Ejercicios resueltos del Tema 3

1. En un multiprocesador SMP con 16 procesadores basado en un bus, que implementa el protocolo MESI para mantener la coherencia, supongamos una dirección de memoria incluida en un bloque que no se encuentra en ninguna cache. Indique los estados de este bloque en las caches y las acciones que se producen en el sistema ante la siguiente secuencia de eventos para dicha dirección:

a. Lectura generada por el procesador 1
b. Lectura generada por el procesador 2
c. Escritura generada por el procesador 1
d. Escritura generada por el procesador 2
e. Escritura generada por el procesador 3

RESPUESTA

Datos del ejercicio

Se accede a una dirección de memoria cuyo bloque no se encuentra en ninguna cache, luego debe estar actualizado en memoria principal.

Hay 16 nodos con cache y procesador, los notaremos por N1… N16

ESTADO INICIAL	EVENTO	ACCIÓN	ESTADO SIGUIENTE
N1) Inválido N2) Inválido N3) Inválido	P1 lee k	- N1 genera petición de lectura del bloque k (PtLec(k)) - N1 recoge la respuesta con el bloque (RpBloque(k)) depositada en el bus por la memoria principal, el bloque entra en la cache de N1 en estado exclusivo ya que no hay copia en otra cache del bloque (la salida de la OR cableada será 0).	N1) Exclusivo N2) Inválido N3) Inválido
N1) Exclusivo N2) Inválido N3) Inválido	P2 lee k	- N2 genera PtLec(k) - N1 observa PtLec(k) y, como tiene el bloque en estado exclusivo, lo pasa a compartido (la copia que tiene ya no es la única en caches) - N2 recoge RpBloque(k) que ha depositado la memoria, el bloque entra en estado compartido en la cache de N2.	N1) Compartido N2) Compartido N3) Inválido
N1) Compartido N2) Compartido N3) Inválido	P1 escribe en k	- N1 genera petición de lectura con acceso exclusivo del bloque k (PtLecEx(k)) (suponemos que no hay petición de acceso exclusivo sin lectura) - N2 observa PtLecEx(k) y, como tiene el bloque en estado compartido, lo invalida. - N1 no recoge RpBloque(k) depositada por la memoria (la petición también es de lectura) ya que tiene el bloque válido. El bloque pasa a estado modificado en la cache de N1.	N1) Modificado N2) Inválido N3) Inválido
N1) Modificado N2) Inválido N3) Inválido	P2 escribe en k	- N2 genera petición de lectura con acceso exclusivo de k (PtLecEx(k)) - N1 observa PtLecEx(k) y, como tiene el bloque en estado modificado, genera respuesta con el bloque RpBloque (k) inhibiendo la respuesta de memoria, y además, como el paquete pide exclusividad, invalida su copia del bloque. - N2 recoge RpBloque(k). El bloque entra en estado modificado en la cache de N2	N1) Inválido N2) Modificado N3) Inválido
N1) Inválido N2) Modificado	P3 escribe en k	- N3 genera petición de lectura con acceso exclusivo de k PtLecEx(k) - N2 observa PtLecEx(k) y, como tiene el	N1) Inválido N2) Inválido N3)

N3) Inválido		bloque en estado modificado, genera respuesta con el bloque RpBloque (k), y además, como el paquete pide exclusividad, invalida su copia del bloque. - N3 recoge RpBloque(k). El bloque entra en estado modificado en la cache de N3	Modificado

---

2. Se dispone de un multiprocesador de memoria distribuida con 64 nodos. Cada nodo dispone de 4 GB de memoria principal y una cache de 2 MB. Se quiere implementar un protocolo para mantenimiento de coherencia basado en directorios. Suponiendo que se necesitan dos bits de estado para un bloque en el directorio de memoria principal y que el tamaño de una línea de cache es de 64 bytes, calcular el porcentaje de memoria principal que supone el directorio en cada uno de las siguientes implementaciones: **(a)** Directorio de vector de bits completo. **(b)** Directorio de vector de bits asignados a grupos en el que un bit representa a un conjunto de 4 nodos. **(c)** Directorio limitado con capacidad para almacenar 8 punteros por bloque de memoria. **(d)** Directorio distribuido entre caches doblemente enlazado.

RESPUESTA

Datos del ejercicio:

- Memoria principal por nodo: 4GB = TMN
- Memoria cache por nodo: 2MB =TMC
- Línea de cache (bloque de memoria): 64 bytes = TLC
- Dos bits de estado por bloque en el directorio

(a) Directorio de vector de bits completo

Tamaño_por_nodo = Nº_de_bloques_memoria_nodo × Espacio_por_bloque

NOTA: para ahorrar cálculos es mejor no hacer operaciones hasta que sea necesario

$$Tamaño_{por_{nodo}} = \frac{TMN}{TLC} \times (64\,b + 2\text{b})$$

$$\%\_de_{TMN} = \frac{\frac{TMN}{TLC} \times 66\,b}{TMN} \times 100 = \frac{66\,b}{TMC} \times 100 = \frac{66\,b}{64 \times 8\text{b}} \times 100 = \frac{33}{256} \times 100 = 12,89$$

(b) Directorio de vector de bits asignados a grupos (grupos de 4 nodos)

Tamaño_por_nodo = Nº_de_bloques_memoria_nodo × Espacio_por_bloque

$$Tamaño_{por_{nodo}} = \frac{TMN}{TLC} \times \left(2\text{b} + \frac{64}{4}\,b\right)$$

$$\%\_de_{TMN} = \frac{\frac{TMN}{TLC} \times (2\text{b} + 16\,b)}{TMN} \times 100 = \frac{9\text{b}}{64 \times 4\text{b}} \times 100 = \frac{9}{256} \times 100 = 3,51$$

(c) Directorio limitado con capacidad para almacenar 8 punteros por bloque de memoria.

Tamaño_por_nodo = Nº_de_bloques_memoria_nodo × Espacio_por_bloque

$$Tamaño_{por_{nodo}} = \frac{TMN}{TLC} \times (2\text{b} + 8 \times \lg_2 64\,b) = \frac{TMN}{TLC} \times (2 + 8 \times 6)\,b$$

$$\%\_de_{TMN} = \frac{\frac{TMN}{TLC} \times (2 + 8 \times 6)\,b}{TMN} \times 100 = \frac{25\,b}{64 \times 4\text{b}} \times 100 = \frac{25}{256} \times 100 = 9,76$$

(d) Directorio distribuido entre caches doblemente enlazado

Tamaño_por_nodo = (Nº_de_bloques_memoria_nodo × Espacio_por_bloque) +
(Nº_de_bloquesCache × Espacio_para_dos_punteros)

Se supondrá que en el directorio asociado a la memoria principal del nodo hay dos punteros: uno a la última cache que leyó el bloque y otro a la primera que lo leyó.

$$Tamaño_{por_{nodo}} = \left[\frac{TMN}{TLC} \times (2 + 2 \times \lg_2 64)\, b\right] + \left[\frac{TMC}{TLC} \times (2 \times \lg_2 64)\, b\right]$$

$$Tamaño_{por_{nodo}} = \left[\frac{TMN}{TLC} \times (2 + 2 \times 6)\, b\right] + \left[\frac{TMC}{TLC} \times (2 \times 6)\, b\right] = \left[\frac{TMN}{TLC} \times 14\, b\right] + \left[\frac{TMC}{TLC} \times 12\, b\right]$$

$$\%\_de_4 GB = \frac{\left[\frac{TMN}{TLC} \times 14\, b\right] + \left[\frac{TMC}{TLC} \times 12\, b\right]}{TMN} \times 100 = \frac{TMN \times 14\, b + TMC \times 12\, b}{TMN \times TLC} \times 100$$

$$\%\_de_4 GB = \frac{2^{32} B \times 14\, b + 2^{21} B \times 12\, b}{2^{32} B \times 2^6 \times 2^3 b} \times 100 = \frac{2^9 \times 14 + 3}{2^{18}} \times 100 = \frac{2^{10} \times 7 + 3}{2^{18}} \times 100 = 2,73$$

---

3. Suponga que en un CC-NUMA de red malla tridimensional de 64 nodos se implementa un protocolo MESI basado en directorios con tres estados en el directorio (exclusivo, local y compartido). Cada nodo tiene 2 GBytes de memoria y una línea de cache supone 64 Bytes. Considere que se implementa un directorio limitado con 4 punteros. **(a)** Calcule el tamaño del directorio. **(b)** Indique cual sería el contenido del directorio, las transiciones de estados (en cache y en el directorio) y la secuencia de paquetes generados por el protocolo de coherencia en los siguientes accesos sobre una dirección D que se encuentra en la memoria del nodo 5 (inicialmente D no está en ninguna cache):

a. Lectura generada por el procesador del nodo 1
b. Escritura generada por el procesador del nodo 1
c. Lectura generada por el procesador del nodo 2
d. Lectura generada por el procesador del nodo 3
e. Escritura generada por el procesador del nodo 0

RESPUESTA

Datos del ejercicio

- Memoria principal por nodo: 2GB = TMN
- Línea de cache (bloque de memoria): 64 B = TLC
- Dos bits de estado por bloque en el directorio ya que hay que codificar tres estados (exclusivo, local y compartido).
- Directorio limitado para 4 punteros
- Se accede a una dirección de la memoria del nodo 5 cuyo bloque no se encuentra en ninguna cache, luego debe estar actualizado en memoria principal.

(a)

$$Tamaño_{por_{nodo}} = \frac{TMN}{TLC} \times (2b + 4 \times \lg_2 64\, b) = \frac{2^{31} B}{2^6 B} \times (2 + 4 \times 6)\, b = 2^{25} \times 26\, b = 104\ MB$$

(b)

Intervienen los nodos N0, N1, N, N3 y N5. L denota local, C compartido y E exclusivo. BD denota el bloque de la dirección D.

ESTADO INICIAL	EVENTO	ACCIÓN	ESTADO SIGUIENTE
N0) Inválido **N1) Inválido** **N2) Inválido** **N3)Inválido** **D) Local** **L** |			
Ningún puntero	P1 lee D	- N1 envía petición de lectura del bloque BD (PtLec(BD)) a N5 - N5 recibe PtLec(BD). Como tiene el bloque BD en estado Local, envía paquete de respuesta con el bloque confirmando estado Exclusivo para el bloque en la caché de N1 (RpBloqueEx) y pasa el estado del bloque en el directorio a Exclusivo.	N0) Inválido N1) Exclusivo N2) Inválido N3)Inválido D) Exclusivo E | N 1 |
N0) Inválido **N1)**	P1 escribe	- N1 modifica el estado de BD y lo pasa a modificado. No tiene que informar a nadie	N0) Inválido N1)

Exclusivo N2) Inválido N3)Inválido D) Exclusivo **E N 1**	en D	de que escribe en BD porque tiene BD en estado Exclusivo	Modificado N2) Inválido N3)Inválido D) Exclusivo E N 1
N0) Inválido N1) Modificado N2) Inválido N3)Inválido D) Exclusivo **E N 1**	P2 lee D	- N2 envía PtLec(BD) a N5 porque no tiene BD. - N5, como tiene BD en estado Exclusivo (no sabe si tiene copia válida del bloque): (1) reenvía (RvPetLec(BD)) la petición al nodo N1 (que según el directorio tiene copia válida del bloque), (2) pone en la entrada del bloque en el directorio estado pendiente de Compartido y un puntero a N2. - N1 recibe RvPetLec(BD) y: (1) envía a N2 un paquete de respuesta con el bloque confirmándole estado Compartido (RpBloqueComp), (2) envía un paquete a N5 que le confirma estado Compartido y que además incluye el bloque porque tiene N1 el bloque en estado Modificado, (3) pasa BD en su cache a Compartido. - N5 recibe la respuesta de N1 (RpBloqueComp) y pone el estado del bloque en el directorio a compartido	N0) Inválido N1) Compartido N2) Compartido N3)Inválido D) Compartido C N 1 N 2
N0) Inválido N1) Compartido N2) Compartido N3)Inválido D) Compartido **C N 1 N 2**	P3 lee D	- N3 envía PtLec(BD) a N5 porque no tiene BD. - N5, como tiene BD en estado compartido: (1) envía paquete de respuesta con el bloque confirmando estado Compartido (RpBloqueComp) a N3, y (2) pone en el directorio un puntero a N3. - El bloque entra a N3 en estado Compartido	N0) Inválido N1) Compartido N2) Compartido N3) Compartido D) Compartido C N 1 N 2 N 3
N0) Inválido N1) Compartido N2) Compartido N3) Compartido D) Compartido **C N 1 N 2 N 3**	P0 escribe en D	- N0 envía a N5 petición de lectura de BD con acceso exclusivo PtLecEx(BD) - N5 recibe PtLecEx(BD) y, como tiene el bloque en estado compartido: (1) genera una lista con los nodos a los que tiene que enviar invalidaciones (RvInv(BD)) informando de que el N0 es quien pide acceso exclusivo, (2) pone en el directorio, como único puntero, un puntero a N0 y pone el estado de BD en el directorio en Exclusivo, (3) envía paquete (RpBloqueExc3Inv) con el bloque al nodo N0 confirmando estado modificado (acceso exclusivo) e indicándole el número de paquetes de respuesta a invalidaciones que debe recibir (en total 3), y envía a la lista de nodos las invalidaciones (RvInv(BD, N0)). - Los nodos N1, N2 y N3, cuando reciben RvInv de BD invalidan su copia de BD y envían a N0 RpInv para confirmar la invalidación y permitir así a N0 garantizar un orden en los accesos a BD para garantizar coherencia.	N0) Modificado N1) Inválido N2) Inválido N3) Inválido D) Exclusivo E N 0

		- N0 usa BD en cuanto recibe RpBloqueExc3Inv de N5 pero no responde a ninguna petición sobre BD que le envía N5 hasta que no ha recibido las tres RpInv (para garantizar atomicidad, se incluye coherencia)	

---

4. Supongamos que se va a ejecutar en paralelo el siguiente código (inicialmente *x* e *y* son 0):

P1	**P2**
x=1;	y=1;
x=2;	y=2;
print y ;	print x ;

Qué resultados se pueden imprimir si:

(a) Se ejecuta en un multiprocesador con consistencia secuencial.

(b) Se ejecuta en un multiprocesador basado en un bus que garantiza todos los órdenes excepto el orden W→R. Esto es debido a que los procesadores tienen buffer de escritura, permitiendo el procesador que las lecturas en el código que ejecuta adelanten a las escrituras que tiene su buffer. (Obsérvese que hay varios posibles resultados).

RESPUESTA

Suponemos que el compilador no altera ningún orden garantizado.

(a) Si P1 es el primero que imprime puede imprimir 0, 1 o 2, pero P2 podrá imprimir sólo 2. Esto es así porque se mantiene orden secuencial y, por tanto, si P1 ha leído "y", ha asignado ya a "x" un 2. Si P2 es el primero que imprime podrá imprimir 0, 1 o 2, pero entonces P1 sólo puede imprimir 2. Combinaciones:

P1 P2
0 2
1 2
2 2
2 0
2 1

(b) Si no se mantiene el orden W→R además de los resultados anteriores, los dos procesos pueden imprimir:

P1 P2
1 1
0 1
0 2
1 0
2 0
0 0

ya que no se asegura que cuando un proceso ejecuta la lectura de la variable que imprime print haya ejecutado las instrucciones anteriores que escriben en x (P1) o en y (P2). P1 puede imprimir 1 o 2 o 0, y P2 1 o 2 o 0. Todas las combinaciones son posibles.

---

5. Supongamos que se va a ejecutar en paralelo el siguiente código (inicialmente *x* e *y* son 0):

P1	**P2**

x=30;
y=40;
flag=1;
	while (flag==0) {};
	r1=x;
	r2=y;

Qué datos puede obtener P2 en r1 y r2 si:

(a) Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.

(b) Se ejecutan en un multiprocesador con consistencia de liberación. Razone su respuesta.

RESPUESTA

Suponemos que el compilador no altera ningún orden garantizado.

(a) Si se implementa consistencia secuencial en r1 se almacena 30 y en r2 40.

RAZONAMIENTO:

Esto es así porque se garantizan los órdenes de acceso a memoria W->W y R->R que aparecen en el código que ejecuta un proceso:

1) En P1, al garantizarse el orden W->W, la escritura de 1 en flag no se realiza hasta que no se han realizado las escrituras en x e y que preceden a la escritura de flag en el código de P1.

2) Hasta que P2 no encuentra en flag un 1 no almacena en r1 el contenido de x ni en r2 el contenido de y. Al mantenerse el orden R->R, en P2 no se adelanta la lectura de x ni la lectura de y a la lectura de flag aunque se permita la lectura especulativa (ejecutar instrucciones cuya ejecución depende de una condición de salto antes de verificar que se cumple la condición).

Por tanto, como se garantiza W->W y además se garantiza R->R, si P2 ve en flag un 1 y, por tanto, sale del bucle, va a ver al leer en x 30 y en y 40.

(b) Si se implementa consistencia de liberación se pueden dar los siguientes resultados

r1	r2
0	0
0	40
30	0
30	40

RAZONAMIENTO:

1) Al no garantizarse el orden entre accesos de escritura (W->W), P1 podría escribir en flag un 1 antes de escribir 30 en x o 40 en y (obsérvese que la escritura y=40 podría también adelantar a x=30). Por lo que P2 podría leer en flag un 1 y, por tanto, salir del bucle, antes de que en x o en y pudiera ver los valores que escribe P1 en estas variables (vería entonces 0).

2) Al no garantizarse el orden entre accesos de lectura (R->R), si se permite ejecutar lecturas cuya ejecución depende de una condición de salto antes de verificar que se cumple la condición (ejecución especulativa), en P2 se podría, en la última iteración del bucle, adelantar la lectura de x o la lectura de y o ambas a la de flag. En este caso P2 podría entonces leer de forma especulativa los valores de x e y que tienen en su cache antes de que P1 los modifique y modifique flag. En estas circunstancias podría obtenerse en r1 o en r2 o en ambos un 0.

---

6. 6. Se quiere implementar un cerrojo simple en un multiprocesador basado en procesadores de la línea x86 de Intel, en particular, procesadores basados en la microarquitectura NetBurst o Core. **(a)** Teniendo en cuenta el modelo de consistencia de memoria que ofrece el hardware de este multiprocesador ¿podríamos implementar la función de liberación del cerrojo simple usando “mov k, 0”, siendo k la variable cerrojo? Razone su respuesta. **(b)** ¿Cómo se debería implementar la función de liberación de un cerrojo simple en un Itanium? Razone su respuesta.

7. En la red multietapa del NYU Ultracomputer y el IBM RP3, los conmutadores tenían soporte para combinar peticiones *Fetch&Add* simultáneas a la misma dirección de memoria en una única petición. De esta forma pretendían reducir los conflictos en la red al acceder a variables (conflictos debidos a peticiones concurrentes a la misma dirección o contención *“hot-spot”)*. Los conmutadores combinantes tienen hardware para:

a. Reducir peticiones *Fetch&Add* con una misma variable que llegan simultáneamente por distintas entradas, a una sola con los mismos efectos sobre la variable compartida que las dos peticiones que combina ((a)). El conmutador suma los valores que cada petición pretende añadir a la variable compartida, de forma que este resultado será el que la petición combinada tenga como valor a acumular en la variable compartida.

b. Devolver a cada petición *Fetch&Add* un valor diferente, simulando una ejecución secuencial ((b)). Los valores que se obtienen con cada petición han de corresponder con los valores que se obtendrían según alguna ejecución secuencial de las primitivas *Fetch&Add*. El conmutador genera los valores que va a devolver cuando recibe la respuesta a la petición *Fetch&Add* combinada que generó.

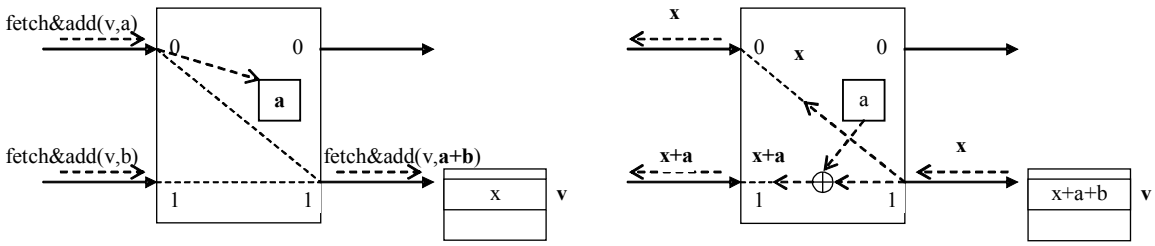


Figura 1. **(a)** Reducción en un conmutador de peticiones Fetch&Add simultáneas. Se almacena el valor de la petición por la entrada 0 del conmutador para simular al devolver resultados, que la petición que llega por 0 se realiza antes. **(b)** Se generan los valores a devolver a cada petición previamente combinada, el valor de entrada **x** directamente se devuelve por 0, y por 1 se devuelve el resultado de sumar **x** con el valor almacenado. Obsérvese que si la petición de arriba se realizara antes que la de abajo, la de abajo, Fetch&Add(v,b), devolvería x+a.

Sea un multiprocesador con 8 procesadores (*P0* a *P7*) y 8 módulos de memoria (de 8 Mbyte y con entrelazado de orden inferior) conectados con una red Omega que combina peticiones *Fetch&Add* en sus conmutadores. Los procesadores *P0*, *P1*, *P4*, inician en paralelo las siguientes peticiones a la red:

P0: *Fetch&Add*(v,6) P1: *Fetch&Add*(v,2) P4: *Fetch&Add*(v,7)

*Nota: la variable **v** se corresponde con la posición de memoria 1B76ACh; el valor inicial de **v** es 10*

(a) Dibujar la red y explicar razonadamente cuántos accesos a la variable **v** llegan a la memoria.

(b) Qué valor tendrá dicha variable (v) una vez ejecutados los *Fetch&Add* por los tres procesadores, y

(c) Qué valor se devuelve a cada uno de los procesadores.

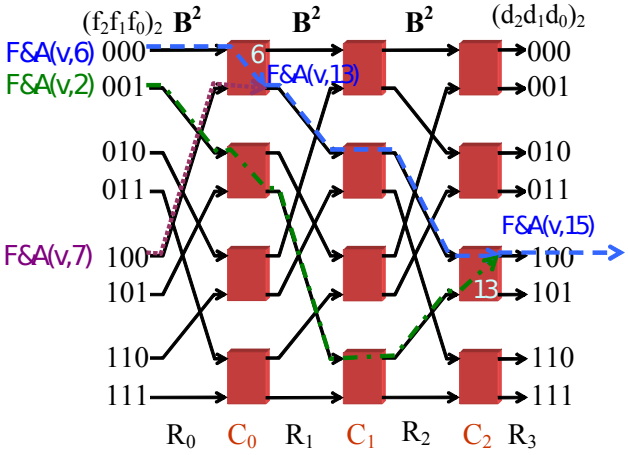
RESPUESTA

Datos del ejercicio:

Necesitamos conocer el módulo de memoria (salida de la red) en la que se encuentra la variable compartida a la que se accede. Esta variable se corresponde con la dirección 1B76ACh. Como el entrelazado de memoria es de orden inferior, direcciones consecutivas de memoria se encuentran en distinto módulo, por tanto, los bits menos significativos de la dirección identifican el módulo en el que se encuentra la dirección. Como hay 8 módulos, se necesitan 3 bits para identificarlos. Los tres bits menos significativos de 1B76ACh nos indican en qué módulo se encuentra la variable, en este caso en el módulo 4 (ya que Ch es 1100b).

(a).Dibujar la red y explicar razonadamente cuántos accesos a la variable **v** llegan a la memoria.

1) En el siguiente gráfico se pueden ver los conmutadores que reciben simultáneamente peticiones Fetch&Add sobre la misma variable. Estos conmutadores reducen las peticiones Fetch&Add sobre la misma variable en una única petición que tendrá los mismos efectos en la memoria que una ejecución secuencial de las dos peticiones.



2) Un conmutador de la etapa 0 recibe *Fetch&Add*(v,6) por arriba y *Fetch&Add*(v,7) por abajo, combina estas dos peticiones en *Fetch&Add*(v,7+6), y almacena 6 en el conmutador; 6 es el valor que quiere sumar la petición que se recibe por la entrada superior del conmutador. Este valor almacenado se va a utilizar para generar el valor que se va a devolver como resultado (respuesta) a la petición que llega por la entrada inferior del conmutador; se va a simular que la petición que entra por arriba se realiza antes.

3) Un conmutador de la etapa 2 recibe *Fetch&Add*(v,13) por arriba y *Fetch&Add*(v,2) por abajo, combina estas dos peticiones en *Fetch&Add*(v,13+2), y almacena 13 en el conmutador; 13 es el valor que quiere sumar la petición que se recibe por la entrada superior del conmutador. Este valor almacenado se va a utilizar para generar el valor que se va a devolver como resultado a la petición que llega por la entrada inferior del conmutador; se va a simular que la petición que entra por arriba se realiza antes.

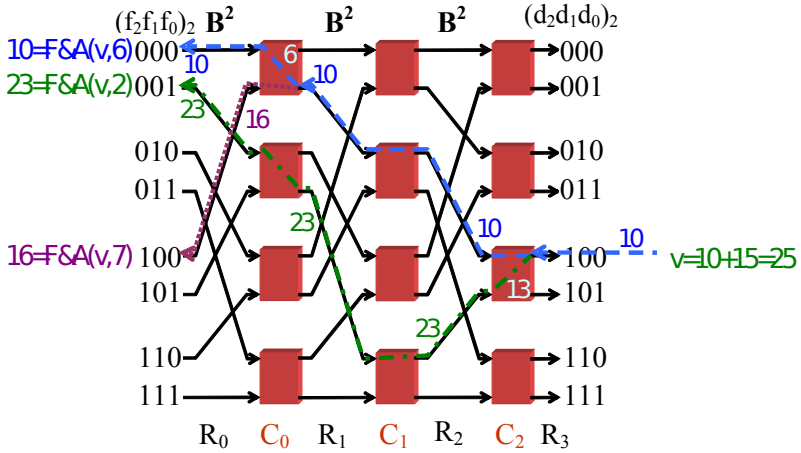
4) Finalmente al módulo de memoria donde está v sólo llega *Fetch&Add(v,15)*, hay pues un solo acceso a v.

(b) Qué valor tendrá dicha variable (v) una vez ejecutados los *Fetch&Add* por los tres procesadores

La petición que llega a la memoria es *Fetch&Add(v,15)*. Esta petición hará que se sume 15 a la variable y que se devuelva el valor de la variable antes de esta modificiación. Como el contenido de v es 10, se devuelve 10 y queda almacenado en memoria 15+10=25.

(c) Qué valor se devuelve a cada uno de los procesadores.

1) El dibujo de la figura ilustra como se van propagando por la red y generando los valores a devolver a cada petición *Fetch&Add* pendiente (habrá canales dedicados a las trasferencias de las repuestas a las peticiones independientes de los utilizados para las peticiones).



2) Al conmutador de la etapa 2 que realizó la reducción de dos paquetes de petición de v, cuando le llega la respuesta a estas peticiones con el valor a devolver (10) devuelve directamente este valor como respuesta a la petición que llego por la

entrada superior, y el resultado de sumar 10 con el valor almacenado en la reducción (13) a la petición que llegó por la entrada inferior, le devuelve por tanto 10+13=23.

3) Al conmutador de la etapa 0 que realizó la reducción de dos paquetes de petición de v, cuando le llega la respuesta a estas peticiones con el valor a devolver (10) devuelve directamente este valor como respuesta a la petición que llego por la entrada superior, y el resultado de sumar 10 con el valor almacenado en la reducción (6) a la petición que llegó por la entrada inferior, le devuelve por tanto 10+6=16.

4) Los valores que se devuelven como respuesta a las peticiones simulan una ejecución secuencial, es como si se hubieran ejecutado las peticiones Fetch&Add sobre v en este orden: 1) Primero la petición del procesador P0, que dejaría la memoria a 10+6=16 y devolvería como respuesta a P0 10. 2) A continuación la petición de P4, que dejaría en memoria 16+7=23 y devolvería como respuesta a P4 16. 3) Por último, la petición de P1 que dejaría en memoria 23+2=25 y devolvería como respuesta 23.

---

8. ¿Qué ocurre si en el segundo código para implementar barrera visto en clase eliminamos la variable local, *cont_local*, sustituyéndola en los puntos del código donde aparece por el contador compartido asociado a la barrera *bar[id].cont*?

RESPUESTA

Pueden surgir problemas pudiendo incluso no funcionar bien como barrera.

Si se usa el contador global en el if que comprueba si contador es ya igual al número de procesos que se sincronizan con la barrera, un proceso puede encontrar, cuando llegue al if, que contador es igual al número de procesos sin ser el último proceso que ha incrementado el contador. Esto puede provocar el siguiente problema:

Además del proceso que ha hecho contador igual al número de procesos, otros que lo han incrementado antes pueden encontrar la condición del if verdadera y escribir, por tanto, en contador global y en la bandera global. Todas estas escrituras provocan accesos a través de la red para transferir el bloque de memoria que contiene la información sobre la barrera. Estas transferencias adicionales simplemente devalúan prestaciones, nada más. El problema aparece si la barrera se vuelve a reutilizar por los mismos procesos y el SO suspende a uno de los procesos que encuentran contador igual a *num_procesos* antes de escribir un 0 en el contador global. Puede ocurrir que, mientras este proceso está bloqueado, el resto de procesos salgan de la barrera y vuelvan a reutilizarla. Conforme los procesos van ejecutando de nuevo la barrera, incrementan el contador global y se quedarán esperando a que éste llegue a ser igual a *num_procesos*. Pero nunca llegará a alcanzar este valor porque cuando el proceso suspendido vuelva a ejecutarse pondrá a 0 el contador global perdiéndose la cuenta del número de procesos que están ya esperando en la barrera.

---

9. Suponiendo que la arquitectura dispone de instrucciones Fetch&Add, simplifique el segundo código para barreras visto en clase.

RESPUESTA

En la siguiente figura se destaca en rojo el cambio realizado. Se decrementa uno a num_procesos en el if porque Fetch&Add devuelve el valor antes de la modificación, no después de la modificación. Por tanto, cuando llega el último proceso a la barrera se devuelve nom_procesos-1.

Barrera *sense-reversing*

```
Barrera(id, num_procesos)
{
  bandera_local= !(bandera_local)       //se complementa la bandera local
  cont_local = Fecht&Add (bar[id].cont,1);        //cont_local es una variable privada
    if (cont_local ==num_procesos-1)  {
     bar[id].cont=0;                    //se hace 0 el contador asociado a la barrera
     bar[id].bandera= bandera_local;    //para liberar los procesos en espera
  }
  else  while (bar[id].bandera!= bandera_local) {};   //espera ocupada
}
```

---

10. Se quiere paralelizar el siguiente ciclo de forma que la asignación de iteraciones a los procesadores disponibles se realice en tiempo de ejecución (dinámicamente):

```
For (i=0; i<100; i++) {
    Código que usa i
}
```

Nota: Considerar que las iteraciones del ciclo son independientes y que el único orden no garantizado por el sistema de memoria es W->R.

(a) Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva *Fetch&Or* para garantizar exclusión mutua.

(b) Paralelizar el anterior ciclo en un multiprocesador que además tiene la primitiva *Fetch&Add*.

RESPUESTA

Se supone que el único orden que no garantiza el hardware es el orden W->R.

(a) Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva *Fetch&Or* para garantizar exclusión mutua.

Bucle

```
for (i=0;i<100;i++) {
   código para i   }
```

Dinámica

```
while (Fetch&Or(k,1)==1); //lock(k)
   n=i;  i=i+1;
mov k,0;    //unlock(k)
while (n<100) {
      código para n ;
      while (Fetch&Or(k,1)==1);
         n=i;  i=i+1;
      mov k,0;
 }
```

Nota: la variable i estaría inicializada a 0 (por ejemplo, se puede iniciar cuando se declara)

Habrá que indicar al compilador que no debe cambiar las instrucciones “mov k,0” de sitio.

(b) Paralelizar el anterior ciclo en un multiprocesador que además tiene la primitiva *Fetch&Add*.

Bucle

```
for (i=0;i<100;i++) {
   código para i   }
```

Dinámica

```
n=Fetch&Add(i,1);
while (n<100) {
      código para n ;
      n=Fetch&Add(i,1);
 }
```

Nota: la variable i estaría inicializada a 0 (por ejemplo, se puede iniciar cuando se declara)
