Tema 7

Procesadores Superescalares: Paralelismo Implícito a Nivel de Instrucción

Se denomina arquitectura superescalar a aquella implementación capaz de ejecutar más de una instrucción por ciclo de reloj. Para ello se usan múltiples cauces, con lo que varias instrucciones pueden iniciar su ejecución de manera independiente. El término superescalar se emplea como contraposición a la arquitectura escalar, que solo es capaz de ejecutar una instrucción por ciclo de reloj.

En la figura 7.1 se representa un esquema de la organización superescalar. Se tienen múltiples unidades funcionales que admiten la ejecución en paralelo, y por separado, de varias instrucciones. Sin embargo, nótese que, en general, cada unidad funcional no tiene por qué admitir la ejecución de cualquier instrucción, hay dos unidades funcionales que pueden ejecutar instrucciones de operaciones con enteros, dos de instrucciones de operaciones con coma flotante, y una de instrucciones de transferencia con memoria (operaciones de carga y almacenamiento). Así, en este caso, pueden estar ejecutándose al mismo tiempo dos operaciones con enteros, dos operaciones con coma flotante y una operación de transferencia a memoria.

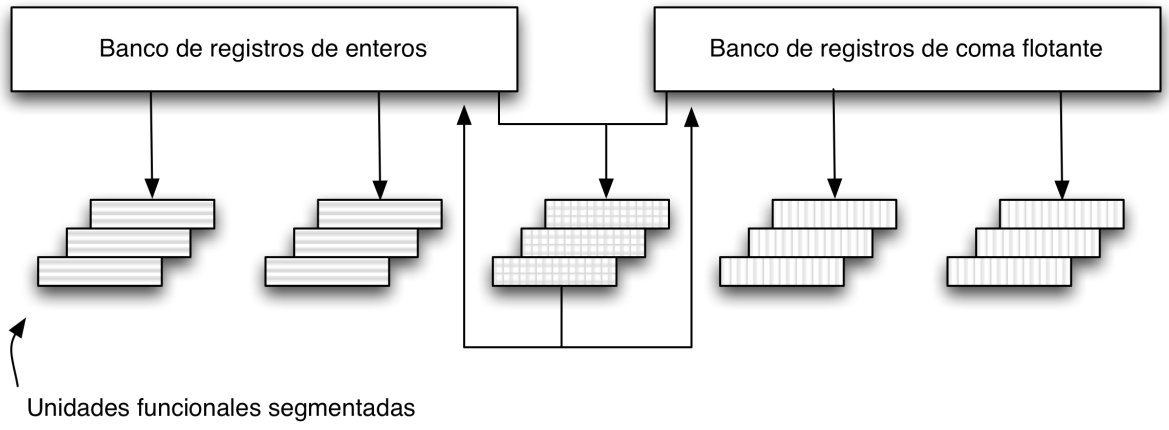


Figura 7.1 Ejemplo de organización superescalar.

El diseño superescalar es compatible con la segmentación. Esto es, como muestra la figura 7.1, las unidades funcionales pueden realizarse como cauces segmentados.

El concepto de diseño superescalar apareció poco después que la arquitectura RISC. Si bien se puede realizar una implementación superescalar tanto de una arquitectura RISC como de una arquitectura CISC, la arquitectura de conjunto reducido de instrucciones de una máquina RISC se presta mejor a la utilización de técnicas superescalares.

Lo esencial del diseño superescalar es su habilidad para ejecutar instrucciones de manera independiente en diferentes cauces. Un procesador superescalar es capaz de ejecutar más de una instrucción, solo si estas no presentan algún tipo de dependencia, como veremos más adelante.

La inmensa mayoría de las CPUs desarrolladas desde 1998 son superescalares.

En este tema se presenta una introducción al diseño superescalar con paralelismo implícito, examinando los diferentes tipos de dependencia de datos, políticas de emisión y otras cuestiones relacionadas con el diseño de la implementación superescalar.

7.1 Explotación del paralelismo a nivel de instrucción

El hecho de que la implementación superescalar tenga varios cauces introduce un nuevo nivel de paralelismo. Los procesadores superescalares sacan provecho del *paralelismo a nivel de instrucción*, que hace referencia al grado en el que, en promedio, las instrucciones de un programa se pueden ejecutar en paralelo. Para ello implementan técnicas que permiten romper el flujo secuencial de instrucciones de un programa, para simultanear la ejecución de varias, en el mismo procesador.

No hay procesador moderno que no use alguna forma de paralelismo a nivel de instrucción. Los cauces segmentados, escalares y supersegmentados son diferentes técnicas que utilizan el paralelismo a nivel de instrucción.

En este tema se estudiarán las características de las implementaciones superescalares que explotan el paralelismo a nivel de instrucción de forma implícita, internamente. Es decir, aquellos procesadores superescalares cuyo hardware implementa las técnicas que permiten romper el flujo de instrucciones del programa, para ejecutarlas en paralelo, incluso de forma desordenada, y lo hacen de manera transparente al programador, y en tiempo de ejecución.

7.2 Diferencias entre superescalar y supersegmentación

Si bien el diseño superescalar es compatible con la supersegmentación, no son técnicas similares. Por ello, se dedica este apartado a la descripción de la supersegmentación. Y es que la palabra supersegmentación nos lleva a la tentación de aplicar la ecuación *superescalar + segmentado = supersegmentado*, cuando en realidad la ecuación correcta sería *segmentado + segmentado = supersegmentado*.

La supersegmentación es una técnica que saca provecho del paralelismo a nivel de instrucción con el objeto de alcanzar mayores prestaciones.

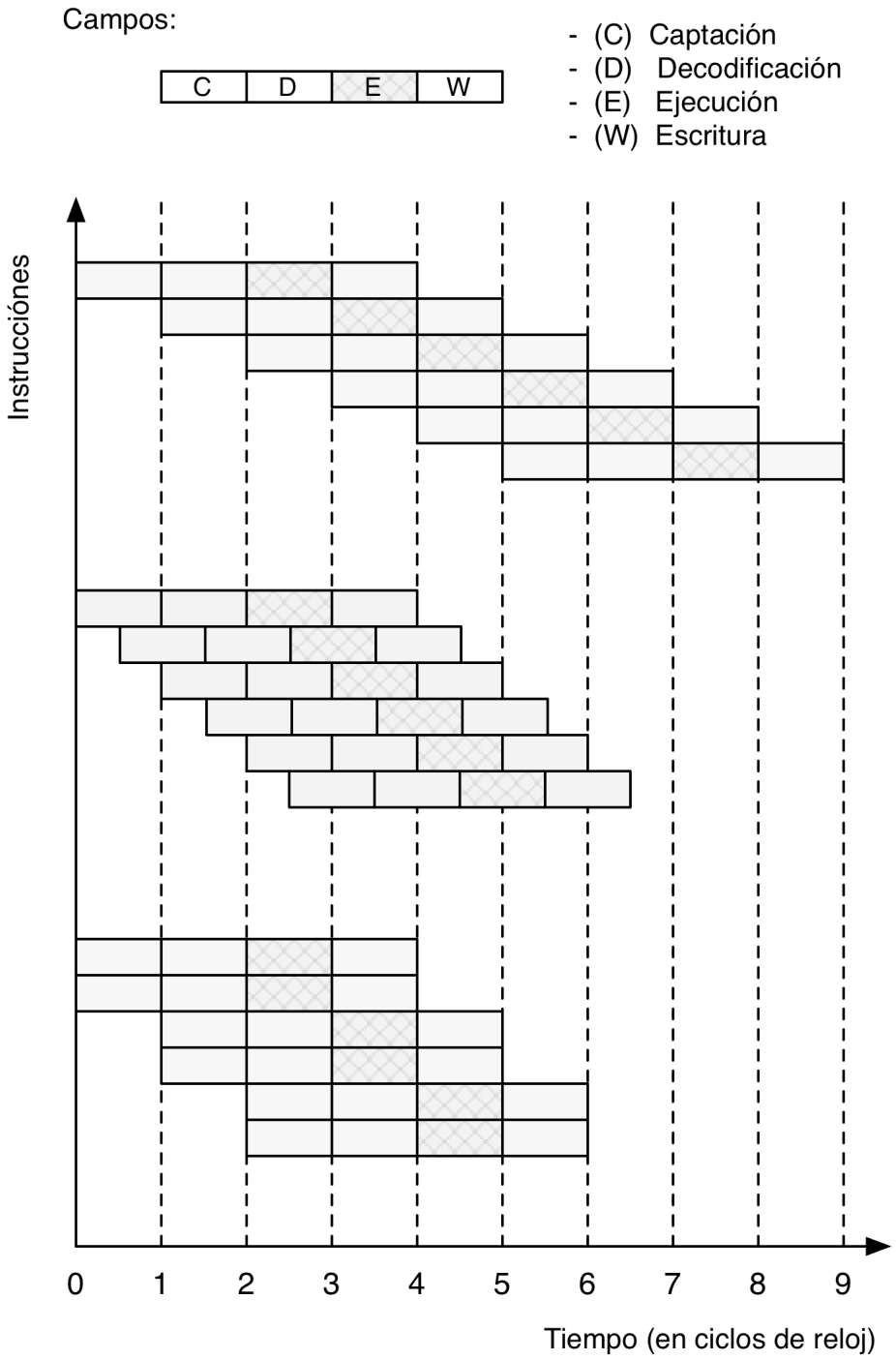


Figura 7.2 Comparación entre diversas aproximaciones.

La supersegmentación consiste en segmentar algunas de las etapas (en concreto las más lentas) de los procesadores segmentados, en dos o más etapas de forma que se permita que existan dos instrucciones a la vez dentro de la misma etapa y unidad funcional sin tener que replicar ésta. (la réplica constituiría superescalaridad). Así, un procesador supersegmentado es aquel que aplica dos veces el concepto de segmentación, la primera a nivel global, y la segunda a nivel interno de sus unidades funcionales.

En la figura 7.2, tenemos varios diagramas que muestran las aproximaciones superescalar, segmentada y supersegmentada, con el propósito de poder compararlas y establecer diferencias.

El diagrama superior ilustra un cauce normal segmentado. Éste se usa como base de la comparación. Está formado por cuatro etapas, de forma que puede ejecutar una de ellas en cada ciclo. La etapa de ejecución se diferencia mediante una trama. En este cauce solo hay una instrucción en la etapa de ejecución en cada instante.

Otro de los diagramas representados es un cauce supersegmentado. Como puede verse, es capaz de ejecutar dos instrucciones por ciclo de reloj. Sin embargo esto no implica superescalaridad. La supersegmentación puede entenderse aquí como el resultado de dividir las funciones realizadas en cada etapa en dos partes iguales, no solapadas, cada una de las cuales se ejecuta en 1/2 ciclo de reloj. La implementación de un cauce supersegmentado de este tipo, se denomina de grado 2. También son posibles implementaciones supersegmentadas de mayor grado.

De esta forma, dividiendo cada etapa en 2, el ciclo de reloj de periodo T se verá reducido a la mitad, T/2, doblando así la capacidad del cauce. Nótese que la supersegmentación de la figura provoca un desdoble de la señal de reloj al doble de la frecuencia que en los demás diagramas. En general, la supersegmentación lleva asociada la subdivisión interna del ciclo de reloj base en ciclos supersegmentados, lo cual implica diseños que trabajan a muy alta frecuencia.

El diagrama de la figura presenta un cauce superescalar, una implementación capaz de ejecutar dos instrucciones en paralelo en cada etapa, en unidades funcionales diferentes, por supuesto. La implementación de un cauce superescalar de este tipo se denomina de grado 2. También son posibles implementaciones superescalares de mayor grado.

7.3 Limitaciones fundamentales del paralelismo

La aproximación superescalar introduce la capacidad de ejecutar múltiples instrucciones en paralelo. Sin embargo, no se puede realizar le ejecución en paralelo de instrucciones de forma indiscriminada, debido a la existencia de ciertas limitaciones fundamentales del paralelismo a las que el sistema tiene que enfrentarse. Estas son:

- Dependencia de datos verdadera.

- Dependencia relativa al procedimiento.

- Conflictos con los recursos.

- Dependencia de salida.

- Antidependencia.

A continuación se describen las tres primeras limitaciones. Las demás limitaciones se verán posteriormente, pues su descripción necesita de algunos desarrollos previos.

7.3.1 Dependencia de datos verdadera

Se produce una dependencia verdadera de datos entre dos instrucciones cuando una instrucción no se puede ejecutar hasta que finalice la ejecución de la otra.

Considérese la secuencia siguiente:

```
add r1,r2       ; cargar el registro r1 con el contenido de r2 más el contenido de r1
move r3,r1      ; cargar el registro r3 con el contenido de r1
```

Como vemos, la segunda instrucción necesita un dato producido por la primera, por lo que debe esperar a que concluya la ejecución de la primera para que esté listo el dato que necesita. Así, ambas instrucciones no pueden ejecutarse en paralelo. En este caso se dice que existe una *dependencia verdadera de datos* entre ambas instrucciones (también llamada *dependencia de flujo* o *dependencia de escritura/lectura*).

El primer diagrama representado en la figura 7.3 ilustra la ejecución de un par de instrucciones independientes, es decir, aquellas que carecen de dependencias de datos entre ellas y, por tanto, pueden ejecutarse en paralelo. Como puede observarse, la ejecución de ambas es simultánea.

Del diagrama de la figura 7.3 correspondiente a la ejecución de instrucciones con dependencia de datos se puede observar que, aunque ambas instrucciones puedan ser captadas y decodificadas simultáneamente, el hecho de que la segunda instrucción necesite un dato producido por la primera fuerza el retraso en la ejecución de la segunda hasta que estén disponibles todos sus valores de entrada necesarios. En esta secuencia, la primera instrucción es una operación de suma, que podemos asumir que tarda un ciclo de reloj en ejecutarse, y por ello introduce un retraso de un ciclo en la ejecución de la siguiente instrucción.

Consideremos ahora la secuencia siguiente:

```
load r1, ef     ; cargar el registro r1 con el contenido de la dirección de memoria
move r3,r1      ; cargar el registro r3 con el contenido de r1
```

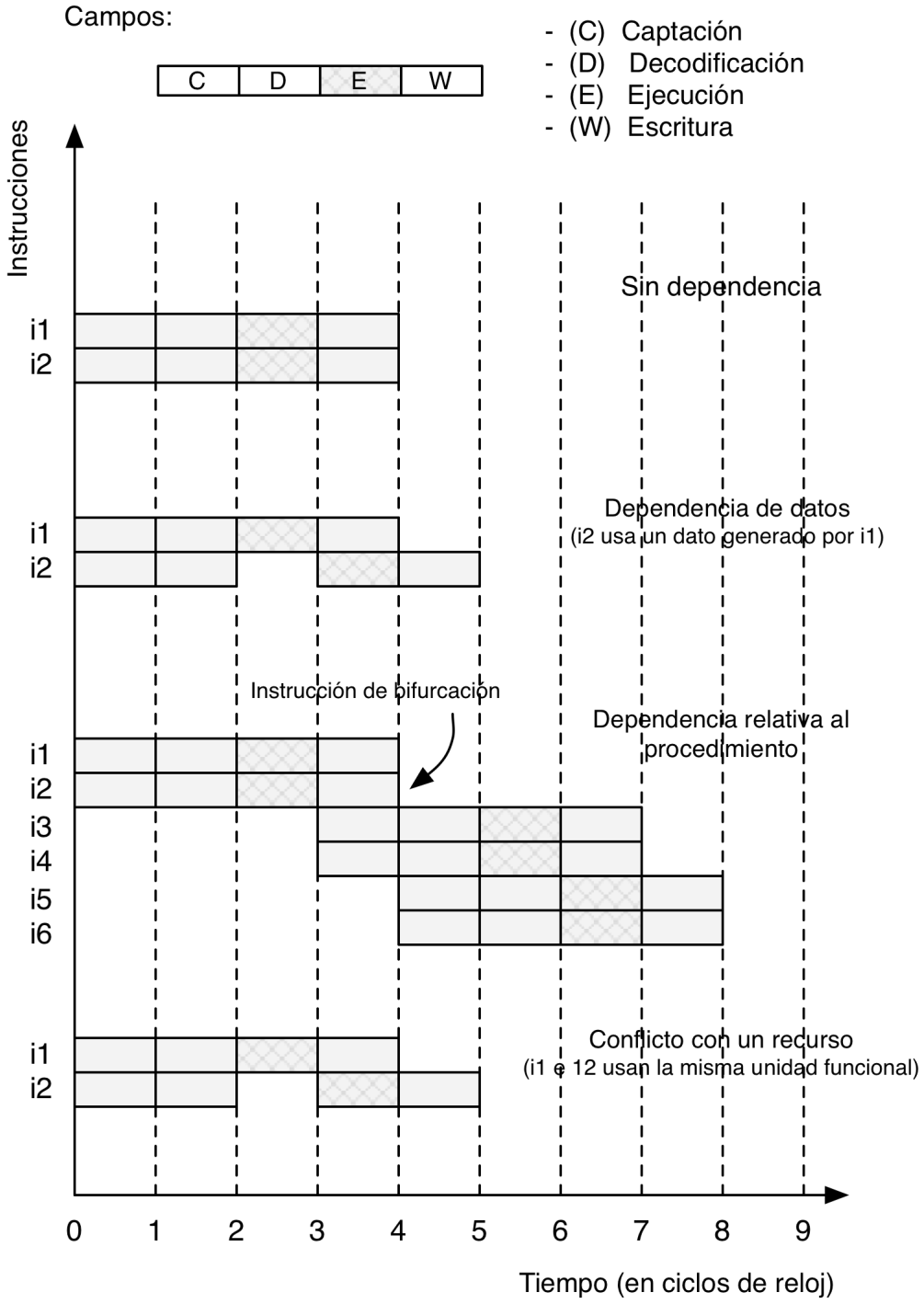


Figura 7.3 Comparación entre diversas aproximaciones.

En esta secuencia la primera instrucción es una carga desde memoria, la cual causará un retraso para la ejecución de la siguiente instrucción. En el mejor de los casos se tardará dos o más ciclos de reloj, y eso suponiendo que el dato se encuentra en la memoria caché. Si el dato no estuviera en caché el retraso de la segunda instrucción sería mayor, tantos ciclos como se necesite para cargar el dato de memoria a registro.

Para atenuar este retraso, se puede utilizar la carga retardada, que se introdujo durante el tema anterior. Sin embargo, esta técnica es menos efectiva en un cauce superescalar, pues las instrucciones independientes que se efectúan durante la carga posiblemente se ejecuten en un ciclo, dejando al procesador parado hasta que concluya la carga.

7.3.2 Dependencias relativas al procedimiento

Como sabemos, la presencia de una bifurcación (una instrucción de salto condicional) puede afectar al funcionamiento del cauce. Las instrucciones que siguen a la instrucción de la bifurcación tienen una dependencia relativa al procedimiento. Esto es, como dicho salto puede realizarse o no (dependiendo del flag de condición), las instrucciones que siguen a la bifurcación no pueden ejecutarse hasta que se conozca el camino que tomará el flujo del programa. Suponiendo la existencia de predicción de saltos, el peor de los casos supone un fallo en la predicción del salto una vez conocido el camino de la bifurcación. En este caso, figura 7.3, se tienen varios ciclos de penalización por regularización del cauce, que comenzará a llenarse, tras la ejecución de la bifurcación, captando las instrucciones del camino correcto. En el mejor de los casos, ante un acierto en la predicción del salto, las instrucciones que siguen a la bifurcación pueden haber sido captadas y decodificadas, pero su ejecución sufrirá el retraso necesario hasta que se conozca el camino de la bifurcación.

Aunque las dependencias relativas al procedimiento afectan tanto a cauces escalares como superescalares, las consecuencias de estas dependencias en cauces superescalares son más graves, ya que por cada ciclo de retraso se pierde un mayor número de oportunidades de ejecutar instrucciones.

Si se usan instrucciones de longitud variable, características de la arquitectura CISC, surge otro tipo de dependencia de datos relativa al procedimiento. Al no conocerse la longitud de una instrucción, es necesario realizar una decodificación, al menos parcial, para conocer cual es el comienzo de la siguiente instrucción antes de captarla. Ello impide la captación simultánea necesaria en un cauce superescalar. Esto se evitaría si todo el repertorio de instrucciones tuviera una longitud fija. Ésta es la razón por la que las técnicas superescalares se aplican más fácilmente a las arquitecturas RISC, que tienen longitud fija.

7.3.3 Conflictos con los recursos

Un conflicto en un recurso es una pugna de dos o más instrucciones por el mismo recurso, al mismo tiempo. Ejemplos de recursos son: puertos del banco de registros, unidades funcionales, cachés, etc.

Como puede verse en la figura 7.3, cuando se produce un conflicto con un recurso, p.ej. ambas instrucciones necesitan la misma unidad funcional para ejecutarse, sólo puede ejecutarse una instrucción mientras que la otra debe esperar para poder acceder al recurso y poder ejecutarse. Nótese que desde el punto de vista del cauce, este conflicto presenta el mismo comportamiento que la dependencia verdadera de datos. Sin embargo, no es una dependencia

verdadera, porque se puede evitar duplicando recursos, mientras que las dependencias verdaderas no se pueden evitar.

7.4 Paralelismo a nivel de instrucción y paralelismo a nivel de la máquina

Ya se ha hecho referencia anteriormente al **paralelismo a nivel de instrucción** definiéndolo como el grado en el que, en promedio, las instrucciones de un programa se pueden ejecutar en paralelo. Éste paralelismo se puede explotar cuando las instrucciones de una secuencia de programa son independientes, y por ello pueden solapar su ejecución en paralelo.

Nótese que este paralelismo depende de la secuencia de instrucciones del programa. Como ejemplo de este concepto consideremos los siguientes fragmentos de código:

```
Load  R1 <-- R2          Add  R3 <-- R3, "1"
Add  R3 <-- R3, "1"      Add  R4 <-- R3, R2
Add  R4 <-- R4, R2       Store  [R4] <-- R0
```

Las tres instrucciones de la secuencia de la izquierda son independientes, nótese que usan registros destino diferentes y por ello podrían ejecutarse en cualquier orden sin que ello variara el resultado. De la misma manera, también podrían ejecutarse todas a la vez, en paralelo, sin alterar el resultado. Por otra parte, la secuencia de instrucciones de la derecha no puede ejecutarse en paralelo. Nótese que la segunda instrucción usa el resultado de la primera y, a su vez, la tercera instrucción usa el resultado de la segunda.

Así, en la secuencia izquierda existe paralelismo a nivel de instrucción, mientras que en la secuencia derecha no existe.

El paralelismo a nivel de instrucción de un programa depende de la frecuencia con que ocurren las dependencias verdaderas de datos y las dependencias relativas al procedimiento que haya en el programa.

El **paralelismo de la máquina** es una medida de la capacidad del procesador para sacar partido del paralelismo a nivel de instrucciones. El paralelismo de la máquina dependerá del número de instrucciones que se pueden captar y ejecutar al mismo tiempo, y de la capacidad del procesador para localizar instrucciones independientes.

Se denomina **grado de paralelismo** al número máximo de instrucciones que se puede ejecutar en paralelo.

Es importante diferenciar bien los conceptos de paralelismo a nivel de instrucción y paralelismo de la máquina, y la relación entre ellos. Ambos son factores importantes para incrementar las prestaciones de la ejecución. Un programa que tenga un bajo nivel de paralelismo a nivel de instrucción (es decir, una gran cantidad de dependencias entre sus instrucciones) no podrá sacar el máximo partido de una máquina con un alto grado de paralelismo, quedando infrautilizada dicha máquina. Por otra parte, una máquina con un bajo

grado de paralelismo limitará las prestaciones de la ejecución de un programa con un alto nivel de paralelismo a nivel de instrucción.

Para maximizar el paralelismo a nivel de instrucción de un programa, los compiladores realizan ciertas optimizaciones y los procesadores implementan en hardware ciertas técnicas, algunas de las cuales veremos a continuación.

7.5 Cuestiones relacionadas con el diseño

7.5.1 Políticas de emisión de instrucciones

El paralelismo de la máquina no se logra simplemente replicando varias veces cada etapa del cauce. El procesador debe ser capaz de analizar el flujo de instrucciones del programa e identificar el paralelismo a nivel de instrucción. También tiene que organizar la captación, decodificación y ejecución de las instrucciones.

Se denomina *emisión de instrucciones* al proceso de iniciar la ejecución de la ejecución de instrucciones en las unidades funcionales.

Se denomina *política de emisión de instrucciones* al protocolo utilizado para emitir instrucciones.

El procesador va a intentar localizar instrucciones independientes más allá del punto de ejecución en curso, que puedan introducirse en el cauce para ser ejecutadas en parlelo. Respecto a esto, son importantes tres ordenaciones:

- El orden en que se captan las instrucciones.
- El orden en que se ejecutan las instrucciones.
- El orden en que las instrucciones actualizan los contenidos de registros y posiciones de memoria.

Cuanto más sofisticado sea un procesador, menos limitado estará por la estrecha relación entre estas ordenaciones. El procesador puede alterar cualquiera de los órdenes anteriores con respecto al orden que tendrían secuencialmente. La única restricción es que el resultado debe ser correcto.

Las políticas de emisión de instrucciones de los procesadores superescalares se pueden clasificar en:

- Emisión en orden y finalización en orden.
- Emisión en orden y finalización desordenada.
- Emisión desordenada y finalización desordenada.

7.5.1.1 Emisión en orden y finalización en orden

Es la política de emisión de instrucciones más sencilla. Consiste en emitir instrucciones secuencialmente, en el orden en que lo haría una ejecución secuencial, y escribir los resultados en el mismo orden. Es una política de emisión tan simple que solo tiene utilidad como base para la comparación de otras aproximaciones.

La figura 7.4 muestra un ejemplo de esta política de emisión de instrucciones. En esta podemos apreciar la representación de un cauce superescalar con dos vías de captación y decodificación de instrucciones, tres unidades funcionales para la ejecución de instrucciones, y dos copias de la etapa de escritura.

El ejemplo mostrado se corresponde a un fragmento de código, del cual se obtienen las siguientes restricciones:

- I1 necesita dos ciclos para ejecutarse.
- I3 e I4 compiten por la misma unidad funcional.
- I5 depende de un valor producido por I4.
- I5 e I6 compiten por una unidad funcional.

Como se puede observar, un procesador que implementa esta política de emisión capta las instrucciones en orden, de dos en dos (porque hay dos vías de captación), pasándolas a la unidad de decodificación. Como las órdenes se captan por parejas y en orden, no se producirá otra captación hasta que se hayan emitido las instrucciones y las etapas de decodificación se encuentren vacías. Para garantizar la finalización en orden, la emisión de instrucciones se detiene temporalmente cuando hay una pugna por una unidad funcional o cuando una instrucción necesita más de un ciclo de reloj para ejecutarse.

Ciclo	Cauce de captación y decodificación		Unidades funcionales de ejecución			Etapa de escritura	
1	I1	I2					
2	I3	I4	I1	I2			
3	I3	I4	I1				
4		I4			I3	I1	I2
5	I5	I6			I4		
6		I6		I5		I3	I4
7				I6			
8						I5	I6

Figura 7.4 Ejemplo de emisión en orden y finalización en orden.

Nótese que no hace falta conocer exactamente cuales son las instrucciones del fragmento de código del ejemplo. Con sólo conocer las dependencias de datos entre ellas, los ciclos que necesitan para ejecutarse y los recursos que necesitan (p.ej. unidades funcionales) para su ejecución, es suficiente para determinar la forma del patrón de emisión, ejecución y escritura.

7.5.1.2 Emisión en orden y finalización desordenada

La finalización desordenada implica que puede haber cualquier número de instrucciones en la etapa de ejecución en un instante dado, hasta alcanzar el máximo grado de paralelismo de la máquina (todas las unidades funcionales ocupadas). La emisión de instrucciones se detiene temporalmente cuando hay una pugna por un recurso, una dependencia de datos o una dependencia relativa al procedimiento. Así, en el ejemplo anterior, figura 7.5, la instrucción I3 puede ejecutarse y concluir antes de que acabe I1. El resultado es el ahorro de un ciclo en la ejecución del fragmento de código.

La finalización desordenada necesita una lógica de emisión de instrucciones más compleja que la finalización en orden.

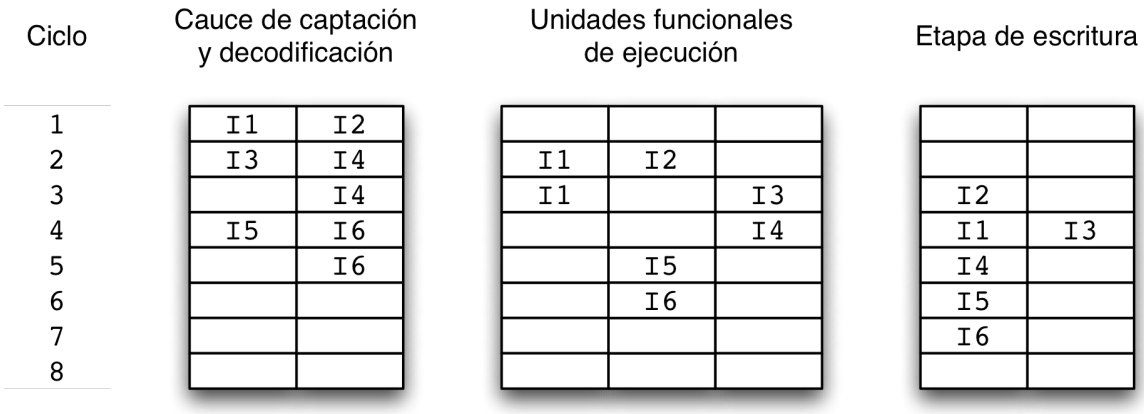


Figura 7.5 Ejemplo de emisión en orden y finalización desordenada.

Además de las limitaciones mencionadas, en este caso hay que tener en cuenta un nuevo tipo de dependencia, denominada *dependencia de salida* (también llamada *dependencia escritura-escritura*). Para ilustrar esta dependencia consideremos el siguiente fragmento de código (*op* hace referencia a cualquier operación):

```
I1: r3 ← r3 op r5
I2: r4 ← r3 + 1
I3: r3 ← r5 + 1
I4: r7 ← r3 op r4
```

Podemos observar una dependencia de datos verdadera de la instrucción I2 respecto de la I1 (I2 necesita operar con un dato que produce I1). También existe una dependencia de datos verdadera de la instrucción I4 respecto de la I3., y otra de la instrucción I4 respecto de la I2. Sin embargo, no parece haber dependencia, tal como se han definido hasta ahora, entre I1 e I3. ¿Podrían I1 e I3 ejecutarse en paralelo? Obsérvese que si estas se ejecutan en paralelo puede ocurrir:

- Si I3 finaliza antes que I1, la instrucción I2 utilizará un valor incorrecto contenido en R3.
- Si I1 finaliza antes que I3, la instrucción I4 utilizará un valor incorrecto contenido en R3.

La ejecución de este fragmento de código sería posible realizarla con I1 en el primer ciclo, I2 e I3 en el segundo ciclo, e I4 en un tercer ciclo.

7.5.1.3 Emisión desordenada y finalización desordenada

La emisión en orden tiene una seria limitación, el procesador decodifica instrucciones hasta que aparece una dependencia o conflicto. A partir de dicho punto no se decodificarán más instrucciones hasta que se resuelva dicho conflicto. Por ello, el procesador no puede seguir buscando aquellas instrucciones independientes que pudieran haber más allá de este punto y que podrían ser ejecutadas, sacando así más partido a la máquina y aumentado la eficiencia de la ejecución. Éste es el concepto de emisión desordenada.

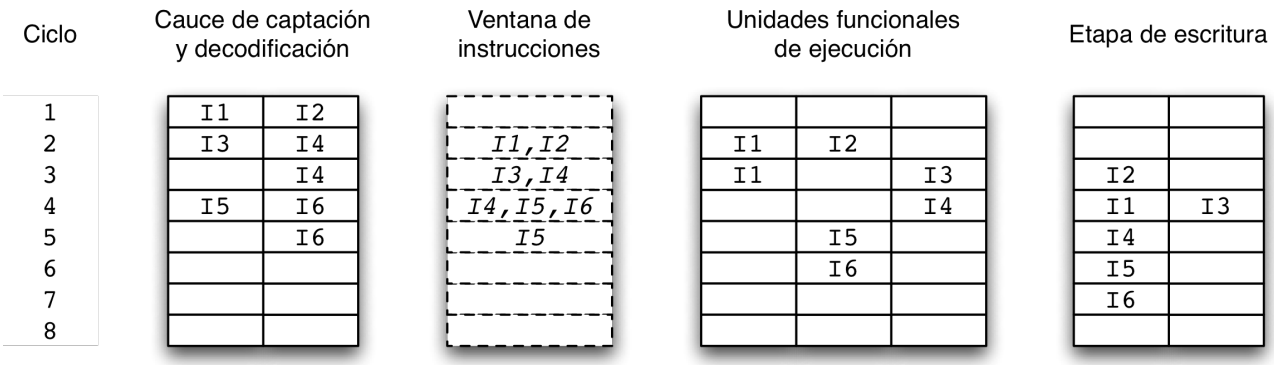


Figura 7.6 Ejemplo de emisión desordenada y finalización desordenada.

Para permitir la emisión desordenada es necesario desacoplar las etapas del cauce de decodificación y ejecución. Esto se hace mediante un buffer llamado *ventana de instrucciones*. Así, cuando un procesador termina la decodificación de una instrucción, coloca ésta en la ventana de instrucciones. Mientras la ventana de instrucciones no se llene, las vías de captación y decodificación pueden captar y decodificar libremente, sin verse condicionadas por la etapa de ejecución.

Cuando una unidad funcional queda disponible en la etapa de ejecución se puede emitir una instrucción desde la ventana de instrucciones a dicha unidad funcional. Podrá emitirse cualquier instrucción que a) necesite y pueda utilizar la unidad funcional que está disponible, b) no la bloqueen ningún conflicto o dependencia.

El resultado de esta organización es que el procesador tiene capacidad de anticipación, que le permite examinar las siguientes instrucciones del código para identificar instrucciones independientes que puedan ser ejecutadas. Esta capacidad de anticipación será mayor cuanto mayor sea el tamaño de la ventana de instrucciones, pues al aumentar dicho tamaño es posible realizar la búsqueda de instrucciones independientes sobre un mayor número de instrucciones.

La utilización de la ventana de instrucciones permite que hayan más instrucciones dispuestas a ser emitidas, con lo cual se reduce la probabilidad de que el cauce tenga que pararse, y aumenta la eficiencia de la ejecución.

Las instrucciones se emiten desde la ventana de instrucciones sin tener muy en cuenta el orden de las instrucciones en el programa. La única restricción es que el programa funcione correctamente.

En la figura 7.6 se muestra el funcionamiento de esta política de emisión. Nótese que en cada ciclo se captan y decodifican dos instrucciones. En cada ciclo, también se transfieren dos instrucciones decodificadas a la ventana de instrucciones (siempre que no esté llena). La ventana de instrucciones desacopla las etapas de codificación y ejecución. A partir de la ventana las instrucciones pueden ser emitidas desordenadamente. Podemos ver la ejecución desordenada en la ejecución de I6 antes de I5 (dado que I5 depende de I4, pero no de I6). De este modo, se ahorra un ciclo en la ejecución del código, comparado con la política de emisión en orden y finalización desordenada.

Con esta política de emisión aparece una nueva dependencia denominada *antidependencia* (también *dependencia lectura-escritura*). Consideremos el fragmento de código que se utilizó anteriormente:

```
I1: r3 ← r3 op r5
I2: r4 ← r3 + 1
I3: r3 ← r5 + 1
I4: r7 ← r3 op r4
```

Con esta política aparece la posibilidad de que I3 se ejecute antes que I2. Nótese que si esto ocurre I3 modificaría el registro R3, que es utilizado operando fuente de I2. Por ello, no es posible que I3 se ejecute antes que I2. El término antidependencia hace referencia a que esta limitación es inversa a la dependencia verdadera (en lugar de que la primera produce un valor que usa la segunda, la segunda destruye un valor que usa la primera).

7.5.2 Renombramiento de registros

Como hemos visto, el hecho de permitir la emisión desordenada de instrucciones y la finalización desordenada puede originar dependencias de salida y antidependencias. La naturaleza de estas dependencias es diferente a la de las dependencias de datos verdaderas, que reflejan el flujo de datos a través de un programa y su secuencia de ejecución. Las dependencias de salida y las antidependencias, por otra parte, no son verdaderas dependencias, surgen porque los valores de los registros no pueden reflejar ya la secuencia de valores establecida por el flujo del programa.

Las antidependencias y las dependencias de salida son realmente conflictos de almacenamiento. Son un tipo de conflicto por los recursos en el que varias instrucciones compiten por los mismos registros. En cuyo caso el procesador debe resolver el conflicto deteniendo temporalmente alguna etapa del cauce. Se puede comprender entonces que la frecuencia de aparición de este tipo de instrucciones aumenta con el uso de las técnicas de optimización de registros, que intentan maximizar el uso de los registros.

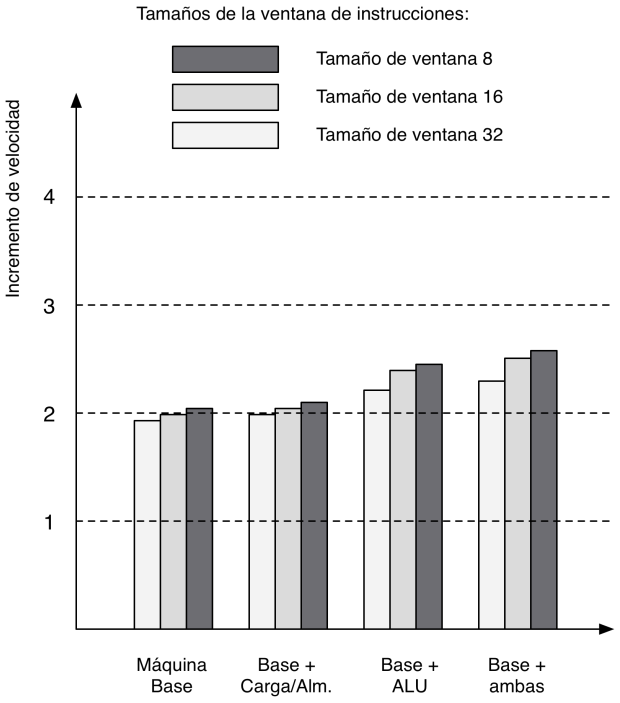
Un método para resolver tales conflictos se basa en la solución tradicional de los conflictos con los recursos: la duplicación de recursos. Esta técnica se denomina *renombramiento de registros*. Consiste en que el hardware del procesador asigne dinámicamente los registros, que están asociados con los valores que necesitan las instrucciones en diversos instantes de tiempo. Cuando se ejecuta una instrucción, donde su operando destino es un registro, se le asigna un nuevo registro físico para almacenar el resultado, y las instrucciones posteriores que accedan a ese valor como operando fuente en ese registro, tienen que atravesar un proceso de renombramiento, donde se revisan las referencias a registros, para que definitivamente hagan referencia al registro físico que contiene el valor que se necesita. De este modo, diferentes instrucciones que tienen referencias a un único registro de la arquitectura (registro lógico), pueden referirse a diferentes registros reales (registros físicos), con valores diferentes.

Para ilustrar esta técnica, vamos a considerar la secuencia de instrucciones utilizada anteriormente:

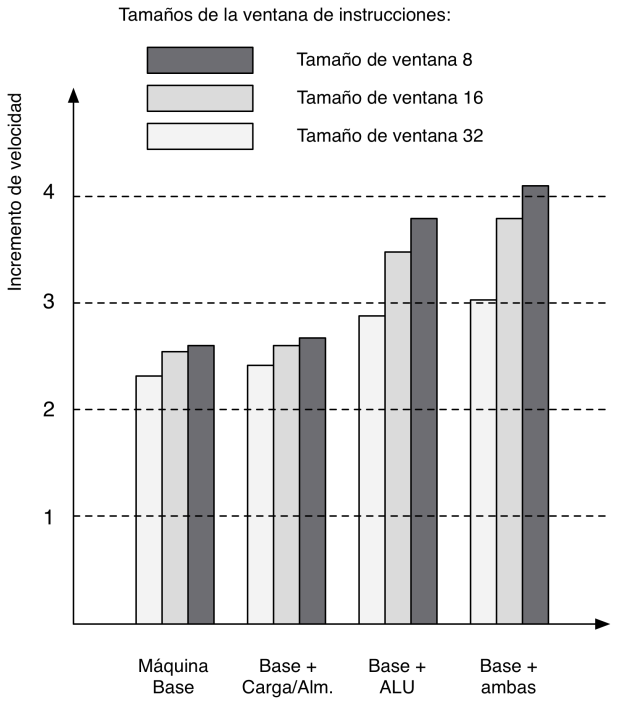
```
I1: r3 ← r3 op r5
I2: r4 ← r3 + 1
I3: r3 ← r5 + 1
I4: r7 ← r3 op r4
```

que, como ya hemos visto, tiene varias dependencias, incluidas una dependencia de salida y una antidependencia. Si aplicamos el renombramiento de registros a dicho código tendremos:

```
I1: r3b ← r3a op r5a
I2: r4b ← r3b + 1
I3: r3c ← r5a + 1
I4: r7b ← r3c op r4b
```



(a) SIN renombramiento de registros



(a) CON renombramiento de registros

Figura 7.7 Mejoras de velocidad para diferentes organizaciones.

Las referencias a un registro sin la letra del subíndice hacen referencia a un registro lógico, un registro de la arquitectura. Las referencias a un registro con la letra del subíndice hacen referencia a un registro físico, un registro hardware.

Nótese que en el ejemplo la creación del registro R3b en la instrucción I3 evita la antidependencia (entre I2 e I3) y la dependencia de salida (entre I1 e I3). El resultado es que utilizando el renombramiento de registros I1 e I3 pueden ejecutarse en paralelo.

7.5.3 Paralelismo de la máquina

Se han estudiado tres técnicas hardware que se pueden utilizar en un procesador superescalar para aumentar sus prestaciones. Estas son: duplicación de recursos, emisión desordenada y renombramiento de registros. El siguiente estudio aclara la relación entre estas técnicas, evitando la influencia de las dependencias relativas al procedimiento.

La figura 7.7 muestra los resultados. Tenemos dos gráficas. En cada una de ellas el eje vertical indica el incremento de velocidad media de la máquina superescalar respecto a la máquina escalar. En el eje horizontal se muestran cuatro organizaciones diferentes. La máquina base es una implementación superescalar que no duplica ninguna de las unidades funcionales, las siguientes organizaciones duplican la unidad de carga y almacenamiento, la ALU, y ambas. La diferencia entre ambas gráficas es que en una de ellas se utiliza renombramiento de registros y en la otra no. Finalmente, cada una de las organizaciones se ha simulado con diferentes tamaños de la ventana de instrucciones.

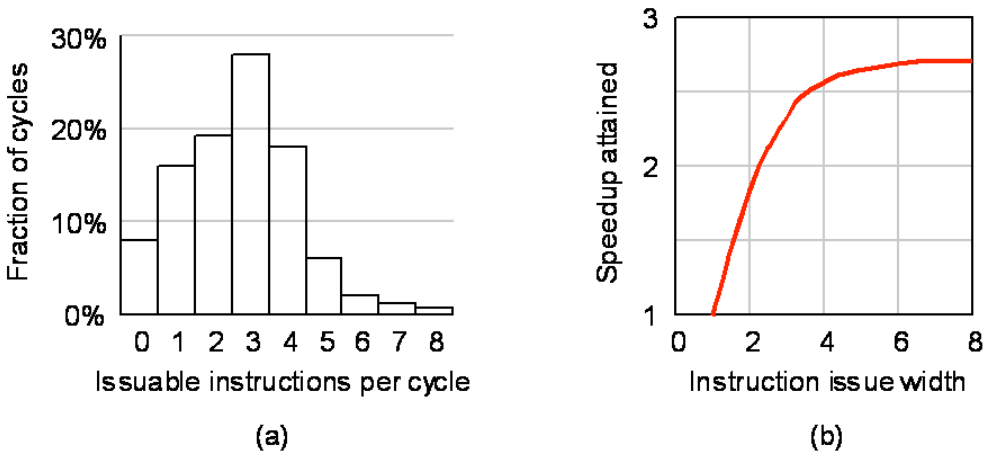


Figura 7.8 Estudio del paralelismo a nivel de instrucciones bajo condiciones ideales.

Este par de gráficas ofrecen importantes conclusiones. La primera proviene de la gráfica que no utiliza renombramiento de registros. Como vemos, es poco importante el incremento de velocidad al duplicar unidades funcionales sin renombramiento de registros. Sin embargo, si se utiliza el renombramiento de registros, se obtienen incrementos significativos al duplicar

unidades funcionales. Por otro lado, si utilizamos renombramiento de registros y duplicamos unidades funcionales, el incremento alcanzable de velocidad es significativamente mayor utilizando los mayores tamaños de ventana.

Por otra parte, la figura 7.8a muestra un estudio del paralelismo a nivel de instrucciones bajo condiciones ideales de no limitación de recursos, predicción perfecta de bifurcaciones, etc. Incluso con estas suposiciones se pueden emitir 5 o más instrucciones en no más del 10% de los casos. Por lo tanto no es sorprendente que la aceleración obtenida a partir de la emisión de múltiples instrucciones se colapse más allá del ancho de emisión de 4 instrucciones, figura 7.8b.

Ciertos estudios indican que la implementación superescalar puede lograr un grado de mejora de las prestaciones de 1,5 a 8. Siendo habitual un grado de mejora entre 2 y 4.

7.5.4 Predicción de saltos

En los cauces segmentados, el tratamiento de saltos es un factor importante, ya que el principal obstáculo para un flujo estable de instrucciones en las etapas iniciales del cauce son las bifurcaciones (saltos condicionales). Para cauces segmentados se han desarrollado diversas aproximaciones para el tratamiento de saltos, como son: *flujos múltiples*, *precaptación del destino del salto*, *buffers de bucles, predicción de saltos y salto retardado*.

Con la llegada de los microprocesadores *RISC*, se exploró intensamente la estrategia de *salto retardado*, tema 6, en la que el procesador siempre ejecuta la instrucción que sigue inmediatamente a la de salto para mantener el cauce lleno mientras se capta un nuevo flujo de instrucciones.

Sin embargo, con la llegada de las máquinas superescalares la estrategia de salto retardado perdió interés. El motivo es que se pueden ejecutar múltiples instrucciones en cada ciclo de retardo, lo cual plantea varios problemas relacionados con las dependencias entre las instrucciones. Por ello, las máquinas superescalares han vuelto a las técnicas de predicción de saltos anteriores a los *RISC*, como la predicción dinámica de salto basada en el análisis de la historia de los saltos.

Como ejemplo, el Pentium II usa esta predicción dinámica del salto basada en el análisis de las ejecuciones recientes de las instrucciones de bifurcación. Para ello utiliza un buffer de destino de saltos (BTB), que almacena información de la ejecución de las instrucciones de bifurcación encontradas recientemente. Cuando aparece una instrucción de bifurcación en el flujo de instrucciones, se examina el BTB para comprobar si existe una entrada correspondiente a dicha bifurcación. Si existe una entrada, la predicción se guía por la información de la historia de esta bifurcación, y dicha historia se actualiza una vez se determine si se produce el salto. Si no existe una entrada en el BTB, correspondiente a esta bifurcación, se crea una nueva entrada y se realiza inicialmente una predicción estática.

7.5.5 Ejecución especulativa

La *ejecución especulativa* es una estrategia que se usa en la mayoría de procesadores de altas prestaciones. Consiste en realizar la ejecución del código (instrucciones o partes de instrucciones), antes de estar seguro de si esta ejecución se requiere. Es una técnica de optimización de prestaciones.

Los procesadores usan ejecución especulativa para reducir el coste de las bifurcaciones (instrucciones de salto condicional). Así, cuando se encuentra un salto condicional, el procesador realiza una predicción sobre cual es el camino más probable a seguir (utilizando técnicas de predicción de saltos), e inmediatamente prosigue la captación, decodificación y ejecución de instrucciones desde dicho punto, sin esperar a saber si es el camino correcto. Si, posteriormente, la predicción resulta ser errónea, el procesador descarta las instrucciones ejecutadas a partir del punto de salto, y continúa la ejecución de las instrucciones del camino correcto. Si la predicción resulta correcta, el procesador 'retira' las instrucciones ejecutadas y continua la ejecución de las instrucciones.

La ejecución especulativa evita la introducción de ciclos de retardo, aumentando la eficiencia de la ejecución y las prestaciones de la máquina a costa de utilizar al máximo sus unidades funcionales. Sin embargo, los ciclos utilizados en las ejecuciones 'extra' consumen ciclos de CPU y por tanto un mayor consumo.

Los procesadores que utilizan ejecución especulativa, ejecutan muchas más instrucciones de las que necesita el flujo del programa. Así pues, esta estrategia debe poseer un mecanismo para que una instrucción ejecutada pueda ser descartada tras su ejecución, pues ¿cómo sino se podría deshacer la ejecución de una instrucción, cuando se decide que ésta nunca hubiera sido ejecutada por una máquina secuencial? Este mecanismo consiste en que:

- El almacenamiento y los registros visibles no se pueden actualizar inmediatamente después de su ejecución.

- Se han de mantener en algún tipo de almacenamiento temporal para después convertirlo en permanente una vez que se determine que el modelo secuencial habría ejecutado la instrucción.

Así pues, en estas condiciones es posible descartar una instrucción ejecutada simplemente ignorando el valor almacenado temporalmente y liberando el almacenamiento y registros bloqueados. Por otro lado, si se verifica que la instrucción ejecutada corresponde al camino correcto, el proceso de retiro de la instrucción simplemente refresca el registro bloqueado con el valor contenido en el almacenamiento temporal.

7.6 Ejecución superescalar

La figura 7.9 considera el proceso de ejecución superescalar. A continuación se enumeran los pasos de este proceso:

- El programa estático es el código del programa a ejecutar, formado por instrucciones ordenadas, tal como se escribió por el programador o se generó por el compilador.
- El proceso de captación de instrucciones forma un flujo dinámico de instrucciones.
- A continuación se analizan las dependencias de este flujo (el procesador puede eliminar las dependencias artificiales mediante renombramiento de registros).
- El procesador envía las instrucciones a una ventana. En la ventana las instrucciones ya no están estructuradas secuencialmente, sino en función de sus dependencias de datos verdaderas.
- Se realiza la ejecución de las instrucciones en el orden que determinan las dependencias de datos y disponibilidad de recursos hardware.
- Finalmente, las instrucciones vuelven a ponerse conceptualmente en orden secuencial, y se registran sus resultados.

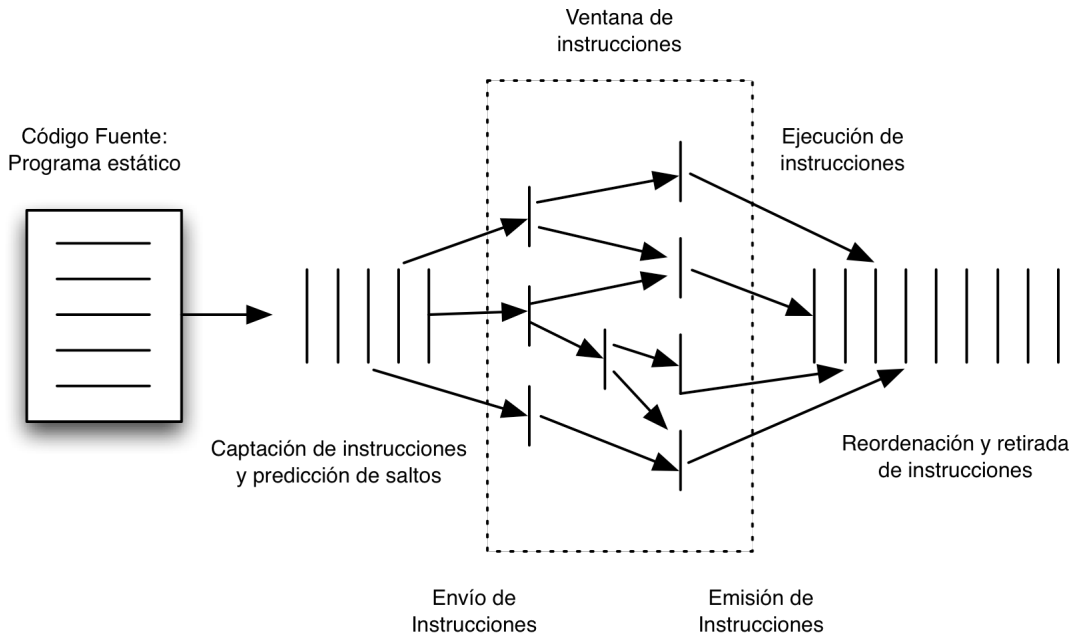


Figura 7.9 Representación esquemática del procesamiento superescalar.

El último paso enumerado es muy importante. Como se ha indicado anteriormente se denomina *retiro* al proceso que se ocupa de los resultados de las instrucciones, registrando sus resultados siguiendo un orden conceptualmente secuencial.

El retiro es necesario debido a que:

- Debido a que una máquina superescalar dispone de múltiples cauces paralelos, las instrucciones pueden finalizar en un orden diferente al del programa estático.

- La utilización de la ejecución especulativa y la predicción de saltos implica que algunas instrucciones pueden completar su ejecución y después ser desechadas porque la bifurcación que llevaba a ellas no se produjo.

7.6.1 Elementos de la implementación superescalar

En definitiva, el hardware que requiere una implementación superescalar es complejo. Los elementos principales serían:

- Captación simultánea de múltiples instrucciones con predicción de saltos.
- Lógica para localizar dependencias de datos.
- Mecanismos para emisión de múltiples instrucciones.
- Ejecución especulativa y mecanismos para el retiro de instrucciones.