4. Tema 4

4.1. Microarquitecturas ILP. Cauces superescalares

En una secuencia de instrucciones se pueden distinguir tres órdenes:

- El orden en que se captan las instrucciones (el orden de las instrucciones en el código).
- El orden en que las instrucciones se ejecutan.
- El orden en el que las instrucciones cambian los registros y la memoria.

El procesador superescalar debe ser capaz de identificar el paralelismo entre instrucciones (ILP) del programa y organizar la captación, decodificación y ejecución de instrucciones en paralelo, utilizando eficazmente los recursos existentes (el paralelismo de la máquina).

Cuanto más sofisticado sea un procesador superescalar, menos tiene que ajustarse a la ordenación de las instrucciones. La única restricción es que el resultado del programa sea correcto.

Un procesador superescalar es un procesador segmentado que puede procesar más de una instrucción en cada etapa. Las etapas son similares a las de un procesador segmentado, aunque hay diferencias:

1. La etapa de **captación** (*IE*) es capaz de leer varias instrucciones por ciclo desde el nivel de caché más alto (L1).
2. Las instrucciones pasan a una cola donde esperan a la **decodificación** (*ID*).
3. Las instrucciones decodificadas deben esperar a que las unidades funcionales necesarias estén disponibles. Para ello, se almacenan en una estructura (ventana de instrucciones, buffer de reorden, de renombramiento, etc.).
4. La etapa de **emisión** (*ISS*) se encarga de determinar las instrucciones disponibles que pueden ejecutarse.
5. La etapa de **ejecución** se implementa con las unidades funcionales que tiene el procesador. Cuantas más tenga, más instrucciones podrán ejecutarse en un mismo ciclo.
6. Por último, se pasa a la etapa de **escritura** (*WB*) que almacena los datos en los registros del procesador.

En los procesadores superescalares se distingue entre *ejecución de la instrucción* (la instrucción está en su etapa de ejecución) y *procesamiento de la instrucción* (la instrucción está en alguna etapa del cauce). Cuando se dice que una instrucción ha *terminado de ejecutarse* nos referimos a que ha terminado su etapa de ejecución, mientras que cuando se retira o sale del cauce se dice que ha *finalizado su procesamiento*.

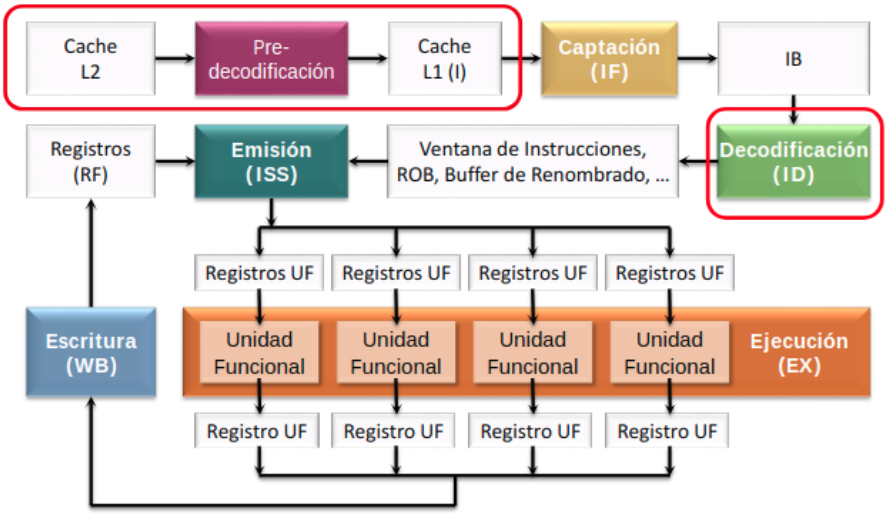


Figura 73: Etapas de un procesador superescalar.

En un procesador superescalar las instrucciones se captan y decodifican en el orden en el que aparecen en el programa. Sin embargo, el orden en el que se ejecutan y el orden en el que cambian los registros y la memoria pueden no coincidir con el orden de captación (orden de las instrucciones en el código). Las razones para no respetar el orden se encuentran en la posibilidad de aprovechar de forma más eficaz el paralelismo entre instrucciones y el disponible en el procesador (unidades funcionales libres, etc.).

La emisión desoredenada supone implementar aspectos del paradigma del **flujo de datos**, según el cual la disponibilidad de datos es quien determina qué instrucciones pueden pasar a ejecutarse y no el orden de las instrucciones del programa. Por tanto, las instrucciones comienzan a ejecutarse cuando los datos de los que dependen están disponibles.

4.1.1. Decodificación paralela y predecodificación

En un procesador superescalar en el que se decodifican varias instrucciones por ciclo no se pueden comprobar al mismo tiempo las dependencias entre los operandos de las instrucciones decodificadas yt las que ya se están ejecutando. Por ello se añade la etapa de *emisión*. Dada la cantidad de instrucciones que hay que decodificar en cada ciclo pueden requerirse varias etapas para la decodificación. En muchos casos, una de las etapas de decodificación se implementa entre la caché L2 y la caché L1i (etapa de *predecodificación*).

La etapa de **predecodificación** se encarga de determinar el tipo de instrucción, facilitando la identificación posterior de los recursos necesarios. En esta etapa se añaden iuna serie de bits que aceleran la decodificación completa de las instrucciones en la etapa (o etapas) de decodificación posterior.

Emisión de instrucciones

La **decodificación** toma varias instrucciones por ciclo según el orden en el que se encuentran las instrucciones en la cola (orden del programa). Cuando las instrucciones son decodificadas

se escriben en una serie de estructuras.

Una de esas estructuras es la **ventana de instrucciones** (o ventana de emisión). Es una cola de registros donde se almacenan las instrucciones que han sido decodificadas y están a la espera de ser emitidas a las unidades funcionales. La etapa de **emisión** será la que se encargue de determinar la instrucción que puede ejecutarse (cuando las unidades funcionales necesarias estén disponibles). En caso de colisión, aplicará una política determinada.

Si la ventana de instrucciones es centralizada, almacenará todas las instrucciones pendientes. Si es distribuida, almacenará sólo las de un único tipo.

Las políticas de emisión se pueden clasificar según el *alineamiento* de la ventana de instrucciones y el *orden* en el que se emiten las instrucciones a las correspondientes unidades funcionales. Según el alineamiento podemos distinguir entre:

- Emisión **alineada**. No se pueden introducir nuevas instrucciones en la ventana deinstrucciones hasta que no esté totalmente vacía, es decir, hasta que no se hayan emitido todas las instrucciones que se introdujeron en un ciclo anterior.
- Emisión **no alineada**. Se pueden introducir nuevas instrucciones mientras hayya espacio en la ventana.

Según el orden:

- Emisión **ordenada**. Se respeta el orden en el que entran las instrucciones (orden del programa). De esta forma, si una instrucción no puede emitirse, las siguientes tampoco podrán, aunque tengan sus operandos y unidades funcionales disponibles.
- Emisión **desordenada**. No existe bloqueo, ya que pueden emitirse todas las instrucciones que tengan disponibles sus operandos y unidades funcionales.

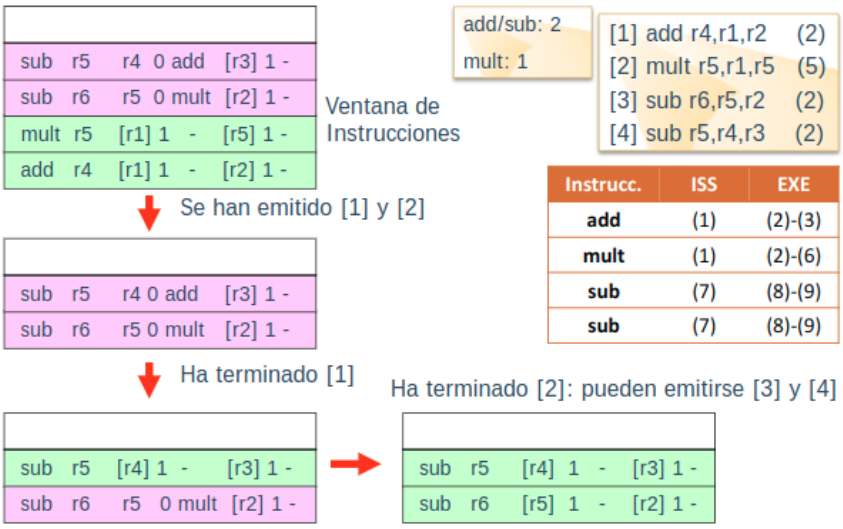


Figura 74: Emisión ordenada.

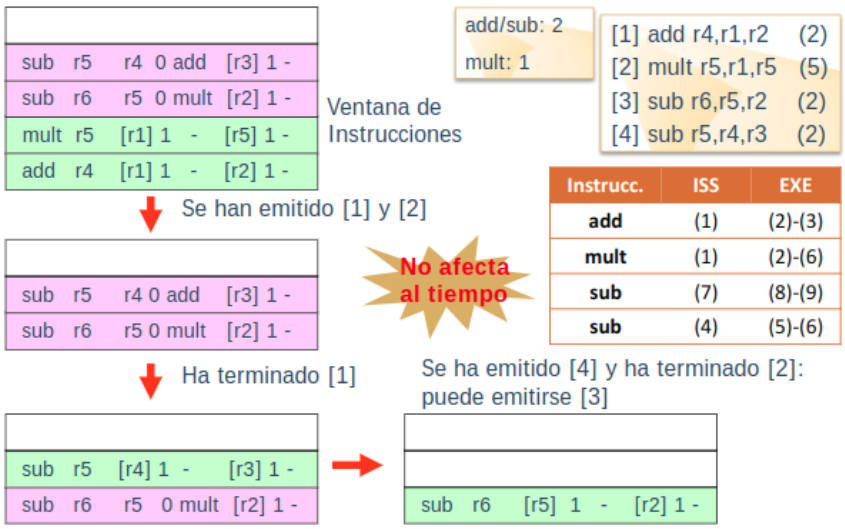


Figura 75: Emisión desordenada.

Estaciones de reserva

En muchas microarquitecturas superesacalares la ventan de instrucciones se distribuye en varias estructuras (estaciones de reserva o *consignas*, *shelving*). La idea consiste en tener una ventana de instrucciones específica para cada unidad funcional o conjunto de unidades funcionales.

En una microarquitectura con estaciones de reserva, las funciones de la etapa de emisión de desdoblan. La primera parte se sigue denominando *emisión* y se implementa en la misma etapa de decodificación, que pasa a llamarse etapa de *decodificación/emisión* (ID/ISS). La determinación de qué instrucciones de la estación de reserva tiene sus operandos disponibles y pasa a ejecutarse se denomina *envío* (*dispatch*).

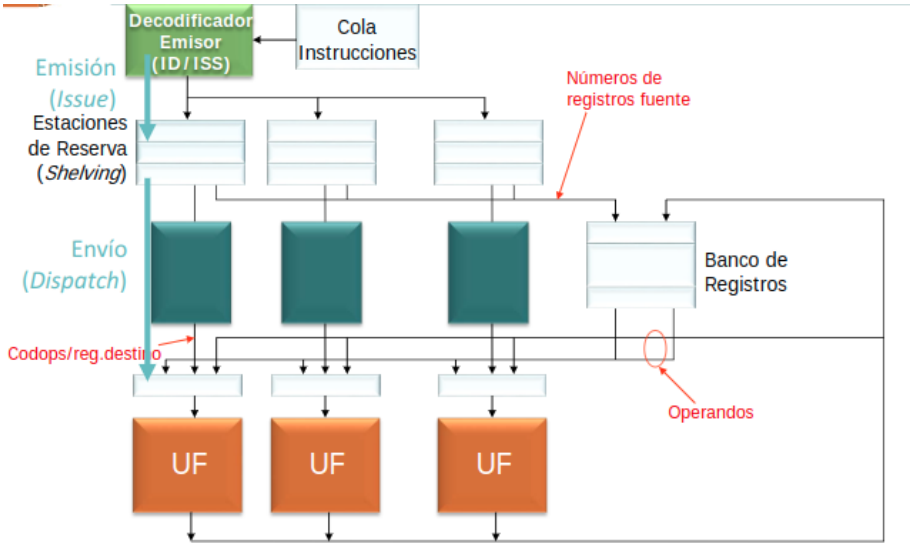
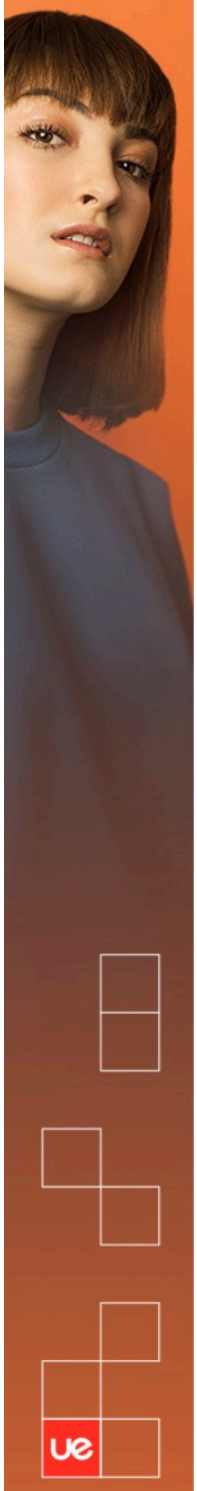


Figura 76: Ventana de reserva distribuida.



Aternativas para el envío a las unidades funcionales

- **Envío de instrucciones**. Las instrucciones se envían a la unidad (o unidades) funcional desde la estación de reserva cuando se puedan empezar a ejecutar nuevas instrucciones. La política de envío implica la *selección de instrucciones ejecutables*. (aquellas que tengan operandos disponibles). Si hay más de una instrucción ejecutable hay que aplicar una regla de arbitraje (usualmente FIFO). Si una estación de reserva sólo envía instrucciones a una unidad funcional, puede enviar una instrucción por ciclo como máximo. Si es compartida, podrá enviar varias.
- **Captación y comprobación de la rapidez de operandos**. Cada registro del banco de registros tiene un bit adicional (bit de validez) que indica si el dato está disponible para poder ser utilizado. En caso de que se use una política de captación de operandos en la emisión, si el bit de validez está a 1, se leerá el operando. En caso contrario, en lugar del dato leído se guardará el número de registro en la estación de reserva. Cuando el resultado esté disponible, se actualizarán sus bits de disponibilidad en el banco de registros y en la estación de reserva.

Renombramiento de registros

En los procesadores en los que las instrucciones pueden terminar de ejecutarse desordenadamente debemos disponer de estrategias que eviten los riesgos RAW (dependencia verdadera), WAR y WAW.

El **renombramiento de registros** consiste en utilizar registros de la microarquitectura (visibles o no visibles al programador) como elementos de almacenamiento con el objetivo de eliminar dependencias.

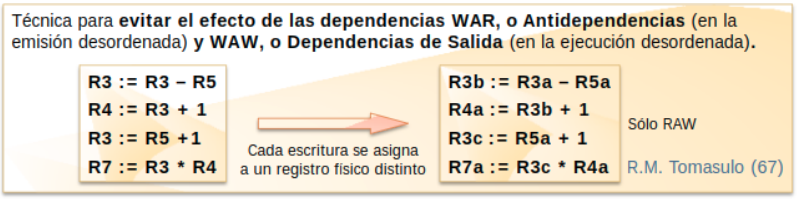


Figura 77: Renombramiento de registros.

Este proceso de renombramiento se puede implementar tanto mediante compilador (utilizando los registros visibles propios de la arquitectura) como mediante compilador. Sin embargo, también se puede implementar mediante hardware incluyendo una serie de registros no accesibles al programador para este propósito. Por tanto, el renombramiento puede ser *estático* si se realiza durante la compilación o *dinámico* si se realiza durante la ejecución utilizando los registros extra.

Si el renombramiento es dinámico, las prestaciones del procesador se verán afectadas por la *velocidad de renombramiento*, es decir, el número máximo de binbres asignados por ciclo. La velocidad y el coste del renombramiento vendrán dados por el número y tupo de buffers que se utilizan como registros temporales y por los *mecanismos* (indexados o asociativos) para acceder a los mismos.

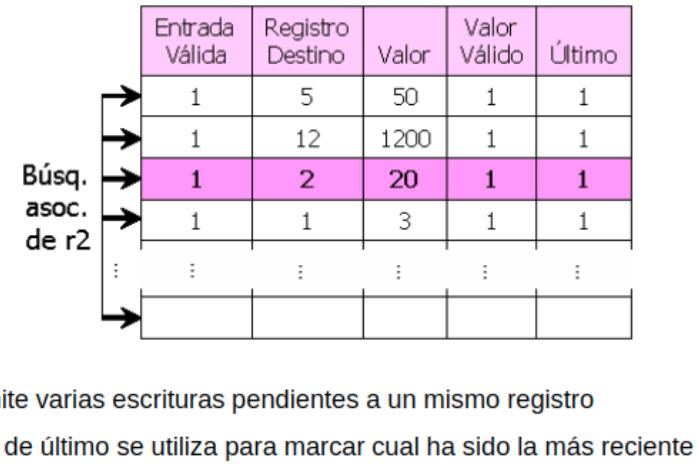


Figura 78: Buffer de renombramiento.

En el acceso *asociativo*, cada línea tiene cinco campos:

- Asignación válida. La línea en cuestión se ha utilizado para renombrar algún registro, por lo que contiene información válida.
- Registro de destino. Registro que se ha renombrado.
- Contenido. Almacena los datos del registro renombrado hasta que se almacenen en el original.
- Contenido válido. Bit de validez. Si está a 0 significa que alguna instrucción que se ha emitido o enviado va a escribir su resuiltado en el campo de contenido.
- Bit de asignación última. Indica si es la última línea en la que se ha hecho una asignación al registro. Si una instrucción necesita un registro en concreto, elegirá aquel del buffer que tenga este campo a 1.

4.2. Consistencia del procesador y procesamiento de saltos

En el procesamiento de una instrucción debemos distinguir entre:

- El final de la ejecución de la operación codificada en la instrucción (se dispone de los resultados producidos por las unidades funcionales pero no se han modificado los registros).
- El final del procesamiento de la instrucción o momento en el que se completa la instrucción (los resultados se escriben en los registros). Si se usa un buffer de reorden (ROB), se utiliza el término *retirar* en lugar de completar.

La actualización de los registros de la arquitectura a partir de los registros internos en los que se ha renombrado ocurre en la última etapa del cauce, denominada etapa de *finalización o retirada de instrucciones*.

La consistencia secuencial de un programa es:

- El orden en que las instrucciones se completan.
- El orden en que se accede a memoria para leer (LOAD) o escribir (STORE).

Cuando se procesan instrucciones en paralelo, el orden en que termina el procesamiento de las instrucciones puede variar con respecto al que tenían en el programa. Sin embargo, debe existir consistencia entre el orden en que se completan y el orden secuencial que tienen en el código.

Consistencia de Procesador Consistencia en el orden en que se completan las instrucciones	Débil: Las instrucciones se pueden completar desordenadamente siempre que no se vean afectadas las dependencias	Deben detectarse y resolverse las dependencias	Power1 (90) PowerPC 601 (93) Alpha R8000 (94) MC88110 (93)
	Fuerte: Las instrucciones deben completarse estrictamente en el orden en que están en el programa	Se consigue mediante el uso de ROB	PowerPC 620 PentiumPro (95) UltraSparc (95) K5 (95) R10000 (96)
Consistencia de Memoria Consistencia del orden de los accesos a memoria	Débil: Los accesos a memoria (Load/Stores) pueden realizarse desordenadamente siempre que no afecten a las dependencias	Deben detectarse y resolverse las dependencias de acceso a memoria	MC88110 (93) PowerPC 620 UltraSparc (95) R10000 (96)
	Fuerte: Los accesos a memoria deben realizarse estrictamente en el orden en que están en el programa	Se consigue mediante el uso del ROB	PowerPC 601 (93) E/S 9000 (92)

Tendencia

Tendencia / Prestaciones

Figura 79: Tipos de consistencia.

Las instrucciones LOAD y STORE implican cambios en el Procesador y en Memoria

LOAD: - Cálculo de Dirección en ALU o Unidad de Direcciones
- Acceso a Cache
- Escritura del Dato en Registro

STORE: - Cálculo de Dirección en ALU o Unidad de Direcciones
- Esperar que esté disponible el dato a almacenar (en ese momento acaba)

La Consistencia de Memoria Débil (reordenación de los accesos a memoria):

- **'Bypass' de Loads / Stores:**
  Los Loads pueden adelantarse a los Stores pendientes y viceversa (siempre que no se violen dependencias)
- **Permite los Loads y Stores Especulativos:**
  Cuando un Load se adelanta a un Store que le precede antes de que se haya determinado la dirección se habla de Load especulativo. Igual para un Store que se adelanta a un Load o a un Store.
- **Permite ocultar las Faltas de Cache:**
  Si se adelanta un acceso a memoria a otro que dio lugar a una falta de cache y accede a Memoria Principal.

Figura 80: Reordenamiento de LOAD y STORE.

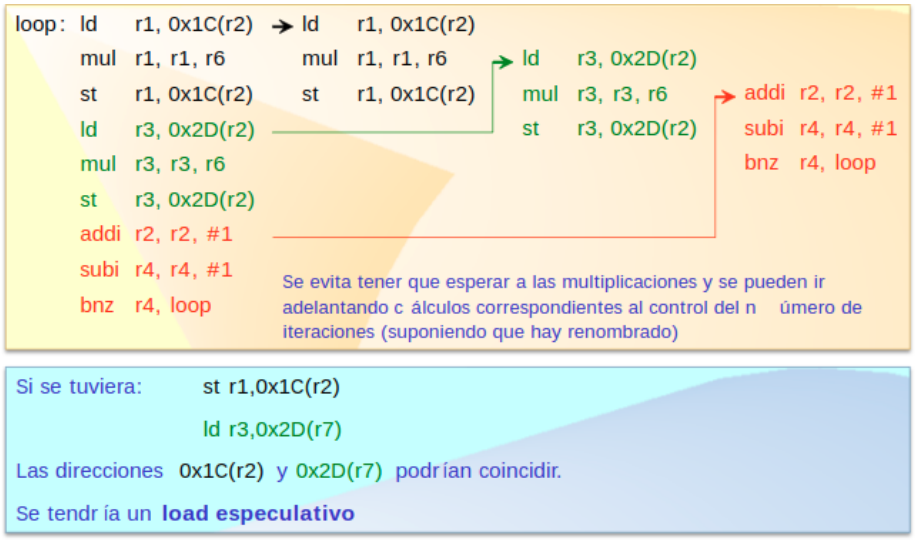


Figura 81: Ejemplo de reordenamiento de LOAD y STORE.

4.2.1. Buffer de reordenamiento (ROB)

Al utilizar renombramiento de registros, los resultados de las instrucciones se alojarán en las líneas del buffer de renombrado. El problema reside en decidir el momento en el que pasen a escribirse en los registros, completando la instrucción. Para ello usaremos el *buffer de reordenamiento* (ROB):

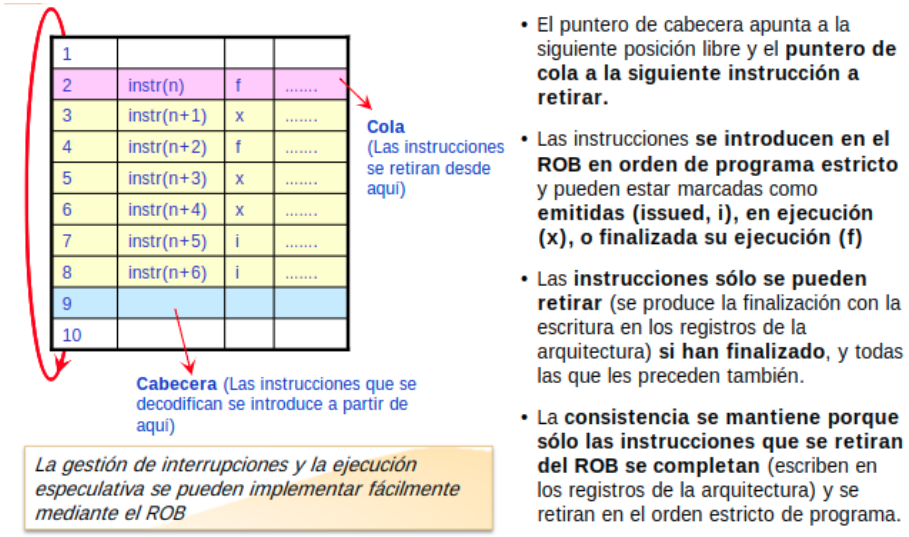


Figura 82: Buffer de reordenamiento.

4.2.2. Procesamiento especulativo de saltos

Los saltos causan una gran reducción de prestaciones en los procesadores superescalares. Cuanto antes detectemos una instrucción de salto, menor será la posible penalización. Los saltos se detectan normalmente en la decodificación (o incluso en la captación, si hay predecodificación).

Si en el momento de evaluar un salto la condición no se puede verificar utilizamos el *procesamiento especulativo de saltos*. En cuanto al acceso a las instrucciones del destino del salto, debemos implementar procedimientos para acceder a ellas lo más rápido posible.

Gestión de Saltos Condicionales no Resueltos (Una condición de salto no se puede comprobar si no se ha terminado de evaluar)	Bloqueo del Procesamiento del Salto	Se bloquea la instrucción de salto hasta que la condición esté disponible (68020, 68030, 80386)
	Procesamiento Especulativo de los Saltos	La ejecución prosigue por el camino más probable (se especula sobre las instrucciones que se ejecutarán). Si se ha errado en la predicción hay que recuperar el camino correcto. (Típica en los procesadores superescalares actuales)
	Múltiples Caminos	Se ejecutan los dos caminos posibles después de un salto hasta que la condición de salto se evalúa. En ese momento se cancela el camino incorrecto. (Máquinas VLIW experimentales: Trace/500 , URPR2)
Evitar saltos condicionales	Ejecución Vigilada (*Guarded Exec.*)	Se evitan los saltos condicionales incluyendo en la arquitectura instrucciones con operaciones condicionales (IBM VLIW, Cydra-5, Pentium, HP PA, Dec Alpha)

Figura 83: Gestión de saltos condicionales no resueltos.

La predicción de saltos se basa en la idea de que el comportamiento de una instrucción de salto condicional presenta cierta regularidad, por lo que puede predecirse con una tasa de aciertos bastante elevada. En caso de fallar en la predicción habrá una penalización, por lo que debemos adoptar un modelo de predicción de saltos con una tasa de acierto lo más elevada posible.

Los principales esquemas para la predicción de salto son:

- **Predicción fija**. Se toma siempre la misma decisión: el salto siempre se realiza (*taken*) o nunca (*not taken*).
- **Predicción verdadera**. La decisión depende de las características del salto.
  - **Predicción estática**. La decisión se toma en tiempo de compilación.
  - **Predicción dinámica**. La decisión se toma a partir de las ejecuciones pasadas de la instrucción (historia de la instrucción de salto).

La **predicción estática** puede ser basada en el código de operación (dependiendo del tipo de instrucción de salto), en el desplazamiento del salto (si es positivo es un salto hacia delante, si es negativo es hacia atrás, probablemente un bucle) o dirigida por el compilador (fija un bit de predicción en cada instrucción de salto).

Predicción basada en el Código de Operación Para ciertos códigos de operación (ciertos saltos condicionales específicos) se predice que el salto se toma, y para otros que el salto no se toma	MC88110 (93) PowerPC 603(93)
Predicción basada en el Desplazamiento del Salto Si el desplazamiento es positivo (salto hacia delante) se predice que no se toma el salto y si el desplazamiento es negativo (salto hacia atrás) se predice que se toma.	Alpha 21064 (92) PowerPC 603 (93)
Predicción dirigida por el Compilador El compilador es el que establece la predicción fijando, para cada instrucción, el valor de un bit específico que existe en la instrucción de salto (bit de predicción)	AT&T 9210 (93) PowerPC 603 (93) MC88110 (93)

Ejemplo: Predicción Estática en el MC88110

Formato	Instrucción		Predicción
	Condición Especificada	Bit 21 de la Instr.	
bcnd (*Branch Conditional*)	≠ 0	1	Tomado
	= 0	0	No Tomado
	> 0	1	Tomado
	< 0	0	No Tomado
	≥ 0	1	Tomado
	≤ 0	0	No Tomado
	bb1 (*Branch on Bit Set*)		Tomado
	bb0 (*Branch on Bit Clear*)		No Tomado

Figura 84: Predicción estática.

En cuanto a la **predicción dinámica**, para cada instrucción de salto condicional debe guardarse información sobre el comportamiento pasado que permita tomar la decisión. Según la forma de almacenarla, podemos distinguir dos tipos de predicción dinámica:

- **Implícita**. No hay bits de historia, sino que se almacena la instrucción que se ejecutó tras el salto la última vez que se captó. Es bastante limitado, ya que predice hacer lo mismo que la última vez.
- **Explícita**. Se almacenan bits de historia. La primera vez que se ejecuta, al no haber historia, se utiliza predicción estática o bien se presupone un valor para los bits de estado.

El porcentaje de aciertos (*prediction accuracy*) es fundamental para mantener un promedio elevado de instrucciones por ciclo.

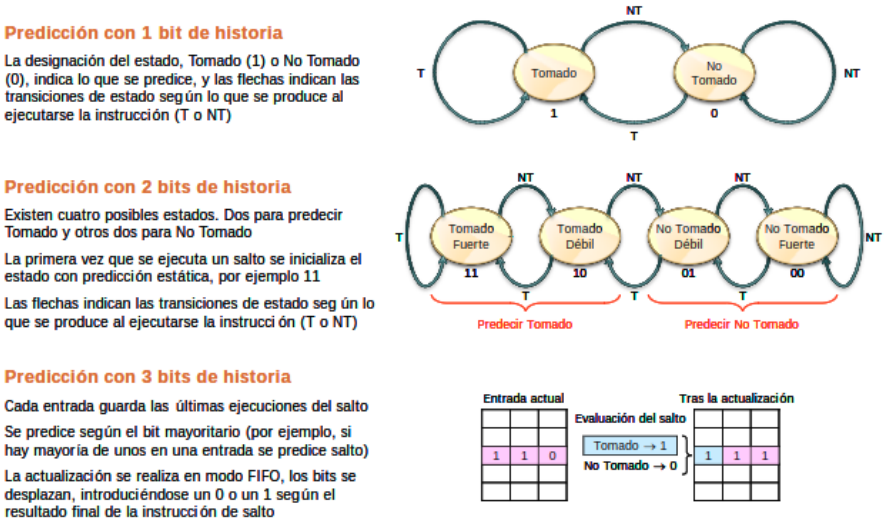


Figura 85: Predicción dinamica.

A medida que utilizamos más recursos hardware para implementar un esquema de predicción, se pujeden conseguir mayores porcentajes de aciertos. Sin embargo, hemos de tener en cuenta el retardo que introduce la circuitería adicional.

Existen varias alternativas para almacenar la información de la historia de saltos: caché de instrucciones, tabla de historia de salto (BHT), caché de instrucciones de salto (BTIC), etc.

Tras realizar la predicción, el procesador continúa ejecutando instrucciones especulativamente hasta que se resuelve la condición. Puede suceder que aparezcan más instrucciones de salto en la ejecución especulativa, lo que puede hacer que la penalización crezca muchísimo si erramos. Por tanto, debemos buscar una predicción lo mejor posible para evitar penalizaciones en el rendimiento.

El **nivel de especulación** es el número de instrucciones de salto condicional que pueden ejecutarse especulativamente.

El **grado de especulación** indica la etapa hasta la que se procesan las instrucciones que siguien con un camino especulativo tras un salto.

Instrucciones de ejecución condicional

Pretenden eliminar las instrucciones de salto en los códigos incluyendo en el repertorio máquina instrucciones con operaciones condicionales (instrucciones con predicado).

Las instrucciones de ejecución condicional tienen dos partess, la condición (guardia) y la operación propiamente dicha.

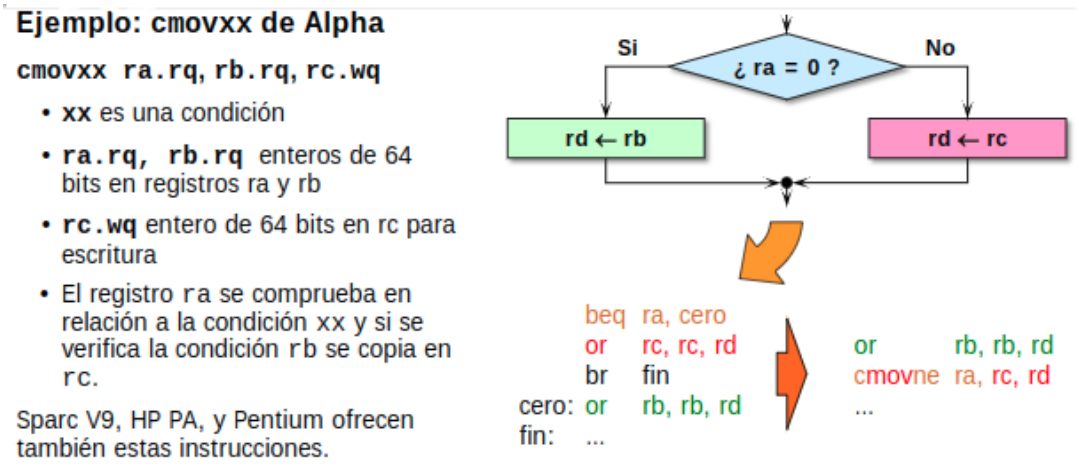


Figura 86: Ejecución condicional.

4.3. Procesamiento VLIW

4.3.1. Características generales y motivación (ILP hardware vs software)

Los procesadores VLIW son similares a los superescalares, ya que:

- Son procesadores segmentados que pueden emitir varias instrucciones en cada ciclo.
- Disponen de varias unidades funcionales independientes, por lo que pueden ejecutar varias operaciones a la vez.

Sin embargo, mientras que en los superescalares es el hardware el encargado de detectar el paralelismo, aquí el paralelismo se indica explícitamente en cada instrucción captada. En lugar de enviar varias instrucciones independientes a las unidades funcionales, una arquitectura VLIW empaqueta varais operaciones en una instrucción muy larga (*Very Long Instruction Word*). Por tanto, cada instrucción VLIW codifica operaciones que se ejecutan a la vez.

La decisión de qué instrucciones se ejecutarán a la vez (el formamiento de la VLIW) corresponde al compilador, lo que implica un hardware más sencillo que el de un procesador superescalar. Cuantas más instrucciones se emitan por ciclo, más ventajosa será la arquitectura VLIW, ya que no se requiere añadir hardware.

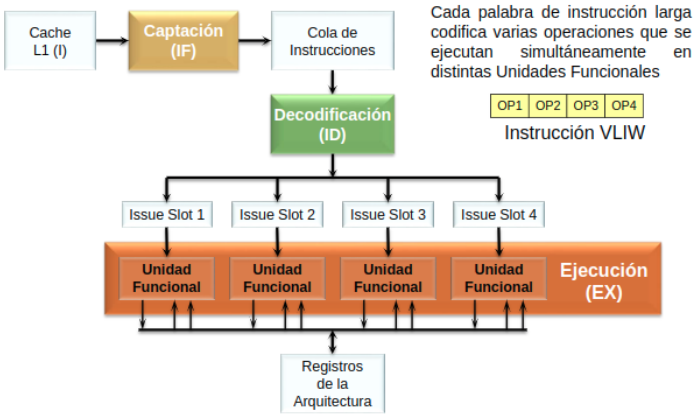


Figura 87: Esquema VLIW.

4.3.2. Planificación estática

En un procesador **VLIW**, la planificación es estática. Por tanto, necesitamos la asistencia del compilador, que puede realizar renombrados, reorganizaciones de código, etc. para mejorar el uso de los recursos, el esquema de predicción de saltos...

En un procesador **escalar**, la predicción es dinámica. Requiere menos asistencia del compilador a cambio de más coste hardware. Facilita la portabilidad de código entre la misma familia de procesadores.

El compilador construye paquetes de instrucciones (ventanas de emisión) sin dependencias, de forma que el procesador no tiene que comprobarlas de forma explícita.

Hay dos tipos de planificación estática:

- Planificación **local**. Actúa sobre un bloque básico mediante desenrollado de bucle y planificación de las instrucciones del cuerpo aumentado del bucle.
- Planificación **global**. Actúa considerando bloques de código entre instrucciones de salto.

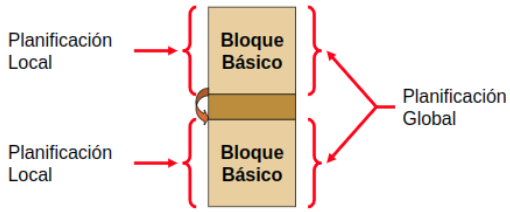


Figura 88: Tipos de planificacion estatica.

Planificación estática local

El **desenrollado de bucles** consiste en crear bloques básicos más largos, con más sentencias.

Estas sentencias suelen ser independientes, ya que operan sobre diferentes datos.

La **segmentación software** (*software pipelining*) consiste en reorganizar los bucles de forma que cada iteración del código transformado contenga instrucciones tomadas de distintas iteraciones del bloque original. De esta forma podemos separar las instrucciones dependientes del bucle original en diferentes iteraciones del bucle nuevo. Es decir, se pretende alejar al máximo las instrucciones dependientes.

Planificación estática global

Esta planificación mueve código a través de los saltos condicionales (que no correspondan al control del bucle). Se parte de una aproximación de las frecuencias de ejecución de los posibles caminos tras una instrucción de salto condicional. Se apoya en las **instrucciones con predicado** y en la **especulación**.

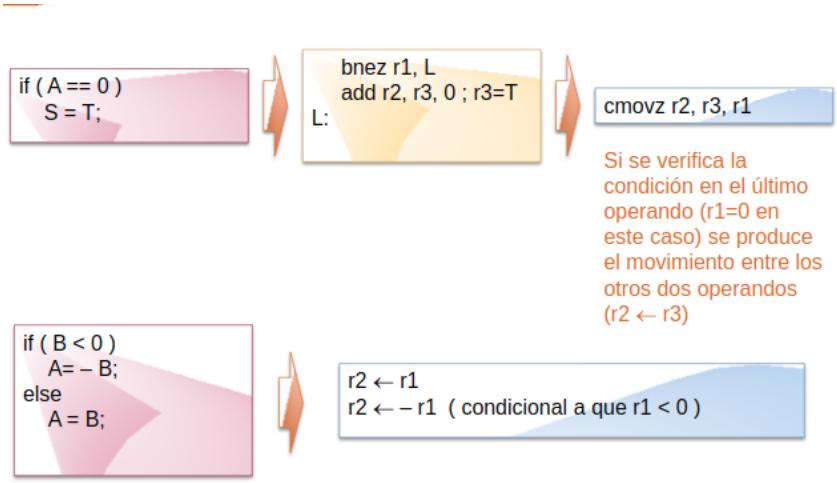


Figura 89: Ejemplo de instrucciones de ejecución condicional.

Una operación con predicado es aquella cuyo resultado modifica o no el destino de dicha operación en función del valor del predicado. Se expresa como <*p*> *operacion*. Si $p = 1$, la operación se realizará, si no, no. Ejemplo:

p1 cmp.eq r1,0
<p1> add r3,r2,r0
Si r1=0, entonces r3=r2+r0

Figura 90: Ejemplo de instrucciones con predicado (I).

También se pueden asignar varios predicados a la vez. Por ejemplo:

p1,p2 cmp.eq r1,0
<p1> add r3,r2,r0
<p2> sub r3,r2,r0
Si r1=0, p1=1 y p2=0. Por tanto, r3=r2+r0
Si r1≠0, p1=0 y p2=1. Por tanto, r3=r2-r0

Figura 91: Ejemplo de instrucciones con predicado (II).

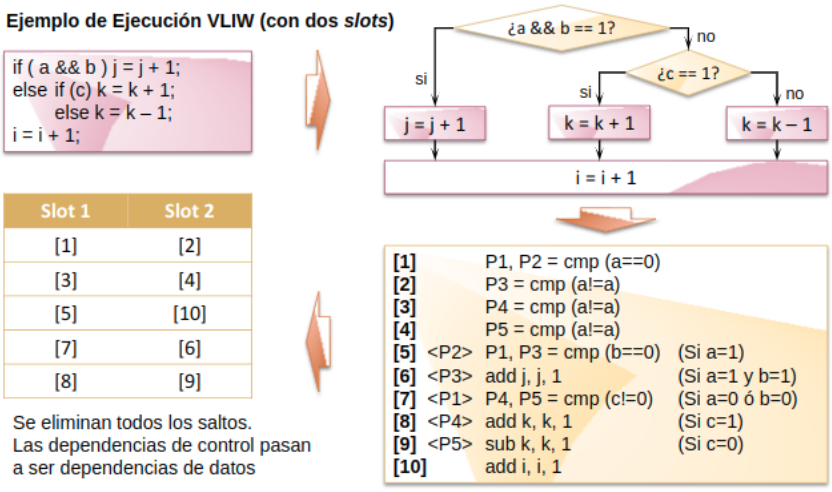


Figura 92: Instrucciones con predicado.

4.3.3. Procesamiento especulativo

El procesamiento especulativo se basa en la predicción de que una instrucción, condición, etc. es muy probable que se realice, por lo que se adelanta su procesamiento, mejorando las prestaciones.

Si la predicción es errónea, el procesamiento especultativo tendrá una penalización, que puede ir desde deshacer el camino hasta tener que incluir código extra para deshacer los cambios realizados, vigilar el comportamiento frente a excepciones, etc.

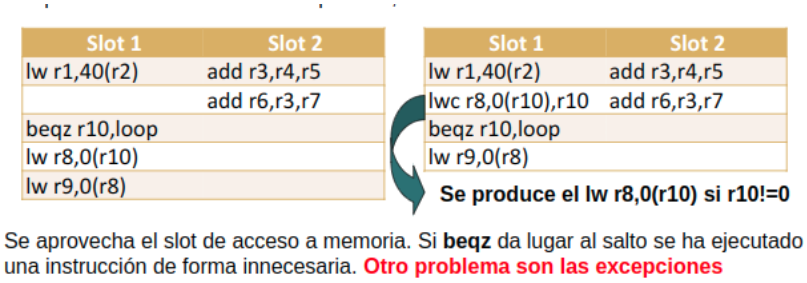


Figura 93: Procesamiento especulativo.

Uso de centinelas para permitir la especulación de las referencias de memoria
Cuando no hay ambigüedad, el compulador adelanta los LOADs respecto a los STOREs

para reducir la longitud del camino crítico en el código.

Sin embargo, cuando la hay:

1. Se incluye en la arquitectura una instrucción para comprobar los conflictos de direcciones.
2. La instrucción se sitúa en la posición original del LOAD (**centinela**).
3. Cuando se ejecutael LOAD especulativo, el hardware guarda la dirección a la que se ha accedido.
4. Si los sucesivos STOREs no acceden a esa dirección, la especulación es correcta. En caso contrario, habrá fallado.
5. Si la especulación falla:
   - Si afecta sólo al LOAD, se vuelve a ejecutar al llegar al centinela.
   - Si se han ejecutado instrucciones que dependen del LOAD, habrá que repetirlas.

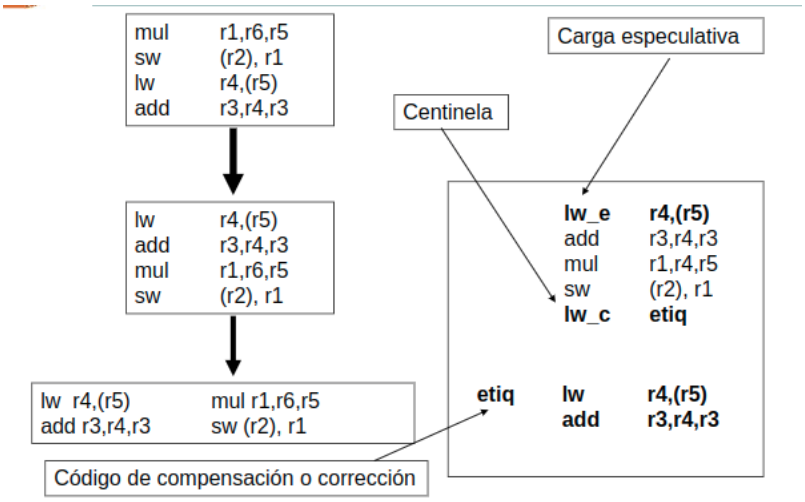


Figura 94: Uso de centinelas.

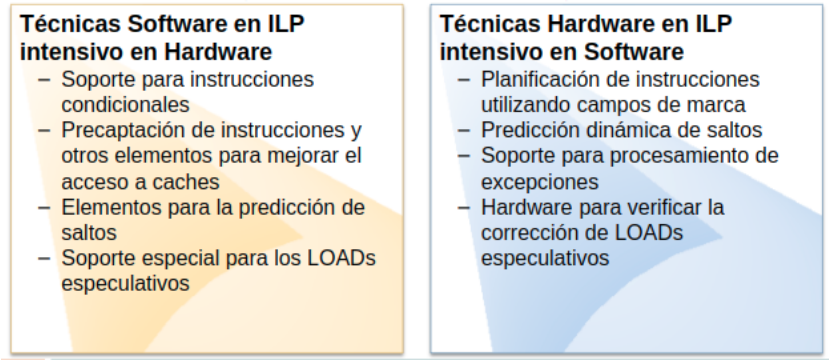


Figura 95: Especulación software vs hardware.

Al final, los procesadores que implementan ILP hardware han tomado ideas de la especulación software y viceversa.