Apuntes AC

Tema 1 - Rendimiento / Consumo / Coste

Def. Ley de Moore: El número de transistores de una maquina se dobla cada 2 años, 52 años después (actualmente) se sigue cumpliendo.

Esta ley conlleva mejoras tanto en el rendimiento como en la frecuencia (esta última debido a dos motivos: 1 transistores más pequeños, 2 mejoras micro arquitectónicas).

Problemas de la Ley de Moore:

- **Consumo energético**: Con el paso de los años los nuevos procesadores son más rápidos pero van elevando el consumo hasta límites que superan los 120 W, con los elementos de refrigeración actuales no era posible disipar adecuadamente el calor del chip y se acababa quemando.

- **Retardo interconexiones**: Según avanza la tecnología:
    - El transistor cada vez tarda menos tiempo en hacer la conmutación, pero la interconexión no sigue esta evolución y se queda atrás comparado con la mejora de la tecnología, es decir, el retardo de la interconexión no escala con la tecnología, y va aumentando a medida que ésta mejora.
    - La fracción del chip a la que podemos llegar mandando información en un ciclo de reloj por interconexión es prácticamente nula. (porcentaje alcanzable del chip nulo, cada vez puedo cubrir menos área del chip).

¿Qué es lo que motivo el cambio en el paradigma de diseño de las industrias que desarrollaban chips?

A pesar de que la ley de Moore (que consistía en que el numero de transistores de una maquina se doblaba cada 2 años) se seguía cumpliendo 52 años después, y que conllevaba mejoras tanto en el rendimiento como en la frecuencia, esta ley tenía una serie de problemas que motivaron a la industria a adaptarse a este cambio de paradigma.
Por un lado existía el problema del consumo energético, que consistía en la imposibilidad de disipar el calor generado por los procesadores que cada vez eran más rápidos pero también consumían más a su vez.
Por otro también existía el problema del retardo de las interconexiones según el cual el retardo de la interconexión no escala con la tecnología, es decir, las interconexiones no evolucionan al mismo ritmo que los transistores (que hacen la conmutación cada vez más rápido) y se quedan atrás comparado con la mejora de la tecnología. También la fracción del chip a la que podemos llegar enviando información en un ciclo de reloj por interconexión es prácticamente nula.

Latencia y ancho de banda

Problema del *Memory Wall*: la CPU ha evolucionado a un ritmo mucho mayor que la memoria, a pesar de haber evolucionado esta también. El procesador es capaz de hacer peticiones mucho más rápido que la memoria es capaz de responder a esas peticiones, por lo cual existe un gap (hueco) entre ambos, ya que el ritmo de mejora de la CPU es mucho mayor.

Def. Ancho de banda: El número de trabajos que se pueden realizar por unidad de tiempo.

Def. Latencia: Tiempo comprendido desde el inicio hasta el final de un determinado evento.

Ancho de banda siempre es más alto que la latencia.

Rendimiento

Actualmente el rendimiento aumenta a un ritmo menor que antes debido a varios problemas:

- La disipación del calor (más cores tan o menos potentes como los anteriores).
- El agotamiento del paralelismo a nivel de instrucciones (ILP).
- La latencia de memoria.

"Desde 2003, la mejora del rendimiento de un solo procesador ha disminuido a menos de 22% por año debido a los obstáculos gemelos de la máxima disipación de potencia del aire chips refrigerados y la falta de más paralelismo de nivel de instrucción para explotarla eficientemente" Hennessy-Patterson

Medida fiable para el rendimiento: tiempo de ejecución de programas reales, que a su vez tiene dos aspectos:

- Rendimiento del procesador: $T_{CPU} = N * CPI * t$
  - *N* (Número de instrucciones)
  - t (periodo de reloj)
  - CPI (Ciclos medios por instrucción) $CPI = \frac{T_{CPU} * Frecuencia\ de\ reloj\ (\sim Ciclos)}{N}$

    CPI medida promedio, el de la formula es el GLOBAL (no el de un tipo de instr.)

    Frecuencia de reloj: Inversa del tiempo de ciclo.
- Rendimiento del computador: se mide mediante "*benchmarks*" que son programas de prueba que se diseñan específicamente para evaluar las prestaciones de los computadores, esto se hace mediante programas reales con cargas de trabajo fijas. Los SPEC son conjuntos de benchmarks destinados a evaluar el rendimiento de los computadores.

¿Cuál es la ecuación que nos da el número SPECxx que define el rendimiento?

La suite de programas SPECxx está formada por n programas patrón o aplicaciones, de esta manera definimos una serie de factores (r_i) que consisten en:

$$r_i = \frac{Tejec.patrón\ i\ en\ maquina\ de\ referencia}{T\ ejec.patrón\ i\ en\ maquina\ evaluada}\ ,\forall\ i = 1 \dots n$$

Una vez calculados todos los cocientes de cada uno de los programas entonces ya se define el nº SPECxx:

$$SPEC_{XX} = \sqrt[n]{r_1 * r_2 * \dots * r_n}$$

El número que obtengamos (media geométrica de los n ratios) indica su rendimiento, cuanto más alto sea el nº SPECxx mayor rendimiento tiene esa máquina que queremos evaluar.

Def. Productividad (throughput): Número de tareas que puede realizar una máquina por unidad de tiempo.

Def. Ley de Amdahl: Calcula ganancia en rendimiento (cuánto conseguimos reducir el tiempo de ejec. de un programa) cuando aplicamos una mejora sobre una fracción de tiempo muy concreta del programa original.

$$T_M = T_{SM}\left[(1-F) + \frac{F}{x}\right]$$ F (Fracción de mejora) x (veces más rápido)

Def. Speedup: $$\frac{T_{SM}}{T_M} = \frac{1}{(1-F) + \frac{F}{x}}$$

(A mayor tiempo de ejec. menor tiempo de rendimiento y viceversa)

Def. Eficiencia: $$E = \frac{Speedup}{x}$$ Valor máximo posible = 1.

Potencia y energía (consumo)

El consumo de potencia es uno de los principales retos en el diseño de procesadores, ya que existe el problema de que el consumo de los transistores se transforma en calor a disipar. Hay que tener en cuenta varios aspectos:

- Consumo máximo de potencia posible.
- Consumo de potencia sostenido (Consumo más habitual que se va a dar, está entre el consumo medio y el pico).
- Energía y eficiencia energética (Cuál es la métrica más adecuada para comparar el rendimiento energético de dos procesadores -> Energía vs. potencia).

Los objetivos al hablar de consumo de energía son autonomía, tamaño compacto, reducción de coste y consumo del sistema de refrigeración.

Tipos de energía que se consume:

- Energía dinámica: La que se consume en la conmutación de los transistores, asociada a su trabajo. La generan aparatos electrónicos funcionando.
- Energía estática: La que se consume aunque los transistores no estén conmutando, esta energía está asociada a las corrientes de fuga. La generan aparatos electrónicos cuando no están funcionando pero no están apagados.

La frecuencia de reloj de los procesadores desde 2003 crece a un ritmo extremadamente bajo debido a varios motivos:

- Ley de Moore
- Rendimiento
- Consumo
- Paso de un solo core a multicores por el problema de la disipación del calor.

Respecto a los años anteriores a 2003 se ha producido un frenazo en el aumento de la frecuencia de reloj de los procesadores.

Coste

El coste del circuito integrado es proporcional al área del circuito al cuadrado, cuanto más grande sea el circuito mayor va a ser el coste. Fundamentalmente el proceso de testeo y el proceso de empaquetado son la mayor parte del coste.

Para determinar si una máquina es mejor que otra es fundamental tener en cuenta la interrelación existente entre coste rendimiento y consumo, ya que si únicamente tenemos en cuenta uno de estos aspectos obtendremos resultados muy distintos.

Tema 2 - ILP, Planificación dinámica, predicción de saltos, especulación

Técnicas para mejorar el rendimiento de una máquina: Planificación dinámica, predicción de saltos y especulación, derivan del ILP (paralelismo a nivel de instrucción).

Def. Paralelismo: Capacidad de ejecutar de forma simultánea varias tareas.

El objetivo de todas las técnicas que vamos a estudiar es ejecutar el mayor nº de instr. por ciclo, o ejecutar en menor tiempo un programa determinado.

La idea es obtener el máximo número de instrucciones independientes, ya que estas se podrán ejecutar de forma solapada.

El CPI ideal en procesadores segmentados suele ser 1, ya que se espera que cada ciclo de reloj termine una instr. y comience la ejec. de una nueva (sin paradas ni conflictos).

Para los distintos conflictos o dependencias que se pueden dar conocemos unas técnicas para tratarlos:

- Conflictos de recursos (riesgos estructurales): técnicas de replicación y segmentación para evitarlos.
- Dependencias de datos: mediante cortocircuitos (bypassing).
- Dependencias de control: se usan saltos retardados ("mover" instr. en los ciclos que tarda el salto en resolverse para ejecutar instr. que sabemos seguro se van a ejecutar).

Mecanismos para explotar ISP:

- Basados en HW en tiempo de ejecución (dinámicos): disponen de todos los valores que se van generando en la propia ejecución del código.
- Basados en SW en tiempos de compilación (estáticos): Las dependencias de memoria son difíciles de determinar.

Def. Paralelismo a nivel de instr. (ILP): Técnica consistente en explotar paralelismo entre instrucciones próximas en la secuencia.
Def. Bloque básico (BB): Conjunto de instr. que no tiene saltos.

Las dependencias son propias de los programas. Es importante diferenciar entre dependencia y riesgo, dependencia es algo "matemático", por ejemplo una instr. genera un dato que necesita otra instrucción que viene después. Un riesgo es una dependencia que ocasiona parada. Si se tiene un código con pocas dependencias se va a poder obtener un mayor beneficio en la ejec. del mismo una vez se haya planificado.

Existen distintos tipos de dependencias:

- Dependencias de datos
  - Dependencia verdadera (LDE) (se llama así porque no se puede evitar).
    instr1: **L.D F0, 0(R1)** Escribe en **F0** el contenido de la posición de memoria **0(R1)**
    instr2: **ADD.D F4, F0, F2** Escribe en **F4** la suma de **F0** y **F2**
    La instr1 produce un resultado que usa la instr2 (**F0**).
  - Dependencia de nombre (reutilización de registros)
    - Antidependencia (EDL)

    instr1: **ADD.D F4, F0, F2** Escribe en **F4** la suma de **F0** y **F2**)
    instr2: **L.D F0, 0(R1)** Escribe en **F0** el contenido de la posición de memoria **0(R1)**
    La instr2 escribe antes de que la instr1 lea.
    - Dependencia de salida (EDE)

    instr1: **ADD.D F4, F0, F2** (Escribe en **F4** la suma de **F0** y **F2**)
    instr2: **SUB.D F4, F3, F2** (Escribe en **F4** la resta de **F3** menos **F2**)
    Las instr1 e instr2 escriben en el mismo reg. o memoria.

ILP y dependencias de datos: Los mecanismos de ejecución deben preservar el comportamiento del programa, mismo resultado que en ejec. secuencial. Lo ideal es explotar todo el paralelismo posible sin afectar al resultado de la ejec. Las dependencias de nombre se eliminan renombrando registros.

- Dependencias de control: Cada instr. depende de un cjto. de saltos, son asociadas a los datos. Al contrario que el otro tipo de dependencias, estas si pueden violarse siempre y cuando no afecte al resultado correcto del programa, lo importante es que el comportamiento de las excepciones y el flujo de datos deben preservarse gestionándose en el mismo orden que si el código no estuviera planificado.

  instr1: **DADDU R1, R2, R3**
  instr2: **BEQZ R4, L1**
  instr3: **DSUBU R1, R5, R6**
  L1: --- ---
  instr4: **OR R7, R1, R8**

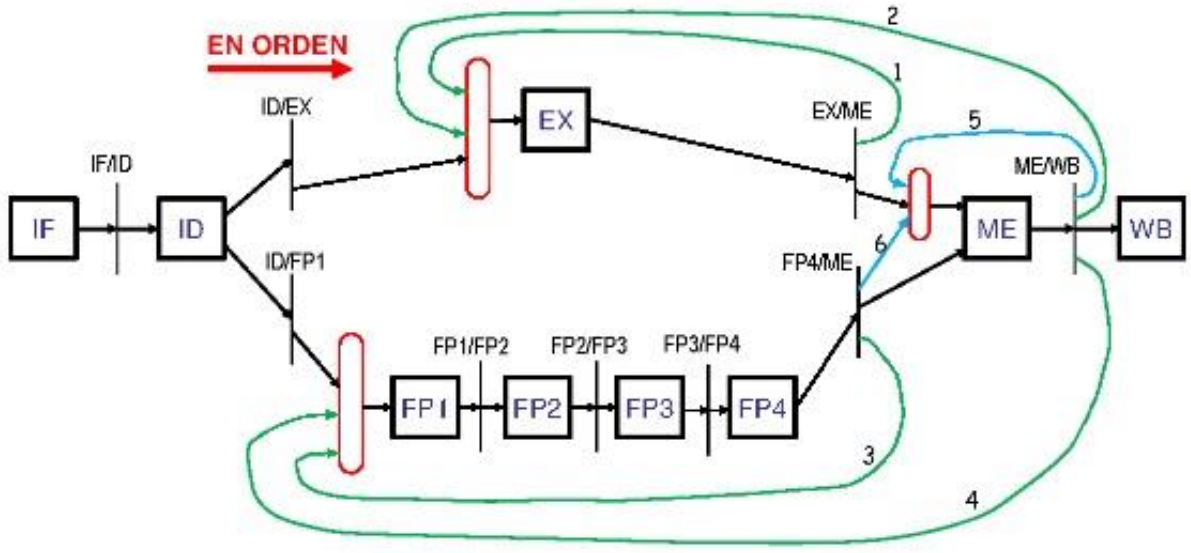
La instr4 usa el resultado de la instr1 o la instr3 dependiendo del comportamiento del salto, el flujo se debe preservar.

Las dependencias de datos son fáciles de determinar para los registros, ya que comparten el mismo registro en el que se lee o se escribe, no ocurre lo mismo para la memoria, por ejemplo cuando hay dos stores intentando acceder a la misma región de memoria, en estos casos es más complicado determinar estas dependencias.

Respetando dos reglas muy concretas aseguramos evitar dependencias de control:

- Una instr. dependiente de un salto no puede moverse antes del salto.
- Una instr. no dependiente de un salto no puede moverse después de un salto.

Técnicas SW para explotar ILP



Red de anticipación de operandos (by-pass)

1: Resultado de op int
2: Res. de op int o load INT
3: Res. de op FP
4: Resultado de op FP o load FP
5: Res. de op int, load INT → store int
Res. de op FP, load FP → store FP
6: Res. de op FP → store FP

5 etapas básicas del procesador MIPS:

- **Etapa fetch**
- **Etapa decodificación** (Detección de posibles riesgos, los saltos también se resuelven en esta etapa)
- **Etapa de ejecución** (una UF para instr. enteras y otra UF segmentada en 4 para instr. punto flotante)
  Hasta la etapa de ejecucion el programa se ejecuta en orden, es posible que esta etapa se finalice en desorden (que la segunda instr. acabe antes de la primera).
  Hacen uso de la etapa de ejecución de enteros (EX): load, store, e instr. aritméticas de lógica entera, ya que son registros enteros, no en punto flotante, todas las demás instrucciones hacen uso de la etapa de ejecución de FP.
- **Etapa de memoria.**
- **Etapa de write**: se escribe el valor de los registros modificados (instr. aritméticas y también load), los stores no hacen nada en la etapa de write.

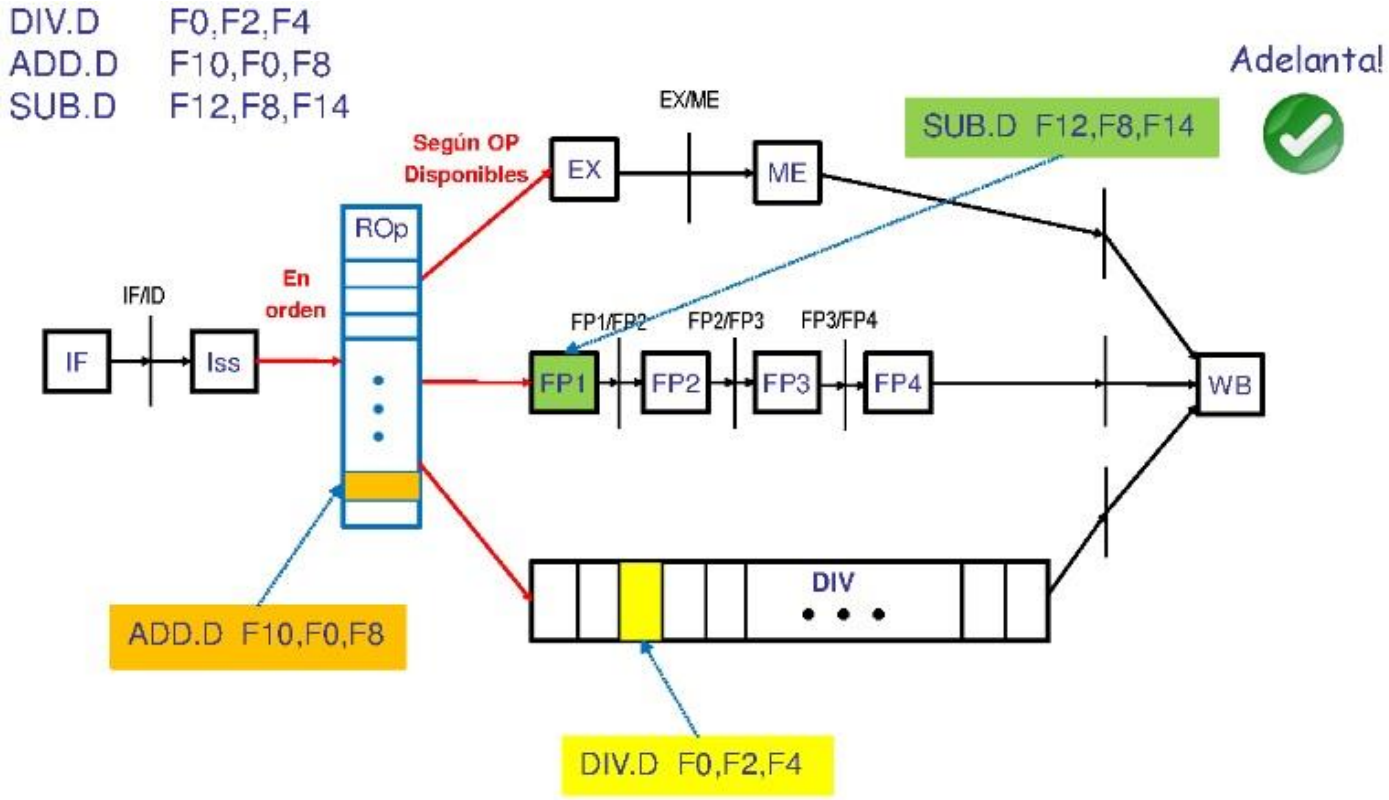
Existen distintas técnicas SW para explotar el ISP:

- **Planificación de instrucciones**: Consiste en planificar (reordenar) las instr. para intentar reducir el nº de ciclos (ocultar latencias).
- **Desenrollado de bucles**: Consiste en coger el cuerpo del bucle (las instr. que no pertenecen a la gestión de los índices) y repetirlo varias veces y situar al final las instr. de gestión de índices, de esta forma reducimos las iteraciones del bucle en un factor (depende del grado). Para usar esta técnica se necesita hacer uso de más registros (renombrado) para evitar dependencias de nombre. Se mantienen las dependencias y las paradas.
  Cabe la posibilidad de combinar ambas técnicas y se obtienen resultados muy notables, pudiendo alcanzar incluso el CPI ideal de 1, al no haber paradas. Este tipo de técnicas las lleva a cabo el compilador.
  Ventajas: Bloque grande para planificar (tenemos más instr. y más posibilidades para reordenar), también se reduce el nº de saltos.
  Desventajas: Incrementamos el tamaño del código, puede ser que tengamos que incluir iteraciones extra, uso de gran número de registros.
- **Software "pipelining"**: Consiste en reorganizar el bucle de manera que cada instr. pertenece a una iteración diferente. Se tiene una cabecera previa al bucle en la cual se hacen los LD y ADD de las primeras componentes, luego se ejecuta el bucle y finalmente se tiene una cola posterior en la cual se hacen los ST de las últimas componentes del bucle.
  Ventajas: se maximiza la distancia del momento en el que se genera un resultado y el momento en el que se usa, se consigue un tamaño menor del código respecto al desenrollado de bucles, se reduce siempre el número de ciclos de parada y además nunca hay dependencias en el cuerpo del bucle.
  Desventajas: No se reduce el nº de saltos y se necesita un inicio y finalización especial.

Las mejoras HW no necesitan una ejecución previa para poder aplicarse, se aplican directamente en tiempo de ejecución (principal diferencia respecto a las mejoras SW), para evitar esperas y bloqueos en el lanzamiento de instrucciones en orden se opta por una solución que consiste en dividir la etapa de ID (decodificación) en dos etapas diferenciadas:

- **Issue**: Decodifica y chequea riesgos estructurales.
- **Lectura de operandos**: Chequea la disponibilidad de operandos, debe implementarse para permitir el flujo de instr.

La ejecución fuera de orden implica finalización fuera de orden.

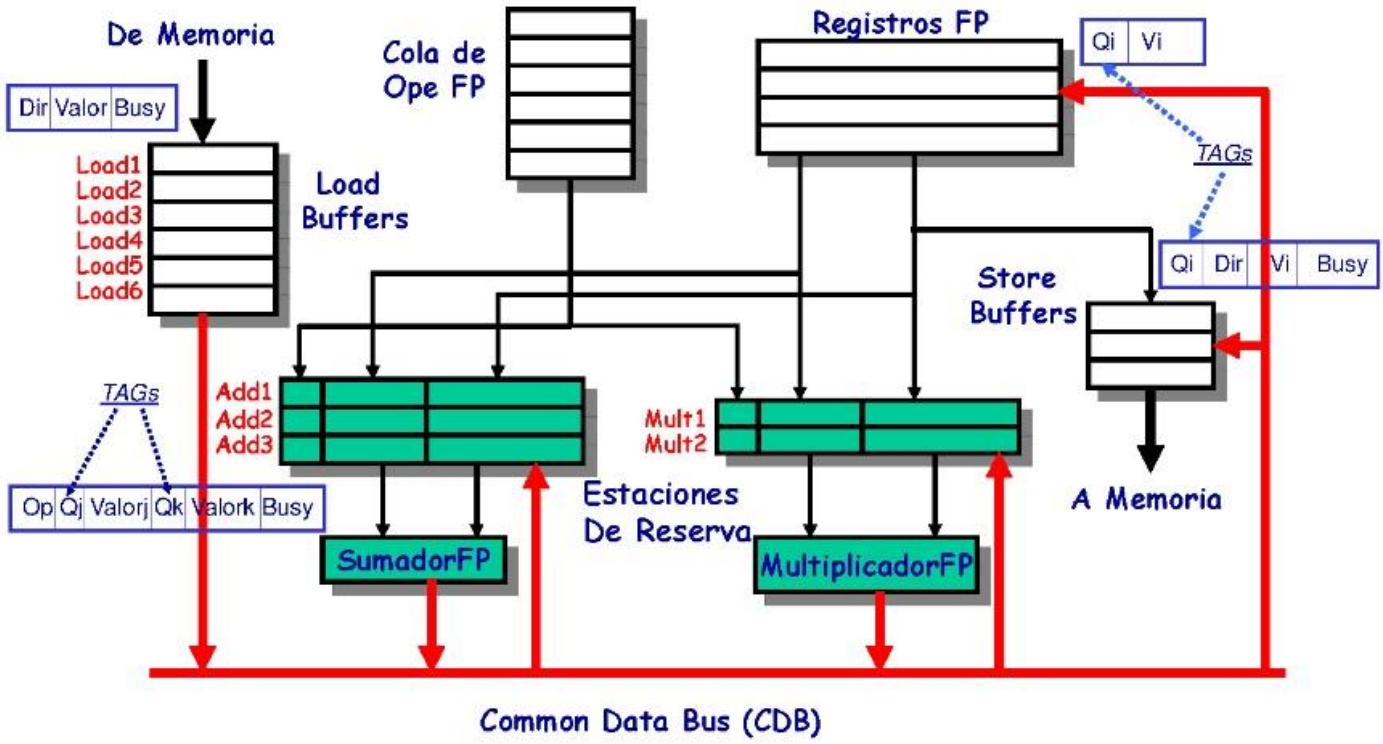


SUB.D acabará antes que DIV.D y que ADD.D: Se evita el bloqueo.

Gracias a la división en subfases de la fase de ID conseguimos que la instr. SUB.D adelante a la instr. ADD.D ya que no necesita el registro F0 pendiente de resolver por la instr. DIV.D.

No obstante si cambiamos el código es posible que se den riesgos de EDL (reg. destino de la resta F8) y que alguna instr. leyera un valor incorrecto, para evitar esto se hace uso del renombrado dinámico, que como su nombre indica renombra registros y guarda los valores antiguos en registros auxiliares, de esta manera solventamos las dependencias de nombre que se puedan dar (EDL y EDE).

La técnica que hace uso de estas ideas que hemos visto se conoce por el **algoritmo de Tomasulo**, que limita el efecto de las dependencias LDE, e introduce el renombrado dinámico de registros, eliminando de forma dinámica los riesgos EDL y EDE.



Excepciones imprecisas
Issue en orden-- Ejecución fuera de orden—Finalización fuera de orden

Este es el HW necesario para implementar el algoritmo de Tomasulo:

- Una cola de oper. FP donde se almacenan las instr en FP según se van trayendo de la caché.
- Estaciones de reserva (recuadros verdes), que son estructuras HW que sirven como un paso intermedio entre la cola de oper. en FP y las correspondientes UF.
  Las instr. se alojan en las estaciones a la espera de tener disponibilidad de sus operandos para poder entrar en sus correspondientes UF, cada ER tiene 6 campos en total (ocupado, operación (alojan distintos tipos de instr.), Qj, Vj, Qk y Vk).
  Las Q se refieren a las etiquetas o tag (que hace referencia al momento en el que todavía no está disponible el campo valor, me dice que instr, va a generar ese valor que necesito en base al identificador del campo de la estación de reserva donde está alojada esa instr.) mientras que los subindices hacen referencia a los operandos que necesita conocer de una instr. (reg. fuente), la "j" hace referencia al primer operando y la "k" hace referencia al segundo operando.
- loadBuffer y storeBuffer, que son estaciones de reserva para las instr. load y store respectivamente, con 3 campos para los load buffer (dirección, valor y ocupado) y 4 campos para los store buffer (campo Qi extra). Tenemos capacidad para 6 instr. de load y 3 de store, los load leen de la memoria (por eso no necesitan campo tag), los store escriben a memoria.
- Un registro para las instr. de FP donde llegan los datos generados en las correspondientes UF, tienen un campo Qi (tag) y otro Vi (valor). El dato se traslada a través del bus común de datos (CDB), que también se comunica con los store y los load buffers por si necesitan consultar algún tipo de dato (escribir o leer).

Es importante resaltar que este algoritmo no gestiona de forma precisa las excepciones (no se generaban en el mismo orden que si no se hubiese aplicado este algoritmo, ya que divide el procesamiento de cada instr. en 3 etapas (issue, ejecución y finalización) Este algoritmo solo hace en orden la etapa de Issue, las otras dos etapas pueden ser fuera de orden.

Ejecución del algoritmo de Tomasulo:

- **Fase Issue**: esta fase se ejecuta en orden siempre (no tiene por qué ser en ciclos consecutivos, pero si crecientes), en ella se toma la instr. de la cola de instr., la que toque, luego se envía a la ER correspondiente a su código de operación si hay entradas disponibles (incluidos load buffer y store buffer), si no está disponible ninguna (todas las entradas ocupadas) se produce un bloqueo (parada).
  También se envían los operandos si están disponibles, se chequea la disponibilidad de los operandos una vez se han alojado en la estación de reserva, y se copian el campo Tag y el campo valor de los dos registros fuentes sobre el campo tag y el campo valor de la estación de reserva, y finalmente se coge el tag del registro destino y se marca (se pone el identificador de la ER donde se aloja la instrucción).
- **Fase ejecución**: En esta fase se espera a que estén listos los operandos y cuando lo estén se mandan ejecutar, gestiona riesgos de LDE. No se realiza en orden.
- **Fase escritura**: En esta fase se pone en el bus común de datos el resultado generado y el identificador (nombre) de la ER donde estaba alojada la instr. y se vacía la ER donde estaba alojada. De esta manera no se chequean riesgos EDE ni EDL (eliminados por el renombrado, al copiar los campos tag y valor garantizamos que las dependencias de este tipo no afectan al comportamiento HW). En el bus común de datos nos interesa saber en qué ER estaba alojada la instr, porque vamos a tener otras instr esperando por el dato que se acaba de generar. Los store no realizan esta fase en este algoritmo.

Este algoritmo tiene una serie de ventajas e inconvenientes:

Ventajas: Elimina el cuello de botella de los registros, evita riesgos EDL y EDE al ir la fase de Issue en orden, permite el desenrolladlo en HW, no está limitado a bloques básicos si existe predicción de saltos.

Inconvenientes: Es complejo, el CDB limita el rendimiento y es impreciso con las excepciones.

¿Cómo hace el algoritmo de Tomasulo para evitar las dependencias de EDE y LDE? Pon un ejemplo.

Lo hace mediante el renombrado dinámico de registros que es capaz, en tiempo de ejecución, de (como su nombre indica) renombrar registros y guardar los valores antiguos en registros auxiliares, de esta manera se evitan riesgos y lecturas de valores erróneos, provocados por las dependencias.

Ejemplo:

Sin renombrado de registros:

instr1: **DIV.D F0,F2,F4**
instr2: **ADD.D F10,F0,F8**
instr3: **SUB.D F8,F8,F14**

Dep. EDL entre la instr1 e instr2 (**F8**)

Con renombrado de registros:

instr1: **DIV.D F0,F2,F4**
instr2: **ADD.D F10,F0,F8**
instr3: **SUB.D T40,F8,F14**

Dep. evitada por el uso del reg. auxiliar (**T40**)

Tratamiento de saltos: predicción

La idea básica de la predicción de saltos consiste en que cuando se detecta una instr. de salto condicional sin resolver se predice el camino del salto (Tomado o No tomado), si el salto se predice como tomado se predice la dirección destino del mismo y la ejecución continua de forma especulativa a lo largo del camino supuesto, después si la predicción fue correcta la ejecución se confirma y se continua normalmente, pero si en cambio la predicción fue incorrecta (fallo de predicción) se descartan todas las instr. ejecutadas especulativamente y se reanuda la ejecución a lo largo del camino correcto.

BTAC: Caché que almacena la dir. destino de los últimos saltos tomados, cuando se accede a una instr. se accede al mismo tiempo a la BTAC usando la dir. de la instr., si esa dir. está en la BTAC sabemos que es un salto, y si se predice como tomado la dir. destino del salto se lee de la BTAC.
Se actualiza cuando se conoce si un salto se ha tomado (se chequea si ya estaba (lo actualizamos si ha cambiado la dir. destino del salto) o no estaba (se introduce en la BTAC) o no se ha tomado (si estaba en la BTAC se elimina (ya no hay que almacenar esa dir. ya que en el futuro tampoco se tomara el salto)).
Existen diferentes diseños de BTAC:

- **Caché de acceso directo**: Cada salto solamente puede acceder a una única entrada de la tabla.
  Ventajas: Tiene menor coste.
  Desventajas: Aliasing (Destrucción de info. debido a la compartición de entradas entre celdas distintas.
- **Caché asociativa**: Cada salto puede acceder a un cjto de entradas.
  Ventajas: Tiene menor aliasing y menos conflictos.
  Desventajas: Tiene mayor coste HW.
- **Caché asociativa por conjuntos**: Cada salto puede acceder a cualquiera de las entradas, es una solución intermedia a las dos anteriores.

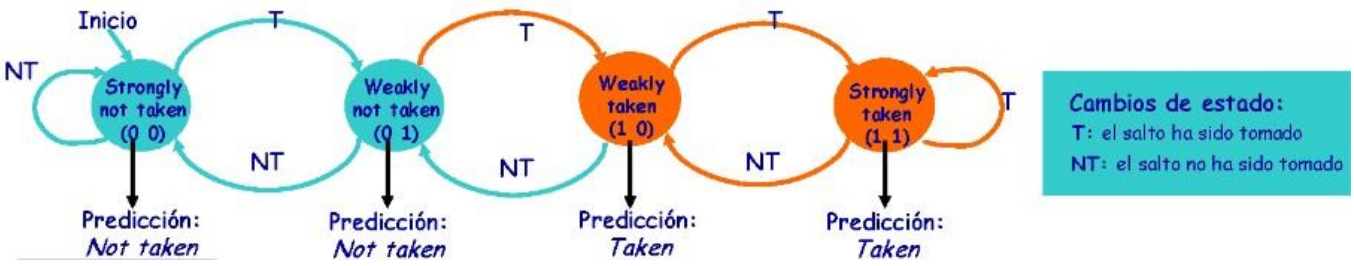
Principalmente hay dos tipos de técnicas de predicción de saltos:

- **Estática**:
  - Fija: Siempre predecimos lo mismo, se toman todos los saltos o no se toman. 75% de aciertos (una tasa pésima para un predictor de saltos, rentables a partir del 95%).
  - Basada en el sentido del salto: Si el salto va hacia una instr. anterior entonces se predice como tomado, si va a una posterior se predice como no tomado, se hace así porque la mayoría de saltos hacia atrás corresponden a bucles, y los saltos hacia delante corresponden a if-then-else. Mal comportamiento en general.
  - Basada en opcode: El salto se predice en función de la operación que tenga. Esto se hace porque la probabilidad de que un salto sea tomado depende del tipo de salto.
  - Dirigida por compilador:
    - Basada en tipo de instr: Dependiendo del tipo de instr. de control (For…)
    - Basada en profiling: Tomamos el salto o no en función de lo que ocurra en un ejec. de prueba.
    - Especificado por el programador: Programador indica si se toma el salto o no.

- **Dinámica**: Suele dar mejores resultados que la estática, se realiza observando el comportamiento de un salto en particular en las ejec. previas del programa. En función de cuanta cantidad de info. almacenemos distinguimos predictores de 1, 2 (bimodal) o 3 bits de historia.

    - Predictor de 1 bit: Usa 1 bit de historia por cada instr. de salto, ese bit refleja el comportamiento de la última ejec. de esa instr., con él podemos indicar si el salto se tomó (T) o no se tomó (NT) en la ejec. previa.
      La predicción consiste en que si en la última ejec. el salto fue T entonces en la siguiente se predecirá como T también, el comportamiento del salto siempre se va a predecir de la misma forma que se ha comportado en la ejec. anterior. Es similar al comportamiento de una máquina de dos estados, hay un reg. de historia (contador de 1 bit) y la predicción es el valor de ese registro.
      Este predictor tiene limitaciones, ya que solo se registra el comportamiento de la ejec. anterior. Esto ocasiona que las predicciones en los cambios de tendencia del salto sean malas, ya que siempre vamos a fallar en la predicción dos veces (si un salto se T siempre estaríamos siempre en el estado T, pero en una única ejec. el salto es NT, por lo que fallamos, la siguiente vez que se ejecute volverá a su comportamiento normal y se tomará (T), por lo que fallaremos otra vez al ser nuestra predicción NT, por la ejec. anterior insólita en la que no se tomó el salto).

    - Predictor bimodal (2 bits): Usa 2 bits por cada instr. de salto, refleja el comportamiento de las últimas ejecuciones de ese salto (no solo las dos últimas)
      Un salto que se toma repetidamente se predice como T y uno que no se toma repetidamente se predice como NT, si el salto se comporta de manera inusual una vez el predictor mantiene la predicción usual, ya que en vez de 2 estados ahora tenemos 4 estados, fuertemente no tomado (0,0), débilmente no tomado (0,1), débilmente tomado (1,0) y fuertemente tomado (1,1), el bit más significativo indica si se ha tomado (1) o no (0).
      La idea es que las líneas de transición nos mueven entre estados contiguos, por ejemplo de (0,1) si se actualiza nunca pasará a (1,1) en un solo ciclo.
      En el ejemplo anterior de los cambios de tendencia con este predictor fallamos solamente una vez, ya que rectificamos a tiempo para el segundo fallo (acierto en este caso).



    - Predictor de 3 bits: En este caso con este predictor tenemos 8 estados distintos. Funciona de la misma manera que los predictores anteriores.

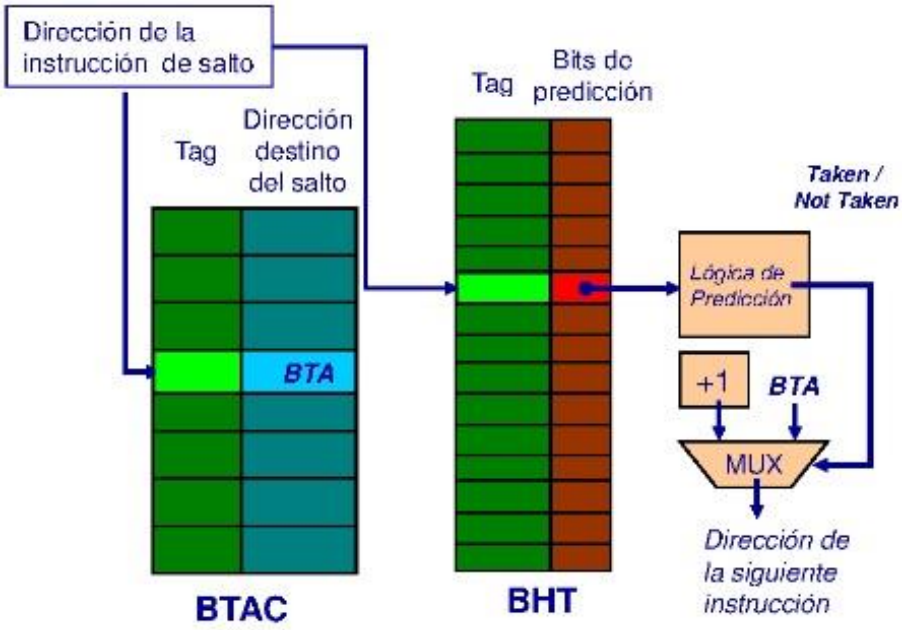
Se implementan los bits de predicción añadiéndolos a las BTAC, integrándolos en estructuras llamadas BTB o BHT.

- BTB: Similar a una BTAC pero con una columna añadida además de las que ya tenía (tag, dir. destino del salto (BTA)) que contiene los bits de predicción. La señal que indica T o NT viene de los bits de predicción asociados a esa entrada donde está alojado el salto que se quiere predecir, según la predicción cogemos la dir. destino del salto (BTA) o la siguiente instr. al salto.

  Independientemente de si el salto fue T o NT se actualiza la BTB, en concreto la columna con los bits de predicción y la BTA (dir. destino del salto (si ha cambiado)). También existe la predicción implícita, que consiste en una BTB en la que se quita la columna de los bits de predicción, pero implícitamente se supone una predicción, si la instr. de salto está en la BTB entonces se predice como T, sino se predice como NT, solo se puede aplicar usando un predictor de 1 bit.

- BHT: Estructura con una tabla distinta de la BTAC para almacenar los bits de predicción, la info. está desacoplada, se usa una tabla para la dir. destino (BTA) y otra para los bits de predicción (con mayor nº de entradas), la ventaja es que predice instr. que no están en la BTAC, pero también aumenta el HW necesario.

  Se accede a esta estructura usando los bits menos significativos del PC de la instr. de salto para saber a qué entrada accedemos, también es posible el uso del campo tag para evitar el problema de la compartición de entradas. Otra configuración posible consiste en hacer la caché asociativa por cjtos haciendo las tablas más pequeñas (mismo coste HW, menor rendimiento).



La razón de no usar predictores de más de 3 bits es que la diferencia entre usar 4096 entradas o infinitas es despreciable, no se obtiene una diferencia que merezca la pena.

Para programas enteros este tipo de predictores (1, bimodal o 3 bits) fallan mucho, ya que los saltos son más frecuentes en este tipo de programas.

Para este tipo de programa se hace uso de predictores más complejos que se basan en la historia local del programa (almacenar el comportamiento de un determinado salto para predecir cómo se comportará en el futuro).

Otros programas hacen uso de la historia global en sus predictores (como muchas instr. de salto dependen de otras instr. de salto recientes, es decir, los saltos están relacionados, se almacena el comportamiento de los últimos n saltos para usarlo en la predicción).

Existe un tipo de predictor que hace uso de la historia global llamado predictor dinámico de dos niveles que trata de resolver los fallos de algunos programas enteros en los que hay correlación entre algunos saltos. Se especifica con (m,n) esto quiere decir que para cada salto hay 2 elevado a m predictores de n bits.
En un ejemplo concreto si tenemos un predictor de 2 niveles (1,1) quiere decir que tenemos 2 predictores de 1 bit para cada salto (P0 y P1), por lo que tenemos 4 combinaciones posibles, en función del comportamiento del último salto usaremos un predictor u otro. Con un predictor de 2 niveles (2,2) escogeríamos un predictor en función del comportamiento de los últimos 2 saltos, y así consecutivamente.

Viendo ejemplos concretos comprobamos que el predictor de dos niveles nos da resultados mucho mejores que el predictor de 1 bit, fallando en muchas menos ocasiones.

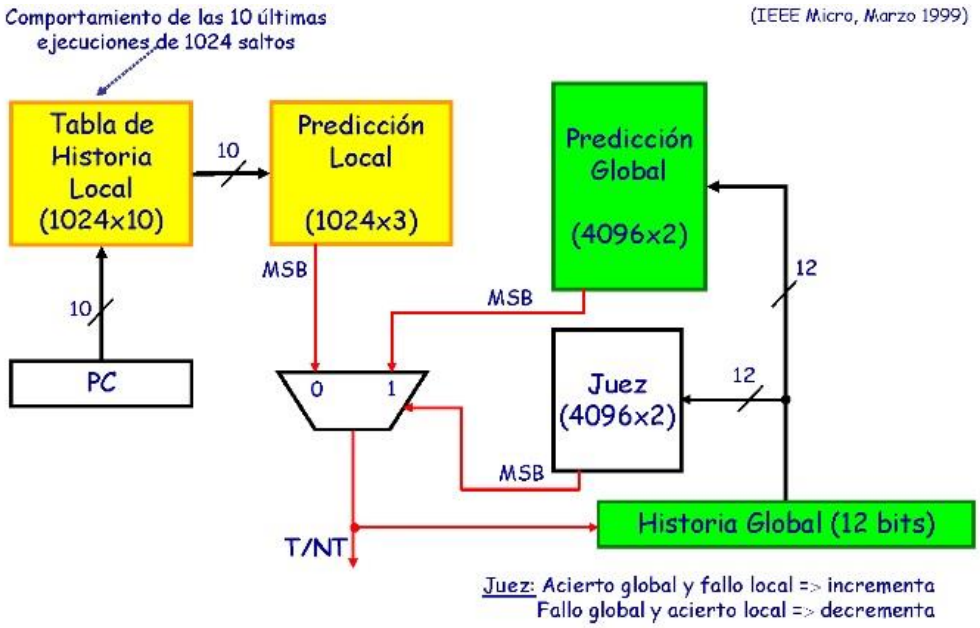
Existe otro tipo de predictores llamados híbridos, se llaman así porque no se limitan al uso de un solo predictor sino que usan varios y aplican la predicción de uno u otro según convenga, esto lo hacen mediante una tabla de selección que elige el predictor que haya dado mejores resultados, el que tenga una mayor tasa de aciertos, esto se sabe gracias al uso de una tabla de contadores saturados de 2 bits, que se actualizan a posteriori en función de si ambos predictores han tenido el mismo comportamiento (el contador no varía) o si tienen un comportamiento distinto (entonces el contador se incrementa o decrementa).

P1	P2	Actualiz. del contador
Fallo	Fallo	Cont no varía
Fallo	Acierto	Cont = Cont +1
Acierto	Fallo	Cont = Cont -1
Acierto	Acierto	Cont no varía

- Si P2 acierta más que P1 ⇒ *Cont* aumenta
- Si P1 acierta más que P2 ⇒ *Cont* disminuye

Bit más signif. del contador	Predictor seleccionado
0	P1
1	P2

Un ejemplo de este tipo de predictores es el Alpha 21264 (Tournament Predictor) que hace uso de un predictor local (predice en función de las ultimas 10 ejecuciones de un salto en concreto) y un predictor global (predice en función del comportamiento de los últimos 12 saltos ejecutados). Un juez decide cuál de los dos predictores se aplica (el que esté manifestando el mejor comportamiento). Tras cada salto se actualizan los predictores, si ambos hicieron una predicción distinta se actualiza el juez para que favorezca al que acertó.



Las tablas en amarillo conforman el predictor local y las verdes el predictor global. Cuando llega un salto, se accede (usando 10 bits del PC) a una tabla de la historia local (THL) que tiene 1024 entradas con anchura de 10 bits, transmitimos 10 bits al predictor local que funciona teniendo en cuenta el valor de esos bits para acceder a una entrada en concreto, cada una de las entradas guarda el comportamiento real de las 10 últimas ejecuciones de cada salto, hay 1024 entradas en total cada una con una anchura de 3 bits, la predicción se escoge usando el bit más significativo (MSB).

Aclaración: Un mismo salto viene identificado por su PC, el salto X siempre accederá a la entrada 0 de la tabla, en ella tenemos 10 bits que me dan su comportamiento las 10 últimas ejecuciones (THL), si la predicción a ese comportamiento es T el salto X se toma, la siguiente vez que se vaya a ejecutar el mismo salto y se acceda a la THL tendrá un 0 menos por la izq. y un 1 más por la derecha, por lo que accederá a una entrada distinta de la tabla de predicción, en resumen, un mismo salto accederá a una entrada distinta del predictor en función de su comportamiento en las últimas 10 ejecuciones.

Por otro lado en la predicción global accederemos a alguna de las entradas de la tabla de historia global (THG) en función del comportamiento de los últimos 12 saltos ejecutados, y según el contenido de esa entrada haremos una predicción.

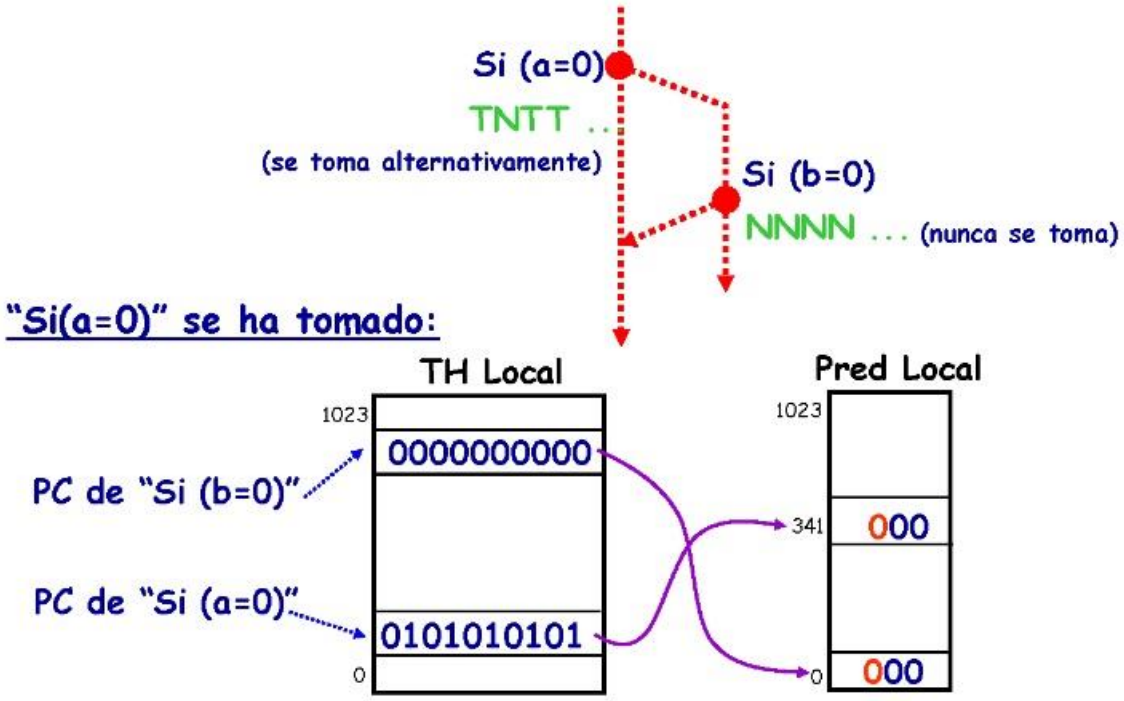
Con ambas predicciones (local y global) que pueden ser iguales o distintas el juez decidirá cuál de las dos tomamos (el juez también está indexado por el predictor de historia global, es decir, se accede a la misma entrada a la que se haya accedido en el predictor global)) en función del bit más significativo (MSB) de la entrada.

La conexión existente entre la THG y la predicción realizada es usada para la actualización de la THG según el comportamiento real.

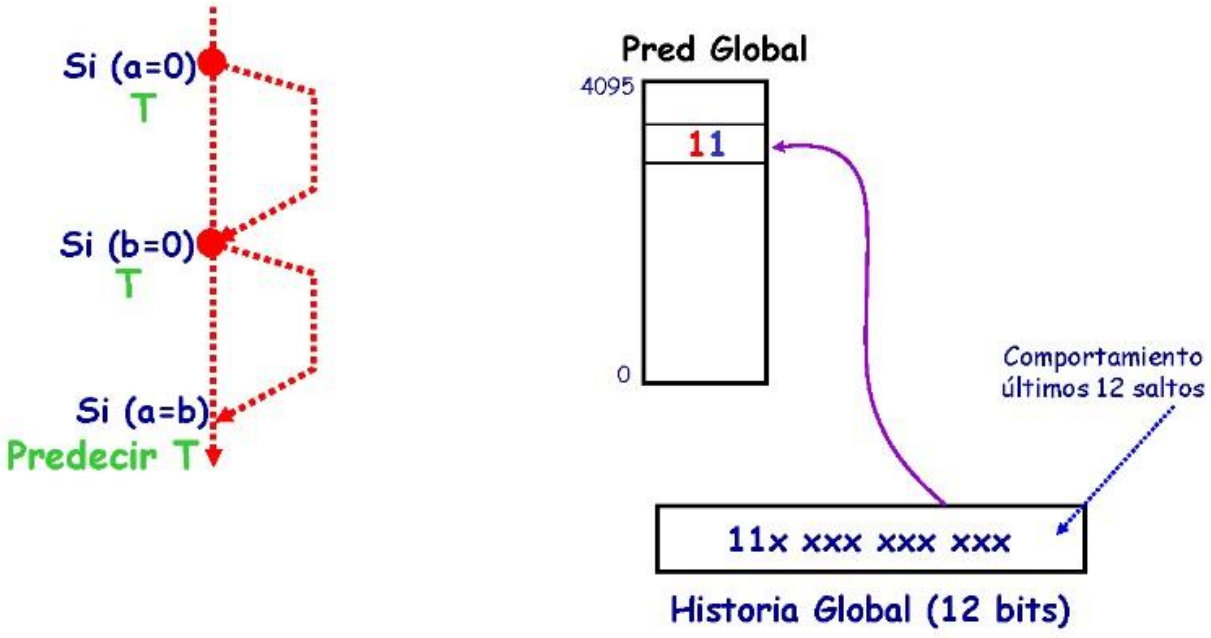
El juez funciona de la misma manera vista anteriormente en los predictores híbridos, con un contador que se incrementa o decrementa en función de qué predictor acierte o falle, esto se hace para intentar acertar en la predicción de la próxima ejecución intentando llegar a cambiar el bit más significativo (MSB) de la entrada a la que acceda el juez.

Ejemplos de funcionamiento:

- THL (Tabla de Historia Local)



- THG (Tabla de Historia Global)



Si hacemos una comparativa entre los predictores estáticos, dinámicos (bimodal) e híbridos la tendencia es que en general el híbrido es el que tiene mayor tasa de aciertos, esto se debe a que este tipo de predictores tiene la ventaja de tener la capacidad de seleccionar el predictor correcto para un determinado salto, esto es importante sobre todo en el tema de los programas enteros, ya que soluciona el problema de los saltos correlacionados (que se dan con mucha más frecuencia en este tipo de programas), ya que los predictores híbridos seleccionan el predictor global casi el 40% de las veces para programas enteros.

A la hora de intentar hacer predicciones en la dir. de retorno de una subrutina los predictores no son demasiado precisos, por lo que normalmente se acaba usando una pila LIFO de dir. de retorno de 8 a 16 entradas, así reducimos la tasa de fallos prácticamente a 0.

Recuperación de fallos de predicción (misprediction): Las tareas a realizar si se falla en la predicción son, por una parte descartar los resultados de las instr. ejec. especulativamente (estas instr. se almacenan en un buffer de reordenamiento que contiene registros temporales, si se falla la predicción se descartan los resultados del buffer), y por otra reanudar la ejec. por el camino correcto con un retardo mínimo (el procesador debe guardar la dir. de comienzo del camino alternativo (al menos) y en ocasiones prebuscar las primeras instr. siguientes a ese camino).

Especulación

Con la predicción de saltos introducimos el concepto de especulación, con el cual tenemos dos tipos de instr. en el procesador, las independientes (que se ejecutan si o si) y las que dependen de una predicción de salto. Para implementar esta distinción entre instr. tenemos que modificar el algoritmo de Tomasulo introduciendo este nuevo concepto.

Algoritmo de Tomasulo con especulación: Tiene 4 fases, una más que en el algoritmo normal, ya que ahora en la etapa write (antes era la última) no escribimos el resultado de la instr. más el identificador de reserva en los registros, sino que lo hacemos en una nueva estructura llamada ROB que es un buffer de reordenamiento.

La 4ª y última fase ahora se llama commit, y es en la cual se escriben ya los resultados en registros o en memoria, esta etapa se hace en orden (al igual que la etapa de Issue), así toda instr. que esté en esta etapa deja de ser especulativa (ya no es dependiente).

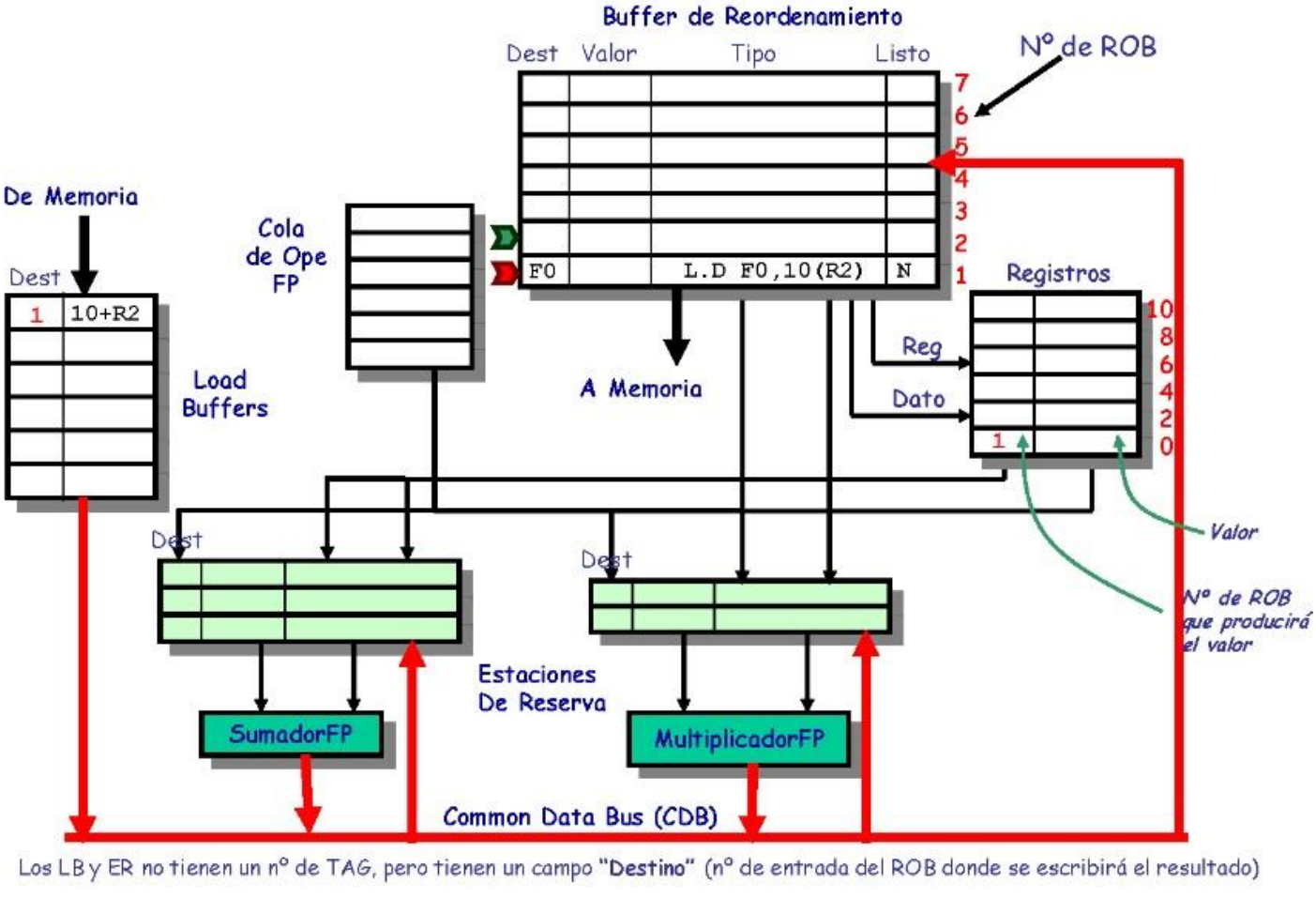
Con esta nueva configuración desaparecen estructuras como los store buffers, además de que los registros ya no van a ser accedidos por el CDB, sino que desde el CDB se va a acceder al ROB, los resultados que salen de las UF van allí y se quedan a la espera de saber si están bien ejecutadas o no, una vez ya lo saben se pueden actualizar los registros.

El ROB almacena los resultados de instr. cuya ejec. ha terminado pero todavía no hemos actualizado ni los reg. arquitectónicos ni la memoria, también se almacenan las instr. que dependen de un predictor de saltos, esta estructura permite el forwarding (paso de operandos entre instr. especuladas con dependencia LDE)

Los operandos de una instr. pueden llegar hasta las ER desde el CDB, desde el ROB o desde los registros, la diferencia tiene que ver con el estado de la ejec. que va a proporcionar esos datos.

El ROB es una estructura con 4 campos:

- **Destino**: En caso de tener un store la dir. de memoria, si en cambio es una instr. aritmética o load el nº de registro.
- **Valor**: Guarda el resultado de la ejec. de la instr. hasta que se pueda actualizar en el registro destino o en memoria (en este campo para los store (al no haber store buffers) si no tiene el valor disponible porque está a la espera lo que se guarda es el nº de entrada del ROB de la instr. que lo producirá).
- **Tipo de instr**.: Salto, Store, Aritmetica/Load (estas últimas solo si tienen como destino final un registro).
- **Listo**: Se indica si la instr. ha completado la fase de ejec. y el resultado está disponible en el campo valor.



Los LB y ER no tienen un nº de TAG, pero tienen un campo "**Destino**" (nº de entrada del ROB donde se escribirá el resultado)

Algoritmo de Tomasulo con especulación, fases:

- **Issue**: Toma la instr. de la cola de instr., necesitamos una ER con entrada libre y (ahora que tenemos especulación) también necesitamos una entrada libre en el ROB. En esta fase también se toman operandos de reg. o de resultados almacenados en el ROB por instr. previas. Una vez tenemos entradas libres copiamos el campo valor y tag de los reg. fuente en el campo valor y tag del reg. destino.
  Hay una gran diferencia respecto a la ejec. del algoritmo sin especulación, ahora NO marcamos el identificador de la ER sino que ponemos el nº de entrada de la ROB asignada, además en las ER hay que indicar cuál es el nº de entrada del ROB en la que está alojada esa instr.
- **Ejecución**: Se opera sobre los operandos, espera a que los operandos estén disponibles y chequea el CDB.
- **Finalización (write)**: Finaliza la ejec., escribe a través del CDB en todas las ER y en todas las entradas del ROB que estén a la espera del resultado, libera ER y LB, no escribe ni en reg. ni en memoria. Envía por CDB el resultado más el nº de entrada del ROB a la que se dirige.
- **Commit**: Se actualizan los registros (que ahora tienen un campo que indica el nº de entrada del ROB que producirá el valor correspondiente) desde el ROB, que funciona como una cola FIFO con dos punteros (en la cabecera y en la cola), cuando una instr. alcanza la cabecera, se le permite actualizar el registro (load/airtm.) o escribir en memoria (store) y finalmente se elimina esa instr. de su entrada en el ROB.

En la ejec. del algoritmo con especulación no pueden aparecer dependencias EDE ni EDL, ya que al hacerse el commit en orden garantizamos que no va a haber riesgos de este tipo, al haber acabado su fase de commit las instr. anteriores previamente.
En cuanto a los riesgos LDE, podrían producirse por ejemplo si un LD accede a una posición de memoria habiendo en el ROB un ST previo que almacena el resultado en esa misma posición, por ello tenemos que garantizar que siempre se lea después de que se escriba, usando un mecanismo que consiste en que a un LD no se le permite leer de mem. Si hay un ST previo en el ROB con la misma dir. de mem. Tampoco se ejecuta el LD si está pendiente el cálculo de la dir. efectiva de algún ST del ROB.

El ROB permite recuperarse de saltos mal predichos, si hemos hecho bien la predicción la instr. llega a la cabecera y se elimina del ROB en la fase de commit, si en cambio, hemos hecho mal la predicción tenemos que hacer una serie de acciones, borrar el contenido del ROB (ya que contiene instr. mal especuladas, que se han ejec. tras el salto), borrar las marcas (campo nº del ROB (Dest) de todos los reg.) y buscar la instr. correcta (el procesador siempre guarda la dir. del camino alternativo).

Si una instr. genera una interrupción se registra la petición en el ROB, si la instr. llega a la cabecera es porque no es especulada, por lo que reconocemos y tratamos la interrupción, cualquier instr. anterior ya habrá acabado por lo que ninguna instr. anterior puede haber provocado una excepción, con esto ejecutamos las excepciones como si se hubiera seguido una ejecución secuencial del programa.

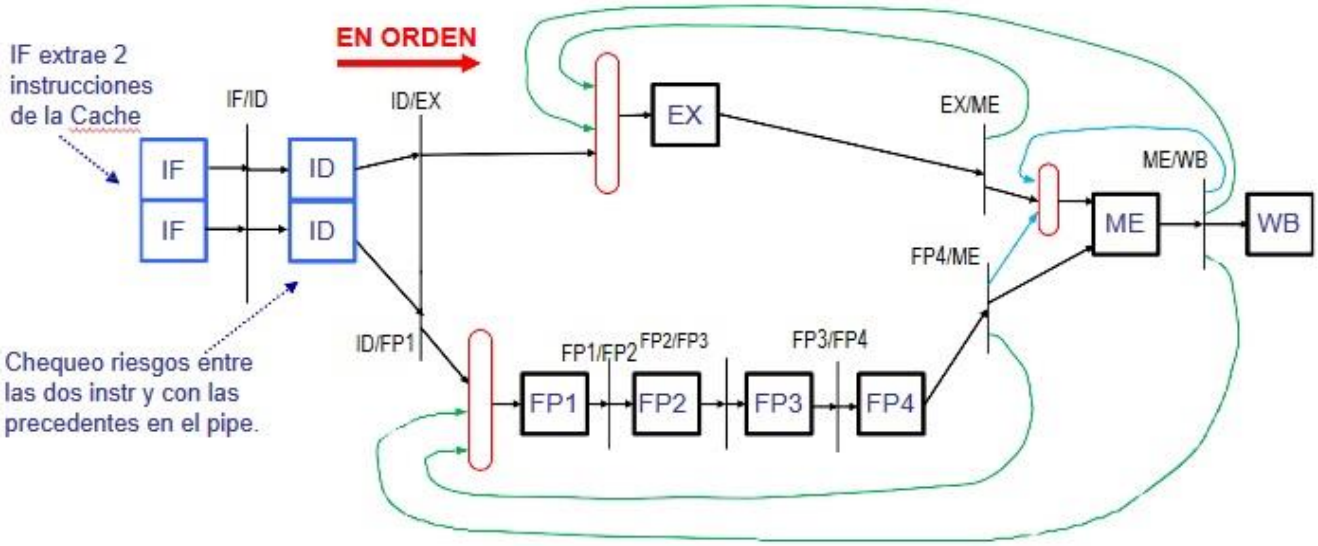
Tema 3 – Lanzamiento múltiple, Límites de ILP, Multithreading

Un procesador escalar es aquel que no puede ejec. más de una instr. en al menos una de sus etapas, es decir, de todas las etapas del pipeline al menos una es incapaz de procesar de forma simultánea más de una instr., esto implica que su CPI máximo es 1.

Un procesador superescalar en cambio, es aquel en el cual en todas sus etapas se permite la ejec. de más de una instr. Tenemos recursos para construir este tipo de procesadores ya que tenemos más área de silicio disponible y técnicas para resolver las dependencias de datos (planificació, desenrollado de bucles, SW pipelining) y de control (predicción de saltos, especulación).

Existen dos alternativas en cuanto a procesadores superescalares:

- **Procesador superescalar con planificación estática**: La planificación la hace el compilador en tiempo de compilación, chequea riesgos en etapa de ID (capacidad de chequear conflictos de varias instr. a la vez), si una instr. presenta conflictos bloquea esa instr. y las siguientes, la implementación típica son procesadores escalares de grado 2 (2 instr. a la vez).
  Existe un caso particular llamado VLIW (Very Long Instr. Word) en el cual en la etapa de ID no se chequean riesgos, el compilador conoce el HW disponible en la etapa de ejecución y sus latencias por lo que forma paquetes de instr independientes que se lanzan a ejec. simultáneamente, en cada ciclo se lanza un mismo nº fijo de instr. en cada paquete, si el compilador no encuentra suficientes instr. libres de riesgos lo rellena con instrucciones NOP.



En cada nuevo ciclo de reloj el fetch y la decodificación es de 2 instr por ciclo, de esta manera si una instr. es de FP y la otra entera se pueden enviar a ejec. en el mismo ciclo ya que usan recursos distintos (si escriben en reg. separados pueden acabar también en el mismo ciclo de reloj), los riesgos surgen cuando hay instr. de load y store, ya que si una instr. entera modifica el valor de un reg. entero existe una dependencia LDE (Ej. LD F0, 0(R2) y ADD.D F4, F0, F6).

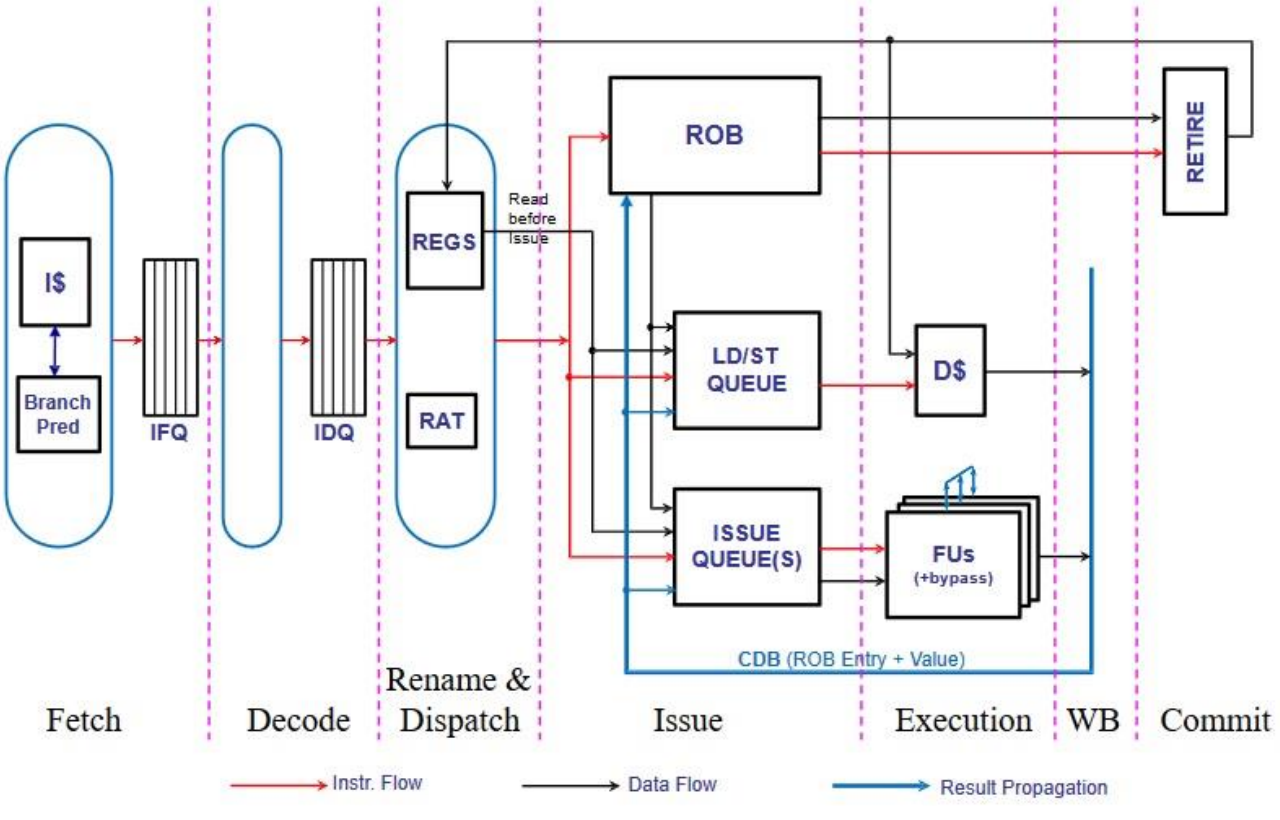
El tener más instr. ejecutándose de modo simultáneo incrementa las dependencias y las penalizaciones, ya que con un procesador escalar la instr. tendría que parar un ciclo, con uno superescalar el doble de instr. tendrán que parar, esto es una contrapartida negativa.

- **Ventajas**: No modifica el código (compatibilidad binaria) y no hay riesgos en ejec. (se analizan todos en tiempo de compilación).
- **Desventajas**: Mezcla de instr. (solamente si tenemos un programa 50% FP 50% ent. podríamos explotar toda la potencia del procesador), bloqueos en el lanzamiento (si se encuentra algún riesgo), planificación física (el compilador no es capaz de adaptarse a cambios que puedan surgir en la ejec. del programa) y para nuevos procesadores con distintas UF y latencias hay que replanificar el código.

- **Procesador superescalar con planificación dinámica**: La planificación la hace el procesador en tiempo de ejecución, analiza dependencias de varias instr. a la vez, si hay conflictos se envían las instr. a las colas de espera, puede mandar un nº variable de instr. a las UF por ciclo (en función de operandos y disponibilidad, hay un máximo de instr. para ejec. en cada ciclo pero no mínimo).
  La etapa de ID (decodificación) se hace en orden, la ejecución en desorden y la finalización (commit) en orden (con especulación siempre funciona así).

Tenemos que buscar las instr. de la caché y predecir los saltos, esto genera un flujo dinámico de instr. que hay que decodificar y renombrar sus reg., una vez hecho esto se hace la emisión de las instr., que consiste en pasar esa instr. desde un dispositivo de almacenamiento (cola de emisión (ER en Tomasulo)) a las correspondientes UF's cuando tengan sus operandos listos, tan pronto como esto ocurre las instr. se ejecutan y finalmente se reordenan y finalizan consultando el ROB.

A la primera parte del pipeline se le llama front-end y en ella se genera el flujo de instr. más probable teniendo en cuenta los saltos, a la segunda parte se le llama back-end y en ella se generan los resultados.



Pipeline de un **procesador superescalar dinámico** real:

- **Etapa fetch**: Su resultado es una secuencia de bytes que todavía están sin decodificar, su objetivo es predecir cuál va a ser la siguiente instr. a ejecutar, se gastan 4 ciclos en toda la etapa, se segmenta en varios ciclos para poder mantener altas frecuencias de reloj, cada ciclo se puede comenzar una nueva búsqueda (yendo a la caché, trayendo una línea de caché que incluye varias instr., vamos a explotar la localidad espacial (varias instr. almacenadas en posiciones contiguas) porque suponemos que si ejecutamos una instr. la siguiente va a ser la que está almacenada después (esto es lo normal salvo en los saltos)).

    - **1er ciclo**: Se trata de determinar cuál es la siguiente instr. a ejecutar, al multiplexor le entran varios inputs entre ellos la siguiente dir. secuencial (PC), la info. de la BTAC, (que nos dice la dir. de un salto probable), la info. del predictor (nos dice si el salto va a ser tomado o no, aunque actua sin saber si la instr. va a ser un salto, ya que no se sabe hasta la etapa ID) y una estructura llamada RAS que funciona como una pila en la cual se apilan las dir. de retorno para luego desapilar y tenerlas disponibles.

    - **2do ciclo**: Con la info. obtenida del primer ciclo se calcula la siguiente dir. a buscar (destino dir. salto o siguiente dir. secuencial).

    - **3er ciclo**: Una vez calculada la dir. se accede a la caché de direcciones para buscarla, tenemos un TLB (cachés virtualmente accedidas y físicamente marcadas).

    - **4o ciclo**: Se ve si hay acierto o no en la búsqueda y en caso de que lo haya se conduce la instr. encontrada a la fase de decodificación.

    Aclaracion: En una caché convencional se accede a grupos de instr. consecutivas en memoria, pero si tenemos un salto y se toma nos vamos a otra linea de cache y todas las instr. que vengan despues no las vamos a ejecutar, para evitar este problema se diseña la caché de trazas, que se diferencia respecto a la caché convencional en que almacena grupos de instr. consecutivos en el flujo dinámico del programa (no tal cual estan en la mem., se fija en cómo se ejecutan realmente y se guardan de acuerdo al orden de ejec. (secuencias dinámicas de instr.)), este tipo de cachés necesitan una info. adicional (control) en la cual se almacena el tag de la 1a instr. de la secuencia y luego el comportamiento de las siguientes instr. de salto, tenemos un acierto si el tag de la siguiente dir. a buscar y el comportamiento predicho del salto coincide con el control de una linea de caché, este tipo de cachés pueden acceder en un unico ciclo mientras que una convencional suele tardar 5 ciclos, con esto tenemos un mayor ancho de banda, la desventaja es la alta complejidad y la posible repetición de instr.

- **Etapa decodificación**: A esta etapa le llega una secuencia de bytes, su tarea es interpretarla y decodificarla para identificar las instr. contenidas en esa secuencia, pero existe el problema de que las instr. no tienen una long. fija (excepto RISC, que por su facilidad en este caso se hace la etapa en un solo ciclo de reloj), debido a esto no se puede partir la secuencia en partes iguales, por lo que la complejidad aumenta, para estos casos hay que usar la traducción dinámica que traduce las instr. mas complejas a operaciones más sencillas (micro-operaciones) con la misma longitud, puede traducirse a una o varias micro-operaciones.
  Existen casos especiales de instr. que no somos capaces de traducir a 4 micro-operaciones (máximo) por lo que usamos una memoria ROM que almacena un programa que almacena a su vez micro-operaciones. En el caso de los bucles esto es bueno ya que almacenamos trazas y ahorramos accesos a la caché.

  Con instr. tipo CISC la etapa se divide en dos:
  - **1a etapa:** Se delimita cada instr. (se parte de donde empieza cada una).
  - **2a etapa**: Se toman de la instr. queue y se hacen traducciones paralelas de varias a la vez, cada una en un decodificador (varios tipos: simples (1 micro-operacion), complejo (hasta 4 micro-operaciones) aparte de la ROM para los casos más complejos).

- **Etapa renombrado y dispatch**: Compuesta a su vez por otras dos:
  - **Renombrado**: Se hace en el alg. de Tomasulo usando el campo tag de los reg., por eso en la etapa de Issue se marca el reg. destino de la instr. con el tag de la ER en la que está alojada la instr. que iba a ir ahí, en un procesador superescalar se usan nuevos nombres para los reg. destino y también para los reg. fuente, se hace uso de elem. de almacenamiento intermedio llamados reg. físicos que no son visibles al procesador de forma directa, estos reg. son distintos a los arquitectónicos.
    Hay 3 maneras posibles:
    - **Renombrado mediante el ROB**: Es el más sencillo, guarda resultados especulados y se quedan a la espera de ser confirmados definitivamente en la etapa de commit, hace uso de la register map table (indica donde encontrar el valor más reciente de un reg. arquitectónico, según el valor de RDY (0 o 1) en el ROB (1) o en el banco de registros (0).
    - **Renombrado mediante un buffer de reordenamiento**: Se guardan en el ROB punteros a un cjto auxiliar de reg. en lugar de valores, los resultados se guardaran en otro buffer adicional.
    - **Renombrado mediante un fichero de reg. unificados**: Los resultados se guardan en un fichero de reg. fisicos que van a ser mucho más abundantes que los arquitectonicos, el ROB sigue existiendo pero cambia su funcionalidad, ahora solo mantiene el orden original de las instr (ya no hay que copiar resultados del ROB a reg. en la fase de commit) para que las entradas se asignen en orden (cada entrada guarda el nº del reg. fisico y el nº del correspondiente reg. arq., se debe guardar la correspondencia entre ambos mediante tablas de mapeado Frontend RAT (RA -> RF, última vez que RA era reg. destino) y Retirement RAT (Para cada RA indica el RF que contiene su último valor consolidado)).

A la hora de renombrar varias instr. en paralelo existe un problema con las tablas de renombrado, ya que éstas solamente contienen info. sobre algunas de las instr. anteriores, y deben tener en cuenta todas las instr. anteriores, hay 2 alternativas para solucionar este problema con el renombrado:

- Renombrado secuencial: Segmentar (cada nuevo ciclo de reloj aumentamos la frecuencia para subdividirlo en ciclos más pequeños).
- Renombrado combinacional: Teniendo en cuenta los operandos y destinos se chequean todas las instr. del grupo y se comparan los reg. destino con los reg. fuente de las siguientes instr.

- Dispatch: Etapa muy sencilla que consiste en reservar entradas (espacio) en estructuras que luego van a necesitar guardar valores, como el ROB o las IQ.

  Existen varias alternativas a la hora de leer operandos:
  - Leer antes de emisión.
  - Leer después de emisión: Se pospone la lectura al momento de la ejec.

  Al igual que se hacía con los campos tag de los reg. en el alg. de Tomasulo se hace un renombrado (una estructura llamada RAT es la encargada de determinar qué asociación tiene el renombrado), también se hace el dispatch que consiste en reservar recursos en queue's.
  Para las instr. de ST se asigna una entrada en ROB, otra en la MIQ y otra en la SQ (ésta en orden).

- **Etapa Issue**: Se pone en el CDB tanto el valor como el tag de la instr. que ha terminado, en un procesador superescalar va a ser el nº de entrada (ROB) o el nº de RF (banco de reg.). Una instr. que termina recorre la issue queue (IQ) buscando coincidencias entre su tag y los guardados en las demás entradas de la IQ, si hay coincidencia de tags se actualiza el campo de valor (source data) e indicamos en el campo ready si está listo (1) o no (0).

  Existe un mecanismo despertador llamado wake-up mechanism que busca despertar a las instr. que estaban esperando un resultado por si ya se ha generado (si coinciden los tags) y el bit de valid = 1. En esta etapa cabe la posibilidad de que se dé un problema si una instr. produce un resultado (productora, escribe en un reg. destino) y luego viene otra instr. que necesita leer el valor de ese reg. (consumidora) ya que la productora no pone su resultado en el CDB hasta que termine su fase de ejec. por lo que la segunda pierde 3 ciclos hasta que acabe la primera (que la instr. productora haga el mecanismo de wake-up).

  Para solucionar este problema y las dependencias ocasionadas se usa el wake-up anticipado, que se basa en que las instr. (productoras) aritméticas tienen una latencia conocida, por lo que podemos despertar a la instr. consumidora varios ciclos antes, mandando la señal de wake-up antes (wake-up especulativo (que consuma un resultado todavía no producido)), en caso de que se de algún tipo de latencia o conflicto producido por disponibilidad de UF o por el CDB este método no nos sirve ya que el dato no estará a tiempo para ser consumido.

Este método solo nos sirve para instr. aritméticas ya que tienen una latencia conocida, con otras instr. con latencia variable como las de memoria necesitamos mecanismos de recuperación. Para el tratamiento de instr. de memoria tenemos una estructura llamada mem. Issue queue (MIQ) con unas entradas que guardan estas instr (desordenadas) de manera que cuando la instr. esta lista (se conoce la dir. que se tiene que buscar o el dato que se tiene que escribir) se pasa a la UF.

También tenemos otra estructura llamada AGU que calcula direcciones y las pasa a las entradas correspondientes de la load queue (LQ) o store queue (SQ) (que guardan las instr. en orden (para saber cuáles son los ST más antiguos que un LD)). Para los ST si los operandos están disponibles se manda la instr. a la AGU y se guarda en la SQ la entrada correspondiente de la IQ. (la entrada de la MIQ se libera).

Existe una técnica usada con las instr. de memoria llamada desambiguación dinámica de mem., que consiste en quitar la ambigüedad, es decir, asegurar que las instr. de LD y ST no tienen dependencias con instr. previas, para que haya riesgos entre instr. tiene que haber dir. coincidentes.

Se pueden dar varios casos, por ejemplo un ST en la SQ solo puede escribir en la mem. (caché) si tiene tanto su dato como su dir. y además está en la cabecera del ROB (instr. más antigua), todos los LD y ST previos tienen que haber hecho commit. Otro caso posible se da cuando un LD puede leer de la caché si tiene su dir. y además si esta dir. no coincide con la de ningún ST previo dentro de la SQ.

Cuando se hace la lectura de operandos después de la etapa de issue hay que tener varias cosas en cuenta, en cuanto al commit tenemos que actualizar la retirement RAT (ya que guarda el último RF al cual fue mapeada una instr. que si hizo commit para cada RA, en la frontend RAT puede haber valores especulados, aquí no). Normalmente se usa un fichero de reg. unificado para tratar de reducir el nº de accesos al bando de reg., guardamos ahí los valores y no tenemos que actualizar en commit, las IQ no almacenen valores, solo si están disponibles o no, la identificación de los operandos fuente se hace usando el tag o nº del RF (no el nº de ROB), la lectura de reg. se hace en el momento que la instr. se emite a las UF, los resultados del CDB van acompañados del nº de reg. destino, no se escriben en el ROB, sino directamente en los reg.

- **Etapa ejecución y write back**: En esta etapa aumenta la complejidad al tener varias UF operando en paralelo, cada UF opera sobre los datos que le llegan y genera el valor correspondiente, cuando termina genera un resultado y se trata de despertar a la instr. de la IQ que pudieran estar esperando ese resultado generado, que se puede mandar directamente a las entradas de las UF's que las necesitan en el ciclo siguiente mediante un mecanismo combinacional de anticipación de operandos (by-pass), es importante ver si compensa lo que ganamos (ciclos de espera entre instr. dependientes) con lo que perdemos (por la complejidad combinacional implicada).

Sin mecanismo de by-pass la instr consumidora tiene que esperar a que la productora escriba el dato en el banco de registros, con by-pass se evita la espera pero se aumenta la complejidad, en procesadores con alto grado de segmentación se segmenta el camino entre los RF y las UF's para evitar perder hasta 4 ciclos de espera. Según la proximidad de las instr. podremos hacer implementar un mecanismo de by-pass más directo o menos directo.

- **Etapa commit**: Esta etapa consiste en finalizar las instr. actualizando el estado arquitectónico (valores que en un momento dado tiene tanto la mem. como los RA) exactamente en el mismo orden que un procesador secuencial.

  Una instr. hace commit en el mismo orden en el que se ejec. el orden dinámico del programa, es decir, una instr. finaliza (si usamos ROB) cuando está en la cabecera copiando el valor de la cabecera en el reg. correspondiente o (si usamos fichero de reg. unificado) no hay que hacer nada en esta fase ya que se hizo un renombrado del reg. en la fase de dispatch y el contenido de una fila de la retirement RAT pasó a ser el valor del reg. en cuestión (el resultado ya esta en el reg).

  En cualquier caso si una instr. "i" hace commit es porque todas las anteriores ya han terminado (han hecho commit, no pueden activar excepciones), pueden producir o haber producido resultados especulativos pero no van a modificar el resultado de la instr. que está haciendo ahora commit.

  Si la instr. a finalizar y hacer su commit es un salto mal predicho, hay varios casos:
  - **Si tenemos un ROB**: el contenido del ROB y todas las instr. en la cola deben eliminarse (ya que son instr. mal especuladas), el ROB debe guardar la predicción de los saltos para poder compararlo en el commit.
  - **Si tenemos un fichero de reg. unificado**: Se restaura la frontend RAT, ya que su contenido puede tener instr. mal especuladas por lo que hay que actualizarla debido al fallo de predicción, se copia el contenido de la retirement RAT a la frontend RAT, ya que la retirement no contiene valores especulados (en la cual cada vez que llega un salto que queremos predecir guardamos el contenido).

    El gran número de instr. en las colas puede ser un problema, existen soluciones más ágiles que consisten en anticipar la detección de fallos mal predichos (ejec. y finalizando las instr. más antiguas y desechando el salto y todas las instr. más jóvenes).

Límites del paralelismo a nivel de instr. (ILP)

Def. ILP: Se basa en tratar de ejec. instr. de la forma más concurrente posible a nivel de instr. (hacer grupos, reordenar, planificar…). Tiene una serie de límites:

- Permite mejorar el rendimiento sin afectar al modelo de programación, pero en los últimos años el nº de instr. a lanzar por ciclo no ha cambiado.
- La diferencia entre el rendimiento ideal (pico) y el obtenido crece, cada vez nos quedamos más lejos del rendimiento teórico máx. que podemos alcanzar.

Lo que limita el ILP son las dependencias, en resumen, podemos tener todos los recursos HW que queramos pero si tenemos un programa lleno de dependencias no vamos a poder mejorar el ILP.

Si tenemos un código con muchas dependencias LDE no tenemos más remedio que secuencializar, la conclusión es que con HW realizable todo el ILP que se puede obtener ya se ha obtenido.

Def. Arquitectura: Repertorio de instrucciones. Ej: Intel (x86).
Def. Micro-arquitectura: Distintas formas de organizar componentes.

Pentium 4: Procesador de Intel, primero que hacía uso de multithreading (soporta la ejec. de 2 hilos de forma simultánea (2 procesadores lógicos)), su característica más distintiva era su caché de trazas, en vez de almacenar instr. x86 como hace la caché de instr. se almacenan secuencias de micro-operaciones que se han ejec. de manera consecutiva de manera que si se ha dado un acierto con la caché de trazas nos ahorramos calcular las operaciones otra vez.
Hacía uso de una arq. netburst, con mejor predicción de saltos (en un factor 8) y con más UF's (7 en total), también tenía una caché de trazas (en caso de acierto eliminamos la necesidad de decodificar esas instr.) y una BTB más grande (incrementa un 30% la tasa de acierto).

La forma de intel de desarrollar micro-procesadores sigue el modelo de desarrollo tick-tock, que debe su nombre a su comportamiento repetitivo, se crea una nueva micro-arquitectura en la cual se usa un tamaño determinado de long. de puerta del transistor, en la siguiente (tick) se usa la misma micro-arquitectura pero con distinto tamaño de long. de puerta del transistor, la siguiente (tock) se usa el mismo tamaño de la puerta del transistor pero distinta micro-arquitectura.
Tick: Misma micro-arq., distinto tam. transistor.
Tock: Mismo tam. transistor, distinta micro-arq.

Para superar los límites del ILP se podría:

- Mejorar el SW, mejorando compiladores y predictores.
- Eliminar riesgos EDL y EDE en memoria.
- Eliminar riesgos LDE en reg. y en memoria.
- Investigar otros paralelismos (thread) (Opción más eficiente)

Def. Paralelismo a nivel de thread: La idea es crear flujos de instr. independientes (threads) que se ejecuten concurrentemente.

Def. Multiprocesador: Múltiples procesadores, cada uno de ellos puede a su vez ser **multithreading** (característica que permite ejec. flujos de instr. de manera concurrente).

Multicore: nº de cores reducido.
Manycore: Cientos o miles de cores.

La diferencia entre un core y un procesador consiste en que un core no es más que la CPU (unidad de procesamiento) más algún nivel de caché, un procesador en cambio aparte de la CPU y la caché también tiene distintos niveles de jerarquía de mem., reg. arquitectónicos…etc.

Def. Paralelismo a nivel de datos: Operaciones o instr. idénticas que operan sobre grandes volúmenes de datos.

Tipos de paralelismo:

- A nivel de instr.: Reordenar código (ILP).
- A nivel de thread: Separamos el código en distintos flujos (TLP).
- A nivel de dato: Me permite realizar la misma operación pero sobre múltiples datos (DLP).

Diferencias entre ILP y TLP:

- ILP se centra en una cantidad limitada de instr.
- TLP son distintos flujos de instr. que se han construido de tal forma que ya son inherentemente paralelas, su objetivo es usar múltiples flujos de instr. para mejorar el throughput de computadores (nº de programas que el procesador es capaz de ejec. por unidad de tiempo) y reducir el tiempo de ejec. de un programa multi-thread. Aporta mayor rendimiento respecto a ILP.

Multithreading

En los procesadores superescalares existen dos problemas fundamentales:

- En un procesador escalar con 3 UF's, en un programa típico en muchos ciclos se están infrautilizando las UF's que tenemos, esto puede estar causado porque no hay instr. que ejec. o si las hay pero no tienen los operandos disponibles todavía o que haya UF libres pero no del tipo necesario.

- La latencia de memoria crece, esto no quiere decir que las mem. sean cada vez más lentas, sino que el ritmo al cual el procesador es cada vez capaz de aceptar nuevas peticiones no es comparable al ritmo al que evoluciona la mem., el procesador es capaz de servir datos de manera más rápida que la mem., ya que cada vez la diferencia de ritmos es mayor (a esto se refiere la crecida de latencia).

Con **multithreading** (característica propia de algunos procesadores que permite ejec. flujos de instr. de manera concurrente) podemos conmutar hilos de ejec., cuando un hilo hace una parada debida a algún tipo de evento, gracias a esto incrementamos el trabajo procesado por unidad de tiempo, o si los hilos son del mismo trabajo se reduce el tiempo de ejec.

A pesar de sus ventajas la técnica multithreading no es ideal y se pueden producir pérdidas de rendimiento, por ejemplo si tuviéramos 20 hilos cuando acaba el último y le vuelve a tocar al primero si no hubiésemos usado multithreading puede que el primer hilo ya hubiera acabado, es decir, que un programa o hilo individual puede ver incrementado su tiempo de ejec. usando multithreading en algunas ocasiones.

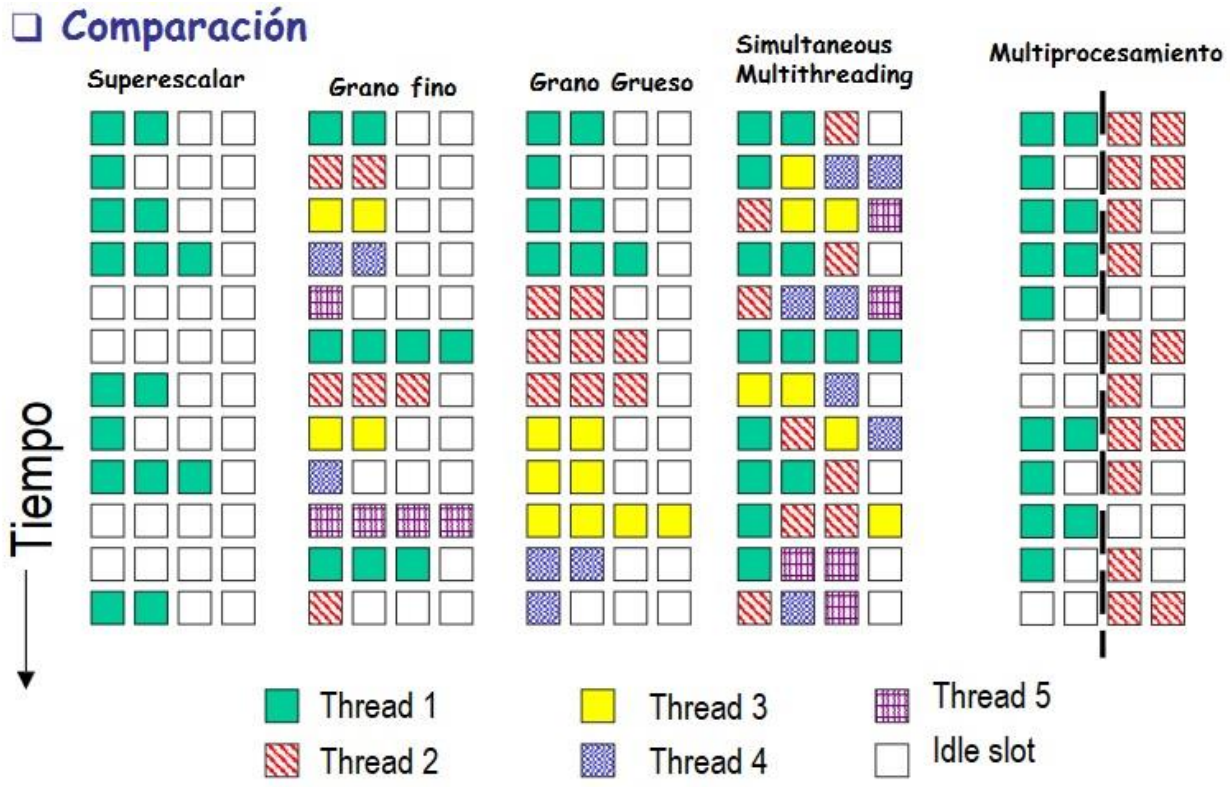
Los múltiples hilos comparten recursos del procesador, que debe mantener el estado de cada hilo, una copia en el bloque de reg., un PC separado y tablas de página separadas.
Es necesario un HW capaz de cambiar de hilo muy rápido, más rápido que el tiempo que tenemos para cambiar entre procesos, ya que no nos puede llevar más tiempo cambiar de un hilo a otro. Dependiendo de cuando cambiemos de hilo existen dos tipos de multithreading:

- Multithreading de grano fino (cada ciclo de reloj): En cada ciclo de reloj se cogen instr. de un thread distinto, se forma un flujo de instr. que pertenecen a distintos threads, la CPU debe ser capaz de cambiar de thread cada ciclo de reloj, normalmente se usa política round-robin (RR) (los threads parados se saltan).
  - Ventajas: Puede ocultar paradas tanto de alta como de baja latencia, ya que al conmutar threads cada ciclo nos da igual cómo sea que vamos a ser capaces de ocultarla, también puede invertir el tiempo derivado de las paradas ejecutando otros threads.
  - Desventajas: Puede que un thread que hemos parado pudiera seguir ejecutándose, por lo que estamos retardando el tiempo total, en resumen si un thread no tiene paradas lo estamos retrasando.

- Multithreading de grano grueso (cuando un thread haga una parada larga): Conmuta entre threads solo con paradas de larga duración.
  - Ventajas: No necesita un HW de conmutación muy rápida entre threads, ya que ejec. muchas instr. de un mismo thread, que acumulan retardo.
  - Desventajas: No elimina las pérdidas por paradas cortas debido al coste de conmutación en ciclos (al no tener un HW rápido), además el pipeline se debe vaciar y llenar al cambiar entre hilos, lo que provoca un retraso.

Def. SMT (Simultaneum multithreading): Otro tipo de multithreading es el SMT que, a diferencia del de grano fino y el de grano grueso, en un mismo ciclo de reloj permite el lanzamiento de instr. pertenecientes a más de un thread, es decir, permite la ejec. simultanea de varios threads.

Se necesita gran nº de RF para mapear los RA, debido a los tags podemos mezclar instr. de distintos threads sin confundirlas y la ejec. fuera de orden permite un uso más eficiente de los recursos ya que así las instr. entran en las UF's nada más tengan listos sus operandos, necesitamos añadir HW adicional como tablas de renombrado, un ROB (commit) y un PC por cada uno de los threads.



IBM Power5: Tiene el doble que capacidad que su antecesor el "*Power 4*", a pesar de que cambian muy pocos recursos en su arquitectura, se añaden dos recursos HW de fetch, dos de la etapa inicial de ID y 2 de commit.

Todos los recursos HW son compartidos a excepción de algunos elem. que son necesarios por separado por cada hilo, como el PC, el buffer de instr., la pila de retorno, la estructura de grupo, o la SQ.

Este procesador ejecuta hasta 2 threads, esto ocurre porque con 4 los recursos compartidos se convierten en un cuello de botella, es decir, fallarían más threads, tendríamos más peticiones a mem, se aumentaría la presión sobre el ancho de banda, sobre los RF, sobre la caché…etc y esto nos lleva a una degradación del rendimiento.

El power 5 cuenta con un sist. de balanceo de carga dinámica que trata de garantizar que un hilo no se apropie de demasiados recursos dejando al otro sin demasiados a su alcance, este sistema se compone de tres fases:

- Monitorización: Se monitorizan los fallos en el segundo nivel de caché, estos suponen ir a mem. principal que supone un gran coste en ciclos, si en un thread tenemos muchos fallos de este tipo ese thread se quedará parado a la espera de que la mem. suministre los datos y se gastarían entradas en la IQ ocupando espacio que no puede usar el otro thread que a lo mejor no debe esperar a la memoria.
- Quitar recursos: Tras la monitorización se determinan umbrales que si se cumplen se toma algún tipo de acción correctora, generalmente quitarle recursos al hilo que está haciendo un uso abusivo de los mismos, se hace de varias maneras, reduciendo la prioridad del hilo, no decodificando más instr o eliminando instr. desde emisión y parando decodificación (aumento progresivo de la agresividad).
- Ajuste de prioridad del hilo: Por último se ajusta la prioridad del hilo, que tiene que ser baja cuando está en espera activa o alta en tiempo real, el power 5 cuenta con hasta 8 niveles de prioridad (el que tenga más decodifica instr. durante más ciclos).

Con el paso de los años se han realizado una serie de cambios en el power 5 para soportar SMT:

- Se ha aumentado la asociatividad del primer nivel de caché (L1) y la TLB.
- Se ha añadido una cola de ST/LD por thread
- Se ha incrementado el tamaño de los niveles 2 y 3 de caché (L2 y L3)
- Se ha añadido un buffer de prebúsqueda separado por thread.
- Se ha incrementado el nº de RF y del tamaño de las colas de emisión.

En términos de área y consumo el power 5 era más costoso que su predecesor.

Niagara (SUN 2005): ejemplo de multiprocesador en un chip, 8 cores y cada uno de ellos soporta multithreading de grano fino, pueden conmutar entre 4 threads, se incrementa el uso del procesador y es necesario un gran ancho de banda, tiene una mem. caché L2 compartida por todos los cores.
Montar muchos cores tiene una contrapartida y es que demanda un elevadísimo tráfico por unidad de tiempo.

Como conclusión a todo esto podemos sacar en claro que hacer un procesador superescalar que procese el doble de instr. es un proceso nada gratuito, requiere de 3 a 4 accesos a caché, predecir 2 o 3 saltos, acceder a más de 20 reg. y buscar en la cache de 12 a 24 instr (prebúsqueda), todas estas acciones por ciclo de tiempo.

La complejidad de implementar estas capacidades conlleva sacrificar la duración del ciclo, si aumentamos la complejidad todas esas tareas que antes podíamos realizar en un ciclo ahora no es posible hacerlo, por lo que tenemos que aumentar el tiempo de ciclo y reducir la frecuencia, además esto afecta al consumo incrementándolo.

En general la inmensa mayoría de las técnicas que incrementan el rendimiento conllevan también un aumento notable del consumo, por ello una técnica es eficiente en energía si el incremento que nos da en rendimiento es mayor que el incremento del consumo, todos aquellos procesadores que lanzan muchas instr. por ciclo de reloj al final se convierten en poco eficientes desde el punto de vista energético.

Todos estos motivos han llevado a la industria a focalizarse sobre multiprocesadores en un chip en vez de seguir explotando el ILP, tenemos que seguir centrándonos en el paralelismo existente en las aplicaciones y para ello tenemos que tener un balance satisfactorio entre velocidad de procesamiento y consumo de energía.

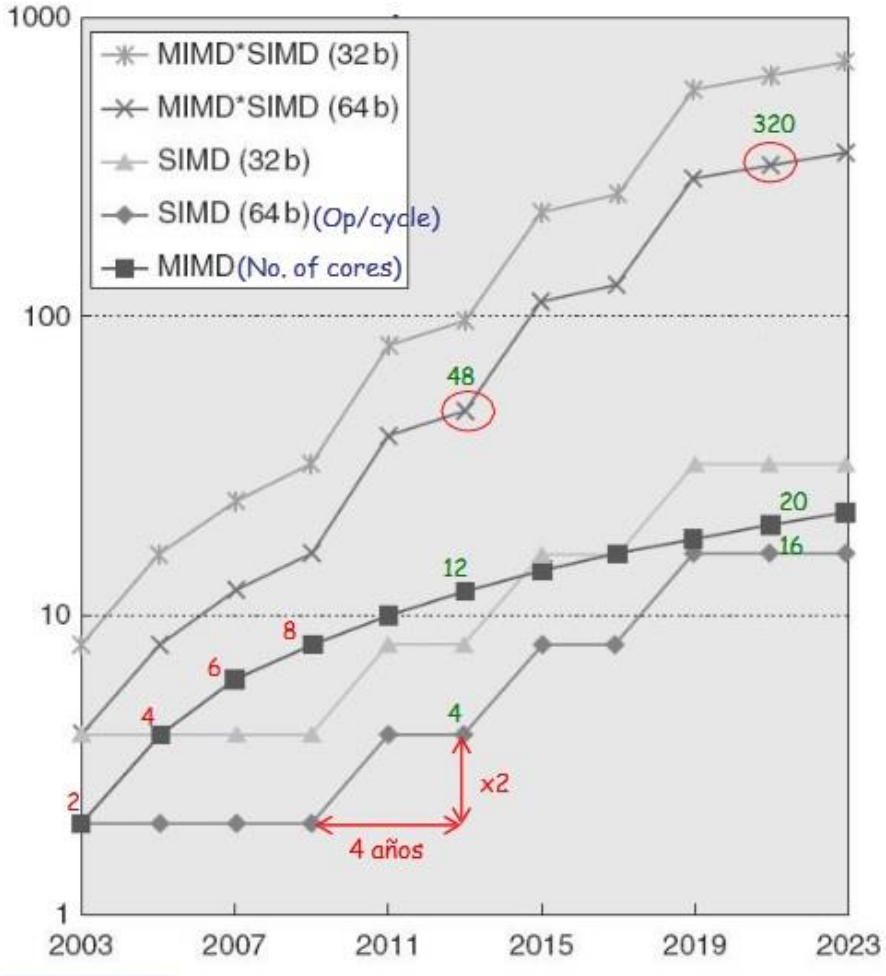
Tema 4 - Paralelismo a nivel de datos: arq. vectorial, instr. SIMD par multimedia, GPUs

Def. SISD (Single Instr. Single Data): Monoprocesador. ADDD F0, F2, D4.

Def. SIMD (Single Instr. Multiple Data): Una única instr. me da suficiente info. como para operar sobre múltiples datos, arq. o procesadores vectoriales, GPU's o extensiones multimedia. ADDV V3, V2, V1.

Def. MIMD (Multiple Instr. Multiple Data): Multiprocesador o manycore (TLP), paralelismo a nivel de thread. Ej: Power 5.

El speedup (factor por el cual voy a mejorar el potencial de los programas) esperable mediante paralelismos SIMD y MIMD y su combinación se representan en la figura de abajo, en arqu. intel el nº de cores se dobla cada dos años, en promedio el nº de operaciones de 32/64 bits por ciclo de reloj se dobla cada 4 años. Si combinamos ambos paralelismos vemos como el speedup tiene un crecimiento notable, en 2021 de un rendimiento de 16-20 (sin combinar) a uno de 320 (combinados), en resumen si combinamos ambos tipos de paralelismos obtenemos los beneficios del paralelismo a nivel de thread y del paralelismo a nivel de datos.



El consumo de potencia/energía es un problema muy grande, una de las grandes ventajas de las SIMD en comparación con MIMD es que es más eficiente energéticamente, ya que solo es necesario hacer un "fetch" para operar sobre varios datos, en cambio en MIMD tenemos que hacer 64 fecth para el mismo nº de oper.

Otra ventaja de SIMD es que el programador sigue pensando en un flujo secuencial de instr., no hay nada que coordinar por lo que la dificultad a la hora de elaborar código no incrementa.

Arq. propias para explotar SIMD: Arq. vectorial, Extensiones SIMD y GPU's.

Def. Arquitectura vectorial: Arquitectura en donde en vez de usar reg. ya conocidos (R0) se usan reg. vectoriales (V0), la manera de operar es la misma que con los otros tipos de reg. el uso de reg. vectoriales sirve para ocultar la latencia de mem. (mientras unas instr. vectoriales hacen su trabajo otras se ejec. simultáneamente).

Las diferencias entre las operaciones vectoriales y las anteriores son:

- Es necesario que las instr. de los programas sobre los que se va a operar sean independientes (que no tengan riesgos o dependencias que motiven paradas en el pipeline).
- Alto contenido semántico (con solo una instr. indicamos al procesador que tiene que hacer muchas oper. de forma simultánea).
- Patrón de acceso a mem. conocido (sabemos cómo están colocadas las componentes que forman esas matrices o vectores en mem. por lo que sabemos exactamente en qué posiciones de la mem. física residen los datos).
- Explotación eficiente de mem. entrelazada.
- Reducción del nº de instr. de salto (como agrupamos las oper. en paquetes de hasta 64 estamos reduciendo el nº de saltos ya que solo tenemos que chequear la oper. de salida del bucle cada 32/64 oper), parecido al desenrollado de bucles, solo necesitamos un salto cada vez que se ejec. una nueva instr. compuesta por hasta 64 oper.

Cray-1: Primera unidad escalar que hace uso de arq. vectorial, contaba con 8 reg. vectoriales, cada uno de ellos almacenaba 64 elem., también tenía reg. de máscara (VM) y reg. vectorial de longitud (VLR).

Operaciones vectoriales aritméticas sobre reg.: los datos residen en mem., se llevan a los reg. vectoriales y una vez allí operamos con ellos.

Modos de direccionamiento especiales para vectores no contiguos: Se tiene una dir. inicial y a partir de ella tenemos los 64 elem. contiguos (siguientes) y se cargan en mem., si no se quieren cargar elementos contiguos se hace uso de un tipo especial de load LVWS (With Stride), que carga elem. equiespaciados a una cierta distancia, también existe otro tipo de load LVI (Index) que usa un reg. vectorial donde se almacenan las posiciones de los elementos a cargar usando un índice.

Reg. de long. vectorial (VLR): Indica la longitud de los vectores a procesar (para poder manejar vectores de elem. menores a 64), se pone el VLR al nº de componentes que tienen los vectores sobre los cuales queramos ejec. una operación.

Reg. de máscara (VM): Indica de forma selectiva sobre qué componentes quiero operar y sobre cuáles no para una determinada operación, se pone a 0 o a 1 según sea cierta la condición o no, funciona a modo de filtro para hacer ejec. selectiva de oper.

Instr. de comparación:

- CVI V1, R1: Crea un vector de índices guardando los valores 0, 1*R1, etc. en V1.
- S—VV.D V1, V2: Compara los elementos (EQ, NE, GT, LT, GE, LE) en V1 y V2. Si la condición se cumple pone un 1 en el correspondiente bit del vector, sino un 0.
  Se pone el vector de bits resultante en el vector del reg. de máscara (VM).
- POP R1, VM: Cuenta el nº de 1 (veces que se ha cumplido la condición) del registro de máscara y lo guarda en R1.
- CVM: Operación atómica, resetea el reg. de máscara poniéndolo todo a 1's.
- MTC1 y MFC1: Mueven o quitan contenido al reg. de long. vectorial (VLR).
- MVTM Y MVFM: Mueven el contenido del reg. de máscara a un escalar o viceversa.

En un ejemplo concreto con el bucle AXPY (Y = a*X + Y) en su versión escalar tenemos que ejec. aproximadamente en total unas 578 instr. mientras que en la versión vectorial basta con ejecutar 6 instr. vectoriales.

Si observamos otro ejemplo más complejo como el producto de dos matrices, vemos que con el código escalar la complejidad de MULVV.D tiende a ser n^2 instr. más la complejidad de ADDVV.D, que al no ser esta última una oper. vectorial se debe hacer una ejecución secuencial escalar (no vectorial), teniendo una complejidad de n^3.

Si en cambio usamos código vectorial podemos ir calculando todos los elementos de una fila de la matriz resultante a la vez, de manera que las operaciones de MULVS.D son vectoriales y las de ADDVV.D también lo son, teniendo ambas una complejidad n^2.

Para medir el tiempo de ejec. de un código vectorial se ha de tener en cuenta una serie de factores que limitan el rendimiento:

- Long. de los vectores operandos (MVL).
- Riesgos estructurales existentes en el código (UF necesarias ocupadas o puertos ocupados del banco de reg. disponible).
- Dependencias de datos.
- Velocidad de procesamiento: Las UF's del VMIPS consumen y producen un elem. por ciclo de reloj, por lo que puede entrar una nueva oper. en la UF cada ciclo de reloj y salir una vez haya pasado el tiempo de latencia, en total el tiempo de ejec. en ciclos de reloj de una oper. vectorial es aproximadamente igual a la long. del vector.
- Convoy: Cjto de una o varias instr. vectoriales consecutivas que potencialmente pueden ejecutarse juntas, esto lo determina la ausencia de riesgos estructurales, pueden tener riesgos LDE, dos convoyes nunca se solapan.
- Paso (chime): Tiempo que tarda en ejec. un convoy.

```
1: LV       V1, Rx      ; load vector X
2: MULVS.D  V2, V1, F0  ; vector-scalar multiply
3: LV       V3, Ry      ; load vector Y
4: ADDVV.D  V4, V2, V3  ; Vector addition
5: SV       V4, Ry      ; Store result Y
```

En la imagen superior podemos apreciar un ejemplo de código vectorial dividido en 3 convoyes, en el primero existe una dependencia LDE pero no tienen riesgo estructural (LV -> Unidad vector, MULVS.D -> UF) por lo que forman un convoy las 2 primeras instr., la tercera no entra dentro de ese convoy porque usa la misma unidad que el 1er LV, por lo cual el segundo convoy está formado por la 3ª y 4ª instr. Es necesario que el SV vaya en otro convoy ya que usa la misma UF que el LV.
Al tener un VLR de 64 y un $T_{chime} = 3$ (nº convoys) el tiempo de cálculo aprox. es de 3 x 64 = 192 ciclos.

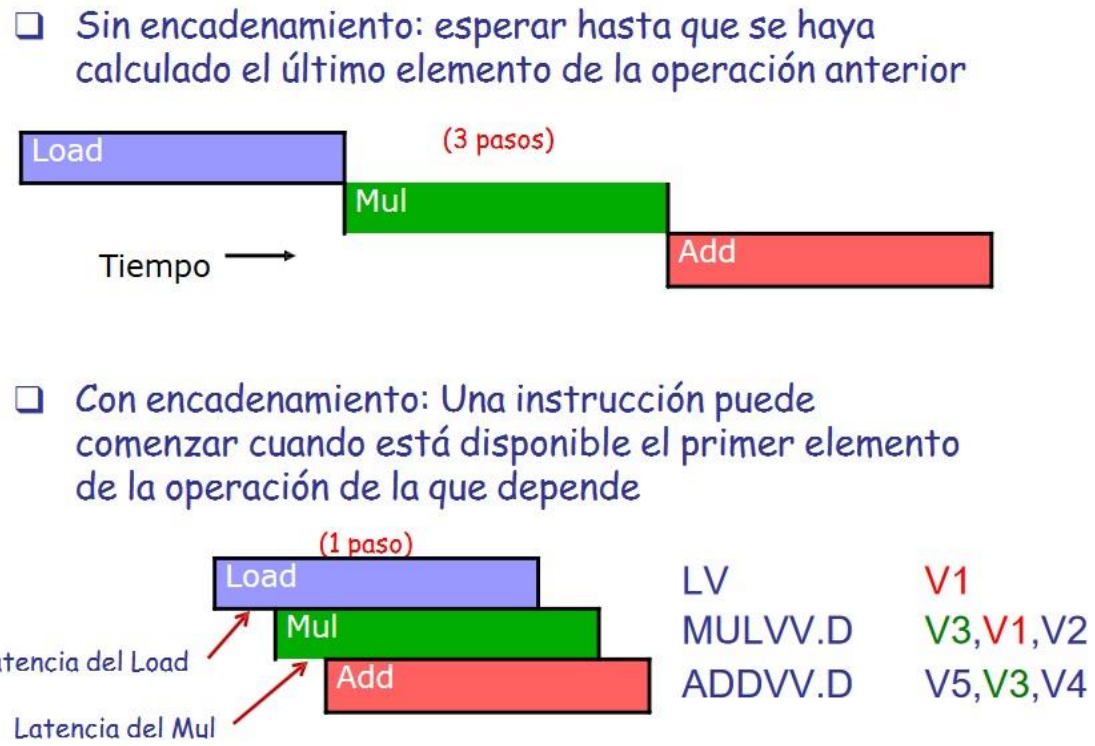
Para la ejec. de oper. aritméticas se hace uso de un pipe profundo que permite la reducción del ciclo de reloj, de esta manera segmentamos en muchas etapas para elevar la frecuencia, a pesar de que obtenemos una latencia mayor (nº de ciclos que se tarda en producir el resultado de una operación), esto se suele hacer frecuentemente en las arq. vectoriales.

Para la ejec. de oper. aritméticas es de vital importancia resaltar que la instr. que viene después de una anterior necesariamente debe empezar un ciclo después, haya o no conflictos, ya que solamente una instr. puede empezar en cada ciclo de reloj.

En cuanto a la ejec. de accesos a memoria normalmente ésta suele estar subdividida en una serie de bancos que permiten accesos múltiples de forma simultánea, es posible que se dé la situación de querer acceder por 2ª vez a un banco en concreto y que todavía no haya terminado el 1er acceso, en este caso deberíamos hacer paradas, estos conflictos dependen de la latencia.

Def. Memoria entrelazada: Se reparten los accesos entre los distintos bancos.

Def. Encadenamiento: Mecanismo que me permite hacerle llegar un dato a una instr. independiente que lo necesite tan pronto como esté listo (by-passing en arq. vectorial)
Si por ejemplo tuviésemos 3 instr. consecutivas con dependencias de LDE entre ellas, sin encadenamiento tendríamos que esperar a que se completase la oper. en su totalidad antes de acceder al dato, en cambio con encadenamiento permitimos la anticipación de operandos de las oper. en un solo paso.



Otro mecanismo usado en las arq. vectoriales es el procesamiento con vías múltiples, con el cual en vez de tener una UF de cada tipo se tienen varias UF's de cada tipo segmentadas, de manera que se aceleran los cálculos repartiendo el trabajo entre más UF's, es un mecanismo similar al de la mem. entrelazada.

MVL: Nº de elem. que caben en un reg. de la arq. vectorial, es un nº fijo que no puede cambiar y depende de la arquitectura.

Para determinar el tiempo de ejec. de un código vectorial se diferencian dos casos, dependiendo del tamaño del MVL:

- Vectores cortos: n <= MVL, se carga el reg. VLR y luego se ejec. la oper. vectorial.
- Vectores largos: n > MVL, es necesario descomponer la operación en varias suboperaciones (strip mining), se calcula el nº de operaciones necesarias de tamaño n / MVL y la operación vectorial restante (n mod MVL). Siempre se ejecuta primero la operación n mod MVL y luego tantas operaciones como n / MVL.

Existe un modelo de rendimiento a seguir para las operaciones por bloques (vectores largos), se tienen en cuenta:

- T_{base} : Tiempo de inicialización de las operaciones, en este modelo es despreciable.
- T_{start} : Nº de ciclos para el llenado de los pipes, depende de las oper. vectoriales del bucle.
- T_{loop} : Tiempo de actualización de punteros y detección de fin, siempre es 15.
- T_{chime} : Nº de convoyes en el bucle.

Con todos estos factores existe una ecuación que nos da el tiempo de ejec. de un procesador vectorial en función de n (siendo n el nº de componentes del vector):

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil * \left(T_{loop} * T_{start}\right) + n * T_{chime}$$

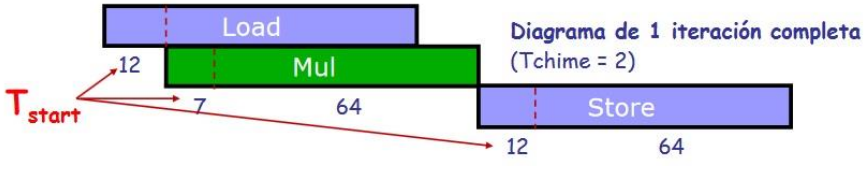
$\left\lceil \frac{n}{MVL} \right\rceil$ nos da el nº de iteraciones del bucle, redondeando siempre hacia arriba.

❑ Ejemplo: A= B x s, para vectores de 200 componentes

```
        DADDUI   R2,R0,#1600   ;total # bytes in vector
        DADDU    R2,R2,Ra      ;address of the end of A vector
        DADDUI   R1,R0,#8      ;loads length of 1st segment
        MTC1     VLR,R1        ;load vector length in VLR
        DADDUI   R1,R0,#64     ;length in bytes of 1st segment (8 elements)
        DADDUI   R3,R0,#64     ;vector length of other segments (64 elements)
Loop:   LV       V1,Rb         ;load B
        MULVS.D  V2,V1,Fs      ;vector * scalar
        SV       Ra,V2         ;store A (structural hazard)
        DADDU    Ra,Ra,R1      ;address of next segment of A
        DADDU    Rb,Rb,R1      ;address of next segment of B
        DADDUI   R1,R0,#512    ;load byte offset next segment
        MTC1     VLR,R3        ;set length to 64 elements
        DSUBU    R4,R2,Ra      ;at the end of A?
        BNEZ     R4,Loop       ;if not, go back
```

❑ $T_n = \left\lceil \frac{n}{MVL} \right\rceil \times \left(T_{loop} + T_{start}\right) + n \times T_{chime} =$

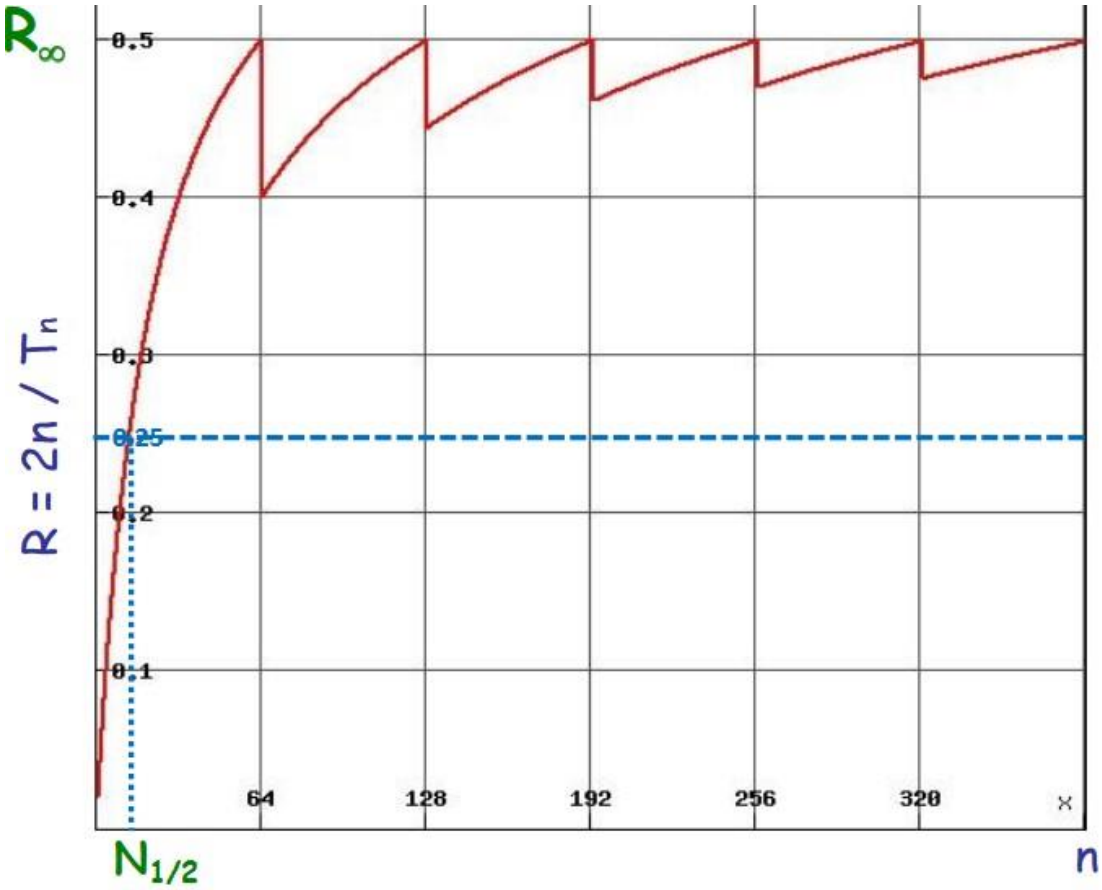
$= \left\lceil \frac{200}{64} \right\rceil \times \left(15 + (12 + 7 + 12)\right) + 200 \times 2 = 4 \times 46 + 400 = 584\ ciclos$



Def. Rendimiento asintótico (R_∞): MFLOPS (oper FP/seg) obtenidos para supuestos vectores de long. infinita, se calcula la misma ecuación que antes pero con n tendiendo a infinito ($n \to \infty$), una vez se obtiene el resultado de T_n se calcula: $R_\infty = \lim \mathrm{n} \to \infty \frac{\text{Nº Oper.vect. FP}}{T_n}$.

Def. Longitud del rendimiento mitad del asintótico ($N_{1/2}$): Long. que tendrían que tener los vectores para que el rendimiento obtenido con esa long. sea la mitad del rend. asintótico, suponemos siempre que se alcanza en la primera iteración por lo que $\left[\frac{n}{MVL}\right] = 1$.

En resumen, hay 3 medidas fundamentales para medir el rendimiento de las operaciones vectoriales, el tiempo de ejec., el rendimiento asintótico y la long. del rendimiento mitad del asintótico.



En la figura superior podemos observar cómo el rendimiento cuando n tiene a infinito es 0.5, y como cae el rendimiento a medida que se van completando iteraciones y aumentando el nº de elem. que ejecutamos.

Reg. de long. vectorial (VLR): Especifica en cada momento sobre cuántas componentes de los vectores tiene que aplicarse cada ejec.

Reg de mascara (VM): Permite la ejec. selectiva de los componentes de los vectores.

```
LV      V1,Rx       ;load vector X into V1
LV      V2,Ry       ;load vector Y
L.D     F0,#0       ;load FP zero into F0
SNEVS.D V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D V1,V1,V2    ;subtract under vector mask
SV      Rx,V1       ;store the result in X
```

En este ejemplo en concreto (arriba) cargamos el vector X en V1 (LV V1, Rx), también el vector Y en V2 (LV V2, Ry), y un 0 en el reg. F0, tras las cargas hacemos una comparación con la condición NE (Not Equal) con V1 y F0, para los elem. distintos de 0 en el VM ponemos un 1 (se cumple la condición), para las demás un 0.

Tras todo esto y a menos que reseteemos el VM (CVM) todas las instr. que vengan después (en este caso SUBVV.D y SV) se realizan bajo las condiciones del VM, es decir, solo para aquellas componentes que cumplan la condición de la comparación anterior y esten a 1 en el VM.

El sist. de mem. debe soportar un elevado ancho de banda debido a los accesos de lectura y escritura (LV y SV), para sobrellevar esto dispersamos los accesos dividiendo la mem. en bancos para acceder a uno o a otro en función de la dir., gracias a esto tenemos control independiente de las dir. de cada banco, esto permite hacer LV y SV de las palabras no secuenciales (3 tipos: LV, LVWS y LVI) y que múltiples procesadores vectoriales puedan compartir la misma mem., como contrapartida a todo esto el nº de bancos necesarios para dar servicio a los procesadores es demasiado grande, siendo necesarios por ejemplo aprox. 1400 bancos para 32 procesadores.

Para acceder a elem. equiespaciados en mem. se hace uso del LVWS con el cual accedemos a elem distanciados por un nº fijo de palabras, existe un problema con este tipo de loads, la repetición del acceso al mismo banco antes de que acabe el acceso anterior.

En caso de querer acceder a elem. que no están consecutivos ni equiespaciados se usan vectores dispersos con un índice (LVI), a la lectura de elem. dispersos en mem. (load) se le llama compresión (gather) ya que juntamos todos esos elem. dispersos en otro reg. vectorial, con el store hacemos lo contrario, expandimos una serie de datos que tenemos comprimidos en un reg. vectorial (scatter).

Def. Extensiones multimedia (SIMD): Tipo de arq. que nace como respuesta a características muy específicas de las aplicaciones multimedia, tratan de gestionar de forma más eficiente estas aplicaciones, que en general necesitan menos bits, sin usar una anchura tan grande como las que se usan en las arq. vectoriales, esta es la principal diferencia, ya que en arq. vectoriales se usan elem. de tamaño fijo (64), mientras que las SIMD usan long. variables (4 de 16, 8 de 8, etc).

La idea es realizar varias operaciones a la vez, dividiendo los datos en cadenas de 16 bits evitando los acarreos, comparado con las oper. vectoriales tienen una serie de limitaciones como por ejemplo que la long. de los vectores se codifica en el propio código de oper. (necesario especificar en cuántas cadenas se dividen los datos en el código de operación), no existe direccionamiento sofisticado (equiespaciado o disperso) ni VM.

Def. Unidades para procesamiento gráfico (GPU's): Nacieron para el uso de aplicaciones gráficas, pero hoy en día se usan para muchos más ámbitos, tienen una gran cantidad de elem. de computo.

¿Una GPU de NVIDIA es un multiprocesador compuesto por un cjto de procesadores SIMD MT, ¿qué significa esto?

Esto significa que cada uno de esos procesadores es capaz de ejecutar instr. vectoriales (aplica una oper. sobre varios elem. de forma simultánea con una sola instr.) y además es capaz de llevar a cabo multithreading (MT).

Similitudes y diferencias entre GPU's de NVIDIA y los procesadores vectoriales:

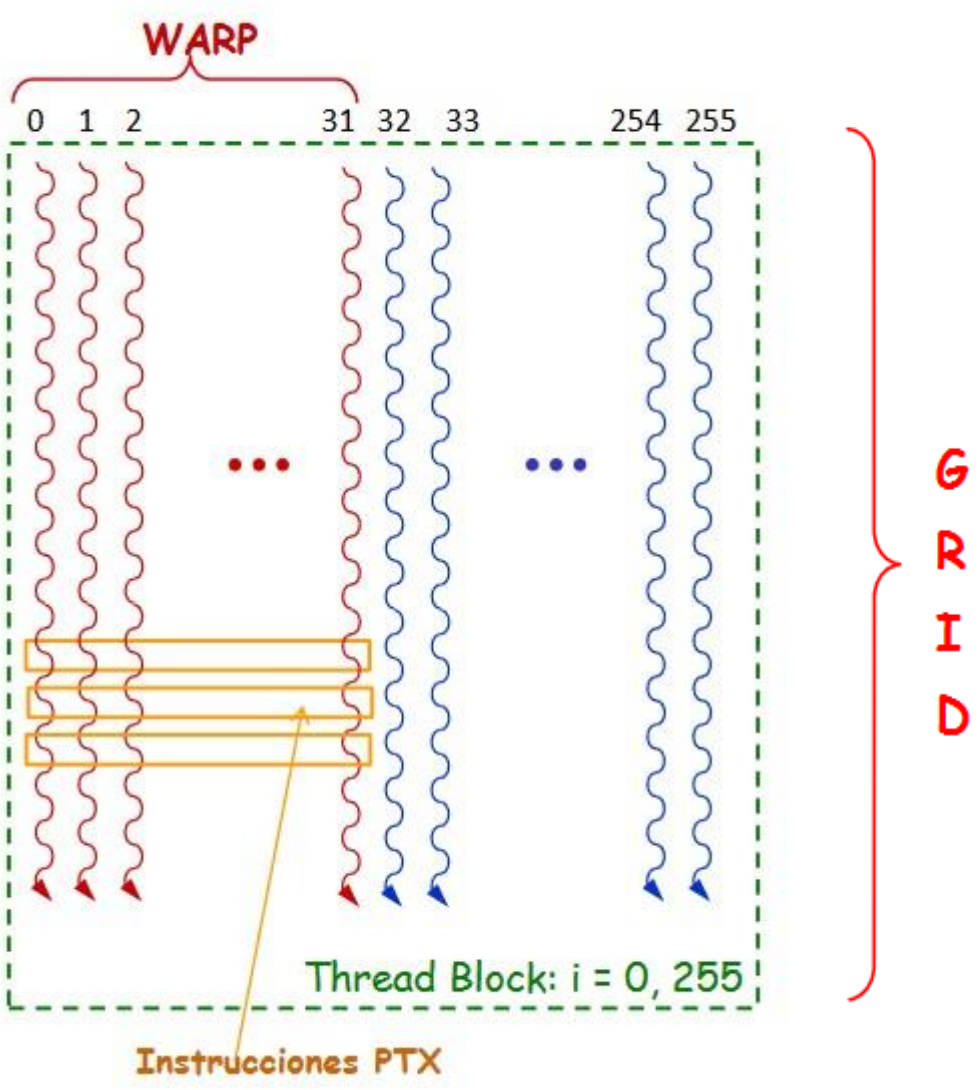
- Similitudes:
  - Funcionan bien en problemas con paralelismo de datos.
  - Permiten transferencias en mem. con scatter y gather (leer de mem. o escribir en mem. elem. equiespaciados o dispersos).
  - Comparten VM.
  - Es común la existencia de grandes ficheros de reg. (en las GPU hay muchos reg. disponibles mientras que en las arq. vectoriales no suele haber demasiados).
- Diferencias:
  - En una GPU de NVIDIA no hay procesador escalar.
  - Las GPU's usan multithreading para ocultar la latencia de mem, los procesadores vectoriales no.
  - Existen gran cantidad de UF en las GPU's, en las arq. vectoriales hay pocas UF muy segmentadas.

Def. CUDA Thread: Mínimo nivel de paralelismo que podemos extraer de una GPU, explotamos de forma simultánea el paralelismo a nivel de datos (instr. tipo vectorial) y el paralelismo a nivel de thread.
Es un modelo que produce C o C++ para el host (CPU) y un dialecto para la GPU, la idea básica es crear un thread independiente para cada elem. de los vectores a procesar, estos threads se agrupan en bloques que define el programador, cada bloque lo ejecuta un procesador distinto de la GPU, de esta manera varios bloques se ejec. simultáneamente en paralelo sobre varios procesadores, el cjto de bloques que implementan un cálculo vectorial sobre la GPU se llama Grid.

Def. Instrucción PTX: Tipo de instr. que realiza un mismo cálculo sobre varios datos (SIMD, igual que una instr. vectorial), opera sobre varios CUDA threads, en el caso de NVIDIA 32 datos, también se ve afectada por el reg. de VM.

Mientras que cada CUDA thread realiza una de las instr. PTX esta a su vez opera sobre 32 elem.

Def. WARP: Secuencia de instr. PTX que se pueden agrupar de forma que conforman un WARP, que el procesador ejecuta en modo multithreading (MT).



Si por ejemplo tenemos bloques de 256 threads (2^8), tenemos 8 posibles warps.

En este procesador se da paralelismo a nivel de threads y paralelismo a nivel de datos, una única instr. sobre múltiples datos (SIMD) que se agrupa sobre múltiples WARPS.

Ej procesadores GPU: El Fermi GTX 480 con 2 vías con 16 procesadores SIMD MT, cada procesador tiene asignado un bloque de threads por lo que se hace un reparto en paralelo de 16 bloques.

Toda GPU suele tener dos niveles de planificación, en el caso del Fermi GTX 480 el 1er nivel es el GigaThread planificador, que hace bloques a partir del grid y los asigna a cada procesador correspondiente, el segundo nivel es el planificador de threads, que en este caso tiene a su vez dos planificadores de threads en paralelo, uno por cada columna de vías de cada procesador.

Estos planificadores de threads seleccionan una instr. de cada uno de los 2 threads que puede ejec. de forma simultánea y la envían a los 2 cjtos de 16 vías físicas, al procesar 32 elem. cada instr. SIMD y ejecutarse sobre 16 vías necesitaremos 2 ciclos / instr, esto puede darse de forma desacoplada y desordenada.

En las GPU's existen unos registros especiales llamados reg. de predicado para componentes enmascaradas, existe una instr. PTX especial "SETP" (compara y pone a 1 un reg. de predicado) que permite implementar construcciones del tipo IF-THEN-ELSE, funciona de forma parecida a un VM, además de estas instr. PTX existen otras del mismo tipo pero de salto, que tienen el mismo formato, para preservar la máscara hay que hacer un apilamiento de las máscaras, es decir, necesitamos una pila semejante a la que teníamos pero con las direcciones de retorno de las subrutinas.
Para no perdernos y recuperar las máscaras es importante apilar (con push antes de entrar a una construcción IF-THEN-ELSE) y desapilar (con pop al salir de la construcción IF-THEN-ELSE) las máscaras, en la parte del ELSE tenemos que complementar los elementos de la máscara con COMP.

Es importante saber que primero se ejecutan los threads que no realizan el salto, es decir los que ejecutan el cuerpo del if, y una vez han acabado estos entonces los threads que saltan al else pasan a ejecutarse, complementando antes la máscara.

En resumen, las <u>diferencias más importantes</u> entre las arq. vectoriales y las GPU's son:

- El PC (Program Counter), en arq. vectorial solo necesitamos uno, mientras que en GPU's necesitamos varios porque en cada procesador tenemos 2 vías distintas que ejecutan distintos threads.
- En las GPU's hay 2 niveles de planificación mientras que en arq. vectorial no tenemos esta planificación.
- En las GPU's tenemos muchos más reg. disponibles que en un procesador de arq. vectorial convencional.

Las dependencias entre sentencias de una misma iteración no impiden la vectorización eficiente, se les llaman dependencias directas y obligan a ejecutar en el orden dado, en caso de que los datos sean dependientes de resultados generados en iteraciones previas se llaman dependencias en el espacio de iteraciones, y en algunos casos pueden impedir la vectorización y producir reordenamiento de sentencias, las dependencias de nombre pueden evitarse renombrando variables.

Tema 5 – Multiprocesadores y redes de interconexión

Def. Computador paralelo (multiprocesador): Computador con varios elem. de procesamiento que trabajan de forma cooperativa y que se tienen que comunicar entre ellos de forma totalmente necesaria.

Los computadores paralelos se crearon antes de que se diera el problema del consumo de potencia debido a factores como la necesidad de aumento del rendimiento, la garantía de la eficiencia (me importa que el computador sea capaz de resolver varias tareas por unidad de tiempo) o la replicación (necesidad de replicar la información para tener mayor tolerancia a fallos al estar almacenada en más de un lugar).

Se dice que un sist. es escalable si al aumentar el nº de procesadores otros factores como el ancho de banda, la mem., etc aumentan de la misma manera, si algún elemento se convierte en un cuello de botella (se satura) entonces no es escalable.

El problema que se solía dar consistía en que si encontrabas una técnica que mejorase el rendimiento también incrementaba el consumo de energía, y normalmente en un factor mayor, por lo que no compensaba usarla.

Def. Modelo de programación: Herramienta mediante la cual el programador tiene las capacidades necesarias para decirle al programa cómo hay que compartir la info., es necesario que el programador pueda invocar primitivas disponibles que estén en el espacio del SO y que correspondan con lo que se va a hacer a nivel HW.

Lo importante es el modelo que usa el programador, no el HW que hay detrás, existen 2 grandes modelos de programación, el de mem. compartida y el de paso de mensajes.

Def. Arquitecturas de mem. compartida: Consiste en tener un espacio de dir. de memoria compartidas entre todos los procesos, el objetivo es la productividad, en estas arquitecturas las comunicaciones son implícitas (lo que escribe un proceso es visto por los demás), en las sincronizaciones se invoca al SO o se usan operaciones atómicas.

Def. Multiprocesador de mem. compartida: Igual que en el caso anterior eran distintos procesos ahora podemos hablar de distintos procesadores, haciendo una extensión natural de un sist. en el que diferentes procesos comparten mem. añadiendo uno o varios procesadores.

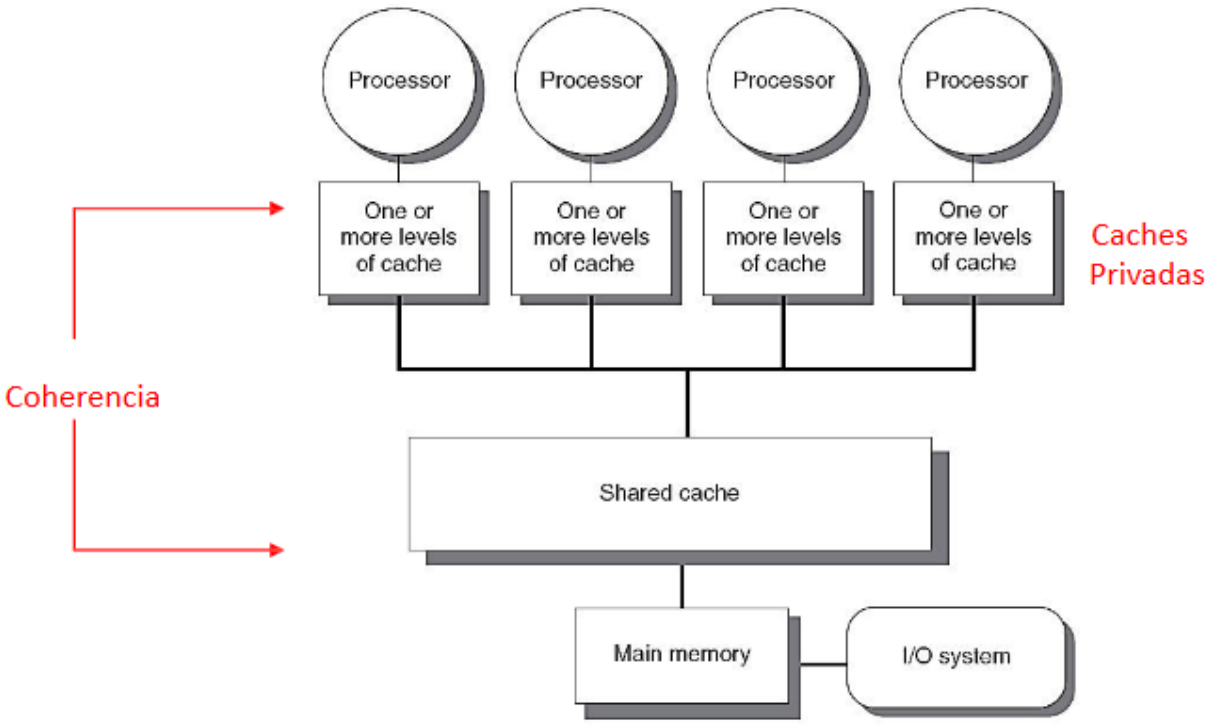
La red de interconexión usada en estos sist. depende del nº de procesadores, si tenemos pocos tenemos una red sencilla (bus), si tenemos muchos la red de interconexión en general es una red multietapa.

¿Cuántos procesadores se pueden conectar sin que la red se convierta en un cuello de botella que degrade el rendimiento global?

Depende de la red de interconexión, si tenemos una red que sea un bus no se puede escalar, ya que con muchos procesadores al final se colapsa y forma un cuello de botella, si en cambio tenemos otra red que no sea un bus entonces escalará mejor.

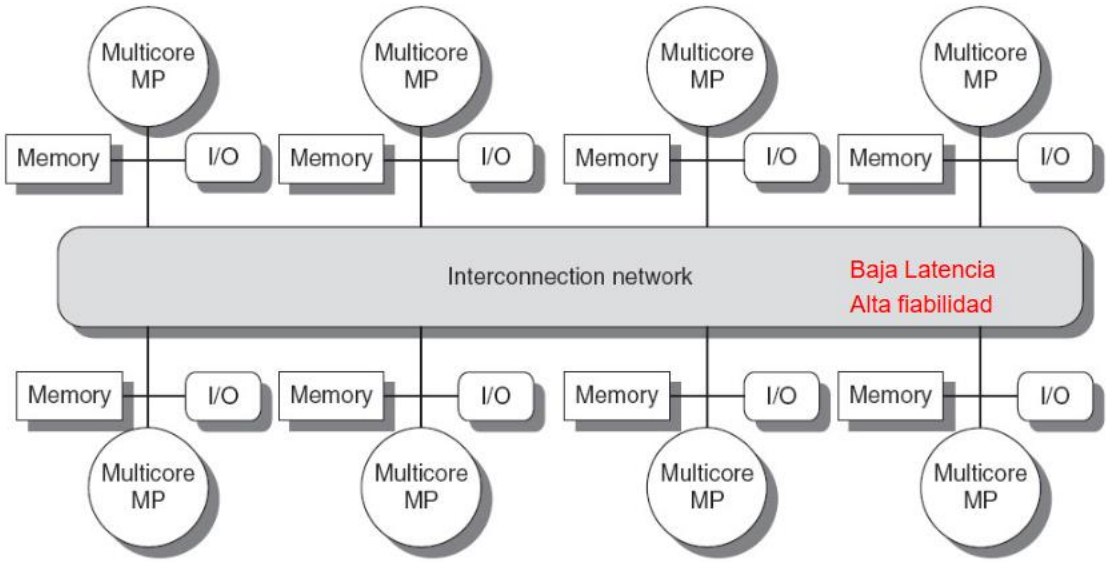
El diseño conocido para un nº reducido de procesadores se conoce como SMP (procesadores simétricos, ya que suelen ser los mismos procesadores y suelen tardar lo mismo en acceder a mem.), estos comparten una memoria (centralizada) y tienen un tiempo de acceso uniforme (UMA) independiente del procesador que necesita comunicarse con mem., en este diseño la interconexión no es escalable.

Además tienen características como un ancho de banda limitado o el problema de la coherencia de cachés (cada procesador tiene asociado uno o más niveles de caché privados (solo accesibles por cada procesador), además de estos existe un nivel de caché compartido entre todos los procesadores, el problema surge porque todos los procesadores pueden usar variables compartidas y modificarlas en su zona privada de caché, y es posible que otro procesador lea un valor antiguo de esa variable).



Para un gran nº de procesadores el diseño típico es con la memoria distribuida (se divide en bloques que se asignan a cada uno de los procesadores) y con tiempo de acceso no uniforme (NUMA), la interconexión si es escalable al contrario del diseño anterior.

Al estar la mem. distribuida si un procesador solicita un dato que reside en su región de mem. no necesita usar la red de interconexión y lo consigue de forma muy rápida, si en cambio pide el dato por la red de interconexión a otra región de mem. el tiempo que se tarda en obtener el dato será mayor, por eso es NUMA, el tiempo de acceso depende de si el dato está en su región de mem. o no.



Def. Arquitecturas de paso de mensajes: En estas arq. no hay memoria compartida, las comunicaciones son individuales punto a punto, a un proceso solo le puede interesar comunicarse con otro proceso en concreto.

El elem. de proceso en esta arq. es el microprocesador completo, por lo que el modelo de programación es distinto, se tiene acceso directo al espacio de dir. privado, a diferencia de lo que ocurre con la arq. de mem. compartida, la comunicación se establece mediante operaciones de E/S haciendo llamadas al SO usando dos primitivas (send/recv).

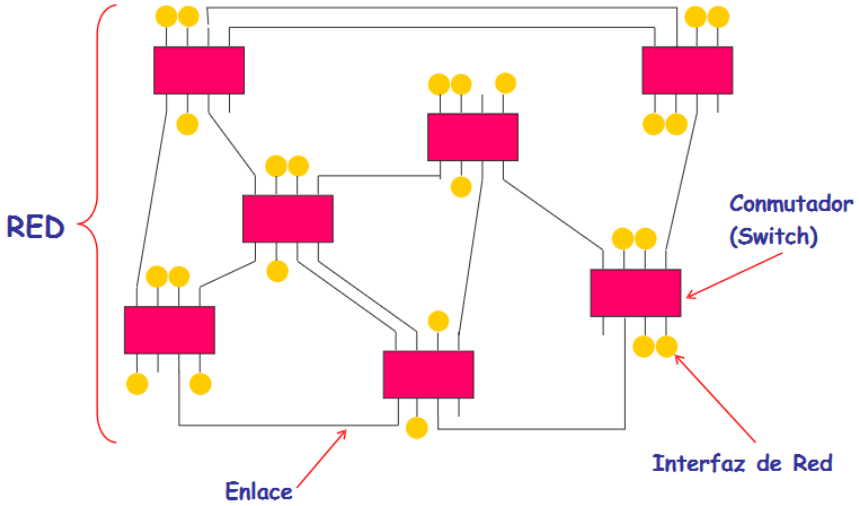
Existe una sobrecarga considerable en esta arq. respecto a la de mem. compartida ya que toda la comunicación es mediante msgs, y para cada uno hay que hacer una serie de acciones (construir la cabecera, copiar datos en un buffer, enviar el msg, etc).

Def. Redes de interconexión: Existen diversos tipos de redes de interconexión dependiendo del nº de dispositivos que queramos conectar en la red o del rango (espacio) que exista entre esos dispositivos, en función de estos factores existen redes OCN (Network On Chip), SAN (System Area Network), LAN (Local Area Network) y WAN (Wide Area Network).

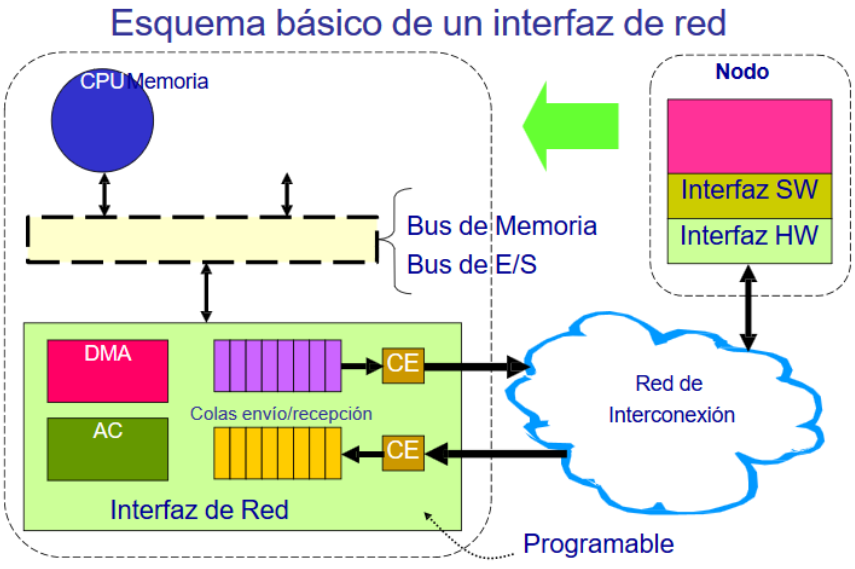
Existen varios factores de diseño muy relevantes a la hora de ver las prestaciones de la red, que son por un lado la latencia (tiempo que transcurre desde que se inicia el envío de un msg hasta que se completa el envío de ese msg) y por otro la productividad (cantidad de info. que es capaz de transmitir por unidad de tiempo), existen otros como la escalabilidad (posibilidad de aumentar el nº de procesadores de la red y que siga siendo eficiente), la fiabilidad (que si un enlace de la red falla se pueda garantizar que el msg llegará a su destino), etc.

Una red de interconexión está compuesta por los siguientes elementos:

- Enlaces (cables): Medio físico que me permite la transmisión de la información.
- Conmutadores (switches): Son repetidores que deciden (reasignan) qué camino debe seguir el msg en función de a dónde vaya.
- Interfaces de red: Encargadas de inyectar la info. en la red, tienen que ver con la comunicación existente en la red.



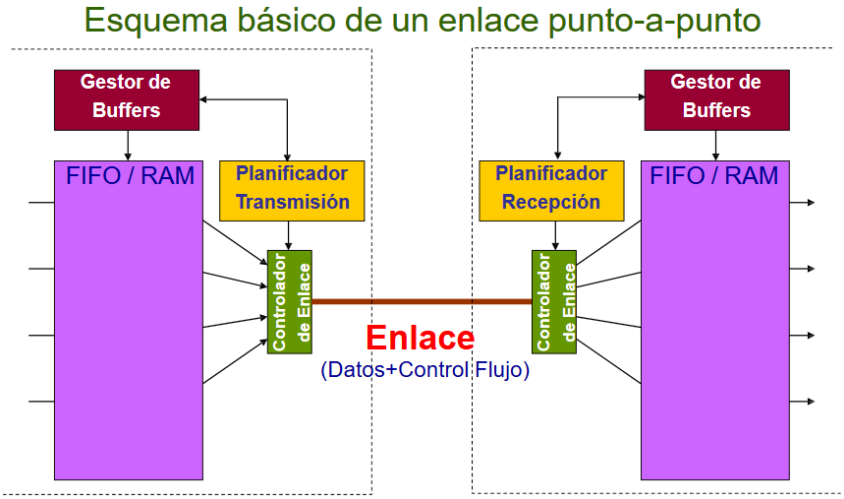
En las redes de interconexión a cada elem. de procesamiento se le suele llamar nodo, que contiene una CPU, una mem., un bus de E/S y una interfaz de red (encargado de inyectar una comunicación en la red, dentro del interfaz existen una serie de elem. como buffers de envío y recepción, asociado a cada uno de ellos tenemos un controlador de enlace (CE) que controla cómo es el envío o recepción del msg, además puede existir un asistente de comunicaciones (AC) que es otro procesador distinto a la CPU principal que la libera de todas estas tareas, puede existir también un DMA que se encarga del acceso a mem. y libera a la CPU de esta tarea.



Def. Mensaje: Unidad lógica de comunicación (lo que queremos transmitir), puede estar dividido en uno o más paquetes.

Def. Paquete: Unidad de transferencia entre interfaces de red.

Def. Enlace punto a punto: Conecta directamente dos unidades de procesamiento, tiene una serie de colas FIFO y de otro tipo que sirven de almacenamiento, también tiene un gestor de esas colas y un planificador que dice qué msg de qué buffer se envía o se recibe.



Def. Phit (w): Unidad de transferencia por ciclo de reloj.

Def. Flit: Unidad de control de flujo.

Un flit no puede ser más pequeño que un phit.

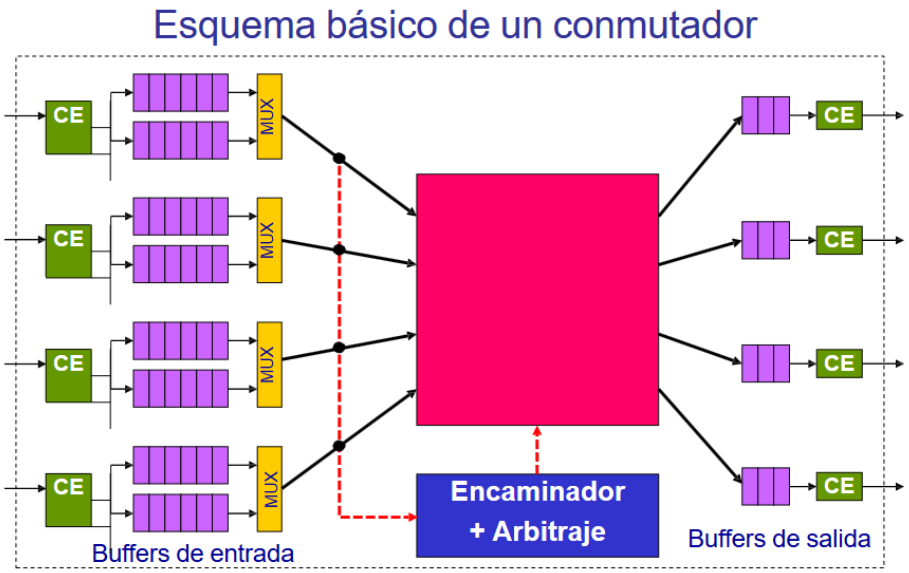
Un enlace tiene una serie de características:

- Longitud: Corta si solo tenemos un phit en el enlace o larga si tenemos varios phits en el enlace (envío segmentado).
- Ancho: Medido en phits (w).
- Velocidad de señal: $f = 1/\tau$
- Ancho de banda: Cantidad de info. que viaja por la red dividido entre unidad de tiempo.

Def. Conmutador: Elem. que redirige la info. que le llega por uno de sus canales de entrada haciendo que continúe por alguno de sus canales de salida.

En caso de que simultáneamente le lleguen varios msgs que quieran hacer uso de un mismo canal de salida es necesario arbitraje, que decide cuál es el 1er msg que tiene que salir por ese canal.

Además del encaminador existen una serie de colas de almacenaje donde se guardan los msgs que van llegando o saliendo del encaminador.



En algunas redes la función de un conmutador es la misma que la de un nodo (procesador que conforma la red de interconexión), tiene una serie de características:

- Grado de un conmutador/nodo (d): Nº de canales de entrada y salida.
- Implementación conmutador interno: Cómo está hecho por dentro.
- Buffers de entrada/salida.
- Lógica de control: Determina cómo se hace el encaminamiento de los msgs (por qué canal de salida debe salir un msg), mediante arbitraje + encaminamiento.

Def. Red de interconexión: Grafo en el que los vértices son conmutadores y nodos que están interconectados por canales de comunicación.
Def. Ruta: Secuencia de nodos y canales seguidos por un msg (todo lo que se atraviesa desde el origen hasta el destino).

Características de las redes de interconexión:

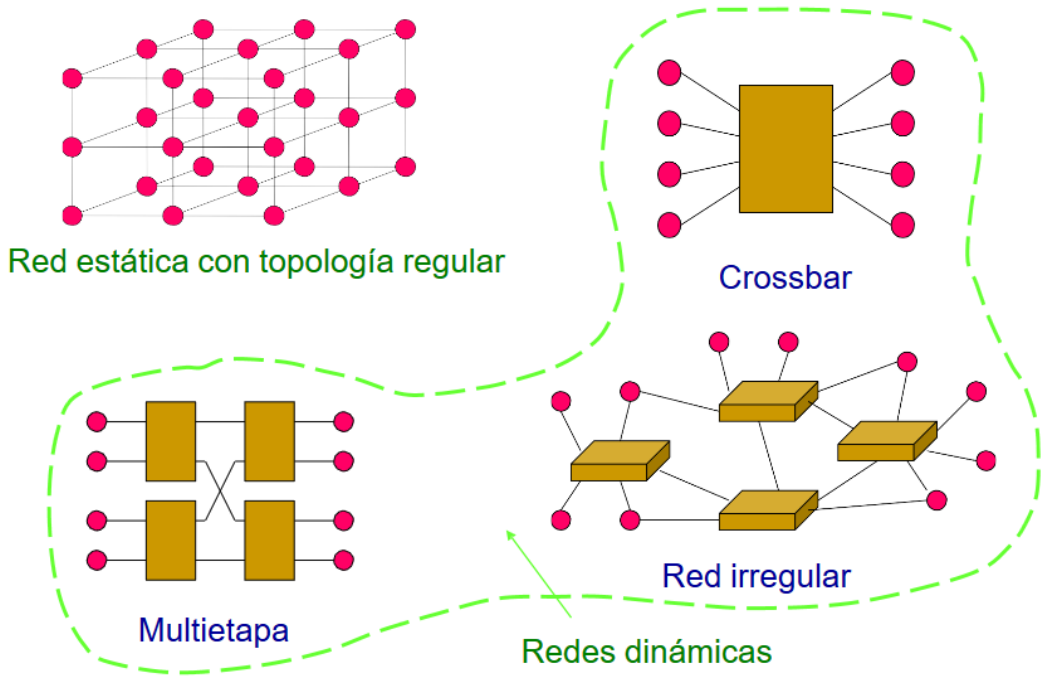
- Topología: Estructura de la interconexión física de la red, definida por su función de interconexión (nos dice con quién está conectado cada nodo).
  - Redes regulares (diseño homogéneo) vs Redes irregulares (diseño irregular).
  - Clasificación:
    - Medio compartido: Red a la que tienen acceso más de 2 nodos.
    - Estáticas: Red en donde todos los nodos están conectados mediante conexión directa a un nº fijo de otros llamados nodos vecinos.
    - Dinámicas: No tiene por qué haber conexiones con nodos vecinos, puede haber nodos que no estén conectados directamente con otro nodo.
- Conmutación: De qué manera se atraviesa la parte de la red que sigue el msg de origen a destino.
  - Conmutación de circuitos vs conmutación de paquetes.
- Control de flujo: En qué momento las porciones del msg se mueven a lo largo de la ruta.
- Encaminamiento: Por qué ruta van los msgs para llegar de un nodo origen al nodo destino.
  - Encaminamiento determinista: Dado un origen y un destino automáticamente tenemos un camino.
  - Encaminamiento adaptativo: Contempla distintas rutas entre un mismo nodo origen y otro destino y elige una u otra en función de varias variables.

Latencia en el contexto de redes de interconexión: Tiempo que tarda en enviarse un msg desde que el nodo emisor comienza el envío del msg hasta el momento en el que el nodo receptor contempla la recepción de ese msg.
Se compone de:

- Inicio de envío: Tiempo de preparación necesario hasta que se puede comenzar realmente el envío.
- Tiempo de vuelo: Tiempo que tarda el 1er bit del msg en alcanzar el destino.
- Tiempo de transmisión: Tiempo que tardan los datos en transmitirse.
- Latencia de transporte: Tiempo para que el msg pase a través de la red asumiendo que no hay contención y sin incluir las sobrecargas de inyectar ni de salir. Es la suma del tiempo de vuelo + el tiempo de transmisión.
- Latencia total: Inicio de envío + latencia de transporte + fase de recepción.

Propiedades de una topología de red:

- Grado de la red.
- Tamaño de la red: Nº de elem. de procesamiento (nodos) conectados.
- Nº de conmutadores: Nº de esos nodos que se encargan de redirigir el tráfico.
- Diámetro de la red: Máxima longitud de los caminos óptimos (los más cortos) entre pares de elem. de procesamiento.
- Simetría: Visión de la propia red de interconexión que tienen los nodos interconectados, si un nodo mira a los lados y ve lo mismo es una red simétrica, sino es asimétrica, en caso de que sea simétrica tenemos un patrón de tráfico uniforme que produce una carga uniforme de los canales de la red.
- Homogeneidad: Si los nodos son iguales, si tienen las mismas características.
- Regularidad: Si los nodos siguen un patrón de organización en la red (construida de igual forma en todos sus puntos) los nodos tienen que tener el mismo grado.
- Ancho de bisección: Nº de enlaces físicos que tenemos que cortar para dividir una red en dos mitades iguales.
  - Ancho de banda de bisección: Cantidad de tráfico por unidad de tiempo que tenemos en esos canales que necesitamos cortar.



En la imagen superior vemos que las 3 redes redondeadas son dinámicas ya que no hay conexión punto a punto entre todos los nodos, además son irregulares porque el patrón de organización no está dispuesto de la misma forma.

Ejemplos de redes estáticas:

- Anillo unidireccional: 4 nodos, la info. solo fluye en un sentido, la función de interconexión nos dice con qué se conecta cada uno de los nodos, el grado es 1, el diámetro es el camino más largo (p-1), simétrico.
- Anillo bidireccional: Igual que el anterior pero la info. fluye en ambas direcciones, el grado es 2, el diámetro se ha acortado respecto al anterior.
- Mallas: 4 funciones de interconexión, una por cada eje (positivo y negativo), el grado es 4, el diámetro es 6, no hay simetría.
- Toro: Se conectan los nodos de forma circular, mismo grado todos los nodos, si es simétrico y el diámetro es 4.
- Hipercubo binario: Red donde tenemos un nº de nodos = 2^n nodos, hay n funciones de interconexión, simplemente se complementa uno de los bits que es el que representa al nodo. Ej: Hipercubo de orden 3, nodo 4 ¿con qué nodos está conectado? 4 en binario -> 100, se conectará con 000, 110 y 101, con el 0, 5 y 6.
- Árboles binarios:
  - Árbol equilibrado: Mismo nº de nodos por la derecha y por la izquierda.
  - Árbol no equilibrado: Distinto nº de nodos por la derecha o por la izquierda.
- Topologías híbridas: Su objetivo es reducir el grado del nodo manteniendo un diámetro pequeño.

Ejemplos de redes dinámicas:

- Crossbar: Tipo de red que nos da una interconexión global, ya que cada una de las intersecciones representa un punto de conmutación (establece la conexión entre una entrada y una salida concreta). N entradas y M salidas, que permite hasta min (M, N) interconexiones punto a punto sin contención.
  - Propiedades:
    - Es posible una conexión siempre que los puertos origen y dest. estén libres.
    - El coste es proporcional al producto de NxM, caro y difícil de construir.
  - Aplicaciones:
    - Procesadores UMA, para conectar procesadores con un módulo de mem. compartido a través de un crossbar.
    - Para construir los conmutadores de otra red de interconexión.
- MIN (Multistage Interconnection Network): Conectan dispositivos de entrada a los de salida a través de etapas compuestas por conmutadores y patrones de conexión, cada conmutador puede conectar sus entradas con sus salidas en cualquier permutación deseada, se usan en redes multietapa.

  En las redes multietapa tenemos columnas de procesadores (etapas), entre cada una de ellas antes o después tenemos una serie de patrones de conexión o permutaciones que me dicen para cada entrada a qué salida se conecta el enlace. Siempre tenemos un patrón de interconexión más que el nº de etapas.

Dentro de las redes MIN existen varios tipos:

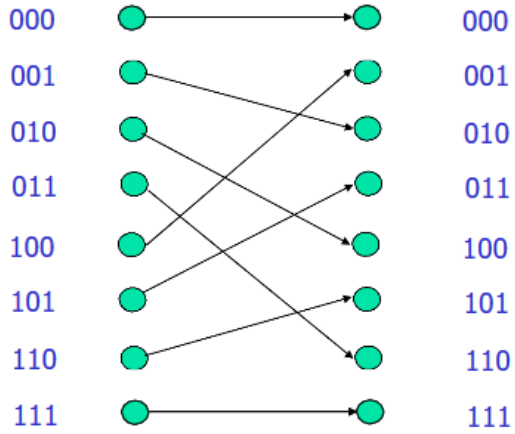
- Redes banyan: Red que permite solamente un camino entre un origen y un destino
- Red delta de N nodos: Subclase Banyan con nº de nodos $N = k^n$, siendo k el grado de cada computador y n el nº de etapas, N/k nos da el nº de computadores, que están conectados como crossbars con conectividad total.
  En aspectos de conectividad distinguimos dos tipos de redes:
  - Redes bloqueantes: Si enviamos un msg de un nodo origen a otro destino se establece una ruta que puede impedir el establecimiento de otras conexiones.
  - Redes no bloqueantes: El cjto de entradas puede conectarse con el cjto de salidas simultáneamente en cualquier permutación, similar a un crossbar.

Dentro de las redes MIN existen una serie de permutaciones, que definen cómo se conecta la salida de un conmutador con el siguiente conmutador de la siguiente etapa:

- Perfect shuffle (σ): Coge el bit más significativo y lo mueve a la posición menos significativa, estos bits identifican al nodo en la red, también existe la permutación inversa que hace lo contrario.

$$\sigma(x_{n-1}x_{n-2}\dots x_1x_0) = (x_{n-2}\dots x_1x_0\,x_{n-1})$$
$$\sigma^{-1}(x_{n-1}x_{n-2}\dots x_1x_0) = (x_0x_{n-1}\,x_{n-2}\dots x_1)$$

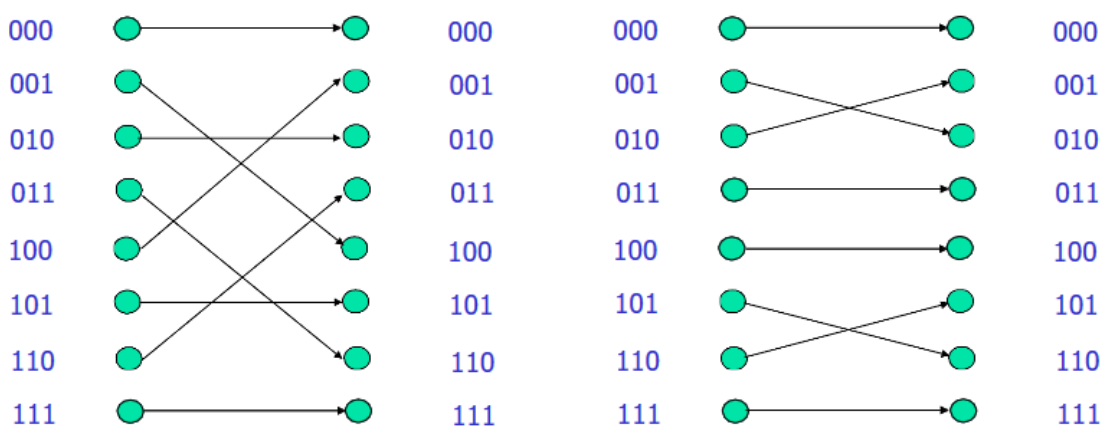
Ejemplo: N=8, n=3, k=2.



- Butterfly (β): Se intercambian los bits de la posición i con el menos significativo.

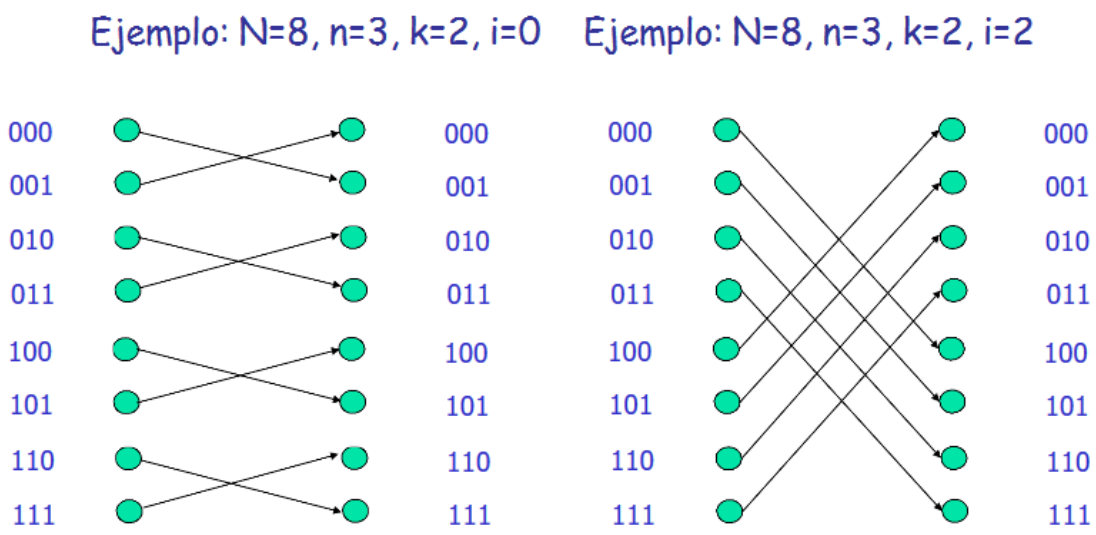
$$\beta(x_{n-1}\dots x_{i+1}x_i\,x_{i-1}\,\dots x_1x_0) = (x_{n-1}\dots x_{i+1}x_0\,x_{i-1}\,\dots x_1x_i)$$

Ejemplo: N=8, n=3, k=2, i=2 Ejemplo: N=8, n=3, k=2, i=1



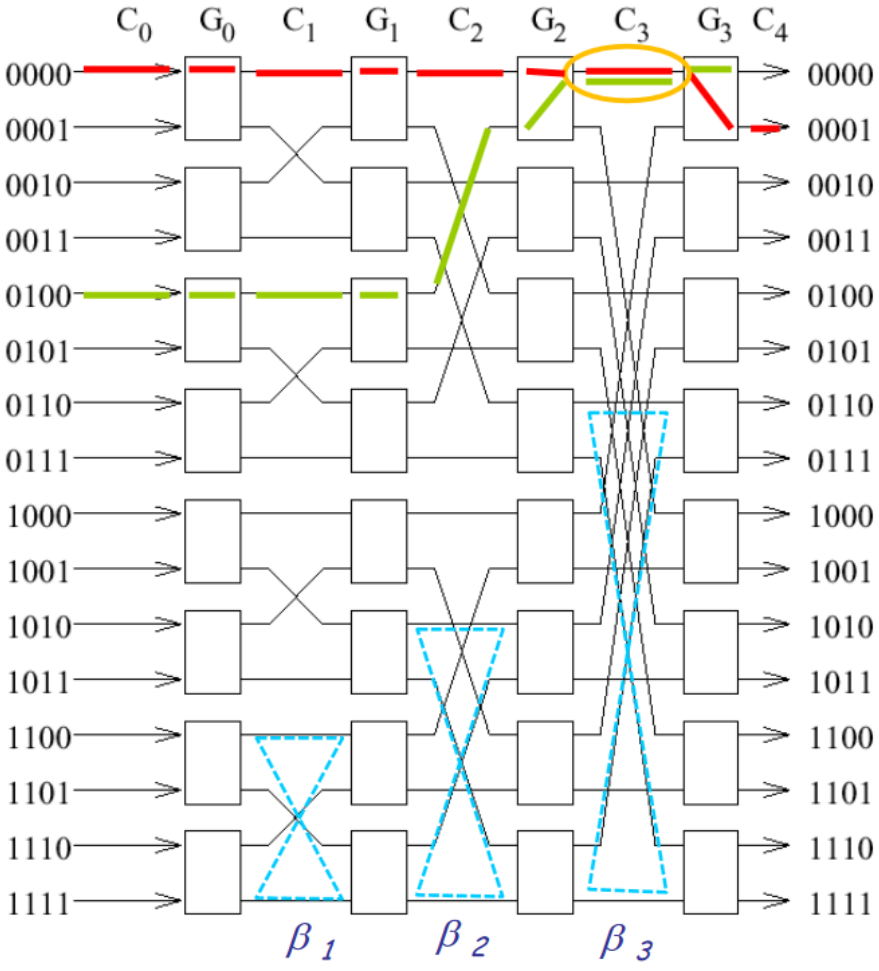
- Exchange (E): Para cada posición i se complementa el bit i.

$$E(x_{n-1} \dots x_{i+1} x_i \, x_{i-1} \dots x_1 x_0) = (x_{n-1} \dots x_{i+1} x'_i \, x_{i-1} \dots x_1 x_0)$$

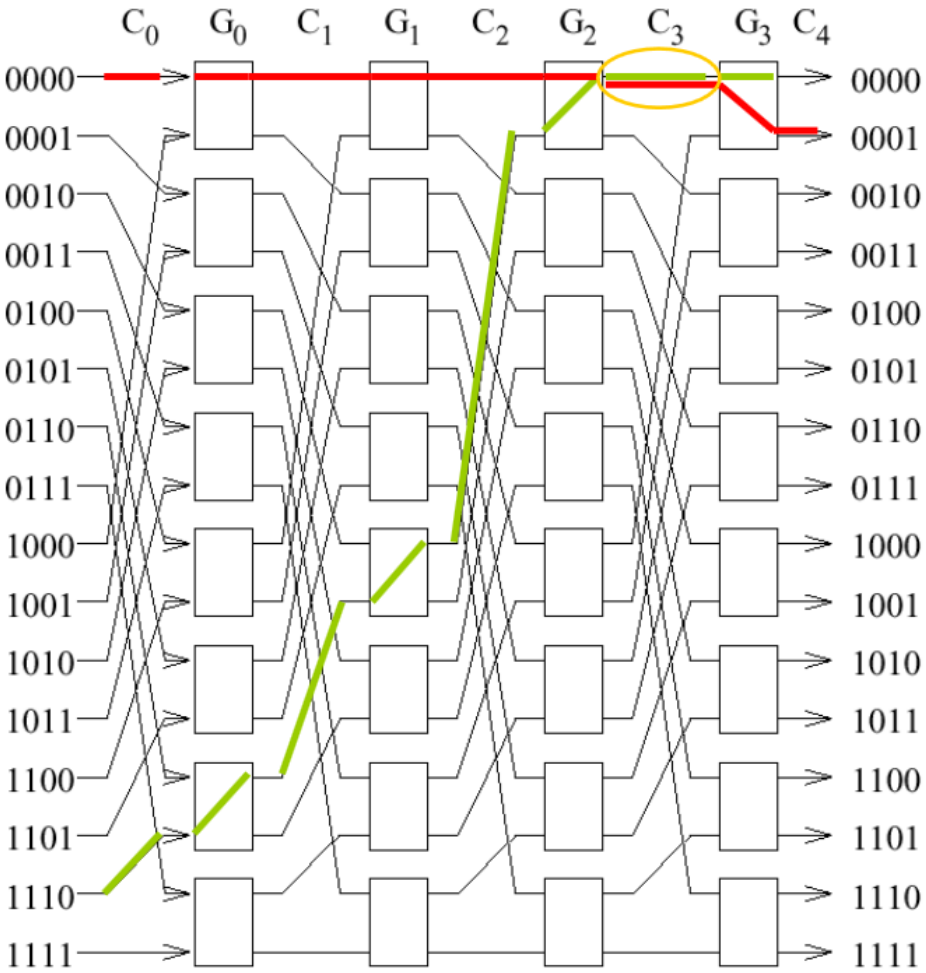


Redes MIN relevantes:

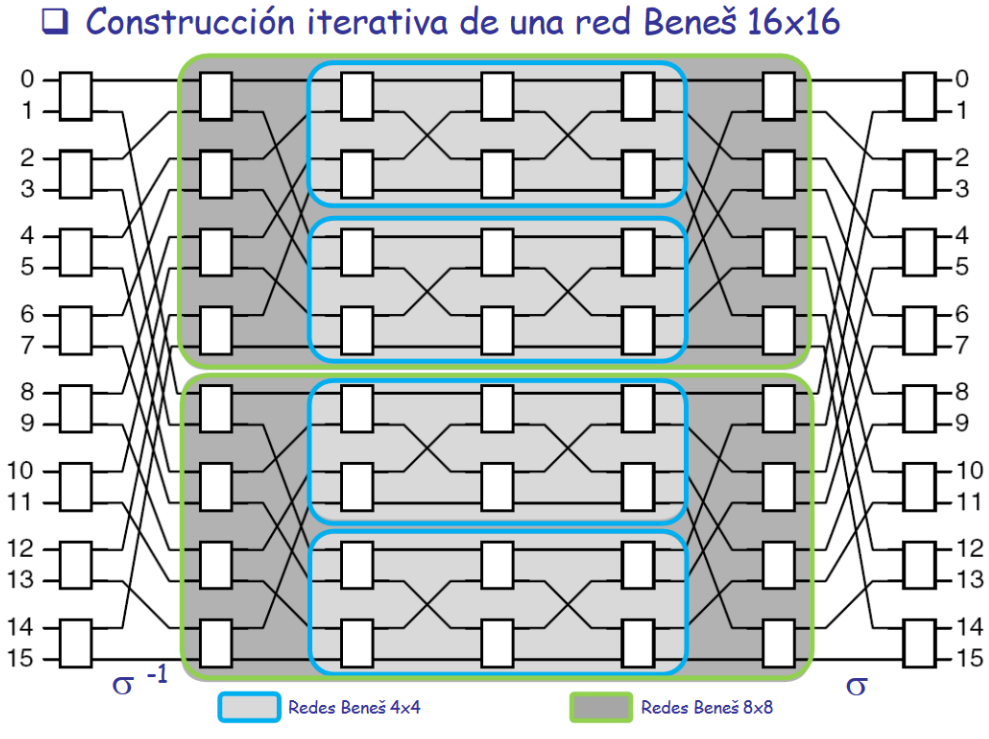
- Red MIN Butterfly: Se conectan los bits de la posición i con el menos significativo. Red Banyan (bloqueante), los patrones inicial y final usan permutaciones B0 (identidad, se conecta consigo mismo).



- Red MIN Omega: Los patrones de interconexión C0, C1, C2 y C3 siempre son los mismos, C_i -> σ, menos en la última, C_n -> identidad. Red Banyan (bloqueante).



- Red MIN Beneš: Red multietapa, no es una red banyan, tiene como misión evitar el bloqueo. Se pueden construir redes con mayor nº de elem. a partir de redes más pequeñas. $N = n^k$. Ej: 4x4: grado de cada conmutador (k) = 2, nº conmutadores = 2, nº de etapas = $2(\log_2 N) - 1$.



Def. Técnicas de conmutación:

- Conmutación de circuitos: Antes del envío del msg se establece cuál será la secuencia de enlaces que atravesarán los datos desde el origen al destino, los enlaces que componen este camino quedan reservados, no se pueden usar para otras comunicaciones. Un msg sin datos (cabecera) recorre el camino completo y va reservando enlaces, cuando se alcanza el destino se devuelve un ACK al origen por el mismo camino.
- Conmutación de paquetes Cada decisión de enrutado se va tomando según el msg avanza por la red, los enlaces no usados en un momento dado pueden usarse para otra comunicación. Existen variantes como Store-and-forward o Cut-through.

En este tipo de técnicas es importante tener en cuenta la latencia, para la cual se supone que no hay congestión en la red, el msg atresará h enlaces, el ancho de banda es B y el tiempo para decidir cuál debe ser el siguiente enlace a recorrer y establecer la conexión es Δ.

- Latencia con conmutación de circuitos: $T_{cc}(n,h) = h\Delta + n/B$, tiempo previo para decidir la ruta más tiempo de transmisión del msg completo.
- Latencia con Store-and-forward: $T_{sf}(n,h) = h(\Delta + n/B)$, el msg completo se transmite de un nodo al siguiente, cuando se ha recibido se decide el siguiente enlace y se vuelve a transmitir, h transmisiones + h decisiones.
- Latencia con Cut-through: $T_{ct}(n,h) = h\Delta + n/B$, el paquete está dividido en trozos de forma natural (flits), cuando se recibe el 1er flit se decide el siguiente enlace y se comienza la retransmisión sin esperar a la recepción de los demás flits.

Un problema básico que puede darse en la gestión de paquetes de una red es la contención, por lo que toda red debe estar diseñada de manera que sea capaz de evitar los bloqueos, en caso de que se produzcan es necesario un mecanismo de arbitraje, dependiendo de la técnica de conmutación que se utilice el paquete bloqueado puede recibirse y almacenarse en buffers del encaminador (store-forwad) o bien solo almacenar en los buffers unos pocos flits del msg bloqueado de manera que el resto permanece en los encaminadores previos sin avanzar (cut-through- wormhole)

Mientras el ancho de banda que demandan los procesadores va aumentando, la latencia va creciendo hasta que llegamos a un punto llamado punto de saturación de red, que es donde crece de manera muy exponencial.

Un multiprocesador es un sist. cerrado, esto significa que al aumentar el ancho de banda demandado aumenta la latencia de las comunicaciones, por lo que se reduce el avance de los programas (ya que estos necesitan datos de la mem. o de otro procesador) y estos cada vez hacen menos peticiones, la consecuencia final es que se reduce el ancho de banda, en resumen, aumentar el AB demandado al final acaba reduciéndolo.

Def. Encaminamiento: Determina qué ruta que debe seguir un msg desde el origen hasta el destino, implicando poca latencia y haciendo una distribución de la carga (repartiendo el tráfico de manera uniforme) con tolerancia a fallos (si un enlace se estropea que el msg siga llegando) y libre de interbloqueos (que la ruta no los genere).

Existen varios mecanismos de encaminamiento:

- Aritmético: Determina el siguiente tramo de la ruta mediante oper. aritméticas.
- Selección de la ruta en origen: La info. de la ruta acompaña al paquete, esto incrementa la long. del msg (problema).
- Mediante tablas: En cada nodo tenemos una tabla de rutas de manera que cada msg que llega a un determinado nodo accede a una determinada entrada de la tabla de rutas y ahí obtiene el enlace por el que tiene que salir.

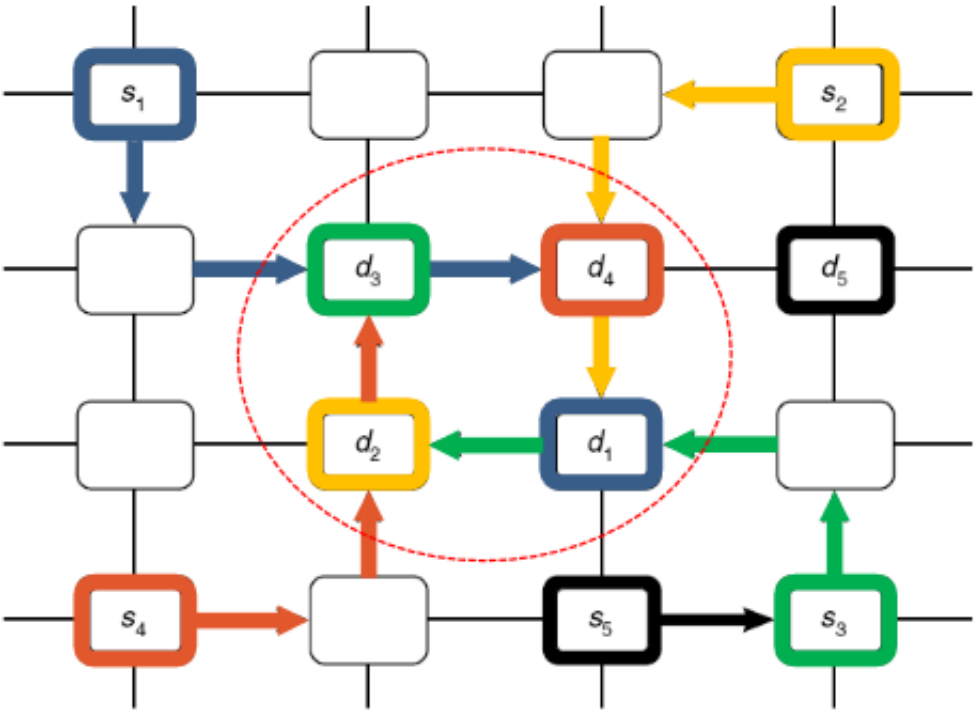
Propiedades del encaminamiento:

- Mínimo / No mínimo: Siempre se escoge la ruta más corta o no.
- Determinista / Adaptativo: Siempre la misma ruta de origen a destino o la ruta varía adaptándose a las circunstancias del tráfico de la red.

Existen una serie de problemas en el encaminamiento:

- Situación de interbloqueo: Ocurre cuando un determinado msg o paquete está a la espera de un evento que es imposible que ocurra.

En la imagen inferior vemos una red (malla 2 dimensiones) en la cual cada cuadrado representa un conmutador y queremos enviar 4 msgs simultáneos de s_i a d_i.
En los primeros ciclos de reloj los msgs avanzan cómo se indica en la figura, hasta llegar todos al centro de la red, en esta situación a cada msg solo le falta un salto más para llegar a sus respectivos destinos, pero ninguno llega porque cada uno está esperando en un enlace a que otro paquete deje disponible un buffer de entrada, están formando un ciclo entre ellos, ya que cada uno está a un paso de la meta pero nadie va al siguiente porque ninguno libera el buffer de entrada.



Existen otras situaciones parecidas al interbloqueo como la espera indefinida (paquete a la espera de un evento que puede ocurrir pero que no ocurre) o ciclos infinitos (un paquete está recorriendo nodos por la red pero nunca llega a su destino).

Para evitar los problemas como el interbloqueo:

- Orden dimensional: Existen dimensiones preferentes, los enlaces en una cierta dimensión no pueden ser usados por un paquete hasta que no ha recorrido todos los enlaces necesarios para alcanzar su dest. en todas las dimensiones precedentes, es decir, un paquete solo se puede mover por una coordenada cuando llega a otra que coincide con el destino en esa dimensión. Ej. Mecanismo encaminamiento usando oper. aritméticas.

Una de las redes estudiadas es la red MIN Omega (permutación perfect shuffle), cada bit de la dir. destino (di) se usa para seleccionar el funcionamiento de los conmutadores de la etapa n -i -1, por ejemplo, d2 se usa para la etapa 0 (primera).
Ej: los msgs que iban al nodo 6, dir. destino en binario (110), un 0 indica salida superior y un 1 inferior, así con el 6 sería salida inferior, inferior y superior, para las 3 etapas existentes (de la primera a la última) solo depende del nodo de salida.

Otra es la red MIN Butterfly (permutación beta), se intercambian los bits di con el menos significativo, la dir. destino (d3, d2, d1, d0) se usa para el funcionamiento de los conmutadores en la etapa n – i.
Ej: camino de 2 a 12: 12 es 1100 en binario, por lo que, d1 -> etapa 0 (0, superior), d2 -> etapa 1 (1, inferior), d3 -> etapa 2 (1, inferior), d0 -> etapa 3 (0, superior).

Tienen que existir una serie de condiciones para que se produzca el interbloqueo:

- Tiene que haber un recurso compartido al que se quiera acceder simultáneamente.
- Tiene que ser adquirido el recurso incrementalmente, es decir, primero se adquiere el de envío y después el de recepción.
- No puede tener un mecanismo de expulsión.
- Agentes de espera forman una cola circular.

Maneras de evitar el interbloqueo:

- Imponiendo restricciones sobre qué se tiene que producir para que se adquiera un canal.
- Aportando recursos adicionales (canales virtuales, se duplican los buffers dando una vía de escape, por si necesitamos escapar de allí, de esta manera no se aumenta el nº de enlaces en la red, un paquete que entra por un canal usa siempre el mismo a menos que hagamos un giro norte-oeste, en ese caso cambia y liberamos un recurso (tramo).

Tema 6 – Multiprocesadores: coherencia, sincronización y consistencia

Def. Jerarquía de mem. extendida: La jerarquía de mem. engloba distintos niveles (mem. principal, secundaria, cache…etc), siempre se representa usando una pirámide en la cual en el nivel más alto está el procesador, luego los niveles de caché y luego la mem. principal, cuanto más arriba estamos más rápido obtenemos los datos pero más caro es el acceso a ellos.

Cada uno de los procesadores tiene uno o más niveles de caché privados, donde solo pueden acceder ellos, estos mecanismos permiten ocultar la latencia de los accesos (menos ciclos, ganamos rendimiento) y permiten reducir la contención (si hay 2 CPU's que quieren acceder a un mismo dato y cada una lo tiene en su caché privada evitamos que compitan en el acceso a la mem. principal y el retraso de ciclos).

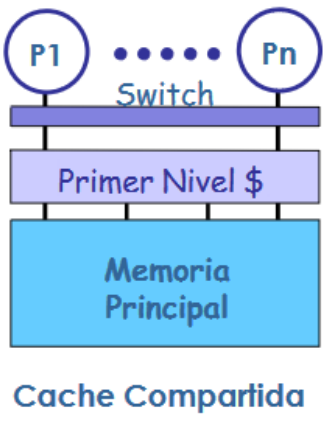
Como contrapartida existe un problema de coherencia que tiene que ver con la replicación de los datos, es decir, hay variables compartidas que residen en mem. principal pero a las cuales pueden acceder distintas CPU's, cada una de ellas tiene sus variables privadas y además variables compartidas a las que acceden otras CPU's.

La coherencia hace referencia a la correcta actualización de las variables compartidas.

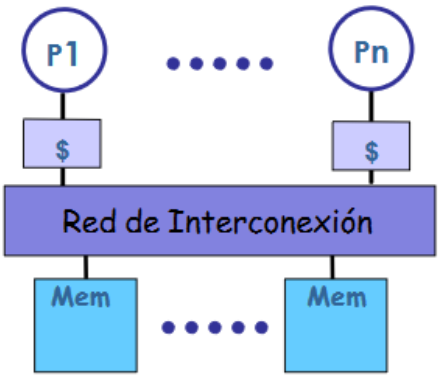
Otro problema existente es el de la escalabilidad, que hace referencia al nº de procesadores que podemos dotar a los sist. sin tener problemas adicionales.

Existen varias alternativas de implementación en cuanto a la memoria:

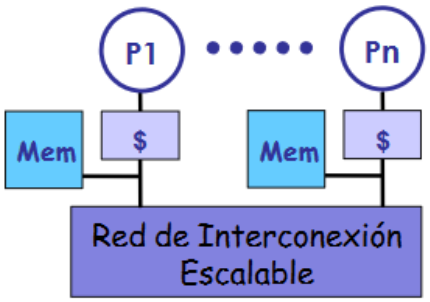
- Caché compartida: Elimina directamente el problema de la coherencia, ya que tenemos un 1er nivel de caché compartido por todos los procesadores, si alguno modifica una variable compartida todas las CPU's lo van a ver, no es válida por su baja escalabilidad.
- UMA: Tenemos distintas CPU's, algún nivel de caché privado y una red de interconexión que nos permite llegar a una mem. compartida, suele ser un bus o una red punto a punto, en la cual el acceso es uniforme (UMA).
- NUMA: La mem. está fragmentada entre los distintos procesadores, de manera que solo se usa la red de interconexión si el dato que se solicita no está en su porción de mem. privada, estos sistemas son más escalables pero más complejos a la hora de lidiar con el problema de la mem. caché.



Cache Compartida



UMA, SMP



NUMA: Memoria Distribuida

Def. SMP: Multiprocesador simétrico, se llaman así porque suelen ser iguales y porque el coste de acceso de una CPU a una región de mem. siempre es el mismo, es una alternativa muy común ya que son el bloque básico para construir sist. más grandes.
Existen dos problemas con este tipo de sistemas, uno es la coherencia, ya que todas las CPU tienen niveles privados de caché y hay que actualizar de manera correcta las variables compartidas y otro el tener un bus, ya que tenemos capacidad limitada y si usamos muchos procesadores puede que el sistema se acabe saturando.

El problema de la coherencia se da cuando ejecutamos programas en sist. multiprocesador, ya que sino el problema no existe.

Def. Modelo intuitivo de mem.: Es la referencia que se usa para "comunicar valores" entre distintos puntos del mismo programa, en sist. multiprocesador siempre se ha de leer el último valor que se haya modificado en una variable compartida.

Un único procesador puede ejecutar varios programas (multiprogramación), al leer una posición de mem. se devuelve el último valor escrito en ella, siempre se lee lo último al margen de qué thread lo haya escrito.

En el caso de los sist. con un único procesador las cachés no interfieren ya que aunque tengamos distintos niveles de jerarquía de mem. todos los programas ven exactamente lo mismo en mem. por lo que no existe el problema de coherencia en mem. caché.

Si tenemos en cambio un sist. multiprocesador de mem. compartida si existe el problema de la coherencia caché ya que tenemos cachés privadas y no visibles para algunas CPU's, el objetivo es que al ejec. un programa con varios threads al final de la ejecución tenga el mismo resultado independientemente de si los procesos se han ejec. de forma secuencial o entrelazada, independientemente del orden el resultado debe ser el mismo, el problema se da cuando dos programas acceden a cachés diferentes ya que puede pasar que vean valores inconsistentes (distintos).

Este tipo de problemas de coherencia se evita mediante políticas (protocolos) de coherencia, según los cuales si un procesador cambia el valor de una variable los demás que quieran leer tienen que esperar para leer el último valor.

En sist. monoprocesador pueden existir problemas de coherencia si tenemos dispositivos DMA (acceso directo a mem.) en operaciones de E/S, ya que pueden escribir en mem., que la CPU no se percate y siga leyendo valores antiguos en la caché, también en caso de que se use una política write-back de actualización (solo se actualiza la mem. principal cuando se desaloja un bloque de nivel inmediatamente superior que ha sido modificado) es posible que el DMA lea un valor antiguo.

Las primeras soluciones que se dieron a este problema fueron las "de grano grueso":

- Los segmentos de mem. que se involucren en oper. de E/S se marcan como no cacheables (no se les permite ser guardadas en caché).
- Sacar de la caché todo su contenido (flush) antes de cada oper. de E/S.
- Usar la caché directamente para E/S, así el tráfico pasa por todos los niveles de la jerarquía de mem. (latencia enorme y el contenido de la caché puede corromperse).

Hoy en día ninguna de estas soluciones se implementa al ser del todo ineficientes.

En sist. multiprocesadores existe el problema de que la escritura o lectura de variables compartidas es un evento frecuente, por tanto las soluciones pensadas para sist. uniprocesadores no son válidas.

Def. Sist. de mem coherente: Se tiene que garantizar tanto que una oper. de lectura tiene que retornar siempre el último valor que se haya escrito en esa posición (independientemente del procesador que efectúa la lectura o escritura) como la serialización de escrituras (todas las escrituras a una misma posición de mem. deben ser vistas por todas las demás CPU's en el mismo orden correcto).

Dentro de estos sist. existen dos aspectos muy relevantes:

- Coherencia: Qué valor debe ser devuelto cuando lea un procesador, debe garantizar que cualquier CPU que lee un dato de mem. debe leer el último que se haya escrito por cualquier CPU que haya accedido a esa posición de mem.
- Consistencia: Cuándo un valor escrito debe ser devuelto por una operación de lectura, este aspecto forma parte de la coherencia.

Existen dos grandes tipos de políticas que tratan estos aspectos, ambas actúan de formas diferenciadas:

- Invalidación en escritura: Al escribir en un bloque se invalidan todas las otras copias, es decir, existen múltiples lectores pero un solo escritor, con esta política escrituras consecutivas a un mismo bloque por una misma CPU se pueden realizar localmente, esto es bueno porque solo es necesaria una transacción en la red de interconexión (invalidar copia) cuando hay múltiples escrituras a un mismo dato.
  En un fallo de lectura distinguimos entre:
  - Write-back: Valor de bloque en caché y en mem. principal puede ser distinto, por lo que es necesario rastrear en cachés remotas para encontrar el último valor, una vez se encuentra se le da al solicitante y a la mem. principal.
  - Write-through: Valor del bloque en caché y en mem. principal es el mismo (siempre actualizado).
- Actualización en escritura: Al escribir en un bloque se actualizan todas las demás copias, normalmente solo se hace con políticas donde siempre se tiene actualizado el valor de un bloque tanto en caché como en mem. principal (write-through) y con pocos procesadores, ya que el tráfico es elevado al tener que actualizar demasiadas CPU's, en escrituras consecutivas todas las veces que se modifique un mismo dato se necesita notificarlo a todos los demás, en un fallo de lectura se busca en la mem. principal, ya que ésta siempre va a estar actualizada.

Las políticas aplicadas dependen de la red de interconexión del sist. multiprocesador:

- Si la red no permite broadcast las órdenes de invalidación o actualización se envían solamente a aquellas cachés que tienen una copa del bloque.
  - Protocolos basados en directorio: estructura HW donde tenemos una entrada por cada bloque en la cual se indica en qué cachés existe copia y el estado de la misma.
- Si la red permite broadcast las órdenes de invalidación o actualización se pueden enviar de forma simultánea a todos los controladores de caché.
  - Protocolo Snoopy: usa un bus como red de interconexión, cada controlador de caché está continuamente observando lo que sucede en el bus y tomando decisiones en función de esos eventos, el controlador de caché envía eventos al bus dependiendo de las peticiones de mem. que recibe de su CPU, puede enviar órdenes de actualización o de invalidación de forma simultánea a todos los demás controladores, también vigila los bosques de datos con sus correspondiente tags y su estado.

    En cuanto al envío de información a la red de interconexión (bus) es necesario un arbitraje para gestionar el orden en el cual se va poniendo la información. Los cambios de estado en este protocolo son provocados por operaciones LD/ST de la CPU y por las transacciones realizadas y observadas en el bus.

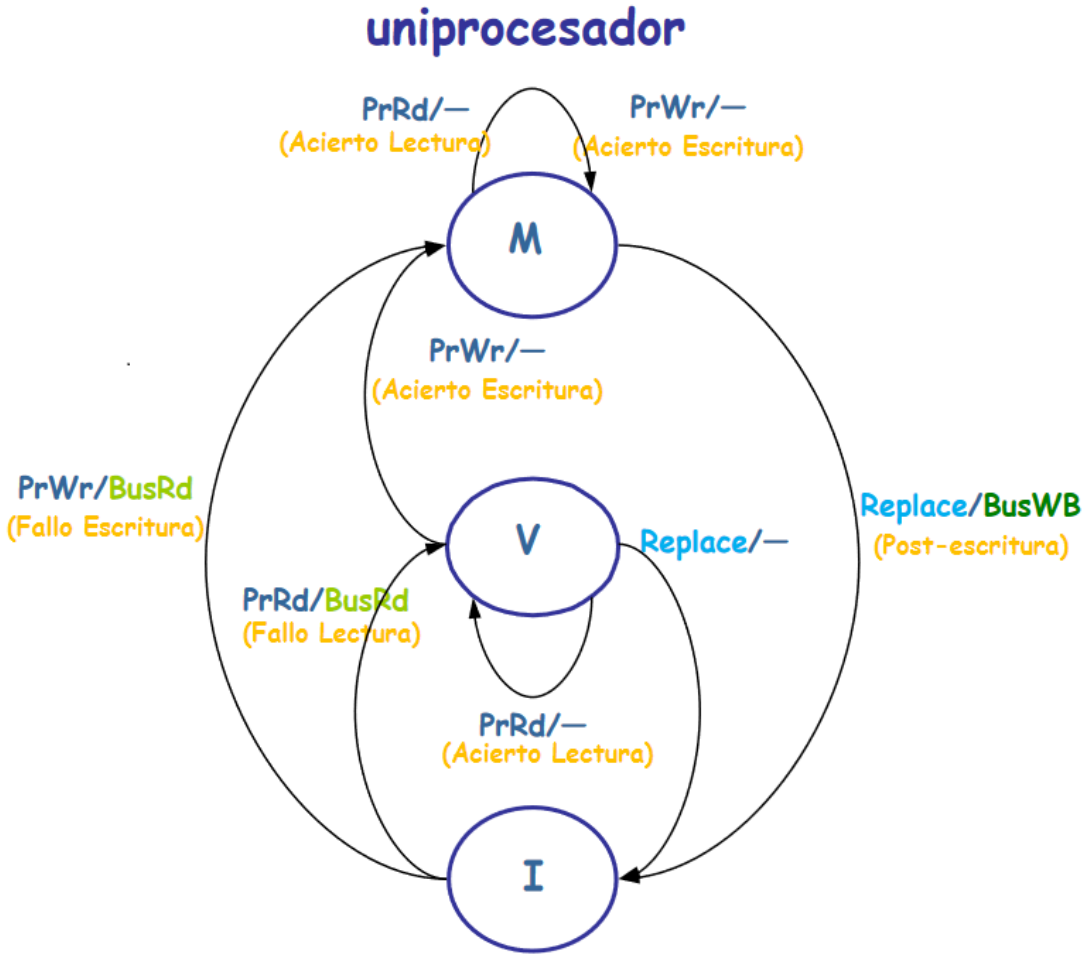
  - ➢ Protocolo Snoopy de 2 estados: Política de invalidación en escritura, con write through y sin asignación en escritura (si hay fallo de escritura nos llevamos el dato a mem. principal pero no a nuestra caché).
    - Uniprocesador: Tenemos dos estados, I (Invalido) y V (Válido), cuando no tienes un bloque en caché estas en estado Invalido, solo un fallo de lectura ocasiona un reemplazamiento y un cambio de estado I -> V.
    - Multiprocesador: En este tipo de sist. hay que tener en cuenta que otra CPU puede solicitar un dato para escritura y modificarlo, por lo cual la copia que tenía en cache cualquier otro procesador ha de ser invalidada.

  Existe un problema con el protocolo Snoopy si usamos write-through, cada vez que un procesador escribe en su caché también lo hace en mem. principal, suponiendo una transacción de escritura en el bus que consume ancho de banda, esto se traduce en muchas peticiones de escritura (store), por lo que si usamos más de 4 procesadores el bus se puede saturar y ralentizar el rendimiento del sist., por todo esto podemos usar otra política de escritura, como write-back.

En las cachés con política write-back normalmente se suele usar asignación en escritura, es decir, cuando ha habido un fallo de escritura nos llevamos el bloque a caché.

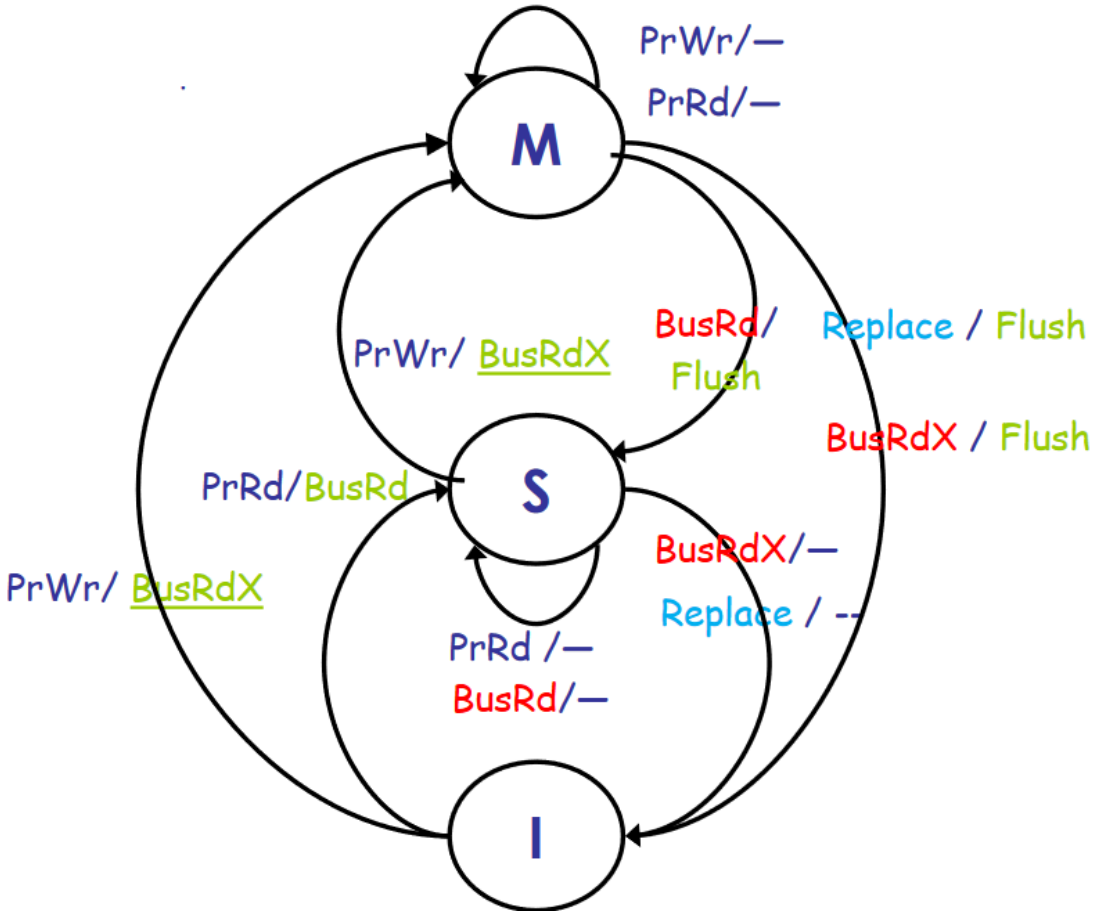
Al usar esta política añadimos un nuevo estado, modificado, para indicar si ese bloque que reside en la caché está modificado respecto a la copia de mem. principal, si es así lo llamamos bloque sucio, sino limpio, solo se modifica la mem. principal cuando se desaloja un bloque sucio.

Hay 2 tipos de acceso a mem. que implican transacciones en el bus, uno cuando hay lecturas provocadas por fallos de lectura o de escritura y otro cuando hay post escrituras (en una cache se reemplaza un bloque, por lo que está sucio y hay que actualizar la mem. principal).



Protocolo MSI: En sist. con varios procesadores existen tres estados distintos:

- Modificado (M): Implica que solo hay una caché en el sist. que tiene copia válida de ese bloque, todas las demás pueden tener copia pero en estado inválido, también la copia de la mem. principal está desactualizada por lo que es un bloque sucio.
- Compartido (S): El bloque está presente en la caché pero no ha sido modificado, está en uno o más procesadores, pero está limpio (coincide con la copia de la mem. principal).
- Invalido (I): El bloque no está presente en la caché.



Protocolo de Snoopy de 4 estados:

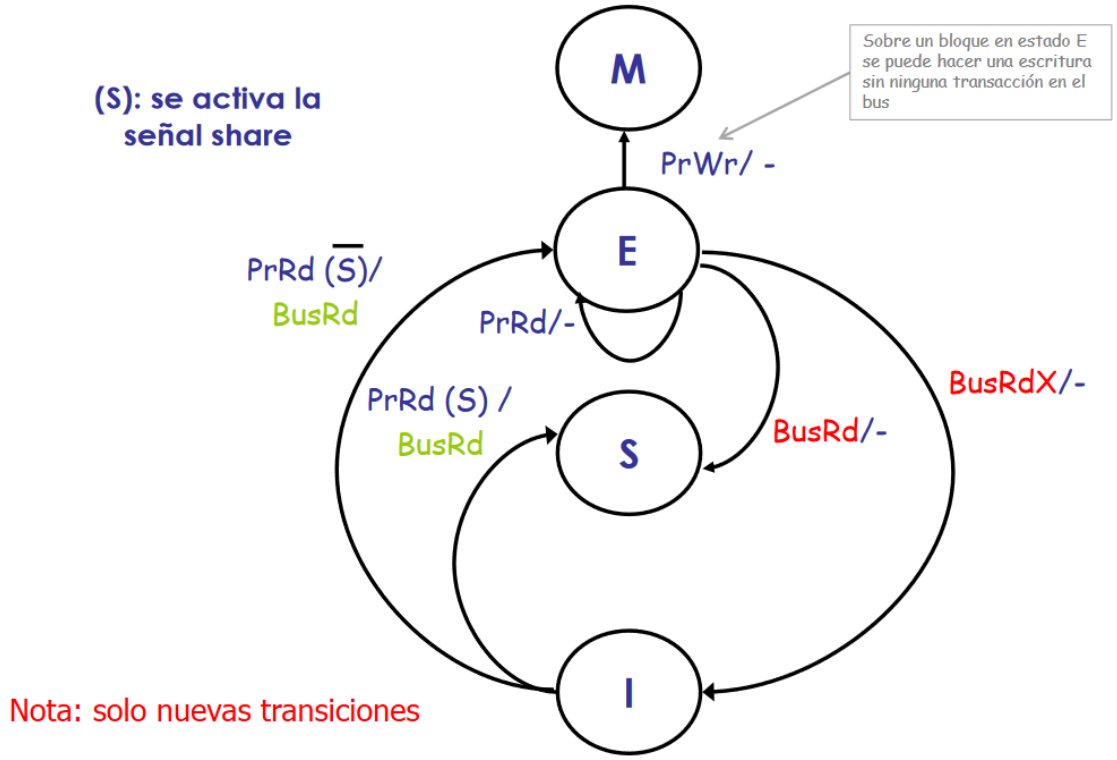
Carrera entre escrituras: no puede escribirse en la caché hasta que se tiene el bus disponible, es decir, varios procesadores no pueden escribir a la vez, ya que sino otro procesador podría tomar el bus antes y escribir el mismo bloque en su caché, para evitar esto se arbitra el bus para secuencializar las escrituras.

Todos los procesadores deben acceder al bus, y tienen que chequear tanto datos como dir. (tags), si hay coincidencias entonces habrá que invalidar o actualizar, esa consulta de datos y tags interfiere con el uso de la caché por parte de la CPU por lo que se hace en paralelo con la ejec. del programa, duplicando los tags.

Protocolo MESI: Tiene un nuevo estado llamado exclusivo, y arregla el problema del protocolo MSI, según el cual es muy común que los programas lean un dato y luego lo modifiquen, esto genera 2 transacciones en el bus (cuantas más evitemos menor tráfico y retardo) al leer el bloque y al escribirlo.

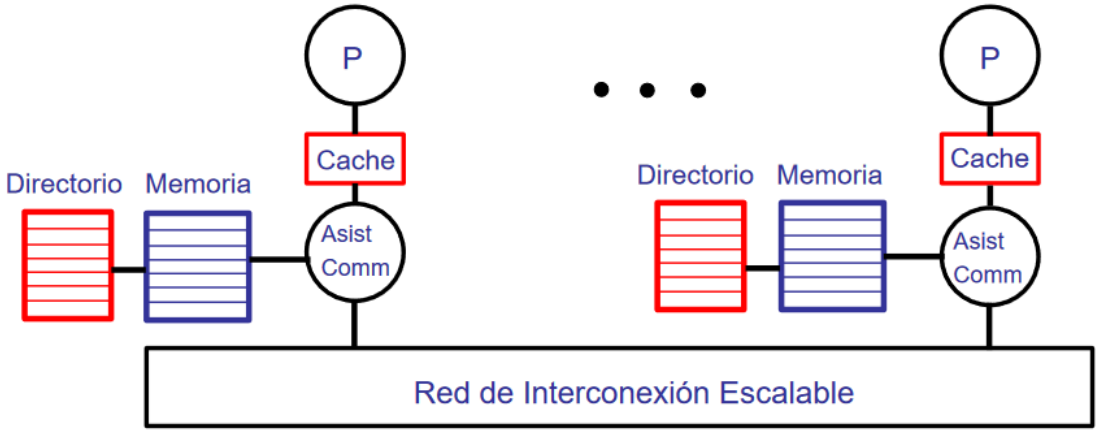
El nuevo estado exclusivo indica que el bloque es la única copia pero no ha sido modificada (bloque limpio) por lo cual no tenemos que notificar nada ya que coincide con la copia de la mem. principal, este nuevo estado reduce el tráfico del bus.

Existe una señal S (Share) por cada bloque que el procesador revisa para saber si se introduce el bloque en caché como compartido o como exclusivo, 0 si ninguna caché contiene el bloque que se solicita (exclusivo) o a 1 en caso contrario (modificado).



Protocolo basado en directorio: En cuanto a los estados son los mismos pero hay una diferencia fundamental y es que no hay mem. centralizada, se aplica fundamentalmente a arq. NUMA, se incluye otro elem. de almacenamiento llamado directorio que está dividido en regiones, cada procesador accede a su región de directorio sin acudir a la red de interconexión a menos que necesite un dato que no esté en su región de directorio.

Al no haber un bus donde observar las transacciones de las demás CPU's se tiene un directorio (elem. de almacenamiento que mantiene el estado de cada uno de los bloques en una región asociada a cada CPU, cada región tiene los datos que residen en su respectiva región de mem). Solo con saber la dirección del bloque ya sabemos en qué región del directorio y de la mem. se aloja, ya que tienen tantas entradas como bloques en la memoria.



Protocolo de coherencia basado en directorio: Es similar al snoopy en el nº de estados y características de los mismos solo que los cambios de estado en este caso vienen dados por los msgs que se transmiten a través de la red.

El directorio me permite determinar para cada bloque de mem:

- Si está copiado o no en la caché de algún procesador.
- Si existe una o varias copias del bloque sin actualizar.
- Quiénes comparten las copias del bloque sin actualizar.
- Si solo un procesador tiene la copia y la mem. no la tiene actualizada (sucio).

En resumen, me dice en qué cachés está el bloque y en qué estado, para plasmarlo en bits para cada bloque de mem. en el directorio se ponen 8 bits que representan si está en la caché del procesador (1) o no (0), más un bit para indicar si está limpio o sucio.

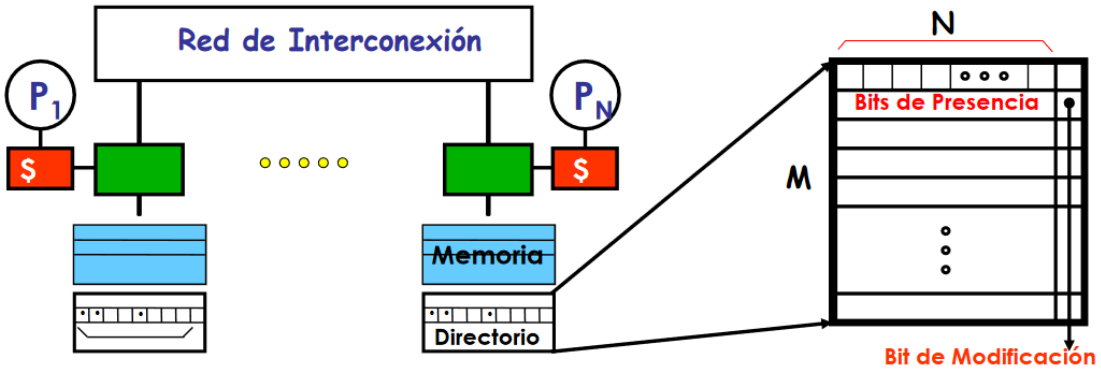
Se usa una nomeclatura especial:

- Nodo peticionario: Aquella CPU que inicia una solicitud pidiendo un dato para leer o escribir.
- Nodo home: Procesador en donde está la zona de mem. donde se almacena el bloque en cuestión.
- Nodo remoto: Procesador cuya caché tiene copias válidas de ese bloque que se está solicitando, en estado compartido o modificado (nodo propietario).

Existen diversos esquemas de directorio:

- Directorio centralizado.
- Directorio distribuido.
  - Jerárquicos.
  - Planos (El origen de la info. de un directorio para un bloque de mem. está en un lugar fijo y conocido a priori que sabemos gracias a la dir. del bloque).
    - Basados en memoria (Info del directorio incluida en el nodo origen).
    - Basados en caché (Info. repartida entre distintos nodos (lista enlazada)).

Protocolos de directorio planos basados en memoria: Se usa en arq. NUMA, en el directorio tenemos un bit de presencia por cada nodo y un bit de modificación, de manera que tenemos tantas entradas en el directorio como bloques de mem. tengamos en la región asociada.



Tenemos un nodo (que contiene un procesador, una cache, un AC, una porción de mem. y porcion de dir.), la nomenclatura usada ahora varía un poco ya que ahora llamamos nodo sucio al que tiene una copia en estado modificado, y nodo exclusivo al que tiene una copia en estado exclusivo.

De esta manera si queremos obtener el bloque para lectura y ese bloque tiene el bit de suciedad a 0 (limpio), el nodo local lo solicita al nodo home, que al estar limpio existe copia del mismo en la caché del nodo home o en mem. principal, por lo que se lo proporciona. En caso de que el bloque este sucio el nodo home le proporcionaría la identidad del propietario de ese bloque, el local se pondría en contacto con él y este le proporcionaría el dato, además habría que actualizar la info. del directorio para poner ese bloque en estado compartido tanto en el nodo local como en el propietario.

Para la escritura (bloque limpio) el local se pondría en contacto con el home que le proporciona la lista de procesadores en cuyas cachés está el bloque, en caso de leer directamente el home se lo proporciona, pero al haberlo pedido para escritura además de darle el dato le tiene que decir qué procesadores tienen copia, para que el local se ponga en contacto con ellos y actualicen su directorio, que pasa de estado compartido a invalidado, quedando en estado modificado solo en la caché del nodo local.

En caso de que el bloque este sucio solo hay una copia válida, por lo que el local se pone en contacto con home que le dice quién es el propietario, contacta con él y transfiere la característica de propiedad al nodo local, actualizando su directorio para indicar que el bloque en la caché del nodo home va a pasar a estado modificado y el anterior propietario a estado inválido.

Rendimiento en las escrituras: Cuántos msgs tengo que mandar cuando se invalida un bloque, el nº es proporcional al nº de nodos que comparten.

Nº de msgs en el camino crítico: Tiempo que tardo en enviar estos msgs, en los basados en mem. existe una ventaja ya que los msgs de invalidación se pueden mandar en paralelo, al tener en el directorio del nodo home info. que nos dice en qué procesadores está y en cuáles no.

Sobrecarga de almacenamiento: El directorio es un elem. extra que añadimos y eso tiene un coste, metemos más info. que antes no estaba, la sobrecarga es el % de tamaño que representa el directorio con respecto a la mem.

$$TamDir = Nº\ Bloques * \ Nº\ nodos\ (bits)$$

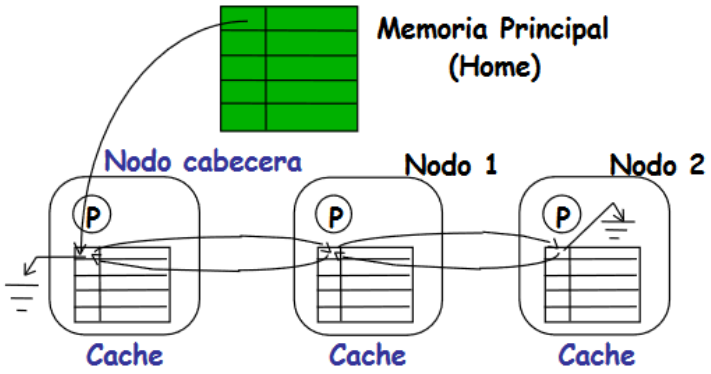
$$TamMem = Nº\ Bloques * (TamBloque * 8)\ (bits)$$

$$Sobrecarga = \frac{TamDir}{TamMem}$$

Con 1024 nodos por ejemplo el tam. del directorio es el doble del tamaño de la mem., esto no es razonable por lo que se han ideado una serie de soluciones:

- Nodos multiprocesador: agrupamos un nº determinado de procesadores y a eso lo llamamos nodo, de manera que solo tenemos 1 bit por nodo.
- Uso de punteros: Otra posible solución consiste es en vez de almacenar 1 bit por nodo, almacenar punteros a los nodos que comparten cada bloque, con esto ahora lo que almacenamos en el directorio en vez de bits de presencia son punteros, de esta manera usamos los bits para decir quiénes son los procesadores que tienen copia, esto reduce la anchura del directorio.
- Organizar el directorio como una caché: Manteniendo el ancho de las columnas reducimos la altura del directorio, de esta manera no tenemos una entrada por cada bloque de mem. sino que organizamos el directorio como una caché almacenando info para unos pocos.

Protocolos de directorio planos basados en caché: La diferencia respecto a los basados en mem. reside en que no está toda la info. en la entrada del directorio del nodo home, lo que hay es un puntero al 1ero de los nodos que tiene copia de dicho bloque, en vez de tener un bit de presencia por cada procesador solo se indica uno de los procesadores que tiene copia (el primero que comparte ese bloque), para saber el resto de nodos que tienen copia se usa una lista doblemente enlazada que indica para cada nodo su anterior y posterior.



En este tipo de directorios cuando tenemos un fallo de lectura el nodo peticionario le envía una petición al nodo home, éste solo nos devuelve el nodo cabecera (si existe), el peticionario envía una solicitud al nodo cabecera diciéndole que le ponga a él como último nodo cabecera, es decir, que le inserte en la lista para reflejar que es el último que ha referenciado ese bloque.

El nodo peticionario sabe cuál es el valor que tiene que almacenar en su caché si el nodo home tiene copia, ya que solo tiene que dársela, si no la tiene, entonces será el nodo cabecera porque es el último que ha referenciado ese bloque.

Tanto insertar como eliminar de la lista es una oper. compleja ya que es necesario ponerse de acuerdo con el nodo anterior y el posterior, además de esto se suma la complejidad de sincronizar y coordinar las operaciones de todos los nodos.

En caso de querer un dato para escribir es necesario recorrer la lista e invalidar las copias de ese dato, ya que se va a modificar, el nº de msgs a enviar es proporcional al nº de copias del bloque existentes, a diferencia de los directorios basados en mem. ahora no tenemos toda la info. necesaria del nº de msgs del camino critico por lo que no se pueden mandar los msgs de invalidación de forma paralela, esto es una desventaja.

En cuanto a las ventajas respecto a los basados en mem. hay una menor sobrecarga del directorio (solamente tenemos info. de punteros) el trabajo realizado por los controladores de caché para mandar msgs de invalidación está distribuido entre todos los nodos que comparten el bloque y por último la lista enlazada guarda el orden en el que se han hecho los accesos, esta info puede ser muy útil.

Def. Sincronización: Mecanismo encargado de garantizar acceso seguro de un proceso a variables compartidas.

El problema de la sincronización está relacionado con la coherencia, ya que pretende garantizar un acceso seguro a una variable compartida, de manera que un procesador pueda modificar esa posición de mem. sin que ningún otro procesador modifique ni toque nada que resida en esa posición de mem, y en caso de que lo haga, saberlo.

Este problema aparece en uniprocesadores con multiprogramación, ya que todos los programas que ejecute tienen acceso a la misma mem. principal, donde pueden existir variables compartidas en las que es necesario sincronizar los accesos.

Def. Primitiva hw: Instr. no interrumpible (atómica) que lee y actualiza una cierta posición de memoria (cerrojo).

Las oper. de sincronización se construyen usando como bloque básico las primitivas hw, mediante rutinas sw (instr. convencionales).

Para los sist. MPP (Massive Paralel Processing) con muchos procesadores las oper. de sincronizaciñon pueden ser un cuello de botella del sist., ya que implican mucho tráfico en la red de interconexión, ya que puede haber un cjto de procesos compitiendo por leer y modificar el cerrojo, el que lo consiga decimos que ha adquirido el cerrojo.

Incluso los procesadores que no consiguen adquirir el cerrojo también escriben sobre la variable compartida, escriben sobre el bit que controla el acceso al cerrojo y meten tráfico en la red de interconexión, por todo esto es necesaria la sincronización.

Existen distintos tipos de primitivas hw:

- Exchange (intercambio): Intercambia el valor de un registro con el valor de una posición de mem, para la posición de mem. referente al cerrojo existe un convenio:
  - Si hay un 0 el cerrojo está libre.
  - Si hay un 1 el cerrojo está adquirido (ocupado) y el derecho a escribir sobre la variable de mem. la ha cogido otro procesador distinto.

  Cada procesador chequea el valor del reg para ver si hay un 0 y pueden intentar competir por adquirir el cerrojo (ver si en mem. hay un 0 también) y hacer el intercambio, o si hay un 1 y el cerrojo está adquirido por otro procesador.

- Test-and-set: Comprueba el valor de una variable y si está a 0 la pone a 1.
- Fetch-and-increment: Lee el valor de una variable y lo incrementa en memoria.

Todas estas primitivas son oper. atómicas que una vez iniciadas no se pueden interrumpir, la sincronización garantiza que si otro procesador se mete en medio de una oper. no va a poder modificar el valor de esas variables.

En cuanto a las rutinas sw existentes:

- Load Linked (LL): Funciona como un load convencional (el dato almacenado en la dir. de mem. se guarda en un reg) pero también se guarda en un reg. especial (link register) la dir. de la cual leemos, si después de hacer el load ocurre algo inesperado (una interrupción o el bloque es invalidado en caché) entonces el link register se borra y se pone todo a 0.
- Store conditional (SC): Funciona como un store convencional (almacena el contenido del reg. en una dir. de memoria) pero si la dir. a la que accede el store coincide con la dir. almacenada en el link register, entonces tiene éxito y devuelve un valor en el reg. que se pone a 1, en caso contrario (fracaso) se pone a 0.
  Es decir, analizando el valor de retorno del store conditional sabemos si se han ejec. como si fueran atómicas (1) o si se han ejec. de manera errónea (0).

Def. Spin locks: Para tratar de acceder a una variable compartida los procesadores entran en un bucle en el cual intentan constantemente conseguir acceso al cerrojo, cuando uno lo consigue tiene permiso para acceder a la sección crítica de la variable y modificarla, todos los demás deberán esperar a que se libere mientras generan tráfico.

No llevamos la variable a caché por lo que no existe el problema de la coherencia, pero si se genera tráfico, esto puede convertirse en un cuello de botella para el procesador.

En un sist. con coherencia caché además de la problemática ya existente los procesos también intentarían llevar la variable cerrojo a su caché por lo que sería incluso peor.

Esta situación se mejora haciendo antes de la primitiva exchange un testeo (lectura) del valor de la variable, así solo si vemos un 0 entonces tratamos de hacer el exchange.

Notas:

- Memoria segmentada: Cada nuevo ciclo de reloj se puede iniciar el procesamiento de una nueva instr. de memoria.

DAVID ZAMORA REY