Ejemplo de software pipelining.

Partamos de un programa muy sencillo que haga la suma de dos vectores.

```
        for (i=0;i<n;i++)
                z[i]=v[i]+w[i];
```

Una versión muy sencilla en assembler sería algo así:

Asignación de Registros:

```
        R42=V
        R43=W
        R44=Z
        R45=i
        R46=n
```

Supongamos también datos de 64 bits (8 bytes)

```
        add     R45 = R0,R0                     Inicializo R45 en 0
while:  cmp.lt  p2,p3 = R45,R46                 Comparo R45 con R46, a ver si es menor
(p3)     br.cond fin                                Si no es menor, hay que saltar al
final

        ld8     R32, [R42] ,8 (post-update)     Cargo el elemento de V
        ld8     R33, [R43] ,8 (post-update)     Cargo el elemento de W
        add     R33 = R32,R33                   Los sumo
        st8     [R34], R33, 8 (post-update)     Almaceno en Z
        add     R45 = 1,R45                     Incremento el contador
        br      while                           Salto al principio del ciclo.
fin:    ...
```

Notar que la versión post-update de las instrucciones de load y store
hacen un incremento del registro
con el que estamos direccionando una vez completada la instrucción.

Para hacer software pipelining, lo primero que hay que hacer es
determinar cuales serán las etapas
de nuestro pipeline.
Yo voy a decidir que tenemos 4 etapas en este ciclo, a saber:

```
LD1     Carga del primer operando
LD2     Carga del segundo operando
ADD     Suma de los operandos
STR     Almacenamiento del resultado
```

Ahora, hagamos un dibujito de cómo funcionaría nuestro pipeline de software:

```
         1       2       3       4       5       6       7       8

        LD1     LD2     ADD     STR
                LD1     LD2     ADD     STR
                        LD1     LD2     ADD     STR
                                LD1     LD2     ADD     STR
                                        LD1     LD2     ADD     STR
```

Aqui distinguimos 3 etapas en el funcionamiento de nuestro pipeline:

Prologo: Entre los ciclos 1 a 3, el funcionamiento del pipeline no es
completo, o sea, no se están ejecutando
todas las etapas del mismo. Empezamos ejecutando sólo la etapa LD1
y de a una, vamos agregando etapas
hasta llegar al Nucleo o Kernel.

kernel: Todas las etapas están funcionando y están entrando nuevos elementos para procesar en cada ciclo.
Son los ciclos 4 y 5 de nuestro ejemplo.

Epílogo: Dejan de entrar nuevos elementos a nuestro pipeline. Son los ciclos 6 a 8. Notar que empezamos a
dejar de ejecutar etapas de nuestro pipeline, empezando por la LD1, siguiendo por LD2 y así sucesivamente.

Usando este dibujo es mas fácil ver el tema de asignación de registros.
Como vimos ayer, los registros R32 en adelante (segun los que hayamos declarado en el alloc, siempre múltiplo de 8),
rotan con cada ejecución de la instrucción de salto con rotación de registros (br.cexit, br.ctop, br.wexit, br.wtop).
Por ejemplo, si tengo un dato en R33, despues de ejecutar la instrucción de salto, ese dato va a estar en R34. Suponiendo que haya reservado 8 registros para rotación (32 al 39), el contenido de R39 estará en R32.

Ciclo 1

Reg:	R32	R33	R34	R35	R36	R37	R38	R39
Dato:	A	B	C	D	E	F	G	H

Ciclo 2, despues del salto:

Reg:	R32	R33	R34	R35	R36	R37	R38	R39
Dato:	H	A	B	C	D	E	F	G

Usando esto, planifiquemos la asignación de registros.

Supongamos que el primer load lo hacemos en R32 y el segundo en R33, veamos qué pasaría...
Ahora vamos a

Ciclo 1

Reg:	R32	R33	R34	R35	R36	R37	R38	R39
Dato:	V0							

Ahora, al final, se ejecuta el salto y queda así al comenzar la siguiente vuelta del ciclo.

Ciclo 2

Reg:	R32	R33	R34	R35	R36	R37	R38	R39
Dato:		V0						

Y si ahora hacemos los dos loads que tocan ejecutar en la segunda vuelta del ciclo...

Ciclo 2

Reg:	R32	R33	R34	R35	R36	R37	R38	R39
Dato:	V1	W0						V0

^
|
¡¡¡Problemas!!!!

Acabamos de pisar V0 con W0... Eso no es lo que queremos, pues necesitamos el dato de W0 para poder sumarlo con V0.
Lo que tenemos que determinar en este punto es, entonces, durante cuantos ciclos vamos a necesitar a cada dato,
a fin de asignarle la cantidad de registros adecuado para que con la rotación, lo tengamos disponible mientras lo necesitemos.

Analizando el pipeline, esto es bastante fácil.

```
 1       2       3       4       5       6       7       8

LD1     LD2     ADD     STR
        LD1     LD2     ADD     STR
                LD1     LD2     ADD     STR
                        LD1     LD2     ADD     STR
                                LD1     LD2     ADD     STR
```

Si en el ciclo 1 cargo un dato, y recién en el 3 lo voy a usar,
entonces voy a necesitar 3 registros para almacenamiento.
El dato del 'LD2'se carga en el segundo ciclo y se usa en el 3er y
4to, así que también son 3 registros.
Entonces, cambiemos el código....

```
        add     R45 = R0,R0                     Inicializo R45 en 0
while:  cmp.lt  p2,p3 = R45,R46                 Comparo R45 con R46, a ver si es menor
(p3)     br.cond fin                                Si no es menor, hay que saltar al
final

        ld8     R32, [R42] ,8 (post-update)     Cargo el elemento de V
        ld8     R35, [R43] ,8 (post-update)     Cargo el elemento de W
        add     R35 = R32,R35                   Los sumo
        st8     [R34], R35, 8 (post-update)     Almaceno en Z
        add     R45 = 1,R45                     Incremento el contador
        br      while                           Salto al principio del ciclo.
fin:    ...
```

Esto parece bastante fácil, sin embargo, hay un detalle que se nos
está pasando por alto...
Fijense en la instrucción 'add R35 = R32,R35'.
Si volvemos a hacer el dibujito de asignación de registros que
habíamos hecho arriba, con la nueva asignación de registros....

```
Ciclo 1
Reg:    R32     R33     R34     R35     R36     R37     R38     R39
Dato:    V0
```

Ahora, al final, se ejecuta el salto y queda así:

```
Ciclo 2
Reg:    R32     R33     R34     R35     R36     R37     R38     R39
Dato:            V0
```

Y si ahora hacemos los dos loads que tocan ejecutar en la segunda
vuelta del ciclo...

```
Ciclo 2
Reg:    R32     R33     R34     R35     R36     R37     R38     R39
Dato:   V1       V0             W0
```

Ejecutamos el salto al final...

```
Ciclo 3
Reg:    R32     R33     R34     R35     R36     R37     R38     R39
Dato:           V1      V0             W0
```

Ahora, ejecutamos los dos primeros loads...

Ciclo 3

```
Reg:    R32     R33     R34     R35     R36     R37     R38     R39
Dato:   V2      V1      V0      W1      W0
```

Y además, el add R35 = R32,R35
Con lo cual quedaría R35 = V2 + W1 !! Esto no es lo que queremos!!

Cual es la solución?? Facil, cambiamos add R35 = R32,R35 por add R36 = R34,R36.
¿Porqué esta diferencia con respecto a los load?
Simplemente, porque una cosa es 'producir' un dato, y otra cosa es el momento en que se lo utiliza.
En nuestro caso, si usaramos R35 y R32, lo que estaríamos diciendo es 'sumar el V que acabo de cargar en este ciclo
con el W que acabo de cargar en este ciclo'. Y como vemos en nuestro esquemita del pipeline, lo que se carga
en cada ciclo, no es lo que tengo que sumar! Por eso es que hay que fijarse donde quedó lo que tengo que sumar.

Entonces, nuestro código queda:

```
        add     R45 = R0,R0                     Inicializo R45 en 0
while:  cmp.lt  p2,p3 = R45,R46                 Comparo R45 con R46, a ver si es menor
(p3)     br.cond fin                                Si no es menor, hay que saltar al final

        ld8     R32, [R42] ,8 (post-update)     Cargo el elemento de V
        ld8     R35, [R43] ,8 (post-update)     Cargo el elemento de W
        add     R36 = R34,R36                   Los sumo
        st8     [R34], R35, 8 (post-update)     Almaceno en Z
        add     R45 = 1,R45                     Incremento el contador
        br      while                           Salto al principio del ciclo.
fin:    ...
```

Ok... Ya tenemos nuestro código casi listo. Ahora tenemos que empezar con el control del pipeline.
Lo que precisamos es controlar de alguna menera la ejecución de las etapas de nuestro pipe, de forma tal
que se ejecuten segun nuestro diseño. La manera mas natural de hacer que una instrucción se ejecute o no
en IA-64 es la predicación, y esa es otra parte del soporte para software pipelining que tiene la arquitectura.

Los registros p16 a p64 también rotan con las instrucciones de salto para software pipelining, así que podemos
usarlos para el control de nuestro pipeline. Por ejemplo:

```
        add     R45 = R0,R0                     Inicializo R45 en 0
while:  cmp.lt  p2,p3 = R45,R46                 Comparo R45 con R46, a ver si es menor
(p3)     br.cond fin                                Si no es menor, hay que saltar al final

(p16)   ld8     R32, [R42] ,8 (post-update)     Cargo el elemento de V
(p17)   ld8     R35, [R43] ,8 (post-update)     Cargo el elemento de W
(p18)   add     R36 = R34,R36                   Los sumo
(p19)   st8     [R34], R35, 8 (post-update)     Almaceno en Z
        add     R45 = 1,R45                     Incremento el contador
        br      while                           Salto al principio del ciclo.
fin:    ...
```

Ahora, como controlamos los valores de los p16-p19?

Fácil, primero hagamos un dibujo de qué es lo que precisamos que pase en estos registros durante
cada vuelta del ciclo.

Ciclo 1: (1 0 0 0)
Ciclo 2: (1 1 0 0)
Ciclo 3: (1 1 1 1)
ciclo 4: (1 1 1 1)
ciclo 5: (1 1 1 1)
ciclo 6: (0 1 1 1)
Ciclo 7: (0 0 1 1)
Ciclo 8: (0 0 0 1)

Notemos, durante el prólogo y el nucleo, se van 'metiendo' unos por la izquierda, y luego en el epílogo,
se van metiendo ceros por la izquierda, hasta vaciar el pipeline.
Primero, inicialicemos los registros de predicado. La manera mas fácil de hacerlo es:

```
mov pr.rot, 0x10000
```

REVISAR ESTA INSTRUCCIÓN EN EL MANUAL Y PROBARLO

Y ahora que tenemos p16 en 1 y el resto en 0, tenemos que buscar la manera de ir insertando 1 a izquierda
durante el prólogo y nucleo, y luego ceros durante el epílogo.
Para eso, las instrucciones de salto con soporte para predicación hacen exactamente eso.
Las instrucciones son:

Para loops con contador:

```
br.ctop
br.cexit
```

Para 'while'

```
br.wtop
br.wexit
```

Estas instrucciones, además de hacer la rotación ya explicada con los registros de uso general que hayamos
declarado para rotar, hacen rotar los registros de predicado p16-p63,
y segun en que lugar del ciclo estemos,
insertan el 1 o 0 correspondiente.

Para indicar si estamos en el prologo o nucleo es bien fácil:
Loops con contador:
Mientras el registro ar.lc (loop count) sea mayor a 0, es que estamos en el prologo o kernel (todavía llenando
el pipeline)

Ciclos 'while':
Mientras la condición de entrada en el ciclo sea verdadera, estamos en el prologo o kernel - todavía llenando
el pipeline.

Una vez que salimos del kernel (ya sea que ar.lc quedó en 0 o que la condición del ciclo dió false),
las instrucciones de salto son controladas por el registro ar.ec (epilog count), que indica cuantos ciclos
se tarda en vaciar el pipeline.

BUSCAR LA ESPECIFICACIÓN COMPLETA DE ESTAS INSTRUCCIONES EN LA BIBLIOGRAFÍA.

Una diferencia importante entre las variantes 'top' y 'exit' es que la top salta cuando la condición es verdadera, mientras que la 'exit' salta cuando la condición es falsa - ideal para un ciclo cuando la condición se chequea al comienzo del ciclo.

Entonces, esto quedaría así:

```
        add     R45 = R0,R0                     Inicializo R45 en 0

        mov     ar.ec = 3
        mov     pr.rot = 0x10000

while:  cmp.lt  p2,p3 = R45,R46                 Comparo R45 con R46, a ver si es menor

(p2)    br.wexit        fin                    Si no es menor, y ar.ec=0, voy al final
(p16)   ld8     R32, [R42] ,8 (post-update)     Cargo el elemento de V
(p17)   ld8     R35, [R43] ,8 (post-update)     Cargo el elemento de W
(p18)   add     R36 = R34,R36                   Los sumo
(p19)   st8     [R34], R35, 8 (post-update)     Almaceno en Z
        add     R45 = 1,R45                     Incremento el contador
        br      while                           Salto al principio del ciclo.
fin:    ...
```

Y con esto, tendríamos listo nuestro ciclo usando software pipeline.

Aclaraciones:
1) No hicimos nada de análisis del código con respecto a latencia de las instrucciones, bundles, etc.
Obviamente, esto podría llegar a cambiar el dibujo. Para ver un ejemplo donde se tiene en cuenta esto,
referirse al ejemplo del libro electrónico que les pasé hace unas semanas.