Actividad 1.1 *Autor: Longinos Recuero Bustos*

Suponiendo que se aplica una mejora a una máquina de tal forma que el rendimiento es 20 veces superior al que tenía y considerando que la mejora únicamente se puede aplicar durante el 20% del tiempo, ¿cuál es la ganancia obtenida?

Como el rendimiento es 20 veces superior, entonces el factor $p = 20$.

La mejora sólo se puede aplicar durante el 20% del tiempo, tenemos: $1 - f = 0,2 \rightarrow f = 0,8$

$$S_p = \frac{p}{1 + f\,(p-1)} = \frac{20}{1 + 0,8 \cdot (20-1)} = 1,234$$

Actividad 1.2 *Autor: Longinos Recuero Bustos*

Tras añadir un nuevo procesador a un ordenador se logra un aumento de la velocidad de ejecución en un factor de 8. Se observa que tras aplicar esta mejora, el 60% del tiempo de ejecución se está utilizando un nuevo procesador. ¿Qué porcentaje del tiempo de ejecución original se ha reducido gracias a al mejora?

Como la velocidad de ejecución es 8 veces superior, entonces el factor $p = 8$.

La mejora se puede aplicar durante el 60% del tiempo, tenemos: $1 - f = 0,6 \rightarrow f = 0,4$

$$S_p = \frac{p}{1 + f\,(p-1)} = \frac{8}{1 + 0,4 \cdot (8-1)} = \frac{8}{3,8} = 2,1$$

$$S_p = \frac{T_{CPU\_original}}{T_{CPU\_mejorada}}; \qquad t_m = \frac{t_o}{2,1}$$

$$\left.\begin{array}{l} Si\ t_o \rightarrow 100\% \\ t_m \rightarrow\ x \end{array}\right\} \qquad x = \frac{100 \cdot t_m}{t_o} = \frac{\frac{100 \cdot t_o}{2,1}}{t_o} = 47,6\%$$

El porcentaje del tiempo de ejecución original que se ha reducido gracias a la mejora es:

$100\% - 47,6\% = 52,4\%$

Solución propuesta por Demetrio Quirós Santos

Tiempo de ejecución de mejora: $0,6 \cdot t + 0,4 \cdot 8t = 3,8t$.

$$\left.\begin{array}{l} 1 \rightarrow 8 \\ x \rightarrow 3,8 \end{array}\right\} \qquad x = \frac{3,8}{8} = 0,475$$

Reducción: $\qquad 1 - \left(\frac{3,8}{8}\right) = 0,525 \rightarrow 52,5\%$

Actividad 1.3 *Autor: Sebastián Dormido Cantó*

Un procesador sin segmentación necesita 400 nseg. para procesar una instrucción. Con respecto a este procesador, calcular la aceleración que se obtiene en los dos casos siguientes:

a) Un procesador A dotado de una segmentación de 5 etapas, consumiendo cada etapa el mismo tiempo. Cada etapa ocasiona una sobrecarga de 10 nseg. no existiendo ningún tipo de detención en la segmentación.

De acuerdo con el enunciado el tiempo medio de ejecución de una instrucción en el procesador sin segmentar es de 400 nseg. La segmentación de 5 etapas de este apartado se caracteriza por acortar el tiempo medio de ejecución de una instrucción a 90 nseg.:

$$\frac{400\ nseg}{5\ etapas}+10\ nseg=90\ nseg$$

Por lo tanto, la aceleración obtenida por la máquina A con respecto a la máquina sin segmentar es 4,45:

$$\frac{400\ nseg}{90\ nseg}=4,45$$

b) Un procesador b con una segmentación de 5 etapas, consumiendo cada una de ellas 60 nseg., 60 nseg., 80 nseg. y 100 nseg. respectivamente, y siendo la sobrecarga por cada etapa de 10 nseg. un 40% de todas las instrucciones de la segmentación son detenidas durante un ciclo de reloj y un 10% durante dos ciclos.

La etapa más lenta es la que dicta la velocidad de las restantes etapas, por lo que cada etapa consumirá 110 nseg. (100 nseg. más los 10 nseg. de retardo).

El 10% de todas las instrucciones ocasionan una detención de dos ciclos, por lo que consumen 330 nseg. $\left(3\ ciclos\cdot 110\ nseg\right)$. Por otra parte, un 40% ocasiona una detención de un ciclo, consumiendo 220 nseg. $\left(2\ ciclos\cdot 110\ nseg\right)$

El resto de las instrucciones, un 50%, no provocan detenciones, empleando sólo un ciclo de reloj (110 nseg.).

De acuerdo con esto, el tiempo medio consumido por una instrucción es:

$0,1\cdot 330\ nseg+0,4\cdot 220\ nseg+0,50\cdot 110\ nseg=176\ nseg$.

Por lo tanto, la aceleración obtenida por la máquina B con respecto a la máquina sin segmentar es de 2,27:

$$\frac{400\ nseg}{176\ nseg}=2,27$$

Aclaración de Sebastián Dormido Cantó:

Una etapa necesita de un ciclo para ejecutarse (110 ns) y en un caso tiene una detención de 2 ciclos (220 ns) por lo que ese 10% necesita 330ns para ejecutarse

En el otro caso ocasiona un retraso de un ciclo, por lo que a los 110 ns de su ejecución hay que sumar otros 110ns de la detención, por lo que un 40% necesita 220 ns

Actividad 1.4 *Autor: Sebastián Dormido Cantó*

Al diseño A de un procesador se le propone añadir una instrucción ALU que tenga un operando fuente en memoria (el otro se encuentra cargado en un registro en el 25% de las operaciones de la ALU) obteniendo de esta manera el diseño del procesador B. si en este nuevo diseño B aumenta en 1 el número de ciclos de reloj de los saltos y suponiendo que la nueva instrucción consume 2 ciclos, ¿cuál de los dos diseños es el más rápido considerando el recuento de instrucciones de la siguiente tabla?

Operación	Frecuencia	Ciclos/instrucción
ALU	43%	1
Cargas	21%	2
Almacenamiento	12%	2
Saltos	24%	2

CPI original: número medio de ciclos por instrucción.

$$CPI\ original = 0,43 \cdot 1 + 0,21 \cdot 2 + 0,12 \cdot 2 + 0,24 \cdot 2 = 1,57$$

$$Tiempo\ original\ = 1,57 \cdot Recuento\ original\ de\ instrucciones \cdot Duración\ de\ ciclo$$

Al realizar el recuento en el nuevo sistema hay que tener en cuenta las instrucciones de la ALU y carga que desaparecen a causa de la nueva instrucción, así como las que surgen. Tenemos así:

ALU: 43% − 43% × 25% = 32.25%
Cargas: 21% − 43% × 25% = 10.25%
Almacenamientos: 12 %
Saltos: 24%
Nuevas: 43% × 25% = 10.75%
TOTAL: 89.25%

Para proceder a calcular el CPI necesitamos normalizar el recuento de instrucciones que hemos obtenido. De acuerdo con esto, el factor de normalización es 1 / 0,8925 = 1,1204.

Tendremos los siguientes valores:

ALU: 32.25% × 1.1204 = 36.1329%
Cargas: 10.25% × 1.1204 = 11.4841%
Almacenamientos: 12% × 1.1204 = 13.4448%
Saltos: 24% × 1.1204 = 26.8896%
Nuevas: 10.75% × 1.1204 = 12.0443%
TOTAL: 100%

El nuevo CPI y el nuevo tiempo de ejecución son:

$$CPI\ nuevo = \begin{pmatrix} 0,361329 \cdot 1\ ciclo + 0,114841 \cdot 2\ ciclos + 0,134448 \cdot 2\ ciclos + \\ 0,268896 \cdot 3\ ciclos + 0,120443 \cdot 2\ ciclos \end{pmatrix} = 1,908$$

$$Tiempo\ nuevo\ ejecución = \left(0,8925 \cdot Recuento\ original\ de\ instrucciones\right) \cdot 1,908 \cdot Duración\ de\ ciclo$$
$$= 1,703 \cdot Recuento\ original\ de\ instrucciones \cdot Duración\ de\ ciclo$$

Por lo tanto, el sistema original es un 8,47% más rápido que el nuevo:

$$\frac{1,703 - 1,57}{1,57} \cdot 100 = 8,47\%$$

Actividad 1.5 *Autor: Sebastián Dormido Cantó*

Suponga que se tiene un procesador segmentado en el que el 20% de las instrucciones son cargas y un 50% de las veces la instrucción siguiente se detiene un ciclo de reloj debido a un riesgo por dependencia de datos. Además, el 10% de las instrucciones de carga producen un fallo de caché que se tarda 4 ciclos en resolver. Ignorando cualquier otro tipo de riesgo, ¿cuántas veces es más rápida la segmentación ideal de CPI = 1 que esta nueva segmentación?

La máquina ideal será más rápida según el cociente entre el CPI de la máquina con segmentación ideal que es 1, y el CPI de la máquina cuyas características se explican en el enunciado. Teniendo en cuenta el riesgo por dependencia de datos, el CPI de la instrucción siguiente a la de carga es:

CPI siguiente instrucción = 1 *ciclo* + 1 *ciclo* × 50% *detenciones* = 1,5 *ciclos*

Debido a los fallos de la caché, el CPI de una instrucción de carga es:

CPI instrucción de carga = 1 *ciclo* + 4 *ciclo* × 10% = 1,4 *ciclos*

Por lo tanto, el CPI de la máquina segmentada es:

CPI = 60% × 1 *ciclo* + 20% × 1,5 *ciclos* + 20% × 1,4 *ciclos* = 1,18 *ciclos*

De acuerdo con esto, la máquina ideal es un 18% más rápida.

Actividad 1.6 *Autor: Sebastián Dormido Cantó*

Un procesador sin segmentación necesita 200 nseg. para procesar una instrucción. Con respecto a este procesador, calcular la aceleración que se obtiene en los dos siguientes casos:

a) Un procesador A dotado de una segmentación de 5 etapas, consumiendo cada etapa el mismo tiempo. Cada etapa ocasiona una sobrecarga de 4 nseg., no existiendo ningún tipo de detención en la segmentación

De acuerdo con el enunciado el tiempo medio de ejecución de una instrucción en el procesador sin segmentar es de 200 nseg. La segmentación de 5 etapas de este apartado se caracteriza por acortar el tiempo medio de ejecución de una instrucción a 44 nseg:

- (200 nseg / 5 etapas) + 4 nseg = 44 nseg

Por lo tanto la aceleración obtenida por la máquina A con respecto a la máquina sin segmentar es 4,54:

- 200 nseg / 44 nseg = 4,54

b) Un procesador B con una segmentación de 5 etapas, consumiendo cada una de ellas 30 nseg., 30 nseg., 40 nseg., 50 nseg. y 50 nseg., respectivamente, y siendo la sobrecarga por cada etapa de 4 nseg. un 20% de todas las instrucciones de la segmentación son detenidas durante un ciclo de reloj y un 5% durante dos ciclos.

La etapa más lenta es la que dicta la velocidad de las restantes etapas, por lo que cada etapa consumirá 54 nseg. (50 nseg. más los 4 nseg. de retardo).

El 20% de todas las instrucciones ocasionan una detención de 1 ciclo, por lo que consumen 108 nseg. (2 ciclos * 54). Por otra parte, un 5% ocasiona una detención de 2 ciclos, consumiendo 162 nseg. (3 ciclos * 54).

El resto de las instrucciones, un 75%, no provocan detenciones, empleando sólo un ciclo de reloj (54 nseg.). De acuerdo con esto, el tiempo medio consumido por una instrucción es:

- 0,2 * 108 nseg. + 0,05 * 162 nseg. + 0,75 * 54 nseg. = 70,2 nseg.

Por tanto, la aceleración obtenida por la máquina B con respecto a la máquina sin segmentar es de:

- 200 nseg. / 70,2 nseg. = 2,85

Sea una máquina de carga/almacenamiento a la que se quiere añadir un modo de direccionamiento registro-memoria, de tal forma que se puedan sustituir secuencias de:

```
LD R1, 0(Rb)
ADD R2, R2, R1
```

por:

```
ADD R2, 0(Rb)
```

Suponga que la nueva instrucción hace que el ciclo de reloj se incremente un 10% pero no afecta en ninguna forma al CPI. Las frecuencias de instrucciones para esta máquina son: 18% de saltos, 36% de transferencias y 46% de operaciones ALU, siendo 2/3 de las instrucciones de carga y el resto de almacenamiento. Se pide:

- ¿Qué porcentaje de las instrucciones de carga debe eliminarse de la máquina con la nueva instrucción para que como máximo tenga el mismo rendimiento?

En el problema asumimos que todas las instrucciones de transferencia son de carga.

Para que los dos tipos de máquinas tengan el mismo rendimiento debemos igualar las siguientes expresiones:

$$\textbf{\textit{Rendimiento}}_{\textbf{\textit{nuevo}}} = \textbf{\textit{Rendimiento}}_{\textbf{\textit{viejo}}}$$

El rendimiento de los dos tipos de CPU se expresa como:

$$\textbf{\textit{Rendimiento}}_{\textbf{\textit{viejo}}} = \textbf{\textit{Recuento}}_{\textbf{\textit{viejo}}} \textbf{\textit{ x CPI x Ciclo}}_{\textbf{\textit{viejo}}}$$
$$\textbf{\textit{Rendimiento}}_{\textbf{\textit{nuevo}}} = \textbf{\textit{Recuento}}_{\textbf{\textit{nuevo}}} \textbf{\textit{ x CPI x Ciclo}}_{\textbf{\textit{nuevo}}}$$

y teniendo en cuenta las duraciones de los ciclos de reloj:

$$\textbf{\textit{Ciclo}}_{\textbf{\textit{nuevo}}} = \textbf{\textit{Ciclo}}_{\textbf{\textit{viejo}}} \textbf{\textit{ x 1,1}}$$

Se deduce:

$$\textbf{\textit{Rendimiento}}_{\textbf{\textit{viejo}}} = \textbf{\textit{Recuento}}_{\textbf{\textit{viejo}}} \textbf{\textit{ x CPI x Ciclo}}_{\textbf{\textit{viejo}}}$$

$$\textbf{\textit{Rendimiento}}_{\textbf{\textit{nuevo}}} = \textbf{\textit{Recuento}}_{\textbf{\textit{nuevo}}} \textbf{\textit{ x CPI x Ciclo}}_{\textbf{\textit{viejo}}} \textbf{\textit{ x 1,1}}$$

Igualando estas dos expresiones tendremos:

$$\textbf{\textit{1,1 x Recuento}}_{\textbf{\textit{nuevo}}} = \textbf{\textit{ Recuento}}_{\textbf{\textit{viejo}}}$$
$$\textbf{\textit{Recuento}}_{\textbf{\textit{nuevo}}} = \textbf{\textit{ 0,91 x Recuento}}_{\textbf{\textit{viejo}}}$$

Por lo tanto, para que ambas máquinas tengan el mismo rendimiento es necesario que el recuento nuevo sea un 9% menor que el recuento de la máquina original. De acuerdo con esto, habrá que eliminar ese 9% del 36% de instrucciones de carga que existen en el repertorio de instrucciones antiguo (instrucciones de transferencia), y que equivale a eliminar un **25%** de ellas:

$$\frac{0,09 \cdot 100}{0,36} = 25\%$$

- Escriba un fragmento de código ensamblador en el que una carga de R1 seguida inmediatamente de una operación en la ALU de R1 no se pueda sustituir por una simple instrucción de la forma propuesta al comienzo del enunciado, suponiendo que existe el mismo código de operación.

Ya que la nueva instrucción elimina la utilización del registro en donde se almacena el resultado de la carga, bastará con escribir una secuencia de instrucciones en la cual dicho registro sea utilizado de nuevo por alguna otra instrucción. Por ejemplo:

```
LD R1, 0(Rb)
ADD R2, R2, R1
.....................
ADD R3, R3, R1
```

Si sustituimos las dos primeras instrucciones por la nueva, anulamos el registro **R1** impidiendo la correcta ejecución de la última instrucción. En el código que queda al sustituir las dos primeras instrucciones se aprecia esto claramente:

```
ADD R2, 0(Rb)
……..……
ADD R3, R3, R1
```

Dado el siguiente fragmento de código DLX en el que inicialmente la posición de memoria 2000 contiene el valor 0, ¿cuántas referencias de datos a memoria se efectuarán e instrucciones se ejecutarían?

```
i1:   LD   R1,1500(R0)
i2:   LD   R2,2000(R0)
i3:   LD   R3,500(R2)
i4:   ADD  R4,R3,R1
i5:   SD   0(R2),R4
i6:   ADDI R2,R2,#4
i7:   SD   2000(R0),R2
i8:   SUBI R4,R2,#400
i9:   BENZ 4,i2
```

Bucle de i2 a i9.

El número total de instrucciones ejecutadas es:

1 instrucción para la preparación del bucle
100 * 8 instrucciones del cuerpo del bucle = 800 instrucciones
Total = 801 instrucciones ejecutadas

El total de referencias a datos en memoria es:

1 vez para la carga del valor de R1 en un registro
4 veces por iteración: carga de R2 y R3 (2 LD) y almacenamiento de R4 y R2 (2 SD)
Total = 1 + 100 * 4 = 401 referencias a datos en memoria

Justificación de las 100 veces que se ejecuta el bucle:

Inicialmente, en la instrucción i2, R2 se carga con el valor 0 ya que en M[2000+R0] = M[2000] = 0.
El valor de R2 se incrementa en 4 bytes en cada i6 y se almacena de nuevo en la posición 2000 (en i7).
A continuación, en i8 se comprueba si el contenido en R2 es 400 y si no lo es se salta de nuevo a i2.
Por lo tanto, de 0 a 400 con incrementos de 4 en 4 son 100 iteraciones del bucle.

Actividad 1.9 *Autor: Sebastián Dormido Cantó*

Sea un procesador con una segmentación de 4 etapas: fetch (IF), decodificación (ID), ejecución (EX) y escritura (WB). En la etapa ID se realiza el cálculo de la dirección y los cambios iniciales sobre el registro de estado. La condición de salto se examina en la etapa EX. Las frecuencias de las instrucciones de salto condicional y bifurcación son un 30% y un 7%, respectivamente, siendo efectivos el 70% de los saltos condicionales.

Escribir un diagrama de la segmentación para cada caso (bifurcación, salto condicional efectivo y salto condicional no efectivo) considerando las estrategias predecir como efectivo y predecir como no efectivo.

Tal y como se explica en el enunciado la dirección de salto se calcula en la etapa ID pero la condición de salto se examina en la etapa EX y no en la ID.
Obsérvese que para evitar cambios no deseados en el registro de estado al utilizar la estrategia PREDECIR COMO NO EFECTIVO es necesario situar una detención inmediatamente después de la etapa IF, ya que de lo contrario se ejecutaría la etapa ID de la siguiente instrucción al salto condicional modificando el registro de estado pese a que el salto pudiese ser efectivo.

De acuerdo con esto obtendremos los siguientes diagramas (**Det**: detención):

1. **Estrategia de PREDECIR COMO NO EFECTIVO**

Bifurcación							
	IF	ID	EX	WB			
		IF	IF	ID	EX	WB	

Salto condicional NO Efectivo							
	IF	ID	EX	WB			
		IF	Det	ID	EX	WB	

Salto condicional Efectivo							
	IF	ID	EX	WB			
		IF	Det	IF	ID	EX	WB

2. **Estrategia de PREDECIR COMO EFECTIVO**

Bifurcación							
	IF	ID	EX	WB			
		IF	IF	ID	EX	WB	

Salto condicional NO Efectivo							
	IF	ID	EX	WB			
		IF	IF	IF	ID	EX	WB

Salto condicional Efectivo							
	IF	ID	EX	WB			
		IF	IF	ID	EX	WB	

Calcular para ambas estrategias de salto el valor CPI asumiendo que los riesgos de control son los únicos riesgos existentes.

De acuerdo con los datos dados en el enunciado tendremos los siguientes porcentajes:

Bifurcaciones: 7%

Saltos condicionales no efectivos: 30% × 30% = 9%

Saltos condicionales efectivos: 30% × 70% = 21%

1. **Estrategia de PREDECIR COMO NO EFECTIVO**

Tipo de Salto	Costo (ciclos)	Frecuencia	Contribución
Bifurcación	1	7%	1×0.07=0.07
Salto condicional NO Efectivo	1	9%	1×0.09=0.09
Salto condicional Efectivo	2	21%	2×0.21=0.42

2. **Estrategia de PREDECIR COMO EFECTIVO**

Tipo de Salto	Costo (ciclos)	Frecuencia	Contribución
Bifurcación	1	7%	1×0.07=0.07
Salto condicional NO Efectivo	2	9%	2×0.09=0.18
Salto condicional Efectivo	1	21%	1×0.21=0.21

$CPI_{saltos} = 1 + 0.07 + 0.18 + 0.21 = 1.46$

Como se puede observar en los resultados obtenidos, la estrategia PREDECIR COMO EFECTIVO es superior a la PREDECIR COMO NO EFECTIVO.

En los fragmentos de código situados a continuación:

FRAGMENTO 1	FRAGMENTO 2
`i1: DIV R1,R2,R3`	`i1: LD F2,0(R1)`
`i2: ADD R4,R1,R5`	`i2: MULT F4,F2,F0`
`i3: ADD R5,R6,R7`	`i3: LD F6,0(R2)`
`i4: ADD R1,R8,R9`	`i4: ADDD F6,F4,F6`
	`i5: SD 0(R2),F6`
	`i6: ADDI R1,R1,#8`
	`i7: ADDI R2,R2,#8`
	`i8: SGT R3,R1,#800`
	`i9: BEQZ R3,i1`

- Señale las dependencias de datos y de memoria existentes en ambos fragmentos.
- Analice y explique lo que sucede con el registro R1 en sucesivas iteraciones de la segunda secuencia de código.

Dependencias

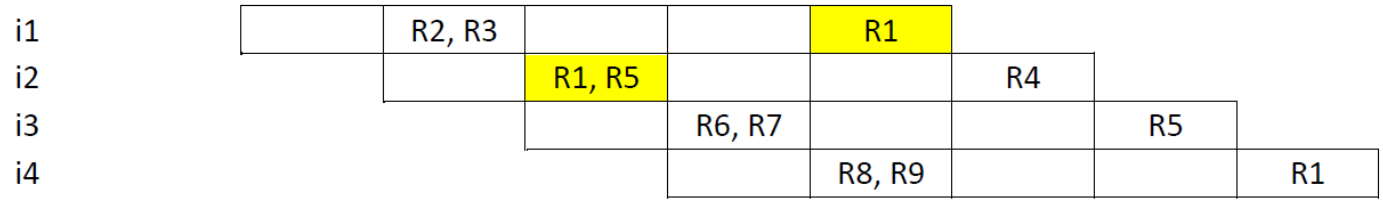
Se va a considerar únicamente el caso de riesgos tipo RAW.

Los riesgos de dependencias de datos se denominan:

Riesgos por dependencia de datos en registros en el caso de que se trate de instrucciones aritmético-lógicas.

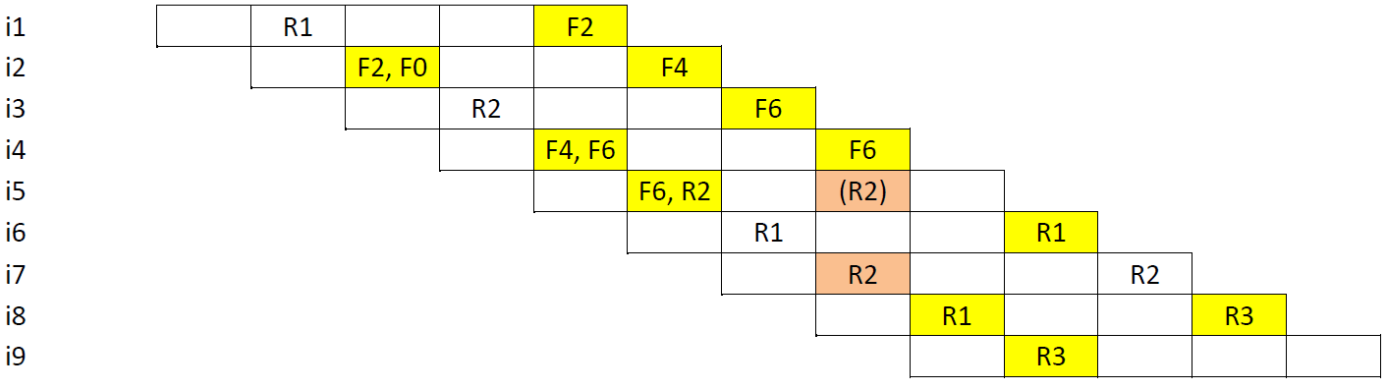
Riesgos por dependencia de datos en memoria cuando suceden con instrucciones de carga y almacenamiento.

Por lo tanto, para el fragmento 1 tenemos:



Riesgo por dependencia de datos en registros, entre i1 e i2 a través de R1.

Para el fragmento 2 tenemos:



Riesgo por dependencias de datos en registros, entre i1 e i2 a través de F2.
Riesgo por dependencias de datos en registros, entre i2 e i4 a través de F4.
Riesgo por dependencias de datos en registros, entre i3 e i4 a través de F6.
Riesgo por dependencias de datos en registros, entre i3 e i5 a través de F6.
Riesgo por dependencias de datos en registros, entre i4 e i5 a través de F6.
Riesgo por dependencias de datos en registros, entre i6 e i8 a través de R1.
Riesgo por dependencias de datos en registros, entre i8 e i9 a través de R3.
Riesgo por dependencias de datos en memoria, entre i5 e i7 a través de R2.

Análisis

En realidad no sé si realmente entiendo bien la pregunta, es decir, no sé si lo que se pide es la evolución del R1 o por el contrario se pide comentar que ocurre con ese registro en temas de dependencia de datos.

Para el primer caso, tenemos que:
En i1, R1 es accedido por una instrucción de carga.
En i6, el contenido de R1 es aumentado en 8 unidades, mediante una instrucción aritmética.
En i8 se compara R1 con el literal 800, mediante una instrucción aritmética.

Para el segundo caso:
En i1 e i6, no existe ningún tipo de dependencia.
En i8 se produce una dependencia RAW a través de R1, debido a que aún la i6 no ha escrito R1.

Actividad 1.11 *Autor: Sebastián Dormido Cantó*

La siguiente secuencia de instrucciones se ejecuta sobre un procesador sin segmentación y empleando una instrucción de salto condicional:

```
i1: LD     R1, X(R7)    // R1  ← M[R7+X]
i2: ADDI   R1, R1, #1   // R1  ← R1+1
i3: BEQZ   Rtest, i6    // Si Rtest = 0 entonces salta a la instrucción i6
i4: ADD    R2, R1, R2   // R2  ← R1+R2
i5: SUB    R2, R2, R3   // R2  ← R2-R3
i6: SD     0(R8), R5    // M[R8+0] ← R5
```

Basándose en el fragmento anterior de código escribir la secuencia de instrucciones en un procesador segmentado similar a ASG usando la técnica de salto retardado en las siguientes precondiciones:

- Inserción de una instrucción de no operación (NOP) con el objeto de rellenar el hueco de la detención.
- Optimización de la secuencia de instrucciones suponiendo que **`Rtest`** sea R4, eliminando la instrucción NOP cuando sea posible.
- Optimización de la secuencia de instrucciones si **`Rtest`** es R1 eliminando la instrucción NOP si es posible.

Apartado a)

Como la instrucción de salto conoce la dirección de destino al final de la etapa MEM hay un retardo de tres huecos por lo que hay que colocar tres instrucciones NOP (Figura 1.33 del libro de teoría).

```
i1: LD R1, X(R7)
i2: ADDI R1, R1, #1
i3: BEQZ Rtest, i6
i*: NOP
i*: NOP
i*: NOP
i4: ADD R2, R1, R2
i5: SUB R2, R2, R3
i6: SD 0(R8), R5
```

Con independencia de que el salto sea o no efectivo, las instrucciones que se han colocado en el hueco de retardo siempre van a comenzar a ejecutarse e introducirse en la segmentación hasta que se conozca el destino del salto. Por ello, es importante colocar instrucciones que sean independientes del resultado del salto, o que se ejecuten con una alta probabilidad (por ejemplo, en un bucle se ejecutarán las n veces que itere el bucle pero habrá una vez que se ejecute de forma errónea, esto, al salir del bucle) de forma que el tener que deshacer los cambios (si los hubiese) que ha introducido la ejecución especulativa compense el tener que deshacerlos.
Si el bucle es efectivo la secuencia dinámica de ejecución es la siguiente:

```
i1: LD R1, X(R7)
i2: ADDI R1, R1, #1
i3: BEQZ Rtest, i6
i*: NOP
i*: NOP
i*: NOP
i6: SD 0(R8), R5
```

La segmentación quedaría:

```
i1: LD R1, X(R7)
i2: ADDI R1, R1, #1
i3: BEQZ Rtest, i6
i*: NOP
i*: NOP
i*: NOP
i6: SD 0(R8), R5
```

IF	ID	EX	MEM	WB						
	IF	ID	---	EX	MEM	WB				
		IF	---	ID	EX	MEM	WB			
			IF	ID	EX	MEM	WB			
				IF	ID	EX	MEM	WB		
					IF	ID	EX	MEM	WB	
						IF	ID	EX	MEM	WB

Un ejemplo claro de aprovechamiento del hueco de retardo es imaginar que en lugar de saltar a i6 se saltase a i1 durante 1000 veces. Esas 3 NOPS se podrían reemplazar por instrucciones del cuerpo del bucle pero habría una vez en que se ejecutarían de forma especulativa, es decir, al salir del bucle.

Apartado b)

Una posible solución muy optimizada es la siguiente:

```
i3: BEQZ Rtest, ix
i1: LD R1, X(R7)
i6: SD 0(R8), R5
i2: ADDI R1, R1, #1
i4: ADD R2, R1, R2
i5: SUB R2, R2, R3
ix: NOP
```

Efectivo:

```
i3: BEQZ R4, ix
i1: LD R1, X(R7)
i6: SD 0(R8), R5
i2: ADDI R1, R1, #1
ix: NOP
```

IF	ID	EX	MEM	WB				
	IF	**ID**	**EX**	**MEM**	**WB**			
		IF	**ID**	**EX**	**MEM**	**WB**		
			IF	**ID**	**EX**	**MEM**	**WB**	
				IF	ID	EX	MEM	WB

No efectivo:

```
i3: BEQZ R4, ix
i1: LD R1, X(R7)
i6: SD 0(R8), R5
i2: ADDI R1, R1, #1
i4: ADD R2, R1, R2
i5: SUB R2, R2, R3
i7: NOP
```

IF	ID	EX	MEM	WB						
	IF	**ID**	**EX**	**MEM**	**WB**					
		IF	**ID**	**EX**	**MEM**	**WB**				
			IF	**ID**	**EX**	**MEM**	**WB**			
				IF	ID	EX	MEM	WB		
					IF	ID	EX	MEM	WB	
						IF	ID	EX	MEM	WB

Apartado c)

Debido a que R1 depende de la ejecución de las instrucciones antes del salto, no es posible realizar la optimización reorganizando el código. El resultado sería el del apartado a)

Dado el código que aparece a continuación y utilizando la segmentación ASG sin ningún tipo de adelantamiento entre etapas:

```
i1:   ADD   R3,R1,R2
i2:   LD    R1,0(R4)
i3:   SUB   R5,R3,R4
i4:   ADD   R6,R1,R2
i5:   SUB   R1,R3,R6
i6:   SD    4(R4),R1
i7:   LD    R2,4(R4)
i8:   SUB   R3,R5,R6
```

- Indicar los riesgos por dependencias de datos que se producen al ejecutar el código. Suponer que las escrituras se producen en la primera mitad de la etapa WB y que las lecturas en ID se producen en la segunda mitad de la etapa.
- Compruebe si la siguiente afirmación es cierta: La instrucción `i7` no carga el valor previamente almacenado por la instrucción `i6` en la posición de memoria `M[R4+4]`. Indicar la razón de ello tanto en caso afirmativo como negativo.
- Reorganizar el código sin cambiar su efecto añadiendo el mínimo número posible de instrucciones NOP.
- Repetir el primer apartado sobre el código original considerando que se añade a la segmentación de ASG adelantamiento entre las etapas. Señalar el dato y la etapa de cada instrucción que hace uso del adelantamiento y a qué instrucción y etapa se adelanta.

Primer punto, veo que el problema trata principalmente sobre dependencias de datos y adelantamiento entre etapas. Además, el tercer punto parece indicar que la implementación ASG del problema carece de interbloqueo entre etapas. En caso contrario, no serían necesarias las instrucciones NOP.
Primer punto. Riesgos por dependencias de datos que se producen al ejecutar el código, sin ningún tipo de adelantamiento. WB primera mitad, ID segunda.

- Riesgo RAW por R3 entre i1 e i3, de 1 ciclo.
- Riesgo RAW por R1 entre i2 e i4, de 1 ciclo.
- Riesgo RAW por R6 entre i4 e i5, de 2 ciclos.
- Riesgo RAW por R1 entre i5 e i6, de 2 ciclos.

Segundo punto. La instrucción `i7` no carga el valor previamente almacenado por la instrucción i6

- No. La instrucción `i7` **sí** carga el valor previamente almacenado por `i6`. Entre una y otra no cambia ni R4 ni el contenido de la posición de memoria `M[4+R4].` La etapa ID de `i7` ocurre al mismo tiempo que la etapa WB de `i4`. Si `i4` fuese "`ADD R4,R1,R2`", y como WB ocurre antes que ID, esta instrucción podría alterar el registro R4 entre las instrucciones `i6` e `i7`:

```
i4: ADD   R4,R1,R2
i5: SUB   R1,R3,R6
i6: SD    4(R4),R1
i7: LD    R2,4(R4)
```

IF	ID	EX	MEM	**WB**			
	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	
			IF	**ID**	EX	MEM	WB
				Nuevo valor de R4			

Tercer punto. Reorganizar con mínimo número de NOPs.

```
i1:   ADD   R3,R1,R2
i2:   LD    R1,0(R4)
      NOP               ;Evita los dos primeros riesgos RAW, i1-i3 e i2-i4
i3:   SUB   R5,R3,R4
i4:   ADD   R6,R1,R2
      NOP
      NOP               ;Evitan el riego RAW i4-i5
i5:   SUB   R1,R3,R6
      NOP
i8:   SUB   R3,R5,R6    ;Junto con el NOP previo evitan RAW i5-i6
i6:   SD    4(R4),R1
i7:   LD    R2,4(R4)
```

Cuarto punto. Considerar ahora que hay adelantamiento entre etapas.

- Adelantamiento de i1 a i3. Registro R3, de la etapa EX1 a EX3.
- Adelantamiento de i2 a i4. Registro R1, de la etapa MEM2 a EX4.
- Adelantamiento de i4 a i5. Registro R6, de la etapa EX4 a EX5.
- Adelantamiento de i5 a i6. Registro R1, de la etapa EX5 a MEM6.
- Ningún riesgo por dependencias de datos.

Actividad 1.13 Autor: Jesús Moreno Monteverde

Considere la segmentación ASG en la que la lectura y decodificación de los registros se realiza en la etapa ID. Dado el siguiente fragmento de código:

```
i1:   ADD   R5,R0,R0   //    R5 ← R0+R0
i2:   LD    R1,0(R2)   //    R1 ← M[R2+0]
i3:   ADD   R6,R1,R5   //    R6 ← R1+R5
i4:   ADD   R5,R1,R0   //    R5 ← R1+R0
i5:   SD    0(R2),R6   //    M[R2+0] ← R6
i6:   ADDI  R2,R2,#4   //    R2 ← R2+4
i7:   SUB   R4,R3,R2   //    R4 ← R3-R2
i8:   BNEZ  R4,i2      //    Si R4 !=0 ir a i2
```

- Dibuje una tabla en la que se muestre la evolución temporal de la secuencia de instrucciones en la segmentación. No considere ningún tipo de adelantamiento salvo que las etapas ID y WB pueden acceder en el mismo ciclo de reloj al banco de registros (la etapa WB accede en la primera mitad y la etapa ID en la segunda mitad). Los saltos condicionales se manipulan con la técnica de predecir-no-efectivo pero el contador de programa se actualiza en la etapa MEM por parte de la instrucción de salto.
- Dibuje una tabla similar a la del primer apartado pero considerando adelantamiento entre etapas así como que las etapas ID y WB pueden acceder en el mismo ciclo de reloj al banco de registros. Los saltos condicionales son manipulados mediante la técnica de predecir como no-efectivo considerando que al final de la etapa ID ya se conoce el destino del salto.
- Asumiendo que el valor inicial del registro R3 es R2+24, calcule para cada uno de los dos supuestos anteriores cuántos ciclos de reloj tarda en ejecutarse el fragmento de código de ASG.

Apartado 1.

Asumo que la segmentación tiene interbloqueo entre etapas. Avanzo las siguientes instrucciones, dependiendo de si el salto es o no efectivo.

Salto no efectivo

CICLO	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
i1 ADD R5,R0,R0	IF	ID 0	EX	ME -	WB 5																			
i2 LD R1,0(R2)		IF	ID 2	EX	ME	WB 1																		
i3 ADD R6,R1,R5			IF	WAR 1,5	WAR 1	ID 1,5	EX	ME -	WB 6															
i4 ADD R5,R1,R0						IF	ID 1,0	EX	ME -	WB 5														
i5 SD 0(R2),R6							IF	WAR 6	ID 2,6	EX	ME	WB -												
i6 ADDI R2,R2,#4									IF	ID 2	EX	ME -	WB 2											
i7 SUB R4,R3,R2										IF	WAR 2	WAR 2	ID 3,2	EX	ME -	WB 4								
i8 BNEZ R4,i2													IF	WAR 4	WAR 4	ID 4	EX nez	ME	WB					
i9																IF	ID	EX	ME	WB				
i10																	IF	ID	EX	ME	WB			
i11																		IF	ID	EX	ME	WB		
i12																			IF	ID	EX	ME	WB	

Salto efectivo

CICLO	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
i1 ADD R5,R0,R0	IF	ID 0	EX	ME -	WB 5																			
i2 LD R1,0(R2)		IF	ID 2	EX	ME	WB 1																		
i3 ADD R6,R1,R5			IF	WAR 1,5	WAR 1	ID 1,5	EX	ME -	WB 6															
i4 ADD R5,R1,R0						IF	ID 1,0	EX	ME -	WB 5														
i5 SD 0(R2),R6							IF	WAR 6	ID 2,6	EX	ME	WB -												
i6 ADDI R2,R2,#4									IF	ID 2	EX	ME -	WB 2											
i7 SUB R4,R3,R2										IF	WAR 2	WAR 2	ID 3,2	EX	ME -	WB 4								
i8 BNEZ R4,i2													IF	WAR 4	WAR 4	ID 4	EX nez	ME pc	WB					
i9																IF	ID	EX						
i10																	IF	ID						
i11																		IF						
LD R1,0(R2)																			IF	ID 2	EX	ME	WB 1	

Apartado 2.

Con adelantamiento entre etapas. En las etapas EX y MEM se muestra el dato que se adelanta. Sólo se muestra el caso de salto efectivo.

CICLO	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
i1 ADD R5,R0,R0	IF	ID 0	EX 5	ME -	WB 5																			
i2 LD R1,0(R2)		IF	ID 2	EX -	ME 1	WB 1																		
i3 ADD R6,R1,R5			IF	ID 1,5	WAR 1	EX 6	ME -	WB 6																
i4 ADD R5,R1,R0				IF		ID 1,0	EX 5	ME -	WB 5															
i5 SD 0(R2),R6						IF	ID 2,6	EX	ME	WB -														
i6 ADDI R2,R2,#4							IF	ID 2	EX 2	ME -	WB 2													
i7 SUB R4,R3,R2								IF	ID 3,2	EX 4	ME -	WB 4												
i8 BNEZ R4,i2									IF	ID 4	EX nez	ME	WB											
i9										IF	ID	EX												
i10											IF	ID												
i11												IF												
LD R1,0(R2)													IF	ID 2	EX	ME 1	WB 1							

Apartado 3.

Si R3 = R2-24, significa que inicialmente, R3 – R2 = 24. Precisamente, esta es la comparación que hace la instrucción i7 e i8. Cuando R3 – R2=0, deja de ejecutarse el bucle. Por cada pasada, R2 se incrementa en 4, por lo que el bucle se repite 6 veces. En cinco de las veces, debemos de esperar tres ciclos extra para determinar la posición del salto en PC. En la última iteración, al terminar la instrucción i8 se termina el algoritmo.

En el primer caso, por cada iteración se cumplen 17 ciclos, más la primera instrucción, y la última iteración son 14. Esto hace 1 + 17*5 + 14 = 100 ciclos

En el segundo caso, por cada iteración se cumplen 11 ciclos y la última 8.
Así nos queda 1 + 11*5 + 8 = 64 ciclos.

Actividad 1.14 *Autor: Jesús Moreno Monteverde*

Mostrar la evolución de los Registros en coma flotante (FF) y las estaciones de Reserva (RS) para todos los ciclos que sean necesarios en la ejecución del siguiente fragmento de código utilizando el algoritmo de Tomasulo.

```
i1: ADDD   F2,F0,F6
i2: MULTD  F4,F0,F2
i3: ADDD   F2,F2,F6
i4: MULTD  F6,F2,F4
i5: ADDD   F4,F4,F6
i6: ADDD   F6,F2,F4
```

Considere las siguientes hipótesis de partida:

- Para reducir el número de ciclos máquina se permite que la FLOS distribuya hasta dos instrucciones en cada ciclo según el orden del programa.
- Una instrucción puede comenzar su ejecución en el mismo ciclo en que se distribuye a una estación de reserva.
- La operación suma tiene una latencia de dos ciclos y la de multiplicación de tres ciclos.
- Se permite que una instrucción reenvíe su resultado a instrucciones dependientes durante su último ciclo de ejecución. De esta forma una instrucción a la espera de un resultado puede comenzar su ejecución n el siguiente ciclo si se detecta una coincidencia.
- Los valores de etiqueta `01`, `02` y `03` se utilizan para identificar las tres estaciones de reserva de la unidad funcional de multiplicación/división. Estos valores de etiqueta son los ID de las estaciones de reserva.
- Inicialmente, el valor de los registros es `F0=4.0, F2=2.5, F4=10.0 y F6=3.5`.

Actividad 1.14

Situación Inicial.

	bitOc	etiqueta	dato
F0			4,0
F2			2,5
F4			10,0
F6			3,5

Ciclo 1.

1. Distribución de FLOS de i1:**ADDD F2,F0,F6** a RS01
2. Carga de los operandos F0 y F6 en RS01. Ninguno bloqueado.
3. Se activa F2 como bloqueado por RS01.
4. Se inicia la ejecución de RS01(i1). Ciclo 1/2 de la suma.
5. Distribución de FLOS de i2:**MULTD F4,F0,F2** a RS04
6. Carga del operando F0 en RS04. F2 bloqueado por RS01.
7. Se activa F4 como bloqueado por RS04.
8. RS04(i2) no puede ejecutarse, al tener un operando bloqueado.

	bitOc	etiqueta	dato
F0			4,0
F2	si	01	2,5
F4	si	04	10,0
F6			3,5

		Etiq-1	Oper-1	Etiq-2	Oper-2
i1:	01	00	4,0	00	3,5
	02				
	03				

		Etiq-1	Oper-1	Etiq-2	Oper-2
i2:	04	00	4,0	01	¿?
	05				

Ciclo 2.

1. Distribución de FLOS de i3:**ADDD F2,F2,F6** a RS02
2. Carga del operando F6 en RS02. F2 bloqueado por RS01
3. Se activa F2 como bloqueado por RS02.
4. RS02(i3) no puede ejecutarse, al tener un operando bloqueado.
5. Distribución de FLOS de i4:**MULTD F6,F2,F4** a RS05
6. F2 bloqueado por RS02. F4 bloqueado por RS04.
7. Se activa F6 como bloqueado por RS05.
8. RS05(i4) no puede ejecutarse, al tener los dos operandos bloqueados.
9. RS01(i1) termina de ejecutarse. Ciclo 2/2 de la suma.
10. CDDB publica el resultado de RS01: 4 + 3,5 = 7,5.

	bitOc	etiqueta	dato
F0			4,0
F2	si	02	2,5
F4	si	04	10,0
F6	si	05	3,5

		Etiq-1	Oper-1	Etiq-2	Oper-2
i1:	01	00	4,0	00	3,5
i3:	02	01	¿?	00	3,5
	03				

		Etiq-1	Oper-1	Etiq-2	Oper-2
i2:	04	00	4,0	01	¿?
i4:	05	02	¿?	04	¿?

Ciclo 3.

1. Se actualiza el resultado de RS01(i1:7,5) en RS02(Oper-1) y RS04(Oper-2)
2. Se vacía RS01.
3. Se inicia la ejecución de RS04(i2), ya que sus operandos ya no están bloqueados. Ciclo 1/3 de la multiplicación.
4. Se inicia la ejecución de RS02(i3), ya que sus operandos ya no están bloqueados. Ciclo 1/2 de la suma.
5. Distribución de FLOS de i5:**ADDD F4,F4,F6** a RS03
6. F4 bloqueado por RS04. F6 bloqueado por RS05.
7. Se activa F4 como bloqueado por RS03.
8. RS03(i5) no puede ejecutarse, al tener los dos operandos bloqueados.
9. Distribución de FLOS de i6:**MULTD F6,F2,F4** a RS01
10. F2 bloqueado por RS02. F4 bloqueado por RS03.
11. Se activa F6 como bloqueado por RS01.
12. RS01(i6) no puede ejecutarse, al tener los dos operandos bloqueados.

	bitOc	etiqueta	dato
F0			4,0
F2	si	02	2,5
F4	si	03	10,0
F6	si	01	3,5

		Etiq-1	Oper-1	Etiq-2	Oper-2
i6:	01	02	¿?	03	¿?
i3:	02	00	7,5	00	3,5
i5:	03	04	¿?	05	¿?

		Etiq-1	Oper-1	Etiq-2	Oper-2
i2:	04	00	4,0	00	7,5
i4:	05	02	¿?	04	¿?

Ciclo 4.

1. Continúa la ejecución de RS04(i2). Ciclo 2/3 de la multiplicación.
2. Termina la ejecución de RS02(i3). Ciclo 2/2 de la suma.
3. CDDB publica el resultado de RS02: 7,5 + 3,5 = 11,0.

Las tablas no varían.

Ciclo 5.

1. Se actualiza el resultado de RS02(i3:11,0) en RS01(Oper-1), RS05(Oper-1) y en F2
2. Se vacía RS02 y se desbloquea F2.
3. Termina la ejecución de RS04(i2). Ciclo 3/3 de la multiplicación.
4. CDDB publica el resultado de RS04: 4,0 · 7,5 = 30,0

	bitOc	etiqueta	dato
F0			4,0
F2			11,0
F4	si	03	10,0
F6	si	01	3,5

		Etiq-1	Oper-1	Etiq-2	Oper-2
i6:	01	00	11,0	03	¿?
	02				
i5:	03	04	¿?	05	¿?

		Etiq-1	Oper-1	Etiq-2	Oper-2
i2:	04	00	4,0	00	7,5
i4:	05	00	11,0	04	¿?

Ciclo 6.

1. Se actualiza el resultado de RS04(i2:30,0) en RS05(Oper-2), RS03(Oper-1).
2. Se vacía RS04.
3. Se inicia la ejecución de RS05(i4), ya que sus operandos ya no están bloqueados. Ciclo 1/3 de la multiplicación.

	bitOc	etiqueta	dato
F0			4,0
F2			11,0
F4	si	03	10,0
F6	si	01	3,5

		Etiq-1	Oper-1	Etiq-2	Oper-2
i6:	01	00	11,0	03	¿?
	02				
i5:	03	00	30,0	05	¿?

		Etiq-1	Oper-1	Etiq-2	Oper-2
	04				
i4:	05	00	11,0	00	30,0

Ciclo 7.

1. Continúa la ejecución de RS05(i4). Ciclo 2/3 de la multiplicación.

Las tablas no varían.

Ciclo 8.

1. Termina la ejecución de RS05(i4). Ciclo 3/3 de la multiplicación.
2. CDDB publica el resultado de RS05: 11,0 · 30,0 = 330,0

Las tablas no varían.

Ciclo 9.

1. Se actualiza el resultado de RS05(i4:330,0) en RS03(Oper-2).
2. Se vacía RS05.
3. Se inicia la ejecución de RS03(i5), ya que sus operandos ya no están bloqueados. Ciclo 1/2 de la suma.

	bitOc	etiqueta	dato
F0			4,0
F2			11,0
F4	si	03	10,0
F6	si	01	3,5

		Etiq-1	Oper-1	Etiq-2	Oper-2
i6:	01	00	11,0	03	¿?
	02				
i5:	03	00	30,0	00	330,0

	Etiq-1	Oper-1	Etiq-2	Oper-2
04				
05				

Ciclo 10.

1. Termina la ejecución de RS03(i5). Ciclo 2/2 de la suma.
2. CDDB publica el resultado de RS03: 30,0 + 330,0 = 360,0

Las tablas no varían.

Ciclo 11.

1. Se actualiza el resultado de RS03(i5:360,0) en RS01(Oper-2) y F4.
2. Se vacía RS03 y desbloquea F4.
3. Se inicia la ejecución de RS01(i6), ya que sus operandos ya no están bloqueados. Ciclo 1/2 de la suma.

	bitOc	etiqueta	dato
F0			4,0
F2			11,0
F4			360,0
F6	si	01	3,5

		Etiq-1	Oper-1	Etiq-2	Oper-2
i6:	01	00	11,0	00	360,0
	02				
	03				

	Etiq-1	Oper-1	Etiq-2	Oper-2
04				
05				

Ciclo 12.

1. Termina la ejecución de RS01(i6). Ciclo 2/2 de la suma.
2. CDDB publica el resultado de RS01: 11,0 + 360,0 = 371,0

Las tablas no varían.

Ciclo 11.

1. Se actualiza el resultado de RS01(i6:371,0) en F6.
2. Se vacía RS01 y desbloquea F6.

	bitOc	etiqueta	dato
F0			4,0
F2			11,0
F4			360,0
F6			371,0

3.

	Etiq-1	Oper-1	Etiq-2	Oper-2
01				
02				
03				

	Etiq-1	Oper-1	Etiq-2	Oper-2
04				
05				

Actividad 1.15 *Autor: Juan Diego Bueno Prieto*

Mostrar la evolución de los Registros en coma flotante (FF) y las estaciones de Reserva (RS) para todos los ciclos que sean necesarios en la ejecución del siguiente fragmento de código utilizando el algoritmo de Tomasulo.

```
i1: MULTD   F2,F2,F6
i2: MULTD   F4,F2,F6
i3: ADDD    F2,F4,F6
i4: ADDD    F6,F2,F6
i5: ADDD    F4,F4,F6
```

Considere las siguientes hipótesis de partida:

- Para reducir el número de ciclos máquina se permite que la FLOS distribuya hasta dos instrucciones en cada ciclo según el orden del programa.
- Una instrucción puede comenzar su ejecución en el mismo ciclo en que se distribuye a una estación de reserva.
- La operación suma tiene una latencia de dos ciclos y la de multiplicación de cuatro ciclos.
- Se permite que una instrucción reenvíe su resultado a instrucciones dependientes durante su último ciclo de ejecución. De esta forma una instrucción a la espera de un resultado puede comenzar su ejecución n el siguiente ciclo si se detecta una coincidencia.
- Los valores de etiqueta `01`, `02` y `03` se utilizan para identificar las tres estaciones de reserva de la unidad funcional de suma, mientras que `04` y `05` se utilizan para identificar las dos estaciones de reserva de la unidad funcional de multiplicación/división. Estos valores de etiqueta son los ID de las estaciones de reserva.
- Inicialmente, el valor de los registros es `F0=2.0, F2=4.5, F4=8.0 y F6=3.0`.

Actividad 1.15

Secuencia de instrucciones
i1: MULTD F2, F2, F6
i2: MULTD F4, F2, F6
i3: ADDD F2, F4, F6
i4: ADDD F6, F2, F6
i5: ADDD F4, F4, F6

Ciclo 1: Distribución de i1 e i2 (en orden)

FR

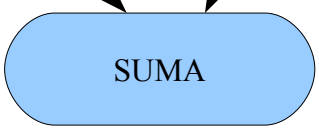
	bitOc	etiqueta	dato
F0			2.0
F2	Sí	04	4.5
F4	Sí	05	8.0
F6			3.0

RS

ID	etiqueta_1	operando_1	etiqueta_2	operando_2

RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i1	04	00	4.5	00	3.0
i2	05	04	---	00	3.0



Ciclo 2: Distribución de i3 e i4 (en orden)

FR

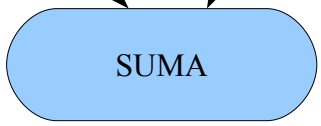
	bitOc	etiqueta	dato
F0			2.0
F2	Sí	01	4.5
F4	Sí	05	8.0
F6	Sí	02	3.0

RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i3	01	05	---	00	3.0
i4	02	01	---	00	3.0
	03				

RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i1	04	00	4.5	00	3.0
i2	05	04	---	00	3.0



Ciclos 3 y 4: Distribución de i5

FR

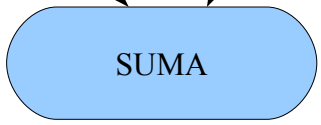
	bitOc	etiqueta	dato
F0			2.0
F2	Sí	01	4.5
F4	Sí	03	8.0
F6	Sí	02	3.0

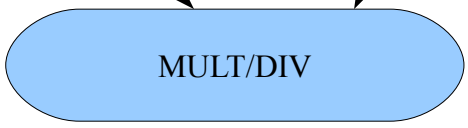
RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i3	01	05	---	00	3.0
i4	02	01	---	00	3.0
i5	03	05	---	02	---

RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i1	04	00	4.5	00	3.0
i2	05	04	---	00	3.0





Ciclos 5, 6, 7 y 8:

FR

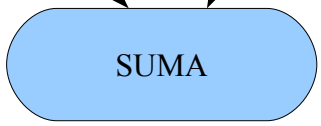
	bitOc	etiqueta	dato
F0			2.0
F2	Sí	01	4.5
F4	Sí	03	8.0
F6	Sí	02	3.0

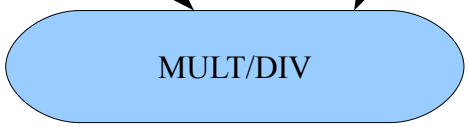
RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i3	01	05	---	00	3.0
i4	02	01	---	00	3.0
i5	03	05	---	02	---

RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i2	05	00	13.5	00	3.0





Ciclos 9 y 10:

FR

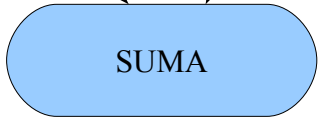
	bitOc	etiqueta	dato
F0			2.0
F2	Sí	01	4.5
F4	Sí	03	8.0
F6	Sí	02	3.0

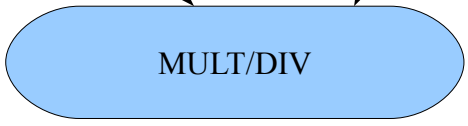
RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i3	01	00	40,5	00	3.0
i4	02	01	---	00	3.0
i5	03	00	40,5	02	---

RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2





Ciclos 11 y 12:

FR

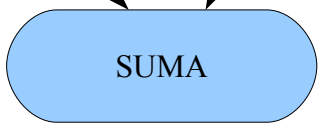
	bitOc	etiqueta	dato
F0			2.0
F2			43.5
F4	Sí	03	8.0
F6	Sí	02	3.0

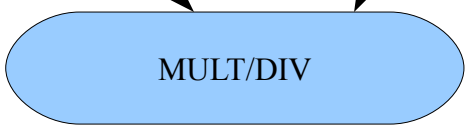
RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i4	02	00	43,5	00	3.0
i5	03	00	40,5	02	---

RS

ID	etiqueta_1	operando_1	etiqueta_2	operando_2





Ciclos 13 y 14:

FR

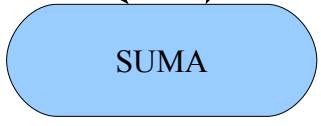
	bitOc	etiqueta	dato
F0			2.0
F2			43.5
F4	Sí	03	8.0
F6			46.5

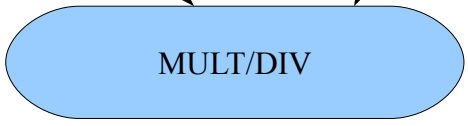
RS

	ID	etiqueta_1	operando_1	etiqueta_2	operando_2
i5	03	00	40,5	00	46,5

RS

ID	etiqueta_1	operando_1	etiqueta_2	operando_2





Ciclo 15:

FR

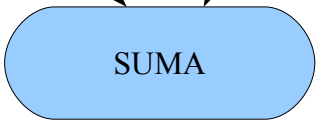
	bitOc	etiqueta	dato
F0			2.0
F2			43.5
F4			87.0
F6			46.5

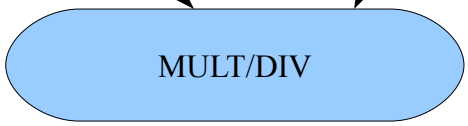
RS

ID	etiqueta_1	operando_1	etiqueta_2	operando_2

RS

ID	etiqueta_1	operando_1	etiqueta_2	operando_2





Actividad 2.1 *Autor: José Sánchez Moreno*

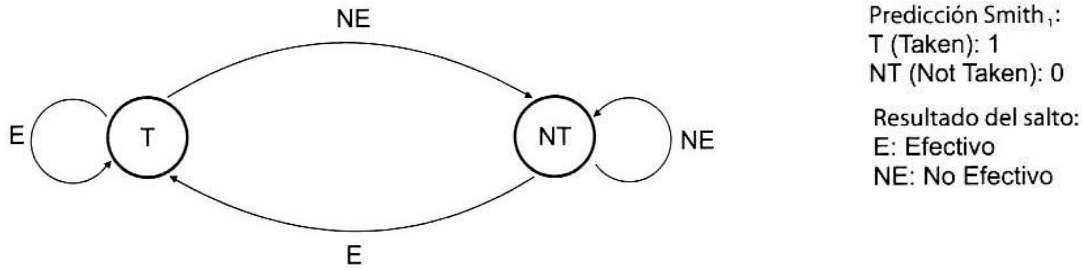
Suponga que un compilador es capaz de generar código objeto para un procesador superescalar, carente de una red hardware de alineación, de forma que las instrucciones destino de los saltos estén alineadas para que siempre sean las primeras en los grupos de lectura. La alineación del grupo de lectura permite así maximizar el número de instrucciones captadas de la I-caché dado que no hay huecos en el grupo de lectura. Por ejemplo en la figura 2.11, si la instrucción destino fuese la *i+43*, esto implicaría que el grupo de lectura solo aprovecharía una de las cuatro instrucciones, por lo que quedarían 3 huecos en el grupo de lectura. Analice el impacto que tiene el tamaño del grupo de lectura en el rendimiento de la etapa de fetch como consecuencia de la existencia de saltos efectivos. ¿cuán importante es el tamaño del grupo de lectura con respecto al lineamiento de las instrucciones que son destino de saltos?

Es actividad es más de reflexión que otra cosa. Adjunto la solución que he preparado para ella.
Si el tamaño del grupo de lectura se aumenta para extraer un mayor número de instrucciones, la probabilidad de extraer una instrucción de salto efectivo aumenta. Esto implica un mayor desaprovechamiento de la etapa de lectura de instrucciones al captar instrucciones que luego habrá que expulsar del cauce al apuntar el PC al destino del salto efectivo.
Por otra parte, a menor tamaño del grupo de lectura, un alineamiento correcto de la instrucción destino de un salto adquiere mayor relevancia dado que el porcentaje de instrucciones desaprovechadas es elevado en cualquier caso. Por ejemplo, en la Figura 2.11, la pérdida que se ocasiona en el grupo de lectura es del 75% si el destino del salto es la instrucción *i+43*; si el destino es la *i+41*, la pérdida es del 25%, que sigue siendo una pérdida elevada. Pero, además, si el grupo de lectura es muy grande, las pérdidas pueden ser porcentualmente pequeñas pero, también, muy elevadas si el destino se encuentra al final de un grupo de lectura debido a que el desaprovechamiento del grupo de lectura es casi total.
Lo importante es que el tamaño del grupo de lectura siempre sea el suficiente para asegurar el abastecimiento de instrucciones al cauce y que no se introduzcan burbujas en la segmentación. Por ello, el compilador optimizador debería estar diseñado para alinear las instrucciones destino en los grupos de lectura cuando el ancho de banda de lectura pudiese situarse por debajo de un cierto nivel de peligro.

Considere el siguiente fragmento de pseudocódigo

```
0x000000 x:=(1,2,5,8,9,15,22,25,26,30);
0x000100 Desde i:=1 hasta 10 hacer {
0x001000    Si (x es impar)              // Salto_1
0x001100       a:=a+1;                   // Salto_1 es efectivo
0x010000    Si (x es múltiplo de 5)      // Salto_2
0x010100       a:=a-1;                   // Salto_2 es efectivo
0x011000 }
```

Utilizando un predictor de un nivel basado en una PHT de dos entradas que almacena la predicción para cada salto realizada mediante un contador de Smith de 1 bit como el de la Figura 2.16.a, dibuje una tabla en la que se muestre el resultado real de los dos saltos y las predicciones realizadas para la secuencia de valores de x. El valor inicial de las dos entradas de la PHT es 0. ¿Cuál es la tasa de predicción del predictor para cada salto? ¿Y la tasa de predicción global?



La siguiente tabla muestra la evolución de los dos saltos y las predicciones realizadas por el algoritmo de Smith de 1 bit. Cada fila de la tabla indica para cada valor de x el resultado real de los dos saltos, el valor del contador de saturación de cada salto y la predicción. Es muy importante comprender que las predicciones de una fila son para la ejecución de los saltos con el valor de x de esa fila, no para el valor de x siguiente. Por ejemplo, en la primera fila, la predicción de lo que ocurrirá con x=1 para el salto 1 es NT pero la realidad es que el salto es efectivo (T); la predicción para el salto 2 es que será no efectivo (NT) y el resultado real es que el salto no se produce, hay un acierto.

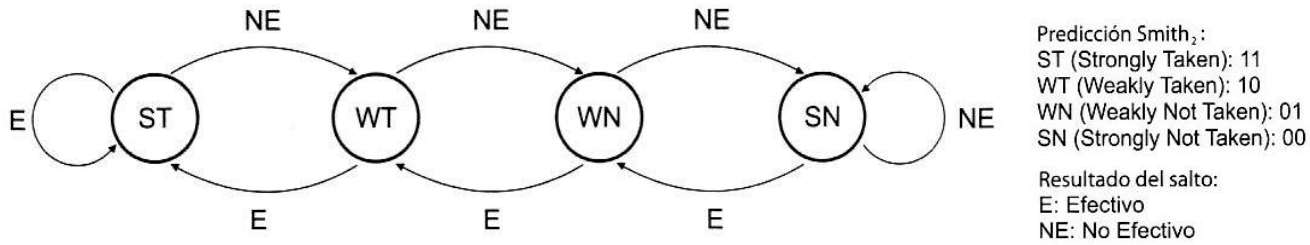
Valor	Resultado salto 1	Entrada PHT salto 1	Predicción salto 1	Resultado salto 2	Entrada PHT salto 2	Predicción salto 2
1	E	0	NE	**NE**	**0**	**NE**
2	NE	1	E	**NE**	**0**	**NE**
5	E	0	NE	E	0	NE
8	NE	1	E	NE	1	E
9	E	0	NE	**NE**	**0**	**NE**
15	**E**	**1**	**E**	E	0	NE
22	NE	1	E	NE	1	E
25	E	0	NE	E	0	NE
26	NE	1	E	NE	1	E
30	**NE**	**0**	**NE**	E	0	NE

La tasa de predicción se obtiene calculando el porcentaje de predicciones correctas. Para el salto 1, el predictor ha acertado 2 de las 10 predicciones (valores 15 y 30) por lo que la tasa de predicción es del 20%. Para el salto 2, el predictor ha acertado 3 de las 10 predicciones (valores 1, 2 y 9) por lo que la tasa de predicción es del 30%.

La tasa de predicción global se calcula de forma similar a la local pero teniendo en cuenta todas las predicciones realizadas. Así, de 20 predicciones se han acertado 5 por lo que la tasa de predicción global es del 25%.

Con el pseudocódigo del ejercicio A 2.2 y utilizando un predictor de un nivel basado en una PHT de dos entradas que almacena la predicción para cada salto realizada mediante un contador de Smith de 2 bit como el de la figura 2.16.b, dibuje una tabla en la que se muestre el resultado real de los dos saltos y las predicciones realizadas para la secuencia de valores de x. ¿Cuál es el porcentaje de acierto del predictor para cada salto?¿Y el porcentaje de acierto global del predictor?

```
0x000000 x:=(1,2,5,8,9,15,22,25,26,30);
0x000100 Desde i:=1 hasta 10 hacer {
0x001000    Si (x es impar)              // Salto_1
0x001100       a:=a+1;                   // Salto_1 es efectivo
0x010000    Si (x es múltiplo de 5)      // Salto_2
0x010100       a:=a-1;                   // Salto_2 es efectivo
0x011000 }
```



La tabla muestra la secuencia del resultado de los saltos y las predicciones realizadas por el algoritmo de Smith con contadores de saturación de 4 estados. Dado que los contadores son de 2 bits, el bit más significativo es el que determina la predicción: 1 el salto será efectivo (T), 0 el salto no será efectivo (NT).

Valor	Resultado salto 1	Entrada PHT salto 1	Predicción salto 1	Resultado salto 2	Entrada PHT salto 2	Predicción salto 2
1	E	00	NE	**NE**	**00**	**NE**
2	**NE**	**01**	**NE**	**NE**	**00**	**NE**
5	E	00	NE	E	00	NE
8	**NE**	**01**	**NE**	**NE**	**01**	**NE**
9	E	00	NE	**NE**	**00**	**NE**
15	E	01	NE	E	00	NE
22	NE	10	E	**NE**	**01**	**NE**
25	E	01	NE	E	00	NE
26	NE	10	E	**NE**	**01**	**NE**
30	**NE**	**01**	**NE**	E	00	NE

Para el salto 1, el predictor ha acertado 3 de las 10 predicciones (valores 2, 8 y 30) por lo que la tasa de predicción es del 30%. Para el salto 2, el predictor ha acertado 6 de las 10 predicciones (valores 1, 2, 8, 9, 22 y 26) por lo que la tasa de predicción es del 60%.

De 20 predicciones se han acertado 9 por lo que la tasa de predicción global es del 45%.

En un procesador superescalar que dispone de un predictor de saltos de dos niveles basado en historial global se ejecuta una secuencia de instrucciones con las siguientes direcciones y resultados:

i1:	0x00001	E	i11:	0x01011	E
i2:	0x00010	NE	i12:	0x01100	E
i3:	0x00011	NE	i13:	0x01101	NE
i4:	0x00100	NE	i14:	0x01110	NE
i5:	0x00101	E	i15:	0x01111	E
i6:	0x00110	E	i16:	0x10000	E
i7:	0x00111	NE	i17:	0x10001	NE
i8:	0x01000	NE	i18:	0x10010	NE
i9:	0x01001	E	i19:	0x10011	NE
i10:	0x01010	NE	i20:	0x10100	E

La longitud del registro BHR es de 2 bits. La función hash que se utiliza para reducir el número de bits de la dirección de memoria de la instrucción consiste en quedarse con el bit menos significativo de la dirección. El BHR y todos los contadores de la PHT se encuentran inicialmente en el estado 00. Dibuje una tabla en la que se muestre la evolución del BHR y de los contadores de la PHT. ¿Cuál es el porcentaje de acierto global del predictor?

La siguiente tabla muestra la evolución del estado del BHR y de los contadores de 2 bits de la PHT. La PHT cuenta con 8 contadores ya que el puntero de acceso se compone de 3 bits: 2 bits del historial global más el bit que proporciona el *hash* de la dirección de la instrucción.

x				PHT de 8 entradas								
	Resultado	Hash	BHR	0	1	2	3	4	5	6	7	Predicción
I1	E	1	00	00	00	00	00	00	00	00	00	NE
I2	NE	0	01		00			01				NE
I3	NE	1	10		00					00		NE
I4	NE	0	00	00						00		NE
I5	E	1	00	00				01				NE
I6	E	0	01		00			10				NE
I7	NE	1	11		01						00	NE
I8	NE	0	10			00					00	NE
I9	E	1	00			00		10				E
I10	NE	0	01		01			11				NE
I11	E	1	10		00					00		NE
I12	E	0	01		00					01		NE
I13	NE	1	11		01						00	NE
I14	NE	0	10			00					00	NE
I15	E	1	00			00		11				E
I16	E	0	01		01			11				NE
I17	NE	1	11		10						00	NE
18	NE	0	10			00					00	NE
19	NE	1	00			00		11				E
I20	E	0	00	00				10				NE

De 20 predicciones se han acertado 12 por lo que la tasa de predicción global es del 60%.

Actividad 2.5 Autor: José Sánchez Moreno

Un procesador superescalar que dispone de un predictor de saltos de dos niveles basado en historial local compuesto por una BHT de dos entradas de 2 bits de longitud y una PHT dotado de contadores de saturación de 2 bits. Las dos funciones hash que se utilizan para reducir el número de bits de que consta la dirección de una instrucción son iguales y consisten en quedarse con el bit menos significativo de la dirección. Utilizando la siguiente secuencia de 20 instrucciones con direcciones y resultados:

i1:	0x00001	E	i11:	0x01011	E
i2:	0x00010	NE	i12:	0x01100	E
i3:	0x00011	NE	i13:	0x01101	NE
i4:	0x00100	NE	i14:	0x01110	NE
i5:	0x00101	E	i15:	0x01111	E
i6:	0x00110	E	i16:	0x10000	E
i7:	0x00111	NE	i17:	0x10001	NE
i8:	0x01000	NE	i18:	0x10010	NE
i9:	0x01001	E	i19:	0x10011	NE
i10:	0x01010	NE	i20:	0x10100	E

Rellene una tabla en la que se muestre la evolución del estado del BHT y de la PHT, indicando el resultado de la dirección. Todas las entradas de las dos tablas se encuentran inicialmente el estado 00. ¿Cuál es el porcentaje de acierto global del predictor?

De acuerdo con los datos del enunciado, el predictor de dos niveles basado en historial local consta de una tabla BHT de 2 entradas de 2 bits y de una PHT con 8 contadores de saturación de 2 bits, todas ellas inicialmente a 00. El puntero que señala al contador que permite determinar la predicción del salto se forma concatenando los 2 bits de una de las entradas de la BHT con el bit menos significativo de la dirección de la instrucción obtenido mediante la función hash.

			BHT		PHT de 8 entradas								
	Resultado del salto	Hash	0	1	0	1	2	3	4	5	6	7	Predicción del salto
i1	E	1	00	00					**0**0				NE
i2	NE	0	00	01	**0**0				01				NE
i3	NE	1	00	01	00					**0**0			NE
i4	NE	0	00	10	**0**0					00			NE
i5	E	1	00	10	00						**0**0		NE
i6	E	0	00	01	**0**0						01		NE
i7	NE	1	01	01	01					**0**0			NE
i8	NE	0	01	10		**0**0				00			NE
i9	E	1	10	10		00					**0**1		NE
i10	NE	0	10	01			**0**0				10		NE
i11	E	1	00	01			00			**0**0			NE
i12	E	0	00	11	**0**1					01			NE
i13	NE	1	01	11	10							**0**0	NE
i14	NE	0	01	10	**0**0							00	NE
i15	E	1	10	10	00						**1**0		E
i16	E	0	10	01			**0**0				11		NE
i17	NE	1	01	01		01				**0**1			NE
i18	NE	0	01	10		**0**0				00			NE
i19	NE	1	10	10		00					**1**1		E
i20	E	0	10	00			**0**1				10		NE

De 20 predicciones se han acertado 11 por lo que la tasa de predicción global es del 55%.

Actividad 2.6 *Autor: José Sánchez Moreno*

Un procesador superescalar utiliza un predictor de saltos ***gshare*** compuesto por una BHR de 2 bits de longitud y una PHT con contadores de saturación de 2 bits. La función ***hash*** para reducir el número de bits de que consta la dirección de la instrucción consiste en quedarse con los tres bits menos significativos de la dirección. Utilizando la siguiente secuencia de 20 instrucciones con direcciones y resultados:

i1:	0x00001	E	i11:	0x01011	E
i2:	0x00010	NE	i12:	0x01100	E
i3:	0x00011	NE	i13:	0x01101	NE
i4:	0x00100	NE	i14:	0x01110	NE
i5:	0x00101	E	i15:	0x01111	E
i6:	0x00110	E	i16:	0x10000	E
i7:	0x00111	NE	i17:	0x10001	NE
i8:	0x01000	NE	i18:	0x10010	NE
i9:	0x01001	E	i19:	0x10011	NE
i10:	0x01010	NE	i20:	0x10100	E

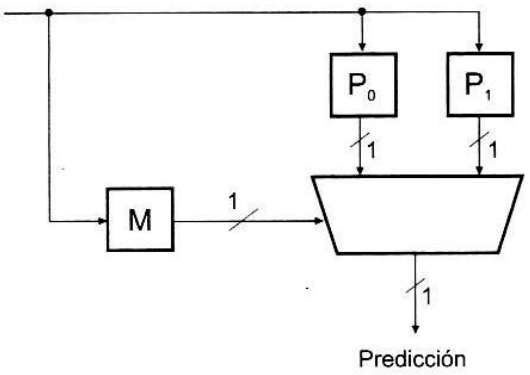
Rellene una tabla en la que se muestre la evolución del estado del BHR y de la PHT, indicando el resultado de la predicción en cada salto. Tanto el BHR como las entradas de la PHT se encuentran inicialmente el estado 00. ¿Cuál es el porcentaje de acierto global del predictor?

Para obtener el puntero que permite acceder al contador de la PHT primero es necesario realizar la XOR del contenido del BHR con los dos bits más significativos del resultado de la función *hash* para, a continuación, concatenar al resultado de la XOR con el bit menos significativo de la función *hash*. Dado que el puntero tiene una longitud de 3 bits, la PHT dispondrá de 8 contadores de saturación.

						PHT de 8 entradas								
	Salto	BHR	Hash	XOR	Puntero	0	1	2	3	4	5	6	7	Predicción del salto
i1	E	00	00 1	00⊕00=00	001 (1)		**0**0							N
i2	N	01	01 0	01⊕01=00	000 (0)	**0**0	01							N
i3	N	10	01 1	10⊕01=11	111 (7)	00							**0**0	N
i4	N	00	10 0	00⊕10=10	100 (4)					**0**0			00	N
i5	E	00	10 1	00⊕10=10	101 (5)					00	**0**0			N
i6	E	01	11 0	01⊕11=10	100 (4)					**0**0	01			N
i7	N	11	11 0	11⊕11=00	001 (1)		**0**1			01				N
i8	N	10	00 0	10⊕00=10	100 (4)		00			**0**1				N
i9	E	00	00 1	00⊕00=00	001 (1)		**0**0			00				N
i10	N	01	01 0	01⊕01=00	000 (0)	**0**0	01							N
i11	E	10	01 1	01⊕10=11	111 (7)	00							**0**0	N
i12	E	01	10 0	10⊕01=11	110 (6)							**0**0	01	N
i13	NE	11	10 1	10⊕11=01	011 (3)				**0**0			01		N
i14	NE	10	11 0	11⊕10=01	010 (2)			**0**0	00					N
i15	E	00	11 1	11⊕00=11	111 (7)			00					**0**1	N
i16	E	01	00 0	00⊕01=01	010 (2)			**0**0					10	N
i17	NE	11	00 1	00⊕11=11	111 (7)			01					**1**0	E
i18	NE	10	01 0	01⊕10=11	110 (6)							**0**1	01	N
i19	NE	00	01 1	01⊕00=01	011 (3)				**0**0			00		N
i20	E	00	10 0	10⊕00=10	100 (4)				00	**0**0				N

De 20 predicciones se han acertado 10 por lo que la tasa de predicción global es del 50%.

Otra forma de especular el resultado de un salto es mediante un predictor híbrido que recurre a combinar varios algoritmos de predicción. Uno de los más conocidos es el predictor de torneo formado por dos predictores, P0 y P1, y un mecanismo de selección o selector, denominado M. La figura muestra un esquema de esta clase de predictores híbridos.



En cada salto, ambos predictores generan su predicción y el selector decide cuál de los dos ofrece la mejor predicción en base a los resultados obtenidos con anterioridad. El selector es una PHT con contadores de saturación de 2 bits a la que se accede con el último bit de la dirección de la instrucción de salto. Las reglas de actualización de los contadores del selector son las siguientes:

- Si en la última predicción P0 y P1 fallaron, el contador no se modifica
- Si P0 falló y P1 acertó el contador se incrementa.
- Si P0 acertó y P1 falló el contador se decrementa.
- Si los dos acertaron, el contador no se modifica.
- Con el bit de más peso del contador se elige P0 (si vale 0) o P1 (si vale 1)

Con un predictor de torneo con las siguientes características:

- M utiliza una PHT de dos entradas, todas ellas a 00.
- P0 es un *gshare* con BHR de 2 bits y PHT de 4 entradas de 2 bits. La función *hash* produce los 2 bits menos significativos de la dirección.
- P1 es un predictor de historial global con BHR de 1 bit y PHT de 4 entradas de 2 bits. La función *hash* produce el bit menos significativo de la dirección.
- Los BHRs de P0 y P1 están inicialmente a 00.
- Para P0 y P1, PHT0= 00, PHT1=01, PHT2=10, PHT3=11.
- P0 y P1 se actualizan según sus respectivas reglas de actualización.

y dada la siguiente secuencia de direcciones de instrucciones de salto y sus resultados

```
i1:  576 NT
i2:  604 T
i3:  599 T
i4:  604 T
i5:  599 NT
i6:  604 T
i7:  599 NT
i8:  604 T
i9:  599 NT
i10: 604 T
```

Rellene una tabla con la evolución del estado de M, P0 y P1. Calcule el porcentaje de acierto de los predictores individuales y del híbrido.

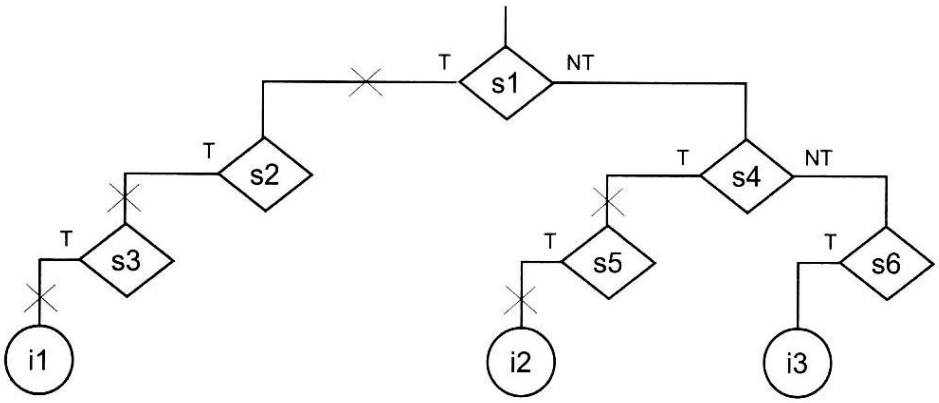
La siguiente tabla muestra la evolución de los dos predictores y del selector.

					P0							P1					
		M					PHT						PHT				
	Res	P0	P1	2 bits Direc	BHR	XOR	0	1	2	3	Pred	BHR	0	1	2	3	Pred
i1	NT	00	00	10	00	10	00	01	10	11	T	0	00	01	10	11	**NT**
i2	T	01	00	00	00	00	00		01		NT	0	00				NT
i3	T	01	00	01	01	00	01				NT	1	01			11	**T**
i4	T	01	01	00	11	11	10			11	**T**	1		01		11	NT
i5	NT	00	01	01	11	10			01	11	**NT**	1		10		11	T
i6	T	00	00	00	10	10			00		NT	0	01			10	NT
i7	NT	00	00	01	01	00	10		01		T	1	10			10	T
i8	T	00	00	00	10	10	01		01		NT	0	10			01	**T**
i9	NT	01	00	01	01	00	01		10		NT	1	11			01	NT
i10	T	01	00	00	10	10	00		10		**T**	0	11			00	T

Resultado real	**Pred. M**	**Pred. P0**	**Pred. P1**
NT	T	T	NT
T	NT	NT	NT
T	NT	NT	T
T	T	T	NT
NT	NT	NT	T
T	NT	NT	NT
NT	T	T	T
T	NT	NT	T
NT	TT	NT	NT
T	T	T	T
	40%	40%	50%

Actividad 2.8 *Autor: José Sánchez Moreno*

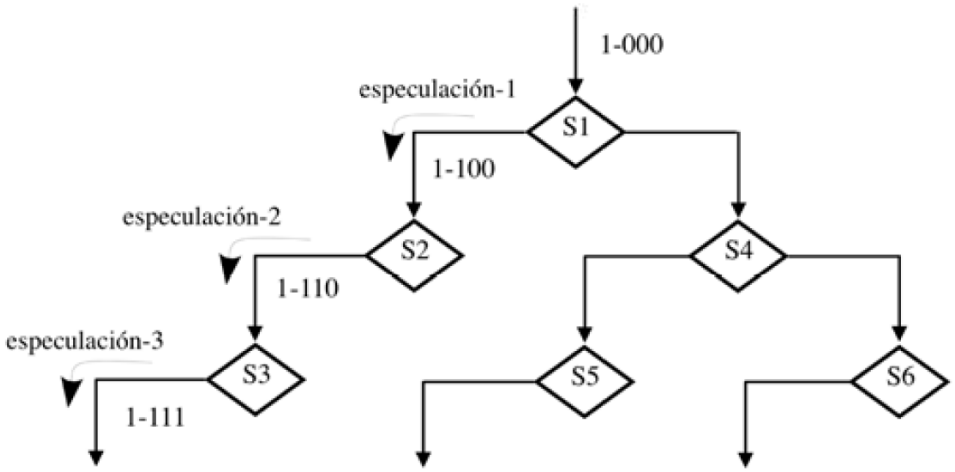
La siguiente figura muestra la ejecución de un conjunto de instrucciones, incluyendo las rutas especuladas que han sido invalidadas y las que no lo han sido.



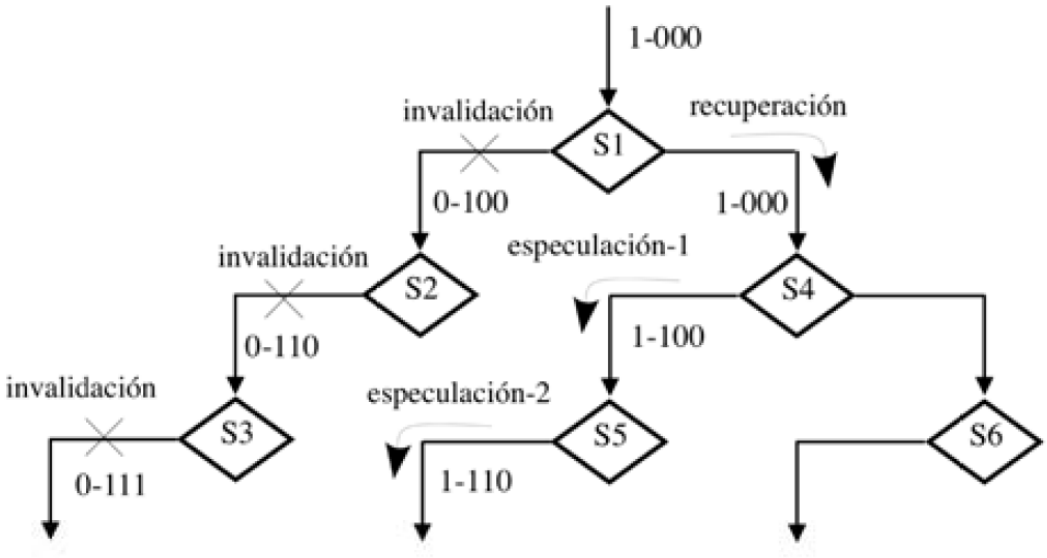
Dibuje la evolución paso a paso de los bits de validez y de especulación de cada instrucción según se hayan ido especulando, validando, invalidando y recuperando. Considere que:

- El campo de especulación tiene una longitud de 3 bits.
- El procesador siempre especula que los saltos son efectivos.
- El ancho de la segmentación es 1.
- Las cinco etapas IF, ID, II, EX, WB son de un ciclo de duración.
- El resultado real del salto se conoce al final de la etapa EX.

Dado que el procesador siempre especula con que un salto es efectivo, las instrucciones s2, s3 e i1 se ejecutan normalmente a continuación de s1, dando lugar a tres rutas de especulación: la que inicia s1 (100), la que inicia s2 (110) y la que inicia s3 (111).

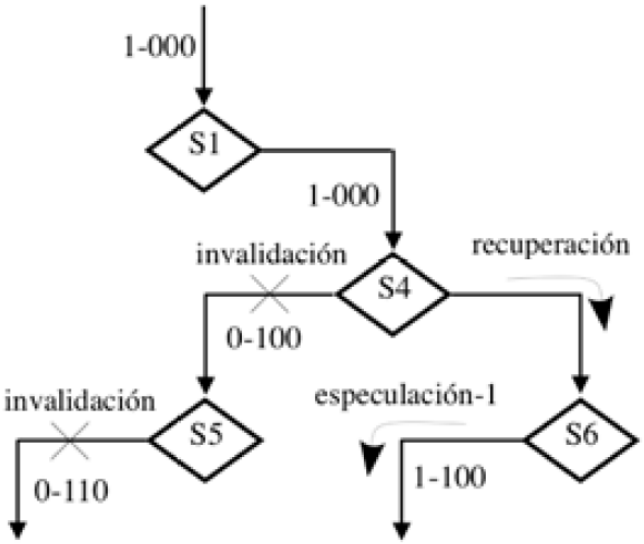


Al llegar s1 al final de su etapa EX se conoce el resultado real del salto y resulta no coincidir con su especulación. Por ello, las tres rutas especuladas a que dio lugar s1 se invalidan. Ello significa que se cambian los bits de validez a 0 de todas las instrucciones que hubiesen sido especuladas. En este ejemplo, se eliminan las instrucciones s2, s3 e i1.



Tras esto se realiza la recuperación de s1 que consiste en continuar con la ejecución de la instrucción siguiente, la s4. De nuevo, el procesador especula dando lugar a dos nuevas rutas especuladas.

Al llegar s1 al final de su etapa EX, el resultado del salto vuelve a no coincidir con la especulación y se anulan las dos rutas especuladas, esto es, las instrucciones s5 e i2. La instrucción s6 especula e inicia el procesamiento de i3. El resultado verdadero de i6 coincide con la especulación y se validan la ruta especulada, lo que provoca el cambio de los bits de especulación de las instrucciones que haya en la ruta, en este caso, la i3.



¿Cómo modificaría el formato de la entrada de la estación de reserva para operaciones aritméticas si uno de los operandos fuente fuese un valor inmediato y no identificador/valor de un registro?¿como sería el formato de las entradas de una estación de reserva individual que alimenta la unidad funcional de carga/almacenamiento?

Esta pregunta no tiene una solución única ya que es más de razonamiento y de proporcionar soluciones lógicas y sensatas. Aquí va la que yo propongo.

Para incluir campos con inmediatos en la estación de reserva no es necesario realizar ninguna modificación ya que en el propio código de operación de la instrucción, y que se almacena en la entrada de la estación, incluye la información de que uno de los campos es un valor numérico contenido en el formato de la instrucción (importante este aspecto: el valor del inmediato va en los bytes que forman la instrucción). En ese caso, al distribuir la instrucción, el campo de la entrada de la estación de reserva en el que se almacenase el operando inmediato colocaría su bit de validez a 1 ya que el operando estaría ya disponible (se habría copiado en la entrada al distribuir la instrucción).

Las entradas de una estación de reserva para una unidad funcional de carga/almacenamiento serían ligeramente diferentes a las entradas para las instrucciones aritméticas. Si se considera que el formato genérico de las instrucciones de carga/almacenamiento es el siguiente:

```
LD registro destino, desplazamiento(registro base)
SD desplazamiento(registro base), registro fuente
```

Los campos de una entrada podrían ser:

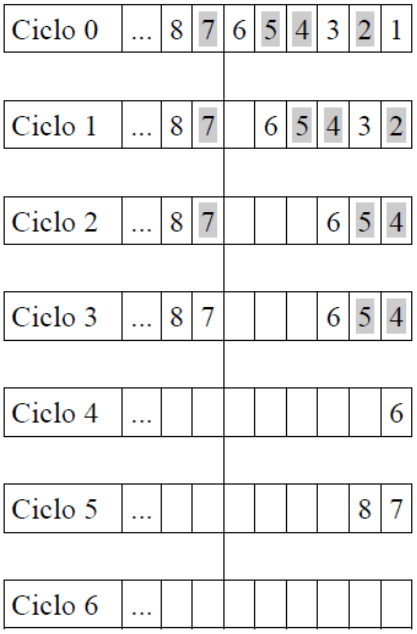
- Bit de entrada ocupada (O).
- Código de operación (COp).
- Operando fuente 1 (Op1). El registro fuente si se trata de un almacenamiento. En caso de tratarse de una carga, no se utilizaría.
- Bit de validez 1 (V1).
- Campo desplazamiento (Desp). No necesita bit de validez ya que es un valor inmediato que está almacenado en el formato de la instrucción.
- Operando fuente 2 (Op2). Corresponde al registro base.
- Bits de validez 2 (V2).
- Operando destino (D). El registro destino de una carga. En caso de tratarse de un almacenamiento, no se utilizaría.
- Bit de instrucción lista (L).

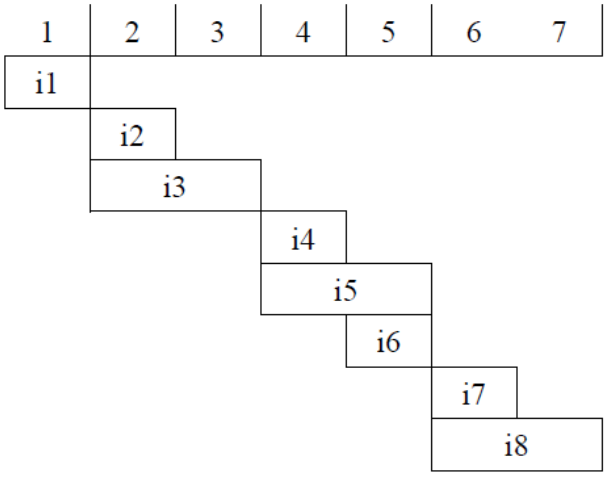
El siguiente fragmento de código:

```
i1: ADD   R3,R2,R1
i2: ADD   R8,R3,R7
i3: MULT  R6,R5,R4
i4: ADD   R9,R6,R1
i5: MULT  R10,R3,R6
i6: ADD   R11,R2,R1
i7: ADD   R12,R3,R8
i8: MULT  R13,R2,R1
```

Se encuentra ubicado en una ventana de instrucciones de 10 entradas desde la que se distribuye a una estación de reserva de 6 entradas que alimenta a una unidad funcional de suma/resta (1 ciclo) y a una de multiplicación/división (2 ciclos y segmentada). Se considera que en el mismo ciclo en que las unidades funcionales generan el resultado, la estación de reserva actualiza sus bits de validez; esto permite que en el ciclo siguiente se pueda emitir la instrucción. Se pueden emitir un máximo de dos instrucciones/ciclo, una a cada unidad funcional.¿cual es la secuencia temporal de ejecución y la evolución de la estación de reserva si se realiza emisión alineada ordenada?¿cual sería la mejora en el rendimiento que se obtendría si se realizase emisión no alineada y desordenada?

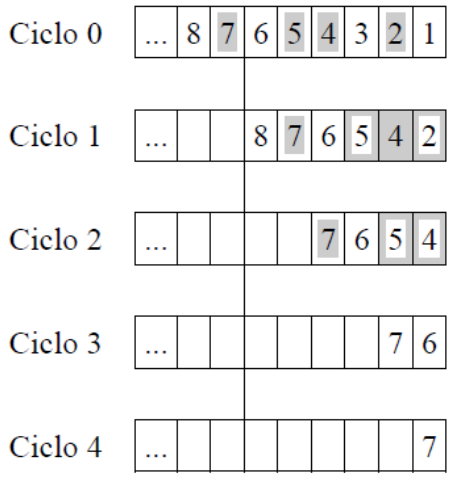
La emisión alineada implica que no se puedan distribuir nuevas instrucciones hasta que la estación de reserva no quede vacía. Dado que la secuencia de código cuenta con 8 instrucciones, en el ciclo 0 se distribuirán 6 instrucciones a la estación de reserva y quedarán 2 en la ventana de instrucciones a la espera. Que la emisión sea con bloqueo implica que una instrucción sólo puede emitirse cuando todas las instrucciones previas a ella hayan sido emitidas. La siguiente figura muestra el proceso de emisión alineada y con bloqueo de la secuencia de código:

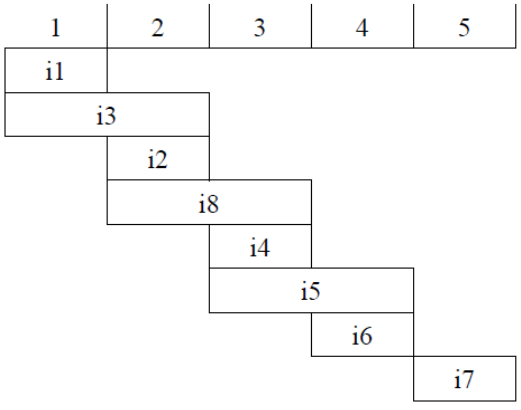




Observe que en el ciclo 2 se emiten dos instrucciones simultáneamente. Ello es debido a que las dos unidades están libres y las instrucciones i2 e i3 tienen los operandos listos por lo que no hay ningún impedimento para proceder a la emisión. Lo mismo sucede en el ciclo 4, en el que se emiten i4 e i5, y en el ciclo 6, en el que se emiten i7 e i8. En los ciclos 3 y 5 no se ha producido emisión ya que ninguna instrucción estaba disponible como consecuencia de las dependencias verdaderas de datos. El total de ciclos consumidos en la ejecución es 7.

La siguiente figura corresponde a la emisión sin alineamiento y sin bloqueo de las instrucciones. Ahora una instrucción se emite cuando tiene sus dos operandos disponibles y la unidad funcional puede admitirla. Los ciclos empleados en la ejecución son 5, obteniéndose una mejora del 40% con respecto a la emisión ordenada y alienada



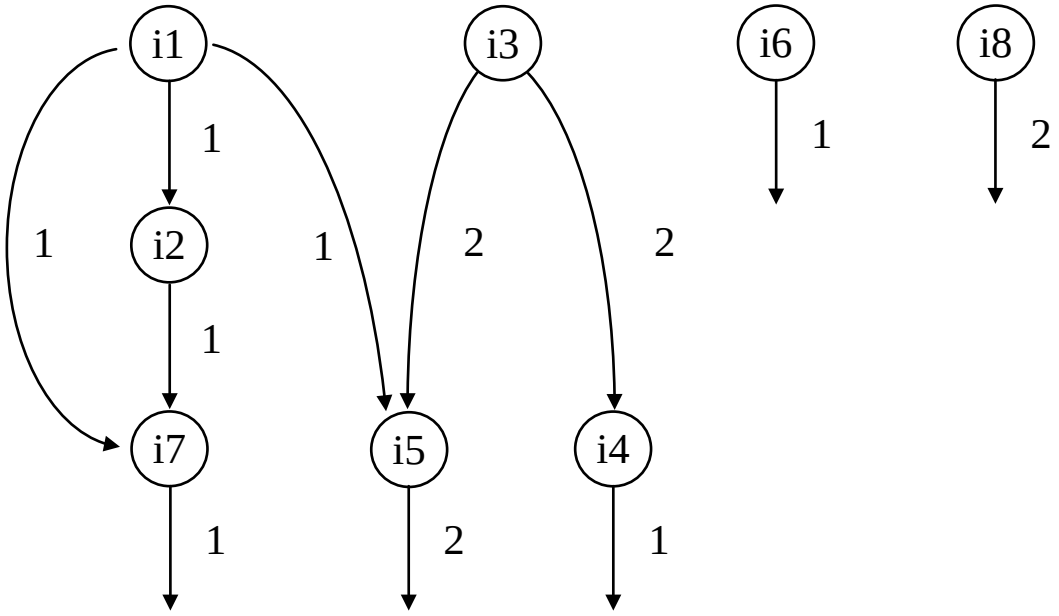


Actividad 2.11 Autor: José Sánchez Moreno

El siguiente fragmento de código, ¿Cuál el límite del flujo de datos?

```
i1: ADD   R3,R2,R1
i2: ADD   R8,R3,R7
i3: MULT  R6,R5,R4
i4: ADD   R9,R6,R1
i5: MULT  R10,R3,R6
i6: ADD   R11,R2,R1
i7: ADD   R12,R3,R8
i8: MULT  R13,R2,R1
```

El límite del flujo de datos es 4 ciclos. Recuerda que el límite de flujo de datos es el mejor rendimiento que se podría obtener en caso de que el procesador tuviese recursos ilimitados. El límite del flujo de datos viene impuesto por las dependencias verdaderas. Adjunto una figura en la que se aprecia la razón de que el límite sea 4.



He visto un ejemplo en el tema 3. No sé si habrá otras referencias en el tema 2.

Respecto al ejercicio, y ahora que creo entenderlo, ¿no debería haber un arco entre i1 e i5, de un ciclo?

Un saludo.

Jesús Moreno

Sí, falta un arco que exprese la existencia de una dependencia RAW entre ambas instrucciones.

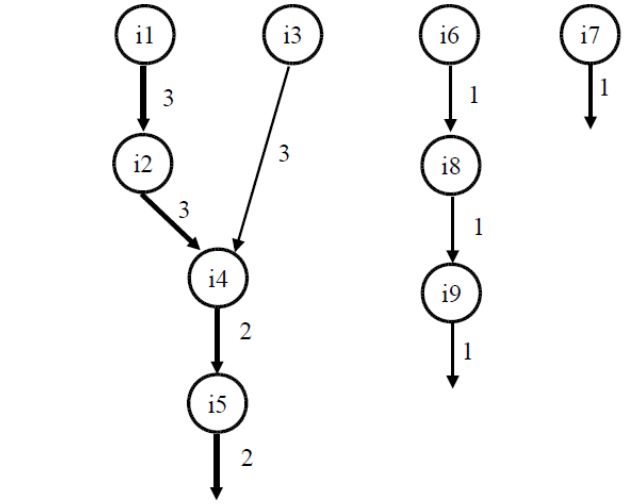
Actividad 2.12 *Autor: José Sánchez Moreno*

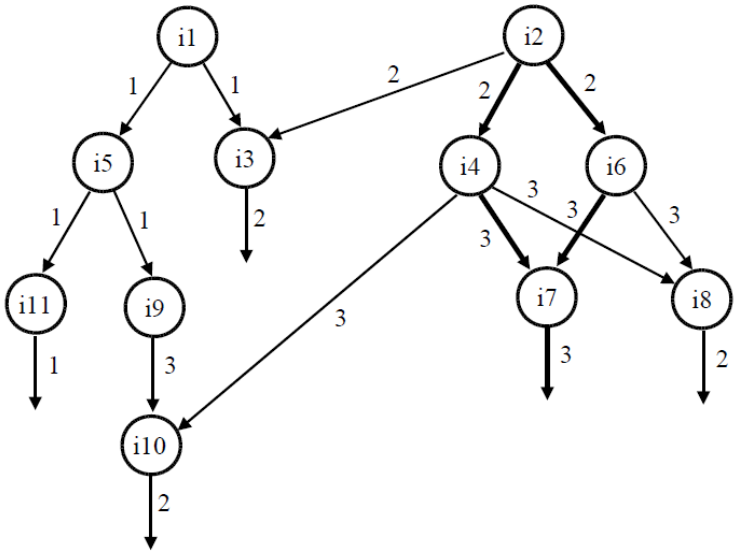
Dadas las siguientes secuencias de código:

```
i1: LD    F2,0(R1)      i1: ADDI  R1,R0,#3
i2: MULT  F4,F2,F0      i2: SUBD  F4,F8,F6
i3: LD    F6,0(R2)      i3: SD    0(R1),F4
i4: ADDD  F6,F4,F6      i4: DIVD  F2,F4,F8
i5: SD    0(R6),F6      i5: SUBI  R1,R1,#1
i6: ADDI  R1,R1,#8      i6: DIVD  F6,F4,F8
i7: ADDI  R2,R2,#8      i7: MULTD F5,F2,F6
i8: SGT   R3,R1,#800    i8: SUBD  F10,F2,F6
i9: BEQZ  R3,i1         i9: LD    F4,0(R1)
                        i10:ADDD  F0,F4,F2
                        i11:BNEQZ R1,i2
```

Calcule los límites del flujo de datos para las dos secuencias. Las operaciones enteras consumen 1 ciclo, las cargas 3 ciclos, los almacenamientos 2 ciclos, las sumas/restas en coma flotante 2 ciclos y las multiplicaciones/divisiones 3 ciclos.

Las dos figuras situadas a continuación muestran el diagrama de flujo de datos de las secuencias de código. En el primer caso, la límite del flujo de datos es de 10 ciclos y en el segundo caso de 8 ciclos.





Actividad 2.13 Autor: José Sánchez Moreno

Dados los siguientes fragmentos de código:

```
i1: DIV R1,R2,R3
i2: ADD R4,R1,R5
i3: ADD R5,R6,R7
i4: ADD R1,R8,R9
```

```
i1: LD F2,0(R1)
i2: MULT F4,F2,F0
i3: LD F6,0(R2)
i4: ADD F6,F4,F6
i5: SD 0(R2),F6
i6: ADDI R1,R1,#8
i7: ADDI R2,R2,#8
i8: SGT R3,R1,#800
i9: BEQZ R3,i1
```

- Señale las dependencias de datos y de memoria existentes.
- Renombre el código e indique qué dependencias permanecen.
- Analice lo que sucede con el registro R1 en sucesivas iteraciones de la segunda secuencia de código.

En el primer fragmento de código, las dependencias existentes son:

```
i1: DIV R1,R2,R3
i2: ADD R4,R1,R5     // dependencia RAW con i1 por R1
i3: ADD R5,R6,R7     // dependencia WAR con i2 por R5
i4: ADD R1,R8,R9     // dependencia WAR con i2 por R1
                     // dependencia WAW con i1 por R1
```

y el código con los registros renombrados:

```
i1: DIV Rr1,R2,R3
i2: ADD Rr2,Rr1,R5
i3: ADD Rr3,R6,R7
i4: ADD Rr4,R8,R9
```

En lo relativo al segundo fragmento de código, las dependencias de datos existentes con respecto a la primera iteración del bucle son:

```
i1: LD F2,0(R1)
i2: MULT F4,F2,F0    // dependencia RAW con i1
i3: LD F6,0(R2)
i4: ADD F6,F4,F6     // dependencia RAW con i2 e i3
                     // dependencia WAW con i3
i5: SD 0(R2),F6      // dependencia RAW con i4
i6: ADDI R1,R1,#8    // dependencia WAR con i1
i7: ADDI R2,R2,#8    // dependencia WAR con i3 e i5
i8: SGT R3,R1,#800   // dependencia RAW con i6
i9: BEQZ R3,i1       // dependencia RAW con i8
```

Obsérvese que existe una dependencia de memoria WAR entre las instrucciones `i3` e `i5` debida a los dos accesos a la dirección de memoria `0(R2)` La instrucciónde carga `i3` lee de esa posición de memoria mientras que la instrucción de almacenamiento `i5` tiene que escribir, lo que implica la existencia de un riesgo WAR.

El código con los registros renombrados es el siguiente:

```
i1: LD Fr1,0(R1)
i2: MULT Fr2,Fr1,F0
i3: LD Fr3,0(R2)
i4: ADD Fr4,Fr2,Fr3
i5: SD 0(R2),Fr4
i6: ADDI Rr1,R1,#8
i7: ADDI Rr2,R2,#8
i8: SGT Rr3,Rr1,#800
i9: BEQZ Rr3,i1
```

Las dependencias que permanecen son las de tipo RAW ya que el renombramiento no las elimina.
De ello se ocupan las estaciones de reserva, comprobando cuándo están disponibles los operandos que se necesitan para poder ejecutar una instrucción.
El renombramiento del segundo fragmento de código corresponde a la primera iteración del bucle. El registro R1 cuando hace de operando fuente en las instrucciones i1 e i6 no se renombra en la primera iteración pero cuando actúa como destino en i6 se renombra como Rr1. En la segunda iteración, el valor de R1 como operando fuente en i1 e i6 viene dado por la ejecución de la instrucción i6 en la primera iteración, cuyo resultado se almacenó en Rr1 (renombramiento de R1). Cuando i1 se distribuya en la segunda iteración, se accederá al ARF para analizar el estado del operando fuente R1.
Si el renombramiento de R1 sigue vigente ya que la i6 de la primer iteración no ha terminado, el ARF indicará que R1 está ocupado y se accederá a su registro de renombramiento en el RFF, esto es, el Rr1; ahora, según el estado de Rr1 (pendiente de escritura, pendiente de terminar) al operando R1 en la entrada de la estación de reserva se le asignará el valor de Rr1 o su identificador. Si el registro R1 ya estuviese libre (por terminación de la i6 de la primera iteración), se leería el valor de R1 en ARF.
Si la i6 de la segunda iteración se distribuye antes que la i6 de la primera haya terminado, será necesario un segundo renombramiento de R1. Análogamente podría suceder en las sucesivas iteraciones.

Actividad 2.14 Autor: Fernando Silva

El siguiente conjunto de operaciones:

```
R1 ← R2+R3     % 1 ciclo de latencia
R4 ← R1-F5     % 1 ciclo de latencia
R6 ← R4*R7     % 2 ciclo de latencia
R7 ← R2+R8     % 1 ciclo de latencia
```

Está ubicado en un buffer de distribución que alimenta a dos estaciones de reserva individuales asignadas cada una de ellas a una unidad funcional de suma/resta (1 ciclo) y a una de multiplicación/división (2 ciclos y segmentada). Dibuje la evolución del estado del buffer de distribución, de las estaciones de reserva, del ARF, del RRF y del buffer de terminación. Tenga en cuenta que:

- Se utiliza planificación con lectura de operandos.
- El ARF consta de 8 registros, R1 a R8, con valores iniciales de 10 a 80, respectivamente.
- El RRF consta de 4 registros Rr1, Rr2, Rr3 y Rr4.
- El buffer de distribución distribuye 4 instrucciones/ciclo.
- Las estaciones de reserva individuales disponen de 2 entradas.
- En el mismo ciclo en que una unidad funcional genera un resultado, la estación de reserva actualiza sus bits de validez; esto permite que en el ciclo siguiente se pueda emitir otra instrucción y terminar la finalizada.

EVENTOS - CICLO 1
Se distribuye i1. R1 se renombra a Rr1. Los operandos están listos.
Se distribuye i2. R1 está renombrado como Rr1. Se renombra R4 a Rr2. No está lista.
Se distribuye i3. R4 está renombrado como Rr2. Se renombra R6 a Rr3. No está lista.

ARF	Valor	Índice
R1	**10**	**Rr1**
R2	**20**	--
R3	**30**	--
R4	**40**	**Rr2**
R5	**50**	--
R6	**60**	**Rr3**
R7	**70**	--
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	--	--	**1**
Rr2	--	--	**1**
Rr3	--	--	**1**
Rr4	--	--	**0**

instrucciones					
	Cod.	**Op1**	**Op2**	**Dest.**	**Ocup.**
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	1

Estacion de reserva de unidad Sumadora							
	Cod	**Op1**	**V1**	**Op2**	**v2**	**Dest**	**L**	**Ocup**
i1	ADD	R2 (20)	1	R3 (30)	1	Rr1	1	1
i2	SUB	Rr1	0	R5 (50)	1	Rr2	0	1

Estacion de reserva de unidad Multiplicadora								
	Cod	**Op1**	**V1**	**Op2**	**v2**	**Dest**	**L**	**Ocup**
i3	MUL	Rr2	0	R7 (70)	1	Rr3	0	1
	--	--	--	--	--	--	--	0

Buffer de Terminación								
Ocup.	**Emit.**	**Final.**	**Dir.**	**R.Dest.**	**R.Renom.**	**Espec.**	**Valid.**	**Ptr**
1	0	0	i1	R1	Rr1	No	1	Head
1	0	0	i2	R4	Rr2	No	1	
1	0	0	i3	R6	Rr3	no	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 2 (INICIO)
La unidad sumadora emite i1.
Se distribuye i4. Se renombra R7 a Rr4. Los operandos estan listos.

ARF	Valor	Índice
R1	**10**	**Rr1**
R2	**20**	**--**
R3	**30**	**--**
R4	**40**	**Rr2**
R5	**50**	**--**
R6	**60**	**Rr3**
R7	**70**	**Rr4**
R8	**80**	**--**

RRF	Valor	Valido	Ocup.
Rr1	--	--	**1**
Rr2	--	--	**1**
Rr3	--	--	**1**
Rr4	--	--	**1**

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	ADD	R2 (20)	1	R8 (80)	1	Rr4	1	1
i2	SUB	Rr1	0	R5 (50)	1	Rr2	0	1

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	MUL	Rr2	0	R7 (70)	1	Rr3	0	1
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
1	1	0	i1	R1	Rr1	No	1	Head
1	0	0	i2	R4	Rr2	No	1	
1	0	0	i3	R6	Rr3	No	1	
1	0	0	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 2 (FIN)
Finaliza la ejecución de i1. Se publica **Rr1 = 50**.
i2 coge el resultado y queda lista para ejecutarse.

ARF	Valor	Índice
R1	**10**	**Rr1**
R2	**20**	--
R3	**30**	--
R4	**40**	**Rr2**
R5	**50**	--
R6	**60**	**Rr3**
R7	**70**	**Rr4**
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	**50**	**1**	**1**
Rr2	--	--	**1**
Rr3	--	--	**1**
Rr4	--	--	**1**

	Estacion de reserva de unidad Sumadora							
	Cod	**Op1**	**V1**	**Op2**	**v2**	**Dest**	**L**	**Ocup**
i4	ADD	R2 (20)	1	R8 (80)	1	Rr4	1	1
i2	SUB	Rr1 (50)	1	R5 (50)	1	Rr2	1	1

	Buffer de distribucion de instrucciones				
	Cod.	**Op1**	**Op2**	**Dest.**	**Ocup.**
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

	Estacion de reserva de unidad Multiplicadora							
	Cod	**Op1**	**V1**	**Op2**	**v2**	**Dest**	**L**	**Ocup**
i3	MUL	Rr2	0	R7 (70)	1	Rr3	0	1
	--	--	--	--	--	--	--	0

Buffer de Terminación								
Ocup.	**Emit.**	**Final.**	**Dir.**	**R.Dest.**	**R.Renom.**	**Espec.**	**Valid.**	**Ptr**
1	1	1	i1	R1	Rr1	No	1	Head
1	0	0	i2	R4	Rr2	No	1	
1	0	0	i3	R6	Rr3	No	1	
1	0	0	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 3 (INICIO)
Se termina i1, escribiendo el resultado en el ARF y liberando el RRF.
Se emite i2.
i4 permanece a la espera ya que la unidad sumadora está ocupada por i2.

ARF	Valor	Índice
R1	**50**	--
R2	**20**	--
R3	**30**	--
R4	**40**	**Rr2**
R5	**50**	--
R6	**60**	**Rr3**
R7	**70**	**Rr4**
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	--	--	**0**
Rr2	--	--	**1**
Rr3	--	--	**1**
Rr4	--	--	**1**

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	ADD	R2 (20)	1	R8 (80)	1	Rr4	1	1
i2	SUB	Rr1 (50)	1	R5 (50)	1	Rr2	1	0

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	MUL	Rr2	0	R7 (70)	1	Rr3	0	1
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
0	--	--	--	--	--	--	--	
1	1	0	i2	R4	Rr2	No	1	Head
1	0	0	i3	R6	Rr3	No	1	
1	0	0	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 3 (FIN)
Finaliza i2. Publica **Rr2 = 0**.
i3 intercepta el Rr2. Ahora está lista para ejecutarse.
i4 ya tiene lugar en la unidad funcional. Se ejecutará en el siguiente ciclo.

ARF	Valor	Índice
R1	**50**	**0**
R2	**20**	**0**
R3	**30**	**0**
R4	**40**	**Rr2**
R5	**50**	**0**
R6	**60**	**Rr3**
R7	**7**	**Rr4**
R8	**80**	**0**

RRF	Valor	Valido	Ocup.
Rr1	--	--	**0**
Rr2	**0**	**1**	**1**
Rr3	--	--	**1**
Rr4	--	--	**1**

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	ADD	R2 (20)	1	R8 (80)	1	Rr4	1	1
	--	--	--	--	--	--	--	0

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	MUL	Rr2 (0)	1	R7 (70)	1	Rr3	1	1
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
0	--	--	--	--	--	--	--	
1	1	1	i2	R4	Rr2	No	1	Head
1	0	0	i3	R6	Rr3	No	1	
1	0	0	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 4 (INICIO)
Se termina i2 del buffer de terminación. Escribiendo en ARF y limpiando el RRF.
Se emite i4.
Se emite i3.

ARF	Valor	Índice
R1	**50**	--
R2	**20**	--
R3	**30**	--
R4	**0**	--
R5	**50**	--
R6	**60**	**Rr3**
R7	**70**	**Rr4**
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	--	--	**0**
Rr2	--	--	**0**
Rr3	--	--	**1**
Rr4	--	--	**1**

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	ADD	R2 (20)	1	R8 (80)	1	Rr4	1	0
	--	--	--	--	--	--	--	0

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	MUL	Rr2 (0)	1	R7 (70)	1	Rr3	1	0
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	
1	1	0	i3	R6	Rr3	No	1	Head
1	1	0	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 4 (FIN)
i4 finaliza de ejecutarse, publicando **Rr4 = 100**
i3 termina su primer ciclo. Aun le falta uno más.

ARF	Valor	Índice
R1	**50**	--
R2	**20**	--
R3	**30**	--
R4	**0**	--
R5	**50**	--
R6	**60**	**Rr3**
R7	**70**	**Rr4**
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	--	--	**0**
Rr2	--	--	**0**
Rr3	--	--	**1**
Rr4	**100**	**1**	**1**

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	ADD	R2 (20)	1	R8 (80)	1	Rr4	1	0
	--	--	--	--	--	--	--	0

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	MUL	Rr2 (0)	1	R7 (70)	1	Rr3	1	0
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	
1	1	0	i3	R6	Rr3	No	1	Head
1	1	1	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 5 (INICIO)
i3 comienza su 2do ciclo de ejecución. Una nueva instrucción podría entrar al cauce de la U. funcional.
i4 está lista para ser terminada, pero el buffer aún tiene pendiente i3.

ARF	Valor	Índice
R1	**50**	--
R2	**20**	--
R3	**30**	--
R4	**0**	--
R5	**50**	--
R6	**60**	**Rr3**
R7	**70**	**Rr4**
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	--	--	**0**
Rr2	--	--	**0**
Rr3	--	--	**1**
Rr4	**100**	**1**	**1**

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	--	--	--	--	--	--	--	0
	--	--	--	--	--	--	--	0

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	MUL	Rr2 (0)	1	R7 (70)	1	Rr3	1	0
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	
1	1	0	i3	R6	Rr3	No	1	Head
1	1	1	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 5 (FIN)
i3 termina de ejecutarse y publica **Rr3 = 0**

ARF	Valor	Índice
R1	**50**	--
R2	**20**	--
R3	**30**	--
R4	**0**	--
R5	**50**	--
R6	**60**	**Rr3**
R7	**70**	**Rr4**
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	--	--	**0**
Rr2	--	--	**0**
Rr3	**0**	**1**	**1**
Rr4	**100**	**1**	**1**

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	--	--	--	--	--	--	--	0
	--	--	--	--	--	--	--	0

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	MUL	Rr2 (0)	1	R7 (70)	1	Rr3	1	0
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	
1	1	1	i3	R6	Rr3	No	1	Head
1	1	1	i4	R7	Rr4	No	1	
0	--	--	--	--	--	--	--	Tail

EVENTOS - CICLO 6
El buffer de terminacion termina i3 e i4, escribiendo en el ARF y limpiando el RRF.

ARF	Valor	Índice
R1	**50**	--
R2	**20**	--
R3	**30**	--
R4	**0**	--
R5	**50**	--
R6	**0**	--
R7	**100**	--
R8	**80**	--

RRF	Valor	Valido	Ocup.
Rr1	--	--	**0**
Rr2	--	--	**0**
Rr3	**0**	**1**	**0**
Rr4	**100**	**1**	**0**

Buffer de distribucion de instrucciones

	Cod.	Op1	Op2	Dest.	Ocup.
i1	ADD	R2	R3	R1	0
i2	SUB	R1	R5	R4	0
i3	MUL	R4	R7	R6	0
i4	ADD	R2	R8	R7	0

Estacion de reserva de unidad Sumadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i4	--	--	--	--	--	--	--	0
	--	--	--	--	--	--	--	0

Estacion de reserva de unidad Multiplicadora

	Cod	Op1	V1	Op2	v2	Dest	L	Ocup
i3	--	--	--	--	--	--	--	0
	--	--	--	--	--	--	--	0

Buffer de Terminación

Ocup.	Emit.	Final.	Dir.	R.Dest.	R.Renom.	Espec.	Valid.	Ptr
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	
0	--	--	--	--	--	--	--	Head/Tail

Actividad 2.15 Autor: José Sánchez Moreno

Considere un esquema de planificación dinámica con lectura de operandos que utiliza renombramiento basado en un RRF independiente y con acceso asociativo. En el esquema propuesto, los valores de los operandos fuente que no tienen escrituras pendientes se leen directamente del ARF sin sufrir ningún tipo de renombramiento. Describa un posible proceso de renombramiento si se considerase que todos los operandos tienen que ser renombrados ¿sobraría o faltaría algún campo en el ARF y en el RRF?

Al ser todos los registros renombrados, ya se trate de operandos fuente o destino, no sería necesario un campo *Ocupado* en el ARF. Las lecturas de operandos se realizarían directamente en el RRF, efectuando una búsqueda asociativa para saber qué registro de renombramiento se ha asociado al registro original y utilizando el valor o el identificador del registro de renombramiento según su bit de *Válido*.

Si hay que leer el valor de un operando fuente, se consulta si tiene una entrada en el RRF. Pueden darse varias posibilidades:

- No tiene entrada el registro en el RRF: La primera entrada libre en el RRF se asigna al registro fuente. Se copia el identificador del registro en el campo *Destino*, se copia el campo *Datos* del ARF en el campo *Datos* del RRF y se colocan el bit de *Válido* y el bit de *Último* a 1. Se utiliza como operando el valor del campo *Datos* del RRF.
- Ya dispone de una entrada: Una vez localizada la entrada, si tiene el bit de *Válido* a 0 se utiliza como operando fuente el identificador del registro de renombramiento. Si el bit está a 1 es que el contenido del campo *Datos* es válido y se procede a la lectura del valor.

Si se trata de un operando destino al que hay que renombrar, se busca la primera entrada libre en el RRF, se copia el identificador del ARF en el campo *Destino* y se colocan el bit de *Válido* a 0 y el bit de *Último* a 1. Además, es necesario comprobar que el operando destino no se encuentre ya renombrado en el RRF. En caso afirmativo, hay que colocar el bit de *Último* a 0 en la entrada del RRF que contenga el último renombramiento vigente que haya del registro ARF.

La actualización de los registros del ARF se produciría desde el RRF en el momento en que las instrucciones terminasen. Observe que el esquema de funcionamiento que se obtendría sería muy similar al sistema basado en registros de futuro, donde la función del ARF es la de permitir la recuperación de los valores correctos de los registros en caso de interrupciones.

Actividad 3.1 *Autor: José Sánchez Moreno*

Considere un procesador VLIW con un formato de instrucción que permite emitir simultáneamente operaciones a 2 unidades funcionales para el acceso a memoria (2 ciclos de latencia), a 2 unidades funcionales para operaciones en coma flotante (3 ciclos de latencia) y a una unidad funcional para operaciones enteras y de salto (1 ciclo de latencia donde el salto tiene un hueco de retardo de 1 ciclo). Dado el siguiente fragmento de código intermedio:

```
inicio: LD     F2,0(R1)
        MULTD  F2,F2,F0
        LD     F4,0(R2)
        ADDD   F4,F2,F4
        SD     0(R2),F4
        SUBI   R1,R1,#8
        SUBI   R2,R2,#8
        BNEZ   R1,inicio
```

- Desenrolle la secuencia cuatro veces y planifique el bucle desenrolladlo agrupando las instrucciones por su tipo.
- Planifique el bucle desenrollado en el apartado anterior en forma de instrucciones VLIW teniendo en cuenta las características del procesador para evitar cualquier detención del cauce.
- Si el tamaño de una instrucción de código intermedio es 4 bytes, calcule el tamaño del código VLIW resultante y el espacio de almacenamiento que se desaprovecha.
- Considerando que el bucle original y el desenrollado y planificado se ejecuta sin detenciones, esto es, un ciclo por instrucción, calcule los ciclos consumidos al ejecutar 1000 veces el código original, al desenrollado planificado y el VLIW.

En el enunciado se indica que sólo hay una ALU

Por otra parte, con el mismo desenrrollamietnto, yo he realizado el agrupamiento manteniendo el orden del bucle, y con el diagrama de flujo de datos (RAW), que ayuda bastante (un poco simplificado):
Si las ordenamos por tipo se obtiene la siguiente secuencia de código:

```
inicio:  LD F2, 0(R1)          //Iteración i
         MULTD F2, F2, F0
         LD F4, 0(R2)
         ADDD F4, F2, F4
         SD 0(R2), F4
         LD F6, -8(R1)         //Iteración i+1
         MULTD F6, F6, F0
         LD F8, -8(R2)
         ADDD F8, F6, F8
         SD -8(R2), F8
         LD F10, -16(R1)       //Iteración i+2
         MULTD F10, F10, F0
         LD F12, -16(R2)
         ADDD F12, F10, F12
         SD -16(R2), F12
         LD F14, -24(R1)       //Iteración i+3
         MULTD F14, F14, F0
         LD F16, -24(R2)
         ADDD F16, F14, F16
         SD -24(R2), F16
         SUBI R1, R1, #32
         SUBI R2, R2, #32
         BNEZ R1, inicio
```

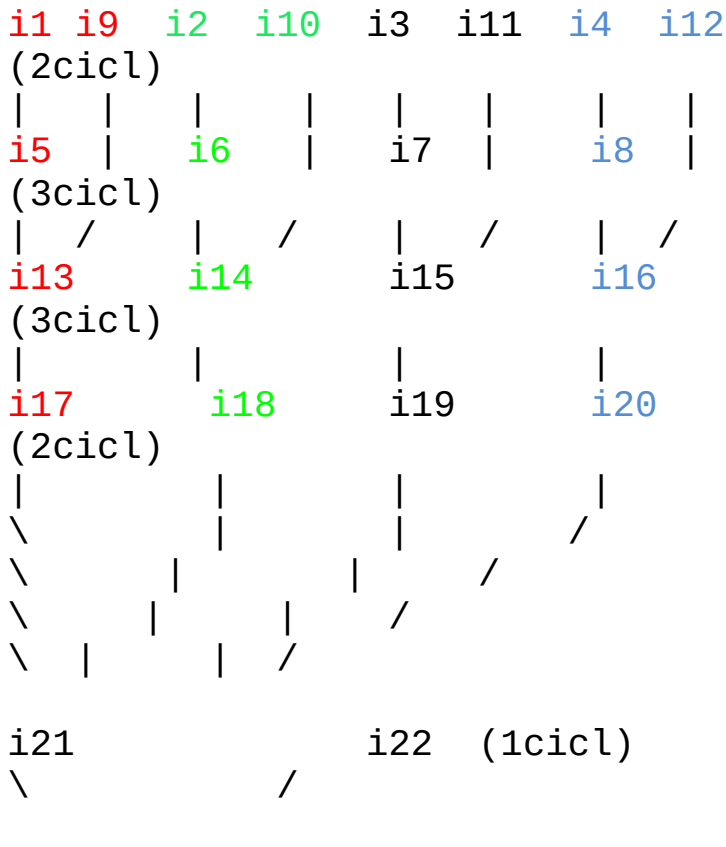
```
inicio:  LD F2, 0(R1)          //Iteración i
         LD F4, 0(R2)
         LD F6, -8(R1)
         LD F8, -8(R2)
         LD F10, -16(R1)
         LD F12, -16(R2)
         LD F14, -24(R1)
         LD F16, -24(R2)
         MULTD F2, F2, F0
         MULTD F6, F6, F0
         MULTD F10, F10, F0
         MULTD F14, F14, F0
         ADDD F4, F2, F4
         ADDD F8, F6, F8
         ADDD F12, F10, F12
         ADDD F16, F14, F16
         SD 0(R2), F4
         SD -8(R2), F8
         SD -16(R2), F12
         SD -24(R2), F16
         SUBI R1, R1, #32
         SUBI R2, R2, #32
         BNEZ R1, inicio
```

Bucle

```
inicio:
i1  LD f2, 0(r1)
i2  LD f6, -8(r1)
i3  LD f10, -16(r1)
i4  LD f14, -24(r1)
i5  MULTD f2, f2,f0
i6  MULTD f6, f6,f0
i7  MULTD f10, f10,f0
i8  MULTD f14, f14,f0
i9  LD f4, 0(r2)
i10 LD f8, -8(r2)
i11 LD f12, -16(r2)
i12 LD f16, -24(r2)
i13 ADDD f4, f2,f4
i14 ADDD f8, f6,f8
i15 ADDD f12, f10,f12
i16 ADDD f16, f14,f16
i17 SD 0(r2), f4
i18 SD -8(r2), f8
i19 SD -16(r2), f12
i20 SD -24(r2), f16

i21 SUBI r1, r1, #32
i22 SUBI r2, r2, #32

i23 BNEZ r1, inicio
```



Y por tanto, el código VLIW resultante (teniendo en cuenta la segmentación y los RAW con los ciclos de espera):

VLIW

Nº	LD/SD1	LD/SD2	FPU1	FPU2	ALU
1	i1 LD f2,0(r1)	i2 LD f6,-8(r1)			
2	i3 LD f10,-16(r1)	i4 LD f14,-24(r1)			
3	i9 LD f4, 0(r2)	i10 LD f8,-8(r2)	i5 MULTD f2,f2,f0	i6 MULTD f6,f6,f0	
4	i11 LD f12,-16(r2)	i12 LDf16,-24(r2)	i7 MULTD f10,f10,f0	i8 MULTD f14,f14,f0	
5			--------	--------	**i21 SUBI r1,r1,#32**
6			i13 ADDD f4,f2,f4	i14 ADDD f8,f6,f8	**i22 SUBI r2,r2,#32**
7			i15 ADDD f12,f10,f12	i16 ADDD f16,f14,f16	
8			--------	--------	
9	i17 SD 0(r2),f4 [32(r2),f4]	i18 SD -8(r2),f8 [24(r2),f8]			
10	i19 SD -16(r2),f12 [16(r2),f12]	i20 SD -24(r2),f16 [8(r2),f16]			
11					i21 SUBI r1,r1,#32
12					i22 SUBI r2,r2,#32
13					i23 BENZ r1,inicio

La reducción a 11 instrucciones se consigue con la estrategia indicada en el libro de adelantar el decremento de los índices, pasando i21 e i22 a huecos en la ALU después de las cargas, por ejemplo las VLIW Nº 5 y 6, y luego actualizando en los almacenamientos (i17-i20) el desplazamiento sobre r2 (lo pongo en azul en la tabla)

c)

Tamaño código original: 8 instrucciones * 4 bytes = 32 bytes
Tamaño código desenrollado: 23 instrucciones * 4 bytes = 92 bytes
Tamaño código VLIW: 11 instrucciones * 20 bytes = 220 bytes
Espacio desaprovechado en código VLIW: 32 slots * 4 bytes = 128 bytes

d)

Ciclos código original: 1000 iteraciones * 8 ciclos = 8000 ciclos
Ciclos código desenrollado: 250 iteraciones * 23 ciclos = 5750 ciclos
Ciclos código VLIW: 250 iteraciones * 11 ciclos = 2750 ciclos

Considere el siguiente bucle:

```
inicio: LD     F0,0(R1)
        ADDD   F4,F0,F2
        SD     0(R1),F4
        SUBI   R1,R1,#8
        BNEZ   R1,inicio
```

- Genere el código intermedio aplicando segmentación software pero evite la utilización de desplazamientos negativos en los accesos a memoria del patrón y considere que la latencia de las instrucciones es de 1 ciclo.
- Utilizando el código obtenido en el apartado anterior, escriba el código VLIW teniendo en cuenta que el formato de instrucción admite una operación de carga/almacenamiento (2 ciclos de latencia), una operación de coma flotante (3 ciclos de latencia) y una operación entera/salto (1 ciclo de latencia y el salto consume un hueco de retardo).
- ¿Por qué si se aplica la técnica de segmentación software al bucle original sin considerar las latencias verdaderas no se aprovecha realmente la segmentación hardware de las unidades funcionales?

Apartado a)

```
        LD    F0,0(R1)  // Carga de X[i]
        ADDD  F4,F0,F2  // X[i] = X[i] + a
        LD    F0,-8(R1) // Carga de X[i-1]
        SUBI  R1,R1,#16
inicio: SD    16(R1),F4 // Almacenamiento de X[i]
        ADDD  F4,F0,F2  // X[i-1] = X[i-1] + a
        LD    F0,0(R1)  // Carga de X[i-2]
        SUBI  R1,R1,#8
        BNEZ  R1,inicio // ¿Fin del bucle: R1=0?
        SD    16(R1),F4 // Almacenamiento de X[2]
        ADDD  F4,F0,F2  // X[1] = X[1] + a
        SD    8(R1),F4  // Almacenamiento de X[1]
```

Apartado b) El código VLIW sería el siguiente:

	Memoria	Coma Flotante	Entero/Salto
prólogo:	`LD F0,0(R1)`	-	-
	`LD F0,-8(R1)`	-	-
	-	`ADDD F4,F0,F2`	`SUBI R1,R1,#16`
	-	-	-
	-	-	-
inicio:	`SD 16(R1),F4`	`ADDD F4,F0,F2`	`SUBI R1,R1,#8`
	`LD F0,-8(R1)`	-	`BNEZ R1,inicio`
	-	-	-
epílogo:	`SD 16(R1),F4`	`ADDD F4,F0,F2`	-
	-	-	-
	-	-	-
	`SD 8(R1),F4`	-	-

El total de ciclos ejecutados para procesar un vector de 1000 elementos sería de 3009 ciclos (5+1000*3+ 4).

Apartado c) No se aprovechan correctamente los cauces de las unidades funcionales ya que en un momento dado solo hay una operación en cada unidad funcional: solo se admite un dato en las unidades funcionales, no tantos como segmentos tienen. Ello es debido a que si no se consideran las latencias de las unidades, la técnica no se aplica de forma óptima.
El patrón correcto se obtiene reflejando las latencias de las unidades:

LD F0,0(R1)	-	-	-	-	-
-	LD F0,-8(R1)	-	-	-	-
ADDD F4,F0,F2	-	LD F0,-16(R1)	-	-	-
-	ADDD F4,F0,F2	-	LD F0,-24(R1)	-	-
-	-	ADDD F4,F0,F2	-	LD F0,-32(R1)	-
SD 0(R1),F4	-	-	ADDD F4,F0,F2	-	LD F0,-40(R1)
-	SD -8(R1),F4	-	-	ADDD F4,F0,F2	-
-	-	SD -16(R1),F4	-	-	ADDD F4,F0,F2
-	-	-	SD -24(R1),F4	-	-
-	-	-	-	SD -32(R1),F4	-
-	-	-	-	-	SD -40(R1),F4
-	-	-	-	-	

Si se transforma la secuencia anterior en instrucciones VLIW genéricas, se obtiene el siguiente fragmento de código:

```
LD F0,0(R1)
LD F0,-8(R1)
LD F0,-16(R1)   ADDD F4, F0, F2
LD F0,-24(R1)   ADDD F4, F0, F2
LD F0,-32(R1)   ADDD F4, F0, F2
SD 0(R1),F4     ADDD F4, F0, F2 LD F0, -40(R1)
SD -8(R1),F4    ADDD F4, F0, F2
SD -16(R1),F4   ADDD F4, F0, F2
SD -24(R1),F4
SD -32(R1),F4
SD -40(R1),F4
```

Observe que si el formato de la instrucción VLIW no se ajustase al patrón, por ejemplo, por disponer solo de una operación de acceso a memoria, habría problemas ya que el patrón habría que descomponerlo en dos instrucciones VLIW al contar con una operación de carga y otra de almacenamiento. Esto produciría la pérdida de datos al romperse la secuencialidad en el flujo de datos:

```
LD F0,0(R1)
LD F0,-8(R1)
LD F0,-16(R1)   ADDD F4,F0,F2
LD F0,-24(R1)   ADDD F4,F0,F2   // Se pierde su resultado
LD F0,-32(R1)   ADDD F4,F0,F2
SD 0(R1),F4     ADDD F4,F0,F2
LD F0,-40(R1)                   // Nadie lee el valor de F4
SD -8(R1),F4    ADDD F4,F0,F2   // ya que se machaca el valor previo de F4
SD -16(R1),F4   ADDD F4,F0,F2
SD -24(R1),F4
SD -32(R1),F4
SD -40(R1),F4
```

Dado el siguiente fragmento de código:

```
inicio: LD     F2,0(R1)
        MULTD  F4,F2,F10
        LD     F6,0(R2)
        ADDD   F8,F6,F4
        SD     0(R2),F8
        SUBI   R1,R1,#8
        SUBI   R2,R2,#8
        BNEZ   R1,inicio
```

Donde **F6** y **F10** contienen valores previamente cargados. Se pide que:

- Aplique segmentación software y obtenga el patrón de comportamiento considerando que los accesos a memoria consumen dos ciclos y las operaciones de coma flotante tres ciclos.
- Genere el correspondiente pseudocódigo VLIW para un procesador VLIW genérico en el que la unidad de salto consume un ciclo.

a) La siguiente tabla muestra el patrón de ejecución que se ha obtenido al realizar 9 iteraciones del bucle y considerando las latencias de las unidades funcionales:

LD F2, 0(R1) LD F6, 0(R2)								
	LD F2, -8(R1) LD F6, -8(R2)							
MULTD F4, F2, F10		LD F2, -16(R1) LD F6, -16(R2)						
	MULTD F4, F2, F10		LD F2, -24(R1) LD F6, -24(R2)					
		MULTD F4, F2, F10		LD F2, -32(R1) LD F6, -32(R2)				
ADDD F8, F6, F4			MULTD F4, F2, F10		LD F2, -40(R2) LD F6, -40(R2)			
	ADDD F8, F6, F4			MULTD F4, F2, F10		LD F2, -48(R2) LD F6, -48(R2)		
		ADDD F8, F6, F4			MULTD F4, F2, F10		LD F2, -56(R2) LD F6, -56(R2)	
SD 0(R2), F8			**ADDD F8, F6, F4**			**MULTD F4, F2, F10**		**LD F2, -64(R2) LD F6, -64(R2)**
	SD -8(R2), F8			ADDD F8, F6, F4			MULTD F4, F2, F10	
		SD -16(R2), F8			ADDD F8, F6, F4			MULTD F4, F2, F10
			SD -24(R2), F8			ADDD F8, F6, F4		
				SD -32(R2), F8			ADDD F8, F6, F4	
					SD -40(R2), F8			ADDD F8, F6, F4
						SD -48(R2), F8		
							SD -56(R2), F8	
								SD -64(R2), F8

b) El pseudocódigo VLIW derivado del anterior patrón de comportamiento es el siguiente:

	LD F2, 0(R1)	LD F6, 0(R2)				
	LD F2, -8(R1)	LD F6, -8(R2)				
	LD F2, -16(R1)	LD F6, -16(R2)	MULTD F4, F2, F10			
	LD F2, -24(R1)	LD F6, -24(R2)	MULTD F4, F2, F10			
	LD F2, -32(R1)	LD F6, -32(R2)	MULTD F4, F2, F10			
	LD F2, -40(R1)	LD F6, -40(R2)	MULTD F4, F2, F10	ADDD F8, F6, F4		
	LD F2, -48(R1)	LD F6, -48(R2)	MULTD F4, F2, F10	ADDD F8, F6, F4		
	LD F2, -56(R1)	LD F6, -56(R2)	MULTD F4, F2, F10	ADDD F8, F6, F4		
Inicio:	**LD F2, -64(R1)**	**LD F6, -64(R2)**	**MULTD F4, F2, F10**	**ADDD F8, F6, F4**	**SD 0(R2), F8**	**if (R1<>72) {R1:=R1-8; R2:=R2-8; goto inicio}**
	SD -8(R2), F8	ADDD F8, F6, F4	MULTD F4, F2, F10			
	SD -16(R2), F8	ADDD F8, F6, F4	MULTD F4, F2, F10			
	SD -24(R2), F8	ADDD F8, F6, F4				
	SD -32(R2), F8	ADDD F8, F6, F4				
	SD -40(R2), F8	ADDD F8, F6, F4				
	SD -48(R2), F8					
	SD -56(R2), F8					
	SD -64(R2), F8					

Observe que si se trata de transformar el pseudocódigo en instrucciones que admiten una o dos operaciones de acceso a memoria surgen problemas. El cuerpo del bucle segmentado se debe descomponer en dos o tres instrucciones VLIW lo que rompe la secuencialidad en el flujo de datos que se obtiene al aplicar segmentación software en el patrón. Al partir el cuerpo del bucle en dos o tres instrucciones VLIW, las unidades de coma flotante producen resultados que nadie recoge en las primeras iteraciones ya que sus segmentaciones estaban cargadas con tantas operaciones consecutivas como profundas son sus cauces. Al tener que añadir instrucciones VLIW con slots de operación vacíos, los resultados que quedan en las etapas interiores de su segmentación machacan los resultados previos, ya que nadie los recoge.

Actividad 3.4 Autor: José Sánchez Moreno

Un procesador VLIW cuenta con un formato de instrucción que admite una operación de carga/almacenamiento (2 ciclos de latencia), una unidad de coma flotante (3 ciclos de latencia) y una unidad para operaciones enteras y de salto (1 ciclo de latencia y un hueco de retardo). Aplique segmentación software al siguiente bucle y genere el correspondiente código VLIW:

```
inicio: LD     F0,0(R1)
        ADDD   F4,F0,F2
        SD     0(R1),F4
        SUBI   R1,R1,#8
        BNEZ   R1,inicio
```

LD F0, 0(R1)	-	-	-	-	-
-	LD F0, -8(R1)	-	-	-	-
ADDD F4, F0, F2	-	LD F0, -16(R1)	-	-	-
-	ADDD F4, F0, F2	-	LD F0, -24(R1)	-	-
-	-	ADDD F4, F0, F2	-	LD F0, -32(R1)	-
SD 0(R1), F4	-	-	**ADDD F4, F0, F2**	-	**LD F0, -40(R1)**
-	SD -8(R1), F4	-	-	ADDD F4, F0, F2	-
-	-	SD -16(R1), F4	-	-	ADDD F4, F0, F2
-	-	-	SD -24(R1), F4	-	-
-	-	-	-	SD -32(R1), F4	-
-	-	-	-	-	SD -40(R1), F4
-		-	-	-	-

El principal problema que surge es que el formato de instrucción VLIW dispone de únicamente un campo para operaciones de acceso a memoria y el cuerpo del bucle consta de una carga y un almacenamiento. Ello obliga a generar varias instrucciones VLIW para el cuerpo del bucle segmentado con el riesgo de pérdidas de datos. Una posible solución es intentar generar un patrón que responda al formato de instrucción VLIW, esto, es, una operación de acceso a memoria, una de coma flotante y una entera. La siguiente tabla presenta una solución.

LD F0, 0(R1)	-	-	-
-	-	-	-
ADDD F4, F0, F2	LD F0, -8(R1)	-	-
-	-	-	-
-	ADDD F4, F0, F2	LD F0, -16(R1)	-
SD 0(R1), F4	-	-	-
-	-	**ADDD F4, F0, F2**	**LD F0, -24(R1)**
-	SD -8(R1), F4	-	-
-	-	-	ADDD F4, F0, F2
-	-	SD -16(R1), F4	-
-	-	-	-
-	-	-	SD -24(R1), F4

Una posible transformación de la secuencia previa en instrucciones VLIW se presenta a continuación. Preste atención al juego que se tiene que hacer con los valores del desplazamiento para aprovechar el hueco de retardo del salto.

	Carga/almacenamiento	Coma flotante	Entera
	LD F0, 0(R1)		
	LD F0, -8(R1)	ADDD F4, F0, F2	
	LD F0, -16(R1)	ADDD F4, F0, F2	SUBI R1, R1, #32
Inicio:	**SD 32(R1), F4**	**-**	**BNEZ R1, inicio**
	LD F0, 8(R1)	**ADDD F4, F0, F2**	**SUBI R1, R1, #8**
	SD 32(R1), F4		
		ADDD F4, F0, F2	
	SD 24(R1), F4		
	SD 16(R1), F4		

Dado el siguiente bucle escrito en pseudocódigo:

```
for (i=0; i<n; i++)
   if (A[i]==0) then
      X[i]:= X[i]+a;
   else
      Y[i]:= Y[i]-a;
   end if;
end for;
```

- Escriba el código intermedio genérico.
- Escriba el código intermedio aplicando la técnica ***if – conversion***.
- Escriba el código VLIW de los apartados anteriores teniendo en cuenta que el formato de instrucción admite una operación de carga/almacenamiento (2 ciclos de latencia), una operación de coma flotante (3 ciclos de latencia) y una operación entera/salto (1 ciclo de latencia y el salto consume un hueco de retardo). Compare los tiempos de ejecución para procesar en vector de 1000 elementos suponiendo que la probabilidad de ambas ramas del **`if`** es la misma.

a) Un posible código intermedio genérico correspondiente al bucle es el siguiente:

```
inicio: LD R5, 0(R1)        // Cargar A[i]
        BNEZ R5, else       // Si (A[i] <> 0) saltar a else
then:   LD F4, 0(R2)        // Cargar X[i]
        ADDD F4, F4, F2     // X[i] = X[i] + a
        SD 0(R2), F4        // Almacenar X[i]
        JMP final
else:   LD F4, 0(R3)        // Cargar Y[i]
        SUBD F4, F4, F2     // Y[i] = Y[i] - a
        SD 0(R3), F4        // Almacenar Y[i]
final:  SUBI R2, R2, #8     // Decrementar índice de X en 8 bytes
        SUBI R3, R3, #8     // Decrementar índice de Y en 8 bytes
        SUBI R1, R1, #4     // Decrementar índice de A en 4 bytes
        BNEZ R1, inicio     // Comenzar una nueva iteracción
```

b)

```
inicio: LD R5, 0(R1)
        PRED_EQ p1, p2, R5, #0
        LD F4, 0(R2)  (p1)
        ADDD F4, F4, F2  (p1)
        SD 0(R2), F4 (p1)
        LD F4, 0(R3)   (p2)
        SUBD F4, F4, F2  (p2)
        SD 0(R3), F4 (p2)
final:  SUBI R2, R2, #8
        SUBI R3, R3, #8
        SUBI R1, R1, #4
        BNEZ R1, inicio
```

c) Un posible código VLIW con optimizaciones derivado del código intermedio del apartado (a) sería el siguiente:

	Carga/almacenamiento	Coma flotante	Entera/salto
Inicio:	LD R5, 0(R1)		SUBI R1, R1, #4
Then:	LD F4, 0(R2)		
			BNEZ R5, else
		ADDD F4, F4, F2	
	SD 0(R2), F4		JMP final
Else:	LD F4, 0(R3)		SUBI R2, R2, #8
		SUBD F4, F4, F2	
	SD 0(R3), F4		
Final:			BNEZ R1, inicio
			SUBI R3, R3, #8

Observe que se ha realizado una carga especulativa en la rama then ya que la instrucción de carga LD F4, 0(R2) se ejecuta con independencia del resultado de la comparación. De esta forma, si el salto es efectivo ya se habrá iniciado el procesamiento de la carga y si no es efectivo, el valor almacenado en F4 quedará anulado al iniciarse la ejecución de la instrucción de carga correspondiente a la rama else, esto es, LD F4, 0(R3).

Ciclos consumidos al ejecutar la rama then: 10 ciclos
Ciclos consumidos al ejecutar la rama else:12 ciclos
Ciclos consumidos ejecutar bucle de 1000 elementos: 10*500+12*500=11000 ciclos
Tamaño del código VLIW: 15 instrucciones * 12 bytes = 180 bytes
Porción de código VLIW vacío: 32 slots * 4 bytes = 128 bytes

Y para el apartado (b), un posible código VLIW sería:

	Carga/almacenamiento	**Coma flotante**	**Entera/salto**
Inicio:	LD R5, 0(R1)		
			PRED_EQ p1, p2, R5, #0
	LD F4, 0(R2) (p1)		
	LD F4, 0(R3) (p2)		
		ADDD F4, F4, F2 (p1)	
		SUBD F4, F4, F2 (p2)	
			SUBI R2, R2, #8
	SD 0(R2), F4 (p1)		SUBI R1, R1, #4
	SD 0(R3), F4 (p2)		BNEZ R1, inicio
			SUBI R3, R3, #8

Ciclos consumidos ejecutar bucle de 1000 elementos: 11*1000=11000 ciclos
Tamaño del código VLIW: 11 instrucciones * 12 bytes = 132 bytes
Porción de código VLIW vacío: 21 slots * 4 bytes = 84 bytes

Actividad 3.6 *Autor: José Sánchez Moreno*

Dado el siguiente bucle escrito en pseudocódigo:

```
for (i=0; i<n; i++)
   if (A[i]==0) then
      X[i]:= X[i]+a;
   else
      Y[i]:= Y[i]-a;
   end if;
end for;
```

- Generar el código VLIW aplicando predicación y realizando la carga especulativa de los datos.

	Carga/almacenamiento	Coma Flotante	Entera/Salto
inicio:	**LD R5,0(R1)**		
	LD F4,0(R2)		
	LD F6,0(R3)		**PRED EQ p1,p2,R5,#0**
		ADDD F4,F4,F2 (p1)	
		SUBD F6,F6,F2 (p2)	
			SUBI R2,R2,#8
	SD 8(R2),F4(p1)		**SUBI R1,R1,#4**
	SD 0(R3),F6(p2)		**BNEZ R1,inicio**
			SUBI R3,R3,#8

Ciclos consumidos ejecutar bucle de 1000 elementos: 9*1000 = 9000 ciclos
Tamaño del código VLIW: 9 instrucciones * 12 bytes = 108 bytes
Porción de código VLIW vacío: 15 slots * 4 bytes = 60 bytes

Dado el siguiente bucle escrito en pseudocódigo:

```
if (a>0) then
    a:= a+a;
else
   if (b<0) then
      b:= b+b;
   else
      c:= c+c;
   end if;
   a:= b;
end if;
```

- Escriba el código intermedio utilizando instrucciones condicionales con el fin de reducir al mínimo las sentencias de salto condicional. Las direcciones de memoria de las variables enteras a, b y c se encuentran almacenadas en `M[R5], M[R6]` y `M[R7]`, respectivamente.

```
LD R1,0(R5)              // carga de a
LD R2,0(R6)              // carga de b
LD R3,0(R7)              // carga de c
PRED_GT p1,p2,R1,#1      // If (a>0)
PRED_LT p3,p4,R2,#-1(p2) // If (b<0)
ADDD R1, R1, R1(p1)      // a:=a+a
ADDD R2,R2,R2 p3)        // b:=b+b
ADDD R3,R3,R3 p4)        // c:=c+c
SD 0(R7),R3(p4)          // almacenamiento de c:=c+c
SD 0(R6),R2(p3)          // almacenamiento de b:=b+b
SD 0(R5),R2(p2)          // almacenamiento de a:=b
SD 0(R5),R1(p1)          // almacenamiento de a:=a+a
```

Dado el siguiente fragmento de pseudocódigo:

```
if (a==0) then
    a:= b;
else
    a:= a+1;
end if;
```

- Y asumiendo que la rama **then** es la que tiene mayor probabilidad de ser ejecutada, genere el código intermedio aplicando planificación de trazas

```
        LD R1, 0(R5) // R1 <= a
        BNEZ R1, else
        LD R1, 0(R6) // R1 <= b
        JMP final
else:   ADD R1, R1, #1 // R1 <= R1+1
final:  SD 0(R5), R1    // a <= R1
```

y el adelantamiento de la carga ubicada en la rama del **if** con mayor probabilidad de ser ejecutada produciría el siguiente código (he utilizado una instrucción BEZ para optimizar más la secuencia):

```
        LD R1, 0(R5) // R1 <= a
        LD R2, 0(R6) // R2 <= b
        BEZ R1, final
        ADD R2, R1, #1 // R2 <= R1+1
final:  SD 0(R5), R2 // a <= R2
```

Dado los dos ejemplos de código de la figura 3.17, intente reducir el número de ciclos de ejecución mediante una adecuada reorganización de las instrucciones.

	Operaciones Carga / Almacenamiento	Operaciones Coma flotante	Operaciones Enteras y Saltos	
inicio:	LD R1, 0(R2)	------	------	1
	------	------	------	2
	------	------	SUBI R3, R1, #50	3
	------	------	BNEZ R3, else	4
	------	------	------	5
then:	------	------	ADDI R5, R5, #2	6
	------	------	JMP final	7
	------	------	------	8
else:	------	------	ADDI R5, R5, #1	9
final:	------	------	SUBI R2, R2, #4	10
	------	------	BNEZ R2, inicio	11
	------	------	------	12

Código figura 3.17a

	Carga/almacenamiento	Coma flotante	Enteras/salto	
inicio:	**LD R1,0(R2)**			1
				2
			SUBI R2,R2,#8	3
			BNEZ R1,inicio	4
			SUBI R2,R2,#8	5
then:			**JUMP final**	6
			ADDI R5,R5,#2	7
else:			**ADDI R5,R5,#1**	8
final:			**BNEZ R1,inicio**	9

inicio:	**LD R1,0(R2)**	------	------	1
	------	------	------	2
	------	------	**PRED_EQ P1,P2,R1,#50**	3
	------	------	**ADDI R5,R5,#2(P1)**	4
	------	------	**ADDI R5,R5,#1(P2)**	5
	------	------	**SUBI R2,R2,#8**	6
	------	------	**BNEZ R2,inicio**	7
	------	------	------	8

Código figura 3.17b

	Carga/almacenamiento	Coma flotante	Enteras/salto	
inicio:	**LD R1,0(R2)**		**SUBI R2,R2,#8**	1
				2
			PRED_EQ P1,P2,R1,#50	3
			ADDI R5,R5,#2(P1)	4
				5
			BNEZ R2,inicio	6
			ADDI R5,R5,#1(P2)	7

Actividad 3.11 Autor: José Sánchez Moreno

En un procesador vectorial con las siguientes características:

- Longitud vectorial máxima 64 elementos.
- Una unidad de ***suma vectorial*** con tiempo de arranque de 6 ciclos.
- Una unidad de ***multiplicación*** con tiempo de arranque de 7 ciclos.
- Una unidad de ***carga/almacenamiento vectorial*** con tiempo de arranque de 12 ciclos.
- La frecuencia del trabajo del procesador es 500 MHz.

Se pretende ejecutar el siguiente fragmento de código vectorial:

```
LV      V1,R1
MULTSV  V2,V1,F1
ADDSV   V3,V1,F1
SV      R2,V2
SV      R3,V3
```

Calcule T_{64}, $T_{elemento}$ y R_{64} bajo las siguientes condiciones:

- Sin encadenamiento entre unidades.
- Con encadenamiento entre unidades.
- Con encadenamiento y tres unidades de carga/almacenamiento.

a) Del análisis de las dependencias estructurales y de datos que existen en el fragmento de código vectorial se obtendrían cuatro convoyes:

```
Convoy 1:   LV      V1, R1
Convoy 2:   MULTSV  V2, V1, F1      // Dependencia de datos RAW
            ADDSV   V3, V1, F1
Convoy 3:   SV      R2, V2          // Dependencia de datos RAW
Convoy 4:   SV      R3, V3          // Dependencia estructural
```

La siguiente tabla ilustra cómo se obtiene el tiempo total de ejecución de la secuencia de código vectorial para vectores de longitud 64

Convoy	Comienza	Termina	Comentario
LV V1, R1	0	12+64=76	Latencia sencilla
MULTSV V2, V1, F1 ADDSV V3, V1, F1	76	76+7+64=147	Espera por convoy 1
SV R2, V2	147	147+12+64=223	Espera por convoy 2
SV R3, V3	223	223+12+64=299	Espera por convoy 3

$$T_{64} = (12+64) + (7+64) + (12+64) + (12+64) = 299 \text{ ciclos}$$

$$T_{64} = \frac{299 \text{ ciclos}}{500 MHz}$$

$$T_{64} = 0{,}598 \mu seg.$$

El tiempo por elemento es equivalente al número de convoyes, es decir, 4 ciclos u 8 nseg. En lo que respecta al rendimiento que se alcanza:

$$R_{64} = \frac{64 \text{ elementos} * 2 \text{ FLOP}}{299 \text{ ciclos}}$$

$$R_{64} = 0{,}42 \text{ FLOP/ciclo}$$

que expresado en FLOPS sería:

$$R_{64} = 0{,}42 \text{ FLOP/ciclo} * 500 \text{ MHz}$$

$$R_{64} = 214 \text{ MFLOPS}$$

b) Ahora ya se permite realizar el encadenamiento de resultados entre unidades funcionales. Por lo tanto, las instrucciones de multiplicación y suma del segundo convoy pueden fusionarse con la carga del primer convoy reduciendo el número total de convoyes a tres:

```
Convoy 1:   LV      V1, R1
            MULTSV  V2, V1, F1      // Encadenada a LV
            ADDSV   V3, V1, F1      // Encadenada a LV
Convoy 2:   SV      R2, V2          // Dependencia estructural y de datos RAW
Convoy 3:   SV      R3, V3          // Dependencia estructural
```

Se tendría ahora la siguiente secuencia temporal de ejecución:

Convoy	Comienza	Termina	Comentario
LV V1, R1 MULTSV V2, V1, F1 ADDSV V3, V1, F1	0	12+7+64=83	Latencia sencilla
SV R2, V2	83	83+12+64=159	Espera por convoy 1
SV R3, V3	159	159+12+64=235	Espera por convoy 2

$$T_{64} = (12 + 7 + 64) + (12 + 64) + (12 + 64) = 235 \text{ ciclos}$$

$$T_{64} = \frac{235 \text{ ciclos}}{500 MHz}$$

$$T_{64} = 0{,}47 \mu seg$$

El tiempo por elemento es equivalente al número de convoyes, es decir, 3 ciclos ó 6 nseg. En lo que respecta al rendimiento que se alcanza:

$$R_{64} = \frac{64 \text{ elementos} * 2 \text{ FLOP}}{235 \text{ ciclos}}$$

$$R_{64} = 0{,}5447 \text{ FLOP/ciclo}$$

que expresado en FLOPS sería:

$$R_{64} = 0{,}5447 \text{ FLOP/ciclo} * 500 \text{ MHz}$$

$$R_{64} = 272 \text{ MFLOPS}$$

c) Dado que ahora se dispone de tres unidades de carga/almacenamiento y el encadenamiento es posible, el número de convoyes se puede reducir a uno:

```
Convoy 1:   LV      V1, R1
            MULTV   V2, V1, F1    // Encadenada a LV
            ADDV    V3, V1, F1    // Encadenada a LV
            SV      R2, V2        // Encadenada a MULTV
            SV      R3, V3        // Encadenada a ADDV
```

La secuencia temporal de ejecución sería la siguiente:

Convoy	Comienza	Termina	Comentario
LV V1, R1 MULTSV V2, V1, F1 ADDSV V3, V1, F1 SV R2, V2 SV R2, V3	0	12+7+12+64=95	Latencia sencilla

$$T_{64} = (12 + 7 + 12 + 64) = 95 \text{ ciclos}$$

$$T_{64} = \frac{95 \text{ ciclos}}{500 MHz}$$

$$T_{64} = 0{,}19 \mu seg$$

El tiempo por elemento es equivalente al número de convoyes, en este caso 1 ciclo que equivale a 2 nseg. En lo que respecta al rendimiento que se alcanza:

$$R_{64} = \frac{64 \text{ elementos} * 2 \text{ FLOP}}{95 \text{ ciclos}}$$
$$R_{64} = 1{,}347 \text{ FLOP/ciclo}$$

que expresado en FLOPS sería:

$$R_{64} = 1{,}347 \text{ FLOP/ciclo} * 500 \text{ MHz}$$
$$R_{64} = 673 \text{ MFLOPS}$$

Actividad 3.12 *Autor: José Sánchez Moreno*

En un procesador vectorial con las siguientes características:

- Registros con una longitud vectorial máxima 64 elementos.
- Una unidad de ***suma vectorial*** con tiempo de arranque de 6 ciclos.
- Una unidad de ***multiplicación*** con tiempo de arranque de 7 ciclos.
- Una unidad de ***carga/almacenamiento vectorial*** con tiempo de arranque de 12 ciclos.
- La frecuencia del trabajo del procesador es 500 MHz.
- T_{base} de 10 ciclos y T_{bucle} de 15 ciclos.

Se pretende ejecutar el siguiente bucle:

```
for (i=1; i<n; i++)
   A[i]:= A[i]+B[i]);
   B[i]:= a*B[i];
end for;
```

- Escriba el código vectorial que realizaría las operaciones ubicadas en el interior del bucle considerando la posibilidad de encadenamiento de resultados.
- Calcule T_n, T_{1000} , R_{1000} y R_∞.
- Calcule T_n, T_{1000} , R_{1000} y R_∞ pero ahora considerando encadenamientos y dos unidades de carga/almacenamiento.
- ¿Qué ocurre si se considera encadenamientos, dos unidades de carga/almacenamiento y se permite el solapamiento entre convoyes?

Convoy 1: LV V2, R2 // Carga de B

Convoy 2: LV V1, R1 // Carga de A

MULTSV V3, F0, V2 // B := x * B

ADDV V1, V1, V2 // A := A+B

Convoy 3: SV R1, V1 // Almacenamiento de A

Convoy 4: SV R2, V3 // Almacenamiento de B

Ya he dicho que yo creo que el compilador elimina las falsas dependencias pero que no creo que haya problemas en esa solución. La misma solución utilizando otro registro sería válida sin alterar el orden de las operaciones. se supone que el valor de a se encuentra cargado ya que se trata de una operación escalar que no he contemplado.

	LD	F1,0(R3)
convoy1:	LV	V1,R1
convoy2:	LV	V2,R2
	MULTSV	V3,V2,F1
	ADDV	V1,V1,V2
convoy3:	SV	R1,V1
convoy4:	SV	R2,V3

ACTIVIDAD 3.12

a) Código Vectorial usando encadenamiento

```
            LD      F1,0(R3)
convoy1:    LV      V1,R1
convoy2:    LV      V2,R2
            MULTSV  V3,V2,F1
            ADDV    V1,V1,V2
convoy3:    SV      R1,V1
convoy4:    SV      R2,V3
```

b) Calcular $T_n,\ T_{1000}, R_{1000}\ y\ R_\infty$

$$T_{arranque} = \underbrace{12}_{C1} + \underbrace{(12+7)}_{C2} + \underbrace{12}_{C3} + \underbrace{12}_{C4} = 55\ ciclos$$

$$T_{elementos} = 4\ ciclos$$

$$T_n = T_{base} + \lceil \frac{n}{64} \rceil \cdot (T_{bucle} + T_{arranque}) + n \cdot T_{elementos}$$

$$T_n = 10 + \lceil \frac{n}{64} \rceil \cdot (15+55) + 4n = 10 + \lceil \frac{n}{64} \rceil \cdot 70 + 4n$$

$$T_{1000} = 10 + \lceil \frac{1000}{64} \rceil \cdot 70 + 4 \cdot 1000 = 10 + 16 \cdot 70 + 4000 = 5130\ ciclos$$

$$R_\infty = \lim_{n\to\infty} (\frac{2n}{T_n}) =$$
$$= \lim_{n\to\infty} (\frac{2n}{10 + (\frac{n}{64} + 1) \cdot 70 + 4n}) = \lim_{n\to\infty} (\frac{2n}{10 + \frac{70n}{64} + 70 + 4n}) =$$
$$= \lim_{n\to\infty} (\frac{2n}{(\frac{163n}{32} + 80}) \simeq 0.392\ FLOPS\ /\ ciclo$$

Luego

$R_\infty \simeq 0.385\ FLOPS\ /\ ciclo \cdot 500\ Mhz \simeq 192,31\ MFLOPS$

c) Calcular $T_n,\ T_{1000}, R_{1000}\ y\ R_\infty$ **con encadenamientos y 2 unidades de carga/almacenamiento**

```
            LD      F1,0(R3)
convoy1:    LV      V1,R1
            LV      V2,R2
            MULTSV  V3,V2,F1
            ADDV    V1,V1,V2
convoy2:    SV      R1,V1
            SV      R2,V3
```

$$T_{arranque} = \underbrace{(12+7)}_{C1} + \underbrace{(12)}_{C2} = 31\ ciclos$$

$$T_{elementos} = 2\ ciclos$$

$$T_n = 10 + \lceil \frac{n}{64} \rceil \cdot (15+31) + 2n = 10 + \lceil \frac{n}{64} \rceil \cdot 46 + 2n$$

7

$$T_{1000} = 10 + \lceil \frac{1000}{64} \rceil \cdot 46 + 2 \cdot 1000 = 10 + 16 \cdot 46 + 2000 = 2746\ ciclos$$

$$R_\infty = \lim_{n\to\infty} (\frac{2n}{T_n}) =$$

$$= \lim_{n\to\infty} (\frac{2n}{10 + (\frac{n}{64} + 1) \cdot 46 + 2n}) = \lim_{n\to\infty} (\frac{2n}{10 + (\frac{46n}{64} + 46 + 2n}) =$$

$$= \lim_{n\to\infty} (\frac{2n}{(\frac{87n}{32} + 56}) \simeq 0.392\ FLOPS\ /\ ciclo$$

Luego

$$R_\infty \simeq 0.392\ FLOPS\ /\ ciclo \cdot 500\ Mhz \simeq 367,81\ MFLOPS$$

d) Calcular T_n, T_{1000}, R_{1000} y R_∞ con encadenamientos y 2 unidades de carga/almacenamiento y solapamiento entre convoyes

He hecho el diagrama en papel y es similar al de la página 272, salvo que hay 2 instrucciones de almacenamiento que empiezan en el ciclo 64 y la suma y la multiplicación empiezan en el mismo ciclo, que es el 12. El resto de cálculos son iguales a los de la página 272 y siguientes, cambiando el tiempo de arranque por 31 ciclos, si no me he equivocado.

Actividad 3.13 *Autor: Oscar Moya Martínez*

Dispone de un procesador vectorial en el que el tiempo de acceso a los bancos de memoria es de 6 ciclos y el tiempo por elemento es de 1 ciclo. Los bancos de memoria tienen un ancho de una palabra de 8 bytes y se direccionan por bytes. En base a estos datos,

- ¿Cuál es el mínimo número de bancos de memoria que hay que utilizar para que el tiempo de acceso a la memoria permanezca oculto salvo en lo que respecta al primer acceso?
- Suponiendo que el procesador vectorial cuanta con el número mínimo de bancos de memoria, ¿Cuántos ciclos tardará en completarse la carga de un vector de longitud 64 con una separación entre elementos de 20 palabras?
- ¿Qué porcentaje de ancho de banda de memoria se conseguirá con una carga de 64 elementos si se mantiene una separación entre elementos de 1 en comparación con una separación de 20?

Si $T_a = 6\,ciclos$ y m es el número de bancos de memoria, para que el acceso a memoria permanezca oculto, se debe cumplir que $m \geq T_a$. Por lo que el número mínimo de bancos de memoria ha de ser $m = T_a$, es decir 6 bancos de memoria.

En primer lugar podríamos pensar en compactar la memoria, de manera que todos los elementos del vector queden consecutivos, pero esto no siempre es posible, ya que puede que hayan elementos intermedios en uso, y además esto consumiría muchos ciclos y es muy difícil(ni idea) de calcular, a parte no creo que el ejercicio vaya de esto.

En segundo lugar podríamos gastar 6 ciclos en acceder a cada palabra del vector, lo que daría:

$T_a \cdot n \cdot T_{elementos} = 6 \cdot 64 \cdot 1 = 384\ ciclos$, pero es bastante ineficiente.

Y en tercer lugar si dibujamos una tabla con la memoria, pensando que en el banco 0 y elemento 0 se encuentra la primera palabra del vector tendríamos:

Bancos: 0	1	2	3	4	5
0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	**21**	22	23
24	25	26	27	28	29
30	31	32	33	34	35
36	37	38	39	40	41
42	43	44	45	46	47

Cada 20 elementos tenemos una palabra del vector, que es lo que está en negrita
En este caso, en un sólo ciclo se podrían solicitar 2 palabras del vector y no 6 como en el caso de tener las palabras consecutivas.

Lo del cálculo podría ser 6/2=3 veces más lento de esta forma para acceder a la memoria. Por tanto:

$T_a + \left(n \cdot T_{elementos}\right) \cdot 3 = 6 + \left(64 \cdot 1\right) \cdot 3 = 198\ ciclos$

Si los elementos del vector son consecutivos se tiene que:

$$T_a + \left(n \cdot T_{elementos}\right) = 6 + \left(64 \cdot 1\right) = 70 \ ciclos$$

Y el porcentaje de ancho de banda que se ahorra sería:

$$\frac{70 - 198}{198} \cdot 100 \simeq -64,64$$

Se ahorra un 64,64 % de ancho de banda si se tienen las palabras del vector consecutivas respecto a tener las palabras separadas cada 20 elementos

Actividad 3.14 Autor: José Sánchez Moreno

En un procesador vectorial dotado de una unidad funcional de suma y otra de multiplicación se ejecuta el siguiente programa en C:

```
for (i=0; i<10000; i++){
    s[i]:= a[i]+b[i]);
    t[i]:= a[i]*s[i];
}
```

Las características del procesador vectorial son las siguientes:

- La latencia de la unidad de memoria es de 14 ciclos de reloj.
- El tiempo de arranque de la unidad de ***suma es*** de 8 ciclos de reloj.
- El tiempo de arranque de la unidad de ***multiplicación es*** de 10 ciclos de reloj.
- T_{bucle} = 25.
- T_{base} se considera que no existe.
- La frecuencia del reloj es 800 MHz.
- MVL = 64.

Se pide que:

- Escriba una secuencia de instrucciones vectoriales que sustituya a la parte interior del bucle del programa C. considere que las direcciones de `a, b, s` y `t` estás almacenadas respectivamente en `Ra, Rb, Rs` y `Rt`, respectivamente.
- Calcule el tiempo de ejecución T_{10000} y el rendimiento teórico máximo R_{∞} en MFLOPS bajo las siguientes precondiciones:
    - Sin encadenamiento.
    - Con encadenamiento.
    - Con encadenamiento y 2 unidades de carga/almacenamiento.

ACTIVIDAD 3.14

Código Vectorial:

```
LV      V1,Ra      ;Cargar a
LV      V2,Rb      ;Cargar b
ADDV    V3,V1,V2   ;s := a + b
MULTV   V4,V1,V3   ;t := a * s
SV      Rs,V3      ;Almacenar s
SV      Rt,V4      ;Almacenar t
```

Sin encadenamiento

```
convoy 1:  LV      V1,Ra      ;Cargar a
convoy 2:  LV      V2,Rb      ;Cargar b
convoy 3:  ADDV    V3,V1,V2   ;s := a + b
convoy 4:  MULTV   V4,V1,V3   ;t := a * s
           SV      Rs,V3      ;Almacenar s
convoy 5:  SV      Rt,V4      ;Almacenar t
```

$T_{arranque} = (14) + (14) + (8) + (14) + (14) = 64\ ciclos$

Con encadenamiento

```
convoy 1:  LV      V1,Ra      ;Cargar a
convoy 2:  LV      V2,Rb      ;Cargar b
convoy 3:  ADDV    V3,V1,V2   ;s := a + b
           MULTV   V4,V1,V3   ;t := a * s
           SV      Rs,V3      ;Almacenar s
convoy 4:  SV      Rt,V4      ;Almacenar t
```

$T_{arranque} = (14) + (14) + (8 + 14) + (14) = 64\ ciclos$

Con encadenamiento y 2 un. funcionales de memoria

```
convoy 1:  LV      V1,Ra      ;Cargar a
           LV      V2,Rb      ;Cargar b
           ADDV    V3,V1,V2   ;s := a + b
           MULTV   V4,V1,V3   ;t := a * s
convoy 2:  SV      Rs,V3      ;Almacenar s
           SV      Rt,V4      ;Almacenar t
```

$T_{arranque} = (14 + 8 + 10) + (14) = 46\ ciclos$

Actividad 3.15 Autor: José Sánchez Moreno

Aplicando la técnica de seccionamiento de bucles, escriba el código escalar y vectorial necesario para realizar la operación vectorial `Y:=Y+X*s`, donde s es un escalar en coma flotante y `X` e `Y` son vectores de 150 elementos con una longitud de 8 bytes. Considere que `s`, `X` e `Y` se encuentran almacenados en las direcciones de memoria M[R1], M[R2] y M[R3], respectivamente. El direccionamiento de memoria se realiza por bytes y MLV es 64. ¿Qué cambios habría que realizar en el código si la longitud del vector no se conociese en tiempo de compilación sino en tiempo de ejecución?

Dado que el compilador recurre a la técnica de seccionamiento para vectorizar el bucle, el código resultante estará compuesto por una mezcla de instrucciones escalares y vectoriales. La técnica de seccionamiento consiste en realizar el procesamiento de un vector de n elementos en secciones, donde la primera sección tiene una longitud de (*n mod MVL*) elementos y las restantes MVL elementos. Dado que la longitud de los vectores es de 150 elementos y MVL es 64, la primera sección tendrá una longitud de (*150 mod 64*) *= 32* elementos y el total de secciones a procesar será de tres: una de 32 elementos y dos de 64.
Si consideramos que MVL es un registro especial cuyo contenido es inalterable y es 64, un posible código sería el siguiente:

```
      ADDI    R4,R0,#1200  // Longitud vector de 150 elementos en bytes
      ADD     R4,R4,R2     // Cálculo del fin del vector X
      ADDI    R5,R0,#32    // Elementos de la primera sección a procesar
      MOVI2S  VLR,R5       // Longitud en elementos de la primera sección
      MULTI   R5,R5,#8     // Longitud en bytes de primera sección (32*8)
      LD      F1,0(R1)     // Carga del escalar s
Inicio:LV     V1,R2        // Carga del vector X
      LV      V2,R3        // Carga del vector Y
      MULTSV  V1,V1,F1     // X*s
      ADDV    V2,V2,V1     // Y+X*s
      SV      R3,V2        // Almacenamiento de Y
      ADDI    R2,R2,R5     // Siguiente sección de X
      ADDI    R3,R3,R5     // Siguiente sección de Y
      MULTI   R5,MVL,#8    // Longitud en bytes de sección siguiente
      MOVI2S  VLR,MVL      // Longitud en elementos de sección siguiente
      SUB     R6,R4,R2     // ¿Se ha llegado al final de A?
      BNZ     R6,Inicio    // Repetir iteración si no es el fin del vector
```

En el código se pueden apreciar con claridad las 6 instrucciones escalares que ocasionan el T_{base} antes de iniciar el procesamiento del bucle y las 6 instrucciones escalares, una vez dentro del bucle, que generan el T_{bucle}. El fragmento de código vectorial que produce el gasto expresado por $T_{arranque}$ y $T_{elemento}$ está compuesto por 5 instrucciones vectoriales.
En caso de que en tiempo de compilación no se conociese la longitud del vector sería necesario modificar las 6 primeras instrucciones escalares iniciales para que se realizase el cálculo de la primera sección. Para ello, supondremos la existencia de una instrucción MOD y que la longitud del vector se encuentra almacenado en el registro R10:

```
MULTI   R4,R10,#8    // Longitud del vector en bytes
ADD     R4,R4,R2     // Cálculo del fin del vector X
MOD     R5,R10,MVL   // Elementos de la primera sección a procesar
MOVI2S  VLR,R5       // Se fija la longitud de la primera sección
MULTI   R5,R5,#8     // Longitud de la primera sección en bytes
```

Dado el siguiente bucle:

```
for (i=0; i<256; i++)
   if (i mod 2) then
      B[i]:= B[i]+A[i];
   end if;
end for;
```

Genere el correspondiente código vectorial y calcule el tiempo de ejecución del bucle vectorizado considerando que MVL es 64, T_{bucle} de 10 ciclos y T_{base} de 5 ciclos. El coste de arranque de la unidad de suma vectorial es de 6 ciclos y el de la unidad de carga/almacenamiento de 12 ciclos y ambas se pueden encadenar. Las direcciones de A y B se encuentran ubicadas en los registros R1 y R2.

```
          ADDI    R10,R0,#8    ;Separación entre elementos, 8 bytes, 1 elemento de separación
          ADDI    R15,R1,#2048 ;Dirección de fin del vector A
          ADDI    R20,R0,#64   ;Elementos de la sección a procesar
inicio:   MOVI2S  VLR,R20      ;Fijamos a 64 elementos por sección con el registro especial VLR
          LVWS    V1,(R1,R10)  ;Cargar elementos pares del vector A
          LVWS    V2,(R2,R10)  ;Cargar elementos pares del vector B
          ADDV    V3,V2,V1     ;B+A
          SVWS    (R2,R10),V3  ;B:=B+A
          ADDI    R1,R1,#1024  ;Avanzar 128 elementos del vector A
          ADDI    R2,R2,#1024  ;Avanzar 128 elementos del vector B
          SGE     R25,R1,R15   ;Si R1>=R15 (i>=256) entonces R25:=1 si no R25:=0.
          BEQZ    R25,inicio   ;Si R25=0 entonces ir a inicio
```

```
convoy1:  LVWS V1,(R1,R10)
convoy2:  LVWS V2,(R2,R10)
          ADDV V3,V2,V1
convoy3:  SVWS (R2,R10),V3
```

Normalmente cuando se evalúa el rendimiento evaluamos la diferencia de rendimiento debido a una mejora introducida. De manera más formal:

$$Speedup\ debido\ a\ la\ mejora\ E = \frac{Tiempo_{\text{sin } E}}{Tiempo_{con\ E}} = \frac{Rendimiento_{con\ E}}{Rendimiento_{\text{sin } E}}$$

En particular, nos referimos al ***speedup*** como una función del paralelismo de la máquina. Suponga que tiene un programa que realiza una cantidad fija de trabajo, del que una fracción ***s*** debe realizarse de manera secuencial. El resto del trabajo puede ser paralelizado sobre *P* procesadores. Asumiendo que T_1 es el tiempo que tarda en ejecutar el programa en un único procesador, encontrar una fórmula para describir T_p, como el tiempo que tarda en ejecutar en *P* procesadores. Usar esto para encontrar una fórmula para el límite superior del ***speedup*** potencial sobre *P* procesadores (esto es una variante de la Ley de Amdahl). Explicar este límite superior.

s : fracción de programa con ejecución secuencial.
$(1-s)$: fracción de programa con posibilidad de ejecución paralela.
t_s : tiempo de ejecución del programa en un procesador.

Para T_1 (1 procesador): $T_1 = s \cdot t_s + (1-s) \cdot t_s$

Para T_p (M procesadores): $T_p = s \cdot t_s + \frac{(1-s) \cdot t_s}{M}$

Según la ***ley de Amdahl***, si el número de procesadores M tiende a infinito, se tiene que:

$$\lim_{M \to \infty} \frac{M}{1+(M-1) \cdot s} = \frac{1}{s}$$

Es decir que la aceleración utilizando un número grande o infinito de procesadores está acotada o tiende a la fracción del programa con ejecución secuencial.

Un ejemplo:

$t_s = 20s$
$s = 15\%$
$M = 10000$
$$S(M) = \frac{10000}{1+(10000-1) \cdot 0{,}15} = \frac{10000}{1500{,}85} = 6{,}66289$$

Si miramos la eficiencia cuando tenemos un número infinito de procesadores sería:

$$E = \lim_{M \to \infty} \frac{1}{1+(M-1) \cdot s} \cdot 100 = 0$$

Que quiere decir que si tenemos un número infinito de procesadores el tiempo de utilización de procesadores tiende a cero.

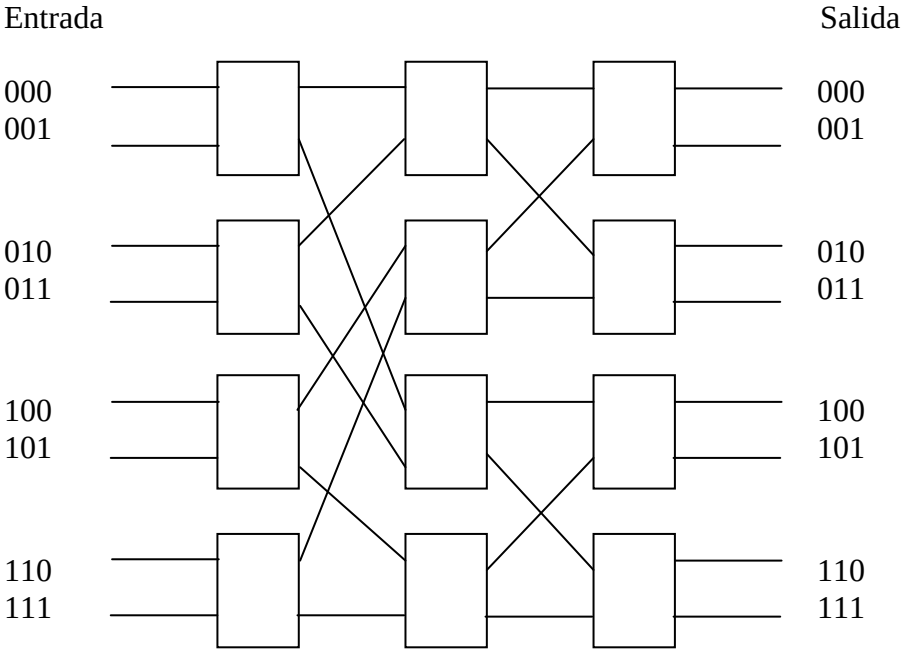
Dibuje una red ***baseline*** de 8 entradas y 8 salidas. Explique detalladamente el proceso de construcción de la red.

La red está en la página 325, serían 3 bloques,

1 bloque de 8 conmutadores (lo que está en línea discontinua)

2 bloques de 4 conmutadores(a la derecha de la línea discontinua)

La construcción sería desplazar cada bit a la derecha circularmente, que si no me equivoco sería así:

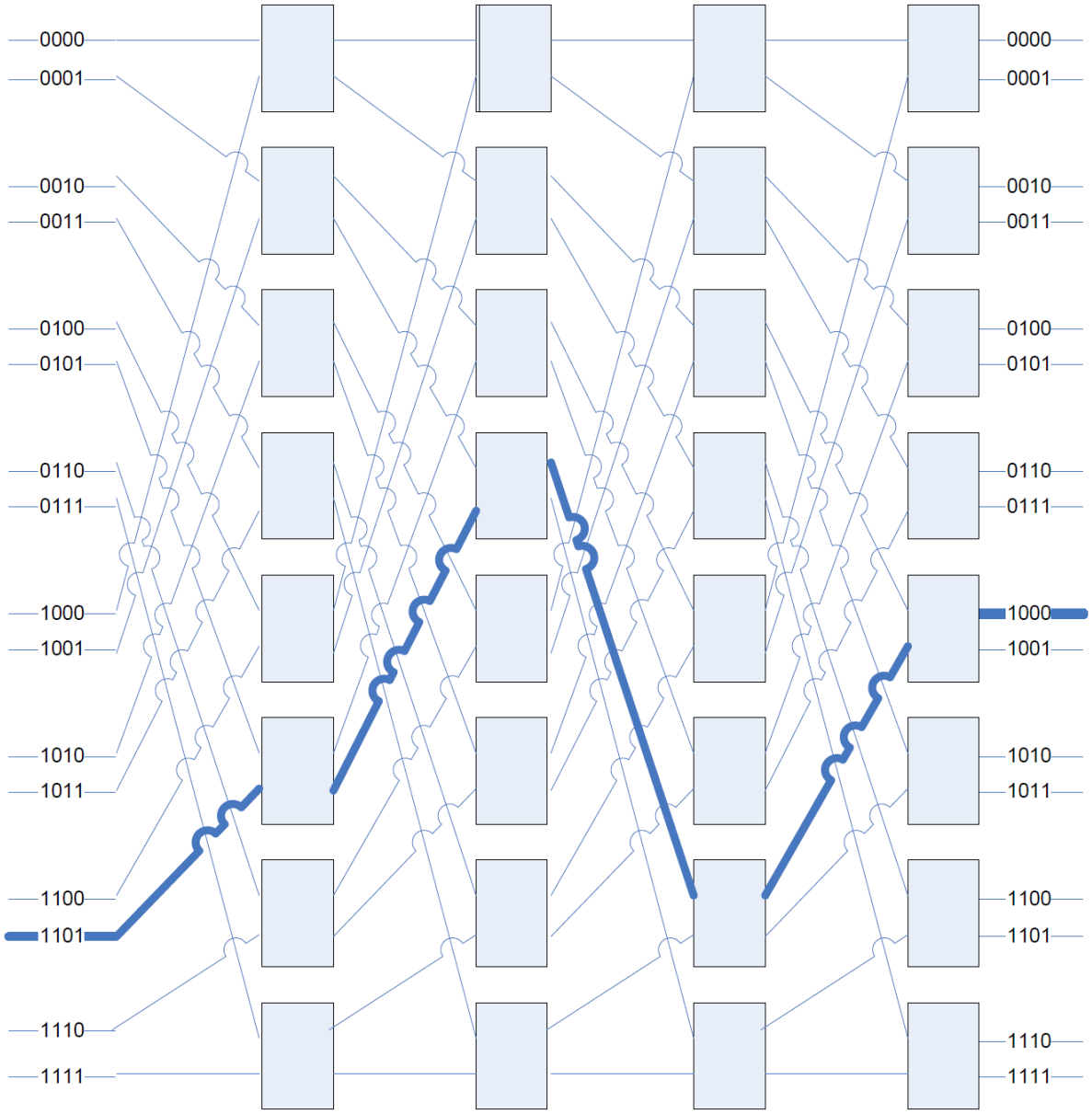


Esto en el bloque de 8x8

En un bloque de 4x4 se haría igual pero con 2 bits.

Este algoritmo de permutar elementos se llama barajamiento perfecto inverso.

Defina qué entiende por una red omega y dibuje una para 16 procesadores. Explique razonadamente la conmutación de cada conmutador de la red que ha dibujado para enviar un mensaje del procesador 1101 al banco de memoria 1000.



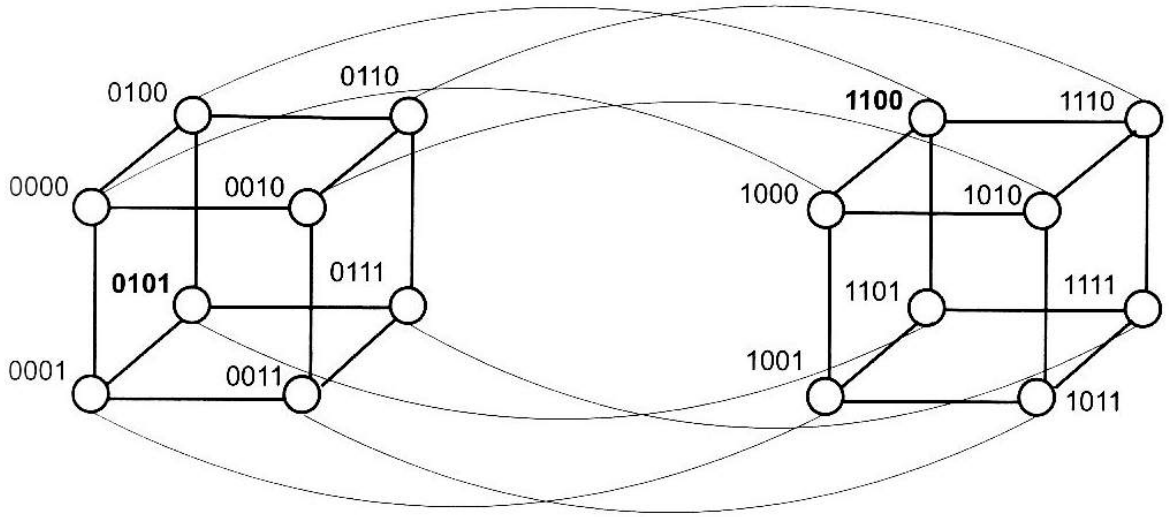
Para ir del procesador 1101 al 1000:
Se coge el procesador de destino 1000 y desde el procesador origen 1101

0 SALIDA SUPERIOR
1 SALIDA INFERIOR
1 INFERIOR
0 SUPERIOR
0 SUPERIOR
0 SUPERIOR

Es lo que está en trazo grueso en la figura
Se construye la red rotando los bits a izquierdas, con el algoritmo de barajamiento perfecto

Actividad 4.4 *Autor: Oscar Moya Martínez*

A partir de la red estática de la siguiente figura:



1. ¿Qué tipo de red es?

2. Se desea transmitir un mensaje desde el procesador a = 0101 al procesador b = 1100. Si se comienza a buscar el camino por el bit menos significativo, explique razonadamente cuál es el camino que debe seguir el mensaje.

3. Defina que entiende por conectividad de arco y por ancho de bisección. Calcule la conectividad de arco y el ancho de bisección de la red de la figura.

1) Es una red estática hipercubo de dimensión 4.

2) La distancia Hamming: $0101 \otimes 1100 = 1001$
   Es 2(dos unos). Luego:
   Del nodo 010**1** nos movemos a 010**0**
   Del nodo **0**100 nos movemos a **1**100

3) **Conectividad de arco**: para tener dos redes disjuntas, y por tanto desconexas, de manera que no tengan ningún nodo en común tenemos que eliminar como mínimo: $\log_2 p = \log_2 16 = 4$

   Si dado un nodo eliminamos sus 4 enlaces se desconecta del resto de la red, y tenemos dos redes disjuntas, una con un nodo y otra con 15.
   Ancho de bisección: para dividir la red en dos subredes iguales, que sean desconexas y que tengan los mismo nodos(8 nodos/red), tenemos que eliminar: $\frac{p}{2} = \frac{16}{2} = 8$
   Que serían los 8 enlaces que unen un cubo con el otro.

Se dispone de un procesador que tarda 6 segundos en ejecutar un determinado programa. Sea *f = 0,3*, donde *f* representa la fracción del programa que no puedo dividirse en tareas paralelas. Si se considera que no hay sobrecarga cuando el programa se divide en tareas paralelas, ¿Cuál es el tiempo de computación necesario para ejecutar el programa 12 procesadores de las mismas prestaciones que el original?

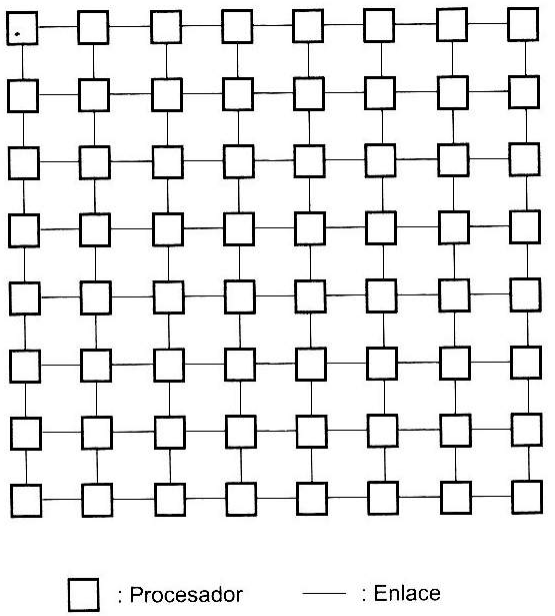
$t_s = 6s$

$f = 0,3$

$M = 12$

$$t_p(M) = f \cdot t_s + \frac{(1-f)\cdot t_s}{M} = 0,3 \cdot 6 + \frac{(0,7)\cdot 6}{12} = 1,8 + 0,35 = 2,15$$

Para transmitir un mensaje de ***n*** bytes a través de H enlaces de una red sin saturación, que utiliza el protocolo de almacenamiento y reenvío, el enrutamiento tarda $H = \frac{n}{W} + (H-1)\cdot R$, donde W es el ancho de banda del enlace y R es el tiempo de salto. En una red que utilice el algoritmo ***cut-through*** el tiempo sería $\frac{n}{W} + (H-1)\cdot R$. Si consideramos una red de tipo ***mesh*** cuadrada de 8 x 8 (ver figura adjunta), cada procesador con un retraso de 250 ns y con enlaces de 40 MB/s ¿Cuál es el tiempo de transferencia mínimo, máximo y medio para un mensaje de 64 bytes? ¿y para uno de 256 bytes? Calcule los mismos tiempos para el caso del enrutamiento mediante el algoritmo ***cut-through***.



ACTIVIDAD 4.6

$W = 40\ MB\ /\ s = 41943040\ b\ /\ s$
$R = 250\ ns = 250 \cdot 10^{-9} segundos$
$n = 64\ bytes$
$H = número\ de\ enlaces\ que\ se\ atraviesan$

Para almacenamiento y reenvío:

Tiempo de transferencia máximo:
El número máximo de enlaces es 14 entre dos procesadores(sin bucles), por tanto $H = 14$

$H \cdot \frac{n}{W} + (H-1) \cdot R = 14 \cdot \frac{64}{41943040} + 13 \cdot 250 \cdot 10^{-9} = 0,000024612\ segundos = 24,6123\ \mu seg.$

Tiempo de transferencia mínimo:
Serían dos procesadores adyacentes, por tanto $H = 1$

$H \cdot \frac{n}{W} + (H-1) \cdot R = \frac{64}{41943040} = 1,52587\ \mu seg.$

Tiempo de transferencia medio:
Podría ser la media entre el maximo y el mínimo que son $12,306150\ \mu seg.$

Para cut-through

Tiempo de transferencia máximo:
$\frac{n}{W} + (H-1)R = \frac{64}{41943040} + 13 \cdot 10^{-9} = 1.53887\ \mu seg.$

Tiempo de transferencia minimo:
$1,52587\ \mu seg.$

Tiempo de transferencia medio:
$1.53237\ \mu seg.$

Considere un simple esquema de diferencias finitas en dos dimensiones en el que en cada paso cada punto de la matriz es actualizado con la media ponderada de sus cuatro vecinos:

$$A[i, j] = A[i, j] - w\left(A[i-1, j] + A[i+1, j] + A[i, j-1] + A[i, j+1]\right)$$

Todos los valores son números en coma flotante de 64 bits. Asumiendo que cada procesador calcula un elemento y la matriz tiene tamaño 1024 x 1024, ¿Qué cantidad de datos debe transmitirse entre procesadores en cada paso? Explicar cómo se puede dividir este cálculo entre 64 procesadores para poder minimizar la cantidad de datos transmitidos, y calcular cuantos datos se deberían transmitir en cada paso.

Actividad 4.7 Andrés Heredia Castro

Cada procesador necesita conocer el dato de sus cuatro vecinos (excepto los extremales que tendrían 3 o 2) para actualizar su valor. Por lo tanto, 1023x1023 procesadores necesitan recibir cuatro mensajes de 64 bytes, 1022x4 necesitan 3 mensajes, y 4 procesadores necesitan 2 mensajes, resultando la cantidad de datos transmitida (1023*1023*4+1022*4*3+4*2)*64=268.696.832 bytes o 256,2 Mbytes

La forma de minimizar la cantidad de datos transmitidos es dividir los datos de la matriz en 64 bloques, de forma que cada procesador tendrá en su memoria los datos de la mayoría de los vecinos de los elementos de la matriz, excepto los de los bordes del bloque. Así la matriz quedaría dividida en bloques de 1024*1024/64 = 16384 elementos que forman una submatriz 128x128 de la original.

Por tanto, y considerando en este caso por simplicidad, que todos los bloques son interiores (esto es, tienen bloques adyacentes en sus cuatro lados, y por tanto todos los elementos tienen 4 vecinos), cada procesador necesitaría recibir los datos de los vecinos externos de sus elementos extremales, que son un total de 129x4=516 por lo que cada procesador requiere en cada paso 33024 bytes de datos transmitidos, y por tanto un total de 33024*64 = 2113536 bytes = 2,01 Mbytes

La solución propuesta por Andrés es correcta, aunque los cálculos son erroneos. Revisalos.

Victor.

Cierto. El error, si no me vuelvo a equivocar, está en el número de procesadores internos en el primer apartado, que sería 1022x1022, con lo que la expresión de la cantidad de información transmitida por paso sería (1022*1022*4+1022*4*3+4*2)*64=268173312 bytes o 255,75 Mbytes

El segundo apartado, con la aproximación realizada, creo que es correcto.

En cuanto a eso de "esquema de diferencias finitas" ¿a que se refiere?

Hola Andrés,

Has corregido uno de los errores. Pero, estás seguro de que las unidades que has usado son correctas?

En el segundo apartado, siguiendo la suposición de que todos los sub-bloques son internos, por qué el número de datos a transmitir entre procesadores es 129x4?

Victor.

Vale. El paso de bits a Mbytes es incorrecto. Realmente las últimas unidades serían Mbits.

En cuanto a por qué el número de datos transmitidos a cada procesador, creo que veo mi error, y es 128x4 (había contado los vecinos de las esquinas, pero no deben contarse pues requieren la variación de dos índices de la matriz y no solo uno).

Un saludo y gracias

Hola Andrés,

Exacto, el cálculo lo has medido en bytes cuando son bits.

La otra corrección también es correcta. Para afectar a los vecinos en las diagonales tienes que variar simultáneamente ambos índices.

Victor.

Hola Andrés,

Perdona, se me ha olvidado contestar a tu pregunta sobre las diferencias finitas. La fórmula incluida en el ejercicio corresponde a un cálculo de diferencias finitas (http://es.wikipedia.org/wiki/Diferencia_finita). Lo único relevante de la fórmula para el problema es el número de datos que un procesador necesita conocer en cada momento para poder realizar el cálculo (es decir, sus 4 vecinos). Podría darse el caso, por ejemplo, de que la fórmula utilizase 8 vecinos, realizando variaciones en los dos índices (i y j).

Saludos.

Victor.

Actividad 4.8 *Autor: Oscar Moya Martínez*

Considere un multiprocesador de memoria compartida distribuida. Considere un modelo simple de coste donde los accesos a la caché local tardan 10 ns, los accesos a memoria local tardan 100 ns y los accesos a memoria remota tardan 400 ns. Si ejecutamos en este sistema un programa paralelo correctamente balanceado con un 80% de accesos a caché, 10% de accesos a memoria local y 10% de accesos remotos, ¿Cuál será el tiempo efectivo de acceso a memoria? Si el cálculo tiene un límite de memoria, ¿Cuál será la máxima tasa de cálculo?
Considere el mismo programa ejecutándose en un sistema uniprocesador. Ahora el procesador tiene un ratio de acierto de caché del 70% y el 30% de accesos a memoria local.¿Cuál será la tasa efectiva de cálculo para un procesador?¿Cuál será la tasa de cálculo?

Hola Oscar,

La solución que propones es correcta, teniendo en cuenta las suposiciones iniciales.

Si te das cuenta, la tasa de acceso a memoria del monoprocesador es menor que la del multi-procesador. Incluso si en tus cálculos supones que el monoprocesador tiene una caché de gran tamaño y todos los datos se encuentran en ella, la tasa de cálculo del multiprocesador se ve afectada por el factor de escala (el número de procesadores del sistema), dando un mejor resultado a medida que aumenta el tamaño del sistema.

Saludos.

Victor.

ACTIVIDAD 4.8

Para un sistema multiprocesador

El tiempo efectivo de acceso a memoria sería:
$10 \cdot 0.8 + 100 \cdot 0.1 + 400 \cdot 0.1 = 58\ ns.$

Para no complicarlo mucho, podemos considerar un programa muy sencillo que se ejecute una arquitectura escalar, que realice 2 cargas, 1 FLOP, y 1 almacenamiento, y suponiendo que un FLOP tarda 40 ns:

LD	F1,0(R1)	58 ns
LD	F2,0(R2)	58 ns
ADDD	F3,F1,F2	40 ns
SD	0(R3),F3	58 ns

Este programa se debe ejecutar secuencialmente, pero podemos pensar en un paralelismo en el que este programa se ejecuta en otros procesadores(cada uno con sus registros y sus memorias locales), por lo que estaríamos hablando de un paralelismo SPMD.
La tasa efectiva de cálculo sería:
$58 + 58 + 40 + 58 = 214\ ns\ /\ FLOP$

La inversa
$$\frac{1}{214 \cdot 10^{-9}} = 4672897,19\ FLOPS = 4,67289719\ MFLOPS$$

La máxima tasa de cálculo es cuando todos los datos se encuentran en la cache de los procesadores:
$10 + 10 + 40 + 10 = 70\ ns\ /\ FLOP$

$$\frac{1}{70 \cdot 10^{-9}} = 14285714,28\ FLOPS. = 14,28571428\ MFLOPS$$

Estos datos son para un procesador, habría que multiplicar el número de procesadores por los MFLOPS, por lo que se considera que $f = 0$ y la aceleración es lineal.

Para un sistema con un procesador

Utilizando el mismo programa de pruebas que en sistema multiprocesador. La tasa efectiva de acceso a memoria:
$0,7 \cdot 10 + 0,3 \cdot 100 = 37\ ns$

La tasa efectiva de cálculo sería:
$37 + 37 + 40 + 37 = 151\ ns\ /\ FLOP$

$$\frac{1}{151 \cdot 10^{-9}} = 6622516,55\ FLOPS = 6,622516\ MFLOPS$$

Y la máxima tasa de cálculo se calcula igual que en el sistema multiprocesador.

Considere el enrutamiento de mensajes en un sistema paralelo que utiliza el algoritmo de almacenamiento y reenvío. En este sistema el coste de envío de un mensaje de tamaño ***m*** desde P_{origen} hasta $P_{destino}$ a través de un camino de longitud ***d*** es $t_s + t_w \cdot d \cdot m$. Un método alternativo para el envío de un mensaje de tamaño m sería el siguiente. El usuario divide el mensaje en ***k*** partes, cada uno de tamaño $\frac{m}{k}$, y envía las distintas partes por una desde P_{origen} hasta $P_{destino}$. Para este nuevo método, encontrar una expresión para el tiempo de transferencia del mensaje de tamaño ***m*** hasta un destino situado a una distancia de ***d*** saltos, teniendo en cuenta los siguientes casos:

- Asumir que un mensaje puede enviarse desde P_{origen} tan pronto como el mensaje anterior ha alcanzado el siguiente nodo en el camino.
- Asumir que un mensaje puede enviarse desde P_{origen} únicamente cuando el mensaje anterior ha alcanzado $P_{destino}$.

Para cada caso, analizar el comportamiento de la expresión cuando ***k*** varía entre 1 y ***m***. a su vez, calcular el valor óptimo de ***k*** si t_s es muy grande, y si $t_s = 0$.

ACTIVIDAD 4.9

t_s : *tiempo de inicialización*
t_w : *tiempo de salto de la cabecera*
d : *longitud de camino*
m : *tamaño del mensaje en palabras*
k : *partes en las que se divide el mensaje*

Para almacenamiento y reenvío nos dicen que:
$t_s \cdot t_w \cdot d \cdot m$

a) Mezcla de almac. y reenvio y cut-through
$\underbrace{t_s + t_w \cdot d \cdot \frac{m}{k}} + (k-1) \cdot t_w \cdot \frac{m}{k} = t_s + \frac{mdt_w}{k} + kmt_w - t_w - \frac{m}{k}$
Coste de enviar el primer flit(compuesto de 1, 2, 3...m palabras)

b) Parecido al almacenamiento y reenvío.
$k \cdot \underbrace{(t_s + t_w \cdot d \cdot \frac{m}{k})} = kt_s + mdt_w$
Coste de enviar el primer flit

Analizar cuando k varia entre 1 y m. **Encontrar un valor optimo de k si t_s es muy grande y si t_s=0**

a) $t_s + \frac{mdt_w}{k} + kmt_w - t_w - \frac{m}{k}$
Si k tiende a m, substituyendo k por m resulta que:
$t_s + \frac{mdt_w}{m} + m^2t_w - t_w - \frac{m}{m} = t_s + dt_w + m^2t_w - t_w - 1 = \boxed{t_s + t_w(d + m^2 - 1) - 1}$

Si k tiende a 1, substituyendo k por 1
$t_s + mdt_w + mt_w - t_w - m = \boxed{t_s + t_w(md + m - 1) - m}$
En ambos casos el valor de k no depende de t_s, por lo que daría igual que valor tiene k según si t_s es grande o no.

b) $kt_s + mdt_w$
Si k tiende a m, substituyendo k por m resulta que:
$mt_s + mdt_w$
Si t_s es grande, interesa un k pequeño(dividir el mensaje en paquetes más grandes)

Si k tiende a 1, substituyendo k por 1
$t_s + mdt_w$
En este caso el valor de k, no se ve tan afectado por el valor de t_s, por lo que a medida que k tiende a 1 el valor de t_s si es grande o no se hace menos importante.

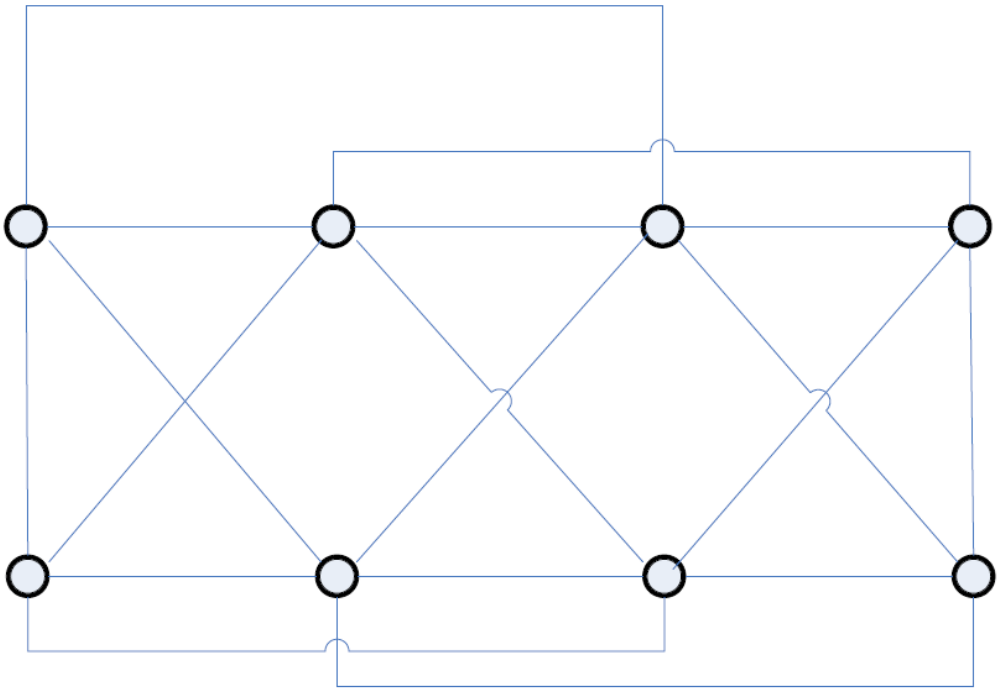
Calcular el diámetro, número de enlaces y ancho de bisección de un ***d-cubo k-ario*** con ***p*** nodos. Definir $\boldsymbol{l_{av}}$ como la distancia media entre dos nodos de la red. Calcular $\boldsymbol{l_{av}}$ para el ***d-cubo k-ario***.

Actividad 4.11 *Autor: Oscar Moya Martínez*

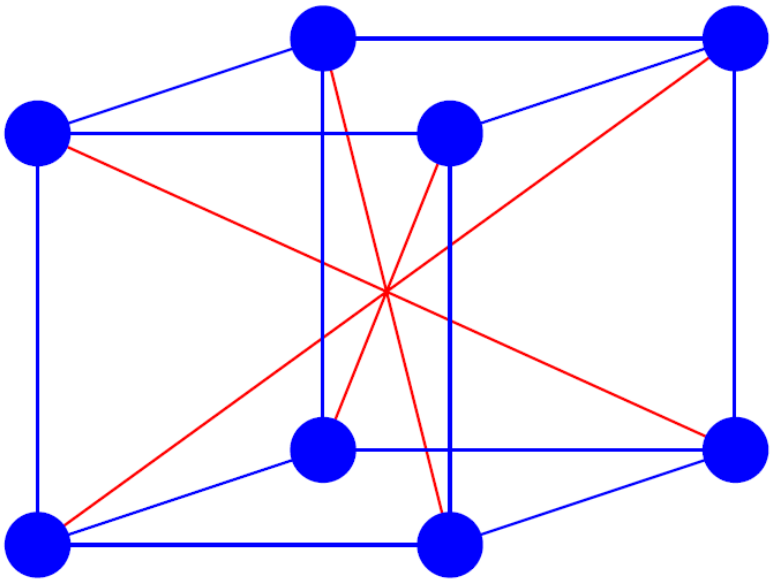
Las etiquetas de los nodos de un ***hipercubo d-dimensional*** utilizan ***d*** bits. Fijando un bit ***k*** cualquiera de la etiqueta, demostrar que los nodos cuyas etiquetas difieren en los ***d–k*** bits restantes forman un subcubo ***(d–k)-dimensional*** compuesto por $2^{(d-k)}$ nodos.

Deducir la condición a partir del método de construcción/división de un hipercubo, pág. 315

Dibuje una red ***estática*** con los siguientes valores en sus parámetros: diámetro = 2, conectividad de arco = 4 y ancho de bisección = 8. ¿Cuál sería el coste de la red que ha dibujado?

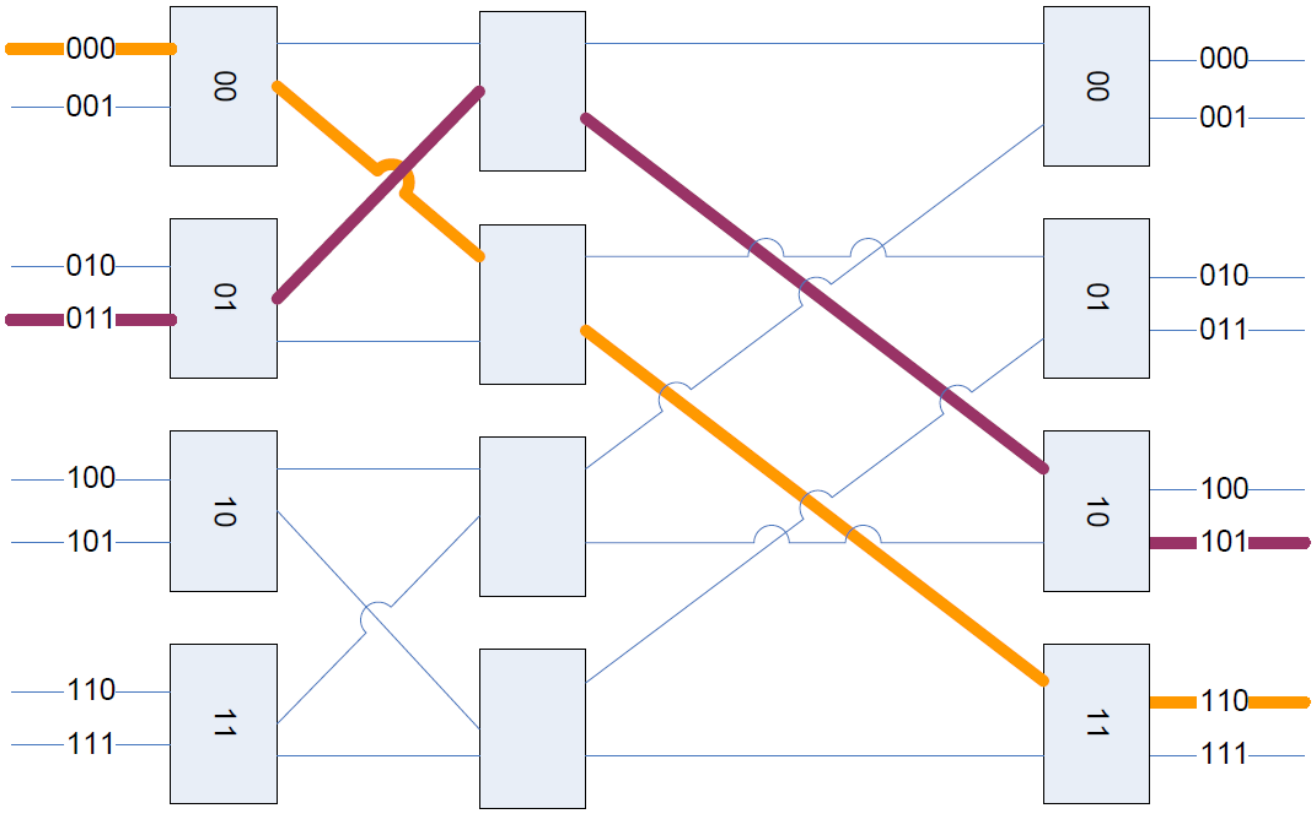


Diámetro = 2
Conectividad de arco = 4
Ancho de bisección = 8
Coste = 18



Defina los siguientes conceptos de redes estáticas: distancia Hamming y ancho de banda del canal.

Dibuje una red ***butterfly*** de 8 x 8 y describa el enrutamiento desde el procesador 011 hasta el 101 y desde el procesador 000 hasta el 110.



0 SALIDA DIRECTA
1 SALIDA CRUZADA

Para ir del procesador 011 al 101:
Se cogen los conmutadores donde están conectados los procesadores, en este caso el 01, y el 10

01 XOR 10 = 11

1 CRUZADA
1 CRUZADA

Para ir del procesador 000 al 110

00 XOR 11 = 11

1 CRUZADA
1 CRUZADA