

Resumen Final Orga2

Tecnología de integración

Escenario actual

Ley de Moore

Evolución tecnológica

Tendencias tecnológicas

Arquitectura de Computadores

Arquitectura vs Microarquitectura

Arquitectura

Micro Arquitectura

Organización

Hardware

Por qué necesitamos saber todo esto?

Sistema de Memoria

Jerarquías de Memorias

Baja latencia o alta capacidad

Principio de Vecindad o Localidad

Principio de funcionamiento

Tecnologías de Memoria

Clasificación de memorias

Memorias No Volátiles (ROM)

Memorias Volátiles (RAM)

Uso en un computador

Memorias RAM dinámicas (DRAM)

Memorias RAM estáticas (SRAM)

Memorias y velocidad del Procesador

Conexión básica (Según Von Neumann)

Memoria Cache

Principio de Funcionamiento

Características y métricas

Controlador de cache

Operación de acceso a memoria para lectura

Organización de un cache

Líneas

Sistema Cache de Mapeo Directo

Sistema Cache Asociativo de 2 Vías

Coherencia de un cache

¿Qué ocurre durante las escrituras?

Políticas de escritura
Coherencia en sistemas SMP
Snoop BUS
Protocolo MESI
Coherencia en sistemas SMP con MESI
Operación del Protocolo MESI
Read For Ownership
Protocolo MESI - Diagrama de estados
Arquitecturas de cache avanzadas
Cache Multinivel
2do. Nivel On chip
3er. Nivel On chip
Smart Cache
Paralelismo a Nivel de Instrucción
Pipeline
Máquina de estados elemental
Pipeline
Hazards (obstáculos)
Unidades de Predicción de saltos
Necesidad de predicción de saltos
Asumir non-taken
Asumir taken
Delayed branch
Loop unrolling
Predicción de saltos dinámica
Superscalar
Superscalar de dos vías
Problemas
Scheduling Dinámico
Obstáculos de Datos
Ejecución Fuera de Orden
Conceptos fundamentales
Idea fuerza
Ejecución Fuera de Orden vs En Orden
Riesgos
Excepciones imprecisas
Método prehistórico
Scoreboarding
Método actual
Modelo de Tomasulo
Register Alias Table (RAT)
Reservation Station (RS)

Common Data Bus:
FPU de IBM 360/91 con la mejora de Tomasulo
Algoritmo de Tomasulo
Ejemplo del Algoritmo
Especulación por Hardware
Reordenando los resultados
Casos prácticos que mezclan todo lo visto

Pentium Pro
Three Cores Engine
Unidad de Interfaz con el Bus
Unidad de decodificación y búsqueda
Unidad de despacho y ejecución
Unidad de retiro
Intel NetBurst (Pentium 4)
Procesadores Multithread
Multicore y Manycore

Tecnología de integración

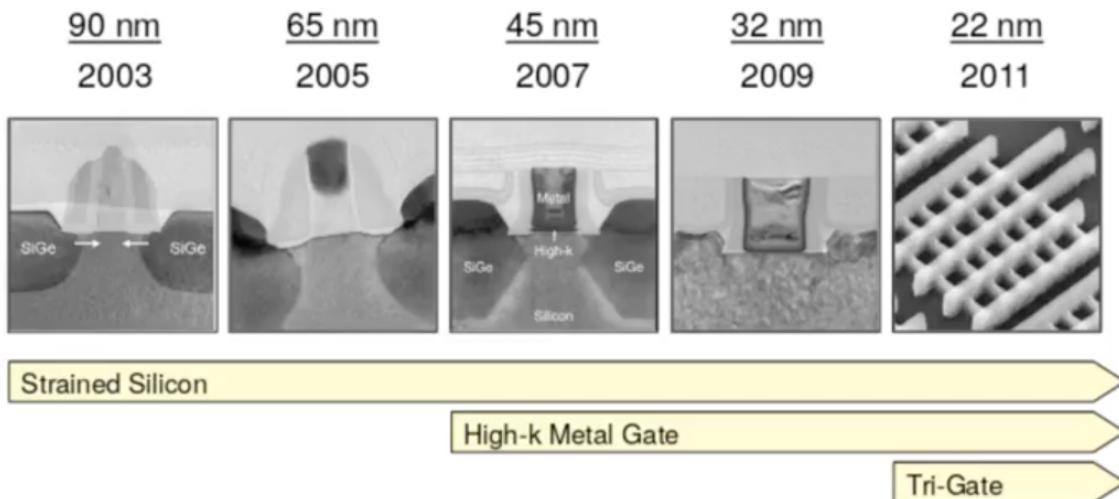
Escenario actual

- Queremos achicar los nanómetros que ocupa un transistor
- En 2012 teníamos 22nm
- Actualmente se trabaja en lograr 7 y 5nm.
- **Y luego? Violamos las leyes de la física. No podemos achicar más.**

Ley de Moore

El número de transistores que incorporemos en un chip aproximadamente se duplicará cada 24 meses.

Evolución tecnológica



Los progresos en scaling muchas veces obedecen a mejoras en el proceso de fabricación, pero cada tanto se produce algún salto cuántico en esto, que es cuando aparecen tecnologías de fabricación de semiconductores disruptivas.

- Strained Silicon
- High-k Metal Gate
- Tri-Gate:

Ademas del scaling, disminuye mucho el consumo. En 2011 el consumo de energía es una preocupación muy fuerte.

Velocidad de conmutación: Es la velocidad que tarda en la que un transistor pasa de 0 a 1. **Evoluciona en función del scaling: cuanto más chico es el transistor, más alta es la velocidad de conmutación.**

Tendencias tecnológicas

La tarea de un diseñador está inevitablemente influida por el rumbo de las tecnologías.

1. La densidad de transistores por unidad de superficie aumenta 35% por año en promedio (Ley de Moore alternativa)
2. El tamaño del die (disco con circuitos integrados) aumenta de 10 a 20% por año. Esto deriva en un crecimiento en la

cantidad de transistores de entre 40% y 55% de un año a otro.

3. La velocidad de clock ya no crece, Parecería haber alcanzado un techo en los 3 GHz aproximadamente.
4. La capacidad de almacenamiento de las memorias DRAM crece a razón de un 40% por año.
5. Los discos rígidos aumentan su capacidad 25% a 30% por año. Su costo por bit de almacenamiento se mantiene entre 50 y 100 veces por debajo del costo de un bit de memoria DRAM.

Arquitectura de Computadores

Arquitectura vs Microarquitectura

Arquitectura

Es el conjunto de recursos que nosotros tenemos como programadores para poder trabajar y poder desarrollar programas. **Se mantiene constante de un modelo a otro (en general).**

- Registros
- Set de instrucciones
- Estructuras de memoria (descriptores de segmento y de página)

Micro Arquitectura

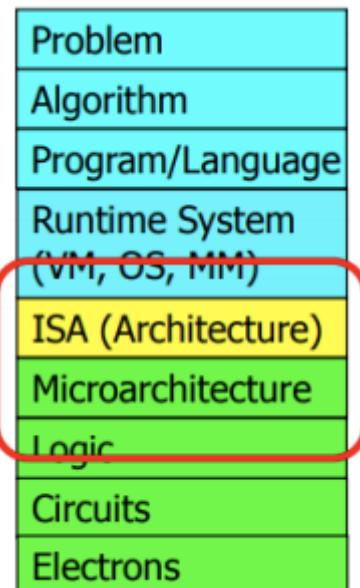
Es la implementación es el silicio de la arquitectura. Lo que está detrás del set de instrucciones. Determina de qué manera se ejecutan las instrucciones, cuán rápido o no. Cuestiones que están relacionadas a electrónica, tecnología, materiales, etc. Un procesador con una misma arquitectura tenga una estructura muy sólida, robusta, performante, y otro procesador con la misma arquitectura pero con una organización o microarquitectura diferente por debajo sea un procesador menos performante, más económico, más lento, que consuma menos.

Cambian de un modelo a otro. Es "levantar el capó y ver qué hay debajo de la arquitectura".



IA-32 es una **arquitectura**. Se inicia con el procesador 80386 en 1985 y llega hasta los procesadores Intel Core i7, i5, i3, ATOM y Xeon.

En el camino han pasado diferentes generaciones de **Microarquitectura** para más de 25 modelos de procesadores.



Microarquitectura = Organización + Hardware

Organización

Se refiere a los detalles de implementación de la ISA (Instruction Set Architecture).

- Organización e interconexión de la memoria.

- Diseño de los diferentes bloques de la CPU que van a ejecutar la ISA. Qué características van a tener. Qué grado de paralelismo va a tener. Puede ejecutar varias instrucciones a la vez, o no? Puede ejecutar instrucciones fuera de orden o no?
- Implementación de paralelismo a nivel de instrucciones, y/o de datos.
- Podemos encontrar procesadores que poseen la misma arquitectura pero organización diferente (**ejemplo: AMD FX e Intel Core i7 tienen la misma ISA, pero su organización es completamente diferente**).

Hardware

Se refiere a cosas todavía más "de abajo".

- Diseño lógico y tecnología de fabricación.
- Se pueden tener procesadores que tengan la misma ISA, la misma organización y distinto hardware.

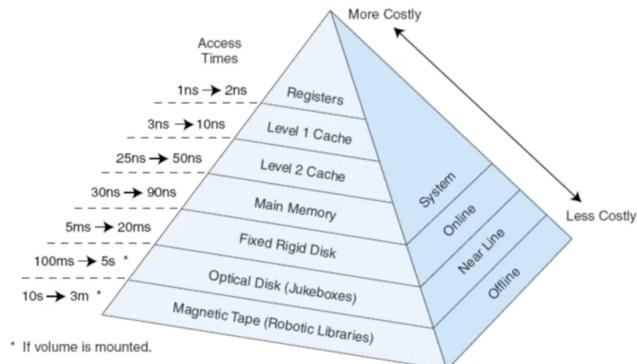
Por qué necesitamos saber todo esto?

- Comprender qué hay debajo del software.
- Comprender cómo el diseño de un hardware en particular impacta nuestras tareas como programadores.

Sistema de Memoria

Jerarquías de Memorias

Baja latencia o alta capacidad



- Cuanto más cerca de la cúspide estás, tenés memorias más veloces pero a la vez de menor capacidad, porque su costo por bit aumenta a medida que subimos.
- Queremos que nuestros datos estén lo más cerca posible de la CPU.
- Lo que tenemos más cercano al procesador son los registros.

Principio de Vecindad o Localidad

Principio de funcionamiento

El comportamiento de los algoritmos de software que se emplean habitualmente, es el que rige cómo y para qué se diseña el hardware. Muchas de las cosas que se diseña en hardware no son inventos, son simplemente observaciones de cómo se comportamiento y, entonces, buscar soluciones para agilizar ese comportamiento.



Principio de vecindad temporal

Una dirección de memoria que está siendo accedida actualmente tiene muy alta probabilidad de seguir siendo accedida en el futuro inmediato.



Principio de vecindad espacial

Si se está accediendo a una dirección determinada de memoria actualmente, la probabilidad de que esta dirección y sus direcciones vecinas sean accedidas en el futuro inmediato es muy alta.

Tecnologías de Memoria

Clasificación de memorias

Memorias No Volátiles (ROM)

- Retienen la información almacenada cuando se les desconecta la alimentación.
- Han evolucionado tecnológicamente, desde ROM que se grababan en la fábrica hasta ROMs borrables y reprogramables eléctricamente.

Memorias Volátiles (RAM)

- Pierden la información almacenada cuando se les desconecta la alimentación.
- Pueden almacenar mayores cantidades de información y modificarla en tiempo real mucho más rápido que las No Volátiles.
- Se clasifican, de acuerdo con la tecnología y diseño interno, en dinámicas (DRAM) y estáticas (SRAM).

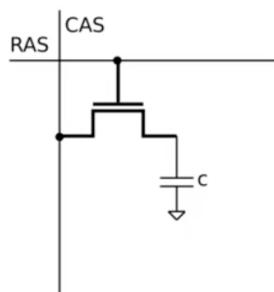
Uso en un computador

- La memoria no volátil se usa fundamentalmente para almacenar el programa de arranque de cualquier sistema.
- Se conectan en un espacio de direcciones determinado por el propio microprocesador de acuerdo a la dirección en la que éste irá a buscar la primera instrucción luego de encender el equipo.

- El resto es RAM y allí el sistema copia incluso buena parte del código de arranque para que se ejecute más rápido (recordemos que las memorias volátiles tienen menor tiempo de acceso)

Memorias RAM dinámicas (DRAM)

Tenemos una celda de memoria representada por un transistor y un capacitor.



- Almacena la información en forma de estado de carga en un capacitor y la sostiene durante un breve lapso con la ayuda de un transistor.
- Una celda (un bit) se implementa con un sólo transistor → máxima capacidad de almacenamiento por CI.
- Ese transistor está generalmente en estado de Corte. Consumo mínima energía.
- Al leer el bit, se descarga la capacidad (lectura destructiva), con lo cual necesita regenerar la carga cada vez que se la lee.
- Se descargan solas por las corrientes de fuga de los transistores, con lo cual hay que refrescarlas manualmente periódicamente. La recarga se hace haciendo una lectura "dummy" de toda una fila a la vez.

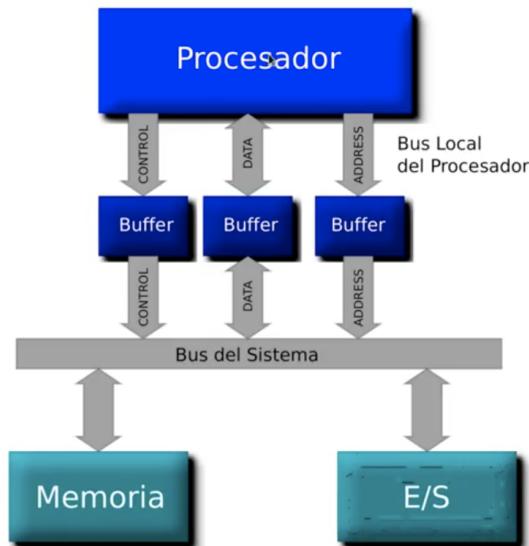
Memorias RAM estáticas (SRAM)

- Tienen 6 transistores por cada bit. Es decir, la densidad es peor que en la RAM dinámica.

- 3 de los 6 transistores están cerrados, con lo cual consumen mucha corriente, con lo cual cada celda consume mucha energía.
- No se destruye la información al leer un bit.
- La velocidad de lectura es casi instantánea.

Memorias y velocidad del Procesador

Conexión básica (Según Von Neumann)



Intel presenta su arquitectura muy parecida a la de Von Neumann. El 8088 tenía una línea de control llamada **Ready** que servía para saber cuándo una memoria estaba lista para leer o escribir un dato. Cuando los procesadores aumentaron su velocidad, las RAM se quedaron atrás.

El problema consiste en decidir qué tipo de RAM usar en el sistema. Hay dos opciones:

- **RAM dinámica (DRAM)**
 - Consumo mínimo.
 - Capacidad de almacenamiento comparativamente alta (1 transistor por bit).

- Costo por bit bajo.
- Tiempo de acceso alto (lento), debido al circuito de regeneración de carga.
- **RAM estática (SRAM)**
 - Alto consumo relativo.
 - Capacidad de almacenamiento comparativamente baja (6 transistores por bit).
 - Costo por bit alto (por ser menos densa).
 - Tiempo de acceso bajo (es más rápida).



Conclusión:

No podemos poner un banco de RAM estática porque el costo y consumo es muy elevado. Entonces, elegimos usar RAM dinámica, aunque al ser lentas no podemos aprovechar la velocidad superior del procesador (cuello de botella). **Esto último se resolvió con la memoria cache.**

Memoria Cache

Principio de Funcionamiento

Es un banco de **RAM estática** que contiene una copia de los datos e instrucciones que están en la memoria principal y el procesador necesita en este momento. La dificultad está en lograr que esta copia esté disponible justo cuando el procesador la necesita permitiéndole acceder a estos ítems sin utilizar *wait states*.

Este banco de RAM estática tiene que ser pequeño para no comprometer el costo y consumo de nuestro sistema (mismo problema de antes), pero a la vez tiene que ser

suficientemente grande para contener siempre lo que el procesador necesita.

Esta memoria cache de pequeño tamaño combinada con una gran cantidad de memoria DRAM donde almacenemos el resto de las cosas resuelve el problema mediante una típica solución de compromiso.

Quién se encarga de que las cosas que están en la DRAM que el procesador va a utilizar estén en la cache? Esto cuesta hardware. Tiene que haber un controlador que intermedie entre el procesador y las dos memorias para asegurar que eso esté completamente funcional en el momento que el procesador lo necesite. A este se lo denomina **Controlador cache**.

Características y métricas

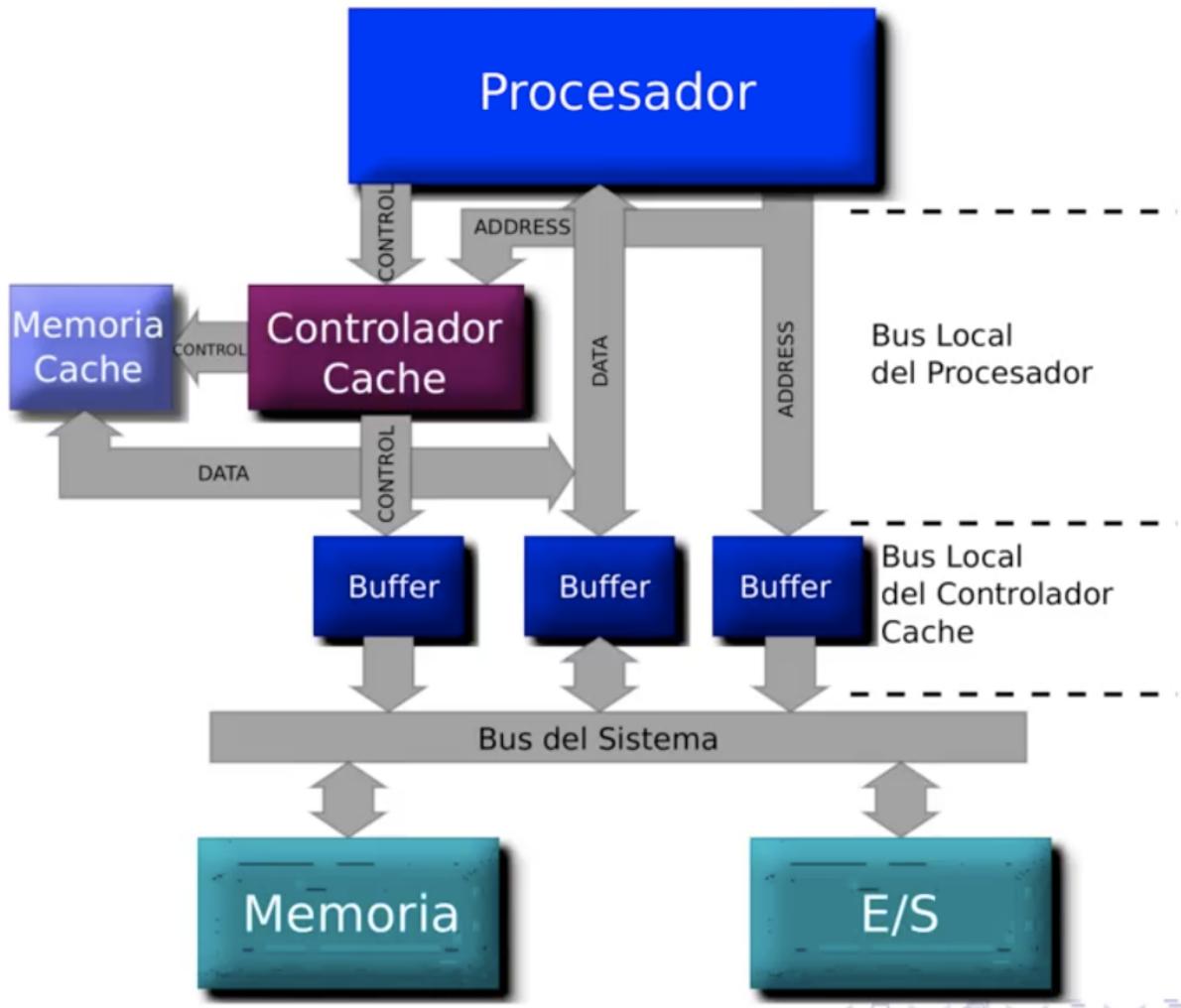
Se maneja con los principios de vecindad. Al principio el cache está vacío y el sistema es ineficiente porque tiene que ir a buscar todo a la RAM. Cuando se va a buscar algo a la RAM, el controlador la guarda también en la cache. Si se va a buscar una instrucción, el controlador busca esa instrucción y más, ya que seguro se van a utilizar las siguientes instrucciones.

Para medir la eficiencia de esto, se establecieron métricas. La operación de memoria es un **hit** o un **miss** según encuentre o no encuentre lo que busca en el cache (cuando lo encuentra es un **hit**, y cuando no lo encuentra es un **miss**). El **hitrate** se mide con la relación entre la **cantidad de hit** sobre la **cantidad de accesos totales**.

$$\text{hitrate} = \frac{\text{Cantidad de Accesos con hit}}{\text{Cantidad de Accesos Totales}}$$

Queremos que el hitrate tienda a 1.

Controlador de cache

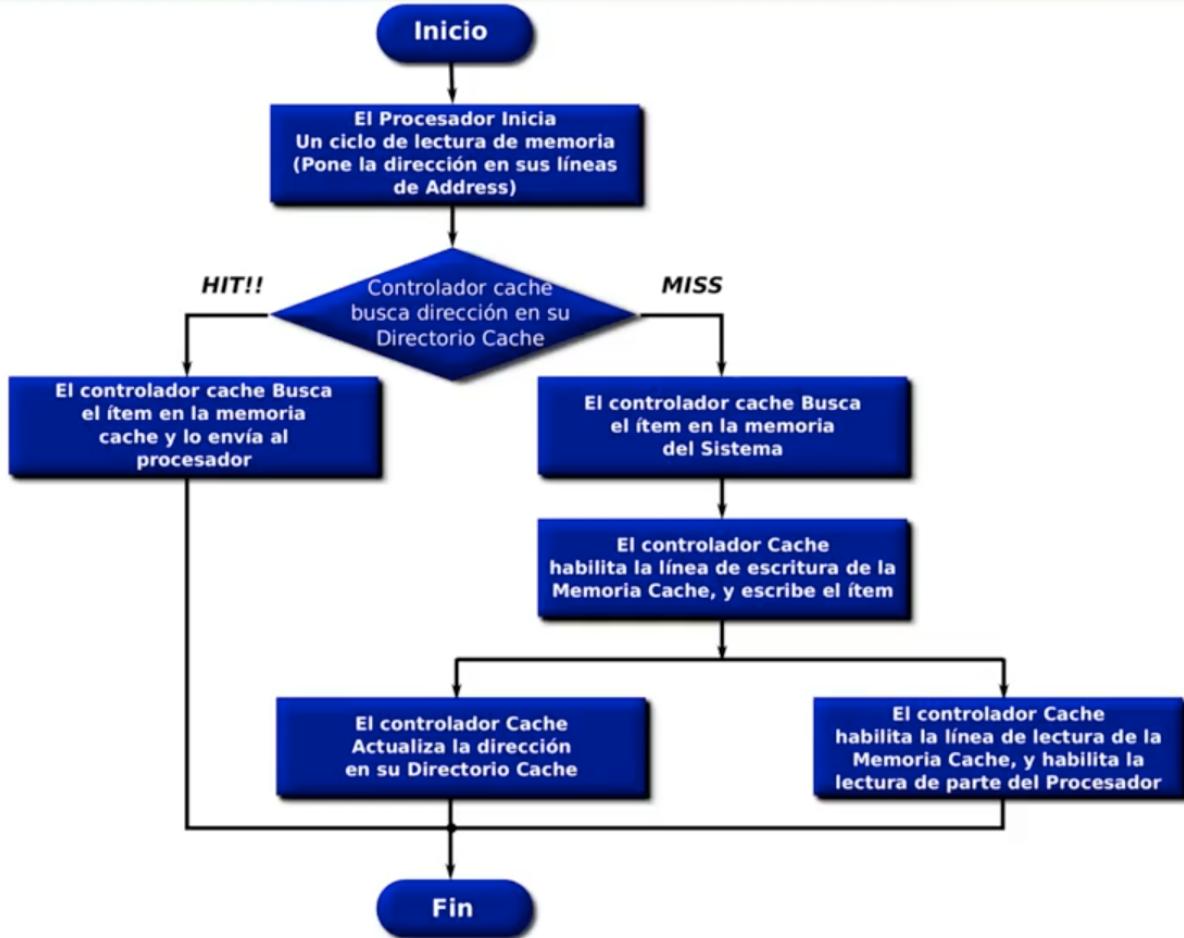


Quien maneja el BUS de control es quien maneja el BUS del sistema. **Este BUS ahora es manejado por el Controlador de Cache. El Controlador de Cache es el master del BUS de control. La idea es que intercepta los pedidos del procesador y se los manda o bien a la Memoria cache o bien a la memoria RAM según haya un hit o miss.**



El BUS de control es un bus con líneas de control del tipo **Read**, **Write**, etc.

Operación de acceso a memoria para lectura



1. El procesador inicia un ciclo de lectura. Pone la dirección de memoria en el BUS de address y **activa la línea Memory Read en el BUS de control**.
2. El controlador de Cache toma la línea de **Memory Read** (pues maneja el BUS de control) y dice, "Ok, el procesador quiere leer". También le llega el BUS de address y entonces puede saber qué dirección quiere leer. El controlador de Cache mira si la dirección de memoria está guardada en la cache (usa el directorio interno). **Sabe si tiene un hit o un miss.**
 - a. Si tiene un **hit**, le manda el **Memory Read** a la cache en vez de enviarselo a la memoria RAM. La cache responde el dato por el BUS de data y el procesador lo lee. **Fin.**
 - b.
 1. Si tiene un **miss**, le manda el **Memory Read** a la RAM. La memoria RAM se toma su tiempo para acceder al dato y

luego mete el dato en el BUS de datos.

2. Además, el controlador de cache le dice a la memoria cache que un dato va a ser escrito (seteando la linea **Memory Write** en su BUS de control) y le manda la misma data que devolvió la RAM en el punto anterior. Este dato lo guarda para honrar el principio de vecindad temporal.
3. Luego, el controlador de cache actualiza la dirección en su directorio de cache.
4. Por último, el mismo controlador de caché le devuelve el dato al procesador.

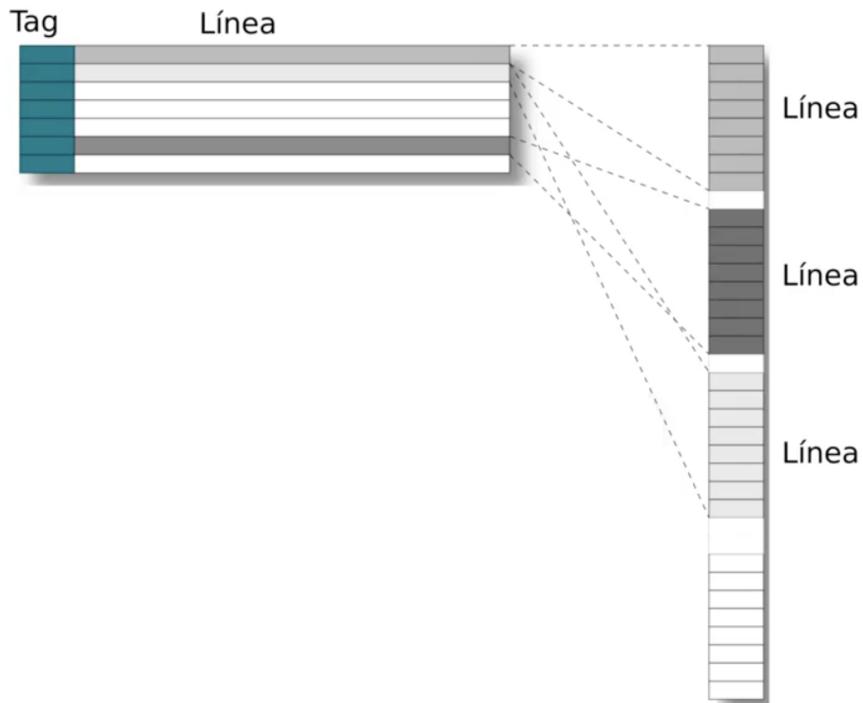
Organización de un cache

Líneas

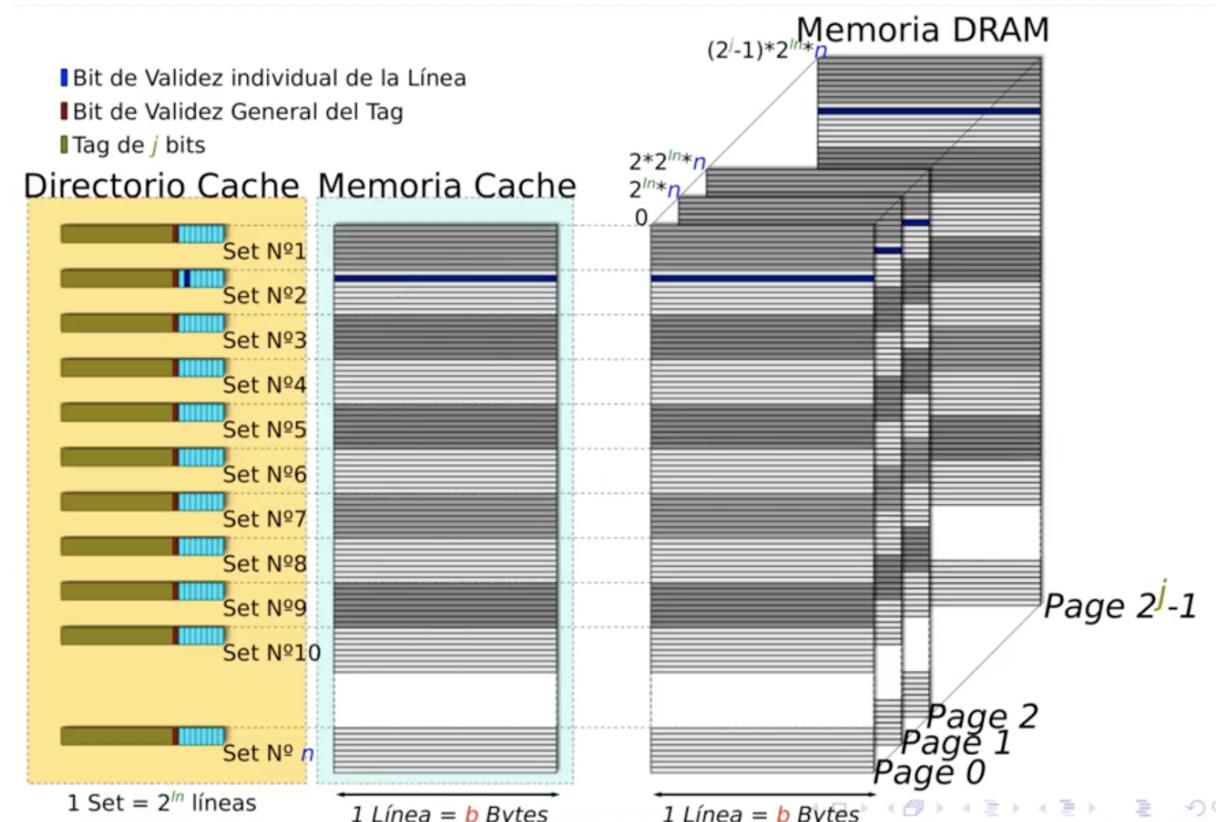
El controlador caché no ve a la memoria principal como una sucesión de bytes, sino como una sucesión de líneas. **La mínima unidad de memoria para un controlador cache es la línea.** La línea está compuesta de una cantidad b de bytes. **Esto está pensado así para aprovechar el principio de vecindad espacial.** El controlador de caché se trae una línea completa en vez de un único byte.

Tag

A cada línea dentro del caché se le asigna una etiqueta (tag). El tag indica la posición de memoria, en la DRAM, en la cual comienza la línea. Dentro del cache las líneas no necesariamente están en el mismo orden en el que aparecen en memoria.



Sistema Cache de Mapeo Directo



- La DRAM se divide en bloques del mismo tamaño de la cache, los cuales se referencian con los bits más significativos

de la dirección.

- Cada bloque se divide en sets.
- Cada set se divide en líneas. Generalmente 8 líneas.
- El directorio cache almacena, para cada set, el número de bloque (tag) que está guardado en la cache. Además tiene un bit de validez general del tag y un bit de validez individual de la línea.

Funcionamiento:

1. Particiono la dirección.

						Número de banco										Número de set				Número de línea	Offset										
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

2. Accedo al número de set en el Directorio Cache.

3. Comparo el número de banco en la dirección con el tag almacenado en la entrada del directorio cache.

- Si el tag del directorio cache NO coincide con el número de banco de la dirección, miss.

4. Si el bit de validez general del tag está apagado, miss.

5. Accedo al bit de validez individual de la línea.

- a. Si el bit de validez de la línea está apagado, miss.

6. Devuelvo hit.

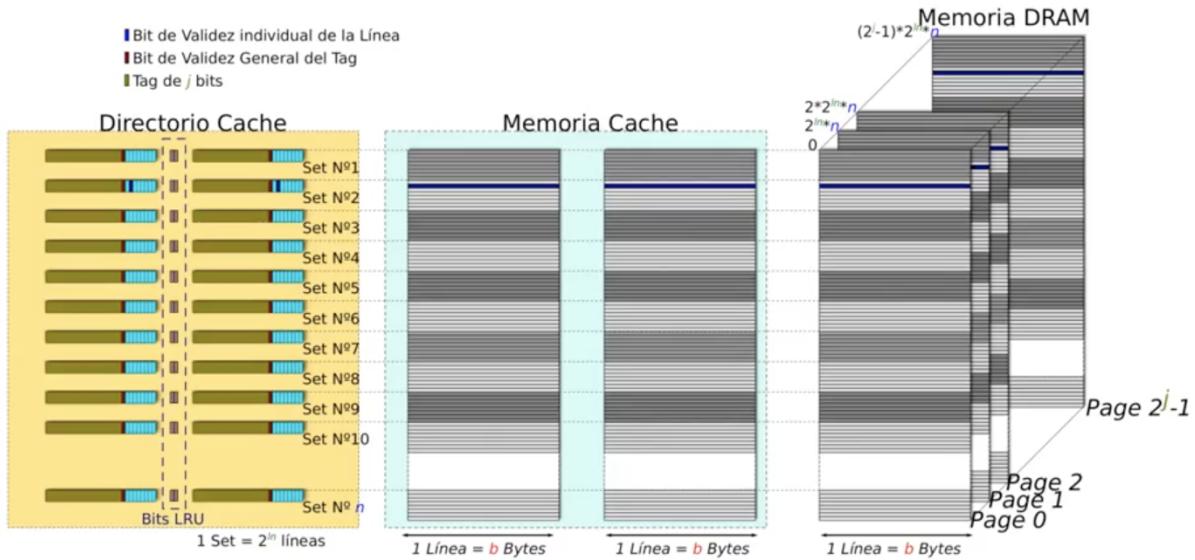
Problema: El mapeo directo es el menos eficiente de todos.

Para que sea eficiente, todas las líneas del set deben ser del mismo banco.

Solución: Sistema de Cache asociativo de n vías

Sistema Cache Asociativo de 2 Vías

Este es el que se usa en la práctica.



Ahora la cache se divide en 2 vías o bancos. Ahora con dos vías puedo tener el mismo número de set de dos bancos de DRAM distintos. **Cuantas más vías tengo, más sets con el mismo número y distinto banco puedo tener.** La eficiencia de esto es muy alta. **Con un cache asociativo de entre 4 y 8 vías se logran eficiencias superiores al 90%.**

Los bits LRU sirven para marcar la vía menos recientemente usada. Sirve para saber a cuál de las dos vías desalojar en caso que quiera almacenar un set de un tercer banco. De nuevo, esto sigue el principio de vecindad temporal.

Coherencia de un cache

¿Qué ocurre durante las escrituras?

Cuando una variable está alojada en la cache, también está alojada en alguna dirección de la DRAM. Idealmente, ambas copias de la variable tendrían que mantener el mismo valor. Esto es lo que se entiende por coherencia. Cuando un procesador modifica la variable y escribe en un sólo lugar, se dice que se perdió la condición coherencia de esa variable. Hay que definir qué política de escritura vamos a utilizar.

Políticas de escritura

Write through:

El procesador va y escribe en la DRAM, al costo de tiempo que sea. Luego, el controlador cache, una vez que esa escritura se efectuó, refresca el dato en el caché para mantenerlo actualizado. **Con esto la coherencia es perfecta, pero la desventaja ahora es que el tiempo de escritura me queda siempre penalizado por la demora de la DRAM. O sea, para las escrituras, la cache no cumple su cometido. Es lo mismo que no tenerlo.**

Write through buffered:

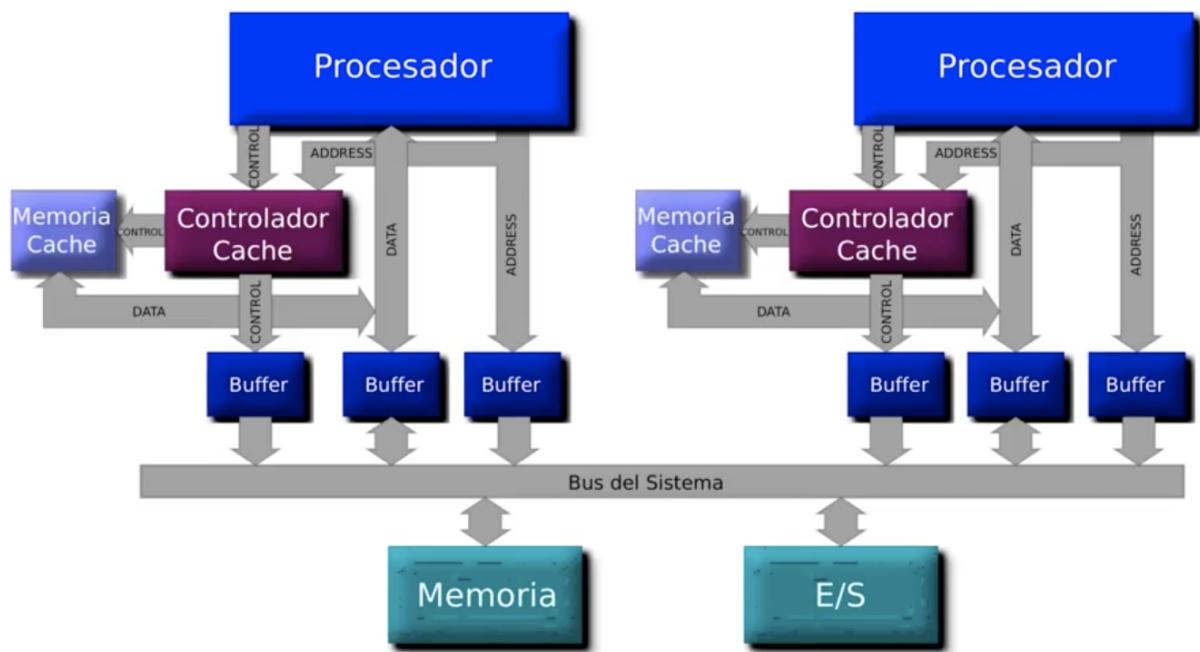
Es write through mejorado con un buffer. Cuando el procesador quiere escribir, en realidad escribe en el cache. Luego, el controlador cache actualiza el dato en la DRAM más tarde. El buffer en el controlador de cache se utiliza para encolar las operaciones de escritura a memoria. **Este método no penaliza tanto las escrituras, salvo que el procesador se ponga a escribir demasiado frecuentemente.**

Write back o copy back:

El procesador escribe en la cache. El controlador de cache marca línea escrita como sucia (dirty). La cache queda completamente incoherente de la copia en DRAM. Luego, cuando la línea tiene que ser desalojada, va y copia su valor en la DRAM. **Esta política no mantiene coherencia pero es muy performante ya que el procesador siempre escribe en la cache.**

Coherencia en sistemas SMP

En un sistema multi-procesador, la política de write back es, en principio, inviable. Uno quiere mantener la coherencia lo más posible, así que utilizaría write thorough o write through buffered.



SMP = Symmetric Multi-Processing

Se dice simétrico porque los procesador son iguales, son simétricos.

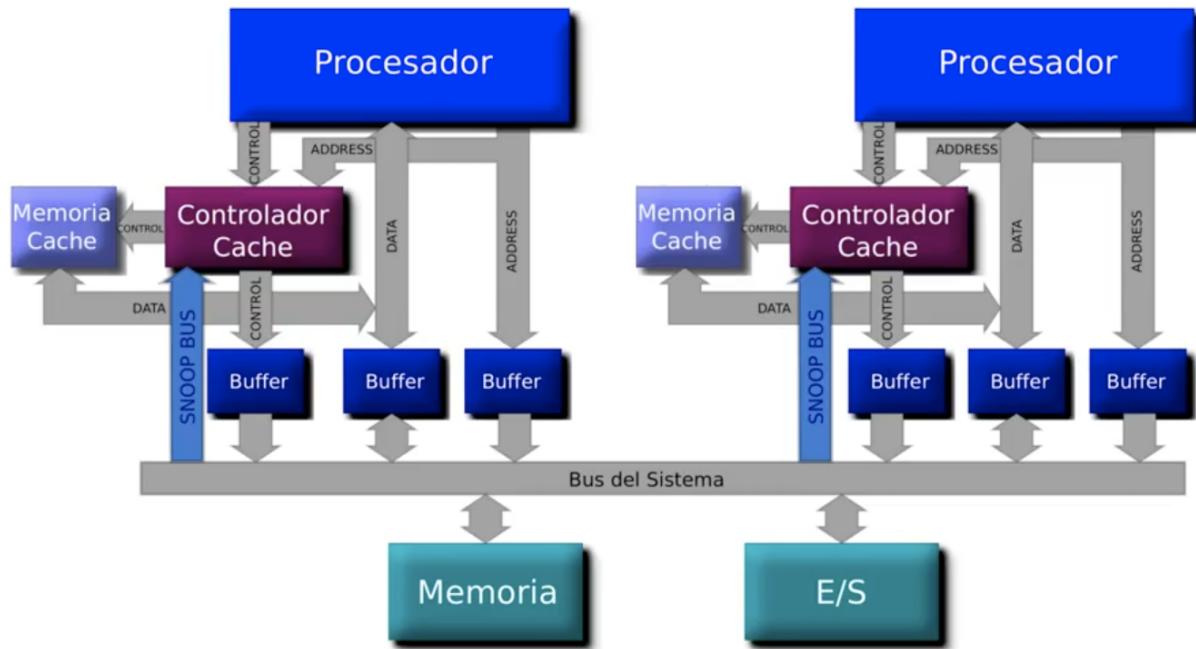
Escenario:

- Tenemos dos thread de un mismo proceso ejecutando en los dos procesadores con las mismas variables y justo tienen la misma variable cada uno en su cache.
- Uno de los dos thread modifica la variable en cache, con lo cual, el dato queda incoherente para la cache del otro procesador y para la DRAM.
- **Problema: Ni la DRAM ni la otra cache se enteran que la variable se modificó.**

Independientemente de la política que utilicemos, tenemos que restaurar la coherencia lo más rápido posible. Lo más difícil es que el otro procesador se entere del cambio.

Solución: Agregamos el Snoop BUS ("BUS de espiar").

Snoop BUS



El procesador 1 escribió una variable. La variable se refrescó también en la DRAM (asumiendo que utilizamos write through o write through buffered). Esto quiere decir que por el BUS del sistema viajó la variable. Con el Snoop BUS, los controladores de cache espían el BUS de address y el BUS de control dentro del BUS del sistema para saber qué hacen los demás. Entonces, cuando un procesador detecta una dirección que se está escribiendo por otro procesador, marca la dirección como inválida en su memoria cache, así la próxima vez la va a buscar a la DRAM. **El Snoop BUS resuelve el problema de la coherencia.**



El Snoop BUS no es BUS entre los controladores. No une a los controladores. Une a cada controlador con el BUS de address del sistema, pero en sentido inverso.

Siguiente motivación:

Si bien el Snoop BUS resuelve el problema de coherencia, nos gustaría hacer optimizaciones. **Queremos ver si existe alguna forma de usar Copy Back, ya que es la mejor política de escritura.** Si bien podríamos usar Write Through Buffered siempre, y utilizar siempre la cache, de todas formas estaríamos utilizando el Bus del Sistema todo el tiempo cuando a no es necesario. En el escenario que vimos hasta ahora, copy back no es factible, ya que depende de que la memoria DRAM esté siempre luego de escribir. Deberíamos buscar una forma de utilizar copy back todo lo que se pueda, y cuando no queda más remedio, utilizar write through o write through buffered.

- Cuando un procesador escribe un dato que está en otra cache, copy back no sirve.
- Pero cuando sólo ese procesador tiene esa variable, no hace falta actualizarlo en la DRAM inmediatamente (para qué la va a estar actualizando en la DRAM del sistema si no la tiene nadie más?).
- Si el procesador pudiera saber si alguien más tiene esa variable, podría hacer copy back mientras solo él la utilice.

Protocolo MESI



M.E.S.I. = Modified, Exclusive, Shared, Invalid

Son los cuatro estados que puede tomar cada línea de cache en un controlador.

Modified :

Esta es la única copia de la línea en una caché pero la modifiqué. La tengo dirty. **Esto me permite hacer copy back.**

En este estado tengo la certeza de que soy el único poseedor de la variable.

Exclusive :

Esta es la única copia de la línea en una caché y además está limpia (sin modificar). **En este estado tengo la certeza de que soy el único poseedor de la variable.**

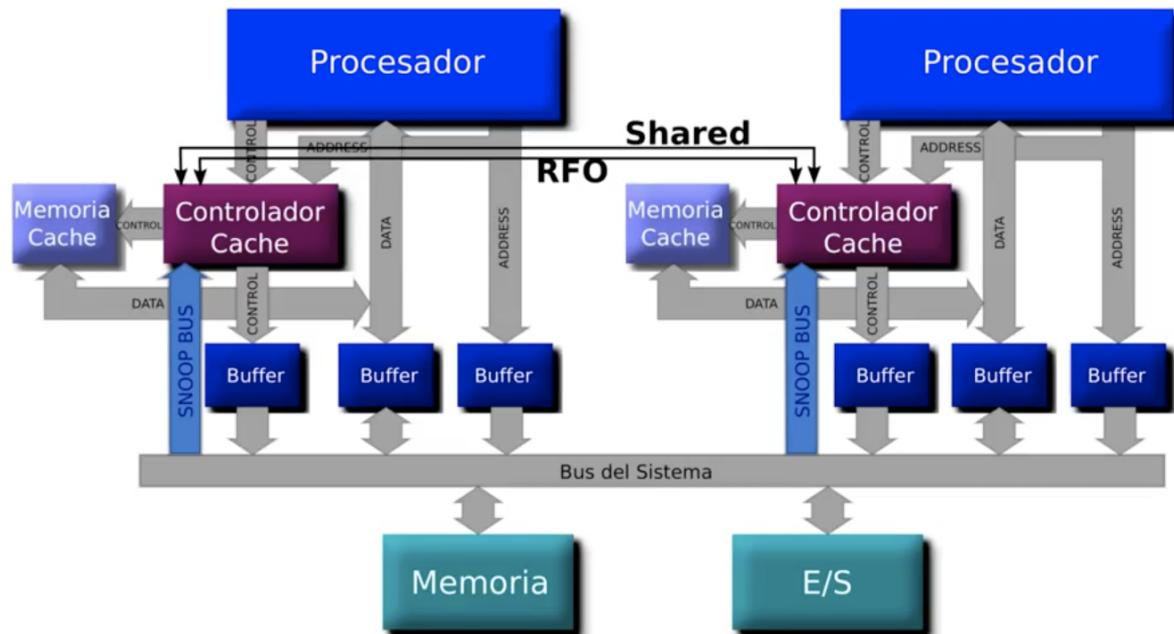
Shared :

No soy el único que la tiene. **Puede** estar almacenada en los caches de otros procesadores. **No puedo hacer copy back. Hago write through y que los demás utilicen el Snoop BUS.**

Invalid :

La línea de cache no es válida. Es basura.

Coherencia en sistemas SMP con MESI



Se agregan las líneas Shared y RFO (Request For Ownership).

Operación del Protocolo MESI

- Todas las transacciones a líneas que un procesador quiere ejecutar (ya sea una lectura o escritura de línea) se

resuelven en el cache, excepto si la línea en el cache está marcada como `Invalid`.

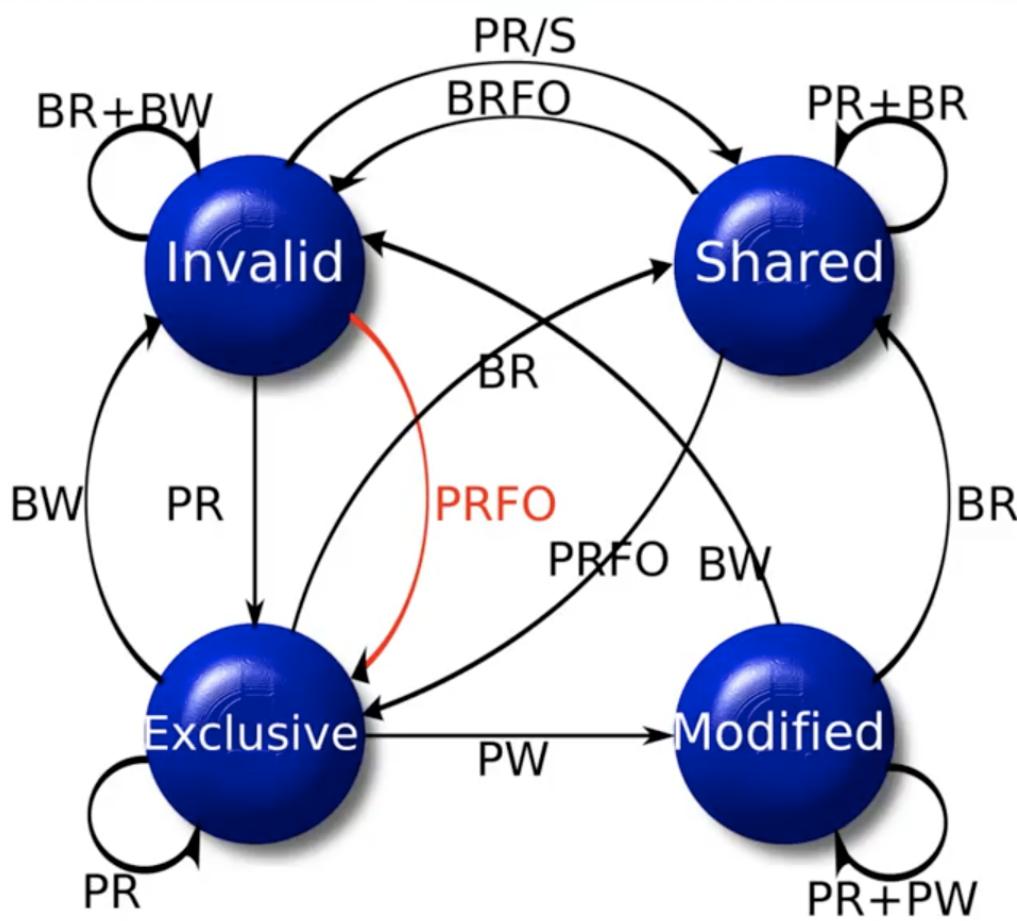
- Las líneas marcadas como `Invalid` se buscan en DRAM.
- El controlador cache que posee líneas en estado `Exclusive`, monitorea siempre por el Snoop BUS cada transacción en la DRAM.
 - Si detecta un acceso a una línea que tiene almacenada en su cache, el controlador owner cambia su estado a `Shared` y activa la línea Shared para enviar un broadcast al resto. Todos se enteran que alguien tiene esa línea shared.
- Una línea que está en estado `Shared` o en estado `Exclusive` puede ser descartada y pasar a inválida en cualquier momento sin avisarle a nadie.
- Una línea `Modified` también puede ser descartada, sólo que en este caso se requiere actualizar previamente la DRAM.
- Una línea que está `Modified` o `Exclusive` se puede escribir desde la CPU en cualquier momento. En caso de que esté `Exclusive`, pasa a `Modified`.
- En el caso en que se necesite escribir en una línea cuyo estado sea `Shared`, el protocolo indica que todas las demás caches que tienen esa línea la invaliden previamente. Para ello se emplea una operación broadcast denominada **Request For Ownership**.
- Un cache que tiene una línea en estado `Modified` y detecta por el Snoop BUS que alguien la lee, también debe insertar el dato contenido por la línea, ya que está incoherente con la DRAM. Para ello realiza lo siguiente:
 - Activa la línea RFO, indicándole a los demás que ese dato está incoherente.
 - Escribe en la DRAM el valor actual de la línea. El lector copiará ese valor correcto a su cache cuando aparece en el BUS de datos.
 - Ambos pondrán la línea en `Shared`.

- Una línea en estado `Shared` pasa a `Invalid` cuando se recibe un RFO.

Read For Ownership

- El protocolo de coherencia combina una escritura de una línea con un broadcast de invalidación al resto de los controladores.
- Es enviada por un controlador cache si intenta escribir una línea que está en estado `Shared` o está en estado `Invalid`.
- Como resultado, el resto de los cache que tengan esa línea la tienen que invalidar.
- Desde el punto de vista del controlador cache, es una lectura de línea cacheada con toma de BUS del sistema para escribir el contenido de la línea en la memoria DRAM.

Protocolo MESI - Diagrama de estados

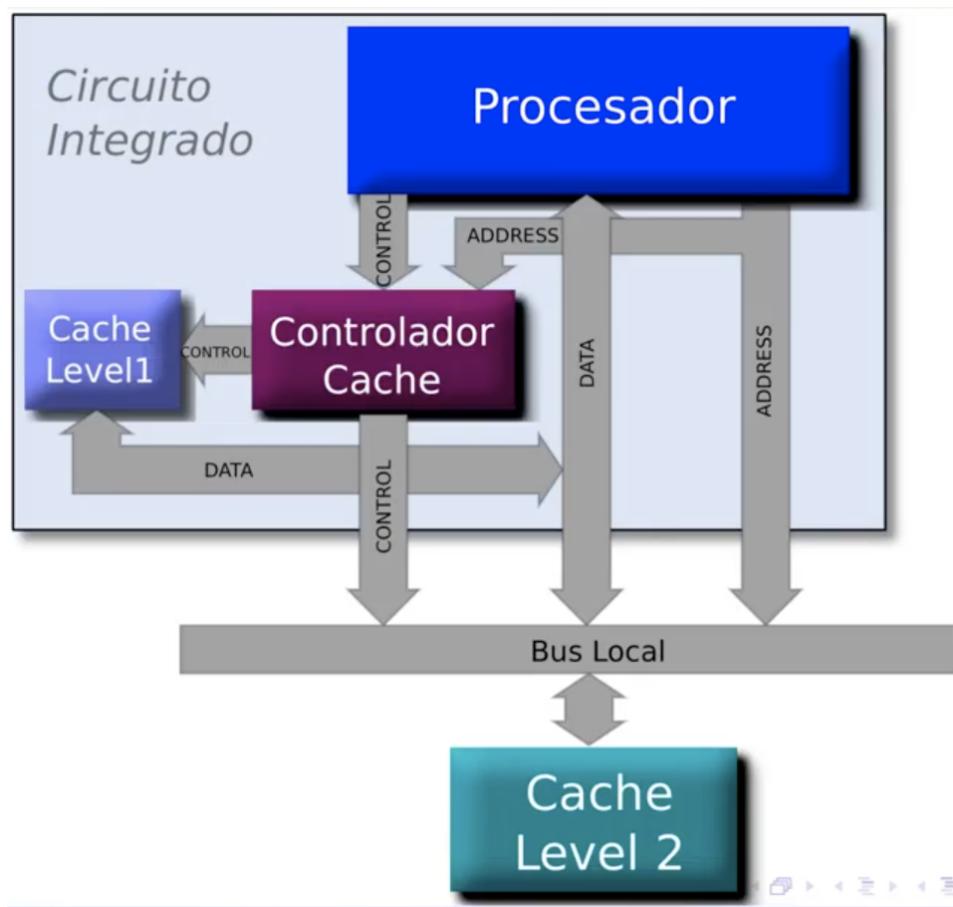


- PR: Processor Read (yo)
- BR: Bus Read (otro controlador lee, detecto un read en el bus)

Arquitecturas de cache avanzadas

Cache Multinivel

En algún momento de la evolución de los procesadores, la tecnología de integración alcanzó scalings que permitieron meter controladores cache + cache level 1 en un chip, y dejar una cache level 2 afuera del chip.



La cache level 1 la metieron adentro del chip del procesador para incrementarle la frecuencia. La cache level 2 estaba afuera del chip y su frecuencia tenía que ser la del estándar PCI. Adentro del chip, la frecuencia se podía duplicar.

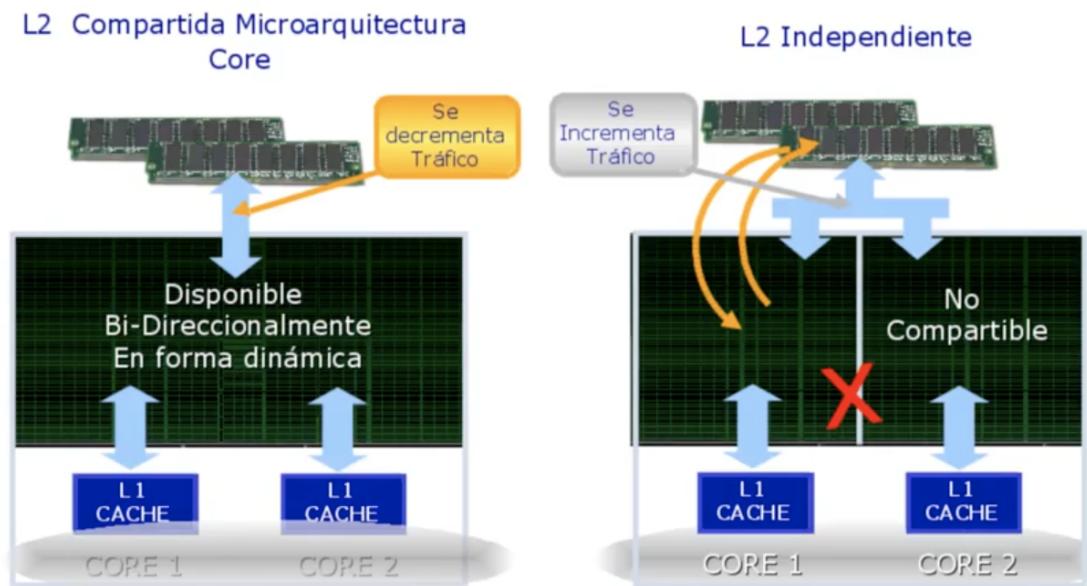
2do. Nivel On chip

- Mete el cache level 2 también dentro del chip.
- Cache level 1 era como una arquitectura Harvard (separaba datos de instrucciones).

3er. Nivel On chip

- Arranca con Intel Netburst.
- Mete el cache level 3 también dentro del chip.

Smart Cache



Paralelismo a Nivel de Instrucción

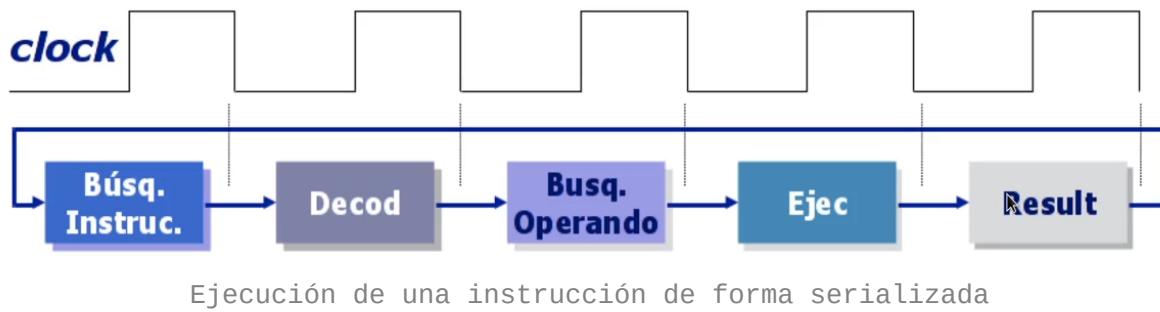
Pipeline

Máquina de estados elemental

Una instrucción en un computador se ejecuta en etapas. Según el modelo de Von Neumann, las etapas son: fetch, decode y execute. Hoy en día cada instrucción se puede desdoblar en más etapas.

Antes del 80286, la mayoría de los microprocesadores ejecutaban cada etapa en serie. Es decir, en el primer ciclo

de clock, se ejecutaba una etapa de una instrucción, luego se ejecutaba la siguiente etapa de la misma instrucción, y así sucesivamente hasta haber ejecutado la última etapa de esa instrucción, para luego empezar ejecutando la primer etapa de la siguiente instrucción.



Problema: Para ejecutar la siguiente instrucción hay que esperar a que la primer instrucción termine de ejecutarse completamente. No podríamos ejecutar dos instrucciones a la vez?

Pipeline

Arquitectura que permite crear el **efecto** de superponer en el tiempo la ejecución de varias instrucciones a la vez con el objetivo de mejorar la performance.

Requiere muy poco hardware adicional.

Idea: Mientras se ejecu

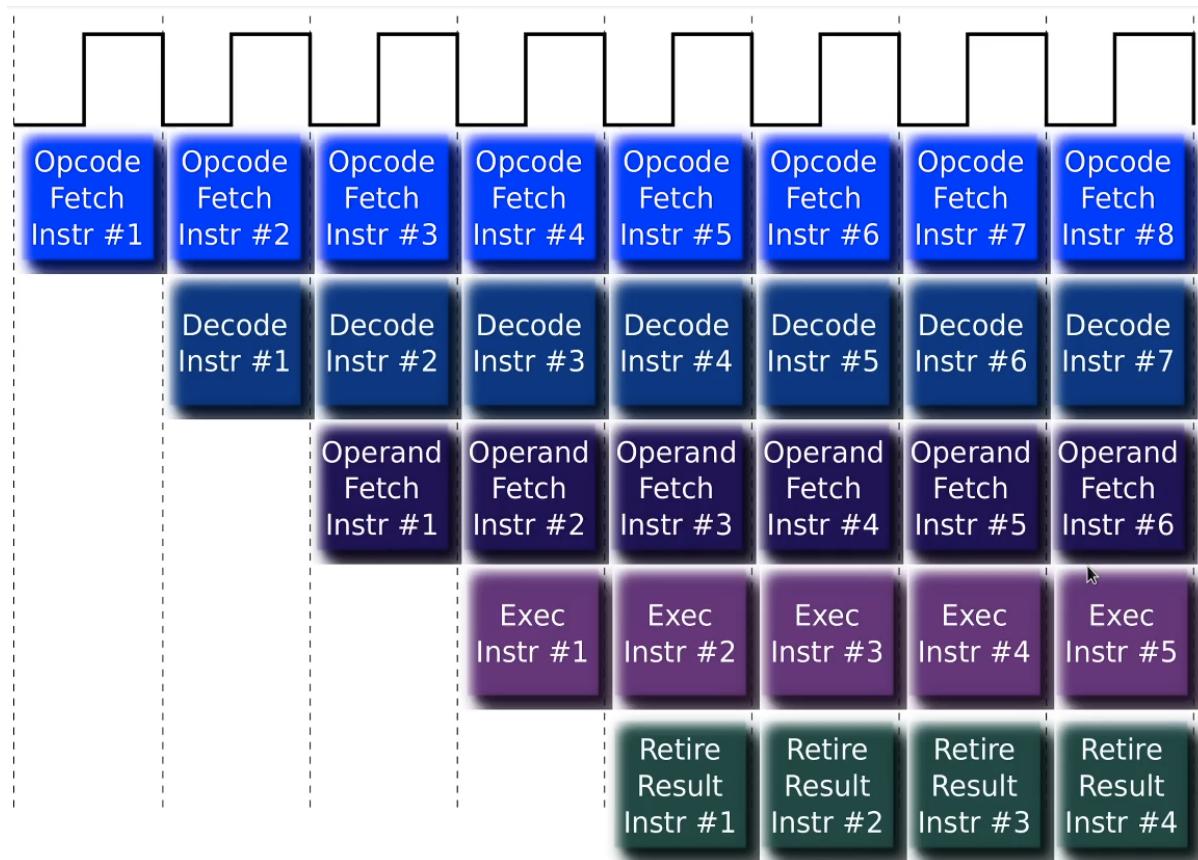
Ejemplo:

5 stages (etapas):

1. Opcode fetch
2. Decode
3. Operand fetch
4. Execution

5. Retire result

Pipeline **ideal** para esas 5 etapas:



Primer ciclo: Opcode fetch instr #1.

Segundo ciclo: Opcode fetch instr #2 + Decode instr #1.

Tercer ciclo: Opcode fetch instr #3 + Decode instr #2 + Operand fetch instr #1.

Cuarto ciclo: Opcode fetch instr #4 + Decode instr #3 + Operand fetch instr #2 + Exec instr #1.

Quinto ciclo: Opcode fetch instr #5 + Decode instr #4 + Operand fetch instr #3 + Exec instr #2 + Retire result instr #1. **Recién acá se libera el primer resultado (el de la instrucción #1).**

Conclusiones:

- El pipeline demora tantos ciclos de clock en llegar al primer resultado como etapas tenga.

- En un pipeline de 5 etapas, en el caso de que cada etapa consuma solamente 1 ciclo de clock, obtenemos un resultado de instrucción por cada ciclo de clock a partir de la quinta etapa.
- Este escenario es teórico y no responde a la situación real.
- Agregar etapas parecía mejorar la performance del pipeline.

Procesador / uArquitectura	Etapas	Procesador / uArquitectura	Etapas
ARM7TDMI(-S)	3	ARM7EJ-S	5
ARM810	5	ARM9TDMI	5
ARM1020E	6	XScale PXA210/PXA250	7
ARM1136J(F)-S	8	ARM1156T2(F)-S	9
ARM Cortex-A5	8	ARM Cortex-A8	13
AVR32 AP7	7	AVR32 UC3	3
DLX	5	Intel P5 (Pentium)	5
Intel P6 (Pentium Pro)	14	Intel P6 (Pentium III)	10
Intel NetBurst (Willamette)	20	Intel NetBurst (Northwood)	20
Intel NetBurst (Prescott)	31	Intel NetBurst (Cedar Mill)	31
Intel Core	14	Intel Atom	16
LatticeMico32	6	R4000	8
StrongARM SA-110	5	SuperH SH2	5
SuperH SH2A	5	UltraSPARC	9
UltraSPARC T1	6	UltraSPARC T2	8
WinChip	4	LC2200 32 bit	5

Hazards (obstáculos)

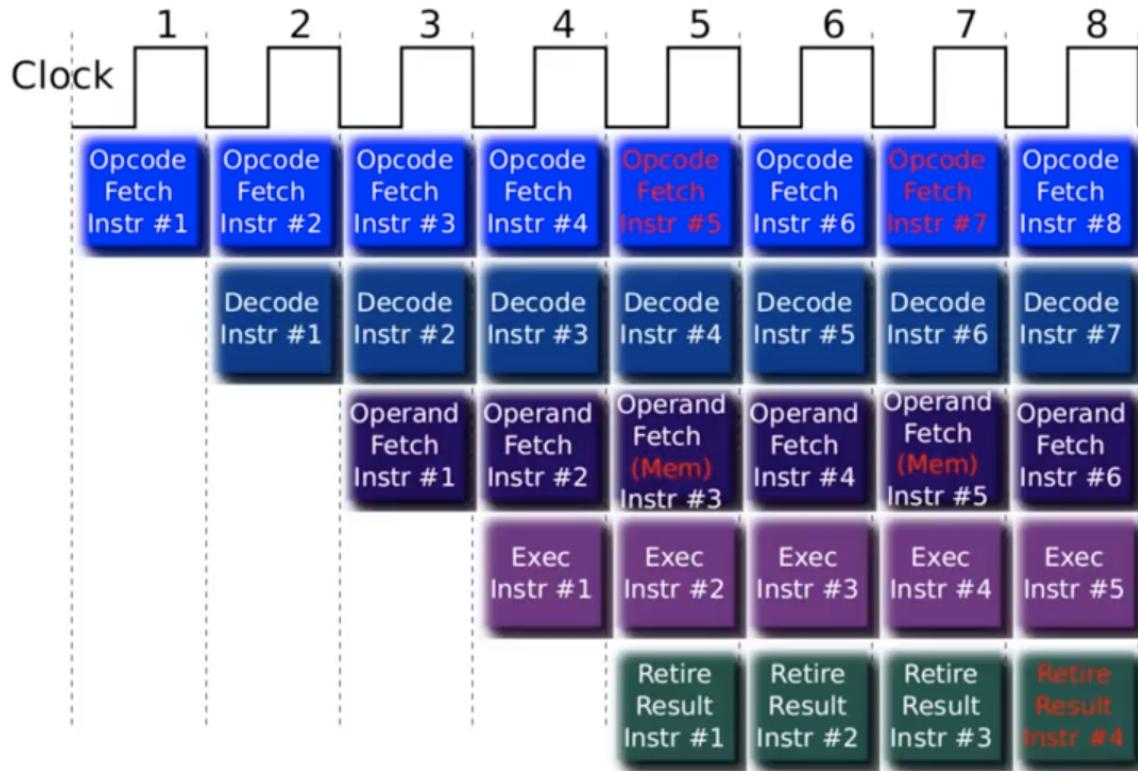
Cuando hay un obstáculo se denomina **pipeline stall (congestión del pipeline)**.

Obstáculos estructurales:

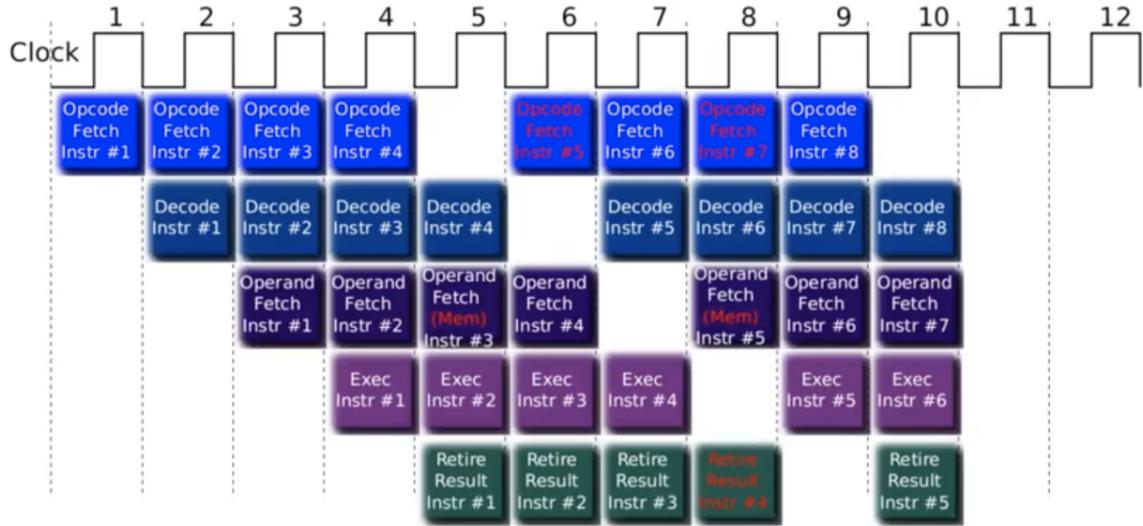
- Una etapa no está suficientemente atomizada, tiene muchas cosas que hacer y no se puede completar en un ciclo de clock (tarda mucho).
- Si dos instrucciones que van a utilizar esta etapa están más o menos próximas en el tiempo, o están a una distancia temporal mayor a la que la etapa consume para procesar, van a caer en un conflicto de recursos para su

ejecución. La que está primera en el pipeline va a tomar la etapa y cuando llegue la que venía en segundo lugar y quiera esa etapa del pipeline no va a poder.

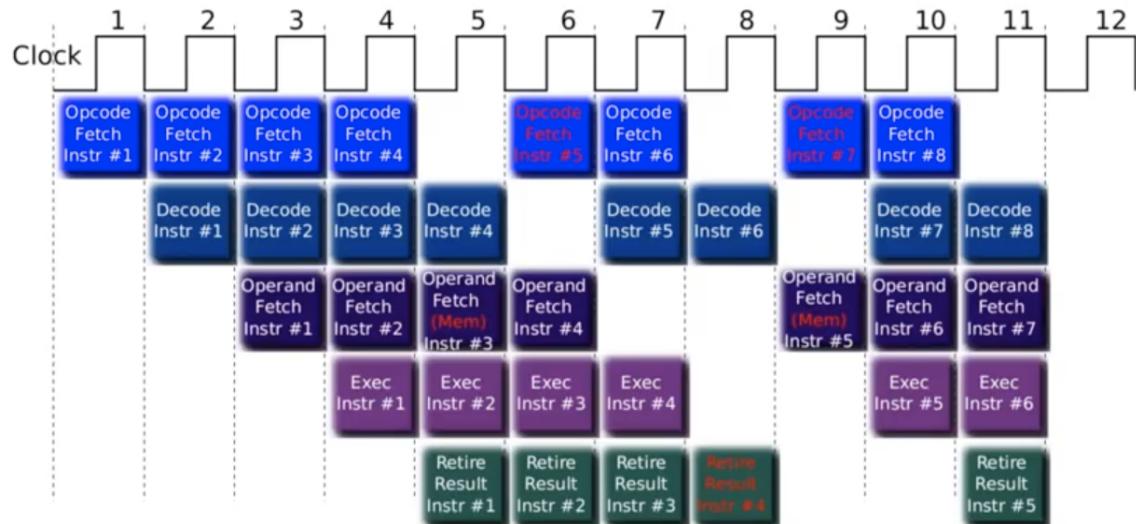
Ejemplo:



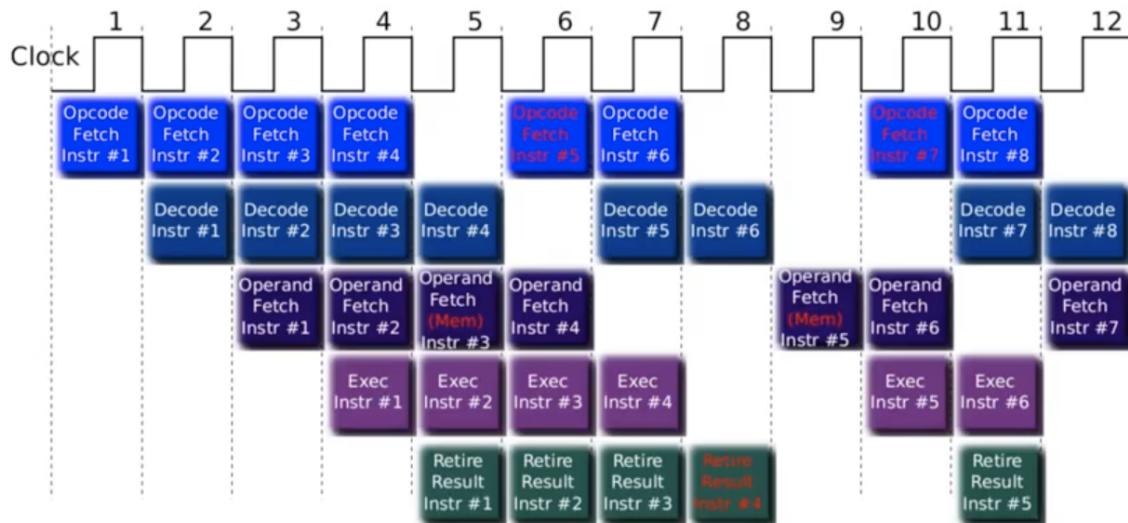
En el ciclo 5, la etapa de Opcode Fetch de la instrucción #5 y la etapa de Operand Fetch de la instrucción #3 quieren acceder a memoria a la vez. La solución es elegir alguna de las dos y demorar la otra, corriendo todo el pipeline. Un criterio posible es priorizar la instrucción que este en la etapa más avanzada. En este caso, la instrucción #3, la cual ya está en la etapa de Operand Fetch.



Luego, en el ciclo 8, también hay un conflicto. Lo soluciono haciendo lo mismo (priorizando la instrucción #4, que la más avanzada).



Luego, en el ciclo 9, también hay un conflicto. Lo soluciono haciendo lo mismo (priorizando la instrucción #5, que la más avanzada).



Posibles soluciones:

- **Cualquier solución cuesta hardware**
- En el caso de accesos a memoria:
 - Podemos desdoblar el cache L1 en cache de datos y cache de instrucciones.
 - Emplear buffers de instrucciones implementados como pequeñas colas FIFO.
 - Ensanchamiento de los buses más allá de los anchos de palabra del procesador. Esto permite, por ejemplo, hacer que un opcode fetch se resuelva en un ciclo de clock. Si ensancho los buses, en un ciclo de clock leo mucha más información. Es una forma de fatchear muchas instrucciones de una, o traerse muchos operandos de una. Tiene sentido por el principio de vecindad.
- Aumentar la capacidad del pipeline. Hacer etapas más simples, capaces de resolver en un ciclo de clock su tarea.

Obstáculos de datos:

- Una instrucción pide un dato antes de que éste esté disponible por efecto de la secuencia lógica que contiene el programa.

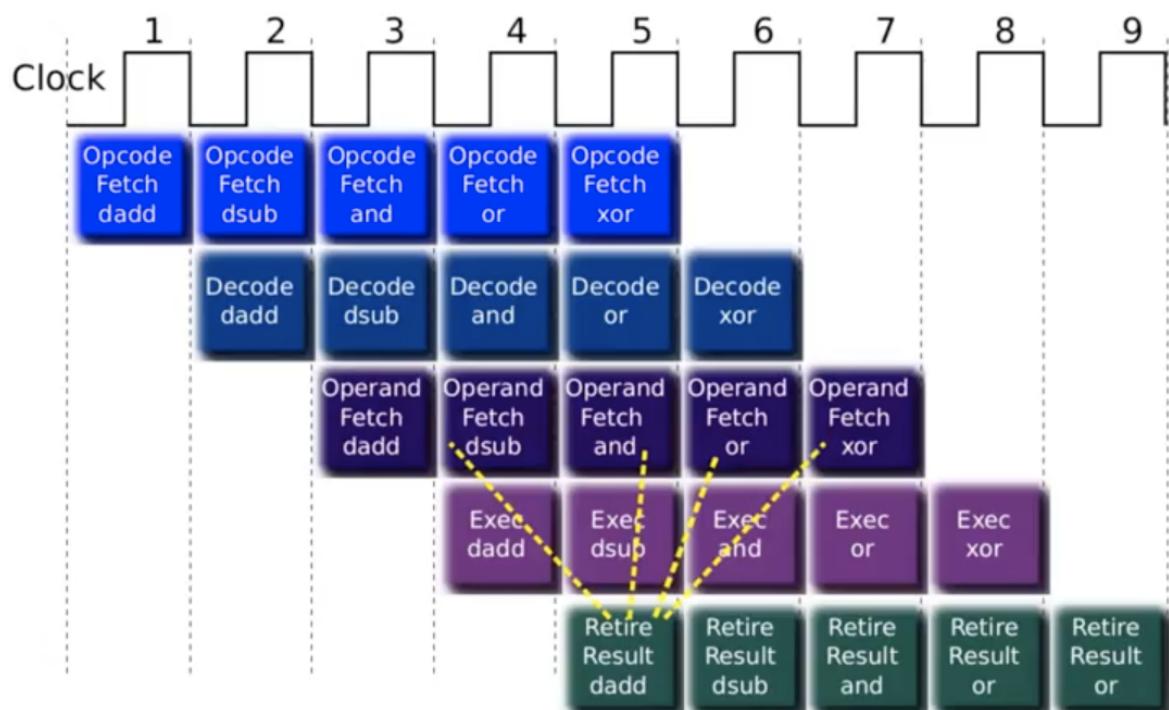
Ejemplo:

```

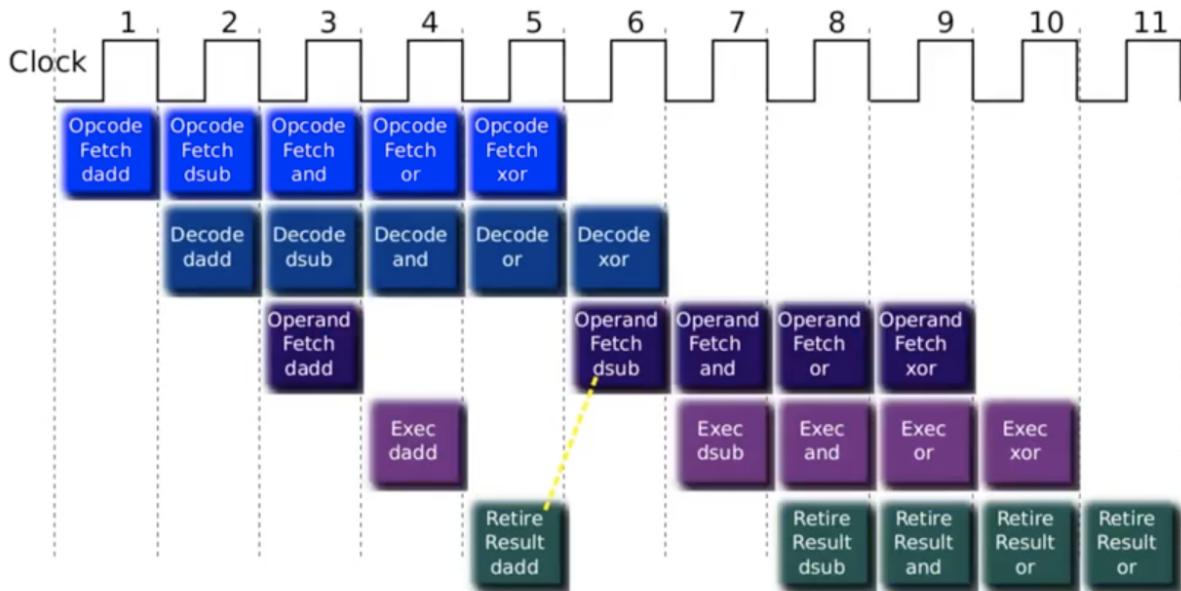
1  dadd   R1, R2, R3
2  dsub   R4, R1, R5
3  and    R6, R1, R7
4  or     R8, R1, R9
5  xor    R10, R1, R11

```

La instrucción dsub depende del resultado de la instrucción dadd.



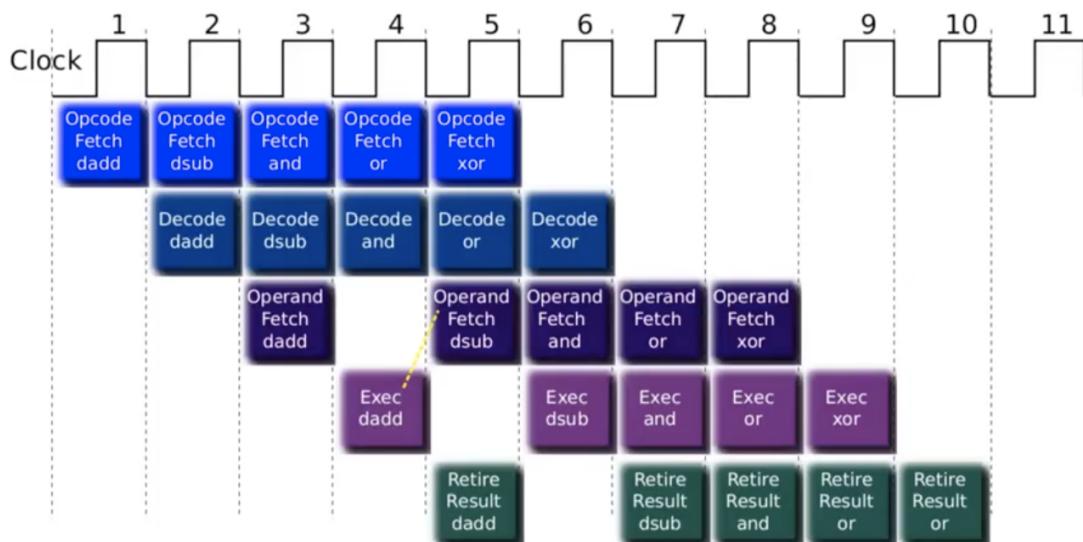
En el quinto ciclo, se genera el resultado de la suma, pero dicho resultado es requerido por `dsub`, `and`, `or` y `xor`. Las instrucciones `dsub` y `and` no pueden conseguir a tiempo el resultado en R1 para poder usarlo como operando.



Posibles soluciones:

- **Forwarding:**

Retroalimentando la ALU con la salida en el mismo ciclo que termina de ejecutarse la operación aritmética, así, en el ciclo 5, la ALU ya tiene el operando correcto sin haber pasado por R1. No soluciona el problema del todo, pero se ahorra un ciclo.

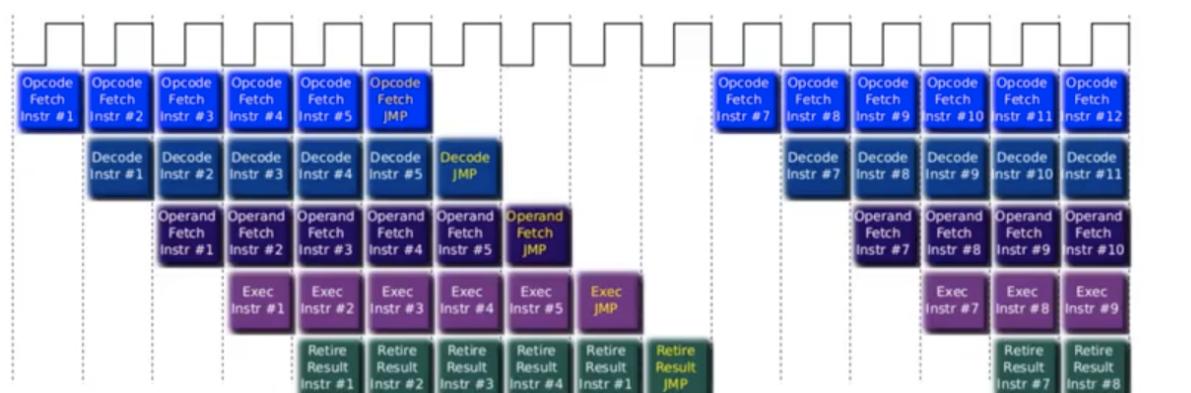


No siempre se puede hacer forwarding. Si la primer operación no pasa por la ALU no se puede hacer forward.

1	Id	R1, 0(R2)
2	dsub	R4, R1, R5
3	and	R6, R1, R7
4	or	R8, R1, R9
5	xor	R10, R1, R11

Obstáculos de control:

- Son causados por saltos.
- Un branch es lo peor que le puede pasar a un pipeline.
- El branch hace que todo lo que tenias preprocesado deba descartarse. Esto se conoce como **branch penalty**.



Unidades de Predicción de saltos

Necesidad de predicción de saltos

- Cuanto más profundo es un pipeline, más grande es la penalización de un branch. Ejemplo: para un pipeline de 5 etapas de 1 ciclo de clock cada una, la penalización es de 4 ciclos de clock.
- A medida que aumenta la complejidad del procesador, también es lógico que las etapas del pipeline aumenten (sencillamente porque el procesador es más complicado).
- **En la práctica los Branch Predictors se implementan en la etapa de fetch.**

Predecir un salto es poder predecir el valor del program counter. Predecir un salto significa empezar a fetcpear de una manera anticipada, para reducir el hueco en el pipeline. Hay situaciones en las que es imposible predecir.

Asumir non-taken

Asume por defecto que el salto nunca se toma. Entonces el pipeline sigue fetcpeando las instrucciones siguientes a la instrucción del branch. Es útil cuando el salto es hacia adelante. Por ejemplo:

instrucción *branch*
instrucción sucesora secuencial.
instrucción destino para *branch taken*.

Asumir taken

Asume por defecto que el salto siempre se toma. Entonces el pipeline empieza a fetcpear las instrucciones que están en la dirección de salto. Es útil en un loop donde se salta hacia atrás. En general las iteraciones se hacen muchas veces, con lo cual los saltos son más probables en esos casos. Por ejemplo:

instrucción destino para *branch taken*.
grupo de instrucciones a ejecutar iterativamente.
instrucción *branch*

Delayed branch

- Se introdujo en los primeros procesadores RISC.
- Tenemos N slots llamados Delay Slots.
- Las N instrucciones que vienen después de la instrucción de branch se ejecutan siempre, sin importar del camino tomado por el branch.

- Luego aplicas el resultado según se necesite o no.
- Ejemplo con N=1:

instrucción *branch*

instrucción sucesora secuencial.

instrucción destino para *branch taken*.

Loop unrolling

Es una tarea para el compilador. Consiste en convertir los ciclos en instrucciones secuenciales, sin branches.

Predicción de saltos dinámica

Los métodos anteriores dependían del compilador, del set de instrucciones. El hardware del procesador no hacía ningún análisis de código para analizar la instrucción.

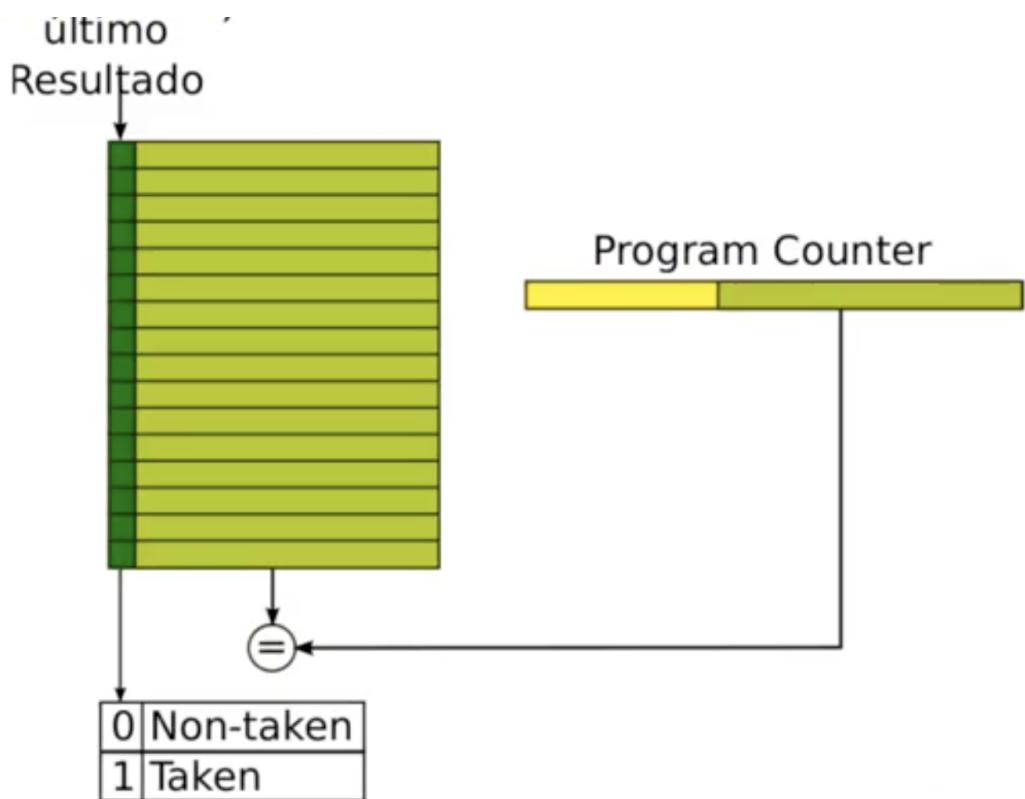
Idea: Que ahora el procesador haga análisis de flujo y tome decisiones acorde a lo que va viendo adelante de la instrucción de salto.

Debemos predecir:

1. Si la instrucción es de salto o no.
2. Si es taken o non-taken.
3. La dirección de salto (if taken).

Branch Prediction Buffer (Last Time Predictor)

- Tabla en caché que sirve para saber si el salto va a ser taken o non-taken.
- Se guarda como un bit extra en la BTB.



- Funciona bien para patrones tipo:
TTTTTTTTTTTTNNNNNNNNNN.
- **El problema:** Funciona mal para patrones tipo
TNTNTNTNTNTNTNTNTNT (0% accuracy en ese caso).
- **Solución:** Armar una máquina de estados con dos bits.
- **Idea:** No vamos a cambiar nuestra predicción al primer desacuerdo, la vamos a cambiar al segundo.

Si estamos en el estado de "Predice TAKEN (fuerte)" y la branch fue non-taken, pasamos al estado de "Predice TAKEN (debil)" en vez de pasar a un estado de "Predice Non-TAKEN".



Branch Target Buffer

- Tabla en caché que sirve para saber la dirección de salto (if taken).
- Cada entrada contiene la dirección de la instrucción de salto y la dirección target resuelta (en vez de los bits taken o non-taken, a diferencia de la Branch Prediction Buffer).
- Es más rápido que el Branch Prediction Buffer (ya que ahora directamente podemos obtener la dirección de salto de esta tabla), pero el criterio sigue siendo el mismo.
- La proxima vez que nos topamos con el branch, la buscamos en la BTB. Si tenemos un hit, usamos la address guardada en la BTB para fetchear instrucciones.
- Si el valor no se encuentra en la BTB, se asume taken.
 - Si el resultado es Non-taken se acepta el delay en el pipeline y no almacena nada en la BTB.

- Si el resultado es taken, ingresa el valor en la BTB.
- Si el valor se encuentra en la BTB, se aplica el campo de la dirección target almacenado.
 - Si el resultado es taken, no hay penalidad. No se guarda en el BTB ningún nuevo valor ya que nos sirve.
 - Si el resultado es non-taken, sigue la máquina de estados de **Branch Prediction Buffer**.

Superscalar

Consiste en paralelizar el pipeline, agregando otro.

Ventaja: Poder ejecutar más de una instrucción por ciclo de clock, en paralelo.

Superscalar de dos vías



El Pentium fue el primer procesador superscalar de la familia Intel.

Problemas

- Los obstáculos estructurales ahora quedan más expuestos que antes. Ahora cada etapa, ademas de lidiar con las otras etapas de su propio pipeline (que pueden acceder en simultáneo a la misma instrucción o dato), también tiene que lidiar con las mismas etapas del pipeline de al lado, que también pueden ir a buscar datos o código. **Por eso los buses son muy importantes. Es importante separar buses de datos y address para minimizar los obstáculos estructurales.**
- Cuantas más vias tenga el pipeline, más concurrencia de memoria.
- Si tengo dos ALUS, puedo ejecutar una instrucción en cada uno. En el caso de que una ALU dependa del resultado de la otra, esto no es posible.
- Una falla en el branch prediction limpia todos los pipelines.

Scheduling Dinámico

El Scheduling de instrucciones es la forma en que el procesador organiza la ejecución de instrucciones.

Hasta antes de superscalar, el scheduling era estático. Ahora, con superscalar, el scheduling es dinámico.

Obstáculos de Datos

Para los obstáculos de Datos con superscalar, no podemos usar forwarding. El estudio de dependencias de datos se debe extender a instrucciones en los diferentes pipelines.

Idea fuerza:

```

1  div.d    F0,F2,F4
2  add.d   F10,F0,F8
3  sub.d   F12,F8,F14

```

- La instrucción 2 (add.d) depende de la instrucción 1 (div.d), que para completarse requiere muchos ciclos de

clock (debido a otros tipos de obstáculos que afectan a la instrucción 1 pero no afectan a la instrucción 2).

- O sea, las instrucciones 2 y 3 no pueden ser ejecutadas hasta que la instrucción 1 termine.
- Esta obstrucción deja a todo el procesador sin uso hasta que la instrucción 1 termine.
- **La instrucción 3 (sub.d) no tiene ninguna dependencia con las instrucciones previas, sin embargo, la instrucción queda esperando.**

Solución: Ejecución fuera de orden. Ejecuto la instrucción 3 igual, sin esperar.

Ejecución Fuera de Orden

Conceptos fundamentales

Idea fuerza

- Enviar las instrucciones a ejecución independientemente del orden que tengan en el código. Con qué criterio?
- Una vez que una instrucción es decodificada, puedo saber si esa instrucción tiene atascos (estructurales, de datos, etc).

Ejecución Fuera de Orden vs En Orden



In Order Dispatch

div.d F0,F2,F4
add.d F6,F0,F8
sub.d F8,F10,F14
mul.d F6,F10,F8

	F	D	O	E	E	E	E	R	W	
div.d	F	D	O					E	R	W
add.d	F	D	O					E	R	W
sub.d	F	D	O					E	R	W
mul.d	F	D						O	E	R

Out Of Order (OOO) Dispatch

div.d F0,F2,F4
add.d F6,F0,F8
sub.d F8,F10,F14
mul.d F6,F10,F8

	F	D	O	E	E	E	E	R	W	
div.d	F	D	O					E	R	W
add.d	F	D	O					E	R	W
sub.d	F	D	O	E	R			WAIT	W	
mul.d	F	D	O	E	R			WAIT	W	

2 Clocks

- Separamos la etapa de resultado en resultado y escritura.
- Si bien vamos a ejecutar fuera de orden, no podemos aplicar los resultados de las operaciones en un orden distinto al que está establecido en el programa porque sencillamente alteraríamos la lógica del programa y eso es inadmisible (por algo las instrucciones están en ese orden).
- Como las instrucciones están atomizadas en distintas etapas, lo que vamos a tratar de hacer es que una parte de ese pipeline trabaje fuera de orden, y otra parte trabajen en orden.
- En general, la etapa de ejecución es la etapa que se hace fuera de orden.
- La etapa de write va en orden para no alterar el comportamiento.

Riesgos

```

1  div.d    F0,F2,F4
2  add.d    F6,F0,F8
3  sub.d    F8,F10,F14
4  mul.d    F6,F10,F8

```

- **Write, After Read (WAR):**

La instrucción 3 escribe F8 y después la instrucción 2 lee F8. **La operación de la instrucción 2 tiene un operando inválido.**

- **Write, After Write (WAW):**

La instrucción 4 escribe F6 y después la instrucción 2 escribe F6. **Me quedo con un valor más viejo de F6 en la instrucción 5.**

- **Read, After Write (RAW):**

La instrucción 2 lee F0 y después la instrucción 1 escribe F0. **Este es el clásico obstáculo de datos que vimos al principio, antes de superscalar. Se soluciona stalleando el pipeline.**

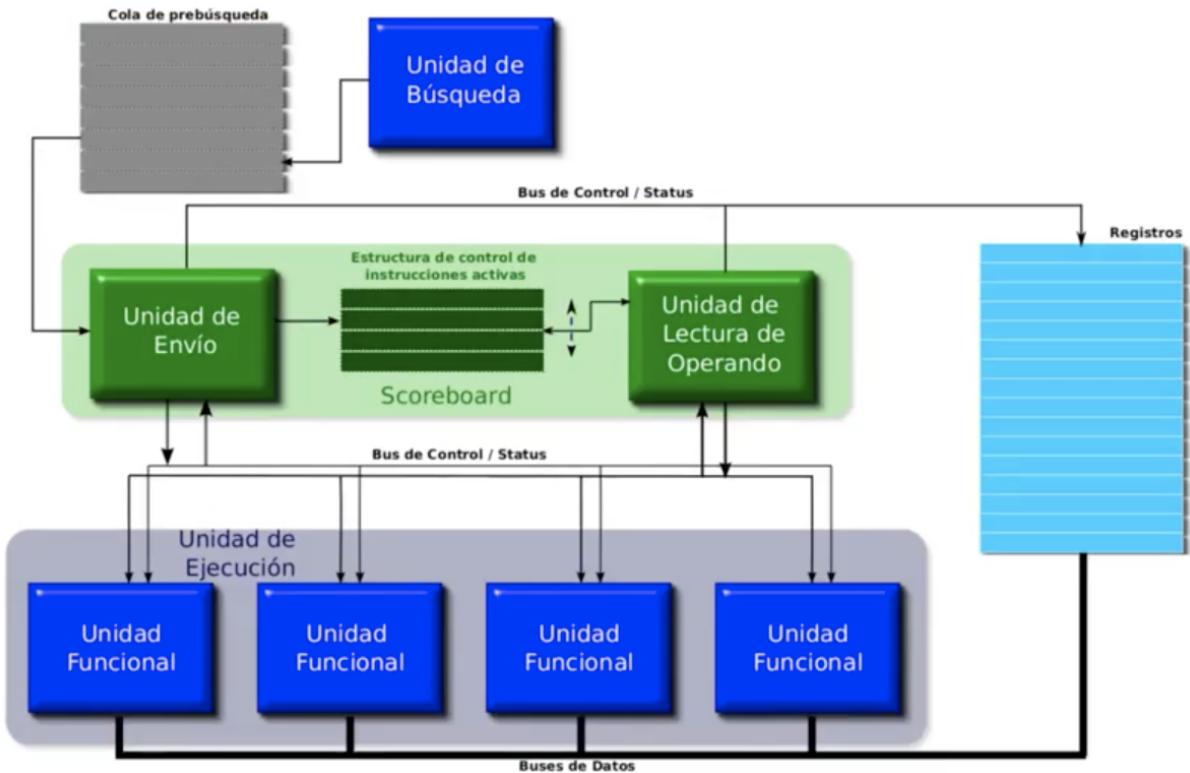
Excepciones imprecisas

- Las operaciones generan excepciones, que a su vez generan interrupciones para resolver las excepciones.
- Si estamos ejecutando fuera de orden una instrucción que genera una excepción, no deberíamos generar la excepción fuera de orden, ya que eso limpiaría el pipeline para saltar a la interrupción.
- Se debe preservar ese escenario.
- Tendríamos que poder "buffear" el estado de esa excepción.

Método prehistórico

Scoreboarding

- Es el primer método que se desarrolló para ejecutar fuera de orden.
- Es el menos sofisticado.
- Se implementó en la CDC 6600 en 1964.
- No se usa.



- Una Unidad de Búsqueda arma una cola de prebúsqueda que contiene las próximas instrucciones.
- La Unidad de Envío toma lo que hay en la cola de prebúsqueda y lo pone en un Scoreboard de capacidad 4.
- La Unidad de Lectura de Operando se fija si los operandos de las instrucciones en el Scoreboard estaban disponibles y en función de eso las despacha a las Unidades Funcionales.

Desventajas:

- Los operandos se leen directamente en los registros, así que no se podía hacer forwarding. No tenía una lectura prolífica de los operandos.

Método actual

Modelo de Tomasulo

- 1967: Tomasulo presentó un trabajo de Scheduling dinámico en la unidad de punto flotante del mainframe IBM 360/91.
- **Trata de minimizar los riesgos RAW.**
- **Neutraliza los riesgos WAR y WAW.**

Tomasulo se preguntó qué necesita un procesador para poder ejecutar fuera de orden?

1. Mantener un "enlace" entre quien produce un dato y quien o quieren lo consumen. La lógica del procesador tiene que tener claro, para cada operando destino, quién lo va a utilizar en las instrucciones posteriores.
2. Mantener en espera las instrucciones que todavía no se puedan ejecutar, es decir, que tengan al menos un operando en espera.
3. Las instrucciones tienen que saber cuándo están disponibles sus operandos.
4. Ejecutar la instrucción cuando todos sus operandos estén "Ready".

Register Alias Table (RAT)

- Soluciona el problema de mantener un enlace entre el productor de un dato y sus consumidores.
- A cada operando, se le asocia un alias y se indica si es válido o no.
 - **Tag:** Alias/renombre del registro.
 - **Valor:** Valor del registro.
 - **Validez:** Dice si el valor es válido o no. Si es inválido significa que otra

	Tag	Valor	Válido
R0		1	
R1		1	
R2		0	
R3		1	
R4		0	
R5		1	
R6		1	
R7		0	
R8		0	
R9		0	
R10		1	
R11		0	
R12		1	
R13		1	
R14		1	
R15		0	

instrucción todavía no escribió el nuevo valor de este registro.

Ejemplo:

```

1  div.d    F0,F2,F4
2  add.d   F6,F0,F8  # Riesgo WAR por F8 con sub.d
3  s.d     F6,0(R1)
4  sub.d   F8,F10,F14
5  mul.d   F6,F10,F8  # Riesgo WAW por F6 con add.d

```

- La instrucción 2 tiene un riesgo WAR en F8 con la instrucción 4 .
- La instrucción 5 tiene un riesgo WAR en F6 con la instrucción 2.
- Reemplazamos los operandos problemáticos por sus respectivos aliases (S y T) .

```

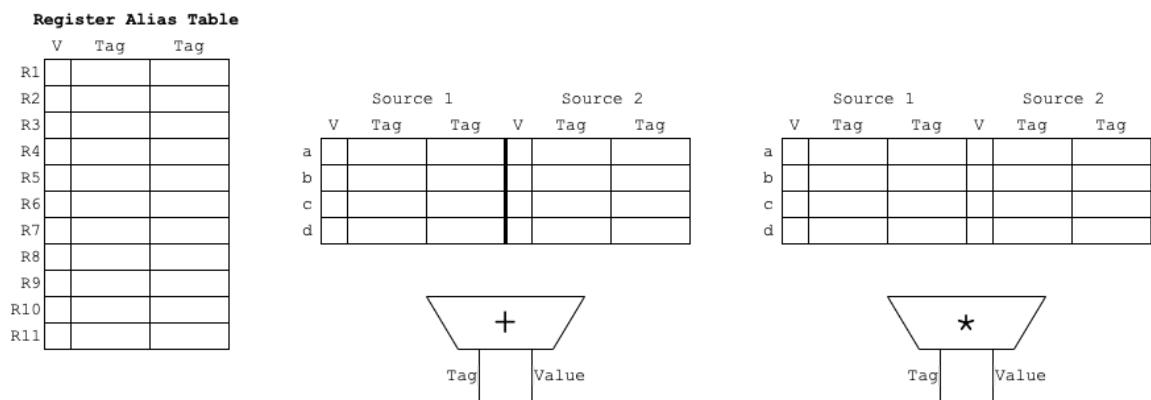
1  div.d    F0,F2,F4
2  add.d   S,F0,T  # WAR hazard por F8 con sub.d
3  s.d     S,0(R1)
4  sub.d   T,F10,F14
5  mul.d   F6,F10,T  # WAW hazard por F6 con add.d

```

Reservation Station (RS)

- Soluciona los otros tres problemas del algoritmo de Tomasulo .
 - Soluciona el problema de mantener en espera las instrucciones que todavía no se pueden ejecutar.
 - Soluciona el problema de saber cuándo están disponibles los operandos de una instrucción.
 - Soluciona el problema de ejecutar la instrucción cuando todos sus operandos estén disponibles.
- Cada línea de la RS es una instrucción y sus dos operandos .

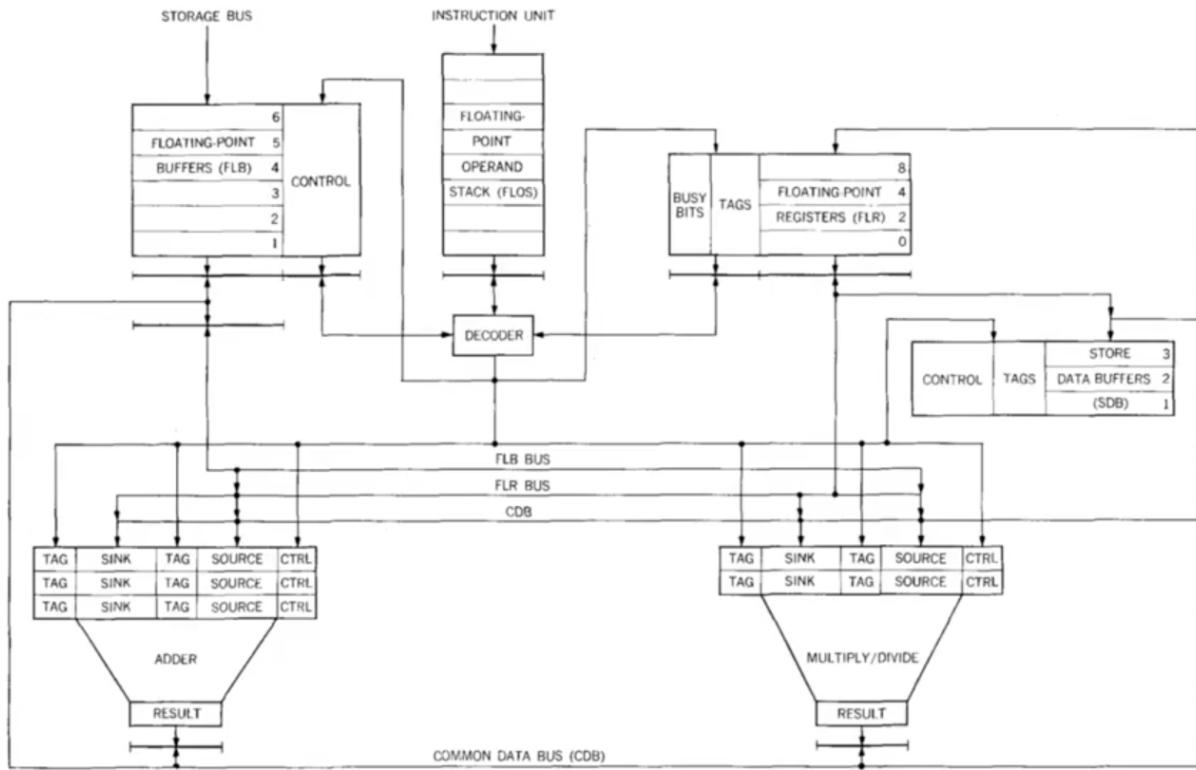
- La RS tiene un tamaño limitado.
- Una vez realizado el renombre se le asigna, a la instrucción, una Reservation Station.
- Una Reservation Station es un subsistema de hardware compuesto de bancos de registros internos que se encarga de mantener las instrucciones en espera hasta que estén listas para ser ejecutadas. Éste chequea constantemente por la disponibilidad de los operandos de la instrucción. Para cada uno de ellos cuyo valor no esté disponible, la RS guarda el tag que se le asignó en el paso anterior.
- Cada vez que una unidad de ejecución pone Ready un operando, transmite su tag asociado junto con su valor a todas las RS. Cuando una instrucción tiene todos sus operandos Ready, la RS espera a que la unidad funcional asociada a ella esté libre y luego la despacha.



Common Data Bus:

- Es un datapath que va por toda la arquitectura.
- Se usa para broadcastear el tag y value que salen de las ALU hacia la Register Alias Table y las RS para que tengan esos valores actualizados.

FPU de IBM 360/91 con la mejora de Tomasulo



Algoritmo de Tomasulo

If (RS tiene recursos disponibles antes de renaming)
Se inserta en la RS la instrucción y los operandos renombrados (valor fuente / tag)
Se renombra si y solo si la RS tiene recursos disponibles.

Else

stall

While (esté en la RS, cada instrucción debe:)

Mirar el tráfico por el Common Data Bus (CDB) en busca de tags que correspondan a sus Operandos fuente.
Cuando se detecta un tag, se graba el valor de la fuente y se mantiene en la RS.
Cuando ambos operandos están disponibles, la instrucción se marca Ready para ser despachada.

If (Unidad Funcional disponible)
Se despacha la instrucción a esa Unidad Funcional

If (Finalizada la ejecución de la instrucción)
La Unidad Funcional arbitra el CDB

Pone el valor correspondiente al tag en el CDB (tag broadcast)

If (El archivo de Registros está conectado al CDB)

Cada Registro contiene un tag que indica el último escritor en el registro.

If (tag del Archivo de Registro == tag broadcast)

Registro = valor broadcast

bit de validez = '1'

Recupera tag renombrado

No queda copia válida del tag en el sistema

El algoritmo se ejecuta para cada instrucción del programa.

// Defino si meto la instrucción en la RS o si stalleo el pipeline

- Chequeamos que haya lugar en la RS para todos los operandos de la instrucción (tanto fuentes como destino).

// Hay lugar en la RS

1. Creo un alias para cada operando no válido de la instrucción
2. Insertamos la instrucción y sus operandos renombrados en la RS. Estos valores los sacamos directamente

de la Register Alias Table.

- Si no hay lugar, stall

// La instrucción ya está en la RS

- Cada instrucción almacenada en la RS hace:
 1. Chequear el Common Data Bus en busca de tags que correspondan a sus operandos fuente.
 2. Cuando se detecta el tag que se esta esperando, se graba el valor de la fuente en la RS.
 3. Cuando ambos operandos estan Ready, la instrucción se marca Ready para ser ejecutada.

// La instrucción ya está lista para ser ejecutada

- Me fijo si hay alguna Unidad Funcional disponible (quien ejecuta la instrucción)

// Hay una Unidad Funcional disponible

1. Despacho la instrucción

// Listo, se despachó la instrucción ✓

- Me fijo si finalizó la ejecución de la instrucción


```
// Finalizó la ejecución de la instrucción
```

 1. La Unidad Funcional que ejecutó la instrucción broadcastea, usando el Common Data Bus, el tag y valor resultante.


```
// Esto significa que el operando destino de esta instrucción ahora está Ready. Entonces se lo comunico a los demás vía broadcast
```

```
// A la Register Alias Table le llega, por broadcast, el tag y valor
```
 2. Si tiene ese tag guardado:
 - a. Escribe el valor que le llega en el registro (R1, R2, etc), según corresponda con el tag.
 - b. Marca ese tag como válido.
 - c. Libera el tag.

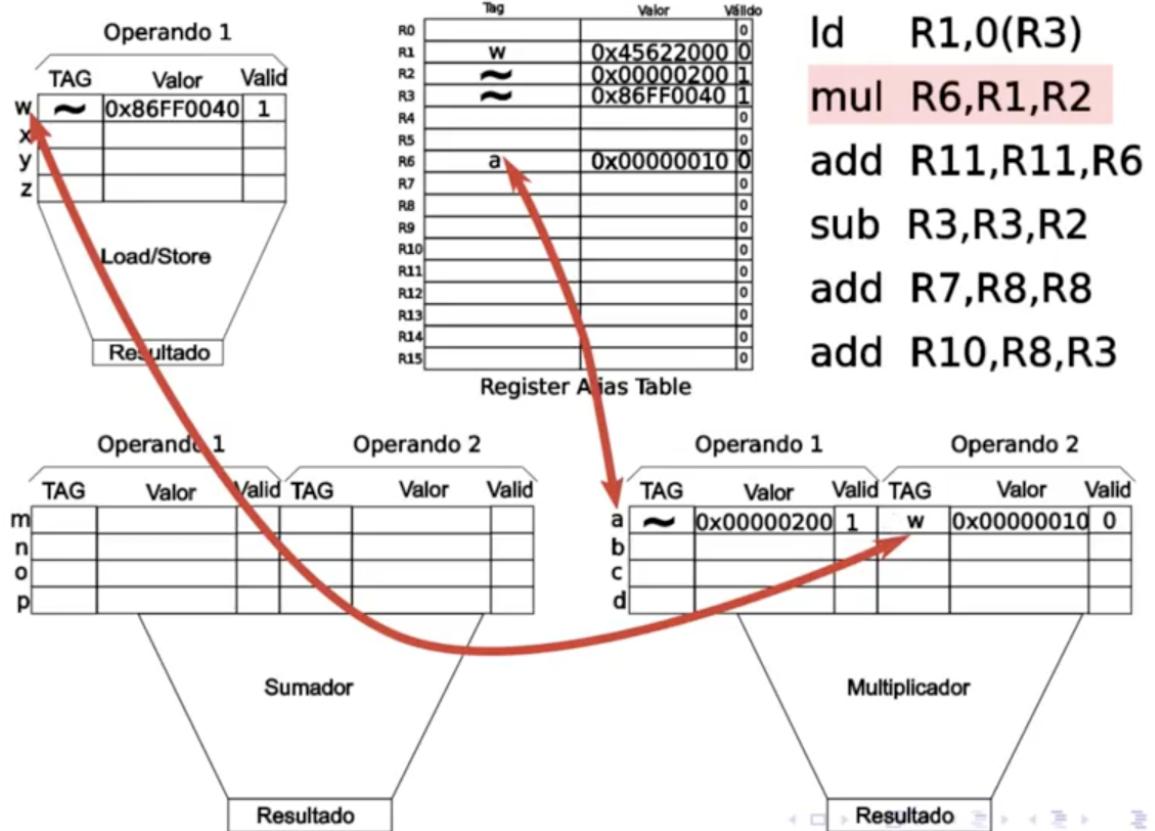
Ejemplo del Algoritmo

```

1   Id R1,0(R3)
2   mul R6,R2,R1
3   add R11,R11,R6
4   sub R3,R3,R2
5   add R7,R8,R9
6   add R10,R8,R3

```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
															R	W				
															E	E	R	W		
															E	R	W			
																	W			
																	W			
																		W		



Especulación por Hardware

Reordenando los resultados

- En el modelo de Tomasulo no hay ningún bloque que asegure que los resultados se escriben en orden.
- Más allá de que las instrucciones puedan ejecutarse en paralelo, la aplicación de los resultados necesariamente debe ser en orden.

- Las implementaciones posteriores introdujeron un módulo para hacer eso que se llama ReOrder Buffer (ROB).
 - Es un bloque parecido a la Reservation Station de Tomasulo. Contiene los valores de resultado de la etapa de ejecución y a qué registros deben ir.
 - El resultado permanece en el ROB desde que se obtenga hasta que se copie en el operando destino.
 - Contiene los campos `Tipo de instrucción`, `Destino`, `Valor` y `Ready`.
 - Escribe un valor en su destino si los anteriores valores están en ready o ya fueron aplicados. Se pueden escribir varios valores en el mismo ciclo si están todos en ready.

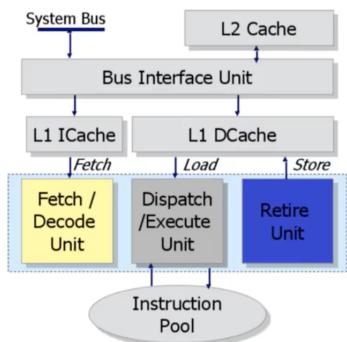
Casos prácticos que mezclan todo lo visto

Pentium Pro

- Fue un procesador muy importante dentro de la genealogía de Intel. Inauguró la microarquitectura P6, cuyo fundamento de ejecución fuera de orden se mantiene hasta la fecha.
- Fue el primer procesador de Intel que ejecutaba fuera de orden. Los Pentium ejecutaban en orden. Los Pentium Pro ejecutaban fuera de orden.
- Luego vinieron el Pentium II, Pentium II Xeon, Celeron, Pentium III y Pentium III Xeon.
- Todos estos se basaban en una estructura interna denominada Three Cores Engine.

Three Cores Engine

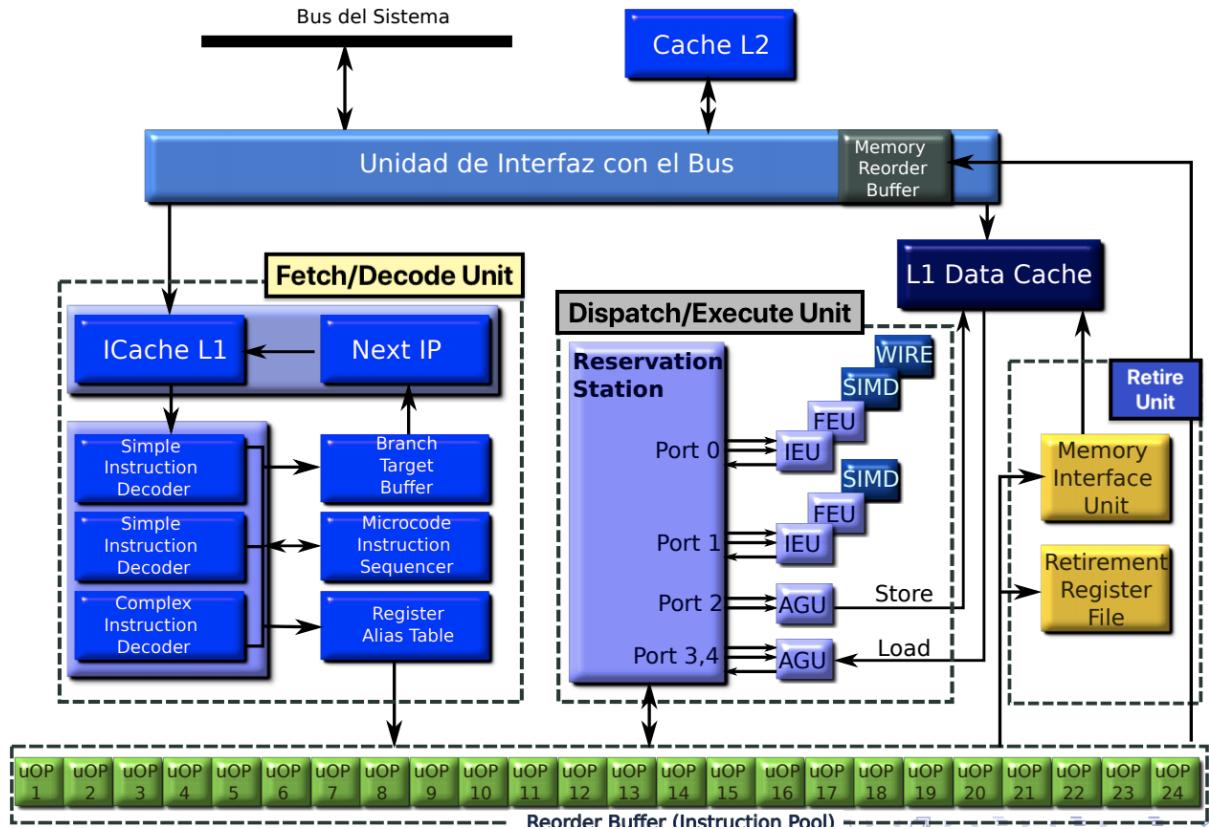
- **Tres cores:**
 1. **Unidad de decodificación y búsqueda (Fetch/Decode Unit)**



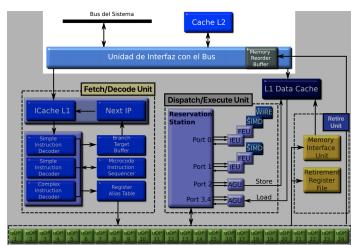
2. Unidad de despacho y ejecución (Dispatch/Execute Unit)

3. Unidad de retiro (Retire Unit)

- Cache de datos y cache de código separadas
- Scheduling dinámico
- Basado en una ventana de instrucciones y no en un pipeline superescalar
- **Las instrucciones se traducen en microoperaciones básicas.**
- **Las microoperaciones se almacenan en el instruction pool (que no es más que un reorder buffer).**
- Los tres cores tienen visibilidad de esa ventana de ejecución.
- La ejecución fuera de orden y especulativa es para las microoperaciones.
- Las microoperaciones tenían todas el mismo tamaño, como si fuera RISC.

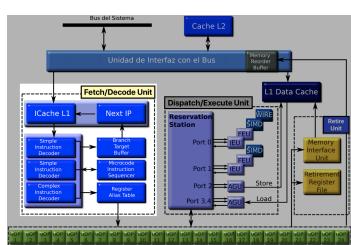


Unidad de Interfaz con el Bus



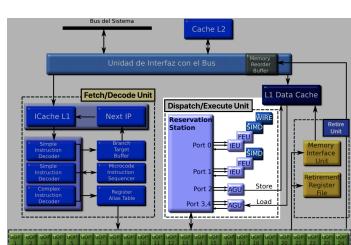
- Es un bloque que trabaja parcialmente en orden.
- Está encargada de conectar los tres cores con el mundo exterior.
- Divide los modelos:
 - De la Unidad de Interfaz hacia afuera (incluyendo la Cache L2 afuera), es Von Neumann.
 - De la Unidad de Interfaz hacia adentro es Harvard. La Cache L1 está separada en datos y código.
- Soporta hasta cuatro accesos concurrentes a la Cache L2.
- El acceso al Bus del sistema se regula todo por MESI y snooping para asegurar la coherencia de la memoria.

Unidad de decodificación y búsqueda



- **Instruction Cache L1:** tiene 16 bytes por línea. Entrega los 16 bytes alineados a la Unidad de Decodificación.
- **Puntero Next IP:** Apunta a la línea donde está la siguiente instrucción o a la de la instrucción target de un salto. Es decir, indica qué instrucción tiene que ir fletcheando el pipeline.
- **Tres Instruction Decoders:** Dos de instrucciones simples y uno de instrucciones complejas.
 - Los decodificadores de instrucciones simples traducen la instrucción en una microoperación
 - El decodificador de instrucciones complejas traduce la instrucción en dos o más microoperaciones.
- **Microcode Instruction Sequencer:** Convierte una instrucción todavía más compleja en una secuencia de microoperaciones.

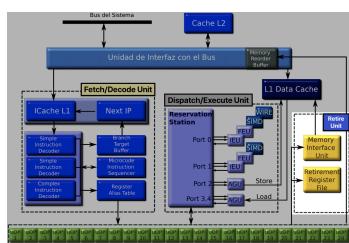
Unidad de despacho y ejecución



- **Esta unidad es la única que trabaja fuera de orden.**
- Selecciona las microoperaciones dependiendo del estado (no del orden) desde el pool de instrucciones.

- Si el estado de la microoperación indica que sus operandos están disponibles, la Reservation Station verifica que alguna unidad funcional esté libre para su ejecución y le envía la microoperación.
- El resultado se escribe en el pool de operaciones (ReOrder Buffer).
- **Branches:** Las microoperaciones se marcan como saltos en el ReOrder Buffer y también se escribe allí la dirección predicta por el BTB (Branch Target Buffer). Si el salto falla, limpia el ROB, las RS (limpia todo).

Unidad de retiro



- Mira permanentemente el estado de cada instrucción en el ROB.
- No sólo chequea su estado, sino que las reinserta en el orden en que ocupan en el programa.
- Esto incluye el procesamiento de interrupciones, excepciones, traps, breakpoints y predicciones fallidas.
- **Retirement Register File:** Se encarga de escribir los operandos resultantes en los registros de la arquitectura.
- **Unidad de interfaz de memoria:** Aplica los datos en el Cache L1 de datos.

Intel NetBurst (Pentium 4)

- Es el siguiente en la línea que marcó una cierta innovación.
- En vez de Cache L1 de código tenía un **Trace Cache**:
 - Es una cache que guarda microoperaciones en vez de instrucciones.

Procesadores Multithread

- Los procesadores multithread tienen dos estados arquitecturales.
- **No son dos procesadores, sino que son dos vistas de procesadores.**
- **Se duplica lo necesario y se comparte lo crítico.**
- Se duplica:
 - La cola de microoperaciones
 - El Instruction Pointer
 - Register Alias Table
 - El ROB
- Se comparte:
 - La cache L1 de datos
 - Las unidades de ejecución
 - Los registros internos de la arquitectura
 - Las Reservation Station
 - etc
- **Conclusión:** La mayoría de cosas se comparte. Con un 15% más de electrónica se logra un aumento mucho mayor a 15% en la performance.

Multicore y Manycore

- **Es poner dos procesadores completos dentro del chip.**

- Dejaron obsoletos a los procesadores multithread.
- **Es mejor muchos cores sencillos que un sólo core superpoderoso. Con más cores tenemos igual performance y menor consumo.**