Capítulo 2

PROCESADORES SUPERESCALARES

2.1. Guión-esquema

Los contenidos que se tratan a lo largo del tema se resumen en los siguientes puntos:

- Características y arquitectura genérica de un procesador superescalar.
- Problemática de la ejecución de instrucciones fuera de orden.
- Técnicas de prelectura y lectura de instrucciones para mejorar el ancho de banda.
- Técnicas dinámicas para la predicción del resultado y dirección de las instrucciones de salto.
- Técnicas de predecodificación, decodificación y traducción de instrucciones.
- Distribución de instrucciones con y sin lectura de operandos.
- Buffer de distribución. Estaciones de reserva centralizadas, individuales y compartidas.
- Renombramiento de registros.
- Emisión y finalización de instrucciones fuera de orden.
- Gestión de dependencias falsas de datos y de memoria.
- Buffer de terminación. Mantenimiento de la consistencia del procesador.
- Buffer de almacenamiento y de cargas. Mantenimiento de la consistencia de la memoria.
- Tratamiento de interrupciones.

2.2. Introducción

Una vez estudiados los fundamentos en los que se basan los procesadores segmentados, también denominados escalares, se está en condiciones de abordar el estudio de la arquitectura y el funcionamiento de los procesadores superescalares. Para darse una idea de la importancia que esta clase de procesadores tiene actualmente, todos los procesadores de propósito general son superescalares; incluso, los procesadores con varios núcleos o multi-core no son más que una combinación de varios procesadores superescalares conectados dentro de un mismo circuito integrado. Por otra parte, con independencia del tipo del repertorio de instrucciones que hoy en día maneje un procesador, esto es, CISC o RISC, su arquitectura es superescalar.

Al igual que un procesador segmentado, un procesador superescalar se caracteriza por intentar aumentar al máximo el número de instrucciones ejecutadas por ciclo de reloj, aprovechando para ello el paralelismo implícito a nivel de instrucción (ILP - *Instruction-Level Paralelism*). La principal diferencia con respecto a un procesador escalar es que un procesador superescalar procesa varias instrucciones simultáneamente en cada una de las etapas de que consta su segmentación. Recuerde que un procesador escalar solo puede mantener una instrucción en cada una de sus etapas, de forma que si la profundidad de su segmentación es cinco, el procesador mantendrá en su cauce un máximo de cinco instrucciones por ciclo. Un procesador superescalar actual puede llegar a estar manejando más de cien instrucciones por ciclo de reloj, distribuidas de forma desigual entre sus diferentes etapas.

Este capítulo comienza con una descripción de las características más importantes de un procesador superescalar para, a continuación, analizar las etapas de que consta su segmentación. Por lo general, estas etapas son: lectura de instrucciones, decodificación, distribución, ejecución, terminación y retirada. A lo largo del capítulo se estudian los problemas que surgen en cada etapa al tener que manejar varias instrucciones simultáneamente, prestando especial atención a la gestión de las dependencias de datos y de memoria tanto verdaderas como falsas. La gestión correcta de estas dependencias es un tema clave ya que otra de las características notorias de estos procesadores es su capacidad para ejecutar instrucciones fuera de orden. Muchas de las técnicas que se utilizan en un procesador superescalar son las mismas que se aplican en un procesador segmentado básico, como, por ejemplo, los predictores estáticos de saltos. La principal diferencia es que, ahora, adquieren una relevancia aún mayor ya que la detención de una segmentación superescalar o un fallo de predicción conlleva la pérdida de muchos ciclos de reloj.

Los objetivos que se pretenden alcanzar con el estudio de este capítulo son:

- Conocer las características de los procesadores superescalares y sus diferencias con respecto a una segmentación escalar.
- Entender la problemática que plantean las etapas de una segmentación superescalar.
- Saber qué son el *front-end*, el núcleo de ejecución fuera de orden y el *back-end* de un procesador superescalar.
- Conocer las técnicas dinámicas más utilizadas para la especulación del resultado y la dirección de destino de una instrucción de salto.

- Conocer técnicas que mejoran el ancho de banda del procesador, como la prelectura, la predecodificación y la traducción de instrucciones.
- Comprender la finalidad del renombramiento de registros y conocer las diferentes formas que existen para incorporar esta técnica en un procesador superescalar.
- Entender la relevancia de las etapas de distribución y terminación.
- Comprender las técnicas para la resolución de las dependencias falsas de memoria.
- Entender cómo se mejora el procesamiento de las instrucciones de carga y almacenamiento.
- Conocer cómo se consigue que un procesador superescalar tenga precisión de excepción.
- Conocer otras técnicas para mantener interrupciones precisas.

2.3. Características de los procesadores superescalares

Tras estudiar el funcionamiento de los procesadores basados en segmentaciones de un único cauce, se desprende que este tipo de procesadores posee límites en su rendimiento. En el caso ideal, es decir, ante la ausencia de detenciones, el número máximo de instrucciones que pueden emitir en cada ciclo de reloj es uno. Una alternativa para mejorar su rendimiento es reducir la duración del ciclo de reloj para aumentar el número de instrucciones ejecutadas por segundo pero esto, ciertamente, conlleva peligros. Por ejemplo, al tener menos tiempo por etapa hay que reducir el número de operaciones a realizar por el hardware y, por lo tanto, aumentar la profundidad de la segmentación para poder realizar todos los pasos que conlleva el procesamiento de una instrucción. Sin embargo, este aumento del número de etapas se traduce en un aumento de los buffers que las separan y el consiguiente aumento de las dependencias entre instrucciones. Esto último se traduce en un aumento de los riesgos y, por consiguiente, en un aumento de las detenciones y de las burbujas en la segmentación o, lo que es lo mismo, en una pérdida mayor de ciclos. Es decir, se llega a un punto de inflexión en donde la sobrecarga que introduce una segmentación muy profunda es mayor que las ventajas obtenidas por el aumento de la frecuencia, dándose el contrasentido de que un aumento de la frecuencia de reloj provoca un decremento del rendimiento global del procesador. Otro factor que no hay que olvidar es que, a medida que la segmentación se hace más profunda, intervienen más transistores, y se produce un incremento de la potencia consumida y de la cantidad de calor a disipar.

En el tema anterior se han estudiado técnicas que permiten mejorar el rendimiento de los procesadores segmentados y superar algunos de sus inconvenientes. Sin embargo, las necesidades de procesamiento que demandan las aplicaciones informáticas de hoy en día, exigen un rendimiento a este tipo de procesadores que, por su diseño arquitectónico, no son capaces de proporcionar desde hace ya algunos años.

Una de las soluciones más utilizadas para alcanzar esos niveles de rendimiento son las segmentaciones superescales que, implementadas en forma de componente electrónico, dan lugar a lo que se conoce

como procesadores superescalares. Hoy en día, la casi totalidad de los computadores se construyen en base a este tipo de procesadores. Otras alternativas actuales son los procesadores construidos conectando varios núcleos superescalares y los procesadores VLIW y EPIC, que dejan en manos del compilador el extraer el máximo provecho del paralelismo a nivel de instrucción. No hay que olvidar la existencia del procesamiento vectorial, que muchos procesadores superescalares incorporan gracias a la inclusión de unidades funcionales diseñadas para procesar instrucciones SIMD.

La diferencia más destacada de una segmentación superescalar con respecto a una segmentación clásica o escalar es que varias instrucciones avanzan simultáneamente por las etapas de la segmentación, lo que implica la existencia de varias unidades funcionales para poder efectuar esa concurrencia. Otra característica fundamental de estas segmentaciones es la ejecución de instrucciones en un orden diferente al que se especifica en el programa. En resumen, las segmentaciones superescalares se caracterizan por tres atributos: paralelismo, diversificación y dinamismo.

Paralelismo Los procesadores segmentados presentan paralelismo de máquina temporal, es decir, en un mismo instante de tiempo varias instrucciones se encuentran ejecutándose pero en diferentes etapas. Así, si un procesador cuenta con una segmentación de k etapas, estarán ejecutándose k instrucciones en el instante de tiempo t.

En un procesador superescalar se dan simultáneamente el paralelismo de máquina temporal y el espacial. El temporal es una consecuencia directa de ser procesadores segmentados. El espacial surge como consecuencia de la replicación del hardware que permite que varias instrucciones se estén procesando simultáneamente en la misma etapa. El nivel de paralelismo espacial de un procesador se especifica con el término *ancho o grado de la segmentación* y especifica el número de instrucciones que se pueden procesar simultáneamente en una misma etapa. Por lo tanto, en un procesador superescalar de ancho y profundidad de segmentación 4 y 5, respectivamente, como el de la Figura 2.1, pueden llegar a estar ejecutándose 20 instrucciones diferentes en un ciclo de reloj. Evidentemente, el coste y la complejidad del hardware aumenta ya que, por citar solo algunos de los elementos afectados, hay que aumentar los puertos de lectura/escritura del fichero de registros para que varias instrucciones puedan acceder a sus operandos simultáneamente, lo mismo sucede con las escrituras/lecturas en las cachés de datos y de instrucciones, las unidades aritméticas, los buffers de contención entre etapas, etc.

De lo descrito hasta este punto se puede desprender que un procesador superescalar no es más que la réplica de una segmentación escalar rígida. Así, valdría con replicar 4 veces una segmentación escalar para obtener un procesador superescalar de ancho 4. Pero no es tan fácil. Por un lado hay que tener en cuenta la posibilidad de la ejecución de instrucciones fuera de orden y el conocido problema de las dependencias de datos y de memoria. Además, si se reflexiona por unos instantes, se deduce que los buffers de contención entre etapas ya no pueden ser simples réplicas de los buffers de las segmentaciones escalares; ahora pasarían a ser complejos buffers multientrada que deberían permitir, por ejemplo, pasar a unas instrucciones pero no a otras. Otro elemento a incluir es una compleja red de interconexión entre las entradas y salidas de las múltiples unidades funcionales que permita la resolución de las dependencias verdaderas de datos.

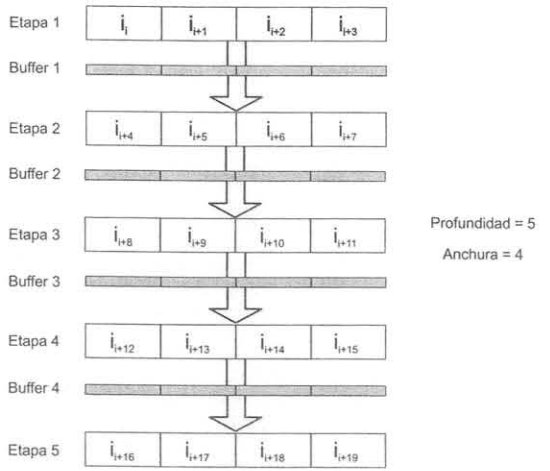


Figura 2.1: Esquema de una segmentación de profundidad 5 y anchura 4. En condiciones ideales pueden concluir su procesamiento de forma simultánea hasta 4 instrucciones por ciclo.

Diversificación Debido a que los diferentes tipos de instrucción que constituyen un repertorio de instrucciones (aritméticas, lógicas, cargas, almacenamientos, saltos, bifurcaciones, etc.) implican diferentes operaciones, disponer de un único tipo de cauce en la segmentación es ineficiente, incluso aunque se trate de una segmentación con varios cauces en paralelo. Hay instrucciones que no tienen que pasar por todas las etapas ya que solo utilizan un subconjunto. Por ejemplo, las instrucciones aritméticas en una máquina de carga/almacenamiento no necesitan acceder a la memoria. Pero, además, el aglutinamiento de operaciones de varios tipos de instrucciones en una misma etapa hace ineficiente el uso de los recursos hardware (por ejemplo, diferentes tipos de operaciones enteras y flotantes se realizan en una única unidad funcional). Por esta razón, los procesadores superescalares incluyen en la etapa de ejecución múltiples unidades funcionales diferentes e independientes, siendo habitual la existencia de varias unidades del mismo tipo.

Un ejemplo de segmentación diversificada lo constituye el procesador Alpha 21264 en el que la etapa de ejecución consta de 6 unidades funcionales de la cuales 2 realizan operaciones con enteros, 2 son para el cálculo de direcciones, 1 para sumas, divisiones y raíces cuadradas en coma flotante y 1 para multiplicaciones en coma flotante. Otro ejemplo es el procesador PowerPC 970, que dispone de 12 unidades de ejecución: 2 unidades para operaciones enteras (FXU0 y FXU1), 2 unidades de coma flotante (FPU0 y FPU1), 2 unidades de carga/almacenamiento (LSU0 y LSU1), 4 unidades especiales

para el procesamiento de instrucciones vectoriales SIMD (VP, VF, VX, VC), 1 unidad para la resolución de saltos (BRU) y 1 unidad para operar con registros de condición o de estado (CRU). El procesador AMD Opteron cuenta con 3 unidades enteras, 3 unidades de punto flotante, 3 unidades para la generación de direcciones y 2 unidades de carga/almacenamiento.

Hay que destacar que el ancho de la segmentación en las diferentes etapas puede variar. Por ejemplo, el PowerPC 604 tiene una segmentación de 6 etapas, es capaz de leer y decodificar 4 instrucciones, cuenta con 6 unidades funcionales en la etapa de ejecución (3 unidades enteras, 1 de coma flotante, 1 para procesamiento de saltos y 1 unidad de carga/almacenamiento), y puede concluir el procesamiento de hasta 6 instrucciones por ciclo. El PowerPC 970 está diseñado para leer y decodificar hasta 8 instrucciones por ciclo, emitir 8 instrucciones a las unidades funcionales y terminar hasta 5 por ciclo.

Lo anterior no implica que queden unidades funcionales ociosas. Las unidades funcionales están segmentadas pudiendo ocurrir que todas estén ocupadas por instrucciones que se hayan emitido en diferentes ciclos de reloj. Volviendo al PowerPC 970, éste dispone de 12 unidades de ejecución pero, según especificaciones del fabricante, envía a las unidades un máximo de 8 instrucciones por ciclo, lo que implica que en un ciclo de reloj pueden convivir en la fase de ejecución instrucciones que fueron emitidas en diferentes ciclos.

Dinamismo Las segmentaciones superescalares se etiquetan como dinámicas al permitir la ejecución de instrucciones fuera de orden. Las instrucciones se leen, decodifican, distribuyen y terminan en el orden secuencial en que aparecen en el programa pero se pueden emitir y finalizar en las unidades funcionales en un orden diferente al establecido por el programa. La segmentación superescalar cuenta con los mecanismos necesarios para garantizar que se obtendrán los mismos resultados, es decir, que se respeta la semántica del código fuente.

En una segmentación rígida paralela todas las instrucciones situadas en la etapa i pasan a la etapa $i+1$ en el siguiente ciclo de reloj. Todas las instrucciones entran y avanzan simultáneamente por la segmentación y terminan simultáneamente. Los registros que separan las etapas en una segmentación paralela rígida son buffers multientrada o multipuerto y, básicamente, son una réplica de los registros que separan las etapas en una segmentación escalar clásica. Tal y como se ha indicado, la problemática de este tipo de segmentaciones es la detención de todo el cauce ante una dependencia por parte de una de las instrucciones que se han emitido.

Una segmentación dinámica paralela utiliza buffers multientrada que permiten que las instrucciones entren y salgan de los buffers fuera de orden. Puede suceder que una instrucción se quede detenida en el buffer en espera de un operando, mientras que se da paso a instrucciones posteriores pero que disponen de todos sus operandos. En el momento de concluir su procesamiento puede suceder que estas instrucciones (si hay dependencias WAW) tengan que quedar retenidas en otro buffer para respetar la semántica del programa y terminar en orden.

La ventaja que aporta la ejecución fuera de orden es clara: intenta aprovechar al máximo el paralelismo que permiten las instrucciones y el que permite el hardware al disponer de múltiples unidades funcionales. Esto se traduce en:

- Unas instrucciones pueden adelantar a otras si no hay dependencias falsas, evitando ciclos de detención innecesarios.

- Una reducción de los ciclos de detención por dependencias verdaderas de datos y memoria. Se dispone de mecanismos que permiten que una instrucción se pueda emitir a una unidad funcional en cuanto sus operandos fuente están disponibles en las salidas de otras unidades funcionales.

La Figura 2.2 muestra un esquema por bloques de una segmentación superescalar. Hay dos partes claramente diferenciadas: el bloque de etapas compuesto por el *front-end* y el *back-end* en el que las instrucciones avanzan en orden y se asegura su terminación ordenada y el *núcleo de ejecución dinámica* que es el conjunto de unidades funcionales en el que se ejecutan las instrucciones fuera de orden.

Otra característica de los procesadores superescalares es su capacidad para especular, es decir, realizar predicciones sobre las instrucciones que se ejecutarán tras una instrucción de salto. En un procesador superescalar no solo las instrucciones se ejecutan fuera de orden sino que se ejecutan instrucciones que puede que no sean necesarias si el destino de salto no coincide con la especulación realizada por el procesador.

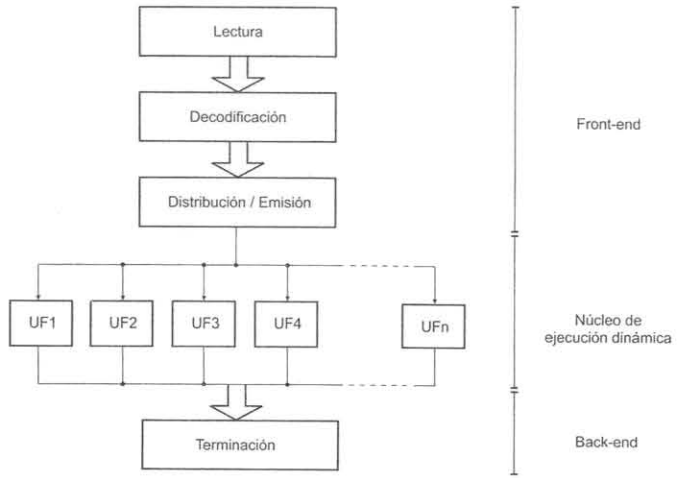


Figura 2.2: Esquema de bloques de un procesador superescalar segmentado con ejecución fuera de orden.

2.4. Arquitectura de un procesador superescalar genérico

Conocidas las características genéricas de las segmentaciones superescalares, en esta sección se presenta un sencillo modelo de este tipo de arquitectura. Este modelo servirá como guía para explicar el funcionamiento de cada una de las etapas de que consta, así como para profundizar en las técnicas que se aplican para solucionar los problemas que surgen al permitir la ejecución de las instrucciones fuera de orden. El modelo de segmentación superescalar genérica consta de 6 etapas:

- Etapa de lectura de instrucciones (IF - *Instruction Fetch*).
- Etapa de decodificación (ID - *Instruction Decoding*).
- Etapa de distribución/emisión (II - *Instruction Issue*).
- Etapa de ejecución (EX - *Execution*).
- Etapa de terminación (WR - *Write-Back Results*).
- Etapa de retirada (RI - *Retirement Instructions*).

Tomando como referencia este modelo, la Figura 2.3 muestra un ejemplo de segmentación superescalar con 7 unidades funcionales en la etapa de ejecución: 2 para operaciones con enteros, 1 para sumas/restas en coma flotante, 1 para multiplicación/división en coma flotante, 1 para instrucciones de salto, 1 para las instrucciones de almacenamiento y 1 para las instrucciones de carga desde memoria. En este ejemplo se considera que el ancho de la segmentación es 4 a lo largo de toda la segmentación, aunque en los procesadores reales no es así y el ancho varía según la etapa.

En el ejemplo de la Figura 2.3, la etapa IF es capaz de leer simultáneamente 4 instrucciones de 4 bytes desde la I-caché. A continuación, las 4 instrucciones pasan a la etapa ID en donde se decodifican en paralelo. Tras esto, las instrucciones pasan ordenadamente a la etapa II donde, según su tipo y disponibilidad de sus operandos fuente, quedan en la estación de reserva centralizada (más adelante, se estudiará este elemento clave) a la espera de ser emitidas a las unidades funcionales correspondientes. Desde este momento es cuando comienza la ejecución fuera de orden de las instrucciones, ya que puede suceder que unas instrucciones posteriores a otras en el orden del programa se ejecuten antes en las unidades funcionales debido a que dispongan de todos sus operandos fuente y su unidad funcional esté libre. Finalizada su ejecución, las instrucciones se almacenan en el buffer de reordenamiento o de terminación donde quedan a la espera de poder terminar arquitectónicamente su procesamiento. Tras salir del buffer, las instrucciones pasan por la etapa de terminación para realizar la escritura de resultados en registros, de forma que la actualización del estado de la máquina no viole la semántica del programa y se mantenga la consistencia del procesador. Para concluir su procesamiento, las instrucciones de almacenamiento quedan detenidas en el buffer de almacenamiento para proceder a la escritura ordenada de sus resultados en memoria en la etapa de retirada, es decir, para actualizar el estado de la memoria y mantener la consistencia de la memoria. La finalidad de esta etapa de retirada es permitir que los

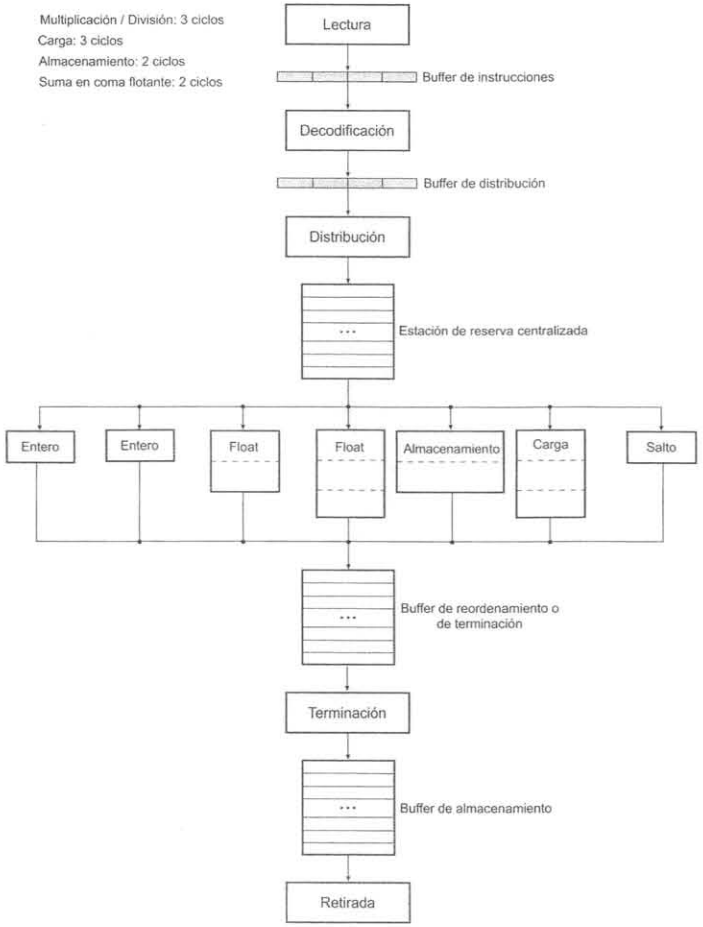


Figura 2.3: Ejemplo de segmentación superescalar genérica.

almacenamientos concluyan ordenadamente cuando no esté ocupado el bus de acceso a la caché de datos, dando mayor prioridad a las instrucciones de lectura.

Existe una gran confusión en lo que respecta a algunos de los términos utilizados en la literatura para describir los estados por los que pasa una instrucción en una segmentación superescalar. En líneas generales y en este texto se entiende que una instrucción ha sido:

- *Distribuida* (*dispatched*) cuando ha sido enviada a una estación de reserva asociada a una o varias unidades funcionales del mismo tipo.
- *Emitida* (*issued*) cuando sale de la estación de reserva hacia una unidad funcional, es decir, cuando comienza su verdadera ejecución. En algunas ocasiones, y dependiendo de la arquitectura, la distribución y emisión se realizan en la misma fase.
- *Finalizada* (*finished*) cuando abandona la unidad funcional y pasa al buffer de reordenamiento. Los resultados se encuentran almacenados temporalmente en registros no accesibles al programador, conocidos como registros de renombramiento.
- *Terminada* (*completed*) o *terminada arquitectónicamente* es cuando ha realizado la escritura de los resultados desde los registros de renombramiento a los registros arquitectónicos, ya visibles al programador. En esta fase se realiza la actualización del estado del procesador para garantizar así su consistencia.
- *Retirada* (*retired*) cuando ha realizado la escritura en memoria (actualización del estado de la memoria). Si las instrucciones no necesitan escribir en memoria (por ejemplo, las instrucciones aritmético-lógicas) se considera que la finalización de una instrucción coincide con su retirada.

Veamos algunos ejemplos para comprender un poco mejor la problemática que entraña el diseño de un procesador superescalar. En los siguientes ejemplos se va a emplear una segmentación superescalar similar a la del modelo de la Figura 2.3. Como única técnica para minimizar las detenciones causadas por las dependencias de datos se considerará el adelantamiento de resultados entre unidades funcionales. Las instrucciones leen sus operandos fuente en cuanto están disponibles en el fichero de registro o les son adelantados y se emiten cuando ya disponen de todos los operandos y la unidad está libre.

Ejemplo 1 La Figura 2.4 muestra la ejecución de un fragmento del código ensamblador del bucle DAXPY en el modelo de segmentación superescalar. El bucle DAXPY (*Double Precision A Times X Plus Y*) es parte del benchmark LINPACK e implementa la operación vectorial $Y(i)=a*X(i)+Y(i)$. En este ejemplo la longitud del vector es de 100 elementos de 8 bytes.

Siguiendo el modelo de la Figura 2.3, inicialmente se extrae de la caché de instrucciones y se decodifica un bloque de cuatro instrucciones. Debido al riesgo RAW entras las dos primeras instrucciones por el registro F2, la instrucción i2 se detiene 3 ciclos hasta que la instrucción i1 le adelanta el resultado de la carga al finalizar su etapa EX. La instrucción i3 queda detenida un ciclo debido a un riesgo estructural ya que la única unidad funcional de carga se encuentra ocupada por la instrucción i1. Una vez

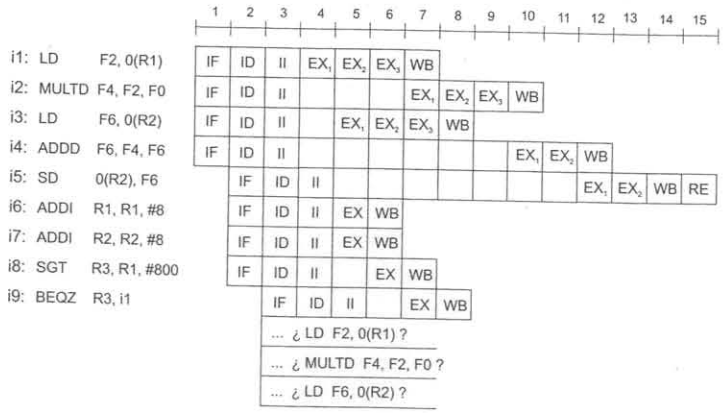


Figura 2.4: Secuencia de la ejecución del bucle DAXPY en la segmentación superescalar de 6 etapas.

liberada la primera etapa de la unidad de carga, la instrucción i3 puede comenzar su ejecución y concluye antes que la instrucción i2 ya que no viola la semántica del programa. Debido a la dependencia RAW que tiene la instrucción i4 con respecto a la i2, i4 se tiene que detener hasta que se pueda adelantar el resultado de la multiplicación. Aunque la instrucción i4 presenta una dependencia de datos verdadera con la instrucción i3, esta no constituye ningún riesgo ya que el resultado se genera a tiempo no forzando ninguna detención.

En el ciclo 2, el siguiente grupo de cuatro instrucciones se extrae de la I-caché y se decodifica. La instrucción i5 mantiene una dependencia RAW con i4 lo que la obliga a detenerse hasta que la unidad de multiplicación pueda adelantar el resultado. La instrucción de almacenamiento hace uso de la etapa de retirada para escribir en memoria y mantenerla actualizada. Las instrucciones i6 e i7 no presentan ningún tipo de riesgo verdadero y avanzan a lo largo de la segmentación sin ningún impedimento. La instrucción i8 presenta una dependencia RAW con respecto a i6 a causa del registro R1, por lo que queda detenida hasta disponer del resultado.

En el ciclo 3 se efectúa la lectura del siguiente bloque de cuatro instrucciones, de las cuales solo la i9 no es una especulación mientras que las otras tres serán resultado de la política de predicción de saltos que implemente el procesador. En este caso, la instrucción de comparación i9 mantiene una dependencia con la instrucción i8 ya que necesita el valor de R3 por lo que deberá detenerse un ciclo en espera de que le adelanten el resultado.

En el ejemplo anterior, la emisión y terminación de las instrucciones se realiza de forma desordenada. En el momento en que una instrucción dispone de sus operandos (no hay riesgos RAW) y hay unidades funcionales libres (no hay riesgos estructurales) pasa a ejecutarse. Por lo tanto, como la terminación de

las instrucciones se realiza de forma desordenada se impide un tratamiento preciso de las interrupciones al no garantizarse la consistencia del procesador como consecuencia de las escrituras desordenadas en los registros sin atender al orden del programa.

Por otra parte, la ejecución desordenada no garantiza el cumplimiento de las dependencias falsas en las sucesivas iteraciones del bucle. En este ejemplo, si los operandos se leyesen del fichero de registros en el momento de la emisión de la instrucción hacia una unidad funcional, se violaría la dependencia WAR entre las instrucciones `i5` e `i7` por culpa del registro R2. Si se continúa desarrollando el ejemplo con las sucesivas iteraciones del bucle se puede apreciar, con toda claridad, la magnitud del problema.

Ejemplo 2 En el ejemplo de la Figura 2.5 se aprecia, claramente, cómo la finalización de las instrucciones de forma desordenada puede producir la violación de dependencias WAR. Las instrucciones `i2` e `i3` presentan detenciones ya que deben esperar por la generación de sus operandos fuente. La instrucción `i4` no presenta ninguna detención y concluye su procesamiento con la etapa de retirada que es cuando se produce la escritura en memoria del contenido de `F5`.

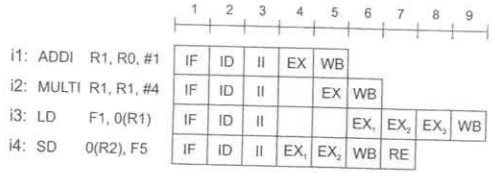


Figura 2.5: Ejecución de una secuencia de instrucciones en la segmentación superescalar de 6 etapas.

Si se analiza con detalle el procesamiento de las instrucciones, se observa que se puede haber producido la violación de una dependencia WAR ya que el almacenamiento de `i4` ha escrito en memoria antes que la instrucción de carga haya realizado la lectura. La violación de la dependencia WAR se habrá producido si el contenido de R1 y R2 coinciden.

En estos dos ejemplos se puede apreciar que una terminación ordenada de las instrucciones evita los riesgos WAW y WAR ya que se impide que una instrucción posterior a otra en la secuencia de código

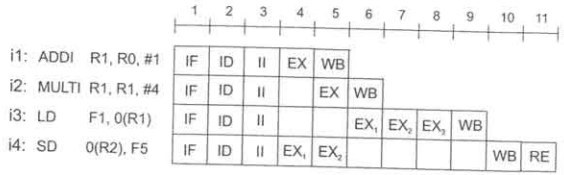


Figura 2.6: Terminación ordenada de instrucciones mediante la introducción de ciclos de detención.

escriba antes. Una solución inmediata para forzar la terminación ordenada de las instrucciones es la introducción de ciclos de detención, tal y como muestra la Figura 2.6. Sin embargo, esta solución no es muy adecuada y en segmentaciones anchas y profundas, como las superescalares, provoca grandes pérdidas de rendimiento.

Es muy importante diferenciar entre el concepto de etapa lógica y etapa física. Las etapas lógicas permiten agrupar un conjunto de operaciones que realizan una tarea completa, como por ejemplo, la lectura de instrucciones, la decodificación, la ejecución, etc. En las implementaciones reales, las etapas lógicas se encuentran segmentadas, es decir, compuestas por varias etapas físicas o segmentos, donde cada segmento consume un ciclo de reloj. En el modelo de segmentación superescalar de la Figura 2.3 se ha considerado que todas las etapas salvo la de ejecución consumen un ciclo de reloj y que la etapa de ejecución oscila de 1 a 3 ciclos según la unidad funcional de que se trate. Es muy habitual que cada etapa lógica tenga una profundidad y anchura diferentes, dependiendo del diseño del fabricante. Un ejemplo muy ilustrativo lo constituye la segmentación del procesador PowerPC 970 (Figura 2.7) cuya anchura y profundidad varían dependiendo de la etapa y del tipo de instrucción. En lo que respecta a su anchura:

- Lee hasta 8 instrucciones por ciclo de reloj.
- Decodifica en instrucciones internas hasta 5 instrucciones por ciclo.

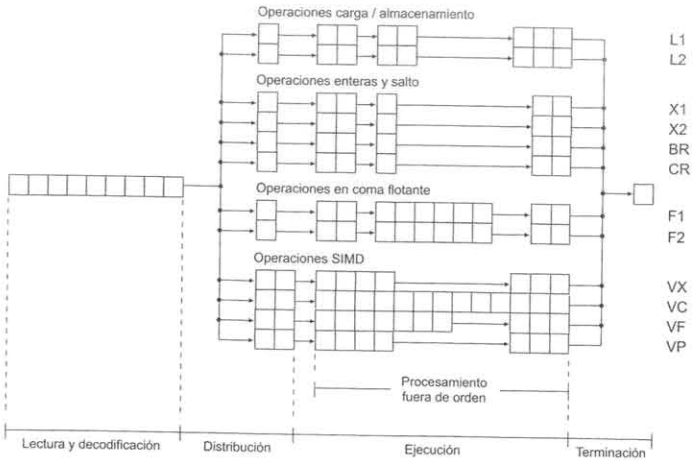


Figura 2.7: Esquema lógico de la segmentación real de un procesador PowerPC 970. Observe que las cinco etapas lógicas (*fetch*, *decode*, *dispatch*, *execute*, *complete*) se descomponen en segmentos que consumen un ciclo de reloj.

- Distribuye hasta 5 instrucciones internas por ciclo.
- Emite hasta 10 instrucciones internas por ciclo a las unidades de ejecución.
- Termina hasta 5 instrucciones internas por ciclo.

Mientras que la profundidad de la segmentación varía según el tipo de instrucción:

- 16 etapas para operaciones con enteros y saltos.
- 18 etapas para operaciones de carga/almacenamiento.
- 21 etapas para las operaciones de coma flotante.
- 19 etapas para operaciones SIMD con números enteros.
- 19 etapas para operaciones SIMD de permutación.
- 22 etapas para operaciones SIMD con números complejos.
- 25 etapas para operaciones SIMD con número en coma flotante.

lo que le permite estar procesando simultáneamente un máximo de 215 instrucciones, que se distribuirían de la siguiente forma:

- La etapa de *fetch* dispone de un buffer interno (buffer de *fetch/overflow*) que mantiene hasta 16 instrucciones.
- El buffer de instrucciones previo a la decodificación mantiene hasta 32 instrucciones.
- En cada ciclo se transfieren 5 instrucciones del buffer de instrucciones a la etapa de decodificación (3 etapas). Por lo tanto, en la etapa de decodificación se mantienen 15 instrucciones.
- Hay 4 buffers de distribución, cada uno de los cuales mantiene un grupo de distribución compuesto por 5 operaciones. Por lo tanto, en un ciclo se mantienen hasta 20 instrucciones en los buffers de distribución.
- El buffer de reordenamiento puede mantener información de hasta 20 grupos de distribución de las instrucciones que hay en fase de ejecución y de las que ya han sido terminadas. Esto hace un total de 100 instrucciones.
- El buffer de almacenamiento puede mantener hasta 32 instrucciones de almacenamiento.

Un rápido análisis de estas cifras permite apreciar la enorme complejidad que conlleva el diseño de un procesador superescalar comercial y las simplificaciones que se realizan en los ejemplos didácticos.

Después de conocer algunas de las características de la segmentación del PowerPC 970, se deduce rápidamente lo importante que es una correcta gestión de la lectura de instrucciones y su decodificación para poder alimentar a las unidades funcionales de forma constante.

Pero, además, la resolución de los tres tipos de dependencias de datos que pueden existir entre las numerosas instrucciones que están en la etapa de ejecución, ya sea a la espera de su emisión o en una unidad funcional, no es tampoco evidente. Las dependencias RAW entre instrucciones se pueden resolver mediante el adelantamiento de resultados entre unidades funcionales. Para ello es necesario incluir buses de interconexión o de reenvío entre las unidades así como el hardware adicional con la lógica de control de acceso a los buses. Sin embargo, esto provoca la existencia de dependencias estructurales al estar limitadas las interconexiones entre unidades.

Pero al ser varias las instrucciones que pueden estar esperando por uno o varios de sus operandos fuente para emitirse a las unidades funcionales, el reenvío de las salidas de las unidades funcionales no es tan trivial como en una segmentación escalar. Las estaciones de reserva, que son las encargadas de emitir las instrucciones a las unidades funcionales una vez que estén listas, deben poder saber cuándo el resultado de una unidad funcional que circula por el bus de reenvío corresponde a uno de los operandos fuente de las instrucciones que están en espera de poder iniciar su ejecución.

En lo que respecta a los riesgos WAR y WAW, la inserción de ciclos de reloj para forzar la terminación ordenada de las instrucciones y respetar la semántica del programa es conservadora y no permite aprovechar todo el paralelismo que brinda la existencia de múltiples unidades funcionales así como evitar detenciones innecesarias en las unidades. Es decir, penaliza el rendimiento. Como se ha visto en el ejemplo del PowerPC 970, las unidades funcionales tienen segmentaciones profundas lo que implica pagar un precio elevado en ciclos de reloj si hay que mantener una unidad funcional detenida para resolver unas dependencias WAW y WAR como, por ejemplo, las que se presentan en:

```
i1: ADD   R1,R1,R2
i2: MULTI R3,R1,#1
i3: ADDI  R1,R2,#1
i4: MULTI R5,R1,#8
```

Observe en el ejemplo que no sería necesario detener las instrucciones `i3` e `i4` hasta que la `i2` leyese el contenido de `R1` si se reemplazase el registro `R1` en las instrucciones `i1` e `i2` por otro no utilizado.

Como ya se ha señalado, las dependencias WAR y WAW se conocen como dependencias falsas ya que no existe una dependencia real, tal y como sucede en las RAW, donde un operando fuente es el resultado de una operación previa. Es decir, en las dependencias falsas no existe una relación productor-consumidor entre dos instrucciones como ocurre en las dependencias verdaderas. Las dependencias de datos falsas se resuelven en los procesadores superescalares recurriendo a un almacenamiento temporal en el que se realiza la escritura que entraña riesgo, es decir, la de `i3` en el ejemplo anterior. Para ello se emplea una técnica conocida como *renombramiento dinámico de registros* que consiste en utilizar un conjunto de registros ocultos al programador en los que se realizan los almacenamientos temporales. De

forma resumida, la técnica consta de dos pasos:

1. *Resolución de riesgos WAW y WAR.* Se renombran de forma única los registros arquitectónicos que son objeto de una escritura. Se resuelven así las dependencias WAW y WAR ya que se manejan registros diferentes. Utilizando el ejemplo anterior se tendría:

```
ADD   Rr1,R1,R2
MULTI Rr3,R1,#1
ADDI  Rr2,R2,#1
MULTI Rr5,R1,#8
```

2. *Mantenimiento de dependencias RAW.* Se renombran los registros arquitectónicos fuente que corresponden a una escritura previa. El objeto de este paso es mantener las dependencias RAW:

```
ADD   Rr1,R1,R2
MULTI Rr3,Rr1,#1
ADDI  Rr2,R2,#1
MULTI Rr5,Rr2,#8
```

Una vez que se han resuelto todas las dependencias en la etapa de distribución y las instrucciones se han procesado fuera de orden en la etapa de ejecución, se procede en la fase de terminación a deshacer el renombramiento y a actualizar ordenadamente los registros arquitectónicos.

En lo que respecta a las dependencias entre instrucciones de carga/almacenamiento, se presentan las mismas situaciones que con las instrucciones que operan con registros. Suponiendo que la posición de memoria es común se tiene que:

- Una carga seguida de un almacenamiento produce una dependencia RAW:

```
SD 0(R1),R5
LD R5,0(R1)
```

- Un almacenamiento tras una carga implica una dependencia WAR:

```
LD R3,0(R1)
SD 0(R1),R5
```

- Dos almacenamientos seguidos implican una dependencia WAW:

```
SD 0(R1),R3
SD 0(R1),R5
```

La solución para respetar las dependencias que pueden existir entre las instrucciones de carga/almacenamiento es su ejecución ordenada pero esto no resulta eficiente. Lo habitual es que se realice el renombramiento y la terminación ordenada de los almacenamientos ya que así se resuelven las

dependencias WAR y WAW, pero no las RAW. Al igual que con las operaciones con registros, la solución clásica para resolver las dependencias RAW es el adelantamiento de los operandos. Ahora, una instrucción de almacenamiento es la que adelanta el valor a escribir en memoria a la carga pero sin pasar por la memoria, recurriendo al reenvío interno. Otra forma de mejorar el rendimiento es adelantar la ejecución de las cargas ya que suelen existir muchas instrucciones aritméticas que dependen del dato que recupera de la memoria una instrucción de carga.

Para finalizar este rápido recorrido por algunos de los problemas que implica el procesamiento fuera de orden de varias instrucciones en un procesador superescalar, es necesario detenerse unas líneas en la problemática de la consistencia. El que se adelanten resultados o se renombren los registros para evitar dependencias WAR y WAW y aumentar así el rendimiento de las unidades funcionales no garantiza la consistencia semántica del procesador y de la memoria. Para lograr la consistencia, una vez deshecho el renombramiento, hay que realizar la escritura ordenada de los registros arquitectónicos en la etapa de terminación y de las posiciones de memoria en la etapa de retirada. Además, solo es posible terminar aquellas instrucciones que no sean el resultado de especular con una instrucción de salto, ya que si actualizan el estado de la máquina puede que, a posteriori, sea necesario reestablecerlo si no hubo lugar a su ejecución. Por otra parte, el mantenimiento de la consistencia es un aspecto fundamental para realizar un tratamiento correcto de las interrupciones. Como se verá posteriormente, el buffer de reordenamiento o terminación se convierte en la pieza fundamental para conseguir esta consistencia del procesador ya que es el lugar en donde se realiza el seguimiento de una instrucción desde que se distribuye hasta que se termina.

2.5. Lectura de instrucciones

La principal diferencia en la etapa de lectura de instrucciones entre un procesador superescalar y un escalar es el número de instrucciones que se extraen de la caché de instrucciones en cada ciclo de reloj. En un procesador escalar es una, mientras que en un superescalar el número de instrucciones está determinado por el ancho de la etapa de lectura. En el modelo de la Figura 2.3 se ha considerado que el ancho es 4, mientras que en el PowerPC 970 es 8. Todo esto implica que la caché de instrucciones tiene que estar diseñada para que en un único acceso proporcione tantas instrucciones como ancha sea la etapa de lectura. Dado que en los accesos a la caché pueden aparecer fallos de lectura que detengan el suministro de instrucciones, los procesadores disponen de mecanismos auxiliares de almacenamiento entre las etapas de lectura y decodificación. Habitualmente, esta función la realiza el buffer de instrucciones aunque cada fabricante incorpora sus propios mecanismos complementarios. El buffer de instrucciones se comporta como un amortiguador o *shock absorber* entre la etapa de decodificación y la etapa de fetch. Aunque la etapa de lectura no suministre instrucciones durante uno o varios ciclos de reloj, la etapa de decodificación puede continuar extrayendo instrucciones del buffer y continuar su tarea sin detenerse. Por esta razón, los procesadores superescalares cuentan con mayor ancho de banda para la lectura de instrucciones que para la decodificación.

La Figura 2.8 corresponde a un ejemplo de memoria caché en la que el ancho de un bloque

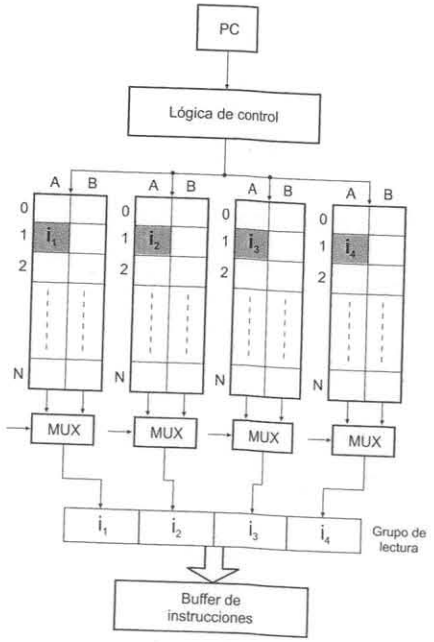


Figura 2.8: Organización de memoria caché en la que el tamaño del bloque físico corresponde al ancho de la etapa de decodificación.

corresponde exactamente con el tamaño de la etapa de lectura del modelo de procesador superescalar, es decir, 4 instrucciones. En este ejemplo, la caché es asociativa por conjuntos de dos vías, estando el bloque físicamente distribuido en cuatro arrays de memoria, de forma que cada array almacena una de las cuatro instrucciones captadas por ciclo. Una vez que se introduce una dirección en el contador de programa, la lógica de control accede al conjunto y bloque correspondiente, distribuido entre los cuatro arrays, realiza la lectura y suministra al buffer de instrucciones 4 instrucciones en un único ciclo. Al conjunto de instrucciones que se extrae simultáneamente de la I-caché se le denomina *grupo de lectura*.

Veamos las características de un procesador real como el PowerPC 970 en lo relativo a los mecanismos con que cuenta para suministrar un flujo continuo de instrucciones a la segmentación. Dispone de una I-caché de 64 Kbytes por correspondencia directa con bloques de 128 bytes a los que se puede acceder por sectores de 32 bytes. Por cada palabra (4 bytes) dispone de 5 bits adicionales

para realizar la predecodificación de instrucciones. La I-caché admite una lectura y una escritura por ciclo de reloj. Las instrucciones entran al procesador en bloques alineados de 32 bytes que constan de 8 instrucciones de 32 bits. Estos bloques o grupos de lectura quedan encolados en el buffer de instrucciones, que es quien se ocupa de alimentar la etapa de decodificación. Este buffer es capaz de almacenar hasta 32 instrucciones, de admitir la entrada de 8 instrucciones por ciclo de reloj y de emitir hasta 5. Además, para garantizar el suministro continuo de instrucciones al procesador, la etapa de fetch dispone internamente de un buffer auxiliar, denominado de fetch/overflow, que puede almacenar hasta 16 instrucciones.

Tal y como se ha indicado, el objetivo que debe cubrir la etapa de lectura de instrucciones es garantizar un suministro constante de instrucciones al procesador. No obstante, aunque el ancho de la memoria caché de instrucciones sea suficiente pueden surgir dos problemas: la falta de alineamiento de los grupos de lectura y el cambio en el flujo de instrucciones debido a las instrucciones de salto.

2.5.1. Falta de alineamiento

Un alineamiento incorrecto de un grupo de lectura implica que las instrucciones que forman el grupo superan la frontera que delimita la unidad de lectura de una caché. Esto conlleva la necesidad de dos accesos a la caché ya que el grupo de lectura ocupará dos bloques consecutivos (Figura 2.9). Esto ya reduce el ancho de banda, como mínimo, a la mitad al ser necesarios dos ciclos de reloj para suministrar las instrucciones que forman el grupo de lectura pero, además, puede provocarse un fallo de lectura si la segunda parte del grupo de lectura pertenece a un bloque que no está en la caché.

Por lo general, las máquinas se diseñan con ciertas restricciones de alineamiento para evitar problemas de lectura. Por ejemplo, la lectura de instrucciones en el PowerPC 970 se realiza de forma alineada en grupos de lectura de 32 bytes (8 instrucciones de 4 bytes). Se indica la dirección en el PC, y la I-caché del PowerPC suministra un sector de 32 bytes que, a su vez, forma parte de un bloque de 128 bytes.

Si no hay restricciones de alineación, una solución es recurrir a hardware adicional para realizar la extracción parcial de la unidad de lectura con las instrucciones desordenadas y, tras esto, proceder a la colocación correcta de las instrucciones en el grupo de lectura mediante una red de alineamiento (Figura 2.10.a) o una red de desplazamiento (Figura 2.10.b). La red de alienamiento consiste en reubicar las salidas de la caché mediante multiplexores que conducen las instrucciones leídas a su posición correcta dentro del grupo de lectura. La red de desplazamiento recurre a registros de desplazamiento para mover las instrucciones.

Una técnica utilizada para minimizar el impacto de los fallos de lectura en la caché de instrucciones se conoce como *prefetching* o *prelectura*. Consiste en disponer de una cola de *prefetch* o de prelectura de instrucciones a la que se recurre cuando sucede un fallo de lectura en la caché de instrucciones. Si las instrucciones se encuentran en la cola de prefetch, estas se introducen en la segmentación de forma similar a como si hubieran sido extraídas de la I-caché.

Por ejemplo, el PowerPC 970 dispone de una cola de prefetch de 4x128 bytes. Cuando una lectura de un sector de 32 bytes (8 instrucciones de 4 bytes) de un bloque de la I-caché produce un fallo, el PowerPC analiza el contenido de la cola de prefetch en busca de las instrucciones. En caso de que estén,

se leen y se introducen en la segmentación de forma análoga a como si hubiesen sido extraídas de la I-caché pero con una penalización de 3 ciclos. Al mismo tiempo, el sector crítico que ha producido el fallo de lectura se trae y se escribe en la I-caché para evitar futuros fallos. En caso de que el grupo de instrucciones no esté tampoco en la cola de prefetch, se lanza un fallo al siguiente nivel de caché, el L2, y se procesa con máxima prioridad. Además de estas peticiones bajo demanda a la cola de prefetch, el procesador mantiene una lógica de control que, de forma automática, extrae bloques de la I-caché con una elevada probabilidad de ser referenciados con prontitud y los almacena en la cola de prefetch, que dispone de capacidad para almacenar 4 bloques.

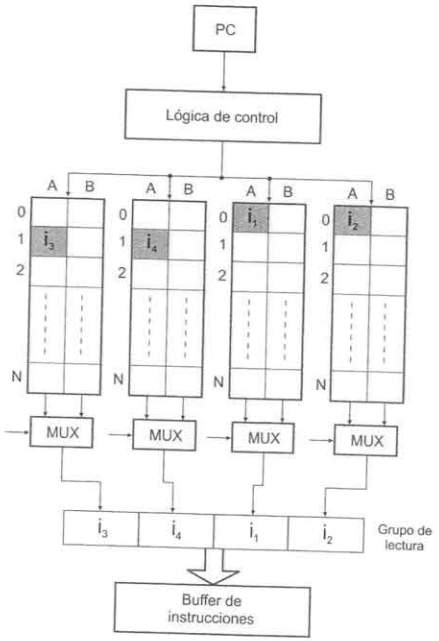


Figura 2.9: Grupo de lectura de cuatro instrucciones desalineado con respecto a la unidad de lectura, provocando la necesidad de dos accesos.

2.5.2. Rotura de la secuencialidad

El segundo problema que se presenta es la rotura de la secuencialidad en el flujo de instrucciones que componen un grupo de lectura. Puede suceder que una de las instrucciones que forme parte de un grupo de 4 instrucciones sea un salto incondicional o un salto condicional efectivo, por ejemplo, la segunda del grupo. Esto provoca que las instrucciones posteriores del grupo, en este caso la tercera y la cuarta, sean inválidas, reduciéndose el ancho de banda de lectura considerablemente.

Pero además de la reducción del ancho de banda de lectura de instrucciones, la rotura de la secuencialidad implica pagar un elevado coste en ciclos de reloj desperdiciados. En una segmentación de ancho s, cada ciclo de parada equivale a no emitir s instrucciones (o a leer s instrucciones de no operación, NOP). A esto se le denomina *coste mínimo de la oportunidad perdida* y se expresa como el

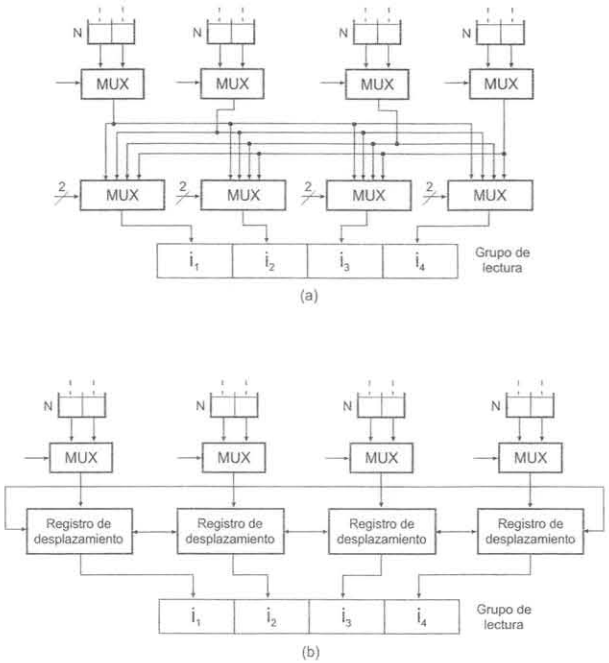


Figura 2.10: Red de alineamiento basada en multiplexores (a) y basada en registros de desplazamiento (b).

producto entre el ancho de la segmentación y el número de ciclos de penalización.

Cuando se detecta que la instrucción es un salto incondicional hay que insertar en el PC la dirección de salto, dejando inservible el procesamiento de las instrucciones posteriores al salto que ya se encontrasen en alguna etapa de la segmentación. Análogamente, si la instrucción es un salto condicional, hay que esperar a conocer si el salto es efectivo o no y, en caso afirmativo, calcular la dirección de salto, proceder a su inserción en el PC y anular las instrucciones posteriores que estuviesen en el cauce. En ambos tipos de saltos se pierden ciclos de procesamiento, pérdida que se agudiza en función del ancho y de la profundidad de la segmentación.

Detengámonos a analizar el coste de la pérdida en el procesador superescalar de la Figura 2.3 para los saltos incondicionales y los condicionales sin recurrir a ninguna técnica de mejora y suponiendo que todas las instrucciones son operaciones enteras que consumen un ciclo de reloj en la etapa de ejecución. En este modelo de procesador, las instrucciones de salto se procesan en la unidad funcional de salto que consume un ciclo de reloj, de forma que al salir de la unidad ya se conoce el valor del nuevo contador de programa. Si el ancho de la segmentación fuese uno, un cambio del PC por parte de la instrucción de salto implicaría 3 ciclos de penalización ya que habría que anular las instrucciones posteriores que hubiese en las etapas de distribución, decodificación y fetch; habría 3 ciclos sin producir ningún resultado. Pero si el ancho de la segmentación es 4, el coste de la oportunidad perdida es mucho mayor.

En el ejemplo de la Figura 2.11, cuando la instrucción de salto *i* genera el nuevo contador de programa al finalizar la etapa EX, es necesario vaciar el cauce de las instrucciones posteriores que se encuentren en las etapas EX, II, ID e IF, así como las instrucciones posteriores a *i* pero que se hayan ejecutado previamente como consecuencia del procesamiento fuera de orden. En este ejemplo, el coste mínimo de la oportunidad perdida sería de 4*3 ciclos=12 ciclos. En realidad, el coste de la pérdida es mayor ya que no se incluyen en este cómputo las pérdidas causadas al anular las instrucciones posteriores al salto que están en la etapa de ejecución y las instrucciones posteriores pero ejecutadas con anterioridad.

La técnica de salto retardado que se aplica en los procesadores escalares para rellenar huecos en la segmentación no es válida en un procesador superescalar. En un procesador escalar las instrucciones se procesan en orden y la reubicación de instrucciones a continuación de un salto tiene por finalidad rellenar las burbujas, que de otra forma, surgirían hasta detectar la efectividad del salto y la dirección de destino. Sin embargo, dado que los procesadores superescalares son capaces de procesar las instrucciones fuera de orden, no tiene mucho sentido que se coloquen instrucciones no dependientes a continuación del salto. Precisamente por esa razón, la no dependencia, el núcleo de ejecución dinámica puede decidir ejecutar esas instrucciones antes que el salto con lo que el problema permanece. En el ejemplo que se presenta en la Figura 2.11, si las instrucciones situadas a continuación del salto (de la $i + 1$ a la $i + 15$) no guardasen ninguna dependencia con el salto y hubiesen sido colocadas ahí para rellenar huecos, el procesador podría decidir ejecutar algunas de esas instrucciones antes que el salto, siendo otras las instrucciones las que pasasen a ocupar su lugar en la segmentación tras el salto y volviéndose a repetir la misma situación: la anulación del procesamiento de todas esas instrucciones al tener que procesar la instrucción $i + 40$.

La solución a este problema pasa por la detección temprana de las instrucciones que cambian el flujo de instrucciones. Lo habitual es detectar las instrucciones en la etapa de decodificación ID pero cuanto

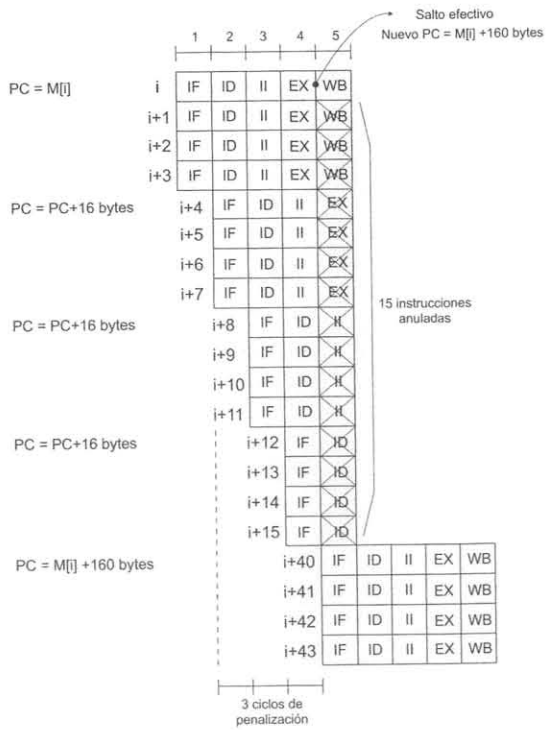


Figura 2.11: El nuevo valor del PC generado por la instrucción de salto en la etapa EX obliga a vaciar el cauce y proceder a la extracción del nuevo grupo de lectura.

antes el procesador conozca que hay una instrucción de salto, antes podrá comenzar a aplicar alguna de las técnicas destinadas a minimizar el impacto de este tipo de instrucciones. Una técnica es efectuar una detección integrada con la extracción del grupo de lectura. Esta técnica consiste en comenzar el análisis antes de que se extraiga el grupo de lectura de la I-caché. Para ello se dispone de una pre-etapa de decodificación situada entre la I-caché y el siguiente nivel de memoria, el nivel L2 de caché. Entre otras funcionalidades, en esta pre-etapa se analiza la instrucción y como resultado se le concatenan unos bits con diversas indicaciones, como por ejemplo, que se trata de un salto.

2.5.3. Tratamiento de los saltos

El principal problema que plantea el tratamiento de los saltos no es descubrir que se trata de una instrucción de salto, sino los ciclos que hay que esperar para conocer el resultado de la evaluación de la condición que determina si el salto es efectivo o no y la dirección de destino, en caso de que sea efectivo. Por lo tanto, el procesamiento de los saltos se realiza, por lo general, en una unidad funcional específica, ya que detener todo el cauce hasta saber cuál es la siguiente instrucción implicaría desperdiciar demasiados ciclos.

Lo habitual es que el tratamiento de los saltos se inicie, realmente, en la misma etapa IF, durante la lectura de la I-caché, incluso sin que haya confirmación de que hay una instrucción de salto en el grupo de lectura. Una vez que las instrucciones han sido extraídas, y se confirma en la etapa ID que una instrucción es un salto, se puede mantener el tratamiento iniciado o proceder a su anulación.

La técnica que se emplea en los procesadores superescalares para tratar las instrucciones de salto es especular, es decir, realizar predicciones. Al mismo tiempo que se están leyendo las instrucciones de la I-caché, se comienza a predecir su efectividad y su dirección de destino, sin saber realmente si se trata de un salto (lo que se confirma en la etapa ID).

Así, por ejemplo, si durante la etapa IF para una instrucción i del grupo de lectura el predictor señala un destino y que el salto es efectivo, nada más concluir la etapa IF ya se puede comenzar el procesamiento especulativo de la instrucción destino del salto: se procede a leer de la I-caché la instrucción de destino y siguientes. Mientras tanto y en paralelo, se continúa procesando el salto i, y si a la salida de la unidad funcional de salto el resultado real no coincide con la especulación, el hardware procederá a vaciar la segmentación de las instrucciones especuladas y se redirige la segmentación a la dirección correcta (la instrucción siguiente al salto, la $i + 1$). Observe que la especulación también puede realizarse con los saltos incondicionales o bifurcaciones: en este caso la especulación siempre acabará coincidiendo con el resultado del salto.

Volviendo a recurrir al procesador PowerPC 970 como ejemplo, éste emplea una técnica de predecodificación que añade 5 bits de información a cada instrucción que se almacena en la I-caché y en la cola de *prefetch*. De esta forma, la longitud de la instrucción pasa a ser de 37 bits. Una vez extraído el grupo de lectura, el PowerPC 970 es capaz de analizar hasta 8 instrucciones por ciclo en busca de saltos y realizar la predicción de 2 de ellos, pudiendo mantener en el cauce hasta 16 secuencias de código especuladas. Todos los saltos condicionales son especulados en el PowerPC 970, mientras que no sucede lo mismo con los saltos incondicionales. Dependiendo del tipo de salto condicional, algunos de ellos son estáticamente predichos como efectivos mientras que otros tienen que utilizar los mecanismos de predicción dinámica de que consta el procesador. El compilador es el encargado de realizar la predicción estática cuando puede conocer de antemano el resultado del salto condicional.

2.5.4. Estrategias de predicción dinámica

Tal y como ya se ha señalado al abordar el estudio de los procesadores segmentados, las técnicas que se emplean para predecir el comportamiento de las instrucciones de salto se dividen en dos grandes

grupos: estáticas y dinámicas. A diferencia de las técnicas estáticas, las técnicas dinámicas se basan en el hecho de que un salto es una instrucción cuya ejecución se repite con cierta frecuencia, lo que permite derivar un patrón de comportamiento futuro basado en el análisis de su comportamiento pasado. Gracias a los numerosos estudios realizados, se sabe que las técnicas de predicción estática logran tasas de predicción de saltos condicionales de entre un 70 % y 80 %. Sin embargo, debido al flujo de instrucciones tan elevado y a la complejidad de las segmentaciones superescalares, la utilización de técnicas que minimicen todo lo posible los riesgos que implican las instrucciones de salto es fundamental para extraer el máximo rendimiento posible de estos procesadores. Las técnicas de predicción dinámica obtienen tasas de acierto que oscilan entre el 80 % y el 95 %, de ahí la importancia que tienen para aumentar el rendimiento de los procesadores superescalares. El inconveniente que presentan las técnicas de predicción dinámica es el incremento en el coste económico del procesador debido a que la complejidad del hardware aumenta, lo que implica un incremento en el tamaño del área del chip y, por tanto, en toda la ingeniería asociada.

Para realizar una especulación completa de una instrucción de salto, es necesario predecir los dos componentes que produce su procesamiento: la dirección de destino y el resultado (efectivo, no efectivo). A continuación, se presentan algunas de las técnicas de predicción dinámica más utilizadas. Inicialmente, se describe una técnica de predicción de la dirección de destino que lleva implícita la predicción del resultado del salto. Tras esto, se generaliza la técnica de predicción de destino para poder utilizarse con otras técnicas de predicción del resultado que se irán introduciendo: el predictor de Smith, el predictor de dos niveles basado en el historial local, el predictor de dos niveles basado en el historial global y el predictor *gshare*.

2.5.4.1. Predicción de la dirección de destino de salto mediante BTAC

Conocer cuanto antes si una instrucción es un salto efectivo y cuál es su dirección de destino más probable es fundamental para poder reducir la penalización por salto mediante la ejecución especulativa de instrucciones. Una técnica sencilla para predecir la dirección de destino es recurrir a una BTAC (*Branch Target Address Cache*). Un BTAC es una pequeña memoria caché asociativa en la que se almacenan las direcciones de las instrucciones de saltos efectivos ya ejecutados o BIAs (*Branch Instruction Address*) y las direcciones de destino de esos saltos o BTAs (*Branch Target Address*). La Figura 2.12 muestra un esquema de la estructura de la BTAC.

El funcionamiento de la BTAC es sencillo. El acceso a la BTAC se realiza en paralelo con el acceso a la I-caché utilizando el valor del contador de programa. Si ninguna de las direcciones de las instrucciones que forman el grupo de lectura coincide con alguna de las BIAs que hay en la BTAC es debido a que no hay ninguna instrucción de salto efectivo o se trata de un salto efectivo que nunca antes había sido ejecutado. La Figura 2.13 ilustra de forma esquemática el funcionamiento de la BTAC.

Si la BTAC no devuelve ninguna dirección de destino pero, posteriormente, en la etapa ID se verifica que en el grupo de lectura hay una instrucción de salto, ¿cómo se debe proceder? Si la BTAC no devuelve ningún valor lo habitual es aplicar, por defecto, la técnica de predecir como-no-efectivo; se procede normalmente con la ejecución de las instrucciones que siguen al salto y se deshace su procesamiento en

caso de que el salto sea finalmente efectivo. En este caso, las penalizaciones serían:

- 0 ciclos de detención si el salto finalmente no es efectivo. Al aplicarse la predicción como-no-efectivo el procesamiento se realiza normalmente.
- (Ciclos_ID + Ciclos_II + Ciclos_EX_salto) de detención si el salto es efectivo, suponiendo que el salto se resuelve en la etapa EX, y el vaciado del cauce de las instrucciones que seguían al salto. En este caso hay que actualizar la BTAC con la dirección de la instrucción de salto y el destino real del salto. No tiene sentido almacenar en la tabla los saltos no efectivos ya que aunque se almacenen, su resultado real siempre coincidiría con la predicción por defecto, es decir, la ejecución de la siguiente instrucción.

Observe que si como predicción por defecto se opta por la técnica predecir-como-efectivo, no se obtiene ninguna mejora ya que siempre hay que esperar a la resolución real del salto para poder efectuarlo.

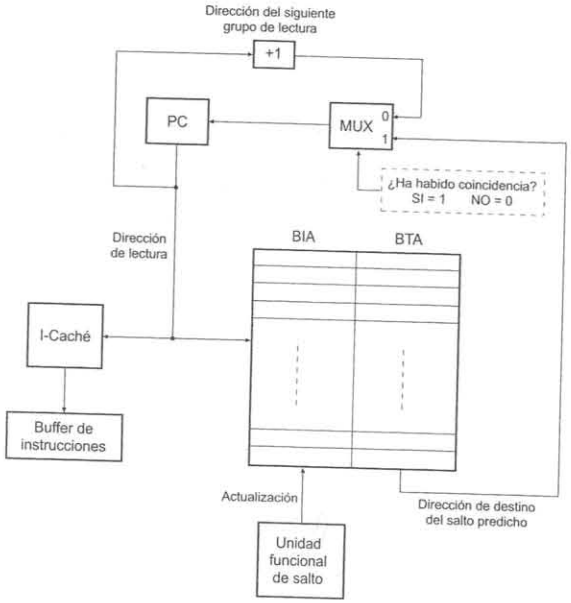


Figura 2.12: Esquema de una BTAC.

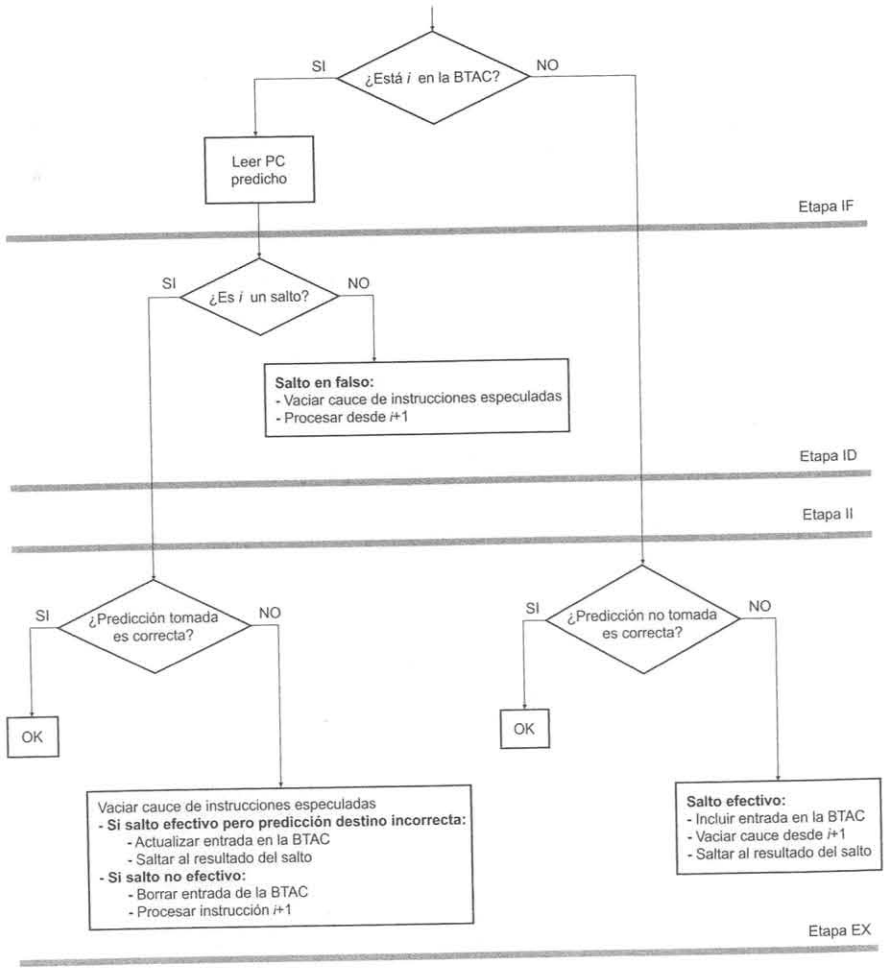


Figura 2.13: Diagrama con las posibles situaciones que se pueden presentar al realizar predicción mediante la BTAC.

Si la dirección de la instrucción que se busca en la BTAC coincide con una de las BIAs lo más probable (aunque puede haber falsos positivos, como se verá) es que se trate de un salto efectivo que ha sido ejecutado con anterioridad (de lo contrario, no estaría en la tabla) y se procede a extraer el valor de BTA que tenga asociado, es decir, la dirección de destino asociada a la última vez que fue ejecutada esa instrucción de salto.

Una vez que se dispone de la predicción de la dirección de destino, esta dirección pasará a utilizarse como nuevo contador de programa, se leerá de la I-caché y comenzará la ejecución especulativa de esa instrucción y de las siguientes. Mientras tanto, la instrucción de salto debe finalizar su ejecución con el fin de validar que la especulación realizada es correcta. Se pueden dar dos situaciones al conocer el resultado del salto:

- La predicción realizada es incorrecta debido a que se trataba de un salto no efectivo o a que la dirección de destino especulada no coincidía con la real. En ambos casos hay que anular las instrucciones especuladas que ya estaban procesándose en el cauce. Hay que eliminar de la BTAC la instrucción de salto si se trataba de un salto no efectivo o actualizarla con la dirección de destino correcta si el salto fue efectivo. Considerando que en la etapa EX ya se conoce el resultado del salto, la penalización es de (ciclos_ID+ciclos_II+ciclos_EX_salto) ciclos.
- Si la predicción fue correcta se continúa con el procesamiento de instrucciones. No es necesario modificar la tabla.

Un problema que puede surgir es el de los *falsos positivos*, es decir, la BTAC devuelve una dirección de destino pero en el grupo de lectura no hay realmente ningún salto (se descubre en la etapa ID). Esto se conoce como *saltos fantasmas* o *saltos en falso*. El tratamiento correcto para un salto fantasma es desestimar la predicción de la BTAC una vez que se confirma que se trata de un falso positivo. El coste es incurrir en una pequeña penalización (ciclos_ID) ya que hay que anular el procesamiento del salto fantasma. En conclusión, cuanto antes se detecte que se trata de una instrucción de salto, las penalizaciones serán menores en caso de error en la predicción.

Los falsos positivos son debidos a que el campo BIA de la BTAC no almacena una dirección completa, sino una parte de ella. Esto puede provocar que a direcciones distintas se les asocie la misma BIA dado que el subconjunto de bits que se utiliza para el BIA sea similar en ese grupo de direcciones. El motivo de esta resolución parcial en la búsqueda es reducir el tamaño de la BTAC a cambio de admitir las penalizaciones producidas por los falsos positivos. Si la tasa de falsos positivos es suficientemente baja, es admisible pagar ese coste dado que no ralentiza mucho el procesador. Pero si la tasa es alta, puede no ser admisible y la solución es que la longitud del campo BIA se incremente hasta alcanzar una tasa de saltos fantasma tolerable. Si la longitud del BIA llega a igualarse a la longitud de las direcciones, el problema de los falsos positivos deja de existir.

Otra técnica para predecir la dirección de salto es utilizar una BTIB (*Branch Target Instruction Buffer*). Su estructura es muy similar a la de la BTAC pero se almacena la instrucción de destino (BTI - *Branch Target Instruction*) y algunas posteriores (BTI+1, BTI+2, etc.) en lugar de solo la dirección

de salto efectivo BTA (Figura 2.14). Así, la BTIB entrega una secuencia de instrucciones al buffer de instrucciones, de forma análoga a como si se hubiesen extraído de la I-caché, y en el PC se coloca la dirección que corresponde a la siguiente instrucción que hay después de la última BTI que forma el paquete.

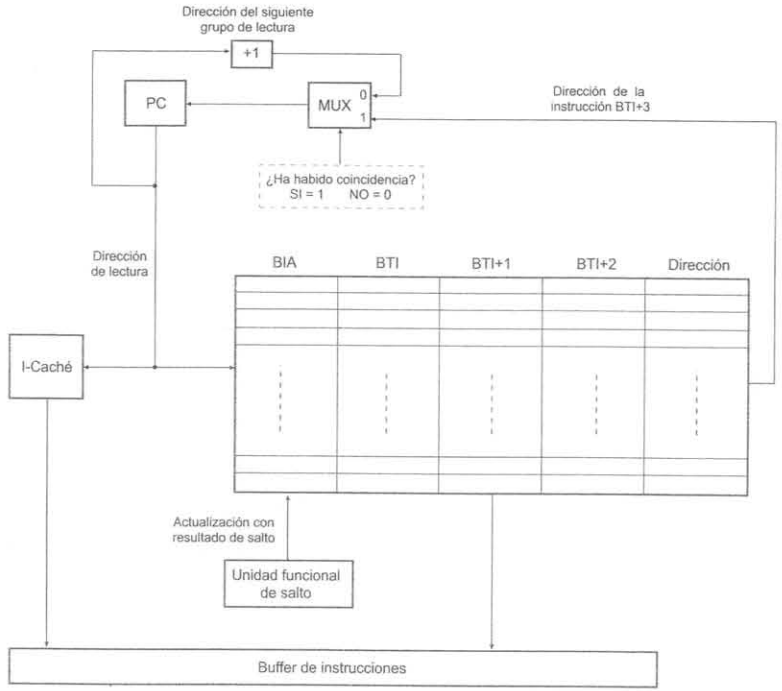


Figura 2.14: Esquema de una BTIB en la que se entregan al buffer de instrucciones paquetes de 3 instrucciones. Esto obliga a modificar el PC para que apunte a la instrucción que sigue a la última que forma el paquete, en este caso, el PC debe apuntar a la dirección de la instrucción BTI+3.

2.5.4.2. Predicción de la dirección de destino de salto mediante BTB con historial de salto

La principal diferencia de una BTB (*Branch Target Buffer*) con respecto a la técnica anterior de predicción es que junto con la dirección de destino predicha, BTA, se almacena un conjunto de bits que representan el historial de la efectividad del salto (BH - *Branch History*). De hecho, la predicción mediante BTAC se puede considerar un caso particular de la predicción mediante BTB considerando un historial de salto de 1 bit, de forma que si la dirección de la instrucción está en la BTAC el salto se predice siempre como efectivo y si no lo está se predice como no efectivo. La Figura 2.15 corresponde a un esquema en el que se aprecia el funcionamiento de una BTB.

El funcionamiento es muy similar al de la técnica anterior. Al mismo tiempo que se extrae el grupo

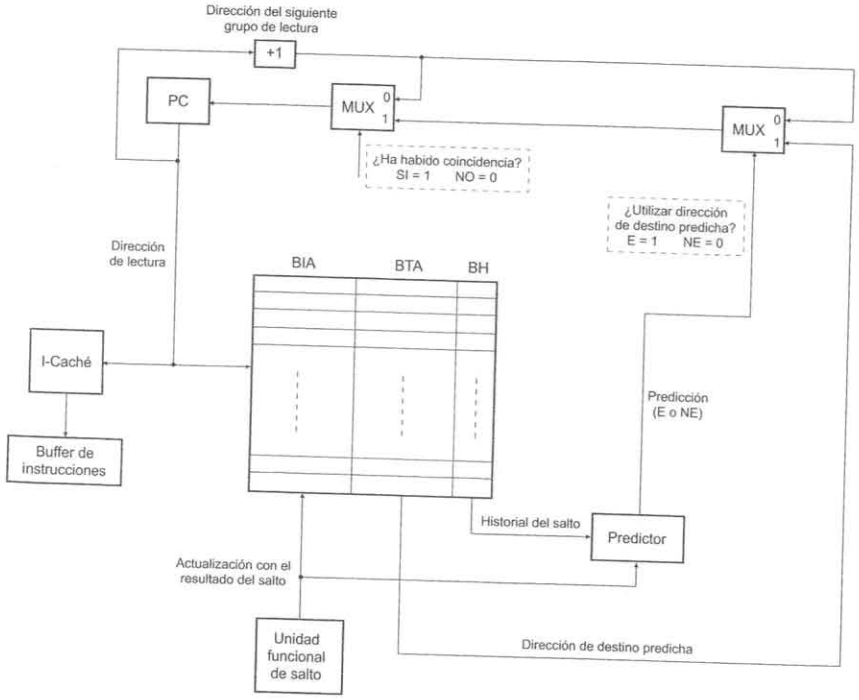


Figura 2.15: Esquema de una BTB con historial de salto.

de lectura de la I-caché, se accede a la BTB en busca de alguna de las instrucciones del grupo de lectura. Si hay un acierto (y no se trata de un salto fantasma) se analizan los bits del historial del salto y se decide si la dirección de destino predicha ha de ser utilizada o no (en el siguiente apartado se explica el predictor de Smith como forma de decidir si utilizar o no la dirección predicha). Al igual que en el caso anterior, la instrucción de salto se continúa procesando para validar el resultado de la especulación. Las cuatro situaciones que se pueden dar con sus respectivas acciones son:

- Se predice como efectivo y no hay error en la predicción (tanto en el resultado como en la dirección destino): No sucede nada, no hay penalización de ninguna clase, se actualiza el historial del salto.
- Se predice como efectivo pero hay algún error en la predicción (en el resultado, en la dirección, o en ambos): Se vacía el cauce de las instrucciones especuladas, se actualiza el historial de salto con el resultado, se actualiza la entrada de la BTB con la dirección de destino y se comienza a procesar la instrucción indicada por el resultado del salto.
- Se predice como no efectivo y el salto no lo es: No sucede nada, no hay penalización de ninguna clase, solo hay que actualizar el historial del salto.
- Se predice como no efectivo pero el salto sí lo es: Se vacía el cauce con las instrucciones especuladas (en este caso, las siguientes al salto), se salta a la dirección de destino obtenida, se actualiza el historial del salto, y se actualiza la entrada de la BTB con la dirección de destino del salto.

Si ninguna de las instrucciones del grupo de lectura proporciona una coincidencia en la BTB, se predice como no efectivo. Y pueden plantearse dos situaciones:

- El salto no es efectivo: No sucede nada en el cauce.
- El salto es efectivo: Se limpian del cauce las instrucciones siguientes al salto (las especuladas) y se salta a la dirección de destino obtenida, se incluye una entrada en la BTB con los siguientes datos: dirección de la instrucción de salto, dirección de destino e historial del salto.

En caso de que la instrucción no sea un salto, no sucede nada. El aplicar predecir como no efectivo implica la ejecución de la siguiente instrucción al salto por defecto, es decir, una situación análoga a como si no se tratase de una instrucción de salto.

Otro aspecto que hay que resolver en las dos técnicas anteriores es la eliminación de entradas en la tabla debido a fallos de capacidad o de conflicto. Como se trata de una especie de memoria caché, la solución depende de la organización de la tabla:

- Correspondencia directa. Se sustituye la entrada.
- Asociatividad. Se opta por descartar la entrada que tenga menos potencial para mejorar el rendimiento. Dos opciones posibles son la aplicación de un algoritmo LRU (*Least Recently Used*), mediante el que se elimina la entrada que lleva sin utilizarse más tiempo, o eliminar la entrada con mayor posibilidad de ser no efectiva según su historial de salto.

2.5.4.3. Predictor de Smith o predictor bimodal

Como se ha detallado en el algoritmo anterior, cada entrada de la BTB contiene un campo, el BH, donde se almacena un pequeño conjunto de bits que refleja el historial de la instrucción de salto. El historial de salto no es más que un registro con los resultados del salto en las últimas veces que fue invocado. Por lo tanto, a mayor número de bits, mayor longitud del registro temporal, aunque esto se ha demostrado que no siempre conduce a mejores predicciones.

El *predictor de Smith*, o predictor bimodal, es el algoritmo de predicción dinámica del resultado de salto más sencillo. Se basa en asociar un contador de saturación de k bits a cada salto de forma que el bit más significativo del contador indica la predicción para el salto: si el bit es 0, el salto se predice como no efectivo (en inglés se expresa como *Not Taken* o NT); si el bit es 1, el salto se predice como efectivo (*Taken* o T). Los k bits que forman el contador de saturación constituyen el historial de salto y es la información que se almacena en el campo BH de la BTB. Estos bits junto con el resultado actual del salto se utilizan para realizar la predicción de lo que sucederá la vez siguiente que ese salto sea ejecutado.

Un contador de saturación es un contador clásico con la diferencia de que cuando está en su valor máximo y se incrementa no vuelve a 0, sino que se queda igual; a su vez cuando alcanza el valor de 0 y se decrementa, el valor continúa siendo 0. Cada vez que un salto es invocado, se extrae el historial del salto (el valor del contador) de la BTB, se mira la predicción (el bit más significativo) y se aplica. Tras esto se procede a actualizar el contador con el resultado real del salto para dejar preparada la próxima predicción:

- Si el salto fue efectivo, el contador se incrementa.
- Si el salto no fue efectivo, el contador se decrementa.

Esto significa que para valores altos del contador, el salto se predecirá como efectivo, mientras que para valores bajos será predicho como no efectivo.

Aunque se puede utilizar un contador de 1 bit (Smith_1), el más utilizado es el de 2 bits (Smith_2). El contador de 2 bits presenta cuatro posibles estados que se suelen denominar:

- SN (*Strongly Not Taken*): 00
- WN (*Weakly Not Taken*): 01
- WT (*Weakly Taken*): 10
- ST (*Strongly Taken*): 11

Los estados SN y WN predicen que el salto no será efectivo, mientras que WT y ST predicen que sí lo será. La Figura 2.16 muestra la representación en forma de autómata finito de un predictor Smith_1 y de un predictor Smith_2. Las transiciones representan el resultado real del salto mientras que los estados del autómata representan el historial del salto.

Se pueden utilizar contadores de 3 bits aunque los estudios han demostrado que la mejora que se obtiene en la predicción es muy pequeña con respecto al empleo de 2 bits. Sin embargo, la mejora que proporciona un Smith_2 es bastante superior con respecto a un Smith_1 .

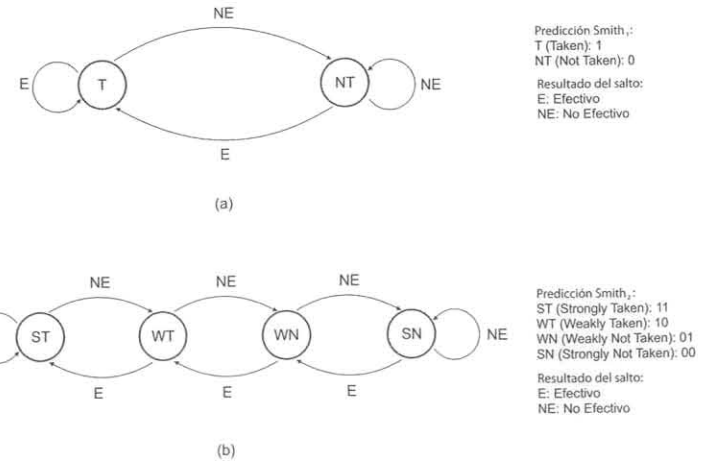


Figura 2.16: Predictores de Smith de 1 bit (a) y 2 bits de historial (b).

2.5.4.4. Predictor de dos niveles basado en el historial global

Debido a los elevados requisitos de los procesadores superescalares, los predictores de un nivel, como el bimodal, no proporcionan una tasa de acierto en las predicciones suficientemente alta. Por esta razón se han desarrollado predictores más complejos, como son los basados en dos niveles de historial. Estos predictores mantienen en un primer nivel de información un historial de los últimos saltos ejecutados (*historial global*) o de las últimas ejecuciones de un salto concreto (*historial local*). En un segundo nivel, la información del primer nivel en combinación con la dirección de la instrucción de salto se utiliza para acceder a una tabla que almacena contadores de saturación que son los que determinan la predicción.

El predictor de dos niveles de historial global se basa en un registro en el que se almacena el resultado de los saltos más recientes. El historial de los últimos h saltos, que no es más que una secuencia de bits, se almacena en un registro de desplazamiento de h bits denominado registro del historial de saltos (BHR - *Branch History Register*). Cada vez que un salto se ejecuta se introduce su resultado por el extremo derecho del registro, se desplaza el contenido una posición y se expulsa el resultado más antiguo por el extremo izquierdo. Si el salto fue efectivo se introduce un 1 y, en caso contrario, un 0. Por ejemplo, si el BHR es de 8 bits y de los últimos 8 saltos los cuatro más antiguos fueron efectivos y los otros cuatro no, el contenido del registro sería BHR=11110000; en el supuesto de que el próximo salto fuese efectivo el contenido del BHR pasaría a ser 11100001.

Para poder conocer la predicción para un salto, los h bits del BHR se concatenan con un subconjunto

de m bits obtenido mediante la aplicación de una función *hash* a la dirección de la instrucción de salto. Un ejemplo básico de *hashing* para reducir la dirección de la instrucción a m bits puede ser el quedarse con los m bits menos significativos de la dirección.

Esta combinación de $h + m$ bits se utiliza para acceder a una tabla de historial de patrones (PHT - *Pattern History Table*). Esta tabla almacena en cada una de sus 2^{h+m} entradas un contador de saturación de 2 bits. Al igual que en el predictor de Smith, el bit más significativo del contador representa la predicción del salto: 1 indica efectuar el salto y 0 lo contrario. La Figura 2.17 representa un esquema de un predictor de dos niveles basado en historial global con $h = 5$ y $m = 3$ siendo la longitud de la dirección de la instrucción de 30 bits. El número total de entradas de la PHT es $2^{5+3} = 2^8 = 256$.

Una vez que se ha evaluado el salto y se conoce su resultado hay que proceder a la actualización de los dos componentes del predictor: se incrementa/decrementa el contador de la PHT y se actualiza el historial del BHR, quedando así todo dispuesto para la próxima predicción.

Es fundamental la aplicación de la función *hash* para reducir la longitud de la dirección de la instrucción a m bits. En las arquitecturas actuales, en las que se manejan contadores de programa de 30 y 62 bits resulta impensable el concatenar todos los bits de la dirección de la instrucción con el contenido del BHR ya que se obtendría una PHT muy grande. Por ejemplo, si la longitud del BHR fuese de 5 bits y la longitud de la dirección de 30 bits, la tabla tendría 2^{35} entradas de 2 bits que equivalen a 8.192 Mbytes lo que resulta, hoy en día, inviable.

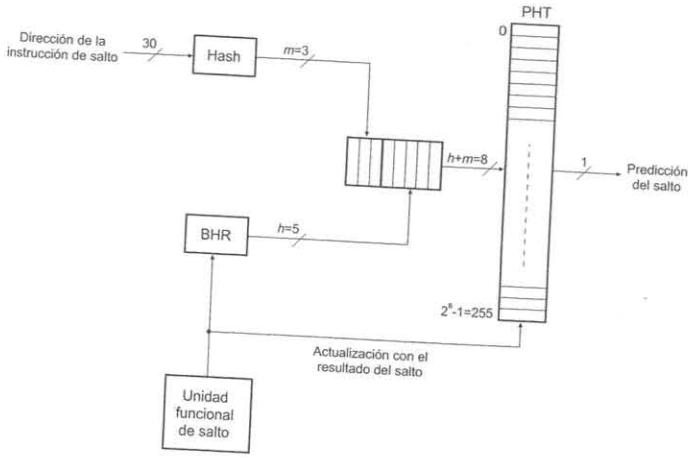


Figura 2.17: Esquema de un predictor de dos niveles basado en el historial global.

2.5.4.5. Predictor de dos niveles basado en el historial local

El predictor de dos niveles basado en historial local es muy similar al anterior. La única diferencia es que, en lugar de un único registro con el historial de los últimos saltos, ahora se utiliza una tabla en la que se almacena el historial particular de cada salto. Es decir, se reemplaza el registro BHR por una tabla de historial de saltos (BHT - *Branch History Table*) compuesta por varios BHR.

Para obtener la predicción de un salto, en primer lugar hay que recuperar el historial del salto. El acceso a la BHT se realiza mediante un *hashing* de la dirección de la instrucción de salto que los reduce a k bits ya que el número de entradas de la BHT es 2^k. Una vez que se dispone de los h bits del historial, estos se concatenan con los m bits obtenidos mediante otro *hashing* de la dirección de la instrucción de salto. Con esta secuencia de $h + m$ bits se accede a la PHT para recuperar el estado del contador de saturación de 2 bits. Al igual que en el predictor de historial global, la función *hash* puede ser muy básica y consistir en quedarse con los m o k bits menos significativos de la dirección, aunque se puede recurrir a funciones más complejas. La Figura 2.18 presenta un esquema genérico de un predictor de dos niveles

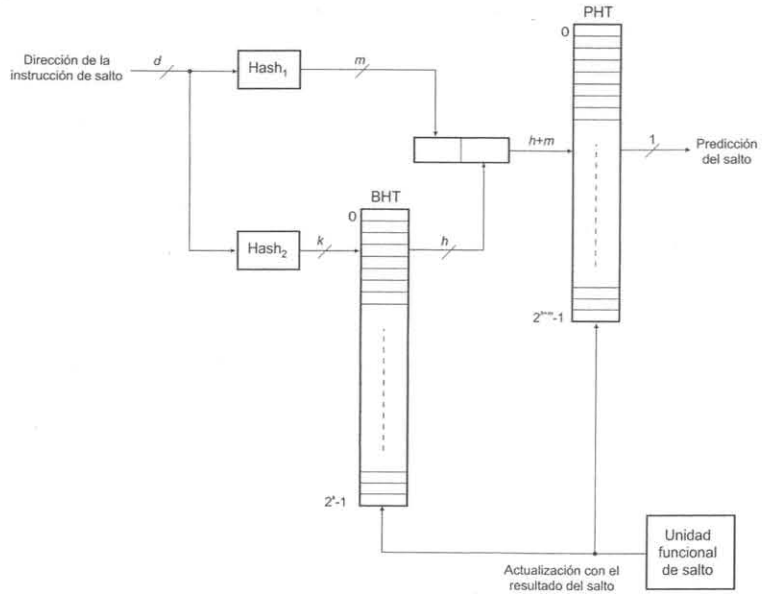


Figura 2.18: Esquema de un predictor de dos niveles basado en el historial local.

basado en el historial local en donde una función *hash* reduce los d bits de longitud de la dirección de la instrucción a k bits para acceder a la BHT y otra función *hash* los reduce a m bits.

Una vez que el salto se ha evaluado y se conoce su resultado es necesario proceder a actualizar los dos componentes del predictor: la BHT y la PHT. Para ello, primero se accede a la PHT para actualizar el contador de saturación con el resultado (se suma 1 si fue efectivo, 0 en caso contrario). Tras ello, se accede al historial de la BHT para introducir el resultado del salto (1 si fue efectivo, 0 si no lo ha sido), desplazando su contenido. De esta forma, los dos niveles del predictor quedan actualizados para realizar la predicción del siguiente salto.

2.5.4.6. Predictor de dos niveles de índice compartido *gshare*

Una variante del predictor de dos niveles de historial global es el predictor *gshare*. En este predictor, en lugar de concatenar los m bits obtenidos de la función *hash* con los h recuperados del historial global, se realiza una XOR entre los h bits superiores de los m. Los h bits obtenidos de la XOR se concatenan con los $m - h$ bits restantes para poder acceder a la PHT. La Figura 2.19 muestra un esquema genérico de un predictor *gshare*.

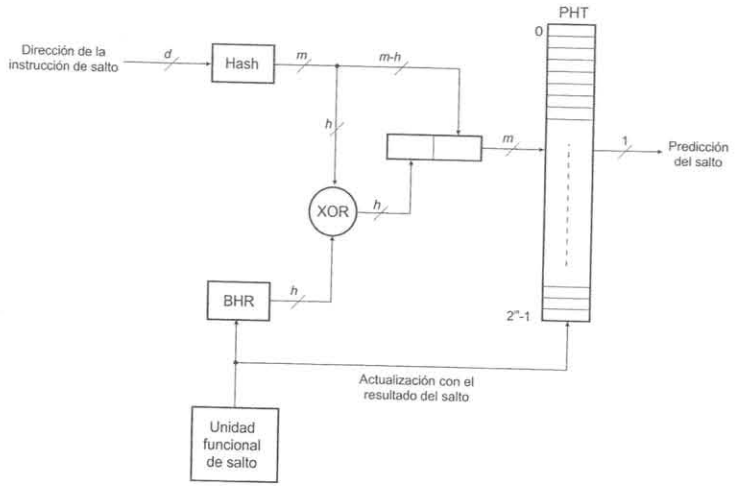


Figura 2.19: Esquema de un predictor *gshare*.

2.5.4.7. Predictores híbridos

Los procesadores superescalares actuales no realizan las predicciones utilizando un único método de predicción sino que recurren a dos predictores que generan un resultado y un selector se ocupa de decidir cuál de las dos predicciones hay que utilizar. Este mecanismo de predicción se denomina *predicción híbrida*. El PowerPC 970 emplea un predictor híbrido compuesto por un predictor bimodal, un predictor *gshare* y un selector. El predictor bimodal cuenta con una PHT de 16K entradas de 1 bit que se indexa con la dirección de la instrucción de salto. El predictor *gshare* utiliza un BHR de 11 bits de historial y una PHT de 16K entradas de 1 bit. Por último, el selector es una PHT de 16K entradas de 1 bit que se indexa con la dirección de la instrucción de salto y determina cuál de las dos predicciones hay que utilizar; en función del éxito de la predicción se realiza la actualización de la correspondiente entrada de la tabla del selector.

2.5.5. Pila de dirección de retorno

Validar una predicción de una instrucción de salto implica no solo comprobar que el resultado del salto coincide con la predicción sino que también es necesario saber si la dirección de destino predicha coincide con la dirección de destino real. La mayor parte de las veces, la predicción del destino coincide con el resultado real del salto ya que los saltos, normalmente, siempre tienen la misma dirección de destino.

Sin embargo, los repertorios de instrucciones suelen disponer de una instrucción especial de salto que no puede predecirse adecuadamente ya que cada vez que se la invoca puede saltar a una dirección completamente diferente. Para esta instrucción la BTB generaría predicciones de la dirección de destino con una elevada tasa de fallos. Esta instrucción es el *retorno de subrutina*. Tal y como muestra la Figura 2.20, las invocaciones a una subrutina se pueden realizar desde diferentes lugares de un programa. Por ello, la instrucción de salto que hay al final de la subrutina para devolver el control, no tiene una dirección de destino fija, sino que varía según desde donde se realice la invocación. Esto provoca que esa instrucción de salto nunca obtenga predicciones correctas de la BTB ya que corresponderán a la dirección de retorno de la invocación previa (debido a la actualización de la BTB al detectarse un fallo).

Para manejar estas situaciones, la mayor parte de los procesadores actuales disponen de una pila de direcciones de retorno (RAS - *Return Address Stack*) o buffer de pila de retornos (RSB - *Return Stack Buffer*). Cuando se invoca a una subrutina mediante una instrucción de salto se efectúan tres acciones:

- Se accede a la BTB para obtener la predicción de la dirección de destino.
- Se especula el resultado del salto.
- Se almacena en la RAS la dirección de la instrucción siguiente al salto.

Una vez procesadas las instrucciones que componen la subrutina, se invoca una instrucción de retorno de subrutina. Como con cualquier otra instrucción se accede a la BTB para obtener la predicción de la dirección de destino pero, en el momento en que se detecta que la instrucción es un retorno, se accede

también a la RAS para obtener la dirección correcta de retorno y se desestima la predicción de la BTB. Las instrucciones que se hayan procesado como consecuencia de la especulación incorrecta dada por la BTB serán anuladas. El principal problema que presenta la RAS es su desbordamiento cuando se manejan subrutinas con muchos anidamientos y la pila no ha sido dimensionada adecuadamente.

Como ejemplo, el procesador PowerPC 970 dispone de una RAS de 8 entradas denominada *link stack* o *call-return stack*. Durante la extracción de la I-caché, el procesador evalúa si se trata de una instrucción de retorno mediante el análisis del valor de un campo de 2 bits de la instrucción (campo HB - *Hint Bits*); si se trata de un retorno se extrae la dirección del *link stack* sin necesidad de acceder al BTB. Es el compilador, cuando genera el código objeto, quien determina mediante el campo HB si se trata de un retorno de subrutina o no. Un ejemplo de procesador CISC que recurre a una RAS para optimizar los retornos de subrutina es el AMD Opteron que incorpora una RAS de 12 entradas

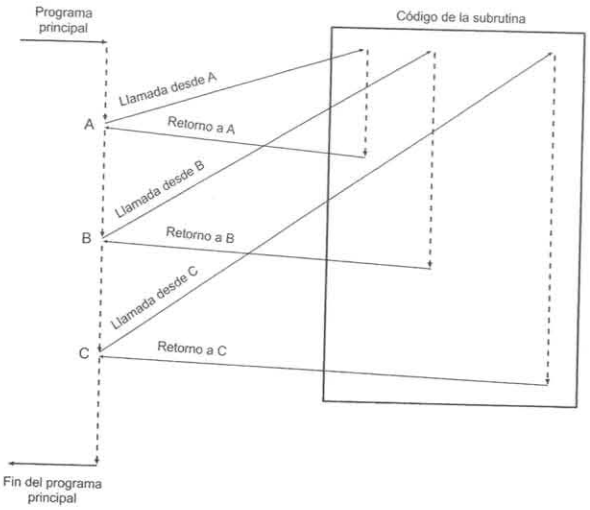


Figura 2.20: El salto que produce la llamada a la subrutina siempre se dirige a la misma dirección de memoria. Por el contrario, el salto que realiza el retorno de la subrutina puede dirigirse a distintas direcciones de memoria dependiendo de la instrucción que la invocó.

2.5.6. Tratamiento de los errores en la predicción de los saltos

Uno de los problemas que plantea la ejecución especulativa de instrucciones es que puede ser incorrecta, es decir, que el resultado de la predicción del salto no coincida con el resultado verdadero del salto. Cuanto esto sucede es necesario deshacer el procesamiento de todas las instrucciones incorrectamente ejecutadas debido al error de predicción y continuar con el procesamiento correcto. Esto se conoce como recuperación de la secuencia correcta. El ejemplo de la Figura 2.21 muestra la predicción incorrecta de un salto condicional en sus dos variantes. Una vez que se conoce el resultado real del salto, los bloques de instrucciones incorrectamente especulados deben ser expulsados de la segmentación para proceder a la ejecución correcta de la secuencia de código. En el ejemplo de la Figura 2.21, aunque las instrucciones $i + 4$ e $i + 5$ se ejecutan siempre, con independencia del resultado del salto, es necesaria su anulación ya que puede suceder que se ejecuten antes de tiempo (predicción efectiva incorrecta) o después (predicción no efectiva incorrecta).

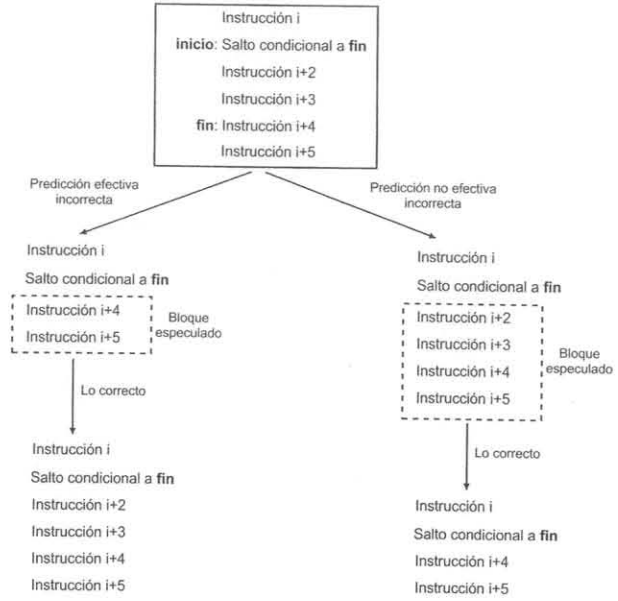


Figura 2.21: Un error de predicción conlleva la ejecución incorrecta de instrucciones que es necesario deshacer una vez conocido el resultado del salto.

Sin embargo, el problema planteado es más complicado debido a que en la secuencia de instrucciones ejecutadas como resultado de la predicción puede suceder que aparezca otra instrucción de salto y se vuelva a producir la ejecución especulativa de una nueva secuencia de instrucciones. La forma habitual para conocer y validar o anular las secuencias de instrucciones que se están ejecutando de forma especulativa es recurrir a etiquetar las instrucciones durante todo el tiempo que permanecen en la segmentación, de forma que se puede conocer su estado en cualquier etapa del cauce.

Desde que entran en la fase de distribución hasta que son terminadas, todas las instrucciones llevan asociadas etiquetas con información de su estado. Dos de estas etiquetas son la especulativa y la de validez. Las etiquetas son campos de uno o más bits que hay en el buffer de reordenamiento, donde todas las instrucciones tienen una entrada en la que se actualiza su estado desde que son distribuidas y hasta que son terminadas arquitectónicamente (se dice que están *en vuelo*). Si la etiqueta *especulativa* es de un bit, un valor de 1 identifica a la instrucción como *instrucción especulada*. Si el procesador permite que haya varias rutas especulativas ejecutándose en paralelo, lo que se denomina *nivel de especulación*, la etiqueta dispondrá de más bits para identificar cada uno de los bloques de instrucciones especuladas. Antes de continuar, es necesario especificar que la dirección de toda instrucción de salto sobre la que se realiza una especulación se almacena en una tabla junto con la etiqueta especulativa que se asociará a todas las instrucciones que formen la ruta especulada directa y sus ramificaciones. Posteriormente, esta etiqueta permitirá identificar y validar las instrucciones especuladas si la predicción es correcta o realizar la recuperación de la secuencia correcta en caso de un error de predicción. La etiqueta que se genera para cada nueva ruta especulada a raíz de un salto tiene que recoger información sobre si proviene a su vez de otra ruta especulada. Una forma de generar las etiquetas es que cada nueva ruta especulada forme su etiqueta tomando la etiqueta de la ruta de que proviene y asignando a 1 el primer bit más significativo que esté a 0. Por ejemplo, si la actual ruta especulada A tiene asignada la etiqueta 100 y se deriva una nueva ruta B a consecuencia de un salto condicional en la ruta A, la etiqueta para las instrucciones especuladas en la ruta B pasará a ser 110. La validación de una ruta implica la asignación a 0 de su bit en las etiquetas especulativas de sus instrucciones y ramificaciones. En el ejemplo anterior, si la ruta A fuese finalmente correcta, el bit que tenía asignado (en este caso, el más significativo) se colocaría a 0 en todas las instrucciones de la ruta A y en sus ramificaciones, en este caso, la B. Las instrucciones de la ruta B pasarían a tener como etiqueta especulativa 010 hasta su validación final o expulsión del cauce.

La etiqueta *validez* permite saber si la instrucción debe o no terminarse arquitectónicamente, es decir, si puede escribir sus resultados en un registro arquitectónico o en la memoria. Inicialmente, todas las instrucciones son válidas. Aunque una instrucción sea válida, no podrá terminarse arquitectónicamente hasta que la etiqueta que la define como especulativa cambie a no especulativa (todos los bits a 0). Mientras esto no suceda, la instrucción especulada y todas las que la suceden, permanecerán retenidas en el buffer de reordenamiento.

Tal y como se ha indicado al explicar los predictores, aunque se inicie el procesamiento de una ruta especulativa, hay que concluir el procesamiento del salto para conocer su resultado y validar o no la predicción. En el momento en que el salto se evalúa, si la predicción coincide con el resultado, las etiquetas especulativas se cambian de forma que las instrucciones especuladas asociadas a ese salto son

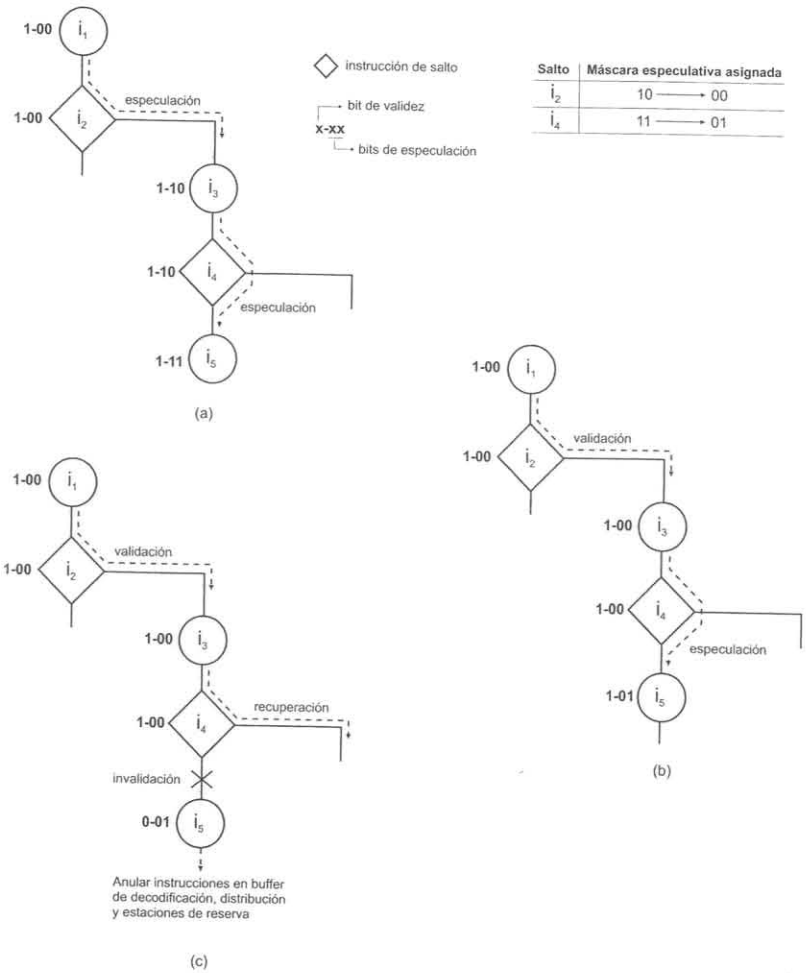


Figura 2.22: Secuencia de especulación, validación, invalidación y recuperación de dos rutas especuladas.

correctas y se pueden terminar arquitectónicamente. Si la predicción es incorrecta, hay que realizar dos acciones:

- *Invalidar las instrucciones especuladas*. El bit de validez de todas esas instrucciones pasa a indicar invalidez y no son terminadas arquitectónicamente, es decir, no acceden al banco de registros o a la memoria para escribir sus resultados. Sus resultados, almacenados temporalmente, se pierden. Por otro lado, todas las instrucciones asociadas a esa ruta especulada que estén en los buffers de decodificación y distribución y en las estaciones de reserva se eliminan.

- *Recuperar la ruta correcta*. Esto implica iniciar la lectura de las instrucciones desde la dirección de salto correcta. Si la predicción incorrecta fue no realizar el salto, se utiliza el resultado del salto como nuevo valor del PC. Si la predicción incorrecta fue realizar el salto, se accede a la tabla en la que se almacenó la dirección de la instrucción de salto y se utiliza para obtener el nuevo valor del PC, es decir, el de la siguiente instrucción (denominada ruta *fall-through*).

La Figura 2.22.a representa una secuencia de instrucciones en la que hay dos rutas especuladas derivadas de dos saltos, el `i2` y el `i4`. Inicialmente, todas las instrucciones son válidas por lo que el bit de validez se establece, por defecto, a `1`. Al iniciar el procesamiento de una ruta especulada como consecuencia de la predicción para el salto `i2`, en las instrucciones especuladas se asigna el bit más significativo de la etiqueta de especulación a `1` por lo que la etiqueta para la ruta es la `10`. Al llegar al salto `i4`, el siguiente bit menos significativo de la etiqueta de especulación se asigna a `1` para indicar que se inicia otra ruta especulada derivada de este nuevo salto: la etiqueta pasa a ser `11`. Una vez que el salto `i2` se evalúa y el resultado coincide con la predicción, la ruta se valida por lo que el bit más significativo de la etiqueta de especulación pasa a ser `0` en todas las instrucciones de la ruta iniciada por `i2` (Figura 2.22.b). La evaluación del salto `i4` no coincide con la ruta especulada lo que obliga a invalidar las instrucciones posteriores en vuelo y las que estén en el buffer de reordenamiento, por ejemplo, la `i5`. Al mismo tiempo, es necesario iniciar la recuperación de la ruta correcta derivada del salto `i4` (Figura 2.22.c).

Un ejemplo real de un procesador superescalar con un elevado nivel de especulación es el PowerPC 970. Dispone de una estación de reserva individual dotada de 16 entradas que alimenta exclusivamente a la unidad funcional de gestión de saltos (BRU). Esto posibilita que el procesador alcance un nivel de especulación de 16.

2.6. Decodificación

Tras la extracción simultánea de varias instrucciones en la etapa de fetch para poder alimentar de forma sostenida el cauce, el paso siguiente es la decodificación. Como se podrá comprobar, esta etapa es una de las más críticas en un procesador superescalar.

En un procesador escalar segmentado del tipo RISC, en la etapa de decodificación se manipula una única instrucción. En base a la decodificación del código de operación de la instrucción se realiza la extracción de los operandos del banco de registros y se establece el comportamiento de la unidad

de control para con esa instrucción a lo largo de toda la segmentación. También en esta etapa de decodificación se analizan las dependencias de datos con las instrucciones que ya están ejecutándose en el cauce y, en caso de varias unidades funcionales, se determina la unidad de destino.

Sin embargo, en un procesador RISC superescalar una etapa clásica de decodificación como la de un procesador escalar tendría que decodificar varias instrucciones en paralelo, e identificar las dependencias de datos con todas las instrucciones que componen el grupo de lectura y con las que están ejecutándose en las unidades funcionales. Otra tarea que complicaría esta etapa es que ahora serían varias las instrucciones que necesitarían leer todos sus operandos o una parte de ellos (si hubiese dependencia de otra instrucción, solo se leerían los que estuviesen ya disponibles). Además, durante la decodificación se debe confirmar si hay saltos en falso en el grupo de lectura con el fin de poder anular la ruta especulada y minimizar el impacto que tiene su procesamiento erróneo. Por todas estas razones, los procesadores superescalares reparten las tareas que realiza una etapa de decodificación clásica en dos etapas denominadas *decodificación* y *distribución*. Por lo general, la etapa de decodificación detecta los saltos en falso y realiza la adaptación del formato de las instrucciones a la estructura interna de control y datos del procesador, mientras que la etapa de distribución se ocupa, entre otras cosas, del renombramiento de los registros, de la lectura de los operandos y del análisis de las dependencias verdaderas. Dependiendo del diseño del procesador, algunas tareas de la etapa de distribución se desplazan a la de decodificación.

Una complicación adicional que surge en los procesadores superescalares CISC es que las instrucciones pueden tener longitudes diferentes. En los procesadores RISC, la longitud de la instrucción es fija, y no hay ningún problema en saber dónde empieza y termina una instrucción. La extracción del buffer de instrucciones se puede realizar a toda velocidad, ya sean una o varias las instrucciones a decodificar. En un procesador CISC es necesario analizar el código de operación de cada instrucción para conocer su formato y longitud y poder determinar dónde termina y dónde comienza la siguiente instrucción y cuáles son los operandos. En el peor de los casos, una instrucción puede comenzar en cualquier byte del grupo de lectura, lo que obliga a introducir hardware adicional para atender cualquier posible situación.

Por lo tanto, los factores que determinan la complejidad de la etapa de decodificación son el tipo arquitectónico del procesador (RISC o CISC) y el número de instrucciones a decodificar en paralelo (ancho de la segmentación). Dada la complejidad de esta etapa las soluciones que se suelen adoptar son tres:

- Descomponer la etapa de decodificación en varias subetapas o fases, aumentando el número de ciclos de reloj que se emplean.

- Realizar una parte de la decodificación antes incluso que la extracción de las instrucciones de la I-caché. Esto se conoce como *etapa de predecodificación* o *decodificacion previa*.

- Traducción de instrucciones. Consiste en descomponer una instrucción en instrucciones más básicas, o unir varias instrucciones en una única instrucción interna. El número de etapas físicas de que se compone la etapa lógica de decodificación de los procesadores superescalares actuales

oscila entre dos y tres. Por ejemplo, el PowerPC 970 recurre a tres etapas que se denominan D1, D2 y D3.

Tras completar la decodificación, las instrucciones ya pasan al buffer de distribución y a las estaciones de reserva para iniciar su verdadero procesamiento. Desde estos buffers se comenzará el reparto de las instrucciones entre las distintas unidades funcionales de que conste el procesador.

2.6.1. Predecodificación

La etapa de predecodificación o decodificación previa se sitúa, habitualmente, antes de la I-caché, de forma que las instrucciones que recibe la I-caché desde el nivel de memoria superior (la caché de nivel 2 o la memoria principal) como consecuencia de un fallo de lectura tienen que pasar forzosamente por esta etapa. La etapa de predecodificación está constituida por hardware situado antes de la I-caché que realiza una decodificación parcial de las instrucciones. Este proceso de decodificación parcial analiza cada instrucción y la concatena un pequeño conjunto de bits con información sobre ella. En los procesadores RISC, el número de bits que se añaden por instrucción oscila entre 4 (UltraSparc) y 7 (PowerPC 920), mientras que en las arquitecturas CISC esta cifra es mayor. Por ejemplo, en el microprocesador CISC

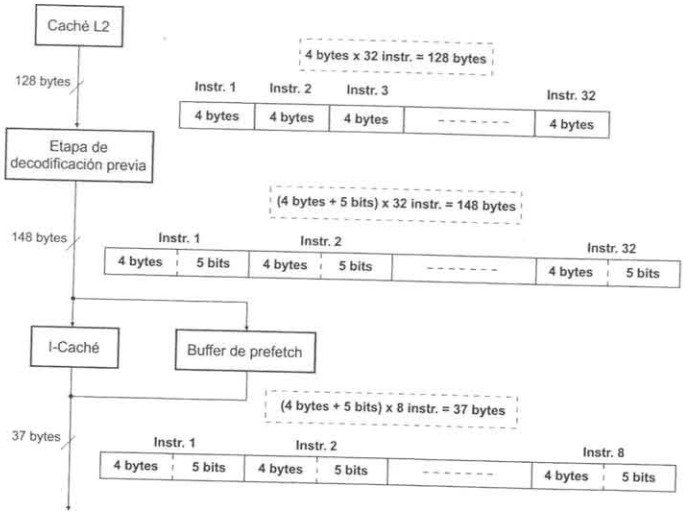


Figura 2.23: Decodificación previa en el PowerPC 970.

AMD 64 se añaden 3 bits por cada byte de la I-caché más otros 4 bits por cada 16 bytes.

Para simplificar las operaciones que se realizan en la etapa de extracción y en la decodificación, el PowerPC 970 cuenta con una etapa de predecodificación situada delante de la I-caché y del buffer de prefetch (Figura 2.23). Esta etapa se ocupa de añadir 5 bits de predecodificación a cada instrucción, de forma que las instrucciones almacenadas en la I-caché y en el buffer de prefetch pasan a tener una longitud de 37 bits. Estos bits de predecodificación se utilizan posteriormente en la etapa de fetch para determinar el tipo de instrucciones de salto (condicional o incondicional) que hay en el grupo de lectura y proceder o no a su especulación, y en la etapa de decodificación para determinar la forma en que se agrupan para su posterior distribución. Además, los bits de predecodificación identifican las instrucciones que son susceptibles de provocar una excepción.

El procesador AMD Opteron utiliza un esquema de predecodificación similar al del PowerPC 970 con la diferencia de que añade 3 bits de predecodificación por cada byte de instrucción y 4 bits adicionales por cada 16 bytes (Figura 2.24). Dado que sus instrucciones son de longitud variable, la misión de los dos primeros bits del grupo de tres es establecer el comienzo (bit START) y el final (bit END) de las instrucciones. Cuando se detecta el final de la instrucción, el tercer bit (bit FUNCTION) se utiliza para indicar si se trata de una instrucción sencilla que puede procesarse directamente por las unidades

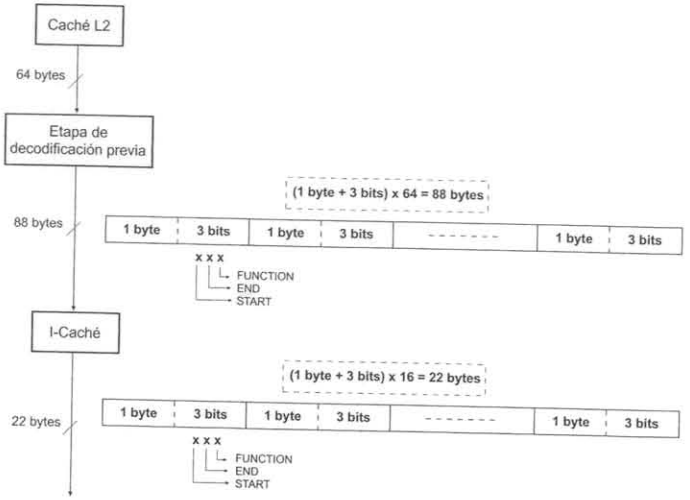


Figura 2.24: Decodificación previa en el AMD Opteron. Por simplificar, no se reflejan los 4 bits que se añaden cada 16 bytes.

funcionales o se trata de una instrucción compleja que tiene que descomponerse en varias operaciones sencillas, conocidas como *microoperaciones* o *micro-ops*.

Pero no todo en la decodificación previa son ventajas. De un rápido estudio de las Figuras 2.23 y 2.24 es fácil percatarse de dos inconvenientes:

- La necesidad de un mayor ancho de banda. En el PowerPC, de un ancho de banda de 32 bytes/ciclo sin predecodificación se pasa a necesitar 37 bytes/ciclo, además del coste del hardware adicional que hay que añadir. Sin embargo, la tendencia es incluir esta etapa debido a que las segmentaciones cada vez son más anchas y complicadas.

- El incremento del tamaño de la I-caché. En el caso del PowerPC 970, el almacenamiento de 5 bits por cada 4 bytes de instrucción se traduce en un aumento del tamaño del 15 %, aproximadamente. En el AMD Opteron, los 64 Kbytes de la I-caché se tienen que incrementar con 20 Kbytes adicionales, un 30 % adicional.

Otra forma de realizar la predecodificación de instrucciones es situar esta fase entre el buffer interno de fetch, que recibe las instrucciones directamente de la I-caché, y el buffer de instrucciones, que alimenta a la etapa de decodificación. Debido a su situación previa al buffer de instrucciones, esta forma de predecodificación se suele considerar parte de la etapa de fetch y no de la etapa de decodificación como tal. La técnica es propia de arquitecturas CISC donde la finalidad de estos bits de predecodificación es:

- Determinar la longitud de las instrucciones. De esta forma, la etapa de decodificación se simplifica ya que se conoce el comienzo y el final de cada instrucción que forme el grupo de lectura.

- Decodificar ciertos prefijos asociados a las instrucciones.

- Señalar determinadas propiedades de las instrucciones a los decodificadores, por ejemplo, el tratarse de instrucciones de salto.

Un ejemplo de este tipo de predecodificación lo proporciona la arquitectura Intel Core. En esta arquitectura, la I-caché entrega secuencias de 16 bytes que, de media, contienen algo menos de 4 instrucciones (depende del tipo de instrucción y del desalineamiento dentro del bloque) del repertorio x86 de Intel. Estos 16 bytes se procesan en una etapa de predecodificación que añade bits para marcar el comienzo y final de las instrucciones y otras características. Esta etapa de predecodificación puede suministrar hasta 6 instrucciones por ciclo al buffer de instrucciones para su posterior decodificación (Figura 2.25).

Lo verdaderamente notorio de la predecodificación es que adelanta parte del trabajo que realiza la etapa de decodificación de forma que se reduce la profundidad de su segmentación. Esto es muy importante, ya que una segmentación menos profunda permite que la recuperación de un fallo en la especulación de un salto no sea tan costosa en ciclos de reloj.

2.6.2. Traducción de instrucciones

Otra tarea que se realiza en la fase de decodificación es la traducción de una instrucción compleja, por ejemplo, de tipo CISC, en un conjunto de instrucciones más básicas de tipo RISC. Estas operaciones básicas se conocen en terminología Intel como *microoperaciones* (*micro-ops*), como *operaciones internas* (IOPs - *Internal OPerations*) en el ámbito de PowerPC o como ROPs (*RIPS Operations*) en terminología AMD. Esta técnica es propia de arquitecturas CISC, aunque también se utiliza en los procesadores RISC para reducir la complejidad de determinadas instrucciones. La aplicación de esta

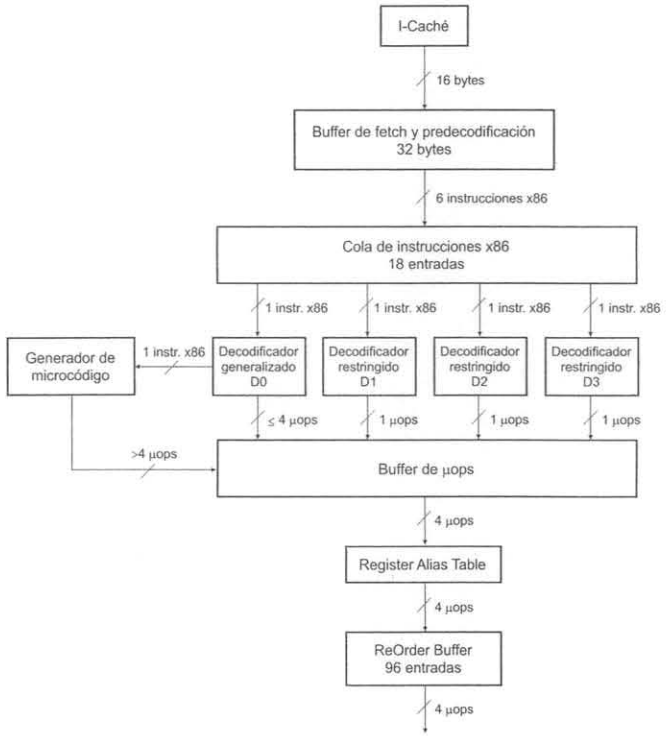


Figura 2.25: Decodificación en la arquitectura Intel Core Microarchitecture.

técnica en los procesadores CISC viene a significar que el interior del procesador es, realmente, un núcleo de procesamiento RISC aunque el repertorio de instrucciones ISA que maneje el programador sea CISC. La idea que subyace es simplificar el repertorio original de instrucciones para que su procesamiento hardware se pueda realizar directamente en el procesador. Nótese que la aplicación de esta técnica implica que, a partir de la etapa de decodificación, todas las instrucciones que se ejecutan en el cauce son ya instrucciones básicas tipo RISC (IOPs, micro-ops, ROPS), no las instrucciones originales que se extrajeron de la I-caché.

Un ejemplo de arquitectura CISC que emplea la traducción de instrucciones es la Intel Core Microarchitecture (ver Figura 2.25). La etapa de decodificación consta de cuatro decodificadores (uno generalizado y lento y tres restringidos pero rápidos) y un generador de microcódigo (*Microcode Engine*) que permiten decodificar cuatro instrucciones de longitud variable en paralelo. El primero de los cuatro decodificadores, el D0, puede manipular cualquier instrucción, generando entre una y cuatro micro-ops por ciclo de reloj a partir de una instrucción; si se trata de una instrucción muy compleja que requiere de cinco o más micro-ops, entonces el D0 la envía al generador de microcódigo que produce secuencias de más de cuatro micro-ops a una velocidad de tres micro-ops por ciclo de reloj. Por su parte, los tres decodificadores restringidos están limitados a instrucciones sencillas (por ejemplo, operaciones registro-registro) que generan una única micro-op. La secuencia de micro-ops obtenida se envía, en grupos ordenados de siete, a un buffer de micro-ops desde donde pasan en grupos de cuatro al RAT (*Register Alias Table*) para el renombrado de registros y, posteriormente, al buffer de reordenamiento (ROB - *ReOrder Buffer*).

Pero, además, la arquitectura Intel Core se caracteriza por complementar la traducción de instrucciones con dos técnicas adicionales: la *macro-fusión* y la *micro-fusión*. La macro-fusión consiste en fusionar o unir ciertos tipos de instrucciones en la fase de predecodificacion (situada a continuación de la I-caché) y enviarlas a uno de los decodificadores para producir una única micro-ops, denominada *macro-fused micro-op*. Esta característica únicamente puede aplicarse a ciertos tipos de instrucciones. Por ejemplo, una instrucción de comparación puede unirse con una instrucción de salto condicional si son adyacentes en el flujo de instrucciones. Cualquiera de los cuatro decodificadores puede producir una *macro-fused micro-op* pero solo se puede producir una por ciclo de reloj. Así, el número de instrucciones decodificadas por ciclo se incrementa de 4 a 5 por ciclo.

La macro-fusión permite a la arquitectura Core realizar más trabajo con menos recursos hardware puesto que son necesarias menos entradas en el buffer de reordenamiento y en las estaciones de reserva al convertirse dos instrucciones en una única macro-fused micro-op. El efecto es un incremento del ancho de banda de decodificación ya que el buffer de instrucciones puede vaciarse mucho más rápidamente si un decodificador retira dos instrucciones por ciclo en lugar de una. Finalmente, la macro-fusión produce un incremento virtual de la anchura de la segmentación en lo que respecta al núcleo de ejecución dinámica (esto es, el número de unidades funcionales). Esto es debido a que una unidad funcional puede estar ejecutando una macro-fused micro-op que corresponde a dos instrucciones. Esto libera unidades funcionales para instrucciones no-macro-fusionadas y permite que el cauce del procesador aparente ser mucho más ancho.

La micro-fusión es una técnica que consiste en decodificar una instrucción que genera dos micro-ops y fundirlas en una única micro-op. Esta nueva micro-op solo ocupa una entrada en el buffer de reordenamiento (ROB) pero al emitirse se desdobla en sus dos componentes originales, ejecutándose en paralelo en unidades funcionales diferentes, o en serie si se trata de la misma unidad funcional. Cuando ambas micro-ops se terminan, se vuelven a considerar como una única micro-op en lo que respecta a su retirada de la segmentación. La aplicación más habitual de la micro-fusión es con las instrucciones de

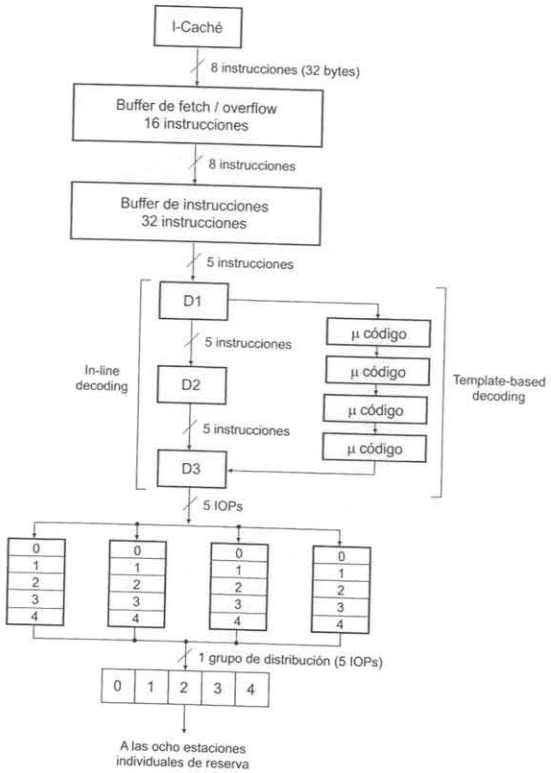


Figura 2.26: Etapa de decodificación del PowerPC 970.

almacenamiento ya que implica su desdoblamiento en dos micro-ops: una para realizar el cálculo de la dirección de destino y otra para la escritura del dato. Ambas micro-ops son inherentemente paralelas ya que su procesamiento implica a dos unidades funcionales diferentes con dos puertos de emisión diferentes. Una vez que las dos micro-ops se completan, se vuelven a fundir en una única micro-op y, en el momento de su retirada, se realiza la escritura en memoria. Al igual que la macro-fusión, la micro-fusión produce un aumento de la capacidad de decodificación y de la eficiencia energética de la arquitectura Intel Core al emitir más micro-ops recurriendo a menos hardware (menos entradas en el buffer de reordenamiento).

Otro ejemplo de procesador, esta vez RISC, que utiliza traducción de instrucciones es el PowerPC 970 (Figura 2.26). Cada ciclo de reloj extrae cinco instrucciones del buffer de instrucciones y se comienzan a procesar en la etapa de decodificación compuesta por las tres etapas D1, D2 y D3 y conocida como *in-line decoding*. Cuando una instrucción tiene que ser traducida en más de dos IOPs, el procesador recurre a una extensión de la etapa de decodificación compuesta por un cauce de cuatro subetapas y que realiza una decodificación basada en plantillas. Esta etapa se denomina *template-based decoding*. Esta extensión es capaz de generar hasta 4 IOPs por ciclo de reloj para emular el comportamiento de una instrucción normal. En terminología PowerPC se denominan *instrucciones rotas* (*cracked instruction*) a aquellas que se decomponen en dos IOPs, e *instrucciones microcodificadas* (*microcoded instructions*) a las que es necesario descomponer en tres o más IOPs.

El objetivo final de la etapa de decodificación del PowerPC 970 es producir grupos de distribución (*dispatch groups*) formados por 5 IOPs, los cuales pueden formarse con IOPs obtenidas de instrucciones pertenecientes a grupos de lectura consecutivos. Los bits de predecodificación asociados a cada instrucción ayudan a determinar cómo componer estos grupos para maximizar el rendimiento del núcleo de ejecución fuera de orden.

2.7. Distribución

La etapa de distribución es la que establece el punto de partida para la ejecución de instrucciones en paralelo y fuera de orden en una segmentación superescalar. A diferencia de lo que sucede en una segmentación escalar, donde todas las instrucciones se ejecutan una tras otra, en un procesador superescalar la etapa de distribución se ocupa de repartir las instrucciones según su tipo entre las distintas unidades funcionales para que se pueda proceder a su ejecución en paralelo. La distribución es el último componente del *front-end* de un procesador superescalar tras las etapas de fetch y de decodificación. Es el punto de inflexión entre el procesamiento centralizado de las instrucciones y su procesamiento distribuido.

Tras la decodificación, las instrucciones se depositan temporalmente en dos buffers conocidos como *buffer de distribución* o *ventana de instrucciones* y *buffer de terminación* o *de reordenamiento*. Por el momento, la ventana de instrucciones será la que recibirá más atención en los párrafos siguientes ya que el buffer de terminación se estudiará posteriormente con mayor detalle. Aunque el buffer de terminación se sitúa en el camino lógico de la segmentación justo después de la etapa de ejecución, es importante

empezar a adquirir una visión global del funcionamiento de un procesador superescalar y saber que las instrucciones se almacenan en él al distribuirse y en el mismo orden que tienen en el programa. Cuando las instrucciones concluyan su procesamiento fuera de orden, será el buffer de terminación el que permitirá volver a reestablecer el orden y garantizar la consistencia del procesador y de la memoria.

Para poder ejecutar una instrucción, esta debe disponer de todos sus operandos fuente y estar libre una de las unidades funcionales que le corresponda según su tipo (aritmética entera, aritmética coma flotante, salto, etc.) Sin embargo, puede suceder que los operandos fuente no estén disponibles y haya que esperar por algún resultado de las instrucciones que están ya ejecutándose. También puede ocurrir que los operandos estén disponibles, pero no así su unidad funcional. O que esté todo disponible pero no se pueda enviar la instrucción a la unidad funcional debido a que no haya suficientes buses ya que hay un límite en la cantidad de instrucciones por ciclo que se pueden enviar a través de la red de enrutamiento desde la ventana de instrucciones a las unidades funcionales (los ya conocidos riesgos estructurales).

La solución más inmediata a este problema es, simplemente, detener la instrucción en la etapa de decodificación hasta que todo esté listo para poder emitirla. Pero esto reduce el rendimiento de toda la decodificación y, por ello, de todo el procesador ya que introduce burbujas en la segmentación. La solución por la que se opta es *desacoplar la etapa de decodificación de la de ejecución* utilizando para ello la ventana de instrucciones. El desacoplo consiste en no detener la instrucción sino decodificar lo que se pueda y hacer que avance hacia la ventana de instrucciones. Para ello, en la ventana de instrucciones se deposita la instrucción con los identificadores de los operandos fuente pero, además, se indica mediante un bit de validez por operando si éste está disponible en ese momento o no. Tras esto, la instrucción permanecerá a la espera en la ventana de instrucciones para poder emitirse a la unidad funcional correspondiente una vez que se cumplan las condiciones necesarias para ello.

2.7.1. Organización de la ventana de instrucciones

Existen varias formas de organizar la ventana de instrucciones:

- *Estación de reserva centralizada* (Figura 2.27.a). Es lo que se ha definido hasta ahora como ventana de instrucciones o buffer de distribución. También se conoce con el término ventana de emisión o cola de emisión. La arquitectura Intel Core utiliza esta organización.

- *Estaciones de reserva distribuidas o individuales* (Figura 2.27.b). Cada unidad funcional dispone de una estación de reserva propia. Existe un buffer de distribución que recibe las instrucciones de la etapa de decodificación y que se ocupa de distribuirlas a las estaciones de reserva individuales según su tipo. Los procesadores PowerPC 750 utilizan estaciones de reserva particulares para cada una de las unidades funcionales.

- *Estaciones de reserva en clústers o compartidas* (Figura 2.27.c). Las estaciones de reserva reciben las instrucciones del buffer de distribución pero una estación de reserva puede servir a varias unidades funcionales del mismo tipo. Un ejemplo de esta configuración es el PowerPC 970.

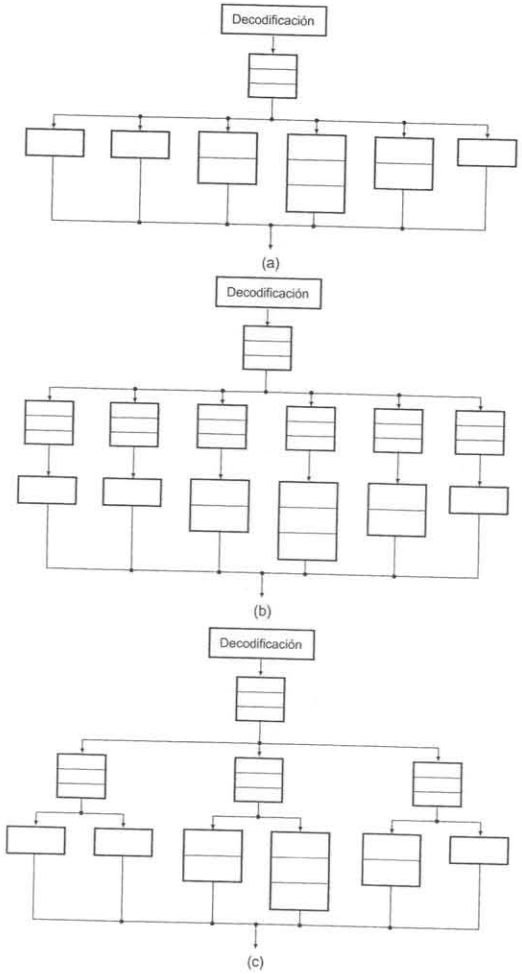


Figura 2.27: Organizaciones de la ventana de instrucciones y estaciones de reserva: (a) centralizada, (b) distribuidas, (c) compartidas.

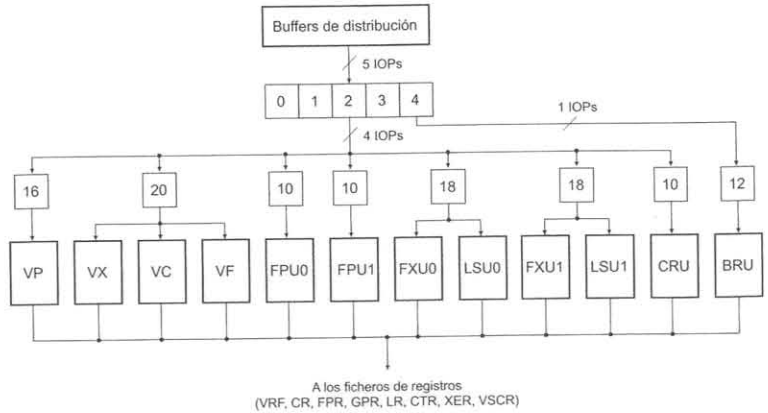


Figura 2.28: Detalles de la etapa de distribución del PowerPC 970.

El PowerPC 970 cuenta con ocho estaciones de reserva individuales, denominadas colas de emisión (*issue queues*), que atienden a doce unidades funcionales (Figura 2.28). Las colas de emisión reciben en cada ciclo reloj un grupo de distribución formado por 5 IOPs que se distribuyen por las colas según su tipo de operación. Las colas se organizan de la siguiente forma:

- Una estación de 18 entradas para las unidades FXU0 y la LSU0.
- Una estación de 18 entradas para las unidades FXU1 y la LSU1.
- Una estación de 12 entradas para la unidad de salto BRU.
- Una estación de 10 entradas para la unidad CRU.
- Una estación de 10 entradas para la unidad FPU0.
- Una estación de 10 entradas para la unidad FPU1.
- Una estación de 20 entradas para las unidades de cálculo vectorial VF, VX y VC.
- Una estación de 16 entradas para la unidad vectorial de permutación VP.

La utilización de una u otra configuración es parte de las decisiones que debe tomar el equipo de ingeniería que diseña la arquitectura del procesador. Una estación de reserva centralizada conlleva un hardware de control muy complejo ya que hay que acceder de forma asociativa a un único buffer

multipuerto con un elevado número de entradas. Recurrir a estaciones de reserva individuales implica que cada buffer dispondrá de un menor número de entradas y de menor longitud ya que cada estación alimenta a una única unidad funcional y no necesita almacenar el tipo de instrucción. Sin embargo, la estructura distribuida aumenta la complejidad de los buses que hay que utilizar para reenviar los resultados de las distintas unidades funcionales (los operandos fuente pendientes) a las estaciones de reserva y a los bancos de registros y poder emitir así nuevas instrucciones.

La organización de las estaciones de reserva obliga a matizar el significado de ciertos términos, ya de por sí confusos en la literatura. Se entiende por *distribución* el asociar una instrucción a una unidad funcional y por *emitir* el enviar la instrucción a la unidad funcional para comenzar su ejecución. De acuerdo con esto, en una estación de reserva centralizada los términos distribución y emisión significan lo mismo ya que la asociación de la instrucción a la unidad funcional se produce en el momento del envío. En una organización basada en estaciones de reserva individuales, la distribución es el envío desde el buffer de distribución a la estación de reserva individual y la emisión es el envío desde la estación de reserva individual a la unidad funcional para que se ejecute. En una organización basada en estaciones de reserva compartidas existe distribución y distribución/emisión: la primera distribución es el envío desde el buffer de distribución a una estación de reserva y la distribución/emisión se produce al enviar la instrucción a una de las unidades funcionales que tiene asignada.

2.7.2. Operativa de una estación de reserva individual

Una estación de reserva es un buffer de almacenamiento con múltiples entradas en donde se almacenan las instrucciones ya decodificadas. Cada entrada del buffer es un conjunto de bits agrupados por campos y representa a una instrucción. Aunque, el número y longitud de los campos que componen cada una de las entradas varía en función del diseño del procesador y del tipo de unidad funcional que debe servir, una estructura que se puede considerar como genérica para una estación de reserva individual constaría de los siguientes campos:

- *Ocupado* (O): Indica que la entrada está ocupada por una instrucción válida pendiente de emisión.
- *Código de operación* (CO): Contiene el código de operación de la instrucción.
- *Operando 1* (Op1): Si el registro que corresponde al primer operando fuente está disponible, este campo almacena el valor del registro o el identificador (aunque, como se verá, esto depende del modo de lectura de los operandos). Si no está disponible contiene el identificador del registro.
- *Válido 1* (V1): Indica si el operando fuente está disponible o no.
- *Operando 2* (Op2): Similar al Op1.
- *Válido 2* (V2): Similar al V1.

- *Destino* (D): El identificador del registro destino que almacenará el resultado de la operación de forma temporal. Posteriormente, cuando se explique el renombramiento de registros se entenderá con claridad la razón de que sea un almacenamiento temporal.
- *Listo* (L): Indica que todos los operandos ya están disponibles y la instrucción puede emitirse a la unidad funcional correspondiente.

Cuando una instrucción se distribuye y se almacena en una estación de reserva, el bit O se coloca a 1 para indicar que la instrucción está pendiente de ser ejecutada. Si los dos operandos fuente estuviesen disponibles, los campos V1 y V2 contendrían el valor 1. Que los operandos fuente estén disponibles significa que los campos Op1 y Op2 almacenarán el identificador del registro fuente o su contenido según sea la forma en que se proceda a leer los operandos (más adelante se tratará este punto). Si los operandos no están disponibles entonces se almacena el identificador del registro fuente. Una vez que los operandos estén listos, es decir, V1 y V2 a 1, entonces el bit L se coloca a 1 y la instrucción ya se encuentra preparada para ser emitida en cuanto la unidad funcional asignada y el bus de enrutamiento estén libres. Observe que el formato descrito es para operaciones aritméticas registro-registro. Dependiendo del tipo de instrucción, el formato de las entradas de la estación de reserva variará.

Con independencia de la forma de organizar las estaciones de reserva, es decir, individualmente o compartidas, el formato de las entradas es similar. La diferencia surge en las entradas del buffer de distribución que se ocupa de distribuir las instrucciones a las estaciones individuales o agrupadas. En el caso de estaciones individuales o compartidas, las entradas del buffer de distribución no disponen de bits de validez ya que se asignan cuando las instrucciones son enviadas desde el buffer a las estaciones. Si se

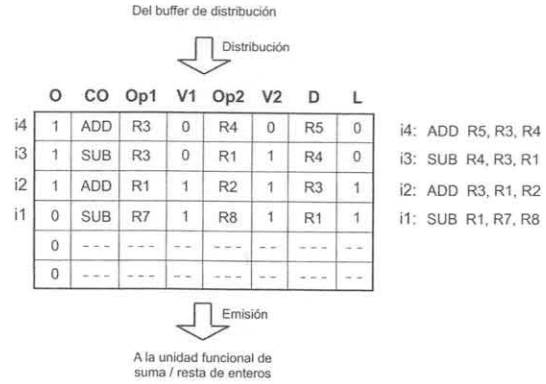


Figura 2.29: Estados de una instrucción en una estación de reserva individual.

recurre a una única estación de reserva centralizada, las entradas sí son similares a la descrita y los bits de validez se establecen al salir de la etapa de decodificación.

La Figura 2.29 presenta un ejemplo de una estación de reserva individual asociada a una unidad funcional de suma/resta de enteros. La estación cuenta con 6 entradas en la que hay 4 instrucciones en tres situaciones diferentes. La instrucción `i1` ubicada en la primera entrada (de arriba hacia abajo) ya ha sido emitida (L=1) por lo que tiene el bit de ocupado a 0; esto indica que la entrada puede ocuparse por una nueva instrucción decodificada que venga del buffer de instrucciones. La instrucción `i2` tiene todos los operandos disponibles (`V1=1`, `V2=1`) por lo que está en espera de ser enviada a la unidad funcional (L=1). La tercera y cuarta instrucciones tienen parte o todos sus operandos fuente no disponibles ya que son resultados de instrucciones previas. Las dos últimas entradas contienen información de instrucciones que fueron emitidas con anterioridad por lo que las entradas pueden ser reutilizadas.

La Figura 2.30 presenta otro ejemplo académico en el que se aprecia la evolución de un grupo de cinco instrucciones en el buffer de distribución y en las estaciones de reserva individuales asociadas a una unidad funcional de suma/resta y a una de multiplicación/división. Por simplificar el ejemplo, los bits de validez ya vienen establecidos desde la etapa de decodificación, por lo que el buffer de distribución cuenta con ellos. Para poder seguir la evolución del ejemplo, hay que considerar que el buffer de distribución puede recibir y suministrar hasta 5 instrucciones/ciclo, que la unidad de suma/resta consume 1 ciclo y la de multiplicación/división 2 ciclos estando segmentada, y que, al final de su último ciclo de ejecución, las unidades funcionales comunican a las estaciones de reserva la disponibilidad de sus resultados. De esta forma, la estación de reserva podría emitir una instrucción en el ciclo de reloj siguiente, ya que la unidad funcional le habrá ya comunicado la disponibilidad de un resultado y éste podría ser uno de los operandos por los que espera alguna de sus instrucciones. También se ha considerado que la lectura de los operandos fuente del fichero de registros no consume tiempo y se efectúa al emitir la instrucción a la unidad funcional. Teniendo en cuenta estas simplificaciones, la evolución del grupo de instrucciones ciclo a ciclo es la siguiente:

- Ciclo *i*. Las 5 instrucciones son recibidas por el buffer de distribución desde la etapa de decodificación.
- Ciclo $i + 1$. De las 5 instrucciones se distribuyen las 4 primeras instrucciones a las estaciones de reserva individuales dado que la capacidad de estas últimas es de 2 instrucciones y ya se han saturado. Ello provoca que la instrucción `i5` permanezca en el buffer de distribución a la espera de una entrada libre en la estación de reserva. Las otras entradas del buffer de distribución se marcan como no ocupadas.
- Ciclo $i + 2$. Las instrucciones `i1` e `i4` tienen todos sus operandos fuente disponibles por lo que son emitidas a sus respectivas unidades funcionales. En este mismo ciclo, el buffer de distribución envía la instrucción `i5` a la estación de reserva al quedar una entrada libre tras la emisión de la `i4`.
- Ciclo $i+3$. La unidad de suma/resta ha concluido el procesamiento de `i1` al final del ciclo anterior y ha comunicado a las estaciones que el resultado que va al registro R1 está disponible. Las estaciones

i1: SUB R1, R7, R0
i2: MULT R5, R1, R7
i3: ADD R5, R2, R1
i4: DIV R4, R3, R7
i5: MULT R6, R4, R5

(a) Secuencia de instrucciones

Ciclo i: Recepción de la etapa de decodificación de i1, i2, i3, i4 e i5

	O	CO	Op1	V1	Op2	V2	D	L
i5	1	MULT	R4	0	R5	0	R6	0
i4	1	DIV	R3	1	R7	1	R4	1
i3	1	ADD	R2	1	R1	0	R5	0
i2	1	MULT	R1	0	R7	1	R5	0
i1	1	SUB	R7	1	R0	1	R1	1

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

Ciclo i+1: Distribución a las estaciones de reserva de i1, i2, i3 e i4

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
i5	1	MULT	R4	0	R5	0	R6	0

	O	CO	Op1	V1	Op2	V2	D	L
i3	1	ADD	R2	1	R1	0	R5	0
i1	1	SUB	R7	1	R0	1	R1	1

	O	CO	Op1	V1	Op2	V2	D	L
i4	1	DIV	R3	1	R7	1	R4	1
i2	1	MULT	R1	0	R7	1	R5	0

Figura 2.30: Ejemplo de emisión de instrucciones entre estaciones de reserva distribuidas (continúa).

Ciclo i+2: Emisión de i1 e i4. Distribución de i5

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

	O	CO	Op1	V1	Op2	V2	D	L
i3	1	ADD	R2	1	R1	0	R5	0

	O	CO	Op1	V1	Op2	V2	D	L
i5	1	MULT	R4	0	R5	0	R6	0
i2	1	MULT	R1	0	R7	1	R5	0

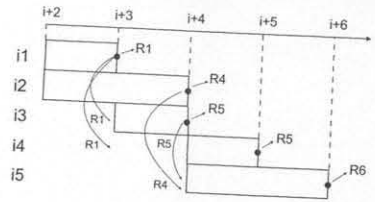
Ciclo i+3: Fin de ejecución de i1. Emisión de i2 e i3

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

	O	CO	Op1	V1	Op2	V2	D	L
	--	---	---	--	---	--	---	--
	--	---	---	--	---	--	---	--

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
i5	1	MULT	R4	1	R5	1	R6	1

(b) Evolución de las estaciones de reserva



(c) Secuencia temporal de ejecución y reenvío de operandos

Figura 2.30: [Continuación] Ejemplo de emisión de instrucciones entre estaciones de reserva distribuidas.

han detectado que R1 estaba disponible por lo que, en el siguiente ciclo, ya podrán emitir i3 e i2 al tener todos los operandos disponibles.

- Ciclo $i + 4$. La instrucción i4 ha concluido en el ciclo anterior ya que consume 2 ciclos de reloj. También ha concluido la i3 al consumir un ciclo. Por ello se produce la emisión de i5 debido a que los operandos que requiere ya se han generado por parte de i3 e i4.

- Ciclo $i + 5$. Concluye el procesamiento de i4.

- Ciclo $i + 6$. Concluye el procesamiento de i5.

Observe que, de forma implícita, el mecanismo de las estaciones de reserva resuelve el problema de las dependencias RAW. La solución se obtiene al forzar la espera por la disponibilidad de los operandos pero permitiendo la ejecución distribuida y fuera de orden de instrucciones no dependientes. Pero el ejemplo anterior presenta el problema de las falsas dependencias, cuya solución vendrá dada al estudiar el renombramiento de registros y el buffer de terminación. En este caso, observe la existencia de un riesgo WAW entre las instrucciones i2 e i3.

Aunque el formato de las entradas de la estación de reserva individual sea sencillo, lo verdaderamente complejo es el hardware de control necesario para la asignación y emisión de las instrucciones. Las fases por las que pasa una instrucción desde que abandona el buffer de instrucciones hasta que se emite desde la estación de reserva individual son tres: *distribución*, *supervisión* y *emisión* (Figura 2.31).

2.7.2.1. Fase de distribución

Consiste en el envío de una instrucción desde el buffer de instrucciones a la estación de reserva individual que le corresponda según su tipo. La introducción de las instrucciones en la estación de reserva se realiza de forma ordenada por parte del hardware conocido como *lógica de asignación*. La lógica de asignación se ocupa de ubicar correctamente la instrucción recibida para lo que debe llevar un registro de las instrucciones almacenadas y de las entradas que están libres (para ello se dispone de los bits de ocupado en la estación de reserva). Además, también se ocupa se establecer inicialmente los bits de validez. En caso de que se utilice una estación de reserva centraliza, esta fase sería similar con la única diferencia de que el origen de las instrucciones sería la etapa de decodificación.

Si la lectura de los operandos fuente se realizase en esta fase, los identificadores de los operandos que estuviesen disponibles serían sustituidos por los valores obtenidos del fichero de registros. Si no hubiese lectura, se mantendrían los identificadores con independencia de si el operando estuviese disponible o no. En la Figura 2.31 no se contempla la lectura de los operandos fuente hasta que la instrucción se envía a la unidad funcional por lo que las entradas de la estación solo almacenan los identificadores de registros y sus bits de validez.

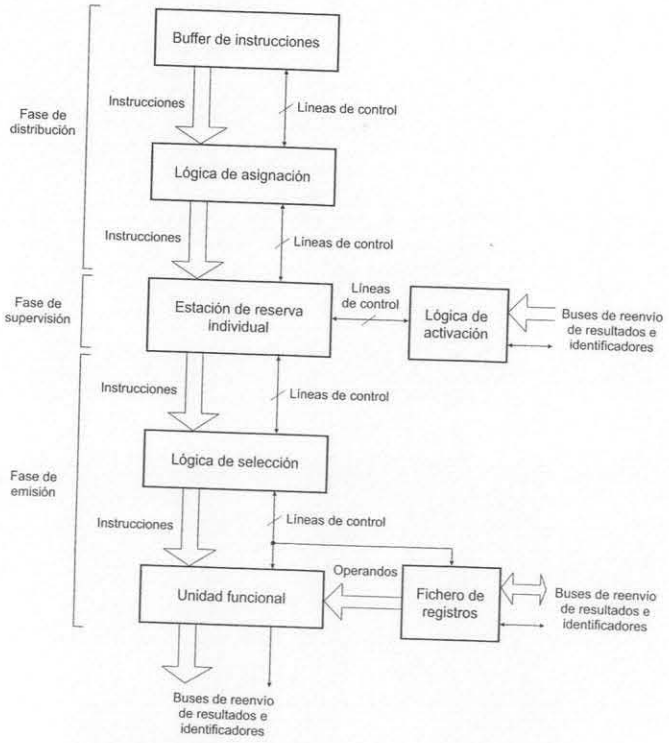


Figura 2.31: Fases de una instrucción en la etapa de distribución.

2.7.2.2. Fase de supervisión

Una vez que la instrucción se encuentra almacenada en la estación de reserva, comienza su fase de supervisión, que concluye en el momento en que tiene los dos operandos fuente disponibles y está en condiciones de ser emitida. Durante la fase de supervisión, las instrucciones que tienen algún operando fuente marcado como no disponible se encuentran en una espera activa, supervisando continuamente los buses de reenvío, (también conocido como CDB - *Common Data Bus*) que es donde cada unidad funcional, una vez concluida su operación, publica el resultado y el identificador del registro destino en

el que almacenar el resultado. El hardware que realiza la supervisión de los buses se suele denominar *lógica de activación*.

Durante esta espera activa, la lógica de activación está comparando continuamente los identificadores de los operandos no disponibles de todas las instrucciones que hay en la estación de reserva con los identificadores que se publican en los buses de reenvío por parte de las unidades funcionales. En el momento en que se produce una coincidencia de identificadores, se cambia el bit de validez del operando fuente correspondiente (`V1`, `V2`), se lee el valor del operando del bus de reenvío (si se realiza la lectura previa de los operandos disponibles; en caso contrario, solo se cambia el bit de validez) y, si todos los operandos ya están listos, se activa el bit que señala a la instrucción como preparada para ser emitida (bit `L`). La activación del bit `L` se conoce como *activación de la instrucción*. En la Figura 2.31 como no se utiliza lectura de operandos hasta el envío de la instrucción a la unidad funcional, la única modificación que realiza la lógica de asignación en la estación de reserva es el cambio de los bits de validez (bits `V1` y `V2`) y de listo (bit `L`) cuando los operandos fuente están disponibles.

La complejidad de la lógica de activación es elevada y aumenta con el tamaño y el número de las estaciones de reserva. Por una parte, la lógica de activación tiene que examinar continuamente la coincidencia de alguna de las etiquetas que circulan por los buses de reenvío con todos los operandos no disponibles, con lo que a mayor número de entradas, mayor número de operandos a comparar. Por otra parte, a mayor número de unidades funcionales y estaciones de reserva, la complejidad de los buses de reenvío aumenta al igual que la red de enrutamiento necesaria para distribuir las instrucciones a las estaciones de reserva individuales.

2.7.2.3. Fase de emisión

Una vez que la instrucción tiene todos sus operandos disponibles comienza la fase de emisión. Al igual que la anterior, esta fase se caracteriza por la espera activa hasta que la *lógica de selección* determina la instrucción que se puede emitir de entre todas las disponibles. La fase de emisión da paso a la etapa de ejecución, tal y como se muestra en la Figura 2.31. En el esquema de la Figura 2.31 se puede apreciar que la lectura de los operandos se realiza en el momento en que la instrucción se emite a la unidad funcional. De forma simultánea, el código de la operación y el identificador del registro destino se envían a la unidad funcional desde la estación de reserva y los operandos, con la información proporcionada por la estación de reserva, se leen del fichero de registros y se remiten a la unidad funcional.

Por lo general, cuando se emite una instrucción se libera la entrada asociada mediante la asignación de un 0 en el bit de ocupado para que se pueda distribuir una nueva instrucción. Sin embargo, para las instrucciones que pueden provocar algunos tipos de interrupciones, esto no es así. Estas instrucciones se mantienen en la estación de reserva hasta que no hayan concluido completamente su ejecución. Un ejemplo de este tipo de instrucciones son las de carga. Si una instrucción de carga produce un fallo de lectura en la caché de datos, lo habitual es detener la segmentación durante muchos ciclos hasta que se obtiene el dato. El rendimiento puede mejorar si la instrucción de carga es expulsada de la segmentación mientras se extrae el dato, no deteniendo la unidad funcional y, posteriormente, volver a emitirla, pero ya con la seguridad de que el dato está disponible y no habrá fallo.

Si en una estación de reserva individual solo hay una instrucción en condiciones de ser emitida, no existe ningún problema: se envía a la unidad funcional y se marca su entrada como libre para que pueda ser ocupada por una nueva instrucción. El problema surge cuando hay varias instrucciones en condiciones de ser emitidas en el mismo ciclo de reloj y la lógica de selección debe decidir cuál de entre todas ellas

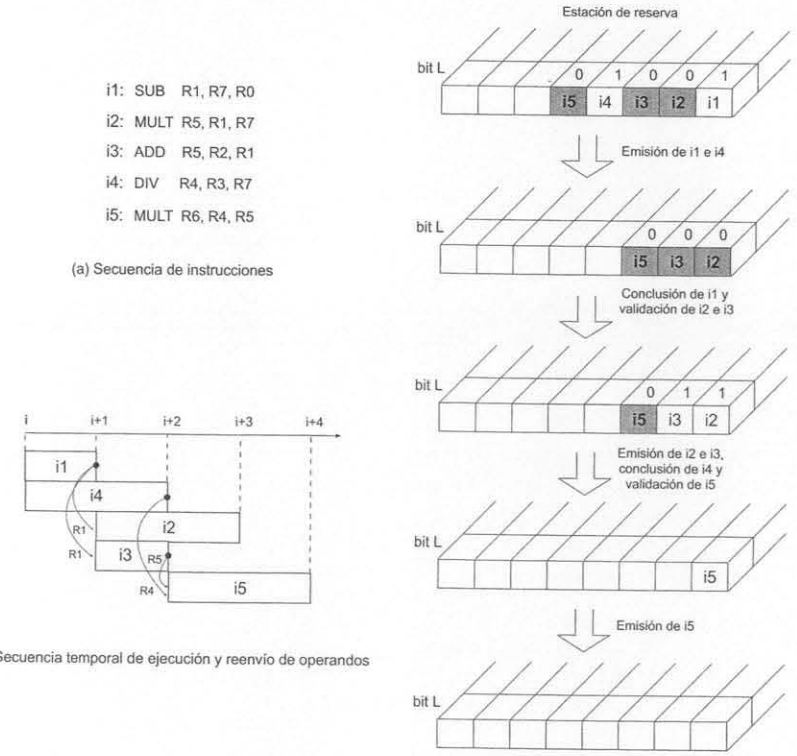


Figura 2.32: Ejemplo de emisión de una secuencia de cinco instrucciones a una estación de reserva individual que alimenta dos unidades funcionales.

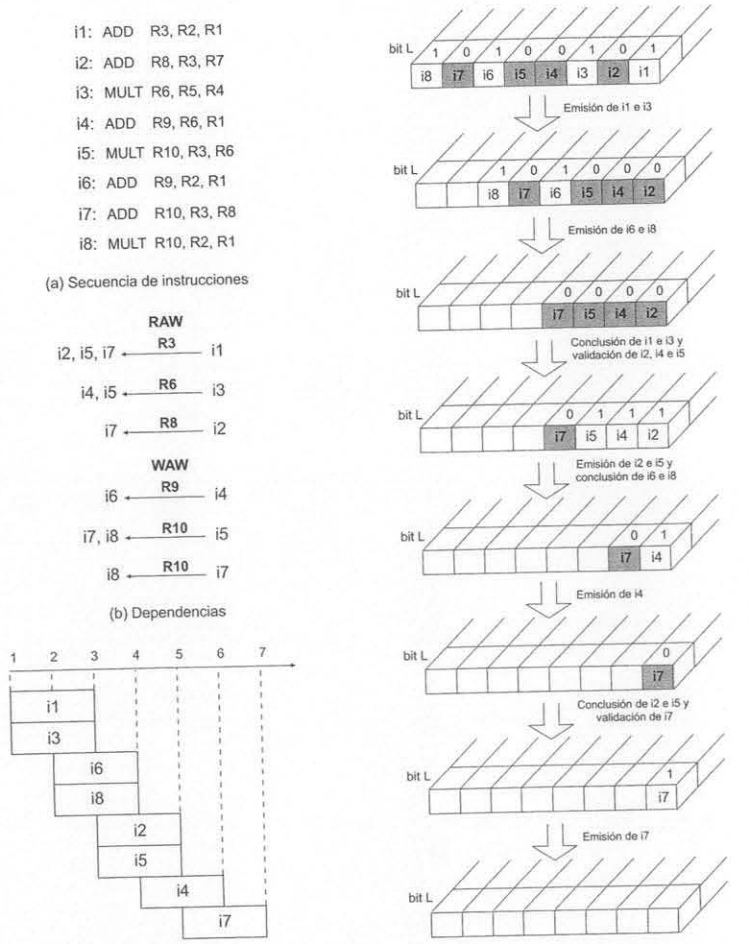


Figura 2.33: Ejemplo de emisión de una secuencia de 8 instrucciones a una estación de reserva individual que alimenta a 2 unidades funcionales.

hay que emitir. El problema es análogo en las estaciones de reserva centralizadas o compartidas. Estas estaciones de reserva, que alimentan a varias unidades funcionales, pueden emitir varias instrucciones por ciclo de reloj pero el problema se presenta cuando el conjunto de instrucciones que están listas para emitirse es mayor que el número de instrucciones que el hardware permite emitir por ciclo. En este caso también es necesario seleccionar, no una, pero sí un subconjunto.

Centrando la discusión en una estación de reserva individual, la lógica de selección no es más que un *algoritmo de planificación* o *planificador dinámico*. El más habitual consiste en seleccionar la instrucción más antigua de entre todas las disponibles, es decir, aplicar el orden del programa. Una vez que se conoce la instrucción a emitir hay que esperar a que la unidad funcional esté libre y pueda comenzar a ejecutar la instrucción. Los primeros procesadores superescalares se caracterizaban por realizar una *emisión con bloqueo y ordenada*. Esto implicaba que las instrucciones salían en orden de la estación de reserva y si una instrucción no contaba con sus operandos disponibles debía esperar, bloqueando la emisión de las instrucciones posteriores tuviesen o no los operandos listos. Los procesadores actuales ya realizan *emisión sin bloqueo y desordenada*, lo que mejora notablemente su rendimiento.

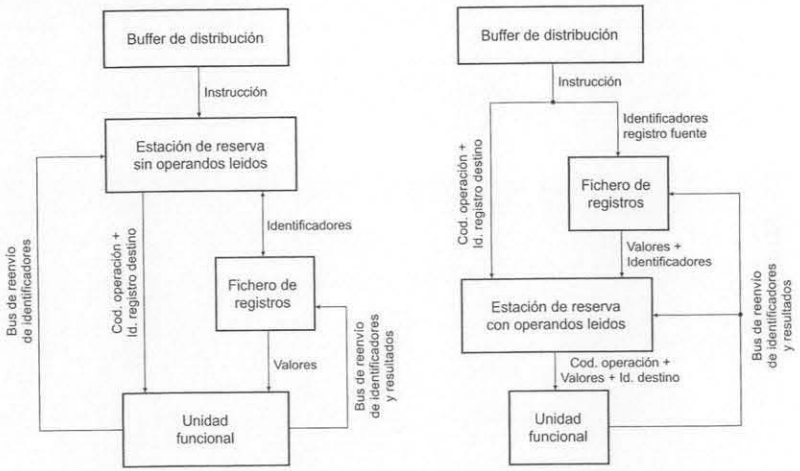
La Figura 2.32 presenta un ejemplo de emisión desordenada en el que la política de planificación es emitir la instrucción más antigua de entre las disponibles. En el ejemplo se considera que una estación de reserva alimenta una unidad funcional de suma/resta (1 ciclo) y una de multiplicación/división (2 ciclos y segmentada). Además, se supone que en el mismo ciclo en el que la unidad funcional genera el operando, la estación de reserva actualiza sus bits de validez; esto permite que en el ciclo siguiente se pueda emitir la instrucción. Observe que, inicialmente, la instrucción `i4` está disponible y se emite sin esperar a que las instrucciones `i2` e `i3` lo estén. En caso de recurrir a la emisión con bloqueo, la instrucción `i4` tendría que esperar a la emisión de `i2` e `i3`, una vez concluida la ejecución de `i1`. Esto tendría como repercusión que la secuencia habría tardado en ejecutarse en las unidades funcionales un ciclo más (5 ciclos) que con respecto a la emisión sin bloqueo (4 ciclos) lo que supone una mejora del rendimiento del 20 %.

La Figura 2.33 muestra otro ejemplo de emisión de un grupo de instrucciones por parte de una estación de reserva individual que alimenta una unidad de suma/resta (2 ciclos) y una unidad de multiplicación/división (2 ciclos). Se puede apreciar con claridad en el diagrama de tiempos (Figura 2.33.c) que las instrucciones se ejecutan de forma totalmente desordenada en las unidades funcionales pero respetando las dependencias RAW entre las diferentes instrucciones. La ejecución fuera de orden produce la violación de las dependencias WAW que existen entre las instrucciones `i5`, `i7` e `i8`. El renombramiento de registros y el buffer de terminación será la solución para respetar este tipo de dependencias de datos.

La emisión de instrucciones desde las estaciones de reserva puede realizarse de forma alineada o no alineada. Alineada significa que la ventana de distribución no puede enviar nuevas instrucciones a la estación de reserva hasta que esta no esté completamente vacía. La emisión no alineada implica que se pueden distribuir instrucciones desde el buffer de distribución siempre que queden entradas libres en las estaciones de reserva. En los ejemplos que se han estudiado hasta aquí se ha recurrido a la emisión no alineada de instrucciones.

2.7.3. Lectura de los operandos

Hasta este punto se ha considerado que la lectura de los operandos fuente se realizaba siempre en el momento en que se emitían las instrucciones desde las estaciones de reserva individuales a las unidades funcionales (ver Figura 2.31). Esta forma de organizar la emisión y ejecución de las instrucciones se denomina *planificación sin lectura de operandos* (Figura 2.34.a). Ello se debe a que cuando la instrucción se emite por parte del planificador, todavía no se han extraído los valores de los operandos fuente del fichero de registros. En este esquema, cuando una instrucción se distribuye desde la ventana de instrucciones a una estación de reserva individual, las entradas contienen únicamente identificadores de registros fuente (ver Figura 2.29).



(a) Planificación sin lectura de operandos (b) Planificación con lectura de operandos

Figura 2.34: Formas de lectura de los operandos.

Aunque se estudiará posteriormente, hay que precisar que el fichero de registros también cuenta con bits de validez de cada registro. Cuando se distribuyen las instrucciones desde la ventana de instrucciones a las estaciones de reserva se analizan las instrucciones en busca de dependencias verdaderas, y se marcan en el fichero de registros como inválidos (bit de validez a 0) aquellos registros que resulten ser el destino de una operación. La lógica de asignación es la que se encarga de asignar los bits de validez de los operandos fuente mediante una consulta al fichero de registros. De esta forma, los operandos fuente de una instrucción (instrucción consumidora) se marcarán como disponibles si el bit de validez en el

fichero de registros está a 1 (válido) ya que indica que no es el resultado de ninguna instrucción previa (instrucción productora). Observe que en el ejemplo correspondiente a la Figura 2.30, por simplificar la explicación, se ha considerado que la asignación de los bits de validez ya venía dada por la etapa de decodificación. En este texto se considera que la asignación de los bits de validez al fichero de registros y a las entradas de las estaciones de reserva se realiza en la fase de distribución y no en la de decodificación. Cuando la instrucción se emite, el código de operación y el identificador del registro destino se envían directamente a la unidad funcional, mientras que los identificadores de los operandos se envían al fichero de registros para la lectura de sus valores, desde donde se remiten a la unidad funcional. La unidad funcional, una vez que concluye su operación, coloca el resultado y el identificador del registro destino en los buses de reenvío. Por un lado, el identificador del registro destino se envía a la estación de reserva, la cual realiza las comparaciones oportunas y activa los bits de validez de aquellos operandos cuyo identificador coincida con el que se ha publicado en el buffer. Por otro lado, el fichero de registros lee el identificador del registro destino y el resultado y lo escribe en el fichero actualizando su bit de validez.

Una de las ventajas de este modo de organización es que el ancho de las estaciones de reserva y de los buses de reenvío se reduce considerablemente. Ello se debe a que las entradas de la estación de reserva solo tienen que almacenar los identificadores de los registros y no sus valores, lo que consume muchos menos bytes. Por otra parte, la reducción de los buses se produce debido a que la estación únicamente necesita leer identificadores, no valores. Un inconveniente de este modo de planificación es que el tiempo que transcurre desde la emisión de la instrucción hasta que comienza su ejecución es mayor debido al tiempo necesario para extraer los operandos.

La segunda forma de efectuar la lectura de los operandos es cuando la instrucción se distribuye desde el buffer de distribución a las estaciones de reserva. A este estilo de captación de los operandos fuente se le denomina *planificación con lectura de operandos* (Figura 2.34.b). El código de operación y el identificador del registro destino se copian directamente en la estación de reserva desde el buffer de distribución mientras que los identificadores de los operandos fuente se envían al fichero de registros. Si el operando está disponible en el fichero de registros, la estación de reserva recibe el valor y coloca el bit de validez de su entrada a 1. Por el contrario, si el registro no está disponible se reenvía el identificador a la estación y se coloca el bit de validez a 0.

Ahora, tras la emisión de una instrucción, cuando la unidad funcional concluye y publica en el bus de reenvío el resultado de su operación junto con el identificador del registro destino, el resultado se copia tanto en el fichero de registros como en algunas entradas de la estación de reserva. En el fichero de registros no solo se actualiza el valor del registro sino que también se modifica el bit de validez para indicar que ya está disponible y no es el registro destino de ninguna otra instrucción posterior. Por su parte, las entradas de la estación de reserva que presentan alguna coincidencia de los identificadores de los operandos no disponibles con el identificador publicado en el bus de reenvío, reemplazan el identificador por el valor del operando y colocan el bit de validez a 1.

En la descripción del funcionamiento de las estaciones de reserva se ha recurrido al identificador de los registros para detectar la disponibilidad de un operando y así proceder a la activación de las instrucciones. Esto no es obligatorio y existen otras formas de realizarlo. Por ejemplo, el algoritmo

de Tomasulo recurre a los identificadores de las entradas de las estaciones de reserva. Las entradas de las estaciones de reserva tienen una etiqueta por cada operando fuente que indica si el operando está disponible. Si lo está, la etiqueta se coloca a 0 y si no lo está, se almacena el identificador de la entrada que almacena la instrucción que producirá el resultado. Tras la ejecución de la instrucción en la unidad funcional, se publica el identificador de la entrada en el bus de reenvío para que las estaciones de reserva que detecten coincidencias se actualicen.

La Figura 2.35 presenta un ejemplo de planificación con lectura de operandos. Se ha considerado que la estación de reserva centralizada distribuye 4 instrucciones/ciclo a las dos estaciones de reserva individuales que alimentan una unidad funcional de suma/resta (1 ciclo) y una unidad segmentada de multiplicación/división (2 ciclos). Las estaciones de reserva individuales emiten un máximo de una instrucción por ciclo de reloj a la unidad funcional que tienen asignada. Inicialmente, todos los registros son válidos. La figura no solo detalla el estado de las tres estaciones de reserva y del fichero de registros al comienzo de cada ciclo, sino también su estado al final de alguno de los ciclos. Esta precisión se realiza dado que, en este ejemplo, cuando una unidad funcional produce el resultado, éste junto con el identificador de su registro destino se publican en el buffer de reenvío y se actualizan las estaciones de reserva y el fichero de registros. De esta forma, las estaciones de reserva son capaces de emitir una instrucción en el ciclo siguiente a la publicación del resultado en el buffer. A continuación, se analiza ciclo a ciclo lo que sucede en el núcleo del procesador:

- Ciclo *i*. Se reciben las cinco instrucciones desde la etapa de decodificación.

- Ciclo $i+1$. Se distribuyen las instrucciones `i1`, `i2`, `i3` e `i4` a las estaciones de reserva individuales según su tipo, quedando en espera la `i5` para su distribución en el ciclo $i+2$. Se establecen los bits de validez de los registros `R1`, `R4` y `R5` a 0 en el fichero de registros dado que son destinatarios de los resultados de las instrucciones distribuidas. En las estaciones de reserva se almacenan los valores de los operandos disponibles y se establecen sus bits de validez a 1. Las instrucciones `i1` e `i4` se marcan como listas para ser emitidas al tener disponibles todos sus operandos.

- Ciclo $i+2$ (inicio). Se emiten `i1` e `i4`; se recibe `i5` de la estación de reserva centralizada y se coloca el bit de validez de `R6` a 0 en el fichero de registros.

- Ciclo $i+2$ (final). Concluye la ejecución de `i1` en la unidad de suma/resta y se publica su resultado y el identificador de su registro destino, `R1`, en el buffer de reenvío. Inmediatamente, se actualizan el bit de validez de `R1` (pasa a 1) y su contenido en el fichero de registros. También se actualizan las entradas de las estaciones de reserva que mantienen el operando `R1` marcado como no disponible. Como resultado de esta actualización, las instrucciones `i2` e `i3` se marcan como listas para ser emitidas al tener ya disponibles sus dos operandos.

- Ciclo $i+3$ (inicio). Se emiten `i2` e `i3`. En la segmentación de la unidad funcional de multiplicación/división coinciden dos instrucciones: la `i4` y la `i2`.

i1: SUB R1, R7, R0
i2: MULT R5, R1, R7
i3: ADD R5, R2, R1
i4: DIV R4, R3, R7
i5: MULT R6, R4, R5

(a) Secuencia de instrucciones

Ciclo i:

	O	CO	Op1	Op2	D
i5	1	MULT	R4	R5	R6
i4	1	DIV	R3	R7	R4
i3	1	ADD	R2	R1	R5
i2	1	MULT	R1	R7	R5
i1	1	SUB	R7	R0	R1

	Datos	V
R0	0	1
R1	1	1
R2	2	1
R3	35	1
R4	4	1
R5	5	1
R6	6	1
R7	7	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

Ciclo i+1: Distribución de i1, i2, i3 e i4

	O	CO	Op1	Op2	D
	0				
	0				
	0				
	0				
i5	1	MULT	R4	R5	R6

	Datos	V
R0	0	1
R1	1	0
R2	2	1
R3	35	1
R4	4	0
R5	5	0
R6	6	1
R7	7	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
i3	1	ADD	2	1	R1	0	R5	0
i1	1	SUB	7	1	0	1	R1	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
i4	1	DIV	35	1	7	1	R4	1
i2	1	MULT	R1	0	7	1	R5	0

Figura 2.35: Ejemplo de aplicación de la planificación con lectura de operandos a una secuencia de 5 instrucciones (continúa).

Ciclo i+2: Se ejecutan i1 e i4

	O	CO	Op1	Op2	D
	0				
	0				
	0				
	0				
	0				

	Datos	V
R0	0	1
R1	1	0
R2	2	1
R3	35	1
R4	4	0
R5	5	0
R6	6	0
R7	7	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
i3	1	ADD	2	1	R1	0	R5	0

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
i5	1	MULT	R4	0	R5	0	R6	0
i2	1	MULT	R1	0	7	1	R5	0

Final ciclo i+2: Finalización de i1. Quedan disponibles i2 e i3

	O	CO	Op1	Op2	D
	0				
	0				
	0				
	0				
	0				

	Datos	V
R0	0	1
R1	7	1
R2	2	1
R3	35	1
R4	4	0
R5	5	0
R6	6	0
R7	7	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
i3	1	ADD	2	1	7	1	R5	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
i5	1	MULT	R4	0	R5	0	R6	0
i2	1	MULT	7	1	7	1	R5	1

Figura 2.35: [Continuación] Ejemplo de aplicación de la planificación con lectura de operandos a una secuencia de 5 instrucciones (continúa).

Ciclo i+3: Se emiten i2 e i3

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

	Datos	V
R0	0	1
R1	7	1
R2	2	1
R3	35	1
R4	4	0
R5	5	0
R6	6	0
R7	7	1

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
i5 1	MULT	R4	0	R5	0	R6	0

Final ciclo i+3: Finalización de i3 e i4. Queda disponible i5

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

	Datos	V
R0	0	1
R1	7	1
R2	2	1
R3	35	1
R4	5	1
R5	9	1
R6	6	0
R7	7	1

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
i5 1	MULT	5	1	9	1	R6	1

Figura 2.35: [Continuación] Ejemplo de aplicación de la planificación con lectura de operandos a una secuencia de 5 instrucciones (continúa).

Ciclo i+4: Se emite i5

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

	Datos	V
R0	0	1
R1	7	1
R2	2	1
R3	35	1
R4	5	1
R5	9	1
R6	6	0
R7	7	1

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

Final ciclo i+4: Finalización de i2

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

	Datos	V
R0	0	1
R1	7	1
R2	2	1
R3	35	1
R4	5	1
R5	49	1
R6	6	0
R7	7	1

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

O	CO	Op1	V1	Op2	V2	D	L
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--
0	---	---	--	---	--	---	--

Figura 2.35: [Continuación] Ejemplo de aplicación de la planificación con lectura de operandos a una secuencia de 5 instrucciones (continúa).

Ciclo i+5: Continúa el procesamiento de i5

	O	CO	Op1	Op2	D
	0				
	0				
	0				
	0				
	0				

	Datos	V
R0	0	1
R1	7	1
R2	2	1
R3	35	1
R4	5	1
R5	49	1
R6	6	0
R7	7	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

Final ciclo i+5: Finalización de i5

	O	CO	Op1	Op2	D
	0				
	0				
	0				
	0				
	0				

	Datos	V
R0	0	1
R1	7	1
R2	2	1
R3	35	1
R4	5	1
R5	49	1
R6	45	1
R7	7	1

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

	O	CO	Op1	V1	Op2	V2	D	L
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--
	0	---	---	--	---	--	---	--

Figura 2.35: [Continuación] Ejemplo de aplicación de la planificación con lectura de operandos a una secuencia de 5 instrucciones.

- Ciclo $i + 3$ (final). Concluye la ejecución de las instrucciones `i3` e `i4`. Como consecuencia de ello se actualizan los bits de validez y los valores de `R4` y `R5` en el fichero de registros. También se actualiza la entrada de la estación de reserva que corresponde a la instrucción `i5`, que queda lista para ser emitida en el ciclo siguiente.
- Ciclo $i + 4$ (inicio). Se emite la instrucción `i5`. Las estaciones de reserva quedan vacías.
- Ciclo $i + 4$ (final). Concluye el procesamiento de `i2`. Se actualiza el valor y el bit de validez del registro `R5` en el fichero de registros.
- Ciclo $i + 5$ (inicio). La instrucción `i5` comienza su procesamiento en la segunda etapa de la segmentación de la unidad de multiplicación/división.
- Ciclo $i + 5$ (final). `i5` concluye su procesamiento en la unidad funcional y se publica el resultado en el buffer de reenvío. Se actualiza el bit de validez y el valor del registro `R5`.

Observe que el registro `R5` es el registro destino de las instrucciones `i2` e `i3` lo que da lugar a la existencia de dependencias WAW. Dado que es posible ejecutar instrucciones fuera de orden, la instrucción `i2` concluye después que la `i3`. Esto produce una violación de la dependencia WAW existente entre ellas dos ya que la instrucción `i2` sobrescribe el resultado de la `i3`. También hay que percatarse que tras la ejecución de `i3`, el bit de validez de `R5` indica que el registro está ya disponible, cuando en realidad no es así ya que `i3` también tiene que escribir sus resultados en ese registro. Estos problemas se solucionarán mediante la técnica conocida como renombramiento de registros.

2.7.4. Renombramiento de registros

La ejecución fuera de orden de instrucciones es una de las características de los procesadores superescalares que permiten maximizar su rendimiento, pero introduce nuevos problemas: la gestión de las dependencias falsas WAR (antidependencias) y WAW (dependencias de salida). Las dependencias falsas son consecuencia de la necesidad de reutilizar los registros accesibles por el repertorio de instrucciones para efectuar el almacenamiento temporal de resultados, ya que si el ISA dispusiese de un número infinito de registros, este tipo de dependencias de datos no existiría. Sin embargo, los registros accesibles al programador son un recurso limitado y se les conoce como *fichero de registros arquitectónicos* o *fichero de registros creados*. La mayor parte de estos registros son accesibles por el programador a través del repertorio de instrucciones, pero otros no (principalmente, registros que almacenan estados internos del procesador).

Para entender la necesidad de reutilizar o reciclar registros hay que conocer cómo el compilador finaliza la generación del código ensamblador. Para generar el código objeto, el compilador recurre a un número ilimitado de registros simbólicos en los que realiza almacenamientos temporales, manteniendo tantos datos como sea posible en los registros, minimizando así los accesos al sistema de memoria dado que consumen muchos ciclos de máquina. Sin embargo, ya que el número de registros arquitectónicos está limitado, para generar el código definitivo el compilador tiene que realizar una asignación

del conjunto infinito de registros simbólicos al conjunto finito de registros arquitectónicos (una correspondencia de muchos a uno). Esto obliga al compilador a reutilizar registros arquitectónicos en base a la siguiente regla: se escribe un nuevo valor en un registro cuando se detecta que el valor almacenado en el registro ya no es necesario para operaciones futuras, es decir, su validez ha caducado. El siguiente fragmento de código muestra un ejemplo de reciclaje al detectar el compilador que, a partir de la lectura de R1 en i2, el valor almacenado en este registro ya no se utiliza por lo que se reutiliza R1 en i3 para escribir un nuevo resultado:

```
i1: ADDI  R1,R0,#10
....................
i2: MULTI R2,R1,#40
i3: ADDI  R1,R0,#20
....................
i4: MULTI R3,R1,#80
```

El tiempo durante el que un dato almacenado en un registro es válido se denomina *rango de vida del registro*. Es el tiempo desde que se almacena el valor en el registro hasta que se hace uso de ese valor por última vez, antes de reemplazarlo mediante una nueva escritura. En el fragmento anterior, hay dos rangos de vida de R1, el que va desde i1 hasta i2 y el que va desde i3 hasta i4 (Figura 2.36.a). Dentro de un rango de vida se pueden efectuar múltiples lecturas del valor (en el ejemplo, múltiples lecturas de R1) ya que esas operaciones no le quitan validez al dato. El final de un rango de vida y el comienzo de otro rango de vida del mismo registro se produce debido a que se realiza el reciclaje de ese registro. Evidentemente, rangos de vida de registros diferentes se pueden intercambiar y mezclar siempre que se respeten las dependencias verdaderas.

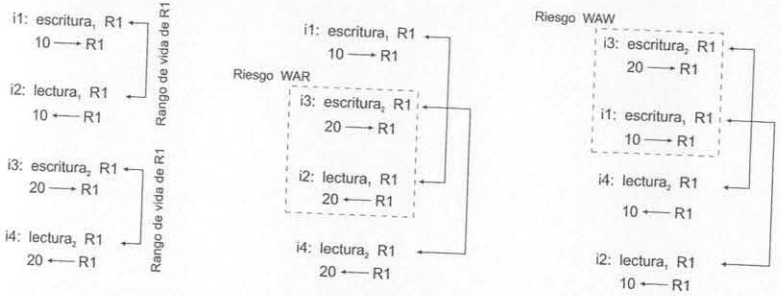


Figura 2.36: Rangos de vida de un registro. Los diferentes posibles solapamientos de dos rangos de vida de un mismo registro inducen diferentes tipos de riesgos.

En el fragmento anterior de código se dan los tres tipos de dependencias de datos. Cada rango de vida conforma por sí mismo una dependencia verdadera RAW ya que i2 e i4 dependen, respectivamente, de i1 e i3 para leer el valor correcto de R1. Entre i1 e i3 existe una dependencia WAW, mientras que entre i2 e i3 existe una dependencia WAR ya que la lectura de R1 por parte de i2 debe realizarse antes que la escritura de i3. En las Figuras 2.36.b y 2.36.c se aprecia que las violaciones de las dependencias falsas se producen por el solapamiento de los rangos de vida de un mismo registro. Estos solapamientos de los rangos de vida se pueden presentar en un procesador superescalar debido a la capacidad de ejecutar instrucciones fuera de orden.

La Figura 2.37.a presenta un fragmento de código en el que la ejecución fuera de orden de las instrucciones provoca el solapamiento de los dos rangos de vida del registro R1. Se considera que hay una unidad funcional de suma/resta (1 ciclo) y una unidad de multiplicación/división (2 ciclos) y que en el mismo ciclo en que se obtiene un resultado se actualizan los bits de validez. Al depender i3 de un operando de i2, se produce un retraso en su emisión hasta que i2 concluya y publique el valor de R5. Además, i4 depende de i3 por lo que también queda retenida hasta que el registro R1 se marque como disponible. Esta situación permite que la única instrucción de suma con todos sus operandos disponibles, la i5, se emita y se ejecute, lo que posibilita la posterior ejecución de i4 e i6 al validarse R1. La instrucción i4 se ejecuta antes que i6 ya que el planificador detecta que es más antigua puesto que está antes en la secuencia. La ejecución de la instrucción i4 dentro del rango de vida definido por i5-i6 introduce un riesgo WAR ya que se produce un solapamiento de los rangos. Además, la ejecución tardía de i3 introduce un riesgo WAW al permanecer el valor escrito por i3 en R1, cuando lo semánticamente correcto sería que permaneciese el valor de R1 dejado por i6.

Una solución inmediata y sencilla para respetar todas las dependencias de datos es la ejecución secuencial de las instrucciones y la escritura ordenada de los registros que hacen de operandos destino, es decir, el mantenimiento de los rangos de vida. Sin embargo, esto va en contra de una de las señas

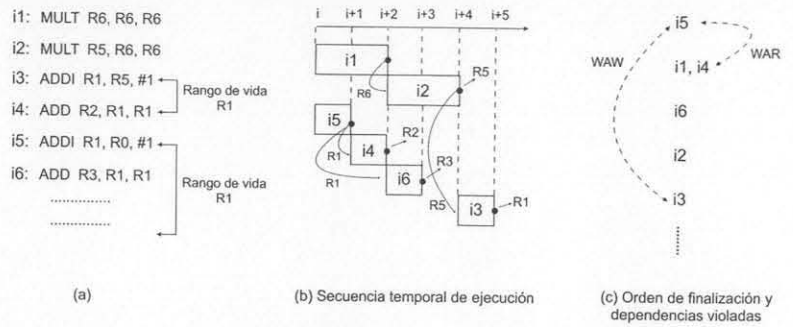


Figura 2.37: Violación de dependencias WAR y WAW como consecuencia de la ejecución fuera de orden.

de identidad de los procesadores superescalares: la ejecución de instrucciones fuera de orden. Otra solución es detener aquellas instrucciones dependientes hasta que la instrucción inicial haya terminado de acceder al registro dependiente. La detención de las instrucciones dependientes es obligada para respetar las dependencias verdaderas ya que hay una auténtica relación productor-consumidor, no así en las dependencias falsas donde el problema original surge por el reciclaje de registros, no por la semántica del programa. Es decir, no es necesario detener el cauce para cumplir las dependencias falsas de datos ya que el problema está ocasionado por la falta de espacio para realizar el almacenamiento temporal de resultados.

La solución adoptada para resolver las dependencias falsas de datos es el *renombramiento dinámico de los registros de la arquitectura mediante hardware*. Hasta ahora, los registros que se han utilizado en los ejemplos previos eran los registros arquitectónicos. El renombramiento de registros consiste en utilizar un conjunto de registros auxiliares, invisibles al programador, de forma que se reestablezca la correspondencia única entre resultados temporales y registros. Estos registros se conocen como *registros físicos*, *registros no creados* o *de renombramiento*. Así, las instrucciones escriben sus resultados en los registros no creados para, posteriormente, deshacer el renombramiento y proceder a la escritura ordenada de los registros arquitectónicos (los que ve el programador) utilizando los valores de los registros no creados. Este reestablecimiento de la correspondencia única entre resultados temporales y registros permite eliminar todas las dependencias falsas entre las instrucciones emitidas.

Antes de describir cómo se articula el renombramiento dinámico de registros mediante hardware, hay que precisar que el proceso consta de dos pasos:

- Paso 1: *Resolución de los riesgos WAW y WAR*. Se renombran de forma única los operandos destino de las instrucciones. Se resuelven así las dependencias WAW y WAR.
- Paso 2: *Mantenimiento de las dependencias RAW*. Se renombran todos los registros fuente que son objeto de una escritura previa utilizando el mismo nombre que se empleó en el paso 1 para renombrar el registro destino de la escritura previa. El objeto de este paso es respetar las dependencias RAW, las cuales quedan determinadas al establecer un especificador de registro común entre las instrucciones que mantienen una relación productor-consumidor.

El siguiente fragmento de código muestra la aplicación sucesiva de los dos pasos. El registro `R0` no necesita renombrarse ya que es un operando fuente que no es objeto de escritura previa. Tras los dos pasos, el reciclaje del registro `R1` realizado por el compilador ha quedado deshecho. Ahora, y siempre que se respeten las dependencias RAW, es factible alterar el orden de ejecución de las instrucciones sin consecuencias.

Código original

```
ADDI  R1,R0,#1
MULT  R2,R1,#4
ADDI  R1,R0,#2
MULTI R3,R1,#8
```

Paso 1

```
ADDI  Rr1,R0,#1
MULTI Rr2,R1,#4
ADDI  Rr3,R0,#2
MULTI Rr4,R1,#8
```

Paso 2

```
ADDI  Rr1,R0,#1
MULTI Rr2,Rr1,#4
ADDI  Rr3,R0,#2
MULTI Rr4,Rr3,#8
```


Esta forma de renombrar los registros mediante dos pasos es ilustrativa y didáctica pero el hardware no la efectúa así. En realidad, el procesador realiza el renombramiento instrucción tras instrucción, analizando si los dos operandos fuente, renombrados o no, están disponibles y efectuando el renombramiento del registro destino. Para realizar el renombramiento del registro destino y la lectura de los operandos fuente se recurre a hardware adicional que trabaja en estrecha colaboración con el tradicional fichero de registros arquitectónicos. La fase de renombramiento se puede realizar en la etapa de decodificación o en la de distribución. En este texto se considera que es parte de la etapa de distribución y que se efectúa cuando se distribuye una instrucción desde la estación de reserva centralizada a una estación de reserva individual.

El renombramiento dinámico se realiza incluyendo en el procesador un nuevo fichero de registros, denominado, *fichero de registros de renombramiento* (RRF - *Rename Register File*) o *buffer de renombramiento*. A partir de ahora, el fichero de registros de la arquitectura se denominará ARF (*Architected Register File*) para diferenciarlo del fichero de registros renombrados RRF. Existen tres formas de organizar el RRF: como un único fichero de registros formado por la suma del RRF y del ARF, como una estructura independiente pero accesible desde el ARF, o como parte del buffer de reordenamiento y accesible desde el ARF.

2.7.4.1. Organización independiente del RRF con acceso indexado

Un esquema de la implementación del RRF como una estructura independiente del ARF se muestra en la Figura 2.38. Las entradas del ARF se componen de tres campos: el campo *Datos* que contiene el valor del registro, el campo *Ocupado* que indica si el registro ha sido renombrado y un campo *Índice* que apunta a la entrada del RRF que corresponde al último renombramiento realizado.

ARF (16 entradas)

	Datos	Índice	Ocupado
R0	45	2	1
R1	15	1	0
R2	7	0	1
R3			
	⋮	⋮	⋮
R15			

RRF (8 entradas)

	Datos	Válido	Ocupado
Rr0	37	1	1
Rr1	15	1	0
Rr2	10	0	1
Rr3			
	⋮	⋮	⋮
Rr7			

Figura 2.38: Organización del RRF como estructura independiente del ARF con acceso indexado.

En lo que respecta a la estructura de los registros del RRF, estos constan de un campo *Datos*, un campo *Válido* y un campo *Ocupado*, estos dos últimos de un bit de longitud. En el campo *Datos* se

escribe el resultado de la instrucción que ha causado el renombramiento, valor que, posteriormente, se utilizará para actualizar el contenido del registro arquitectónico que tiene asociado. El campo *Ocupado* se emplea para saber si el registro está siendo utilizado todavía por instrucciones pendientes de ejecución y no puede liberarse. El campo *Válido* indica que todavía no se ha realizado la escritura en el RRF (es el sustituto del campo *Válido* utilizado en el fichero de registros de los ejemplos anteriores).

Si se opta por lectura de los operandos al distribuir las instrucciones desde la estación de reserva centralizada a las estaciones individuales, la lectura de un registro puede encontrarse frente a tres situaciones:

- El registro del ARF no está pendiente de ninguna escritura. Su bit de *Ocupado* permanece a 0 indicando que no hay renombramiento. Se procede a la lectura del valor almacenado en el campo *Datos* de su entrada en el ARF.
- El registro del ARF es destinatario de una escritura por lo que ha sido renombrado, su bit de *Ocupado* está a 1 y el campo *Índice* contiene un puntero a una entrada del RRF. El puntero no es otra cosa que el identificador de uno de los registros de renombramiento del RRF. Una vez que se accede al RRF, se pueden plantear dos situaciones según el estado del registro de renombramiento:
  - Campo *Válido*=0. La actualización del contenido del registro RRF con el resultado de la instrucción que provocó el renombramiento está pendiente. Por lo tanto, el operando no está disponible y se procede a enviar el identificador del registro de renombramiento a la estación de reserva.
  - Campo *Válido*=1. El valor del registro de renombramiento con el resultado de la instrucción de escritura se ha actualizado. El operando está disponible por lo que se puede extraer su valor del RRF si una instrucción lo necesita como operando fuente.

El proceso de renombrar el registro destino de una instrucción cuando ésta se distribuye a la estación de reserva individual es relativamente sencillo. Primero se accede mediante su identificador a la entrada que le corresponde en el ARF, se establece el bit de *Ocupado* a 1, se selecciona un registro de renombramiento del RFF que esté libre y se copia el identificador del registro RRF seleccionado en el campo *Índice*. A su vez, el campo *Ocupado* del registro seleccionado en el RFF se marca a 1 y el campo *Válido* se establece a 0 ya que todavía no se ha realizado la escritura (la instrucción se está distribuyendo, no se ha llegado a emitir). El identificador del registro del RRF que ha sido seleccionado se utilizará como identificador de registro destino en la entrada de la estación de reserva individual y se almacenará en la entrada que la instrucción tiene en el buffer de terminación, de forma que cuando la instrucción sea terminada se sepa qué registro del RRF hay que liberar.

Una vez que una instrucción finaliza su ejecución, se produce la escritura del resultado de la operación en el registro RRF y la colocación del campo *Válido* a 1. Sin embargo, la escritura diferida del valor del RRF al ARF se efectúa cuando la instrucción termina, lo que sucederá cuando lo indique el buffer de terminación, que puede ser en el siguiente ciclo o muchos ciclos después. Como se verá más

adelante, la terminación de las instrucciones ya se efectúa de forma ordenada puesto que todas las instrucciones al abandonar la etapa de decodificación se insertaron ordenadamente en dos buffers: la estación de reserva centralizada y el buffer de reordenamiento.

La Figura 2.38 presenta las situaciones que se pueden plantear al realizar el renombramiento en función del estado de la instrucción de escritura que forzó el renombramiento:

- *Instrucción pendiente de escritura.* El registro `R0` del ARF está marcado como ocupado indicando que el registro se encuentra renombrado como consecuencia de una instrucción de escritura y, por ello, el contenido de *Datos* es información no actualizada. El valor de su índice apunta al registro `Rr2` del RRF que es el identificador que estará en las entradas de las estaciones de reserva que corresponden a aquellas instrucciones que tenían `R0` como operando fuente y eran posteriores a la instrucción de escritura que causó el renombramiento de `R0`. En el RRF el campo *Válido* se encuentra a 0 indicando que el resultado de la operación todavía no se ha obtenido y, por ello, el contenido de su campo *Datos* tampoco es válido. El campo *Ocupado* está a 1 indicando que la entrada está en uso. Observe que si ahora una nueva instrucción con `R0` como operando fuente se distribuyese a una estación de reserva individual, el identificador `R0` se reemplazaría por el identificador de `Rr2` ya que todavía no estaría disponible su valor actualizado (la instrucción de escritura se encuentra pendiente de emisión o está ejecutándose).
- *Instrucción finalizada.* El registro `R2` está marcado como ocupado lo que indica que una instrucción de escritura lo renombró, utilizando para ello el `Rr0` del RRF. El registro `Rr0` tiene su campo *Válido* a 1 para señalar que la actualización del registro se ha realizado, lo que es indicativo de la finalización de la ejecución de la instrucción de escritura. El campo *Ocupado* permanece a 1 ya que la actualización del registro `R2` del ARF todavía no se ha realizado como consecuencia de que la instrucción de escritura no ha sido terminada. El valor almacenado en el campo *Datos* del `R2` no tiene validez ya que está a falta de actualización, es decir, de la escritura del contenido del campo *Datos* del `Rr0` en el campo *Datos* del `R2`. Si durante el tiempo que `R2` permanece en este estado, una nueva instrucción de lectura con `R2` como operando fuente es distribuida, el identificador de `R2` será sustituido por el valor de `Rr0`.
- *Instrucción terminada.* El registro `R1` está marcado como no ocupado lo que indica que la instrucción de escritura que lo utilizó ya terminó y actualizó el contenido de su campo *Datos*, deshaciendo el renombramiento. Aunque el renombramiento de `R1` ya no exista, es interesante analizar el estado en que quedó el registro de renombramiento que se le asoció, el `Rr1`. Como la instrucción de escritura que lo renombró ya terminó, el contenido del campo *Datos* de `Rr1` coincide con el de `R1`. La indicación de que la instrucción de escritura terminó es que el campo *Ocupado* quedó marcado como 0. El campo *Válido* quedó a 1 ya que, previamente, a la terminación, la instrucción finalizó y escribió el resultado en el campo *Datos* de `Rr1`, colocando el bit de válido a 1. Cualquier nueva instrucción que se distribuyese a continuación y tuviese como operando fuente el registro `R1`, utilizaría el valor del campo *Datos* del ARF.

Tras estudiar el proceso de renombramiento de un registro utilizando un RRF independiente y con acceso indexado, surgen algunas preguntas. La primera de ellas es: ¿qué sucede si al renombrar un registro el campo *Ocupado* está ya asignado a 1, es decir, el registro arquitectónico se encuentra ya renombrado, tal y como podría ocurrir en el siguiente fragmento de código con el registro R1?

```
i1: ADDI  R1,R0,#1
i2: MULTI R2,R1,#4
i3: ADDI  R1,R0,#2
i4: MULTI R3,R1,#8
```

Si se analiza el funcionamiento del núcleo del procesador en su conjunto la respuesta es sencilla: simplemente se efectúa un nuevo renombramiento buscando un registro en el RRF que esté libre. Analizando el ejemplo anterior se entenderá mejor. En la instrucción `i1`, el registro `R1` se renombra a, por ejemplo, `Rr1`. Una vez que esté disponible el valor de `Rr1` debido a que `i1` haya finalizado su ejecución, la instrucción `i2` leerá el campo *Datos* de `Rr1` y estará en condiciones de ser emitida. Por otro lado, al distribuir `i3` hay que renombrar nuevamente el `R1` y se le asigna, por ejemplo, el `Rr3`. Observe que aunque `i3` finalice antes que `i2`, no sucede nada ya que `i2` utiliza como fuente el valor de `Rr1` producido por `i1`, mientras que `i3` ha escrito su resultado en el registro `Rr3` (tras finalizar). Además, cualquier instrucción situada entre `i2` e `i3` que utilizase `R1` como registro fuente tendría como operando fuente el registro de renombramiento `Rr1`. Finalmente, cuando se deshiciese el primer renombramiento de `R1` al terminar `i1`, el valor de `Rr1` se escribiría antes que el de `Rr3`, con lo que prevalecería la semántica del programa. Al analizar el buffer de terminación se verá en detalle cómo se deshacen los renombramientos de forma que no se interfiera con la lectura de los operandos.

Una segunda pregunta interesante es: ¿qué ocurre si tras terminar una instrucción de escritura y deshacer el renombramiento de su registro destino hay que distribuir una nueva instrucción cuyo operando fuente coincide con el destino de la escritura terminada? La instrucción de lectura leerá el valor del operando del ARF ya que en ese momento el registro ya no está renombrado (bit de ocupado a 0). Además, en la estaciones de reserva no quedarán instrucciones con el identificador del registro de renombramiento ya que, en el momento en que la instrucción de escritura finalizó, se reemplazó el identificador por el valor en las entradas en que hubo coincidencias. Por ello, el deshacer el renombramiento no provoca ningún efecto sobre las instrucciones almacenadas en las estaciones de reserva.

¿Y si la instrucción de lectura se distribuye mientras que la escritura que forzó el renombramiento ha finalizado pero no ha terminado? En ese caso, la instrucción de lectura se almacenará en la entrada de la estación de reserva utilizando como operando fuente el valor del registro de renombramiento, no su identificador, ya que la escritura finalizó y actualizó el valor del campo *Datos* del registro de renombramiento.

Por tanto, es muy importante comprender que la técnica del renombrado de registros implica que el contenido de los operandos fuente en las estaciones de reserva puede ser de tres tipos: el valor del registro original (no hay escritura pendiente), el identificador del registro de renombramiento (escritura pendiente de finalizar) o el valor del registro de renombramiento (escritura finalizada pero no terminada).

2.7.4.2. Organización independiente del RRF con acceso asociativo

Otra forma de organizar el RRF de forma independiente pero que introduce cambios en su estructura es si se accede de forma asociativa, es decir, mediante una búsqueda en el RRF del identificador del registro ARF que provoca el renombramiento. Ahora, el ARF no necesita disponer de un campo *Índice* ya que para acceder al registro de renombramiento se utiliza el identificador del registro destino ARF que provoca el renombrado.

La Figura 2.39 muestra un esquema de la organización independiente del RRF. El campo *Destino* del RRF es donde se almacena el identificador del registro del ARF que provoca el renombramiento y que se utiliza para realizar la búsqueda y validar la coincidencia. Los campos *Datos*, *Válido* y *Ocupado* tienen el mismo significado que en el RRF con acceso indexado. La diferencia fundamental está en el campo denominado *Último*. En el acceso indexado al RRF era posible conocer el último renombramiento vigente asociado a un registro del ARF ya que, con independencia de las veces que fuera renombrado, siempre quedaba almacenado en el campo *Índice* del ARF el identificador del último registro de renombramiento. Esta labor ahora la efectúa el campo *Último*, de forma que si está a 1 indica que es el último registro de renombramiento asignado al registro del ARF. Cada vez que se hace un nuevo renombramiento del mismo registro destino, el campo *Último* del último registro de renombramiento asignado a ese registro del ARF pasa de 1 a 0 y el del nuevo a 1. De esta forma, si llega una instrucción con un registro fuente que ha sido renombrado, mediante el campo *Último* del RRF puede determinar de entre los múltiples renombramientos cuál es el más actual asociado a ese registro fuente.

Las tres primeras entradas del ARF y del RRF de la Figura 2.39 replican las situaciones que se daban en la Figura 2.38. La cuarta entrada del ARF, la correspondiente al R3, indica que se encuentra renombrado por lo que hay que acceder a la RRF. Observe que la cuarta y quintas entradas del RRF son dos renombramientos del R3. De los dos registros renombrados, el `Rr4` es el que tiene el campo *Último* a 1 lo que indica que su renombramiento es más actual que el `Rr3`, que tiene *Último* a 0. Si en la situación

ARF (16 entradas)

	Datos	Ocupado
R0	45	1
R1	15	0
R2	7	1
R3	28	1
⋮	⋮	⋮
R15		

RRF (8 entradas)

	Destino	Datos	Válido	Último	Ocupado
Rr0	2	37	1	1	1
Rr1	1	15	1	1	0
Rr2	0	10	0	1	1
Rr3	3	35	1	0	1
Rr4	3	8	0	1	1
⋮	⋮	⋮	⋮	⋮	⋮
Rr7					

Figura 2.39: Organización del RRF como estructura independiente del ARF con acceso asociativo.

actual, una instrucción con operando fuente R3 al distribuirse accediese al ARF para leer el operando, obtendría que el registro está renombrado y al acceder al RRF tendría que emplear como operando fuente el Rr4, no el Rr3, ya que es el último renombramiento vigente y está pendiente de escritura.

2.7.4.3. Organización del RRF como parte del buffer de reordenamiento

Otra forma de acomodar el RRF en el núcleo del procesador es como parte del buffer de reordenamiento o terminación. Aunque se analizará posteriormente con mayor detalle, tras finalizar la ejecución de las instrucciones fuera de orden, el buffer de reordenamiento permite recuperar el orden de las instrucciones y concluir su procesamiento manteniendo la consistencia semántica del programa. Cada instrucción que sale de la etapa de decodificación se almacena simultáneamente en el buffer de distribución y en el buffer de terminación siguiendo el orden secuencial del programa. De esta forma, tras la etapa de ejecución no se pierde el conocimiento relativo al orden en que deben terminarse arquitectónicamente las instrucciones.

Para incluir el RRF en el buffer de reordenamiento hay que añadir dos campos a sus entradas: un campo *Datos* y un campo *Válido*, con significado similares a los ya vistos (Figura 2.40). La diferencia estriba en que ahora el campo *Índice* del ARF contiene un puntero a una de las entradas del buffer de reordenamiento, ya que el buffer hace de RRF. No hay que añadir un campo *Ocupado* ya que ese campo ya existe en el buffer de terminación puesto que se utiliza para saber si una entrada está libre o no cuando hay que almacenar una nueva instrucción.

ARF (16 entradas)

	Datos	Ocupado	Índice
R0	45	1	2
R1	15	0	1
R2	7	1	0
R3			
⋮	⋮	⋮	⋮
R15			

Buffer de reordenamiento (128 entradas)

	Datos	Válido	.....	.....	Ocupado
0	37	1			1
1	15	1			0
2	10	0			1
⋮	⋮	⋮	⋮	⋮	⋮
127					

Figura 2.40: Implementación del RRF en el buffer de reordenamiento.

De esta forma, si se produce un renombramiento de un registro por ser el operando destino de una instrucción, el campo *Índice* del registro apuntará a la entrada del buffer de reordenamiento que ocupa la instrucción causante del renombramiento de su operando destino. Ahora en la entrada de esa instrucción de la estación de reserva se almacenará en el campo de destino el identificador de la entrada del buffer de reordenamiento. Como en los casos anteriores, aunque haya dos o más renombramientos

de un mismo registro del ARF, la terminación ordenada de las instrucciones garantiza la actualización correcta del registro en el ARF. En el instante en que la instrucción termine y se libere su entrada en el buffer de terminación, se accederá con el identificador de esa entrada al ARF y si hubiese coincidencia se actualizaría el campo *Datos* y el bit de *Ocupado* se establecería a 0, dando por concluido el renombramiento.

Si al realizar la distribución de una instrucción se accede al ARF para efectuar la lectura de un operando, las situaciones que se pueden producir son similares a las ya analizadas en la organización independiente del RRF con acceso indexado o asociativo:

- Si el registro no está renombrado, se lee su valor del campo *Datos* del ARF y se utiliza como operando fuente en la estación de reserva.
- Si está renombrado, se sigue el puntero del campo *Índice* y se accede a una entrada del buffer de reordenamiento. Pueden acontecer dos situaciones:
  - Si el bit de *Válido* está a 0 es que se está a la espera de la finalización de la instrucción que tiene que actualizar el contenido del registro de renombramiento, es decir, el campo *Datos* de la entrada del buffer. Como operando fuente en las estaciones de reserva se utiliza el identificador de la entrada del buffer.
  - Si el bit de *Válido* está a 1 es que se está pendiente de la terminación de la instrucción. Como operando fuente para la estación de reserva se utiliza el valor almacenado en el campo *Datos* del buffer de terminación. En el instante en que la instrucción se termine, es decir, se actualice el contenido de *Datos* en el ARF con la información almacenada en el buffer, el bit de *Ocupado* se colocará a 0 tanto en el ARF como en el buffer de reordenamiento para indicar que el renombramiento del registro ha concluido.

Aunque la inclusión del RFF en el buffer de terminación parece que no aporta nada nuevo, no es así. Por un lado, esta organización provoca que haya que añadir un campo *Datos* y un campo *Válido* al buffer de terminación cuando existen instrucciones que no provocan renombramiento de registros (por ejemplo, un salto incondicional). Por otro lado, y debido a las funciones que realiza el buffer de terminación, éste ya dispone del hardware y de los puertos necesarios para intercambiar información con las unidades funcionales y con el ARF. Sin embargo, una estructura independiente para ubicar el RRF implica añadir una nueva unidad hardware al procesador con todo lo que conlleva en el diseño lógico.

Lo realmente importante al realizar el diseño del RRF es alcanzar una elevada velocidad de renombramiento. Esto garantiza que los renombramientos de los registros destino y las lecturas de los registros fuente se realicen en el menor tiempo posible, lo que influye de forma notoria en el rendimiento de la etapa distribución, ejecución y terminación. Recuerde que para poder distribuir una instrucción son necesarias tres condiciones: una entrada libre en el RRF, una entrada libre en la estación de reserva individual que le corresponda según su tipo y una entrada libre en el buffer de reordenamiento. En caso de que falle una de las tres condiciones, el envío de instrucciones desde la estación de reserva centralizada

a las estaciones individuales se detiene. Por ello, es fundamental que el diseño del RRF sea el más adecuado para las características del procesador.

2.7.5. Ejemplo de procesamiento de instrucciones con renombramiento

En este apartado se va a describir ciclo a ciclo el procesamiento de un conjunto de instrucciones en un sencillo núcleo de ejecución basado en planificación dinámica con lectura de operandos y renombramiento de registros mediante un RRF independiente con acceso indexado (Figura 2.41). La estación de reserva centralizada puede distribuir un máximo de 4 instrucciones/ciclo a las dos estaciones de reserva individuales que alimentan, respectivamente, una unidad funcional de suma/resta (1 ciclo) y una unidad segmentada de multiplicación/división (2 ciclos). Las estaciones de reserva individuales

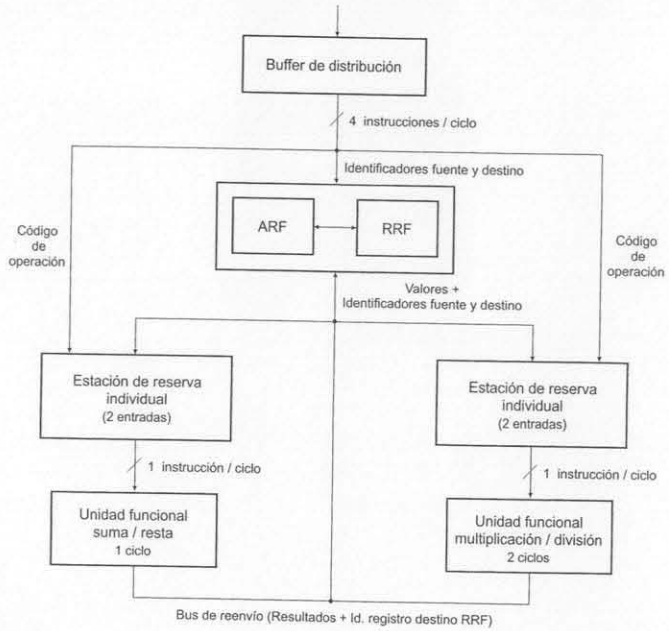


Figura 2.41: Núcleo de ejecución con planificación dinámica con lectura de operandos y RRF independiente con acceso indexado.

emiten a una velocidad de 1 instrucción/ciclo a la unidad funcional que tienen asignada. También se ha considerado que los accesos al ARF y al RRF se realizan en el mismo ciclo en que se distribuye la instrucción (el renombramiento no penaliza), que se pueden finalizar dos instrucciones simultáneamente y que las instrucciones son terminadas en el ciclo siguiente a su finalización.

El fragmento de código que se va a procesar consta de las siguientes instrucciones:

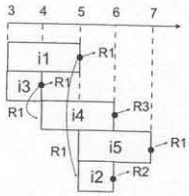
```
i1: MULT R1,R0,R3
i2: ADD  R2,R1,R0
i3: SUB  R1,R0,R4
i4: MULT R3,R1,R0
i5: MULT R1,R4,R0
```

En el código se puede apreciar la existencia de tres rangos de vida para el registro R1: el definido por i1-i2, el i3-i4 y el iniciado por la i5. Si no se realizase renombramiento de registros, la ejecución fuera de orden de las instrucciones produciría la violación de las dependencias falsas de datos existentes, como la WAR que hay entre i2 e i3 y la WAW entre i1 e i3. Gracias al renombramiento, esto no sucederá aunque, todavía, las instrucciones seguirán terminando fuera de orden. La Figura 2.42 muestra de forma gráfica la evolución ciclo a ciclo del procesamiento de las cinco instrucciones en el núcleo de ejecución.

- Ciclo 1. Las cinco instrucciones llegan de la etapa de decodificación y se almacenan en el buffer de distribución.
- Ciclo 2. Se produce el renombramiento y distribución de las instrucciones i1, i2, i3 e i4 a las dos estaciones de reserva individuales. Observe cómo se ha realizado el renombramiento de los registros destino, en concreto, preste atención al doble renombramiento de R1 al ser operando destino en las instrucciones i1 e i3. La instrucción i1 provoca que R1 se renombre como Rr0, la instrucción i2 fuerza a que R2 se renombre como Rr1, y la i3 origina el segundo renombramiento de R1 que pasa a apuntar en su campo *Índice* a Rr2. Aunque Rr0 haya perdido su conexión con R0 en el ARF, esto no tendrá consecuencias ya que el valor que deberá prevalecer en R1, según el orden del programa y a falta de procesar i5, es el generado por i3. En todos los registros de renombramiento los bits de *Válido* se han fijado a 0 para indicar que están pendientes de escritura y los bits de *Ocupado* a 1, al igual que en los registros del ARF.

  Tras el renombramiento, las estaciones de reserva reciben las instrucciones con los valores de los operandos fuente que estén disponibles o con los identificadores de los registros del RRF. La instrucción i1 ya tiene todos sus operandos disponibles por lo que podrá emitirse en el ciclo siguiente. La i2 está a la espera de valor que se almacenará en el registro Rr0 (el resultado de i1). La i3 también cuenta con todos los operandos por lo que está en condiciones de emitirse en el siguiente ciclo, almacenándose su resultado en Rr2. La i4 está a la espera del valor que se almacenará en Rr2 para poder emitirse.
- Ciclo 3 (inicio). Ya que ambas unidades funcionales están libres, las instrucciones i1 e i3 se emiten. La emisión de i1 e i3 provoca que queden entradas libres en las estaciones de reserva,

i1: MULT R1, R0, R3
i2: ADD R2, R1, R0
i3: SUB R1, R0, R4
i4: MULT R3, R1, R0
i5: MULT R1, R4, R0



(a) Secuencia de instrucciones

Ciclo 1: Recepción de las instrucciones del decodificador

	O	CO	Op1	Op2	D
i5	1	MULT	R4	R0	R1
i4	1	MULT	R1	R0	R3
i3	1	SUB	R0	R4	R1
i2	1	ADD	R1	R0	R2
i1	1	MULT	R0	R3	R1

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10		0
R2	20		0
R3	30		0
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0			0
Rr1			0
Rr2			0
Rr3			0
Rr4			0

	O	CO	Op1	v1	Op2	v2	D	L
	0							
	0							

	O	CO	Op1	v1	Op2	v2	D	L
	0							
	0							

Ciclo 2: Distribución de i1, i2, i3, i4 y renombramiento de R1, R2 y R3

	O	CO	Op1	Op2	D
	0				
	0				
	0				
	0				
i5	0	MULT	R4	R0	R1

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr2	1
R2	20	Rr1	1
R3	30	Rr3	1
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0		0	1
Rr1		0	1
Rr2		0	1
Rr3		0	1
Rr4			0

	O	CO	Op1	v1	Op2	v2	D	L
i3	1	SUB	5	1	40	1	Rr2	1
i2	1	ADD	Rr0	0	5	1	Rr1	0

	O	CO	Op1	v1	Op2	v2	D	L
i4	1	MULT	Rr2	0	5	1	Rr3	0
i1	1	MULT	5	1	30	1	Rr0	1

Figura 2.42: Evolución del procesamiento de instrucciones empleando RRF (continúa).

Ciclo 3 (inicio): Se emiten i1 e i3. Se distribuye i5 y se renombra R1 por tercera vez

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	20	Rr1	1
R3	30	Rr3	1
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0		0	1
Rr1		0	1
Rr2		0	1
Rr3		0	1
Rr4		0	1

	O	CO	Op1	v1	Op2	v2	D	L
	0							
i2	1	ADD	Rr0	0	5	1	Rr1	0

	O	CO	Op1	v1	Op2	v2	D	L
i5	1	MULT	40	1	5	1	Rr4	1
i4	1	MULT	Rr2	0	5	1	Rr3	0

Ciclo 3 (final): Concluye i3 y se publica su resultado (-35) y su destino (Rr2)

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	20	Rr1	1
R3	30	Rr3	1
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0		0	1
Rr1		0	1
Rr2	-35	1	1
Rr3		0	1
Rr4		0	1

	O	CO	Op1	v1	Op2	v2	D	L
	0							
i2	1	ADD	Rr0	0	5	1	Rr1	0

	O	CO	Op1	v1	Op2	v2	D	L
i5	1	MULT	40	1	5	1	Rr4	1
i4	1	MULT	-35	1	5	1	Rr3	1

Figura 2.42: [Continuación] Evolución del procesamiento de instrucciones empleando RRF (continúa).

Ciclo 4 (inicio): Se emite i4. i1 se encuentra en el 2º ciclo. Se libera Rr2 por terminación de i3

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	20	Rr1	1
R3	30	Rr3	1
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0		0	1
Rr1		0	1
Rr2	-35	1	0
Rr3		0	1
Rr4		0	1

	O	CO	Op1	v1	Op2	v2	D	L
	0							
i2	1	ADD	Rr0	0	5	1	Rr1	0

	O	CO	Op1	v1	Op2	v2	D	L
i5	1	MULT	40	1	5	1	Rr4	1

Ciclo 4 (final): Finaliza i1. Se publica su resultado (150) y su destino (Rr0)

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	20	Rr1	1
R3	30	Rr3	1
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0	150	1	1
Rr1		0	1
Rr2	-35	1	0
Rr3		0	1
Rr4		0	1

	O	CO	Op1	v1	Op2	v2	D	L
	0							
i2	1	ADD	150	1	5	1	Rr1	1

	O	CO	Op1	v1	Op2	v2	D	L
	0							
i5	1	MULT	40	1	5	1	Rr4	1

Figura 2.42: [Continuación] Evolución del procesamiento de instrucciones empleando RRF (continúa).

Ciclo 5 (inicio): Se emiten i2 e i5. i4 se encuentra en el 2º ciclo. Se libera Rr0 por terminación de i1

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	20	Rr1	1
R3	30	Rr3	1
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0	150	1	0
Rr1		0	1
Rr2	-35	1	0
Rr3		0	1
Rr4		0	1

O	CO	Op1	v1	Op2	v2	D	L
0							
0							

O	CO	Op1	v1	Op2	v2	D	L
0							
0							

Ciclo 5 (final): Finaliza i4 y se publica su resultado (-175) y su destino (Rr3)
Finaliza i2 y se publica su resultado (155) y su destino (Rr1)

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	20	Rr1	1
R3	30	Rr3	1
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0	150	1	0
Rr1	155	1	1
Rr2	-35	1	0
Rr3	-175	1	1
Rr4		0	1

O	CO	Op1	v1	Op2	v2	D	L

O	CO	Op1	v1	Op2	v2	D	L

Figura 2.42: [Continuación] Evolución del procesamiento de instrucciones empleando RRF (continúa).

Ciclo 6 (inicio): Termina i2, se actualiza R2 y se libera Rr1
Termina i4, se actualiza R3 y se libera Rr3
i_5 se encuentra en el 2º ciclo

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	155	Rr1	0
R3	-175	Rr3	0
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0	150	1	0
Rr1	155	1	0
Rr2	-35	1	0
Rr3	-175	1	0
Rr4		0	1

O	CO	Op1	v1	Op2	v2	D	L

O	CO	Op1	v1	Op2	v2	D	L

Ciclo 6 (final): Finaliza i5, se publica su resultado (200) y su destino (Rr4)

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	10	Rr4	1
R2	155	Rr1	0
R3	-175	Rr3	0
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0	150	1	0
Rr1	155	1	0
Rr2	-35	1	0
Rr3	-175	1	0
Rr4	200	1	1

O	CO	Op1	v1	Op2	v2	D	L

O	CO	Op1	v1	Op2	v2	D	L

Figura 2.42: [Continuación] Evolución del procesamiento de instrucciones empleando RRF (continúa).

Ciclo 7 (inicio): Termina i5, se actualiza R1 y se libera Rr4

O	CO	Op1	Op2	D
0				
0				
0				
0				
0				

ARF

	Datos	Índice	Ocupado
R0	5		0
R1	200	Rr4	0
R2	155	Rr1	0
R3	-175	Rr3	0
R4	40		0

RRF

	Datos	Válido	Ocupado
Rr0	150	1	0
Rr1	155	1	0
Rr2	-35	1	0
Rr3	-175	1	0
Rr4	200	1	0

O	CO	Op1	v1	Op2	v2	D	L

O	CO	Op1	v1	Op2	v2	D	L

Figura 2.42: [Continuación] Evolución del procesamiento de instrucciones empleando RRF.

por lo que se distribuye la `i5`. La distribución de `i5` produce un tercer renombramiento de `R1`, que ahora se renombra como `Rr4`, con lo que ya todos los registros del RRF están en uso: cinco registros destino, cinco entradas del RRF.

- Ciclo 3 (final). Finaliza la ejecución de `i3` y se publica su resultado ($5-40=-35$) y el identificador del registro destino (`Rr2`) en el buffer de reenvío. Esto provoca la actualización de `Rr2` en el RRF: se copia el resultado en el campo *Datos* y se cambia el bit de *Válido* a 1. Al mismo tiempo, en la entrada de la estación de reserva correspondiente a `i4` se produce una coincidencia de operando con `Rr2`. Esto provoca que se reemplace en el campo `Op1` el identificador de `Rr2` por el resultado publicado y se marque `V1` con 1. Ahora `i4` ya tiene los dos operandos disponibles por lo que cuando el planificador lo considere oportuno podrá emitirse.

- Ciclo 4 (inicio). Tras finalizar su ejecución, la instrucción `i3` termina por lo que deshace el renombramiento de `Rr2` y lo libera. Observe que deshacer el renombramiento no tiene ninguna consecuencia en el ARF, ya que el `Rr2` perdió su conexión con `R1` debido al renombramiento provocado por `i5` (al inicio del ciclo 3). Además, antes de liberarse, el valor de `Rr2` se copió en las entradas en las que había coincidencia de identificador, esto es, en la entrada de la `i4` (final del ciclo 3). También se produce la emisión de `i4` al quedar lista al final del ciclo anterior por disponibilidad de operandos y poder admitir la unidad de multiplicación/división una segunda instrucción al encontrarse la instrucción `i1` en su segundo ciclo de procesamiento.

- Ciclo 4 (final). Finaliza el procesamiento de `i1` y se publica su resultado ($5*30=150$) y el identificador de su registro destino (`Rr0`) en el bus de reenvío. Esto provoca cambios en el RRF y en las estaciones de reserva. En el RRF se actualiza el registro `Rr0` con el resultado y se cambia su

bit de validez a 1. En las entradas de las estaciones de reserva en las que se produce coincidencia de operando con `Rr0` se reemplaza el identificador por el resultado (150). Esto sucede únicamente en la `i2`, que queda lista para ser emitida.

- Ciclo 5 (inicio). Se produce la emisión de `i2` a la unidad de suma/resta ya que está libre. Al estar `i4` en su segundo ciclo dentro de la unidad de multiplicación/división, también se emite `i5`. Simultáneamente se produce la liberación de `Rr0` al terminarse la instrucción `i1`. Salvo el cambio del bit de *Ocupado* de 1 a 0 en la entrada de `Rr0` en el RRF, su liberación no tiene ninguna otra consecuencia ya que su conexión con `R1` se perdió tras el renombramiento de `i3` (ciclo 2).

- Ciclo 5 (final). Se produce la finalización de `i4`, la publicación de su resultado ($-35 * 5 = -175$) y del identificador de su registro destino (`Rr3`). También finaliza `i2` y se publica su resultado (155) y el identificador de su destino (`Rr1`). La consecuencia de estas dos finalizaciones es la actualización de las entradas de `Rr1` y `Rr3` en el RRF, tanto en lo que respecta a sus valores como a sus bits de ocupado. No hay coincidencias en las estaciones de reserva al no haber instrucciones pendientes de emisión.

- Ciclo 6 (inicio). Termina la instrucción `i2`, se actualiza `R2` dado que el renombramiento con `Rr1` se encuentra vigente, y tras la escritura diferida se libera `Rr1`. También termina `i4`, actualizándose el valor de `R3` y liberándose su registro de renombramiento `Rr3`. La instrucción `i5` se encuentra en la segunda etapa de la unidad de multiplicación/división.

- Ciclo 6 (final). Finaliza el procesamiento de `i5` lo que provoca la publicación de su resultado ($40 * 5 = 200$) y el identificador de su registro destino (`Rr4`). Consecuencia de ello es la actualización de la entrada de `Rr4` en el RRF.

- Ciclo 7 (inicio). Termina la instrucción `i5`, se actualiza el valor de `R1` al permanecer vigente el renombramiento con `Rr4` y se libera `Rr4`.

En el ejemplo anterior, pese a la ejecución fuera de orden, ninguna dependencia de datos ha sido violada pero, sin embargo, las instrucciones todavía terminan fuera de orden ya que no hay ningún mecanismo que reestablezca el orden original. Por ejemplo, la `i3` termina antes que la `i1`. La operativa establecida por el renombramiento de registros y las estaciones de reserva garantizan el cumplimiento de todas las dependencias de datos pero si surgiese una excepción o interrupción no podría tratarse correctamente. Para un correcto tratamiento de estas situaciones anómalas es necesario asegurar que el estado del procesador tras terminar una instrucción i_n sea el mismo que se tendría si todas las instrucciones anteriores a ella, i_{n-1}, i_{n-2}, etc. se hubiesen ejecutado ordenadamente una tras otra, incluida i_n. Una terminación desordenada viola esta condición.

2.8. Terminación

Una vez que las instrucciones han completado su etapa de ejecución, estas quedan almacenadas en el buffer de reordenamiento a la espera de su terminación arquitectónica. Formalmente, una instrucción se considera terminada arquitectónicamente cuando actualiza el estado del procesador manteniendo la *consistencia*. Se dice que hay *consistencia del procesador* cuando las instrucciones concluyen su procesamiento en el mismo orden secuencial en el que se encuentran en el programa, es decir, en el mismo orden en que iniciaron su procesamiento. El mantener la consistencia del programador es fundamental por dos razones, íntimamente interrelacionadas:

- Para garantizar el resultado final del programa, es decir, respetar la semántica del programa que viene definida por el orden secuencial de las instrucciones.

- Para permitir un tratamiento correcto de las interrupciones. Es necesario conocer con exactitud el estado en que se encuentra el procesador antes de procesar una instrucción.

Para todas aquellas instrucciones cuyo resultado final consiste en la escritura de un valor en un registro arquitectónico (aritmético-lógicas, cargas desde memoria, transferencia entre registros, etc.), estar a la espera de su terminación arquitectónica significa que están a la espera de copiar en un registro arquitectónico el resultado temporal almacenado en un registro de renombramiento. Aunque no escriben en un registro arquitectónico, las instrucciones de almacenamiento necesitan pasar por una etapa más, la de retirada, que es cuando escriben sus resultados de forma ordenada en la D-caché. Las instrucciones de almacenamiento cuando finalizan guardan los datos a escribir en una parte reservada del buffer de almacenamiento, y cuando terminan arquitectónicamente pasan los valores a otra parte del buffer de almacenamiento en espera de que se realice la escritura en memoria durante la etapa de retirada. Por lo tanto, hay que insistir en que:

- Una instrucción ha finalizado cuando abandona la unidad funcional y queda a la espera en el buffer de terminación.

- Una instrucción ha terminado cuando ha actualizado el estado de la máquina. Solamente se pueden terminar las instrucciones que han finalizado y que no están marcadas como especulativas o como inválidas en el buffer de terminación.

- Una instrucción se ha retirado cuando ha escrito su resultado en la memoria. Este estado solo pueden alcanzarlo las instrucciones de almacenamiento. Así, las instrucciones que no necesitan escribir en memoria son terminadas y retiradas al mismo tiempo.

La pieza clave que garantiza la consistencia del procesador es el buffer de reordenamiento o terminación ya que gestiona la terminación ordenada de todas las instrucciones que están en vuelo. La terminación ordenada asegura que las interrupciones se puedan tratar correctamente. En el momento en que una instrucción fuerce una excepción o sufra una interrupción, el buffer de reordenamiento permitirá expulsar

del cauce todas las instrucciones posteriores sin que se actualicen los registros arquitectónicos y garantizar la terminación ordenada de las anteriores. De esta forma, se puede guardar el estado del procesador en el instante previo a la ejecución de la instrucción que sufrió la anomalía (excepción o interrupción) para volver a reanudar su ejecución una vez la anomalía sea tratada.

El buffer de reordenamiento es la pieza del procesador que pone fin a la ejecución fuera de orden de las instrucciones y forma el *back-end* de un procesador superescalar junto con la etapa de retirada. Este buffer es el elemento que decide cuándo los resultados almacenados temporalmente en el RRF tienen que escribirse en el ARF y una instrucción puede darse por concluida (salvo los almacenamientos). El buffer de reordenamiento es una estructura que mantiene entradas con el estado de todas y cada una de las instrucciones que hay en vuelo: instrucciones que se encuentran en las estaciones de reserva individuales a la espera de emisión, instrucciones en ejecución en las unidades funcionales o instrucciones ya finalizadas a la espera de su terminación arquitectónica. La Figura 2.43.a muestra la composición de una entrada del buffer de reordenamiento con los campos más habituales si se considera que el RRF es una estructura independiente. El significado de estos campos es el siguiente:

- *Ocupada* (O): Es un campo de un bit que indica que la instrucción se ha distribuido. Permanece a 1 hasta que es terminada, momento en que se coloca a 0.
- *Emitida* (E): Campo de un bit que pasa a valer 1 cuando la instrucción inicia su ejecución en una unidad funcional.
- *Finalizada* (F): Campo de un bit que indica que la instrucción ha salido de la unidad funcional y espera ser terminada arquitectónicamente.
- *Dirección* (Dir): Campo que contiene la dirección de memoria de la instrucción como forma de identificarla. Aunque no se ha indicado explícitamente en las secciones previas, la dirección en memoria de una instrucción la acompaña a lo largo de toda la segmentación como etiqueta identificativa. De esta forma, se puede saber en qué estado de procesamiento se encuentra.
- *Registro de destino* (Rd): Es el identificador del registro de destino asociado a la instrucción. Se actualiza una vez que se termine la instrucción para garantizar el correcto tratamiento de las excepciones. Se libera cuando no hay renombramientos pendientes.
- *Registro de renombramiento* (Rr): Es el identificador del registro de renombramiento asociado a la instrucción. De esta forma se sabe el registro de renombramiento que hay que liberar en el RRF.
- *Especulativa* (Es): Campo que identifica a la instrucción como parte de una ruta especulativa. No se pueden terminar aquellas instrucciones marcadas como especulativas.
- *Validez* (V): Campo de un bit para saber si la instrucción puede terminarse o hay que ignorarla y no realizar acción alguna cuando le corresponda su terminación. Desde el instante en que la instrucción entra en el buffer de reordenamiento se considera válida. Sin embargo, posteriormente

la instrucción puede pasar a ser inválida si se trata de una instrucción especulativa cuya predicción se comprueba que es errónea.

Si el RRF fuese parte del buffer de terminación habría que añadir a cada entrada los campos *Datos* (D) y *Validez de los datos* (Vdatos) y eliminar el campo Rr ya que la propia entrada haría de registro de renombramiento. La Figura 2.43.b muestra el formato de una entrada para este tipo de organización del

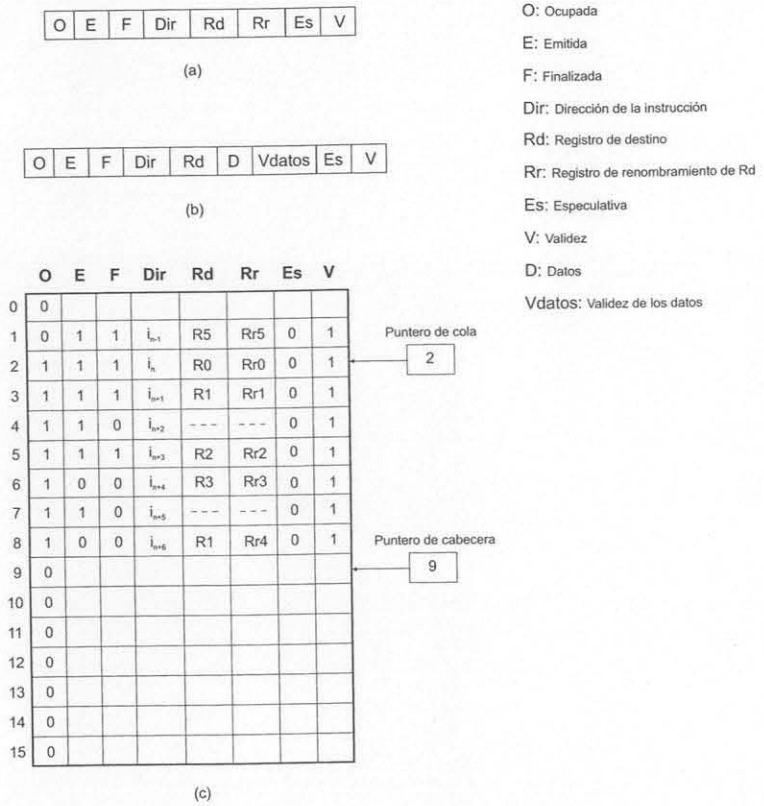


Figura 2.43: Entradas y organización del buffer de reordenamiento.

buffer de terminación.

A medida que la instrucción cambia de etapa y avanza por la segmentación, su estado se modifica y con ello los bits de algunos campos. Si se unen los campos *Ocupada*, *Emitida* y *Finalizada* en un único campo de tres bits, denominado *Estado*, una instrucción pasará sucesivamente por las siguientes situaciones:

- *Estado* = 100: Instrucción en espera de emisión.
- *Estado* = 110: Instrucción en ejecución.
- *Estado* = 111: Instrucción pendiente de terminación.

El funcionamiento del buffer de terminación es similar al de una estructura de datos circular tipo anillo con un puntero de cola y un puntero de cabecera. La Figura 2.43.c muestra la organización del buffer de terminación y el estado de algunas instrucciones si se considera que el RRF es independiente. El puntero de cola apunta a la entrada de la siguiente instrucción a terminar una vez que su estado sea 111 y el puntero de cabecera apunta a la siguiente entrada libre en el buffer. Cada vez que una instrucción se distribuye, el puntero de cabecera se incrementa y se inicializan los campos de la entrada; análogamente el puntero de cola se incrementa cuando una instrucción se termina. Una vez que el puntero de cola o el de cabecera llegan al final del buffer vuelven al comienzo. Recuerde que para que una instrucción pueda distribuirse, entre otras condiciones, debe existir una entrada libre en el buffer de reordenamiento.

Por lo tanto, cuando una instrucción es terminada arquitectónicamente sucede lo siguiente:

- Su registro de renombramiento asociado `Rr` se libera (si lo tiene y el RRF es una estructura independiente).
- El registro destino `Rd` se actualiza con el valor de su registro de renombramiento `Rr` asociado (el valor se obtiene del RRF o del propio buffer si el RRF está integrado) y se libera si no tiene renombramientos pendientes. Una forma de saber que no hay renombramientos pendientes es comprobando que el contenido del campo *Índice* del `Rd` en el ARF coincide con el contenido del campo `Rr` que la instrucción tiene en el buffer de terminación. Es decir, la instrucción terminada fue la última que provocó el renombramiento del registro destino por lo que se puede liberar.
- El campo *Ocupado* se fija de nuevo a 0.
- El puntero de cola se incrementa para apuntar a la siguiente instrucción a terminar.

El número de instrucciones que pueden terminarse simultáneamente depende de la capacidad del procesador para transferir información desde el RRF al ARF o desde el buffer de terminación al ARF. Así, el ancho de banda de terminación quedará establecido por el número de puertos de escritura del ARF y la capacidad de enrutamiento de datos hacia el ARF. Como mínimo será necesario disponer de tantos puertos de escritura en el ARF como instrucciones se quieran terminar por ciclo de reloj. La velocidad

Ciclo 2: Se distribuye i1, i2, i3 e i4

	O	E	F	Dir	Rd	Rr	Es	V
0	1	0	0	i1	R1	Rr0	0	1
1	1	0	0	i2	R2	Rr1	0	1
2	1	0	0	i3	R1	Rr2	0	1
3	1	0	0	i4	R3	Rr3	0	1
4								

Puntero de cola: 0

Puntero de cabecera: 4

Ciclo 3 (inicio): Se emiten i1 e i3. Se distribuye i5

	O	E	F	Dir	Rd	Rr	Es	V
0	1	1	0	i1	R1	Rr0	0	1
1	1	0	0	i2	R2	Rr1	0	1
2	1	1	0	i3	R1	Rr2	0	1
3	1	0	0	i4	R3	Rr3	0	1
4	1	0	0	i5	R1	Rr4	0	1

Puntero de cabecera: 0

Puntero de cola: 0

Ciclo 3 (final): Finaliza i3

	O	E	F	Dir	Rd	Rr	Es	V
0	1	1	0	i1	R1	Rr0	0	1
1	1	0	0	i2	R2	Rr1	0	1
2	1	1	1	i3	R1	Rr2	0	1
3	1	0	0	i4	R3	Rr3	0	1
4	1	0	0	i5	R1	Rr4	0	1

Puntero de cabecera: 0

Puntero de cola: 0

Ciclo 4 (inicio): Se emite i4

	O	E	F	Dir	Rd	Rr	Es	V
0	1	1	0	i1	R1	Rr0	0	1
1	1	0	0	i2	R2	Rr1	0	1
2	1	1	1	i3	R1	Rr2	0	1
3	1	1	0	i4	R3	Rr3	0	1
4	1	0	0	i5	R1	Rr4	0	1

Puntero de cabecera: 0

Puntero de cola: 0

Figura 2.44: Evolución del buffer de terminación (continúa).

Ciclo 4 (final): Finaliza i1

	O	E	F	Dir	Rd	Rr	Es	V
0	1	1	1	i1	R1	Rr0	0	1
1	1	0	0	i2	R2	Rr1	0	1
2	1	1	1	i3	R1	Rr2	0	1
3	1	1	0	i4	R3	Rr3	0	1
4	1	0	0	i5	R1	Rr4	0	1

Puntero de cabecera: 0
Puntero de cola: 0

Ciclo 5 (inicio): Termina i1, se actualiza R1 y se libera Rr0
No se libera R1 porque tiene dos renombramientos pendientes (Rr2 y Rr4)
Se libera la entrada i, del buffer. Se emiten i2 e i5

	O	E	F	Dir	Rd	Rr	Es	V
0	0	1	1	i1	R1	Rr0	0	1
1	1	1	0	i2	R2	Rr1	0	1
2	1	1	1	i3	R1	Rr2	0	1
3	1	1	0	i4	R3	Rr3	0	1
4	1	1	0	i5	R1	Rr4	0	1

Puntero de cabecera: 0
Puntero de cola: 1

Ciclo 5 (final): Finalizan i2 e i4

	O	E	F	Dir	Rd	Rr	Es	V
0	0	1	1	i1	R1	Rr0	0	1
1	1	1	1	i2	R2	Rr1	0	1
2	1	1	1	i3	R1	Rr2	0	1
3	1	1	1	i4	R3	Rr3	0	1
4	1	1	0	i5	R1	Rr4	0	1

Puntero de cabecera: 0
Puntero de cola: 1

Figura 2.44: [Continuación] Evolución del buffer de terminación (continúa).

Ciclo 6 (inicio): Termina i2, se actualiza R2 y se libera Rr1
Termina i3, se actualiza R1 y se libera Rr2
No se libera R1 porque tiene un renombramiento pendiente (Rr4)
Se liberan las entradas 1 y 2 del buffer

	O	E	F	Dir	Rd	Rr	Es	V
0	0	1	1	i1	R1	Rr0	0	1
1	0	1	1	i2	R2	Rr1	0	1
2	0	1	1	i3	R1	Rr2	0	1
3	1	1	1	i4	R3	Rr3	0	1
4	1	1	0	i5	R1	Rr4	0	1

Puntero de cabecera: 0
Puntero de cola: 3

Ciclo 6 (final): Finaliza i5

	O	E	F	Dir	Rd	Rr	Es	V
0	0	1	1	i1	R1	Rr0	0	1
1	0	1	1	i2	R2	Rr1	0	1
2	0	1	1	i3	R1	Rr2	0	1
3	1	1	1	i4	R3	Rr3	0	1
4	1	1	1	i5	R1	Rr4	0	1

Puntero de cabecera: 0
Puntero de cola: 3

Ciclo 7 (inicio): Termina i4, se actualiza R3 y se libera Rr3
Termina i5, se actualiza R1 y se libera Rr4
Se liberan las entradas 3 y 4 del buffer

	O	E	F	Dir	Rd	Rr	Es	V
0	0	1	1	i1	R1	Rr0	0	1
1	0	1	1	i2	R2	Rr1	0	1
2	0	1	1	i3	R1	Rr2	0	1
3	0	1	1	i4	R3	Rr3	0	1
4	0	1	1	i5	R1	Rr4	0	1

Puntero de cabecera: 0
Puntero de cola: 0

Figura 2.44: [Continuación] Evolución del buffer de terminación.

pico del procesador se puede estimar multiplicando el máximo número de instrucciones terminadas por ciclo por la frecuencia del procesador

$$V_{pico} = \text{Máx. instrucciones terminadas por ciclo} * \text{Frecuencia procesador}$$

Antes de analizar la etapa de retirada, se va a estudiar el ejemplo de la Figura 2.41 pero ya aplicando el mecanismo de terminación ordenada que establece el buffer de reordenamiento. El código que se procesa en el ejemplo es:

```
i1: MULT R1,R0,R3
i2: ADD  R2,R1,R0
i3: SUB  R1,R0,R4
i4: MULT R3,R1,R0
i5: MULT R1,R4,R0
```

y el núcleo de procesamiento mantiene las mismas condiciones que para la Figura 2.42 con la salvedad de que ahora dispone de un buffer de terminación un ancho de banda de 2 instrucciones/ciclo. Comparando los comentarios de las Figuras 2.42 y 2.44 podrá apreciar que la única diferencia viene dada por la variación de la secuencia de terminación. En este ejemplo las instrucciones ya no terminan en el ciclo siguiente a su finalización, sino que deben esperar a que les toque su turno que viene establecido por el buffer de terminación. Observe que cuando una entrada del buffer se libera, se mantiene la información en los campos aunque ya no sea válida. Cuando una nueva instrucción se distribuya, su estado sobrescribirá la información correspondiente a la instrucción que hubiese ocupado esa entrada previamente.

2.9. Retirada

La etapa de retirada, ubicada tras la etapa de terminación, es exclusiva de las instrucciones de almacenamiento y es donde estas realizan el acceso a memoria para la escritura de sus resultados. El objetivo de esta etapa para con los almacenamientos es el mismo que el de la etapa de terminación para con las instrucciones que escriben en registros: garantizar la consistencia de memoria, es decir, que las instrucciones de almacenamiento se completen en el orden establecido por el programa. En pocas palabras, evitar los riesgos de memoria WAW y WAR. El mecanismo que se utiliza en esta etapa para lograrlo es el *buffer de almacenamiento* o *buffer de retirada* que consta de dos partes: la de finalización y la de terminación.

El camino que siguen las instrucciones de carga/almacenamiento en una segmentación superescalar una vez que han sido emitidas consta de tres pasos: cálculo de la dirección de memoria, traducción de la dirección de memoria virtual en memoria física y acceso a memoria. La sintaxis genérica de las instrucciones de carga y almacenamiento que emplean direccionamiento basado en el desplazamiento es

```
LD registro_destino,desplazamiento(registro_base)
SD desplazamiento(registro_base),registro_fuente
```


El primer paso, la generación de la dirección de memoria, implica acceder al registro base para leer su valor y sumar este valor al desplazamiento. El desplazamiento es un valor entero que se almacena en un campo del formato de la propia instrucción. Las instrucciones de almacenamiento además deben extraer del registro fuente el valor a almacenar. Al igual que las demás instrucciones, las de carga/almacenamiento también sufren el renombrado de sus registros y las esperas en la estaciones de reserva hasta que sus operandos estén disponibles. Una vez que se dispone de la dirección de memoria, hay que realizar la traducción de memoria virtual a memoria principal. Si el procesador utiliza el modo de direccionamiento real no es necesario realizar este paso ya que el espacio de direcciones que maneja el procesador se corresponde con el espacio de direcciones que proporciona la memoria física. La traducción se realiza consultando la TLB (*Translation Lookaside Buffer* - Buffer de Traducción Adelantada), que es donde se mantienen las equivalencias entre direcciones virtuales y direcciones físicas. Si al realizar el acceso a la TLB, la dirección virtual se encuentra mapeada en la memoria física, la TLB devuelve la dirección física y el acceso se realizará con normalidad. Si la dirección virtual no se encuentra en memoria principal, se produce un fallo de página que da pie al levantamiento de una *excepción de fallo de página*. Esta excepción es gestionada por el sistema operativo ya que el copiar una página desde los discos duros a la memoria principal consume cientos de miles de ciclos de reloj, lo que obliga al sistema operativo a suspender la ejecución del proceso que ha provocado la excepción y entregar el procesador a otro proceso hasta que la copia de la página concluya. Recuperada la página y el estado del procesador previo a la instrucción que provocó la excepción, la instrucción de carga/almacenamiento puede volver a emitirse ya que ahora hay seguridad de que no se producirá un fallo de página. Para más detalles sobre el funcionamiento de la memoria virtual, consulte el Apéndice A. En este mismo paso, una vez traducida la dirección se coloca en el bus de direcciones para tener el dato disponible en el siguiente ciclo.

Realizada la traducción y el envío de la dirección a la caché de datos, el paso final de una instrucción de carga/almacenamiento es la lectura/escritura de/en memoria. Es en este paso cuando el procesamiento de las instrucciones de carga y almacenamiento difiere. Las instrucciones de carga realizan los tres pasos en la etapa de ejecución mientras que las instrucciones de almacenamiento dejan el acceso a memoria para la etapa de retirada. Al llegar una instrucción de carga/almacenamiento al tercer paso, no debe producirse ningún fallo de página ya que la posición de memoria en la que leer/escribir tiene que estar ubicada en la D-caché o en la memoria física. Si el dato se encuentra en la D-caché, la lectura por parte de la instrucción de carga consume un ciclo de reloj. Si el dato no está, se produce un fallo de lectura y la instrucción de carga queda detenida en la segmentación hasta que el fallo se resuelva. Una vez que la instrucción ha recuperado el dato, lo coloca en el bus de reenvío junto con el identificador del registro destino del RRF para que se actualicen el RRF (o la entrada del buffer de terminación si el RRF es parte de él) y las entradas de las estaciones de reserva individuales que detecten coincidencia de operandos. En la etapa de terminación se realiza la actualización del correspondiente registro ARF, tal y como sucede con cualquier otra instrucción para garantizar el tratamiento correcto de las interrupciones.

Las instrucciones de almacenamiento acceden a memoria en la etapa de retirada, para dar mayor prioridad al acceso de las instrucciones de carga a la D-caché. Por esta razón, los almacenamientos

se consideran finalizados tras realizar solo dos pasos de la etapa de ejecución, y no tres como las cargas. Una vez que una instrucción de almacenamiento es finalizada, el dato a escribir y la dirección de memoria se guardan en la entrada que a la instrucción se le asigna en el buffer de almacenamiento, y en esa misma entrada se marca la instrucción como finalizada. Esta actualización del buffer de almacenamiento se realiza al mismo tiempo que la actualización del buffer de reordenamiento, es decir, están sincronizados. Cuando la instrucción es terminada por mandato del buffer de reordenamiento, la instrucción de almacenamiento se marca en el buffer de almacenamiento como terminada, quedando lista

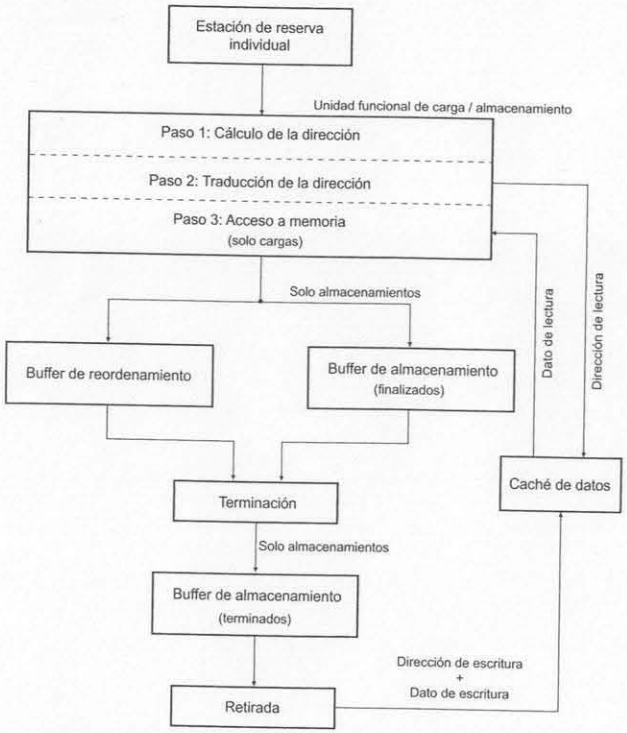


Figura 2.45: Organización de la etapa de ejecución y del *back-end* de la segmentación para el procesamiento diferenciado de las instrucciones de carga y almacenamiento.

para escribir en memoria en cuanto el bus de acceso esté libre. La Figura 2.45 muestra un esquema de los pasos de ejecución de las instrucciones de carga y almacenamiento. En el esquema se ha recurrido a un buffer de almacenamiento para las instrucciones finalizadas y otro para las terminadas aunque en realidad se puede utilizar uno único con solo añadir un campo de un bit para diferenciar entre el estado finalizado y el terminado.

La escritura diferida por parte de una instrucción de almacenamiento tras el abandono del buffer de reordenamiento evita una actualización errónea y precipitada de la memoria. Suponga que en el ejemplo de la Figura 2.46 la escritura en memoria de `i3` se ha realizado en la etapa de ejecución y la instrucción `i2`, previa a `i3` en el orden del programa pero posterior en su ejecución, provoca una excepción. El tratamiento de la excepción obliga a vaciar el cauce, anular el procesamiento de todas las instrucciones situadas después en el orden del programa (lo que incluye el almacenamiento `i3`), y así poder guardar el estado de la máquina en el momento previo a la ejecución de la instrucción que forzó la excepción, la `i2`. Una situación como la anterior obligaría a acceder a la memoria para deshacer el almacenamiento anticipado de `i3`, con el consiguiente gasto en ciclos de reloj que ello conllevaría. Posteriormente, se abordará con más detalle el tratamiento de las interrupciones y excepciones.

Al igual que las restantes instrucciones, una instrucción de almacenamiento puede estar marcada como especulativa en el buffer de reordenamiento. Por ello, si se produce una predicción incorrecta, la instrucción de almacenamiento se invalida en el buffer de reordenamiento y se purga del buffer de almacenamiento. Como nunca se pueden terminar las instrucciones especulativas, no existe el problema de que existan instrucciones marcadas como terminadas en el buffer de almacenamiento. Si están marcadas como terminadas, es que no son especulativas.

Si en un instante dado una instrucción que está ejecutándose produce una excepción, las instrucciones de almacenamiento marcadas como terminadas en el buffer de almacenamiento deben retirarse en orden

```
i1: MULT  R1, R1, R1      // Terminada
i2: DIV   R2, R1, R0      // Lanza una excepción: división por cero
i3: SD    100(R5), R0     // Terminada con escritura en memoria del contenido de R0
i4: .....
```

	O	E	F	Dir	Rr	Es	V	
0	0	1	1	i1	Rr1	0	1	
1	1	1	0	i2	Rr2	0	1	← Puntero de cola
2	1	1	1	i3	Rr0	0	1	
3								← Puntero de cabecera

Buffer de reordenamiento

Figura 2.46: Ejemplo de situación anómala si la escritura en memoria se realizase en la fase de ejecución y se produjese una excepción.

para asegurar la consistencia de memoria. Las instrucciones de almacenamiento que estén marcadas como finalizadas pero que sean anteriores a la instrucción que lanzó la excepción también se terminarán y retirarán en orden. Las que sean posteriores, se eliminarán de todos los buffers en que se encuentren.

Como colofón a este apartado, hay que comprender que el orden en la terminación y retirada de las instrucciones de carga/almacenamiento garantiza las dependencias de memoria tipo WAR y WAW. Los riesgos de memoria WAW se evitan, claramente, mediante el buffer de almacenamiento que asegura que las escrituras en la D-caché se realizan respetando el orden en que aparecen en el programa. Los riesgos de memoria WAR se evitan gracias a que una instrucción de almacenamiento posterior a una carga en el orden del programa, nunca podrá escribir antes que la carga ya que lo evita la escritura diferida: aunque el almacenamiento vaya por delante en el cauce, no podrá escribir hasta después de que la carga termine, y ésta, llegada a ese punto, ya habrá leído su operando de la memoria.

2.10. Mejoras en el procesamiento de las instrucciones de carga/almacenamiento

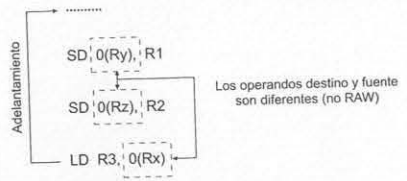
En los múltiples análisis realizados por investigadores y diseñadores de computadores al código que se ejecuta en los procesadores, se ha observado que hay instrucciones de carga que escriben en un registro que constituye el punto de arranque crítico para un conjunto de instrucciones aritmético-lógicas que dependen, directa o indirectamente, de ese operando. Por esa razón, una mejora notable en el rendimiento de los procesadores superescalares se obtiene permitiendo que esas instrucciones de carga terminen antes que otras instrucciones, sin que se violen dependencias RAW tanto de datos como de memoria. La lectura adelantada del dato por la carga permite que todas las instrucciones aritmético-lógicas dependientes de ella, directa o indirectamente, tengan el camino libre para su ejecución.

Otro mecanismo que mejora ligeramente el rendimiento de un procesador es el reenvío de datos desde una instrucción de almacenamiento hacia una de carga que tengan operandos destino y fuente comunes, respectivamente. La aplicación de esta técnica permite que la carga no necesite acceder a la D-caché para su lectura ya que el almacenamiento le reenviará el dato directamente. El término anglosajón con que se define esta técnica es *load-and-store forwarding*.

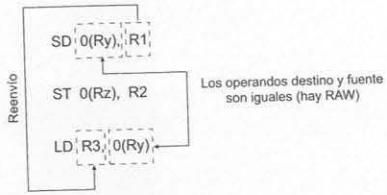
Tanto la terminación adelantada de las cargas como el reenvío son dos técnicas que se pueden aplicar simultáneamente. En la Figura 2.47 se puede apreciar la razón de ello. Si una carga se puede terminar antes que un almacenamiento es porque no hay dependencia de memoria RAW entre ellas (acceden a direcciones diferentes), y si no se puede es porque existe dependencia de memoria RAW (acceden a direcciones iguales) y el almacenamiento puede pasar el dato a la carga ahorrando a esta un acceso a memoria.

2.10.1. Reenvío de datos entre instrucciones de almacenamiento y de carga

El reenvío se puede realizar cuando existe una dependencia de memoria RAW entre una instrucción de almacenamiento (productora) y una instrucción de carga (consumidora). Para poder realizar el



(a) Situación que permite terminar una carga sin que terminen los adelantamientos



(b) Situación que permite que una instrucción de almacenamiento transfiera un dato a una carga para evitar el acceso a la caché de datos

Figura 2.47: Esquemas de adelantamiento y reenvío de datos entre instrucciones de carga y almacenamiento.

reenvío es necesario comprobar que las direcciones de memoria de una instrucción de carga y de los almacenamientos pendientes sean coincidentes. Se pretende evitar que la carga acceda a la D-caché para leer la misma posición de memoria que va a escribir algunos de los almacenamientos que todavía están pendientes de retirarse. Es decir, el dato que necesita la carga está en un registro del procesador y no hay que ir a la memoria para obtenerlo.

Los riesgos de datos RAW se resuelven en las estaciones de reserva pero, sin embargo, los riesgos de memoria RAW todavía no han sido tratados y son los que hay que analizar para reenviar datos y adelantar cargas. Examine con detalle el siguiente fragmento de código

```
i1: SD   100(R2),R1   % Mem[100+R2] <= R1
.....................
i2: LD   R3,50(R4)    % R3 <= Mem[50+R4]
```

En principio, las direcciones de memoria de `i1` e `i2` son diferentes y no sería correcto reenviar el contenido de R1 a R3, ya que, aparentemente, no hay dependencia de memoria RAW entre `i1` e `i2`. El problema es que la coincidencia de direcciones no es posible conocerla cuando las instrucciones se encuentran en la estaciones de reserva. Por ello la comprobación de coincidencias se realiza una vez que las instrucciones de carga y almacenamiento han sido emitidas y finalizadas/terminadas, respectivamente.

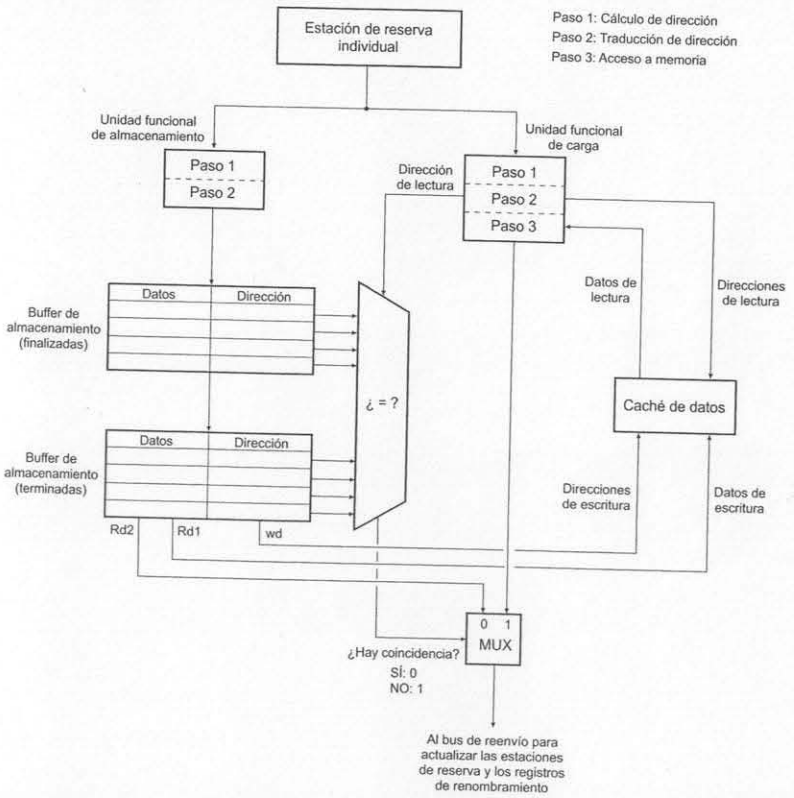


Figura 2.48: Esquema de reenvío de datos entre cargas y almacenamientos en el que se contempla la emisión ordenada de las cargas y los almacenamientos.

En el ejemplo anterior, si R2= 50 y R4= 100, las dos direcciones de memoria serían iguales, `M[150]`, y surgiría una dependencia de memoria tipo RAW. La situación entre `i1` e `i2` se denomina *dependencia ambigua* y es propia de las instrucciones de carga/almacenamiento.

La Figura 2.48 muestra un esquema que permite realizar el reenvío de datos y resolver las dependencias ambiguas. En el esquema hay una unidad funcional de carga y otra de almacenamiento,

ambas atendidas por la misma estación de reserva individual que garantiza la emisión de las instrucciones en orden. Esto asegura que todos los almacenamientos precedentes a una carga ya se encontrarán en el buffer de almacenamiento en espera de terminación o retirada al ejecutarse la carga. De acuerdo con el esquema de la figura, cuando una carga ya ha traducido la dirección de memoria virtual en dirección física (paso 2), se examina el campo *Dirección* de las entradas ocupadas en el buffer de almacenamiento, ya estén las entradas marcadas como finalizadas o terminadas, en busca de una coincidencia con la dirección de la instrucción de carga. En este momento ya se ha eliminado la ambigüedad de las dependencias puesto que los almacenamientos y la carga ya han calculado y traducido sus direcciones de acceso a memoria. En el proceso de búsqueda de direcciones coincidentes pueden plantearse dos situaciones:

- Hay coincidencia (hay una dependencia de memoria RAW): Existe en el buffer de almacenamiento una instrucción que va a escribir un dato en la misma posición de memoria que la carga tiene que leer. El valor que hay en el campo *Datos* de la entrada que presenta la coincidencia se coloca en el bus de reenvío junto con el identificador del registro destino RRF de la carga. La carga no necesita acceder a la D-caché.

- No hay coincidencia (no hay dependencia de memoria RAW): Se realiza el acceso a la caché de datos de forma normal.

Un problema que puede surgir es la aparición de una carga que presenta varias coincidencias en el buffer de almacenamiento. En ese caso, el procesador debe contar con el hardware adecuado para saber cuál es el almacenamiento más reciente, que es el que hay que utilizar como dato. Para ello es necesario mantener los almacenamientos ordenados según su orden de llegada y utilizar codificadores con prioridades para utilizar la dirección más reciente en caso de múltiples coincidencias.

Otro aspecto a considerar es el número de puertos necesarios en el buffer de almacenamiento para la lectura de datos. Hasta ahora era necesario un único puerto de lectura (Rd) para enviar los datos a la caché y un puerto de escritura (Wd) para introducir los datos tras la finalización de los almacenamientos. Pero con el reenvío puede ocurrir que una carga lea el dato en el mismo ciclo de reloj en que se envía el dato a la caché para su escritura, ocurriendo un bloqueo. Es necesario un segundo puerto de lectura para garantizar la ausencia de bloqueos. En el esquema de la Figura 2.48 se recoge la existencia de dos puertos de lectura de datos (Rd1, Rd2).

2.10.2. Terminación adelantada de las instrucciones de carga

El adelantamiento de las instrucciones aritmético-lógicas respetando las dependencias de datos ya se realiza gracias a los mecanismos con que cuentan los procesadores superescalares: varias unidades funcionales, renombrado de registros, estaciones de reserva individuales, etc. Sin embargo, adelantar la ejecución de una instrucción de carga frente a varias instrucciones de almacenamiento es más complicado ya que es necesario comprobar que las direcciones de memoria que manejan no son iguales. Es decir, hay que comprobar la ausencia de dependencias de memoria RAW teniendo en cuenta la problemática de las dependencias ambiguas.

Al igual que en el caso anterior, suponga que las instrucciones de carga/almacenamiento fuesen emitidas en orden por la estación de reserva a las unidades funcionales de carga y almacenamiento. La carga, mientras que realiza el acceso a la caché de datos (tercer paso), podría comprobar la coincidencia de su dirección en el campo *Dirección* de las entradas del buffer de almacenamiento. Dos son las posibles situaciones que se podrían presentar:

- Hay coincidencia de direcciones (dependencia de memoria RAW): No se puede adelantar la carga. El dato recuperado por la carga de la D-caché se descarta y se utiliza el dato del almacenamiento coincidente (se produce un reenvío de datos).
- No hay coincidencia de direcciones (no hay dependencia RAW): El dato recuperado de la D-caché es válido y se puede almacenar en su registro destino. La carga puede finalizar y terminarse, aunque no lo hayan hecho los almacenamientos (se produce un adelantamiento).

La emisión ordenada de las instrucciones de carga/almacenamiento se utiliza en algunos procesadores ya que reduce la complejidad del hardware que es necesario para resolver las dependencias ambiguas. Pero esta aproximación ofrece un menor rendimiento que si se permitiese una ejecución desordenada de estas instrucciones por lo que es deseable permitir el adelantamiento de cargas con emisión fuera de orden.

Una técnica utilizada para realizar el adelantamiento de cargas permitiendo una emisión desordenada de las cargas y de los almacenamientos se basa en el empleo de un *buffer de cargas finalizadas*. La Figura 2.49 muestra un esquema con el buffer de cargas en el que la estación de reserva individual emite fuera de orden las instrucciones de carga y de almacenamiento. La idea de esta técnica se basa en permitir que, cuando sea posible, las cargas se ejecuten de forma especulativa sin comprobar la existencia de dependencias ambiguas.

Las cargas cuando llegan al segundo paso de su unidad funcional comprueban la existencia de coincidencias de dirección con los almacenamientos que, en ese instante, se encuentren en el buffer de almacenamiento. Se plantean dos situaciones:

- No hay coincidencia: Se permite que la carga continúe normalmente ya que no hay ningún tipo de dependencia de memoria RAW.
- Hay coincidencia: Hay una dependencia de memoria RAW con un almacenamiento. Se anula la carga y todas las instrucciones posteriores para su emisión posterior. No se reenvían datos ya que pueden quedar almacenamientos anteriores pendientes de ejecución.

Tras finalizar, las cargas quedan almacenadas en el buffer de cargas finalizadas. Llegados a este punto, se ha comprobado que las cargas no dependen de los almacenamientos que están en el buffer de almacenamiento. Quedan pendientes los almacenamientos que no conocen sus direcciones de acceso a memoria, estos son los pendientes de emisión que se encuentran en las estaciones de reserva individuales y los que hay en ejecución.

Cuando una instrucción de almacenamiento termina tiene que comprobar la existencia de una coincidencia con las cargas que estén en el buffer de cargas finalizadas. Pueden darse dos situaciones:

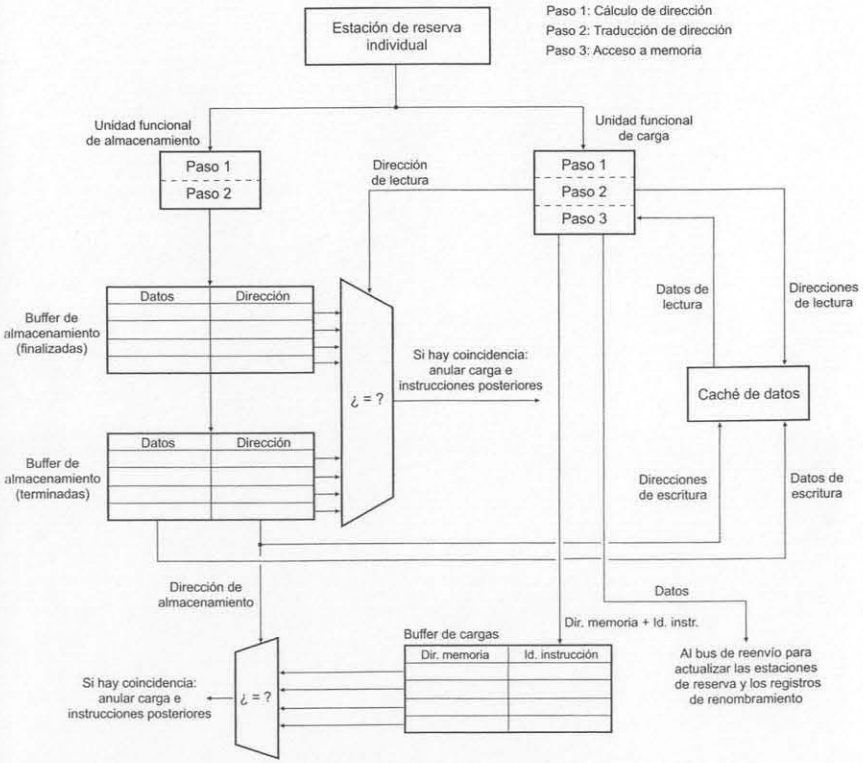


Figura 2.49: Esquema para realizar el adelantamiento de las cargas permitiendo la emisión fuera de orden de las cargas y los almacenamiento.

- Hay coincidencia: Esto significa que una instrucción de carga que depende del almacenamiento se ha ejecutado de forma especulativa. Se ha violado una dependencia RAW de memoria ya que la carga ha leído el dato antes de que el almacenamiento lo escribiese. Hay que invalidar esa carga y todas las instrucciones posteriores a ella para su posterior emisión. Observe que esta técnica de adelantamiento de cargas especulativas no es compatible con el reenvío descrito en el apartado anterior. El campo *Id. Instrucción* permite identificar la instrucción de carga coincidente

y utilizar el identificador para acceder al buffer de reordenamiento y eliminarla a ella y a todas las posteriores.

- No hay coincidencia: La carga y todas las instrucciones posteriores a ella pueden terminarse ya que no hay ningún tipo de dependencia entre el almacenamiento y la carga.

2.11. Tratamiento de interrupciones

Dada la naturaleza de los procesadores superescalares, el tratamiento de las interrupciones es más complicado que en un procesador con un único cauce segmentado. Ello se debe a que la ejecución de instrucciones fuera de orden en un procesador superescalar puede modificar el estado de un proceso (programa en ejecución) en un orden diferente al que se obtendría en una ejecución secuencial. Cuando se produce una interrupción, el estado del proceso interrumpido es, por lo general, guardado por el hardware, el software o una combinación de ambos. El estado del procesador está definido por el contador de programa, los registros arquitectónicos y el espacio de memoria asignado.

Si un procesador es capaz de tratar las interrupciones de forma que se respeten las condiciones previas se dice que el procesador tiene *precisión de excepción* o que maneja *interrupciones precisas*. Si no se cumple, se dice que las interrupciones son imprecisas. Resumiendo, para poder tener interrupciones precisas es necesario que el procesador mantenga su estado y el de la memoria de forma análoga a como si las instrucciones se ejecutasen en orden y una tras otra (como en los procesadores originales de 8 bits). El estado del procesador debe evolucionar secuencialmente cuando una instrucción termina con independencia de que, previamente, se haya ejecutado fuera de orden.

Es habitual clasificar las interrupciones en dos grandes grupos:

- *Interrupciones externas* producidas por eventos externos a la ejecución del programa (operación de E/S por un dispositivo externo, cambio de contexto, malfunción hardware, fallos de potencia, etc.) Una estrategia habitual de actuación consiste en detener la lectura de nuevas instrucciones, vaciar el cauce terminando arquitectónicamente las instrucciones ya existentes y guardar el estado en que quedó el procesador tras terminar la instrucción previa a la que fue interrumpida. Una vez que el sistema operativo haya realizado el tratamiento de la interrupción, se podrá recuperar el estado del procesador y continuar con la ejecución del programa a partir de la última instrucción que se terminó.

- *Excepciones, interrupciones de programa o traps* causadas, en general, en la etapa IF o por las instrucciones del programa en ejecución. Los ejemplos más conocidos son los errores numéricos (divisiones por cero, desbordamientos aritméticos), aunque no hay que olvidar los fallos de página en el sistema de memoria virtual tanto para instrucciones como datos. El tratamiento de las excepciones es similar al de las interrupciones externas aunque en algunos tipos de excepción no se considera la reanudación del programa como, por ejemplo, las divisiones por cero o algunos desbordamientos. Incluso en estas situaciones, en las que el programa no se reanuda, el estado

Tabla 2.1: Clasificación de las interrupciones del PowerPC 970.

Síncrona/Asíncrona	Precisa/Imprecisa	Tipos de interrupción
Asíncrona, no enmascarable	Imprecisa	Reinicialización del computador, test de la máquina.
Asíncrona, enmascarable	Precisa	Interrupción externa, excepción térmica, excepción de mantenimiento.
Síncrona	Precisa	Excepciones causadas por instrucciones.

del procesador se puede almacenar para tareas de depuración (*debugging*) que permitan conocer y analizar las causas del problema.

Las técnicas para el tratamiento de interrupciones en un procesador que permite ejecución fuera de orden se pueden agrupar en cuatro categorías:

- Ignorar el problema y no garantizar la precisión de excepción. Muy utilizada en los años 60 y principios de los 70, y todavía empleada por algunos supercomputadores que no permiten ciertas interrupciones o si lo son se tratan vía hardware sin detener la ejecución del proceso.

- Permitir que las interrupciones sean "algo" imprecisas pero guardando suficiente información para que la rutina de tratamiento software pueda recrear una secuencia precisa para recuperarse de la interrupción.

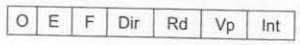
- Permitir que una instrucción sea emitida solo cuando se está seguro de que las instrucciones que la preceden terminarán sin causar ninguna interrupción.

- Mantener los resultados de las operaciones en un buffer de almacenamiento temporal hasta que todas las operaciones previas hayan terminado correctamente y se puedan actualizar los registros arquitectónicos sin ningún tipo de riesgo. Entre estas técnicas se encuentran el buffer de historia (*history buffer*) y el fichero de futuro (*future file*). Esta forma de tratar las excepciones ha ido adaptándose a la evolución que han experimentado los procesadores superescalares hasta llegar al mecanismo actual basado en la combinación del renombramiento de registros, las estaciones de reserva y el buffer de terminación. A continuación, se describen con más detalle las técnicas pertenecientes a esta categoría.

Por ejemplo, en el procesador PowerPC 970 las interrupciones pueden ser precisas o imprecisas y síncronas o asíncronas. Por asíncronas se entiende que pueden aparecer en cualquier momento ya que son causadas por eventos externos a la ejecución del procesador y por enmascarables que pueden deshabilitarse por parte del programador. La Tabla 2.1 presenta una clasificación de algunas de las interrupciones de este procesador.

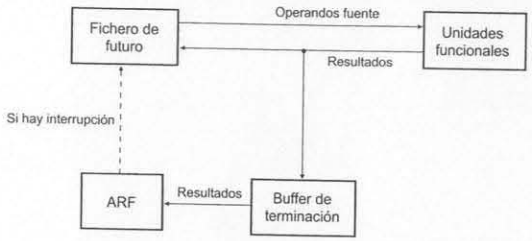
La precisión de excepción basada en un almacenamiento temporal de los resultados se ideó a mediados de los 80, cuando los resultados se mantenían provisionalmente en el buffer de terminación y

(a) Procesamiento de interrupciones precisas basado en el buffer de historia



O: Ocupada
E: Emitida
F: Finalizada
Dir: Dirección de la instrucción
Rd: Registro de destino
Vp: Valor previo
Int: Interrupción

(b) Entrada del buffer de historia



(c) Procesamiento de interrupciones precisas basado en fichero de futuro

Figura 2.50: Esquemas para el mantenimiento de interrupciones precisas.

desde allí se copiaban al ARF y se adelantaban a las unidades funcionales cuando existían dependencias RAW. El control se realizaba mediante una estructura denominada *Result Shift Register* (RSR) que garantizaba una emisión y terminación ordenada de las instrucciones. Para lograr la precisión de excepción se diseñaron dos técnicas: el buffer de historia y el fichero de futuro. Para entender correctamente el funcionamiento de ambas técnicas hay que considerar que cuando se idearon, la emisión y terminación de instrucciones era individual y no se disponía de renombrado de registros, ni de estaciones de reserva. Durante 1983 y 1984 ya se utilizaba el término superescalar pero todavía la decodificación y emisión de múltiples instrucciones era un tema en pura efervescencia.

La primera de las dos técnicas se basa en utilizar un buffer de historia, muy similar en su estructura y funcionamiento al buffer de reordenamiento, ya que permite una terminación desordenada de las instrucciones (Figura 2.50.a). La diferencia radica en que en sus entradas (Figura 2.50.b) se almacena el valor de los registros destino al comienzo de la ejecución de las instrucciones (campo `Vp`), no al finalizar. De esta forma, cuando una instrucción es interrumpida, parte del contenido del ARF se recupera de la información mantenida en las entradas del buffer de historia que corresponden a la instrucción interrumpida y a las posteriores. Así, si una instrucción hubiese finalizado antes que la interrumpida pese a ser posterior en el orden secuencial, el estado de su registro destino se sacaría del campo `Vp` de su entrada en el buffer de historia y se copiaría en el ARF. De esta forma el registro quedaría en el estado previo a cuando comenzó el procesamiento de la instrucción interrumpida.

La segunda técnica combina un fichero de registros de futuro con el buffer de reordenamiento (Figura 2.50.c). Los operandos fuente se leen del fichero de futuro y al finalizar las instrucciones los resultados se escriben en dos ubicaciones: en el fichero de futuro y en el buffer de terminación. Cuando las instrucciones terminan, el ARF se actualiza con la información del buffer de terminación. Si una instrucción es interrumpida, el fichero de futuro recupera los valores originales de los registros destino de la instrucción interrumpida y de las posteriores desde el ARF.

La Figura 2.51 presenta un ejemplo de aplicación de la técnica del buffer de historia para gestionar de forma precisa una interrupción que aparece durante la ejecución de la instrucción `i4`. La secuencia de código está formada por 6 instrucciones que lo único que hacen es multiplicar por 100 el contenido de su registro destino:

```
i1: MULTI R1,R1,#100
i2: MULTI R2,R2,#100
i3: MULTI R3,R3,#100
i4: MULTI R4,R4,#100
i5: MULTI R5,R5,#100
i6: MULTI R6,R6,#100
```

El ejemplo comienza presentando el estado del ARF y del buffer de historia con las instrucciones `i1` e `i2` ya terminadas, las instrucciones `i3` e `i4` en ejecución, la `i5` en espera de ser emitida y la `i6` finalizada (ciclo 1). El contenido del campo `Vp` del buffer de historia contiene al valor inicial que tenían los registros destino en el ARF al iniciarse el procesamiento de las instrucciones. Por otra parte, el contenido de los registros `R1`, `R2` y `R6` en el ARF ya contiene el resultado de las instrucciones que han finalizado, esto es,

i1, i2 e i6, esta última fuera de orden (antes que i3, i4 e i5). Cuando finaliza i3 se escribe su resultado en el registro R3 del ARF, terminándose en el siguiente ciclo. Análogamente, cuando la instrucción

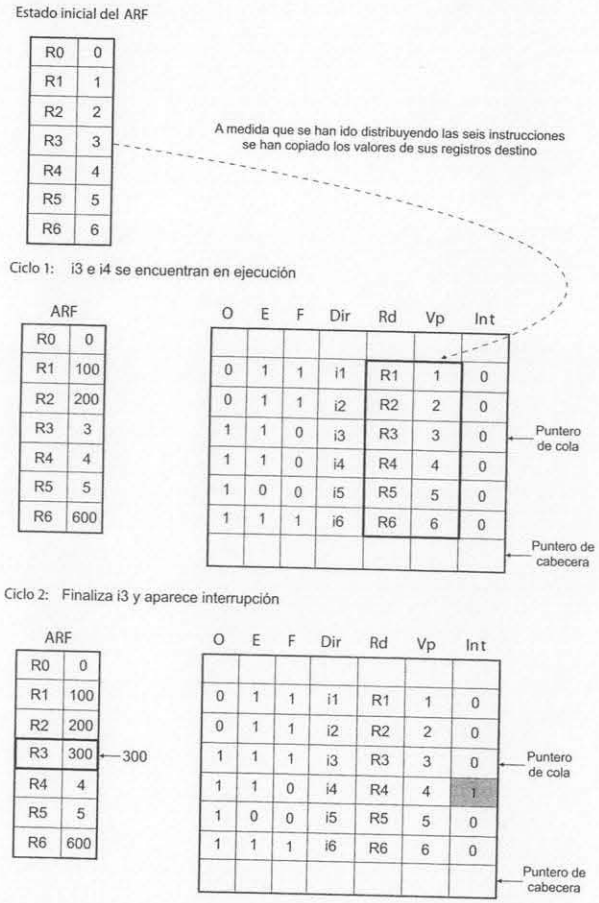


Figura 2.51: Ejemplo de funcionamiento del buffer de historia (continúa).

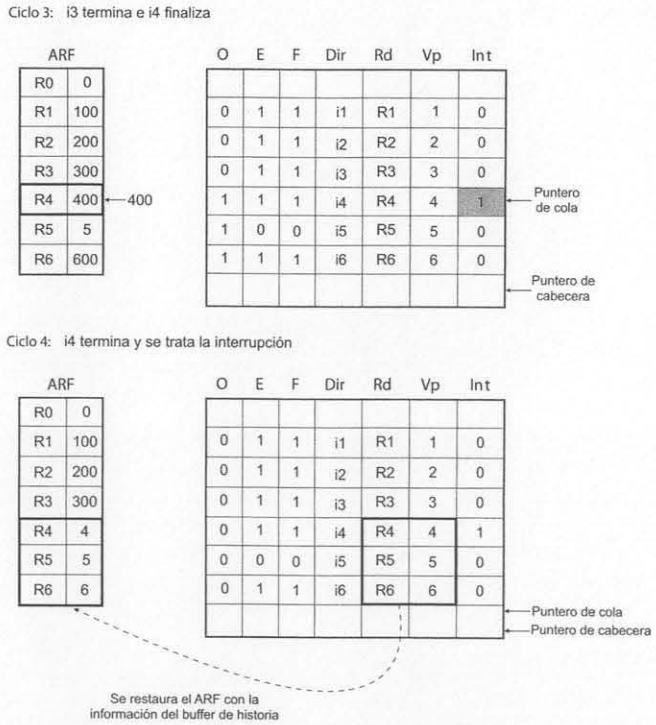
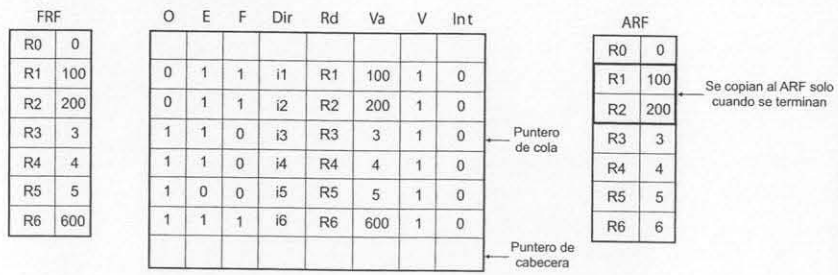


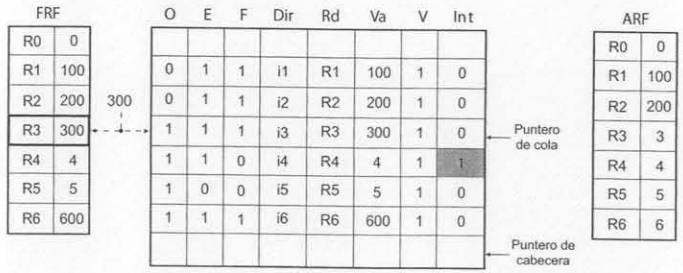
Figura 2.51: [Continuación] Ejemplo de funcionamiento del buffer de historia.

i4 finaliza, escribe su resultado en el ARF, y queda lista para ser terminada en el ciclo siguiente. Sin embargo, durante la etapa de ejecución de i4 apareció una interrupción que quedó marcada en su campo Int del buffer de historia, lo que modifica su terminación. Ahora, al terminar i4 y detectar que se trata de una instrucción interrumpida, se anula su resultado y el de las siguientes instrucciones. Esto consiste en reestablecer los valores de los registros R4, R5 y R6 utilizando la información almacenada en sus respectivas entradas del buffer de historia. Una vez recuperado el contenido de los registros del ARF con su valor previo a la ejecución de i4, se guarda el estado del procesador y se da paso a la rutina de tratamiento de la interrupción.

Ciclo 1: i3 e i4 se encuentran en ejecución



Ciclo 2: Finaliza i3 y aparece interrupción



Ciclo 3: i3 termina e i4 finaliza

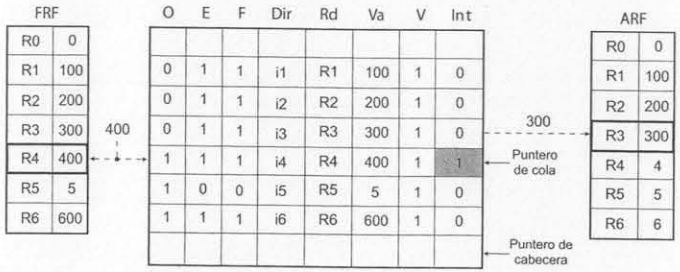


Figura 2.52: Ejemplo de funcionamiento del fichero de futuro (continúa).

Ciclo 4: i4 termina y se trata la interrupción

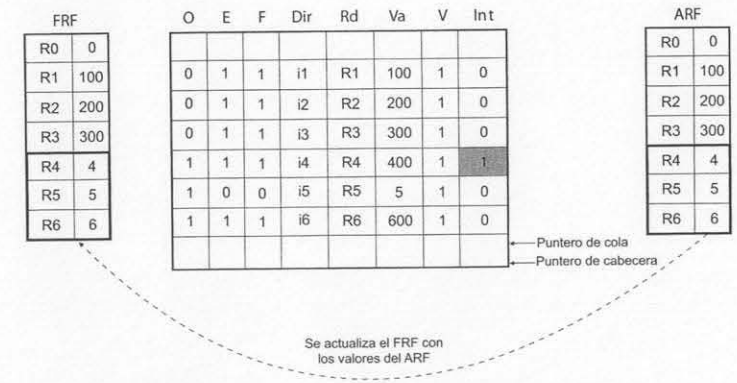


Figura 2.52: [Continuación] Ejemplo de funcionamiento del fichero de futuro.

En la Figura 2.52 se plantea el mismo ejemplo que con el buffer de historia pero aplicando la técnica del fichero de futuro con el buffer de reordenamiento. El estado de las instrucciones al comenzar el ejemplo es el mismo que en el caso anterior. La principal diferencia es que al finalizar `i1`, `i2` e `i6`, sus resultados se almacenan temporalmente en las entradas del buffer de terminación y en el fichero de registros de futuro (*Future Register File* - FRF). Como el ejemplo se inicia con las instrucciones `i1` e `i2` terminadas, sus resultados ya se encuentran en el ARF. Al finalizar la ejecución de `i3`, su resultado se almacena en el FRF y en el buffer de reordenamiento para, una vez terminada, copiarlo al ARF. La instrucción `i4` cuando finaliza también copia su resultado al buffer de terminación y al FRF. Al ir a terminarla, ya que se trata de una instrucción marcada, es necesario anular su procesamiento y el de las instrucciones siguientes. Ello se consigue reestableciendo el contenido del FRF al estado previo a iniciar la ejecución de `i4`, es decir, el estado en que quedaría el procesador tras ejecutar ordenamente las instrucciones que componen el programa hasta `i3`. Para ello hay que recuperar los valores de los registros `R4`, `R5` y `R6` del ARF y copiarlos al FRF. Tras esto, se guarda el estado del procesador y se trata la interrupción.

Las técnicas basadas en el buffer de historia y en el registro de futuro proporcionan precisión de excepción retrasando el almacenamiento definitivo de los resultados mientras que las instrucciones no sean terminadas arquitectónicamente y dadas por válidas. Actualmente, la mayor parte de los procesadores superescalares basan la precisión de excepción en el empleo del mecanismo constituido por la combinación del renombramiento, las estaciones de reserva y los buffers de terminación, almacenamientos y cargas. En el apartado siguiente se analiza con mayor detalle este esquema.

2.11.1. Excepciones precisas con buffer de reordenamiento

El elemento clave que permite mantener la consistencia del procesador es el buffer de reordenamiento o de terminación, ya que posibilita que las instrucciones no interrumpidas terminen en el mismo orden en que se sitúan en el programa, almacenando sus resultados definitivos de forma ordenada. Para conocer si una instrucción ha sufrido una interrupción durante su etapa de ejecución, las entradas del buffer de reordenamiento cuentan con un campo `Int`. Inicialmente a 0, este campo pasa a valer 1 para indicar que una instrucción ha sido interrumpida, de forma que cuando haya que terminarla se sepa que hay que anular su procesamiento y el de las siguientes.

```
i1: LD    R1, 100(R0)     // R1 renombrado como Rr1
i2: ADD   R3, R2, R1      // R3 renombrado como Rr3
i3: DIV   R4, R4, R3      // R4 renombrado como Rr4
i4: MULT  R6, R5, R5      // R6 renombrado como Rr5
i5: SD    100(R1), R4     // No hay registro destino
i6: ADD   R4, R4, R3      // R4 renombrado como Rr6
```

O	E	F	Dir	Rd	Rr	Es	V	Int	
0	1	1	i1	R1	Rr1	0	1	0	
0	1	1	i2	R3	Rr3	0	1	0	
1	1	0	i3	R4	Rr4	0	1	0	← Puntero de cola
1	1	1	i4	R6	Rr5	0	1	1	
1	0	0	i5	.....	.....	0	1	0	← Se anula su procesamiento
1	0	1	i6	R4	Rr6	0	1	0	
									← Puntero de cabecera

Figura 2.53: Estado del buffer de reordenamiento al producirse una interrupción.

La Figura 2.53 muestra el estado del buffer de reordenamiento durante el procesamiento de una secuencia de seis instrucciones. La situación de las instrucciones es la siguiente: `i1` e `i2` ya han finalizado mientras que `i3` se encuentra todavía en ejecución; la instrucción `i4` también ha finalizado pero recibió una interrupción durante su etapa de ejecución, por eso mantiene el campo `Int`= 1 en el buffer de reordenamiento; finalmente, las instrucciones `i5` e `i6` se encuentran a la espera de poder ser emitidas ya que dependen del resultado de `i3`. En el momento en que el puntero de cola llegue a la entrada correspondiente a la instrucción `i4` y compruebe que se puede terminar pero que el campo `Int` está a 1, se iniciará el tratamiento de la excepción. Este consistiría en:

- Expulsar del cauce a `i4` y a todas las instrucciones posteriores a ella. Aunque en el ejemplo solo son `i5` e `i6`, esta acción afectaría a todas las instrucciones que estuviesen leídas, decodificadas, distribuidas, en vuelo o finalizadas. También se eliminan los almacenamientos que estuviesen marcados como finalizados en el buffer de almacenamiento y, en caso de existir, las cargas que hubiese en el buffer de cargas especulativas. Es decir, todas las instrucciones sin terminar.

- Almacenar el estado del procesador, es decir, guardar el estado de los registros arquitectónicos y del contador de programa. El estado debe corresponder a la situación del procesador tras completar la ejecución secuencial del programa hasta la instrucción `i3`, ella incluida.

- Tratar la interrupción según del tipo que sea.

En el ejemplo de la Figura 2.53, una vez que la interrupción sea tratada, el estado de la máquina podrá reestablecerse en el punto previo a la ejecución de `i4`, es decir, tras la conclusión de `i3` que fue la última instrucción que se terminó. La ejecución del programa se reiniciaría desde la `i4`, extrayendo las instrucciones de la I-caché.

Aunque no se ha hecho mucho hincapié hasta ahora, observe la importancia que tiene para lograr interrupciones precisas que al terminar una instrucción se actualice su registro arquitectónico con el valor del registro de renombramiento que se le asignó al distribuirla, aunque el renombramiento ya no esté reflejado en el ARF. Esta situación se plantea si el acceso al RRF es indexado ya que el renombramiento puede desaparecer del ARF al ser sobrescrito por otro posterior. Por esa razón se mantiene un campo `Rr` en el ROB aunque si el acceso al RRF es asociativo no sería necesario ya que en el RRF están reflejados todos los renombramientos. Si la actualización del ARF solo se realizase para aquellas instrucciones con registros destino renombrados reflejados en el ARF, el tratamiento de los riesgos WAR y WAW seguiría siendo correcto pero, sin embargo, no habría interrupciones precisas. Si no hubiese interrupciones ni actualización del ARF en el ejemplo de la Figura 2.53, el valor que permanecería en `R4` tras concluir la ejecución del programa sería el producido por `i6`, no el de `i3`, lo cual es correcto. Pero si hubiese una interrupción en `i4`, como es el caso, no podría recuperarse el estado ya que el valor de `R4` no habría sido actualizado por `i3` sino por `i6`. La actualización es posible gracias a que la correspondencia entre registro destino y registro de renombramiento siempre se mantiene en el ROB.

Cuando no es una instrucción del programa durante la ejecución la que lanza la interrupción es posible realizar un tratamiento del problema ligeramente diferente en función de dónde suceda la interrupción. Si la interrupción ocurriese en la etapa IF por un fallo de página de memoria virtual al intentar leer una instrucción, no se leerían nuevas instrucciones y se terminarían todas las instrucciones ya existentes en el cauce. Si la interrupción se produjese en la etapa de decodificación debido a un código de operación ilegal o indefinido, se terminaría la ejecución de las instrucciones alojadas en el buffer de distribución y siguientes.

2.12. Limitaciones de los procesadores superescalares

El aumento de prestaciones que se puede obtener en un procesador superescalar explotando exclusivamente el paralelismo a nivel de instrucción está restringido, principalmente, por dos factores:

- La complejidad y coste de la fase de distribución de las instrucciones y el análisis de las dependencias existentes entre ellas.
- El grado de paralelismo intrínseco existente en el flujo de instrucciones de un programa.

Los programas presentan distintos grados de paralelismo intrínseco. En muchos casos, las instrucciones no dependen unas de otras y pueden ejecutarse simultáneamente sin ninguna restricción. En otros casos, existen restricciones entre ellas: una instrucción afecta al resultado de otra instrucción (dependencias RAW de datos) o al número de recursos disponibles con que cuenta (dependencias estructurales), lo que restringe el paralelismo entre ellas. Pero, además, a medida que el número de instrucciones que pueden emitirse simultáneamente se incrementa, el coste de la gestión de las dependencias de datos se incrementa rápidamente, y se agrava por el hecho de tener que realizarlo en tiempo de ejecución y a la velocidad que marca la CPU. Este coste no solo viene dado por la complejidad de la lógica necesaria para realizar el análisis de las dependencias, sino también por el hardware para implementarla y los consiguientes retardos de comunicación. Se ha demostrado matemáticamente que en un procesador superescalar con recursos hardware limitados y un ancho de emisión de k instrucciones, la complejidad de la lógica de distribución/emisión de instrucciones es de orden n^k y el retardo es de orden $k^2 \log n$, donde n es el número de instrucciones del repertorio. Debido a los cientos o miles de millones de transistores que contienen los procesadores actuales en un único chip, los problemas de los retardos de comunicación son críticos. Como forma de paliar esos retardos ya hay procesadores que incorporan etapas en su segmentación exclusivamente dedicadas a tareas de sincronización.

Aunque un programa no contenga dependencias entre las instrucciones que lo componen, un procesador superescalar siempre tiene que comprobar su existencia. No hacerlo conduciría a una posible generación de resultados incorrectos ante la aparición de una única dependencia. Con independencia de la velocidad del reloj o de la tecnología de fabricación, el examen obligado de las dependencias establece un límite práctico en el número de instrucciones que pueden ser simultáneamente emitidas. Sin embargo, aunque la lógica para examinar las dependencias fuese infinitamente rápida, el flujo de instrucciones siempre tiene dependencias lo cual limita el número de instrucciones a emitir. Es decir, el grado de paralelismo intrínseco a nivel de instrucciones constituye una limitación.

Estos dos límites en el rendimiento de los procesadores superescalares, unidos a la creciente necesidad de mayor potencia de procesamiento, impulsan la investigación y el desarrollo de nuevas arquitecturas de procesadores. En la actualidad, se pueden diferenciar dos grandes líneas de trabajo en el diseño de nuevos procesadores: una enfocada a aumentar el paralelismo a nivel de instrucción (ILP) y otra orientada a incrementar el paralelismo a nivel de hilo (*Thread-Level Paralelism* - TLP). Existe una tercera línea de trabajo, el aumento del paralelismo a nivel de proceso, pero se puede considerar

como más orientada al desarrollo de multicomputadores. Los multiprocesadores mejoran el rendimiento explotando tanto el paralelismo a nivel de hilo como el paralelismo a nivel de programa.

Entre las arquitecturas de computador basadas en explotar el ILP destacan las VLIW (*Very Long Instruction Word*), las EPIC (*Explicitly Parallel Instruction Computing*), los procesadores multiescalares o las mejoras introducidas en los procesadores superescalares. Actualmente, la tendencia para mejorar las prestaciones de un procesador superescalar pasa por su escalabilidad: aumento del ancho y profundidad de las segmentaciones, aumento del tamaño y niveles de las memoria caché, aumento de la frecuencia de reloj, todo ello unido a avanzadas técnicas especulativas que mejoren el flujo de control (por ejemplo, predictores de saltos híbridos) y que permitan superar el límite del flujo de datos que imponen las dependencias RAW (por ejemplo, mediante la ejecución de instrucciones utilizando valores especulativos de los operandos). Otra mejora por la que optan muchos procesadores superescalares es el procesamiento vectorial, es decir, la inclusión dentro del procesador de una o varias unidades funcionales para el procesamiento de instrucciones SIMD.

Para aumentar el paralelismo a nivel de hilo se trabaja en procesadores SMT (*Simultaneous MultiThreading*) y CMT (*Chip MulTiprocessors*). Estos dos tipos de procesadores permiten aprovechar el paralelismo a nivel de hilos de dos formas distintas pero no excluyentes. Un procesador SMT dispone de recursos que permiten la ejecución en paralelo de instrucciones pertenecientes a hilos diferentes. Un procesador CMT consiste en colocar varios procesadores similares o núcleos en un único circuito integrado pudiéndose ser estos procesadores a su vez procesadores superescalares, procesadores SMT o procesadores VLIW.

2.13. Resumen: Una visión global del núcleo de ejecución dinámica

En este capítulo se ha estudiado el funcionamiento de un procesador superescalar describiendo las diferentes etapas por las que una instrucción transcurre hasta que completa su procesamiento y, simultáneamente, analizando las técnicas que se pueden aplicar para resolver los diferentes problemas que se plantean en cada etapa. A lo largo todo el capítulo se ha realizado una descripción incremental, casi segmentada, describiendo los problemas que se resuelven en cada etapa e indicando mediante ejemplos los que han quedado pendientes de resolver en las siguientes. Con este enfoque se ha pretendido que el lector se haya ido percatando por sí mismo de los problemas a los que han tenido que enfrentarse los ingenieros de computadores en estas últimas décadas hasta dar lugar a los procesadores actuales que implementan los computadores como el que se ha utilizado para escribir este libro de texto (un Intel Core 2 Duo a 2.2 GHz).

Es cierto que los procesadores reales son infinitamente más complicados ya que contienen y gestionan mucha más información de estado que la utilizada en las explicaciones de este texto, en donde se pretende ofrecer una visión didáctica y comprensible. Sin embargo, no es menos cierto, que esta visión simplificada, motivada porque se utiliza una imagen de baja resolución del procesador, permite conocer con claridad las principales tareas que realiza cualquier procesador superescalar. Pueden existir y existirán procesadores superescalares que cambien la forma en que se implementa una u otra etapa, que

incluyan una u otra técnica para afrontar los riesgos, que se organicen internamente de forma diferente, pero en la actualidad, los procesadores superescalares responden al modo de funcionamiento descrito en este texto. Puede que un procesador lea los operandos en la etapa de decodificación, otro en la de distribución pero la lectura del operando hay que realizarla y, por el momento, hay que renombrar los registros para respetar las dependencias de datos falsas.

Para finalizar el capítulo, se plantea un resumen de los pasos que se dan en cada etapa considerando que el procesador realiza planificación dinámica con lectura de operandos al distribuir las instrucciones:

- Lectura.
  - Se extrae el grupo de lectura de la I-caché.
  - Si es un salto, se especula el resultado y la dirección de destino y se cambia el contador de programa según la especulación.
- Decodificación.
  - Se decodifican las instrucciones en paralelo.
  - Se envían al buffer de distribución.
- Distribución.
  - Si hay espacio en las estaciones de trabajo, en el buffer de terminación y en el RRF se distribuye la instrucción.
  - Se leen los operandos (identificadores o valores) desde el ARF o el RRF.
  - Se introduce la instrucción en el buffer de reordenamiento.
  - Se renombra el registro destino de la instrucción.
  - Se introduce la instrucción en la estación de reserva que corresponda.
  - Las estaciones de reserva examinan los buses de reenvío en busca de coincidencias de identificadores de registro destino.
  - Si todo los operandos fuente se encuentran disponibles, se emite la instrucción a la unidad funcional. Por lo general, se libera la entrada que ocupaba en la estación de reserva.
- Ejecución.
  - Se cambia el estado de la instrucción en el buffer de reordenamiento.
  - Si hay una interrupción, se marca el campo correspondiente en el buffer de terminación.
  - Si es una instrucción de salto se comprueba que la especulación coincida con el resultado del salto. Si es así, se validan las instrucciones especuladas; en caso contrario, se invalidan.

  - Cuando finaliza la ejecución, se publica el resultado de la instrucción y el identificador del registro destino en los buses de reenvío para permitir las actualizaciones del RRF, de las entradas de las estaciones de reserva y del buffer de reordenamiento. Si es un almacenamiento, se introduce en el buffer de almacenamiento y se marca como finalizado.
- Terminación.
  - Se cambia el estado de la instrucción en el buffer de reordenamiento.
  - Si no es una instrucción interrumpida, se libera su entrada del buffer de reordenamiento y se da por terminada arquitectónicamente.
  - Se actualiza el ARF desde el RRF y se libera el registro de renombramiento.
  - Si es un almacenamiento, se marca en el buffer de almacenamiento como terminado.
  - Si es una instrucción interrumpida, se eliminan ella y todas las siguientes del buffer sin actualizar el ARF, estén o no finalizadas. También se eliminan los almacenamientos del buffer de almacenamiento y las cargas del buffer de cargas especuladas (en caso de que exista) que sean posteriores a la instrucción interrumpida.
  - Si es una instrucción especulada, queda en espera de terminación.
  - Si es una instrucción invalidada, se termina sin actualizar el ARF.
- Retirada (solo almacenamientos).
  - Esperar a que el bus de acceso a memoria esté libre y realizar la escritura.
  - La instrucción se retira del buffer de almacenamiento.

2.14. Referencias

AMD *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, 2007.

Cotofana S., Vassiliadis S., On the Design Complexity of the Issue Logic of Superscalar Machines, *Proceedings of the 24th Conference on EUROMICRO*, Vasteras, Suecia, pp. 10277-10284, 1998.

Hennessy J.L., Patterson D.A. *Computer Architecture. A Quantitative Approach*, 4ª edición, Morgan Kaufmann, San Francisco, California, 2007.

IBM Corporation *User Manuals for Power PC 970FX & 970MP processors*, 2008.

Keltcher C.N. et al. The AMD Opteron processor for multiprocessor servers, *IEEE Micro*, vol. 23, nº 2, pp. 66-76, 2003.

Kessler R. E. The Alpha 21264 Microprocessor, *IEEE Micro*, vol. 19, nº 2, pp. 24-36, 1999.