3. 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 Instruction Word - Palabra de Instrucción Muy Larga)
Un procesador VLIW se caracteriza por emitir en cada ciclo de reloj una única instrucción pero que contiene varias operaciones o mini-instrucciones tipo RISC.

- Ejecutar en paralelo es responsabilidad exclusiva del compilador.

3.3. El concepto arquitectónico VLIW

El hardware no tiene que intervenir para descubrir el paralelismo entre instrucciones, ya que es responsabilidad exclusiva del compilador el generar un código binario formado por instrucciones que comporten operaciones paralelas.

Planificación estática → VLIW la realiza vía software → la complejidad hardware se reduce y se desplaza hacia el software.

.

VLIW han fracasado	• La incapacidad para desarrollar compiladores que aprovechen al máximo las características del enfoque VLIW. • Los problemas de compatibilidad entre generaciones de procesadores VLIW.

3.4. Arquitectura de un procesador VLIW genérico

La principal diferencia con respecto a un procesador superescalar es la ausencia de los elementos necesarios para la distribución, emisión y reordenación de las instrucciones.

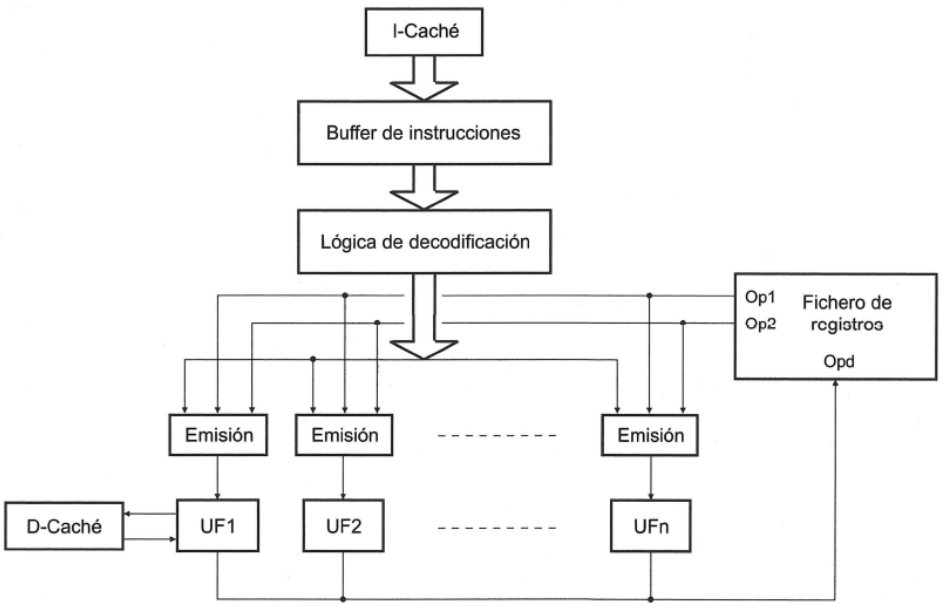


Figura 3.2: Arquitectura básica de un procesador VLIW genérico.

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 y/o de control entre ellas.

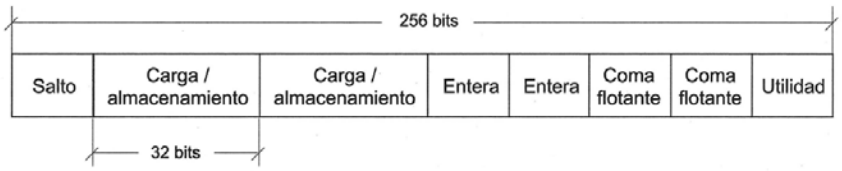


Figura 3.3: Formato de instrucción del computador Multiflow Trace 7.

- Por lo general, el número y tipo de operaciones que contiene una instrucción VLIW se corresponde con el número y tipo de unidades funcionales existentes en el procesador.
- 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.

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
```

El no poder rellenar completamente una instrucción VLIW tiene dos consecuencias:	
	• El código objeto de un procesador VLIW es de mayor tamaño que el equivalente para un procesador superescalar.
	• No se aprovechan al máximo los recursos del procesador ya que se tienen unidades funcionales ociosas, los buses transmiten información carente de valor y las memorias cachés pierden parte de su eficiencia.

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.

3.5. Planificación estática o basada en el compilador

El compilador VLIW recibe como entrada el código fuente de una aplicación, realiza una serie de tareas encaminadas a optimizar el código.

Estas tareas pasan por producir tres elementos	
	• Un código intermedio
	• Un grafo del flujo de control
	• Un grafo del flujo de datos

- 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.

Un ***bloque básico*** → se compone de un grupo de instrucciones que forman una línea de ejecución secuencial por lo que en su interior no existen instrucciones de salto con la salvedad de la última: no hay puntos intermedios de entrada y salida.

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

Una vez que se conocen los bloques básicos que hay en el programa, las instrucciones de cada bloque se combinan para formar instrucciones VLIW. Para ello se recurre al grafo de flujo de datos que tiene asociado cada bloque.

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

De esta forma, el grafo de flujo de datos muestra las secuencias de instrucciones que no presentan dependencias entre ellas y, por lo tanto, son susceptibles de combinar para formar instrucciones VLIW. A la combinación de instrucciones de un único bloque básico para producir instrucciones VLIW se le denomina planificación local.

Algunas técnicas basadas en la planificación local	• Desenrollamiento de bucles (Apdo 3.6)
	• La segmentación software (Apdo 3.7)

- Planificación local está limitada.

Planificación global → combinar instrucciones de diferentes bloques básicos con el fin de producir una planificación con mayor grado de paralelismo.

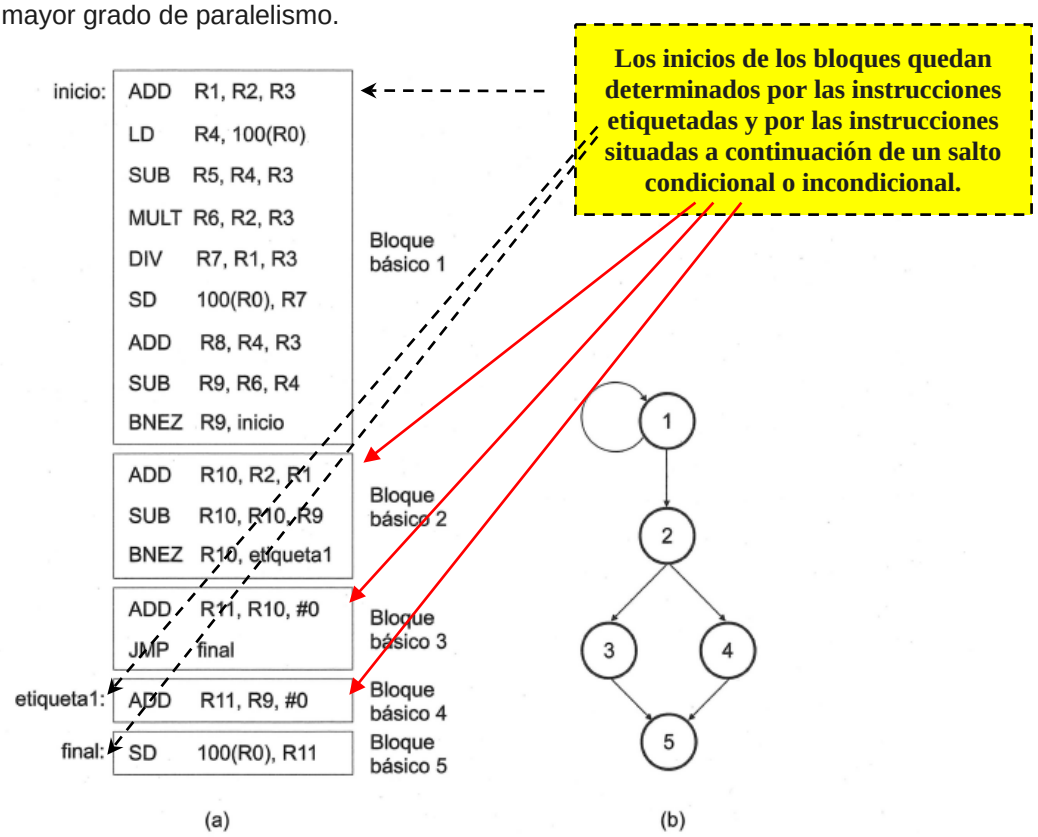


Figura 3.4: Ejemplo de secuencia de código intermedio con delimitación de los bloques básicos (a) y diagrama de flujo de control en el que se muestra la secuencia de ejecución de los bloques y la existencia de bucles (b).

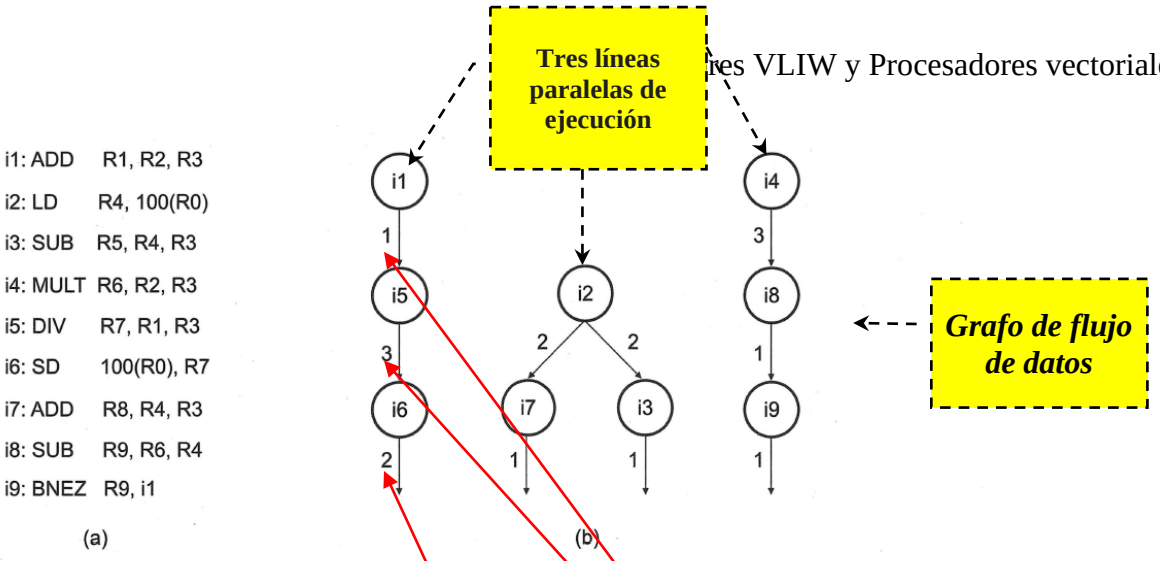


Figura 3.5: Secuencia de operaciones de un bloque básico (a) y diagrama de flujo de datos de la secuencia de instrucciones (b).

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

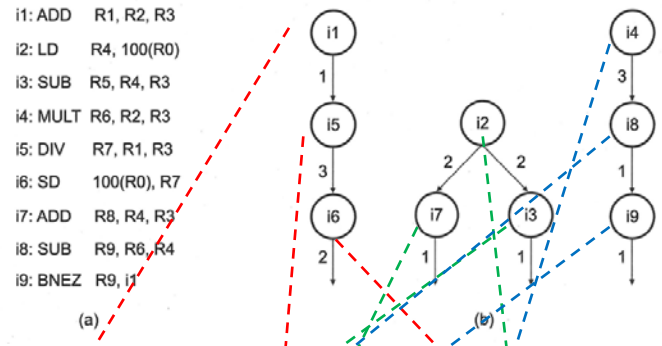


Figura 3.5: Secuencia de operaciones de un bloque básico (a) y diagrama de flujo de datos de la secuencia de instrucciones (b).

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

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

- No es posible ocupar todas las operaciones.

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. Desenrollamiento de bucles

loop unrolling → permite aprovechar el paralelismo existente entre las instrucciones que componen el cuerpo de un bucle. La técnica consiste en replicar múltiples veces el cuerpo del bucle utilizando diferentes registros en cada réplica y ajustar el código de terminación en función de las veces que se replique el cuerpo.

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 desenrollar 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
```

```
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
```

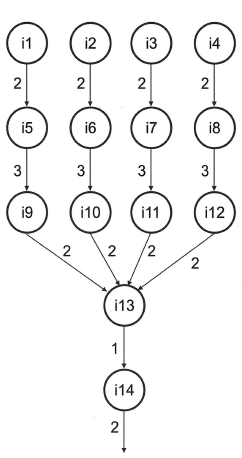
Decremento adelantado

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

← 4 bytes →



- 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.

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 → 9 instrucciones*12 bytes = **108 bytes**

Operaciones originales → 14 instrucciones*4 bytes = **56 bytes**

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**

- Bucle original sin aplicar ***NADA*** 1000 veces → 1000 iteraciones*5 instrucciones = 5000 ciclos
  **** En el mejor de los casos, supuesta una segmentación ideal y sin riesgos.***
- El cuerpo del bucle desenrollado cuatro veces → 250 iteraciones*14 instrucciones= 3500 ciclos.
- El VLIW → 250 iteraciones*9 instrucciones = 2250 ciclos

$$\frac{3500 - 2250}{2250} = 55{,}55\%\ \ más\ rápido$$

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

	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

2 ciclos
3 ciclos
1 ciclo

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

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

3.7. Segmentación software

La técnica 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).

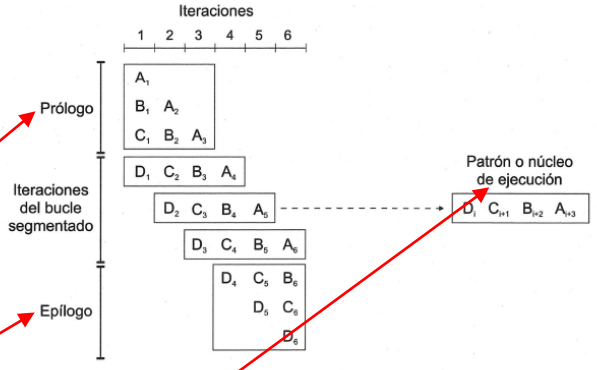


Figura 3.9: Aplicación de la segmentación software al cuerpo de un bucle genérico.

- 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.
- ***Prólogo*** del nuevo bucle.
- ***Epílogo.***

Planificación policíclica → mediante esta técnica se ejecutan al mismo tiempo instrucciones provenientes de múltiples iteraciones.

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

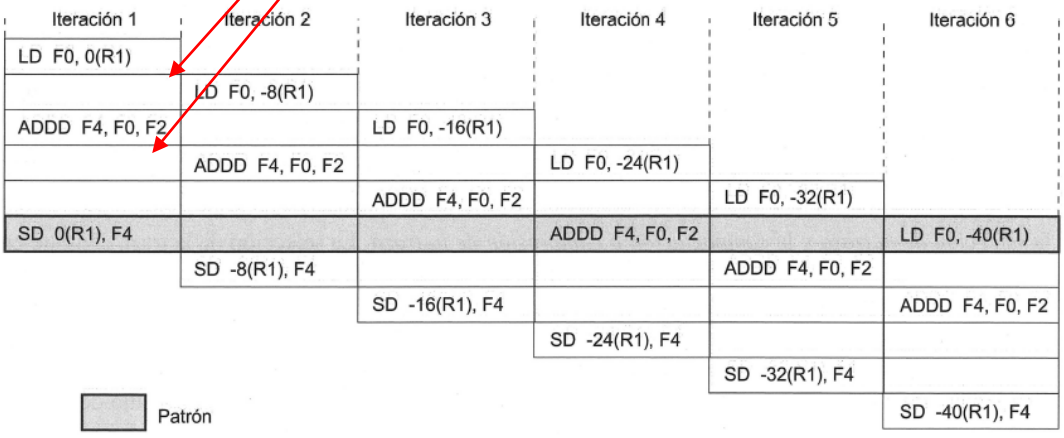


Figura 3.10: Esquema del patrón de ejecución obtenido al aplicar la técnica de segmentación software.

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

Si las **instrucciones** VLIW son de 16 bytes, el tamaño total del código es de (11 inst.*16 byt/inst).= 176 bytes.

En lo referente al **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.

3.8. Planificación de trazas (*trace scheduling*)

Es una técnica de planificación global.

Planificación de trazas	1. Selección de la traza. **Traza** = camino de ejecución más probable.	Encontrar un conjunto de bloques básicos que formen una secuencia de código sin bucles.
	2. Compactación o compresión de la traza	

```
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.

8 bytes

4 bytes

- 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

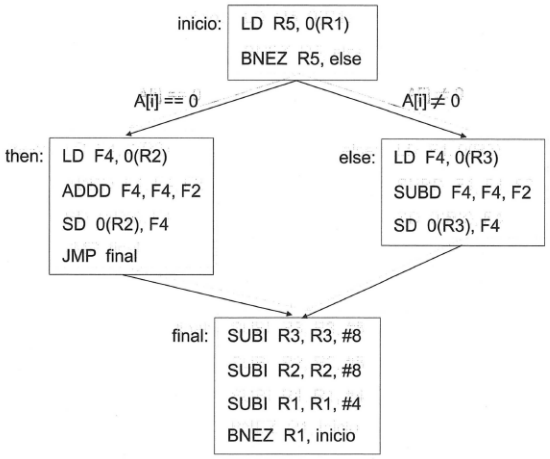
Código intermedio

R2 puntero a X[i]

R3 puntero a Y[i]

R1 puntero a A[i]

```
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
```



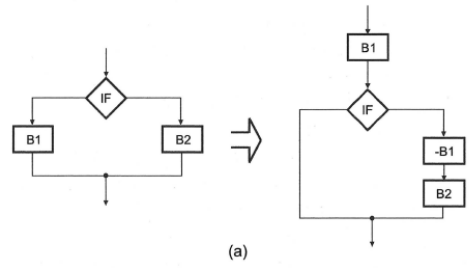
3 U.U.F.F. con ciclos de latencia:

	2 Operaciones Carga / Almacenamiento	3 Operaciones Coma flotante	1 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

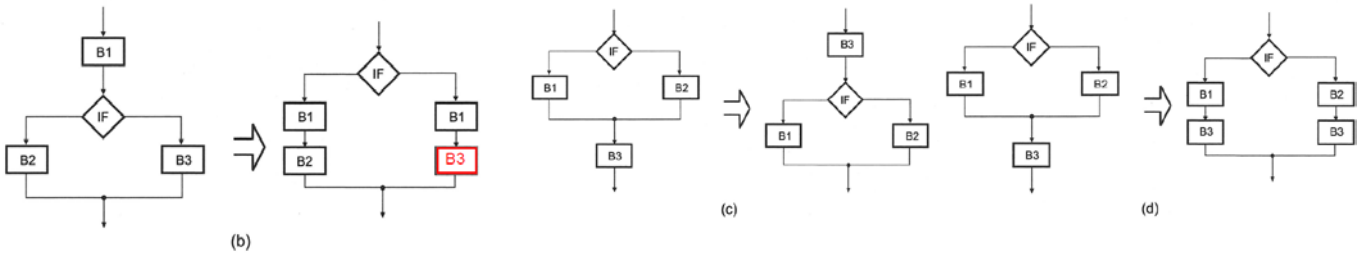
Ruta de ejecución más probable: (caso A[i] == 0) → 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16, 17 → 12 ciclos

Ruta menos probable: (caso A[i] <> 0) → 1, 2, 3, 4, 10, 11, 12, 13, 14, 15, 16, 17 → 12 ciclos

Algunos de los posibles desplazamientos de operaciones que se pueden realizar dentro de una traza



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.



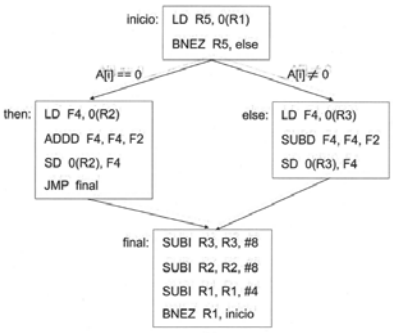
Situaciones en las que un bloque que se encuentra en un punto de ejecución común puede desplazarse con el objetivo de producir código más compacto.

Las consideraciones que debe realizar el compilador para desplazar operaciones dentro de una traza son las siguientes:

- Conocer cuál es la secuencia de ejecución más probable.
- Conocer las dependencias de datos existentes para garantizar su mantenimiento.
- 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.

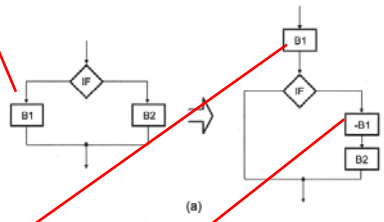
Ejemplo

```
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 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.

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***.

3.9. Operaciones con predicado

Tratar las instrucciones condicionales como cualquier otra instrucción

Operaciones con predicado (predicated operations) = ***operaciones condicionadas (conditioned operations)*** = ***operaciones con guarda guarded operations***

Una operación con predicado una instrucción en la que su resultado se almacena o se descarta dependiendo del valor de un operando que tiene asociado.

Operando = predicado → 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 → Instrucción (*p*)

- *p = 1* → *resultado de la operación se almacena*
- *p = 0* → *resultado de la operación se anula*

Ejemplo

Poner a "1" P1 o P2 en función de la comparación de R1 y el nº #50

```
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

```
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
```

Almacena el resultado de ADDI si P1 = "1"

(c) 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)

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

(b)

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.

Ejemplo

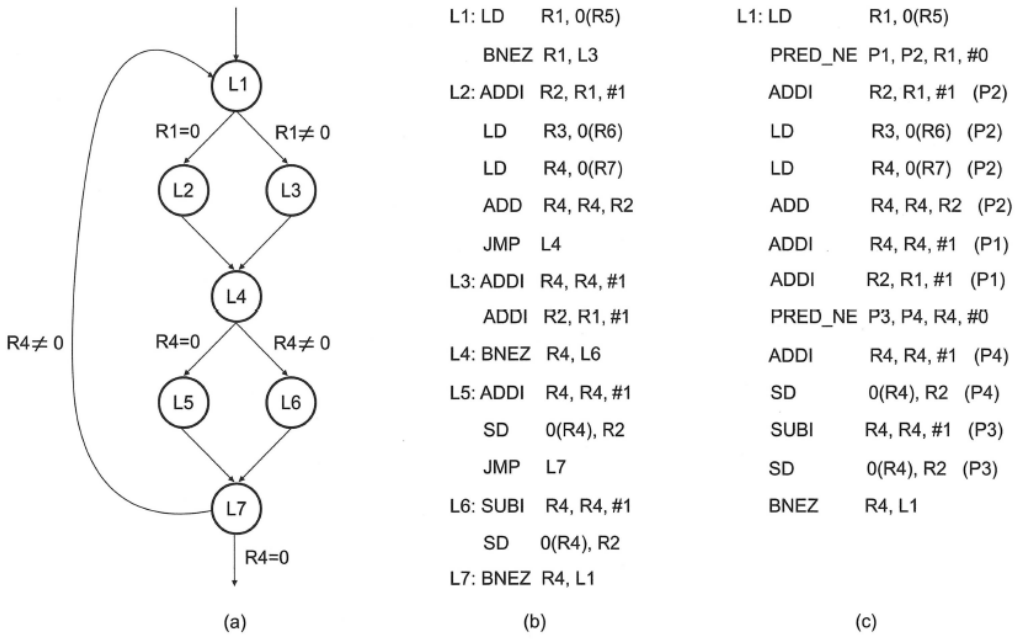


Figura 3.18: Ejemplo de agrupamiento de bloques básicos mediante la transformación de estructuras *if-then* en operaciones con predicado.

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 una etiqueta y en el lugar del programa en el que estaba el código especulado que ha sido desplazado sitúa un centinela vinculado a esa etiqueta.

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

Problemas del VLIW puro	Los repertorios de instrucciones VLIW no son compatibles entre diferentes implementaciones.
	La planificación estática de las instrucciones de carga por parte del compilador resulte muy complicada.
	La importancia de disponer de un compilador que garantice una planificación óptima del código de forma que se maximice el rendimiento del computador y se minimice el tamaño del código.

EPIC (*Explicitly Parallel Instruction Computing*)
El objetivo de EPIC es retener la planificación estática del código pero mejorarla con características arquitectónicas que permitan hacer frente dinámicamente a diferentes situaciones, tales como retardos en las cargas o unidades funcionales nuevas o con diferentes latencias.
Es el ***compilador*** el que determina al ***agrupamiento*** de ***instrucciones*** pero, a la vez, ***comunica*** de forma explícita en ***el propio código cómo se ha realizado el agrupamiento***.

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

A :=B+C

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

Procesador VLIW	• Dos unidades de carga/almacenamiento (2 ciclos de latencia) • Una unidad funcional para operaciones en coma flotante (2 ciclos de latencia) • Una unidad para operaciones enteras y saltos (1 ciclo de latencia).

Código producido por el compilador

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

Código VLIW

De las 20 operaciones | 16 NOPs 4 Útiles

Se puede emitir en paralelo con ella la siguiente instrucción que aparezca en el código

NO se puede emitir en paralelo con ella la siguiente instrucción que aparezca en el código

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

(b)

Código EPIC

Características arquitectónicas más destacadas del estilo EPIC	• Planificación estática con paralelismo explícito. • Operaciones con predicado. • Descomposición o factorización de las instrucciones de salto condicional. • Especulación de control. • Especulación de datos. • Control de la jerarquía de memoria.

3.12. Procesadores vectoriales

Son máquinas que permiten la manipulación de vectores → Operadores fuente y destino son vectores.

Sumar dos vectores A y B de 64 elementos, la suma del elemento A[i] al B[i] no implica ningún tipo de dependencia con la operación previa, es decir, la suma del elemento A[i-1] al B[i-1], o con la suma del A[i-2] al B[i-2], etc.

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

Cálculo con un bucle

→ 1. Cálculo vectorial.

3.13. Arquitectura vectorial básica

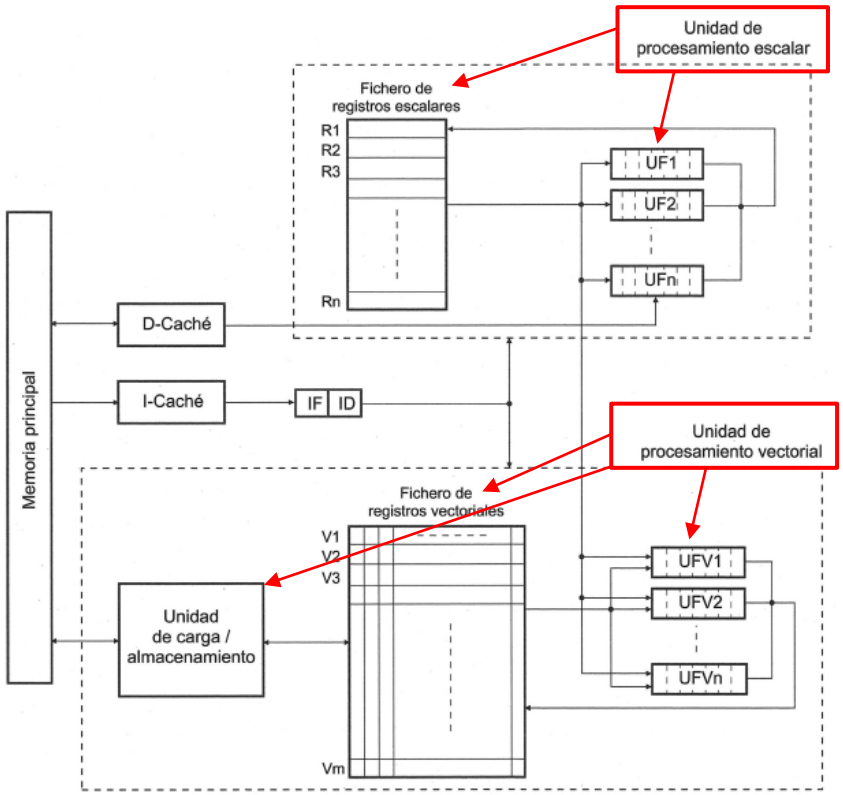


Figura 3.20: Esquema de un procesador vectorial básico.

Características del fichero de registros vectoriales	El número de registros vectoriales.	64 ÷ 256
	El número de elementos por registro.	MVL (*Maximum Vector Length*- Máxima Longitud del Vector). 8 ÷ 256
	El tamaño en bytes de cada elemento.	8 bytes

- Las unidades funcionales vectoriales están segmentadas y pueden iniciar una operación en cada ciclo de reloj.
- Son necesarios mecanismos hardware que detecten los riesgos estructurales y los riesgos de datos que pueden aparecer entre las instrucciones vectoriales y escalares
- Las unidades funcionales vectoriales de algunos procesadores es que pueden estar internamente organizadas en varios ***lanes*** o ***carriles*** con el objeto de aumentar el número de operaciones que pueden procesar en paralelo por ciclo de reloj.
    - Básicamente, una unidad funcional con *n* carriles implica que, internamente, el hardware necesario para realizar la operación se ha replicado *n* veces de forma que el número de ciclos que se emplea en procesar un vector se reduce por un factor de *n*.

Ejemplo

C: =A+B A, B, C son registros vectoriales de 20 elementos.

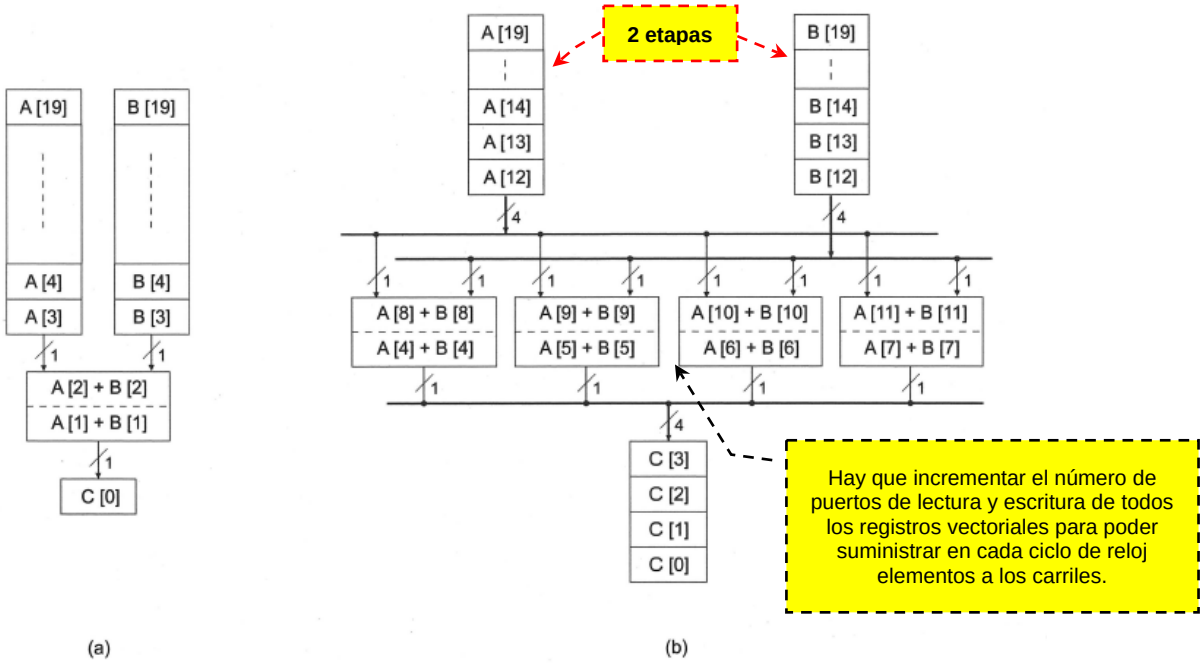


Figura 3.21: Diferencia entre realizar la suma de 2 vectores de 20 elementos en una unidad funcional con 2 etapas y un único carril (a) y en otra unidad funcional de 2 etapas pero con 4 carriles (b).

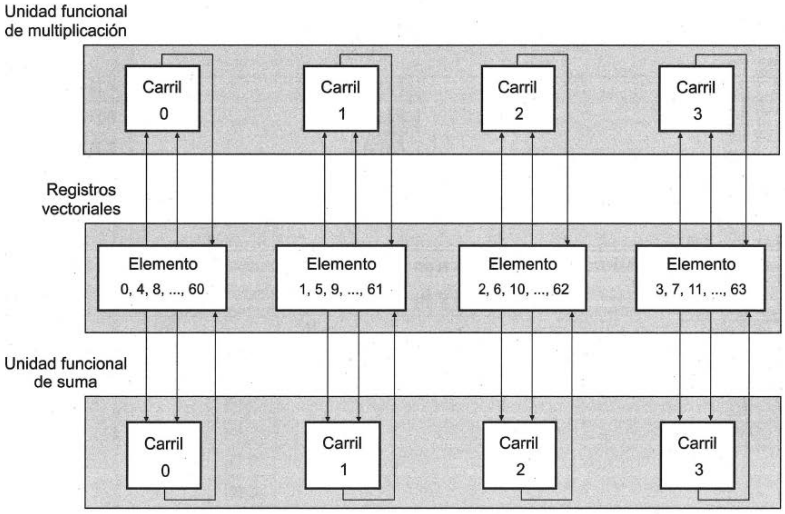


Figura 3.22: Esquema de unidades vectoriales de suma y multiplicación en coma flotante con cuatro carriles y distribución de los elementos de los registros vectoriales entre ellas.

Otra estructura que aprovecha el paralelismo de datos es → **Procesador matricial:**
Mal llamado procesador vectorial

- Se engloban dentro de la categoría SIMD (Single Instruction -Multiple Data).
- Cuenta con una unidad de procesamiento vectorial, una unidad escalar y una unidad de control que discrimina según el tipo de instrucción.
- La principal diferencia es que el procesador matricial con *n* elementos de procesamiento puede procesar de forma simultánea en el mismo ciclo de reloj un vector de *n* elementos

3.14. Repertorio genérico de instrucciones vectoriales

Similar al de las operaciones escalares

Instrucción vectorial

`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]`.

Ejemplo del bucle DAXPY (*Double Precision A Times X Plus Y*)

Y(i) = a*X(i) + Y(i) con vectores de 64 elementos de 8 bytes

Código escalar:

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

Se considera que la longitud de los vectores es de 64 elementos, en la aproximación escalar se ejecutan 578 instrucciones (2 + 9 * 64).

Código vectorial:

```
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]
```

CUESTIONES

¿Cómo proceder cuando la longitud del vector es diferente al número de elementos de un registro vectorial?
¿Qué sucede cuando los elementos del vector se almacenan de forma uniforme pero no consecutiva en memoria?
¿Cómo se vectoriza el cuerpo de un bucle que contiene instrucciones ejecutadas condicionalmente?

- El registro de longitud vectorial VLR (*Vector Length Register*) → Controla la longitud de cualquier operación vectorial.
- El registro de máscara VM (*Vector Mask*).

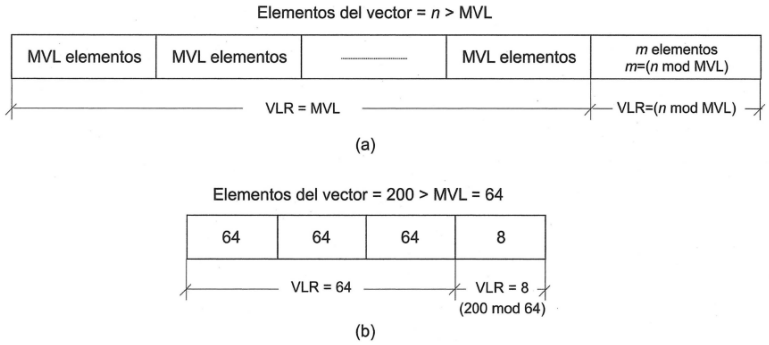


Figura 3.24: Seccionamiento de un vector genérico en secciones de MVL elementos (a) y de un vector de 200 elementos en secciones de 64 (b).

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

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

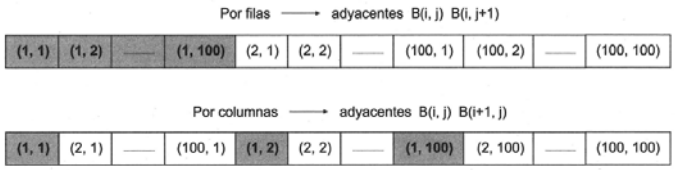


Figura 3.25: Almacenamiento unidimensional por filas o por columnas de una estructura bidimensional de 100x100.

SOLUCIÓN:
Instrucciones especiales →

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

Vectorizar bucles en cuyo cuerpo hay instrucciones ejecutadas condicionalmente. Dependiendo de la condición, ciertos elementos de un vector no tengan que ser manipulados. → 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). → Instrucciones especiales

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

Ejemplo

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

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

3.15. Medida del rendimiento de un fragmento de código vectorial

Factores que afectan al tiempo de ejecución

T_n

- La latencia en producir el primer resultado o tiempo de arranque.
- El número de elementos a procesar por la unidad funcional.
- El tiempo que se tarda en completar cada resultado o tiempo por elemento.

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

$T_{arranque}$ → El tiempo de arranque es similar al número de segmentos de que constan ya que es el tiempo que transcurre desde que se solicita el primer bloque de datos del vector hasta que se dispone del mismo para ser tratado por la unidad funcional.
Número de etapas en las que está segmentada la unidad funcional.
Unidad vectorial de 6 etapas (1 ciclo/etapa) → 6 ciclos.

$T_{elemento}$ → El tiempo que se tarda en completar cada uno de los restantes n elementos (Unidad segmentada→1 ciclo).

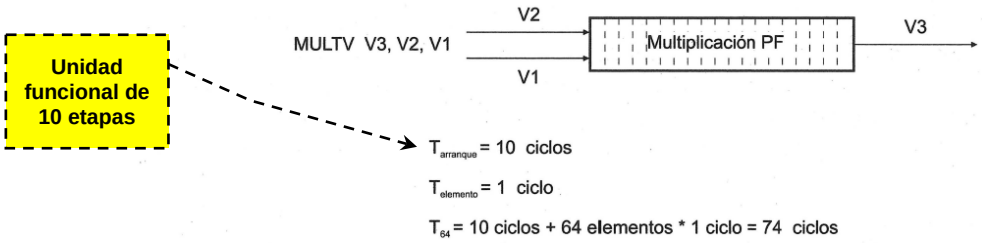
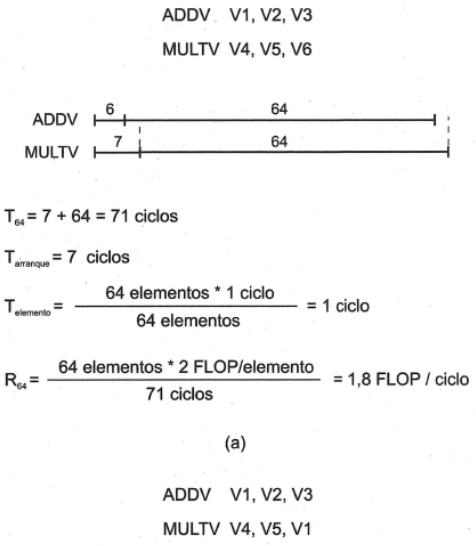


Figura 3.26: Tiempo de ejecución de una operación de multiplicación sobre 2 vectores de 64 elementos.

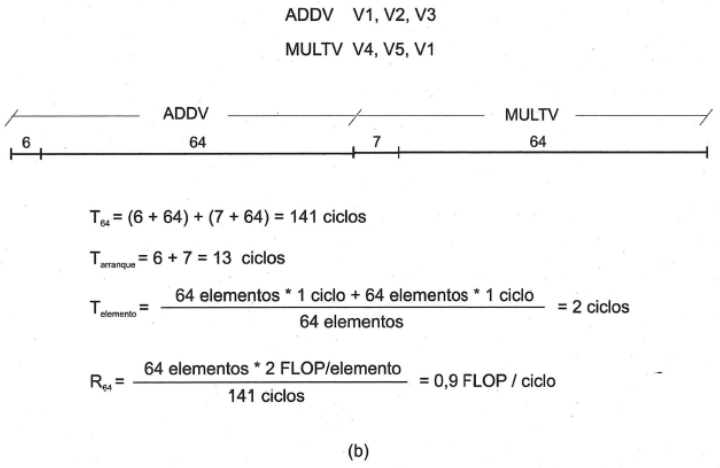
Situaciones	Si no hay riesgos estructurales ni dependencias verdaderas	→	Planificación por *convoy* o paquetes
	Si hay riesgos o interdependencias	→	Esperar a que los operandos estén disponibles.

3.15.1.1. Rendimiento de un procesador vectorial en FLOPs por ciclo

R_n = (Operaciones en coma flotante * n elementos) / T_n → A mayor Rn mayor rendimiento



Ejecución de un convoy de 2 instrucciones



2 convoyes de una única instrucción (b)

Encadenamiento (*chaining*) → El encadenamiento permite que una unidad funcional pueda comenzar a operar tan pronto como los resultados de la unidad funcional de que depende estén disponibles.

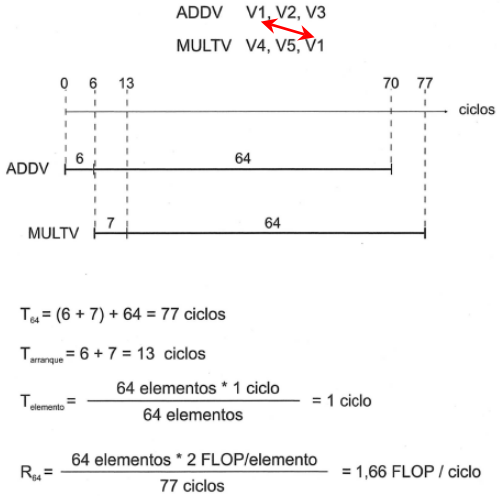
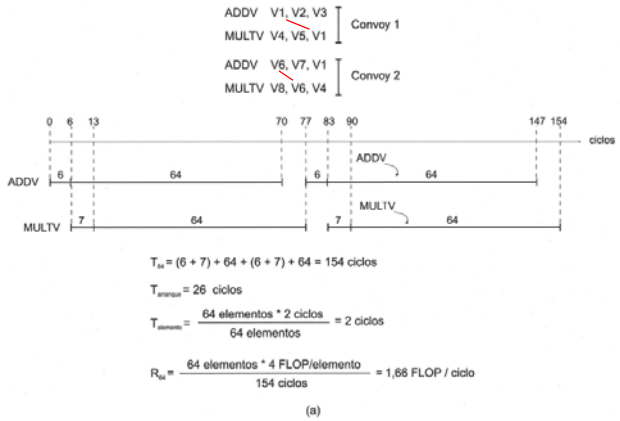
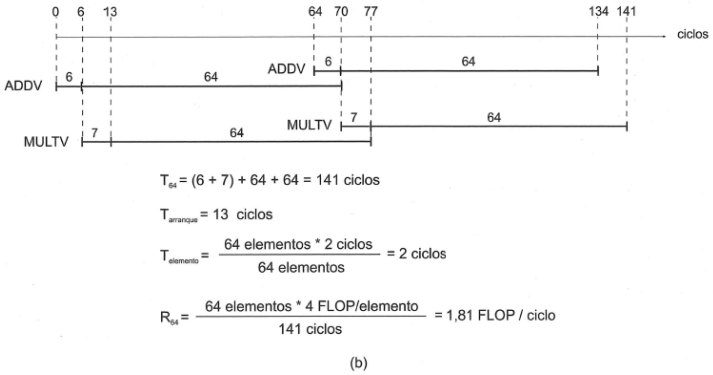


Figura 3.28: Ejecución de dos instrucciones con encadenamiento entre las unidades funcionales.





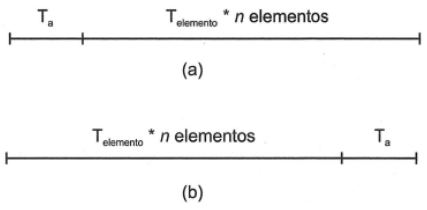
Convoyes sin solapamiento Convoyes con solapamiento

Ejecución de dos convoyes de instrucciones con encadenamiento entre unidades.

3.16. La unidad funcional de carga/almacenamiento vectorial

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

Solución → organizando físicamente la memoria en varios bancos o módulos y distribuyendo el espacio de direccionamiento de forma uniforme entre todos ellos.

<table>
<tr><td>Operación de carga de un registro vectorial</td><td>• Tiempo de arranque es igual a Ta ya que es el tiempo que transcurre desde que se solicita el primer elemento del vector al sistema de memoria hasta que está disponible en el puerto de lectura del banco para su posterior transferencia al registro vectorial.
• El tiempo por elemento se considera como el número de ciclos que se consumen en transferir el dato ya disponible en el banco de memoria al registro vectorial (por lo general, inferior a un ciclo pero se iguala a uno para equipararlo al $T_{elemento}$ de las unidades funcionales).</td><td rowspan="2"> 
el tiempo que consume una operación de acceso a memoria para un vector de n elementos =
$(T_a + n*T_{elemento})$ ciclos</td></tr>
<tr><td>Operación de escritura en memoria</td><td>• 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 se puede ver como el tiempo que emplea el banco en escribir el último elemento del vector en una posición del banco de memoria.</td></tr>
</table>

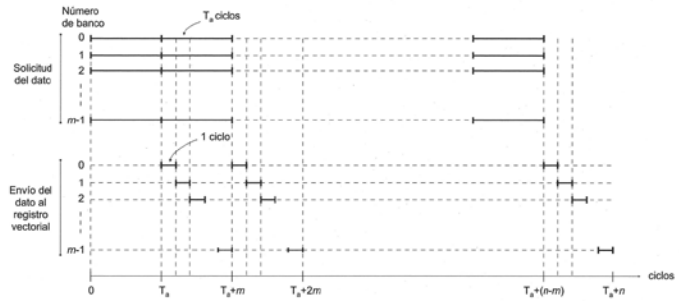
La lectura de los *n* elementos de que consta un vector supondría un coste de $n * T_a$ ciclos, lo que es totalmente **inadmisible**.

Para poder **ocultar** toda esta **latencia** es fundamental **dimensionar** correctamente el **número** de **bancos** de **memoria** de forma que solo se vea el tiempo de acceso correspondiente al primer elemento del vector y que los tiempos de acceso de los demás elementos queden ocultos. Esto se consigue **distribuyendo los *n* elementos** de **un vector entre *m* bancos** de memoria para que se solapen.

Un dimensionamiento correcto → $m \geq T_a$ (T_a en ciclos de reloj)

Solapamientos

- Síncrono
- Asíncrono



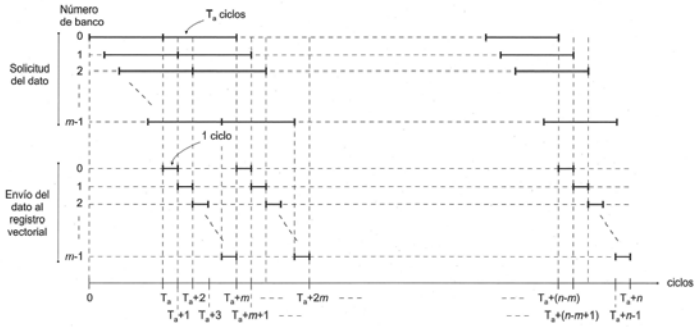
Síncrono

Solicitar simultáneamente un dato a los m bancos cada T_a ciclos.

Realizada la primera petición y transcurridos T_a ciclos estarán disponibles los m primeros elementos del vector para ser transferidos al registro vectorial, lo que consume m*$T_{elemento}$ ciclos.

Ahora, cada T_a ciclos se realizan dos acciones:

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



Asíncrono

Solicitar los elementos de que consta el vector a cada uno de los m bancos de forma periódica con periodo T_a con un desfase entre bancos consecutivos de $T_{elemento}$ ciclos.

De esta forma; comenzando en el ciclo 0 y cada T_a ciclos el banco 0 solicita un dato, en el ciclo 1 y cada T_a ciclos el banco 1 solicita un nuevo dato, y así sucesivamente hasta el banco m-1.

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

1. Efectúa la nueva solicitud de dato.
2. Transfiere el dato ya disponible al registro vectorial.

La dirección de memoria

136: 0x10**001**000 (banco 1)
144: 0x10**010**000 (banco 2)
152: 0x10**011**000 (banco 3)
160: 0x10**100**000 (banco 4)
168: 0x10**101**000 (banco 5)
176: 0x10**110**000 (banco 6)
184: 0x10**111**000 (banco 7)
192: 0x11**000**000 (banco 0)

8 bytes → $2^3 \rightarrow 3$

8 bancos → $2^3 \rightarrow 3$ bits

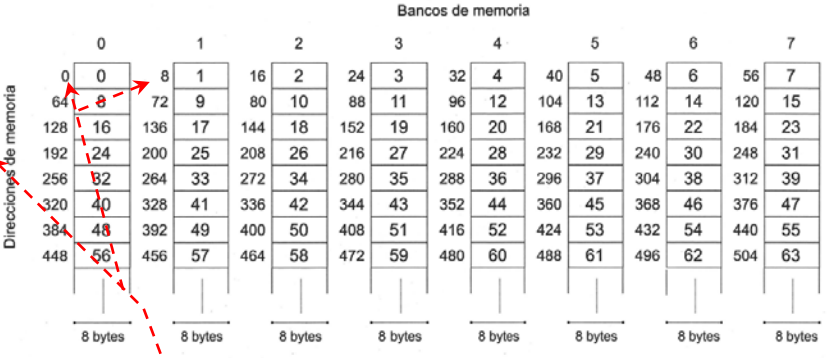
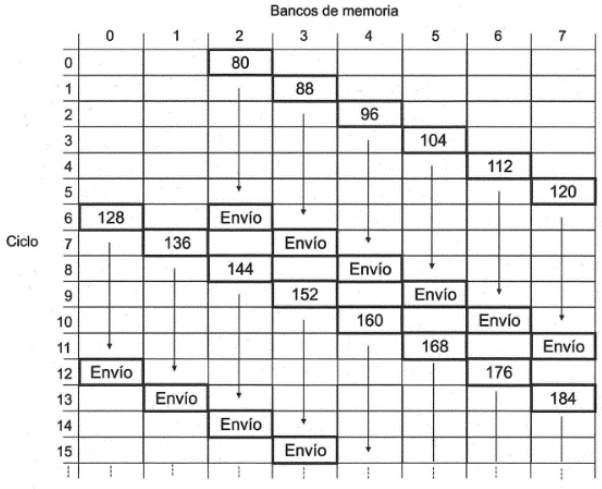


Figura 3.32: Almacenamiento en 8 bancos de un vector de 64 elementos de doble precisión a partir de la dirección de memoria 0. Los números contenidos en las celdas representan el orden del elemento en el vector, no su valor.

64 elementos (de 8 bytes cada uno) ubicado a partir de la posición de memoria 80 en un sistema de memoria organizado en 8 bancos con acceso asíncrono y con T_a de 6 ciclos



Ancho de palabra del banco = 8 bytes

80 - 143	144 - 207	208 - 271	272 - 335	336 - 399	400 - 463	464 - 527	528 - 591	Direcciones de memoria
6	14	22	30	38	46	54	62	70 Ciclos

3.17. Medida del rendimiento de un bucle vectorizado

Bucle DAXPY de *n* elementos con la técnica *strip mining*

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

Nº elementos del vector

Posición en el vector del 1º elemento

Nº total de secciones de MVL elementos

Componentes del coste	T_{base}: Es el tiempo que consumen las instrucciones escalares de preparación antes de abordar el bucle exterior.	Líneas 1, 2 y 3
	T_{bucle}: Son los costes derivados de ejecutar en cada iteración del bucle exterior las instrucciones escalares necesarias para realizar el seccionamiento.	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.

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

Valor entero inmediatamente superior

Otra medida de rendimiento:

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

Ejemplo

Bucle DAXPY para vectores de *n* elementos.

- 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.
- La frecuencia de reloj es 500 MHz.
- VLR = 64

El fragmento de código vectorial que se genera para realizar las operaciones

Y(i): =a*X(i)+Y(i)

```
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
```

	$T_{base} = 10$ ciclos y $T_{bucle} = 15$ ciclos

3.17.1. 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
```

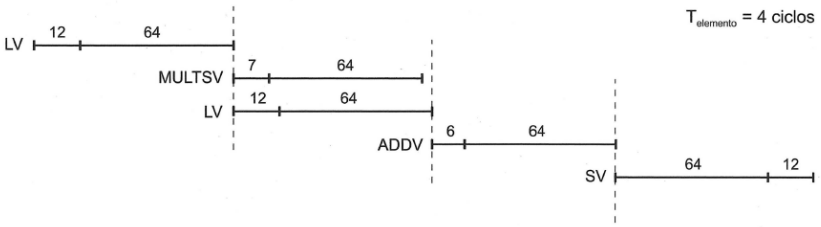


Figura 3.34: Secuencia de ejecución de los cuatro convoyes con VLR = 64.

Dado que hay cuatro convoyes, $T_{elemento}$ es 4 ciclos

$T_{arranque}$ total es igual a la suma de los tiempos de arranque visibles de los cuatro convoyes.

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

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

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

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

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

Para *n*=1000 $\Rightarrow$

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

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

FLOPs/ciclo

Para simplificar los cálculos, la expresión $\lceil n/64 \rceil$ se puede reemplazar por una cota superior dada por (n /64 + 1).
Nº oper. Vectoriales = 2 (1 ADDV y 1 MULTSV)

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

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

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

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

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

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

3.17.2. Caso 2: Con encadenamiento de resultados entre unidades

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

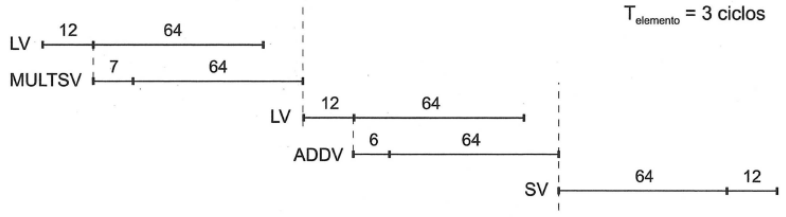


Figura 3.35: Secuencia de ejecución de los tres convoyes con VLR = 64

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

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

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

Con estos valores la expresión del tiempo total de ejecución queda

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

que para el caso particular de $n = 1000$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

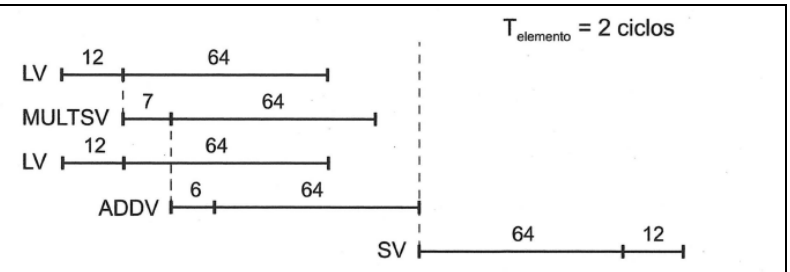


Figura 3.36: Secuencia de ejecución de los dos convoyes con VLR = 64.

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

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

Se tiene ahora

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

y para $n = 1000$

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

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

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

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

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

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

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

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

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

Si se expresa en FLOPS, se obtiene

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

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

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

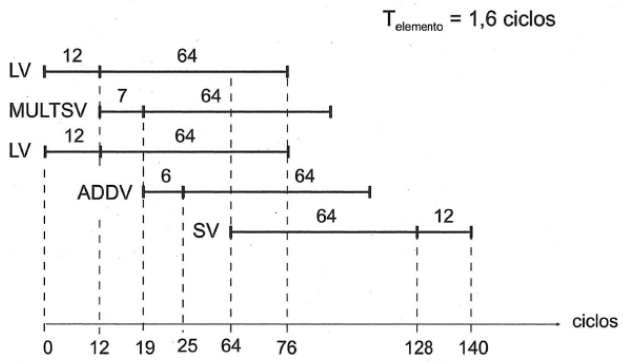


Figura 3.37: Secuencia de ejecución de los dos convoyes con VLR = 64.

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

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

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

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

Se tiene así

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

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

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

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

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

y particularizando para un vector de 1000 elementos

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

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

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

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

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

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

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

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

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

y si se expresa R_∞ en FLOPS se obtiene

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

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