

2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores. Algunos ejercicios resueltos

## Tema 1. Arquitecturas Paralelas: Clasificación y Prestaciones

Material elaborado por los profesores responsables de la asignatura:  
Mancia Anguita, Julio Ortega

Licencia Creative Commons



## 1 Ejercicios

### Ejercicio 1. ◀

**Ejercicio 2.** (Tenga en cuenta que la redacción de este ejercicio no coincide totalmente con la redacción del ejercicio 2 de la relación de problemas del tema 2) En un procesador no segmentado que funciona a 300 MHz, hay un 20% de instrucciones LOAD que necesitan 4 ciclos, un 10% de instrucciones STORE que necesitan 3 ciclos, un 25% de instrucciones con operaciones de enteros que necesitan 6 ciclos, un 15% de instrucciones con operandos en coma flotante que necesitan 8 ciclos, y un 30% de instrucciones de salto que necesitan 3 ciclos. Las operaciones con enteros se realizan en una ALU con un retardo de 3 ciclos, y las de coma flotante en una unidad funcional con un retardo de 5 ciclos.

**(a)** ¿Cuál es la máxima ganancia que se puede obtener por reducción en el tiempo en la ALU de las operaciones con enteros? y **(b)** ¿cuál es la máxima ganancia que se puede obtener por reducción en el tiempo en la unidad funcional de las operaciones en coma flotante?

### Solución

Datos del ejercicio:

Tipo i de instr.	CPI <sub>i</sub> (c/i)	NI <sub>i</sub> /NI	Comentarios
LOAD	4	0,20	
STORE	3	0,10	
ENTEROS	6	0,25	Con retardo de 3 ciclos en ALU
FP	8	0,15	Con retardo de 5 ciclos en unidad funcional
BR	3	0,30	
TOTAL		100	

La frecuencia de reloj no hará falta.

La ley de Amdahl establece que la máxima ganancia de velocidad,  $S$ , que se puede conseguir al mejorar la velocidad de un recurso en un factor  $p$  depende de la fracción  $f$  del tiempo de procesamiento sin la mejora en la que NO se puede aprovechar dicha mejora según la expresión:

$$S \leq p/(1+f(p-1))$$

La máxima ganancia que se podría conseguir debido a mejoras en algún recurso se podría obtener aplicando el límite cuando  $p$  tiende a infinito a la expresión de la ley de Amdahl. Es decir que  $S_{\max} \leq (1/f)$



Por lo tanto, para resolver el problema utilizando esta expresión habría que evaluar los valores de  $f$  en cada uno de los casos que se indican.

Tipo i de instr.	CPI <sub>i</sub> (c/i)	NI <sub>i</sub> /NI	CPI <sub>i</sub> con mejora max. ENT. en ALU	CPI <sub>i</sub> con mejora max. FP en UF
LOAD	4	0,20	4	4
STORE	3	0,10	3	3
ENTEROS	6	0,25	3 (se eliminan los 3 de la ALU)	6
FP	8	0,15	8	3 (se eliminan los 5 de la UF)
BR	3	0,30	3	3
TOTAL		100		

El tiempo total de procesamiento antes de la mejora es igual a:

$$T_{\text{sin\_mejora}} = (NI \cdot 0.2) \cdot 4 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.1) \cdot 3 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.25) \cdot 6 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.15) \cdot 8 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.3) \cdot 3 \cdot (T_{\text{ciclo}}) = CPI \cdot NI \cdot T_{\text{ciclo}}$$

donde NI es el número de instrucciones y  $T_{\text{ciclo}} = 1/\text{frecuencia}$

**(a)** En el caso de mejoras en la ALU de enteros, el tiempo en el que no se puede aplicar la mejora sería (se supone que el tiempo de procesamiento de la operación en la ALU es 0 ciclos):

$$T_{\text{mejora\_ent}} = (NI \cdot 0.2) \cdot 4 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.1) \cdot 3 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.25) \cdot 3 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.15) \cdot 8 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.3) \cdot 3 \cdot (T_{\text{ciclo}})$$

y por tanto

$$S_{\text{max}} \leq (1/f) = T_{\text{mejora\_ent}}/T_{\text{sin\_mejora}} = 0.2 \cdot 4 + 0.1 \cdot 3 + 0.25 \cdot 6 + 0.15 \cdot 8 + 0.3 \cdot 3 / 0.2 \cdot 4 + 0.1 \cdot 3 + 0.25 \cdot 6 + 0.15 \cdot 8 + 0.3 \cdot 3 = 4,7/3,95 = 1,8987$$

$$S_{\text{max}} \leq 1.19$$

**(b)** En el caso de mejoras en operaciones de coma flotante, el tiempo en el que no se puede aplicar la mejora sería (se supone que el tiempo de procesamiento de la operación en coma flotante es 0 ciclos):

$$T_{\text{mejora\_fp}} = (NI \cdot 0.2) \cdot 4 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.1) \cdot 3 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.25) \cdot 6 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.15) \cdot 3 \cdot (T_{\text{ciclo}}) + (NI \cdot 0.3) \cdot 3 \cdot (T_{\text{ciclo}})$$

y por tanto

$$S_{\text{max}} \leq (1/f) = T_{\text{mejora\_fp}}/T_{\text{sin\_mejora}} = 0.2 \cdot 4 + 0.1 \cdot 3 + 0.25 \cdot 6 + 0.15 \cdot 8 + 0.3 \cdot 3 / 0.2 \cdot 4 + 0.1 \cdot 3 + 0.25 \cdot 6 + 0.15 \cdot 3 + 0.3 \cdot 3 = 4,7/3,95$$

$$S_{\text{max}} \leq 1.19$$

Como se puede comprobar se obtiene lo mismo puesto que  $(NI \cdot 0.25) \cdot 3 \cdot T_{\text{ciclo}}$ , que es el tiempo máximo que se puede reducir al acelerar las operaciones de enteros, es igual a  $(NI \cdot 0.15) \cdot 5 \cdot T_{\text{ciclo}}$ , que es el tiempo máximo que se puede reducir al acelerar las operaciones de coma flotante.



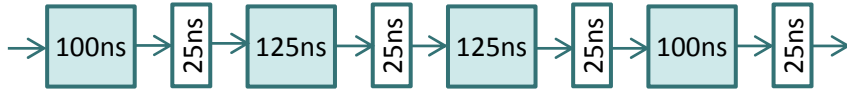
**Ejercicio 3.** Un circuito que implementaba una operación en  $T_{\text{op}}=450$  ns. se ha segmentado mediante un cauce lineal con cuatro etapas de duración  $T_1=100$  ns.,  $T_2=125$  ns.,  $T_3=125$  ns., y  $T_4=100$  ns. respectivamente, separadas por un registro de acoplo que introduce un retardo de 25 ns. **(a)** ¿Cuál es la



máxima ganancia de velocidad posible? ¿Cuál es la productividad máxima del cauce? **(b)** ¿A partir de qué número de operaciones ejecutadas se consigue una productividad igual al 90% de la productividad máxima?

### Solución

#### Datos del ejercicio:



Cauce del circuito

**(a)** El circuito se ha segmentado en cuatro etapas separadas por un registro de acoplo con un retardo de  $d=25$  ns. El tiempo de ciclo del cauce,  $t$ , se obtiene a partir de la expresión:

$$t = \max \{T_1, T_2, T_3, T_4\} + d = \max \{100, 125, 125, 100\} + 25 \text{ ns.} = 150 \text{ ns}$$

En ese cauce, una operación tarda un tiempo  $TLI = 4 * 150 \text{ ns.} = 600 \text{ ns.}$  (tiempo de latencia de inicio) en pasar por las cuatro etapas del mismo, y la ganancia de velocidad del cauce para  $n$  operaciones viene dada por:

$$S(n) = \text{Top} * n / (TLI + (n-1) * t) = 450 * n / (600 + (n-1) * 150)$$

El valor máximo de la ganancia de velocidad se obtiene aplicando el límite cuando  $n$  tiende a infinito. Es decir

$$S_{\max} = \lim_{n \rightarrow \infty} (\text{Top} * n / (TLI + (n-1) * t)) = \text{Top} / t = 450 / 150 = 3$$

**(b)** La productividad del cauce es  $W(n) = n / (TLI + (n-1) * t) = n / (600 + (n-1) * 150)$

La productividad máxima es  $W_{\max} = \lim_{n \rightarrow \infty} n / (TLI + (n-1) * t) = 1/t = 1/150 \text{ (op} * \text{ns}^{-1}) = 6.67 \text{ Mop/s.}$

El valor de  $n$  para el que se consigue una productividad igual al 90% de la productividad máxima es:

$$0.9 * (1/150) = n / (600 + (n-1) * 150) \rightarrow n = (0.9 * 4 + 0.9(n-1)) \rightarrow n * 0.1 = (0.9 * 3) \rightarrow \mathbf{n = 27}$$

Con 27 operaciones se alcanza el 90% de la productividad máxima. La productividad aumentará conforme se incremente  $n$ .



**Ejercicio 4.** En un procesador sin segmentación de cauce, determine cuál de estas dos alternativas para realizar un salto condicional es mejor:

- ALT1: Una instrucción COMPARE actualiza un código de condición y es seguida por una instrucción BRANCH que comprueba esa condición.

- ALT2: Una sola instrucción incluye la funcionalidad de las instrucciones COMPARE y BRANCH.

Hay que tener en cuenta que hay un 20% de saltos condicionales en ALT1; que las instrucciones BRANCH en ALT1 y COMPARE+BRANCH en ALT2 necesitan 4 ciclos mientras que todas las demás necesitan sólo 3; y que el ciclo de reloj de la ALT1 es un 25% menor que el de la ALT2, dado que en éste caso la mayor funcionalidad de la instrucción COMPARE+BRANCH ocasiona una mayor complejidad en el procesador.

### Solución

#### Datos del ejercicio:



Tiempo de ciclo de ALT1 =  $0,75 * \text{Tiempo de ciclo de ALT2} = \text{Tiempo de ciclo de ALT2} - 0,25 * \text{Tiempo de ciclo de ALT2}$

$$T_{\text{ciclo}}^1 = T_{\text{ciclo}}^2 - 0,25 * T_{\text{ciclo}}^2 = 0,75 * T_{\text{ciclo}}^2$$

ALT1

Instrucciones i	ciclos	NI <sub>i</sub> /NI <sup>1</sup>	Proporción (%)	Comentarios
BRANCH en ALT1	4	0,2	20	
RESTO en ALT1	3	0,8	80	Incluye instrucciones COMPARE
TOTAL			100	

ALT2

Instrucciones i	ciclos	NI <sub>i</sub> /NI <sup>1</sup>	Proporción (%)	Comentarios
COMPARE+BRANCH en ALT2	4	0,2	$20 * 100 / 80 = 25$	ALT2 no tendrá aparte las instrucciones COMPARE, que son un 20% en ALT1
RESTO en ALT2	3	0,6	$(80 - 20) * 100 / 80 = 75$	ALT2 no tendrá aparte las instrucciones COMPARE, que son un 20% en ALT1
TOTAL		0,8	100	NI <sup>2</sup> = 0,8 * NI <sup>1</sup> (al no tener instr. COMPARE aparte)

El tiempo de CPU se puede expresar como:

$$T_{\text{CPU}} = \text{CPI} * \text{NI} * T_{\text{CICLO}}$$

donde CPI es el número medio de ciclos por instrucción, NI es el número de instrucciones, y  $T_{\text{CICLO}}$  el tiempo de ciclo del procesador.

Para la alternativa primera ALT1, se tiene:

$$\text{CPI}(1) = 0,2 * 4 + 0,8 * 3 = 3,2$$

y por tanto

$$T_{\text{CPU}}(1) = \text{CPI}(1) * \text{NI}(1) * T_{\text{ciclo}}(1) = 3,2 * \text{NI}(1) * T_{\text{ciclo}}(1) = 3,2 * \text{NI}(1) * 0,75 * T_{\text{ciclo}}(2) = 2,4 * \text{NI}(1) * T_{\text{ciclo}}(2)$$

Para la alternativa primera ALT2, se tiene:

$$T_{\text{CPU}}(2) = \text{CPI}(2) * \text{NI}(2) * T_{\text{ciclo}}(2) = (0,2 * 4 + 0,6 * 3) * \text{NI}(1) / \text{NI}(2) * \text{NI}(2) * T_{\text{ciclo}}(2) = 2,6 * \text{NI}(1) * T_{\text{ciclo}}(2)$$

A continuación se calcula el tiempo de CPU para la segunda alternativa, ALT2, que se expresará en términos del número de instrucciones y del tiempo de ciclo de la alternativa ALT1.

Por una parte, se tiene que como en ALT2 no se ejecutan instrucciones COMPARE, se ejecutarán:

- **0.2 NI(1)** instrucciones BRANCH de ALT1 que ahora serán BRANCH+COMPARE
- **0.6 NI(1)** instrucciones que quedan al restar a las **0.8 NI(1)** de ALT1 las **0.2 NI(1)** instrucciones COMPARE (ya que había una por cada BRANCH y se tenían **0.2 NI(1)**)

Así, el número de instrucciones que quedan en ALT2 son

$$\text{NI}(2) = 0,2 \text{NI}(1) + 0,6 \text{NI}(1) = 0,8 \text{NI}(1)$$

y por lo tanto el valor de  $\text{CPI}(2)$  es:



$$CPI(2) = \frac{0.2 * NI(1) * 4 + 0.6 * NI(1) * 3}{0.8 * NI(1)} = \frac{0.8 + 1.8}{0.8} = 3.25$$

Además, se tiene que el ciclo de reloj de *ALT2* es más largo que el de *ALT1*. Concretamente, se dice que el ciclo de *ALT1* es un 25% menor que el de *ALT2*, lo que implica que:

$$T_{CICLO}(1) = 0.75 * T_{CICLO}(2) \rightarrow T_{CICLO}(2) = 1.33 * T_{CICLO}(1)$$

Sustituyendo  $NI(2)$ ,  $CPI(2)$ , y  $T_{CICLO}(2)$  en la expresión de  $T_{CPU}(2)$  se tiene que:

$$T_{CPU}(2) = 3.25 * (0.8 * NI(1)) * (1.33 * T_{CICLO}(1)) = 3.458 * NI(1) * T_{CICLO}(1)$$

Así, como  $T_{CPU}(1) < T_{CPU}(2)$  se tiene que *ALT2* no mejora a la *ALT1*. Es decir, aunque un repertorio con instrucciones más complejas puede reducir el número de instrucciones de los programas, reduciendo el número de instrucciones a ejecutar, esto no tiene que suponer una mejora de prestaciones si al implementar esta opción la mayor complejidad del procesador lo hace algo más lento.



**Ejercicio 5.** ¿Qué ocurriría en el problema anterior si el ciclo de reloj fuese únicamente un 10% mayor para la *ALT2*?

#### Solución

Como en el problema anterior, para la primera alternativa, *ALT1*, se tiene:

$$T_{CPU}(1) = 3.2 * NI(1) * T_{CICLO}(1)$$

Para la segunda alternativa, se tiene también que

$$NI(2) = 0.2NI(1) + 0.6NI(1) = 0.8NI(1)$$

y que  $CPI(2)$  es:

$$CPI(2) = \frac{0.2 * NI(1) * 4 + 0.6 * NI(1) * 3}{0.8 * NI(1)} = \frac{0.8 + 1.8}{0.8} = 3.25$$

No obstante, como ahora el tiempo de ciclo de *ALT2* es un 10% mayor que el de la *ALT1* se cumple que:

$$T_{CICLO}(2) = 1.10 * T_{CICLO}(1)$$

y sustituyendo,

$$T_{CPU}(2) = 3.25 * (0.8 * NI(1)) * (1.10 * T_{CICLO}(1)) = 2.86 * NI(1) * T_{CICLO}(1)$$

Así, ahora  $T_{CPU}(2) < T_{CPU}(1)$  y por lo tanto, la segunda alternativa sí que es mejor que la primera. Es decir, el mismo planteamiento que antes no mejoraba la situación de partida ahora sí lo consigue. Sólo ha sido necesario que el aumento del ciclo de reloj que se produce al hacer un diseño más complejo del procesador sea algo menor.

La situación reflejada en estos dos ejercicios (4 y 5) pone de manifiesto la necesidad de estudiar la arquitectura del computador desde un enfoque cuantitativo, y no sólo teniendo en cuenta razonamientos que pueden ser correctos, pero que pueden llevar a resultados finales distintos según el valor de las magnitudes implicadas.



**Ejercicio 6.** Considere un procesador no segmentado con una arquitectura de tipo LOAD/STORE en la que las operaciones sólo utilizan como operandos registros de la CPU. Para un conjunto de programas



representativos de su actividad se tiene que el 43% de las instrucciones son operaciones con la ALU (3 CPI), el 21% LOADs (4 CPI), el 12% STOREs (4 CPI) y el 24% BRANCHs (4 CPI).

Además, un 25% de las operaciones con la ALU utilizan operandos en registros, que no se vuelven a utilizar. ¿Se mejorarían las prestaciones si, para sustituir ese 25% de operaciones se añaden instrucciones con un dato en un registro y otro en memoria, teniendo en cuenta que para ellas el valor de CPI es 4 y que ocasionarían un incremento de un ciclo en el CPI de los BRANCHs, pero que no afectan al ciclo de reloj?

### Solución

#### Datos del ejercicio:

##### Alternativa 1

$I_i^1$	$CPI_i^1$	Fracción $f_i^1$ ( $NI_i^1 = f_i^1 * NI^1$ )	Comentarios
LOAD	4	0,21	
STORE	4	0,12	
ALU	3	0,43	25 % inst. que usan operados en registros que no se vuelven a utilizar
BR	4	0,24	
TOTAL		1	

##### Alternativa 2

$I_i^2$	$CPI_i^2$	Fracción $f_i$ ( $NI_i^2 = f_i^2 * NI^1$ )	Comentarios
LOAD	4	$0,21 - 0,25 * 0,43 = 0,1025$	El 25 % de 43% desaparece al usarse ese mismo número de operaciones con la ALU que acceden a memoria
STORE	4	0,12	
ALU r-r	3	$0,75 * 0,43 = 0,3225$	
ALU r-m	4	$0,25 * 0,43 = 0,1075$	25% de las operaciones con la ALU; es decir el 25% del 43%
BR	5	0,24	
TOTAL		0,8925	Es decir, $NI^2 = 0,8925 * NI^1$

En primer lugar se calcula el tiempo de CPU para la situación inicial. Para ello, se tiene que:

$$CPI(1) = 0,43 * 3 + 0,21 * 4 + 0,12 * 4 + 0,24 * 4 = 1,29 + 0,57 * 4 = 1,29 + 2,28 = 3,57$$

y, por tanto:

$$T_{CPU}(1) = CPI(1) * NI(1) * T_{CICLO}(1) = 3,57 * NI(1) * T_{CICLO}(1)$$

$$T_{CPU}(1) = CPI(1) * NI(1) * T_{ciclo} = 3,57 * NI(1) * T_{ciclo}$$

En segundo lugar se calcula el tiempo de CPU para la alternativa 2. Para ello, se tiene que:

$$CPI(2) = [0,3225 * 3 + (0,12 + 0,1025 + 0,1075) * 4 + 5 * 0,24] * NI(1) / NI(2) = (0,9675 + 0,33 * 4 + 0,24 * 5) * NI(1) / NI(2) = (0,9675 + 1,32 + 1,2) * NI(1) / NI(2) = 3,4875 * NI(1) / NI(2)$$

y, por tanto:

$$T_{CPU}(2) = CPI(2) * NI(2) * T_{ciclo} = 3,4875 * NI(1) / NI(2) * NI(2) * T_{ciclo} = 3,4875 * NI(1) * T_{ciclo}$$

En la Tabla anterior para Alternativa 2 se muestra que, al introducir las nuevas instrucciones de operación con la ALU con uno de los operandos en memoria:

- Se reduce el 25% de las  $0,43 * NI(1)$  instrucciones de operación con la ALU y operandos en registros a las que las nuevas instrucciones sustituyen.
- Se reduce en  $0,25 * 0,43 * NI(1)$  el número de LOADs, ya que según se indica en el enunciado, esos LOADs sólo están en el programa para cargar uno de los operandos de las operaciones con la ALU, que no se vuelven a utilizar nunca más.



- Hay que contabilizar las  $0.25 \cdot 0.43 \cdot NI(1)$  nuevas instrucciones de operación con la ALU que se introducen en el repertorio (y que sustituyen a las operaciones con la ALU y operandos en registros).
- Como resultado, al sumar todas las instrucciones que se tienen en la nueva situación, el número total de instrucciones se reduce (lógicamente, ya que se han ahorrado instrucciones LOADs), siendo igual a

$$NI(2) = 0.8925 \cdot NI(1)$$

Teniendo en cuenta la nueva distribución de instrucciones y sus nuevos CPIs, se tiene que:

$$CPI(2) = \frac{0.43 - 0.25 \cdot 0.43}{0.8925} \cdot 3 + \frac{0.21 - 0.25 \cdot 0.43}{0.8925} \cdot 4 + \frac{0.25 \cdot 0.43}{0.8925} \cdot 4 + \frac{0.12}{0.8925} \cdot 4 + \frac{0.24}{0.8925} \cdot 5 = 3.908$$

Como el tiempo de ciclo no varía se tiene que:

$$T_{CPU}(2) = CPI(2) \cdot NI(2) \cdot T_{CICLO}(2) = 3.908 \cdot 0.8925 \cdot NI(1) \cdot T_{CICLO}(1) = 3.488 \cdot NI(1) \cdot T_{CICLO}(1)$$

Como se puede ver, en este caso,  $T_{CPU}(1) > T_{CPU}(2)$ , y por lo tanto se mejoran las prestaciones, pero si el porcentaje de instrucciones sustituidas fuese un 20% en lugar de un 25%, en ese caso, se puede ver que

$$CPI(2) = 3.9302$$

$$NI(2) = 0.914 \cdot NI(1)$$

y, por tanto:

$$T_{CPU}(2) = CPI(2) \cdot NI(2) \cdot T_{CICLO}(2) = 3.9302 \cdot 0.914 \cdot NI(1) \cdot T_{CICLO}(1) = 3.59 \cdot NI(1) \cdot T_{CICLO}(1)$$

Ahora, en cambio, la segunda opción no mejora la primera. Como conclusión, se puede indicar que una determinada decisión de diseño puede suponer una mejora en el rendimiento del computador correspondiente según sean las características de las distribuciones de instrucciones en los programas que constituyen la carga de trabajo característica del computador.

Por tanto, queda clara también la importancia que tiene el proceso de definición de conjuntos de *benchmarks* para evaluar las prestaciones de los computadores, y las dificultades que pueden surgir para que los fabricantes se pongan de acuerdo en aceptar un conjunto de *benchmarks* estándar.



**Ejercicio 7.** Se ha diseñado un compilador para la máquina LOAD/STORE del problema anterior. Ese compilador puede reducir en un 50% el número de operaciones con la ALU, pero no reduce el número de LOADs, STOREs, y BRANCHs. Suponiendo que la frecuencia de reloj es de 50 Mhz. ¿Cuál es el número de MIPS y el tiempo de ejecución que se consigue con el código optimizado? Compárelos con los correspondientes del código no optimizado.

### Solución

Datos del ejercicio:

Alternativa 1

$I_i^1$	$CPI_i^1$	Fracción $f_i^1$ ( $NI_i^1 = f_i^1 \cdot NI^1$ )
LOAD	4	0,21
STORE	4	0,12
ALU	3	0,43
BR	4	0,24
TOTAL		1



## Alternativa 2

$I_i^2$	$CPI_i^2$	Fracción $f_i$ ( $NI_i^2 = f_i^2 * NI^1$ )	Comentarios
LOAD	4	0,21	
STORE	4	0,12	
ALU r-r	3	$0,5 * 0,43 = 0,215$	Se reducen las instrucciones que usan la ALU en un 50%
BR	4	0,24	
TOTAL		<b>0,785</b>	<b>Es decir, <math>NI^2 = 0,785 * NI^1</math></b>

En la situación inicial del problema anterior se tenía que

$$T_{CPU}(1) = CPI(1) * NI(1) * T_{CICLO}(1) = 3,57 * NI(1) * T_{CICLO}(1)$$

$$T_{CPU}(1) = CPI(1) * NI(1) * T_{ciclo} = 3,57 * NI(1) * T_{ciclo}$$

y, por lo tanto

$$MIPS(1) = \frac{NI(1)}{T_{CPU}(1)(sg.) * 10^6} = \frac{NI(1)}{(CPI(1) * NI(1) * T_{CICLO}(1)) * 10^6} = \frac{f(MHz)}{CPI(1)} = \frac{50}{3,57} = 14,005 MIPS$$

$$T_{CPU}(1) = CPI(1) * NI(1) * T_{ciclo} = 3,57 * NI(1) * T_{ciclo}$$

En segundo lugar se calcula el tiempo de CPU para la alternativa 2. Para ello, se tiene que:

$$CPI(2) = [0,215 * 3 + (0,21 + 0,12 + 0,24) * 4] * NI(1) / NI(2) = (0,645 + 0,57 * 4) * NI(1) / NI(2) = (0,645 + 2,28) * NI(1) / NI(2) = 2,925 * NI(1) / NI(2)$$

y, por tanto:

$$T_{CPU}(2) = CPI(2) * NI(2) * T_{ciclo} = 2,925 * NI(1) / NI(2) * NI(2) * T_{ciclo} = 2,925 * NI(1) * T_{ciclo}$$

$$MIPS(2) = NI(2) / (CPI(2) * NI(2) * T_{ciclo} * 10^6) = f(MHz) / CPI(2) = 50 * 0,785 / 2,925 MIPS = 13,4188 MIPS$$

En la Tabla de la Alternativa 2 se muestra que se reduce el número de operaciones con la ALU a la mitad, y que, por tanto, también se reduce el número total de instrucciones a ejecutar,  $NI(2)$ :

$$NI(2) = 0,785 * NI(1)$$

Teniendo en cuenta la nueva distribución de instrucciones y sus nuevos CPIs, se tiene que:

$$CPI(2) = \frac{0,43 - 0,43/2}{0,785} * 3 + \frac{0,21}{0,785} * 4 + \frac{0,12}{0,785} * 4 + \frac{0,24}{0,785} * 4 = 3,72$$

Como el tiempo de ciclo no varía se tiene que:

$$T_{CPU}(2) = CPI(2) * NI(2) * T_{CICLO}(2) = 3,72 * 0,785 * NI(1) * T_{CICLO}(1) = 2,92 * NI(1) * T_{CICLO}(1)$$

El número de MIPS para este caso es:

$$MIPS(2) = \frac{NI(2)}{T_{CPU}(2)(sg.) * 10^6} = \frac{NI(2)}{(CPI(2) * NI(2) * T_{CICLO}(2)) * 10^6} = \frac{f(MHz)}{CPI(2)} = \frac{50}{3,72} = 13,44 MIPS$$

Como se puede ver, se consigue una reducción de tiempo de ejecución ( $T_{CPU}(2)$  es un 21% menor que  $T_{CPU}(1)$ ), pero sin embargo, el número de MIPS para la segunda opción es menor. Se tiene aquí un ejemplo de que los MIPS dan una información inversamente proporcional a las prestaciones. La razón de esta situación, en este caso, es que se ha reducido el número de las instrucciones que tenían un menor valor de CPI. Así, se incrementan las proporciones de las instrucciones más lentas en el segundo caso (por eso crece el valor de CPI), y claro, el valor de los MIPS se reduce. No obstante, hay que tener en cuenta que aunque las instrucciones que se ejecutan son más lentas, hay que ejecutar un número mucho menor de instrucciones, y al final, el tiempo se reduce.







### Ejercicio 8. ◀

### Ejercicio 9. ◀

**Ejercicio 10.** Suponga que en el código siguiente  $N=10^9$ , y que  $a[]$  es un array de números de 32 bits en coma flotante y  $b$  es otro número de 32 bits en coma flotante:

```
for (i=0; i<N; i++) a[i+2]=(a[i+2]+a[i+1]+a[i])*b
```

Teniendo en cuenta que el programa debe completarse en menos de 0.5 segundos, conteste a las siguientes cuestiones:

- (a) ¿Cuántos GFLOPS se necesitan como mínimo suponiendo que el producto y la suma en coma flotante tienen un coste similar?
- (b) Suponga que el programa en ensamblador tiene  $7N$  instrucciones y que se ha ejecutado en un procesador de 32 bits a 2 GHz. ¿Cual es el número medio de instrucciones que el procesador tiene que poder completar por ciclo?.
- (c) Estimando que el programa pasa el 75% de su tiempo de ejecución realizando operaciones en coma flotante, ¿cuánto disminuiría como mucho el tiempo de ejecución si se redujesen un 75% los tiempos de las unidades de coma flotante?

### Solución

**(a)** Puesto que hay tres operaciones en coma flotante por iteración (dos sumas y un producto, con igual coste para el producto y la suma) el número de operaciones en coma flotante a realizar tras las  $N$  iteraciones es  $N_{\text{flot}}=3*N=3*10^9$ .

Por lo tanto,  $\text{GFLOPS} = N_{\text{flot}}/(t*10^9)$  donde  $t$  es el tiempo en segundos. Así:

$$\text{GFLOPS} = N_{\text{flot}}/(t*10^9) = 3*10^9/0.5*10^9 = 6 \text{ GFLOPS}$$

**(b)** Si el programa tiene  $NI=7N=7*10^9$  instrucciones, teniendo en cuenta que

$$T = NI * \text{CPI} * T_{\text{ciclo}} = NI * \text{CPI} / f = (7*10^9 \text{ instrucciones}) * \text{CPI} / (2*10^9 \text{ ciclos/s}) = 0.5 \text{ s}$$

Al despejar  $\text{CPI} = 0.5/3.5$ , y el número de instrucciones por ciclo  $\text{IPC} = 1/\text{CPI} = 7$

**(c)** Para resolver la última cuestión se puede recurrir a la ley de Amdahl. En este caso  $f=0.25$  es la fracción de tiempo en el que no se puede aprovechar la mejora; el incremento de velocidad es  $p=1/0.25$  (si el tiempo de las operaciones en coma flotante era  $t$ , el tiempo de las operaciones en coma flotante con la mejor es  $0.25t$ , dado que se ha reducido un 75%); y la mejora de velocidad que se observaría se obtiene aplicando la ley de Amdahl, sustituyendo convenientemente:

$$S \leq p / (1+f(p-1)) = 4 / (1+0.25*3) = 2.286$$

Como  $S = T_{\text{sin\_mejora}}/T_{\text{con\_mejora}} \leq 2.286$  se tiene que  $T_{\text{sin\_mejora}}/2.286 = 0.437*T_{\text{sin\_mejora}} \leq T_{\text{con\_mejora}}$ . Es decir, el tiempo con la mejora podría pasar a ser el 43.7% del tiempo sin la mejora.





### Ejercicio 11.



## 2 Cuestiones

**Cuestión 1.** Indique cuál es la diferencia fundamental entre una arquitectura CC-NUMA y una arquitectura SMP.

### Solución.

Ambos, SMP y CC-NUMA, son multiprocesadores, es decir, comparten el espacio de direcciones físico. También en ambos se mantiene coherencia entre caches de distintos procesadores (CC- *Cache-Coherence*).

En un SMP (*Symmetric Multiprocessor*) la memoria se encuentra centralizada en el sistema a igual distancia (latencia) de todos los procesadores mientras que, en un CC-NUMA (*Cache-Coherence Non Uniform Memory Access*), cada procesador tiene físicamente cerca un conjunto de sus direcciones de memoria porque los módulos de memoria están físicamente distribuidos entre los procesadores y, por tanto, los módulos no están a igual distancia (latencia) de todos los procesadores.

La memoria centralizada del SMP hace que el tiempo de acceso a una dirección de memoria en un SMP sea igual para todos los procesadores y el tiempo de acceso a memoria de un procesador sea el mismo independientemente de la dirección a la que se esté accediendo; por estos motivos se denomina multiprocesador simétrico (*Symmetric Multiprocessor*) o UMA (*Uniform Memory Access*).

Los módulos de memoria físicamente distribuidos entre los procesadores de un CC-NUMA hacen que el tiempo de acceso a memoria dependa de si el procesador accede a una dirección de memoria que está en la memoria física que se encuentra en el nodo de dicho procesador (cercana, por tanto, al procesador que accede) o en la memoria física de otro nodo. Por este motivo el acceso no es uniforme o simétrico y recibe el nombre de NUMA.



**Cuestión 2.** ¿Cuándo diría que un computador es un multiprocesador y cuándo que es un multicomputador?

**Solución.** Será un multiprocesador si todos los procesadores comparten el mismo espacio de direcciones físico y será un multicomputador si cada procesador tiene su espacio de direcciones físico propio.



**Cuestión 3.** ¿Un CC-NUMA escala más que un SMP? ¿Por qué?

**Solución.** Sí, un CC-NUMA escala más que un SMP, porque al añadir procesadores al cálculo el incremento en el tiempo de acceso a memoria medio aumenta menos que en un SMP si el sistema operativo, el compilador y/o el programador se esfuerzan en ubicar en la memoria cercana a un procesador la carga de trabajo que va a procesar. La menor latencia media se debe a un menor número de conflictos en la red en el acceso a datos y a la menor distancia con el módulo de memoria físico al que se accede. Al aumentar menos la latencia se puede conseguir un tiempo de respuesta mejor en un CC-NUMA que en un SMP al ejecutar un código paralelo o múltiples códigos secuenciales a la vez.



**Cuestión 4.** Indique qué niveles de paralelismo implícito en una aplicación puede aprovechar un PC con un procesador de 4 cores, teniendo en cuenta que cada core tiene unidades funcionales SIMD (también llamadas unidades multimedia) y una microarquitectura segmentada y superscalar. Razone su respuesta.



**Solución.**

Paralelismo a nivel de operación. Al ser una arquitectura con paralelismo a nivel de instrucción por tener una arquitectura segmentada y superescalar, puede ejecutar operaciones en paralelo, luego puede aprovechar el paralelismo a nivel de operación.

Paralelismo a nivel de bucle (paralelismo de datos). Las operaciones realizadas en las iteraciones del bucle sobre vectores y matrices se podrían implementar en las unidades SIMD de los cores, lo que reduciría el número de iteraciones de los bucles. Además las iteraciones de los bucles, si son independientes, se podrían repartir entre los 4 cores.

Paralelismo a nivel de función. Las funciones independientes se podrían asignar a cores distintos para que se puedan ejecutar a la vez.

Paralelismo a nivel de programa. Los programas del mismo o distinto usuario se pueden asignar a distintos cores para así ejecutarlos al mismo tiempo.



**Cuestión 5.** Si le dicen que un ordenador es de 20 GIPS ¿puede estar seguro que ejecutará cualquier programa de 20000 instrucciones en un microsegundo?

**Solución.** Los MIPS se definen como el número de instrucciones que puede ejecutar un procesador (en millones) divididos por el tiempo que tardan en ejecutarse.

$$MIPS = \frac{\text{Numero\_de\_Instrucciones}}{\text{tiempo}(sg.) * 10^6}$$

Los MIPS suelen utilizarse como medida de las prestaciones de un procesador, pero realmente sólo permiten estimar la velocidad pico del procesador, que solo permitiría comparar procesadores con el mismo repertorio de instrucciones en cuanto a sus velocidades pico. Esto se debe a que esta medida

- No tiene en cuenta las características del repertorio de instrucciones de procesador. Se da el mismo valor a una instrucción que realice una operación compleja que a una instrucción sencilla.
- Pueden tenerse valores de MIPS inversamente proporcionales a la velocidad del procesador. Si un procesador tiene un repertorio de instrucciones de tipo CISC que permite codificar un algoritmo con, por ejemplo, 1000 instrucciones, y otro con un repertorio de tipo RISC lo codifica con 2000, si el primero tarda 1 microsegundo y el segundo 1.5 microsegundos, tendremos que el primer procesador tiene 1000 MIPS y el segundo 1333 MIPS. Por tanto, si nos fijamos en los MIPS, el segundo procesador será mejor que el primero, aun precisando un 50% más de tiempo que el primero, para resolver el mismo problema.

De ahí que incluso se haya dicho que MIPS significa *Meaningless Information of Processor Speed* (información sin sentido de la velocidad del procesador).

En relación a la pregunta que se plantea, la respuesta puede tener en cuenta dos aspectos:

- El número de instrucciones que constituyen un programa (número estático de instrucciones) puede ser distinto del número de instrucciones que ejecuta el procesador finalmente (número dinámico de instrucciones), ya que puede haber instrucciones de salto, bucles, etc. que hacen que ciertas instrucciones del código se ejecuten más de una vez, y otras no se ejecuten nunca. Si la pregunta, al hacer referencia al número de instrucciones, se refiere al número estático, la respuesta es que NO se puede estar seguro puesto que puede que al final no se ejecuten 20000 millones de instrucciones.
- Si el repertorio de instrucciones contiene instrucciones que tardan en ejecutarse tiempos diferentes, para que el programa tardase el mismo tiempo es preciso que se ejecutase el mismo número de instrucciones y del mismo tipo (en cuanto a tiempo de ejecución de cada una). En este caso,



también, la respuesta es NO.



**Cuestión 6.** ¿Aceptaría financiar/embarcarse en un proyecto en el que se plantease el diseño e implementación de un computador de propósito general con arquitectura MISD? (Justifique su respuesta).

**Solución:** El tipo de procesamiento que realiza un procesador MISD puede implementarse en un procesador MIMD con la sincronización correspondiente entre los procesadores del computador MIMD para que los datos vayan pasando adecuadamente de un procesador a otro (definiendo un flujo único de datos que pasan de procesador a procesador). Por esta razón un computador MISD **no tiene mucho sentido** como computador de propósito general. Sin embargo, puede ser útil un diseño MISD para un dispositivo de propósito específico que sólo tiene que ejecutar una aplicación que implica un procesamiento en el que los datos tienen que pasar por distintas etapas en las que sufren ciertas transformaciones que pueden programarse (en cada uno de los procesadores). La especificidad del diseño puede permitir generar diseños muy eficientes desde el punto de vista del consumo, de la complejidad hardware, etc.



**Cuestión 7.** Deduzca la ley de Amdahl para la mejora de la velocidad de un procesador suponiendo que hay una probabilidad  $f$  de no utilizar un recurso del procesador cuya velocidad se incrementa en un factor  $p$ .

**Solución:** El tiempo necesario para ejecutar un programa en el procesador sin la mejora es igual a:

$$T_{\text{sin\_mejora}} = f \cdot T_{\text{sin\_mejora}} + (1-f) \cdot T_{\text{sin\_mejora}}$$

donde  $f \cdot T_{\text{sin\_mejora}}$  es el tiempo que no podría reducirse al aplicar la mejora, y  $(1-f) \cdot T_{\text{sin\_mejora}}$  es el tiempo que podría reducirse como máximo en un factor igual a  $p$  al aplicar la mejora. Por tanto, el tiempo al aplicar la mejora sería:

$$T_{\text{con\_mejora}} \geq f \cdot T_{\text{sin\_mejora}} + ((1-f) \cdot T_{\text{sin\_mejora}}) / p$$

Según esto, la ganancia de velocidad que se podría conseguir sería:

$$S = T_{\text{sin\_mejora}} / T_{\text{con\_mejora}} \leq T_{\text{sin\_mejora}} / (f \cdot T_{\text{sin\_mejora}} + ((1-f) \cdot T_{\text{sin\_mejora}}) / p) = 1 / (f + (1-f)/p) = p / (1 + f(p-1))$$

y así llegamos a la expresión que se utiliza para describir la Ley de Amdahl:  $S \leq p / (1 + f(p-1))$



**Cuestión 8.** ¿Es cierto que para una determinada mejora realizada en un recurso se observa experimentalmente que, al aumentar el factor de mejora, llega un momento en que se satura el incremento de velocidad que se consigue? (Justifique la respuesta)

**Solución:** Para responder a esta pregunta recurrimos a la ley de Amdahl, que marca un límite superior a la mejora de velocidad. Como se tiene que  $S \leq p / (1 + f(p-1))$ , donde  $p$  es el factor de mejora y  $f$  la fracción de tiempo sin la mejora en el que dicha mejora no puede aplicarse, se puede calcular la derivada de  $p / (1 + f(p-1))$  con respecto a  $p$ , y ver qué pasa cuando cambia  $p$ . Así:

$$d(p / (1 + f(p-1))) / dp = ((1 + f(p-1)) - pf) / (1 + f(p-1))^2 = (1-f) / (1 + f(p-1))^2$$

y como puede verse, a medida que aumenta  $p$  el denominador se va haciendo mayor, y como  $(1-f)$  se mantiene fijo, la derivada tiende a cero. Eso significa que la pendiente de la curva que describe la variación de  $p / (1 + f(p-1))$  con  $p$  va haciéndose igual a cero y por lo tanto no aumenta la ganancia de velocidad: se



satura, o lo que es lo mismo, llega a una asíntota que estará situada en  $1/f$  (que es al valor al que tiende  $p/(1+f(p-1))$  cuando  $p$  tiende a infinito).



**Cuestión 9.** ¿Es cierto que la cota para el incremento de velocidad que establece la ley de Amdahl crece a medida que aumenta el valor del factor de mejora aplicado al recurso? (Justifique la respuesta).

**Solución:** La respuesta a esta pregunta se deduce de la misma expresión de la derivada de la cota que establece la ley de Amdahl, calculada al responder la cuestión 4:

$$d(p/(1+f(p-1)))/dp = ((1+f(p-1))-pf)/(1+f(p-1))^2 = (1-f)/(1+f(p-1))^2$$

dado que  $(1-f)$  es positivo (solo en el peor de los casos, si no se pudiera aplicar el factor de ganancia a ninguna parte del programa, tendríamos que  $(1-f)=0$  y no se obtendría ninguna ganancia de velocidad), y puesto que  $(1+f(p-1))^2$  siempre es positivo, la derivada es positiva. Eso significa que  $1+f(p-1)$  crece al crecer  $p$ . Lo que ocurre es que, tal y como se ha visto en la cuestión 4, este crecimiento es cada vez menor (tiende a 0).



**Cuestión 10.** ¿Qué es mejor, un procesador superescalar capaz de emitir cuatro instrucciones por ciclo, o un procesador vectorial cuyo repertorio permite codificar 8 operaciones por instrucción y emite una instrucción por ciclo? (Justifique su respuesta).

**Solución:** Considérese una aplicación que se codifica con  $N$  instrucciones de un repertorio de instrucciones escalar (cada instrucción codifica una operación). La cota inferior para el tiempo que un procesador superescalar podría tardar en ejecutar ese programa sería

$$T_{\text{superescalar}} = (N * T_{\text{ciclo\_superesc}})/4$$

Un procesador vectorial ejecutaría un número de instrucciones menor puesto que su repertorio de instrucciones permitiría codificar hasta ocho operaciones iguales (sobre datos diferentes) en una instrucción. Si suponemos que la aplicación permite que todas las operaciones que hay que ejecutar se puedan “empaquetar” en instrucciones vectoriales, el procesador vectorial tendría que ejecutar  $N/8$  instrucciones. Por tanto, considerando también que el procesador vectorial consigue terminar instrucciones según su velocidad pico (como supusimos en el caso del superescalar), la cota inferior para el tiempo que tarda la ejecución del programa en el procesador vectorial sería:

$$T_{\text{vectorial}} = (N/8) * T_{\text{ciclo\_vectorial}}/1$$

Por tanto

$$T_{\text{superescalar}} / T_{\text{vectorial}} = 8 * T_{\text{ciclo\_superesc}} / 4 * T_{\text{ciclo\_vectorial}} = 2 * T_{\text{ciclo\_superesc}} / T_{\text{ciclo\_vectorial}}$$

Según esto, si los dos procesadores funcionan a la misma frecuencia, el procesador vectorial podría ser mejor (siempre y cuando las hipótesis que se están considerando de que están procesando instrucciones según su máximo rendimiento se puedan considerar suficientemente aproximada a la realidad).

No obstante, hay que tener en cuenta que usualmente, los procesadores superescalares han sido los que más rápidamente han incorporado las mejoras tecnológicas, y normalmente, los procesadores superescalares funcionan a frecuencias más altas que los procesadores vectoriales contemporáneos. En el ejemplo, si la frecuencia del superescalar fuera más del doble de la del vectorial, sería mejor utilizar un procesador superescalar.



Además, hay que tener en cuenta que para que los procesadores vectoriales sean eficientes (se puedan admitir las hipótesis en cuanto a codificación de las operaciones y funcionamiento según la velocidad pico) es necesario que el programa tenga un alto grado de paralelismo de datos (son procesadores más específicos que los superescalares). Por eso, para dar una respuesta más precisa habría que conocer más detalles de la carga de trabajo que se piensa ejecutar en los procesadores.



2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores. Algunos ejercicios resueltos

## Tema 2. Programación paralela

Profesores responsables: Mancia Anguita, Julio Ortega

Licencia Creative Commons 

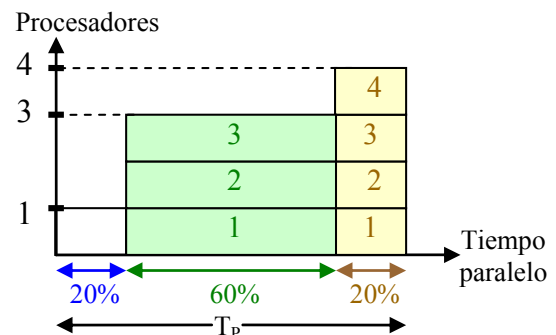
### 1 Ejercicios

**Ejercicio 1.** Un programa tarda 40 s en ejecutarse en un multiprocesador. Durante un 20% de ese tiempo se ha ejecutado en cuatro procesadores (core); durante un 60%, en tres; y durante el 20% restante, en un procesador (consideramos que se ha distribuido la carga de trabajo por igual entre los procesadores que colaboran en la ejecución en cada momento, despreciamos sobrecarga). ¿Cuánto tiempo tardaría en ejecutarse el programa en un único procesador? ¿Cuál es la ganancia en velocidad obtenida con respecto al tiempo de ejecución secuencial? ¿Y la eficiencia?

#### Solución

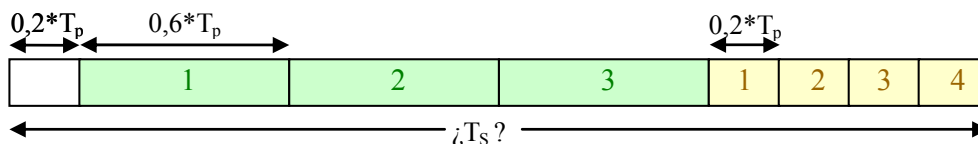
Datos del ejercicio:

En el gráfico de la derecha se representan los datos del ejercicio. Estos datos son la fracción del tiempo de ejecución paralelo ( $T_p$ ) que supone el código no paralelizable (20% de  $T_p$ , es decir,  $0,2 \cdot T_p$ ), la fracción que supone el código paralelizable en 3 procesadores ( $0,6 \cdot T_p$ ) y la que supone el código paralelizable en 4 procesadores ( $0,2 \cdot T_p$ ). Al distribuirse la carga de trabajo por igual entre los procesadores utilizados en cada instante, los trozos asignados a 3 procesadores suponen todos el mismo tiempo (por eso se han dibujado en el gráfico de igual tamaño) e, igualmente, los trozos asignados a 4 procesadores también suponen todos el mismo tiempo.



¿Tiempo de ejecución secuencial,  $T_s$ ?:

Los trozos de código representados en la gráfica anterior se deben ejecutar secuencialmente, es decir, uno detrás de otro, como se ilustra el gráfico siguiente:



$$T_s = 0,2 \times T_p + 3 \times 0,6 \times T_p + 4 \times 0,2 \times T_p = (0,2 + 1,8 + 0,8) \times T_p = 2,8 \times T_p = 2,8 \times 40 \text{ s} = 112 \text{ s}$$

¿Ganancia en velocidad,  $S(p)$ ?:

$$S(4) = \frac{T_s}{T_p(4)} = \frac{2,8 \times T_p(4)}{T_p(4)} = 2,8$$

¿Eficiencia,  $E(p)$ ?:

$$E(4) = \frac{S(p)}{p} = \frac{S(4)}{4} = \frac{2,8}{4} = 0,7$$

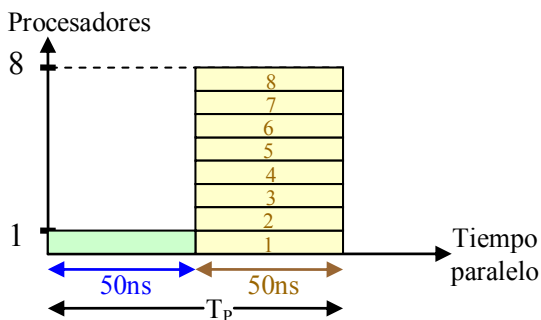


### Ejercicio 2. ◀

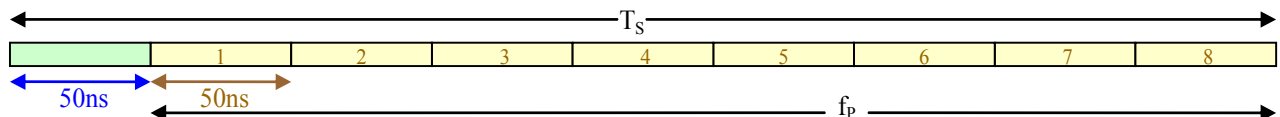
**Ejercicio 3.** ¿Cuál es fracción de código paralelo de un programa secuencial que, ejecutado en paralelo en 8 procesadores, tarda un tiempo de 100 ns, durante 50ns utiliza un único procesador y durante otros 50 ns utiliza 8 procesadores (distribuyendo la carga de trabajo por igual entre los procesadores)?

### Solución

Datos del programa:



¿Fracción de código paralelizable del programa secuencial,  $f_p$ ?:



$$f_p = \frac{8 \times 50 \text{ ns}}{T_s} = \frac{8 \times 50 \text{ ns}}{50 \text{ ns} + 8 \times 50 \text{ ns}} = \frac{8}{9}$$



### Ejercicio 4. ◀

### Ejercicio 5. ◀

### Ejercicio 6. ◀

**Ejercicio 7.** Se quiere paralelizar el siguiente trozo de código:

```
{Cálculos antes del bucle}
for( i=0; i<w; i++) {
    Código para i
}
{Cálculos después del bucle}
```

Los cálculos antes y después del bucle suponen un tiempo de  $t_1$  y  $t_2$ , respectivamente. Una iteración del ciclo supone un tiempo  $t_i$ . En la ejecución paralela, la inicialización de  $p$  procesos supone un tiempo  $k_1 p$  ( $k_1$

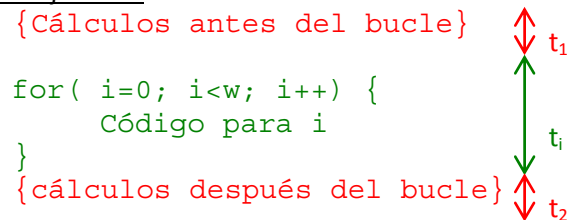


constante), los procesos se comunican y se sincronizan, lo que supone un tiempo  $k_2p$  ( $k_2$  constante);  $k_1p + k_2p$  constituyen la sobrecarga.

- Obtener una expresión para el tiempo de ejecución paralela del trozo de código en  $p$  procesadores ( $T_p$ ).
- Obtener una expresión para la ganancia en velocidad de la ejecución paralela con respecto a una ejecución secuencial ( $S_p$ ).
- ¿Tiene el tiempo  $T_p$  con respecto a  $p$  una característica lineal o puede presentar algún mínimo? ¿Por qué? En caso de presentar un mínimo, ¿para qué número de procesadores  $p$  se alcanza?

### Solución

Datos del ejercicio:



Sobrecarga:  $T_0 = k_1p + k_2p = (k_1 + k_2)p$

Llamaremos  $t$  a  $t_1 + t_2$  ( $= t$ ) y  $k$  a  $k_1 + k_2$  ( $= k$ )

(a) ¿Tiempo de ejecución paralela,  $T_p$ ?

$$T_p(p, w) = t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p$$

$w/p$  se redondea al entero superior

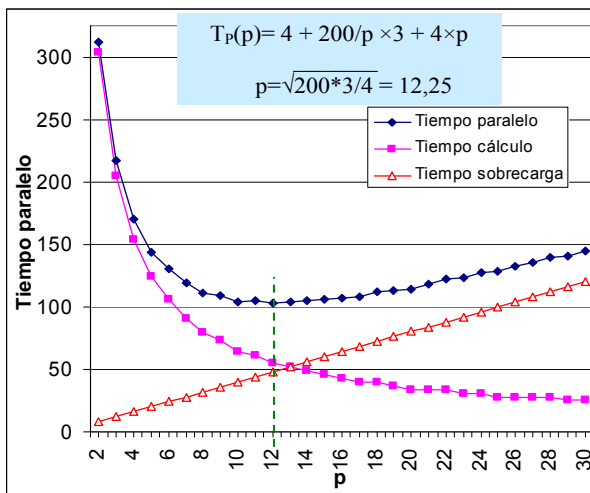
(b) ¿Ganancia en prestaciones,  $S(p, w)$ ?

$$S(p, w) = \frac{T_s}{T_p(p, w)} = \frac{t + w \times t_i}{t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p}$$

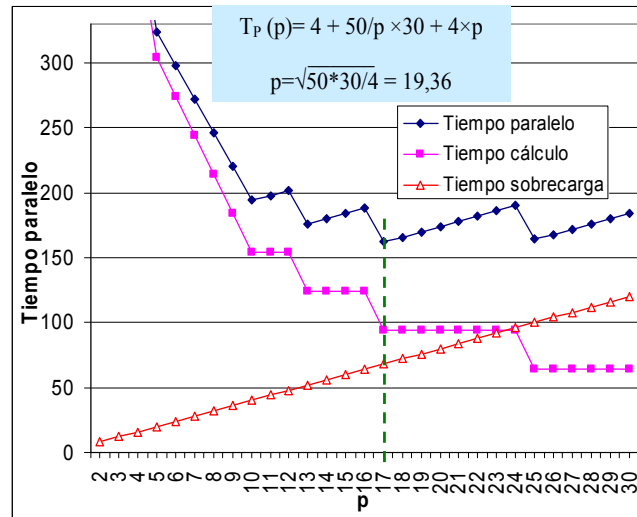
(c) ¿Presenta mínimo? ¿Para qué número  $p$ ?

$$T_p(p, w) = t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p \quad (1)$$

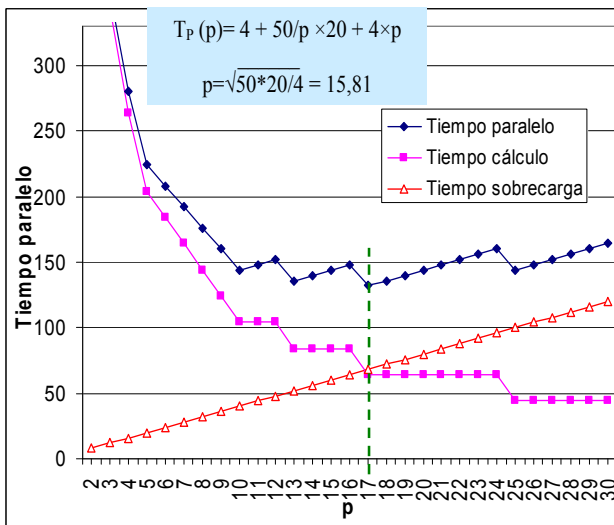
En la expresión (1), el término englobado en línea continua, o tiempo de cálculo paralelo, tiene tendencia a decrecer con pendiente que va disminuyendo conforme se incrementa  $p$  (debido a que  $p$  está en el denominador), y el término englobado con línea discontinua o tiempo de sobrecarga ( $k \times p$ ), crece conforme se incrementa  $p$  con pendiente  $k$  constante (ver ejemplos en Figura 3, Figura 4, Figura 5 y Figura 6). Las oscilaciones en el tiempo de cálculo paralelo se deben al redondeo al entero superior del cociente  $w/p$ , pero la tendencia, como se puede ver en las figuras, es que  $T_p$  va decreciendo. Dado que la pendiente de la sobrecarga es constante y que la pendiente del tiempo de cálculo decrece conforme se incrementa  $p$ , llega un momento en que el tiempo de ejecución en paralelo pasa de decrecer a crecer.



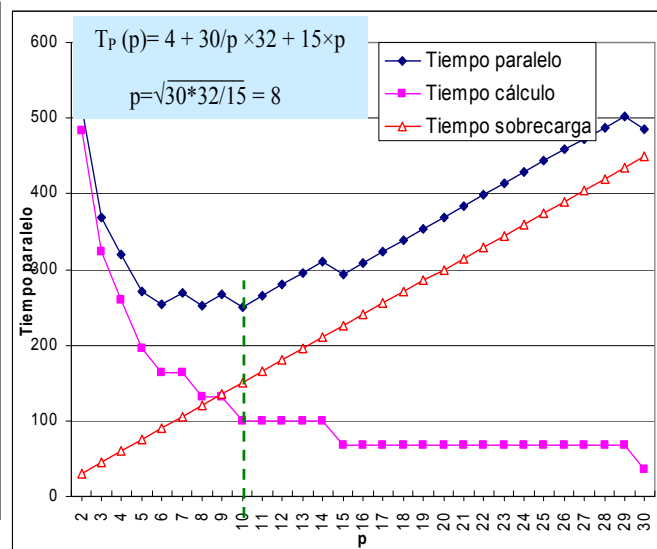
**Figura 3.** Tiempo de ejecución paralelo para un caso particular en el que  $w=200$ ,  $t_i$  es 3 unidades de tiempo,  $t$  son 4 unidades de tiempo y  $k$  son 4 unidades de tiempo. El mínimo se alcanza para  $p=12$



**Figura 4.** Tiempo de ejecución paralelo para un caso particular en el que  $w=50$ ,  $t_i$  es 30 unidades de tiempo,  $t$  son 4 unidades de tiempo y  $k$  son 4 unidades de tiempo. El mínimo se alcanza para  $p=17$ . Aquí el efecto del redondeo se ve más claramente que en la Figura 3



**Figura 5.** Tiempo de ejecución paralelo para un caso particular en el que  $w=50$ ,  $t_i$  es 20 unidades de tiempo,  $t$  son 4 unidades de tiempo y  $k$  son 4 unidades de tiempo. El mínimo se alcanza para  $p=17$ .



**Figura 6.** Tiempo de ejecución paralelo para un caso particular en el que  $w=30$ ,  $t_i$  es 32 unidades de tiempo,  $t$  son 4 unidades de tiempo y  $k$  son 15 unidades de tiempo. El mínimo se alcanza para  $p=10$ .

Se puede encontrar el mínimo analíticamente igualando a 0 la primera derivada de  $T_p$ . De esta forma se obtiene los máximos y mínimos de una función continua. Para comprobar si en un punto encontrado de esta forma hay un máximo o un mínimo se calcula el valor de la segunda derivada en ese punto. Si, como resultado de este cálculo, se obtiene un valor por encima de 0 hay un mínimo, y si, por el contrario, se obtiene un valor por debajo de 0 hay un máximo. Para realizar el cálculo se debe eliminar el redondeo de la expresión (1) (más abajo se comentará la influencia del redondeo en el cálculo del mínimo):

$$T_p(p, w) = t + \frac{w}{p} \times t_i + k \times p$$

$$T'_p(p, w) = 0 - \frac{w}{p^2} \times t_i + k \Rightarrow \frac{w}{p^2} \times t_i = k \Rightarrow p = \sqrt{\frac{w \times t_i}{k}} \quad (2)$$

El resultado negativo de la raíz se descarta. En cuanto al resultado positivo, debido al redondeo, habrá que comprobar para cuál de los naturales próximos al resultado obtenido (incluido el propio resultado si es un número natural) se obtiene un menor tiempo. Debido al redondeo hacia arriba de la expresión (1), se deberían comprobar necesariamente:

1. El natural  $p'$  menor o igual y más alejado al resultado  $p$  generado con (2) para el que  $\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil$  (obsérvese, que en el ejemplo de la Figura 4, aunque de (2) se obtiene 19,36, el  $p$  con menor tiempo es 17 debido al efecto del redondeo) y
2. El natural  $p'$  mayor y más próximo al  $p$  generado con (2) para el que  $\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil - 1$  (obsérvese, que en el ejemplo de la Figura 5, aunque de (2) se obtiene 15,81, el  $p$  con menor tiempo es 17 debido al redondeo).

En cualquier caso, el número de procesadores que obtengamos debe ser menor que  $w$  (que es el grado de paralelismo del código).

La segunda derivada permitirá demostrar que se trata de un mínimo:

$$T_p(p, w) = t + \frac{w}{p} \times t_i + k \times p$$

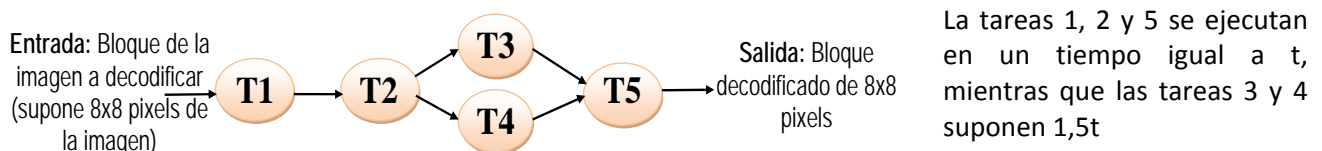
$$T''_p(p, w) = + \frac{2 \times p \times w \times t_i}{p^4} + 0 = \frac{2 \times w \times t_i}{p^3} > 0$$

La segunda derivada es mayor que 0, ya que  $w$ ,  $p$  y  $t_i$  son mayores que 0; por tanto, hay un mínimo.



### Ejercicio 8. ◀

**Ejercicio 9.** Se va a paralelizar un decodificador JPEG en un multiprocesador. Se ha extraído para la aplicación el siguiente grafo de tareas que presenta una estructura segmentada (o de flujo de datos):



El decodificador JPEG aplica el grafo de tareas de la figura a bloques de la imagen, cada uno de 8x8 píxeles. Si se procesa una imagen que se puede dividir en  $n$  bloques de 8x8 píxeles, a cada uno de esos  $n$  bloques se aplica el grafo de tareas de la figura. Obtenga la mayor ganancia en prestaciones que se puede conseguir paralelizando el decodificador JPEG en (suponga despreciable el tiempo de comunicación/sincronización): **(a)** 5 procesadores, y **(b)** 4 procesadores. En cualquier de los dos casos, la ganancia se tiene que calcular suponiendo que se procesa una imagen con un total de  $n$  bloques de 8x8 píxeles.

### Solución

Para obtener la ganancia se tiene que calcular el tiempo de ejecución en secuencial para un tamaño del problema de  $n$  bloques  $T_s(n)$  y el tiempo de ejecución en paralelo para un tamaño del problema de  $n$  y los  $p$  procesadores indicados en (a), (b) y (c)  $T_p(p, n)$ .

**Tiempo de ejecución secuencial**  $T_s(n)$ . Un procesador tiene que ejecutar las 5 tareas para cada bloque de  $8 \times 8$  píxeles. El tiempo que dedica el procesador a cada bloque es de  $3t$  (tareas 1, 2 y 5) +  $3t$  (tareas 2 y 4) =  $6t$ , luego para  $n$  bloques el tiempo de ejecución secuencial será el siguiente:

$$T_s = n \times (6t)$$

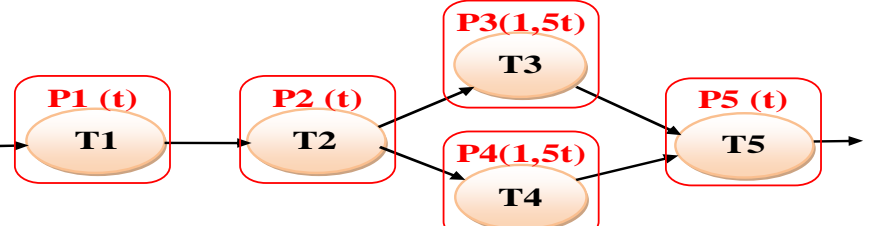
**(a) Tiempo de ejecución paralelo y ganancia en prestaciones para 5 procesadores**  $T_p(5, n)$ ,  $S(5, n)$ . Cada tarea se asigna a un procesador distinto, por tanto todas se pueden ejecutar en paralelo (ver asignación de tareas a procesadores en la Tabla 1). El tiempo de ejecución en paralelo en un pipeline consta de dos tiempos: el tiempo que tarda en procesarse la primera entrada (ver celdas con fondo verde en la tabla) + el tiempo que tardan cada uno del resto de bloques (entradas al cauce) en terminar. Este último depende de la etapa más lenta, que en este caso es la etapa 3 ( $1,5t$  frente a  $t$  en el resto de etapas).

$$T_p(5, n) = 4,5t + (n - 1) \times 1,5t = 3t + n \times 1,5t$$

$$S(5, n) = \frac{T_s(n)}{T_p(5, n)} = \frac{n \times 6t}{3t + n \times 1,5t} \xrightarrow{n \rightarrow \infty} 4$$

La ganancia se aproxima a 4 para  $n$  suficientemente grande.

**Tabla 1.** Asignación de tareas a procesadores, ocupación de las etapas del cauce para los primeros bloques procesados y tiempo de procesamiento del primer bloque y de los siguientes para 5 procesadores

					
<b>Etapa 1 (t)</b>	<b>Etapa 2 (t)</b>	<b>Etapa 3 (1,5t)</b>	<b>Etapa 4 (t)</b>	<b>Terminado</b>	<b>Tiempo procesa.</b>
T1 (procesador P1)	T2 (procesador P2)	T3 (procesador P3) y T4 (procesador P4)	T5 (procesador P5)		
0 - Bloque 1 - t					Procesamiento Bloque 1 (4,5t)
t - Bloque 2 - 2t	t - Bloque 1 - 2t				
2t - Bloque 3 - 3t	2t - Bloque 2 - 3t	2t - Bloque 1 - 3,5t			
3t - Bloque 4 - 4t	3t - Bloque 3 - 4t	3,5t - Bloque 2 - 5t	3,5t - Bloque 1 - 4,5t		
4t - Bloque 5 - 5t	4t - Bloque 4 - 5t	5t - Bloque 3 - 6,5t	5t - Bloque 2 - 6t	Bloque 1	$t + t + 1,5t + t = 4,5t$
5t - Bloque 6 - 6t	5t - Bloque 5 - 6t	6,5t - Bloque 4 - 8t	6,5t - Bloque 3 - 7,5t	Bloque 2	$4,5t + 1,5t = 6t$
6t - Bloque 7 - 7t	6t - Bloque 6 - 7t	8t - Bloque 5 - 9,5t	8t - Bloque 4 - 9t	Bloque 3	$6t + 1,5t = 7,5t$
...	...	...	...	Bloque 4	$7,5t + 1,5t = 9t$
$T_{\text{entrada\_etapa}} - \text{Bloque } x - T_{\text{salida\_etapa}}$					

**(b) Tiempo de ejecución paralelo y ganancia en prestaciones para 4 procesadores**  $T_p(4, n)$ ,  $S(4, n)$ . Se deben asignar las 5 tareas a los 4 procesadores de forma que se consiga el mejor tiempo de ejecución paralelo, es decir, el menor tiempo por bloque una vez procesado el primer bloque. Una opción que nos lleva al menor tiempo por bloque consiste en unir la Etapa 1 y 2 del pipeline del caso (a) en una única etapa asignando T1 y T2 a un procesador (ver asignación de tareas a procesadores en la Tabla 2). Con esta asignación la etapa más lenta supone  $2t$  (Etapa 1); habría además una etapa de  $1,5t$  (Etapa 2) y otra de  $t$  (Etapa 3). Si, por ejemplo, se hubieran agrupado T3 y T4 un procesador la etapa más lenta supondría  $3t$ .

$$T_p(4, n) = 4,5t + (n - 1) \times 2t = 2,5t + n \times 2t$$

$S(4, n) = \frac{T_s(n)}{T_p(4, n)} = \frac{n \times 6t}{2,5t + n \times 2t} \xrightarrow{n \rightarrow \infty} 3$  La ganancia se aproxima a 3 para  $n$  suficientemente grande.

**Tabla 2.** Asignación de tareas a procesadores, ocupación de las etapas del cauce para los primeros bloques procesados y tiempo de procesamiento del primer bloque y de los siguientes para 4 procesadores

<p>Possible asignación de tareas a 4 procesadores</p>				
Etapas 1 (2t)	Etapas 2 (1,5t)	Etapas 3 (t)	Terminado	Tiempo procesamiento
T1 y T2 (procesador P1)	T3 (procesador P2) y T4 (procesador P3)	T5 (procesador P4)		
0 - Bloque 1 - 2t				Procesamiento Bloque 1 (4,5t)
2t - Bloque 2 - 4t	2t - Bloque 1 - 3,5t			
4t - Bloque 3 - 6t	4t - Bloque 2 - 5,5t	3,5t - Bloque 1 - 4,5t		
6t - Bloque 4 - 8t	6t - Bloque 3 - 7,5t	5,5t - Bloque 2 - 6,5t	Bloque 1	2t + 1,5t + t = 4,5t
8t - Bloque 5 - 10t	8t - Bloque 4 - 9,5t	7,5t - Bloque 3 - 8,5t	Bloque 2	4,5t + 2t = 6,5t
10t - Bloque 6 - 12t	10t - Bloque 5 - 11,5t	9,5t - Bloque 4 - 10,5t	Bloque 3	6,5t + 2t = 8,5t
12t - Bloque 7 - 14t	12t - Bloque 6 - 13,5t	11,5t - Bloque 5 - 12,5t	Bloque 4	8,5t + 2t = 10,5t
...	...	...	Bloque 5	10,5t + 2t = 12,5t
T_entrada_etapa - Bloque x - T_salida_etapa				



**Ejercicio 10.** Se quiere implementar un programa paralelo para un multicomputador que calcule la siguiente expresión para cualquier  $x$  (es el polinomio de interpolación de Lagrange):  $P(x) = \sum_{i=0}^n (b_i \cdot L_i(x))$ , donde

$$L_i(x) = \frac{(x-a_0) \cdot \dots \cdot (x-a_{i-1}) \cdot (x-a_{i+1}) \cdot \dots \cdot (x-a_n)}{k_i} = \frac{\prod_{\substack{j=0 \\ j \neq i}}^n (x-a_j)}{k_i} \quad i = 0, 1, \dots, n$$

$$k_i = (a_i - a_0) \cdot \dots \cdot (a_i - a_{i-1}) \cdot (a_i - a_{i+1}) \cdot \dots \cdot (a_i - a_n) = \prod_{\substack{j=0 \\ j \neq i}}^n (a_i - a_j) \quad i = 0, 1, \dots, n$$

Inicialmente  $k_i$ ,  $a_i$  y  $b_i$  se encuentra en el nodo  $i$  y  $x$  en todos los nodos. Sólo se van a usar funciones de comunicación colectivas. Indique cuál es el número mínimo de funciones colectivas que se pueden usar, cuáles serían y en qué orden se utilizarían y para qué se usan en cada caso.

### Solución

Los pasos ((1) a (5)) del algoritmo para  $n=3$  y un número de procesadores de  $n+1$  serían los siguientes:

Pr.	Situación Inicial				(1) Resta paralela $A_i = (x - a_i)$ ( $(x - a_i)$ se obtiene en $P_i$ )	(2) Todos reducen $A_i$ con resultado en $B_i$ : $B_i = \sum_{i=0}^n (A_i)$
P0	$a_0$	$x$	$k_0$	$b_0$	$(x - a_0)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P1	$a_1$	$x$	$k_1$	$b_1$	$(x - a_1)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P2	$a_2$	$x$	$k_2$	$b_2$	$(x - a_2)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P3	$a_3$	$x$	$k_3$	$b_3$	$(x - a_3)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$



Pr.	(3) Cálculo de todos los $L_i(x)$ en paralelo $L_i = B_i / (A_i \cdot k_i)$ ( $L_i(x)$ se obtiene en $P_i$ )	(4) Cálculo en paralelo $C_i = b_i \cdot L_i$ ( $b_i \cdot L_i(x)$ se obtiene en $P_i$ )	(5) Reducción del contenido de $C_i$ con resultado en $P_0$ ( $P(x)$ se obtiene en $P_0$ )
P0	$(x-a_1) \cdot (x-a_2) \cdot (x-a_3) / k_0$	$b_0 \cdot (x-a_1) \cdot (x-a_2) \cdot (x-a_3) / k_0$	$P = \sum_{i=0}^n (C_i) = \sum_{i=0}^n (b_i \times L_i)$
P1	$(x-a_0) \cdot (x-a_2) \cdot (x-a_3) / k_1$	$b_1 \cdot (x-a_0) \cdot (x-a_2) \cdot (x-a_3) / k_1$	
P2	$(x-a_0) \cdot (x-a_1) \cdot (x-a_3) / k_2$	$b_2 \cdot (x-a_0) \cdot (x-a_1) \cdot (x-a_3) / k_2$	
P3	$(x-a_0) \cdot (x-a_1) \cdot (x-a_2) / k_3$	$b_3 \cdot (x-a_0) \cdot (x-a_1) \cdot (x-a_2) / k_3$	

Como se puede ver en el trazado del algoritmo para  $n=3$  mostrado en las tablas, se usan un total de 2 funciones de comunicación colectivas (pasos (2) y (5) en la tabla). En el paso (2) del algoritmo se usa una operación de “todos reducen” para obtener en todos los procesadores los productos de todas las restas  $(x-a_i)$ . En el paso (5) y último se realiza una operación de reducción para obtener las sumas de todos los productos  $(b_i \times L_i)$  en el proceso 0.



**Ejercicio 11. (a)** Escriba un programa secuencial con notación algorítmica (podría escribirlo en C) que determine si un número de entrada,  $x$ , es primo o no. El programa imprimirá si es o no primo. Tendrá almacenados en un vector,  $NP$ , los  $M$  números primos entre 1 y el máximo valor que puede tener tener un número de entrada al programa.

**(b)** Escriba una versión paralela del programa anterior para un multicomputador usando un estilo de programación paralela de paso de mensajes. El proceso 0 tiene inicialmente el número  $x$  y el vector  $NP$  en su memoria e imprimirá en pantalla el resultado. Considere que la herramienta de programación ofrece funciones `send()/receive()` para implementar una comunicación uno-a-uno asíncrona, es decir, con función `send(buffer, count, datatype, idproc, group)` no bloqueante y `receive(buffer, count, datatype, idproc, group)` bloqueante. En las funciones `send()/receive()` se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación.
- `idproc`: identificador del proceso al que se envía o del que se recibe.
- `buffer`: dirección a partir de la cual se almacenan los datos que se envían o los datos que se reciben.
- `datatype`: tipo de los datos a enviar o recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- `count`: número de datos a transferir de tipo `datatype`.

### Solución

#### (a) Programa secuencial que determina si $x$ es o no primo

##### Versión 1 Pre-condición

$x$ : número de entrada  $\{2, \dots, MAX\_INPUT\}$ .  $MAX\_INPUT$ : máximo valor de la entrada

$M$ : número de primos entre 2 y  $MAX\_INPUT$  (ambos incluidos).

$NP$ : vector con  $M+1$  componentes (los  $M$  nº primos desde 2 hasta  $MAX\_INPUT$  en  $NP[0]$  hasta  $NP[M-1]$ , a  $NP[M]$  se asignará en el código  $x+1$ )

##### Versión 1 Pos-condición

Imprime en pantalla si el número  $x$  es primo o no



Código**Versión 1 A**

```

if (x>NP[M-1]) {
    print("%u supera el máximo primo a
detectar %u \n", x, NP[M-1]);
    exit(1);
}
NP[M]=x+1;
i=0;
while (x<NP[i]) do i++;

if (x==NP[i])
    printf("%u ES primo\n", x);
else printf("%u NO ES primo\n", x);

```

**Versión 1 B**

```

if (x>NP[M-1]) {
    print("%u supera el máximo primo a
detectar %u \n", x, NP[M-1]);
    exit(1);
}
NP[M]=x+1;

for (i=0; x<NP[i];i++) {}

if (x==NP[i])
    printf("%u ES primo\n", x);
else printf("%u NO es primo\n", x);

```

**Versión 1:** El número de instrucciones de comparación depende de en qué posición se encuentre el número primo  $x$  en  $NP$ . En el peor caso (cuando  $x$  no es primo) el bucle recorre todo el vector realizándose  $M+1$  comparaciones en el bucle y 1 comparaciones en cada `if/else` (segundo `if` en el código). Resumiendo:  $M+1$  comparaciones en el bucle (Orden de complejidad de  $M$ ).

Versión 2 Pre-condición

$x$ : número de entrada  $\{2, \dots, \text{MAX\_INPUT}\}$ .  $\text{MAX\_INPUT}$ : máximo valor de la entrada

$M$ : número de primos entre 2 y  $\text{MAX\_INPUT}$  (ambos incluidos)

$NP$ : vector con los  $M$  nº primos desde 2 hasta  $\text{MAX\_INPUT}$

$xr$ : raíz cuadrada de  $x$

Versión 2 Pos-condición

Imprime en pantalla si el número  $x$  es primo o no

Código**Versión 2**

```

if (x>NP[M-1]) {
    print("%u supera el máximo primo a detectar %u \n", x, NP[M-1]);
    exit(1);
}
xr = sqrt(x);
for ( i=0 ; (NP[i]<=xr) && (x % NP[i]!=0); i++) {} ; // % = módulo

if (NP[i]<=xr) printf("%u ES primo", x);
else printf("%u NO ES primo", x);

```

**Versión 2:** El número de instrucciones de comparación depende de en qué posición,  $nr$ , de  $NP$  se encuentre el primer número primo mayor que  $\text{sqrt}(x)$ . En el peor caso (cuando  $x$  no es primo) se recorre hasta la posición  $nr$ , realizándose  $nr+1$  comparaciones y  $nr$  módulos/comparaciones en el bucle (se supone que el bucle se implementa con dos saltos condicionales y que el primero evalúa  $NP[i] \leq xr$  y el segundo si el resultado del módulo ha sido o no cero) y 1 comparación en cada `if/else`. Resumiendo: Orden de complejidad de  $nr$ .

**(b)** Programa paralelo para multicomputador que determina si  $x$  es o no primo



Se van a repartir las iteraciones del bucle entre los procesadores del grupo. Todos los procesos ejecutan el mismo código

#### Pre-condición

x: número de entrada {2,...,MAX\_INPUT}; MAX\_INPUT: máximo valor de la entrada

xr: almacena la raíz cuadrada de x (sólo se usa en la versión 2)

M: número de primos entre 2 y MAX\_INPUT (ambos incluidos)

grupo: identificador del grupo de procesos que intervienen en la comunicación.

num\_procesos: número de procesos en grupo.

idproc: identificador del proceso (dentro del grupo de num\_procesos procesos) que ejecuta el código

tipo: tipo de los datos a enviar o recibir

NP: vector con los M nº primos entre 2 y MAX\_INPUT (en la versión 1 tendrá M+1 componentes)

b, baux: variables que podrán tomar dos valores 0 (=false) o 1 (=true, si se localiza x en la lista de números primos)

#### Pos-condición

Imprime en pantalla si el número x es primo o no

#### Código

Versión 1	Versión 2
<pre> if (x&gt;NP[i]) {     print("%u supera el máximo primo a detectar %u \n", x, NP[M-1]);     exit(1); } NP[M]=x+1;  //Difusión de x y NP if (idproc==0)     for (i=1; i&lt;num_procesos) {         send(NP,M+1,tipo,i,grupo);         send(x,1,tipo,i,grupo);     } else {     receive(NP,M+1,tipo,0,grupo);     receive(x,1,tipo,0,grupo); }  //Cálculo paralelo, asignación estática i=id_proc; while (x&lt;NP[i]) do {     i=i+num_procesos; } b=(x==NP[i])?1:0;  //Comunicación resultados if (idproc==0)     for (i=1; i&lt;num_procesos) {         receive(baux,1,tipo,i,grupo);         b = b   baux;     } else send(b,1,tipo,0,grupo);  //Proceso 0 imprime resultado if (idproc==0)     if (b!=0)         printf("%d ES primo", x);     else printf("%d NO es primo", x); </pre>	<pre> if (x&gt;NP[i]) {     print("%u supera el máximo primo a detectar %u \n", x, NP[M-1]);     exit(1); }  //Difusión de x y NP if (idproc==0)     for (i=1; i&lt;num_procesos) {         send(NP,M,tipo,i,grupo);         send(x,1,tipo,i,grupo);     } else {     receive(NP,M,tipo,0,grupo);     receive(x,1,tipo,0,grupo); }  //Cálculo paralelo, asignación estática i=id_proc; xr = sqrt(x); while ((NP[i]&lt;=xr)&amp;&amp;(x % NP[i]!=0)) do {     i=i+num_procesos; } b=(NP[i]&lt;=xr)?1:0;  //Comunicación resultados if (idproc==0)     for (i=1; i&lt;num_procesos) {         receive(baux,1,tipo,i,grupo);         b = b   baux;     } else send(b,1,tipo,0,grupo);  // Proceso 0 imprime resultado if (idproc==0)     if (b!=0)         printf("%d ES primo", x);     else printf("%d NO es primo", x); </pre>





**Versión 1:** El número de instrucciones de comparación en la parte de cálculo paralelo depende de en qué posición se encuentre el número primo en NP. En el peor caso (cuando  $x$  no es primo) se recorre todo el vector entre todos los procesos, pero cada uno accede a un subconjunto de componentes del vector; en particular, a  $M/\text{num\_procesos}$  componentes si  $\text{num\_procesos}$  divide a  $M$ . Entonces, si  $\text{num\_procesos}$  divide a  $M$ , cada proceso realizará  $M/\text{num\_procesos}+1$  comparaciones y  $M/\text{num\_procesos}$  en el bucle y 1 comparación más para obtener  $b$ . Se realiza ese número de comparaciones en el bucle porque cada proceso actualiza el índice  $i$  del `while` en cada iteración sumándole el número de procesos  $\text{num\_procesos}$ . Por tanto, se asignan a los procesos las posiciones de NP con las que comparar en Round-Robin; por ejemplo, el proceso 0 ejecutará las iteraciones para  $i$  igual a 0,  $\text{num\_procesos}$ ,  $2*\text{num\_procesos}$ ,  $3*\text{num\_procesos}$ ,... En total, todos los procesos en conjunto harían  $(M/\text{num\_procesos}+1+1)*\text{num\_procesos}$  comparaciones en paralelo  $(M+2*\text{num\_procesos})$ . Todos los procesos realizarán el mismo número de comparaciones si  $\text{num\_procesos}$  divide a  $M$  (como se ha comentado más arriba), en caso contrario, debido a la asignación Round-Robin, algunos harán una iteración más del bucle que otros y, por tanto, una comparación más.

Resumiendo:

- Si  $\text{num\_procesos}$  divide a  $M$ : cada proceso realiza  $M/\text{num\_procesos}+1$  comparaciones en el bucle y 1 comparación más para obtener  $b$  (Orden de  $M/\text{num\_procesos}$ )
- Si  $\text{num\_procesos}$  NO divide a  $M$ : algunos procesos realizan una comparación más que el resto, estos procesos realizan  $\text{Truncado}(M/\text{num\_procesos})+2$  comparaciones en el bucle y 1 comparación más para obtener  $b$  (Orden de  $M/\text{num\_procesos}$ )

**Versión 2:** El número de instrucciones de comparación en la parte de cálculo paralelo depende de en qué posición,  $nr$ , de NP se encuentre el primer número primo mayor que  $\sqrt{x}$ . En el peor caso (cuando  $x$  no es primo) se recorre hasta  $nr$  entre todos los procesos, pero cada uno accede a un subconjunto de componentes del vector entre 0 y  $nr$ ; en particular, a  $nr/\text{num\_procesos}$  componentes si  $\text{num\_procesos}$  divide a  $nr$ . Entonces, si  $\text{num\_procesos}$  divide a  $nr$ , cada proceso realizará  $nr/\text{num\_procesos} + 1$  comparaciones y  $nr/\text{num\_procesos}$  módulos/comparaciones en el bucle y 1 comparación más para obtener  $b$ . Se realiza ese número de operaciones en el bucle porque cada proceso actualiza el índice  $i$  del `while` en cada iteración sumándole el número de procesos  $\text{num\_procesos}$ . Por tanto, se asignan a los procesos las posiciones de NP con las que comparar en Round-Robin; por ejemplo, el proceso 1 ejecutará las iteraciones para  $i$  igual a 1,  $\text{num\_procesos}+1$ ,  $2*\text{num\_procesos}+1$ ,  $3*\text{num\_procesos}+1$ ,.... En total, todos los procesos en conjunto harían  $2*nr+2*\text{num\_procesos}$  operaciones (operación comparación y operación módulo/comparación). Todos los procesos realizarán el mismo número de operaciones si  $\text{num\_procesos}$  divide a  $nr$ , en caso contrario, debido a la asignación Round-Robin, algunos harán una iteración más del bucle que otros y, por tanto, dos operaciones más (una operación comparación más una operación módulo/comparación).

Resumiendo:

- Si  $\text{num\_procesos}$  divide a  $nr$ : cada proceso realiza  $2*nr+2*\text{num\_procesos}$  operaciones (operación comparación y operación módulo/comparación). El orden de complejidad es de  $nr/\text{num\_procesos}$ .
- Si  $\text{num\_procesos}$  NO divide a  $nr$ : algunos procesos realizan dos comparaciones y un módulo más que el resto, estos procesos realizan  $2*\text{Truncado}(nr/\text{num\_procesos})+3$  operaciones en el bucle y 1 operación de comparación para obtener  $b$ . El orden de complejidad es de  $nr/\text{num\_procesos}$ .



## Ejercicio 12.





Ejercicio 13.



Ejercicio 14.



2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores. Algunos ejercicios resueltos

## Tema 3. Arquitecturas con paralelismo a nivel de thread (TLP)

Profesores responsables: Mancia Anguita, Julio Ortega

Licencia Creative Commons 

### 1 Ejercicios

**Ejercicio 1.** En un multiprocesador SMP con 4 procesadores o nodos (N0-N3) 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:

1. Lectura generada por el procesador 1
2. Lectura generada por el procesador 2
3. Escritura generada por el procesador 1
4. Escritura generada por el procesador 2
5. Escritura generada por el procesador 3

#### Solución

##### Datos del ejercicio

Se accede a una dirección de memoria cuyo bloque  $k$  no se encuentra en ninguna cache, luego debe estar actualizado en memoria principal y el estado en las caches se considera inválido.

##### Estado del bloque en las caches y acciones generadas ante los eventos que se refiere a dicho bloque

Hay 4 nodos con cache y procesador (N0-N3). Intervienen N1, N2 y N3. En la tabla se van a utilizar las siguientes siglas y acrónimos:

MP: Memoria Principal.

PtLec( $k$ ): paquete de petición de lectura del bloque  $k$ .

PtLecEx( $k$ ): paquete de petición de lectura del bloque  $k$  y de petición de acceso exclusivo al bloque  $k$ .

RpBloque( $k$ ): paquete de respuesta con el bloque  $k$ .

Se va a suponer que no existe en el sistema paquete de petición de acceso exclusivo a un bloque sin lectura (no existe PtEx).

ESTADO INICIAL	EVENTO	ACCIÓN	ESTADO SIGUIENTE
N1) Inválido N2) Inválido N3) Inválido	P1 lee $k$	1.- N1 (el controlador de cache de N1) genera y deposita en el bus una petición de lectura del bloque $k$ (PtLec( $k$ )) porque no lo tiene en su caché válido  2.-MP (el controlador de memoria de MP), al observar PtLec( $k$ ) en el bus, genera la respuesta con el bloque (RpBloque( $k$ )).	N1) Exclusivo N2) Inválido N3) Inválido



		3.- N1 (el controlador de cache de N1) recoge del bus la respuesta depositada por la memoria principal (RpBloque(k)), el bloque entra en la cache de N1 en estado exclusivo ya que no hay copia en otra cache del bloque (es decir, la salida de la OR cableada con entradas procedentes de todas las caches es 0).	
<b>N1) Exclusivo N2) Inválido N3) Inválido</b>	P2 lee k	<p>1.- N2 genera y deposita en el bus una PtLec(k) porque no tiene k en su caché en estado válido</p> <p>2.- N1 observa PtLec(k) en el bus y, como tiene el bloque en estado exclusivo, lo pasa a compartido (la copia que tiene ya no es la única válida en caches). MP, al observar PtLec(k) en el bus, genera la respuesta con el bloque (RpBloque(k)).</p> <p>3.- N2 recoge RpBloque(k) que ha depositado la memoria, el bloque entra en estado compartido en la cache de N2 (la salida de la OR cableada será 1).</p>	<b>N1) Compartido N2) Compartido N3) Inválido</b>
<b>N1) Compartido N2) Compartido N3) Inválido</b>	P1 escribe en k	<p>1.- N1 genera petición de lectura con acceso exclusivo del bloque k (PtLecEx(k)) (suponemos que no hay petición de acceso exclusivo sin lectura, no hay PtEx). N1 modifica la copia de k que tiene en su cache y lo pasa a estado modificado.</p> <p>2.- N2 observa PtLecEx(k) y, como la petición incluye acceso exclusivo (Ex) a un bloque que tiene en su cache en estado compartido, pasa su copia a estado inválido. MP genera RpBloque(k) porque observa en el bus una petición de k con lectura (Lec), pero esta respuesta no se va a recoger del bus. N1 no recoge RpBloque(k) depositada por la memoria porque tiene el bloque válido.</p>	<b>N1) Modificado N2) Inválido N3) Inválido</b>
<b>N1) Modificado N2) Inválido N3) Inválido</b>	P2 escribe en k	<p>1.- N2 genera petición de lectura con acceso exclusivo de k (PtLecEx(k))</p> <p>2.- N1 observa PtLecEx(k) y, como tiene el bloque en estado modificado (es la única copia válida en todo el sistema), inhibe la respuesta de MP y genera respuesta con el bloque RpBloque (k), y además, como el paquete pide acceso exclusivo a k (Ex), invalida su copia del bloque k.</p> <p>3.- N2 recoge RpBloque(k), introduce k en su cache, lo modifica y lo pone en estado modificado</p>	<b>N1) Inválido N2) Modificado N3) Inválido</b>
<b>N1) Inválido N2) Modificado N3) Inválido</b>	P3 escribe en k	<p>1.- N3 genera petición de lectura con acceso exclusivo de k PtLecEx(k)</p> <p>2.- N2 observa PtLecEx(k) y, como tiene el bloque en estado modificado, inhibe la respuesta de MP y genera respuesta con el bloque RpBloque (k), y además, como el paquete pide acceso exclusivo a k (Ex), invalida su copia de k.</p> <p>3.- N3 recoge RpBloque(k), introduce el k en su cache, lo modifica y lo pone en estado modificado</p>	<b>N1) Inválido N2) Inválido N3) Modificado</b>



## Ejercicio 2. .



**Ejercicio 3.**

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

<u>P1</u>	<u>P2</u>
x=1;	y=1;
x=2;	y=2;
print y ;	print x ;

Qué resultados se pueden imprimir si (considere que el compilador no altera el código):

- (a) Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.
- (b) Se ejecutan en un multiprocesador basado en un bus que garantiza todos los órdenes excepto el orden  $W \rightarrow R$ . 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.

**Solución**

El compilador no altera ningún orden garantizado ya que se supone, según el enunciado, que no altera el código.

(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 (el hardware parece ejecutar los accesos a memoria del código que ejecuta un procesador en el orden en el que están en dicho código) y, por tanto, cuando P1 lee “y” (instrucción 1.3 en el código, esta instrucción lee “y” para imprimir su contenido), ha asignado ya a “x” un 2 (punto 1.2 en el código) ya que esta asignación está antes en el código que la lectura de “y”.

<u>P1</u>	<u>P2</u>
(1.1) x=1;	(2.1) y=1;
(1.2) x=2;	(2.2) y=2;
(1.3) print y ;	(2.3) print x ;

Si P2 es el primero que imprime podrá imprimir 0, 1 o 2, pero entonces P1 sólo puede imprimir 2. Esto es así porque se mantiene orden secuencial y, por tanto, cuando P2 lee “x” (punto 2.3 en el código), ha asignado ya a “y” un 2 (punto 2.2 en el código) ya que esta asignación está antes en el código que la lectura de “x” y se mantiene orden secuencial en los accesos a memoria, es decir, los accesos parecen completarse en el orden en el que se encuentran en el código.

Se puede obtener como resultado de la ejecución las combinaciones que hay en cada una de las líneas:

P1 P2

0 2 (en este caso P1 imprime 0 y P2 imprime 2)

1 2

2 2

2 0

2 1

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

P1 P2

1 1 (en este caso P1 imprime 1 y P2 imprime 1)

0 1

0 2

1 0

2 0

0 0

Se pueden imprimir también las combinaciones anteriores porque no se asegura que cuando un procesador ejecute la lectura de la variable que imprime print (puntos 1.3 y 2.3 en los códigos) haya ejecutado las



instrucciones anteriores que escriben en  $x$  (P1 en los puntos 1.1 y 1.2 del código) o en  $y$  (P2 en los puntos 2.1 y 2.2). Esto es así porque no se garantiza el orden  $W \rightarrow R$  y, por tanto, una lectura puede adelantar a escrituras que estén antes en el código secuencial. P1 puede leer  $y$  (1.3) antes de escribir en  $x$  2 (1.2) o incluso antes de escribir en  $x$  1 (1.1). Igualmente P2 puede leer  $x$  (2.3) antes de escribir en  $y$  2 (2.2) o antes de escribir en  $y$  1 (2.1).

Teniendo esto en cuenta P1 puede imprimir 1 o 2 o 0, y P2 1 o 2 o 0. Todas las combinaciones son posibles.



#### Ejercicio 5.



#### Ejercicio 6. .



#### Ejercicio 7. .



#### Ejercicio 8. .



#### Ejercicio 9. .



**Ejercicio 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, que el único orden no garantizado por el sistema de memoria es  $W \rightarrow R$ , que las primitivas atómicas garantizan que sus accesos a memoria se realizan antes que los accesos posteriores y que el compilador no altera el código.

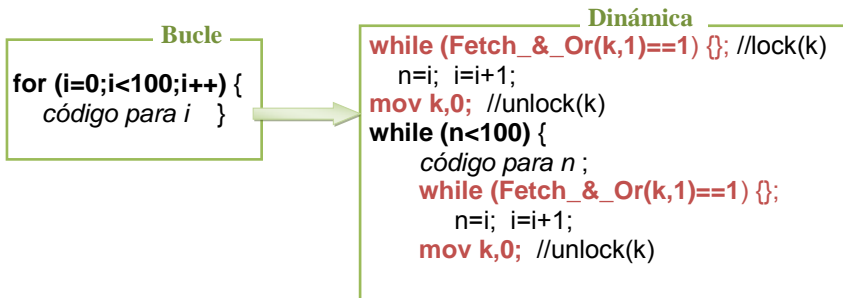
- (a) Paralelizar el ciclo para su ejecución en un multiprocesador que implementa la primitiva `Fetch&Or` para garantizar exclusión mutua.
- (b) Paralelizar el anterior ciclo en un multiprocesador que además tiene la primitiva `Fetch&Add`.

#### Solución

Se debe tener en cuenta que el único orden que no garantiza el hardware es el orden  $W \rightarrow R$ .

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

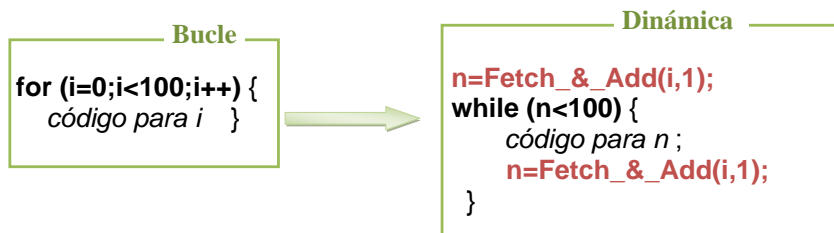




Nota: la variable i estaría inicializada a 0 (por ejemplo, se puede iniciar cuando se declara)

Se supone que el compilador no cambia de sitio "mov k, 0".

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



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



**Ejercicio 11.**



**Ejercicio 12.**



**Ejercicio 13.** Se ha extraído la siguiente implementación de cerrojo (spin-lock) para x86 del kernel de Linux (<http://lxr.free-electrons.com/source/arch/x86/include/asm/spinlock.h>):

```
typedef struct {
    unsigned int slock;
} raw_spinlock_t;

...
/*Para un número de procesadores menor que 256=2^8
-#if (NR_CPUS < 256)
...
-static __always_inline void __ticket_spin_lock(raw_spinlock_t *lock)
-{
-    short inc = 0x0100;
-
-    asm volatile (
-        "lock xaddw %w0, %1\n" /*w: se queda con los 16 bits menos significativos*/
-        "1: \t" /*b: se queda con el byte menos significativo*/
-        "cmpb %h0, %b0 \n\t" /*h: coge el byte que sigue al menos significativo*/
-        "je 2f \n\t" /*f: forward */
-        "rep ; nop \n\t" /*retardo, es equivalente a pause*/
-        "movb %1, %b0 \n\t"
-        /* don't need lfence here, because loads are in-order */
-        "jmp 1b \n" /*b: backward */
-        "2:"
-    );
-}
```



```
-      : "+Q" (inc), "+m" (lock->slock) /*%0 es inc, %1 es lock->slock */
- /*Q asigna cualquier registro al que se pueda acceder con rh: a, b, c y d; ej. ah, bh ...
*/
-      :
-      : "memory", "cc");
-}
-
-static __always_inline void __ticket_spin_unlock(raw_spinlock_t *lock)
-{
-    asm volatile( "incb %0" /*%0 es lock->slock */
-                  : "+m" (lock->slock)
-                  :
-                  : "memory", "cc");
-}
```

Conteste a las siguientes preguntas:

- (a) Utiliza una implementación de cerrojo con etiquetas ¿Cuál es el contador de adquisición y cuál es el contador de liberación?
- (b) Describa qué hace `xaddw %w0, %1` ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?
- (c) Describa qué hace `cmpb %h0, %b0` ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?
- (d) ¿Por qué cree que se usa el prefijo `lock` delante de la instrucción `xaddw`?

NOTAS: (1) Puede consultar las instrucciones en el manual de Intel con el repertorio de instrucciones (Volumen 2 o volúmenes 2A, 2B y 2C) que puede encontrar aquí <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.

(2) Si no recuerda la interfaz entre C/C++ y ensamblador en `gcc` (se ha presentado en Estructura de Computadores), consulte el manual de `gcc` aquí <http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Extended-Asm.html#Extended-Asm> (<http://gcc.gnu.org/onlinedocs/>)

### Solución

- (b) `lock->slock` contiene el contador de liberación en los bits de 0 a 7 liberación (`lock->slock[7...0]`) y el de adquisición en los bits de 8 a 15 (`lock->slock[15...8]`).
- (c) `xaddw %w0, %1` almacena en los 16 bits menos significativos del registro al que se ha asigna `inc` (%0) los 16 bits (sufijo w) menos significativos de `lock->slock` (%1) y asigna a `lock->slock` (contador de adquisición y contador de liberación) el resultado de sumarlo con `inc`. Como consecuencia: **(1)** incrementa en uno el contador de adquisición (`lock->slock[15...8]`) dado que `inc` tiene un 1 en el bit 8 (`inc` contiene `0x0100`) y **(2)** almacena en `inc[15...8]` (%h0) el valor de este contador antes de la modificación y en `inc[7...0]` (%b0) el valor del contador de liberación (`lock->slock[7...0]`).
- (d) `cmpb %h0, %b0` compara el valor actual del contador de liberación `inc[7...0]` (%b0) y el de adquisición `inc[15...8]` (%h0); es decir, resta ambos contadores modificando sólo el registro de estado. En las instrucciones posteriores se usa el resultado de la comparación (los bits de estado resultantes de la comparación). Si son iguales ambos contadores (bit z del registro de estado a 1), abandona la función `lock` del cerrojo, y si son distintos actualiza el valor del contador de liberación cargando lo que hay en `lock->slock[7...0]` en `inc[7...0]` (%b0)
- (e) Se requiere el prefijo `lock` para que la lectura y escritura en memoria que realiza la instrucción `xaddw` se hagan de forma atómica. Si `xaddw` no fuese atómica dos flujos de control podrían leer el mismo valor del contador de adquisición y, como consecuencia, más de un flujo podría entrar a la vez en una sección crítica.







2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores. Ejercicios y Cuestiones

## Tema 4. Arquitecturas con Paralelismo a nivel de Instrucción (ILP)

Material elaborado por los profesores responsables de la asignatura:  
Julio Ortega, Mancia Anguita

Licencia Creative Commons



### 1 Ejercicios

**Ejercicio 1.** Para el fragmento de código siguiente:

```
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
```

y suponiendo que se pueden captar, decodificar, y emitir cuatro instrucciones por ciclo, indique el orden en que se emitirán las instrucciones para cada uno de los siguientes casos:

- Una ventana de instrucciones centralizada con emisión ordenada
- Una ventana de instrucciones centralizada con emisión desordenada
- Una estación de reserva de tres líneas para cada unidad funcional, con envío ordenado.

Nota: considere que hay una unidad funcional para la carga (2 ciclos), otra para el almacenamiento (1 ciclo), tres para la suma/resta (1 ciclo), y una para la multiplicación (4 ciclos). También puede considerar que, en la práctica, no hay límite para el número de instrucciones que pueden almacenarse en la ventana de instrucciones o en el buffer de instrucciones.

#### Solución

A continuación se muestran la evolución temporal de las instrucciones en sus distintas etapas para cada una de las alternativas que se piden. En esas figuras se supone que tras la decodificación de una instrucción, ésta puede pasar a ejecutarse sin consumir ciclos de emisión si los operandos y la unidad funcional que necesita están disponibles. Por esta razón, no se indicarán explícitamente los ciclos dedicados a la emisión en los casos (a) y (b), y al envío en el caso (c)

a) Emisión ordenada con ventana centralizada

Como se muestra la tabla correspondiente, las instrucciones se empiezan a ejecutar en orden respetando las dependencias de datos o estructurales que existan. Por ejemplo la segunda instrucción de acceso a memoria lw debe esperar que termine la instrucción lw anterior porque solo hay una unidad de carga de memoria. La



emisión de la instrucción `add r3, r3, r4` debe esperar a que termine la ejecución de la instrucción de multiplicación que la precede.

La instrucción `add r3, r0, r0` debe esperar a que se hayan emitido las instrucciones que la preceden a pesar de que no depende de ellas y podría haberse emitido en el ciclo 3, si la emisión fuese desordenada.

También debe comprobarse que no se emiten más de cuatro instrucciones por ciclo. En este caso, como mucho, se emiten tres instrucciones, en el ciclo 11.

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

#### b) Emisión desordenada con ventana centralizada

En este caso, hay que respetar las dependencias de datos y las estructurales. En cuanto las instrucciones tengan una unidad disponible y los datos que necesitan se pueden emitir, siempre que no sean menos de cuatro instrucciones por ciclo las que se emitan. En este caso, como mucho se emiten tres instrucciones en los ciclos 4 y 5

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

#### c) Estación de reserva con tres líneas para cada unidad funcional, y envío ordenado.

En esta alternativa, se ha supuesto que las instrucciones decodificadas se emiten a una estación de reserva desde la que se accede a la unidad funcional correspondiente enviando las instrucciones de forma ordenada (por eso las dos instrucciones `sw` se ejecutan ordenadamente). La asignación de instrucciones de suma o resta a las tres estaciones de reserva que existen para suma/resta se ha hecho de forma alternativa pero



tratando de minimizar tiempos. Es decir, se ha asignado a una estación de reserva que no tuviera su unidad funcional correspondiente para obtener los tiempos más favorables.

ESTACIÓN DE RESERVA	INSTRUCCIÓN	1	2	3	4	5	6	7	8	9	10	11	12	13
LW	<i>lw</i> <i>r1</i> , 0x1ac	IF	ID	EX										
LW	<i>lw</i> <i>r2</i> , 0xc1f	IF	ID			EX								
ADD(1)	<i>add</i> <i>r3</i> , <i>r0</i> , <i>r0</i>	IF	ID	EX										
MULT(1)	<i>mul</i> <i>r4</i> , <i>r2</i> , <i>r1</i>	IF	ID					EX						
ADD(2)	<i>add</i> <i>r3</i> , <i>r3</i> , <i>r4</i>		IF	ID								EX		
ADD(3)	<i>add</i> <i>r5</i> , <i>r0</i> , 0x1ac		IF	ID	EX									
ADD(1)	<i>add</i> <i>r6</i> , <i>r0</i> , 0xc1f		IF	ID	EX									
ADD(3)	<i>sub</i> <i>r5</i> , <i>r5</i> , #4		IF	ID		EX								
ADD(1)	<i>sub</i> <i>r6</i> , <i>r6</i> , #4			IF	ID	EX								
SW	<i>sw</i> ( <i>r5</i> ), <i>r3</i>			IF	ID								EX	
SW	<i>sw</i> ( <i>r6</i> ), <i>r4</i>			IF	ID									EX

**Ejercicio 2.** Considere que el fragmento de código siguiente:

```

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) ← r6
10. sw     -8(r1), r5     ; M(r1-8) ← r5
11. sub    r2, r2, #16    ; r2 ← r2-16

```

se ejecuta en un procesador superescalar que es capaz de captar 4 instrucciones/ciclo, de decodificar 2 instrucciones/ciclo; de emitir utilizando una ventana de instrucciones centralizada 2 instrucciones/ciclo; de escribir hasta 2 resultados/ciclo en los registros correspondientes (registros de reorden, o registros de la arquitectura según el caso), y completar (o retirar) hasta 3 instrucciones/ciclo.

Indique el número de ciclos que tardaría en ejecutarse el programa suponiendo finalización ordenada y:

- a) Emisión ordenada
- b) Emisión desordenada

Nota: Considere que tiene una unidad funcional de carga (2 ciclos), una de almacenamiento (1 ciclo), tres unidades de suma/resta (1 ciclo), y una de multiplicación (6 ciclos), y que no hay limitaciones para el número de líneas de la cola de instrucciones, ventana de instrucciones, buffer de reorden, puertos de lectura/escritura etc.)

### Solución

a) En la emisión ordenada los instantes en los que las instrucciones empiezan a ejecutarse (etapa EX) deben estar ordenados de menor a mayor, a diferencia de lo que ocurre en la emisión desordenada. Se supone que



el procesador utiliza un buffer de reordenamiento (ROB) para que la finalización del procesamiento de las instrucciones sea ordenada. Por ello, las etapas WB de las instrucciones (momento en que se retiran las instrucciones del ROB y se escriben en los registros de la arquitectura) deben estar ordenadas (tanto en el caso de emisión ordenada como desordenada). La etapa marcada como ROB es la que corresponde a la escritura de los resultados en el ROB (se ha supuesto que las instrucciones de escritura en memoria también consumen un ciclo de escritura en el ROB y un ciclo de escritura WB en el banco de registros, aunque estos ciclos se podrían evitar para estas instrucciones).

También 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.

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

b) En el caso de emisión desordenada, la traza de ejecución de las instrucciones es la siguiente:

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



**Ejercicio 3.** En el problema anterior, (a) indique qué mejoras realizaría en el procesador para reducir el tiempo de ejecución en la mejor de las opciones sin cambiar el diseño de las unidades funcionales (multiplicador, sumador, etc.) y sin cambiar el tipo de memorias ni la interfaz entre procesador y memoria



(no varía el número de instrucciones captadas por ciclo. (b) ¿Qué pasaría si se reduce el tiempo de multiplicación a la mitad?.

### Solución

(a) En primer lugar, se considera que se decodifican el mismo número de instrucciones que se captan, ya que no existen limitaciones impuestas por las instrucciones al ritmo de decodificación (éste viene determinado por las posibilidades de los circuitos de decodificación y la capacidad para almacenar las instrucciones decodificadas hasta que se emitan). También se considera que no existen limitaciones para el número de instrucciones por ciclo que se emiten, escriben el ROB, y se retiran. Por último, se consideran que están disponibles todas las unidades funcionales que se necesitan para que no haya colisiones (riesgos estructurales).

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

Teniendo en cuenta las características de la traza, si se redujese el tiempo de la multiplicación a la mitad (es decir tres ciclos) el tiempo de ejecución de dicha traza se reduciría también en esos mismos tres ciclos ya que todas las instrucciones que siguen a la multiplicación se encuentran esperando a que termine esta para poder proseguir. Es decir, la operación de multiplicación es el cuello de botella en este caso.

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

Por lo tanto se tendrían 12 ciclos. Si se tiene en cuenta que se tienen 11 instrucciones, que el tiempo mínimo que tarda la primera instrucción en salir son 6 ciclos (lo tomamos como tiempo de latencia de inicio del cauce), y que el tiempo total de ejecución en este caso es de 12 ciclos, se puede escribir:



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

Y, si se despeja, se tiene que el procesador superescalar presenta una media de 0.6 ciclos por instrucción, o lo que es lo mismo, ejecuta 1.67 instrucciones por ciclo. Tiene un comportamiento superescalar, pero está muy lejos de las tres instrucciones por ciclo que pueden terminar como máximo.



**Ejercicio 4.** En el caso descrito en el problema 3, indique cómo evolucionaría el buffer de reorden, utilizado para implementar finalización ordenada, en la mejor de las opciones.

#### Solución

La Tabla que se proporciona a continuación muestra la evolución del buffer de reordenamiento (ROB) marcando en negrita los cambios que se producen en cada ciclo.

- En el ciclo 2 se decodifican las instrucciones (1) – (4) y se introducen en el ROB.
- En el ciclo 3 se decodifican las instrucciones (5) – (8) y se introducen en el ROB.
- En el ciclo 4 se decodifican las instrucciones (9) – (11) y se introducen en el ROB. Simultáneamente se almacena en el ROB los resultados de las instrucciones (2) y (3), cuya ejecución finalizó en el ciclo anterior.
- En el ciclo 5 se escribe en el ROB el resultado de la instrucción (1).
- En el ciclo 6 se retiran las tres primeras instrucciones y se escriben los resultados de las instrucciones (4), (5) y (11) en el ROB.
- En el ciclo 7 se retiran las instrucciones (4) y (5) y se escribe en el ROB el resultado de la instrucción (8).
- En el ciclo 8 se activa el bit valor válido de la instrucción (9) en el ROB, indicando que la instrucción de almacenamiento ya ha escrito en memoria.
- En el ciclo 9 se escribe el resultado de la instrucción (6) en el ROB.
- En el ciclo 10 se retira la instrucción (6) y se escribe el resultado de la instrucción (7) en el ROB.
- En el ciclo 11 se retiran las instrucciones (7), (8) y (9) y se indica que la instrucción (10) ya ha escrito en memoria.
- En el ciclo 12 se retiran las dos últimas instrucciones del ROB.

CICLO	#	CÓDIGO OPERACIÓN	REGISTRO DESTINO	VALOR	VALOR VÁLIDO
2	0	<b>Lw</b>	<b>r3</b>	–	<b>0</b>
	1	<b>Addi</b>	<b>r2</b>	–	<b>0</b>
	2	<b>Add</b>	<b>r1</b>	–	<b>0</b>
	3	<b>Lw</b>	<b>r4</b>	–	<b>0</b>



CICLO	#	CÓDIGO OPERACIÓN	REGISTRO DESTINO	VALOR	VALOR VÁLIDO
3	0	<i>Lw</i>	r3	–	0
	1	<i>Addi</i>	r2	–	0
	2	<i>Add</i>	r1	–	0
	3	<i>Lw</i>	r4	–	0
	4	<b><i>Lw</i></b>	<b>r5</b>	–	<b>0</b>
	5	<b><i>Mult</i></b>	<b>r6</b>	–	<b>0</b>
	6	<b><i>Add</i></b>	<b>r5</b>	–	<b>0</b>
	7	<b><i>Add</i></b>	<b>r6</b>	–	<b>0</b>
4	0	<i>Lw</i>	r3	–	0
	1	<b><i>Addi</i></b>	<b>r2</b>	<b>128</b>	<b>1</b>
	2	<b><i>Add</i></b>	<b>r1</b>	<b>0x0a</b>	<b>1</b>
	3	<i>Lw</i>	r4	–	0
	4	<i>Lw</i>	r5	–	0
	5	<i>Mult</i>	r6	–	0
	6	<i>Add</i>	r5	–	0
	7	<i>Add</i>	r6	–	0
	8	<b><i>Sw</i></b>	–	–	<b>0</b>
	9	<b><i>Sw</i></b>	–	–	<b>0</b>
	10	<b><i>Sub</i></b>	<b>r2</b>	–	<b>0</b>
5	0	<b><i>Lw</i></b>	<b>r3</b>	<b>[0x1a]</b>	<b>1</b>
	1	<i>Addi</i>	r2	128	1
	2	<i>Add</i>	r1	0x0a	1
	3	<i>Lw</i>	r4	–	0
	4	<i>Lw</i>	r5	–	0
	5	<i>Mult</i>	r6	–	0
	6	<i>Add</i>	r5	–	0
	7	<i>Add</i>	r6	–	0
	8	<i>Sw</i>	–	–	0
	9	<i>Sw</i>	–	–	0
	10	<i>Sub</i>	r2	–	0
6	3	<b><i>Lw</i></b>	<b>r4</b>	<b>[0x0a]</b>	<b>1</b>
	4	<b><i>Lw</i></b>	<b>r5</b>	<b>[0x0a – 8]</b>	<b>1</b>
	5	<i>Mult</i>	r6	–	0
	6	<i>Add</i>	r5	–	0
	7	<i>Add</i>	r6	–	0
	8	<i>Sw</i>	–	–	0
	9	<i>Sw</i>	–	–	0
	10	<b><i>Sub</i></b>	<b>r2</b>	<b>112</b>	<b>1</b>



CICLO	#	CÓDIGO OPERACIÓN	REGISTRO DESTINO	VALOR	VALOR VÁLIDO
7	5	<i>mult</i>	r6	–	0
	6	<i>add</i>	r5	–	0
	7	<b><i>add</i></b>	<b>r6</b>	<b>r4 + r3</b>	<b>1</b>
	8	<i>sw</i>	–	–	0
	9	<i>sw</i>	–	–	0
	10	<i>sub</i>	r2	112	1
8	5	<i>mult</i>	r6	–	0
	6	<i>add</i>	r5	–	0
	7	<i>add</i>	r6	r4 + r3	1
	8	<b><i>sw</i></b>	–	–	<b>1</b>
	9	<i>sw</i>	–	–	0
	10	<i>sub</i>	r2	112	1
9	5	<b><i>mult</i></b>	<b>r6</b>	<b>r5 × r3</b>	<b>1</b>
	6	<i>add</i>	r5	–	0
	7	<i>add</i>	r6	r4 + r3	1
	8	<i>sw</i>	–	–	1
	9	<i>sw</i>	–	–	0
	10	<i>sub</i>	r2	112	1
10	6	<b><i>add</i></b>	<b>r5</b>	<b>r6 + r3</b>	<b>1</b>
	7	<i>add</i>	r6	r4 + r3	1
	8	<i>sw</i>	–	–	1
	9	<i>sw</i>	–	–	0
	10	<i>sub</i>	r2	112	1
11	9	<b><i>sw</i></b>	–	–	<b>1</b>
	10	<i>sub</i>	r2	112	1

□

**Ejercicio 5.** En un procesador superescalar con renombramiento de registros se utiliza un buffer de renombramiento para implementar el mismo. Indique como evolucionarían los registros de renombramiento al realizar el renombramiento para las instrucciones:

1. `mul r2,r0,r1` ;  $r2 \leftarrow r0 * r1$
2. `add r3,r1,r2` ;  $r3 \leftarrow r1 + r2$
3. `sub r2,r0,r1` ;  $r2 \leftarrow r0 - r1$
4. `add r3,r3,r2` ;  $r3 \leftarrow r3 + r2$

### Solución

Las instrucciones se renombra de forma ordenada, aunque podrían emitirse desordenadamente a medida que estén disponibles los operandos y las unidades funcionales. La estructura típica de un buffer de renombramiento con acceso asociativo podría ser la que se muestra a continuación, en la que se ha supuesto que se han hecho las asignaciones (renombrados) de r0 y r1 (las líneas 0 y 1 tienen valores de “Entrada válida” a 1 correspondiente a los registros r0 y r1, con sus valores válidos correspondientes y los bits de último a 1 indicando que han sido los últimos renombramientos de esos registros)

:





#	Entrada válida	Registro destino	Valor	Valor Válido	Bit de Último
0	1	r0	[r0]	1	1
1	1	r1	[r1]	1	1
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
8	0				

Al decodificarse la instrucción `mul r2,r0,r1` los valores de r0 y r1 se tomarán de las líneas 0 y 1 del buffer de renombrado y se renombrará r2. La instrucción se puede emitir para su ejecución y el resultado de la misma se almacenará en la línea 2 del buffer de renombrado. El contenido del campo “Valor válido” de esta línea será igual a 0 hasta que el resultado no se haya obtenido

#	Entrada válida	Registro destino	Valor	Valor Válido	Bit de Último
0	1	r0	[r0]	1	1
1	1	r1	[r1]	1	1
2	1	r2		0	1
3	0				
4	0				
5	0				
6	0				
7	0				
8	0				

Después se considera la instrucción `add r3,r1,r2`, como r2 no está disponible no se podrá empezar a ejecutar, pero se utilizará una marca en la estación de reserva indicando que se espera el valor que se escribirá en la línea 2. También se hace el renombrado de r3. El valor de r1 sí se puede tomar de la línea 1.

#	Entrada válida	Registro destino	Valor	Valor Válido	Bit de Último
0	1	r0	[r0]	1	1
1	1	r1	[r1]	1	1
2	1	r2		0	1
3	1	r3		0	1
4	0				
5	0				
6	0				
7	0				
8	0				

Cuando se decodifica la instrucción `sub r2,r0,r1`, se realiza otro renombramiento de r2. Como r0 y r1 tienen valores válidos, esta instrucción podría empezar a ejecutarse si existe una unidad funcional disponible. El resultado de esta operación se almacenará en la línea 4 (último renombrado de r2) y de ahí se tomarán los operandos r2 en instrucciones posteriores (para eso se utiliza el “Bit de Último”):



#	Entrada válida	Registro destino	Valor	Valor Válido	Bit de Último
0	1	r0	[r0]	1	1
1	1	r1	[r1]	1	1
2	1	r2		0	0
3	1	r3		0	1
4	1	r2		0	1
5	0				
6	0				
7	0				
8	0				

La última instrucción, `add r3,r3,r2`, origina otro renombrado de r3. Sus operandos r3 y r2 se tomarían de las líneas 3 y 4 respectivamente porque son los últimos renombrados que se han hecho para r3 y r2. Como no tienen valores válidos, se utilizarán las marcas correspondientes a dichas líneas 3 y 4 del buffer para que se escriban los resultados correspondientes en la línea de la ventana de instrucciones donde se almacena esta instrucción hasta que pueda emitirse.

#	Entrada válida	Registro destino	Valor	Valor Válido	Bit de Último
0	1	r0	[r0]	1	1
1	1	r1	[r1]	1	1
2	1	r2		0	0
3	1	r3		0	0
4	1	r2		0	1
5	1	r3		0	1
6	0				
7	0				
8	0				

Cuando termine `mul r2,r0,r1`, se escribirá el resultado en la línea 1 y podrá continuar `add r3,r1,r2`, y cuando termine `sub r2,r0,r1`, se escribirá el resultado en la línea 4. Cuando termine `add r3,r1,r2`, se escribirá el resultado en la línea 3 y podrá continuar `add r3,r3,r2`.



### Ejercicio 6. ◀

**Ejercicio 7.** En la situación descrita en el problema anterior. ¿Cuál de los tres esquemas es más eficaz por término medio si hay un 25% de probabilidades de que  $c$  sea menor o igual a 0, un 30% de que sea mayor o igual a 10; y un 45% de que sea cualquier número entre 1 y 9, siendo todos equiprobables?

### Solución

En el ejercicio 6 se ha determinado la penalización de la secuencia para cada tipo de predictor, en este ejercicio se trata de utilizar la distribución de probabilidad de la variable  $c$  que se da en el enunciado para evaluar la penalización media de cada predictor, y por lo tanto cuál es el mejor para este código y estas circunstancias. Para el caso del predictor fijo, y teniendo en cuenta que todos los valores de  $c$  entre 1 y 9 son equiprobables, tenemos:



$$P_{\text{fijo}} = 0.25 \times P_{\text{fijo}_1} + 0.45 \times P_{\text{fijo}_2} + 0.30 \times P_{\text{fijo}_3} = 0.25 \times 44 + 0.45 \times \left( \frac{\sum_{c=1}^9 (10-c) \times 4}{9} \right) + 0.30 \times 0 = 20 \text{ ciclos}$$

Para el predictor estático tenemos:

$$P_{\text{estático}} = 0.25 \times P_{\text{estático}_1} + 0.45 \times P_{\text{estático}_2} + 0.30 \times P_{\text{estático}_3} = 0.25 \times 4 + 0.45 \times 4 + 0.30 \times 4 = 4 \text{ ciclos}$$

Y para el predictor dinámico:

$$P_{\text{dinámico}} = 0.25 \times P_{\text{dinámico}_1} + 0.45 \times P_{\text{dinámico}_2} + 0.30 \times P_{\text{dinámico}_3} = 0.25 \times 8 + 0.45 \times 8 + 0.30 \times 0 = 5.6 \text{ ciclos}$$

Como se puede ver, para este bucle, y para la distribución de probabilidad de los valores de  $c$ , el esquema de predicción más eficiente corresponde a la predicción estática.

En cualquier caso, esta situación depende de las probabilidades de los distintos valores de  $c$ . Si, por ejemplo, la probabilidad de que  $c$  esté entre 1 y 9 fuera de 0.15, la penalización para la predicción dinámica con 1 bit de historia sería de 3.2 ciclos, mientras que la predicción estática seguiría presentando una penalización de 4 ciclos.



**Ejercicio 8.** En un programa, una instrucción de salto condicional (a una dirección de salto anterior) dada tiene el siguiente comportamiento en una ejecución de dicho programa:

SSNNNSSNSNSNSSSSN

donde S indica que se produce el salto y N que no. Indique la penalización efectiva que se introduce si se utiliza:

- a) Predicción fija (siempre se considera que se no se va a producir el salto)
- b) Predicción estática (si el desplazamiento es negativo se toma y si es positivo no)
- c) Predicción dinámica con dos bits, inicialmente en el estado (11).
- d) Predicción dinámica con tres bits, inicialmente en el estado (111).

Nota: La penalización por saltos incorrectamente predichos es de 5 ciclos y para los saltos correctamente predichos es 0 ciclos.

### Solución

En el caso de usar predicción fija, se produciría un fallo del predictor cada vez que se tome el salto, tal y como muestra la Tabla siguiente. Por tanto, la penalización total sería de:

$$P_{\text{fijo}} = F_{\text{fijo}} \times P = 11 \times 5 = 55 \text{ ciclos}$$



PREDICCIÓN FIJA	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
EJECUCIÓN	S	S	N	N	N	S	S	N	S	N	S	N	S	S	S	S	S	N
PENALIZACIÓN	P	P				P	P		P		P		P	P	P	P	P	

Si se usa el predictor estático, como la dirección de destino del salto es anterior, se producirá un fallo en la predicción cuando no se produzca el salto, tal y como muestra la tabla correspondiente. Por tanto, la penalización total sería de:

$$P_{\text{estático}} = N_{\text{estático}} \times P = 7 \times 5 = 35 \text{ ciclos}$$

PREDICCIÓN ESTÁTICA	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
EJECUCIÓN	S	S	N	N	N	S	S	N	S	N	S	N	S	S	S	S	S	N
PENALIZACIÓN			P	P	P			P		P		P						P

En cuanto al predictor dinámico con dos bits de historia, la tabla siguiente muestra el estado del predictor antes de ejecutar el salto, la predicción que realiza el predictor, el comportamiento del salto y si se produce o no penalización. Teniendo en cuenta los fallos que se producen, la penalización total sería de:

$$P_{2 \text{ bits}} = F_{2 \text{ bits}} \times P = 11 \times 5 = 55 \text{ ciclos}$$

ESTADO	11	11	11	10	01	00	01	10	01	10	01	10	01	10	11	11	11	11
PREDICCIÓN DINÁMICA (2 BITS)	S	S	S	S	N	N	N	S	N	S	N	S	N	S	S	S	S	S
EJECUCIÓN	S	S	N	N	N	S	S	N	S	N	S	N	S	S	S	S	S	N
PENALIZACIÓN			P	P		P	P	P	P	P	P	P	P					P

Por último, para el predictor dinámico de tres bits, en la tabla siguiente se indican los bits de historia que existen antes de que se ejecute la instrucción, la predicción que determinan esos bits, y lo que finalmente ocurre (según se indica en la secuencia objeto del problema). La última fila indica si ha habido penalización (no coinciden la predicción y lo que ocurre al final). Teniendo en cuenta esta información, tenemos que la penalización total es de:

$$P_{3 \text{ bits}} = F_{3 \text{ bits}} \times P = 10 \times 5 = 50 \text{ ciclos}$$



ESTADO	111	111	111	011	001	000	100	110	011	101	010	101	010	101	110	111	111	111
PREDICCIÓN DINÁMICA (3 BITS)	S	S	S	S	N	N	N	S	S	S	N	S	N	S	S	S	S	S
EJECUCIÓN	S	S	N	N	N	S	S	N	S	N	S	N	S	S	S	S	S	N
PENALIZACIÓN			P	P		P	P	P		P	P	P	P					P

Como se puede ver, en la secuencia de ejecuciones de la instrucción de salto considerada en este problema, el mejor esquema de predicción es la predicción estática que aquí se utiliza. El esquema de predicción de salto dinámico con dos bits es igual de bueno (o malo) que la predicción fija.

Así, se puede indicar que la eficacia de un esquema de salto depende del perfil de saltos a que de lugar la correspondiente instrucción de salto condicional. En la práctica, los esquemas de predicción dinámica suelen funcionar mejor que los de predicción estática porque las instrucciones de salto suelen repetir el comportamiento de su ejecución previa. En ese sentido, la secuencia utilizada en este problema es bastante atípica.

**Ejercicio 9.** (NOTA: el tratamiento de excepciones no entra en el examen) ◀

**Ejercicio 10.** En un procesador todas las instrucciones pueden predicarse. Para establecer los valores de los predicados se utilizan intrucciones de comparación (cmp) con el formato (p) p1, p2 cmp.cnd x,y donde cnd es la condición que se comprueba entre x e y (lt, ge, eq, ne). Si la condición es verdadera p1=1 y p2=0, y si es falsa, p1=0 y p2=1. La instrucción sólo se ejecuta si el predicado p=1 (habrá sido establecido por otra instrucción de comparación).

En estas condiciones, utilice instrucciones con predicado para escribir sin ninguna instrucción de salto el siguiente código:

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

### Solución

A continuación se muestra el código que implementa el programa anterior usando predicados:

```
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
(p1)    p3         cmp.ne    r0, r0 ; Inicializamos p3 a 0
(p1)    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      r5, x      ; Almacenamos el resultado
```



## 2 Cuestiones

**Cuestión 1.** ¿Qué tienen en común un procesador superescalar y uno VLIW? ¿En qué se diferencian?

### Solución

Tanto un procesador superescalar como uno VLIW son procesadores segmentados que aprovechan el paralelismo entre instrucciones (ILP) procesando varias instrucciones u operaciones escalares en cada una de ellas. La diferencia principal entre ellos es que mientras que el procesador superescalar incluye elementos hardware para realizar la planificación de instrucciones (enviar a ejecutar las instrucciones decodificadas independientes para las que existen recursos de ejecución disponibles) dinámicamente, los procesadores superescalares se basan en el compilador para realizar dicha planificación (planificación estática).



**Cuestión 2.** ¿Qué es un buffer de renombramiento? ¿Qué es un buffer de reordenamiento? ¿Existe alguna relación entre ambos?

### Solución

Un buffer de renombramiento es un recurso presente en los procesadores superescalares que permite asignar almacenamiento temporal a los datos asignados a registros del banco de registros de la arquitectura. La asignación de registros de la arquitectura la realiza el compilador o el programador en ensamblador. Mediante la asignación de registros temporales a los registros de la arquitectura se implementa el renombramiento de estos últimos con el fin de eliminar riesgos (dependencias) WAW y WAR.

Un buffer de reordenamiento permite implementar la finalización ordenada de las instrucciones después su ejecución.

Es posible implementar el renombramiento de registros aprovechando la estructura del buffer de reordenamiento.



**Cuestión 3.** ¿Qué es una ventana de instrucciones? ¿Y una estación de reserva? ¿Existe alguna relación entre ellas?

### Solución

Una ventana de instrucciones es la estructura a la que pasan las instrucciones tras ser decodificadas para ser emitidas desde ahí a la unidad de datos donde se ejecutarán cuando dicha unidad esté libre y los operandos que se necesitan para realizar la correspondiente operación estén disponibles. La emisión desde esa ventana de instrucciones puede ser ordenada o desordenada.

Una estación de reserva es una estructura presente en los procesadores superescalares en los que se distribuye la ventana de instrucciones de forma que en las instrucciones decodificadas se envían a una u otra estación de reserva según la unidad funcional (o el tipo de unidad funcional) donde se va a ejecutar la instrucción.

Una ventana de instrucciones puede considerarse un caso particular de estación de reserva desde la que se puede acceder a todas las unidades funcionales del procesador. La transferencia de una instrucción desde la



unidad de decodificación a la estación de reserva se denomina emisión, y desde la estación de reserva correspondiente a la unidad funcional, envío.

**Cuestión 4.** ¿Qué utilidad tiene la predicación de instrucciones? ¿Es exclusiva de los procesadores VLIW?

#### Solución

La predicación de instrucciones permite eliminar ciertas instrucciones de salto condicional de los códigos. Por lo tanto, contribuye a reducir el número de riesgos de control (saltos) en la ejecución de programas en procesadores segmentados.

Aunque es una técnica muy importante en el contexto de los procesadores VLIW, porque al reducir instrucciones de salto condicional se contribuye a aumentar el tamaño de los bloques básicos (conjunto de instrucciones con un punto de entrada y un punto de salida), existe la posibilidad de utilizar predicados tanto en procesadores superescalares como VLIW. Por ejemplo, en el conjunto de instrucciones de ARM se pueden predicar todas las instrucciones.

**Cuestión 5.** ¿En qué momento se lleva a cabo la predicción estática de saltos condiciones? ¿Se puede aprovechar la predicción estática en un procesador con predicción dinámica de saltos?

#### Solución

La predicción estática se produce en el momento de la compilación teniendo en cuenta el sentido más probable del salto (sobre todo es efectiva en instrucciones de control de bucles).

En el caso de los procesadores con predicción dinámica, se suele tener en cuenta algún tipo de predicción estática para cuando no se dispone de información de historia de la instrucción de salto (en la primera aparición de la misma).

**Cuestión 6.** ¿Qué procesadores dependen más de la capacidad del compilador, un procesador superescalar o uno VLIW?

#### Solución

Los procesadores superescalares pueden mejorar su rendimiento gracias a algunas técnicas aplicadas por el compilador, sobre todo en los procesadores de Intel a partir de la microarquitectura P6 que llevaba a cabo una traducción de las instrucciones del repertorio x86 a un conjunto de microoperaciones internas con estructuras fijas como las instrucciones de los repertorios RISC. Sin embargo, el papel del compilador en un procesador VLIW es fundamental, ya que es él el que se encarga de realizar la planificación de las instrucciones para aprovechar el paralelismo entre instrucciones. En los procesadores superescalares existen estructuras como las estaciones de reserva, los buffer de renombramiento, los buffer de reordenamiento, etc. que extraen el paralelismo ILP dinámicamente.

**Cuestión 7.** ¿Qué procesadores tienen microarquitecturas con mayor complejidad hardware, los superescalares o los VLIW? Indique algún recurso que esté presente en un procesador superescalar y no sea necesario en uno VLIW.

#### Solución



Los procesadores superescalares son los que tienen una microarquitectura más compleja. Precisamente, la idea de dejar que sea el compilador el responsable de extraer el paralelismo entre instrucciones perseguía conseguir núcleos de procesamiento más sencillos desde el punto de vista hardware para reducir el consumo energético y/o sustituir ciertos elementos hardware por un mayor número de unidades funcionales que permitiesen aumentar el número de operaciones finalizadas por ciclo.

Estructuras como las estaciones de reserva, los buffers de renombramiento, y los buffers de reordenamiento son típicas de los procesadores superescalares.

**Cuestión 8.** Haga una lista con 5 microprocesadores superescalares o VLIW que se hayan comercializado en los últimos 5 años indicando alguna de sus características (frecuencias, núcleos, tamaño de cache, etc.)

### Solución

Procesador	Año	Frecuencia (GHz)	Fabricante	Observaciones
Core i7	2008	2.66 – 3.33	Intel	Intel x86-64. HyperThreading Tecnología de 45 nm o 32 nm QPI Transistores: 1.170 millones en el Core i7 980x, con 6 núcleos y 12 MB de memoria caché
Phenom II	2008	2.5 -3.8	AMD	x86-64 Proceso: 45 nm Serie 1000: 6 núcleos con 3MB de cache L2 (512KB por núcleo) y 6MB de cache L3 compartidos
Itanium 9300 (Tukwila)	2010	1.33 – 1.73	Intel	2 – 4 Cores Cache L2: 256 kB (D)+ 512 kB (I) Cache L3: 10–24 MB QPI Proceso: 65 nm
POWER7	2010	2.4 – 4.25	IBM	Repertorio: Power Arch. 4, 6, 8 Cores Proceso 45 nm CacheL1 32+32KB/core CacheL2 256 KB/core Cache L3 32 MB
Xeon 5600	2010	2.4 – 3.46	Intel	6 Cores (HT: 12 threads) QPI Cache 12 MB





