está ubicado en un buffer de distribución que alimenta a dos estaciones de reserva individuales asignadas cada una de ellas a una unidad funcional de suma/resta (1 ciclo) y a una de multiplicación/división (2 ciclos y segmentada). Dibuje la evolución del estado del buffer de distribución, de las estaciones de reserva, del ARF, del RRF y del buffer de terminación. Tenga en cuenta que:

- Se utiliza planificación con lectura de operandos.
- El ARF consta de 8 registros, `R1` a `R8`, con valores iniciales de 10 a 80, respectivamente.
- El RRF consta de 4 registros `Rr1`, `Rr2`, `Rr3` y `Rr4`.
- El buffer de distribución distribuye 4 instrucciones/ciclo.
- Las estaciones de reserva individuales disponen de 2 entradas.
- En el mismo ciclo en que una unidad funcional genera el resultado, la estación de reserva actualiza sus bits de validez; esto permite que en el ciclo siguiente se pueda emitir otra instrucción y terminar la finalizada.

A2.15 Considere un esquema de planificación dinámica con lectura de operandos que utiliza renombramiento basado en un RRF independiente y con acceso asociativo. En el esquema propuesto, los valores de los operandos fuente que no tienen escrituras pendientes se leen directamente del ARF sin sufrir ningún tipo de renombramiento. Describa un posible proceso de renombramiento si se considerase que todos los operandos tienen que ser renombrados ¿Sobraría o faltaría algún campo en el ARF y en el RRF?

Capítulo 3

PROCESADORES VLIW Y PROCESADORES VECTORIALES

3.1. Guión-esquema

Los contenidos que se tratan a lo largo del tema se resumen en los siguientes puntos:

- Características y arquitectura de un procesador VLIW (*Very Long Instruction Word*).
- Ventajas e inconvenientes con respecto a los procesadores superescalares.
- Técnicas de planificación estática: Desenrollamiento de bucles, segmentación software y planificación de trazas.
- El estilo arquitectónico EPIC (*Explicity Parallel Instruction Computing*).
- Arquitectura básica de un procesador vectorial genérico. Procesador matricial.
- Características de las unidades vectoriales aritméticas. Registros vectoriales.
- Repertorio de instrucciones vectoriales.
- Vectorización de bucles de longitud desconocida mediante seccionamiento.
- Manipulación de vectores almacenados no consecutivamente en memoria. Vectorización de bucles con instrucciones condicionales.
- Medida del rendimiento de un bucle vectorizado.
- Características de una unidad de carga/almacenamiento vectorial.

3.2. Introducción

Desde el momento en que se construyó el primer computador electrónico digital, los esfuerzos en pro de obtener computadores con mayor capacidad de procesamiento que sus predecesores no han parado, ni es previsible que se detengan, ya sea profundizando en el paradigma de la computación clásica o mediante los avances que proporcionará la investigación en otras formas de computación (como, por ejemplo, la cuántica) a lo largo de los próximos años. En las últimas décadas, los continuos avances en arquitectura de computadores han dado lugar a la aparición de toda una plétora de nuevas ideas, conceptos y técnicas. Si, inicialmente, se perseguía la mejora de prestaciones mediante un incremento de la frecuencia de reloj con el fin de procesar más instrucciones por segundo, posteriormente se llegó al convencimiento de que lo mejor era una aproximación combinada, es decir, reducir la duración del ciclo de reloj pero intentando, simultáneamente, aumentar el paralelismo en todo lo relativo al procesamiento. Los estudios iniciales pusieron de manifiesto la existencia de dos vías para incrementar la potencia de un computador recurriendo al paralelismo: el paralelismo funcional y el paralelismo de datos.

El *paralelismo funcional* se obtiene mediante la replicación de las funciones de procesamiento que realiza el computador, pudiendo variar desde el procesamiento paralelo a nivel de instrucciones (granularidad fina) hasta llegar al procesamiento paralelo a nivel de programas (granularidad gruesa). Los procesadores segmentados, los procesadores superescalares y los procesadores VLIW representan la aproximación de más bajo nivel al paralelismo funcional, basada en el aumento del paralelismo a nivel de instrucciones. Otras aproximaciones al paralelismo funcional son los multiprocesadores y los multicomputadores, que se apoyan en el aumento del paralelismo a nivel de programas, funciones, bloques, etc., pudiendo a su vez recurrir para ello a distintos tipos de procesadores: superescalares, vectoriales, VLIW, etc. Aunque, en la actualidad, se prevé que los procesadores superescalares sean la plataforma sobre la que construir computadores de mayores prestaciones, a lo largo de las últimas dos décadas ya se observó que, por su propia concepción, un procesador superescalar planteaba límites al rendimiento que podría alcanzar como unidad de procesamiento aislada.

De forma muy resumida, la mejora del rendimiento de los procesadores superescalares se logra, principalmente, aumentando el número de instrucciones emitidas por ciclo de reloj. Ello se consigue gracias a un incremento en el número y complejidad de los recursos hardware de que consta el procesador: unidades funcionales, etapa de emisión, estaciones de reserva, buffer de terminación, múltiples ficheros de registros, etc. Todo esto provoca que estos procesadores consuman grandes cantidades de energía y produzcan mucho calor, pese a los mecanismos con que se les dota para minimizar estos inconvenientes. Cualquier intento de extraer mayor paralelismo a nivel de instrucción se traduce en un incremento de la complejidad hardware y del coste económico. Por esta razón, con el fin de extraer mayor paralelismo a nivel de instrucción sin que ello repercutiese en la complejidad del procesador, en la década de los 80 aparece por primera vez el término VLIW (*Very Long Instruction Word* – Palabra de Instrucción Muy Larga). Aunque la idea de los procesadores VLIW comienza a fraguarse en la década previa con las arquitecturas LIW (*Long Instruction Word*), no es hasta mediados de los 80 cuando surgen los primeros computadores comerciales basados en el concepto

VLIW. Aunque en los siguientes apartados se estudiará con mayor detalle, un procesador VLIW se caracteriza por emitir en cada ciclo de reloj una única instrucción pero que contiene varias operaciones o mini-instrucciones tipo RISC (enteras, flotantes, accesos a memoria, saltos, etc.) que se pueden ejecutar en paralelo en las diferentes unidades funcionales segmentadas de que consta el procesador. La responsabilidad de encontrar y planificar correctamente las instrucciones del código fuente para que se puedan codificar como instrucciones VLIW y ejecutar en paralelo es responsabilidad exclusiva del compilador, y no del hardware como sucede en los procesadores superescalares. El hardware de los procesadores superescalares se ocupa de detectar en tiempo de ejecución las instrucciones que no son dependientes y que se pueden emitir en paralelo a las unidades funcionales. Para ello utilizan múltiples recursos hardware (buffers y estaciones de reserva) pero que proporcionan una visión limitada y parcial de las instrucciones que componen el código ensamblador. El aumento de la visibilidad por hardware está limitado por la complejidad y los consabidos problemas que se derivan de ello. Por el contrario, un compilador dispone de una visión completa de todas las instrucciones que constituyen el código fuente: una ventana software de instrucciones ilimitada. Su capacidad para detectar en tiempo de compilación el paralelismo intrínseco existente entre las instrucciones del código fuente es mucho mayor que la del hardware, pero no solo por la ventana de visibilidad de que dispone sino porque no hay restricciones temporales, sencillamente tiene más tiempo para analizar el código y tomar decisiones. Además, el tener acceso a todo el código fuente proporciona mucha información sobre el comportamiento dinámico del programa (bucles, saltos, condiciones), que puede utilizarse para aumentar al máximo el paralelismo a nivel de instrucción.

La alternativa al paralelismo funcional lo constituye el paralelismo de datos representado, fundamentalmente, por los computadores vectoriales. El origen de este tipo de computadores está en la década de los 60, cuando se comienza a comprender que el incremento de las prestaciones de un procesador segmentado mediante un aumento de la frecuencia de reloj y de la profundidad de su cauce planteaba muchos problemas, bien causados por la imposibilidad para disminuir el ciclo de reloj o por la incapacidad para la búsqueda y emisión de varias instrucciones por ciclo de reloj. Resumiendo, resultaba tan complicado gestionar un procesador con una segmentación de n etapas como emitir n instrucciones simultáneamente. En aquellos momentos, la solución a esos problemas vino dada por el empleo de procesadores vectoriales, los cuales se caracterizan por proporcionar operadores de alto nivel que trabajan sobre vectores. Así, una operación vectorial tomaría dos vectores de n elementos, realizaría la correspondiente operación aritmética con ellos y generaría como resultado un vector de otros n elementos. Por lo tanto, una instrucción vectorial es equivalente a un bucle constituido por instrucciones secuenciales, donde en cada iteración se realiza una operación aritmética y se genera uno de los elementos del resultado. Hasta la aparición de nuevos conceptos de computación paralela, como son el procesamiento distribuido, el multiprocesamiento simétrico o el procesamiento masivamente paralelo, un computador vectorial era sinónimo de supercomputación y de computación de altas prestaciones. Hoy en día, aunque se siguen comercializando y utilizando en aplicaciones científicas (por ejemplo, predicción atmosférica, física de partículas), los computadores vectoriales han quedado desbancados por la relación coste/rendimiento que ofertan los multiprocesadores y multicomputadores.

De acuerdo con esto, los objetivos que se pretenden alcanzar con el estudio de este capítulo son:

- Conocer otras arquitecturas de computador distintas al concepto de segmentación superescalar.
- Entender otras formas de explotar el paralelismo a nivel de instrucción, como son las aproximaciones VLIW y EPIC.
- Conocer las características de un procesador VLIW junto con sus ventajas, sus inconvenientes, sus orígenes y su proyección de futuro.
- Conocer las técnicas de planificacion estática que se utilizan para evitar la problemática que conlleva el procesamiento VLIW: desenrrollamiento de bucles, segmentación software y planificación de trazas.
- Entender las aportaciones del concepto EPIC para evitar los inconvenientes de los procesadores VLIW.
- Conocer las características básicas de los procesadores vectoriales, su evolución y sus perspectivas de futuro.
- Entender las diferencias entre procesadores vectoriales y matriciales.
- Comprender cómo funcionan las unidades vectoriales aritmético-lógicas y de acceso a memoria para poder conseguir un flujo de datos muy elevado.
- Conocer un repertorio genérico de instrucciones vectoriales y las técnicas que se utilizan para resolver determinados inconvenientes como son el almacenamiento de datos en memoria con separación superior a la unidad, los bucles con instrucciones ejecutadas condicionalmente o el procesamiento de vectores con una longitud superior a la de los registros vectoriales.
- Saber estimar el tiempo de ejecución de un conjunto de instrucciones vectoriales y de un bucle escalar vectorizado mediante la técnica de seccionamiento.

3.3. El concepto arquitectónico VLIW

Un procesador VLIW es similar a un procesador superescalar en cuanto que puede emitir y terminar varias operaciones en paralelo. La diferencia es que el hardware no tiene que intervenir para descubrir el paralelismo entre instrucciones, ya que es responsabilidad exclusiva del compilador el generar un código binario formado por instrucciones que comporten operaciones paralelas. El hardware se limitará a emitir una instrucción por ciclo de reloj, sin preocuparse de la existencia de dependencias de datos. Esto implica que el paralelismo que puede proporcionar el hardware gracias a sus múltiples unidades funcionales debe ser conocido por el creador del compilador (o el programador en ensamblador) de forma que se pueda obtener un mayor aprovechamiento del hardware.

Por lo tanto, la diferencia clave entre el enfoque superescalar, ya sea CISC o RISC, y el VLIW es cómo se realiza la planificación de las instrucciones. En una arquitectura superescalar la planificación se realiza vía hardware y se adjetiva como *dinámica*, y en la VLIW se realiza vía software y se denomina *planificación estática*. Se define como estática ya que es el compilador el que establece la secuencia paralela de instrucciones de forma que las dependencias entre las operaciones que componen una instrucción VLIW no sean violadas y se reduzcan las detenciones de la segmentación. En un procesador VLIW se emite una instrucción por ciclo y una detención de una unidad funcional implica detener todas las unidades funcionales para mantener la sincronía en la emisión.

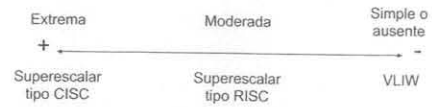


Figura 3.1: Evolución de la complejidad de los buffer de instrucciones, distribución, terminación y estaciones de reserva según el tipo de arquitectura.

La idea inmediata que se deriva de la aproximación VLIW es que la complejidad hardware, teóricamente, se reduce considerablemente y se desplaza hacia el software. Ya no son necesarios todos los recursos asociados a las etapas de decodificación, distribución y terminación de un procesador superescalar (Figura 3.1) pues la planificación de instrucciones viene dada por el compilador. Esto se traduce en una reducción de la cantidad de transistores necesarios, de la energía consumida, del calor disipado y, lógicamente, de la inversión económica a realizar en horas de ingeniería (por ejemplo, en horas de depuración). Otra ventaja de la aproximación VLIW es que la complejidad que implica el desarrollo del compilador se paga una sola vez, cuando se escribe el compilador, y no cada vez que se fabrica un chip. Además, nuevas mejoras en el compilador pueden introducirse una vez que los procesadores ya están en fase de producción. Por el contrario, en un procesador superescalar una mejora en el hardware dedicado a la distribución y emisión de las instrucciones implica modificaciones hardware, lo que incurre en todos los gastos asociados a la construcción de un nuevo circuito.

Pese a las evidentes ventajas del enfoque VLIW, la paradoja es que los procesadores superescalares triunfan comercialmente mientras que los VLIW han fracasado, por el momento. La explicación se encuentra en el análisis de las dos razones siguientes:

- La incapacidad para desarrollar compiladores que aprovechen al máximo las características del enfoque VLIW. Uno de los principales problemas es que el tamaño del código objeto para un procesador VLIW, en general, es mayor que para un procesador superescalar. Ello se debe a que no siempre es posible conseguir que el compilador rellene todas las operaciones de una instrucción VLIW con instrucciones del código fuente, estando forzado a completar instrucciones VLIW con operaciones NOP. Ello implica un uso inadecuado de los recursos del computador: espacio desaprovechado de la memoria principal y de la I-caché, unidades funcionales ociosas, aumento del tráfico de los buses, etc. Por ello se dice que el código VLIW es *código de baja densidad*.

- Los problemas de compatibilidad entre generaciones de procesadores VLIW. Procesadores VLIW con el mismo repertorio de instrucciones pero con una arquitectura diferente no son compatibles a efectos de código objeto. Por ejemplo, si las unidades funcionales tienen distintas latencias el código objeto no es compatible ya que el compilador ha planificado el código incluyendo las operaciones NOP adecuadas para que no haya violaciones de dependencias RAW en una arquitectura con unas latencias concretas. Además, dado que la emisión de una instrucción VLIW hacia las unidades funcionales es síncrona, es fundamental que no se produzcan detenciones en espera de resultados de instrucciones previas pues ello implica la detención de todas las unidades funcionales para asegurar la emisión simultánea de operaciones.

Los primeros computadores comerciales VLIW se produjeron a mediados de los años 80. Aunque algunos años antes ya se desarrollaron algunos prototipos, la primera máquina comercializada fue el computador Trace producido por la compañía Multiflow Computer creada por Joseph A. Fisher, profesor en la Universidad de Yale en el periodo 1979-1981, y primera persona en publicar un trabajo en el que aparece el término VLIW. La segunda compañía que puso en venta una máquina VLIW fue Cydrome con el computador Cydra-5. Aunque científicamente triunfaron por sus aportaciones, el éxito empresarial de estas dos compañías fue efímero ya que ambas cerraron a finales de los 80 (Multiflow en 1990 y Cydrome en 1988) por diversas razones. Por un lado, eran compañías pequeñas y desconocidas que vendían un producto completamente nuevo, incompatible con la base de computadores instalada, de precio elevado y sin un gran soporte comercial. Aunque hoy en día se considera que se adelantaron a su tiempo en un década, lo cierto es que la razón fundamental de su fracaso fue que no pudieron hacer frente a la revolución imparable que se produjo en el desarrollo de microprocesadores escalares y superescalares.

Los notables avances producidos por estas empresas no cayeron en el olvido. En el año 1989, Intel presenta el procesador RISC i860 (también conocido como 80860) que se caracterizaba por tener dos modos de operación: uno escalar y otro VLIW. Una instrucción VLIW del i860 podía contener hasta dos operaciones: una entera y una de coma flotante. La razón de su completo fracaso fue la ausencia de compiladores adecuados que aprovechasen al máximo sus características, lo que lo inhabilitaba como un procesador de propósito general. Sin embargo, en el ámbito de aplicaciones empotradas dedicadas al procesado digital de señal consiguió un relativo éxito ya que alcanzaba un rendimiento de entre 20 y 40 MFLOPS, algo muy elevado en su momento y, especialmente, para un procesador trabajando a una velocidad de 25-50 Mhz. El testigo VLIW fue tomado por la compañía Transmeta en el año 1995, fabricando hasta su cierre, en el año 2009, los procesadores Crusoe y Affineon de 128 y 256 bits de tamaño de instrucción, respectivamente. Estos procesadores se caracterizaban por su compatibilidad con el repertorio de instrucciones x86, lo que lograban traduciendo las instrucciones CISC a instrucciones VLIW mediante una capa de abstracción de software o máquina virtual, conocida como *Code Morphing Software* (CMS). Esta aplicación nativa VLIW se ejecutaba directamente en el núcleo VLIW y convertía el flujo entrante de instrucciones x86 en instrucciones VLIW. A finales de los 90, Intel reconsideró nuevamente las posibilidades de la alternativa VLIW diseñando la arquitectura IA-64 en colaboración con Hewlett-Packard. Esta arquitectura, vigente hoy en día, trata de resolver algunos de los problemas de dependencia hardware que plantea el enfoque VLIW y por ello recibe el nombre

de EPIC (*Explicity Parallel Instruction Computing* - Computación de Instrucciones Explícitamente Paralelas). Modelos de procesadores de propósito general desarrollados por Intel basándose en el concepto EPIC son los Itanium (producido en 2001 y 2002), Itanium 2 (lanzado el 2002) e Itanium 9300 (anunciado el 2010).

3.4. Arquitectura de un procesador VLIW genérico

La arquitectura de un procesador VLIW genérico se muestra en la Figura 3.2. La principal diferencia con respecto a un procesador superescalar es la ausencia de los elementos necesarios para la distribución, emisión y reordenación de las instrucciones, es decir, para su planificación dinámica. Estos elementos hardware no son necesarios dado que es el compilador quién realiza el trabajo de planificación gracias a un conocimiento preciso de las características del procesador. Aunque la Figura 3.2 no los recoge, todos aquellos mecanismos que utilizan los procesadores superescalares para garantizar el flujo continuo de instrucciones al procesador desde memoria, evitando así detener el cauce, son perfectamente válidos para un procesador VLIW. Algunos de estos mecanismos son las colas de prefetch o el buffer de fetch/overflow. La lógica de decodificación se utiliza para realizar la extracción de las operaciones de la instrucción VLIW y enviarlas a las correspondientes unidades funcionales junto con los valores de los operandos fuente y el identificador del registro destino. En las ventanas de emisión quedan preparados

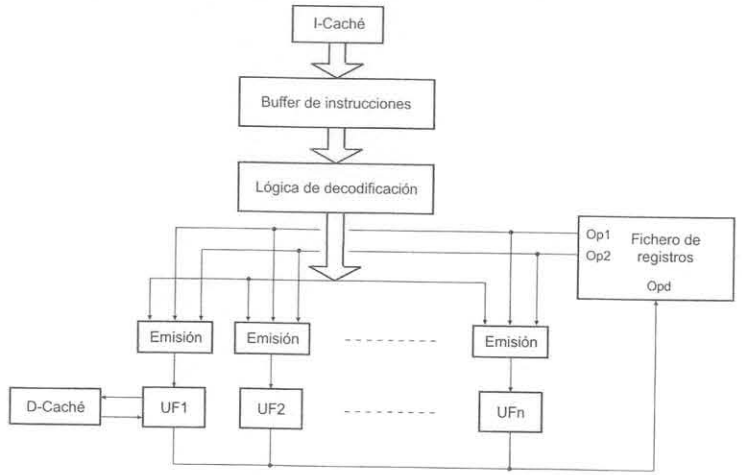


Figura 3.2: Arquitectura básica de un procesador VLIW genérico.

los códigos de operación y los valores de los operandos fuente para proceder a su emisión simultánea a las unidades funcionales. Observe que el fichero de registros debe disponer de los suficientes puertos de lectura para suministrar los operandos a todas las unidades funcionales en un único ciclo de reloj. Análogamente sucede con el número de puertos necesarios para la escritura de los resultados en el fichero de registros. Por lo tanto, a mayor número de unidades funcionales, mayor es la complejidad asociada al fichero de registros y a la red de interconexión. En esta arquitectura genérica la unidad funcional UF1 es la que se ocupa de las instrucciones de acceso a memoria, de ahí su conexión con la caché de datos, mientras que las restantes son unidades para la realización de operaciones aritmético/lógicas, considerando a los saltos y bifurcaciones como operaciones enteras. Al igual que en los procesadores superescalares, las unidades funcionales están segmentadas y presentan diferentes latencias.

Los repertorios de instrucciones de las arquitecturas VLIW siguen una filosofía RISC con la excepción de que el tamaño de instrucción es mucho mayor ya que contienen múltiples operaciones o mini-instrucciones. Una instrucción VLIW equivale a la concatenación de varias instrucciones RISC que se pueden ejecutar en paralelo, es decir, son implícitamente paralelas. Ello es posible dado que las operaciones recogidas dentro de una instrucción VLIW no presentan dependencias de datos, de memoria y/o de control entre ellas. Aunque varía según el diseño, el tamaño de instrucción VLIW más habitual es de 256 bits, si bien se han diseñado computadores con longitudes de hasta 512 bits (por ejemplo, la serie Multiflow Trace 14). La Figura 3.3 muestra el formato de instrucción del MultiFlow Trace 7. La instrucción tiene una longitud de 256 bits y consta de 7 operaciones de 32 bits y un campo de utilidad de 32 bits para fines diversos. De esta forma, el Trace 7 era capaz de iniciar simultáneamente siete instrucciones en cada ciclo de reloj.

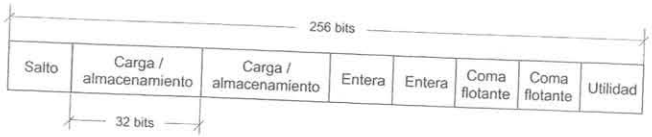


Figura 3.3: Formato de instrucción del computador Multiflow Trace 7.

Por lo general, el número y tipo de operaciones que contiene una instrucción VLIW se corresponde con el número y tipo de unidades funcionales existentes en el procesador. Si el formato de la instrucción VLIW admite una operación entera y una operación en coma flotante, el procesador debe contar con una unidad funcional entera y una unidad funcional de coma flotante. Una instrucción VLIW con menor número de operaciones que unidades funcionales prohibiría al compilador el extraer el máximo rendimiento del procesador al dejar unidades ociosas. Otro aspecto que debe tener en cuenta el compilador es si las operaciones que forman una instrucción pueden situarse en cualquier posición dentro de la instrucción. Si es así, la red de interconexión que comunica la lógica de decodificación con las ventanas de emisión se complica ya que debe contemplar cualquier posibilidad. Si por el contrario, cada operación tiene una posición fija asignada dentro de la instrucción, la red de interconexión necesaria

se simplifica al máximo ya que basta con una conexión directa entre el campo o slot asignado a una operación y la ventana de emisión asociada a la unidad funcional.

Debido a la ausencia de hardware para la planificación, los procesadores VLIW no detienen las unidades funcionales en espera de resultados. No existen interbloqueos por dependencias de datos ni hardware para detectarlas ya que el compilador se encarga de generar el código objeto para evitar estas situaciones. Para ello recurre a la inserción de operaciones NOP que fuerzan el avance del contenido del cauce de las unidades funcionales, evitando así su detención. Considere un sencillo procesador VLIW con un formato de instrucción que admite una operación entera y una operación en coma flotante; las latencias de las unidades entera y de coma flotante son 1 y 2 ciclos, respectivamente. Se pretende ejecutar el siguiente código:

```
i1: ADD    R1,R2,R3
i2: MULTD  F1,F2,F3
i3: MULT   R4,R1,R1
i4: DIVD   F4,F1,F1
i5: ADD    R5,R1,R1
```

donde se aprecian dos dependencias de datos RAW entre las instrucciones `i1`-`i3` e `i1`-`i5` por el contenido del registro `R1` y otra dependencia RAW entre `i2`-`i4` por el valor de `F1`. En un procesador superescalar, la instrucción `i4` quedaría en la estación de reserva en espera de que la instrucción `i2` finalizase su ejecución y publicase el valor de `F1` en el bus. Análogamente, las instrucciones `i3` e `i5` tendrían que esperar en su correspondiente estación por el resultado de `i1`. En un procesador VLIW, el compilador conoce las unidades funcionales con que cuenta el procesador y sus respectivas latencias. De acuerdo con esto, las instrucciones VLIW que produciría el compilador serían las siguientes:

```
I1: i1 + i2
I2: i3 + NOP
I3: i5 + i4
```

La dos primeras operaciones (o instrucciones RISC) se pueden colocar en la misma instrucción `I1` ya que se pueden ejecutar en paralelo al no tener ningún tipo de dependencia entre ellas. Ya que la latencia de la unidad entera es de un ciclo, la instrucción `I2` puede contener la operación entera `i3`. Sin embargo, el slot correspondiente a la operación en coma flotante debe quedar vacío ya que hay que esperar un ciclo por el resultado de `i2`. La instrucción `I3` está formada por la operación en coma flotante `i4` junto con la siguiente operación entera, la correspondiente a `i5`.

Del anterior ejemplo se puede deducir uno de los grandes problemas que plantea el enfoque VLIW: la necesidad de que el compilador encuentre instrucciones fuente independientes para poder rellenar todas las operaciones de que consta una instrucción VLIW. El no poder rellenar completamente una instrucción VLIW tiene dos consecuencias:

- El código objeto de un procesador VLIW es de mayor tamaño que el equivalente para un procesador superescalar. Aunque el compilador deje vació el slot correspondiente a la operación, el espacio que ocupa la instrucción en memoria no se reduce, permanece igual. Simplemente, se coloca como código de operación el de no operación, NOP.

- No se aprovechan al máximo los recursos del procesador ya que se tienen unidades funcionales ociosas, los buses transmiten información carente de valor y las memorias cachés pierden parte de su eficiencia.

Pero, además, aunque un compilador pueda ser capaz de evitar la detención de las unidades funcionales mediante una adecuada planificación de las operaciones, predecir qué accesos a memoria producirán fallos de caché es muy complicado. Esto condiciona a que las memorias caché sean *bloqueantes*, es decir, que tengan la capacidad de poder detener todas las unidades funcionales ante la aparición de un fallo de caché. A medida que el ancho de la instrucción VLIW es mayor y aumenta el número de referencias a memoria, la detención de todas las unidades funcionales llega a ser inaceptable, limitando gravemente el rendimiento del procesador.

Con el fin de incrementar el número de instrucciones independientes que se puedan utilizar para explotar al máximo el paralelismo de un procesador VLIW, limitando el número de huecos, se han desarrollado diferentes técnicas basadas en la manipulación de los bucles. Sin embargo, un efecto colateral de estas técnicas es que, pese a aumentar el paralelismo entre instrucciones, producen un aumento del tamaño del código objeto.

3.5. Planificación estática o basada en el compilador

Cuando un compilador VLIW recibe como entrada el código fuente de una aplicación, antes de producir como salida el código objeto, realiza una serie de tareas encaminadas a optimizar el código para aprovechar al máximo el paralelismo del procesador. Estas tareas pasan por producir tres elementos: un código intermedio, un grafo del flujo de control y un grafo del flujo de datos.

Las principales características del código intermedio es que está formado por sencillas instrucciones tipo RISC y las únicas dependencias de datos que permanecen en él son las verdaderas, las RAW. Las dependencias de salida WAW y las antidependencias WAR se han eliminado dado que el compilador dispone de un número infinito de registros simbólicos. Al reasignar los registros simbólicos en registros arquitectónicos aparecerán dependencias falsas pero el compilador las resolverá al generar el código VLIW mediante una adecuada política de planificación de las instrucciones que componen el código intermedio. El principal problema para el compilador es encontrar un número suficiente de operaciones en el código intermedio para formar instrucciones VLIW de forma que se maximice el rendimiento del procesador, es decir, que trabajen todas las unidades funcionales en paralelo y que no se produzcan paradas en la segmentación por dependencias. Un problema adicional con el que se encuentra el procesador es la necesidad de extraer el máximo paralelismo posible en tiempo de compilación, cuando todavía no se conoce con total seguridad la secuencia de ejecución del código.

Para poder generar un grafo de flujo de control es necesario, en primer lugar, conocer los bloques básicos de que consta el código intermedio. Un *bloque básico* se compone de un grupo de instrucciones que forman una línea de ejecución secuencial por lo que en su interior no existen instrucciones de salto con la salvedad de la última: no hay puntos intermedios de entrada y salida. Para obtener los bloques básicos se analiza el código intermedio teniendo en cuenta que:

- Una instrucción etiquetada (es decir, que es posible destino de un salto) o la siguiente instrucción a una instrucción de salto establecen el comienzo de un bloque básico o instrucción inicial.
- El bloque básico se compone por todas las instrucciones que hay desde la instrucción inicial hasta la siguiente instrucción de salto que se detecte.
- Los bloques se numeran de forma secuencial. La interconexión de las entradas y salidas de los diferentes bloques básicos conforma el diagrama de flujo de control.

Una vez que se conocen los bloques básicos que hay en el programa, las instrucciones de cada bloque se combinan para formar instrucciones VLIW. Para ello se recurre al grafo de flujo de datos que tiene asociado cada bloque. Un grafo de flujo de datos es un grafo dirigido en el que los nodos son las instrucciones de un bloque básico, y los arcos se inician en una instrucción de escritura en un registro (instrucción productora) y tienen como destino una instrucción que lee el valor de ese registro (instrucción consumidora). De esta forma, el grafo de flujo de datos muestra las secuencias de instrucciones que no presentan dependencias entre ellas y, por lo tanto, son susceptibles de combinar para formar instrucciones VLIW. A la combinación de instrucciones de un único bloque básico para producir instrucciones VLIW se le denomina *planificación local*. Algunas técnicas basadas en la planificación local son el *desenrollamiento de bucles* y la *segmentación software*. Sin embargo, en los numerosos estudios realizados se ha observado que el número medio de instrucciones que forman un bloque básico oscila entre cinco y seis, por lo que la cantidad de paralelismo que se puede extraer mediante técnicas de planificación local está limitada. Para superar estos inconvenientes se recurre a la *planificación global* consistente en combinar instrucciones de diferentes bloques básicos con el fin de producir una planificación con mayor grado de paralelismo. Una técnica que se apoya en la planificación global es la planificación de trazas.

La Figura 3.4.a muestra una secuencia de código intermedio en la que se han delimitado con cajas los bloques básicos que la componen. Los inicios de los bloques quedan determinados por las instrucciones etiquetadas y por las instrucciones situadas a continuación de un salto condicional o incondicional. La Figura 3.4.b corresponde al diagrama de flujo de control en el que la interconexiones de los bloques básicos, atendiendo a las posibles direcciones que puede seguir la ejecución del código, permiten visualizar los bucles existentes y las interrelaciones entre bloques. De cada bloque, el compilador intentará extraer el máximo paralelismo existente entre sus instrucciones teniendo en cuenta el diagrama de flujo de datos de cada bloque, las unidades funcionales del procesador y las latencias que presentan. La Figura 3.5.a corresponde a la secuencia de instrucciones que forma el primer bloque básico de la figura previa. En la Figura 3.5.b se muestra el diagrama de flujo de datos de esa secuencia en el que se puede apreciar la existencia de tres líneas paralelas de ejecución que el compilador utilizará para componer las instrucciones VLIW. Los números colocados junto a cada arco representan los ciclos de reloj que consume la unidad funcional en que se ejecuta cada operación y que el compilador utilizará para realizar la planificación de la secuencia de instrucciones VLIW.

Para completar un ejemplo de planificación local recurriendo a la secuencia de instrucciones del bloque básico de la Figura 3.5.a, considere que dispone de un procesador VLIW con un formato

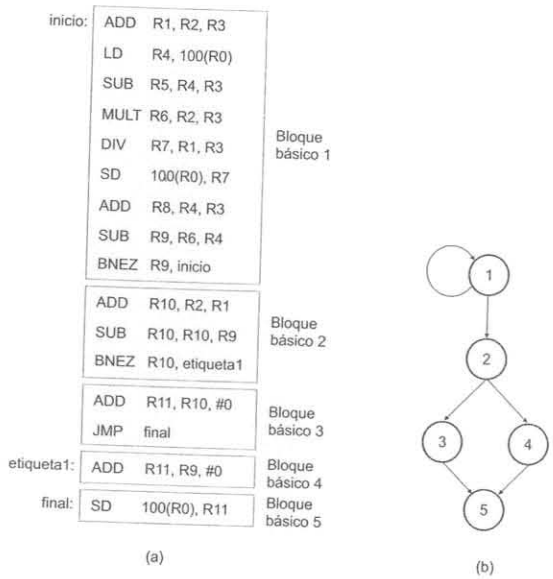


Figura 3.4: Ejemplo de secuencia de código intermedio con delimitación de los bloques básicos (a) y diagrama de flujo de control en el que se muestra la secuencia de ejecución de los bloques y la existencia de bucles (b).

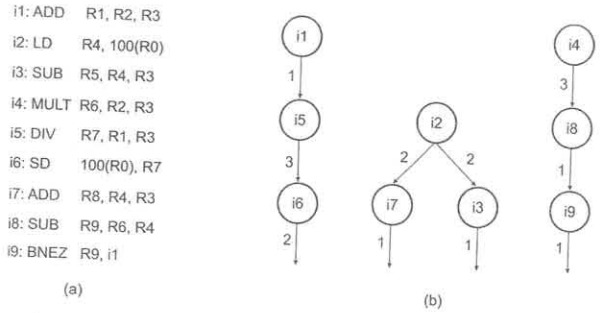


Figura 3.5: Secuencia de operaciones de un bloque básico (a) y diagrama de flujo de datos de la secuencia de instrucciones (b).

de instrucción que admite dos operaciones de suma/resta (un ciclo de latencia), una operación de multiplicación/división (tres ciclos de latencia), una operación de carga (dos ciclos de latencia) y otra de almacenamiento (dos ciclos de latencia). Por simplicidad, las instrucciones de salto son consideradas como instrucciones de suma/resta que escriben en el registro contador de programa, por lo que consumen un ciclo. La Figura 3.6 representa la secuencia planificada de instrucciones VLIW generada por el compilador utilizando como información las latencias de las unidades funcionales y las relaciones productor-consumidor entre instrucciones dadas por el diagrama de flujo de datos. En cada ciclo de reloj se emite una instrucción VLIW hacia las unidades funcionales pero la correcta planificación permite que no se produzcan detenciones. En el ejemplo se aprecia que no es posible ocupar todas las operaciones de una instrucción VLIW por lo que algunas unidades funcionales quedarán ociosas durante la ejecución del código. Además, si se considera que las instrucciones VLIW tienen una longitud de 20 bytes y que cada operación básica ocupa 4 bytes, el desaprovechamiento del código VLIW es del 64 % ya que el programa consume 100 bytes de almacenamiento en memoria pero solo un 36 % está ocupado por operaciones útiles.

	Operaciones Suma / Resta 1	Operaciones Suma / Resta 2	Operaciones Mult / Div	Operaciones de carga	Operaciones de almacenamiento
inicio:	ADD R1, R2, R3	-------	MULT R6, R2, R3	LD R4, 100(R0)	-------
	-------	-------	DIV R7, R1, R3	-------	-------
	SUB R5, R4, R3	ADD R8, R4, R3	-------	-------	-------
	SUB R9, R6, R4	-------	-------	-------	-------
	BNEZ R9, inicio	-------	-------	-------	SD 100(R0), R7

Figura 3.6: Codificación de las operaciones del bloque básico en instrucciones VLIW.

3.6. Desenrollamiento de bucles

El desenrollamiento de bucles (*loop unrolling*) es una técnica de planificación local que permite aprovechar el paralelismo existente entre las instrucciones que componen el cuerpo de un bucle. Básicamente, la técnica consiste en replicar múltiples veces el cuerpo del bucle utilizando diferentes registros en cada réplica y ajustar el código de terminación en función de las veces que se replique el cuerpo. Tres son las ventajas que se derivan de esta operación. La primera de ellas es que se reduce el número de iteraciones del bucle, lo que implica una reducción de las dependencias de control al ejecutarse menos instrucciones de salto. La segunda ventaja es que el total de instrucciones ejecutadas es menor ya que no solo se eliminan saltos sino que se reduce el número de instrucciones de incremento/decremento de los índices que se utilicen en el bucle. La tercera ventaja es que se proporciona al compilador un mayor número de oportunidades para planificar las instrucciones ya que al desenrollar el bucle queda mucho más cálculo al descubierto al incrementarse el tamaño de los fragmentos de código lineal. De esta forma, el compilador puede realizar una planificación más efectiva y generar código VLIW más compacto.

Para entender con claridad en qué consiste el desenrollamiento de un bucle aplicado a los procesadores VLIW se recurrirá a un ejemplo. Observe el siguiente fragmento de código intermedio generado por el compilador:

```
inicio: LD    F0,0(R1)
        ADDD  F4,F0,F2
        SD    0(R1),F4
        SUBI  R1,R1,#8
        BNEZ  R1,inicio
```

El código representa un bucle en el que se realiza la suma de una constante, almacenada en el registro F2, a todos los elementos de un vector almacenado en memoria cuyos elementos son de doble precisión, es decir, tienen una longitud de 8 bytes. El índice que permite acceder a los elementos del bucle se almacena en el registro entero R1, que inicialmente contiene la posición en memoria que ocupa el último elemento del vector. Si, por ejemplo, se desenrolla el cuerpo del bucle cuatro veces se obtiene la siguiente secuencia de instrucciones:

```
inicio: LD    F0,0(R1)     % Iteración i
        ADDD  F4,F0,F2
        SD    0(R1),F4
        LD    F6,-8(R1)    % Iteración i+1
        ADDD  F8,F6,F2
        SD    -8(R1),F8
        LD    F10,-16(R1)  % Iteración i+2
        ADDD  F12,F10,F2
        SD    -16(R1),F12
        LD    F14,-24(R1)  % Iteración i+3
        ADDD  F16,F14,F2
        SD    -24(R1),F16
        SUBI  R1,R1,#32    % Decremento en 4 elementos
        BNEZ  R1,inicio
```

La secuencia de código generada contiene cuatro copias o réplicas del cuerpo original del bucle con sus correspondientes registros. Además, observe que el índice se decrementa de cuatro en cuatro elementos, que son los elementos del vector que se procesan en cada iteración del nuevo bucle desenrrollado. El desenrollamiento ha permitido que el número de iteraciones necesarias para recorrer todo el bucle se haya dividido por cuatro y el total de instrucciones ejecutadas sea inferior ya que se han eliminado dos instrucciones por cada nuevo desenrollamiento (una de salto y otra para decrementar el índice), lo que representa un ahorro de seis instrucciones en cada iteración del bucle desenrrollado. La secuencia desenrollada se puede reorganizar con el fin de ilustrar la existencia de un mayor paralelismo entre las instrucciones. Esta reorganización consiste en agrupar las instrucciones por tipo:

```
inicio: LD    F0,0(R1)     % i1
        LD    F6,-8(R1)    % i2
        LD    F10,-16(R1)  % i3
```



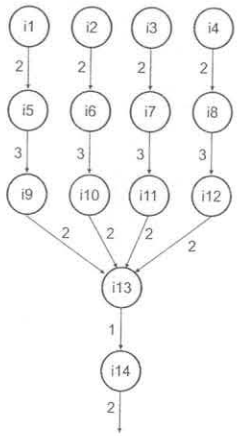
```
        LD    F14,-24(R1)  % i4
        ADDD  F4,F0,F2     % i5
        ADDD  F8,F6,F2     % i6
        ADDD  F12,F10,F2   % i7
        ADDD  F16,F14,F2   % i8
        SD    0(R1),F4     % i9
        SD    -8(R1),F8    % i10
        SD    -16(R1),F12  % i11
        SD    -24(R1),F16  % i12
        SUBI  R1,R1,#32    % i13
        BNEZ  R1,inicio    % i14
```

Considere que dispone de un procesador VLIW dotado de tres unidades funcionales: una para operaciones enteras (un ciclo de latencia), una para operaciones en coma flotante (tres ciclos de latencia) y otra para operaciones de carga/almacenamiento (dos ciclos de latencia). Las operaciones de salto se ejecutan en la unidad entera con un hueco de retardo de un ciclo por lo que permiten la planificación de una instrucción a continuación. De acuerdo con esta información, una posible planificación del código desenrollado para el procesador VLIW se recoge en la Figura 3.7.

En cada ciclo de reloj, la lógica de emisión del procesador emite una instrucción del código VLIW de

	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras y Saltos
inicio:	LD F0, 0(R1)	------	------
	LD F6, -8(R1)	------	------
	LD F10, -16(R1)	ADDD F4, F0, F2	------
	LD F14, -24(R1)	ADDD F8, F6, F2	SUBI R1, R1, #32
	------	ADDD F12, F10, F2	------
	SD 32(R1), F4	ADDD F16, F14, F2	------
	SD 24(R1), F8	------	------
	SD 16(R1), F12	------	------
	SD 8(R1), F16	------	BNEZ R1, inicio

(a)



(b)

Figura 3.7: Instrucciones VLIW (a) y diagrama de flujo de datos (b) generado a partir de la secuencia de código desenrollada.

la Figura 3.7 hacia las unidades funcionales, no deteniéndose en ningún momento el proceso de emisión como consecuencia de una dependencia de datos entre instrucciones (salvo por fallos en el acceso a la caché de datos). Esta emisión continua de instrucciones se consigue gracias a la planificación estática que realiza el compilador y que asegura que las dependencias se satisfacen al mantener la separación necesaria en ciclos de reloj entre parejas de operaciones que plantean riesgos. Esta es la razón por la que las operaciones de suma en coma flotante no se inician en el primer ciclo sino en el tercero ya que es necesario tener en cuenta el retardo asociado a las instrucciones de carga, esto es, dos ciclos; análogamente sucede con las instrucciones de almacenamiento que deben esperar a que los resultados de las operaciones de suma en coma flotante estén disponibles. Por otro lado, la dependencia WAR existente entre la lectura del registro R1 por las instrucciones de almacenamiento y su escritura por la instrucción SUBI se ha resuelto teniendo en cuenta el efecto que produce el decremento adelantado del índice. Por ese motivo, las cuatro instrucciones de almacenamiento se modifican para recoger el decremento adelantado del registro R1: como se decrementa por adelantado en 32, se suma un valor de 32 a los desplazamientos de los almacenamientos, 0, -8, -16 y -24, dando como resultado que el adelanto en la escritura de R1 provoque que los nuevos desplazamientos tengan que pasar a ser 32, 24, 16 y 8.

En este ejemplo se vuelve a poner de manifiesto el problema que plantea el tamaño del código VLIW. Si la instrucción VLIW y cada operación escalar necesitan un tamaño de 12 y 4 bytes, respectivamente, el espacio de almacenamiento desaprovechado es, aproximadamente, del 50 %. Las instrucciones VLIW ocupan 108 bytes (9 instrucciones*12 bytes) mientras que las operaciones originales consumen únicamente 56 bytes (14 instrucciones*4 bytes) en el caso del bucle desenrollado y 20 bytes (5 instrucciones*4 bytes) en el bucle original. Donde sí se aprecia una mejora es en el rendimiento. Si el vector constase de 1000 elementos, el cuerpo del bucle desenrollado cuatro veces y planificado consumiría, en el mejor de los casos, un total de 3500 ciclos (250 iteraciones*14 instrucciones) mientras que el VLIW únicamente necesitaría 2250 ciclos (250 iteraciones*9 instrucciones). En este caso, el enfoque VLIW es un 55 % más rápido que la aproximación escalar desenrollada y planificada.

Una vez que se conoce la versión VLIW del bucle original que se obtiene tras aplicar la técnica de desenrollamiento de bucles, es interesante compararla con la versión VLIW que se obtendría del bucle original sin recurrir a ninguna técnica de planificación local. La Figura 3.8 muestra el código VLIW que consta de seis instrucciones y tiene un tamaño de 72 bytes, de los cuales solo 20 bytes están ocupados con operaciones. En lo que respecta a la velocidad de ejecución del código VLIW de la figura, si el vector constase de 1000 elementos se tardaría en procesarlo 6000 ciclos de reloj, cifra muy superior al tiempo de ejecución de 2250 ciclos obtenido tras aplicar desenrollamiento. Por lo tanto, la utilización de esta técnica de planificación local en este sencillo ejemplo como paso previo a la generación del código VLIW permite acelerar la ejecución un 166 %.

3.7. Segmentación software

La segmentación software es otra técnica que se utiliza para intentar aprovechar al máximo el paralelismo a nivel de instrucción existente en el cuerpo de algunos bucles. La técnica consiste en

	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras y Saltos
inicio:	LD F0, 0(R1)	------	------
	------	------	------
	------	ADDD F4, F0, F2	------
	------	------	------
	------	------	SUBI R1, R1, #8
	SD 8(R1), F4	------	BNEZ R1, inicio

Figura 3.8: Instrucciones VLIW generadas a partir de la secuencia original del bucle sin aplicar ninguna técnica de planificación local.

producir un nuevo cuerpo del bucle compuesto por la intercalación de instrucciones correspondientes a diferentes iteraciones del bucle original. Al reorganizar un bucle mediante segmentación software, siempre es necesario añadir unas instrucciones de arranque (el prólogo) antes del cuerpo del bucle y otras de terminación tras su finalización (el epílogo). Una vez que se dispone del nuevo bucle ya reorganizado, y dado que las operaciones que lo forman pertenecen a diferentes iteraciones del bucle original y son dependientes, es posible planificarlas y emitirlas en paralelo bajo la forma de instrucciones VLIW. A diferencia de lo que sucede con el desenrollamiento de bucles, un bucle escalar reordenado mediante esta técnica no consta de más instrucciones que el bucle original salvo en lo que se refiere a las instrucciones que forman el prólogo y el epílogo.

La Figura 3.9 presenta un esquema de la aplicación de esta técnica a un bucle compuesto por cuatro

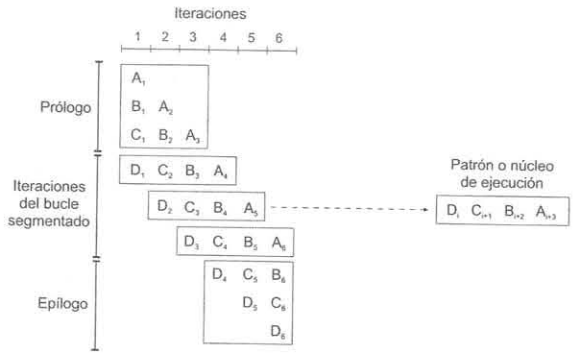


Figura 3.9: Aplicación de la segmentación software al cuerpo de un bucle genérico.

instrucciones genéricas, *A*, *B*, *C* y *D*, donde cada una de ellas consume un ciclo de reloj y presenta una dependencia RAW con la instrucción que la precede, esto es, la *D* depende del resultado de la *C*, la *C* del de la *B* y la *B* del de la *A*. En la Figura 3.9 se puede apreciar que tras seis iteraciones del bucle original aparece un patrón de ejecución compuesto por instrucciones pertenecientes a cuatro iteraciones diferentes del bucle original y que ya no presentan dependencias RAW entre ellas. Este patrón de ejecución o núcleo constituye el cuerpo del nuevo bucle reorganizado. También se puede observar que las instrucciones que hay antes de la aparición del primer patrón forman el prólogo del nuevo bucle y las que se encuentran tras el último patrón constituyen el epílogo. Tanto prólogo como epílogo están constituidos por las instrucciones que son necesarias para completar las iteraciones afectadas por las instrucciones que conforman el patrón de ejecución. En el ejemplo de la Figura 3.9, el prólogo consta de las instrucciones necesarias para completar las tres primeras iteraciones del bucle original y el epílogo está formado por las instrucciones que quedan pendientes para completar las tres últimas iteraciones. Dado que mediante esta técnica se ejecutan al mismo tiempo instrucciones provenientes de múltiples iteraciones se la conoce también como *planificación policíclica*.

Una vez que se aplica la técnica de segmentación software a un bucle, las instrucciones que componen el prólogo, el patrón y el epílogo se utilizan para generar la versión VLIW del bucle original. A continuación, se presenta otro ejemplo en el que la técnica se aplica al ya conocido bucle consistente en sumar una constante a todos los elementos de un vector donde el contenido del registro R1 marca la posición del último elemento del vector:

```
inicio: LD    F0,0(R1)
        ADDD  F4,F0,F2
        SD    0(R1),F4
        SUBI  R1,R1,#8
        BNEZ  R1,inicio
```

En la Figura 3.10 se puede observar el patrón de ejecución que se obtiene tras iterar el bucle seis veces y teniendo en cuenta las latencias de las unidades funcionales: dos ciclos para los accesos a memoria, tres ciclos para las operaciones en coma flotante. No se han incluido las instrucciones que decrementan el valor de R1 aunque sí se ha reflejado el necesario decremento en los desplazamientos de las instrucciones de carga y almacenamiento. Además del patrón, se pueden observar las instrucciones que forman el prólogo y el epílogo y que no son más que las instrucciones necesarias para iniciar y completar las iteraciones del bucle que se recogen en el patrón. Una vez que se dispone del patrón, el compilador debe proceder a generar el código VLIW. Una secuencia de código VLIW genérico correspondiente al bucle segmentado se muestra en la Figura 3.11 e incluye tanto el prólogo como el epílogo. Se puede apreciar que las instrucciones VLIW se obtienen prácticamente de forma directa del patrón. Su adaptación a un procesador VLIW específico implicaría generar el patrón teniendo en cuenta el número de unidades funcionales disponibles y sus latencias.

Si en este ejemplo se considera que las instrucciones VLIW son de 16 bytes, el tamaño total del código es de 176 bytes. En lo referente al tiempo para procesar un vector de 1000 elementos, la aproximación VLIW emplearía 1010 ciclos de los cuales 5 corresponderían al prólogo, 4 al epílogo

y 1000 a las iteraciones del bucle. Aunque el concepto en que se basa es sencillo, la segmentación software puede llegar a ser extremadamente complicada de aplicar en un procesador VLIW cuando hay instrucciones condicionales en el cuerpo del bucle que impiden la aparición de un patrón de comportamiento regular, cuando el número de iteraciones no se conoce a priori o cuando el número y tipo de operaciones que forman el patrón de ejecución no se ajusta al formato de la instrucción VLIW.

En este último ejemplo ha quedado reflejado la necesidad de recurrir a técnicas de planificación que permitan reubicar operaciones de diferentes bloques básicos con el fin de intentar ocupar aquellos huecos que quedan libres en el código VLIW cuando se recurre únicamente a la planificación local y aislada de

Iteración 1	Iteración 2	Iteración 3	Iteración 4	Iteración 5	Iteración 6
LD F0, 0(R1)					
	LD F0, -8(R1)				
ADDD F4, F0, F2		LD F0, -16(R1)			
	ADDD F4, F0, F2		LD F0, -24(R1)		
		ADDD F4, F0, F2		LD F0, -32(R1)	
SD 0(R1), F4			ADDD F4, F0, F2		LD F0, -40(R1)
	SD -8(R1), F4			ADDD F4, F0, F2	
		SD -16(R1), F4			ADDD F4, F0, F2
			SD -24(R1), F4		
				SD -32(R1), F4	
					SD -40(R1), F4

Patrón

Figura 3.10: Esquema del patrón de ejecución obtenido al aplicar la técnica de segmentación software.

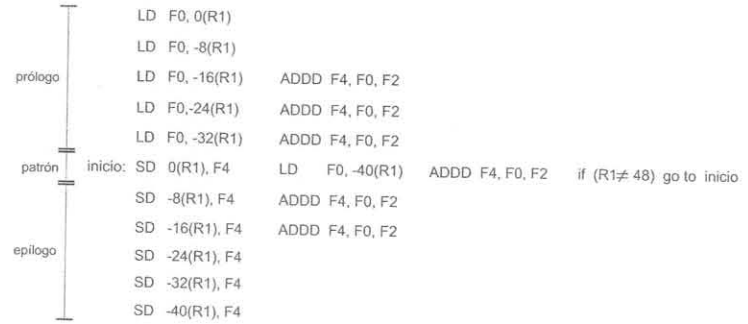


Figura 3.11: Instrucciones VLIW genéricas obtenidas a partir de la secuencia del bucle segmentado.

cada bloque básico. Este es el objetivo de las técnicas de planificación global, entre las que se encuentran la planificación de hiperbloques, la planificación de superbloques y la planificación de trazas, objetivo del siguiente apartado.

3.8. Planificación de trazas

La planificación de trazas (*trace scheduling*) es una técnica de planificación global que permite tratar una secuencia de bloques básicos como si fuese uno único y, a partir de ese nuevo bloque extendido dotado de un mayor número de operaciones, producir código VLIW más óptimo mediante una adecuada planificación de las operaciones. La diferencia de unas técnicas de planificación global con otras (trazas, hiperbloques, superbloques, etc.) recae en cómo manipulan diferentes estructuras de bloques básicos para generar bloques básicos de mayor tamaño. En el caso de la planificación de trazas, ésta se aplica a secuencias de bloques básicos donde cualquier bloque puede tener varias entradas y salidas. Sin embargo, la planificación de hiperbloques se orienta hacia el tratamiento de secuencias de bloques básicos con una única entrada pero cuyos bloques intermedios pueden presentar múltiples salidas.

La planificación de trazas es una combinación de dos pasos que se efectúan de forma consecutiva: la *selección de la traza* y la *compactación o compresión de la traza*. La selección de la traza consiste en encontrar un conjunto de bloques básicos que formen una secuencia de código sin bucles. A ese conjunto de bloques básicos es a lo que se le denomina *traza*. La selección de los bloques básicos dentro del grafo de flujo de control se realiza especulando sobre cuáles son las rutas o caminos de ejecución considerados como más probables. Para conocer cuáles serán las rutas de ejecución más probables, el compilador recurre a un *grafo de flujo de control con pesos o ponderado*, que es similar al grafo de flujo de control pero asignando a cada bloque básico una etiqueta o peso que indica la probabilidad o frecuencia de ejecución. La obtención de los pesos se puede realizar de diferentes formas: perfiles de ejecución del programa, estimaciones software, planificación estática de saltos, etc. Por ello, una traza representa el camino de ejecución más probable. El siguiente código corresponde a un bucle en el que se recorre un vector de números enteros, `A`, y dos vectores de valores en coma flotante, `X` e `Y`. En cada iteración del bucle, y en función del contenido de `A[i]`, se incrementa `X[i]` o se decrementa `Y[i]` con una constante almacenada en `F2`:

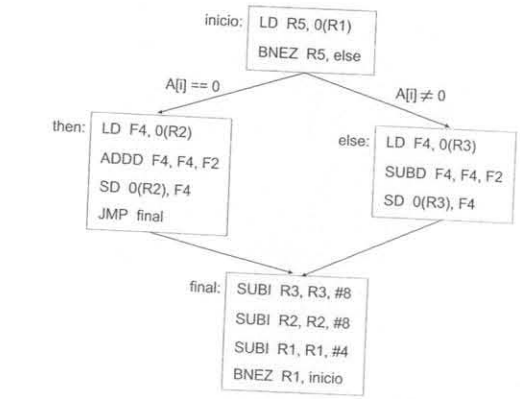
```
for (i=0;i<n;i++)
   if (A[i]==0) then
      X[i]:=X[i]+a;
   else
      Y[i]:=Y[i]-a;
   end if;
end for;
```

La representación en forma de código intermedio del cuerpo del bucle corresponde al siguiente fragmento de código intermedio:


```
inicio: LD    R5, 0(R1)    % Cargar A[i]
        BNEZ  R5,else      % Si (A[i]<>0) ir a else
  then: LD    F4,0(R2)     % Cargar X[i]
        ADDD  F4,F4,F2     % X[i]:=X[i]+a
        SD    0(R2),F4     % Almacenar X[i]
        JMP   final
  else: LD    F4,0(R3)     % Cargar Y[i]
        SUBD  F4,F4,F2     % Y[i]:=Y[i]-a
        SD    0(R3),F4     % Almacenar Y[i]
 final: SUBI  R2,R2,#8     % Decrementar en 8 bytes
        SUBI  R3,R3,#8     % Decrementar en 8 bytes
        SUBI  R1,R1,#4     % Decrementar en 4 bytes
        BNEZ  R1,inicio    % Nueva iteración
```

La anterior secuencia de código intermedio da lugar al diagrama de flujo de control de la Figura 3.12.a compuesto por cuatro bloques básicos. El compilador, en base a una técnica cualquiera, ha considerado que la secuencia de ejecución más probable es la que corresponde a que `A[i]` sea igual a cero lo que implica ejecutar la rama que produce el incremento de los elementos de `X`. La Figura 3.12.b muestra el programa VLIW que se obtendría de codificar el código intermedio recurriendo a un procesador con las mismas características que el utilizado en los ejemplos anteriores: tres unidades funcionales de las cuales una para operaciones enteras (un ciclo de latencia), una para operaciones en coma flotante (tres ciclos de latencia) y otra para operaciones de carga/almacenamiento (dos ciclos de latencia), las operaciones de salto se ejecutan en la unidad entera con un hueco de retardo de un ciclo. La única mejora que se ha introducido para optimizar el código ha sido aprovechar los huecos de retardo que presentan las instrucciones de salto para ubicar los incrementos de los índices.

Una vez que se ha realizado la selección de la traza el compilador debe proceder a su compactación para generar código en el que se minimice el número de operaciones vacías en las instrucciones VLIW y se reduzcan los ciclos de ejecución en beneficio de las rutas de ejecución más probables. Para ello, el compilador efectúa desplazamientos de las operaciones siendo los saltos los que introducen los mayores impedimentos ya que constituyen puntos de entrada y de salida de la traza. Sin embargo, la reorganización de operaciones dentro de la traza no es una tarea trivial ya que hay que garantizar que se preserva la corrección del programa con independencia de cuál sea la ruta más probable de ejecución. En el ejemplo de la Figura 3.12, por muy probable que sea la ruta decidida por el compilador, siempre se puede presentar la otra alternativa, `A[i]<>0`, por lo que será necesario tenerlo en cuenta al reorganizar el código. Antes de proseguir y para facilitar la posterior comparación del código sin planificar de la Figura 3.12 con el planificado hay que analizar el comportamiento de la ejecución del código original en las dos situaciones posibles, es decir, contando los ciclos de ejecución de ambas ramas de la sentencia `if`. En el listado de la Figura 3.12.b se aprecia que la ejecución de la rama supuestamente más probable del `if` (caso `A[i]==0`) conlleva la ejecución de las instrucciones 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16, 17, mientras que la menos probable, (caso `A[i]<>0`) provocaría la ejecución de las instrucciones 1, 2, 3, 4, 10, 11, 12, 13, 14, 15, 16, 17. En resumen, la ejecución de una iteración del bucle consumiría siempre 12 ciclos con



(a)

	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras y Saltos	
inicio:	LD R5, 0(R1)	-------	-------	1
	-------	-------	-------	2
	-------	-------	BNEZ R5, else	3
then:	LD F4, 0(R2)	-------	SUBI R1, R1, #4	4
	-------	-------	-------	5
	-------	ADDD F4, F4, F2	-------	6
	-------	-------	-------	7
	-------	-------	-------	8
	SD 0(R2), R4	-------	JMP final	9
else:	LD F4, 0(R3)	-------	SUBI R2, R2, #8	10
	-------	-------	-------	11
	-------	SUBD F4, F4, F2	-------	12
	-------	-------	-------	13
	-------	-------	-------	14
	SD 0(R3), F4	-------	-------	15
final:	-------	-------	BNEZ R1, inicio	16
	-------	-------	SUBI R3, R3, #8	17

(b)

Figura 3.12: Ejemplo de diagrama de flujo de control en el que se muestran las dos posibles secuencias de ejecución (a) y el código VLIW generado (b).

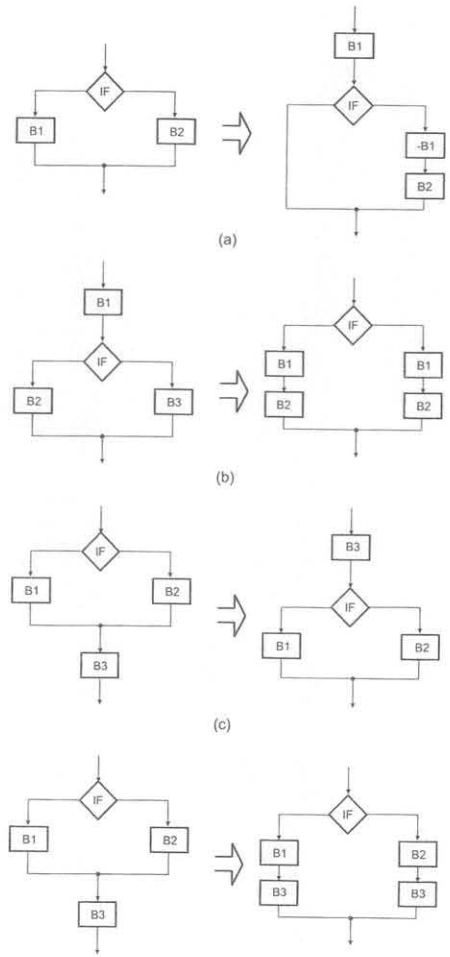


Figura 3.13: Posibles situaciones en las que se pueden plantear desplazamientos de operaciones entre bloques.

independencia de cuál fuese la rama del `if` que tuviese mayor probabilidad de ser ejecutada.

La Figura 3.13 presenta algunos de los posibles desplazamientos de operaciones que se pueden realizar dentro de una traza. La situación correspondiente a la Figura 3.13.a implica que las operaciones representadas en el bloque B1, que se encuentra en la secuencia de ejecución más probable, se podrían desplazar antes de la condición lo que obligaría a anular sus efectos mediante un código de compensación en caso de que no fuese necesaria su ejecución. Esto se ha representado en la figura mediante el bloque etiquetado como –B1 que indica que es necesario realizar las operaciones que anulen el efecto de la ejecución del contenido de B1. Los otros tres casos recogidos en la Figura 3.13 corresponden a situaciones en las que un bloque que se encuentra en un punto de ejecución común puede desplazarse con el objetivo de producir código más compacto. Las consideraciones que debe realizar el compilador para desplazar operaciones dentro de una traza son las siguientes:

- Conocer cuál es la secuencia de ejecución más probable.
- Conocer las dependencias de datos existentes para garantizar su mantenimiento. Evidentemente, cualquier desplazamiento de operaciones no debe provocar la violación de ninguna dependencia de datos y si se da el caso sería necesario contemplarlo en el código de compensación.
- La cantidad de código de compensación que es necesario añadir.
- Saber si compensa el desplazamiento de operaciones dentro de la traza, midiéndose el coste tanto en ciclos de ejecución como en espacio de almacenamiento.

En el ejemplo de la Figura 3.12 se puede apreciar que es posible aplicar el desplazamiento de operaciones correspondiente al bloque B1 de la Figura 3.13.a. En este ejemplo, el desplazamiento correspondería a las operaciones relacionadas con la expresión `X[i]:=X[i]+a` por considerarse la ruta de ejecución más probable. Por lo tanto, el código de compensación necesario, bloque –B1, sería la expresión contraria, es decir, `X[i]:=X[i]-a`. Aplicando esta transformación al código original, el nuevo código intermedio tendría el siguiente aspecto:

```
inicio: LD    F4,0(R2)     % Desplazamiento: Cargar X[i]
        ADDD  F4,F4,F2     % Desplazamiento: X[i]:=X[i]+a
        SD    0(R2),F4     % Desplazamiento: Almacenar X[i]
        LD    R5,0(R1)     % Cargar A[i]
        BNEZ  R5,else      % Si (A[i]<>0) ir a else
 then:  JMP   final
 else:  LD    F4,0(R2)     % Compensación: Cargar X[i]
        SUBD  F4,F4,F2     % Compensación: X[i]:=X[i]-a
        SD    0(R2),F4     % Compensación: Almacenar X[i]
        LD    F4,0(R3)     % Cargar Y[i]
        SUBD  F4,F4,F2     % Y[i]:=Y[i]-a
        SD    0(R3),F4     % Almacenar Y[i]
final:  SUBI  R2,R2,#8     % Decrementar índice de X
        SUBI  R3,R3,#8     % Decrementar índice de Y
```



```
        SUBI  R1,R1,#4     % Decrementar índice de A
        BNEZ  R1,inicio    % Nueva iteración
```

En la Figura 3.14 se muestra el nuevo código VLIW generado a partir del código intermedio ya planificado. Al igual que en el código VLIW original, se han aprovechado los huecos de retardo que presentan las instrucciones de salto para ubicar las instrucciones de incremento de los índices.

	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras y Saltos	
inicio:	LD F4, 0(R2)	------	------	1
	LD R5, 0(R1)	------	------	2
	------	ADDD F4, F4, F2	------	3
	------	------	------	4
	------	------	BNEZ R5, else	5
	SD 0(R2), F4	------	SUBI R1, R1, #4	6
then:	------	------	JMP final	7
else:	LD F4, 0(R2)	------	SUBI R2, R2, #8	8
	LD F4, 0(R3)	------	------	9
	------	SUBD F4, F4, F2	------	10
	------	SUBD F4, F4, F2	------	11
	------	------	------	12
	SD 8(R2), F4	------	------	13
	SD 0(R3), F4	------	------	14
final:	------	------	BNEZ R1, inicio	15
	------	------	SUBI R3, R3, #8	16

Figura 3.14: Código VLIW generado tras la planificación de traza.

La secuencia de ejecución de las instrucciones del código planificado considerando la ruta de ejecución más probable ha pasado a ser 1, 2, 3, 4, 5, 6, 7, 8, 15, 16, lo que implica un consumo de 10 ciclos y un decremento de 2 ciclos con respecto a la secuencia sin planificar. Si se recorre la ruta menos probable, las instrucciones ejecutadas pasan a ser 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, que equivale a un consumo de 15 ciclos y un incremento de 3 ciclos si se compara con la ejecución de la ruta menos probable en la secuencia no planificada. Aunque la reducción no es muy elevada, el tiempo medio de ejecución del código planificado será mejor que el código original siempre que se cumpla la siguiente expresión $[10\,\text{ciclos} * p + 15\,\text{ciclos} * (1-p) < 12\,\text{ciclos} * p + 12\,\text{ciclos} * (1-p)]$, donde p es la probabilidad de que la rama (`A[i]==0`) sea la ejecutada. En este ejemplo, siempre que $p > 0,6$ la secuencia de código planificado ya consumiría un número menor de ciclos. Aunque se ha recurrido a un sencillo ejemplo académico en el que el ahorro en ciclos no es sustancial dado el reducido número de operaciones que se realiza, en él se aprecia con claridad uno de los problemas que presenta la

planificación de trazas: a mayor cantidad de operaciones desplazadas, mayor es la penalización en que se incurre cuando se realiza una predicción errónea. En el ejemplo previo, la penalización por predicción errónea que hay que pagar es de 3 ciclos ya que se ha pasado de consumir 12 ciclos a 15 en la rama considerada como menos probable.

	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras y Saltos	
inicio:	LD F4, 0(R2)	------	------	1
	LD R5, 0(R1)	------	------	2
	LD F6, 0(R3)	ADDD F4, F4, F2	------	3
	------	------	BNEZ R5, else	4
	------	SUBD F6, F6, F2	SUBI R1, R1, #4	5
then:	SD 0(R2), F4	------	JMP final	6
else:	------	------	SUBI R2, R2, #8	7
	------	------	------	8
	SD 0(R3), F6	------	------	9
final:	------	------	BNEZ R1, inicio	10
	------	------	SUBI R3, R3, #8	11

Figura 3.15: Código VLIW con planificación óptima.

En el ejemplo que se ha tratado se pueden incluir diferentes optimizaciones encaminadas a reducir el número de instrucciones, generando un código más compacto. Así, el compilador podría no incluir el código de compensación sino evitar almacenar el resultado de la operación adelantada, `X[i]:=X[i]+a`, si no se cumple la predicción, decidir adelantar todas las cargas recurriendo al renombrado de registros, o ubicar la operación `Y[i]:=Y[i]-a` en algún hueco de retardo de salto. El resultado de estas optimizaciones se presenta en la Figura 3.15. Ahora, las instrucciones VLIW ejecutadas por cada rama del `if` son 9 (instrucciones 1, 2, 3, 4, 5, 6, 7, 10, 11) y 10 ciclos (instrucciones 1, 2, 3, 4, 5, 7, 8, 9, 10, 11) y el tamaño del código ha pasado de 16 a 11 instrucciones.

Sin embargo, muchas de las optimizaciones que se pueden aplicar de forma manual no se realizan de forma automática por el compilador ya que requieren técnicas muy complejas de implementar y que pueden producir grandes penalizaciones si la traza seleccionada no constituye la ruta más frecuente. Lo que sí resulta evidente es que cuando el tamaño de los bloques que se pueden desplazar es mayor, mayor es la capacidad que tiene el compilador para combinar operaciones básicas en instrucciones VLIW y obtener un código más compacto. Aunque la penalización por predicción errónea en que se incurra sea grande, debido a que el compilador introduce código de compensación sin optimización alguna, si la traza seleccionada es correcta, las mejoras son notables.

3.9. Operaciones con predicado

Como se ha podido apreciar, las técnicas de planificación local y global funcionan pero son difíciles de aplicar por parte del compilador, especialmente debido a las instrucciones de salto condicional, que rompen la ejecución secuencial del programa con puntos de entrada y salida. Con el objetivo de poder tratar las instrucciones condicionales como cualquier otra instrucción se idearon las *operaciones con predicado* (*predicated operations*), también conocidas como *operaciones condicionadas* (*conditioned operations*) u *operaciones con guarda* (*guarded operations*). La utilización de operaciones con predicado permite al compilador reducir el número de saltos condicionales que hay en el código de forma que un diagrama de flujo de control compuesto por diferentes ramas con pequeños bloques básicos pueda transformarse en un único bloque básico extendido y aplicarle técnicas de planificación local. Al proceso de eliminar los saltos condicionales de un programa y reemplazarlos por instrucciones con predicados se conoce habitualmente en la literatura con el término *if-conversion*.

Una operación con predicado no es más que una instrucción en la que su resultado se almacena o se descarta dependiendo del valor de un operando que tiene asociado. Este operando recibe el nombre de *predicado* y se implementa como un registro de un bit que se añade como un nuevo operando de lectura en cada una de las operaciones que conforman una instrucción VLIW. Según el valor de este registro de 1 bit, cada una de las operaciones que forman la instrucción VLIW almacenará su resultado (escritura en memoria o escritura en registro) o lo abandonará. Por lo tanto, es la condición que determina el que una operación tenga efecto o no sobre el estado del procesador y de la memoria.

Una representación habitual para indicar que una instrucción de código intermedio tiene asociado un predicado es

Instrucción (p)

donde si p vale 1 indica que el resultado de la operación que realice la instrucción se almacenará y lo contrario en caso de que valga 0. Por ejemplo,

```
ADDD  F4,F2,F0 (p1)
```

significa que el resultado de sumar los registros `F2` y `F0` solo se almacenará en `F4` si el predicado `p1` es igual a 1. Si el predicado se aplica a una instrucción de carga

```
LD  R1,100(R2) (p2)
```

quiere decir que el contenido de la posición de memoria `M[100+R2]` solo se almacenará en el registro `R1` si `p2` vale 1. Un formato para las instrucciones de manipulación de predicados podría ser el siguiente:

```
PRED_CLEAR   p1,p2              % p1:= false
                                % p2:= false
PRED_EQ      p1,p2,reg,valor    % p1:=(reg==valor)
                                % p2:=NOT(p1)
PRED_NE      p1,p2,reg,valor    % p1:=(reg<>valor)
                                % p2:=NOT(p1)
```

```
PRED_LT     p1,p2,reg,valor     % p1:=(reg<=valor)
                                % p2:=NOT(p1)
PRED_GT     p1,p2,reg,valor     % p1:=(reg>=valor)
                                % p2:=NOT(p1)
```

donde la utilización del segundo predicado, p2, es opcional. Si se utilizan los dos predicados en las instrucciones de manipulación, nunca ambos predicados podrán tener el valor true/true, solo está permitido true/false, false/true y false/false. La opción false/false está contemplada ya que es útil en estructuras condicionales tipo *if-then-else* anidadas en donde instrucciones de ambas ramas puede que no tengan que ser ejecutadas. Es posible incluso asignar un predicado a las operaciones de manipulación de predicados de forma que se ejecuten en función de una condición previamente establecida. Por ejemplo,

```
PRED_EQ  p1,p2,R1,#100 (p3)
```

indica que la instrucción solo se ejecutará si el predicado p3 es 1 y, en caso de que sea así, p1 y p2 se asignarán a 1 y 0, respectivamente, si el contenido de R1 es igual a 100 y a 0 y 1 en caso contrario. Ya que un predicado es un operando origen o destino, está sujeto a las dependencias de datos al igual que sucede con los operandos fuentes y destino de cualquier instrucción.

La Figura 3.16 muestra un ejemplo de aplicación de operaciones con predicado a un bucle en cuyo interior hay una estructura *if-then*. La Figura 3.16.b corresponde al código intermedio correspondiente al bucle genérico y la Figura 3.16.c presenta el código con instrucciones condicionadas. Mientras que el código intermedio consta de cuatro bloques básicos, el código intermedio condicionado está formado por un solo bloque. La Figura 3.17 presenta la transformación de ambos fragmentos de código intermedio en instrucciones VLIW considerando un procesador con las mismas características que en los ejemplos anteriores (una unidad funcional para operaciones enteras con 1 ciclo de latencia, una unidad para operaciones en coma flotante con 3 ciclos de latencia, una unidad para cargas/almacenamientos que consume 2 ciclos, los saltos se ejecutan en la unidad entera y tienen un hueco de retardo) y ubicando las

```
for (i = 0; i<100; i++)
    if (A[i] == 50) then
        j = j+2;
    else
        j = j+1;
    end if;
end for;
```

(a)

```
inicio: LD    R1, 0(R2)
        SUBI  R3, R1, #50
        BNEZ  R3, else
then:   ADDI  R5, R5, #2
        JMP   final
else:   ADDI  R5, R5, #1
final:  SUBI  R2, R2, #4
        BNEZ  R2, inicio
```

(b)

```
inicio: LD       R1, 0(R2)
        PRED_EQ  P1, P2, R1, #50
        ADDI     R5, R5, #2  (P1)   // then
        ADDI     R5, R5, #1  (P2)   // else
        SUBI     R2, R2, #4
        BNEZ     R2, inicio
```

(c)

Figura 3.16: Código fuente de un bucle genérico (a). Código intermedio (b). Código intermedio aplicando operaciones con predicado (c).

instrucciones para manipulación de predicados en la posición asignada a saltos y operaciones enteras. Mientras que el código VLIW sin operaciones condicionadas consume 11 y 9 ciclos según sea la rama de la estructura *if-then* que se ejecute, el código VLIW con operaciones condicionadas consume únicamente 8 ciclos con independencia de la resolución del *if-then*. En ambos casos, no se ha aplicado ninguna optimización al código intentando aprovechar los huecos de retardo de los saltos.

La aplicación de la técnica *if-conversion* a un conjunto de bloques básicos que contienen varias rutas de ejecución condicionales tiene por objetivo generar un único bloque con instrucciones condicionadas. Un ejemplo de aplicación de esta técnica a una diagrama de flujo de control perteneciente a un bucle con

	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras y Saltos	
inicio:	LD R1, 0(R2)	------	------	1
	------	------	------	2
	------	------	SUBI R3, R1, #50	3
	------	------	BNEZ R3, else	4
	------	------	------	5
then:	------	------	ADDI R5, R5, #2	6
	------	------	JMP final	7
	------	------	------	8
else:	------	------	ADDI R5, R5, #1	9
final:	------	------	SUBI R2, R2, #4	10
	------	------	BNEZ R2, inicio	11
	------	------	------	12

(a)

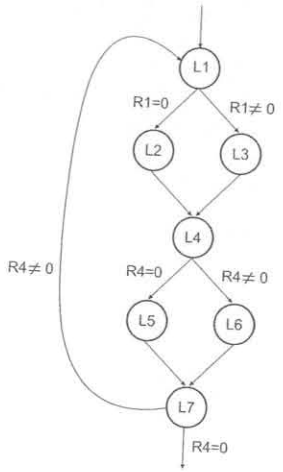
inicio:	LD R1, 0(R2)	------	------	1
	------	------	------	2
	------	------	PRED_EQ P1, P2, R1, #50	3
	------	------	ADDI R5, R5, #2 (P1)	4
	------	------	ADDI R5, R5, #1 (P2)	5
	------	------	SUBI R2, R2, #4	6
	------	------	BNEZ R2, inicio	7
	------	------	------	8

(b)

Figura 3.17: Instrucciones VLIW derivadas del código intermedio (a) e instrucciones VLIW derivadas del código intermedio con operaciones condicionadas (b).

varias estructuras *if-then* en su interior se presenta en la Figura 3.18. En este ejemplo se puede apreciar con total claridad cómo la utilización de instrucciones con predicado permite agrupar varios bloques básicos en un único bloque formado por un mayor número de instrucciones.

Las operaciones con predicado son adecuadas para eliminar saltos condicionales difíciles de predecir, especialmente aquellos que no forman parte de un bucle como las estructuras de tipo *if-then*. Pero, hay que tener en cuenta que este tipo de operaciones condicionadas no se expulsan cuando entran en la segmentación si no se cumple su condición de guarda. Las operaciones condicionadas siempre se ejecutan, pero si su predicado sigue sin cumplirse al concluir su procesamiento, el resultado se deshecha, es decir, pasan a ser equivalentes a instrucciones NOP. Por lo tanto, una instrucción con predicado es una forma de especulación y un uso excesivo de este tipo de operaciones puede producir una degradación en el rendimiento del procesador ya que consumen recursos pero puede que no cambien el estado de la máquina. Si todas las rutas de ejecución que hay en una región de código tienen aproximadamente el mismo tamaño en número de instrucciones y la misma frecuencia de ejecución, la eliminación de las estructuras *if-then* mediante instrucciones condicionadas es una técnica muy efectiva. Sin embargo, si las diferentes rutas de ejecución presentan grandes diferencias en cuanto a tamaño y frecuencia,



(a)

```
L1: LD    R1, 0(R5)
    BNEZ  R1, L3
L2: ADDI  R2, R1, #1
    LD    R3, 0(R6)
    LD    R4, 0(R7)
    ADD   R4, R4, R2
    JMP   L4
L3: ADDI  R4, R4, #1
    ADDI  R2, R1, #1
L4: BNEZ  R4, L6
L5: ADDI  R4, R4, #1
    SD    0(R4), R2
    JMP   L7
L6: SUBI  R4, R4, #1
    SD    0(R4), R2
L7: BNEZ  R4, L1
```

(b)

```
L1: LD      R1, 0(R5)
    PRED_NE P1, P2, R1, #0
    ADDI    R2, R1, #1  (P2)
    LD      R3, 0(R6)   (P2)
    LD      R4, 0(R7)   (P2)
    ADD     R4, R4, R2  (P2)
    ADDI    R4, R4, #1  (P1)
    ADDI    R2, R1, #1  (P1)
    PRED_NE P3, P4, R4, #0
    ADDI    R4, R4, #1  (P4)
    SD      0(R4), R2   (P4)
    SUBI    R4, R4, #1  (P3)
    SD      0(R4), R2   (P3)
    BNEZ    R4, L1
```

(c)

Figura 3.18: Ejemplo de agrupamiento de bloques básicos mediante la transformación de estructuras *if-then* en operaciones con predicado.

el rendimiento que se obtiene se reduce ya que todas las instrucciones condicionadas tienen que ser procesadas aunque solo el predicado asociado a una ruta concreta sea válido. Por otra parte, la aplicación de predicados a las instrucciones implica transformar dependencias de control en dependencias de datos por lo que resulta clave el tener en cuenta las nuevas dependencias de datos RAW que pueden surgir a la hora de desplazar las instrucciones.

3.10. Tratamiento de excepciones

Como se ha podido apreciar, muchas de las técnicas aplicadas para aumentar el paralelismo a nivel de instrucción se basan en el desplazamiento de operaciones con el fin de generar un código VLIW más compacto y que aproveche mejor los recursos del procesador. Parte de estos desplazamientos de código se realizan en base a predicciones de los resultados de los saltos condicionales de forma que cuando la predicción falla hay que ejecutar un cierto código de compensación para deshacer los efectos del bloque de operaciones ejecutadas especulativamente. Todo ello obliga a habilitar mecanismos para el tratamiento de las excepciones de forma que se garantice la consistencia del procesador y de la memoria frente a la ejecución especulativa de operaciones, de forma análoga a como se realiza en los procesadores superescalares.

Una de las estrategias aplicadas en los procesadores VLIW es la utilización de *centinelas*, que es una técnica análoga al buffer de reordenamiento de los procesadores superescalares. El uso de centinelas consiste en que el compilador marca las operaciones especulativas con una etiqueta y en el lugar del programa en el que estaba el código especulado que ha sido desplazado sitúa un centinela vinculado a esa etiqueta. El centinela no es más que un fragmento de código que indica que la operación ejecutada de forma especulativa con la que está relacionado ha dejado de serlo. Cuando una operación marcada como especulativa se ejecuta, su resultado se almacena o marca como temporal. La ejecución del código de centinela es lo que determina que los resultados temporales se pueden escribir en sus ubicaciones definitivas: fichero de registro o memoria. Al estar el código del centinela en la posición del código especulado, la ejecución del centinela indica que las operaciones ya no son especulativas.

Esta estrategia se implementa mediante una especie de buffer de terminación en el que las instrucciones se retiran cuando les corresponde salvo las marcadas como especulativas, que se retirarán cuando lo señale la ejecución del centinela que tienen asociado. De esta forma, la ejecución especulativa de operaciones no tendrá impacto en el estado del procesador ni de la memoria, por lo que si una instrucción posterior lanza una excepción no habrá que deshacer el resultado de ninguna operación especulada sino eliminarla del buffer.

3.11. El enfoque EPIC

El aumento del nivel de ILP mediante un enfoque VLIW puro presenta una serie de problemas que constituyen un impedimento para que procesadores de propósito general basados en esta concepción

arquitectónica representen una opción alternativa a los actuales procesadores superescalares. Algunos de estos problemas son:

- Los repertorios de instrucciones VLIW no son compatibles entre diferentes implementaciones. Nuevos procesadores VLIW con un mayor número de unidades funcionales implican nuevos repertorios de instrucciones VLIW más anchas, con más operaciones o con una disposición distinta en la instrucción. Por otra parte, código VLIW generado para un procesador determinado puede no funcionar en otro procesador con las mismas unidades funcionales pero latencias diferentes debido a la planificación realizada por el compilador.
- Las instrucciones de carga presentan comportamientos no determinísticos por la propia naturaleza de sistema de memoria. Ello implica que la planificación estática de las instrucciones de carga por parte del compilador resulte muy complicada.
- La importancia de disponer de un compilador que garantice una planificación óptima del código de forma que se maximice el rendimiento del computador y se minimice el tamaño del código. La abundancia de operaciones vacías en las instrucciones VLIW implica un rendimiento inadecuado de la I-caché y un aumento del tráfico de los buses en movimientos inútiles.

Para superar tales inconvenientes surgió el estilo de arquitectura EPIC (*Explicitly Parallel Instruction Computing*), concepto acuñado por primera vez en 1997 por la alianza Intel-HP aunque el proyecto de investigación en que se sustenta se remonta a 1989 y es originario de HP. EPIC constituye una evolución del enfoque VLIW en el que se han absorbido varios conceptos del ámbito de los procesadores superescalares. Aunque muchas veces se asocia exclusivamente el concepto EPIC a la familia de procesadores Intel Itanium, EPIC representa una filosofía para construir procesadores que se apoya en ciertas características arquitectónicas. Al igual que los conceptos RISC o VLIW aglutinan conjuntos de características que dan lugar a diferentes arquitecturas, es posible tener diferentes arquitecturas basadas en la filosofía EPIC. Dependiendo de qué características del EPIC se usen o potencien, un procesador EPIC estará optimizado para dominios diferentes: propósito general o mercado embebido. Actualmente, los únicos procesadores comerciales basados en el concepto EPIC son los Itanium, Itanium 2 e Itanium 9300 que corresponden a sucesivas generaciones de procesadores derivados de la arquitectura IA-64 o Intel Itanium, desarrollada por Intel en colaboración con HP.

El principal objetivo de EPIC es retener la planificación estática del código pero mejorarla con características arquitectónicas que permitan hacer frente dinámicamente a diferentes situaciones, tales como retardos en las cargas o unidades funcionales nuevas o con diferentes latencias. La Tabla 3.1 representa de forma resumida las diferencias entre un procesador superescalar, un procesador VLIW y un procesador EPIC.

En un procesador superescalar todo lo relativo a analizar las dependencias entre instrucciones, distribuirlas por tipo a las unidades funcionales y planificar su emisión según disponibilidad de operandos y unidad funcional se realiza mediante mecanismos hardware. En el extremo opuesto se sitúa el enfoque VLIW, donde todas las responsabilidades del hardware las asume el software, esto es, el compilador.

Tabla 3.1: Diferencias principales entre las arquitecturas superescalar, VLIW y EPIC.

	Agrupamiento de operaciones	Asignación de unidad funcional	Secuencia de emisión a las unidades funcionales
Superescalar	Hardware	Hardware	Hardware
EPIC	Compilador	Hardware	Hardware
VLIW	Compilador	Compilador	Compilador

En un enfoque VLIW es el compilador el encargado de realizar el agrupamiento de operaciones básicas en instrucciones, estando la operación asignada a la unidad funcional según la posición que ocupe en la instrucción y quedando preestablecido el instante de emisión por la posición que la instrucción ocupa en el código, dado que en cada ciclo se emite una instrucción. El enfoque EPIC se sitúa en un punto intermedio pues es el compilador el que determina al agrupamiento de instrucciones pero, a la vez, comunica de forma explícita en el propio código cómo se ha realizado el agrupamiento. De esta forma, EPIC posibilita la compatibilidad a través de diferentes implementaciones sin necesidad de recurrir a los complicados mecanismos hardware de los procesadores superescalares.

El siguiente ejemplo de código permitirá entender las diferencias con el enfoque VLIW. Considere el siguiente fragmento de código intermedio que corresponde a la operación `A:=B+C`:

```
LD    F2,0(R1)    % Carga de B en F2 desde M[0+R1]
LD    F4,0(R2)    % Carga de C en F4 desde M[0+R2]
ADDD  F6,F4,F2    % Suma en F6
SD    0(R3),F6    % Almacenamiento de A en M[0+R3]
```

y que dispone de un procesador VLIW con dos unidades de carga/almacenamiento (2 ciclos de latencia), una unidad funcional para operaciones en coma flotante (2 ciclos de latencia) y una unidad para operaciones enteras y saltos (1 ciclo de latencia). La Figura 3.19.a muestra el fragmento de código que produciría el compilador VLIW y que, claramente, ilustra el problema de ocupación de los campos de operación en formatos de instrucciones muy largos: de las 20 operaciones, 16 corresponden a NOPS y solo 4 contienen operaciones útiles. Por otra parte, el código generado no sería válido para otro procesador VLIW con las mismas latencias pero una única unidad de carga/almacenamiento. Sin embargo, el código intermedio original sí podría ejecutarse en cualquier procesador superescalar, con independencia del número y tipo de las unidades funcionales que tuviese. La Figura 3.19.b es un ejemplo de aplicación del concepto EPIC en el que se incluye información explícita del número de operaciones que se pueden emitir en paralelo dada su independencia; mediante la información contenida entre los corchetes se puede saber cuántas operaciones consecutivas se pueden emitir en paralelo sin necesidad de conocer las características del procesador en el momento de compilar. En el ejemplo de la Figura 3.19.b, el valor `[2]` ubicado a la izquierda de la primera instrucción de carga indica que se puede emitir en paralelo con ella la siguiente instrucción que aparezca en el código; las siguientes instrucciones llevan

Operaciones Carga / Almacenamiento	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras
LD F2, 0(R1)	LD F4, 0(R2)	------	------
------	------	------	------
------	------	ADDD F6, F4, F2	------
------	------	------	------
SD 0(R3), F6	------	------	------

(a)

```
[2] LD    F2, 0(R1)
[1] LD    F4, 0(R2)
[1] ADDD  F6, F4, F2
[1] SD    0(R3), F6
```

(b)

Figura 3.19: Ejemplo de código VLIW (a) y de aplicación de información explícita sobre el agrupamiento de instrucciones independientes (b).

asociado el valor `[1]` que explicita que la instrucción no puede emitirse en paralelo con ninguna de las instrucciones que la preceden.

A continuación, se enumeran las características arquitectónicas más destacadas del estilo EPIC:

- Planificación estática con paralelismo explícito.
- Operaciones con predicado.
- Descomposición o factorización de las instrucciones de salto condicional.
- Especulación de control.
- Especulación de datos.
- Control de la jerarquía de memoria.

3.12. Procesadores vectoriales

Los procesadores vectoriales son máquinas diseñadas en base a una arquitectura que permite la manipulación de vectores (arrays unidimensionales). Este tipo de arquitecturas se conoce como arquitecturas vectoriales, en contraposición a las arquitecturas escalares en que se basan los procesadores clásicos, los segmentados, los superescalares o los VLIWs. Para poder manipular vectores, los

procesadores vectoriales cuentan con un repertorio de instrucciones en el que los operadores fuente y destino de las instrucciones no son valores escalares almacenados en los tradicionales registros sino que son vectores almacenados en unos registros especiales denominados registros vectoriales. Típicas operaciones vectoriales son la suma, resta, multiplicación y división de dos vectores, la suma o multiplicación de todos los elementos de un vector por un escalar, etc. Por lo tanto, en un procesador vectorial existen dos tipos de operandos, los operandos vectoriales y los operandos escalares, y las operaciones vectoriales pueden tener como fuente dos operandos vectoriales o uno vectorial y otro escalar pero el resultado siempre será un operando vectorial.

Una ventaja de este tipo de procesadores es que pueden utilizar unidades funcionales vectoriales con segmentaciones muy profundas sin tener que preocuparse por la existencia de dependencias de datos. Ello se debe a que una operación vectorial implica procesar de forma independiente pero continua cada uno de los elementos de que constan los operandos fuente vectoriales: para sumar dos vectores `A` y `B` de 64 elementos, la suma del elemento `A[i]` al `B[i]` no implica ningún tipo de dependencia con la operación previa, es decir, la suma del elemento `A[i-1]` al `B[i-1]`, o con la suma del `A[i-2]` al `B[i-2]`, etc. Al igual que sucede en el enfoque VLIW, el encargado de generar código formado por instrucciones vectoriales a partir del análisis de las operaciones que se realizan en el código original es el compilador.

Otra ventaja de los procesadores vectoriales es que una única instrucción vectorial equivale a un bucle completo de instrucciones escalares. Un ejemplo permite ilustrar esta afirmación. En la tabla ubicada a continuación, en la parte izquierda se indican las acciones que habría que realizar en un procesador escalar para, por ejemplo, programar un bucle con el que sumar los elementos de dos vectores almacenados en memoria; en la parte derecha se presenta la equivalencia en un procesador vectorial.

1. Inicialización de los índices.		
2. Cálculo escalar (por ejemplo, una suma).		
3. Actualización de los índices.	$\equiv$	1. Cálculo vectorial.
4. Comparación.		
5. Salto a la instrucción 2.		

El que una única instrucción vectorial equivalga a un bucle escalar completo tiene dos consecuencias inmediatas. Por un lado, el ancho de banda de instrucciones es menor ya que el tamaño del código es más reducido y no es necesario extraer tantas instrucciones de la I-caché y, por otro lado, no existen riesgos de control al eliminarse las instrucciones de salto condicional que conlleva todo bucle.

El que los procesadores vectoriales fuesen diseñados pensando en el ámbito de las aplicaciones científicas les añade una ventaja adicional para manipular vectores. Las aplicaciones científicas e ingenieriles, a menudo, manipulan grandes volúmenes de datos con una baja latencia temporal y espacial lo que provoca que un sistema de jerarquía tradicional con memoria caché no sea muy eficaz. Pero, además, estos grandes volúmenes de datos, que se organizan de forma lógica en vectores y matrices, se ubican en memoria principal siguiendo unos patrones conocidos (el más habitual es ubicar los elementos consecutivamente, fila tras fila, o columna tras columna) con lo que es posible amortizar el coste de acceso directo a la memoria principal distribuyendo el espacio de direccionamiento entre varios bancos

de memoria. De esta forma, el elevado coste que hay que pagar por acceder directamente a la memoria principal en comparación con el que tiene la D-caché se paga una única vez para todos los elementos de un vector en lugar de tener que pagar el coste de acceso a la D-caché por cada elemento.

Al igual que sucede con los procesadores VLIW, el compilador es el encargado de detectar y extraer el paralelismo intrínseco que existe entre las instrucciones que componen el código fuente. Por lo tanto, el éxito con que un programa puede aprovechar las características de un procesador vectorial viene determinado por la estructura del flujo de datos y de control del programa y por la capacidad del compilador. Según el método aplicado para detectar bucles escalares que puedan ser vectorizados y, una vez detectados, según el algoritmo utilizado para vectorizar el bucle, los resultados obtenidos serán muy diferentes. Sin embargo, la capacidad de este tipo de procesadores para manipular vectores es inigualable por lo que su uso se ha desplazado desde los grandes centros de supercomputación a los computadores de uso genérico mediante extensiones vectoriales añadidas a los procesadores superescalares.

3.13. Arquitectura vectorial básica

Al igual que los procesadores escalares, los procesadores vectoriales pueden estar basados en una arquitectura de carga-almacenamiento, con todos los operandos ubicados en los registros vectoriales, o en una arquitectura memoria-memoria, con los operandos almacenados en la memoria. Dado que todos los supercomputadores vectoriales actuales así como las extensiones vectoriales de los procesadores superescalares son del tipo carga-almacenamiento con repertorios de instrucciones tipo RISC, el contenido de este tema se centrará en ellos. Queda como curiosidad que las primeras máquinas vectoriales, los computadores CDC-Star 100 y el Texas Instruments ASC, eran del tipo memoria-memoria.

Un procesador vectorial básico consta de una unidad de procesamiento escalar y de una unidad de procesamiento escalar (Figura 3.20). La razón de ello es que el código objeto de un programa vectorizado se compone de una combinación de instrucciones escalares e instrucciones vectoriales, en mayor o menor proporción en función de la capacidad de vectorización del compilador. La unidad de procesamiento escalar se ocupa de atender las instrucciones escalares y las unidad vectorial las instrucciones vectoriales. Curiosamente, si en los grandes supercomputadores vectoriales la unidad de procesamiento escalar se consideraba como el hermano menor, en los actuales procesadores superescalares, la unidad vectorial se considera como un añadido a la unidad escalar que es la que constituye el núcleo fuerte de procesamiento.

Las etapas IF e ID del esquema de la Figura 3.20 podrían organizarse mediante mecanismos similares a los ya descritos en los dos primeros capítulos pero con la diferencia de que, según el código de operación de la instrucción indique si se trata de una operación escalar o vectorial, se activarían unas unidades funcionales u otras. La unidad de procesamiento escalar está compuesta por el fichero de registros escalares y por las unidades funcionales en las que se ejecutan las instrucciones escalares. Su organización y funcionamiento es similar al de cualquier procesador escalar, es decir, dispondrá de unidades segmentadas para realizar operaciones enteras, en coma flotante, saltos, cargas/almacenamientos, etc. que trabajarán con valores del fichero de registros escalares o leyendo y

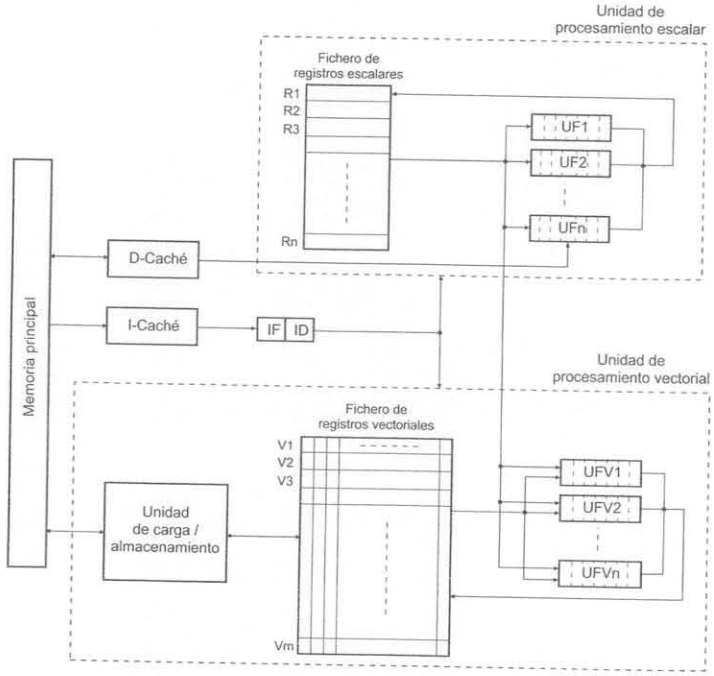


Figura 3.20: Esquema de un procesador vectorial básico.

escribiendo datos en memoria a través de la D-caché.

La unidad de procesamiento vectorial consta de un *fichero de registros vectoriales*, *unidades funcionales vectoriales* y una *unidad de carga/almacenamiento vectorial*. El fichero de registros vectoriales está compuesto por un conjunto de entradas en donde cada entrada almacena los elementos de que consta un vector. Unas especificaciones técnicas muy importantes en un procesador vectorial son las características del fichero de registros vectoriales ya que son fundamentales para los algoritmos de vectorización. Estos parámetros son el número de registros vectoriales, el número de elementos por registro y el tamaño en bytes de cada elemento. Al número de elementos por registro se le suele identificar por las siglas MVL (*Maximum Vector Length* - Máxima Longitud del Vector). El número de registros vectoriales oscila de los 64 a los 256, el número de elementos por registro de 8 a 256 y el

tamaño de cada elemento suele ser de 8 bytes para poder almacenar valores en coma flotante en doble precisión. Por lo tanto, la cantidad de información que puede llegar a contener un fichero de registros vectoriales es considerable en comparación con un fichero escalar. Por ejemplo, un fichero vectorial de 256 registros de 64 elementos de 8 bytes puede almacenar hasta 128 Kbytes. Algunos procesadores vectoriales, principalmente de fabricantes japoneses como Fujitsu o NEC, tienen ficheros de registros vectoriales de geometría variable, aumentando o disminuyendo el número de registros mediante una reducción o un aumento del número de elementos por registro. Otra característica a destacar de los ficheros de registros vectoriales es la necesidad de disponer de multitud de puertos de lectura y de escritura, como mínimo de dos puertos de lectura y de uno de escritura por registro. El motivo de ello es poder mantener el suministro continuo de datos a todas las unidades funcionales vectoriales cuando operan simultáneamente. La conexión de todos los puertos de lectura y de escritura con las unidades funcionales vectoriales se realiza mediante una red de interconexión total de tipo *crossbar* (ver Capítulo 4).

Al igual que las escalares, las unidades funcionales vectoriales están, por lo general, segmentadas y pueden iniciar una operación en cada ciclo de reloj. Las unidades vectoriales se alimentan con los elementos provenientes de los registros vectoriales pero también pueden recibir datos de los escalares para ejecutar operaciones de vectores con escalares (por ejemplo, sumar una constante a un vector). Aunque no se ha recogido en el esquema de la Figura 3.20, y al igual en que en los procesadores superescalares, son necesarios mecanismos hardware que detecten los riesgos estructurales y los riesgos de datos que pueden aparecer entre las instrucciones vectoriales y escalares que componen un programa. Al igual que sucede en los procesadores escalares, el número de unidades vectoriales varía según el modelo y fabricante del computador vectorial aunque siempre hay unidades para aritmética con enteros, sumas, multiplicaciones y divisiones en coma flotante, e incluso para calcular raíces cuadradas. Una característica que presentan las unidades funcionales vectoriales de algunos procesadores es que pueden estar internamente organizadas en varios *lanes* o carriles con el objeto de aumentar el número de operaciones que pueden procesar en paralelo por ciclo de reloj. Básicamente, una unidad funcional con n carriles implica que, internamente, el hardware necesario para realizar la operación se ha replicado n veces de forma que el número de ciclos que se emplea en procesar un vector se reduce por un factor de n. La Figura 3.21 muestra la diferencia existente al realizar la operación `C:=A+B`, donde `A`, `B` y `C` son registros vectoriales de 20 elementos, en una unidad funcional de 4 etapas con un único carril y en otra de similar profundidad de segmentación pero con 4 carriles. Si la unidad funcional con un único carril genera un resultado por ciclo y tarda 20 ciclos en generar el vector resultado, la unidad con 4 carriles produce 4 resultados por ciclo, reduciendo el tiempo de procesamiento del vector en un factor de 4.

Un problema que conlleva el empleo de unidades segmentadas organizadas internamente en varios carriles es que hay que incrementar el número de puertos de lectura y escritura de todos los registros vectoriales para poder suministrar en cada ciclo de reloj elementos a los carriles. En el caso de la Figura 3.21, solo para operar con la unidad de suma con cuatro carriles son necesarios cuatro puertos de lectura/escritura por registro vectorial. Una posible organización es especializar los carriles de forma que solo tengan acceso a un subconjunto de los elementos de cada registro vectorial. La Figura 3.22

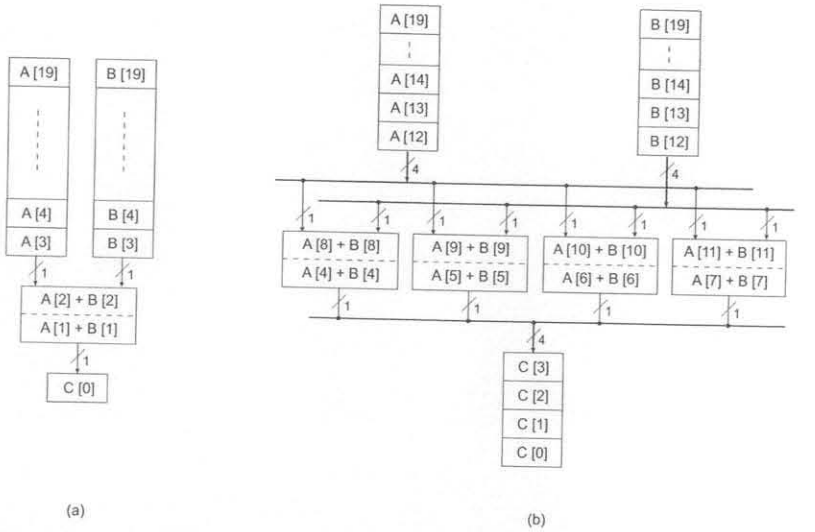


Figura 3.21: Diferencia entre realizar la suma de 2 vectores de 20 elementos en una unidad funcional con 2 etapas y un único carril (a) y en otra unidad funcional de 2 etapas pero con 4 carriles (b).

representa la organización interna de una unidad vectorial de suma y multiplicación con cuatro carriles en donde cada carril únicamente tiene acceso a la cuarta parte de los elementos de cada registro vectorial.

La unidad funcional de carga/almacenamiento vectorial se ocupa de cargar los registros vectoriales con datos extraídos de la memoria y de almacenar en ella el contenido de los mismos. Al igual que las restantes unidades funcionales también puede estar segmentada, pero su principal característica es que, tras la latencia de acceso inicial, es capaz de mantener un ancho de banda sostenido de una palabra por ciclo de reloj. El número de unidades funcionales de carga/almacenamiento de un procesador vectorial es más reducido que el de las aritméticas y suele ser de una o dos. La Tabla 3.2 recoge ciertas características de los procesadores vectoriales de supercomputadores comercializados durante las últimas décadas.

Dentro de la categoría de arquitecturas de computador basadas en aprovechar el paralelismo de los datos, existe otro tipo de procesador que, en algunas ocasiones, se califica erróneamente como vectorial: el *procesador matricial*. Este tipo de procesador junto con el vectorial son los dos únicos representantes que se engloban dentro de la categoría SIMD (*Single Instruction-Multiple Data*). Al igual que un procesador vectorial, un procesador matricial cuenta con una unidad de procesamiento vectorial, una unidad escalar y una unidad de control que discrimina según el tipo de instrucción.

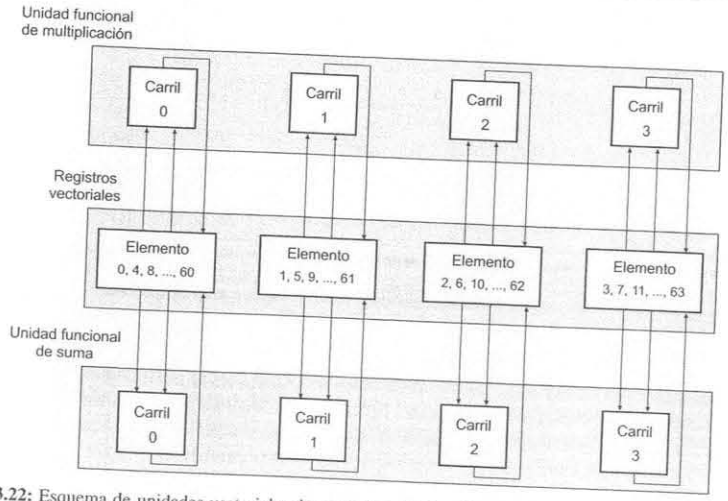


Figura 3.22: Esquema de unidades vectoriales de suma y multiplicación en coma flotante con cuatro carriles y distribución de los elementos de los registros vectoriales entre ellas.

La diferencia surge en la organización de la unidad vectorial. En un procesador vectorial, la unidad vectorial se compone de n unidades funcionales especializadas. En un procesador matricial, la unidad vectorial consta de n elementos de procesamiento o EPs, constituidos por una unidad aritmético-lógica de propósito general o ALU, un conjunto de registros o REP y una memoria local o MEP. Todos los EPs se encuentran comunicados a través de una red de interconexión y pueden funcionar de forma independiente o sincronizada. La principal diferencia con respecto a un procesador vectorial es que un procesador matricial con n elementos de procesamiento puede procesar de forma simultánea en el mismo ciclo de reloj un vector de n elementos.

En un procesador matricial, la distribución de los elementos de un vector entre las MEPs es el factor clave para aprovechar al máximo el paralelismo que brindan los EPs. En condiciones ideales, un procesador matricial con n EPs podría producir n resultados simultáneamente al realizar una operación vectorial y, en el peor de los casos, todos los resultados podrían llegar a generarse por un único EP (en este caso, la vectorización sería inexistente). Por lo tanto, un vector de m elementos que actúe como operando fuente de una operación vectorial podría estar distribuido entre todas las MEP si $m \leq n$ y, en caso contrario, habría que distribuir los m elementos del vector de forma cíclica entre las n MEPs para paralelizar la operación correctamente. En el ejemplo de la Figura 3.23, si la longitud de los vectores fuese superior a ocho habría que distribuirlos de forma cíclica entre las ocho MEPs con el fin de balancear

Tabla 3.2: Características de los procesadores vectoriales de supercomputadores comercializados durante las últimas décadas. Se muestran las características de un único procesador vectorial aunque el computador pueda estar formando por varias unidades.

Procesador vectorial	Año anuncio	Reloj (Mhz.)	Total de registros vectoriales	Elementos de 8 bytes por registros	Unidades funcionales	Lanes por U.F.	Unidades de c/a
Cray-1	1976	80	8	64	6	1	1
Cray-2	1985	244	8	64	5	1	1
Cray Y-MP	1988	166	8	64	8	1	2 c., 1 a.
Cray C-90	1991	240	8	128	8	2	4
Convex C-4	1994	135	16	128	3	1	4
Cray T-90	1996	460	8	128	8	2	4
NEC SX/5	1998	312	8-64	512	4	16	1
Fujitsu VPP500	1999	300	8-256	4096-128	3	16	1 c., 1 a.
Cray SV1	1998	300	8	64	8	1	1+1 a.
Cray SV1ex	2001	500	8	64	8-2	2-8	1+1 a.
VMIPS	2001	500	8	64	5	1	1
NEC SX/6	2001	500	72	32	5	1	1
Cray X1	2002	800	32	64	4	2	?
NEC SX/8	2004	2200	72	64	5	1	1
NEC SX/9	2008	3300	72	64	6	2	1

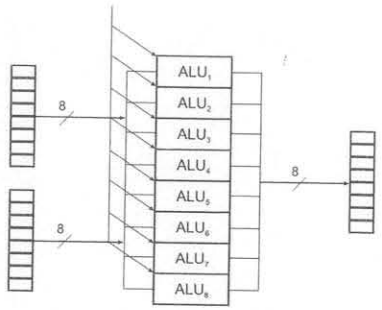


Figura 3.23: Ejemplo simplificado de la realización de una operación sobre dos vectores en un procesador matricial compuesto por ocho EPs.

correctamente la carga de trabajo de los EPs y aprovechar al máximo su paralelismo.

Hoy en día no se fabrican computadores comerciales basados en procesadores de este tipo. Computadores basados en una arquitectura matricial que alcanzaron cierta notoriedad fueron el Illiac IV y las supercomputadoras CM-1, CM-2 y CM-200. El Illiac IV comenzó a desarrollarse a mediados de los años 60 en la Universidad de Illinois y fue terminado por la NASA en el año 1974. Aunque originalmente fue diseñado para contener 256 EPs de 64 bits y alcanzar una velocidad teórica de 1 GFLOPS, el sistema final contó únicamente con 64 EPs y una velocidad teórica máxima de 200 MFLOPS. La serie de supercomputadores Connection Machine 1, 2 y 200 fueron fabricadas por la empresa Thinking Machines Corporation durante los años 80. Las CM-1, CM-2 y CM-200 originales podían configurarse con hasta 65.536 EPs de un bit aunque, posteriormente, se comercializó el modelo CM-2a con configuraciones más reducidas de 16.384 y 4.096 procesadores.

3.14. Repertorio genérico de instrucciones vectoriales

Aunque existen diferencias, el repertorio de instrucciones vectoriales aritmético-lógicas y de acceso a memoria es muy similar al de las operaciones escalares y por cada operación escalar suele existir su equivalente vectorial. Para distinguir en los párrafos siguientes a las instrucciones vectoriales de las escalares y evitar así confusiones, se concatenará el sufijo `V` a su código de operación y los registros vectoriales se denotarán como `Vi`. Aunque el repertorio de instrucciones varía mucho dependiendo del fabricante y del procesador, un subconjunto de instrucciones vectoriales que está presente de uno u otro modo en cualquier repertorio comercial se presenta a continuación.

`ADDV Vi,Vj,Vk`	Almacena en `Vi` el resultado de sumar los elementos de `Vj` y `Vk`.
`ADDSV Vi,Vj,Fi`	Almacena en `Vi` el resultado de sumar `Fi` a cada elemento de `Vj`.
`SUBV Vi,Vj,Vk`	Almacena en `Vi` el resultado de restar los elementos de `Vk` a los de `Vj`.
`SUBSV Vi,Vj,Fi`	Almacena en `Vi` el resultado de restar `Fi` a cada elemento de `Vj`.
`SUBSV Vi,Fi,Vj`	Almacena en `Vi` el resultado de restar cada elemento de `Vj` a `Fi`.
`MULTV Vi,Vj,Vk`	Almacena en `Vi` el resultado de multiplicar los elementos de `Vj` y `Vk`.
`MULTSV Vi,Vj,Fi`	Almacena en `Vi` el resultado de multiplicar `Fi` por cada elemento de `Vj`.
`DIVV Vi,Vj,Vk`	Almacena en `Vi` el resultado de dividir los elementos de `Vj` por los de `Vk`.
`DIVSV Vi,Vj,Fi`	Almacena en `Vi` el resultado de dividir los elementos de `Vj` por `Fi`.
`DIVSV Vi,Fi,Vj`	Almacena en `Vi` el resultado de dividir `Fi` por los elementos de `Vj`.
`LV Vi,Ri`	Carga en `Vi` los elementos ubicados en memoria a partir de la posición `M[Ri]`.
`SV Ri,Vi`	Almacena los elementos de `Vi` a partir de la posición de memoria `M[Ri]`.

Aunque con otros nombres y formatos, en cualquier repertorio de instrucciones vectoriales existen instrucciones para realizar distintos tipos de operaciones aritméticas entre escalares y vectores. Análogamente, hay instrucciones de acceso a memoria que permiten cargar y almacenar todos los elementos de un registro vectorial.

A continuación, se recurre al ejemplo del bucle DAXPY (*Double Precision A Times X Plus Y*) para

mostrar con mayor detalle las diferencias entre el código escalar y el vectorial utilizando para ello el repertorio genérico de instrucciones vectoriales. El código escalar que se muestra realiza la operación vectorial $Y(i) = a * X(i) + Y(i)$ con vectores de 64 elementos de 8 bytes:

```
        LD      F10,0(R5)      % Carga valor de a desde M[0+R5]
        ADDI    R3,R1,#512     % Cálculo posición del último elemento
bucle:  LD      F2,0(R1)       % Carga de X(i) desde M[0+R1]
        MULTD   F4,F2,F10      % a*X(i)
        LD      F6,0(R2)       % Carga de Y(i) desde M[0+R2]
        ADDD    F6,F4,F6       % Y(i):=a*X(i)+Y(i)
        SD      0(R2),F6       % Almacenamiento de Y(i) en M[0+R2]
        ADDI    R1,R1,#8       % Sumar 1 al índice de X
        ADDI    R2,R2,#8       % Sumar 1 al índice de Y
        SUB     R4,R3,R1       % Comparar R1 con posición del último
        BNZ     R4,bucle       % Si no último, repetir bucle
```

Dado que se considera que la longitud de los vectores es de 64 elementos, en la aproximación escalar se ejecutan 578 instrucciones ($2 + 9 * 64$). El código vectorial equivalente al escalar es:

```
        LD      F10,0(R5)      % Carga valor de a desde M[0+R5]
        LV      V1,R1          % Carga vector X desde M[R1]
        MULTSV  V2,V1,F10      % a*X
        LV      V3,R2          % Carga vector Y desde M[R2]
        ADDV    V4,V3,V2       % Y:=Y+a*X
        SV      R2,V4          % Almacenamiento vector Y en M[R2]
```

A primera vista se aprecia que el número de instrucciones ejecutadas es mucho menor, solo 6 instrucciones, que en comparación con las casi 600 del código escalar significan una reducción notable del ancho de banda dinámico. Además, el código vectorial no necesita instrucciones de sobrecarga como sucede en el bucle escalar con las instrucciones para incrementar los índices o para comprobar el final del bucle. Otra ventaja adicional es que el código vectorial sufre menos detenciones ya que solo se detiene una vez por operación y no una vez por cada elemento del vector, como sucede en el código escalar (si no se consideran técnicas de planificación como el desenrollado de bucles). En el código vectorial, la instrucción `MULTSV` espera una única vez por la finalización de la instrucción `LV`, la `ADDV` una vez por la finalización de `MULTSV`, y la `SV` una vez por la `ADDV`. En el código escalar, las esperas son similares pero se repiten para cada uno de los elementos del vector.

Tras analizar el código vectorial anterior y reflexionar sobre la vectorización de otros bucles surgen inmediatamente algunas cuestiones: ¿cómo proceder cuando la longitud del vector es diferente al número de elementos de un registro vectorial?, ¿qué sucede cuando los elementos del vector se almacenan de forma uniforme pero no consecutiva en memoria? o ¿cómo se vectoriza el cuerpo de un bucle que contiene instrucciones ejecutadas condicionalmente? Como se verá a continuación, dos registros escalares especiales tienen un papel determinante para responder a estas tres preguntas: el registro de longitud vectorial `VLR` (*Vector Length Register*) y el registro de máscara `VM` (*Vector Mask*).

El `VLR` controla la longitud de cualquier operación vectorial, ya sean cargas, almacenamientos u

operaciones aritméticas. Si el tamaño del vector a procesar es menor que el valor de MVL (*Maximum Vector Length*), entonces basta con almacenar en el registro VLR la longitud del vector. Si el tamaño del vector a procesar es mayor que el valor MVL, el compilador recurre a una técnica denominada *strip mining* y que se traduce como troceado del vector o seccionamiento. El seccionamiento consiste en procesar un vector de longitud mayor al valor MVL en secciones de longitud igual o inferior al MVL. Para ello, el compilador genera un bucle con instrucciones escalares y vectoriales que se ocupa de procesar el vector en secciones de longitud MVL (y una de longitud menor si el tamaño del vector no es múltiplo de MVL). La Figura 3.24 ilustra de forma gráfica el seccionamiento de un vector genérico en secciones de MVL y un ejemplo de aplicación a un vector de 200 elementos

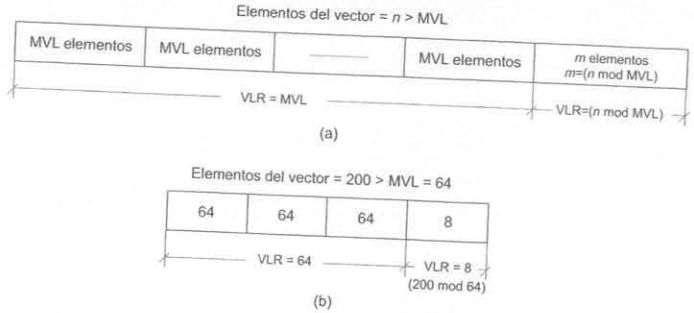


Figura 3.24: Seccionamiento de un vector genérico en secciones de MVL elementos (a) y de un vector de 200 elementos en secciones de 64 (b).

Para ello, en cada iteración se ajusta el valor del registro VLR y se cargan los registros vectoriales con la sección que corresponda. Si el tamaño del vector es de n elementos, en la primera iteración se procesan (n mod MVL) elementos y en las siguientes ya se procesan secciones de longitud MVL. Para ello, en la primera iteración se asigna el valor (n mod MVL) al registro VLR y en las siguientes el valor MVL. Para poder manipular el registro VLR, los procesadores vectoriales cuentan con algún tipo de instrucciones especiales análogas a las dos siguientes:

`MOVI2S VLR,Ri` Almacena en VLR el contenido del registro escalar `Ri`.
`MOVS2I Ri,VLR` Almacena en el registro escalar `Ri` el contenido de VLR.

El problema de ubicar en memoria los elementos de un vector de forma no consecutiva surge cuando hay que almacenar estructuras de datos que presentan dimensiones superiores a la unidad, tal y como sucede, por ejemplo, con los arrays bidimensionales. Un ejemplo de esta problemática aparece al multiplicar dos matrices de 100x100 almacenadas en los arrays A y B:


```
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
      C[i,j]:= 0.0;
      for (k=0; k<100; k++)
         C[i,j]:=C[i,j]+A[i,k]*B[k,j];
      end for;
   end for;
end for;
```

Así, dependiendo de si el almacenamiento se realiza por filas o columnas, el patrón de ubicación de los elementos de los arrays bidimensionales A, B y C variará. La Figura 3.25 ilustra las diferencias que existen entre almacenar un array bidimensional de 100x100 por filas o por columnas. Según el tipo de operación a realizar con el array, puede ser necesario acceder a los elementos que componen una fila o una columna, los cuales, según la forma de ubicación, tendrán una separación unitaria o de cien elementos.

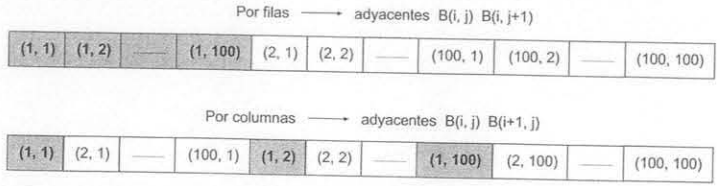


Figura 3.25: Almacenamiento unidimensional por filas o por columnas de una estructura bidimensional de 100x100.

Para poder solucionar el problema de cargar y almacenar datos que se encuentran ubicados en memoria de forma no consecutiva pero siguiendo un patrón uniforme, los procesadores vectoriales disponen de instrucciones especiales de carga y almacenamiento con indicación de la separación entre datos:

`LVWS Vi,(Ri,Rj)` Carga `Vi` comenzando desde la posicion `M[Ri]` con una separación de `Rj`.
`SVWS (Ri,Rj),Vi` Almacena `Vi` a partir de la posición `M[Ri]` con una separación de `Rj`.

En estas instrucciones, el contenido del registro `Ri` es la posición del primer dato y `Rj` mantiene la separación en memoria que existe entre los datos restantes. El sufijo `WS` corresponde a la expresión inglesa *with stride* que puede traducirse como *con distancia de separación*. Al igual que las instrucciones de carga y almacenamiento clásicas, estas instrucciones están también afectadas por el valor del registro especial VLR. Otra forma de manejar este tipo de cargas y almacenamientos es mantener instrucciones únicas de cargas y almacenamiento pero que utilizan un registro de propósito especial en el que se indica la separación que debe aplicarse en los accesos a memoria.

El tercer interrogante al que tiene que dar repuesta el repertorio de instrucciones de un procesador vectorial es el que se plantea cuando hay que vectorizar bucles en cuyo cuerpo hay instrucciones ejecutadas condicionalmente. Esto provoca que, dependiendo de la condición, ciertos elementos de un vector no tengan que ser manipulados. Para ello se recurre a una máscara de MVL bits de longitud almacenada en el registro especial VM. El valor del bit que ocupa la posición i en la máscara determina si se realizarán (bit a 1) o no (bit a 0) las operaciones sobre los elementos que ocupan la posición i en los registros vectoriales. Al igual que sucede con el registro VLR, existen instrucciones especiales para gestionar el contenido de este registro:

`S_V Vi,Vj`	Compara (EQ, NE, GT, LT, GE, LE) elemento a elemento el contenido de `Vi` y `Vj`. El resultado de la comparación de cada elemento es un bit (cierto=1, falso=0) que se almacena en el registro `VM` para formar una máscara.
`S_SV Fi,Vi`	Similar a la anterior pero utilizando un valor escalar en la comparación.
`RVM`	Inicializa a 1 todos los bits del registro `VM`.
`MOVF2S VM,Fi`	Almacena en `VM` el contenido del registro en coma flotante `Fi`.
`MOVS2F Fi,VM`	Almacena en el registro `Fi` el contenido del registro `VM`.

Un ejemplo habitual en el que es necesario recurrir a la utilización del registro VM es en la vectorización de bucles como el que se muestra a continuación:

```
for (i=0; i<64; i++)
   if (A[i]!=0) then
      B[i]:=B[i]/A[i];
   end if;
end for;
```

Dado que existe una instrucción con ejecución condicionada, es necesario recurrir a la utilización del registro de máscara vectorial, colocando previamente a 0 en el VM los bits correspondientes a los elementos de A[i] que son iguales a 0. Una posible codificación del bucle anterior con instrucciones vectoriales podría ser:

```
LV      V1,R1       % Carga en V1 el vector A
LV      V2,R2       % Carga en V2 el vector B
SNESV   F0,V1       % Compara cada elemento de A con 0
DIVV    V2,V2,V1    % Operación con control de máscara
RVM                 % Inicializa a 1 la máscara
SV      R2,V2       % Almacena todos los elementos de B
```

La mayor parte de los procesadores vectoriales incluyen varios registros de máscara para controlar las operaciones que se realizan sobre los registros vectoriales. Sin embargo, según cómo esté implementado el control de operaciones mediante máscara pueden surgir algunos inconvenientes. Por un lado, puede suceder que el tiempo de procesamiento no se reduzca pese a que no se realicen las operaciones, es

decir, que las no operaciones indicadas por la máscara se reemplacen por operaciones tipo NOP. Por otro lado, puede ocurrir que el enmascaramiento afecte únicamente al almacenamiento del resultado pero no evite la operación con lo que se podría dar lugar a la aparición de excepciones. En el ejemplo del bucle anterior, si se realizase una operación de división por cero se produciría una excepción pese al control de la máscara ya que la máscara evita el almacenamiento del resultado de la división, no la operación en sí misma.

3.15. Medida del rendimiento de un fragmento de código vectorial

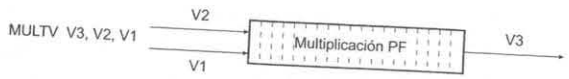
Para poder calcular la cantidad de ciclos de reloj que consume la ejecución de un fragmento de código vectorial es necesario, en primer lugar, entender cómo calcular el tiempo de ejecución de una única instrucción vectorial. Tras esto, y en función de las mejoras que se apliquen, como son el encadenamiento de resultados o el solapamiento de paquetes de instrucciones, podrá estimarse el tiempo total de ejecución de un fragmento de código vectorial o el cuerpo de un bucle vectorizado.

Tres son los factores que afectan al tiempo de ejecución de una única instrucción vectorial: la latencia en producir el primer resultado o tiempo de arranque, el número de elementos a procesar por la unidad funcional y el tiempo que se tarda en completar cada resultado o tiempo por elemento. La expresión que indica el número de ciclos que se tarda en completar una instrucción vectorial que procesa un vector de n elementos es

$$T_n = T_{arranque} + n * T_{elemento}$$

Con la excepción de la unidad funcional de carga/almacenamiento, en las restantes unidades funcionales vectoriales el tiempo de arranque es similar al número de segmentos de que constan. Es decir, si una unidad vectorial está compuesta por 6 etapas y se considera que cada etapa consume un ciclo, el tiempo de arranque es equivalente a la latencia inicial de la unidad, es decir, 6 ciclos. Tras esto, el $T_{elemento}$ que se tarda en completar cada uno de los restantes n resultados es un ciclo. Esto es así gracias a que la unidad funcional vectorial se encuentra segmentada y a que las operaciones que realiza sobre los elementos de un vector son independientes unas de otras. Esto permite un aprovechamiento muy eficaz de su cauce ya que tras la latencia inicial, correspondiente al tiempo de arranque, es posible tener en un instante dado ocupadas todas las etapas con elementos del vector. La Figura 3.26 ilustra el tiempo, expresado en ciclos de reloj, que tarda en procesarse una multiplicación de dos vectores de 64 elementos en una unidad funcional compuesta de 10 etapas donde cada etapa consume un ciclo de reloj.

Cuando se trata de calcular el tiempo de ejecución de una secuencia de instrucciones vectoriales es necesario tener en cuenta si las instrucciones vectoriales afectan o no a diferentes unidades funcionales y si existen dependencias verdaderas entre ellas, es decir, si son operaciones independientes. Si no existen dependencias verdaderas y no hay riesgos estructurales, varias instrucciones pueden planificarse y ejecutarse sin ningún tipo de penalización formando lo que se conoce como *convoy* o *paquete*. En principio, solo se puede emitir un nuevo convoy cuando todas las instrucciones del convoy anterior han



$T_{arranque}$ = 10 ciclos

$T_{elemento}$ = 1 ciclo

T_{64} = 10 ciclos + 64 elementos * 1 ciclo = 74 ciclos

Figura 3.26: Tiempo de ejecución de una operación de multiplicación sobre 2 vectores de 64 elementos.

finalizado su ejecución. A lo largo del capítulo se considerará que la velocidad de emisión es de un convoy por ciclo de reloj.

En caso de que existan dependencias verdaderas o estructurales entre instrucciones será necesario esperar a que los operandos estén disponibles o a que la unidad funcional se encuentre libre para poder formar varios convoyes. La Figura 3.27 ilustra estas dos posibles situaciones utilizando una unidad de suma y otra de multiplicación con tiempos de arranque de 6 y 7 ciclos, respectivamente, y una longitud de vector de 64.

Evidentemente, cuando ya se trata de una secuencia de instrucciones vectoriales, el tiempo por elemento es equivalente al tiempo promedio que emplean el conjunto de unidades funcionales implicadas en generar un nuevo resultado. Análogamente sucede con el tiempo de arranque, es decir, es la suma total de los ciclos de reloj que se consumen sin que las unidades produzcan resultados. En la Figura 3.27 se ilustra cómo se calcula el tiempo de arranque y el tiempo por elemento en ambas situaciones. Lo habitual para obtener el $T_{elemento}$ es obtener el tiempo total de ejecución de toda la secuencia, descontar los ciclos no productivos por inicialización de las unidades y dividir el valor final por el número de elementos del vector. Por lo general, el tiempo por elemento expresado en ciclos es igual al número de convoyes.

Además del T_n, otra medida que permite expresar el rendimiento de un procesador vectorial es el número de operaciones en coma flotante (FLOP) realizadas por ciclo de reloj:

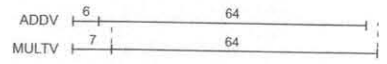
$$R_n = (\text{Operaciones en coma flotante} * n \text{ elementos}) / T_n$$

En los ejemplos de la Figura 3.27, las operaciones en coma flotante que se realizan sobre los 64 elementos son dos (una suma y un producto), por lo que el total de FLOP es 128. A mayor valor de R_n, mejor será el rendimiento obtenido al ejecutar el código vectorial.

Una técnica que permite mejorar el rendimiento de un procesador vectorial es el *encadenamiento* de resultados entre unidades funcionales (*chaining*). El encadenamiento permite que una unidad funcional pueda comenzar a operar tan pronto como los resultados de la unidad funcional de que depende estén disponibles. Este concepto es similar al mecanismo de adelantamiento de la etapa EX en las segmentaciones escalares si se considera que un registro vectorial de n elementos está formado por un conjunto de n registros individuales independientes. De esta forma, aunque dos operaciones sean

ADDV V1, V2, V3

MULTV V4, V5, V6



T_{64} = 7 + 64 = 71 ciclos

$T_{arranque}$ = 7 ciclos

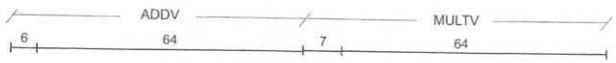
$$T_{elemento} = \frac{64 \text{ elementos} * 1 \text{ ciclo}}{64 \text{ elementos}} = 1 \text{ ciclo}$$

$$R_{64} = \frac{64 \text{ elementos} * 2 \text{ FLOP/elemento}}{71 \text{ ciclos}} = 1{,}8 \text{ FLOP / ciclo}$$

(a)

ADDV V1, V2, V3

MULTV V4, V5, V1



T_{64} = (6 + 64) + (7 + 64) = 141 ciclos

$T_{arranque}$ = 6 + 7 = 13 ciclos

$$T_{elemento} = \frac{64 \text{ elementos} * 1 \text{ ciclo} + 64 \text{ elementos} * 1 \text{ ciclo}}{64 \text{ elementos}} = 2 \text{ ciclos}$$

$$R_{64} = \frac{64 \text{ elementos} * 2 \text{ FLOP/elemento}}{141 \text{ ciclos}} = 0{,}9 \text{ FLOP / ciclo}$$

(b)

Figura 3.27: Ejecución de un convoy de 2 instrucciones (a) y de 2 convoyes de una única instrucción (b).

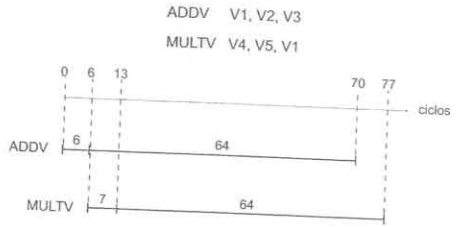
dependientes, el encadenamiento permite que se realicen en paralelo sobre elementos diferentes de un vector y formen parte del mismo convoy. Si se permite encadenamiento, dentro de un convoy pueden existir instrucciones dependientes pero nunca deben existir riesgos estructurales entre ellas. La Figura 3.28 presenta un ejemplo de encadenamiento en el que la unidad de multiplicación puede comenzar a operar en cuanto el primer resultado de la unidad de suma está disponible. El encadenamiento permite

reducir el tiempo por elemento gracias al solapamiento pero no así los tiempos de arranque.

Como se ha indicado previamente, no podrá emitirse otro convoy hasta que todas las instrucciones que componen el convoy que se ha emitido no concluyan su ejecución. Otra posibilidad es permitir la ejecución solapada de diferentes convoyes. Esto implica que una instrucción vectorial pueda comenzar a utilizar la unidad funcional antes de que una operación previa haya concluido. El permitir el solapamiento de convoyes complica mucho la lógica de emisión pero permite ocultar los tiempos de arranque para todos los convoyes excepto para el primero. La Figura 3.29.a presenta un ejemplo de 2 convoyes con encadenamiento entre unidades funcionales y sin solapamiento por lo que hasta que la última instrucción del primer convoy no se complete no se podrá emitir el siguiente convoy (ciclo 77). La Figura 3.29.b corresponde a la ejecución de los 2 convoyes pero con encadenamiento y solapamiento. Ahora, antes de que se completen las primeras instrucciones de suma y multiplicación se inicia la ejecución de las segundas de forma que los tiempos de arranque quedan ocultos.

3.16. La unidad funcional de carga/almacenamiento vectorial

El elemento hardware más crítico en un procesador vectorial para poder alcanzar un rendimiento óptimo es la unidad de carga/almacenamiento. La razón de ello es que debe ser capaz de poder intercambiar datos con los registros vectoriales de forma sostenida y a una velocidad igual o superior a la que las unidades funcionales aritméticas consumen o producen nuevos elementos, esto es, a



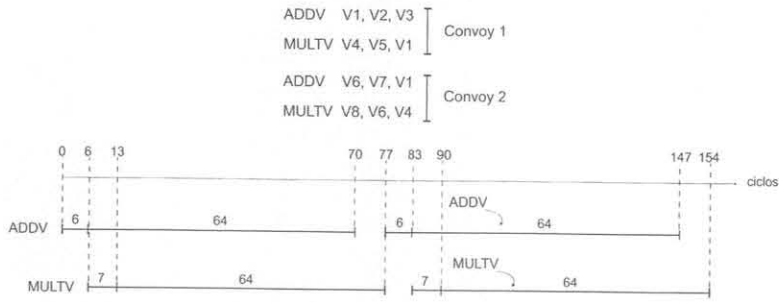
$T_{64} = (6 + 7) + 64 = 77$ ciclos

$T_{arranque} = 6 + 7 = 13$ ciclos

$$T_{elemento} = \frac{64 \text{ elementos} * 1 \text{ ciclo}}{64 \text{ elementos}} = 1 \text{ ciclo}$$

$$R_{64} = \frac{64 \text{ elementos} * 2 \text{ FLOP/elemento}}{77 \text{ ciclos}} = 1{,}66 \text{ FLOP / ciclo}$$

Figura 3.28: Ejecución de dos instrucciones con encadenamiento entre las unidades funcionales.



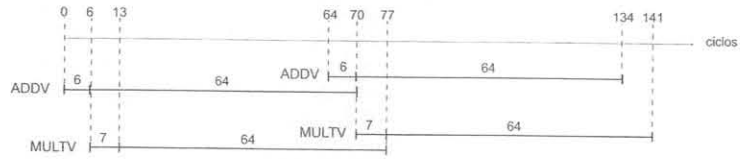
$T_{64} = (6 + 7) + 64 + (6 + 7) + 64 = 154$ ciclos

$T_{arranque} = 26$ ciclos

$$T_{elemento} = \frac{64 \text{ elementos} * 2 \text{ ciclos}}{64 \text{ elementos}} = 2 \text{ ciclos}$$

$$R_{64} = \frac{64 \text{ elementos} * 4 \text{ FLOP/elemento}}{154 \text{ ciclos}} = 1{,}66 \text{ FLOP / ciclo}$$

(a)



$T_{64} = (6 + 7) + 64 + 64 = 141$ ciclos

$T_{arranque} = 13$ ciclos

$$T_{elemento} = \frac{64 \text{ elementos} * 2 \text{ ciclos}}{64 \text{ elementos}} = 2 \text{ ciclos}$$

$$R_{64} = \frac{64 \text{ elementos} * 4 \text{ FLOP/elemento}}{141 \text{ ciclos}} = 1{,}81 \text{ FLOP / ciclo}$$

(b)

Figura 3.29: Ejecución de dos convoyes de instrucciones con encadenamiento entre unidades. Sin solapamiento entre convoyes (a) y con solapamiento (b).

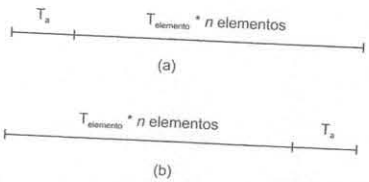


Figura 3.30: Diferencias entre instrucciones de carga (a) y almacenamiento (b) en la visibilidad de los tiempos de acceso y por elemento.

$T_{elemento}$. Esta alta velocidad de transferencia se consigue organizando físicamente la memoria en varios bancos o módulos y distribuyendo el espacio de direccionamiento de forma uniforme entre todos ellos. Dimensionando adecuadamente el número de bancos de memoria en función del tiempo de acceso a un banco, T_a, se consigue que la latencia para extraer/almacenar un dato de/en memoria quede oculta mientras se transfieren los demás elementos que componen un vector a/desde un registro vectorial. Por dato o elemento se entiende una palabra que en el caso de un procesador vectorial suele tener una longitud de 8 bytes.

Aunque la unidad de carga/almacenamiento difiere en su organización y funcionamiento de las aritmético-lógicas, también se puede caracterizar por un tiempo de arranque y un tiempo por elemento, aunque su significado físico sea diferente. En una operación de carga de un registro vectorial, el tiempo de arranque es igual a T_a ya que es el tiempo que transcurre desde que se solicita el primer elemento del vector al sistema de memoria hasta que está disponible en el puerto de lectura del banco para su posterior transferencia al registro vectorial. El tiempo por elemento se considera como el número de ciclos que se consumen en transferir el dato ya disponible en el banco de memoria al registro vectorial (por lo general, inferior a un ciclo pero se iguala a uno para equipararlo al $T_{elemento}$ de las unidades funcionales). Análogamente, en una operación de escritura en memoria, el tiempo por elemento se considera como el tiempo que se emplea en transferir un elemento desde un registro vectorial al puerto de escritura del banco de memoria y el tiempo de arranque se puede ver como el tiempo que emplea el banco en escribir el último elemento del vector en una posición del banco de memoria. Por lo tanto, el tiempo de arranque y el tiempo por elemento se ven de forma opuesta según se trate de una operación de carga o de almacenamiento (Figura 3.30) aunque, en ambos casos, el tiempo que consume una operación de acceso a memoria para un vector de n elementos es similar, esto es, $(T_a + n*T_{elemento})$ ciclos.

Dado que cada vez que se solicita un dato de un vector al sistema de memoria hay que pagar siempre la latencia de acceso T_a, la lectura de los n elementos de que consta un vector supondría un coste de $n*T_a$ ciclos, lo que es totalmente inadmisible si se pretende mantener un ancho de banda sostenido igual a $T_{elemento}$. Para poder ocultar toda esta latencia es fundamental dimensionar correctamente el número de bancos de memoria de forma que solo se vea el tiempo de acceso correspondiente al primer elemento del vector y que los tiempos de acceso de los demás elementos queden ocultos. Esto se consigue distribuyendo los n elementos de un vector entre m bancos de memoria para que se solapen los $(n-1)*T_a$

ciclos de acceso de $n-1$ elementos del vector con los $n*T_{elemento}$ ciclos que se emplean en enviar los n elementos a un registro vectorial. Esta latencia inicial es lo que se considera como el tiempo de arranque de la unidad de carga/almacenamiento. Un dimensionamiento correcto siempre tiene que cumplir que el número de bancos de memoria m sea mayor o igual que la latencia de acceso al banco de memoria expresada en ciclos de reloj, es decir, $m \geq T_a$.

Aunque el resultado final es similar, existen dos formas de realizar el solapamiento de las latencias de acceso a los n datos de un vector: de forma *síncrona* y de forma *asíncrona*. Por simplicidad, al describir las dos formas de acceso se considerará que T_a es igual a m y que se está realizando una operación de carga vectorial. La aproximación síncrona implica solicitar simultáneamente un dato a los m bancos cada T_a ciclos. Realizada la primera petición y transcurridos T_a ciclos estarán disponibles los m primeros elementos del vector para ser transferidos al registro vectorial, lo que consume $m*T_{elemento}$ ciclos. Ahora, cada T_a ciclos se realizan dos acciones: (1) se efectúa una nueva petición simultánea a todos los bancos para extraer los m elementos siguientes del vector y (2) se comienza a transferir ciclo a ciclo los m elementos ya disponibles y que fueron solicitados en la petición anterior. La Figura 3.31.a ilustra de forma gráfica esta forma de solapar los tiempos de acceso al realizar la carga vectorial.

El acceso asíncrono implica solicitar los elementos de que consta el vector a cada uno de los m bancos de forma periódica con periodo T_a con un desfase entre bancos consecutivos de $T_{elemento}$ ciclos. De esta forma, comenzando en el ciclo 0 y cada T_a ciclos el banco 0 solicita un dato, en el ciclo 1 y cada T_a ciclos el banco 1 solicita un nuevo dato, y así sucesivamente hasta el banco $m-1$. En el momento en que un banco tiene el dato disponible realiza dos acciones: (1) efectúa la nueva solicitud de dato y (2) transfiere el dato ya disponible al registro vectorial. La Figura 3.31.b muestra cómo se realiza el acceso asíncrono a los bancos. El mismo razonamiento se aplicaría para las escrituras pero de forma inversa: primero se transfieren los elementos y luego se realizan los accesos al banco.

Evidentemente, para que todo funcione correctamente las palabras que componen un vector deben estar distribuidas correctamente entre los bancos de memoria. Los bancos de memoria, por lo general, tienen un ancho de palabra de 8 bytes, se direccionan por bytes y son un número que es potencia de 2 (para simplificar el direccionamiento). La distribución de los elementos de un vector en los bancos de memoria se realiza de forma consecutiva y cíclica a partir de una posición de memoria inicial que es múltiplo del ancho de palabra en bytes (en este caso, 8 bytes). Cuando se va a leer un vector de memoria, se analiza la posición de memoria del primer elemento para así conocer el número del banco en que se encuentra, tras lo que se inicia la lectura de todo el vector tal y como se ha indicado anteriormente. La Figura 3.32 muestra un ejemplo en el que se distribuyen las palabras entre 8 bancos de memoria direccionados por bytes.

Dada una dirección de memoria, el número de banco en que se encuentra está determinado por los bits de orden inferior de la dirección de memoria teniendo en cuenta que la distancia entre datos es de 8 bytes. Para entender cómo se realiza, a continuación se muestran en binario algunas de las direcciones de memoria tratadas en el ejemplo de la Figura 3.32.

136: 0x10**001**000 (banco 1)

144: 0x10**010**000 (banco 2)

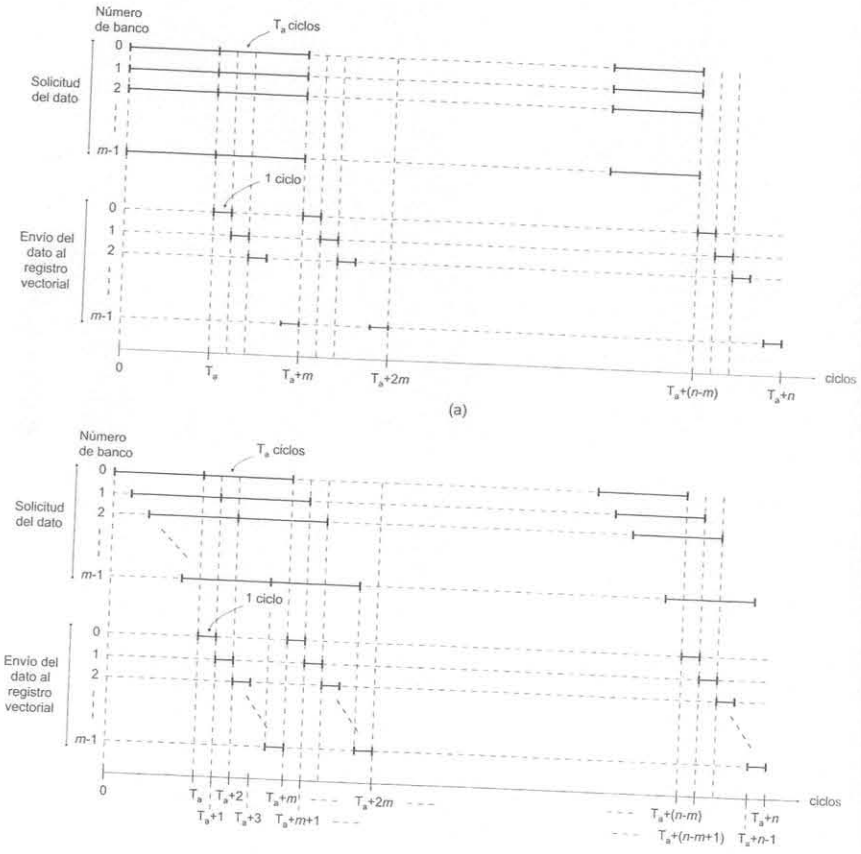


Figura 3.31: Acceso síncrono (a) y asíncrono (b) a los bancos de memoria para transferir un vector a un registro vectorial mediante una operación de carga. Se considera $T_a = m$.

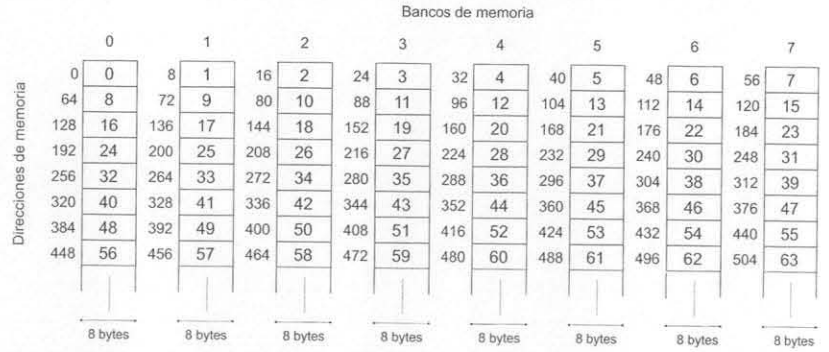


Figura 3.32: Almacenamiento en 8 bancos de un vector de 64 elementos de doble precisión a partir de la dirección de memoria 0. Los números contenidos en las celdas representan el orden del elemento en el vector, no su valor.

152: 0x10**011**000 (banco 3)

160: 0x10**100**000 (banco 4)

168: 0x10**101**000 (banco 5)

176: 0x10**110**000 (banco 6)

184: 0x10**111**000 (banco 7)

192: 0x11**000**000 (banco 0)

Despreciando los tres bits de orden inferior al direccionar datos de 8 bytes, los tres bits siguientes determinan el banco de memoria en que se encuentra la dirección de memoria que se pretende leer. En este caso, los bits que determinan el banco de memoria son 3 ya que hay 8 bancos de memoria, es decir, $\log_2(8) = 3$.

Evidentemente, aunque en los ejemplos anteriores el primer elemento de un vector se ubicaba en el banco 0, el segundo elemento en el banco 1 y así sucesivamente, lo habitual es que el primer elemento del vector se ubique en cualquier otro banco. Si, por ejemplo, el acceso a los bancos fuese asíncrono con T_a de 6 ciclos y el primer elemento del vector se ubicase en la dirección 80, la primera petición se realizaría al banco 2 (80=0x01**010**000), la segunda al banco 3 (88=0x01**011**000), la tercera al banco 4 (96=0x01**100**000), y así sucesivamente. En lo que respecta al banco 2 y una vez solicitado el contenido de la dirección 80 en el ciclo 0, en el ciclo 6 ya se dispondría del dato y se enviaría al registro vectorial, en el ciclo 8 se solicitaría la dirección 144 y así sucesivamente. La Figura 3.33 ilustra el proceso de solicitud y transferencia de los datos.

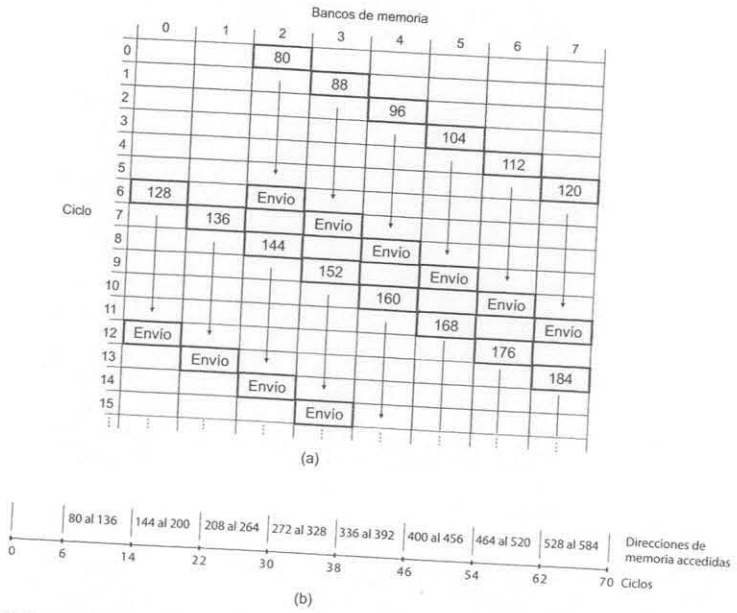


Figura 3.33: Esquema de la transferencia de un vector de 64 elementos ubicado a partir de la posición de memoria 80 en un sistema de memoria organizado en 8 bancos con acceso asíncrono y con T_a de 6 ciclos (a). Relación temporal entre los accesos a los bancos y las palabras enviadas (b).

3.17. Medida del rendimiento de un bucle vectorizado

En los apartados anteriores se ha descrito cómo calcular el rendimiento de un conjunto de instrucciones vectoriales organizadas en uno o dos convoyes. Sin embargo, la estimación del tiempo de ejecución en un procesador vectorial de un bucle que ha sido vectorizado implica tener en cuenta factores adicionales. Como ya se ha indicado, para vectorizar un bucle, un compilador aplica la técnica *strip mining* y produce una mezcla de código escalar y vectorial en el que el procesamiento del bucle se realiza por secciones o trozos. El siguiente pseudo-código muestra las operaciones que son necesarias para vectorizar el bucle DAXPY de n elementos con la técnica *strip mining*:

```
1: primero:=1;
2: secciones:=n/MVL;                    % Primer elemento de la sección
                                        % Secciones de MVL elementos
```



```
3: VLR:=n mod MVL;                      % Longitud de la primera sección
4: for (i=0;i<=secciones;i++)           % Bucle exterior
5:    último:=primero+VLR-1;            % Último elemento de la sección
6:    for (j=primero; j<=último;j++)    % Bucle interior
7:       Y[i]:=a*X[i]+Y[i];             % Operaciones vectoriales
8:    end for;
9:    primero:=primero+VLR;             % Primer elemento de la nueva sección
10:   VLR:=MVL;                         % Inicio longitud de la nueva sección
11:end for;
```

donde n contiene el número de elementos del vector, `primero` indica la posición en el vector del primer elemento de la sección a procesar, `secciones` es el número total de secciones de MVL elementos que hay en el vector, `VLR` indica la longitud de la sección del vector a procesar en cada iteración del bucle interior (la primera vez será menor o igual que MVL) y `último` es la posición en el vector del último elemento de la sección que se procesa. Mediante el bucle exterior (líneas 4 a 11), el pseudocódigo descompone el procesamiento del vector de n elementos en secciones de una longitud máxima de MVL elementos, con la salvedad de la primera sección que puede tener una longitud inferior si n no es múltiplo de MVL (línea 3). Por esa razón, tras la primera iteración del bucle interior es necesario reiniciar el valor de `VLR` a MVL (línea 10).

Con la salvedad del bucle interior (líneas 6, 7 y 8), que el compilador traduce en instrucciones vectoriales, el resto de las operaciones es código escalar con sus costes adicionales. Los costes de ejecución de las instrucciones vectoriales y escalares obtenidas de la vectorización del bucle se pueden desglosar en cuatro componentes:

- T_{base}: Es el tiempo que consumen las instrucciones escalares de preparación antes de abordar el bucle exterior. Corresponderían a las instrucciones escalares generadas para las líneas 1, 2 y 3. En los procesadores vectoriales actuales este tiempo se ha reducido mucho pudiendo incluso llegar a ser despreciable con respecto a los demás factores.

- T_{bucle}: Son los costes derivados de ejecutar en cada iteración del bucle exterior las instrucciones escalares necesarias para realizar el seccionamiento. Son los tiempos de ejecución de las instrucciones escalares derivadas de las líneas 4, 5, 9, 10 y 11.

- $T_{arranque}$: Es la suma de los tiempos de arranque visibles de las unidades funcionales que se utilizan en cada convoy de instrucciones.

- $T_{elemento}$: Es igual al número de convoyes en que se organizan las instrucciones vectoriales que se derivan del bucle interior (líneas 6, 7 y 8). Su valor depende fuertemente de las características de la unidad vectorial: encadenamiento y número de unidades funcionales.

En base a estos tiempos, la expresión que permite determinar el tiempo total de ejecución en ciclos de reloj de un bucle vectorizado que realiza operaciones sobre vectores de longitud n es

$$T_n = T_{base} + \left\lceil \frac{n}{\text{MVL}} \right\rceil * \left(T_{bucle} + T_{arranque}\right) + n * T_{elemento}$$

donde el operador *ceiling* $\lceil x \rceil$ representa el siguiente valor entero superior a x. En la expresión de T_n, este operador obtiene el número total de secciones en que se dividen los vectores a procesar, con independencia de que haya una sección de longitud inferior a MVL. El resultado del operador se multiplica por los costes asociados al procesamiento de cada sección, T_{bucle} y $T_{arranque}$, ya que ambos dependen del número de secciones en que se descompone el vector. Finalmente, para obtener el número total de ciclos hay que añadir el $T_{elemento}$ que consumen los n elementos del vector y los costes adicionales que introducen las instrucciones escalares iniciales, es decir, el T_{base}.

Otra medida del rendimiento de un bucle vectorizado es la velocidad de procesamiento de un bucle de longitud infinita expresada en FLOP por ciclo de reloj. Se expresa como

$$R_\infty = \lim_{n \to \infty} \left(\frac{\text{Operaciones vectoriales} * n}{T_n} \right)$$

Si se desease expresar el rendimiento en FLOPS (FLOP por segundo) habría que multiplicar el resultado de la expresión previa por la frecuencia de reloj del procesador vectorial.

Para concluir este apartado, se va a realizar un ejemplo del análisis del rendimiento de un procesador vectorial al ejecutar el código obtenido de vectorizar el conocido bucle DAXPY para vectores de n elementos. El procesador vectorial consta de una unidad de suma (6 ciclos de latencia), una unidad de multiplicación (7 ciclos de latencia), una unidad de carga/almacenamiento (12 ciclos de latencia), MVL es 64 y la frecuencia de reloj es 500 MHz. El fragmento de código vectorial que se genera para realizar las operaciones `Y(i):=a*X(i)+Y(i)` es:

```
LV      V1,R1       % Carga de una sección de X
MULTSV  V2,V1,F0    % Operación vectorial a*X
LV      V3,R2       % Carga de una sección de Y
ADDV    V4,V3,V2    % Operación vectorial a*X+Y
SV      R2,V4       % Almacenamiento sección de Y
```

y los costes debidos a las instrucciones escalares son $T_{base} = 10$ ciclos y $T_{bucle} = 15$ ciclos. Inicialmente, se considera que solo se emite un nuevo convoy cuando las instrucciones del convoy anterior han terminado.

Caso 1: Sin encadenamiento de resultados entre unidades

Analizando los riesgos estructurales y de datos existentes entre las cinco instrucciones vectoriales, los convoyes existentes son cuatro:


```
Convoy 1:   LV      V1,R1
Convoy 2:   MULTSV  V2,V1,F0
            LV      V3,R2
Convoy 3:   ADDV    V4,V3,V2
Convoy 4:   SV      R2,V4
```

La secuencia de ejecución de los cuatro convoyes, si se considera que VLR es 64, es la que se muestra en la Figura 3.34.

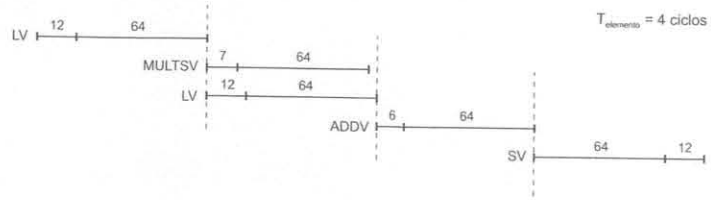


Figura 3.34: Secuencia de ejecución de los cuatro convoyes con VLR = 64.

Dado que hay cuatro convoyes, $T_{elemento}$ es 4 ciclos y el $T_{arranque}$ total es igual a la suma de los tiempos de arranque visibles de los cuatro convoyes. Esto es

$$T_{arranque} = 2 * T_{arranqueLV} + T_{arranqueADDV} + T_{arranqueSV}$$

$$T_{arranque} = (2 * 12 + 6 + 12) \text{ ciclos} = 42 \text{ ciclos}$$

Sustituyendo los valores conocidos de $T_{arranque}$ y $T_{elemento}$ en la expresión que determina el tiempo de ejecución de un bucle vectorizado para vectores de longitud n se tiene

$$T_n = 10 + \left\lceil \frac{n}{64} \right\rceil * (15 + 42) + 4 * n$$

que para el caso particular de $n = 1000$ es

$$T_{1000} = 10 + \left\lceil \frac{1000}{64} \right\rceil * (15 + 42) + 4 * 1000$$

$$T_{1000} = 10 + 16 * (15 + 42) + 4 * 1000$$

$$T_{1000} = 4922 \text{ ciclos}$$

El rendimiento expresado en FLOP/ciclo es

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{T_n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10+\left\lceil\frac{n}{64}\right\rceil*(15+42)+4*n}\right)$$

Para simplificar los cálculos, la expresión $\lceil n/64\rceil$ se puede reemplazar por una cota superior dada por $(n/64+1)$. Sustituyendo esta cota en R_∞ y teniendo en cuenta que el número de operaciones vectoriales que se realizan en el bucle DAXPY son dos, una multiplicación y una suma, se tiene

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10+\left(\frac{n}{64}+1\right)*(15+42)+4*n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{67+4,89*n}\right)$$

$$R_\infty = 0,409\,\text{FLOP/ciclo}$$

Para expresar R_∞ en FLOPS habría que multiplicar el valor en FLOP/ciclo por la frecuencia del procesador. Se tendría así

$$R_\infty = 0,409\,\text{FLOP/ciclo} * (500 * 10^6)\,\text{Hz}$$

$$R_\infty = 204,5\,\text{MFLOPS}$$

Caso 2: Con encadenamiento de resultados entre unidades

Dado que ahora es posible encadenar los resultados de las unidades, la organización del código vectorial en convoyes quedaría de la siguiente forma:

```
Convoy 1:   LV      V1,R1
            MULTSV  V2,V1,F0
Convoy 2:   LV      V3,R2
            ADDV    V4,V3,V2
Convoy 3:   SV      R2,V4
```


El tercer convoy se mantiene debido a que no se permite que en un mismo convoy haya instrucciones que presenten riesgos estructurales. Por esa razón, en el segundo convoy no pueden coexistir dos instrucciones con acceso a la única unidad de carga/almacenamiento disponible. La secuencia de ejecución de los tres convoyes se muestra en la Figura 3.35.

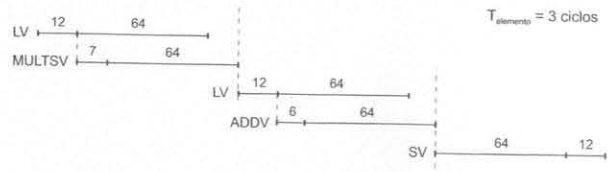


Figura 3.35: Secuencia de ejecución de los tres convoyes con VLR = 64

El $T_{elemento}$ ha pasado a ser de 3 ciclos dado que ahora se tienen 3 convoyes. El $T_{arranque}$ total se obtiene de sumar los tiempos de arranque visibles de las unidades funcionales. Si se analiza la Figura 3.35 se tiene

$$T_{arranque} = 2 * T_{arranqueLV} + T_{arranqueMULTSV} + T_{arranqueADDV} + T_{arranqueSV}$$

$$T_{arranque} = (2 * 12 + 7 + 6 + 12)\,\text{ciclos} = 49\,\text{ciclos}$$

Con estos valores la expresión del tiempo total de ejecución queda

$$T_n = 10 + \left\lceil\frac{n}{64}\right\rceil * (15+49) + 3 * n$$

que para el caso particular de $n = 1000$

$$T_{1000} = 10 + \left\lceil\frac{1000}{64}\right\rceil * (15+49) + 3 * 1000$$

$$T_{1000} = 10 + 16 * (15+49) + 3 * 1000$$

$$T_{1000} = 4034\,\text{ciclos}$$

Con respecto al caso 1, el permitir encadenamiento de resultados entre unidades funcionales ha reducido el tiempo de ejecución un 18 %, pasando de 4922 a 4034 ciclos. En lo que respecta al rendimiento expresado en FLOP por ciclo

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{T_n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10+\left\lceil\frac{n}{64}\right\rceil*(15+49)+3*n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10+\left(\frac{n}{64}+1\right)*(15+49)+3*n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{74+4*n}\right)$$

$$R_\infty = 0,5\,\text{FLOP/ciclo}$$

Para expresar R_∞ en FLOPS hay que multiplicar el valor anterior por la frecuencia del procesador. Se tiene así

$$R_\infty = 0,5\,\text{FLOP/ciclo} * (500 * 10^6)\,\text{Hz}$$

$$R_\infty = 250\,\text{MFLOPS}$$

Claramente se aprecia la mejora en el rendimiento del procesador gracias al encadenamiento de los resultados entre las unidades funcionales.

Caso 3: Con encadenamiento y dos unidades de carga/almacenamiento

Si se permite el encadenamiento y el número de unidades de carga/almacenamiento se duplica se obtiene la siguiente secuencia de convoyes:

```
Convoy 1:   LV      V1,R1
            MULTSV  V2,V1,F0
            LV      V3,R2
Convoy 2:   ADDV    V4,V3,V2
            SV      R2,V4
```

Ahora el primer y el segundo convoy del caso 2 se pueden unir gracias a que la existencia de una segunda unidad de acceso a memoria permite colocar dos instrucciones de carga en el mismo convoy sin que existan riesgos estructurales. La Figura 3.36 muestra la secuencia de ejecución.

Dado que hay dos convoyes, el $T_{elemento}$ es de 2 ciclos y del análisis de la Figura 3.36 se obtiene que

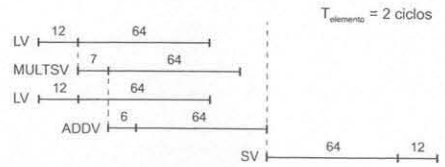


Figura 3.36: Secuencia de ejecución de los dos convoyes con VLR = 64.

el tiempo de arranque total de las unidades funcionales es

$$T_{arranque} = T_{arranqueLV} + T_{arranqueMULTSV} + T_{arranqueADDV} + T_{arranqueSV}$$

$$T_{arranque} = (12+7+6+12)\,\text{ciclos} = 37\,\text{ciclos}$$

Se tiene ahora

$$T_n = 10 + \left\lceil\frac{n}{64}\right\rceil * (15+37) + 2*n$$

y para $n = 1000$

$$T_{1000} = 10 + \left\lceil\frac{1000}{64}\right\rceil * (15+37) + 2*1000$$

$$T_{1000} = 10 + 16*(15+37) + 2*1000$$

$$T_{1000} = 2842\,\text{ciclos}$$

Con respecto al caso 1 la mejora es del 73 % ya que el total de ciclos consumidos para procesar el bucle DAXPY con 1000 elementos se ha reducido de 4922 a 2842. El rendimiento en FLOP/ciclo es

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{T_n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10+\left\lceil\frac{n}{64}\right\rceil*(15+37)+2*n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10+\left(\frac{n}{64}+1\right)*(15+37)+2*n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{62+2,8125*n}\right)$$

$$R_\infty = 0,711\ \text{FLOP/ciclo}$$

Si se expresa en FLOPS, se obtiene

$$R_\infty = 0,711\ \text{FLOP/ciclo} * (500 * 10^6)\ \text{Hz}$$

$$R_\infty = 355,55\ \text{MFLOPS}$$

Caso 4: Con encadenamiento, dos unidades de carga/almacenamiento y solapamiento entre convoyes dentro de la misma iteración

El último caso contempla que se puedan solapar convoyes en una misma iteración del bucle permitiendo así que una unidad funcional pueda ser utilizada por otra instrucción antes de que abandone la unidad la instrucción actual. Además, se considerará que las instrucciones escalares se pueden ejecutar en paralelo a las instrucciones vectoriales, por lo que T_{bucle} desaparece al quedar oculto tras los tiempos de ejecución de las instrucciones vectoriales.

En el bucle DAXPY, el número de convoyes continúa siendo dos. Sin embargo, es posible solapar en una unidad de carga/almacenamiento una de las dos instrucciones LV del primer convoy y la instrucción SV del segundo convoy. De esta forma, la ejecución de los dos convoyes se superpone dando lugar a un $T_{elemento}$ inferior al número de convoyes. En el diagrama de la Figura 3.37 se aprecia esta situación.

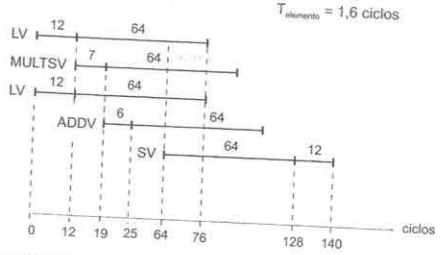


Figura 3.37: Secuencia de ejecución de los dos convoyes con VLR = 64.

Para calcular el $T_{elemento}$ en situaciones con solapamiento, hay que recordar que para un fragmento de código vectorial se tiene que

$$T_{elemento} = \left(T_n - T_{arranque}\right)/n$$

Dado que para un valor de VL de 64, el tiempo de ejecución de las instrucciones vectoriales en este caso es de 140 ciclos (Figura 3.37) y los tiempos de arranque son

$$T_{arranque} = T_{arranqueLV} + T_{arranqueMULTSV} + T_{arranqueADDV} + T_{arranqueSV}$$

$$T_{arranque} = (12+7+6+12)\,\text{ciclos} = 37\,\text{ciclos}$$

Se tiene así

$$T_{elemento} = \left(T_{64} - T_{arranque}\right)/64$$

$$T_{elemento} = (140-37)/64$$

$$T_{elemento} = 1,6\,\text{ciclos}$$

Como T_{bucle} es cero dado su solapamiento con el código vectorial, el tiempo de ejecución del bucle vectorizado para n elementos es

$$T_n = 10 + \left\lceil \frac{n}{64} \right\rceil * 37 + 1,6 * n$$

y particularizando para un vector de 1000 elementos

$$T_{1000} = 10 + \left\lceil \frac{1000}{64} \right\rceil * 37 + 1,6 * 1000$$

$$T_{1000} = 10 + 16 * 37 + 1600$$

$$T_{1000} = 2202\,\text{ciclos}$$

La mejora que se obtiene con respecto al caso 1 es del 123 % al reducirse el total de ciclos consumidos de 4922 a 2202. El rendimiento en FLOP/ciclo es

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{T_n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10 + \left\lceil \frac{n}{64} \right\rceil * 37 + 1,6*n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{10+\left(\frac{n}{64}+1\right)*37+1,6*n}\right)$$

$$R_\infty = \lim_{n\to\infty}\left(\frac{2*n}{47+2,18*n}\right)$$

$$R_\infty = 0,918\,\text{FLOP/ciclo}$$

y si se expresa R_∞ en FLOPS se obtiene

$$R_\infty = 0,918\,\text{FLOP/ciclo} * (500 * 10^6)\,\text{Hz}$$

$$R_\infty = 459\,\text{MFLOPS}$$

3.18. Resumen: Visión global y perspectivas de futuro

A lo largo de este capítulo se han descrito en profundidad dos estilos arquitectónicos que persiguen la mejora de las prestaciones de un procesador individual de formas completamente distintas. Por un lado, un procesador VLIW busca un incremento del rendimiento aprovechando el paralelismo a nivel de instrucción mediante la emisión de una instrucción por ciclo de reloj compuesta por varias operaciones o microinstrucciones que pueden ejecutarse en paralelo. Por otro lado, los procesadores vectoriales persiguen la mejora recurriendo al paralelismo a nivel de datos para lo que disponen de instrucciones capaces de procesar un conjunto de datos de forma continua y sin detenciones. En ambos casos, el factor determinante para extraer el máximo rendimiento de estos procesadores es el compilador. En las arquitecturas VLIW, el compilador debe ser capaz de formar y planificar las instrucciones utilizando para ello las instrucciones básicas del código intermedio y las características del procesador. En un procesador vectorial, el compilador extrae el máximo rendimiento mediante una detección y vectorización de los bucles presentes en el código original.

Sin embargo, un computador construido en base a un procesador individual, ya sea VLIW o vectorial, no es rival hoy en día para un procesador superescalar dado el enorme rendimiento que este último es capaz de obtener. Pero, como ya se ha señalado, los procesadores superescalares también tienen sus limitaciones. Por esa razón, la tendencia actual y futura para poder construir computadores cada vez más potentes es la replicación de procesadores individuales, obviamente sin abandonar la investigación en posibles mejoras que se puedan introducir a nivel de procesador individual ya sea mediante hardware o software.

Si a nivel de computador personal se avanza mediante la combinación de varios núcleos (procesadores superescalares con extensiones vectoriales) en un único chip, a nivel de supercomputación la evolución consiste en la construcción de multicomputadores y multiprocesadores mediante la replicación masiva de componentes disponibles comercialmente o mediante diseños a medida. En el ámbito de los computadores personales y pequeños servidores, ya se pueden adquirir máquinas con hasta 8 núcleos, mientras que ya se encuentran operativos supercomputadores constituidos por más de 200.000 núcleos.

En el caso de los supercomputadores, la evolución se realiza de dos formas: mediante la replicación de unidades superescalares con extensiones vectoriales o, al contrario, mediante la replicación de unidades vectoriales con extensiones superescalares. En el primer caso lo habitual es utilizar procesadores disponibles comercialmente (AMD Opteron, PowerPC, Intel Xeon, IBM PowerXCell, IBM PowerPC 450, etc.) y conectarlos mediante redes comerciales (Gigabit Ethernet, Infiniband, Myrinet, etc.), propietarias o diseñadas a medida. Si se analiza la configuración de los computadores que forman el ranking de supercomputadores TOP500 (*http://www.top500.org*), claramente, la primera línea es la predominante. En noviembre de 2010, de los 500 supercomputadores que formaban el ranking, 497 eran sistemas escalares y solo había uno vectorial.

Ejemplos de supercomputadores construidos mediante replicación de unidades vectoriales y redes propietarias son los basados en la arquitectura SX de NEC Corporation, siendo la serie SX-9 su representante más moderno. La arquitectura SX se basa en replicar nodos de procesamiento compuestos por procesadores vectoriales NEC y comunicarlos mediante una red de interconexión total NEC IXS. En concreto, la serie SX-9 permite construir desde computadores constituidos por un nodo con cuatro procesadores vectoriales, 256 Gbytes de espacio de almacenamiento y un rendimiento pico de 409,6 GFLOPS hasta llegar a sistemas compuestos por 512 nodos de 16 procesadores vectoriales, 512 TBytes de memoria y un rendimiento máximo de 839 TFLOPS. Los procesadores vectoriales SX-9 se basan en una versión mejorada del procesador vectorial que introdujo NEC en la serie SX-6 y llegan a alcanzar un rendimiento máximo de 102 GFLOPS. Cada procesador vectorial SX-9 consta de 8 unidades vectoriales y una escalar (Figura 3.38). La unidad vectorial corre a una velocidad de 3,2 GHz y se compone de siete unidades funcionales y una de carga/almacenamiento todas ellas con acceso a 72 registros vectoriales de 64 elementos de 8 bytes y 16 registros de máscara de 64 bits. El número de instrucciones vectoriales que se pueden emitir de forma simultánea es siete. Por su parte, la unidad escalar funciona a la mitad de velocidad que la vectorial, consta de 128 registros de 64 bits, seis unidades funcionales segmentadas y permite la ejecución de instrucciones fuera de orden.

Otra forma de utilización de la aproximación vectorial es la replicación de varias unidades escalares en coma flotante para formar unidades vectoriales virtuales. IBM desarrolló e implementó la tecnología ViVA (*Virtual Vector Architecture*) en el proyecto BluePlanet de forma que 8 procesadores IBM Power5 de doble núcleo trabajaban como un procesador vectorial logrando un rendimiento de entre 60 y 80 GFLOPS. Actualmente, la tecnología ViVA-2 está incorporada en los procesadores IBM Power6.

Paradójicamente, donde más se encuentra en vigor el concepto de procesamiento vectorial es actualmente en el mercado de los computadores de propósito general ya que la mayor parte de los procesadores superescalares incorporan una o varias unidades funcionales para realizar operaciones vectoriales. Ejemplos de repertorios de instrucciones para procesamiento vectorial que incorporan los

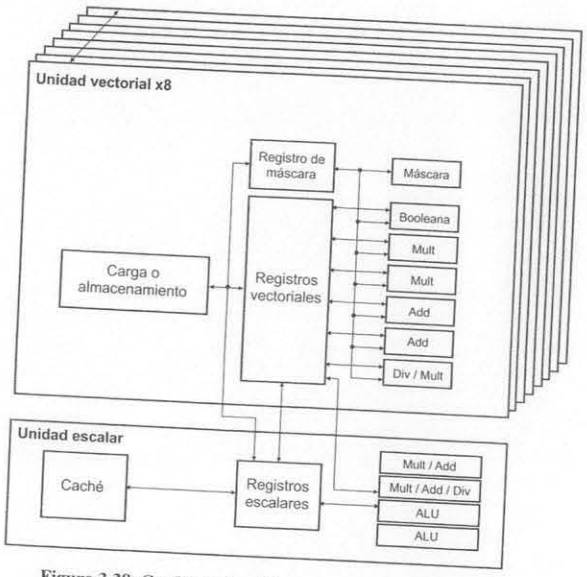


Figura 3.38: Configuración del procesador vectorial NEC SX-9.

procesadores superescalares actuales son los SSE2, SSE3, SSSE3, SSE4, FMA3, AVE, XOP, FMA4, CVT16 en procesadores de Intel y/o AMD o el AltiVec presente en los procesadores PowerPC. Otro ámbito en el que el procesamiento vectorial ha encontrado un campo de aplicación muy prometedor es el de las consolas de videojuegos y las tarjetas gráficas. Un ejemplo de ello es el procesador Cell, desarrollado por IBM, Sony y Toshiba, y que consta de un procesador escalar y ocho procesadores vectoriales. En la actualidad, este procesador se utiliza en tarjetas para el procesamiento de vídeo, consolas de videojuegos (Sony PlayStation 3) y sistemas de cine para el hogar (*home-cinema*). Curiosamente, tanto el Cell como su evolución, el IBM PowerXCell 8i, son utilizados actualmente por IBM en algunos de sus computadores de altas prestaciones (por ejemplo, el supercomputador híbrido IBM Roadrunner compuesto por procesadores AMD Opteron e IBM PowerXCell) y en sus mainframes.

En lo que respecta al futuro de los procesadores VLIW, con la salvedad de los procesadores Intel Itanium, el concepto VLIW todavía permanece en el ámbito de los procesadores empotrados o embebidos de alto rendimiento. ¿Cuál es la razón de ello? Los procesadores superescalares surgieron como consecuencia de la necesidad de desarrollar procesadores escalares más potentes pero capaces de mantener la compatibilidad hacia atrás que siempre demanda el mercado. Los usuarios y empresas a

nivel mundial invierten ingentes cantidades de dinero en la compra de software por lo que exigen que las nuevas máquinas que adquieran sean capaces de ejecutar el software de que ya disponen. Se podría argumentar que el problema se soluciona recompilando el código fuente de las aplicaciones pero la inmensa mayoría del software se compra como código binario, directamente ejecutable en el procesador y sin acceso al código fuente. Este problema se conoce como *inercia del repertorio de instrucciones*. Esta inercia provoca que los nuevos procesadores superescalares que se diseñan tengan que seguir siendo capaces de ejecutar repertorios de instrucciones ideados hace décadas (por ejemplo, el x86 es del año 1986) y que, hoy en día, se sabe que son ineficientes.

Sin embargo, el mercado de la electrónica de consumo (teléfonos móviles, PDAs, reproductores portátiles MP3 y MP4, reproductores de libros electrónicos, receptores de GPS, etc.) se caracteriza por su baja inercia puesto que en él se comercializan productos que ya contienen el software instalado de fábrica y que ejecutan un reducido número de aplicaciones software. Dado que estos dispositivos no son programables por el usuario final, la necesidad de mantener compatibilidad hacia atrás en nuevas versiones no es nada acuciante. Hay mayor libertad para que el fabricante recurra a procesadores más modernos y con mejores prestaciones pese a que no exista compatibilidad con el software que ya tiene desarrollado. Las perspectivas de producir mejores productores con mejor relación calidad-precio compensan el esfuerzo de la migración software. Por otro lado, por sus características, estos dispositivos necesitan reducir el consumo de energía al máximo lo que les convierten en el nicho de mercado ideal para los procesadores VLIW. Un ejemplo de la baja inercia de este mercado es que la mayor parte de los procesadores que utilizan en sus desarrollos son de tipo RISC, no dejándose arrastrar por la arquitectura IA-32 como la inmensa mayoría de los computadores de propósito general. Algunas familias de procesadores VLIW para sistemas empotrados y el procesamiento digital de señales son la FR-V de Fujitsu, la CEVA-X de CEVA, la Trimedia de NXP Semiconductor, la SHARC ADSP-2136x de Analog Devices, la línea de procesadores DPS TMS320C6x de Texas Instruments y la ST200 de STMicroelectronics. Otro nicho de mercado en donde es posible encontrar diseños VLIW es en el campo de los procesadores gráficos. Un ejemplo de ello son las líneas de procesadores gráficos desarrollados por AMD.

3.19. Referencias

Evans J.S., Trimper G.L. *Itanium architecture for programmers: Understanding 64-bit processors and EPIC principles*, Prentice Hall PTR, Upper Saddle River, New Jersey, 2003.

Fisher J.A. Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Transactions on Computers*, vol. 30, nº 7, pp. 478-490, 1981.

Fisher J.A., Faraboschi P., Young C. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, San Francisco, California, 2004.

Hennessy J.L., Patterson D.A. *Computer Architecture. A Quantitative Approach*, 4ª edición,

Morgan Kaufmann, San Francisco, California, 2007.

Intel *Itanium Processor Microarchitecture Reference*, 2000.

Intel *Itanium Architecture Software Developers Manual*, 2010.

Mahlke S.A., Lin D.C., Chen, W.Y., Hank R.E., Bringmann R.A. Effective Compiler Support for Predicated Execution Using the Hyperblock *Proc. of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, Portland, Oregon, pp. 45-54, 1992.

McNairy C., Soltis D. Itanium 2 processor microarchitecture, *IEEE Micro*, vol. 23, nº 2, pp. 44-55, 2003.

Ortega J., Anguita M., Prieto A. *Arquitectura de computadores*, Thomson Paraninfo. 2005.

Ramakrishna Rau B., Fisher J.A. Instruction-level parallel processing: History, overview, and perspective, *The Journal of Supercomputing*, vol. 7, nº 1, pp. 9-50, 1993.

Schlansker M.S., Ramakrishna Rau B. EPIC: Explicitly Parallel Instruction Computing, *IEEE Computer*, vol. 33, nº 2, pp. 37-45, 2000.

Zomaya A.Y. *Parallel and Distributed Computing Handbook*, McGraw-Hill Professional, New York City, New York, 1996.

3.20. Preguntas de autoevaluación

P3.1 ¿Qué tipo de procesadores aprovechan el paralelismo funcional? ¿Y el de datos?

Los procesadores superescalares y VLIW son claros ejemplos de aprovechamiento del paralelismo funcional basado en la ejecución de múltiples instrucciones en paralelo. Un ejemplo de paralelismo de datos lo representan los procesadores vectoriales.

P3.2 ¿Por qué se caracteriza un procesador VLIW?

Se caracteriza por emitir en cada ciclo de reloj una única instrucción compuesta de varias mini-instrucciones tipo RISC (enteras, flotantes, accesos a memoria, etc.) que se ejecutan en paralelo en la unidades funcionales de que consta el procesador.

P3.3 ¿Quién realiza la planificación de las instrucciones en un procesador VLIW?

Es responsabilidad exclusiva del compilador.

P3.4 ¿Cuántas instrucciones emite un procesador VLIW por ciclo de reloj?

Una única instrucción.

P3.5 ¿Qué calificativo recibe la planificación que realiza el compilador de un computador VLIW?

Se denomina planificación estática, en contraposición a la planificación dinámica que se realiza vía hardware en los procesadores superescalares.

P3.6 ¿Qué caracteriza a la planificación estática que realiza el compilador VLIW?

El compilador debe generar un código objeto VLIW en el que las dependencias de datos y de memoria no sean violadas y se pueda emitir una instrucción por ciclo de reloj sin necesidad de detener la segmentación.

P3.7 ¿Qué implica detener una unidad funcional en un procesador VLIW?

Implica la detención de todas las unidades funcionales y de la segmentación. El código VLIW se genera teniendo en cuenta las dependencias de datos y de memoria existentes por lo que frenar una unidad funcional y dejar avanzar a las restantes puede provocar que no se respete la semántica del programa.

P3.8 ¿Por qué dos razones han fracasado los procesadores VLIW en el ámbito de los computadores de propósito general?

Por la incapacidad para desarrollar compiladores que aprovechen al máximo las características de este estilo arquitectónico y por los problemas de compatibilidad entre procesadores.

P3.9 ¿Por qué el tamaño del código objeto generado para un procesador VLIW es mayor que el generado para un procesador superescalar?

Se debe a que el compilador VLIW no siempre es capaz de rellenar todas las operaciones de que consta una instrucción VLIW con instrucciones originales del código fuente.

P3.10 ¿Qué inconvenientes ocasiona la existencia de operaciones NOP en las instrucciones VLIW?

Espacio desaprovechado en el almacenamiento del programa, uso inadecuado de la I-caché, unidades funcionales ociosas y un aumento innecesario del tráfico en los buses.

P3.11 ¿Por qué hay problemas de compatibilidad entre diferentes generaciones de un procesador VLIW?

Es debido a que el compilador genera código objeto teniendo en consideración las características específicas de un procesador en cuanto a número de unidades funcionales y latencias. Por ejemplo, el mismo código objeto generado para un procesador A no será válido en un procesador B, si este último cuenta con unidades funcionales con mayor latencia.

P3.12 ¿Qué es la inercia del repertorio de instrucciones?

Es la actitud negativa o reticente para migrar las aplicaciones software disponibles para una arquitectura dada a otro tipo de computadores.

P3.13 ¿Qué provoca la inercia del repertorio de instrucciones?

Provoca el tener que seguir diseñando procesadores que sean capaces de mantener la compatibilidad hacia atrás.

P3.14 ¿En qué nichos de mercado se siguen utilizando procesadores VLIW?

En el mercado de los procesadores embebidos orientados a la electrónica de consumo y en el de los procesadores gráficos.

P3.15 ¿Qué elementos característicos de un procesador superescalar no están presentes en un procesador VLIW?

La lógica de distribución, emisión y reordenación, es decir, aquellos elementos que posibilitan la planificación dinámica de las instrucciones.

P3.16 Las instrucciones VLIW, ¿siguen la filosofía RISC o CISC?

La RISC.

P3.17 ¿Qué implica una instrucción VLIW con menos campos de operación que unidades funcionales haya disponibles en el procesador?

Implica limitar el rendimiento que se puede extraer ya que no se aprovecha todo el paralelismo que proporciona el procesador.

P3.18 ¿Qué relación tiene que existir entre el formato de una instrucción VLIW y las unidades funcionales para que la lógica de decodificación se simplifique más?

Que el número y tipo de las operaciones que se pueden situar en una instrucción VLIW coincidan con el número y tipo de las unidades funcionales.

P3.19 ¿Qué es una memoria caché bloqueante?

Es una memoria caché que tiene la capacidad de poder detener todas las unidades funcionales ante la aparición de un fallo.

P3.20 ¿Por qué son necesarias memorias cachés bloqueantes en un procesador VLIW?

El compilador no puede predecir la aparición de fallos de lectura/escritura en la caché de datos por lo que no puede generar una planificación que oculte las latencias derivadas de estos fallos.

P3.21 ¿Qué es un bloque básico?

Un bloque básico es un conjunto de instrucciones que forman una línea de ejecución secuencial por lo que en su interior no existen instrucciones de saltos con la salvedad de la última.

P3.22 ¿Qué es un diagrama de flujo de control?

Es el diagrama que se obtiene de interconectar las entradas y salidas de los diferentes bloques básicos que existen en un programa.

P3.23 ¿Qué es un grafo de flujo de datos?

Es un grafo dirigido en el que los nodos son las instrucciones de un bloque básico, y los arcos se inician en una instrucción de escritura en un registro (instrucción productora) y tienen como destino una instrucción que lee el valor de ese registro (instrucción consumidora).

P3.24 ¿Qué es la planificación local?

Es la combinación de instrucciones procedentes de un único bloque básico para producir instrucciones VLIW.

P3.25 Indique dos técnicas de planificación local.

El desenrollamiento de bucles y la segmentación software.

P3.26 ¿Qué es la planificación global?

Es la combinación de instrucciones de diferentes bloques básicos para producir instrucciones VLIW con mayor grado de paralelismo. Al poder extraer instrucciones de diferentes bloques es posible combinarlas de forma que se minimicen los huecos en las instrucciones VLIW.

P3.27 Indique tres técnicas de planificación global.

La planificación de trazas, la planificación de superbloques y la planificación de hiperbloques.

P3.28 ¿Qué técnica permite aprovechar el paralelismo existente en el cuerpo de un bucle replicando el contenido del mismo?

El desenrollamiento de bucles.

P3.29 ¿Qué tres ventajas aporta el desenrollamiento de bucles?

Se reduce el número de iteraciones del bucle, lo que implica una reducción de las instrucciones de salto, el total de instrucciones ejecutadas es menor ya que se eliminan instrucciones de salto y de incremento/decremento de los índices y se posibilita el mejorar la planificación de las instrucciones pues al desenrollar el bucle queda mucho más cálculo al descubierto al incrementarse el tamaño de los fragmentos de código lineal.

P3.30 ¿Qué es la segmentación software?

Es una técnica de planificación local consistente en producir un nuevo cuerpo del bucle compuesto por la intercalación de instrucciones extraídas de diferentes iteraciones del bucle original.

P3.31 ¿Qué instrucciones del bucle original constituyen el prólogo y el epílogo cuando se aplica segmentación software?

El prólogo y el epílogo están constituidos por las instrucciones que son necesarias para completar las iteraciones afectadas por las instrucciones que conforman el patrón de ejecución o núcleo.

P3.32 ¿Con qué otro nombre se conoce a la segmentación software?

Planificación policíclica.

P3.33 ¿Qué es una traza?

Se trata de un conjunto de bloques básicos que forman una secuencia de código sin bucles. En realidad, representa el camino de ejecución más probable ante una bifurcación.

P3.34 ¿Qué es la planificación de trazas?

Es una técnica de planificación global que permite tratar una secuencia de bloques básicos como si fuese uno único y, a partir de ese nuevo bloque, dotado de un mayor número de operaciones, producir código VLIW más óptimo mediante una adecuada planificación de las operaciones.

P3.35 ¿Cuál es la diferencia entre la planificación de trazas y la planificación de hiperbloques?

La diferencia está en las estructuras de bloques básicos que pueden manejar. La planificación de trazas se aplica a secuencias de bloques básicos donde cualquier bloque puede tener varias entradas y salidas. La planificación de hiperbloques se orienta hacia secuencias de bloques básicos con una única entrada pero cuyos bloques intermedios pueden presentar múltiples salidas.

P3.36 ¿De qué dos pasos consta la planificación de trazas?

De la selección de la traza y de la compactación o compresión de la traza.

P3.37 ¿Qué es la selección de la traza?

Es la búsqueda de un conjunto de bloques básicos para formar una secuencia de código sin bucles, es decir, una traza.

P3.38 ¿Cómo se realiza la selección de una traza?

La selección de los bloques básicos que forman una traza se realiza especulando sobre cuáles son las rutas o caminos de ejecución considerados como más probables.

P3.39 ¿Cómo puede conocer el compilador cuales son las rutas de ejecución más probables?

Mediante un grafo de flujo de control con pesos o ponderado que es similar al grafo de flujo de control pero asignando a cada bloque básico una etiqueta o peso que indica la probabilidad o frecuencia de ejecución. La obtención de los pesos se puede realizar de diferentes formas: perfiles de ejecución del programa, estimaciones software, planificación estática de saltos, etc.

P3.40 ¿Por qué es necesario colocar código de compensación cuando se realiza planificación de trazas?

Es necesario para deshacer los efectos producidos al ejecutar incorrectamente las operaciones que han sido desplazadas al considerar que su ejecución es altamente probable. El código de compensación se coloca en la ruta de ejecución que se considera que tiene menos probabilidad de ser ejecutada y elimina (compensa) los efectos de las instrucciones que se han ejecutado de forma especulativa.

P3.41 ¿Cuál es la expresión genérica que determina cuándo el código planificado es mejor que la secuencia de código original al desplazar parte de las operaciones que componen una de las ramas de una bifurcación condicional?

La expresión genérica es $a * p + b * (1 - p) < c * p + d * (1 - p)$ donde p representa la frecuencia de ejecución de la rama más probable, a los ciclos de ejecución de la rama más probable de la secuencia planificada, b los ciclos de ejecución de la rama menos probable de la secuencia planificada, c los ciclos de ejecución de la rama más probable de la secuencia original y d los ciclos de ejecución de la rama menos probable de la secuencia original.

P3.42 ¿Cuál es uno de los problemas que presenta la planificación de trazas?

A mayor cantidad de operaciones desplazadas, mayor es la penalización en que se incurre cuando se realiza una predicción errónea.

P3.43 ¿Cuándo la planificación de trazas proporciona mejores resultados?

Cuando la traza seleccionada es correcta, es decir, cuando se cumple la predicción que se ha realizado sobre cuál es la ruta de ejecución más probable.

P3.44 ¿Qué es una operación con predicado?

Es una instrucción en la que su resultado se almacena o se descarta dependiendo del valor de un operando que tiene asociado. Este operando recibe el nombre de predicado.

P3.45 ¿Cuál es la finalidad de las operaciones con predicado?

El permitir al compilador reducir el número de saltos condicionales que hay en el código de forma que un diagrama de flujo de control compuesto por diferentes ramas con pequeños bloques básicos pueda transformarse en un único bloque extendido y aplicarle técnicas de planificación local.

P3.46 ¿En qué consiste la técnica *if-conversion*?

Al proceso de eliminar los saltos condicionales de un programa y reemplazarlos por instrucciones con predicados.

P3.47 ¿Cómo se implementa un predicado en una instrucción VLIW?

Mediante un nuevo operando de lectura en cada una de las operaciones que conforman una instrucción VLIW.

P3.48 ¿De qué depende el tamaño en bits que tiene el identificador de un predicado?

Del número de entradas del fichero de registros de predicados de que conste el procesador.

P3.49 ¿Cuál es la sintaxis para representar una instrucción con predicado?

Por lo general, colocado delante o detrás de la instrucción el identificador del predicado entre paréntesis. Por ejemplo, instrucción (p) o (p) instrucción.

P3.50 ¿Un predicado está sometido a dependencias de datos?

Sí ya que se trata de un operando de lectura o de escritura.

P3.51 ¿Qué sucede cuando las operaciones condicionadas no cumplen su condición de guarda?

Se ejecutan como cualquier otra pero el resultado no se almacena en ningún registro por lo que vienen a equivaler a instrucciones NOP.

P3.52 ¿Cuándo es muy efectiva la aplicación de la técnica *if-conversion*?

Cuando todas las rutas de ejecución que hay en una región de código tienen aproximadamente el mismo tamaño en número de instrucciones y la misma frecuencia de ejecución.

P3.53 ¿Cómo se garantiza el tratamiento correcto de las excepciones en un procesador VLIW?

Mediante un buffer similar al de terminación en el que las instrucciones especuladas se mantienen marcadas hasta que se llega al punto del programa en que se considera que ya no son instrucciones especulativas. Mientras permanezcan marcadas en el buffer, sus resultados no se almacenan de forma definitiva, sino temporal.

P3.54 ¿Qué es un centinela?

Es un fragmento de código que indica que una operación o varias operaciones ejecutadas de forma especulativa han dejado ya de serlo.

P3.55 ¿Dónde se ubica un centinela?

Se ubica en la posición del código en que se encontraba originalmente la o las instrucciones especuladas.

P3.56 ¿Por qué surge el concepto EPIC?

Para resolver algunos de los problemas que plantea el enfoque VLIW puro como, por ejemplo, los problemas de compatibilidad de código por una dependencia excesiva de las características del procesador.

P3.57 ¿Qué relación hay entre EPIC e Itanium? ¿Son sinónimos?

EPIC representa un conjunto de principios arquitectónicos en base a los que diseñar diferentes arquitecturas y procesadores. Al igual que existen muchas arquitecturas RISC y diferentes implementaciones de una misma arquitectura RISC, pueden existir múltiples arquitecturas EPIC (por ejemplo, IA-64) y de cada arquitectura desarrollar diferentes implementaciones (Itanium, Itanium-2).

P3.58 ¿Cuál es el principal diferencia entre el trabajo que realiza un compilador para un procesador VLIW y para un EPIC?

En VLIW, el compilador planifica y agrupa código considerando, básicamente, las características del hardware. En EPIC, el compilador agrupa las instrucciones pero comunica en el propio código qué grupos de instrucciones se pueden ejecutar en paralelo y cuáles no.

P3.59 ¿Qué son los procesadores vectoriales?

Son procesadores diseñados según una arquitectura que permite el procesamiento de vectores (arrays unidimensionales). Están en contraposición a las arquitecturas escalares en que se basan los procesadores superescalares o VLIW.

P3.60 ¿Qué tipos de operandos maneja un procesador vectorial?

Tanto operandos escalares como operandos vectoriales.

P3.61 ¿Por qué los procesadores vectoriales pueden utilizar unidades funcionales vectoriales con segmentaciones muy profundas?

Debido a que no hay que preocuparse por la existencia de dependencias de datos entre los elementos que forman los operandos de una operación vectorial. Una operación vectorial implica procesar de forma independiente pero continua cada uno de los elementos de que constan los operandos fuente vectoriales.

P3.62 ¿Qué dos consecuencias tiene que una única instrucción vectorial equivalga a un bucle escalar completo?

Primera, el ancho de banda de instrucciones es menor ya que el tamaño del código es más reducido y, segunda, no existen riesgos de control al eliminarse las instrucciones de salto condicional que conlleva todo bucle.

P3.63 ¿Qué tres factores determinan el que un programa se pueda aprovechar de las características de un procesador vectorial?

La estructura de flujo de datos del programa, la estructura de flujo de control del programa y la capacidad del compilador para detectar las estructuras de datos y de control susceptibles de ser vectorizadas.

P3.64 ¿De qué unidades de procesamiento consta un procesador vectorial genérico? ¿Cuál es la razón de esta organización?

Consta de una unidad de procesamiento escalar, con características similares a las de un procesador superescalar, y de una unidad de procesamiento vectorial. La razón de ello es que todo programa se compone de una porción de instrucciones escalares y de otra porción de instrucciones vectoriales de forma que cada unidad se ocupa de procesar las instrucciones de su misma naturaleza.

P3.65 ¿En qué tipo de arquitectura se basan los procesadores vectoriales actuales?

En una arquitectura carga-almacenamiento con repertorio de instrucciones tipo RISC.

P3.66 ¿De qué elementos consta la unidad de procesamiento vectorial?

De un fichero de registros vectoriales, de unidades funcionales vectoriales y de una unidad de carga/almacenamiento vectorial. También tiene acceso al fichero de registros de la unidad de procesamiento escalar.

P3.67 ¿Qué parámetros caracterizan al fichero de registros vectoriales?

El número de registros vectoriales, el número de elementos por registro y el tamaño en bytes de los elementos. Otro parámetro fundamental es el número de puertos de lectura/escritura necesarios para poder intercambiar datos con las unidades funcionales vectoriales sin provocar detenciones por riesgos estructurales.

P3.68 ¿Son necesarios mecanismos para detectar la existencia de los riesgos estructurales y de datos que puedan aparecer al ejecutar un programa en un procesador vectorial?

Sí. Puede ocurrir que las instrucciones vectoriales tengan que esperar a que una unidad funcional vectorial quede libre (dependencias estructurales) o a que una unidad escalar o vectorial genere un resultado (dependencias de datos). Lo mismo puede suceder con las instrucciones escalares.

P3.69 ¿Qué significa que una unidad funcional vectorial se organice internamente en varios carriles?

Una unidad funcional vectorial con n carriles o *lanes* implica que, internamente, el hardware necesario para realizar la operación se ha replicado n veces de forma que el número de ciclos que se emplea en procesar un vector se reduce por un factor de n.

P3.70 ¿Qué problema conlleva organizar una unidad funcional vectorial con varios carriles?

La necesidad de tener que incrementar el número de puertos de lectura y escritura de todos los registros vectoriales para poder suministrar elementos a todos los carriles en cada ciclo de reloj.

P3.71 ¿Qué solución se utiliza para resolver el problema que se plantea con los puertos de lectura/escritura al organizar una unidad funcional vectorial con varios carriles?

La solución es especializar los carriles de forma que solo tengan acceso a un subconjunto de elementos de cada registro vectorial.

P3.72 ¿Cuál es la principal característica de la unidad de carga/almacenamiento vectorial?

La unidad de carga/almacenamiento vectorial es capaz de mantener un ancho de banda sostenido de una palabra por ciclo reloj tras la latencia de acceso inicial.

P3.73 ¿Qué procesadores se enmarcan dentro de la categoría SIMD?

Los procesadores vectoriales y los procesadores matriciales.

P3.74 ¿De qué elementos consta un procesador matricial?

Consta de una unidad de procesamiento vectorial, una unidad escalar y una unidad de control que discrimina según el tipo de instrucción.

P3.75 ¿Qué diferencia existe entre la unidad de procesamiento vectorial de un procesador vectorial y de un procesador matricial?

En un procesador vectorial, la unidad vectorial consta de n unidad funcionales especializadas. Sin embargo, en un procesador matricial, la unidad vectorial consta de n elementos de procesamiento o EPs, constituidos por una unidad aritmético-lógica de propósito general o ALU, un conjunto de registros o REP y una memoria local o MEP.

P3.76 ¿Qué diferencia existe en relación al tiempo de procesamiento de n elementos entre un procesador vectorial y un procesador matricial?

El procesador matricial con n elementos de procesamiento puede procesar de forma simultánea en el mismo ciclo de reloj un vector de n elementos. El procesador vectorial necesita n ciclos de reloj para que su unidad funcional consuma los n elementos.

P3.77 ¿Qué función tiene el registro escalar VLR?

El registro VLR controla la longitud de cualquier operación vectorial, ya sean cargas, almacenamientos u operaciones aritméticas.

P3.78 ¿Cómo se denomina la técnica mediante la que se procesan vectores de un tamaño superior a MVL?

El compilador recurre a una técnica denominada *strip mining* y que se traduce como troceado del vector o seccionamiento.

P3.79 ¿Qué es el seccionamiento o *strip mining*?

Consiste en procesar un vector de longitud mayor al valor MVL en secciones de longitud igual o inferior a MVL. Para realizarlo, el compilador genera un bucle con instrucciones escalares y vectoriales que se ocupa de procesar el vector original en varias secciones de longitud MVL y una de longitud menor si el tamaño del vector no es múltiplo de MVL.

P3.80 ¿Cómo gestionan los procesadores vectoriales la carga y el almacenamiento de datos que se encuentran ubicados en memoria de forma no consecutiva pero siguiendo un patrón uniforme?

Mediante instrucciones especiales de carga y almacenamiento que permiten indicar la separación entre datos con un operando adicional. Otra forma es utilizar las instrucciones genéricas de carga y almacenamiento en conjunción con un registro de propósito especial que establece la distancia de separación que debe aplicarse en los accesos.

P3.81 ¿Cómo se soluciona el problema de vectorizar un bucle en cuyo cuerpo hay instrucciones que se ejecutan condicionalmente?

Se recurre a una máscara, almacenada en un registro especial `VM`, que establece qué elementos de un vector tienen que ser manipulados por la unidad funcional al realizarse la operación.

P3.82 ¿Qué representa cada uno de los bits que componen el registro `VM`?

Cada bit del registro `VM` representa a un elemento del vector y especifica si se hacen efectivas las siguientes operaciones sobre él o no.

P3.83 ¿Por qué en el procesador vectorial genérico el registro `VM` es de 8 bytes de longitud?

8 bytes de longitud son 64 bits, que es el número de elementos de un registro vectorial.

P3.84 ¿Qué inconvenientes pueden surgir según cómo se haya implementado el control de operaciones vectoriales mediante máscara?

Puede suceder que el tiempo de procesamiento no se reduzca pese a que no se realicen las operaciones, es decir, que las no operaciones indicadas por la máscara se reemplacen por operaciones tipo NOP. También puede ocurrir que el enmascaramiento afecte únicamente al almacenamiento del resultado pero no evite la operación con lo que podrían surgir excepciones.

P3.85 ¿Qué factores determinan el tiempo de ejecución de una instrucción vectorial?

La latencia en producir el primer resultado o tiempo de arranque, el número de elementos n a procesar por la unidad funcional y el tiempo que se tarda en generar cada resultado o tiempo por elemento.

P3.86 ¿Cuál es la expresión matemática que determina el número de ciclos que se tarda en completar una instrucción vectorial?

$$T_n = T_{arranque} + n * T_{elemento}$$

P3.87 ¿Qué hay que tener en cuenta cuando se trata de calcular el tiempo de ejecución de una secuencia de instrucciones vectoriales?

Si las instrucciones vectoriales afectan o no a diferentes unidades funcionales (riesgos estructurales) y si existen dependencias verdaderas entre ellas, es decir, si son operaciones independientes.

P3.88 ¿Qué es un convoy o paquete de instrucciones vectoriales?

Es un conjunto de instrucciones vectoriales que se pueden emitir simultáneamente debido a que no existen dependencias verdaderas entre ellas ni riesgos estructurales.

P3.89 ¿Cómo se expresa el rendimiento de un procesador vectorial al ejecutar un programa?

Por el tiempo de ejecución que se ha consumido y por el número de operaciones en coma flotante realizadas por unidad de tiempo. Ambas medidas pueden expresarse en ciclos, aunque lo correcto es darlas en segundos.

P3.90 ¿Cómo se denomina la técnica que se emplea en los procesadores vectoriales y que consiste en reenviar el resultado de una unidad funcional a otra sin esperar a que se complete la operación sobre un registro vectorial?

Encadenamiento de resultados o *chaining*.

P3.91 ¿En qué afecta el adelantamiento a la formación de convoyes de instrucciones?

Si se permite adelantamiento es posible que varias instrucciones con dependencias verdaderas de datos formen parte del mismo convoy.

P3.92 ¿Qué tipo de riesgo no se permite que exista entre las instrucciones que forman un convoy?

Los riesgos estructurales.

P3.93 ¿Qué dos consecuencias positiva y negativa tiene permitir la ejecución solapada de convoyes?

La consecuencia negativa es que se complica la lógica de emisión de instrucciones. La consecuencia positiva es que se pueden llegar a ocultar los tiempos de arranque de todos los convoyes con la excepción del primero.

P3.94 ¿Qué representa el tiempo de arranque en una unidad de carga/almacenamiento vectorial?

Se corresponde con la latencia de acceso a la memoria.

P3.95 ¿Qué representa el tiempo por elemento en una unidad de carga/almacenamiento vectorial?

Es el tiempo que se emplea en transferir un elemento desde un banco de memoria hasta un registro vectorial o viceversa. Suele ser inferior a un ciclo de reloj aunque se iguala a un ciclo.

P3.96 ¿Cómo se consigue en un procesador vectorial que las latencias de acceso a memoria de los elementos de un vector de longitud n queden ocultas de forma que solo sea visible la latencia de acceso al primer elemento?

Por un lado, distribuyendo los n elementos de un vector entre m bancos de memoria para que se solapen los $(n-1) * T_a$ ciclos de acceso de $n-1$ elementos del vector con los $n * T_{elemento}$ ciclos que se emplean en enviar los n elementos a un registro vectorial. Por otro lado, realizando un dimensionamiento correcto del número de bancos de memoria de forma que sean una potencia de dos mayor o igual que el tiempo de acceso expresado en ciclos.

P3.97 ¿Qué dos formas existen para realizar el solapamiento de las latencias de acceso a memoria de los n elementos de un vector?

De forma síncrona y de forma asíncrona.

P3.98 ¿En qué consiste el acceso síncrono a los bancos de memoria de un procesador vectorial?

En solicitar simultáneamente un dato a los m bancos cada T_a ciclos. Así, cada T_a ciclos se realizan dos acciones: se efectúa una nueva petición simultánea a todos los bancos para extraer los m elementos siguientes y se comienza a transferir ciclo a ciclo los m elementos ya disponibles y que fueron solicitados en la petición anterior.

P3.99 ¿En qué consiste el acceso asíncrono a los bancos de memoria de un procesador vectorial?

Implica solicitar los elementos de que consta el vector a cada uno de los m bancos de forma periódica con periodo T_a y con un desfase entre bancos consecutivos de $T_{elemento}$ ciclos. En el momento en que un banco tiene ya el dato disponible realiza dos acciones: efectúa una nueva solicitud de dato y transfiere el dato ya disponible al registro vectorial.

P3.100 ¿Qué factores determinan el número de bancos de memoria que son necesarios en un procesadores vectorial?

El número total de bancos de memoria siempre tiene que ser una potencia de dos igual o superior al tiempo de acceso expresado en ciclos.

P3.101 ¿Cómo se conoce el banco de memoria en que se almacena una dirección?

El número de banco está determinado por los $\log_2(m)$ bits de orden inferior de la dirección de memoria una vez que se ignoran los bits correspondientes a la longitud de un dato expresada en bytes. Si la separación entre datos es por palabras de k bytes, los $\log_2(k)$ bits de orden inferior son ignorados, y los siguientes $\log_2(m)$ especifican el banco de memoria.

P3.102 ¿Qué representa el T_{base} en el cálculo del rendimiento de un bucle vectorizado?

Es el tiempo que consumen las instrucciones escalares de preparación antes de abordar el bucle que secciona el vector en secciones de longitud igual o inferior a MVL.

P3.103 ¿Qué representa el T_{bucle} al calcular el rendimiento de un bucle vectorizado?

El coste de ejecutar las instrucciones escalares necesarias para realizar el seccionamiento en cada iteración del bucle mediante el que se procesa el vector sección a sección.

P3.104 ¿Qué representa el $T_{arranque}$ en el cálculo del rendimiento de un bucle vectorizado?

Es la suma de los tiempos de arranque visibles de las unidades funcionales que se utilizan en los convoyes de instrucciones que se ejecutan en cada iteración del bucle que secciona el vector.

P3.105 ¿Qué representa el $T_{elemento}$ al calcular el rendimiento de un bucle vectorizado?

El número de convoyes en que se organizan las instrucciones vectoriales que se derivan de vectorizar el bucle.

P3.106 ¿Qué expresión permite determinar el tiempo total de ejecución en ciclos de reloj de un bucle vectorizado que realiza operaciones sobre vectores de longitud n?

$$T_n = T_{base} + \left\lceil \frac{n}{\text{MVL}} \right\rceil * \left(T_{bucle} + T_{arranque}\right) + n * T_{elemento}$$

P3.107 ¿Es posible que $T_{elemento}$ sea inferior al número de convoyes? ¿Cuándo puede suceder?

Sí. Puede sucede cuando se permite el solapamiento de instrucciones vectoriales en las unidades funcionales.

3.21. Actividades

A3.1 Considere un procesador VLIW con un formato de instrucción que permite emitir simultáneamente operaciones a 2 unidades funcionales para el acceso a memoria (2 ciclos de latencia), a 2 unidades funcionales para operaciones en coma flotante (3 ciclos de latencia) y a una unidad funcional para operaciones enteras y de salto (1 ciclo de latencia donde el salto tiene un hueco de retardo de 1 ciclo). Dado el siguiente fragmento de código intermedio:

```
inicio: LD     F2,0(R1)
        MULTD  F2,F2,F0
        LD     F4,0(R2)
        ADDD   F4,F2,F4
        SD     0(R2),F4
        SUBI   R1,R1,#8
        SUBI   R2,R2,#8
        BNEZ   R1,inicio
```

- Desenrolle la secuencia cuatro veces y planifique el bucle desenrollado agrupando las instrucciones por su tipo.
- Planifique el bucle desenrollado en el apartado anterior en forma de instrucciones VLIW teniendo en cuenta las características del procesador para evitar cualquier detención del cauce.

- Si el tamaño de una instrucción de código intermedio es 4 bytes, calcule el tamaño del código VLIW resultante y el espacio de almacenamiento que se desaprovecha.
- Considerando que el bucle original y el desenrollado y planificado se ejecuta sin detenciones, esto es, un ciclo por instrucción, calcule los ciclos consumidos al ejecutar 1000 veces el código original, el desenrollado planificado y el VLIW.

A3.2 Considere el siguiente bucle:

```
inicio: LD      F0,0(R1)
        ADDD    F4,F0,F2
        SD      0(R1),F4
        SUBI    R1,R1,#8
        BNEZ    R1,inicio
```

mediante el que se suma una constante, almacenada en F2, a los elementos de un array.

- Genere el código intermedio aplicando segmentación software pero evite la utilización de desplazamientos negativos en los accesos a memoria del patrón y considere que la latencia de las instrucciones es de 1 ciclo.
- Utilizando el código obtenido en el apartado anterior, escriba el código VLIW teniendo en cuenta que el formato de instrucción admite una operación de carga/almacenamiento (2 ciclos de latencia), una operación de coma flotante (3 ciclos de latencia) y una operación entera/salto (1 ciclo de latencia y el salto consume un hueco de retardo).
- ¿Por qué si se aplica la técnica de segmentación software al bucle original sin considerar las latencias verdaderas no se aprovecha realmente la segmentación hardware de las unidades funcionales?

A3.3 Dado el siguiente fragmento de código:

```
inicio: LD      F2,0(R1)
        MULTD   F4,F2,F10
        LD      F6,0(R2)
        ADDD    F8,F6,F4
        SD      0(R2),F8
        SUBI    R1,R1,#8
        SUBI    R2,R2,#8
        BNEZ    R1,inicio
```

donde F6 y F10 contienen valores previamente cargados. Se pide que:

- Aplique segmentación software y obtenga el patrón de comportamiento considerando que los accesos a memoria consumen dos ciclos y las operaciones de coma flotante tres ciclos.

- Genere el correspondiente pseudocódigo VLIW para un procesador VLIW genérico en el que la unidad de salto consume un ciclo.

A3.4 Un procesador VLIW cuenta con un formato de instrucción que admite una operación de carga/almacenamiento (2 ciclos de latencia), una unidad de coma flotante (3 ciclos de latencia) y unidad para operaciones enteras y de salto (1 ciclos de latencia y una hueco de retardo). Aplique segmentación software al siguiente bucle y genere el correspondiente código VLIW:

```
inicio: LD      F0,0(R1)
        ADDD    F4,F0,F2
        SD      0(R1),F4
        SUBI    R1,R1,#8
        BNEZ    R1,inicio
```

A3.5 Dado el siguiente bucle escrito en pseudocódigo:

```
for (i=0; i<n; i++)
   if (A[i]==0) then
      X[i]:= X[i]+a;
   else
      Y[i]:=Y[i]-a;
   end if;
end for;
```

- Escriba el código intermedio genérico.
- Escriba el código intermedio aplicando la técnica *if-conversion*.
- Escriba el código VLIW de los apartados anteriores teniendo en cuenta que el formato de instrucción admite una operación de carga/almacenamiento (2 ciclos de latencia), una operación de coma flotante (3 ciclos de latencia) y una operación entera/salto (1 ciclo de latencia y el salto consume un hueco de retardo). Compare los tiempos de ejecución para procesar un vector de 1000 elementos suponiendo que la probabilidad de ambas ramas del `if` es la misma.

A3.6 Considerando el bucle del ejercicio anterior, genere código VLIW aplicando predicación y realizando la carga especulativa de los datos.

A3.7 Dado el siguiente fragmento de pseudocódigo:

```
if (a>0) then
   a:=a+a;
else
   if (b<0) then
      b:=b+b;
   else
```

```
        c:=c+c;
      end if;
      a:=b;
    end if;
```

escriba el código intermedio utilizando instrucciones condicionadas con el fin de reducir al mínimo las sentencias de salto condicional. Las direcciones de memoria de las variables enteras a, b y c se encuentran almacenadas en `M[R5]`, `M[R6]` y `M[R7]`, respectivamente.

A3.8 Dado el siguiente fragmento de pseudocódigo:

```
if (a==0) then
   a:=b;
else
   a:=a+1;
end if;
```

y asumiendo que la rama `then` es la que tiene mayor probabilidad de ser ejecutada, genere el código intermedio aplicando planificación de trazas.

A3.9 Dado los dos ejemplos de código de la Figura 3.17, intente reducir el número de ciclos de ejecución mediante una adecuada reorganización de las instrucciones.

A3.10 La velocidad pico o rendimiento máximo de un procesador vectorial se alcanza cuando todas las unidades funcionales aritméticas se encuentran funcionando a pleno rendimiento una vez superadas las etapas de arranque. Por lo general, esta medida se expresa en MFLOPS. Suponiendo que dispone de un procesador vectorial con una unidad de suma vectorial, una unidad de multiplicación y una de división que funcionan a una frecuencia de reloj de 500 MHz, escriba una secuencia de código vectorial que alcance el rendimiento máximo y calcule su valor en MFLOPS.

A3.11 En un procesador vectorial con las siguientes características:

- Longitud vectorial máxima 64 elementos.
- Una unidad de suma vectorial con tiempo de arranque de 6 ciclos.
- Una unidad de multiplicación con tiempo de arranque de 7 ciclos.
- Una unidad de carga/almacenamiento con tiempo de arranque de 12 ciclos.
- La frecuencia de trabajo del procesador es 500 MHz.

se pretende ejecutar el siguiente fragmento de código vectorial:


```
LV      V1,R1
MULTSV  V2,V1,F1
ADDSV   V3,V1,F1
SV      R2,V2
SV      R3,V3
```

Calcule T_{64}, $T_{elemento}$ y R_{64} bajo las siguientes condiciones:

- Sin encadenamiento entre unidades.
- Con encadenamiento entre unidades.
- Con encadenamiento y tres unidades de carga/almacenamiento.

A3.12 En un procesador vectorial con las siguientes características:

- Registros con una longitud vectorial máxima de 64 elementos.
- Una unidad de suma vectorial con tiempo de arranque de 6 ciclos.
- Una unidad de multiplicación con tiempo de arranque de 7 ciclos.
- Una unidad de carga/almacenamiento con tiempo de arranque de 12 ciclos.
- La frecuencia de trabajo del procesador es 500 MHz.
- T_{base} de 10 ciclos y T_{bucle} de 15 ciclos.

se pretende ejecutar el siguiente bucle:

```
for (i=1; i<=n; i++)
   A[i]:=A[i]+B[i];
   B[i]:=a*B[i];
end for;
```

- Escriba el código vectorial que realizaría las operaciones ubicadas en el interior del bucle considerado la posibilidad de encadenamiento de resultados.
- Calcule T_n, T_{1000}, R_{1000} y R_∞.
- Calcule T_n, T_{1000}, R_{1000} y R_∞ pero ahora considerando encadenamientos y dos unidades de carga/almacenamiento.
- ¿Qué ocurre si se considera encadenamientos, dos unidades de carga/almacenamiento y se permite el solapamiento entre convoyes?

A3.13 Dispone de un procesador vectorial en el que el tiempo de acceso a los bancos de memoria es de 6 ciclos y el tiempo por elemento es de 1 ciclo. Los bancos de memoria tienen un ancho de una palabra de 8 bytes y se direccionan por bytes. En base a estos datos,

- ¿Cuál es el mínimo número de bancos de memoria que hay que utilizar para que el tiempo de acceso a la memoria permanezca oculto salvo en lo que respecta al primer acceso?
- Suponiendo que el procesador vectorial cuenta con el número mínimo de bancos de memoria, ¿cuántos ciclos tardará en completarse la carga de un vector de longitud 64 con una separación entre elementos de 20 palabras?
- ¿Qué porcentaje de ancho de banda de memoria se conseguirá con una carga de 64 elementos si se mantiene una separación entre elementos de 1 en comparación con una separación de 20?

A3.14 En un procesador vectorial dotado de una unidad funcional de suma y otra de multiplicación se ejecuta el siguiente programa en C:

```
for (i=0; i<10000; i++) {
   s[i]:=a[i]+b[i];
   t[i]:=a[i]*s[i];
}
```

Las características del procesador vectorial son las siguientes:

- La latencia de la unidad de memoria es de 14 ciclos de reloj.
- El tiempo de arranque de la unidad de suma es de 8 ciclos de reloj.
- El tiempo de arranque de la unidad de multiplicación es de 10 ciclos de reloj.
- $T_{bucle} = 25$.
- T_{base} se considera que no existe.
- Frecuencia de reloj = $800Mhz$.
- MVL = 64.

Se pide que:

- Escriba una secuencia de instrucciones vectoriales que sustituya a la parte interior del bucle del programa C. Considere que las direcciones de `a`, `b`, `s` y `t` están almacenadas respectivamente en `Ra`, `Rb`, `Rs` y `Rt`, respetivamente.
- Calcule el tiempo de ejecución T_{10000} y el rendimiento teórico máximo R_∞ en MFLOPS bajo las siguiente precondiciones:

- Sin encadenamiento.
- Con encadenamiento.
- Con encadenamiento y 2 unidades de carga/almacenamiento.

A3.15 Aplicando la técnica de seccionamiento de bucles, escriba el código escalar y vectorial necesario para realizar la operación vectorial `Y:=Y+X*s`, donde `s` es un escalar en coma flotante y `X` e `Y` son vectores de 150 elementos con una longitud de 8 bytes. Considere que `s`, `X` e `Y` se encuentran almacenados en la direcciones de memoria `M[R1]`, `M[R2]` y `M[R3]`, respectivamente. El direccionamiento de memoria se realiza por bytes y MVL es 64. ¿Qué cambios habría que realizar en el código si la longitud del vector no se conociese en tiempo de compilación sino en tiempo de ejecución?

A3.16 Dado el siguiente bucle:

```
for (i=0; i<256; i++)
   if (i mod 2) then
      B[i]:=B[i]+A[i];
   end if;
end for;
```

genere el correspondiente código vectorial y calcule el tiempo de ejecución del bucle vectorizado considerando que MVL es 64, T_{bucle} es de 10 ciclos y T_{base} de 5 ciclos. El coste de arranque de la unidad de suma vectorial es de 6 ciclos y el de la unidad de carga/almacenamiento de 12 ciclos y ambas se pueden encadenar. Las direcciones de `A` y `B` se encuentran ubicadas en los registros `R1` y `R2`.