Autoevaluación

¿Qué excepción debería reconocerse en primer lugar en esta secuencia?

```
1. add  $1, $2, $1    # desbordamiento aritmético
2. XXX  $1, $2, $1    # instrucción no definida
3. sub  $1, $2, $1    # error del hardware
```

4.10 Paralelismo y paralelismo a nivel de instrucciones avanzado

Se advierte con antelación que esta sección proporciona una visión general breve de temas fascinantes pero avanzados. Si se desea aprender más detalles, se debería consultar el libro más avanzado, *Arquitectura de Computadores: Un enfoque cuantitativo,* ¡donde el material cubierto por las siguientes 13 páginas se amplía a más de 200 páginas (incluyendo apéndices)!

La segmentación aprovecha el paralelismo potencial entre las instrucciones. Este paralelismo se denomina **paralelismo a nivel de instrucciones (*instruction-level Parallelism*, ILP)**. Existen dos estrategias básicas para incrementar la cantidad potencial del ILP. La primera consiste en aumentar la profundidad del pipeline para solapar la ejecución de más instrucciones. Empleando la analogía de la lavandería y suponiendo que el ciclo de lavado fuera más largo que los otros, se podría dividir nuestra lavadora en tres máquinas que realizaran los pasos de lavado, aclarado y centrifugado de la máquina tradicional. De este modo, convertiríamos el pipeline de cuatro etapas en un pipeline de seis etapas. Para conseguir la máxima ganancia en velocidad, tanto en el caso del procesador como en el caso de la lavandería, los pasos restantes se deberían volver a equilibrar para que tuvieran la misma longitud. La cantidad de paralelismo que se aprovecharía es mayor, puesto que se solapa la ejecución de más instrucciones. Las prestaciones también serían potencialmente mayores puesto que se puede reducir el ciclo de reloj.

Paralelismo a nivel de instrucciones: paralelismo entre instrucciones.

La segunda estrategia consiste en replicar los componentes internos del computador para poder ejecutar múltiples instrucciones dentro de cada etapa de segmentación. El nombre general para esta técnica es **ejecución múltiple (*multiple issue*)**. Una lavandería con ejecución múltiple reemplazaría muestra lavadora y secadora, por ejemplo, por tres lavadoras y tres secadoras. También se tendrían que reclutar más asistentes para doblar y guardar en el mismo tiempo una cantidad de ropa tres veces mayor. El inconveniente de la estrategia es que se requiere trabajo adicional para conseguir mantener todas las máquinas ocupadas y para transferir las cargas de ropa de una etapa de segmentación a la siguiente.

Ejecución múltiple: esquema que permite lanzar varias instrucciones para ejecutarse durante un ciclo de reloj.

Ejecutar varias instrucciones por etapa permite que la frecuencia de instrucciones ejecutadas sea mayor que la frecuencia del reloj, o, dicho de otra manera, que el CPI sea menor que 1. En ocasiones es beneficioso invertir la métrica del CPI y usar el IPC (*instructions per clock cycle*, instrucciones ejecutadas por ciclo de reloj). Por ejemplo, un microprocesador con ejecución múltiple de hasta cuatro instrucciones que funcione a 4 GHz podrá llegar a ejecutar instrucciones a una velocidad pico de 16 mil millones de instrucciones por segundo, y tener en el mejor de los casos un CPI de 0,25, o un IPC de 4. Suponiendo un pipeline de cinco etapas, este procesador tendría en todo momento hasta 20 instrucciones válidas en ejecución. Los microprocesadores de gama alta de hoy en día tratan de ejecutar entre tres y seis nuevas instrucciones (4 vías) en cada ciclo de

Ejecución múltiple con planificación estática: estrategia de implementación de un procesador con ejecución múltiple en la que el compilador toma muchas decisiones antes de la ejecución del programa.

Ejecución múltiple con planificación dinámica: estrategia de implementación de un procesador con ejecución múltiple en la que el propio procesador toma muchas decisiones durante la ejecución del programa.

Ranuras de ejecución: posiciones desde las que las instrucciones pueden ser enviadas a ejecutar en cada ciclo de reloj; usando una analogía, podrían corresponder a las posiciones de salida para una carrera.

reloj. Sin embargo, es habitual encontrar muchas restricciones en el tipo de instrucciones que se pueden ejecutar de forma simultánea y diferencias en la forma de tratar las dependencias que se van encontrando entre las instrucciones.

Hay dos formas fundamentales de implementar un procesador de ejecución múltiple. La diferencia principal entre ellas es la forma de repartir el trabajo entre el compilador y el hardware. Puesto que esta división del trabajo determina si las decisiones se hacen estáticamente (esto es, en tiempo de compilación) o dinámicamente (durante la ejecución), a estas estrategias frecuentemente se las denomina **ejecución múltiple con planificación estática (*static multiple issue*)** y **ejecución múltiple con planificación dinámica (*dynamic multiple issue*)**. Veremos que ambas estrategias tienen otros nombres más conocidos, pero que pueden ser menos precisos o más restrictivos.

Un pipeline de ejecución múltiple debe responsabilizarse de dos tareas principales muy distintas:

1. Empaquetar instrucciones en **ranuras de ejecución (*issue slots*)**: ¿Cómo determina el procesador cuántas y qué instrucciones pueden ser lanzadas a ejecutar en un determinado ciclo de reloj? En muchos procesadores que planifican el lanzamiento de instrucciones de forma estática, este proceso es gestionado, al menos parcialmente, por el compilador; en procesadores que planifican el lanzamiento de instrucciones de forma dinámica, esta tarea se realiza en tiempo de ejecución por parte del procesador, aunque es frecuente que el compilador haya tratado previamente de colocar las instrucciones en un orden beneficioso, que ayude a mejorar la velocidad de ejecución.
2. Gestionar los riesgos de datos y de control: En procesadores con ejecución de instrucciones planificado estáticamente algunas o todas las consecuencias de los riesgos de datos y de control son gestionadas por el compilador. Por el contrario, muchos procesadores con lanzamiento de instrucciones planificado dinámicamente intentan aliviar al menos algunas clases de riesgos usando técnicas hardware aplicadas en tiempo de ejecución.

Aunque se describan las dos estrategias como si fueran completamente distintas, en realidad algunas técnicas usadas por una de las estrategias son tomadas prestadas por la otra, y ninguna de las dos puede proclamarse como perfectamente pura.

El concepto de la especulación

Especulación: estrategia utilizada por el compilador o por el procesador para predecir el resultado de una instrucción, y de ese modo poder eliminar la dependencia que esa instrucción genera en la ejecución de otras instrucciones.

Uno de los métodos más importantes para encontrar y aprovechar más el ILP es la **especulación** que es una estrategia que permite al compilador o al procesador "adivinar" las propiedades de una instrucción, y de este modo habilitar la ejecución inmediata de otras instrucciones que puedan depender de la instrucción sobre la cual se ha especulado. Por ejemplo, es posible especular con el resultado de un salto, de forma que las instrucciones que siguen al salto puedan ser ejecutadas antes. O se puede especular que una instrucción de almacenamiento que precede a una instrucción de carga no hace referencia a la misma dirección, lo cual permitiría que la carga se ejecute antes que el almacenamiento. La dificultad de la especulación es que puede ser errónea. Así, cualquier mecanismo de especulación debe incluir tanto un método para verificar si la predicción fue correcta, como un método para deshacer los efectos de las instrucciones que fueron ejecutadas especulativamente. La implementación de esta capacidad de vuelta atrás añade complejidad a los procesadores que soportan la especulación.

La especulación la puede realizar tanto el compilador como el hardware. Por ejemplo, el compilador puede usar la especulación para reordenar las instrucciones, moviendo una instrucción antes o después de un salto, o una instrucción de carga antes o después de un almacenamiento. El procesador puede realizar la misma transformación por hardware y durante la ejecución usando técnicas que veremos un poco más adelante en esta sección.

Los mecanismos usados para recuperarse de una especulación incorrecta son bastante diferentes. En el caso de la especulación por software, el compilador generalmente inserta instrucciones adicionales para verificar la precisión de la especulación y proporciona rutinas que arreglan los desperfectos para que sean utilizadas en el caso de que la especulación sea incorrecta. En la especulación por hardware el procesador normalmente guarda los resultados especulativos hasta que se asegura de que ya no son especulativos. Si la especulación ha sido correcta, las instrucciones se completan permitiendo que el contenido de los búferes en los que se guardan temporalmente los resultados se escriban en los registros o en memoria. Si la especulación ha sido incorrecta, el hardware vacía el contenido de los búferes y se vuelve a ejecutar la secuencia de instrucciones correcta.

La especulación introduce otro posible problema: especular con ciertas instrucciones puede introducir excepciones que previamente no se producían. Por ejemplo, suponga que una instrucción de carga se mueve para ser ejecutada de forma especulativa, pero que la dirección que utiliza no es válida cuando la especulación es incorrecta. El resultado sería la generación de una excepción que nunca debería haber ocurrido. El problema se complica por el hecho de que si la instrucción de carga se está ejecutando de forma especulativa y la especulación es correcta, ¡entonces la excepción sí que debe ocurrir! En la especulación basada en el compilador estos problemas se evitan añadiendo un soporte especial a la especulación que permite que las excepciones sean ignoradas hasta que se pueda clarificar si éstas realmente deben ocurrir o no. En la especulación basada en hardware las excepciones son simplemente retenidas temporalmente en un búfer hasta que se verifica que la instrucción que causa la excepción ya no es especulativa y está lista para ser completada; llegados a este punto es cuando se permite que la excepción sea visible y se procede a su manejo siguiendo el mecanismo normal.

Puesto que la especulación puede mejorar las prestaciones cuando se hace de forma adecuada y puede reducirlas si se hace sin cuidado, se debe dedicar un esfuerzo importante a decidir cuándo es apropiado especular y cuándo no. Más adelante, en esta sección, veremos técnicas especulativas tanto estáticas como dinámicas.

Ejecución múltiple con planificación estática

Todos los procesadores con ejecución múltiple y planificación estática utilizan el compilador para que les asista en la tarea de empaquetar las instrucciones y en la gestión de los riesgos. En estos procesadores se puede interpretar que el conjunto de instrucciones que comienzan su ejecución en un determinado ciclo de reloj, lo que se denomina **paquete de ejecución (*issue packet*)**, es una instrucción larga con múltiples operaciones. Esta visión es mucho más que una analogía. Como este tipo de procesadores generalmente restringen las posibles combinaciones de instrucciones que pueden ser iniciadas en un determinado ciclo, es muy conveniente pensar en el paquete de ejecución como en una instrucción única que contiene ciertos campos predefinidos que le permiten especificar varias operaciones. Esta visión fue la que dio pie al nombre original de esta estrategia: **Very Long Instruction Word (VLIW, literalmente: palabra de instrucción de tamaño muy grande)**.

Paquete de ejecución: conjunto de instrucciones que se lanzan a ejecutar juntas durante un mismo ciclo de reloj; el paquete de instrucciones puede ser determinado de forma estática por parte del compilador o puede ser determinado de forma dinámica por parte del procesador.

Very Long Instruction Word, VLIW (palabra de instrucción de tamaño muy grande): estilo de arquitectura de repertorio de instrucciones que junta muchas operaciones independientes en una única instrucción más ancha, típicamente con muchos campos de código de operación separados.

La mayoría de los procesadores de planificación estática de la ejecución también confían al compilador cierta responsabilidad en la gestión de los riesgos de datos y de control. Entre estas responsabilidades se suele incluir la predicción estática de saltos y la reordenación de instrucciones para reducir o incluso evitar todos los riesgos de datos. Antes de describir el uso de las técnicas de planificación estática de ejecución con procesadores más agresivos, echaremos un vistazo a una versión simple del procesador MIPS.

Ejemplo: ejecución múltiple con planificación estática con MIPS

Para dar una idea de la planificación estática consideraremos un procesador MIPS capaz de ejecutar dos instrucciones por ciclo (dos vías, two issue), en el cual una de las instrucciones puede ser una operación entera en la ALU o un salto condicional, mientras que la otra puede ser una instrucción de memoria (carga o almacenamiento). Este tipo de diseño es similar al que se usa en algunos procesadores MIPS empotrados. Ejecutar dos instrucciones por ciclo requerirá la búsqueda y descodificación de un total de 64 bits de instrucciones. En muchos procesadores de planificación estática, y esencialmente en todos los procesadores VLIW, las posibilidades para seleccionar las instrucciones que se ejecutan simultáneamente están restringidas, para así simplificar la tarea de descodificar las instrucciones y de lanzarlas a ejecución. Así, suele ser necesario que la pareja de instrucciones esté alineada en una frontera de 64 bits, y que la operación ALU o de salto aparezca siempre en primer lugar. Si una de las instrucciones de la pareja no se puede utilizar, entonces será necesario reemplazarla por una no-operación (`nop`). Por tanto, las instrucciones se lanzarán a ejecutar siempre en parejas, aunque en ocasiones una de las ranuras de ejecución contenga un `nop`. La figura 4.68 muestra cómo avanzan las instrucciones a pares dentro del pipeline.

Los procesadores con planificación estática de la ejecución varían en la forma en que gestionan los riesgos potenciales de datos y de control. En algunos diseños es el compilador quien asume la total responsabilidad de eliminar *todos* los riesgos, planificando el código e insertando instrucciones `nop` para que se ejecute sin necesidad de la lógica de detección de riesgos y sin necesidad de que el hardware deba bloquear la ejecución. En otros diseños, el hardware detecta los riesgos de datos y genera bloqueos entre dos paquetes de ejecución consecutivos, aunque requiere que sea el compilador quien evite

Tipo de instrucción	Etapas del pipeline							
Instrucción ALU o salto condicional	IF	ID	EX	MEM	WB			
Instrucción carga o almacenamiento	IF	ID	EX	MEM	WB			
Instrucción ALU o salto condicional		IF	ID	EX	MEM	WB		
Instrucción carga o almacenamiento		IF	ID	EX	MEM	WB		
Instrucción ALU o salto condicional			IF	ID	EX	MEM	WB	
Instrucción carga o almacenamiento			IF	ID	EX	MEM	WB	
Instrucción ALU o salto condicional				IF	ID	EX	MEM	WB
Instrucción carga o almacenamiento				IF	ID	EX	MEM	WB

FIGURA 4.69 Funcionamiento de un pipeline de ejecución múltiple de dos instrucciones con planificación estática (*static two-issue*). Las instrucciones ALU y de transferencias de datos se lanzan al mismo tiempo. Se ha supuesto que se tiene la misma estructura de cinco etapas que la que se usa en un pipeline de ejecución simple (*single-issue*). Aunque esto no es estrictamente necesario, proporciona algunas ventajas. En particular, la gestión de las excepciones y el mantenimiento de un modelo de excepciones precisas, que se complica en los procesadores de ejecución múltiple, se ven simplificados por el hecho de mantener las escrituras en los registros en la etapa final del pipeline.

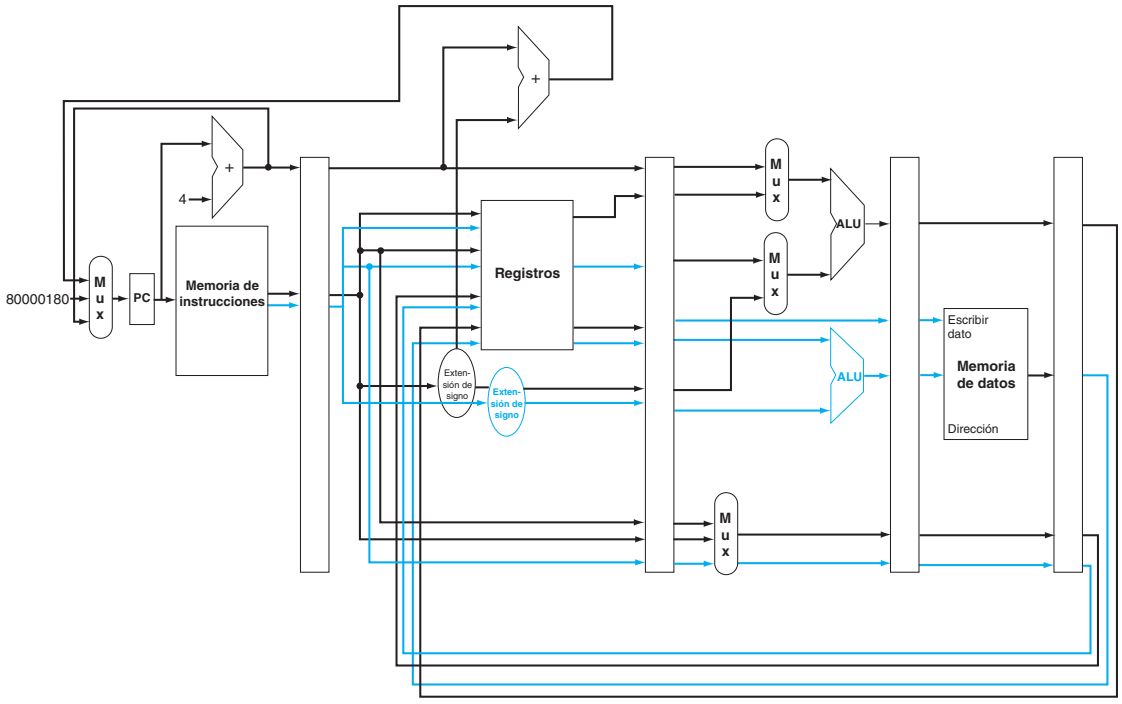

FIGURA 4.68 Camino de datos para la ejecución múltiple de dos instrucciones con planificación estática. Se resaltan los elementos adicionales para hacer que el camino de datos permita dos instrucciones simultáneas: 32 bits más desde la memoria de instrucciones, dos puertos de lectura más y uno más de escritura en el banco de registros, y otra ALU. Suponga que la ALU de la parte inferior calcula las direcciones de las transferencias de datos con memoria y que la ALU de la parte superior gestiona todas las demás operaciones enteras.

las dependencias entre las instrucciones internas de cada paquete de ejecución. Incluso así, un riesgo generalmente forzará a que se bloquee el paquete de ejecución entero que contiene la instrucción dependiente. Tanto si el software debe manejar todos los riesgos como si sólo debe tratar de reducir el porcentaje de riesgos entre diferentes paquetes de ejecución, se refuerza la apariencia de tener una única instrucción compuesta de múltiples operaciones. En el siguiente ejemplo se supondrá la segunda estrategia.

Para ejecutar en paralelo una operación de la ALU y una transferencia de datos a memoria, el primer hardware adicional que sería necesario, además de la habitual lógica de detección de riesgos, es disponer de más puertos de lectura y de escritura en el banco de registros (véase la figura 4.69). En un ciclo podríamos necesitar leer dos registros para la operación ALU y dos más para un almacenamiento, y también podría ser necesario un puerto de escritura para la operación ALU y uno más para una carga. Ya que la ALU está ocupada ejecutando su operación, también se necesitará un sumador adicional para calcular la dirección efectiva de las transferencias de datos a memoria. Sin estos recursos adicionales, el pipeline de dos instrucciones se vería limitado por riesgos estructurales.

Claramente, este procesador de ejecución doble puede llegar a mejorar las prestaciones en un factor 2. Pero para ello es necesario que puedan llegar a solapar su ejecución el doble de instrucciones, y este solapamiento adicional incrementa la pérdida relativa de rendimiento a causa de los riesgos de datos y de control. Por ejemplo, en nuestro pipeline simple de cinco etapas, las cargas tienen una **latencia de uso** de 1 ciclo de reloj, lo

Latencia de uso: número de ciclos de la señal de reloj entre una instrucción de carga y una instrucción que utiliza el resultado de la carga sin bloquear el pipeline.

cual impide que la instrucción siguiente pueda usar el dato inmediatamente y hace que deba bloquearse. En el pipeline de cinco etapas con doble capacidad de ejecución, el resultado de una instrucción de carga tampoco puede usarse en el siguiente ciclo de reloj, y esto significa ahora que las siguientes *dos* instrucciones no pueden usar el dato de la carga inmediatamente, y tendrían que bloquearse. Además, las instrucciones de la ALU que no tenían latencia de uso en el pipeline sencillo de cinco etapas tienen ahora una latencia de uso de 1 ciclo, ya que el resultado no puede utilizarse en la carga o el almacenamiento correpondiente. Para explotar el paralelismo disponible en un procesador de ejecución múltiple de forma eficiente, se necesitan técnicas más ambiciosas de planificación por parte del compilador o del hardware. En un esquema de planificación estática es el compilador el que debe asumir este rol.

Planificación simple de código para ejecución múltiple

EJEMPLO

¿Cómo se debería planificar este lazo de forma estática en un procesador MIPS con ejecución de dos instrucciones?

```
Loop: lw    $t0, 0($s1)     # $t0=elemento de un vector
      addu  $t0,$t0,$s2     # sumar valor escalar en $s2
      sw    $t0, 0($s1)     # almacenar resultado
      addi  $s1,$s1,-4      # decrementar puntero
      bne   $s1,$zero,Loop  # saltar si $s1!=0
```

Reordenar las instrucciones para evitar el máximo número de bloqueos que sea posible. Suponer que los saltos son predichos, de forma que los riesgos de control son gestionados por el hardware.

RESPUESTA

Las primeras tres instrucciones tienen dependencias de datos, así como las dos últimas. La figura 4.70 muestra la mejor planificación posible para estas instrucciones. Observe que sólo un par de instrucciones completan ambas ranuras del paquete de ejecución. Se tardan 4 ciclos por cada iteración del lazo; al ejecutar 5 instrucciones cada 4 ciclos se obtiene un decepcionante CPI de 0.8 frente un máximo valor alcanzable de 0.5, o un IPC de 1.25 frente a 2. Observe también que al calcular CPI o IPC los *nops* ejecutados no se cuentan como instrucciones útiles. Hacer eso podría mejorar el CPI, ¡pero no las prestaciones!

	Instrucción ALU o branch	Instrucción de transferencia de dato	Ciclo de reloj
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

FIGURA 4.70 El codigo planificado para ser ejecutado en un pipeline MIPS de dos instrucciones por ciclo. Los huecos vacíos representan nops.

Una técnica importante aplicada por el compilador para conseguir un mayor rendimiento en los lazos es el **desenrollado de lazos (*loop unrolling*)**, que consiste en realizar múltiples copias del cuerpo del lazo. Después del desenrollado, se dispone de más ILP para poder solapar la ejecución de instrucciones que pertenecen a diferentes iteraciones.

Desenrollado de lazos: técnica para obtener mayor rendimiento en los lazos de acceso a vectores, en la que se realizan múltiples copias del cuerpo del lazo para poder planificar conjuntamente la ejecución de instrucciones de diferentes iteraciones.

Desenrollado de lazos para pipelines con ejecución múltiple

EJEMPLO

Investigue cómo se mejora las prestaciones del ejemplo anterior desenrollando el lazo y planificando de nuevo la ejecución. Por simplicidad, suponga que el índice del lazo es múltiplo de cuatro.

RESPUESTA

Se necesita hacer cuatro copias del cuerpo del lazo para poder planificar las instrucciones sin que haya retardos. Después de desenrollar el lazo y de eliminar las instrucciones innecesarias, el lazo contendrá cuatro copias de cada instrucción `lw`, `add` y `sw`, además de una instrucción `addi` y una `bne`. La figura 4.71 muestra el código desenrollado y planificado.

Durante el proceso de desenrollado el compilador introduce el uso de registros adicionales (`$t1`, `$t2`, `$t3`). El objetivo de este proceso, llamado **renombrado de registros (*register renaming*)**, es eliminar aquellas dependencias que no son dependencias de datos verdaderas, pero que pueden conducir a riesgos potenciales o limitar la flexibilidad del compilador para planificar el código. Consideremos cuál habría sido el resultado de desenrollar el código si sólo se hubiera usado `$t0`. Habría copias repetidas en las instrucciones `lw $t0,0($s1)`, `addu $t0,$t0,$s2` seguidas por la instrucción de `sw $t0,4($s1)`. Aunque únicamente se use el registro `$t0`, estas secuencias de instrucciones siguen siendo en realidad completamente independientes (entre un par de instrucciones y el siguiente par de instrucciones no se produce ninguna transferencia de datos). El orden entre las instrucciones está forzado única y exclusivamente por la reutilización de un nombre (el registro `$t0`), en lugar de estar forzado por una dependencia de datos verdadera, y a esto se le conoce como **antidependencia (*antidependence*) o dependencia de nombre (*name dependence*)**.

Renombrado de registros: volver a nombrar registros, mediante el compilador o el hardware, para eliminar las antidependencias.

Renombrar los registros durante el proceso de desenrollado permite que posteriormente el compilador pueda mover estas instrucciones independientes y así planificar mejor el código. El proceso de renombrado elimina las dependencias de nombre, mientras que preserva las dependencias de datos reales.

Antidependencia (dependencia de nombre): ordenación forzada por la reutilización de un nombre, generalmente un registro, en lugar de una dependencia de dato verdadera que acarrea un valor entre dos instrucciones.

Observe que ahora se logra que 12 de las 14 instrucciones del lazo se puedan ejecutar combinadas en una pareja. Cuatro iteraciones del lazo tardan 8 ciclos, que son 2 ciclos por iteración, lo cual supone un CPI de 8/14 = 0.57. El desenrollado de lazos y la subsiguiente planificación de instrucciones dan un factor de mejora de dos, en parte debido a la reducción en el número de instrucciones de control del lazo y en parte debido a la ejecución dual de instrucciones. El coste de esta mejora de las prestaciones es el uso de cuatro registros temporales en vez de uno, además de un incremento significativo del tamaño del código.

Procesadores con ejecución múltiple y planificación dinámica

Los procesadores con planificación dinámica de la ejecución múltiple de instrucciones se conocen también como procesadores **superescalares (*superscalar*)**, o simplemente superescalares. En los procesadores superescalares más simples, las instrucciones se

Superescalar: técnica avanzada de segmentación que permite al procesador ejecutar más de una instrucción por ciclo de reloj.

	Instrucción ALU o branch	Instrucción de transferencia de dato	Ciclo de reloj
Loop:	addi $s1,$s1,-16	lw $t0, 0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2, 8($s1)	3
	addu $t1,$t1,$s2	lw $t3, 4($s1)	4
	addu $t2,$t2,$s2	sw $t0, 16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1,$zero,Loop	sw $t3, 4($s1)	8

FIGURA 4.71 Código de la figura 4.70 desenrollado y planificado para un procesador MIPS de ejecución dual. Los huecos vacíos representan *nops*. Como la primera instrucción de lazo decrementa $s1 en 16 unidades, las direcciones de donde se cargan los datos son el valor original de $sl, y luego esta dirección menos 4, menos 8 y menos 12.

lanzan a ejecutar en orden, y en un determinado ciclo de reloj el procesador decide si se pueden ejecutar una o más instrucciones nuevas o ninguna. Obviamente, para alcanzar unas buenas prestaciones en esto procesadores, se requiere que el compilador planifique las instrucciones de forma que se eliminen dependencias y se mejore la frecuencia de ejecución. Incluso con esa planificación realizada por el compilador, existe una diferencia importante entre este superescalar sencillo y un procesador VLIW: el hardware garantiza que el código, tanto si está planificado como si no lo está, se ejecutará correctamente. Más aún, independientemente de la estructura del pipeline o de la capacidad del procesador para ejecutar instrucciones simultáneamente, el código compilado siempre funcionará de forma correcta. Esto no es así en algunos diseños VLIW, que requieren que se recompile el código cuando éste se quiere ejecutar en diferentes modelos del procesador. En otros procesadores de planificación estática el código sí que se ejecuta correctamente en las diferentes implementaciones de la arquitectura, pero frecuentemente el rendimiento acaba siendo tan pobre que la recompilación acaba siendo necesaria.

Planificación dinámica del pipeline: soporte hardware para reordenar la ejecución de las instrucciones de modo que se eviten bloqueos.

Muchos superescalares extienden el ámbito de las decisiones dinámicas para incluir la **planificación dinámica del pipeline (*dynamic pipeline scheduling*)**. La planificación dinámica del pipeline escoge las instrucciones a ejecutar en cada ciclo de reloj intentando en lo posible evitar los riesgos y los bloqueos. Comenzaremos con un ejemplo simple que evita un riesgo de datos. Considere la siguiente secuencia de código

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

Aunque la instrucción `sub` está preparada para ser ejecutada, debe esperar a que primero se completen las instrucciones `lw` y `addu`, lo cual puede suponer muchos ciclos si los accesos a memoria son lentos. (El capítulo 5 explica el funcionamiento de las cachés, la razón de que los accesos a memoria sean en ocasiones muy lentos). La planificación dinámica permite evitar estos riesgos parcial o completamente.

Planificación dinámica del pipeline

La planificación dinámica del pipeline escoge las siguientes instrucciones a ejecutar, posiblemente reordenándolas para así evitar bloqueos. En estos procesadores, el pipeline se divide en tres unidades fundamentales: una unidad de búsqueda instrucciones y de decisión de qué instrucciones ejecutar (*issue unit*), múltiples unidades funcionales (12 o incluso más en los diseños de gama alta en el 2008), y una **unidad de confirmación (*commit unit*).** La figura 4.72 muestra el modelo. La primera unidad busca instrucciones, las descodifica y envía cada una de ellas a la unidad funcional que corresponda para ser ejecutada. Cada unidad funcional dispone de búferes, llamados **estaciones de reserva (*reservation stations*)**, que almacenan la operación y los operandos. (En la siguiente sección veremos una alternativa a las estaciones de reserva que usan muchos procesadores recientes). Tan pronto como el búfer contiene todos los operandos de la operación y la unidad funcional está preparada, se ejecuta la instrucción y se calcula el resultado. Una vez obtenido el resultado, se envía a todas las estaciones de reserva que puedan estar esperando por él, además de enviarlo a la unidad de confirmación, que lo guardará temporalmente hasta que sea seguro escribirlo en el banco de registros o, si es un almacenamiento, escribirlo en memoria. Este búfer que se encuentra en la unidad de confirmación, llamado con frecuencia **búfer de reordenación (*reorder buffer*)**, se usa también para proporcionar operandos de una forma muy similar a como hace la lógica de anticipación de resultados en un procesador planificado de forma estática. Una vez que el resultado es confirmado en el banco de registros, puede ser accedido directamente desde allí, igual que se hace en un pipeline normal.

Unidad de confirmación: en un procesador con ejecución fuera de orden, es la unidad que decide cuando es seguro confirmar el resultado de una operación y hacerlo visible, bien a los registros, bien a la memoria.

Estación de reserva: búfer acoplado a las unidades funcionales que almacena temporalmente los operandos y las operaciones pendientes de ser ejecutadas.

Búfer de reordenación: búfer que almacena temporalmente los resultados producidos en un procesador planificado de forma dinámica hasta que es seguro almacenarlos en memoria o en un registro y hacerlos así visibles.

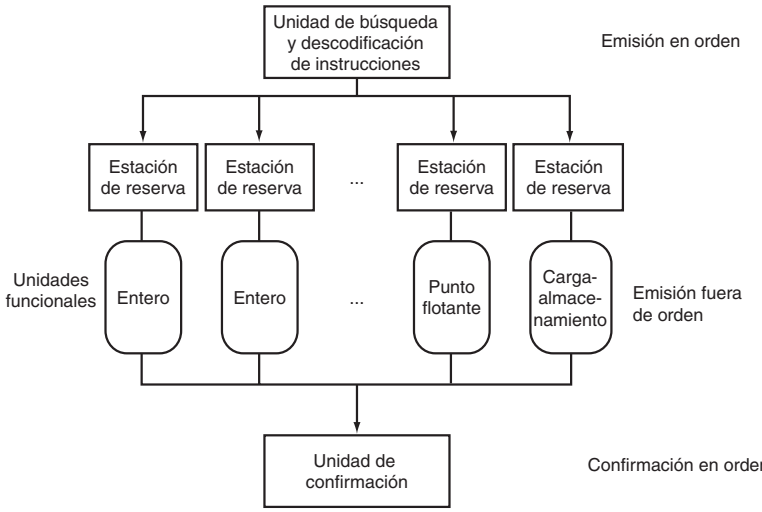

FIGURA 4.72 Las tres unidades principales de un pipeline planificado dinámicamente. El paso final de actualización del estado también se llama jubilación o graduación.

La combinación de almacenar los operandos temporalmente en las estaciones de reserva y de almacenar los resultados temporalmente en el búfer de reordenación supone una forma de renombrado, justamente como la usada por el compilador en el ejemplo anterior del desenrollado de lazos mostrado en la página 397. Para ver cómo funciona todo esto de forma conceptual es necesario considerar los siguientes pasos:

1. Cuando se lanza una instrucción a la unidad de ejecución, si alguno de sus operandos se encuentra en el banco de registros o en el búfer de reordenación, entonces se copia inmediatamente a la estación de reserva correspondiente, donde se mantiene temporalmente hasta que todos los operandos y la unidad funcional están disponibles. Para la instrucción que se ha lanzado, la copia del operando que hay en el registro ya no es necesaria, y se puede permitir realizar una escritura sobre ese registro y sobrescribir su valor actual.
2. Si un operando no está en el banco de registros ni en el búfer de reordenación, entonces debe esperarse a que sea producido por una unidad funcional. Se toma nota de la unidad funcional que producirá el resultado y, cuando finalmente esto ocurra, se copiará directamente desde la unidad funcional, mediante la red de anticipación, a la estación de reserva que está esperando.

Estos pasos usan de forma efectiva el búfer de reordenación y las estaciones de reserva para implementar el renombrado de registros.

Conceptualmente se puede pensar que un pipeline con planificación dinámica analiza la estructura del flujo de datos del programa. El procesador entonces ejecuta las instrucciones en un cierto orden que preserva el orden del flujo de datos del programa. Este estilo de ejecución se llama **ejecución fuera de orden**, ya que las instrucciones pueden ser ejecutadas en un orden diferente del orden en que fueron capturadas.

Ejecución fuera de orden: propiedad de la ejecución segmentada que permite que el bloqueo de la ejecución de una instrucción no provoque que las instrucciones que le sigan tengan que esperarse.

Para que los programas se comporten como si fueran ejecutados en un pipeline simple con ejecución ordenada, las etapas de búsqueda de instrucciones y de descodificación se deben realizar también en orden, lo cual permite tomar nota de las dependencias, y la unidad de confirmación de la ejecución también debe escribir los resultados en registros y en memoria en el orden original del programa. Este modo conservador de operación se denomina **confirmación en orden** (*in-order commit*). De este modo, si ocurre una excepción, el computador podrá apuntar a la última instrucción ejecutada, y los únicos registros que habrán sido actualizados serán aquellos escritos por las instrucciones anteriores a la que ha causado la excepción. Aunque las etapas denominadas *front-end* (búsqueda y lanzamiento de instrucciones) y la etapa denominada *back-end* (confirmación) del pipeline funcionen en orden, las unidades funcionales son libres de iniciar la ejecución de las instrucciones cuando los datos que se necesitan estén disponibles. Hoy en día, todos los pipelines planificados dinámicamente usan finalización en orden.

Confirmación en orden: confirmación de los resultados de la ejecución segmentada que se escriben en el estado visible para el programador en el mismo orden en que las instrucciones se traen de memoria.

La planificación dinámica frecuentemente se extiende para incluir especulación basada en hardware, especialmente en lo que se refiere a los resultados de los saltos. Prediciendo el sentido y la dirección destino de un salto condicional, un procesador con planificación dinámica puede seguir buscando y ejecutando instrucciones a lo largo del camino del flujo de control predicho. Puesto que las instrucciones se confirman en orden, antes de que alguna instrucción en el camino predicho sea confirmada se sabrá si el salto ha sido correctamente predicho o no. Es también fácil para un procesador segmentado con ejecución especulativa y planificación dinámica que dé soporte a la especulación de direcciones de las instrucciones de carga, permitiendo la reordenación entre cargas y almacenamientos, usando la unidad de confirmación para evitar los problemas de una especulación incorrecta. En la siguiente sección revisaremos el uso de la ejecución con planificación dinámica y especulativa en el diseño del AMD Opteron X4 (Barcelona).

Comprender las prestaciones de los programas

Puesto que los compiladores son capaces de planificar el código para sortear las dependencias de datos, uno puede preguntarse para qué los procesadores superescalares usan la planificación dinámica. Existen tres razones principales. En primer lugar, no todos los bloqueos en el pipeline son predecibles. En particular, los fallos de caché (véase capítulo 5) son la causa de muchos bloqueos impredecibles. La planificación dinámica permite que el procesador oculte el efecto de algunos de estos bloqueos al seguir ejecutando instrucciones mientras el bloqueo no finaliza.

En segundo lugar, si un procesador utiliza la predicción dinámica de saltos para especular con los resultados de los saltos, entonces no será posible conocer el orden exacto de las instrucciones en tiempo de compilación, puesto que el orden dependerá del propio comportamiento de los saltos. La incorporación de la especulación dinámica para aprovechar mejor el paralelismo de instrucciones ILP sin disponer a su vez de una planificación dinámica de la ejecución, restringiría de forma significativa el beneficio de la especulación.

En tercer lugar, como la latencia en el pipeline y la anchura de ejecución varían de una implementación a otra, la mejor forma de compilar una secuencia de código también varía. Por ejemplo, la mejor manera de planificar la ejecución de una secuencia de instrucciones dependientes se ve afectada tanto por la anchura de ejecución como por la latencia de las operaciones. La estructura del pipeline afecta al número de veces que un lazo debe ser desenrollado para evitar los bloqueos, así como al proceso de renombrado de registros realizado por el compilador. La planificación dinámica permite al hardware ocultar muchos de estos detalles. De este modo, los usuarios y los distribuidores de software no necesitan preocuparse de tener diferentes versiones de un programa para diferentes implementaciones del mismo repertorio de instrucciones. De forma similar, el código generado para versiones previas de un procesador podrá aprovechar mucha parte de las ventajas de una nueva implementación del procesador sin necesidad de tener que recompilarlo.

IDEA clave

Tanto la segmentación como la ejecución múltiple de instrucciones intentan aprovechar el paralelismo de instrucciones ILP para aumentar el ritmo máximo de ejecución de instrucciones. Pero las dependencias de datos y de control determinan un límite superior en las prestaciones que se puede conseguir de forma sostenida, ya que a veces el procesador debe esperar a que una dependencia se resuelva. Las estrategias para aprovechar el paralelismo de instrucciones ILP que están centradas en el software dependen de la habilidad del compilador para encontrar estas dependencias y reducir sus efectos, mientras que las estrategias centradas en el Hardware dependen de extensiones a los mecanismos de segmentación y de lanzamiento múltiple de instrucciones. La especulación, tanto si es realizada por el compilador como por el harware, incrementa la cantidad de paralelismo de instrucciones ILP que puede llegar a ser aprovechado, aunque se debe tener cuidado, ya que la especulación incorrecta puede llegar a reducir las prestaciones.

Interfaz hardware software

Los modernos microprocesadores de altas prestaciones son capaces de lanzar a ejecutar bastantes instrucciones por ciclo, pero desafortunadamente es muy difícil alcanzar esa capacidad de forma sostenida. Por ejemplo, a pesar de que existen procesadores que ejecutan 4 y 6 instrucciones por ciclo, muy pocas aplicaciones son capaces de ejecutar de forma sostenida más de dos instrucciones por ciclo. Esto se debe básicamente a dos razones.

Primero, dentro del pipeline, los cuellos de botella más importantes se deben a dependencias que no pueden evitarse o reducirse, y de este modo el paralelismo entre instrucciones y la velocidad sostenida de ejecución se reducen. Aunque poca cosa se puede hacer con las dependencias reales, frecuentemente ni el compilador ni el hardware son capaces de asegurar si existe o no una dependencia y, de una forma conservadora, deben suponer que la dependencia existe. Por ejemplo, un código que haga uso de punteros, particularmente si lo hace de forma tal que crea muchas formas de referenciar la misma variable (*aliasing*), dará lugar a muchas dependencias potenciales implícitas. En cambio, la mayor regularidad de los accesos a un vector frecuentemente permite que el compilador deduzca que no existen dependencias. De forma similar, las instrucciones de salto que no pueden ser predichas de forma precisa tanto en tiempo de ejecución como en tiempo de compilación limitarán las posibilidades de aprovechar el paralelismo de instrucciones ILP. En muchas ocasiones existe paralelismo ILP adicional disponible, pero al encontrarse muy alejado (a veces a una distancia de miles de instrucciones) la habilidad del compilador o del hardware para encontrar el paralelismo ILP es muy limitada.

En segundo lugar, las deficiencias del sistema de memoria (el tema del capítulo 7) también limitan la habilidad de mantener el pipeline lleno. Algunos bloqueos producidos por el sistema de memoria pueden ser ocultados, pero si se dispone de una cantidad limitada de paralelismo ILP, esto también limita la cantidad de estos bloqueos que se puede ocultar.

Eficiencia energética y segmentación avanzada

El inconveniente de aumentar la explotación del paralelismo a nivel de instrucciones vía la planificación dinámica con búsqueda de múltiples instrucciones y la especulación es la eficiencias energética. Cada innovación fue capaz de transformar los transitores adicionales en más prestaciones, pero a menudo se hizo de forma muy ineficiente. Ahora que tenemos que luchar contra el muro de la potencia, estamos buscando diseños con varios procesadores por chip, donde los procesadores no tienen pipelines tan profundos ni técnicas de especulación tan agresivas como sus predecesores.

La opinión general es que aunque los procesadores más sencillos no son tan rápidos como sus hermanos más sofisticados, tienen mejores prestaciones por vatio, de modo que pueden proporcionar mejores prestaciones por chip cuando los diseños están limitados más por la potencia disipada que por el número de transistores.

La figura 4.73 muesra el número de etapas, el ancho de emisión, el nivel de especulación, la frecuencia del reloj, el número de núcleos por chip y la potencia de varios microprocesadores pasados y recientes. Obsérvese el descenso del número de etapas y la potencia a medida que los fabricantes introducen diseños multinúcleo.

Microprocesador	Año	Frecuencia de reloj	Etapas	Ancho de ejecución	Fuera de orden/ especulación	Núcleos/ chip	Potencia
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Si	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Si	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Si	1	103 W
Intel Core	2006	2930 MHz	14	4	Si	2	75 W
Sun UltraSPARC III	2003	1950 MHz	14	4	No	1	90 W
Sun UltraSPARC T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W

FIGURA 4.73 Relación de microprocesadores Intel y Sun en términos de complejidad del pipeline, número de núcleos y potencia. Entre las etapas del pipeline del Pentium4 no se incluyen las etapas de confirmación (commit). Si estas etapas se incluyen, los pipelines del Pentium 4 serían todavía más profundos.

Extensión: Los controles de una unidad de confirmación actualizan el banco de registro y la memoria. Algunos procesadores con planificación dinámica actualizan el banco de registros de forma inmediata durante la ejecución, utilizando registros adicionales para implementar el renombrado y manteniendo una copia de los contenidos anteriores hasta que la instrucción que actualiza los registros ya no es especulativa. Otros procesadores guardan los resultados, típicamente en una estructura llamada búfer de reordenación, y la actualización real de los registros se realiza más tarde como parte de la finalización. Los almacenamientos en memoria deben mantenerse en un búfer hasta la finalización, bien en un búfer de almacenamientos (véase el capítulo 5) o bien en un búfer de reordenación. La unidad confirmación permite que la instrucción de almacenamiento escriba el contenido del búfer en memoria cuando el búfer tiene una dirección y un dato válidos y el almacenamiento ya no depende de saltos condicionales con predicción.

Extensión: Los accesos a memoria se benefician de las caches no bloqueantes (*nonblocking caches*), que continúan procesando accesos a la cache durante un fallo de cache (véase el capítulo 5). Los procesadores con ejecución fuera-de-orden necesitan que la cache esté diseñada de forma que permita la ejecución de instrucciones durante un fallo de cache.

Autoevaluación

Diga si las siguientes técnicas o componentes se asocian fundamentalmente con estrategias bien software o hardware de aprovechamiento del paralelismo de instrucciones ILP. En algunos casos, la respuesta puede ser ambas estrategias.

1. Predicción de saltos
2. Ejecución múltiple
3. VLIW
4. Superescalar
5. Planificación dinámica
6. Ejecución fuera-de-orden
7. Especulación
8. Búfer de reordenación
9. Renombrado de registros

4.11 Casos reales: El pipeline del AMD Opteron X4 (Barcelona)

Como la mayoría de los computadores modernos, los microprocesadores x86 incorporan técnicas de segmentación sofisticadas. Sin embargo, estos procesadores están afrontando todavía el reto de implementar el complejo repertorio de instrucciones x86, descrito en el capítulo 2. Tanto AMD como Intel capturan instrucciones x86 y las traducen internamente a instrucciones tipo MIPS, que AMD llama operaciones RICS (Rops) e Intel llama microoperaciones. Las operaciones RISC se ejecutan en un pipeline sofisticado con planificación dinámica y especulativo, capaz de mantener una velocidad de tres operaciones RISC por ciclo de reloj en el AMD Opteron X4 (Barcelona). En esta sección nos centramos en el pipeline de las operaciones RISC.

Cuando se considera el diseño de sofisticados procesadores con planificación dinámica, el diseño de las unidades funcionales, de la caché, del banco de registros, de la lógica para lanzar a ejecutar las instrucciones, y el control total del pipeline está muy interrelacionado, haciendo muy difícil separar lo que es sólo el camino de datos de la totalidad del pipeline. Por esta razón, muchos ingenieros e investigadores han adoptado el término **microarquitectura** para referirse a la arquitectura interna detallada de un procesador. La figura 4.74 muestra la microarquitectura del X4, centrándose en las estructuras que se utilizan para ejecutar las operaciones RISC.

Microarquitectura: organización del procesador, que incluye sus principales unidades funcionales, sus elementos de interconexión y el control.

Otra forma de considerar el X4 es mirando las etapas de segmentación que atraviesa una instrucción típica. La figura 4.75 muestra la estructura del pipeline y el número típico de ciclos de reloj que se gastan en él. Por supuesto, el número de ciclos variará debido a la naturaleza de la planificación dinámica, así como a los requerimientos individuales propios de cada operación RISC.

Extensión: El Pentium 4 usa un esquema para resolver las antidependencias y para arreglar las especulaciones incorrectas. Este esquema utiliza un búfer de reordenación junto al renombrado de registros. El renombrado de registros renombra de forma explícita los **registros de la arquitectura (*architectural registers*)** del procesador (16 en el caso de la versión de 64 bits de la arquitectura X86) a un conjunto mayor de registros físicos (72 en el X4). El X4 usa el renombrado de registros para eliminar las antidependencias. El renombrado de registros necesita que el procesador mantenga un asignación entre los registros de la arquitectura y los registros físicos, indicando qué registro físico es la copia más actual de cada registro de la arquitectura. Si se mantiene la información de los renombrados que se han producido, el renombrado de registros ofrece una estrategia alternativa para recuperarse cuando se produce una especulación incorrecta: simplemente se deshacen los mapeos que han ocurrido desde la primera instrucción producto de la especulación incorrecta. Esto provocará que se retorne al estado del procesador que se tenía después de la última instrucción correctamente ejecutada, logrando recuperar la asignación correcto entre registros de la arquitectura y físicos.

Registros de la arquitectura: registros visibles en el repertorio de instrucciones del procesador; por ejemplo, en MIPS, estos son 32 registros enteros y 16 registros de punto flotante.

Autoevaluación

Las siguientes afirmaciones, ¿son ciertas o falsas?

1. El pipeline de ejecución múltiple del Opteron X4 ejecuta directamente instrucciones x86.
2. El X4 usa planificación dinámica pero sin especulación.

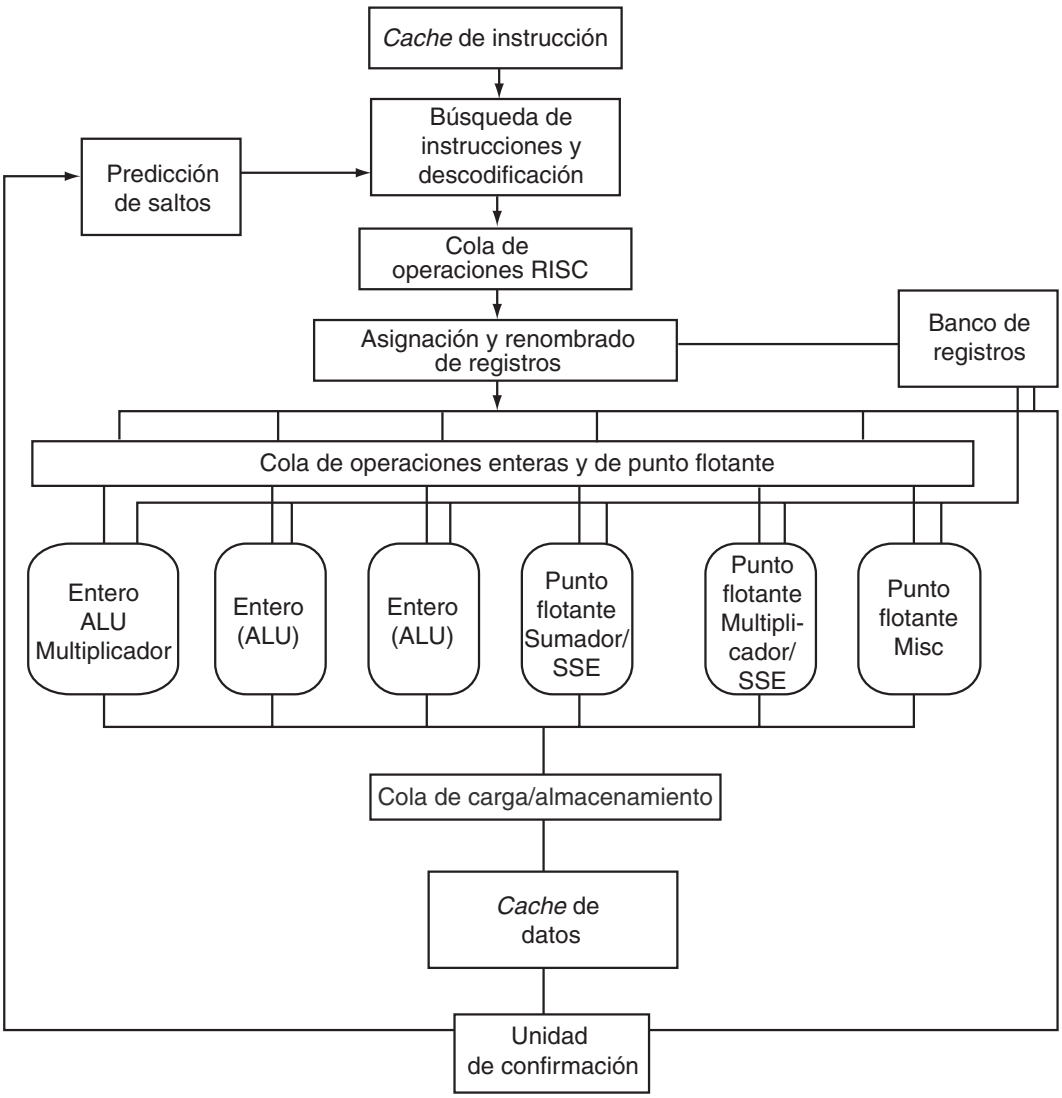

FIGURA 4.74 La microarquitectura del AMD Opteron X4. Las amplias colas disponibles permiten que hasta 106 operaciones RISC estén pendientes de entrar en ejecución, incluyendo 24 operaciones de enteros, 36 operaciones de punto flotante/SSE y 44 cargas y almacenamientos. Las unidades de carga y almacenamiento están realmente separadas en dos partes, la primera parte se encarga del cálculo de la dirección en unidad ALU de enteros y la segunda parte es responsable de la referencia a memoria. Hay un amplio circuito de anticipación entre unidades funcionales; puesto que el pipeline es dinámico en lugar de estático, la anticipación se hace etiquetando los resultados y rastreando los operandos fuente, para detectar cuando un resultado ha sido obtenido por una instrucción en una de las colas que están esperando por este resultado.

3. La microarquitectura del X4 tiene muchos más registros de los que requiere la arquitectura x86.
4. El pipeline del X4 tiene menos de la mitad de etapas que el Pentium 4 Prescott (véase la figura 4.73).

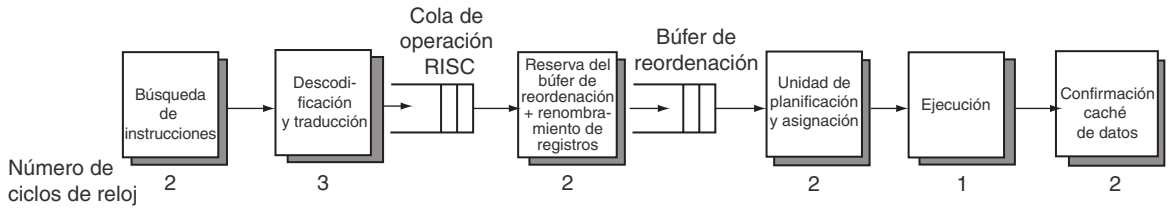

FIGURA 4.75 El pipeline del Opteron X4 mostrando el flujo de una instrucción típica y el número de ciclos de reloj de los principales pasos del pipeline de 12 etapas de las operaciones RISC de enteros. La cola de ejecución de punto flotante tiene una longitud de 17 etapas. Se muestran también los principales búferes donde esperan las operaciones RISC.

Comprender las prestaciones de los programas

El Opteron X4 combina un pipeline de 12 etapas con una ejecución múltiple agresiva para poder alcanzar altas prestaciones. El impacto de las dependencias de datos se reduce haciendo que las latencias entre operaciones consecutivas (*back-to-back*) sean reducidas. Para los programas que se ejecutan en este procesador, ¿cuáles son los cuellos de botella potenciales más serios para las prestaciones? La siguiente lista incluye algunos de estos problemas potenciales para las prestaciones; los tres últimos también se pueden aplicar de alguna manera a cualquier procesador segmentado de altas prestaciones.

- El uso de instrucciones x86 que no se traducen en pocas operaciones RISC simples.
- Las instrucciones de salto difíciles de predecir, que causan bloqueos debido a fallos de predicción y hacen que se deba reiniciar la ejecución por especulación errónea.
- Largas dependencias, generalmente causadas por instrucciones con latencias elevadas o por fallos en la caché de datos, que dan lugar a bloqueos durante la ejecución.
- Retrasos en los accesos a la memoria que provocan que el procesador se bloquee (véase capítulo 5).

Tema avanzado: una introducción al diseño digital utilizando un lenguaje de descripción hardware para describir y modelar un pipeline y más figuras sobre segmentación

El diseño digital moderno se hace utilizando lenguajes de descripción hardware y modernas herramientas de síntesis asistida por computador que son capaces de crear diseños hardware detallados a partir de una descripción con bibliotecas y síntesis lógica. Se han escrito libros enteros sobre estos lenguajes y su utilización en diseño digital. Esta sección, incluida en el CD, da una breve introducción y muestra como un lenguaje de descripción hardware, Verilog en este caso, puede utilizarse para des-