AC 21-22, tema 4: arquitecturas con paralelismo a nivel de instrucción

1. Introducción

Mejora de la tecnología nos lleva a mejoras en los transistores, más pequeños y rápidos. La velocidad en microprocesadores llega de la mano de dos estrategias: aumento de la frecuencia del reloj, y diseño de microarquitecturas capaces de completar varias instrucciones por ciclo (superescalares).

> **EXPRESIONES VELOCIDAD DE PROCESAMIENTO:**
>
> Vcpu = NI / Tcpu = NI / (NI · CPI · Tciclo) = 1 / (CPI · Tciclo) = instr por ciclo · freq

La frecuencia F y el num instr ciclo IPC han permitido aumentar la velocidad exponencialmente, como dicta la Ley de Moore, hasta los primeros años del sXXI donde se empieza a estancar. En ese momento surgen los primeros sistemas multicore.

En este tema vamos a analizar como estos sistemas multicore nos permiten el **paralelismo entre instrucciones ILP**. Los sistemas multicore se basan en la segmentación de cauce o pipelining, según la cual el procesador sigue varias etapas que procesan independientemente las distintas fases de una instrucción.

Ejemplo: etapa de captación IF (instruction fetch); decodificación ID que toma lo obtenido de la fase previa y obtiene sus operandos; etapa de ejecución EX y así.

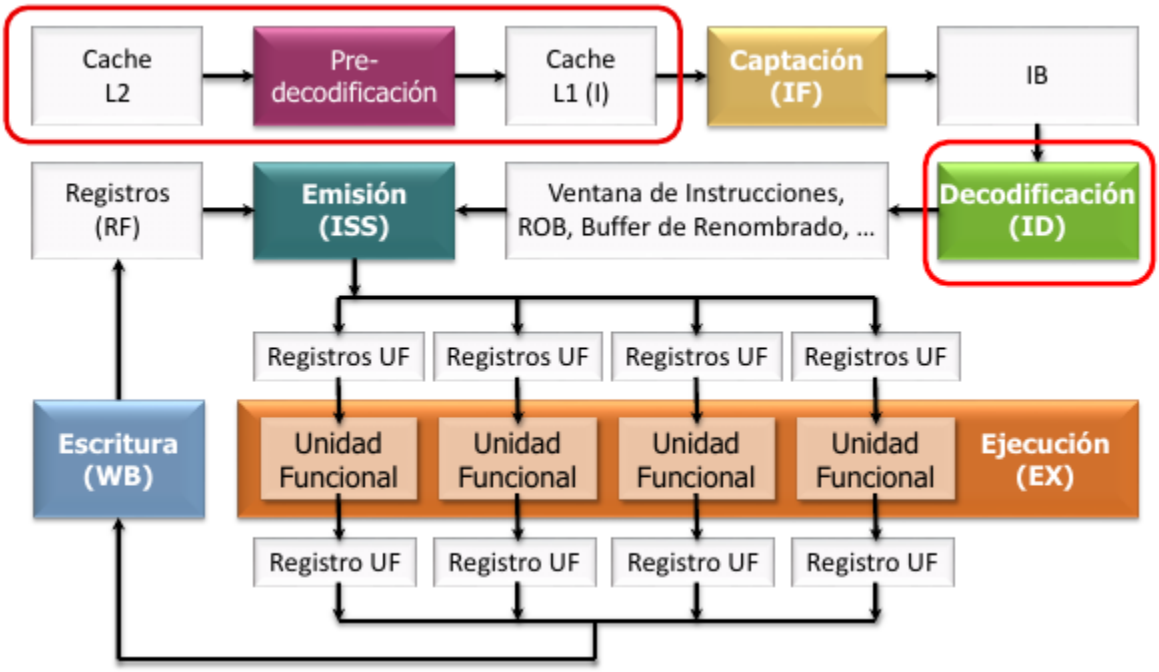
Las etapas definen un cauce en el que cada etapa trabaja con la salida de la anterior y genera resultados para la posterior. Así, se puede aprovechar el ILP, procesando distintas etapas de distintas instrucciones simultaneamente. Pero hay que tener cuidado con las dependencias de datos, dependencias de control (salto incondicional) y colisiones entre instrucciones que necesitan el mismo recurso a la vez. Estos problemas generan esperas que reducen el rendimiento del cauce.

Las arquitecturas que nos van a permitir segmentación de cauce son dos:

- Superescalares, mediante el hardware el procesador se encarga de ordenar instrucciones, renombrar operandos, predecir saltos... para el mejor uso posible del cauce y obtener el máximo IPC posible para la arquitectura.
- Very Long Instruction Word VLIW, mediante el compilador se ordenan las instrucciones para evitar dependencias y mejor uso del cauce. Dado que en el momento de compilar no contamos con toda la información del procesamiento de las instrucciones, se han desarrollado múltiples estrategias para aumentar el rendimiento, mayormente basadas en el procesamiento especulativo.

2. Procesadores superescalares

Un procesador superescalar es un procesador segmentado que puede finalizar más de una instrucción por ciclo mediante recursos hardware para extraer el paralelismo entre instrucciones dinámicamente.



Todas las etapas de un procesador superescalar pueden procesar varias instr por ciclo.

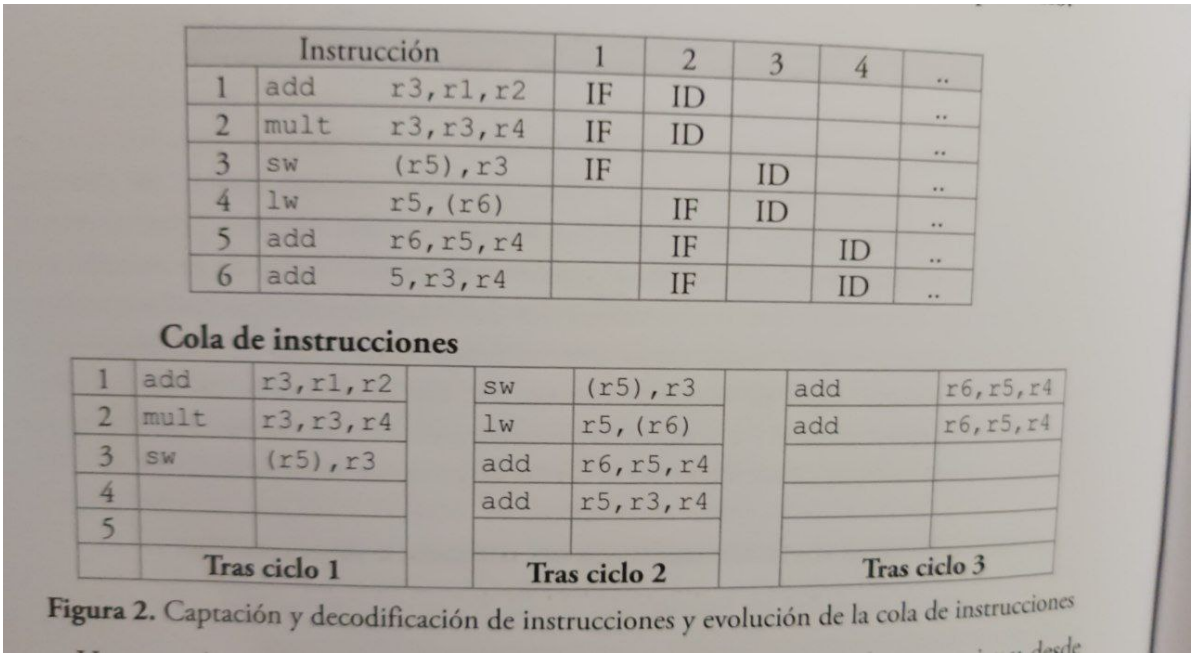
2.1. Procesamiento de instr en un cauce superescalar

Para explicar las distintas etapas vamos a basarnos en la siguiente tabla de ejemplo.

Instrucción	Significado	Ciclos para ejecución EX
1. add r3, r1, r2	r3=r1+r2	1
2. mul r3, r3, r4	r3=r3*r4	3
3. sw (r5), r3	M(r5)=r3	1
4. lw r5, (r6)	r5=M(r6)	2
5. add r6, r5, r4	r6=r5+r4	1
6. add r5, r3, r4	r5=r3+r4	1

- **Precaptación**: añade algunos bits a instrucciones cuando pasan desde la memoria o niveles de caché inferiores (L2 L3...) a la memoria caché de primer nivel L1. Con ello aceleramos la decodificación o algunas instr complejas empiezan a ser procesadas desde su captación.
- **Captación** IF: toma instr de la caché y las pasa a la ***cola de instr***. Estas se captan en el orden que traen desde memoria (el que tienen en el código).
- **Decodificación** ID: desde la cola de instr se decodifican en orden. De ahí pasan a la ***ventana de instr/estación de reserva***.

  En la figura siguiente vemos el progreso de las instr de la tabla hasta ahora durante los cuatro primeros ciclos.



	Instrucción		1	2	3	4	..
1	add	r3,r1,r2	IF	ID			..
2	mult	r3,r3,r4	IF	ID			..
3	sw	(r5),r3	IF		ID		..
4	lw	r5,(r6)		IF	ID		..
5	add	r6,r5,r4		IF		ID	..
6	add	5,r3,r4		IF		ID	..

Cola de instrucciones

1	add	r3,r1,r2		sw	(r5),r3		add	r6,r5,r4
2	mult	r3,r3,r4		lw	r5,(r6)		add	r6,r5,r4
3	sw	(r5),r3		add	r6,r5,r4			
4				add	r5,r3,r4			
5								
	Tras ciclo 1			Tras ciclo 2			Tras ciclo 3	

Figura 2. Captación y decodificación de instrucciones y evolución de la cola de instrucciones

Las instr se emitirán desde la ventana de instr a sus unidades funcionales correspondientes, donde se ejecutarán si sus operandos están disponibles y la unidad funcional está ociosa.

Ventana de instrucciones

Cada línea de la ventana de inst tiene varios campos: el código de la operación (del que se saca la unidad funcional que le corresponde), el destino en el que se almacenará el resultado y dos campos por operando, que dependen de sus respectivos campos "OK": el campo operando contendrá el valor correcto si el campo OK está a 1. Si no, contendrá el sitio al que acceder para hacerlo cuando el campo OK deje de ser 0.

Vamos a ver la ventana de instrucciones para la secuencia de instrucciones planteada en la primera tabla:

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	add	r3	[r1]	1	[r2]	1
2	mult	r3	r3	0	[r4]	1
...						

Esta tabla de la ventana de instrucciones corresponde al final del ciclo 2 en la figura superior. Observamos como el OK1 de la operación mult está a 0, por que el r3 no será válido hasta que no se ejecute la suma previa.

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	mult	r3	[r3]	1	[r4]	1
2	sw	*no hay destino pq va a memoria*	[r5]	1	r3	0
3	lw	r5	[r6]	1	*solo 1 operando*	/
	...					

La ventana de instrucciones tras el ciclo 3. La instrucción de suma ha pasado a ejecución porque sus operadores y su unidad funcional estaban disponibles, así que se elimina y la multiplicación queda la primera de la cola. La operación de suma tarda un ciclo, así que hasta el final del ciclo 3 no conoceremos el resultado de r3. La multiplicación se ejecutará en el ciclo 4 porque tendrá todos sus operadores disponibles.

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	sw	*no hay destino pq va a memoria*	[r5]	1	r3	0
2	lw	r5	[r6]	1	*solo 1 operando*	/
3	add	r6	r5	0	[r4]	1
4	add	r5	r3	0	[r4]	1

La ventana de instrucciones tras el ciclo 4. Mult se está ejecutando y sabremos su resultado al final de este mismo ciclo, se han intruducido las dos últimas sumas (que antes se estaban descodificando).

Emisión ordenada/desordenada

Como vemos, las instr se ejecutan en orden, es una **emisión ordenada**. Es por ello que la instr de carga lw no se ejecuta, incluso si sus operandos y unidad están disponibles. Si se permitiera que lw se ejecutara, se trataría de emisión desordenada. Hay que destacar un supuesto:

- sw escribe en [r5]
- lw lee de [r6]
- Si sucede que [r5] = [r6], la instr lw debería esperar a sw, porque si no se cometería una dependencia tipo RAW.
    - Esto puede evitarse bien porque se conoce lo que [r5] y [r6] o bien asumiendo que la posibilidad de que [r5] = [r6] es baja, en cuyo caso estaríamos cayendo en un **adelantamiento especulativo**.
        - En este último caso, es obvio que el sistema debe contar con recursos para deshacer el efecto de la instr lw adelantada, por si se diera el caso de que,

finalmente, las direcciones sí son iguales. Lo describiremos con más detalle más adelante.

Estaciones de reserva

Hasta ahora hemos hablado de ventana de instrucciones, pero lo más común es contar con las llamadas **estaciones de reserva**: cumplen la misma función, pero dividiendo el contenido de la ventana en estructuras más pequeñas. Lo más común es contar con estaciones de reserva según la unidad funcional a la que vamos a mandar la instrucción. Tras la decodificación, la instr se manda a la reserva correspondiente. Es frecuente un esquema como:

- Dos estaciones de reserva para el procesador: uno para las operaciones enteras y otro para las operaciones en coma flotante
- Dos estaciones de reserva para la memoria: una para la lectura de datos desde memoria y otra para la escritura

De hecho, la ventana de instrucciones corresponde al caso de **una única estación de reserva centralizada**.

Cuando tenemos un esquema basado en múltiples estaciones de reserva, a la decodificación ID hay que añadirle un paso extra, será una operación de decodificación y emisión (issue), ID/ISS. El inicio de ejecución desde una reserva se denomina "envío" o *dispatch*.

Ejemplo - emisión ordenada/desordenada

Volviendo a la tabla de ejemplo, estas son las trazas de una emisión ordenada y una emisión desordenada. Asumimos que las unidades funcionales siempre están disponibles y que hay suficientes para una ejecución sin colisiones (hay dos sumadores, pero solo una ud. de acceso a memoria).

Emisión ordenada:

instr	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**...**
1. add r3, r1, r2	IF	ID	EX								
2. mul r3, r3, r4	IF	ID		EX	EX	EX					
3. sw (r5), r3	IF		ID				EX				
4. lw r5, (r6)		IF	ID					EX	EX		
5. add r6, r5, r4		IF		ID						EX	
6. add r5, r3, r4		IF		ID						EX	

Emisión desordenada: cambios en negrita

instr	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**...**
1. add r3, r1, r2	IF	ID	EX								
2. mul r3, r3, r4	IF	ID		EX	EX	EX					
3. sw (r5), r3	IF		ID				**EX**				
4. lw r5, (r6)		IF	ID	**EX**	**EX**						

instr	1	2	3	4	5	6	7	8	9	10	...
5. add r6, r5, r4		IF		ID		**EX**					
6. add r5, r3, r4		IF		ID			**EX**				

Dependencias, buffer de renombramiento

Como vemos, en la emisión ordenada la ejecución de una instr comienza, como muy pronto, simultáneamente a la finalización de una instrucción que la precede en orden. La emisión desordenada termina en menos ciclos, pero para permitir estos adelantos el procesador ha seguido algunas estrategias:

- Primero, las dependencias de datos RAW (Read after write, uns instr necesita un operando después de que este sea generado). Las instr 2,1; 3,2; y 5,4 la presentan. Se soluciona como ya hemos comentado con el campo OK.
- Las dependencias WAR y WAW no son realmente dependencias porque simplemente son resultado de usar el mismo registro para guardar varias cosas a lo largo de la ejecución, podría solucionarse simplemente usando otro registro como destino. Esta técnica se denomina **renombramiento** de registros.
    - El renombramiento puede aplicarlo el compilador o el hardware. Esto es lo usual en el caso de procesadores superescalares, donde se incluye una estructura de **buffers de renombramiento**. Por ejemplo:
        - Entrada válida, si está a 1 marca que en esa entrada se está utilizando para hacer un renombrado
        - Registro de destino indica el registro que se ha renombrado en esa línea
        - Valor es el espacio reservado para el renombrado, la nueva ubicación
        - Valor válido indica si lo guardado en el nuevo valor es válido
        - Último indica si en la línea está contenido el nuevo renombramiento que se ha hecho para el registro

      El buffer de renombramiento se consulta al captar los operandos para pasarlos a la ventana de instr/estación de reserva: antes de comprobar si los operandos están en el banco de registros, se comprueba si se ha hecho algún renombramiento en el buffer de renom. Si no hay renom del reg, el operando se toma del banco de registros.

#	Entrada válida	Reg destino	Valor	Valor válido	Último
1	1	r3	351	1	0
2	1	r3	-	0	1
3	1	r5	-	0	1
4					

"Recordatorio" de la traza de la ejecución, para estudiar más cómodamente su interacción con el buffer de renom:

instr	1	2	3	4	5	6	7	8	9	10	...
1. add r3, r1, r2	IF	ID	EX								

instr	1	2	3	4	5	6	7	8	9	10	...
2. mul r3, r3, r4	IF	ID		EX	EX	EX					
3. sw (r5), r3	IF		ID				EX				
4. lw r5, (r6)		IF	ID					EX	EX		
5. add r6, r5, r4		IF		ID						EX	
6. add r5, r3, r4		IF		ID						EX	

Se muestra el estado del buffer de renom tras el ciclo 3. Al inicio de la ejecución se encontraba vacío:

1. La instr 1 escribe sobre r3 y, al estar vacío el buffer de renom, toma el r2 y r1 desde el banco de registros. Al final del ciclo 3, se ha hecho la suma y ha dado (por ejemplo) 351.
2. En valor metemos 351, y ponemos valor válido a 1.
3. La inst2 termina de decod en el ciclo 2 y da lugar a un renombramiento de r3 (línea 2 del buffer renom). Como es el último renom a r3, cambiamos Último de la línea 2 a 1, y el antiguo de la línea 1 se pone a 0. El valor de r4 se toma del banco de regs, y como el valor de r3 no está disponible aún, la instr mul se quedará en la ventana de instr.
4. Cuando el campo OK de la ventana de regs de la operación mul pase a ser 1, podemos continuar con la operación. Esto pasa al final del ciclo 3.
5. Al final del ciclo 3 se han descod las instr 3 y 4 - La instr 3 sw no dará lugar a renom porque escribe directamente en memoria, y como r5 no se había usado antes se toma desde el banco de registros. Pero r3 está ahora mismo en la línea 2 del buffer de renom, y no está marcado como válido. Hasta que no termine la multiplicación iniciada antes, no tenemos ese valor.
6. La instr 4 lw accede a r6 desde banco de registros, y da lugar a renom de r5. Se añade al buffer. Si la emisión es desordenada, se podría emitir la instr.

Modelo de consistencia, ROB

Quedaría por hablar de la forma en la que, cuando una inst termina su ejecución, actualiza con sus resultados el banco de registros. Existen dos alternativas para el **modelo de consistencia del procesador**:

- Modelo con finalización desordenada: al igual que las instr terminan de ejecutarse de forma desordenada (porque unas tienen más ciclos que otras, independientemente de su orden en el programa), su resultado se escribe en el banco de regs desordenandamente.
- Modelo con finalización ordenada: actualiza con los resultados el banco de regs en el orden en el que las instr aparecen en el programa, independientemente de cuándo se generen.

- Este modelo es más común en procesadores modernos, porque se ha demostrado que **la mejor forma de aprovechar el paralelismo** es emisión desordenada (según estén disponibles las instr) y almacenado de resultados en orden.

  El modelo ordenado se puede implementar gracias al **buffer de reordenamiento ROB**, una estructura relativamente sencilla y versátil, pero un poco exigente con los recursos hardware. El ROB se puede usar, además de para finalización ordenada, como buffer de renom.

El ROB es una cola circular (la línea 1 va después de la 10, y la 10 va antes de la 1) con dos punteros marcados con **: el de la línea 9 es donde se escribe la siguiente instr que le llegue desde decod, y el de la línea 3 es a la que le corresponde escribir su resultado en el banco de registros.

Las instr se escriben ordenadamente en el ROB desde ID o ID/ISS y también se retiran ordenadamente tras escribir sus resultados en los registros. Se podrían introducir tantas IPC como instr se decodifiquen en el susodicho ciclo. El número de registros que se pueden escribir cada vez es un parámetro dado por el procesador, y nos da una idea de su velocidad pico:

> El número de instr que pueden retirarse por ciclo * freq reloj = num max de instr que se pueden procesar por ud. de tiempo

Si se llena el ROB, se detienen las decodificaciones hasta que queden líneas libres.

El significado de los campos es:

- CodOP: código de la operación
- RegDest: registro del banco de regs donde vamos a escribir el resultado
- Valor: campo donde se almacena temporalmente el resultado, hasta que lo podamos escribir y se retire
- OK: indica si el valor almacenado en Valor es válido
- Unidad: unidad funcional necesaria
- Pred: indica si se introdujo la línea al ROB como resultado de una predicción
- Flush: indica si el resultado se ha retirado del ROB sin almacenar el resultado en el registro
- Estado: f-> finalizado, x-> ejecutándose, i-> estación de reserva y esperando para su ejecución

Para que una instr pueda retirarse, tanto la propia instr como las de las lineas previas a ésta deben tener el campo OK a 1 (y estado F) ; o el campo flush a 1.

Ejemplo - ROB

No vamos a tener en cuenta los campos pred y flush porque pertenecen más bien al ámbito especulativo y eso lo vamos a ver más adelante.

#	Cod Op	Reg destino	Valor	OK	Unidad	Pred	Flush	Estado
1								
2								
3**	add	r3	351	1		0	0	f
4	mult	r3	--	0	mult	0	0	x
5		[no hay, va		0		0	0	i

#	Cod Op	Reg destino	Valor	OK	Unidad	Pred	Flush	Estado
5	sw	a mem]	--	0		0	0	i
6	lw	r5	--	0	carga(load)	0	0	x
7	add	r6	--	0		0	0	i
8	add	r5	--	0		0	0	i
9**								
10								

Este es el buffer ROB tras el ciclo 4 de la traza de emisión desordenada, la añado aquí abajo otra vez para más facilidad:

instr	1	2	3	4	5	6	7	8	9	10	...
1. add r3, r1, r2	*IF*	*ID*	*EX*								
2. mul r3, r3, r4	*IF*	*ID*		EX	EX	EX					
3. sw (r5), r3	*IF*		*ID*				**EX**				
4. lw r5, (r6)		*IF*	*ID*	**EX**	**EX**						
5. add r6, r5, r4		*IF*		ID		**EX**					
6. add r5, r3, r4		*IF*		ID			**EX**				

Al finalizar el ciclo 4, la instrucción add (línea 3 del ROB) escribirá su resultado en Valor, y podrá retirarse al siguiente ciclo.

La instr mul ha empezado a ejecutarse y está en la línea 4 del ROB. La inst sw está en la línea 5 del ROB y está esperando en la ventana de inst/estación de reserva al resultado de la multiplicación.

La instr de lectura lw como sabemos puede adelantar a la de almacenamiento sw, y se encuentra en la línea 6 del ROB.

Las dos sumas finales están esperando en la ventana a recibir info sobre sus operandos, y por tanto en el ROB aún no tienen marcada la unidad que van a utilizar cuando se ejecuten.

#	Cod op	Reg dest	Oper 1	OK1	Oper2	OK2
1	sw	*no hay destino pq va a memoria*	[r5]	1	4	0
3	add	7	6	0	[r4]	1
4	add	8	4	0	[r4]	1
...	...					

Ventana de instr al final del ciclo 4, el mismo momento en el que hemos visto el ROB un par de tablas más arriba: en reg dest de las operaciones de suma **se indican las líneas que les corresponden en el ROB**. En los operandos cuyo OK es 0, se indica **la línea del ROB que corresponde a las instrucciones que producirán esos resultados**.

instr	1	2	3	4	5	6	7	8	9	10	...
1. add r3, r1, r2	IF	ID	EX	**ROB**	**WB**						
2. mul r3, r3, r4	IF	ID		EX	EX	EX	**ROB**	**WB**			
3. sw (r5), r3	IF		ID				EX	**WB**			
4. lw r5, (r6)		IF	ID	EX	EX	**ROB**		**WB**			
5. add r6, r5, r4		IF		ID		EX	**ROB**		**WB**		
6. add r5, r3, r4		IF		ID			EX	**ROB**	**WB**		

Traza de emisión desordenada, con las etapas de ROB (escritura en ROB en su correspondiente línea del resultado generado por la ejecución) y WB (retirada del resultado almacenado en ROB.Valor para escribirlo en el banco de registros).

Sw no necesita pasar por ROB porque escribe en memoria directamente. Se ha supuesto que se pueden escribir dos resultados en ROB por ciclo, y que se pueden retirar 2 resultados al banco de regs por ciclo. No cuentan las instr de almacenamiento y por eso en el ciclo 8 hay tres WB, porque una es de sw y va a memoria directo sin ayuda del ROB.

Podemos observar que la finalización es ordenada: las instrucciones se retiran WB en 5, 8, 8, 8, 9, 9; aunque hayan terminado de ejecutarse desordenadamente de la forma 3, 6, 7, 4, 5, 7

2.2. Procesamiento de las instrucciones de salto

Los procesadores superescalares son segmentados. Los saltos pueden ser muy dañinos a la ejecución, como cambian el orden de la secuencia de instrucciones puede ser que hayamos malgastado ciclos haciendo instr que no tocaban y encima tenemos que deshacer los efectos de esas instrucciones sobre el estado del procesador.

Además, el hecho de procesar múltiples instr por ciclo sólo hace que aumente este riesgo: más posibilidades de procesar un salto en un ciclo y, por tanto, más instrucciones que potencialmente tenemos que deshacer.

Predicción de saltos, BTB

Es muy fácil lidiar con saltos incondicionales, porque en la etapa de precaptación son marcados y en la etapa de captación IF se empiezan a procesar (calcular la dirección de destino del salto). El problema son los saltos condicionales: aunque se podría hacer lo mismo, hasta que no conozcamos el valor de la condición no sabremos si se va a producir el salto o no. Para

solucionarlo se tiende a la **predicción de saltos**, determinar la alternativa (saltar o no) más probable y continuar el procesamiento con la secuencia de instrucciones más probable. Cuando lleguemos a la condición sabremos si nuestra predicción es correcta o no: si no acertamos, el rendimiento se verá afectado porque tendremos que volver atrás además de deshacer las instr previas. Es decir: la eficacia de este procedimiento dependerá de nuestra precisión a la hora de hacer predicciones.

- Existen procedimientos estáticos basados en la propia instr de salto. Se tiene en cuenta dirección del salto (palante o patrás), la condición (<, >, <=...), si permite bit de predicción... Estas predicciones no tienen en cuenta la dinámica real de ejecución del código
- Los procedimientos dinámicos tienen en cuenta el historial de comportamiento de cada instr de salto condicional para determinar el comportamiento más probable. Para ello se emplea una estructura llamada **BTB (*Branch Target Buffer*)** en la que se introduce información sobre las instr de salto de un programa que se haya ejecutado con anterioridad.

  De esta forma, con la dirección de la instr de salto se almacena además una serie de bits que codifica la historia de ejecución del salto.

Cuando una instr de salto llega al cauce, se comprueba si tiene entrada en BTB. Si no la tiene, se hace una predicción estática y se le asigna una entrada, donde apuntaremos en el campo de bits si la predicción ha acertado o no. Si sí tiene entrada, se realiza la predicción según lo que indique la mayoría de bits y se actualiza con el nuevo resultado. Por ejemplo, una entrada que almacena los 3 últimos intentos que sea 011 quiere decir que ha fallado una vez y acertado dos, así que se hará el salto. Luego se actualizan los bits de la forma X01, donde X es si el salto ha sido correcto o no. Hay ríos de tinta en este tema sobre cuánta memoria conviene gastar en la historia.

La recuperación tras una predicción incorrecta implica volver a la primera instr de la secuencia alternativa de instr y deshacer las instr ejecutadas. La dirección de la instr se puede determinar sin mayor problema si, previa predicción...

- ...en el caso de predecir saltar, guardamos la instr siguiente al salto
- ...en el caso de predecir NO saltar, calculamos la dir de destino del salto

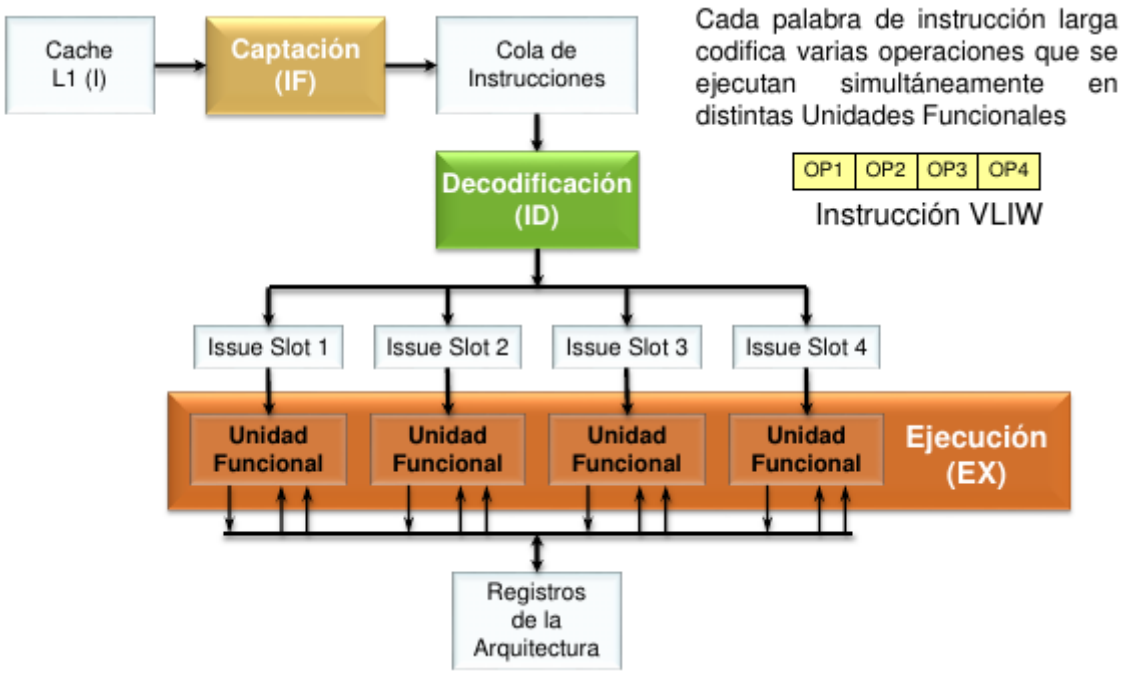
Respecto a deshacer las instr ejecutadas, con el ROB se incluye el campo pred que determina las instr introducidas en el ROB especulativamente, así como el campo flush que permite retirar sin ninguna modificación en el procesador las instr introducidas marcadas con pred.

3. Procesadores VLIW

Un VLIW es un procesador segmentado que puede ejecutar más de una instrucción por cicle mediante agrupaciones de instr que se pueden ejecutar paralelamente generadas por el compilador. Los VLIW pueden ejecutar operaciones empaquetadas gracias al trabajo del compilador, que se ejecutan sin mayores comprobaciones.

Mientras que **los procesadores superescalares planifican dinámicamente, los VLIW lo hacen estáticamente**. Esto contribuye a que la microarquitectura de un VLIW sea **más simple** que la de un superescalar, porque no necesitan estructuras complejas como los buffers o estaciones de reserva. Es por ello que los VLIW se plantearon como una alternativa a los superescalares que consiguiera más IPC con menor consumo y mejor escalabilidad, empleando el aumento de transistores en más unidades funcionales para trabajar en paralelo -- Sin embargo, **para aprovechar realmente el VLIW se necesita encontrar un número suficientemente elevado de operaciones** paralelizables, además de aplicar ciertas estrategias que requieren de ciertos recursos hardware. Estas estrategias son necesarias para compensar el hecho de contar con menos información en el momento de la compilación sobre la ejecución.

También son menos portables, porque requieren de información específica sobre el cauce para el que está generando código (número de uds funcionales, latencia...).



Esquema de cauce de un VLIW:

- La captación IF toma una o varias instr VLIW y las pasa a la cola de instr. Una instr VLIW encompasa varias instr así que aunque "solo sea una" se están ejecutando varias inst por ciclo.
- Las instr VLIW se toman desde la cola y se pasan a la ud de decod ID, donde se determina las operaciones que hay que mandar a cada uno de los slots de emisión. Cada uno de los slots debe poder ejecutarse en alguna unidad.

Por lo tanto, para aprovechar un VLIW es necesario que el compilador encuentre tantas operaciones independientes como slots de emisión haya, y que cada uno de esos slots reciba operaciones que se puedan ejecutar en las unidades a las que puede acceder el slot. Si no se encuentra una operación independiente, se inserta una operación tipo nop (no operar), dando lugar a códigos poco densos que si no tienen alguna estrategia adecuada requerirán más espacio que un programa equivalente en un superescalar.

3.1. Técnicas software para ampliar los bloques básicos

Los riesgos de control asociados a los saltos condicionales dificultan la planificación del código al compilador porque, en tiempo de compilación, no se saben los comportamientos del salto en ejecución. Así que es difícil encontrar operaciones independientes, y cuantos menos saltos haya, mejor.

Se denomina bloque básico al cjto de operaciones situadas entre dos saltos, el compilador debe encontrar la mejor planificación para las operaciones de cada uno de los bloques básicos del programa. Ahora veremos unas técnicas para aumentar el número de instr indep en los bloques básicos: desenrrollado de bucles y segmentación software. Para explicarlas usaremos el siguiente código:

```
inicio:     lw  r1,n;           //carga en r1 el num de iteraciones
            ld  f0,a;           //carga en f0 el valor de a
            add r3,r0,r0;       //hace r3 = 0 (índice del vector)

bucle:      ld  f2,x(r3);       //f2 se carga con el elemento x[i]
            add f4,f2,f0;       //f4 = x[i] + s
            sd  x(r3),f4;       //guardar en x[i] = f4
            subi r1,r1,#1;      //decrementar el número de elementos
            addi r3,r3,#8;      //apuntar al sig elemento del vector
            bnez r1,bucle;      //si quedan elementos, saltar de vuelta a bucle
```

Para construir las operaciones, el compilador debe:

1. Encontrar instr paralelizables
2. Cuántos slots hay
3. Qué tipo de operaciones puede manejar cada slot
4. Retardo de cada operación

Vamos a suponer un procesador con 3 slots, uno a una ALU de operaciones aritmetico-lógicas con enteros, el siguiente a otra ALU pero para float, y uno para accesos a memoria.

Etiqueta	**slot1: op mem**	**slot2: op ALU**	**slot3: op FP**
inicio	ld f0,a	add r3,r0,r0	*nop*
	lw r1,n	*nop*	*nop*
bucle	ld f2,x(r3)	*nop*	*nop*
	nop	subi r1,r1,#1	add f4,f2,f0
	nop	bnez r1,bucle	*nop*
	sd x(r3),f4	addi r3,r3,#8	*nop*

HASTA AQUÍ HEMOS LLEGADO, SI PREGUNTAN MUCHO POR VLIW CAGUÉ 04:27 09-06-2022

3.2. Instrucciones con predicado

3.3. Procesamiento especulativo