INTRODUCCIÓN A LA SEGMENTACIÓN

ARQUITECTURA DEL PROCESADOR II

1. Una mirada rápida de la segmentación o tubería

La segmentación es una técnica de implementación en la cual múltiples instrucciones se superponen en ejecución[1]. Hoy en día, la segmentación es practicamente universal. Esta sección se basa principalmente en una analogía del mundo real para dar una descripción general y los términos usados en la segmentación.

> ***SEGMENTACIÓN O TUBERÍA (en inglés PIPELINING):***
> Es una técnica de implementación en la cual múltiples instrucciones se superponen en ejecución, algo muy similar a una línea de ensamblaje.

Quienes van mucho a la lavandería usan segmentación de manera intuitiva.
La aproximación no segmentada de la lavandería sería la siguiente:

1. Colocar una carga de ropa sucia en la lavadora.
2. Cuando la lavadora termina, pasar la ropa húmeda a la secadora.
3. Cuando la secadora termina, ubicar la ropa seca en la mesa y doblarla.
4. Cuando termina de doblar, le píde a un conocido que lleve la ropa a casa.

Una vez que su conocido terminó, Ud. comienza todo para la siguiente carga de ropa sucia.

La aproximación segmentada de la lavandería toma mucho menos tiempo, ver Figura 1.

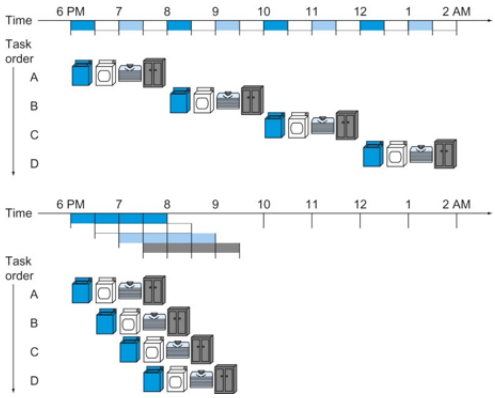

Figura 1. La lavandería sin (arriba) y con (abajo) segmentación

[1]Este material de estudio es la traducción de la sección 4.5 del libro de Patterson y Hennessy [1].

Tan pronto como la lavadora ha finalizado con la primer carga y ubicada en la secadora se puede cargar la lavadora con la segunda carga de ropa sucia. Cuando la primer tanda está seca: se la coloca sobre la mesa y se comienza a doblar, se pasa la carga mojada a la secadora y se pone la siguiente carga en la lavadora. Luego su conocido llevará la primer tanda a la casa, mientras Ud. dobla la segunda carga, la secadora tiene la tercer carga y Ud. coloca la cuarta tanda de ropa en la lavadora. En este punto todos los pasos - llamados etapas de la segmentación - están operando al mismo tiempo. Siempre que haya recursos independientes para cada etapa, se pueden segmentar las tareas.

La paradoja de la segmentación es que el tiempo desde que se ubica un canasto de ropa sucia en la lavadora hasta que es secada, doblada y guardada no es más corto con la segmentación; la razón por la cual la segmentación es más rápida para varias cargas es que todo está trabajando en paralelo, de esa manera más cargas son finalizadas por hora. La segmentación incrementa el *rendimiento* (en inglés thruoghput) de nuestro sistema de lavandería. Por eso, la segmentación no reduce el tiempo en que se completa una única carga de lavado, sin embargo, cuando hay muchas cargas de lavado, la mejora en el rendimiento decrementa el tiempo total para completar el trabajo.

Si todas las etapas toman la misma cantidad de tiempo y hay suficiente trabajo para hacer, entonces la ganancia debido a la segmentación es igual al número de etapas o segmentos, en este caso cuatro: lavado, secado, doblado y guardado. Por lo tanto, la lavandería segmentada es potencialmente cuatro veces más rápida que la no segmentada: 20 cargas podrían llevar 5 veces el tiempo de una sola carga, mientras que 20 cargas en la lavandería secuencial llevaría 20 veces un solo llavado completo. La Figura 1 muestra que es sólo 2,3 veces más rápido, debido a que sólo se muestran cuatro cargas de ropa. Notar que al comienzo y al final del trabajo en la versión segmentada de la Figura 1, no todos los recursos son utilizados. Esta situación de preparación y detención afectan el desempeño cuando el número de tareas no es grande, comparado con el número de etapas en la segmentación. Si el número de cargas es mucho mayor que 4, entonces las etapas estarán ocupadas casi todo el tiempo y el incremento del rendimiento será cercano a 4.

El mismo principio se aplica en los procesadores al dividir la ejecución de las operaciones. Las instrucciones MIPS típicamente toman 5 etapas:

1. Recuperar la instrucción desde la memoria.
2. Leer los registros mientras se decodifica la instrucción.
   El formato regular de las instrucciones permite leer y decodificar al mismo tiempo.
3. Ejecutar la instrucción o calcular una dirección en memoria.
4. Acceder a un operando en la memoria de datos.
5. Escribir el resultado en un registro.

En esta sección se analiza la segmentación del MIPS con cinco etapas. El siguiente ejemplo muestra que la segmentación acelera la ejecución de las instrucciones de la misma manera que se acelera en la lavandería.

Ejemplo. Desempeño de único ciclo vs segmentado. Para hacer las cosas concretas, se creó un procesador segmentado. Este ejemplo se concentra en 8 instrucciones: load word(ld), storage word(sw), add(add), subtract(sub), AND(and), OR(or), set less then(slt) y branch on equal(be).

Compare el tiempo promedio entre las instrucciones de la implementación en la cual todas las instrucciones requieren un ciclo, con la implementación segmentada. Los tiempos de operación para las principales unidades funcionales en este ejemplo son: 200ps para accceder a memoria, 200ps para la ALU, 100ps para leer o escribir registros. En el modelo de un único ciclo, cada instrucción toma exactamente un ciclo de reloj, de esa manera el ciclo de reloj debe ser ajustado para soportar la instrucción más lenta.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURA 2. **Tiempo total para las instrucciones, calculado a partir del tiempo de cada componente.** Este cálculo asume que los multiplexores, unidad de control, acceso al PC y la extensión de signo no tiene demora.

Respuesta. La Figura 2 muestra el tiempo requerido por cada una de las ocho instrucciones. El diseño de un único ciclo debe soportar la instrucción más lenta – en la Figura 2 es la instrucción lw – por lo que el tiempo requerido para cada instrucción es 800ps. De manera similar a la Figura 1, la Figura 3 compara la ejecución segmentada y no segmentada de tres instrucciones load word. De esa manera, el tiempo que transcurre entre el comienzo de la primer instrucción y el comienzo de la cuarta en el caso no segmentado es $3 \times 800ps = 2400ps$.

Cada una de las etapas del procesador segmentado toma un solo ciclo, de manera que el ciclo debe soportar la ejecución de la unidad funcional más lenta. Al igual que en el caso del procesador de un ciclo por instrucción, que tomó el caso de la instrucción más lenta, el procesador segmentado debe tomar el tiempo de la unidad más lenta, $200ps$, a pesar que algunas etapas sólo demoran $100ps$. De todos modos, la segmentación ofrece un desempeño cuatro veces mejor: el tiempo entre la primer y cuarta instrucción es $3 \times 200ps = 600ps$. □

Es posible llevar la discución anterior a una fórmula. Si las etapas están perfectamente balanceadas, entonces el tiempo entre instrucciones del procesador segmentado – asumiendo condiciones ideales – es igual a:

$$Tiempo\,entre\,instrucciones_{segmentado} = \frac{Tiempo\,entre\,instrucciones_{no\,segmentado}}{Nro\,de\,etapas}$$

Bajo condiciones ideales y con un gran número de instrucciones, la aceleración por la segmentación es aproximadamente igual al número de etapas del procesador: si el procesador tiene cinco etapas, es aproximadamente cinco veces más rápido.

La fórmula sugiere que un procesador de cinco etapas debiera ofrecer una mejora por aproximadamente cinco sobre los $800ps$ del tiempo no segmentado, o un ciclo de $160ps$. El ejemplo sin embargo muestra que las etapas pueden estar fuera de balance. Es más, la segmentación puede introducir algunas demoras adicionales, las cuales serán aclaradas más adelante. Por esta razón, el tiempo por instrucción en el procesador segmentado exederá el mínimo posible y alcanzará una aceleración menor al número de etapas.

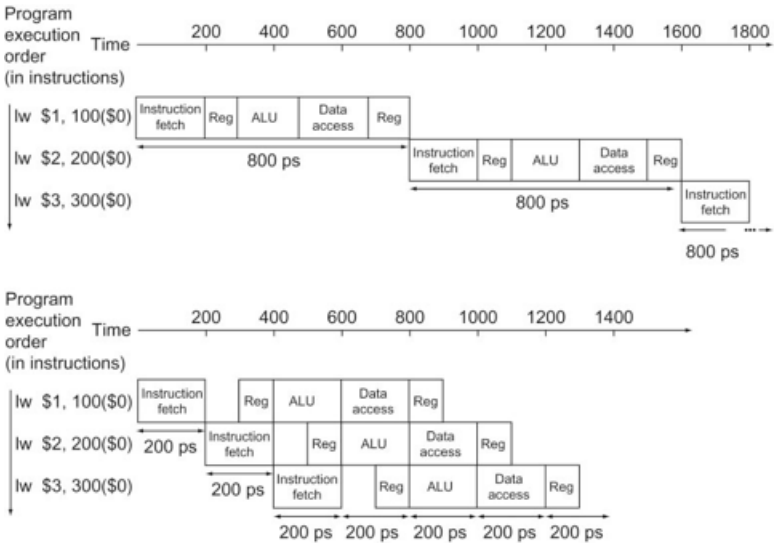

FIGURA 3. **Ejecución no segmentada vs segmentada.** Ambos usan los mismos componentes de hardware, con los tiempos listados en la Figura 2. En este caso se observa una aceleración de cuatro en el tiempo promedio entre instrucciones, de $800ps$ se reduce a $200ps$. Compare esta figura con la Figura 1. Para el caso de la lavandería, se asume que todas las etapas son iguales. Si la secadora fuese más lenta, entonces la etapa de la secadora podría establecer el tiempo por etapa. El tiempo por etapa del procesador también está limitado por el recurso más lento, sea el funcionamiento de ALU o el acceso a memoria. Se asume que la escritura sobre el banco de registros ocurre en la primer mitad del tiempo del ciclo de reloj y la lectura al banco de registros ocurre en la segunda mitad.

A pesar que para los ejemplos se estimó una mejora por cuatro, no se vió reflejada en el tiempo total de ejecución para tres instrucciones: fue $1400ps$ vs $2400ps$. Desde luego esto se debe a que la cantidad de instrucciones no es suficiente. ¿Qué hubiera ocurrido si incrementamos el número de instrucciones?. Se podrían extender las figuras previas a 1.000.003 instrucciones, agregar 1.000.000 de instrucciones en el ejemplo segmentado, cada instrucción agrega 200ps al tiempo total de ejecución, que ahora sería $1{,}000{,}000 \times 200\,ps +$

$1400\,ps$ lo que saría $200{,}001{,}400\,ps$. En el ejemplo sin segmentación si agregamos 1,000,000 de instrucciones más, como cada instrucción toma 800 ps el tiempo total de ejecución de estas instrucciones sería: $1{,}000{,}000 \times 800\,ps + 2400\,ps = 800{,}002{,}400\,ps$. Bajo estas condiciones, la relación de los tiempos totales de ejecución de los programas reales sobre procesadores segmentados y no segmentados es cercana a la relación entre los tiempos para una instrucción:

$$\frac{800{,}002{,}400ps}{200{,}001{,}400ps} \backsimeq \frac{800ps}{200ps} = 4{,}00$$

La segmentación mejora el desempeño incrementando el rendimiento de instrucción, como opuesto a decrementar el tiempo de ejecución de una instrucción individual, pero el rendimiento de instrucción es una métrica importante porque los programas reales ejecutan miles de millones de instrucciones.

2. Diseño del conjunto de instrucciones para segmentación

A pesar de que la explicación de segmentación vista hasta ahora es introductoria, es suficiente para analizar el diseño del conjunto de instrucciones del MIPS.

Primero, todas las instrucciones del MIPS tiene el mismo tamaño. Esta restricción facilita mucho la recuperación de instrucciones en la primer etapa y para la decodificación en la segunda etapa. En un conjunto de instrucción como del X86, donde las instrucciones varían entre 1 y 15 bytes, la segmentación es bastante más complicada. Las implementaciones más recientes de la arquitectura x86, en realidad traducen las instrucciones x86 en instrucciones más simples que se parecen a las del MIPS y luego las operaciones simples son segmentadas en lugar de las instrucciones x86 nativas.

Segundo, MIPS solo tiene unos pocos formatos de instrucción, donde los campos de los registros fuente se posicionan siempre en el mismo lugar de cada instrucción. Esta simetría significa que en la segunda etapa es posible comenzar a leer el banco de registros al mismo tiempo que el hardware determina de que tipo de instrucción se trata. Si los formatos de instrucción del MIPS no fuesen simétricos podría ser necesario dividir la etapa 2, resultando en un procesador de seis etapas. Más adelante se verán las consecuencias en el desempeño que tienen los procesadores de más etapas.

Tercero, los operandos en memoria sólo aparecen en las instrucciones de load o store. Esta restricción significa que es posible usar la etapa de ejecución para calcular la dirección en memoria y luego en la siguiente etapa acceder a la memoria. Si fuese posible mantener los operandos en memoria, como el x86 lo hace, las etapas 3 y 4 debieran expandirse con etapas de dirección, etapa de memoria y recién luego, etapa de ejecución.

Cuarto, como se sabe los operandos deben estar alineados en memoria. Eso evita que una única instrucción de memoria requiera dos accesos a memoria de dato; el operando requerido siempre podrá ser transferido entre memoria y procesador en una única etapa.

3. Los riesgos en la segmentación

En segmentación hay situaciones en las cuales la próxima instrucción no puede ejecutar en el siguiente ciclo de reloj. Estos eventos son llamados *riesgos* y pueden ser de tres tipos distintos.

3.1. Riesgo estructural.

El primer tipo de riesgo es llamado *riesgo estructural*.

> ***RIESGO ESTRUCTURAL:*** Es cuando la instrucción planificada no puede ejecutar en el ciclo previsto porque el hardware no soporta la combinación de instrucciones establecidas para ejecutar.

Significa que el hardware no puede soportar, en un mismo ciclo de reloj, la combinación de instrucciones que se desea ejecutar. Un riesgo estructural en la lavandería podría ocurrir al usar una máquina que combina dos funciones: lavado y secado, en lugar de tener una lavadora y una secadora por separado. En ese caso la cuidadosa planificación de la segmentación no sería de utilidad.

Como se dijo antes, el conjunto de instrucciones del MIPS ha sido diseñado para la segmentación, lo cual ayuda a los diseñadores a evitar riesgos estructurales cuando diseñan el procesador. Supongamos, por ejemplo, que se cuenta con una única memoria en lugar de dos. Si la segmentación de la Figura 3 tuviese una cuarta instrucción, se podría ver que en el mismo ciclo de reloj la primer instrucción está accediendo a datos desde memoria mientras que la cuarta instrucción está recuperando una instrucción de la misma memoria. Sin tener dos memorias, este procesador podría tener un riesgo estructural.

3.2. Riesgo de dato.

> ***RIESGO DE DATO:*** También llamado *riesgo de dato en segmentación*. Es cuando la instrucción planificada no puede ejecutar en el ciclo previsto porque el dato necesario para ejecutar la instrucción aún no está disponible.

Un riesgo de dato ocurre cuando el procesador debe ser atascado porque una etapa debe esperar para que otra termina. Suponga que en la etapa que se dobla la ropa se encuentra una sola media sin su compañera. Una estrategia posible es volver hasta su casa y buscar entre su ropa menos usada para ver si puede encontrar la compañera perdida. Obviamente mientras hace esta búsqueda, las tandas de ropa deben esperar, las que han completado el secado y están listas para ser dobladas, de la misma manera que aquellas que han completado el lavado y están listas para el secado.

En un procesador segmentado, los riesgos de datos surgen de las dependencias de una instrucción con una instrucción previa que aún está dentro del procesador (una relación que realmente no existe cuando se lava ropa). Por ejemplo, supongamos que una instrucción de suma es seguida inmediatamente por una instrucción de resta que usa el resultado de la suma ($s0):

Código 1. Riesgo de dato real

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Si nada interviene, un riesgo de dato podría atascar severamente al procesador. La instrucción de suma no escribe su resultado hasta el quinto ciclo, por lo que la resta debe aguardar tres ciclos en una misma etapa. Aunque se podría confiar en los compiladores para eliminar todos estos riesgos, los resultados podrían no ser satisfactorios. Estas dependencias ocurren demasiado frecuentemente y la demora es demasiado larga para esperar que el compilador resuelva este dilema.

Los riesgos de datos ocurren por las dependencias de datos entre las instrucciones que están al mismo tiempo dentro de la tubería. Más específicamente, se asume que la ejecución de la instrucción i debe preceder a ejecución de la instrucción j, de manera que se respete la semántica del orden del programa. Se dice que la instrucción j mantiene una dependencia de dato con la instrucción i, si la salida de la instrucción i es un operando de entrada para la instrucción j. Esta dependencia es llamada RAW por *Read-After-Write* en inglés, que en español se podra traducir como *lectura después de escritura*. Considerar el siguiente ejemplo:

Código 2. Distintos tipos de riesgos

```
add $a3, $t4, $t7    // instruccion i
sub $t0, $a3, $t4    // instruccion i+1
add $t7, $t0, $a3    // instruccion i+2
```

Hay tres dependencias RAW:

1. La instrucción $i+1$ tiene una dependencia RAW con la instrucción i porque uno de sus registros de entrada, el registro $a3, es el registro de salida de la instrucción i.
2. La instrucción $i+2$ tiene una dependencia RAW con la instrucción i, por la misma razón.
3. La instrucción $i+2$ tiene una dependencia RAW con la instrucción $i+1$, porque uno de sus registros de entrada, el registro $t0, es el registro de salida de la instrucción $i+1$.

El registro $t4, que es un registro fuente común para las instrucciones i e $i+1$ no produce dependencia.

Otra forma de dependencia, que en español sería llamada *escritura después de lectura*, en inglés es *Write-After-Read*, o WAR (también llamada anti dependencia) aparece en Código 2, ya que el registro $t7, un operando de entrada de la instrucción i, es el registro de salida de la instrucción $i+2$. Esto no causará ningún problema en nuestro procesador segmentado simple — ni tendrá una dependencia de salida o *escritura después de escritura* (*Write-After-Write*) WAW, la cual ocurre cuando dos instrucciones tienen el mismo registro de salida. Sin embargo estas dependencias deben ser resueltas en las arquitecturas más complejas que veremos más adelante. Hay que destacar en este punto que la dependencia RAW, es un aspecto propio del procedimiento: el algoritmo (programa) que está corriendo lo dictamina. Por otro lado, WAR y WAW son dependencias de nombre. Si el compilador hubiese sido más inteligente, o si hubiesen más registros, una instancia de $t7 podría haber sido llamada por ejemplo $t8, de esa manera el WAR podría haber sido evitado.

La solución principal para los riesgos RAW está en la observación de que no es necesario esperar a que la instrucción termine para tratar de resolverlos. Para las secuencias de Código 1 y Código 2, tan pronto como la ALU calcula el resultado de una operación, es posible proveerlo como entrada para la siguiente operación. El agregado de hardware adicional que recupere anticipadamente el item faltante desde los recursos internos es llamado *adelantamiento* o *forwarding*.

Ejemplo. Adelantamiento con dos instrucciones. Para las dos instrucciones de Código 1, mostrar que etapas del procesador pueden ser conectadas por adelantamiento. Usando el cuadro de la Figura 4 es posible representar el camino de datos durante las cinco etapas del procesador. Aliñe una copia del camino de datos para cada una de las instrucciones, similar a las etapas de la lavandería en la Figura 1.

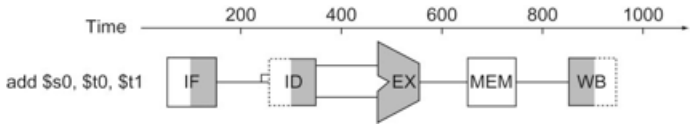

FIGURA 4. **Representación gráfica de la segmentación de instrucción, en espíritu similar a la segmentación de la lavandería en la Figura 1.** Aquí se usan los símbolos que representan los recursos físicos con la abreviatura de la etapa de segmentación usada a lo largo del apunte. Los símbolos para las cinco etapas: IF para la etapa de recuperación (fetch) de la instrucción, con la caja que representa la memoria de instrucción; ID para la etapa de decodificación/lectura del banco de registros, con el dibujo mostrando el banco de registros siendo leído; EX para la etapa de ejecución, con el dibujo representando la ALU; MEM para la etapa de acceso a la memoria, con la caja representando la memoria de datos; WB para la etapa de escritura de resultado, con el dibujo mostrando el banco de registros siendo escrito. El sombreado indica que el elemento es usado por la instrucción. En este caso, MEM tiene un fondo blanco porque add no accede a la memoria de datos. El sombreado en la mitad derecha del baco de registros o de la memoria significa que el elemento es leído en esa etapa, el sombreado en la parte izquierda significa que es escrito en esa etapa. Por eso, la mitad derecha de ID está sombreada en la segunda etapa porque el banco de registros es leído, y la mitad izquierda de WB está sombreada en la quinta etapa porque el banco de registros es escrito.

Respuesta. La Figura 5 muestra la conexión para adelantar el valor $s0 después de la etapa de ejecución de la instrucción de suma como entrada a la etapa de ejecución de la instrucción sub.

En esta representación gráfica de los eventos, los caminos de adelantamiento son válidos solamente si la etapa destino está después en el tiempo que la etapa fuente. Por ejemplo no puede haber un camino de adelantamiento desde la salida de la etapa de acceso a memoria en la primer instrucción hacia la entrada en la etapa de ejecución de la siguiente, dado que significaría ir hacia atrás en el tiempo.

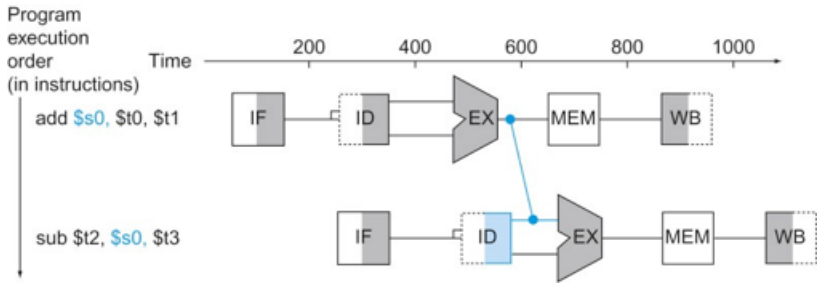

FIGURA 5. **Representación gráfica del adelantamiento** La conexión muestra el camino de adelantamiento desde la salida de la etapa EX del add hacia la entrada de la etapa EX del sub, reemplazando el valor del registro $s0 leído en la segunda etapa del sub.

El adelantamiento funciona bien y será descripto en detalle más adelante. Pero, sin embargo, no puede prevenir todos los atascos del procesador. Por ejemplo supongamos que la primer instrucción fue un load en el registro $s0 en lugar de un add. Como es posible imaginarse mirando la Figura 5 el dato deseado podría estar disponible recién después de la cuarta etapa de la primer instrucción en la dependencia, lo cual es demasiado tarde para la entrada de la tercer etapa del sub.

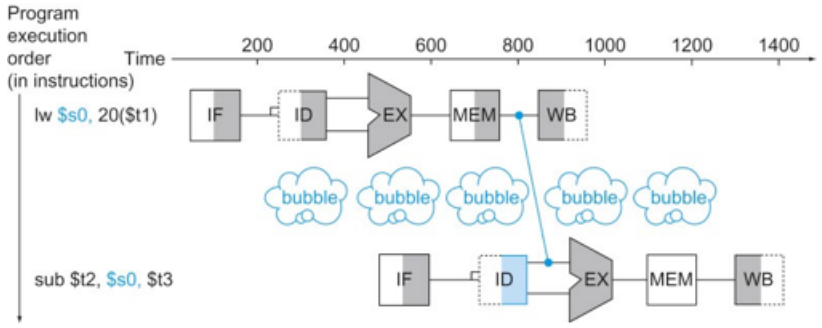

FIGURA 6. **Cuando se planifica una instrucción de tipo R, que le sigue a un load y usa el dato recuperado de memoria, es necesario un atasco; aunque ya se cuente con adelantamiento.** Sin el atasco, el camino desde la salida de la etapa de acceso a la memoria a la entrada de la etapa de ejecución, debiera ir hacia atrás en el tiempo, lo cual es imposible. Esta figura realmente es una simplificación, debido a que no es posible saber si o no el atasco será necesario, hasta después que la instrucción sub es recuperada y decodificada . Más adelante se brindan detalles de lo que ocurre realmente cuando existe un riesgo.

RIESGO DE DATO LOAD-USE: Una forma específica de riesgo de dato, en la cual, el dato que es cargado por la instrucción load es requerido por la siguiente instrucción y aún no se ha hecho disponible.

Por eso, aún con adelantamiento, es necesario atascar una etapa si hay un *riesgo de dato load-use*, como lo muestra la Figura 6. Esta figura muestra un importante concepto de segmentación, llamado oficialmente *atasco del pipe* o *tubería*, pero también es conocido con el apodo de *burbuja*. El atasco también se verá en otra etapa del procesador. Más adelante se mostrará cómo se pueden manejar casos difíciles como este, usando ya sea detección de riesgo y atasco o software que reordene el código para tratar de evitar el atasco load-use de tubería, como este ejemplo lo ilustra.

ATASCO DE TUBERÍA: También llamado *burbuja*.
Es un atasco iniciado para resolver un riesgo.

Ejemplo. Reordenamiento de código para evitar atasco de tubería. Considere el siguiente segmento de código en C:

```
a = b + e;
c = b + f;
```

Aquí está el código MIPS generado para ese segmento, asumiendo que todas las variables están en memoria y que son direccionables como desplazamientos de $t0:

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1, $t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add     $t5, $t1, $t4
sw      $t5, 16($t0)
```

Encuentre los riesgos en el segmento de código anterior y reordene las instrucciones para evitar cualquier atasco de tubería.

Respuesta. Ambas instrucciones add tienen riesgo por sus respectivas dependencias con la instrucción lw que le precede inmediatamente. Notar que el adelantameinto elimina varios otros riesgos potenciales, incluyendo la dependencia del primer add con el primer lw y cualquier riesgo con la instrucción store. Subiendo el tercer lw, poniéndolo como tercera instrucción, se eliminan ambos riesgos:

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
lw      $t4, 8($t0)
add     $t3, $t1, $t2
sw      $t3, 12($t0)
add     $t5, $t1, $t4
sw      $t5, 16($t0)
```

En un procesador segmentado con adelantamiento, la secuencia reordenada completará en dos ciclos menos que la versión original.

El adelantamiento muestra otros detalles de la arquitectura del MIPS, además de los ya mencionados en la Sección 2. Cada instrucción MIPS escribe a lo más un resultado y lo hace en la última etapa de la tubería. El adelantamiento es más complicado si hubieran múltiples resultados por instrucción para adelantar o si fuese necesario escribir un resultado temprano en la ejecución de la instrucción.

Elaboración. El nombre "adelantamiento" proviene de la idea de que el resultado es pasado hacia adelante desde una instrucción anterior a una posterior. "Cortocircuito" surge por dirigir el resultado que va al banco de registro, hacia la unidad que lo requiere.

3.3. Riesgo de Control. El tercer tipo de riesgo es llamado **riesgo de control**, surge de la necesidad de tomar una decisión en base al resultado de una instrucción mientras otras se están ejecutando.

Supongamos que en el caso de la lavandería se deba lavar la ropa de un equipo de rugby. Dependiendo de cuan percudida esté la ropa, es necesario determinar la cantidad de jabón y la temperatura del agua adecuadas para que los equipos queden limpios, pero sin exederse porque se puede dañar la tela. En la versión segmentada es necesario esperar que termine la segunda etapa (de secado) para determinar si se deben cambiar la cantidad de jabón y temperatura del agua elegidas. ¿Cómo proceder?.

> ***RIESGO DE CONTROL:*** También llamado *riesgo por salto condicional*. Es cuando la instrucción planificada no puede ejecutar en el ciclo de reloj previsto porque la instrucción que fue recuperada no es la que se necesita; es decir, el flujo de direcciones de instrucción no es el esperado por el procesador.

Aquí está la primera de dos soluciones posibles a los riesgos de control en la segmentación de la lavandería y su equivalente para los procesadores.

Atascar: Simplemente opear secuencialmente hasta que el primer lote de ropa esté seca (se verifica la limpieza) y luego se repite el lavado hasta que se consigue la combinación correcta.

Esta opción conservadora funciona bien, pero es lenta.

La tarea de decisión equivalente en el procesador es la instrucción de salto condicional (branch). Notar que se debe comenzar a recuperar en el siguiente ciclo del reloj la instrucción que le sucede al branch. Más allá de eso, la tubería posiblemente no puede saber cual debiera ser la siguiente instrucción, debido a que solamente se ha recibido la instrucción de salto desde la memoria!. Al igual que en la lavandería, una posible solución sería parar inmediatamente después de recuperar una instrucción de salto, hasta que el procesador determine el resultado del salto y conozca desde que dirección de instrucción hacer la recuperación.

Asumiendo que se agrega suficiente hardware como para comparar el contenido de los registros, calcular la dirección destino del salto y actualizar el PC durante la segunda etapa de la tubería. Aún con este hardware extra, la tubería que maneja saltos condicionales se vería como en la Figura 7. La instrucción lw, ejecutada si el salto no es tomado, se atasca un ciclo de reloj de 200ps antes de comenzar.

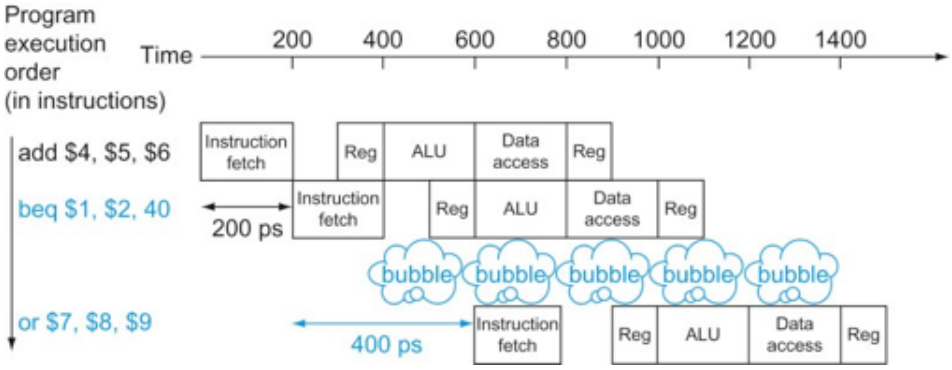

FIGURA 7. **Segmentación mostrando una solución al riesgo de control mediante un atasco por cada salto condicional.** Este ejemplo asume que el salto condicional es tomado y que la instrucción destino del salto es la instrucción or. Hay un atasco de un ciclo en la tubería, o burbuja, después del salto. En realidad, el proceso de crear un atasco, es algo más complicado, como se verá más adelante. El efecto en el desempeño, sin embargo, es igual a que si una burbuja fuese insertada.

Ejemplo. Desempeño del "Atasco con cada branch". Estimar el impacto en los *ciclos de reloj por instrucción* (CPI) debido a los atascos por branch. Asuma que las demás instrucciones tienen un CPI de 1.
Respuesta. Estudios realizados indican que el 17 % de las instrucciones en los programas son saltos condicionales. Considerando que las otras instrucciones que se ejecutan tiene un CPI de 1 y que las instrucciones de salto condicional, requieren un ciclo extra por los atascos, se puede ver un CPI de 1,17. En consecuencia hay un deterioro en el desempeño de 1,17 contra el CPI ideal. □

Si no es posible resolver el salto en la segunda etapa, como ocurre en muchos procesadores grandes, entonces se observará un deterioro mayor si hay un atasco por cada branch. El costo de esta opción es demasiado alto para la mayoría de las computadoras y motiva una segunda solución para los riesgos de control:

> ***PREDICCIÓN:*** Si ya se conoce una fórmula correcta para lavar los equipos, entonces se puede predecir que trabajará correctamente y se puede comenzar a lavar la segunda tanda mientras la primera se está secando.

Esta alternativa no deteriora el desempeño cuando es correcta. Cuando es erronea, es necesario volver a lavar la carga que fue lavada mientras se aguardaba la decisión.

Las computadoras también usan predicción para manejar los saltos. Una aproximación simple es predecir siempre los saltos como no tomados. Cuando es correcta, el procesador funciona a máxima velocidad. Solamente cuando los saltos son tomados, la tubería se detiene, la Figura 8 muestra tal ejemplo.

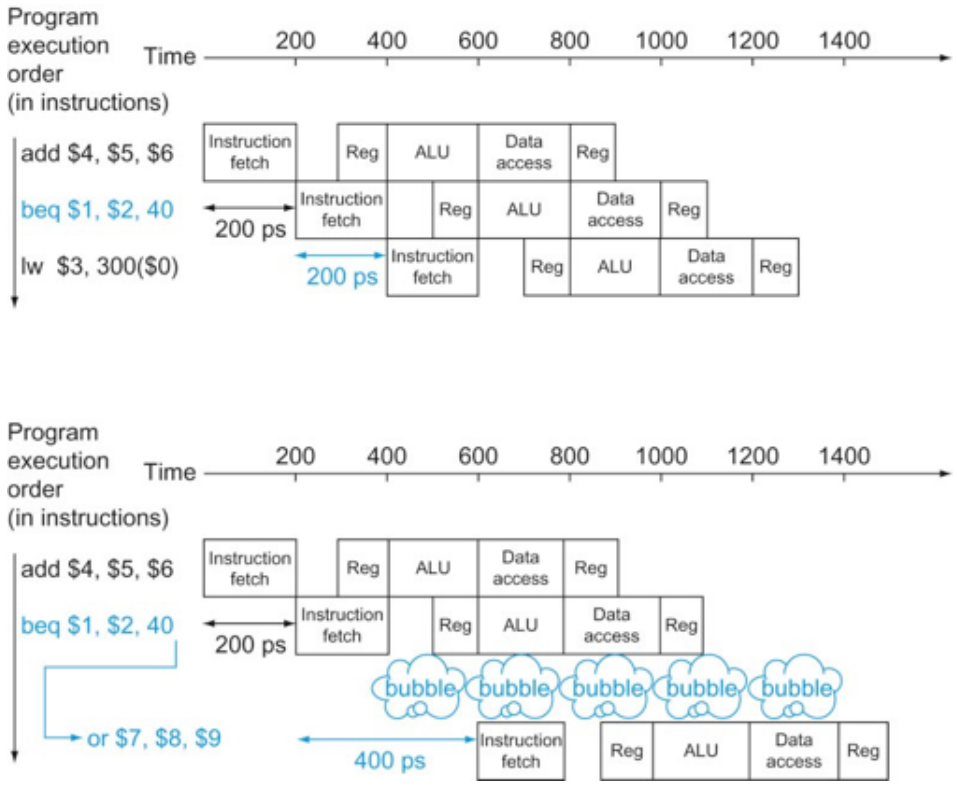

FIGURA 8. **Una solución a los riesgos de control es predecir que los saltos serán no tomados.** La parte superior muestra la tubería cuando el salto es no tomado. La parte inferior muestra la tubería cuando el salto es tomado. Como se resaltó en la Figura 7, la inserción de una burbuja de esta manera simplifica lo que realmente ocurre, al menos durante el primer ciclo inmediatamente después del branch. Luego se analizarán más detalles.

PREDICCIÓN DE SALTOS: Es un método para resolver un riesgo de control que asume que ocurrirá una alternativa determinada y procede según esa predicción más que aguardar a verificar el resultado real.

Una versión más sofisticada de predicción de salto debiera tener que predecir algunos de los saltos como tomados y otros como no tomados. En la analogía con la lavandería, la ropa de todos los días podría requerir una fórmula para más suciedad, mientras que la ropa de salir puede tener otra más suave. En el caso de la programación, suele ocurrir que en la base de los loops aparecen saltos hacia atrás (direcciones más bajas), hasta el comienzo del loop. Debido a que probablemente sean tomados y que saltan hacia atrás, se podría predecir siempre como tomados los saltos hacia las direcciones más bajas.

Estas aproximaciones rígidas a la predicción de saltos se basan en el comportamiento típico y no tienen en cuenta las características propias de cada salto individual. Los

predictores *dinámicos* de hardware, en claro contraste, hacen su predicción dependiendo del comportamiento de cada branch y puede cambiar su predicción para un salto durante la vida del programa. Siguiendo con la analogía, en la predicción dinámica, una persona podría mirar cuan sucia está la ropa y adivinar la fórmula, ajustando la siguiente conjetura dependiendo del éxito de las conjeturas recientes.

Una aproximación popular a la predicción dinámica de saltos, es mantener la historia de cada salto como tomado o no tomado y de esa manera, usando el comportamiento del pasado reciente predecir el futuro. Como se verá más adelante, es relevante la cantidad y tipo de historia empleada; sorprendentemente los predictores dinámicos pueden predecir los saltos con una exactitud de más del 90 %. Cuando la conjetura es incorrecta, es el control de la tubería quien debe asegurarse que no tengan efecto las instrucciones que le siguen al salto estimado erroneamente; además se debe recomenzar la tubería desde la dirección correcta del salto. En el caso de la lavandería, se debe detener la carga de más ropa, de manera que sea posible volver a comenzar con las cargas que fueron predichas incorrectamente.

Como en el caso de las demás soluciones a los riesgos de control, las tuberías más largar exacerban el problema, en este caso incrementando el costo de la mala predicción. Las soluciones a los riesgos de control son descriptas con más detalle más adelante.

Elaboración. Hay una tercera aproximación a los riesgos de control, llamada decisión demorada. En la analogía de la lavandería, cada vez que se debe tomar una decisión sobre el lavado, se coloca una tanda de ropa que no sea de rugby en la lavadora mientras se espera que se termine de secar la ropa de rugby. Mientras que una buena cantidad de ropa sucia no se perjudique con la combinación aplicada, esta solución funciona bien.

Denominada *salto demorado* en computación, como se dijo antes, esta es la solución adoptada por la arquitectura MIPS. El salto demorado siempre ejecuta la instrucción que sigue en la secuencia, por lo que el salto toma lugar después de la demora de una instrucción. Esto se oculta al programador del MIPS porque el ensamblador puede automáticamente reubicar las instrucciones para que el salto se comporte de la menera deseada por el programador. El software MIPS ubica una instrucción inmediatamente después de la instrucción de salto demorado que no sea afectada por el salto y un salto tomado cambia la dirección de la instrucción que le sigue a esta instrucción segura. En el código ejemplo, la instrucción add anterior al salto en la Figura 7 no afecta el salto y puede ser movida después del salto para ocultar completamente el salto demorado. Debido a que los saltos demorados son útiles cuando los saltos son cortos, ningún procesador usa salto demorado de más de un ciclo. Para saltos demorado más largos, la predicción de saltos basada en hardware es usada.□

Latencia en segmentación: Es el número de etapas en la tubería o el número de etapas entre dos instrucciones durante la ejecución.

Entendiendo el desempeño de los programas. Sin considerar el sistema de memoria, la operación del procesador segmentado es usualmente el factor más importante en la determinación del CPI de un procesador y por ende de su desempeño. Como se verá más adelante, entender el desempeño de un procesador segmentado de despacho múltiple es algo complejo y requiere entender más que solamente las situaciones surjidas en un procesador

segmentado simple. Más allá de eso, los riesgos estructurales, de control y de dato, siguen siendo importante para ambos tipos de procesadores: los de segmentación simple y los más sofisticados.

En la segmentación moderna, los riesgos estructurales giran alrededor de la unidad de punto flotante, la cual puede no estar completamente segmentada, por su parte los riesgos de control son normalmente más un problema de los programas de enteros, los cuales tienen una mayor frecuencia de branches, además de branches menos predecibles. Los riesgos de dato pueden ser un cuello de botella para el desempeño en ambos tipos de programas: enteros y de punto flotante. Normalmente es más fácil tratar con riesgos de datos en programas de punto flotante por la baja frecuencia de saltos y modelos de acceso a memoria más regulares, que permiten al compilador tratar de planificar instrucciones para evitar riesgos. Es más difícil realizar esa optimización en programas de enteros que tienen acceso más irregular a la memoria, debido al uso de punteros. Existen hardware y compiladores más ambiciosos al reducir las dependencias de datos por medio de la planificación. □

Verifique. Para cada una de las siguientes secuencias de código, establezca si debe haber atasco, se puede evitar el atasco solamente con adelantamiento, o se pueden ejecutar sin atasco ni adelantamiento.

Secuencia 1	Secuencia 2	Secuencia 3

lw $t0,0($t0)
add $t1,$t0,$t0
	add $t1,$t0,$t0
	addi $t2,$t0,#5
	addi $t4,$t1,#5
		addi $t1,$t0,#1
		addi $t2,$t0,#2
		addi $t3,$t0,#2
		addi $t3,$t0,#4
		addi $t5,$t0,#5

□

La **segmentación** incrementa el número de instrucciones ejecutadas simultáneamente y la tasa de instrucciones iniciadas y completadas. La segmentación no reduce el tiempo que requiere finalizar una instrucción particular, lo que suele llamarse *latencia*. Por ejemplo, la tubería de cinco etapas requiere cinco ciclos para que una instrucción se complete. La segmentación mejora el *rendimiento* (*throughput*) más que la *latencia* o *tiempo de ejecución* de las instrucciones

El conjunto de instrucciones puede simplificar o hacer más difícil la tarea de los diseñadores del procesador segmentado, quienes ya deben lidiar con los riesgos estructural, de control y de dato. La predicción de saltos y el adelantamiento, colaboran en hacer más veloz una computadora manteniendo el comportamiento correcto.

Referencias

[1] Patterson, David A. and Hennessy, John L., Computer Organization and Design, Fifth Edition: The Hardware/Software Interface, Morgan Kaufmann Publishers Inc., 2013