

# 6. Procesadores Superescalares y VLIW

## 1. Introducción

## 2. El cauce superescalar

## 3. El modelo VLIW (*Very Long Instruction Word*)

Las técnicas que hemos visto en los capítulos anteriores las hemos utilizado para eliminar los distintos tipos de riesgos y conseguir la deseada tasa de ejecución de una instrucción por ciclo.

No obstante, se siguen buscando mejores prestaciones, esto es, un mayor número de instrucciones ejecutadas por unidad de tiempo. Para ello hay dos opciones. Una es la de los **procesadores supersegmentados**, en los que, aprovechando las mejoras tecnológicas que favorecen circuitos más rápidos, se diseña un cauce con un mayor número de etapas de menor duración, con lo que, al reducirse el tiempo de la etapa, el cauce puede funcionar a frecuencias mayores.

La otra opción es la que se aborda en este capítulo, en la que los procesadores disponen de un cauce en el que cada etapa puede procesar más de una instrucción simultáneamente. Esto significa que se pueden **arrancar o emitir varias instrucciones en paralelo**.

La planificación de qué instrucciones van a ejecutarse en paralelo puede hacerse estática o dinámicamente (la hace el compilador o la hace el procesador). El modelo de planificación estática que vamos a ver es el de los **procesadores VLIW** (*Very Long Instruction Word*), y en la planificación dinámica vamos a abordar los **procesadores superescalares**.

Objetivo de los procesadores segmentados

Una instrucción por ciclo

¿ Cómo mejorar la tasa de una instrucción por ciclo ?

Todas las etapas del cauce deben ejecutar más de una instrucción por ciclo

Como hemos visto anteriormente, una de las formas de aumentar las prestaciones de los procesadores segmentados consiste en utilizar varias unidades funcionales que puedan trabajar simultáneamente con instrucciones distintas. Esto, no obstante, no permite sobreponer el límite máximo de una instrucción terminada por cada ciclo, dado que la última etapa del cauce (WB) solo puede procesar una instrucción por ciclo, como les ocurre a las etapas de extracción, decodificación y acceso a memoria.

La idea que subyace en los procesadores superescalares y VLIW es la de conseguir que puedan procesarse simultáneamente varias instrucciones **en todas las etapas**, con lo que entonces sí podría superarse la tasa de una instrucción terminada por ciclo.

## Paralelismo a nivel de instrucción (ILP)

Número de instrucciones de un programa que pueden ejecutarse en paralelo (en promedio).

ADD	R1 , R2 , R3
SUB	R4 , R5 , R6
AND	R7 , R8 , R9

Paralelismo de grado 3

ADD	R1 , R2 , R3
SUB	R4 , R5 , R1
AND	R7 , R8 , R4

Paralelismo de grado 1

## Paralelismo a nivel de máquina (MLP)

Número máximo instrucciones que la máquina puede ejecutar en paralelo  
Grado de paralelismo del procesador

El mayor o menor grado de **paralelismo a nivel de instrucción** o ILP (*Instruction Level Parallelism*), es decir, la mayor o menor facilidad para encontrar en un programa instrucciones que puedan procesarse simultáneamente, depende de la frecuencia con la que aparecen dependencias de datos y de control.

Por ejemplo, en el fragmento de programa de la izquierda, no hay ninguna dependencia de datos o de control entre ellas. Esto quiere decir que las tres instrucciones podrían ejecutarse simultáneamente. En cambio, en el grupo de instrucciones de la derecha hay dependencias de datos de tipo RAW, por lo que deben procesarse secuencialmente, incluso con ciertos retardos en su emisión, que permitan tener los operandos necesarios solo cuando estén disponibles. Así, podíamos afirmar que esta última secuencia de instrucciones tiene un grado de paralelismo igual a 1, ya que solo se pueden ejecutar las instrucciones de una en una, mientras que en la primera secuencia tiene un grado de paralelismo igual a 3.

En cualquier caso, el aprovechamiento del grado de paralelismo que posea una secuencia de instrucciones depende de los recursos que tenga el procesador. A esta capacidad para procesar instrucciones en paralelo se le denomina **paralelismo del procesador** o MLP (*Machine Level Parallelism*). Este paralelismo viene determinado por el número de instrucciones que pueden procesarse al mismo tiempo en cada una de las etapas del procesador, es decir, el número de instrucciones que pueden extraerse de memoria, decodificarse, ejecutarse y escribir sus resultados al mismo tiempo.

Como vemos, el paralelismo a nivel de instrucción viene determinado por los riesgos de datos y de control, mientras que el paralelismo a nivel de máquina está limitado por los riesgos estructurales.

Para ejecutar varias instrucciones en paralelo  
hay que formar  
grupos de instrucciones sin dependencias

### Paquete de Emisión

Estáticamente  
(el compilador)

**Procesadores VLIW**  
(Very Long Instruction Word)

Dinámicamente  
(el procesador)

**Procesadores Superescalares**

Ya que se desea emitir varias instrucciones a ejecución de manera simultánea, aprovechando el grado de paralelismo de los programas, hay que ir formando paquetes o grupos de instrucciones que puedan ejecutarse de manera paralela, de tal manera que se vaya alimentando y emitiendo un paquete tras otro, y las instrucciones de cada paquete se envíen a la unidad funcional correspondiente que ejecuta cada instrucción. Finalizada la etapa de ejecución, habrá una etapa de escritura para cada una de las instrucciones ejecutadas.

Hay dos vías principales para implementar estos procesadores de emisión múltiple, atendiendo al modo o el momento en que se realiza la elección de las instrucciones que conforman cada grupo de procesamiento: estática o dinámicamente.

**Estáticamente:** La elección de los grupos de instrucciones se realiza de manera estática, es decir, en tiempo de compilación. El compilador conoce el programa y las características del procesador, así que agrupa instrucciones en paquetes, ocupándose de evitar los riesgos de datos y de control, de tal manera que cada paquete de instrucciones (paquete de emisión) está formado por instrucciones independientes que pueden ejecutarse en paralelo sin ningún tipo de riesgos. A este tipo de arquitecturas se las conoce como **VLIW** (*Very Long Instruction Word*).

**Dinámicamente:** La elección de los grupos de instrucciones se realiza dinámicamente, esto es, en tiempo de ejecución. Las instrucciones se alimentan en orden y es el propio procesador el que se encarga de elegir el conjunto de instrucciones (0, 1, 2, ...) que pueden emitirse de manera paralela en cada ciclo de reloj. Esta arquitectura es la que corresponde a los **procesadores superescalares**. Debe mencionarse que la obtención de un buen rendimiento en estos procesadores también requiere que el compilador planifique y ordene las instrucciones tratando de minimizar las dependencias de tal manera que la emisión pueda realizarse lo más rápidamente posible. No obstante, el procesador es el responsable de garantizar que las instrucciones emitidas pueden ejecutarse en paralelo.

## 2. Procesadores superescalares

Un procesador superescalar es un procesador segmentado que puede procesar más de una instrucción en cada una de sus etapas



Como nos dice arriba, un procesador superescalar es un procesador segmentado que puede procesar **más de una instrucción en todas y cada una de sus etapas**.

Obsérvese que un procesador segmentado procesa varias instrucciones simultáneamente, pero en un momento dado, solamente hay una instrucción en cada etapa.

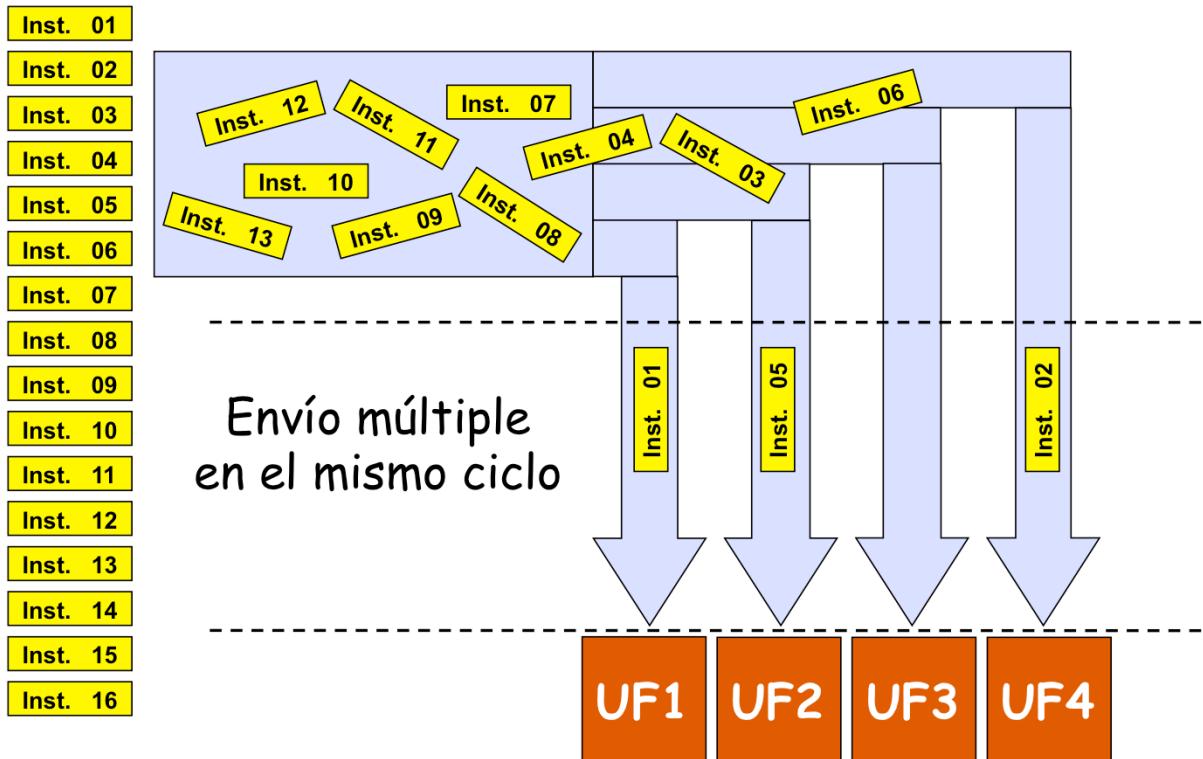
La definición anterior de procesador superescalar no permite distinguirlo de un procesador VLIW que veremos al final de este capítulo, pero nos basta por el momento.

Ateniéndonos a la definición de "superescalar", por tanto, la etapa de alimentación de instrucciones (*Fetch*), por ejemplo, debe ser capaz de alimentar varias instrucciones en cada ciclo, mientras que el procesador segmentado convencional solamente extrae una instrucción de memoria por ciclo.

En el contexto del estudio de los procesadores superescalares es conveniente distinguir entre estos dos conceptos:

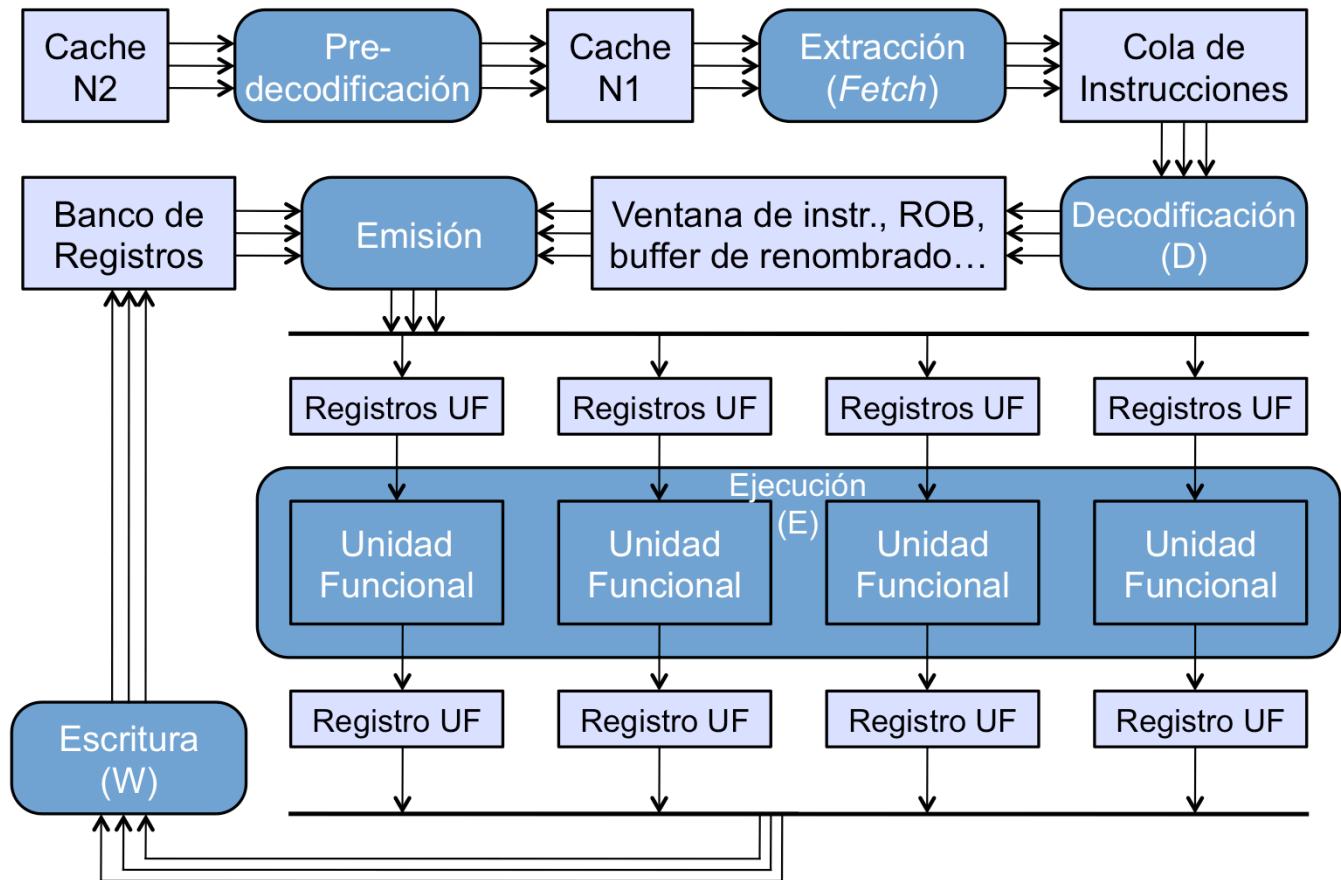
- **Ejecución de la instrucción**, que se refiere a la instrucción que está en su etapa de ejecución.
- **Procesamiento de la instrucción**, que indica que la instrucción está en alguna de las etapas del cauce.

Es decir, si una instrucción ha entrado en el cauce y está en cualquiera de sus etapas, decimos que la instrucción se está procesando. Si la instrucción está concretamente en la etapa de Ejecución, decimos que se está ejecutando. Si la instrucción ha salido de la etapa de ejecución, pero no ha abandonado el cauce, decimos que la instrucción ya se ha ejecutado. Si la instrucción ya ha abandonado el cauce (no está en ninguna de sus etapas), decimos que ya se ha procesado.



Por lo que hemos comentado hasta ahora, tenemos que un procesador superescalar va a extraer varias instrucciones de memoria en el mismo ciclo y atravesando las etapas del cauce llegarán a las diversas unidades funcionales que constituyen la etapa de ejecución, donde se ejecutarán varias instrucciones simultáneamente.

En el procesador segmentado convencional, también puede haber varias unidades funcionales en las que se ejecutan simultáneamente varias instrucciones, pero en este último las instrucciones se envían a ejecución (a su unidad funcional correspondiente) a razón de una por ciclo. Sin embargo, en el procesador superescalar, en un único ciclo se envían múltiples instrucciones a las unidades funcionales correspondientes.



Las etapas de un procesador superescalar son muy similares a las del procesador segmentado convencional, aunque existen algunas diferencias que vamos a comentar.

Al igual que en otros procesadores, las instrucciones se suponen en una caché de nivel 1, de donde se alimentan en la etapa de extracción (F), aunque ahora se alimentan varias instrucciones por ciclo, y éstas se pasan, en el orden con que se han alimentado, a una cola o buffer de instrucciones, desde donde se introducen ordenadamente en la etapa de Decodificación (D), tantas como dicha etapa sea capaz de decodificar por ciclo.

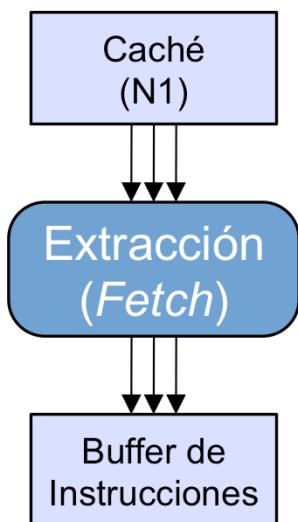
Las instrucciones decodificadas deben esperar a que estén disponibles tanto los operandos necesarios, como su correspondiente unidad funcional. Para ello, al final de la etapa de decodificación se almacenan en una serie de estructuras —ventana de instrucciones, buffer de reordenación (ROB), buffer de renombrado etc.— que veremos más adelante.

La etapa de Emisión (*/Issue*) se encarga de seleccionar las instrucciones que pueden pasar a ejecutarse, es decir, las que tienen disponibles tanto sus operandos como la unidad funcional que requiere.

La etapa de Ejecución se implementa a través de las distintas unidades funcionales que realizan las operaciones indicadas en las instrucciones (aritméticas, acceso a memoria, saltos...). El número de unidades funcionales determina el máximo número de instrucciones que pueden ejecutarse en paralelo.

Por último, se tiene una etapa de finalización o Escritura que se encarga de almacenar los resultados (varios resultados por ciclo) en el banco de registros del procesador.

En un procesador en el que se decodifican varias instrucciones por ciclo no es posible comprobar al mismo tiempo las dependencias de los operandos de las instrucciones en proceso, además de tener que determinar a qué unidad funcional se envía cada instrucción en el momento en que quede libre la UF y estén disponibles sus operandos. Por esto, en lugar de tener una única etapa de decodificación, todo este trabajo se reparte con otras etapas, como Pre-decodificación y la Emisión. En las siguientes páginas iremos detallando el cometido de cada una de estas etapas.

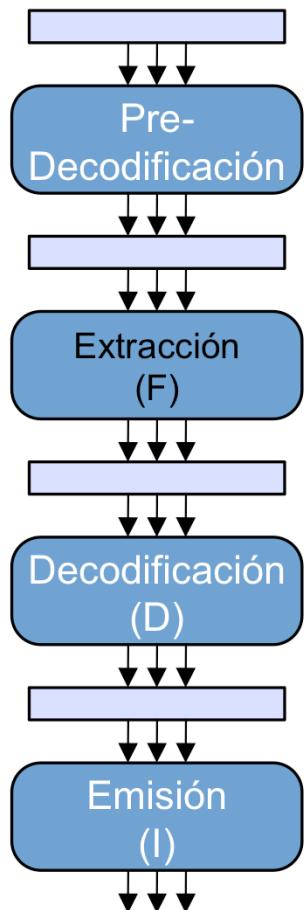


- ✓ Se encarga de la alimentación de instrucciones desde la caché.
- ✓ Las instrucciones leídas se almacenan en un buffer.
- ✓ Este proceso se realiza en orden.
- ✓ Tiene que ser capaz de leer varias instrucciones por ciclo.

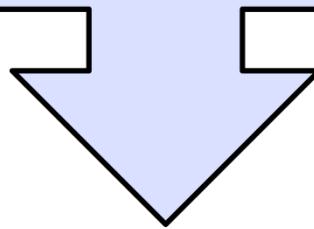
La etapa de Extracción (*Fetch*) se encarga de alimentar instrucciones desde memoria y dejarlas a disposición de las siguientes etapas del cauce. Actualmente, lo normal es que las instrucciones se encuentren en una memoria caché de instrucciones, seguramente en una caché de nivel 1.

Las instrucciones se extraen de memoria en el orden en que se encuentran, y en ese mismo orden se van depositando en un buffer o cola de instrucciones, de donde se van a ir suministrando a la etapa de Decodificación.

Esta etapa tiene que ser capaz de extraer no una instrucción, sino varias instrucciones por ciclo, para permitir así una alimentación múltiple y en paralelo al resto de las etapas, incluidas las diversas unidades funcionales con que cuente el procesador.



- Decodificación de la instrucción
- Comprobación de dependencias
- Asignación de Unidad Funcional
- ¿Operandos disponibles?



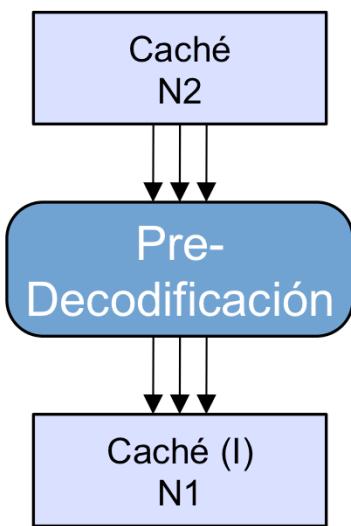
**Reparto de funciones  
en varias etapas  
(P, D, I)**

En un procesador segmentado, la etapa de Decodificación se encarga, además de la propia decodificación de la instrucción, de la comprobación de las dependencias entre los operandos de las instrucciones en proceso, así como de determinar la unidad funcional a la que debe ser enviada la instrucción y de comprobar si los operandos y la unidad funcional están disponibles.

En un procesador superescalar, en el que estas tareas deben realizarse para múltiples instrucciones por ciclo, resulta una tarea demasiado laboriosa, por lo que todas estas funciones suelen repartirse entre dos o tres etapas.

Aquí veremos cómo estas tareas se reparten entre la **pre-Decodificación**, la propia **Decodificación** y la **Emisión (Issue)**.

Veamos a continuación el cometido de cada una de estas etapas.



- ✓ Se determina el tipo de instrucción, facilitando la posterior identificación de los recursos necesarios (dir. salto, UF, dir. mem.)
- ✓ Se añaden unos bits a la instrucción con la información obtenida.

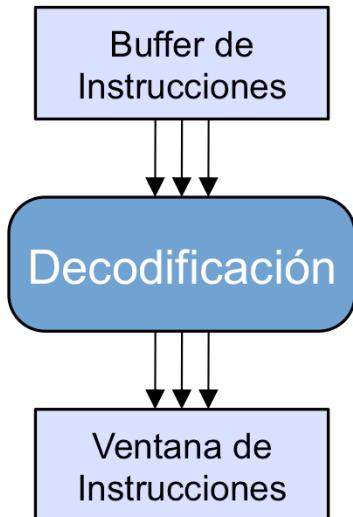


**Aumenta el ancho de banda de los buses**

La etapa de Pre-Decodificación se suele encargar de determinar el tipo de instrucción para facilitar la identificación posterior de los recursos que va a necesitar la instrucción. Por ejemplo:

- En el caso de instrucciones salto, se adelanta el procesamiento de los saltos incluso antes de llegar a la etapa de Decodificación, intentando determinar **la dirección del salto** y facilitando el procesamiento especulativo del salto.
- Si en la pre-decodificación se obtiene información de **la unidad funcional** que va a utilizarse, se favorece una emisión más rápida de la instrucción a los cauces de enteros o de coma flotante.
- Si se determina que una instrucción va a hacer una **referencia a memoria**, se puede avanzar su procesamiento.

Así, en la etapa de Pre-Decodificación se añaden una serie de bits a la instrucción que permiten acelerar su decodificación completa en la etapa o etapas posteriores de decodificación. Obsérvese que esto obliga a aumentar el ancho de banda necesario hacia la caché de instrucciones de nivel 1. Este aumento también está multiplicado por el número de instrucciones por ciclo que esta etapa sea capaz de extraer de la caché de nivel 2.



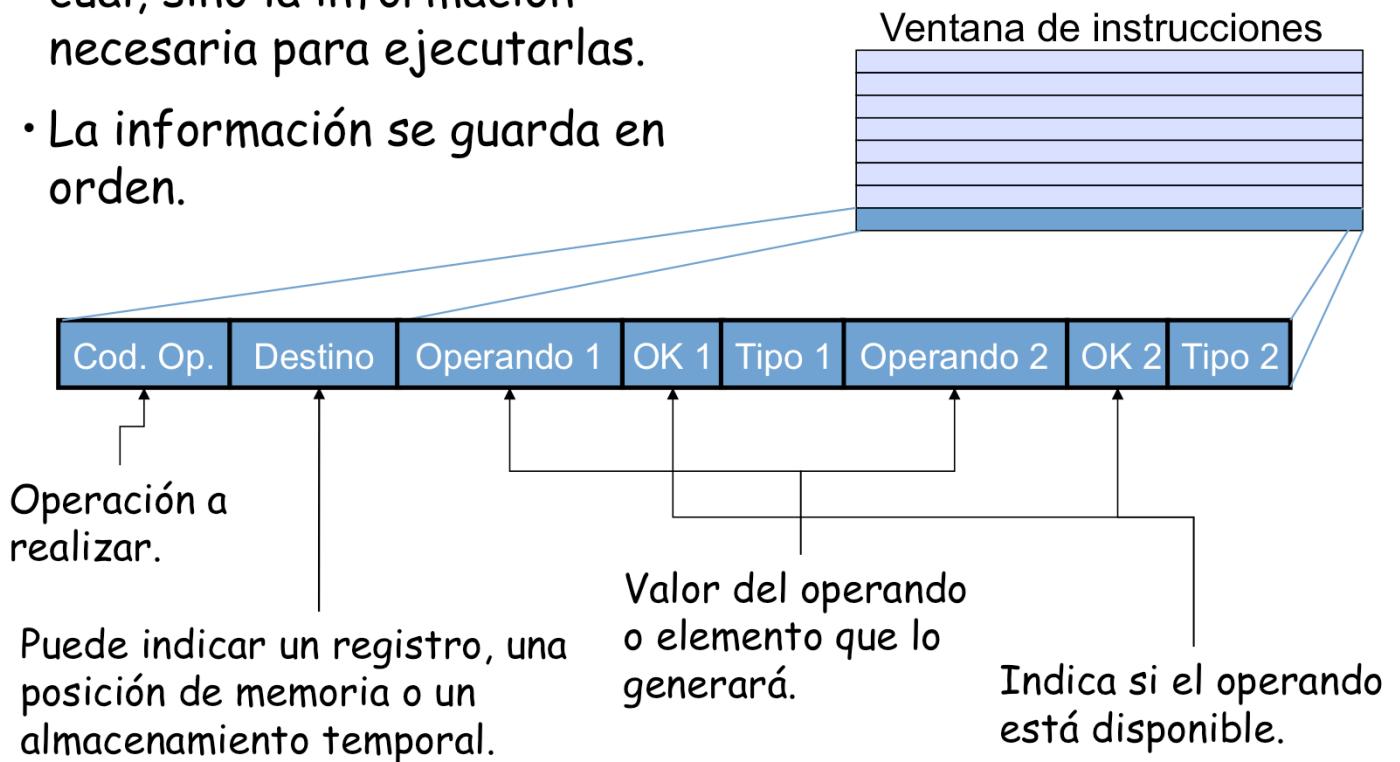
- ✓ Se decodifican completamente las instrucciones (operación concreta).
- ✓ Se identifican los recursos u operandos necesarios.
- ✓ Las instrucciones se depositan, en orden, en la *Ventana de Instrucciones*

La etapa propia de Decodificación se encarga básicamente de realizar la decodificación completa de la instrucción, es decir, de establecer concretamente **la operación** requerida por la instrucción, así como de **identificar los operandos** necesarios. Debido al reparto de trabajo de la decodificación entre varias etapas que ya hemos comentado, la propia obtención de los operandos se realizará posteriormente en otra etapa.

Las instrucciones se van alimentando, en orden, desde el buffer de instrucciones donde las deja la etapa de Extracción y, también en orden, se van poniendo en la **ventana de instrucciones**.

Recuérdese que al igual que las etapas anteriores, y por tratarse de un procesador superescalar, ésta debe procesar varias instrucciones por ciclo.

- No se guardan instrucciones tal cual, sino la información necesaria para ejecutarlas.
- La información se guarda en orden.



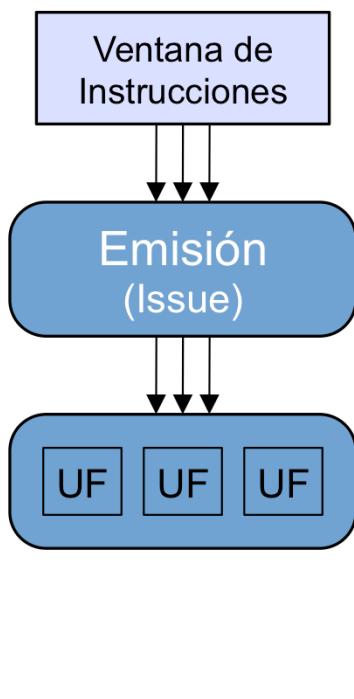
Ya hemos comentado que tras decodificar las instrucciones (en la etapa de Decodificación), éstas se van dejando en una serie de estructuras cuya organización depende de cada implementación.

Una de estas estructuras es la **Ventana de Instrucciones**. Esta estructura se implementa mediante una cola de registros donde se almacenan las instrucciones decodificadas y que están a la espera de ser emitidas a las unidades funcionales correspondientes. Así, la etapa de Emisión se encargará de ir determinando cuáles de las instrucciones de la Ventana de Instrucciones pueden pasar a ejecución, dependiendo de si **están disponibles sus operandos y unidad funcional necesaria**.

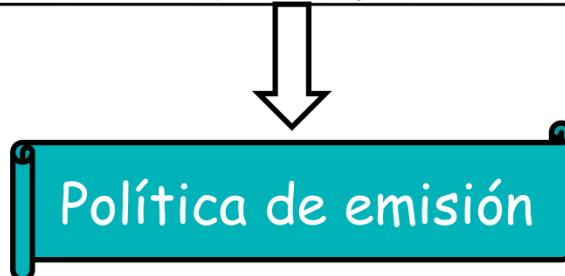
Tras ser decodificadas las instrucciones, lo que se deja en la Ventana de Instrucciones ya no son instrucciones tal cual, sino unas estructuras más elaboradas que contienen toda la información necesaria para la ejecución de la instrucción.

En la figura de arriba se pueden ver los campos de una entrada de la Ventana de Instrucciones:

- **Operación:** Código de la operación correspondiente a la instrucción decodificada.
- **Destino:** Indica el lugar donde se escribirá el resultado de la instrucción. Este destino no tiene por qué ser el registro o dirección de memoria que se indique en la instrucción, sino que puede ser una zona de almacenamiento temporal para evitar riesgos de tipo WAR o WAW.
- **Operandos:** Existen varios campos (tantos como operandos intervengan en la instrucción) dedicados a almacenar los operandos. Aquí hemos supuesto que hay un máximo de dos operandos, por lo que se tienen los dos campos **Operando 1** y **Operando 2**. Asociados a cada operando hay otro campo denominado **OK**, a través del cual se indica si el valor almacenado en el campo **Operando** corresponde al valor del operando, es decir, si el operando está disponible para iniciar la operación en la unidad funcional. Si **OK1=1**, el contenido de **Operando 1** es el valor del operando. Si **OK1=0**, el contenido de **Operando 1** es el identificador del registro o unidad funcional que proporcionará el valor del operando.
- **Tipo:** A veces existen registros para enteros y para datos en coma flotante. En ese caso, se añade un campo más para cada operando, *tipo*, que indica si el registro al que se hace referencia en el campo **Operando** es un registro para enteros, para coma flotante o es un valor inmediato.



- ✓ Si una instrucción está lista para ejecutarse y está disponible la UF correspondiente, se envía a Ejecución.
- ✓ Se pueden emitir varias instrucciones por ciclo.
- ✓ ¿Si hay más instrucciones disponibles de las que se pueden emitir?

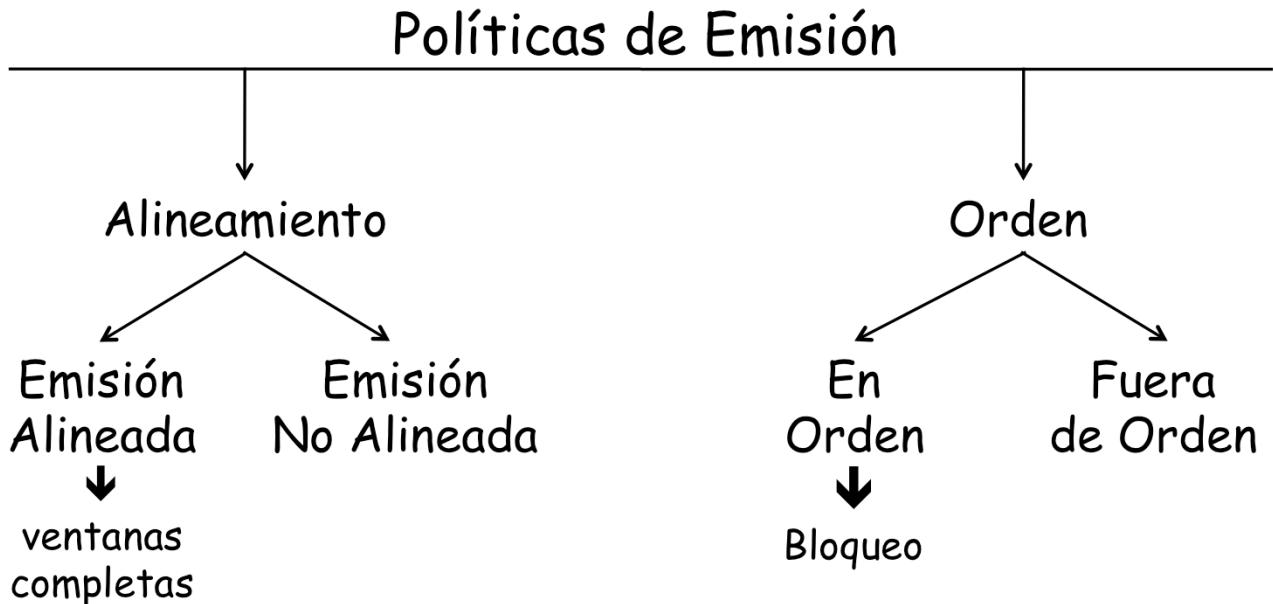


Hemos visto que una vez extraídas las instrucciones se almacenan en una cola o buffer de instrucciones. La etapa de Decodificación toma varias instrucciones por ciclo de la cola y tras ser decodificadas, se añade cierta información y se van depositando en la Ventana de Instrucciones.

La Ventana de Instrucciones es una cola de registros donde se almacenan las instrucciones decodificadas que están a la espera de ser *emitidas* a las unidades funcionales correspondientes. Para ser emitida una instrucción necesita que estén disponibles tanto sus operandos como la unidad funcional que implementa la operación de la instrucción.

La etapa de Emisión se encarga, por tanto, de determinar qué instrucciones pueden emitirse por estar disponibles sus operandos y existir alguna unidad funcional disponible para su ejecución.

La etapa de Emisión tiene ciertas limitaciones en cuanto al número máximo de instrucciones que puede emitir por ciclo. Estas limitaciones tienen que ver con el número de puertos de lectura de las estructuras utilizadas (ventana de instrucciones, banco de registros, buffer de renombramiento, buffer de reordenamiento...). En el caso de que, en un ciclo determinado, haya más instrucciones disponibles para ser emitidas de las que realmente pueden ser emitidas (porque estén disponibles sus operandos y la correspondiente unidad funcional), se aplica alguna política para seleccionar las instrucciones que se van a emitir en ese ciclo. Normalmente, esa política tiene en cuenta el orden de las instrucciones de la Ventana de Instrucciones, es decir, el orden de las instrucciones en el programa.



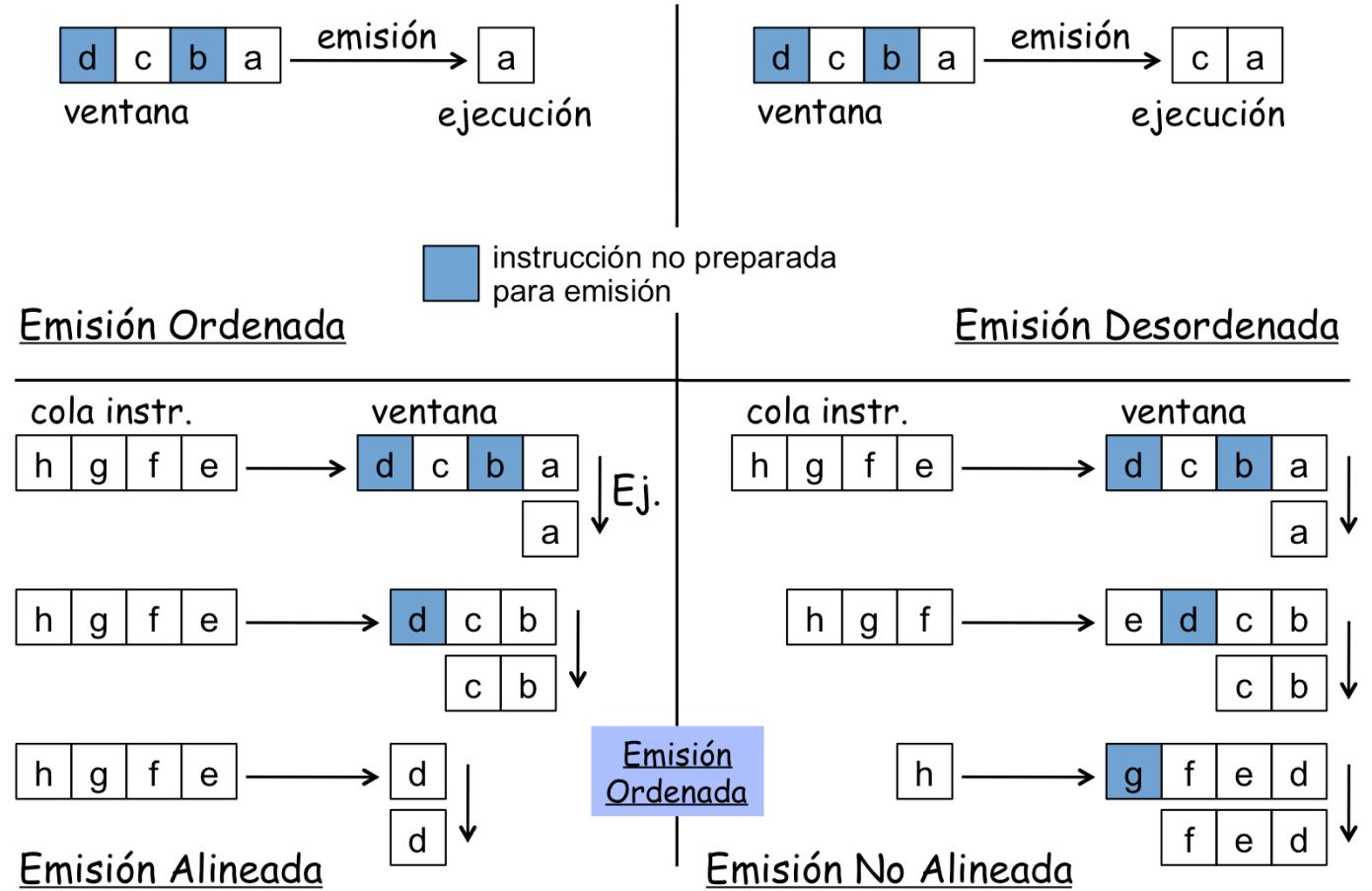
El máximo rendimiento se obtiene con emisión no alineada y fuera de orden.

Las políticas de emisión de instrucciones pueden clasificarse atendiendo al **alineamiento** de la ventana de instrucciones y al **orden** en que se emiten las instrucciones a las correspondientes unidades funcionales.

Según el alineamiento, la emisión puede ser alineada o no alineada. **La emisión es alineada** si no pueden introducirse nuevas instrucciones en la Ventana de Instrucciones hasta que ésta no está totalmente vacía. Es decir, hasta que no se hayan emitido todas las instrucciones que en un ciclo anterior se introdujeron en la Ventana de Instrucciones. En la **emisión no alineada**, mientras que exista un espacio libre en la ventana, se pueden ir introduciendo nuevas instrucciones para ser emitidas.

En cuanto al orden, la emisión puede ser ordenada o desordenada. En la **emisión ordenada** se respeta el orden en que las instrucciones se han ido introduciendo en la Ventana de Instrucciones. Este orden es el mismo en que las instrucciones se han ido decodificando, y coincide con el orden de las instrucciones en el programa. De esta forma, si una instrucción de la Ventana de Instrucciones no puede emitirse, las instrucciones que le siguen tampoco podrían emitirse aunque tuvieran sus operandos y la unidad funcional disponibles. Es decir, existe un bloqueo entre instrucciones: si las instrucciones que aparecen primero en el orden de un programa no pueden emitirse, bloquean la emisión de las instrucciones que las siguen. En el caso de la **emisión desordenada** no existe bloqueo, ya que pueden emitirse todas las instrucciones que dispongan de sus operandos y de una unidad funcional del tipo apropiado.

El máximo rendimiento en la emisión se obtiene con emisión no alineada y desordenada, pues permite aprovechar el paralelismo existente entre las instrucciones. Obviamente, este tipo de emisión también requiere mucho más control de dependencias por parte de procesador.



Veamos unos ejemplos de las políticas de emisión que hemos comentado.

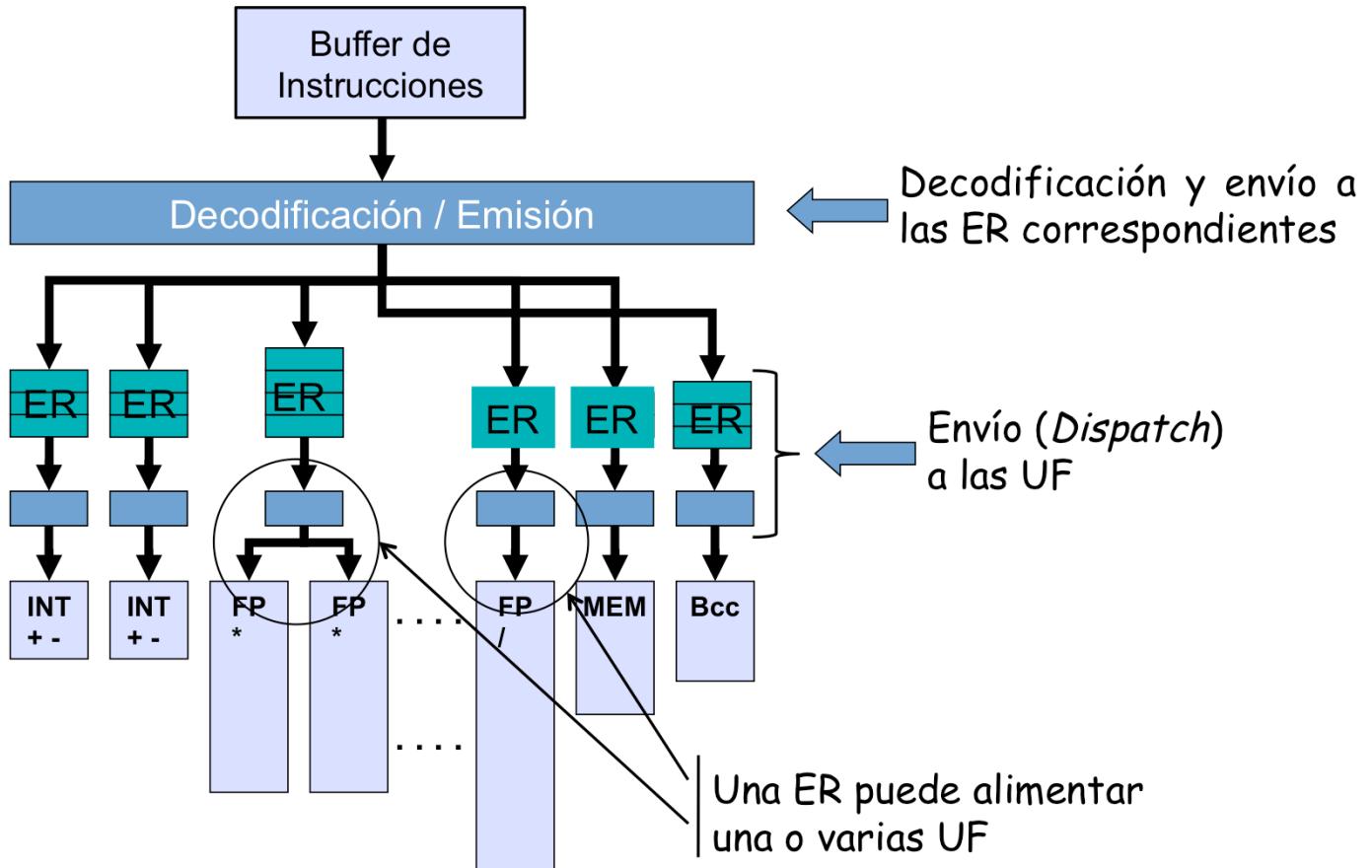
Arriba a la izquierda tenemos la **emisión ordenada**. Están disponibles las instrucciones *a* y *c*, pero la instrucción *b* no lo está, por lo que bloquea la emisión de la instrucción *c* y solamente se emite la instrucción *a*.

A la derecha, la **emisión es desordenada**. Así, aunque la instrucción *b* no está preparada para ejecutarse, no bloquea a la instrucción *c*, por lo que se emiten las dos instrucciones disponibles de la ventana: la *a* y la *c*.

Abajo a la izquierda se muestra un ejemplo de **emisión alineada** ordenada. En el primer ciclo, las instrucciones preparadas de la ventana son la *a* y la *c*. Como la *b* no está preparada, solamente la instrucción *a* pasa a Ejecución, por lo que quedan 3 instrucciones en la ventana, y no se inserta ninguna nueva. Posteriormente, están preparadas las instrucciones *a* y *c*, por lo que ambas se emiten. Solamente hay una instrucción en la ventana, pero se continua sin llenar los huecos con nuevas instrucciones. Al emitir la instrucción *d*, la ventana queda vacía y ya se pasa a rellenarla con nuevas instrucciones.

Abajo a la derecha tenemos el ejemplo de **emisión no alineada** ordenada. Aquí se puede ver que a medida que se van emitiendo instrucciones, cada hueco que queda en la ventana se llena con nuevas instrucciones del buffer de instrucciones.

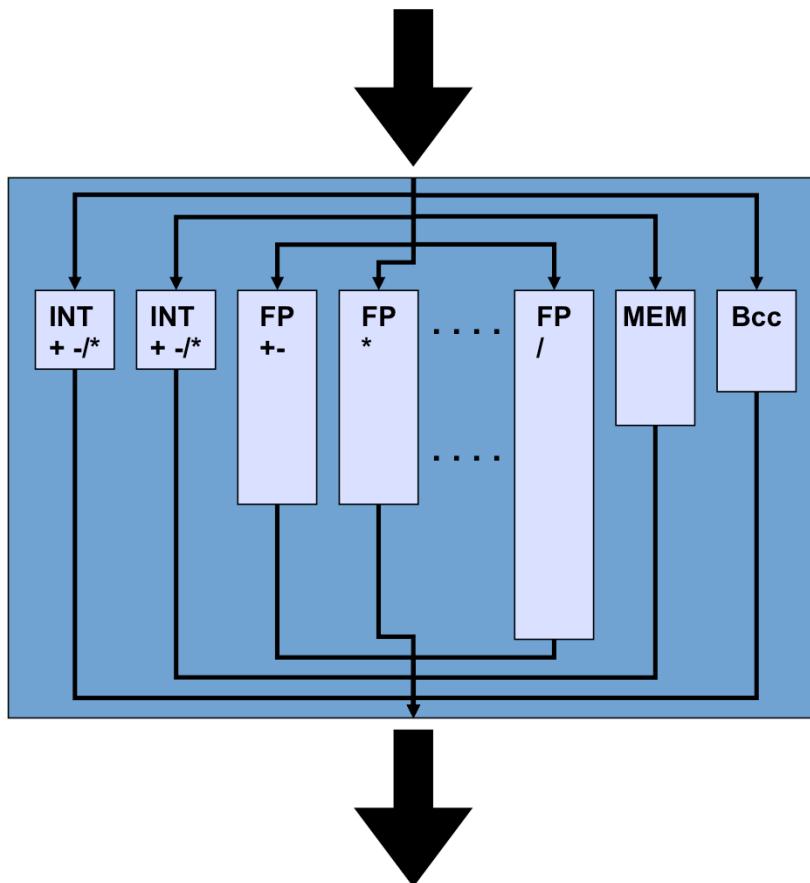
Como ya dijimos, la emisión no alineada y desordenada permite mejorar el rendimiento de la unidad de Emisión, aprovechando el paralelismo entre instrucciones. Como vemos en el ejemplo de la emisión no alineada, es posible emitir hasta 3 instrucciones por ciclo, mientras que en el caso de la emisión alineada, se emiten, como mucho, dos instrucciones por ciclo.



En muchas microarquitecturas superescalares es frecuente que la Ventana de Instrucciones se distribuya en varias estructuras que reciben el nombre de **Estaciones de Reserva**.

La idea consiste en que, en lugar de existir una única ventana desde donde se emiten las instrucciones a las distintas unidades funcionales, existe una estructura, similar a la ventana, pero específica para cada unidad funcional o para un conjunto homogéneo de unidades funcionales (por ej. una para las unidades funcionales con enteros, otra para las de coma flotante, etc.).

En una microarquitectura con Estaciones de Reserva, las funciones de la etapa de Emisión se suelen repartir en dos partes. En primer lugar, las instrucciones decodificadas deben pasarse a la estación de reserva adecuada (desde la que se pueda acceder a la unidad funcional correspondiente), y es en esa estación de reserva donde cada instrucción debe esperar a que le toque el turno de pasar a la unidad funcional para ejecutarse, una vez que disponga de sus operandos. Esta **primera parte** se suele seguir llamando *Emisión (Issue, I)*, y se implementa en la etapa de Decodificación, que pasa a denominarse etapa de *Decodificación/Emisión*. La **segunda parte** de la emisión se denomina *Envío (dispatch)*, y se ocupa de determinar qué instrucciones de la estación de reserva tienen sus operandos disponibles y pueden pasar a ejecutarse cuando haya una unidad funcional apropiada disponible.



- ✓ Una o varias UF por Estación de Reserva.
- ✓ UF del mismo tipo replicadas.
- ✓ Orden arbitrario de finalización de ejecución

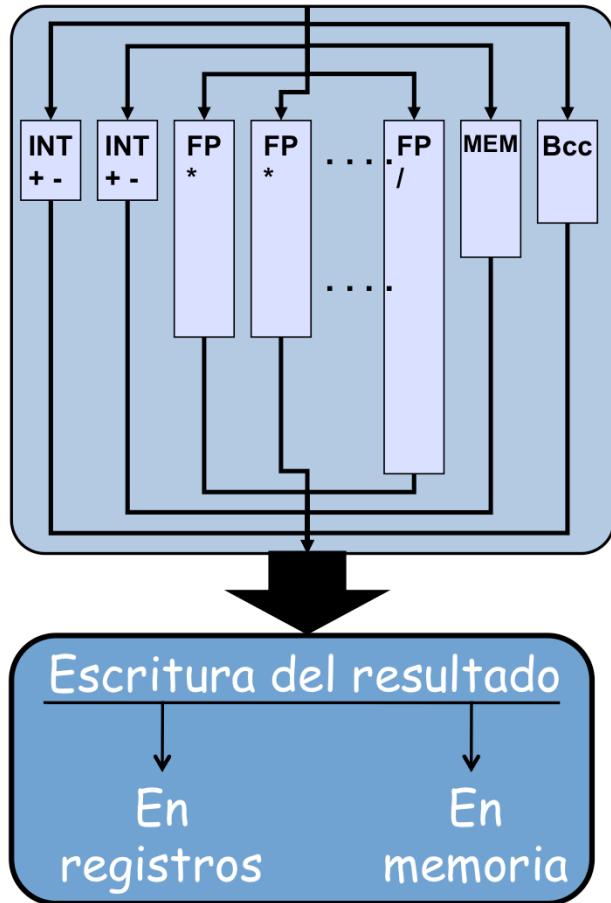
Riesgos de datos  
WAR y WAW

Aquí, poco tenemos que decir sobre la etapa de Ejecución, pues ya sabemos que está compuesta por las distintas unidades funcionales que ejecutan las distintas operaciones que pueden tener codificadas las instrucciones.

Cada unidad funcional puede implementar una o diversas operaciones. Así, puede haber una UF para cualquier operación con enteros, mientras que la aritmética en coma flotante, más compleja, puede tener sus operaciones distribuidas en distintas UF (por ej. una para sumas y restas, otra para multiplicaciones y otra para divisiones).

También debemos recordar que puede haber unidades funcionales replicadas, para permitir la ejecución en paralelo de varias instrucciones del mismo tipo.

Por último, tener presente que, puesto que se emiten múltiples instrucciones por ciclo, puede comenzar simultáneamente la ejecución de varias instrucciones en paralelo. El orden de finalización de la ejecución de las instrucciones es arbitrario, pues depende del número de ciclos que se requiere para cada operación, lo que conlleva riesgos de datos de tipo WAR y WAW.



- ✓ Orden arbitrario de emisión de las instrucciones
- ✓ Escritura del resultado fuera de orden

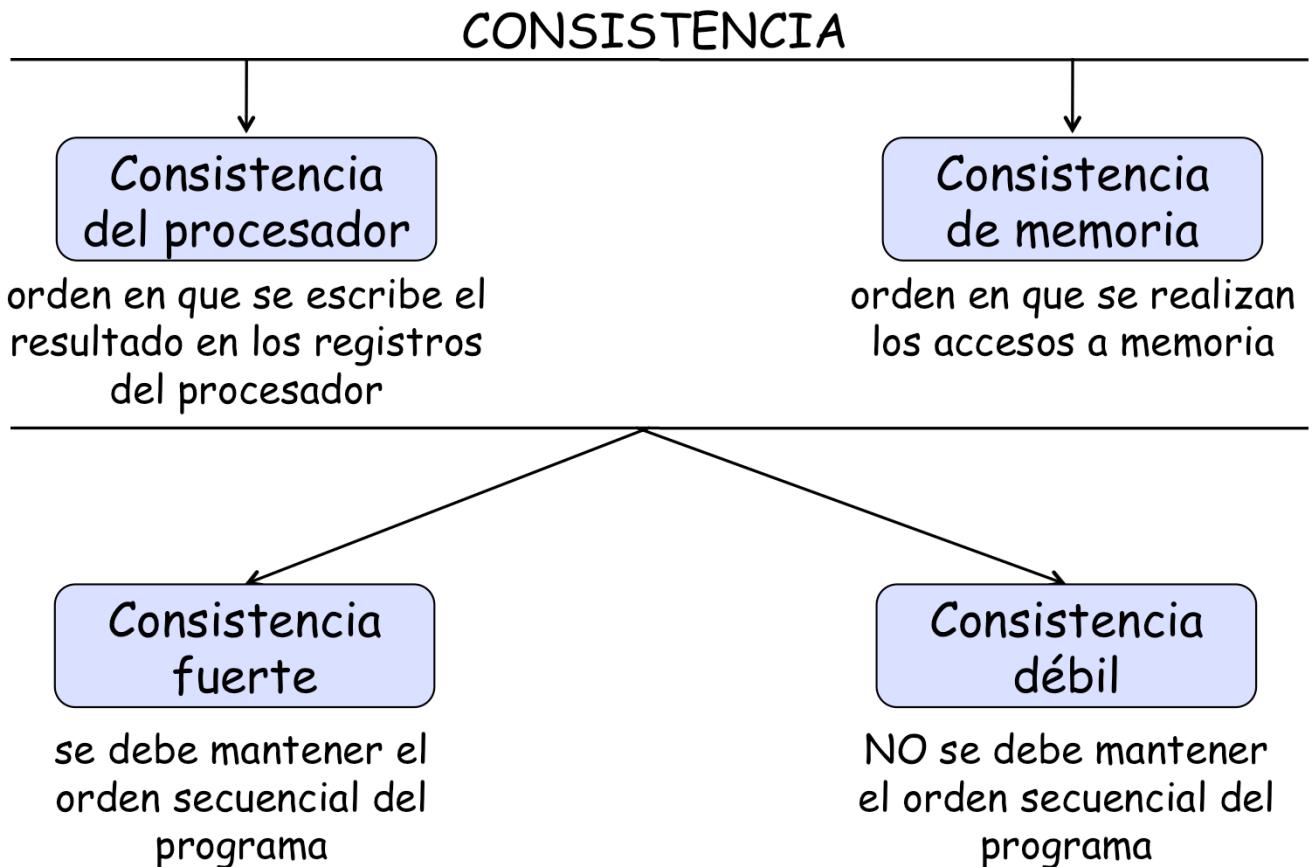
Riesgos de datos  
WAR y WAW

Se debe cuidar la consistencia (semántica correcta)

Al terminar la etapa de Ejecución, lo que resta por hacer es la escritura del resultado en el destino indicado en la instrucción, bien sea en un registro general o en una posición de memoria. Como ya hemos planteado anteriormente, el orden de la escritura de los resultados de las múltiples instrucciones que se ejecutan en paralelo puede acarrear problemas. Las acciones tomadas para la resolución de estos problemas más la propia escritura de los resultados constituyen la etapa de Finalización.

Ya hemos visto que las instrucciones se pueden emitir o enviar a ejecución fuera de orden. Es más, incluso llegando en orden, la terminación de la ejecución puede producirse fuera de orden, debido a las distintas duraciones de las instrucciones. Si la escritura del resultado se produce inmediatamente después de la finalización de cada instrucción, tenemos que la escritura de los resultados se produce fuera de orden, y esto, como ya hemos visto anteriormente, conlleva riesgos de datos de tipo WAR y WAW.

Cuando se procesan instrucciones en paralelo, el orden en que termina el procesamiento de estas instrucciones puede variar con respecto al orden que las instrucciones tenían en el programa, pero independientemente de si la terminación del procesamiento es en orden o fuera de orden, se debe disponer de los dispositivos adecuados para mantener la semántica del programa, es decir, para que el programa se ejecute correctamente.



La consistencia secuencial de un programa se refiere a:

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

Cuando se procesan instrucciones en paralelo, el orden en que termina el procesamiento puede variar respecto al orden correspondiente que tenían esas instrucciones en el programa, pero **debe existir consistencia entre el orden en que se completan las instrucciones y el orden secuencial que tienen en el programa.**

Hay dos tipos de consistencia que se deben tener en cuenta en el procesamiento de las instrucciones:

- La **consistencia del procesador** se refiere al orden en que se escriben los resultados en los registros del procesador.
- La **consistencia de memoria** se refiere al orden en que se realizan los accesos a memoria.

Si para que un programa se ejecute correctamente, el orden al que hacen referencia tanto la consistencia del procesador como la consistencia de memoria debe coincidir con el orden en que las instrucciones o los accesos a memoria aparecen en el programa, se dice que hay **consistencia fuerte**.

Se habla de **consistencia débil** si el orden de procesamiento de las instrucciones y de los accesos a memoria puede ser distinto al orden del programa y se mantiene la semántica del mismo. Para permitir esta consistencia deben resolverse los posibles riesgos WAR y WAW.

## Consistencia de Memoria

```

LOOP LD    R1 ,200 (R2)
      DMUL R1 ,R1 ,R6
      SD    R1 ,200 (R2)

      LD    R3 ,300 (R2)
      DMUL R3 ,R3 ,R6
      SD    R3 ,300 (R2)

      ADDI R2 ,R2 ,#1
      SUBI R4 ,R4 ,#1
      BNZ   R4 ,LOOP
  
```

El tiempo de ejecución de una instrucción de acceso a memoria puede ser muy alto

Con consistencia fuerte se retrasan mucho las instrucciones que le siguen

Una consistencia de memoria débil mejora mucho las prestaciones si las instrucciones que siguen no tienen dependencias de las primeras

Hay que tener en cuenta que el tiempo de acceso a memoria puede ser considerable si la dirección a la que se accede no está en la caché y hay que acceder a memoria principal. En este caso, si la instrucción es **LOAD**, se retrasarían considerablemente las instrucciones que le sigan (en el caso de que se pretenda mantener el orden del programa). Se podrían mejorar las prestaciones del procesador si se deja que puedan ejecutarse instrucciones que sigan a **LOAD**, pero que no dependan del resultado de esa carga. Incluso se podría permitir la ejecución de instrucciones **LOAD** y **STORE** que están detrás de la instrucción **LOAD** que genera la falta de caché. En este caso, una consistencia de memoria débil que permita una ejecución desordenada de los accesos a memoria enmascararía los fallos de caché, mejorando considerablemente las prestaciones.

Por otra parte, si una instrucción **STORE** no puede iniciarse porque no está disponible el operando a almacenar, se podría aprovechar que los buses de acceso a memoria no se están utilizando, para arrancar alguna otra instrucción de acceso a memoria que esté después del **STORE**. De esta manera se mejora el aprovechamiento del sistema, sobre todo si se piensa que un **LOAD** que esté detrás del **STORE** puede dar lugar a faltas de caché y necesitar un mayor tiempo de acceso.

En el fragmento de programa de arriba, hay dos instrucciones de carga tras las cuales hay una instrucción de multiplicación que utiliza como operando el resultado de la carga, y una instrucción de almacenamiento del resultado de la multiplicación. Hasta que no se termine la multiplicación, no se puede iniciar la instrucción **STORE** correspondiente y, si se mantiene una consistencia de memoria fuerte, tampoco podría iniciarse el segundo **LOAD**, a pesar de no tener dependencias con el **STORE** anterior, y tampoco podría arrancarse la segunda multiplicación. Si, como es usual, la multiplicación introduce un retardo considerable, las prestaciones serían realmente bajas. En cambio, si se permite que el segundo **LOAD** adelante al primer **STORE**, aparece un mayor paralelismo entre instrucciones que la arquitectura podría aprovechar. Mientras se está ejecutando la multiplicación, los buses de acceso a memoria se utilizan para el **LOAD** ya que no se puede efectuar el acceso correspondiente al **STORE** que debe esperar el resultado de la multiplicación.

## Consistencia de Memoria

### Direcciones diferentes

```

LOOP LD    R1,200 (R2)
      DMUL R1,R1,R6
      SD    R1,200 (R2)

      LD    R3,300 (R2)
      DMUL R3,R3,R6
      SD    R3,300 (R2)

      ADDI R2,R2,#1
      SUBI R4,R4,#1
      BNZ   R4,LOOP
  
```

### ¿Direcciones diferentes?

```

LOOP LD    R1,200 (R2)
      DMUL R1,R1,R6
      SD    R1,200 (R2)

      LD    R3,300 (R5)
      DMUL R3,R3,R6
      SD    R3,300 (R5)

      ADDI R2,R2,#1
      SUBI R4,R4,#1
      BNZ   R4,LOOP
  
```

**Problemas**

Puede hacerse "adelantamiento especulativo"  
y, en caso de fallo,  
se deben deshacer las instrucciones ejecutadas erróneamente

En el fragmento de programa de la izquierda, está claro que no hay dependencias entre la primera instrucción STORE y la segunda instrucción LOAD que la adelanta, ya que las direcciones de memoria a las que acceden son diferentes, pues independientemente del valor que pueda tener el registro R2, el desplazamiento es distinto (200 y 300), luego la dirección efectiva será distinta.

Sin embargo, esta situación no se da siempre, y es frecuente que exista cierta ambigüedad en las direcciones de memoria de las instrucciones, de tal manera que hasta que no se calcula la dirección efectiva, no se puede asegurar que las direcciones sean distintas. Así, veamos el caso del fragmento de programa de la derecha. Las direcciones de acceso a memoria de la primera instrucción STORE y la segunda LOAD son 200 (R2) y 300 (R5), respectivamente. Como vemos, no se puede asegurar que R2+200 sea distinto a R5+300.

Si estas direcciones fuesen iguales, estaríamos ante una dependencia de datos RAW entre las dos instrucciones, por lo que la instrucción de carga no podría adelantar a la de almacenamiento.

Si, aún en el caso de que exista esta ambigüedad, se permite que la instrucción LOAD adelante a la STORE, se dice que se ha producido un **adelantamiento especulativo**.

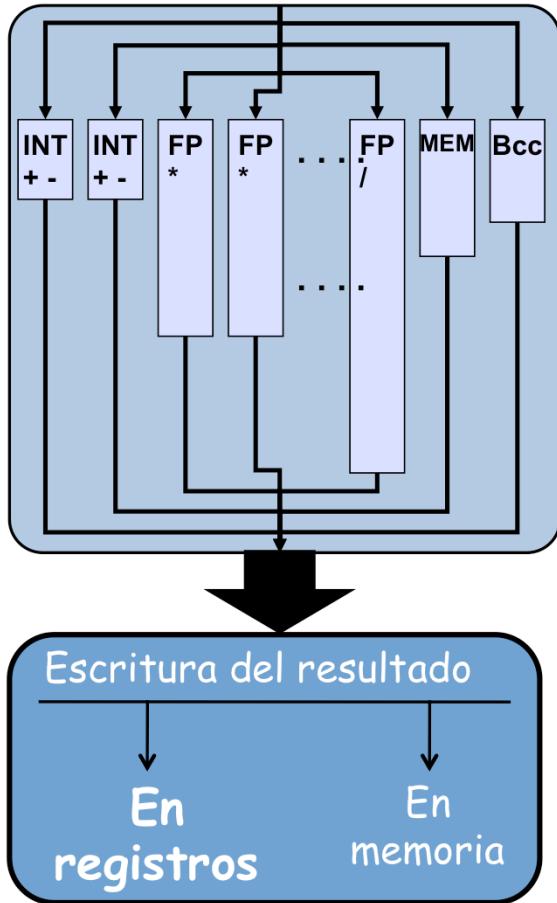
Cuando se permite el adelantamiento especulativo, si una vez calculadas las direcciones efectivas, se comprueba que las dos direcciones coinciden habrá que deshacer el acceso adelantado a memoria (la instrucción LOAD) puesto que no se habrá respetado la dependencia RAW entre ellas.

Deshacer las acciones realizadas por la instrucción LOAD supone que se habrá perdido más tiempo que si no se hubiese permitido el adelantamiento especulativo. Esto quiere decir que se debería acertar la mayoría de las veces si se quiere sacar provecho del adelantamiento especulativo.

## Consistencia de Memoria

La tendencia actual es la **consistencia de memoria débil** ya que permite grandes ahorros de tiempo

Por lo que hemos visto, queda claro que, respecto a la consistencia de memoria, merece la pena utilizar consistencia débil, pues puede proporcionar grandes ahorros de tiempo, ya que permite aprovechar los grandes lapsos de tiempo que supone el acceso a memoria arrancando la ejecución de otras instrucciones posteriores sin haber iniciado o finalizado las primeras.



## Consistencia del Procesador



Recordemos que la consistencia del procesador se refiere al **orden en que se escriben los resultados en los registros del procesador**. Una vez que la operación de una instrucción se ha ejecutado, la instrucción termina su procesamiento actualizando el correspondiente registro de la arquitectura y abandona el cauce.

Como se ha visto anteriormente, si se utiliza el renombramiento de registros para evitar los efectos de los riesgos WAW y WAR y aprovechar el máximo paralelismo entre instrucciones, al final de la ejecución de la operación, el resultado se encontrará en una de las líneas de ese buffer de renombramiento.

El problema que se plantea ahora es el de decidir el momento en que los resultados que están temporalmente en el buffer de renombramiento se escriben en los registros de la arquitectura, completando así el procesamiento de las instrucciones que generaron los resultados.

El orden en que el procesamiento de las instrucciones se completa (y por lo tanto, el orden en que los registros de la arquitectura se actualizan con los resultados) puede coincidir o no con el orden en que las instrucciones estén en el programa pero, en cualquier caso, debe respetarse la semántica del programa, teniendo en cuenta las dependencias entre las instrucciones para que el resultado final sea correcto.

En el caso de la consistencia del procesador, existe una **tendencia hacia la finalización ordenada de instrucciones (consistencia fuerte)**, dado que una finalización desordenada no aporta mejoras claras en cuanto a prestaciones y a que, gracias al uso de estructuras como el **buffer de reordenamiento (Reordering Buffer, ROB)**, se permite una emisión y ejecución desordenada de las instrucciones que permite aprovechar el paralelismo entre ellas y, después, tener una finalización ordenada que asegura el mantenimiento de la semántica del programa original. Es decir, para tener una consistencia débil (finalización desordenada) se requiere un hardware adicional que se encargue de resolver los riesgos WAR y WAW, mientras que si se mantiene el orden de las instrucciones (consistencia fuerte), apenas se penaliza el rendimiento, y solamente se requiere una sencilla estructura de datos para asegurar el orden de finalización (el ROB).

## Buffer de Reordenación (ROB)

Las instrucciones entran en orden de decodificación

Reg. Destino	Unidad Funcional	Resultado	ok	Estado
...	...	...	...	...
r1	int_mult	33	1	f
-	store	-	0	x
r4	int_mult	27	1	f
r2	int_mult	-	0	x
...	...	...	...	...

Cola →

Cabecera ←

También permite

- Renombrado de registros
- Adelantamiento especulativo

Como vemos en la figura, el ROB (Buffer de Reordenamiento) es un buffer circular en el que en cada una de sus líneas o entradas se va introduciendo información de cada una de las instrucciones en el mismo orden que han sido decodificadas (el orden del programa).

El ROB contiene un **puntero de cabecera** que señala a la siguiente posición libre, es decir, donde se insertará la siguiente instrucción decodificada, y un **puntero de cola**, que señala la siguiente instrucción que se retirará del buffer cuando termine su procesamiento.

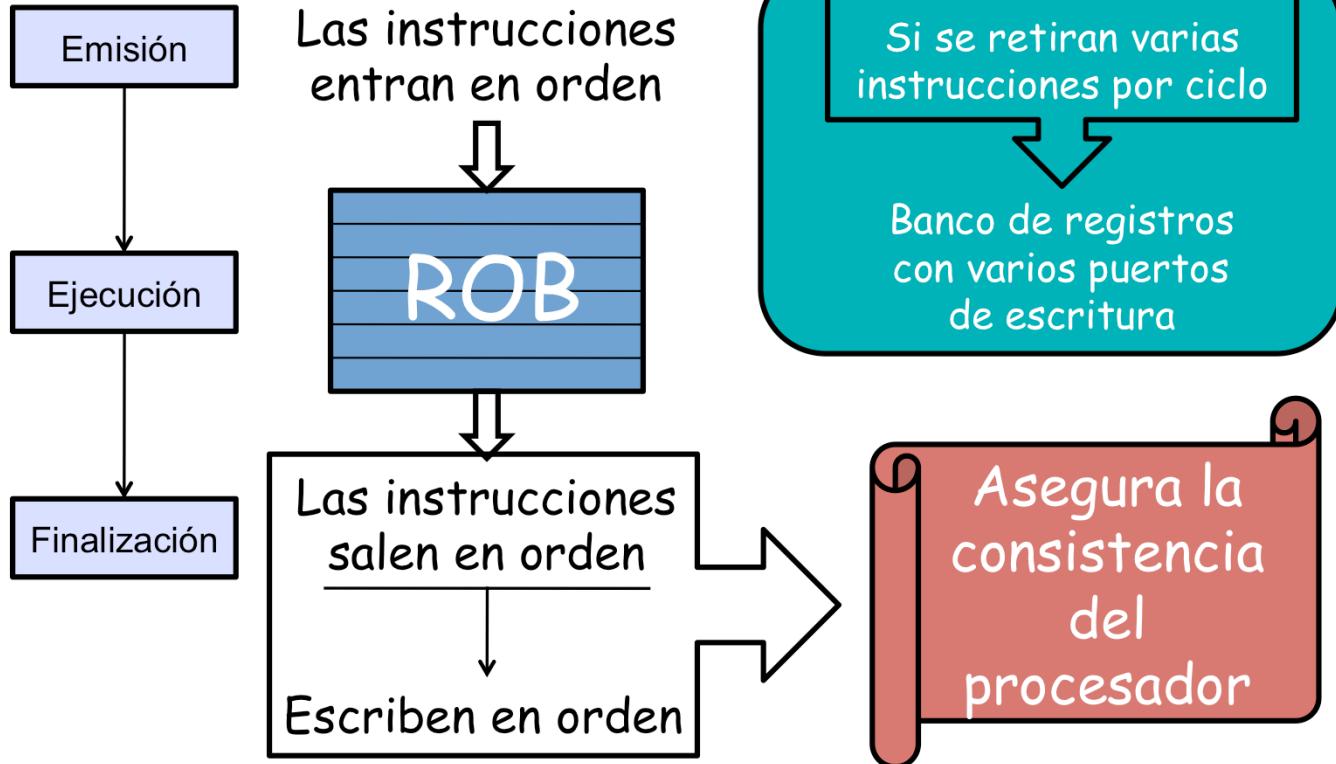
Cada una de las entradas contiene los siguientes campos:

- **Registro de destino:** indica el registro de la arquitectura en el que se va a escribir el resultado de la instrucción cuando se retire.
- **UF (Unidad Funcional):** señala la unidad funcional donde se va a ejecutar la instrucción y que generará el valor que se almacenará en el campo **Resultado**.
- **OK:** es un campo de un bit que indica si el contenido del campo Resultado es válido o no.
- **Estado:** Indica el estado de procesamiento de la instrucción. Así, puede tener los valores correspondientes a *Emitida (i)*, *en Ejecución (x)* y *Finalizada la ejecución (f)*. Este campo permite determinar si una instrucción que haya finalizado su ejecución podría retirarse escribiendo su resultado cuando se hayan retirado todas las instrucciones que le anteceden en el ROB.

El ROB también se utiliza como Buffer de Renombramiento, pues se guardan los resultados de las operaciones en el campo *Resultado*.

También permite el adelantamiento especulativo, pues si se decide que deben eliminarse los efectos de una instrucción indebidamente ejecutada, simplemente se elimina su entrada del ROB, con lo que su resultado nunca llega a escribirse.

## Consistencia del Procesador



Como ya hemos dicho, las instrucciones se introducen en el ROB en el orden estricto del programa, marcándose como *emitidas*, y pasando posteriormente por los estados de *en ejecución* y *finalizada*.

Una instrucción solo se puede retirar del ROB (y al retirarse se produce la actualización del resultado en un registro de la arquitectura, finalizando su procesamiento) si ha terminado la ejecución de su operación (pasa a *finalizada*) y todas las que la preceden en el ROB también se han retirado.

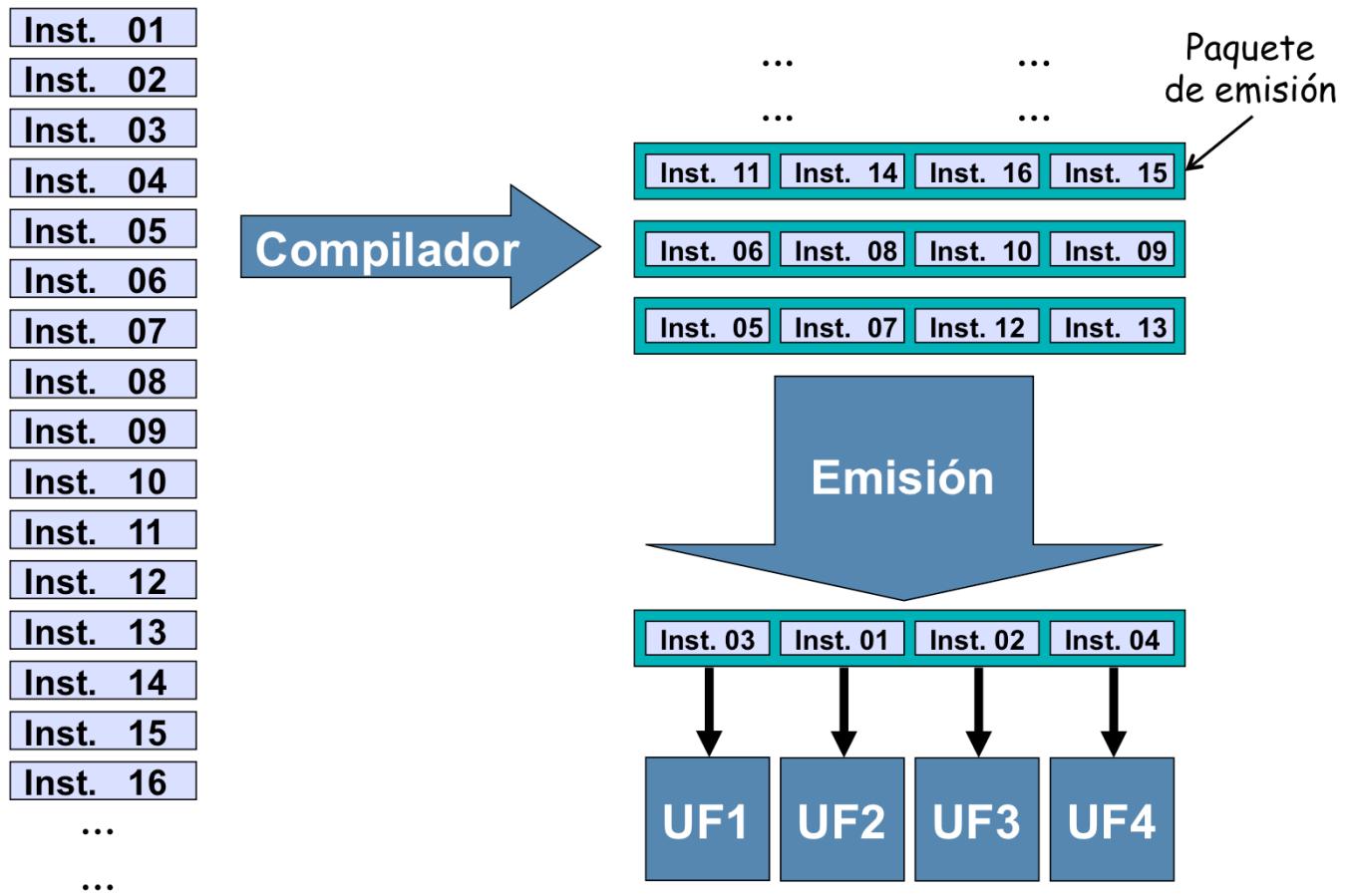
La consistencia se mantiene porque solo las instrucciones que se retiran del ROB (es decir, que completan su procesamiento) escriben en los registros de la arquitectura, y esas instrucciones se retiran en el orden en que aparecen en el programa.

Obsérvese que si en la etapa de Finalización se retiran varias instrucciones por ciclo, quiere decir que se debe poder escribir el resultado de varias instrucciones en el mismo ciclo, por lo que el banco de registros debe tener varios puertos de escritura (tantos como instrucciones se puedan retirar simultáneamente del ROB).

Si en un momento dado se pueden retirar del ROB simultáneamente varias instrucciones que además escriben su resultado en un mismo registro de destino, solamente se debe considerar la última de ellas y desechar las anteriores, pues tal situación podría ser la consecuencia de un riesgo WAW (*Write After Write*).

En el texto de Ortega y otros (apartado 3.3.4, fig. 3.30) hay un buen ejemplo del orden en que se van retirando las instrucciones de un buffer de reordenamiento.

## 3. El modelo VLIW (*very long instruction word*)

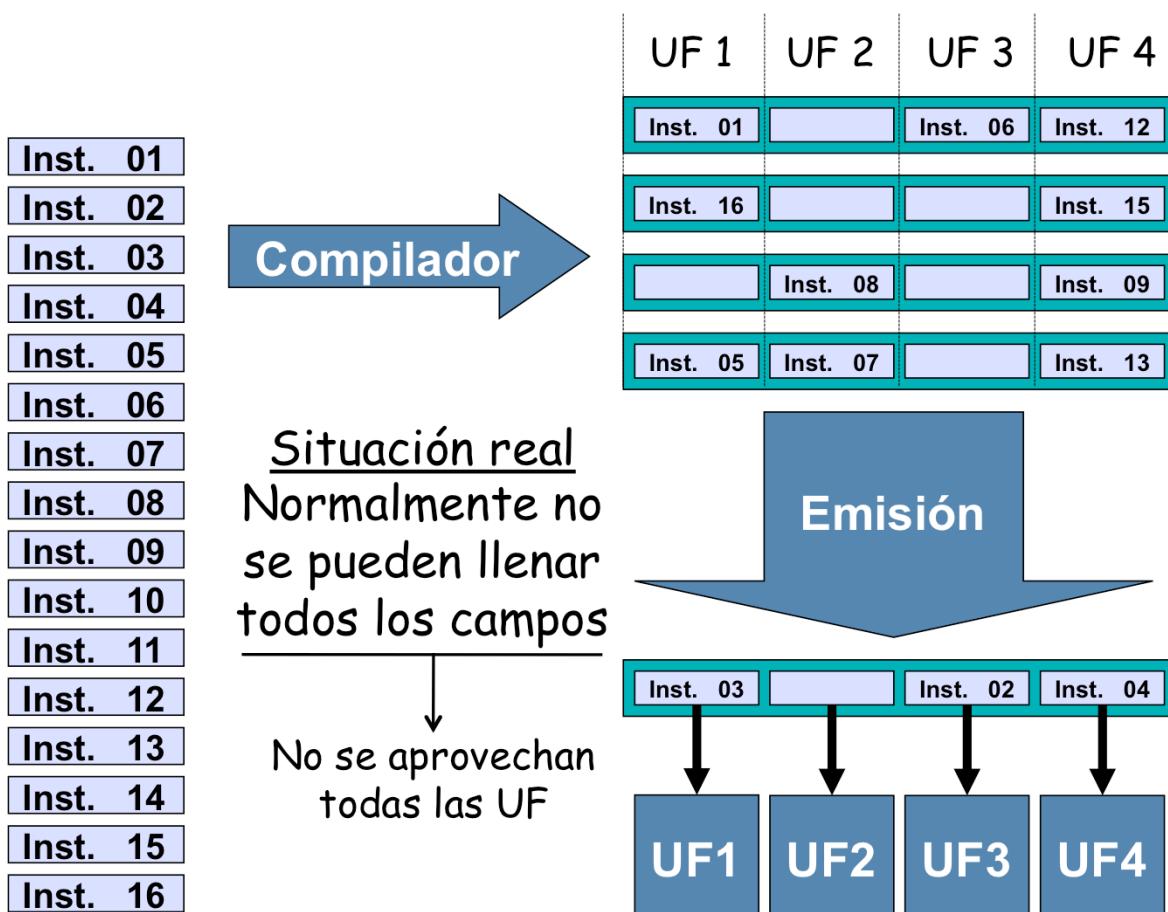


Las arquitecturas VLIW también corresponden a los procesadores de emisión múltiple, es decir, además de que se ejecutan varias instrucciones en paralelo (en la etapa E), también se emiten varias instrucciones en el mismo ciclo de reloj.

Pero a diferencia de los procesadores superescalares, el encargado de planificar, en cada ciclo, las instrucciones que van a ejecutarse en paralelo, es el compilador. El compilador se encarga de averiguar las dependencias entre la secuencia de instrucciones y de ir formando **paquetes de emisión** que contienen varias instrucciones. Todo ello, sin alterar, claro está, la semántica del programa.

El paquete de emisión es tratado por las etapas de Extracción y Emisión del procesador como si fuera una única instrucción. Desde la etapa de emisión se emite el paquete a una estación en la que se descomponen en las varias instrucciones que contiene y cada una se envía a su unidad funcional correspondiente, con lo que se ejecutan todas ellas en paralelo.

Un paquete de emisión está formado por varios campos, donde cada uno de ellos corresponde a una unidad funcional de procesador. El compilador va formando grupos de instrucciones sin dependencias y cada una de ellas se pone en el campo correspondiente a su unidad funcional. Al llegar el paquete de emisión al procesador, no hay que ocuparse de las dependencias (datos y control) y riesgos estructurales, pues ya están resueltas de antemano, y en cada ciclo se van emitiendo estas macro-instrucciones o paquetes. Es decir, la unidad de emisión es el paquete.

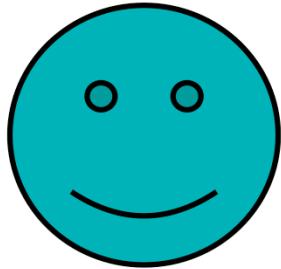


Los distintos campos del paquete de emisión se corresponden con las unidades funcionales de que dispone el procesador. Así, es normal que se disponga de una unidad de aritmética entera, otra para multiplicaciones y divisiones, otra de coma flotante, otra de carga/almacenamiento, otra de tratamiento de bifurcaciones...

Y estos campos deben rellenarse con instrucciones del tipo correspondiente (una suma/resta, una multiplicación o división, una carga/almacenamiento, una bifurcación...).

Pero claro, no siempre se dispone de todas estas instrucciones para cada paquete, que, además, debe estar compuesto de instrucciones sin dependencias de datos.

Por esto, es normal que los paquetes de emisión no estén siempre llenos, sino que algunos de sus campos están vacíos, o contienen instrucciones NOP.



✓ Ventana de instrucciones "en compilación" más grande.

↓  
Mejor paralelización

✓ Hardware del procesador es más simple pues el compilador se encarga de encontrar el paralelismo.

↓  
Más espacio libre en el chip

↓  
Mayor número de unidades funcionales

↓  
Aumenta el paralelismo a nivel máquina (MLP)

↓  
Un banco de registros más numeroso

↓  
Facilita ejecución especulativa  
No es necesario el buffer de reordenación

La implementación de un procesador VLIW consigue el mismo efecto que un superescalar, pero el diseño del VLIW se hace sin las dos partes más complejas del diseño del superescalar:

- Ya que las instrucciones VLIW especifican explícitamente varias operaciones independientes (es decir, que pueden ejecutarse de forma paralela) no es necesario un hardware para decodificar que trate de encontrar el paralelismo en una secuencia de instrucciones.
- Se elimina el buffer de reordenación de las etapas finales. A continuación veremos el porqué.

Los procesadores VLIW se apoyan en el compilador para generar código explícitamente paralelo, lo cual tiene las siguientes ventajas:

- El compilador puede formar una ventana de instrucciones mucho más grande que la del procesador superescalar (dejando a su vez, espacio libre en el chip), con lo que resulta más fácil encontrar instrucciones que puedan ejecutarse en paralelo. También se elimina del procesador la lógica para encontrar instrucciones independientes, que puede ralentizar velocidad del reloj.
- El hardware es mucho más simple, ya que es el compilador el que se encarga de crear un flujo de instrucciones paralelas.
- Con un hardware más sencillo, en el que se eliminan las unidades encargadas de buscar las dependencias de datos y control, y de ocuparse de los riesgos estructurales, se deja mucho espacio libre en el área del chip, lo que permite emplearlo para aumentar el número de unidades funcionales y para disponer de un mayor banco de registros. Con un buen banco de registros, los resultados de instrucciones ejecutadas de manera especulativa se pueden poner temporalmente en registros. Así, si la ejecución de una instrucción ejecutada de manera especulativa resulta ser un fallo, su resultado se desecha y no se lleva finalmente al registro de destino. Obsérvese que esto hace innecesario el buffer de reordenación de las etapas finales.



✓ Mayor complejidad del compilador, al tenerse que ocupar de la paralelización

✓ Dificultad para llenar todos los campos de las instrucciones

Se desaprovechan los recursos del procesador

Mayor tamaño del código

✓ No se aprovechan las nuevas versiones de procesadores con nuevas distribuciones de las unidades funcionales.

(Solución: Traducir Binario → Binario)

Los procesadores VLIW también tienen ciertos inconvenientes:

- La arquitectura VLIW simplemente mueve la complejidad del hardware (paralelización de instrucciones) al software, por lo que se requiere construir un compilador mucho más complejo que los convencionales. No obstante, se puede defender que este compromiso resulta beneficioso, pues aquí, la complejidad solo se paga una vez (al escribir el compilador), mientras que en el superescalar, cada vez que se fabrica el chip, sale más caro. Además, suele resultar más fácil tratar con la complejidad en el diseño del software que en el diseño del hardware.
- No resulta fácil asignar con instrucciones todos los campos del paquete de emisión, por lo que a menudo hay campos vacíos. Lo que esto significa es que se desaprovechan los recursos del procesador. (Obsérvese que esto también es normal que suceda en los procesadores superescalares).
- Debido al problema anterior, el tamaño del código se incrementa, pues hay muchos huecos vacíos y desaprovechados en los paquetes de emisión.
- Cuando la tecnología de integración de los chips permita la construcción de procesadores con más unidades funcionales, los programas compilados para las generaciones anteriores de procesadores, no se ejecutarán o no aprovecharán los nuevos procesadores, ya que la codificación de los paquetes de emisión dependen del número de unidades funcionales. Una solución a este problema puede ser la construcción de compiladores que traduzcan de código binario a código binario, es decir, tomar un programa binario compilado para la generación anterior y convertirla a un nuevo programa binario para la nueva generación de procesadores.