ChatGPT

Apuntes Tema 4: Arquitecturas con Paralelismo a Nivel de Instrucción (ILP)

1. Introducción al ILP

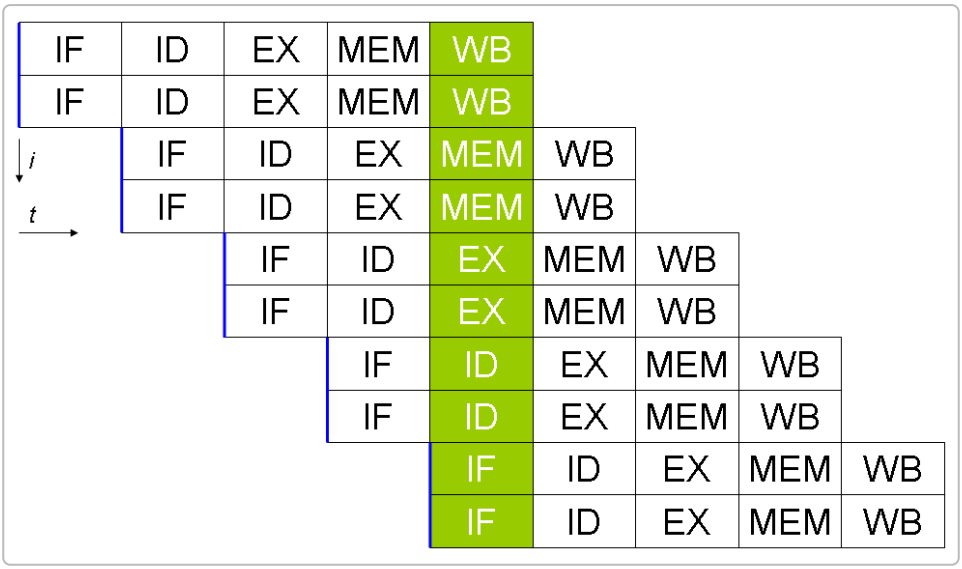
En arquitecturas ILP se busca ejecutar **múltiples instrucciones en paralelo** dentro de un mismo procesador para aumentar el rendimiento. En vez de procesar una instrucción completa antes de comenzar la siguiente, se **fragmenta el flujo** de instrucciones en etapas (IF, ID, EX, MEM, WB) y se superpone su ejecución. Además, en un procesador *superscalar* se instalan **varias unidades funcionales** (ALUs, caches, etc.) para poder emitir/ejecutar varias instrucciones por ciclo. De este modo el rendimiento crece hasta donde lo permiten las dependencias del programa y los recursos hardware. Como analogía, imagina una **línea de montaje** de fábrica: cada pieza (instrucción) pasa por varias etapas (IF, ID...) y se ensamblan varias en paralelo.

Ventajas del ILP/superscalar	**Límite**
Permite completar >1 instr/ciclo, aumentan throughput.	Depende del paralelismo que el código tenga.
Mejor aprovechamiento de unidades (sumadores, mult.).	Aparecen **riesgos** (dependencias) y necesidad de hardware extra (renombrado, ROB).
Puede ejecutarse fuera de orden para evitar stalls.	Complejidad de microarquitectura y consumo energético.

Diferencia con VLIW: En VLIW el compilador agrupa instrucciones *estáticamente* en palabras anchas para emitirlas en paralelo; el hardware es más simple (no revisa dependencias en tiempo de ejecución). En un *superscalar* tradicional el hardware **dinámicamente** selecciona y reordena instrucciones en tiempo real. Por ejemplo:
- *Superscalar:* emisión **dinámica**, detección de dependencias en hardware, buffers de renombrado y ROB complejos.
- *VLIW:* emisión **estática** (el compilador inserta 'nop' si es necesario), hardware más sencillo (no hay estaciones de reserva ni ROB).

2. Cauce superescalar y canalización múltiple

Un **cauce (pipeline)** tradicional de 5 etapas (IF–ID–EX–MEM–WB) permite iniciar una instrucción por ciclo y completarla en 5 ciclos. Un procesador *superescalar* duplica o triplica este cauce: fetch, decode, execute de 2+ instrucciones al mismo tiempo en diferentes unidades funcionales. El diagrama siguiente ilustra un pipeline dual-issue (dos instr. paralelas), donde se muestran dos instrucciones en IF/ID, luego EX/MEM simultáneas, etc. Esto equivale a tener dos "hilos" en el pipeline:



Ejemplo de canalización dual-issue: dos instrucciones se emiten y ejecutan en paralelo. Cada etapa (IF, ID, EX, MEM, WB) puede procesar dos instrucciones simultáneamente si no hay dependencias. El color verde resalta la escritura de resultado (WB).

Algunas ideas clave:
- **Emisión múltiple (issue width):** indica cuántas instrucciones puede iniciar el procesador por ciclo. Ej.: "procesador 2-wide" inicia 2 instr. en cada ciclo.
- **Requisitos:** se necesitan tantas ALUs, caches, puertos de escritura, etc. como ancho de issue. Si faltan unidades, aparecen **riesgos estructurales** (véase más abajo).
- **Paralelismo explotable:** el rendimiento sólo mejora si hay instrucciones independientes que *no* dependan de los resultados entre sí. El compilador e instrucciones "nop" ayudan a llenar huecos, pero el hardware buscará paralelismo dinámicamente en cada ciclo.

Ejemplo simple: un microprocesador *2-issue* puede, en un ciclo, buscar dos instrucciones independientes en secuencia y enviarlas a ejecución simultánea (una quizás a un sumador, otra a un multiplicador). Si no hay suficientes independientes, una unidad se queda sin uso ese ciclo.

3. Tipos de dependencias (riesgos)

Las dependencias de datos y de control limitan el ILP. Se distinguen tres dependencias de datos (hazards) entre instrucciones sucesivas:
- **RAW (Read After Write):** instrucción lee un registro antes de que otra anterior lo escriba. Ej. `I1: R1=…; I2: …=R1`. Es la dependencia *verdadera* (necesita que I1 termine para dar valor).
- **WAR (Write After Read):** antidependencia. I2 escribe un registro que I1 leerá. Ej. `I1: …=R1; I2: R1=…`. Si I2 escribiera primero, I1 leería mal. Se evita fácilmente con renombrado.
- **WAW (Write After Write):** dependencia de salida. Ambas instrucciones escriben el mismo destino. Ej. `I1: R1=…; I2: R1=…`. Se debe garantizar que la última instrucción del orden de programa sea la que escriba al final. También se resuelve renombrando.

Además hay **hazard estructural** si dos instrucciones quieren usar al mismo tiempo el mismo recurso (misma ALU, puerto de memoria, etc.). Y **hazard de control** cuando aparece un salto (branch): el flujo puede cambiar y hay que adivinar adónde ir. Cada tipo causa *stalls* o anulaciones. Se emplean técnicas (renombrado, predicción de saltos, buffers) para mitigar estos riesgos.

Por ejemplo, en un pipeline sin soluciones contra WAR/WAW, si I1 e I2 escriben en el mismo registro, habría que forzar que I2 espere a que I1 retire su resultado antes de emitir I2. Con renombrado, I2 recibe otro registro físico, evitando el stall.

4. Emisión dinámica: Tomasulo y Scoreboarding

Para ejecutar *fuera de orden* (out-of-order) y tolerar latencias, se usan algoritmos de emisión dinámica. Dos métodos clásicos: **Scoreboarding** (como en CDC 6600) y **Tomasulo** (IBM 360/91). En ambos casos se comprueba cada ciclo qué instrucciones pueden emitir, ejecutar o escribir resultados respetando dependencias.

Tomasulo: muy popular en la práctica moderna. Utiliza:
- **Estaciones de Reserva (Reservation Stations):** cada unidad funcional (ALU, FP, load/store) tiene varias estaciones de reserva donde esperar operandos.
- **Bus de Datos Común (CDB):** canal que difunde resultados cuando están listos para que otras estaciones los capten.
- **Tabla de Estado de Registros:** indica qué estación producirá el valor de cada registro (para renombrado implícito).

Flujo de Tomasulo (simplificado):
1. **Issue (emisión):** Se lee la siguiente instrucción en orden de programa. Si hay estación y entrada en ROB libres, se asigna a una estación de reserva. Los operandos ya listos se copian; los no listos se marcan con una etiqueta (p.ej., apuntan a la estación que los producirá). El registro destino se "renombra" apuntando a esta estación (evita WAR/WAW).
2. **Execute (ejecución):** Una vez que una instrucción en una estación tiene todos sus operandos, la unidad funcional ejecuta la operación. Mientras tanto otras instrucciones pueden emitirse o esperar.
3. **Write result (escritura):** Al terminar, la estación transmite el resultado por el CDB: las estaciones que esperaban este operando lo reciben, y se guarda el resultado en un buffer intermedio (p.ej., ROB). Las entradas en tabla de registros se actualizan para indicar que el valor está listo.
4. **Commit (retirar):** Las instrucciones se comprometen en orden de programa desde el ROB (ver siguiente sección). Se escriben realmente los resultados en el registro arquitectónico y se libera la estación de reserva. Esto asegura **precisión de estado** (precise exception).

> **Analogía:** Tomasulo es como un gestor de tráfico. Cada instrucción es un vehículo que entra, espera en un carril (estación) hasta que están listos todos los pasajeros (operandos), sigue la vía (ejecución) y entrega su carga (resultado) por un depósito común (CDB). El tránsito se reordena y controla para maximizar flujo sin accidentes (hazards).

Scoreboarding difiere en que mantiene un único registro de dependencias y decide en qué orden emitir/ejecutar; suele requerir más stalls estáticos. Tomasulo, al renombrar registros, elimina prácticamente WAR y WAW automáticamente y resuelve RAW esperando en las estaciones.

Resumen de Tomasulo: (pasos clave)
- *Registro de estaciones:* cada una sabe los operandos necesarios (o etiquetas de quien los producirá).
- *Renombrado:* al emitir, el destino recibe nombre de estación; elimina conflictos WAR/WAW.
- *CDB:* motor de difusión, cada resultado va a todas las estaciones que lo necesitan.
- *Precisión:* se conserva haciendo *commit* en orden.

5. Renombrado de Registros (Register Renaming)

El renombrado es la base del out-of-order moderno. Consiste en traducir los registros arquitectónicos (ej. R1, R2...) a **registros físicos adicionales**. Cada vez que se emite una instrucción con destino en R, se le asigna un nuevo registro físico libre. Los siguientes usos de R usan ese nuevo nombre. Esto rompe las dependencias falsas WAR/WAW.

- **Ejemplo:**

```
I1: R3 = R3 / R5
I2: R4 = R3 + 1
I3: R3 = R5 + 1
I4: R7 = R3 * R4
```

  Sin renombrado, hay WAR entre I1-I2 (I2 lee R3 antes de que I1 lo escriba) y WAW entre I1-I3 (ambos escriben R3). Con renombrado, supongamos que I1 libera un físico p3→R3, luego I2 lee p3, I3 asigna un **nuevo** pX→R3, y I4 leería los valores apropiados. Así garantizamos que sólo las dependencias *verdaderas* (RAW) quedan.

- **Mecanismo:** Un *map table* traduce el registro arquitectónico (clave) a un registro físico o estación de reserva. Tras el commit, se actualiza el registro real. Se usan bancos o buffers especiales ("archivo de renombrado") donde cada escritura tiene su copia.

- **Ventajas:** elimina WAR/WAW permitiendo emitir muchas instrucciones consecutivas incluso si usan el mismo registro destino originalmente. Mejora muchísimo el paralelismo. La complejidad está en la lógica extra, pero procesadores modernos (Intel, ARM, etc.) emplean renombrado en hardware.

6. Buffer de Reordenamiento (ROB) y coherencia de procesador

El **ROB (Reorder Buffer)** es una estructura donde se anotan instrucciones emitidas para asegurar su compromiso in-order. Sus objetivos:
- **Precisión de estado:** mantiene resultados hasta que se retire cada instrucción en orden; si hay un fallo (ej. excepciones, predicción de salto errada) se pueden descartar las instrucciones posteriores aún pendientes.
- **Commit in-order:** aunque se ejecute fuera de orden, las escrituras finales al registro/memoria se hacen secuencialmente según el programa original.

Funcionamiento básico:
- Al emitir una instrucción, se le asigna una entrada en el ROB (FIFO circular). Se almacena información como opcode, registro destino, orden de programa, etc.
- Cuando la instrucción termina de ejecutarse, en lugar de escribir directamente, su resultado se graba en la entrada del ROB (se marca como 'listo').
- El ROB lleva un *head* apuntando a la instrucción más vieja pendiente. Si está completada (ready), se **retira** (commit): escribe el valor en el registro arquitectónico o memoria y se libera la entrada. Luego el head avanza a la siguiente.

Esto hace que, aunque varias instrucciones acaben desordenadamente, sólo efectúen cambios visibles al programa en orden correcto. También permite descartar las que vienen después del punto de falla (speculation). Además, el ROB actúa como buffer de nivel temporal.

Ejemplo ilustrativo:
Supongamos las instrucciones (en orden):

```
I1: R1 = …
I2: R2 = … (depende de I1)
I3: R3 = …
```

I3 puede terminar antes que I2, pero no se compromete hasta que I1 e I2 hayan sido retiradas. El ROB garantiza que los efectos de I3 se apliquen después. Si al final I1 fallara (por error de predicción), las instrucciones I2, I3 (y otras especulativas) se pueden invalidar quitándolas del ROB antes de commit.

Además de registros, el ROB puede contener buffers de direcciones para tiendas de memoria. Todo ello coopera con el renombrado para implementar la **consistencia secuencial** aparente del procesador (cada instrucción parece ejecutarse en orden a pesar del paralelismo interno).

7. Procesamiento de saltos (branch processing)

Los saltos condicionales (branch) interrumpen la canalización normal porque no se sabe en una etapa temprana qué instrucción cargar después. Para mitigarlo:
- **Predicción de saltos (static/dinámica):** intentar adivinar la dirección de la próxima instrucción. Ejemplos:
- *Estática:* siempre tomados, o “tomado si negativo”, o análisis en tiempo de compilación.
- *Dinámica:* histórico de branch predictors. Por ejemplo, contador de 2 bits por instrucción en una tabla (BHT). Permite predecir ramas basadas en comportamientos anteriores. Un predictor común es el **Bimodal** o esquemas de dos niveles. También hay **racimos de retorno (return stack)** para instrucciones RET.
- **Ejecución especulativa:** el procesador puede ejecutar *por adelantado* las instrucciones predichas. Si la predicción fue correcta, se gana tiempo; si fue incorrecta, se descartan (flush) las instrucciones especulativas y se recarga la tubería con la ruta correcta, perdiendo algunos ciclos.

- **Slot de retraso (delay slot):** en arquitecturas RISC antiguas, tras un salto se incluía una instrucción “inofensiva” (nop) o utilizable que siempre se ejecutaba. Actualmente, es más común confiar en predicción dinámica.

En resumen, un buen predictor de saltos (y suficiente hardware especulativo) reduce la penalización de saltos (que de otra forma causa burbujas en el pipeline). Por ejemplo, sin predicción se puede detener el fetch hasta saber la condición; con predicción se continúa llenando el pipeline basándose en la conjetura.

8. Resumen de conceptos clave

- **Superscalar vs VLIW:** Superscalar: *issue* dinámico en HW; VLIW: empaca instrucciones en compilación.
- **Dependencias:** RAW (**verdadera**), WAR (antidependencia), WAW (depend. de salida). WAR/WAW se eliminan renombrando registros.

- **Renombrado:** asignar nuevos nombres físicos a cada escritura arquitectónica, evitando conflictos de nombres.
- **Buffer de reorden:** commit in-order, preciso. Entrada FIFO con resultados listos y orden de programa.
- **Tomasulo:** algoritmos con estaciones de reserva y bus común para scheduling dinámico. Permite OoO mientras manteniendo orden lógico.
- **Predicción de saltos:** necesaria para mantener el pipeline lleno. Puede ser de 1 bit (predicción simple) o 2 bits (estadística moderada), etc.

Tabla comparativa: Superscalar vs VLIW

Característica	Superscalar	VLIW (Instrucciones de longitud fija)
Emisión	Dinámica (HW detecta paralelismo)	Estática (compilador organiza bundling)
Dependencias	HW comprueba en tiempo real	Compilador elimina dependencias (nofect2)
Hardware (complejidad)	Alta (estaciones, ROB, predicción)	Menor (no stations, unida funcional simples)
Número de unidades	Varias unit. independ. (ALU, FP...)	Varias unidades (como superscalar)
Código máquina	Variable por ciclo	Agrupa instrucciones en paquetes fijos
Robustez con cambios futuros	Mayor (HW se adapta a código)	Menor (código ligado a hardware específico)

9. Ejercicios típicos con solución paso a paso

Ejemplo 1: Diagrama de canalización con dependencias.
Dado el siguiente fragmento de código RISC (sin forwarding):

```
1: lw r3, 0x10(r2)   // carga en r3
2: addi r2, r2, #5   // r2 = r2 + 5
3: mul r1, r3, r4    // multiplica usando r3
4: add r5, r3, r2    // suma r3 con r2
5: sw 0x00(r2), r5   // almacena r5 en [r2+0]
```

Asumimos un pipeline de 5 etapas (IF, ID, EX, MEM, WB) que puede emitir 1 instrucción por ciclo. Describimos el timeline por ciclos:

Instr	1	2	3	4	5	6	7	8	9
lw r3...	IF	ID	EX	MEM	WB				
addi r2...		IF	ID	EX	MEM	WB			
mul r1...			IF	ID	—	EX	MEM	WB	

Instr	1	2	3	4	5	6	7	8	9
add r5...				IF	ID	EX	MEM	WB	
sw r5...					IF	ID	EX	MEM	WB

- En este caso hay **stall** en el ciclo 5 para `mul r1,…`: el pipeline introduce un hueco (—) en EX, porque `mul r1` depende del resultado de `lw r3` (RAW), que termina en WB ciclo 5. Hasta que r3 esté listo, `mul` no puede EX.
- Posteriormente, la `add r5` también depende de r3 y r2, pero r3 ya está lista en ciclo 5 y r2 se actualiza en ciclo 6 (de `addi`), por lo que no hay stalls adicionales.

Si mejoramos la latencia de `mul` (por ejemplo, pasa a tardar 3 ciclos EX en vez de 6), todo el lote final puede terminar **3 ciclos antes**: las instrucciones posteriores ya no estarían esperando tanto. En resumen, el cuello de botella era el `mul` lento; al acelerarlo, el tiempo de ejecución cae en la misma magnitud.

Ejemplo 2: Renombrado de registros.
Considera las dependencias WAR/WAW:

```
I1: R3 = R3 / R5
I2: R4 = R3 + 1
I3: R3 = R5 + 1
I4: R7 = R3 * R4
```

– *Sin renombrado:* I1 e I3 escriben en R3 (WAW), I2 lee R3 antes de I3 escribir (WAR). Esto fuerza empalmes estrictos.
– *Con renombrado:* Asignamos nuevos registros físicos p1, p2 para las escrituras: I1 escribe p1, I2 lee p1, I3 escribe p2 (independiente de p1), y I4 usará el valor correcto (p2) con el R4 calculado. Así **no hay peligro WAR/WAW** y se puede emitir más paralelismo.

Ejemplo 3: Tomasulo paso a paso.
Imaginemos 3 instrucciones FP con dependencias:

```
I1: F1 = F2 + F3
I2: F4 = F1 * F5
I3: F6 = F2 - F3
```

1. *Emisión:*
- I1 se asigna a RS1 (p.ej. estación de reserva de ALU), F2 y F3 están listos, renombra destino F1→RS1.
- I2 ve que F1 aún no está listo (sale de I1), y F5 sí. Se asigna a RS2 (estación FMUL), se marca que le falta F1 (etiqueta RS1). Renombra F4→RS2.
- I3 no depende de F1/F5; se asigna a RS3, F2,F3 ya listos, renombra F6→RS3.
2. *Ejecución:* I1 e I3 tienen operandos disponibles, asumen distintas UFs, ejecutan. I2 espera a que RS1 produzca F1.
3. *Escritura:* I1 e I3 terminan (quizás I3 más rápido si resta simple), envían resultados por CDB: F1 por RS1 y F6 por RS3.
- Al llegar a RS2 (I2), I2 recibe F1 en la estación, ahora tiene todos sus operandos (F1 y F5). Ejecuta.

4. *Commit:* I1 sale del ROB primero (escrito F1→reg. físico), luego I3 (F6), luego I2 (F4) según orden programa.

Este proceso ilustra cómo Tomasulo reorganiza la ejecución (I3 se puede completar antes de I2 aunque I2 se emitió segundo) y cómo renombra elimina conflictos.

Ejemplo 4: Predicción de saltos simple.
Secuencia con salto condicional:

```
100: BEQ R1, R2, L1
101: ADD R3, R4, R5
102: SUB R6, R7, R8
103: …
L1: MUL R9, R10, R11
```

Sin predicción, después de IF de la BEQ se detiene el fetch hasta saber la comparación (2 ciclos), introduciendo stalls. Con predicción (e.g. *no tomado* por defecto) el procesador *especulativamente* ejecuta ADD y SUB. Si al final la BEQ resulta *no tomada*, no pasa nada y ganamos ciclos. Si en cambio era tomada, se descartan ADD/SUB y se corrige cargando `MUL` en la tubería, con una penalización de varios ciclos. Un predictor más sofisticado (basado en historial) reduciría los errores de predicción y las pérdidas de rendimiento.

10. Resumen Visual y Comparativo

- Las **arquitecturas ILP** explotan paralelismo dividiendo el trabajo en etapas superpuestas (pipelining) y/o emitiendo múltiples instrucciones.
- Los **riesgos de datos** obligan a stalls o a usar técnicas de renombrado y buffering.
- Las **emisión dinámicas** (Tomasulo/ROB) permiten *tomar decisiones en hardware* para maximizar paralelismo conservando la coherencia del programa.
- Las **unidades de proceso superscalars** suelen incluir: cola de instrucciones, buffers de renombrado, estaciones de reserva y ROB, junto a unidades funcionales múltiples.

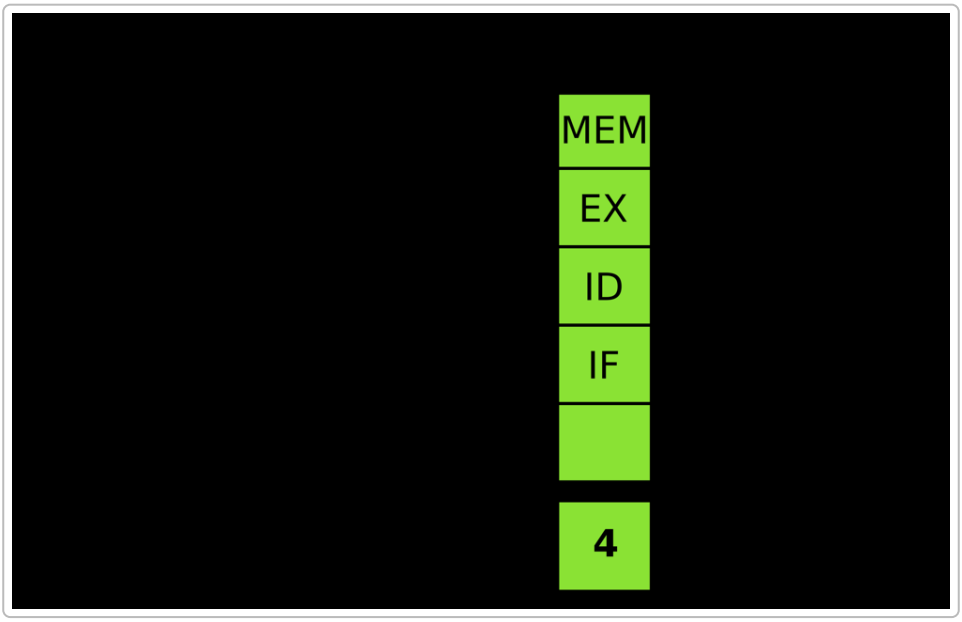


Diagrama resumido de canalización clásica de 5 etapas. En cada ciclo la primera instrucción (en azul) avanza un paso y la siguiente (naranja) ocupa la etapa anterior. Con múltiples unidades (superscalar) se podrían procesar varias líneas como ésta simultáneamente, evitando burbujas.

Para facilitar el estudio, recordemos con una tabla simplificada:

Concepto	Modo In-order (convencional)	Out-of-Order (Tomasulo, ROB)
Emisión	Una instrucción por ciclo (hasta hazards)	Múltiples/instrucciones emitidas si hay recursos
Dependencias RAW	Stall hasta que valor listo	Stall en estación hasta operandos disponibles
Dependencias WAR/ WAW	Stall obligatorio	Renombrado elimina el conflicto
Orden de Commit	Suelen retirarse fuera de orden (sin ROB)	Siempre in-order gracias al ROB
Implementación	Pipeline simple, sin registers extra	Pipeline complejo, reg. físicos extra, CDB, ROB

Estos apuntes condensan los principales conceptos del tema 4. Es útil repasarlos revisando ejemplos de pipeline, escribiendo cruces de dependencias y dibujando timelines. Un **resumen visual final** podría incluir diagramas esquemáticos de un canalizador superscalar (como los mostrados) y tablas comparativas de dependencias y técnicas, a modo de chuleta. Con estos apuntes claros y estructurados, el estudio del paralelismo a nivel de instrucción será más accesible.
