4

CONJUNTOS DE INSTRUCCIONES

4.1. Introducción

En este capítulo estudiaremos los distintos tipos de instrucciones de que constan los juegos de instrucciones de los ordenadores. Se verán con especial interés las instrucciones de control del flujo del programa (bifurcaciones, bucles, procedimientos, etc.).

También se estudiarán los principios y filosofía de diseño de los ordenadores con conjunto reducido de instrucciones (RISC).

4.2. Características generales de los conjuntos de instrucciones

Ante el diseño de un nuevo ordenador de propósito general hay que plantearse la siguiente cuestión: ¿Qué tipos de instrucciones deben ser incluidos en su conjuntos de instrucciones? Antes de responder a esta pregunta, analizaremos las características que deben tener los juegos de instrucciones de las máquinas.

Los conjuntos de instrucciones de las máquinas deben tender a poseer una serie de propiedades, bastante ideales e imprecisas, que pueden resumirse en las siguientes:

- El conjunto de instrucciones de un computador debe ser **completo** en el sentido de que se pueda construir un programa para evaluar una función computable usando una cantidad de memoria razonable y empleando un tiempo moderado, es decir, el número de instrucciones de ese programa no debe ser demasiado elevado.
- Los juegos de instrucciones también tienen que ser **eficientes**, esto significa que las funciones más necesarias deben poder realizarse usando pocas instrucciones.
- El conjunto de instrucciones de una máquina debe ser **regular**, es decir debe ser **simétrico** (por ejemplo, si existe una instrucción de desplazamiento a la izquierda, debe haber otra de desplazamiento a la derecha, etc.) y **ortogonal**, es decir, deben poder combinarse, en la medida de lo posible, todos las operaciones con todos los tipos de datos y modos de direccionamiento.

- En muchas ocasiones, también se le debe exigir a un computador que su juego de instrucciones sea **compatible** con modelos anteriores.

4.3. Tipos de instrucciones

Una máquina puede llegar a funcionar con un juego de instrucciones muy limitado (recuérdese, por ejemplo, la máquina de Turing que sólo tiene 4 instrucciones, incluso se han diseñado máquinas teóricas con menos instrucciones), esto simplificaría mucho los circuitos de la máquina. Sin embargo, un conjunto de instrucciones demasiado simplificado origina, como consecuencia, unos programas demasiado complejos e ineficientes. Es necesario encontrar un compromiso entre la simplicidad del hardware y del software. Un mínimo para llegar a ese compromiso se consigue con los tipos de instrucciones siguientes:

- Instrucciones de **transferencia de datos**.
- Instrucciones **aritméticas**.
- Instrucciones **lógicas**.
- Instrucciones de **control del flujo del programa** (bifurcaciones, bucles, procedimientos, etc.)
- Instrucciones de **entrada y salida**.

En los apartados siguientes iremos viendo con detalle algunos de estos tipos de instrucciones.

Si bien es cierto que el conjunto de instrucciones debe de cumplir unos mínimos para conseguir una mínima eficiencia en los programas, también se verá que ésta no se aumenta indefinidamente al incrementar el número de instrucciones de la máquina.

4.3.1. Instrucciones de transferencia de datos

La operación de copiar datos de un lugar a otro es la operación más simple y a la vez importante. Las palabras *mover* o *cargar* que aparecen en los juegos de instrucciones de muchos ordenadores pueden dar lugar a confusión porque no se trata de mover o cargar sino de **copiar** (generalmente, en Informática, la palabra *mover* tiene el significado de *copiar borrando el original*). Las instrucciones de transferencia de datos necesitan que se especifiquen el original (**fuente** u **origen**) y el lugar donde se desea la copia (**destino**). Esta especificación variará según sean estos lugares que pueden estar en tres sitios: registros del procesador, memoria o cima de pila. Si el acceso es a una dirección de memoria habrá que especificarla de forma explícita, si se trata de la cima de pila normalmente la especificación será implícita, lo mismo ocurrirá si se trata del acumulador.

Existen algunas variantes de instrucciones de transferencia de datos que difieren de la idea anterior de copiar informaciones de un lugar a otro. Por ejemplo, la instrucción POP saca un dato de la cima de pila modificando el valor del apuntador de pila, lo que significa que, si bien no

destruye físicamente el dato fuente, anula su validez. Por otro lado, la mayoría de las máquinas también disponen de instrucciones de intercambio (SWAP). También, en muchas máquinas, existen instrucciones de transferencia de información entre bloques o cadenas en la que deben especificarse las direcciones de fuente y destino y la longitud del bloque o cadena.

En general, el dato a transferir podríamos definirlo como una terna con las siguientes componentes:

- **Dirección**
- **Tipo**
- **Valor**

Normalmente la componente del dato que se transfiere es el valor pero existen instrucciones especiales para transferir las demás componentes, en especial la dirección. La extracción de la dirección de un dato se hace necesaria para facilitar la relocalización de los programas. Para aplicar muchos modos de direccionamiento (indexados, autoindexados, etc.) es necesaria la transferencia de una dirección a un registro. En muchos ordenadores esta dirección no se conoce a la hora de compilar el programa (precisamente porque el programa es relocalizable), por tanto, son necesarias instrucciones que calculen la dirección de un dato para transferirla a un registro, actualmente la mayoría de las máquinas poseen este tipo de instrucción bajo el nombre de *move address*.

Dependiendo del nivel de ortogonalidad de la máquina, podemos tener más o menos instrucciones de transferencia. Por ejemplo el IBM-370 tiene más de 20 instrucciones de transferencia por su falta de ortogonalidad; sin embargo, el PDP-11 sólo tiene 2 para datos enteros (MOV y MOVB) y 4 más para datos en punto flotante, ya que en las instrucciones de punto flotante se prescindió bastante de la ortogonalidad por falta de espacio para la codificación. El MC68000 está en un punto intermedio, tiene varios formatos distintos dependiendo del direccionamiento utilizado. El Z-80 en cuanto a instrucciones de transferencia entre registros es bastante ortogonal, pero esa ortogonalidad se pierde cuando se trata de otros direccionamientos.

4.3.2. Instrucciones aritméticas y lógicas

Todos los ordenadores incorporan instrucciones aritméticas en sus juegos de instrucciones; la utilidad de este tipo de operaciones es evidente y no la comentaremos. En cuanto a las operaciones lógicas (AND, OR, NOT y XOR) tienen un uso muy variado: desde operaciones con bits individuales (TEST, SET, RESET y CHANGE) hasta el empaquetamiento y desempaquetamiento de caracteres. También pueden considerarse dentro de este grupo los desplazamientos y rotaciones cuya utilidad queda fuera de toda duda.

En primer lugar, veremos cómo se emplean las instrucciones lógicas para realizar operaciones de bit individual. La utilidad de estas operaciones es muy variada, probablemente su aplicación más importante sea empaquetar en un byte varias variables booleanas, ocupando cada una de ellas un solo bit.

Para analizar un determinado bit se debe tener en cuenta las siguientes propiedades de la operación AND (. o $\wedge$):

$$\begin{array}{rccccccc} & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ \wedge & x & \dots & x & a & x & \dots & x \\ \hline & 0 & \dots & 0 & a & 0 & \dots & 0 \end{array} \quad \begin{array}{l} \text{Máscara} \\ \text{Dato} \\ \\ \text{Resultado} \end{array} \quad \begin{array}{l} = 0 \text{ si } a = 0 \Rightarrow Z = 1 \\ \neq 0 \text{ si } a = 1 \Rightarrow Z = 0 \end{array}$$

Fig. 4.1. Análisis de un bit.

$$\begin{array}{rccccccc} & 1 & \dots & 1 & 0 & 1 & \dots & 1 \\ \wedge & x & \dots & x & x & x & \dots & x \\ \hline & x & \dots & x & 0 & x & \dots & x \end{array} \quad \begin{array}{l} \text{Máscara} \\ \text{Dato} \end{array}$$

Fig. 4.2. Puesta a 0 de un bit.

$$\begin{aligned} 1 \wedge x &= x, \quad \forall x \\ 0 \wedge x &= 0, \quad \forall x \end{aligned}$$

Para aplicarlas, se realiza una operación AND del dato que se quiere analizar con una máscara que tenga un 1 en el lugar cuyo bit se quiere probar con el resto de la máscara a 0. El resultado de la operación será 0 si el bit en cuestión es 0, o distinto de 0 si el bit que se quiere probar es 1; por tanto, el resultado de la prueba quedará en el flag Z en forma complementada como se muestra en la figura 4.1. Muchas máquinas tienen esta operación bajo el nombre de **BIT TEST**.

Para poner a 0 un bit de un dato, se aprovechan las mismas propiedades del operador AND. Para aplicarlas, se construye una máscara que tenga a 1 todos los bits excepto el correspondiente al lugar que se quiere borrar que se pone a 0 y se hace un AND de esta máscara con el dato que se quiere tratar; como resultado de la operación tendremos el mismo dato pero con el bit puesto a cero. El proceso se muestra en la figura 4.2. Esta operación puede realizarse con varios bits a la vez, basta poner en la máscara 0 todos los bits que se quieran borrar. Muchos ordenadores poseen esta instrucción con el nombre de **BIT CLEAR** o **BIT RESET**.

Para poner un bit a 1 en un dato, aplicaremos las propiedades de la operación OR (+ o $\vee$):

$$\begin{aligned} 0 \vee x &= x, \quad \forall x \\ 1 \vee x &= 1, \quad \forall x \end{aligned}$$

En este caso, se construye una máscara con un 1 en el lugar del bit que se quiera poner a 1 y 0 en el resto. Se realiza la operación OR entre esta máscara y el dato y el resultado será el mismo dato pero con el bit en cuestión puesto a 1 (figura 4.3). Esta operación puede realizarse con varios bits a la vez poniendo a 1 en la máscara todos los bits que se quieran poner a 1. Muchos ordenadores tienen esta operación con el nombre de **BIT SET**.

Para complementar bits en un dato aplicaremos las propiedades de la operación OR exclusivo o XOR ($\oplus$):

$$\begin{array}{cccccccc} & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & \text{Máscara} \\ \vee & x & \ldots & x & x & x & \ldots & x & \text{Dato} \\ \hline & x & \ldots & x & 1 & x & \ldots & x & \end{array}$$

Fig. 4.3. Puesta a 1 de un bit.

$$\begin{array}{cccccccc} & 0 & \ldots & 0 & 1 & 0 & \ldots & 0 & \text{Máscara} \\ \oplus & x & \ldots & x & x & x & \ldots & x & \text{Dato} \\ \hline & x & \ldots & x & \overline{x} & x & \ldots & x & \end{array}$$

Fig. 4.4. Complemento de un bit.

$$0 \oplus x = x, \quad \forall x$$
$$1 \oplus x = \overline{x}, \quad \forall x$$

Para ello se construye una máscara con un 1 en el bit que se quiere complementar y 0 en el resto de las posiciones. Después de hacer un XOR de la máscara con el dato se tendrá el mismo dato con el bit complementado (figura 4.4). Esta operación también se puede hacer con varios bits a la vez poniendo a 1 en la máscara todos los bits que se quieran complementar. Algunas máquinas tienen esta operación en su juego de instrucciones como **BIT CHANGE**.

Otra utilidad de la operación XOR es la **determinación de los bits diferentes** que hay entre dos datos, para ello se hace un XOR entre ellos y los lugares que queden a 1 son los bits diferentes entre los dos datos. Esto es debido a que la tabla de verdad de XOR es la complementaria a la de la igualdad entre dos bits. Esta propiedad también es útil para **determinar cuándo dos datos son iguales** en todos sus bits: en este caso el OR exclusivo entre ellos nos dará 0 y, en caso contrario, tendremos un número no nulo; este resultado puede luego ser analizado mediante el flag Z. De aquí también se puede deducir que **haciendo la operación XOR de un operando consigo mismo el resultado da 0**. Esta es una forma muy rápida de poner a 0 los registros en algunos procesadores.

Otra propiedad muy útil del operador XOR es que si se opera dos veces con el mismo operando se regenera el dato inicial, es decir,

$$(x \oplus y) \oplus y = x, \forall x, y \qquad [4.1]$$

Esta propiedad es útil en aplicaciones gráficas para mover cursores por pantalla.

Una utilidad directa de las operaciones lógicas es el manejo de conjuntos tal y como se hace en el lenguaje Pascal. Supongamos el conjunto universal,

$$U = \{x_0, x_1, x_2, \ldots, x_{n-1}\}$$

Si A es un conjunto, contenido en este conjunto universal, puede representarse internamente en la memoria de la máquina mediante la cadena de n bits

$$a_{n-1}a_{n-2}\ldots a_1a_0$$

en que el valor del bit i indicará la pertenencia del elemento x_i al conjunto A. En otras palabras,

$$a_i = \begin{cases} 1 \text{ si } x_i \in A \\ 0 \text{ si } x_i \notin A \end{cases}$$

A partir de lo anterior resulta bastante evidente que la unión e intersección de dos conjuntos vendrá representada respectivamente por las operaciones OR y AND sobre sus cadenas de bits. De la misma forma el complemento de un conjunto se representa mediante la negación de la cadena de bits que le corresponde.

Mediante esta técnica puede determinarse:

- La **igualdad** de dos conjuntos, analizando si sus cadenas de bits son iguales.
- La **pertenencia** de un elemento x a un conjunto A, realizando un test del bit correspondiente a x en la cadena de bits que representa al conjunto A.
- Si un conjunto es **vacío**, analizando si su cadena de bits es nula.
- La **inclusión** de un conjunto A en otro B; esto podría hacerse analizando en un bucle si todos los bits activados en la cadena correspondiente a A también lo están en la cadena que corresponde a B. Esto sería algo complicado en cuanto a la programación y llevaría un tiempo de cálculo desproporcionado. Sin embargo, se puede realizar la misma comprobación de una forma más simple aplicando alguna de las tres propiedades siguientes:

$$A \subset B \Leftrightarrow A \cup B = B, \quad A \subset B \Leftrightarrow A \cap B = A \quad o \quad A \subset B \Leftrightarrow A \cap \overline{B} = \emptyset$$

  Evidentemente, comprobar cualquiera de estas propiedades se reduce a una operación lógica y una comparación, lo que resulta mucho más simple que el procedimiento anterior.

Curiosamente, la tarea de inicializar la cadena de bits correspondiente a un conjunto es más complicada que las operaciones sobre los conjuntos y depende de cada caso. Normalmente hay que construir una máscara individualizada para cada elemento con el fin de poner a 1 los bits que correspondan en la cadena.

Dentro de las operaciones lógicas también deben considerarse los **desplazamientos**, que pueden definirse mediante la expresión

$$d_i \leftarrow d_{i+k}$$

donde d_i representa a cada uno de los bits del operando y k es el número de lugares que se desplaza. En función del signo de k, el desplazamiento puede ser a la izquierda (k negativo) o a la derecha (k positivo), donde se está suponiendo que los bits se numeran de derecha a izquierda.

Los desplazamientos pueden ser de tres tipos: **lógicos**, **aritméticos** y **rotaciones o desplazamientos circulares** dependiendo del bit entrante. El desplazamiento lógico a la derecha difiere del aritmético en que el bit entrante es, en éste último, el mismo bit de mayor peso (bit de

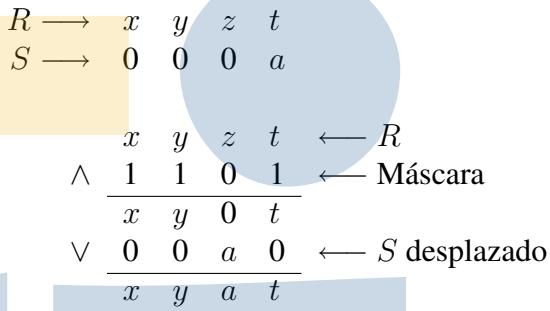

Fig. 4.5. Empaquetamiento de caracteres.

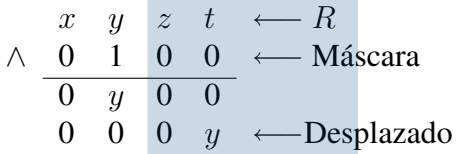

Fig. 4.6. Desempaquetamiento de caracteres.

signo), mientras que en aquél es siempre un 0; Así se consigue que los desplazamientos aritméticos sean equivalentes a multiplicaciones (izquierda) y divisiones (derecha) por 2. En las rotaciones el valor del bit entrante es el mismo que el del saliente.

Una aplicación en que se combinan muchas de las operaciones lógicas anteriores es el empaquetamiento y desempaquetamiento de caracteres. Imaginemos la situación mostrada en la figura 4.5 en la que tenemos, en un registro R, 4 caracteres x, y, z y t y que, en otro registro S, tenemos un carácter a que queremos que sustituya, por ejemplo, al carácter z. En primer lugar, efectuaremos una operación AND entre el registro R y una máscara con 0 en los lugares ocupados por el carácter z y 1 en el resto, con lo que tendremos los caracteres x, y y t inalterados y los lugares correspondientes al carácter z puestos a 0; después sólo quedará realizar una operación OR entre este resultado y el registro S desplazado convenientemente para que el carácter a quede alineado con el carácter z que queremos sustituir. Una técnica parecida se puede emplear para desempaquetar caracteres (figura 4.6): supongamos que del registro R original del ejemplo anterior queremos extraer el carácter y; para ello construiremos una máscara con 1 en los lugares ocupados por el carácter y y 0 en el resto y luego haremos una operación AND entre esa máscara y el registro R. Después sólo tendremos que desplazar convenientemente el resultado para obtener el carácter deseado.

4.4. Instrucciones de control del flujo de programa

Las instrucciones de control de flujo son las que modifican el secuenciamiento de la ejecución de las instrucciones del programa. En general, el secuenciamiento es implícito, es decir,

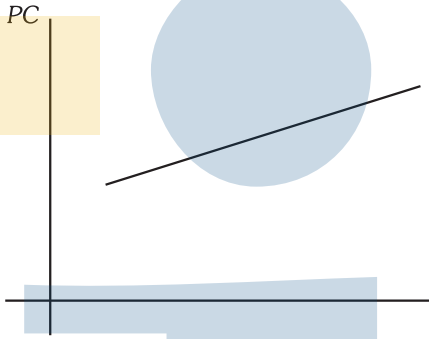

Fig. 4.7. Secuencia lineal de ejecución de instrucciones.

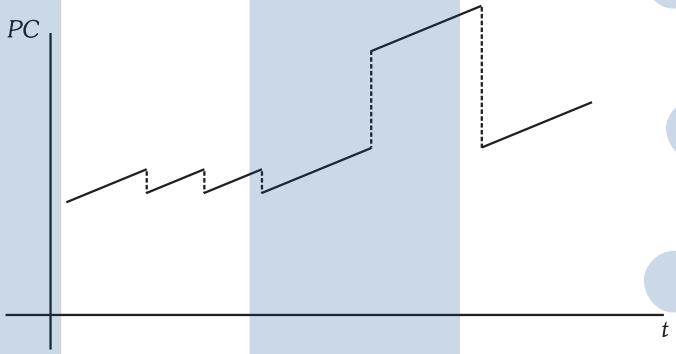

Fig. 4.8. Secuencia con puntos de discontinuidad en la ejecución de instrucciones.

la siguiente instrucción en ejecutarse es la que está físicamente detrás en el programa. Cuando esto no es así es por la acción de una instrucción de control de flujo.

Todas las instrucciones que modifican el flujo de la ejecución manejan el contador de programa. También, si la modificación del flujo de instrucciones se hace de forma condicional, esta condición vendrá dada por los bits de estado (N, Z, V y C). Vemos, por tanto, que para las instrucciones de control de flujo hay dos variables fundamentales: el **contador de programa** y los **bits de condición**.

Las instrucciones de control de flujo suponen discontinuidades en la secuencia lineal de ejecución de las instrucciones (figuras 4.7 y 4.8).

4.4.1. Instrucciones de bifurcación

La mayoría de los programas requieren de los ordenadores en que se ejecutan la capacidad de examinar datos y luego alterar la evolución del programa en función de los resultados de esa comprobación (figura 4.9); esto nos lleva a la necesidad de la existencia de instrucciones de salto o bifurcación que pueden ser de dos tipos: **condicionales** e **incondicionales**. Las bifurcaciones incondicionales se realizan siempre, mientras que las bifurcaciones condicionales se realizan o no en función del valor de uno o varios de los **bits de estado o condición**, simbólicamente, de forma general:

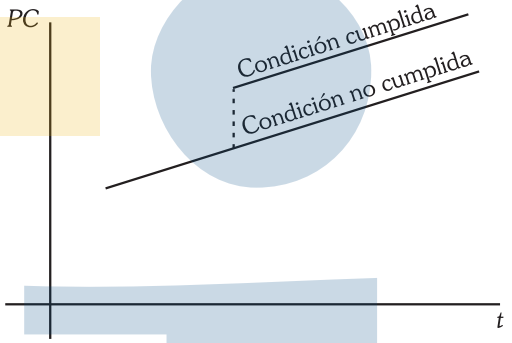

Fig. 4.9. Bifurcación condicional.

```
if (Condición)
   PC ← Dirección de bifurcación;
else
   PC++;
```

En esta representación, $Condición$ simboliza una expresión que involucra a uno o más bits de condición. Los bits de condición más usuales son:

- N: Bit que indica si el resultado de la última operación ha sido **negativo**.
- Z: Bit que indica si el resultado de la última operación ha sido **cero**.
- V: Bit que indica si en la última operación hubo **desbordamiento**.
- C: Bit que indica si en la última operación se produjo **acarreo** o **llevada**.

Algunas máquinas incluyen otros bits de estado adicionales.

Es importante distinguir la diferencia entre los bits C y V: mientras C indica que ha habido llevada en el bit de orden más alto, V señala que el resultado de una operación tiene demasiados bits para ser representado. En las operaciones realizadas en binario natural ambos conceptos coinciden, sin embargo, en aritmética de complemento a 2 son conceptos diferentes ya que la llevada en el bit de orden más alto no es, en este caso, una situación incorrecta y se produce con mucha frecuencia sin que haya desbordamiento.

Veremos a continuación cómo se realiza la detección de desbordamiento para dar el valor correcto al bit V. La operación que puede dar más problemas de desbordamiento es la suma (o derivados, como, por ejemplo, el incremento). En principio, el desbordamiento se produce, en una operación en complemento a 2, cuando se suman dos operandos del mismo signo y el resultado de la suma es de signo contrario. Esto se origina porque, al desbordarse la capacidad del registro, se invade el bit de signo; por otra parte, no habrá peligro de desbordamiento si se suman dos operandos de signo contrario ya que, si ambos operandos caben en un registro su diferencia también cabrá. Estas observaciones nos llevan a dar valor al bit V por una de las siguientes ecuaciones lógicas:

$$V = \overline{(S_1 \oplus S_2)}.(S_1 \oplus S_R)$$
$$V = S_R\,\overline{S_1}\,\overline{S_2} + \overline{S_R}\,S_1\,S_2 \qquad [4.2]$$

Tabla 4.1. Tabla de verdad de la suma para el bit de signo.

Entradas			Salidas		
x_{n-1}	y_{n-1}	c_{n-1}	c_n	s_{n-1}	V
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

En estas ecuaciones S_1, S_2 y S_R son, respectivamente, los signos de ambos operandos y del resultado.

Esta forma de detectar el desbordamiento tiene el inconveniente de que involucra a demasiadas variables. Vamos a tratar de encontrar otro método que nos lleve a una ecuación más simple. Para ello partiremos de la tabla de verdad de la suma que se muestra en la tabla 4.1. Aplicando esta tabla al bit de signo (bit de orden $n-1$), que en las operaciones en complemento a 2 se trata igual que los demás, se puede deducir de lo anterior, que en las filas de la tabla donde los dos sumandos (x_{n-1} e y_{n-1}) sean iguales y el resultado (s_{n-1}) sea diferente tendremos desbordamiento ($V = 1$). De la tabla podemos deducir que el valor del bit V viene dado por:

$$V = \overline{x}_{n-1}\overline{y}_{n-1}c_{n-1} + x_{n-1}y_{n-1}\overline{c}_{n-1}$$

Esto podría ser una forma de determinar por hardware la existencia de desbordamiento; sin embargo, a la vista de la tabla, puede deducirse que se produce desbordamiento si la llevada entrante al bit de signo es diferente de la saliente, es decir:

$$V = c_{n-1} \oplus c_n$$

Esta ecuación es más simple que la anterior para detectar el desbordamiento por hardware. Este método tiene, además, la ventaja de que también es válida para otras operaciones como, por ejemplo, los desplazamientos.

En cuanto al posicionamiento de los demás bits de estado, es bastante simple: N y C se detectan directamente a partir de la unidad aritmética y Z se consigue mediante una puerta NOR cuyas entradas sean todos los bits del resultado.

Los conjuntos de instrucciones de las máquinas suelen facilitar instrucciones auxiliares para posicionar los bits de estado sin alterar el resto de los registros de la máquina, estas instrucciones son `TST` (compara con 0) y `CMP` (compara dos números restándolos). Suele ser conveniente utilizar estas instrucciones inmediatamente antes de las instrucciones de bifurcación condicional, ya que si, entre las instrucciones de comparación y las de bifurcación, se insertan otras instrucciones los bits de estado cambiarán y la comparación quedará sin efecto. Hay que señalar que en algunos procesadores RISC (por ejemplo, SPARC) la mayoría de las instrucciones

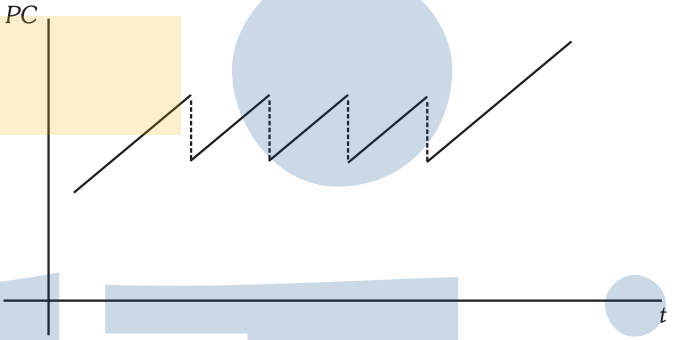

Fig. 4.10. Iteraciones.

no cambian los *flags*, sólo los cambian instrucciones específicas. Esto es así porque, en este tipo de procesadores, es frecuente tener que cambiar el orden de ejecución de las instrucciones. En estos procesadores no es necesario que las instrucciones de comparación estén inmediatamente antes que las de bifurcación.

Atendiendo al modo de direccionamiento utilizado para expresar la dirección de continuación de la ejecución, las bifurcaciones pueden ser **absolutas** y **relativas**. Son bifurcaciones absolutas aquéllas en que se especifica la dirección de la instrucción donde debe continuar la ejecución del programa, normalmente, a este tipo de bifurcaciones se les denomina **saltos**. Se llaman bifurcaciones relativas a aquéllas en que se especifica la diferencia entre el nuevo valor del contador de programa, es decir la dirección donde debe continuar la ejecución, y su valor actual. Simbólicamente una bifurcación relativa funcionaría así:

```
if (Condición)
  PC ← PC + Desplazamiento;
else
  PC + +;
```

En las máquinas en que las instrucciones deben estar obligatoriamente alineadas, el desplazamiento suele multiplicarse por el tamaño de la palabra básica de instrucción para permitir rangos mayores en este tipo de bifurcaciones. Las bifurcaciones relativas facilitan la relocalización de los programas, por ello deben utilizarse siempre que sea posible.

4.4.2. Iteraciones

Es muy frecuente que se necesite ejecutar un grupo de instrucciones cierto número de veces (figura 4.10), por ello, la mayoría de las máquinas tienen instrucciones específicas para ello. Un método para realizar iteraciones se basa en poner un valor inicial en un registro para luego pasar a ejecutar el código de la iteración, la última instrucción del bucle actualizará el valor del registro y comprobará si se cumple la condición de terminación, si es así se ejecutará la siguiente instrucción y si no se comienza una nueva iteración. Esta forma de actuar se caracteriza por hacer la comprobación de la condición al final del bucle por lo que éste se ejecuta una vez como mínimo en todos los casos, incluso aunque la condición de terminación se cumpla antes de

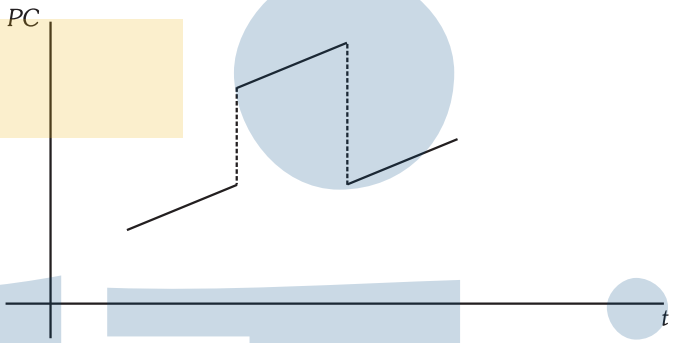

Fig. 4.11. Llamada a un procedimiento.

entrar. Se puede analizar también la condición al principio del bucle pero esto emplea algunas instrucciones más. En lenguaje máquina lo que se suele hacer es poner en un registro el número de veces que se debe iterar, antes del comienzo del bucle. Al final del bucle se decrementa el registro y se compara con 0: si es 0 se continúa con la siguiente instrucción y si no, se ejecuta otra vez el bucle desde el comienzo. Normalmente, la mayoría de las máquinas tienen alguna instrucción para realizar estas dos últimas operaciones con una sola instrucción (`SOB` en el PDP-11, `SOBGTR` y otras en el VAX, `DJNZ` en el Z-80, `LOOP` en el 8086, etc.). Muchas máquinas, además de las instrucciones para implementar bucles, tienen también instrucciones para realizar operaciones con bloques y cadenas, lo que evita la ejecución de muchos bucles.

4.4.3. Procedimientos

La técnica principal para estructurar programas es el uso de **procedimientos** que, según el lenguaje, también se llaman **subprogramas**, **subrutinas** o **funciones**. Desde el punto de vista de la máquina, una llamada a un procedimiento altera el flujo de instrucciones como un salto con la diferencia importante de que el procedimiento devuelve el control a la instrucción siguiente a la llamada una vez que se ha concluido (figura 4.11). Sin embargo, desde el punto de vista de un programador de lenguaje de alto nivel, la llamada a un procedimiento puede considerarse como la ejecución de una instrucción de otro nivel; desde este punto de vista, una llamada a un procedimiento puede considerarse como una simple instrucción, aunque pueda ser bastante complicada.

Definiremos un **procedimiento** como una *secuencia de instrucciones que realiza una tarea y a la que se puede llamar desde diversos puntos del programa*.

Uno de los problemas planteados por los procedimientos es el retorno a la siguiente instrucción a la llamada: debe tenerse previsto un lugar seguro para guardar la dirección de la siguiente instrucción (**dirección de retorno**) ya que, al pasar el control al procedimiento, esta dirección desaparece del PC. Puede haber diversas soluciones: la más elemental es reservar un registro o dirección de memoria para guardar la dirección de retorno; este método tiene el inconveniente de que, si el procedimiento llamara a otro (**anidamiento**), esta llamada haría que se perdiera la dirección de retorno de la primera. Una mejora de este método consiste en que la instrucción de llamada a procedimiento almacene la dirección de retorno en la primera palabra del procedimiento, estando la primera instrucción ejecutable en la palabra siguiente; el procedimiento

podría retornar mediante una instrucción de salto con direccionamiento indirecto a la primera palabra del procedimiento. Con este método, el procedimiento podría llamar a otros, ya que cada uno tiene sitio para una dirección de retorno. Si el procedimiento se llamara a sí mismo (**recursión**), este esquema fallaría ya que la segunda dirección de retorno destruiría a la primera. La recursión es una característica muy importante por lo que este método no se emplea. Tampoco sería válido este esquema para la **recursión indirecta**, es decir en el caso de que un procedimiento A llamara a otro B y éste llamara a su vez a A.

La mejor solución al problema de la dirección de retorno, y por otra parte la más frecuente, es guardar la dirección de retorno en una pila antes de efectuar el salto al procedimiento; cuando el procedimiento concluya sacará la dirección de retorno de la pila y la pondrá en el registro contador de programa. Este método no plantea ningún problema a la hora de la recursión ya que salva la dirección de retorno por encima de las anteriores, evitando su destrucción. Un problema similar se plantea para guardar el **estado del procesador** que está materializado en sus registros; éstos también deben guardarse en la pila, pero existen dos variantes en cuanto al momento de hacerlo: puede ser el programa que llama el que guarde los registros (*caller-saving*) o puede ser el procedimiento llamado quien lo haga (*called-saving*). Existe una solución intermedia proporcionada por algunas arquitecturas en que no hay que añadir instrucciones para guardar los registros puesto que la misma instrucción de llamada lo hace, este es el caso de la arquitectura VAX en que la primera palabra del procedimiento indica los registros que el procedimiento usa y la instrucción de llamada los guarda.

Otros problemas planteados por los procedimientos son el paso de parámetros y el almacenamiento de sus variables. Estos problemas se resuelven reservando en la pila lugar también para los parámetros y las variables. Toda la información que se almacena en la pila cuando se llama a un procedimiento se llama **trama de pila** o **registro de activación del procedimiento**. Podrían arbitrarse otras soluciones para el paso de parámetros y el almacenamiento de las variables del procedimiento (por ejemplo almacenarlas en registros) pero estos métodos no funcionan correctamente para procedimientos recursivos (la mayoría no lo son, y estos métodos pueden emplearse). En los procesadores con ventanas de registros sí que pueden almacenarse los parámetros y variables del procedimiento en registros, aunque sean recursivos, ya que las ventanas hacen la función de la pila.

Si se guarda demasiada información en la trama de pila de los procedimientos resulta conveniente disponer de algún apuntador auxiliar para referirse a las informaciones de la trama de pila sin recurrir al apuntador de pila. Incluso existen computadores que poseen varios apuntadores para estructurar la información de la trama de pila. La misión esencial de este tipo de apuntadores es aportar la posibilidad de direccionar los argumentos y las variables del procedimiento mediante desplazamientos referidos al apuntador (que apunta a un lugar fijo de la trama de pila de cada procedimiento). Este apuntador suele llamarse **apuntador de trama** (*frame pointer*, FP). Con todos estos condicionantes la trama de pila tomará la forma de la figura 4.12.

La secuencia de operaciones realizadas para llamar a un procedimiento (**secuencia de llamada**) es la siguiente:

1. El programa que llama al procedimiento reserva en la pila lugar para los argumentos.
2. La instrucción de llamada guarda en la pila la dirección de retorno y también se guarda el valor del apuntador de trama del procedimiento anterior.

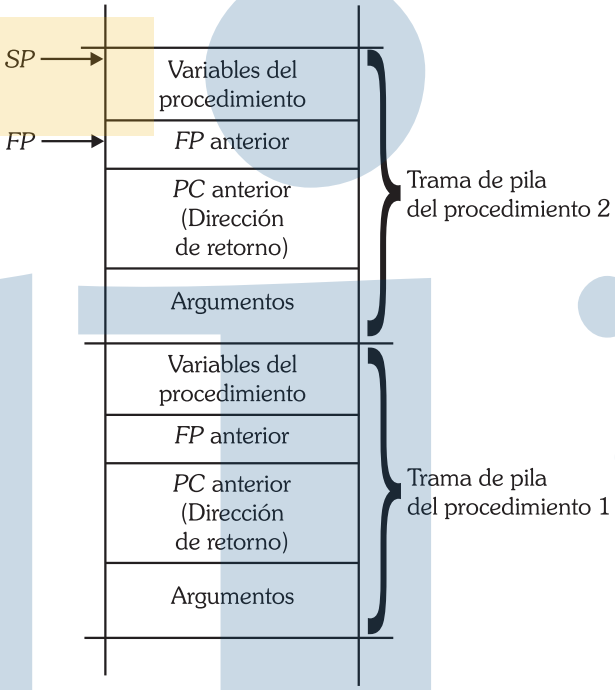

Fig. 4.12. Trama de pila de dos procedimientos anidados.

3. El procedimiento reserva en la pila lugar para sus variables.

Debe observarse que el orden de esta secuencia de acciones no puede alterarse y este orden es el que causa el aspecto de la trama de pila (figura 4.12). Hay que tener también en cuenta que en algún momento también habrá que guardar los registros del procesador (esto no está representado en la figura 4.12).

En el momento del retorno se liberará el espacio ocupado en la pila por el procedimiento y se cargará el contador de programa con la dirección de retorno guardada en la pila. Asimismo se restaurará el contenido de los registros.

La estructuración de los programas exige muchas llamadas a procedimientos lo que hace de ellas uno de los puntos más críticos a la hora de mejorar el rendimiento de la máquina; esto hace que sea conveniente que las operaciones anteriores se realicen de la forma más eficiente posible.

Una alternativa para mejorar la velocidad en las llamadas a procedimientos consiste en expandir el código del procedimiento en el lugar donde ocurre la llamada. Este método, que es empleado a veces por algunos compiladores, se denomina **integración del procedimiento** o **procedimiento en línea** y puede usarse allí donde el tiempo de ejecución sea crítico. Sin embargo el método tiene el inconveniente de que aumenta el espacio de memoria necesario para el código, por ello puede ser una buena alternativa para procedimientos cortos.

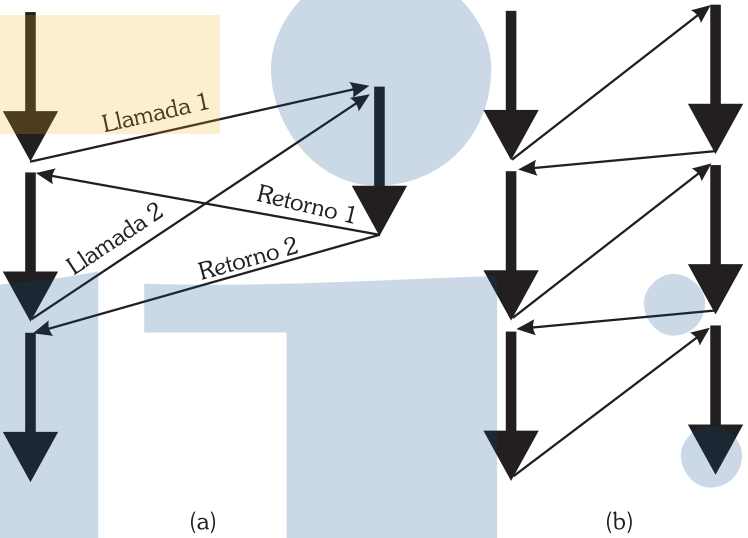

Fig. 4.13. Flujo de instrucciones en una llamada a procedimiento (a) y entre corrutinas (b).

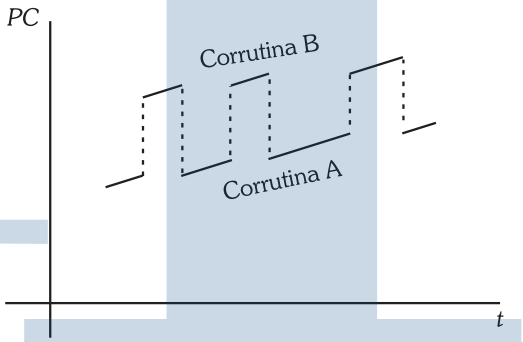

Fig. 4.14. Corrutinas.

4.4.4. Corrutinas

En la secuencia ordinaria de llamada a procedimiento la distinción entre el procedimiento que llama y el llamado es clara. Consideremos el caso de dos procedimientos que se llamen mutuamente. A primera vista esta situación se podría considerar simétrica. La asimetría procede del hecho de que, cuando se pasa el control del procedimiento que llama al procedimiento llamado, éste comienza a ejecutarse desde el principio; sin embargo, cuando se produce el retorno, la ejecución del procedimiento que hizo la llamada sigue en la instrucción siguiente. Si se llamara más veces al procedimiento, éste comenzaría a ejecutarse nuevamente desde el principio (situación mostrada en la figura 4.13 (a)). La asimetría se acentúa si analizamos el proceso llevado a cabo a nivel máquina: en el momento de la llamada el valor del contador de programa pasa a la pila y cuando el procedimiento retorna el proceso es justamente el contrario; además las instrucciones que realizan ambas funciones son distintas. En algunas ocasiones es útil tener dos procedimientos que se llamen el uno al otro como un procedimiento (sin retorno) sin comenzar la ejecución desde el principio (figura 4.13 (b)). Dos procedimientos que llamen

uno a otro de esta forma se denominan corrutinas (figura 4.14). La forma de llevar a cabo en la práctica una llamada a una corrutina es intercambiar los contenidos del contador de programa y de la cima de pila, sin variar el apuntador de pila, ya que se saca un dato de la pila para meter otro; de esta forma se intercambia el control entre las dos rutinas.

4.4.5. Desvíos o excepciones (traps)

Un desvío es un *tipo especial de llamada automática a procedimiento iniciada por alguna condición debida al programa*.

Hay que decir que la nomenclatura de la literatura técnica cambia mucho de unos fabricantes a otros de forma que unos llaman excepciones a lo que otros llaman interrupciones e incluso algunos llaman a ambas cosas de la misma forma. Aquí trataremos de dar más importancia a los conceptos que a los nombres.

Los desvíos normalmente se deben a condiciones importantes aunque no muy frecuentes. El caso más usual es la condición de desbordamiento en operaciones aritméticas (TRAPV). Si después de una operación se produce desbordamiento el microprograma lo detecta y bifurca a una rutina de tratamiento de ese error. Usar esta técnica es más rápido que hacer una bifurcación explícita en función de V después de cada instrucción aritmética ya que la dirección de bifurcación es siempre la misma y el análisis de la condición, en el caso del desvío, lo realiza automáticamente el microprograma después de cada instrucción aritmética.

También provocan desvíos la división por 0, la lectura de un código de operación indefinido, el acceso a una palabra mal alineada en los ordenadores en que la alineación es obligatoria, etc. Otro tipo de desvíos se producen por causas debidas al sistema operativo tales como faltas de página, errores de protección de segmentos, etc. También pueden existir desvíos para funciones de depuración como puntos de ruptura, ejecución paso a paso, etc.

Los desvíos son **recuperables** si, después de tratarse con la rutina de excepción adecuada, la ejecución puede continuar en la siguiente instrucción, y son **no recuperables** si la ejecución del programa tiene que detenerse; en este caso, habitualmente, se devuelve el control al Sistema Operativo de la máquina.

Un problema de los desvíos es encontrar la dirección del procedimiento, llamado **rutina de tratamiento de excepción**. La vía mas usual de obtener esta dirección es la **vectorización** que consiste en numerar todas las causas de desvío y tener una tabla en memoria con todas las direcciones de las rutinas de tratamiento; estas direcciones se denominan **vectores** y a la tabla anterior se la denomina **tabla de vectores de excepción**. Si la tabla de vectores comienza en una dirección A y cada vector ocupa t palabras, la dirección correspondiente a la excepción n vendrá dada por:

$$\text{Dirección de la rutina de tratamiento del desvío } n = (A + nt)$$

En esta expresión el paréntesis denota el contenido de la dirección de su interior. El cálculo lo realiza internamente el procesador cada vez que se produce una excepción, normalmente es un cálculo rápido ya que las constantes A e t facilitan la tarea: normalmente t es una potencia de 2 y A muchas veces es 0, por lo que la operación se reduce a un desplazamiento.

4.4.6. Interrupciones

Las interrupciones son *llamadas automáticas a procedimiento no debidas al programa sino a una causa exterior*.

La diferencia entre las interrupciones y los desvíos es que éstos son provocados por el mismo programa mientras que las interrupciones son provocadas por causas externas de forma totalmente asíncrona.

Normalmente las causas de interrupción están relacionadas con las operaciones de entrada y salida. Una interrupción detiene el programa en curso y transfiere el control al procedimiento de tratamiento de la interrupción denominado **rutina de servicio de interrupción**; cuando esta rutina concluye se debe devolver el control al proceso interrumpido que debe continuar su ejecución en el mismo estado en el que estaba cuando se produjo la interrupción. Esto significa que se deben salvar el estado del procesador (es decir el contenido de los registros) antes de comenzar la ejecución de la rutina de servicio y restaurarse al finalizar ésta.

Las interrupciones son necesarias cuando las entradas o salidas pueden desarrollarse en paralelo con la ejecución de instrucciones en el procesador. Esto normalmente es así debido a que, mientras un dispositivo de entrada/salida efectúa una sola operación, el procesador puede ejecutar muchas instrucciones convencionales. Los sistemas de interrupciones permiten que la CPU funcione concurrentemente con los dispositivos de entrada y salida, siendo las interrupciones el sistema de comunicaciones entre ambos procesos para que el procesador sepa cuando el dispositivo de entrada/salida ha concluido.

El mecanismo de vectorización descrito en el apartado anterior también es válido para encontrar la dirección de la rutina de servicio de interrupción. En las interrupciones la diferencia radica en que el número de la interrupción lo manda el propio dispositivo a través del bus de datos.

4.4.7. Instrucciones de control de flujo en el VAX

En este apartado estudiaremos un caso concreto de instrucciones de control de flujo: las correspondientes al VAX (Baase, 1983). Esta máquina tiene diversas clases de instrucciones de control de flujo:

Instrucciones de bifurcación

El VAX tiene dos instrucciones de bifurcación incondicional: `BRB` y `BRW`; ambas instrucciones se diferencian sólo en el tamaño del desplazamiento (1 o 2 bytes): para bifurcaciones en que la dirección de destino se encuentre cerca de la dirección actual (menos de 128 bytes) se emplea `BRB`, en caso contrario se emplea `BRW`. En cuanto a las bifurcaciones condicionales todas tienen un desplazamiento de 8 bits, esto limita algo su uso y obliga en ocasiones a emplear una programación menos estructurada en las bifurcaciones condicionales ya que es necesaria la instrucción `BRW`. En las instrucciones de bifurcación del VAX sólo se especifica el desplazamiento, sin el byte de especificación de operando convencional en la forma modo-registro. También existen instrucciones específicas para bifurcar en función del valor del bit de orden más bajo de un operando (`BLBS`, *branch if low bit set* y `BLBC`, *branch if low bit clear*: bifurcar si el bit

Tabla 4.2. Instrucciones de bifurcación condicional en el VAX.

Nemónico	Nombre (Bifurcar si...)	Condición
`BEQL`	igual	$Z = 1$
`BNEQ`	distinto	$Z = 0$
`BGTR`	mayor	$N = 0 \wedge Z = 0$
`BLEQ`	menor o igual	$N = 1 \vee Z = 1$
`BGEQ`	mayor o igual	$N = 0$
`BLSS`	menor	$N = 1$
`BGTRU`	mayor	$C = 0 \wedge Z = 0$
`BLEQU`	menor o igual	$C = 1 \vee Z = 1$
`BGEQU`	mayor o igual	$C = 0$
`BLSSU`	menor	$C = 1$
`BVS`	V activado	$V = 1$
`BVC`	V desactivado	$V = 0$
`BCS`	C activado	$C = 1$
`BCC`	C desactivado	$C = 0$
`BLBS`	bit 0 activado	
`BLBC`	bit 0 desactivado	

de orden más bajo es 1 ó 0 respectivamente), en estas instrucciones, además del destino de la bifurcación, hay que especificar el operando en que se analiza el bit en cuestión.

En la tabla 4.2 puede verse un resumen de las instrucciones de bifurcación condicional del VAX. Los nemónicos terminados en `U` corresponden a condiciones sobre números en binario natural (sin signo) y las restantes a condiciones sobre números en complemento a 2 (con signo).

Existe una instrucción de salto incondicional (`JMP`: *jump*, salto). La diferencia entre esta instrucción y las bifurcaciones incondicionales reside en que en `JMP` se puede usar cualquier modo de direccionamiento para la dirección de destino y sin embargo en las instrucciones de bifurcación esa dirección especifica siempre mediante un desplazamiento.

Instrucciones para el control de iteraciones

En el VAX existen diversas instrucciones para el control de iteraciones, unas se basan en el decremento del índice (`SOB`xxx: *subtract one and branch if* xxx, restar uno y bifurcar si se cumple la condición xxx) y otras en su incremento (`AOB`xxx: *add one and branch if* xxx, sumar uno y bifurcar si se cumple la condición xxx). Existe también una instrucción (`ACB`x: *add, compare and branch*, sumar, comparar y bifurcar) que permite un incremento diferente de 1 que, además, puede ser tanto positivo como negativo e incluso no entero. En la tabla 4.3 se muestran las instrucciones de control de iteraciones del VAX.

Llamadas a subrutinas y procedimientos

En el VAX se puede distinguir entre llamadas a subrutinas y llamadas a procedimientos: la diferencia estriba en que una llamada a subrutina simplemente guarda la dirección de retorno en

Tabla 4.3. Instrucciones para el control de iteraciones del VAX.

Nemónico	Operandos	Operación
SOBGTR	índice, destino	if (–índice>0)goto destino;
SOBGEQ		if (–índice>=0)goto destino;
AOBLEQ	límite, índice, destino	if (++índice<=límite)goto destino;
AOBLSS		if (++índice<límite)goto destino;

ACBx	límite, incremento, índice, destino	índice ← índice+incremento;
		if (incremento>=0)
		{if (índice<=límite)goto destino;}
		else
		if (índice>=límite)goto destino;

la pila y bifurca. Podríamos decir que una llamada a subrutina es una bifurcación con retorno; sin embargo, una llamada a procedimiento es más complicada porque exige pasar los parámetros, guardar el contenido de los registros, etc. Para remarcar más la diferencia entre ambos conceptos, a las subrutinas se les llama a veces subrutinas internas ya que se consideran parte del programa que las llama.

Las instrucciones para llamar a una subrutina son BSBB y BSBW que se diferencian por el tamaño del desplazamiento (1 o 2 bytes). También existe la instrucción RSB (return from subroutine, retorno de subrutina) que recoge la dirección guardada anteriormente en la pila, por BSBB o BSBW, y bifurca a ella.

En cuanto a las llamadas a procedimientos, en el VAX hay dos instrucciones para efectuarlas: CALLS y CALLG. La diferencia entre ambas radica en la forma de pasar los argumentos: mientras que en CALLS se pasan a través de la pila con el fin de facilitar la recursión, en CALLG los argumentos se pueden pasar en otra zona de memoria.

La sintaxis de estas instrucciones es:

CALLS número de argumentos, dirección del procedimiento

CALLG dirección de la lista de argumentos, dirección del procedimiento

En lenguaje ensamblador las direcciones que aparecen es estas instrucciones se representan mediante etiquetas.

Si se utiliza la instrucción CALLS los argumentos deben ponerse previamente en la pila en orden inverso mediante las instrucciones PUSHL (argumento por valor) o PUSHAx (argumento por referencia). Sin embargo, si se emplea CALLG la lista de argumentos debe definirse previamente de la siguiente forma:

```
Etiqueta:   .LONG      Número de argumentos
            .ADDRESS   Lista de etiquetas de los argumentos
```

Puede ocurrir que la dirección de alguno de los argumentos no se conozca al escribir el programa (por ejemplo, porque sea producto de un cálculo), en este caso, se le pone una etiqueta independiente a esa dirección y, en ejecución, una vez conocido el valor de la dirección se transfiere mediante una instrucción MOV. Es necesario comentar que no es conveniente poner .LONG en lugar de .ADDRESS ya que éste genera código independiente de la posición y aquél

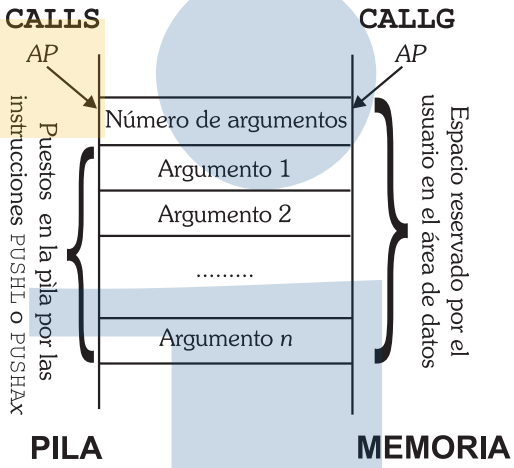

Fig. 4.15. Lista de argumentos de un procedimiento en un VAX.

no.

Ejemplo 4.1: *Supongamos que tenemos una lista con tres argumentos con uno de ellos no conocido en el momento de escribir el programa. La lista de argumentos tendría la forma:*

```
Lista_argum:   .LONG      3
               .ADDRESS   Etiqueta_argumento_1
Desconocido:   .ADDRESS   0
               .ADDRESS   Etiqueta_argumento_3
```

En el programa, si la dirección desconocida es producto de un cálculo cuyo resultado queda, por ejemplo, en R3, sencillamente pondremos

```
MOVL R3, Desconocido
```

○

Tanto si la llamada a procedimiento se realiza con `CALLS` como con `CALLG` la lista de argumentos tiene la forma mostrada en la figura 4.15. La diferencia entre ambos casos radica en que la lista de argumentos en el caso de `CALLS` reside en la pila y en el caso de `CALLG` en cualquier lugar de memoria. En los dos casos el registro AP apunta al comienzo de la lista de argumentos. También existe la diferencia en cuanto a la forma de construir la lista de argumentos: mientras en el caso de `CALLS` los argumentos se ponen mediante las instrucciones `PUSHL` y `PUSHA`x y el número de parámetros lo pone la misma instrucción `CALLS` (porque es uno de sus operandos), en el caso de `CALLG`, la lista la construye el usuario incluyendo el número de parámetros. Desde el punto de vista del procedimiento no influye que la llamada se haya realizado mediante `CALLS` o `CALLG`. De hecho un mismo procedimiento puede llamarse de las dos formas (salvo que sea recursivo). En ambos casos se hace referencia a los argumentos a través del apuntador de argumentos (AP): `4(AP)` el primero, `8(AP)` el segundo, etc. si los argumentos se pasan por valor o `@4(AP)` el primero, `@8(AP)` el segundo, etc. si se pasan por referencia.

La trama de pila para las llamadas a procedimientos del VAX se muestra en la figura 4.16. Los pasos necesarios para estas llamadas son los siguientes:

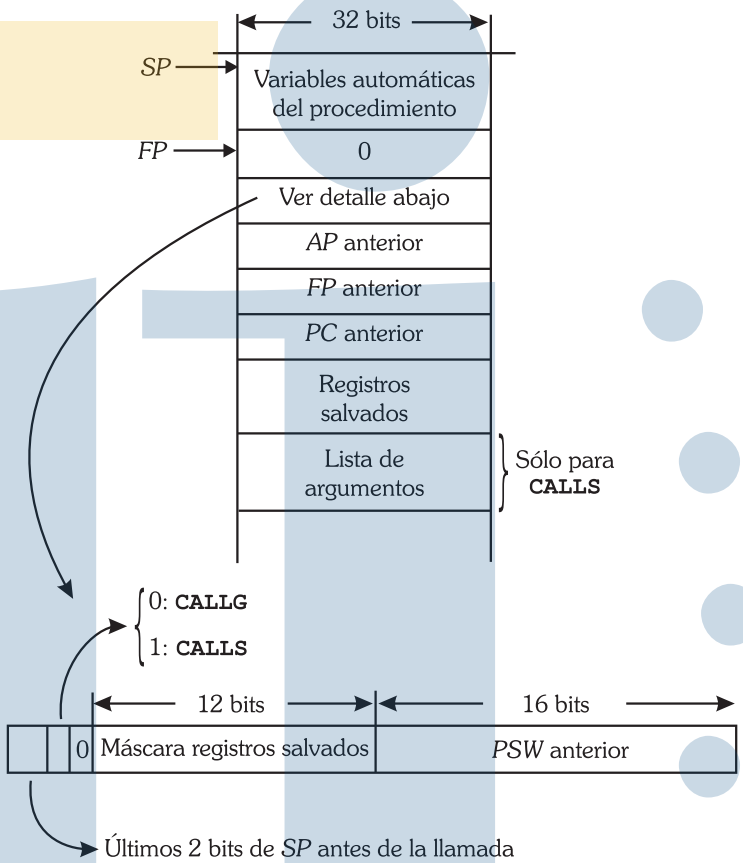

Fig. 4.16. Trama de pila de un procedimiento en el VAX.

1. Alinear la pila a doble palabra. Esto se consigue poniendo a 0 los dos últimos bits del apuntador de pila. El valor anterior de estos bits se guarda en un almacenamiento intermedio.

2. Salvar el contenido de los registros comprendidos en la máscara situada en la primera palabra del procedimiento. Esta máscara la construye el compilador. En lenguaje ensamblador los registros que se deben salvar antes de entrar en el procedimiento se ponen en el directivo `.ENTRY`. A partir de esta información el ensamblador construye la máscara. Por ejemplo, si un procedimiento altera los registros $R2$ y $R3$ en su cabecera pondremos:

```
.ENTRY Nombre del procedimiento,^M<R2, R3>
```

3. Guardar en la pila los registros PC (contador de programa), FP (apuntador de trama) y AP (apuntador de argumentos).

4. Guardar en la pila la PSW (palabra de estado), la máscara de registros salvados, los dos bits menos significativos del apuntador de pila (SP), antes de alinearlo, y un bit que indica si la llamada se ha efectuado con `CALLG` (0) o `CALLS` (1).

5. Poner a 0 la palabra de estado.

6. Poner una doble palabra con 0 en la pila.

7. Copiar el apuntador de pila (SP) en el apuntador de trama (FP).

8. Poner la dirección de la lista de argumentos en el registro AP (apuntador de argumentos). En el caso de `CALLG` esta dirección es uno de los operandos; sin embargo, en el caso de `CALLS`, antes de salvar los registros (paso 2), se salva el número de argumentos y, en ese momento, el SP es almacenado en un registro temporal para depositarlo ahora en el registro AP.

9. Cargar la dirección del procedimiento + 2 en el contador de programa (el hecho de sumar dos a la dirección del procedimiento se debe a que la primera palabra del procedimiento es la máscara de registros).

De forma análoga los pasos realizados por la instrucción `RET` (retorno de procedimiento) son los siguientes:

1. Copiar en SP el contenido de $FP + 4$. Esto elimina todas las informaciones puestas en la pila después de la trama del procedimiento (variables o informaciones apiladas que no se han eliminado antes).

2. Salvar la cima de pila (PSW, máscara de registros, etc) en un almacenamiento intermedio.

3. Restaurar los registros AP, FP y PC.

4. Restaurar los registros de uso general a partir de la máscara guardada en el almacenamiento intermedio.

5. Se restauran los bits más bajos del SP (del almacenamiento intermedio).

6. Se restaura la palabra de estado.

7. Si se usó `CALLS` se incrementa el apuntador de pila en $4(n + 1)$, siendo n el número de argumentos que radica en la pila encabezando la lista de argumentos, esto se realiza para eliminar los argumentos de la pila.

4.5. Relación entre el lenguaje máquina y los lenguajes de alto nivel

En esta sección estudiaremos como se almacenan las variables y estructuras a las que se hace referencia en los lenguajes de alto nivel. También estudiaremos qué modos de direccionamiento son más convenientes para cada caso. Esta información será muy útil para implementar cualquier tipo de compilador.

4.5.1. Clases de variables

En un programa de alto nivel, e incluso también en lenguaje ensamblador, existe un concepto denominado **visibilidad**. Este concepto se refiere al *ámbito de validez del* ***nombre*** *de las variables*. Atendiendo a la visibilidad, los nombres de las variables pueden ser de dos tipos:

Globales: son nombres de variable reconocidos en todo el programa.

Locales: son nombres de variable que sólo se reconocen en una parte del programa, normalmente uno o más procedimientos.

El ámbito de visibilidad de los nombres de las variables es un concepto que maneja solamente el compilador, concretamente en la tabla de símbolos haciendo que un símbolo sólo sea reconocido en su ámbito de validez, y no transciende al código ejecutable del programa. Sin embargo, hay otro concepto similar que es el de **vida de la variable** que se define como el *tiempo que transcurre desde que una variable se comienza a usar hasta que se la menciona por última vez* o, lo que es lo mismo, el *tiempo en que es necesario que la variable tenga espacio reservado en memoria*. Según este criterio las variables pueden ser de tres tipos:

Permanentes: son *variables cuya vida es el tiempo total de ejecución del programa*. Estas variables se almacenan en una zona de memoria accesible desde cualquier punto del programa denominada **área global**. Se accede a este área con direccionamiento relativo o direccionamiento por base y desplazamiento, tomando como base un registro que apunte a la dirección de comienzo del programa. Un tipo especial de variables de este tipo son las variables **estáticas** que son *variables permanentes cuyo nombre es local a uno o varios procedimientos*. Las variables estáticas conservan su valor aunque el procedimiento retorne ya que también se almacenan en el área global.

Automáticas: son *variables cuya vida es, como máximo, el tiempo de ejecución del procedimiento donde se han definido*. Los nombres de estas variables son locales a ese procedimiento. Los parámetros de los procedimientos se consideran variables de este tipo. Las variables automáticas se almacenan en la pila, de esta forma su espacio se reserva cada vez que se llama al procedimiento y es liberado cuando retorna. El acceso a estas variables se realiza mediante direccionamiento indexado al apuntador de trama de procedimiento o *frame pointer*. En el caso de los parámetros pasados por referencia o de las variables automáticas que representen apuntadores, se aplicará el mismo direccionamiento pero en su versión indirecta.

Dinámicas: son *variables cuyo espacio se reserva y libera durante la ejecución de un procedimiento*. Este tipo de variables se emplea para almacenar estructuras de datos que pueden crecer o decrecer en ejecución tales como listas enlazadas, árboles, etc. El lugar de almacenamiento de estas variables se denomina **área dinámica** o *heap* y puede no ser contiguo ya que el sistema operativo va concediendo memoria según el programa la va solicitando y no siempre la memoria está disponible de forma contigua.

Las variables escalares de cualquiera de los tipos anteriores pueden almacenarse en registros del procesador, salvo las variables dinámicas y las variables automáticas en los procedimientos recursivos, incluso en este caso también es posible en procesadores RISC con ventanas de registros como se verá en el apartado 4.6.

La nomenclatura de las clasificaciones anteriores no concuerda exactamente con la de los lenguajes de programación ya que ésta puede cambiar bastante de unos lenguajes a otros.

4.5.2. Direccionamiento de estructuras de datos

Una de las características fundamentales de los lenguajes de alto nivel es la posibilidad de definir y manejar estructuras de datos complejas. Estas estructuras plantean los siguientes problemas:

- Organizar su almacenamiento en memoria.
- Encontrar la dirección de cada uno de sus componentes más simples.

En estas circunstancias el compilador del lenguaje de alto nivel debe producir una secuencia de instrucciones para calcular la dirección del componente deseado.

El acceso a las estructuras de datos más simples (un vector unidimensional, una estructura o *record*, etc) puede requerir un número muy pequeño de instrucciones si se utiliza la potencia de los modos de direccionamiento. Por ejemplo, supongamos que queremos acceder al elemento k de un vector cuyos elementos ocupan 2^n bytes cada uno y están situados consecutivamente en memoria; si el registro Rx contiene la dirección del primer elemento y Ry contiene el índice (k), la dirección del elemento k vendrá dada por

$$Rx + [Ry << n]$$

Si la dirección del primer elemento viene dada por un desplazamiento respecto a otro registro, normalmente el *frame pointer* o el propio PC, la expresión anterior, llamando genéricamente R_{ref} al registro de referencia, se convierte en:

$$R_{ref} + desplazamiento + [Ry << n]$$

Se puede usar direccionamiento indexado para efectuar la suma, lo que nos eliminaría una operación. En el VAX el direccionamiento indexado es tan potente que realiza toda la operación: Si A es la dirección del primer elemento, especificada habitualmente mediante direccionamiento relativo, el elemento k se especifica mediante $A[Ry]$ ya que el direccionamiento indexado del VAX multiplica el índice por el tamaño del operando.

También puede darse el caso de que la dirección del primer elemento venga dada por un apuntador, en cuyo caso hay que añadir un nivel de indirección, con lo que queda:

$$(R_{ref} + desplazamiento) + [Ry << n]$$

En el VAX el direccionamiento indexado soporta cualquier modo como índice, por lo que también se puede aplicar sobre direccionamiento indirecto relativo, en este caso, el elemento k del vector se especifica mediante `@A[Ry]`.

De una manera similar se podría gestionar el acceso a elementos de matrices de varias dimensiones.

Un caso algo menos regular es el direccionamiento de estructuras de datos tipo `record` o `struct`. La irregularidad reside en que el tamaño de cada uno de los campos de una estrucutura de este tipo es diferente. En este caso, la dirección de uno de los campos viene dada por

$$R_{ref} + desplazamiento_{primer\ elemento} + desplazamiento_{campo}$$

En esta expresión, $R_{ref} + desplazamiento_{primer\ elemento}$, representa la dirección del primer elemento y desplazamientocampo, es el desplazamiento del campo deseado respecto al primer elemento. Esto puede hacerse así, o, si el compilador está optimizado, éste puede sumar ambos desplazamientos y utilizar direccionamiento indexado simple. No puede hacerse esta última simplificación en el caso de un apuntador a un `struct` (operador -> del lenguaje C) ya que, en este caso, los dos primeros términos están afectados de una indirección y la dirección del campo buscado es:

$$(R_{ref} + desplazamiento_{primer\ elemento}) + desplazamiento_{campo}$$

Para este caso se adecúa muy bien el direccionamiento indirecto postindexado del microprocesador MC68020.

Una técnica similar puede usarse si una estructura es un campo de otra. En este caso, para obtener la dirección de uno de los campos de la estructura más interna, es necesario sumar tres desplazamientos: el desplazamiento del primer campo de la estructura principal respecto al registro de referencia, el desplazamiento del primer campo de la estructura interna respecto al primer campo de la exterior y el desplazamiento del campo solicitado respecto al primer campo de la estructura interna:

$$R_{ref} + despl_{primer\ elemento\ struct\ principal} + despl_{primer\ elemento\ struct\ interno} + despl_{campo}$$

De la misma forma que en el caso anterior, con un compilador optimizado, se pueden sumar todos los desplazamientos en compilación y aplicar direccionamiento indexado simple. Si, por ejemplo, a la estructura externa se accede a través de un apuntador, la dirección anterior se obtendrá mediante la expresión:

$$(R_{ref} + despl_{primer\ elemento\ struct\ principal}) + despl_{primer\ elemento\ struct\ interno} + despl_{campo}$$

De forma similar se puede actuar si alguno de los otros campos es un apuntador. En este caso también pueden resultar interesantes los direccionamientos indirectos preindexado y postindexado incorporados al microprocesador MC68020.

4.6. Características de los procesadores RISC

Tradicionalmente ha habido una gran diferencia de nivel entre el lenguaje máquina y los lenguajes de alto nivel. Esta diferencia, conocida como **barrera semántica** o **brecha semántica**, puede minimizarse de dos formas: aumentando la complejidad de los compiladores o elevando la complejidad del lenguaje máquina. En general, se ha optado por la segunda vía lo que, a lo largo del tiempo, ha llevado a ordenadores con lenguajes máquina de muchas instrucciones y muy elaboradas pero muy lentos debido a que un microprograma para decodificar muchas instrucciones complejas será más lento que otro microprograma para decodificar un conjunto más

reducido de instrucciones más simples; por otra parte, en este último caso, el control puede ser cableado. Por otro lado, los programas escritos en lenguajes de alto nivel no usan, tanto como se piensa, las instrucciones complejas. Haciendo una estadística sobre programas escritos en lenguajes de alto nivel, se puede llegar a la conclusión de que las instrucciones que más se usan (más de un 90 %) son las asignaciones, las instrucciones condicionales (*if, while*, etc), las llamadas a procedimientos y los bucles (*for*). También hay que tener en cuenta que un porcentaje muy elevado de las asignaciones (alrededor del 80 %) no involucran a operaciones aritméticas, es decir, son asignaciones simples (transferencias), aunque también se debe mencionar que algunas instrucciones condicionales involucran también a operaciones aritméticas. En cuanto a las llamadas a procedimientos, la mayor parte de ellas son llamadas con pocos parámetros e incluso, en la mayoría de los casos, sin ellos.

De una forma análoga si analizamos el código máquina de los programas también se llega a una conclusión similar respecto a los modos de direccionamiento utilizados: entre los direccionamientos inmediato y por desplazamiento se cubre más de un 80 % de las referencias a memoria de la mayoría de los programas (si bien esto depende mucho del compilador utilizado).

Quiere decirse con todo esto que los programas escritos en lenguajes de alto nivel no usan apenas las instrucciones más complejas, aunque en principio pudiera pensarse los contrario. Por ello cabe aquí aplicar el principio de diseño "*mejorar en lo posible los casos más frecuentes*". También hay que tener en cuenta que muchas máquinas están orientadas a un solo tipo de aplicaciones (gestión, diseño asistido, cálculo científico, etc.), esto hace que se puedan encontrar las instrucciones más necesarias para ese tipo de aplicaciones concretas.

Así se llega a los **pasos de diseño** de las máquinas RISC (*Reduced Instruction Set Computer*: Computadores con conjunto reducido de instrucciones):

- Analizar las aplicaciones que van a funcionar en esa máquina para encontrar las operaciones más necesarias para esas aplicaciones (**operaciones clave**).
- Diseñar el conjunto de registros y las comunicaciones entre ellos más apropiados para la ejecución de las operaciones clave.
- Diseñar las instrucciones y modos de direccionamiento para realizar las operaciones clave en el conjunto de registros anteriormente diseñado.
- Añadir nuevas instrucciones sólo si no hacen más lenta la máquina.

Estos pasos de diseño tienen la estructura del llamado diseño descendente, esto quiere decir que se analizan las especificaciones de lo que se necesita y luego se va *descendiendo* hacia los elementos de más bajo nivel necesarios para cumplir esas especificaciones.

Ahora introduciremos el concepto de **ciclo máquina** que es el *tiempo necesario para extraer los operandos de los registros, llevarlos a la ALU para operar con ellos y depositar el resultado en un registro*. Este **tiempo de ciclo** debe ser tan pequeño como sea posible, esto se consigue sacrificando todo lo que incremente la duración del ciclo, por ejemplo, deben eliminarse las instrucciones que tarden en ejecutarse más de un ciclo.

Los principios de diseño de los ordenadores con arquitectura RISC son los siguientes:

- Relativamente **pocas instrucciones** y **pocos modos de direccionamiento**.

Tabla 4.4. Comparación entre las arquitecturas RISC y CISC.

RISC	CISC
Instrucciones sencillas de un solo ciclo	Instrucciones complejas de varios ciclos
Pocas instrucciones y direccionamientos	Muchas instrucciones y direccionamientos
Sólo LOAD y STORE acceden a memoria	Cualquier instrucción puede acceder a memoria
Formato de instrucción fijo	Muchos formatos de instrucción
Control cableado	Control microprogramado
Procesamiento segmentado o *pipe-line*	Procesamiento convencional
Gran número de registros en el procesador	Pocos registros en el procesador
La complejidad reside en el compilador	La complejidad reside en el microprograma

- **Gran cantidad de registros en el procesador** y **uso de ventanas de registros** para las variables de los procedimientos.
- **Formatos de instrucción fijos** y fáciles de decodificar.
- Unidad de control **cableada**.
- **Ejecución de una instrucción en un ciclo de máquina**, mediante procesamiento segmentado o *pipe-line*.
- **Accesos a memoria muy restringidos** con una arquitectura **registro-registro** (recordar el párrafo 1.9). Esto significa que los accesos a memoria se limitan, en general, a las instrucciones de carga y almacenamiento (**LOAD** y **STORE**).
- Uso de **compiladores optimizados** que sepan aprovechar las características anteriores y tengan en cuenta las posibilidades y características de la máquina.

Estas características se resumen en la tabla 4.4 comparándolas con las de las arquitecturas CISC (*Complex Instruction Set Computers*: computadores con conjunto de instrucciones complejo).

Analizaremos brevemente algunas de estas características:

La ejecución de las instrucciones en un solo ciclo de máquina es la propiedad más importante de las máquinas RISC. Esto se consigue por dos razones: en primer lugar, la mayor parte de las instrucciones operan sobre registros, y las que accedan a memoria deben eliminarse del juego de instrucciones, salvo LOAD y STORE; esto evita muchos accesos a memoria que es la operación más lenta; por otra parte, los procesadores RISC utilizan procesamiento segmentado o *pipe-line*, ello significa que se procesan varias instrucciones a la vez en diferente fase, como en una fabricación en cadena, de la forma mostrada en la figura 4.17 donde L significa que la instrucción correspondiente está en la fase de lectura de la instrucción y X significa que se

Ciclo	1	2	3	4	5
Instrucción A	L	X			
Instrucción B		L	X		
Instrucción C			L	X	
Instrucción D				L	X

Fig. 4.17. Procesador segmentado.

Ciclo	1	2	3	4	5	6	7
Instrucción A	L	X					
LOAD		L	M	X			
Instrucción C			L	X			
Instrucción D				L	X		
STORE					L	X	M
Instrucción F						L	X

Fig. 4.18. Procesador segmentado con instrucciones LOAD y STORE.

encuentra en la fase de ejecución. Esto es así porque en este tipo de máquinas hay, al menos, un módulo para procesar cada fase. Como puede verse en la figura, en el ciclo 2 se está leyendo la instrucción B y ejecutando, a la vez, la instrucción A. Con este tipo de procesamiento no se consigue ejecutar una instrucción por ciclo, pero sí, ejecutar n instrucciones en n ciclos, lo que, a efectos prácticos, es equivalente. Sin embargo, aunque los accesos a memoria estén limitados a las instrucciones LOAD y STORE, éstas no encajan en el esquema anterior ya que necesitan una etapa más en el *pipe-line*, para el acceso a memoria adicional (M) como muestra la figura 4.18. Esta modificación de los planteamientos supone algunos problemas, porque, como se ve en la figura, la instrucción LOAD que comienza a ejecutarse en el ciclo 2 termina de ejecutarse al menos al mismo tiempo que la instrucción C que comienza en el ciclo 3 (también esto puede suponer problemas por estar los órganos de ejecución ocupados). Mientras la instrucción C no intente usar el registro cargado por LOAD no habrá ningún problema. Depende del compilador asegurarse de que la instrucción siguiente a LOAD no use el registro cargado por ésta (puede hacerlo reordenando las instrucciones). Si aún así el compilador no encuentra la solución, puede insertar, después de la instrucción LOAD, una instrucción de no operación (NOP) que resuelve el problema aunque alguna vez se pierda un ciclo. Todas estas cosas se consiguen añadiendo al compilador un módulo denominado reorganizador. La instrucción STORE no plantea estos problemas ya que la instrucción siguiente, salvo que sea LOAD sobre la misma dirección, no hará uso del dato almacenado debido a que opera sobre registros y si la instrucción siguiente a STORE es un LOAD que opere sobre el dato almacenado por STORE tampoco habrá problemas debido a que un compilador optimizado deberá eliminar ambas instrucciones que tienen efectos contrarios. También hay problemas similares con los saltos, especialmente con los condicionales, en que el procesador no sabe cuál será la instrucción siguiente. Otra característica importante de los procesadores RISC es la existencia de un banco de registros muy amplio, esto

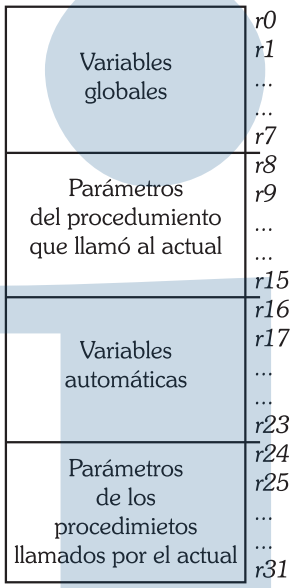

Fig. 4.19. Estructura de una ventana de registros en un procesador RISC.

se hace así para minimizar los accesos a memoria y que las instrucciones LOAD y STORE se usen lo menos posible. Evidentemente la organización de los registros no será la misma que en un ordenador convencional con registros de uso general (PDP-11 o VAX) ya que si fuera así el número de bits necesarios para la especificación del registro sería excesiva. Para ver cuál será la mejor organización para un ordenador de este tipo volvemos a las etapas de diseño descritas anteriormente: al hacer el análisis de las operaciones que más frecuentemente realizan accesos a memoria se puede ver que son las llamadas a procedimientos, por ello habrá que organizar los registros de forma que las llamadas a procedimientos optimicen los accesos a memoria. De esta forma se llegó a la organización de registros que utilizan la mayoría de los procesadores RISC: las llamadas ventanas de registros con solapamiento parcial. Esto significa que, cada vez que se llama a un procedimiento, se cambia de ventana y, además, que cada procedimiento sólo puede acceder a los registros de su ventana. Esta ventana, que suele contener alrededor de 32 registros, está dividida en cuatro zonas (ver figura 4.19):

- Los registros globales, que sirven para almacenar las variables globales y son los mismos para todas las ventanas.
- Los registros para los parámetros de llamada del procedimiento que llamó al actual.
- Los registros con las variables automáticas del procedimiento.
- Los registros para los parámetros de llamada del procedimiento actual a otros procedimientos.

Las ventanas se solapan porque los parámetros de llamada de un procedimiento son los parámetros con que le llamó el procedimiento anterior y también porque las variables globales son las mismas para todas las ventanas. El conjunto de ventanas actúa como un buffer circular con dos apuntadores: el de la ventana actual (CWP: *current window pointer*) y el de la primera

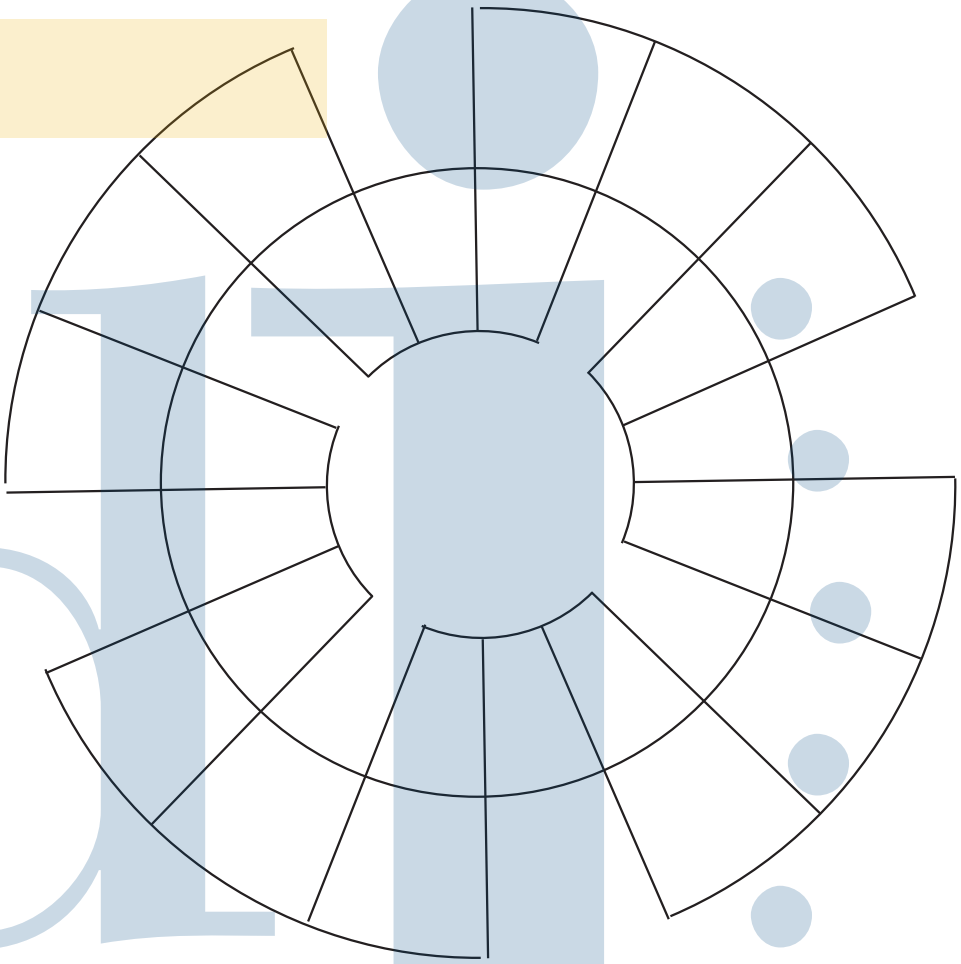

Fig. 4.20. Organización de las ventanas de registros en forma de lista circular.

ventana utilizada (OWP: *overflow window pointer*), si $CWP = OWP$ significa que todas las ventanas están ocupadas lo que provoca un desvío que salvaría en memoria la ventana correspondiente al procedimiento con nivel de anidamiento más profundo. El tamaño del banco de registros debe calcularse para que este desvío se produzca pocas veces; en general sólo debe producirse en casos de recursividad fuerte. Este mecanismo además de hacer más ágil el paso de parámetros de un procedimiento a otro también consigue que el número de bits necesarios en los campos de registro sea menor porque cada procedimiento sólo necesita acceder a los registros de su ventana. En el caso de la figura 4.19 el número de bits necesarios para direccionar un registro será $\log_2(4n)$ siendo n el número de registros de cada una de las zonas de una ventana.

Una forma de optimizar el uso de los registros es tener en cuenta el concepto de vida de las variables introducido en el apartado 4.5.1. Si el compilador realiza un estudio detallado de la vida de todas las variables de los procedimientos, puede utilizar el mismo registro para varias variables cuyo periodo de vida no se superponga.

Otra de las características de los procesadores RISC es la que le da nombre, esto no sólo debe entenderse como un conjunto de pocas y sencillas instrucciones sino también como un conjunto especializado de instrucciones para un tipo de aplicación concreto (estaciones de tra-

bajo, sistemas expertos, etc.). También se ha de mencionar que esta reducción no sólo se refiere a las instrucciones sino también, e incluso con más énfasis, a los modos de direccionamiento. Esto es debido a que los modos de direccionamiento que exijan cálculos de direcciones deben eliminarse debido a que introducirían ciclos máquina adicionales.

Bibliografía y referencias

Baase, S. 1983. *VAX-11 Assembly Language Programming*. Prentice-Hall.

Brown, F. 1991. *Processeurs RISC. L'example de l'Am29000*. Masson.

De Blasi, M. 1990. *Computer Architecture*. Addison Wesley.

Hennessy, J.L., & Patterson, D.A. 2003. *Computer Architecture. A Quantitative Approach*. 3 edn. Morgan Kaufmann Publishers.

Heudin, J.C., & Panetto, C. 1990. *Les architectures RISC*. Dunod Informatique.

Tanenbaum, A.S. 2006. *Structured Computer Organization*. 5 edn. Prentice-Hall International. Existe traducción al castellano de la edición anterior: Organización de computadores: un enfoque estructurado, 4ª edición, Prentice-Hall Hispanoamericana, 2000.

CUESTIONES Y PROBLEMAS

Nota importante: Para la resolución de muchos problemas de este capítulo es necesario consultar los apéndices.

4.1 Escribir las instrucciones de VAX necesarias para poner a 1 los bits 3, 6 y 17 del registro $R7$ y a 0 los bits 2, 5 y 8.

4.2 Repetir el problema anterior en ensamblador de arquitectura SPARC con el registro $l0$.

4.3 Escribir en ensamblador de VAX las instrucciones necesarias para complementar todos los bits del segundo byte comenzando por la derecha del registro $R9$ dejando inalterados el resto de los bits de ese registro.

4.4 a) En el registro C de un Z-80 hay dos cifras codificadas en BCD empaquetadas. Escribir un programa en lenguaje ensamblador de Z-80 que pase esas cifras a los registros D y E.

b) Codificar el programa anterior en código máquina.

4.5 Repetir el problema anterior para un MC68000 suponiendo que en el registro $D0$ hay 8 cifras en BCD y que las queremos depositar separadas en bytes seguidos a partir de la dirección A cuyo valor es 00456F00H.

4.6 En el registro $R5$ de un PDP-11 hay dos caracteres empaquetados. Escribir las instrucciones necesarias para pasar el carácter situado en el byte de orden más alto de este registro al byte más bajo del registro $R2$ sin alterar el resto del contenido de este registro ni el contenido original de $R5$.

4.7 Supongamos que a partir de la dirección relocalizable X de la memoria de un PDP-11 hay una cadena que representa un número decimal positivo de dos cifras codificado en ASCII. Escribir un programa en lenguaje ensamblador para construir, a partir de la dirección Y, otra cadena que represente el mismo número en hexadecimal también en ASCII.

4.8 Demostrar la propiedad dada por la ecuación 4.1.

4.9 Si los registros $l0$ y $l1$ de una máquina con arquitectura SPARC representan respectivamente sendos conjuntos A y B, indicar las instrucciones necesarias para:

a) Comprobar si un cierto elemento situado en la posición genérica i pertenece a B.

b) Calcular $A \cap B$.

c) Comprobar si $A \subset B$.

4.10 Supongamos que X e Y son dobles palabras que representan a los elementos de sendos conjuntos A y B. Escribir un fragmento de programa en lenguaje ensamblador de VAX que bifurque a la etiqueta $Incluido$ si y, sólo si, $A \subset B$, en caso contrario la ejecución debe continuar secuencialmente.

4.11 Supóngase que X es una palabra que representa un conjunto A dentro del conjunto universal de los números enteros comprendidos entre 0 y 15. Escribir en lenguaje ensamblador del PDP-11 las instrucciones necesarias para bifurcar a la etiqueta $Pertenece$ si el número contenido en la dirección Y es un elemento del conjunto A.

4.12 ¿Cuál de los tres métodos descritos en el texto para analizar la inclusión de un conjunto en otro es menos eficiente en un VAX?¿Y en una máquina con arquitectura SPARC?

4.13 Supongamos que en una máquina con arquitectura SPARC, sendas palabras de memoria a y b representan a dos conjuntos A y B. Escribir las instrucciones necesarias en lenguaje ensamblador de dicha máquina para:

a) Calcular la palabra que represente a la unión de A y B.

b) Calcular la palabra que represente a la intersección de A y B.

c) Calcular una variable entera que valga 1 si, y sólo si, $A \subset B$.

d) Calcular una variable entera que valga 1 si, y sólo si, A y B son disjuntos.

4.14 Inventar un método para intercambiar dos variables sin utilizar una tercera variable ni ningún otro registro.

Sugerencia: Recordar las propiedades de la operación OR exclusivo.

4.15 Describir la forma más rápida de borrar un registro si no se dispone de una instrucción específica para ello.

4.16 Todas las instrucciones siguientes tienen el mismo efecto: borrar el registro $R6$ de un VAX. Sin embargo, requieren diferente número de bytes al codificarlas en código máquina. ¿Cuántos bytes requiere cada una? ¿Cuál es más eficiente?

a) `SUBL2 R6, R6` b) `CLRL R6` c) `MOVL #0, R6`
d) `XORL2 R6, R6` e) `MULL2 #0, R6`

4.17 Todas las instrucciones siguientes tienen el mismo efecto: borrar el registro $l1$ de un procesador con arquitectura SPARC ¿Cuál es más eficiente?

a) `mov 0,%l1` b) `xor %l1,%l1,%l1` c) `or %g0,%g0,%l1`
d) `clr 0,%l1` e) `sub %l1,%l1,%l1` f) `mulx 0,%l1,%l1`

4.18 Escribir, en ensamblador del VAX, las instrucciones para bifurcar a la etiqueta X si las dobles palabras contenidas en las direcciones A, B y C son todas mayores que 1024. En caso contrario debe ejecutarse la siguiente instrucción.

4.19 Repetir el problema anterior para que la ejecución del programa continúe en la etiqueta X sólo si alguna de las dobles palabras mencionadas es mayor que 1024.

4.20 Dibujar la gráfica de dependencia del contador del programa con respecto al tiempo para dos llamadas anidadas a procedimientos. Se debe suponer que el código dentro de ambos procedimientos es lineal.

4.21 Dibujar la gráfica de dependencia del contador del programa con respecto al tiempo para dos bucles anidados. Supóngase que el código dentro de los bucles es lineal, que el bucle interno se ejecuta tres veces y el externo dos.

4.22 Dibujar la gráfica de variación del contador de programa respecto al tiempo para un programa que llama a un procedimiento recursivo que se ejecuta tres veces. La llamada recursiva dentro del procedimiento se produce aproximadamente en la mitad del procedimiento y cuando se produce la salida se salta la instrucción de llamada.

4.23 Dibujar la gráfica de dependencia del contador de programa con respecto al tiempo para dos procedimientos que se llaman recursivamente de forma indirecta y se ejecutan dos veces cada uno. La salida se produce en uno de los procedimientos mediante una bifurcación condicional efectiva que provoca que se salte la instrucción de llamada al otro.

4.24 Escribir las instrucciones de VAX necesarias para el intercambio de control entre dos corrutinas.

4.25 Escribir las instrucciones de VAX necesarias para llamar a un procedimiento que tiene dos parámetros mediante la instrucción CALLG pero pasando los parámetros a través de la pila.

4.26 a) Supóngase que, a partir de una dirección, cuya etiqueta es CADENA, existe una cadena de caracteres cuya longitud está localizada en la dirección L. Escribir un programa en ensamblador del VAX que ponga en la dirección POSICION la dirección del primer

blanco que exista en la cadena. Si la cadena no contiene blancos la dirección POSICION quedará a 0.

b) Traducir el programa escrito para resolver el apartado anterior a lenguaje máquina suponiendo que el programa comienza en la dirección 0000C000H y que las etiquetas representan los siguientes valores: CADENA: 0000A015H, POSICION: 0000BA10H, L: 0000BA20H.

4.27 Repetir el problema anterior para el MC68000.

4.28 a) Escribir un procedimiento en lenguaje ensamblador del VAX que tome como parámetros dos números enteros (una doble palabra, N, y un byte, M) y una dirección. El procedimiento escribirá, a partir de la dirección suministrada, M números consecutivos a partir de N. El procedimiento tendrá también un parámetro de error (de tipo byte) que devolverá un 1 si M es negativo o cero).

b) Utilizar el procedimiento escrito en el apartado anterior para confeccionar un programa que pida por teclado dos números enteros, A y B, y escriba por pantalla B números consecutivos a partir de A.

4.29 a) Escribir los directivos del lenguaje ensamblador del SPARC necesarios para reservar espacio para la siguiente variable descrita en lenguaje Pascal Se puede suponer que el tamaño de *integer* es de 2 bytes.

```
var v: array [0..2] of
  record
    a: integer;
    b: array [0..2] of integer
  end
```

b) Escribir las instrucciones de lenguaje ensamblador más eficientes para depositar en `v[2].a` la suma

$$\sum_{i=0}^{2} v[2].b[i]$$

c) Repetir el apartado anterior para un PDP-11.

4.30 Describir brevemente los pasos necesarios para acceder a un elemento genérico de una matriz de dos dimensiones.