3

PROCESADORES VECTORIALES

3.1. Introducción y definiciones

En este capítulo estudiaremos los conceptos básicos del procesamiento vectorial. Estableceremos las diferencias entre los procesadores escalares y vectoriales. Al final del capítulo se analizarán algunos ejemplos de procesadores vectoriales reales, con ello se verá como la teoría se hace realidad en estas máquinas.

Para comenzar veremos algunas definiciones previas:

Se llama **vector** a *una secuencia de datos escalares del mismo tipo almacenados en memoria*, normalmente en posiciones contiguas, aunque no siempre. Para ilustrar este hecho, supóngase una matriz bidimensional almacenada en memoria por filas. En esta matriz, podríamos considerar vectores a las filas, columnas o diagonales; en este caso, sólo las filas estarán en posiciones contiguas de memoria.

Un **procesador vectorial** es un *conjunto de recursos para efectuar operaciones sobre vectores*. Estas operaciones consistirán en funciones aritméticas y lógicas aplicadas sobre las componentes de los vectores. La diferencia entre un procesador vectorial y uno escalar estriba en que el procesador vectorial puede decodificar instrucciones cuyos operandos son vectores completos. La conversión de un programa correspondiente a un procesador escalar a otro vectorial se llama **vectorización**.

3.2. Procesamiento vectorial

Un **operando vectorial** contiene una secuencia de n elementos, llamados componentes, donde n es la **longitud del vector**. Cada componente del vector es un escalar de cualquier tipo (entero, punto flotante, etc.). Los operadores vectoriales pueden tener una de estas cinco formas

Tabla 3.1. Ejemplos de operadores vectoriales

Nombre	Nemónico	Operación	Tipo
Máxima componente	VMAX	$s = \max(a_i)_{i=1,..,n}$	Reducción unitaria
Módulo	VMOD	$s = \sqrt{\sum_{i=1}^{n} a_i^2}$	Reducción unitaria
Raíz cuadrada vectorial	VSQR	$b_i = \sqrt{a_i}$	Operación vectorial unitaria
Suma de componentes	VSUM	$s = \sum_{i=1}^{n} a_i$	Reducción unitaria
Suma de vectores	VADD	$c_i = a_i + b_i$	Operación vectorial binaria
Media de las componentes	VMEAN	$s = \frac{\sum_{i=1}^{n} a_i}{n}$	Reducción unitaria
Multiplicación de escalar por vector	SMUL	$b_i = s * a_i$	Escalado
Multiplicación de vectores	VMUL	$c_i = a_i * b_i$	Operación vectorial binaria
Producto escalar	VDOT	$s = \sum_{i=1}^{n} a_i * b_i$	Reducción binaria

(podría haber algunas más, pero no tienen interés práctico):

$$
\begin{array}{ll}
f_1 : V \longrightarrow V & \text{(operación vectorial unitaria)} \\
f_2 : V \times V \longrightarrow V & \text{(operación vectorial binaria)} \\
f_3 : V \longrightarrow K & \text{(reducción unitaria)} \\
f_4 : V \times V \longrightarrow K & \text{(reducción binaria)} \\
f_5 : K \times V \longrightarrow V & \text{(operación de escalado)}
\end{array}
$$

donde V y K representan, respectivamente, a un espacio vectorial y un cuerpo. En la tabla 3.1 pueden verse una serie de ejemplos de operadores vectoriales. En esta tabla, **a**, **b**, **c** $\in V$, a_i, b_i y c_i son, respectivamente, las componentes i-ésimas de esos vectores, $s \in K$ y $n = \dim V$. Casos especiales de operaciones vectoriales binarias son las operaciones de **empaquetamiento** y **desempaquetamiento**, en que uno de los vectores actúa como máscara y el vector resultado tiene un tamaño diferente al de los operandos. Estas operaciones tienen bastante utilidad en el tratamiento de **matrices dispersas**, que son matrices con la mayoría de sus elementos nulos.

Las máquinas vectoriales proporcionan operaciones que trabajan sobre vectores. Una instrucción vectorial es equivalente a la ejecución de un bucle completo de instrucciones ordinarias, donde cada iteración trabaja sobre cada una de las componentes del vector. Las operaciones vectoriales tienen algunas ventajas sobre las escalares:

- En las operaciones vectoriales, cada resultado es independiente de los anteriores. Esto

permite efectuar los cálculos en un procesador segmentado sin que existan conflictos por dependencias de datos.

- Una simple instrucción vectorial sustituye a muchas escalares. Por ello, el cuello de botella producido por la lectura de esa instrucción es pequeño, comparado con el que produciría el conjunto de instrucciones escalares a las que equivale.
- Las instrucciones vectoriales que precisan acceder a memoria, lo hacen con un patrón de acceso fijo (normalmente serán adyacentes). Esto facilitará su lectura paralela mediante una memoria entrelazada. En cualquier caso, si no se dispusiera de memorias entrelazadas, las posiciones de memoria adyacentes se cargarán en caché, con el consiguiente ahorro de tiempo.
- Si se utiliza una instrucción vectorial, evitaremos el riesgo de control del salto del bucle, que se produciría si procesáramos las instrucciones escalares equivalentes en un procesador segmentado.

Por todas estas razones, las operaciones vectoriales pueden ejecutarse de forma mucho más rápida que la secuencia de instrucciones equivalentes sobre el mismo conjunto de datos. Esto hace que se deba tender al diseño y fabricación de máquinas vectoriales en aplicaciones donde este tipo de operaciones se usen con suficiente frecuencia.

3.3. Segmentación y procesadores vectoriales

Visto lo anterior, parece claro que los computadores vectoriales deben basar su unidad de ejecución en un procesador segmentado que tomará uno a uno todos las componentes del vector y los irá procesando sin dependencias de datos ni control durante la ejecución de toda la instrucción vectorial. La mayoría de los computadores vectoriales actuales trabajan así. Otra posibilidad más cara sería la **división funcional** (recuérdese el apartado 1.4.1) manteniendo una unidad de ejecución para cada componente del vector. El alto coste de esta solución no suele justificarla. Por otra parte, este tipo de solución precisa que los vectores tengan una longitud limitada; con un procesador segmentado, las componentes pueden ir entrando y saliendo por el cauce sin límite de componentes, aunque, evidentemente, un vector más largo tardará más tiempo en procesarse.

3.4. Arquitectura de los procesadores vectoriales

Una primera versión, muy simplificada, de procesador vectorial es el mostrado en la figura 3.1. Este tipo de máquina sería un **procesador vectorial memoria-memoria** que es capaz de extraer dos vectores de memoria y operar sobre ellos. El inconveniente de este tipo de máquina sería el cuello de botella que supondrían los accesos a memoria. Por ello, sobre esta primera arquitectura pueden hacerse algunas mejoras:

1. Aumentar el ancho de banda de la memoria: esto se consigue **entrelazando la memoria**, de forma que la ésta tenga varios módulos y se pueda acceder simultáneamente a varias posiciones consecutivas que se hallen en módulos diferentes.

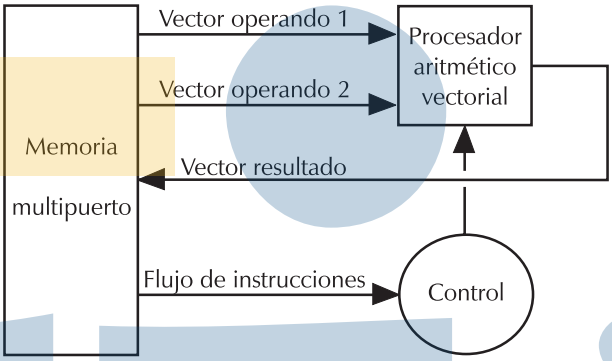

Fig. 3.1. Procesador vectorial memoria-memoria.

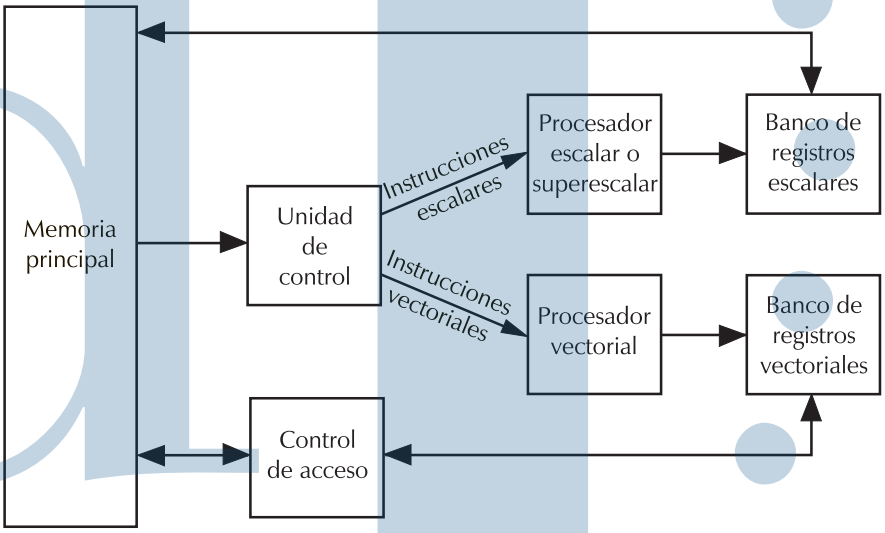

Fig. 3.2. Procesador vectorial registro-registro.

2. Añadir una memoria intermedia de mayor velocidad entre la memoria y el procesador. Una forma de hacer esto, además de incorporar una memoria caché, es añadir un banco de registros vectoriales.

Por todo esto, es habitual que los procesadores vectoriales dispongan de un banco (o varios) de registros vectoriales que hagan de memoria intermedia; se habla entonces de **procesadores vectoriales registro-registro**. Por otra parte, la mayoría de los procesadores vectoriales actúan como coprocesadores de un procesador escalar convencional que trata las instrucciones no vectoriales. Ello hace que la arquitectura de un computador vectorial actual sea la que se muestra en la figura 3.2. Como puede verse, los computadores vectoriales pueden considerarse como computadores del tipo SIMD de Flynn, sin embargo, hay que tener en cuenta que, si bien los computadores vectoriales ejecutan la misma instrucción sobre datos diferentes, estos datos forman parte del mismo flujo. Por ello, en algunos casos, a estas máquinas se les llama SIMD-vectoriales (Germain-Renaud & Sansonnet, 1991).

Bajo estas arquitecturas, estudiaremos ahora algunos problemas que se plantean en los computadores vectoriales:

- En primer lugar plantearemos el problema de la longitud de los vectores; éste sólo se plantea en los procesadores vectoriales de tipo registro-registro, ya que la capacidad almacenamiento de los registros vectoriales es limitada. La longitud "natural" de los vectores que se tratan en un procesador vectorial será la de sus registros vectoriales; sin embargo, la longitud de los vectores declarados en los programas rara vez coincidirá con ella, es más, en muchos casos la longitud de los vectores sólo se conocerá en el momento de la ejecución (por ejemplo, cuando uno de los límites de un bucle sea una variable). Una solución a este problema es mantener un **registro de longitud vectorial** (**VLR,** ***vector length register***) que contenga la longitud del vector que se procesa. En principio, la longitud del vector debe ser inferior a la longitud de los registros vectoriales (MVL**,** ***maximum vector length***). Si la longitud del vector fuera mayor recurriremos a la técnica denominada **seccionamiento** (***strip-mining***) que consiste en que el compilador divida la operación vectorial en lotes cuyo tamaño máximo sea MVL. Veamos ahora un ejemplo de seccionamiento:

  Ejemplo 3.1: *Sea el siguiente programa FORTRAN (lenguaje muy utilizado en máquinas vectoriales, aunque en diferentes versiones):*

  ```
  do i=1,n
       a(i)=b(i)+c(i)
  enddo
  ```

  Donde `n` *es una variable que podría contener un valor superior a* MVL*. Si se aplica la técnica de seccionamiento para procesar el bucle anterior, el compilador sustituirá ese bucle por*

  ```
  linf=1
  vl=n mod MVL
  do j=0,n/MVL
       do i=linf,linf+vl-1
            a(i)=b(i)+c(i)
       enddo
       linf=linf+vl
       vl=MVL
  enddo
  ```

  En este programa, `MVL` *es una constante que contiene la longitud de los registros vectoriales, y el bucle más interno será sustituido por una sola instrucción vectorial, en que* `vl` *es una variable que contendrá su longitud.*

  ○

  Hay que tener en cuenta que el seccionamiento realiza algunas operaciones adicionales. Este costo adicional normalmente compensará debido al mayor rendimiento del procesamiento vectorial.

- Otro problema que puede plantearse es el proceso de vectores cuyas componentes no son adyacentes en memoria. Este es el caso de la multiplicación de matrices, que puede transformarse en una multiplicación de vectores. Para comprender mejor esto, veamos el siguiente programa FORTRAN que multiplica dos matrices:

```
do i=1,n
     do j=1,n
          p(i,j)=0.0
          do k=1,n
               p(i,j)=p(i,j)+a(i,k)*b(k,j)
          enddo
     enddo
enddo
```

En este programa, `n` es la dimensión de las matrices que se pretenden multiplicar. El programa también puede interpretarse como la multiplicación de la fila `i` de la matriz `a` por la columna `j` de `b` (consideradas ambas como vectores). Esta multiplicación de vectores puede efectuarse con una operación vectorial (aunque pudiera estar seccionada) con el inconveniente de que uno de los dos vectores no tendrá sus componentes adyacentes en memoria. Esto será así ya que, si las matrices se almacenan por filas, las componentes de la columna `j` de `b` no serán adyacentes; algo simétrico ocurrirá, si las matrices se almacenan por columnas, con las componentes de la fila `i` de `a`. Esto no tiene, en la práctica, ninguna implicación sobre la propia operación de multiplicación de vectores, ya que las filas y columnas podrán residir en los registros vectoriales de la máquina y la multiplicación se efectuará como si de dos vectores cualesquiera se tratase. El problema radica, esencialmente, en la operación de carga del vector no adyacente en el registro vectorial. En este caso se dice que las componentes del vector tienen **separación (*stride*)** no unitaria. La solución a este problema puede ser incorporar una instrucción que realice una carga vectorial de elementos no adyacentes aunque con una separación fija. Esta separación puede ser el contenido de alguno de los registros escalares de la máquina. Hay máquinas vectoriales en que todas las instrucciones de carga vectorial llevan asociada una separación (aunque sea unitaria), se evita así duplicar instrucciones. Esto hace que para leer un vector de memoria sean necesarios tres datos: la **dirección inicial** del vector, la **separación (*stride*)** y la **longitud**.

Queda todavía analizar cómo se comportará la memoria ante una carga con separación no unitaria. Una memoria entrelazada no producirá muchos beneficios en estos casos, a excepción de que la separación sea múltiplo del número de módulos de la memoria. Sin embargo, una memoria caché en que se haya dimensionado correctamente el grado de asociatividad puede funcionar muy bien. Aun así se podría producir alguna pérdida de rendimiento debida a los accesos no adyacentes. Una posibilidad para evitar estas pérdidas de rendimiento es permitir la ejecución simultánea de instrucciones de carga e instrucciones aritméticas (sobre registros diferentes); esto no es muy difícil porque ambos tipos de instrucciones involucran a órganos diferentes del procesador.

- Por último, puede plantearse el problema de las dependencias de datos entre los diferentes vectores que intervienen en el cálculo. Este problema se resuelve, en parte, con un método

denominado **encadenamiento** que consiste imponer sólo dos condiciones para comenzar una operación vectorial:

- Que la primera componente del vector esté disponible. Esto puede mejorarse si se emplea **anticipación**, es decir, si se toma el resultado de una operación, para efectuar la siguiente, antes de almacenarse en el registro destino.
- Que la unidad funcional necesaria para efectuar la operación se encuentre disponible.

Otra forma de reducir las detenciones por dependencias de datos es incluir en el juego de instrucciones del procesador vectorial las llamadas **funciones vectoriales compuestas (*compound vector functions*, CVF)**. Este tipo de funciones integran dos o más operaciones en una sola instrucción. Para ver esto, tomemos el siguiente ejemplo:

Ejemplo 3.2: *Sea el siguiente código FORTRAN:*

```
do i=1,n
   y(i)=a*x(i)+y(i)
enddo
```

Este tipo de código se denomina SAXPY *o* DAXPY *(single precision o double precision a*x *plus* y*, simple o doble precisión $a\mathbf{x}+\mathbf{y}$), y se utiliza para evaluar el rendimiento de muchos computadores. Este código forma parte del test de Linpack y también de la resolución de sistemas de ecuaciones lineales.*

En un procesador vectorial sin CVF*, este cálculo se haría en dos partes: el producto del escalar* `a` *por el vector* `x` *y la suma del vector resultado con el vector* `y`*. Esto, expresado en un lenguaje máquina vectorial, tomaría la forma siguiente:*

```
LOAD N, VLR    ; Carga de la longitud en el registro de longitud
LOAD X, V1     ; Carga del vector X
LOAD Y, V2     ; Carga del vector Y
MUL A, V1      ; Multiplicación de escalar por vector (escalado)
ADD V1, V2     ; Suma de vectores
STORE V2, Y    ; Almacenamiento del resultado
```

Como puede apreciarse, existirá una dependencia de datos entre las operaciones de multiplicación y adición debido al vector `V1`*. Incluso empleando la técnica de encadenamiento explicada anteriormente, perderíamos algunos ciclos a causa de esta dependencia.*

Si el procesador dispusiera de la función vectorial compuesta SMULADD *(multiplicar por escalar y sumar), el programa anterior se transformaría en:*

```
LOAD N, VLR       ; Carga del registro de longitud
LOAD X, V1        ; Carga del vector X
LOAD Y, V2        ; Carga del vector Y
SMULADD A, V1,V2  ; Multiplicación de escalar por vector y suma
STORE V2, Y       ; Almacenamiento del resultado
```

De esta forma, la dependencia se puede resolver a nivel interno en la unidad aritmética: cuando se tenga el resultado de la multiplicación, éste puede entrar en la unidad sumadora sin necesidad de decodificar o detener la instrucción de suma.

○

3.5. Rendimiento de los procesadores vectoriales

Basándonos en lo anterior, presentaremos un modelo para evaluar el rendimiento de los procesadores vectoriales. Tomaremos para hacer esta evaluación, un cálculo vectorial sobre vectores de n componentes en un bucle que deba seccionarse. En el cálculo de este rendimiento intervienen las siguientes variables (Hennessy & Patterson, 1990):

1. El tiempo necesario para procesar cada componente en una pasada del bucle. Denominaremos a ese tiempo t_p.

2. El tiempo de inicialización de cada bucle provocado por el seccionamiento. Este tiempo se puede desglosar en dos partes: el tiempo del código escalar necesario para el seccionamiento (t_s) y el tiempo de arranque de la instrucción vectorial (t_a).

3. El tiempo ocupado por el arranque del programa, carga inicial de los vectores en los registros, etc. (t_b)

Con todo esto, el tiempo necesario para el cálculo total será:

$$T_{total} = t_b + \left\lceil \frac{n}{MVL} \right\rceil (t_s + t_a) + nt_p$$

El tiempo ganado respecto a un procesador segmentado escalar radica, más que en la mejora del tiempo de cálculo propiamente dicho, en el tiempo necesario para leer y decodificar las instrucciones. También puede ganarse tiempo si se dispone de varios cauces aritméticos para que sea posible ejecutar varias instrucciones vectoriales de forma simultánea.

Hay algunos parámetros que pueden ayudar a evaluar la calidad de un procesador vectorial referidos a la longitud del vector que se está tratando, éstos son:

R_n: es la velocidad en MFLOPS sobre un vector de longitud n. Por extensión y de forma ideal, R_∞ es la velocidad del procesador para un vector de longitud infinita. La diferencia entre estas medidas nos dará las penalizaciones en el rendimiento debidas a la longitud finita de los vectores.

$N_{1/2}$: es la longitud necesaria para alcanzar una velocidad de $R_\infty/2$.

N_v: es la longitud necesaria para hacer el modo vectorial más rápido que el escalar. Esta medida evalúa las pérdidas de rendimiento debidas a las operaciones adicionales necesarias en el proceso vectorial respecto al escalar.

Veamos ahora una forma de evaluar la eficiencia de los procesadores vectoriales: sea r la relación entre las velocidades de proceso vectorial y escalar y sea f_v la fracción de código vectorizable. Tomando como unidad el tiempo que tardaría en ejecutarse el proceso si la máquina fuera totalmente escalar, tendremos que la ganancia de velocidad de un procesador vectorial será:

$$S = \frac{1}{1 - f_v + \frac{f_v}{r}} = \frac{r}{r(1 - f_v) + f_v}$$

La eficiencia del sistema vendría dada entonces por:

$$E = \frac{\frac{r}{r(1-f_v)+f_v}}{r} = \frac{1}{r(1 - f_v) + f_v}$$

Estas expresiones se refieren al caso ideal en el que el procesador vectorial no tenga pérdidas por causas diferentes al código no vectorizable. Por otra parte, las expresiones anteriores no son más que otra forma de expresar la ley de Amdahl dada por la expresiones 1.9 y 1.11.

3.6. Características de los lenguajes para proceso vectorial

El uso de lenguajes secuenciales en computadores vectoriales puede hacer perder el paralelismo de un algoritmo vectorizable. Es, por tanto, necesaria la existencia de lenguajes de alto nivel, adecuados a los computadores vectoriales. Estos lenguajes deberían tener una serie de propiedades para expresar el paralelismo de los algoritmos y así poderlo explotar con más eficiencia:

- **Flexibilidad** para declarar diferentes clases de objetos con distintas estructuras y formas de almacenamiento. El lenguaje debe poder expresar las diferentes formas de almacenar las componentes de un mismo objeto: por filas, columnas, diagonales, etc.

- **Efectividad** para la manipulación de matrices y vectores **dispersos**, es decir, matrices o vectores con la mayoría de sus elementos nulos. Las matrices dispersas son muy habituales en problemas reales, por ello, el lenguaje de programación debe suministrar los medios para almacenarlas de forma eficiente sin ocupar excesiva memoria.

- Disponer de **operaciones vectoriales** nativas que trabajen directamente con las estructuras de datos anteriormente declaradas sin necesidad de bucles. Será el compilador del lenguaje el que transforme estas operaciones de alto nivel en las instrucciones vectoriales de la máquina. Muchas veces, el número de componentes del vector declarado en el lenguaje de alto nivel, será superior a la capacidad de los registros vectoriales. Por ello, el compilador deberá aplicar la técnica de seccionamiento para descomponer algunas instrucciones vectoriales de alto nivel, en otras instrucciones vectoriales más sencillas que puedan ejecutarse con la capacidad de almacenamiento de los registros vectoriales de la máquina. Esto no significa, sin embargo, que estos lenguajes no admitan programación mediante bucles. Esto es debido a que puede haber muchas operaciones combinadas que sólo pueden detallarse mediante bucles específicos. Evidentemente, el lenguaje vectorial no puede tener como operaciones nativas todas las posibles combinaciones de operaciones vectoriales, aunque sí las más frecuentes.

Veremos algunos ejemplos típicos de las propiedades anteriores:

Ejemplo 3.3: *El primer ejemplo lo tomaremos del problema* SAXPY *visto en el ejemplo 3.2. Veremos la potencia de un lenguaje vectorial sobre este ejemplo. En FORTRAN 90 (un lenguaje vectorial) esto se expresaría de la siguiente forma (se suponen los datos ya declarados:* `a` *es un escalar y* `x` *e* `y` *son dos vectores de* `n` *componentes):*

```
y(1:n)=a*x(1:n)+y(1:n)
```

Si bien en FORTRAN 90 se puede expresar todo el bucle con una sola instrucción, esto no significa que ese código se traduzca por una única instrucción vectorial, como se vio en el ejemplo 3.2.

○

Este ejemplo sólo es un caso especial de la potente sintaxis del FORTRAN 90 en cuanto al tratamiento de vectores y matrices. En general, en el citado lenguaje, un índice de una matriz o vector tiene la forma general siguiente:

$$i_1 : i_2 : i_3$$

donde todas las i_j son expresiones enteras cuyo significado es el siguiente:

i_1: es el valor inicial del índice.

i_2: es el valor final del índice. En lugar de este parámetro puede ponerse el símbolo *; ello indicará que el índice varía desde i_1 hasta el valor máximo del índice indicado en la declaración de la matriz; si, además, se omite el parámetro i_1 el índice variará en todo su rango.

i_3: es el incremento o separación de las componentes. Este parámetro puede omitirse, en cuyo caso adquiere el valor 1.

En el caso de que sólo se ponga un valor concreto, se interpretará que nos referimos a una de las componentes de la matriz o vector.

Las asignaciones y operaciones aritméticas entre matrices **sólo pueden hacerse si todas las matrices involucradas tienen el mismo número de dimensiones e implican en cada dimensión al mismo número de elementos**.

Ejemplo 3.4: *En este segundo ejemplo veremos como algunas operaciones no son fácilmente vectorizables. Este es el caso de muchas operaciones de* ***reducción*** *(recordar el párrafo 3.2). Tomemos como ejemplo de este tipo de operaciones el producto escalar de dos vectores* `a` *y* `b`*. El código FORTRAN para efectuar esta operación será:*

```
program pescalar
parameter (n=64)
real a(n),b(n)
real producto
```

```
.......
producto=0.0
do i=1,n
     producto=producto+a(i)*b(i)
enddo
end
```

Un compilador vectorial no podrá transformar completamente este código en vectorial debido a la dependencia de datos creada por la variable `producto`*. Sin embargo, sí podrá hacer alguna mejora que consistiría en separar el problema en una parte escalar y otra vectorial. Esto podrá conseguirse transformando la variable* `producto` *en un vector. A este artificio se le denomina* ***expansión escalar*** *y su resultado es el siguiente código:*

```
do i=1,n
     producto(i)=a(i)*b(i)
enddo
do i=2,n
     producto(1)=producto(1)+producto(i)
enddo
```

Como puede verse, el primer bucle es vectorizable, no siéndolo el segundo debido a la dependencia de datos. Esta última versión en FORTRAN 90 se escribiría de la forma siguiente:

```
producto(*)=a(*)*b(*)
producto(1)=sum(producto(*))
```

Suponiendo que la función `sum` *sea una función escalar, con argumento vectorial, que suma las componentes del vector que recibe como argumento.*

○

3.7. Compiladores para procesadores vectoriales

Un compilador con vectorización analiza si las instrucciones situadas dentro de los bucles pueden ser ejecutadas en paralelo y genera código objeto con instrucciones vectoriales. En tanto en cuanto el compilador sea capaz de vectorizar más el código, el rendimiento mejorará en la misma medida. En esto estriba la importancia de los compiladores vectoriales. Existen importantes dificultades en la vectorización del código. Estas dificultades radican en las instrucciones de control, especialmente en las condicionales, en las dependencias de datos, en las indexaciones indirectas (es decir componentes de vectores o matrices que son índice de otros vectores o matrices) que sólo se resuelven en ejecución. La indexación indirecta es uno de los tratamientos más comunes de las matrices dispersas. Como debe conocerse, las etapas de un compilador son las siguientes:

1. Análisis léxico.
2. Análisis sintáctico.
3. Análisis semántico, como consecuencia de este análisis se generará un código intermedio.
4. Optimización del código.
5. Generación de código objeto.

En un compilador con vectorización, las dos primeras fases no cambian. El código generado a partir del análisis semántico debe ser código vectorial y debe someterse en la siguiente etapa a una serie de optimizaciones diferentes a las de los compiladores convencionales. Esta optimización vectorial afectará, entre otros, a los aspectos siguientes:

Reducción de constantes en compilación: Esto significa que el compilador evalúe todas las expresiones que pueda en compilación. Este tipo de reducción, que habitualmente se efectúa para el código escalar, adquiere algunos nuevos aspectos en el código vectorial. Por ejemplo, sea el siguiente código FORTRAN para inicializar un vector:

```
do i=1,n
     a(i)=2*i
enddo
```

Un compilador optimizado debe ser capaz de evitar este código en la ejecución e inicializar el vector completamente en compilación, con el consiguiente ahorro de tiempo. Esto generaría un vector inicializado, en compilación, al valor (2,4,6...).

Traslado de expresiones invariantes: frecuentemente los bucles más internos incluyen operaciones que, en determinadas condiciones, pueden ser trasladadas a una zona más externa. Esta acción se denomina **traslado de código**. Supongamos el siguiente fragmento de código FORTRAN:

```
do i=1,m
     do j=1,n
          a(i,j)=a(i,j)+b(j)*c(j)
     enddo
enddo
```

Este código podría vectorizarse así:

```
do i=1,m
     a(i,1:n)=a(i,1:n)+b(1:n)*c(1:n)
enddo
```

Sin embargo, un análisis más detallado podría establecer que la multiplicación de los vectores `b` y `c` puede sacarse del bucle, almacenándolo en una variable temporal como sigue:

```
tmp(1:n)=b(1:n)*c(1:n)
do i=1,m
     a(i,1:n)=a(i,1:n)+tmp(1:n)
enddo
```

Este tipo de optimización sería diferente que en un procesador escalar, ya que, si bien podría sacarse la operación del bucle de la misma forma, esta operación seguiría siendo un bucle, mientras que en el procesador vectorial es una sola instrucción.

3.8. Ejemplos reales de computadores vectoriales

Estudiaremos aquí algunos ejemplos clásicos de computadores vectoriales. Algunos de ellos están ya obsoletos, pero son ejemplos clásicos que deben mencionarse por la importancia que tuvieron en su momento.

3.8.1. El Cray-1 de Cray Reseach

El Cray-1, operativo desde 1976, puede considerarse como el primer computador vectorial. Como muchos supercomputadores (el Cray-1 en su tiempo lo fue), necesita un procesador auxiliar (*front end host*) que efectúe labores de administración, E/S, etc. De esta forma, el Cray-1 se dedica sólo a computación pura. El Cray-1 posee tres secciones (Russell, 1978): la sección de computación, la sección de memoria y la sección de E/S.

La sección de memoria está organizada en 8 o 16 bancos con 72 módulos por banco (1 módulo por bit) de los que 8 son para detección y corrección de errores: el sistema puede detectar hasta 2 errores y corregir 1, siendo la palabra eficaz de memoria de 64 bits. La memoria es entrelazada con 16 vías.

En la figura 3.3 puede verse la estructura de la sección de computación del Cray-1. Como se aprecia en la citada figura, esta máquina dispone de tres tipos de registros: registros de dirección (A), registros escalares (S) y registros vectoriales (V), habiendo 8 registros de cada clase. También existen unos registros temporales, tanto escalares (T) como de direcciones (B), que actúan a modo de memorias caché para datos y direcciones. Existen 64 registros temporales de cada una de estas clases. Tiene también 4 bancos de registros temporales de instrucciones (caché de código). Los registros de direcciones (tanto temporales como primarios) tienen 24 bits, los escalares 64 y los vectoriales tienen 64 registros componentes de 64 bits cada uno. La máquina dispone de un registro de longitud (VL) que indica la longitud de los operandos vectoriales. Si la longitud de los vectores del problema es mayor, debe recurrirse a la técnica de seccionamiento. Los contenidos de los registros vectoriales se transfieren desde (o a) memoria indicando la dirección inicial, el incremento y la longitud del vector.

El sistema dispone de 12 unidades funcionales segmentadas divididas en cuatro grupos: vectoriales (enteras y de punto flotante), escalares y de direcciones. Cada unidad funcional puede trabajar con total independencia de las demás, por lo que todas ellas pueden funcionar concurrentemente siempre que no haya conflictos en cuanto a los registros que utilicen. Las unidades funcionales pueden acceder directamente a los registros primarios, pero no a los temporales.

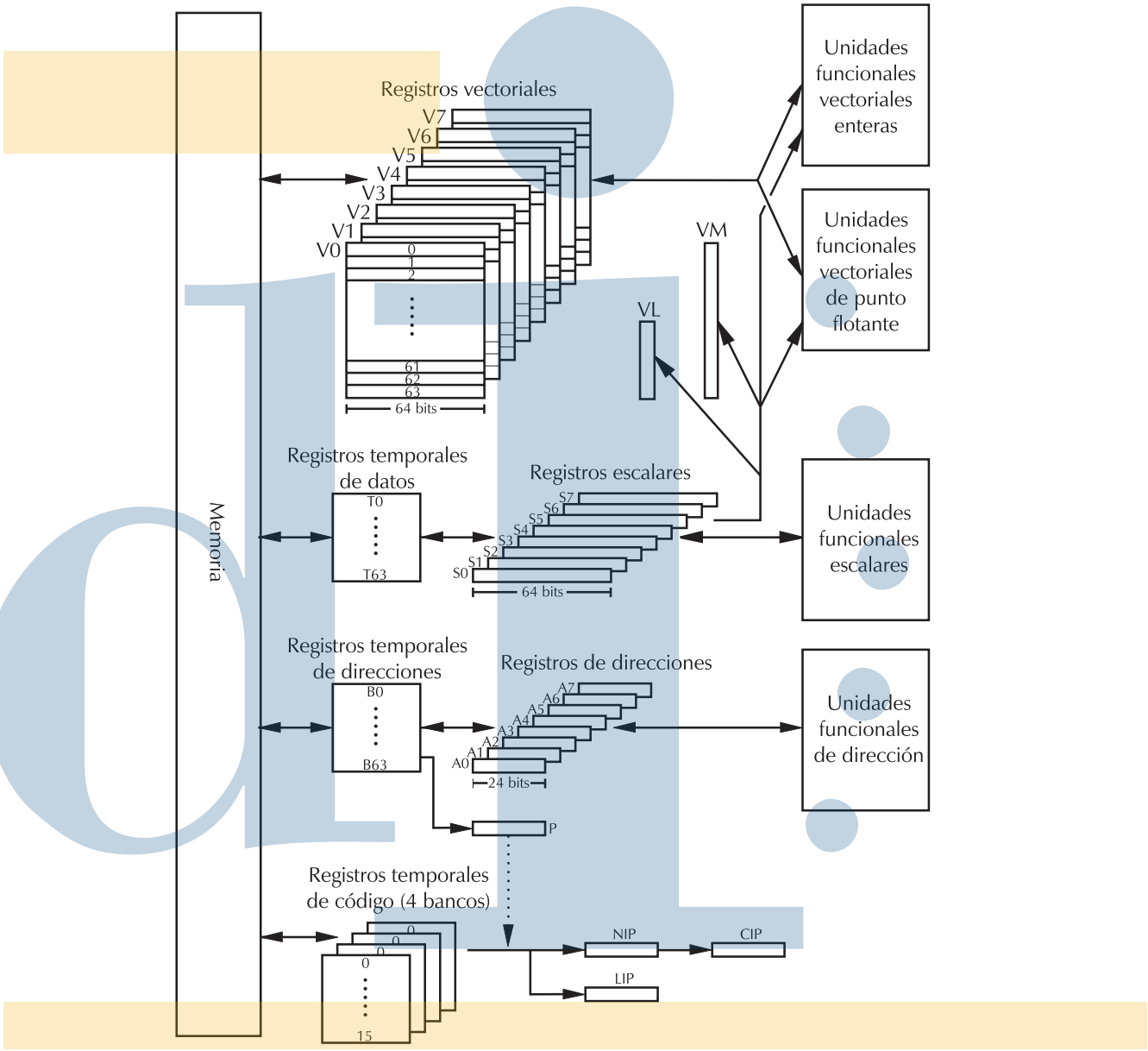

Fig. 3.3. Arquitectura del computador vectorial Cray-1 de Cray Research.

En cuanto al control, el registro P hace las funciones de contador de programa. Las instrucciones pueden tener tanto 16 como 32 bits, por ello, el registro de instrucción tiene dos partes: el CIP (*current instruction parcel*, parte de la instrucción en curso) y el LIP (*low instruction parcel*, parte baja de la instrucción en curso), ambas de 16 bits. En caso de que la instrucción tenga 16 bits sólo se emplea CIP, si tiene 32 se emplean ambos. También existe un registro para una instrucción en espera, o al menos parte de ella, que es el NIP (*next instruction parcel*, parte de la instrucción siguiente) y tiene 16 bits. Otros registros de esta máquina son el VM (*vector mask*, registro de máscara para vectores), que se utiliza para operaciones de empaquetamiento y desempaquetamiento de vectores y F, que es el registro de señalizadores.

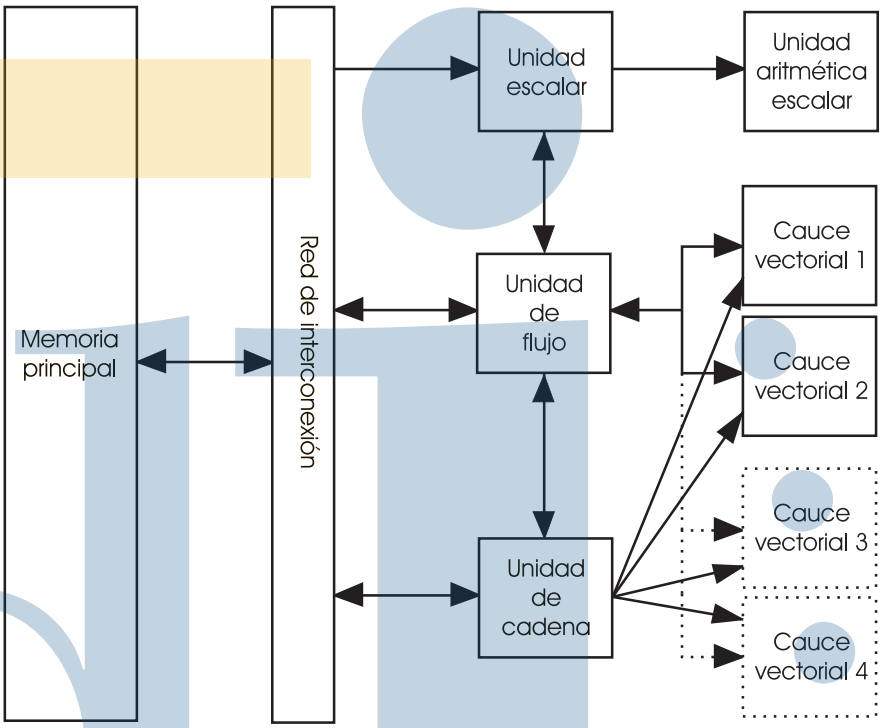

Fig. 3.4. Diagrama general de bloques del Cyber-205 de Control data.

3.8.2. El Cyber-205 de Control Data

Esta máquina es un curioso ejemplo de computador vectorial memoria-memoria. Por ello, la memoria de esta máquina es muy rápida y está entrelazada con cuatro vías. En la figura 3.4 se muestra un diagrama de bloques del Cyber-205. La ventaja de que este computador sea del tipo memoria-memoria radica en que la longitud de los vectores puede ser muy grande. Concretamente en el Cyber-205, la longitud máxima de los vectores es de 65635 palabras. El control de la ejecución de las instrucciones reside en la **unidad escalar**, que recibe y decodifica las instrucciones procedentes de la memoria y las reparte entre los diferentes bloques en función de su tipo: escalar, vectorial o de cadena (aquí la palabra cadena se refiere a cadenas de bits); a partir de ahí, diferentes instrucciones pueden ejecutarse de forma concurrente en diferentes unidades funcionales. La unidad de flujo sirve para controlar el tráfico de datos entre la memoria y los cauces. La unidad aritmética escalar tiene cinco procesadores aritméticos segmentados (suma/resta, multiplicación, logaritmo, desplazamiento y división/raíz cuadrada) y trabaja con operandos escalares de 32 o 64 bits. La unidad vectorial puede disponer de 1, 2 o 4 procesadores aritméticos que también disponen de varios cauces que pueden efectuar diversas operaciones sobre operandos de 32 y 64 bits (suma/resta, multiplicación, división, raíz cuadrada, operaciones lógicas y desplazamiento). La unidad de cadena sirve para procesar vectores de control, o vectores de máscara, que son muy útiles para trabajar con matrices y vectores dispersos. Estos vectores de control también son útiles para acceder a vectores no consecutivos en memoria. Este procesador está capacitado para efectuar encadenamiento entre los diversos cauces, de forma que puede hacer que la salida de un cauce vaya directamente a la entrada de otro.

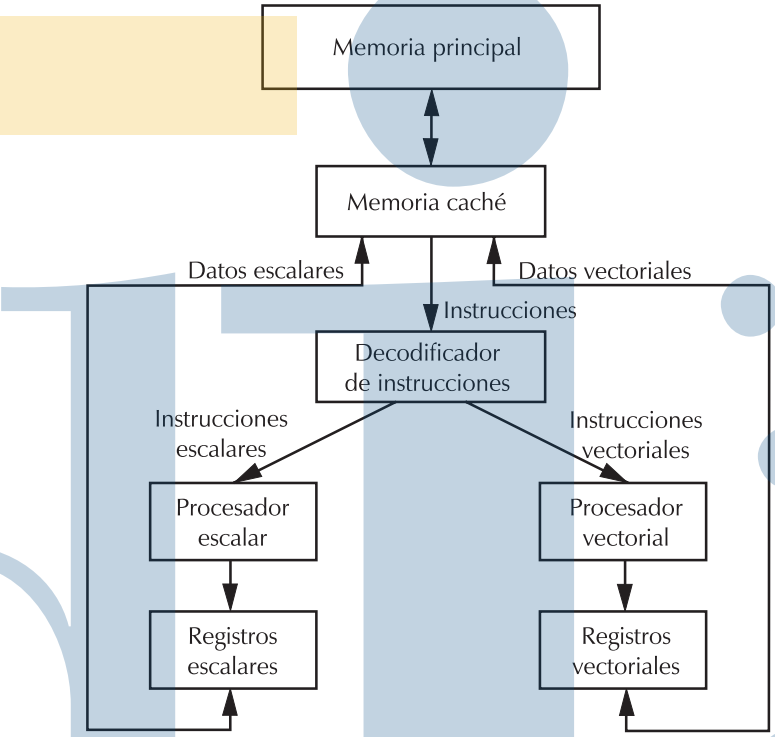

Fig. 3.5. Diagrama de bloques del IBM-3090 con la opción de proceso vectorial.

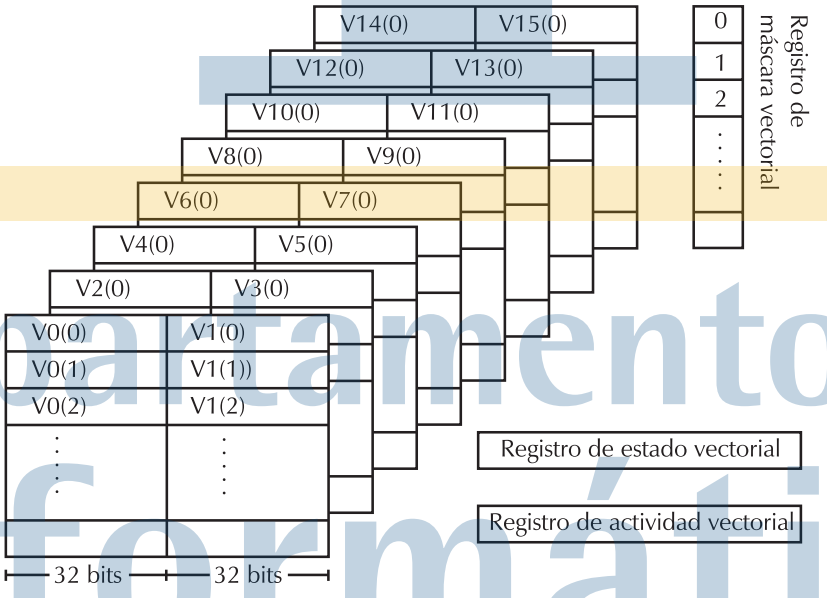

Fig. 3.6. Estructura de los registros vectoriales del IBM-3090 con la opción de proceso vectorial.

3.8.3. El IBM-3090

El procesamiento vectorial en esta máquina es una opción llamada *vector facility*. En la figura 3.5 puede verse un diagrama de bloques de esta opción. Por otra parte, en la figura 3.6 se muestra la estructura de los registros vectoriales. Como puede apreciarse en la citada figura, existen 16 registros de 32 bits que pueden unirse de dos en dos para convertirse en 8 registros de 64 bits. Cualquiera de los registros puede contener tanto números enteros como representados en punto flotante (ambos de 32 o 64 bits). En el diseño de la arquitectura no se determina de forma concreta la longitud de los registros vectoriales que pudiera oscilar entre 8 y 512. En la implementación del 3090 se eligió el número de 128. Esta elección se debió a un compromiso entre el tiempo perdido en los arranques, que será alto si los registros son pequeños, porque habrá que efectuar más arranques de instrucciones vectoriales si los vectores son largos, y el tiempo empleado en la carga y almacenamiento de los registros vectoriales, que será tanto más grande cuanta más longitud tengan. Como también puede apreciarse en la figura, son necesarios algunos registros de control: el registro de máscara vectorial, que, como ya sabemos, se utiliza para las operaciones de empaquetamiento y desempaquetamiento y, también, en el tratamiento de vectores y matrices dispersos; el registro de estado vectorial, que tiene diversas informaciones de control, entre las que se encuentra la información sobre la longitud de los vectores, y el contador de actividad vectorial que cuenta el tiempo empleado en la ejecución de las operaciones vectoriales.

Una de las más importantes características de esta máquina es que su juego de instrucciones vectoriales contiene instrucciones de tipo CVF (recuérdese: funciones vectoriales compuestas) tales como multiplicar y sumar, multiplicar y restar, etc., esto da mucha agilidad y velocidad a las operaciones sin tener que recurrir al encadenamiento.

3.8.4. El *Earth Simulator*

El *Earth Simulator* es un computador que ha estado durante más de un año en la cabeza del TOP500. Se trata de un multicomputador constituido por 640 nodos (llamados PN, *processor nodes*) interconectados. Cada nodo, a su vez, es un multiprocesador con 8 procesadores vectoriales (denominados AP, *arithmetic procesor*) que comparten una memoria de 16 Gbytes, un procesador de E/S y una unidad de control para el acceso remoto (RCU, *Remote Control Unit*). Cada uno de los 5.120 (640×8) procesadores vectoriales AP (figura 3.7), está constituido por una unidad de procesamiento superescalar (SU, *superescalar unit*) capaz de emitir 4 instrucciones por ciclo, y de una unidad de procesamiento vectorial (VU,*vectorial unit*). Estas dos unidades, junto con las correspondientes cachés y la unidad de control de acceso a la memoria principal están integradas en un único circuito integrado.

Cada unidad superescalar dispone de 128 registros de uso general, de cachés separadas de código y datos (64Kb. cada una) y cada unidad vectorial incluye 72 registros vectoriales de 256 componentes. En cada unidad vectorial hay 8 conjuntos de cauces, cada uno de los cuales dispone de 6 cauces vectoriales (suma/desplazamiento, multiplicación, división, operaciones lógicas, enmascaramiento y operaciones sobre memoria.

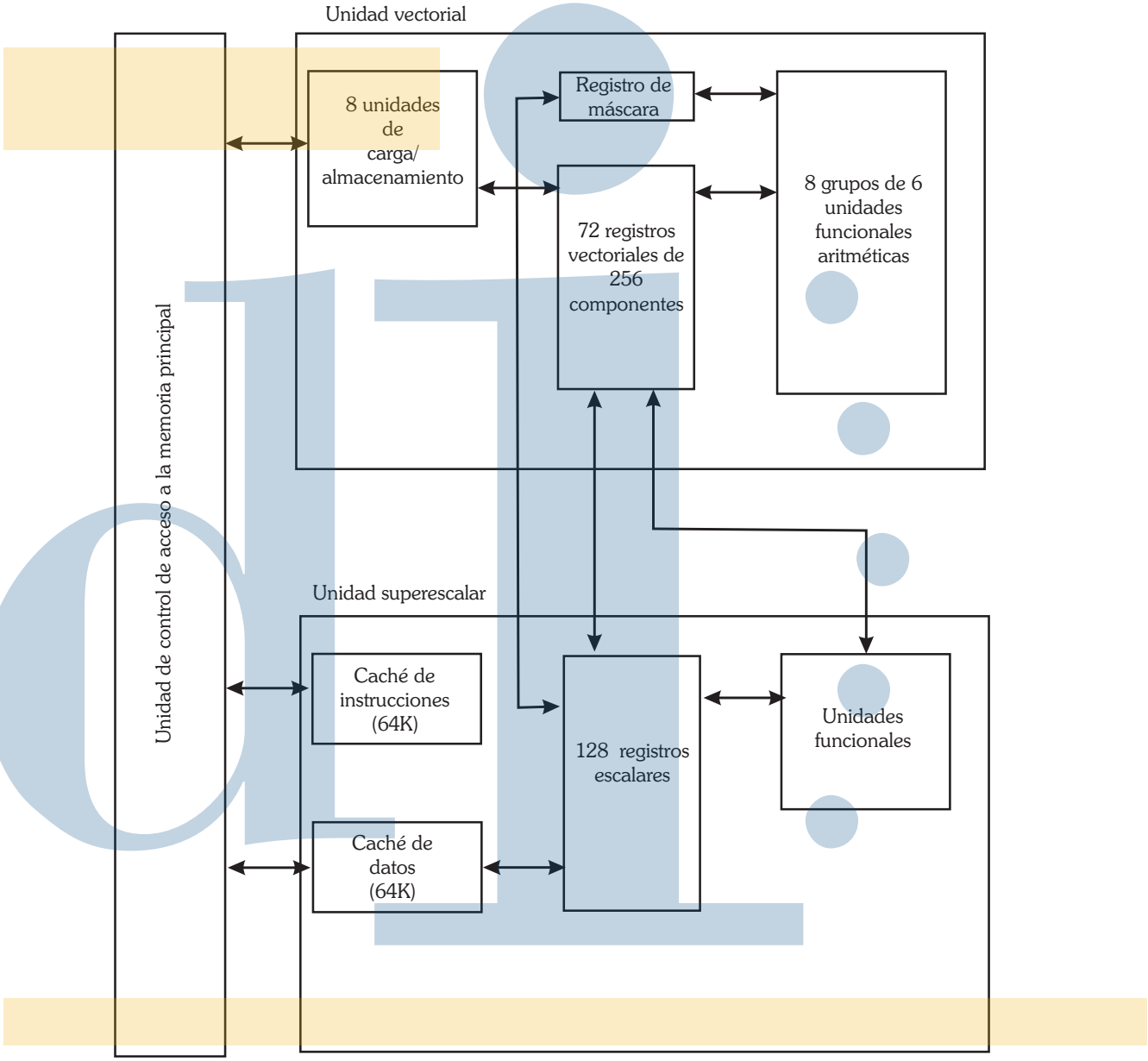

Fig. 3.7. Estructura de uno de los procesadores aritméticos del *Earth Simulator*

Bibliografía y referencias

DASGUPTA, S. 1989. *Computer Architecture: A Modern Synthesis*. Vol. 2: advanced topics. John Wiley & Sons.

GERMAIN-RENAUD, C., & SANSONNET, J.P. 1991. *Les ordinateurs massivement paralleles*. Armand Colin Editeur. Existe traducción al castellano: Ordenadores masivamente paralelos, Paraninfo, 1993.

HENNESSY, J.L., & PATTERSON, D.A. 1990. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann Publishers. Existe traducción al castellano: Arquitectura de computadores: Un enfoque

cuantitativo, McGraw-Hill, 1993.

HWANG, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill.

HWANG, K., & BRIGGS, F.A. 1984. *Computer Architecture and Parallel Processing*. McGraw-Hill. Existe traducción al castellano: Arquitectura de computadoras y procesamiento paralelo, McGraw-Hill, 1988.

RUSSELL, R.M. 1978. The CRAY-1 Computer System. *Communications of the ACM*, **21**(1).

STONE, H.S. 1990. *High-performance Computer Architecture*. 2 edn. Addison-Wesley.

CUESTIONES Y PROBLEMAS

3.1 Un procesador puede operar en forma escalar o vectorial. Cuando trabaja en forma vectorial (si la naturaleza del código lo permite), funciona 16 veces más rápido que cuando lo hace en modo escalar. Cierto programa necesita un tiempo t para ejecutarse en esta máquina. Se sabe que un 30 % de ese tiempo la máquina trabaja en modo vectorial, el resto funciona en modo escalar:

a) Calcular la ganancia de velocidad del sistema en las condiciones anteriores, comparada con el caso de que el sistema funcionara siempre en modo escalar.

b) Calcular la fracción f_v de código vectorizable.

c) Suponer que la relación de velocidad entre el modo vectorial y el escalar fuera del doble: calcular la ganancia de velocidad para este nuevo caso.

3.2 Sea un computador vectorial que puede trabajar en dos modos de ejecución: vectorial y escalar. Cuando trabaja en modo vectorial, su velocidad es de 20 Mflops y cuando lo hace en el modo escalar, es de 2 Mflops. Sea f_v la fracción de código vectorizable de un cierto programa:

a) Calcular la velocidad de ejecución media en Mflops en función de f_v.

b) Calcular la fracción de código vectorizable necesaria para obtener una velocidad de procesamiento media de 15 Mflops.

3.3 Sea el siguiente programa escrito en FORTRAN 90:

```
program prueba1
real a(200),b(100)
a(*)=b(*)
end
```

a) ¿Sería correcto?

b) Corregir el programa anterior de varias formas distintas para que sea correcto.

c) Escribir alguno de los programas obtenidos como fruto del apartado anterior en FORTRAN secuencial.

3.4 Sea el siguiente programa escrito en FORTRAN 90:

```
program prueba2
real x(100),y(200),z(300)
x(*)=y(1:*:2)+z(201:300)
z(1:*:3)=x(*)+y(1:100)
end
```

Escribir ese programa en FORTRAN secuencial.

3.5 Sea el siguiente fragmento de programa FORTRAN:

```
parameter (n=100)
real a(100),b(100),c(100),w(104)
do i=1,n
     w(i+4)=a(i)*b(i)+c(i)
enddo
```

Este fragmento de programa se ejecuta en una máquina vectorial cuyos registros son capaces de almacenar 16 componentes.

a) Aplicando la técnica de seccionamiento, escribir un programa FORTRAN, equivalente al anterior, de forma que el programa pueda ejecutarse en la citada máquina vectorial.

b) Escribir ese programa en FORTRAN 90.

c) Escribir el programa anterior con instrucciones máquina vectoriales (de una supuesta máquina vectorial genérica).

3.6 a) Escribir un programa FORTRAN para calcular el producto escalar de dos vectores de 300 componentes y lo deposite en la variable `pescalar`.

b) Aplicar la técnica de seccionamiento sobre el programa anterior para ejecutarlo sobre una máquina vectorial cuyos registros pueden almacenar hasta 32 elementos. Debe tenerse en cuenta que deben eliminarse las dependencias de datos.

c) Escribir el código anterior en FORTRAN 90.

d) Escribir ese programa en lenguaje máquina vectorial.

3.7 Sea el siguiente fragmento de programa escrito en FORTRAN:

```
parameter (n=300)
real a(n),b(n)
do i=1,n
     p=p+a(i)/b(i)
enddo
```

a) Escribir este programa en FORTRAN 90.

b) Seccionar el programa para que se pueda ejecutar eficientemente en un procesador vectorial cuyos registros pueden almacenar 64 componentes.

c) Escribir el resultado obtenido en FORTRAN 90.

3.8 Sea el siguiente código FORTRAN

```
real a(20),b(10)
...........
do i=2,20,2
     b(i/2)=a(i-1)+a(i)
enddo
```

a) Escribir un programa equivalente en FORTRAN 90.

b) Si se hubieran declarado las variables de la forma

```
real a(30),b(20)
```

¿Cambiaría en algo el programa anterior?

3.9 Sea el siguiente programa FORTRAN

```
program prueba3
real a(120),b(123),c(121)
do i=1,120,2
     a(i)=b(i+3)+c(i+1)
enddo
end
```

Escribir ese programa en FORTRAN 90.

3.10 a) Escribir un programa en FORTRAN secuencial y otro en FORTRAN 90 para evaluar el vector **y** dado por la combinación lineal siguiente:

$$\mathbf{y} = \sum_{i=1}^{300} a_i \mathbf{x}_i$$

donde cada $\mathbf{x}_i$ es un vector de 300 componentes.

b) ¿Existen dependencias de datos en alguno de los programas anteriores?

c) Escribir el programa anterior en lenguaje máquina vectorial.

d) ¿Cómo mejoraría el programa anterior si el procesador vectorial dispusiera de operaciones de tipo CVF?

3.11 Sean A y B dos matrices de 500×500 ya inicializadas. Se pretende efectuar el siguiente cálculo:

$$A_{ij} = A_{ij}(anterior) + 0{,}25 * (B_{i-1,j} + B_{i+1,j} + B_{i,j-1} + B_{i,j+1})$$

donde i y j varían entre 2 y 499.

a) Escribir un programa en FORTRAN secuencial para efectuar dicho cálculo.

b) Escribir el programa anterior en FORTRAN 90.

3.12 Sea el siguiente programa en FORTRAN:

```
program a
parameter (n=200,m=100)
real a(n), p(m), q(m)
........
Do i=1,n
    do j=1,m
       a(i) = a(i)+p(j)*q(j)
    enddo
enddo
........
```

Supóngase que ese programa se ejecuta en una máquina vectorial cuyos registros vectoriales pueden almacenar 64 componentes:

a) Escribir otra versión del mismo programa en que se optimice, en la medida de lo posible, el tiempo de ejecución del programa.

b) Escribir esa versión en FORTRAN 90.

c) Aplicar al programa obtenido la técnica de seccionamiento para adaptarlo a los registros vectoriales de la máquina.

d) Escribir el programa resultante en FORTRAN 90.

3.13 Sean dos conjuntos de vectores A=$\{\mathbf{a_1}, \mathbf{a_2}, ..., \mathbf{a_n}\}$ y B=$\{\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}\}$, cuyos vectores tiene m componentes cada uno. Se desea calcular el módulo del vector formado por los productos escalares $\mathbf{a_i} * \mathbf{b_i}$, es decir, el vector $(\mathbf{a_1} * \mathbf{b_1}, \mathbf{a_2} * \mathbf{b_2}, ..., \mathbf{a_n} * \mathbf{b_n})$ donde "*" representa el producto escalar.

a) Escribir un programa en FORTRAN para resolver el problema.

b) Vectorizar el programa aplicando la técnica de **expansión escalar** escribiendo el programa vectorial en FORTRAN 90.

c) Suponiendo que $m = 1000$, $n = 2000$ y que el programa se ejecuta en un procesador vectorial cuyos registros pueden almacenar 16 componentes, aplicar la técnica de seccionamiento al programa anterior.