PROCESADORES VECTORIALES SEGMENTADOS

Víctor Viñals Yúfera y Jesús Alastruey Benedé

Multiprocesadores

Curso 2019-20

3º Grado en Ingeniería Informática

Especialidad Ingeniería de Computadores

- Basado en los apuntes de la asignatura "Procesadors Vectorials" de la profesora Montserrat Peirón Guardia, Facultat d'Informàtica de Barcelona (FIB), UPC.
- Algunas transparencias tomadas de cursos de los profesores José María LLabería y Mateo Valero Cortés, FIB, UPC.
- Basada en la asignatura "Fundamentos de Arquitecturas Paralelas", de Ingeniería informática, impartida desde el Curso 1995-96 hasta el 2012-13.

OBJETIVOS

(1) ↓ tiempo de ejecución de una aplicación

(2) ↑ productividad múltiples usuarios

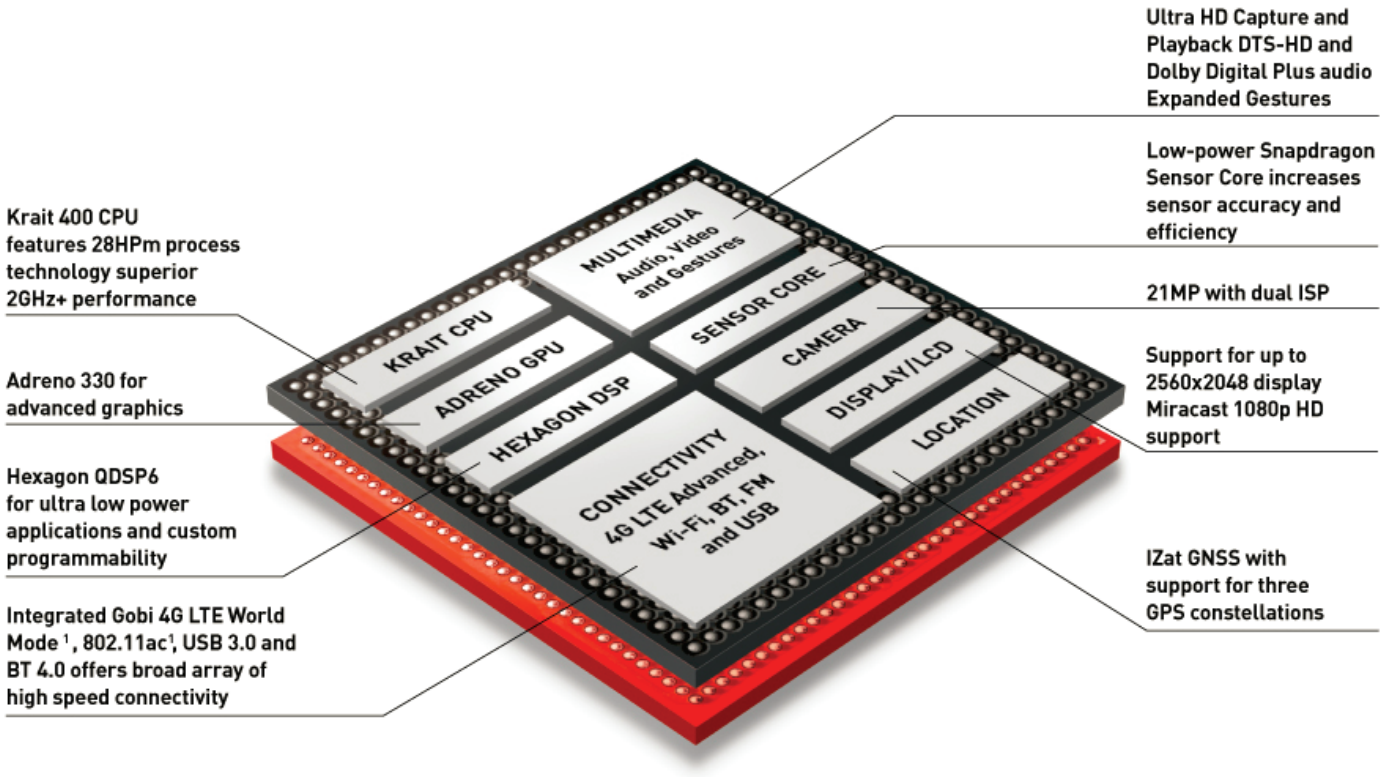
(3) ↑ productividad aplicaciones multihilo

- servidores web, bases de datos, ...

Consecuencias
- Hardware
- Compilación
- Software

(4) Tolerancia a fallos: p.ej. sistemas navegación de un avión

(5) Simplificar componentes y especializar función.
Por ejemplo: sistemas empotrados en-chip (teléfono móvil)



http://www.qualcomm.com/snapdragon/processors/800
Snapdragon 800 series

CONSECUENCIAS

HARDWARE para (1, 2, 3)

- Combinación de:
  - ✓ ejecutar varias instrucciones por ciclo → ILP
  - ✓ la misma instrucción opera sobre varios datos → SIMD
  - ✓ un procesador ejecuta varios hilos (*threads*) → MT
  - ✓ muchos procesadores interconectados → MIMD
- Mucha memoria accesible desde los procesadores con gran ancho de banda
- Mucho disco accesible desde la memoria con gran ancho de banda (entrada/salida)

AMD EPYC Rome

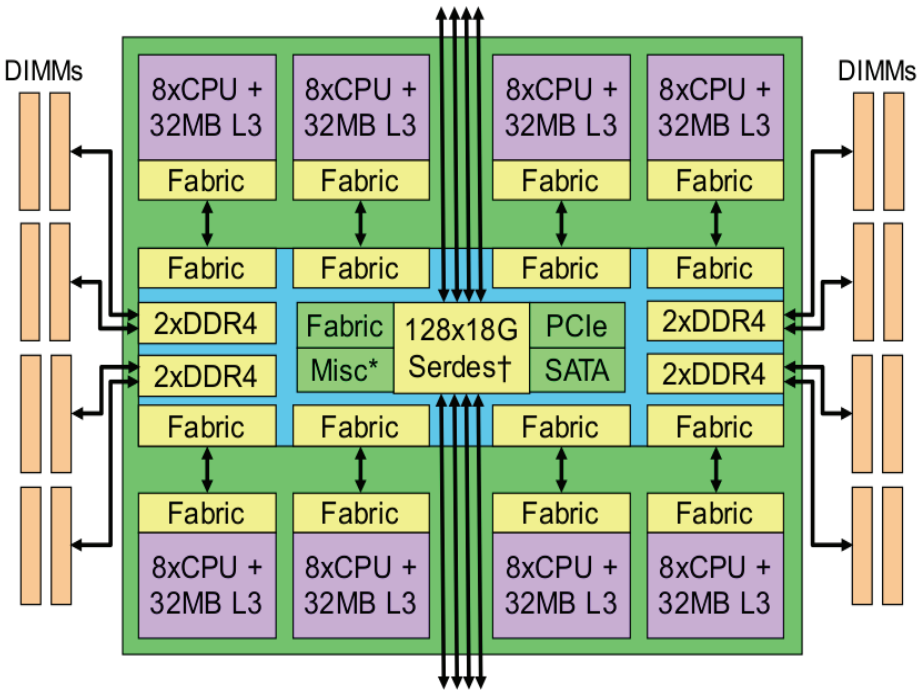


Figura: Linley Gwennap. AMD Rome Ruins Intel Hegemony. Microprocessor Report. Agosto 2019.

CONSECUENCIAS

COMPILACIÓN

- Extracción automática de paralelismo
  a partir de códigos secuenciales
  - ✓ vectorización → SIMD
  - ✓ paralelización → MIMD

Por ejemplo: <https://godbolt.org/z/B67Rxu>

SOFTWARE: modelos de Programación Paralela

- Paralelismo vectorial: FORTRAN 90
- Paralelismo de datos: p.ej. High Performance FORTRAN
- Single-program, multiple-data (SPMD): Co-Array FORTRAN
- Memoria compartida:
  - ✓ OpenMP <http://www.openmp.org>
  - ✓ Pthreads (ANSI/IEEE POSIX std. 1003.1)
  - ✓ Java
  - ✓ Intel Threading Building Blocks (TBB) en C++
  - ✓ C11, C++11, ...
- Paso de mensajes:
  - ✓ MPI
  - ✓ ...

OBJETIVO (1): ↓ Tex de una aplicación
Supercomputación

- Aplicaciones numéricas
  - ✓ predicción meteorológica, dinámica de fluidos, dinámica molecular, alineamiento genético
  - ✓ simulación aerodinámica
    ala, estacionario: 10^{18} operaciones doble precisión
    ala, turbulencia: 10^{20} DP FLOPs
    avión,turbulencia: 10^{23} DP FLOPs
  - ✓ ¡Y muchas más!, ver p.ej.
    <http://www.bsc.es/index.php>
    computer, earth and life sciences
- Estructuras de datos: grandes matrices densas o dispersas
- Tipos de datos: generalmente números reales (coma flotante), 32/64 bits, IEEE 754
- "Pocos" bucles con muchas iteraciones
- Tiempo de ejecución limitado
  por el cálculo *compute-bound*
  o por el acceso a memoria *memory-bound*
- Paralelismo de datos

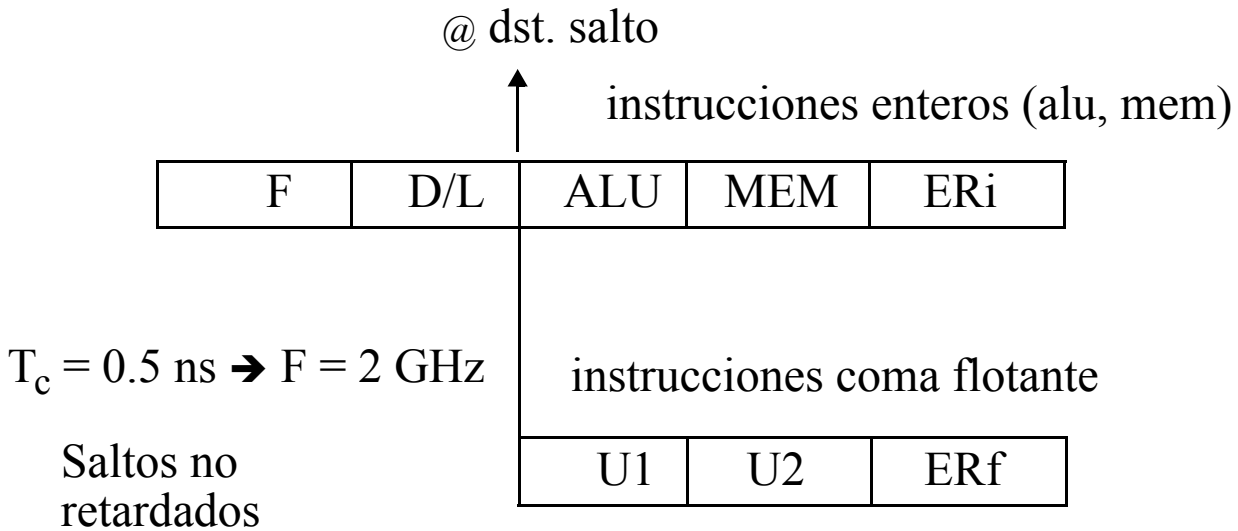
  = Supercomputadores numéricos

1.2 RENDIMIENTO DE UNA SUMA DE VECTORES EN PROCESADORES ESCALARES

CÓDIGO

```
Real*8 C(max),A(max),B(max)
DO I = 1, max
       C(I) = A(I) + B(I)
ENDDO
```

SEGMENTACIÓN



T_c = 0.5 ns ➔ F = 2 GHz

Saltos no retardados

Todos los cortos necesarios

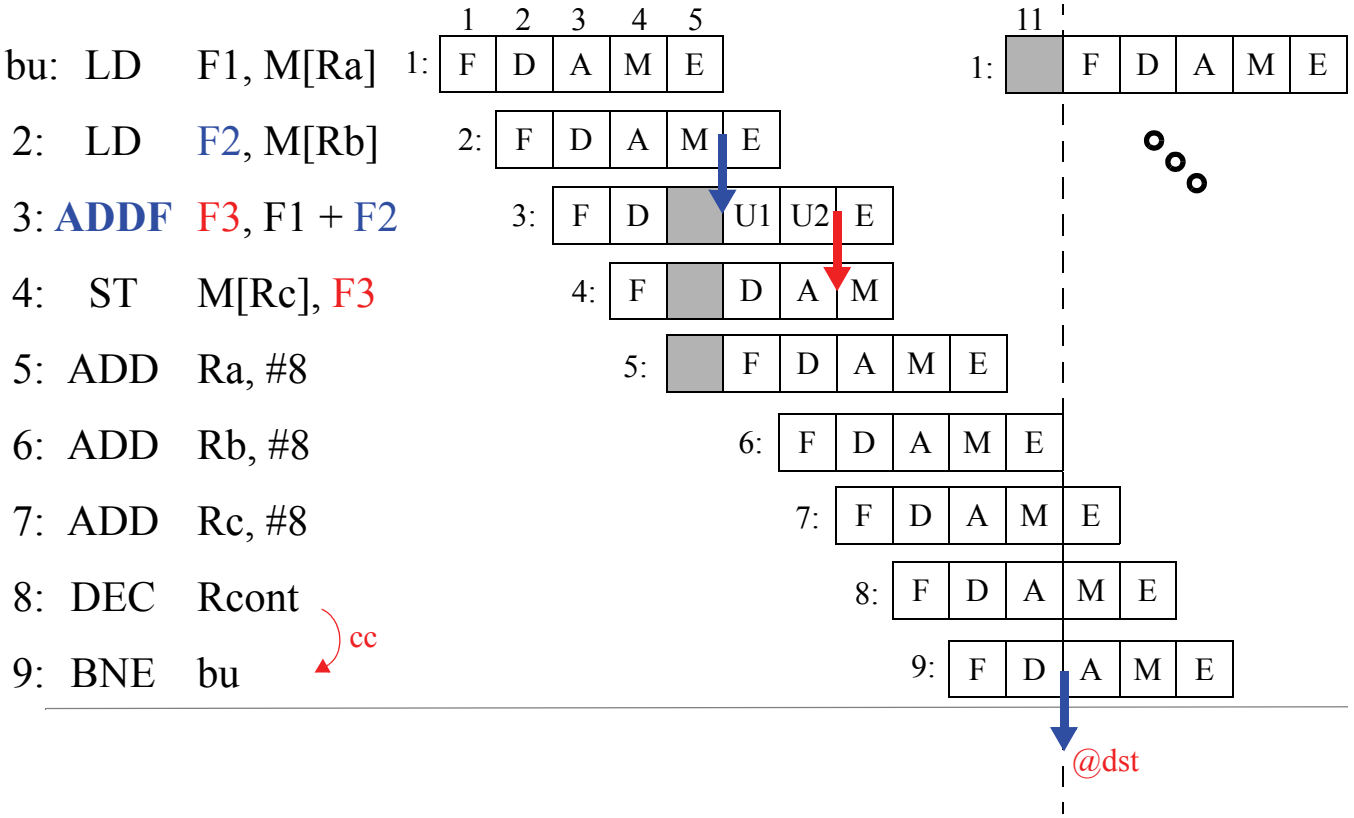
Riesgos en D/L

F:	fetch
D/L:	decod. y lect. operandos en Rx
ALU:	operaciones enteras
MEM:	acceso a memoria
ERx:	escritura en registros
U1, U2:	operaciones coma flotante

1.2 RENDIMIENTO DE UNA SUMA DE VECTORES EN PROCESADORES ESCALARES

PROCESADOR SEGMENTADO

```
Rcont = max; Ra,Rb,Rc = &A, &B, &C
```



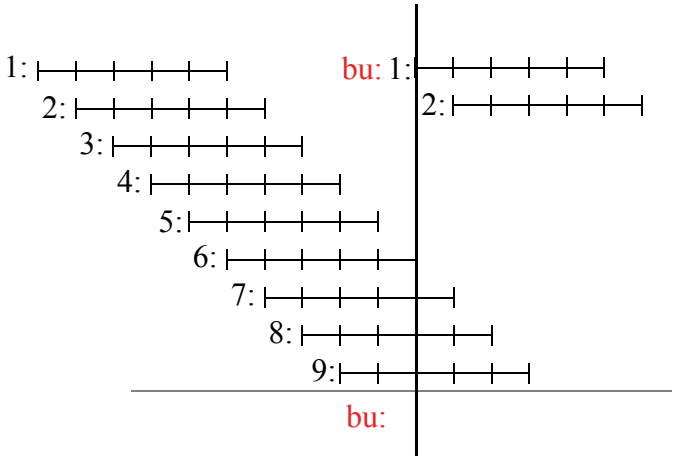
CÓDIGO Y DATOS cargados en caches

- Ciclos / iteración =
- CPF(ciclos/flop) =
- TPF(ns/flop) =
- R(MFLOPS) =

REORDENANDO CÓDIGO

- ○ bu: LD F1, M[Ra]
- □ 2: ADD Ra, #8
- ○ 3: LD F2, M[Rb]
- □ 4: ADD Rb, #8
- ➔ 5: **ADDF** F3, F1, F2
- ○ 6: ST M[Rc], F3
- □ 7: ADD Rc, #8
- □ 8: DEC Rcont
- □ 9: BNE bu

  [--]



- ❒ Un **flop/iteración** (instr. ➔)

  TPF = 5 ns, R = 200 MFLOPS

  rendimiento máximo: 2 GFLOPS

- ❒ Sobrecarga (*overhead*) del bucle:

  lectura y escritura de operandos (instr. ○)

  control de bucle y punteros (instr. □)

DESENROLLANDO D = 4

- bu: LD F10, M[Ra]
- 2: LD F11, M[Ra+8]
- 3: LD F12, M[Ra+16]
- 4: LD F13, M[Ra+24]
- 5: LD F20, M[Rb]
- 6: LD F21, M[Rb+8]
- 7: LD F22, M[Rb+16]
- 8: LD F23, M[Rb+24]
- ➔ 9: **ADDF** F30, F10, F20
- ➔ 10:**ADDF** F31, F11, F21
- ➔ 11:**ADDF** F32, F12, F22
- ➔ 12:**ADDF** F33, F13, F23
- 13: ST M[Rc], F30
- 14: ST M[Rc+8], F31
- 15: ST M[Rc+16], F32
- 16: ST M[Rc+24], F33
- 17: ADD Ra, #32
- 18: ADD Rb, #32
- 19: ADD Rc, #32
- 20: SUB Rcont, #4
- 21: BNE bu

[--]

- Ciclos / iteración =
- CPF(ciclos/flop) =
- TPF(ns/flop) =
- R(MFLOPS) =

DESENROLLANDO D = ∞

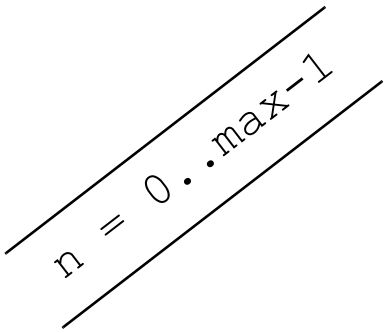
- LD F1n, M[Ra+n*8]
- LD F2n, M[Rb+n*8]

[---]

➔ **ADDF** F3n, F1n, F2n

[---]

- ST M[Rc+n*8], F3n

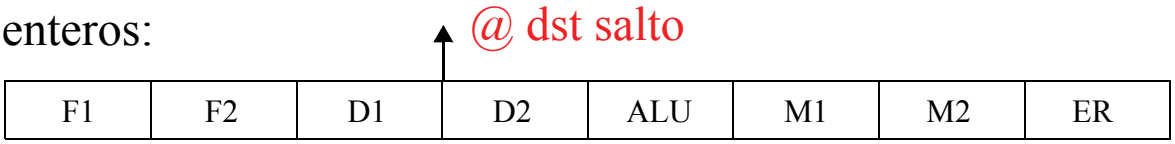


¡ IRREALIZABLE ! ➔ 3 x `max` registros

- CPF(ciclos/flop) =
- TPF(ns/flop) =
- R(MFLOPS) =

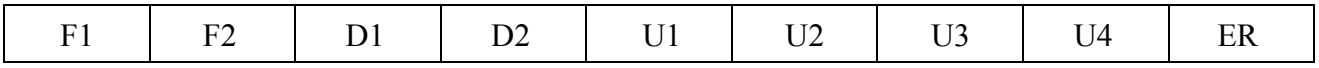
PROCESADOR SUPERSEGMENTADO

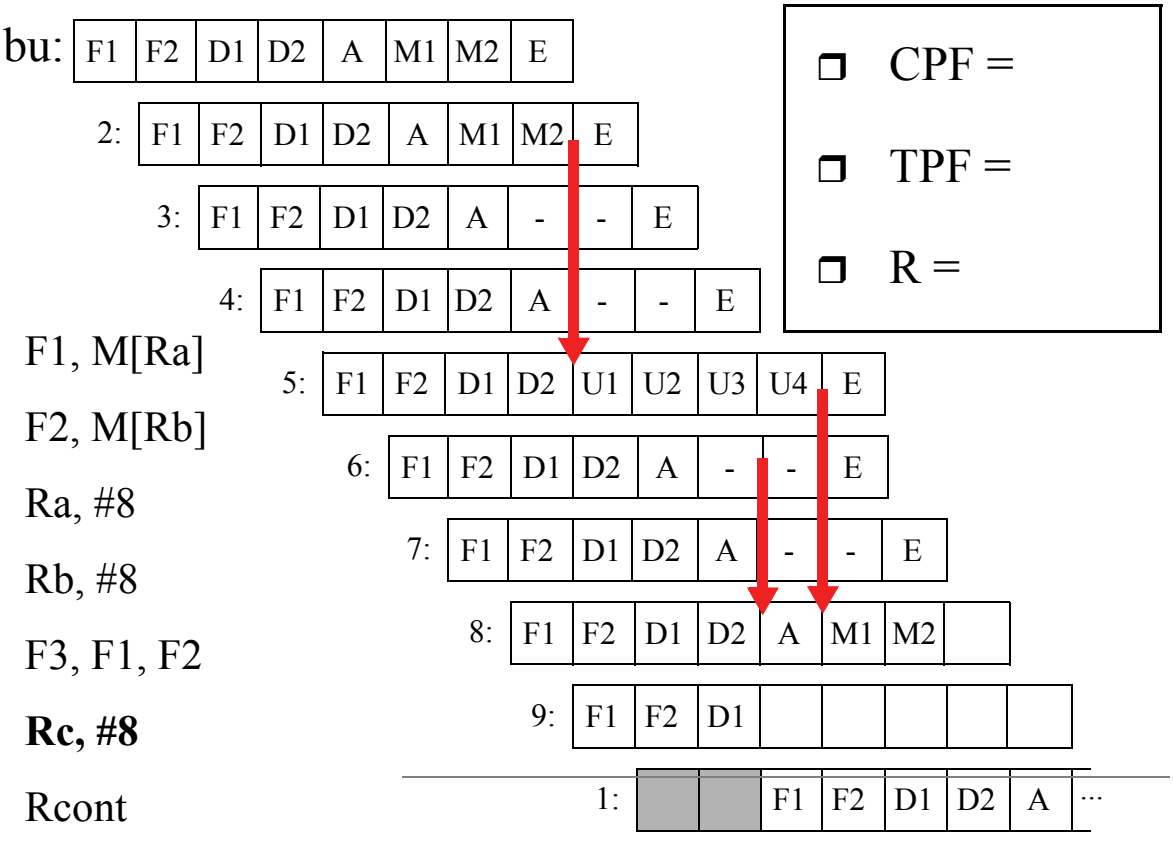
enteros:



Tc = 0.25 ns ➔ 4 GHz

coma flotante:





- CPF =
- TPF =
- R =

```
bu:  LD    F1, M[Ra]
2:   LD    F2, M[Rb]
3:   ADD   Ra, #8
4:   ADD   Rb, #8
5:   ADDF  F3, F1, F2
6:   ADD   Rc, #8
7:   DEC   Rcont
8:   ST    M[Rc], F3   ; inicializamos
9:   BNE   bu          ; Rc con 8 menos!
     [--]
```

PROCESADOR SUPERESCALAR

- Cambiamos el control

  Mantenemos rutas de datos (≈ coste)

  T_c = 0.5 ns ➔ 2 GHz

- Suponemos grado 2:
  se buscan dos instrucciones por ciclo

- Pero sólo se ejecutan en paralelo
  si utilizan las dos rutas de datos: entera y coma flotante

- Hay que planificar para formar parejas de instrucciones
  sin dependencias entera-coma flotante

- El efecto es "esconder" los ciclos de ejecución de ADDF

- Ciclos/FLOP

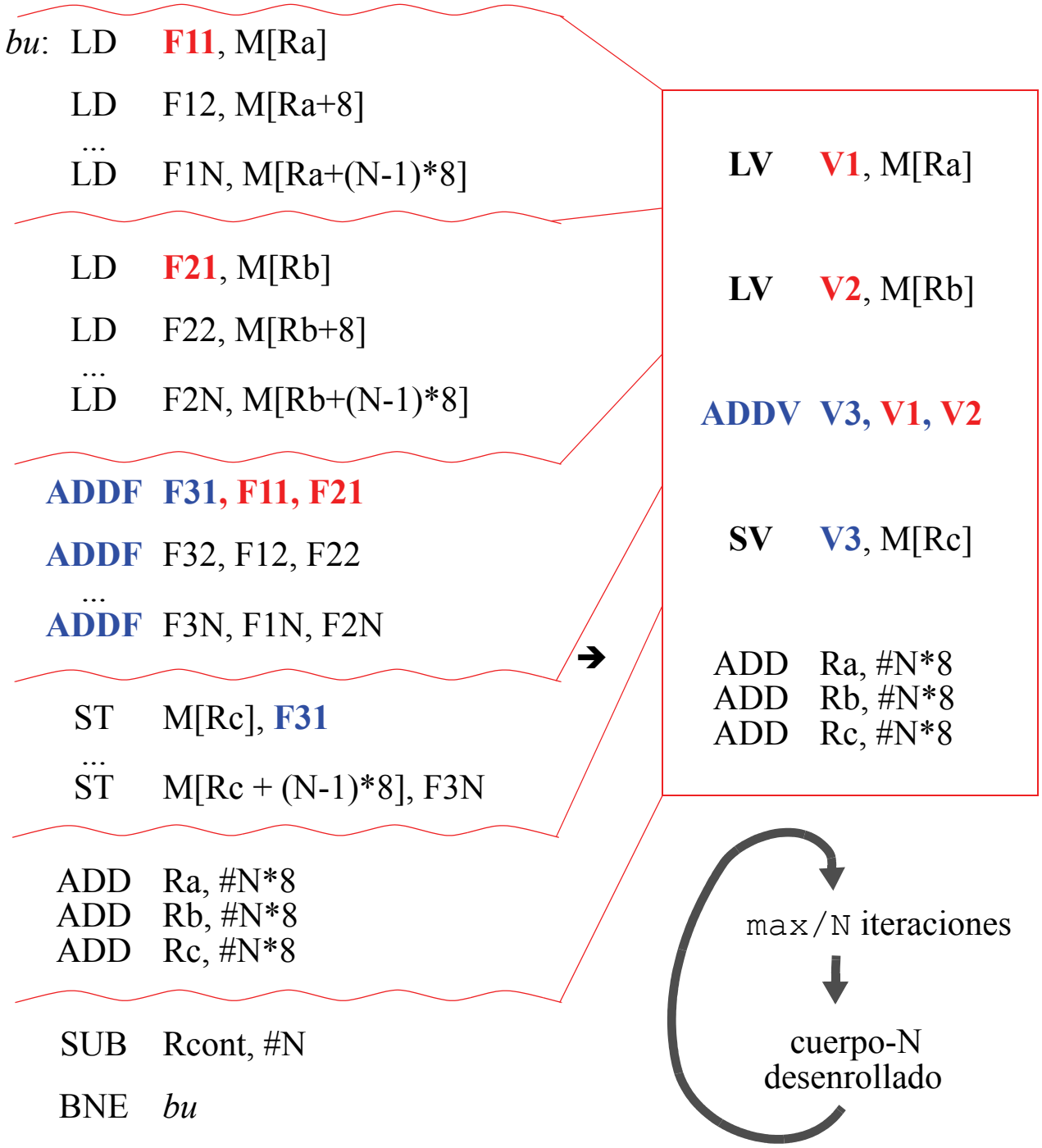
	d = 1	d = 4	d = ∞
Ciclos/ FLOP	9	4.5	3

RESUMEN

- TPF obtenidos en los diferentes casos en **ns**
  (entre paréntesis, la velocidad R en **MFLOPS**)

	d = 1	**d = 4**	**d = ∞**	**Tc** ns
Segmentado R_{max} = 2 GFLOPS	5 (200)	2.75 (363.6)	2 (500)	0.5
Supersegmentado R_{max} = 4 GFLOPS	2.75 (363.6)	1.4375 (695)	1 (1000)	0.25
Superescalar R_{max} = 2 GFLOPS	4.5 (222)	2.25 (444)	1.5 (666)	0.5

1.3 VERSIÓN VECTORIAL DE LA SUMA DE VECTORES



- 1 instrucción vectorial corresponde a N instrucciones del bucle N-desenrollado
  - ✓ Instrucciones por Iteración =
  - ✓ Ancho de Banda con Memoria =
  - ✓ Tiempo de ejecución =

2.1 ARQUITECTURA Y ORGANIZACIÓN

- Primeros supercomputadores tipo CISC (M-M)
  - ✓ instrucción compleja: **ADDV @md, @mf1, @mf2**

    lee dos flujos desde memoria
    calcula
    escribe el flujo resultado hacia memoria
  - ✓ Organización segmentada

    ⇓

    - Latencia de obtención de operandos muy grande, sobre todo para paso no secuencial. Es difícil encadenar
    - CDC Star 100 (‘72), TI ASC (‘72)

- Después, y en la actualidad, filosofía RISC (R-R), o sea,

  desacoplar instrucciones de cálculo y de acceso a memoria:
  - ✓ Instr. vectoriales de acceso a memoria para carga/descarga de registros vectoriales: **LV**, **SV**
  - ✓ Instr. vectoriales de cálculo sobre los registros vectoriales: **ADDV**
  - ✓ VLR: Vector Length Register
    nº de elementos a procesar
    p.ej. para vectores de 64 elementos necesito 6 + 1 bits
  - ✓ VMR: Vector Mask Register
    impide que algunas operaciones se realicen

- Ventajas de un repertorio vectorial:

  Compacto

  - ✓ Una instrucción pequeña codifica N operaciones

  Expresivo:

  La instrucción indica al hardware

  - ✓ que las operaciones son independientes
  - ✓ el número de operaciones
  - ✓ el "patrón" de acceso a memoria:
    en secuencia (stride = 1 elemento)
    o
    a saltos (stride > 1 elemento)

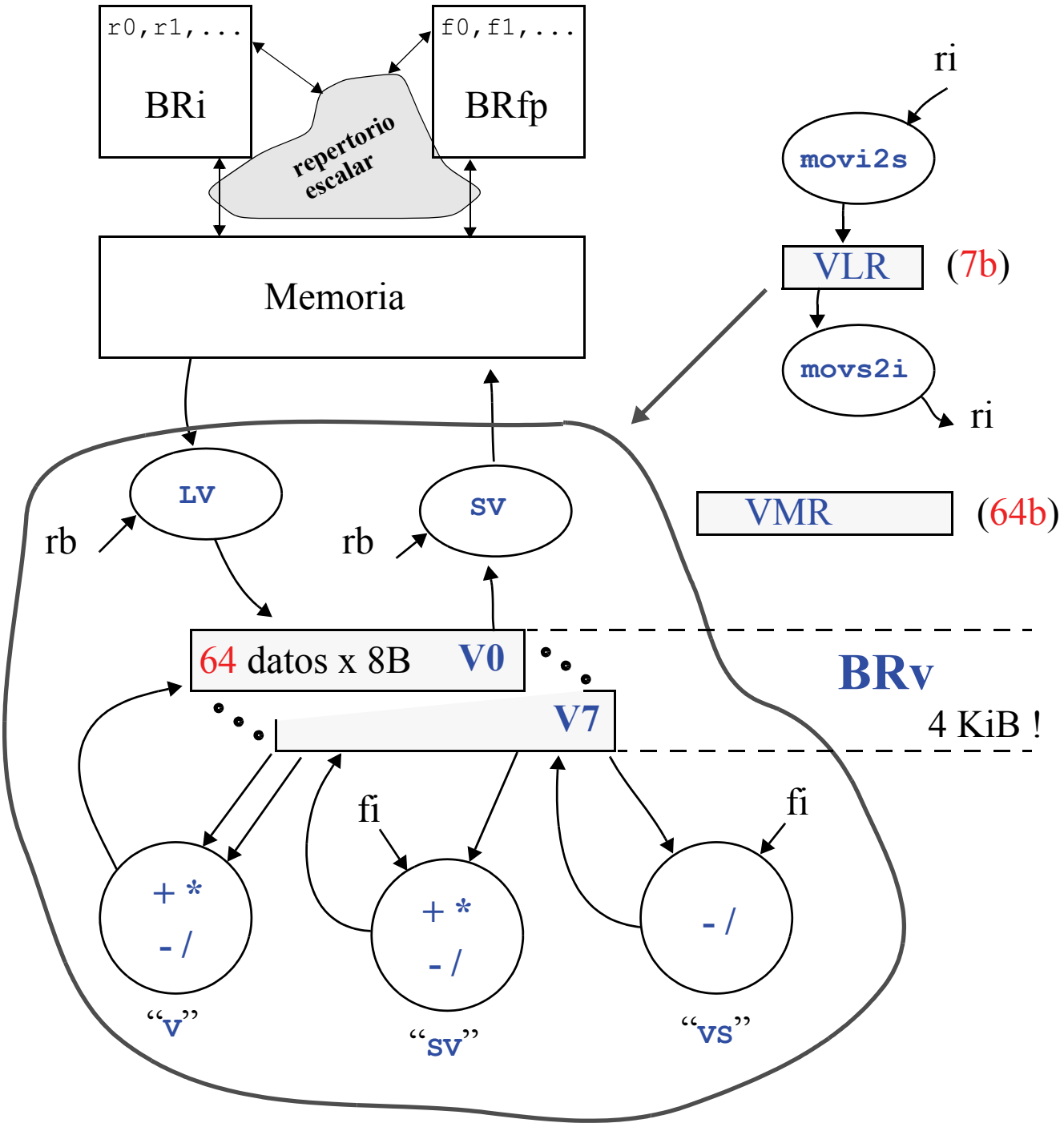
  Escalable

  - ✓ el mismo binario puede ejecutarse
    en una o varias "pistas" segmentadas
    (*parallel pipelines or lanes)*

  Bajo consumo de energía

  - ✓ Ahorramos energía en el acceso a la memoria de instrucciones y en la decodificación (1/N)
  - ✓ Cuando se ejecutan las operaciones
    de una instrucción vectorial
    no hay que gastar energía en comprobar dependencias

- Ejemplo: DLXV[1] Diagrama de flujo de la ALMa[2]



Simple precisión: `addv.F` doble precisión: `addv.D`

1. [HePa12]: Appendix G
2. **ALMa**: Arquitectura de Lenguaje Máquina (en inglés *ISA, Instruction Set Architecture*)

EJEMPLOS

```
movi2s VLR,r2        ; VLR = r2[6:0]   ∈ {0:64}

lv     V0,[r3]       ; V0.i = mem [r3 + i*8]
sv     [r5],V4       ; mem [r5 + i*8] = V4.i

addv   V1,V2,V3      ; V1.i = V2.i + V3.i

addsv                          +
subsv  V1,f0,V3      ;  V1.i =  f0 -V3.i
subvs  V1,V3,f5      ;  V1.i = V3.i - f5
```

VLR	acción[a]	
0	NOP	
1	$i = 0:0$	i = 0:VLR-1
2	$i = 0:1$	
• • •		
64	$i = 0:63$	
65	$i = 0:63$	
• • •		
127		

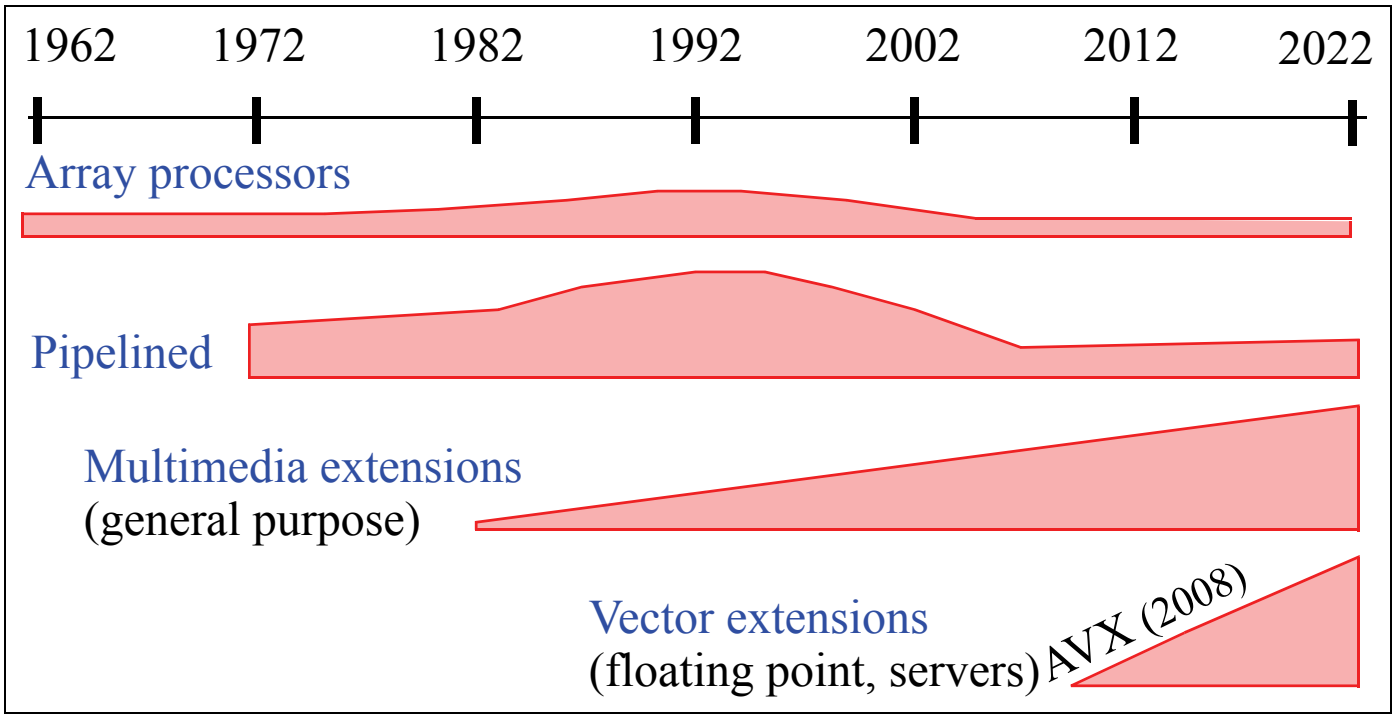
a. nº elementos a procesar = max (VLR mod 64, 64 x VLR div 64)

```
integer i, max
parameter (max = multiplo de 64)
real*8 C(max), A(max), B(max)
do i= 1,max
       C(i) = A(i) + B(i)
enddo
```

Suponemos registros enteros inicializados:

R_A = &A(0), R_B = &B(0), R_C = &C(0), R_{cont} = max, R_{64} = 64

SIMD: Single Instruction Multiple Data



- Extensiones Multimedia: Intel *MMX* (1982), luego Intel *SSE*
  - ✓ Vectores cortos
  - ✓ No hay registro VLR: la longitud vectorial va en el CO

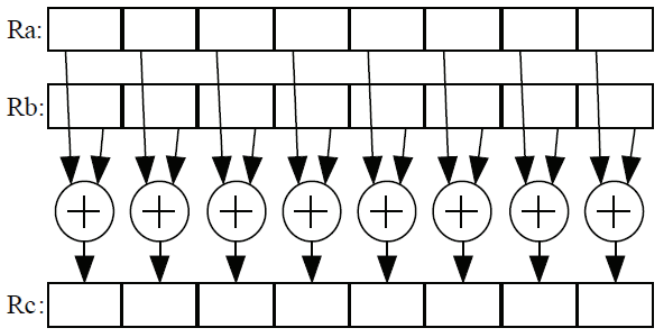


Figure 1. `Packed add` instruction on 8-bit subwords.

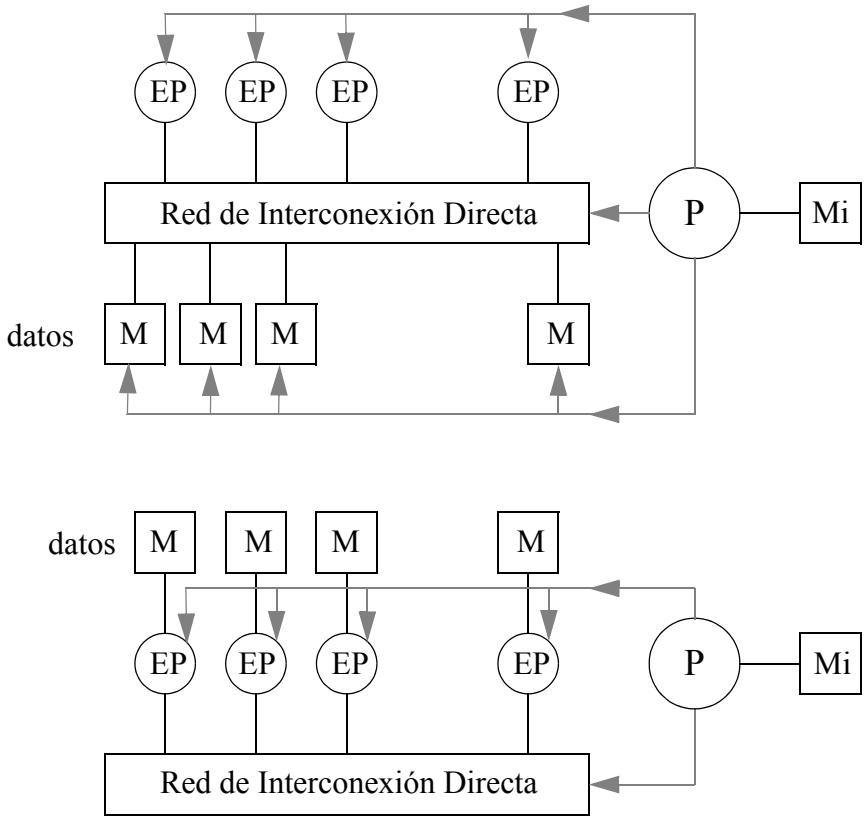
ALU de enteros convencional de 64b: basta impedir la propagación del acarreo en las fronteras de 8 bits ...

  - ✓ Freescale/Apple/IBM *Altivec,* SPARC *VIS*, ARM *Neon*
- Extensiones Vectoriales: Intel *AVX* (4 real*8, 8 real*4, ...)
  - ✓ Soporte hw: ALUs replicadas y BR vectorial (BRV)
  - ✓ BRV de AVX → 8 registros de 256 bits (YMM*0-7*)

❒ **Procesadores Matriciales** (Array Processors)

Históricamente fueron los primeros:

- ✓ Solomon computers
  (Westinghouse Electric Corporation, 1960-62)
  → ILLIAC IV (8x8) en 1968-74
- ✓ Soporte hw: Nodos (mem+regs+ALU) +
  Red Interconexión Directa entre nodos
- ✓ STARAN (74), BSP (82)
  Connection Machines:
  - CM1 - CM2 - CM5 (1985 - 87 - 93)
- ✓ Actualidad: procesado de imagen y de señal



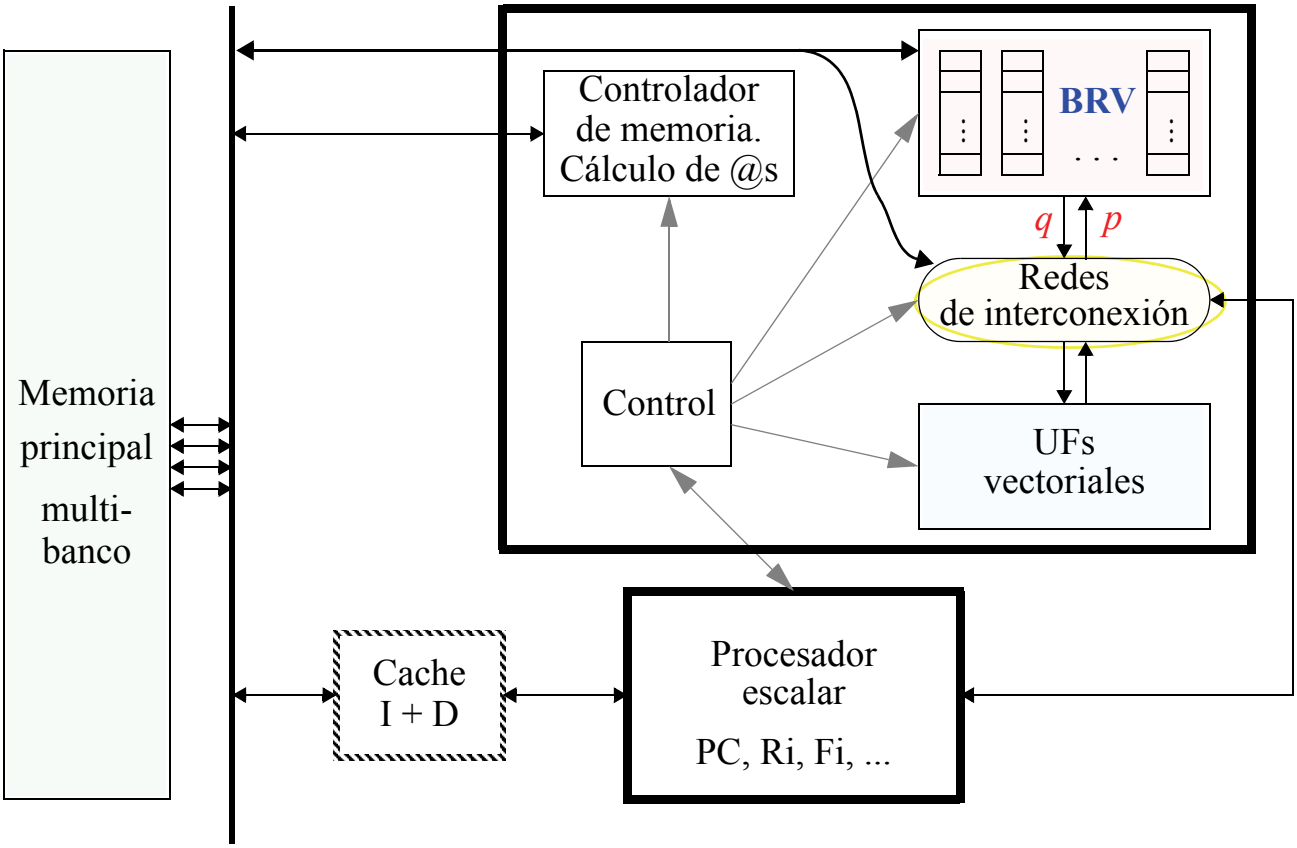
- **Procesadores Vectoriales Segmentados** (pipelined)

  Memoria-memoria

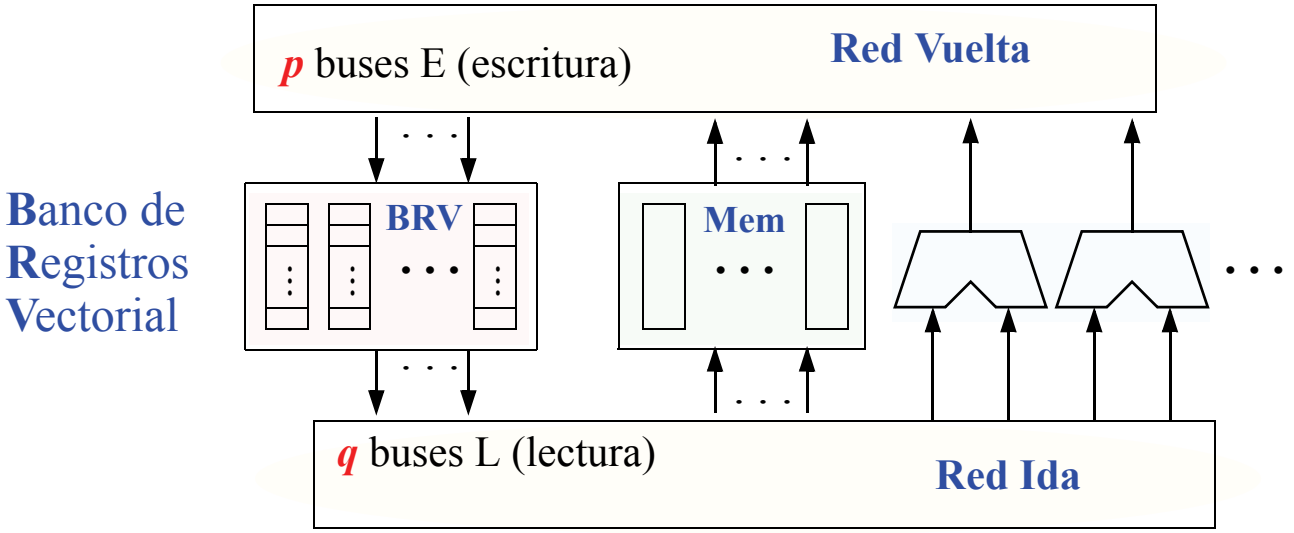
  - ✓ ADDV @md, @mf1, @mf2
  - ✓ CDC Star 100 (72), TI ASC (72)

  Memoria-registros (Seymour Cray, 1975)

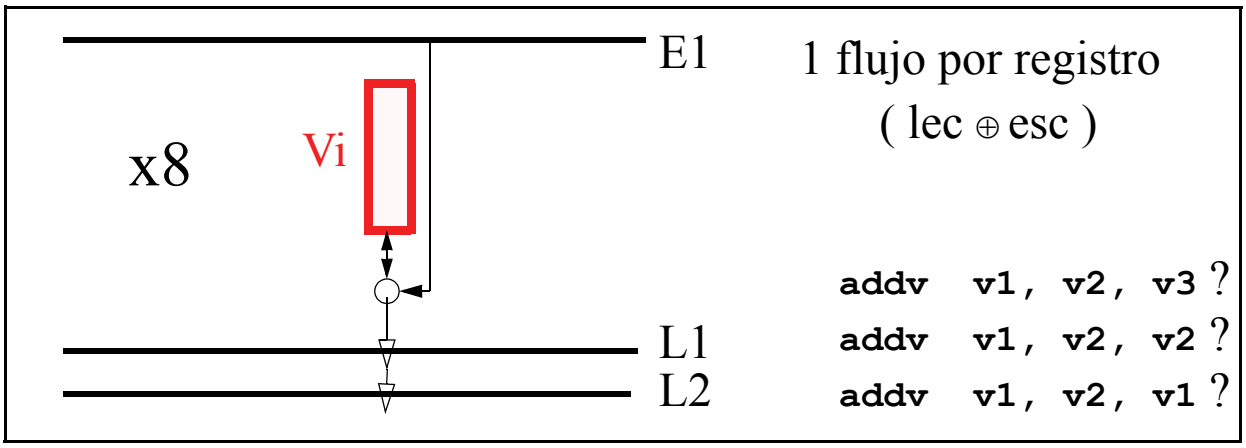
  - ✓ Vectores de tamaño grande, hasta 4K elementos
  - ✓ Soporte hw: Banco Registros Vectorial +
    Memoria Multibanco +
    ALUs + segmentación
  - ✓ Cray 1 (76), Cray2, Cray X-MP y Cray Y-MP
    CDC Eta y Cyber
    IBM 3090 VF
    Fujitsu VP200
    Hitachi S-810
    Convex
    Alliant
    NEC SX/2 - ... - SX/9 - SX/ACE (85 - 07 -13)

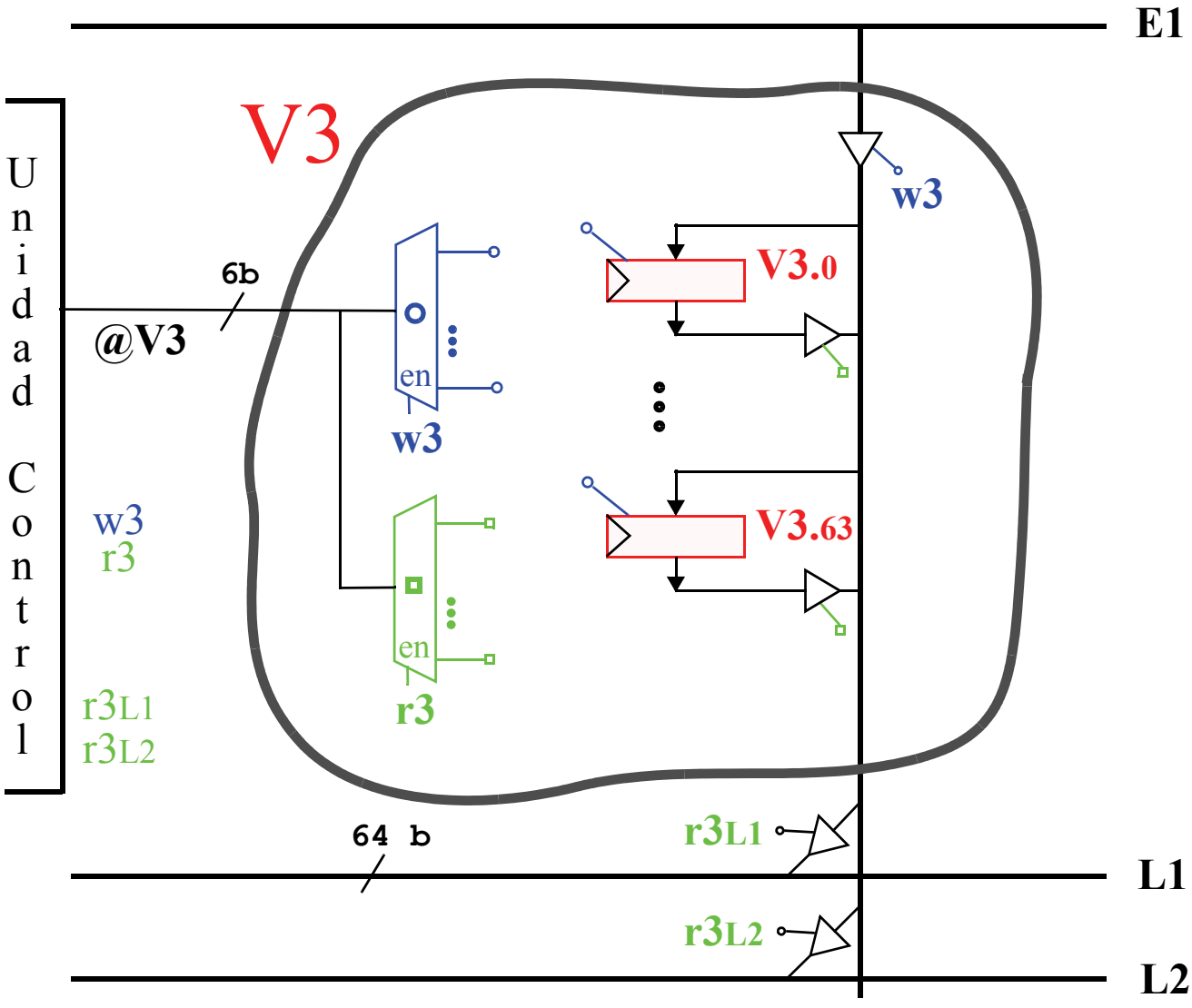


otro punto de vista:

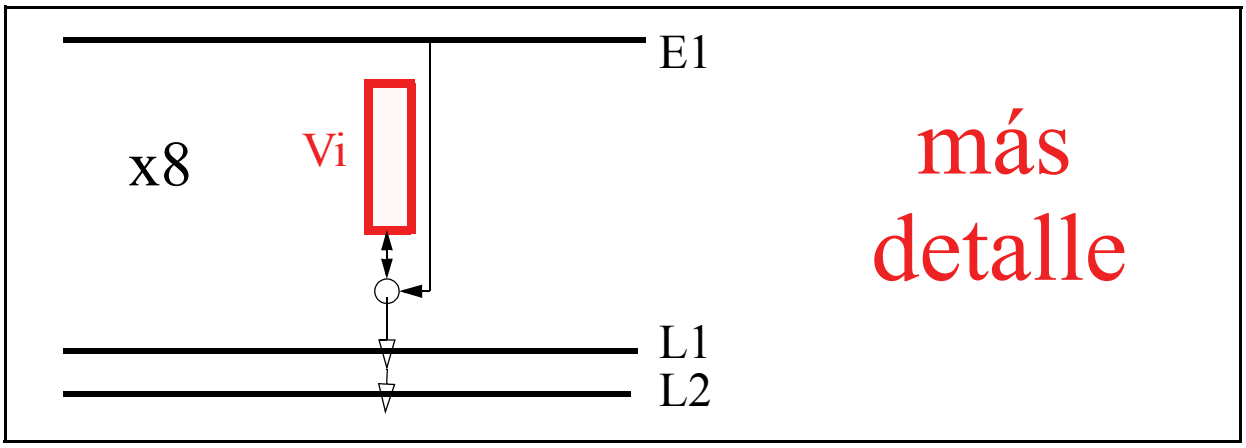


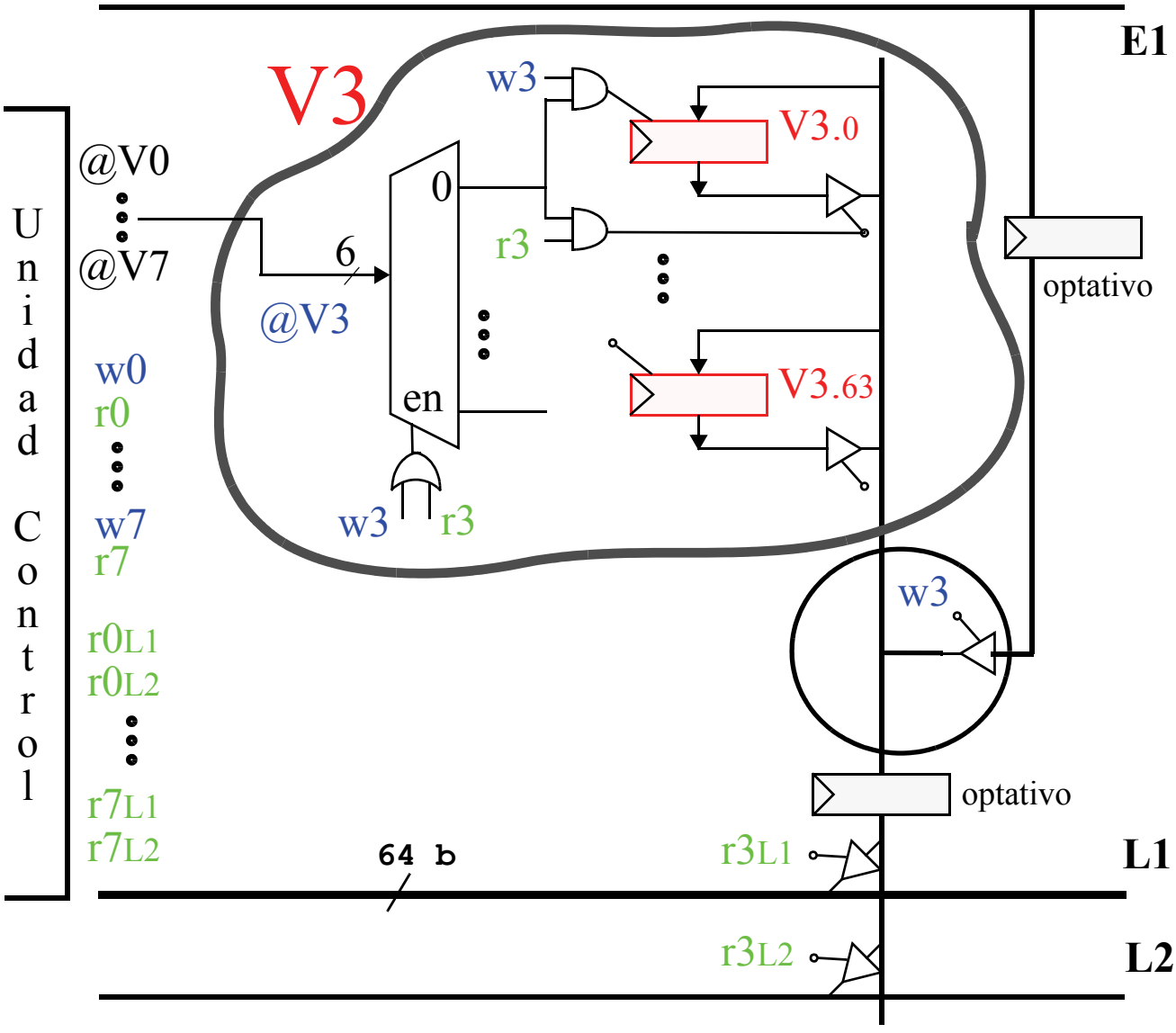
- BRV con **2 buses L**, **1 bus E** y Reg Vec de **1 puerto** (1L⊕1E)



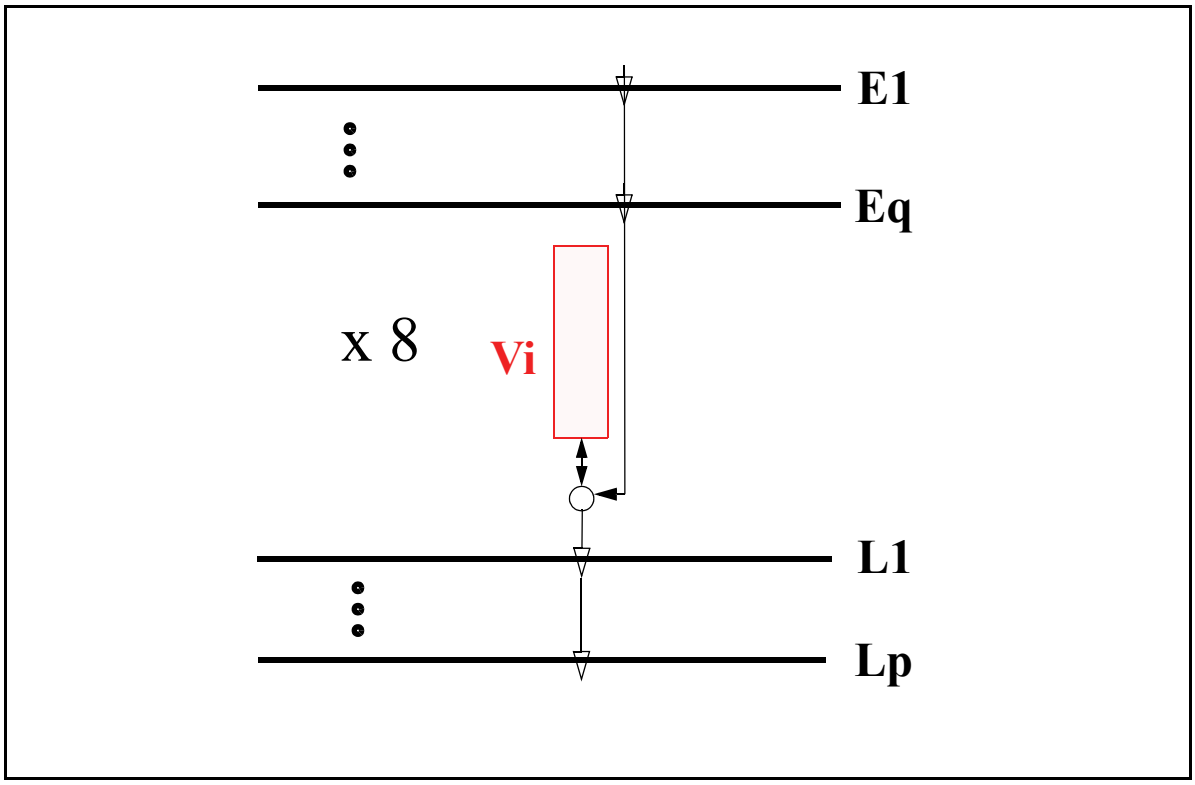


- BRV con **2 buses L**, **1 bus E** y Reg Vec de **1 puerto** (1L⊕1E)



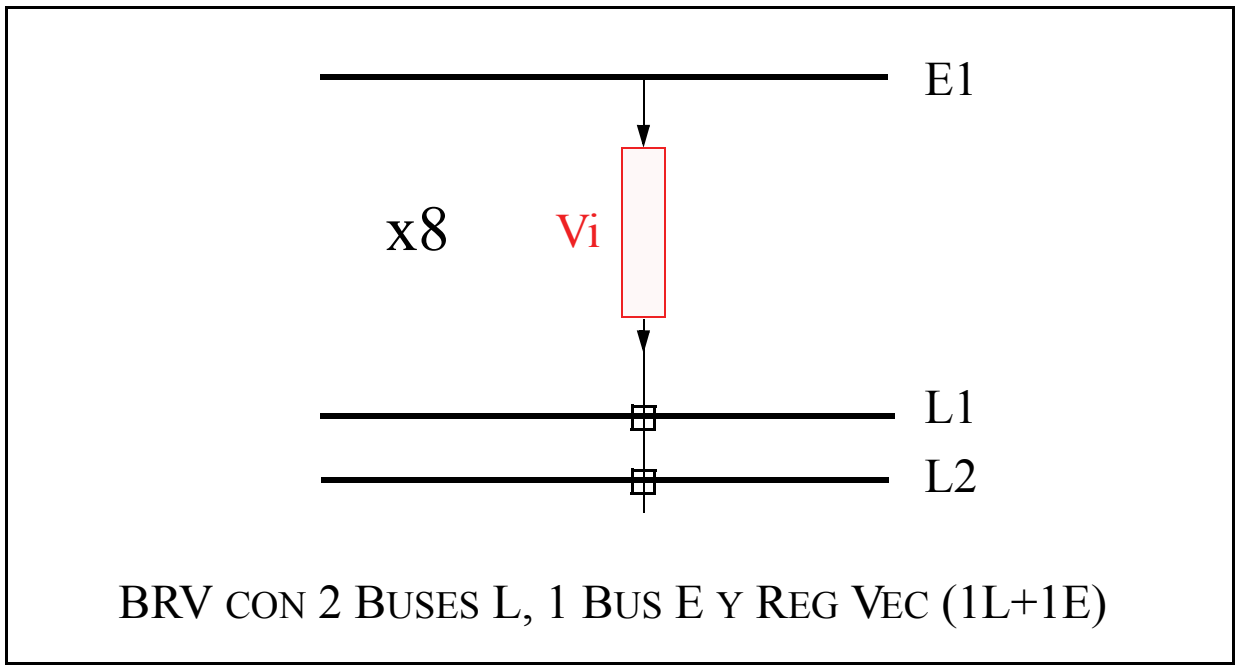


- BRV con *p* buses L, *q* buses E y Reg Vec de **1 puerto**



- 1 flujo por registro
  - ✓ o bien lectura desde un bus L (L1..Lp)
  - ✓ o bien escritura desde un bus E (E1..Eq)
- Globalmente max. 8 flujos
- Ejemplo: 5 buses L y 3 buses E permiten a la vez
  - ✓ dos operaciones de dos operandos, y
  - ✓ una operación de un operando

- Mejora: **2 puertos** por Vi: lectura y escritura (1L+1E)



BRV CON 2 BUSES L, 1 BUS E Y REG VEC (1L+1E)

- Ya es posible `addv v1, v5,v1`
- Solapar dependencias?

```
addv  V1,V2,V3
subv  V4,V5,V1
```

- Solapar antidependencias?

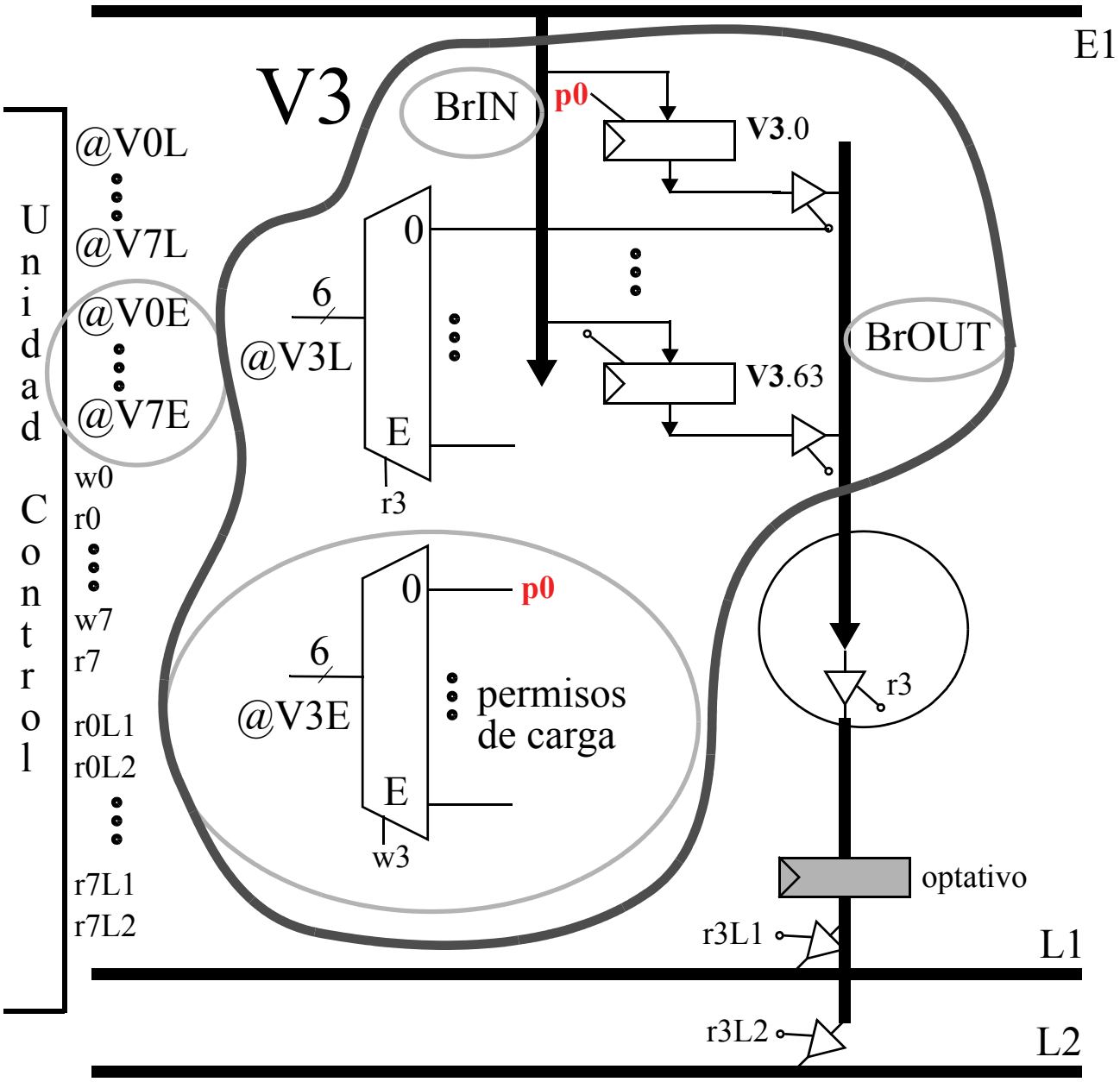
```
addv  V1,V2,V3
subv  V3,V4,V5
```

SI, se llama → encadenamiento general y necesita un control elaborado (lo veremos)

También, y usa un control similar

- Podemos
  - ✓ añadir puertos de lectura en cada **Vi**
  - ✓ tener un BRV con *p* buses L y *q* buses E

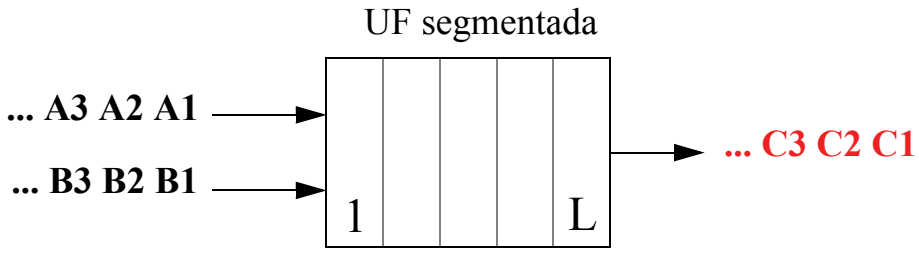
REG VEC CON 2 PUERTOS SEPARADOS: LECTURA Y ESCRITURA



- Descripción temporal:
  - ✓ Latencia (ciclos): tiempo desde que entran los operandos hasta que sale el resultado.
  - ✓ Lat. iniciación (LI) = lat. finalización (ciclos/operación)[1]: nº min. ciclos entre entradas consecutivas de operandos

- Sumadores, Multiplicadores

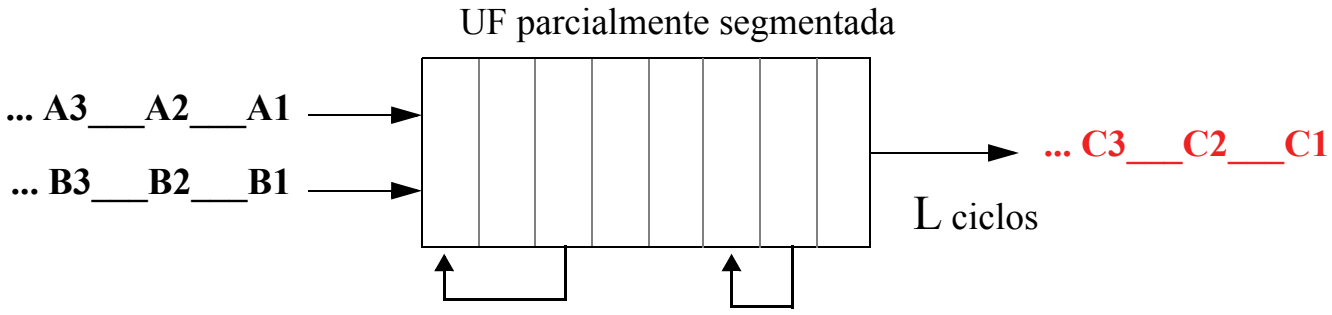
  L = nº de etapas, LI = 1 ciclo/op



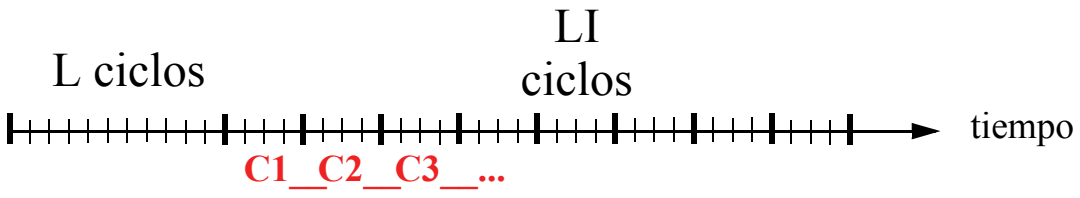
- División, Raíces, Exponenciación, Trigonométricas

  L > nº etapas, por reutilización

  LI > 1

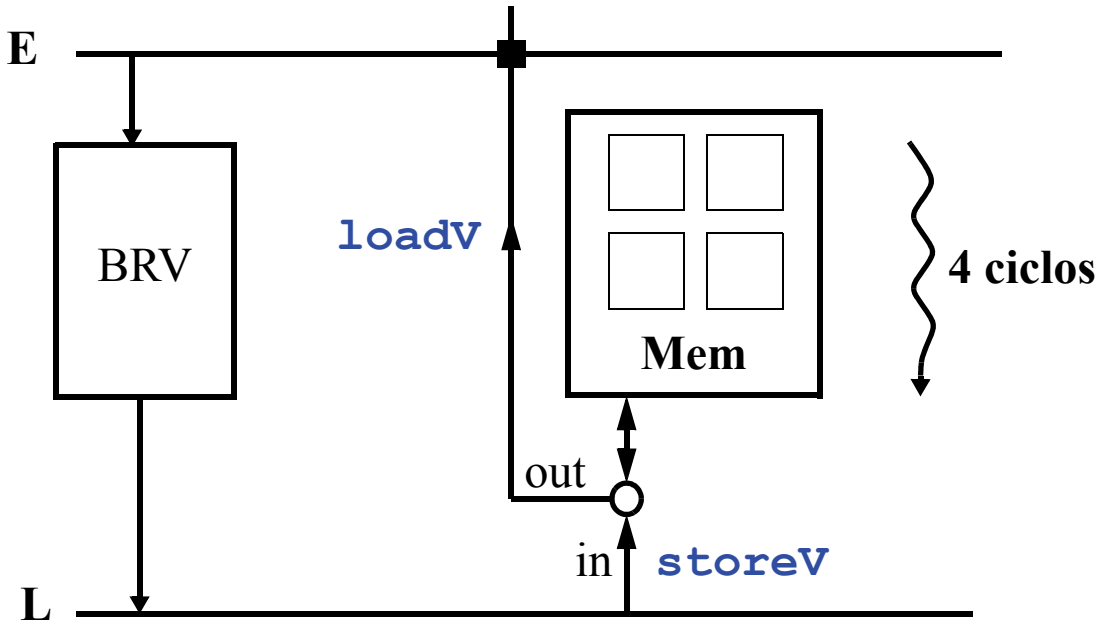


- Modelo de ejecución:

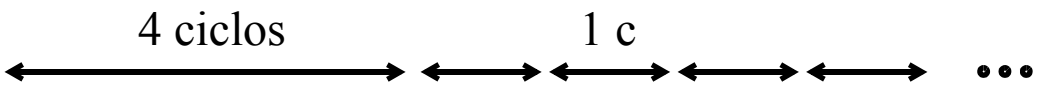


1. La inversa se llama tasa iniciación = tasa finalización (operaciones/ciclo)

- Ejemplo de memoria:
  - 4 GiB con **1 puerto** compartido de lec/escr
  - 4 bancos[1] de 4 ciclos de latencia

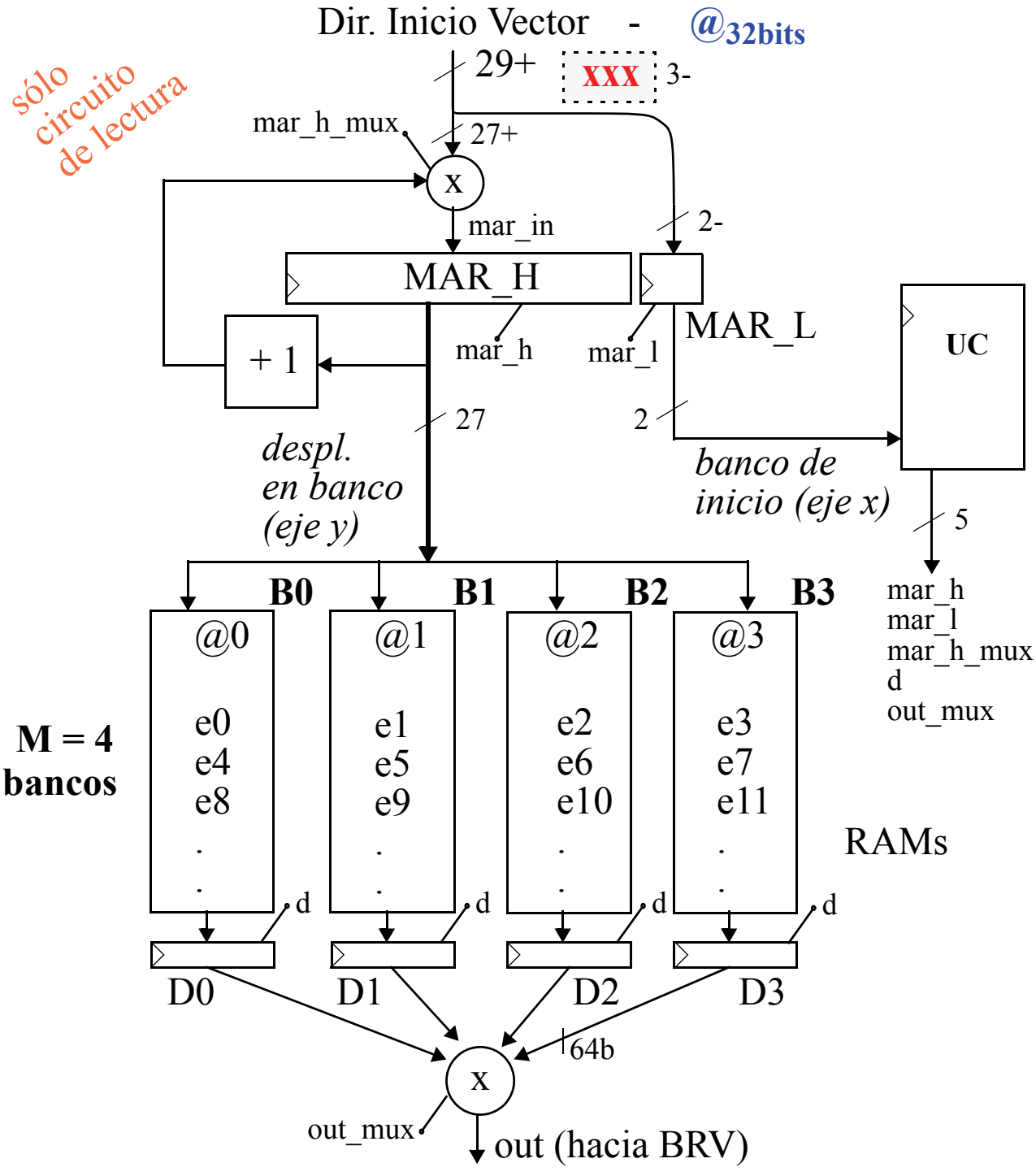


- Queremos este comportamiento en lectura:



- ¿CÓMO?
  - Multibanco: nº bancos = latencia memoria
  - Entrelazado "por palabras":
    "palabras consecutivas en bancos consecutivos",
    asumimos números de 8 bytes **alineados**
  - **L** = 4 ciclos → **M** = 4 bancos de 1GiB
  - Acceso **síncrono** o acceso **desfasado**

1. También podemos decir "módulos"



- $@_{32bits}$ → **nº banco** = ($@_{32bits}$/8) mod 4 → {0, 1, 2, 3}

- Parte del control:
  - Cargar **MAR_H** cada cuatro ciclos (ojo primer acceso!)
  - Cargar todos los **Di** cada cuatro ciclos
  - Multiplexar **Di** en **out** cada ciclo

Ej1. @ = 32 → fila 1, columna 0 (e4, e5, e6, ...)

MAR	**mar_in**	**1**										
	MAR_H											
B0	**D0**											
B1	**D1**											
B2	**D2**											
B3	**D3**											
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**ciclos**
OUT												

Ej2. @ = 48 → fila 1, columna 2 (e6, e7, e8, ...)

MAR	**mar_in**	**1**										
	MAR_H											
B0	**D0**											
B1	**D1**											
B2	**D2**											
B3	**D3**											
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**ciclos**
OUT												

Ej1. @ = 32 → fila 1, columna 0 (e4, e5, e6, ...)

MAR	mar_in	1	2	2	2	2	3	3	3	3	4	4	
	MAR_H		1				2				3		
B0	D0						e4				e8		
B1	D1						e5				e9		
B2	D2						e6				e10		
B3	D3						e7				e11		
		1	2	3	4	5	6	7	8	9	10	11	ciclos
OUT			L = 4				e4	e5	e6	e7	e8	e9	

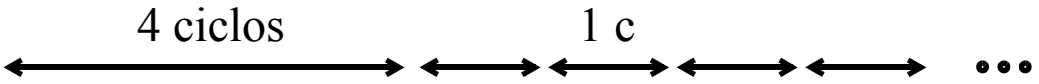
Ej2. @ = 48 → fila 1, columna 2 (e6, e7, e8, ...)

MAR	mar_in	1	2	2	2	2	3	3	3	3	4	4	
	MAR_H		1				2				3		
B0	D0						e4				e8		
B1	D1						e5				e9		
B2	D2						e6				e10		
B3	D3						e7				e11		
		1	2	3	4	5	6	7	8	9	10	11	ciclos
OUT			L = 4				-	-	e6	e7	e8	e9	

- Problemas con acceso **Síncrono**:
  - ✓ lectura: latencia variable o irregular
  - ✓ escritura
    - un registro MDRin para cada banco
    - es lenta: (1 1 1 1 4) (1 1 1 1 4) ...

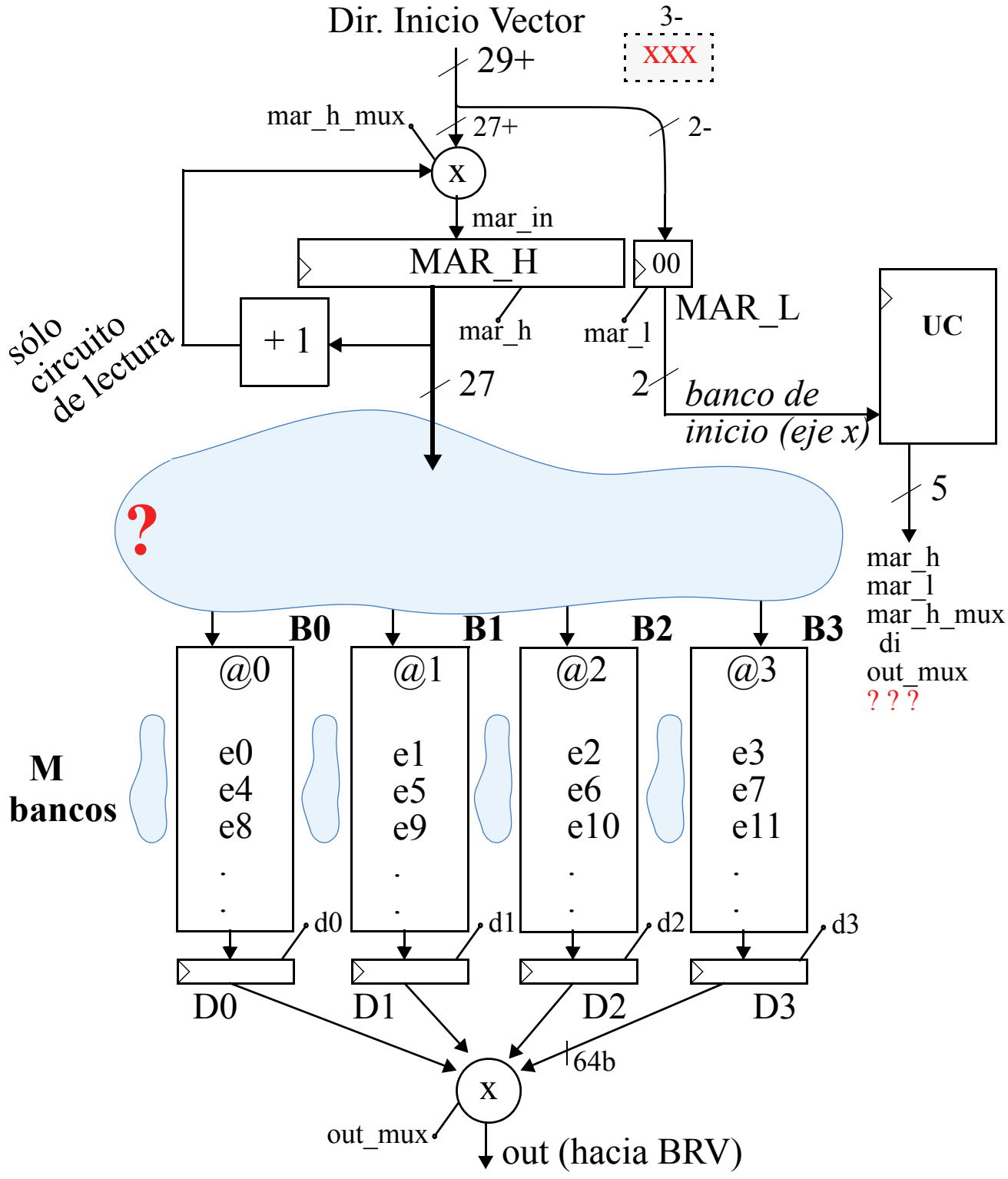
- Una solución: acceso **desfasado**
  los bancos son más independientes
  - ✓ un registro MAR para cada banco
  - ✓ control más complejo
  - ✓ resultado:

LECTURA



ESCRITURA

Veamos cómo se implementa -->



- **Control ?**:

  mar_h, mar_l, mar_h_mux, d_i,
  outmux, ? ? ?

Ej1. @ = 32 → fila 1, columna 0 (e4, e5, e6, ...)

MAR	**mar_in**	**1**				**2**				**3**		
	MAR_H		**1**				**2**				**3**	
MAR_0												
MAR_1												
MAR_2												
MAR_3												
Actividad B0												
Actividad B1												
Actividad B2												
Actividad B3												
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**ciclos**
OUT												

Ej2. @ = 48 → fila 1, columna 2 (e6, e7, e8, ...)

MAR	**mar_in**	**1**				**2**				**3**		
	MAR_H		**1**				**2**				**3**	
MAR_0												
MAR_1												
MAR_2												
MAR_3												
Actividad B0												
Actividad B1												
Actividad B2												
Actividad B3												
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**ciclos**
OUT												

Ej1. @ = 32 → fila 1, columna 0 (e4, e5, e6, ...)

MAR	mar_in	1	1	1	1	2				3		
	MAR_H		1	1	1	1	2			3		
MAR_0				1			2			3		
MAR_1					1			2				
MAR_2						1			2			
MAR_3							1			2		
Actividad B0				e4			e8					
Actividad B1					e5			e9				
Actividad B2						e6			e10			
Actividad B3							e7			e11		
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*ciclos*
OUT			L = 1 + 4				e4	e5	e6	e7	e8	e9	

- Síncrono

Ej2. @ = 48 → fila 1, columna 2 (e6, e7, e8, ...)

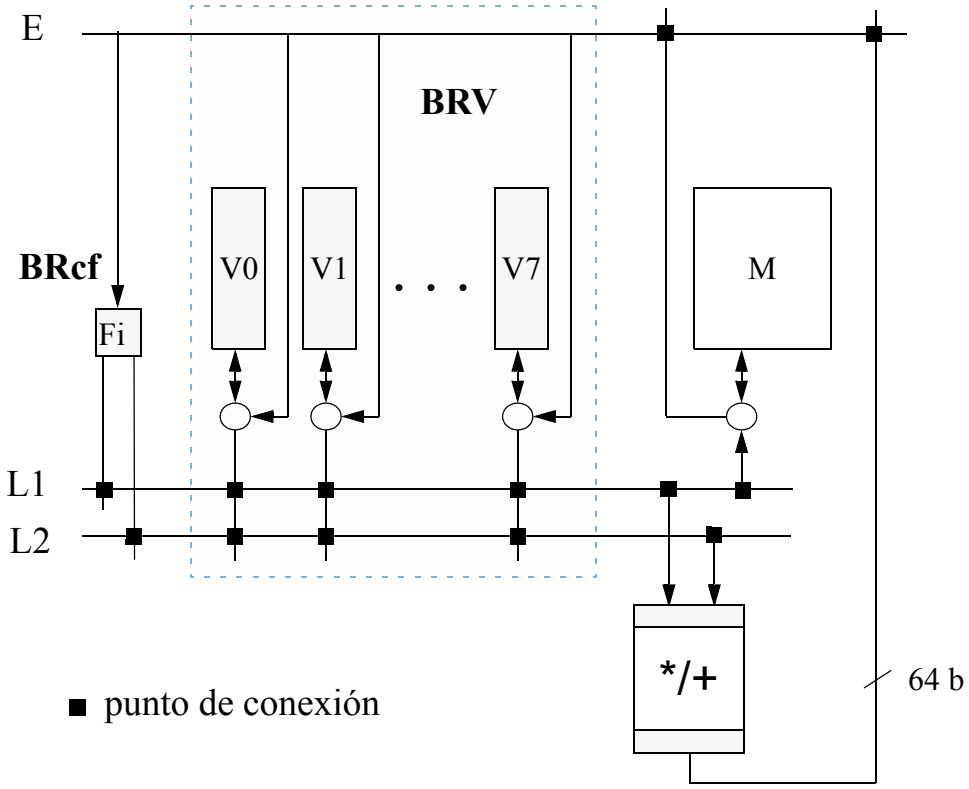
MAR	**mar_in**	1				2				3		
	MAR_H		1				2				3	
Actividad B0						e4				e8		
Actividad B1						e5				e9		
Actividad B2						e6				e10		
Actividad B3						e7				e11		
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*ciclos*
OUT			L = 4 + ...				...		e6	e7	e8	e9	e10

- Desfasado

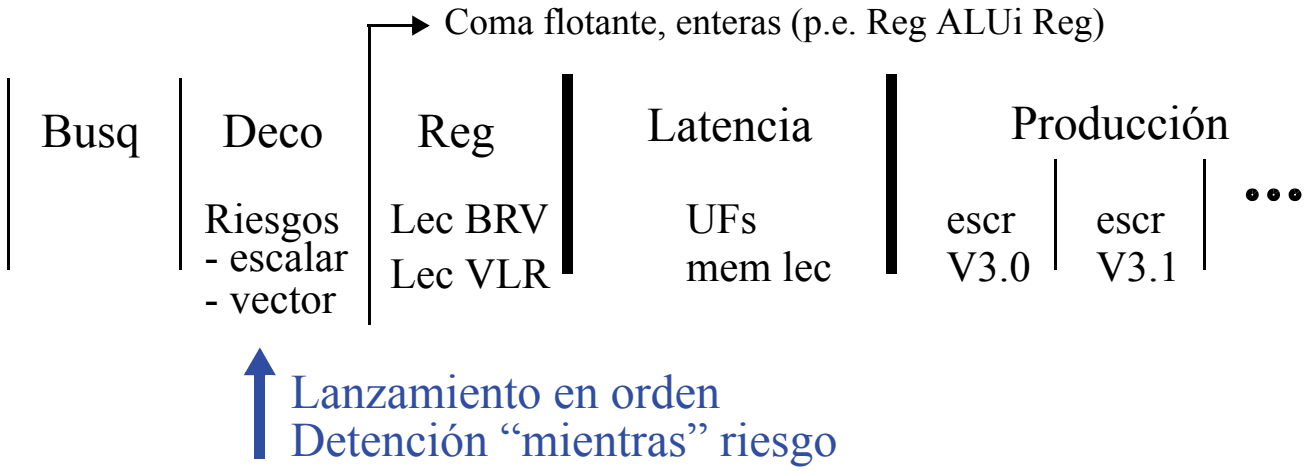
Ej2. @ = 48 → fila 1, columna 2 (e6, e7, e8, ...)

MAR	**mar_in**	1		2				3				
	MAR_H		1		2				3			
MAR_0						2				3		
MAR_1							2					
MAR_2				1				2				
MAR_3					1				2			
Actividad B0						e8				e12		
Actividad B1							e9				e13	
Actividad B2				e6				e10				
Actividad B3					e7				e11			
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*ciclos*
OUT			L = 1 + 4					e6	e7	e8	e9	e10	

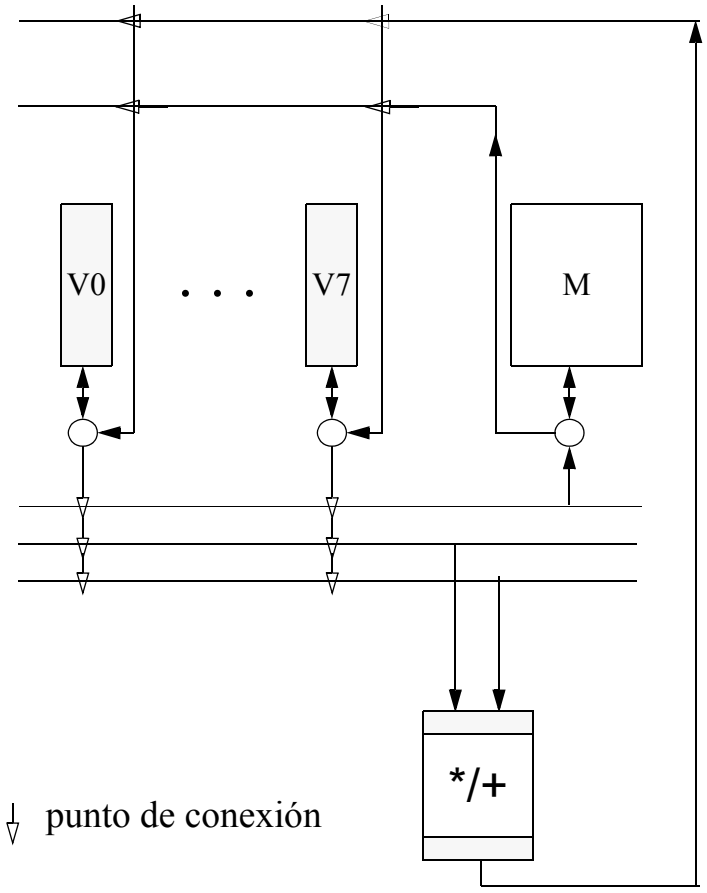
- BRV con 2L + 1E (y reg. vec. mínimos)
  - ✓ 2 buses L: BRV → {M ⊕ ALU}
  - ✓ 1 bus E: {M ⊕ ALU} → BRV
- Memoria de 1 puerto {L ⊕ E}. Sólo 1 UF



- Segmentación. Ejemplo.



- 3 buses lectura: BRV → {M, ALU}
- 2 buses escritura: {M, ALU} → BRV



- Instrucciones independientes y sin riesgos estructurales

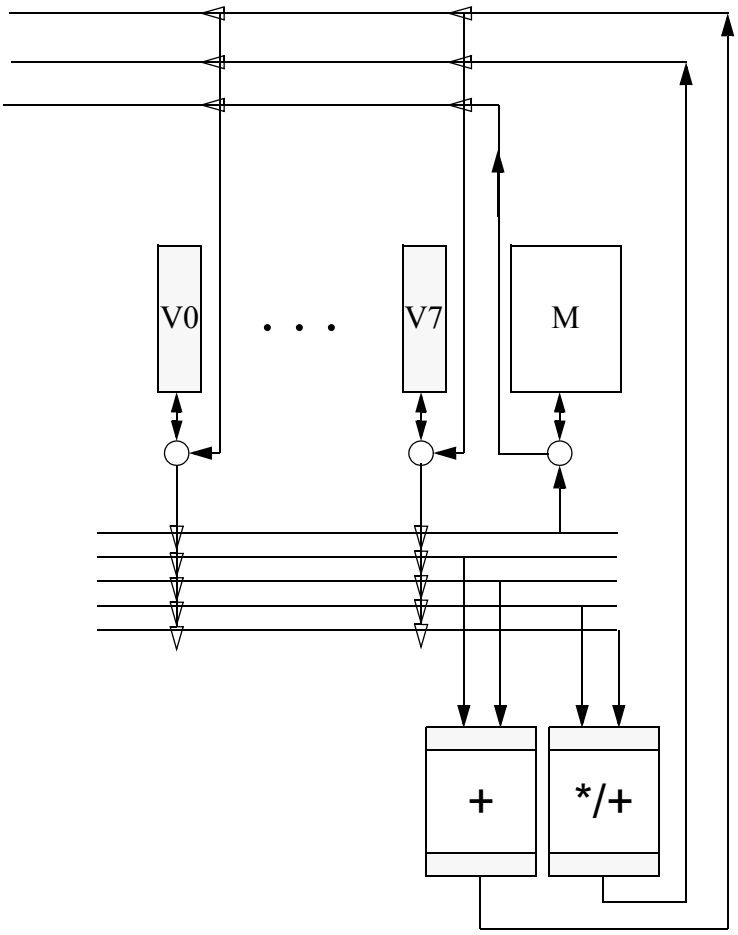
B	D^1	R	L		W^{VLR}
	B	D^1	R	L	W^{VLR}

- Instrucciones dependientes o con riesgos estructurales

B	D^1	R	L			W^{VLR}							
	B	D^1	D^2	D^3	D^4	D^5	...	...	...	...	D^n	R	L

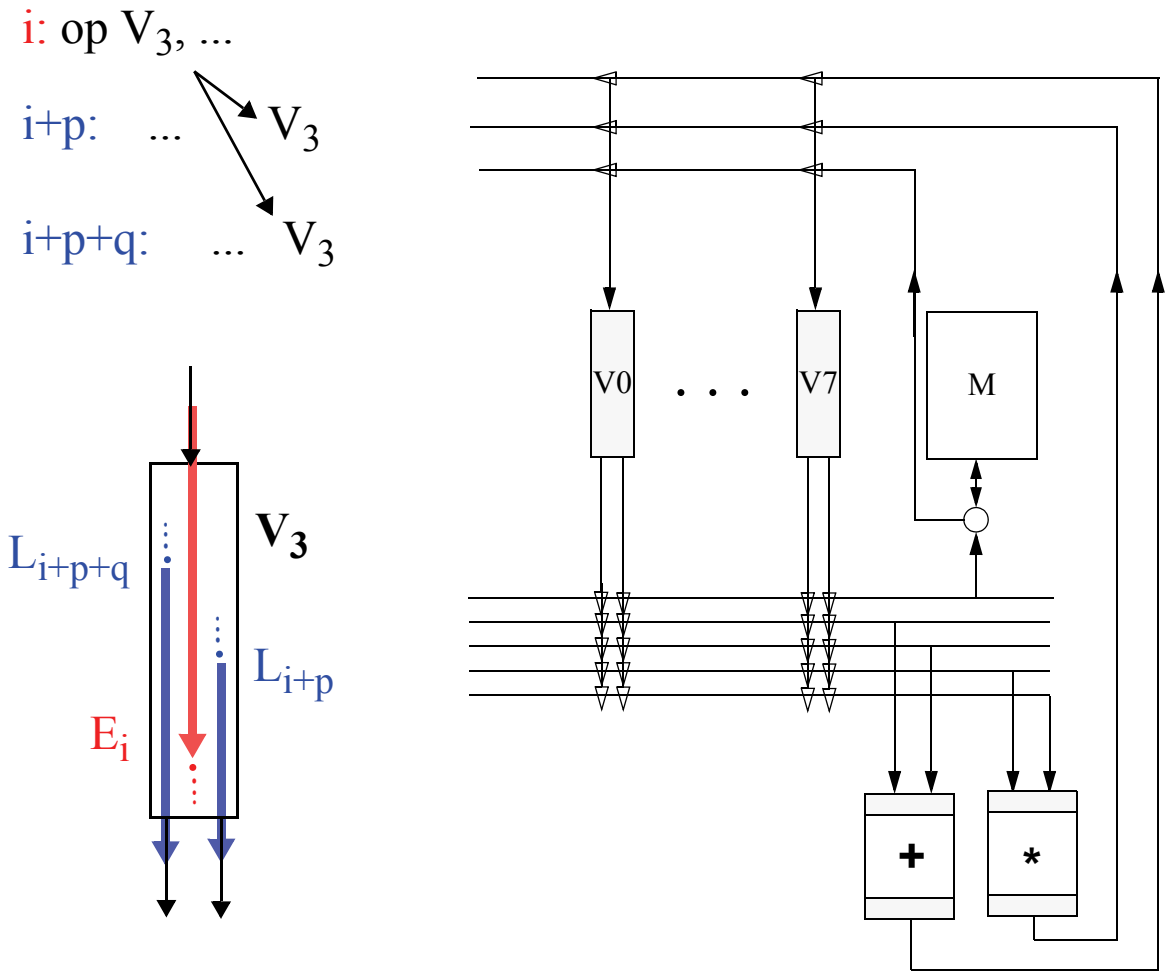
✓ Regla: último D = último W

- 2 UFs
  - ✓ ALU para sumar
  - ✓ ALU para sumar/multiplicar
- 3 buses de Escritura y 5 de Lectura



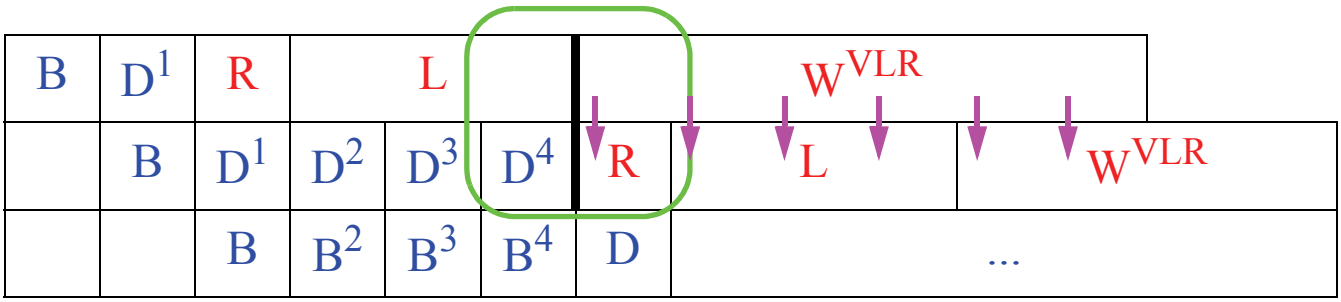
- Dos ALUs dedicadas
- **Dos** puertos de lectura y **uno** de escritura por reg. vectorial.
  La Unidad de Control permite dos flujos de lectura y uno de escritura en cada registro vectorial

Encadenamiento general → solapar *dependencias* datos



- Las *antidependencias* tampoco bloquean !!
- Temporización riesgos de datos productor-consumidor:

- Instrucciones dependientes



Encadenamiento ↓

- "Primera lectura de registros bajo primera producción"

  o bien,

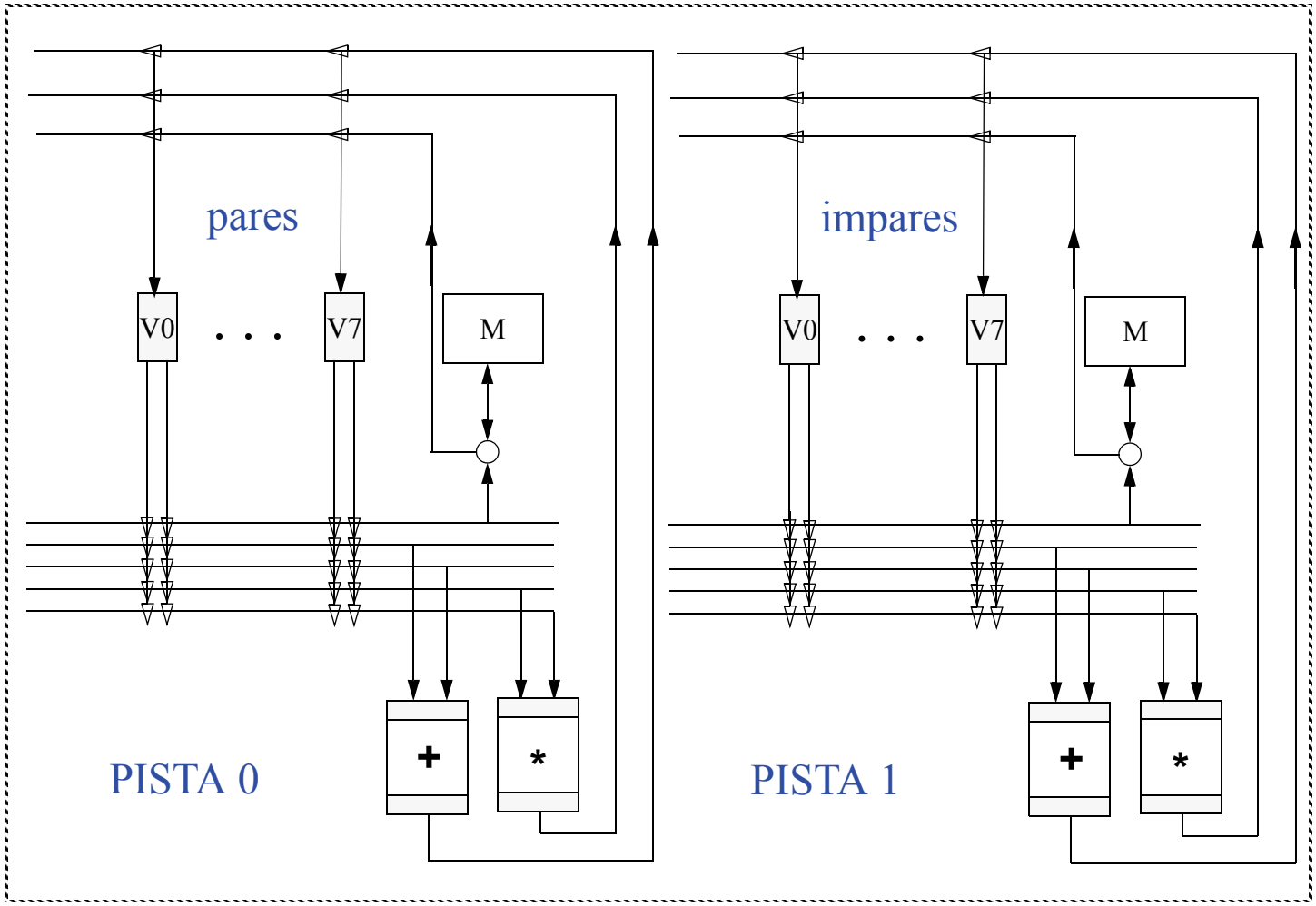
  Regla: último D = último L

- **Ejercicio**. Para el siguiente código:
  - ✓ dibujar el diagrama de ciclos
  - ✓ en el *primer ciclo de encadenamiento*
    - dibujar flechas verticales de productor a consumidor
    - determinar las posiciones de los punteros de escritura y lectura de los registros V1 y V4

```
addv    V1,V2,V3
 lv     V4,(r1)
addv    V5,V4,V1
```

- Para casa: cambiar **addv** V5,V4,V1 por **mulv** V5,V4,V1

- Replicar los caminos de proceso y memoria.
  Ejemplo: el Processador 4 con ***2*** *parallel lanes*



- El Banco de Registros Vectorial (BRV) y la memoria se *particionan*: pares en una partición e impares en la otra.

- Cada una de las dos ALUs (FP add, FP mul) se *duplican*.

- En el primer ciclo de ejecución entran en paralelo a las dos pistas los elementos (0, 1) de los registros vectoriales fuente. En el segundo ciclo entran los elementos (2, 3), y así sucesivamente ...

- Comparando con el Procesador 4 de una pista, el número de ciclos por elemento, **Ce**, se divide por el número de pistas.

código ejemplo

```
lv    V0,Ra
mulv  V1,V2,V3
addv  V4,V5,V6
```

GFLOPS ?

- El rendimiento depende de *n*, el tamaño del vector

  supongamos $n \le \text{MVL}$[1]

  - ✓ $\mathbf{C_n}\ (n \le \text{MVL}) = \mathbf{C_{fijos}} + n \cdot \mathbf{C_e}$ ciclos

    $\mathbf{C_{fijos}}$ = latencia UFs + penalizaciones

    penal. = riesgos estructurales y dependencias

    $\mathbf{C_e}$ = ciclos por elemento

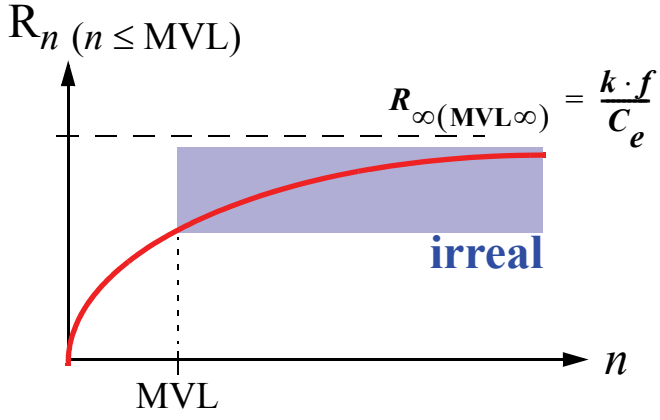
  - ✓ $\mathbf{T_n}\ (n \le \text{MVL}) = C_n \cdot Tc$

- **R** = velocidad en FLOPS; se aplican ***k*** FLOPs a cada elemento

  - ✓ $\mathbf{R_n}\ (n \le \text{MVL}) = n \cdot \boldsymbol{k} / T_n = n \cdot k \cdot F / C_n$

    R en *GFLOPS* → Tc en *ns* o F en *GHz*

$$R_n = \frac{n \cdot k}{(C_{fijos} + n \cdot C_e) \cdot Tc}$$

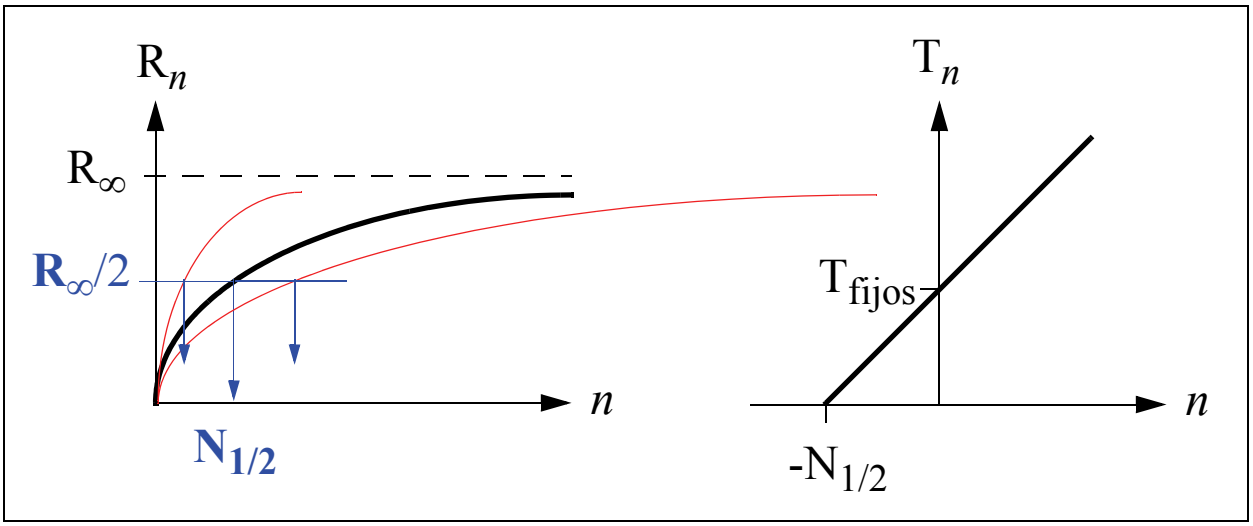


1. MVL: Maximum Vector Length, número de elementos de un Registro Vectorial.

- **$N_{1/2}$**

  longitud de vector que consigue
  MITAD del rendimiento máximo

$$R_{N1/2} = R_\infty(MVL\infty)/2 \quad \rightarrow \quad \mathbf{N_{1/2}} = C_{fijos}/C_e = T_{fijos}/T_e$$



  - ✓ Da idea de la *sobrecarga* vectorial (latencias y penalizaciones)
  - ✓ es independiente de T_c
    sólo depende de la arquitectura y del algoritmo

- **N_v**

  Longitud mínima de vector que consigue
  igual velocidad
  en las versiones escalar y vectorial

$$R_{Nv} = R_{esc} \quad \rightarrow \quad \mathbf{N_v} = C_{fijos} / (C_{esc} - C_e) = T_{fijos} / (T_{esc} - T_e)$$

  - ✓ mide la *sobrecarga* vectorial y la velocidad relativa entre los procesadores vectorial y escalar

- Supongamos los siguientes parámetros para los cinco procesadores anteriores:

  - ✓ T_c = 1 ns
  - ✓ Lat { +, *, mem } = 6, 7, 12 ciclos
  - ✓ LI = 1 ciclo
  - ✓ Segmentación página 39:
    - aritméticas, loads: B, D^n, R, Lat, W^{VLR}
    - stores: B, D^n, R, W^{VLR}, Lat

  n = 1 si no hay riesgo estructural ni dependencia;

  n > 1 en caso contrario

  VLR = valor del registro VLR ∈ {0..64}

- Vamos a calcular

  C_n ($n \leq$ MVL)

  R_{64} (MVL = *64*)

  R_∞ (MVL ∞)

  R_{pico}

C*n* se calcula con reg. vect de *n* elem.

R_{64} con $n = 64$

R_∞ con $n = \infty$ $\rightarrow R_\infty$ (MVL ∞)

Código	Proc *i* (Rpico, GFLOPS) C_n ciclos R_{64} GFLOPS R_∞ GFLOPS

Cód 1

```
lv    V0, Ra
mulv  V1, V2,V3
addv  V4, V5,V6
```

Cód 2

```
mulv  V1, V2,V3
addv  V4, V5,V2
```

Cód 3

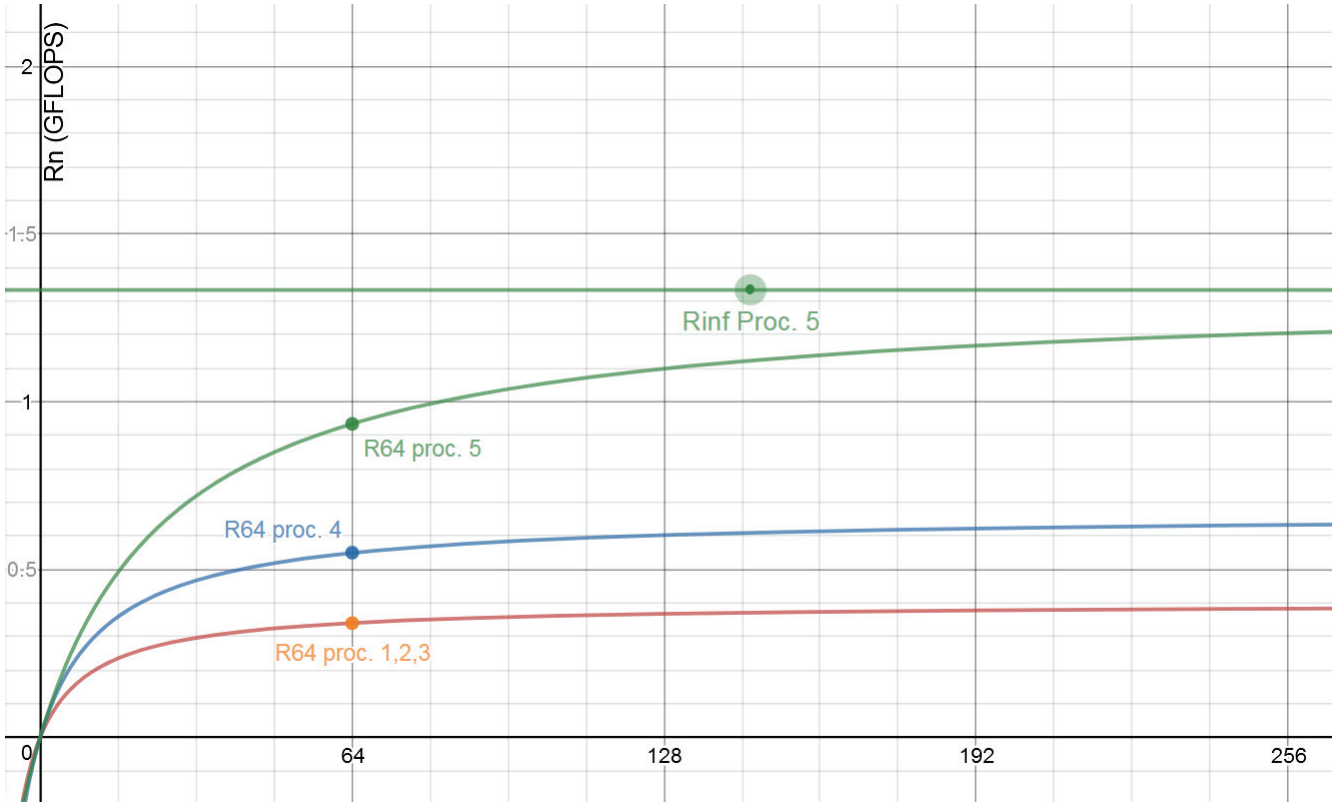
```
lv    V0, Ra
mulsv V1, F0,V0
addv  V2, V2,V1
```

Cód 4

```
lv    V0, Ry
lv    V1, Rx
mulsv V2, Fa,V1
addv  V3, V0,V2
sv    Ry, V3
```

	Proc1 (1)	Proc2 (1)	Proc3 (2)	Proc4 (2)	Proc5 (4)
Cód 1	30 + 3n 0.58 0.66	18 + 2n 0.88 1	15 + n 1.62 2	=	?
Cód 2	17 + 2n 0.88 1	=	=	10 + n 1.73 2	?
Cód 3	no	no	no	30 + n 1.36 2	?
Cód 4	56 + 5n 0.34 0.4	=	=	41 + 3n 0.55 0.66	?

https://www.desmos.com/calculator/lcvokcya7s



(página en blanco)

2.6 ZV: Organización de un procesador vectorial segmentado con ALMa tipo DLXV

Un procesador vectorial de una arquitectura tipo DLXV se segmenta en 8 etapas para las instrucciones vectoriales básicas: `aluV`, `LV`, `SV`. El trabajo en cada etapa para cada instrucción se muestra en la tabla:

<table>
<tr><th></th><th></th><th>aluV</th><th>LV</th><th>SV</th></tr>
<tr><td>B</td><td>búsqueda instrucción</td><td colspan="3" rowspan="3"></td></tr>
<tr><td>D1</td><td>deco 1/ @dst salto</td><td colspan="2">lectura BRint</td></tr>
<tr><td>D2</td><td>deco 2</td><td colspan="2">cálculo @ base</td></tr>
<tr><td>L</td><td>¿riesgos en recursos?
¿riesgos en registros?
lanzamiento y
reserva recursos</td><td>lectura reg
vectoriales</td><td>nada</td><td>lectura reg
vectoriales</td></tr>
<tr><td>X1</td><td>red ida</td><td>paso crossbar 1</td><td colspan="2">generación @</td></tr>
<tr><td>alu/
mem</td><td>operación o
acceso a memoria</td><td>op</td><td>mem_rd</td><td>mem_wr
liberar recursos
en último ciclo</td></tr>
<tr><td>X2</td><td>red vuelta</td><td colspan="2">paso crossbar 2</td><td></td></tr>
<tr><td>W</td><td>escritura</td><td colspan="2">escritura registro vectorial destino
liberar recurso en último ciclo</td><td></td></tr>
</table>

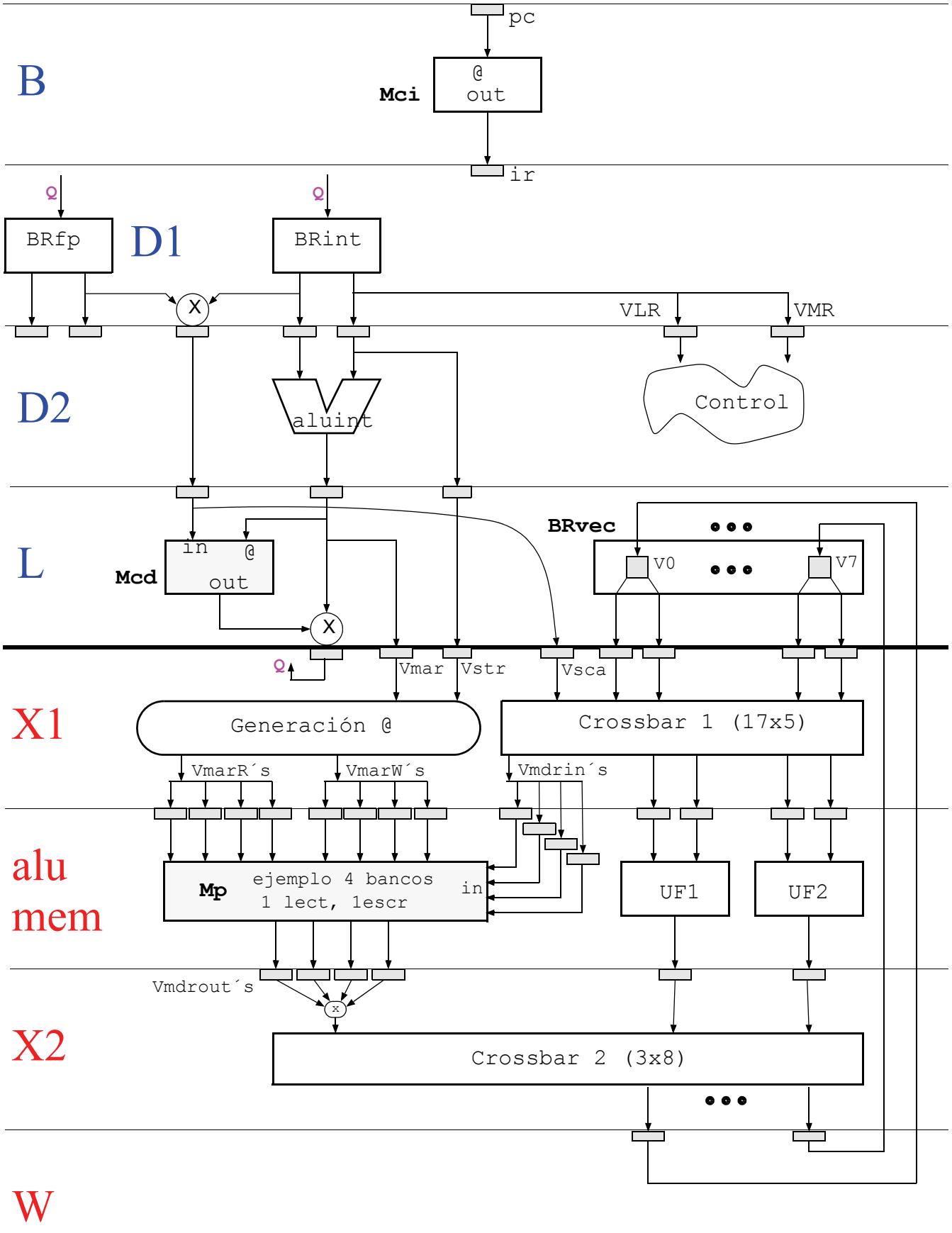
La ruta de datos se muestra en la siguiente hoja. La parte izquierda corresponde a un trozo del procesador escalar. Su operación y su relación con la parte vectorial puede deducirse observando la ruta de datos.

Hay dos unidades de cálculo multioperación (UF1 y UF2) y una memoria *multibanco de dos puertos*, uno de escritura y otro de lectura. O sea, cada banco de memoria soporta una lectura y una escritura simultáneas, con la misma latencia que una lectura o una escritura por separado. La latencia de la primera lectura no depende del banco de inicio porque el *acceso es desfasado*.

No hay control hardware de colisión entre lectura y escritura en memoria. Se solapan los flujos y el compilador ya se preocupará de *no* generar código vectorial si las *dependencias en memoria* pueden dar lugar a riesgos (RAW, WAR o WAW).

Cada registro vectorial tiene dos puertos de lectura y uno de escritura. La unidad de control permite ejecutar de forma solapada y segura a instrucciones dependientes en cualquier caso:

- Dependencia prod-cons. Encadenamiento general.
- Antidependencia. Flujo de escritura por detrás del de lectura.


B
pc
Mci
@
out
ir
D1
BRfp
BRint
X
VLR
VMR
D2
aluint
Control
L
Mcd
in
@
out
BRvec
V0
V7
X
Vmar
Vstr
Vsca
X1
Generación @
Crossbar 1 (17x5)
VmarR´s
VmarW´s
Vmdrin´s
alu
mem
Mp
ejemplo 4 bancos
1 lect, 1escr
in
UF1
UF2
Vmdrout´s
X2
Crossbar 2 (3x8)
W

En una instrucción vectorial de tamaño mínimo (VLR = 1) se ocupa cada etapa un ciclo, salvo alu/mem, que se ocupan un número de ciclos igual a la latencia de las unidades funcionales o de memoria. Para VLR>1 todas las etapas por debajo de L pueden ser multiciclo. Los siguientes ejemplos muestran la ocupación de las etapas para **VLR=6**:

Ejemplo 1: ADDV, Lat_sum = 2, Tasa iniciación = 1

B	x													
D1		x												
D2			x											
L				x	x	x	x	x	x					
X1					x	x	x	x	x	x				
+						x	x	x	x	x	x	x		
X2						Lat_sum		x	x	x	x	x	x	
W								x	x	x	x	x	x	x

resumen:

B	D1	D2	L	X1	+ 2c	X2	W 6c

:

Ejemplo 2: LV, Lmem = 4, sin conflictos de banco

B	x														
D1		x													
D2			x												
L				x											
X1					x	x	x	x	x	x					
mem						x	x	x	x	x	x	x	x	x	
X2						Lat_mem				x	x	x	x	x	x
W											x	x	x	x	x

resumen:

B	D1	D2	L	X1	mem 4c	X2	W 6c

:

Ejemplo 3: SV, Lmem = 4, sin conflictos de banco

B	x													
D1		x												
D2			x											
L				x	x	x	x	x	x		Lat_mem			
X1					x	x	x	x	x	x				
mem						x	x	x	x	x	x	x	x	x
X2														
W														

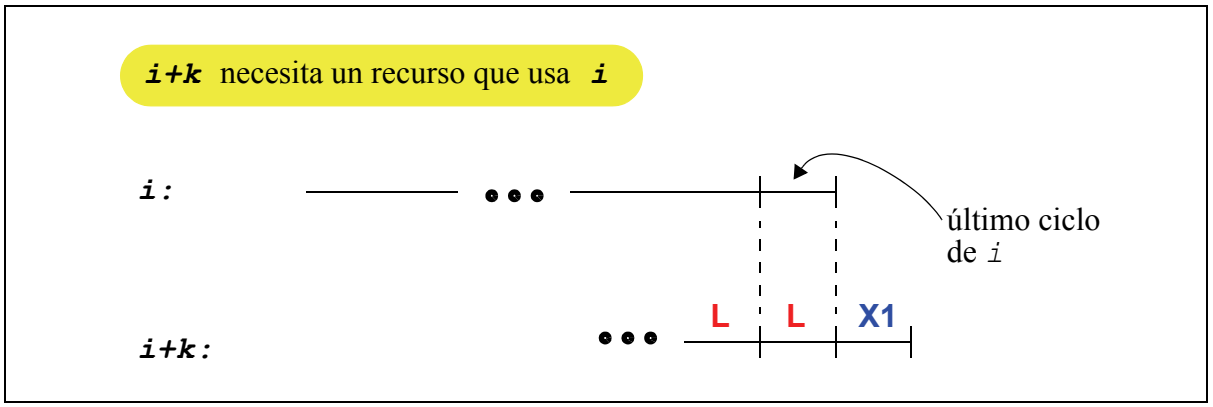
resumen:

B	D1	D2	L	X1 6c	mem 4c

Este procesador lanza en orden hasta 1 instrucción vectorial o escalar por ciclo. Instrucciones escalares y vectoriales independientes pueden solaparse. Pueden ejecutarse a la vez hasta cuatro instrucciones vectoriales (1 LV, 1 SV y 2 aluV)

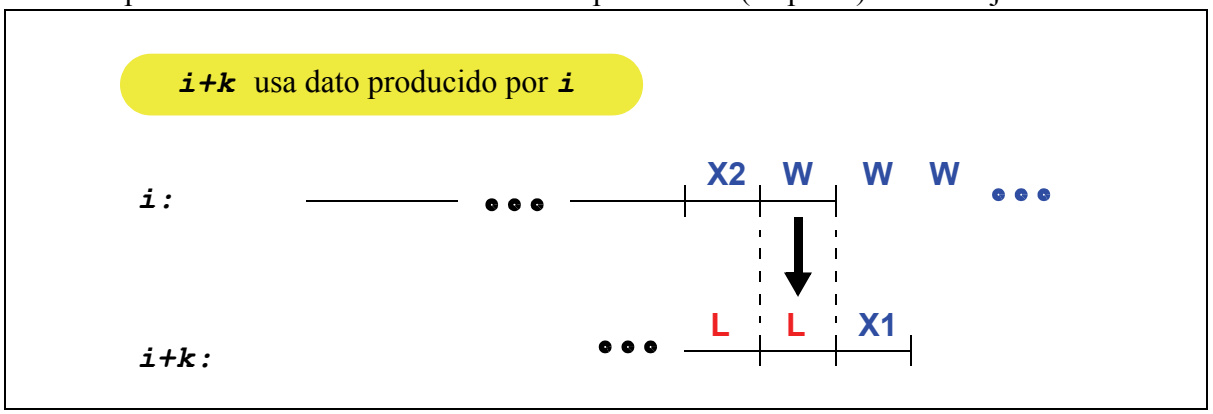
La etapa **L** se encarga de controlar los riesgos en las instrucciones vectoriales por dependencias de datos (en registros o memoria) y por falta de recursos:

- RECURSOS (UFs, puertos de memoria, puertos de registros, ...). Se reservan en la etapa **L** y se liberan en el *último* ciclo de ejecución. En caso de bloqueo por recurso, el último ciclo de detención coincide con el ciclo de liberación. Ver dibujo.



- DEPENDENCIAS prod/cons en REGISTROS. Existe encadenamiento **general**. No existen restricciones, incluso `Loads` y `Stores` encadenan.

  Puede solaparse la primera lectura de una instrucción consumidora (etapa **L**) con la primera escritura de una instrucción productora (etapa **W**). Ver dibujo.



El procesador ZV de referencia tiene las siguientes características:

MVL	64	tamaño registros vectoriales
+	Lsum = 2c, TI = 1c	Latencia, Tasa de Iniciación
*	Lmul = 3c, TI = 1c	
mem	8 bancos, Lmem = 4c	sistema sobrado
Tc (*f*)	1ns (*1000 Mhz = 1GHz*)	
Coherencia	entre **Mcd** y **Mp** asegurada por hardware (ya veremos)	

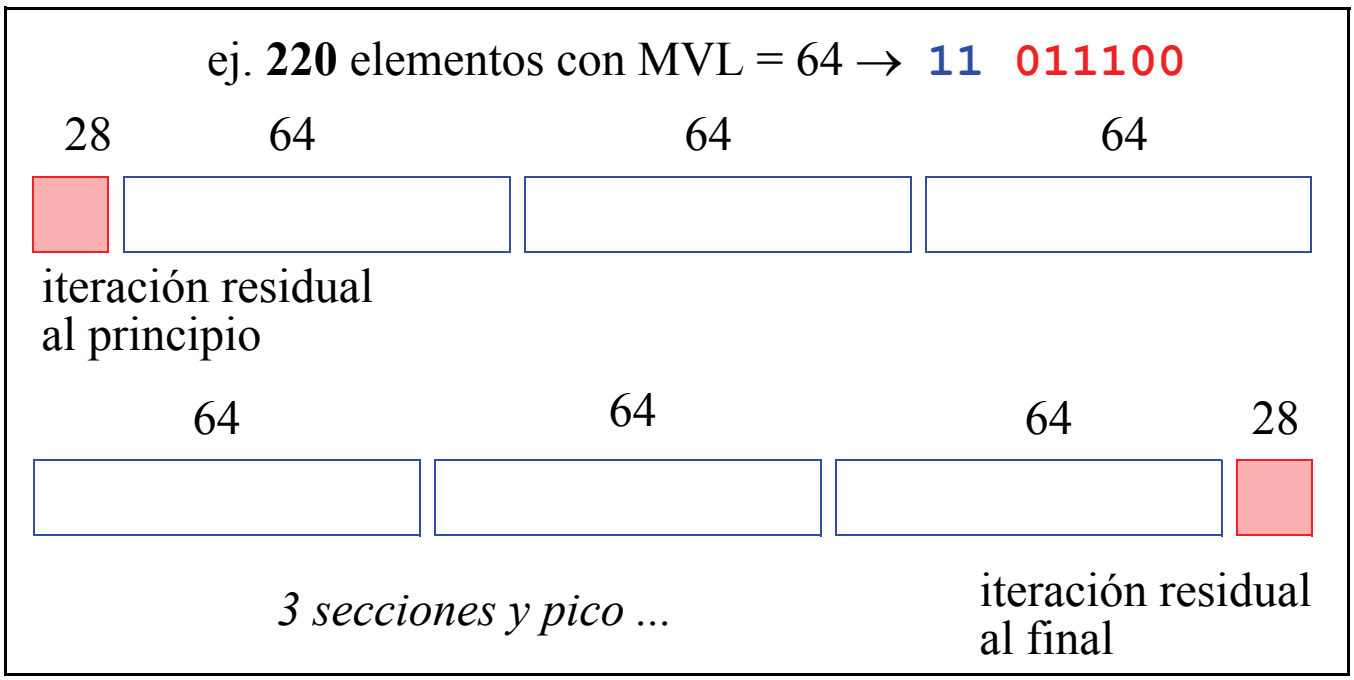
3.1 VECTOR LENGTH

Vectores de *n* elementos, pero
Registros vectoriales de MVL elementos:

SOLUCIONES:

- $n \leq 64$
  se programa VLR con *n* (instr. DLXV `movi2s`)
- $n > 64$

Técnica de Seccionado o *Strip-Mining*



- Módulo y división entera por MVL (potencia de dos):
  - Operaciones con máscara, desplazamientos
  - Soporte especial: p.e. repertorio de Convex[1]

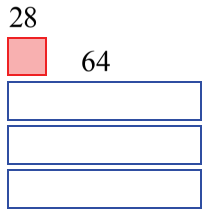
1. Paqui Quintana, R. Espasa y M. Valero. "An ISA comparison between Superscalar and Vector Processors". En Proc. of the Int. Conf. on Vector and Parallel Processing (VECPAR98). Lecture Notes in Computer Science, issue 1573, pp. 548-560, 1999. Springer Verlag publisher.

Código AXPY (SAXPY o DAXPY: simple o doble prec.)

```
DO i = 1, n
    y(i) = a · x(i) + y(i)
ENDDO
```

```
IND = 1
VL = n mod MVL
DO ii = 1, ⌈n/MVL⌉
    DO i = IND, IND+VL-1
        y(i) = a · x(i) + y(i)
    ENDDO
    IND += VL
    VL = MVL
ENDDO
```

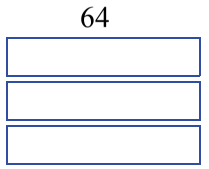
vect.



iteración residual al principio

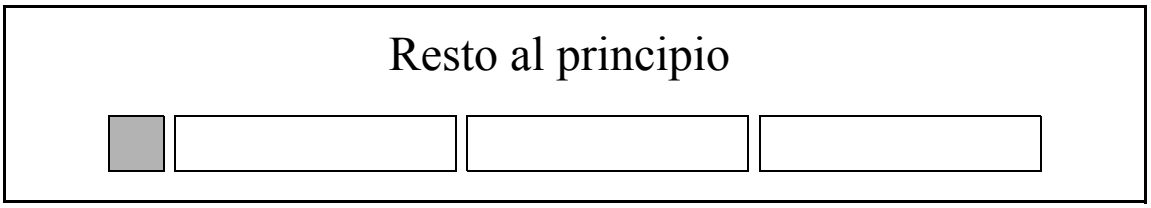
```
DO ii = 1, ⌊n/MVL⌋·MVL, MVL
    DO i = ii, ii+MVL-1
        y(i) = a · x(i) + y(i)
    ENDDO
ENDDO
DO i = ⌊n/MVL⌋·MVL + 1, n
    y(i) = a · x(i) + y(i)
ENDDO
```

vect.



iteración residual al final

Bucle ejecutado en modo vectorial (3 secciones y pico)



Código AXPY seccionado con MVL = 64

```
1     and    Rmod, Rn, R3F
2     srl    Rent, Rn, #6
3   movi2s   VLR, Rmod

buc:  LV     V0, Rx
5   MULSV    V1, Fa, V0
6     LV     V2, Ry
7   ADDV     V3, V2, V1
8     SV     Ry, V3

9   movs2i   Rn,VLR
10    bz     out, Rent
11    nop
12    sll    Rsize, Rn, #3
13    add    Rx, Rx, Rsize
14    add    Ry, Ry, Rsize
15  movi2s   VLR, #64
16    jmp    buc
out:  dec    Rent
```

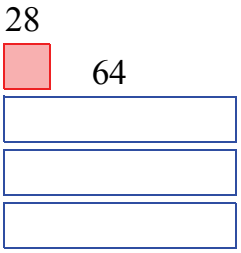
Rx = &X[0];

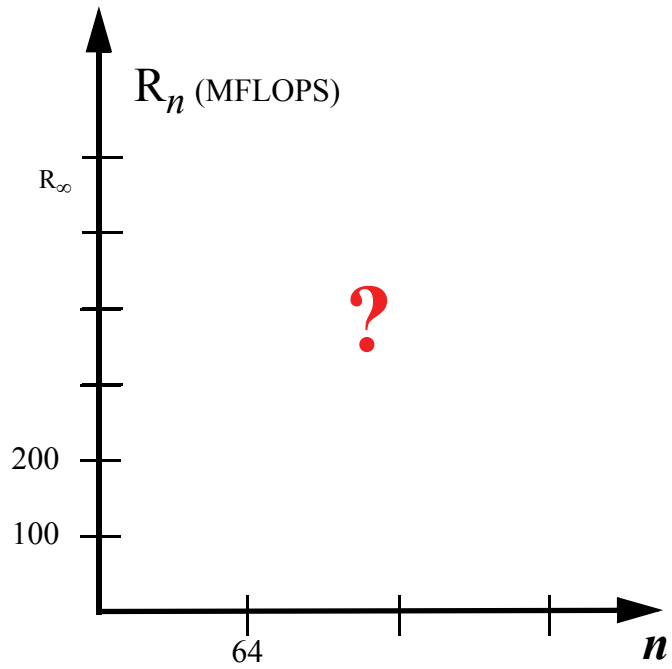
Fa = a

Ry = &Y[0];

Rn = *n*;

R_{3F} = 0x3F





Rendimiento de ZV en función de *n* ?

Modelo de ejecución

- C_{base} ciclos del prólogo y del epílogo:

  instr. escalares {1:3}

  tiempo desde último bloque n hasta final

- C_{fij} costes fijos (incluso con VLR = 1)

  Latencias + Penalizaciones

- C_{bucle} ciclos de control de bucle

  instr. escalares {9:17}
  en paralelo con las vectoriales anteriores

  - ✓ ciclos
    desde que se lanza la última instr. vect. de una sección
    hasta que se lanza la primera de la sección siguiente.
  - ✓ a veces, $C_{bucle} = 0$,
    porque la primera instr. vect. de la sección siguiente está detenida por recursos o dependencias.

$$C_n = C_{base} + \lceil n/MVL \rceil \cdot (C_{fij} + C_{bucle}) + n \cdot C_e$$

- A partir de C_n, podemos derivar
  R_n, R_{64}, R_∞, ...
  $N_{1/2}$, N_v

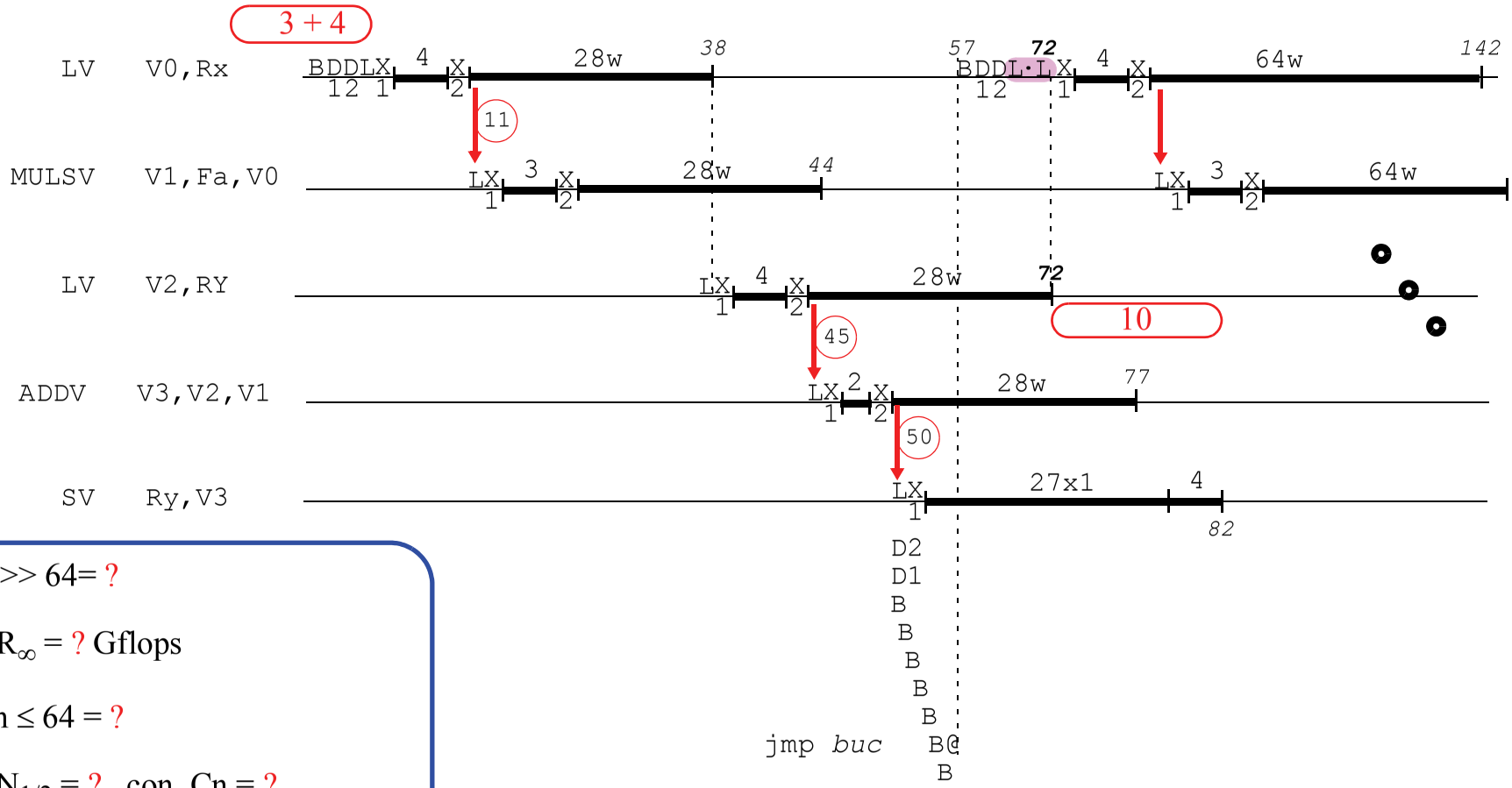
$\lceil n \rceil$	$$Rn(GFLOPS) = \frac{FLOPn}{Cn} \times F(GHz)$$

Ejercicio

obtener fórmulas para $N_{1/2}$ y N_v

Hipótesis: son menores que MVL

Ejecución detallada de AXPY de 220 elementos en ZV



Cn, n>> 64= ?

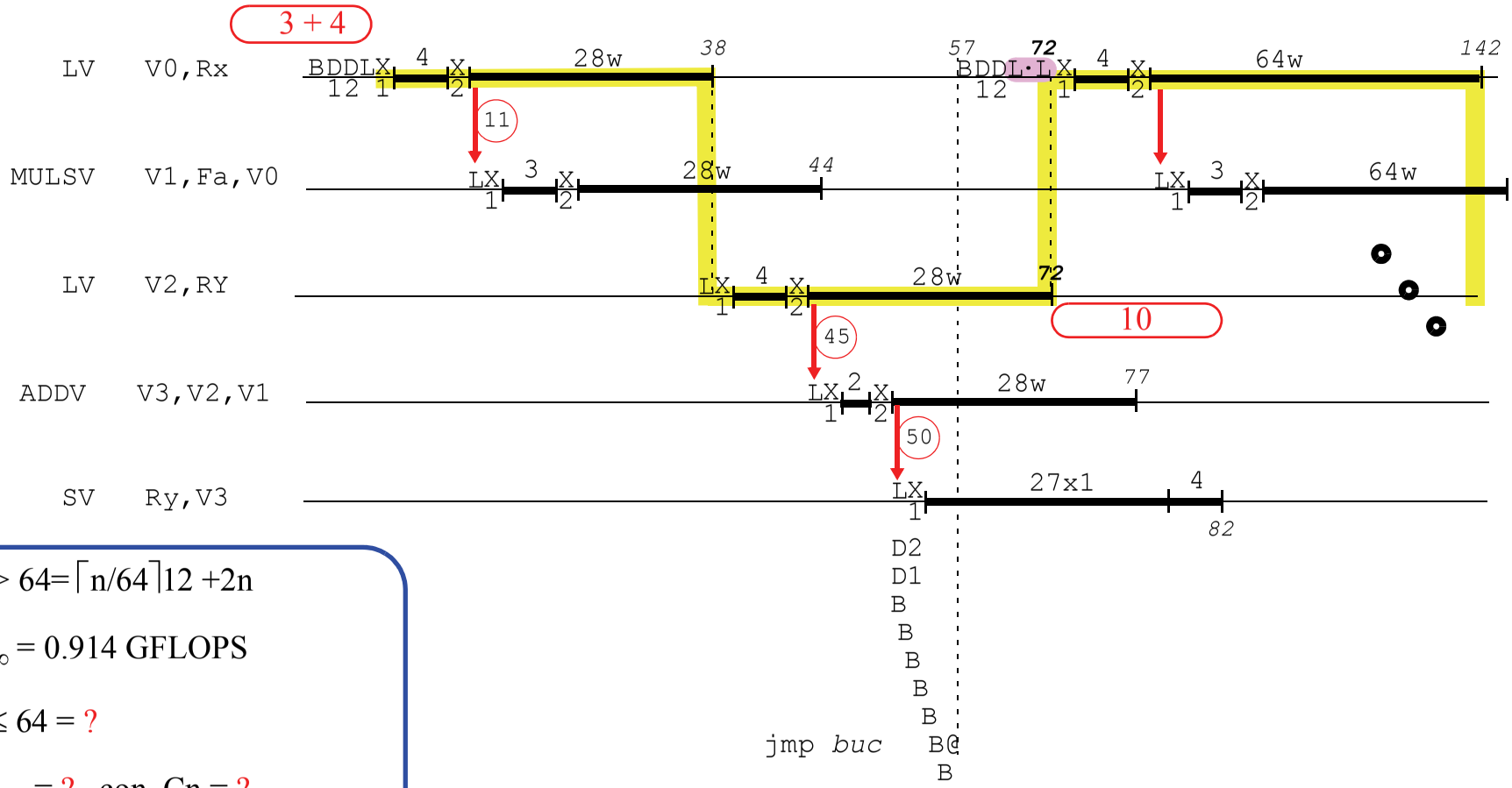
R_∞ = ? Gflops

Cn , n ≤ 64 = ?

$N_{1/2}$ = ? con Cn = ?

Cn, ∀n = ?

Ejecución detallada de AXPY de 220 elementos en ZV



Cn, n>> 64= $\lceil n/64 \rceil 12 + 2n$

R_∞ = 0.914 GFLOPS

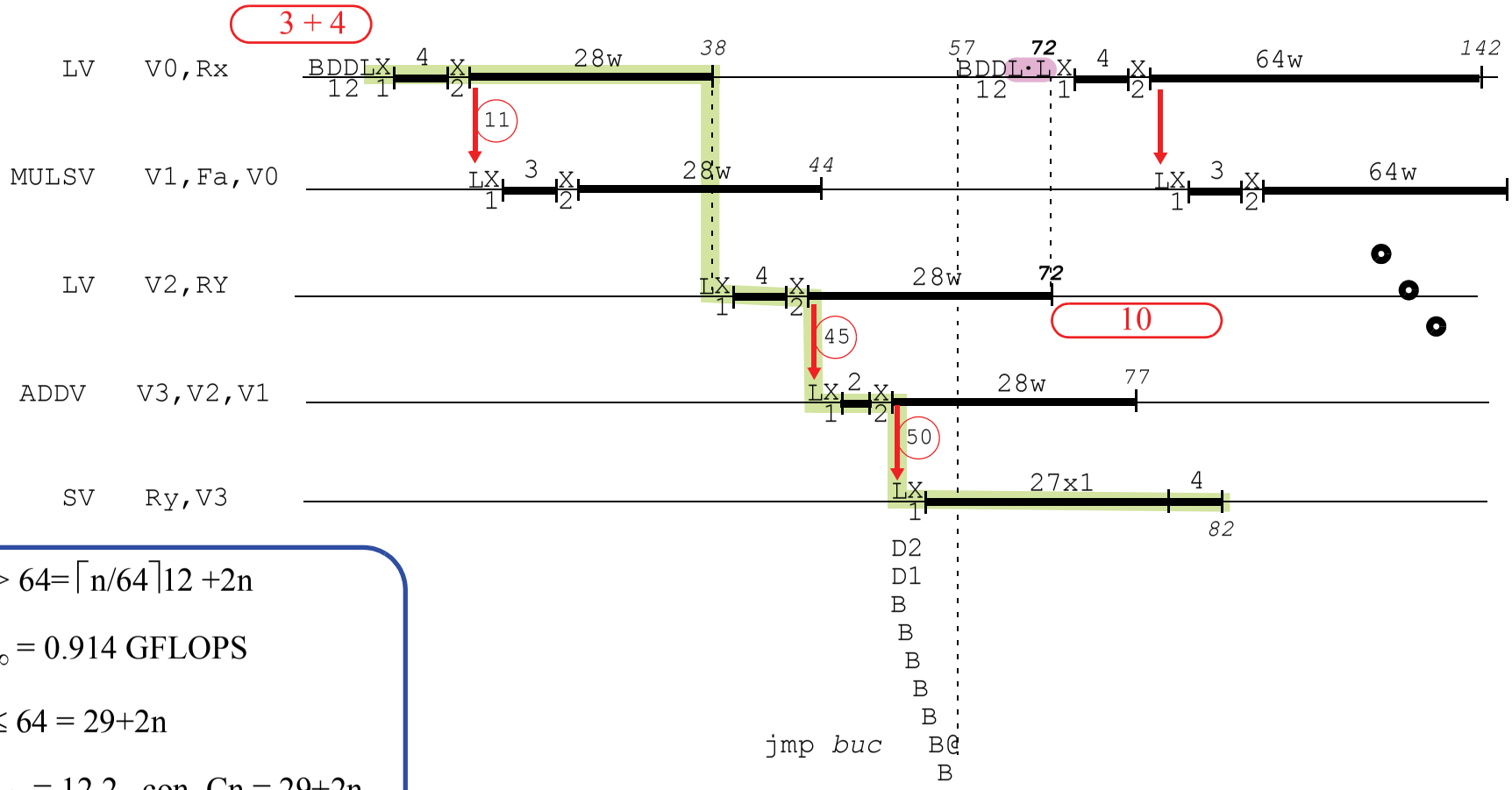
Cn , n ≤ 64 = ?

$N_{1/2}$ = ? con Cn = ?

Cn, ∀n = ?

Cn , n>>64 =

Ejecución detallada de AXPY de 220 elementos en ZV



Cn, n>> 64= $\lceil n/64 \rceil 12 + 2n$

$R_\infty = 0.914$ GFLOPS

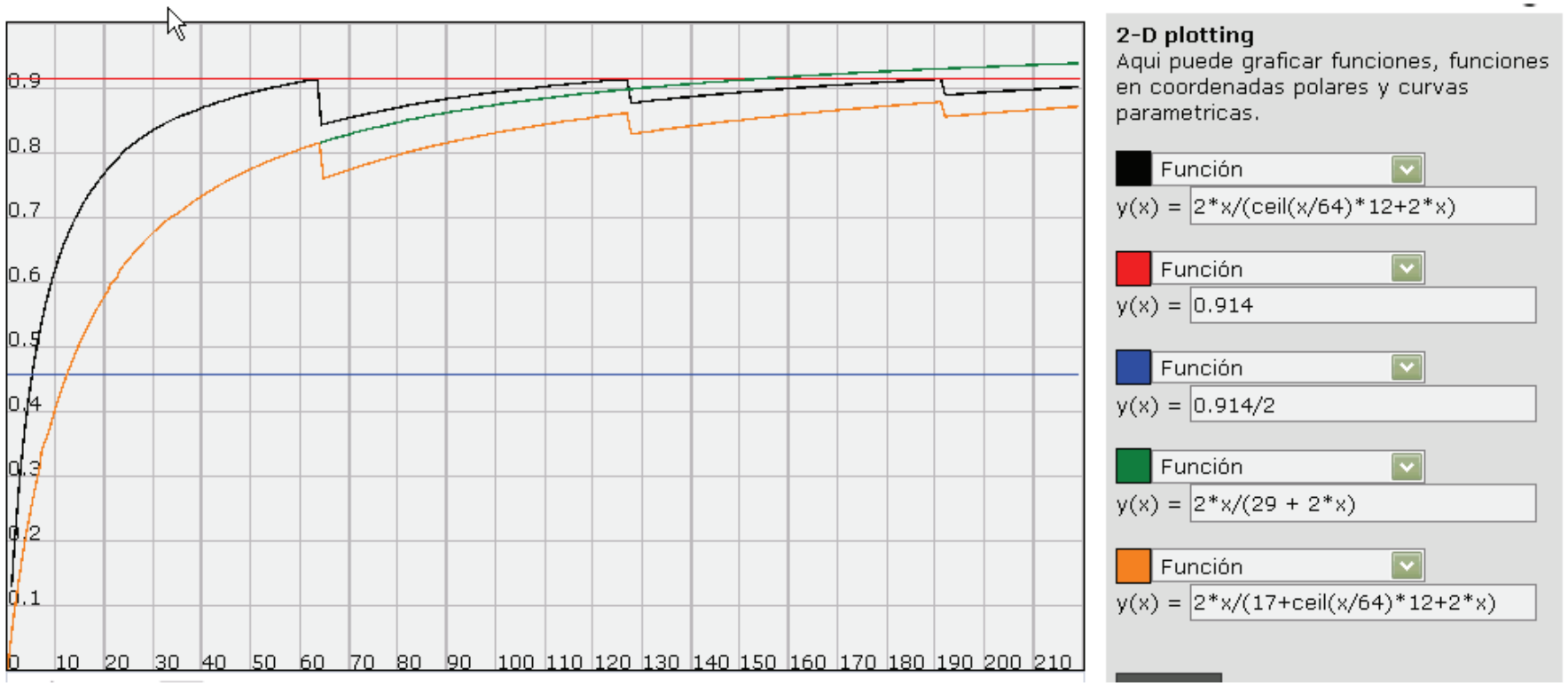
Cn , $n \leq 64 = 29+2n$

$N_{1/2} = 12.2$ con $Cn = 29+2n$

Cn, $\forall n = 7 + \lceil n/64 \rceil 12 + 2n + 10$

Cn , n>>64 =

Cn , $n \leq 64$ =



permalink:

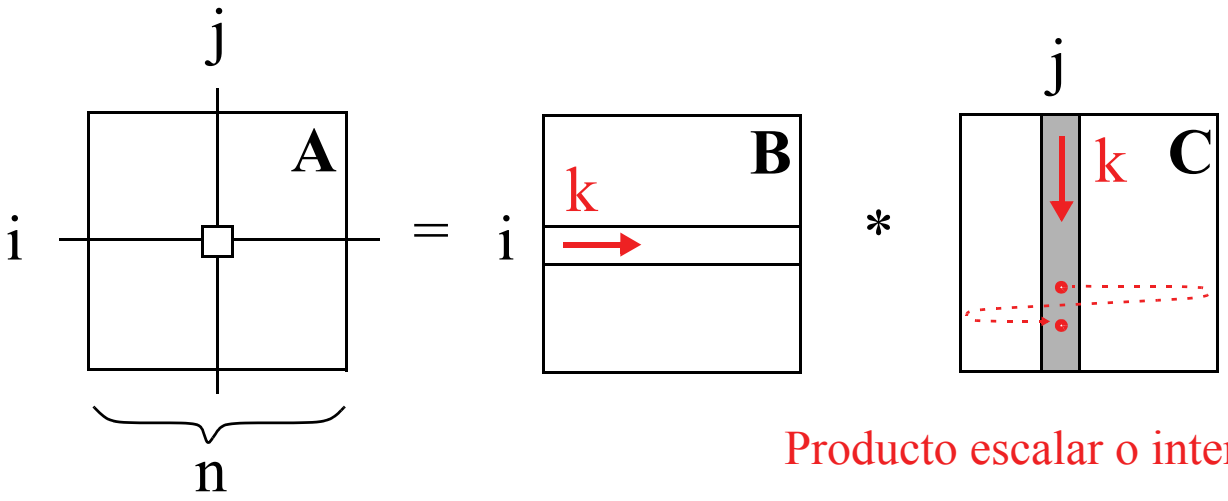
http://fooplot.com/index.php?&type0=0&type1=0&type2=0&type3=0&type4=0&y0=2*x/%28ceil%28x/64%29*12%2B2*x%29&y1=0.914&y2=0.914/2&y3=2*x/%2829%20%2B%202*x%29&y4=2*x/%2817%2Bceil%28x/64%29*12%2B2*x%29&r0=&r1=&r2=&r3=&r4=&px0=&px1=&px2=&px3=&px4=&py0=&py1=&py2=&py3=&py4=&smin0=0&smin1=0&smin2=0&smin3=0&smin4=0&smax0=2pi&smax1=2pi&smax2=2pi&smax3=2pi&smax4=2pi&thetamin0=0&thetamin1=0&thetamin2=0&thetamin3=0&thetamin4=0&thetamax0=2pi&thetamax1=2pi&thetamax2=2pi&thetamax3=2pi&thetamax4=2pi&ipw=1&ixmin=0&ixmax=220&iymin=0&iymax=1&igx=10&igy=0.1&igl=1&igs=0&iax=1&ila=1&xmin=0&xmax=220&ymin=0&ymax=1

- Procesado de elementos no contiguos

  a separación constante (*stride* = paso)

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
  { A[i][j] = 0.0;
    for (k=0; k<n; k++)
      A[i][j] = A[i][j] + B[i][k]*C[k][j];
  }
```

A = B*C



Producto escalar o interno

- `B[i][k]*C[k][j]` es vectorizable en K si cargamos `C[0:n-1][j]` en un reg. vectorial (si n>64 secc.)

- s = n*8 bytes = separación en un registro **R**

```
LVWS V1, (R1,R2)   ⟶   for (i=0; i<VLR; i++)
                         V1.i = mem (R1 + i*R2);
SVWS (R1,R2),V1
```

DLXV

- Vectorización de la multiplicación de matrices en forma *ijk*:

```
LV        V0, R_B
LVWS      V1, (R_C, R_n*8)
MULV      V2, V0,V1

RED_SUM   R_tmp, V2          ; R_tmp = R_tmp + sum_{i=0}^{VLR} V2.i

st        R_A, R_tmp
```

4. Conflictos en el Acceso a Bancos de Memoria

4.1 Introducción

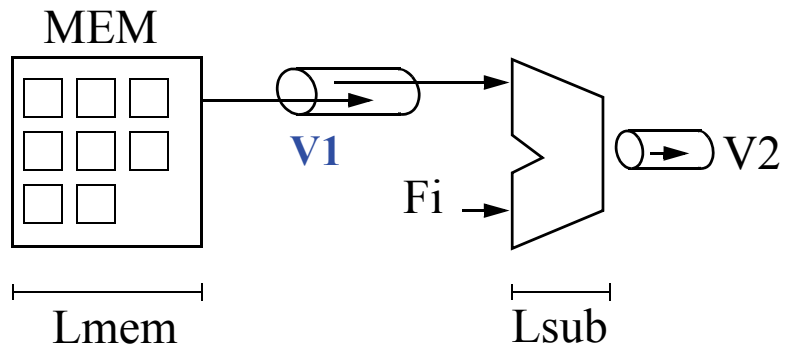
4.2 Sistemas Ajustados

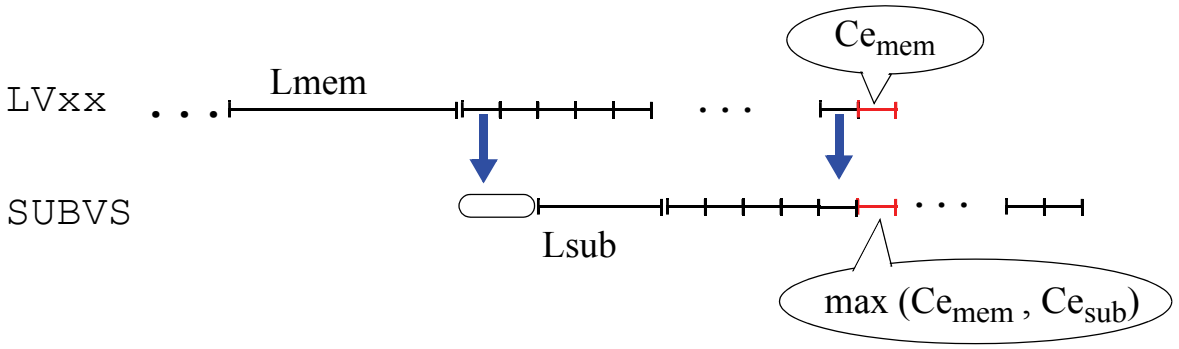
4.3 Sistemas Sobrados

4.1 INTRODUCCIÓN

Memoria = carga y descarga de registros vectoriales

S1: LVxx **V1**, ...
S2: SUBVS V2, **V1**, Fi





LV V1, R@ → Ce_{mem} = 1 ciclo/dato

LV**WS** V1, (R@,Rs) → Ce_{mem} = **?**

ALMACENAMIENTO

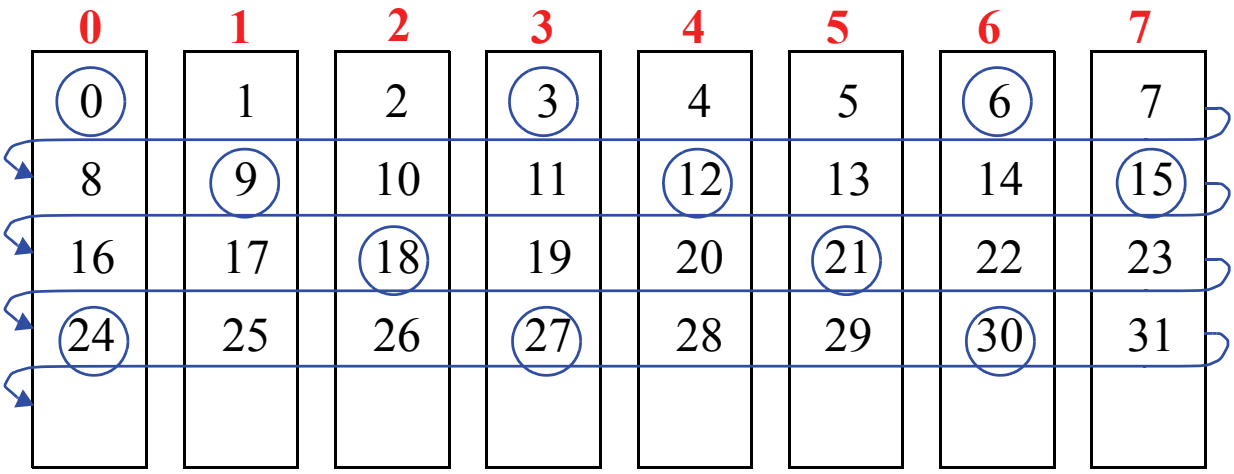
Palabras consecutivos en bancos consecutivos[1]

"bancos entrelazados por palabras ó entrelazado de menos peso"

Suponemos:

ancho banco = ancho BUS memoria/registros vect. = tamaño palabra

Ejemplo: **M = 8 bancos, S = 3**



PROPIEDAD FUNDAMENTAL

Una secuencia de accesos con separación S bancos visita un subconjunto de P bancos del total de M

$M = 8$

S	1	2	**3**	4	5	8
P	8	4				

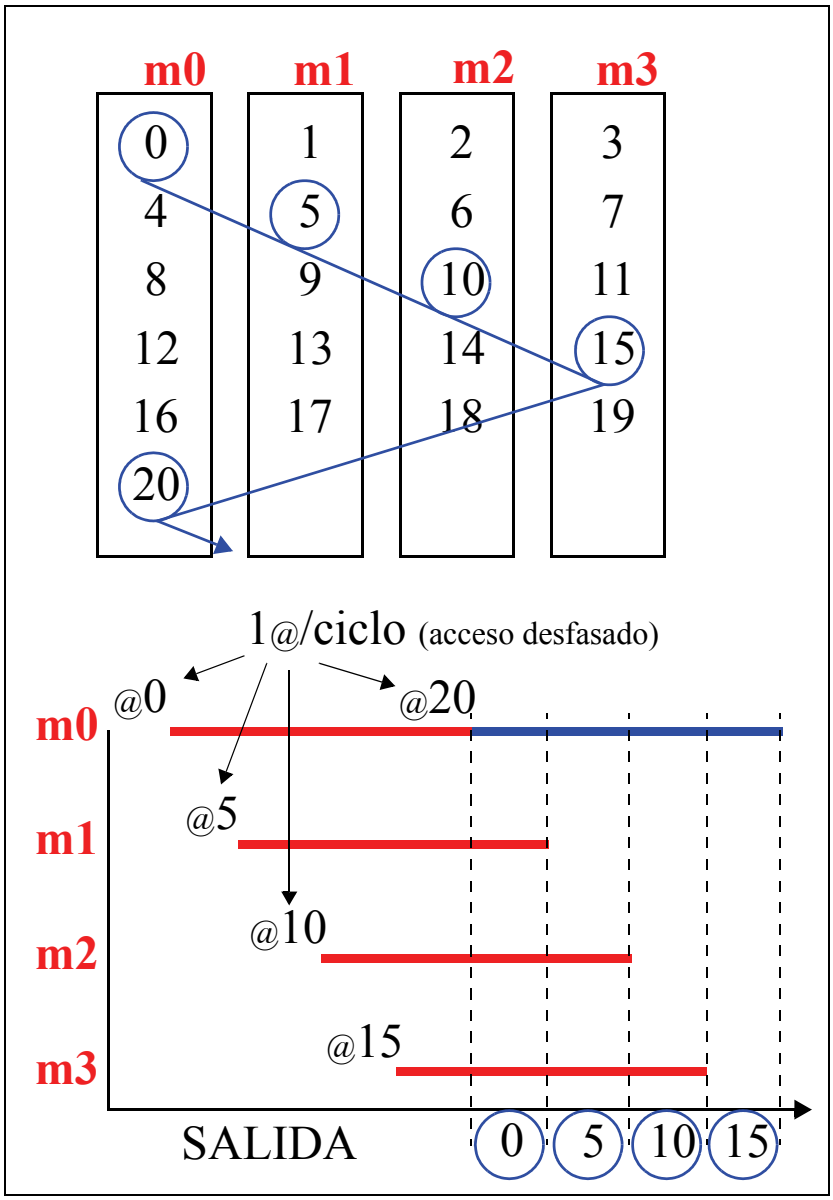
$$P = \frac{M}{mcd(S, M)}$$

1) En otros sistemas, p.e. en un multiprocesador, las palabras que forman un bloque de cache pueden residir consecutivas en el mismo banco de memoria principal. Se dice entonces que los bancos están entrelazados por bloques (no por palabras).

4.2 SISTEMAS AJUSTADOS: *M* = *L*

Ejemplo: ***L*** = 4 ciclos y ***M*** = 4 bancos

❒ Acceso sin conflicto ***S*** = 5 módulos



✓ $P = \frac{4}{mcd(5,4)} = 4$

✓ Cada banco diferente visitado aporta 1 dato / *L* ciclos

✓ Visitamos *P* bancos
Flujo = BW =
P datos / *L* ciclos

✓ Ce_{mem} = Flujo^{-1} =
L/*P* ciclos/dato =
1 cpe

❒ Acceso con conflicto ***S*** = 4 módulos

✓ únicamente visitamos el banco 0: $P = \frac{4}{mcd(4,4)} = 1$

✓ $Ce_{mem} = L/P =$ **4 cpe**

4.2 SISTEMAS AJUSTADOS: $M = L$

$$Ce = \frac{L}{P} = \frac{L \cdot mcd(M,S)}{M} = mcd(M,S)$$

- *Strides* IMPARES → sin conflicto → **$Ce_{mem} = 1$**

  $S = \sigma \bullet 2^0$, con σ impar

DEMO, suponiendo nº de bancos es potencia de 2, $M = 2^m$
$Ce = mcd(M,S) = mcd\,(2^m, \sigma \bullet 2^0) = 2^0 = 1$

- *Strides* PARES → con conflicto → **$Ce_{mem} > 1$**

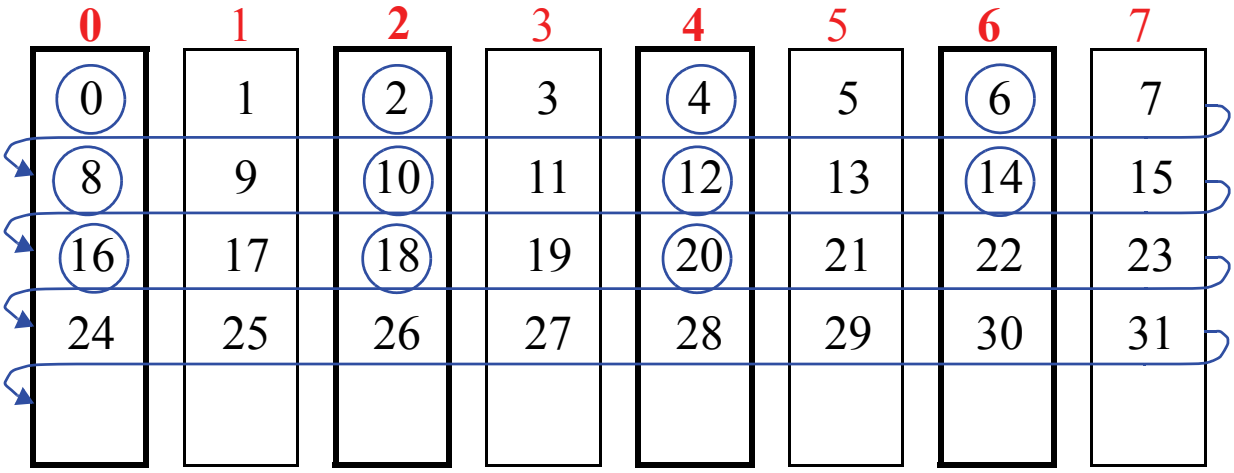
  $S = \sigma \bullet 2^k$, con σ impar y $k \geq 1$

DEMO ($M = 2^m$)
Conflicto → $Ce_{mem} > 1$
$Ce = mcd(M,S) = mcd(\sigma \bullet 2^k, 2^m) = 2^{min(k,m)}$

4.3 SISTEMAS SOBRADOS: *M* > *L*

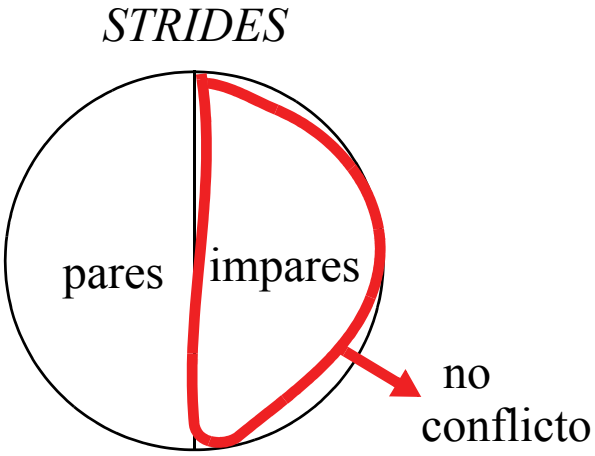
Ejemplo: $M = 8$, $L = 4$

- Acceso sin conflicto **S = 2**

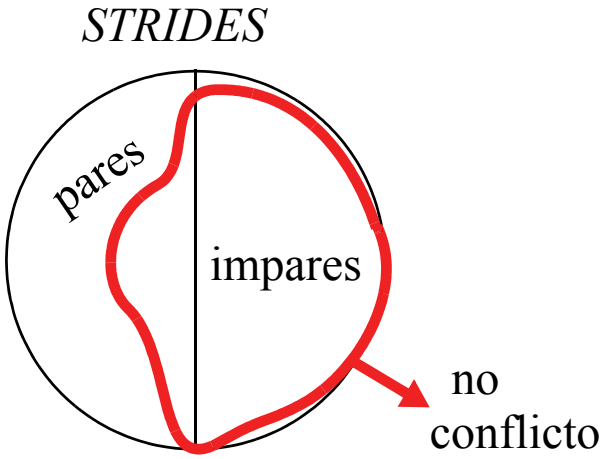


$$\text{Flujo} = \text{BW} = \left.\frac{Pelementos}{Lciclos}\right|_{P = L = 4} = 1 \text{ elemento/ciclo}$$

permitimos más *strides* libres de conflicto



sist. **AJUSTADOS**



sist. **SOBRADOS**

4.3 SISTEMAS SOBRADOS: $M > L$

Suponemos $L = 2^l$ (irreal)

- *Strides* sin conflicto: IMPARES y algunos PARES

$S = \sigma \bullet 2^k$, con σ impar y $\mathbf{0 \le k \le m\text{-}l}$

DEMO ($M = 2^m > L = 2^l$)
Sin conflicto → $P \ge L$ $P = \dfrac{2^m}{mcd(S, 2^m)} \ge 2^l \rightarrow 2^{m-l} \ge mcd(S, 2^m) = 2^{min(k, m)}$ $\rightarrow m - l \ge min(k, m) \rightarrow m - l \ge k$

No hay conflicto con PARES si tienen "pocas" potencias de dos.

- *Strides* con conflicto: parte de los PARES

$S = \sigma \bullet 2^k$, con σ impar y $\mathbf{k > m\text{-}l}$

DEMO ($M = 2^m$)	
Conflicto →	$Ce_{mem} > 1$ mcd (2^m, S) > 1

$$Ce_{mem} = \frac{L}{P} = \frac{L \bullet mcd(S, M)}{M} = \frac{2^l \bullet 2^{min(k, m)}}{2^m} = 2^{l + min(k, m) - m}$$

Ejemplo: Ce (sistema L, M, S) = Ce(8, 16, 30) = 

5. ARQUITECTURA DLXV

Repertorio completo de instrucciones

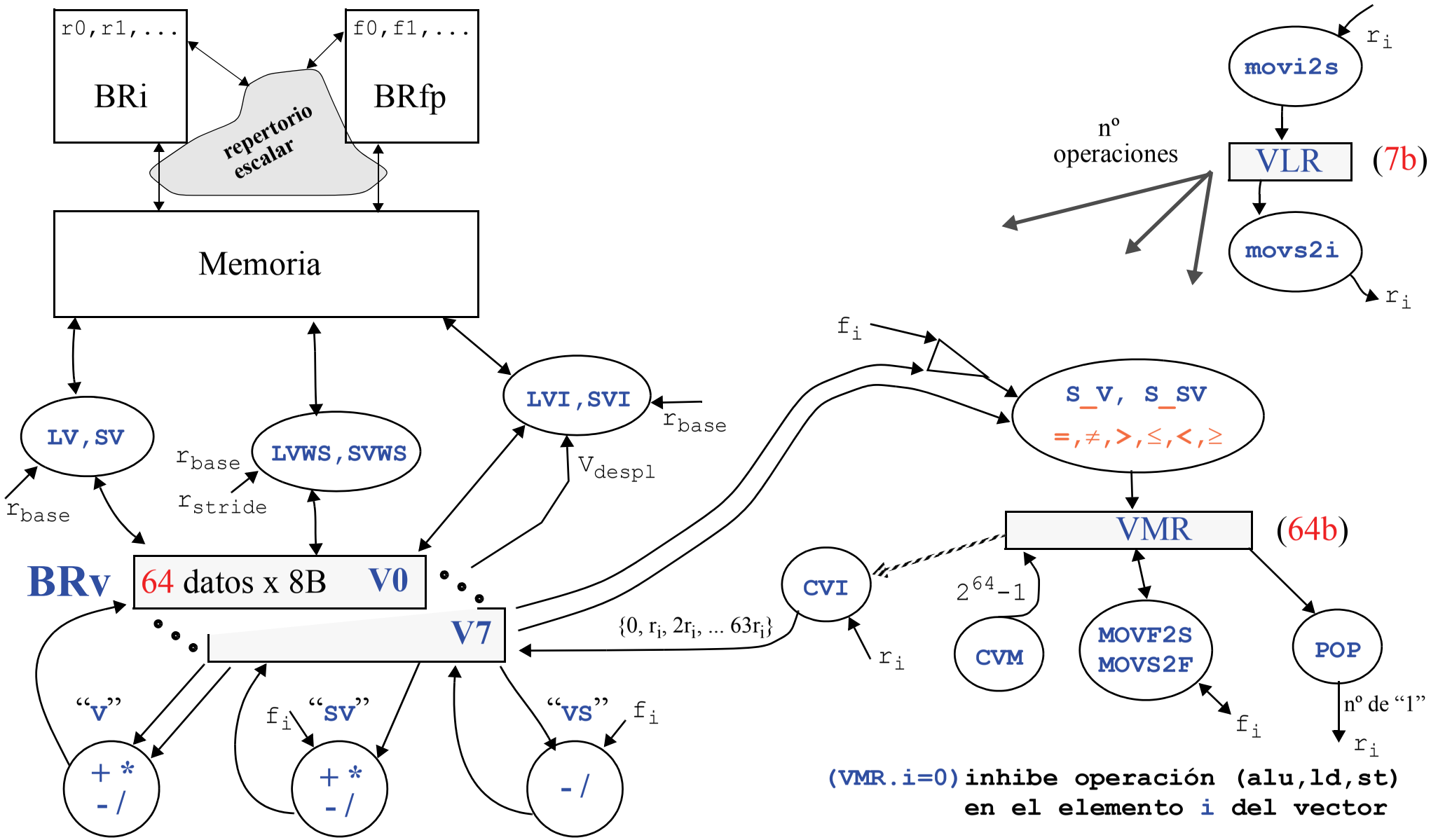
DLXV - VMIPS: CÓDIGOS DE OPERACIÓN

Instruction	Operands	Function
ADDV.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.
SUBV.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.
MULV.D	V1,V2,V3	Multiply elements of 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.
DIVV.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 × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × 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 × R1, 2 × R1,...,63 × R1 into V1.
S--V.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 the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to the vector-length register. Move the contents of the vector-length register to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

Figure F.3 The VMIPS vector instructions. Only the double-precision FP operations are shown. In addition to the vector registers, there are two special registers, VLR (discussed in Section F.3) and VM (discussed in Section F.4). These special registers are assumed to live in the MIPS coprocessor 1 space along with the FPU registers. The operations with stride are explained in Section F.3, and the uses of the index creation and indexed load-store operations are explained in Section F.4.

- [HePa12] J. HENNESSY and D. PATTERSON, Computer Architecture: a quantitative approach. 5th Edition, Morgan Kaufmann, 2012.
  - Chapter 4 and Appendix G: Vector Processors in More Depth

❒ DLXV completo: diagrama de flujo de la ALMa



6. COMPILACIÓN

6.1 Introducción.
Fases en el *back-end* del compilador: extracción automática de paralelismo vectorial

6.2 Transformaciones previas que simplifican el análisis de dependencias

6.3 Análisis y grafo de dependencias. Tests aproximados

6.4 Optimizaciones independientes de la arquitectura: renombrar, expansión escalar, copia de vectores

6.5 Vectorización
- Procedimiento básico
- Vectorización parcial vs. total: distribución e intercambio de bucles
- Reducción

6.1 FASES EN EL *BACK-END* DEL COMPILADOR

Front-end: análisis léxico y sintáctico

Programa escalar representado en lenguaje intermedio

⇓

Transformaciones que simplifican el análisis de las dependencias	• Propagación y evaluación de expres. constantes • Extracción de invariantes • Normalización de bucles
⇓	
Grafo y análisis de dependencias	• Construcción del grafo • Tipos y distancia de las dependencias • Análisis de dependencias. Test MCD
⇓	
Optimizaciones independientes de la arquitectura vect/par	• Eliminar antidependencias y dep. de salida - renombrar - expansión escalar - copia de vectores
⇓	
Vectorización	• Vectorización total y parcial - distribución e intercambio de bucles • Reducción, punteros en C
⇓	
Generación de código. Optimización	• Asignación de registros vectoriales • Selección de instrucciones vectoriales • Seccionado
⇓	
Programa vectorial optimizado	EJEC(progr. escalar) = EJEC (prog. vectorial) + f estática vect ↑↑

6.2 TRANSFORMACIONES QUE SIMPLIFICAN EL ANÁLISIS DE LAS DEPENDENCIAS

- Propagación y evaluación de expresiones constantes + Extracción de invariantes

```
DO i = 1,N
  PI = 3.14
  PD = 2*PI
  A(i) = PD * R(i) **2
ENDDO
```

⇒

```
PI = 3.14
PD = 6.28
DO i = 1,N
  A(i)= 6.28 * R(i) **2
ENDDO
```

Otro ejemplo de invariantes

```
DO i = 1,N
  DO j = 1,M
    A(i,j) = B(j) * C(i)
  ENNDDO
ENDDO
```

⇒

```
?
```

- Normalización de bucles:
  - normalizar el paso de la variable de control
  - eliminar variables de inducción, o sea, indexar vectores únicamente con las variables de control del bucle

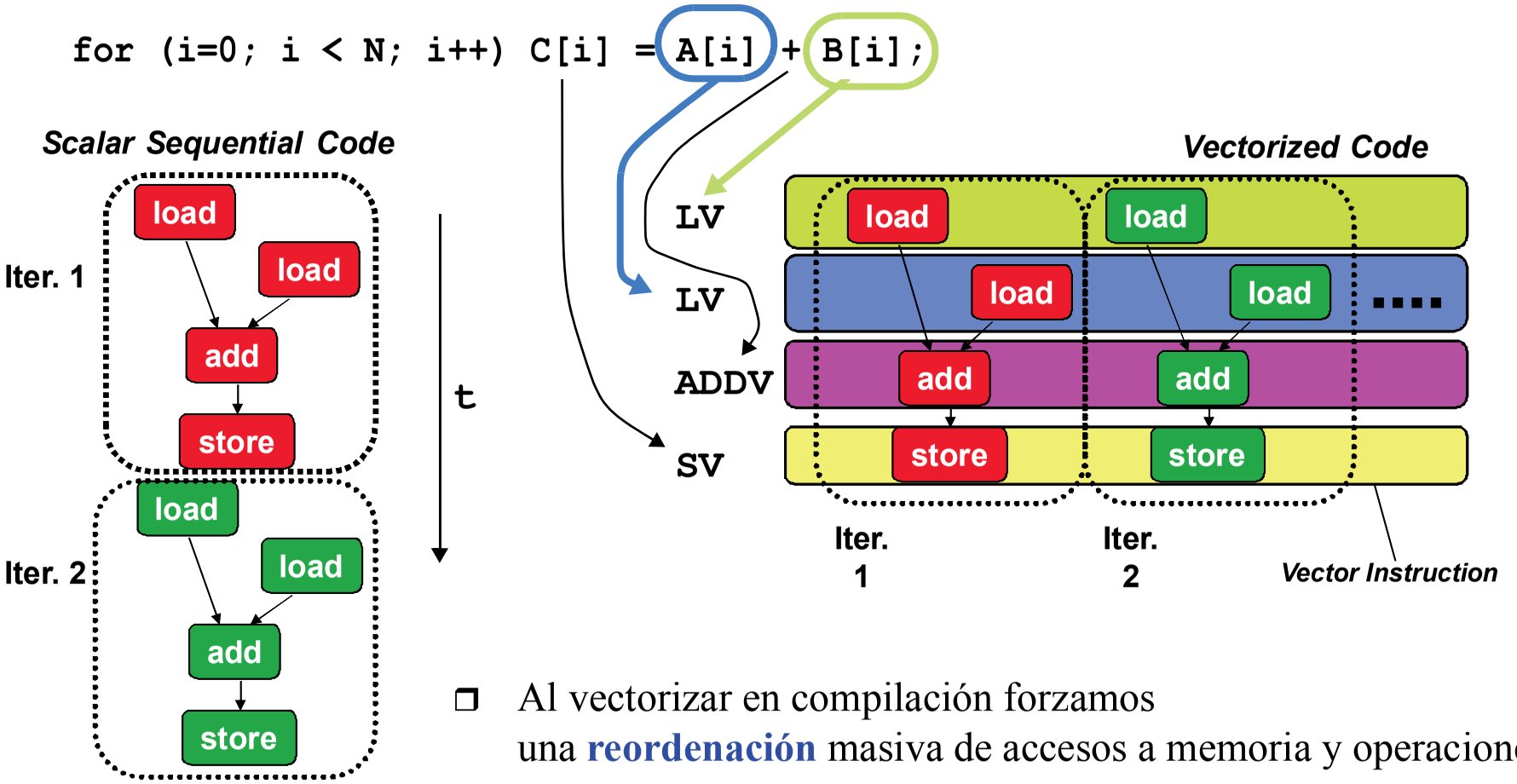
```
DO i = 1,100
  aux = i
  DO j = 1,300,3
    aux = aux + 2
    A(j) = A(j-1) + B(aux)
  ENNDDO
ENDDO
```

⇒

```
DO i = 1,100
  ~~aux = i~~
  DO nj = 1,100
    ~~aux = aux + 2~~
    A(3*nj-2) = A(3*nj-3) + B(i+2*nj)
  ENNDDO
ENDDO
```

3*nj-2, 3*nj-3, 2*nj+i ⇒ combinación lineal de las variables de control de bucle

6.3 ANÁLISIS Y GRAFO DE DEPENDENCIAS



- Al vectorizar en compilación forzamos una **reordenación** masiva de accesos a memoria y operaciones

- Ejemplo de reordenación: C[2] $<_p$ B[3], pero B[3] $<_m$ C[2] !

- Es necesario analizar las dependencias con cuidado:
  que pasa si el store rojo escribe donde lee el load verde ? ⇒ `C[i] = C[i-1] + ...`

DEPENDENCIAS EN MEMORIA ENTRE SENTENCIAS[1]

```
R: ------
   ...
S: ------
```

S y **R** son sentencias de asignación

S depende de **R** si:

- **R** antes que **S** en orden de programa ($R <_p S$)
- **R** y **S** referencian a la misma posición de memoria
- una referencia al menos es una escritura

- Las dependencias establecen relaciones de orden parcial que cualquier ejecución "legal" debe respetar.

Tres tipos		
R escribe y **S** lee Dependencia verdadera *flow dependence, RAW hazard*	**R** δ **S**	**R** → **S**
R lee y **S** escribe Antidependencia *antidependence, WAR hazard*	**R** δ^- **S**	**R** ⊣→ **S**
R y **S** escriben Dependencia de salida *output dependence, WAW hazard*	**R** δ^o **S**	**R** ⊖→ **S**
Si **R** y **S** están en un bucle, $\mathbf{R_i}$ y $\mathbf{S_i}$ indican la ejecución de la iteración **i**-ésima		

1) La investigación en vectorización automática se inicia en los años 70, y el siguiente trabajo de la Universidad de Illinois se considera uno de los mas importantes para su herramienta básica, el análisis de dependencias.
Kuck, Kuhn, Padua, Leasure, and Wolfe. "Dependence graphs and compiler optimizations". In Proc. of the 8th ACM Symposium on Principles of Programming Languages, pp. 207-218. 1981.

- Ejemplo 1 -

```
i= 1,64
  S1: A(i)= B(i)*C
  S2: B(i)= B(i-1) + A(i)
```

por inspección: $S1_i\ \delta\ S2_i$ posición A(i)

$S1_i\ \delta^{-}\ S2_i$ posición B(i)

Pero ...

$S1_1$	A(1)= B(1)*C
$S2_1$	B(1)= B(0) + A(1)
$S1_2$	A(2)= B(2)*C
$S2_2$	B(2)= B(1) + A(2)
	⋮

¡ Dependencias entre iteraciones !

$S2_i\ \delta\ S2_{i+1}$ en pos B(i)

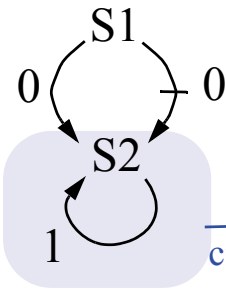
loop carried dependence, LCD
o sea, dependencias generadas por el bucle

λ = distancia de la dependencia: número de iteraciones que separa el uso de las mismas posiciones de memoria

$R_i\ \delta\ S_j \rightarrow \lambda = j-i \geq 0$

$\mathbf{R} \xrightarrow{\lambda} \mathbf{S}$

- Grafo de dependencias: grafo dirigido, anotado con distancias:



- Vectorizable en parte

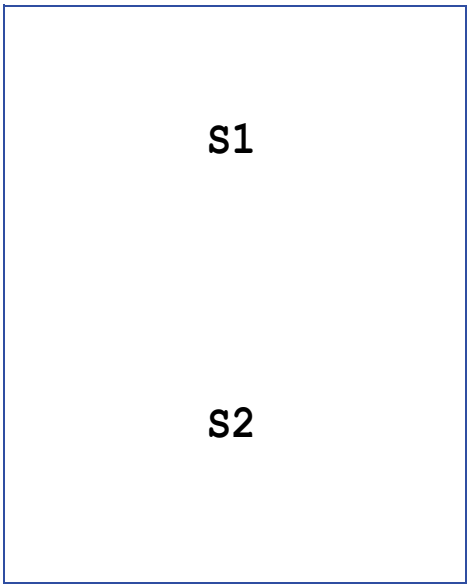
```
A(1:64)= B(1:64)*C
i= 1,64
  S2: B(i)= B(i-1) + A(i)
```

ciclo unitario de dependencia

- Ejemplo 2 -

```
i= 1,64
  S1: A(i+1)= A(i) + B(i)
  S2: B(i+1)= B(i) - A(i)
```

ejecución
$S1_1$ $S2_1$
$S1_2$ $S2_2$
⋮



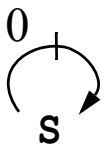
- O sea: NO es vectorizable !

```
A(2:65)= A(1:64) + B(1:64)
B(2:65)= B(1:64) - A(1:64)
```

```
LV     B(1:64)
LV     A(1:64)
ADDV   +
SV     A(2:65)=
A(1:64)
B(1:64)
-
B(2:65)=
```

- Ejemplo 3: AXPY -

```
i= 1,64
  S: Y(i)= a*X(i) + Y(i)
```



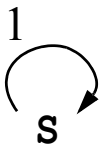
S

... o simplemente

Autociclos de antidep. a distancia 0, pueden obviarse: *Vectorizable*

- Ejemplo 4 -

```
i= 1,64
  S: fac(i)= fac(i-1) * i
```



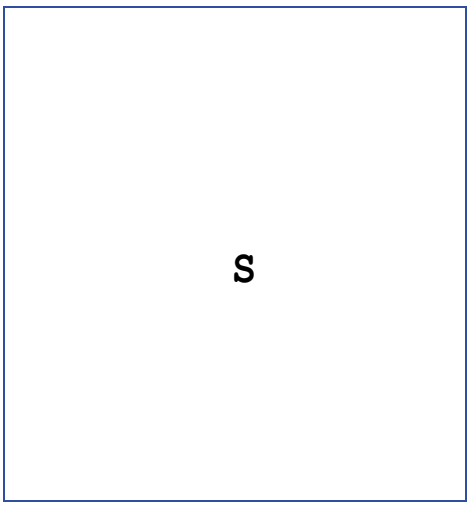
Autociclos dependencias verdaderas distancia > 0, *No Vectorizable*

- Ejemplo 5 -

```
i= 1,64
  S: A(i)= A(i+3) + A(i)
```

secuencia de cálculos:

S_1
S_2
S_3
S_4
S_5



Vectorizable: `A(1:64)= A(4:67) + A(1:64)`[a]

a. Esta notación corresponde a un lenguaje real y en uso: FORTRAN 90, el sucesor directo de FORTRAN 77

- Vectorizable: `A(1:64)= A(4:67) + A(1:64)` [1]

- Ejemplo 6: escalado -

```
i= 1,64
  S: A(i)= k*A(i
```

S

VECT, PAR

- Ejemplo 7: progresión geométrica-

```
i= 1,64
  S: A(i+1)= k*A(i)
```

S

no VECT, no PAR

- Ejemplo 8: escalado con desplazamiento a laizquierda -

```
i= 1,64
  S: A(i)= k*A(i+1)
```

S

si VECT, no PAR

- Ejemplo 9 -

```
i= 1,64
  S1: A(i)= B(i+2)
  S2: B(i)= A(i-1)
```

S1

S2

si VECT, no PAR

- Escribir el código vectorial

1) Esta notación corresponde a un lenguaje real y en uso: FORTRAN 90, el sucesor directo de FORTRAN 77

- Ejemplo 10 -

```
i= 1,64                          S1
  S1: A(i+2)= B(i)
  S2: B(i+2)= A(i+1)             S2
```

no VECT, no PAR

ANÁLISIS DE DEPENDENCIAS

Caso bastante general:

- conjunto de bucles anidados normalizados
- código en el bucle mas interno y sin sentencias condicionales
- espacio de iteraciones conocido en compilación

```
real*8 A(dim1, dim2, ...dimm)

i1= 1,max1
   ...
   ik-1= 1,maxk-1
...   ik=  1,maxk
      S1: A(f1(i1, ...ik),f2(i1, ...ik),...fm(i1, ...ik)) = ...
      S2: ... = A(g1(i1, ...ik),g2(i1, ...ik),...gm(i1, ...ik))
```

En cada iteración del código `S1 S2`, las variables de control forman una k-tupla $(i_1, \ldots i_k)$ diferente. La unión de todas ellas se llama *Espacio de Iteraciones* (EspIt). EspIt es el producto cartesiano de los k conjuntos de números naturales que recorre cada variable de control.

Puede establecerse un *orden de programa total* entre k-tuplas ($<_p$)

Las funciones **f** y **g** son *funciones de indexación*. Calculan un índice a partir de los valores $(i_1, \ldots i_k)$ de las variables de control

PREGUNTAS ...

$\exists$ P, Q $\in$ EspIt $\mid$ escr en $S1_P$ δ lect en $S2_Q$ para P $\leq_p$ Q; ?[a]

$\exists$ T, Z $\in$ EspIt $\mid$ lect en $S2_T$ δ^- escr en $S1_Z$ para T $<_p$ Z; ?[b]

a. P.e. para P = (i_1, i_2, i_3) = (2, 3, 4), en S1 se escribe el elemento A(1,2) y para Q = (4, 1, 1), en S2 se lee *el mismo* elemento.

b. P.e. para T = (3, 3, 3), en S2 se lee el elemento A(9,1) y para Z = (4, 1, 1), en S1 se escribe *el mismo* elemento.

Caso sencillo:

- K = 1 variable de control
- m = 1 dimensión

```
i= 1, max
S1: A(f(i))= ...
S2: ...    = A(g(i))...
```

$\exists$ dependencia prod/cons a distancia λ sii[a]

$S1_p$ — λ → $S2_q$

$\exists$ p, q $\in$ EspIt $\mid$ f(p) = g(q) para p $\leq$ q

λ = q - p

a. P.e. para p = 6 en S1 se escribe el elemento A(19) y para q = 6, en S2 se lee *el mismo* elemento.

$\exists$ antidependencia a distancia λ sii[a]

$S1_z$ — λ → $S2_t$

$\exists$ t, z $\in$ EspIt $\mid$ g(t) = f(z) para t < z

λ = z - t

a. P.e. para t = 10, en S2 se lee el elemento A(31) y para z = 12, en S1 se escribe *el mismo* elemento.

- Ejemplo 11 - f y g son lineales

```
i= 1,100
  S1: A(2i+7)= B(i)-3
  S2: C(i)= A(3i+1)
```

S1

S2

VECT ?, PAR ?

secuencia de cálculos:

i	escr +2 / lect +3
1	9 / 4
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Eqn. dependencias prod/cons: S1 δ S2

$\exists$ p, q $\in$ [1..100] | f(p) = g(q) para p $\leq$ q

2p+7 = 3q+1 (λ = q - p)

Despejamos q y enumeramos p; **q = 2p/3+2**

p	1	2	**3**	4	5	**6**	7	8	9	10 ...
q	no	no	**4**	no	no	**6**	no	no	8	no
$\lambda\geq 0$			**1**			**0**			-1	

Eqn. antidependencias: S2 δ^{-} S1

$\exists$ t, z $\in$ [1..100] | g(t) = f(z) para t < z

3t+1 = 2z+7 (λ = z - t)

Despejamos z y enumeramos t; **z = 3t/2-3**

t	1	2	3	4	5	**6**	7	**8**	9	**10** ... **68**
z	no	0	no	3	no	**6**	no	**9**	no	**12** **99**
$\lambda>0$		-2		-1		0		**1**		**2** ... **31**

- Este tipo de ecuación se llama ecuación Diofántica

- Ejemplo 12 - f y g son lineales

```
i= 1,100                    S1
  S1: C(i)= A(3i+1)
  S2: A(2i+7)= B(i)-3       S2
```

VECT ?, PAR ?

i	lect +3 / escr +2
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Eqn. dependencias prod/cons: S2 δ S1

Despejamos q y enumeramos p;

p	1	2	3	4	5	6	7	8	9	10 ...
q										
λ										

Eqn. antidependencias: S1 δ^{-} S2

Despejamos z y enumeramos t;

t	1	2	3	4	5	6	7	8	9	10 ...
z										
λ										

TESTS APROXIMADOS

Resolver las ecuaciones directamente es costoso en tiempo

Los límites inferior y superior a veces no son constantes

- Tests aproximados, fáciles de calcular:
  - ✓ condición necesaria para que existan ciclos de dependencias
    - se cumple ? → no vectorizamos
    - muy conservador ...
  - ✓ condición suficiente para la independencia
    - se cumple ? → vectorizamos
    - muy exigente
- Test MCD para funciones de indexación lineales (cond. necesaria)

$$f \rightarrow \mathbf{a}\cdot i+\mathbf{b} \qquad g \rightarrow \mathbf{c}\cdot i+\mathbf{d}$$

Asumimos posible ciclo de dependencias si **mcd(a, c)** divide **(d-b)**

- ejemplos 11 y 12 -

$a = 3, b = 1$
$c = 2, d = 7$ } mcd(3, 2) = 1 divide a (6) → SI

... pues asumimos dependencias cíclicas
y no vectorizamos

Ejercicios:

- ✓ demostración test MCD
- ✓ buscar un ejemplo vectorizable que no pase el test MCD

EJERCICIO E5

En bucles con un solo nivel de anidación pueden aparecer varios tipos de dependencias.

```
i ∈ rango N
  R:    var(r)
  S:    var(s)
```

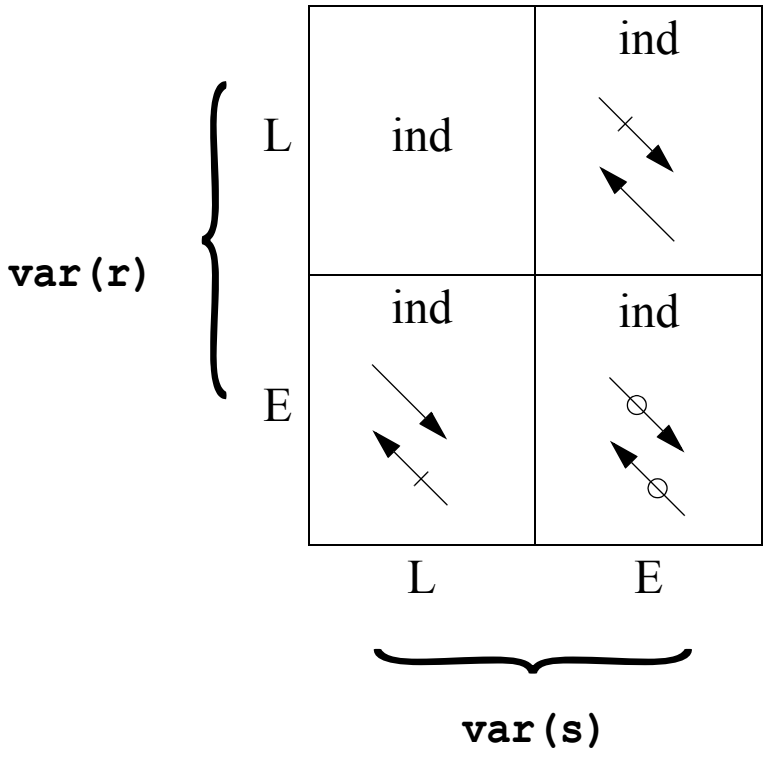
R

S

Los índices **r** y **s**
son funciones lineales de i: r(*i*), s(*i*)

VECT ?, PAR ?

En la tabla siguiente
se resumen todas las posibilidades.



Por ejemplo:

```
i ∈ rango N
  R: var(r) =  ...
  S: var(s) =  ...
```

r y **s** son funciones lineales de *i*
$\lambda \in$ N es la distancia de la dependencia

Podemos formular las tres ecuaciones de dependencias:

- ✓ **R** δ^o **S** sii $\exists\lambda |\ r(i) = s(i+\lambda);\ \lambda \geq 0;\ i, i+\lambda \in rango$
- ✓ **S** δ^o **R** sii $\exists\lambda |\ s(i) = r(i+\lambda);\ \lambda \geq 1;\ i, i+\lambda \in rango$
- ✓ **R** ind **S** caso contrario
  ejemplos de este caso:

```
DO i= 2, 100
R:   A(2i-1)=
S:   A(6i+4)=
ENDDO
```

```
DO i= 1, 100
R:   A(2i)=
S:   A(2i+1)=
ENDDO
```

Ejercicio. Se pide lo siguiente:

Repetir el análisis anterior para el caso
"**var(r)** es Lectura y **var(s)** es Escritura"

- ✓ las tres ecuaciones de dependencias
- ✓ un ejemplo de cada caso, con funciones de indexación diferentes de las empleadas en los ejemplos anteriores.

6.4 Optimizaciones no ligadas a la arquitectura

⇓

Grafo y análisis de dependencias

⇓

Optimizaciones independientes de la arquitectura vect/par

- **Eliminar antidependencias y dep. de salida**
  - renombrar
  - expansión escalar
  - copia de vectores

⇓

Vectorización

⇓

Renombrar

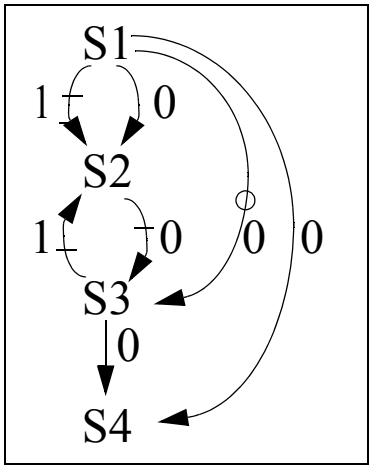
Problema: reutilizar vectores o escalares para ahorrar memoria o variables

Transformación:

- ✓ nombre diferente a cada asignación (parte izquierda)
- ✓ propagar nombres nuevos a las sentencias posteriores (partes derechas)

```
DO i= 1, N
S1:  U(i)= A(i) + 4
S2:  A(i-1)= U(i)*2 + C(i)
S3:  U(i)= A(i) - 4
S4:  B(i)= U(i)*3 + D(i)
ENDDO
```

⇒

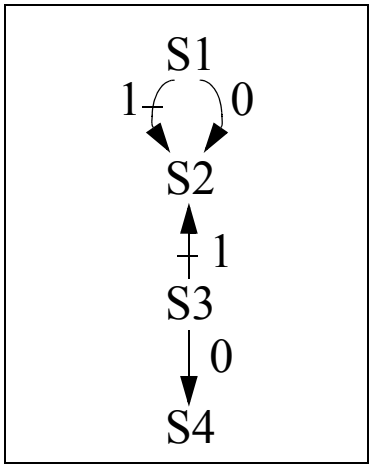


- Ejemplo 13 - no vect

Transformación:
- U(i) → T(i) en S1
- propagar T(i) en S2

```
DO i= 1, N
S1:  T(i)= A(i) + 4
S2:  A(i-1)= T(i)*2 + C(i)
S3:  U(i)= A(i) - 4
S4:  B(i)= U(i)*3 + D(i)
ENDDO
```

⇒



casi vect

- ✓ ¿ Aumentan las necesidades de almacenamiento ?

Expansión Escalar

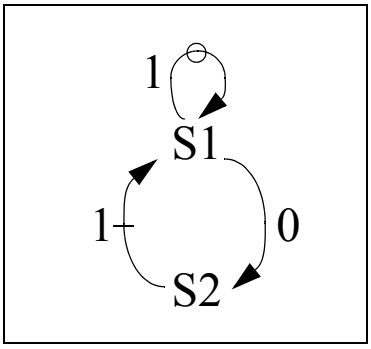
Problema: variables escalares que se utilizan en iteraciones sucesivas para almacenar valores diferentes

Transformación:

- ✓ promocionar el escalar a vector (parte izquierda)
- ✓ propagar nombres nuevos a las sentencias posteriores (partes derechas)

```
DO i= 1, N
S1:  U= A(i) + B(i)
S2:  D(i)= U*2 + C(i)
ENDDO
```

⇒

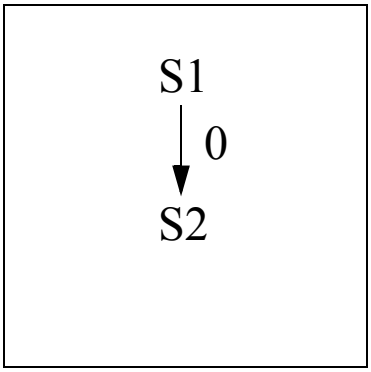


- Ejemplo 14 - no vect

Transformación:
- U → U(i) en S1
- propagar U(i) en S2

```
DO i = 1, N
S1:  U(i)= A(i) + B(i)
S2:  C(i)= U(i)*2 + C(i)
ENDDO
U= A(N) + B(N)
```

⇒

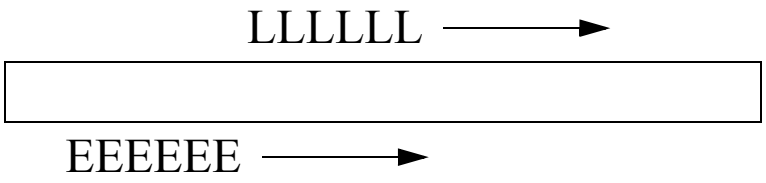


vect

- ✓ ¿ Aumentan las necesidades de almacenamiento ?

Copia de vectores[1]

Problema: pre-uso y post-definición de los elementos de un vector

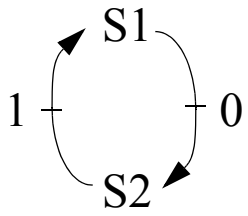


Transformación:

- ✓ crear una copia del vector de solo-lectura
- ✓ propagar nombre copia a las sentencias posteriores (partes derechas)

```
DO i= 1, N
S1:  A(i)= B(i) * 2
S2:  B(i)= A(i+1) + 3
ENDDO
```

⇒

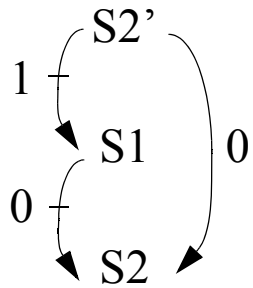


- Ejemplo 15 - no vect

Transformación:
- A → tmpA en S2'

```
DO i= 1, N
S2': tmpA(i)= A(i+1)
S1:  A(i)= B(i) * 2
S2:  B(i)= tmpA(i) + 3
ENDDO
```

⇒



vect

- ✓ ¿ Aumentan las necesidades de almacenamiento ?

1) También llamado *node splitting*, particionado de nodos

6.5 VECTORIZACIÓN

Optimizaciones independientes de la arquitectura vect/par

⇓

Vectorización

- Vectorización total y parcial
  - distribución e intercambio de bucles
- Reducción
- Punteros en C

⇓

Generación de código. Optimización

Vectorización total

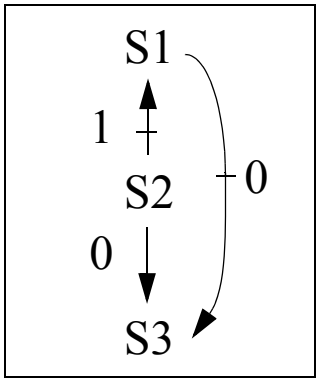
No existen ciclos de dependencias

Transformación:

- ✓ cambiar el orden de las sentencias para que todos los arcos vayan hacia abajo[1]
- ✓ generar código vectorial sentencia a sentencia, empezando por cualquiera que sea independiente

```
DO i= 1, N
S1:  A(i)= D(i) - C(i)
S2:  B(i)= A(i+1) + 1
S3:  C(i)= B(i) * 2
ENDDO
```

⇒

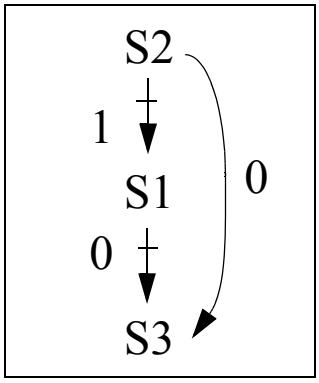


- Ejemplo 16 -

- algoritmo de "flotación": peso proporcional al nº de arcos entrantes

```
S2(1:N) es B(1:N)= A(2:N+1)+1
S1(1:N) es ...
S3(1:N)
```

⇒



- ✓ **con varios niveles de anidación**, una sentencia puede vectorizarse con respecto a un bucle si no está en un ciclo de dependencias generadas por ese bucle → veremos ejemplos

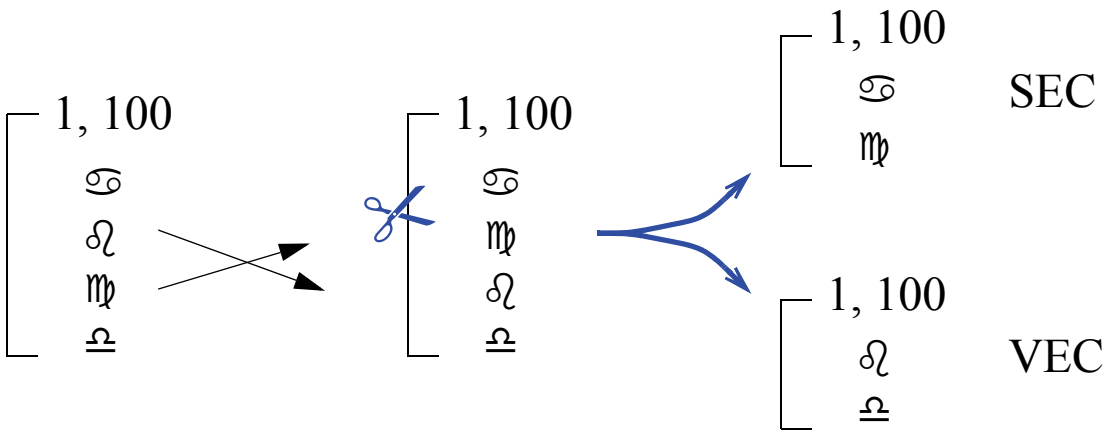
1) Comrobad que este nuevo orden de jecución secuencial es correcto

Vectorización parcial, distribución de bucles[1]

Problema 1 de 2: existen ciclos de dependencias

Transformación: extraer las sentencias sin ciclos de dependencias

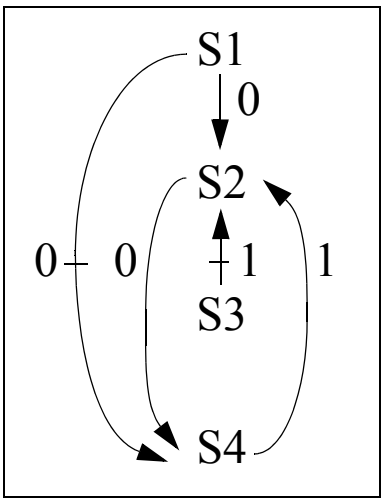
- Particionar el grafo en subgrafos nodo-disjuntos relacionados entre sí de forma acíclica (π bloques)
- reordenar con las dependencias hacia abajo
- Distribuir/cortar en varios bucles



1) También llamado *loop fision*, corte de bucles

```
DO i = 1, N
S1:  A(i) = B(i) + 4
S2:  C(i) = A(i) + B(i-1)
S3:  E(i) = C(i+1)
S4:  B(i) = C(i) + 2
ENDDO
```

⇒

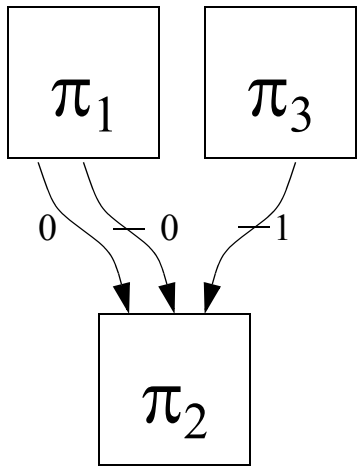


- Ejemplo 17 -

$\pi_1 = S1$

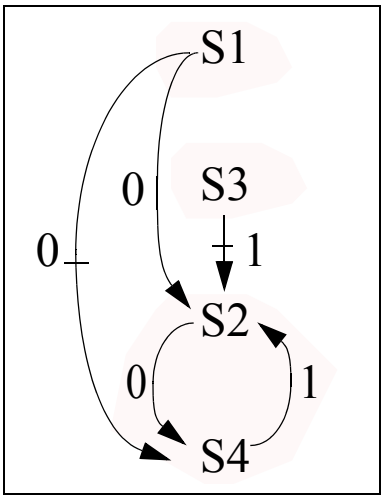
$\pi_2 = S2, S4$

$\pi_3 = S3$




```
S1(1:N)
S3(1:N)
DO i = 1, N
S2:  C(i) = A(i) + B(i-1)
S4:B(i) = C(i) + 2
ENDDO
```

⇒

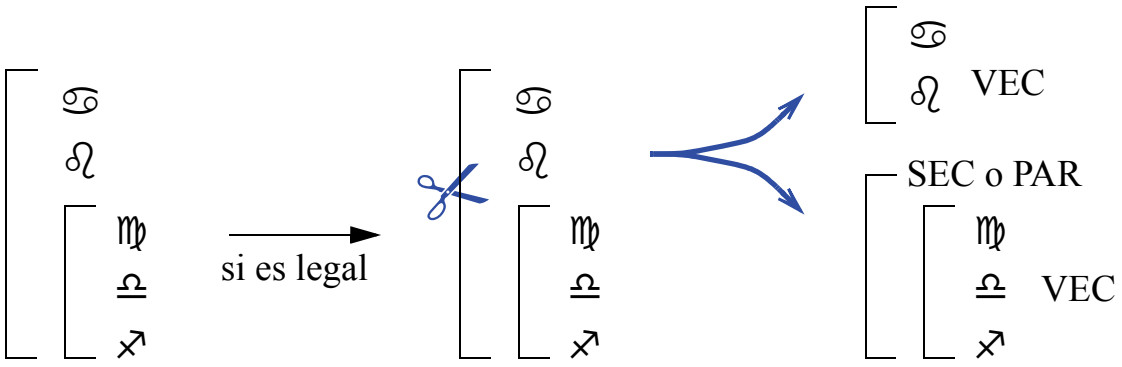


Vectorización parcial, distribución de bucles[1]

Problema 2 de 2: dos niveles de anidación de bucles no perfectos

Transformación:

- si es legal, distribuir el bucle mas externo
- **vectorizar el bucle mas interno**, si es posible, aplicando lo anterior
- **paralelizar el bucle mas externo**, si es posible



no tocamos los espacios de iteración

1) También llamado *loop fision*, corte de bucles

```
   do j= 1, maxcols
S1:    a(j)= 0.0
       do i= 1, maxrows
S2:       b(i,j)= 0.0
       enddo
   enddo
```

⇒

S1

S2

- Ejemplo 18 -

Transformación:
- distribuir el bucle exterior

```
   do j= 1, maxcols
S1:    a(j)= 0.0
   enddo
   do j= 1, maxcols
       do i= 1, maxrows
S2:       b(i,j)= 0.0
       enddo
   enddo
```

⇒

S1(1:maxcols)

PAR(1:maxcols)
S2(1:maxrows)

Vectorización parcial/total: intercambio de bucles

Problema:

- ✓ en bucles anidados el recorrido del espacio de iteraciones impide:

1. una fracción vectorial elevada
2. un acceso eficiente a memoria
3. vectorización

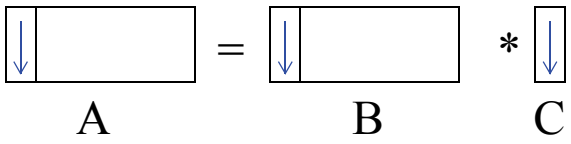
Transformación:

- ✓ intercambiar el orden de anidación de los bucles, si es legal

---

```
do j= 1, 64
  do i=1, 8
    A(i,j)= B(i,j)*C(i)
  enddo
enddo
```

⇒

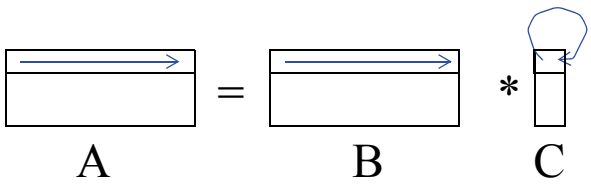


- Ejemplo 19 -

1. Intercambiar para aumentar la fracción vectorial

```
do i= 1, 8
  A(i,1:64)= B(i,1:64)*C(i)
enddo
```

⇒



```
integer maxlen
parameter (maxlen = 64)
real A(maxlen, maxlen)

do i= 1, maxlen              ! intercambiar para mejorar el
  do j= 1, maxlen            ! acceso a la memoria multibanco[a]
    A(i,j)= 0.0
  enddo
enddo

do i= 1, 8                   ! no intercambiamos para
  do j= 1, maxlen            ! preservar vectores largos
    A(i,j)= float(i+j)
  enddo
enddo
```

- Ejemplo 20 -

2. Intercambiar para mejorar el acceso a memoria

a. En FORTRAN las matrices se almacenan por columnas: $a_{\text{fila columna}} \rightarrow a_{11}, a_{21}, a_{31}, \ldots a_{12}, a_{22}, a_{32}, \ldots$
En C, se almacenan por filas.

```
real A(64,64)
do j= 2, 63
  do i= 1, 64
S: A(i,j)= (A(i,j-1) + 2*A(i,j) + A(i,j+1))*0.25
  enddo
enddo
```

- Ejemplo 21 -

3. No es necesario intercambiar para poder vectorizar

En este ejemplo el bucle `j` genera ciclos de dependencias, pero el bucle `i` no, y por tanto la sentencia es vectorizable en `i`

```
do j= 2, 63
  A(1:64,j)= (A(1:64,j-1) + 2*A(1:64,j) + A(1:64,j+1))*0.25
enddo
```

¿ Como lo hemos sabido ?

```
do j= 1, 64
  do i= 1, 64
S: A(i+1,j)= A(i,j) * B(i)
  enddo
enddo
```

El bucle i tiene
una recurrencia a distancia 1
que impide vectorizar

- Ejemplo 22 -

3. Intercambiar para poder vectorizar

En este ejemplo

1. el intercambio es legal, y
2. posibilita la vectorización del bucle interno

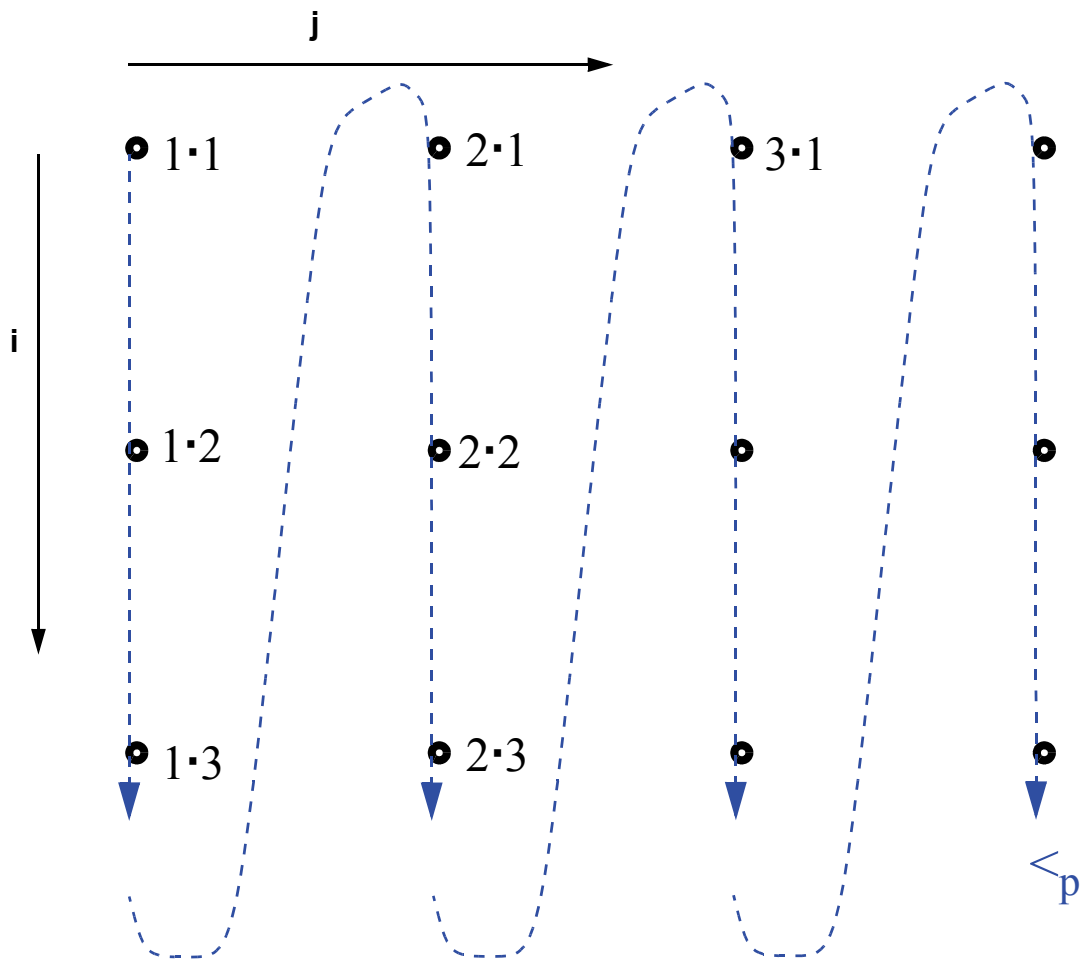
```
do i= 1, 64
  A(i+1,1:64)= A(i,1:64) * B(i)
enddo
```

¿ Como lo hemos sabido ?

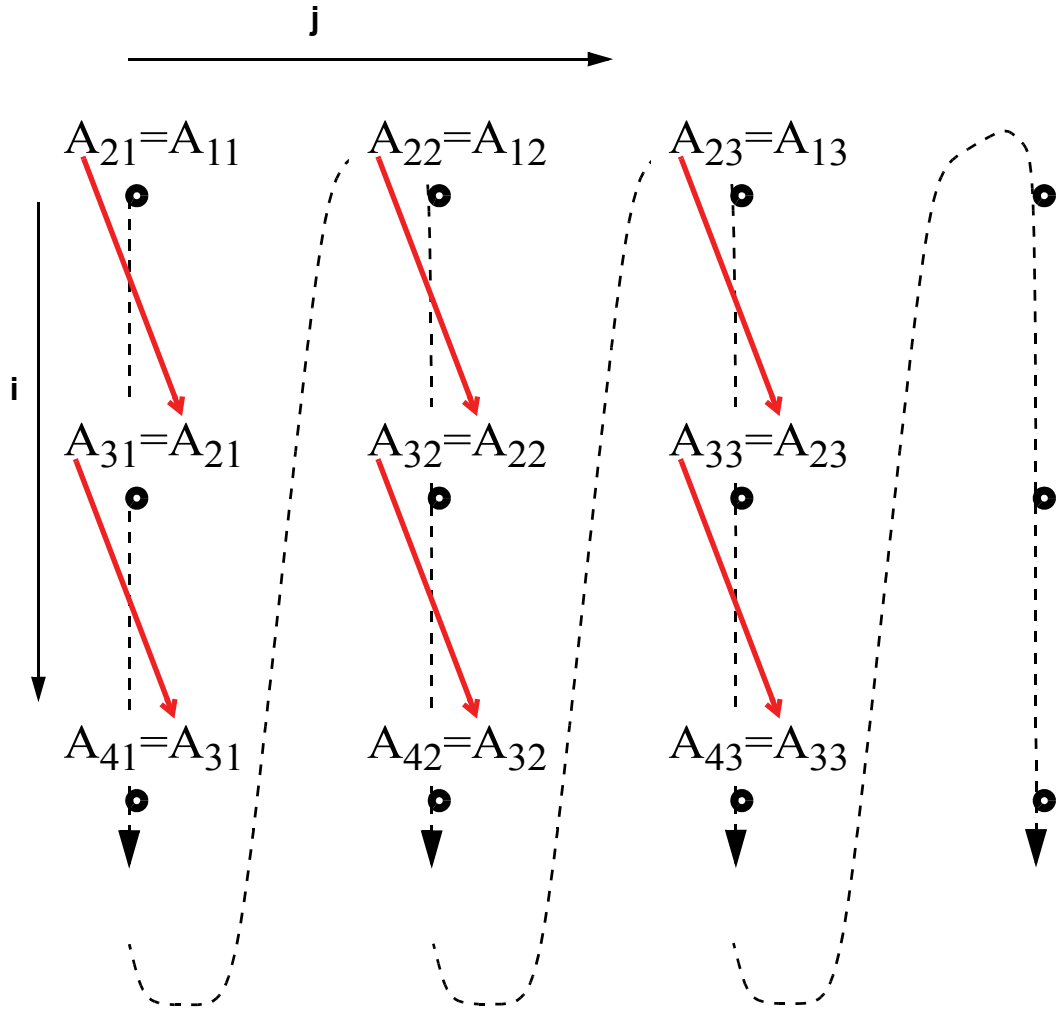
DIAGRAMA DE DEPENDENCIAS CON ANIDACIONES

```
do j= 1, 64
  do i= 1, 64
S: A(i+1,j)= A(i,j) * B(i)
  enddo
enddo
```

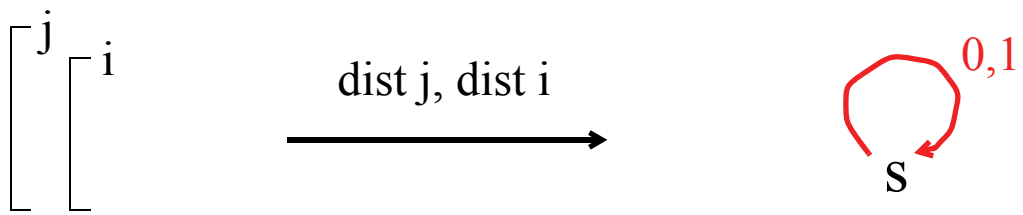
EspIter → { j=1:64 × i =1:64} ={1·1, 1·2, 1·3, ... 2·1, 2·2, 2·3, ... 3·1, ...}



- Al recorrer EspIt en el orden de programa, camino a trazos, los accesos a memoria determinan las dependencias. Cualquier otro orden que las respete, *es legal*

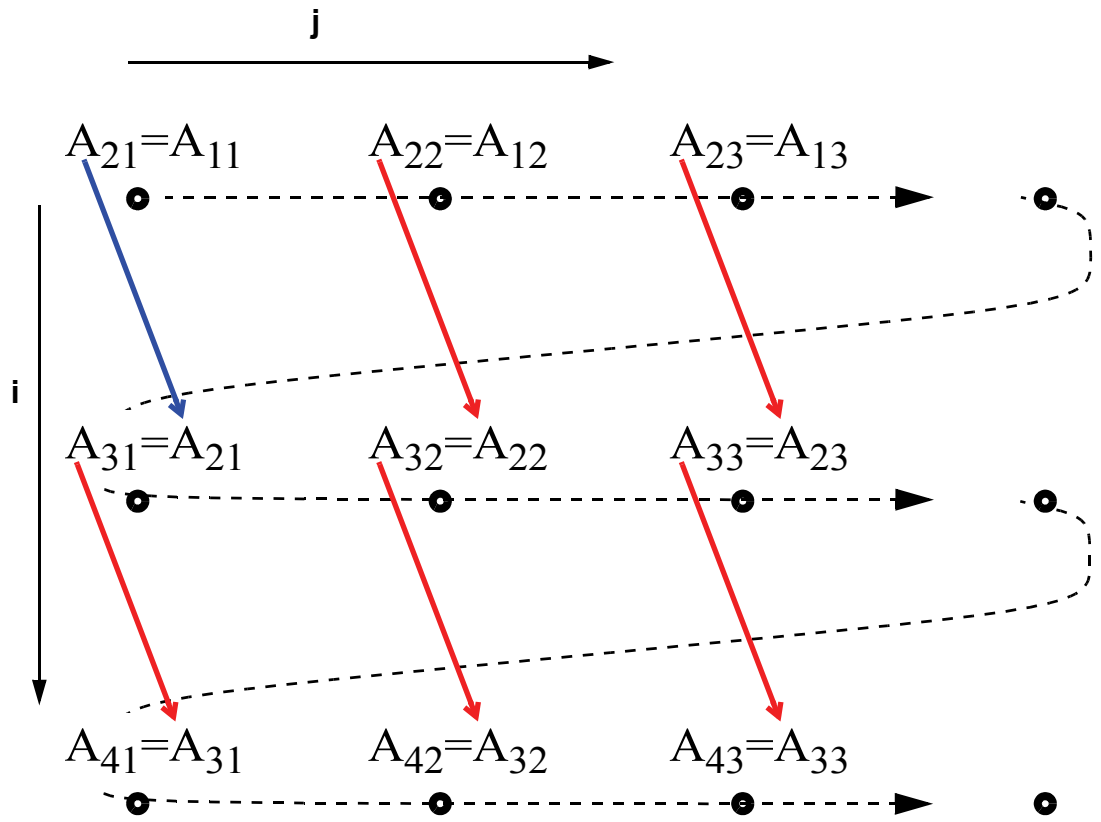


¿ Como representar las dependencias ?



Es una recurrencia en (i), y por tanto no es vectorizable !!

- Intercambiar los bucles es recorrer EspIt en otro orden:



$<_{m'}$: $E(A_{21})$, $E(A_{22})$, $E(A_{23})$, ... $L(A_{21})$, $E(A_{31})$, $L(A_{22})$, $E(A_{32})$...

- El nuevo recorrido, respeta las dependencias ?
  - ✓ SI → el intercambio es legal
  - ✓ NO → no es legal

- Ahora el bucle interno (j) ejecuta los cálculos en otro orden.
  En este nuevo orden NO hay dependencias entre cálculos consecutivos
  Es posible vectorizar en (j) !!

```
do i= 1, 64
  A(i+1,1:64)= A(i,1:64) * B(i)
enddo
```

RESUMEN DEL PROCEDIMIENTO MANUAL

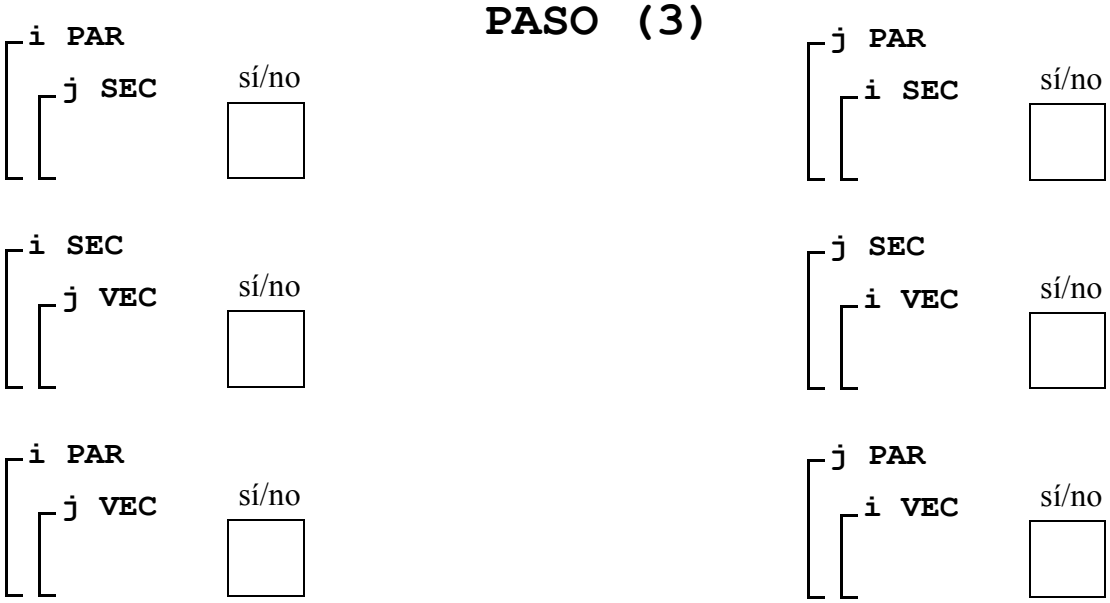
1) desplegar EspIt,
   anotando referencias a memoria y dependencias
2) grafo de dependencias
3) ¿ se puede vectorizar y paralelizar ?
4) ¿es legal el intercambio?
   en caso afirmativo, repetir (3)
5) escoger la opción mejor

```
do i= 2, 64
   do j= 2, 32
S1:    A(i,j)= B(i,j)
S2:    B(i-1, j-1)= A(i-1,j)
   enddo
enddo
```

- Ejemplo 23 -

Estudiar las posibilidades de codificación y valorar su rendimiento

PASO (3)



```
void calc_array (int** a)
{
  for (i=1; i<N; ++i)
    for (j=1; j<N; ++j)
      a[i][j] = a[i-1][j] + a[i][j-1]
)
```

- Ejemplo 24 -

Leer el texto y reflexionar

"This code shows a nested loop operating on a 2D array with cross-iteration dependencies over both loops, making it appear serial. From the dependence graph it can be shown that iterations can be grouped into independent sets, allowing parallel execution if ***loop skewing*** and *interchange* are used. Techniques relying on dependence testing would overlook this parallelism. Furthermore, the 2D array in (a) is represented as an array of pointers to arrays, thwarting a parallelizing compiler's attempt to statically analyze this section of code."

Garcia, S.; Donghwan Jeon; Louie, C.; Taylor, M.B.; , "The Kremlin Oracle for Sequential Code Parallelization," IEEE Micro, vol.32, no.4, pp. 42-53, July-Aug. 2012
IEEE Micro Special Issue on Parallelization of Sequential Code

---

Reducción

En este contexto *reducir* es una operación matemática que quita una dimensión a una estructura de datos:

matriz → vector, **vector → escalar**, ...

Es una operación frecuente en cálculo científico.
Los repertorios vectoriales tienen instrucciones para operadores de reducción *asociativos*:

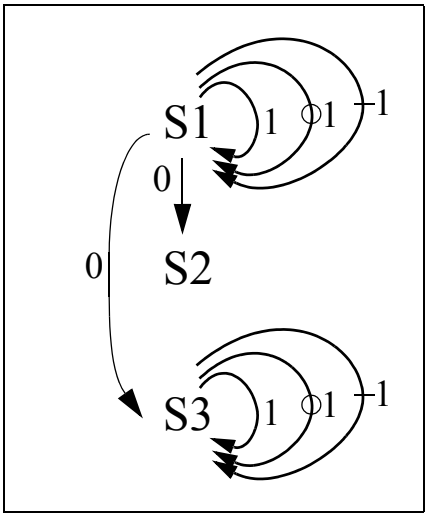
- ✓ máximo, mínimo, OR, AND, suma, producto
- ✓ ojo con el problema asociatividad / precisión finita

Transformación:

- ✓ detectar los autociclos de dependencias debidos al reuso de la variable escalar de reducción
- ✓ sustituir el bucle de reducción por la instrucción apropiada

```
s=0
p=1
DO i= 1, 128
S1:  A(i)= B(i) + C(i)
S2:  s = s + A(i)
S3:  p = p * A(i)
ENDDO
```

⇒



- Ejemplo 25 -

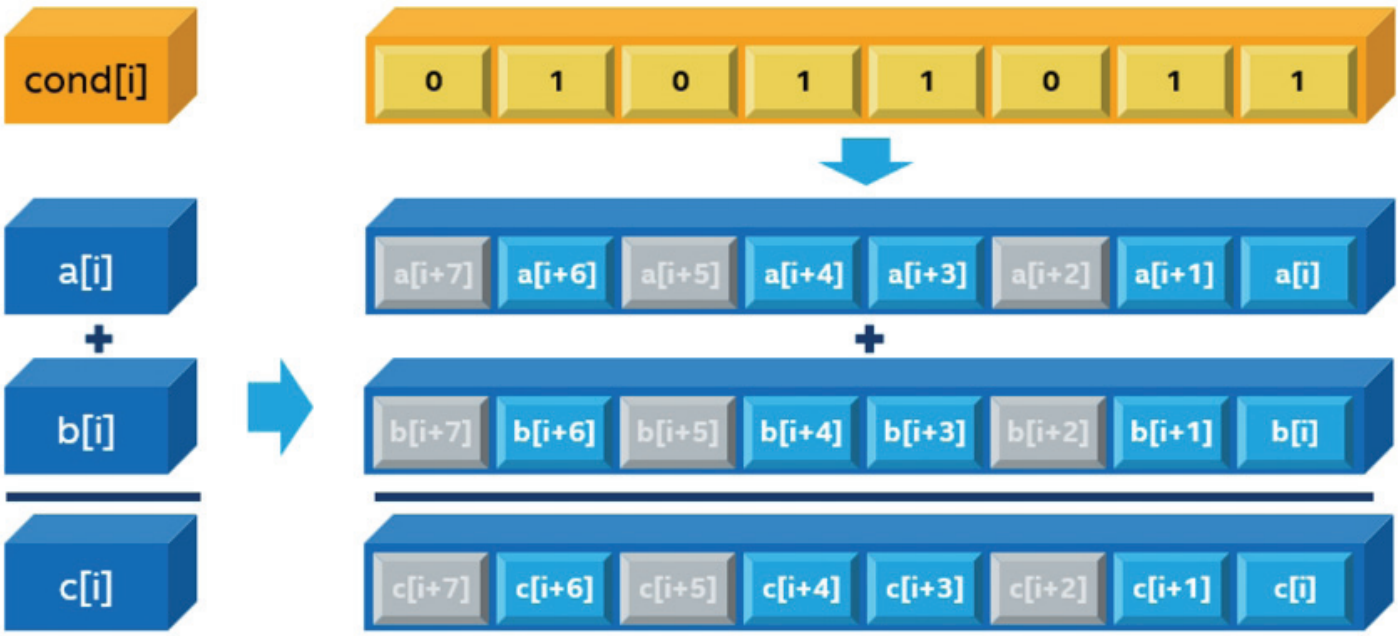
- vectorizar usando las instrucciones de lenguaje máquina apropiadas, y en este caso seccionar

```
s= 0
p= 1
do i=1,2
  S1: A(64(i-1)+1:64i)= B(64(i-1)+1:64i) + C(64(i-1)+1:64i)
  S2: s = s + SUM(64(i-1)+1:64i)
  S3: s = p * PROD(64(i-1)+1:64i)
enddo
```

Bucles con condicionales: operaciones con máscara

Códigos con muchos saltos, por ejemplo, un bucle con una sentencia condicional en su cuerpo, pueden ser vectorizados usando máscaras vectoriales que bloquean las operaciones para las cuales la condición no es cierta

```
for (i = 0; i < MAX; i++)
  if (cond(i))
    c[i] = a[i] + b[i]
```



Punteros en C[1]

The following loop may not get vectorized because of a potential aliasing problem between pointers `a`, `b` and `c`

```
void add (float *a, float *b, float *c) {
  for (int i=0; i<SIZE; i++)
    c[i] += a[i-1] + b[i];
}
```

If the **`restrict`** keyword is added to the parameters,
the compiler will trust you, that you will not access the memory in question with any other pointer and vectorize the code properly

```
void add (float * __restrict a,
          float * __restrict b, float * __restrict c) {
  for (int i=0; i<SIZE; i++)
    c[i] += a[i-1] + b[i];
}
```

The downside of using `restrict` is that not all compilers support this keyword, so your source code may lose portability.

- `restrict` es un cualificador complicado, que puede ayudar al compilador a generar ejecutables mas rápidos en contextos no relacionados con vectorizar. Para mas información:
  How to Use the restrict Qualifier in C, Douglas Walls, Sun ONE Tools Group, Sun Microsystems, July 2003 (revised March 2006)
  http://web.archive.org/web/20120225055041/http://developers.sun.com/solaris/articles/cc_restrict.html

1) http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/optaps/common/optaps_vec_use.htm

7. LEY DE AMDHAL

- Modelo matemático sencillo que calcula la mejora en rendimiento al ejecutar parte de las operaciones de un programa en menos tiempo:
  - un modo lento (escalar)
  - un modo rápido (vectorial)
- Sea **N** el número de operaciones en coma flotante de un programa y $\mathbf{f_v}$ el porcentaje que puede realizarse en modo vectorial
  - $TPF_{(s,v)}$: tiempo por FLOP en modo (escalar, vectorial)
  - $G = TPF_s/TPF_v$

$$\text{Speed\_up} = T_{escalar} / T_{vectorial} = \frac{N \cdot TPF_s}{N \cdot f_v \cdot TPF_v + N \cdot (1 - f_v) \cdot TPF_s} =$$

$$= \frac{1}{f_v \cdot \frac{TPF_v}{TPF_s} + (1 - f_v)} = \frac{1}{(1 - f_v) + f_v \cdot \frac{1}{G}} = \frac{1}{f_s + f_v / G}$$

- Speed_up ($f_v \to 1$) $\to$ G
- $G \to \infty \Rightarrow$ Speed_up $\to 1/fs$

$$\Downarrow$$

$$fs \uparrow\uparrow \Rightarrow \text{Speed\_up} \downarrow\downarrow$$

"El modo lento va a limitar, aunque el modo rápido sea fantástico"

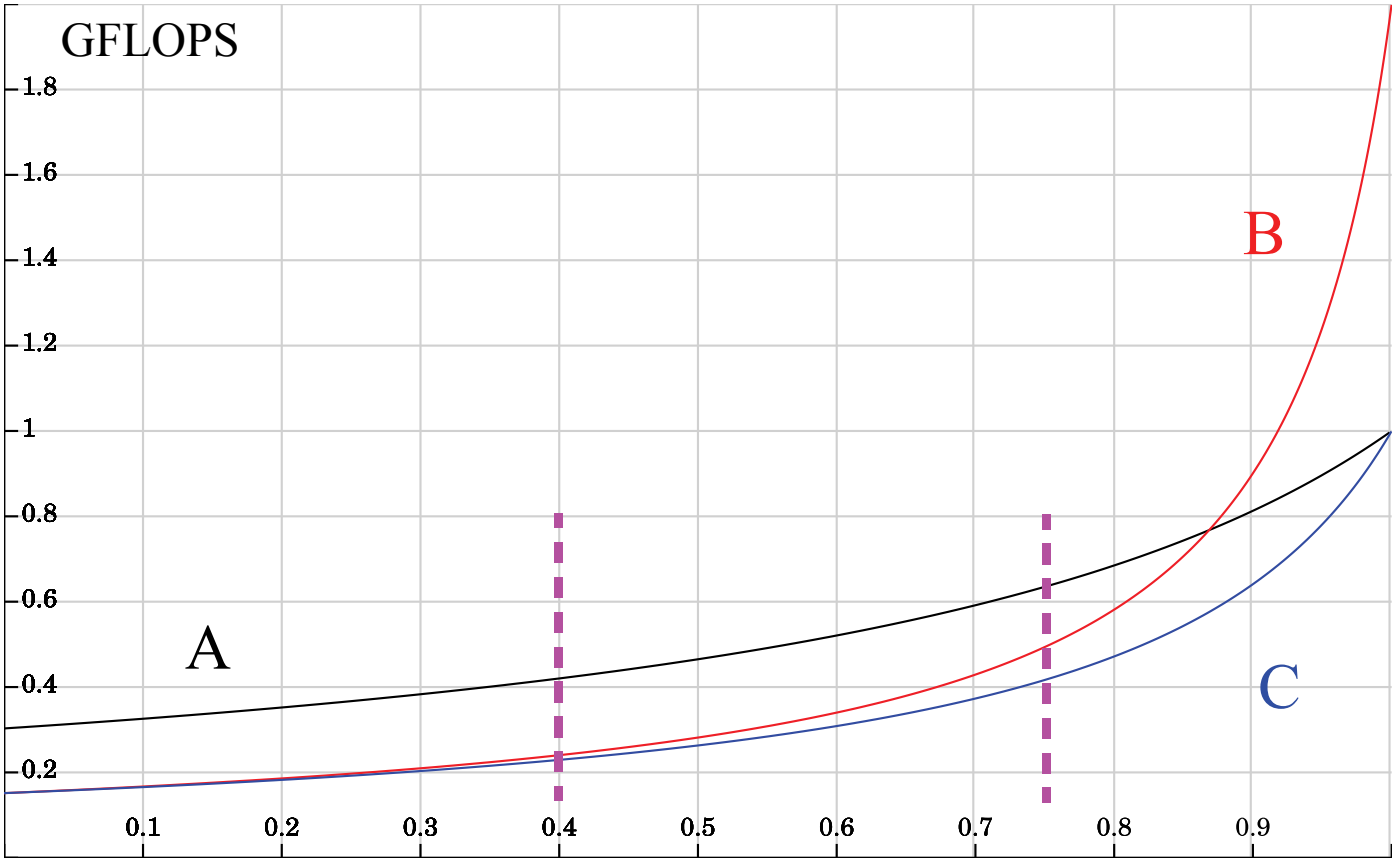
- Valor habitual de f_v entre 0,4 y 0,75 para programas compilados

$$\Downarrow$$

Un buen computador vectorial debe tener un buen procesador escalar

- Caso de estudio:
  - Computador A: $TPF_s = 3{,}3ns$; $TPF_v = 1ns$ (G= 3,3)
  - Computador B: $TPF_s = 6{,}6ns$; $TPF_v = 0{,}5\ ns$ (G= 13,2)
  - Computador C: $TPF_s = 6{,}6ns$; $TPF_v = 1\ ns$ (G= 6,6)

$$R_v = R_s \cdot Sup = R_s \cdot \frac{1}{f_s + f_v / G}$$



Permalink to this graph:

http://fooplot.com/plot/2hqbw5aoc7