ILP (Instruction Level Parallelism)

Pipelining

“Arquitectura que permite crear el efecto de superponer en el tiempo, la ejecución de varias instrucciones a la vez.”

En los primeros procesadores, cada etapa del ciclo de instrucción (F/D/E/R) se ejecutaba en un ciclo de clock.

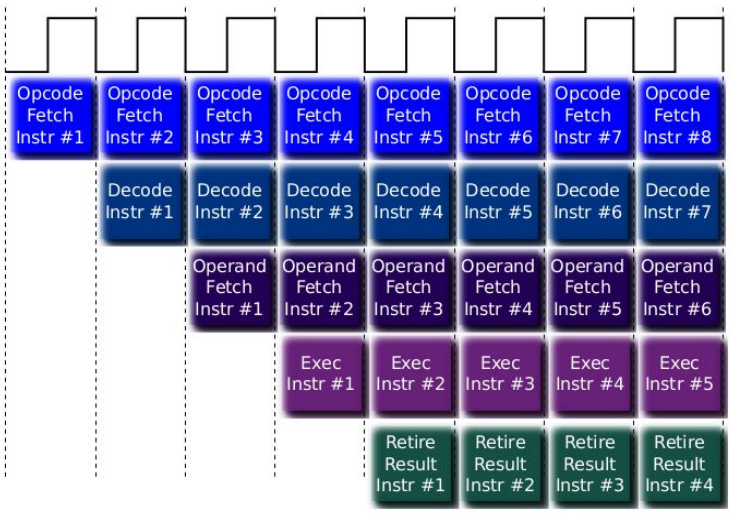
Motivación

Por ejemplo, mientras se está haciendo el Decode de una instrucción, la unidad de Fetch no se utiliza para nada.

Idea

Las diferentes etapas para ejecutar una instrucción funcionan en paralelo sobre distintas instrucciones. Mientras se hace Decode de la #2, se puede ir haciendo Fetch de la #1.

- Requiere de poco hardware adicional, solo se necesita que los distintos bloques del procesador operen en simultáneo sobre distintas instrucciones.
- Análogo a la idea de una “línea de montaje automóvil”.



Más concurrencia => Más trabajo realizado en un ciclo de clock => **Mayor throughput** (cantidad de instrucciones completadas por ciclo de clock).

No reduce la latencia de las instrucciones (el tiempo que tardan en completarse). De hecho, puede aumentar la latencia debido al overhead de dividir las instrucciones, o debido a que el pipeline debe pararse.

An ideal pipeline

- Repetition of identical operations:
  The same operation is repeated on a large number of different inputs
  - Different instructions => Not all need the same stages

  External Fragmentation: Some pipeline stages idle for some instructions

- Uniformly partitionable sub-operations
  Processing can be evenly divided into uniform-latency sub-operations
  - Different pipeline stages => Not the same latency

  Internal Fragmentation: Need to force each stage to be controlled by the same clock

- Repetition of independent operations:
  No dependencies between repeated operations
  - Instructions are not independent of each other

  Pipeline Stalls: Pipeline is not always moving

Causas de Pipeline "*Stalls*"

Operaciones multiciclo de larga latencia

Si una operación demora muchos ciclos en completarse (por ejemplo, un Load) se traba el Pipeline.

Obstáculos estructurales (resource contention)

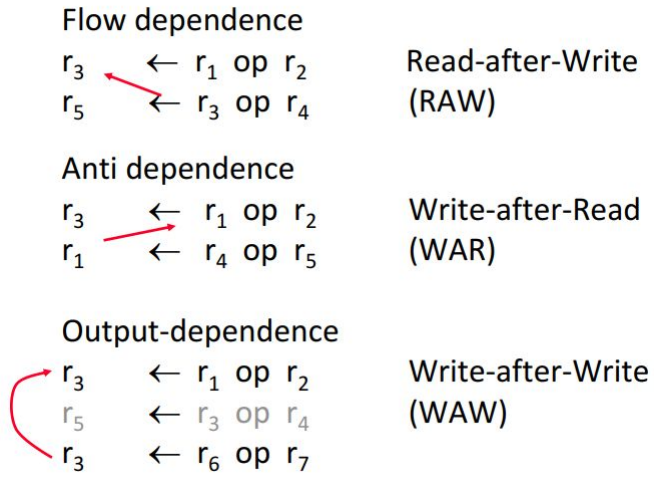
Cuando hay un conflicto por el uso de los recursos de alguna etapa.
Ejemplo, si el procesador solo tiene una etapa de acceso a memoria y la comparte entre datos e instrucciones. Si una instrucción necesita buscar un operando en memoria, entrará en conflicto con el Fetch de otra instrucción. Alguna deberá detenerse.
Posibles soluciones (casi siempre implican agregar hardware):

- Duplicar el recurso de hardware
- Usar caches separadas para datos e instrucciones
- Buffers de instrucciones como pequeñas colas FIFO
- Tener múltiples puertos de acceso para memoria
- Etc.

Dependencias entre instrucciones

1. Dependencias de Datos

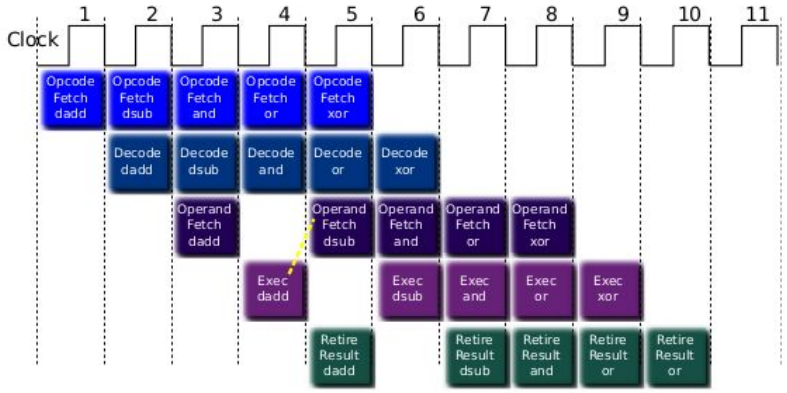
Las dependencias de datos pueden ser de tres tipos: Flow, Anti, Output.
Las dependencias Flow son las verdaderas dependencias de datos del programa (las que le dan su semántica).
Las otras no son dependencias reales, (son más bien dependencias de nombres, no de valores).



Las Anti y Output dependences no generan problema dado que toda la ejecución es en orden (generarán más problemas luego en **Ejecución Fuera de Orden**).

Las Flow Dependences hay que detectarlas y esperar (*Stall*) en ese caso.
Una manera de hacer esto es con **Scoreboarding** (explicación más adelante en Ejecución Fuera de Orden), aunque también genera *Stalls* con las anti y output dependences.

También puede aplicarse **Forwarding**, que consiste en extraer el resultado directamente de la salida de la unidad de ejecución (ALU, FP, etc.) en lugar de esperar a que el resultado sea retirado.
Esto mejora un poco el tiempo de *Stall*, aunque tampoco es perfecto y no siempre puede aplicarse (por ejemplo, antes una operación de Load, el resultado no puede adelantarse, hay que esperar a que impacte el resultado en el procesador).



También se puede eliminar la dependencia a nivel de Software (en tiempo de compilación).

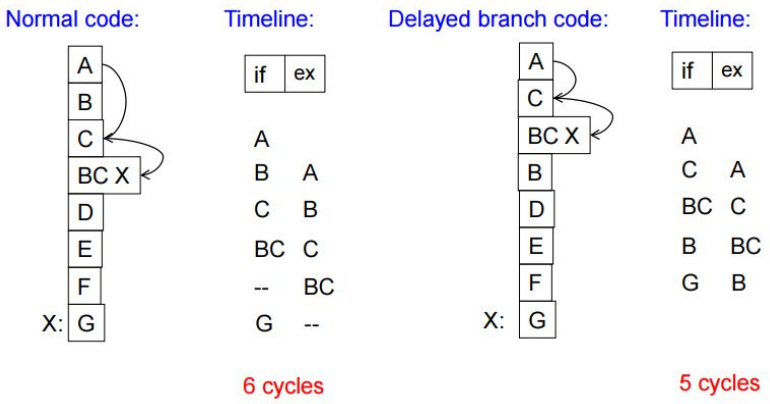
2. Dependencias de Control

Las instrucciones de programa se ejecutan en orden secuencial, pero esto no es siempre así dado que existen instrucciones de control de flujo. ¿Qué pasa si no encontramos con un salto (*Branch*)? Hay una discontinuidad en el flujo del programa, esto dificulta la ejecución del Pipeline. Existen varios tipos de *Branches*:

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Soluciones posibles:

- **Stall** (no conviene prácticamente jamás)
  El Pipeline se dejará de llenar hasta resolver el branch.
- **Branch Prediction** (más adelante), adivinar la siguiente instrucción.
- **Branch delay slot**
  El procesador provee N (normalmente 1) slots de instrucciones que son ejecutadas sí o sí, independientemente de la evaluación del branch. El compilador puede meter instrucciones independientes al branch en estos slots, de esta manera, se ejecutan instrucciones útiles mientras el branch todavía se está resolviendo.



Funciona bien si la cantidad de delay slots es suficiente para mantener el Pipeline lleno, y se pueden encontrar suficientes instrucciones independientes para llenar los slots. En general es difícil, además la cantidad de delay slots crece con la profundidad del Pipeline (y más si es un procesador **Superscalar**).

- **Predicated execution**
  Tener instrucciones en la ISA que permitan hacer operaciones de manera condicional. Por ejemplo tener una instrucción `CMOV condition, R1, R2` que mueve R2 a R1 dependiendo del valor de condition. De esta manera, se elimina la discontinuidad en el flujo del programa.
  Tiene sus limitaciones de todas maneras (no todo puede resolverse así) además de que se necesita soporte de la ISA. Además, se hace “trabajo demás” para las instrucciones donde la condición no se cumple (la instrucción se convierte en un NOP).

Branch Prediction

A medida que aumenta la complejidad de un procesador, y aumentan la cantidad de etapas de un Pipeline, el tiempo que demora el mismo en recuperarse de un branch (*Branch Penalty*) aumenta también.
Se vuelve crucial lograr una buena predicción de saltos entonces. La idea consiste en tratar de adivinar la dirección que tomará el salto (si es condicional además, predecir si será tomado o no).
Ante una mala predicción, el Pipeline se debería vaciar (*Flush*) de las instrucciones que venían luego del branch.

Se deben hacer 3 cosas en la etapa de Fetch:

1. Saber si la instrucción es un branch
2. Si es condicional, predecir taken/not taken.
3. Predecir la dirección destino (target address), en el caso que sea taken.

BTB (Branch Target Buffer)

Es una caché de instrucciones de salto cuyas entradas contienen un par; dirección de la instrucción de salto, y dirección del target resuelta.
Se trabaja como una memoria de acceso por contenido (CAM), y se accede mediante el valor del Program Counter (PC).

Observación: Para un branch condicional de salto directo, el target (del caso taken) es siempre el mismo. Con la BTB, y prediciendo si el branch es taken/not taken alcanza.

Direction Prediction (Predicción Taken/Not Taken)

Predicción estática (tiempo de compilación)

- **Always Not Taken**: el procesador asume siempre que no será tomado. Fácil de implementar, no se necesita BTB. Baja accuracy (~30-40%).
- **Always Taken**: el procesador asume que siempre será tomado. Funciona un poco mejor (~60-70%).

Observación: Los branches de loops son usualmente tomados. Suelen ser tomados muchas veces, y luego una vez no tomados. En general estos saltos son “hacia atrás” (para formar el loop).

- **Backward taken, forward not taken (BTFN)**: Si el salto es hacia atrás, asumir taken, sino, not taken.
- **Profiler based**: el compilador determina qué tan probable es que cada branch sea tomado usando algún profiler. Es más preciso que las anteriores por tener granularidad por branch, pero es muy dependiente de qué tan representativo es el profiling. Puede no ser muy bueno, por ejemplo:
  TTTTTTTTTTNNNNNNNNNN => 50% accuracy
  TNTNTNTNTNTNTNTNTNTN => 50% accuracy
- **Program analysis based**: uso de heurísticas basadas en en el programa. Por ejemplo, según la instrucción específica, o tratar de identificar un loop, etc.
- **Programmer based**: el programador puede dar pistas también mediante código en ciertos casos.
- **Loop unrolling**: el compilador puede “desenrollar” los loops, eliminando así branches, pero aumentando el tamaño del código.

Predicción dinámica (runtime)

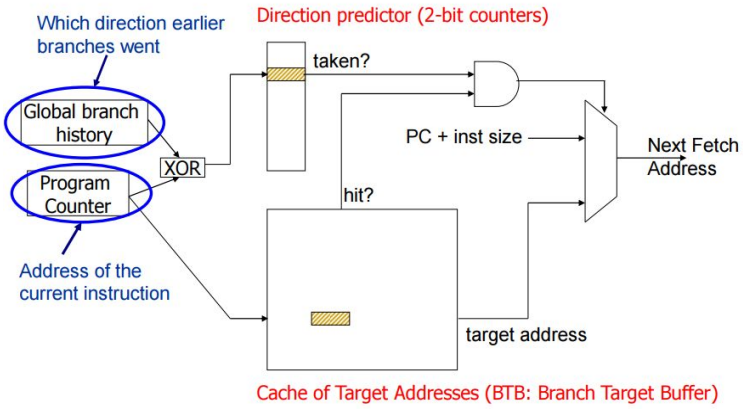
En general son mejores estrategias que las estáticas, por poder adaptarse, pero requerirán más hardware y lógica extra.

- **Last time predictor (1 bit predictor)**: también llamada *Branch Prediction Buffer*. Tabla indexada por la dirección de memoria del salto, que almacena un bit, indicando el último resultado para ese branch (1: taken, 0: not take).
  Simple de implementar. Siempre predecirá mal 2 veces para un loop (la primera, y la última, por tanto el accuracy para un loop de N ciclos será (N-2)/N).
  TNTNTNTNTNTNTNTNTNTN => 0% accuracy
- **2 bit predictor**: el problema con el predictor anterior es que cambia muy rápidamente de estado. La idea es agregarle un poco de "histéresis". Para esto, se implementa la misma idea pero con 2 bits, y se tiene la siguiente máquina de estado.

  Tiene bastante mejor accuracy que el anterior (~80%).

Observación 1 (Global Branch Correlation): el resultado de un branch está correlacionado con los resultados de otros branches.
Observación 2 (Local Branch Correlation): el resultado de un branch está correlacionado con los resultados pasados del mismo branch.

- **Global History Register (GHR) + Pattern History Table (PHT)**: es un predictor de 2 niveles. Se tiene un registro de N bits (GHR) que almacena los resultados Taken/Not taken de los últimos N branches. Además, se tiene una tabla de 2^N entradas que indica la predicción para un GHR dado. La tabla puede almacenar, por ejemplo, un predictor de 2 bits para cada valor posible de la GHR.
  - **Gshare predictor**: misma idea, pero el GHR se hashea junto con el PC (con un XOR por ejemplo), para darle más contexto al GHR. Utiliza mejor la PHT (podría haber muchas colisiones en la tabla).



- **Local History Register (LHR) + Pattern History Table (PHT)**: misma idea pero para varios branches individualmente. Se tienen varios registros, LHR, para distintos branches que guardan los últimos N resultados de cada uno.
- **Predictores híbridos**: combinar predictores, seleccionar el mejor o más confiable según alguna lógica.

Ejecución Fuera de Orden

Idea

Los Pipelines buscan la siguiente instrucción y la envían a ejecutar. Si una instrucción es dependiente de otra, el envío es demorado. Si además las instrucciones de las que depende demoran mucho en completarse, esta instrucción, y todas las subsiguientes serán demoradas. Tal vez hay instrucciones más adelante que no son realmente dependientes de la instrucción demorada. La idea es tratar de enviar a ejecutar instrucciones independientemente del orden del programa, y hacerlo a medida que los operandos de las instrucciones están listos.
Para hacer esto bien y correctamente (conservando la semántica del programa) hay que tener cuidado con los riesgos antes vistos (**RAW, WAR y WAW**).

Implementación

Es necesario partir la etapa de Decodificación en 2 sub etapas:

1. Envío: trabaja en orden, decodifica las instrucciones y las envía a algún buffer.
2. Lectura de operandos: esta unidad trabaja fuera de orden, la idea es que levante las instrucciones del buffer en cuanto sus operandos estén listos (más allá del orden de programa).

Manejo de excepciones

No solo es necesario mantener la semántica del programa correcta, pero además con ejecución fuera de orden, surgen otros problemas. Dado que ejecutamos fuera de orden,

instrucciones que están más adelante en el programa podrían terminar antes que instrucciones anteriores, y escribir sus resultados al estado arquitectónico.
Ante una excepción, lo deseable sería que el estado arquitectónico visible de la ISA sea el mismo que se vería en caso de que la ejecución haya sido en orden (pensar, por ejemplo, en *debugging*, sin lograr esto, se vuelve una tarea muy difícil).
Surge el concepto de **Excepciones Imprecisas**, que son aquellas que al producirse, el estado arquitectónico de la ISA no es exactamente el mismo que si la ejecución hubiese sido en orden. Esto sucede cuando se cumple al menos una de las siguientes:

1. Hay alguna instrucción previa a la que generó la excepción que aún no ha completado.
2. Hay alguna instrucción posterior a la que generó la excepción que ha sido completada.

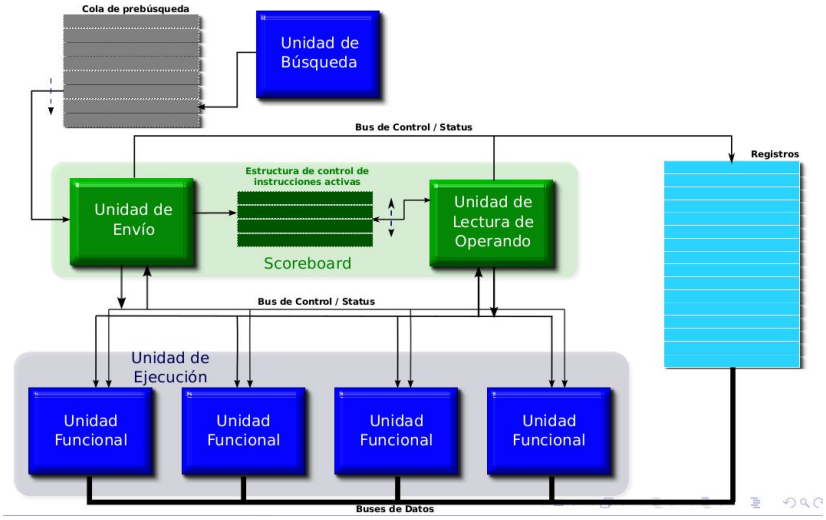
Algunos procesadores que implementaban ejecución fuera de orden, solo tenían excepciones imprecisas.
Para poder garantizar **Excepciones Precisas**, hay varias soluciones posibles:

1. Usar un ROB.
2. Usar un History Buffer.
3. Tener dos Register Files separados, uno especulativo (Future File) y otro arquitectónico (Retirement Register File), combinado con un ROB, para poder mantener este último. Esta solución la emplea el Pentium 4 Netburst por ejemplo.

Para más detalles, ver.

Scoreboarding

Método sencillo para poder permitir ejecución fuera de orden.

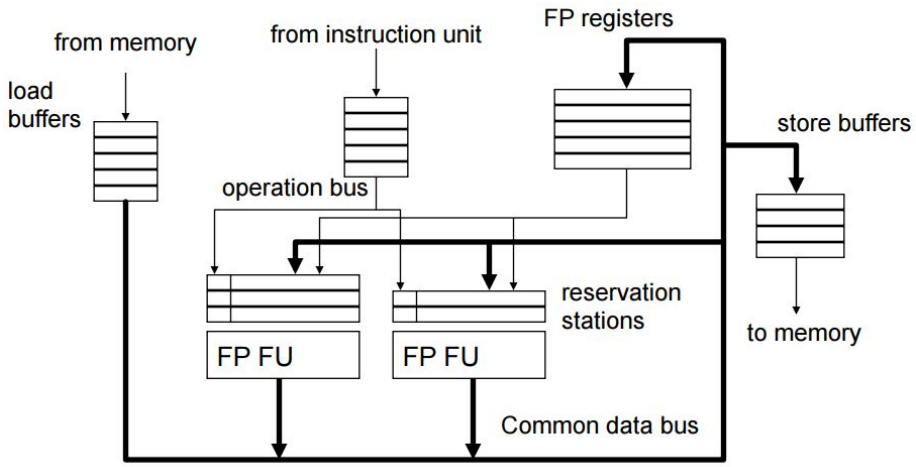


1. Envío: al llegar una instrucción, se chequea qué registros necesitará leer/escribir, y se guarda esta información (que se necesitará más adelante). Si hay otras instrucciones pendientes que escriben al mismo destino que ésta, o si no hay una unidad de ejecución disponible, la instrucción se detiene (Stall) hasta que finalicen. De esta manera se evitan riesgos WAW.

2. Lectura de operandos: una vez enviada la instrucción al módulo correspondiente, la instrucción espera hasta que sus operandos estén listos, es decir, hasta que previas instrucciones escriban sus resultados en los registros fuente de ésta. De esta manera se evitan riesgos RAW, y se mantiene la semántica del programa.
3. Ejecución: si los operandos están listos, la instrucción empieza a ejecutar. Al terminar, el Scoreboard recibe el resultado.
4. Escritura de resultado: en esta etapa, la instrucción está lista para escribir su resultado. Sin embargo, esta etapa se demora si existen instrucciones previas (en el orden de programa) que leen sus operandos del mismo registro al cual la instrucción desea escribir, hasta que éstas instrucciones terminen su etapa de lectura. De esta manera, se evitan los riesgos WAR, dado que las instrucciones previas pueden leer los valores correctos, antes que la nueva instrucción pise el registro.

Tomasulo

El método de Scoreboarding nos permite ejecutar fuera de orden evitando los riesgos, pero es muy restrictivo (ante un riesgo WAW o WAR este se detiene). Recordando, las anti y output dependences no son dependencias reales (no hacen a la semántica del programa), sino que son conflictos de nombres.



La idea de Tomasulo es utilizar **Register Renaming** para así evitar los riesgos WAR y WAR, además considerando los RAW como lo veníamos haciendo (esperando hasta que los operando esté listo).
La idea es la siguiente:

1. En la etapa de Envío de una instrucción, se renombra (asocia nuevo un **Tag**) al registro que la instrucción va a escribir, y se coloca la instrucción en una entrada de alguna **Reservation Station**. Si no hay ninguna RS disponible, se produce un Stall.
2. Las instrucciones esperan en las RS a que sus operandos estén listos.
3. Cuando una instrucción finaliza, ésta hace un **Broadcast** del valor que produjo y el Tag que le había sido asociado.

4. Las RS capturan esto y comparan el Tag con los de sus entradas. Si coinciden, se captura el valor y se marca que ese operando está listo.
5. Hay lógica chequeando constantemente las RS, y cuando alguna entrada tiene todos sus operandos listos, la instrucción se levanta (**Wake Up**) y se envía a la unidad de ejecución. De esta manera, se evitan los riesgos RAW.

Como los registros son renombrados todo el tiempo, esto evita potenciales riesgos WAR y WAW, dado que se trabaja con varias “versiones” de los registros simultáneamente.

Para además poder tener commit de los resultados en orden, puede agregarse un ROB a la arquitectura anterior. Con esto, en la etapa de Envío, además de reservar un lugar en una RS para una instrucción, también se debe reservar una entrada en el ROB.

Memoria

Jerarquía de Memoria

(insert pirámide)

Principio de localidad

e.e

Cache

Esquemas de cache

Mapeo directo

Cada bloque de memoria se mapea a un único bloque de cache. El mapeo está dado por $i = j \bmod k$. La principal desventaja es que dos bloques de memoria podrían estar compitiendo constantemente por la misma línea.

Cache totalmente asociativa

No hay ninguna regla para qué bloques se almacenan en qué líneas. Pero para saber si un bloque está en la caché, se necesita mucha lógica (circuito) adicional.

Cache asociativa por conjuntos

Funciona como varias cachés asociativas, divididas entre distintos sets. Cada bloque de memoria es asignado a uno de estos sets según su dirección (al igual que con mapeo directo), pero pueden ir a parar a cualquier línea de ese set (como en una cache asociativa).

Políticas de escritura

- **Write through**: se escribe directamente en caché y a memoria principal.
- **Write through buffered**: se escribe a caché. El procesador sigue operando usando la caché, mientras que el controlador de caché actualiza la memoria principal.
- **Copy/Write back**: se escribe solo a caché. Solo se escribe a memoria cuando la línea de caché se va a desalojar. Para identificar esto, se marca la línea con un *Dirty Bit*. Minimiza las escrituras a memoria principal.

Algoritmos de reemplazo

- **Random**
- **FIFO**
- **LRU**: Least Recently Used. En la práctica es difícil de implementar “verdaderamente”.
- **LFU**: Least Frequently Used.
- **Not MRU**: Not Most Recently Used.
- **Victim-Next Victim**

Coherencia de Cache en sistemas SMP

En los sistemas con varios procesadores, cada uno suele tener sus propios niveles de caché. Esto puede producir inconsistencias entre lo que tengan las distintas cachés (aún si la política de escritura es de **Write Through**).

Para mantener la coherencia entre las cachés, se implementa algún tipo de protocolo.
El protocolo puede ser centralizado, y a éstos se los conoce como **Protocolos de Directorio**, donde el directorio es el encargado de mantener toda la información pertinente al estado de cada caché. La principal desventaja de este esquema, es que el directorio representa un cuello de botella para el protocolo (por ser centralizado el mismo).

Hay protocolos descentralizados, donde las cachés se intercomunican entre ellas mediantes buses, que son llamados **Protocolos de Snooping** (“hurgar”). Dentro de estos, existen 2 grandes variantes:

1. Write Invalidate: puede haber varios lectores, pero un solo escritor. Cuando un procesador desea hacer una escritura, envía un mensaje a los demás para que estos invaliden la línea de caché.
2. Write Update: puede haber múltiples lectores y escritores. Cuando un procesador desea hacer una escritura, envía un mensaje a los demás con la palabra que debe ser actualizada en las otras cachés.

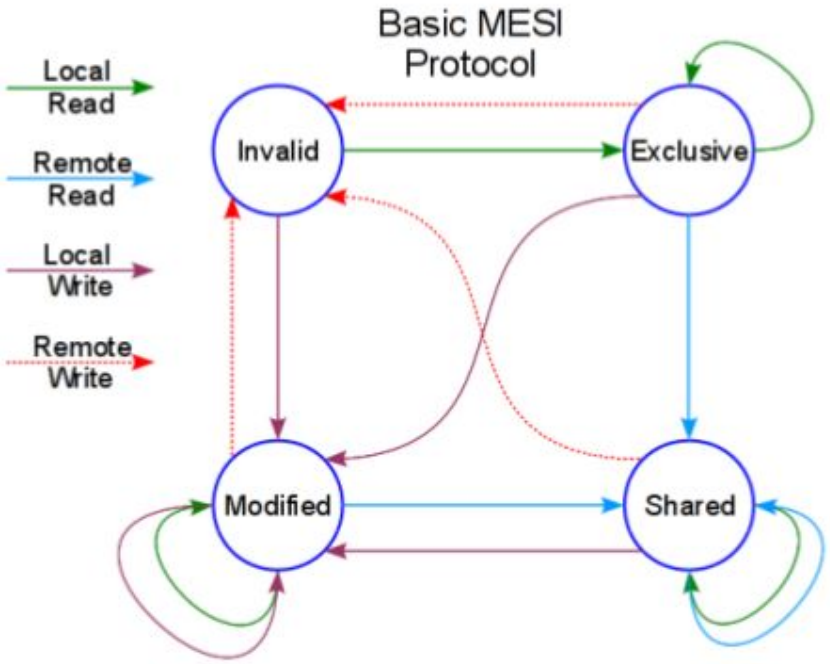
Protocolo MSI

Protocolo MESI

En general para Write-back cachés.

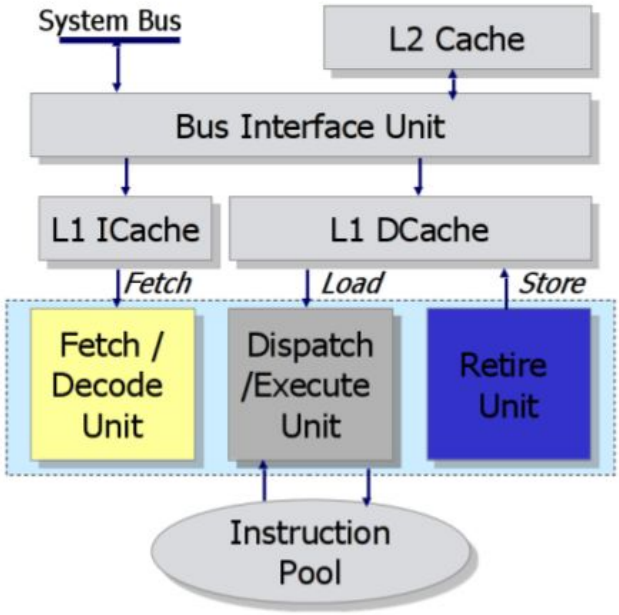
Modified: la línea de caché existe únicamente en esta caché, y ha sido modificada.
Exclusive: la línea de caché existe únicamente en ésta caché, y no ha sido modificada.
Shared: la línea de caché podría estar presente en varias cachés, y no ha sido modificada.
Invalid: la línea de caché es inválida.

Este protocolo reduce la cantidad de intercambios con la memoria principal, en comparación con el protocolo MSI, por tener el estado Exclusive. Cuando está en Exclusive, si va a modificar la línea, no es necesario avisar a las otras cachés (porque tiene la única copia), así ahorra tráfico de mensajes.



Casos prácticos

Three core engine (Intel P6)

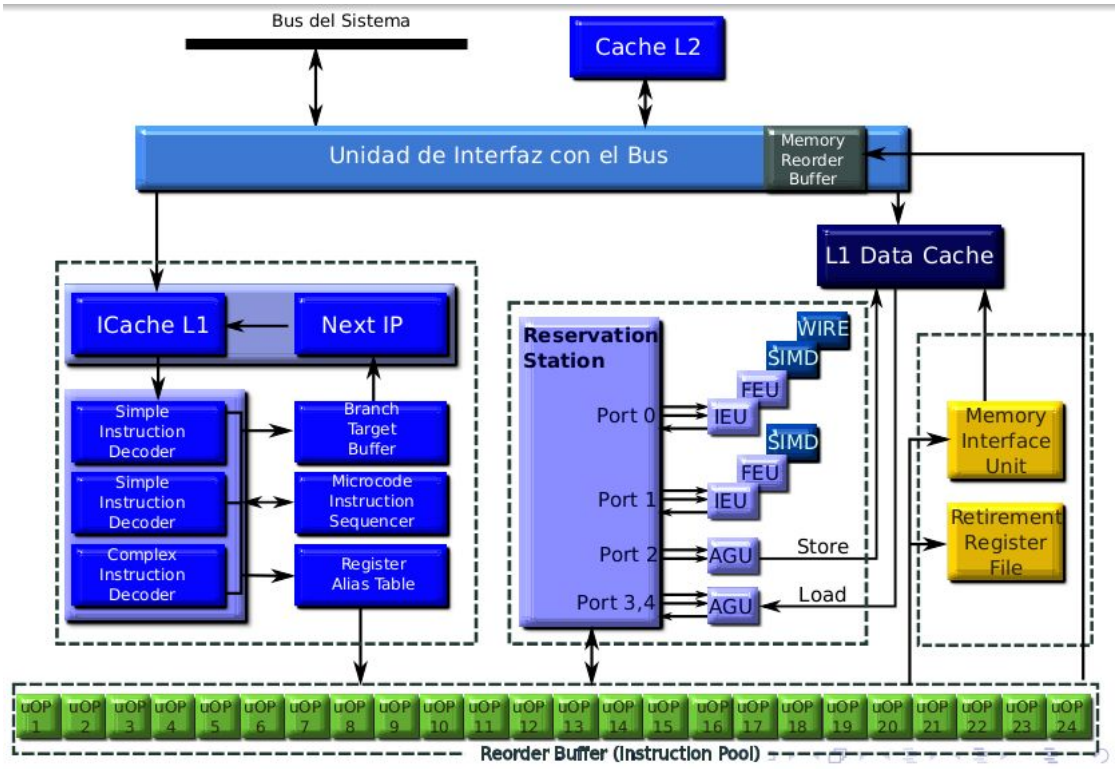


Tiene 3 unidades principales (de ahí su nombre):

1. **Fetch/Decode Unit**, que trabaja in-order sobre el programa.
2. **Dispatch/Execute Unit**, que trabaja out-of-order, a medida que los operandos de las instrucciones están listos.
3. **Retire Unit**, que trabaja in-order, el encargado de hacer el commit del estado arquitectónico.

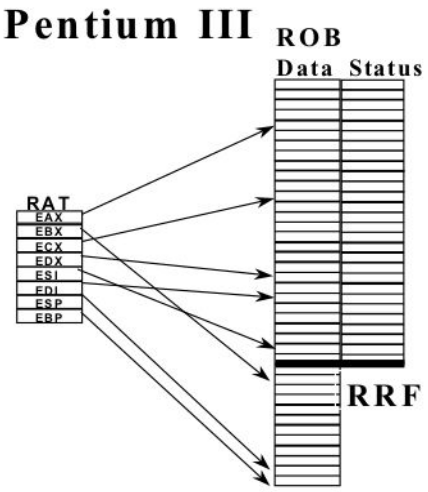
Las 3 unidades tienen acceso al **Instruction Pool**, que funciona como un ROB básicamente. Las instrucciones de programa son traducidas en varias instrucciones más pequeñas (micro instrucciones, *μops*), que son depositadas en el Instruction Pool a medida que se decodifican. Implementa ejecución especulativa además.

La ICache L1 es la caché de instrucciones. NextIP apunta a la siguiente instrucción secuencial, o en el caso de un branch, al target predicho por el BTB. Los tres decodificadores trabajan en paralelo. La mayoría de las instrucciones se traducen en 1 sola μops, otras en 4, y algunas otras (las más complejas) pasan por el Complex Instruction Decoder que generará las que sean necesarias (podrían ser muchas más).



El dispatch y ejecución de las operaciones se hace fuera de orden, según el estado de los operandos de las mismas. Cuando esto sucede, la Reservation Station verifica que haya alguna unidad de ejecución libre para la instrucción. Los resultados son escritos en el ROB.

Posee un solo RAT (Register Alias Table) que marca el estado especulativo de los registros. Guarda punteros (es decir, emplea Register Renaming), que bien apuntan a entradas del Instruction Pool (ROB, aquellas que aún no llegaron al commit), o a entradas del RRF (Retirement Register File). El RRF guarda el estado arquitectónico luego del commit de las instrucciones.



La Retirement Unit verifica constantemente el estado de la ROB, para hacer los commits en orden, recién ahí los cambios se ven impactados en el RRF. La Memory Interface Unit Interface impacta estos cambios en la Cache L1 de datos.

En caso de excepciones, o saltos mal predichos, se limpian las entradas de la ROB subsiguientes a la instrucción del salto/excepción. Las entradas del RAT ahora apuntarán a las del RRF correspondiente.

Pentium 4 (Intel Netburst)

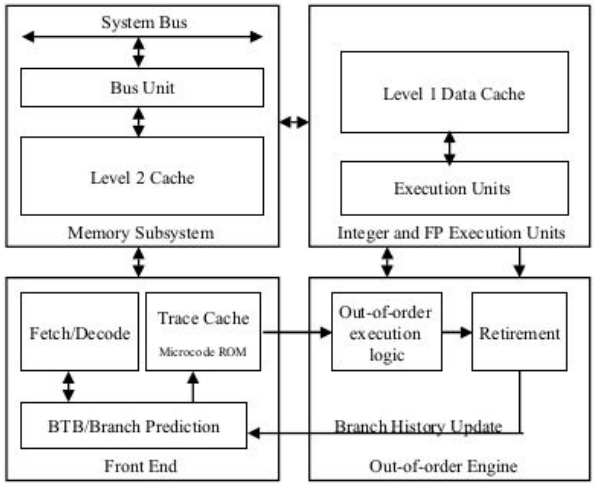
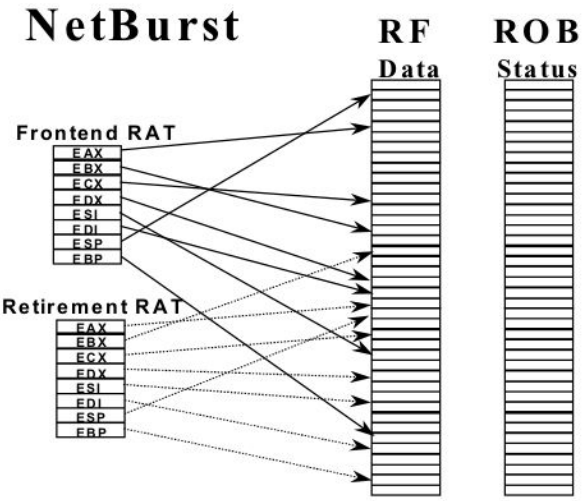


Figure 1: Basic block diagram



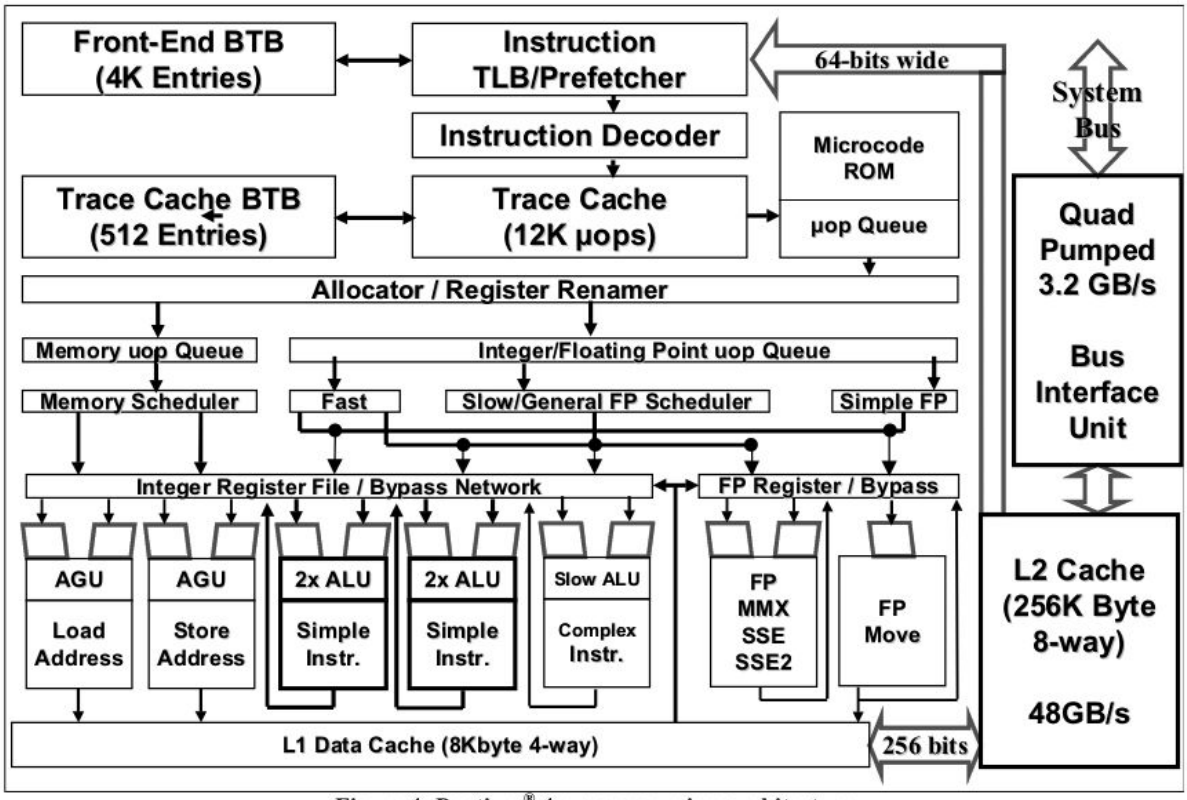


Figure 4: Pentium® 4 processor microarchitecture

Hyperthreading Pentium (Intel Xeon Server)

La idea es tratar de procesar 2 threads en simultáneo usando el mismo procesador físico. Esto se conoce como Simultaneous Multi-Threading (SMT).
Se duplica el estado arquitectónico del procesador (registros, IP, control de interrupciones, etc), de manera que se tienen 2 procesadores lógicos, pero comparten la mayoría de los recursos de hardware (buses, memoria caché, branch predictor).

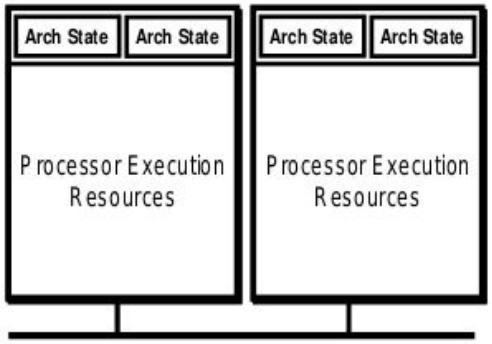


Figure 3: Processors with Hyper-Threading Technology

Esto representa un pequeña fracción del procesador en cuanto a circuitería (~5%).

Los objetivos (cumplidos) de este diseño fueron:

1. Minimiza el costo y tamaño, y mejora la performance. Ya que la mayor parte la microarquitectura se comparte y solo se duplica la parte arquitectural (+5% de tamaño).
2. Si un procesador lógico está en Stall (por ejemplo, por un cache miss), el otro puede continuar ejecutando y hay *fairness*. Esto se logra partiendo a la mitad las colas entre etapas de Pipeline.
3. Si solo se está ejecutando un thread solo, la performance es igual que si no hubieras dividido los recursos.

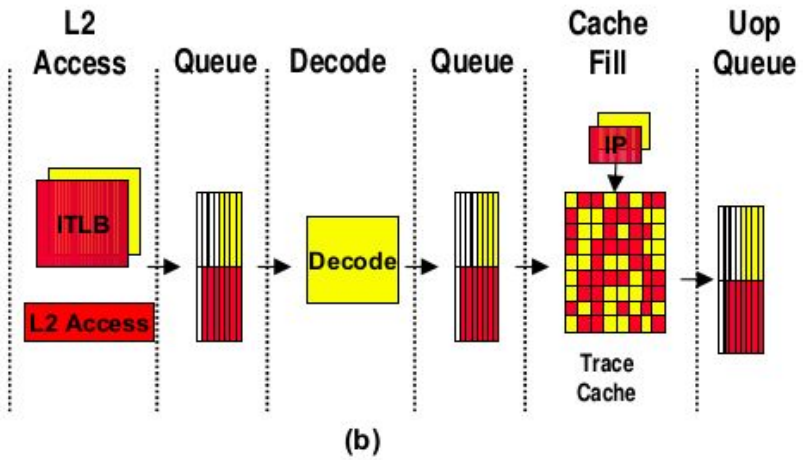


Figure 5: Front-end detailed pipeline (a) Trace Cache Hit (b) Trace Cache Miss

FrontEnd

Trace Cache: Dos IPs, y alterna ciclos para enviar microops. Si uno está Stall le da el 100% al otro mientras. La cantidad de entradas no está partida 50/50.

Microcode Rom: decodifica instrucciones complejas, y se arbitra de la misma manera que TC.

ITLB & Branch Prediction: El fetch se hace en orden de pedidos, pero mínimo se asegura un pedido en cola por thread. El buffer del decoder es dedicado por thread. Cada uno tiene su ITLB, Branch Prediction History Buffer, y Return Stack. El Global Branch History Array se comparte.

Decode: Cuando ambos threads decodifican en simultáneo, los buffers alternan. El decoder guarda dos copias de los estados necesarios para los dos threads aunque decodifica para uno solo por vez. El switch entre procesadores lógicos es tosco: de a uno por vez para facilitar complejidad. Por supuesto, si uno solo decodifica, se le dedica 100%.

Out of Order Execution Engine

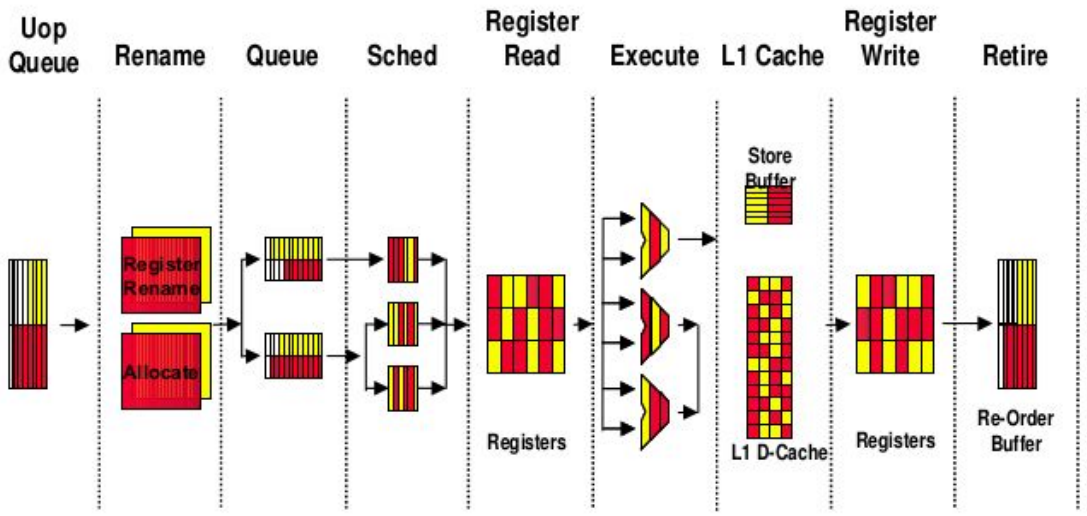


Figure 6: Out-of-order execution engine detailed pipeline

Uop Queue: esta cola desacopla el FrontEnd con el OoOE Engine. Está particionada de manera que cada procesador lógico tenga la mitad de las entradas.
Allocator: toma Uops y hace los alojos necesarios para ejecutarlas, incluyendo 126 entradas de ROB, 128 registros físicos int y 128 registros físicos FP, 48 entradas de load buffer y 24 store buffer. De los ROB y L/S Buffers, cada thread puede usar hasta la mitad. El Allocator también se alterna entre procesadores con cada ciclo de clock.
Register rename: Funciona paralelo al allocator, renombra los registros de la ISA a los 128 físicos con la RAT (Register Alias Table) del thread. Luego mandan las Uops a uno de buffers de; memoria o general, según corresponda (también están particionadas estas colas).
Scheduling/Dispatch: según la disponibilidad de sus inputs y unidades de ejecución, los schedulers mandan hasta 6 Uops por clock. Las colas de memoria y general envían alternadamente por thread, pero de la cola interna de cada scheduler se envían sin preferencia (aunque hay cantidad límite en cola por thread).
Unidades de ejecución/Registros: solo saben de renamings (tags), no saben nada de threads. Los resultados se ponen en el ROB, que está partido 50/50.
Retirement: el retirement se alterna, salvo que no haya nadie, y el Write Back a cache también.

Memory Subsytem

Memoria no tiene ni idea de threading. Stores/loads se levantan sin orden.
TLB de datos (DTLB): Es compartida (es grande) pero se utiliza un ID para saber de qué unidad es.
Caches: Compartidas. En un server, si los threads usan memoria en común, el rendimiento es alto.
Bus: No hay prioridad para procesadores lógicos particulares, pero hay distinción mediante MD, sobre todo porque las interrupciones son por thread (sino se interrumpiría a los dos siempre).

Single-Task - MultiTask modes

Hay dos modos, Single-Task o Multi-Task. Single task combina todo lo compartido para un solo thread. ST0 pasa cuando el procesador lógico 1 hace HALT y viceversa. Si se esté en ST y hay una interrupción en el que está detenido vuelve a MT. HALT solo lo puede hacer el SO.

Operating System

Si bien el sistema ve los procesadores lógicos como físicos hay dos optimizaciones que estaría bueno que haga:

1. Enviar HALT cuando un proceso termine, para evitar tener el thread en IDLE.
2. Hacer scheduling priorizando procesadores físicos para no compartir cuando no haga falta.

Para servers y online transactions la mejora es notable (porque es usar los mismos datos en paralelo), es de aproximadamente 30%. Agregando únicamente 5% de hardware y consumo.