición 400 queremos tener un apuntador a este entero, basta con guardar el valor 1000 en la posición 400. Una forma de hacer esto es:

```
mov  [400], 1000
```

Esta es la realidad, pero es mejor pensar en términos abstractos y no en términos operativos. Pensar en términos de: "ebx contiene la dirección de otra variable en memoria", genera confusión; es mejor pensar "ebx apunta a esta variable".

Para los apuntadores se usa el direccionamiento indirecto. A continuación se muestra un ejemplo de su uso a la izquierda y a la derecha de una asignación:

Código C	Código Ensamblador

`int x, y;`
`int * p;`
`x = *p;`
	`mov  ebx, p`
	`mov  ecx, [ebx]`
	`mov  x, ecx`

`*p = y;`	`mov  ecx, y` `mov  [ebx], ecx`

Tenga presente que los apuntadores son valores de 32 bits, no necesariamente así lo apuntado por ellos. Por ejemplo:

Código C	Código Ensamblador

`char * p, *q;`
`*p = *q;`
	`mov  ebx, q`
	`mov  cl, [ebx]`
	`mov  ebx, p`
	`mov  [ebx], cl`

`p = q;`	`mov  ecx, q` `mov  p, ecx`

La primera asignación mueve caracteres; la segunda, apuntadores. Note que la primera no modifica `p`, sino lo apuntado por `p`, en tanto que la segunda sí modifica `p`.

Asignación con vectores

En el caso de los vectores se usa el direccionamiento indexado. Se debe tener presente que en los lenguajes de alto nivel, para acceder a un elemento, se usa un índice; en ensamblador, se usa un desplazamiento. En el caso de los caracteres los dos conceptos coinciden; empezaremos por este caso:

Código C	Código Ensamblador

`int  i;`
`char v[100], c;`
`c = v[i];`
	`mov  edi, i`
	`mov  cl, v[edi]`
	`mov  c, cl`

Como hemos mencionado anteriormente, para obtener el desplazamiento en el caso general, es necesario multiplicar el índice por el tamaño de los elementos; por ejemplo, para un vector de `short int` es necesario multiplicar el índice por 2, o, lo que es lo mismo, sumar el índice consigo mismo:

Código C	Código Ensamblador

`int i;`
`short int v[100], s;`
`s = v[i];`
	`mov edi, i`
	`add edi, edi`
	`mov cx, v[edi]`
	`mov s, cx`

Esta es una forma de hacerlo; pero también, es posible usar el indexamiento con escalamiento:

Código C	Código Ensamblador

`int i;`
`short int v[100], s;`
`s = v[i];`
	`mov edi, i`
	`mov cx, v[2*edi]`
	`mov s, cx`

Un vector de enteros se trata igual que el anterior pero usando un escalamiento de 4, y un registro de 4 bytes —eax, por ejemplo—.

En el caso general, si se tiene un vector cuyos elementos ocupan n bytes cada uno, la posición del i-ésimo elemento viene dada por $i*n$ —ver fig. 5.1—. Recuerde que el escalamiento solo se puede utilizar si n es 1, 2, 4 ó 8; en otros casos es necesario hacer la multiplicación explícitamente.

Tamaño de las variables

Si las variables son más grandes que cualquier registro —por ejemplo, si son `long long` o estructuras—, es necesario implementar la asignación copiando por partes; eventualmente con iteración. Por ejemplo:

Código C	Código Ensamblador

`long long x, y;`
`x = y;`
	`mov ebx, y`
	`mov x, ebx`
	`mov ebx, y+4`
	`mov x+4, ebx`

Note que `x` y `y` designan posiciones de memoria, en consecuencia, en notación

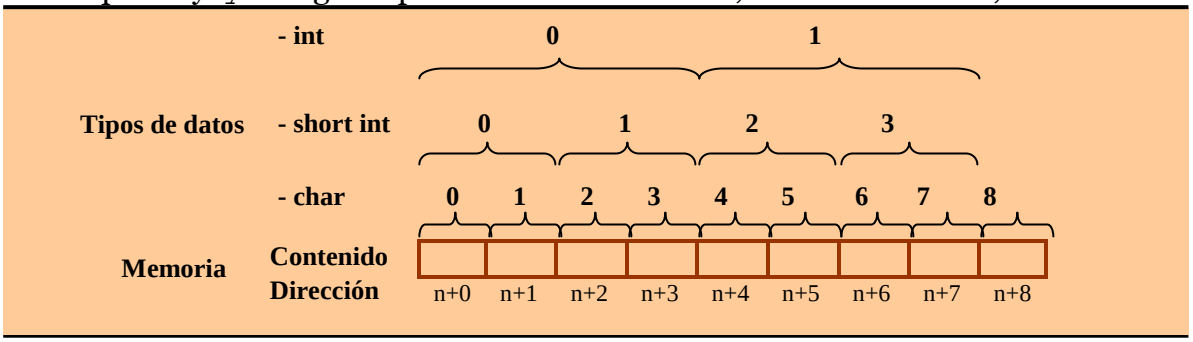


Fig. 5.1. Relación entre índices y desplazamientos para varios tipos de datos

simbólica, `x+4` no es el contenido de `x` más 4, sino la posición de memoria que se encuentra 4 bytes adelante de `x`. Puesto que `x` es un `long long`, ocupa 8 bytes; los 4 primeros están en las posiciones `x` a `x+3` y los 4 segundos están de `x+4` a `x+7`.

¿Qué ocurre cuando se mezclan tamaños? Según la definición de C, `y` debe truncarse a su byte menos significativo:

Código C	Código Ensamblador

`int  y;`
`char x;`
`x = y;`
	`mov  ebx, y`
	`mov  x, bl`

Recuerde que `bl` es el byte menos significativo de `ebx`.

También está el caso contrario: se asigna algo más pequeño a algo más grande:

```
char  y;
int x;
x = y;
```

Según la definición de C, la variable `y` debe extenderse al tamaño de `x`; es decir, se debe convertir el valor de `y` de 8 a 32 bits. Esto depende de si `y` tiene signo o no:

- Número sin signo: basta con agregarle a `y` 24 ceros a la izquierda.
- Número con signo: el bit de signo de `y` se replica 24 veces a la izquierda: si el signo es cero, el número sigue siendo positivo; si es 1, se mantiene negativo.

Lo primero es fácil de hacer, lo segundo es un poco más difícil. A continuación damos una posible solución que se entenderá plenamente más adelante:

Código C	Código Ensamblador

`int  x;`
`unsigned char y;`
`x = y;`
	`mov  ebx, 0`
	`mov  bl, y`
	`mov  x, ebx`

`int  x;`
`signed char y;`
`x = y;`
	`mov  bl, y`
	`shl  ebx, 24`
	`sar  ebx, 24`
	`mov  x, ebx`

Ahora, en realidad, es más fácil, porque la IA32 ¡tiene instrucciones que hacen esto!: movzx (*MOVe with Zero eXtend*) y movsx (*MOVe with Sign eXtend*). En consecuencia los dos ejemplos anteriores se resuelven así:

Código C	Código Ensamblador

`int  x;`
`unsigned char y;`
`x = y;`
	`movzx  ebx, y`
	`mov  x, ebx`

`int  x;`
`signed char y;`
`x = y;`
	`movsx  ebx, y`
	`mov  x, ebx`

Aquí la pregunta interesante es: ¿vale la pena incluir estas instrucciones en una arquitectura? Por supuesto, son útiles, pero, ¿se necesitan con frecuencia?, ¿no nos volverán la máquina un poco más complicada sin ganar mucho con ello? Este es el tipo de balances entre hardware y software que un diseñador debe resolver.

Cast

El programador en C percibe el *cast* como una conversión entre tipos de datos. Pero, desde el punto de vista de la máquina, el cast ejecuta diversas acciones según el caso:

- Conversión entre el mismo tipo de datos con diferentes tamaños. Este es el caso de la conversión entre `short` e `int`, etc. Se trata como se describió en la sección anterior.
- Conversión entre diferentes tipos de datos del mismo tamaño. Este es el caso de la conversión entre apuntadores, o entre enteros y apuntadores, etc. Se trata de la siguiente manera:

Código C	Código Ensamblador

`int  * p;`
`char * q;`
`p = (int *)q;`
	`mov  ebx, q`
	`mov  p, ebx`

  Es decir, en lo esencial, no se hace nada: sencillamente se asigna una variable a la otra.
- Conversión entre tipo de datos con representaciones diferentes. Este es el caso de la conversión entre `float` e `int`, etc. En este caso es necesario generar código para transformar una representación en la otra (por ejemplo, llamar una función para convertir de entero a flotante).

5.3 Instrucciones de la máquina

En ensamblador no se puede escribir expresiones como en los lenguajes de alto nivel. Para evaluar expresiones, hay que proceder paso a paso, evaluando cada operación en su turno. Para esto se dispone de las instrucciones de la máquina, las cuales implementan diversidad de operaciones. Empezaremos introduciendo algunas de estas instrucciones.

Instrucciones aritméticas

La instrucción ADD tiene dos operandos, y adiciona el segundo al primero. Los operandos son registros o posiciones de memoria; en general, pueden tener cualquier modo de direccionamiento, aunque, como es usual, los dos operandos no pueden estar en memoria. Los operandos pueden ser de 8, 16 ó 32 bits, pero los dos deben tener el mismo tamaño. No es posible sumar directamente un operando de 8 con uno de 16; es necesario convertir el número de 8 bits a 16[2].

[2] O el de 16 a 8, pero es más riesgoso porque pueden perderse dígitos.

La instrucción SUB resta el segundo operando al primero; de resto tiene las mismas características que ADD.

La instrucción NEG sirve para obtener el complemento a dos de un número. Tiene un solo parámetro, y deja el resultado en el mismo; este puede ser de 8, 16 ó 32 bits y tener cualquier modo de direccionamiento.[3]

En cuanto a la multiplicación, existen dos tipos de instrucciones; las del primer tipo son las más sencillas y las explicaremos en primer lugar (también es recomendable utilizarlas siempre que se pueda). Estas instrucciones se llaman IMUL (*Integer MULtiply*), todas sirven para multiplicar números con signo o sin signo. Tienen 3 sintaxis:

```
imul   reg, const
imul   reg, oper, const
imul   reg, oper
```

Donde *reg* es un registro, *const* una constante y *oper* un registro o una posición de memoria. Los operandos pueden ser de 16 ó de 32 bits. Las de dos operandos multiplican el uno por el otro y dejan el resultado en el registro. La de tres operandos multiplica los dos últimos y deja el resultado en el registro.

El segundo tipo de instrucciones de multiplicación son una herencia del 8086. En primer lugar hay dos: MUL e IMUL; la primera sirve para multiplicar números sin signo; la segunda, para números con signo. En cualquier caso, tienen un solo operando, que puede ser una posición de memoria o un registro, y puede tener 8, 16 ó 32 bits:

```
mul   oper
```

El otro operando es implícito, y depende de cuántos bits tiene *oper*:

- `ax ← al * oper8`
- `dx:ax ← ax * oper16`
- `edx:eax ← eax * oper32`

Donde la notación *reg1*`:`*reg2* denota la concatenación de los dos registros.[4] Note que el uso de `ax`, `dx`, etc. no es un ejemplo: se usan exactamente esos registros.

Nada lo obliga a utilizar la parte del resultado que va en `dx` o en `edx`: si sabe que el resultado cabe en `ax` o en `eax` —según sea el caso—, puede despreciar la parte alta. No olvide que, aun en ese caso, la instrucción afecta a `dx` o `edx` —así sea poniéndolos en cero—.

La división es el complemento de la instrucción de multiplicación que acabamos de describir. Hay dos: DIV e IDIV, que sirven para dividir números sin signo y con

---

[3] Excepto inmediato, que no tiene sentido.

[4] Note que la representación del producto de dos números de *n* bits puede requerir 2**n* bits, por esto es necesario que el resultado tenga el doble de tamaño que los operandos.

signo, respectivamente. Como la multiplicación, tienen un solo operando —posición de memoria o registro—:

```
div  oper
```

El otro operando es implícito, y depende de cuántos bits tiene *oper*:

- `al ← ax / oper8;  ah ← ax mod oper8`
- `ax ← dx:ax / oper16;  dx ← dx:ax mod oper16`
- `eax ← edx:eax / oper32;  edx ← edx:eax mod oper32`

Note que la división, a diferencia de las instrucciones anteriores, retorna dos resultados: el cociente y el residuo.

Por supuesto, además de las descritas, existen otras instrucciones útiles, como DEC e INC, pero no presentan mayores dificultades y no las trataremos aquí; consultar un manual para ver su operación.

Instrucciones lógicas

Las instrucciones para el manejo de expresiones lógicas son las usuales: AND, OR, XOR (o-excluyente) y NOT.

En su funcionamiento, son parecidas a las aritméticas: todas (excepto NOT) tienen dos operandos. Como es usual, operan el primero con el segundo y guardan el resultado en el primero. En cuanto a NOT, invierte los bits de su operando dejando el resultado en el mismo operando.

Es importante resaltar que todas estas operaciones se efectúan bit a bit, es decir, equivalen a los operadores de C: &, |, ^ y ~ (no equivalen a &&, || y !).

Instrucciones de corrimiento

Estas instrucciones son un poco menos usuales que las vistas hasta el momento; sin embargo, algunas de ellas existen en C: >> y << (en Java, además, existe >>>).

Hay de dos tipos: corrimientos y las rotaciones; y se pueden hacer en dos direcciones: derecha e izquierda. Además, los corrimientos pueden ser lógicos o aritméticos y las rotaciones pueden ser simples o a través del bit de acarreo.

En resumen, tenemos:

Nombre	Sigla	Acción
SHift Right	SHR	Corrimiento a la derecha
SHift Left	SHL	Corrimiento a la izquierda
Shift Arithmetic Right	SAR	Corrimiento aritmético a la derecha
ROtate Right	ROR	Rotación a la derecha
ROtate Left	ROL	Rotación a la izquierda
Rotate through Carry Right	RCR	Rotación a la derecha a través del carry
Rotate through Carry Left	RCL	Rotación a la izquierda a través del carry

Todos tienen dos operandos: el primero es el que se quiere correr o rotar, el segundo indica cuántos bits se desea desplazarlo. El resultado se guarda en el primer operando.

El primer operando se puede indicar con cualquier modo de direccionamiento (excepto inmediato, no tendría sentido). El segundo puede ser una constante o el registro `cl`. Así, si queremos correr el registro `ax` dos posiciones a la derecha, escribimos:

```
shr  ax, 2
```

Pero, si queremos hacer un corrimiento de `n` posiciones, donde `n` es una variable, a lo apuntado por `esi`, debemos escribir:

```
mov  cl, n
shr  [esi], cl
```

El último bit en salir se guarda en el bit de acarreo. En el caso de los corrimientos, entran ceros por el otro extremo para remplazar los bits perdidos. En el caso de las rotaciones, el bit que sale por un extremo vuelve a entrar por el otro.

Las rotaciones a través del bit de acarreo son como las rotaciones simples, pero tratando el bit de acarreo como si fuera parte del registro.

En cuanto al corrimiento aritmético a la derecha, en lugar de recibir ceros por el extremo izquierdo, el bit de signo se va replicando con cada corrimiento. Es decir, si tenemos 10001000, y le hacemos un corrimiento aritmético a la derecha, obtenemos 11000100.

La fig. 5.2 muestra una síntesis de estas instrucciones.

5.4 Evaluación de expresiones

Toda expresión tiene un orden de evaluación que viene dado por los paréntesis y la prioridad de los operadores (ver fig. 5.3). En ensamblador es necesario evaluar expresión por expresión siguiendo el orden indicado; los resultados intermedios se pueden almacenar en registros o en variables temporales en memoria. En lo que sigue mostramos cómo se hace esta evaluación.

Los recuadros de la figura 5.3 denotan un orden de evaluación: cuánto más claros, más pronto deben ser evaluados, porque otras expresiones dependen de ellos. Note que puede haber diversos ordenes de evaluación; por ejemplo, en la figura 5.3,

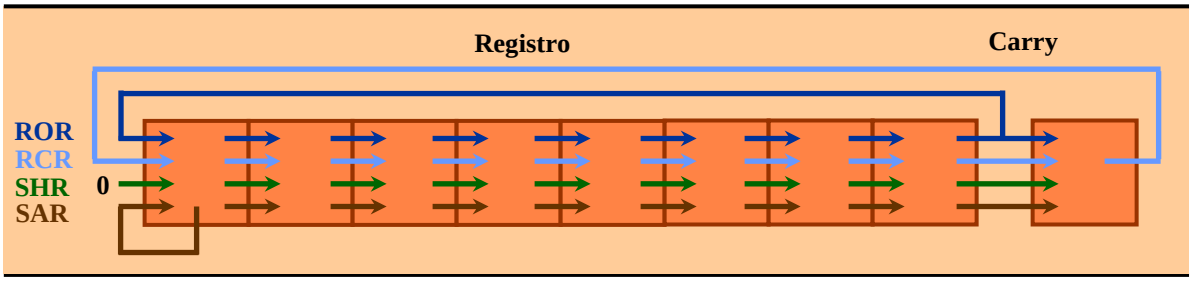


Fig. 5.2. Rotaciones y corrimientos

podemos empezar por `a*b` o por `e/2`. Una posible evaluación es:

Código C	Código Ensamblador
`a*b`	`mov ebx, a` `imul ebx, b`
`a*b + c`	`add ebx, c`
`a*b + c + 3`	`add ebx, 3`

`e/2`	`mov eax, e`
	`cdq`
	`mov ecx, 2`
	`idiv ecx`

`d - e/2`	`mov edx, d` `sub edx, eax`
`(a*b + c + 3)*(d - e/2)`	`imul ebx, edx`
`-((a*b + c + 3)*(d - e/2))`	`neg ebx`

Algunos comentarios: Note que después de evaluar (`a*b + c + 3`), el resultado queda en `ebx`, lo cual implica que este registro no se puede usar en lo subsiguiente; estamos usando ebx como una variable temporal. En la medida en que la expresión se vuelva más complicada, se necesitarán más temporales, lo cual irá conduciendo a que se agoten los registros.

Una estrategia para minimizar el impacto de esto consiste en evaluar primero las expresiones más complicadas, de esta manera se tienen más registros disponibles cuando se están evaluando las expresiones que requieren más temporales. Si aun así no es suficiente, será necesario guardar algunos registros en variables temporales en memoria.

La división `e/2` luce más extraña; la situación es la siguiente: la división de 32 bits supone que el dividendo está en `edx:eax`; el programa carga `e` en `eax`, pero es necesario expandir este número para que ocupe la pareja de registros `edx:eax`; la instrucción `cdq` — *Convert Doubleword to Quadword*— hace exactamente eso.

Note que en el momento de hacer la división ya tenemos ocupados cuatro registros: `eax`, `ebx`, `ecx` y `edx`; es decir, para esta expresión, que no es demasiado complicada, alcanzamos a ocupar la mitad de los registros disponibles.

En ocasiones, analizando el código es posible disminuir el número de registros; por, ejemplo, en nuestro caso, podemos evaluar la resta `d-e/2` de la siguiente manera:

```
neg  eax
add  eax, d
```

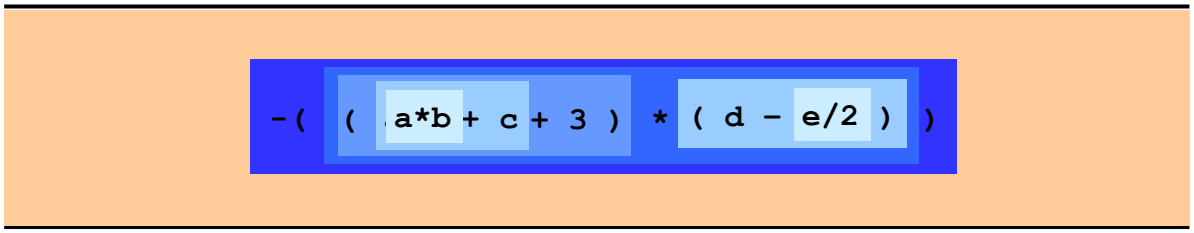


Fig. 5.3. Orden de evaluación

Así no tenemos que usar el registro `edx`.

En cuanto a la división, podemos recurrir a un truco que usa menos registros y es más eficiente en ejecución:

```
mov  eax, e
sar  eax, 1
```

En principio, un corrimiento a la derecha equivale a dividir por 2. Aunque este tipo de trucos se suele usar, es necesario hacerlo con cuidado; por ejemplo, en este caso, el corrimiento a la derecha no es completamente equivalente a la división. La situación es así: los corrimientos se comportan como la división euclidiana; en consecuencia, -1/2 = -1 y -3/2 = -2, etc. Es decir, cuando el dividendo es negativo, redondea al entero negativo inferior; en cambio, en la misma situación, la división de los lenguajes de alto nivel redondea al entero superior: -1/2 = 0 y -3/2 = -1, etc.

Expresiones y asignaciones

En principio, las expresiones retornan un valor pero no modifican variables. Sin embargo en C se presentan dos circunstancias:

- Hay operadores que retornan un valor pero también modifican una variable.
- Hay asignaciones que efectúan una operación.

En realidad, en C, podemos reducir todo a la primera afirmación: para C la asignación es un operador que modifica la variable a la izquierda y retorna como valor la variable a la derecha. Por esto en C se puede escribir:

```
a = b = 1;
```

En esta expresión `a` y `b` quedan valiendo 1.

Sin embargo, por claridad, trataremos las dos situaciones por aparte.

Operadores con efecto de borde

Los operadores de C ++ y --, además de retornar un valor, afectan una variable. Adicionalmente, la semántica de estos operadores depende de si están antes o después de la variable: no es lo mismo `++i` que `i++`. En los dos casos la variable se incrementa en 1, pero, la primera expresión retorna el valor de la variable incrementada, en tanto que la segunda retorna el valor antes de incrementar:

Código C	Código Ensamblador

`char  x;`
`char  * p;`
`x = *p++;`
	`mov  ebx, p`
	`mov  dh, [ebx]`
	`mov  x, dh`
	`inc  ebx`
	`mov  p, ebx`

`x = *++p;`	`mov  ebx, p`
	`inc  ebx`
	`mov  dh, [ebx]`
	`mov  x, dh`
	`mov  p, ebx`

Note que el código es esencialmente el mismo excepto por la posición del incremento.

Operadores de asignación

En C existen múltiples operadores del estilo +=. Son operadores que, más que asignar, acumulan; es decir, almacenan un valor en una variable pero efectuando una operación con el valor que ya se encuentra en ella. A continuación mostramos un ejemplo:

Código C	Código Ensamblador
`int  x, y;` `x += y;`	`mov  ebx, y` `add  x, ebx`

Expresiones y direccionamiento

En las expresiones estudiadas hasta ahora siempre operamos sobre datos; las expresiones también operan sobre direcciones.[5]

Tenemos expresiones como: `v[i]`, `*p`, `p++`, `p+i`, `p-q`, `&x`, `s.x`, `p->x`; todas las cuales son alguna forma de operación sobre una dirección. A continuación trataremos algunas particularidades relacionadas con estas operaciones.

Apuntadores

Los apuntadores son variables de tipo dirección; en principio, su contenido es un número que corresponde a alguna dirección de la memoria. Como mencionamos anteriormente, cuando usamos el apuntador (`p`), estamos hablando del contenido del apuntador —la dirección en sí—; cuando usamos la indirección (`*p`), estamos hablando de la entidad apuntada. En consecuencia, `p` y `*p` tienen tipos de datos diferentes: si `p` es un apuntador a un entero, `*p` es un entero. Ahora, *p no solo es un valor entero, sino que corresponde a una variable en memoria; por ende puede ser usado en contextos donde solo se pueden usar variables —por ejemplo, `(*p)++`, o aparecer a la izquierda en una asignación—. A continuación presentamos algunos ejemplos de traducción:

Código C	Código Ensamblador

`char  x;`
`char  * p;`
`*p = x;`
	`mov  ebx, p`
	`mov  dh, x`
	`mov  [ebx], dh`

`(*p)++;`	`mov  ebx, p` `inc  [ebx]`

`*(p++) = x;`	`mov  ebx, p`
	`mov  dh, x`
	`mov  [ebx], dh`
	`inc  ebx`
	`mov  p, ebx`

[5] También se puede ver como que las direcciones son un tipo de datos que tiene sus particularidades propias diferentes de los enteros, caracteres, etc.

Note la diferencia entre las dos últimas asignaciones: en la tercera, a lo apuntado se le está asignando `x`, y, posteriormente, se incrementa el apuntador;[6] en la segunda, solo se incrementa lo apuntado, y no se modifica el apuntador.

Con respecto a los incrementos en los apuntadores, en los lenguajes de alto nivel, el programador no se preocupa por el tamaño de los tipos de datos; no así en ensamblador, donde el tamaño de las entidades se vuelve un aspecto relevante en la programación.

En concreto, en C escribimos `p++` sin preocuparnos del tipo de datos apuntado por `p`. Los programadores en ensamblador, o los compiladores al generar código, sí deben ocuparse de este aspecto. En efecto, normalmente, si se incrementa el apuntador, es porque está apuntando a un vector y queremos ponerlo a apuntar al siguiente elemento, luego debe incrementarse en lo que ocupe el elemento en memoria; así, un apuntador a entero debe incrementarse de 4 en 4 porque un entero ocupa 4 bytes (ver fig. 5.4). Por ejemplo:

Código C	Código Ensamblador

`char * p;`
`short int * q;`
`int  * r;`
`p++; q++ ; r++;`
	`add  p, 1`
	`add  q, 2`
	`add  r, 4`

Lo dicho sobre el incremento es aplicable en general a las operaciones aritméticas con apuntadores.

- `p+i`: debe multiplicarse la `i` por el tamaño del elemento apuntado y esto se suma al contenido de `p`.
- `p-q`: debe dar como resultado el número de elementos que hay entre `p` y `q`; en consecuencia, corresponde a la resta de `p` y `q` dividida por el tamaño del elemento apuntado.

Vectores

C es un lenguaje muy basado en apuntadores, por lo cual la notación vectorial puede ser considerada "azúcar sintáctico": el programador se siente cómodo con ella, pero, en realidad, en el fondo son los mismos apuntadores.

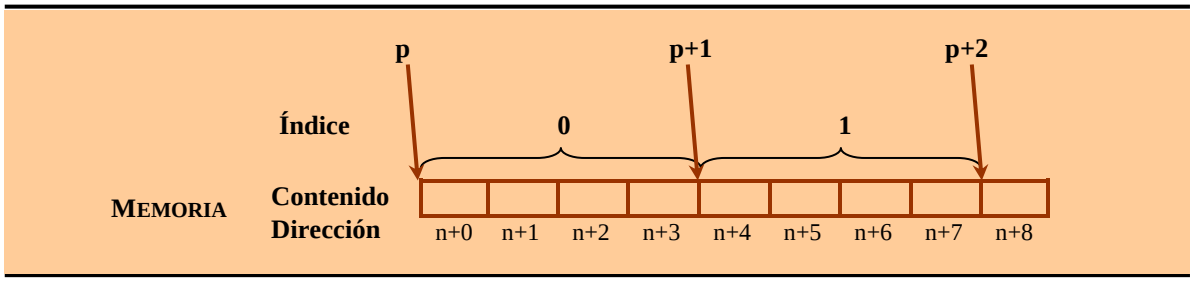


Fig. 5.4. Relación entre incrementos de apuntadores y desplazamientos

[6] Como nota al margen, debido a la prioridad de operadores en C, `*(p++)` es lo mismo que `*p++`.

Esto va a tal punto que cuando el compilador encuentra expresiones del tipo:

$expr_1$[$expr_2$]

lo primero que hace es traducirlas a:

*($expr_1$+$expr_2$)

Es decir, la notación vectorial se reduce a notación de apuntadores; en consecuencia, aplica lo dicho en la sección anterior.

Sin embargo, hay un aspecto que se debe tener en mente: `v[i]` equivale a `(v+i)`, pero, a diferencia de `(p+i)`, `v` es una constante de tipo apuntador y, por ende, no se le puede asignar nada. Dicho en otros términos, cuando se declara un apuntador `p`, efectivamente se está declarando una variable que puede contener direcciones; cuando se declara un vector `v`, se separa espacio en memoria para los elementos que contiene, pero `v`, como tal, es solo la dirección inicial del vector. Complementariamente, cuando se declara:

```
int * p;
```

la variable `p` existe, no así el entero `*p`; es necesario asignarle algo a `p`, de lo contrario, `*p` es inválido (no apunta a nada).

Estructuras

Las estructuras no constituyen una nueva forma de direccionamiento; se pueden manejar con los modos vistos hasta ahora. Note que las estructuras son las mismas variables de siempre, solo que se declaran varias agrupadas bajo un nombre; en consecuencia, su manejo no difiere mucho del de otras variables, excepto porque se direccionan con respecto a una dirección base. Por ejemplo, si tenemos la estructura:

```
struct {
  int    a;
  short  b;
  char   c;
} s;
```

La estructura como un todo se llama `s`, lo cual corresponderá a una cierta dirección en memoria. La variable a se encuentra `a` un desplazamiento de cero de esa dirección; la variable `b`, a un desplazamiento de 4 —`a` es un `int` de 4 bytes— y la `c`, a 6 bytes:

Código C	Código Ensamblador

`s.a = 1;`
`s.b = 2;`
`s.c = 'c';`
	`mov  s, 1`
	`mov  s+4, 2`
	`mov  s+6, 'c'`

Veamos la misma situación pero con un apuntador a `s`:[7]

[7] Tenga presente que `p->x` es lo mismo que `(*p).x`.

Código C	Código Ensamblador

p->a = 1;
p->b = 2;
p->c = 'c';
	mov ecx, p
	mov [ecx], 1
	mov [ecx+4], 2
	mov [ecx+6], 'c'

Se usa el direccionamiento indexado, con la dirección base en el registro y el desplazamiento en la constante.

El código antes presentado corresponde a lo que realmente se genera en lenguaje de máquina. Sin embargo, el ensamblador dispone de una notación más cómoda; se debe tener presente que es solo una notación, y no corresponde a ningún nuevo modo de direccionamiento. La realidad sigue siendo la antes descrita.

El acceso a los campos se puede hacer por medio de la notación:

```
mov  eax, s1.a
```

Se puede combinar con otras notaciones del ensamblador:

```
mov  bx, v[esi].b
mov  [ecx].c, 'z'
```

En los dos últimos casos se presenta una ambigüedad; si hubiera varias estructuras diferentes con un campo llamado *c*, ¿cómo podría el ensamblador determinar cuál de ellas se está utilizando? Es necesario que el programador especifique a qué estructura se está refiriendo, para lo cual existe la sintaxis:

```
mov  (S ptr [ecx]).c, 'z'
```

Recuerde que esta notación es solo una comodidad, en el fondo, la instrucción anterior sigue siendo: `mov [ecx+6], 'z'`.

Expresiones lógicas

En algunos lenguajes TRUE se representa con 1, y FALSE con cero —o alguna convención similar—. En dichos lenguajes, las expresiones lógicas se pueden evaluar usando las instrucciones lógicas de la máquina de manera parecida a como se calculan las expresiones aritméticas.

Ahora bien, la situación es menos clara en un lenguaje como C, puesto que este define falso como cero y cierto como cualquier cosa diferente de cero. En el ejemplo anterior, si A vale 1 y B vale 2, el *and* bit a bit daría cero; es decir, falso. Sin embargo, desde el punto de vistas de C, los dos son ciertos, luego el *and* debería dar cierto. Esto implica que las condiciones lógicas de C no se pueden implementar usando estas instrucciones; en C se calculan usando el flujo de control, aspecto que estudiaremos más adelante.

Las expresiones y el registro de indicadores

Como mencionamos anteriormente, al ejecutar una instrucción, los indicadores pueden ser afectados. Cómo son afectados depende en parte de la instrucción. En general, las operaciones (ADD, SUB, SHR, etc.) afectan los indicadores S y Z como es

de esperarse: si el resultado es cero, `Z` = 1; si no, `Z` = 0. Si es negativo, `S` = 1; de lo contrario, `S` = 0.

No todas las instrucciones afectan estos indicadores; en particular, las rotaciones no afectan ni Z, ni S. La instrucción `MOV` no afecta ningún indicador en absoluto. Los demás indicadores no son afectados de una manera estándar y lo mejor es consultar un manual en caso de duda.

Veamos el caso particular del indicador de acarreo: las instrucciones de corrimiento —`SHL`, `ROR`, etc.— lo afectan con el bit que expulsan fuera del operando. La adición, `ADD`, lo afecta con el acarreo que produce la suma. La resta, `SUB`, es menos evidente; se podría pensar que basta con hallar el complemento a dos del substraendo y sumarlo al minuendo, con lo cual quedaría reducido al caso de la suma. Sin embargo, no es así, puesto que la resta pone en 1 el bit de acarreo si el minuendo es menor que el substraendo, y en cero si no —simboliza lo que "se tomó prestado"—.

5.5 Las declaraciones

Para terminar con las expresiones, vamos a estudiar cómo se declaran las variables en ensamblador. En las secciones siguientes, veremos 4 tipos de entidades que podemos declarar: escalares, vectores, estructuras y cadenas de caracteres; estas últimas van a ser un tipo derivado de los anteriores.

Empecemos por describir los identificadores: el primer carácter debe ser una letra, los siguientes pueden ser letras, dígitos o caracteres especiales como '$' y '_'. No se puede usar como identificadores las palabras que se han destinado a otros fines, tales como los nombres de las instrucciones (`MOV`, `ADD` etc.) y de los registros (`AX`, `SP`, etc.).

Escalares

La forma general de una declaración es:

Identificador *Tipo* *Valor inicial*

El *identificador* es el nombre de la variable y sigue las reglas antes explicadas. El *Tipo* dice cuántos bytes ocupa la variable en cuestión; vamos a utilizar tres tamaños,[8] representados por las palabras:

`BYTE`	1 byte
`WORD`	2 bytes
`DWORD`	4 bytes

El *valor inicial* nos permite suponer que, al comenzar la ejecución del programa, dicha variable ya tiene un valor asignado. La asignación solamente se hace una única vez antes de empezar la ejecución; una vez que esta comienza, el

[8] El ensamblador dispone de otros tamaños, pero nosotros solo usaremos estos.

programador es responsable de todo lo que ocurra o deje de ocurrir. Como ejemplos, tenemos:

```
posicionX       WORD       0
cambioLinea     BYTE       10
```

nos definen una variable de 16 bits, con valor inicial 0, llamada `posicionX`; y una variable de 8 bits, con valor inicial 10, llamada `cambioLinea`. El campo de valor inicial puede ser una interrogación (`?`), lo cual indica que la variable no tiene un valor inicial (no sabemos qué contiene cuando empieza la ejecución).

Al tener declaraciones, el ensamblador puede verificar que las variables sean bien utilizadas. Por ejemplo, si se tiene:

```
a     WORD       0
mov   al,  a
```

el ensamblador avisará que la variable está siendo mal utilizada (porque el registro no es de tipo palabra). Además, las declaraciones ayudan a resolver ciertas ambigüedades; este es el caso de las instrucciones que tienen un solo operando como `MUL a`: sin declaraciones, no sería posible saber si es una multiplicación de 32, 16 o de 8 bits.

Este mecanismo no es suficiente en ciertos casos v.g.:

```
mul   [esi]
```

Aquí persiste la ambigüedad, ya que no sabemos si `esi` apunta a un dato de 8, 16 ó 32 bits. Para solucionarla, se usa la siguiente notación: se le antepone al operando el tipo de datos de la entidad seguido de la palabra `ptr`:

```
mul   byte ptr [esi]
mul   word ptr [esi]
```

Esta construcción también nos permite cambiarle el tipo a una variable; por ejemplo, tenemos una variable `a` definida con `WORD`, y, por alguna razón, queremos cargar únicamente su parte baja, podemos escribir:

```
mov   al, byte ptr a
```

En este caso, la construcción `byte ptr` quiere decir "suponga que la variable no es de tipo palabra sino de tipo byte". Tenga en cuenta que este mecanismo no transforma de ninguna manera el contenido de la variable; sencillamente es una forma de especificar cuántos bytes se deben traer a partir de la dirección de la variable.

Vectores

Para definir un vector es necesario separar espacio para todos los elementos que lo componen. Esto se logra por medio de la construcción:

Identificador *tipo* *n* `DUP (` *c* `)`

Esta construcción reserva espacio para *n* elementos del tipo dado. Todos los elementos son inicializados con la constante *c*. Tenemos, por ejemplo:

```
vectorCaracteres    byte      5 dup (' ')
vectorNumeros       word      15 dup (0)
matriz              word      10 dup ( 5 dup (?) )
```

La variable `vectorCaracteres` define un vector de bytes con 5 elementos, dichos elementos tienen un valor inicial de blanco (' '). La segunda variable, `vectorNumeros`, define un vector de 15 posiciones, cada una es una palabra y todas valen 0 inicialmente. La última variable define 50 dobles palabras cuyo valor inicial no está especificado.

La inicialización también puede ser elemento por elemento:

```
vectorCaracteres    byte      'a', 'b', 'c', 'd', 'e'
```

esta última declaración es completamente equivalente a:

```
vectorCaracteres    byte      "abcde"
```

Atención: no es una cadena de caracteres; las cadenas de caracteres no son un tipo primitivo en la IA32, sino que toca definirlas (como se verá en una próxima sección). Esta es sencillamente una notación para no tener que escribir los caracteres uno por uno.

Estructuras

Este tipo de datos es el equivalente a las estructuras de lenguajes como C. También sirven como base para implementar los objetos. Las declaraciones son de la forma:

```
S     struct
  a   word   0
  b   dword  ?
  c   byte   ?
S     ends
```

La anterior es una estructura, llamada `S`, con tres campos: el primero de tipo palabra —inicializado en 0—, el segundo de tipo doble palabra y el último de tipo byte. Esta declaración no separa espacio en memoria, sólo define una estructura en abstracto —similar a la declaración de una estructura en C o de una clase en C++ o Java—. Las inicializaciones son los valores por defecto que van a tomar las entidades declaradas de ese tipo.

Para separar efectivamente espacio en memoria, se hacen declaraciones de la forma:

Identificador *Tipo* **{** Valor, ... , Valor **}**

Donde *Identificador* es el nombre de la variable. *Tipo* debe haber sido declarado como `struct`. La lista de valores sirve para inicializar los campos de la estructura; si no se pone un valor quiere decir que se debe tomar el valor por defecto —el valor que se puso en la definición de `struct` —.

Algunos ejemplos de declaración:

```
s1    S   {,,}
vS    S   10 dup ( {,1,} )
```

```
ss    S  {1,2,'a'}, {,1,}, {1,,'b'}
```

Cadenas de caracteres

Las cadenas de caracteres no son un tipo primitivo de datos en la IA32; el software debe definir una representación. Hay diversas convenciones; aquí usaremos la de C.

La convención de C es la siguiente: una cadena de caracteres es una secuencia de bytes que termina en el carácter nulo (ASCII NUL, 00H). Por ejemplo, la cadena "HOLA" quedaría:

Byte	0	1	2	3	4
	48H	4FH	4CH	41H	00H

Por esta razón las cadenas de caracteres en C acaban en "\0": es una convención de C para denotar el carácter cuyo código ASCII es 0.

Para declarar esta misma cadena en ensamblador, haríamos lo siguiente:

```
c     byte  'H', 'O', 'L', 'A', 0
```

Equivalentemente, y más cómodo:

```
c     byte  "HOLA", 0
```

Esta cadena se maneja como cualquier vector: con apuntadores o indexamiento.

Con las cadenas de caracteres es importante diferenciar entre tres conceptos: una cosa es el espacio total asignado para la cadena, otra es cuánto ocupa efectivamente y, por último, cuántos caracteres tiene; por ejemplo:

```
c     byte  "HOLA", 0, 20 dup (?)
```

- La cadena tiene los 4 caracteres de "HOLA".
- El espacio total ocupado son 5 bytes: los anteriores más el carácter nulo.
- El espacio asignado para la cadena son 25 bytes: los 5 ya mencionados y 20 que no se están utilizando (todavía).

Procesamiento de las declaraciones

Las declaraciones de variables son solo una comodidad del ensamblador; la máquina, como tal, solo trabaja con direcciones. Dicho de otra manera: en un programa ya traducido a lenguaje de máquina no aparecen los identificadores.

El procesamiento básico que hace el ensamblador es el siguiente: cuando encuentra una declaración, calcula cuánto espacio necesita para la entidad y le asigna un sitio en la memoria del tamaño adecuado. La dirección inicial de esa zona es la dirección asignada a la variable. Posteriormente, cuando encuentra el mismo identificador en una instrucción, lo reemplaza por la dirección asignada.

En el caso de las estructuras, cuando el ensamblador encuentra una declaración, anota los nombres de los campos junto con su desplazamiento con respecto al inicio de la estructura. Cuando el ensamblador encuentra una instrucción que usa alguno de los campos, reemplaza el campo por su desplazamiento correspondiente.

Generación de apuntadores

Las declaraciones son útiles para el programador, pero introducen un inconveniente: puesto que el ensamblador asigna las direcciones, el programador no sabe dónde quedan localizadas las variables.

Los apuntadores son direcciones de memoria, y las direcciones son números; en consecuencia, si se conoce la dirección, es fácil crear un apuntador: basta con asignar la dirección a un registro. Pero no conocemos las direcciones debido a que el ensamblador se encarga de la traducción de las etiquetas a números.

El ensamblador provee un mecanismo para que se puedan recuperar las direcciones —lo cual sirve permite generar apuntadores—; se trata del operador `offset`:

```
mov  eax, offset x
```

`eax` queda apuntando a `x`. Este es un operador del ensamblador, no de la máquina. El ensamblador se encarga de reemplazar la expresión `offset x` con la dirección de `x`. Esto implica, en particular, que el `mov` anterior es de constante a registro: el ensamblador reemplaza `offset x` con la dirección de `x`, la cual es una constante.

Hay otro mecanismo de generar apuntadores, este sí basado en una instrucción de máquina: la instrucción `lea` (—*Load Effective Address*—). Esta instrucción tiene el mismo formato que un `mov`, pero en lugar de cargar en el operando destino el valor del operando fuente, carga su dirección. Así, la instrucción:

```
lea  eax, x
```

también deja `eax` apuntando a `x`.

Los dos mecanismos no son completamente equivalentes: `offset` sirve para valores estáticos, mientras que `lea` sirve para valores dinámicos. Por ejemplo, el siguiente apuntador no se puede crear con `offset`:

```
lea  eax, v[esi]
```

Cuando se está ensamblando el programa, no se sabe el valor de `esi` —de hecho, puede cambiar dinámicamente en ejecución—, luego no se puede saber cuál es la dirección de `v[esi]`; esto solo se puede determinar en ejecución. Complementariamente, la siguiente declaración solo se puede hacer con `offset`:

```
p    dword    offset x
```

`p` es una variable que queda inicializada con un apuntador a `x`.

EJERCICIOS

1- Programación mixta. Escriba un programa en C que declara una variable de tipo `short int` y una de tipo `char`; después escriba el siguiente código en línea en ensamblador:

a- Asigne los siguientes números a la variable de tipo `short int` e imprímalos en hexadecimal en la pantalla (esta última acción en C). Compare los resultados.

A5H, 165, 10101001B, 10, -10, 0, -0, 32767, -32768, 65535.

b- Asigne los siguientes números a la variable de tipo `char` e imprímalos en hexadecimal en la pantalla (esta última acción en C). Compare los resultados.

41H, 'A', 65, 01000001B, 127, 128, 255, -128, -1, 0.

c- Asigne las siguientes parejas de números a `ah` y `al`, respectivamente, después asigne `ax` a la variable de tipo `short int` e imprímalos en hexadecimal en la pantalla (esta última acción en C).

15 y 1, 'A' y 'B', 41H y 42H, -1 y -1.

2- Indique cuáles de los siguientes direccionamientos son correctos:

Instrucción	¿Correcto?
`mov esi, [esi+1]`	
`mov esi, esi+1`	
`sub [esi], [edi + ebx]`	
`add esi, [edi + edx]`	
`mov ax, var + 1`	
`mov esi, [esi + 2] - esi`	
`or ebx, [ebp - 2]`	
`mov [esi], 4`	
`mov 4, [esi]`	
`mul [esi + ebx + ebp]`	
`and ax, [esi - ebx]`	

3- En un programa se tienen las siguientes declaraciones:

```
v     word      3815H, A3B5H, 26F7H, 4855H, 0666H
b     byte      10, 20, 30, 40, "CABE"
```

a- ¿Qué queda en `ax` después de ejecutar cada uno de los siguientes grupos de instrucciones?

Código	ax =
`mov esi, 4` `mov ax, v[esi]`	
`mov esi, 3` `mov ax, v[esi]`	
`mov esi, 10` `mov ax, v[esi]`	

b- ¿Qué queda en AL después de ejecutar cada uno de los siguientes grupos de instrucciones?

Código	al =
`mov esi, 0` `mov al, b[esi]`	
`mov esi, 5` `mov al, b[esi]`	
`mov esi, 8` `mov al, b[esi]`	

4- En los ejercicios que siguen, se da un segmento de código en C y 4 segmentos de código en ensamblador; diga cuál de ellos es equivalente al segmento en C (si varios lo son, diga cuál es mejor y por qué).

a-
```
int a, b;
a = b;
```

1.	2.	4.	3.
`mov a, b`	`mov eax, b` `mov a, eax`	`mov ax, b` `mov a, ax`	`mov b, eax` `mov eax, a`

b-
```
char a, v[5]; int i;
a = v[i];
```

1.	2.	4.	3.

`mov eax, v[i]`
`mov a, eax`
	`mov ch, i`
	`mov ah, v[ch]`
	`mov a, ah`
		`mov ecx, i`
		`mov eax,v[ecx]`
		`mov a, eax`
			`mov ecx, i`
			`mov ah, v[ecx]`
			`mov a, ah`

c-
```
int a, i, v[5];
a = v[i];
```

1.	2.	4.	3.

`mov eax, v[i]`
`mov a, eax`
	`mov ecx, i`
	`mov eax,v[4*ecx]`
	`mov a, eax`
		`mov ecx, i`
		`mov eax, v[ecx]`
		`mov a, eax`
			`mov eax, v[4*i]`
			`mov a, eax`

d-
```
short a[4], b; short *p = a;
p++; b = *p;
```

1.	2.	4.	3.

`add p, 2`
`mov eax, p`
`mov ebx,[eax]`
`mov a, ebx`
	`add p, 2`
	`mov bx, [p]`
	`mov a, bx`
		`add p, 2`
		`mov eax, p`
		`mov bx, [eax]`
		`mov a, bx`
			`mov eax, p`
			`add eax, 2`
			`mov bx, [eax]`
			`mov a, bx`

e-
```
unsigned short a; int b;
b = a;
```

1.	2.	4.	3.

`mov edx, a`
`mov b, edx`
	`mov dx, a`
	`mov b, dx`
		`mov edx, 0`
		`mov dx, a`
		`mov b, edx`
			`mov b, 0`
			`mov dx, a`
			`mov b, dx`

5- Una forma de calcular el cuadrado de un número que se encuentra en `eax` es: `mul eax`

a- Si se sabe que en `eax` hay un natural ¿funciona este método?

b- Si en `eax` hay un entero ¿funciona?

6- Dadas las declaraciones mostradas, escriba en ensamblador la evaluación de las siguientes expresiones (trate de optimizar los cálculos):

```
int   a, b, c, d, e, f;
char  x, y;
```

Expresión

`(a + b)*(a + c)*(a + d)`
`(a + b)*(c - d) + (a + b)* (e + f)`
`(a + b)*(c - d) + (a + b)* (c + d)`
`a - - b`
`x + y`
`x - 'a'`
`x * y`
`x + a`

7- En el registro `al` se tiene un dígito- carácter ('0' a '9') codificado en ASCII. Calcule su valor numérico (0 a 9) y déjelo en `al` mismo.

8- Dadas las declaraciones mostradas, analice las siguientes instrucciones de C; diga si compilan, y, si es así, tradúzcalas a ensamblador.

```
int   a, b;
char  x;
char * p = &x, *q;
```

Expresión	¿Compila?
`a = b++;`	
`a = a++;`	
`p+1 = q + 1;`	
`*(p+1) = *(q+1);`	
`q = p++;`	
`*p++ = 5;`	
`(*p)++ = 'a';`	
`(*(++q))++;`	
`q = *p++;`	
`p++ = q;`	

9- Dadas las declaraciones mostradas, analice las siguientes instrucciones de C; diga si compilan, y, si es así, tradúzcalas a ensamblador.

```
int   a, i, j;
int   v[100];
int  * p, *q;
```

Expresión	¿Compila?
`v[6] = v[i];`	
`v[i] += a;`	
`p = v + i;`	
`v = p + i;`	
`*p = *(v + i);`	
`*v = *(p + i);`	
`p[i] = a;`	
`v[j] = a*(v[i] + v[j]);`	
`a = v[v[i]];`	
`p = 3*q;`	

10- Se tiene la siguiente declaración para un vector:

```
v   byte  100 dup (?)
```

a- Escriba código en ensamblador para generar un apuntador a `v[i]`; `i` es una variable entera (de tipo `dword`). El apuntador se debe dejar en `eax`.

b- Repita el problema anterior con un vector de dobles palabras (`dword`).

c- Repita el problema anterior con un vector de elementos de tamaño *t*.

11- Se tiene la siguiente declaración para una matriz:

```
m   byte  100 dup ( 200 dup (?) )
```

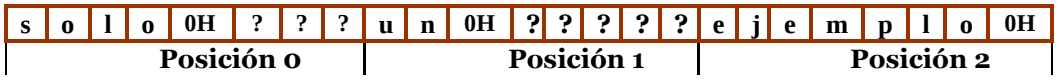
a- Escriba código en ensamblador para generar un apuntador a `m[i][j]`; `i` y `j` son variables enteras (de tipo `dword`). El apuntador se debe dejar en `eax`.

b- Repita el problema anterior con una matriz de dobles palabras (`dword`).

c- Repita el problema anterior con una matriz de elementos de tamaño *t*.

d- Se tiene una matriz representada en un vector de apuntadores a vectores. Repita las partes a- a c- con esta representación.

12- Se tienen cadena de caracteres representadas con convenciones de C.
a- Una forma de representar un vector de cadenas de caracteres es reservar un espacio máximo para cada cadena. Por ejemplo, el vector que se muestra a continuación, tiene tres cadenas cada una con 8 bytes disponibles (pero no necesariamente usados):

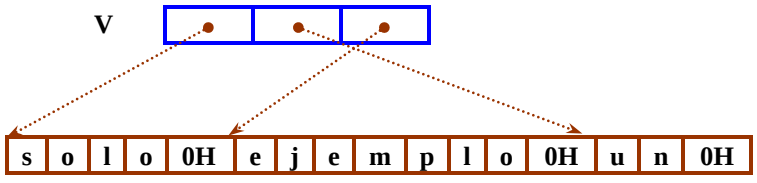


Las posiciones marcadas con "?" no están ocupadas.

- Se tiene un apuntador `p` al comienzo del vector, escriba código en ensamblador para que quede apuntando al *i*-ésimo elemento (`i` es una variable entera).

- Escriba código en ensamblador para poner un blanco en la posición `j` de la *i*-ésima cadena (`i` y `j` son variables enteras).

b- Otra forma de representar vectores de cadenas es por medio de un vector de apuntadores a las cadenas. En este caso, solo se reserva el mínimo espacio requerido por cada cadena:



Repita el punto anterior con esta representación.

13- Se tiene la declaración:

```
S     struct
  x   word   ?
```

```
   y  dword  ?
   z  byte   3 dup (?)
S     ends
```

Hay dos variables de tipo `S` llamadas `r` y `t`, y un vector de estructuras de tipo `S` llamado `vs`. Traduzca las siguientes asignaciones a ensamblador:

Instrucción
`a =r.x;`
`r.x = a;`
`r.y = t.y;`
`r.z[j] = c;`
`r = t;`
`a = vs[i].x;`
`vs[i].y = vs[j].y;`
`vs[i].z[j] = c;`
`vs[i] = vs[j];`
`p1 = &vs[i];`
`p2 = &vs[i].y;`
`p3 = &vs[i].z[j];`

14- Se dispone de una lista encadenada compuesta por estructuras de la forma:

```
Nodo  struct
   valor  byte   ?
   siguiente  dword  ?
Nodo  ends
```

`siguiente` es un apuntador al próximo elemento. La información del nodo está almacenada en `valor`. La variable `cabeza` tiene un apuntador al principio de la lista. Traduzca las siguientes asignaciones:

Instrucción
`a = (*cabeza).valor;`
`(*cabeza).valor = a;`
`cabeza = (*cabeza).siguiente;`

15- Escriba código en ensamblador para calcular (`eax mod 4`), sin hacer divisiones. Generalice a (`eax mod` 2^n).

16- Escriba un programa que recibe en `eax` un número y lo convierte, sobre `eax` mismo, en 1 ó 0 dependiendo de si es impar o par.

17- Suponiendo que `eax` vale 1 ó 0, escriba un programa que pone el valor de `eax` en el primer bit de `ebx` sin modificar los otros bits de `ebx`.

18- Una función lógica de tres variables tiene 8 casos posibles. Estos 8 casos se pueden codificar en un byte, por ejemplo en `dl`, de la siguiente manera $f(0,0,0)$ = bit_0 de `dl`, $f(0,0,1)$ = bit_1 de `dl`, etc. Suponiendo que `eax`, `ebx` y `ecx` valen 0 ó 1, calcule f(`eax`, `ebx`, `ecx`).

19- Se tiene un número en `eax` y se quiere activar los indicadores para saber si dicho número es cero, negativo etc. ¿Cómo se puede hacer esto sin dañar el valor que se encuentra en `eax`?

20- Como vimos anteriormente, en el código ASCII, los caracteres se almacenan en un byte pero sólo usan 7 bits. El 8º bit se puede utilizar para guardar un bit de paridad.

Suponiendo que se tiene un carácter en `al`, escriba un programa que calcula el bit de paridad para dicho carácter y lo almacena en el octavo bit de `al`.

Hay una instrucción, `lahf`, que copia algunos bits del registro de indicadores en `ah`: signo (`S`) en el bit 7, cero (`Z`) en el 6, acarreo auxiliar (`A`) en el 4, paridad (`P`) en el 2 y acarreo (`C`) en el bit 0 —los bits 5,3 y 1 de `ah` quedan indefinidos—.

21- Se tiene un vector de bits como el del ejercicio 4.9.

a- Escriba un programa en ensamblador para obtener el *i*-ésimo bit del vector. El valor del bit (1 ó 0) debe quedar en `eax`.

b- Escriba un programa para modifica el i-ésimo bit del vector sin modificar los otros. El programa dispone de la posición del bit que se quiere modificar en una variable entera, y, en otra, del valor que se quiere poner (1 ó 0).

22- Se quiere manejar números de 3 bytes. Los números se almacenan en variables del estilo:

```
n     byte  3 dup(?)
```

El formato de los números es *little endian* —byte menos significativo "a la izquierda"—.

a- Desarrolle código en ensamblador para leer un número de 3 bytes en `eax`.

b- Desarrolle código en ensamblador para escribir un número de 3 bytes que se encuentra en `eax` en una variable de 3 bytes en memoria.

23- Se tiene un subvector representado con las convenciones del ejercicio 4.35. Escriba un programa en ensamblador para obtener el i-ésimo elemento del subvector.

24- Se tiene un subvector representado con las convenciones del ejercicio 4.36. Escriba un programa en ensamblador para obtener el i-ésimo elemento del subvector.

25- Se tiene un vector disperso representado con las convenciones del ejercicio 4.37. Escriba un programa en ensamblador para obtener el i-ésimo elemento del vector.