3º Grado en Ingeniería Informática
Multiprocesadores
Coleccion de Problemas 2012/2013

Índice

VECTORIALES

1 Memoria multibanco. *1/2 Resuelto*. Sep. 98 2

2 Rendimiento de código vectorial. *1/2 Resuelto*. Sep. 08 3

3 Indices de prestaciones de un código vectorial: C_n, R_n, R_∞, N1/2, Nv. Sep. 96 7

4 Memoria cache escalar en un computador vectorial. Feb. 96 10

5 Rendimiento de código con stride. *Resuelto*. Ene. 04 11

6 Acceso a memoria usando vector de índices: programación y rendimiento. Feb. 97 . . 13

7 Seccionado con soporte de Lenguaje Máquina.
Stride y rendimiento.*1/2 resuelto*. Feb. 98. 17

8 Procesador con Encadenamiento General y red Crossbar. *1/2 resuelto*. Sep. 98 21

PROCESADOR VECTORIAL ZV

Descripción del procesador 23

COMPILACIÓN Y DEPENDENCIAS

9 Dependencias 1-D y vectorizar. *Resuelto*. Jul. 98 27

10 Dependencias 1-D, vectorizar y paralelizar. *Resuelto*. Sept. 08 29

11 Dependencias 2-D, vectorizar y paralelizar. Ene. 08. 31

REDES DE INTERCONEXION

12 Caminos posibles en un hipercubo. 33

13 Patrón alternativo al perfect-shuffle. 34

14 Conmutación de circuitos vs. paquetes en bus compartido. *1/2 resuelto*. 35

COHERENCIA CON INTERCONEXIÓN BUS: OBSERVACIÓN (SNOOPY)

15 Protocolo sencillo basado en invalidación. Feb. 96. 37

16 Evolución temporal en un protocolo de invalidación. Jul. 96 38

17 Protocolo de actualización MESI. Patrones de compartición: coste en ciclos. Sep. 96 43

18 Caracterización de un programa paralelo real con invalidación MESI. Jul. 97. 51

19 Caracterización de un programa paralelo real con actualización MOESI.
1/2 Resuelto. Sep. 97 55

20 Caches multinivel y coherencia por invalidación MSI. Jul. 98 59

21 Primitiva de sincronización FETCH&INC y coherencia MSI. *Resuelto*. Feb. 98. 63

COHERENCIA CON INTERCONEXIÓN GENERAL: DIRECTORIO

22 Coherencia MSA+MSI basada en directorio + Hipercubo. Jun. 07 69

23 Multiprocesador en chip tipo Niagara + coherencia por invalidación. Sept. 08 71

24 Multiprocesador para cargas de trabajo comerciales.
1/2 Resuelto. Sept. 2011 74

25 Coherencia en un CMP con directorio y comunicación en malla. Feb. 2012 80

Víctor Viñals Yúfera y Jesús Alastruey Benedé

Área Arquitectura y Tecnología de Computadores

Dpto. Informática e Ingeniería de Sistemas

Escuela de Ingeniería y Arquitectura. Universidad de Zaragoza

http://www.youtube.com/watch?v=8FkGyJGw68k

Witness (1985)
Único testigo

Directed by Peter Weir.Paramount
Harrison Ford, Kelly McGillis, Josef Sommer, Lukas Haas, Jan Rubes, Alexander Godunov, Danny Glover.

Pero también procede ver:
What a wonderful world this would be
http://es.youtube.com/watch?v=3p_tvjqSrBk

1) **Memoria multibanco.** *1/2 Resuelto* Sep. 98

Sea un sistema de memoria de 32 GB, construido con 16 bancos de 8 ciclos de latencia. En bancos consecutivos tenemos palabras consecutivas de 8 bytes (entrelazado de palabras o entrelazado de orden bajo).

a) Descomponer la dirección de memoria en los campos necesarios

Dirección

b) Proponer dos valores diferentes para **Rstr** de forma que la instrucción **LVWS** suministre a una latencia de finalización (Ce) de 2 ciclos por elemento.

`LVWS` carga palabra de doble precisión en `V0` y `Rstr` codifica separación en bytes

```
LVWS    V0,(Rbase, Rstr)
```

Valor 1 = *32 (bytes)*

Valor 2 = *96, 160, 224, 352, ...*

2) Rendimiento de código vectorial. *1/2 Resuelto* Sep. 08

Estudia el rendimiento de este código en el procesador vectorial ZV. Restricción: las lecturas con stride que tardan más de un ciclo en suministrar cada elemento NO encadenan. Se añade una una Unidad Funcional de Reducción, con Latencia = 5 ciclos y Ce = 2 ciclo/elem.

R_A= 0x1000, R_B= 0x8000, R_C= 0xF000
R_{s1}=96, R_{s2}=80, R_{s3}=72

```
1$:   SVWS   RA,V1,Rs1
      LVWS   V2,RB,Rs2
      LVWS   V3,RC,Rs3
      MULSV  V1,V2,V3
      SUM    F1,V1
```

sección residual al final

```
   código
   de seccionado
```

; 5 instrucciones escalares independientes
; salto no retardado, @dst se calcula en **D1**

SUM F1,V1; es una instrucción de reducción ; reducción: $F1 = F1 + \sum_{i=0}^{MVL-1} V(1)\langle i\rangle$

```
SVWS    RA,V1,Rs1   BDDL
                    12
LVWS    V2,RB,Rs2
LVWS    V3,RC,Rs3
MULSV   V1,V2,V3
  SUM   F1,V1
```


a) Dibuja el diagrama de dependencias de registros (entre iteraciones también)

SVWS

LVWS

LVWS

MULSV

SUM

b) Dibuja el diagrama de tiempos para n=64 elementos, señalando el principio de los encadenamientos y las detenciones por cualquier causa. Anota el ciclo de finalización de cada instrucción vectorial y de los puntos de interés (¡ hazlo primero en sucio!).

c) Calcula Cn para n <= 64

Cn = 31 + 3n

d) Repite el diagrama de tiempos para las dos primeras secciones (ABAJO). Anota los ciclos de inicio y finalización de cada instrucción vectorial para la segunda sección. Señala el principio de los encadenamientos y las detenciones que no hayan aparecido antes.

e) Calcula Cn para n >> 64

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

f) Calcula Rn para n >> 64

Rn =

g) Calcula R∞

R∞ = 914,3 MFLOPS

h) Calcula $N_{1/2}$

$N_{1/2}$ = 23

SVWS	R_A,V1,R_{s1}	BDDL / 12	
LVWS	V2,R_B,R_{s2}		
LVWS	V3,R_C,R_{s3}		
MULSV	V1,V2,V3		
SUM	F1,V1		

3) **Indices de prestaciones de un código vectorial**: C_n, R_n, R_∞, $N_{1/2}$, N_v Sep. 96

Sea una procesador de la arquitectura vectorial DLXV con la siguientes características:

- 8 registros vectoriales de 64 elementos.
- 2 caminos de lectura y uno de escritura por registro.
- 1 sumador (L = 5 ciclos) y un multiplicador (L = 8 ciclos).
- 1 puerto de acceso a memoria.
- memoria entrelazada con 16 módulos, latencia = 12 ciclos.
- puede haber encadenamiento general entre dos instrucciones cualesquiera, *incluso* después de una instrucción `LV`.
- las instrucciones escalares se solapan con las vectoriales y se ejecutan con un CPI unidad.
- frecuencia del procesador = 1Ghz
- suponed la segmentación del procesador ZV.

Sea el siguiente código:

```
            ; 7 instrucciones escalares para preparar
            ; el procesamiento vectorial

1$:    lv     v1, r1        ; v1[0..63]= Mem[r1..r1+63*sizeof(elem)]
     multsv   v2, v1,f0
       sv     r2, v2

            ; 6 instrucciones escalares para realizar
            ; seccionado y salto a 1$
```


Se pide lo siguiente:

a) Diagrama temporal de ejecución: una instancia y media del bucle `1$`.

b) Ciclos de ejecución de este código para valores de *n* menores o iguales de 64.

C*n* =

c) Ciclos de ejecución de este código para valores de *n* elevados (en relación al tamaño de los registros vectoriales)

C*n* =

d) Ciclos de ejecución de este código para cualquier valor de *n*.

e) Calcular R_∞

R_∞ =

f) Calcular $N_{1/2}$

$N_{1/2}$ =

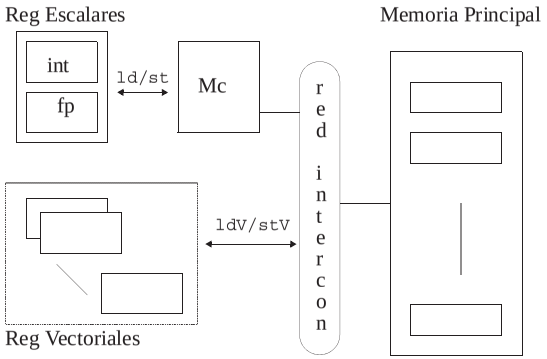
g) Calcular N_v. Para ello considerad que el bucle escalar equivalente tiene 10 instrucciones y que la multiplicación en punto flotante escalar tiene una latencia de 4 ciclos. La memoria es la misma. El procesador escalar tiene 5 etapas y la etapa de memoria o de cálculo es multiciclo. No existen otras dependencias que las del cálculo vectorial. El tiempo de ciclo es el mismo.

Diagrama temporal:

N_v =

4) Memoria cache escalar en un computador vectorial

Los computadores vectoriales tienen una memoria cache *escalar* para los datos referenciados desde las instrucciones escalares (datos enteros o de punto flotante). La actividad de las instr. `load/store` vectoriales se dirige directamente a la memoria principal entrelazada. El esquema siguiente sintetiza la situación.



a) Ventajas de esta organización frente a la mas conservadora de no tener memoria cache escalar.

b) Inconvenientes de la incorporación de la memoria cache escalar.

c) Si has detectado algún freno o inconveniente, expon solución(es) hardware.

5) Rendimiento de código con stride. *Resuelto.* Ene. 04

Estudiad el rendimiento de este código en el procesador vectorial ZV con las siguientes modificaciones:

- existen 32 bancos de memoria y cada banco contiene palabras de 8B
- las lecturas con stride NO encadenan.

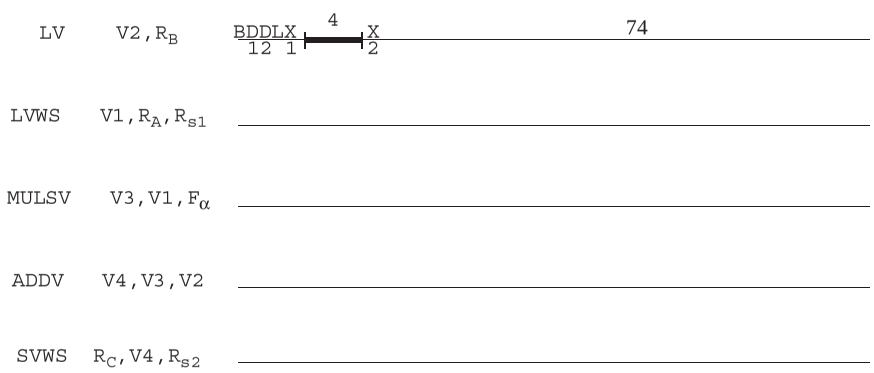
```
                  R_A= 0x1000, R_s1= 272, R_B= 0x8000
                  F_α= 3.1416, R_s2= 640, R_C= 0xF000
1$:    LV     V2,R_B
       LVWS   V1,R_A,R_s1
       MULSV  V3,V1,F_α
       ADDV   V4,V3,V2
       SVWS   R_C,V4,R_s2
```

parte residual al final

```
código
de seccionado
```

; 5 instrucciones escalares independientes

; salto no retardado, @dst se calcula en **D1**



a) Dibuja la primera sección y el principio de la siguiente, señalando el principio de los encadenamientos y las detenciones por cualquier causa. Anota el ciclo de finalización de cada instrucción vectorial (¡ hazlo primero en sucio!).

b) Calcula Cn para n >> 64

$Cn = \lceil n/64 \rceil \cdot 30 + 2 \cdot n$

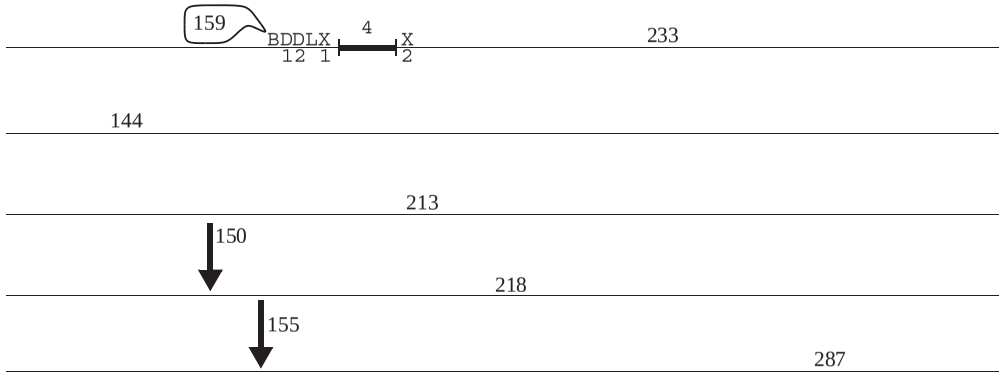
c) Calcula R∞, por supuesto con MVL=64.

$$R\infty = \frac{2 \cdot n \text{ FLOPS}}{(30/64 + 2) \cdot n \cdot 10^{-9} s \cdot 10^{6}} = 810,12 \text{MFLOPS}$$

d) Calcula C_{69} exactamente, escogiendo como último ciclo el último ciclo de la instrucción que termine la última.

$C_{69}^{exacto} = 301$ ciclos

$C_{69}^{apartadob} - C_{69}^{exacto} = -103$ ciclos



6) Acceso a memoria usando vector de índices: programación y rendimiento Feb. 97

Sea el siguiente fragmento de código:

```
INTEGER K(0:1000)
REAL*8  A, B(0:1000), C(0:2000), D(0:1000)

DO i= 0, 1000
  A = A +   B(i) * C(K(i))
  D(i) = B(i) + 3.0
ENDDO
```

Suponed que la variable `A` reside en el registro f_a, que la constante 3.0 está en f_3 y que las direcciones de inicio de los vectores `K, B, C` y `D` están en los registros: r_K, r_B, r_C y r_D.

a) Escribid el equivalente en ensamblador DLXV sin reordenar código, vectorizando lo más posible, pero sin incluir la parte escalar de seccionado. Esta parte de seccionado gestiona los valores de r_K, r_B, y r_D. Si necesitáis alguna instrucción vectorial no incluida en el repertorio DLXV, indicadlo, explicando su comportamiento. Notad la diferencia entre *índices* `k(i)` y *desplazamientos* `k(i)*8`.

b) Diagrama de dependencias de las instrucciones vectoriales (no de las sentencias Fortran)

c) Dibujad un cronograma de ejecución de la primera sección y media, ejecutándose en un procesador ZV con las siguientes variaciones:

- Solamente una UF (suma, producto, reducción) con latencia de 6 ciclos. Para la reducción suponed latencia de iniciación de 3 ciclos/elem.
- 1 único puerto de memoria (lec. o escr.). 8 módulos. Latencia de 8 ciclos.
- Encadenamiento general salvo después de cualquier instrucción de acceso a memoria.
- Suponed C_{base}= 15 y 5 instrucciones escalares de fin de bucle.
- **El vector `K` está inicializado con una secuencia de índices pares:** `K(i) = 2i`

Cn (n > 64) =

d) Reordenar el código para minimizar el tiempo de ejecución.

e) Dibujad un cronograma simplificado de ejecución, indicando el nuevo Cn .

Cn (n > 64) =

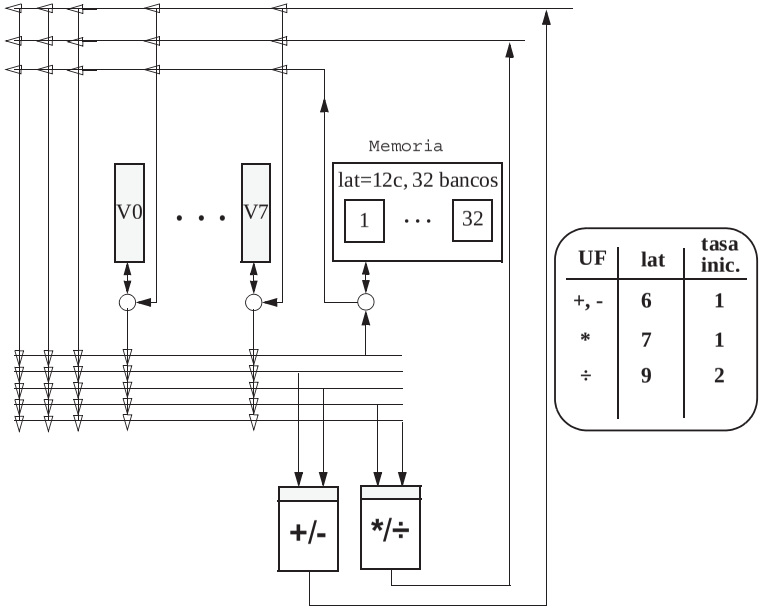
7) Seccionado con soporte de Lenguaje Máquina.
Stride y rendimiento. *1/2 resuelto* Feb. 98

Partimos de la arquitectura DLXV. Para que el seccionado (*strip mining*) sea mas eficiente, se modifica la instrucción `movi2s`,de forma que su comportamiento sea como en la arquitectura vectorial Convex C-2 (r.i.p.):

```
1$:  movi2s   VLR,Rx        ; si Rx <= 0 ent VLR = 0
                            ; sino si Rx >= 64 ent VLR = 64
                            ;      sino VLR = Rx
```

Suponemos una organización como la de la figura, que permite encadenamiento directo entre cualquier pareja de instrucciones SALVO si a) la instrucción fuente es `LVWS` ó b) la velocidad de consumo es menor que la de producción. La segmentación en etapas es:

- E1 (A): Búsqueda de instrucción.
- E2 (B): Lectura de reg. escalares, decod., detención por riesgo de dato o recurso.
- E3 (C): Establecer caminos en las redes de interc., lectura de reg. vectoriales.
- E4: Primer ciclo de ejecución o memoria. Observad en la figura las latencia y tasas de iniciación para cada UF. Observad también las particularidades de la memoria.
- En: Escritura de registro vectorial una vez transcurrida la latencia de la UF.



UF	lat	tasa inic.
+, -	6	1
*	7	1
÷	9	2

Los recursos para las primeras etapas (A y B) se comparten entre instrucciones escalares y vectoriales, pero más allá existen recursos separados que permiten su ejecución paralela.

Considerad el siguiente fragmento de programa que procesa elementos de doble precisión:

```
...
Do i= 1,n
  B(i)= (A(i) - A(2*i))/5.0
EndDo
```

`n` es una variable, desconocida en compilación y asignada al registro `Rn`. La constante 5.0 está en `F5` y las direcciones de inicio de A y de B están en `Ra` y `Rb`. La constante 2 está `R2`.

a) Compilad el fragmento, recorriendo las expresiones de izquierda a derecha y desconectando cualquier optimización que reordene. Incluid el código de seccionado.

```
        mov     Ri, Rn          ; temporal para el seccionado
        addi    Raa, Ra, #8     ; @ A(2)
        shl     R2, R2, #3      ; en R2 el stride en bytes
secc:   movi2s  VLR, Ri
        LV      V0, Ra
        LVWS    V1, (Raa, R2)
        SUBV    V2, V0, V1
        DIVVS   V3, V2, F5
        SV      Rb, V3
        addi    Ra, #64*8, Ra
        addi    Raa, #64*16, Raa
        addi    Rb, #64*8, Rb
        subi    Ri, #64, Ri
        sgt     R1, Ri, R0
        bnz     secc, R1        ; rellenar hueco si el salto es retardado
```

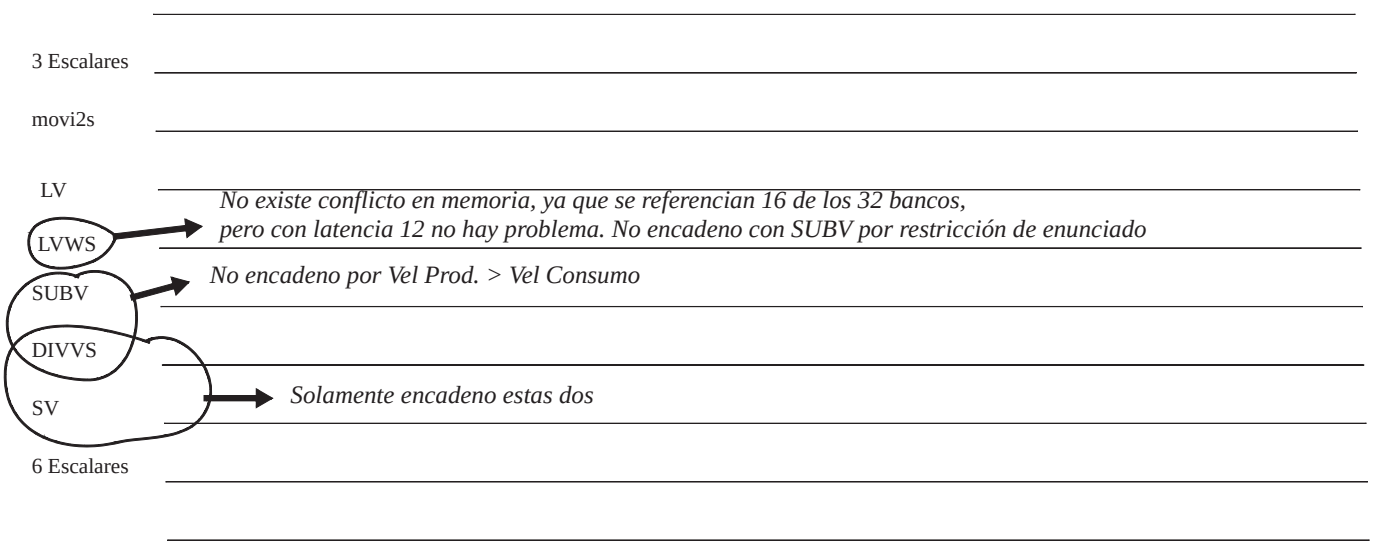
b) Mostrad un diagrama de ejecución de las instrucciones vectoriales, de forma que se aprecie el comportamiento para vectores grandes. Determinad *Cn* para *n* grandes y R_{30} suponiendo un tiempo de ciclo de 5 ns. Contesta en *página siguiente.*

c) Si es posible, reordenad el código vectorial para mejorar el tiempo de ejecución:

Las dependencias sólo permiten intercambiar los dos primeros LOADS VECTORIALES. Es beneficioso porque LV en segundo lugar puede encadenar con SUBV

d) Mostrad el nuevo diagrama de ejecución de las instrucciones vectoriales, de forma que se aprecie el comportamiento para vectores grandes. Determinad *Cn* para *n* grandes y R_{30} suponiendo un tiempo de ciclo de 5 ns. Contesta en página siguiente.

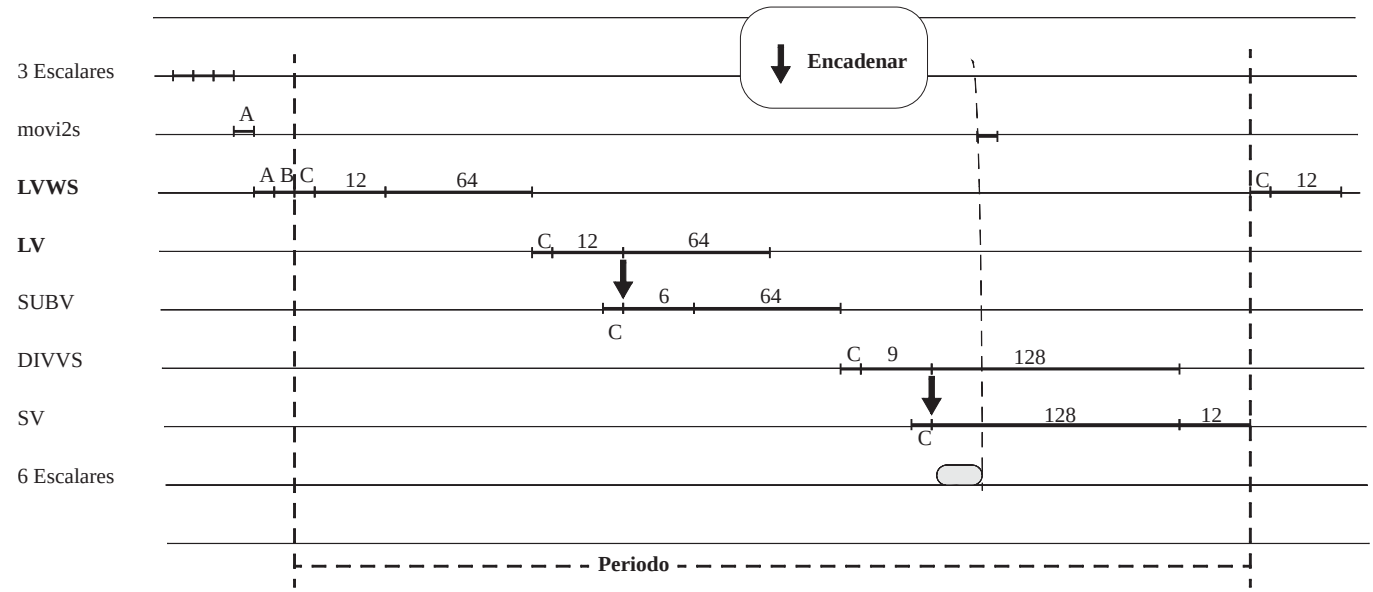
b) *diagrama de ejecución de las instrucciones vectoriales* **sin reordenar**, de forma que se aprecie el comportamiento para vectores grandes. Determinad *Cn* para *n* grandes y R_{30} suponiendo un tiempo de ciclo de 5 ns.



Cn = $Cn|_{periodo} = \left\lceil \frac{n}{64} \right\rceil \cdot 54 + 5 \cdot n \approx 5,8 \cdot n$

R_{30} = 57,1 Mflops

d) *diagrama de ejecución de las instrucciones vectoriales* ***reordenadas***, de forma que se aprecie el comportamiento para vectores grandes. Determinad *Cn* para *n* grandes y R_{30} suponiendo un tiempo de ciclo de 5 ns.



Cn = $Cn|_{periodo} = \left\lceil \frac{n}{64} \right\rceil \cdot 54 + 4 \cdot n \approx 4,8 \cdot n$

R_{30} = 66,6 Mflops

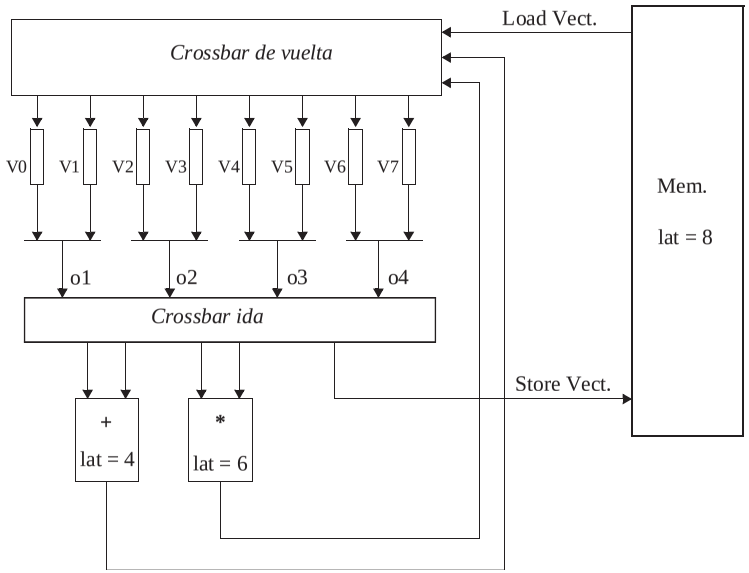
(6+54+4x30 = 180c)

8) **Procesador con Encadenamiento General y red Crossbar.** *1/2 resuelto* Sep. 98

En la figura se muestra el camino de datos vectorial de un procesador vectorial con:

- 8 registros vectoriales de 64 elementos cada registro (V0, ...,V7).
- 2 caminos vectoriales con memoria, uno para `loads` y otro para `stores`.
- 2 unidades aritméticas, una para suma y otra para multiplicación.
- 2 redes de interconexión *crossbar*. Una red *crossbar* puede interconectar cualquier entrada con cualquier salida.

La unidad de control es capaz de utilizar eficientemente el hardware disponible. Suponemos una la segmentación ZV.



Se pide el diagrama instrucciones-tiempo para los siguientes trozos de código, que operan con longitud vectorial 64. En cada diagrama indicad las razones de detención, los encadenamientos y el *Ce* conseguido (ciclos por elemento).

a)
```
V1 <- V2 + V4
V5 <- V1 * V3
```

b)
```
V0 <- V2 + V5
V2 <- V0 * V6
```

c)
```
V0 <- Load(Rx)
V6 <- V0 * V5
V0 <- V6 + V3
```

d)
```
V4 <- Load(Rx)
V2 <- V4 * V1
V1 <- Load(Ry)
V2 <- V1 + V2
```


Códigos **a)** y **b)**

`V1 <- V2 + V4`	
`V5 <-V1*V3`	

Ce = *2, detención por conflicto en o2, en la instr. I2*

`V0 <- V2 + V5`	
`V2 <- V0 * V6`	

Ce = *1, sin detención*

Códigos **c)** y **d**

`V0 <- Load(Rx)`	
`V6 <- V0 * V5`	
`V0 <-V6+V3`	

Ce = *2, detención por V0in ocupado en I3*

`V4 <- Load(Rx)`	
`V2 <- V4 * V1`	
`V1 <- Load(Ry)`	
`V2 <- V1+V2`	

Ce = *2, detención por un sólo puerto de mem. en I3*

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:

		aluV	LV	SV
B	búsqueda instrucción			
D1	Decod. 1/ @dst salto			
D2	Decod. 2		cálculo @ base	
L	lanzamiento. ¿riesgos en recursos? ¿riesgos en registros? reserva recursos	lectura reg vectoriales	*retardo*	lectura reg vectoriales
X1	red IDA	paso crossbar 1	generación @	paso crossbar 1
alu/ mem	operación o acceso a memoria	op	mem_rd	mem_wr liberar recursos en último ciclo
X2	red VUELTA	paso por crossbar 2		
W		escritura registro vectorial destino liberar recurso en último ciclo		

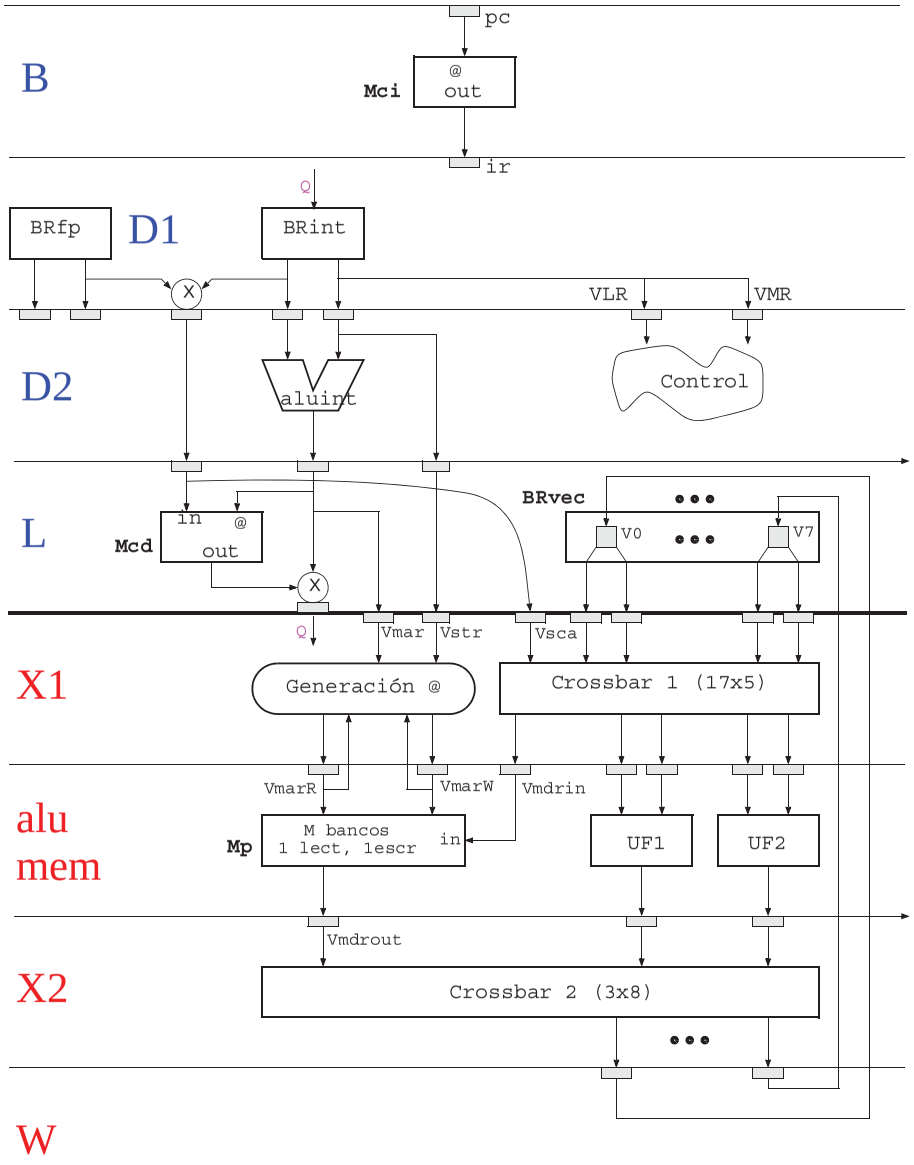
La ruta de datos se muestra en la siguiente hoja. La parte izquierda corresponde a un trozo del procesador escalar. Su operación y su relación con la parte vectorial puede deducirse observando la ruta de datos.

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 concurrente*.

No hay control hardware de colisión entre lectura y escritura en memoria. Se solapan los flujos y el compilador se preocuparía 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. El control está preparado para ejecutar de forma solapada una instrucción dependiente en dos casos:

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



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:

Tabla 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

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

:

Tabla 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	x

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

:

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

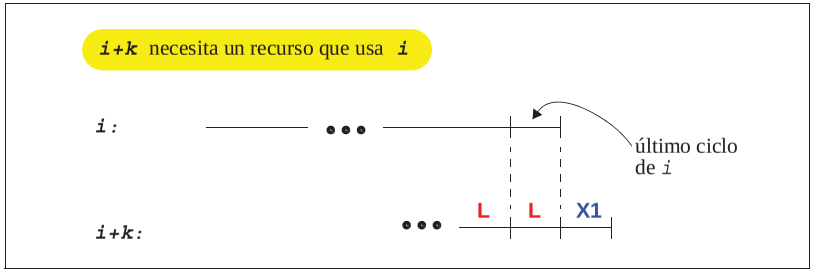
B	x													
D1		x												
D2			x											
L				x	x	x	x	x	x		Lat_mem			
X1					x	x	x	x	x	x				
mem						x	x	x	x	x	x	x	x	x
X2														
W														

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)

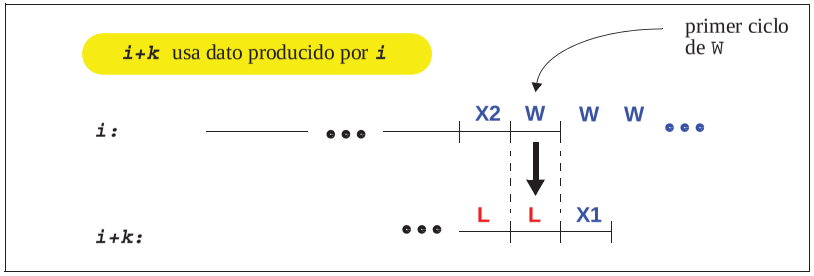
La etapa **L** se encarga de controlar los riesgos en las instrucciones vectoriales por dependencias de datos (en registros o memoria) y por falta de recursos:

- 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.



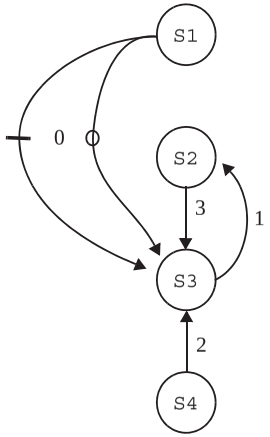
Suponed un procesador con las siguientes características:

MVL	64	Tamaño registros vectoriales
+	Lsum = 2, TI = 1	Latencia, tasa de Iniciación
*	Lmul = 3, TI = 1	
mem	8bancos, Lmem = 4	
Tc (*f*)	1ns (*1000 Mhz = 1GHz*)	

Dado el siguiente código,

```
    Do i=1, n
S1:   A(i)= A(i) + C(i)
S2:   B(i)= A(i-1) + C(i+2)
S3:   A(i)= B(i-3) + D(i-2)
S4:   D(i)= C(i) + D(i)
    EndDo
```

a) dibuja el diagrama de dependencias



b) vectoriza el código, indicando si deben mantenerse bucles escalares o si pueden lanzarse ejecuciones vectoriales. Indica este último caso con la notación **`Si(1:N)`**

```
        S1(1:N)                    S4(1:N)
        S4(1:N)                    S1(1:N)
esc [   S2          ó      esc [   S2
        S3                         S3
```

¿ Qué transformación has utilizado?

REORDENAR (S4 antes que S2-S3) y

Vectorizando PARCIAL (S4 y S1 completa, S2 y S3 escalar)

Dado el siguiente código:

```
    real*8  A(),B(),C()
      ...
    do i= 1,4
S1:   B(i)= A(i+14)
S2:   A(3i+1)= C(i)
    enddo
```

a) Aplica el test MCD y comenta si pueden existir dependencias en el código anterior.

SI pueden en el Vector A(), pq mcd(1,3) divide a (14-1)

b) Dibuja las dependencias que encuentres, etiquetándolas con iteración_fuente - iteración_destino
y comenta brevemente el resultado del apartado **a)**.

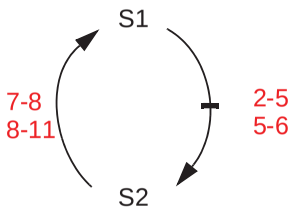
S1

En el espacio de iteraciones definido (i=1,4) no existe NINGUNA dependencia.

Bien, pq el test MCD sólo calcula una condición NECESARIA pero no SUFICIENTE

S2

c) Cambiamos el límite superior del bucle, de 4 a 12.
Dibuja las dependencias que encuentres, etiquetándolas con iteración_fuente - iteración_destino
y comenta brevemente el resultado del apartado **a)**.



En el nuevo espacio de iteraciones definido (i=1,12) existen dependencias.

Bien, pq el test MCD ya había anticipado la posibilidad

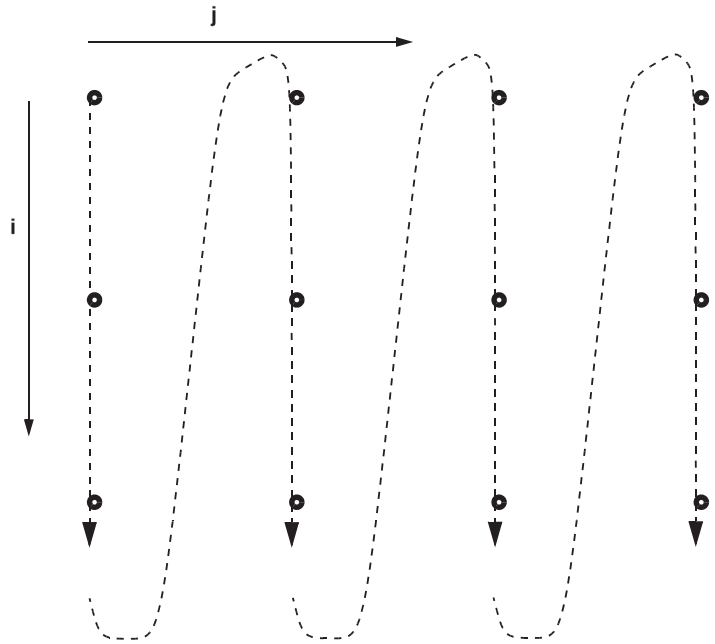
d) Escribe la lista de iteraciones independientes (pueden ejecutarse en paralelo).

i = { 1, 2, 3, 4, 7, 9, 10, 12 }

Dado el siguiente código:

```
      real*8  A( , )
        ...
      do j= 1,128
        do i= 1,4
S1:       A(j+2,i)= A(j,i) + B(i,j)
        enddo
      enddo
```

a) Dibuja las dependencias sobre el diagrama de recorrido del Espacio de Iteraciones:



Queremos ejecutar este código en un multiprocesador de memoria compartida, en el que los procesadores tienen capacidad de ejecución vectorial.

Los apartados siguientes preguntan sobre el paralelismo y las propiedades de intercambio de los bucles.

b) Decid si son posibles los modos de ejecución siguientes, añadiendo una explicación muy corta. SEC, VEC y PAR significa ejecución secuencial, vectorial y paralela.

```
j PAR
  i SEC
```
si/no ☐

```
j SEC
  i VEC
```
si/no ☐

```
j PAR
  i VEC
```
si/no ☐

c) ¿Es correcto intercambiar los bucles?. Explicación muy corta

d) SI es correcto intercambiar los bucles, repetid el análisis anterior.

```
i PAR
  j SEC
```
si/no ☐

```
i SEC
  j VEC
```
si/no ☐

```
i PAR
  j VEC
```
si/no ☐

e) En un multiprocesador con cuatro procesadores vectoriales,
¿qué cuestión debe plantear el compilador para seleccionar un código de los anteriores?.

12) Caminos posibles en un hipercubo

En una red hipercubo de grado 12 cuantos caminos alternativos de *n* pasos existen entre nodos cuya dirección se diferencia en *n* bits? (1 =< n =< 12) ¿Por qué?

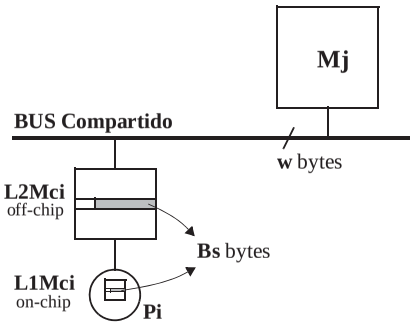
13) Patrón alternativo al perfect-shuffle

Proponed un patrón *alternativo* al barajado perfecto (perfect-shuffle) de una red Ω 8x8 con conmutadores 2x2.

a) Suponiendo que los 8 procesadores están en la izquierda y los ocho módulos de memoria en la derecha, dibujar sobre la red propuesta los ocho caminos desde el procesador P7 a los módulos M0..M7

b) Sobre el control de los conmutadores 2x2: ¿que bits deben utilizarse en cada etapa para dirigir una petición a su módulo? Para indexar los módulos se utilizan los tres bits de mas peso de la dirección de memoria: d_2(+peso), d_1, d_0(-peso)

14) Conmutación de circuitos vs. paquetes en bus compartido *1/2 resuelto*



Dado este multi de memoria compartida, en primer lugar definimos las características temporales del bus y de la memoria principal:

• Tiempo de acceso a N palabras consecutivas en el módulo Mj de memoria principal:

$$T(N) = T_{LAT} + (N-1)T_{RAF};\ N = Bs/w$$

• El *ciclo de bus* (*cb*) es el tiempo de la transacción más simple a través del bus, siendo un múltiplo del Tc del proc.

$$T_{BUS} = cb = T_{RAF} = kTc$$

- La latencia de la memoria medida en ciclos de BUS es:

$$LAT = T_{LAT}/T_{BUS} \text{ ciclos de bus.}$$

Queremos calcular el ancho de banda (BW) en palabras de *w* bytes por segundo, suponiendo que las únicas transferencias de datos en el bus son bloques de las Mc de segundo nivel. A continuación detallamos el protocolo de bus en los dos modos de funcionamiento.

- Conmutación de circuitos: fallo de L2Mci al bloque *x* del módulo Mj:

`{Mci,Mj}(rB,@x)`[1]	1 *cb*	arbitraje, conmutación de tristates y transferencia de dirección. Se adquiere la propiedad del bus
lectura en Mj	LAT -1 *cb*	el bus sigue ocupado
`Mci + x`[2]	N *cb*	transferencia y carga de las N palabras del bloque en la cache de segundo nivel

- Conmutación de paquetes: idem

`{Mci,Mj}(rdB,@x)`	1 *cb*	arbitraje, conmutación de tristates y transferencia de dirección
lectura Mj	LAT -1 *cb*	el bus esta libre
`{Mj,Mci}(devB)`	1 *cb*	arbitraje, conmutación de tristates y anuncio de devolución de bloque
`Mci + x`	N *cb*	transferencia y carga de las N palabras del bloque en la cache

Para dar mayor eficiencia al sistema, cada módulo de memoria tiene una cola de peticiones (*buffer*). Una entrada de la cola tiene: <@, tipo operación, número procesador.>.

1. {fuente, destino}(acción, [argumentos de la acción]). "rB" = *read block*
2. Indica la carga del bloque *x* en memoria cache, en este caso mediante N escrituras consecutivas.

Se pide lo siguiente:

a) Calcular el ancho de banda máximo del modo conmutación de circuitos, en palabras/sg.

RESPUESTA: $BW_{cc} = \dfrac{N}{(LAT + N) \times T_{BUS}}$

b) Calcular el ancho de banda máximo del modo conmutación de paquetes, en palabras/sg, suponiendo que los huecos de ocupación del bus se ocupan con tráfico útil.

RESPUESTA: $BW_{cp} = \dfrac{N}{(N + 2) \times T_{BUS}}$

c) Calcular la latencia frontera que determina cual de los dos modos es más eficiente. Haced una gráfica con los anchos de banda relativos al máximo teórico (1 pal / ciclo bus) para N= (4, 8) y variando la latencia entre 0 y 16 ciclos.

d) Suponiendo que los anchos de banda calculados son netos, es decir que contribuyen íntegramente a la ejecución de programas, calculad el número de procesadores que puede soportar un sistema como el descrito con las siguiente características:

- Tc = 5ns; T_{BUS}= T_{RAF}= 10ns; T_{LAT}= 80ns
- N= 8
- L1Mc unificada, con tasa de fallos del 10%; L2Mc unificada, con tasa de fallos *local*[1] del 1%.

e) ¿Cómo podríamos tener en cuenta la disminución en ancho de banda debida a los conflictos en los módulos de memoria?. Solamente ideas.

1. Es decir fracción de fallos con respecto a las peticiones de L1Mc.

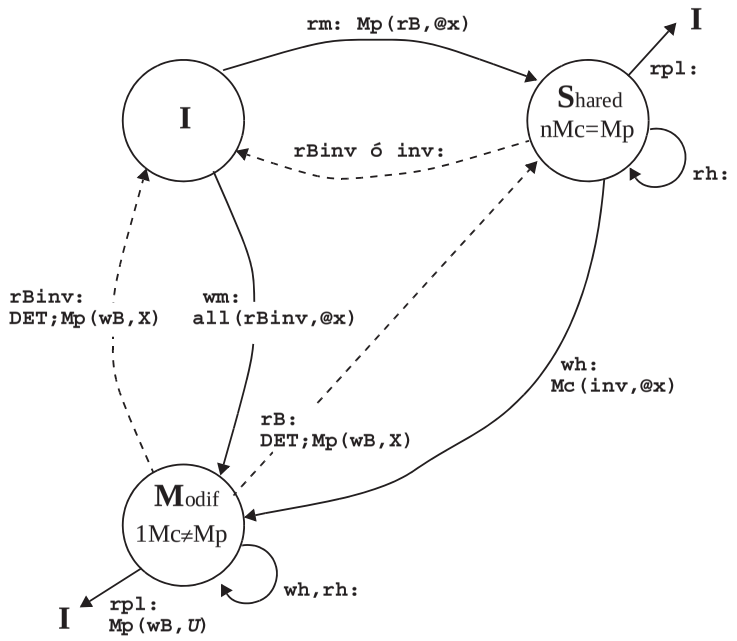
a) Diseñar el diagrama de estados de un protocolo snoopy basado en invalidación que cumpla las siguientes restricciones: caches con copy_back, tres estados, suministro de bloques limpios siempre desde Mp, 1 solo escritor no coherente con Mp, *n* copias de lectura coherentes con Mp.

b) Modificad el protocolo para forzar el suministro de bloques limpios entre Mc siempre que sea posible. Dibujad sólo lo que haya cambiado. ¿Qué problema aparece y cómo podría solucionarse?

Suponemos un multiprocesador de memoria compartida con memorias cache de correspondencia directa y escritura retardada. El tamaño de bloque es de dos palabras. El protocolo de coherencia tiene tres estados {I, S, M} y está basado en invalidación.

- Los eventos del procesador en relación con su memoria cache son:
  - `rh, wh, rm, wm`: aciertos en lec/escr y fallos en lec/escr.
  - `rpl:` reemplazo.
- Los comandos de bus son:
  - `inv, rBinv, wB`: invalidación, lect. bloque con invalid., escrit. de bloque.
  - `DET`: cuando un procesador encarga una lectura a la memoria y otro activa `DET`, el proc. lector libera al bus, la memoria abandona su lectura y queda libre, y el proc. lector reanuda la lectura detenida en cuanto se desactive `DET`.

    P/e, el comando compuesto: `DET;Mp(wB,X)` obliga la retirada del proc. con propiedad del bus y escribe el bloque `X` en Mp. En cuanto termina la escritura, se desactiva `DET` y el antiguo propietario del bus ya puede repetir el comando que le fue cancelado.
- El diagrama de estados de coherencia es el siguiente:

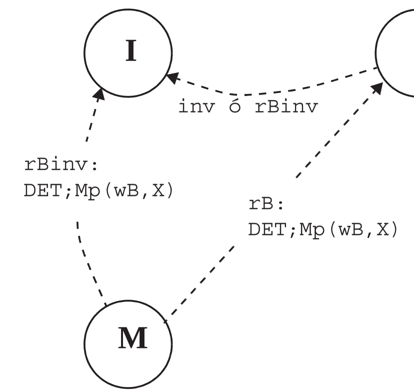


NOTACION: `evento: destino(comando,parametro)`

Estamos suponiendo, pues, un bus sencillo que no permite servicio de transferencia entre caches. Además, si un procesador necesita un bloque que está modificado por ahí, el mecanismo **DET** asegura la recepción desde Mp (previa escritura de la misma).

- En caso de reemplazo de bloque sucio, primero se despacha al bloque víctima y después se termina lo necesario

a) Etiquetad los inicios de los arcos punteados con los eventos remotos que los disparan.



b) Suponed que las variables **a** y **b** ocupan el bloque de Mp que denominamos **C**. Las variables **p** y **q** ocupan el **R**. Los dos bloques se corresponden con un contenedor único en Mc (conflicto) que llamamos **RC**.

C	**a**	**b**
R	**p**	**q**

Mostrad en la siguiente tabla la evolución de estados y comandos provocados por esta secuencia de referencias de los procesadores P1 y P2:

```
1    P1 lee b
2    P2 escribe q
3    P1 escribe b
4    P1 escribe a
5    P2 escribe b
6    P1 lee b
7    P1 escribe p
8    P2 escribe a
9    P1 escribe q
10   P1 lee p
```

Los números indican orden: no se lanza la referencia *i* hasta que no ha terminado la actividad de la *i-1*. Los contenedores RC de las dos caches están inicialmente vacíos.

	Proc. 1		Proc. 2				
Referencias	Estado: {I, S, M}	BloqueRC: {-, R, C}	Estado: {I, S, M}	BloqueRC: {-, R, C}	Comandos de BUS	Comentario muy pequeño	Ciclos
1 P1 lee **b**	S	C	I	-	Mp(rB,@C)	Mc1 falla, Mp suministra.	

TOTAL =

c) Suponemos los ciclos de procesado interno del procesador despreciables. Dados los siguientes costes temporales relacionados con el BUS:

- completar una lectura de Mp sin detención (con o sin invalidación): 10c.
- emitir una invalidación: 2c.
- reemplazo o escritura de bloque en Mp: 10c.
- reacción del K_{snoopy} a un comando de BUS que le concierne: 3c. (excepto `DET`)
- reacción del K_{snoopy} a un comando de BUS `DET`: 1c

Calculad los ciclos necesarios para la secuencia anterior, anotando en el campo "Ciclos" de la tabla anterior los ciclos empleados en cada fila. Si necesitáis algún coste mas, explicad en qué consiste y porqué le dais ese valor.

TOTAL=

d) Modificad el protocolo para reflejar el comportamiento de un BUS *mejor*, que permite la carga de un bloque en Mp y en Mc *a la vez*. Solamente dibujad lo que haya cambiado.

e) Recalculad los ciclos necesarios en este protocolo mejorado, reproduciendo un resumen de las filas de la tabla donde se gana y la explicación.

Suponemos un multiprocesador de memoria compartida con memorias cache privadas. El protocolo de coherencia tiene cuatro estados {M, E, S, I} y está basado en *actualización*. Se inventó para el multiprocesador DEC Firefly en el año 1988.

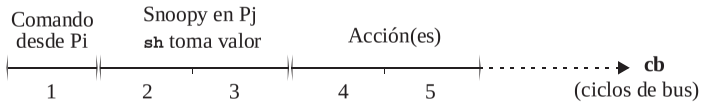
- Los eventos del procesador en relación con su memoria cache son:
  - `rh, wh, rm, wm`: aciertos en lec/escr y fallos en lec/escr.
  - `rpl`: reemplazo.
- Los comandos de coherencia son:
  - `rB`: lect. bloque.
  - `rB/wW`: lect. bloque *y escritura posterior de palabra.*
  - `wW`: escritura de palabra.
- El comando `wB` (escritura de bloque) no precisa actividad de observación (snoopy). Se produce en reemplazo o como respuesta a un comando de coherencia.
- La novedad de este protocolo es la existencia de una línea de bus especial (`sh`) que toma valor después de un comando de coherencia. Cuando el procesador Pi pone en el Bus un comando referido al bloque *x*, y ese bloque no está en ninguna cache:

  `sh = no` ($\overline{sh}$ en el diagrama)

  En cambio si el bloque *x* está presente en las caches de uno o mas Pj (j≠i):

  `sh = si` (`sh` en el diagrama)

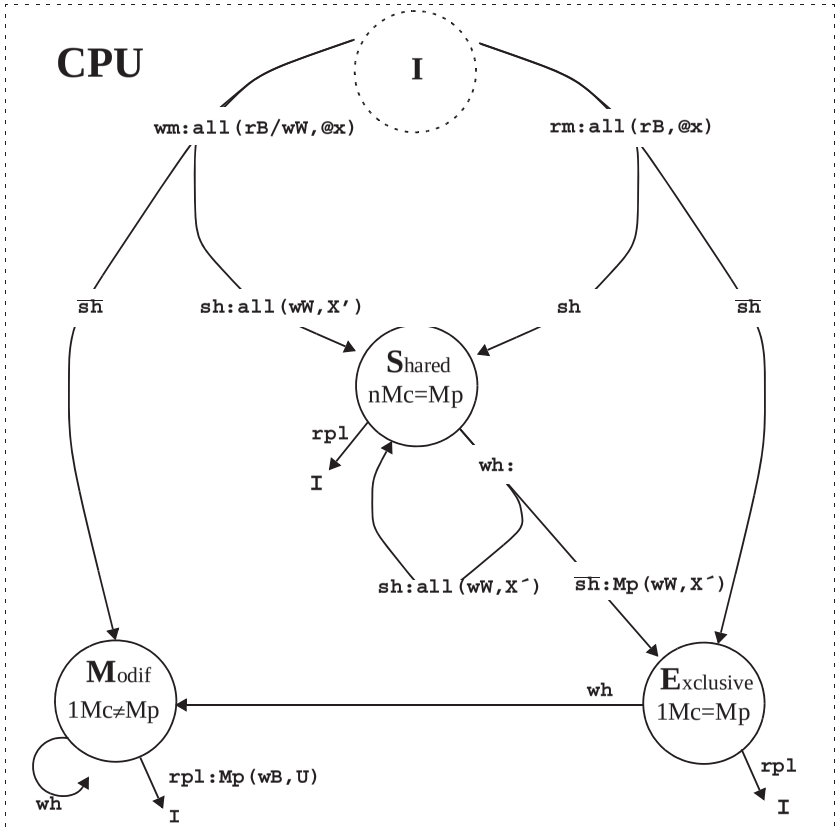
La actividad en los primeros tres ciclos de bus de un comando de coherencia es siempre la misma:

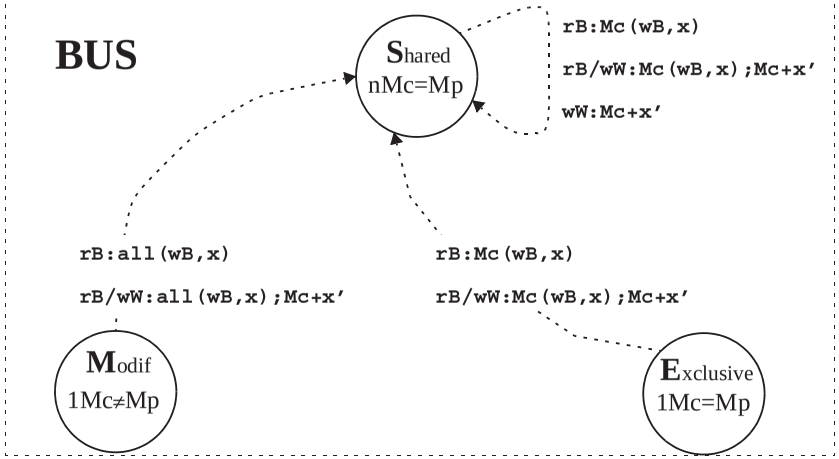


Veamos los diagramas de estado con las transiciones activadas por la CPU y las inducidas por comandos de Bus. Asumimos que:

- Los cambios de estado se realizan cuando ha finalizado la acción(es) desencadenada por el comando.
- En este protocolo, `I` es un estado transitorio, se incluye para no dibujar las transiciones de fallo entre todos los estados.
- En el caso (`m & sh`) el bloque es suministrado directamente por una Mc, no por Mp. Para ello en el tercer ciclo de bus se arbitra entre las Mc que poseen el bloque.
- En el caso (`m & sh`) el bloque viene desde Mp.

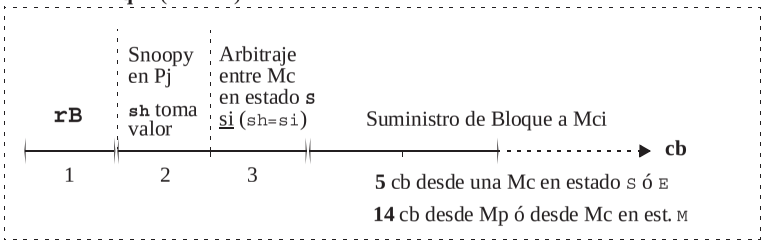
NOTACION: x, u: *bloque* referenciado y reemplazado. Con símbolo prima: *palabra* (e.g. x'). Con mayúsculas: palabra o bloque + *dirección*. Significado de **`all`**: `Mc&Mp`. Significado de **`Mc+x'`**: copia de la palabra x' en cache.



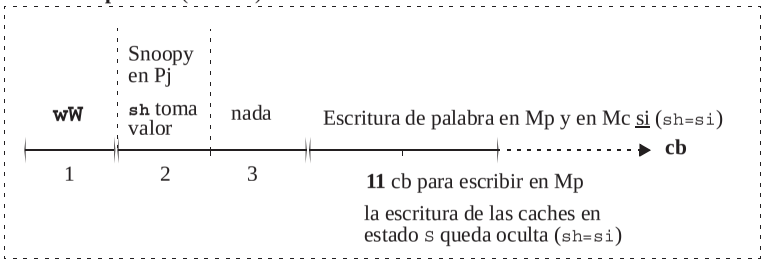


- Para responder las preguntas hay que entender la temporización de los comandos de Coherencia. Suponemos comandos procedentes del proc. Pi.

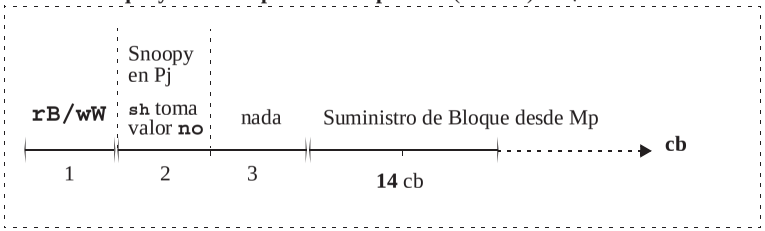
Lectura de Bloque (desde `I`): **`rB`**



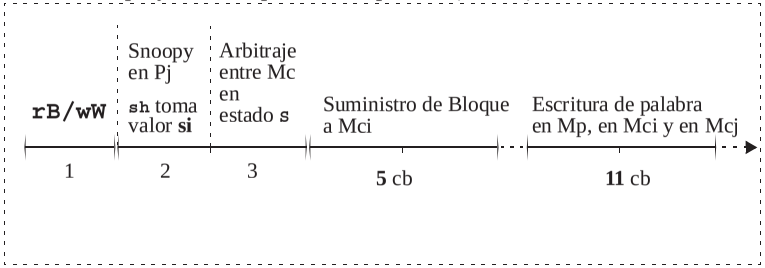
Escritura de palabra (desde `S`): **`wW`**



Lectura de bloque y escritura posterior de palabra (desde `I`): **`rB/wW & sh`**



Lectura de bloque y escritura posterior de palabra (desde `I`): **`rB/wW & sh`**



a) Una demanda de *bloque* del procesador Pi puede ser satisfecha directamente desde una Mcj, sin necesidad de leer en Mp y quizás con necesidad de arbitraje. Listar en la siguiente tabla las situaciones en que esto es posible (Mci demanda, Mcj suministra):

El nº de filas de la tabla no tiene relación con el nº de soluciones...

Evento en Mci	Comando coherencia emitido por Mci	Estado Inicial bloque en Mci {`NoEstá, M, E, S`}	Estado Inicial bloque en Mcj {`M, E, S`}	Necesidad de arbitrar {si, no}

b) Listar las situaciones de suministro de *palabras* entre Mc (Mci suministra, Mcj recibe):

El nº de filas de la tabla no tiene relación con el nº de soluciones...

Evento en Mci	Comando coherencia emitido por Mci	Estado Inicial bloque en Mci {`NoEstá,M, E, S`}	Estado Inicial bloque en Mcj {`M, E, S`}	Necesidad de arbitrar {si, no}

c) ¿Bajo qué combinación de eventos y estados este protocolo realiza una escritura retardada (copy-back)?

d) ¿Bajo qué combinación de eventos y estados este protocolo realiza una escritura inmediata (write-through)?

En los apartados siguientes se trata de calcular los *ciclos de bus* que se gastan en ciertas secuencias de accesos. Los bloques que intervienen en las secuencias:

- No están inicialmente en ninguna cache, salvo que se diga lo contrario.
- No se reemplazan, salvo que se diga lo contrario.

e) Un procesador hace*n* accesos a una variable no compartida (si es necesario tenerlo en cuenta, suponed lecturas y escrituras alternadas).

cb TOTAL

f) Un procesador hace*n* lecturas de una variable compartida y presente en otras caches en estado **S**. Durante las *n* lecturas no existe ninguna escritura.

cb TOTAL

g) Sobre la misma variable compartida (inicialmente en ninguna cache): un conjunto de (*n* lecturas, seguidas de 1 escritura) del Proc. 1, después un conjunto de (*n* lecturas, seguidas de 1 escritura) del Proc. 2. Finalmente expulsión de bloque en el Proc. 2.

cb TOTAL

h) Compartición secuencial de una variable compartida entre procesadores:

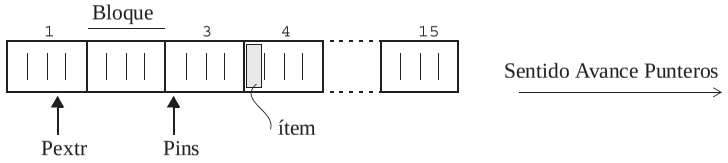
```
S1: R procesadores leen la misma variable compartida
S2: un proc. escribe (el i)
S3: leen los restantes R-1 proc.
S4: un proc. escribe (el j)
S5: leen los restantes R-1 proc.
    goto S2
```

Calcular los ciclos de Bus para *n* iteraciones del bucle

```
S1:

S2:

S3:

S4:

S5:
```

cb TOTAL

i) Eficiencia en el acceso a un buffer circular de 15 bloques, compartido entre un proceso Productor y uno Consumidor. El tamaño de bloque es 4 veces el del ítem que se escribe/lee. Suponemos una secuencia de *n* escrituras y *n* lecturas. *n* es múltiplo de 4. El buffer



cabe en cache. *n* es grande con respecto al tamaño del buffer.

i1) SITUACION 1: El Productor siempre va por delante del Consumidor al menos en un bloque

cb TOTAL

i2) ¿ por qué secuencia de estados pasa cada bloque del buffer?

• En la cache del Productor:

• En la cache del Consumidor:

i3) SITUACION 2: El Consumidor siempre va pisándole los talones al Productor, es decir, en el tiempo hay un entrelazado perfecto: escribe Buf[i], lee Buf[i], escribe Buf[i+1], lee Buf[i+1], etc.

cb TOTAL

i4) ¿ por qué secuencia de estados pasa cada bloque del buffer?

• En la cache del Productor:

• En la cache del Consumidor:

j) OPTATIVO. Diseñar el comportamiento de la instrucción `Test&Set`:

```
T&S rd, @   ; lógicamente indivisible:
            ;   1° rd= M(@)
            ;   2° M(@)= 1
```

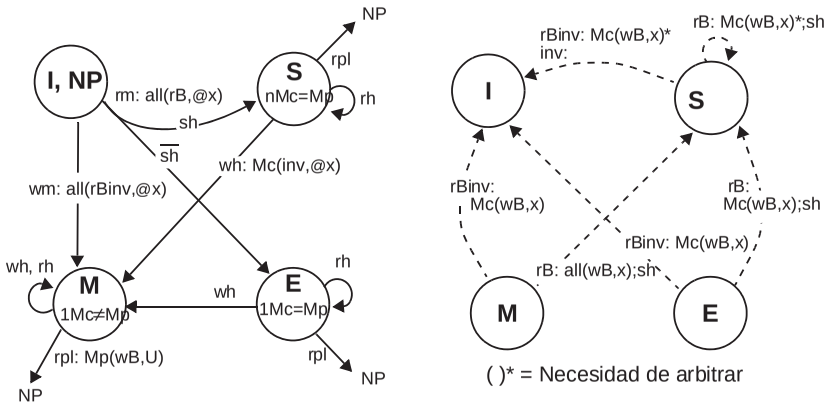
Tened en cuenta en primer lugar la corrección, y en segundo lugar cualquier optimización que reduzca el tráfico (por ejemplo, mientras *n* procesadores esperan en un spin-lock a que otro procesador abandone su sección crítica).

Para caracterizar un programa paralelo ejecutándose en un multi de memoria compartida, basado en bus y con coherencia por invalidación, se ha construido un simulador con las siguientes características:

- 16 procesadores con Mc privadas de datos (1 MB, Bsize = 64B, Asoc. 4, LRU).
- protocolo de coherencia snoopy, tipo Illinois, con línea de compartición (`sh`).
- Memorias y Bus ideales en los siguientes aspectos:
  - *Latencia de cualquier servicio 1 ciclo*. Un ejemplo: detectar fallo en una Mc, seguido de expulsión de bloque sucio y seguido de suministro desde Mp = 1c. Otro ejemplo: Acierto en escritura a una variable compartida, seguido de invalidación en el Bus = 1c.
  - *Contención del Bus nula*. Ejemplo: Dos o más peticiones de bloques diferentes desde dos o más Mc se sirven en el mismo ciclo.
- Procesador ideal. Toda instrucción, sencilla o compleja, se ejecuta en un ciclo. Tiempo de ciclo = 5 ns. Suponemos un suministro de instrucciones ideal, que ni siquiera genera tráfico de Bus.

Por tanto, la simulación no sirve para conocer el tiempo real de ejecución, pero sí para aproximarnos al comportamiento de compartición, a las tasas de fallos o tráficos, etc.

Con estas hipótesis la memoria nunca detiene a los procesadores, y la única razón para que no tengan Utilización = 1, es el tiempo perdido por no estar balanceado el trabajo entre procesadores (los *spin-locks* cuentan como instrucciones útiles).



Protocolo MSI Univ. Illinois

El suministro "cache to cache" desde el estado **S** se ha desactivado en los experimentos de este problema

Solamente a efectos de simulación, para tener una idea de cuantos bloques invalidados por coherencia vuelven a ser cargados, a) reservamos el estado I para identificar a los bloques invalidados solamente por coherencia, b) introducimos el estado NP (No Presente), que es el estado inicial de los bloques en la Mc vacía y al que se cae transitoriamente después de un reemplazo y, c) guardamos la marca de los bloques en estado I, para poder contar las transiciones I → {NP, E, S, M}. Veamos un ejemplo con un conjunto cualquiera de una Mc:

nº bloque	ESTADO	MARCA	CONTENIDO
1	M	0xFFF	- - -
2	E	0xAB5	- - -
3	NP	n.i.	n.i.
4	I	0x123	n.i.

Una lectura al bloque de marca 0x690 provoca un fallo y una transición NP → {E ó S}, según el valor de `sh`. Pero una lectura del bloque 0x123, además de fallo, provoca una transición I → {E ó S}, según el valor de `sh`.

Datos de simulación del programa que vamos a caracterizar[1]:

nº de instrucciones	282,3 x10^6	suma total, agregando los números para los 16 procesadores
nº de lecturas	81,6 x10^6	
nº de escrituras	18,53 x10^6	
nº de ciclos	17,8 x10^6	inicio → fin de la ejecución paralela

Número de transiciones entre estados por cada 100 referencias a las memorias cache de datos:

Aplicación	de → a	NP	I	E	S	M
OCEAN	NP			0,12	0,1	0,17
	I	0,07		0	0,18	0,0002
	E	0,02	0	1,43	0,003	0,1
	S	0,05	0,25		14,2	0,22
	M	0,27	0,0002		0,23	83,5

Estas cuentas se han hecho sumando en una o más casillas en cada referencia, ya que como sabemos, una referencia puede provocar:

- una transición, p.e.
  `rh` en E (E → E) 1,43%
- dos transiciones, p.e.
  `rm` a un bloque ausente en las 16 caches con expulsión de víctima en estadoM
  (1º M → NP, 2º NP → {E, S }) 0,27% + (0,12% + 0,1%)
- mas de dos, p.e.
  `wh` a un bloque replicado en *cuatro* caches
  S (S → M y *tres* S → I) 0,22% + parte de 0,25%

1. OCEAN de Splash-2, es un programa que calcula difusión de contaminantes en el mar.

Se pide lo siguiente:

a) Utilización media de los procesadores.

U =

b) Breve comentario sobre el comportamiento de la aplicación, seleccionando para ello las cuatro transiciones más frecuentes y las dos menos frecuentes, indicando que actividad implican:

c) Tasa de fallos agregada de las 16 caches
(suma de fallos en cualquier cache/ 100 referencias).

m =

d) Número total de transacciones de Bus por referencias a datos, desglosadas en las categorías: carga de bloque en Mc, invalidación pura y reemplazo. Transiciones implicadas.

carga de bloque en una Mc	
invalidación pura	
reemplazo	

e) Calculad el número total de bytes por referencias a datos que han circulado por el Bus, contando de la siguiente forma: un movimiento de bloque genera un tráfico de 70B (6B de dirección/comando + 64B de tamaño de bloque), mientras que `inv` genera un tráfico de 6B (dirección/comando).

f) Calculad el ancho de banda *por procesador*, por referencias a datos, en MB/s que el tráfico del apartado anterior requiere

g) Calculad qué fracción del ancho de banda anterior corresponde a tráfico de recarga de bloques previamente invalidados

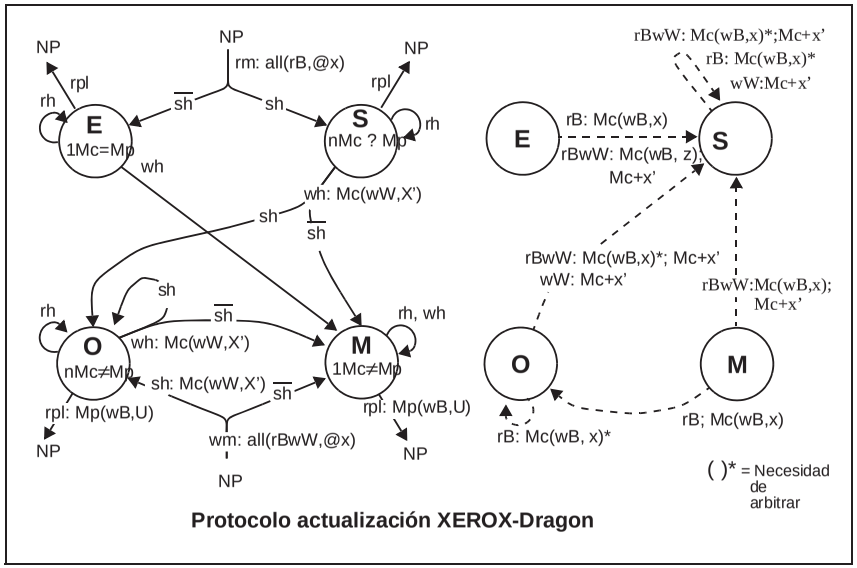
Para caracterizar un programa paralelo ejecutándose en un multi de memoria compartida, basado en bus, se ha construido un simulador con las siguientes características:

- 16 procesadores con Mc privadas de datos (1 MB, Bsize = 64B, Asoc. 4, LRU)
- protocolo de coherencia snoopy con actualización, con línea de compartición (`sh`)
- Memorias y Bus ideales en los siguientes aspectos:
  - *Latencia de cualquier servicio 1 ciclo*. Un ejemplo: detectar fallo en una Mc, seguido de expulsión de bloque sucio y seguido de suministro desde Mp = 1c. Otro ejemplo: Acierto en escritura a una variable compartida, seguido de actualización en el Bus = 1c.
  - *Contención del Bus nula*. Ejemplo: Dos o más peticiones de bloques diferentes desde dos o más Mc se sirven en el mismo ciclo.
- Procesador ideal. Toda instrucción, sencilla o compleja, se ejecuta en un ciclo. Tiempo de ciclo = 5 ns. Suponemos un suministro de instrucciones ideal, que ni siquiera genera tráfico de Bus.

Por tanto, la simulación no sirve para conocer el tiempo real de ejecución, pero sí para aproximarnos al comportamiento de compartición, a las tasas de fallos o tráficos, etc.

Notad que con estas hipótesis la memoria nunca detiene a los procesadores, y la única razón para que no tengan Utilización = 1, es el tiempo perdido en sincronizaciones (espera para entrar a sección crítica, espera en barriers, etc).

El protocolo de actualización tipo Xerox-Dragon, es el siguiente:



Protocolo actualización XEROX-Dragon

NP es un estado transitorio, que se da al inicializar la Mc, pero que en régimen normal sólo sirve para indicar que un bloque víctima es reemplazado (MOES → NP) y luego se recibe al bloque que ha fallado, *x* (NP → MOES)

Veamos un ejemplo de conjunto que sólo ha recibido referencias a tres bloques diferentes:

nº bloque	ESTADO	MARCA	CONTENIDO
1	M	0xFFF	- - -
2	E	0xAB5	- - -
3	NP	n.i.	n.i.
4	O	0x123	- - -

Una lectura al bloque de marca 0x690 provoca un fallo y una transición NP → {E ó S}, según el valor de `sh`. Una posterior escritura al bloque de marca 0x976 provoca un transitorio del bloque víctima hacia NP y luego otra transición NP → {O ó M}, según el valor de `sh`.

Datos de simulación del programa que vamos a caracterizar[1]:

nº de instrucciones	282,3 x10^6	suma total, agregando los números para los 16 procesadores
nº de lecturas	81,6 x10^6	
nº de escrituras	18,53 x10^6	
nº de ciclos	17,8 x10^6	desde *inicio* a *fin* de ejecución

Número de transiciones entre estados por cada 100 referencias a la memoria cache de datos:

Aplicación	de → a	NP	S	E	O	M
OCEAN	NP		0,1	0,14	0,01	0,2
	S	0,1	11,0		0,01	0,00001
	E	0,025	0,0015	1,72		0,1
	O	0,03	0,0005		11,45	0,04
	M	0,3	0		0,05	77,3

Además, el 11.45 de O → O se descompone en 9.81 por `rh` y 1.64 por `wh` ∧ `sh`

Estas cuentas se han hecho sumando en una o más casillas en cada referencia, ya que como sabemos, una referencia puede provocar en el conjunto de la 16 caches:

- una transición en 1 Mc, p.e. `rh` en E (E → E)
- dos transiciones en 1 Mc, p.e `rm` con víctima en M
  1º M → NP, 2º NP → {E, S}
- una transición en varias Mc, p.e. `wh` en S
  S → M y múltiples S → I

sin embargo, en (S → S y O → O) solamente se han contado las transiciones por acierto local, no las inducidas por los comandos `rB` ó `wW`.

1. OCEAN de Splash-2, es una aplicación para el cálculo de la difusión de contaminantes en el mar.

a) Utilización media de los procesadores.

U =

b) Breve comentario sobre el comportamiento de la aplicación, seleccionando para ello las tres transiciones más frecuentes y las dos menos frecuentes, indicando que actividad implican:

c) Número total de transacciones de Bus por referencias a datos, desglosadas en las categorías de la tabla. Transiciones implicadas.

carga de bloque en una Mc	$NP \rightarrow \{S, E, O, M\}$ 450 585 trans.
actualización de palabra	$S \rightarrow \{O, M\}$; $O \rightarrow O$; wh; $NP \rightarrow O$. 1 662 168 trans.
reemplazo	$\{O, M\} \rightarrow NP$. 330 429 trans.

d) Calculad el número total de bytes por referencias a datos que han circulado por el Bus, contando de la siguiente forma: un movimiento de bloque genera un tráfico de 70B (6B de dirección/comando + 64B de bloque), mientras que `wW` genera un tráfico de 14B (6 +8) y finalmente, `rBwW` genera un tráfico de 84B (70 + 14) si el bloque está compartido.

$$74{,}3\ MB = 77{,}91 \times 10^6 B$$

e) Calculad el ancho de banda *por procesador*, por referencias a datos, en MB/s que el tráfico del apartado anterior requiere

$$52{,}198\ MB/s$$

f) Calculad la tasa global de fallos, definida como:

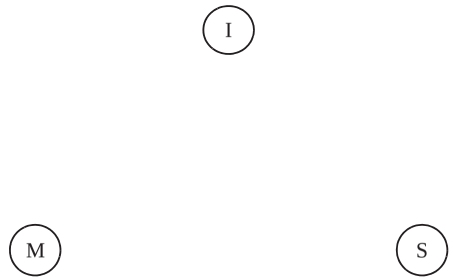
$$m_g = \frac{\Sigma \text{ fallos en las 16 caches}}{\Sigma \text{ referencias en los 16 proc.}} =$$

Diseñar un protocolo de coherencia basado en invalidación con las siguientes propiedades:

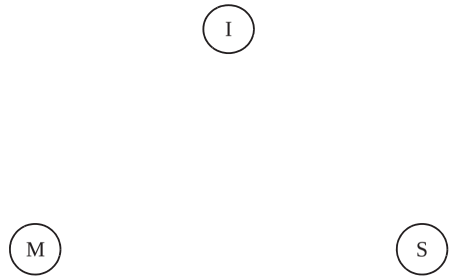
- Memorias cache con escritura retardada (*copy-back*)
- Estados M, S e I
- Servicio de copias entre Mc en todos los casos posibles y con el menor uso de Mp. Si varias Mc pueden dar un bloque, existe lo necesario para seleccionar a un proveedor único. Los comandos que precisen de este arbitraje se indicarán con asterisco (*).

Se pide lo siguiente:

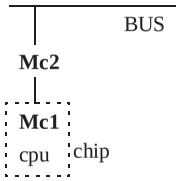
a) Transiciones iniciadas por el *procesador* y acciones resultantes



b) Transiciones iniciadas por el *bus* y acciones resultantes



Vamos a considerar ahora dos niveles de Mc en cada procesador. El segundo nivel, Mc2, es similar al estudiado en los apartados anteriores: escr. retardada, 3 estados básicos, etc.



El primer nivel, Mc1, funciona con escritura inmediata sobre Mc2 (*Write-through*), por tanto sus bloques sólo tienen estado **V**álido o **I**nválido.

El objetivo de Mc1 es servir al procesador y el de Mc2 servir a Mc1, molestando lo menos posible con la gestión de la coherencia. Un fallo simultáneo en Mc1 y Mc2 carga el bloque en ambas caches

Mc2 es un superconjunto de Mc1 (inclusión de Mc1 en Mc2) a base de tener el mismo número de conjuntos, el mismo tamaño de bloque, mayor asociatividad y *nunca* reemplazar un bloque presente en Mc1. Para conseguir esto último, cada bloque en Mc2 tiene un *bit de presencia* que indica si existe o no copia (¡coherente!) en Mc1:

- bit presencia = **P**, bloque presente en Mc1
- bit presencia = **NP**, bloque no presente en Mc1
- la gestión de este bit en Mc2 precisa de mensajes `rpl` desde Mc1

c) Significado de cada uno de los nuevos estados que aparecen en Mc2. Podéis combinar notación del tipo:

$$1(Mc1=Mc2) \neq Mp$$

con explicación en palabras

Estados	Significado
M ∧ P	
M ∧ NP	
S ∧ P	
S ∧ NP	

d) Transiciones en Mc1 iniciadas por el procesador y comandos hacia Mc2. Unicamente podéis usar los comandos

`rB, rBwW, wW y rpl`

Estos comandos nacen en Mc1 y atacan a Mc2; utilizamos los mismos nombres que para los comandos de Bus por simplificar la notación

e) Transiciones en Mc2 iniciadas por comandos de Mc1 que *aciertan.* en Mc2. Añadid donde convenga:

`Mc2 + x'`

Notación: Mc1 y Mc2 = mis caches; Mc = resto de las caches (otros procesadores)

f) Transiciones en Mc2 inducidas por comandos de Mc1 que *fallan* en Mc2. Describimos precisamente el contenedor del bloque víctima (que para mantener la inclusión se habrá escogido entre los que tengan estado I o NP). Añadid donde convenga:

`Mc2 + x'`

g) Transiciones en Mc2 inducidas por el bus y acciones resultantes, tanto hacia el propio bus como hacia Mc1. Recordad el asterisco (*).

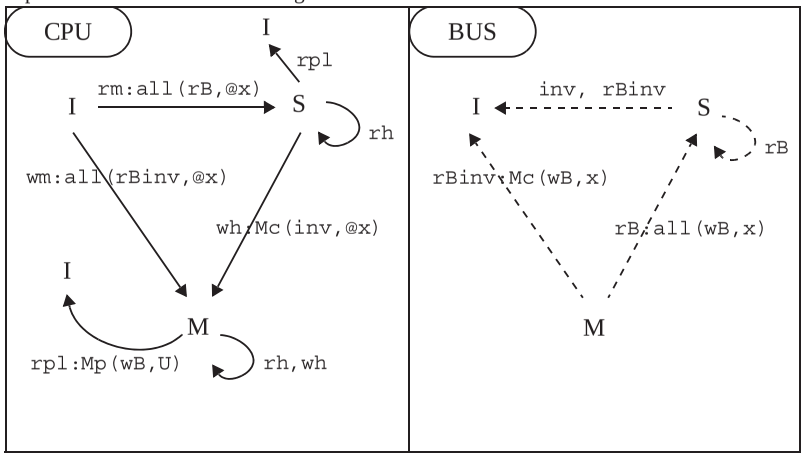
21) Primitiva de sincronización FETCH&INC y coherencia. *Resuelto* Feb. 98

En este problema se tratan dos aspectos de un multi de memoria compartida basado en bus: sincronización y protocolo de coherencia.

La coherencia se mantiene con un protocolo sencillo de tres estados {M, S, I} e invalidación y la única instrucción de sincronización es Fetch&Inc.

Si en algún momento es necesario asumir algo sobre el bus, suponed que funciona en conmutación de circuitos. El modelo de consistencia es secuencial, y por tanto un procesador no avanza hasta que sus invalidaciones son completadas en todas la Mc implicadas.

El protocolo de coherencia es el siguiente:



La instrucción de sincronización F&I funciona así:

- explicación alto nivel.:

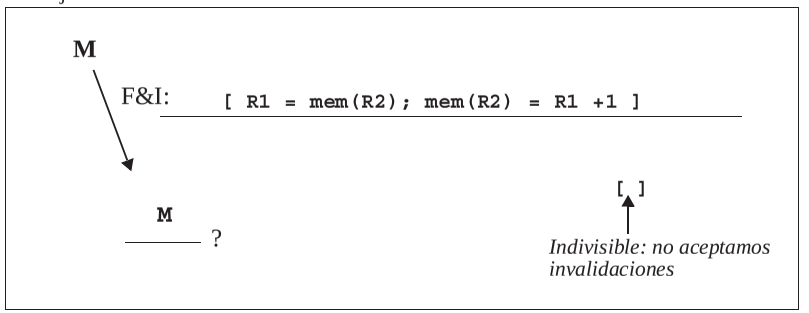
```
F&I(cont) -->
[   tmp = cont;
    cont = tmp+1;
    return tmp      ]            /* indivisible ! */
```

- comportamiento a nivel máquina:

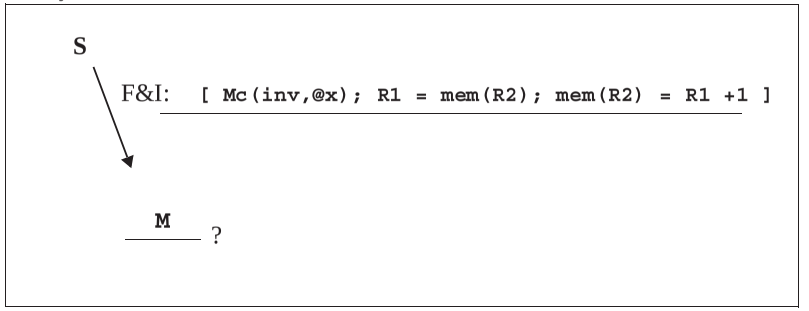
```
F&I R1,(R2) -->
[   R1 = mem(R2);
    mem(R2) = R1+1  ]            /* indivisible ! */
```

A continuación explicad como puede implementarse F&I respondiendo a los tres apartados siguientes. Añadid lo mínimo o nada al protocolo original (comandos, estados). Suponed que la variable `cont ( ó mem(R2) )` pertenece al bloque ***x***.

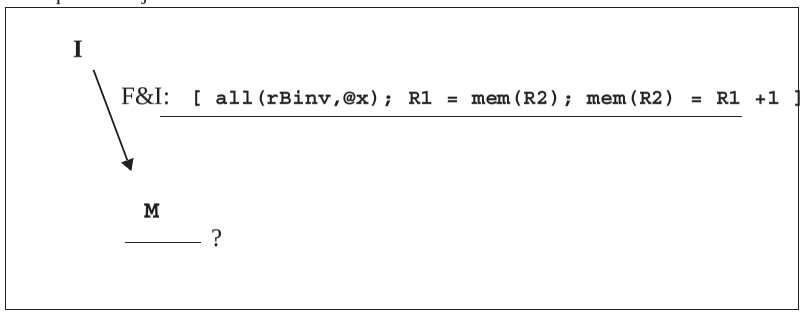
a) Transición y actividades de F&I si *x* está en estado M. Poned en evidencia lo que deba ejecutarse de forma indivisible.



b) Transición y actividades de F&I si *x* está en estado S. Poned en evidencia lo que deba ejecutarse de forma indivisible.



c) Transición y comando de F&I si *x* está en estado I, o sea, *no* está. Poned en evidencia lo que deba ejecutarse de forma indivisible.



d) Escribid la solución mas sencilla basada en espera activa para entrar en exclusión a una sección crítica (SC), suponiendo que `R2` está cargado con la dirección de la variable `cont`. Supón un ensamblador con saltos no retardados.

- Significado que se dará a la variable `cont`

  cont = 0 ---> Sección crítica libre

  cont ≠ 0 ---> Sección crítica ocupada

- `init_mutex1(cont)`

```
{
        st      0(R2), R0       ; 1 vez solamente
                                ; antes de la contienda
}
```

- `begin_mutex1(cont)`

```
{
esp:    F&I     R1, (R2)
        bnz     esp, R1
}
```

- `end_mutex1(cont)`

```
{
        st      0(R2), R0
}
```

---

e) Si un procesador permanece dentro de la SC mucho tiempo, ¿se produce algún problema?

El resto de procesadores están dentro de `begin_mutex1()`*, y por tanto incrementan continuamente la variable* `cont`*. En un caso improbable la variable podría desbordarse, y al pasar por cero, permitir la entrada a la sección crítica de forma indebida.*

f) Supongamos que 4 procesadores ejecutan `begin_mutex1(cont)`. Uno entra y durante su permanencia en la SC, cada uno de los que quedan por entrar ejecuta 10 instrucciones `F&I`, independientemente de pelearse con 2 o 1 procesadores. Inicialmente el bloque *x* no está en ninguna Mc.

- Determinar el valor de `cont` justo antes de que cada procesador ejecute `end_mutex1(cont)`.

`cont=` *31* `cont=` *21* `cont=` *11* `cont=` *1*

- Determinar el número total de comandos de invalidación por ejecución de `F&I` debido a la entrada de los 4 procesadores a la SC. No distingas entre `inv` o `rBinv`. Supón que los accesos se entrelazan.

nº de inv por F&I = nº de ejecuciones de F&I =

31 + 21 + 2 + 1 = 55

g) Programa `begin_mutex2()`, que consigue disminuir el número de invalidaciones a base de realizar la espera activa sobre una copia en estado S de la variable `cont`.

- `begin_mutex2(cont)`

```
{
esp:    ld      R1, (R2)        ; si cont ¦ 0 sólo genero tráfico de lectura
        bnz     R1, esp         ; 1 vez, y luego realizo lecturas locales
        F&I     R1, (R2)
        bnz     esp, R1
}
```

- Utilizando `begin_mutex2()`, determina el número de invalidaciones por ejecución de `F&I` debido a la entrada de los 4 procesadores a la SC. No distingas entre `inv` o `rBinv`. Inicialmente el bloque *x* no está en ninguna Mc. Considera el mejor de los casos (menor número de invalidaciones) y el peor de los casos.

mejor caso

Cada vez que hay contienda, uno llega con gran ventaja y el resto lee `cont=1`.

Se ejecutan 4 F&I:

4 `inv` *de S a M*

peor caso

el `ld` *de* `begin_mutex2()` *es ejecutado "a la vez" por los cuatro procesadores. Se ejecutan 4 F&I para entrar por primera vez a la Sección Crítica, luego 3, luego 2, luego 1:*

4 + 3 + 2 + 1 invalid

(4 `inv` *y 6* `rBinv`*)*

Puede realizarse otra programación con menor actividad de coherencia. El método que consideraremos se puede denominar llave con "turno" ya que utiliza el concepto de numerar las peticiones y concederlas utilizando tal numeración. Las estructuras de datos son 2 variables globales denominadas `orden y turno`. La primera se utiliza para obtener la numeración de las peticiones de acceso a la SC y la segunda se utiliza para indicar a quien le toca ejecutar la SC.

h) Utiliza F&I para programar `init_mutex3, begin_mutex3, y end_mutex3`, de acuerdo con esta idea. Supón que `Ro y Rt` apuntan respectivamente a las variables `orden y turno` (que están en bloques de cache diferentes).

- `init_mutex3(Ro,Rt)`

```
{
        st      (Ro), R0        ; orden = 0
        st      (Rt), R0        ; turno = 0
}
```

- `begin_mutex3(Ro,Rt)`

```
{
        F&I     R1, (Ro)        ; [ mi_turno = orden; orden++ ]
esp:    LD      R2, (Rt)
        seq     R3, R1, R2      ; mi turno = turno?
        bnz     R3, esp
}
```

- `end_mutex3(Ro,Rt)`

```
{                               ; turno++ sin exclusión mutua !!
        ld      R1, (Rt)
        addi    R1, R1, #1
        st      (Rt), R1
}
```

- Determina el número de invalidaciones y cargas del bloque `turno` por ejecución de `F&I` debido a la entrada de los 4 procesadores a la SC. Determinar también las cargas del bloque donde reside *turno*. Inicialmente las variables *orden y turno* no están en ninguna Mc. No distingas entre `inv o rBinv`. Considera casos si es necesario.

4 F&I para obtener `mi_turno =` *4* `rBinv` *de I a M*

lecturas del bloque turno = ?

i) Todavía es posible una solución con menor tráfico de coherencia, a costa de ocupar más memoria, expandiendo la variable escalar *turno* en un vector.

Repite todo el desarrollo del apartado anterior.

! Atención con la compartición falsa (*false sharing*) !

22) Coherencia basada en directorio + Hipercubo — Jun. 07

Sea un multiprocesador de memoria compartida distribuida, construido mediante nodos conectados por un Hipercubo de grado 6 (6 enlaces/nodo). En cada nodo existe un solo procesador, una cache y 1 GB de memoria principal.

La coherencia se basa en Directorios asociados a la memoria principal, en base a un tamaño de bloque de 128B. El protocolo implementado tiene las siguientes características:

- Invalidación
- Carga de bloque en caso de fallo en escritura + copy-back
- Control de presencia con Full-Map, implementado mediante un vector binario
- Estado en Directorio: (M, S, A) + Estado en Cache: (M, S, I).

Responded a las siguientes cuestiones:

a) Tamaño del Directorio de un nodo en GBytes y explicación mínima.

b) Tamaño agregado del Directorio en GBytes (todos los nodos) y explicación mínima.

c) Un cierto bloque x se encuentra replicado en las caches de los nodos 0, 1 y 8.

- los nodos se numeran a partir de 0
- el nodo Hogar (Home) de x es el nodo 2.

Supongamos que el nodo 4 experimenta un fallo en escritura. Describid en la siguiente tabla la secuencia de mensajes de coherencia que se producen para gestionar este fallo.

El tiempo crece hacia abajo

Mensajes de coherencia			
Nodo Emisor	Nodo Receptor	Comando	¿ Cuántos enlaces hay que atravesar?

d) ¿ Qué información ha quedado del bloque x en el nodo 2?

e) ¿ En qué estado queda el bloque x en las caches de los nodos 0, 1, 4, 8?

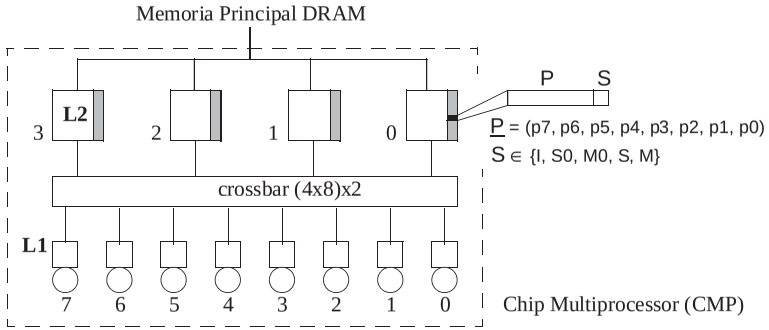
Nodo 0

Nodo 1

Nodo 4

Nodo 8

En la figura se muestra un multiprocesador y su jerarquía de memoria. Hay dos niveles de memoria cache, el primero privado para cada procesador (L1 x 8 proc.) y el segundo compartido entre los ocho procesadores (L2 x 4 bancos). En los dos niveles el tamaño de bloque es igual. Se mantiene inclusión estricta (L2 es un superconjunto de los bloques contenidos en todas las L1) impidiendo selectivamente el reemplazo en L2, ver apartado `b1`.



Memorias cache de primer nivel (L1):

- escritura a través (*write-through)*
- *no* cargan bloque en caso de fallo en escritura (*no fetch on write miss*)
- comunican la identidad de los bloques reemplazados a L2.

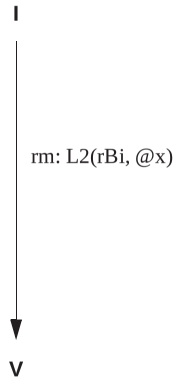
La memoria cache de segundo nivel (L2) está partida en cuatro bancos entrelazados que gestionan la coherencia en todas las caches L1 **por invalidación**. Todos lo bancos tienen:

- escritura retardada (*copy-back*)
- *sí* cargan bloque en caso de fallo en escritura (*fetch on write miss*)
- directorio con 8 bits de presencia de bloque en L1 (vector binario P) y tres bits de estado por bloque
- los estados de coherencia son:
  - **I**
  - **S0** → **L2=Mp; P=0**
  - **M0** → **L2≠Mp; P=0**
  - **S** → **nL1=L2=Mp; n>0**
  - **M** → **(1L1=L2)≠Mp** → suponemos que es la cache **L1*j***
- no se permite el suministro de bloques limpios entre caches L1.

A continuación vamos a definir el comportamiento de las caches de primer y segundo nivel, asumiendo conexión directa con memoria, **sin preocuparnos** de la posible interconexión de este CMP con otros CMPs.

`a)` Transiciones y acciones por interacción procesador-cache L1.

Supón que estás describiendo la cache número **i**, **L1i**. Etiqueta los comandos siguiendo el ejemplo.



`b)` Transiciones y acciones en L2 por comandos que vienen de L1:

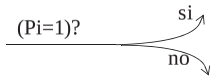
`b1)` **L1i** envía **rBi** a un banco de L2. ¡ No te olvides de los bits de presencia! Considera por tanto aciertos, fallos y reemplazos en L2 provocadas por **rBi**. Respecto a los reemplazos ten en cuenta que L2 es responsable de garantizar la inclusión.

M **S**

I

M0 **S0**

b2) **L1i** envía un **wWi**. Si hace falta utiliza la operación L2+x' y la transición comparar/bifurcar:



M **S**

I

M0 **S0**

A la vista de lo que has hecho, si es necesario, completa el diagrama del apartado (a)

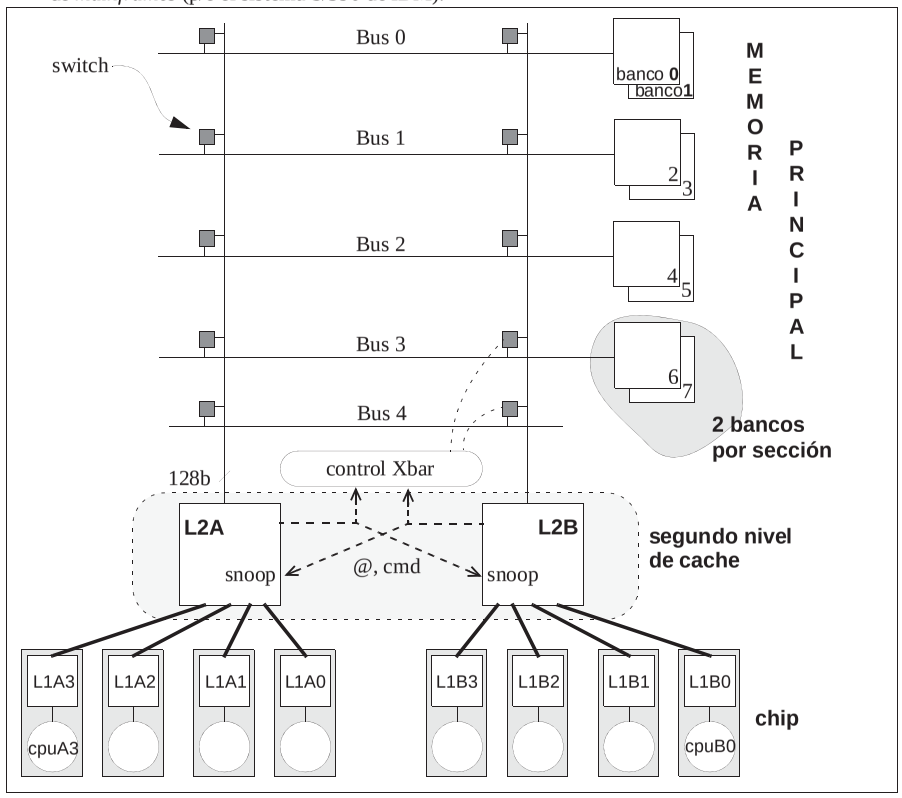
b3) Resto de transiciones y acciones.

M **S**

I

M0 **S0**

En la figura se muestra un multiprocesador con 8 procesadores, representativo de la gama alta de *mainframes* (p/e el sistema S/390 de IBM).



Cada uno de los 8 procesadores tiene una cache *on-chip*: L1A3, L1A2, ... L1B1, L1B0.

Las caches L2A y L2B forman el segundo nivel de cache. Tienen cuatro puertos hacia el nivel L1. Una red de interconexión *crossbar* une L2A y L2B con cuatro secciones de memoria principal. Cada sección está formada por dos bancos. El tamaño de bloque en los dos niveles es 128 bytes, y bloques de cache consecutivos corresponden a bancos de memoria consecutivos. Para mantener la coherencia se usa un protocolo de **invalidación**. En concreto:

- **Nivel 1 (8 caches: L1A3, L1A2, ... y L1B3, L1B2, ...)**

  Escritura *inmediata* en L2 (*write-through*), carga de bloque en caso de fallo en escritura. Estados de coherencia: **M**, **S**, **I**

  - **M** copia sucia y única en todo el Nivel 1.
    Obligatoriamente el Nivel 2 padre tiene otra copia igual (1L1 = 1L2 ≠ Mp).
  - **S** copia limpia y posiblemente replicada (nL1 = mL2 = Mp; n=1:8; m=1:2)

- **Nivel 2 (L2A y L2B)**

  Escritura *retardada* en memoria pincipal (*copy-back*), carga de bloque en caso de fallo en escritura. Mantiene la coherencia mediante un puerto auxiliar que observa

los comandos y direcciones de la cache L2 hermana (*snoop* en la figura).

Está diseñado para contener siempre todos los bloques presentes en sus cuatro caches L1 y algunos más. Para ello, L2 nunca reemplaza un bloque presente en L1. Conseguir esta *inclusión* de contenidos se logra mediante:

- Un vector P de cuatro bits (p3, p2, p1, p0) para cada bloque de L2.
  Indica presencia en las caches L1 hijas.
  P/e p3 = 1 en un bloque de L2B indica que ese bloque está presente en L1B3.
- Cuando la cache ***i*** del nivel L1 reemplaza un bloque, avisa a su L2 con el comando:

  L2(expl, @u)

Posibles estados de cada bloque en L2: **M2**, **S**, **I**

- **M2**: copia sucia y única en Nivel 2. (nL1 = 1L2 ≠ Mp; n = 0 ó 1)
  la copia está o bien en L2A o bien en L2B
  A lo sumo una cache L1 tiene una copia, le llamaremos cache nº "**j**"
- **S**: copia limpia y posiblemente replicada (nL1 = mL2 = Mp; n=1:8; m=1:2)

En resumen, en este sistema pueden existir múltiples copias de sólo lectura residentes en varias caches L1 y L2. También pueden existir bloques de lectura/escritura residentes en una pareja L1/L2 única (p/e en L1B3 y en L2B).

- **Memoria Principal y Crossbar**

  El modo de funcionamiento del *Crossbar* es conmutación de paquetes: direcciones y datos van por separado (*split-transaction*) para lograr un buen ancho de banda.

  En la figura solo se muestra el crossbar de datos (128 bits), pero existe otro parecido para enviar direcciones y comandos.

  A través del Controlador del *Crossbar* la Memoria Principal acepta una petición de L2A y de L2B en cada ciclo de red (250 Mhz) si el banco destino de la petición está libre. Los 8 bancos funcionan con independencia, pero cada dos (una sección) comparten un puerto único del *crossbar*. Cada banco tiene una latencia de 64 ns, transcurridos los cuales puede suministrar a la velocidad de la red todos los trozos del bloque (trozos de 128 bits).

  El último bus del crossbar, Bus 4, sirve para permitir transferencias L2A↔L2B.

Contestar a las preguntas siguientes:

a) Calcular el ancho de banda máximo agregado que puede entregar la Memoria Principal a las dos caches L2. Recordemos que existe un *Crossbar* duplicado e independiente para enviar direcciones a los bancos.

Notación: Megas o Gigas binarios: $Mi = 2^{20}$, $Gi = 2^{30}$
Megas o Gigas decimales: $M = 10^6$, $G = 10^9$

¿en qué condiciones se da este ancho de banda máximo agregado (L2A+L2B) ?

Asumimos un regimen permanente donde el sistema de memoria acumula peticiones y suministra continuamente (actúa como un sistema sobrado)

BWmax =

8 GB/s = 7,44 GiB/s

b) Calcular el ancho de banda agregado que puede entregar la Memoria Principal a las dos caches L2, suponiendo conflicto continuo en un banco único de Memoria Principal.

BWmax =

entre 1,33 y 1,24 GB/s

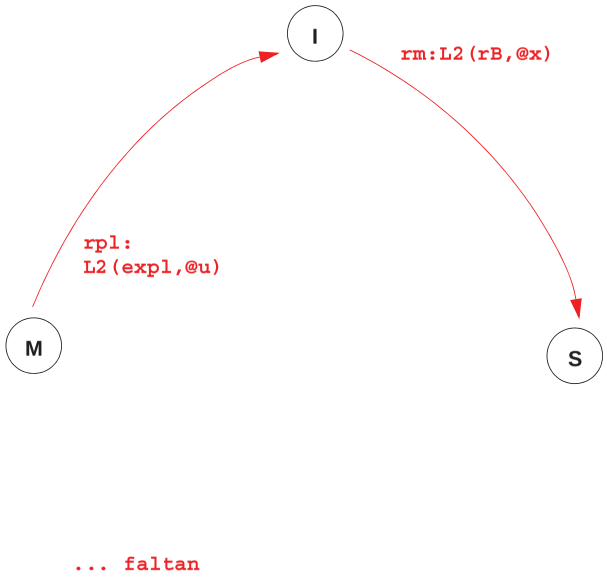
c) ¿Esta red de interconexión introduce algún tipo de degradación?

SI, porque bancos de la misma sección comparten Bus. Por tanto dos peticiones simultáneas a bancos de la misma sección deben servirse en secuencia.

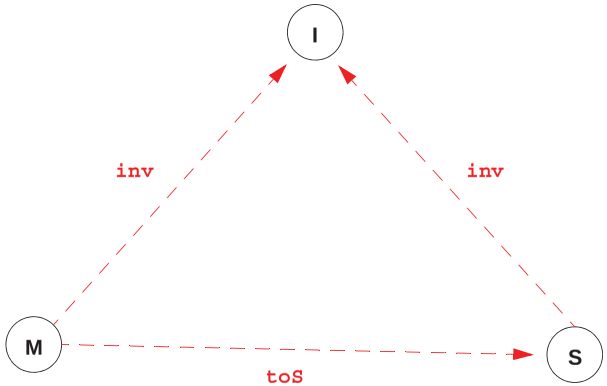
d) ¿Este multi es tipo NUMA o UMA?. ¿Por qué?

UMA, porque el coste de acceso a memoria principal (falla L1 y falla L2) es idéntico para todos los procesadores, independientemente del banco de memoria principal accedido.

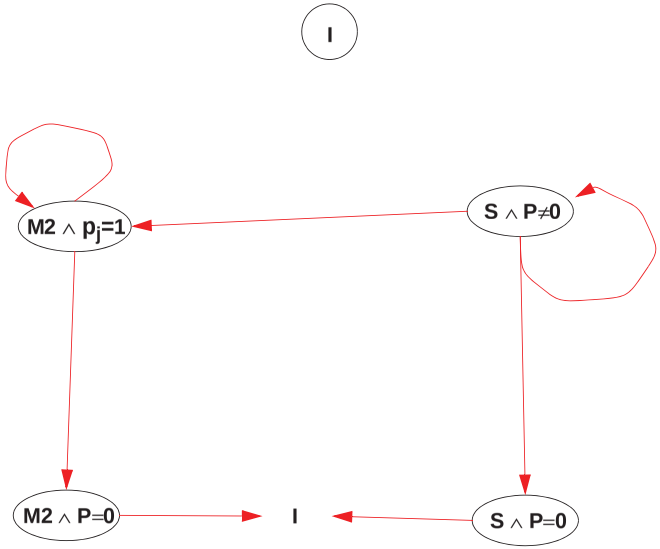
e) Diagrama de estados de una cache L1. Transiciones provocadas por el procesador



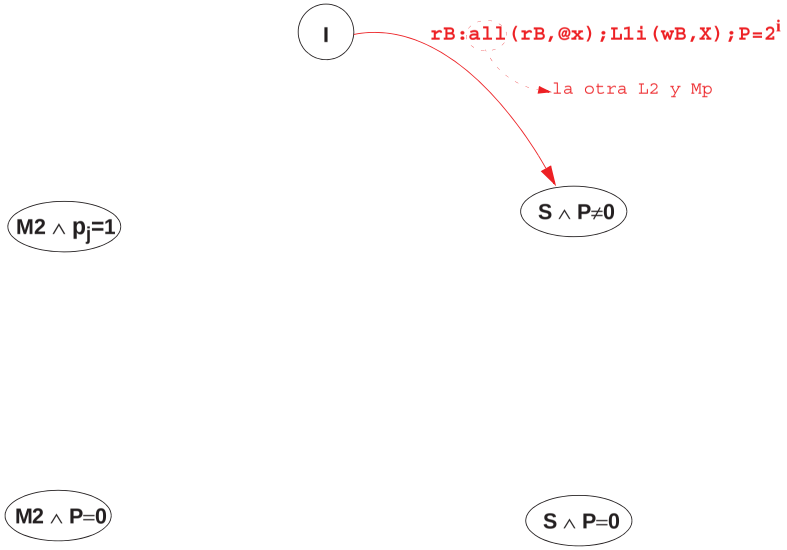
f) Diagrama de estados de una cache L1. Transiciones provocadas por comandos enviados desde L2. *Si no lo ves claro ahora, vuelve mas tarde ...*



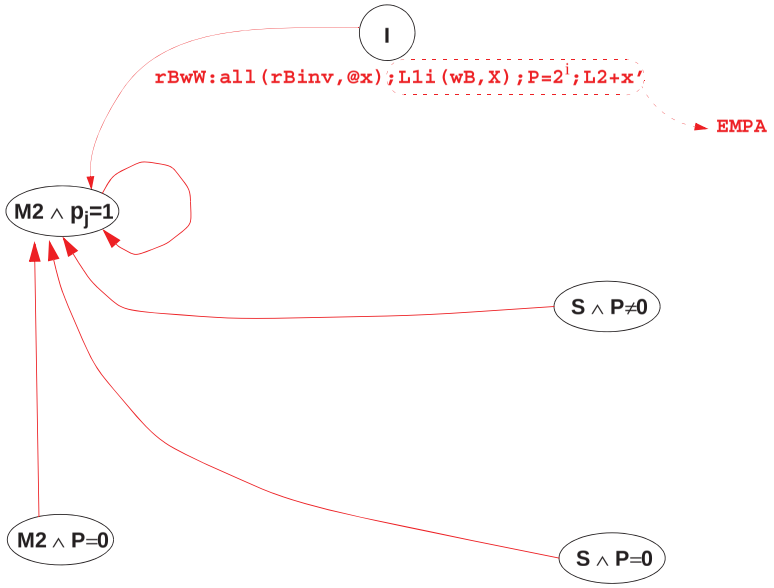
g) Transiciones y acciones de una cache L2 (1 de 4): provocadas por los eventos **acierto en escritura** y **expulsión** en una cache hija de nivel 1 (suponed la nº *i*, L1*i*). Añadid el evento **reemplazo en L2** (**rpl2**). P es el valor del vector de presencia P.



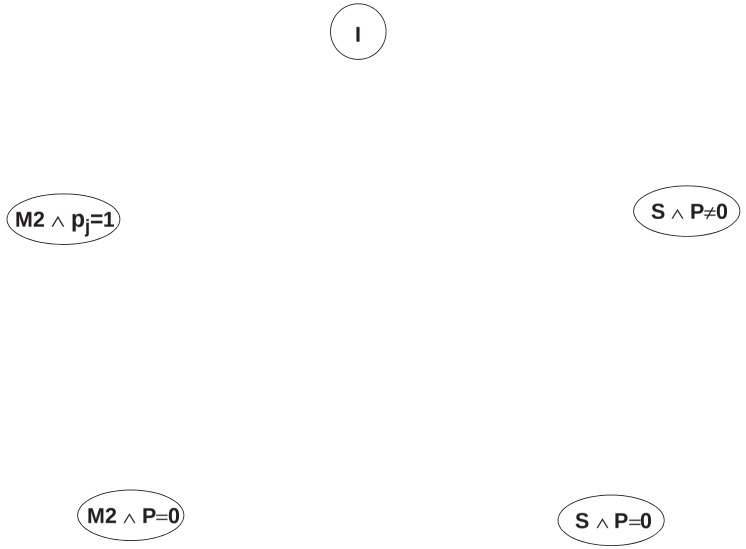
h) Transiciones y acciones de una cache L2 (2 de 4): por un **fallo en lectura** en una cache hija de nivel 1 (suponed la nº *i*, L1*i*).



i) Transiciones y acciones de una cache L2 (3 de 4): por un **fallo en escritura** en una cache hija de nivel 1 (suponed la nº *i*, L1*i*). Podeis utilizar el comando compuesto **`EMPA`** (**E**nviar a L1*i*, **M**arcar **P**resencia y **A**ctualizar palabra).



j) Transiciones y acciones de una cache L2 (4 de 4): por un **fallo en escritura** desde el otro grupo de procesadores (p/e: reacción de L2A a un fallo en escritura en la cache L1B3. Utilizad copia cache a cache siempre que sea posible.



25) Coherencia en un CMP con directorio y comunicación en malla

Feb. 2012

Vamos a diseñar parte del protocolo de coherencia de un CMP de 16 procesadores con una red de interconexión directa en malla. Los nodos de la red tienen dos niveles de cache *copy-back*, el primero privado (L1) y el segundo compartido (L2), ver figura.

L2 está dividida en 16 bancos independientes, uno en cada nodo. Por tanto, un bloque puede estar replicado en varias L1, pero únicamente en una L2. Llamamos **sede** de un bloque (residencia, *home*), a la única cache L2 que puede contenerlo[1].
Ejemplo: el bloque *x* está en el CMP. Su sede es el nodo **11** y existen copias limpias en los nodos **2**, **5** y **8**. Esto significa que el bloque *x* está en $L2_{11}$ y hay copia en $L1_2$, $L1_5$ y $L1_8$

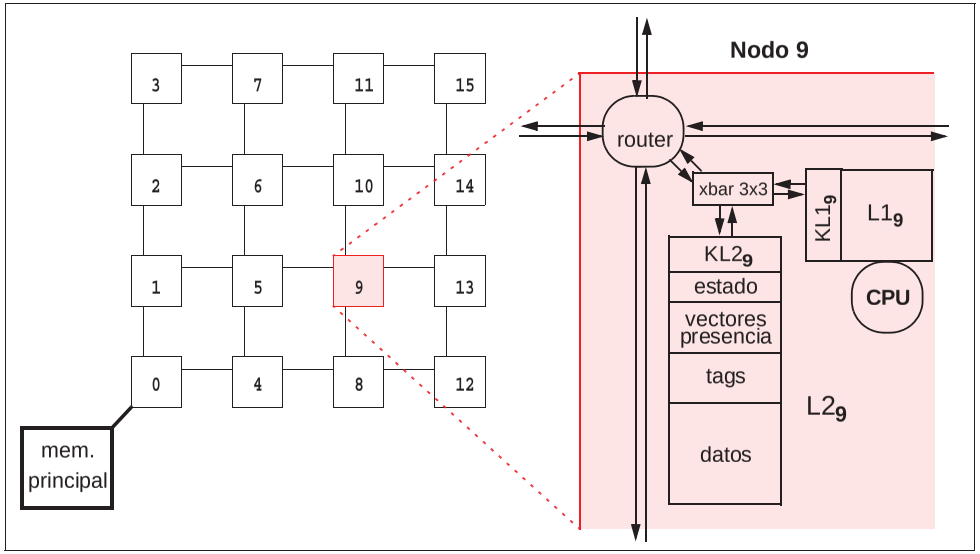
L2 mantiene la coherencia en el chip combinando varias técnicas:

- L2 es inclusiva, o sea, siempre contiene un superconjunto de todos los bloques de L1. Si es necesario, emite invalidaciones para conseguirlo.
- Cada banco de L2 tiene un directorio completo (*full-map*). Es decir, para cada bloque *x* en L2 existe un vector de presencia P que identifica a los nodos que tienen copia de ese bloque *x* en su L1.
- L2 reparte bloques e invalidaciones, y mantienen a cada bloque en estado {`M,S,A`}

  M: 1L1 ≠ (L2?Mp)

  S: nL1 = L2 = Mp; n ≥ 0

  A: Ausente. Estado al que se cae después de la primera transición de fallo en L2.



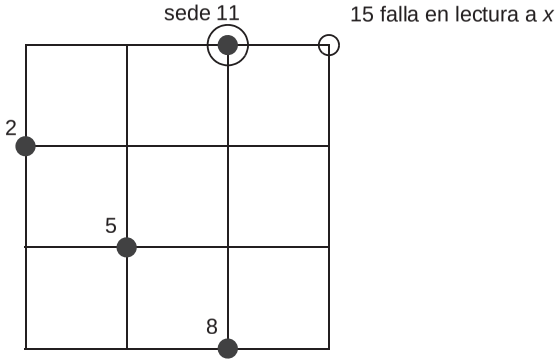
1. Por ejemplo, podemos usar los cuatro bits de menos peso del número de página de SO para distribuir los bloques entre las 16 caches L2.

Para los dos apartados siguientes vamos a suponer la siguiente situación inicial:

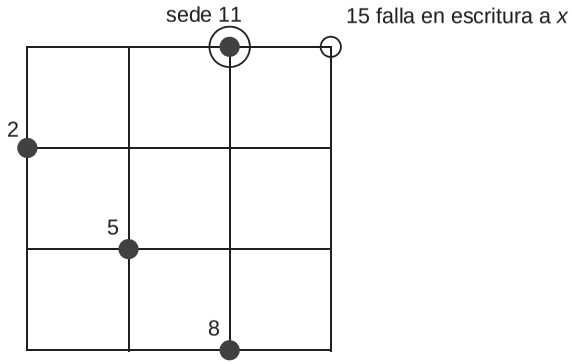
el bloque *x* con sede en el nodo **11** tiene tres copias limpias en los nodos **2**, **5** y **8**, ver figura.

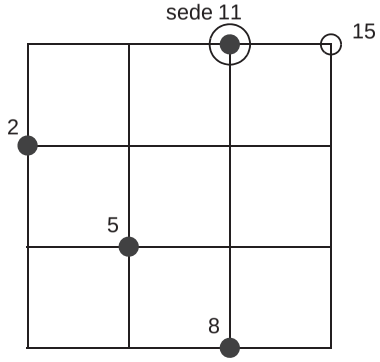
La ruta de encaminamiento es “primero Norte/Sur, después Este/Oeste”

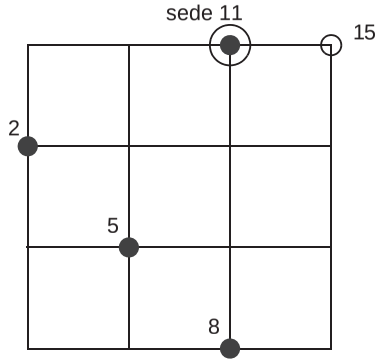
a) En esta situación el procesador del nodo **15** falla en lectura a ese bloque *x*, sin expulsar nada, ni limpio ni sucio. Anotad sobre el dibujo siguiente las transacciones y los cambios de estado, junto con su orden: (1), (2), ...



b) Partiendo de la misma situación inicial, ahora el procesador del nodo **15** falla en escritura, sin expulsar nada, ni limpio ni sucio. Anotad sobre los dibujo siguientes (utilizad dos o tres) las transacciones y los cambios, junto con su orden: (1), (2), (3), ...







Vamos a diseñar ahora parte del protocolo de coherencia de L2, centrándonos únicamente en:

- el controlador de L2 en el nodo *q* (**KL2**$_q$)
- los comados de L1$_q$ dirigidos a bloques con sede en ese nodo *q*.

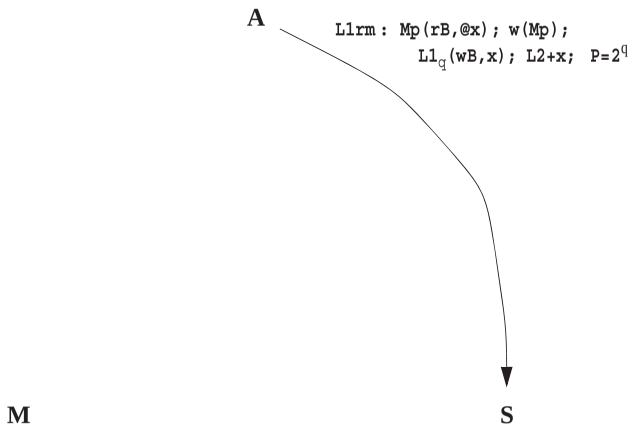
Los eventos de L1$_q$ que vamos a considerar son:

- fallos (L1rm, L1wm)
- primer acierto en escritura (L1wh)
- reemplazos limpios o sucios (L1rpl)

O sea, si L1$_q$ falla, escribe o reemplaza un bloque de otra sede, el comando irá directamente al *router* y no nos procupa.

c) Escribir las transiciones en **KL2**$_q$ provocadas por los eventos de L1$_q$, sin poner ahora los reemplazos de L2: transiciones **M→A** y **S→A** (apartado siguiente). Hay que responder siempre a L1$_q$ y poner puntos de espera con la letra "**w**". Para que os centréis, se da una de las transiciones:

fallo en lectura en L1 y en L2 (**L1rm**)



Notación sugerida:

- Enviar un comando a todas las caches L1 identificadas en el vector de presencia P: **L1(P)**(cmd, @ ...). Equivalente a L1*i*(cmd, @ ...) $\forall i \mid p_i = 1$
- Enviar un comando a todas las caches L1 identificadas en el vector de presencia P menos a una, por ejemplo la ***q***: **L1(P-2^q)**(cmd, @ ...). Equivalente a L1*i*(cmd, @ ...) $\forall i \neq q \mid p_i = 1$

d) Diagrama de transiciones de **KL2**$_q$ para sus propios reemplazos: transiciones **M→A** y **S→A** (1ª transición de fallo).