EXCEPCIONES Y PARALELISMO A NIVEL DE INSTRUCCIONES

ARQUITECTURA DEL PROCESADOR II

1. Introducción a las excepciones

Los branches y los jumps proveen formas de cambiar el flujo de control en un programa[1]. Las excepciones también pueden cambiar el flujo de control en un programa. Por convención en MIPS se denomina excepción a cualquier cambio no esperado en el flujo de control, independientemente de su origen (es decir, sin distinguir entre una fuente dentro del procesador y una fuente externa). Una excepción se dice que es sincrónica si ocurre en el mismo lugar cada vez que el programa es ejecutado con los mismos datos y la misma ubicación en la memoria. Los overflows aritméticos, instrucciones indefinidas, falta de página son algunos ejemplos de excepciones sincrónicas. Las excepciones asincrónicas, por otra parte, suceden sin relación temporal con el programa ejecutado. Los requerimientos de E/S, errores de memoria, falta de energía son ejemplos de eventos asincrónicos. Las excepciones síncronas, que resultan directamente desde la ejecución del programa, son llamadas traps. Cuando ocurre una excepción, el control es transferido a un programa distinto llamado *manejador de excepciones*, que se escribe especialmente con el propósito de tratar con las excepciones. Después de atender la excepción, el control es devuelto al programa que estaba ejecutando cuando ocurrió la excepción: este programa continúa como si nada hubiese ocurrido. Una excepción ocurre como si un procedimiento (sin parámetros y que no devuelve valores) hubiese sido insertado en el programa. Debido a que el código del manejador de excepción puede ser ejecutado en cualquier momento, no se le pueden pasar parámetros: pasarle parámetros requiere una preparación previa. Por la misma razón no puede haber un valor devuelto. Es importante tener en mente que los manejadores de las excepciones deben mantener el estado del programa que fue interrumpido de manera que su ejecución pueda luego continuar. Como todo procedimiento, el manejador de excepción, debe resguardar cualquier registro que modifique y luego restablecerlo antes de retornar el control al programa interrumpido.

En MIPS guardar los registros en memoria tiene un problema: direccionar la memoria requiere de un registro (el registro base) con el cual se forma la dirección. Esto significa que un registro debe ser modificado antes que cualquier registro pueda ser salvado!.

La convención de uso de los registros en MIPS, reserva los registros $26 y $27 ($k0 y $k1) para ser usadas por el manejador de interrupciones. Esto significa que el manejador de interrupciones puede usar estos registros sin antes resguardarlos. Un programa de usuario que usa estos registros puede inesperadamente encontrarlos cambiados.

[1]Esta sección es la traducción de parte de la página [2]

2. Consideraciones generales sobre las excepciones

El mayor desafío en el diseño de un procesador es el control: es la parte más difícil de hacer funcionar de manera correcta y veloz[2]. Una de las cosas más complicadas del control es la implementación de las excepciones e interrupciones – aquellos eventos, que además de los saltos y los jumps, cambian la secuencia de ejecución. Inicialmente fueron creadas para manejar eventos no esperados desde adentro del procesador, como el overflow aritmético. El mismo mecanismo básico fue extendido para que los dispositivos de E/S se comuniquen con el procesador.

Muchas arquitecturas y autores no distinguen entre *interrupciones* y *excepciones*, frecuentemente el nombre clásico de interrupciones se refiere a ambos tipos de eventos. Por ejemplo, en el ámbito del Intel X86 sólo se usa el concepto de interrupción. Nosotros seguimos la convención de MIPS, usando el término excepciones para hacer referencia a cualquier cambio inesperado en el flujo de control, sin distinguir si la causa es interna o externa; usamos el término interrupciones únicamente cuando el evento ha sido generado externamente. Los siguientes cinco ejemplos muestran distintas situaciones generadas interna o externamente.

Tipo de evento	Origen	Nombre en MIPS
Dispositivo de E/S	Externo	Interrupción
Invocación al S.O.	Interno	Excepción
Overflow	Interno	Excepción
Cod Op desconocido	Interno	Excepción
Mal funcionamiento del hard	Externo	Excepción o Interrupción

Definición 1. *Una excepción, también llamada interrupción, es un evento no planificado que altera la ejecución de un programa, por ejemplo se usa cuando hay overflow.*

Definición 2. *Una interrupción es una excepción que proviene desde afuera del procesador.*

Muchos de los requerimientos para soportar excepciones provienen de una situación específica; esta situación es la que causa que una excepción ocurra. En este documento, nos enfocamos en la implementación del control requerido para detectar dos tipos de excepciones, las cuales provienen del conjunto de instrucciones reducido y la implementación que hemos asumido.

Tanto la detección de condiciones excepcionales, como la toma de las acciones apropiadas se hacen sobre el *camino de tiempo crítico* del procesador; ésto determina el tiempo de ciclo de reloj y en consecuencia el desempeño del procesador. Si no se otorga la atención necesaria a las excepciones durante el diseño de la unidad de control, el intento de agregar excepciones en un diseño complicado puede reducir el desempeño de manera significativa, como así también complicar la tarea de conseguir el diseño correcto.

Como se manejan las excepciones en la arquitectura MIPS. Los dos tipos de excepciones que pueden ser generadas con la versión del MIPS hasta aquí considerada son la ejecución de una instrucción indefinida y el overflow aritmético. Usaremos el overflow

[2]Esta sección es la traducción de la sección 4.9 del libro de Patterson y Hennessy [1]

aritmético en la operación `add $1,$2,$1` como un ejemplo de excepción en esta parte del documento. La acción básica que el procesador debe realizar cuando ocurre una excepción es salvar la dirección de la instrucción con problemas en el *exception program counter* (EPC) y luego transferir el control a una dirección específica del Sistema Operativo.

El Sistema Operativo puede entonces tomar la acción adecuada, la cual puede consistir en brindar algún servicio al programa de usuario, tomar alguna acción predefinida en respuesta a un overflow, o detener la ejecución del programa y reportar un error. Después de realizar cualquier acción que es requerida debido a la excepción, el Sistema Operativo puede terminar el programa o continuar la ejecución usando el EPC para saber donde recomenzar la ejecución del programa.

Para que el Sistema Operativo maneje la excepción debe conocer las razones de la misma, también debe conocer la instrucción que la causó. Hay dos métodos principales usados para comunicar las razones de una excepción. El método usado en la arquitectura MIPS es incluir un registro de estado (llamado *registro de causa*), el cual mantiene un campo que indica las razones para la excepción.

Un segundo método es conocido como *interrupciones vectorizadas*. En una interrupción vectorizada, la dirección a la cual el control se transfiere es determinada por la causa de la excepción. Por ejemplo para tratar con los dos tipos de excepciones mencionada más arriba, se podrían definir las siguientes direcciones de vectores de excepción:

Tipo de excepción	Dirección del vector de excepción
Instrucción indefinida	0X8000 0000
Overflow Aritmético	0X8000 0180

Definición 3. *Las interrupciones vectorizadas son aquellas en las que la dirección a la cual se transfiere el control es determinada por la causa de la excepción.*

El Sistema Operativo conoce las razones de la excepción por la dirección en la cual es inicializado. Las direcciones están separadas por 32 bytes u ocho instrucciones, y el Sistema Operativo debe detectar las razones de la excepción y realizar algún procesamiento limitado, usando solamente esas pocas instrucciones. Cuando la interrupción no es vectorizada, se usa un único punto de entrada para todas las excepciones y el Sistema Operativo decodifica el registro de estado para encontrar la causa de la excepción.

Para realizar el procesamiento requerido por las excepciones, sólo es necesario agregar unos pocos registros extras y algunas señales de control sobre la implementación básica. Además se debe extender algo el mecanismo de control. Aquí se asume que se está implementando el sistema de excepciones usado en la arquitectura MIPS, con un único punto de entrada en la dirección 0X8000 0180. (Implementar interrupciones vectorizadas no es más complicado).

Es necesario agregar dos registros adicionales a la actual implementación del MIPS:

- EPC, un registro de 32 bits usado para mantener la dirección de la instrucción con problemas. (Este registro es necesario aun cuando las excepciones fueran vectorizadas)
- Causa, un registro usado para conservar la causa de la excepción. En la arquitectura MIPS, este registro es de 32 bits, aunque efectivamente se usan sólo algunos bits. Se

asume que hay un campo de cinco bits que codifica las dos posibles fuentes de excepciones mencionadas anteriormente, con 10 representado una instrucción indefinida y 12 representando overflow aritmético.

Excepciones en una implementación segmentada. Una implementación segmentada trata las excepciones de la misma manera que los riesgos de control. Por ejemplo supongamos que hay un overflow aritmético en una instrucción `add`. De la misma manera que se hizo para los saltos tomados, se deben vaciar (flush) del pipe las instrucciones que siguen a la instrucción de suma, y comenzar a recuperar instrucciones desde una nueva dirección. Se emplea el mismo mecanismo usado para los saltos tomados, pero ahora es una excepción lo que causa que se pongan en cero las señales de control.

Cuando se consideró el caso de un salto que no sigue el comportamiento de la predicción, se explicó cómo vaciar la instrucción en la etapa IF, convirtiéndola en una `nop`. Para vaciar instrucciones en la etapa ID, es necesario usar el multiplexor que ya está en la etapa ID, el cual permite convertir en ceros las señales de control para hacer un atasco. Una nueva señal de control llamada ID.Flush que es operada mediante OR con la señal generada por la unidad de control de riesgos para vaciar (flush) la instrucción en la etapa ID. Para vaciar la instrucción en la etapa EX, se usa una nueva señal EX.Flush, para hacer que los nuevos multiplexores agregados en la etapa EX pongan en cero las señales de control. Para comenzar a recuperar instrucciones desde la dirección 0X800 01800, que es la dirección predefinida para el manejador de excepciones del MIPS, simplemente se agrega una nueva entrada al multiplexor del PC, que carga el valor 0X8000 0180 al PC. La Figura 1 muestra estos cambios.

El ejemplo de la suma que venimos considerando, hace evidente un problema con las excepciones: si no paramos la ejecución en el medio de la instrucción, el programador no podrá ver el valor original del registro $1 que intervino en la ocurrencia del overflow, porque como es el destino de la suma, será pisado por la instrucción. Gracias a una cuidadosa planificación, la excepción por overflow es detectada durante la etapa de ejecución, por lo que puede ser usada la señal EX.Flush para evitar que la instrucción que está en la etapa EX, escriba el registro destino en la etapa WB. Varias excepciones requieren que eventualmente se completen las instrucciones que causaron la excepción como si éstas hubieran ejecutado normalmente. La forma más fácil de hacer ésto es vaciar la instrucción y recomenzarla desde el comienzo, después de haber manejado la excepción.

El paso final es salvar la dirección de la instrucción problemática en el EPC. En realidad se salva el valor PC+4, de manera que la rutina que maneja la excepción primero debe restar 4 al valor salvado. La figura 1 muestra la versión simplificada del camino de datos, incluyendo el hardware para saltos y los arreglos necesarios para manejar excepciones.

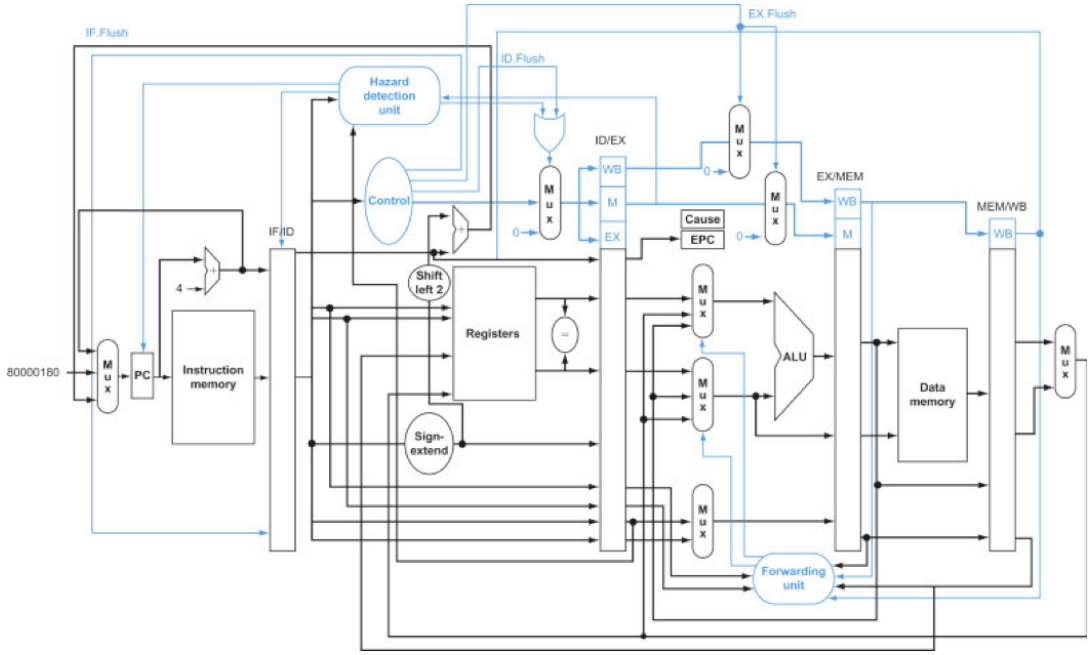

FIGURA 1. **El camino de datos con controles para manejar excepciones.** Las incorporaciones más destacadas incluyen un nuevo valor de entrada con el valor 8000 0180_{hex} en el multiplexor que provee el nuevo valor del PC; un *Registro de Causa* para registrar la causa de la excepción; y un *Registro PC de Excepción* (EPC) para guardar la dirección de la instrucción que causó la excepción. La entrada 8000 0180_{hex} al multiplexor es la dirección inicial para comenzar a recuperar instrucciones en el caso de una excepción. Aunque no se muestra, la señal de overflow de la ALU es una entrada a la unidad de control.

Ejemplo. Excepciones en una computadora segmentada. Dadas las instrucciones:

```
00000040 sub $11,$2,$4
00000044 and $12,$2,$5
00000048 or  $13,$2,$6
0000004C add $1,$2,$1
00000050 slt $15,$6,$7
00000054 lw  $16,50($7)
...
```

asuma que estas son las instrucciones a ser invocadas cuando comienza una excepción:

```
80000180 sw $26,1000($0)
80000184 sw $27,1004($0)
```

¿Qué curre en el procesador si hay una excepción por overflow en la instrucción add?

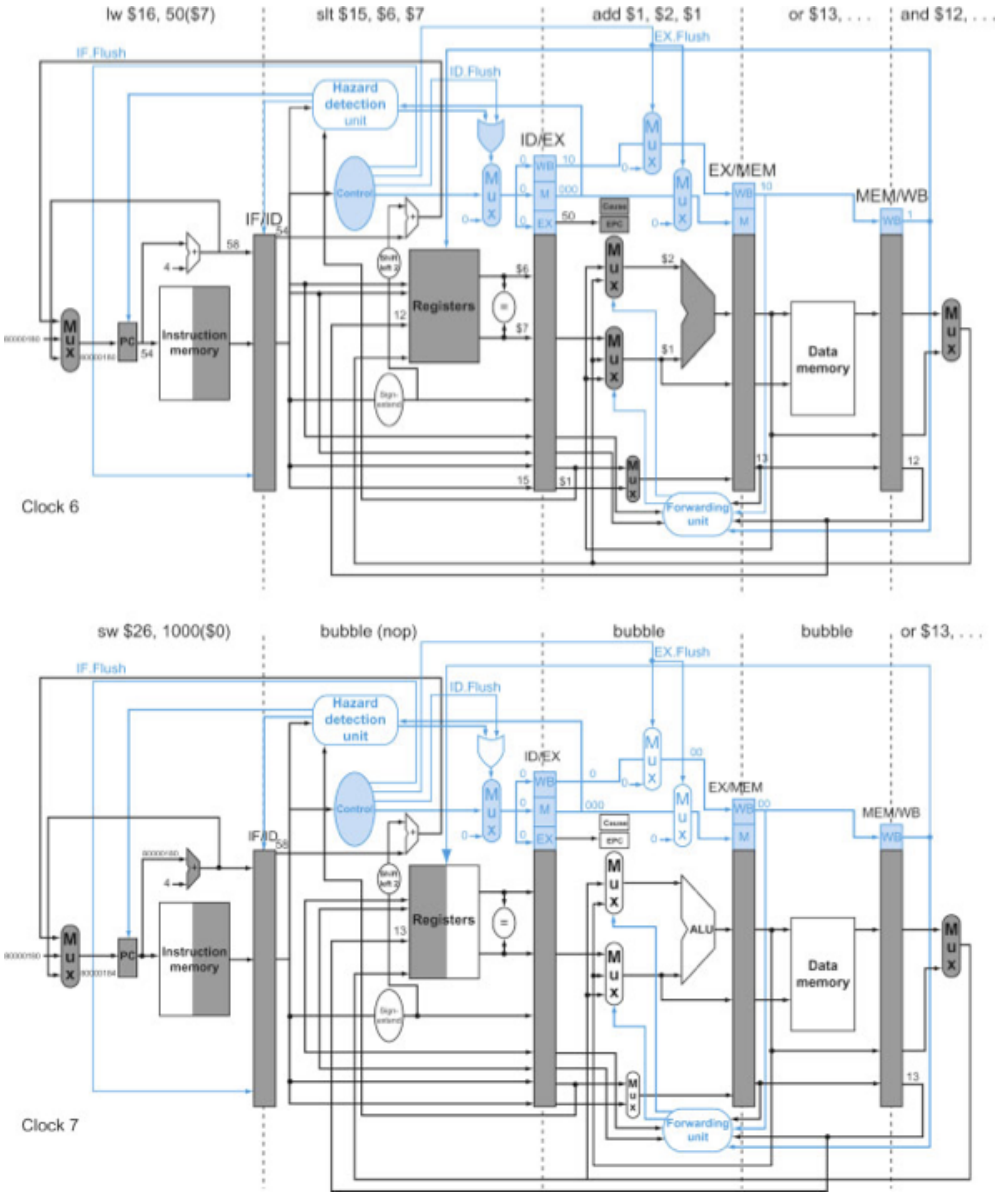

FIGURA 2. El resultado de una excepción debido a un overflow aritmético en la instrucción de suma.

Respuesta. La Figura 2 muestra los eventos, comenzando con la instrucción add en la etapa EX. El overflow es detectado durante esa fase y 8000 0180_{hex} es forzado en el PC. El ciclo 7 muestra que el add y las dos instrucciones que le sigue son vaciadas y la primera

instrucción del código de excepción es recuperada. Notar que la dirección de la instrucción siguiente al add es resguardada: 4C+4=50. □

Anteriormente se mencionaron cinco ejemplos de excepciones y más adelante se verán algunas otras excepciones. Con cinco instrucciones activas en el procesador en cada ciclo, el desafío es asociar una excepción con la instrucción que corresponde. Además, pueden ocurrir simultaneamente múltiples excepciones en un mismo ciclo. La solución es priorizar las excepciones de manera que sea más fácil determinar cual atender primero. En muchas implementaciones MIPS, el hardware ordena las excepciones de manera que la instrucción que ingresó primero sea la interrumpida.

Los requerimientos por dispositivos de E/S y mal funcionamiento del hardware no están asociados con una instrucción específica, de manera que la implementación tiene alguna flexibilidad sobre cuando interrumpir al procesador. Por eso el mecanismo usado para otras excepciones funciona bien.

El EPC captura la dirección de la instrucción interrumpida y el registro de causa del MIPS registra las posibles excepciones en un ciclo de reloj de manera que el software de la excepción debe hacer la correspondencia entre la excepción y la instrucción; una pista importante es conocer que tipo de excepción puede ocurrir en una etapa particular. Por ejemplo una instrucción no definida se descubre en la etapa ID y la invocación al Sistema Operativo ocurre en la etapa de EX. Las excepciones se agrupan en el registro de causa en el campo de excepción pendiente, de manera que el hard puede interrumpir en base a la última excepción, una vez que las anteriores han sido atendidas.

2.1. Interfaz Hardware/Software. El hardware y el sistema operativo deben trabajar en conjunto de manera que las excepciones se comporten de la manera esperada. La responsabilidad del hardware es detener la instrucción con problemas en la mitad del recorrido, permitir que las instrucciones previas completen, vaciar todas las instrucciones que le siguen, cargar un registro donde se mantenga la causa de la excepción, salvar la dirección de la instrucción con problemas y finalmente saltar a la dirección preestablecida. La obligación del sistema operativo es observar la causa de la excepción y actuar de la manera apropiada. En los casos de excepción por: instrucción no definida, falla del hardware u overflow aritmético, el sistema operativo debe eliminar el programa y retornar un indicador que informe el motivo de la eliminación. Para un requerimiento de dispositivo de E/S o una llamada por algún servicio del Sistema Operativo, el SO salva el estado del programa, realiza la acción requerida y en algún momento posterior restablece el programa para continuar con la ejecución. En el caso del requerimiento de un dispositivo de E/S, es habitual elegir otra tarea para ejecutar, antes de restaurar la tarea que solicitó la E/S, debido a que esta tarea posiblemente no pueda continuar hasta que la E/S esté completa. Las excepciones son la razón por la que se necesita poder salvar y recuperar el estado de las tareas. Uno de los usos más frecuentes e importantes de las excepciones es el manejo de las faltas de páginas y las excepciones de TLB (Translation Lookaside Buffer- cache de traducción).

Definición 4. *TLB o Translation Lookaside Buffer, es una cache que mantiene información de las traducciones de direcciones recientemente usadas para tratar de evitar un acceso a la tabla de página.*

Definición 5. *Una interrupción o excepción precisa, es aquella que se asocia siempre con la instrucción correcta.*

Elaboración. La dificultad de asociar siempre correctamente una excepción con la instrucción correspondiente en un procesador segmentado, ha llevado a algunos diseñadores de computadoras a relajar este requerimiento en los casos que no son críticos. Tales procesadores se dice que tienen interrupciones o excepciones *imprecisas*. En el ejemplo anterior, el PC normalmente tendría 0X58 al comienzo del ciclo luego que se detectó la excepción, aun cuando la instrucción con problemas está en la dirección 0X4C. Un procesador con excepciones imprecisas puede poner 0X58 en el EPC y dejar que el sistema operativo determine que instrucción causó el problema. MIPS y la mayoría de las computadoras de hoy día soportan interrupciones precisas. (Esto es así para soportar memoria virtual).□
Elaboración. Aunque MIPS use la dirección 0X8000 0180 como entrada de excepción para casi todas las excepciones, en el caso de las excepciones por falla del TLB, se usa la dirección 0X8000 0000 para mejorar el desempeño del manejador de excepciones.□
Verificar por uno mismo. ¿Cuál excepción debiera ser reconocida primero en la siguiente secuencia?

```
add $1,$2,$1 #overflow aritm'etico
??? $1,$2,$1 #instrucci'on indefinida
sub $1,$2,$1 #error de hardware
```

Respuesta: la primera, porque en la secuencia lógica es la instrucción que comienza antes.□

3. Paralelismo mediante instrucciones

La segmentación explota el paralelismo potencial entre instrucciones[3]. Este paralelismo es llamado *paralelismo a nivel de instrucción* (PNI), o en inglés *instruction-level parallelism* (ILP).

Paralelismo a nivel de instrucción: Es el paralelismo entre las instrucciones.

Hay dos métodos principales para explotar la cantidad potencial de PNI: La primera es incrementar la profundidad del pipeline para superponer más instrucciones. Usando la analogía con el ejemplo de la lavandería y asumiendo que el ciclo del lavado fuese el más lento, sería como dividir el lavado tradicional y usar tres máquinas diferentes:una que lave, otra que enjuage y otra que centrifugue. Se podría pasar de un pipe de 4 etapas a uno de 6 etapas. Para conseguir una aceleración real, se deben re-balancear las otras etapas de manera que ellas tengan la misma duración, tanto en el procesador como en la lavandería. La cantidad de paralelismo que se puede explotar es mayor, debido a que hay más operaciones que se pueden superponer. El desempeño es potencialmente mayor debido a que el ciclo de reloj es menor.

[3]Esta sección es la traducción de la sección 4.10 del libro de Patterson y Hennessy [1].

Emisión múltiple: Es un esquema donde múltiples instrucciones son lanzadas en un ciclo de reloj.

Otra aproximación es replicar las componentes internas de la computadora de manera que pueda lanzar múltiples instrucciones en cada etapa del pipeline. El nombre general para esta técnica es *emisión múltiple*. Una lavandería de despacho múltiple podría reemplazar nuestra lavadora y el secarropa hogareños por tres lavadoras y tres secadoras. También se deberían reclutar más asistentes para doblar y guardar el triple de lavado en el mismo tiempo. La desventaja es el trabajo extra de mantener todas las máquinas ocupadas y pasar las cargas a la siguiente etapa del pipe. Lanzar múltiples instrucciones por etapa permite que la tasa de ejecución de instrucciones exceda la tasa de reloj, o puesto de otra manera que el CPI sea menor que uno. A veces es útil invertir la métrica y usar la *Cantidad de Instrucciones por Ciclo de Reloj* o IPC por sus siglas en inglés: *Intructions Per Clock Cycle*. Digamos un microprocesador de 4GHz de despacho múltiple de cuatro vías puede ejecutar a una tasa máxima de 16 mil millones de instrucciones por segundo y en el mejor caso tener un CPI de 0,25 o un IPC de 4. Asumiendo un pipeline de 5 etapas eso quiere decir que en todo momento habrán 20 instrucciones en ejecución. Los microprocesadores de alto desempeño de hoy en día intentan despachar entre 3 a 6 instrucciones por ciclo. Incluso los diseños moderados intentan alcanzar un IPC máximo de 2. Típicamente hay restricciones sobre el tipo de instrucciones que se pueden ejecutar al mismo tiempo y sobre lo que ocurre cuando surgen dependencias.

Despacho múltiple estático: Una aproximación para implementar procesadores de emisión múltiple donde varias decisiones son tomadas por el compilador antes de la ejecución.

Despacho múltiple dinámico: Una aproximación para implementar un procesador de emisión múltiple donde varias decisiones son tomadas por el procesador durante la ejecución.

Hay dos formas principales para implementar un procesador de despacho múltiple, donde la principal diferencia se encuentra en la división de trabajo entre el compilador y el hardware. Como la división de trabajo determina si las decisiones se hacen de manera estática (en tiempo de compilación) o dinámica (en tiempo de ejecución), las aproximaciones a veces son llamadas emisión múltiple estática o emisión múltiple dinámica. Como veremos ambas aproximaciones tienen otros nombres comúnmente usados, los cuales son menos precisos o más restrictivos.

Ranuras de despacho: Los puestos desde los cuales las instrucciones pueden ser emitidas en un determinado ciclo de reloj; por analogía corresponden a los puestos de partida en una carrera.

Hay dos responsabilidades principales y distintas con las que se debe tratar en un pipeline de despacho múltiple.

1. Empaquetar instrucciones en ranuras de despacho: ¿Cómo hace el procesador para determinar cuántas y cuáles instrucciones pueden ser despachadas en un mismo

ciclo?. En muchos procesadores de despacho estático, este proceso es llevado a cabo al menos parcialmente por el compilador; en los diseños de procesadores de despacho múltiple dinámico, es tratado normalmente en tiempo de ejecución por el procesador, aunque el compilador frecuentemente ya ha procurado ayudar a mejorar la tasa de despacho ubicando las instrucciones en el mejor orden.

2. Tratamiento de los riesgos de dato y de control: en procesadores de despacho estático, el compilador maneja estáticamente algunas o todas las consecuencias de los riesgos de datos y control. En contraste muchos procesadores dinámicos intentan aliviar al menos una clase de riesgo usando técnicas de hardware que operan en tiempo de ejecución.

Aunque se describen como dos aproximaciones distintas, en realidad frecuentemente cada una de las aproximaciones toma cosas de la otra y ninguna aproximación puede reclamar ser perfectamente pura.

4. El concepto de especulación

Uno de los métodos más importante para lograr y explotar más PNI es la especulación. Este método se basa en la idea de predicción, la especulación es una aproximación donde el compilador o el procesador "adivinan" las propiedades de una instrucción y así poder iniciar otras que dependen de la instrucción especulada.

> ***Especulación:*** es una aproximación por la cual el compilador o el procesador adivinan el resultado de una instrucción y de esa manera deja de ser una dependencia en la ejecución de otras instrucciones.

Por ejemplo, se puede especular sobre el resultado de un branch, de manera que las instrucciones que siguen al branch puedan ser iniciadas sin demoras. Otro ejemplo en el cual se puede especular es con las instrucciones *store* que preceden un *load*, considerando que no se refieren a la misma dirección de memoria. De esta manera se podría permitir que el load ejecute antes que el store. La dificultad con la especulación es que puede ser errónea. Por esa razón cualquier mecanismo de especulación debe incluir un método para verificar si la suposición fue correcta y un método para deshacer o volver atrás los efectos de la instrucción que fue ejecutada de manera especulativa. La implementación de esta capacidad de volver hacia atrás agrega complejidad. La especulación puede ser realizada por el compilador o el hardware. Por ejemplo, el compilador puede usar especulación para reordenar instrucciones, mover una instrucción antes de un salto o un load antes de un store. El hardware del procesador puede realizar los mismos cambios en tiempo de ejecución usando técnicas que se discuten más adelante. Los mecanismos de recuperación usados en caso de una especulación incorrecta, son bastante diferentes. En el caso de especulación por software, se agregan instrucciones que controlan la exactitud de la especulación y se provee una rutina de corrección que es invocada cuando la especulación es incorrecta. En especulación por hardware, el procesador usualmente guarda los resultados especulativos en buffers, que no son grabados en memoria o registros hasta verificar que la especulación es correcta. En caso que la especulación sea incorrecta, el hardware descarta el contenido de los buffers y se vuelve atrás para ejecutar la secuencia de instrucciones correcta. La especulación introduce

otro posible problema: especular sobre ciertas instrucciones puede introducir excepciones que originalmente no estaban presentes. Por ejemplo, suponga que una instrucción de load es movida de manera especulativa, pero la dirección que usa no es legal cuando la especulación es incorrecta. El resultado podría ser una excepción que no debería haber ocurrido. El problema es complicado por el hecho que si la instrucción load no fuera especulativa, la excepción debe ocurrir. En la especulación basada en el compilador, tales problemas son evitados agregando soporte adicional de especulación que permite que tales excepciones sean ignoradas hasta determinar que realmente deben ocurrir. En especulación basada en hardware las excepciones simplemente son colocadas en buffers hasta que es claro que la instrucción que causó la excepción no es más especulativa y está lista para terminar; en ese punto se permite que la excepción surja y se procede con el manejo normal. Debido a que la especulación puede mejorar el desempeño cuando es hecha apropiadamente y empeorar el desempeño cuando se hace descuidadamente, se dedica mucho esfuerzo para decidir cuando es adecuado especular. Más adelante se analizan las técnicas para especulación estática y dinámica.

5. Emisión múltiple estática

Todos los procesadores de múltiple emisión estática usan al compilador como asistente para empaquetar las instrucciones y para manejar los riesgos. En un procesador de emisión estática, es posible pensar que en un único ciclo de reloj se emite un conjunto de instrucciones, lo que se llama paquete de emisión, es como una instrucción grande con múltiples operaciones.

> ***Paquete de emisión:*** es el conjunto de instrucciones que se emiten juntas en un ciclo de reloj, el paquete puede ser determinado estáticamente por el compilador o dinámicamente por el procesador.

Esta visión es más que una analogía. Debido a que el procesador de emisión estática múltiple usualmente restringe la mezcla de instrucciones que pueden ser iniciadas en un determinado ciclo de reloj, es útil pensar que el paquete emitido (issue) es una única instrucción que permite varias operaciones en ciertos campos predefinidos. Esta visión conduce al nombre original de esta aproximación: Palabra de Instrucción Muy Grande (en inglés very long instruction word o VLIW). Muchos procesadores de emisión estática también se basan en el compilador para tomar alguna responsabilidad para manejar riesgos de datos y de control. Veremos una versión simple de emisión estática de un procesador MIPS antes de describir esta técnica en procesadores más agresivos.

Ejemplo. Emisión múltiple estática con la ISA MIPS. Para tener una idea de lo que es la emisión múltiple estática, se considera un procesador MIPS de despacho dual simple, donde una de las instrucciones debe ser una operación de ALU entera o un salto y la otra debe ser load o store. Este diseño es como el usado en algunos procesadores MIPS embebidos. Emitir dos instrucciones por ciclo requiere recuperar y decodificar 64 bits de instrucción. En muchos procesadores de emisión estática múltiple, y esencialmente en todos los procesadores VLIW, la estrategia para despachar instrucciones simultáneamente se restringe a simplificar la decodificación y despacho de instrucciones. Por lo cual, se requiere

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURA 3. **Procesador de despacho dual estático en operación.** Las instrucciones de ALU y de transferencia de datos son emitidas al mismo tiempo. Se asume la misma estructura de cinco etapas que la usada para la segmentación de emisión simple. Aunque no es estrictamente necesario, esto tiene algunas ventajas. En particular mantener la escritura en registro al final de la tubería simplifica el manejo de las excepciones y el mantenimiento del modelo de excepción precisa, lo cual se hace más difícil en procesadores de emisión múltiple.

que las instrucciones sean apareadas y alineadas sobre límites de 64 bits, con la porción de ALU o branch apareciendo primero. Además si una instrucción del par no puede ser usada, se requiere que sea reemplazada con una no-operación. De esa manera las instrucciones siempre se despachan en pares, posiblemente con una nop en una de las ranuras. La Figura 3 muestra cómo se ven las instrucciones cuando de a pares recorren el pipe. Los procesadores de emisión estática múltiple varían en la forma de manejar potenciales riesgos de control y de dato. En algunos diseños el compilador asume toda la responsabilidad de eliminar todos los riesgos, planificar el código e insertar las nops. De esa manera el código ejecuta sin requerir la detección de riesgos, ni perjudicarse por los atascos que genere el hardware. En otros, el hardware detecta los riesgos de datos y genera los atascos entre la emisión de dos paquetes, mientras que confía en que el compilador evitará todas las dependencias dentro de cada paquete. Aun así, un riesgo generalmente fuerza el atasco del paquete de emisión que conteniene la instrucción dependiente. Si el software debe manejar todos los riesgos o solamente procurar reducir la fracción de riesgos entre despachos de paquetes, refuerza la idea de tener una única instrucción grande con múltiples operaciones. En este ejemplo se asume la segunda alternativa.

Para despachar una operación de ALU y una de transferencia de datos en paralelo, primero es necesario hardware adicional — más allá de la lógica de detección de riesgos y atascos — como puertos extras en el banco de registros (Ver Figura 4). En un ciclo de reloj es posible leer dos registros para la operación de ALU y dos más para un store, también un puerto de escritura para la operación de ALU y uno de escritura para el load. Debido a que la operación de ALU ocupa la ALU, es necesario agregar un sumador para calcular la dirección efectiva de la operación de transferencia de datos. Sin estos recursos extras el pipeline de despacho dual estaría obstruido por los riesgos estructurales.

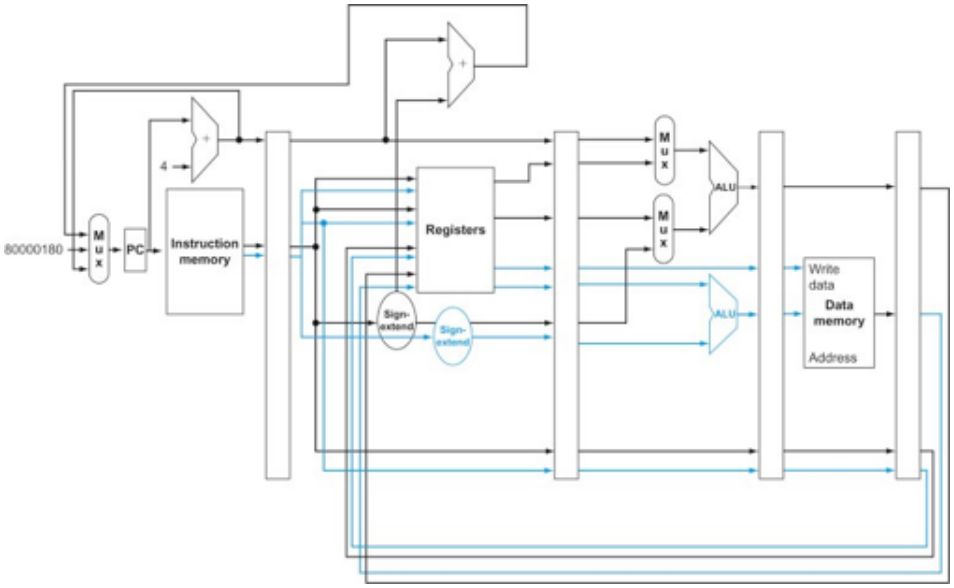

FIGURA 4. **Un camino de datos de emisión dual estática.** Las necesidades adicionales para la emisión dual son resaltadas: otros 32 bits de la memoria de instrucciones, en el banco de registros dos puertos más de lectura y uno más de escritura y otra ALU. Se asume que la ALU de abajo maneja cálculo de direcciones para transferencia de datos y la ALU de arriba maneja todo lo demás.

Claramente este procesador de emisión dual puede mejorar el desempeño por un factor de dos. Para hacerlo, sin embargo, se requiere que el doble de instrucciones sean superpuestas en ejecución y esta superposición adicional incrementa la pérdida de desempeño relativo debido a los riesgos de datos y control. Por ejemplo en el pipe simple de cinco etapas los load tienen una latencia de uso de un ciclo, lo que impide que una instrucción use el resultado sin atasco. En un pipeline de cinco etapas con emisión dual, el resultado de una instrucción load no puede ser usado en el siguiente ciclo de reloj. Esto significa que las siguientes dos instrucciones no pueden usar el resultado del load sin atasco. Además las instrucciones de ALU que no usan latencia en el pipe simple de cinco etapas ahora tienen una latencia de uso de una instrucción, debido a que los resultados no pueden ser usados en el load o store del mismo paquete. Para explotar efectivamente el paralelismo disponible en un procesador de despacho múltiple, compiladores más ambiciosos o técnicas de planificación de hardware son necesarias y el despacho múltiple estático requiere que el compilador tome este rol.

Ejemplo. Planificación simple del código con emisión múltiple. ¿Cómo se podría planificar este loop sobre un pipeline de emisión doble estática para MIPS?

```
Loop: lw   $t0, 0($s1)          # $t0=array element
      addu $t0, $t0, $s2        # add scalar in $s2
      sw   $t0, 0($s1)          # store result
      addi $s1, $s1,-4          # decrement pointer
      bne  $s1, $zero, Loop     # branch $s1!=0
```

Reordene las instrucciones para evitar tantos atascos como sea posible. Asuma que los saltos se predicen correctamente, de manera que los riesgos de control son manejados por el hardware.

Respuesta. Las primeras tres instrucciones tienen dependencias de datos y lo mismo con las dos últimas. La Figura 5 muestra la mejor planificación para estas instrucciones. Notar que solamente un par de instrucciones han usado las dos ranuras de emisión. Se requieren cuatro ciclos por iteración del loop; en cuatro ciclos se ejecutan cinco instrucciones, se alcanza un insuficiente CPI de 0,8 cuando el mejor caso es de 0,5. O lo que es equivalente, un IPC de 1,25 versus 2,0. Notar que para el cálculo de CPI o IPC las no-op‘s ejecutadas no se cuentan como operaciones útiles. Hacerlo podría mejorar el CPI pero no el desempeño. □

	ALU or branch instruction	Data transfer instruction	Clock cycle
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 5. **El código reordenado como podría ser visto por un procesador MIPS de despacho dual.** Las ranuras vacías son no-ops.

Una técnica del compilador importante para mejorar el desempeño en los loops es el desenrollado de loops, donde se hacen múltiples copias del cuerpo del loop. Después del desenrollado, hay disponible más paralelismo a nivel de instrucción por la superposición de instrucciones de distintas iteraciones. El desenrollado de loops es una técnica para conseguir mejor desempeño en los loops que acceden a arreglos, en los que se hacen múltiples copias del cuerpo del loop y las instrucciones de diferentes iteraciones se pueden solapar.

Ejemplo. Desenrrollado de loops para pipelines de despacho múltiple. Observar lo bien que funciona el desenrollado y la planificación para el ejemplo anterior. Por simplicidad se asume que el índice del loop es múltiplo de 4.

Respuesta. Para planificar el loops sin ninguna demora, es conveniente hacer cuatro copias del cuerpo del loop. Después de desenrollar y eliminar las instrucciones de control innecesarias, el loop contendrá cuatro copias de cada lw, add, sw más un addi y un bne. La Figura 6 muestra el código desenrollado y planificado.

Durante el desenrollado, el compilador introduce registros adicionales ($t1, $t2, $t3). El objetivo de este proceso, llamado renombramiento de registros, es eliminar dependencias que no son verdaderas dependencias de datos, pero que pueden llevar a potenciales riesgos o impedir que el compilador planifique el código de una manera flexible. Esto último se

	ALU or branch instruction	Data transfer instruction	Clock cycle
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 6. **El código desenrrollado y reordenado de la Figura 5 como podría ser visto en un procesador MIPS de despacho dual.** Las ranuras vacías son no-operaciones. Debido a que la primer instrucción del loop decrementa $s1 por 16, las sucesivas direcciones cargadas son: el valor original de $s1, luego esa dirección menos 4, menos 8 y menos 12.

observa claramente en el caso de que el código desenrollado sólo hubiese usado el registro $t0. En esa situación habrían secuencias repetidas de lw $t0, 0($s1) add $t0, $t0, $s2 seguido por sw $t0, 4($s1), pero estas secuencias son completamente independientes excepto por el hecho de usar $t0 –ningún valor de dato fluye de un conjunto de instrucciones al siguiente conjunto de instrucciones. Este caso es lo que llamamos antidependencia o dependencia de nombre, lo cual es un ordenamiento forzado puramente por el uso de un nombre más que una dependencia real de datos. El renombramiento de registros durante el desenrollado permite que el compilador mueva estas instrucciones independientes para conseguir una mejor planificación del código. El proceso de renombramiento limita las dependencias de nombre mientras que preserva las dependencias reales. Notar que 12 de las 14 instrucciones en el loop ejecutan de a pares. Eso le toma 8 ciclos para las cuatro iteraciones o 2 ciclos por iteración, lo que produce un CPI de 8/14=0,57. El desenrollado y planificación con despacho dual da un factor de mejora cercano a 2, en parte debido a la reducción de las instrucciones de control del loop y en parte debido a la ejecución de despacho dual. El costo en la mejora del desempeño es el uso de 4 registros temporarios en lugar de uno solo, también el aumento en el tamaño del código. □

6. Procesadores de emisión múltiple dinámica

Los procesadores de emisión múltiple dinámica son conocidos como procesadores super-escalares o simplemente como super-escalares. En el procesador super-escalar más simple la emisión de instrucciones es en orden y el procesador decide si cero, una o más instrucciones se pueden emitir en un determinado ciclo. Obviamente alcanzar buen desempeño en tales procesadores requiere que el compilador planifique las instrucciones procurando eliminar las dependencias y en consecuencia mejorar la tasa de emisión de instrucciones. Aun con esa planificación, hay una importante diferencia entre este super-escalar simple y un procesador VLIW: el código, sea que está planificado o no, está garantizado por el hardware que

ejecutará correctamente. Además el código compilado siempre corre correctamente independientemente de la tasa de emisión o estructura del pipeline del procesador. En algunos diseños VLIW, no se cumple lo anterior y la recopilación se hace necesaria cuando se cambia el modelo del procesador; en otros procesadores de despacho estático, el código puede correr correctamente a través de diferentes implementaciones, pero generalmente es tan pobre que la recopilación se hace necesaria. Varios procesadores super-escalares extienden el sistema básico de decisiones de emisión dinámica para incluir *planificación dinámica del pipeline*. La planificación dinámica del pipeline elige que instrucciones ejecutar en determinado ciclo de reloj mientras trata de evitar riesgos y atascos.

Planificación dinámica del pipeline: es el hardware de soporte para reordenar la secuencia de ejecución de instrucciones procurando evitar atascos.

Comencemos con un ejemplo simple donde se procura evitar riesgos de dato.
Consideremos la siguiente secuencia de código:

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

Aun cuando la instrucción **sub** está lista para ejecutar, debe aguarda que la instrucciones **lw** y **add** finalicen, lo cual puede tomar varios ciclos si la memoria es lenta. La planificación dinámica del pipeline permite que tales riesgos sean evitados total o parcialmente.

6.1. Planificación dinámica del pipeline. La planificación dinámica del pipeline elige que instrucciones serán luego ejecutadas, posiblemente reordenandolas para evitar atascos. En estos procesadores, el pipeline es dividido en tres unidades principales: una unidad de fetch y emisión de instrucciones, unidades funcionales múltiples y una unidad de consolidación o commit.

La Figura 7 muestra el modelo. La primera unidad recupera instrucciones, las decodifica y envía cada instrucción a la correspondiente unidad funcional para ejecutar. Cada unidad funcional tiene un buffer, llamado *estación de reservación*, la cual mantiene los operandos y la operación. Ni bien el buffer contenga todos los operandos y la unidad funcional está lista para ejecutar, se calcula el resultado. Cuando el resultado se completa, es enviado a cualquier estación de reservación que esté esperando por este resultado particular, también se envía a la *unidad de consolidación* (commit) que se encarga de mantener el resultado hasta que es seguro poner el resultado en el registro o en memoria en el caso de un store. El buffer en la unidad de consolidación frecuentemente es llamado *buffer de reordenamiento*, también es empleado para proveer operandos de una manera similar a como la lógica de adelantamiento lo hace en un pipe de planificación estática. Una vez que el resultado es descargado en el banco de registros, puede ser recuperado directamente desde allí, de la misma manera que el pipe normal.

La combinación de resguardar los operando en las estaciones de reservación y los resultados en el buffer de reordenamiento, provee una forma de renombramiento de registros

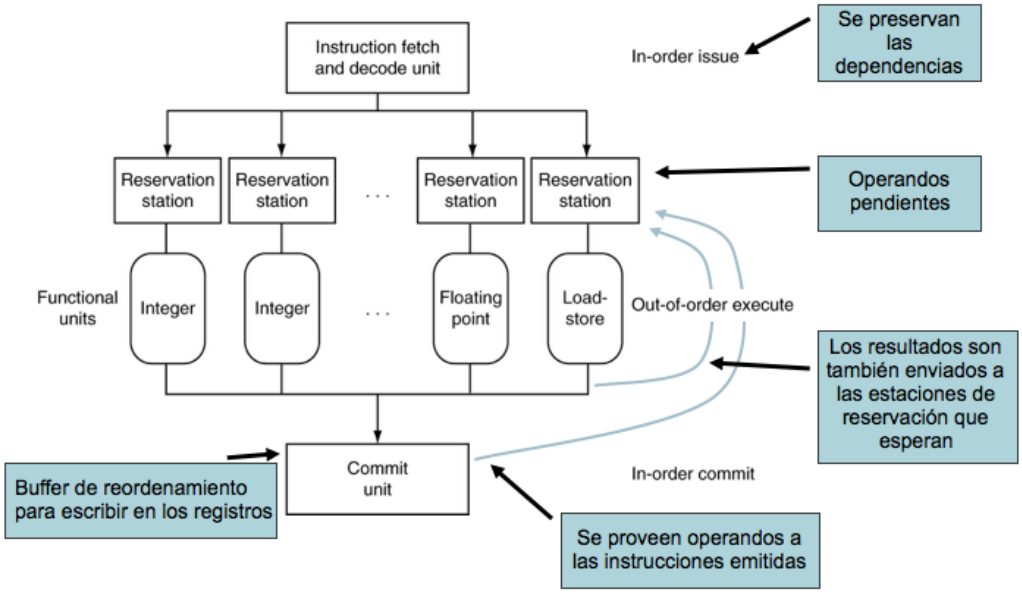

FIGURA 7. **Las tres unidades primarias de un pipiline planificado dinámicamente.** El paso final de actualización del estado también es llamado retiro o graduación.

similar a la empleada por el compilador en el ejemplo de desenrollado del loop, para ver como eso funciona conceptualmente, considere los siguientes pasos:

1. Cuando una instrucción es emitida, se copia en las estaciones de reservación de la unidad funcional apropiada. Cualquier operando que está disponible en el banco de registros o en el buffer de reordenamiento, es copiado en la estación de reservación. La instrucción espera en ese lugar hasta que los operando y la unidad funcional estén disponibles. Para la instrucción emitida la copia del operando en registro no se requiere más y si hay una escritura sobre ese registro, el valor allí guardado ya podría ser sobrescrito sin problema.
2. Si un operando no está en el archivo de registros, ni en el buffer de reordenamiento, se debe aguardar a que sea producido por alguna unidad funcional. El nombre de la unidad funcional que producirá el resultado queda registrado. Cuando aquella unidad eventualmente produzca el resultado, éste es copiado directamente en las estaciones de reservación de las unidades funcionales, sin pasar por los registros.

Estos pasos efectivamente usan el buffer de reordenamiento y las estaciones de reservación para implementar el renombramiento de registros. Conceptualmente, se puede pensar en la planificación dinámica del pipeline como algo similar a realizar el análisis de la estructura de flujo de datos del programa. El procesador entonces ejecuta las instrucciones en algún orden que preserva el orden de flujo de datos del programa. Este estilo de ejecución generalmente

es llamado *ejecución fuera de orden* debido a que las instrucciones pueden ser ejecutadas en un orden distinto al que fueron recuperadas.

> ***Ejecución fuera de orden:*** Es una situación que ocurre en la ejecución segmentada en la cual, una instrucción que está bloqueada para ejecutar no causa que las siguientes instrucciones deban esperar.

Para hacer que los programas se comporten como si estuviesen corriendo en un pipeline en orden, se requiere que la unidad de recuperación y decodificación de instrucciones emita las instrucciones en orden, lo que permite que las dependencias sean registradas y la unidad de finalización debe necesariamente escribir los resultados a los registros y la memoria en el orden de recuperación del programa. Este modo conservador es llamado *consolidación en orden*. Por lo tanto si ocurre una excepción, la computadora puede indicar cual fue la última instrucción ejecutada y los únicos registros actualizados serán aquellos escritos por las instrucciones previas a la instrucción que causó la excepción. Aunque el extremo inicial y el extremo final corren en orden, las unidades funcionales tienen libertad para iniciar la ejecución cuando los datos que necesitan están disponibles. Hoy en día todos los pipelines planificados dinámicamente usan finalización en orden.

> ***Consolidación (commit) en orden:*** Es una consolidación en la cual los resultados de la ejecución segmentada son grabados en el estado visible al programador en el mismo orden que las instrucciones son recuperadas.

La planificación dinámica normalmente es extextendida para incluir especulación basada en hardware, especialmente en el resultado de los saltos. Prediciendo la dirección del salto, un procesador planificado dinámicamente puede continuar recuperando y ejecutando instrucciones de la rama predicha. Debido a que las instrucciones finalizan en orden, sabemos si el salto fue o no correctamente predicho antes que finalice cualquier instrucción de la rama predicha. Un pipeline que tenga planificación dinámica y además emplee especulación, puede también soportar especulación a cerca de la dirección en la que se realiza un load. Es decir, permitir el reordenamiento de load-store, usando la unidad de consolidación para evitar especulaciones erróneas.

6.2. Entendiendo el desempeño de los programas. Considerando que los compiladores también pueden reordenar el código en base a las dependencias de dato, uno se puede preguntar: ¿por qué un procesador super-escalar podría usar planificación dinámica?. Hay tres razones. Primero, no todos los atascos son predecibles. En particular los fallos de cache en la jerarquía de memoria producen atascos impredecibles. La planificación dinámica permite que el procesador oculte algunos de estos atascos, continuando la ejecución de instrucciones mientras espera que el atasco termine.

Segundo, si el procesador especula sobre el resultado de un salto usando predicción dinámica de saltos, no puede conocer el orden exacto de las instrucciones en tiempo de compilación, debido a que depende del comportamiento predicho y del comportamiento real de los saltos. La incorporación de predicción dinámica para explotar más paralelismo a nivel de instrucción (PNI) sin incorporar planificación dinámica podría significativamente restringir los beneficios de la especulación.

Tercero, en la medida que cambia la cantidad de instrucciones emitidas y la latencia del pipeline, también cambia la mejor manera de compilar una secuencia de código. Por ejemplo, cómo una secuencia de instrucciones dependientes se puede planificar, depende del ancho de emisión y de la latencia. La estructura del pipeline afecta tanto el número de veces que el cuerpo de un loop debe ser desenrrollado para evitar atascos, como del proceso de renombramiento de registros basado en el compilador. La planificación dinámica permite que el hardware oculte muchos de estos detalles. De esa manera, los usuarios y los distribuidores de software no se deben complicar con tener múltiples versiones de un programa para diferentes implementaciones del conjunto de instrucciones. De la misma manera, el código viejo heredado se beneficiará de una nueva implementación sin necesidad de recompilación.

La perspectiva amplia: Tanto la segmentación como la emisión múltiple incrementan el rendimiento (throughput) e intentan explotar el paralelismo a nivel de instrucción (PNI). Las dependencias de dato y control en los programas, sin embargo, establecen un límite superior en el desempeño alcanzado porque el procesador a veces debe aguardar que una dependencia se resuelva. La aproximación centrada en software para explotar el PIN descansa en la habilidad del compilador para encontrar y reducir los efectos de tales dependencias, por su parte la aproximación centrada en hardware se basa en las extensiones al pipeline y en mecanismos de emisión. La especulación, realizada por el compilador o el hardware, puede incrementar el PIN que puede ser explotado por medio de la predicción, aunque, algún cuidado se debe tener debido a que especular erroneamente probablemente reduzca el desempeño.

La intefaz Hardware/Software: Los microprocesadores de alto desempeño son capaces de emitir varias instrucciones por ciclo, desafortunadamente, mantener esa tasa de emisión es muy difícil. Por ejemplo, más allá de la existencia de procesadores con cuatro o seis emisiones por ciclo de reloj, muy pocas aplicaciones pueden mantener más de dos instrucciones por ciclo. Existen dos razones principales para esto.

Primero, dentro del pipeline el principal cuello de botella del desempeño surge de las dependencias que no pueden ser superadas, esto reduce el paralelismo entre las instrucciones y una tasa de emisión sostenida. Lamentablemente es poco lo que se puede hacer con las dependencias de dato reales, ya que frecuentemente el compilador o el hardware, no conocen exactamente si existe o no una dependencia, por lo tanto, son conservadores y deben asumir que la misma existe. Por ejemplo, el código que hace uso de punteros, especialmente en formas que pueden llevar a aliasing[4], implica mayor dependencia potencial. En contraste, una mayor regularidad en el acceso a los arreglos, frecuentemente permite que el compilador deduzca que no existe depedencia. De la misma manera, los saltos que no pueden ser anticipados correctamente, ya sea por el compilador o en tiempo de ejecución, limitarán la posibilidad de aprovechar el PNI. Con frecuencia, hay más PNI que está disponible, pero

[4]En procesamiento de señales, computación gráfica y disciplinas relacionadas, el aliasing es el efecto que causa que señales continuas distintas se tornen indistinguibles cuando se muestrean digitalmente.

es limitada la habilidad del compilador o del hardware (a veces requiere la ejecución de cientos de instrucciones) para encontrar PNI que pueda ser claramente identificado.

Segundo, los fallos en la jerarquía de la memoria, también limitan la posibilidad de llenar completamente la tubería. Algunos atascos del sistemas de memoria pueden ser ocultos, pero cantidades limitadas de PNI también reducen la posibilidad de que tales atascos sean transparentes.

Referencias

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

[2] Virgil Bistriceanu, Laboratorio 7: Exceptions in MIPS, URL:http://www.cs.iit.edu/ virgil/cs470/Labs/Lab7.pdf,1996