AEC

Jose Antonio Lorencio Abril

1 Análisis de Prestaciones en Arquitectura de computadores

1.1 Definición de rendimiento

Para el usuario lo importante es el tiempo de respuesta, y para los centros de cálculo es la productividad. Pero, en general, nn ordenador es más rápido que otro cuando realiza la misma cantidad de trabajo en menos tiempo, o sea, si:

- Disminuye el tiempo de respuesta, ya sea por tener un ciclo de reloj más rápido, por procesar en paralelo,...
- Aumenta la productividad, por ejemplo al poner muchos ordenadores trabajando en paralelo.

Ambos aspectos están relacionados, un aumento en la productividad puede suponer una reducción del tiempo de respuesta.

1.1.1 Comparando velocidades

X es n% más rápido que Y si

$$\frac{Rendimiento_X}{Rendimiento_Y} = 1 + \frac{n}{100}$$

Si definimos el rendimiento como la inversa del tiempo de ejecución, quedaría:

$$\frac{Rendimiento_X}{Rendimiento_Y} = \frac{\frac{1}{TEj_X}}{\frac{1}{TEj_Y}} = \frac{TEj_Y}{TEj_X} = 1 + \frac{n}{100}$$

1.2 Métricas populares

Dependiendo del nivel en que nos situemos, las unidades de medición serán distintas:

- A nivel de aplicación, suele medirse en **tiempo de ejecución**
- A nivel de ISA (Instruction Set Architecture) suelen usarse medidas como los **MIPS** o los **MFLOPS**
- A nivel de hardware lo mediremos como **ciclos por segundo, MB/s**,...

1.2.1 Tiempo de CPU

$$T_{CPU} = \frac{segundos}{programa} = \frac{instr}{programa} \cdot \frac{ciclos}{instr} \cdot \frac{segundos}{ciclo} = NI \cdot CPI \cdot T_{ciclo}$$

Cuando el CPI no es uniforme, lo calculamos como una media ponderada

$$CPI = \sum_{i=1}^{n} CPI_i f_i$$

Donde f_i es la frecuencia de aparición de la instrucción i.

1.2.2 MIPS

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{F(en\,MHz)}{CPI}$$

Presenta como ventaja que es fácil de comprender, a mayor MIPS, más velocidad.

Sin embargo, son dependientes del conjunto de instrucciones y pueden variar entre programas en el mismo ordenador. Esto hace que incluso pueda variar inversamente al rendimiento, pudiendo llevarnos a conclusiones erróneas.

1.2.3 MFLOPS

$$MFLOPS = \frac{NI\,en\,PF}{T_{CPU} \cdot 10^6}$$

Tiene como ventaja que la parte en PF depende del algoritmo, no de la máquina.

Pero no se pueden aplicar a todos los programas (por el compilador), todas las máquinas no presentan las mismas instrucciones PF, lo que dificulta la comparación. Además, no todas las instrucciones en PF tardan lo mismo. Aunque esto puede solucionarse normalizándolas, por ejemplo, en base a una operación suma.

1.3 Ley de Amdahl

La ley de Amdahl es usada para medir la aceleración debida a una mejora E, de una fracción f del tiempo de ejecución, en un factor S, sin afectar al resto. Establece que:

$$speedup = \frac{T}{T_E} = \frac{T}{(1-f) \cdot T + \frac{f}{S} \cdot T} = \frac{1}{(1-f) + \frac{f}{S}}$$

La ganancia producida por una mejora está limitada por la fracción de tiempo que puede usarse esa mejora.

Esta ley nos ayuda a tomar decisiones de diseño.

Por otro lado, no debemos caer en el error de esperar que la mejora de un único aspecto de una máquina provoque un aumento del rendimiento proporcional a la misma.

1.3.1 Corolario

Si hacemos rápido el caso frecuente, podremos mejorar en gran medida el rendimiento. Además, normalmente el caso común suele ser el más sencillo de mejorar.

1.4 Cómo comparar resultados

No es nada fácil, pues pueden hacerse de diversas formas, arrojando resultados distintos.

1.4.1 Tiempo total de ejecución

Es una medida resumen consistente. Es equivalente a hacer la media aritmética de los tiempos de ejecución

$$\frac{1}{n}\sum_{i=1}^{n} T_i$$

Si tenemos frecuencias en lugar de tiempos, usaremos la media armónica

$$\frac{n}{\sum_{i=1}^{m} \frac{1}{Vel_i}}$$

1.4.2 Tiempo total de ejecución ponderado

No todos los programas se ejecutan el mismo número de veces.

Puede usarse la media aritmética ponderada

$$\sum_{i=1}^{n} w_i T_i$$

O la media armónica ponderada

$$\frac{1}{\sum_{i=1}^{n} \frac{w_i}{Vel_i}}$$

1.5 Tiempo normalizado

Si hay mezcla desigual de programas, se tiende a normalizar los tiempos de ejecución para una máquina de referencia y después tomar la media geométrica de los tiempos normalizados

$$\sqrt[n]{\prod_{i=1}^{n} TNorm_i}$$

Presenta la ventaja de que es consistente independientemente de la máquina tomada como referencia (al contrario que la media aritmética)

Aunque debemos ser cautos, la media geomrica no predice los tiempos de ejecución, solo sirve para hacer comparaciones ordinales.

1.6 Programas de prueba (Benchmarks)

Son programas usados para medir el rendimiento. Por orden de fiabilidad:

- Programas reales (a veces simplificados)
- Núcleos o kernels: partes de programas reales
- Benchmarks reducidos (toys): pequeños y fácilmente portables
- Benchmarks sintéticos: parecidos a los núcleos, su objetivo es obtener un perfil medio de ejecución.

En ocasiones pueden llevar a conclusiones incorrectas.

Es importante que las pruebas sean reproducibles (entradas al programa, versión, nivel de optimización,...

1.6.1 SPEC CPU (System Performance Eval. Coop)

Conjunto de programas de prueba fundamentalmente centrados en el rendimiento de la CPU. Han ido evolucionando con el tiempo hasta hoy, que son 43 programas organizados en 4 suites.

1.6.2 TPC (Transaction Processing Council)

Miden el rendimiento en transacciones por segundo, incluyendo un requisito de tiempo de respuesta.

Tiene en cuenta CPU, I/O, red, SO,...

A día de hoy existen 4 tipos para simular distintos entornos de trabajo (manejo de pedidos, toma de decisiones, comercio electrónico y servidores web activos 24/7).

2 Segmentación Básica

2.1 Introducción

La **segmentación** es una técnica que consiste en solapar la ejecución de las instrucciones. El cauce de ejecución estará dividido en varias etapas o segmentos.

Esta técnica aprovecha el paralelismo existente entre las acciones necesarias para ejecutar las instrucciones

Hoy en día es una técnica fundamental para conseguir CPUs más rápidas.

Las etapas se conectan una a la sigueinte formando el cauce o **pipeline**.

Definimos la **productividad** como la frecuencia de salida de instrucciones.

Las instrucciones entran por un extremo del pipeline, pasan por las distintas etapas y salen por el otro extremo.

Como las etapas del pipeline están conectadas enc ascada, todas deben de estar listas para avanzar en el mismo instante. A este tiempo se le llama **ciclo máquina** y coincidirá con el tiempo necesario para hacer la etapa más lenta.

De esta forma, el tiempo de instrucción en el procesador segmentado, en el caso ideal, será:

$$\frac{T\,Instr\,NoSegm}{N^{\underline{o}}\text{ etapas}}$$

Así, en el caso ideal se obtiene una aceleración igual al número de etapas del cauce, aunque normalmente estas etapas noo estarán perfectamente balanceadas, además, la segmentación suele tener un coste, por lo que el tiempo de instrucción no suele ser el ideal, aunque se aproxima.

La segmentación consigue aprovecharse del paralelismo existente entre las instrucciones y no es visible al programador.

Depende de como se mire, la segmentación consigue:

- Disminuir el CPI
- Disminuir el T_{ciclo}.

2.1.1 La arquitectura DLX

Es una arquitectura RISC de carga-almacenamiento parecida a MIPS. Tiene 32 registros de propósito general de 32 bits y otros 32 de punto flotante que pueden usarse como 32 registros de simple precisión o 16 de doble.

También tiene los siguientes registros especiales:

- Registro de estado de PF (fp)
- Contador de programa (PC)
- De direcciones de interrupción (IAR)

Los tipos de datos pueden ser bytes, medias palabras (16 bits), palabras (32 bits) y los datos de PF de precisión simple o doble.

Posee una memoria direccionable por bytes con una dirección de 32 bits.

Como modos de direccionamiento tiene registros, inmediatos, base+desplazamiento relativo al PC y pseudodirecto.

Solo tiene tres formatos de instrucciones:

- Instrucciones con inmediato
- Instrucciones entre registros
- Instrucciones de salto

Y su repertorio de instrucciones contiene instrucciones de transferencia de datos, aritmético-lógicas, de control y de punto flotante.

2.1.2 Implementación de DLX sin segmentación

Vamos a implementar un subconjunto de DLX que consta de las siguientes instrucciones: carga y almacenamiento de palabras, saltos y operaciones enteras de ALU.

Cada instrucción puede ejecutar **5 etapas:**

1. Búsqueda de instrucción (IF)
2. Decodificación de la instrucción/Leer registros (ID)
3. Ejecución / Dirección efectiva (EX)
4. Acceso a memoria / Finalización de saltos (MEM)
5. Postescritura (WB)

Veamoslas una a una:

1) Instruction Fetch (IF)

Busca la siguiente instrucción en la memoria y la trae al registro de instrucción (IR).

Incrementa el PC en 4 y se almacena en el registro NPC.

2) Decodificación de la Instrucción / Leer registros (ID)

Decodifica la instrucción y accede al banco de registros para leer dos registros.

La salida de los registros se almacena en dos registros intermedios (A y B) para usarla en los ciclos posteriores.

Se extiene el signo a los 16 bits menos signifativos de la instrucción y se almacena en un registro intermedio (Imm).

3) Ejecución / Dirección efectiva (EX)

Dependiendo del tipo de instrucción, la ALU realiza lo siguiente:

A) Referencia a Memoria: Suma los operandos para formar la dirección efectiva. El resultado se deja en ALU_{output}.

B) Instrucción ALU Registro-Registro: realiza la operación especificada por el código de función con los operandos situados en los registros A y B. El resultado se guarda en ALU_{output}.

C) Instrucción ALU Registro-Inmediato: realiza la operación especificada por el código de operación con los operandos situados en los registros A e Imm. El resultado se guarda en ALU_{output}.

D) Salto: calcula la dirección efectiva del salto sumando el registro NPC al valor del inmediato con signo que tenemos en Imm. Se chequea el registro A, leído en la etapa anterior, para determinar si el salto es tomado o no. La operación de comparación op es el operador relacional especificado en la instrucción de salto.

4) Acceso a Memoria / Finalización de Saltos (MEM)

Las únicas instrucciones DLC activas en esta etapa son:

A) Referencia a Memoria: puede ser una carga o un almacenamiento. En una carga el dato extraído de memoria se almacena en el registro LMD. En un almacenamiento el dato que está en el registro B se escribe en memoria. En ambos casos, la dirección a usar es la calculada en la etapa anterior y almcenada en ALUOutput.

B) Saltos: si la condición es tal que debe tomarse el salto, el PC se reemplaza por la dirección de destino del salto. Si no debe tomarse el PC se reemplaza con el PC incrementado que está almacenado en el registro NPC.

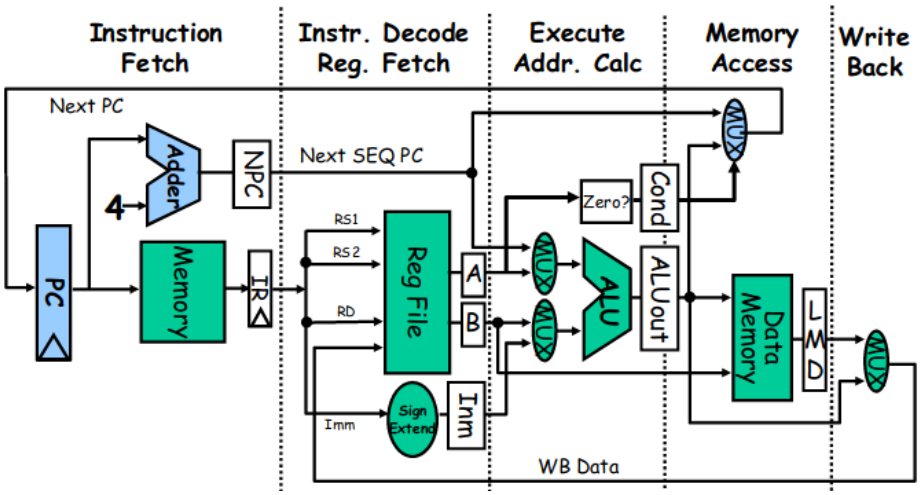
5) Write-Back (WB)

Escribe el resultado en el banco de registros. Para una instrucción de carga, está en LMD y para una instrucción ALU, está en ALUOutput.

El campo que indica el registro destino también puede estar en dos posiciones distintas dependiendo del código de función indicado.

- Instrucción ALU Registro-Registro: se encuentra en los bits 16..20
- Instrucción ALU Registro-Inmediato: se encuentra en los bits 11..15
- Instrucción de carga: también en 11..15

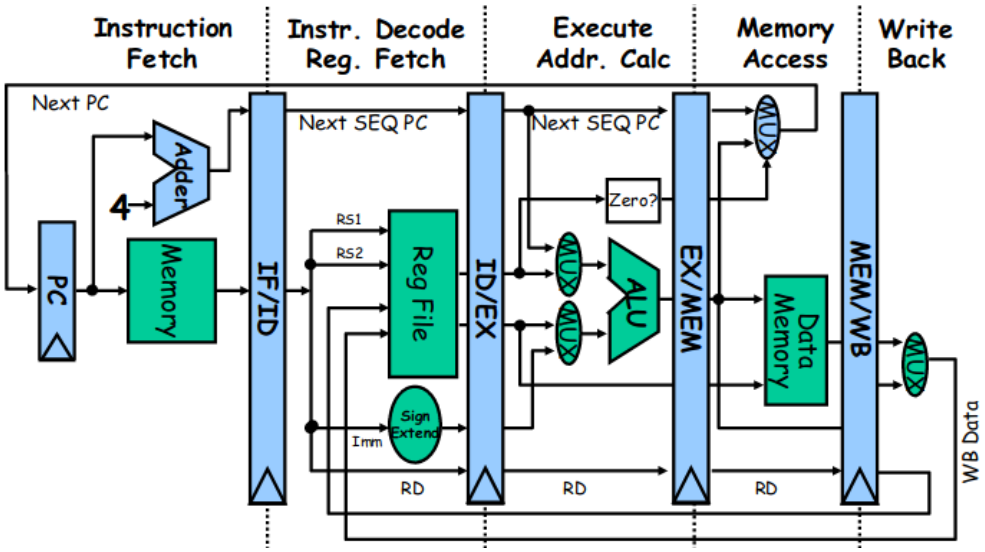
Al final de cada ciclo de reloj, lo calculado en una etapa se almacena en registros ingermedios. Se necesitan 4 ciclos para saltos y almacenamientos, 5 para el resto.



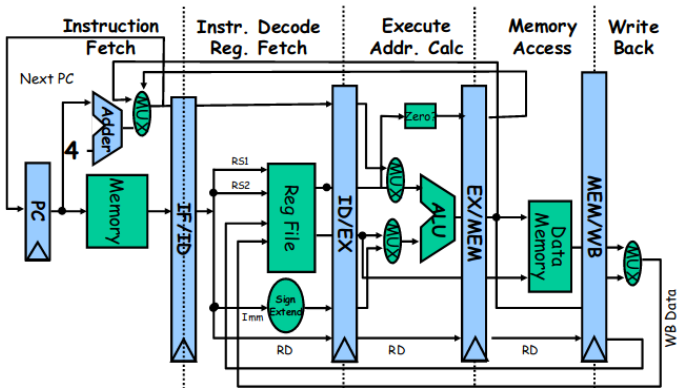
2.2 Segmentación para la ejecución de instrucciones

Podemos segmentar el pipeline de ejecución de instrucciones de DLX casi sin cambios de forma que comience una nueva instrucción en cada ciclo de reloj.

Solo tenemos que añadir los **registros de la segmentación**, que aíslan las distintas etapas, permitiendo que estas trabajen en paralelo.



También debemos modificar el cauce para que se calcule la dirección de la siguiente instrucción en la etapa IF.



Como cada etapa está activa en cada ciclo, cada una debe completar sus acciones en dicho ciclo.

Los registros de la segmentación o pipeline registers aíslan las etapas del cauce. Se llaman y se etiquetan con el nombre de las etapas que separan. Se usan para pasar datos y el control de una etapa a la siguiente, cualquier valor que pueda ser necesario en una etapa posterior debe propagarse a través de estos registros, hasta que deje de ser necesario.

Una instrucción está activa en una única etapa en cada ciclo: las acciones se realizan entre dos registros de segmentación. De esta forma cada instrucción seguiría tardando cinco ciclos, pero ahora se estarían haciendo cinco instrucciones al mismo tiempo, estando cada una de ellas en una etapa distinta.

Hay que tener en cuenta que dos instrucciones distintas no pueden hacer uso de ningún recurso común al mismo tiempo, por lo que se debe asegurar que la segmentación no cause ese conflicto.

Se usan **memorias caché separadas** para instrucciones y para datos, con lo que se puede acceder a la vez a ambas.

Si el ciclo de reloj es el mismo que el de la máquina sin segmentar, el **ancho de banda** del sistema de memoria debe ser cinco veces mayor.

El **banco de registros** se usa tanto en la etapa ID para la lectura de los dos registros, como en WB para la escritura del registro resultado.

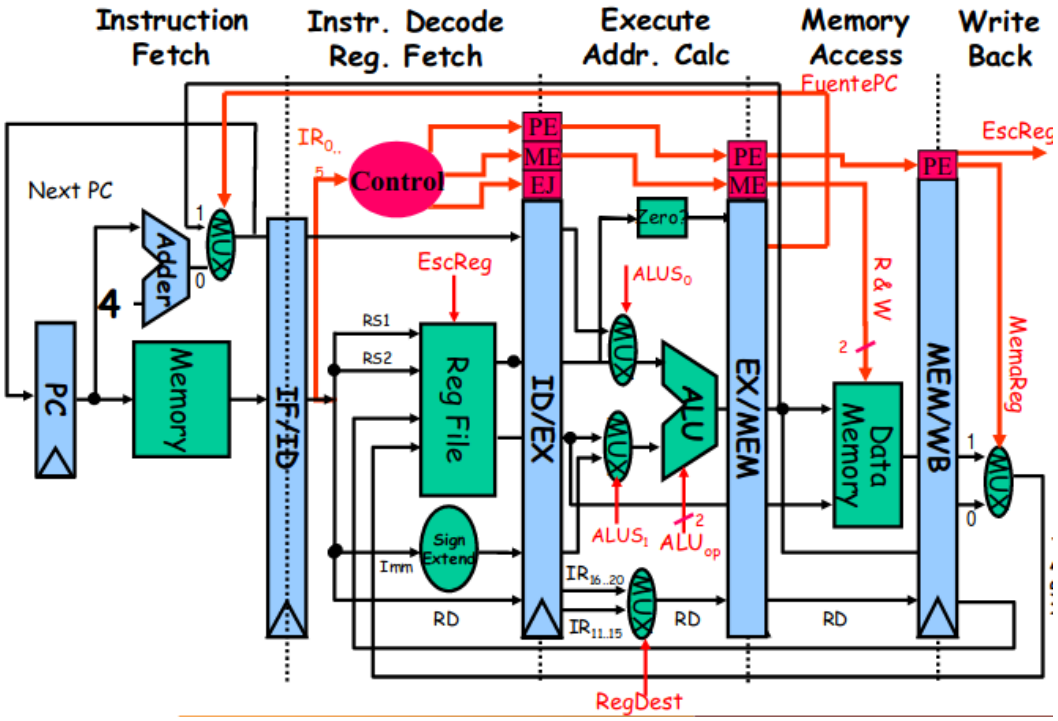
Para poder empezar una nueva instrucción cada ciclo de reloj debemos incrementar el PC.

La segmentación **incrementa la productividad** de instrucciones de la CPU, pero no reduce el tiempo de ejecución de cada una de las instrucciones. De hecho, suele incrementarlo ligeramente debido a la sobrecarga que supone.

El incremento de la productividad de instrucciones significa que un programa se ejecuta más rápido y tiene un tiempo de ejecución total menor, aunque sus instrucciones tarden más en ejecutarse.

El hecho de que el tiempo de ejecución de cada instrucción no disminuya, pone una cota a la profundidad de la segmentación.

Las **señales de control** que la segmentación produce son las siguientes:



2.3 Problemas de la segmentación: los riesgos

Hay situaciones, llamadas riesgos o hazards, en las que una instrucción no puede avanzar a la siguiente etapa del cauce. Estos riesgos provocan degradación del rendimiento y hay de tres tipos: **estructurales** (causados por un conflicto en el uso de un recurso hw si hay combinaciones de instrucciones no soportadas), **de datos** (cuando una instrucción depende del resultado de otra previa, y la ejecución simultánea de ambas instrucciones puede provocar problemas) y **de control** (provocados por la segmentación de los saltos y otras instrucciones que modifican el PC).

Los riesgos pueden ser resueltos **a nivel hw** (detectando la situación y parando la instrucción problemática y las siguientes mientras avanzan las emitidas anteriormente hasta su finalización) o **a nivel sw** (añadiendo en el ISA una instrucción NOP que será empleada por el compilador para evitar los riesgos).

2.3.1 Riesgos estructurales

Surgen cuando dos o más instrucciones necesitan usar el mismo recurso hw para cosas distintas. Esta situación ocurre cuando un recurso no se ha replicado lo suficiente o no se ha segmentado para permitir la combinación de algunas instrucciones.

Posibles soluciones:

- Dedicar **más hw** al problema. Aunque podría no ser posible por el incremento en el costo que supondría, incluso hay casos en los que no merece la pena.
- **Detener el cauce** durante un ciclo de reloj: debemos detectar el riesgo y tener un mecanismo de detención del cauce.
- Que **sea el sw** el que evite el riesgo.

El conjunto de instrucciones debe permitir detectar los riesgos estructurales de forma sencilla. Debe ser sencillo saber los recursos usados por una instrucción, pues el código de operación lo dice todo, y debe ser uniforme en la utilización de los recursos.

Si el resto de factores son iguales, una máquina sin riesgos estructurales siempre tendrá un CPI menor:

$$CPI_{riesgos} = CPI_{ideal} + Ciclos\,de\,detención\,por\,instrucción$$

2.3.2 Riesgos de datos

Dependencia: pueden ser:

- **de datos:** existe un flujo de información entre una instrucción productora de un dato y otra instrucción que consume dicho dato.

La instrucción j depende de la instrucción i, o j depende de k y k depende de i. También se llaman **dependencias verdaderas**.

```
LD  F0,0(R1)
ADDD  F4,F0,F2
```

Vemos como necesitamos el valor de F0 para realizar la segunda instrucción. No podremos ejecutarla hasta tenerlo.

- **de nombre:** no hay flujo real de información entre las instrucciones.

Antidependencia: la instrucción j escribe registro o posición de memoria que i lee (i<j).

```
SD  0(R1),F4
ADDD  F4,F0,F2
```

Si se ejecutara la segunda instrucción antes que la primera, estaríamos guardando un valor erróneo en 0(R1).

Dependencia de salida: las instrucciones i y j escriben en el mismo registro o posición de memoria.

Ambos tipos de dependencias pueden solucionarse renombrando registros.

- **de control:** debidas a instrucciones de control o saltos. Determinan el flujo de ejecución de las instrucciones y el grado de reordenación que se puede realizar en el programa. Una instrucción dependiente de un salto (posterior a él) no puede moverse a una posición anterior al salto. Una instrucción no dependiente de un salto (anterior) no puede moverse a una posición posterior al salto.

Riesgos de datos

Los riesgos de datos ocurren cuando la segmentación cambia el orden de los accesos de lectura/escritura a los operandos que se daría en la versión secuencial sin segmentación.

Se produce un riesgo si hay una dependencia entre instrucciones, y estas están suficientemente cerca para que la segmentación cambie el orden de acceso a los operandos.

Puede darse en el acceso a los registros, pero también en el acceso a una posición de memoria.

En DLX, al tener solo un canal de acceso a memoria y caché bloqueante, los accesos se hacen en orden. Dependiendo del orden entre lecturas y escrituras, podemos clasificar los riesgos de datos:

- **RAW (Read After Write):** una instrucción posterior quiere leer un operando antes de que lo escriba una instrucción anterior.
- **WAR (Write After Read):** una instrucción posterior trata de escribir su resultado antes de que una instrucción anterior haya leído el registro o posición de memoria que pretende escribir. Es provocado por una antidependencia, que el compilador resuelve mediante el renombramiento de registros. **NO** puede ocurrir en DLX pues todas las lecturas se hacen en ID y todas las escrituras en WB.
- **WAW (Write After Write):** una escritura posterior se produce antes que otra escritura anterior en el mismo destino. Es producido por una dependencia de salida. Solo es posble si varias etapas del cauce pueden volcar sus resultados o se permite que otra instrucción continúe mientras una anterior está detenida. Así, en DLX no pueden darse estos riesgos.

Solución HW: detectándolo e insertando ciclos de parada. Habría que añadir hw para detectar el riesgo de datos y provocar una detención. La instrucción que debe producir el dato continuará, mientras que la que necesita el dato y las siguientes esperarán hasta que este esté disponible. El CPI de la instrución detenida aumenta en tantos ciclos ocmo dure la detención.

Se añade una **unidad de detección de riesgos**. Los riesgos por dependencias de datos se analizan en la etapa ID y si aparecen la instrucción no es emitida, siendo parada hasta que el riesgo desaparezca.

Solución SW: mediante técnicas de compilación. El compilador podría evitar el riesgo insertando instrucciones NOP entre la instrucción que produce el dato y la que lo consume, pero esto incrementa el número de instrucciones del programa.

2.3.3 Riesgos de control

Cuando se ejecuta una instrucción de salto, dependiendo de si se toma el salto o no, se cambiará el PC a PC+2 o a otra dirección distinta.

En nuestro DLX, este cambio del PC no se lleva a cabo hasta el final de la etapa MEM cuando se haya completado el cálculo de la dirección y la comparación.

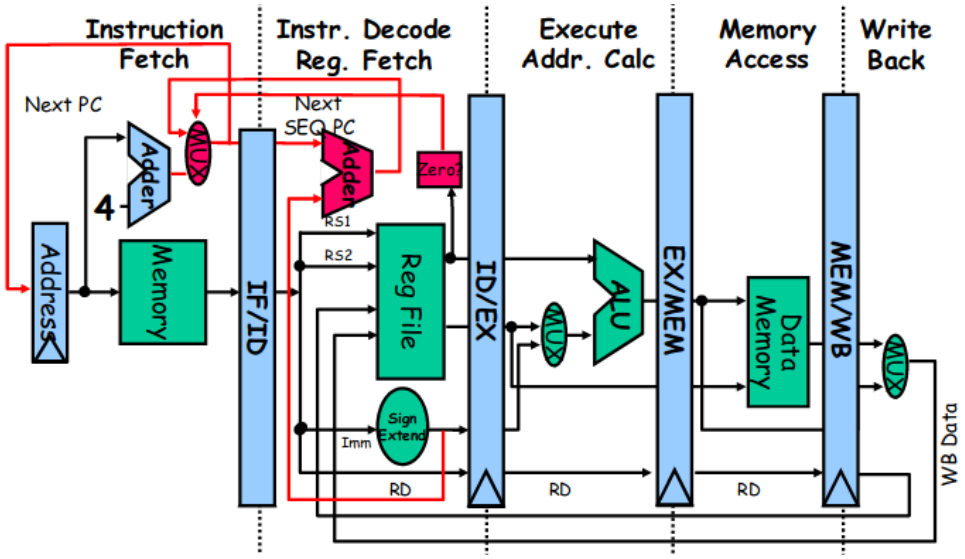
La **solución HW** más básica consiste en ampliar la unidad de detección de reisgos para que cuando se detecte un salto se detenga el cauce hasta saber si el ciclo es tomado o no. Como el salto se resuelve en MEM será necesario introducir 3 ciclos de parada para darle tiempo al salto a que llegue a dicha etapa. En cauces más profundos, en los que los saltos pasarían por más etapas hasta resolverse, habría más ciclos de parada y se degradaría de forma significativa el rendimiento.

Los riesgos de control pueden causar una mayor pérdida de rendimiento que los riesgos de datos.

La latencia de un salto puede reducirse en dos pasos: averiguando si el salto se va a tomar o no en una etapa anterior o calculando antes la dirección efectiva del salto.

Mejora para DLX: mover la unidad detectora de ceros a ID. Calcular la dirección de salto, sea cual sea el resultado de la comparación en ID. Esto obliga a añadir un sumador, ya que la ALU no está disponible en esta etapa. Con este hw añadido podemos tomar la decisión del salto en la etapa ID, con lo que solo tendríamos un ciclo de parada para los saltos.

El cauce quedaría:



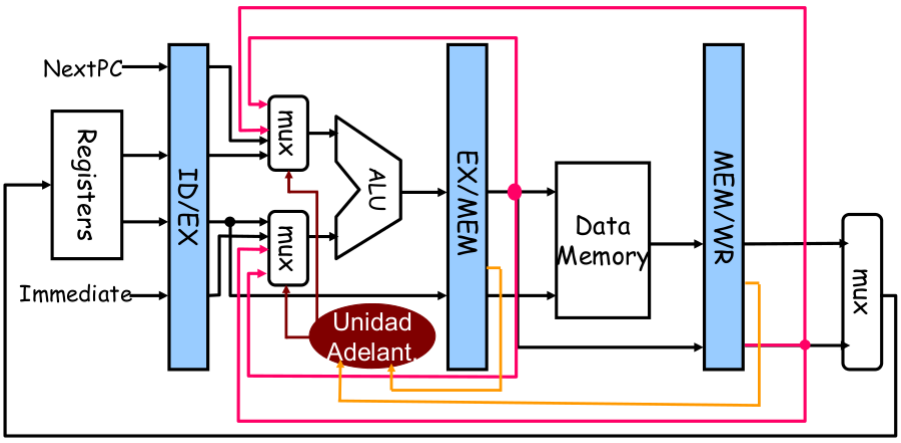
3 Segmentación avanzada y predicción de saltos

3.1 Mitigación de los riesgos de datos

En las soluciones que vimos en el tema 2 para resolver los riesgos de datos se producía una degradación del rendimiento debido a ciclos de parada o la aparición de instrucciones adicionales. Ahora vamos a ver una técnica HW que resuelve los riesgos de datos y lo hace afectando lo menos posible al rendimiento.

El **adelantamiento** consiste en que el trasvase de información entre dos instrucciones dependientes no se haga a través del banco de registros, sino desde la unidad funcional productora del dato hasta la UF consumidora del mismo. Por supuesto, esta técnica minimiza el impacto de los riesgos **RAW**.

Esta técnica puede generalizarse para que cualquier UF pueda obtener un operando directamente del registro de segmentación en el que pudiera encontrarse. Pero, evidentemente, para hacer esto necesitamos circuitería adicional.



En esta figura vemos cómo se implementaría la circuitería que permite hacer adelantamientos a la ALU. Nótese, además, que hemos añadido una **unidad de adelantamiento**, que sirve para manejar las nuevas situaciones de control que plantea nuestro diseño nuevo.

Esta unidad de adelantamiento hará lo siguiente:

Adelantamiento EX:

```
if (EX/MEM.RegWrite && EX/MEM.Rd<>0 && EX/MEM.Rd=ID/EX.Rs)
        anticiparA=10

if (EX/MEM.RegWrite && EX/MEM.Rd<>0 && EX/MEM.Rd=ID/EX.Rt)
        anticiparB=10
```

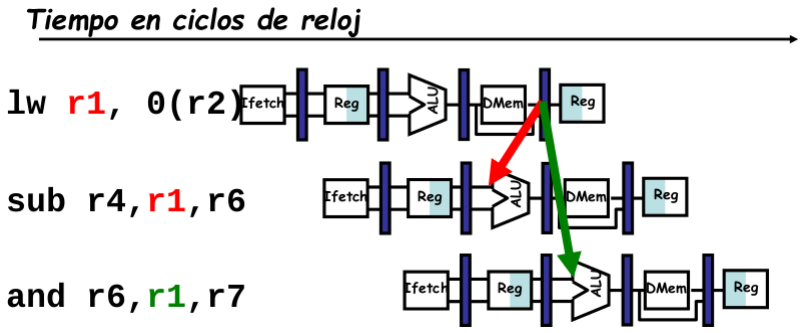
Adelantamiento MEM:

```
if (MEM/WB.RegWrite && MEM/WB.Rd<>0 && MEM/WB.Rd=ID/EX.Rs)
        anticiparA=01

if (MEM/WB.RegWrite && MEM/WB.Rd<>0 && MEM/WB.Rd=ID/EX.Rt)
        anticiparB=01
```

Primero se haría el test de adelantamiento en EX, y si este resulta negativo, se hace el de MEM.

Aunque el adelantamiento es una técnica que mejora enormemente el funcionamiento de DLX, hay algunos riesgos de datos que aún pueden dar problemas. Las instrucciones de carga tienen un retraso que no se puede eliminar totalmente con los adelantamientos, aunque se quedan reducidos a 1 único ciclo de parada.



3.2 Mitigación de los riesgos de control

Para mejorar el rendimiento y mitigar la penalización de los saltos, los procesadores modernos recurren a técnicas de **predicción de saltos**. Estas pueden ser **estáticas** (no miran el comportamiento del salto) o **dinámicas** (usan HW para predecir el resultado de un salto gracias a que los saltos siguen patrones predecibles).

3.2.1 Predicción de saltos estática

- **Predict Not-Taken**

Se tratan todos los saltos como si no se fueran a tomar hasta que se resuelvan, procediendo con las instrucciones del camino NT.

Si el salto **es tomado** se han de descartar todas las instrucciones ejecutadas especulativamente.

En DLX se implementa fácilmente, se siguen emitiendo instrucciones del camino NT y si el salto resulta tomado, se convierten a NOP las instrucciones buscadas del camino NT.

- **Predict Taken**

Se tratan todos los saltos como si se fueran a tomar. En cuanto se tenga la dirección del salto se sigue por ahí.

Si el salto es **no tomado** se ha de deshacer lo hecho.

Nótese que en DLX no tiene sentido esta alternativa, pues sabemos la dirección del salto a la vez que la condición de salto.

3.2.2 Predicción de saltos dinámica

Las técnicas dinámicas de predicción pretenden predecir el resultado de un salto condicional en tiempo de ejecución. Obtienen muy buenas tasas de acierto gracias a que los saltos siguen patrones predecibles de comportamiento, generalmente basados en su historia.

- **Buffer de predicción de saltos**

Es el esquema más simple. Se conoce como **Branch Prediction Buffer (BPB)**, y no es más que una pequeña memoria indexada por la parte baja del PC de la instrucción del salto ($PC \mod 2^m$), donde cada entrada contiene un contador saturado de n bits.

BPB de 1 bit

Guarda 1 bit d historia que dice si el salto fue tomado o no la última vez. Si la predicción falla se invierte el bit para la próxima vez.

Presenta como **problemas** que falla 2 veces en cada bucle, en lugar de una vez, y que se producen interferencias entre saltos que dan el mismo $PC \mod 2^m$. Las **soluciones** pasan por utilizar más de un bit en la predicción y tener más entradaas en la tabla de contadores.

BPB de 2 bits

Usa contadores saturados de 2 bits para cada salto. Ahora actualizamos el contador a cada salto: se incrementa si el salto es tomado (y no está al máximo, 11) y se decrementa si es no tomado (y no está en 00). Para hacer la predicción se toma el bit más significativo (izq), si es 0 se predice NT y si es 1 se predice T.

BPB generalizado de n bits

Ahora se asocia la mitad baja de los valores a NT y la mitad alta a T. Así que, queda como antes: si el bit más significativo es 1, el salto se predice T, si es 0 como NT. Cuando el salto se toma, se aumenta en una unidad el contador, y cuando es NT se decrementa.

- **Predictores con correlación**

Ahora buscamos usar información de otros saltos además de la historia del salto a predecir.

Predictor con correlación con 2 tablas de contadores de 1 bit de historia y 1 bit de correlación

Ahora, para cada salto, tendremos dos bits de historia para elegir. Uno se usa si el último salto no se tomó, y el otro si sí se tomó. La elección del bit de historia utilizado se basa en el bit de correlación global.

Si un salto es tomado, el bit de correlación se pone a 1, se usará el bit de historia correspondiente a último salto tomado.

Si es no tomado, el bit de correlación se pone a 0 y se usará el otro bit de historia.

Predictores de correlación con 2^m tablas de contadores de n bits de historia y m bits de correlación

Se puede generalizar a un predictor (m,n) donde se usa el comportamiento de los últimos m saltos para seleccionar una de las 2^m tablas de contadores saturados de n bits.

Así, un predictor (m,n) ocupa:

$$2^m \cdot n \cdot NumEntradas\ bits$$

La historia global de los últimos m bits se almacenan en un registro de desplazamiento de m bits.

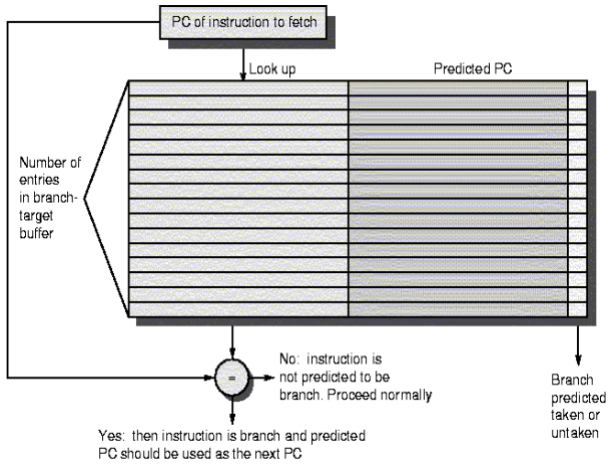
- **Buffer de destino de salto (BTB)**

Para reducir la penalización de los saltos en nuestro DLX a solo 0 ciclos hemos de saber en la etapa IF si la instrucción que acaba de entrar es un salto; de ser así, si es tomado o no; y, si es tomado, a qué dirección se dirige.

Para eso se crea el **Buffer de Destino de Saltos (Branch Target Buffer, BTB)**, que consiste en una caché que proporciona la dirección de la siguiente instrucción que sigue a un salto. Se accede en la etapa IF usando el PC actual, y si hay acierto de BTB suponemos que se trata de un salto y el BTB proporciona el siguiente PC, que puede ser el de la siguiente instrucción secuencial (PC+4) o el PC destino del salto.

A diferencia de los predictores anteriores, el BTB usa etiquetas (parte del PC) para evitar interferencias.

De esta forma, si el PC se encuentra en las etiquetas, esto supone un acierto de TLB, lo que quiere decir que la instrucción es un salto. El segundo campo del BTB proporciona el PC siguiente. Además, en un tercer campo opcional, se podría tener información de predicción.



- **BTB que solo almacena saltos tomados**

Si el PC está en el BTB, predecimos que la instrucción será un salto T. Si en el siguiente ciclo vemos que no es así, lo quitamos del BTB. Si no lo encontramos, usamos el PC secuencial (o sea, como si fuera Predict NT) y si resulta que es un salto tomado, actualizamos el BTB para futuras ocasiones.

No hay penalización cuando la instrucción está en el BTB y la predicción es correcta (salto T) ni cuando la instrucción no está en el BTB y el salto es NT.

Sí hay penalización cuando la instrucción está en el BTB, pero la predicción es incorrecta y cuando la instrucción no está en el BTB, pero el salto es T.

3.3 Excepciones

Una **excepción** es un cambio inesperado en el flujo de control proveniente de una causa interna o externa. Si la causa es externa se puede llamar también interrupción. Estas situaciones son más difíciles de manejar en las máquinas segmentadas debido a la superposición de instrucciones, ya que esto hace más difícil saber cuándo una instrucción puede cambiar de forma segura el estado de la máquina.

Las excepciones suelen suceder en medio de la ejecución de una isntrucción y deben ser capaces de guardar el estado, atender la excepción, restaurar el estado y recomenzar el programa original.

En DLC pueden presentarse los siguientes problemas:

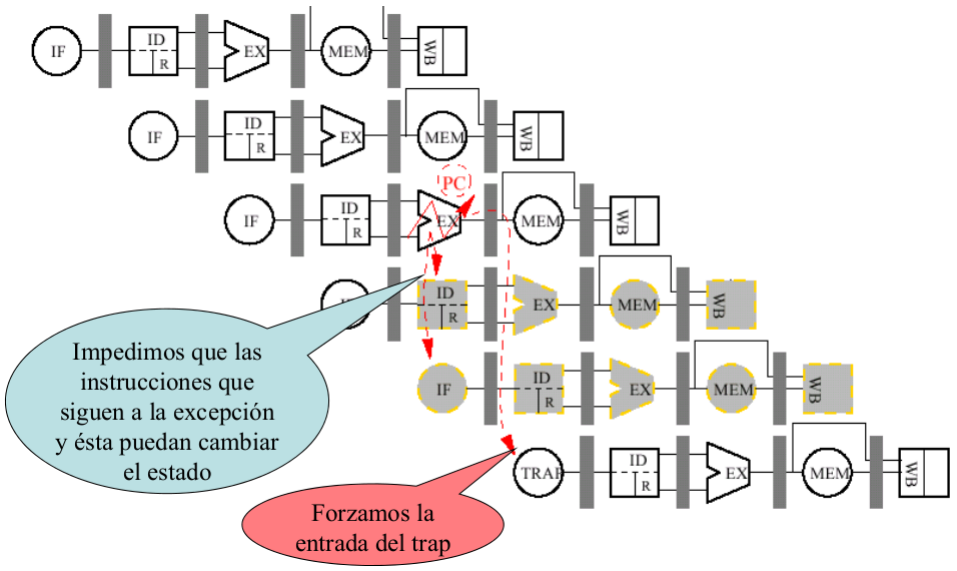
Etapa del Cauce	Posibles excepciones
IF	Fallo de página en la búsqueda de la instrucción Acceso a memoria mal alineado Violación de la protección de memoria
ID	Código de operación indefinido o ilegal
EX	Excepción aritmética
MEM	Fallo de página en la búsqueda de datos Acceso a memoria mal alineado Violación de la protección de memoria
WB	Ninguna

Las excepciones más complejas suceden en la etapa EX o MEM del pipeline y deben poder reiniciarse. Un fallo de página debe ser reiniciable y necesita la intervención del SO.

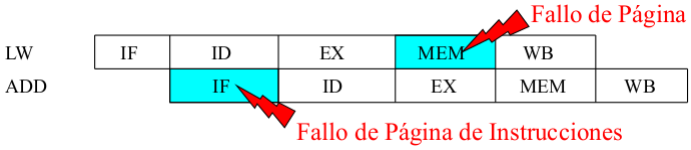
El **tratamiento de excepciones en DLX** se realiza de la siguiente forma:

1. Forzamos la entrada de una instrucción TRAP en IF
2. Impedimos que las instrucciones que siguen a la causante de la excepción y esta misma puedan cambiar el estado de la máquina. Para ello, el SO guarda el PC de la instrucción que causó el fallo y lo usará para volver de la instrucción.

Decimos que el pipeline implementa **excepciones precisas**.

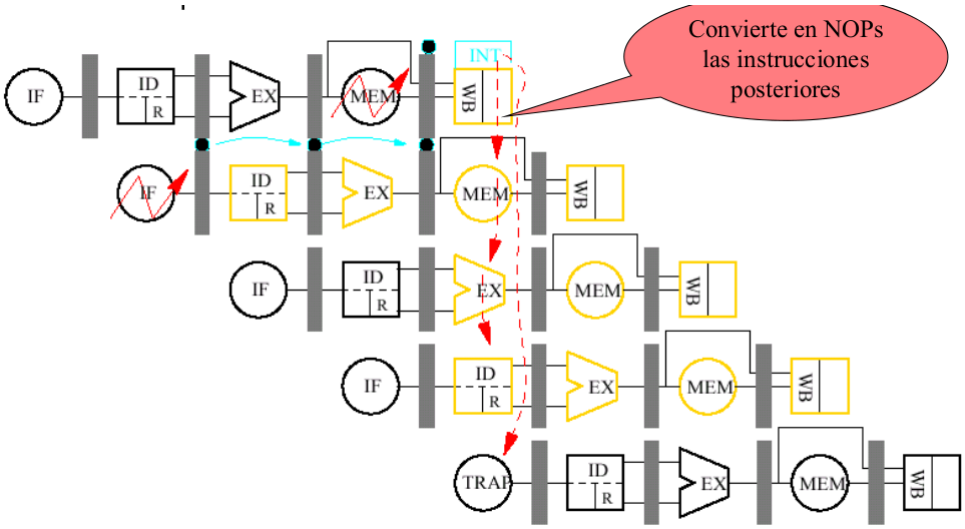


Podría darse el caso de que una instrucción cause una excepción antes de que lo haga otra instrucción anterior:



Esto tiene dos posibles soluciones:

- No usar el mecanismo de excepciones precisas y tratar las excepciones en el orden de aparición.
- Tratar las excepciones en el orden del programa. Para esto, cada instrucción que entra al pipeline tiene asociado un registro con tantos bits como etapas en las que puede originar excepciones, si se produce una excepción se pone a 1 el bit de la etapa correspondiente y se convierte en NOP la instrucción, en la última etapa se mira si ocurrió excepción (algún bit está a 1) y se trata con el mecanismo de excepciones precisas.



3.4 Segmentación para punto flotante

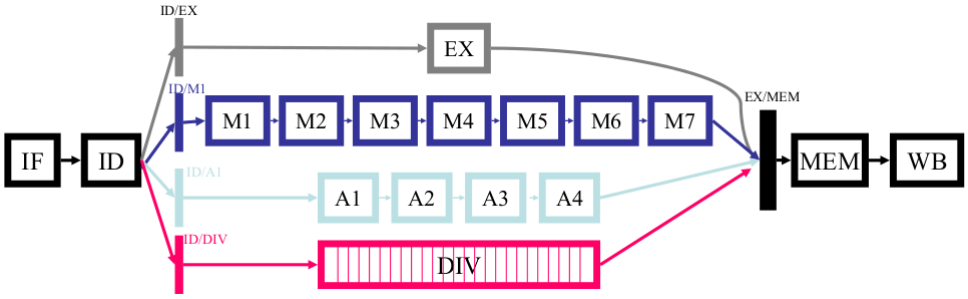
No podemos exigir que todas las operaciones de punto flotate de DLX se completen en un ciclo de reloj, pues obligaría a tener un reloj muy lento o a implementaciones muy costosas en las unidades de coma flotante. Así, estas instrucciones necesitarán varios ciclos en la etapa de ejecución. Puede verse como si las operaciones PF repitieran varias veces la etapa EX y tuviéramos varias UF de PF.

Vamos a considerar una versión de DLX con 4 UFs:

- Unidad principal de enteros, maneja cargas y almacenamientos, ALU enteras y saltos
- Multiplicador entero y PF
- Sumador PF que suma, resta y hace conversiones PF
- Divisor de enteros y PF, no segmentada

Se define la **latencia** de una UF como el número de ciclos requeridos para completar su operación. El **tiempo de iniciación** es el número de ciclos que deben pasar entre la emisión de dos operaciones del mismo tipo.

Nuestro nuevo DLX queda:

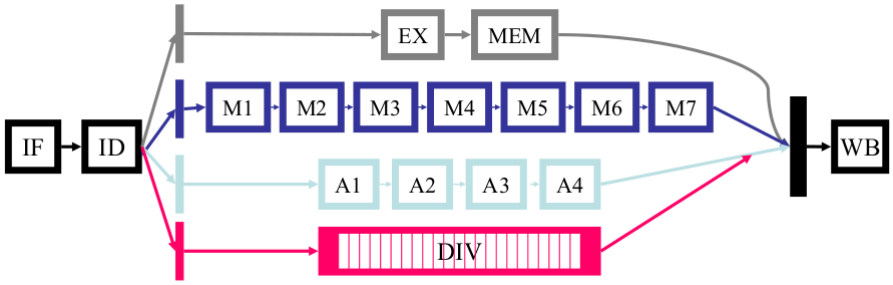


Necesitamos poner **pipeline registers** entre las etapas de las UF de PF, así como extender el registro ID/EX para que ahora sea ID/EX, ID/A1, ID/M1, ID/DIV. Pero solo necesitamos un registro EX/MEM pues únicamente podremos teer una instrucción a la vez entrando en dicha etapa.

3.4.1 Adelantamientos en el nuevo cauce

Si es necesario adelantar el dato PF debe habilitarse el multiplexor correspondiente para que la entrada se coja del registro intermedio de la instrucción que aún no ha completado. Los adelantamientos deben hacerse hacia la primera etapa de la UF consumidora desde la última etapa de la UF productora.

El hecho de que solamente las instrucciones enteras accedan a memoria nos permitiría modificar el esquema anterior, quedando:



3.4.2 Riesgos en PF

El solapamiento en la ejecución de instrucciones cuyos tiempos de ejecución difieren, y el que algunas UF no estén completamente segmentadas, crea varias complicaciones:

- Pueden aparecer riesgos estructurales en el acceso a la UF de la división
- Como cada instrucción tiene diferente número de ciclos de ejecución, podría haber varias instrucciones que quieran escribir a la vez su resultado final en el banco de registros
- Puede ocurrir que las instrucciones terminen en distinto orden a como fueron emitidas, causando esto posibles problemas con las excepciones
- Pueden aparecer riesgos WAW
- Como las operaciones tienen mayores latencias, habrá más detenciones debidas a riesgos RAW

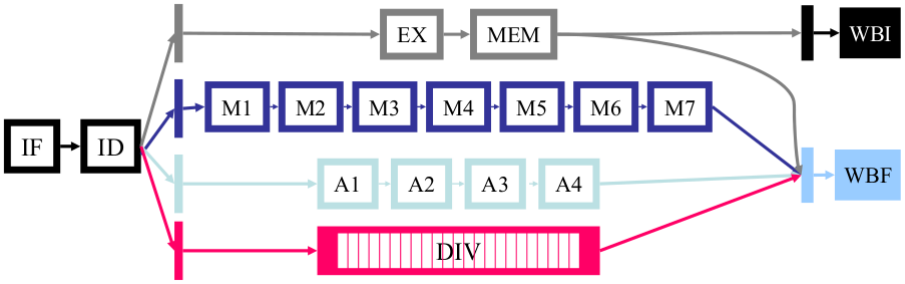
Como **solución** podemos extender la unidad detectora de riesgos para prevenir o eliminar los riesgos que introducen las unidades funcionales PF.

Banco de registros independientes para enteros y PF

Solo las cargas/almac de PF y los movimientos entre registros PF y enteros involucran a los dos bancos de registros en una misma instrucción, con lo que solo en estas situaciones pueden producirse riesgos entre ambos.

Como **ventajas** presenta que no se pueden producir riesgos estructurales en operaciones enteras; se duplica el número de registros sin complicar el número de registros, ni tiempo de acceso, ni añadir bits al formato de instrucción; y se duplica el ancho de banda de registros sin añadir puertos.

Como **desventajas** tiene que a veces hay que transferir información entre los dos bancos de registros y que se limita a priori el número de registros de cada tipo.



- Aún podrían aparecer riesgos en la escritura del banco de registros PF, debido a la diferente latencia de cada UF PF. Con un único puerto de escritura en el banco de registros PF, la máquina debe **serializar la finalización** de las instrucciones. Otra opción es **incrementar el número de puertos**, pero esta opción sería poco utilizada y complica mucho el HW.

  Para tratar este riesgo hay que detectar en qué ciclo de reloj va a usar el puerto de escritura del banco de registros cada instrucción y, si varias coinciden, solo emitir una de ellas y detener el resto. Esto se puede implementar mediante un **vector de bits de reserva** del ciclo para acceso al recurso. Al entrar la instrucción mira si estará libre el recurso (bit a 0) y lo pondrá a 1 reservándolo. Si está a 1 nos detenemos un ciclo. Cada ciclo el vector se desplaza a la izquierda un bit.

- También podrían darse riesgos WAW, aunque estos solo ocurren cuando se ejecuta una instrucción inútil. Un buen compilador nunca generaría dos escrituras en el mismo registro sin lecturas intermedias, pero puede haber situaciones inesperadas, como que ocurra esto entre una instrucción del programa y otra de la rutina de atención a una excepción.

Hay dos posibles soluciones. La primera es detectar estos riesgos en ID e insertar ciclos de parada. Otra es detectarlo en ID y anular la primera instrucción con el fin de que no llegue a escribir su resultado.

- Un último riesgo es el de que una instrucción posterior pudiera terminar antes que otra anterior. Esto pasa porque estamos dejando que las instrucciones terminen en diferente orden a como fueron emitidas, es lo que se llama terminación fuera de orden. Podemos ignorar el problema y conformarnos con excepciones imprecisas, pero, hoy en día, la mayoría de procesadores impiden que una instrucción cambie el estado de la máquina si hay instrucciones previas que aún no han acabado. Para ello las instrucciones realizan las escrituras en el ROB (ReOrder Buffer) y de ahí se pasan al banco de registros o memoria cuando corresponda.

3.5 Emisión de múltiples instrucciones

Ahora vamos a intentar obtener un $CPI < 1$. Esto solo es posible si ejecutamos simultáneamente varias instrucciones por ciclo.

Esto implica una mayor presión sobre la memoria y los registros y una mayor probabilidad de riesgos y dependencias. Serán necesarias técnicas para resolver dependencias de datos (ejecución fuera de orden) y para resolver dependencias de control (especulación). Se necesitará, por tanto, un SW más sofisticado y un mayor consumo energético.

3.5.1 Procesadores superescalares

Lanzan un número variable de instrucciones por ciclo, entre 0 y 8. La planificación es **estática** por el compilador y **dinámica** por el HW del procesador. Y las reglas de ejecución pueden ser **en orden** o **fuera de orden (con especulación)**.

3.5.2 Very Long Instruction Word (VLIW)

Emiten un número fijo de instrucciones independientes empaquetadas en una macroinstrucción. El compilador se encarga de la extracción de ILP mediante **planificación estática**.

Tiene la ventaja de que el diseño HW es más simple, lo que redunda en una mayor frecuencia de reloj y un menor consumo.

Sin embargo, no se beneficia de técnicas de planificación dinámica, presenta incompatibilidad binaria (WTF) en una familia de procesadores y suele incrementar el tamaño del código.

4 Sistema de Memoria de Altas Prestaciones

La gran diferencia en el crecimiento de las prestaciones entre los procesadores y la memoria ha obligado a estudiar nuevas medidas para mejorar los subsistemas de memoria.

Un ordenador típico contiene una jerarquía de memoria formada por los registros de la CPU, una memoria caché (con varios niveles), una memoria principal, memoria secundaria (discos) y memoria de almacenamiento masivo (DVD, penDrive,...). El coste de todo el sistema de memoria excede al de la CPU, así pues es importante optimizar su rendimiento.

Los **objetivos** de la jerarquía de memoria son hacer que el usuario tenga la ilusión de que dispone de una memoria con tiempo de acceso similar al nivel más rápido y un coste por bit similar al del nivel más barato.

La gestión en tiempo de ejecución de la jerarquía de memoria afecta a los niveles de memoria caché, principal y secundaria. Los registros del procesador normalmente los asigna el compilador y la memoria de almacenamiento masivo se usan para backup.

- **Gestión de la memoria caché:** controla la transferencia de información entre caché y memoria y suele llevarse a cabo mediante HW específico (controlador de caché y Memory Management Unit, MMU).
- **Gestión de la memoria virtual:** controla la transferencia de información entre la memoria principal y la secundaria. Se realiza mediante una combinación HW (MMU) y SW (SO).

La memoria **caché** retiene información recientemente usada y también cercana a la recientemente usada. Tanto la memoria principal como la memoria caché se dividen en **bloques** de igual tamaño.

La jerarquía de memoria sigue tres principios:

- **Inclusión:** cualquier información almacenada en un nivel de memoria, debe encontrarse también en los niveles inferiores.
- **Coherencia entre niveles:** si un bloque de información se actualiza en un nivel, deben actualizarse los niveles inferiores. Para ello puede usarse post-escritura o escritura directa.
- **Localidad:** las referencias a memoria (datos e instrucciones) se concentran en regiones del tiempo y el espacio. Puede ser:
  - **Temporal:** las posiciones de memoria recientemente referenciadas serán próximamente referenciadas.
  - **Espacial:** tendencia a referenciar elementos de memoria cercanos a los últimos elementos referenciados

Y algo de terminología:

Bloque: unidad mínima de transferencia entre dos niveles

Acierto: el dato solicitado está en el nivel i. Se mide tanto la **tasa de aciertos**, que es la fracción de accesos encotnrados en el nivel i; como el **tiempo de servicio en caso de acierto**, que es el tiempo de acceso al nivel i más el tiempo de detección del acierto.

Fallo: el dato solicitado no está en el nivel i y es necesario buscarlo en el nivel $i+1$. Se miden la **tasa de fallos**, que coincide con $1 - T_{Aciertos}$; y el **tiempo de penalización por fallo**, que es el tiempo de sustitución de un bloque del nivel i más el tiempo de acceso al dato.

El tiempo de servicio en caso de acierto debe ser mucho menor que el tiempo de penalización en caso de fallo.

4.0.1 ¿Dónde puede situarse un bloque en una caché?

$$\acute{I}ndice\ Conjunto = (Bloque\ Memoria) \mod (Conjuntos\ Cache)$$

Hay varios tipos de cachés:

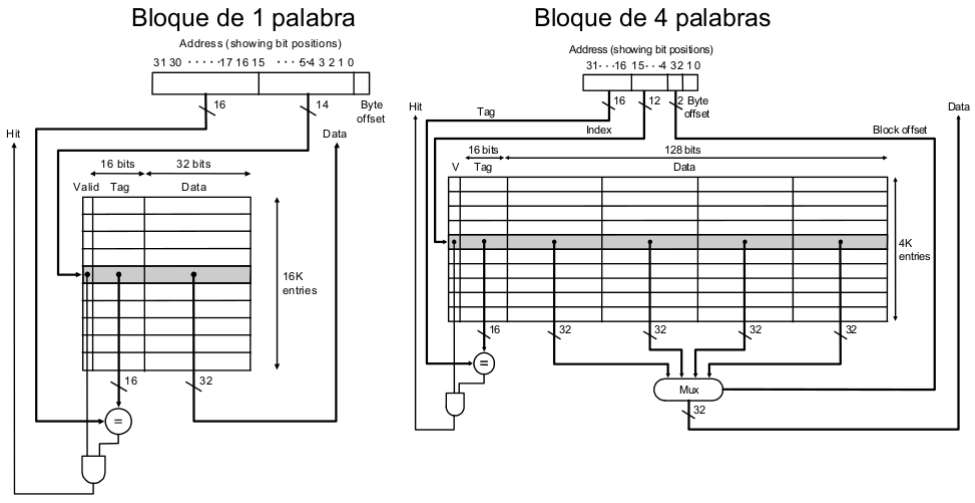
- **Correspondencia directa:** conjuntos de un solo bloque. Cada dato solo puede colocarse en una posición de la caché.
- **Totalmente asociativa:** un solo conjunto con la cantidad máxima posible de bloques.
- **Asociativa por conjuntos:** cada conjunto comprende varios bloques de la caché. El bloque va a un conjunto y luego se puede situar en cualquier hueco del conjunto. Si hay n bloques en un conjunto se llama asociativa por conjuntos de n vías.

4.0.2 Cómo sabemos si un bloque está en la caché?

De la dirección del dato de memoria buscado se extraen los campos índice, etiqueta y offset.

- El **índice** indica el conjunto en el que puede encontrarse el bloque
- La **etiqueta** diferencia todos los bloques de memoria que mapean al mismo conjunto
- El **offset** selecciona la palabra o byte deseado dentro del bloque
- Si el tamaño de la caché no cambia, aumentar la asociatividad hace que disminuya el tamaño del campo índice y aumente el de la etiqueta

Cada bloque de la caché tiene un **bit de validez** que indica si el bloque contiene información válida. Cuando se busca un bloque en una caché, las etiquetas de todos los bloques del conjunto se comparan en paralelo con la etiqueta de la dirección buscada.



4.0.3 ¿Qué bloque se reemplaza en caso de fallo?

Cuando ocurre un fallo, el controlador de memoria debe seleccionar un bloque para ser sustituido, esto dependerá de la asociatividad:

- **Correspondencia directa:** solo hay un bloque posible por lo que no se aplica ninguna política de reemplazo

- **Asociativas:** varops bloques candidates a ser reemplazados. Se aplica una política de reemplazo:
  - **Aleatoria:** sencilla de implementar
  - **LRU:** bloque menos recientemente usado. Más efectiva, pero al aumentar el número de bloques por conjunto se vuelve más costosa
  - **FIFO:** se elige por orden de antigüedad

4.0.4 ¿Qué ocurre en las escrituras?

Las escrituras son más difíciles de implementar, ya que no se puede comenzar a modificar un bloque hasta que no se ha comparado la etiqueta y normalmente solo se modifica una parte del bloque. Para manejarlas, están las políticas de escritura:

- **Escritura directa:** la información se escribe tanto en caché como en el siguiente nivel de memoria. Es más fácil de implementar, el siguiente nivel siempre queda actualizado, solo se escribe la palabra modificada, no el bloque completo, puede provocar más esperas hasta que se realiza la escritura. Una optimización común es un **buffer de escritura** que permite al procesador continuar tan pronto los datos están en el buffer.
- **Post-escritura:** la información se escribe solamente en caché. Cuando el bloque es sustituido, se actualiza el siguiente nivel. Para implementarlo se necesita un **bit de sucio**. Las escrituras se realizan a la velocidad de la caché y varias escrituras en un bloque solo requieren una escritura en memoria. Necesita menos ancho de banda, ya que hay escrituras que no van a memoria.

En caso de ocurrir un fallo de caché al escribir, pueden seguirse dos políticas:

- **Carga en escritura:** si cuando se va a escribir el dato no se encuentra el bloque en caché, se lee el bloque y se carga en caché.
- **No carga en escritura:** el bloque se modifica directamente en el nivel inferior y no se lleva a caché.

Las combinaciones más usuales de políticas son post-escritura/carga en escritura y escritura directa/no carga en escritura.

4.0.5 Memoria virtual

Surge para permitir que los procesos manejen de forma automática un conjunto de datos mayor que la memoria real del sistema. Permite que cada proceso tenga su propio espacio de direcciones y que datos de procesos distintos convivan en la memoria.

Los **objetivos** son poder contar con más memoria de la que físicamente tenemos. Proteger la memoria entre procesos, así como zonas de memoria específicas. La **relocalización** permite que un programa se ejecute en cualquier lugar de la memoria física. Por último, una carga más rápida de procesos.

Una **página o segmento** se usa para referirse al bloque de memoria.

Un **fallo de página** se da cuando no se encuentra una página buscada en nuestro espacio de direcciones virtuales.

La CPU produce direcciones virtuales que se traducen mediante una combinación de HW y SW a una dirección física, con la que se accede a memoria principal. A este proceso se le llama **mapeo de memoria**.

Los sistemas de memoria virtual se dividen en dos clases:

- Los que utilizan bloques de tamaño fijo, páginas.

- Los que utilizan bloques de tamaño variable, segmentos.

El uso de memoria virtual paginada/segmentada afecta a la CPU, pues el direccionamiento para memoria virtual paginada tiene una única dirección de tamaño fija dividida en nº de página y desplazamiento. Para la segmentada se encesitan dos palabras por dirección: una para el nº de segmento y otra para el direccionamiento dentro del segmento.

- **Comparación cuantitativa** de la memoria caché y la memoria virtual:

Parámetro	Caché de primer nivel	Memoria virtual
Tamaño de bloque (página)	16-128 bytes	4096-65,536 bytes
Tiempo de acierto	1-3 ciclos de reloj	100-200 ciclos de reloj
Penalización por fallo	8-200 ciclos de reloj	1,000,000-10,000,000 ciclos de reloj
(tiempo de acceso)	(6-160 ciclos de reloj)	(800,000-8,000,000 ciclos de reloj)
(tiempo de transferencia)	(2-40 ciclos de reloj)	(200,000-2,000,000 ciclos de reloj)
Tasa de fallos	0.1-10%	0.00001-0.001%
Mapeo de direcciones	Dirección física de 25-45 bits a dirección de caché de 14-20 bits	Dirección virtual de 32-64 bits a dirección física de 25-45 bits

Comparación cualitativa:

- El **reemplazo** en las cachés se realiza por HW mientras que en la memoria virtual se hace mediante una combinación HW-SW controlada por el SO
- El **tamaño de la dirección** que maneja el procesador determina el tamaño de la memoria virtual, pero el tamaño de la memoria caché es independiente del tamaño de esa dirección
- En la **memoria secundaria**, además de las direcciones de los procesos, también se ubican los ficheros necesarios por los programas, que normalmente no forman parte del espacio de direcciones del proceso

4.0.6 Las 4 preguntas de la memoria virtual

- **¿Dónde ponemos la página en memoria principal?**

  La penalización es tan alta que el SO permite a la página situarse en cualquier lugar de la memoria principal

- **¿Cómo sabemos si la página está en memoria principal?**

  Con una **tabla de páginas** indexada por el número de páginas virtual, que nos da la página física. Algunos SO aplican una **tabla de páginas invertida** para reducir el tamaño. Para traducir la página rápidamente, se utiliza una pequeña caché llamada **TLB**.

- **¿Qué página deberíamos reemplazar en caso de fallo de memoria virtual?**

  Debido a la penalización por fallo de página, el objetivo del SO es minimizar los fallos de página. La mayoría de SO intentan usar la política LRU usando un bit de referencia que se limpia periódicamente.

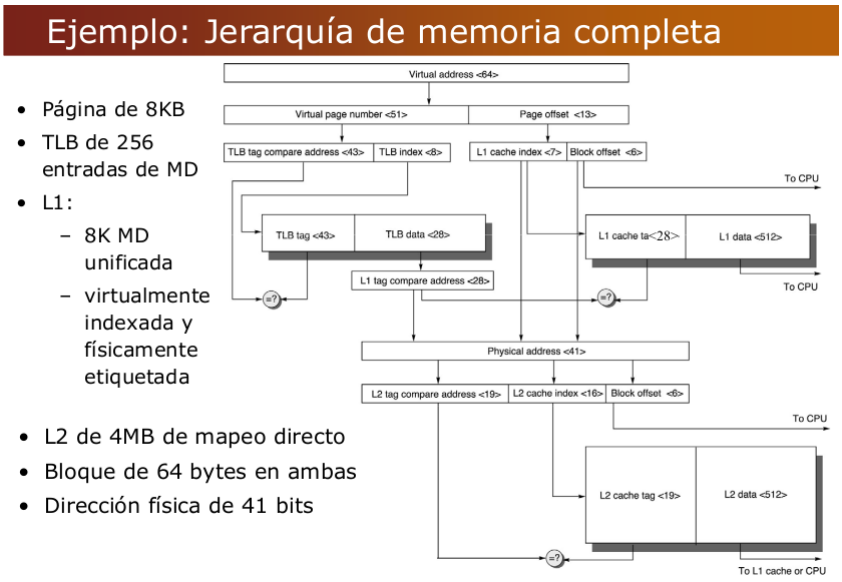
- **¿Qué ocurre en las escrituras?**

  La penalización por acceder a disco es tan alta que es imprescindible usar **postescritura**. Las páginas, como en las cachés, se marcan con un bit de sucio.

4.0.7 TLB

Las tablas de páginas son tan grandes que deben estar en memoria principal y pueden incluso estar también paginadas. Cada acceso a memoria principal, por tanto, cuesta dos accesos a memoria: uno para obtener la dirección física y otro para acceder a ella.

La **solución** pasa por usar una caché con las últimas traducciones realizadas, que se llama **TLB (Translation Lookaside Buffer).** No reemplaza la tabla de páginas, solo acelera las traducciones. Devuelve la página física y los bits de control de la tabla de traducción de páginas (protección, validez, uso y sucio). Para cambiar la dirección física de una página o su código de protección, el SO debe forzar a que esa página salga del TLB. El bit de sucio que cada página tiene en el TLB significa que el contenido de la página de la página se ha modificado, no que la dirección física haya cambiado.



4.1 Evaluación del rendimiento de la jerarquía

$$T_{CPU} = (Ciclos_{CPU} + Ciclos_{ParadaMemoria}) \cdot T_{Ciclo}$$

$$Ciclos_{ParadaMemoria} = Fallos \cdot PF = NI \cdot AMI \cdot TF \cdot PF$$

Donde PF es la penalización por fallo, NI el número de instrucciones, AMI el número medio de accesos a memoria por instrucción y TF la tasa de fallos.

¿Qué métrica podemos emplear para medir el rendimiento de la jerarquía de memoria?

$$T_{am} = T_{sa} + m \cdot T_{pf}$$

T_{am} es el tiempo medio de acceso a memoria, T_{sa} el tiempo de servicio en caso de acierto, m la tasa de fallos y T_{pf} el tiempo de penalización por fallo.

Para mejorar el rendimiento de una jerarquía de memoria, necesitamos mejorar uno o varios de los términos de esta expresión.

4.2 Reducción de la tasa de fallos de Caché

4.2.1 Clasificación de fallos de caché

- **Forzosos:** el primer acceso a un bloque de memoria no puede estar en la caché
- **Capacidad:** la memoria caché no tiene el tamaño suficiente para contener todos los bloques necesatios. En un momento determinado uno de los bloques que se han utilizado tiene que dejar hueco a otro. Se produce fallo si se vuelve a referenciar al bloque desalojado
- **Conflicto:** la memoria caché no es totalmente asociativa. Aciertos en una caché totalmente asociativa que se vuelven fallos en una asociativa por conjuntos de n vías se deben a más de n peticiones sobre algún conjunto

4.2.2 Aumento del tamaño de bloque

Es la manera más sencilla de reducir la tasa de fallos. Esto reduce los fallos forzosos al aumentar la localidad espacial. Aunque aumenta la penalización por fallo, pues cuesta más mover un bloque más grande. Además, al reducir el número de bloques en caché pueden aumentar los fallos por conflicto y por capacidad.

Lo que buscamos es minimizar la tasa de fallos y la penalización por fallo. Por tanto, la selección del tamaño de bloque depende tanto de la latencia como del ancho de banda del nivel inferior:

- Una **gran latencia y un gran ancho de banda** aconsejan un tamaño de bloque grande, pues la caché obtiene muchos más bytes por fallo por un pequeño incremento de la penalización
- Una **baja latencia y un bajo ancho de banda** aconsejan un tamaño de bloque pequeño, pues el tiempo que se gana respecto a un bloque grande es pequeño, y al tener un número grande de bloques pequeños se reducirán los fallos por conflictos

4.2.3 Aumento de la asociatividad

Aumentar la asociatividad incrementa el tiempo de servicio en caso de acierto, pues es necesario complicar el HW, pero reduce la tasa de fallos. Se debe llegar a una solución de compromiso.

Hay dos heurísticas muy usadas:

- Una caché asociativa por conjuntos de 8 vías se comporta, a efectos prácticas, como una caché totalmente asociativa
- Una caché de correspondencia directa de tamaño N tiene aproximadamente la misma tasa de fallos que una asociativa por conjuntos de 2 vías de tamaño $\frac{N}{2}$

4.2.4 Optimizaciones del compilador

Permiten reducir la tasa de fallos sin ningñun cambio en el HW. Vamos a ver algunos tipos:

- **Combinación de arrays:** su objetivo es reducir la tasa de fallos al reducir la localidad espacial. Algunos programas referencian múltiples arrays en la misma dimensión, con los mismos índices al mismo tiempo, estos accesos pueden interferir entre ellos provocando fallos de caché. La **solución** es combinarlos en un único array donde cada casilla contenga los elementos necesarios de cada array original. De esta forma todos los datos podrían estar a la vez en un mismo bloque de caché.

- **Intercambio de iteraciones:** algunos programas tienen bucles con iteraciones consecutivas que no acceden a los datos de forma secuencial. Intercambiando el roden de los bucles podemos hacer que se accedan los datos en el orden en que están almacenados en memoria, reduciendo los fallos de caché. Se maximiza el uso de los datos de un bloque de caché antes de que este se descarte. Se mejora, como antes, la localidad espacial.

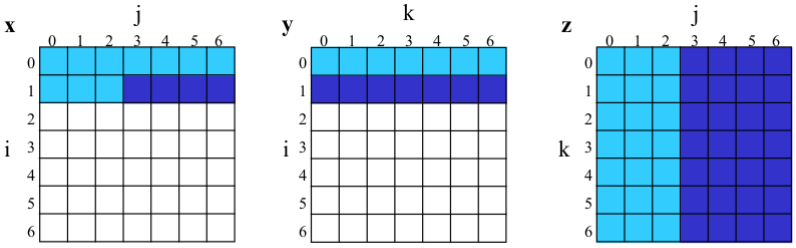
- **Unión de bucles:** otros programas tienen secciones de códigos separadas que acceden los mismos arrays, con los mismos bucles, pero realizando operaciones diferentes en los mismos datos. Si fusionamos el código en un único bucle, los datos que se cargan en caché pueden utilizarse para las distintas operacionea antes de desalojarse. De esta manera mejoramos la localidad temporal

- **Blocking:** el objetivo ahora es reducir la tasa de fallos al mejorar la localidad temporal que resulte útil cuando se usan varios arrays, unos accedidos por filas y otros por columnas. La idea es que en vez de operar sobre columnas o filas enteras, se opere sobre submatrices o bloques, maximizando el acceso a los datos de la caché antes de sustituirlos

  El número de fallos de caché depende de N (tamaño matriz) y del tamaño de la caché. Para asegurar que el número de elementos accedidos cabe en memoria, se cambia el código original para que actúe sobre una submatriz de tamaño $B \times B$, accediendo los bucles interiores en bloques de tamaño B, en lugar de inicio a fin de los arrays iniciales. Al tamaño B se le llama **blocking factor**.

```
/* Antes */
for (i=0; i<N; i=i+1)
for (j=0; j<N; j=j+1)
{       r = 0;
        for (k=0; k<N; k=k+1)
                r = r + y[i][k] * z[k][j];
        x[i][j] = r;
}
```

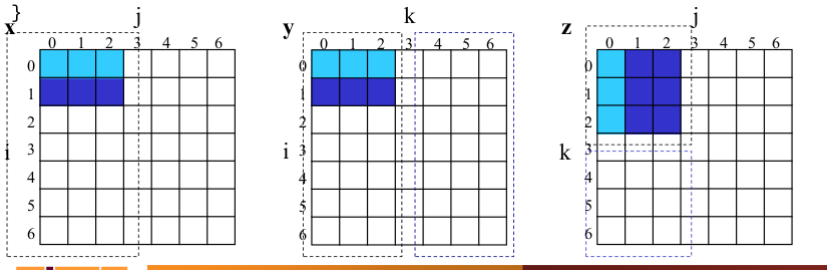
- Los dos bucles interiores leen todos los N × N elementos de z, acceden repetidamente los mismos N elementos en una fila de y, y escriben una fila de N elementos de x.



Situación en un momento dado de los arrays: en blanco las zonas aún no accedidas, en claro las accedidas hace más tiempo, y en oscuro las accedidas recientemente.

```
/* Después */
for (jj=0; jj<N; jj=jj+B)
for (kk=0; kk<N; kk=kk+B)
for (i=0; i<N; i=i+1)
for (j=jj; j<min(jj+B,N); j=j+1)
{       r = 0;
        for (k=kk; k<min(kk+B,N); k=k+1)
                r = r + y[i][k] * z[k][j];
        x[i][j] = x[i][j] + r;
}
```

- Se aprovecha tanto la localidad espacial como temporal, ya que *y* se beneficia de la localidad espacial y z de la temporal
- Puede utilizarse también minimizar el número de *loads* y *stores*, asignando bloques a registros (siempre que el tamaño de bloque sea pequeño)



4.2.5 Optimizaciones HW: caché de víctimas

Consiste en añadir un pequeño buffer totalmente asociativo entre un nivel de caché y el siguiente donde se queden los datos que se van reemplazando en el nivel superior. Intenta conseguir el bajo tiempo de acierto de la correspondencia directa a la vez que reduce los fallos debidos a conflictos.

Se puede acceder al mismo tiempo a la memoria caché y a la caché de víctimas, para que la penalización por fallo no incremente.

4.2.6 Búsqueda anticipada por HW de datos e instrucciones

Una solución para evitar flalos de caché es hacer que el HW busque anticipadamente (**prefetching**) los datos antes de que los pida el procesador. Se trata de iniciar el acceso a memoria antes de que se ejecute una instrucción que produciría un fallo de caché. Tanto las instrucciones como los datos pueden anticiparse, directamente en la caché o en un buffer externo con un tiempo de acceso mucho menor que el de la memoria.

El problema de la prebúsqueda es que pueden producirse búsquedas anticipadas innecesarias y desperdiciar ancho de banda.

El **prefetching de instrucciones** suele realizarse en HW externo a la caché. Lo más típico es que el procesador busque dos bloques en cada fallo: el solicitado, que va a caché, y el contiguo, que se lleva a un buffer de instrucciones. Si el bloque solicitado se encuentra en el buffer de instrucciones se cancela la petición a la caché, el bloque es leído del buffer y se emite una petición de prebúsqueda para el próximo bloque.

El **prefetching de datos** requiere esquemas más sofisticados, como la **prebúsqueda con stride**, que en lugar de prebuscar el bloque $i + 1$, busca el $i + x$, donde x es el stride. También está la **prebúsqueda etiquetada,** que consiste en asociar un bit de etiqueta a cada bloque, inicialmente a 0. Cuando una línea es traída por fallo de caché o referenciada el bit se pone a 1. Cuando una línea es traída por prebúsqueda el bit se pone a 0. La prebúsqueda para la línea $i + x$ se inicia cuando bit de la etiqueta de la línea i pasa de 0 a 1.

4.2.7 Búsqueda anticipada controlada por el compilador

Una alternativa al prefetching HW es dejar que el compilador inserte instrucciones solicitando los datos antes de que sean necesarios. Hay dos alternativas:

- **Register prefetch:** carga el valor en un registro

- **Caché prefetch:** carga los datos en caché

Cualquier de los dos podría ser **faulting** o **nonfaulting**, es decir, la dirección puede o no causar una excepción por un fallo en la dirección virtual y violaciones de protecciones. Usando esta terminología podríamos decir que una instrucción de carga usual es una faulting register prefetch instruction. La mayoría de procesadores ofrecen **nonfaulting caché prefetches**.

Al usar la búsqeuda anticipada por SW se incrementa el número de instrucciones, con lo que se debe tener cuidado de que esta sobrecarga no exceda los beneficios. Para evitar prebúsquedas innecesarias, los compiladores deben concentrarse en las referencias a memoria que tienen mayor probabilidad de fallar y que provocarían mayores detenciones.

La búsqueda anticipada por SW solo tiene sentido si el procesador puede continuar mientras se realiza la búsqueda (procesador con ejecución fuera de orden), ya que, al igual que con la búsqueda anticipada HW, el pobjetivo es que se solape la ejecución con la búsqueda de datos.

4.3 Reducción de la Penalización por Fallo de Caché

4.3.1 Cachés multinivel

La gran diferencia en prestaciones entre procesador y memoria hace que nos interesen cachés más rápidas, para cumplir con la velocidad de la CPU y cachés más grandes para evitar ir a memoria.

La solución es añadir un segundo nivel de caché entre la caché original y memoria. La caché de **primer nivel** puede ser lo suficientemente pequeña para ser casi tan rápida como el procesador. La caché de **segundo nivel** puede ser lo suficientemente grande para capturar la mayoría de los accesos que irían a memoria principal, disminuyendo así la penalización por fallo.

Ahora el **análisis de prestaciones** cambia: sabemos calcular el tiempo de acceso medio a memoria, pero ahora tendremos que distinguir entre los distintos niveles de caché:

$$\begin{cases} T_{am} = T_{SA_{L1}} + m_{L1} T_{PF_{L1}} \\ T_{PF_{L1}} = T_{SA_{L2}} + m_{L2} T_{PF_{L2}} \end{cases} \implies T_{am} = T_{SA_{L1}} + m_{L1}(T_{SA_{L2}} + m_{L2} T_{PF_{L2}})$$

Se adoptan, así, los términos:

- **Tasa de fallos local:** el número de fallos en la caché dividido por el número total de accesos a esa caché.
- **Tasa de fallos global:** número de fallos de caché dividido por el número total de accesos a memoria generados por la CPU.

La tasa de fallos local para la de segundo nivel es mayor ya que la caché de primer nivel captura la mayoria de accesos a memoria. La tasa de fallos global es más útil e indica qué fracción de los accesos a memoria van a memoria principal.

Sobre el diseño de la L2

La principal diferencia entre los dos niveles de caché es que la velocidad de la L1 afecta al tiempo de ciclo de la CPU, mientras que la de la L2 afecta a la penalización por fallo de la L1. Se nos presentan dos posibilidades:

- **Inclusión multinivel:** L1 está contenida en L2. La coherencia entre cachés puede determinarse mirando solamente en la L2. Podemos usar bloques más pequeños para L1 y mayores para L2.
- **Exclusión multinivel:** L2 no contiene a L1. No se permite tener en L2 una copia de un dato que esté en L1. Un fallo de L1 provoca un intercambio de bloques entre L1 y L2 en lugar de reemplazarlo.

Reducción de la tasa de fallos de L2

Un mayor grado de **asociatividad** en L2 es muy interesante, pues incrementaría el tiempo de servicio en caso de acierto de la L2, pero hemos visto que para L2 lo más importante es reducir la tasa de fallos para evitar tener que ir a memoria.

Aumentar el **tamaño** de la caché de segundo nivel reduce los fallos por conflicto al distribuir los datos entre más bloques y elimina muchos de los fallos de capacidad.

Incrementar el **tamaño de bloque** de L2 aumentaría los fallos por conflicto para las cachés de segundo nivel pequeñas, pero como son bastante grandes es posible tener tamaños de bloque de 64, 128 o 256 B. A mayor tamaño de bloque, mayor penalización por fallo.

4.3.2 Buffer de Escritura

Las escrituras dejan el dato en el buffer, en lugar de esperar a que se escriba en memoria, de forma que las siguientes instrucciones puedan seguir ejecutándose. Su uso es imprescindible en cachés de **escritura directa**, pues, sin él, todas las escrituras provocarían largas detenciones. Permite que la caché siga atendiendo otros accesos mientras realiza una escritura en memoria o en el siguiente nivel de caché.

También mejora el rendimiento en cachés con **post-escritura**, acelerando los reemplazos. Supongamos un fallo de lectura que va a provocar el desalojo de un bloque de caché marcado. La lectura debería esperar a que se escribiera en memoria principal el bloque desalojado. Sin embargo, podríamos tener un buffer odnde copar el bloque modificado mientras que se trae de memoria principal el bloque demandado por la lectura para que el procesador pueda continuar. Una vez terminada la lectura, se copia el bloque modificado desde le buffer intermedio a memoria principal.

En ambios casos debemos controlar las posibles lecturas sobre un bloque que estuviera en ese buffer intermedio.

Nótese que el buffer de escritura podría contener valores pendientes para un bloque que se ha pedido. La forma más sencilla de solucionar este problema es que la lectura se espere hasta que se vacíe el buffer de escritura, pero esto incrementaría el tiempo de penalización en caso de fallo para la lectura. La alternativa es que cuando se produzca un fallo por una lectura, se compriuebe el buffer de escritura y, si no está ahí el dato buscado y el sistema de memoria está disponible, se continue con la lectura en memoria principal.

4.3.3 Cachés no bloqueantes

Hasta ahora un acceso a memoria que produce un fallo de caché detiene el cauce hasta que se obitene la palabra que lo provoca. Un procesador que permite terminación fuera de orden no necesita parar cuando hay un fallo en la caché de datos. Así, las **cachés no bloqueantes** permiten que la caché de datos siga permitiendo accesos mientras resuelve un fallo de caché de otra instrucción. Puede permitir solo accesos que acierten (**acierto bajo fallo**) o incluso solapar varios fallos (**acierto bajo múltiples fallos**), lo que reduciría la penalización media por fallo. Sin embargo, esto incrementa la complejidad del controlador de la caché, pues puede haber varios accesos al mismo tiempo a memoria. Solo es beneficiosa si la memoria puede servir varios fallos a la vez.

4.4 Reducción del Tiempo en Caso de Acierto en Caché

4.4.1 Acceso segmentado a Caché

El tiempo de servicio en caso de acierto para L1 es un valor crítico, pues afecta directamente a la frecuencia de la CPU. Los procesadores modernos tienen segmentado el acceso a las cachés a lo largo de varios ciclos para poder soportar una elevada frecuencia de reloj. Esto aumenta el número de etapas

del cauce, provocando mayor penalización en caso de fallo de predicción de saltos y más ciclos entre la entrada de una carga y el uso del datos. Aumenta el ancho de banda de memoria, pero no disminuye la latencia de un acierto de caché-

4.4.2 Cachés pequeñas y sencillas

Una parte importante del tiempo que se tarda en un acierto e caché se gasta en leer la etiqueta y compararla con la dirección. En general, HW más pequeño significa más rápido, por lo uqe interesa que L1 sea **pequeña**. Asimismo, es crítico mantener al menos una L2 on-chip suficientemente grande para evitar la penalización de tener que salir fuera del chip.

A la vez, interesa que la caché sea lo más **sencilla** posible, para tener un tiempo de aciero pequeño. La necesidad de conseguir altas frecuencias sin incrementar mucho el número de etapas del cauce hace que las L1 sean pequeñas y sencillas.

Aunque la cantidad de caché que se integra dentro del chip del procesador va incrementándose a lo largo del tiempo, la tendencia actual es tener la misma cantidad de L1.

También se está poniendo énfasis en aumentar la frecuencia de reloj y ocultar los fallos de la L1 mediante **ejecución dinámica**, o sea, que mientras se resuelve el fallo se ejecutan otras isntrucciones del programa, y el uso de L2 para evitar tener que ir a memoria.

4.4.3 Evitar la traducción de la dirección al indexar la caché

Incluso una caché pequeña y sencilla debe hacer la traducción de la dirección virtual que maneja la CPU a la dirección fñisica para acceder a memoria. Una alternativa es usar la dirección virtual para la caché, con lo que nos evitamos tener que hacer la traducción: **cachés virtuales** frente a las cachés tradicionales que llamaremos **cachés físicas**.

Es importante distinguir la comparación de las etiquetas y la indexación de la caché.

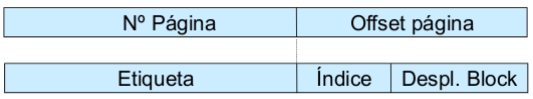
Problemas de las cachés virtuales

1. **Comprobar la protección de memoria:** al acceder a memoria se comprueban los permisos durante la traducción de dirección virtual a física. Esta información se guarda en la tabla de páginas. La **solución** es copiar la información sobre protección de la tabla de páginas en la caché en caso de fallo y comprobarla en cada acceso a caché. Cada vez que hay un fallo de caché se lleva a la misma el bloque de datos y la información de protección. Cada vez que hay un acierto se comprueban los permisos en la caché.

2. **Cambios de contexto:** en cada cambio de contexto, las mismas direcciones virtuales se refieren a distintas direcciones físicas, por tanto la caché debe vaciarse. Puede añadirse a la etiqueta un campo PID que asigna el SO, de forma que solo necesita vaciar la caché cuando se reutilice un PID.

3. **Sinónimos o alias:** se suelen utilizar distintas direcciones virtuales para la misma dirección física para compartir zonas de memoria. Esto provoca tener varias copias del mismo dato en caché. Si se modifica uno, el otro podría mantener su valor antiguo, provocando incoherencias.

4.4.4 Cachés indexadas virtualmente y etiquetadas físicamente



En estas cachés el campo índice de la cachñe ha de caer dentro del campo offset de página de la dirección virtual. Esto permite simultanear el acceso a la caché con la traducción de la etiqueta a su dirección física. Finalmente, la comparación de etiquetas se hace con direcciones físicas.

El **problema** es que el campo índice no puede exceder el campo offser. Esto se traduce en una limitación del tamaño efectivo de las cachés.

La **solución** es aumentar la asociatividad para mantener el tamaño del índice:

$$2^{índice} = \frac{TamCache}{TamBloque \times Asociatividad}$$

Así conseguimos que doblando la asociatividad tengamos una caché el doble de grande y sin aumentar su campo índice.

4.4.5 Tiempos de acceso según el tipo de caché

1. **Cachés físicas**

$$T_{am} = \overset{TLB}{T_{am\_TLB}} + \overset{dato}{T_{am\_caché}} = T_{sa_{TLB}} + m_{TLB}PF + T_{sa_{L1}} + m_{L1}PF$$

2. **Cachés virtuales**

$$T_{am} = \overset{dato}{T_{sa_{L1}}} + m_{L1}[PF + (\overset{TLB(\textbf{solo si fallo})}{T_{sa_{TLB}}} + m_{TLB}PF)]$$

3. **Cachés virtualmente indexadas y físicamente etiquetadas**

$$T_{am} = \overset{dato\ y\ TLB\ en\ paralelo}{\max\{T_{sa_{L1}}, T_{sa_{TLB}}\}} + m_{L1}PF + m_{TLB}PF$$

4.5 Organizaciones de la memoria principal

La memoria principal satisface las peticiones de las cachés y sirve como interfaz para las operaciones de E/S, ya que los datos se suelen transferir desde y hacia memoria.

Las métricas para medir el rendimiento de la memoria son **latencia** (fundamental para cachés) y **ancho de banda** (fundamental para el susbsistema de E/S).

El empleo de cachés de segundo nivel en prácticamente todos los sistemas y los tamaños de bloque bastante grandes que usan hace que el ancho de banda entre memoria principal y caché sea importante. Uno de los motivos por los que los diseñadores incrementan el tamaño de bloque de la caché es para aprovechar el elevado ancho de banda de la memoria.

Vamos a ver distintas organizaciones de memoria principal para incrementar el ancho de banda.

4.5.1 Mayor anchura

Podemos aumentar la anchura de la caché y la anchura del bus que conecta caché y memoria con el fin de aumentar el ancho de banda entre ambas.

4.5.2 Memoria entrelazada

Podemos organizar la memoria en **bancos.** Direcciones consecutivas se almacenan en bancos diferentes. Podemos leer/escribir varias palabras simultáneamente siempre que se encuentren en diferentes bancos. Además, los bancos tienen anchura de una palabra, por lo que no es necesario cambiar la anchura del bus ni de la caché.

Esta técnica es menos costosa que la anterior y proporciona resultados parecidos.

La memoria se entrelaza normalmente con un **factor de entrelazado** de una palabra, con lo que se optimizan los accesos secuenciales.

4.5.3 Bancos de memoria independientes

La memoria entrelazada utiliza bancos que trabajan con las mismas líneas de dirección compartiendo el controlador de memoria. Una generalización del entrelazado es permitir múltiples accesos independientes gracias a tener múltiples controladores de memoria, de forma que cada banco necesita líneas de dirección y datos separadas. Se puede acceder al mismo tiempo a direcciones diferentes en cada banco. Las cachés no bloqueantes solo tienen sentido si tenemos bancos independientes.