Si un pipeline es bueno, entonces dos es mejor

P.ej., la unidad de *instruction fetch* lee dos instrucciones al mismo tiempo

… y coloca cada una es su propio pipeline —***two-issue***:

- las instrucciones no deben tener conflictos con respecto al uso de recursos —no debe haber hazard estructural
- ninguna instrucción debe depender del resultado de la otra

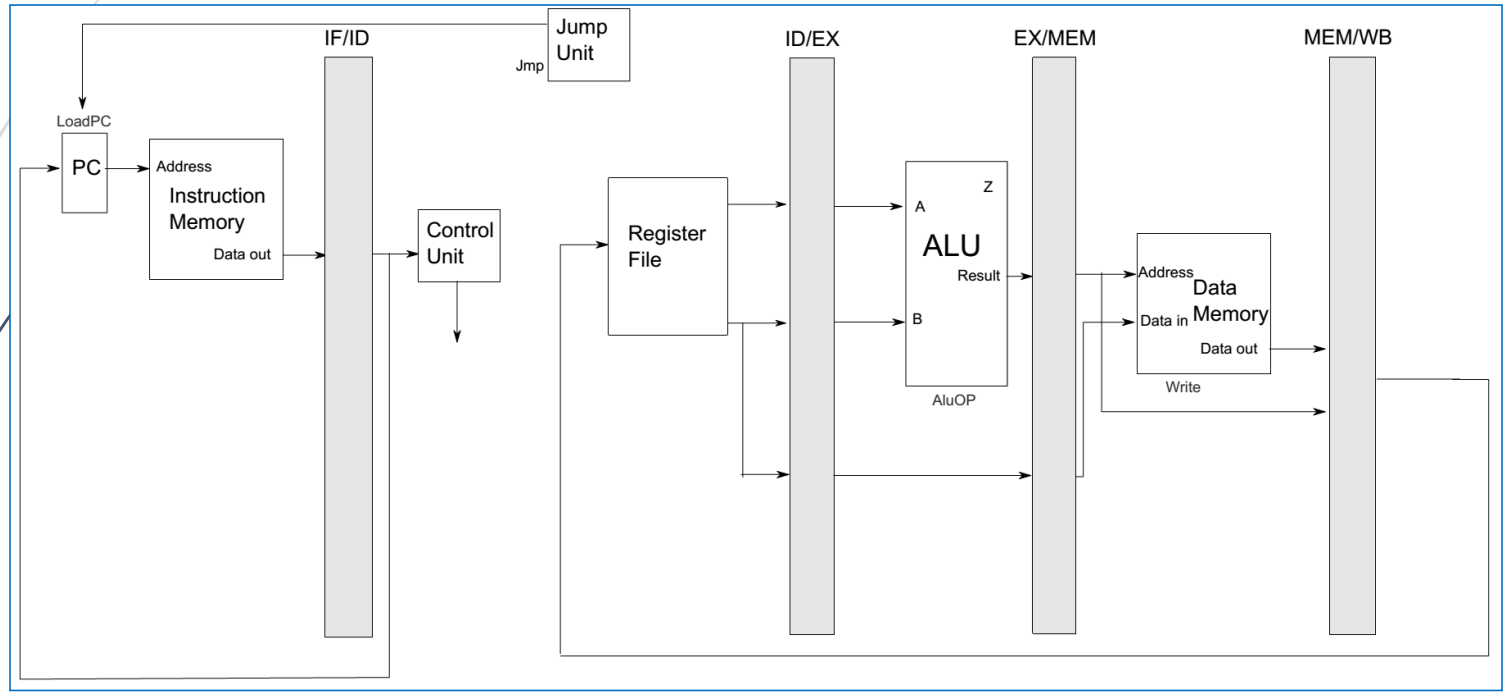
Al igual que en el caso de un pipeline:

- el compilador debe garantizar que estas condiciones se cumplan

  … o bien los conflictos deben ser detectados y eliminados en tiempo de ejecución usando hardware adicional

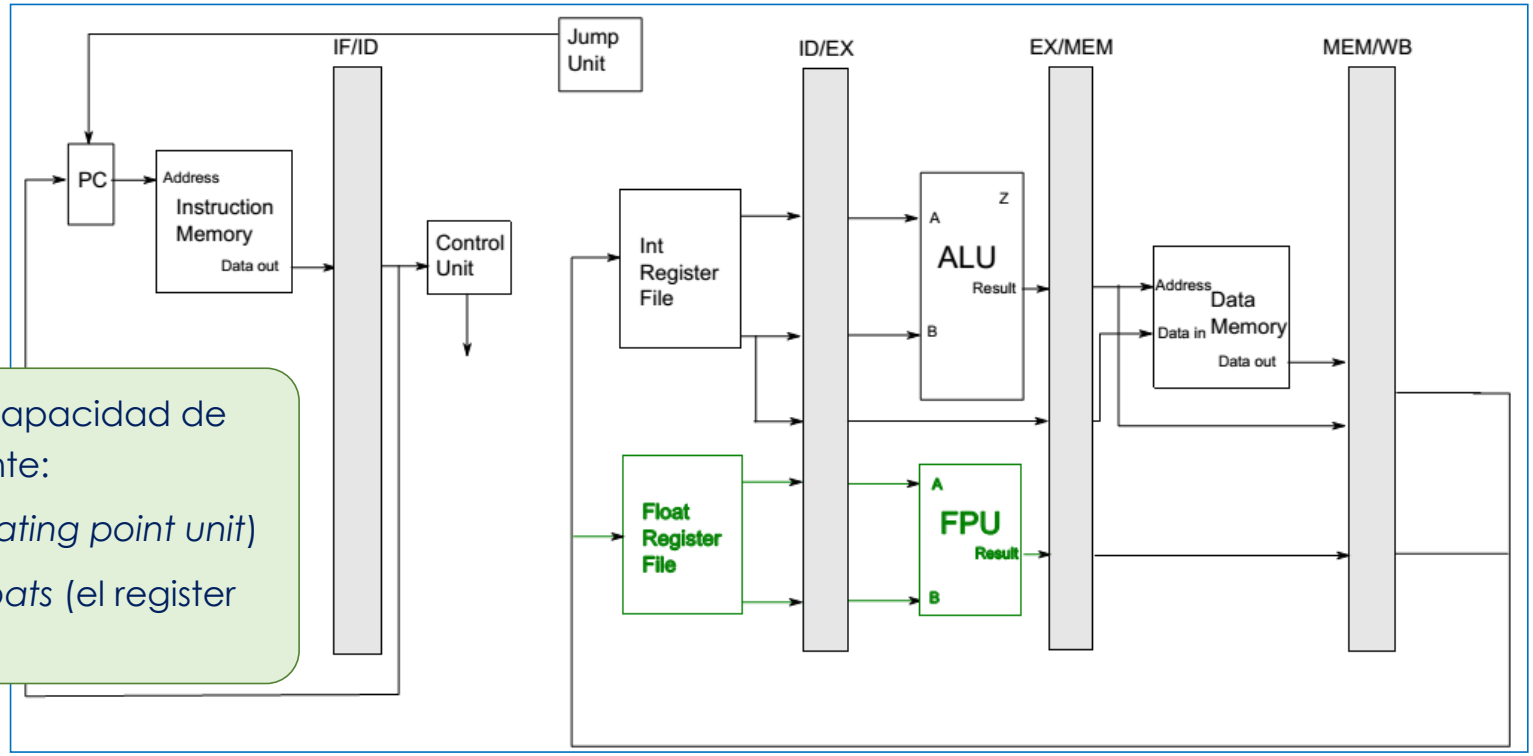
Consideremos el computador básico con pipelining

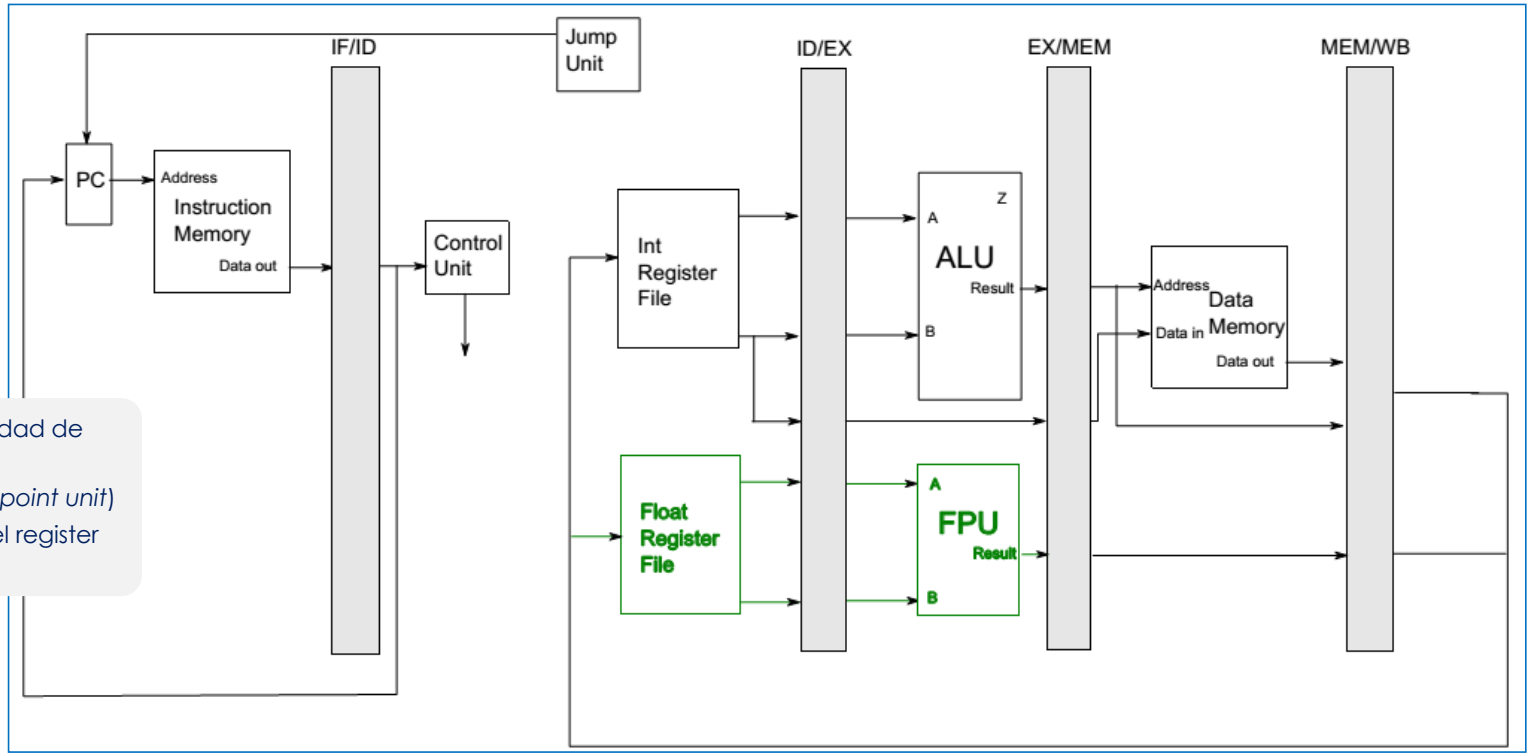
… pero en lugar de tener sólo los registros *A* y *B*, ahora tenemos un ***Register File***: conjunto de registros (p.ej., 16 o 32) a los que se tiene acceso individualmente, para lectura o escritura, especificando el número del registro



Agreguemos además la capacidad de aritmética de punto flotante:

- agregamos una **FPU** (*floating point unit*)
- … y un **register file** de *floats* (el register file original es de ints)



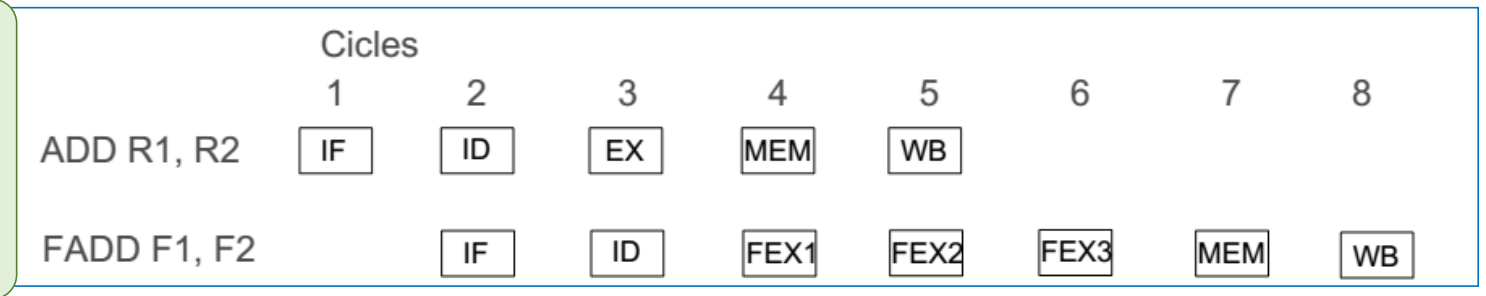


Agreguemos además la capacidad de aritmética de punto flotante:

- agregamos una **FPU** (*floating point unit*)
- … y un **register file** de *floats* (el register file original es de ints)

Supongamos que la etapa *EX* en la FPU se divide en 3 subetapas: *FEX1*, *FEX2*, *FEX3*

… y que ejecutamos un **ADD** seguido de un **FADD**



Notamos que las etapas *EX* (ALU y FPU) y *WB* (registros *int* y registros *float*) de ambas instrucciones son independientes …

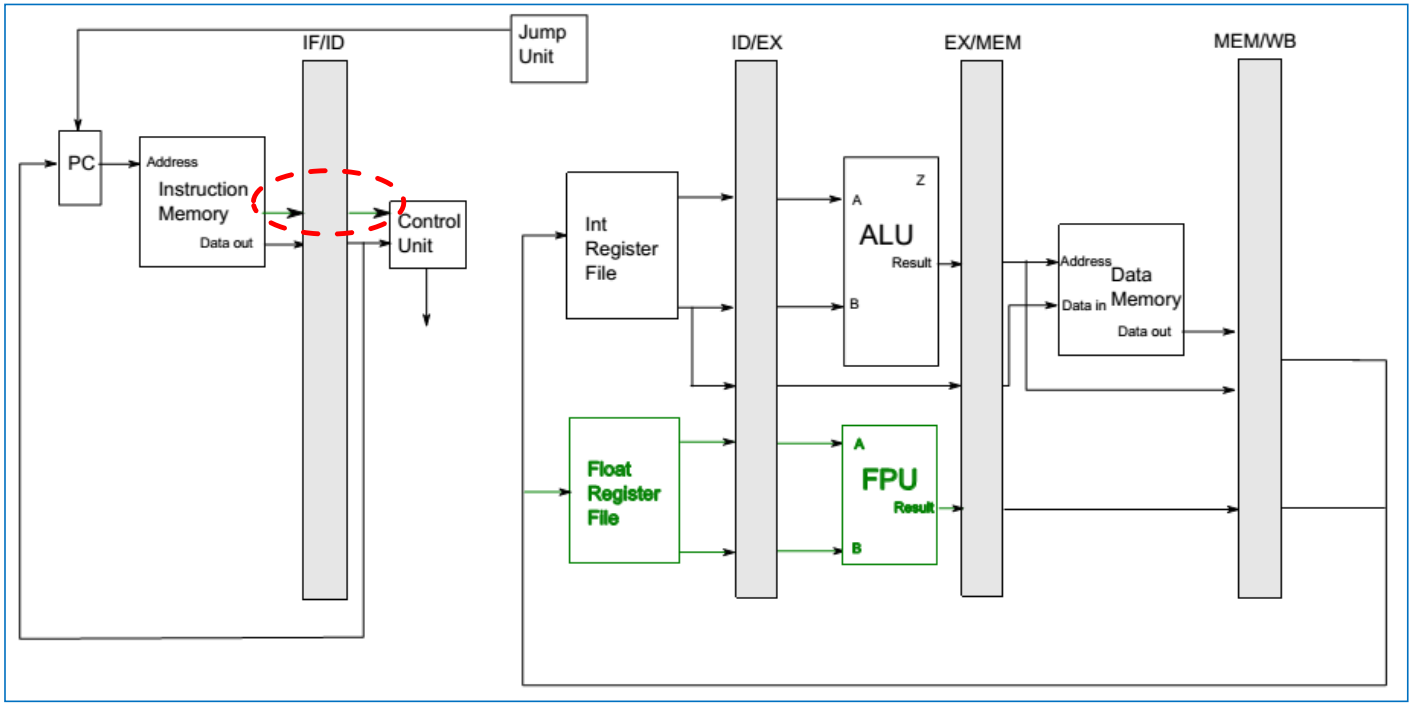
¿cómo podemos aprovecharlo?

Para poder ejecutar dos instrucciones por ciclo —***two-issue***— aumentamos la capacidad de la etapa *IF*:

- un segundo output de 32 bits en la *Instruction Memory*, de modo que en cada ciclo puedan salir dos instrucciones simultáneamente

… y de la etapa *ID*:

- un segundo input y hardware interno adicional en la *Control Unit*



Notamos que las etapas *EX* (ALU y FPU) y *WB* (registros *int* y registros *float*) de ambas instrucciones son independientes …

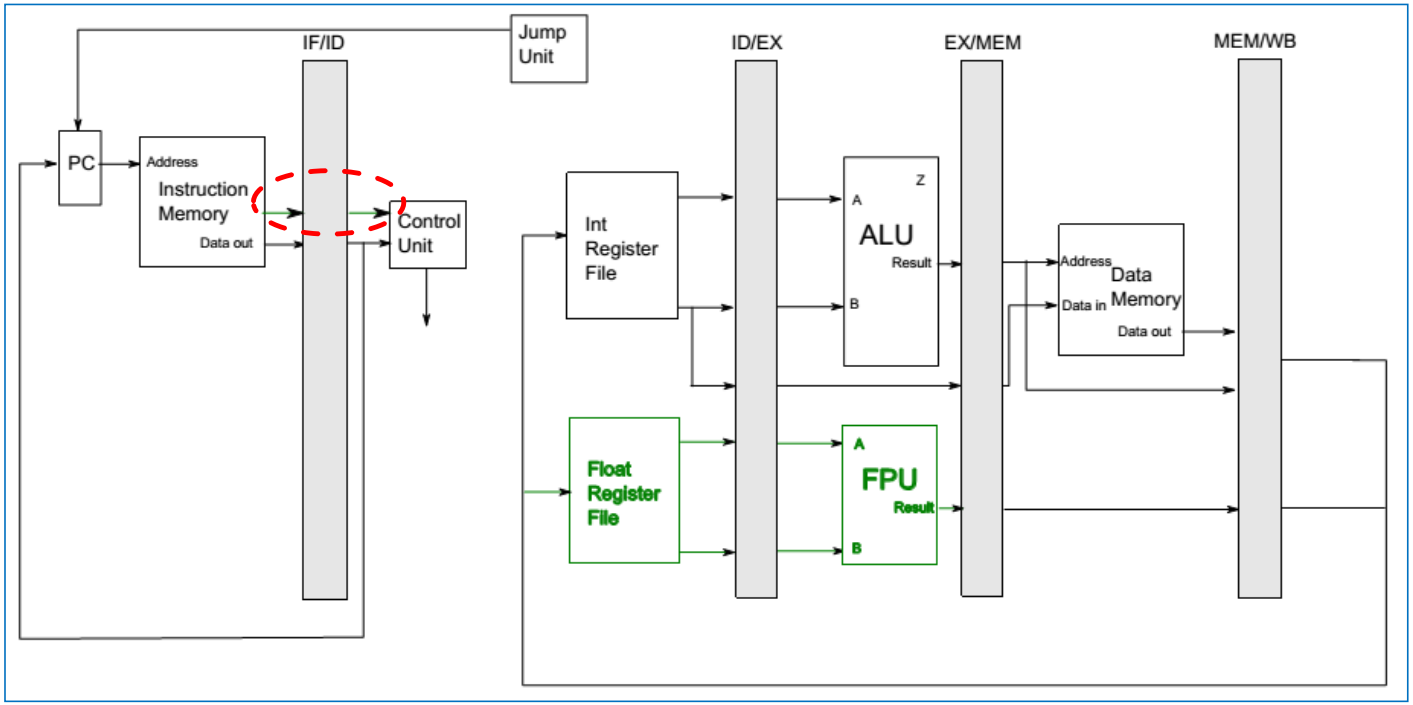
¿cómo podemos aprovecharlo?

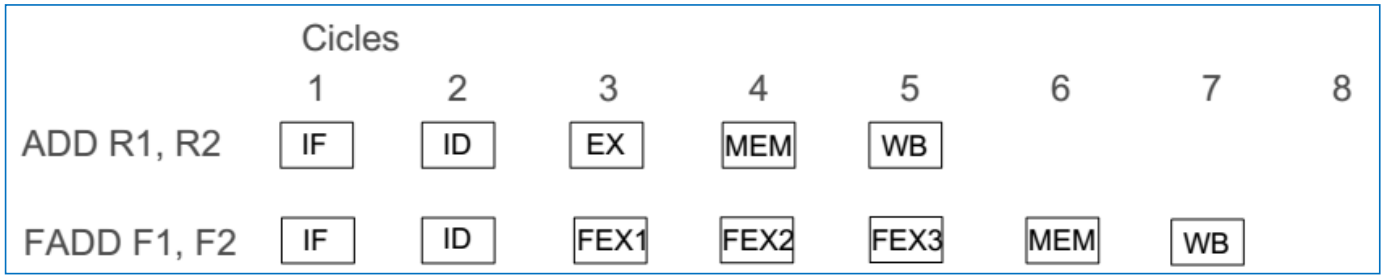
Para poder ejecutar dos instrucciones por ciclo —***two-issue***— aumentamos la capacidad de la etapa *IF*:

- un segundo output de 32 bits en la *Instruction Memory*, de modo que en cada ciclo puedan salir dos instrucciones simultáneamente

… y de la etapa *ID*:

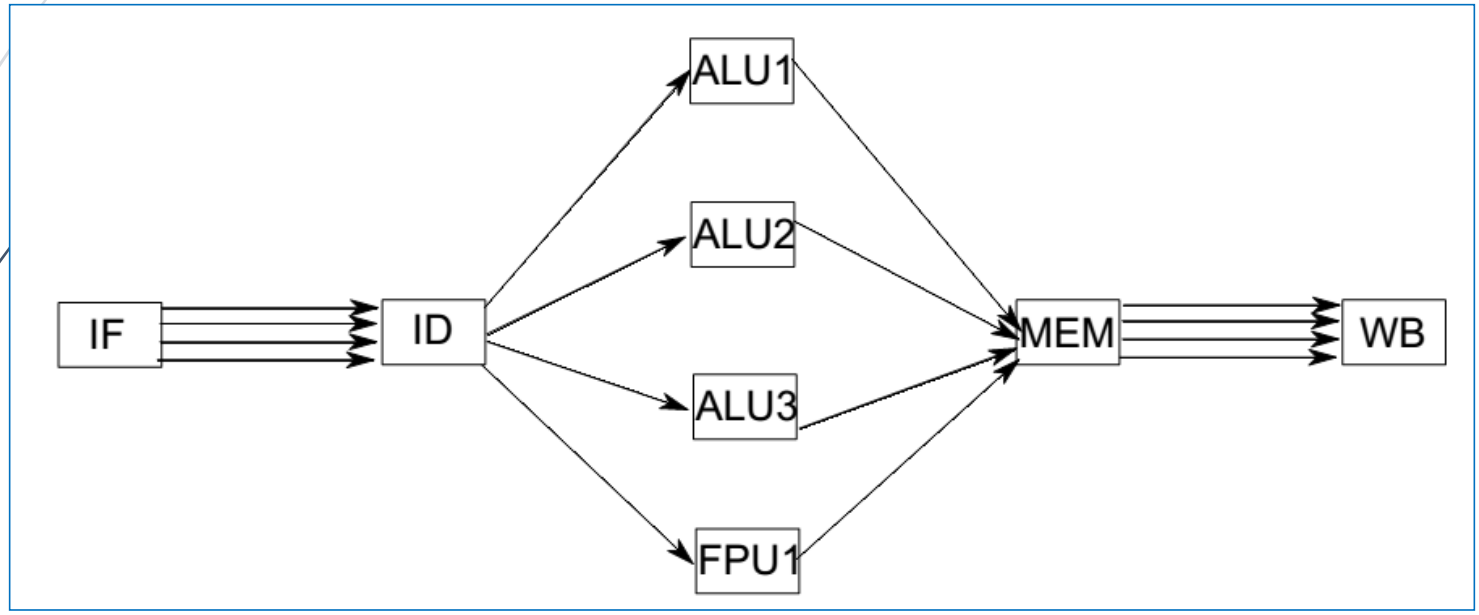
- un segundo input y hardware interno adicional en la *Control Unit*





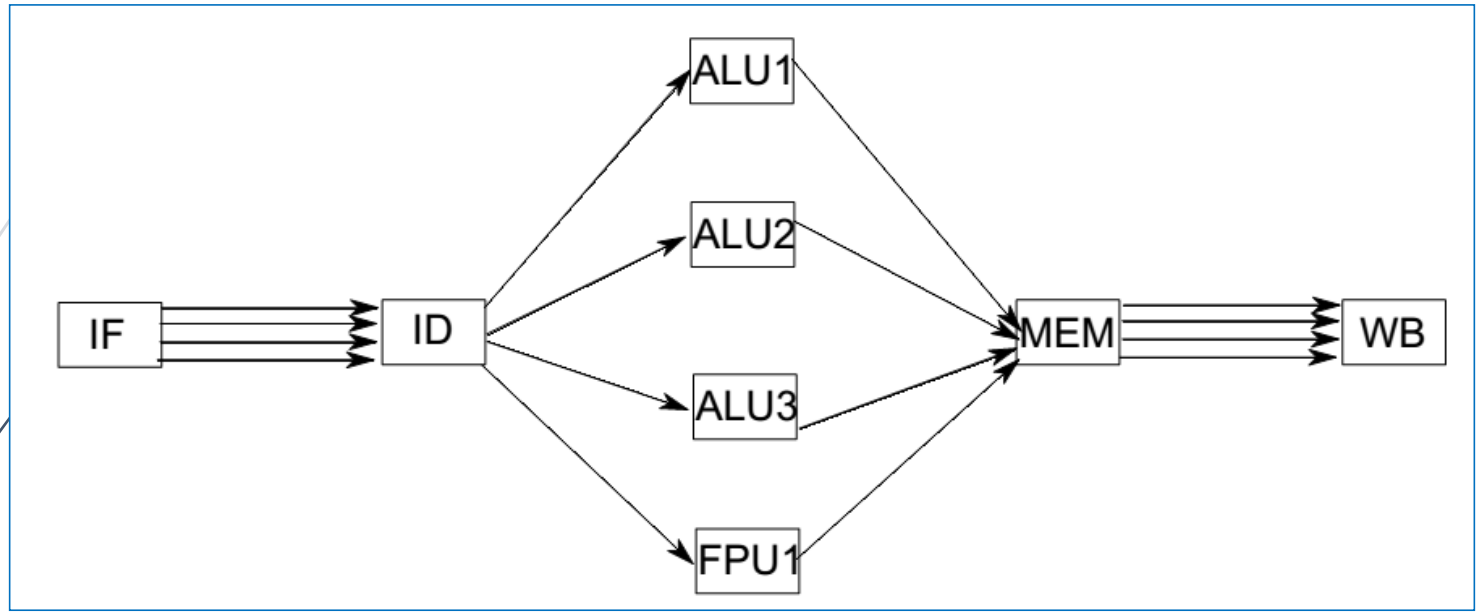
La idea de *two-issue* se puede extender a ***multiple issue***, pero en vez de replicar todo el hardware, tenemos un único pipe-line, pero con varias unidades funcionales en la etapa *EX*:

- p.ej., una FPU y tres ALUs (dos de propósito general y una para el cálculo de direcciones de memoria)



La idea de *two-issue* se puede extender a ***multiple issue***, pero en lugar de replicar todo el hardware, tenemos un único pipeline, pero con varias unidades funcionales en la etapa *EX*:

- p.ej., tres ALUs (dos de propósito general y una para el cálculo de direcciones de memoria) y una FPU



Los procesadores *multiple issue* requieren elementos adicionales para decidir sobre paralelismo:

- ¿cuántas y cuáles instrucciones pueden ser enviadas a ejecución en un mismo ciclo de reloj?
- ¿cómo resolvemos hazards de datos y de control?

Dos tipos de técnicas para realizar esto:

- la diferencia depende de la división de trabajo entre el compilador y el hardware

Técnicas **estáticas**:

- *dependen del compilador* para agrupar instrucciones paralelizables y hacerse cargo de las consecuencias de los hazards

Técnicas **dinámicas**:

- *permiten a la CPU determinar* en tiempo de ejecución las instrucciones a paralelizar —aunque el compilador normalmente ayuda reordenando las instrucciones convenientemente— y hacerse cargo de las consecuencias de los hazards

Especulación: Técnica empleada por el compilador y/o por el procesador para "adivinar" las propiedades de una instrucción:

- p.ej., especular sobre el resultado de un *branch* —si se toma o no

  … o especular que un *store* que precede a un *load* no se refiere a la misma dirección de memoria —y por lo tanto ambas instrucciones se pueden ejecutar simultáneamente

… de modo de permitir que comience la ejecución de otras instrucciones que podrían depender de la instrucción adivinada:

- p.ej., el compilador puede reordenar las instrucciones

  … o el hardware del procesador puede hacer transformaciones similares en tiempo de ejecución

La especulación puede errar:

- se necesita un método para chequear si la adivinanza fue correcta
- se necesita un método para deshacer o anular los efectos de las instrucciones ejecutadas especulativamente ⟶ mayor complejidad

Multiple issue estático, originalmente ***Very Long Instruction Word*** (***VLIW***):

- procesadores especializados en procesamiento de señales (TI C6x, Itanium)
- el compilador genera un *paquete* (*bundle*) de instrucciones ***que pueden ejecutarse en paralelo***
- la mezcla de instrucciones permitidas, y su disposición en el registro de output de la *Instruction Memory*, es restringida para simplificar la decodificación
- el *bundle* es enviado al procesador como una sola instrucción muy larga con varias operaciones

  … p.ej., 4 instrucciones convencionales

  … aunque a veces incluye instrucciones *NOP* porque no siempre es posible encontrar cuatro instrucciones que se puedan ejecutar en paralelo (Tabla 3)

Dirección	Instrucción
0x00	Instrucción 1
0x01	Instrucción 2
0x02	Instrucción 3
0x03	Instrucción 4
0x04	Instrucción 5
0x05	Instrucción 6
0x06	Instrucción 7
0x07	Instrucción 8
0x08	Instrucción 9

Tabla 2: Secuencia de instrucciones sin VLIW.

Dirección	Bundle			
0x00	Instrucción 1	Instrucción 6	Instrucción 7	NOP
0x01	NOP	NOP	Instrucción 3	Instrucción 4
0x02	NOP	Instrucción 2	NOP	NOP
0x03	NOP	Instrucción 5	Instrucción 9	NOP
0x04	NOP	NOP	NOP	Instrucción 8

Tabla 3: Secuencia de bundles con VLIW.

P.ej., consideremos un procesador RISC-V *two-issue* en que

… una instrucción puede ser una operación en la *ALU* o un *branch* y

… la otra un *load* o un *store*, como se muestra en la tabla

… y consideremos el siguiente loop:

```
Loop:   lw      x31, 0(x20)
        add     x31, x31, x21
        sw      x31, 0(x20)
        addi    x20, x20, -4
        blt     x22, x20, Loop
```

Tipo de instr.	Etapas del pipeline						
ALU o branch	IF	ID	EX	MEM	WB		
Load o store	IF	ID	EX	MEM	WB		
ALU o branch		IF	ID	EX	MEM	WB	
Load o store		IF	ID	EX	MEM	WB	
ALU o branch			IF	ID	EX	MEM	WB
Load o store			IF	ID	EX	MEM	WB

2-issue

P.ej., consideremos un procesador RISC-V *dos-issue* en que

… una instrucción puede ser una operación en la *ALU* o un *branch* y

… la otra un *load* o un *store*, como se muestra en la tabla

… y consideremos el siguiente loop:

```
Loop:   lw      x31, 0(x20)
        add     x31, x31, x21
        sw      x31, 0(x20)
        addi    x20, x20, -4
        blt     x22, x20, Loop
```

5 ciclos

Tipo de instr.	Etapas del pipeline						
ALU o branch	IF	ID	EX	MEM	WB		
Load o store	IF	ID	EX	MEM	WB		
…		IF	ID	EX	MEM	WB	

2-issue

	ALU o branch	*Load o store*	ciclo
Loop:	NOP	`lw x31, 0(x20)`	1
	`addi x20, x20, -4`	NOP	2
	`add x31, x31, x21`	NOP	3
	`blt x22, x20, Loop`	`sw x31, 4(x20)`	4

4 ciclos → ***IPC*** = 1.25

Única *VLIW* con ambas operaciones ocupadas

… y consideremos el siguiente loop:

```
Loop:   lw      x31, 0(x20)
        add     x31, x31, x21
        sw      x31, 0(x20)
        addi    x20, x20, -4
        blt     x22, x20, Loop
```

5 ciclos

Tipo de instr.	Etapas del pipeline						
ALU o branch	IF	ID	EX	MEM	WB		
Load o store	IF	ID	EX	MEM	WB		
…		IF	ID	EX	MEM	WB	

2-issue

	ALU o branch	*Load o store*	ciclo
Loop:	`addi x20,x20,-16`	`lw x28, 0(x20)`	1
	NOP	`lw x29,12(x20)`	2
	`add x28, x28, x21`	`lw x30, 8(x20)`	3
	`add x29, x29, x21`	`lw x31, 4(x20)`	4
	`add x30, x30, x21`	`sw x28,16(x20)`	5
	`add x31, x31, x21`	`sw x29,12(x20)`	6
	NOP	`sw x30, 8(x20)`	7
	`blt x22, x20, Loop`	`sw x31, 4(x20)`	8

Loop *unrolling*

Hacemos 4 copias del cuerpo del loop y simplificamos los **`addi`** y **`blt`**; quedan 4 copias de **`lw`**, **`add`** y **`sw`**

Para permitir este *unrolling*, usamos registros adicionales: **`x28`**, **`x29`** y **`x30`**—***register renaming***—eliminando dependencias que no son verdaderas dependencias de datos

Así, en 8 ciclos de reloj (en lugar de 20) ejecutamos 4 ciclos del loop

… y 12 de las 14 instrucciones *VLIW* tienen ambas operaciones ocupadas ⟶ ***IPC*** = 1.75

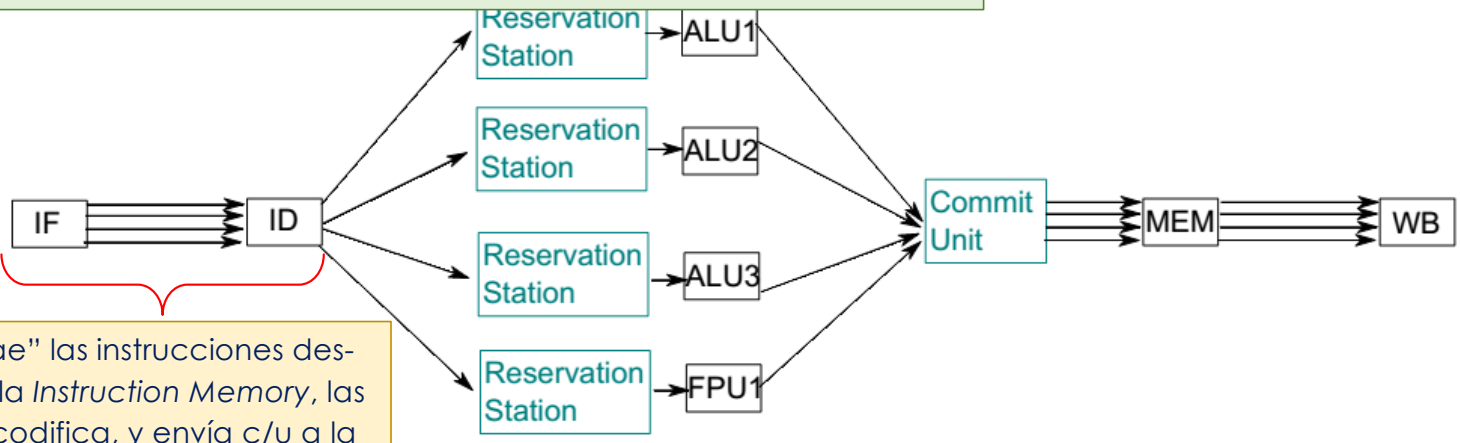
El costo es usar más registros temporales y un aumento significativo del tamaño del código

Multiple issue dinámico, o **procesadores superescalares:**

- las instrucciones son emitidas en orden; el procesador decide si cero, una o más instrucciones pueden ser ejecutadas en un mismo ciclo (el compilador ayuda reordenando las instrucciones evitando al máximo las dependencias)
- la decisión del procesador está basada en tratar de evitar *hazards* y *stalls*; puede incluir reordenación (adicional) de las instrucciones
- el hardware garantiza que el código se ejecuta correctamente

  … independiente de la tasa de emisión (*issue rate*) y de la estructura del pipeline (esto no necesariamente es cierto en VLIW)

Los procesadores "normales" fueron clasificados, en su momento, como "escalares" para diferenciarlos de los procesadores "vectoriales," efectivamente especializados para procesar de vectores y matrices



"Trae" las instrucciones desde la *Instruction Memory*, las decodifica, y envía c/u a la unidad funcional correspondiente para su ejecución

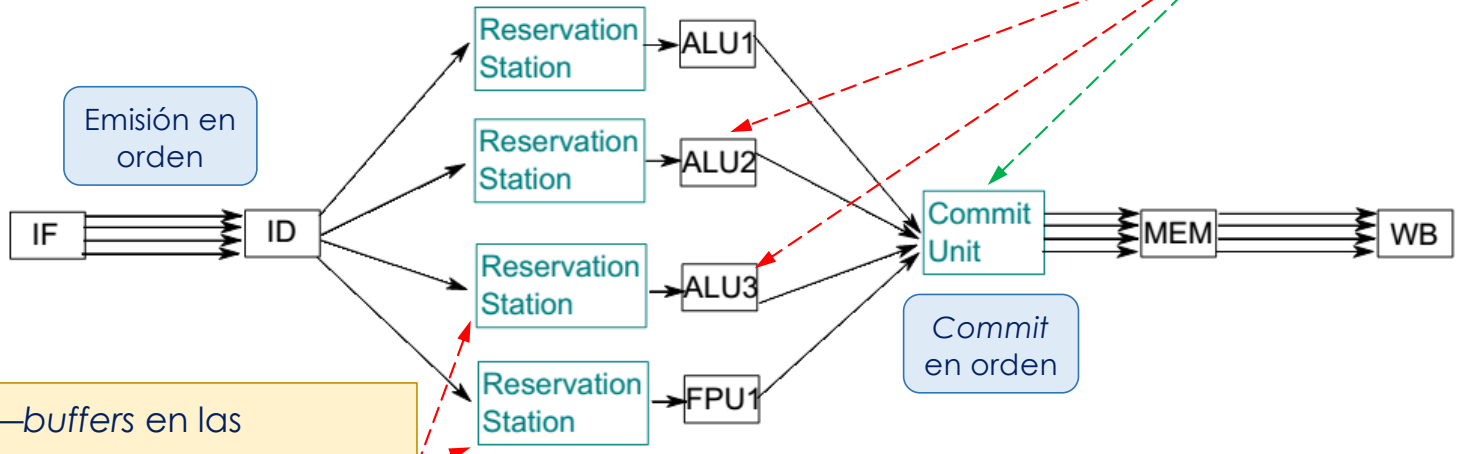
Multiple issue dinámico, o **procesadores superescalares**:

- las instrucciones son emitidas en orden; el procesador decide si cero, una o más instrucciones pueden ser ejecutadas en un mismo ciclo (el compilador ayuda reordenando las instrucciones evitando al máximo las dependencias)
- la decisión del procesador está basada en tratar de evitar *hazards* y *stalls*; puede incluir reordenación (adicional) de las instrucciones
- el hardware garantiza que el código se ejecuta correctamente

  … independiente de la tasa de emisión (*issue rate*) y de la estructura del pipeline (esto no necesariamente es cierto en VLIW)

Cuando una *station* tiene todos los operandos y la ALU o FPU está lista para ejecutar, se calcula el resultado

… que se envía a cualquier otra *station* que lo esté esperando, y también a la *Commit Unit*



Las ***reservation stations*** —*buffers* en las unidades funcionales (ALUs, FPUs)— almacenan temporalmente los operandos y la operación

Multiple issue dinámico, o **procesadores superescalares**:

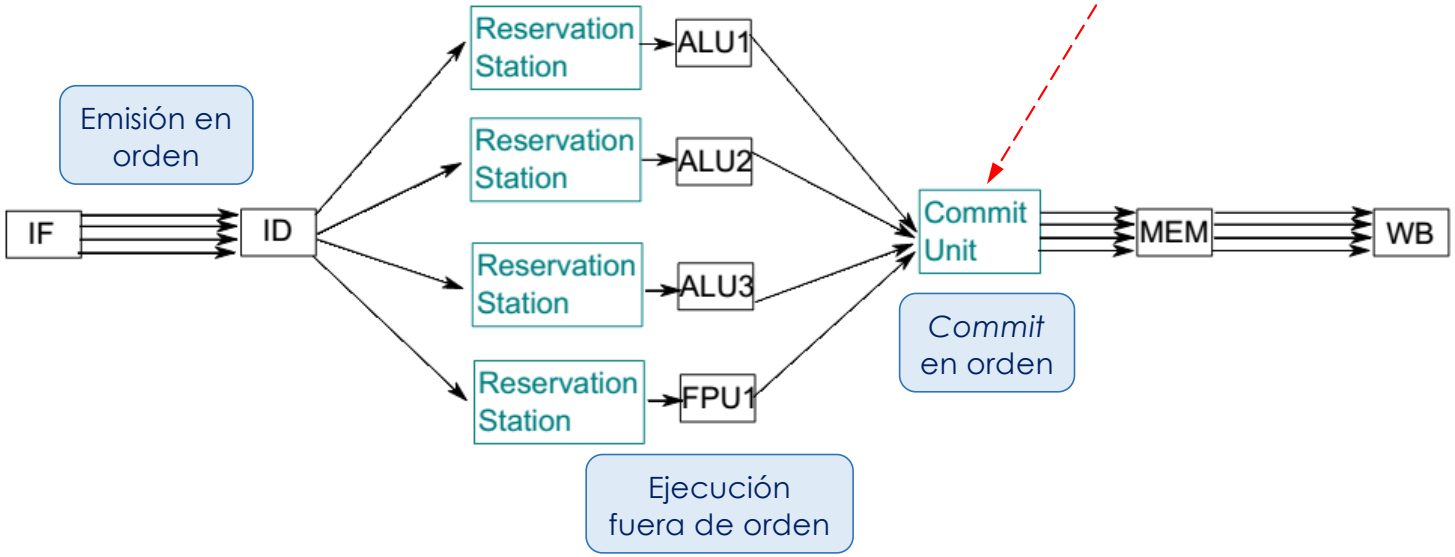
- las instrucciones son emitidas en orden; el procesador decide si cero, una o más instrucciones pueden ser ejecutadas en un mismo ciclo (el compilador ayuda reordenando las instrucciones evitando al máximo las dependencias)
- la decisión del procesador está basada en tratar de evitar *hazards* y *stalls*; puede incluir reordenación (adicional) de las instrucciones
- el hardware garantiza que el código se ejecuta correctamente

  … independiente de la tasa de emisión (*issue rate*) y de la estructura del pipeline (esto no necesariamente es cierto en VLIW)

La ***commit unit*** almacena temporalmente el resultado en un *reorder buffer*,

… hasta que decide que es seguro escribirlo en la memoria (*MEM*) o en los registros (*WB*)

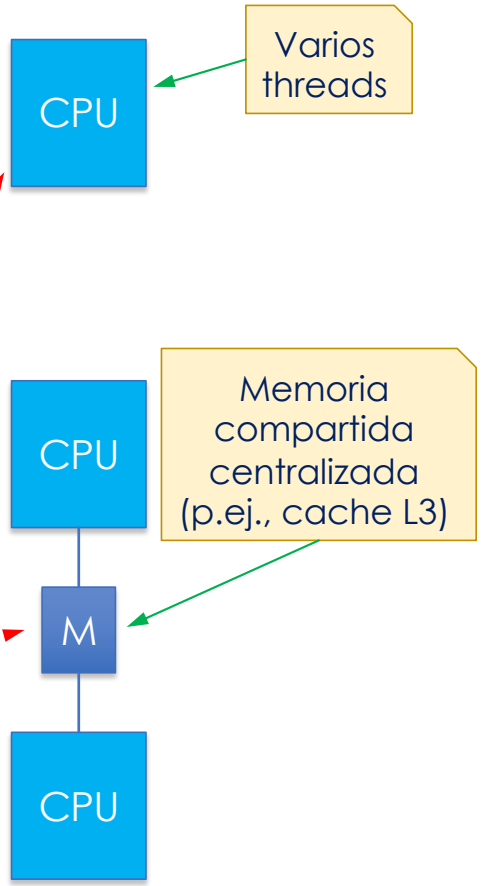
El *buffer* también se usa para enviar operandos, similarmente a la lógica de *forwarding*



Paralelismo en el chip

Una forma de aumentar el *throughput* de un chip —más allá de aumentar la frecuencia del reloj— es conseguir que haga más cosas al mismo tiempo:

- paralelismo a nivel de instrucciones: *pipelining*, *multiple-issue* (VLIW, superescalares), ***multithreading***
- multiprocesadores en un chip (chip *multicores*)



Multithreading en el chip

El paradigma SISD tiene problemas de complejidad:

buen rendimiento y manejo de todas las posibles situaciones ⟶ los procesadores aumentan mucho su complejidad

⟶ aumento en el costo y en uso de energía

Además, problema inherente en *pipelining*:

si una referencia a memoria no está en la caché, hay que esperar un buen rato hasta que la línea correspondiente sea traida a la caché

… durante esa espera, el pipeline se detiene (*stall*)

Usamos **multithreading** para "esconder" las demoras debidas al pipeline y a los accesos a memoria (por eso lo estudiamos al final de ILP):

- varios *threads* comparten las unidades funcionales de un mismo procesador

  … duplicando el estado privado de cada *thread* —registros, PC y tabla de páginas— pero no la totalidad del procesador

- la memoria principal es compartida gracias al mecanismo de memoria virtual (el cual permite multiprogramación)

- el hardware debe poder cambiar a otro *thread* rápidamente

  … un cambio de *thread* debe ser mucho más eficiente que un cambio de proceso (que requiere miles de ciclos)

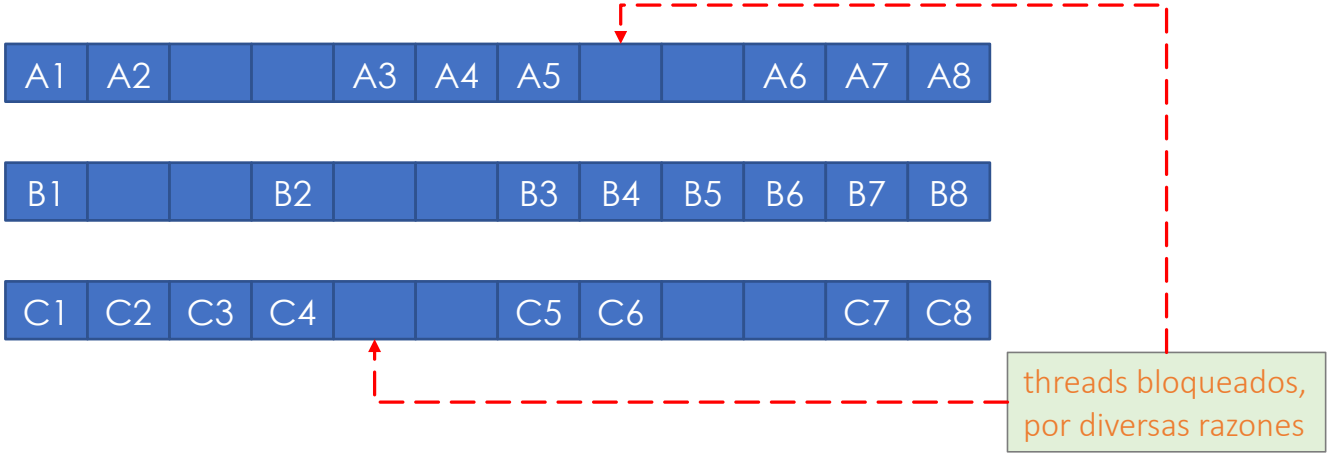
  … idealmente, un mismo programa debe contener varios *threads* que puedan ser ejecutados concurrentemente —los *threads* son identificados por el compilador o definidos por el programador

La técnica de **multithtreading** (en el chip) permite que un único procesador, o una única CPU, ejecute varios ***threads*** al mismo tiempo

⟶ si un *thread* está bloqueado, la CPU puede ejecutar otro

El hardware debe permitir cambiar de *thread* muy rápidamente

P.ej., consideremos los *threads A*, *B* y *C* a lo largo de los 12 primeros ciclos de cada uno



Multithreading de granularidad fina:

- se define el número máximo de threads al diseñar el chip

  ⟶ debe haber un set de registros (*register file* + *PC*) para cada thread (pero la memoria puede ser compartida a través de *memoria virtual*)

- las instrucciones se envían al pipeline intercaladamente (*round robin*) desde cada thread (saltándose los threads que estén bloqueados en ese ciclo)

  … más un puntero al set de registros correspondiente

- requiere identificador de thread para cada operación

A1	B1	C1	A2	B2	C2	A3	B3	C3	A4	B4	C4

A1	B1	C1	A3	B2	C3	A5	B3	C5	A6	B5	C7
A2		C2	A4		C4		B4	C6	A7	B6	C8

CPU *two-issue*

Multithreading de granularidad gruesa:

- se envían al pipeline las instrucciones de un mismo thread, hasta que se ponga a esperar (p.ej., un cache miss en el nivel 3)

  … o hasta que se ejecute una instrucción que podría causar una espera (load, store, jump)

  … entonces, se cambia a otro thread
- requiere pocos threads para mantener ocupada la CPU
- requiere identificador de thread, o limpiar el pipeline cuando cambia el thread

A1	A2		B1		C1	C2	C3	C4	A3	A4	A5

A1	B1	C1	C3	A3	A5	B2	C5	A6	A8	B3	B5
A2		C2	C4	A4			C6	A7		B4	B6

CPU *two-issue*

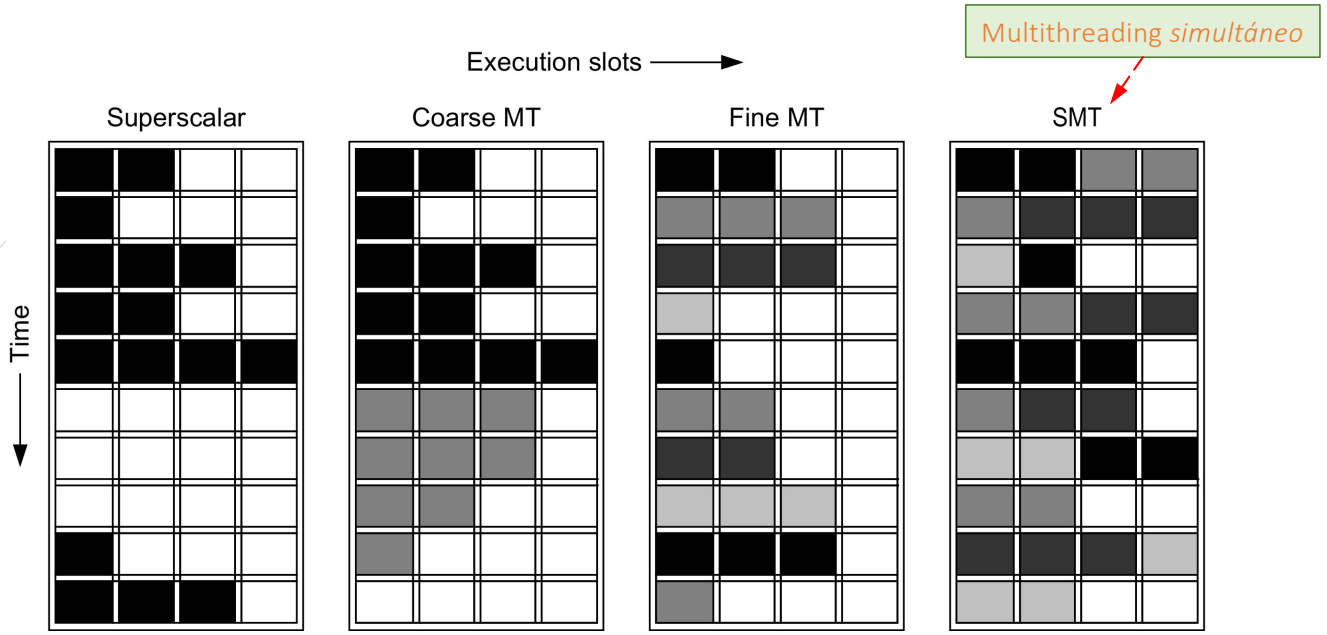


Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support.

The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

ARM Cortex-A53:

- procesador base de tablets y celulares
- two-issue, superescalar

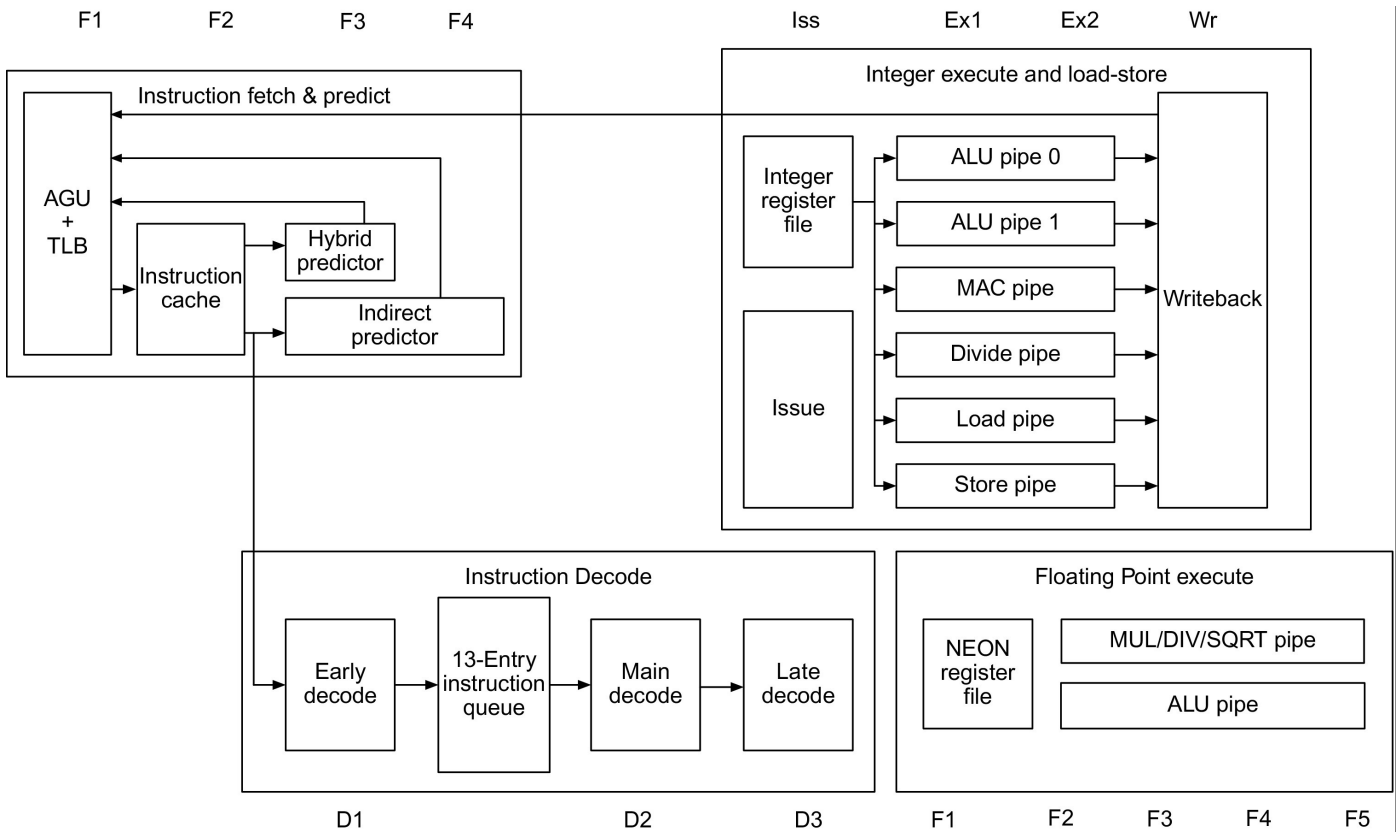


Figure 3.34 The basic structure of the A53 integer pipeline is 8 stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors, depending on the type. The floating-point execution pipeline is 5 cycles deep, in addition to the 5 cycles needed for fetch and decode, yielding 10 stages in total.