

Tema III

Procesadores VLIW y procesadores vectoriales

3.2 Introducción

- ▶ El paralelismo funcional se obtiene mediante la replicación de las funciones de procesamiento que realiza el computador.
 - Granularidad fina → a nivel de instrucciones
 - Granularidad gruesa → a nivel de programas
- ▶ VLIW (very long instructions word – palabras de instrucción muy largas)
 - Se caracteriza por emitir en cada ciclo de reloj una única instrucción pero que contiene varias operaciones
 - La responsabilidad de planificar correctamente las instrucciones fuentes que se puedan codificar como VLIW son del compilador no el hardware
 - En tiempo de compilación se tiene mas tiempo para analizar todos los problemas



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

3.3 El concepto arquitectónico

VLIW

- ▶ En la planificación superescalar la planificación se realiza vía hardware (dinámica)
- ▶ VLIW la planificación es vía software (estática)
 - El compilador establece la secuencia paralela de instrucciones
 - Simplifica el hardware de los procesadores
 - Se emite una instrucción por ciclo
 - Una detención de una unidad funcional, implica la detención de todas las unidades funcionales
 - Causas del fracaso de VLIM
 - Incapacidad de desarrollar compiladores que aprovechen las características VLIW
 - Códigos de baja densidad con numerosas instrucciones NOP
 - Problemas de compatibilidad entre generaciones de procesadores VLIW
 - EPIC Explicit Parallel Instruction Computing



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

3.4 Arquitectura de un procesador VLIW genérico

- ▶ Ausencia de los elementos necesarios para la distribución, emisión y reordenamiento de instrucciones
- ▶ 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 → concatenación de varias instrucciones RISC que se pueden ejecutar en paralelo.
- ▶ Las operaciones recogidas dentro de una instrucción VLIW no presentan dependencias de datos, de memoria v/o de control entre ellas

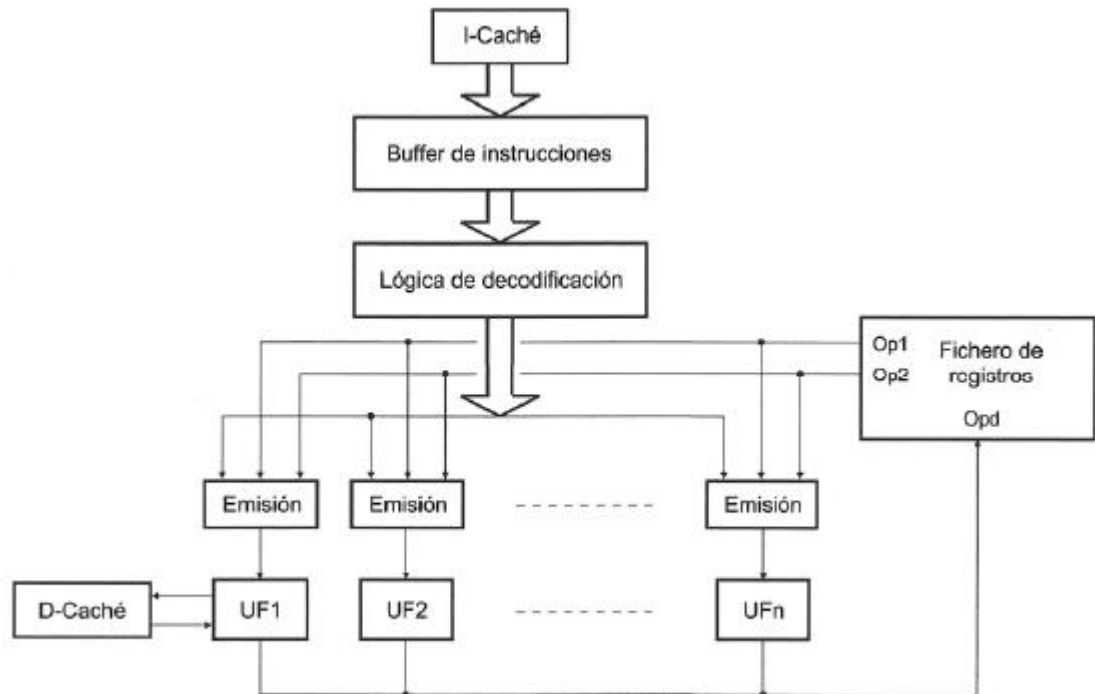


Figura 3.2: Arquitectura básica de un procesador VLIW genérico.

- ▶ El fichero de registro debe disponer de suficientes puertos de lectura para suministrar los operandos a todas las unidades funcionales en un único ciclo de reloj
- ▶ Una instrucción VLIW equivale a una concatenación de varias instrucciones RISC que se pueden ejecutar en paralelo
- ▶ El número y tipos de instrucciones corresponde con el número y tipos de unidades funcionales del procesador
- ▶ 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.
- ▶ Problema para encontrar instrucciones independientes para rellenar todos los slots
 - El código de un procesador VLIW es mayor que uno superescalar
 - No aprovecha al máximo los recursos del procesador ya que tiene unidades funcionales ociosas
- ▶ 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.

Inst. enteras 1 ciclo
 Instr. Coma flotante 2 ciclos

Código ejemplo

```
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
```



Instrucciones VLIW

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

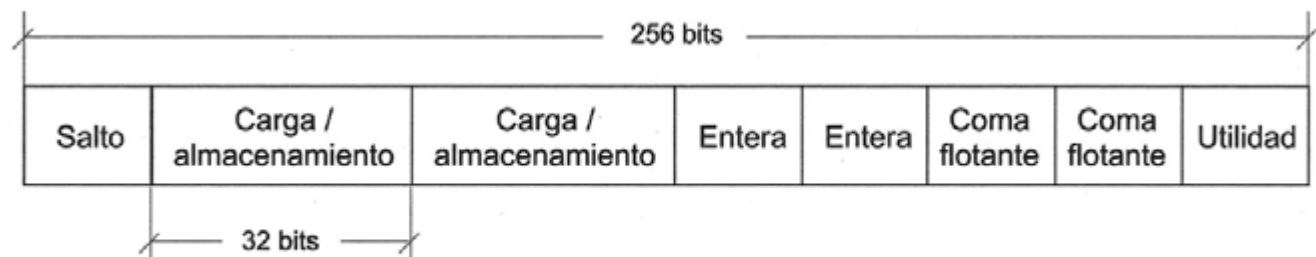


Figura 3.3: Formato de instrucción del computador Multiflow Trace 7.

3.5 Planificación estática o basada en el compilador

- ▶ El compilador VLIW recibe como entrada el código fuente de una aplicación, realiza unas tareas encaminadas a optimizar el código
- ▶ Produce tres elementos:
 - Un código intermedio
 - Un grafo del flujo de control
 - Un grafo del flujo de datos
 - **Código intermedio**
 - Formado por sencillas instrucciones RISC
 - Las únicas dependencias de datos que permanecen en él son las verdaderas, las RAW.
 - Las WAW y las WAR se han eliminado
 - Para poder generar un grafo de flujo de control es necesario conocer los bloques básicos de que consta el código intermedio.
 - **Bloque básico**
 - Un conjunto de instrucciones que conforman una ejecución secuencial
 - No hay instrucciones de salto salvo la última
 - No hay puntos intermedios de entrada salida
 - Para obtener los bloques básicos se analiza el código teniendo en cuenta que
 - Comienzo de un bloque básico:
 - Una instrucción etiquetada o la siguiente a un salto
 - El bloque se compone desde la instrucción inicial hasta la siguiente de salto
 - Los bloques se numeran de forma secuencial
 - 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.

Diagrama de flujo de control y bloques básicos

Un grafo de flujo de control

Las interconexiones de los bloques básicos, atendiendo las posibles direcciones que pueda seguir la ejecución del código

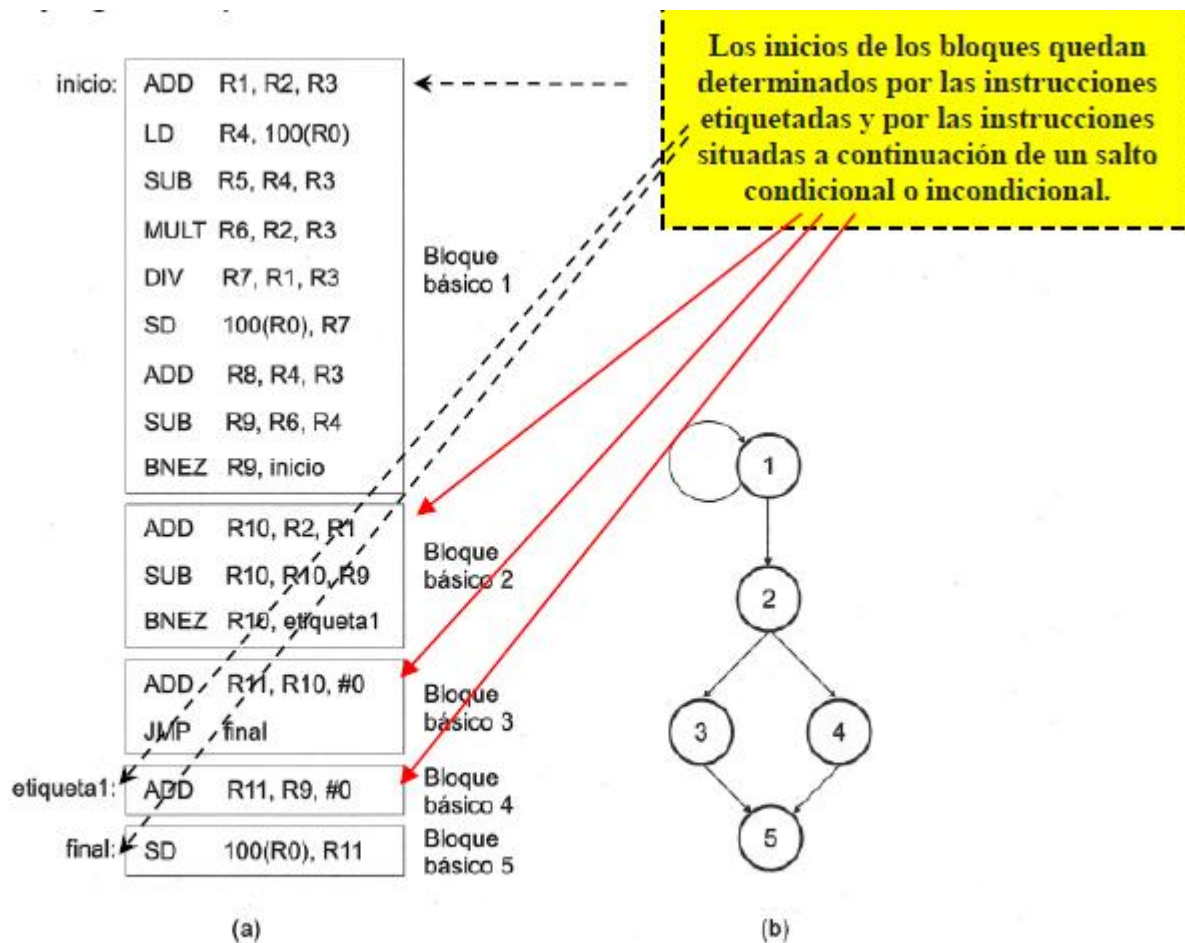


Figura 3.4: Ejemplo de secuencia de código intermedio con delimitación de los bloques básicos (a) y diagrama de flujo de control en el que se muestra la secuencia de ejecución de los bloques y la existencia de bucles (b).

Grafo de flujo de datos

- ▶ Es un grafo dirigido
- ▶ Los nodos son las instrucciones del bloque básico
- ▶ Los arcos se inician en la instrucción de escritura en un registro (instrucción productora) y tiene por destino la instrucción que lee ese registro (instrucción consumidora)
- ▶ 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.
 - Planificación local:
 - Combinación de instrucciones de un solo bloque básico
 - Está limitada, (tamaño bloque básico 5 0 6 instrucciones)
 - Técnicas
 - Desenrollamiento de bucles
 - La segmentación software
 - Planificación global:
 - Combinar instrucciones de diferentes bloques básicos con el fin de producir una planificación con mayor grado de paralelismo

Diagrama de flujo de datos

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)

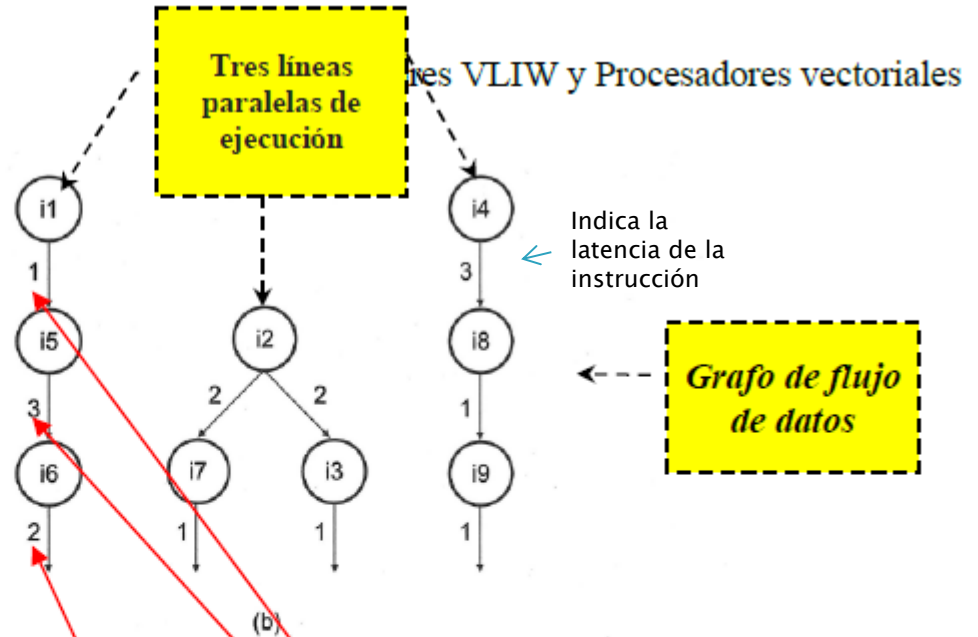


Figura 3.5: Secuencia de operaciones de un bloque básico (a) y diagrama de flujo de datos de la secuencia de instrucciones (b).

Consideraciones sobre formato de instrucción → 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)
- Una de almacenamiento (dos ciclos de latencia)
- Las instrucciones de salto son consideradas como instrucciones de suma/resta que escriben en el registro contador de programa (un ciclo de latencia)

Secuencia planificada de instrucciones VLIW

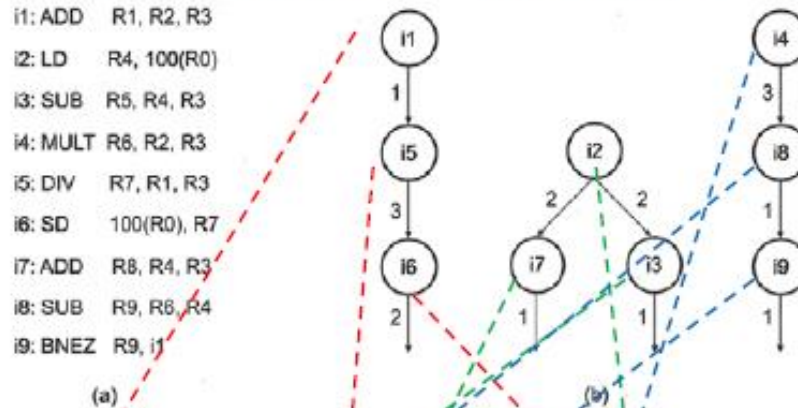


Figura 3.5: Secuencia de operaciones de un bloque básico (a) y diagrama de flujo de datos de la secuencia de instrucciones (b).

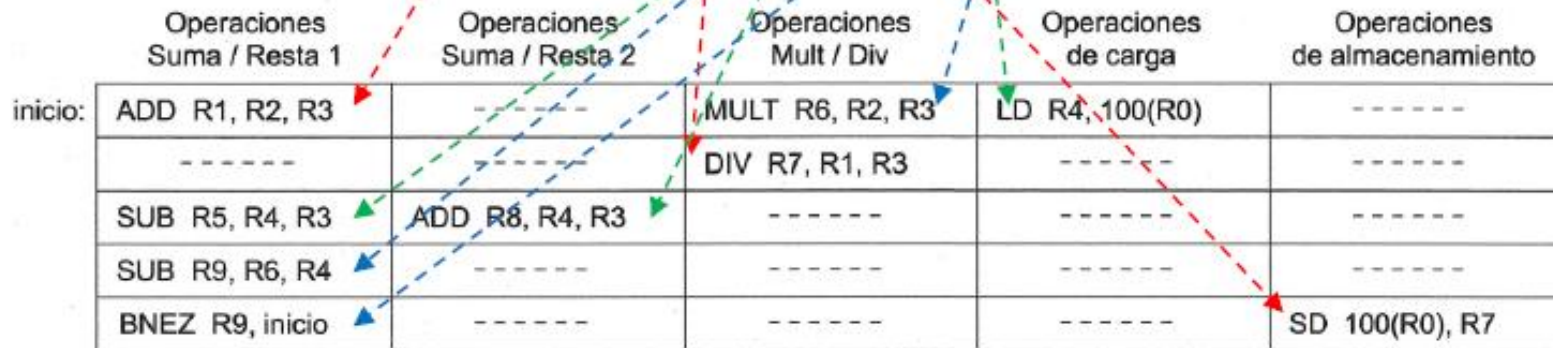


Figura 3.6: Codificación de las operaciones del bloque básico en instrucciones VLIW.

- ▶ Si se considera que las instrucciones VLIW tienen una longitud de 20 bytes. Cada operación básica ocupa 4 bytes.
- ▶ El desaprovechamiento del código VLIW es del 64 %
- ▶ El programa consume 100 bytes de almacenamiento en memoria pero solo un 36 % está ocupado por operaciones útiles.

3.6 Desarrollo de bucles (loop unrolling)

- ▶ Aprovecha el paralelismo existente entre las instrucciones de un bucle.
- ▶ Replicamos 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.
- ▶ Ventajas
 - Reduce el número de iteraciones del bucle
 - El total de instrucciones ejecutadas es menor ya se eliminan saltos y se reduce el número de instrucciones de incremento/decremento de los índices que se utilicen en el bucle.
 - Se proporciona al compilador un mayor número de oportunidades para planificar las instrucciones ya que al desarrollar el bucle queda mucho más cálculo al descubierto al incrementarse el tamaño de los fragmentos de código lineal.
 - El bloque básico se compone por todas las instrucciones que hay desde la instrucción inicial hasta la siguiente instrucción de salto que se detecte
 - Planificación más efectiva
 - Generar código VLIW más compacto

Ejemplo

```
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.

Se desenrolla el cuerpo del bucle

```
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
```

Reordenamiento de las instrucciones

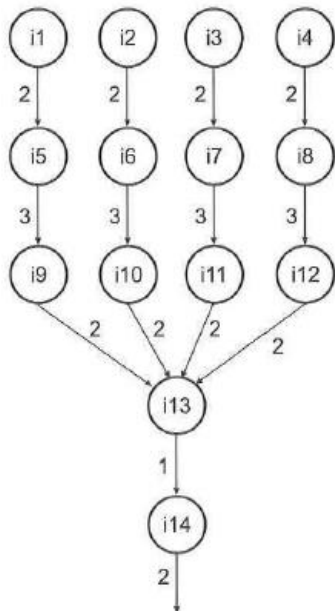
```

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
    
```

3 Unidades funcionales:

- Operaciones enteras (un ciclo de latencia)
- Operaciones en coma flotante (tres ciclos de latencia)
- 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.

	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



Decremento adelantado

Consideraciones

- ▶ La 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.
- ▶ Las instrucciones de almacenamiento que deben esperar a que los resultados de las operaciones de suma en coma flotante estén disponibles.
- ▶ 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.
 - 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.

Rendimiento

- ▶ El tamaño del código VLIW es mayor
 - 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 → $9 \text{ instrucciones} * 12 \text{ bytes} = 108 \text{ bytes}$
 - Operaciones originales → $14 \text{ instrucciones} * 4 \text{ bytes} = 56 \text{ bytes}$
 - Bucle original → $5 \text{ instrucciones} * 4 \text{ bytes} = 20 \text{ bytes}$
- ▶ Mejora es en el rendimiento.
 - Si el vector constase de 1000 elementos
 - Bucle original sin aplicar NADA 1000 veces:
 - $1000 \text{ iteraciones} * 5 \text{ instrucciones} = 5000 \text{ ciclos}$
 - En el mejor de los casos, supuesta una segmentación ideal y sin riesgos.
 - El cuerpo del bucle desenrollado cuatro veces :
 - $250 \text{ iteraciones} * 14 \text{ instrucciones} = 3500 \text{ ciclos.}$
 - El VLIW
 - $250 \text{ iteraciones} * 9 \text{ instrucciones} = 2250 \text{ ciclos}$

Comparación con la versión VLIW que se obtendría del bucle original sin recurrir a ninguna técnica de planificación local

- ▶ $6 * 12$ (bytes/instrucción) = 72 bytes, de los cuales solo 20 bytes están ocupados con operaciones.
- ▶ Velocidad de ejecución
 - Si el vector constase de 1000 elementos se tardaría en procesarlo 6000 ciclos de reloj frente a los 2250 ciclos obtenido tras aplicar desenrollamiento.

$$\frac{6000 - 2250}{2250} = 166,66\% \text{ más rápido}$$

	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.

Reordenamiento de las instrucciones

- ▶ Mejora el paralelismo
- ▶ Mejora el rendimiento
- ▶ El código ocupa mas

3.7 Segmentación software

- ▶ Intenta aprovechar al máximo el paralelismo existente dentro del bucle
- ▶ Consiste en 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).
- ▶ Instrucciones A, B, C y D → cada una un ciclo de reloj
 - Una dependencia RAW con la instrucción que la precede → D depende C, C depende B, B depende A.
 - Tras tres 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.

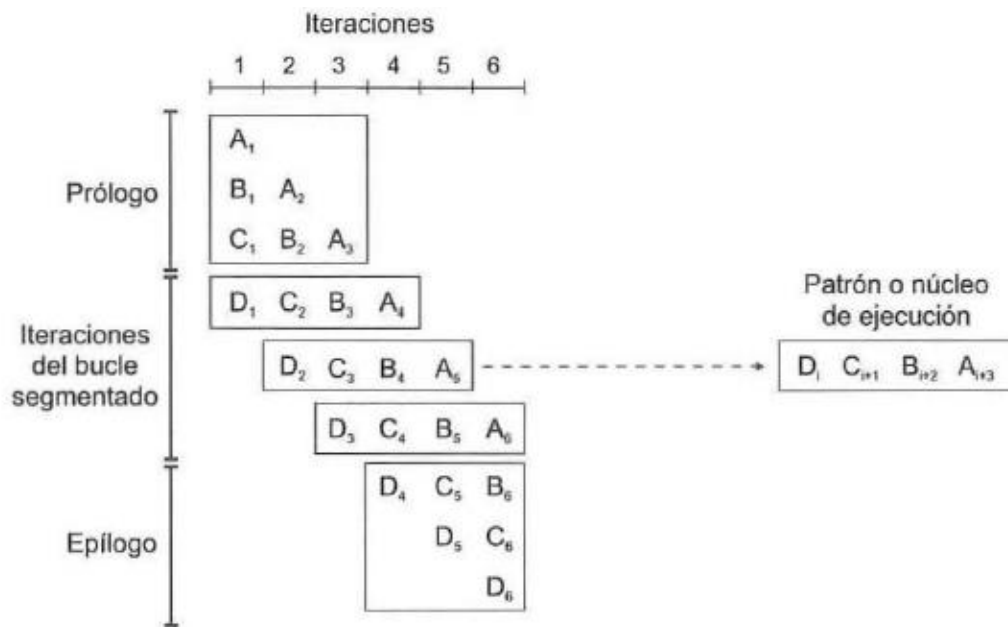


Figura 3.9: Aplicación de la segmentación software al cuerpo de un bucle genérico.

Ejemplo

```

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

Latencias

- 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

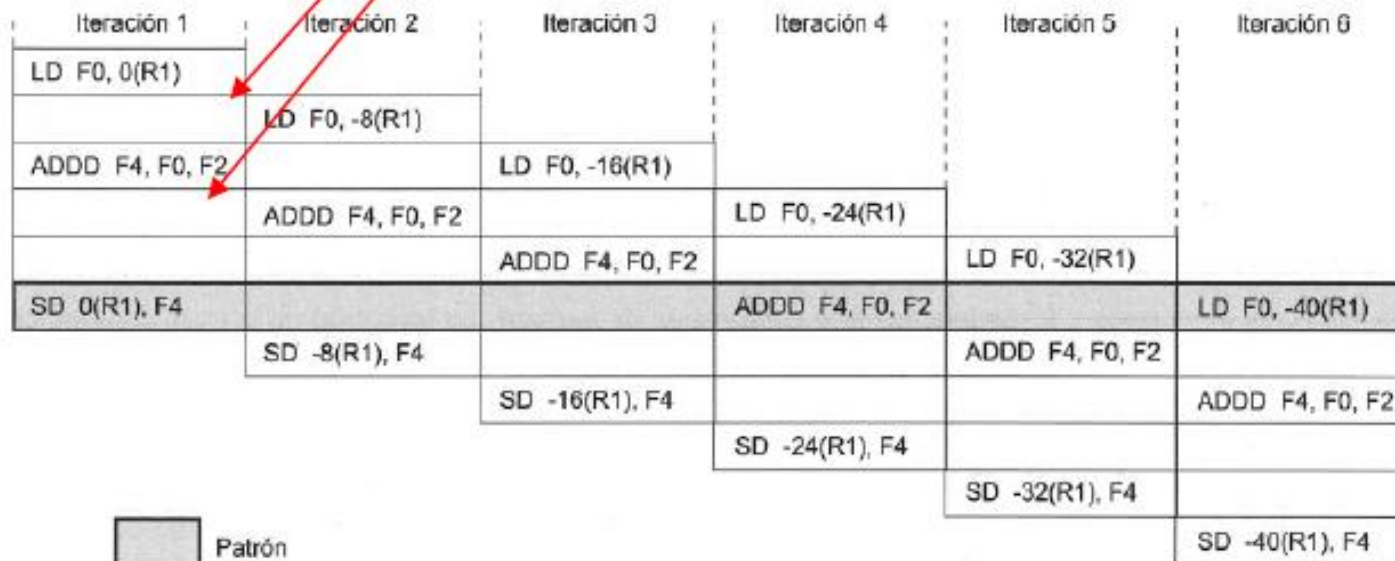


Figura 3.10: Esquema del patrón de ejecución obtenido al aplicar la técnica de segmentación software.

- ▶ Si las instrucciones VLIW son de 16 bytes, el tamaño total del código es de (11 inst.*16 byt/inst).= 176 bytes.
- ▶ Tiempo para procesar un vector de 1000 elementos:
- ▶ La aproximación VLIW emplearía 1010 ciclos.
 - 5 corresponderían al prólogo.
 - 5 al epílogo.
 - 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 hay instrucciones condicionales en el cuerpo del bucle que impiden la aparición de un patrón de comportamiento regular.



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

3.8 Planificación de trazas (trace shedding)

- ▶ Es una técnica de planificación global
- ▶ Traza:
 - Camino de ejecución mas probable
- ▶ Pasos
 - 1.- Selección de la traza
 - Encontrar un conjunto de bloques básicos que conformen una secuencia de código sin bucle
 - Seleccionamos al que especulemos que será mas probable que se ejecute
 - Compilador utiliza un Grafos con pesos (ponderados) por distintos criterios perfiles de ejecución, estimaciones, planificación estática de saltos...
 - 2.- Compactación de la traza

Código intermedio del bucle sin compactar

```
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;
```

- 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

A pesar de ello se hace con decremento

R2 puntero a X[i]

R3 puntero a Y[i]

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
```

R1 puntero a A[i]

8 bytes

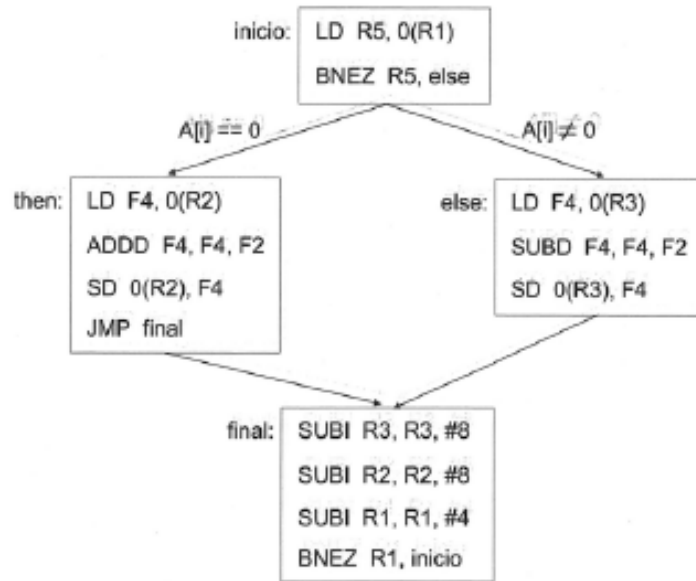
4 bytes

3 U.U.F.F. con ciclos de latencia:

2 3 1

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), F4	-----	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



Ruta de ejecución más probable:
(caso $A[i] == 0$)

→ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16, 17

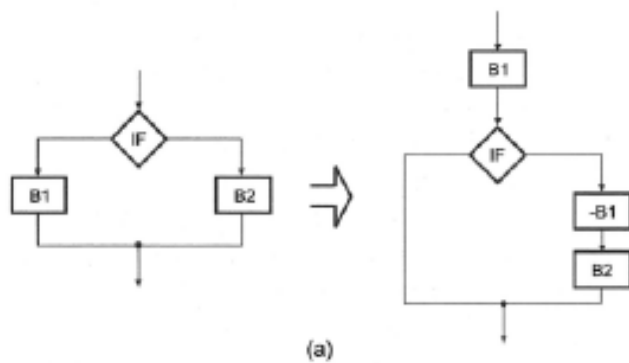
→ 12 ciclos

Ruta menos probable:
(caso $A[i] <> 0$)

→ 1, 2, 3, 4, 10, 11, 12, 13, 14, 15, 16, 17

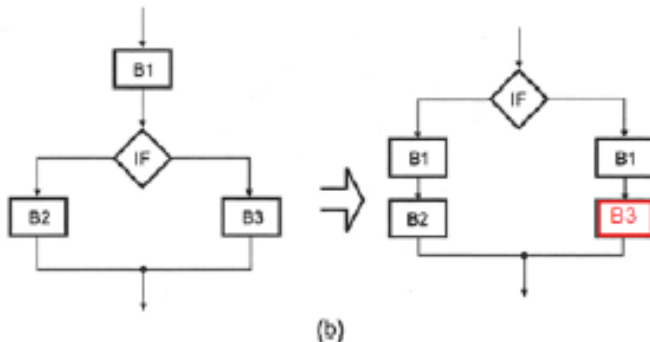
→ 12 ciclos

Posibles situaciones en las que plantera desplazamiento de operaciones

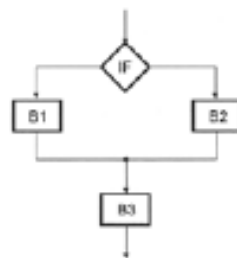


(a)

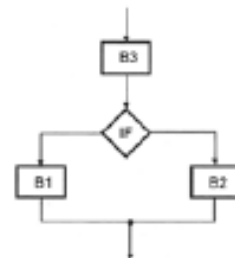
Se podrían desplazar antes de la condición lo que obligaría a anular sus efectos mediante un código de compensación en caso de que no fuese necesaria su ejecución. Esto se ha representado en la figura mediante el bloque etiquetado como -B1 que indica que es necesario realizar las operaciones que anulen el efecto de la ejecución del contenido de B 1.



(b)



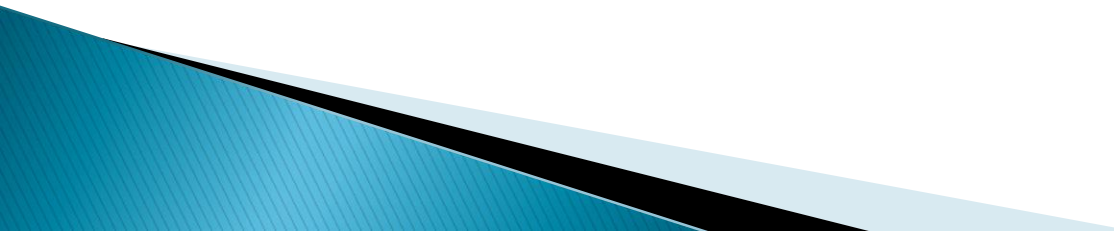
(c)



(d)

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.

Consideraciones del compilador para desplazar operaciones dentro de una traza

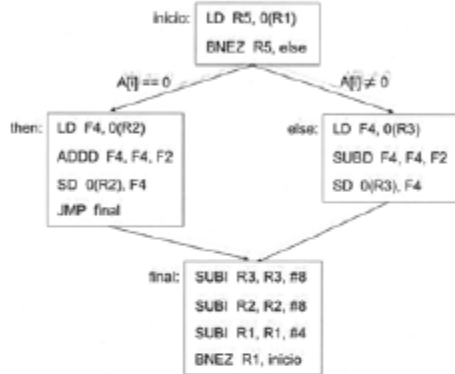
- ▶ Conocer cuál es la secuencia de ejecución más probable.
 - ▶ Conocer las dependencias de datos existentes para garantizar su mantenimiento.
 - ▶ La cantidad de código de compensación que es necesario añadir.
 - ▶ Saber si compensa el desplazamiento de operaciones dentro de la traza, midiéndose el coste tanto en ciclos de ejecución como en espacio de almacenamiento.
- 

Ejemplo

```

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;

```

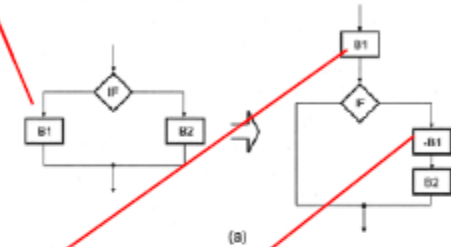


```

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

```

Aplicando el desplazamiento de operaciones del bloque B1



```

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

```

	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 0(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.

- ▶ Ruta de ejecución más probable: 1, 2, 3, 4, 5, 6, 7, 8, 15, 16
 - 10 ciclos decremento de 2 ciclos respecto a la no planificación
- ▶ Ruta menos probable: 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16
 - 15 ciclos Incremento de 3 ciclos
- ▶ 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:
 - $[10 \text{ ciclos} * p + 15 \text{ ciclos} * (1 - p) < 12 \text{ ciclos} * p + 12 \text{ ciclos} * (1 - p)]$
 - p es la probabilidad de que la rama (A[i] == 0) sea la ejecutada.
- ▶ 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.

	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.

	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.

3.9 Operaciones con predicado

- ▶ Operación con predicado es una instrucción en la que su resultado se almacena o se descarta en función de un operando que tiene asociado
 - 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
 - $p = 1 \rightarrow$ resultado de la operación se almacena
 - $p = 0 \rightarrow$ resultado de la operación se anula
 - Es efectiva si todas las rutas de una región tiene el mismo tamaño y la misma frecuencia de ejecución

Ejemplo

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

(a)

Código fuente de un bucle genérico

```
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)

Código intermedio

Poner a "1" P1 o P2 en función de la comparación de R1 y el n° #50

```
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)

Almacena el resultado de ADDI si P1 = "1"

Código intermedio aplicando operaciones con predicado

	2 ciclos	3 ciclos	1 ciclo	
	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)

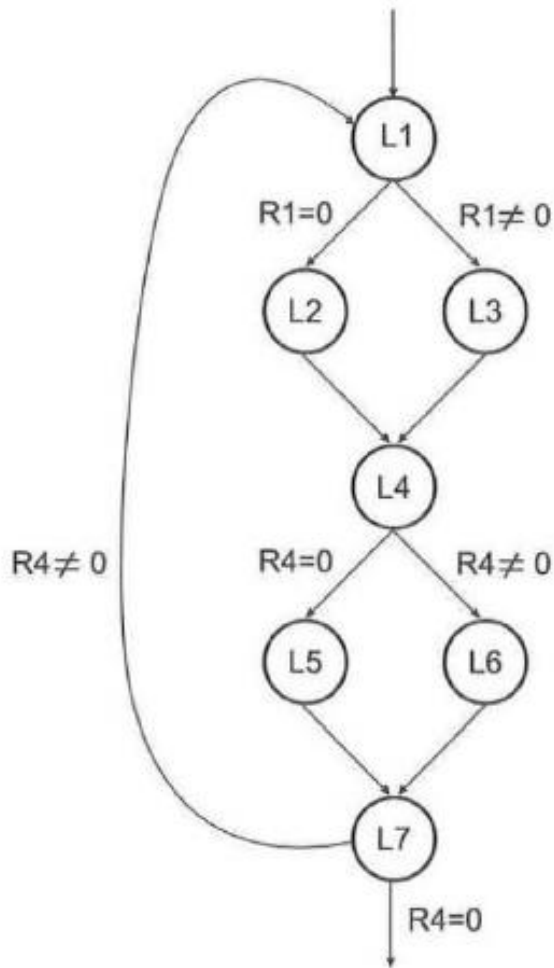
inicio:	LD R1, 0(R2)	-----	-----	1
	-----	-----	-----	2
	-----	-----	PRED_EQ P1, P2, R1, #50	3
	-----	-----	ADDI R5, R5, #2 (P1)	4
	-----	-----	ADDI R5, R5, #1 (P2)	5
	-----	-----	SUBI R2, R2, #4	6
	-----	-----	BNEZ R2, inicio	7
	-----	-----	-----	8

(b)

Instrucciones VLIW derivadas del código intermedio

Instrucciones VLIW derivadas del código intermedio con operaciones condicionadas

- VLIW sin operaciones condicionadas 11 y 9 ciclos según rama.
- VLIW con operaciones condicionadas 8 ciclos.



(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.

3.10 Tratamiento de Excepciones

- ▶ Las técnicas vistas hasta ahora se basan en la predicción de los resultados de los saltos condicionales. Habrá que establecer mecanismos para el tratamiento de las excepciones (interrupciones).
- ▶ Centinelas
 - Un fragmento de código que indica que la operación ejecutada de forma especulativa con la que está relacionado ha dejado de serlo.
 - El compilador marca las operaciones especulativas con un a etiqueta y en lugar del programa en el que estaba el código especulado que ha sido desplazado, sitúa un centinela vinculado a esa etiqueta
 - 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

3.11 El enfoque EPIC

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

- 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.

```
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]
ADD   F6,F4,F2  % Suma en F6
SD    0(R3),F6  % Almacenamiento de A en M[0+R3]
```

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

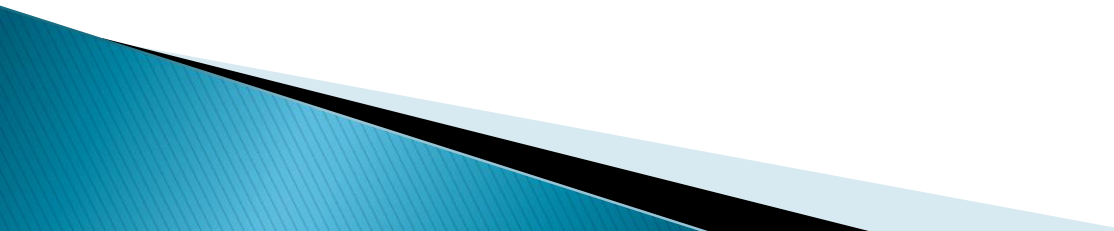
(a)

```
[2] LD    F2, 0(R1)
[1] LD    F4, 0(R2)
[1] ADD   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).

3.12 Procesadores vectoriales

- ▶ Arquitectura que permite la manipulación de vectores
 - Operaciones para manejar vectores
 - Unidades funcionales para operar con vectores
 - Cálculo científico
 - Gran volumen de datos
 - Compilador encargado de detectar y extraer el paralelismo
- 

3.13 Arquitectura vectorial básica

- ▶ Basados en arquitectura carga-almacenamiento
- ▶ Unidad de procesamiento escalar y unidad de procesamiento vectorial
- ▶ MVL Maximum Vector Length – Máxima longitud del vector
 - Número de elementos por registro

Esquema de un procesador vectorial básico

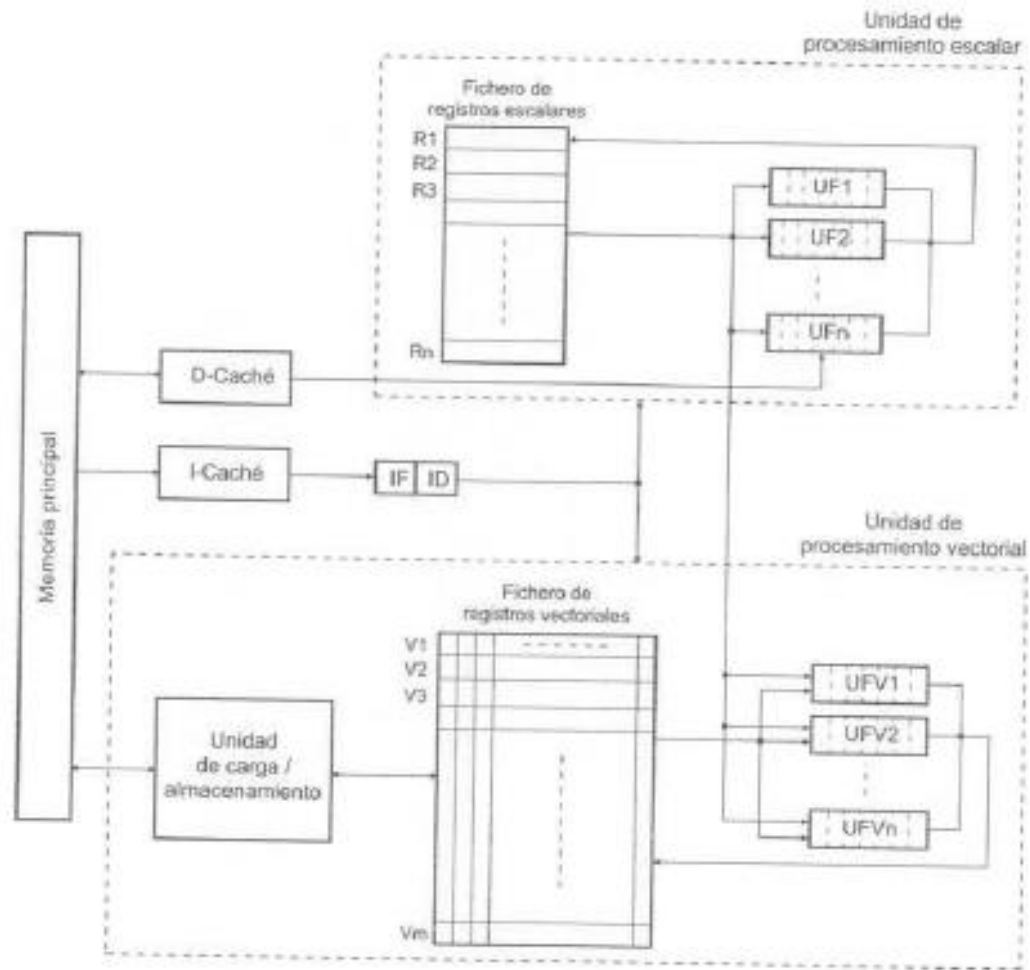


Figura 3.20: Esquema de un procesador vectorial básico.

Unidades funcionales con varios carriles (lanes)

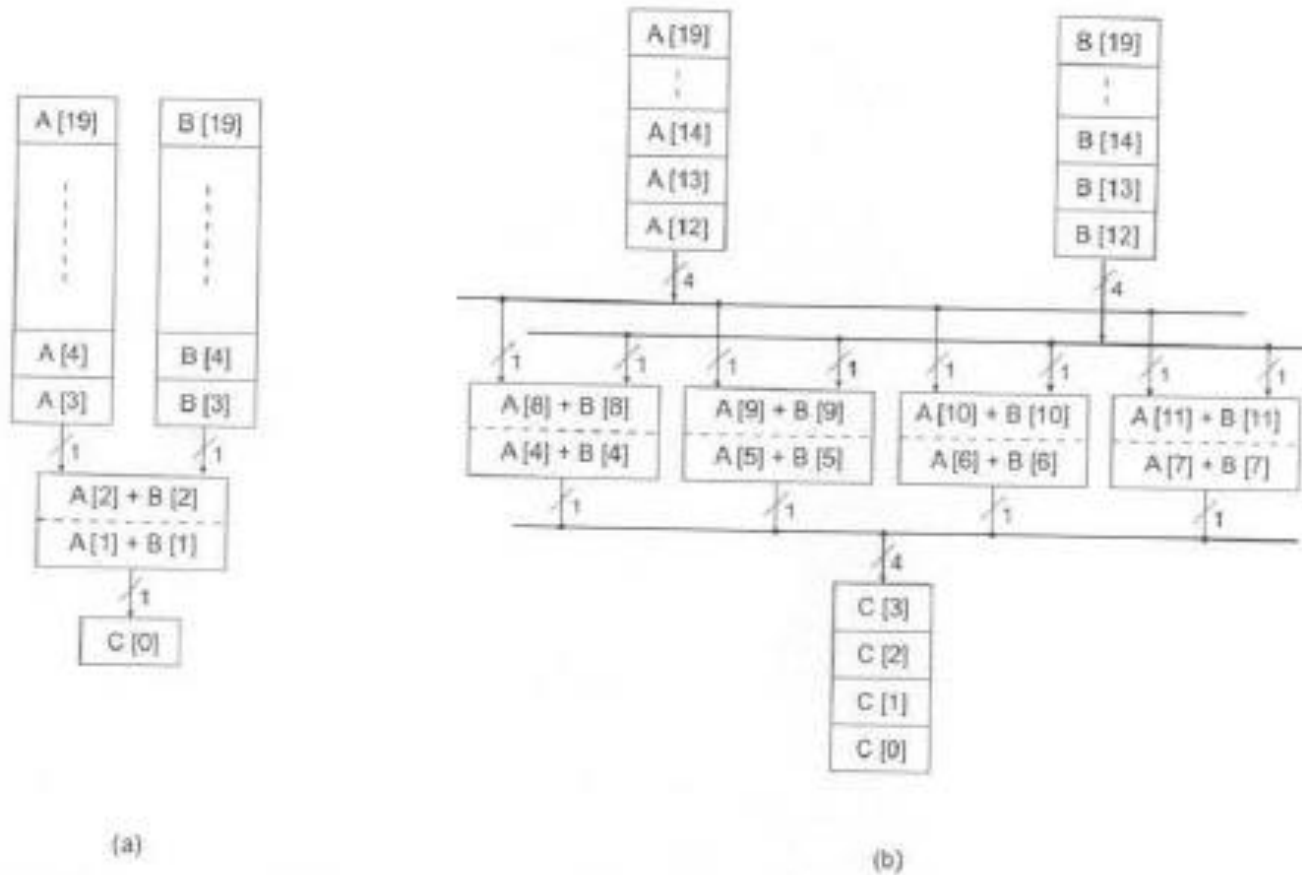


Figura 3.21: Diferencia entre realizar la suma de 2 vectores de 20 elementos en una unidad funcional con 2 etapas y un único carril (a) y en otra unidad funcional de 2 etapas pero con 4 carriles (b).

Unidades de cuatro carriles o lanes

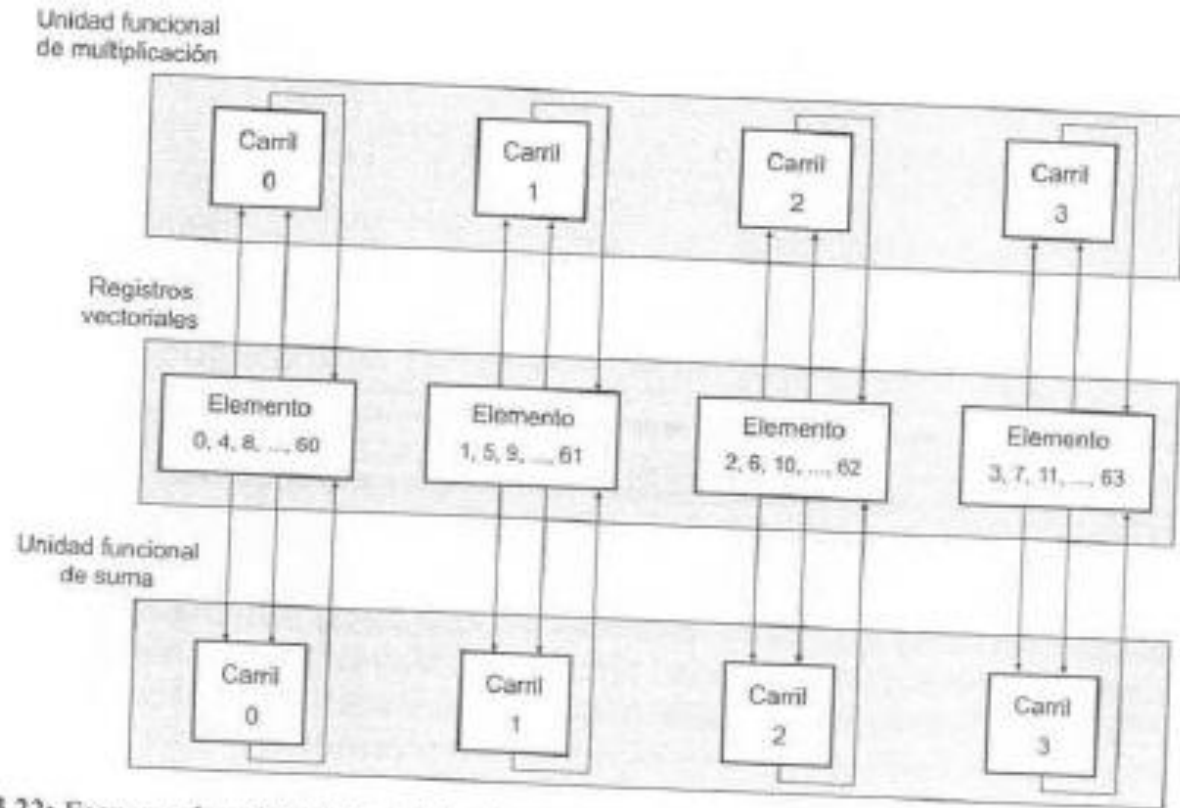


Figura 3.22: Esquema de unidades vectoriales de suma y multiplicación en coma flotante con cuatro carriles y distribución de los elementos de los registros vectoriales entre ellas.

Características procesadores vectoriales

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 SVlex	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

Procesadores matriciales

- ▶ La unidad vectorial se compone de n elementos de procesamiento EP, constituidos por unidades aritmético-lógicas de propósito general, un conjunto de registros REP, y una memoria local MEP
- ▶ Hoy no se fabrican

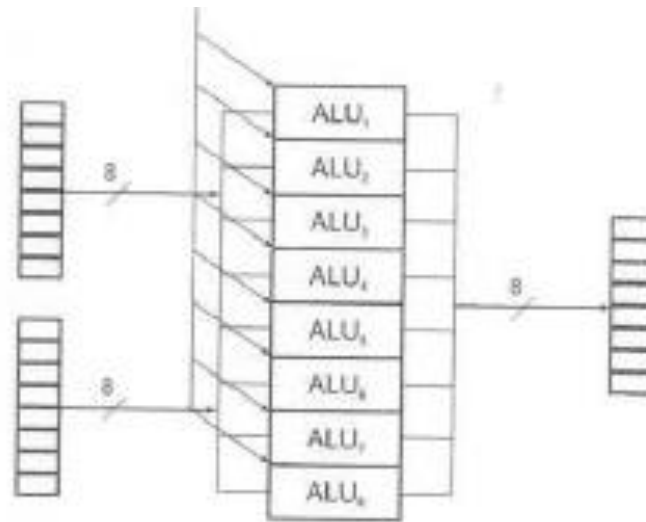


Figura 3.23: Ejemplo simplificado de la realización de una operación sobre dos vectores en un procesador matricial compuesto por ocho EPs.

3.14 Repertorio de instrucciones vectoriales

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].

```

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]
ADD     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

VRL (Vector Length register) Registro de longitud vectorial

- ▶ VRL controla la longitud de cualquier operación vectorial
- ▶ MVL (Maximun Vector Length) máxima longitud del vector
- ▶ Cuando el tamaño del vector es distinto al numero de registros vectoriales (MVL), el VRL, controla la longitud de cualquier operación vectorial
- ▶ Strip mining
 - Troceado del vector, cuando el tamaño del vector (VRL) es mayor que el valor del MVL (máxima longitud del vector)

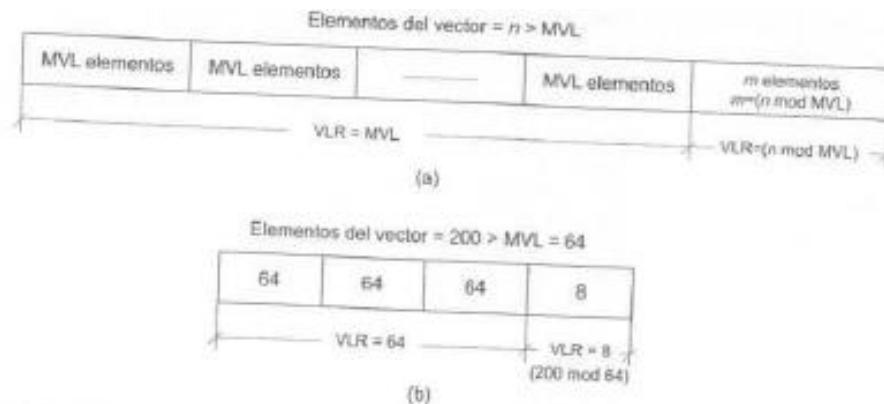


Figura 3.24: Seccionamiento de un vector genérico en secciones de MVL elementos (a) y de un vector de 200 elementos en secciones de 64 (b).

Para ello, en cada iteración se ajusta el valor del registro VRL 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 $(n \bmod MVL)$ elementos y en las siguientes ya se procesan secciones de longitud MVL. Para ello, en la primera iteración se asigna el valor $(n \bmod MVL)$ al registro VRL y en las siguientes el valor MVL. Para poder manipular el registro VRL, los procesadores vectoriales cuentan con algún tipo de instrucciones especiales análogas a las dos siguientes:

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

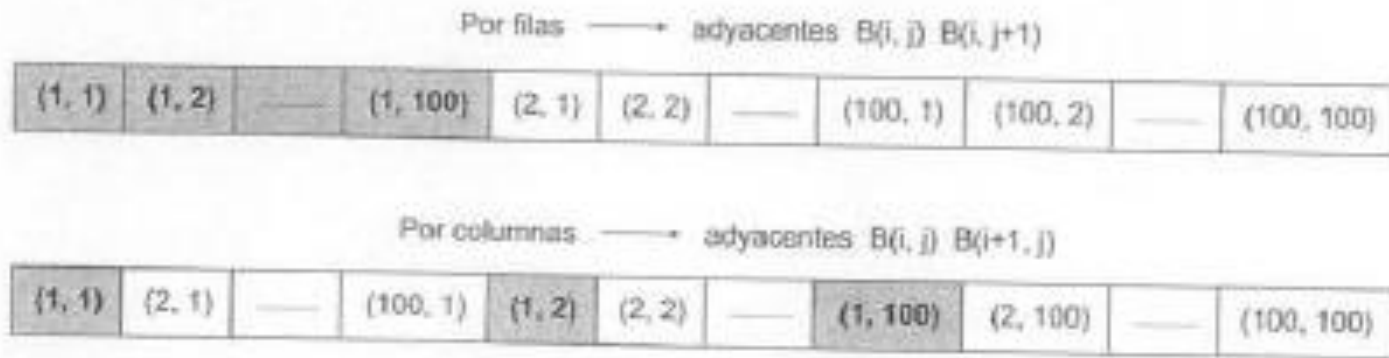


Figura 3.25: Almacenamiento unidimensional por filas o por columnas de una estructura bidimensional de 100×100 .

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 $V_i, (R_i, R_j)$ Carga V_i comenzando desde la posición $M[R_i]$ con una separación de R_j .
- SVWS $(R_i, R_j), V_i$ Almacena V_i a partir de la posición $M[R_i]$ con una separación de R_j .

VM Vector Mask

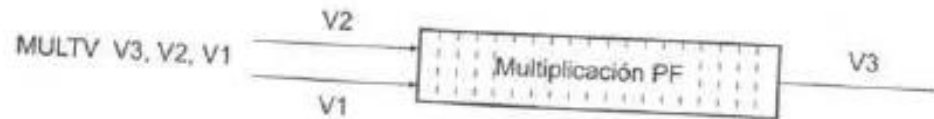
El tercer interrogante al que tiene que dar respuesta el repertorio de instrucciones de un procesador vectorial es el que se plantea cuando hay que vectorizar bucles en cuyo cuerpo hay instrucciones ejecutadas condicionalmente. Esto provoca que, dependiendo de la condición, ciertos elementos de un vector 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:

<code>S_V Vi, Vj</code>	Compara (EQ, NE, GT, LT, GE, LE) elemento a elemento el contenido de V_i y V_j . 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.
<code>S_SV Fi, Vi</code>	Similar a la anterior pero utilizando un valor escalar en la comparación.
<code>RVM</code>	Inicializa a 1 todos los bits del registro VM.
<code>MOV2S VM, Fi</code>	Almacena en VM el contenido del registro en coma flotante F_i .
<code>MOV2F Fi, VM</code>	Almacena en el registro F_i el contenido del registro VM.

3.15 Medidas del rendimiento de un fragmento de código vectorial

- ▶ Factores
 - Latencia en producir el primer resultado
 - Número de elementos a procesar por unidad funcional
 - Tiempo que se tarda en procesar cada datos
- ▶ Convoy o paquete
 - Conjunto de instrucciones sin dependencias reales ni riesgos estructurales que pueden planificarse juntas
- ▶ Tiempo de arranque
 - Tiempo en disponer el primer resultado
 - Será igual al número de etapas

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



$$T_{\text{arranque}} = 10 \text{ ciclos}$$

$$T_{\text{elemento}} = 1 \text{ ciclo}$$

$$T_{\text{tot}} = 10 \text{ ciclos} + 64 \text{ elementos} * 1 \text{ ciclo} = 74 \text{ ciclos}$$

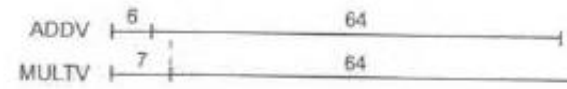
Figura 3.26: Tiempo de ejecución de una operación de multiplicación sobre 2 vectores de 64 elementos.

Sin dependencias



ADDV V1, V2, V3

MULTV V4, V5, V6



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

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

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

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

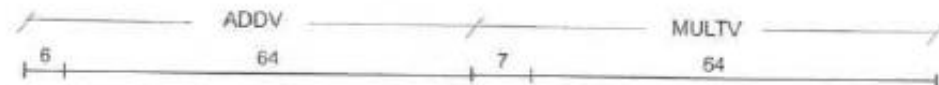
(a)

Con dependencias
(sin encadenamiento)



ADDV V1, V2, V3

MULTV V4, V5, V1



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

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

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

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

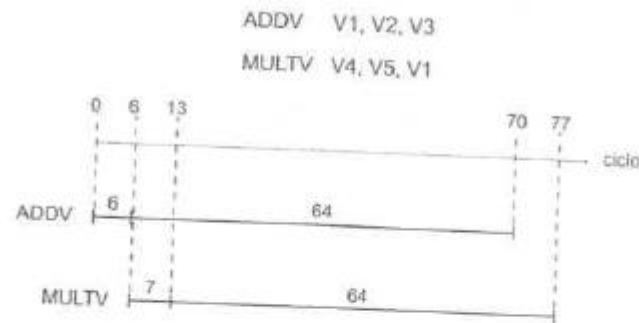
(b)

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

Encadenamiento de resultados entre unidades funcionales (chaining)

- ▶ Permite a una unidad funcional empezar a operar tan pronto como los resultados de la unidad funcional de que depende estén disponibles (pasado el tiempo de arranque)

Con encadenamiento
Con dependendencia



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

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

$$T_{\text{dependencia}} = \frac{64 \text{ elementos} \cdot 1 \text{ ciclo}}{64 \text{ elementos}} = 1 \text{ ciclo}$$

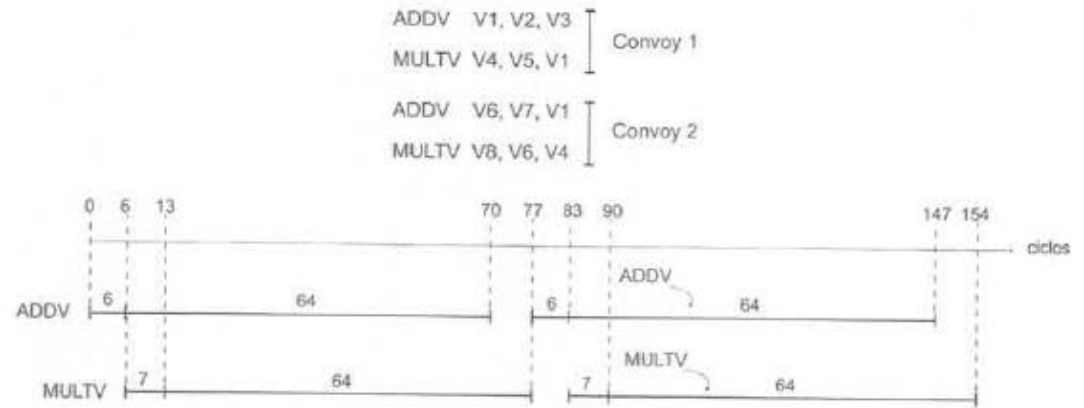
$$R_{\text{tot}} = \frac{64 \text{ elementos} \cdot 2 \text{ FLOP/elemento}}{77 \text{ ciclos}} = 1,66 \text{ FLOP / ciclo}$$

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

Solapamiento

Sin solapamiento:
hasta que no
termina un
convoy no puede
empezar otro

Ejecución
solapada de
varios convoy



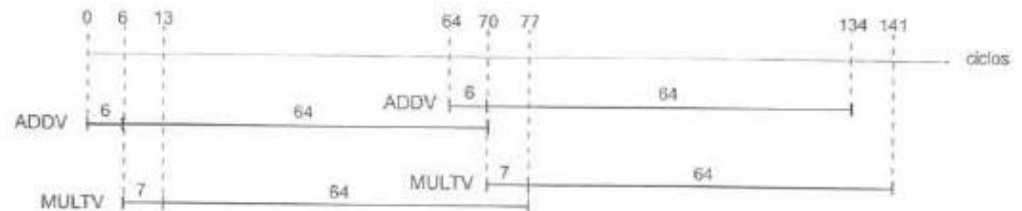
$$T_{\text{ca}} = (6 + 7) + 64 + (6 + 7) + 64 = 154 \text{ ciclos}$$

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

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

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

(a)



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

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

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

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

(b)

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

3.16 La unidad funcional de carga/almacenamiento vectorial

- ▶ T_a = tiempo de arranque
 - 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
- ▶ Carga:
 - primero se solicita (T_a = tiempo de arranque) y después se almacenan los datos
- ▶ Almacenamiento:
 - 1 transfiere los elementos del vector a los puertos de escritura de los bancos de memoria
 - 2 tiempo que tarda en escribir el último elemento del vector en el banco de memoria

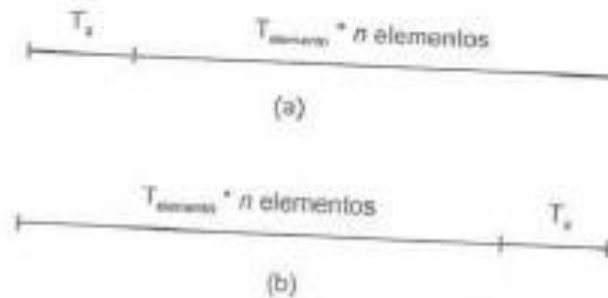
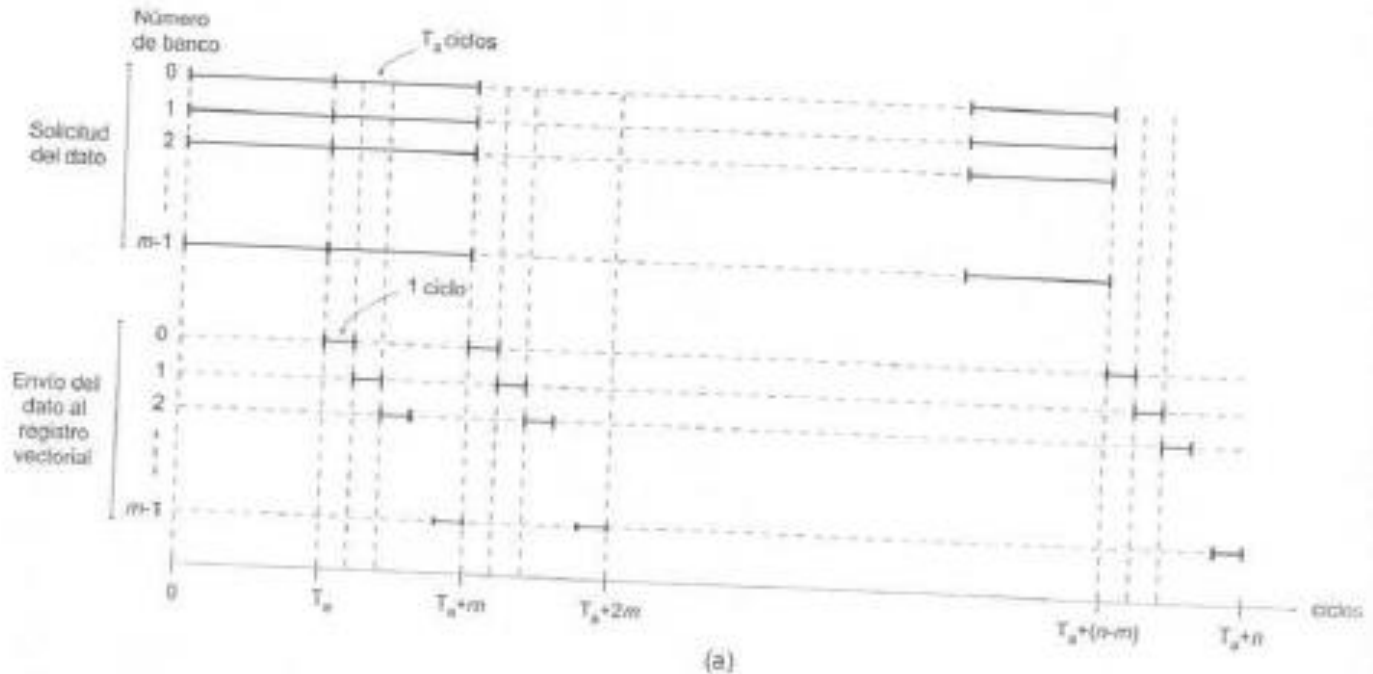


Figura 3.30: Diferencias entre instrucciones de carga (a) y almacenamiento (b) en la visibilidad de los tiempos de acceso y por elemento.

Acceso síncrono

- ▶ Solicita simultáneamente un dato a los m bancos cada T_a ciclos
- ▶ Cada T_a ciclos se realizan dos acciones
 - 1 Efectuar una nueva petición simultáneas a los todos los bancos para extraer los m elementos siguientes del vector
 - 2 se comienza a transferir ciclo a ciclo los m elementos obtenidos en la fase anterior



Acceso asíncrono

- ▶ Solicitar los elementos de que consta el vector a cada uno de los m bancos de forma periódica con periodos de T_a con un desfase entre bancos consecutivos de $T_{\text{elemento ciclo}}$

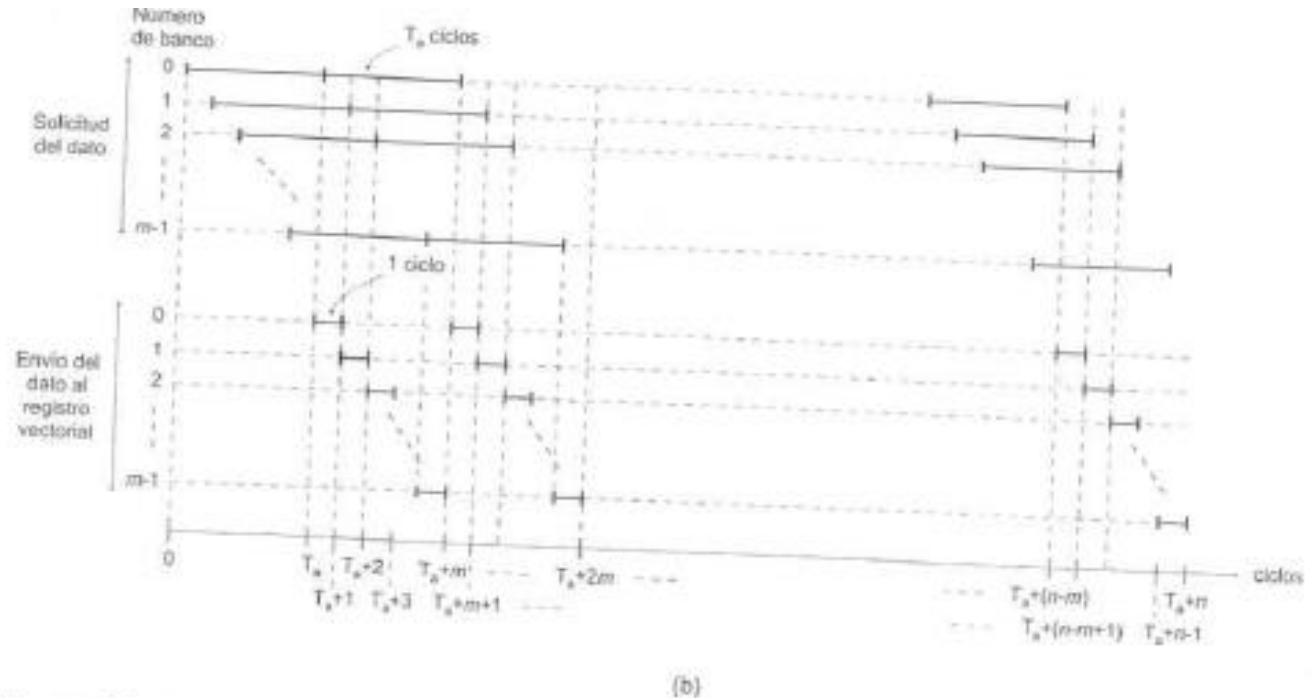


Figura 3.31: Acceso sincrónico (a) y asíncrono (b) a los bancos de memoria para transferir un vector a un registro vectorial mediante una operación de carga. Se considera $T_a = m$.

Almacenamiento

- ▶ 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 palabras en bytes
- ▶ El banco de memoria en que se encuentra está determinado por los bits de orden inferior de la dirección de memoria

136: 0x10001000 (banco 1)
144: 0x10010000 (banco 2)
152: 0x10011000 (banco 3)
160: 0x10100000 (banco 4)
168: 0x10101000 (banco 5)
176: 0x10110000 (banco 6)
184: 0x10111000 (banco 7)
192: 0x11000000 (banco 0)

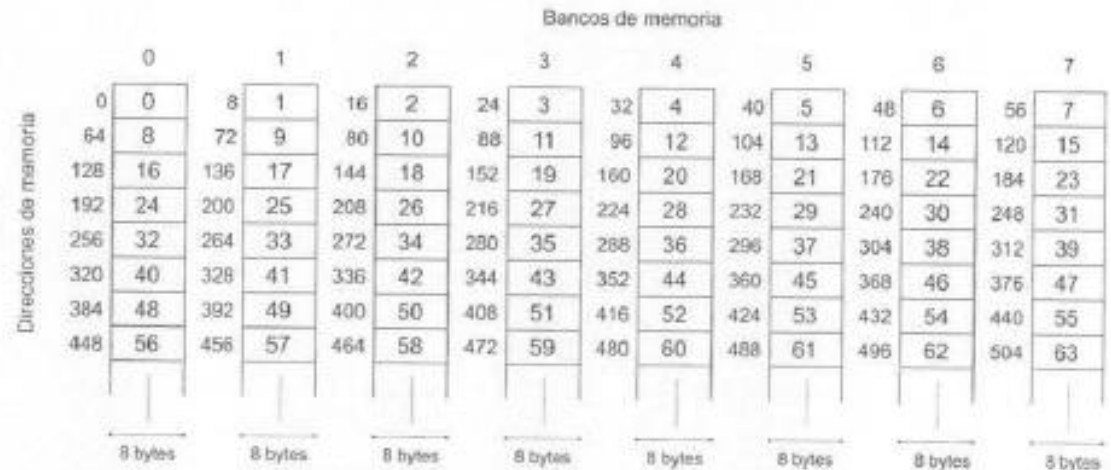


Figura 3.32: Almacenamiento en 8 bancos de un vector de 64 elementos de doble precisión a partir de la dirección de memoria 0. Los números contenidos en las celdas representan el orden del elemento en el vector, no su valor.

Comienzo no en banco cero

- ▶ Comenzando en la dirección 80 con T_a de 6 ciclos
- ▶ $80 = 0x01010000$, el banco 2

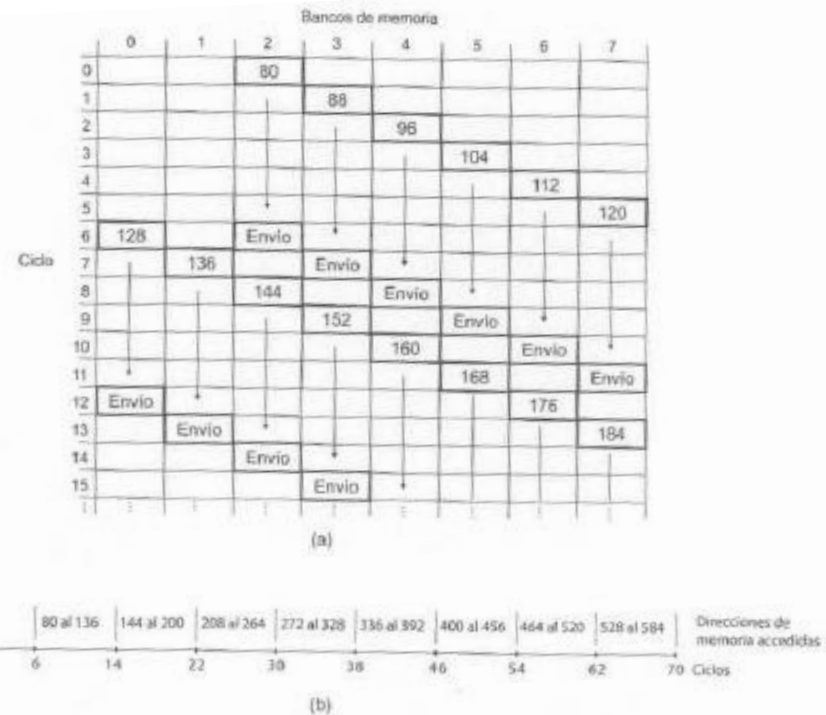


Figura 3.33: Esquema de la transferencia de un vector de 64 elementos ubicado a partir de la posición de memoria 80 en un sistema de memoria organizado en 8 bancos con acceso asincrónico y con T_a de 6 ciclos (a). Relación temporal entre los accesos a los bancos y las palabras enviadas (b).

3.17 Medidas del rendimiento de un bucle vectorizado

- ▶ VLR (vector Length Register)
- ▶ MVL (maximun Vector Length)
- ▶ Strip mining = troceado del vector

```
1: primero:=1;
2: secciones:=n/MVL;
3: VLR:=n mod MVL;
4: for (i=0;i<=secciones;i++)
5:   último:=primero+VLR-1;
6:   for (j=primero; j<=último;j++)
7:     Y[i]=a*X[i]+Y[i];
8:   end for;
9:   primero:=primero+VLR;
10:  VLR:=MVL;
11: end for;
```

% Primer elemento de la sección
% Secciones de MVL elementos
% Longitud de la primera sección
% Bucle exterior
% Último elemento de la sección
% Bucle interior
% Operaciones vectoriales
% Primer elemento de la nueva sección
% Inicio longitud de la nueva sección

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.

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

Ejemplo

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

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.

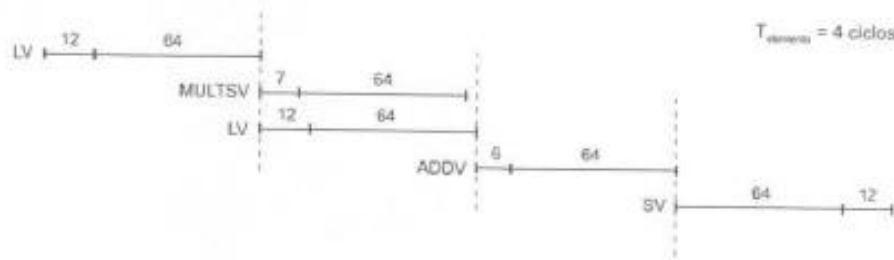


Figura 3.34: Secuencia de ejecución de los cuatro convoyes con VLR = 64.

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

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

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

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

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

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

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

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.

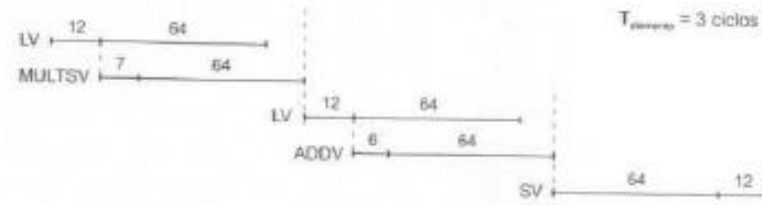


Figura 3.35: Secuencia de ejecución de los tres convoyes con WLR = 64

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 $n = 1000$

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

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

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

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:

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

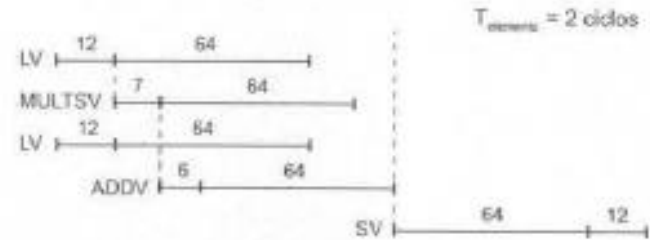


Figura 3.36: Secuencia de ejecución de los dos convoyes con VLR = 64.

el tiempo de arranque total de las unidades funcionales es

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

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

Se tiene ahora

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

y para $n = 1000$

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

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

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

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.

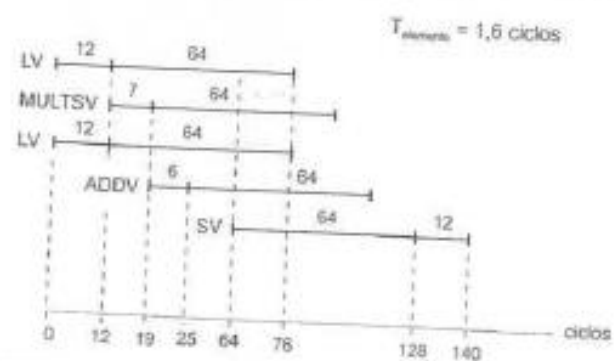


Figura 3.37: Secuencia de ejecución de los dos convoyes con VLR = 64.