6. El lenguaje ensamblador: control

La asignación y las expresiones son muy útiles, pero nos faltan las instrucciones para manejar el flujo de control del programa, que son fundamentales. Instrucciones tales como *si* o *mientras que* —*if* y *while* de C—.

En el lenguaje de la máquina, existen instrucciones comparables, sólo que bastante más rudimentarias. En esta sección, vamos a ver cómo se maneja el control en el lenguaje ensamblador.

6.1 Instrucción condicional

Empecemos por estudiar la construcción *si-entonces*:

```
if condición {
  Cuerpo del si
}
```

La esencia de esta construcción es: si la condición es falsa, se salta las instrucciones que siguen; si la condición es cierta, las ejecuta.

El equivalente, en lenguaje de máquina, son instrucciones de transferencia de control que evalúan una condición, y, si se cumple, se saltan los *n* bytes siguientes; si no se cumple, continúan con la próxima instrucción.

En términos más técnicos: se evalúa una condición, si se cumple, se incrementa el PC en *n*; si no, el PC sigue avanzando en su secuencia normal.

En ensamblador no se pueden evaluar condiciones complejas con una sola instrucción, sino que hay una instrucción por condición; por ejemplo, hay una instrucción para evaluar si el indicador de signo (*flag* `S`) vale 1; hay otra para evaluar si el indicador de cero (*flag* `Z`) está en 1, etc. Las condiciones de salto siempre están basadas, directa o indirectamente, en los indicadores. Estas instrucciones tienen un solo operando: el número de bytes que deben saltar.

Veamos un ejemplo utilizando la instrucción `jnz` (*Jump if Not Zero*), dicha instrucción quiere decir: salte si el indicador de cero es falso; esto es, salte si la

última operación efectuada en la ALU no dio cero. A continuación se muestra un programa en C y su traducción a lenguaje ensamblador:

Código C	Código Ensamblador
`if ( x-y == 0)` `  y = -y;`	`mov  eax, x` `sub  eax, y  ;calcular x - y.` `jnz  + 6  ;si x - y ≠ 0, saltar 4 bytes.` `  ;---- aquí llega si x - y = 0, entonces:` `  neg  y  ;y ← - y.` `  ;---- aquí llega si x - y ≠ 0, y continúa.`

En primero lugar se calcula la resta; si esta no da cero, el programa ejecuta el salto de 6 bytes. Dado que la instrucción "`neg y`" ocupa justamente 6 bytes, evitamos que se ejecute dicha instrucción. Pero, si la resta da cero, el salto no se efectúa, luego se ejecuta la instrucción `neg`.

Veamos un ejemplo de la construcción *si-entonces-sino*. Un programa para calcular el valor absoluto es:

```
if ( numero < 0 )
    valorAbsoluto = - numero;
  else valorAbsoluto = numero;
```

Para escribirlo en lenguaje ensamblador, necesitamos la instrucción `jns` (*Jump if Not Sign*), la cual hace un salto si el indicador `S` vale 0; es decir, si la última operación dio un resultado positivo.

```
mov  eax, numero
add  eax, 0    ;evaluar la condición.
jns  + 2       ; si es positivo, saltar.
  neg  eax     ;si es negativo, obtener el complemento.
               ;en cualquier caso, guardar resultado.
mov  valorAbsoluto, eax
```

Este ejemplo es más sutil que el anterior. En primer lugar, note la instrucción ADD: en apariencia es inútil puesto que estamos sumando 0; el objetivo de esta suma es modificar los indicadores para saber si el número es negativo o positivo. Puesto que `mov` no afecta los indicadores, recurrimos a este truco para modificarlos.

Si el número es positivo, se salta la instrucción de negación —la cual ocupa dos bytes—, y se almacena el número en la variable valorAbsoluto. Si el número es negativo, se efectúa la negación y se guarda el número usando la misma instrucción `mov` del caso anterior.

Observe que en el programa en C se escribió la asignación dos veces, mientras que en lenguaje máquina solamente se escribe una vez. La condición se utiliza únicamente para saber si hay que evaluar la negación o no. La construcción *si-entonces-sino* de C, se convierte en un *si-entonces* en lenguaje máquina.

En el primer ejemplo, dijimos que la instrucción `neg` ocupaba 6 bytes, en el segundo que ocupaba 2. Esto se debe al hecho de que, en el primer caso, la instrucción `neg` tiene un operando en memoria; en consecuencia, hay que codificar

la dirección del operando, lo cual consume 4 bytes. En el segundo caso, el operando es un registro; no se codifica una dirección, así que nos ahorramos 4 bytes.

Las instrucciones de salto son bastante incómodas. El solo hecho de tener que contar el número de bytes, es una complicación. Si a esto le agregamos problemas como el presentado en el párrafo anterior, la situación se pone lamentable.

Para simplificar la situación, se puede ampliar el lenguaje ensamblador. De la misma manera que bautizamos posiciones de memoria donde se almacenan variables, podemos bautizar posiciones de memoria donde hay instrucciones. En lugar de decir "salte *n* bytes", decimos "salte a la posición que se llama xxx". Con esta convención, el ejemplo anterior se transforma en:

```
mov  eax, numero     ; eax = numero.
add  eax, 0
jns  asignar         ; ¿numero ≥ 0?
                     ; numero < 0
  neg  eax           ; eax = - numero
asignar:             ; eax = = | numero |
mov  valorAbsoluto, eax  ; valorAbsoluto = | numero |
```

En este caso, la instrucción de salto dice: "si es positivo vaya a *asignar*", y en "*asignar*", guardamos el resultado. Dichos identificadores son llamados *etiquetas* (*label*, en inglés). Recuerde que esto es sólo una convención, tarde o temprano, por un método u otro, se tienen que traducir los nombres a números binarios; esta es una de las tareas del programa ensamblador.

Los nombres de posiciones siguen las mismas reglas que los nombres de variables, solo que se les ponen dos puntos (:) al final para diferenciarlos de las variables —cuando se declaran, cuando se usan se escriben sin los dos puntos—. En realidad, en ensamblador, no hay una gran diferencia entre nombres de variables y etiquetas de programas; los dos responden al mismo concepto: asignar un nombre simbólico a una posición de memoria.

Hay una gran variedad de instrucciones de salto, veremos únicamente las más importantes, dejándole el resto a su curiosidad.

La instrucción `jmp` (*JuMP*) es muy simple pero importante: no evalúa condiciones; sencillamente efectúa el salto, razón por la cual se llama *salto incondicional*.

Otros saltos se efectúan dependiendo del resultado de una comparación aritmética. Se hace una resta entre los números que desea comparar, después se utiliza la instrucción de salto para hacer efectiva la comparación. Por ejemplo, la instrucción `jl` (*Jump if Less*) ejecuta el salto si el primer operando es menor que el segundo:

```
;comparar eax con ebx.
sub  eax, ebx  ;la resta preliminar a la comparación
jl   menor     ; ¿eax < ebx?
     …         ; aquí llega si no es menor.
menor:         ; aquí llega si es menor
```

Se dispone de otras instrucciones tales como: `jg` (*Jump if Greater*), primer operando mayor que segundo; `jle` (*Jump if Less or Equal*), primer operando menor o igual que segundo, etc.

El prog. 6.1 es un ejemplo de uso de `jg`; en este caso, se trata de un programa que busca el máximo de dos números `A` y `B`.

En los saltos antes descritos se supone que los números están representados en complemento a dos. Si usted representa los números como enteros sin signo, dichos saltos no pueden ser utilizados. Para esto se dispone de otro juego de instrucciones de salto que realizan comparaciones únicamente para enteros sin signo: `ja` (*Jump if Above*), primer operando mayor que segundo; `jbe` (*Jump if Below or Equal*), primer operando menor o igual que segundo etc.

A manera de ejemplo, el siguiente código sirve para decidir si un carácter que está en `bl` es mayor o menor que la letra 'A'.

```
sub  bl, 'A'    ; 'A' es un entero sin signo.
ja   mayorQueA  ; comparación sin signo: ¿bl > A?
     …          ; el carácter es menor que 'A'.
mayorQueA:      ; el carácter es mayor que 'A'.
     …
```

Es importante retener que *greater* (`G`) y *less* (`L`) se utilizan en las instrucciones de comparación con signo, mientras que *above* (`A`) y *below* (`B`) se usan en las comparaciones sin signo.

Saltos e indicadores

Todas las instrucciones de salto están basadas en los indicadores, lo cual puede no ser evidente a primera vista. Por ejemplo, la instrucción `ja` ejecuta el salto únicamente si *Carry* = 0 y *Zero* = 0; recuerde que `sub` pone el bit de acarreo en 0 cuando el minuendo es mayor que el substraendo, como además tenemos que el resultado no fue cero, podemos afirmar que el primer operando es mayor que el segundo. El caso de `jl` es más delicado; en principio, si la resta da negativa, el

```
mov   eax, A
sub   eax, B
  jg  maximoA           ;¿A > B?
                        ;B > A
  mov  eax, B           ;eax = B. Es decir eax = max (A,B)
  jmp  guardarMaximo
maximoA:                ;A > B
  mov  eax, A           ;eax = A. Es decir eax = max (A,B)
guardarMaximo:          ;eax = max (A,B)
mov   maximo, eax       ;maximo = eax = max (A,B)
```

Prog. 6.1 Máximo de A y B

primer número es menor que el segundo. La dificultad estriba en que la resta puede producir un desbordamiento; por ejemplo, 40H - (-40H) da 80H, como el resultado es negativo, habría que concluir 40H < -40H, lo cual obviamente es falso.

En conclusión, si no hay desbordamiento, la interpretación es: si el resultado es negativo, el primer número es menor que el segundo; si el resultado es cero o positivo, el primero es mayor o igual que el segundo. Pero, si hay desbordamiento, el signo se debe interpretar al contrario: si el resultado es negativo, el primer número es mayor o igual que el segundo; si el resultado es cero o positivo, el primero es menor que el segundo. Lo cual se puede resumir en la ecuación:

Primero < segundo $\equiv \left(S \wedge \overline{O}\right) \vee \left(\overline{S} \wedge O\right)$

O, de manera equivalente, S $\oplus$ O.

Un tercer grupo de instrucciones de salto está constituido por operaciones que examinan directamente el estado de los indicadores. Ya conocemos algunas de estas, otras son: `jc` (*Jump if Carry*) y `jo` (*Jump if Overflow*).

Algunas instrucciones tienen seudónimos. Esto se hace por comodidad para el programador, para que él pueda elegir el nombre que considere más diciente. Se puede, por ejemplo, escribir `jng` (*Jump if Not Greater*) o `jle` (*Jump if Less or Equal*). Las dos siglas corresponden a la misma instrucción de la máquina. Un caso interesante es `jz`, que también se puede escribir `je` (*Jump if Equal*). Esto se hace porque la forma de verificar la condición "A == B" es:

```
sub  eax, ebx  ; A está en eax y B en ebx.
jz   AIgualB   ; si la resta da cero, A == B
```

Donde se verifica si A y B son iguales, según si la resta da cero o no. Sin embargo, es quizás más claro escribir:

```
sub  eax, ebx
je   AIgualB
```

La instrucción de máquina es la misma, pero es más natural preguntar si son iguales a preguntar si el resultado fue cero. Por el contrario, si se quisiera evaluar "A == 0", sería mejor usar `jz`.

Si-entonces-si-no

Quizás usted ya haya visualizado cuál es la estructura general del *si-entonces-si-no*; de no ser así, encuentra su esquema a continuación:

```
jx   entonces  ;Si la condición es cierta ir a entonces.
   …           ;Aquí van las instrucciones del si no.
jmp  finSi     ; No ejecutar entonces; estamos en si no.
entonces:
   …           ;Aquí van las instrucciones del entonces.
finSi:
```

Donde `Jx` denota algún salto que evalúa si la condición es cierta.

Indentamos las instrucciones por claridad. Todas están corridas a la derecha del salto donde se toma la decisión, siguiendo el esquema de los lenguajes de alto nivel. Es una buena práctica de programación, ya que facilita saber dónde acaba la instrucción condicional y cuáles son las instrucciones del *entonces* y el *si no*.

Es posible hacer un esquema complementario evaluando la negación de la condición:

```
j!x  si_no      ;Si la condición es falsa ir a si no.
   …            ;Aquí van las instrucciones del entonces.
jmp  finSi      ; No ejecutar si no; estamos en entonces.
si_no:
   …            ;Aquí van las instrucciones del si no.
finSi:
```

Donde `J!x` denota algún salto que evalúa si la condición es falsa.

Esta construcción es mejor ya que es compatible con el *si-entonces* —no hay *si no*—.

Instrucciones de comparación

Los saltos tienen un inconveniente: para tomar la decisión del salto, hay que examinar el estado de los indicadores, lo cual implica hacer una operación para modificarlos. Esto puede ser molesto en ocasiones, ya que, al hacer la resta, afectamos el valor de un registro que nos puede interesar conservar intacto. Para evitar esta situación, la IA32 incluye la instrucción `cmp` (*CoMPare*). Dicha instrucción resta los operandos —lo mismo que `sub`—, y modifica los indicadores, pero no almacena el resultado en ninguna parte; no modifica ningún registro.

El máximo de A y B, usando `cmp`, se encuentra en el prog. 6.2. Nos hemos ahorrado algunas instrucciones con respecto al primer ejemplo, gracias a que no se modifica `eax` en la comparación.

Es importante anotar que no sólo se utiliza la resta para examinar los indicadores, se pueden usar otras instrucciones. En tal caso, hay que ser cautelosos; por ejemplo, los saltos basados en comparaciones aritméticas (`jge`, etc.) deben ser precedidos por `sub` o `cmp`. En otros casos, podemos utilizar instrucciones diferentes, un ejemplo de esto es la condición "A = 0", la cual se puede verificar por medio de:

```
mov   eax, A
cmp   eax, B          ;eax conserva A sin modificar.
jg    guardarMaximo   ;¿A > B?
                      ;B > A
  mov  eax, B         ;eax = max (A,B)
guardarMaximo:        ;eax = max (A,B)
mov   maximo, eax     ;maximo = max (A,B)
```

Prog. 6.2 Máximo de A y B usando CMP

```
and  A, FFFFH  ; si el AND da cero, A vale cero.
jz   …
```

El `and`, como el `sub`, modifica el registro, por lo cual la IA32 dispone de una instrucción `test`; esta, de manera similar a `cmp`, efectúa un `and` bit a bit, modifica los indicadores, pero no almacena el resultado.

Cuando se hacen varias saltos en cascada, no es necesario repetir la operación —`sub`, `cmp`, `and`, etc.—, puesto que las instrucciones de salto no cambian el estado de los indicadores. Un ejemplo de esto es la función *signo*, que se muestra en el prog. 6.3. Como puede ver, todos los saltos se hacen sin necesidad de repetir el `cmp` inicial. La función signo se define así:

$$\text{Signo}(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

Evaluación de condiciones

Si la condición es más complicada, se puede lograr su evaluación de dos maneras: la primera es evaluarla como una expresión lógica y efectuar el salto según el resultado de la evaluación —*cierto* o *falso*—; este método no es muy eficiente porque implica evaluar toda la expresión lógica. La segunda forma es por flujo de control; método más eficiente ya que sólo evalúa el mínimo necesario.

La evaluación por flujo de control se basa en las siguientes ecuaciones lógicas:

$$\begin{aligned} \text{falso} &\wedge x = \text{falso} \\ \text{cierto} &\wedge x = x \\ \text{falso} &\vee x = x \\ \text{cierto} &\vee x = \text{cierto} \end{aligned}$$

```
mov   eax, x
cmp   eax, 0          ;se modifican los indicadores.
jge   mayorIgual      ;¿x ≥ 0?
                      ;x < 0
  mov  signo, -1
  jmp  continuar
mayorIgual:           ;x ≥ 0.
je    igual           ;¿x = 0? (no se repite el cmp).
                      ;x > 0
  mov  signo, 1
  jmp  continuar
igual:                ;x = 0
  mov  signo, 0
continuar:
```

Prog. 6.3 Función Signo

Para evaluar un `and`, debemos evaluar el primer operando. Si es falso, sabemos que toda la expresión es falsa y en consecuencia hay que ejecutar el *si no*. Si es cierto, debemos evaluar el segundo operando, puesto que el resultado total depende de él; en caso de que sea falso, se ejecuta el *si no*; en caso contrario, se ejecuta el *entonces*.

Para el O-lógico, el razonamiento cambia. Si el primer operando es cierto, toda la expresión lo es; por lo tanto, no hay que evaluar el resto de la expresión y se puede proceder a ejecutar el cuerpo del *entonces* . Pero, si el primer operando es falso, el resultado depende del segundo; según si este último es verdadero o falso, se ejecutará el *entonces* o el *si no*. A continuación, se muestran estas construcciones de manera esquemática:

Esquema en C	Esquema en Ensamblador
`if ( cond1 && cond2 )` `  i1;` `else` `  i2;`	`si  !cond1  ir_a  siNo` `si  !cond2  ir_a  siNo` `entonces:` `  i1` `jmp finSi` `siNo:` `  i2` `finSi:`
`if ( cond1 || cond2 )` `  i1;` `else` `  i2;`	`si  cond1  ir_a  entonces` `si  !cond2  ir_a  siNo` `entonces:` `  i1` `jmp finSi` `siNo:` `  i2` `finSi:`

En caso de que la instrucción condicional no tenga *si no* y la evaluación dé falso, se continúa con la ejecución.

Se debe tener cuidado con las condiciones que tienen varios niveles; por ejemplo, en la expresión:

```
(A && B) || (C && D)
```

Si la evaluación de `A` da falso, no podemos concluir que toda la expresión es falsa, únicamente `(A && B)` lo es. Pero sí podemos pasar a evaluar `(C && D)` sin evaluar `B` (ver fig. 6.1).

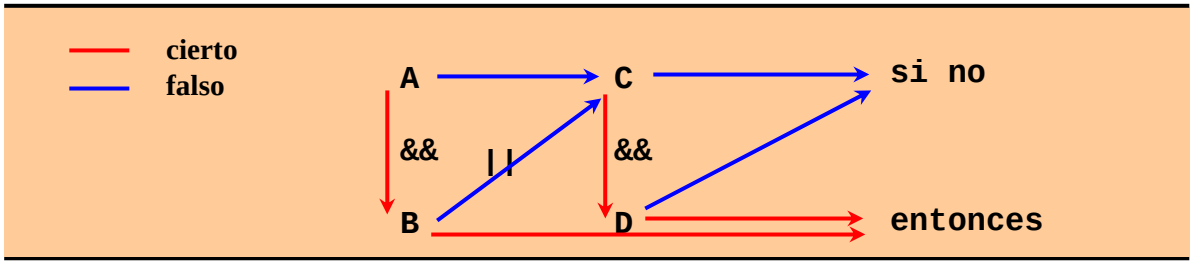

Fig. 6.1. Evaluación de expresiones y ejecución.

A manera de ejemplo, en el prog. 6.4 se muestra la traducción del siguiente *si*:

```
if ( i < n ) && ( v[i] != buscado )
    i++;
  else i = -1;
```

Donde todas las variables son enteros.

Para terminar, note que el flujo de control también se puede usar para evaluar expresiones lógicas; en este caso, en lugar de producir un salto, se obtiene un valor (cierto o falso). Por ejemplo, la asignación:

```
c = ( i < n ) && ( v[i] != buscado );
```

Se puede evaluar como en el programa 6.4, pero la instrucción del *entonces* sería:

```
mov  c, 1       ; c = cierto
```

Equivalentemente, la instrucción del *si_no* sería:

```
mov  c, 0       ; c = falso
```

6.2 INSTRUCCIONES DE ITERACIÓN

En realidad, en lenguaje de máquina no hay mayor diferencia sintáctica entre *si* y *mientras-que*: las dos son construcciones que se escriben usando las mismas instrucciones de salto. La diferencia fundamental es que el *mientras-que* usa saltos que van hacia atrás. Por otro lado, hay algunas instrucciones especialmente pensadas para la iteración; las veremos un poco más adelante.

Iteración *while*, *do-while* y *for*

En programación de bajo nivel, un *mientras-que* puede verse como un *si* que se repite en tanto sea cierta su condición. En consecuencia, la forma de implementarlo es escribir un *si* poniendo al final un `jmp` al principio del *si*. Esquemáticamente:

Esquema en C	Esquema en Ensamblador
`while ( cond )` `  i;`	`principioWhile:` `si  !cond  ir_a  finWhile` `  i` `jmp principioWhile:` `finWhile:`

Por ejemplo, si tomamos el *si* usado en la sección pasada, y le quitamos el entonces, lo podemos convertir en un *mientras-que* (búsqueda de un elemento en un vector):

```
mov   esi, i
cmp   esi, n      ;se modifican los indicadores.
jge   siNo        ;¿i≥n? ¿No se cumple la condición 1? Ir a si no.
            ;i < n
mov   eax, v[4*esi]
cmp   eax, buscado      ;se modifican los indicadores.
je    siNo              ;¿buscado = v[i]? Ir a si no.
            ;i < n && ( v[i] != buscado )
  inc   i
  jmp   finSi
siNo:       ;i ≥ n || ( v[i] = buscado )
  mov   i, -1
finSi:
```

Prog. 6.4 Evaluación de condición con &&

```
while  ( i < n ) && ( v[i] != buscado )
    i++;
```

En el prog. 6.5 está la implementación en ensamblador. Note que es igual al programa 6.4, excepto por la ausencia del *si-no* y por el salto del final.

En el caso de la construcción *do-while*, no es necesario el salto incondicional (`jmp`), como se puede ver en el siguiente esquema:

Esquema en C	Esquema en Ensamblador
do *i;* while (*cond*)	principioDoWhile: *i* si *cond* ir_a principioDoWhile

La iteración tipo *for* no reviste particular interés puesto que se puede reducir a una iteración tipo while:

Esquema en C	Esquema en Ensamblador
for (i_1; i_2; i_3) *i;*	i_1; while (i_2) { *i;* i_3; }

Otras construcciones con saltos

```
principioWhile:
mov   esi, i
cmp   esi, n
jge   finWhile   ;¿i≥n? Ir a finWhile.
           ;i < n
mov   eax, v[4*esi]
cmp   eax, buscado     ;se modifican los indicadores.
je    finWhile         ;¿buscado = v[i]? Ir a finWhile.
           ;i < n && ( v[i] != buscado )
  inc   i
  jmp   principioWhile   ;Volver a realizar la comparación
finWhile:  ;i ≥ n || ( v[i] = buscado )
```

Prog. 6.5 Iteración tipo *While*

Los primeros lenguajes de programación tenían pocas construcciones de control, y se apoyaban fuertemente en la instrucción GO TO[1], equivalente a los saltos del lenguaje de máquina. Con el tiempo, se hizo evidente que esto conducía fácilmente a programas enmarañados y confusos; difíciles de construir y de mantener[2]. Por esta razón, surgió la propuesta de la *programación estructurada*.

El argumento detrás de la programación estructurada consiste en que la instrucción GO TO es demasiado flexible, y, por ende, permite crear una gran variedad de construcciones que pueden no ser muy consistentes, no muy claras en sus propósitos y difíciles de reconocer, entender y manipular por un programador. La propuesta de la programación estructurada es eliminar la instrucción GO TO e implementar un conjunto limitado de construcciones de control; típicamente, *if*, *while* y variantes cercanas (*switch*, *do-while*, *for*).

Ahora bien, el ensamblador, por su naturaleza misma, muy fácilmente puede caer en la programación no estructurada. Sobre esto hay varios comentarios:

- Cuando se trata de un compilador generando código a partir de un programa de alto nivel, no es importante: quien debe ser estructurado es el programa de alto nivel[3]. No es extraño que los compiladores generen código ensamblador que podría ser considerado como no estructurado —para mantener el programa eficiente o compacto, o, sencillamente, por facilidad— .
- Si se trata de código ensamblador escrito por un humano, es mejor imponerse una cierta disciplina de estructuración; que puede consistir en

[1] Por ejemplo, no había *while*, sino que se construía con el *if* y el *go to*.

[2] Conocido como “código espagueti”, por la maraña que crea en el flujo de control.

[3] En efecto, en este caso, el código ensamblador no es generado ni mantenido por nadie, así que no importa si es estructurado o no.

diseñar el programa pensando en construcciones de alto nivel y no directamente en ensamblador. Sin embargo, si se está programando en ensamblador por eficiencia, es natural que aparezcan algunas desviaciones con respecto a la programación estructurada estricta.

- Por último, el programador en ensamblador puede recurrir a construcciones que no por ser poco usuales son menos estructuradas. Es decir, el programador en ensamblador puede tomar otras construcciones de lenguajes de alto nivel o crear unas nuevas propias sin que por ello sean desestructuradas.

Veamos algunos ejemplos. Edsger Dijkstra, en su lenguaje de comandos guardados, propone la primitiva *do-od*, que tiene la siguiente estructura:

```
do
  guarda1 → i1
  …
  guardan → in
od
```

Las $guarda_i$ son expresiones lógicas. Esta construcción itera como un *while* mientras alguna de las guardas sea cierta; si todas son falsas, se detiene. Si hay varias que son ciertas, se selecciona la primera que se encuentre —en la versión original de Dijkstra se selecciona una cualquiera de ellas al azar—. Se puede traducir a ensamblador como:

```
principioDoOd:
si   !guarda1  ir_a   evaluar2
   i1
   jmp principioDoOd
evaluar2:
si   !cond2  ir_a   evaluar3
   i2
   jmp principioDoOd
…
evaluarn:
si   !condn  ir_a   abandonarDoOd
   in
   jmp principioDoOd
abandonarDoOd:
```

La construcción anterior se puede generalizar en ensamblador: si una guarda es cierta, se ejecuta la instrucción asociada y, después, se puede continuar con el ciclo o abandonarlo, a gusto del programador. Esta construcción, aunque no completamente desordenada, pisa los terrenos de lo no estructurado.

¿Para qué podría servir una tal construcción? Permite iterar, pero efectuando unas acciones especiales, justo antes de abandonar el ciclo, que dependen de la condición de salida. A manera de ejemplo, en el prog. 6.6 tomamos el *si-entonces-sino* del prog. 6.4, con algunos cambios en el orden de las instrucciones, y lo

ponemos a iterar. Se convierte en un programa que busca un cierto elemento en un vector y reporta en qué índice lo encontró, o -1, si no lo encuentra.

Expresado en el do-od[4] modificado, daría:

```
do
  ( i == n ) → i = -1; exit;
  ( v[i] !=  buscado ) → i++; continue;
od
```

En términos generales, cuando se está evaluando una condición compleja en ensamblador, se utilizan múltiples saltos; cada uno de estos saltos puede tener un destino distinto, lo cual lo hace muy flexible pero muy propenso al desorden.

Reutilización de registros

Como se mencionó anteriormente, los registros se utilizan para mantener las variables más usadas. Ahora bien, como se tiene un número limitado de registros, puede ocurrir que se agoten.

Un caso donde esto puede ocurrir fácilmente es cuando hay varios ciclos anidados. En tal caso, cuando se pasa de un ciclo externo a un ciclo interno, se pueden guardar los registros en variables y, cuando se sale del ciclo interno, se reponen a partir de las variables en memoria. De esta manera los registros quedan disponibles para el ciclo interno.

Iteración automática

La IA32 dispone de unas instrucciones que realizan todas las operaciones necesarias para iterar, o parte de estas, de una manera automática.

```
principioWhileMultiple:
mov   esi, i
cmp   esi, n
jl    comparar   ;¿i<n? Ir a evaluar la otra guarda.
           ;i = n. No encontró el elemento.
  mov   i, -1
  jmp   finWhileMultiple
comparar:
mov   eax, v[4*esi]
cmp   eax, buscado
je    finWhileMultiple;¿buscado = v[i]? Encontró el elemento.
           ;i < n && ( v[i] != buscado )
  inc   i
  jmp   principioWhileMultiple
finWhileMultiple:
```

Prog. 6.6 *while* múltiple

[4] Recuerde que el *do-od* original no tiene las primitivas *exit* y *continue*.

Es importante aclarar que estas instrucciones son una particularidad de la IA32; otras máquinas no necesariamente tendrán algo similar. Son una facilidad para el programador, pero no son estrictamente necesarias; de hecho, otras máquinas iteran solamente por medio de instrucciones de salto. Consideramos interesante mostrar estas instrucciones como una curiosidad, y como una ilustración de una característica "muy CISC" de la IA32, pero sin entrar en el detalle de su uso.

En primer lugar, se dispone de la instrucción `loop`. Esta instrucción se comporta parecido a un *for*: se le indica en `ecx` el número de iteraciones que se desea realizar y `loop` las efectúa automáticamente.

`loop` es como un salto: su operando es la dirección donde se debe saltar. Pero las acciones que realiza son un tanto más complejas: en cada ciclo, `loop` decrementa `ecx`, si `ecx` es diferente de cero, salta; de lo contrario, ejecuta la instrucción siguiente.

A manera de ejemplo, el prog. 6.7 efectúa la copia de una región de memoria usando `loop`: pasa *n* dobles palabras de *zona1* a *zona2*[5]. ¿Para qué puede servir esto? Para copiar vectores o estructuras.

A pesar de que `loop` decrementa `ecx`, no afecta los indicadores. La IA32 aprovecha este detalle para definir unas variantes de `loop`: `loope` (*LOOP if Equal*) y `loopne` (*LOOP if Not Equal*)[6].

Estas variantes, en principio realizan `ecx` iteraciones, pero además, en cada iteración, revisan la condición; si la condición se cumple, continúan con el ciclo —a menos que `ecx` haya llegado a cero—; si la condición no se cumple, suspenden el ciclo aun si `ecx` no ha llegado a cero.

```
;Precondición: zona1 y zona2 son dos regiones de memoria con, al
menos, n > 0 dobles palabras.
;Poscondición: ∀ i < n . zona1[i] = zona2[i]
mov  ecx, n
mov  esi, 0
copiarDWord:                ;∀ i < esi . zona1[i] = zona2[i]
  mov  eax, zona1[4*esi]
  mov  zona2[4*esi], eax    ; zona1[i] = zona2[i]
  inc  esi
loop copiarDWord            ;ecx = ecx - 1; ¿ecx > 0?
```

Prog. 6.7 Copiado de zonas de memoria

[5] El programa puede ser más compacto si no se usa `esi`; en su lugar se puede usar `ecx` y se hace el copiado del final hacia el principio. Pero, en ese caso, el direccionamiento indexado se debe hacer: "*zona1*`[4*`**`ecx`**`-4]`" y "*zona2*`[4*`**`ecx`**`-4]`"; puesto que `ecx` va de *n* a 1, y no de *n*-1 a 0.

[6] Se dispone también de los seudónimos `loopz` y `loopnz`.

En la primera variante (`loope`), se decrementa `ecx`, después, si `ecx` ≠ 0 y `Z` = 1, realiza el salto; de lo contrario, pasa a la instrucción que sigue. Es como un `JE` que, además, revisa el registro `ecx`.

La segunda variante (`loopne`) decrementa `ecx`, si `ecx` ≠ 0 y `Z` = 0, efectúa el salto; si no, pasa a la instrucción siguiente. Es como un `jne` que revisa `ecx`.

Estas instrucciones, aunque extrañas, tienen su utilidad, si bien son bastante especializadas. En general, se usan en construcciones de la forma:

```
cmp   eax, v[esi]          cmp   eax, v[esi]
loopne  iterar             loope  iterar
```

Sin entrar en detalles, la primera sirve para operaciones de búsqueda —itere mientras sean diferentes— y la segunda, para operaciones de comparación —itere mientras sean iguales—.

Además, la IA32 tiene instrucciones especiales para manejar secuencias de bytes, palabras o dobles palabras. Estas instrucciones se llaman `movs`, `lods`, `stos`, `scas` y `cmps` —*MOVe*, *LOaD*, *STOre*, *SCAn* y *CoMPare String*—. Cada una de ellas tiene variantes para operar sobre cadenas de bytes, palabras o dobles palabras; por ejemplo: `movsb`, `movsw`, `movsd`. Estas instrucciones operan usando los registros `esi`, `edi`, `al`, `ax`, `eax`, exclusivamente, tal como se muestra a continuación:

MOVS	LODS	STOS	SCAS	CMPS
[EDI] ← [ESI] Incrementar ESI Incrementar EDI	*RegA* ← [ESI] Incrementar ESI	[EDI] ← *RegA* Incrementar EDI	*RegA* - [EDI] Incrementar EDI	[EDI] - [ESI] Incrementar ESI Incrementar EDI

RegA se reemplaza por `al`, `ax` o `eax`, según si se trata de la instrucción de bytes, palabras o dobles palabras; el tamaño de lo apuntado por `esi` y `edi` también depende de la versión que se esté utilizando[7]. Igualmente, el incremento de `edi` y `esi` depende de la versión—1,2 ó 4 bytes—; también puede ser positivo o negativo, lo cual permite hacer recorrido de "abajo a arriba" o de "arriba a abajo" —la dirección la determina el indicador `D`: si `D`=1, la memoria se explora "hacia abajo"—
.

A manera de ejemplo en el prog. 6.8 implementamos de nuevo el programa que copia una zona de memoria en otra.

Estas instrucciones se vuelven completamente automáticas con la instrucción-prefijo `rep` (*REPeat string operation*), la cual se pone inmediatamente antes de la instrucción de cadena. La instrucción de cadena se repite tantas veces como se especifique en el registro `ecx`. El programa de copiado de memoria se convierte en:

```
mov   ecx, n
lea   esi, zona1
```

[7] Como nota al margen, estas instrucciones nos explican los nombres de `esi`—*Source Index*— y `edi` —*Destination Index*—; puesto que sirven de fuente y destino de información para ellas.

```
    lea   edi, zona2
    rep   movsd
```

```
;Precondición: zona1 y zona2 son dos regiones de memoria con, al
menos, n > 0 dobles palabras.
;Poscondición: ∀ i < n . zona1[i] = zona2[i]
mov  ecx, n
lea  esi, zona1
lea  edi, zona2
copiarDWord:
  movsd
loop copiarDWord          ;ecx = ecx - 1; ¿ecx > 0?
```

Prog. 6.8 Copiado de zonas de memoria

De la misma manera que `loop`, `rep` tiene unas variantes —`repe` y `repne`— que pueden suspender la iteración cuando se presenta cierta condición: `repe` (*REP if Equal*) repite la iteración si `Z`=1; `repne` (*REP if Not Equal*) repite si `Z`=0. Estas variantes se utilizan únicamente antes de `scas` y `cmps`, con el fin de hacer comparación y búsqueda de cadenas.

6.3 SALTOS Y MODOS DE DIRECCIONAMIENTO

Los saltos se pueden clasificar en dos categorías según la forma de su operando:

- Direccionamiento relativo: el operando es un desplazamiento con respecto al PC. Es decir, el operando se interpreta como: "salte a la dirección *n*".
- Direccionamiento absoluto: el operando es una dirección absoluta. Es decir, el operando se interpreta como: "salte n bytes".

Existe un tercer tipo de direccionamiento para los saltos, llamado *indirecto*, donde el operando es una variable o un registro, cuyo valor es la dirección a donde se debe saltar.

Observe que el operando no es la dirección en sí sino que, de cierta manera, apunta a la dirección. Como consecuencia de esto, el destino del salto solo se conoce en ejecución y puede cambiar dinámicamente —si la variable cambia de valor—.

A manera de ejemplo, supongamos que en un programa tenemos una etiqueta "e", y tenemos la siguiente declaración:

```
;Si escribe solo e, el ensamblador lo interpreta igual.
v dword  offset e
```

Todos los saltos siguientes tendrían el mismo efecto:

```
jmp e      ;el salto usual, directo.
jmp v      ;Salto indirecto por V.
mov eax, v
jmp eax    ;Salto indirecto por eax.
```

```
mov ebx, offset v
jmp [ebx]  ;Salto indirecto por [ebx].
```

La diferencia del primero con los otros, es que el primero no puede cambiar nunca su destino de salto; los otros sí, eventualmente.

Un ejemplo del uso de este modo de direccionamiento es la implementación, en algunos casos, de la instrucción *switch*:

```
switch ( i ) {
   case 0:  Caso0
   case 1:  Caso1
      …
   case n:  Cason
}
```

Esta instrucción se puede codificar en ensamblador escribiendo n+1 etiquetas (e0,..., en), cada una con el código correspondiente a una entrada del *switch*:

```
 e0:   Caso0
 e1:   Caso1
       …
 en:   Cason
```

Y, según el valor de una variable i —0, ... ,n—, se ejecuta el código asociado a la etiqueta correspondiente. Lo cual se puede hacer como se muestra a continuación.

```
;se define un vector con las etiquetas.
v  dword  e0, … , en
 …
mov esi, i
jmp v[4*si]          ;salte a la i-ésima etiqueta.
```

En términos generales, la programación usando los saltos indirectos es truculenta y se debe usar cuidadosamente; sin embargo, es útil para los sistemas operativos y los compiladores. La verdadera importancia de este direccionamiento aparece en conjunción con la llamada a procedimientos, que veremos en el siguiente capítulo.

De hecho, aunque la IA32 tiene instrucciones especiales para hacer llamadas de procedimientos, es posible crear la invocación de procedimientos sin usarlas, aprovechando estos saltos dinámicos.

Un procedimiento es un segmento de código al cual se salta desde diferentes puntos del programa, pero con la capacidad de retornar al sitio donde fue invocado. El código a continuación ilustra cómo se podría hacer un llamado de procedimiento:

```
mov edx, offset e ;en edx está la dirección de retorno.
jmp p     ;p es una etiqueta donde está el procedimiento.
e:        ;la etiqueta e es diferente en cada invocación.
  …       ;las instrucciones que siguen.
```

A continuación mostramos un esquema de cómo sería el código del procedimiento:

```
p:
  …       ;las instrucciones del procedimiento.
jmp edx   ; en edx está la dirección de retorno.
```

Repetimos: la IA32 tiene instrucciones especiales para hacer esto; pero, como puede ver, los saltos dinámicos serían suficientes para implementarlos.

6.4 OPTIMIZACIÓN DE CÓDIGO

Los compiladores modernos no se limitan a hacer fieles traducciones del código fuente; también tienen la capacidad de optimizar el código generado.

La optimización se puede hacer según dos consideraciones generales:

- Optimización por tamaño. Se desea que el código no ocupe mucho espacio, que sea compacto, así sea algo más lento.
- Optimización por tiempo. Se desea que el programa corra lo más rápido posible, aunque eso implique gastar más espacio.

En algunas ocasiones las técnicas aplicadas coinciden, pero con frecuencia se oponen: si se quiere ganar velocidad, se pierde espacio. A continuación veremos algunas técnicas que pueden ser utilizadas.

Optimización de expresiones

Una primera técnica muy simple es la *propagación de constantes*. Consiste en detectar variables que se inicializan pero nunca se modifican; en esta situación, en lugar de crear y usar la variable, se puede utilizar su valor en la compilación.

Otra técnica es la identificación de *subexpresiones comunes*. Consiste en detectar expresiones que se repiten para solo calcularlas una vez. Por ejemplo:

```
x = (y+z)*(y+z);
```

En lugar de calcular dos veces `y+z`, se calcula la primera y el valor obtenido se multiplica por sí mismo.

Las técnicas de más alto nivel hacen transformaciones matemáticas en las expresiones. Por ejemplo:

```
x = (y+z)*a + (y+z)*b;
```

Se puede transformar en:

```
x = (y+z)*(a + b);
```

Note que no es lo mismo que la identificación de subexpresiones comunes; en la identificación de subexpresiones comunes se calcularía `y+z` una sola vez, pero se harían las dos multiplicaciones.

Las técnicas de más bajo nivel se aprovechan de trucos de la máquina para calcular más eficientemente algunas expresiones. Por ejemplo, en lugar de multiplicar por 2 hacer un corrimiento a la izquierda. En la IA32 hay un truco bastante usual basado en la instrucción `lea`:

```
lea eax, [esi + 4]
```

Aunque lea está destinada para la creación de apuntadores, el caso es que la instrucción anterior es equivalente a:

```
mov eax, esi
add eax, 4
```

Por supuesto, se pueden hacer otros trucos similares, como:

```
lea eax, [esi + edi]
```

Lo cual es equivalente a:

```
mov eax, esi
add eax, edi
```

En última instancia esto equivale a crear una operación suma de tres operandos registros —como en los computadores RISC—.

Otra técnica, que ya mencionamos, es la *eliminación de temporales*. Consiste en manejar las variables temporales en los registros en lugar de hacerlo en memoria, con el consiguiente ahorro de espacio y de accesos a la memoria.

Sobra decir que estas técnicas no se pueden aplicar "automáticamente"; el compilador debe analizar el código para asegurarse de no estar adulterando el código. Por ejemplo, con la factorización es necesario garantizar que las operaciones intermedias no tienen efectos de borde (como los operadores ++ y --); de lo contrario el código transformado no será equivalente al original.

Optimización de condicionales

Solo mencionaremos dos técnicas —la misma en el fondo—: la factorización de anterior y posterior de instrucciones:

Código original	Código transformado
if (*cond*){ i_1; i_2; i_3; } else{ i_1; i'_2; i_3; }	i_1; if (*cond*){ i_2; } else{ i'_2; } i_3;

Como mencionamos en la sección anterior, esta técnica requiere que el código sea analizado para evitar la adulteración del código; en concreto, en este caso, si la condición tiene efectos de borde sobre las variables usadas en las instrucciones.

Optimización de ciclos

Las optimizaciones sobre los ciclos son las más importantes, puesto que es en los ciclos donde se gasta la mayor parte del tiempo de ejecución.

Mencionaremos aquí dos técnicas clásicas.

Desenrollado de ciclos (loop unrolling)

Consiste en disminuir el número de iteraciones, incrementando lo que se hace en cada una de ellas.

En ocasiones, cuando se estudia un ciclo, se olvida que la evaluación de la condición toma tiempo; si se disminuye el número de veces que se hace, se disminuye el tiempo de ejecución. Por otra parte, como veremos en la parte de estructura del computador, las instrucciones de salto dificultan, o anulan, algunas técnicas de aceleración que se usan en la estructura del procesador.

Básicamente esta transformación consiste en lo siguiente:

Código original	Código transformado
`for ( i = 0; i < n; i++ ){` `  instrucciones(i);` `}`	`for ( i = 0; i < n; i+= 2 ){` `  instrucciones(i);` `  instrucciones(i+1);` `}`

Es decir, en cada iteración se efectúan dos de las de antes. Por supuesto, el desenrollado de ciclos se puede hacer más agresivo: 4 ó más veces.

De nuevo, se debe analizar el código para garantizar la equivalencia. En este caso particular, se debe tener cuidado con el número de iteraciones: si no es par, se debe iterar `n-1` veces, y después del ciclo se debe agregar lo que se llama *código compensatorio*: un pedazo de código que se encarga de la última iteración.

Segmentación software (software pipelining)

La segmentación es una técnica hardware que consiste en no ejecutar las instrucciones una tras otra, sino en traslaparlas; es decir, mientras se está ejecutando una instrucción se procede a leer la siguiente y a empezar su ejecución. Esto lo veremos con mayor detalle en la sección de estructura del computador.

En la segmentación software, las iteraciones se traslapan de manera que en cada pasada por el ciclo en realidad se están ejecutando partes de varias iteraciones distintas. Una forma de verlo es la siguiente: cada iteración se puede ver compuesta de varios cálculos, llamémoslos `p`, `q` y `r`; estas tres partes se pueden traslapar de la siguiente manera:

Código original	Código transformado
`for ( condición ){` `  p(i);` `  q(i);` `  r(i);` `}`	`p(1);` `q(1);` `p(2);` `for ( condición ){` `  r(i);` `  q(i+1);` `  p(i+2);` `}` `r(n-1);` `q(n);` `r(n);`

Por supuesto, no tienen que ser tres partes; depende de la estructura del código. Puesto que la cantidad de trabajo no disminuye, solo se ejecuta en otro orden, ¿cuál es la utilidad de esta técnica? La importancia se encuentra en la estructura del procesador: los procesadores modernos intentan ejecutar varias instrucciones al tiempo; para hacer esto, necesitan que las instrucciones no dependan la una de la otra; al entremezclar instrucciones de diferentes iteraciones, incrementamos la probabilidad de que las instrucciones no dependan entre ellas. A continuación encuentra un ejemplo:

Código original	Código transformado
`for ( i = 0; i < n; i++){` `  //a[i] = b[i]*c + d[i];` `  m = b[i]*c;` `  s = m + d[i];` `  a[i] = s;` `}`	`m = b[0]*c;` `s = m + d[0];` `m = b[1]*c;` `for ( i = 0; i < n-2; i++){` `  a[i] = s;` `  s = m + d[i+1];` `  m = b[i+2]*c;` `}` `a[n-2] = s;` `s = m + d[n-1];` `a[n-1] = s;`

Note que en el código original, las instrucciones deben ser ejecutadas una detrás de otra; en el código transformado, las tres instrucciones son independientes, luego pueden ser ejecutadas simultáneamente.

EJERCICIOS

1- Escriba un programa en ensamblador para calcular el máximo de tres números. El máximo debe quedar en `eax`.

2- **a-** En el registro `al` se tiene un número entre 0 y 15. Convierta el número en un carácter que representa el dígito hexadecimal correspondiente (un carácter entre '0' y '9' o 'A' y 'F', según sea el caso). El carácter debe quedar en `al` mismo.

b- En el registro `al` se tiene un carácter que representa un dígito hexadecimal (un carácter entre '0' y '9' o 'A' y 'F'). Conviértalo al valor correspondiente entre 0 y 15. El valor debe quedar en `al` mismo.

3- Se tiene un vector `v` de enteros de `n` posiciones, y se va realizar la asignación:

```
v[i] = a;
```

Escriba código en ensamblado para hacer esta asignación pero solo si `i` es un subíndice válido (está entre `0` y `n-1`). Si no lo es, no se hace nada.

4- En el registro `al` se tiene una carácter representado en ASCII. Si el carácter es una minúscula, conviértalo a mayúsculas; si es mayúscula o no es una letra, no le haga nada. El carácter debe quedar en `al` mismo.

5- En el registro `al` se tiene una carácter representado en ASCII. Si el carácter es una minúscula, conviértalo a mayúsculas; si es mayúscula, conviértalo en minúscula; si no es una letra, no le haga nada. El carácter debe quedar en `al` mismo.

6- Un año es bisiesto si es divisible por 4, excepto si es divisible por 100, excepto si es divisible por 400 (es decir, si es divisible por 400, sí es bisiesto). Escriba un programa en ensamblador para determinar si un año es bisiesto. `eax` debe quedar en 1, si lo es; en cero, si no.

7- Para cada una de las condiciones que se encuentran a continuación, haga el diagrama de evaluación respectivo (como el de la figura 6.1) y escriba el programa para evaluarla. Suponga que son condiciones para un `if` y que hay etiquetas de `entonces` y `si_no`.

Expresión
`(a || b)&&(c || d)`
`(a < b)||(c > d)`
`(a < b)&&(c > d)`
`(a < b)||(c == d)`
`(a < b)&&(c >= d)`
`(a < b)||(c != d)`
`!(a || b)&&(c || d)`
`!((a || b)&&(c || d))`
`(a || (b && c))&&(d || e)`
`(a || b)&&(c || d) || e`

8- Traduzca la siguiente expresión de C a ensamblador.

```
x = ( a > b ? a : b );
```

9- El siguiente programa en C tiene unas instrucciones en comentarios. Implemente la instrucción que aparece en el comentario en ensamblador en el sitio donde se encuentra el comentario.

```
int total = 0;
int v[50];
char c[50];

int main(int argc, char* argv[])
{
  int i;
  int * p = v;

  for ( i = 0; i < 50; i++){
    //c[i] = i+1;
  }
  for ( i = 0; i < 50; i++){
    //v[i] = (i+1)*100;
  }
  for ( i = 0; i < 50; i++){
    //total += *p++;
  }
  printf("total = %d\n",total);
```

```
  for ( i = 0; i < 50; i++){
    //v[i] = c[i];
  }
  p = v;
  total = 0;
  for ( i = 0; i < 50; i++){
    total += *p++;
  }
  printf("total = %d\n",total);

  return 0;
}
```

10- Traduzca el siguiente código de C a ensamblador.

```
char v[100], u[100];
int n, i;
   …
n = 1;
i = 1;
while ( (i < 100) && ( n == 1 ) ) {
  if ( (v[i] & 223) != (u[i] & 223) )
    n = 0;
  i++;
}
```

11- Traduzca el siguiente código de C a ensamblador.

```
int year, mes, nDias;
   …
switch (mes) {
  case 4:
  case 6:
  case 9:
  case 11:
    nDias = 30;
    break;
  case 2:
    if ( !(year % 4) && (year % 100)) || !(year % 400)
      nDias = 29;
    else
      nDias = 28;
    break;
  default: nDias = 31;
}
```

12- Traduzca el siguiente código de C a ensamblador.

```
char * s;
int n;
   …
n = 0;
while ( *s != '\0' && '0' <= *s && *s <= '9' ) {
  n = 10*n + (*s - '0');
  s++;
```

```
}
```

13- Traduzca el siguiente código de C a ensamblador.

```
char * s;
…
while( *s!='\0' &&
       (('a'<=*s && *s<='z') || ('A'<=*s && *s <='Z')) )
  s++;
```

14- Traduzca el siguiente código de C a ensamblador.

```
char * s;
int i;
   …
for ( i = 0; s[i] != '\0'; i++ ) {
   if ( s[i] > 64 && s[i] <= 90 )
     s[i] = s[i] | 32;
}
```

15- Escriba un programa en ensamblador para calcular el producto punto de dos vectores.

16- En ensamblador, desarrolle programas para sumar y restar números de cualquier longitud. Los números se almacenan en vectores de palabras. Suponga que los dos operandos tienen la misma longitud.

Nota: las instrucciones `adc` y `sbb` le serán de utilidad.

17- En ensamblador, haga un programa que calcule Fibonacci de N. Recuerde:

Fibonacci(0) = Fibonacci(1) = 1
Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)

18- Manejo de cadenas. Escriba programas para:

a- Concatenar dos cadenas.

b- Comparar dos cadenas. Deja eax en 1 si son iguales; en cero, si no.

c- Mirar si una cadena es prefijo de otra; por ejemplo, "casa" es prefijo de "casados". Si es prefijo, debe poner `eax` en 1; y en 0, si no.

d- Mirar si una cadena se encuentra dentro de otra; por ejemplo, "caso" está al interior de "ocasos". Si está, debe poner en `eax` la posición a partir de la cual se encuentra (1, en el ejemplo); si no está, debe poner `eax` en -1.

e- Eliminar un cierto número de caracteres a partir de una posición. El programa debe tener la posición inicial y el número de caracteres que debe borrar. Por ejemplo, si hay que borrar 3 caracteres desde la posición 1 en la cadena "mañana", queda "mna". Tenga en cuenta todos los casos de error.

f- Insertar una cadena dentro de otra. El programa debe tener la posición donde hay que insertar la cadena. Por ejemplo, al insertar "aña" en "mna" a partir de la posición 1, se obtiene "mañana".

g- Pasar una cadena de mayúsculas a minúsculas. La conversión debe hacerse sobre la misma cadena, no use cadenas auxiliares.

h- Haga un programa que detecta si dos cadenas son iguales o no. Las cadenas se consideran iguales si están compuestas de los mismos caracteres alfabéticos en minúsculas; los caracteres intermedios en mayúsculas y los no alfabéticos se desprecian. Por ejemplo, "una NA ca sa de, madera" es igual a "?una ,%$cas:a de madera.?".

19- Listas encadenadas.

a- Escriba un programa en ensamblador que invierte una lista encadenada. El apuntador a la cabeza de la lista está en una variable *cabeza* de tipo doble palabra; cada nodo tiene dos campos de tipo doble palabra: el primero es la información y el segundo es un apuntador al siguiente nodo.

El apuntador vacío (`NULL`) se representa por 0.

b- Escriba un programa para insertar un nodo en la lista anterior.

c- Escriba un programa que, dado un número, busca el nodo que tiene dicho número en su campo `valor` y lo retira de la lista.

20- Manejo de matrices.

a- Escriba un programa que multiplique dos matrices de `dword`.

b- Desarrolle un programa que calcule la transpuesta de una matriz de `dword`. Debe calcular la transpuesta sobre la misma matriz, no puede usar matrices intermedias.

c- Una matriz inferior diagonal-n es aquella que tiene *n* diagonales en cero, contando las diagonales desde la esquina superior derecha. Por ejemplo, las dos matrices siguientes son diagonal-2 y diagonal -3, respectivamente:

11	6	0	0
10	65	15	0
2	0	5	17
42	7	23	21

11	0	0	0
10	65	0	0
2	0	5	0
42	7	23	1

Escriba un programa en ensamblador que detecte si una matriz es inferior diagonal-*n*. El resultado se deja en `eax`: 1, sí es; cero, no es.

d- Una matriz simétrica-*n* es aquella cuyas *n* diagonales superiores son iguales a las n diagonales inferiores. Ejemplo:

11	6	2	42
0	65	15	7
2	51	5	17
42	7	6	2

Escriba un programa en ensamblador que reconozca si una matriz es simétrica-*n*. Utilice las mismas convenciones del punto anterior.

e- Una matriz banda-*m* es aquella cuyas *m* diagonales centrales son las únicas con valores diferentes de cero. Ejemplo:

11	6	0	0
0	65	15	0
0	51	5	17
0	0	6	2

Escriba un programa en ensamblador que determine si una matriz es banda-*m*. Las convenciones son las mismas de los puntos anteriores.

21- Se tiene un vector de 4 bytes y se quiere:

a- Invertir los cuatro bytes. Es decir, el último debe quedar de primero, el penúltimo debe quedar de segundo, etc.

b- Invertir completamente el contenido de los cuatro bytes. Es decir, el último bit debe quedar de primero, el penúltimo debe quedar de segundo etc.

22- El objetivo de este ejercicio es practicar las conversiones de número binario a cadena de caracteres decimales que debe efectuar el computador.

a- Escriba un programa que lee una cadena de caracteres, la cual debe estar compuesta de caracteres-dígitos ('0' a '9'), y la convierte a binario. El resultado debe quedar en EAX.

b- Desarrolle un programa que convierte un número en EAX a notación decimal en una cadena de caracteres.

c- Repita las partes a- y b- pero convirtiendo de binario a hexadecimal-carácter y viceversa.

d- Repita las partes a- y b- pero convirtiendo de binario a binario-carácter y viceversa.

23- Se quiere hacer un programa que evalúe una expresión aritmética codificada en una cadena de caracteres. Los caracteres son dígitos o signos de operadores aritméticos binarios (+, -, * y /). El final de la cadena se indica por el carácter '='. Por ejemplo, la cadena puede ser "265+73*54=". Evalúe las expresiones de izquierda a derecha sin importar la prioridad del operador.

24- Escriba un programa que calcula el bit de paridad para el número que se encuentra en `eax`, pero sin utilizar el indicador (*flag*) de paridad.

25- Escriba un programa que ordena un vector de `dword`.

26- Escriba un programa para calcular el máximo común divisor de dos números. Utilice la siguiente definición:

{ Suponiendo que a > b }
MCD (a,b) = a, si b = 0
MCD (b, a MOD b), si b > 0

27- Se tienen dos vectores ordenados de números enteros. Se desea mezclarlos en un vector resultado —también ordenado—. Escriba un programa que realice

esta mezcla, con la condición de que los tres vectores se deben recorrer una sola vez; es decir, los dos vectores se deben ir recorriendo al mismo tiempo, y el vector resultado se debe ir construyendo simultáneamente. Suponga que:

a- No hay elementos repetidos.

b- Puede haber elementos repetidos.

c- Puede haber elementos repetidos, en cuyo caso se eliminan todas las ocurrencias excepto una.

28- Se tienen subvectores representados con las convenciones del ejercicio 4.34. Escriba un programa en ensamblador que sume dos subvectores en esta representación; el resultado debe quedar en la misma representación. Tenga en cuenta que el vector de control del resultado no hay que construirlo, ya está definido y hay que respetarlo.

29- Se tienen subvectores representados con las convenciones del ejercicio 4.35. Escriba un programa en ensamblador que calcule el producto punto de dos vectores representados con estas convenciones.

30- Se tienen subvectores representados con las convenciones del ejercicio 4.36.

a- Escriba un programa en ensamblador que implemente la operación *recolectar*. Esta operación toma un subvector y un vector resultado normal, y arma el subvector en el vector resultado como un vector regular. Por ejemplo, si se tiene el vector:

9	13	0	3	6	57	7	3	2	-1	-8	27	5	5	4	1

Y el vector de índices:

3	6	9	14	15

El vector regular será:

3	7	-1	4	1

b- Escriba un programa en ensamblador que implemente la operación *esparcir*. Esta operación toma un subvector y un vector normal, y pasa los elementos del vector normal al subvector. Por ejemplo, si se tiene el vector:

9	13	0	8	6	57	23	3	2	0	-8	27	5	5	76	54

Y el vector de índices:

3	6	9	14	15

Y el vector regular:

3	7	-1	4	1

El vector se transformará en:

9	13	0	3	6	57	7	3	2	-1	-8	27	5	5	4	1

31- Se tienen vectores dispersos representados con las convenciones del ejercicio 4.37. Escriba un programa en ensamblador que sume dos vectores dispersos

—usando esta representación—, cuyo resultado es otro vector en la misma representación.

Atención: al sumar 2 elementos distintos de cero, el resultado puede ser cero.

32- La iteración se puede escribir de manera diferente a la que se vio en el texto. A continuación, se muestra un nuevo esquema:

```
jmp condicion
iterar:
  instrucciones del ciclo
condicion:
cmp …
jyy   iterar
```

¿Es equivalente al método visto en el texto? ¿Tiene alguna ventaja o desventaja?