PRÁCTICA III
Segmentación avanzada.

Como sabemos, para mantener el procesador a pleno rendimiento, se debe explotar el paralelismo entre las instrucciones buscando secuencias de instrucciones no relacionadas que se puedan solapar en la segmentación. Dos instrucciones relacionadas deben estar separadas por una distancia igual a la latencia de la segmentación de la primera de las instrucciones.
Cuando tratamos de ejecutar más instrucciones en cada ciclo de reloj y tratamos de solapar más instrucciones, necesitaremos encontrar y explotar más el paralelismo a nivel de instrucción. En cursos posteriores se examinan organizaciones de segmentación que requieran más paralelismo entre instrucciones.
En esta práctica examinaremos una técnica sencilla de compilación que ayudará a crear paralelismo adicional sin necesidad de entrar en este tipo de organizaciones.

Incremento del paralelismo a nivel de instrucción mediante desenrollamiento de bucles

Para comparar los enfoques explicados en esta sección, utilizaremos un bucle que suma un valor escalar a un vector en memoria. El código de DLX, no teniendo en cuenta la segmentación, será el siguiente:

```
Loop: LD    F0, 0(R1)     ; carga el elemento del vector
      ADDD  F4, F0, F2    ; suma el escalar de F2
      SD    0(R1), F4     ; almacena el elemento del vector
      SUB   R1, R1, #8    ; decrementa el puntero en 8 bytes
      BNEZ  R1, Loop      ; salta cuando no es cero
```

Supondremos las latencias de operaciones que se indican en la siguiente tabla:

Instrucción que produce resultado	Instrucción destino	Latencia en ciclos
Op FP ALU	Otea op FP ALU	3
Op FP ALU	Almacenamiento doble	2
Carga doble	Op FP ALU	1
Carga doble	Almacenamiento doble	0

Por simplicidad, suponemos que el array comienza en la posición 0.
Comencemos viendo cómo ejecutará el bucle cuando se planifica en una segmentación sencilla para DLX con las latencias conocidas en la práctica anterior

Sin planificación el bucle se ejecutaría como sigue:

			Ciclo de reloj
Loop:	**LD**	**F0, 0(R1)**	**1**
	detención		**2**
	ADDD	**F4, F0, F2**	**3**
	detención		**4**
	detención		**5**
	SD	**0(R1), F4**	**6**
	SUB	**R1, R1, #8**	**7**
	BNEZ	**R1, Loop**	**8**
	detención		**9**

Con planificación

Loop:	**LD**	**F0, 0(R1)**	
	detención		
	ADDD	**F4, F0, F2**	
	SUB	**R1, R1, #8**	
	BNEZ	**R1, Loop**	; salto retardado
	SD	**8(R1), F4**	; intercambio con SUB

El tiempo de ejecución se ha reducido de 9 a 6 ciclos de reloj.
Es importante observar que para poder realizar esta planificación, el compilador debe posibilitar el intercambio entre SUB y SD, cambiando la dirección de SD (antes era 0(R1) y ahora 8(R1). Esto no es trivial, ya que la mayoría de los compiladores verían que la instrucción SD depende de SUB y rechazarían el intercambiarlas. Un compilador más inteligente calcularía la relación y realizaría el intercambio. La dependencia entre LD, ADDD y SD determina el número de ciclos de reloj para este bucle.

En el ejemplo anterior, completamos una iteración del bucle y terminamos un elemento del vector cada 6 ciclos de reloj, pero el trabajo real de operación sobre el elemento del vector necesita exactamente 3 de los 6 ciclos de reloj. Los 3 ciclos de reloj restantes se emplean en gastos del bucle –SUB-BNEZ- y en una detención.
Para eliminar estos tres ciclos de reloj necesitamos obtener más operaciones en el bucle. Un sencillo esquema para incrementar el número de instrucciones entre las ejecuciones de los saltos del bucle es ***desenrollar el bucle*** (***loop enrolling***). Esto se hace replicando múltiples veces el cuerpo del bucle, ajustando su código de terminación y planificando entonces el bucle desenrollado. Para lograr una planificación efectiva, utilizaremos diferentes registros para cada iteración, incrementando así el número de registros.

A continuación se expone la técnica *loop enrolling* sobre el bucle del ejemplo anterior, desenrollando el bucle tres veces (manteniendo cuatro copias del cuerpo del bucle), suponiendo que R1, inicialmente es un múltiplo de 4 y sin reutilizar ningún registro.

Desenrollado sin planificación

```
Loop: LD      F0, 0(R1)
      ADDD    F4, F0, F2
      SD      0(R1), F4       ; se elimina SUB & BNEZ
      LD      F6, -8(R1)
      ADDD    F8, F6, F2
      SD      -8(R1), F8      ; se elimina SUB & BNEZ
      LD      F10, -16(R1)
      ADDD    F12, F10, F2
      SD      -16(R1), F12    ; se elimina SUB & BNEZ
      LD      F14, -24(R1)
      ADDD    F16, F14, F2
      SD      -24(R1), F16
      SUB     R1, R1, #32
      BNEZ    R1, Loop
```

Desenrollado con planificación

```
Loop: LD      F0, 0(R1)
      LD      F6, -8(R1)
      LD      F10, -16(R1)
      LD      F14, -24(R1)
      ADDD    F4, F0, F2
      ADDD    F8, F6, F2
      ADDD    F12, F10, F2
      ADDD    F16, F14, F2
      SD      0(R1), F4
      SD      -8(R1), F8
      SD      -16(R1), F12
      SUB     R1, R1, #32     ; dependencia de salto
      BNEZ    R1, Loop
      SD      8(R1), F16      ; 8-32 = -24
```

El tiempo de ejecución del bucle desenrollado se ha reducido a un total de 14 ciclos de reloj.

La ganancia de planificación sobre el bucle desenrollado es aún mayor que en el original. Esto es porque, al desenrollar el bucle se descubre más cálculo que el que se puede planificar. Planificar el bucle de esta forma exige darse cuenta que las instrucciones de carga y almacenamiento son independientes y se pueden intercambiar.

Ejercicio

La práctica consistirá en la programación en ensamblador de DLX de distintas variantes del bucle denominado **SAXP**. El bucle implementa la operación vectorial **Y=a*X+Y** para un vector de longitud R4/8.

El código del bucle en ensamblador de DLX es:

```
inicio: LD F2, 0(R1)          ; carga X(i)
        MULTD F4, F2, F0      ; multiplica a * X(i)
        LD F6, 0(R2)          ; carga Y(i)
        ADDD F6, F4, F6       ; suma a * X(i) + Y(i)
        SD 0(R2), F6          ; almacena Y(i)
        ADDI R1, R1, 8        ; incrementa índice X
        ADDI R2, R2, 8        ; incremente índice Y
        SGT R3, R1, R4        ; test por si finalizado
        BEQZ R3, inicio       ; bucle si no finalizado
```

Considerar las latencias de las unidades funcionales en coma flotante que por defecto trae el simulador.

1. **Programar el código anterior en un fichero denominado BUCLE.S Utilizar las directivas del ensamblador con el objeto de reservar el espacio de memoria para las dos tablas aunque su contenido sea cero, y cargar las direcciones de comienzo en los registros R1 y R2. Modificar el código ensamblador si fuese necesario.**

2. **Mostrar el número de ciclos de detención para cada instrucción y en qué ciclos de reloj comienza la ejecución de la instrucción en la primera iteración del bucle. ¿Cuántos ciclos de reloj necesita cada iteración del bucle?**

3. **Desenrollar el código DLX para el bucle anterior tres veces, y planificarlo para la segmentación estándar de DLX. Al desenrollar se debería optimizar el código tal y como se ha especificado en los ejemplos de la práctica. Para maximizar el rendimiento será necesario reordenar significativamente el código. Calcular la aceleración sobre el bucle original.**

4. **Calcular el CPI medio que se obtiene en cada iteración del bucle propuesto en los apartados (1) y (3) Para la realización de este cálculo sólo se contabilizarán las instrucciones que conforman el cuerpo del bucle, ignorándose las de inicialización del mismo.**

Todas las ejecuciones y cálculos se efectuarán con el adelantamiento de datos entre etapas habilitado
(comando E*nable Forwarding* en el menú *Configuratio*n).