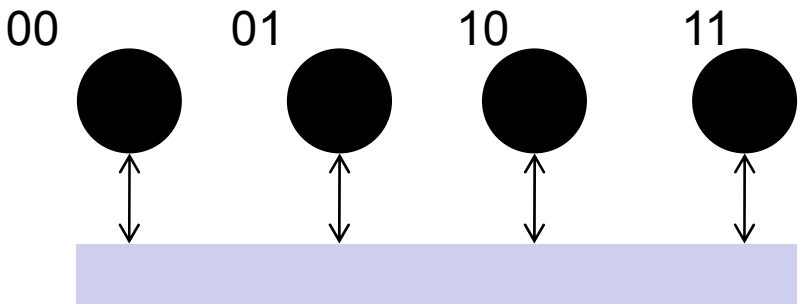
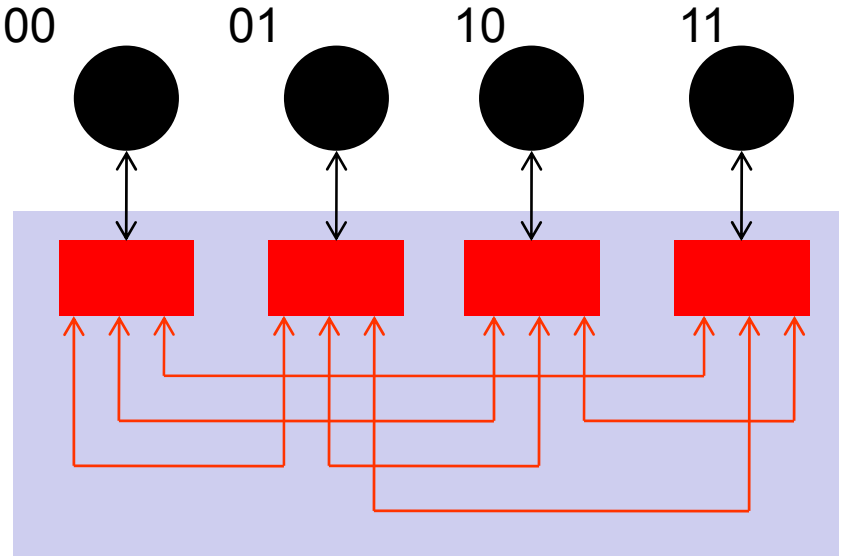
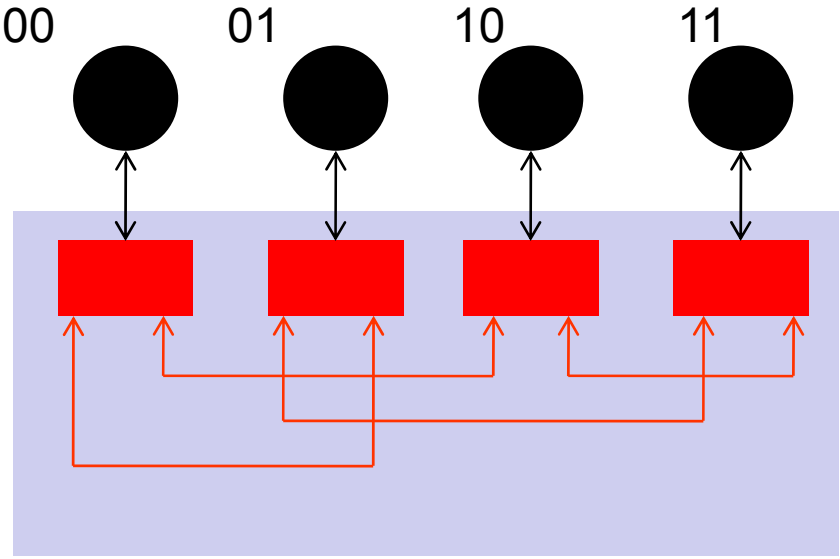
Apéndice para la explicación del problema 8 del Tema 2

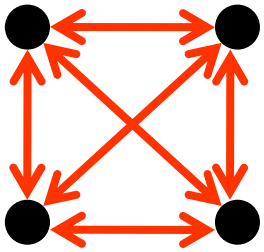
```
/* n   : elementos a sumar
/* p   : procesadores
/* pn  : identificador de procesador (idproc)

for (i=0; i<(n/p); ++i) S=S+V(i);

/* Reduccion
reduction(S,S,1,type,ADD,0,grupo);

if (pn==0) printf("Suma=%d",S);
```

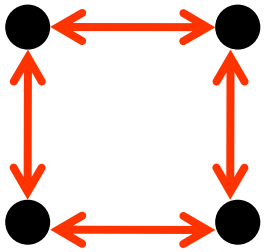
```
/* n   : elementos a sumar
/* p   : procesadores
/* pn  : identificador de procesador (idproc)

for (i=0; i<(n/p); ++i) S=S+V(i);

/* Reduccion
if (pn==0) for (i=1; i<p; i++) {
                        receive(SS,1,type,i,grupo);
                        S=S+SS;
}
else send(S,1,type,0,grupo);

if (pn==0) printf("Suma=%d",S);
```


00
01
10
11

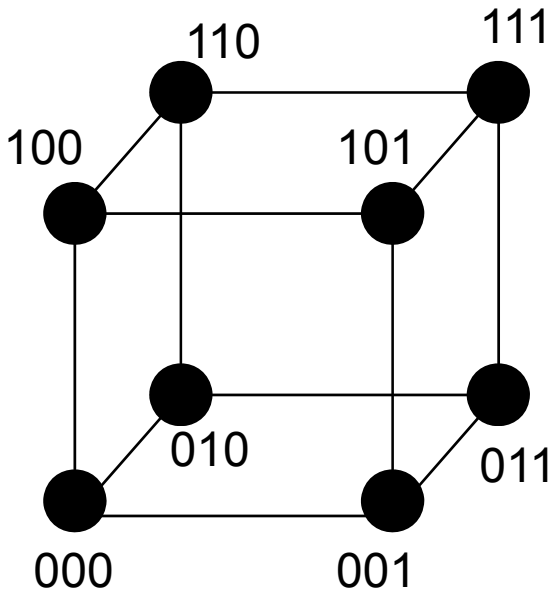

00
01
10
11


00
01
10
11





```
/* n   : elementos a sumar
/* p   : procesadores (potencia de 2)
/* dim : p = power(2,dim)
/* pn  : identificador de procesador (idproc)

for (i=0; i<(n/p); ++i) S=S+V(i);

/* Comunicacion en cada una de las dimensiones
for (i=dim; i>0; --i) {

        if (pn<power(2,i)) {

                npn= pn ^ power(2,i-1);
                /* exor bit a bit pn y power(2,i-1)

                if (npn<pn) send(S,1,type,npn,group);
                else {
                        receive(SS,1,type,npn,group);
                        S=S+SS;
                }
        }
}
if (pn==0) printf("Suma=%d",S);
```



dim=3

i=3; pn < power(2,3)=**8**

npn = pn ^ power(2,2)= pn ^ (100)

pn=0 ⟵ npn=4

pn=1 ⟵ npn=5

pn=2 ⟵ npn=6

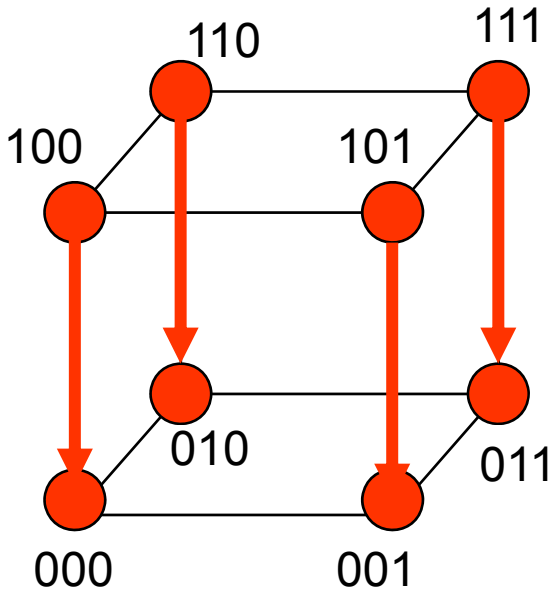
pn=3 ⟵ npn=7

pn=4 ⟶ npn=0

pn=5 ⟶ npn=1

pn=6 ⟶ npn=2

pn=7 ⟶ npn=3



dim=3

i=2; pn < power(2,2)=**4**

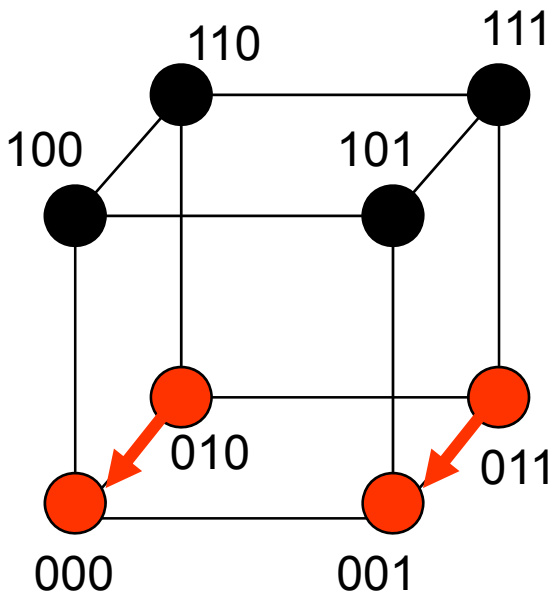
npn = pn ^ power(2,1)= pn ^ (010)

pn=0 ⟵ npn=2

pn=1 ⟵ npn=3

pn=2 ⟶ npn=0

pn=3 ⟶ npn=1



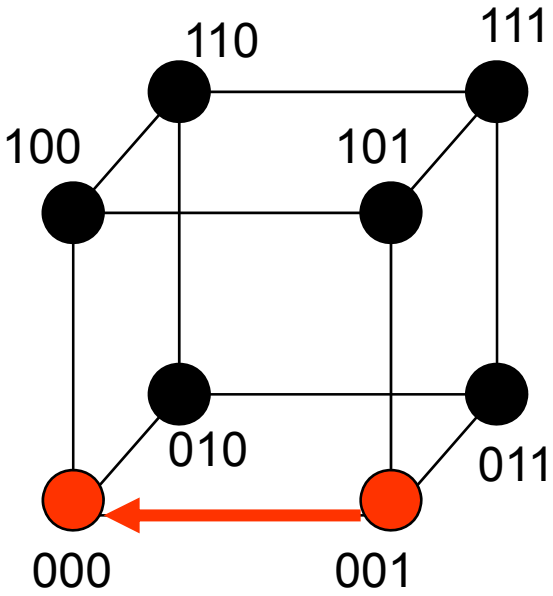
dim=3

i=1; pn < power(2,2)=**2**

npn = pn ^ power(2,0)= pn ^ (001)

pn=0 ⟵ npn=1

pn=1 ⟶ npn=0



TEMA 2

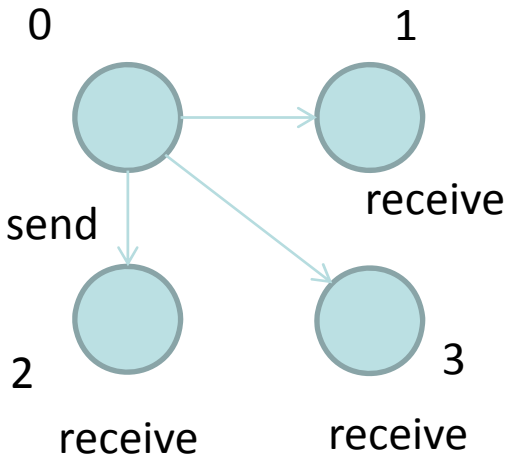
Problema 11

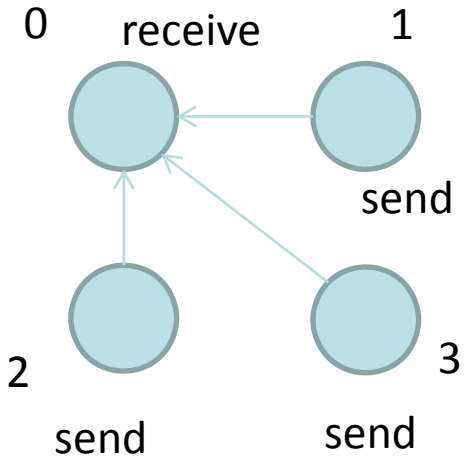
Versión 1	Versión 2

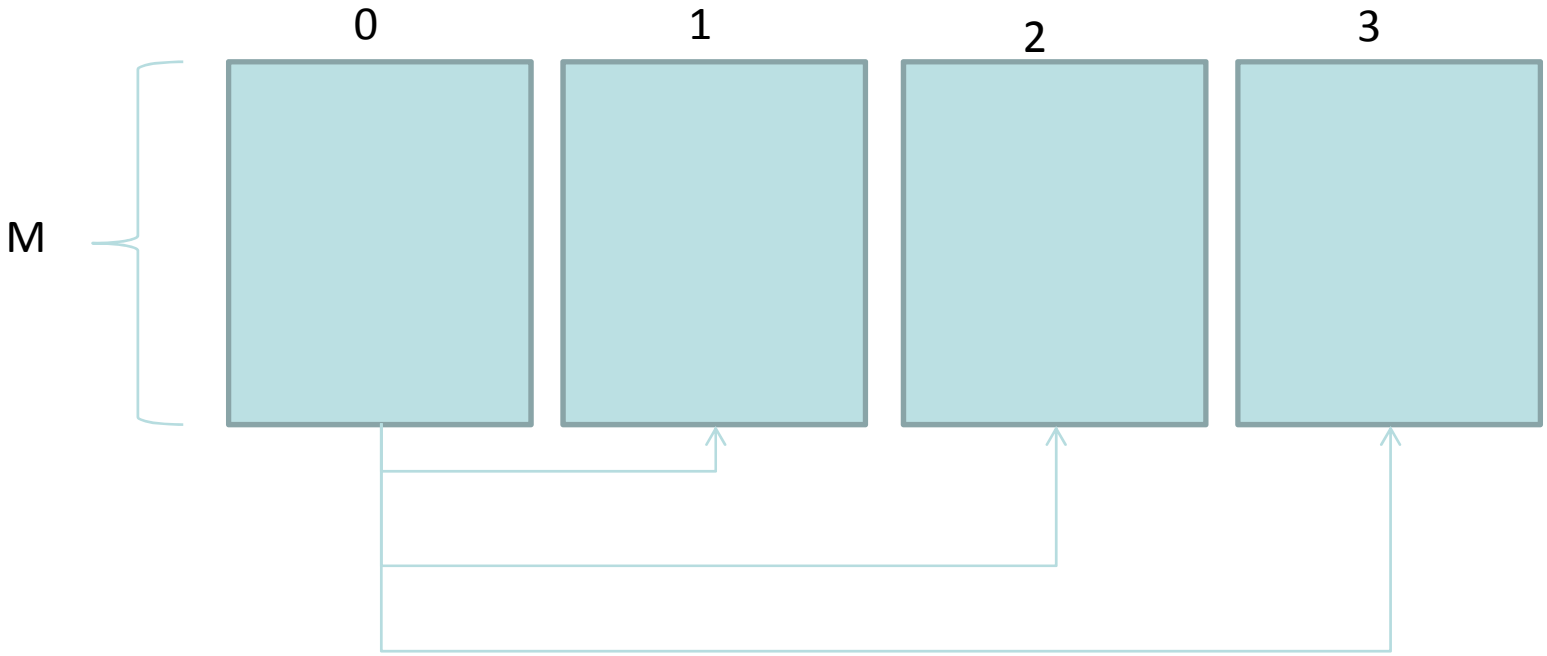
b=0; i=0;
while ((b==0)&& (i<M)) do {
if (NP[i]==x) b=1;
i++;
}
if (b==1) printf(“%d ES primo”, x);
elseprintf(“%d NO es primo”, x);
	b=0;
	for(i=0;i<M;i++) {
	if (NP[i]==x) b=1;
	}
	if (b==1) printf(“%d ES primo”, x);
	elseprintf(“%d NO es primo”, x);

Problema 11

```
b=0; i=0;

//difusión del vector NP y de x
if (idproc==0)
 for (i=1; i<numprocesos) send(NP,M,tipo,i,grupo);
else receive(NP,M,tipo,0,grupo);
if (idproc==0)
 for (i=1; i<numprocesos) send(x,1,tipo,i,grupo);
else receive(x,1,tipo,0,grupo);

//Cálculo
while ((b==0)&& (i<M)) do {
   if (NP[i+idproc]==x) b=1;
  i=i+num_procesos;
}

// Recogida de resultados
if (idproc==0)
   for (i=1; i<num_procesos) {
            receive(baux,1,tipo,i,grupo);
            b=b &baux;
   }
else send(b,1,tipo,0,grupo);

if (idproc==0)
    if (b==1) printf("%d ES primo", x);
elseprintf("%d NO es primo", x);
```



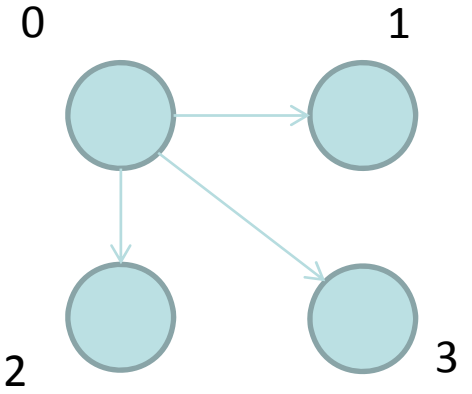




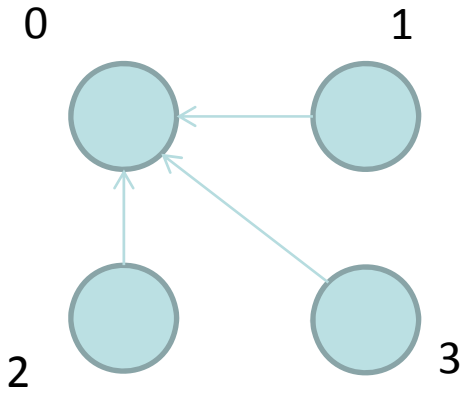
nprod=4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
iprod=0	**0**	1	2	3	**4**	5	6	7	**8**	9	10	11	**12**	13	14	15
iprod=1	0	**1**	2	3	4	**5**	6	7	8	**9**	10	11	12	**13**	14	15
iprod=2	0	1	**2**	3	4	5	**6**	7	8	9	**10**	11	12	13	**14**	15
iprod=3	0	1	2	**3**	4	5	6	**7**	8	9	10	**11**	12	13	14	**15**

Problema 12

```
b=0; i=0;
//difusión del vector NP y de x
broadcast(NP,M,tipo,0,grupo);
broadcast(x,1,tipo,0,grupo);

//Cálculo
while ((b==0)&& (i<M)) do {
   if (NP[i+idproc]==x) b=1;
  i=i+num_procesos;
}
reduction(b,b,1,tipo,OR,0,grupo);
if (idproc==0)
    if (b==1) printf("%d ES primo", x);
elseprintf("%d NO es primo", x);
```



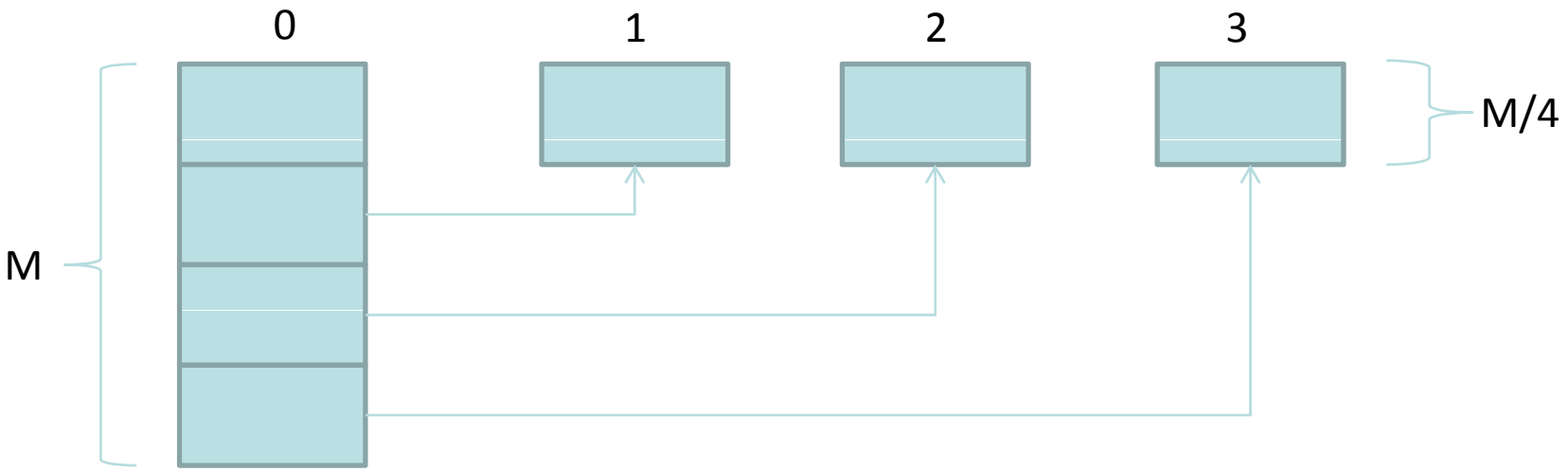
Broadcast desde 0



Reducción en 0

Problema 13

```
b=0; i=0;
//difusión del vector NP y de x
scatter(NP,M,NPl, M/num_procesos,tipo,0,grupo);
broadcast(x,1,tipo,0,grupo);

//Cálculo
while ((b==0)&& (i<M/num_procesos)) do {
   if (NPl[i]==x) b=1;
  i=i+1;
}
reduction(b,b,1,tipo,OR,0,grupo);
if (idproc==0)
    if (b==1) printf("%d ES primo", x);
elseprintf("%d NO es primo", x);
```



Problema 14

Versión 1

```
   b=0; i=0;

//solo un thread escribe en b
#pragma omp parallel\
private(idthread){
intilocal;
   idthread=omp_get_thread_num();
   nthreads=omp_get_num_threads();
   #pragma omp critical {
   ilocal=i; i++;
   }
   while ((b==0)&& (ilocal<M)) do {
      if (NP[ilocal]==x) b=1;
     #pragma ompcritical
     ilocal=i; i++;
     }
   }
}

   if (b==1) printf("%d ES primo", x);
   elseprintf("%d NO es primo", x);
```

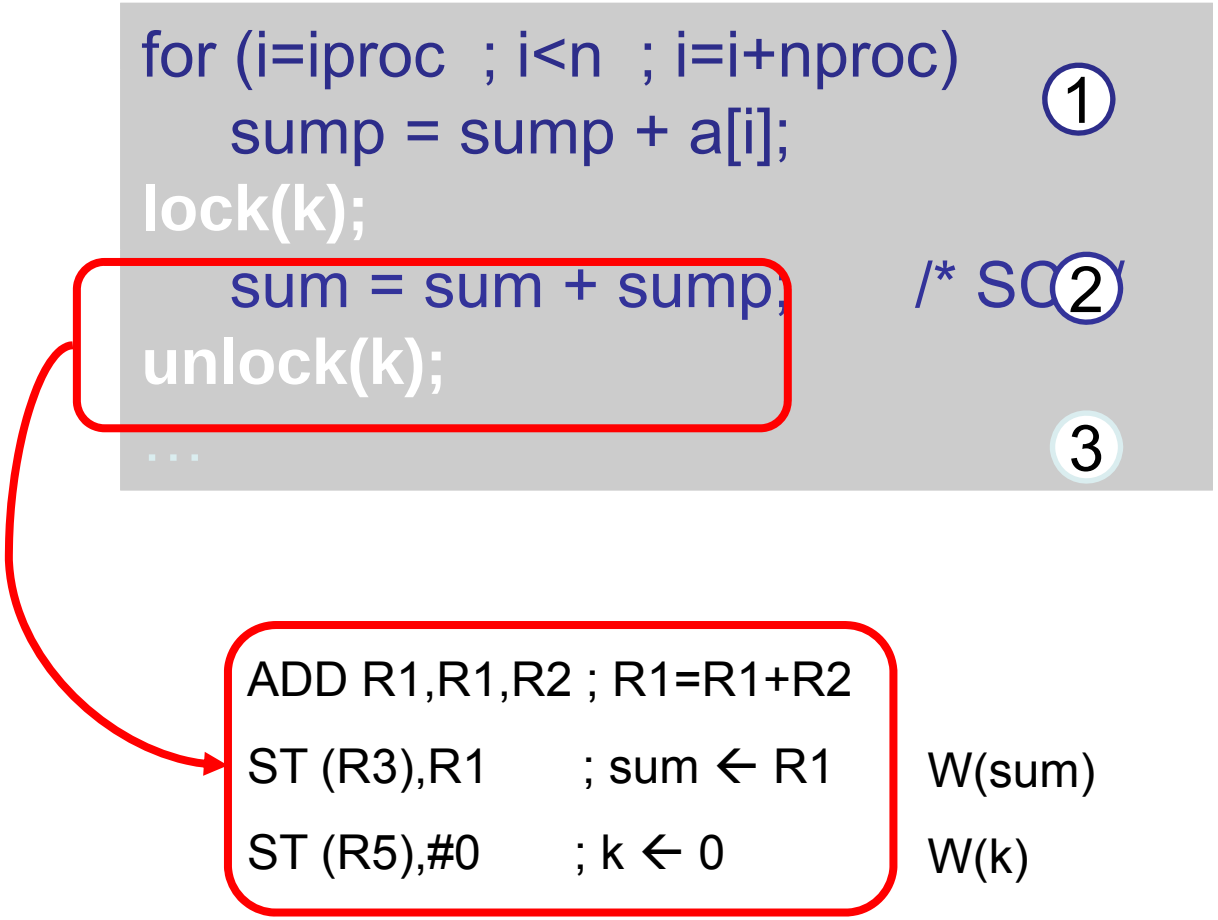
Versión 2

```
   b=0;
#pragma omp parallel for \
reduction(b:|)
   for(i=0;i<M;i++)  {
      if (NP[i]==x) b=1;
   }

   if (b==1) printf("%d ES primo", x);
   elseprintf("%d NO es primo", x);
```

Ejercicio 6. Tema 3. Se quiere implementar un cerrojo simple en un multiprocesador SMP basado en procesadores de la línea x86 de Intel, en particular, procesadores Intel Core.

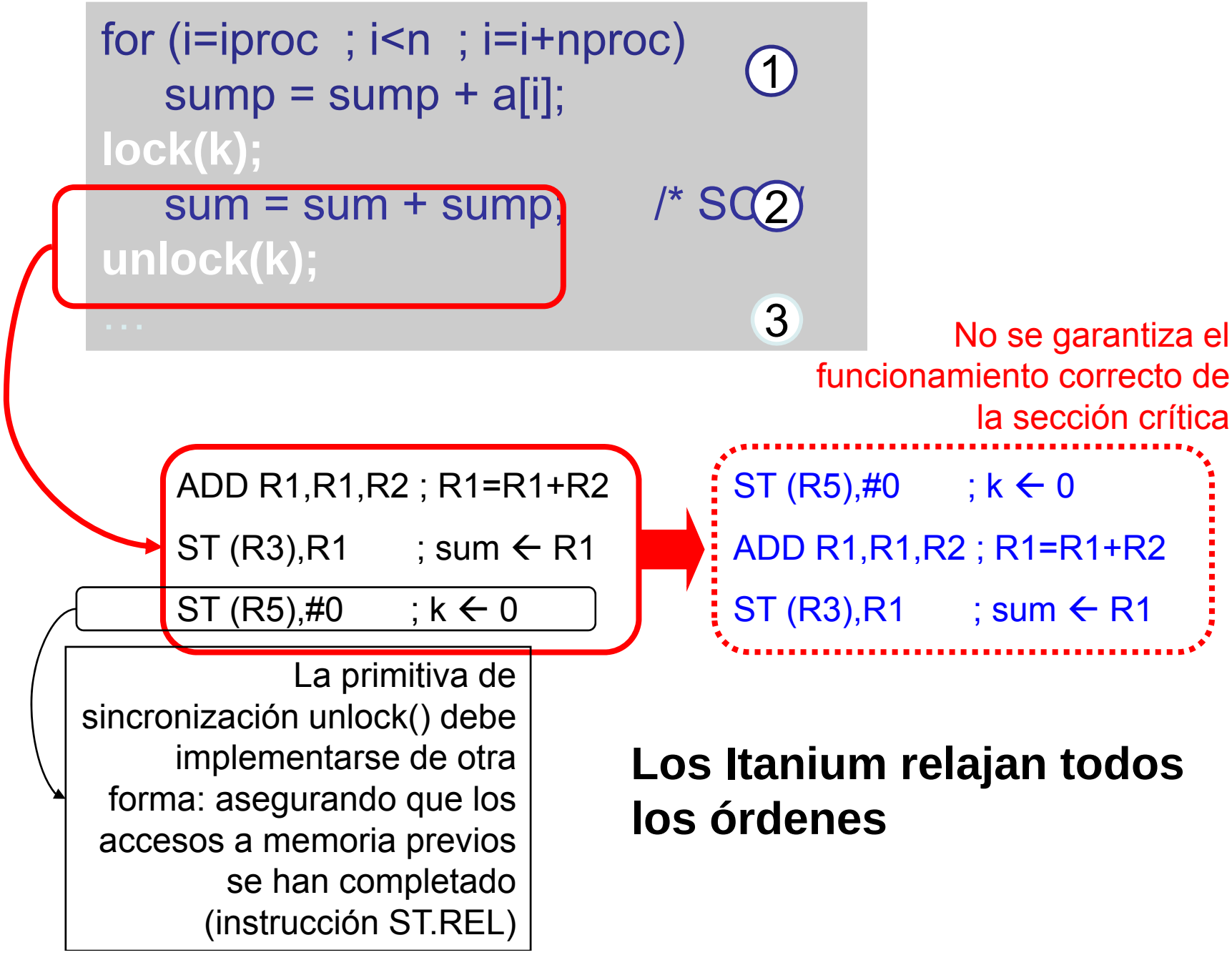
(a) Teniendo en cuenta el modelo de consistencia de memoria que ofrece el hardware de este multiprocesador ¿podríamos implementar la función de liberación del cerrojo simple usando “mov k, 0”, siendo k la variable cerrojo? Razone su respuesta.

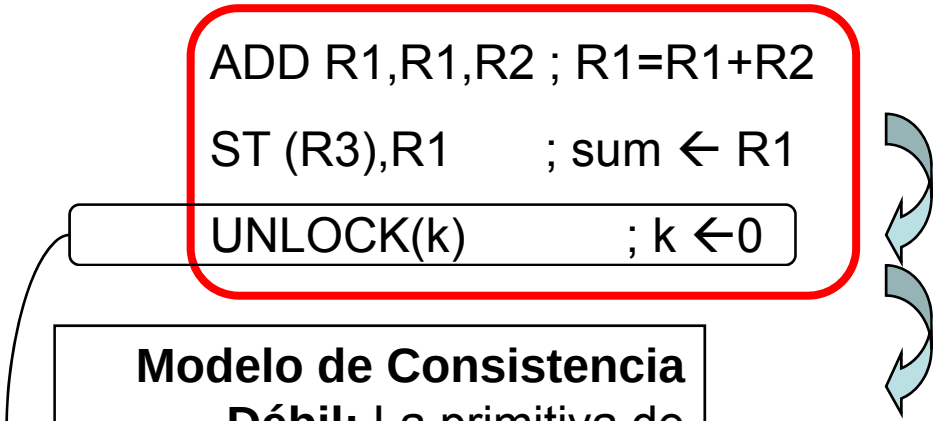
(b) ¿Cómo se debería implementar la función de liberación de un cerrojo simple si se usan procesadores Itanium? Razone su respuesta.



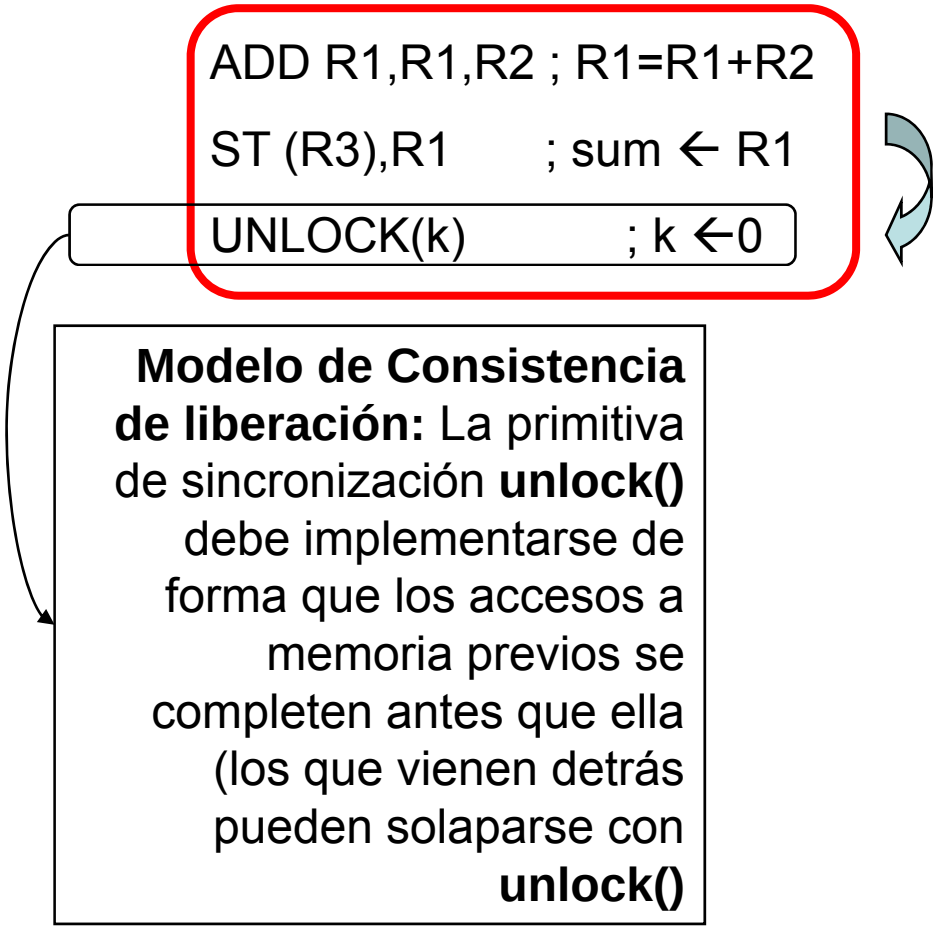
X86: solo relaja W → R, pero no relaja W → W

Por lo tanto, no hay problema


for (i=iproc ; i<n ; i=i+nproc)
sump = sump + a[i];
lock(k);
sum = sum + sump;
unlock(k);
...
1
2
3
No se garantiza el funcionamiento correcto de la sección crítica
ADD R1,R1,R2 ; R1=R1+R2
ST (R3),R1 ; sum ← R1
ST (R5),#0 ; k ← 0
ST (R5),#0 ; k ← 0
ADD R1,R1,R2 ; R1=R1+R2
ST (R3),R1 ; sum ← R1
La primitiva de sincronización unlock() debe implementarse de otra forma: asegurando que los accesos a memoria previos se han completado (instrucción ST.REL)
Los Itanium relajan todos los órdenes



Modelo de Consistencia Débil: La primitiva de sincronización **unlock()** debe implementarse de forma que los accesos a memoria previos se complenten antes que ella y los que vienen detrás solo empiecen cuando termine **unlock()**


ADD R1,R1,R2 ; R1=R1+R2
ST (R3),R1 ; sum ← R1
UNLOCK(k) ; k ←0
Modelo de Consistencia de liberación: La primitiva de sincronización **unlock()** debe implementarse de forma que los accesos a memoria previos se completen antes que ella (los que vienen detrás pueden solaparse con **unlock()**

Ejercicio 7. Se ha ejecutado el siguiente código en un multiprocesador con un modelo de consistencia que no garantiza ni W->R ni W->W (garantiza el resto de órdenes):

```
sump = 0;
for (i=ithread  ; i<8  ; i=i+nthread) {
        sump = sump + a[i];
    }
    while (Fetch_&_Or(k,1)==1)  {};
    sum = sum + sump;
    k=0;
```

(a) Indique qué se puede obtener en sum si se suma la lista a={1,2,3,4,5,6,7,8}. k y sum son variables compartidas que están inicialmente a 0 (el resto de variables son privadas), nthread = 3 y ithread es el identificador del thread en el grupo (0,1,2). Si hay varios posibles resultados, se tienen que dar todos ellos. Justifique su respuesta.

(b) ¿Qué resultados se pueden obtener si lo único que no garantiza el modelo de consistencia es el orden W->R? Justifique su respuesta.

0 (1) 1 (2) 2 (3)

3 (4) 4 (5) 5 (6)

6 (7) 7 (8)

12 15 9

Thread 0 1 2

```
sump = 0;
for (i=ithread  ; i<8  ; i=i+nthread) {
    sump = sump + a[i];
}
while (Fetch_&_Or(k,1)==1)  {};
sum = sum + sump;
k=0;
```

No hay problemas con Fetch_&_Or(k,1) porque es una operación atómica que contiene R (también tiene W) y ni las R ni las W pueden adelantar a R anteriores en el orden del programa.

Se podría liberar el cerrojo antes de que se hubiera completado la escritura de la suma parcial

Se podrían observar distintas combinaciones de sumas parciales

	resultado	comentario
(R0W0)(R1W1)(R2W2)	**12+15+9=36**	Si no hay problemas debido a que la escritura de liberación adelante a los accesos a `sum` anteriores
RxRxRxWxWxW0	**12**	Si los tres flujos de control llegan a leer el valor 0 inicial de `sum` y el último que escribe en `sum` tras actualizar su valor es el thread 0
RxRxRxWxWxW1	**15**	Si los tres flujos de control llegan a leer el valor 0 inicial de `sum` y el último que escribe en `sum` es el thread 1
RxRxRxWxWxW2	**9**	Si los tres flujos de control llegan a leer el valor 0 inicial de `sum` y el último que escribe en `sum` es el thread 2
R0W0RxRxW2W1 R0W0RxRxW1W2	**12+15=27 o 12+9=21**	Si el flujo de control 0 ha logrado acumular primero sin problemas y el 1 y el 2 leen el valor que tiene `sum` tras la acumulación de 0. Si esto ocurre entonces 1 y 2 acceden al mismo valor, 12, le acumulan cada uno lo que han calculado y escriben el resultado de la acumulación en `sum`. El resultado será 27 si el último que escribe es 1 y 21 si el último que escribe es 2.
R1W1RxRxW2W0 R1W1RxRxW0W2	**15+12=27 o 15+9=24**	Si el flujo de control 1 ha logrado acumular primero sin problemas y el 0 y el 2 acceden al valor que tiene `sum` justo tras la acumulación de `1`. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 15, le acumulan cada uno lo que han calculado y escriben el resultado de la acumulación en `sum`. El resultado será 27 si el último que escribe es 0 y 24 si el último que escribe es 2.
R2W2RxRxW1W0 R2W2RxRxW0W1	**9+12=21 o 9+15=24**	Si el flujo de control 2 ha logrado acumular primero sin problemas y el 0 y el 1 acceden al valor que tiene `sum` justo tras la acumulación de `2`. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 9, le acumulan cada uno lo que han calculado y escriben el resultado de la acumulación en `sum`. El resultado será 21 si el último que escribe es 0 y 24 si el último que escribe es 1.

(b) Si se garantiza el orden W → W no hay problema: **k = 0** se hace después de **sum=sum+sump**

Ejercicio 8: ¿Qué ocurre si en el segundo código para implementar barreras 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?

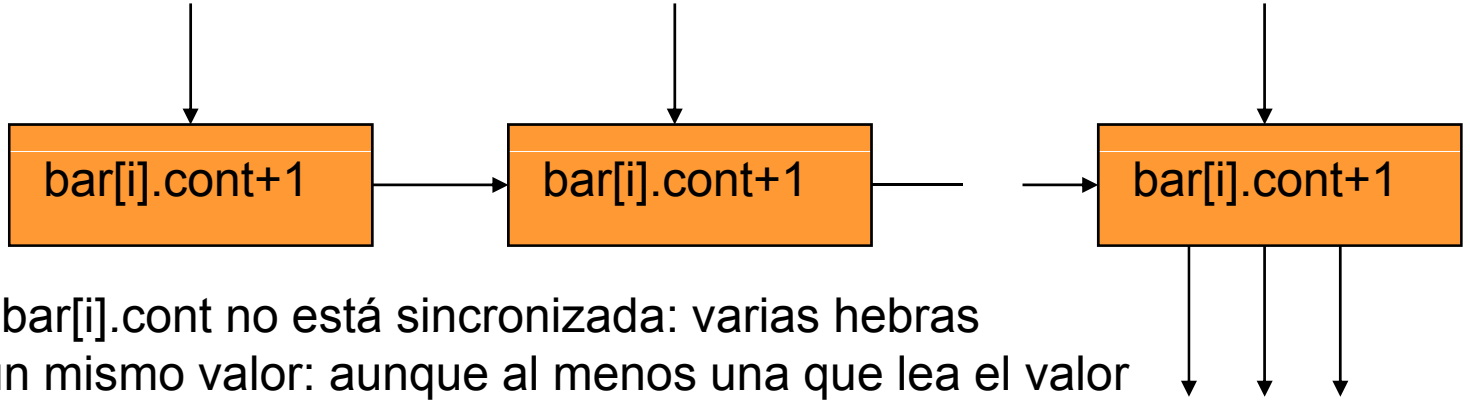
```
Barrera(id, num_procesos) {
  bandera_local = !(bandera_local)   //se complementa bandera local
  lock(bar[id].cerrojo);
      cont_local = ++bar[id].cont      //cont_local es privada
  unlock(bar[id].cerrojo);
  if (cont_local == num_procesos)  {
     bar[id].cont = 0;                    //se hace 0 el cont. de la barrera
     bar[id].bandera = bandera_local; //para liberar thread en espera
  }
  else  while (bar[id].bandera != bandera_local) {};   //espera ocupada
}
```

Para comprobar si se ha llegado a final, en el “if”, hay que acceder a una variable compartida y puede que vea que está en el valor final sin ser el último que la ha incrementado

```
Barrera(id, num_procesos) {
  bandera_local = !(bandera_local)   //se complementa bandera local
  lock(bar[id].cerrojo);
      cont_local = ++bar[id].cont      //cont_local es privada
  unlock(bar[id].cerrojo);
  if (cont_local == num_procesos)  {
     bar[id].cont = 0;                        //se hace 0 el cont. de la barrera
     bar[id].bandera = bandera_local; //para liberar thread en espera
  }
  else  while (bar[id].bandera != bandera_local) {};   //espera ocupada
}
```

bar[id].cont

Las hebras escriben en exclusión mutua (no hay problema el incremento del contador



La lectura de bar[i].cont no está sincronizada: varias hebras pueden leer un mismo valor: aunque al menos una que lea el valor igual a num_procesos

Un proceso puede encontrar que, al llegar al IF el contador compartido por todos es igual an número de procesos y él no es el último que ha incrementado el contador.

Si la barrera solo se utiliza una vez, el único problema es que los accesos de todos los procesadores que encuentran la condición de IF verdadera tendrán que acceder al contador global y se generarán varias escrituras

Si la barrera se puede reutilizar:

Un proceso puede ser suspendido por el SO cuando ha entrado en el IF pero no ha escrito 0 en el contador.

Si vuelve a estar activo cuando se está en otra ejecución de la barrera escribirá 0 en el contador compartido y éste no llegaría al valor final.

(Si no han salido de la barrera, da igual que se utilice la variable compartida)

```
Barrera(id, num_procesos) {
  bandera_local = !(bandera_local)   //se complementa bandera local
  lock(bar[id].cerrojo);
      cont_local = ++bar[id].cont      //cont_local es privada
  unlock(bar[id].cerrojo);
  if (cont_local == num_procesos) {
     bar[id].cont = 0;                    //se hace 0 el cont. de la barrera
     bar[id].bandera = bandera_local; //para liberar thread en espera
  }
  else  while (bar[id].bandera != bandera_local) {};   //espera ocupada
}
```

bar[id].cont

Se podría pasar la barrera porque varias hebras podrían leer bar[id].cont y alguna lo pondría a 0 (si solo queda esa hebra el problema es que todo quedaría detenido hasta que estuviera activa)

Pero si se vuelve a utilizar, y la hebra se “despierta” en este punto haría bar[id].con=0 y cambiaría la memoria local

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

```
Barrera(id, num_procesos) {
  bandera_local = !(bandera_local)   //se complementa bandera local
  lock(bar[id].cerrojo);
     cont_local = ++bar[id].cont       //cont_local es privada
  unlock(bar[id].cerrojo);
  if (cont_local == num_procesos)  {
     bar[id].cont = 0;                    //se hace 0 el cont. de la barrera
     bar[id].bandera = bandera_local; //para liberar thread en espera
  }
  else  while (bar[id].bandera != bandera_local) {};   //espera ocupada
}
```

cont_local=Fetch&Add(bar[id].cont,1)

num_procesos-1 (con el F&A se carga primero en contlocal y después se incrementa bar[i].cont)

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
}
```

Ejercicio 11

```
Barrera(id, num_procesos)
{
  band_local= !(band_local)
  while (fetch_&_or(k,1)==1) {};                  (R,W) atómica
            cont_local = ++bar[id].cont;          R seguida de W
  k=0;                                            W
  if (cont_local == num_procesos)  {
       bar[id].cont=0;
       bar[id].band=band_local;
  }
  else while (bar[id].band != band_local) {};
}
```

(a) No se garantiza W->R: Las escrituras (W) no pueden adelantar a escrituras previas (Garantiza: R->RW, W->W). No habría problema

(b) Ordenación débil (No garantiza R->RW, W->W, W->R): Da problemas en la escritura en k=0 (se ha visto antes). Tener en cuenta que las lecturas no adelantan a escrituras previas que van al mismo dato (dependencia RAW)

CASO a)

----- lock (acceso atómico: se adelantan las dos o ninguna)

R(k) fetch_&_or (k,1)

W(k)

R(bar[id].cont)

W(bar[id].cont)

W(k) ---- unlock

W(bar[id].cont)

W(bar[id].band)

NO EXISTE W->R:

Se mantiene el orden secuencial de los accesos

CASO a)

----- lock (acceso atómico: se adelantan las dos o ninguna)

R(k) fetch_&_or (k,1)

W(k)

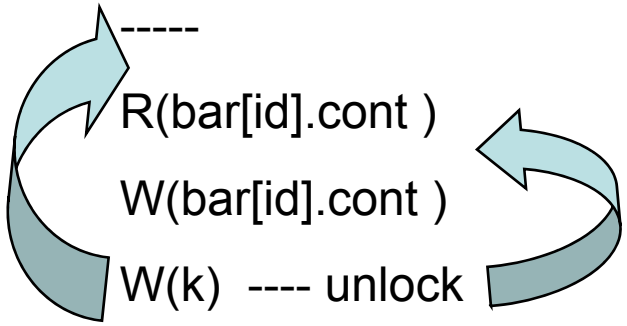
R(bar[id].cont)

W(bar[id].cont)

W(k) ---- unlock

W(bar[id].cont)

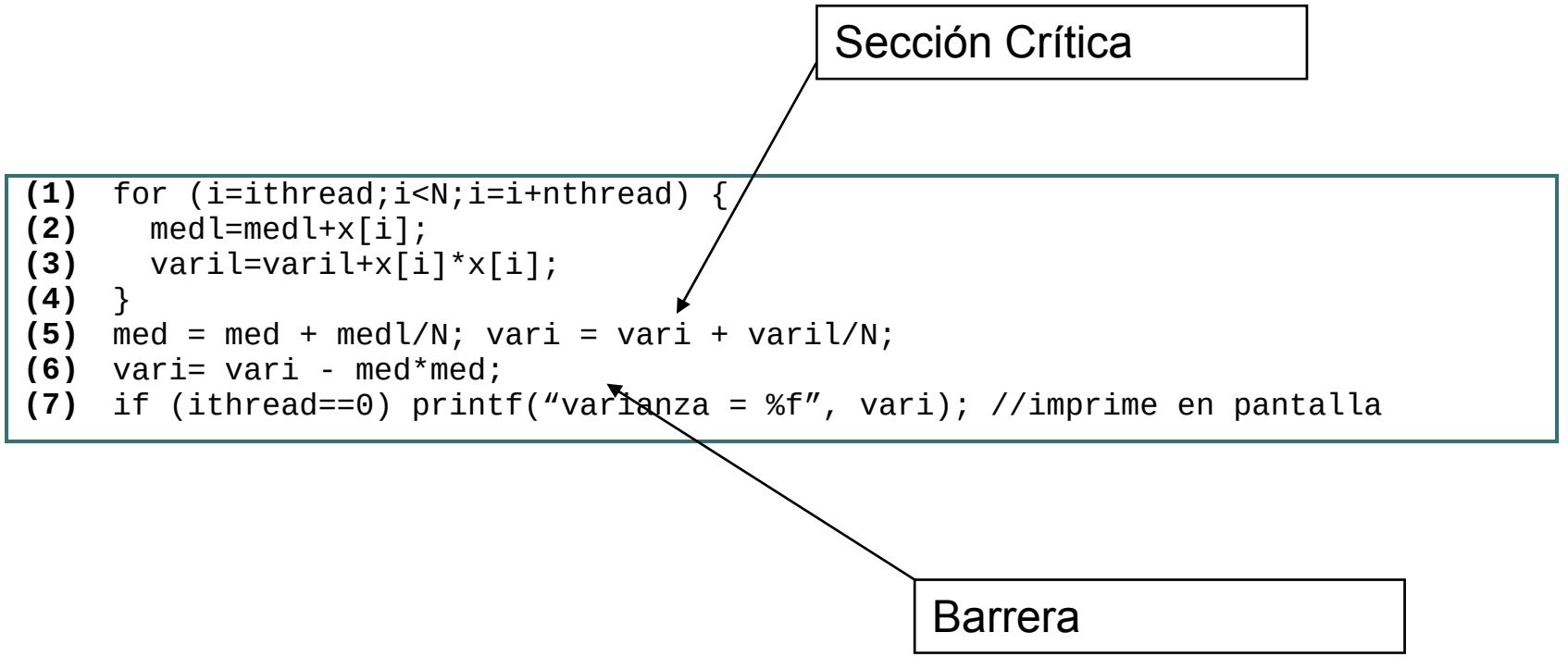
W(bar[id].band)



EXISTEN W->W y R->W:

La apertura de la barrera puede hacerse antes de que concluya la sección crítica

Ejercicio 12



```
(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
     while (test_&_set(k1)) {};   //lock(k1)
(5)  med = med + medl/N; vari = vari + varil/N;
     k1=0;                        //unlock(k1)

     bandera_local= !(bandera_local)    //se complementa la bandera local
     while (test_&_set(k2)) {};     //lock(k2)
     cont_local = bar[id].cont+1;    //cont_local es local
     k2=0;                          //unlock(k2)
     if (cont_local ==num_procesos-1)  {
           bar[id].cont=0;                          //se   hace   0   el   contador
     asociado a la barrera
           bar[id].bandera= bandera_local;   // libera procesos en espera
     }
     else  while (bar[id].bandera!= bandera_local) {};
(6)
(7)  vari= vari - med*med;
     if (ithread==0) printf("varianza = %f", vari); //imprime en pantalla
```

```
(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
(5)  fetch_&_add(med,medl/N); fetch_&_add(vari,varl/N);

     bandera_local= !(bandera_local)    //se complementa la bandera local
     while (test_&_set(k2)) {};        //lock(k2)
     cont_local = fetch_&_add(bar[id].cont,1);   //cont_local es local
     if (cont_local==num_procesos-1)  {
           bar[id].cont=0;                          //se   hace   0   el   contador
     asociado a la barrera
           bar[id].bandera= bandera_local;   // libera procesos en espera
     }
     else  while (bar[id].bandera!= bandera_local) {};
(6)
(7)
     vari= vari - med*med;
     if (ithread==0) printf("varianza = %f", vari); //imprime en pantalla
```

```
(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
(5)  do
        a = med;
        b = a + medl/N;
        compare&swap(a,b,med);
     while (a!=b);
     do
        a = vari;
        b = a + varil/N;
        compare&swap(a,b,vari);
     while (a!=b);

     bandera_local= !(bandera_local)     //se complementa la bandera local
     while (test_&_set(k2)) {};     //lock(k2)
     do
        cont_local = bar[id].cont;
        b = cont_local + 1;
        compare&swap(cont_local,b,bar[id].cont);
     while (cont_local!=b);
     cont_local = fetch_&_add(bar[id].cont,1);    //cont_local es local
     if (cont_local==num_procesos-1)  {
            bar[id].cont=0;                          //se   hace   0   el   contador
     asociado a la barrera
            bar[id].bandera= bandera_local;    // libera procesos en espera
     }
     else  while (bar[id].bandera!= bandera_local) {};
(6)
(7)  vari= vari - med*med;
     if (ithread==0) printf("varianza = %f", vari); //imprime en pantalla
```

Tema 4. Problema 1

```
1.  lw r1,0x1ac ; r1 ← M(0x1ac)
2.  lw r2,0xc1f ; r2 ← M(0xc1f)
3.  add r3,r0,r0 ; r3 ← r0+r0
4.  mul r4,r2,r1 ; r4 ← r2*r1
5.  add r3,r3,r4 ; r3 ← r3+r4
6.  add r5,r0,0x1ac ; r5 ← r0+0x1ac
7.  add r6,r0,0xc1f ; r6 ← r0+0xc1f
8.  sub r5,r5,#4 ; r5 ← r5 - 4
9.  sub r6,r6,#4 ; r6 ← r6 - 4
10.  sw (r5),r3 ; M(r5) ← r3
11.  sw (r6),r4 ; M(r6) ← r4
```

Ventana centralizada con emisión ordenada

INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw r1, 0x1ac	IF	ID	EX	EX										
lw r2, 0xc1f	IF	ID			EX	EX								
add r3, r0, r0	IF	ID			EX									
mul r4, r2, r1	IF	ID					EX	EX	EX	EX				
add r3, r3, r4		IF	ID								EX			
add r5, r0, 0x1ac		IF	ID								EX			
add r6, r0, 0xc1f		IF	ID								EX			
sub r5, r5, #4		IF	ID									EX		
sub r6, r6, #4			IF	ID								EX		
sw (r5), r3			IF	ID									EX	
sw (r6), r4			IF	ID										EX

Ventana de instrucciones centralizada con emisión desordenada

INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12
lw r1, 0x1ac	IF	ID	EX	EX								
lw r2, 0xc1f	IF	ID			EX	EX						
add r3, r0, r0	IF	ID	EX									
mod r4, r2, r1	IF	ID					EX	EX	EX	EX		
add r3, r3, r4		IF	ID								EX	
add r5, r0, 0x1ac		IF	ID	EX								
add r6, r0, 0xc1f		IF	ID	EX								
sub r5, r5, #4		IF	ID		EX							
sub r6, r6, #4			IF	ID	EX							
sw (r5), r3			IF	ID								EX
sw (r6), r4			IF	ID							EX	

Estación de reserva con tres líneas para cada unidad funcional y envío ordenado

ESTACIÓN DE RESERVA	INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LD	*lw r1, 0x1ac*	IF	ID	EX												
LD	*lw r2, 0xc1f*	IF	ID			EX										
ADD(1)	*add r3, r0, r0*	IF	ID	EX												
MULT(1)	*mul r4, r2, r1*	IF	ID					EX								
ADD(2)	*add r3, r3, r4*		IF	ID										EX		
ADD(3)	*add r5, r0, 0x1ac*		IF	ID	EX											
ADD(1)	*add r6, r0, 0xc1f*		IF	ID	EX											
ADD(3)	*sub r5, r5, #4*		IF	ID		EX										
ADD(1)	*sub r6, r6, #4*			IF	ID	EX										
ST	*sw (r5)„ r3*			IF	ID										EX	
ST	*sw (r6)„ r4*			IF	ID											EX

Problema 2

```
1.      lw      r3,0x10a        ; r3 ← M(0x10a)
2.      addi    r2,r0,#128      ; r2 ← r0+128
3.      add     r1,r0,0x0a      ; r1 ← r0+0x0a
4.      lw      r4,0(r1)        ; r4 ← M(r1)
5.      lw      r5,-8(r1)       ; r5 ← M(r1-8)
6.      mult    r6,r5,r3        ; r6 ← r5*r3
7       add     r5,r6,r3        ; r5 ← r6+r3
8       add     r6,r4,r3        ; r6 ← r4+r3
9       sw      0(r1),r6        ; M(r1) ← r5
10.     sw      -8(r1),r5       ; M(r1-8) ← r5
11.     sub     r2,r2,#16       ; r2 ← r2-16
```

Emisión Ordenada

INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
lw r3, 0x10a	IF	ID	EX		ROB	WB													
addi r2, r0, #128	IF	ID	EX	ROB		WB													
add r1, r0, 0x0a	IF		ID	EX	ROB	WB													
lw r4, 0(r1)	IF		ID		EX		ROB	WB											
lw r5, −8(r1)		IF		ID		EX			ROB	WB									
muld r6, r5, r3		IF		ID			EX				ROB	WB							
add r5, r6, r3		IF			ID								EX	ROB	WB				
add r6, r4, r3		IF			ID								EX	ROB	WB				
sw 0(r1), r6			IF			ID								EX	ROB	WB			
sw −8(r1), r5			IF			ID									EX	ROB	WB		
subi r2, r2, #16			IF				ID								EX	ROB	WB		

Tiene que comprobarse que no se decodifican, emiten, ni escriben en el ROB más de dos instrucciones por ciclo, ni se retiran más de tres instrucciones por ciclo.

Emisión Desordenada

INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
lw r3, 0x10a	IF	ID	EX		ROB	WB												
addi r2, r0, #128	IF	ID	EX	ROB		WB												
add r1, r0, 0x0a	IF		ID	EX	ROB		WB											
lw r4, 0(r1)	IF		ID		EX		ROB	WB										
lw r5, −8(r1)		IF		ID			EX		ROB	WB								
mult r6, r5, r3		IF		ID					EX						ROB	WB		
add r5, r6, r3		IF			ID										EX	ROB	WB	
add r6, r4, r3		IF			ID		EX	ROB									WB	
sw 0(r1), r6			IF			ID		EX	ROB								WB	
sw −8(r1), r5			IF			ID										EX	ROB	WB
sub r2, r2, #16			IF				ID	EX		ROB								WB

También en este caso hay que tener en cuenta que no se pueden decodificar, emitir, ni escribir en el ROB, más de dos instrucciones por ciclo (obsérvese que la instrucción sw r2,r2,#16 debe esperar un ciclo para su etapa ROB por esta razón), ni se pueden retirar más de tres instrucciones por ciclo.

3. a)

INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
lw r3, 0x10a	IF	ID	EX		ROB	WB									
addi r2, r0, #128	IF	ID	EX	ROB		WB									
add r1, r0, 0x0a	IF	ID	EX	ROB		WB									
lw r4, 0(r1)	IF	ID		EX		ROB	WB								
lw r5, −8(r1)		IF	ID	EX		ROB	WB								
mult r6, r5, r3		IF	ID			EX						ROB	WB		
add r5, r6, r3		IF	ID									EX	ROB	WB	
add r6, r4, r3		IF	ID			EX	ROB							WB	
sw 0(r1), r6			IF	ID			EX	ROB						WB	
sw −8(r1), r5			IF	ID									EX	ROB	WB

Se decodifican el mismo número de instrucciones que se captan.

No existen limitaciones para el número de instrucciones por ciclo que se emiten, escriben el ROB, y se retiran.

Están disponibles todas las unidades funcionales que se necesiten para que no haya colisiones (riesgos estructurales).

Se reduce el tiempo del multiplicador

INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12
lw r3, 0x10a	IF	ID	EX		ROB	WB						
addi r2, r0, #128	IF	ID	EX	ROB		WB						
addi r1, r0, 0x0a	IF	ID	EX	ROB		WB						
lw r4, 0(r1)	IF	ID			EX	ROB	WB					
lw r5, −8(r1)		IF	ID		EX	ROB	WB					
mult r6, r5, r3		IF	ID			EX			ROB	WB		
add r5, r6, r3		IF	ID						EX	ROB	WB	
add r6, r4, r3		IF	ID			EX	ROB				WB	
sw 0(r1), r6			IF	ID			EX	ROB			WB	
sw −8(r1), r5			IF	ID						EX	ROB	WB
subi r2, r2, #16			IF	ID	EX	ROB						WB

$$T(n) = 12 = \mathrm{TLI} + (n-1) \times \mathrm{CPI} = 6 + (11-1) \times \mathrm{CPI}$$

CPI=0.6

Ejercicio 10

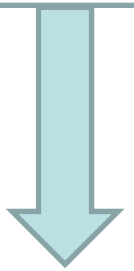
if (A>B) then { X=1;}
else {if (C<D) then X=2; else X=3;}

```
                lw      r1, a       ; r1 = A
                lw      r2, b       ; r2 = B
        p1, p2  cmp.gt  r1,r2       ; Si A > B  p1 = 1 y p2 = 0 (si no, p1 = 0 y p2 = 1)
(p1)            addi    r5, r0, #1
        p3      cmp.ne  r0, r0      ; Inicializamos p3 a 0
        p4      cmp.ne  r0, r0      ; Inicializamos p4 a 0
(p2)            lw      r3, c       ; r3 = C
(p2)            lw      r4, d       ; r4 = D
(p2)    p3, p4  cmp.lt  r3, r4      ; Sólo si p2 = 1 p3 o p4 pueden ser 1
(p3)            addi    r5, r0, #2  ; Se ejecuta si p3 = 1 (y p2 = 1)
(p4)            addi    r5, r0, #3  ; Se ejecuta si p4 = 1 (y p2 = 1)
                sw      (x),r5      ; Almacenamos el resultado
```

TEMA 4

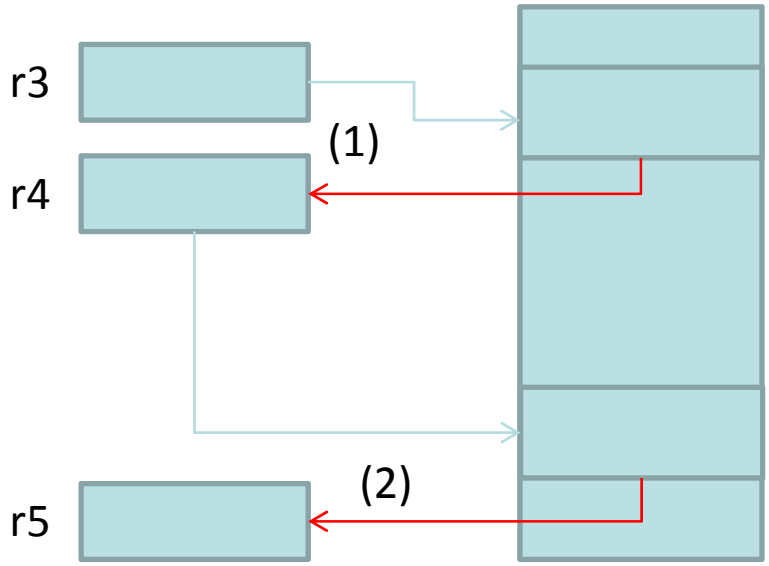
Ejercicio 9

#	OP1	OP2
1	lw r1, x(r2)	add r10, r11, r12
2		add r13, r10, r14
3	beqz r3, direc	
4	lw r4, 0(r3)	
5	lw r5, 0(r4)	



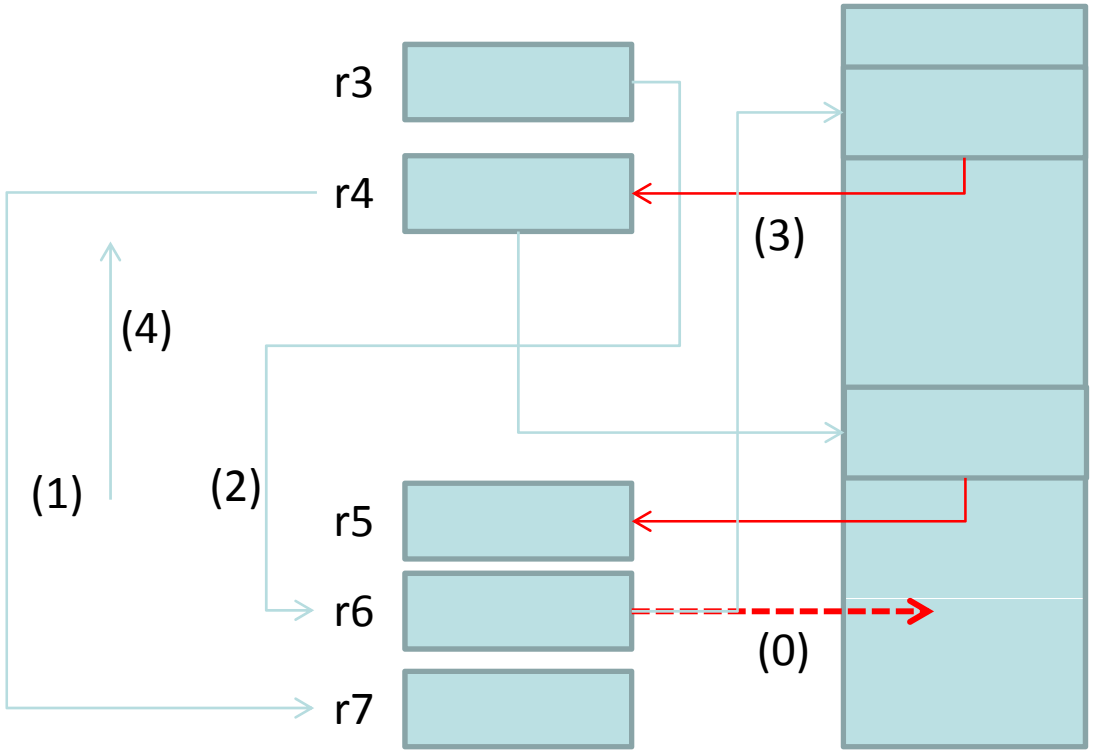
```
lw      r1, x(r2)   ; (1)
nop                 ; (2)
beqz    r3, direc   ; (3)
lw      r4, 0(r3)   ; (4)
lw      r5, 0(r4)   ; (5)
```

```
     lw      r1, x(r2)   ; (1)
     nop                 ; (2)
     beqz    r3, direc   ; (3)
(1)  lw      r4, 0(r3)   ; (4)
(2)  lw      r5, 0(r4)   ; (5)
```




```
addi    r6, r0, #1000       ; Fijamos r6 a una dirección segura
lw      r1, x(r2)
mov     r7, r4              ; Guardamos el contenido original de r4 en r7
cmovnz  r6, r3, r3          ; Movemos r3 a r6 si r3 es distinto de cero
lw      r4, 0(r6)           ; Carga especulativa
cmovz   r4, r7, r3          ; Si r3 es 0, hay que hacer que r4 recupere su valor
beqz    r3, direc
lw      r5, 0(r4)   ; Si r3 no es cero, hay que cargar r5
```

```
(0)  addi   r6, r0, #1000      ; Fijamos r6 a una dirección segura
     lw     r1, x(r2)
(1)  mov    r7, r4             ; Guardamos el contenido original de r4 en r7
(2)  cmovnz r6, r3, r3         ; Movemos r3 a r6 si r3 es distinto de cero
(3)  lw     r4, 0(r6)          ; Carga especulativa
(4)  cmovz  r4, r7, r3         ; Si r3 es 0, hay que hacer que r4 recupere su valor
     beqz   r3, direc
     lw     r5, 0(r4)  ; Si r3 no es cero, hay que cargar r5
```



Si r6 no se carga con r3 en **cmovnz** como luego se hace lw hay que asegurarse que esa dirección no genera excepción

```
addi    r6, r0, #1000     ; Fijamos r6 a una dirección segura
lw      r1, x(r2)
mov     r7, r4            ; Guardamos el contenido original de r4 en r7
mov     r8, r5            ; Guardamos r5 en otro registro temporal r8
cmovnz  r6, r3, r3        ; Movemos r3 a r6 si r3 es distinto de cero
lw      r4, 0(r6)         ; Carga especulativa
lw      r5, 0(r4)         ; Esta carga también es especulativa
cmovz   r4, r7, r3        ; Si r3 es 0, hay que hacer que r4 recupere su valor
cmovz   r5, r8, r3        ; Si r3 es 0 hay que hacer que r5 recupere su valor
```

```
lw      r1, x(r2)   ; (1)
nop                 ; (2)
beqz    r3, direc   ; (3)
lw      r4, 0(r3)   ; (4)
lw      r5, 0(r4)   ; (5)
```


#	OP1	OP2
1	addi r6,r0,#1000	add r10, r11, r12
2	lw r1,x(r2)	add r13, r10, r14
3	cmovnz r6,r3,r3	mov r7,r4
4	lw r4, 0(r3)	mov r8,r5
5	lw r5, 0(r4)	
6	cmovz r4,r7,r3	cmov r5,r8,r3