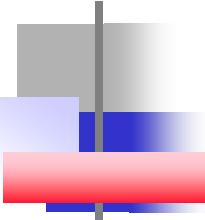


Arquitectura de Computadores



TEMA 3

Lanzamiento múltiple, Límites de ILP, Multithreading

DEPARTAMENTO DE
ARQUITECTURA DE COMPUTADORES
Y AUTOMÁTICA

Curso 2024-2025

- ❑ Introducción: CPI < 1
- ❑ Lanzamiento múltiple de instrucciones: Superescalar
 - Superescalar con planificación estática.
 - Superescalar con planificación dinámica (Out-of-Order):
 - Anatomía del pipeline: alternativas
 - Caso de estudio
- ❑ Límites de ILP
- ❑ Ejemplo: Implementaciones x86, evolución μ -arquitecturas Intel
- ❑ Thread Level Parallelism y Multithreading
- ❑ Bibliografía
 - Básica:
 - Capítulo 3 de Hennessy & Patterson 6th ed., 2019
 - Capítulos 3-8 de González, Latorre & Magklis, 2011. ISBN: 9781608454532
Versión electrónica disponible en la Biblioteca de la UCM
 - Complementaria:
 - Capítulo 3 de Dubois, Annavarapu & Stenstrom, 2012

- ¿Por qué limitar a una instrucción por ciclo?
- Objetivo: $CPI < 1 \rightarrow$ Lanzar y ejecutar simultáneamente múltiples instrucciones por ciclo
- ¿Qué entendemos por **procesador escalar**?
 - No puede ejecutar más de una instrucción en, al menos, una de sus etapas \rightarrow Su CPI máximo ideal es 1
- **Superescalar**: No existe la anterior restricción.
- ¿Tenemos recursos?
 - Más transistores disponibles (Ley de Moore)
 - Técnicas para resolver las dependencias de datos (planificación)
 - Técnicas para resolver las dependencias de control (especulación)

□ Alternativas

□ Procesador Superescalar con planificación estática:

- En general, chequea riesgos en etapa de ID: Capacidad para chequear conflictos de varias instrucciones a la vez.
- Si una instrucción presenta conflictos → bloquea instrucción conflictiva y siguientes
- Implementaciones típicas: 2 instrucciones a la vez (entera, salto, memoria + punto flotante)
- Caso particular: Very Long Instruction Word (VLIW)
 - La etapa ID no chequea riesgos
 - El compilador conoce el HW disponible en la etapa de ejecución y sus latencias de uso → Forma "paquetes" agrupando varias instrucciones independientes que se descodifican y se lanzan a ejecución simultáneamente.
 - El número de instrucciones en cada paquete (p. ej. 3) es fijo. Si el compilador no encuentra suficientes instrucciones libres de riesgos para formar un paquete, rellena con NOP.
 - Puede adaptarse bien al ámbito de los sistemas empotrados.
 - Ref para estudiantes interesados: Hennesy & Patterson, 5th ed. 2012, Sección 3.7 y Apéndice H.

□ Procesador Superescalar con planificación dinámica

- Analiza dependencias de varias instrucciones a la vez.
 - Si conflictos → manda a colas de espera.
 - Puede mandar a las unidades funcionales (UFs) un número variable de instrucciones por ciclo (en función de la disponibilidad de operandos y de UFs)
- Decodificación en orden → Ejecución en desorden → Finalización en orden

Procesadores VLIW

- Caso particular de los superescalares: procesadores VLIW.
 - Procesadores simples que se apoyan en el compilador para ejecutar instrucciones de manera eficiente.
 - No comprueban la existencia de riesgos entre instrucciones.
 - El compilador conoce qué unidades funcionales están libres y las latencias de uso de los riesgos.
 - El compilador forma “packs” de instrucciones independientes que se lanzan a ejecutar simultáneamente.

Clásico

1 R1 * R2 → R3
2 R3 + R4 → R5
3 R1 - R4 → R6

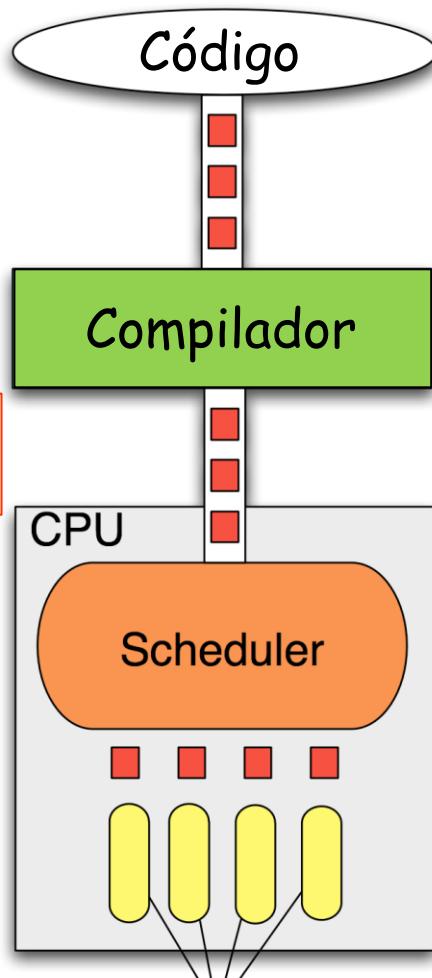
VLIW

1 R1 * R2 → R3 2 - R1 - R4 → R6
 - R3 + R4 → R5 -

- El número de instrucciones por “pack” (por ejemplo, 3) es fijo.

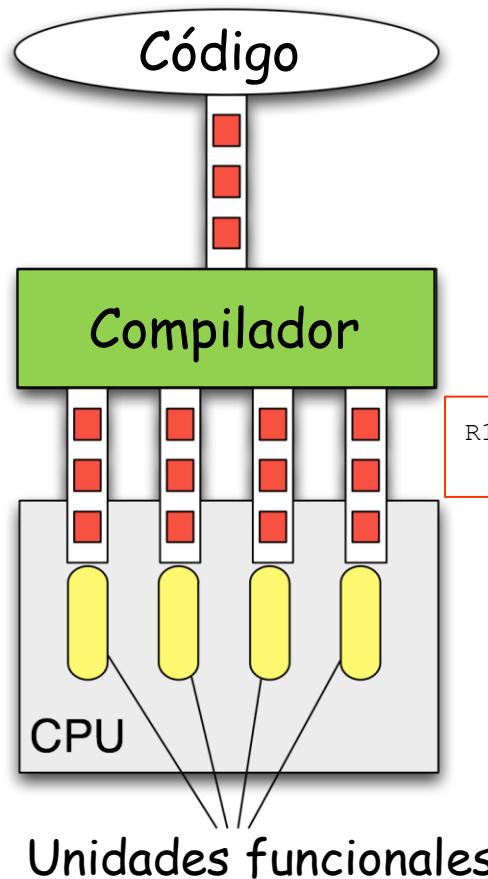
- Ventajas e inconvenientes de VLIW respecto a procesadores fuera de orden:
 - Más simples y más baratos.
 - Menor consumo de energía.
 - Los compiladores deben ser muy complejos.
 - Cualquier cambio en el HW implica falta de compatibilidad con un código ya compilado.

Procesadores VLIW



Unidades funcionales

CPU clásica



Unidades funcionales

CPU VLIW

Procesadores VLIW

- Ejemplo de una arquitectura basada en VLIW: EPIC (Explicitly Parallel Instruction Computing) de Intel.
 - Itanium e Itanium2 (lanzados en 2001).



- Fracasaron y fueron descontinuados en 2020.

<https://www.eejournal.com/article/intel-pulls-the-plug-on-itanium/>

Donald Knuth (professor emérito en Stanford University): "*the Itanium approach [was] supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write.*"

<https://www.informit.com/articles/article.aspx?p=1193856>

Procesadores VLIW

- Las arquitecturas VLIW han fallado como procesadores de propósito general. Muchos expertos las consideran ya obsoletas.

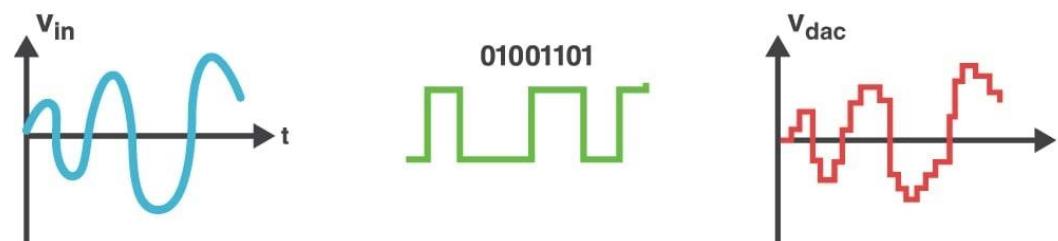
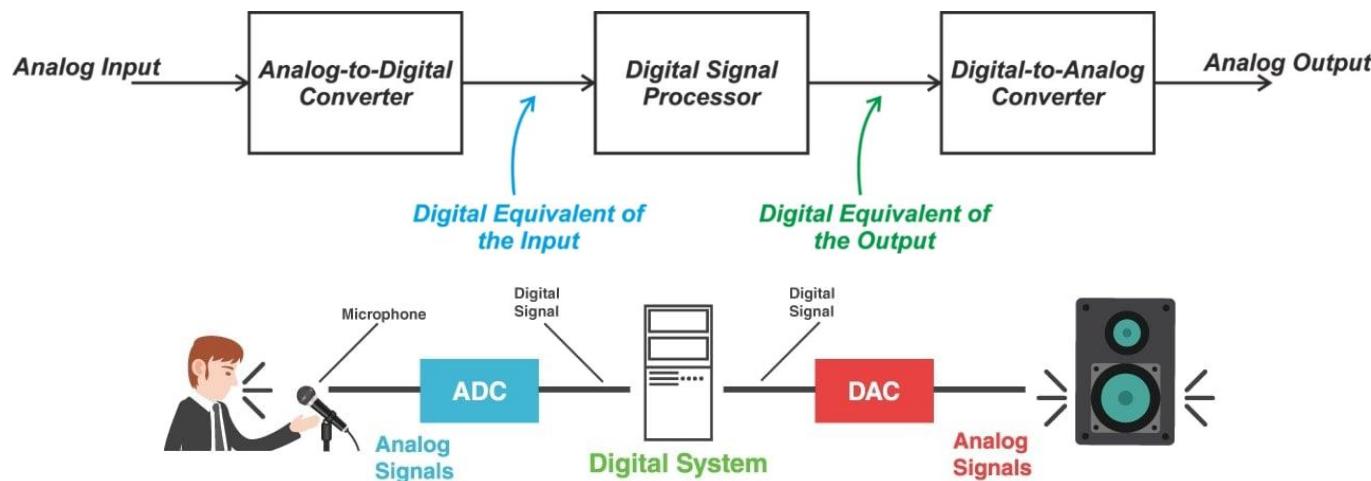
Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop



1. Demasiados espacios vacíos en el código → Desperdicio de memoria.
2. Los saltos son difíciles de predecir → Complica la planificación y empaquetar instrucciones paralelas en "palabras" de instrucciones VLIW.
3. Los fallos de cache impredecibles ralentizan la ejecución y hacen que el tiempo de ejecución sea variable.
4. Un buen compilador para máquinas VLIW resulta ser demasiado difícil de diseñar.

Procesadores VLIW

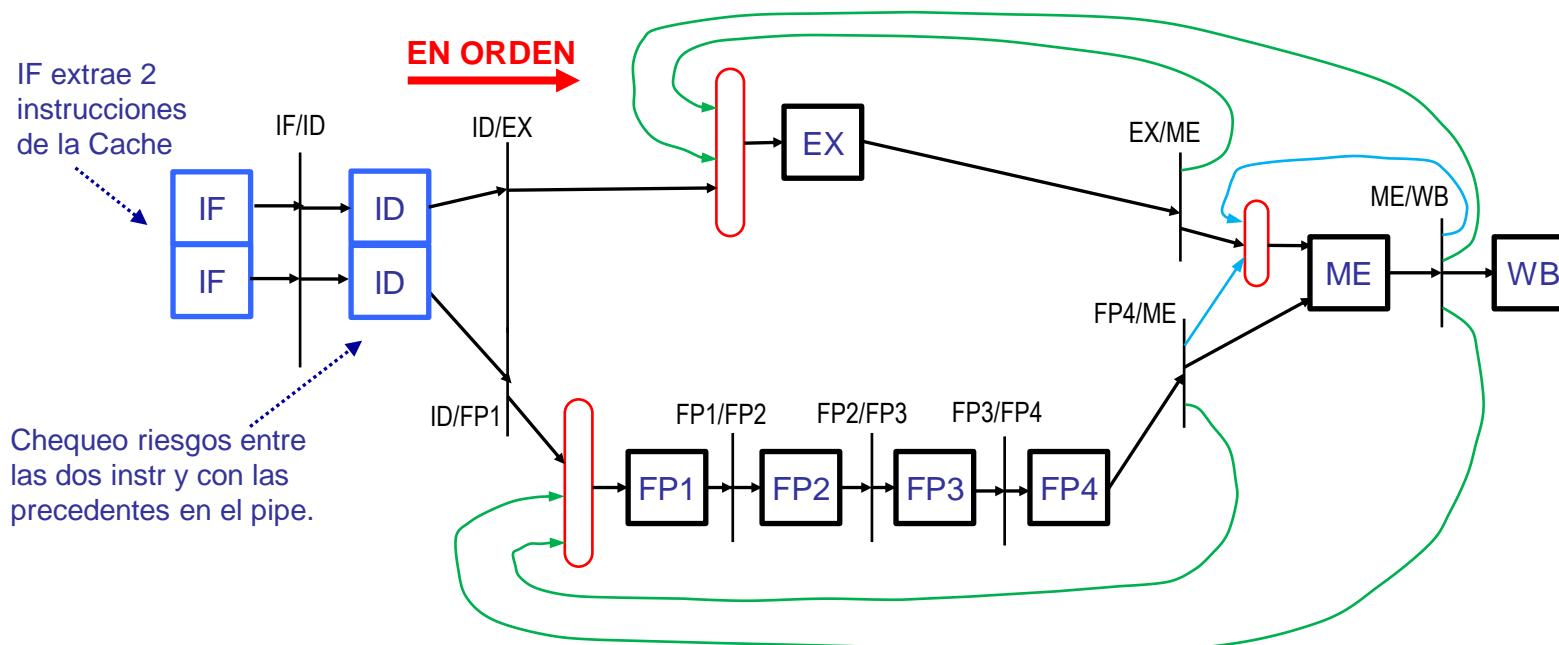
- ❑ Sin embargo, el concepto de VLIW encaja bien en algunas tareas específicas:
 - Código predecible.
 - Pocos saltos (o ninguno).
- ❑ Ejemplo: Digital Signal Processors (DSPs).



<https://www.allaboutcircuits.com/technical-articles/an-introduction-to-digital-signal-processing/>

Superescalar con planificación estática

- ❑ Procesador Superescalar con planificación estática
 - Ejemplo: Completando las etapas que solamente admitían una instrucción
 - Las etapas IF e ID pueden procesar dos instrucciones por ciclo
 - Si una es de FP y la otra de tipo ENT pueden enviarse a ejecución en un mismo ciclo: **Riesgos?**
 - Dependencias entre ENT y FP: instrucciones de LD/ST. Efecto de las dependencias provocadas por Load!
 - Incremento de los problemas provocados por los saltos



- En la etapa ME solamente el camino superior puede acceder realmente a la memoria. El otro la puentea.
- En la etapa WB puede finalizar una instrucción de FP y una ENT si escriben en bancos de registros separados.

Superescalar con planificación estática

□ Ejemplo: Suma vector más escalar

- Desarrollo para ejecución superescalar: se desarrolla una iteración más.
 - 5 iteraciones en 12 ciclos → 2.4 ciclos por iteración
 - $CPI = 12/17 = 0.706$ $IPC = 1/CPI = 1.417$
- El código máquina está compacto en la memoria
 - Instrucciones INT y FP intercaladas. El hw de lanzamiento detecta si puede lanzar una o dos instrucciones.

Loop:	<u>Instrucción entera</u>		<u>Instrucción FP</u>	<u>Ciclo</u>
	L.D	F0,0(R1)		
	L.D	F6,-8(R1)		1
	L.D	F10,-16(R1)		2
	L.D	F14,-24(R1)		3
	L.D	F18,-32(R1)		4
	S.D	F4,0(R1)	ADDD F4,F0,F2	5
	S.D	F8,-8(R1)	ADDD F8,F6,F2	6
	S.D	F12,-16(R1)	ADDD F12,F10,F2	7
	DADDIU	R1,R1,#-40	ADDD F16,F14,F2	8
	S.D	F16,16(R1)	ADDD F20,F18,F2	9
	BNE	R1,R2,Loop		10
	S.D	F20,8(R1)		11
				12

* Recordar: latencias de uso (Tema 2, transp. 13)

Superescalar con planificación estática

➤ Ventajas

- No modifica código. Compatibilidad binaria.
- No riesgos en ejecución: si hay riesgo → bloqueo de lanzamiento

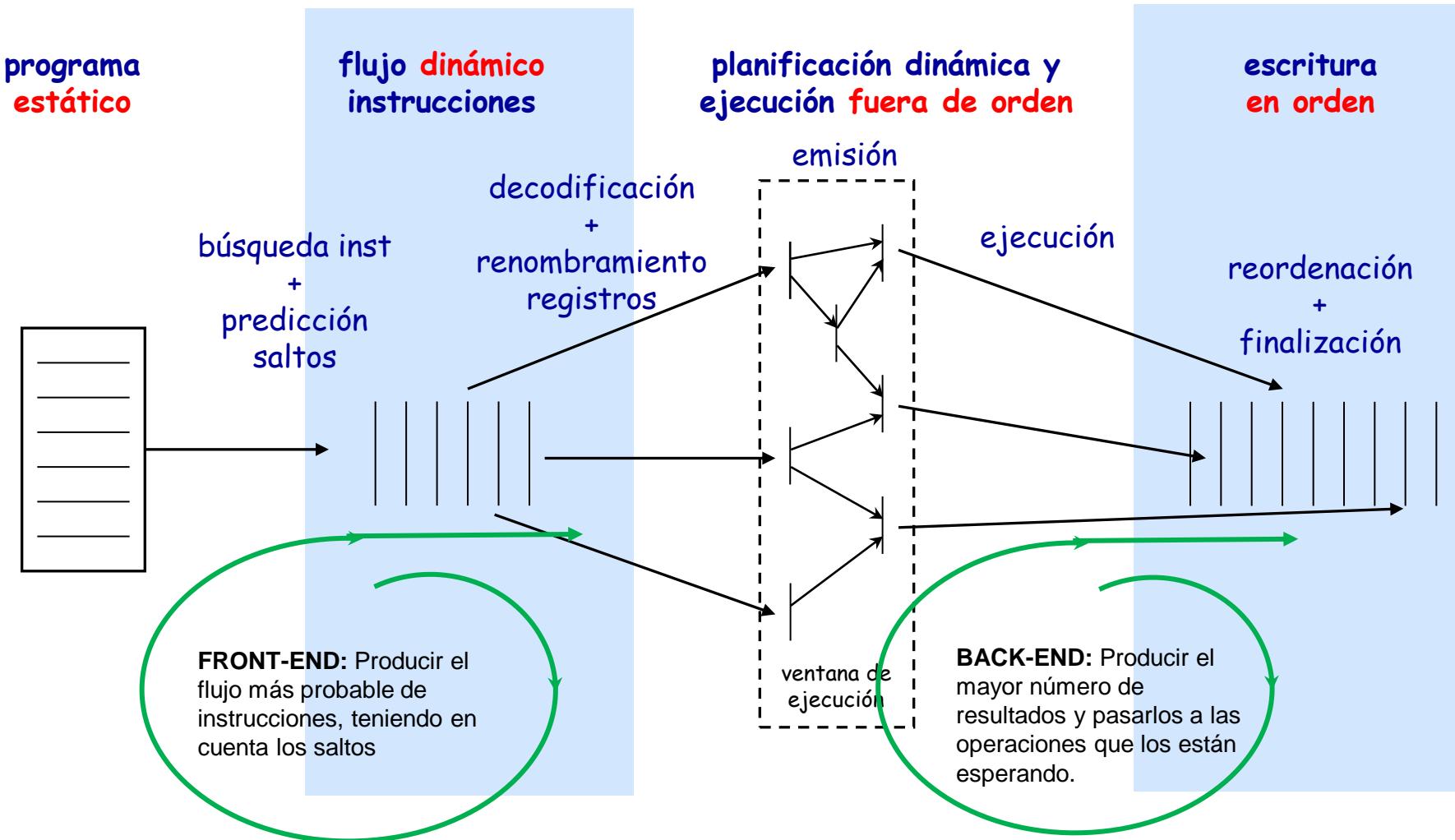
➤ Desventajas

- Mezcla de instrucciones. Solo obtiene CPI de 0.5 en programas con 50 % de FP
- Bloqueos en el lanzamiento (igual que en MIPS básico)
- Planificación fija: No puede adaptarse a cambios en ejecución (p.ej. fallos de cache)
- Los códigos deben replanificarse para cada nueva implementación (razones de eficiencia)

Superescalar con planificación dinámica

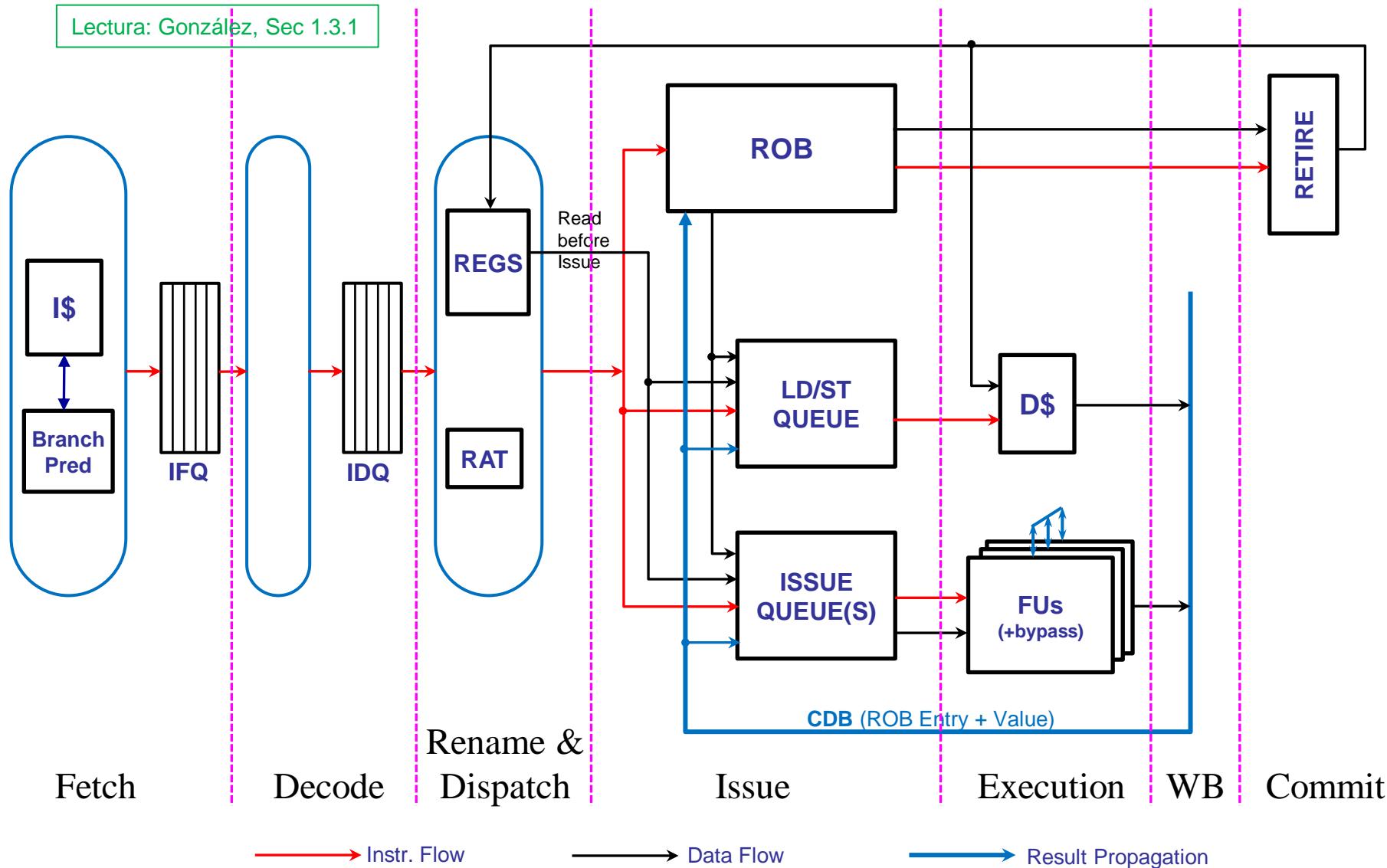
□ Idea Básica

Ejecución fuera de orden. Finalización en orden



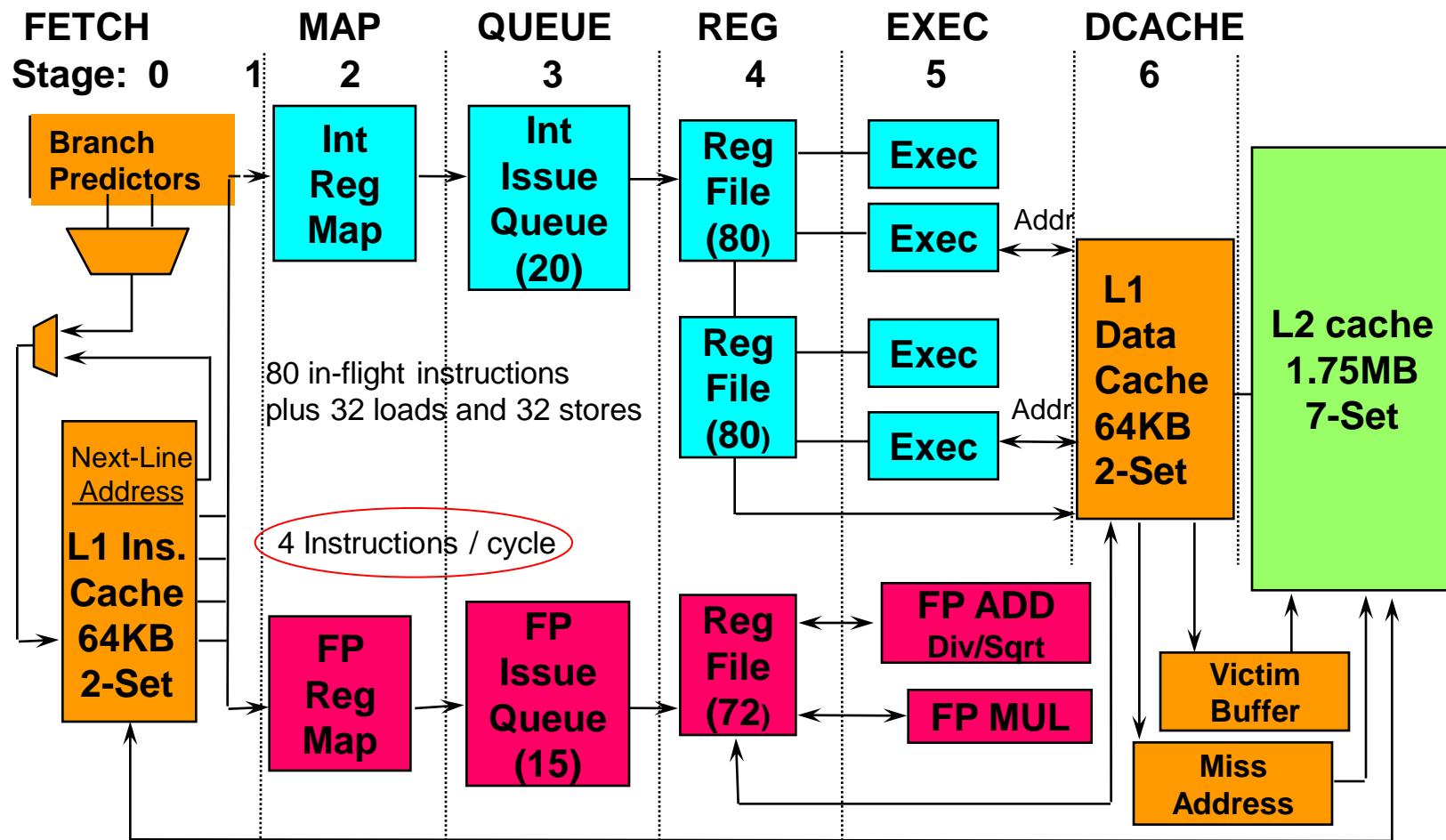
Superescalar con planificación dinámica

- Esquema general de organización en etapas (varias alternativas posibles)



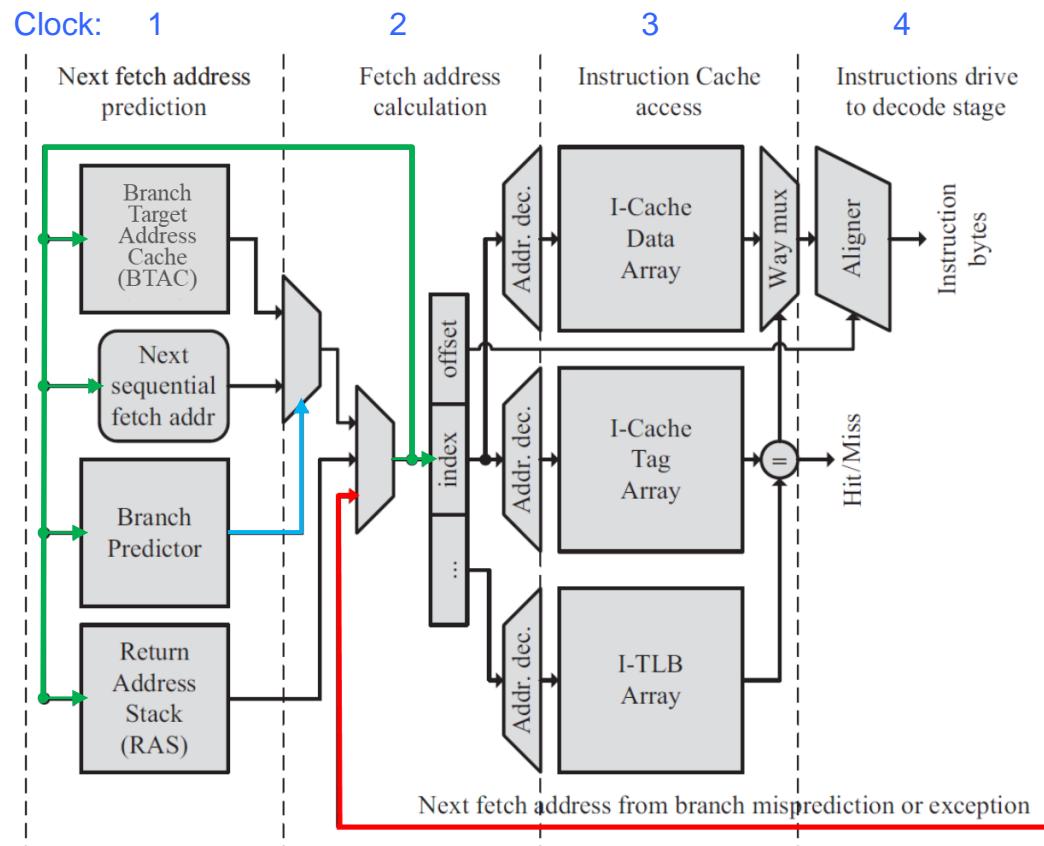
Superescalar con planificación dinámica: Ejemplo

□ EV7 ALPHA 21364 Core (2003)



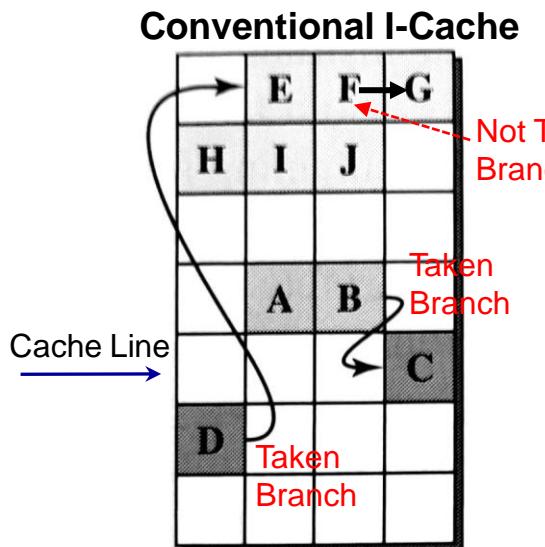
Superescalar con planificación dinámica: Fetch (1)

- El diseño mostrado trabaja en cuatro ciclos para sostener altas frecuencias de reloj
 - ... Pero cada ciclo puede comenzar una nueva búsqueda
- La dirección enviada a la I-Cache (línea verde) se usa también en los predictores
- Lectura de varias instrucciones por ciclo: sencillo cuando las instrucciones están en la misma línea de cache. Efecto de los saltos!

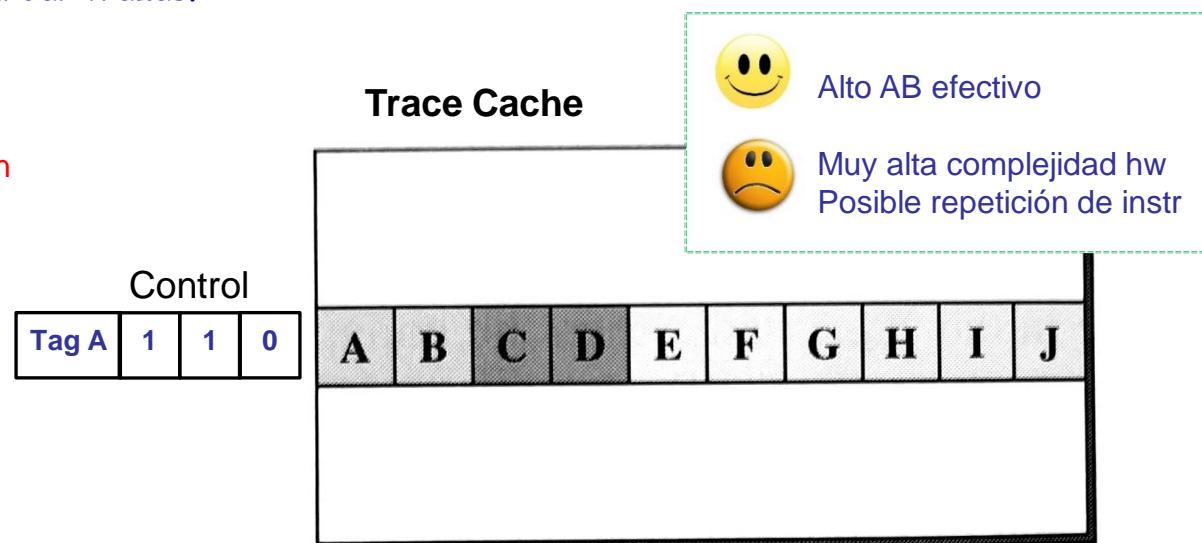


Superescalar con planificación dinámica: Fetch (2)

- Cache convencional. Acceso a grupos de instrucciones consecutivas en memoria (p.ej. cache line) → Un salto tomado (incluso bien predicho) hace inútil el resto del grupo.
- Idea alternativa: **Trace Cache**. Almacenar grupos de instrucciones consecutivas en el flujo dinámico del programa (traza). Usada en Pentium 4.
 - Control (directorio): TAG de la 1^a instr. y comportamiento de las instrucciones de salto (T/NT)
 - El predictor debe predecir varios saltos a la vez
 - CACHE HIT: El TAG de la siguiente dirección a buscar y el comportamiento predicho para los siguientes saltos coincide con Control de una línea de la cache.
 - Problema: Identificar y guardar trazas.



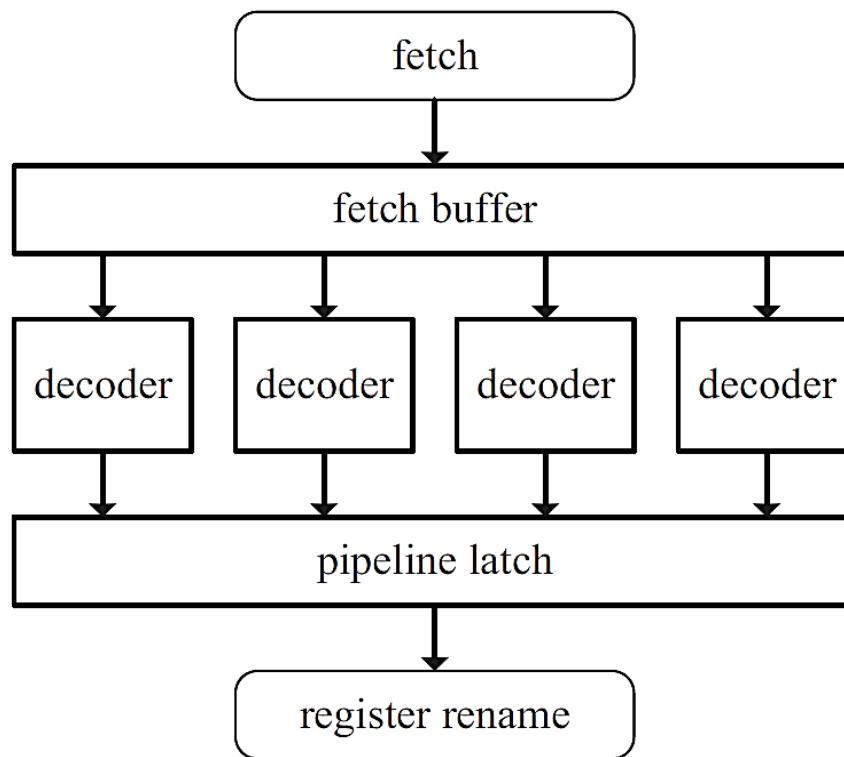
Secuencia de instr A-J (con 2 saltos tomados: B y D) en 5 líneas de cache → 5 ciclos



La misma secuencia de instr almacena en posiciones consecutivas → Acceso posible en 1 ciclo

Superescalar con planificación dinámica: Decode (1)

- Entrada: Cadena de bytes extraídos de la cache
- Salida: Instrucciones separadas + señales control para el pipe.
- Decodificación en RISC: longitud de instrucción fija + pocos formatos → Proceso "relativamente" sencillo
 - Puede consumir un solo ciclo de reloj



Decodifican 4 instrucciones en paralelo, que se pasan a la siguiente etapa

de González: Fig. 4.1

Superescalar con planificación dinámica: Decode (2)

□ Segmentación con repertorios de instrucciones complejos (CISC): Caso x86.

- ¿Como segmentar un repertorio con instrucciones entre 1 y 17 bytes?



- Dónde empieza cada instrucción? Depende del tipo (Opcode), pero...
- El Opcode no está siempre en la misma posición
- El Opcode no siempre tiene la misma anchura
- Instrucciones que mezclan operandos en registros y memoria

□ IDEA! Traducción dinámica de instrucciones:

- El flujo dinámico de instrucciones x86 se traduce en la etapa decode a un flujo de instrucciones tipo RISC



- Primeras implementaciones: Intel P6 (Pentium Pro-Pentium III), AMD K5
- En P6 se traducen las instrucciones originales x86 a "μ-operaciones" de 118 bits
- Cada instrucción original se traduce a una secuencia de 1 a 4 μ-operaciones
- Formato μ-op: código + 3 operandos. Modelo: load/store. Muy descodificadas
- Pero ... La instrucciones más complejas son traducidas por una secuencia almacenada en una memoria ROM (8Kx118) (micropograma)
- **Bucles:** posibilidad de ejecutar μ-ops repetidamente. Sucesivas iteraciones no requerirían traducción dinámica.

Superescalar con planificación dinámica: Decode (3)

□ Pipe de decodificación de Intel Nehalem (2008)

Convierte el flujo de bytes en un flujo ordenado de instrucciones (determinando la longitud de cada una).

En la mayoría de los casos puede trabajar en un solo ciclo. Para algunas instr pue necesitar hasta 6 ciclos.

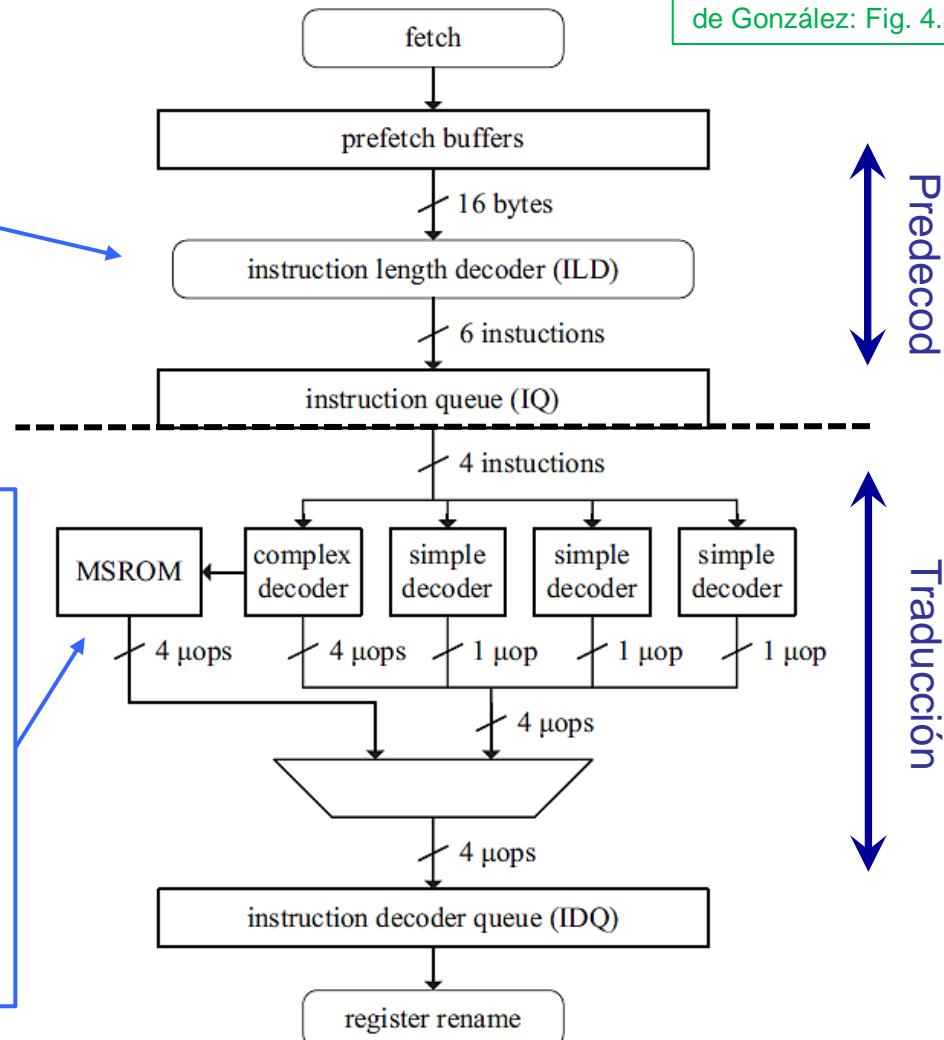
Toma instrucciones de la IQ y las traduce usando:

- Simple decoders (3): Una instr. x86 → Una μ-op
- Complex decoder: Una instr. x86 → **hasta 4 μ-op**
- Secuenciador (MSROM): Para instr. x86 más complejas.

Enfoque adecuado si la mayoría de las instr x86 se traducen en una sola μ-op. Es realista?

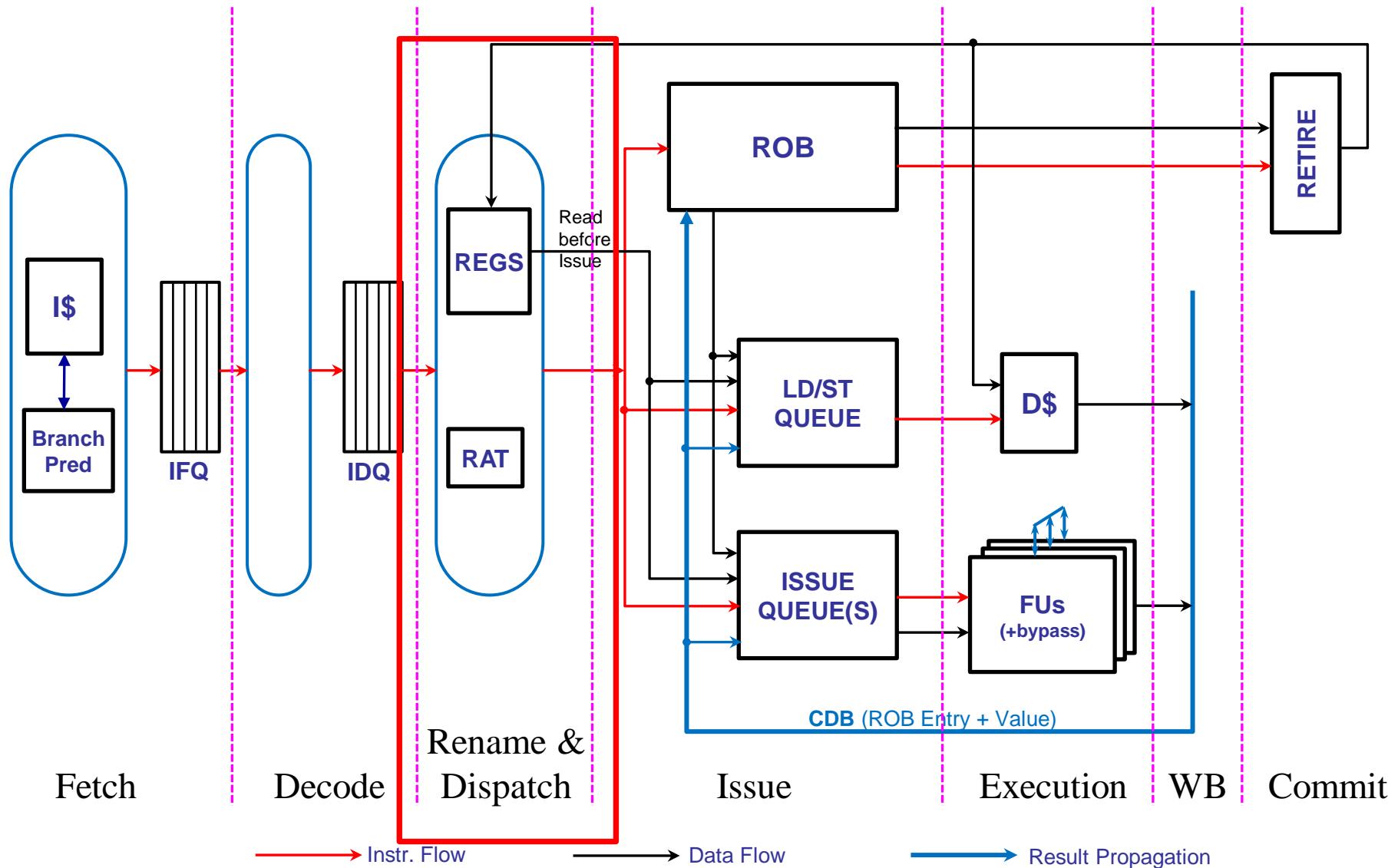
↓
Sí: Relación μ-ops / instr x86 en SPEC,
típicamente <1,5

de González: Fig. 4.3



Superescalar con planificación dinámica

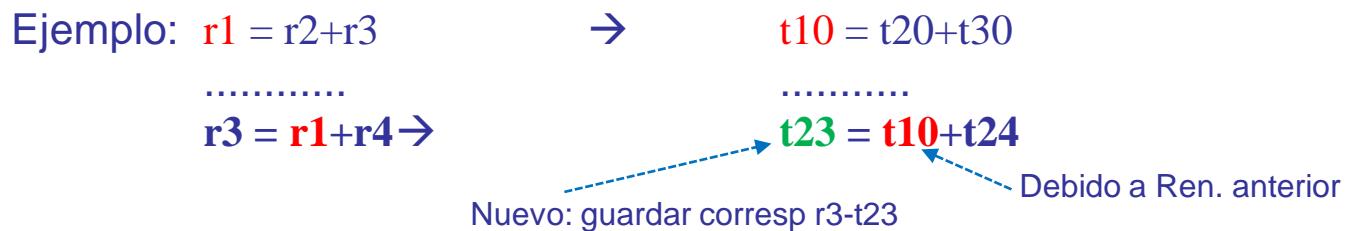
- Esquema general de organización en etapas (varias alternativas posibles)



Superescalar con planificación dinámica: Rename & Dispatch (1)

Rename

- Asignar los elementos de almacenamiento que van a usarse para guardar el resultado (destino) de una instrucción.
 - Además: cambiar los nombres de los registros fuente, teniendo en cuenta los renombramientos de las instr anteriores (tabla)



- ¿Cuándo podemos volver a reusar un registro (p.ej. t10) para guardar el resultado de otra instr?
- Alternativas de diseño
 - Mediante el ROB: entradas del ROB guardan resultados especulados
 - Mediante un Buffer de Renombramiento
 - Los resultados especulados no se guardan en el ROB
 - Buffer adicional para resultados
 - Fichero de registros unificado
 - Solo existen los registros físicos; parte de ellos representan a los registros arquitectónicos

Dispatch

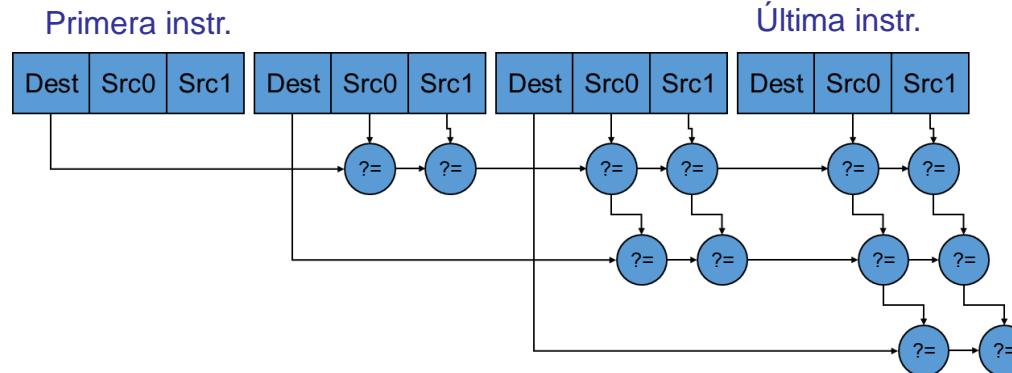
- Reservar entradas en ROB, Issue Queue(s) (y LD/ST Q para las instr de carga/almacenamiento).
 - La Issue Queue se corresponde con el concepto de Estación de Reserva definido por Tomasulo en el IBM 360/91.
 - Aunque en los procesadores modernos esta fase se llama "dispatch", se corresponde con lo que en Tomasulo se denominó "issue".
 - El término "issue" (emisión) se reserva para la fase en que una instrucción se emite hacia la correspondiente UF para comenzar la ejecución.
- Lectura de operandos. Alternativas:
 - Lectura antes de issue (Read before issue). Como Tomasulo
 - Operandos disponibles → copiados a Issue Q. Si no disponibles → copiar identificador del operando (nº de ROB / nº de reg físico).
 - Lectura después de issue (Read after issue)
 - Se pospone la lectura de operandos al momento de la ejecución

Superescalar con planificación dinámica: Rename & Dispatch (3)

□ Renombrado de un grupo de instrucciones en paralelo

- El problema: las tablas de renombrado solamente contienen información sobre las instrucciones **de los grupos** anteriores
- ...pero la 2^a instr del grupo debe tener en cuenta el renombrado de la 1^a. Y la 3^a, el de las dos primeras...
- Alternativas:
 - Renombrado secuencial: Multiplicar (en la medida de lo posible) la frecuencia de reloj de esta etapa por un factor igual al n° de instr. a renombrar
 - Renombrado combinacional: teniendo en cuenta los operandos y destinos de todas las instr del grupo.

Chequeos necesarios para renombrar un grupo de instrucciones

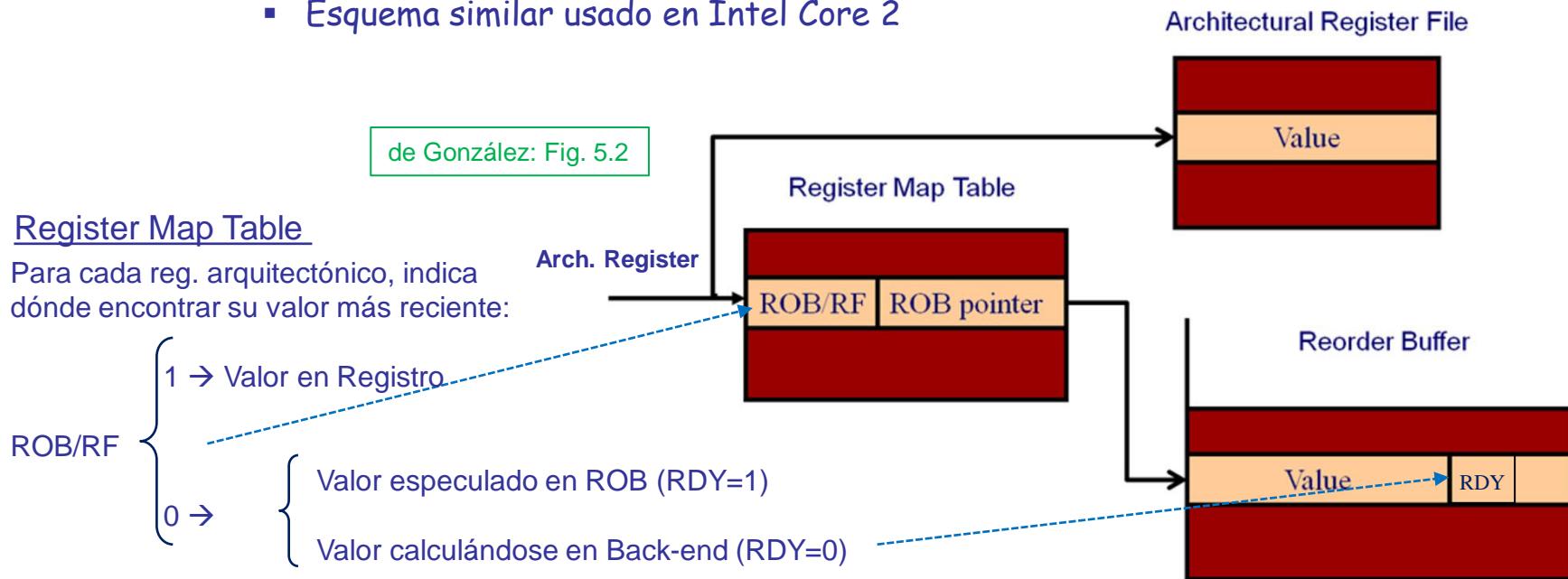


Pensar: ¿Qué ocurre si Src0 de 4^a instr coincide con Dest de 1^a instr. y con Dest de 3^a instr.?

Superescalar con planificación dinámica: Rename & Dispatch (4)

❑ Renombrado mediante el ROB

- Se ajusta en líneas generales a lo estudiado en el Tema 2. ¿Cuál es la correspondencia entre las implementaciones?
 - Esquema similar usado en Intel Core 2



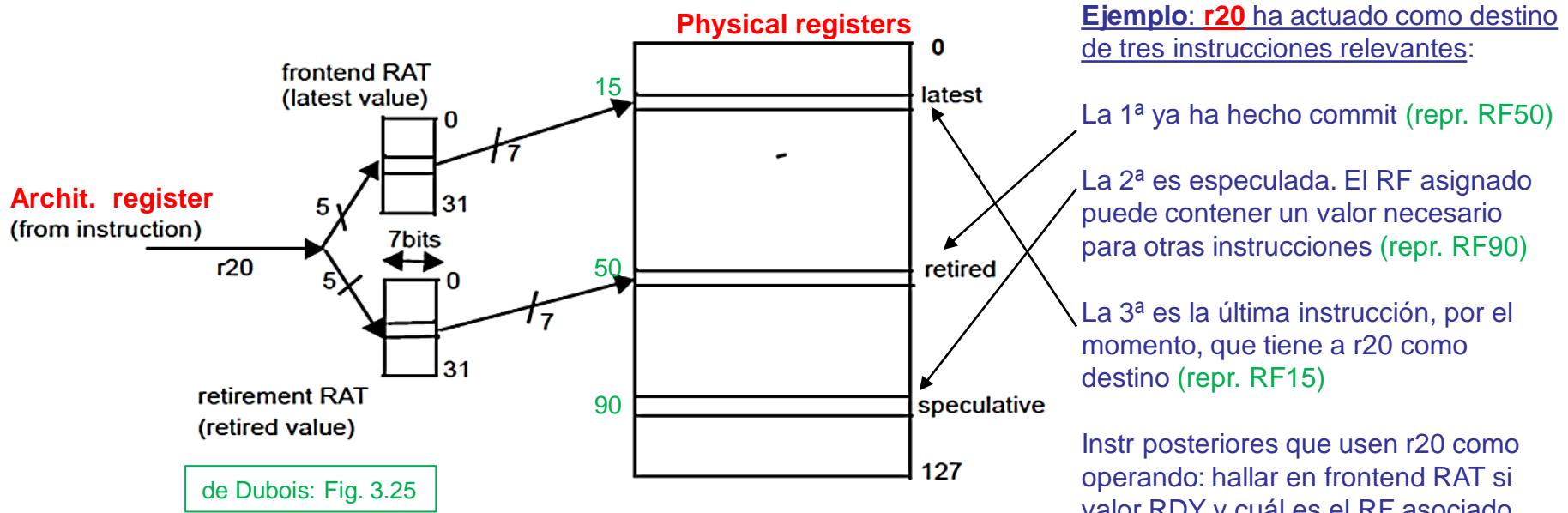
- ## ❑ ALTERNATIVA: Buffer de Renombramiento (esquema usado en IBM POWER3)
- No todas las instrucciones producen un resultado → Desperdicio de espacio en el ROB.
 - Idea: En el ROB se guardan punteros a un conjunto auxiliar de registros (Buffer de Renombramiento) en lugar de valores.
 - Los valores se guardan en el Buffer de Renombramiento
 - Asignar entradas en dispatch, liberar en commit

❑ Fichero de registros unificado

- Los resultados, **especulados o no**, se guardan en un único fichero de registros (**registros físicos, RF**), con un n^o registros $> n^o$ registros arquitectónicos (RA).
- El ROB se limita a mantener el orden de las instr, pero no almacena valores.
 - Ventaja: en la fase commit no hay que copiar resultados del ROB a registros.
 - ... pero cada entrada del ROB guarda el n^o de RF donde se debe almacenar el resultado de la operación y el n^o del correspondiente RA. (Commit)
- Los RA se mapean dinámicamente sobre los RF.
 - En cualquier momento un mismo RA puede estar mapeado sobre varios RF.
- Debe existir un RF encargado de almacenar el último valor asignado no especulativamente (en commit) a cada RA
- Un RF quedará libre (reutilizable) cuándo contenga un valor que ya no vaya a ser utilizado: mantener lista de RF libres ¿Cuándo podemos estar seguros?
 - Si RFi se está usando para contener el valor no especulado de RAj y se ejecuta la fase commit de una instrucción cuyo reg destino es tb RAj → el valor que hay RFi ya no se necesita → RFi puede reutilizarse.

Superescalar con planificación dinámica: Rename & Dispatch (6)

- Fichero de registros unificado. Ejemplo: 32 RA y 128 RF
 - **Frontend RAT**: Para cada RA, indica el RF asignado la última vez que dicho RA actuó como destino de una instr. + Bit RDY
 - **Retirement RAT**: Para cada RA, indica el RF que contiene su último valor no especulado (puede ser el mismo)



- A cada reg destino se le asigna un RF que esté libre (actualizar la frontend RAT).
 - Este nº de RF actuará como TAG (en lugar del nº del ROB asignado)
- Para los registros operando: Encontrar RF asociado a cada operando en frontend RAT.
- Si valor disponible, mandar a Issue Q (lectura antes de issue). Si no, mandar nº de RF (tag)
- Commit: Significa básicamente actualizar la retirement RAT: Qué fila? Qué pasa con el contenido anterior de esa fila?

Superescalar con planificación dinámica: Rename & Dispatch (7)

Ejemplo 1: Supongamos el programa:

Cod1 r20, ---, ---

....
(instrucciones zona A)
....

Cod2 r20, ---, ---

....
(instrucciones zona B)
....

Cod3 r20, ---, ---

....
(instrucciones zona C)
....

Dispatch Cod1: Hallar un RF libre para asignar a r20.
Sup que se elige RF50 → Asignar: Frontend RAT(20)=50

Rename en zona A: Si r20 aparece como operando, la Frontend RAT indicará que r20 está representado por RF50

Dispatch Cod2: Hallar un RF libre para asignar a r20.
Sup que se elige RF90 → Asignar: Frontend RAT(20)=90

Rename en zona B: Si r20 aparece como operando, la Frontend RAT indicará que r20 está representado por RF90

Dispatch Cod3: Hallar un RF libre para asignar a r20.
Sup que se elige RF15 → Asignar: Frontend RAT(20)=15

Rename en zona C: Si r20 aparece como operando, la Frontend RAT indicará que r20 está representado por RF15

- Supongamos que Cod1 hace COMMIT → Asignar: Retirement RAT(20)=50.
 - Todavía puede haber instr en Zona A que precisen RF50 como operando.
 - RF50 representa el valor no especulado de r20
- Supongamos que Cod2 hace COMMIT → todas las instr de Zona A han hecho COMMIT
 - Como Retirement RAT (20) = 50 → RF50 ya no tiene utilidad. Poner RF50 en lista de RF libres.
 - Asignar: Retirement RAT(20) = 90. El valor no especulado de r20 queda representado por RF90

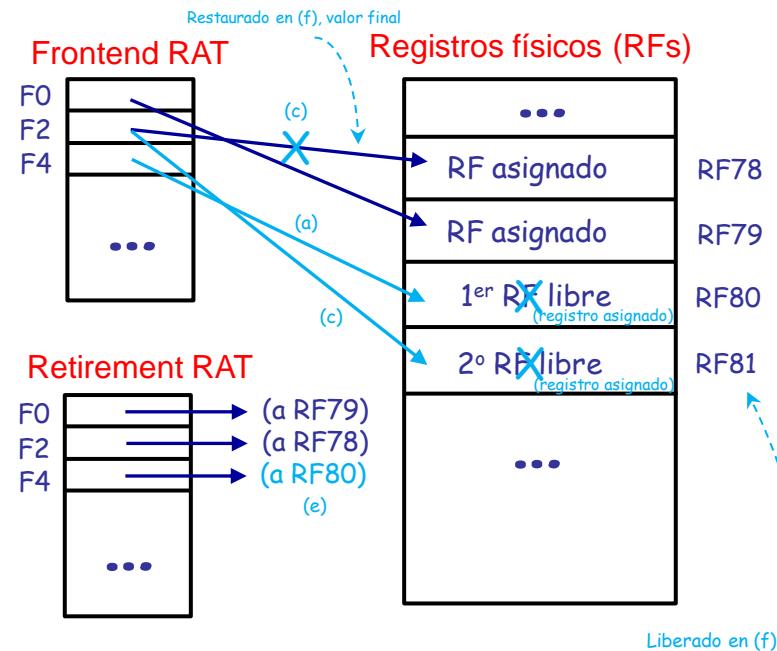
Superescalar con planificación dinámica: Rename & Dispatch (8)

Ejemplo 2:

- Consideremos un procesador superescalar fuera de orden, cuyo renombrado de registros se realiza a través de un fichero de registros unificado. Indicar los cambios que se producen en las Register Allocation Tables (RATs) cuando:

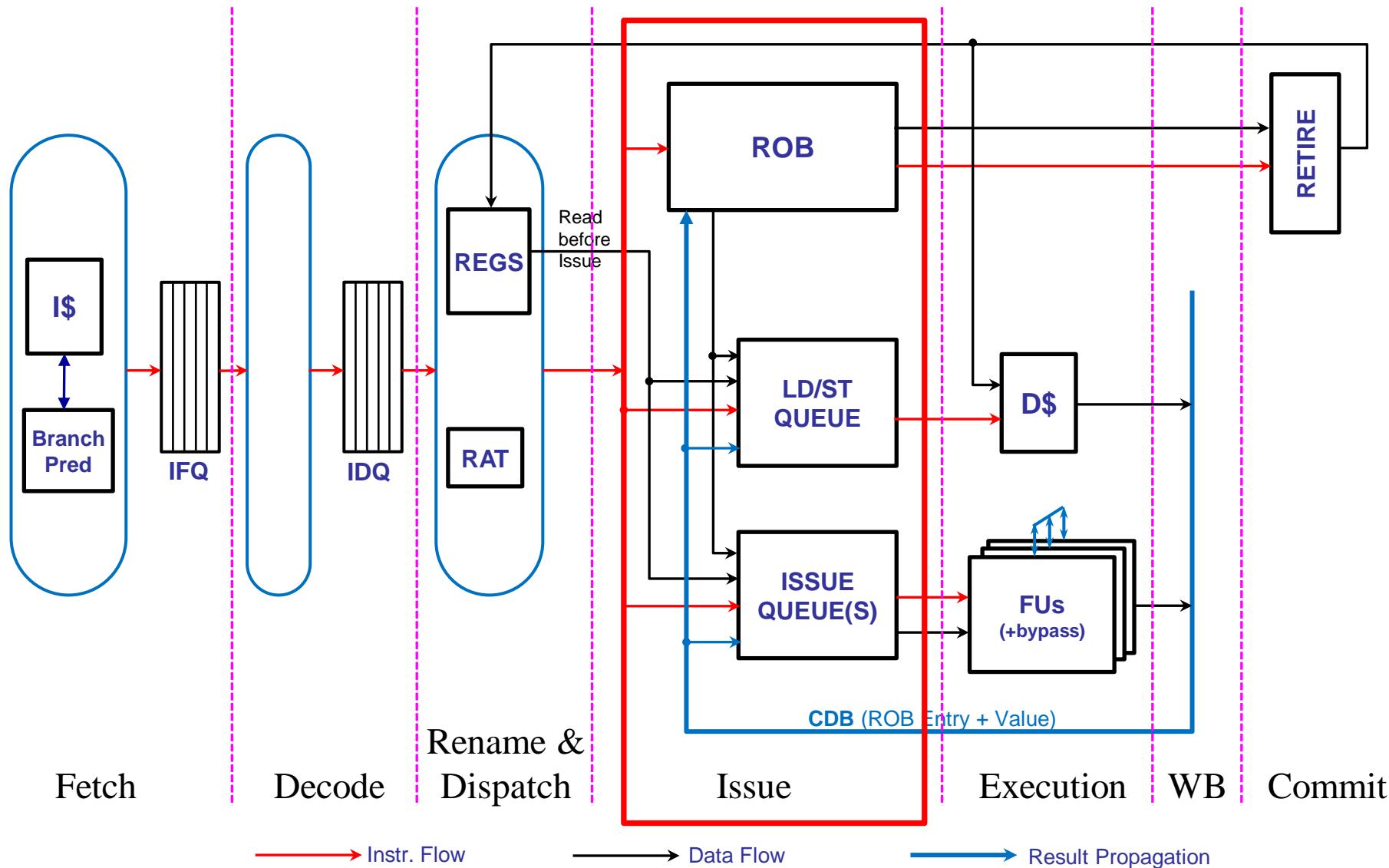
- La instrucción LD F4, 0(R1) se lanza
Frontend RAT(F4) apunta a un nuevo RF (RF80)
- La instrucción BNE #lala, F0, F1 se lanza
Ninguna de las RATs cambia
- La instrucción ADD F2, F1, F4 se lanza
Frontend RAT(F2) apunta a un nuevo RF (RF81)
- La instrucción SD 0(R1), F4 se lanza
Ninguna de las RATs cambia
- La instrucción LD F4, 0(R1) hace "commit"
Retirement RAT(F4) ahora apunta a RF80
- El salto en el paso b) se predice incorrectamente
Frontend RAT(F2) ya no es válido, así que para solucionar esto, toda la Frontend RAT se actualiza como sigue: $\forall x, \text{Frontend_RAT}(Fx) = \text{Retirement_RAT}(Fx)$
En este caso, el efecto es que el paso c) se restaura

Punto de partida: F0 y F2 tienen valores no especulados (entradas de la frontend y retirement RAT apuntando a los mismos RFs)



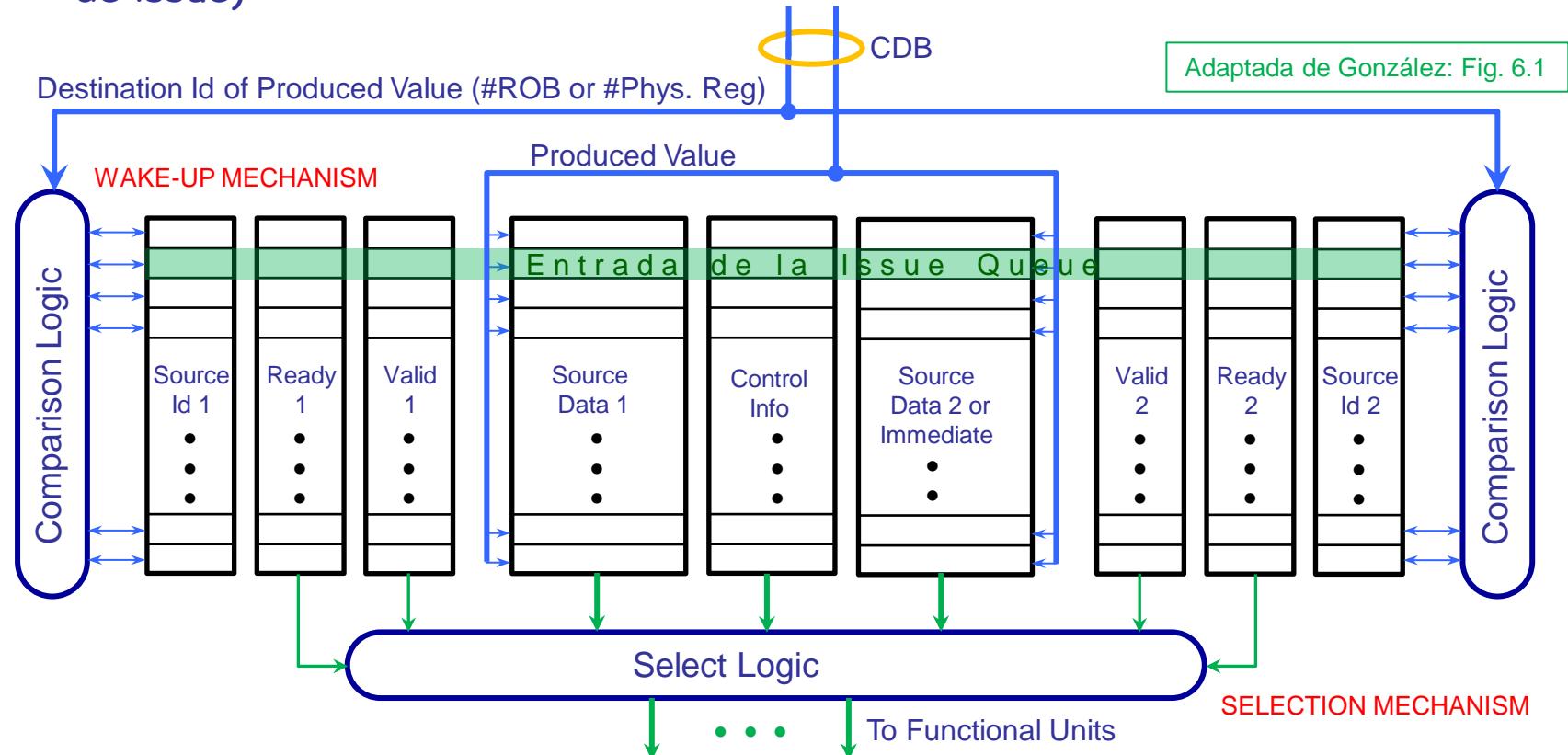
Superescalar con planificación dinámica

- Esquema general de organización en etapas (varias alternativas posibles)



Superescalar con planificación dinámica: Issue (1)

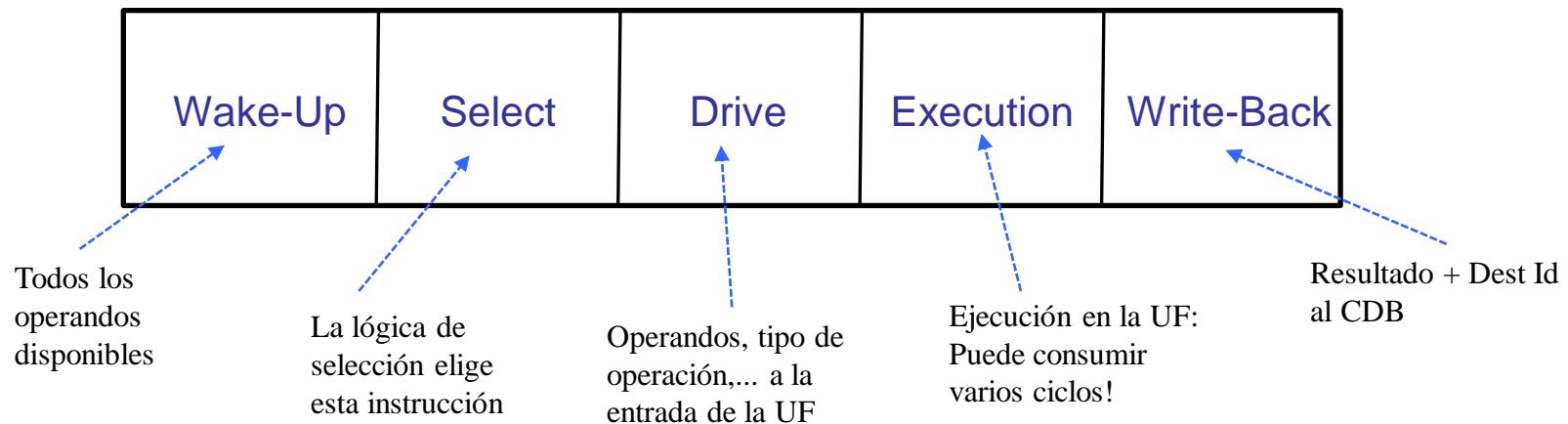
- Esquema general de una Issue Queue (con lectura de operandos antes de issue)



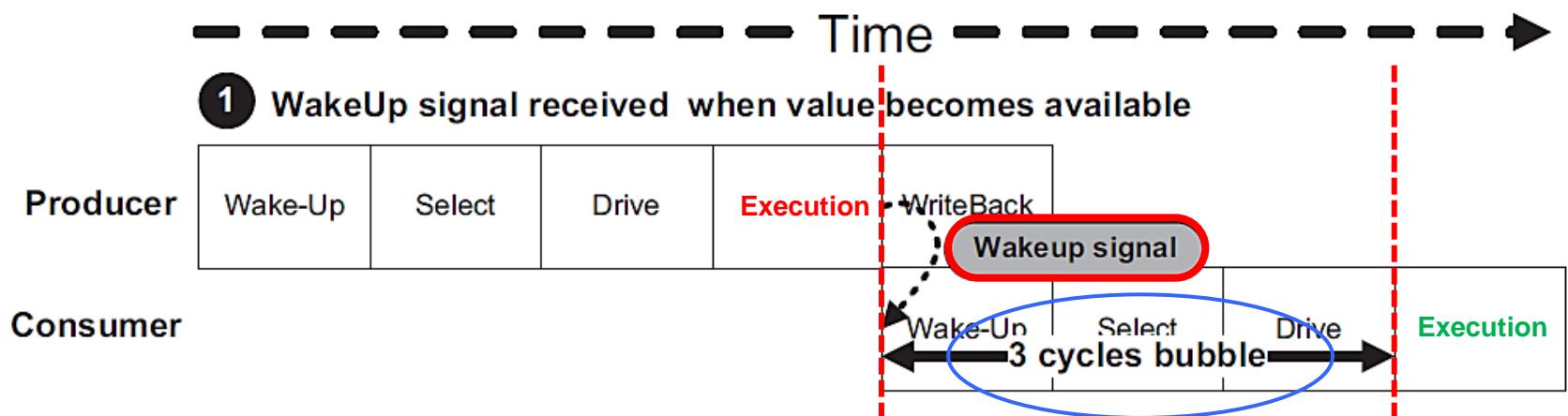
- ❑ **WAKE-UP:** ∀ resultado sobre CDB comparar su Dest Id con Src Id (1 y 2) de todas las entradas de la Issue Q
 - Si coincidencia y Valid bit=1 → Copiar valor y Ready bit \leftarrow 1
 - ❑ Posibilidad de usar más de un CDB → aumento del nº de comparaciones simultáneas
 - ❑ Adaptación a "Lectura después de issue (Read After Issue)". Como?
 - ❑ **SELECTION:** Tomar un subconjunto de las instrucciones que están listas en la Issue Q y llevarlas hasta la entrada de las UF (disponibles y adecuadas al tipo de operación)

Superescalar con planificación dinámica: Issue (2)

□ De wake-up a write-back...



□ El problema: penalización en instr próximas con dep LDE

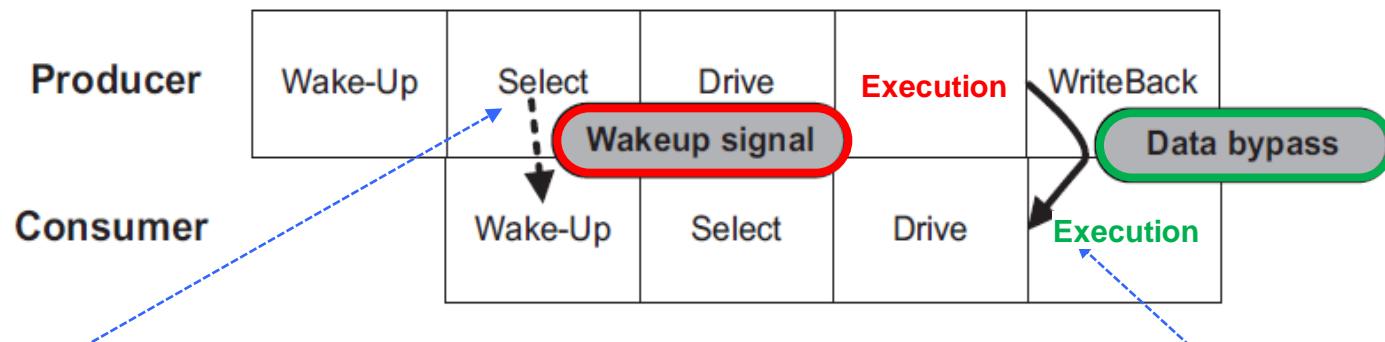


Superescalar con planificación dinámica: Issue (3)

□ Alternativa: wake-up anticipado.

- Las instr aritméticas tienen un tiempo de ejecución conocido de antemano.
- Despertar la instr consumidora varios ciclos antes (dependiendo de la latencia de la productora)

2 WakeUp signal received 3 cycles before becomes available



Al seleccionar la instrucción, activar la señal de wake-up (sup: ejecución en 1 ciclo). Si la ejecución consume N ciclos, generar el wake-up N-1 ciclos más tarde

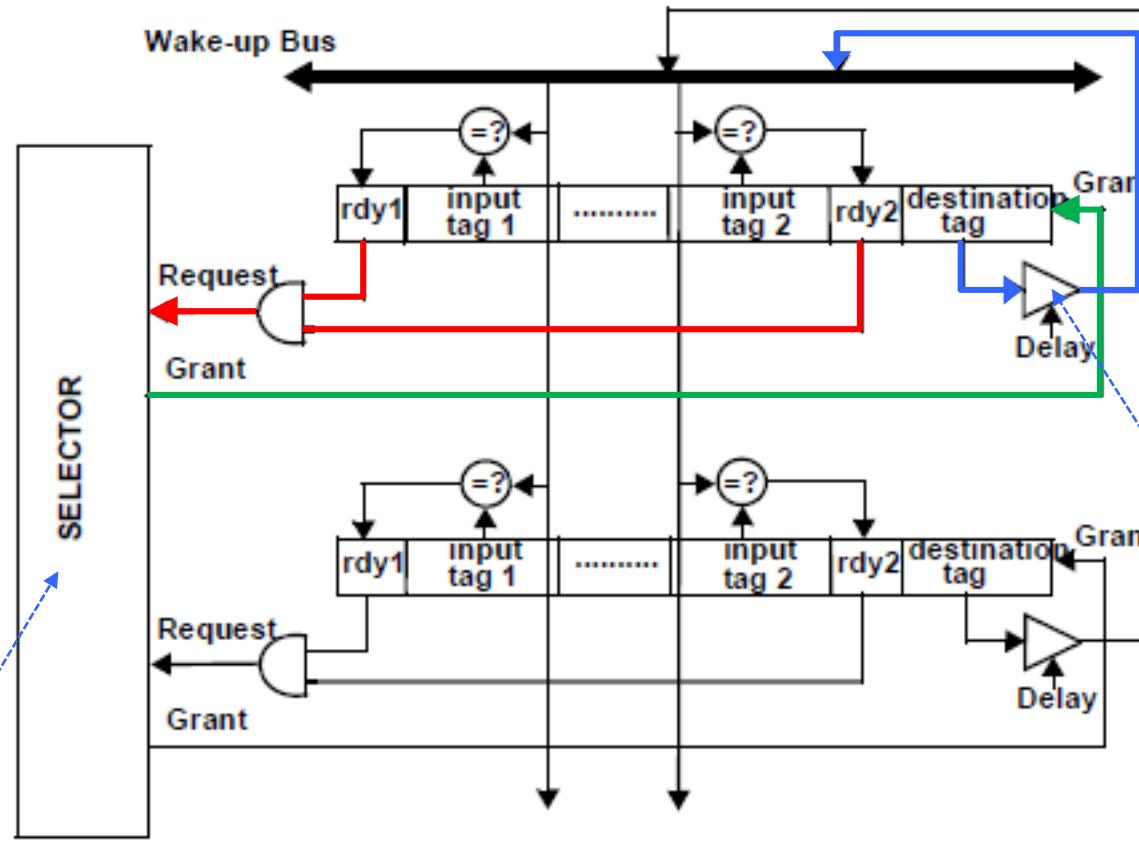
Cuando la instr consumidora llega a la etapa de ejecución puede recibir el operando mediante by-pass desde la salida de la FU productora

- Cero ciclos de penalización (ejecución "back-to-back") para instr que se ejecutan en un ciclo de reloj.
- Debe garantizar el timing previsto (disponibilidad de FU, conflictos de CDB...)
- Problemas cuando el productor es un Load: latencia variable (hit / miss)
 - Necesidad de mecanismo de recuperación

Superescalar con planificación dinámica: Issue (4)

□ Mecanismo de selección con Wake-up anticipado

- Ejemplo: selección de 1 instr en cada ciclo. Se muestran 2 entradas de la Issue Q



de Dubois: Fig. 3.28

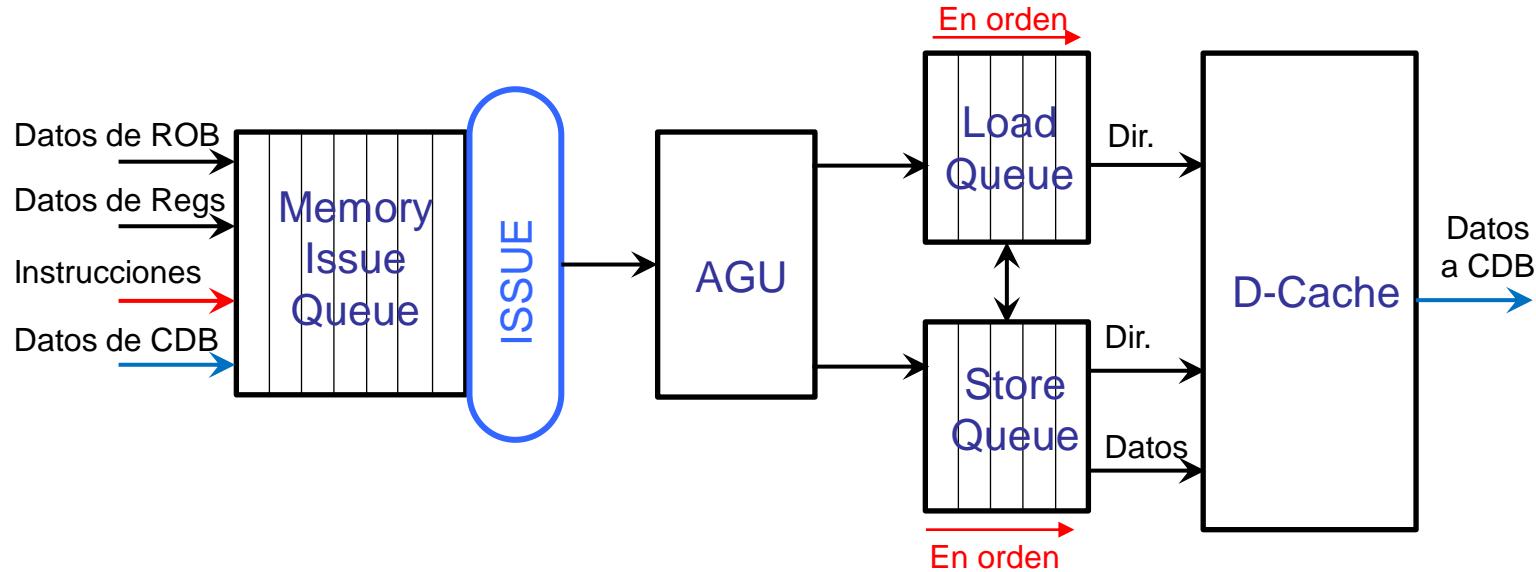
Selecciona una de las instrucciones que han activado la señal “Request”, mediante la activación de “Grant”.

Tras recibir el Grant, la instr seleccionada genera el Wake-up, con el nº de ciclos de retardo (delay) necesario, teniendo en cuenta la latencia de la instr

Superescalar con planificación dinámica: Issue (5)

Tratamiento de Load y Store

Un posible diseño del bloque genérico LD/ST Q mostrado en la tr. 14



- **Memory Issue Q.** Cada entrada contiene una instr de LD/ST.
 - La instr permanece en la cola hasta que captura la información necesaria para calcular la dirección (y para ST tb el dato).
 - Las entradas, como en la Issue Q de las op aritméticas, no tienen por qué estar en orden
- **AGU: Address Generation Unit.** Calcula las dir y las pasa a la entrada correspondiente de la Load o Store Q
- **Store Q.** Mantiene los ST en orden. A cada ST se le asigna una entrada en orden en la fase de dispatch (en ese momento es posible que no esté disponible la dir ni el dato).
- **Load Q.** Mantiene los LD en orden. Cada LD se inserta con un TAG que indica la posición en la Store Q del último ST insertado → Permite conocer cuáles son los ST más antiguos que el LD

Superescalar con planificación dinámica: Issue (6)

Tratamiento de Load y Store

STORE:

- Dispatch:
 - Asignar entrada en ROB en orden
 - Buscar entrada libre en Memory Issue Q (almacenar inst + operandos para calcular la dirección + dato; o Tags que permitan obtenerlos más tarde).
 - Asignar entrada en Store Q **en orden**: Contendrá la dir y el dato del ST cuando estén calculados.
- Issue:
 - Cuando los operandos necesarios están disponibles, se puede mandar la instr a la AGU, y guardar la dir efectiva en la correspondiente entrada de la Store Q.
 - La entrada correspondiente de la Memory Issue Q queda libre.
 - Alternativa: puede ser interesante guardar la dir de un ST en la Store Q, aunque no se disponga todavía del dato. Permite analizar riesgos con los LD más pronto → seguir manteniendo la entrada asignada en la Memory Issue Q.

LOAD:

- Dispatch e issue: Tratamiento similar a los ST en cuanto a la dirección

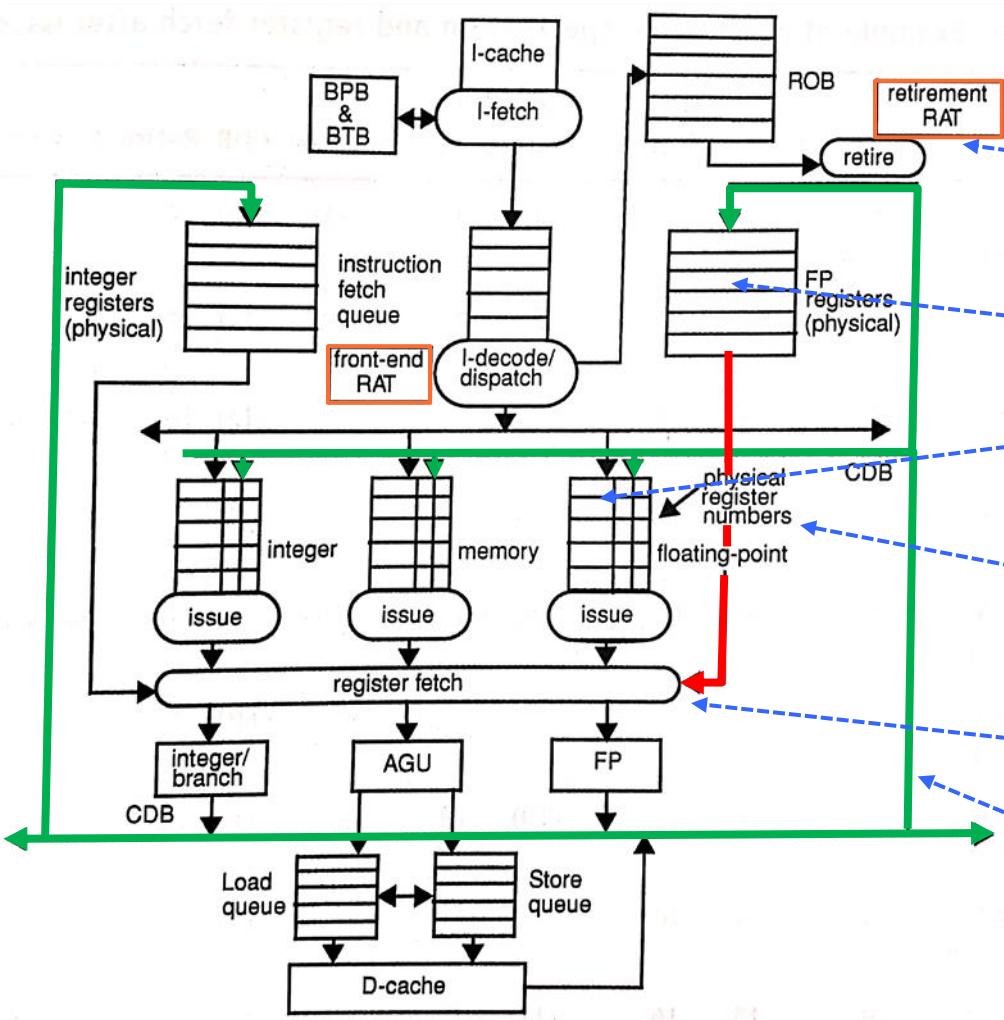
Superescalar con planificación dinámica: Issue (7)

- Desambiguación dinámica de memoria: Comparación de direcciones de LD y ST para evitar riesgos a través de memoria.
 - Un ST en la Store Q solamente **puede ejecutar** la escritura en la D-Cache si tiene su dato+dirección y está en la cabecera del ROB
 - Todos los LD y ST anteriores han hecho commit → las dependencias WAR y WAW no provocan ningún riesgo.
 - Obviamente, el único ST que puede acceder a la D-cache está tb en la cabecera de la Store Q
 - Un LD **puede ejecutar** su lectura en la D-Cache si tiene su dirección y esta dirección no coincide con la de ningún ST más viejo en la Store Q (búsqueda asociativa)
 - Se garantiza la ausencia de riesgos LDE a través de memoria
- Qué ocurre con un LD cuando existen ST más viejos cuya dirección aún no se conoce?
 - Enfoque conservador: Esperar a conocer esas direcciones
 - Enfoque optimista: **Load especulativo**
 - Fundamento: es raro que un LD dependa de un ST próximo → Apostar a que no va haber coincidencia de direcciones (no riesgo LDE) (Guardar dirección del LD)
 - Mecanismo de recuperación: Si cuando se conoce la dirección de un ST hay coincidencia con un LD ya ejecutado, repetir el LD y las instrucciones siguientes (similar a un salto mal predicho).
 - Mecanismo de prevención: Mantener en Fetch una tabla de predicción de Load dependientes, indexada por el PC, con 1 bit por fila (similar a un predictor de saltos)
 - Load mal especulado → poner a 1 el bit correspondiente en la tabla
 - Sucesivas ejecuciones del mismo Load → bit a 1 → Tratamiento conservador
 - Borrar periódicamente el contenido de la tabla

Superescalar con planificación dinámica: Issue (8)

□ Lectura de operandos después de Issue (visión general)

- Implementado en la mayor parte de los procesadores actuales



Hacer Commit: simplemente actualizar la retirement RAT

Suele combinarse con un fichero de registros unificado

Las Issue Q no almacenan operandos; solo información de si están disponibles en los registros (bit Ready).

La identificación de operandos fuente (Source Id 1 y 2) no se hace usando N° de ROB, sino N° de Reg Fisico.

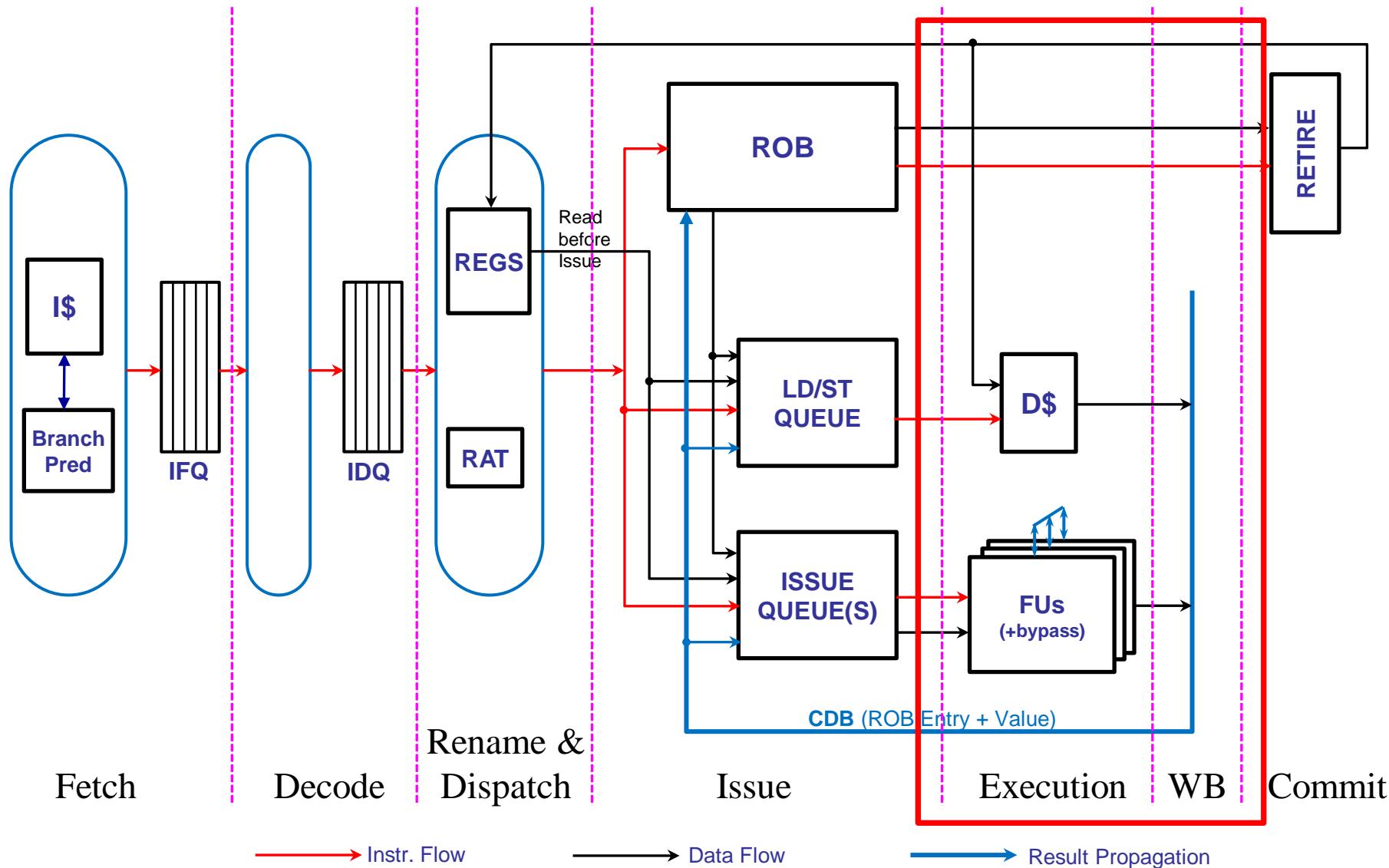
Cuando la instrucción se emite a las UFs se hace la lectura de registros

Los resultados del CBD van acompañados del n° de registro destino → se escriben directamente en registros, no en ROB

de Dubois: Fig. 3.26

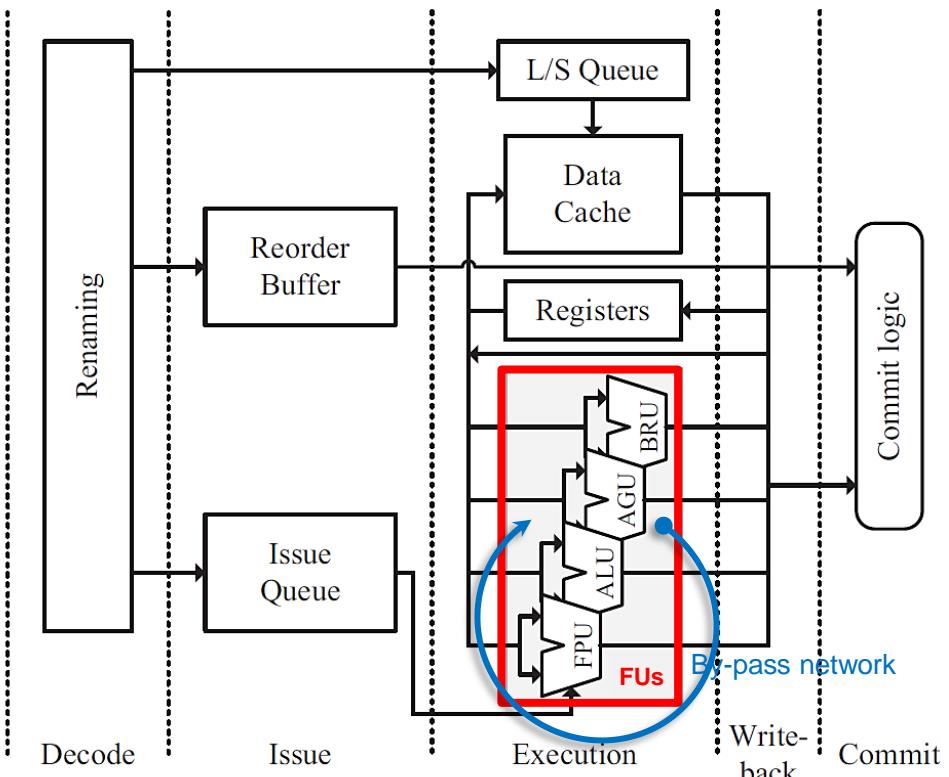
Superescalar con planificación dinámica

- Esquema general de organización en etapas (varias alternativas posibles)



Superescalar con planificación dinámica: Execute&WB (1)

Esquema general de la Unidades Funcionales en su contexto

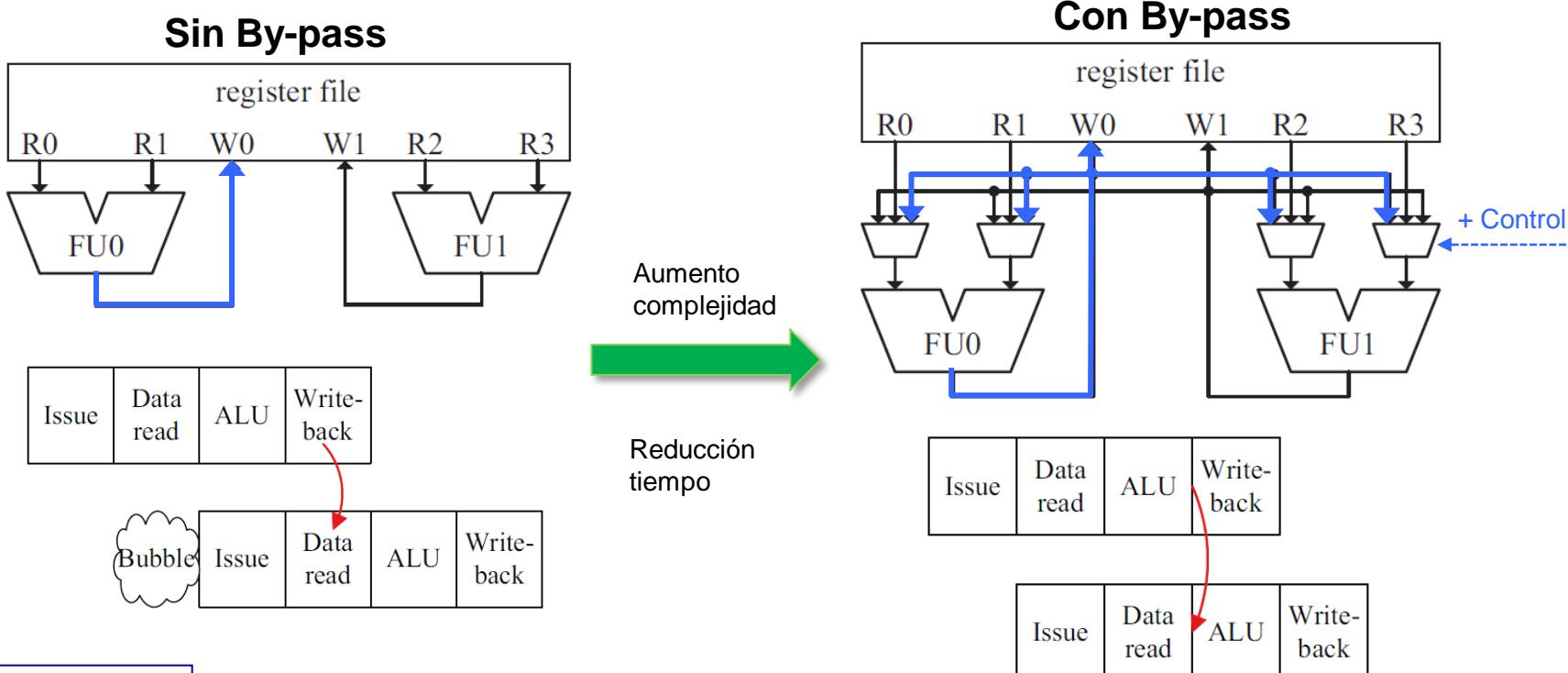


Se asume un fichero de registros unificado

- Las FUs operan sobre los datos que se hacen llegar a sus entradas, junto con la operación a realizar.
- Cuando un resultado está listo (uno o varios ciclos de reloj más tarde) se envía a los elementos de almacenamiento previstos en la instrucción (Write-back) y se realiza el "wake-up" de instrucciones a la espera del resultado.
- El resultado se puede mandar también directamente a las entradas de las FUs que los van a necesitar en el ciclo siguiente, mediante un mecanismo combinacional de anticipación de operandos (by-pass).
 - Concepto similar al estudiado en un procesador segmentado en orden, aunque aumenta la complejidad.

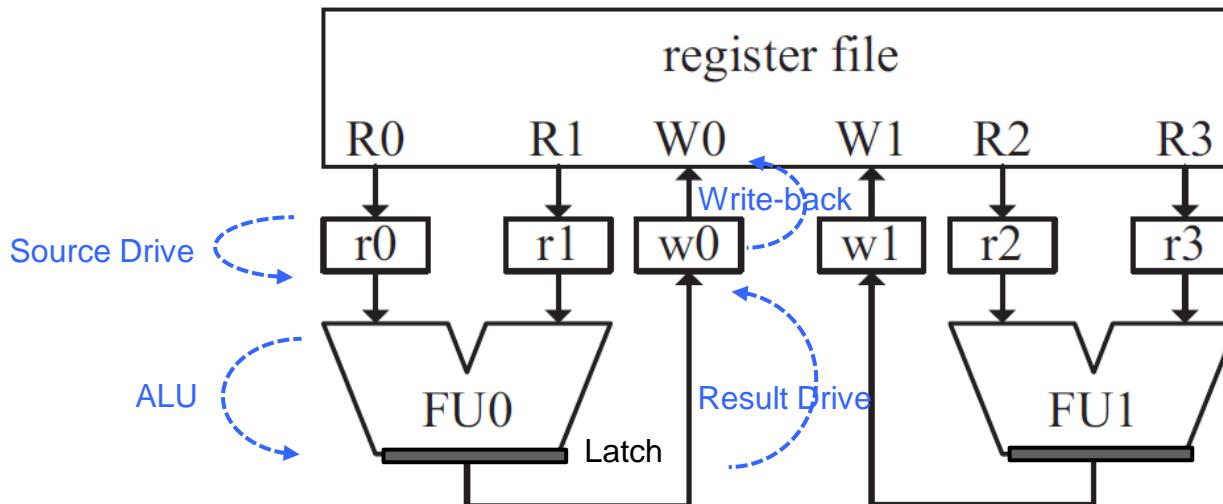
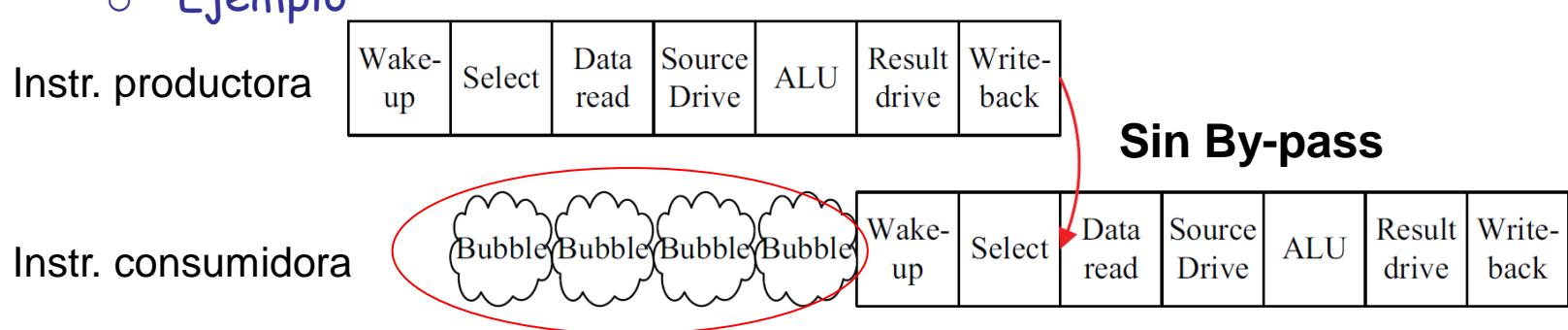
Superescalar con planificación dinámica: Execute&WB (2)

- By-pass: compromiso entre la complejidad combinacional (puede empeorar ciclo de reloj, consumo) y los ciclos de espera entre instrucciones dependientes (mejora IPC).
 - La mayoría de los procesadores actuales usan algún mecanismo de by-pass
 - Hay excepciones importantes (p.ej. IBM POWER5, para disminuir complejidad y aumentar frecuencia de reloj)
- By-pass en un pipeline sencillo.
 - 2 FUs y RF con 4 puertos de lectura / 2 de escritura



Superescalar con planificación dinámica: Execute&WB (3)

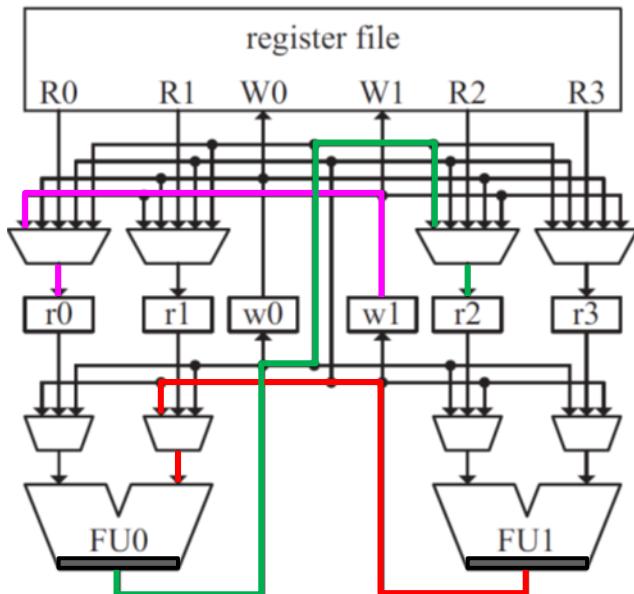
- By-pass en procesadores con alto grado de segmentación. Camino entre RF y FUs segmentado.
 - Más complejidad
 - Sin by-pass: mayor retraso de operaciones dependientes próximas
 - Ejemplo



Superescalar con planificación dinámica: Execute&WB (4)

□ Ejemplo (cont.): red de by-pass con 2 FUs.

- Complejidad ↑↑
 - Alternativa. Clustering: By-pass posible solo entre grupos de FUs
- Paso del resultado de I1 a I2, I3 e I4



Ejemplos de paso de resultados:

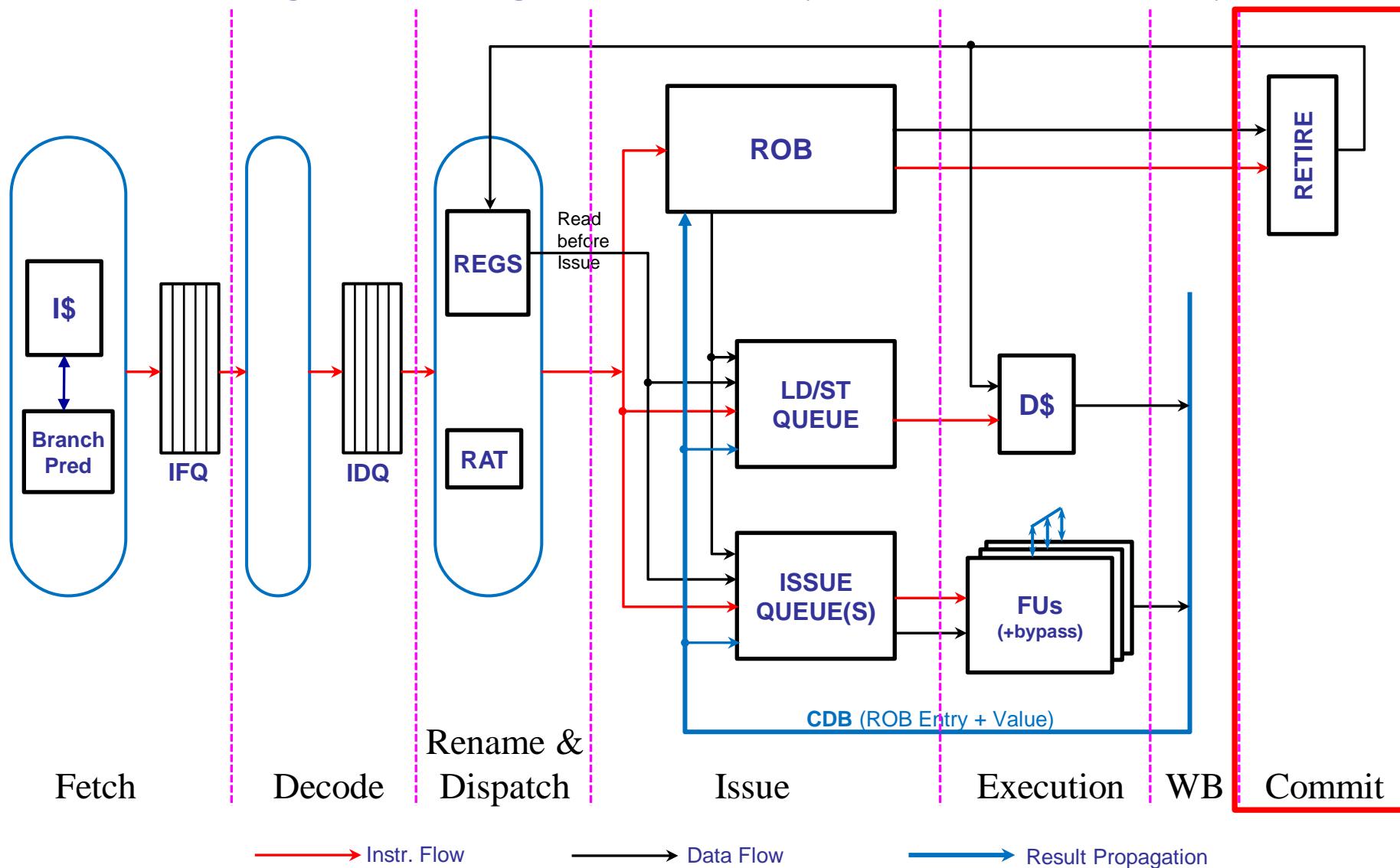
- De I1 a I2 (FU1 → FU0)
- De I1 a I3 (FU0 → r2)
- De I1 a I4 (w1 → r0)

Con By-pass

I1	Wake-up	Select	Data read	Source Drive	ALU	Result drive	Write-back
I2	Wake-up	Select	Data read	Source Drive	ALU	Result drive	Write-back
I3	Wake-up	Select	Data read	Source Drive	ALU	Result drive	Write-back
I4	Wake-up	Select	Data read	Source Drive	ALU	Result drive	Write-back

Superescalar con planificación dinámica

- Esquema general de organización en etapas (varias alternativas posibles)



Superescalar con planificación dinámica: Commit (1)

- ❑ Estado arquitectónico: formado por el contenido de la memoria + contenido de los registros utilizables para codificar instrucciones en LM (es decir, registros arquitectónicos o visibles al programador).
- ❑ Garantiza que el estado arquitectónico, desde el punto de vista de un programador en LM, se actualiza en el mismo orden que en un procesador totalmente secuencial.
 - Las instrucciones hacen la fase commit (es decir, se retiran) en el mismo orden en que se ejecutan en el flujo dinámico del programa (recordar que en dispatch se insertan en el ROB en dicho orden).
 - Una instrucción I se retira cuando está en la cabecera del ROB: Actualiza el estado arquitectónico del procesador. Ejemplo: instr ADD.D F2,F4,F6
 - Con valores en el ROB: Copiar el valor que hay en cabecera del ROB en F2. El resultado del ADD.D se escribe 2 veces: Una en ROB (en WB), otra en F2 (en commit).
 - Con Fichero de Registros Unificado: Sup que en dispatch el Reg. Arq. F2 quedó representado por RF40 → El contenido de la fila 2 de la Retirement RAT pasa a ser 40. El resultado del ADD.D sólo se ha escrito una vez: en RF40
 - Las instrucciones más antiguas que I ya se han retirado → no pueden activar excepciones: Se han ejecutado completamente
 - Las instrucciones más modernas que I están produciendo / han producido resultados especulativos → no modifican el estado arquitectónico.
- ❑ Tratamiento de saltos y excepciones. Como?
- ❑ Procesador superescalar: retirada de varias instrucciones en un ciclo.

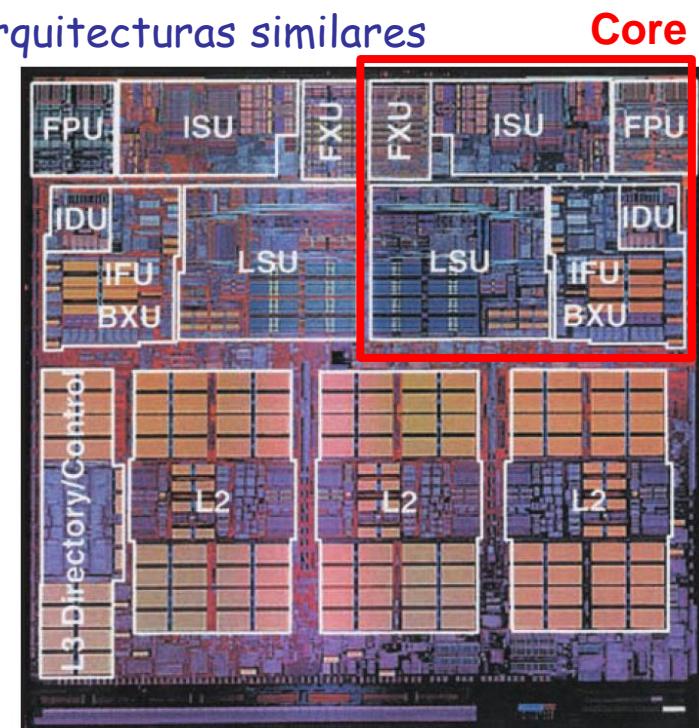
Superescalar con planificación dinámica: Commit (2)

- Si la instrucción I es un salto mal predicho, el contenido del ROB y todas las instrucciones en las colas deben eliminarse
 - EL ROB debe guardar la predicción de los saltos para poder compararlo en commit (o antes) con el comportamiento real
- Suposición: Fichero de Reg Unificado
 - Restaurar la Frontend RAT. Todos los renombramientos hechos corresponden a instr en ROB (a eliminar) o finalizadas (el Reg. Físico asociado al Reg. Arquitectónico está anotado en la Retirement RAT).
 - Solución más simple:
 - La retirement RAT no se ve alterada por el fallo de predicción!! Indica los regs que forman el estado arquitectónico.
 - Los registros indicados en la retirement RAT son lo que deben suministrar los operandos a las nuevas instrucciones que refieren dichos registros → Basta con copiar la retirement RAT sobre la fronted RAT y comenzar a buscar instrucciones por la rama correcta.
 - Problema: Gran número de instrucciones en las colas → muchos ciclos de reloj perdidos, mucha energía gastada en trabajo inútil.
 - Soluciones más ágiles: Anticipar detección saltos mal predichos
 - Ejecutar y finalizar las instr más antiguas, Desechar el salto y todas las instr más jóvenes.
 - Problema: Cómo distinguirlas en las colas de emisión? Cómo reconstruir la frontend RAT?

Superescalar con planificación dinámica: Commit (3)

- Dispatch y commit de grupos de instrucciones. Caso de estudio: IBM POWER4 (2000)
 - Capacidad para emitir (issue) hasta 8 instr/ciclo (P9 12). 8 UFs en paralelo (P9 16UFs)
 - Tasa de finalización (commit) de 5 instr/ciclo
 - Puede albergar más de 200 instrucciones en proceso de ejecución en un instante dado (P9 256)
 - Varios procesadores posteriores de IBM con arquitecturas similares

Fuente: IBM J. Res. & Dev., January 2002



Micro-fotografía del chip:

- Dos procesadores POWER4 encapsulados en un chip (P9 24 cores)
- Frecuencia: hasta 1.3 GHz (P9 3,5GHz)
- Tecnología: 180 nm (P9 14nm)
- 174 Mtransistores en el chip (P9 8000 Mtran.)
- Interconexiones: 7 capas de metal (cobre) (P9 17capas)
- Cache L2 (~1.5 Mbytes) compartida por los dos procesadores: ocupa más de la mitad del chip (P9 L3 120MB)

Superescalar con planificación dinámica: Commit (4)

POWER4: Diagrama de bloques

Instruction Fetch Address Register

Búsqueda de saltos en el grupo de instr leídas en la I-cache (hasta 2 saltos)

Dispatch de grupos (en orden):

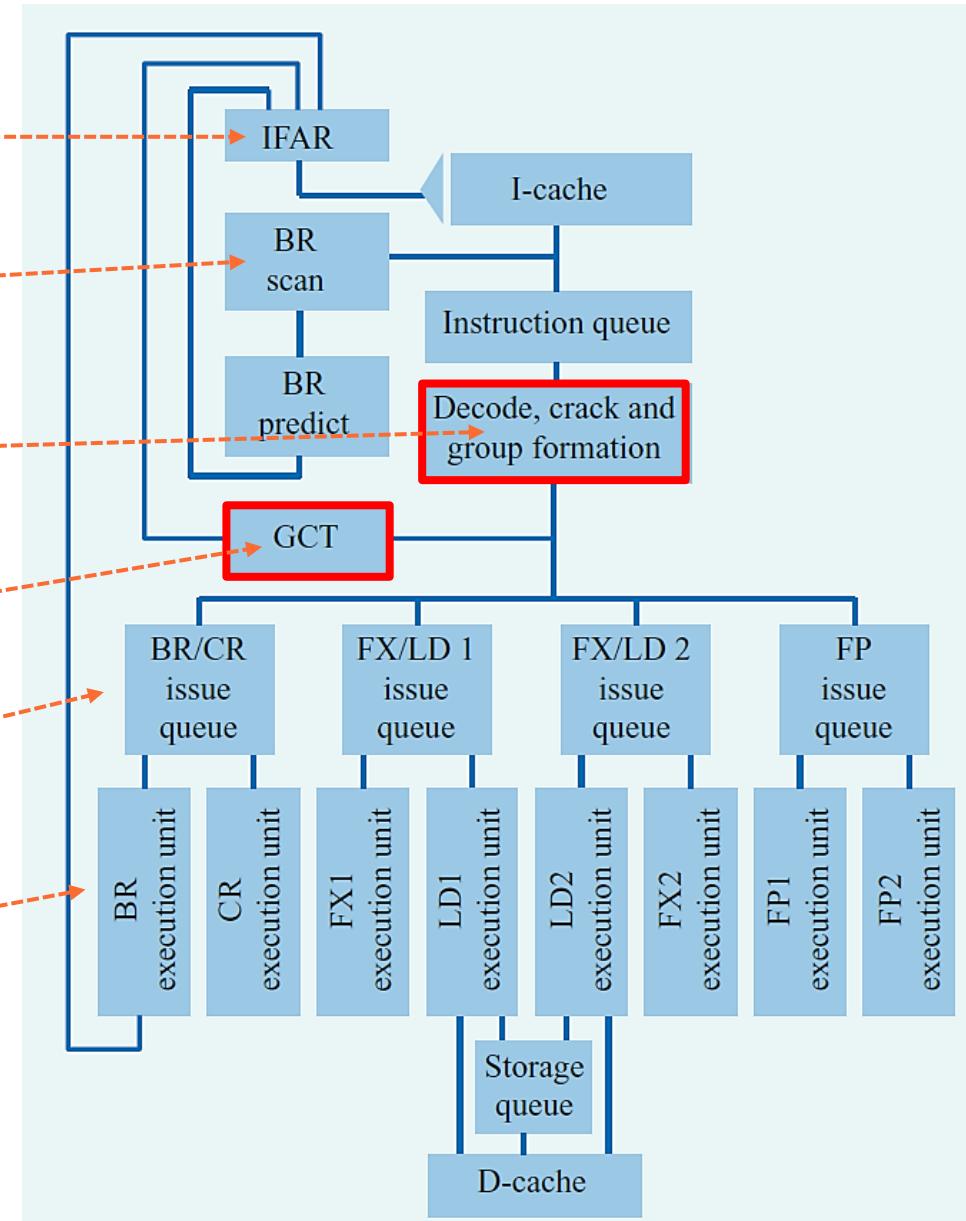
Forma grupos de hasta 5 instr, con una de salto como max. Si el salto existe, va en último lugar. Envío a back-end en orden.

Commit de grupos (en orden):

Group Completion Table. Cada ciclo puede hacer commit de un grupo completo.

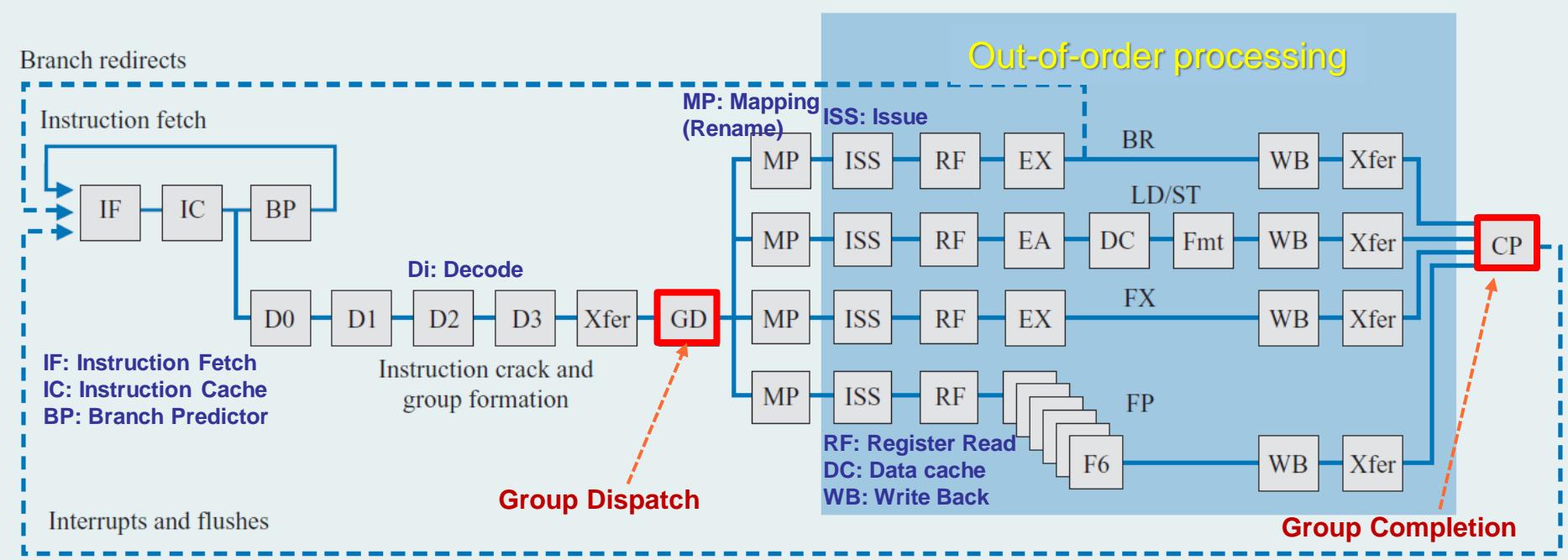
Issue Queues

Unidades Funcionales



Superescalar con planificación dinámica: Commit (5)

□ POWER4: Estructura del pipeline



Organización de 1 grupo:

IOP1	IOP2	IOP3	IOP4	BR / NOP
------	------	------	------	----------

- IOP: operación interna. En general es una instr del LM. Pero algunas instr complejas pueden descomponerse en varias IOPs.
- Solo hay un salto como máximo y va en la última posición. Si aparece antes, entonces los huecos IOPi se rellenan con NOP y el salto se pone en la última posición.
- Dispatch de un grupo precisa: 1 entrada libre en GCT, 1 entrada libre para cada instr del grupo en la Issue Q correspondiente, 1 registro libre para renombrar el reg destino de cada instr del grupo, 1 entrada libre en LD o ST Q para cada instr de LD o ST del grupo.
- La GCT puede alojar hasta 20 grupos. Cuando todas las instr del grupo han finalizado, el grupo entero se retira

Ejemplos: evolución procesadores Intel - P6

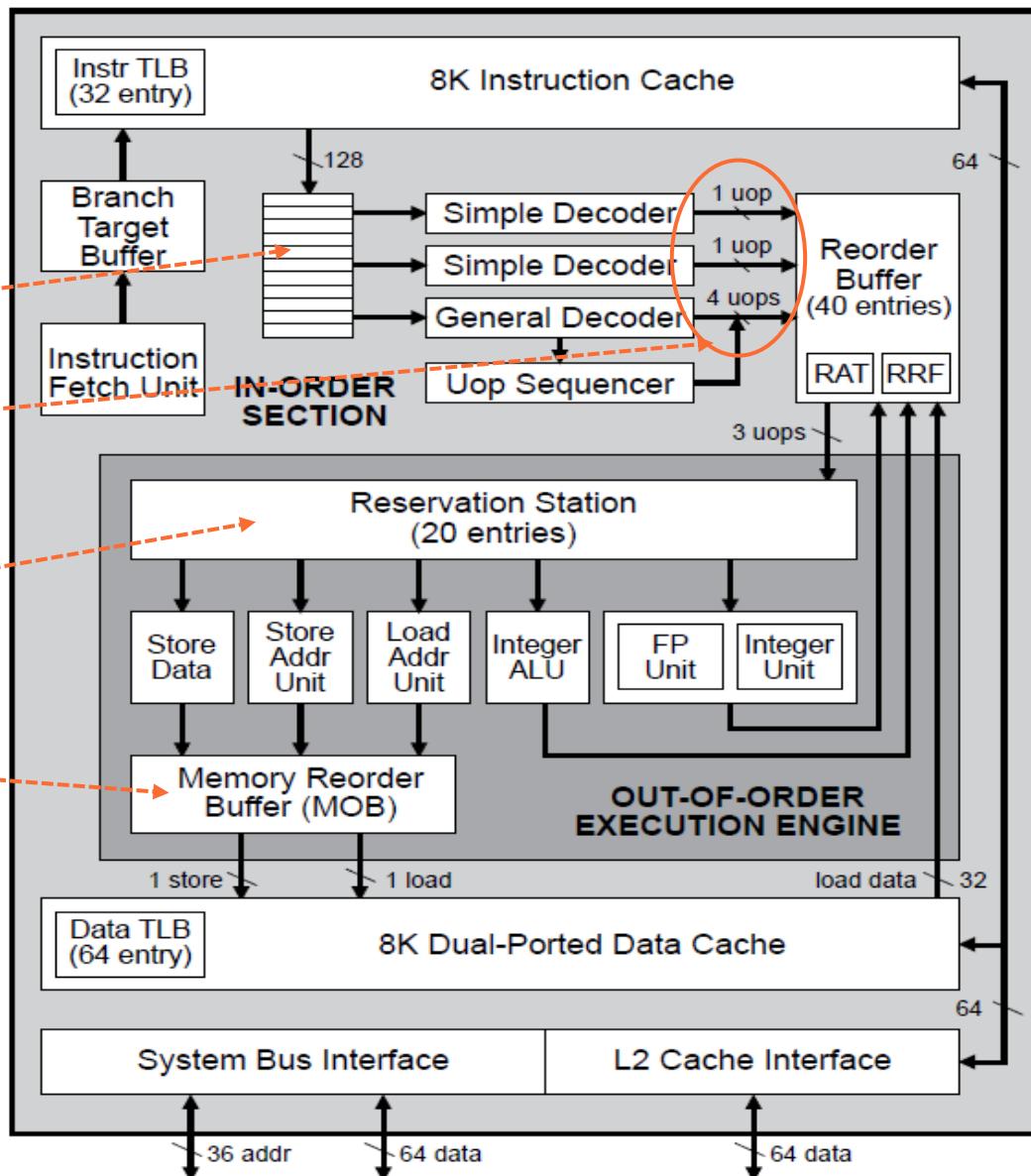
- Micro-Arquitectura usada desde Pentium Pro (1995) hasta Pentium III (1999)

Cola de instrucciones x86

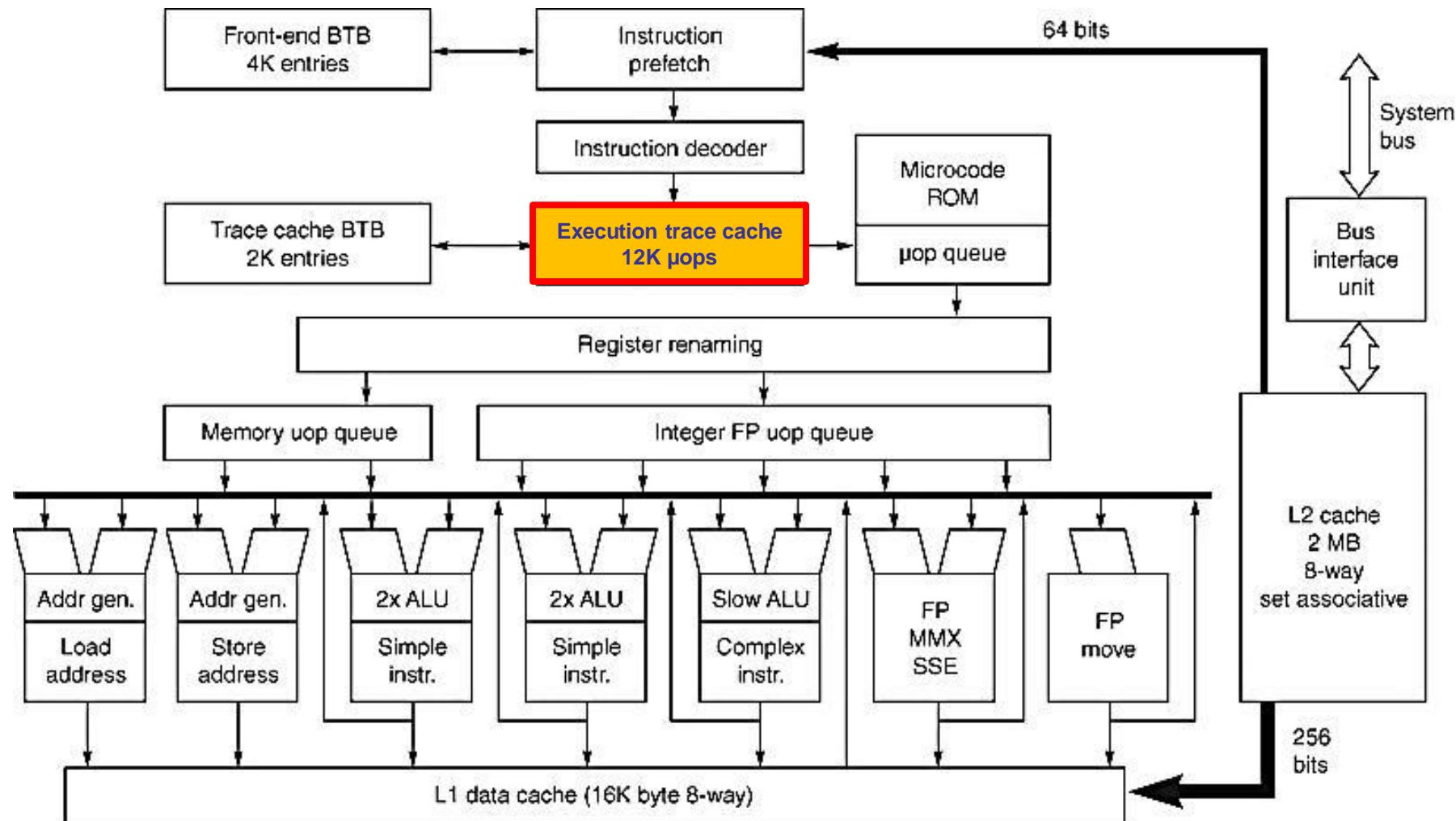
μ -ops descodificadas (2 simple decoder + 1 complex decoder)

Issue Queue unificada

Load Q y Store Q



Ejemplos: evolución procesadores Intel - Pentium 4 (Netburst)



© 2007 Elsevier, Inc. All rights reserved.

Intel Netburst Microarchitecture

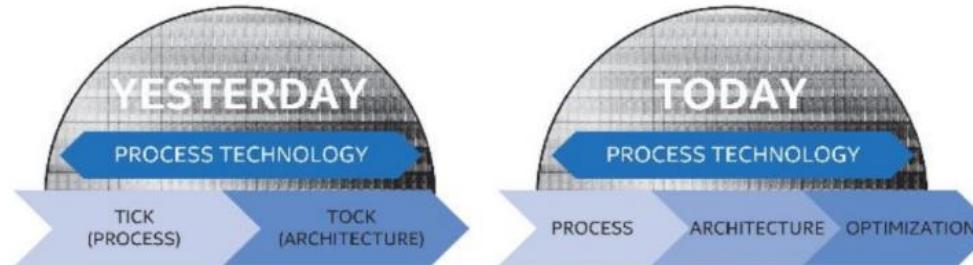
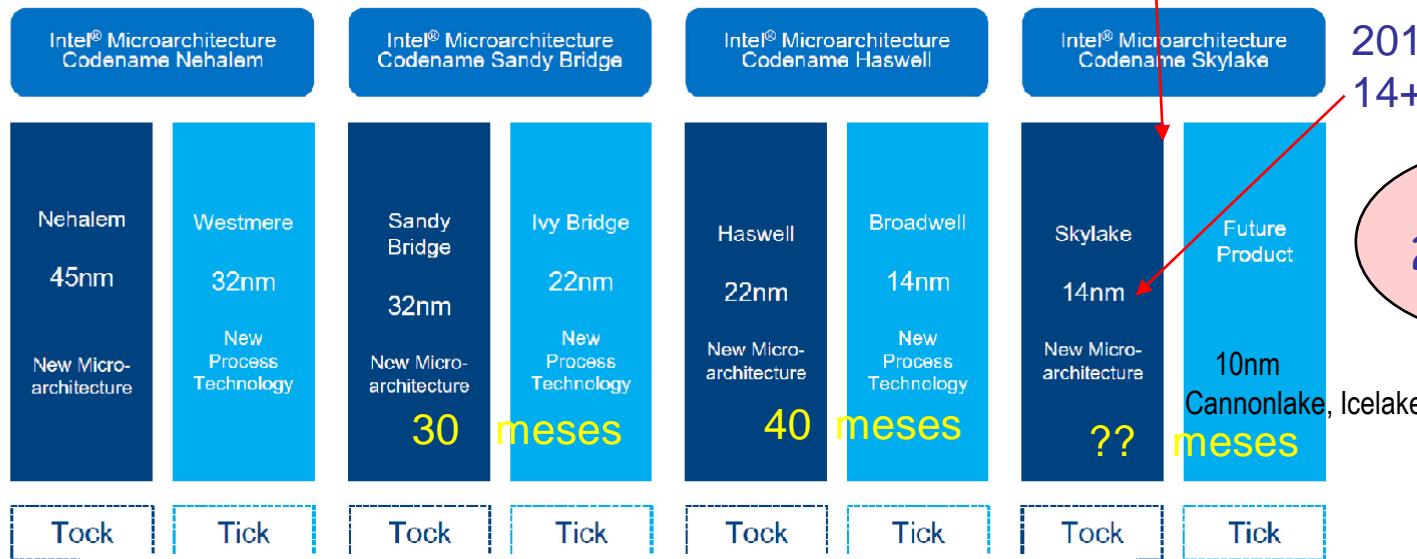
- ❑ Traduce instrucciones 80x86 a micro-ops (como P6)
- ❑ P4 tiene mejor predicción de saltos ($\times 8$) y más FU (7 versus 5)
- ❑ La Cache de instrucciones almacena micro-operaciones vs. 80x86 instrucciones.
“**trace cache**” (TC), BTB TC 2K entradas
 - En caso de acierto elimina decodificación
- ❑ Nuevo bus de memoria: 400(800) MHz vs. 133 MHz (RamBus, DDR, SDRAM)
(Bus@1066 Mhz)
- ❑ Caches
 - Pentium III: L1I 16KB, L1D 16KB, L2 256 KB
 - Pentium 4: L1I **12K uops**, L1D 16 KB 8-way, L2 2MB 8-way
- ❑ Clock :
 - Pentium III 1 GHz v. Pentium 4 1.5 GHz (**3.8 Ghz**)
 - **14 etapas en pipeline vs. 24 etapas en pipeline (31 etapas)**
- ❑ Instrucciones Multimedia: 128 bits vs. 64 bits => 144 instrucciones nuevas.
- ❑ Usa RAMBUS DRAM
 - Más AB y misma latencia que SDRAM. Costo 2X-3X vs. SDRAM
- ❑ ALUs operan al doble del clock para operaciones simples
- ❑ Registros de renombrado: 40 vs. 128; Ventana: 40 v. 126
- ❑ BTB: 512 vs. 4096 entradas. Mejora 30% la tasa de malas predicciones

Tick-Tock a Tick-Tock-Tock...

+3 tock Kaby Lake, Coffee Lake
Core i3-5-7 7^aG, 8^aG, 9^oG, 10^aG

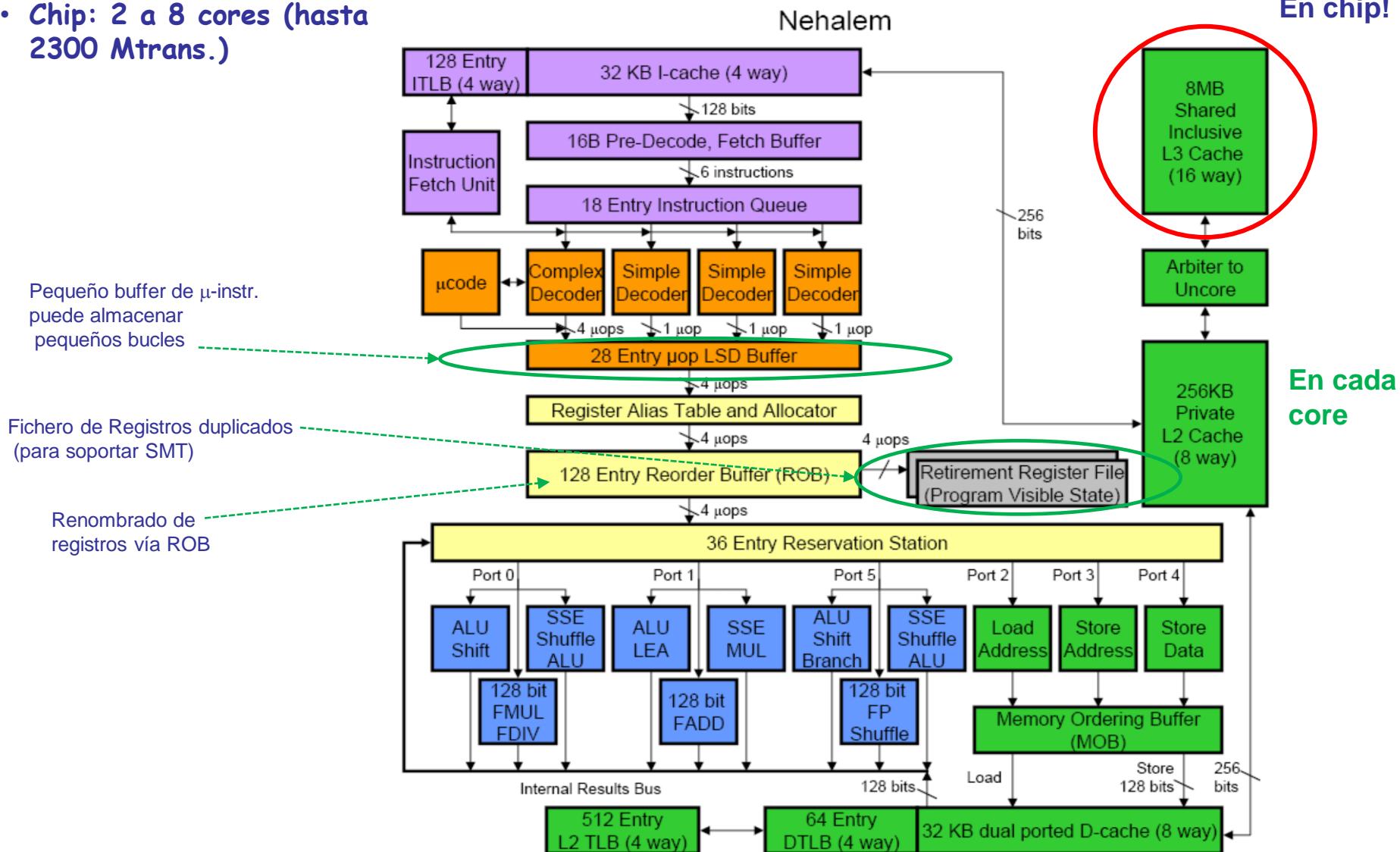
Tick-Tock Development Model:

Sustained Microprocessor Leadership

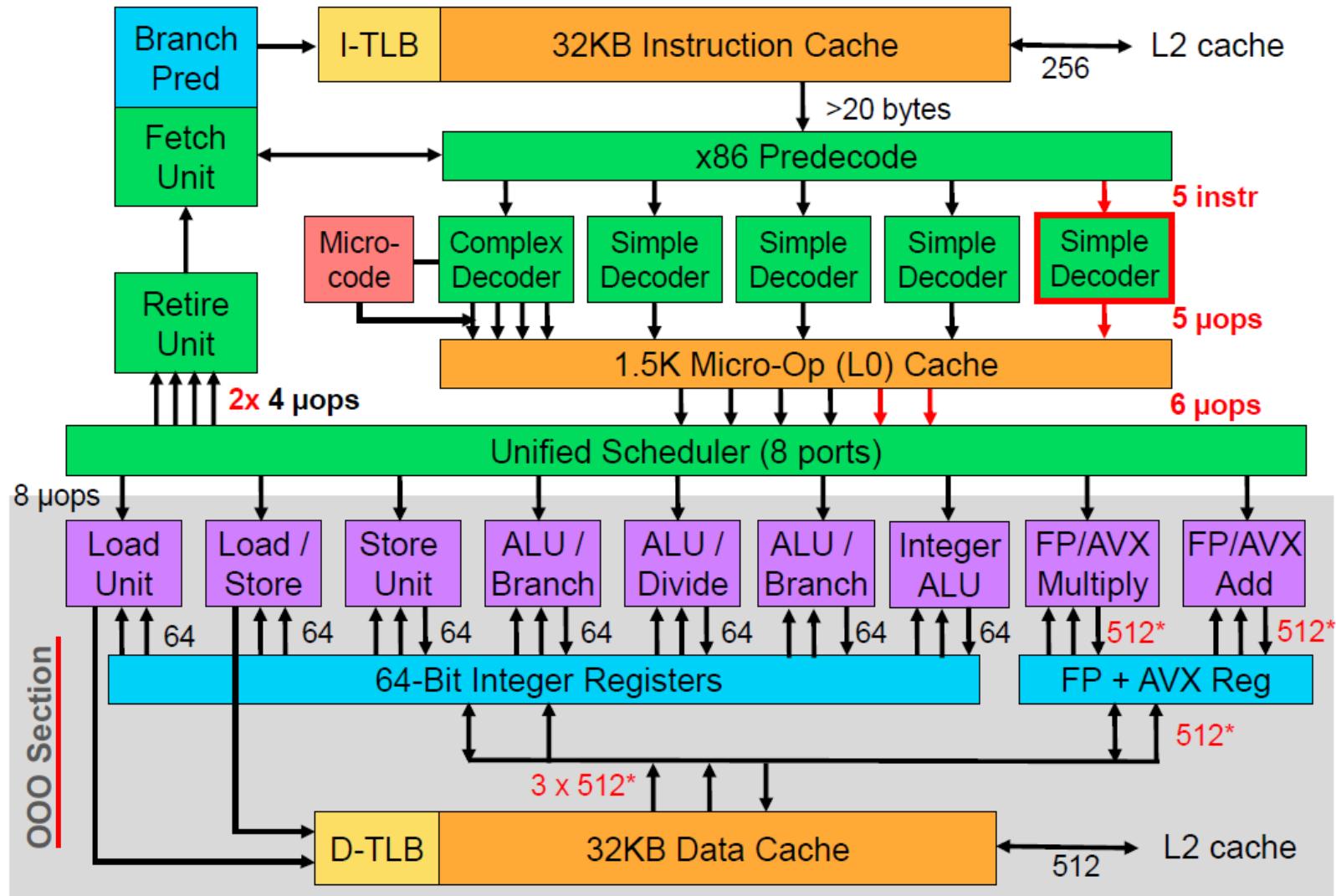


Nehalem Microarchitecture

- Core: 2 threads con SMT
- Chip: 2 a 8 cores (hasta 2300 Mtrans.)



Skylake Microarchitecture



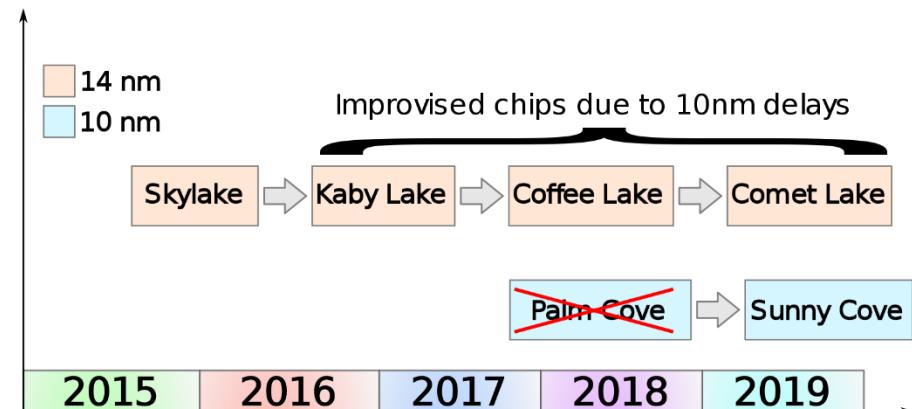
Skylake (2015)/Sunny Cove (2019) Microarchitecture

	Nehalem	Sandy Bridge	Haswell	Skylake	Sunny Cove
x86 Decoders	4 instr	4 instr	4 instr	5 instr	5 instr
Max Instr/Cycle	4 ops	6 ops	8 ops	8 ops	10 ops
Reorder Buffer	128 ops	168 ops	192 ops	224 ops	352 ops
Load Buffer	48 loads	64 loads	72 loads	72 loads	128 loads
Store Buffer	32 stores	36 stores	42 stores	56 stores	72 stores
Scheduler	36 entries	54 entries	60 entries	97 entries	
Integer Rename	In ROB	160 regs	168 regs	180 regs	
FP Rename	In ROB	144 regs	168 regs	168 regs	
Allocation Queue	28/thread	28/thread	56 total	64/thread	

Muchos detalles no públicos. 33% de mejora de rendimiento

Sunny Cove

- Large virtual address (57 bits, up from 48 bits)
- Larger L1 data cache (48 KB, up from 32 KB)
- Larger L2 cache (512 KB, up from 256 KB)



Límites del ILP

- ❑ El lanzamiento múltiple permite mejorar el rendimiento sin afectar al modelo de programación. 
- ❑ En los últimos años se ha mantenido el mismo ancho superescalar que tenían los diseños del 1995/2000. 
- ❑ La diferencia entre rendimiento pico y rendimiento obtenido crece. 
- ❑ El ILP es **inherente a cada aplicación**, y sólo puede ser “explotado” hasta cierto punto.
- ❑ Los límites del ILP están en:
 - La aplicación: dependencias LDE.
 - El hardware: los recursos del procesador (#registros, ERs, UFs, #entradas del ROB, etc).
 - No obstante, incluso con una máquina perfecta, las dependencias LDE imponen un límite teórico al ILP que se puede explotar.
- ❑ Mecanismos para superar el agotamiento del ILP:
 - ❑ Paralelismo a nivel de datos.
 - ❑ Extensiones Single Instruction Multiple Data (SIMD) o “multimedia”: Intel MMX, SSE, AVX.
 - ❑ Arquitecturas vectoriales (VMIPS).
 - ❑ Paralelismo a nivel de hilo.
 - ❑ Multithreading (varios tipos).
 - ❑ Una combinación de ambos: Graphical Processing Units (GPUs).

- Comparaciones necesarias en lanzamiento
 - Ejemplo: Lanzar 6 instrucciones/ciclo (check LDE)
 - $1-2 \ 1-3 \ 1-4 \ 1-5 \ 1-6 = 5 \text{ instr} \times 2 \text{ operandos} = 10 \text{ comp}$
 - $2-3 \ 2-4 \ 2-5 \ 2-6 = 4 \text{ instr} \times 2 \text{ operandos} = 8 \text{ comp}$
 -
 - $5-6 = 1 \text{ instr} \times 2 \text{ operandos} = 2 \text{ comp}$
 - TOTAL = 30 comp
 - En general: lanzar n instrucciones → n^2-n comp

- Comparaciones necesarias para paso de operandos
 - Ejemplo: pueden acabar 6 instr/ciclo, 80 instr en la ventana, 2 operandos/instr.
 - $6 \times 80 \times 2 = 960$ comparaciones/ciclo
 - En general, comparar con los operandos de todas las instrucciones que están esperando a ser ejecutadas.

□ Buscar paralelismo de más de un thread

- Hay mucho paralelismo en algunas aplicaciones (Bases de datos, códigos científicos, videojuegos, multimedia)
- Thread Level Parallelism (TLP)
 - Thread: proceso con sus propias instrucciones y datos
 - Cada thread puede ser parte de un programa paralelo de múltiples procesos, o un programa independiente.
 - Cada thread tiene todo el estado (instrucciones, datos, PC, register state, ...) necesario para permitir su ejecución
 - Arquitecturas (MultiThreading, multiprocesadores/comput. paralelos)
 - Tema 3 (de aquí en adelante)
- Data Level Parallelism (DLP): Operaciones idénticas sobre grandes volúmenes de datos (extensiones multimedia, GPU y arquitecturas vectoriales)
 - Tema 4

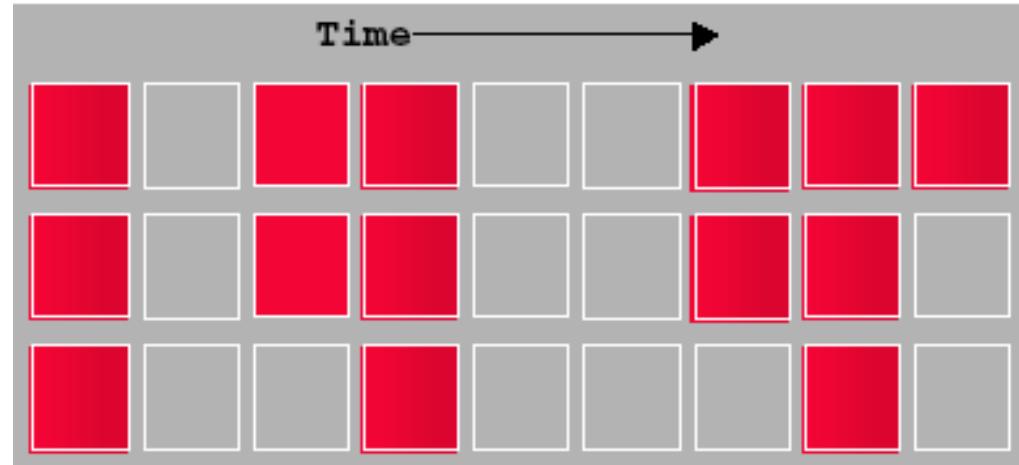
¿Por qué multithreading?

□ Procesador superescalar

(P. ej.: Hasta 3 FUs pueden comenzar una op en cada ciclo)

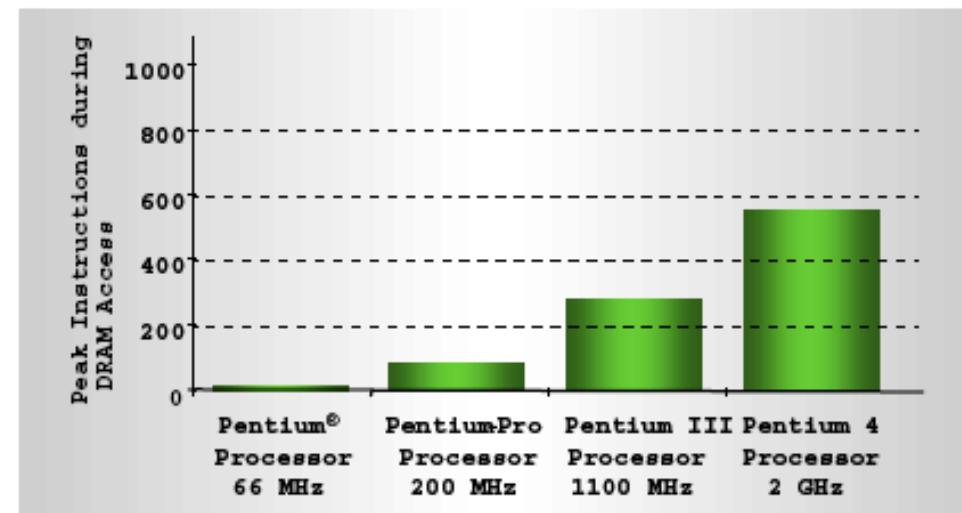
Causas de baja utilización de UF:

- No hay instrucciones que ejecutar (fallo cache)
- Hay instr., pero faltan operandos
- Hay UF libres, pero no del tipo necesario



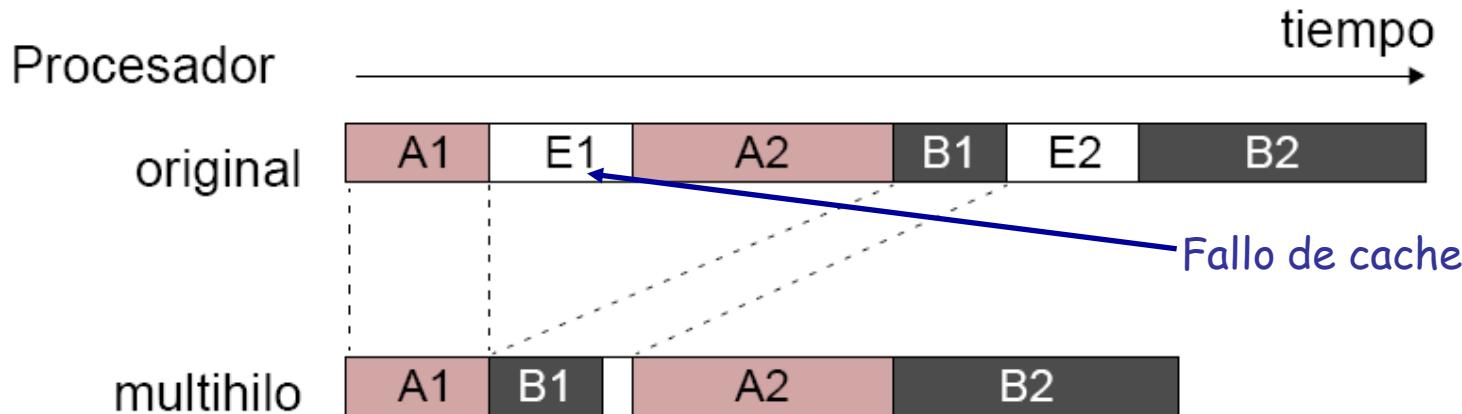
□ La latencia de memoria crece.

¿Cómo soportarla?



□ Cómo funciona el Multithreading

□ Ejemplo fallo de cache



- Incrementar el trabajo procesado por unidad de tiempo
- Si los hilos son del mismo trabajo se reduce el tiempo de ejecución

La técnica multithreading no es ideal y se producen pérdidas de rendimiento. Por ejemplo, un programa puede ver incrementado su tiempo de ejecución aunque el número de programas ejecutados por unidad de tiempo sea mayor cuando se utiliza multithreading.

□ Ejecución Multithreaded

- Multithreading: múltiples threads comparten los recursos del procesador
 - El procesador debe mantener el estado de cada thread e.g., una copia de bloque de registros, un PC separado, tablas de páginas separadas.
 - La memoria compartida ya soporta múltiples procesos.
 - HW para conmutación de thread muy rápido. Mucho mas rápido que entre procesos.
- ¿Cuándo conmutar?
 - Cada ciclo conmutar de thread (grano fino)
 - Cuando un thread debe parar (por ejemplo fallo de cache)
- HEP (1978), Alewife (1990) , M-Machine (1995) , Tera-Computer (1999)

Lectura recomendada

H&P 6th ed., Sección 3.11: págs. 242-245

□ Multithreading de Grano Fino

- Commuta entre threads en cada ciclo, entrelazando la ejecución de los diferentes thread.
- Generalmente en modo “round-robin”, los threads bloqueados se saltan
- La CPU debe ser capaz de commutar de thread cada ciclo.
- Ventaja; puede ocultar stalls de alta y baja latencia, cuando un thread está bloqueado los otros usan los recursos.
- Desventaja; retarda la ejecución de cada thread individual, ya que un thread sin stall es retrasado por reparto de recursos (ciclos) entre threads
- Ejemplo: Niagara y Niagara 2 (SUN-Oracle)

□ Multithreading Grano Grueso

- Commuta entre threads solo en caso de largos stalls, como fallos de cache LLC (L3)
- Ventajas
 - No necesita conmutación entre thread muy rápida.
 - No retarda cada thread, la conmutación solo se produce cuando un thread no puede avanzar.
- Desventajas; no elimina pérdidas por stalls cortos. La conmutación es costosa en ciclos.
 - Como CPU lanza instrucciones de un nuevo thread, el pipeline debe ser vaciado.
 - El nuevo thread debe llenar el pipe antes de que las instrucciones empiecen a completarse.
- Ejemplos: IBM AS/400, Montecito (Itanium2 9000), Sparc64 VI

□ Simultaneous Multithreading

Motivación: Recursos no usados en un procesador superescalar

Un thread, 8 unidades

Ciclo M M FX FX FP FP BR CC

1	Y							Y
2	Y	Y					Y	
3			Y	Y				
4								
5								
6								
7	Y		Y		Y			
8		Y		Y				
9			Y					

Dos threads, 8 unidades

Ciclo M M FX FX FP FP BR CC

1	Y	B		B				
2	Y		Y		B		Y	
3	B				Y	Y		
4	B		B				B	
5			B					B
6								
7	Y			B	Y	B	Y	
8			Y		B	Y		B
9	B	B			Y			

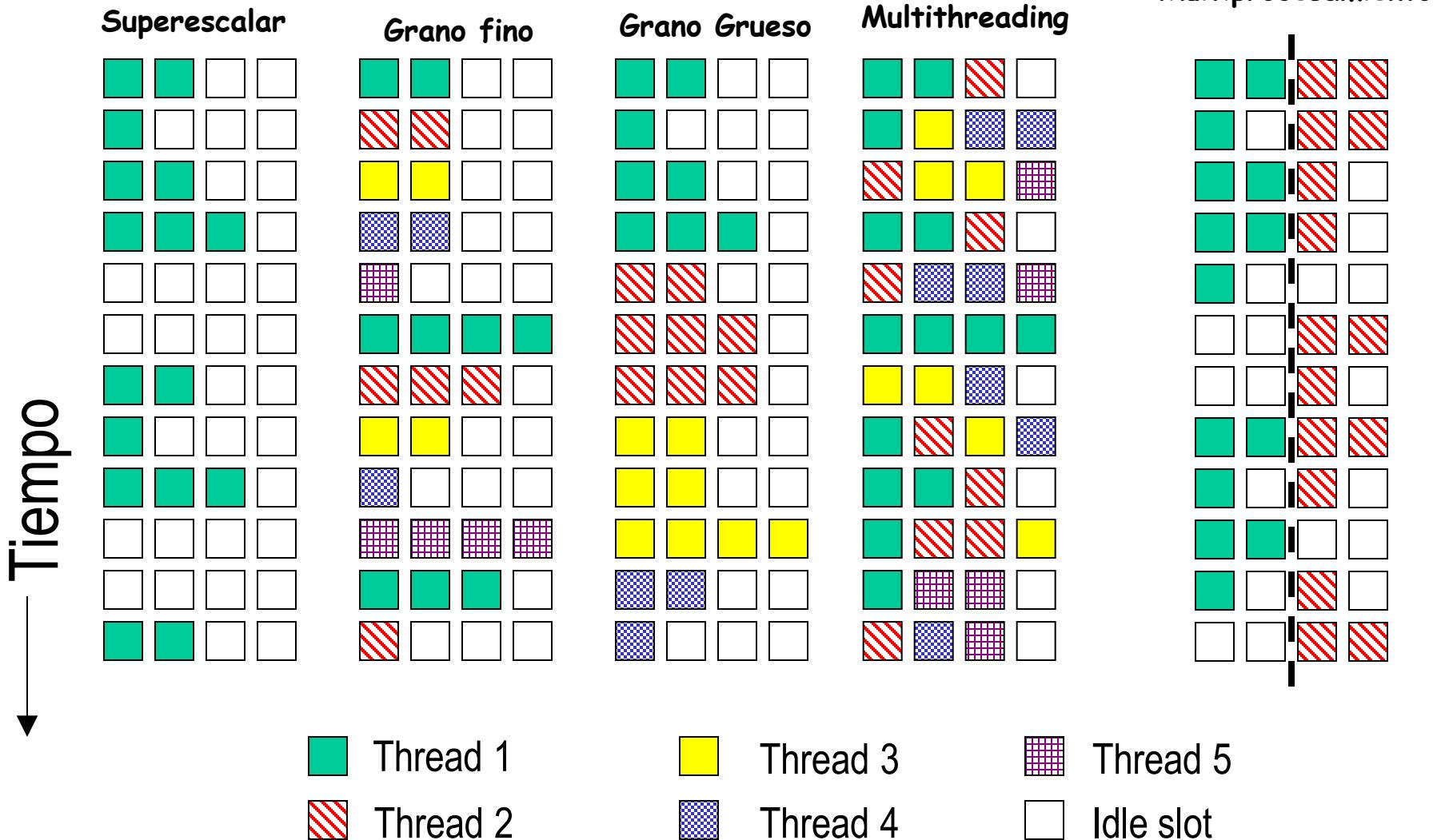
M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

□ Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT): dentro de un procesador superescalar fuera de orden ya hay mecanismos HW para soportar la ejecución de más de un thread
 - Gran número de registros físicos donde poder mapear los registros arquitectónicos de los diferentes threads
 - El renombrado de registros proporciona un identificador único para los operandos de una instrucción, por tanto instrucciones de diferentes thread se pueden mezclar sin confundir sus operados
 - La ejecución fuera de orden permite una utilización eficaz de los recursos.
- Sólo necesitamos sumar una tabla de renombrado por thread y PC separados
 - Commit independiente se soporta con un ROB por thread (Lógico o físico)
- Ojo conflictos en la jerarquía de memoria
- Ejemplos: Pentium4, POWER5,6,7,8,9 Nehalem (2008), ... AMD Zen, SPARC64-VII,...XII

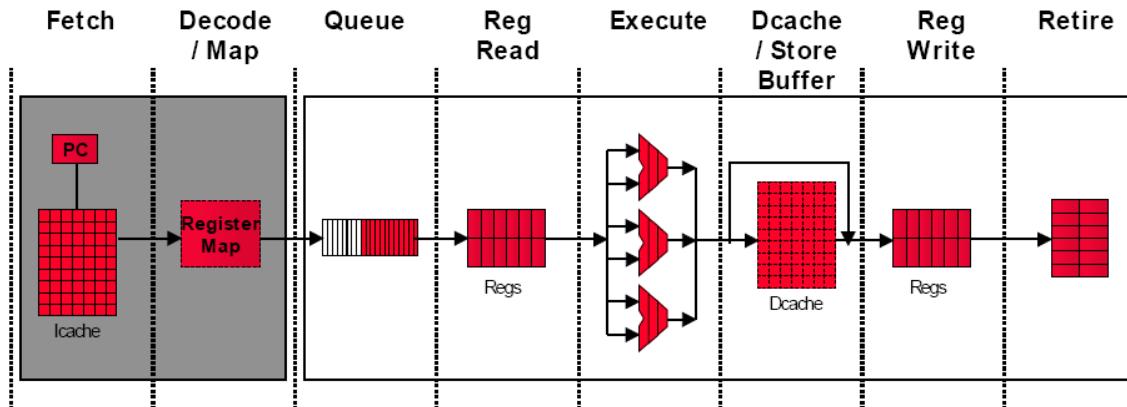
Multithreading

□ Comparación



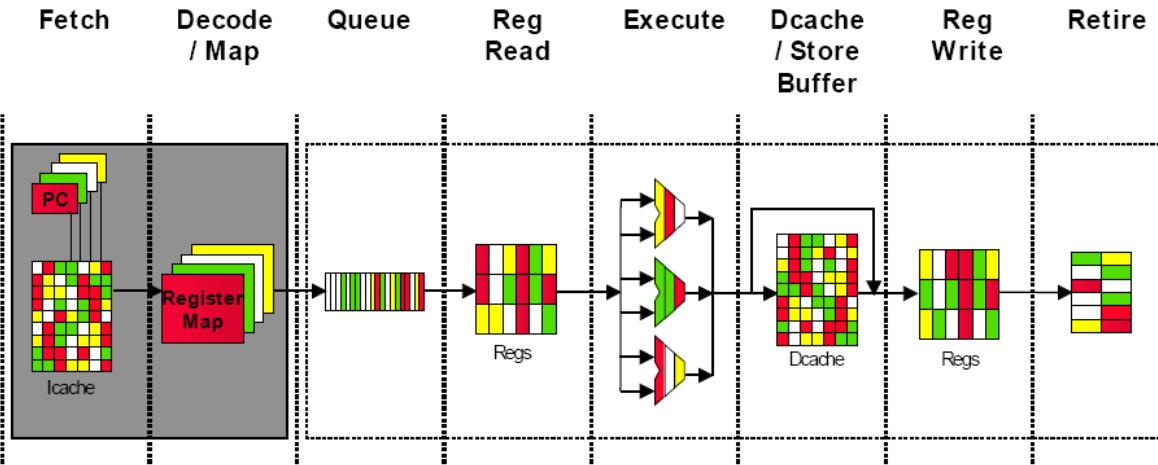
□ Simultaneous multithreading

- Un solo hilo: un flujo de instrucciones



✓ todos los recursos utilizados por un hilo

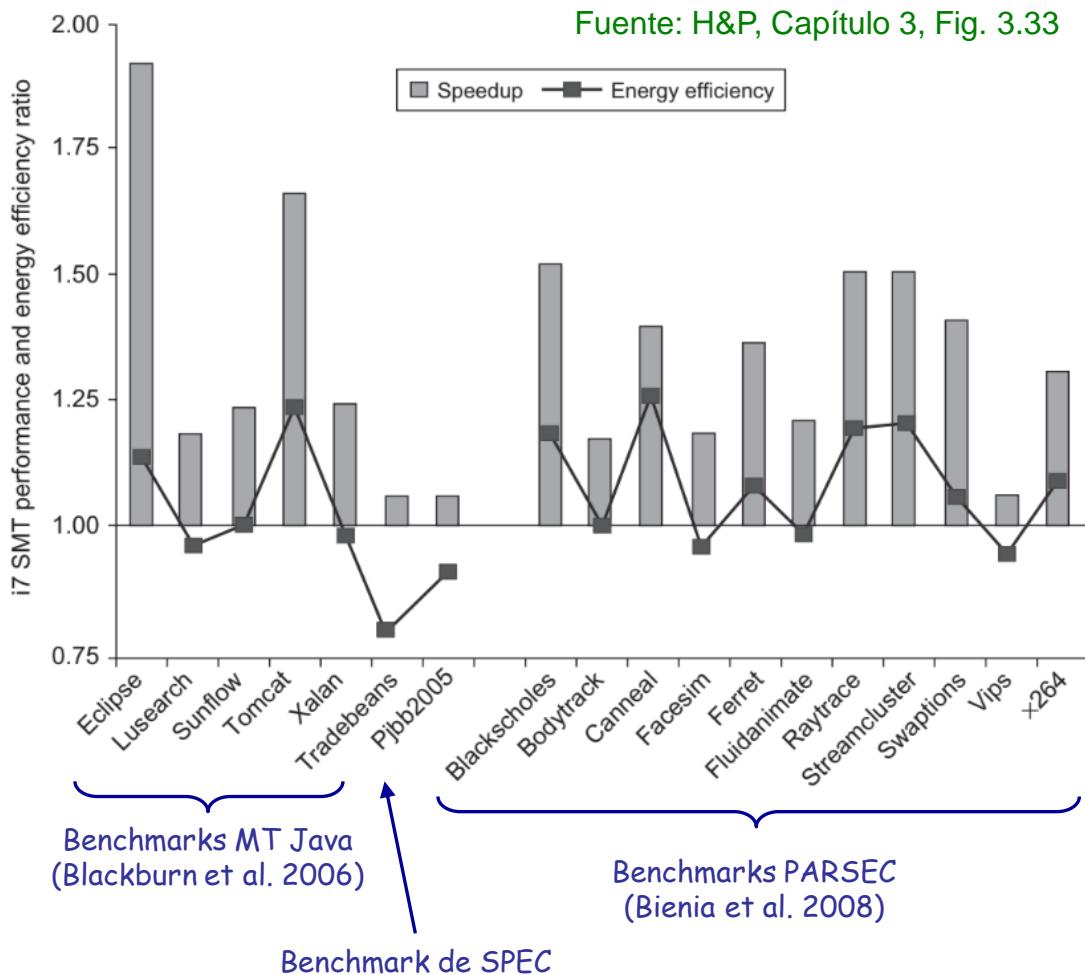
- Multihilo: varios flujos de instrucciones



✓ recursos para distinguir el estado de los hilos
✓ los otros recursos se pueden compartir

SMT: prestaciones vs. eficiencia energética

- El SMT acelera todos los benchmarks considerados (factor de media = 1.28 para MT Java y 1.31 para PARSEC).



- La mayoría son también **eficientes desde el punto de vista energético**.

- Tradebeans y Pjbb2005 son las únicas excepciones notables.
- La eficiencia energética E se define aquí como:

$$E = \frac{T_{no_SMT}}{T_{SMT}} / \frac{P_{SMT}}{P_{no_SMT}}$$

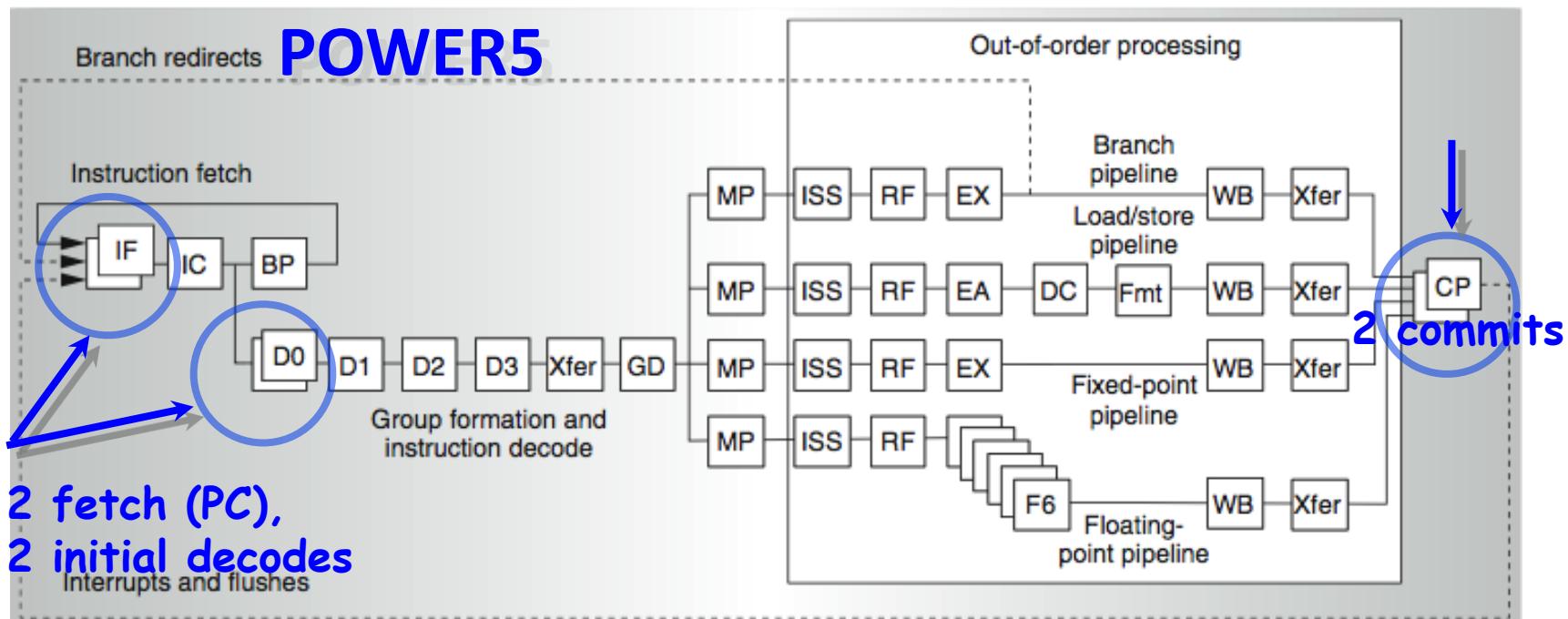
- $E > 1 \rightarrow$ "El tiempo de ejecución decrece en un factor superior al que el consumo de energía aumenta".

Pentium-4 Hyperthreading (2002)

- Primer procesador comercial SMT 2 threads
- Los procesadores lógicos comparten casi todos los recursos del procesador físico.
 - Caches, unidades de ejecución, predictor de saltos
- Overhead de área por hyperthreading ~ 5%
- Cuando un procesador lógico (thread) se detiene el otro puede progresar usando los recursos
 - Ningún thread puede usar todos los recursos (colas) cuando hay 2 threads activos.
- Procesadores ejecutando solo un thread: se ejecuta a la misma velocidad que sin hyperthreading
- Hyperthreading se eliminó de los diseños OoO sucesores del Pentium-4 (Pentium-M, Core Duo, Core 2 Duo), fue de nuevo incluido con el Nehalem en 2008 y se ha mantenido en todos los sucesores.

SMT Multithreading

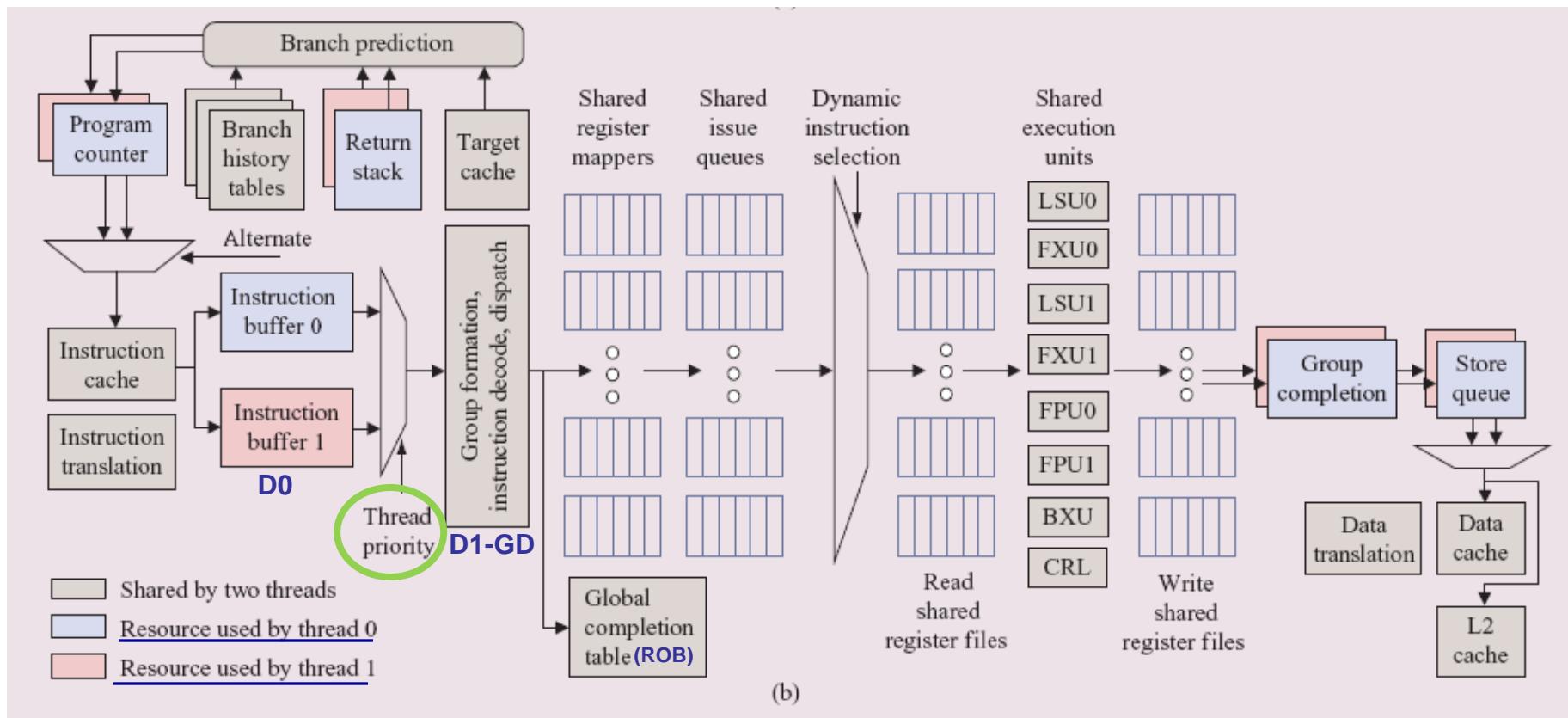
- IBM POWER5: Una adaptación de la arquitectura POWER4 para SMT



Observar: Muy pocos recursos duplicados, en comparación con la arquitectura del POWER4 que hemos estudiado.

Fuente: IBM J. Res. & Dev., Jul./Sep. 2005

□ IBM POWER5 (2005): una vision más detallada



¿Por qué sólo 2 threads? Con 4, los recursos compartidos (registros físicos, cache, AB a memoria) son un cuello de botella.

□ POWER5

Balanceo de la carga dinámica

1- Monitorizar

cola de fallos (load en L2)

entradas ocupadas en ROB (GCT)

2 - Quitarle recursos;

Reducir prioridad del hilo,

Inhibir decodificación (L2 miss)

Eliminar instrucciones desde emisión y
parar decodificación

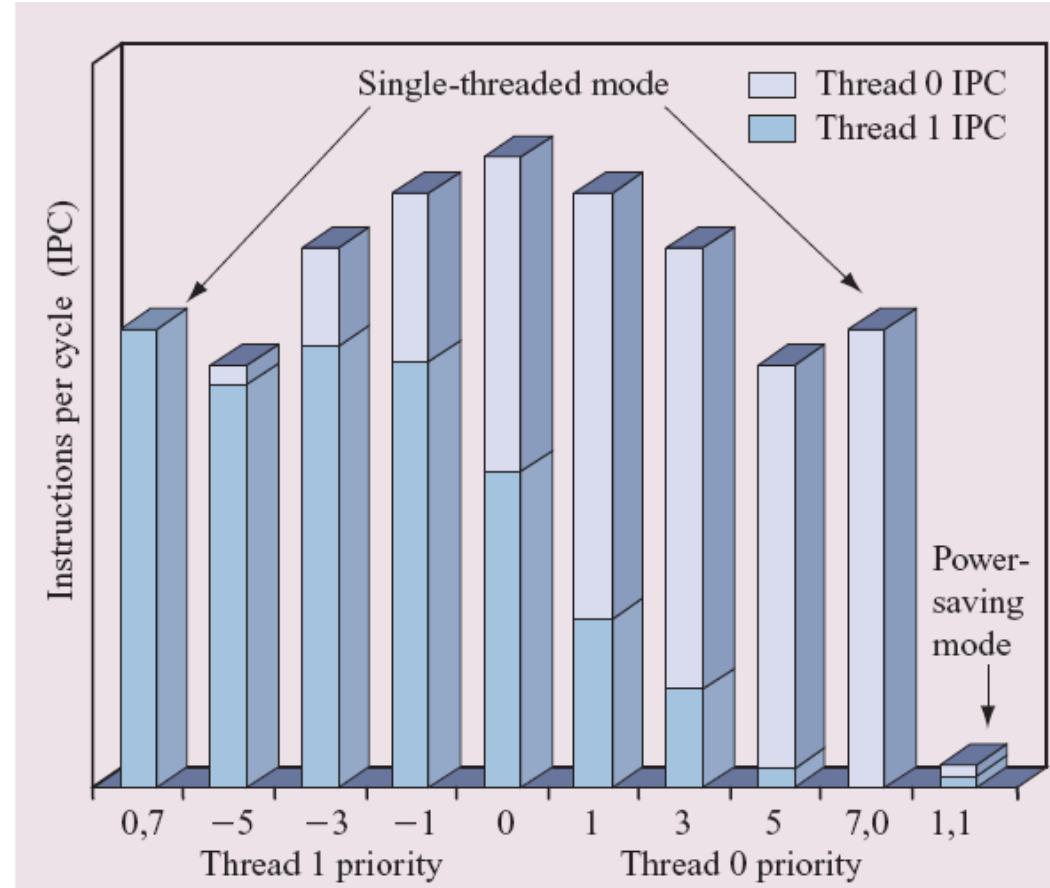
3- Ajustar prioridad del hilo (hardware/software)

Baja en espera activa

Alta en tiempo real

8 niveles de prioridad

Da más ciclos de decodificación al de más
prioridad



❑ Cambios en POWER5 para soportar SMT

- Incrementar asociatividad de la L1 de instrucciones y del TLB
- Una cola de load/stores por thread
- Incremento de tamaño de la L2 (1.92 vs. 1.44 MB) y L3
- Un buffer de prebúsqueda separado por thread
- Incrementar el numero de registros físicos de 152 a 240
- Incrementar el tamaño de las colas de emisión
- El POWER5 core es 24% mayor que el del POWER4 para soportar SMT
- Más consumo, pero soportado con DVFS

SMT Multithreading

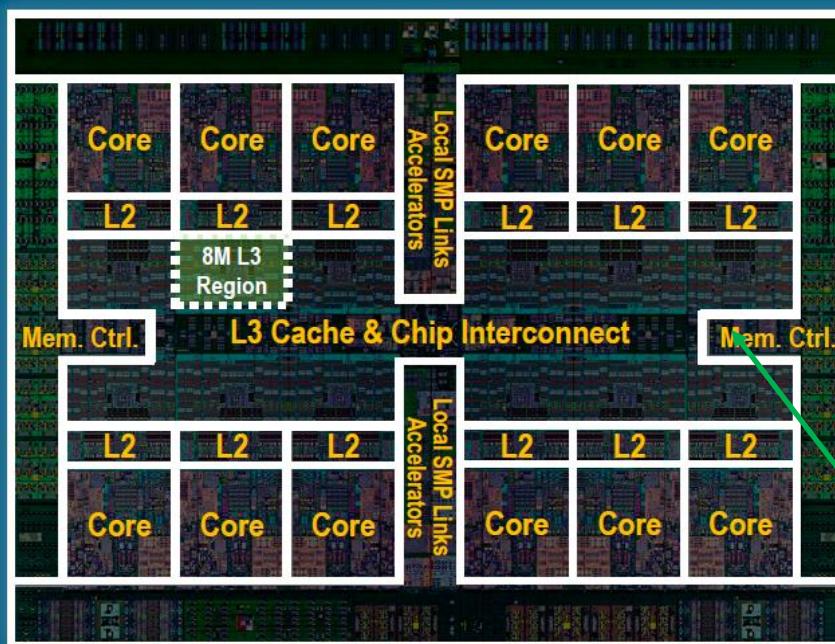
POWER8 (2015)

Cores

- 12 cores (SMT8)
- 8 dispatch, 10 issue, 16 exec pipe
- 2X internal data flows/queues
- Enhanced prefetching
- 64K data cache, 32K instruction cache

Accelerators

- Crypto & memory expansion
- Transactional Memory
- VMM assist
- Data Move / VM Mobility



Energy Management

- On-chip Power Management Micro-controller
- Integrated Per-core VRM
- Critical Path Monitors

Power 9 (2018) 24cores, 14nn, 8000Mtrans, 120MB L3

Power 10 (2021), 32cores, 7nm, 607mm², 128MB L3, 18 ML

Caches

- 512 KB SRAM L2 / core
- 96 MB eDRAM shared L3
- Up to 128 MB eDRAM L4 (off-chip)

Memory

- Up to 230 GB/s sustained bandwidth

Bus Interfaces

- Durable open memory attach interface
- Integrated PCIe Gen3
- SMP Interconnect
- CAPI (Coherent Accelerator Processor Interface)

Área de L3 relativamente pequeña: gracias al uso de "embedded DRAM"

- Doblar en ancho de emisión (issue rates) sobre los valores actuales 4-8 instrucciones por ciclo, a digamos 12 a 16 instrucciones requiere en el procesador
 - Más de 4 accesos a cache de datos por ciclo,
 - Predecir-resolver de 3 a 4 saltos por ciclo,
 - Renombrar más de 12 y acceder a más de 24 registros por ciclo
 - Buscar de la cache de instrucciones de 12 a 24 instrucciones por ciclo.
- La complejidad de implementar estas capacidades implica al menos sacrificar la duración del ciclo e incrementa de forma muy importante el consumo.

- La mayoría de la técnicas que incrementan rendimiento incrementan también el consumo.
- Una técnica es *eficiente en energía* si incrementa más el rendimiento que el consumo. (límite!!!)
- Todas las técnicas de emisión múltiple (más allá de cierto punto) son poco eficientes desde el punto de vista de la energía.
- Hemos visto que el SMT sí es eficiente desde el punto de vista de la energía, en la mayoría de los casos estudiados (tr. 69).

Conclusiones -Limites del ILP

- En lugar de seguir explotando el ILP, los diseñadores se han focalizado sobre multiprocesadores en un chip (CMP, multicores,...)
- En el 2000, IBM abrió el campo con el 1^{er} multiprocesador en un chip, el POWER4, que contenía 2 procesadores POWER3 y una cache L2 compartida. A partir de este punto todos los demás fabricantes han seguido el mismo camino. (Intel, AMD, Sun/Oracle, Fujitsu, y también Apple, Samsung, Qualcomm...).
- La explotación del paralelismo existente en las aplicaciones (a nivel de procesos y/o a nivel de datos) se plantea como la vía adecuada para obtener balances satisfactorios entre velocidad de procesamiento y consumo de energía...
- ...pero las ventajas de lanzar un nº razonable de instrucciones por ciclo, unida a la capacidad de la ejecución especulativa en desorden para adaptarse a eventos imprevisibles en tiempo de compilación (como los fallos de cache L1), hacen que los procesadores **especulativos con un ILP limitado** sigan siendo el bloque básico de los diseños multicore.

Lectura recomendada

H&P 6th ed., Sección 3.14: *Concluding Remarks: What's Ahead?*