❖Tema 4.1 DLP. Arquitectura Vectorial

➢ 4.1.1 Introducción

Hasta ahora hemos hablado de explotar paralelismo a nivel de Instrucción (ILP, Tomasulo, superescalares) y también el paralelismo a nivel de hilo (TLP, Multithreading).

En esta sección nos introducimos en el paralelismo a nivel de datos, DLP.

SIMD: (***S**ingle **I**nstruction **M**ultiple **D**ata*) una sola instrucción se ejecuta sobre un conjunto amplio de datos. Ej:

$$Suma\ vectorial \quad \vec{C} = \vec{A} + \vec{B}$$

Código secuencial

```
for (i=0;i<N;++i){
      C[i]=A[i]+B[i];
}
```

Código vectorial

```
ADDV C,A,B
```

La operación es única, es una operación de suma, pero se aplica a zonas de memoria con un determinado ancho. Esto puede ser muy útil en aplicaciones con un volumen de cálculo matricial enorme (aplicaciones gráficas o de ingeniería).

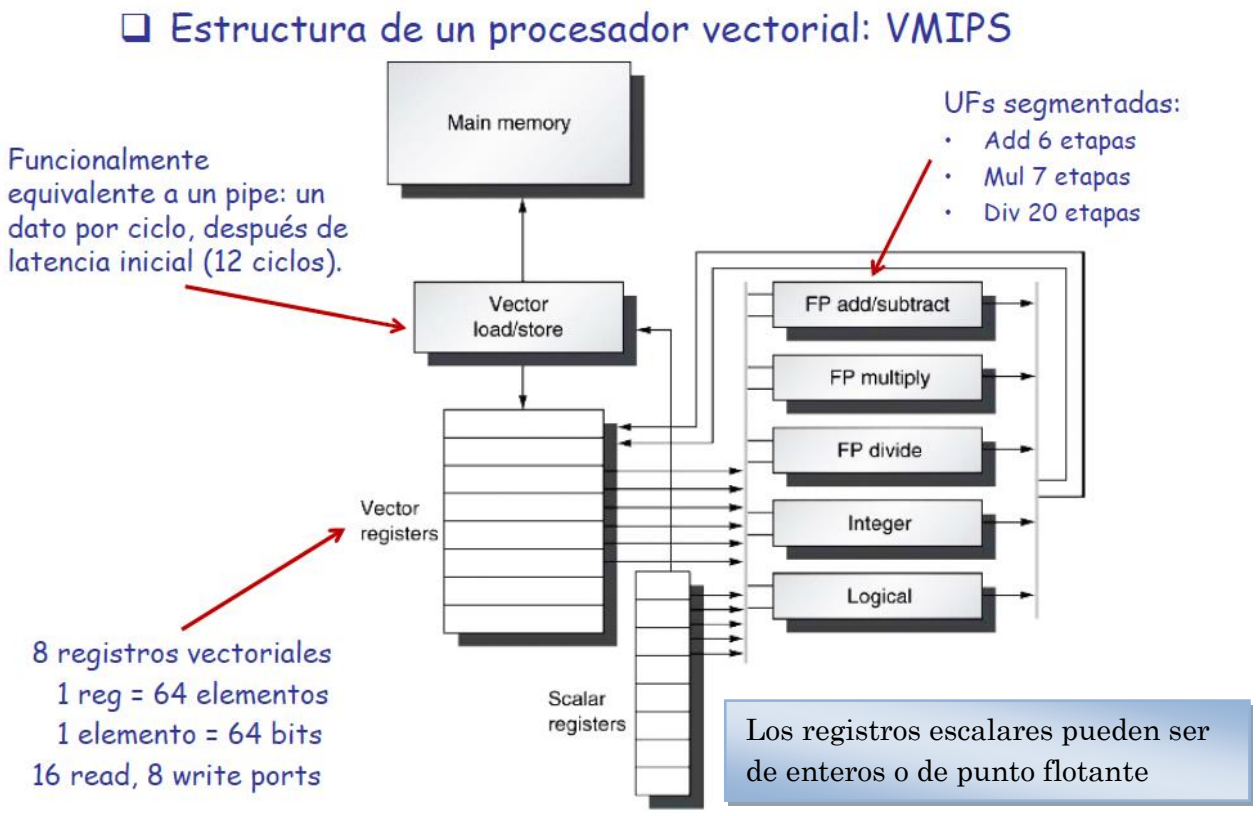
El soporte arquitectónico que usaremos para explotar el paralelismo SIMD tiene 3 vertientes:

- Arquitectura Vectorial.
- Extensiones SIMD (extensiones multimedia).
- GPUs (procesadores gráficos).

➢ 4.1.2 Arquitectura Vectorial

Características Básicas. VMIPS.

1) Lee conjuntos de datos sobre registros vectoriales.
2) Opera sobre los registros vectoriales.
3) Almacena los resultados finales en memoria (usamos los registros vectoriales para ocultar latencias en memoria).
4) Tiene secuencias de cálculos independientes lo que proporciona ausencia de riesgos de datos de todo tipo.
5) Tiene un código compacto, una sola instrucción puede representar muchas operaciones.
6) Se accede a memoria con un patrón conocido y se reducen las instrucciones de salto por lo que hay menos riesgos de control.



Existen además dos **registros especiales** en la arquitectura:

VLR: (*Registro de Longitud Vectorial*) el número indicado en este registro nos determina el tamaño del array que se va a procesar (en esta arquitectura ≤ 64). Si el vector es más largo que 64 elementos se necesita hacer Strip Mining.

VM: (*Máscara Vectorial*) es un registro con el que se pueden evitar operaciones sobre componentes concretas del vector. Es un registro de 64 bits, con un 0 en una posición, esa componente no se procesa. El hecho de que no se procese nos hace perder rendimiento, porque el tiempo se consume igual.

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1, V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e .. R1 + i x R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e .. R1 + i x R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2) ,V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 x R1, 2 x R1, ... ,63 x R1 into V1.
S--VV.D S--VS.D	V1, V2 V1, F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector: otherwise put 0. Put resulting bit vector in vector mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR,R1	Move contents of R1 to vector-length register VL.
MFC1	R1,VLR	Move the contents of vector-length register VL to R1.
MVTM	VM,F0	Move contents of F0 to vector-mask register VM.
MVFM	F0,VM	Move contents of vector-mask register VM to F0.

Las instrucciones tienen como operandos los registros vectoriales V, los registros escalares de punto flotante F, los registros escalares de enteros R y los registros especiales VLR y VM.

En la arquitectura de VMIPS se pueden combinar instrucciones simples con instrucciones vectoriales.

Operación vectorial $\vec{Y} = a * \vec{X} + \vec{Y}$

- Versión escalar

```
        L.D     F0, a          ; load scalar a
        DADDIU  R4,Rx,#512     ; last address to load
Loop:   L.D     F2, 0(Rx)      ; Load X[i]
        MUL.D   F2, F2, F0     ; A x X[i]
        L.D     F4, 0(Ry)      ; Load Y[i]
        ADD.D   F4, F4, F2     ; A x X[i] + Y[i]
        S.D     F4, 0(Ry)      ; Store Y[i]
        DADDIU  Rx, Rx, #8     ; increment index X
        DADDIU  Ry, Ry, #8     ; increment index Y
        DSUBU   R20, R4, Rx    ; loop exhausted?
        BNEZ    R20, Loop
```

- Versión vectorial

```
        L.D     F0, a          ; load scalar a
        LV      V1, Rx         ; Load vector X
        MULVS.D V2, V1, F0     ; Vector-scalar multiply
        LV      V3, Ry         ; Load vector Y
        ADDVV.D V4, V2, V3     ; Vector addition
        SV      V4, Ry         ; Store result Y
```

➢ 4.1.3 Organización y riesgos de VMIPS

Un procesador VMIPS solo dispone de una unidad LOAD/STORE y solo tiene una unidad funcional para cada tipo de operación. Las UFs están segmentadas, pero solo pueden producir el resultado de una componente por ciclo. Una suma de un vector de 64 componentes tarda 64 ciclos (en el mejor de los casos, ya que si tenemos una latencia de ADDD de 6 ciclos hay que rellenar el pipe de la UF).

Pueden existir conflictos estructurales con las UF y también con los puertos del Banco de Registros.

Existen posibles dependencias de datos para instrucciones Vectoriales consecutivas que usen UFs diferentes, por ejemplo miremos el código del apartado anterior donde

```
MULSVS.D   V2,V1,F0
ADDVV.D    V4,V2,V3
```

El vector V2 se usa como destino en la multiplicación y como fuente en la suma. Para resolver este problema necesitamos técnicas de encadenamiento.

Convoy: se denomina así a un conjunto de operaciones vectoriales consecutivas que no tienen ningún riesgo estructural entre ellas, aunque podrían tener riesgos LDE entre sí.

Paso (chime): es la unidad de tiempo que consume un convoy. Con m convoyes consumimos m pasos. Si tenemos vectores de n componentes y ejecutamos m convoyes, se requiere (en aproximación) un tiempo $m \times n$. (Notación: $T_{chime} = m$, que es el número de pasos).

Analicemos el código del ejemplo:

❑ Convoyes: ejemplo

```
1: LV       V1, Rx      ; load vector X
2: MULVS.D  V2, V1, F0  ; vector-scalar multiply
3: LV       V3, Ry      ; load vector Y
4: ADDVV.D  V4, V2, V3  ; Vector addition
5: SV       V4, Ry      ; Store result Y
```

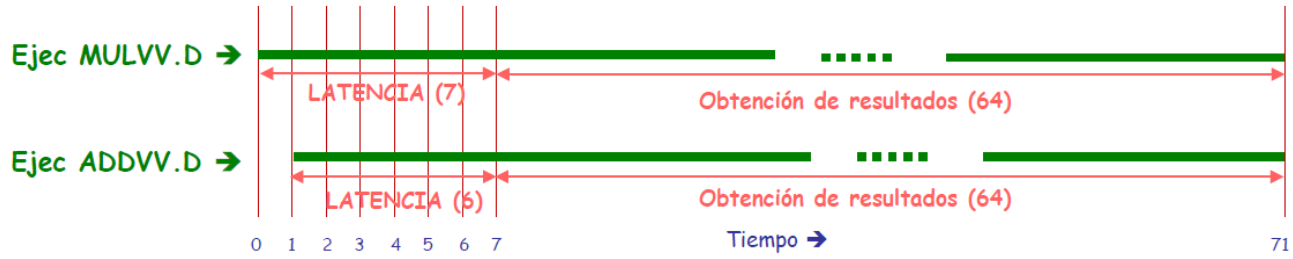
- Conflictos:
  - 1 y 2 no tienen conflictos estructurales
  - 3 tiene conflicto estructural con 1 (una sola unidad de Load/Store)
  - 4 no tiene conflictos estructurales con 3
  - 5 tiene conflicto estructural con 3
- Convoyes resultantes
  - 1. Formado por LV y MULVS.D
  - 2. Formado por LV y ADDVV.D
  - 3. Formado por SV
- Tiempo de cálculo aprox para 64 componentes: 3x64= 192 ciclos (Tchime=3)

➢ 4.1.4 Ejecución de Operaciones aritméticas sin riesgos

```
MULVV.D     V1,V2,V3
ADDVV.D     V4,V5,V6
```

Estas dos operaciones vectoriales forman un convoy. No existe ningún tipo de dependencia estructural entre ellas y en este caso tampoco de datos.

En nuestro VMIPS la instrucción MUL.D tiene una latencia de 7 ciclos y la ADD.D de 6 ciclos.



Debido a la estructura vectorial del computador puedo lanzar las dos instrucciones de forma consecutiva. Cada UF, que está segmentada, va realizando una etapa de cada operación de forma secuencial. Las resultados empiezan a salir en el ciclo 7, se sacan 64 resultados y las dos operaciones vectoriales acaban a la vez.

El VMIPS está preparado para despachar más de un resultado por ciclo de las UFs, el Banco de registros vectorial tiene varios puertos de lectura y escritura.

➢ 4.1.5 Ejecución de Load Vectorial (LV)

Una pregunta interesante que nos puede surgir es cómo se realiza la carga de un vector desde memoria a un registro vectorial, por ejemplo con la instrucción

```
LV      V1, R1
```

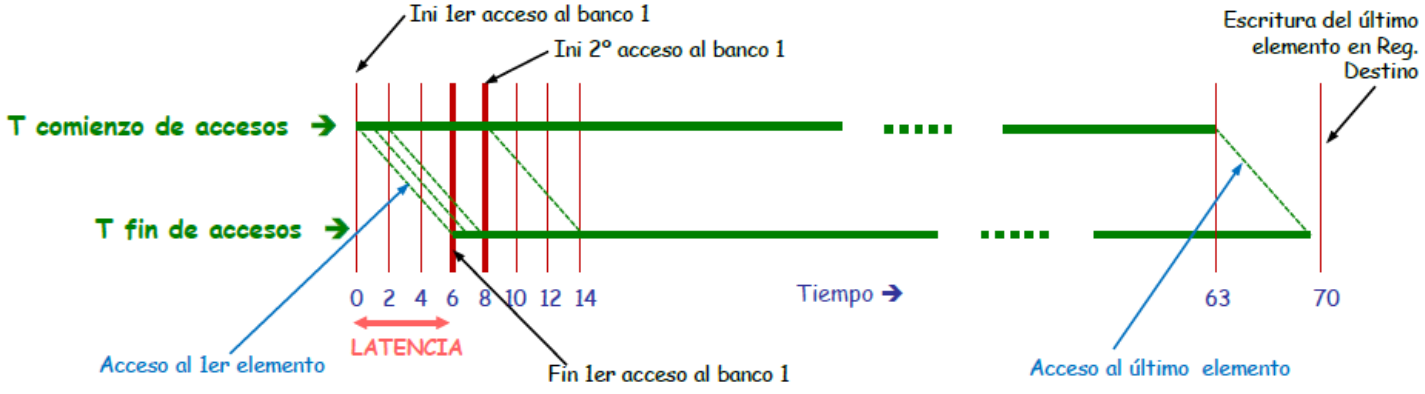
El tiempo de respuesta de una unidad de memoria rara vez es tan rápido como para poder realizarse en un ciclo de reloj. Si suponemos que en cada ciclo de reloj se trae un dato (potencialmente de un total de 64 componentes) el tiempo de carga sería admisible. El problema es que esto no ocurre, en el VMIPS el tiempo promedio de acceso a memoria es de 12 ciclos (desde que se inicia la búsqueda de un dato simple hasta que se escribe finalmente en el Banco de Registros).

Solución. Memoria Entrelazada

La memoria está organizada en bancos. Los bancos se acceden en paralelo con una diferencia de un ciclo entre cada acceso. Si la latencia es de 6 ciclos de reloj para cada banco (es solo de ejemplo, en el VMIPS es de 12 ciclos) podemos dividir la memoria en 8 bancos (del 0 – 7), de esta manera cuando terminamos de acceder al banco 7 podemos volver a acceder al banco 0 que quedó libre.

Supongamos que queremos acceder a un vector de 64 componentes almacenado a partir de la dirección 136 (en decimal). Su primera componente será requerida al banco 1 que inicia la transferencia en el ciclo 0 y la termina en el ciclo 6. Sin embargo inmediatamente después de iniciar la transferencia de la primera palabra, se inicia la segunda que estará en el banco 2. De esta manera el banco 1 no recibe otra petición hasta 8 ciclos después… hemos ocultado así la latencia de acceso de la memoria.

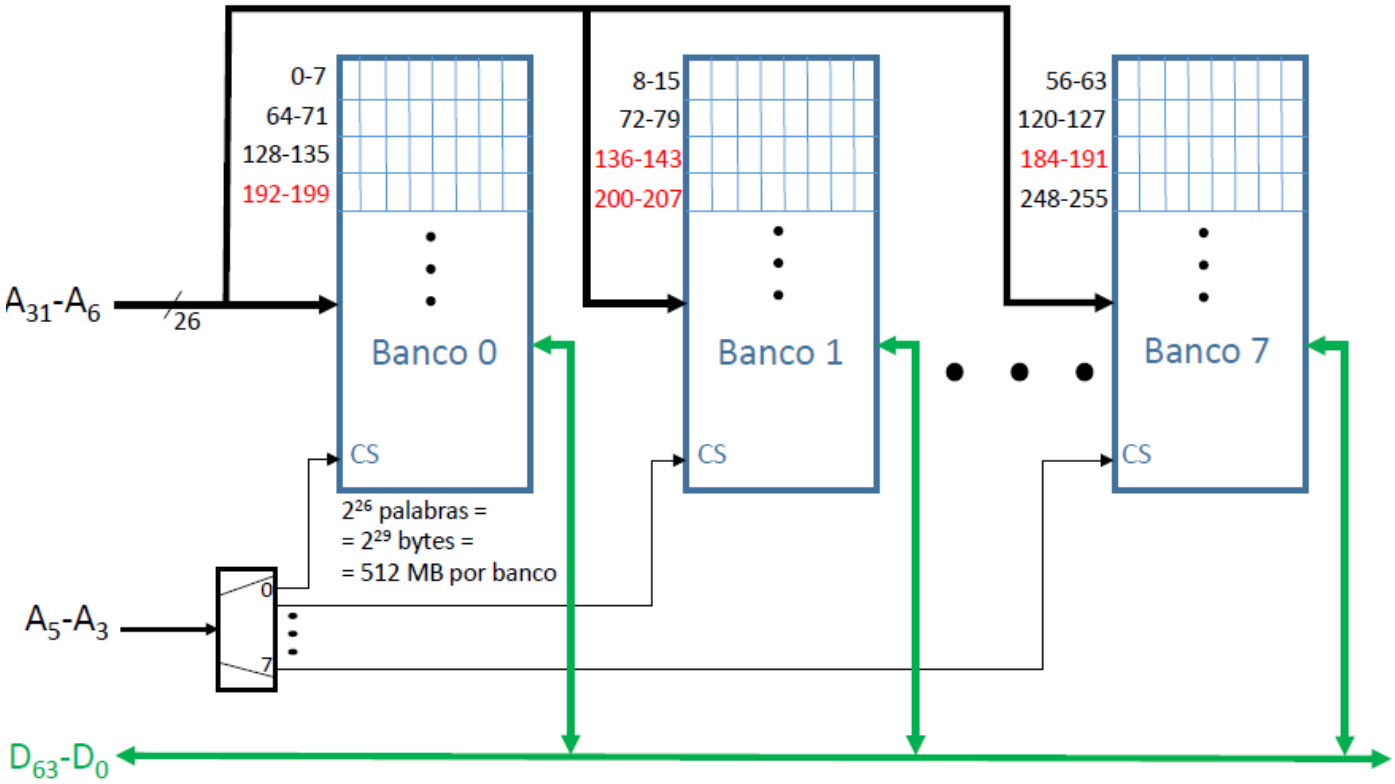
Dir	136	144	152	160	168	176	184	192	200	208	...
Banco	1	2	3	4	5	6	7	0	1	2	...
Tini	0	1	2	3	4	5	6	7	8	9	...
Tfin	6	7	8	9	10	11	12	13	14	15	...



Para entender el funcionamiento de la memoria entrelazada pensemos en una dirección generada de 32 bits. La memoria es direccionable por Bytes y cada palabra ocupa 64 bits (8 Bytes). Tenemos 4 GB de memoria repartidos en 8 bancos de 512 MB cada uno. Las direcciones de una palabra en hexadecimal siempre acaban en 0 ó en 8.

```
0000 0000h  ->  00000000 00000000 00000000 00000000
0000 0008h  ->  00000000 00000000 00000000 00001000
0000 0010h  ->  00000000 00000000 00000000 00010000
0000 0018h  ->  00000000 00000000 00000000 00011000
```

Las palabras consecutivas se almacenan en bancos consecutivos. Los bytes 0-7 corresponden a la palabra 0, los bytes 8-15 corresponden a la palabra 1 y así sucesivamente. Al buscar la palabra 8 que ocupa los bytes 64-71 hemos regresado el banco 0 que quedó libre tras sus 6 ciclos de latencia (antes de volver a ser accedido).



La selección de banco se hace con los 3 bits de ADDRESS A(5), A(4) y A(3). El banco que tenga activado su Chip Select (CS) con el decodificador de selección es el que vuelca el dato en el bus de salida, el resto se ponen en modo de alta impedancia.

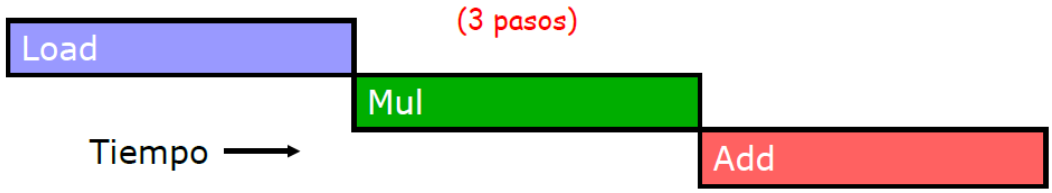
Al tener una organización de bancos independientes, todos ellos con latencia de 6 ciclos para la extracción completa de un dato, podemos realizar la copia de un array completo de 64 componentes en tan solo 70 ciclos.

➢ 4.1.6 Riesgos LDE. Encadenamiento

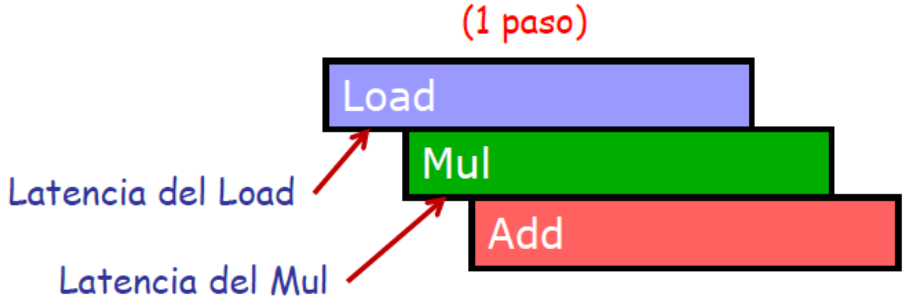
```
LV       V1
MULVV.D  V3,V1,V2
ADDVV.D  V5,V3,V4
```

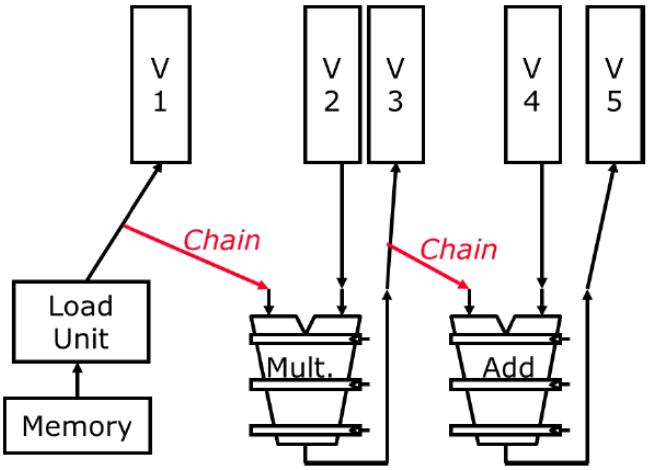
Este conjunto de 3 instrucciones vectoriales forman un convoy puesto que no existen riesgos estructurales entre ellas. Sin embargo presentan cada una con la anterior dependencia de datos.

Si no tuviésemos encadenamiento estaríamos obligados a formar 3 convoyes que consumen 3 pasos.



Si realizamos el encadenamiento podemos lanzarlo todo en el mismo convoy, aunque tenemos un pequeño lapso de espera para comenzar las operaciones aritméticas. 1 Paso = 1 Convoy





Mientras se van cargando las *n* componentes del Load V1 se van procesando las operaciones del MUL.D V3 y así después al ADD.D V5.

Por lo tanto el MUL.D puede comenzar tras realizar la carga del primer elemento de V1 (típicamente 12 ciclos). Y el ADD.D puede comenzar en cuanto se resuelva la primera multiplicación (típicamente 7 ciclos adicionales).

➢ 4.1.7 Vectores de Longitud Arbitraria. Strip Mining.

Vectores Cortos --> $Long\ V \leq Max\ Valor\ (VLR)$

Vectores Largos --> $Long\ V > Max\ Valor\ (VLR)$

Cuando un vector tiene más componentes de las que puede gestionar la arquitectura del VMIPS se subdivide en un procesado por bloques (*Strip Mining*).

- **Strip Mining**

El número máximo de componentes vectoriales se denota por MVL, el tamaño del array se denota por n y por tanto se tienen (n/MVL) "$división\ entera$" bloques de proceso, más una última operación de tamaño $(n\ mod\ MVL)$.

El tiempo que se consume en el procesado de una operación con array de tamaño n se calcula

$$T_n = \left[\frac{n}{MVL}\right] \cdot \left(T_{loop} + T_{start}\right) + n \cdot T_{chime}$$

n → Tamaño del array.

MVL → Es una constante que denota el ancho de los registros vectoriales.

$\left[\frac{n}{MVL}\right]$ → Es una operación que devuelve un entero, es el número de bloques necesarios de ejecución.
Si $n \leq MVL$, la operación devuelve 1. Es como una división entera con redondeo hacia arriba.

T_{loop} → Actualización de punteros en cada vuelta de iteración "de bloque" y detección de fin.
Le damos valor de 15 ciclos por defecto.

T_{start} → Número de ciclos de llenado de cada operación vectorial presente en los convoyes de cada bloque. Para el VMIPS recordemos que Load/Store son 12 ciclos, MUL.D son 7 ciclos y ADD.D son 6 ciclos.

T_{chime} → Número de convoyes que hay en cada iteración de bloque.

Ejemplo de Strip Mining

Operación vectorial $\vec{A} = s * \vec{B}$, donde los vectores tienen 200 componentes y $MVL = 64$.

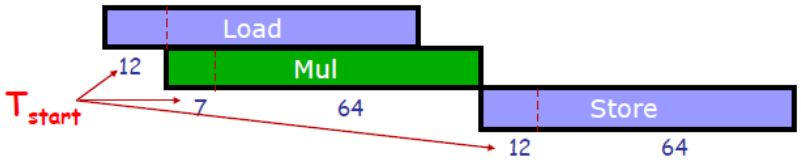
```
        DADDUI   R2,R0,#1600   ;total # bytes in vector
        DADDU    R2,R2,Ra      ;address of the end of A vector
        DADDUI   R1,R0,#8      ;loads length of 1st segment
        MTC1     VLR,R1        ;load vector length in VLR
        DADDUI   R1,R0,#64     ;length in bytes of 1st segment (8 elements)
        DADDUI   R3,R0,#64     ;vector length of other segments (64 elements)
Loop:   LV       V1,Rb         ;load B
        MULVS.D  V2,V1,Fs      ;vector * scalar
        SV       Ra,V2         ;store A (structural hazard)
        DADDU    Ra,Ra,R1      ;address of next segment of A
        DADDU    Rb,Rb,R1      ;address of next segment of B
        DADDUI   R1,R0,#512    ;load byte offset next segment
        MTC1     VLR,R3        ;set length to 64 elements
        DSUBU    R4,R2,Ra      ;at the end of A?
        BNEZ     R4,Loop       ;if not, go back
```

Todo lo que no es el recuadro vectorial son operaciones de control.

El código que precede al lanzamiento del bucle de bloques tiene un retardo que se contabiliza como T_{base}. Esta cantidad no se suele computar en la medida de tiempos porque es muy pequeña respecto a las operaciones del bucle y además solo ocurre una vez.

El código que está dentro del bucle, pero no es del recuadro, lo contamos como retardo T_{loop} (este sí aparece en las ecuaciones y si no dicen nada vale 15 ciclos).

```
LV        V1,Rb
MULVS.D   V2,V1,Fs
SV        Ra,V2
```



El Código vectorial se divide en 2 convoyes, el tiempo de inicialización total es la suma de 12+7+12 que son las latencias de UF. $T_{start} = 31$.

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \cdot \left(T_{loop} + T_{start}\right) + n \cdot T_{chime}$$

$$T_n = [4] \cdot (15 + 31) + 200 \cdot 2 = 584\ ciclos$$

Una iteración del LOOP de bloques (que dará 4 vueltas) se representa por el dibujo de los dos convoyes.

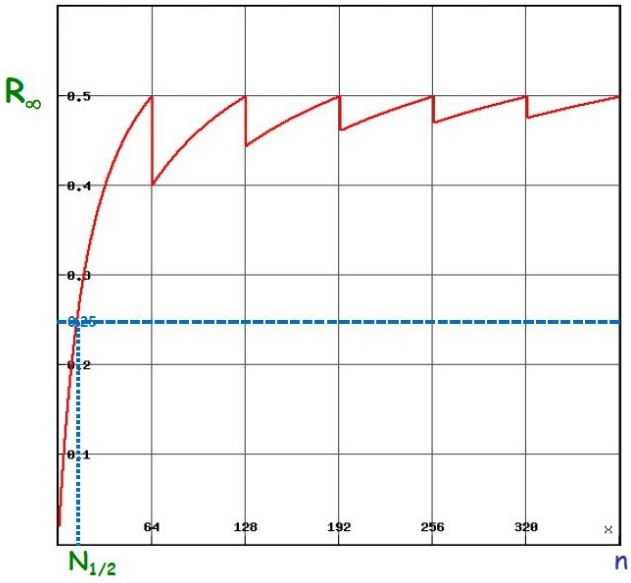
Uno podría pensar que el Store puede empezar inmediatamente después de que acabe el LOAD dado que la UF Load/Store ya está liberada. Sin embargo tenemos que recordar que al asumir la estructura de procesado "en convoyes" que tiene grandes ventajas, tiene el inconveniente de que no se pueden entrelazar lanzamientos de operaciones en convoyes separados. El Store fue pasado a otro convoy porque utiliza la misma UF que el Load.

- ## Medida de Rendimiento en Strip Mining

El rendimiento de la arquitectura de VMIPS se mide con el factor R que representa un cociente entre el número de operaciones en Punto Flotante por ciclo.

$$R(FLOP/ciclo) = \frac{Nº\ OP\ en\ Punto\ Flotante}{T_n}$$

Este cociente depende de n, que como ya hemos dicho es el número de componentes que tienen los vectores a procesar.



Supongamos una expresión típica para el rendimiento como la siguiente:

$$R = \frac{2n}{\left\lceil \frac{n}{64} \right\rceil \cdot 64 + n \cdot 3}$$

Esta expresión nos indica que para cada componente n de los vectores se realizan 2 operaciones de Punto flotante (numerador). Que el tamaño de los registros vectoriales es 64. Cada bloque tiene 3 convoyes.

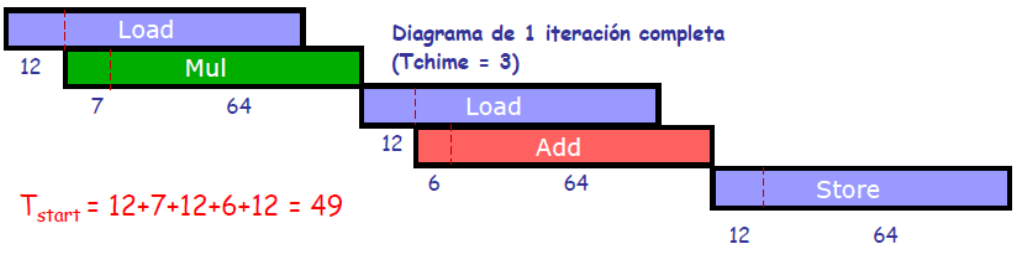
Y finalmente si asumimos que $T_{loop} = 15$ entonces $T_{start} = 49$.

El gráfico nos muestra la evolución del rendimiento en función de n. Como $MVL = 64$, nos encontramos con que el rendimiento máximo (*también llamado asintótico* R_∞) está sobre los puntos múltiplos de 64.

El parámetro $N_{1/2}$ es el primer valor de n con el que se alcanza el 50% del rendimiento asintótico.

Este comportamiento es el que tiene el ejemplo que vimos de código con 3 convoyes:

```
1: LV      V1, Rx      ; load vector X
2: MULVS.D V2, V1, F0  ; vector-scalar multiply
3: LV      V3, Ry      ; load vector Y
4: ADDVV.D V4, V2, V3  ; Vector addition
5: SV      V4, Ry      ; Store result Y
```



➢ 4.1.8 Vectores no consecutivos en memoria

Existen 2 supuestos en los que nos podemos encontrar con vectores no consecutivos:

- Vectores con componentes equiespaciadas.
- Vectores con componentes dispersas.

• Componentes equiespaciadas

Tenemos unas instrucciones especiales LOAD/STORE preparadas para este cometido, las instrucciones con "stride".

```
LVWS      V1,(R1,R2)
SVWS      (R1,R2),V1
```

El equiespaciado se indica en el registro R2. Los accesos son $MEM(R1 + i * R2)$.

• Componentes dispersas

Tenemos unas instrucciones especiales LOAD/STORE preparadas para este cometido, las instrucciones con vectores de índices.

Load disperso -> Operación *gather* (compresión)

```
LVI       V1,(R1+V2)
```

El registro vectorial V2 contiene índices como desplazamientos respecto de la dirección inicial contenida en R1.

Store disperso -> Operación *scatter* (expansión)

```
SVI       (R1+V2),V1
```

El cometido de V2 es exactamente el mismo.