Centro Asociado Palma de Mallorca

Ingeniería de Computadores II
Capítulo 3

Tutor: Antonio Rivero Cuesta

Procesadores VLIW y Procesadores Vectoriales

El Concepto Arquitectónico VLIW

Un procesador **VLIW** (***Very Long Instruction Word***) al igual que un procesador superescalar puede emitir y terminar varias operaciones en paralelo.

La diferencia es que es el compilador el encargado de generar el código binario formado por instrucciones que contengan operaciones paralelas.

El hardware se limitará a emitir una instrucción por ciclo de reloj, sin preocuparse de la dependencia de datos, esto implica que el creador del compilador debe saber el paraleleismo que puede proporcionar el hardware.

La diferencia clave entre el enfoque superescalar y el VLIW es cómo se realiza la planificación de las instrucciones.

En una arquitectura superescalar la planificación se realiza vía hardware y se denomina dinámica.

En la VLIW se realiza vía software y se denomina planificación estática. Es estática porque es el compilador el que establece la secuencia paralela de instrucciones con el objeto de no vulnerar la dependencia entre las instrucciones y reducir las detenciones de la segmentación.

En un procesador VLIW se emite una instrucción por ciclo y una detención en una unidad funcional significa detener todas las unidades para mantener la sincronía.

En los VLIW la complejidad del hardware se reduce considerablemente desplazándose hacia el software, esto se traduce en:

- Reducción de transistores.
- Reducción de la energía consumida.
- Reducción del calor disipado.
- Reducción de la inversión económica en horas de ingeniería.

- El desarrollo del compilador se paga una sola vez y no cada vez que se fabrica un chip.
- Se pueden introducir mejoras en el compilador aunque esté en fase de producción.

Pese a todas estas ventajas los procesadores VLIW no triunfaron debido a:

- La incapacidad para desarrollar compiladores que aprovechen al máximo las características de este estilo arquitectónico. VLIW es un código de baja densidad.
- Por los problemas de compatibilidad entre procesadores VLIW.

Arquitectura de un Procesador VLIW Genérico

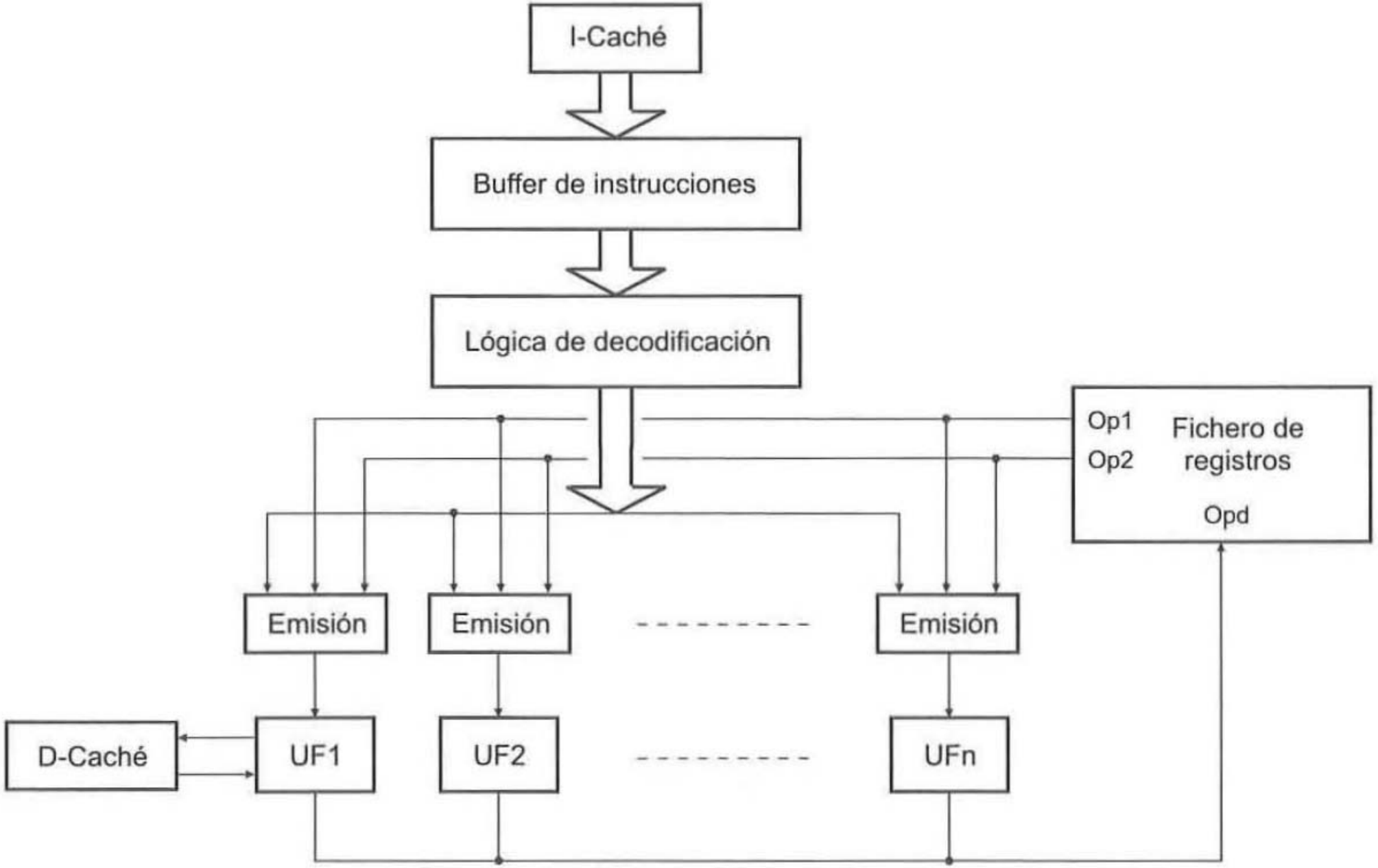


Figura 3.2: Arquitectura básica de un procesador VLIW genérico.

La principal diferencia con respecto a un procesador superescalar es la ausencia de los elementos necesarios para, la distribución, la emisión y el reordenamiento de instrucciones, o lo que es lo mismo su ***planificación dinámica***.

En un procesador VLIW estos elementos no son necesarios ya que es el compilador quien realiza el trabajo de planificación.

Los mecanismos que utilizan los procesadores superescalares para garantizar el flujo de instrucciones son 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 unidades funcionales junto con los valores de los operandos fuente y el identificador del registro destino.

En las ventanas de emisión quedan preparados los códigos de operación y los valores de los operandos fuente para proceder a su emisión simultánea a las unidades funcionales.

El fichero de registros debe de disponer de los suficientes puertos de lectura para suministrar los operandos a todas las unidades funcionales en un único ciclo de reloj.

Análogamente sucede lo mismo con el número de puertos necesarios para la escritura de los resultados en el fichero de registros.

Los repertorios de instrucciones de las arquitecturas VLIW siguen una filosofía RISC, pero con mayor tamaño 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, son implícitamente paralelas.

Las operaciones recogidas dentro de una instrucción VLIW no presentan dependencias de datos, de memoria y/o control entre ellas.

El tamaño más habitual de una instrucción VLIW es de 256 bits.

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 se tienen menos operaciones que unidades funcionales el procesador deja unidades ociosas y no obtiene el máximo rendimiento.

Otra cuestión importante es si las operaciones que forman una instrucción pueden situarse en cualquier posición dentro de la instrucción o tiene posición fija.

Por la ausencia de hardware, los procesadores VLIW no detienen las unidades funcionales en espera de resultados.

No existen interbloqueos por dependencias de datos, ya que el compilador se encarga de evitarlos.

Para esto se insertan operaciones NOP forzando el avance del contenido del cauce de las unidades funcionales, evitando así la detención.

Uno de los problemas de los procesadores VLIW es la necesidad de que el compilador encuentre instrucciones fuente independientes para poder rellenar todas las operaciones de una instrucción VLIW.

El no poder rellenar completamente una instrucción tiene dos consecuencias:

- El código objeto de un procesador VLIW es de mayor tamaño que un superescalar, aunque quede vacío el slot correspondiente a la operación, el espacio que ocupa la instrucción en memoria no se reduce, se coloca como código de operación NOP.
- No se aprovechan al máximo los recursos del procesador ya que se quedan unidades funcionales ociosas.

Predecir qué accesos a memoria producirán fallos de caché es complicado.

Las memorias caché son ***bloqueantes***, esto quiere decir que pueden detener todas las unidades funcionales ante la aparición de un fallo de caché.

Cuanto mayor es el ancho de instrucción VLIW mayor es el número de referencias a memoria llegando a ser inaceptable las detenciones de las unidades funcionales, esto encarece el rendimiento del procesador.

Se han desarrollado unas técnicas para explotar al máximo el paralelismo de un procesador VLIW.

El inconveniente es que aumenta el tamaño del código.

Planificación Estática o Basada en el Compilador

Cuando un compilador VLIW recibe como entrada un código fuente realiza una serie de tareas para optimizar el código objeto que produce como salida para aprovechar al máximo el paralelismo del procesador, produciendo tres elementos:

- Un ***código intermedio***.
- Un ***grafo de flujo de control***.
- Un ***grafo de flujo de datos***.

Código Intermedio

Está formado por instrucciones RISC.

Posee dependencias verdaderas RAW.

Las dependencias de salida WAW y antidependencias WAR se eliminan gracias a que el compilador tiene un número infinito de registros simbólicos.

Al reasignar los registros simbólicos en registros arquitectónicos aparecen dependencias falsas.

El compilador las resuelve al generar el código VLIW planificando las instrucciones que componen el código intermedio.

Lo ideal sería encontrar las suficientes operaciones en el código intermedio para formar instrucciones VLIW y maximizar el rendimiento del procesador.

De esta manera todas las unidades funcionales trabajarían en paralelo.

Evitando paradas en la segmentación por dependencias.

Grafo de Flujo de Control

Para generarlo debemos de conocer los ***bloques básicos*** del código intermedio.

Un ***bloque básico*** se compone de un grupo de instrucciones que forman una línea de ejecución secuencial.

No contiene instrucciones de salto excepto 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 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 distintas entradas y salidas de los diferentes bloques básicos son el diagrama de flujo de control.

Cuando ya conocemos los bloques básicos, las instrucciones de cada bloque se combinan para formar instrucciones VLIW.

Esto lo hacemos con el ***grafo de flujo de datos*** que tiene asociado cada bloque.

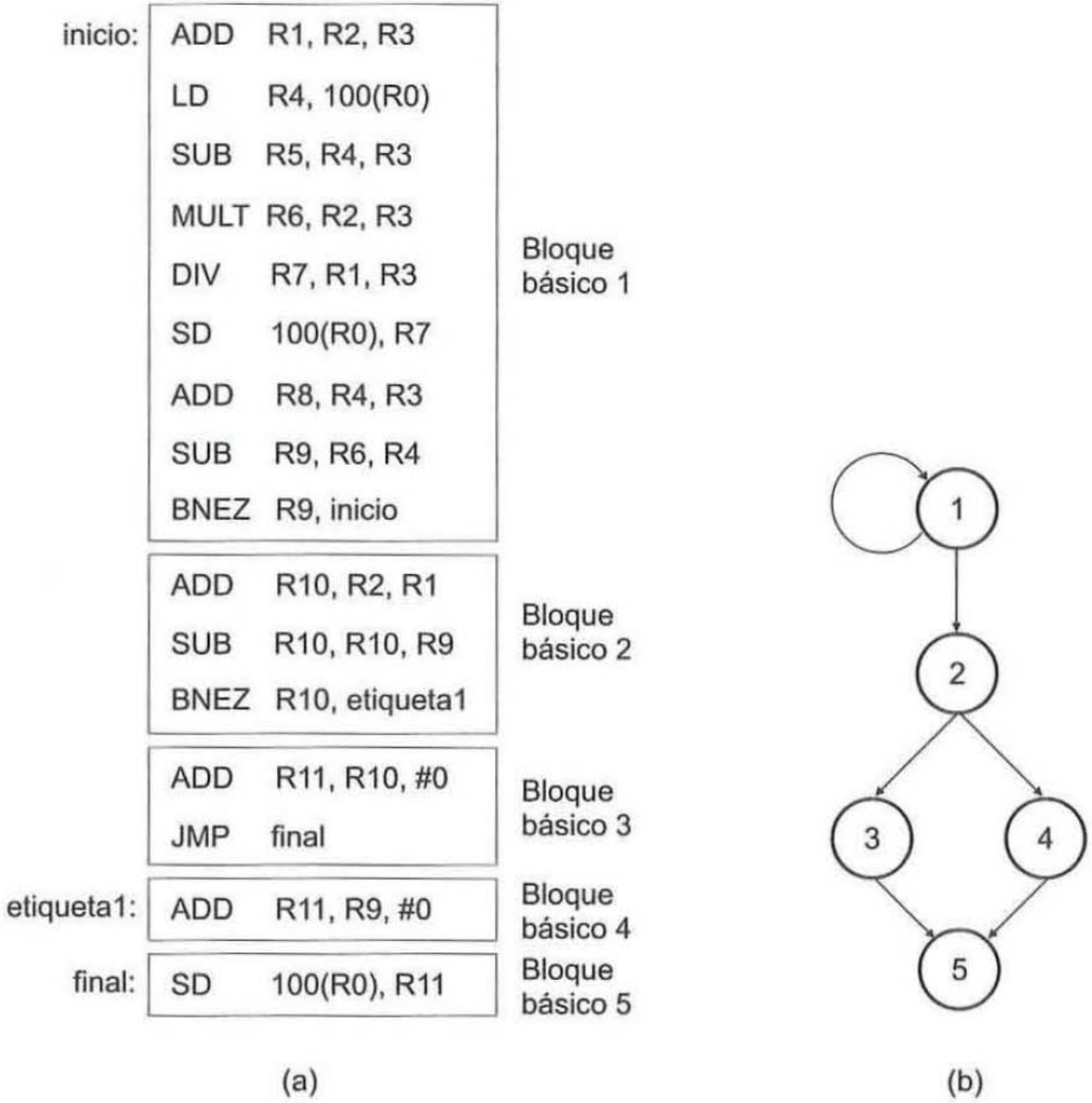


Figura 3.4: Código intermedio y Flujo de Control

Grafo de Flujo de Datos

Es un ***grafo dirigido***.

Los ***nodos*** son las instrucciones de un bloque básico.

Los ***arcos*** se inician en una instrucción de escritura en un registro, ***instrucción productora***, y el destino es una instrucción que lee el valor de ese registro, ***instrucción consumidora***.

El ***grafo de flujo de datos*** nos muestra la secuencia de instrucciones que no presentan dependencias entre ellas y que podemos combinar para formar instrucciones VLIW.

A esta combinación de instrucciones de un bloque básico se le denomina ***planificación local***.

Algunas técnicas de planificación local son:

- El ***desenrollamiento de bucles***.
- La ***segmentación software***.

El paralelismo que se obtiene mediante planificación local está limitado.

Se recurre a la ***planificación global*** con el objeto de combinar instrucciones de diferentes bloques básicos produciendo una planificación con mayor grado de paralelismo.

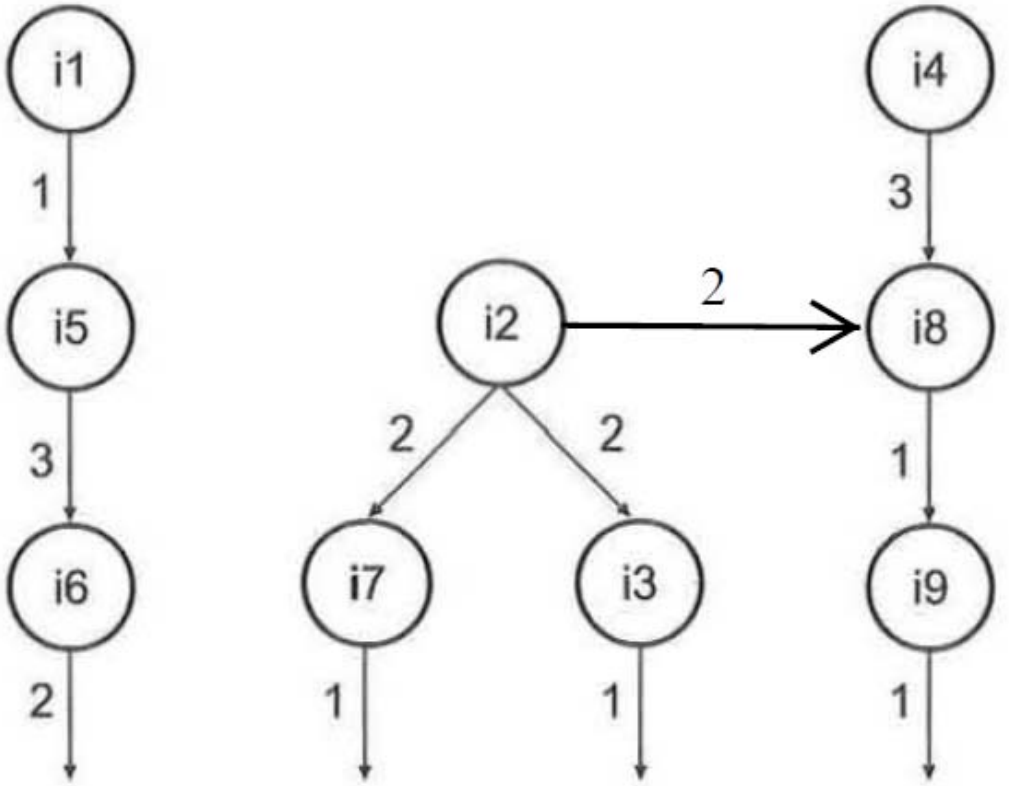
Una técnica es la ***planificación de trazas***.

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.
- Las latencias que presentan.

```
i1: ADD   R1, R2, R3
i2: LD    R4, 100(R0)
i3: SUB   R5, R4, R3
i4: MULT  R6, R2, R3
i5: DIV   R7, R1, R3
i6: SD    100(R0), R7
i7: ADD   R8, R4, R3
i8: SUB   R9, R6, R4
i9: BNEZ  R9, i1
```

(a)



(b)

Figura 3.5: Operaciones Bloque Básico y Diagrama de Flujo de Datos

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

Dos operaciones de suma/resta (un ciclo de latencia).

Una operación de mult/div (tres ciclos de latencia).

Una operación de carga (dos ciclos de latencia)

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.

No es posible ocupar todas las operaciones de una instrucción VLIW por lo que algunas unidades funcionales quedarán ociosas durante la ejecución del código.

Se considera que las instrucciones VLIW tienen una longitud de 20 bytes y que cada operación básica ocupa 4 bytes.

La Figura 3.6 representa la secuencia planificada de instrucciones VLIW generada por el compilador utilizando como información:

- Las latencias de las unidades funcionales.
- Las relaciones productor-consumidor entre instrucciones dadas por el diagrama de flujo de datos.

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.

	1 Ciclo	1 Ciclo	3 Ciclos	2 Ciclos	2 Ciclos
	Operaciones Suma/Resta 1	Operaciones Suma/Resta 2	Operaciones Mult/Div	Operaciones de Carga	Operaciones Almacena.
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),R6

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

Desenrollamiento de Bucles
Loop Unrolling

Es una técnica de planificación local.

Permite aprovechar el paralelismo existente entre las instrucciones del bucle.

Esta técnica replica muchas veces el cuerpo del bucle utilizando diferentes registros en cada réplica y ajusta el código de replicación en función de las veces que se replique el cuerpo del bucle.

Tiene tres ventajas:

- Se reduce el número de iteraciones del bucle, reduciéndose las dependencias de control al ejecutarse menos instrucciones de salto.
- El total de instrucciones ejecutadas es menor ya que se reduce el número de instrucciones de incremento/decremento de los índices que se utilicen en el bucle.
- Se proporciona al compilador más oportunidades para planificar las instrucciones, el compilador realiza una planificación más efectiva y genera código VLIW más compacto.

En cada ciclo de reloj, la lógica de emisión del procesador emite una instrucción del código VLIW hacia las unidades funcionales, el proceso de emisión no se detiene como consecuencia de una dependencia de datos entre instrucciones, salvo por fallos de acceso a caché.

La emisión continua se consigue por la planificación estática que realiza el compilador y asegura que las dependencias se satisfacen al mantener la separación necesaria en ciclos de reloj entre parejas de operaciones que plantean riesgos.

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***, tienen una longitud de 8 bytes.

El índice que permite acceder a los elementos del bucle se almacena en el registro entero Rl, 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,#8     % 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.

El Índice se decrementa de cuatro en cuatro elementos, que son los elementos del vector que se procesan en cada iteración del nuevo bucle desenrolladlo.

El desenrollamiento ha permitido que el número de iteraciones necesarias para recorrer todo el bucle se halla 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.

Que representa un ahorro de seis instrucciones en cada iteración del bucle desenrollado.

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,#8     % i13
        BNEZ R1,inicio    % i14
```

Ahora consideramos que se 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).

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 desenrrollado para el procesador VLIW se recoge en la Figura 3.7.

		2 Ciclos	3 Ciclos	1 Ciclo
	Nº	Operaciones LD/SD	Operaciones Coma Flotante	Operaciones Enteras/Saltos
Inicio:	1	LD F0,0(R1)	------	------
	2	LD F6,-8(R1)	------	------
	3	LD F10,-16(R1)	ADDD F4,F0,F2	------
	4	LD F14,-24(R1)	ADDD F8,F6,F2	SUBI R1,R1,#8
	5	------	ADDD F12,F10,F2	------
	6	SD 0(R1),F4	ADDD F16,F14,F2	------
	7	SD -8(R1),F8	------	------
	8	SD -16(R1),F12	------	------
	9	SD -24(R1),F16	------	BNEZ R1,inicio

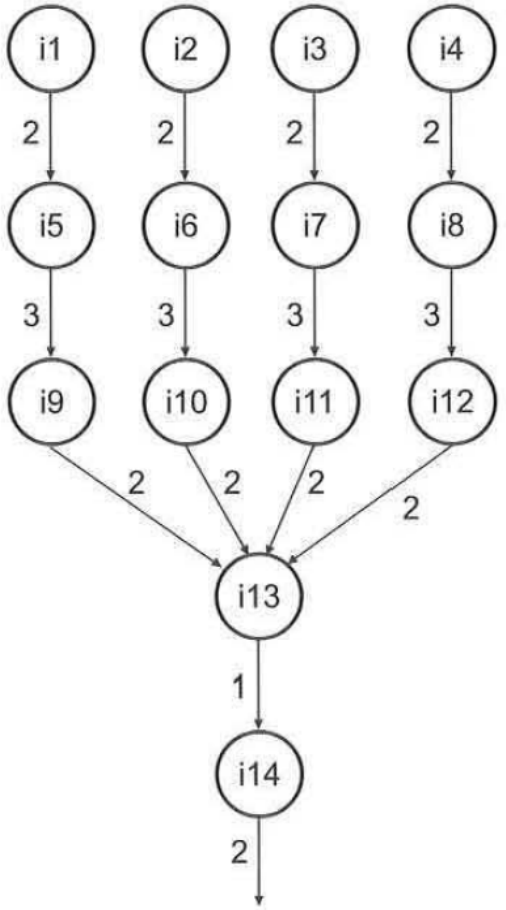


Figura 3.7: Instrucciones VLIW y Diagrama de Flujo de Datos

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: 108 bytes

Las operaciones originales consumen 56 bytes:

En el caso del bucle desenrollado.

14 instrucciones · 4 bytes = 56 bytes.

En el bucle original.

5 instrucciones · 4 bytes: 20 bytes

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:

250 iteraciones · 14 instrucciones: 3500 ciclos.

El VLIW únicamente necesitaría:

250 iteraciones · 9 instrucciones: 2250 ciclos.

En este caso, el enfoque VLIW es un 55 % más rápido que la aproximación escalar desenrollada y planificada.

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.

Es una 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 %.

		2 Ciclos	3 Ciclos	1 Ciclo
	Nº	Operaciones LD/SD	Operaciones Coma Flotante	Operaciones Enteras/Saltos
Inicio:	1	LD F0,0(R1)	------	------
	2	------	------	------
	3	------	ADDD F4,F0,F2	------
	4	------	------	------
	5	------	------	SUBI R1,R1,#8
	6	SD 0(R1),F4	------	BNEZ R1,inicio

Figura 3.8: Instrucciones VLIW Secuencia original

Segmentación Software

Es una técnica que se utiliza para intentar aprovechar al máximo el paralelismo a nivel de instrucción que hay en el cuerpo de algunos bucles.

Esta técnica consiste en producir un nuevo cuerpo del bucle compuesto por la intercalación de instrucciones correspondientes a diferentes iteraciones del bucle original.

Al reorganizar el bucle se añaden unas instrucciones de arranque, ***el prólogo***, antes del cuerpo del bucle y otras de terminación al finalizar el bucle, ***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.

A diferencia del desenrollamiento esta técnica no tiene más instrucciones que el bucle original salvo las que forman el prólogo y el epílogo.

En esta técnica se ejecutan al mismo tiempo instrucciones provenientes de múltiples iteraciones, también se la conoce 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.

Cuando hay instrucciones condicionales en el cuerpo del bucle que impiden la aparición de un patrón de comportamiento regular se complica la aplicación a un procesador VLIW.

Del mismo modo cuando no se conoce a priori el número de iteraciones o cuando el número y tipo de operaciones que forman el patrón de ejecución no se ajusta al formato de la instrucción VLIW.

Tendremos que recurrir a técnicas de planificación que permitan reubicar operaciones de diferentes bloques básicos para intentar ocupar huecos que queden libres en el código VLIW.

Es el objetivo de las técnicas de planificación global, entre las que se encuentran:

- La planificación de hiperbloques.
- La planificación de supebloques.
- La planificación de trazas.

Tras las iteraciones del bucle original aparece un ***patrón*** de ejecución compuesto por instrucciones pertenecientes a 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.

Las instrucciones que hay antes de la aparición del primer patrón forma el ***prólogo*** del nuevo bucle y las que se encuentran tras el último patrón son el ***epílogo***.

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

Vamos a ver un ejemplo:

```
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.

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 Rl aunque sí se ha reflejado el necesario decremento en los desplazamientos de las instrucciones de carga y almacenamiento.

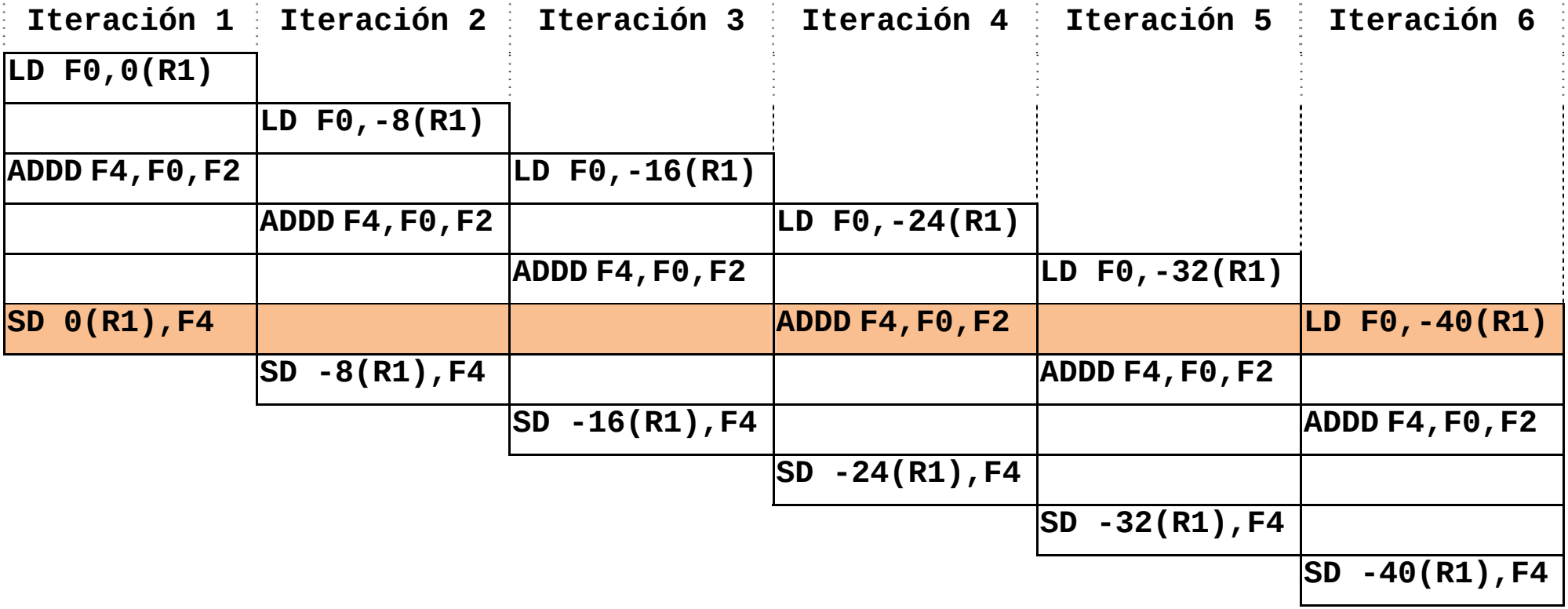


Figura 3.10: Patrón de ejecución obtenido al aplicar la técnica de segmentación software

Además del patrón, se pueden observar las instrucciones que forman el prólogo y el epílogo.

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.

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.

		LD F0,0(R1)		
		LD F0,-8(R1)		
prólogo		LD F0,-16(R1)	ADDD F4,F0,F2	
		LD F0,-24(R1)	ADDD F4,F0,F2	
		LD F0,-32(R1)	ADDD F4,F0,F2	
patron:	inicio:	SD 0(R1),F4	LD F0,-40(R1)	ADDD F4,F0,F2 if (R1<>1){R1:=R1−8; go to inicio}
		SD -8(R1),F4	ADDD F4,F0,F2	
		SD -16(R1),F4	ADDD F4,F0,F2	
epílogo:		SD -24(R1),F4		
		SD -32(R1),F4		
		SD -40(R1),F4		

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

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 con ese nuevo bloque con un mayor número de operaciones producir código VLIW más óptimo mediante una adecuada planificación de las operaciones.

Una ***traza*** es un conjunto de bloques básicos que forman una secuencia de código sin bucles, representan el camino de ejecución más probable ante una bifurcación.

La planificación de trazas es una combinación de dos pasos que se efectúan de forma consecutiva:

- La selección de la traza.
- La compactación o compresión de la traza.

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 cuales serán las rutas o caminos más probables.

Se utiliza un ***grafo de flujo de control con pesos o ponderado***, este lleva una etiqueta que indica la probabilidad o frecuencia de ejecución.

Una traza representa el camino de ejecución más probable.

Compactación de la Traza

Una vez que se ha realizado la selección de la traza el compilador debe proceder a su compactación.

Con el objetivo de:

- Generar código en el que se minimice el número de operaciones vacías en las instrucciones VLIW.
- Se reduzcan los ciclos de ejecución en beneficio de las rutas de ejecución más probables.

El compilador efectúa desplazamientos de las operaciones.

Siendo los saltos los que introducen los mayores impedimentos.

Son los que constituyen puntos de entrada y de salida de la traza.

La reorganización de operaciones dentro de la traza debe de garantizar que se preserva la corrección del programa con independencia de cual sea la ruta más probable.

Cuando se realiza la planificación de trazas colocamos un ***código de compensación*** para deshacer los efectos producidos al ejecutar incorrectamente las operaciones que han sido desplazadas al considerar que su ejecución era altamente probable.

Este código se coloca en la ruta de ejecución con menos probabilidad de ser ejecutada y elimina los efectos de las instrucciones que se han ejecutado de forma especulativa.

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.
- La cantidad de código de compensación que es necesario añadir.
- Saber si compensa el desplazamiento de operaciones dentro de la traza, midiendo el coste tanto en ciclos de ejecución como en espacio de almacenamiento.

Expresión genérica que determina cuándo el código planificado es mejor que la secuencia de código original:

$$a \cdot p + b \cdot (1 - p) < c \cdot p + d \cdot (1 - p)$$

p: frecuencia de ejecución de la rama más probable.

a: ciclos de ejecución de la rama más probable de la secuencia planificada.

b: ciclos de ejecución de la rama menos probable de la secuencia planificada.

c: ciclos de ejecución de la rama más probable de la secuencia original.

d: ciclos de ejecución de la rama menos probable de la secuencia original.

Uno de los problemas que presenta la planificación de trazas es que a mayor cantidad de operaciones desplazadas, mayor es la penalización en que se incurre cuando se realiza una predicción errónea.

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.

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 índice de X en 8 bytes
         SUBI  R3,R3,#8   % Decrementar índice de Y en 8 bytes
         SUBI  R1,R1,#4   % Decrementar índice de A en 4 bytes
         BNEZ  R1,inicio  % Nueva iteración
```

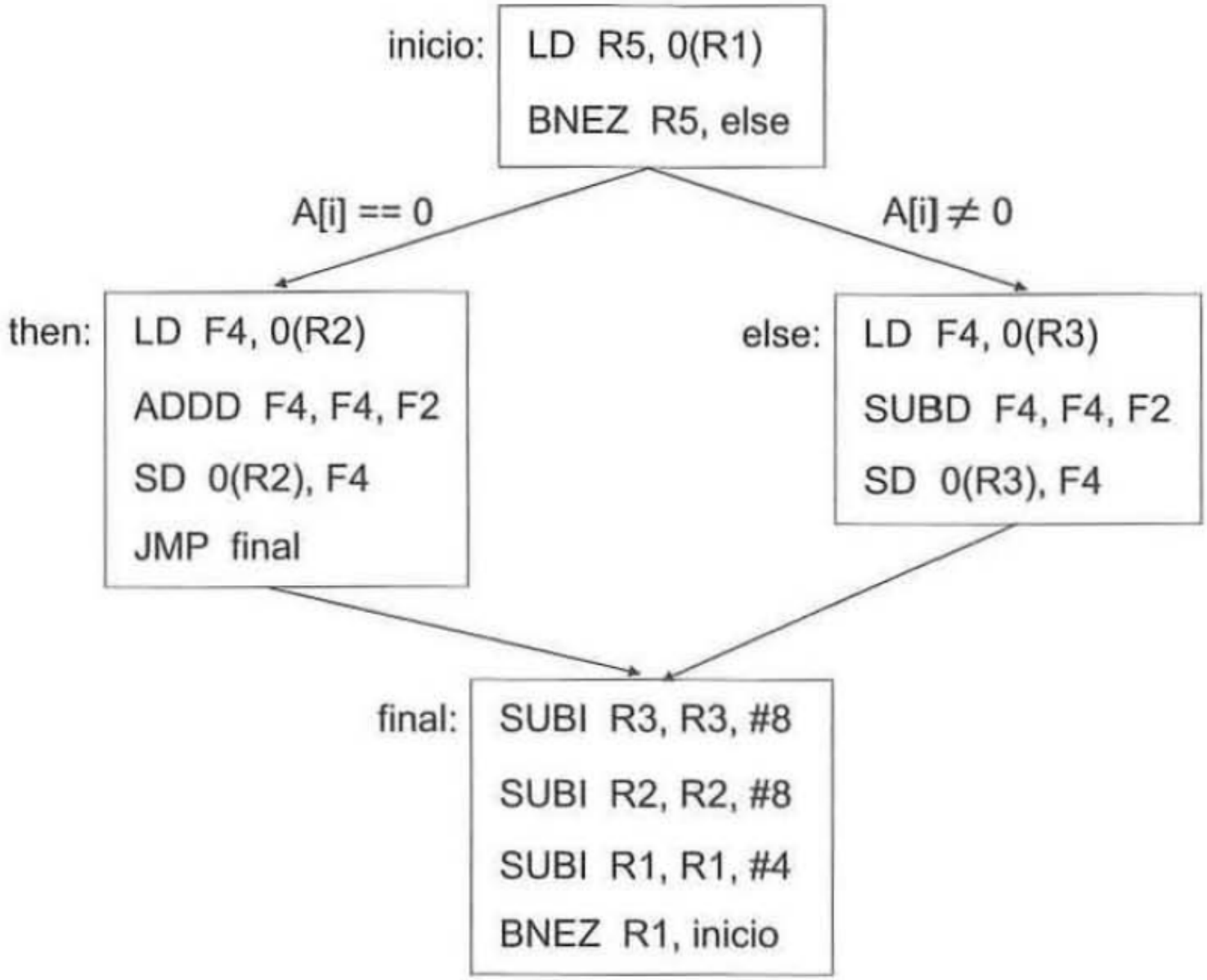


Figura 3.12a: Diagrama de flujo de control

	2 Ciclos	3 Ciclos	1 Ciclo	
	Carga/almacenamiento	Coma flotante	Entera/salto	
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
else:	------	------	------	8
	SD 0(R2),R4	------	JMP final	9
	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

Figura 3.12b: Código VLIW

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 3.13a.

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.

El código de compensación necesario, bloque −B1, sería la expresión contraria:

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 en 8 bytes
         SUBI  R3,R3,#8    % Decrementar índice de Y en 8 bytes
         SUBI  R1,R1,#4    % Decrementar índice de A en 4 bytes
         BNEZ  R1,inicio   % Nueva iteración
```

	Carga/almacenamiento	Coma flotante	Entera/salto	
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(R3), 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

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

Figura 3.15: Código VLIW con planificación óptima

Operaciones con Predicado

Con el objetivo de poder tratar las instrucciones condicionales como cualquier otra instrucción se idearon las ***operaciones con predicado***, también conocidas como ***operaciones condicionadas*** u ***operaciones con guarda***.

El objetivo de las operaciones con predicado es 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 básico extendido y aplicarle técnicas de planificación local.

La técnica ***if-conversión*** es el proceso de eliminar los ***saltos condicionales*** de un programa y reemplazarlos con predicados.

Son instrucciones 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***.

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.

Este registro es la condición que determina el que una operación tenga efecto o no sobre el estado del procesador y de la memoria.

La representación de una instrucción de código intermedio que tiene asociado un predicado es:

Instrucción (p)

Si p vale 1 el resultado de la operación que realice la instrucción se almacenará.

Si p vale 0 no se almacena.

La utilización del segundo predicado, ***p2***, es opcional.

Si se utilizan dos predicados ***p1*** y ***p2***, nunca ambos podrán ser true/true, solo está permitido true/false, false/true y false/false.

Formato para las instrucciones de manipulación de predicados.

```
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

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.

Un predicado es un operando origen o destino que está sujeto a las dependencias de datos al igual que sucede con los operandos fuente 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-fhen*.

La Figura 3.16b corresponde al código intermedio correspondiente al bucle genérico y la Figura 3.16c 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.

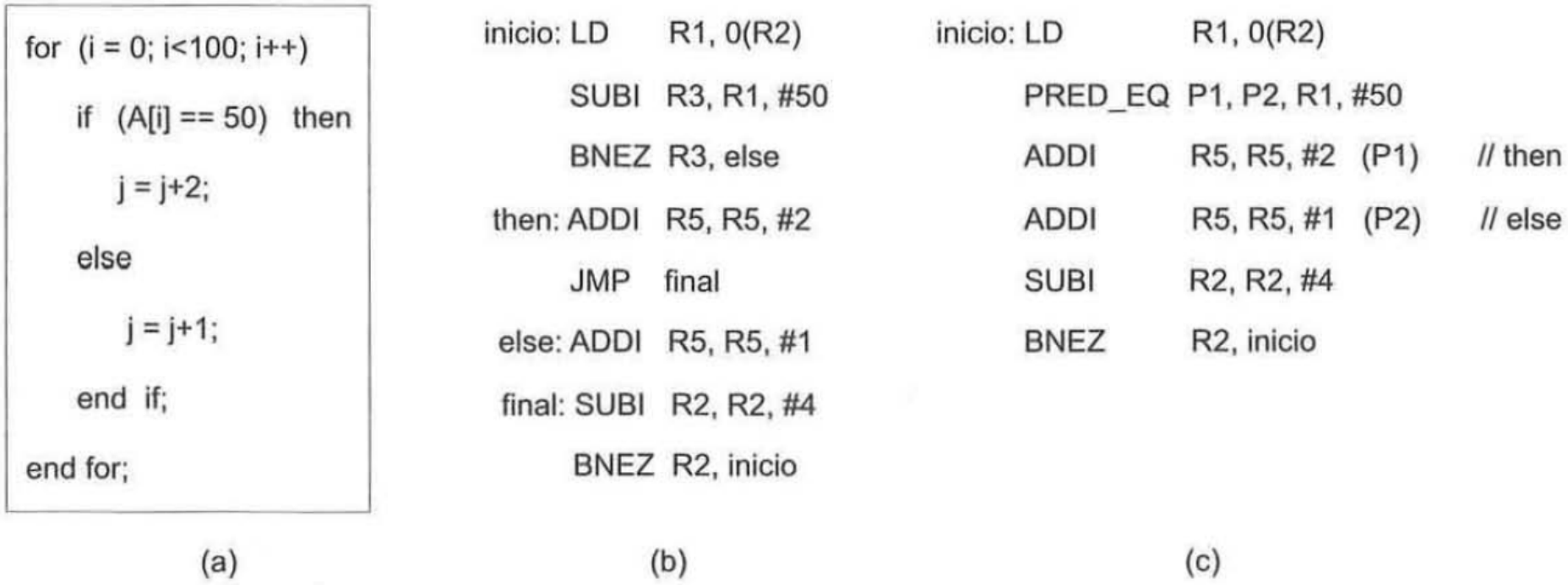


Figura 3.16: Código fuente genérico. Código intermedio. Código con predicado

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.

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.

2 Ciclos	3 Ciclos	1 Ciclo

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

Figura 3.17a: instrucciones VLIW derivadas del código intermedio

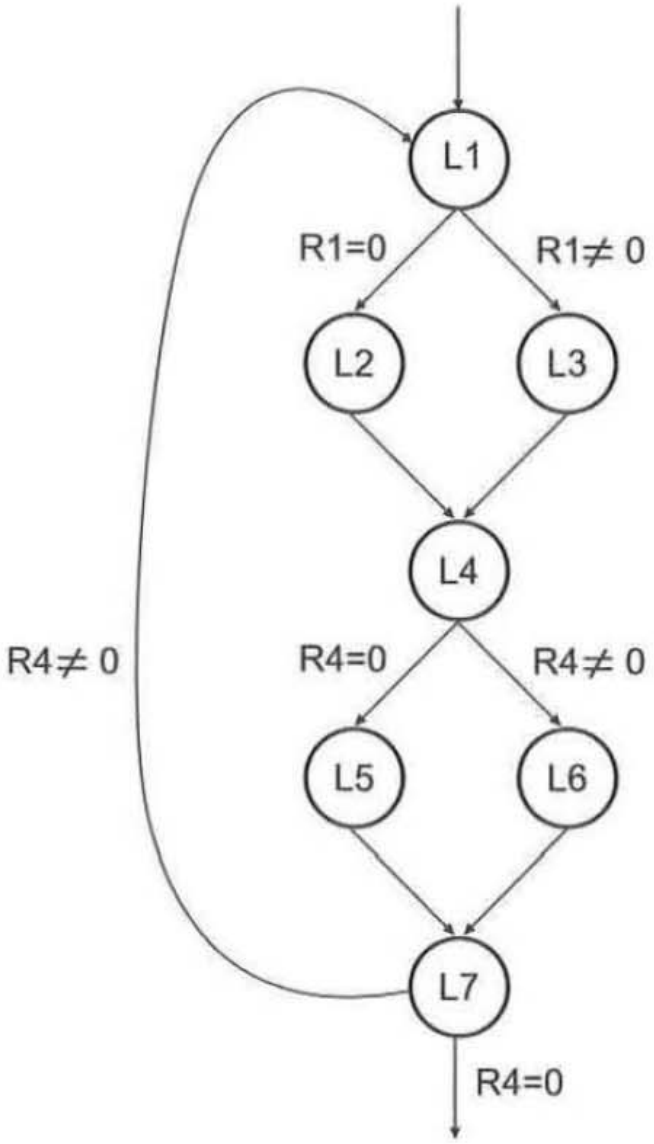
inicio:	**LD R1,0(R2)**	-----	-----	1
	-----	-----	-----	2
	-----	-----	**PRED_EQ P1,P2,R1,#50**	3
	-----	-----	**ADDI R5,R5,#2(P1)**	4
	-----	-----	**ADDI R5,R5,#1(P2)**	5
	-----	-----	**SUBI R2,R2,#4**	6
	-----	-----	**BNEZ R2,inicio**	7
	-----	-----	-----	8

Figura 3.17b: Instrucciones VLIW derivadas del código intermedio con operaciones condicionadas

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 un diagrama de flujo de control perteneciente a un bucle con 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.



(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

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*.

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 finalizar el procesamiento, el resultado se desecha, y resultan equivalentes a las instrucciones NOP.

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, 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.

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.

Tratamiento de Excepciones

Una de las estrategias aplicadas por los procesadores VLIW es la utilización de ***centinelas***.

Es una técnica análoga al ***buffer de reordenamiento*** de los procesadores superescalares.

El ***centinela*** es un fragmento de código que indica que la operación ejecutada de forma especulativa con la que está relacionado ha dejado de serlo.

El ***centinela*** se ubica en la posición del código en el que se encontraba originalmente la instrucción especulada.

La ejecución del código del centinela es lo que determina que los resultados temporales se pueden escribir en sus ubicaciones definitivas, ***fichero de registro o de memoria.***

Para garantizar el tratamiento correcto de las excepciones se utiliza 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 se almacenan de forma temporal.

El Enfoque EPIC

El aumento de ***ILP*** (***Paralelismo implícito a nivel de instrucción***) en procesadores VLIW puro presenta una serie de problemas:

- Los repertorios de instrucciones VLIW no son compatibles entre diferentes implementaciones.
- Las instrucciones de carga presentan comportamientos no determinísticos por la propia naturaleza de sistema de memoria.
- 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.

Para resolver estos problemas surgió la arquitectura **EPIC** (*Explicitly Parallel Instruction Computing)*

EPIC representa una filosofía para construir procesadores que se apoya en ciertas características arquitectónicas.

El principal objetivo de EPIC es retener la planificación estática del código pero introduciendo mejoras con características arquitectónicas que permitan resolver dinámicamente diferentes situaciones como retardos en las cargas o unidades funcionales nuevas o con diferentes latencias.

El enfoque EPIC se sitúa en un punto intermedio entre los procesadores superescalares y los VLIW, el compilador determina el agrupamiento de instrucciones, a la vez comunica de forma explícita en el propio código cómo se ha realizado el agrupamiento.

Las características arquitectónicas más destacadas del estilo EPIC son:

- 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.

En ciencias de la computación, un ***algoritmo determinista*** es un algoritmo que, en términos informales, es completamente predictivo si se conocen sus entradas.

Dicho de otra forma, si se conocen las entradas del algoritmo siempre producirá la misma salida, y la máquina interna pasará por la misma secuencia de estados.

Este tipo de algoritmos ha sido el más estudiado durante la historia y por lo tanto resulta ser el tipo más familiar de los algoritmos.

Procesadores Vectoriales

Son máquinas con una arquitectura que permite la manipulación de vectores.

Las arquitecturas escalares se basan en los:

- Procesadores clásicos.
- Segmentados.
- Superescalares.
- VLIW.

Los procesadores vectoriales cuentan con un repertorio de instrucciones en los que los operadores fuente y destino de las instrucciones son vectores almacenados en registros vectoriales.

Típicas operaciones vectoriales:

- Suma, resta, multiplicación y división de dos vectores
- Suma o multiplicación de todos los elementos de un vector por un escalar.

Existen dos tipos de operandos:

- Los operandos vectoriales.
- Los operandos escalares.

Las operaciones vectoriales pueden tener como fuente dos operandos vectoriales o uno vectorial y otro escalar, pero el resultado será siempre un operando vectorial.

Pueden utilizar unidades funcionales vectoriales con segmentaciones muy profundas sin tener que preocuparse por la existencia de la dependencia de datos, ya que procesan de forma independiente pero continua cada uno de los elementos de que constan los operandos fuente vectoriales.

El compilador genera el código formado por las instrucciones vectoriales a partir de las operaciones que se realizan en el código original.

Una única instrucción vectorial equivale a un bucle completo de instrucciones escalares.

La ventaja de esto es que el ancho de banda de las instrucciones es menor ya que el tamaño del código es más reducido y no hay que extraer tantas instrucciones de la I-caché, por otro lado no existen riesgos de control al eliminarse las instrucciones de salto condicional.

Los factores que determinan que un programa se pueda aprovechar de las características de un procesador vectorial son:

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

Arquitectura Vectorial Básica

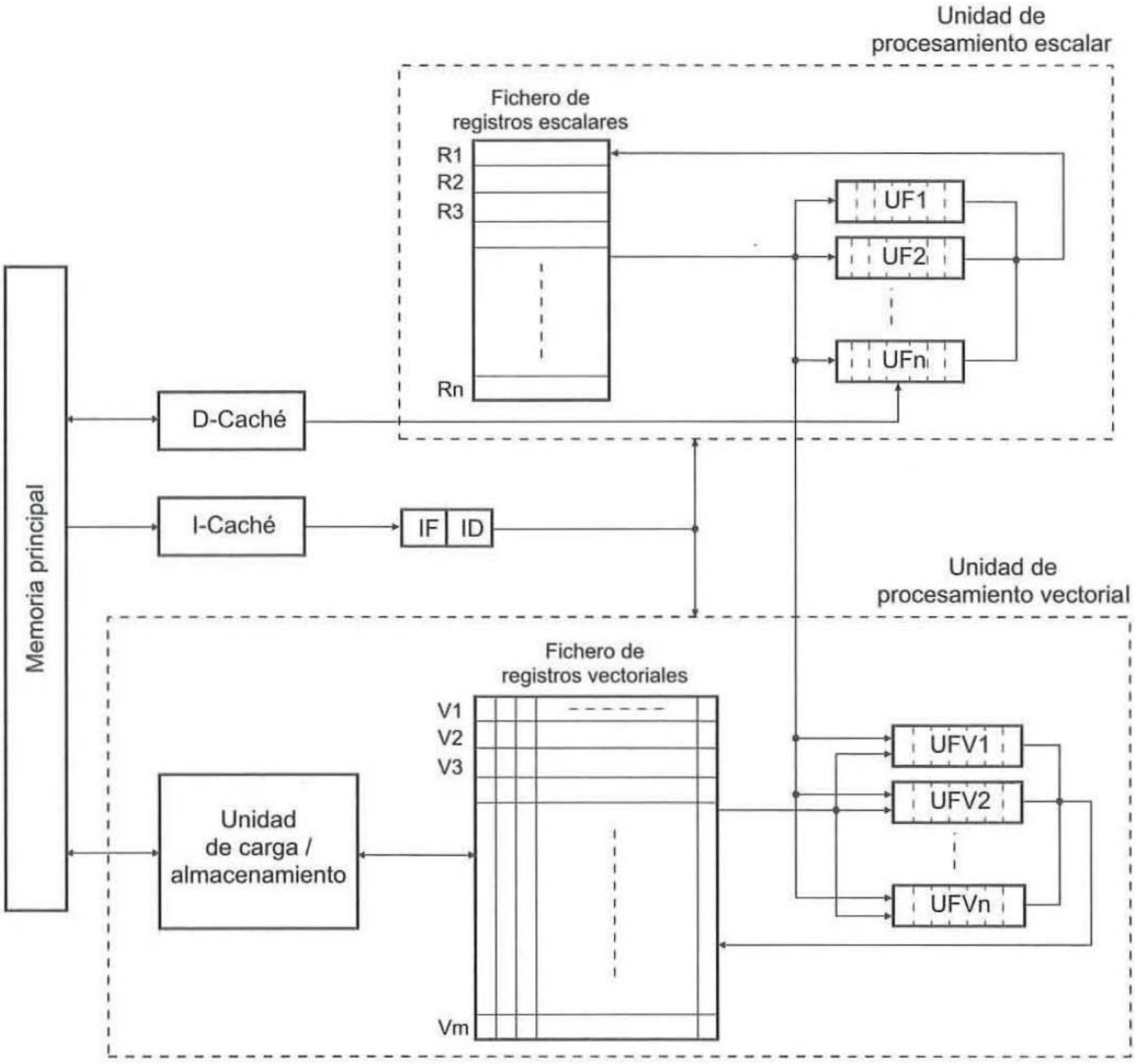


Figura 3.20: Esquema de un procesador vectorial básico.

Un procesador vectorial básico consta de una unidad de procesamiento escalar y de una unidad de procesamiento vectorial.

Todo programa se compone de una parte de instrucciones escalares y de otra parte de instrucciones vectoriales y cada unidad se ocupa de procesar las instrucciones de su misma naturaleza.

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.

La unidad de procesamiento escalar dispone de unidades segmentadas para realizar operaciones enteras, en coma flotante, saltos, cargas/almacenamientos, etc.

La ***unidad de procesamiento vectorial*** tiene un fichero de registros vectoriales, unidades funcionales vectoriales y una unidad de carga almacenamiento vectorial.

También tiene acceso al fichero de registros de la unidad de procesamiento escalar.

El ***fichero de registros vectoriales*** está compuesto por un conjunto de entradas, en donde cada entrada almacena los elementos de que consta un vector.

Este fichero es esencial para los algoritmos de vectorización.

El número de registros vectoriales oscila entre 64 y 256.

Las especificaciones técnicas de un fichero de registro vectorial incluyen los siguientes parámetros:

- El número de registros vectoriales. **MVL** (Maximum Vector Length).
- El número de elementos por registro. El número de elementos por registro son de 8 a 256.
- El tamaño en bytes de cada elemento, suele ser de 8 bytes. Almacena valores en coma flotante en doble precisión.

Otro parámetro a tener en cuenta es el número de puertos de lectura/escritura necesarios para intercambiar datos con las unidades funcionales vectoriales sin provocar detenciones por riesgos estructurales.

La conexión de todos los puertos de lectura y escritura con las unidades funcionales vectoriales se realiza mediante una red de interconexión total de tipo *crossbar*.

Son necesarios mecanismos hardware que detecten los riesgos estructurales y de datos que puedan aparecer

entre las instrucciones vectoriales y escalares del programa.

Las unidades funcionales vectoriales pueden estar a nivel interno organizadas en varios ***lanes*** o ***carriles*** para aumentar las operaciones que pueden procesar en paralelo por ciclo de reloj.

Una unidad funcional vectorial con *n* ***carriles*** implica que 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 en un factor de *n*.

El problema que tiene utilizar unidades sementadas organizadas en carriles es el incremento del número de puertos de lectura y escritura de todos los registros vectoriales para poder suministrar en cada ciclo de reloj los elementos a los carriles.

La solución que se propone para resolver este problema es especializar los carriles de forma que solo tengan acceso a un subconjunto de elementos de cada registro vectorial.

La unidad funcional de carga/almacenamiento vectorial se ocupa de cargar los registros vectoriales

con los datos extraídos de la memoria y de almacenar en ella el contenido de los mismos.

La principal característica es que es capaz de mantener un ancho de banda sostenido de una palabra por ciclo de reloj tras la latencia de acceso inicial.

Dentro de la categoría SIMD se pueden englobar los procesadores vectoriales y matriciales.

Un procesador matricial consta de las siguientes unidades:

- Unidad de procesamiento vectorial.
- Unidad escalar.
- Unidad de control, discrimina según el tipo de instrucción.

La diferencia entre un procesador vectorial y uno matricial está en la organización de la unidad vectorial.

En un procesador matricial la unidad vectorial consta de ***n*** elementos de procesamiento o **EPs**, constituidos por una ALU, un conjunto de registros (**REP**) y una memoria local (**MEP**).

Los EPs se comunican a través de una red de interconexión y pueden funcionar de forma independiente y sincronizada.

Un procesador matricial con *n* elementos de procesamiento puede procesar de forma simultánea en el mismo ciclo de reloj un vector de *n* elementos.

La distribución de los elementos de un vector entre las MEPs es el factor clave para aprovechar al máximo el paralelismo de los EPs.

En condiciones ideales con *n* EPs se podrían producir *n* resultados simultáneamente al realizar una operación vectorial y en el peor de los casos todos los resultados podrían generarse por un único EP.

Resumiendo, 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.

Repertorio Genérico de Instrucciones Vectoriales

El ***registro de longitud vectorial*** **VLR** (Vector Length Register) 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 del MVL (Maximum Vector Length), se almacena en el registro VLR la longitud del vector.

Si el ***tamaño del vector a procesar es mayor*** que el valor del MVL el compilador recurre a una técnica que se llama ***strip mining*** que es un troceado del vector o seccionamiento.

El *strip mining* consiste en procesar un vector de longitud mayor al MVL en secciones de longitud igual o menor al MVL.

Para esto el compilador genera un bucle con instrucciones escalares y vectoriales que se ocupa de procesar el vector en secciones de longitud MVL, y de longitud menor al MVL si el vector no es múltiplo de MVL. Se utiliza (*n mod* MVL)

Para solucionar el problema de cargar y almacenar datos que se encuentran ubicados en memoria de forma no consecutiva pero que siguen un patrón uniforme se utilizan las siguientes instrucciones:

- LVWS Vi,(Ri,Rj) Carga Vi desde la posición M [Ri] con separación en Rj.
- SVWS (Ri,Rj),Vi Almacena Vi a partir de la posición M [Ri] con separación en Rj.

Para solucionar el problema de vectorizar un bucle en cuyo cuerpo hay instrucciones que se ejecutan condicionalmente recurrimos a una máscara almacenada en el ***registro de máscara*** VM (Vector Mask).

Establece qué elementos de un vector tienen que ser manipulados por la unidad funcional al realizarse la operación.

El valor del bit que ocupa la posición *i* en la máscara determina si se realizan o no, *bit a 1 o bit a 0*, las operaciones sobre los elementos que ocupan la posición *i* en los registros vectoriales.

Existen unas instrucciones especiales que gestionan el contenido de ese registro.

S__V Vi,Vj

S__SV Fi,Vi

RVM

MOVF2S VM,Fi

MOVS2F Fi,VM

La mayoría de los procesadores incluyen varios registros de máscara para controlas las operaciones que se realizan sobre los registros vectoriales, pero dependiendo de cómo esté implementado el control de operaciones mediante máscara pueden surgir algunos inconvenientes.

Puede ocurrir que el tiempo de procesamiento no se reduzca pese a que no se realicen las operaciones.

También 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.

Medida del Rendimiento de un Fragmento de Código Vectorial

Tres factores 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 ***n*** a procesar por la unidad funcional.
- El tiempo que se tarda en completar cada resultado o ***tiempo por elemento***.

Expresión que indica el número de ciclos que se tarda en completar una instrucción vectorial que procesa un vector de ***n*** elementos:

$$T_n = T_{arranque} + n \cdot T_{elemento}$$

El $T_{arranque}$ es similar al número de segmentos de que constan. Si una unidad vectorial está compuesta por 6 etapas y cada etapa consume 1 ciclo, el tiempo de arranque es equivalente a la ***latencia inicial de la unidad*** es decir 6 ciclos.

El $T_{elemento}$ que se tarda en completar cada uno de los restantes *n* resultados es 1 ciclo.

Cuando calculamos el tiempo de ejecución de una secuencia de instrucciones vectoriales es necesario tener en cuenta si las instrucciones afectan o no a diferentes unidades funcionales y si existen dependencias verdaderas entre ellas, es decir, si son operaciones independientes.

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 y sin riesgos estructurales.

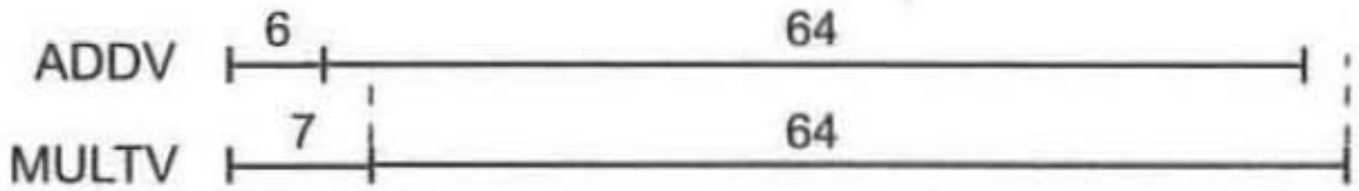
Se puede emitir un convoy cuando todas las instrucciones del convoy anterior han finalizado.

La velocidad de emisión es de 1 convoy por ciclo de reloj.

Por lo general el tiempo por elemento expresado en ciclos es igual al número de convoyes.

ADDV V1, V2, V3

MULTV V4, V5, V6



$$T_{64} = 7 + 64 = 71 \text{ ciclos}$$

$$T_{arranque} = 7 \text{ ciclos}$$

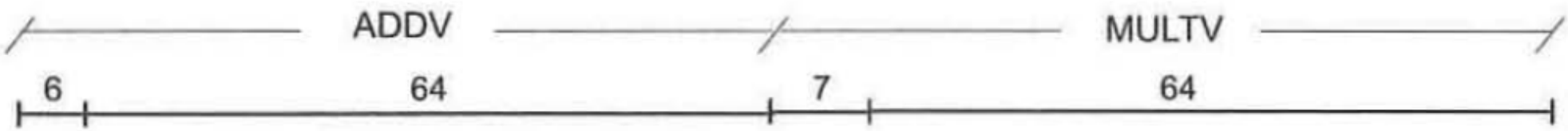
$$T_{elemento} = \frac{64 \text{ elementos} * 1 \text{ ciclo}}{64 \text{ elementos}} = 1 \text{ ciclo}$$

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

Figura 3.27a Ejecución de un convoy de dos instrucciones.

ADDV V1, V2, V3

MULTV V4, V5, V1



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

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

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

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

Figura 3.27b Ejecución de dos convoyes de una instrucción.

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.

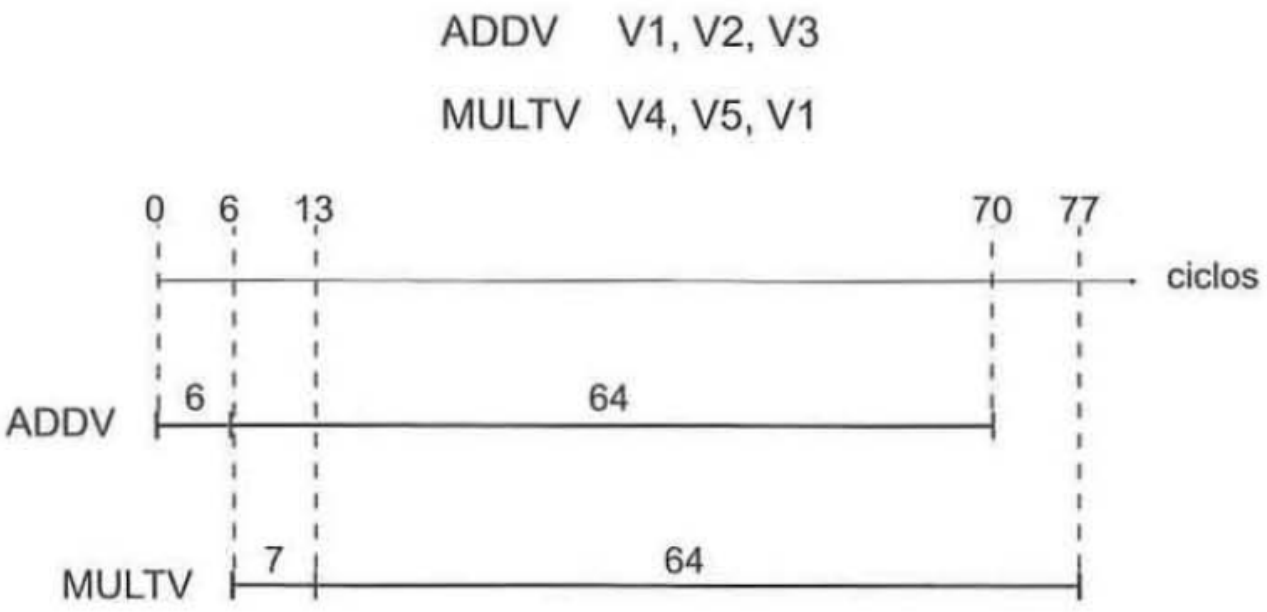
$$R_n = \frac{\left(Operaciones\ en\ coma\ flotante \cdot n\ elementos\right)}{T_n}$$

A mayor valor de R_n mejor será el rendimiento obtenido al ejecutar el código vectorial.

El encadenamiento de resultados entre unidades funcionales, ***chaining***, permite mejorar el rendimiento del procesador. Consiste en reenviar el resultado de una unidad funcional a otra sin esperar a que se complete la operación sobre un registro vectorial.

Si se permite encadenamiento dentro de un convoy pueden existir instrucciones dependientes pero nunca deben existir ***riesgos estructurales*** entre ellas.

El encadenamiento permite reducir el tiempo por elemento gracias al solapamiento pero no los tiempos de arranque.



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

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

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

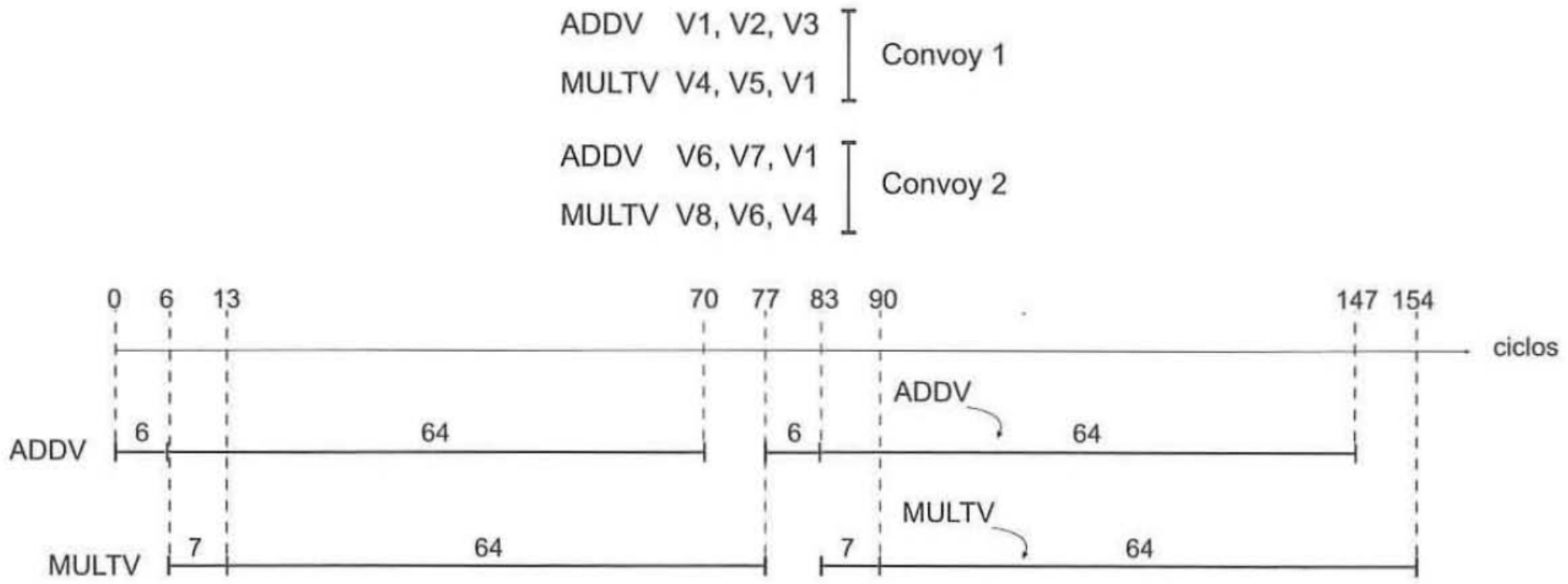
$$R_{64} = \frac{64 \text{ elementos} * 2 \text{ FLOP/elemento}}{77 \text{ ciclos}} = 1{,}66 \text{ FLOP / ciclo}$$

Figura 3.28 Ejecución de dos instrucciones con encadenamiento entre unidades

La ***ejecución solapada*** de diferentes convoyes permite que una instrucción vectorial utilice una unidad funcional antes que una operación previa finalice.

El ***solapamiento de convoyes*** complica la lógica de emisión pero permite ocultar los tiempos de arranque para todos los convoyes excepto para el primero.

La Figura 3.29a 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).



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

$$T_{arranque} = 26 \text{ ciclos}$$

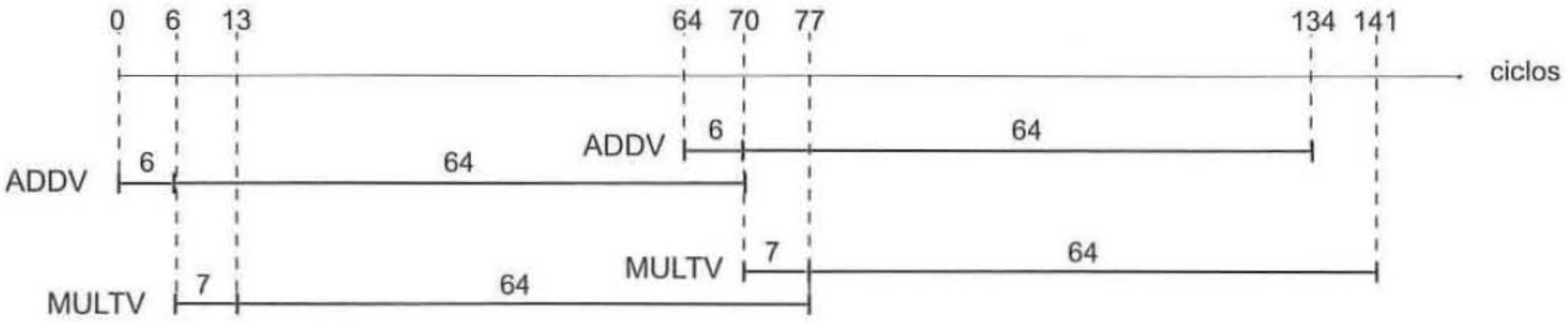
$$T_{elemento} = \frac{64 \text{ elementos} * 2 \text{ ciclos}}{64 \text{ elementos}} = 2 \text{ ciclos}$$

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

Figura 3.29a Ejecución de dos convoyes de instrucciones con encadenamiento entre unidades, Sin solapamiento.

La Figura 3.29b 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.



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

$$T_{arranque} = 13 \text{ ciclos}$$

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

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

Figura 3.29b Ejecución de dos convoyes de instrucciones con encadenamiento entre unidades, Con solapamiento.

La Unidad Funcional de Carga/ Almacenamiento Vectorial

El elemento hardware más crítico en un procesador vectorial para alcanzar un rendimiento óptimo es la unidad de carga/almacenamiento.

Tiene que tener la capacidad de intercambiar datos con los registros vectoriales de forma sostenida y a velocidad igual o superior a la que las unidades funcionales aritméticas consumen o producen nuevos elementos, es decir, a $T_{elemento}$.

Esta 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 de forma adecuada el número de bancos de memoria en función del tiempo de acceso a un banco T_a, se consigue que la latencia para extraer un dato de memoria quede oculta mientras se transfieren los demás elementos que componen un vector a un registro vectorial.

En un procesador vectorial un *dato o elemento*, ***palabra***, suele tener una *longitud de 8 bytes*.

En una ***operación de carga*** de un registro vectorial el ***tiempo de arranque*** es igual a T_a, siendo 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 transferirlo 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 el registro vectorial, suele ser inferior a 1 ciclo pero se iguala a 1 para equipararlo al $T_{elemento}$ de las unidades funcionales.

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.

El ***tiempo de arranque*** es el tiempo que emplea el banco en escribir el último elemento del vector en una posición del banco de memoria.

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, para ambos el tiempo que consume una operación de acceso a memoria para

un vector de $\boldsymbol{n}$ elementos es similar $T_a + n \cdot \boldsymbol{T_{elemento}}$ ciclos.

Cada vez que se solicita un dato de un vector al sistema de memoria supone un coste de $n \cdot \boldsymbol{T_a}$ ciclos, esto es demasiado si pretendemos mantener un ancho de banda igual a $\boldsymbol{T_{elemento}}$.

Para ocultar esta latencia debemos 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.

Distribuimos los n elementos de un vector entre los m bancos de memoria para que se solapen los $(n\text{-}1) \cdot T_a$ ciclos de acceso de $n - 1$ elementos del vector con los $n \cdot T_{elemento}$ ciclos que se emplean en enviar los n elementos a un registro vectorial.

Esta latencia inicial 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, que $\boldsymbol{m \geq T_a}$.

Existen dos formas de realizar el solapamiento de las latencias de acceso a los *n* elementos de un vector:

- De forma síncrona,
- De forma asíncrona,

Se considerará que $\boldsymbol{T_a}$ es igual a *m* y que se está realizando una operación de carga vectorial.

Acceso síncrono a los bancos de memoria de un procesador vectorial.

Se solicita simultáneamente un dato a los ***m*** bancos cada T_a ciclos.

Una vez que realizamos la primera petición y transcurridos T_a ciclos estarán disponibles los ***m*** primeros elementos del vector para ser transferidos al registro vectorial, consumiendo $m \cdot T_{elemento}$ ciclos.

Cada T_a ciclos se realizan dos acciones:

- Se efectúa una nueva petición simultánea a todos los bancos para extraer los ***m*** elementos siguientes al vector.
- Se comienza a transferir ciclo a ciclo los ***m*** elementos ya disponibles y que fueron solicitados en la petición anterior.

Acceso asíncrono a los bancos de memoria de un procesador vectorial.

Se solicitan los elementos del vector a cada uno de los m bancos de forma periódica con periodo T_a con un desfase entre bancos consecutivos de $\boldsymbol{T_{elemento}}$ ciclos.

En el momento en que un banco tiene ya el dato disponible realiza dos acciones:

- Efectúa una nueva solicitud de dato.
- Transfiere el dato ya disponible al registro vectorial.

Aplicamos el mismo razonamiento para las escrituras de forma inversa:

- Primero se transfieren los elementos.
- Luego se realizan los accesos al banco.

Las palabras que componen un vector deben de estar distribuidas correctamente entre los bancos de memoria para que todo funcione bien.

Los bancos de memoria tienen un ancho de palabra de 8 bytes, se direccionan por bytes y son un número que es potencia de 2.

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.

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, iniciándose la lectura de todo el vector.

Dada una dirección de memoria, 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 de los 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.

Medida del Rendimiento de un Bucle Vectorizado

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 que secciona el vector en secciones de longitud igual o inferior a MVL.

T_{bucle}: es 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.

$T_{arranque}$: 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.

$T_{elemento}$: es el número de convoyes en que se organizan las instrucciones vectoriales que se derivan de vectorizar el bucle.

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:

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

$$\left\lceil \frac{n}{MVL} \right\rceil,$$

representa el siguiente valor entero superior a $\left\lceil \frac{n}{MVL} \right\rceil$.

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 $T_{arranque}$ •

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 T_{base}.

Existe la posibilidad que $T_{elemento}$ sea inferior al número de convoyes cuando se permite el solapamiento de instrucciones vectoriales en las unidades funcionales.

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:

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

Si quisiéramos expresar el rendimiento en **FLOPS** (FLOP por segundo), tendríamos que multiplicar el resultado de la expresión previa por la frecuencia de reloj del procesador vectorial.

Para calcular $T_{elemento}$ en situaciones de solapamiento tenemos:

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