

Arquitecturas Paralelas

Aumentar la frecuencia del reloj no es la única manera de acelerar el procesamiento

... hay diversas limitaciones:

- velocidad de la luz
- disipación de calor
- tamaño de los transistores

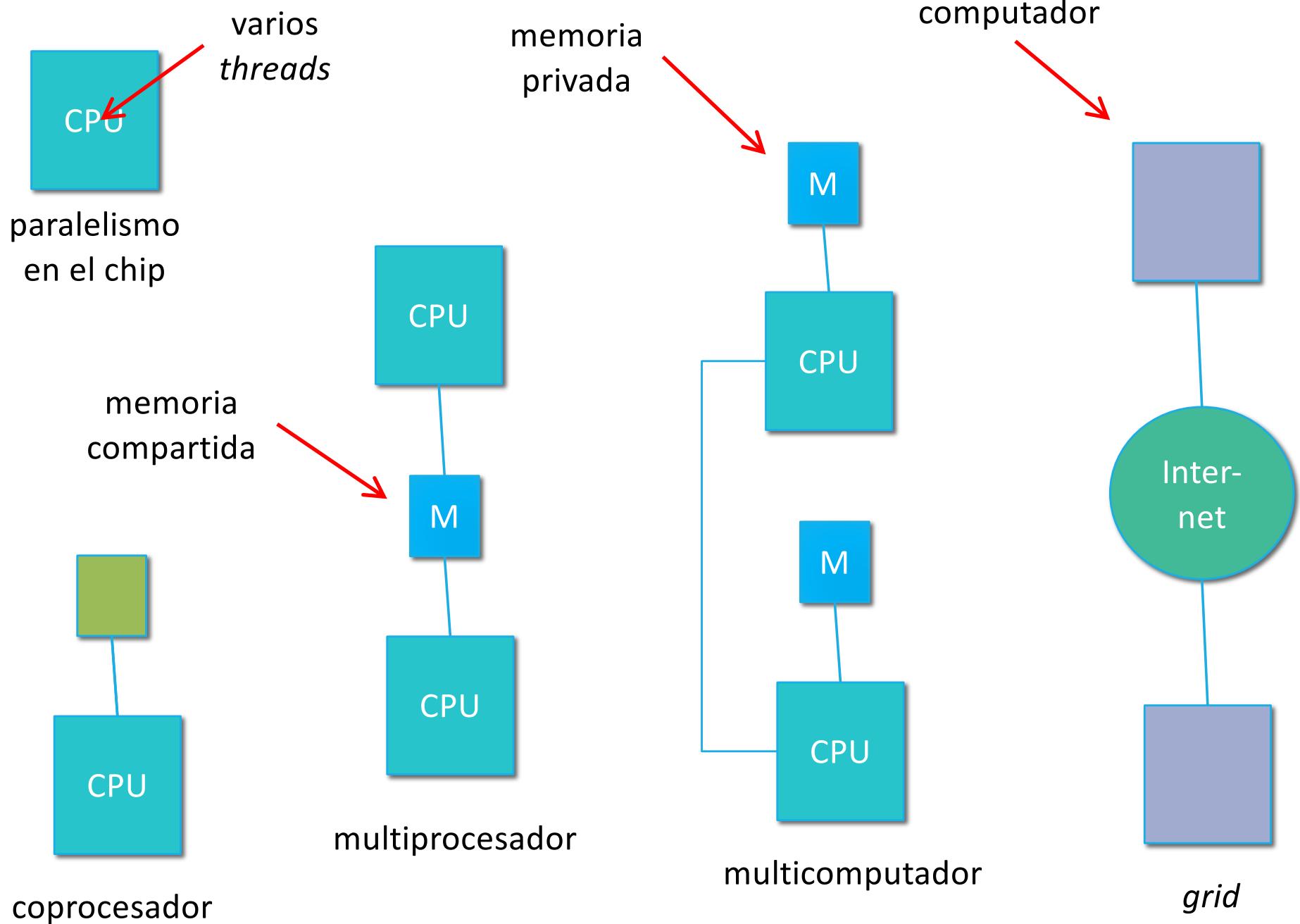
Revisaremos arquitecturas paralelas desde el punto de vista de instrucciones y datos

Estas arquitecturas requieren hardware más complejo

... pero sus ventajas son mayores que las asociadas a sólo aumentar la frecuencia del reloj o dividir una instrucción en varias etapas:

- (tal vez) no es posible construir un computador con una sola CPU y un tiempo de ciclo de 0.001 ns

... pero sí es posible construir uno con 1000 CPUs, cada una con un tiempo de ciclo de 1 ns



I) 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
- *multithreading*
- multiprocesadores en un chip (p.ej., múltiples *cores*) —lo veremos en el contexto de multiprocesadores en general

I-A) Paralelismo a nivel de instrucciones:

- *pipelining* —en particular, aumentando el número de etapas (la profundidad) del pipeline, lo que permite traslapar más instrucciones
- procesadores **VLIW** (*very long instruction word*)
- procesadores superescalares

Si un pipeline es bueno, entonces dos es mejor

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

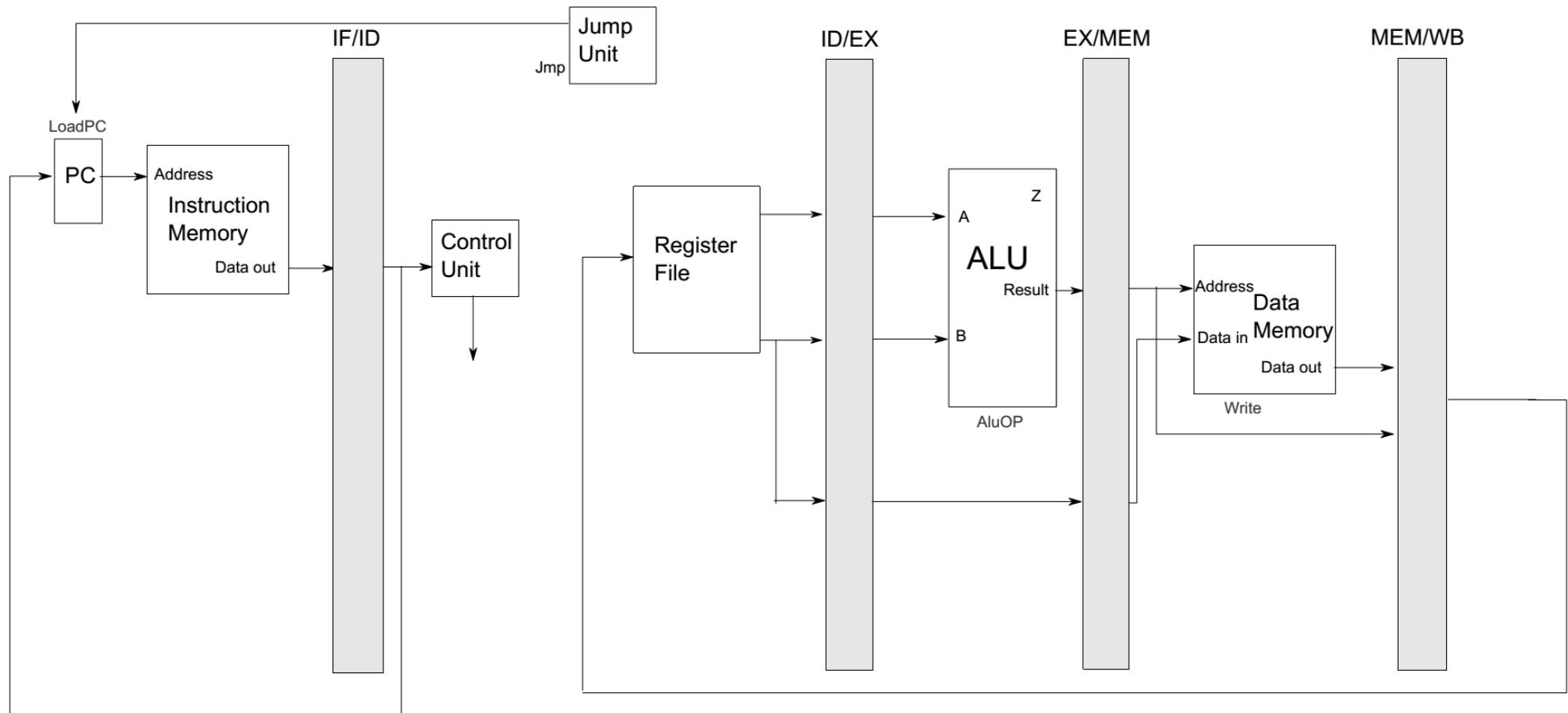
... y coloca cada una en su propio pipeline —***two-issue***:

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

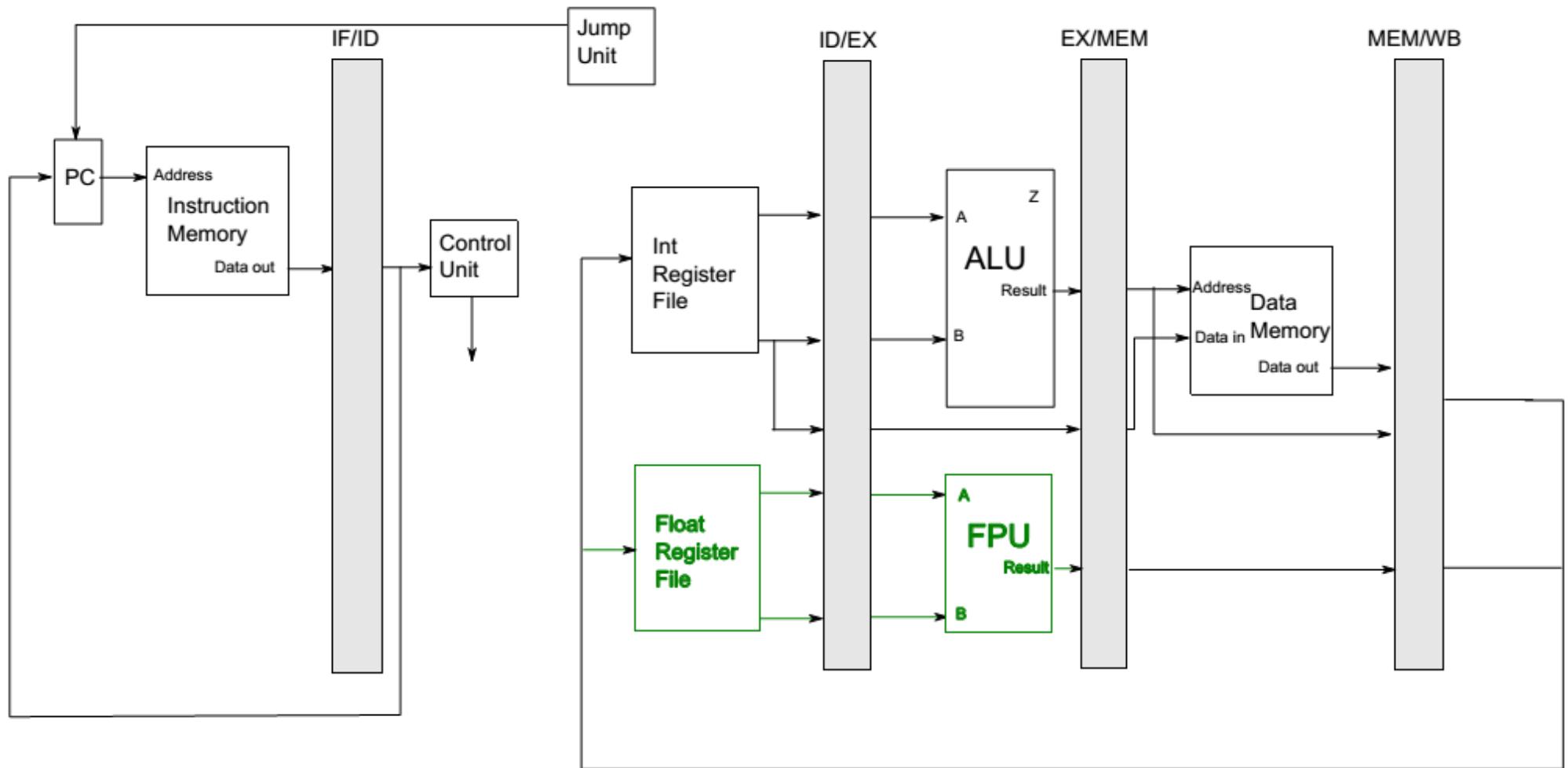
Al igual que en el caso de un pipeline:

- el compilador debe garantizar que estas condiciones se cumplan
... o bien los conflictos deben ser detectados y eliminados en tiempo de ejecución usando hardware extra

Revisemos el computador básico con pipeline



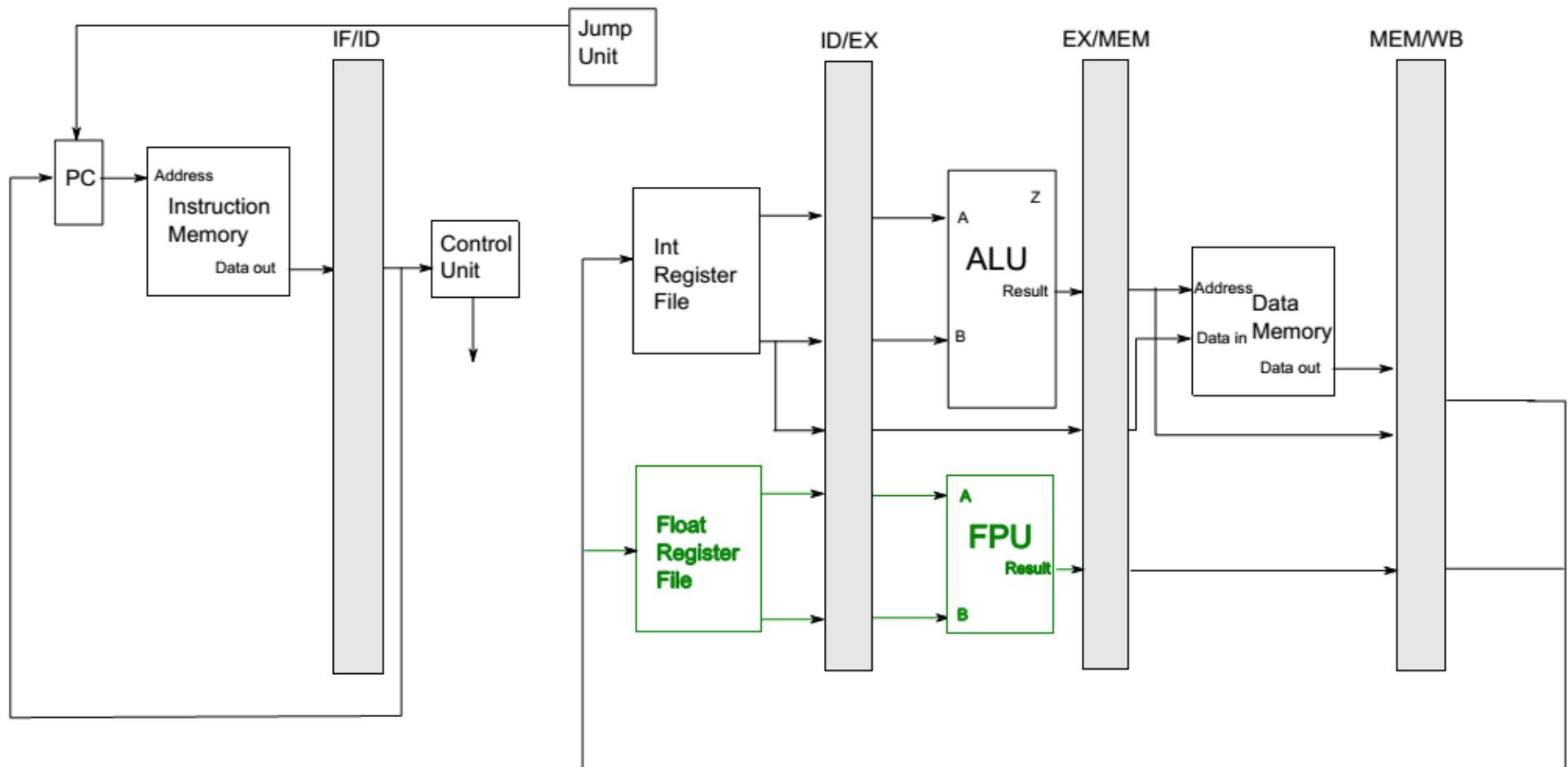
Agregamos un register file de *floats* y una FPU



Supongamos que la etapa EX en la FPU
se puede dividir en 3 subetapas

	Cicles							
	1	2	3	4	5	6	7	8
ADD R1, R2	IF	ID	EX	MEM	WB			
FADD F1, F2		IF	ID	FEX1	FEX2	FEX3	MEM	WB

EX y WB de ambas instrucciones son independientes;
¿cómo podemos aprovechar esto?

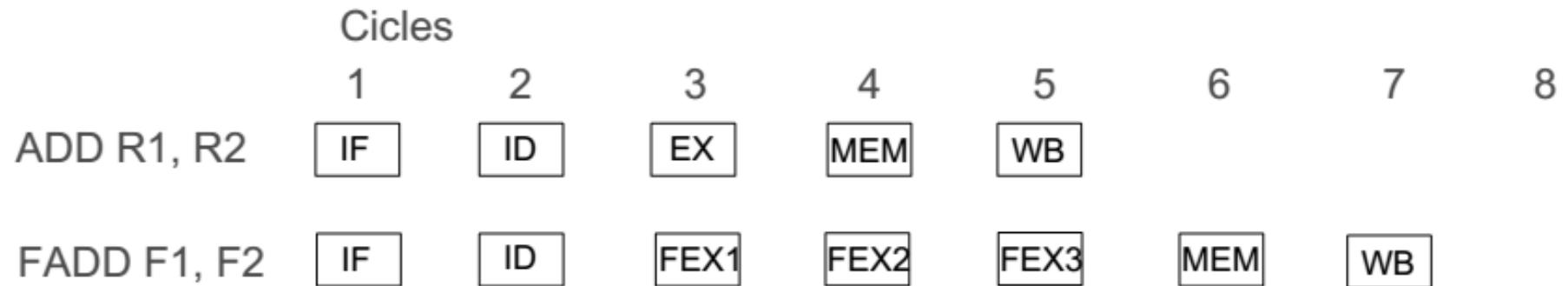


Para lograr paralelismo —*two-issue*— aumentamos la capacidad de las etapas *IF*:

- un segundo output de 32 bits en la memoria de instrucciones

... e *ID*, p.ej.:

- un *register file* para números de punto flotante, como en la diapositiva anterior



Especulación: Técnica empleada por el compilador o por el procesador para “adivinar” las propiedades de una instrucción

... de modo de permitir que comience la ejecución de otras instrucciones que pueden depender de la instrucción adivinada:

- 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 —y por lo tanto se pueden ejecutar simultáneamente

La especulación puede errar:

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

La idea de *two-issue* se puede extender a ***multiple issue***

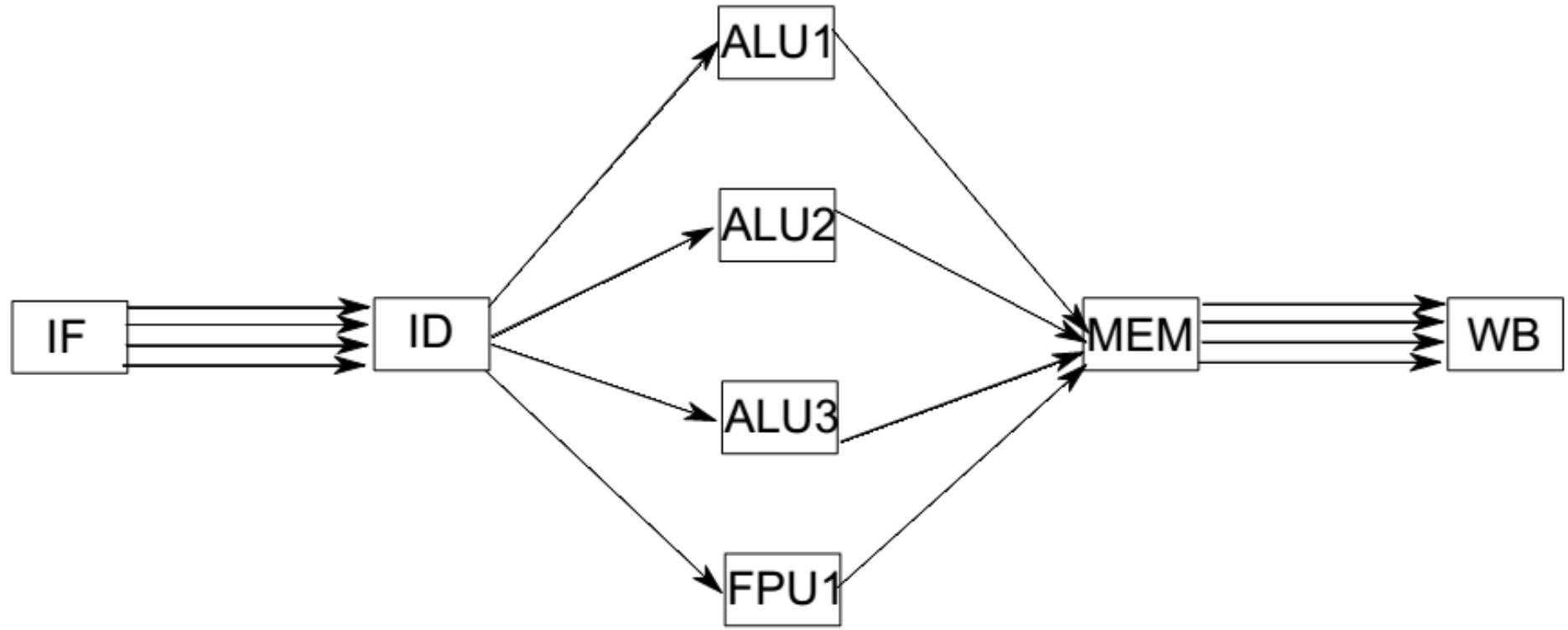
Los procesadores *multiple-issue* requieren elementos adicionales para decidir sobre paralelismo

Existen dos tipos de técnicas para realizar esto:

- técnicas estáticas *dependen del compilador* para agrupar instrucciones paralelizables
- técnicas dinámicas *permiten a la CPU determinar* en tiempo de ejecución las instrucciones a parallelizar, enviándolas a unidades de ejecución distintas

La idea de *two-issue* se puede extender a ***multiple issue***, pero en lugar de duplicar todo el hardware, se tiene un único pipeline con varias unidades funcionales:

- p.ej., tres ALUs y una unidad de punto flotante



La técnica estática más usada es ***Very Long Instruction Word (VLIW)***:

- el compilador genera un paquete (*bundle*) de instrucciones que pueden ejecutarse en paralelo
- el *bundle* es enviado al procesador como una instrucción muy larga
 - ... p.ej., 4 instrucciones convencionales
- la CPU reordena las instrucciones (Tabla 2) del grupo y lo envía en paralelo a las distintas unidades de ejecución
 - ... aunque a veces debe incluir 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

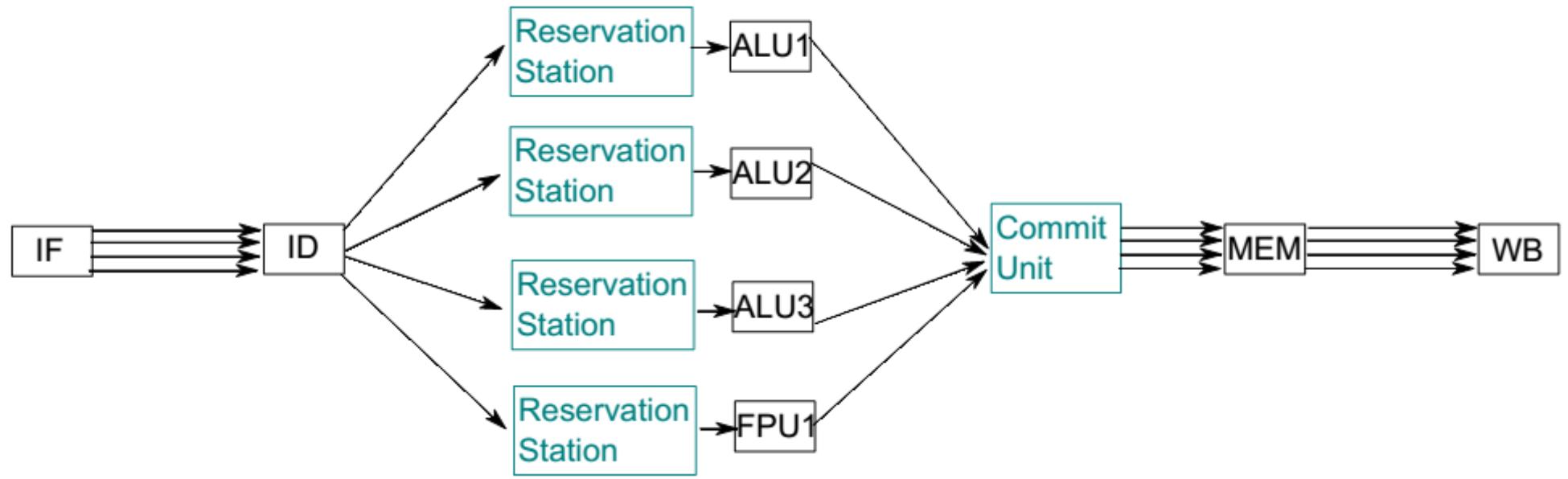
Tabla 2: Secuencia de instrucciones sin VLIW.

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.

La técnica dinámica más usada es **arquitecturas superescalares**:

- *reservation stations* son *buffers* dentro de las unidades funcionales que almacenan temporalmente los operandos y la operación
- una *commit unit* decide cuándo es seguro enviar el resultado de una operación a los registros y a la memoria



I-B) *Multithreading* en el chip

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*:

- cuando una referencia a memoria no está en la caché, hay que esperar un buen rato hasta que la palabra y la línea asociada sean cargadas en la caché
- ... mientras tanto, el pipeline se detiene (*stall*)

Una alternativa es usar múltiples procesadores simples (más adelante)

... otra opción es *multithreading*

La técnica de *multithreading* en el chip permite que la CPU maneje varios *threads* de control al mismo tiempo

→ si un *thread* está bloqueado, la CPU tiene la posibilidad de ejecutar otro

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 para cada *thread*)
- al enviar instrucciones al pipeline, se envían intercaladamente (*round robin*) una instrucción de cada *thread*
 - ... acompañada de 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
 - ... 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*

II) Multiprocesadores y multicomputadores

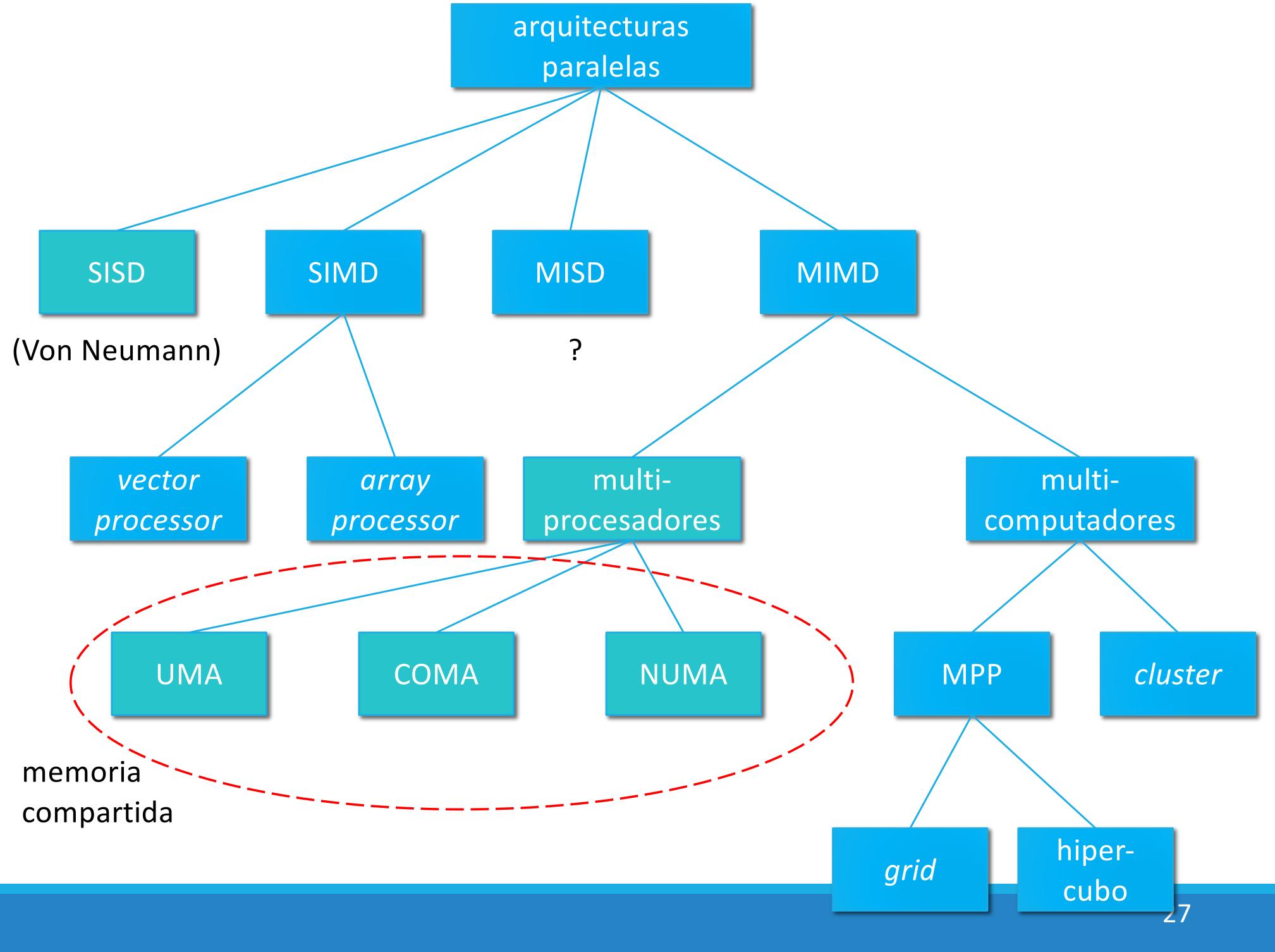
Dependiendo de si empleamos

... múltiples programas (I)

... y/o múltiples fuentes de datos (D)

... la *taxonomía de Flynn* —una aproximación muy simplista— define cuatro tipos de arquitecturas:

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD



¿Cómo pueden combinarse múltiples CPUs para formar sistemas más grandes?

- multiprocesadores
- multicamputadores

En un sistema de computadores paralelos (MIMD)

... las CPUs que trabajan en distintas partes de una misma tarea deben comunicarse para intercambiar información :

- multiprocesadores y multicomputadores se diferencian porque en los primeros existe una memoria compartida
... mientras que los segundos deben enviar y recibir mensajes
- esta diferencia impacta cómo son diseñados, construidos y programados

II-A)

Un computador paralelo en el que todas las CPUs comparten una memoria común es un **multiprocesador**:

- todos los procesos comparten un único espacio de direcciones virtuales mapeado a esta memoria común
- cualquier proceso puede leer o escribir una palabra de memoria ejecutando una instrucción tipo LOAD o STORE
- dos procesos se pueden comunicar haciendo que uno de ellos escriba datos en la memoria y que el otro los lea
- modelo fácil de entender y ampliamente aplicable

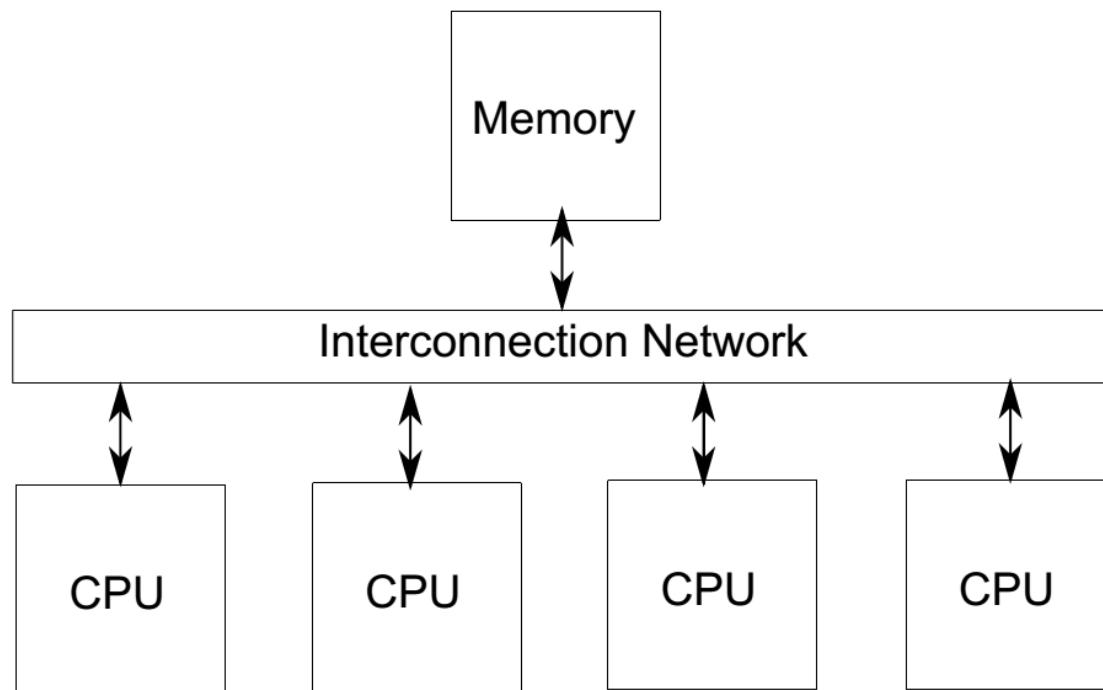
Chips con dos o más CPUs, gracias a la tecnología VLSI —multi-procesador en un chip**:**

- servidores de muy alto desempeño (p.ej., “granja” de servidores Web)
- productos electrónicos de consumo habitual
- comparten caché de nivel 2 y la memoria principal

Multiprocesadores UMA (uniform memory access). Los multiprocesadores más simples se basan en un único bus:

- dos o más CPUs

... y uno o más módulos de memoria compartida usan el mismo bus para comunicarse —la *interconnection network* es simplemente un bus



Cuando una CPU quiere leer una palabra de memoria, mira a ver si el bus está desocupado:

- en ese caso, la CPU coloca la dirección de la palabra en el bus, pone en 1 las señales de control, y espera hasta que la memoria coloque la palabra en el bus
- si el bus está ocupado, la CPU espera hasta que se desocupe

Con dos CPUs, la competición por el bus es manejable

... con 32 o 64, no:

- el sistema estará limitado por el ancho de banda del bus
- la mayoría de las CPUs estarán sin hacer nada la mayor parte del tiempo

Podemos agregar una caché a cada CPU:

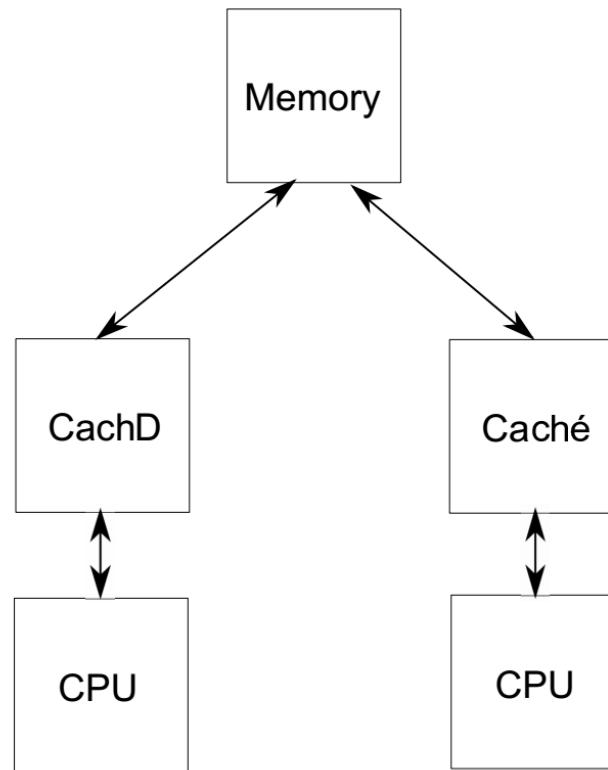
- muchas lecturas pueden hacerse ahora solo desde la caché local
 - menos tráfico en el bus
 - podemos tener más CPUs
- ... pero mantener las cachés consistentes entre ellas no es fácil**

Podemos agregar a cada CPU memoria privada (además de la caché), a través de un bus dedicado (privado):

- el compilador debería colocar el texto del programa, strings, constantes y cualquier dato de tipo *read-only*, stacks y variables locales en la memoria privada
- la memoria compartida es solo para variables compartidas

¿Qué pasa si la CPU 1 tiene una línea en su caché y la CPU 2 trata de leer una palabra que está en la misma línea?

- la CPU 2 debería obtener una copia de la línea en su caché
- en principio, que la misma línea esté en dos cachés es aceptable



¿Y si la CPU 1 hace un cambio en la línea
... y justo después la CPU 2 lee su copia de la línea desde su
caché?

- la CPU 2 obtiene un dato obsoleto
- **problema de coherencia (o consistencia) de cachés**

El problema de coherencia de cachés es serio

Sin una solución, las cachés no podrían usarse

... y los procesadores basados en un bus estarían limitados a unas pocas CPUs

Para resolverlo, se proponen **protocolos de coherencia de cachés**:

- conjuntos de reglas implementadas por las cachés, CPUs y memoria

... que previenen que versiones diferentes de la misma línea aparezcan simultáneamente en dos o más cachés

En todos los casos, el controlador de la caché examina el bus

... monitoreando todas las solicitudes hechas por otras CPUs y
cachés

... y actuando en ciertos casos → *snooping* cachés

El protocolo *write-through*. Cuando la CPU trata de leer una palabra que no está en la caché

... su controlador de caché lee la línea (que contiene la palabra) desde la memoria (que está siempre actualizada) y la pone en la caché

Lecturas subsecuentes se hacen desde la caché

...

...

Si la CPU trata de escribir una palabra que no está en la caché (un *write miss*)

... entonces la palabra es escrita en la memoria, pero la línea no es traída a la caché

Si la palabra está en la caché (un *write hit*)

... entonces la caché es actualizada y la palabra además es escrita en la memoria

¿Y desde el punto de vista de la caché *snoopy*, o *B* ?

Cuando la caché *A* produce un *read miss* → solicitud al bus para leer la línea → la caché *B* lo ve pero no hace nada

- en cambio, si la caché *A* produce un *read hit*, la caché *B* no se entera

...

...

Si la CPU A escribe —y ya sea hay un *hit* o un *miss*— su caché va a poner una solicitud de escritura en el bus

... la caché *B* mira a ver si tiene la palabra

Si no la tiene, no hace nada

... pero si la tiene, entonces su información acaba de quedar obsoleta

... por lo que marca la línea como inválida

Así, en el protocolo *write-through* cuando una palabra es escrita

- ... se la actualiza en la caché de la CPU que hace la escritura (si es que está en una línea de esa caché)
 - ... y en la memoria
 - ... y se la borra de todas las otras cachés

Variaciones:

- en un *write hit*, la caché B podría aceptar el valor y actualizar su contenido (en lugar de invalidar la línea) —conceptualmente, actualizar la caché es lo mismo que invalidar la línea seguido de leer la línea desde la memoria
- en un *write miss*, la caché B podría actualizar su contenido (en lugar de no hacer nada) —política *write-allocate*

En todos los protocolos, hay que elegir entre una **estrategia de actualización** y una **estrategia de invalidación**:

- se comportan differently bajo diferentes cargas
- las actualizaciones son más costosas, pero pueden prevenir caché *misses* en el futuro

El protocolo *write-through* es simple pero ineficiente:

- todos los *writes* van a memoria a través del bus
- ... por lo que el bus se vuelve un cuello de botella

Los protocolos alternativos —llamados ***write-back***— tienen la propiedad de que no todos *writes* van directamente a memoria:

- cuando una línea de caché es modificada, se coloca un bit de la caché en 1 indicando que la línea de la caché está correcta pero la memoria no
- al final, la línea tiene que ser escrita en la memoria, pero posiblemente después de varios *writes*
- un protocolo *write-back* popular es *MESI*

El protocolo MESI, del Core i7. Cada línea de la caché puede estar en uno de cuatro estados:

- *modified (M), exclusive (E), shared (S), invalid (I)*

Cuando la CPU parte, todas las entradas de su caché están en *I*

La primera vez que se lee la memoria, la línea referenciada se lleva a la caché de la CPU que hizo la lectura y se la marca como *E*:

- es la única caché que tiene la línea, y la memoria está actualizada

...

...

Si otra CPU lee la misma línea (a su caché), la anterior ve que ya no la tiene exclusivamente y avisa que tiene una copia

... de modo que ambas copias son marcadas como *S*:

- la línea está en una o más cachés para ser leída y la memoria está actualizada
- lecturas subsecuentes hechas por la CPU a una línea en su caché en estado *S* no usan el bus ni hacen que cambie el estado

...

...

Si esta CPU escribe en esta línea, coloca una señal de invalidación en el bus, para que las otras CPUs descarten sus copias

... y su propia copia queda en estado *M*, sin actualizar la memoria:

- soy la única CPU que tiene esta línea válida, y la memoria está desactualizada

(por supuesto, si una línea está en estado *E* cuando es escrita, no es necesario invalidar las otras cachés)

Si una tercera CPU lee la línea, la CPU anterior (que tiene la línea actualizada) le pide que espere mientras actualiza la memoria:

- cuando la memoria queda actualizada, esta tercera CPU lee la línea y ambas CPUs marcan la línea como *S*
- si cualquiera de estas CPUs escribe la línea nuevamente, invalida la copia en la caché de la otra y deja la propia como *M*

Si ahora la primera CPU escribe una palabra de esta línea (en la memoria)

... la CPU que tiene la línea en estado *M* le pide que espere mientras la escribe en la memoria:

- al terminar, esta CPU marca su línea como *I* (anticipando el cambio que va a hacer la primera CPU)

La primera CPU, entonces, va a escribir una línea que no está en ninguna caché:

- si se emplea la política *write-allocate** , la línea primero se carga en la caché y, una vez escrita, se la marca como *M*

* *write-allocate*: en un *write-miss*, la CPU primero trae a su caché la línea que va a escribir

El uso de un único bus limita el tamaño de un multiprocesador UMA a unas 32 CPUs

Multiprocesadores NUMA. Para multiprocesadores con más de 100 CPUs, abandonamos la idea de que todos los módulos de memoria tengan el mismo tiempo de acceso:

- *nonuniform memory access*

Los multiprocesadores **NUMA** también tienen un único espacio de direcciones para todas las CPUs

... pero el acceso a los módulos de memoria locales es más rápido que el acceso a los remotos:

- todos los programas UMA corren sin cambios en máquinas NUMA,
 - ... pero el desempeño es peor que en una máquina UMA a la misma velocidad de reloj

Características —en conjunto, distintivas— de las máquinas NUMA:

- único espacio de direcciones visible pra todas las CPUs
- acceso a memoria remota usando instrucciones LOAD y STORE
- **acceso a memoria remota es más lento que acceso a memoria local**

Dos tipos de máquinas NUMA:

- NC-NUMA (sin cachés)
- CC-NUMA (con cachés coherentes)

P.ej., una máquina con múltiples CPUs,

... cada una conectada a una memoria local a través de un bus local,

... y todas conectadas entre ellas mediante un bus del sistema, sin cachés —**NC-NUMA**

Cuando una solicitud de acceso a la memoria llega a la MMU, se revisa a ver si la palabra requerida está en la memoria local:

- en ese caso, se envía la solicitud por el bus local
- de lo contrario, la solicitud se envía a través del bus del sistema al componente (CPU+memoria) que tiene la palabra (y se demora mucho más en ser respondida que en el caso local)

No hay cachés → la coherencia de memoria está garantizada:

- cada palabra está en exactamente una localidad (por lo que no hay peligro de copias obsoletas: no hay copias)

Es muy importante cuál página está en cuál memoria:

- se requiere software especial para mover las páginas de una memoria a otra para maximizar el desempeño

P.ej., un proceso llamado *page scanner*, que corre cada unos pocos segundos examina las estadísticas de uso:

- si una página parece estar en el lugar equivocado, la página es invalidada, de modo que la próxima referencia a ella produzca un *page fault*
- cuando ocurre un *page fault*, se toma la decisión de dónde poner la página, posiblemente en una otra memoria
- para evitar *thrashing*, una página puesta en una memoria permanece allí por lo menos un determinado tiempo

Ningún algoritmo particular de *page scanner* es mejor siempre:

- el mejor desempeño depende de la aplicación

Debido a la ausencia de cachés, estos sistemas no escalan bien

Sistemas ***CC-NUMA*** —también llamados **sistemas de memoria compartida distribuida por hardware**:

- al tener cachés, también deben tener coherencia de cachés
- *snooping cachés + bus de sistema* es técnicamente simple, pero, de nuevo, no escala bien

La solución actual es usar un **directorio**:

- una base de datos que dice dónde está cada línea de la caché y cuál es su estado (válida o no)
- consultado por cada instrucción que hace referencia a la memoria → almacenado en hardware especializado extremadamente rápido

Las diapositivas # 66 a 69 ejemplifican el funcionamiento del directorio

P.ej., un sistema con 256 ($= 2^8$) nodos, conectados mediante alguna red de conexión: *grid*, hypercubo, etc.

Cada nodo tiene una CPU y 16 MB ($= 2^{24}$ bytes) de RAM, conectados mediante un bus local

La memoria total del sistema es de 2^{32} ($= 2^8 \times 2^{24}$) bytes

... y la dividimos en 2^{26} líneas de caché, de 64 bytes cada una

Cada nodo también tiene las entradas del directorio para las 2^{18} líneas de caché de su memoria de 2^{24} bytes

¿Qué pasa con una instrucción LOAD de la CPU del nodo 20 que hace referencia a una línea de memoria?

- la CPU 20 pasa la instrucción a su MMU, que la traduce a una dirección física —p.ej., 0x24000108— compuesta por tres campos:
 - ... el nodo 36 (8 bits)
 - ... la línea 4 (18 bits)
 - ... un *offset* de 8 (6 bits)
- la MMU envía una solicitud al nodo 36 preguntando si su línea 4 está en alguna caché, y en cuál

Cuando la solicitud llega al nodo 36, pasa al directorio local:

- el hardware mira su tabla de 2^{18} entradas y, digamos, ve que la línea 4 no está en ninguna caché (bit de presencia = 0)
 - ... por lo que la va a buscar a su memoria local
 - ... se la envía al nodo 20
 - ... y actualiza su tabla → la línea 4 está ahora en la caché del nodo 20

¿Y si luego llega una segunda solicitud al nodo 36 preguntando por su línea 2?

- el hardware mira nuevamente su tabla

... ve, digamos, que la línea 2 está en la caché (bit de presencia = 1) del nodo 82

... actualiza su tabla → la línea 2 está ahora en el nodo 20

... y envía un mensaje al nodo 82 para que éste le envíe la línea al nodo 20 e invalide la línea en su caché

II-B)

Otra arquitectura paralela posible es una en que cada CPU tiene su propia memoria local privada

... y, consecuentemente, su propio espacio físico de direcciones — **multicomputador o sistema de memoria distribuida:**

- la CPU tiene acceso a su memoria mediante las instrucciones LOAD y STORE
- **ninguna otra CPU en el sistema tiene acceso a esta memoria**
- p.ej., el cluster de Google

Las CPUs de un multicomputador no se pueden comunicar escribiendo y leyendo una memoria común (como en un multiprocesador)

... se comunican **pasándose mensajes** a través de una red de interconexión

Implicaciones para el software debido a la ausencia de memoria compartida por hardware:

- si la CPU 0 descubre —de alguna manera— que parte de los datos que necesita están en la memoria de la CPU 1
 - ... le **envía un mensaje** a la CPU 1 solicitándole una copia de los datos
 - ... y **se bloquea** a la espera de que la solicitud sea respondida
- cuando la CPU 1 recibe el mensaje, el software en esa CPU primero analiza el mensaje y luego envía los datos solicitados
- cuando el mensaje de respuesta llega a la CPU 0, el software en esta CPU es desbloqueado y continúa ejecutando

La comunicación entre los procesos normalmente usa operaciones básicas de software (o *primitvas* de software):

- **send**
- **receive**

... por lo que el software es más complicado que en un multiprocesador (en que simplemente usa LOAD y STORE)

Pero ...

... multicomputadores grandes, de miles de CPUs, son más simples y baratos de construir que multiprocesadores con el mismo número de CPUs:

- un multicomputador puede llegar a tener 65,536 CPUs
- además, la competición por memoria en un multiprocesador puede afectar seriamente su desempeño —difícilmente tiene más de 128 CPUs

Cada nodo en un multicomputador:

- una o unas pocas CPUs
- memoria RAM (compartida por las CPUs en ese nodo solamente)
- disco y otros dispositivos de I/O
- interconexión local entre las CPUs, memoria, disco y dispositivos de I/O
- un procesador de comunicaciones conectado a una red de interconexión de alto desempeño

Los procesadores de comunicaciones están conectados a través de una red de interconexión de alta velocidad:

- la topología puede ser muy variada —estrella, árbol, anillo, grilla, cubo, hypercubo, doble toroide, etc.
- cuando una aplicación ejecuta la primitiva **send** (en alguna CPU)
... el procesador de comunicaciones es notificado y transmite un bloque de datos al nodo destino

Las redes de interconexión, para multicomputadores (para conectar las CPUs entre ellas)

... y para multiprocesadores (para conectar las CPUs a los módulos de memoria), son similares:

- en el fondo, ambas usan paso de mensajes
- en una máquina con una sola CPU, cuando el procesador quiere leer o escribir una palabra, coloca en 1 ciertas líneas del bus —i.e., envía un *request*— y espera la respuesta
- en multiprocesadores grandes, la comunicación entre una CPU y memoria remota consiste en que la CPU envía un mensaje explícito—un paquete— a la memoria solicitando algunos datos y la memoria envía de vuelta un paquete de respuesta

La programación de un multicomputador requiere software especial —librerías— para manejar la comunicación y sincronización entre procesos

En un sistema de paso de mensajes, dos o más procesos corren independientemente:

- **paso de mensajes sincrónico** —el emisor permanece bloqueado hasta que el receptor haya recibido el mensaje
 - ... cuando el emisor es reanudado, sabe que el mensaje fue recibido correctamente
- **paso de mensajes basado en *buffers*** —el emisor puede continuar inmediatamente después de enviar el mensaje, independientemente del estado del receptor
 - ... pero no tiene garantía de que el mensaje fue recibido correctamente

¿Cómo escribimos programas para arquitecturas de memoria distribuida?

Hay que definir una interfaz con la *red de interconexión*

Definiremos **operaciones de red** que incluyan sincronización:

- operaciones básicas de *paso de mensajes*

Los procesos compartirán **canales**:

- rutas de comunicación entre procesos
- un canal es una abstracción de una red
- **un canal es una cola de mensajes enviados pero aún no recibidos**

Declaración de un canal:

channel <ch>(<tp₁> <id₁>, ..., <tp_n> <id_n>)

- <ch> es el nombre del canal
- <tp_i> y <id_i> son los tipos (obligatorios) y los nombres (opcionales) de los campos del mensaje transmitido

P.ej.,

- channel input(char)
- channel diskAccess(int cylinder, int block, int count, char* buffer)
- channel[n] result(int) — arreglo de canales

Un proceso envía un mensaje al canal ch ejecutando

send ch(<expr₁>, ..., <expr_n>)

- <expr_i> son expresiones cuyos tipos deben corresponder con los tipos de los campos en la declaración de ch

El efecto de ejecutar **send** es

- evaluar las expresiones
- agregar un mensaje con estos valores al final de la cola asociada con el canal **ch**

La cola es conceptualmente ilimitada,

... por lo tanto la ejecución de **send** no produce demora:

- **send** es una operación **no bloqueante**

Un proceso recibe un mensaje desde el canal ch ejecutando

receive ch(<var₁>, ..., <var_n>)

- <var_i> son variables cuyos tipos deben corresponder con los tipos de los campos en la declaración de ch

El efecto de ejecutar **receive** :

- el proceso se suspende hasta que haya al menos un mensaje en la cola del canal **ch**
- entonces el proceso saca el mensaje que está al comienzo de la cola y asigna sus campos a los <var_i>

receive es una operación **bloqueante**:

- el proceso no necesita usar espera ocupada

Los canales se comportan bien

El acceso a un canal es ininterrumpible (atómico)

La entrega de un mensaje es confiable y libre de errores

Todo mensaje que es enviado a un canal es entregado, y sin ser corrompido

Los canales son colas FIFO:

- los mensajes son recibidos en el mismo orden en que fueron agregados al canal

Ejemplo: Un proceso recibe caracteres, los agrupa en líneas y envía las líneas

```
channel input(char), output(char[MAXLN])

proc CharToLine:
    char[] line = new char[MAXLN]
    int i = 0
    while (true):
        receive input(line[i])
        while (line[i] != CR && i < MAXLN-1):
            i++
            receive input(line[i])
        line[i] = EOL
        send output(line)
        i = 0
```

(CharToLine es un proceso de tipo **filtro**:

- recibe datos por canales de entrada, los procesa, y envía el resultado del procesamiento por canales de salida)

Un proceso puede querer hacer algo si no hay mensajes disponibles

empty(<ch>)

- devuelve **true** si el canal **<ch>** no contiene mensajes
- en otro caso, devuelve **false**

Precauciones:

- si un proceso llama a **empty** y obtiene **true**, puede que haya mensajes en la cola cuando el proceso continúe su ejecución
- si un proceso llama a **empty** y obtiene **false**, puede que no haya mensajes en la cola cuando el proceso trate de recibir uno

Ejemplo: Ordenar n números

proc Sort:

*recibir todos los números desde el canal de input
ordenar los números
enviar los números ordenados al canal de output*

Como **receive** es bloqueante, **Sort** debe poder determinar cuándo ha recibido todos los números:

- incluir n o un centinela en los datos de entrada

(**Sort** también es un proceso de tipo filtro)

Otra posibilidad: Ordenación por mezclas sucesivas

Repetidamente, y en paralelo, mezclar dos listas ordenadas en una lista ordenada más larga

La red es construida a partir de procesos —filtros— **Merge**:

- cada filtro recibe valores desde dos secuencias de entrada ordenadas, in1 e in2, y produce una secuencia de salida ordenada, out
- en particular, cada filtro repetidamente compara los dos próximos valores recibidos desde in1 e in2, y envía el menor a out

Código de cada proceso Merge y declaración de canales

```
channel in1(int), in2(int), out(int)

proc Merge:
    int v1, v2
    receive in1(v1); receive in2(v2)
    while (v1 != EOS && v2 != EOS):
        if (v1 <= v2):
            send out(v1); receive in1(v1)
        else:
            send out(v2); receive in2(v2)
    if (v1 == EOS):
        while (v2 != EOS):
            send out(v2); receive in2(v2)
    else:
        while (v1 != EOS):
            send out(v1); receive in1(v1)
    send out(EOS)
```

Formamos una red de ordenación con procesos Merge y canales de entrada y salida

Disponemos los procesos y los canales en la forma de un árbol binario:

- empleamos $n-1$ procesos **Merge**
- el árbol tiene $\log_2 n$ niveles

Los canales deben ser compartidos:

- el canal de salida de un proceso debe ser el mismo que uno de los canales de entrada de otro proceso

Dos ventajas de los procesos tipo filtro: Interconectividad y reemplazabilidad

Pueden ser interconectados de diversas formas:

- sólo se requiere que la salida de un filtro cumpla las suposiciones de entrada de otro filtro

Si los comportamientos de entrada y salida observables externamente son los mismos, podemos reemplazar un filtro —o red de filtros— por un proceso o una red diferente:

- p.ej., podemos reemplazar el proceso **Sort** por una red de procesos **Merge** más un proceso que distribuya los valores de entrada a la red

Otro modelo: *Pares interactuantes* y el problema de intercambiar valores

Hay n procesos,

... cada uno tiene un valor local v ,

... y el objetivo es que cada proceso sepa cuál es el menor y cuál es el mayor de los n valores

Tres posibles patrones de comunicación:

- centralizado (proceso coordinador)
- simétrico
- anillo

Solución centralizada

Un proceso (coordinador) junta los n valores,

... calcula el mínimo y el máximo de ellos,

... y envía los resultados a los otros procesos

Usa $2(n-1)$ mensajes (sólo n , si hay *broadcast*)

```
channel values(int)
channel[n] results(int, int)

process P[0]: —proceso coordinador
    int v —lo suponemos inicializado
    int new, smallest = v, largest = v
    for (j = 1 ... n):
        receive values(new)
        if (new < smallest):
            smallest = new
        if (new > largest):
            largest = new
    for (j = 1 ... n):
        send results[j](smallest, largest)

process P[k = 1 ... n-1]:
    int v —lo suponemos inicializado
    int smallest, largest
    send values(v)
    receive results[k](smallest, largest)
```

Solución simétrica

Cada proceso ejecuta el mismo algoritmo:

- primero envía su valor local a todos los otros,
- ... y luego calcula el mínimo y el máximo de los n valores, a medida que va recibiendo los valores de los otros procesos

Usa $n(n-1)$ mensajes (sólo n , si hay *broadcast*)

```
channel[n] values(int)

process P[k = 0 ... n-1]:
    int v —lo suponemos inicializado
    int new, smallest = v, largest = v
    for (j = 0 ... n):
        send values[j](v)
    for (j = 1 ... n):
        receive values[k](new)
        if (new < smallest):
            smallest = new
        if (new > largest):
            largest = new
```

Solución basada en un anillo lógico

Cada proceso recibe mensajes de su predecesor y envía mensajes a su sucesor

Cada proceso, primero, recibe dos valores, calcula el mínimo y el máximo de estos junto a su propio valor, y envía los resultados

... luego, recibe el mínimo y máximo globales y los envía

Usa $2(n-1)$ mensajes

```
channel[n] values(int smlst, int lrgst)

process P[0]:
    int v —lo suponemos inicializado
    int smlst = v, lrgst = v
    send values[1](smlst, lrgst)
    receive values[0](smlst, lrgst)
    send values[1](smlst, lrgst)

process P[k = 1 ... n-1]:
    int v —lo suponemos inicializado
    int smlst, largest
    receive values[k](smlst, lrgst)
    if (v < smlst):
        smlst = v
    if (v > lrgst):
        lrgst = v
    send values[k+1](smlst, lrgst)
    receive values[k](smlst, lrgst)
    if (k < n-1):
        send values[k+1](smlst, lrgst)
```